

Data Warehouse Service
9.1.0

Syntax Reference

Issue	01
Date	2025-09-18



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 DWS SQL Overview.....	1
2 DWS SQL Syntax.....	3
3 Differences Between DWS and PostgreSQL.....	4
4 Keyword.....	8
5 Data Types.....	41
5.1 Numeric Types.....	41
5.2 Monetary Types.....	46
5.3 Boolean Type.....	47
5.4 Character Types.....	48
5.5 Binary Data Types.....	51
5.6 Date/Time Types.....	53
5.7 Geometric Types.....	61
5.8 Array.....	63
5.9 Enumeration Type.....	67
5.10 Network Address Types.....	68
5.11 Bit String Types.....	70
5.12 Text Search Types.....	71
5.13 UUID Type.....	74
5.14 JSON Types.....	74
5.15 RoaringBitmap.....	78
5.16 HLL Data Types.....	78
5.17 Object Identifier Types.....	81
5.18 Pseudo-Types.....	83
5.19 Range Types.....	85
5.20 Composite Types.....	89
5.21 Data Types Supported by Column-Store Tables.....	90
5.22 XML.....	92
6 Constant and Macro.....	94
7 Functions and Operators.....	95
7.1 Character Processing Functions and Operators.....	95
7.2 Binary String Functions and Operators.....	126

7.3 Bit String Functions and Operators.....	128
7.4 Numeric Functions and Operators.....	130
7.4.1 Numeric Operators.....	130
7.4.2 Numeric Functions.....	134
7.5 Date and Time Processing Functions and Operators.....	143
7.5.1 Date and Time Operators.....	143
7.5.2 Date and Time Functions.....	146
7.5.3 EXTRACT.....	162
7.5.4 date_part.....	167
7.5.5 date_format.....	167
7.5.6 time_format.....	169
7.6 SEQUENCE Functions.....	173
7.7 Array Functions and Operators.....	175
7.7.1 Array Operators.....	175
7.7.2 Array Functions.....	177
7.8 Logical Operators.....	181
7.9 Comparison Operators.....	182
7.10 Pattern Matching Operators.....	182
7.11 Aggregate Functions.....	187
7.12 Window Functions.....	202
7.13 Type Conversion Functions.....	208
7.14 JSON/JSONB Functions and Operators.....	219
7.14.1 JSON/JSONB Operators.....	219
7.14.2 JSON/JSONB Functions.....	223
7.15 Security Functions.....	243
7.16 Conditional Expression Functions.....	254
7.17 Range Functions and Operators.....	258
7.17.1 Range Operators.....	258
7.17.2 Range Functions.....	261
7.18 Data Redaction Functions.....	263
7.19 Bitmap Functions and Operators.....	265
7.19.1 Roaring Bitmap Operators.....	265
7.19.2 Bitmap Functions.....	268
7.19.3 Roaring Bitmap Aggregation Functions.....	275
7.19.4 Use Cases.....	277
7.20 UUID Functions.....	279
7.21 Text Search Functions and Operators.....	282
7.21.1 Text Search Operators.....	282
7.21.2 Text Search Functions.....	283
7.21.3 Text Search Debugging Functions.....	286
7.22 HLL Functions and Operators.....	289
7.22.1 HLL Operators.....	289

7.22.2 Hash Functions.....	290
7.22.3 Precision Functions.....	294
7.22.4 Aggregate Functions.....	296
7.22.5 Functional Functions.....	298
7.22.6 Built-in Functions.....	301
7.23 Set Returning Functions.....	302
7.23.1 Series Generating Functions.....	302
7.23.2 Subscript Generating Functions.....	303
7.24 Geometric Functions and Operators.....	304
7.24.1 Geometric Operators.....	304
7.24.2 Geometric Functions.....	310
7.24.3 Geometric Type Conversion Functions.....	313
7.25 Network Address Functions and Operators.....	317
7.25.1 cidr and inet Operators.....	318
7.25.2 Network Address Functions.....	321
7.26 System Information Functions.....	324
7.26.1 Session Information Functions.....	324
7.26.2 Access Privilege Inquiry Functions.....	329
7.26.3 Schema Visibility Inquiry Functions.....	333
7.26.4 System Catalog Information Functions.....	335
7.26.5 System Function Checking Function.....	341
7.26.6 Comment Checking Functions.....	342
7.26.7 Transaction IDs and Snapshots.....	343
7.26.8 Computing Node Group Function.....	344
7.26.9 Lock Information Function.....	345
7.27 System Management Functions.....	345
7.27.1 Configuration Settings Functions.....	345
7.27.2 Universal File Access Functions.....	346
7.27.3 Server Signaling Functions.....	348
7.27.4 Snapshot Synchronization Functions.....	352
7.27.5 Advisory Lock Functions.....	353
7.27.6 Replication Functions.....	356
7.27.7 Resource Management Functions.....	365
7.27.8 Memory Management Functions.....	379
7.27.9 Other Functions.....	381
7.28 Backup and Restoration Control Functions.....	407
7.28.1 Backup Control Functions.....	407
7.28.2 Restoration Control Functions.....	413
7.29 Database Object Functions.....	416
7.29.1 Database Object Size Functions.....	416
7.29.2 Database Object Position Functions.....	421
7.29.3 Partition Management Function.....	421

7.29.4 Collation Version Function.....	422
7.29.5 Hot and Cold Table Functions.....	423
7.30 Residual File Management Functions.....	425
7.30.1 Functions for Obtaining the Residual File List.....	425
7.30.2 Functions for Scanning Residual Files.....	427
7.30.3 Functions for Verifying Residual Files.....	434
7.30.4 Functions for Deleting Residual Files.....	437
7.30.5 Using Residual File Management Functions.....	439
7.31 Statistics Information Functions.....	442
7.32 Trigger Functions.....	484
7.33 XML Functions.....	485
7.33.1 Generating XML Content.....	485
7.33.2 XML Predicates.....	488
7.33.3 Processing XML.....	490
7.33.4 Mapping a Table to XML.....	493
7.34 Call Stack Recording Functions.....	495
7.35 Hudi System Functions.....	498
7.36 Funnel and Retention Functions.....	505
7.37 EXTERNAL SCHEMA System Functions.....	510
7.38 Storage-Compute Decoupling Functions.....	511
7.38.1 Data Sharing Functions.....	511
7.38.2 Function for Obtaining File Information of Tablespaces in the Trash Directory.....	512
8 Expressions.....	514
8.1 Simple Expressions.....	514
8.2 Conditional Expressions.....	515
8.3 Subquery Expressions.....	520
8.4 Array Expressions.....	523
8.5 Row Expressions.....	525
9 Type Conversion.....	526
9.1 Overview.....	526
9.2 Operators.....	528
9.3 Functions.....	530
9.4 Value Storage.....	532
9.5 UNION, CASE, and Related Constructs.....	533
10 Full Text Search.....	537
10.1 Introduction.....	537
10.1.1 Full-Text Retrieval.....	537
10.1.2 What Is a Document?.....	538
10.1.3 Basic Text Matching.....	539
10.1.4 Configurations.....	540
10.2 Tables and Indexes.....	540

10.2.1 Searching a Table.....	540
10.2.2 Creating an Index.....	542
10.2.3 Constraints on Index Use.....	543
10.3 Controlling Text Search.....	544
10.3.1 Parsing Documents.....	544
10.3.2 Parsing Queries.....	545
10.3.3 Ranking Search Results.....	546
10.3.4 Highlighting Results.....	548
10.4 Additional Features.....	549
10.4.1 Manipulating tsvector.....	550
10.4.2 Manipulating Queries.....	550
10.4.3 Rewriting Queries.....	551
10.4.4 Gathering Document Statistics.....	552
10.5 Parsers	553
10.6 Dictionaries.....	557
10.6.1 Overview.....	557
10.6.2 Stop Words.....	558
10.6.3 Simple Dictionary.....	559
10.6.4 Synonym Dictionary.....	560
10.6.5 Thesaurus Dictionary.....	561
10.6.6 Ispell Dictionary.....	563
10.6.7 Snowball Dictionary.....	564
10.7 Configuration Examples.....	564
10.8 Testing and Debugging Text Search.....	566
10.8.1 Testing a Configuration.....	566
10.8.2 Testing a Parser.....	567
10.8.3 Testing a Dictionary.....	568
10.9 Limitations.....	568
11 System Operation.....	569
12 Transaction Management.....	570
13 DDL Syntax.....	577
13.1 DDL Syntax Overview.....	577
13.2 ALTER BLOCK RULE.....	584
13.3 ALTER DATABASE.....	586
13.4 ALTER EXCEPT RULE.....	589
13.5 ALTER EXTERNAL SCHEMA.....	590
13.6 ALTER FOREIGN TABLE (GDS Import and Export).....	592
13.7 ALTER FOREIGN TABLE (for HDFS or OBS).....	594
13.8 ALTER FUNCTION.....	596
13.9 ALTER GROUP.....	600
13.10 ALTER INDEX.....	601

13.11 ALTER LARGE OBJECT.....	608
13.12 ALTER MATERIALIZED VIEW.....	609
13.13 ALTER PUBLICATION.....	612
13.14 ALTER REDACTION POLICY.....	614
13.15 ALTER RESOURCE POOL.....	617
13.16 ALTER ROLE.....	619
13.17 ALTER SUBSCRIPTION.....	628
13.18 ALTER ROW LEVEL SECURITY POLICY.....	630
13.19 ALTER SCHEMA.....	631
13.20 ALTER SEQUENCE.....	632
13.21 ALTER SERVER.....	634
13.22 ALTER SESSION.....	639
13.23 ALTER SYNONYM.....	642
13.24 ALTER SYSTEM KILL SESSION.....	643
13.25 ALTER TABLE.....	644
13.26 ALTER TABLE PARTITION.....	665
13.27 ALTER TEXT SEARCH CONFIGURATION.....	680
13.28 ALTER TEXT SEARCH DICTIONARY.....	683
13.29 ALTER TRIGGER.....	685
13.30 ALTER TYPE.....	687
13.31 ALTER USER.....	690
13.32 ALTER VIEW.....	698
13.33 CLEAN CONNECTION.....	701
13.34 CLOSE.....	703
13.35 CLUSTER.....	704
13.36 COMMENT.....	706
13.37 CREATE BARRIER.....	709
13.38 CREATE BLOCK RULE.....	709
13.39 CREATE DATABASE.....	711
13.40 CREATE EXCEPT RULE.....	716
13.41 CREATE EXTERNAL SCHEMA.....	718
13.42 CREATE FOREIGN TABLE (for GDS Import and Export).....	721
13.43 CREATE FOREIGN TABLE (SQL on OBS or Hadoop).....	745
13.44 CREATE FOREIGN TABLE (for OBS Import and Export).....	780
13.45 CREATE FUNCTION.....	801
13.46 CREATE GROUP.....	820
13.47 CREATE INDEX.....	821
13.48 CREATE MATERIALIZED VIEW.....	836
13.49 CREATE PROCEDURE.....	846
13.50 CREATE PUBLICATION.....	854
13.51 CREATE REDACTION POLICY.....	856
13.52 CREATE ROW LEVEL SECURITY POLICY.....	861

13.53 CREATE RESOURCE POOL.....	867
13.54 CREATE ROLE.....	872
13.55 CREATE SCHEMA.....	882
13.56 CREATE SEQUENCE.....	884
13.57 CREATE SERVER.....	889
13.58 CREATE SUBSCRIPTION.....	896
13.59 CREATE SYNONYM.....	900
13.60 CREATE TABLE.....	902
13.61 CREATE TABLE AS.....	945
13.62 CREATE TABLE PARTITION.....	953
13.63 CREATE TEXT SEARCH CONFIGURATION.....	990
13.64 CREATE TEXT SEARCH DICTIONARY.....	993
13.65 CREATE TRIGGER.....	1000
13.66 CREATE TYPE.....	1006
13.67 CREATE USER.....	1016
13.68 CREATE VIEW.....	1024
13.69 CURSOR.....	1027
13.70 DISCARD.....	1029
13.71 DROP BLOCK RULE.....	1032
13.72 DROP DATABASE.....	1032
13.73 DROP EXCEPT RULE.....	1034
13.74 DROP FOREIGN TABLE.....	1035
13.75 DROP FUNCTION.....	1036
13.76 DROP GROUP.....	1037
13.77 DROP INDEX.....	1038
13.78 DROP MATERIALIZED VIEW.....	1039
13.79 DROP OWNED.....	1040
13.80 DROP PUBLICATION.....	1041
13.81 DROP REDACTION POLICY.....	1042
13.82 DROP ROW LEVEL SECURITY POLICY.....	1043
13.83 DROP PROCEDURE.....	1044
13.84 DROP RESOURCE POOL.....	1044
13.85 DROP ROLE.....	1045
13.86 DROP SCHEMA.....	1046
13.87 DROP SEQUENCE.....	1047
13.88 DROP SERVER.....	1048
13.89 DROP SUBSCRIPTION.....	1049
13.90 DROP SYNONYM.....	1050
13.91 DROP TABLE.....	1051
13.92 DROP TEXT SEARCH CONFIGURATION.....	1053
13.93 DROP TEXT SEARCH DICTIONARY.....	1054
13.94 DROP TRIGGER.....	1056

13.95 DROP TYPE.....	1057
13.96 DROP USER.....	1057
13.97 DROP VIEW.....	1059
13.98 FETCH.....	1060
13.99 MOVE.....	1064
13.100 REFRESH MATERIALIZED VIEW.....	1065
13.101 REINDEX.....	1067
13.102 RENAME TABLE.....	1070
13.103 RESET.....	1071
13.104 SET.....	1072
13.105 SET CONSTRAINTS.....	1074
13.106 SET ROLE.....	1076
13.107 SET SESSION AUTHORIZATION.....	1077
13.108 SHOW.....	1078
13.109 TRUNCATE.....	1079
13.110 VACUUM.....	1082
14 DML Syntax.....	1092
14.1 DML Syntax Overview.....	1092
14.2 CALL.....	1093
14.3 COPY.....	1095
14.4 DELETE.....	1117
14.5 EXPLAIN.....	1120
14.6 EXPLAIN PLAN.....	1128
14.7 LOCK.....	1130
14.8 MERGE INTO.....	1135
14.9 INSERT and UPSERT.....	1140
14.9.1 INSERT.....	1140
14.9.2 UPSERT.....	1151
14.10 UPDATE.....	1157
14.11 VALUES.....	1166
15 DCL Syntax.....	1169
15.1 DCL Syntax Overview.....	1169
15.2 ALTER DEFAULT PRIVILEGES.....	1169
15.3 ANALYZE ANALYSE.....	1172
15.4 DEALLOCATE.....	1178
15.5 DO.....	1179
15.6 EXECUTE.....	1180
15.7 EXECUTE DIRECT.....	1181
15.8 GRANT.....	1182
15.9 PREPARE.....	1191
15.10 REASSIGN OWNED.....	1192
15.11 REVOKE.....	1193

16 DQL Syntax.....	1197
16.1 DQL Syntax Overview.....	1197
16.2 SELECT.....	1197
16.3 SELECT INTO.....	1226
17 TCL Syntax.....	1228
17.1 TCL Syntax Overview.....	1228
17.2 ABORT.....	1228
17.3 BEGIN.....	1229
17.4 CHECKPOINT.....	1230
17.5 COMMIT END.....	1231
17.6 COMMIT PREPARED.....	1232
17.7 PREPARE TRANSACTION.....	1232
17.8 SAVEPOINT.....	1233
17.9 SET TRANSACTION.....	1235
17.10 START TRANSACTION.....	1236
17.11 ROLLBACK.....	1238
17.12 RELEASE SAVEPOINT.....	1238
17.13 ROLLBACK PREPARED.....	1239
17.14 ROLLBACK TO SAVEPOINT.....	1240

1 DWS SQL Overview

What Is SQL?

SQL is a standard computer language used to control the access to databases and manage data in databases.

SQL provides different statements to enable you to:

- Query data.
- Insert, update, and delete rows.
- Create, replace, modify, and delete objects.
- Control the access to a database and its objects.
- Maintain the consistency and integrity of a database.

SQL consists of commands and functions that are used to manage databases and database objects. SQL can also forcibly implement the rules for data types, expressions, and texts. Therefore, section "SQL Reference" describes data types, expressions, functions, and operators in addition to SQL syntax.

Development of SQL Standards

Released SQL standards are as follows:

- 1986: ANSI X3.135-1986, ISO/IEC 9075:1986, SQL-86
- 1989: ANSI X3.135-1989, ISO/IEC 9075:1989, SQL-89
- 1992: ANSI X3.135-1992, ISO/IEC 9075:1992, SQL-92 (SQL2)
- 1999: ISO/IEC 9075:1999, SQL:1999 (SQL3)
- 2003: ISO/IEC 9075:2003, SQL:2003 (SQL4)
- 2011: ISO/IEC 9075:200N, SQL:2011 (SQL5)

Supported SQL Standards

DWS is compatible with Postgres-XC features and supports the major features of SQL2, SQL3, and SQL4 by default, and some features of SQL5.

Languages Supported by DWS

DWS supports PL/pgSQL and PL/Java.

SQL Example Description

SQL examples in this manual are developed based on the TPC-DS model. Before you execute the examples, install the TPC-DS benchmark by following the instructions on the official website <https://www.tpc.org/tpcds/>.

2 DWS SQL Syntax

To better understand how to use the syntax, you can refer to the following description of SQL syntax text conventions.

Table 2-1 SQL Syntax Description

Format	Description
Uppercase characters	Keywords must be in uppercase.
Lowercase characters	Parameters must be in lowercase.
[]	Optional syntax item. Items in brackets [] are optional.
{ }	Mandatory syntax item.
...	Preceding elements can appear repeatedly.
[x y ...]	One item is selected from two or more options or no item is selected.
{ x y ... }	One item is selected from two or more options.
[x y ...] [...]	You can choose either multiple parameters or no parameters. If you choose multiple parameters, simply separate them with spaces.
[x y ...] [,...]	You can choose either multiple parameters or no parameters. If you choose multiple parameters, simply separate them with commas (,).
{ x y ... } [...]	You must select at least one parameter. If you select multiple parameters, separate them with spaces.
{ x y ... } [,...]	You must select at least one parameter. If you select multiple parameters, separate them with commas (,).

3 Differences Between DWS and PostgreSQL

The differences between DWS and PostgreSQL are sorted based on PostgreSQL 9.X. The differences are as follows:

Client Differences

DWS **gsql** differs from PostgreSQL **psql** in that the former has made the following changes to enhance security:

- User passwords cannot be set by running the **\password** meta-command.
- The **\i+**, **\ir+**, and **\include_relative+** meta-commands and the input and output parameter **-k** are added to encrypt imported and exported files.
- Historical command lines cannot be printed to files using the **\s** meta-command.
- SQL statements related to sensitive operations, such as those containing passwords, are not recorded. Users cannot see such records when they turn pages or press up or down arrow keys to view the SQL history.
- After a connection is set up, a message is displayed to inform users of password expiration and to show version information.

gsql provides the following additional functions based on **psql**:

- The output format parameter **-r** is added to allow you to adjust the focus by pressing the **Tab** key or arrow keys when entering commands.
- The **\parallel** meta-command is added to improve execution performance.
- The **\set RETRY** meta-command is added to support retry upon statement errors.
- Slashes (/) are used as the default terminator at the end of PL/SQL statements **CREATE OR REPLACE FUNCTION/PROCEDURE**.

libpq:

During the development of certain DWS functions such as the **gsql** client connection tool, PostgreSQL **libpq** is greatly modified. However, the **libpq** interface is not verified in application development. Therefore, underlying risks probably exist. You can use the ODBC or JDBC APIs instead.

SQL Statement Differences

Table 3-1 PostgreSQL syntaxes not supported by DWS

Type	Syntaxes Not Supported by DWS	Description
Data	Geometric lines	For details about the data types supported by DWS, see Data Types .
	pg_node_tree	
Functions	Enumeration functions: - enum_first(anyenum) - enum_last(anyenum) - enum_range(anyenum) - enum_range(anyenum, anyenum)	For details about the functions supported by DWS, see Functions and Operators .
	Access permission query functions: - has_sequence_privilege(user, sequence, privilege) - has_sequence_privilege(sequence, privilege)	
	System directory information functions: - pg_get_triggerdef(trigger_oid) - pg_get_triggerdef(trigger_oid, pretty_bool)	
	Geometric type conversion function: line(point, point)	
	pg_node_tree	
SQL syntax	CREATE TABLE clause: INHERITS (parent_table [, ...])	Inherits tables.
	CREATE TABLE column constraints: REFERENCES reftable [(refcolumn)] [MATCH FULL MATCH PARTIAL MATCH SIMPLE] [ON DELETE action] [ON UPDATE action]	Use REFERENCES reftable [(refcolumn)] [MATCH FULL MATCH PARTIAL MATCH SIMPLE] [ON DELETE action] [ON UPDATE action] to create a foreign key constraint for a table.
	CREATE TABLE table constraints: EXCLUDE [USING index_method] (exclude_element WITH operator [, ...])	Use EXCLUDE [USING index_method] (exclude_element WITH operator [, ...]) to create exclusion constraints for a table.

Type	Syntaxes Not Supported by DWS	Description
	CREATE/ALTER/DROP EXTENSION	Loads, modifies, and deletes extensions.
	CREATE/ALTER/DROP AGGREGATE	Defines, modifies, and deletes aggregate functions.
	CREATE/ALTER/DROP OPERATOR	Creates, modifies, and deletes operators.
	CREATE/ALTER/DROP OPERATOR CLASS	Creates, modifies, and deletes operator classes.
	CREATE/ALTER/DROP OPERATOR FAMILY	Creates, modifies, and deletes operator families.
	CREATE/ALTER/DROP TEXT SEARCH PARSER	Creates, modifies, and deletes text search parsers.
	CREATE/ALTER/DROP TEXT SEARCH TEMPLATE	Create, modify, and delete text search templates.
	CREATE/ALTER/DROP COLLATION	Creates, modifies, and deletes collation rules.
	CREATE/ALTER/DROP CONVERSION	Defines, modifies, and deletes the conversion of character set encoding.
	CREATE/ALTER/DROP RULE	Creates, modifies, and deletes rules.
	CREATE/ALTER/DROP LANGUAGE	Registers, modifies, and deletes procedural languages (LANGUAGE).
	CREATE/ALTER/DROP DOMAIN	Creates, modifies, and deletes domains.
	CREATE/DROP CAST	Defines and deletes casts.
	CREATE/ALTER/DROP USER MAPPING	Defines, modifies, and deletes user mapping.
	SECURITY LABEL	Defines or changes the security tag of an object.
	NOTIFY	Generates a notification.
	LISTEN	Listens to a notification.
	UNLISTEN	Stops listening to a notification.

Type	Syntaxes Not Supported by DWS	Description
	LOAD	Loads or reloads a shared library file.
	DISCARD	Releases session resources of a database. (Clusters of 8.2.0 and later versions support DISCARD .)
	MOVE BACKWARD	Moves a cursor backward.
	COPY FROM FILE and COPY TO FILE	To isolate permissions, COPY FROM FILE and COPY TO FILE are disabled in DWS.
Other	User-defined C functions	For details about user-defined functions supported by DWS, see "DWS User-Defined Functions" .

4 Keyword

The SQL contains reserved and non-reserved words. Reserved keywords must not be used as other identifiers in accordance with standards. Non-reserved keywords have special meanings only in specific environments and can be used as identifiers in other environments. Even so, do not use non-reserved keywords.

You can use **SELECT * FROM pg_get_keywords();** to query the keyword of DWS.

Example: **AUTHORIZATION** is a reserved keyword. Using it as a table name causes an error and prevents the table from being created.

However, you can use **AUTHORIZATION** as a function or type name under certain conditions (see the table below).

Figure 4-1 Using a keyword as a table name

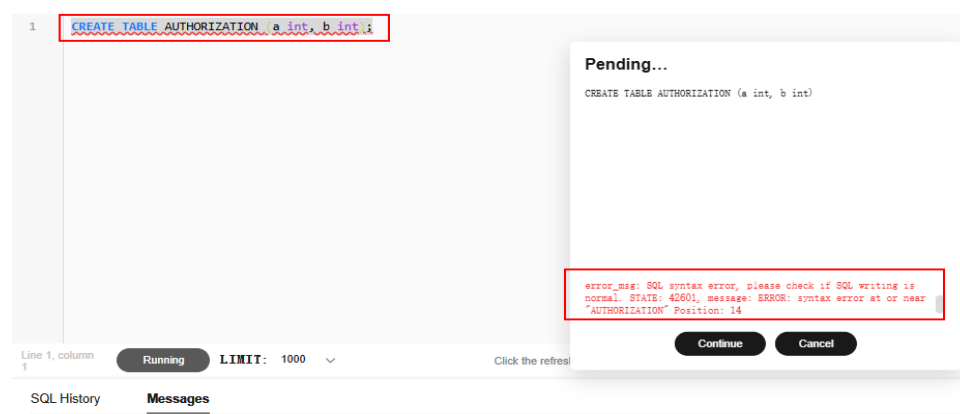
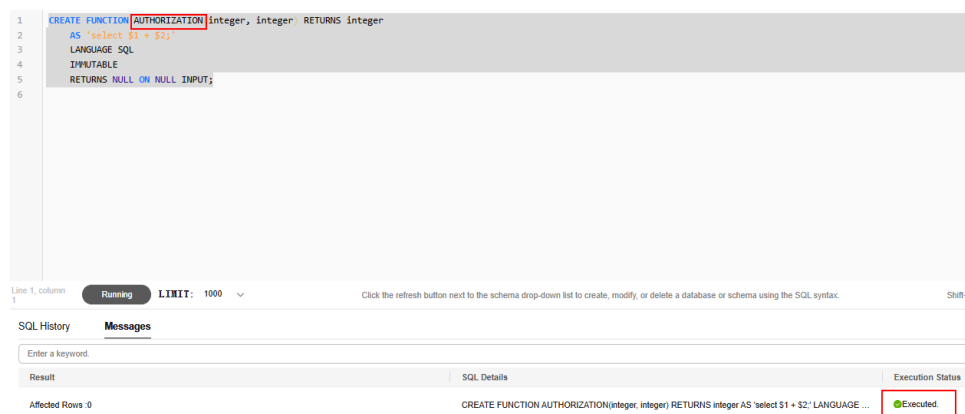


Figure 4-2 Using a keyword as a function name

Identifier Naming Conventions

A database identifier is a name that uniquely identifies an object (such as a table, column, or index) in a database. In a database, identifiers are often used to reference database objects in SQL query statements to ensure the accuracy and consistency of database operations. It is important to name and use identifiers properly. The naming of DWS identifiers must comply with the following rules:

- The identifier must consist of letters (a-z), underscores (_), digits (0-9), and dollar signs (\$).
- An identifier name should start with a letter (a-z) or an underscore (_).
- The identifier can contain a maximum of 63 characters.
- Avoid using SQL reserved words as identifiers. Be careful with non-reserved keywords too.
- If the database uses GBK, UTF8, or SQL_ASCII, identifiers can include Chinese characters. In UTF8 or SQL_ASCII, each Chinese character takes up three bytes, with a maximum of 21 characters. In GBK, each Chinese character takes up two bytes, with a maximum of 31 characters. The **Latin1** character set does not support Chinese characters. The character set is set when creating the database. For details, see [CREATE DATABASE](#).

Table 4-1 SQL keyword list

Keyword	DWS	SQL:1999	SQL:1992
ABORT	Non-reserved	-	-
ABS	-	Non-reserved	-
ABSOLUTE	Non-reserved	Reserved	Reserved
ACCESS	Non-reserved	-	-
ACCOUNT	Non-reserved	-	-
ACTION	Non-reserved	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
ADA	-	Non-reserved	Non-reserved
ADD	Non-reserved	Reserved	Reserved
ADMIN	Non-reserved	Reserved	-
AFTER	Non-reserved	Reserved	-
AGGREGATE	Non-reserved	Reserved	-
ALIAS	-	Reserved	-
ALL	Reserved	Reserved	Reserved
ALLOCATE	-	Reserved	Reserved
ALSO	Non-reserved	-	-
ALTER	Non-reserved	Reserved	Reserved
ALWAYS	Non-reserved	-	-
ANALYSE	Reserved	-	-
ANALYZE	Reserved	-	-
AND	Reserved	Reserved	Reserved
ANY	Reserved	Reserved	Reserved
APP	Non-reserved	-	-
ARE	-	Reserved	Reserved
ARRAY	Reserved	Reserved	-
AS	Reserved	Reserved	Reserved
ASC	Reserved	Reserved	Reserved
ASENSITIVE	-	Non-reserved	-
ASSERTION	Non-reserved	Reserved	Reserved
ASSIGNMENT	Non-reserved	Non-reserved	-
ASYMMETRIC	Reserved	Non-reserved	-
AT	Non-reserved	Reserved	Reserved
ATOMIC	-	Non-reserved	-
ATTRIBUTE	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
AUTHID	Reserved	-	-
AUTHINFO	Non-reserved	-	-
AUTHORIZATION	Reserved (functions and types allowed)	Reserved	Reserved
AUTO	Non-reserved	-	-
AUTOEXTEND	Non-reserved	-	-
AUTOMAPPED	Non-reserved	-	-
AUTOREFRESH	Non-reserved	-	-
AUTOSCALE	Non-reserved	-	-
AVG	-	Non-reserved	Reserved
BACKWARD	Non-reserved	-	-
BARRIER	Non-reserved	-	-
BEFORE	Non-reserved	Reserved	-
BEGIN	Non-reserved	Reserved	Reserved
BETWEEN	Non-reserved (excluding functions and types)	Non-reserved	Reserved
BIGINT	Non-reserved (excluding functions and types)	-	-
BINARY	Reserved (functions and types allowed)	Reserved	-
BINARY_DOUBLE	Non-reserved (excluding functions and types)	-	-
BINARY_INTEGER	Non-reserved (excluding functions and types)	-	-
BIT	Non-reserved (excluding functions and types)	Reserved	Reserved
BITVAR	-	Non-reserved	-
BIT_LENGTH	-	Non-reserved	Reserved
BLOB	Non-reserved	Reserved	-
BLOCK	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
BOOLEAN	Non-reserved (excluding functions and types)	Reserved	-
BOTH	Reserved	Reserved	Reserved
BUCKETS	Reserved	-	-
BUILD	Non-reserved	-	-
BREADTH	-	Reserved	-
BY	Non-reserved	Reserved	Reserved
C	-	Non-reserved	Non-reserved
CACHE	Non-reserved	-	-
CALL	Non-reserved	Reserved	-
CALLED	Non-reserved	Non-reserved	-
CARDINALITY	-	Non-reserved	-
CASCADE	Non-reserved	Reserved	Reserved
CASCADEDED	Non-reserved	Reserved	Reserved
CASE	Reserved	Reserved	Reserved
CAST	Reserved	Reserved	Reserved
CATALOG	Non-reserved	Reserved	Reserved
CATALOG_NAME	-	Non-reserved	Non-reserved
CHAIN	Non-reserved	Non-reserved	-
CHANGE	Non-reserved	-	-
CHAR	Non-reserved (excluding functions and types)	Reserved	Reserved
CHARACTER	Non-reserved (excluding functions and types)	Reserved	Reserved
CHARACTERISTICS	Non-reserved	-	-
CHARACTER_LENGTH	-	Non-reserved	Reserved
CHARACTER_SET_CATALOG	-	Non-reserved	Non-reserved

Keyword	DWS	SQL:1999	SQL:1992
CHARACTER_SET_NAME	-	Non-reserved	Non-reserved
CHARACTER_SET_SCHEMA	-	Non-reserved	Non-reserved
CHAR_LENGTH	-	Non-reserved	Reserved
CHECK	Reserved	Reserved	Reserved
CHECKED	-	Non-reserved	-
CHECKPOINT	Non-reserved	-	-
CLASS	Non-reserved	Reserved	-
CLEAN	Non-reserved	-	-
CLASS_ORIGIN	-	Non-reserved	Non-reserved
CLOB	Non-reserved	Reserved	-
CLOSE	Non-reserved	Reserved	Reserved
CLUSTER	Non-reserved	-	-
COALESCE	Non-reserved (excluding functions and types)	Non-reserved	Reserved
COBOL	-	Non-reserved	Non-reserved
COLLATE	Reserved	Reserved	Reserved
COLLATION	Reserved (functions and types allowed)	Reserved	Reserved
COLLATION_CATALOG	-	Non-reserved	Non-reserved
COLLATION_NAME	-	Non-reserved	Non-reserved
COLLATION_SCHEMA	-	Non-reserved	Non-reserved
COLUMN	Reserved	Reserved	Reserved
COLUMNS	Non-reserved	-	-
COLUMN_NAME	-	Non-reserved	Non-reserved

Keyword	DWS	SQL:1999	SQL:1992
COMMAND_FUNCTION	-	Non-reserved	Non-reserved
COMMAND_FUNCTION_CODE	-	Non-reserved	-
COMMENT	Non-reserved	-	-
COMMENTS	Non-reserved	-	-
COMMIT	Non-reserved	Reserved	Reserved
COMMITTED	Non-reserved	Non-reserved	Non-reserved
COMPACT	Reserved (functions and types allowed)	-	-
COMPATIBLE_ILLEGAL_CHARS	Non-reserved	-	-
COMPLETE	Non-reserved	-	-
COMPRESS	Non-reserved	-	-
COMPLETION	-	Reserved	-
CONCURRENTLY	Reserved (functions and types allowed)	-	-
CONDITION	-	-	-
CONDITION_NUMBER	-	Non-reserved	Non-reserved
CONFIGURATION	Non-reserved	-	-
CONFLICT	Non-reserved	-	-
CONNECT	-	Reserved	Reserved
CONNECTION	Non-reserved	Reserved	Reserved
CONNECTION_NAME	-	Non-reserved	Non-reserved
CONSTRAINT	Reserved	Reserved	Reserved
CONSTRAINTS	Non-reserved	Reserved	Reserved
CONSTRAINT_CATALOG	-	Non-reserved	Non-reserved
CONSTRAINT_NAME	-	Non-reserved	Non-reserved

Keyword	DWS	SQL:1999	SQL:1992
CONSTRAINT_SCHEMA	-	Non-reserved	Non-reserved
CONSTRUCTOR	-	Reserved	-
CONTAINS	-	Non-reserved	-
CONTENT	Non-reserved	-	-
CONTINUE	Non-reserved	Reserved	Reserved
CONVERSION	Non-reserved	-	-
CONVERT	-	Non-reserved	Reserved
COORDINATOR	Non-reserved	-	-
COPY	Non-reserved	-	-
CORRESPONDING	-	Reserved	Reserved
COST	Non-reserved	-	-
COUNT	-	Non-reserved	Reserved
CREATE	Reserved	Reserved	Reserved
CROSS	Reserved (functions and types allowed)	Reserved	Reserved
CSV	Non-reserved	-	-
CUBE	Non-reserved	Reserved	-
CURRENT	Non-reserved	Reserved	Reserved
CURRENT_CATALOG	Reserved	-	-
CURRENT_DATE	Reserved	Reserved	Reserved
CURRENT_PATH	-	Reserved	-
CURRENT_ROLE	Reserved	Reserved	-
CURRENT_SCHEMA	Reserved (functions and types allowed)	-	-
CURRENT_TIME	Reserved	Reserved	Reserved
CURRENT_TIMESTAMP	Reserved	Reserved	Reserved
CURRENT_USER	Reserved	Reserved	Reserved
CURSOR	Non-reserved	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
CURSOR_NAME	-	Non-reserved	Non-reserved
CYCLE	Non-reserved	Reserved	-
DATA	Non-reserved	Reserved	Non-reserved
DATE_FORMAT	Non-reserved	-	-
DATABASE	Non-reserved	-	-
DATAFILE	Non-reserved	-	-
DATETIME_INTERVAL_CODE	-	Non-reserved	Non-reserved
DATETIME_INTERVAL_PRECISION	-	Non-reserved	Non-reserved
DAY	Non-reserved	Reserved	Reserved
DBCOMPATIBILITY	Non-reserved	-	-
DBCOMPATIBILITY_BEHAVIOR	Non-reserved	-	-
DEALLOCATE	Non-reserved	Reserved	Reserved
DEC	Non-reserved (excluding functions and types)	Reserved	Reserved
DECIMAL	Non-reserved (excluding functions and types)	Reserved	Reserved
DECLARE	Non-reserved	Reserved	Reserved
DECODE	Non-reserved (excluding functions and types)	-	-
DEFAULT	Reserved	Reserved	Reserved
DEFAULTS	Non-reserved	-	-
DEFERRABLE	Reserved	Reserved	Reserved
DEFERRED	Non-reserved	Reserved	Reserved
DEFINED	-	Non-reserved	-
DEFINER	Non-reserved	Non-reserved	-
DELETE	Non-reserved	Reserved	Reserved
DELIMITER	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
DELIMITERS	Non-reserved	-	-
DELTA	Non-reserved	-	-
DELTAMERGE	Reserved (functions and types allowed)	-	-
DEMAND	Non-reserved	-	-
DEPTH	-	Reserved	-
DEREF	-	Reserved	-
DESC	Reserved	Reserved	Reserved
DESCRIBE	-	Reserved	Reserved
DESCRIPTOR	-	Reserved	Reserved
DESTROY	-	Reserved	-
DESTRUCTOR	-	Reserved	-
DETERMINISTIC	Non-reserved	Reserved	-
DIAGNOSTICS	-	Reserved	Reserved
DICTIONARY	Non-reserved	Reserved	-
DIRECT	Non-reserved	-	-
DIRECTORY	Non-reserved	-	-
DISABLE	Non-reserved	-	-
DISCARD	Non-reserved	-	-
DISCONNECT	-	Reserved	Reserved
DISPATCH	-	Non-reserved	-
DISTINCT	Reserved	Reserved	Reserved
DISTRIBUTE	Non-reserved	-	-
DISTRIBUTION	Non-reserved	-	-
DO	Reserved	-	-
DOCUMENT	Non-reserved	-	-
DOMAIN	Non-reserved	Reserved	Reserved
DOUBLE	Non-reserved	Reserved	Reserved
DROP	Non-reserved	Reserved	Reserved
DROPCOLUMNS	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
DUPLICATE	Non-reserved	-	-
DYNAMIC	-	Reserved	-
DYNAMIC_FUNCTION	-	Non-reserved	Non-reserved
DYNAMIC_FUNCTION_CODE	-	Non-reserved	-
EACH	Non-reserved	Reserved	-
ELASTIC	Non-reserved	-	-
ELSE	Reserved	Reserved	Reserved
ENABLE	Non-reserved	-	-
ENCODING	Non-reserved	-	-
ENCRYPTED	Non-reserved	-	-
END	Reserved	Reserved	Reserved
END-EXEC	-	Reserved	Reserved
ENFORCED	Non-reserved	-	-
ENUM	Non-reserved	-	-
EOL	Non-reserved	-	-
EQUALS	-	Reserved	-
ERRORS	Non-reserved	-	-
ESCAPE	Non-reserved	Reserved	Reserved
ESCAPING	Non-reserved	-	-
EVERY	Non-reserved	Reserved	-
EXCEPT	Reserved	Reserved	Reserved
EXCEPTION	-	Reserved	Reserved
EXCHANGE	Non-reserved	-	-
EXCLUDE	Non-reserved	-	-
EXCLUDING	Non-reserved	-	-
EXCLUSIVE	Non-reserved	-	-
EXEC	-	Reserved	Reserved
EXECUTE	Non-reserved	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
EXISTING	-	Non-reserved	-
EXISTS	Non-reserved (excluding functions and types)	Non-reserved	Reserved
EXPIRATION	Non-reserved	-	-
EXPLAIN	Non-reserved	-	-
EXTENSION	Non-reserved	-	-
EXTERNAL	Non-reserved	Reserved	Reserved
EXTRACT	Non-reserved (excluding functions and types)	Non-reserved	Reserved
FALSE	Reserved	Reserved	Reserved
FAMILY	Non-reserved	-	-
FAST	Non-reserved	-	-
FENCED	Reserved	-	-
FETCH	Reserved	Reserved	Reserved
FILEHEADER	Non-reserved	-	-
FILL_AVG	Non-reserved (excluding functions and types)	-	-
FILL_FIRST	Non-reserved (excluding functions and types)	-	-
FILL_LAST	Non-reserved (excluding functions and types)	-	-
FILL_MISSING_FIELDS	Non-reserved	-	-
FILTER	Non-reserved	-	-
FINAL	-	Non-reserved	-
FIRST	Non-reserved	Reserved	Reserved
FIXED	Non-reserved	Reserved	Reserved
FLOAT	Non-reserved (excluding functions and types)	Reserved	Reserved
FOLLOWING	Non-reserved	-	-
FOR	Reserved	Reserved	Reserved
FORCE	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
FOREIGN	Reserved	Reserved	Reserved
FORMATTER	Non-reserved	-	-
FORTRAN	-	Non-reserved	Non-reserved
FORWARD	Non-reserved	-	-
FOUND	-	Reserved	Reserved
FREE	-	Reserved	-
FREEZE	Reserved (functions and types allowed)	-	-
FROM	Reserved	Reserved	Reserved
FULL	Reserved (functions and types allowed)	Reserved	Reserved
FUNCTION	Non-reserved	Reserved	-
FUNCTIONS	Non-reserved	-	-
G	-	Non-reserved	-
GENERAL	-	Reserved	-
GENERATED	-	Non-reserved	-
GET	-	Reserved	Reserved
GLOBAL	Non-reserved	Reserved	Reserved
GO	-	Reserved	Reserved
GOTO	-	Reserved	Reserved
GRANT	Reserved	Reserved	Reserved
GRANTED	Non-reserved	Non-reserved	-
GREATEST	Non-reserved (excluding functions and types)	-	-
GROUP	Reserved	Reserved	Reserved
GROUPING	Non-reserved (excluding functions and types)	Reserved	-
HANDLER	Non-reserved	-	-
HAVING	Reserved	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
HDFS DIRECTORY	Reserved (functions and types allowed)	-	-
HEADER	Non-reserved	-	-
HIERARCHY	-	Non-reserved	-
HOLD	Non-reserved	Non-reserved	-
HOST	-	Reserved	-
HOT	Reserved	-	-
HOURL	Non-reserved	Reserved	Reserved
IDENTIFIED	Non-reserved	-	-
IDENTITY	Non-reserved	Reserved	Reserved
IF	Non-reserved (excluding functions and types)	-	-
IFNULL	Non-reserved (excluding functions and types)	-	-
IGNORE	Non-reserved	Reserved	-
IGNORE_EXTRA_DATA	Non-reserved	-	-
ILIKE	Reserved (functions and types allowed)	-	-
IMMEDIATE	Non-reserved	Reserved	Reserved
IMMUTABLE	Non-reserved	-	-
IMPLEMENTATION	-	Non-reserved	-
IMPLICIT	Non-reserved	-	-
IN	Reserved	Reserved	Reserved
INCLUDING	Non-reserved	-	-
INCREMENT	Non-reserved	-	-
INDEX	Non-reserved	-	-
INDEXES	Non-reserved	-	-
INDICATOR	-	Reserved	Reserved
INFIX	-	Non-reserved	-

Keyword	DWS	SQL:1999	SQL:1992
INHERIT	Non-reserved	-	-
INHERITS	Non-reserved	-	-
INITIAL	Non-reserved	-	-
INITIALIZE	-	Reserved	-
INITIALLY	Reserved	Reserved	Reserved
INITTRANS	Non-reserved	-	-
INLINE	Non-reserved	-	-
INNER	Reserved (functions and types allowed)	Reserved	Reserved
INOUT	Non-reserved (excluding functions and types)	Reserved	-
INPUT	Non-reserved	Reserved	Reserved
INSENSITIVE	Non-reserved	Non-reserved	Reserved
INSERT	Non-reserved	Reserved	Reserved
INSTANCE	-	Non-reserved	-
INSTANTIABLE	-	Non-reserved	-
INSTEAD	Non-reserved	-	-
INT	Non-reserved (excluding functions and types)	Reserved	Reserved
INTEGER	Non-reserved (excluding functions and types)	Reserved	Reserved
INTERNAL	Reserved	-	-
INTERSECT	Reserved	Reserved	Reserved
INTERVAL	Non-reserved (excluding functions and types)	Reserved	Reserved
INTO	Reserved	Reserved	Reserved
INVALID	Non-reserved	-	-
INVOKER	Non-reserved	Non-reserved	-
IS	Reserved	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
ISNULL	Non-reserved (excluding functions and types)	-	-
ISOLATION	Non-reserved	Reserved	Reserved
ITERATE	-	Reserved	-
JOIN	Reserved (functions and types allowed)	Reserved	Reserved
K	-	Non-reserved	-
KEEP	Non-reserved	-	-
KEEPALIVE	Non-reserved	-	-
KEY	Non-reserved	Reserved	Reserved
KEY_MEMBER	-	Non-reserved	-
KEY_TYPE	-	Non-reserved	-
LABEL	Non-reserved	-	-
LANGUAGE	Non-reserved	Reserved	Reserved
LARGE	Non-reserved	Reserved	-
LAST	Non-reserved	Reserved	Reserved
LATERAL	-	Reserved	-
LC_COLLATE	Non-reserved	-	-
LC_CTYPE	Non-reserved	-	-
LEADING	Reserved	Reserved	Reserved
LEAKPROOF	Non-reserved	-	-
LEAST	Non-reserved (excluding functions and types)	-	-
LEFT	Reserved (functions and types allowed)	Reserved	Reserved
LENGTH	-	Non-reserved	Non-reserved
LESS	Reserved	Reserved	-
LEVEL	Non-reserved	Reserved	Reserved
LIGHT	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
LIKE	Reserved (functions and types allowed)	Reserved	Reserved
LIMIT	Reserved	Reserved	-
LIST	Non-reserved	-	-
LISTEN	Non-reserved	-	-
LOAD	Non-reserved	-	-
LOCAL	Non-reserved	Reserved	Reserved
LOCALTIME	Reserved	Reserved	-
LOCALTIMESTAMP	Reserved	Reserved	-
LOCATION	Non-reserved	-	-
LOCATOR	-	Reserved	-
LOCK	Non-reserved	-	-
LOG	Non-reserved	-	-
LOGGING	Non-reserved	-	-
LOGIN	Non-reserved	-	-
LOOP	Non-reserved	-	-
LOWER	-	Non-reserved	Reserved
M	-	Non-reserved	-
MAP	-	Reserved	-
MAPPING	Non-reserved	-	-
MASKED	Non-reserved	-	-
MATCH	Non-reserved	Reserved	Reserved
MATCHED	Non-reserved	-	-
MATERIALIZED	Non-reserved	-	-
MAX	-	Non-reserved	Reserved
MAXEXTENTS	Non-reserved	-	-
MAXSIZE	Non-reserved	-	-
MAXTRANS	Non-reserved	-	-
MAXVALUE	Reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
MERGE	Non-reserved	-	-
MESSAGE_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_OCTET_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_TEXT	-	Non-reserved	Non-reserved
METAADDRESS	Non-reserved	-	-
METHOD	-	Non-reserved	-
MIN	-	Non-reserved	Reserved
MINEXTENTS	Non-reserved	-	-
MINUS	Reserved	-	-
MINUTE	Non-reserved	Reserved	Reserved
MINVALUE	Non-reserved	-	-
MOD	-	Non-reserved	-
MODE	Non-reserved	-	-
MODIFIES	-	Reserved	-
MODIFY	Reserved	Reserved	-
MODULE	-	Reserved	Reserved
MONTH	Non-reserved	Reserved	Reserved
MORE	-	Non-reserved	Non-reserved
MOVE	Non-reserved	-	-
MOVEMENT	Non-reserved	-	-
MUMPS	-	Non-reserved	Non-reserved
NAME	Non-reserved	Non-reserved	Non-reserved
NAMES	Non-reserved	Reserved	Reserved
NATIONAL	Non-reserved (excluding functions and types)	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
NATURAL	Reserved (functions and types allowed)	Reserved	Reserved
NBUCKETS	Non-reserved	-	-
NCHAR	Non-reserved (excluding functions and types)	Reserved	Reserved
NCLOB	-	Reserved	-
NEW	-	Reserved	-
NEXT	Non-reserved	Reserved	Reserved
NLSSORT	Reserved	-	-
NO	Non-reserved	Reserved	Reserved
NOCOMPRESS	Non-reserved	-	-
NOCYCLE	Non-reserved	-	-
NODE	Non-reserved	-	-
NOLOGGING	Non-reserved	-	-
NOLOGIN	Non-reserved	-	-
NOMAXVALUE	Non-reserved	-	-
NOMINVALUE	Non-reserved	-	-
NONE	Non-reserved (excluding functions and types)	Reserved	-
NOT	Reserved	Reserved	Reserved
NOTHING	Non-reserved	-	-
NOTIFY	Non-reserved	-	-
NOTNULL	Reserved (functions and types allowed)	-	-
NOWAIT	Non-reserved	-	-
NULL	Reserved	Reserved	Reserved
NULLABLE	-	Non-reserved	Non-reserved
NULLIF	Non-reserved (excluding functions and types)	Non-reserved	Reserved
NULLS	Non-reserved	-	-
NUMBER	Non-reserved (excluding functions and types)	Non-reserved	Non-reserved

Keyword	DWS	SQL:1999	SQL:1992
NUMERIC	Non-reserved (excluding functions and types)	Reserved	Reserved
NUMSTR	Non-reserved	-	-
NVARCHAR2	Non-reserved (excluding functions and types)	-	-
NVL	Non-reserved (excluding functions and types)	-	-
OBJECT	Non-reserved	Reserved	-
OCTET_LENGTH	-	Non-reserved	Reserved
OF	Non-reserved	Reserved	Reserved
OFF	Non-reserved	Reserved	-
OFFSET	Reserved	-	-
OIDS	Non-reserved	-	-
OLD	-	Reserved	-
ON	Reserved	Reserved	Reserved
ONLY	Reserved	Reserved	Reserved
OPEN	-	Reserved	Reserved
OPERATION	-	Reserved	-
OPERATOR	Non-reserved	-	-
OPTIMIZATION	Non-reserved	-	-
OPTION	Non-reserved	Reserved	Reserved
OPTIONS	Non-reserved	Non-reserved	-
OR	Reserved	Reserved	Reserved
ORDER	Reserved	Reserved	Reserved
ORDINALITY	Non-reserved	Reserved	-
OUT	Non-reserved (excluding functions and types)	Reserved	-
OUTER	Reserved (functions and types allowed)	Reserved	Reserved
OUTLINE	Non-reserved	-	-
OUTPUT	-	Reserved	Reserved

Keyword	DWS	SQL:1999	SQL:1992
OVER	Non-reserved	-	-
OVERLAPS	Reserved (functions and types allowed)	Non-reserved	Reserved
OVERLAY	Non-reserved (excluding functions and types)	Non-reserved	-
OVERRIDING	-	Non-reserved	-
OVERWRITE	Non-reserved	-	-
OWNED	Non-reserved	-	-
OWNER	Non-reserved	-	-
PACKAGE	Non-reserved	-	-
PAD	-	Reserved	Reserved
PARAMETER	-	Reserved	-
PARAMETERS	-	Reserved	-
PARAMETER_MODE	-	Non-reserved	-
PARAMETER_NAME	-	Non-reserved	-
PARAMETER_ORDINAL_POSITION	-	Non-reserved	-
PARAMETER_SPECIFIC_CATALOG	-	Non-reserved	-
PARAMETER_SPECIFIC_NAME	-	Non-reserved	-
PARAMETER_SPECIFIC_SCHEMA	-	Non-reserved	-
PARENT_NAME	Non-reserved	-	-
PARSER	Non-reserved	-	-
PARTIAL	Non-reserved	Reserved	Reserved
PARTITION	Non-reserved	-	-
PARTITIONS	Non-reserved	-	-
PASCAL	-	Non-reserved	Non-reserved
PASSING	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
PASSWORD	Non-reserved	-	-
PATH	-	Reserved	-
PCTFREE	Non-reserved	-	-
PER	Non-reserved	-	-
PERM	Non-reserved	-	-
PERCENT	Non-reserved	-	-
PERFORMANCE	Reserved	-	-
PLACING	Reserved	-	-
PLAN	Reserved	-	-
PLANS	Non-reserved	-	-
PLI	-	Non-reserved	Non-reserved
POLICY	Non-reserved	-	-
POOL	Non-reserved	-	-
POSITION	Non-reserved (excluding functions and types)	Non-reserved	Reserved
POSTFIX	-	Reserved	-
PRECEDING	Non-reserved	-	-
PRECISION	Non-reserved (excluding functions and types)	Reserved	Reserved
PREDICATE	Non-reserved	-	-
PREFERRED	Non-reserved	-	-
PREFIX	Non-reserved	Reserved	-
PREORDER	-	Reserved	-
PREPARE	Non-reserved	Reserved	Reserved
PREPARED	Non-reserved	-	-
PRESERVE	Non-reserved	Reserved	Reserved
PRESERVE_BLANKS	Non-reserved	-	-
PRIMARY	Reserved	Reserved	Reserved
PRIOR	Non-reserved	Reserved	Reserved
PRIVATE	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
PRIVILEGE	Non-reserved	-	-
PRIVILEGES	Non-reserved	Reserved	Reserved
PROCEDURAL	Non-reserved	-	-
PROCEDURE	Reserved	Reserved	Reserved
PROFILE	Non-reserved	-	-
PUBLIC	-	Reserved	Reserved
PUBLICATION	Non-reserved	-	-
PURGE	Non-reserved	-	-
QUERY	Non-reserved	-	-
QUOTE	Non-reserved	-	-
RANGE	Non-reserved	-	-
RAW	Non-reserved	-	-
READ	Non-reserved	Reserved	Reserved
READS	-	Reserved	-
REAL	Non-reserved (excluding functions and types)	Reserved	Reserved
REASSIGN	Non-reserved	-	-
REBUILD	Non-reserved	-	-
RECHECK	Non-reserved	-	-
RECURSIVE	Non-reserved	Reserved	-
RECYCLEBIN	Reserved (functions and types allowed)	-	-
REDACTION	Non-reserved	-	-
REF	Non-reserved	Reserved	-
REFRESH	Non-reserved	-	-
REFERENCES	Reserved	Reserved	Reserved
REFERENCING	-	Reserved	-
REINDEX	Non-reserved	-	-
REJECT	Reserved	-	-
RELATIVE	Non-reserved	Reserved	Reserved
RELEASE	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
REOPTIONS	Non-reserved	-	-
REMOTE	Non-reserved	-	-
RENAME	Non-reserved	-	-
REPEATABLE	Non-reserved	Non-reserved	Non-reserved
REPLACE	Non-reserved	-	-
REPLICA	Non-reserved	-	-
RESET	Non-reserved	-	-
RESIZE	Non-reserved	-	-
RESOURCE	Non-reserved	-	-
RESTART	Non-reserved	-	-
RESTRICT	Non-reserved	Reserved	Reserved
RESULT	-	Reserved	-
RETURN	Non-reserved	Reserved	-
RETURNED_LENGTH	-	Non-reserved	Non-reserved
RETURNED_OCTET_LENGTH	-	Non-reserved	Non-reserved
RETURNED_SQLSTATE	-	Non-reserved	Non-reserved
RETURNING	Reserved	-	-
RETURNS	Non-reserved	Reserved	-
REUSE	Non-reserved	-	-
REVOKE	Non-reserved	Reserved	Reserved
REWRITE	Non-reserved	-	-
RIGHT	Reserved (functions and types allowed)	Reserved	Reserved
ROLE	Non-reserved	Reserved	-
ROLLBACK	Non-reserved	Reserved	Reserved
ROLLUP	Non-reserved	Reserved	-
ROUTINE	-	Reserved	-

Keyword	DWS	SQL:1999	SQL:1992
ROUTINE_CATALOG	-	Non-reserved	-
ROUTINE_NAME	-	Non-reserved	-
ROUTINE_SCHEMA	-	Non-reserved	-
ROW	Non-reserved (excluding functions and types)	Reserved	-
ROWS	Non-reserved	Reserved	Reserved
ROW_COUNT	-	Non-reserved	Non-reserved
RULE	Non-reserved	-	-
SAVEPOINT	Non-reserved	Reserved	-
SCALE	-	Non-reserved	Non-reserved
SCHEMA	Non-reserved	Reserved	Reserved
SCHEMA_NAME	-	Non-reserved	Non-reserved
SCOPE	-	Reserved	-
SCROLL	Non-reserved	Reserved	Reserved
SEARCH	Non-reserved	Reserved	-
SECOND	Non-reserved	Reserved	Reserved
SECTION	-	Reserved	Reserved
SECURITY	Non-reserved	Non-reserved	-
SELECT	Reserved	Reserved	Reserved
SELF	-	Non-reserved	-
SENSITIVE	-	Non-reserved	-
SEPARATOR	Non-reserved	-	-
SEQUENCE	Non-reserved	Reserved	-
SEQUENCES	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
SERIALIZABLE	Non-reserved	Non-reserved	Non-reserved
SERVER	Non-reserved	-	-
SERVER_NAME	-	Non-reserved	Non-reserved
SESSION	Non-reserved	Reserved	Reserved
SESSION_USER	Reserved	Reserved	Reserved
SET	Non-reserved	Reserved	Reserved
SETOF	Non-reserved (excluding functions and types)	-	-
SETS	Non-reserved	Reserved	-
SHARE	Non-reserved	-	-
SHIPPABLE	Non-reserved	-	-
SHOW	Non-reserved	-	-
SIMILAR	Reserved (functions and types allowed)	Non-reserved	-
SIMPLE	Non-reserved	Non-reserved	-
SIZE	Non-reserved	Reserved	Reserved
SLOT	Non-reserved	-	-
SMALLDATETIME	Non-reserved (excluding functions and types)	-	-
SMALLDATETIME_FORMAT	Non-reserved	-	-
SMALLINT	Non-reserved (excluding functions and types)	Reserved	Reserved
SNAPSHOT	Non-reserved	-	-
SOME	Reserved	Reserved	Reserved
SOURCE	Non-reserved	Non-reserved	-
SPACE	Non-reserved	Reserved	Reserved
SPECIFIC	-	Reserved	-
SPECIFICTYPE	-	Reserved	-

Keyword	DWS	SQL:1999	SQL:1992
SPECIFIC_NAME	-	Non-reserved	-
SPILL	Non-reserved	-	-
SPLIT	Non-reserved	-	-
SQL	Non-reserved	Reserved	Reserved
SQLCODE	-	-	Reserved
SQLERROR	-	-	Reserved
SQLException	-	Reserved	-
SQLSTATE	-	Reserved	Reserved
SQLWARNING	-	Reserved	-
STABLE	Non-reserved	-	-
STANDALONE	Non-reserved	-	-
START	Non-reserved	Reserved	-
STATE	-	Reserved	-
STATEMENT	Non-reserved	Reserved	-
STATEMENT_ID	Non-reserved	-	-
STATIC	-	Reserved	-
STATISTICS	Non-reserved	-	-
STDIN	Non-reserved	-	-
STDOUT	Non-reserved	-	-
STORAGE	Non-reserved	-	-
STORE	Non-reserved	-	-
STRICT	Non-reserved	-	-
STRIP	Non-reserved	-	-
STRUCTURE	-	Reserved	-
STYLE	-	Non-reserved	-
SUBCLASS_ORIGIN	-	Non-reserved	Non-reserved
SUBLIST	-	Non-reserved	-
SUBSCRIPTION	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
SUBSTRING	Non-reserved (excluding functions and types)	Non-reserved	Reserved
SUM	-	Non-reserved	Reserved
SYMMETRIC	Reserved	Non-reserved	-
SYNONYM	Non-reserved	-	-
SYS_REFCURSOR	Non-reserved	-	-
SYSDATE	Reserved	-	-
SYSID	Non-reserved	-	-
SYSTEM	Non-reserved	Non-reserved	-
SYSTEM_USER	-	Reserved	Reserved
TABLE	Reserved	Reserved	Reserved
TABLES	Non-reserved	-	-
TABLESAMPLE	Reserved (functions and types allowed)	-	-
TABLE_NAME	-	Non-reserved	Non-reserved
TEMP	Non-reserved	-	-
TEMPLATE	Non-reserved	-	-
TEMPORARY	Non-reserved	Reserved	Reserved
TERMINATE	-	Reserved	-
TEXT	Non-reserved	-	-
THAN	Non-reserved	Reserved	-
THEN	Reserved	Reserved	Reserved
TIME	Non-reserved (excluding functions and types)	Reserved	Reserved
TIME_FILL	Non-reserved (excluding functions and types)	-	-
TIME_FORMAT	Non-reserved	-	-
TIMECAPSULE	Reserved (functions and types allowed)	-	-

Keyword	DWS	SQL:1999	SQL:1992
TIMESTAMP	Non-reserved (excluding functions and types)	Reserved	Reserved
TIMESTAMPADD	Non-reserved (excluding functions and types)	-	-
TIMESTAMPDIFF	Non-reserved (excluding functions and types)	-	-
TIMESTAMP_FORMAT	Non-reserved	-	-
TIMEZONE_HOUR	-	Reserved	Reserved
TIMEZONE_MINUTE	-	Reserved	Reserved
TINYINT	Non-reserved (excluding functions and types)	-	-
TO	Reserved	Reserved	Reserved
TRAILING	Reserved	Reserved	Reserved
TRANSACTION	Non-reserved	Reserved	Reserved
TRANSACTIONS_COMMITTED	-	Non-reserved	-
TRANSACTIONS_ROLLED_BACK	-	Non-reserved	-
TRANSACTION_ACTIVE	-	Non-reserved	-
TRANSFORM	-	Non-reserved	-
TRANSFORMS	-	Non-reserved	-
TRANSLATE	-	Non-reserved	Reserved
TRANSLATION	-	Reserved	Reserved
TREAT	Non-reserved (excluding functions and types)	Reserved	-
TRIGGER	Non-reserved	Reserved	-
TRIGGER_CATALOG	-	Non-reserved	-
TRIGGER_NAME	-	Non-reserved	-
TRIGGER_SCHEMA	-	Non-reserved	-

Keyword	DWS	SQL:1999	SQL:1992
TRIM	Non-reserved (excluding functions and types)	Non-reserved	Reserved
TRUE	Reserved	Reserved	Reserved
TRUNCATE	Non-reserved	-	-
TRUSTED	Non-reserved	-	-
TRY_CAST	Non-reserved	-	-
TSTAG	Reserved. This field is used only in the hybrid data warehouse.	-	-
TSTIME	Reserved. This field is used only in the hybrid data warehouse.	-	-
TSFIELD	Reserved. This field is used only in the hybrid data warehouse.	-	-
TYPE	Non-reserved	Non-reserved	Non-reserved
TYPES	Non-reserved	-	-
UESCAPE	-	-	-
UNBOUNDED	Non-reserved	-	-
UNCOMMITTED	Non-reserved	Non-reserved	Non-reserved
UNDER	-	Reserved	-
UNENCRYPTED	Non-reserved	-	-
UNION	Reserved	Reserved	Reserved
UNIQUE	Reserved	Reserved	Reserved
UNKNOWN	Non-reserved	Reserved	Reserved
UNLIMITED	Non-reserved	-	-
UNLISTEN	Non-reserved	-	-
UNLOCK	Non-reserved	-	-
UNLOGGED	Non-reserved	-	-
UNNAMED	-	Non-reserved	Non-reserved
UNNEST	-	Reserved	-

Keyword	DWS	SQL:1999	SQL:1992
UNTIL	Non-reserved	-	-
UNUSABLE	Non-reserved	-	-
UPDATE	Non-reserved	Reserved	Reserved
UPPER	-	Non-reserved	Reserved
USAGE	-	Reserved	Reserved
USER	Reserved	Reserved	Reserved
USER_DEFINED_TYPE_CATALOG	-	Non-reserved	-
USER_DEFINED_TYPE_NAME	-	Non-reserved	-
USER_DEFINED_TYPE_SCHEMA	-	Non-reserved	-
USING	Reserved	Reserved	Reserved
VACUUM	Non-reserved	-	-
VALID	Non-reserved	-	-
VALIDATE	Non-reserved	-	-
VALIDATION	Non-reserved	-	-
VALIDATOR	Non-reserved	-	-
VALUE	Non-reserved	Reserved	Reserved
VALUES	Non-reserved (excluding functions and types)	Reserved	Reserved
VARCHAR	Non-reserved (excluding functions and types)	Reserved	Reserved
VARCHAR2	Non-reserved (excluding functions and types)	-	-
VARIABLE	-	Reserved	-
VARIADIC	Reserved	-	-
VARYING	Non-reserved	Reserved	Reserved
VCGROUP	Non-reserved	-	-
VERBOSE	Reserved (functions and types allowed)	-	-
VERIFY	Non-reserved	-	-

Keyword	DWS	SQL:1999	SQL:1992
VERSION	Non-reserved	-	-
VIEW	Non-reserved	Reserved	Reserved
VOLATILE	Non-reserved	-	-
WARMUP	Reserved	-	-
WHEN	Reserved	Reserved	Reserved
WHENEVER	-	Reserved	Reserved
WHERE	Reserved	Reserved	Reserved
WHITESPACE	Non-reserved	-	-
WINDOW	Reserved	-	-
WITH	Reserved	Reserved	Reserved
WITHIN	Non-reserved	-	-
WITHOUT	Non-reserved	Reserved	-
WORK	Non-reserved	Reserved	Reserved
WORKLOAD	Non-reserved	-	-
WRAPPER	Non-reserved	-	-
WRITE	Non-reserved	Reserved	Reserved
XML	Non-reserved	-	-
XMLATTRIBUTES	Non-reserved (excluding functions and types)	-	-
XMLCONCAT	Non-reserved (excluding functions and types)	-	-
XMLELEMENT	Non-reserved (excluding functions and types)	-	-
XMLEXISTS	Non-reserved (excluding functions and types)	-	-
XMLFOREST	Non-reserved (excluding functions and types)	-	-
XMLNAMESPACES	Non-reserved (excluding functions and types)	-	-
XMLPARSE	Non-reserved (excluding functions and types)	-	-
XMLPI	Non-reserved (excluding functions and types)	-	-

Keyword	DWS	SQL:1999	SQL:1992
XMLROOT	Non-reserved (excluding functions and types)	-	-
XMLSERIALIZE	Non-reserved (excluding functions and types)	-	-
XMLTABLE	Non-reserved (excluding functions and types)	-	-
YEAR	Non-reserved	Reserved	Reserved
YES	Non-reserved	-	-
ZONE	Non-reserved	Reserved	Reserved

5 Data Types

5.1 Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals.

For details about numeric operators and functions, see [Numeric Functions and Operators](#).

DWS supports integers, arbitrary precision numbers, floating point types, and serial integers.

Integer Types

The types TINYINT, SMALLINT, INTEGER, BINARY_INTEGER, and BIGINT store whole numbers, that is, numbers without fractional components, of various ranges. Saving a number with a decimal in any of the data types will result in errors.

The type INTEGER is the common choice. Generally, use the SMALLINT type only if you are sure that the value range is within the SMALLINT value range. The storage speed of INTEGER is much faster. BIGINT is used only when the range of INTEGER is not large enough.

Table 5-1 Integer types

Column	Description	Storage Space	Range
TINYINT	Tiny integer, also called INT1	1 byte	0 ~ 255
SMALLINT	Small integer, also called INT2	2 bytes	-32,768 ~ +32,767

Column	Description	Storage Space	Range
INTEGER	Typical choice for integer, also called INT4	4 bytes	-2,147,483,648 ~ +2,147,483,647
BINARY_INTEGER	INTEGER alias, compatible with Oracle	4 bytes	-2,147,483,648 ~ +2,147,483,647
BIGINT	Big integer, also called INT8	8 bytes	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

Examples:

Create a table containing TINYINT, INTEGER, and BIGINT data.

```
CREATE TABLE int_type_t1
(
  a TINYINT,
  b TINYINT,
  c INTEGER,
  d BIGINT
);
```

Insert data.

```
INSERT INTO int_type_t1 VALUES(100, 10, 1000, 10000);
```

View data.

```
SELECT * FROM int_type_t1;
 a | b | c | d
-----+-----
100 | 10 | 1000 | 10000
(1 row)
```

Arbitrary Precision Types

The type NUMBER can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. The arbitrary precision numbers require larger storage space and have lower storage efficiency, operation efficiency, and poorer compression ratio results than integer types.

The scale of a NUMBER value is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of a NUMBER value is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

To configure a numeric or decimal column, you are advised to specify both the maximum precision (p) and the maximum scale (s) of the column.

If the precision or scale of a value is greater than the declared scale of the column, the system will round the value to the specified number of fractional

digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error will be reported.

Table 5-2 Any-precision types

Column	Description	Storage Space	Range
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p].	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified
NUMBER[(p[,s])]	Alias for type NUMERIC, compatible with Oracle	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified

Examples:

Create a table with DECIMAL values.

```
CREATE TABLE decimal_type_t1 (DT_COL1 DECIMAL(10,4));
```

Insert data.

```
INSERT INTO decimal_type_t1 VALUES(123456.122331);
```

View data.

```
SELECT * FROM decimal_type_t1;  
dt_col1  
-----  
123456.1223  
(1 row)
```

Floating-Point Types

The floating-point type is an inexact, variable-precision numeric type. This type is an implementation of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, OS, and compiler support it.

Table 5-3 Floating point types

Column	Description	Storage Space	Range
REAL, FLOAT4	Single precision floating points, inexact	4 bytes	Six bytes of decimal digits
DOUBLE PRECISION , FLOAT8	Double precision floating points, inexact	8 bytes	1E-307~1E+308, 15 bytes of decimal digits
FLOAT[(p)]	Floating points, inexact. The value range of precision (p) is [1,53]. NOTE p is the precision, indicating the total decimal digits.	4 or 8 bytes	REAL or DOUBLE PRECISION is selected as an internal identifier based on different precision (p). If no precision is specified, DOUBLE PRECISION is used as the internal identifier.
BINARY_DOUBLE	DOUBLE PRECISION alias, compatible with Oracle	8 bytes	1E-307~1E+308, 15 bytes of decimal digits
DEC[(p[,s])]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p]. NOTE p indicates the total digits, and s indicates the decimal digit.	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified
INTEGER[(p[,s])]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p].	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified

Examples:

Create a table with floating-point values.

```
CREATE TABLE float_type_t2  
(
```

```
FT_COL1 INTEGER,  
FT_COL2 FLOAT4,  
FT_COL3 FLOAT8,  
FT_COL4 FLOAT(3),  
FT_COL5 BINARY_DOUBLE,  
FT_COL6 DECIMAL(10,4),  
FT_COL7 INTEGER(6,3)  
) DISTRIBUTE BY HASH ( ft_col1);
```

Insert data.

```
INSERT INTO float_type_t2 VALUES(10,10.365456,123456.1234,10.3214, 321.321, 123.123654, 123.123654);
```

View data.

```
SELECT * FROM float_type_t2;  
ft_col1 | ft_col2 | ft_col3 | ft_col4 | ft_col5 | ft_col6 | ft_col7  
-----+-----+-----+-----+-----+-----+-----  
10 | 10.3655 | 123456.1234 | 10.3214 | 321.321 | 123.1237 | 123.124  
(1 row)
```

Serial Integers

SMALLSERIAL, SERIAL, and BIGSERIAL are not true types, but merely a notational convenience for creating unique identifier columns. Therefore, an integer column is created and its default value plans to be read from a sequencer. A NOT NULL constraint is used to ensure NULL is not inserted. In most cases you would also want to attach a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted unexpectedly. Lastly, the sequence is marked as "owned by" the column, so that it will be dropped if the column or table is dropped. Currently, the SERIAL column can be specified only when you create a table. You cannot add the SERIAL column in an existing table. In addition, SERIAL columns cannot be created in temporary tables. Because SERIAL is not a data type, columns cannot be converted to this type.

Table 5-4 Sequence integer

Column	Description	Storage Space	Range
SMALLSERIAL	Two-byte auto-incrementing integer	2 bytes	1 ~ 32,767
SERIAL	Four-byte auto-incrementing integer	4 bytes	1 ~ 2,147,483,647
BIGSERIAL	Eight-byte auto-incrementing integer	8 bytes	1 ~ 9,223,372,036,854,775,807

Examples:

Create a table with serial values.

```
CREATE TABLE smallserial_type_tab(a SMALLSERIAL);
```

Insert data.


```
INSERT INTO smallserial_type_tab VALUES(default);
```

Insert data again.

```
INSERT INTO smallserial_type_tab VALUES(default);
```

View data.

```
SELECT * FROM smallserial_type_tab;  
a  
---  
1  
2  
(2 rows)
```

5.2 Monetary Types

The **money** type stores a currency amount with fixed fractional precision. The range shown in [Table 5-5](#) assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting, such as \$1,000.00. Output is generally in the latter form but depends on the locale.

Table 5-5 Monetary types

Name	Storage Size	Description	Range
money	8 bytes	Currency amount	-92233720368547758.08 to +92233720368547758.07

Values of the **numeric**, **int**, and **bigint** data types can be cast to **money**. Conversion from the **real** and **double precision** data types can be done by casting to **numeric** first, for example:

```
SELECT '12.34'::float8::numeric::money;
```

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A **money** value can be cast to **numeric** without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

```
SELECT '52093.89'::money::numeric::float8;
```

When a **money** value is divided by another **money** value, the result is **double precision** (that is, a pure number, not money); the currency units cancel each other out in the division.

5.3 Boolean Type

Table 5-6 Boolean type

Name	Description	Storage Space	Value
BOOLEAN	Boolean type	1 byte	• true • false • null (unknown)

Valid literal values for the "true" state are:

TRUE, 't', 'true', 'y', 'yes', '1'

Valid literal values for the "false" state include:

FALSE, 'f', 'false', 'n', 'no', '0'

TRUE and **FALSE** are standard expressions, compatible with SQL statements.

Examples

Data type **boolean** is displayed with letters **t** and **f**.

```
-- Create a table:
CREATE TABLE bool_type_t1
(
    BT_COL1 BOOLEAN,
    BT_COL2 TEXT
) DISTRIBUTE BY HASH(BT_COL2);

--Insert data:
INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');

INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');

-- View data:
SELECT * FROM bool_type_t1;
bt_col1 | bt_col2
-----+-----
t       | sic est
f       | non est
(2 rows)

SELECT * FROM bool_type_t1 WHERE bt_col1 = 't';
bt_col1 | bt_col2
-----+-----
t       | sic est
(1 row)

-- Delete the tables:
DROP TABLE bool_type_t1;
```

5.4 Character Types

Table 5-7 lists the character types that can be used in DWS. For string operators and related built-in functions, see [Character Processing Functions and Operators](#).

Table 5-7 Character types

Name	Description	Length	Storage Space
CHAR(n) CHARACTER(n) NCHAR(n)	Fixed-length character string. If the length is not reached, fill in spaces.	n indicates the string length. If it is not specified, the default precision 1 is used. The value of n is less than 10485761 .	The maximum size is 10 MB.
VARCHAR(n) CHARACTER VARYING(n)	Variable-length string.	n indicates the byte length. The value of n is less than 10485761 .	The maximum size is 10 MB.
VARCHAR2(n)	Variable-length string. It is an alias for VARCHAR(n) type, compatible with Oracle.	n indicates the byte length. The value of n is less than 10485761 .	The maximum size is 10 MB.
NVARCHAR2(n)	Variable-length string.	n indicates the string length. The value of n is less than 10485761 .	The maximum size is 10 MB.
CLOB	Variable-length string. A big text object. It is an alias for TEXT type, compatible with Oracle.	-	The maximum size is 1,073,733,621 bytes (1 GB - 8203 bytes).
TEXT	Variable-length string.	-	The maximum size is 1,073,733,621 bytes (1 GB - 8203 bytes).

 NOTE

- In addition to the size limitation on each column, the total size of each tuple is 1,073,733,621 bytes (1 GB – 8023 bytes).
- For string data, you are advised to use variable-length strings and specify the maximum length. To avoid truncation, ensure that the specified maximum length is greater than the maximum number of characters to be stored. You are not advised to use fixed-length character types such as CHAR(n), NCHAR(n), and CHARACTER(n) unless you know that the data type is a fixed-length character string.

DWS has two other fixed-length character types, as listed in [Table 5-8](#).

The name type is used only in the internal system catalog as the storage identifier. The length of this type is 64 bytes (63 characters plus the terminator). This data type is not recommended for common users. When the name type is aligned with other data types (for example, in multiple branches of **case when**, one branch returns the name type and other branches return the text type), the name type may be aligned but characters may be truncated. If you do not want to have 64-bit truncated characters, you need to forcibly convert the name type to the text type.

The type "**char**" only uses one byte of storage. It is internally used in the system catalogs as a simplistic enumeration type.

Table 5-8 Special character types

Name	Description	Storage Space
name	Internal type for object names	64 bytes
"char"	Single-byte internal type	1 byte

Length

If a field is defined as **char(n)** or **varchar(n)**, **n** indicates the maximum length. Regardless of the type, the length cannot exceed 10485760 bytes (10 MB).

When the data length exceeds the specified length **n**, the error "value too long" is reported. Of course, you can also specify to automatically truncate the data that exceeds the length.

Example:

1. Create table **t1** and specify the character type of its columns.

```
CREATE TABLE t1 (a char(5),b varchar(5));
```
2. An error is reported when the length of data inserted into the table **t1** exceeds the specified length.

```
INSERT INTO t1 VALUES('bookstore','123');  
ERROR: value too long for type character(5)  
CONTEXT: referenced column: a
```
3. Insert data into table **t1** and specify that the data is automatically truncated when the length exceeds the specified bytes.

```
INSERT INTO t1 VALUES('bookstore':char(5),'12345678':varchar(5));  
INSERT 0 1  
  
SELECT a,b FROM t1;
```

```
a | b
---+---
books | 12345
(1 row)
```

Fixed Length and Variable Length

All character types can be classified into fixed-length strings and variable-length strings.

- For a fixed-length string, the length must be specified. If the length is not specified, the default length **1** is used. If the data length does not reach the specified length, spaces are automatically added to the end of the string. However, the added spaces are meaningless and will be ignored in actual use, such as comparison, sorting, and type conversion.
- For a variable-length string, if the length is specified, the specified length indicates the maximum length of the data that can be stored. If the length is not specified, it means any length is available.

Example:

1. Create table **t2** and specify the character type of its columns.

```
CREATE TABLE t2 (a char(5),b varchar(5));
```

2. Insert data into table **t2** and query the byte length of column **a**. During table creation, the character type of column **a** is specified as **char(5)** and fixed-length. If the data length does not reach 5 bytes, spaces are added. Therefore, the queried data length is **5**.

```
INSERT INTO t2 VALUES('abc','abc');
INSERT 0 1
```

```
SELECT a,lengthb(a),b FROM t2;
```

```
a | lengthb | b
---+---+---
abc |      5 | abc
(1 row)
```

3. After the conversion by using the function, the actual queried length of the column **a** is **3** bytes.

```
SELECT a = b from t2;
?column?
```

```
-----
t
(1 row)
```

```
SELECT cast(a as text) as val,lengthb(val) FROM t2;
```

```
val | lengthb
---+---
abc |      3
(1 row)
```

Bytes and Characters

n means differently in **VARCHAR2(n)** and **NVARCHAR2(n)**.

- In **VARCHAR2(n)** **n** indicates the number of bytes.
- In **NVARCHAR2(n)**, **n** indicates the number of characters.

 NOTE

Take an UTF8-encoded database as an example. A letter occupies one byte, and a Chinese character occupies three bytes. **VARCHAR2(6)** allows for six letters or two Chinese characters, and **NVARCHAR2(6)** allows for six letters or six Chinese characters.

Empty Strings and NULL

In Oracle compatibility mode, empty strings and NULL are not distinguished. When a statement is executed to query or import data, empty strings are processed as NULL.

As such, = '' cannot be used as the query condition, and so does is ''. Otherwise, no result set is returned. The correct usage is **is null**, or **is not null**.

Example:

1. Create table **t4** and specify the character type of its columns.
`CREATE TABLE t4 (a text);`
2. Insert data into table **t4**. The inserted value contains an empty string and NULL.
`INSERT INTO t4 VALUES('abc'),(''),(null);`
`INSERT 0 3`
3. Check whether **t4** contains null values.

```
SELECT a,a isnull FROM t4;
a | ?column?
-----+-----
| t
| t
abc | f
(3 rows)

SELECT a,a isnull FROM t4 WHERE a is null;
a | ?column?
-----+-----
| t
| t
(2 rows)
```

5.5 Binary Data Types

[Table 5-9](#) lists the binary data types that can be used in DWS.

Table 5-9 Binary Data Types

Name	Description	Storage Space
BLOB	Binary large object. Currently, BLOB only supports the following external access interfaces: • DBMS_LOB.GETLENGTH • DBMS_LOB.READ • DBMS_LOB.WRITE • DBMS_LOB.WRITEAPPEND • DBMS_LOB.COPY • DBMS_LOB.ERASE For details about these interfaces, see section "DBMS_LOB" in the <i>Data Warehouse Service (DWS) Developer Guide</i> . NOTE Column storage cannot be used for the BLOB type.	The maximum size is 10,7373,3621 bytes (1 GB - 8203 bytes).
RAW	Variable-length hexadecimal string NOTE Column storage cannot be used for the raw type.	4 bytes plus the actual hexadecimal string. The maximum size is 10,7373,3621 bytes (1 GB - 8203 bytes).
BYTEA	Variable-length binary string	4 bytes plus the actual binary string. The maximum size is 10,7373,3621 bytes (1 GB - 8203 bytes).

 NOTE

In addition to the size limitation on each column, the total size of each tuple is 8203 bytes less than 1 GB.

Examples

```
-- Create a table:
CREATE TABLE blob_type_t1
(
    BT_COL1 INTEGER,
    BT_COL2 BLOB,
    BT_COL3 RAW,
    BT_COL4 BYTEA
) DISTRIBUTE BY REPLICATION;

--Insert data:
INSERT INTO blob_type_t1 VALUES(10,empty_blob(),
HEXTORAW('DEADBEEF'),E'\xDEADBEEF');
```

```
-- Query data in the table:
SELECT * FROM blob_type_t1;
bt_col1 | bt_col2 | bt_col3 | bt_col4
-----+-----+-----+-----
      10 |      | DEADBEEF | \xdeadbeef
(1 row)

-- Delete the tables:
DROP TABLE blob_type_t1;
```

5.6 Date/Time Types

Table 5-10 lists date and time types supported by DWS. For the operators and built-in functions of the types, see [Date and Time Processing Functions and Operators](#).

NOTE

If the time format of another database is different from that described in DWS, modify the value of the [DateStyle](#) parameter to keep them consistent.

Table 5-10 Date/Time Types

Name	Description	Storage Space
DATE	In Oracle compatibility mode, it is equivalent to timestamp(0) and records the date and time. In other modes, it records the date.	In Oracle compatibility mode, it occupies 8 bytes. In Oracle compatibility mode, it occupies 4 bytes.
TIME [(p)] [WITHOUT TIME ZONE]	Specifies time within one day. p indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes
TIME [(p)] [WITH TIME ZONE]	Specifies time within one day (with time zone). p indicates the precision after the decimal point. The value ranges from 0 to 6.	12 bytes
TIMESTAMP[(p)] [WITHOUT TIME ZONE]	Specifies the date and time. p indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes

Name	Description	Storage Space
TIMESTAMP[(p)] [WITH TIME ZONE]	Specifies the date and time (with time zone). TIMESTAMP is also called TIMESTAMPTZ. p indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes
SMALLDATETIME	Specifies the date and time (without time zone). The precision level is minute. 31s to 59s are rounded into 1 minute.	8 bytes
INTERVAL DAY (l) TO SECOND (p)	Specifies the time interval (X days X hours X minutes X seconds). l: indicates the precision of days. The value ranges from 0 to 6. To adapt to Oracle syntax, the precision functions are not supported. p: indicates the precision of seconds. The value ranges from 0 to 6. The digit 0 at the end of a decimal number is not displayed.	16 bytes

Name	Description	Storage Space
INTERVAL [FIELDS] [(p)]	Specifies the time interval. • fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND, HOUR TO MINUTE, HOUR TO SECOND, and MINUTE TO SECOND. • p: indicates the precision of seconds. The value ranges from 0 to 6. p takes effect only when fields are SECOND, DAY TO SECOND, HOUR TO SECOND, or MINUTE TO SECOND. The digit 0 at the end of a decimal number is not displayed. NOTE In MySQL compatibility mode of cluster version 9.1.0.200 and later, when the fields are YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND, you can use an integer or floating point number as the input parameter (the number should not include positive or negative signs). For example, you can use INTERVAL 1 YEAR or INTERVAL 2.5 MONTH.	12 bytes
reltime	Relative time interval. The format is: <i>X</i> years <i>X</i> months <i>X</i> days <i>XX:XX:XX</i> • The Julian calendar is used. It specifies that a year has 365.25 days and a month has 30 days. The relative time interval needs to be calculated based on the input value. The output format is POSTGRES.	4 bytes

Example:

```
--Create a table:
CREATE TABLE date_type_tab(coll date);

--Insert data:
INSERT INTO date_type_tab VALUES (date '12-10-2010');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
(1 row)
```

```
-- Delete the tables:
DROP TABLE date_type_tab;

--Create a table:
CREATE TABLE time_type_tab (da time without time zone ,dai time with time zone,dfgh timestamp without
time zone,dfga timestamp with time zone, vbg smalldatetime);

--Insert data:
INSERT INTO time_type_tab VALUES ('21:21:21','21:21:21 pst','2010-12-12','2013-12-11 pst','2003-04-12
04:05:06');

-- View data:
SELECT * FROM time_type_tab;
   da   |   dai   |   dfgh   |   dfga   |   vbg
-----+-----+-----+-----+-----
21:21:21 | 21:21:21-08 | 2010-12-12 00:00:00 | 2013-12-11 16:00:00+08 | 2003-04-12 04:05:00
(1 row)

-- Delete the tables:
DROP TABLE time_type_tab;

--Create a table:
CREATE TABLE day_type_tab (a int,b INTERVAL DAY(3) TO SECOND (4));

--Insert data:
INSERT INTO day_type_tab VALUES (1, INTERVAL '3' DAY);

-- View data:
SELECT * FROM day_type_tab;
 a | b
---+-----
 1 | 3 days
(1 row)

-- Delete the tables:
DROP TABLE day_type_tab;

--Create a table:
CREATE TABLE year_type_tab(a int, b interval year (6));

--Insert data:
INSERT INTO year_type_tab VALUES(1,interval '2' year);

-- View data:
SELECT * FROM year_type_tab;
 a | b
---+-----
 1 | 2 years
(1 row)

-- Delete the tables:
DROP TABLE year_type_tab;
```

Date Input

Date and time input is accepted in almost any reasonable formats, including ISO 8601, SQL-compatible, and traditional POSTGRES. The system allows you to customize the sequence of day, month, and year in the date input. Set the **DateStyle** parameter to **MDY** to select month-day-year interpretation, **DMY** to select day-month-year interpretation, or **YMD** to select year-month-day interpretation.

Remember that any date or time literal input needs to be enclosed with single quotes, and the syntax is as follows:

type [(p)] 'value'

The **p** that can be selected in the precision statement is an integer, indicating the number of fractional digits in the **seconds** column. [Table 5-11](#) shows some possible inputs for the **date** type.

Table 5-11 Date input

Example	Description
1999-01-08	ISO 8601 (recommended format). January 8, 1999 in any mode
January 8, 1999	Unambiguous in any date input mode
1/8/1999	January 8 in MDY mode. August 1 in DMY mode
1/18/1999	January 18 in MDY mode, rejected in other modes
01/02/03	- January 2, 2003 in MDY mode - February 1, 2003 in DMY mode - February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in YMD mode, else error
08-Jan-99	January 8, except error in YMD mode
Jan-08-99	January 8, except error in YMD mode
19990108	ISO 8601. January 8, 1999 in any mode
990108	ISO 8601. January 8, 1999 in any mode
1999.008	Year and day of year
J2451187	Julian date
January 8, 99 BC	Year 99 BC

Example:

```
--Create a table:
CREATE TABLE date_type_tab(coll date);

--Insert data:
INSERT INTO date_type_tab VALUES (date '12-10-2010');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
(1 row)

-- View the date format:
```

```
SHOW datestyle;
DateStyle
-----
ISO, MDY
(1 row)

-- Configure the date format:
SET datestyle='YMD';
SET

-- Insert data:
INSERT INTO date_type_tab VALUES(date '2010-12-11');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
2010-12-11 00:00:00
(2 rows)

-- Delete the tables:
DROP TABLE date_type_tab;
```

Times

The time-of-day types are **TIME [(p)] [WITHOUT TIME ZONE]** and **TIME [(p)] [WITH TIME ZONE]**. **TIME** alone is equivalent to **TIME WITHOUT TIME ZONE**.

If a time zone is specified in the input for **TIME WITHOUT TIME ZONE**, it is silently ignored.

For details about the time input types, see [Table 5-12](#). For details about time zone input types, see [Table 5-13](#).

Table 5-12 Time input

Example	Description
05:06.8	ISO 8601
4:05:06	ISO 8601
4:05	ISO 8601
40506	ISO 8601
4:05 AM	Same as 04:05. AM does not affect value
4:05 PM	Same as 16:05. Input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	Time zone specified by abbreviation

Table 5-13 Time zone input

Example	Description
PST	Abbreviation (for Pacific Standard Time)
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

Example:

```
SELECT time '04:05:06';
time
-----
04:05:06
(1 row)

SELECT time '04:05:06 PST';
time
-----
04:05:06
(1 row)

SELECT time with time zone '04:05:06 PST';
timetz
-----
04:05:06-08
(1 row)
```

Special Values

The special values supported by DWS are converted to common date/time values when being read. For details, see [Table 5-14](#).

Table 5-14 Special Values

Input String	Applicable Type	Description
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	timestamp	Later than any other timestamps
-infinity	timestamp	Earlier than any other timestamps
now	date, time, timestamp	Start time of the current transaction
today	date, timestamp	Today midnight
tomorrow	date, timestamp	Tomorrow midnight
yesterday	date, timestamp	Yesterday midnight

Input String	Applicable Type	Description
allballs	time	00:00:00.00 UTC

Interval Input

The input of **reltime** can be any valid interval in TEXT format. It can be a number (negative numbers and decimals are also allowed) or a specific time, which must be in SQL standard format, ISO-8601 format, or POSTGRES format. In addition, the text input needs to be enclosed with single quotation marks (').

For details, see [Table 5-15](#).

Table 5-15 Interval input

Input	Output	Description
60	2 mons	Numbers are used to indicate intervals. The default unit is day. Decimals and negative numbers are also allowed. Particularly, a negative interval syntactically means how long before.
31.25	1 mons 1 days 06:00:00	
-365	-12 mons -5 days	
1 years 1 mons 8 days 12:00:00	1 years 1 mons 8 days 12:00:00	Intervals are in POSTGRES format. They can contain both positive and negative numbers and are case-insensitive. Output is a simplified POSTGRES interval converted from the input.
-13 months -10 hours	-1 years -25 days -04:00:00	
-2 YEARS +5 MONTHS 10 DAYS	-1 years -6 mons -25 days -06:00:00	
P-1.1Y10M	-3 mons -5 days -06:00:00	Intervals are in ISO-8601 format. They can contain both positive and negative numbers and are case-insensitive. Output is a simplified POSTGRES interval converted from the input.
-12H	-12:00:00	

Example:

```
-- Create a table.
CREATE TABLE reltime_type_tab(col1 character(30), col2 reltime);

-- Insert data.
INSERT INTO reltime_type_tab VALUES ('90', '90');
INSERT INTO reltime_type_tab VALUES ('-366', '-366');
INSERT INTO reltime_type_tab VALUES ('1975.25', '1975.25');
INSERT INTO reltime_type_tab VALUES ('-2 YEARS +5 MONTHS 10 DAYS', '-2 YEARS +5 MONTHS 10 DAYS');
```

```
INSERT INTO reltime_type_tab VALUES ('30 DAYS 12:00:00', '30 DAYS 12:00:00');
INSERT INTO reltime_type_tab VALUES ('P-1.1Y10M', 'P-1.1Y10M');

-- View data.
SELECT * FROM reltime_type_tab;
      col1          |          col2
-----+-----
1975.25             | 5 years 4 mons 29 days
-2 YEARS +5 MONTHS 10 DAYS | -1 years -6 mons -25 days -06:00:00
P-1.1Y10M           | -3 mons -5 days -06:00:00
-366                | -1 years -18:00:00
90                  | 3 mons
30 DAYS 12:00:00    | 1 mon 12:00:00
(6 rows)

-- Delete tables.
DROP TABLE reltime_type_tab;
```

5.7 Geometric Types

Table 5-16 lists the geometric types that can be used in DWS. The most fundamental type, the point, forms the basis for all of the other types.

Table 5-16 Geometric Type

Name	Storage Space	Description	Representation
point	16 bytes	Point on a plane	(x,y)
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular Box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

A rich set of functions and operators is available in DWS to perform various geometric operations, such as scaling, translation, rotation, and determining intersections. For details, see [Geometric Functions and Operators](#).

Points

Points are the fundamental two-dimensional building block for geometric types. Values of the **point** type are specified using either of the following syntaxes:

```
( x , y )
x , y
```


where x and y are the respective coordinates, as floating-point numbers.

Points are output using the first syntax.

Line Segments

Line segments (**lseg**) are represented by pairs of points. Values of the **lseg** type are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]  
( ( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

where (x1,y1) and (x2,y2) are the end points of the line segment.

Line segments are output using the first syntax.

Rectangular Box

Boxes are represented by pairs of points that are opposite corners of the box. Values of the **box** type are specified using any of the following syntaxes:

```
(( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

where (x1,y1) and (x2,y2) are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but in this order, the values will be reordered as needed to store the upper right and lower left corners.

Path

Paths are represented by lists of connected points. Paths can be open, where the first and last points in the list are considered not connected, or closed, where the first and last points are considered connected.

Values of the **path** type are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]  
( ( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1 , ... , xn , yn )  
x1 , y1 , ... , xn , yn
```

where the points are the end points of the line segments comprising the path. Square brackets ([]) indicate an open path, while parentheses (()) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first or second syntax.

Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons are very similar to closed paths, but are stored differently and have their own set of support functions.

Values of the **polygon** type are specified using any of the following syntaxes:

```
(( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1 , ... , xn , yn )  
x1 , y1 , ... , xn , yn
```

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

Circle

Circles are represented by a center point and radius. Values of the **circle** type are specified using any of the following syntaxes:

```
< ( x , y ) , r >  
(( x , y ) , r )  
( x , y ) , r  
x , y , r
```

where **(x,y)** is the center point and **r** is the radius of the circle.

Circles are output using the first syntax.

5.8 Array

An array is a set of data. The array type allows a single database field to have multiple values. It is usually used to store and process data with similar attributes.

Syntax

```
ARRAY [ param ]
```

or

```
'{ param }'
```

The **param** parameter is described as follows:

- **param**: value contained in the array. It can be zero or many. Multiple values are separated by commas (,). If there is no value, set this parameter to NULL.
- When you specify an array constant in '{ *param* }' format, the elements of the string type cannot start or end with a single quotation mark. Instead, a double quotation mark is needed. Two consecutive single quotation marks will be converted to one single quotation mark.
- The data type of the first element is used as the data type of the array. All elements in an array must be of the same type, or their types can be converted to each other.

Definition of the Array Type

An array type is defined by adding values in the square brackets ([]) at the end of a data type.

Create the **books** table. The type of the **price** column (book prices) is a one-dimensional integer array, and the type of the **tags** column (book tags) is a two-dimensional text array.

```
CREATE TABLE books (id SERIAL PRIMARY KEY, title VARCHAR(100), price_by_quarter int[], tags TEXT[][]);
```

The **CREATE TABLE** syntax can specify the array size. For example:

```
CREATE TABLE test ( a int[3]);
```

In current database implementation, the array size limits of statements are ignored. Statements with array size limits behave the same as those without. At the same time, the number of declared dimensions is not required. Arrays of a particular element type are all treated as the same type. Their size or number of dimensions are ignored.

You can also use the keyword **ARRAY** to define a one-dimensional array. The **price** column in the **books** table is defined using **ARRAY** and the array size is specified as follows:

```
price_by_quarter int ARRAY[4]
```

Use **ARRAY** without specifying the array size:

```
price_by_quarter int ARRAY
```

Array Value Input

When entering an array value, write each array value as a literal constant, surround element values with braces, and separate them with commas. The general format of an array constant is as follows:

```
'{ val1 delim val2 delim ... }'
```

delim indicates a delimiter, and each **val** may be a constant or a subarray of an array element type.

An example of an array constant is as follows:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

The constant is a two-dimensional, 3-by-3 array consisting of three integer subarrays.

Insert data into the **books** table and query the **books** table.

```
INSERT INTO books
VALUES (1, 'One Hundred years of Solitude', '{25,25,25,25}', '{{"fiction"}, {"adventure"}}'),
      (2, 'Robinson Crusoe', '{30,32,32,32}', '{{"adventure"}, {"fiction"}}'),
      (3, 'Gone with the Wind', '{27,27,29,28}', '{{"romance"}, {"fantasy"}}');
```

```
SELECT * FROM books;
```

id	title	price_by_quarter	tags
1	One Hundred years of Solitude	{25,25,25,25}	{{fiction},{adventure}}
2	Robinson Crusoe	{30,32,32,32}	{{adventure},{fiction}}
3	Gone with the Wind	{27,27,29,28}	{{romance},{fantasy}}

(3 rows)



CAUTION

When multi-dimensional array data is inserted, each dimension of the array must have a matching length.

Use the **ARRAY** keyword to insert data.

```
INSERT INTO books
VALUES (1, 'One Hundred years of Solitude', ARRAY[25,25,25,25], ARRAY['fiction', 'adventure']),
      (2, 'Robinson Crusoe', ARRAY[30,32,32,32], ARRAY['adventure', 'fiction']),
      (3, 'Gone with the Wind', ARRAY[27,27,29,28], ARRAY['romance', 'fantasy']);
```

Accessing Arrays

Accessing Array Elements

Query the names of the books whose prices change in the second quarter.

```
SELECT title FROM books WHERE price_by_quarter[1] <> price_by_quarter[2];
title
-----
Robinson Crusoe
(1 row)
```

Search for the prices of all books in the third quarter:

```
SELECT price_by_quarter[3] FROM books;
price_by_quarter
-----
29
25
32
(3 rows)
```

Accessing Array Slices

Any rectangular slice or subarray of an array can be accessed. An array slice can be defined by specifying [lower bound: upper bound] on one or more array dimensions.

Query the second label of **Gone with the Wind**.

```
SELECT tags[2:2] FROM books WHERE title = 'Gone with the Wind';
tags
-----
{fantasy}
(1 row)
```

Using Functions to Access Arrays

Use the **array_dims** function to obtain the dimension of an array value.

```
SELECT array_dims(tags) FROM books WHERE title = 'Robinson Crusoe';
array_dims
-----
[1:2]
(1 row)
```

You can also use **array_upper** and **array_lower** to obtain the array dimension. They return the upper and lower bounds of a specified array, respectively.

```
SELECT array_upper(tags, 1) FROM books WHERE title = 'Robinson Crusoe';
array_upper
-----
2
(1 row)
```

The **array_length** function returns the length of a specified array dimension.

```
SELECT array_length(tags, 1) FROM books WHERE title = 'Robinson Crusoe';
array_length
-----
2
(1 row)
```

Modifying an Array

Updating an Array

Update the entire array data:

```
UPDATE books SET price_by_quarter = '{30,30,30,30}'  
WHERE title = 'Robinson Crusoe';
```

Use the ARRAY expression to update the entire array data:

```
UPDATE books SET price_by_quarter = ARRAY[30,30,30,30]  
WHERE title = 'Robinson Crusoe';
```

Update an element in the array:

```
UPDATE books SET price_by_quarter[4] = 35  
WHERE title = 'Robinson Crusoe';
```

Update a slice element in the array:

```
UPDATE books SET price_by_quarter[1:2] = '{27,27}'  
WHERE title = 'Robinson Crusoe';
```

A stored array value can be expanded by assigning a value to a new element. The position between an existing element and the new element is filled with null values. For example, if the array **myarray** currently has four elements, it will have six elements after a value is assigned to **myarray[6]** using **UPDATE**. **myarray[5]** is filled with null. Currently, this method can be used only on one-dimensional arrays.

Building a New Array

New arrays can also be constructed using the concatenation operator ||. The concatenation operator allows a single element to be added to the beginning or end of a one-dimensional array. It can also accept two *N* dimensional arrays, or an *N* dimensional array and an *N*+1 dimensional array.

```
SELECT ARRAY[1,2] || ARRAY[3,4];  
?column?  
-----  
{1,2,3,4}  
(1 row)  
  
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];  
?column?  
-----  
{{5,6},{1,2},{3,4}}  
(1 row)
```

Use the **array_prepend**, **array_append**, or **array_cat** function to build an array.

```
SELECT array_prepend(1, ARRAY[2,3]);  
array_prepend  
-----  
{1,2,3}  
(1 row)  
  
SELECT array_append(ARRAY[1,2], 3);  
array_append  
-----  
{1,2,3}  
(1 row)  
  
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);  
array_cat  
-----
```

```
{1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);

  array_cat
-----
{{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);

  array_cat
-----
{{5,6},{1,2},{3,4}}
(1 row)
```

5.9 Enumeration Type

An enumeration (enum) type is a data type consisting of a static, ordered set of values. They are equivalent to the enum types used in many programming languages. The enumeration type can be a date in a week or a set of status values.

Declaration of Enumeration Types

Enumeration types can be created using the **CREATE TYPE** command. For example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

After an enumeration type is created, it can be used in table and function definitions.

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (name text, current_mood mood);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
-----+-----
 Moe  | happy
(1 row)
```

Sorting

The order of enumeration values is the order of the values listed when the type is created.

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');

SELECT * FROM person WHERE current_mood > 'sad';
 name | current_mood
-----+-----
 Moe  | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name | current_mood
-----+-----
 Curly | ok
 Moe   | happy
(2 rows)
```

```
SELECT name FROM person WHERE current_mood = (SELECT MIN(current_mood) FROM person);
name
-----
Larry
(1 row)
```

Security of Enumeration Types

Each enumeration data type is independent and cannot be compared with other enumeration types.

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (num_weeks integer, happiness happiness);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');

INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"

SELECT person.name, holidays.num_weeks FROM person, holidays
       WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

If comparison is required, you can use a customized operator or add an explicit type to the query.

```
SELECT person.name, holidays.num_weeks FROM person, holidays
       WHERE person.current_mood::text = holidays.happiness::text;
name | num_weeks
-----+-----
Moe  |         4
(1 row)
```

Precautions

- Enumeration tags are case sensitive. 'happy' is different from 'HAPPY'. Spaces in tags also have meanings.
- Although enumeration types are primarily intended for static collections of values, there are ways to add new and renamed values to existing enumeration types (see [ALTER TYPE](#)). Existing values cannot be removed from an enumeration type, and the order of these values cannot be changed unless the enumeration type is deleted and recreated.
- The mapping between enumerated values and text tags is stored in the **PG_ENUM** system catalog.

5.10 Network Address Types

DWS offers data types to store IPv4, IPv6, and MAC addresses.

It is better to use network address types instead of plaintext types to store IPv4, IPv6, and MAC addresses, because these types offer input error checking and specialized operators and functions. For details, see [Network Address Functions and Operators](#).

Table 5-17 Network Address Types

Name	Storage Space	Description
cidr	7 or 19 bytes	IPv4 or IPv6 networks
inet	7 or 19 bytes	IPv4 or IPv6 hosts and networks
macaddr	6 bytes	MAC addresses

When sorting **inet** or **cidr** data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

cidr

The **cidr** type (Classless Inter-Domain Routing) holds an IPv4 or IPv6 network specification. The format for specifying networks is **address/y** where **address** is the network represented as an IPv4 or IPv6 address, and **y** is the number of bits in the netmask. If **y** is omitted, it is calculated using assumptions from the older classful network numbering system, except it will be at least large enough to include all of the octets written in the input.

- Example 1: Convert a value in the CIDR format to an IP address segment.

For example, **10.0.0.0/8** is converted into a 32-bit binary address **00001010.00000000.00000000.00000000**. **/8** indicates an 8-bit network ID. The first eight bits of the 32-bit binary address are fixed. The corresponding network segment is 00001010.00000000.00000000-00001010.11111111.11111111.11111111. **10.0.0.0/8** indicates that the subnet mask is 255.0.0.0 and the corresponding network segment is 10.0.0.0-10.255.255.255.

- Example 2: Convert an IP address segment to the CIDR format.

For the IP address segment **192.168.0.0-192.168.31.255**, the last two segments can be converted into a binary address **00000000.00000000-00011111.11111111**. The first 19 bits (8 x 2 + 3) are fixed. Therefore, the binary address is 192.168.0.0/19.

Table 5-18 cidr type input examples

cidr Input	cidr Output	abbrev (cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
10.1.2	10.1.2.0/24	10.1.2/24

cidr Input	cidr Output	abbrev (cidr)
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1

inet

The **inet** type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the "netmask"). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address.

The input format for this type is **address/y** where **address** is an IPv4 or IPv6 address and **y** is the number of bits in the netmask. If the **/y** portion is missing, the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the **/y** portion is suppressed if the netmask specifies a single host.

The essential difference between the **inet** and **cidr** data types is that **inet** accepts values with nonzero bits to the right of the netmask, whereas **cidr** does not.

macaddr

The **macaddr** type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in the following formats:

```
'08:00:2b:01:02:03'  
'08-00-2b-01-02-03'  
'08002b:010203'  
'08002b-010203'  
'0800.2b01.0203'  
'08002b010203'
```

These examples would all specify the same address. Upper and lower cases are accepted for the digits a through f. Output is always in the first of the forms shown.

5.11 Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store bit masks.

DWS supports two SQL bit types: **bit(n)** and **bit varying(n)**, where **n** is a positive integer.

The **bit** type data must match the length **n** exactly. It is an error to attempt to store shorter or longer bit strings. The **bit varying** data is of variable length up to

the maximum length **n**; longer strings will be rejected. Writing **bit** without a length is equivalent to **bit(1)**, while **bit varying** without a length specification means unlimited length.

NOTE

If one explicitly casts a bit-string value to **bit(n)**, it will be truncated or zero-padded on the right to be exactly **n** bits, without raising an error. Similarly, if one explicitly casts a bit-string value to **bit varying(n)**, the bits after **n** places will be truncated.

The following is an example of using the bit string type:

1. Create a sample table **bit_type_t1**.

```
CREATE TABLE bit_type_t1
(
  BT_COL1 INTEGER,
  BT_COL2 BIT(3),
  BT_COL3 BIT VARYING(5)
) DISTRIBUTE BY REPLICATION;
```

2. Insert data:

```
INSERT INTO bit_type_t1 VALUES(1, B'101', B'00');
```

An error is reported if an inserted string exceeds the length of this data type.

```
INSERT INTO bit_type_t1 VALUES(2, B'10', B'101');
ERROR: bit string length 2 does not match type bit(3)
CONTEXT: referenced column: bt_col2
```

3. Data is converted if it exceeds the length of this data type.

```
INSERT INTO bit_type_t1 VALUES(2, B'10'::bit(3), B'101');
```

4. View data:

```
SELECT * FROM bit_type_t1;
bt_col1 | bt_col2 | bt_col3
-----+-----+-----
1 | 101 | 00
2 | 100 | 101
(2 rows)
```

5.12 Text Search Types

DWS offers **tsvector** and **tsquery** data types to support full text search. The **tsvector** type represents a document in a form optimized for text search. The **tsquery** type similarly represents a text query.

tsvector

The **tsvector** type represents a retrieval unit, usually a row of text fields in a database table or a combination of these fields.

A **tsvector** value is a sorted list of distinct lexemes, which are words that have been normalized to merge different variants of the same word. Sorting and duplicate-elimination are done automatically during input.

The **to_tsvector** function is used to parse and normalize a document string.

Use **tsvector** to segment a string into lexemes by space. The lexemes are sorted by letter and length. The following is an example:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
tsvector
-----
```

```
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
(1 row)
```

To represent lexemes containing whitespace or punctuation, surround them with quotes:

```
SELECT $$the lexeme ' ' contains spaces$$::tsvector;
          tsvector
-----
' ' contains 'lexeme' 'spaces' 'the'
(1 row)
```

If a string is enclosed in common single quotation marks, the single quotation marks (') and backslashes (\) embedded in the string must be double-written for escape.

```
SELECT $$the lexeme 'Joe"s' contains a quote$$::tsvector;
          tsvector
-----
'Joe"s' 'a' 'contains' 'lexeme' 'quote' 'the'
(1 row)
```

Optionally, integer positions can be attached to lexemes:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
          tsvector
-----
'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
(1 row)
```

A position normally indicates the source word's location in the document. Positional information can be used for proximity ranking. Position values range from 1 to 16383. The default maximum value is **16383**. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a weight, which can be **A**, **B**, **C**, or **D**. **D** is the default weight. It is not displayed in the output:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
-----
'a':1A 'cat':5 'fat':2B,4C
(1 row)
```

Weights usually are used to reflect document structure, for example, by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

The following is an example of the standard usage of the tsvector type:

```
SELECT 'The Fat Rats'::tsvector;
          tsvector
-----
'Fat' 'Rats' 'The'
(1 row)
```

For most English-text-searching applications the above words would be considered non-normalized, which should usually be passed through **to_tsvector** to normalize the words appropriately for searching:

```
SELECT to_tsvector('english', 'The Fat Rats');
          to_tsvector
-----
'fat':2 'rat':3
(1 row)
```

tsquery

The **tsquery** type represents a retrieval condition. A **tsquery** value stores lexemes that are to be searched for, and combines them honoring the **Boolean** operators **&** (**AND**), **|** (**OR**), and **!** (**NOT**). Parentheses can be used to enforce grouping of the operators. If there is no parenthesis, (**NOT**) has the highest priority, followed by **&**(**AND**), and finally **|** (**OR**). The **to_tsquery** and **plainto_tsquery** functions will normalize lexemes before the lexemes are converted to the **tsquery** type.

```
SELECT 'fat & rat'::tsquery;
      tsquery
-----
'fat' & 'rat'
(1 row)

SELECT 'fat & (rat | cat)'::tsquery;
      tsquery
-----
'fat' & ( 'rat' | 'cat' )
(1 row)

SELECT 'fat & rat & ! cat'::tsquery;
      tsquery
-----
'fat' & 'rat' & '!cat'
(1 row)
```

Lexemes in a **tsquery** can be labeled with one or more weight letters, which match only **tsvector** lexemes with matching weights.

```
SELECT 'fat:ab & cat'::tsquery;
      tsquery
-----
'fat':AB & 'cat'
(1 row)
```

Also, lexemes in a **tsquery** can be labeled with ***** to specify prefix matching. The following query will match any word in a **tsvector** that begins with "super".

```
SELECT 'super:*'::tsquery;
      tsquery
-----
'super':*
(1 row)
```

Note that prefix matches are first checked by the text search analyzer. For example, the stem extracted from **postgres** is **postgr**, which matches **postgraduate**. The following result is true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' ) AS RESULT;
      result
-----
t
(1 row)
SELECT to_tsquery('postgres:*');
      to_tsquery
-----
'postgr':*
(1 row)
```

The **to_tsquery** function normalizes words before converting them to the **tsquery** type. **'Fat:ab & Cats'** is normalized to the **tsquery** type as follows:

```
SELECT to_tsquery('Fat:ab & Cats');
      to_tsquery
-----
```

```
'fat':AB & 'cat'  
(1 row)
```

5.13 UUID Type

Universally Unique Identifier (UUID) is a 128-bit identifier used in the computer system to identify information.

All elements in a distributed system can be uniquely identified by an UUID, without having to be identified in the central control end. In many application scenarios, an ID is required to identify only one object. A common example is the ID column of a database table. Another example is the front end UI libraries, because they usually need to dynamically create various UI elements that require unique IDs.

For details about UUID functions and UUID application examples supported by DWS, see [UUID Functions](#).

UUID Formats

UUIDs are standardized as part of the Distributed Computing Environment (DCE) in RFC 4122 released by the Internet Engineering Task Force (IETF) of the Open Software Foundation (OSF). A standard UUID consists of 36 characters, including 32 hexadecimal digits and four hyphens (-). The format is 8-4-4-4-12. An example of a standard UUID is as follows:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

DWS also accepts the following alternative forms for input: use of upper-case letters and digits, the standard format surrounded by braces, omitting some or all hyphens, and adding a hyphen after any group of four digits. Examples

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11  
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}  
a0eebc999c0b4ef8bb6d6bb9bd380a11  
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
```

UUID Composition

A UUID uses time and space information to ensure global uniqueness. A UUID consists of a timestamp, a clock sequence, and a node ID (MAC address). However, multiple nodes in a distributed DWS cluster may be deployed on the same machine and have the same MAC address. As a result, UUID conflicts may occur. Therefore, DWS replaces the last 48 bits of the MAC address with the sequence number of the CN or DN that generates the UUID and the current thread ID to ensure that the UUID is globally unique in the distributed cluster.

5.14 JSON Types

JavaScript Object Notation (JSON) data types are used for storing JSON data.

It can be an independent scalar, an array, or a key-value object. An array and an object can be called a container.

1. Scalar: a number, Boolean, string, or null

2. Array: defined in a pair of square brackets (`[]`), in which elements can be of any JSON data type, and are not necessarily of the same type.
3. Object: defined in a pair of braces (`{}`), in which objects are stored in the format of **key:value**. Each key must be a string enclosed in double quotation marks (`""`), and its value can be of any JSON data type. In case of duplicate keys, the last key-value pair will be used.

DWS supports the JSON and JSONB data types to store JSON data. Where:

- `json` copies all entered strings and parses them when they are used. During this process, the entered spaces, duplicate keys, and sequence are retained.
- `jsonb` parses the binary data of the input. During parsing, `jsonb` deletes semantic-irrelevant details and duplicate keys, and sorts key values, so that the data does not to be parsed again during use.

Both JSON and JSONB are of JSON data type, and the same strings can be entered as input. The main difference between JSON and JSONB is the efficiency. Because `json` data is an exact copy of the input text, the data must be parsed on every execution. In contrast, `jsonb` data is stored in a decomposed binary form and can be processed faster, though this makes it slightly slower to input due to the conversion mechanism. In addition, because the JSONB data form is normalized, it supports more operations, for example, comparing sizes according to a specific rule. JSONB also supports indexing, which is a significant advantage.

Input Format

The input must be a JSON-compliant string, which is enclosed in single quotation marks (`'`).

Null (`null-json`): Only null is supported, and all letters are in lowercase.

```
SELECT 'null':json; -- suc
SELECT 'NULL':jsonb; -- err
```

Number (`num-json`): The value can be a positive or negative integer, decimal fraction, or 0. The scientific notation is supported.

```
SELECT '1':json;
SELECT '-1.5':json;
SELECT '-1.5e-5':jsonb, '-1.5e+2':jsonb;
SELECT '001':json, '+15':json, 'NaN':json; -- Redundant leading zeros, plus signs (+), NaN, and infinity are not supported.
```

Boolean (`bool-json`): The value can only be **true** or **false** in lowercase.

```
SELECT 'true':json;
SELECT 'false':jsonb;
```

String (`str-json`): The value must be a string enclosed in double quotation marks (`""`).

```
SELECT '"a":json;
SELECT '"abc":jsonb;
```

Array (`array-json`): Arrays are enclosed in square brackets (`[]`). Elements in the array can be any valid JSON data, and are unnecessarily of the same type.

```
SELECT '[1, 2, "foo", null]':json;
SELECT '[]':json;
SELECT '[1, 2, "foo", null, [], {}]':jsonb;
```

Object (object-json): The value is enclosed in braces ({}). The key must be a JSON-compliant string, and the value can be any valid JSON string.

```
SELECT '{}':json;  
SELECT '{"a": 1, "b": {"a": 2, "b": null}}':json;  
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}':jsonb;
```

CAUTION

- Note that '**null::json**' and **null::jsonb** are different. The difference is similar to that between the strings **str=""** and **str=null**.
- For numbers, when scientific notation is used, JSONB expands them, while JSON stores an exact copy of the input text.

JSONB Advanced Features

The main difference between JSON and JSONB is the storage mode. JSONB stores parsed binary data, which reflects the JSON hierarchy and facilitates direct access. Therefore, JSONB has more advanced features than JSON.

Normalizes formats

- After the input **object-json** string is parsed into JSONB binary, semantically irrelevant details are naturally discarded, for example, spaces:

```
SELECT ' [1, " a ", {"a" :1  }] ':jsonb;  
jsonb  
-----  
[1, " a ", {"a": 1}]  
(1 row)
```

- For **object-json**, duplicate key-values are deleted and only the last key-value is retained. An example is as follows:

```
SELECT '{"a" : 1, "a" : 2}':jsonb;  
jsonb  
-----  
{"a": 2}  
(1 row)
```

- For **object-json**, key-values will be re-sorted. The sorting rule is as follows: 1. Longer key-values are sorted last. 2. If the key-values are of the same length, the key-values with a larger ASCII code are sorted last. An example is as follows:

```
SELECT '{"aa" : 1, "b" : 2, "a" : 3}':jsonb;  
jsonb  
-----  
{"a": 3, "b": 2, "aa": 1}  
(1 row)
```

Compares sizes

Format normalization ensures that only one form of JSONB data exists in the same semantics. Therefore, sizes can be compared according to a specific rule.

- First, type comparison: **object-jsonb** > **array-jsonb** > **bool-jsonb** > **num-jsonb** > **str-jsonb** > **null-jsonb**
- Content is compared if the data type is the same:
 - str-json**: The default text sorting rule of the database is used for comparison. A positive value indicates greater than, a negative value indicates less than, and **0** indicates equal.

- **num-json**: numeric comparison
- **bool-json**: **true** > **false**
- **array-jsonb**: long elements > short elements. If the lengths are the same, compare each element in sequence.
- **object-jsonb**: long key-value pairs > short key-value pairs. If the lengths are the same, compare each key-value pair in sequence, first the key and then the value.

CAUTION

For comparison within the **object-jsonb** type, the final result after format sorting is used for comparison. Therefore, the comparison result may not be so intuitive as direct input.

Creates an index

The JSONB type in row-store tables supports both B-tree and GIN indexes, while the JSONB type in column-store tables only supports B-tree indexes.

If the entire JSONB column uses a B-tree index, the following operators can be used: =, <, <=, >, and >=.

Example: Create the table **test** and insert data into it.

```
CREATE TABLE test(id bigserial, data JSONB, PRIMARY KEY (id));
INSERT INTO test(data) VALUES('{\"name\":\"Jack\", \"age\":10, \"nick_name\":[\"Jacky\",\"baobao\"], \"phone_list\":[\"1111\",\"2222\"]}');:jsonb);
```

- Create a B-tree index.

```
CREATE INDEX idx_test_data_age ON test USING btree(((data->>'age')::int));
```

Use the B-tree index to query data where "age>1".

```
SELECT * FROM test WHERE (data->>'age')::int>1;
```
- Create a GIN index.

```
CREATE INDEX idx_test_data ON test USING gin (data);
```

Use the GIN index to check whether there are top-level keywords.

```
SELECT * FROM test WHERE data ? 'id';
SELECT * FROM test WHERE data ?| array['id','name'];
```
- Use the GIN index to check whether there are non-top-level keywords.

```
CREATE INDEX idx_test_data_nick_name ON test USING gin((data->'nick_name'));
SELECT * FROM test WHERE data->'nick_name' ? 'Jacky';
```
- Use @> to check whether JSON contains nested JSON objects.

```
SELECT * FROM test WHERE data @> '{\"age\":10, \"nick_name\":[\"Jacky\"]}';
```

Includes elements in a JSON

An important capability of JSONB is to query whether a JSON contains some elements or whether some elements exist in a JSON.

- Simple scalar/original values contain only the same value.

```
SELECT '\"foo\"'::jsonb @> '\"foo\"'::jsonb;
```
- The array on the left contains the string on the right.

```
SELECT '[1, \"aa\", 3]'::jsonb ? 'aa';
```
- The array on the left contains all elements of the array on the right. The sequence and repetition are not important.

```
SELECT '[1, 2, 3]'::jsonb @> '[1, 3, 1]'::jsonb;
```


- The **object-json** on the left contains all key-values of the **object-json** on the right.
`SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}::jsonb @> '{"version": 9.4}::jsonb;`
- The array on the left does not contain all elements in the array on the right, because the three elements in the array on the left are **1**, **2**, and **[1,3]**, and the elements on the right are **1** and **3**.
`SELECT '[1, 2, [1, 3]]::jsonb @> '[1, 3]::jsonb; -- false`
- The array on the right does not contain all elements in the array on the left in the following example:
`SELECT '{"foo": {"bar": "baz"}}::jsonb @> '{"bar": "baz"}::jsonb; -- false`

5.15 RoaringBitmap

In DWS 8.1.3 and later, you can use the RoaringBitmap data type to store bitmap datasets.

The RoaringBitmap data type supports row-store and column-store tables.

Table 5-19 RoaringBitmap

Name	Storage Size	Description	Range
RoaringBitmap	32 bytes	Stores bitmap data sets.	-2,147,483,648~2,147,483,647

Example: Create a table with the RoaringBitmap data type.

```
CREATE TABLE r_row (a int ,b text, c roaringbitmap);  
CREATE TABLE r_col (a int ,b text, c roaringbitmap) with (orientation=column);
```

5.16 HLL Data Types

HyperLoglog (HLL) is an approximation algorithm for efficiently counting the number of distinct values in a data set. It features faster computing and lower space usage. You only need to store HLL data structures, instead of data sets. When new data is added to a data set, make hash calculation on the data and insert the result to an HLL. Then, you can obtain the final result based on the HLL.

Table 5-20 compares HLL with other algorithms.

Table 5-20 Comparison between HLL and other algorithms

Item	Sorting Algorithm	Hash Algorithm	HLL
Time complexity	O(nlogn)	O(n)	O(n)
Space complexity	O(n)	O(n)	1280 bytes

Item	Sorting Algorithm	Hash Algorithm	HLL
Error rate	0	0	≈2%
Storage space requirement	Size of raw data	Size of raw data	1280 bytes

HLL has advantages over others in the computing speed and storage space requirement. In terms of time complexity, the sorting algorithm needs $O(n\log n)$ time for sorting, and the hash algorithm and HLL need $O(n)$ time for full table scanning. In terms of storage space requirements, the sorting algorithm and hash algorithm need to store raw data before collecting statistics, whereas the HLL algorithm needs to store only the HLL data structures rather than the raw data, and thereby occupying a fixed space of only 1280 bytes.

NOTICE

- In default specifications, the maximum number of distinct values is 1.6e plus 12, and the maximum error rate is only 2.3%. If a calculation result exceeds the maximum number, the error rate of the calculation result will increase, or the calculation will fail and an error will be reported.
- When using this feature for the first time, you need to evaluate the distinct values of the service, properly select configuration parameters, and perform verification to ensure that the accuracy meets requirements.
 - When default parameter configuration is used, the calculated number of distinct values is 1.6e plus 12. If the calculated result is **NaN**, you need to adjust **log2m** and **regwidth** to accommodate more distinct values.
 - The hash algorithm has an extremely low probability of collision. However, you are still advised to select 2 or 3 hash seeds for verification when using the hash algorithm for the first time. If there is only a small difference between the distinct values, you can select any one of the seeds as the hash seed.

[Table 5-21](#) describes main HLL data structures.

Table 5-21 Main HLL data structures

Data Type	Description
hll	Its size is always 1280 bytes, which can be directly used to calculate the number of distinct values.

Application Scenarios of HLL

- Using the HLL Data Type
 - a. Create an HLL table and insert an empty HLL into the table.

```
CREATE TABLE helloworld (id integer, set hll);
INSERT INTO helloworld(id, set) VALUES (1, hll_empty());
```

- b. Add an integer that has gone through hash calculation into to the HLL.

```
UPDATE helloworld SET set = hll_add(set, hll_hash_integer(12345)) WHERE id = 1;
```

- c. Add a string that has gone through hash calculation into to the HLL.

```
UPDATE helloworld SET set = hll_add(set, hll_hash_text('hello world')) WHERE id = 1;
```

- d. Obtain the number of distinct values of the HLL.

```
SELECT hll_cardinality(set) FROM helloworld WHERE id = 1;
hll_cardinality
-----
                2
(1 row)
```

- Using HLL to Collect Website Visitor Statistics

- a. Create the original data table **facts** to record the time when a user accesses a website.

```
CREATE TABLE facts (
  date      date,
  user_id   integer
);
```

- b. Insert user visits data.

```
INSERT INTO facts VALUES ('2019-02-20', generate_series(1,100));
INSERT INTO facts VALUES ('2019-02-21', generate_series(1,200));
INSERT INTO facts VALUES ('2019-02-22', generate_series(1,300));
INSERT INTO facts VALUES ('2019-02-23', generate_series(1,400));
INSERT INTO facts VALUES ('2019-02-24', generate_series(1,500));
INSERT INTO facts VALUES ('2019-02-25', generate_series(1,600));
INSERT INTO facts VALUES ('2019-02-26', generate_series(1,700));
INSERT INTO facts VALUES ('2019-02-27', generate_series(1,800));
```

- c. Create another table and specify an HLL column: Group data by date and insert the data into the HLL:

```
CREATE TABLE daily_uniques (
  date      date UNIQUE,
  users     hll
);

INSERT INTO daily_uniques(date, users)
SELECT date, hll_add_agg(hll_hash_integer(user_id))
FROM facts
GROUP BY 1;
```

- d. Calculate the numbers of users visiting the website every day:

```
SELECT date, hll_cardinality(users) FROM daily_uniques ORDER BY date;
date      | hll_cardinality
-----+-----
2019-02-20 00:00:00 |          100
2019-02-21 00:00:00 | 203.813355588808
2019-02-22 00:00:00 | 308.048239950384
2019-02-23 00:00:00 | 410.529188080374
2019-02-24 00:00:00 | 513.263875705319
2019-02-25 00:00:00 | 609.271181107416
2019-02-26 00:00:00 | 702.941844662509
2019-02-27 00:00:00 | 792.249946595237
(8 rows)
```

- e. Calculate the number of users who had visited the website in the week from February 20, 2019 to February 26, 2019:

```
SELECT hll_cardinality(hll_union_agg(users)) FROM daily_uniques WHERE date >=
'2019-02-20'::date AND date <= '2019-02-26'::date;
hll_cardinality
-----
702.941844662509
(1 row)
```

- f. Calculate the number of users who had visited the website yesterday but have not visited the website today:

```
SELECT date, (#hll_union_agg(users) OVER two_days) - #users AS lost_uniques FROM
daily_uniques WINDOW two_days AS (ORDER BY date ASC ROWS 1
PRECEDING);
```

date	lost_uniques
2019-02-20 00:00:00	0
2019-02-21 00:00:00	0
2019-02-22 00:00:00	0
2019-02-23 00:00:00	0
2019-02-24 00:00:00	0
2019-02-25 00:00:00	0
2019-02-26 00:00:00	0
2019-02-27 00:00:00	0

(8 rows)

- HLL Format Errors

When inserting data into a column of the HLL type, ensure that the data meets the requirements of the HLL data structure. If the data does not meet the requirements after being parsed, an error will be reported.

For example, when **E\\1234'** is inserted, the data does not comply with the HLL data structure and cannot be parsed. As a result, an error is reported.

```
CREATE TABLE test(id integer, set hll);
INSERT INTO test VALUES(1, 'E\\1234');
ERROR: invalid input syntax for integer: "E\\1234"
```

5.17 Object Identifier Types

Object identifiers (OIDs) are used internally by DWS as primary keys for various system catalogs. OIDs are not added to user-created tables by the system. The **OID** type represents an object identifier.

The **OID** type is currently implemented as an unsigned four-byte integer. So, using a user-created table's **OID** column as a primary key is discouraged.

Table 5-22 Object identifier types

Name	Reference	Description	Examples
OID	-	Numeric object identifier	564182
CID	-	A command identifier. This is the data type of the system columns cmin and cmax . Command identifiers are 32-bit quantities.	-
XID	-	A transaction identifier. This is the data type of the system columns xmin and xmax . Transaction identifiers are also 32-bit quantities.	-

Name	Reference	Description	Examples
TID	-	A row identifier. This is the data type of the system column ctid . A row ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.	-
REGCONFIG	pg_ts_config	Text search configuration	english
REGDICTIONARY	pg_ts_dict	Text search dictionary	simple
REGOPER	pg_operator	Operator name	+
REGOPERATOR	pg_operator	Operator with argument types	*(integer,integer) or -(NONE,integer)
REGPROC	pg_proc	Indicates the name of the function.	sum
REGPROCEDURE	pg_proc	Function with argument types	sum(int4)
REGCLASS	pg_class	Relation name	pg_type
REGTYPE	pg_type	Data type name	integer

The **OID** type is used for a column in the database system catalog.

Example:

```
SELECT oid FROM pg_class WHERE relname = 'pg_type';
oid
-----
1247
(1 row)
```

The alias type for **OID** is **REGCLASS** which allows simplified search for **OID** values.

Example:

```
SELECT attrelid,attname,atttypid,attstattarget FROM pg_attribute WHERE attrelid = 'pg_type'::REGCLASS;
attrelid | attname | atttypid | attstattarget
-----+-----+-----+-----
1247 | xc_node_id | 23 | 0
1247 | tableoid | 26 | 0
1247 | cmax | 29 | 0
1247 | xmax | 28 | 0
1247 | cmin | 29 | 0
1247 | xmin | 28 | 0
1247 | oid | 26 | 0
1247 | ctid | 27 | 0
```

1247	typename	19	-1
1247	typnamespace	26	-1
1247	typowner	26	-1
1247	typplen	21	-1
1247	typbyval	16	-1
1247	typtype	18	-1
1247	typcategory	18	-1
1247	typispreferred	16	-1
1247	typisdefined	16	-1
1247	typdelim	18	-1
1247	typrelid	26	-1
1247	typelem	26	-1
1247	typarray	26	-1
1247	typinput	24	-1
1247	typoutput	24	-1
1247	typreceive	24	-1
1247	typsend	24	-1
1247	typmodin	24	-1
1247	typmodout	24	-1
1247	typanalyze	24	-1
1247	typalign	18	-1
1247	typstorage	18	-1
1247	typnotnull	16	-1
1247	typbasetype	26	-1
1247	typtypmod	23	-1
1247	typndims	23	-1
1247	typcollation	26	-1
1247	typdefaultbin	194	-1
1247	typdefault	25	-1
1247	typacl	1034	-1

(38 rows)

5.18 Pseudo-Types

DWS has a number of special-purpose entries that are collectively called pseudo-types. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type.

Pseudotypes are highly beneficial in functions that require more complex input and output than standard SQL data types. [Table 5-23](#) lists all pseudo-types.

Table 5-23 Pseudo-Types

Name	Description
any	Indicates that a function accepts any input data type.
anyelement	Indicates that a function accepts any data type.
anyarray	Indicates that a function accepts any array data type.
anynonarray	Indicates that a function accepts any non-array data type.
anyenum	Indicates that a function accepts any enum data type.
anyrange	Indicates that a function accepts any range data type.
cstring	Indicates that a function accepts or returns a null-terminated C string.
internal	Indicates that a function accepts or returns a server-internal data type.

Name	Description
language_handler	Indicates that a procedural language call handler is declared to return language_handler .
fdw_handler	Indicates that a foreign-data wrapper handler is declared to return fdw_handler .
record	Identifies a function returning an unspecified row type.
trigger	Indicates that a trigger function is declared to return trigger .
void	Indicates that a function returns no value.
opaque	Indicates an obsolete type name that formerly served all the above purposes.

Functions coded in C (whether built in or dynamically loaded) can be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present the procedural languages all forbid use of a pseudo-type as argument type, and allow only **void** and **record** as a result type. Some also support polymorphic functions using the **anyelement**, **anyarray**, **anynonarray**, **anyenum**, and **anyrange** types.

The **internal** pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct call in an SQL query. If a function has at least one **internal**-type argument, it cannot be called from SQL. You are not advised to create any function that is declared to return **internal** unless the function has at least one **internal** argument.

Example:

Create or replace the showall() function:

```
CREATE OR REPLACE FUNCTION showall() RETURNS SETOF record
AS $$ SELECT count(*) from tpeds.store_sales where ss_customer_sk = 9692; $$
LANGUAGE SQL;
```

Invoke the showall() function:

```
SELECT showall();
showall
-----
(35)
(1 row)
```

Delete a function.

```
DROP FUNCTION showall();
```

5.19 Range Types

A range type is a data type that represents a range of values for certain element types (subtypes of ranges). For example, the range of a timestamp may be used to express a time range in which a conference room is reserved. In this case, the data type is **tsrange** (short for timestamp range), and **timestamp** is the subtype. The subtype must have an overall order so that the element value can be clearly specified as within, before, or after a range.

Range types can express multiple element values in a single range value and can clearly express concepts such as range overlapping. It helps to schedule the timing and duration of an event, and can also be utilized to set price or measurement parameters for equipment.

Built-in Range Types

DWS has the following built-in range types:

- **int4range**: integer range.
- **int8range**: bigint range.
- **numrange**: numeric range.
- **tsrange**: range of timestamp without the time zone.
- **tstzrange**: range of timestamp with the time zone
- **daterange**: date range.

NOTE

DWS does not support user-defined range types.

Create the reservation table and insert data into the table. The **during** column is of the **tsrange** type.

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');
```

Query whether the range contains specific values:

```
SELECT int4range(10, 20) @> 3;
?column?
-----
f
(1 row)
```

Query whether the range overlaps with another value:

```
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
?column?
-----
t
(1 row)
```

Obtain the upper bound of the range:

```
SELECT upper(int8range(15, 25));
upper
-----
25
(1 row)
```


Calculate the intersection with another range:

```
SELECT int4range(10, 20) * int4range(15, 25);
?column?
-----
[15,20)
(1 row)
```

Query whether the range is empty:

```
SELECT isempty(numrange(1, 5));
isempty
-----
f
(1 row)
```

Inclusion and Exclusion of Bounds

Each non-empty range has two bounds: a lower bound and an upper bound. All values between the two bounds are included in the range. Inclusion of bounds means that boundary values are also included in the range, while exclusion of bounds means that boundary values are not included in the range.

Inclusion of lower bound is represented by "[", and exclusion of lower bound is represented by "(". Similarly, inclusion of upper bound is represented by "]", and exclusion of upper bound is represented by ")".

The [lower_inc\(anyrange\)](#) and [lower_inc\(anyrange\)](#) functions are used to test the upper and lower bounds of a range, respectively.

Infinite (Unbounded) Range

When the lower bound of a range is unbounded, it means that all values less than the upper bound are included in the range, for example, `(,3]`. Similarly, when the upper bound of a range is unbounded, all values greater than the lower bound are included in the range. When both the upper and lower bounds are unbounded, all values of the element type are considered within the range. The missing bounds are automatically converted to exclusions, for example, `[,]` is converted to `(,)`. You can think of these missing values as positive infinity or negative infinity, but they are special range type values and are considered to be positive and negative infinity values that go beyond any range element type.

Element types with the infinity values can be used as explicit bound values. For example, in the timestamp range, `[today, infinity)` does not include a special timestamp value **infinity**.

The functions [lower_inf\(anyrange\)](#) and [upper_inf\(anyrange\)](#) are used to test whether a range has infinite upper bound or lower bound, respectively.

Input/Output Values of a Range

Range values must follow one of the following patterns:

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

Parentheses () or square brackets [] indicate whether the upper and lower bounds are excluded or included. The last format is **empty**, which represents an empty range (a range that does not contain values).

The value of *lower-bound* can be a valid string of the subtype, or left empty, which indicates that there is no lower bound. The value of *upper-bound* can be a valid string of the subtype, or null, which indicates that there is no upper bound.

Each bound value can be referenced using the quotation marks ("""). This is necessary if the bound value contains parentheses (), square brackets [], commas (,), quotation marks ("""), or backslashes (\), because otherwise those characters will be considered part of the range syntax. To put a quotation mark or backslash in a bound value, put a backslash in front of it (and a pair of double quotation marks in a referenced bound value represents one quotation mark character, which is similar to the single quotation mark rule in SQL strings). In addition, you can avoid referencing and use backslash escapes to preventing data characters from being used as part of the return syntax. If you want to input a bound value that is an empty string, write "", indicating infinite bounds.

Spaces are allowed before and after a range value, but any space between parentheses() or square brackets[] is used as part of the upper or lower bound value (depending on the element type, the space may or may not represent a value).

Examples

Query all values between 3 and 7, including 3 and excluding 7:

```
SELECT '[3,7)>::int4range;
int4range
-----
[3,7)
(1 row)
```

Query all values between 3 and 7, excluding 3 and 7:

```
SELECT '(3,7)>::int4range;
int4range
-----
[4,7)
(1 row)
```

Query the value 4:

```
SELECT '[4,4]>::int4range;
int4range
-----
[4,5)
(1 row)
```

Query a range containing no values (normalized to **empty**):

```
SELECT '[4,4)>::int4range;
int4range
-----
empty
(1 row)
```

Constructing Range

Each range type has a constructor function with the same name. Using constructor functions is more convenient than writing a range literal constant

because it avoids extra references to bound values. Constructor functions accept two or three parameters. Two parameters form a range in the standard form, where the lower bound is included and the upper bound is excluded. If a range contains three parameters, the third parameter specifies the range exclusion/inclusion type. The third parameter must be one of the following character strings: (), ([,], or []). The following is an example:

The complete form is: lower bound, upper bound, and textual parameters indicating the inclusion/exclusion of bounds.

```
SELECT numrange(1.0, 14.0, '[]');
       numrange
-----
(1.0,14.0]
(1 row)
```

If the third parameter is ignored, the range will be deemed as '[]'.

```
SELECT numrange(1.0, 14.0);
       numrange
-----
[1.0,14.0)
(1 row)
```

Although '[' is specified here, it is converted to the standard form in the return result because `int8range` is a type of **Discrete Ranges**:

```
SELECT int8range(1, 14, '[');
       int8range
-----
[2,15)
(1 row)
```

NULL indicates that the range is unbounded.

```
SELECT numrange(NULL, 2.2);
       numrange
-----
(,2.2)
(1 row)
```

Discrete Ranges

A discrete range is a range in which its elements have a clearly defined "step", such as integer or date. When there is no valid value between two elements, they are adjacent.

Each element value in a discrete range has a next value and a previous value. You can use the next or previous value to convert between inclusion and exclusion. For example, in an integer range, [4,8] and (3,9) represent the same set of values, but this is not the case for ranges other than numeric ranges.

A discrete range should have a canonical function for identifying the step in the range. The normalization function can convert the equivalents of the range type to expressions of the same meanings, which are consistent with the inclusion or exclusion bounds. If no normalization function is specified, ranges with different formats are always considered unequal, even if they actually express the same set of values.

The built-in range types `int4range`, `int8range`, and `daterange` use the normalized form, which includes the lower bound and excludes the upper bound, that is, []. However, user-defined ranges types can use other conventions.

User-defined Range Types

Users can define range types. To create a range type subtype float8, execute the following commands:

```
CREATE TYPE floatrange AS RANGE (  
    subtype = float8,  
    subtype_diff = float8mi  
);  
  
SELECT '[1.234, 5.678]':floatrange;
```

Indexes

You can create GiST indexes for table columns of the range type. Example:

```
CREATE TABLE reservation (room int, during tsrange);  
CREATE INDEX reservation_idx ON reservation USING gist (during);
```

GiST indexes can accelerate queries involving the following range operators: =, &&, <@, @>, <>, -|-, &<, and &. >. For details, see [Range Operators](#).

In addition, you can also create B-tree indexes on table columns of the range type. For these index types, basically the only useful range operation is equivalence. B-tree indexes for range types are primarily used to enable sorting within a query. They are not actually created.

5.20 Composite Types

A composite type represents the structure of a row or record, which is essentially a list of field names and their data types. DWS allows table columns to be declared as composite types. A composite type is essentially the same as the row type of a table. However, using [CREATE TYPE](#) avoids the need to create an actual table when only a type needs to be defined. A stand-alone composite type is useful as the parameter or return type of a function.

Declaration of Composite Types

DWS allows users to use [CREATE TYPE](#) to define composite types.

```
CREATE TYPE complex AS (  
    r    double precision,  
    i    double precision  
);  
  
CREATE TYPE inventory_item AS (  
    name      text,  
    supplier_id integer,  
    price     numeric  
);
```

After defining a composite type, you can create a table or function.

```
CREATE TABLE on_hand (  
    item    inventory_item,  
    count   integer  
);  
  
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);  
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric  
AS 'SELECT $1.price * $2' LANGUAGE SQL;
```

```
SELECT price_extension(item, 10) FROM on_hand;
```

Constructing a Compound Value

To write a composite value as a literal constant, enclose the field value in parentheses and separate them with commas. You can add double quotation marks to any field value. This is mandatory if the field value contains commas or parentheses. The general format of a compound constant is as follows:

```
'( val1 , val2 , ... )'
```

'("fuzzy dice",42,1.99)' is a valid value of the **inventory_item** type.

To set a field to NULL, left the position empty. If you want a field to be an empty string, use quotation marks. In the following example, the first column is a non-null empty string, and the third column is NULL.

```
'( "",42, )'
```

ROW expressions can also be used to build composite values. Example:

```
ROW('fuzzy dice', 42, 1.99)  
ROW("", 42, NULL)
```

Access Composite Types

To access a field in a composite column, you can write a dot and the name of the field, just like selecting a field from a table. To avoid confusion, you must use parentheses to distinguish it. For example, select some fields from table **on_hand**:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;  
ERROR: missing FROM-clause entry for table "item"
```

This writing is confused with selecting fields from a table because **item** is regarded as a table name instead of a column name of **on_hand**. It must be written as follows:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

If you need to use the table name (for example, in a multi-table query):

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Objects with parentheses are now correctly interpreted as references to column **item**, from which fields can be selected.

Similar syntax applies when a field is selected from a compound value. For example, to select a field from the result of a function that returns a compound value, write as follows:

```
SELECT (my_func(...)).field FROM ...
```

If there are no additional parentheses, this generates a syntax error.

5.21 Data Types Supported by Column-Store Tables

Table 5-24 lists the data types supported by column-store tables. Other data types that are not listed are not supported currently.

Table 5-24 Data Types Supported by Column-Store Tables

Category	Data Type	Description	Length
Numeric data	smallint	Small integer, also called INT2	2
	integer	Typical choice for integer, also called INT4	4
	bigint	Big integer, also called INT8	8
	decimal	Arbitrary precision type	Variable length
	numeric	Arbitrary precision type	Variable length
	real	Single-precision floating point	4
	double precision	Double-precision floating point	8
	smallserial	Two-byte auto-incrementing integer	2
	serial	Four-byte auto-incrementing integer	4
	bigserial	Eight-byte auto-incrementing integer	8
Money	money	Currency amount	8
Character data	character varying(n), varchar(n)	Variable-length string	Variable length
	character(n), char(n)	Fixed-length string	n
	character, char	Single-byte internal type	1
	text	Variable-length string	Variable length
	nvarchar2	Variable-length string	Variable length
	clob	Big text object	Variable length
Date/Time	timestamp with time zone	Date and time (with time zone)	8
	timestamp without time zone	Date and time	8

Category	Data Type	Description	Length
	date	Date and time (Oracle compatibility mode); date (other compatibility modes)	When using Oracle compatibility mode, the storage space is 8 bytes, whereas in other compatibility modes, it is 4 bytes.
	time without time zone	Time within one day	8
	time with time zone	Time within one day (with time zone)	12
	interval	Time interval	16
JSON	json	-	-
	jsonb	-	-
RoaringBitmap	-	Efficient bitmap, which supports the bitmap calculation of datasets of the int type.	Variable length

5.22 XML

XML data type stores Extensible Markup Language (XML) formatted data. XML data can also be stored as text, but the advantage of the XML data type is that it checks whether each stored value is a well-formed XML value. XML can store well-formed documents and content fragments defined by XML standards. A content fragment can have multiple top-level elements or character nodes.

For functions that support the XML data type, see [XML Functions](#).

Configuring XML Parameters

The syntax is as follows:

```
SET XML OPTION { DOCUMENT | CONTENT };  
SET xmloption TO { DOCUMENT | CONTENT };
```

If a string value is not converted to XML using the **XMLPARSE** or **XMLSERIALIZE** function, the **XML OPTION** session parameter determines it is **DOCUMENT** or **CONTENT**.

The default value is **CONTENT**, indicating that all types of XML data are allowed.

Example:

```
SET XML OPTION DOCUMENT;  
SET  
SET xmloption TO DOCUMENT;  
SET
```

Configuring Binary Data Encoding Format

Syntax:

```
SET xmlbinary TO { base64 | hex};
```

Example:

```
SET xmlbinary TO base64;  
SET  
  
SELECT xmlelement(name foo, bytea 'bar');  
xmlelement  
-----  
<foo>YmFy</foo>  
(1 row)  
  
SET xmlbinary TO hex;  
SET  
  
SELECT xmlelement(name foo, bytea 'bar');  
xmlelement  
-----  
<foo>626172</foo>  
(1 row)
```

Accessing XML Value

The XML data type is special, and it does not provide any comparison operators, because there is no general comparison algorithm for XML data, so you cannot retrieve data rows by comparing an XML value with a search value. An XML data entry is typically accompanied by an ID for retrieving. Alternatively, you can convert XML values into character strings. However, this is not widely applicable to common scenarios of XML value comparison.

6 Constant and Macro

Table 6-1 lists the constants and macros that can be used in DWS.

Table 6-1 Constants and macros

Parameter	Description	Example SQL Statements
CURRENT_CATALOG	Specifies the current database.	SELECT CURRENT_CATALOG;
CURRENT_ROLE	Current role	SELECT CURRENT_ROLE;
CURRENT_SCHEMA	Current database model	SELECT CURRENT_SCHEMA;
CURRENT_USER	Current user	SELECT CURRENT_USER;
LOCALTIMESTAMP	Current session time (without time zone)	SELECT LOCALTIMESTAMP;
NULL	This parameter is left blank.	-
SESSION_USER	Current system user	SELECT SESSION_USER;
SYSDATE	Current system date	SELECT SYSDATE; or SELECT now()::DATE;
USER	Current user, which is the same as the value of CURRENT_USER .	SELECT USER;

7 Functions and Operators

7.1 Character Processing Functions and Operators

The character processing functions and operators provided by DWS are used for string operations, such as truncating, concatenating, converting, and comparing string values.

bit_length(string)

Description: Returns the length of the string expression in bits (bytes x 8). Its result depends on the database encoding mode.

Return type: integer

Example:

```
SELECT bit_length('world');
bit_length
-----
      40
(1 row)
```

btrim(string text [, characters text])

Description: Removes the specified characters from both ends of the string.

Return type: text

Parameters:

- **string**: (mandatory) target string.
- **characters**: (optional) characters to be removed from both ends of the target string. If it is not specified, all whitespace characters are removed by default.

Example:

```
SELECT btrim('sring' , 'ing');
btrim
-----
sr
(1 row)
```

```
SELECT btrim(' sting ');
btrim
-----
sting
(1 row)
```

char_length(string) or character_length(string)

Description: Obtains the number of characters in a string.

Return type: integer

NOTE

For multi-byte character encoding, one character may occupy multiple bytes.

Example:

```
SELECT char_length('hello');
char_length
-----
5
(1 row)
```

instr(text,text,int,int)

Description: Obtains the position of a substring in the target string. If no match is found, 0 is returned.

- First text: (mandatory) target string.
- Second text: (mandatory) substring to be matched.
- First int: start position of the matching. If this parameter is not set, the matching starts from the beginning by default.
- Second int: which occurrence of the substring the function should return the position for. If this parameter is not set, the first occurrence is used by default.

Return type: integer

Example:

Obtain the position of the 2nd occurrence of **bcd** starting from position 2 in the main string **abcdabcdabcd**.

```
SELECT instr( 'abcdabcdabcd', 'bcd', 2, 2 );
instr
-----
6
(1 row)

SELECT instr( 'abcdabcdabcd', 'lmk', 2);
instr
-----
0
(1 row)
```

lengthb(text/bpchar)

Description: Returns the number of bytes of a specified string.

Return type: integer

Example:

```
SELECT lengthb('hello');
lengthb
-----
      5
(1 row)
```

 NOTE

- For a string containing newline characters, for example, a string consisting of a newline character and a space, the value of **length** and **lengthb** in DWS is 2.
- This function returns the number of bytes in a specified string. For multi-byte character encoding, one character may occupy multiple bytes.

left(str text, n int)

Description: Returns first **n** characters in the string.

- In the ORA- or TD-compatible mode, all but the last **|n|** characters are returned if **n** is negative.
- In the MySQL-compatible mode, an empty string is returned if **n** is negative.

Return type: text

Example:

```
SELECT left('database', 4);
left
-----
data
(1 row)
```

length(string bytea, encoding name)

Description: Obtains the number of characters in a string in the specified encoding format. In the specified encoding format, **string** must be valid.

Return type: integer

Example:

```
SELECT length('database', 'UTF8');
length
-----
      8
(1 row)
```

lpad(string text, length int [, fill text])

Description: Fills up the string to the specified length by appending the characters **fill** (a space by default). If the **string** is already longer than **length** then it is truncated (on the right).

Return type: text

Example:

```
SELECT lpad('data', 8, 'xy');
lpad
-----
xyxydata
(1 row)
```

octet_length(string)

Description: Obtains the number of bytes in a string. It is important for processing multi-byte characters (such as UTF-8 characters and special symbols) and can accurately obtain the storage size of a string.

Return type: integer

Example:

```
SELECT octet_length('data');
octet_length
-----
         4
(1 row)
```

overlay(string placing string FROM int [for int])

Description: Replaces the part of a string starting from a specified position with a specified string. It can flexibly replace substring.

Parameters:

FROM int: starts the replacement from the *n*th character of the first string.

for int: (optional) number of characters to be replaced in the first string. If this parameter is not specified, all characters starting from **from** are replaced by default.

Return type: text

Example: Replace the substring of the original string starting from position 3 with **xyz**. The result is **abxyzef**.

```
SELECT overlay('abcdef' placing 'xyz' FROM 3);
overlay
-----
abxyzf
(1 row)
```

In the string **hello**, replace three consecutive characters starting from the second character with **world**. The result is **hworldo**.

```
SELECT overlay('hello' placing 'world' from 2 for 3 );
overlay
-----
hworldo
(1 row)
```

position(substring in string)

Description: Returns the position of a specified substring in a string. If the string does not contain substrings, 0 is returned.

Return type: integer

Example:

```
SELECT position('ing' in 'string');
position
-----
         4
```

```
(1 row)

SELECT position('ing' in 'strin');
position
-----
      0
(1 row)
```

pg_client_encoding()

Description: Obtains the character encoding mode of the current client. If a character display problem occurs, you can check the client encoding to quickly locate the problem.

Return type: name

Example:

```
SELECT pg_client_encoding();
pg_client_encoding
-----
      UTF8
(1 row)
```

quote_ident(string text)

Description: Returns the given string suitably quoted to be used as an identifier in an SQL statement string (quotation marks are used as required). Quotes are added only if necessary (that is, if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.

Return type: text

Example:

```
SELECT quote_ident('hello world');
quote_ident
-----
"hello world"
(1 row)
```

quote_literal(string text)

Description: Returns the given string suitably quoted to be used as a string literal in an SQL statement string (quotation marks are used as required).

Return type: text

Example:

```
SELECT quote_literal('hello');
quote_literal
-----
'hello'
(1 row)
```

If command similar to the following exists, text will be escaped.

```
SELECT quote_literal(E'O\hello');
quote_literal
-----
'O"hello'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_literal('O\hello');
quote_literal
-----
E'O\\hello'
(1 row)
```

If the parameter is null, return **NULL**. If the parameter may be null, you are advised to use **quote_nullable**.

```
SELECT quote_literal(NULL);
quote_literal
-----
(1 row)
```

quote_literal(value anyelement)

Description: Converts the given value to text and then quotes it as a literal.

Return type: text

Example:

```
SELECT quote_literal(42.5);
quote_literal
-----
'42.5'
(1 row)
```

If command similar to the following exists, the given value will be escaped.

```
SELECT quote_literal(E'O\'42.5');
quote_literal
-----
'O"42.5'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_literal('O\42.5');
quote_literal
-----
E'O\\42.5'
(1 row)
```

quote_nullable(string text)

Description: Returns the given string suitably quoted to be used as a string literal in an SQL statement string (quotation marks are used as required).

Return type: text

Example:

```
SELECT quote_nullable('hello');
quote_nullable
-----
'hello'
(1 row)
```

If command similar to the following exists, text will be escaped.

```
SELECT quote_nullable(E'O\'hello');
quote_nullable
```

```
-----  
'O'hello'  
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_nullable('O\hello');  
quote_nullable  
-----  
E'O\\hello'  
(1 row)
```

If the parameter is null, return **NULL**.

```
SELECT quote_nullable(NULL);  
quote_nullable  
-----  
NULL  
(1 row)
```

quote_nullable(value anyelement)

Description: Converts the given value to text and then quotes it as a literal.

Return type: text

Example:

```
SELECT quote_nullable(42.5);  
quote_nullable  
-----  
'42.5'  
(1 row)
```

If command similar to the following exists, the given value will be escaped.

```
SELECT quote_nullable(E'O\'42.5');  
quote_nullable  
-----  
'O"42.5'  
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_nullable('O\42.5');  
quote_nullable  
-----  
E'O\\42.5'  
(1 row)
```

If the parameter is null, return **NULL**.

```
SELECT quote_nullable(NULL);  
quote_nullable  
-----  
NULL  
(1 row)
```

substring(string [from int] [for int])

Description: Extracts a substring. **from int** indicates the start position of the truncation. **for int** indicates the number of characters truncated.

Return type: text

Example:

Extract three characters starting from the second character **h** in the string **Thomas**, that is, **hom**.

```
SELECT substr('Thomas' from 2 for 3);
substr
-----
hom
(1 row)
```

substr(string from *pattern*)

Description: Extracts substring matching POSIX regular expression. It returns the text that matches the pattern. If no match record is found, a null value is returned.

Return type: text

Example:

```
SELECT substr('Thomas' from '...$');
substr
-----
mas
(1 row)
SELECT substr('foobar' from 'o(.)b');
substr
-----
o
(1 row)
SELECT substr('foobar' from '(o(.)b)');
substr
-----
oob
(1 row)
```

NOTE

If the POSIX pattern contains any parentheses, the portion of the text that matched the first parenthesized sub-expression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception.

substr(string from *pattern* for *escape*)

Description: Extracts substring matching SQL regular expression. The specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern that should be returned on success, the pattern must contain two occurrences of the escape character followed by a double quote ("). The text matching the portion of the pattern between these markers is returned.

Return type: text

Example:

```
SELECT substr('Thomas' from '%"o_a#"_' for '#');
substr
-----
oma
(1 row)
```

rawcat(raw,raw)

Description: Concatenates the given strings.

Return type: raw

Example:

```
SELECT rawcat('ab','cd');
rawcat
-----
ABCD
(1 row)
```

regexp_like(text,text,text)

Description: Performs a regular expression matching.

Return type: bool

Example:

```
SELECT regexp_like('str','[ac]');
regexp_like
-----
f
(1 row)
```

regexp_substr(text,text)

Description: Extracts substrings from a regular expression. Its function is similar to **substr**. When a regular expression contains multiple parallel brackets, it also needs to be processed.

Return type: text

Example:

```
SELECT regexp_substr('str','[ac]');
regexp_substr
-----
(1 row)
```

regexp_matches(string text, pattern text [, flags text])

Description: Returns all captured substrings resulting from matching a POSIX regular expression against the **string**. If the pattern does not match, the function returns no rows. If the pattern contains no parenthesized sub-expressions, then each row returned is a single-element text array containing the substring matching the whole pattern. If the pattern contains parenthesized sub-expressions, the function returns a text array whose *n*th element is the substring matching the *n*th parenthesized sub-expression of the pattern.

The optional **flags** argument contains zero or multiple single-letter flags that change function behavior. **i** indicates that the matching is not related to uppercase and lowercase. **g** indicates that each matching substring is replaced, instead of replacing only the first one.

NOTICE

If the last parameter is provided but the parameter value is an empty string (") and the SQL compatibility mode of the database is set to ORA, the returned result is an empty set. This is because the ORA compatible mode treats the empty string (") as **NULL**. To resolve this problem, you can:

- Change the database SQL compatibility mode to TD.
- Do not provide the last parameter or do not set the last parameter to an empty string.

Return type: setof text[]

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
regexp_matches
-----
{bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
regexp_matches
-----
{barbeque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
regexp_matches
-----
{bar,beque}
{bazil,barf}
(2 rows)
```

**CAUTION**

If there is no subquery, the table data will not be displayed if there is no match for the **regexp_matches** function. This outcome is generally undesirable and should be avoided. It is recommended to use the **regexp_substr** function to achieve the same functionality.

```
SELECT * FROM tab;
c1 | c2
---+---
dws | dws
(1 row)

SELECT c1, regexp_matches(c2, '(bar)(beque)') FROM tab;
c1 | regexp_matches
---+-----
(0 rows)

SELECT c1, c2, (SELECT regexp_matches(c2, '(bar)(beque)')) FROM tab;
c1 | c2 | regexp_matches
---+---+-----
dws | dws |
(1 row)
```

regexp_split_to_array(string text, pattern text [, flags text])

Description: Splits **string** using a POSIX regular expression as the delimiter. The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of text.

Return type: text[]

Example:

```
SELECT regexp_split_to_array('hello world', E'\\s+');
 regexp_split_to_array
-----
{hello,world}
(1 row)
```

regexp_split_to_table(string text, pattern text [, flags text])

Description: Splits **string** using a POSIX regular expression as the delimiter. If there is no match to the pattern, the function returns the string. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string.

The **flags** parameter is a text string containing zero or more single-letter flags that change the function's behavior. **i** indicates that the matching is not related to uppercase and lowercase. **g** indicates that each matching substring is replaced, instead of replacing only the first one.

Return type: setof text

Example:

```
SELECT regexp_split_to_table('hello world', E'\\s+');
 regexp_split_to_table
-----
hello
world
(2 rows)
```

CAUTION

When a subquery is absent, and the **regexp_split_to_table** function fails to find a match, the table data will not be displayed. This outcome is generally undesirable and should be avoided.

```
SELECT * FROM tab;
 c1 | c2
-----+-----
 dws |
(1 row)

SELECT c1, regexp_split_to_table(c2, E'\\s+') FROM tab;
 c1 | regexp_split_to_table
-----+-----
(0 rows)

SELECT c1, (select regexp_split_to_table(c2, E'\\s+')) FROM tab;
 c1 | regexp_split_to_table
```

```
-----+-----  
dws |  
(1 row)
```

repeat(string text, number int)

Description: Outputs a string repeatedly for a specified number of times. If **number int** is set to zero or a negative value, an empty string is returned.

Return type: text

Example:

```
SELECT repeat('abc', 4);  
repeat  
-----  
abcbabcbabc  
(1 row)
```

replace(string text, from text, to text)

Description: Replace all occurrences in string of substring **from** with substring **to**.

NOTE

The case is sensitive, so you need to ensure that the case of **from text** and **to text** matches that of the original string.

Return type: text

Example:

```
SELECT replace('databasedata', 'data', 'xxx');  
replace  
-----  
xxxbasexxx  
(1 row)
```

reverse(str)

Description: Returns a string in reverse order.

Return type: text

Example:

```
SELECT reverse('abcde');  
reverse  
-----  
edcba  
(1 row)
```

right(str text, n int)

Description: Returns the last **n** characters in the string.

- In the ORA- or TD-compatible mode, all but the last **|n|** characters are returned if **n** is negative.
- In the MySQL-compatible mode, an empty string is returned if **n** is negative.

Return type: text

Example:

```
SELECT right('abcde', 2);
right
-----
de
(1 row)

SELECT right('abcde', -2);
right
-----
cde
(1 row)
```

rpad(string text, length int [, fill text])

Description: Fills up **string** to **length** by appending the characters **fill**. If the string is longer than the specified length, the string is truncated.

Parameters:

- **string**: original string to be processed.
- **length**: length of the target string.
- **fill** (optional): character or string used for filling. The default value is a space.

Return type: text

Example:

Fill the original string (**data**) of length 4 to a target length of 10 using the fill characters **xy**.

```
SELECT rpad('data', 10, 'xy');
rpad
-----
dataxyxyxy
(1 row)
```

Fill the original string **database** using the fill characters **xy**. The string **database** is longer than the target length **4**, and the string is truncated from the end. The returned string is **data**.

```
SELECT rpad('database',4,'xy');
rpad
-----
data
(1 row)
```

rtrim(string text [, characters text])

Description: Removes the specified characters from the right (end) of a string.

Parameters:

- **string**: original string to be processed.
- **characters** (optional): set of characters to be removed from the right of the string. By default, all blank characters (for example, spaces) are removed.

Return type: text

Example:

If the parameter **characters** is not specified, all blank characters on the right of the string **Database** are removed by default.

```
SELECT rtrim('Database ');
rtrim
-----
Database
(1 row)
```

Removes the specified character **x** from the string.

```
SELECT rtrim('trimxxxx', 'x');
rtrim
-----
trim
(1 row)
```

substrb(text,int,int)

Description: Extracts a substring within a string. The first **int** indicates the start position of the subtraction. The second **int** indicates the number of bytes subtracted.

Return type: text

Example:

```
SELECT substrb('string',2,3);
substrb
-----
tri
(1 row)
```

substrb(text,int)

Description: Extracts a substring within a string. **int** indicates the start position of the subtraction.

Return type: text

Example:

```
SELECT substrb('string',2);
substrb
-----
tring
(1 row)
```

string || string

Description: Concatenates strings.

Return type: text

Example:

```
SELECT 'DA' || 'TABASE' AS RESULT;
result
-----
DATABASE
(1 row)
```

string || non-string or non-string || string

Description: Concatenates strings and non-strings.

Return type: text

Example:

```
SELECT 'Value: '||123 AS RESULT;
result
-----
Value: 123
(1 row)
```

split_part(string text, delimiter text, field int)

Description: Splits the original string into several parts based on the specified delimiter (**delimiter** parameter) and returns the given field (**field** parameter). This function is used to extract specific fields from composite strings.

Parameters:

- **string**: original string to be split.
- **delimiter**: used to separate the string.
- **field**: given field returned after the split. The value starts from 1.

Return type: text

Example:

Split the string **abc~@~def~@~ghi** by **~@~** to obtain three parts (**abc**, **def**, and **ghi**), and return the second part, that is, **def**.

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
split_part
-----
def
(1 row)
```

strpos(string, substring)

Description: Obtains the position of the first occurrence of the specified substring in the target string. If the substring does not exist in the target string, 0 is returned.

Return type: integer

Example:

```
SELECT strpos('source', 'rc');
strpos
-----
4
(1 row)

SELECT strpos('source', 'data');
strpos
-----
0
(1 row)
```


to_hex(number int or bigint)

Description: Converts a number to a hexadecimal number.

Return type: text

Example:

```
SELECT to_hex(2147483647);
to_hex
-----
7fffffff
(1 row)
```

translate(string text, from text, to text)

Description: Replaces any character in the original string that matches a character in the **from** set by the corresponding character in the **to** set. If **from** is longer than **to**, extra characters occurred in **from** are removed.

Parameters:

- **string**: original string to be processed.
- **from**: specifies the characters to be replaced.
- **to**: specifies the characters after replacement.

Return type: text

Example:

Set the **from** parameter to **143** and the **to** parameter **ax**. Replace **1** and **4** in the string **12345** with **a** and **x** respectively. Delete **3** because there is no corresponding replacement character. **a2x5** is returned.

```
SELECT translate('12345', '143', 'ax');
translate
-----
a2x5
(1 row)
```

length(string)

Description: Obtains the character length (number of characters) of a string.

Return type: integer

Example:

```
SELECT length('database');
length
-----
8
(1 row)
```

lengthb(string)

Description: Obtains the length of a string in bytes. The value depends on character sets (GBK and UTF8).

Return type: integer

Example:

```
SELECT lengthb('hello');
lengthb
-----
      5
(1 row)
```

substr(string,from)

Description: Truncates a specified substring from a string.

from indicates the start position of the substring.

- If **from** starts at 0, the value **1** is used.
- If **from** is a positive value, all characters from **from** to the end are extracted.
- If **from** is a negative value, the last *n* characters in the string are extracted, in which *n* indicates the absolute value of **from**.

Return type: varchar

Example:

When **from** starts at **0**, extract all characters from the first character to the end of the string **database**.

```
SELECT substr('database',0);
substr
-----
database
(1 row)
```

When **from** is a positive value, extract all characters from the fifth character to the end of the string **database**.

```
SELECT substr('database',5);
substr
-----
base
(1 row)
```

When **from** is a negative value, extract all characters from the fourth character from the end to the end of the string **database**.

```
SELECT substr('database',-4);
substr
-----
base
(1 row)
```

substr(string,from,count)

Description: Truncates a substring of a specified length from a string.

from indicates the start position of the substring. **count** indicates the length of the extracted substring.

- If **from** starts at 0, the value **1** is used.
- If **from** is a positive value, extract **count** characters starting from **from**.
- If **from** is a negative value, extract the last *n* **count** characters in the string, in which *n* indicates the absolute value of **from**.
- When **count** is less than 1, null is returned.

Return type: varchar

Example:

When **from** starts at **0**, extract the first four characters from the string **database**.

```
SELECT substr('database',0,4);
substr
-----
data
(1 row)
```

When **from** is a positive number, extract the first four characters from the fifth character of the string **database**.

```
SELECT substr('database',5,4);
substr
-----
base
(1 row)
```

If **from** is a negative number, truncate three characters starting from the last fourth character of **database**.

```
SELECT substr('database',-4,3);
substr
-----
bas
(1 row)
```

substrb(string,from)

Description: The functionality of this function is the same as that of **SUBSTR(string,from)**. However, the calculation unit is byte.

Return type: bytea

Example:

```
SELECT substrb('ABCDEF',-2);
substrb
-----
EF
(1 row)
```

substrb(string,from,count)

Description: The functionality of this function is the same as that of **SUBSTR(string,from,count)**. However, the calculation unit is byte.

Return type: bytea

Example:

```
SELECT substrb('ABCDEF',2,2);
substrb
-----
BC
(1 row)
```

trim([leading |trailing |both] [characters] from string)

Description: Deletes specific characters from the beginning, end, or both ends of a string.

Parameters:

- **leading**: deletes spaces or specified characters at the beginning of a string.
- **trailing**: deletes spaces or specified characters at the end of a string.
- **both**: deletes spaces or specified characters at the beginning and end of a string. (It is a default behavior.)
- **characters**: (optional) specifies specific characters to be deleted. If this parameter is not specified, spaces are deleted by default.
- **string**: original string to be processed.

Return type: varchar

Example:

Deletes the characters **x** at the beginning and end of the string **xTomxx**.

```
SELECT trim(BOTH 'x' FROM 'xTomxx');
btrim
-----
Tom
(1 row)

SELECT trim('x' FROM 'xTomxx');
btrim
-----
Tom
(1 row)
```

Delete the character **x** at the beginning of the string **xTomxx**.

```
SELECT trim(LEADING 'x' FROM 'xTomxx');
ltrim
-----
Tomxx
(1 row)
```

Delete the character **x** at the end of the string **xTomxx**.

```
SELECT trim(TRAILING 'x' FROM 'xTomxx');
rtrim
-----
xTom
(1 row)
```

rtrim(string [, characters])

Description: Deletes the specified characters from the end of a string.

Parameters:

- **string**: original string to be processed.
- **characters** (optional): specifies the set of characters to be deleted from the end of a string. By default, all blank characters (for example, spaces) are deleted.

Return type: varchar

Example:

```
SELECT rtrim('DataxBasex','x');
rtrim
-----
DataBase
```

```
(1 row)
SELECT rtrim(' Data ');
rtrim
-----
Data
(1 row)
```

ltrim(string [, characters])

Description: Deletes the specified characters from the beginning of a string.

Parameters:

- **string**: original string to be processed.
- **characters** (optional): specifies the set of characters to be deleted from the beginning of a string. By default, all blank characters (for example, spaces) are deleted.

Return type: varchar

Example:

```
SELECT ltrim('xxDataxBaSe','x');
ltrim
-----
DataxBaSe
(1 row)

SELECT ltrim(' Data ');
ltrim
-----
Data
(1 row)
```

upper(string)

Description: Converts the string into the uppercase.

Return type: varchar

Example:

```
SELECT upper('database');
upper
-----
DATABASE
(1 row)
```

ucase(string)

Description: Converts the string into the uppercase.

Return type: varchar

Example:

```
SELECT ucase('database');
ucase
-----
DATABASE
(1 row)
```

lower(string)

Description: Converts the string into the lowercase.

Return type: varchar

Example:

```
SELECT lower('DATABASE');
lower
-----
database
(1 row)
```

lcase(string)

Description: Converts the string into the lowercase.

Return type: varchar

Example:

```
SELECT lcase('DATABASE');
lcase
-----
database
(1 row)
```

initcap(string)

Description: Converts the first letter of each word in the string into the uppercase and the other letters into the lowercase.

Return type: text

Example:

```
SELECT initcap('hi THOMAS');
initcap
-----
Hi Thomas
(1 row)
```

rpad(string varchar, length int [, fill varchar])

Description: Adds specified characters to the right of a string until the string reaches the specified length. If the string is longer than the specified length, the string is truncated.

Parameters:

- **string**: original string to be processed.
- **length**: length of the target string. **length** in DWS indicates the character length. One Chinese character is counted as one character.
- **fill** (optional): character or string used for filling. The default value is a space.

Return type: varchar

Example:

```
SELECT rpad('data',10,'xyz');
rpad
```

```
-----  
dataxyzxyz  
(1 row)  
SELECT rpad('hi',5,'abcdefg');  
rpad  
-----  
hiabc  
(1 row)
```

instr(string,substring[,position,occurrence])

Description: Queries and returns the value of the substring position that occurs the occurrence (first by default) times from the position (1 by default) in the string.

- If the value of **position** is **0**, **0** is returned.
- If the value of **position** is negative, searches backwards from the last *n*th character in the string, in which *n* indicates the absolute value of **position**.

In this function, the calculation unit is character. One Chinese character is one character.

Return type: integer

Example:

```
SELECT instr('corporate floor','or', 3);  
instr  
-----  
5  
(1 row)  
SELECT instr('corporate floor','or',-3,2);  
instr  
-----  
2  
(1 row)
```

locate(substring,string[,position])

Description: From the specified **position** (1 by default) in the string on, queries and returns the value of **position** where the substring occurs for the first time. The unit is character. If the string does not contain substrings, 0 is returned.

Return type: integer

Example:

```
SELECT locate('ball','football');  
locate  
-----  
5  
(1 row)  
SELECT locate('er','soccerplayer','6');  
locate  
-----  
11  
(1 row)
```

ascii(string)

Description: Returns the ASCII code of the first character in a string.

Return type: integer

Example:

Return the ASCII code of the first character **x** in the string **xyz**.

```
SELECT ascii('xyz');
ascii
-----
120
(1 row)
```

replace(string varchar, search_string varchar, replacement_string varchar)

Description: Searches for a specific substring (**search_string**) in a string and replaces it with another string (**replacement_string**).

Parameters:

- **string**: original string to be processed.
- **search_string**: substring to be replaced.
- **replacement_string**: new substring used to replace **search_string**.

Return type: varchar

Example:

Replace all substrings **ab** in the string **ababab** and **mabab** with **cd**.

```
SELECT replace('ababab and mabab','ab','cd');
replace
-----
cdcdcd and mcdcd
(1 row)
```

lpad(string varchar, length int[, repeat_string varchar])

Description: Adds specified characters to the left of a string until the string reaches the specified length. If the length of the original string is greater than the target length (**length parameter**), the target string is truncated from the right to a string of the length.

Parameters:

- **string**: original string to be padded.
- **length**: length of the target string.
- **repeat_string**: (optional) characters used for padding. By default, spaces are used for padding.

Return type: varchar

Example:

Set original string length (**base**) to **4** and the target length to **8**, and use asterisks (*) to pad four characters. After the padding, the string becomes ******base**.

```
SELECT lpad('base',8,'*');
lpad
-----
****base
(1 row)
```


Set original string length (**database**) to **8** and the target length to **6**, and truncate the string **database** from the right to a string of 6 characters. **databa** is returned.

```
SELECT lpad('database',6,'data');
lpad
-----
databa
(1 row)
```

concat(str1,str2)

Description: Connects str1 and str2 and returns the string.

- In the ORA- or TD-compatible mode, a combination of all the non-null strings is returned.
- In the MySQL-compatible mode, **NULL** is returned if an input string is **NULL**.

Return type: varchar

Example:

```
SELECT concat('Hello', ' World!');
concat
-----
Hello World!
(1 row)
```

chr(integer)

Description: Converts an integer to the corresponding ASCII character.

Return type: varchar

Example:

```
SELECT chr(65);
chr
-----
A
(1 row)
```

regexp_substr(source_char, pattern)

Description: Extracts substrings from a regular expression.

Return type: varchar

Example:

```
SELECT regexp_substr('500 Hello World, Redwood Shores, CA', '^[^,]+' "REGEXP_SUBSTR";
REGEXP_SUBSTR
-----
, Redwood Shores,
(1 row)
```

regexp_replace(string, pattern, replacement [,flags])

Description: Replaces substring matching POSIX regular expression. The source string is returned unchanged if there is no match to the pattern. If there is a match, the source string is returned with the replacement string substituted for the matching substring.

The replacement string can contain `\n`, where `n` is 1 through 9, to indicate that the source substring matching the *n*th parenthesized sub-expression of the pattern should be inserted, and it can contain `&` to indicate that the substring matching the entire pattern should be inserted.

The optional **flags** argument contains zero or multiple single-letter flags that change function behavior. The following table lists the options of the **flags** argument.

Table 7-1 Options of the flags argument

Option	Description
g	Replace all the matched substrings. (By default, only the first matched substring is replaced.)
B	Preferentially use the boost regex regular expression library and its regular expression syntax. By default, the Henry Spencer's regular expression library and its regular expression syntax are used. In the following cases, the Henry Spencer's regular expression library and its regular expression syntax will be used even if this option is specified: - One or multiple characters of p, q, w, and x are specified for flags. - The string or pattern parameter contains multi-byte characters.
b	Use POSIX Basic Regular Expressions (BREs) for matching.
c	Case-sensitive matching
e	Use POSIX Extended Regular Expressions (EREs) for matching. If neither b nor e is specified and the Henry Spencer's regular expression library is used, Advanced Regular Expressions (AREs), similar to Perl Compatible Regular Expressions (PCREs), are used for matching; if neither b nor e is specified and the boost regex regular expression library is used, PCREs are used for matching.
i	Case-insensitive matching
m	Line feed-sensitive matching, which has the same meaning as option n
n	Line feed-sensitive matching. When this option takes effect, the line separator affects the matching of metacharacters (<code>.</code> , <code>^</code> , <code>\$</code> , and <code>[]</code>).
p	Partial line feed-sensitive matching. When this option takes effect, the line separator affects the matching of metacharacters (<code>.</code> and <code>[]</code>). "Partial" is in comparison with option n .
q	Reset the regular expression to a text string enclosed in double quotation marks ("") and consisting of only common characters.
s	Non-line feed-sensitive matching
t	Compact syntax (default). When this option takes effect, all characters matter.

Option	Description
w	Reverse partial line feed-sensitive matching. When this option takes effect, the line separator affects the matching of metacharacters (^ and \$). "Partial" is in comparison with option n.
x	Extended syntax In contrast to the compact syntax, whitespace characters in regular expressions are ignored in the extended syntax. Whitespace characters include spaces, horizontal tabs, new lines, and any other characters in the space character table.

Return type: varchar

Example:

```
SELECT regexp_replace('Thomas', '[mN]a.', 'M');
regexp_replace
-----
ThM
(1 row)
SELECT regexp_replace('foobarbaz', 'b(..)', 'E'X\\1Y', 'g') AS RESULT;
result
-----
fooXarYXazY
(1 row)
```

concat_ws(sep text, str"any" [, str"any" [, ...]])

Description: Uses the first parameter as the separator, which is associated with all following parameters.

Return type: text

Example:

```
SELECT concat_ws(',', 'ABCDE', 2, NULL, 22);
concat_ws
-----
ABCDE,2,22
(1 row)
```

convert(string bytea, src_encoding name, dest_encoding name)

Description: Converts the bytea string to **dest_encoding**. **src_encoding** specifies the source code encoding. The string must be valid in this encoding.

Return type: bytea

Example:

```
SELECT convert('text_in_utf8', 'UTF8', 'GBK');
convert
-----
\x746578745f696e5f75746638
(1 row)
```

 NOTE

If the rule for converting between source to target encoding (for example, GBK and LATIN1) does not exist, the string is returned without conversion. See the **pg_conversion** system catalog for details.

Example:

```
show server_encoding;
server_encoding
-----
LATIN1
(1 row)

SELECT convert_from('some text', 'GBK');
convert_from
-----
some text
(1 row)

db_latin1=# SELECT convert_to('some text', 'GBK');
convert_to
-----
\x7366d652074657874
(1 row)

db_latin1=# SELECT convert('some text', 'GBK', 'LATIN1');
convert
-----
\x7366d652074657874
(1 row)
```

convert_from(string bytea, src_encoding name)

Description: Converts the long bytea using the coding mode of the database.

src_encoding specifies the source code encoding. The string must be valid in this encoding.

Return type: text

Example:

```
SELECT convert_from('text_in_utf8', 'UTF8');
convert_from
-----
text_in_utf8
(1 row)
SELECT convert_from('\x6461746162617365','gbk');
convert_from
-----
database
(1 row)
```

convert_to(string text, dest_encoding name)

Description: Converts the given string into one whose encoding format is **dest_encoding**.

Return type: bytea

Example:

```
SELECT convert_to('some text', 'UTF8');
convert_to
-----
```

```
\x736f6d652074657874
(1 row)
SELECT convert_to('database', 'gbk');
       convert_to
-----
\x6461746162617365
(1 row)
```

string [NOT] LIKE pattern [ESCAPE escape-character]

Description: Pattern matching function

If the pattern does not include a percentage sign (%) or an underscore (_), this mode represents itself only. In this case, the behavior of LIKE is the same as the equal operator. The underscore (_) in the pattern matches any single character while one percentage sign (%) matches no or multiple characters.

To match with underscores (_) or percent signs (%), corresponding characters in pattern must lead escape characters. The default escape character is a backward slash (\) and can be specified using the **ESCAPE** clause. To match with escape characters, enter two escape characters.

Return type: boolean

Example:

```
SELECT 'AA_BBCC' LIKE '%A@_B%' ESCAPE '@' AS RESULT;
       result
-----
t
(1 row)
SELECT 'AA_BBCC' LIKE '%A@_B%' AS RESULT;
       result
-----
f
(1 row)
SELECT 'AA@_BBCC' LIKE '%A@_B%' AS RESULT;
       result
-----
t
(1 row)
```

REGEXP_LIKE(source_string, pattern [, match_parameter])

Description: Performs a regular expression matching.

source_string indicates the source string and **pattern** indicates the matching pattern of the regular expression. **match_parameter** indicates the matching items and the values are as follows:

- "i": case-insensitive
- "c": case-sensitive
- "n": allowing the metacharacter "." in a regular expression to be matched with a linefeed.
- "m": allows **source_string** to be regarded as multiple rows.

If **match_parameter** is ignored, **case-sensitive** is enabled by default, "." is not matched with a linefeed, and **source_string** is regarded as a single row.

Return type: boolean

Example:

```
SELECT regexp_like('ABC', '[A-Z]');
regexp_like
-----
t
(1 row)
SELECT regexp_like('ABC', '[D-Z]');
regexp_like
-----
f
(1 row)
SELECT regexp_like('abc', '[A-Z]', 'i');
regexp_like
-----
t
(1 row)
SELECT regexp_like('abc', '[A-Z]');
regexp_like
-----
f
(1 row)
```

format(formatstr text [, str"any" [, ...]])

Description: Formats a string.

Return type: text

Example:

```
SELECT format('Hello %s, %1$s', 'World');
format
-----
Hello World, World
(1 row)
```

md5(string)

Description: Encrypts a string using MD5 and returns a hexadecimal number.

NOTE

MD5 is insecure and is not recommended.

Return type: text

Example:

```
SELECT md5('data');
md5
-----
8d777f385d3dfec8815d20f7496026dc
(1 row)
```

decode(string text, format text)

Description: Decodes binary data from textual representation.

Return type: bytea

Example:

```
SELECT decode('ZGF0YWJhc2U=', 'base64');
decode
```

```
-----
\x6461746162617365
(1 row)

SELECT convert_from('\x6461746162617365','utf-8');
convert_from
-----
database
(1 row)
```

encode(data bytea, format text)

Description: Encodes binary data into a textual representation.

Return type: text

Example:

```
SELECT encode('database', 'base64');
encode
-----
ZGF0YWJhc2U=
(1 row)
```

CONV(n, fromBase, toBase)

Description: Converts the given value or string into value of a specific number system and outputs the result as a string. If the parameter contains NULL, NULL will be returned. The output value range is [-36, -2] & [2, 36].

Return type: text

Example:

```
SELECT CONV(-1, 10, 16) as result;
result
-----
FFFFFFFFFFFFFFFF
(1 row)
```

HEX(n)

Description: Returns the hexadecimal string of **n**. **n** can be an integer or a string. If the parameter contains NULL, NULL will be returned.

Return type: text

Example:

```
SELECT HEX(255) as result;
result
-----
FF
(1 row)
SELECT HEX('abc') as result;
result
-----
616263
(1 row)
```

UNHEX(n)

Description: Performs the reverse operation of **HEX(n)**. **n** can be an int or a string. Each pair of hexadecimal digits in the parameter is regarded as a number and

converted into the integer representation of the hexadecimal value. If the parameter contains NULL, NULL will be returned. This function is supported by version 8.2.0 or later clusters.

Return type: bytea

Example:

```
SELECT UNHEX('abc') as result;
result
-----
\x0abc
(1 row)
```

SPACE(n int)

Description: Returns a string consisting of **n** spaces. If the parameter contains NULL, NULL will be returned.

Return type: text

Example:

```
SELECT SPACE(2) as result;
result
-----
(1 row)
```

STRCMP(text, text)

Description: Compares two strings. If all strings are the same, **0** is returned. If the first string is smaller than the second string, **-1** is returned. In other cases, **1** is returned. If the parameter contains NULL, NULL will be returned.

Return type: text

Example:

```
SELECT STRCMP('AA', 'AA'), STRCMP('AA', 'AB'), STRCMP('AA', 'A');
STRCMP | STRCMP | STRCMP
-----
0 | -1 | 1
(1 row)
```

BIN(n bigint)

Description: Converts the bigint type from decimal to binary and returns the result as a string. If the parameter contains NULL, NULL will be returned.

Return type: text

Example:

```
SELECT BIN(16) as result;
result
-----
10000
(1 row)
```


7.2 Binary String Functions and Operators

SQL defines some binary string functions that use keywords, rather than commas, to separate arguments. GaussDB(DWS) also provides the common syntax used for invoking functions.

octet_length(string)

Description: Returns the number of bytes in the given binary string.

Return type: integer

Example:

```
SELECT octet_length(E'jo\000se'::bytea) AS RESULT;  
result  
-----  
5  
(1 row)
```

overlay(string placing string from int [for int])

Description: Replaces the given substring.

Return type: bytea

Example:

```
SELECT overlay(E'Th\000omas'::bytea placing E'\002\003'::bytea from 2 for 3) AS RESULT;  
result  
-----  
\x5402036d6173  
(1 row)
```

position(substring in string)

Description: Returns the location of the given substring.

Return type: integer

Example:

```
SELECT position(E'\000om'::bytea in E'Th\000omas'::bytea) AS RESULT;  
result  
-----  
3  
(1 row)
```

substring(string [from int] [for int])

Description: Truncates the given substring.

Return type: bytea

Example:

```
SELECT substring(E'Th\000omas'::bytea from 2 for 3) AS RESULT;  
result  
-----  
\x68006f  
(1 row)
```

Truncate the time and obtain the number of hours.

```
SELECT substring('2022-07-18 24:38:15',12,2)AS RESULT;  
result  
-----  
24  
(1 row)
```

trim([both] bytes from string)

Description: Removes the longest string containing only bytes from **bytes** from the start and end of **string**.

Return type: bytea

Example:

```
SELECT trim(E'\000'::bytea from E'\000Tom\000'::bytea) AS RESULT;  
result  
-----  
\x546f6d  
(1 row)
```

substring_index(string, delim, count)

Description: Searches for delimiters in a case-sensitive manner and returns the substring before the delimiter that appears for the **count** time. If **count** is a negative number, count for the delimiter reversely from the end. If the parameter contains NULL, NULL will be returned. This function is supported by version 8.2.0 or later clusters.

Return type: text

Example: In the string **www.wWw.cloud.wWw.com**, find the delimiter **.wWw.** that appears the second time in a case-sensitive manner. Return the substring before it: **www.wWw.cloud**.

```
SELECT SUBSTRING_INDEX('www.wWw.cloud.wWw.com', '.wWw.', 2) AS RESULT;  
result  
-----  
www.wWw.cloud  
(1 row)
```

btrim(string bytea,bytes bytea)

Description: Removes the longest string containing only bytes from **bytes** from the start and end of **string**.

Return type: bytea

Example:

```
SELECT btrim(E'\000trim\000'::bytea, E'\000'::bytea) AS RESULT;  
result  
-----  
\x7472696d  
(1 row)
```

get_bit(string, offset)

Description: Returns the number of bits in the given string.

Return type: integer

Example:

```
SELECT get_bit(E'Th\\000omas':::bytea, 45) AS RESULT;  
result  
-----  
1  
(1 row)
```

get_byte(string, offset)

Description: Returns the number of bytes in the given string.

Return type: integer

Example:

```
SELECT get_byte(E'Th\\000omas':::bytea, 4) AS RESULT;  
result  
-----  
109  
(1 row)
```

set_bit(string,offset, newvalue)

Description: Sets bits in the given string.

Return type: bytea

Example:

```
SELECT set_bit(E'Th\\000omas':::bytea, 45, 0) AS RESULT;  
result  
-----  
\x5468006f6d4173  
(1 row)
```

set_byte(string,offset, newvalue)

Description: Sets bytes in the given string.

Return type: bytea

Example:

```
SELECT set_byte(E'Th\\000omas':::bytea, 4, 64) AS RESULT;  
result  
-----  
\x5468006f406173  
(1 row)
```

7.3 Bit String Functions and Operators

Aside from the usual comparison operators, the following operators can be used. Bit string operands of **&**, **|**, and **#** must be of equal length. When bit shifting, the original length of the string is preserved by zero padding (if necessary).

||

Description: Connects bit strings.

For example:

```
SELECT B'10001' || B'011' AS RESULT;  
result  
-----  
10001011  
(1 row)
```

&

Description: AND operation between bit strings

For example:

```
SELECT B'10001' & B'01101' AS RESULT;  
result  
-----  
00001  
(1 row)
```

|

Description: OR operation between bit strings

For example:

```
SELECT B'10001' | B'01101' AS RESULT;  
result  
-----  
11101  
(1 row)
```

#

Description: OR operation between bit strings if they are inconsistent. If the same positions in the two bit strings are both 1 or 0, the position returns **0**.

For example:

```
SELECT B'10001' # B'01101' AS RESULT;  
result  
-----  
11100  
(1 row)
```

~

Description: NOT operation between bit strings

For example:

```
SELECT ~B'10001' AS RESULT;  
result  
-----  
01110  
(1 row)
```

<<

Description: binary left shift

For example:

```
SELECT B'10001' << 3 AS RESULT;  
result  
-----  
01000  
(1 row)
```

>>

Description: binary right shift

For example:

```
SELECT B'10001' >> 2 AS RESULT;  
result  
-----  
00100  
(1 row)
```

The following SQL-standard functions work on bit strings as well as character strings: **length**, **bit_length**, **octet_length**, **position**, **substring**, and **overlay**.

The following functions work on bit strings as well as binary strings: **get_bit** and **set_bit**. When working with a bit string, these functions number the first (leftmost) bit of the string as bit 0.

Integers and bits can be mutually converted. For example:

```
SELECT 44::bit(10) AS RESULT;  
result  
-----  
0000101100  
(1 row)  
  
SELECT 44::bit(3) AS RESULT;  
result  
-----  
100  
(1 row)  
  
SELECT cast(-44 as bit(12)) AS RESULT;  
result  
-----  
111111010100  
(1 row)  
  
SELECT '1110'::bit(4)::integer AS RESULT;  
result  
-----  
14  
(1 row)
```

NOTE

Casting to just "bit" means casting to bit(1), and so will deliver only the least significant bit of the integer.

7.4 Numeric Functions and Operators

7.4.1 Numeric Operators

+

Description: addition

Example:

```
SELECT 2+3 AS RESULT;  
result  
-----  
      5  
(1 row)
```

-

Description: subtraction

Example:

```
SELECT 2-3 AS RESULT;  
result  
-----  
     -1  
(1 row)
```

*

Description: multiplication

Example:

```
SELECT 2*3 AS RESULT;  
result  
-----  
      6  
(1 row)
```

/

Description: division (The result is not rounded.)

Example:

```
SELECT 4/2 AS RESULT;  
result  
-----  
      2  
(1 row)  
SELECT 4/3 AS RESULT;  
result  
-----  
1.3333333333333333  
(1 row)
```

+/-

Description: positive/negative

Example:

```
SELECT -2 AS RESULT;  
result  
-----  
     -2  
(1 row)
```

%

Description: modulo (to obtain the remainder)

Example:

```
SELECT 5%4 AS RESULT;  
result  
-----  
1  
(1 row)
```

@

Description: absolute value

Example:

```
SELECT @ -5.0 AS RESULT;  
result  
-----  
5.0  
(1 row)
```

^

Description: power (exponent calculation)

In MySQL-compatible mode, this operator means exclusive OR. For details, see operator <#> in [Bit String Functions and Operators](#).

Example:

```
SELECT 2.0^3.0 AS RESULT;  
result  
-----  
8.0000000000000000  
(1 row)
```

||

Description: square root

Example:

```
SELECT || 25.0 AS RESULT;  
result  
-----  
5  
(1 row)
```

|||

Description: cubic root

Example:

```
SELECT ||| 27.0 AS RESULT;  
result  
-----  
3  
(1 row)
```

!

Description: factorial

Example:

```
SELECT 5! AS RESULT;  
result  
-----  
120  
(1 row)
```

!!

Description: factorial (prefix operator)

Example:

```
SELECT !!5 AS RESULT;  
result  
-----  
120  
(1 row)
```

&

Description: binary AND

Example:

```
SELECT 91&15 AS RESULT;  
result  
-----  
11  
(1 row)
```

|

Description: binary OR

Example:

```
SELECT 32|3 AS RESULT;  
result  
-----  
35  
(1 row)
```

#

Description: binary XOR

Example:

```
SELECT 17#5 AS RESULT;  
result  
-----  
20  
(1 row)
```

~

Description: binary NOT

Example:

```
SELECT ~1 AS RESULT;  
result
```



```
-----  
-2  
(1 row)
```

<<

Description: binary shift left

Example:

```
SELECT 1<<4 AS RESULT;  
result  
-----  
16  
(1 row)
```

>>

Description: binary shift right

Example:

```
SELECT 8>>2 AS RESULT;  
result  
-----  
2  
(1 row)
```

7.4.2 Numeric Functions

abs(x)

Description: Returns the absolute value.

Return type: same as the input

Example:

```
SELECT abs(-17.4);  
abs  
-----  
17.4  
(1 row)
```

acos(x)

Description: Returns the arccosine.

Return type: double precision

Example:

```
SELECT acos(-1);  
acos  
-----  
3.14159265358979  
(1 row)
```

asin(x)

Description: Returns the arcsine.

Return type: double precision

Example:

```
SELECT asin(0.5);
       asin
-----
0.523598775598299
(1 row)
```

atan(x)

Description: Returns the arctangent.

Return type: double precision

Example:

```
SELECT atan(1);
       atan
-----
0.785398163397448
(1 row)
```

atan2(y, x)

Description: Returns the arctangent of y/x.

Return type: double precision

Example:

```
SELECT atan2(2, 1);
       atan2
-----
1.10714871779409
(1 row)
```

bitand(integer, integer)

Description: Performs AND (&) operation on two integers.

Return type: bigint

Example:

```
SELECT bitand(127, 63);
       bitand
-----
63
(1 row)
```

cbrt(double precision)

Description: Returns the cubic root.

Return type: double precision

Example:

```
SELECT cbrt(27.0);
       cbrt
-----
3
(1 row)
```

ceil(double precision or numeric)

Description: Returns the minimum integer greater than or equal to a specified number.

Return type: same as the input

Example:

```
SELECT ceil(-42.8);
ceil
-----
-42
(1 row)
```

ceiling(double precision or numeric)

Description: Returns the minimum integer greater than or equal to a specified number. This function is the alias of the ceil(x) function.

Return type: same as the input

Example:

```
SELECT ceiling(-95.3);
ceiling
-----
-95
(1 row)
```

cos(x)

Description: Returns the cosine.

Return type: double precision

Example:

```
SELECT cos(-3.1415927);
cos
-----
-0.9999999999999999
(1 row)
```

cot(x)

Description: Returns the cotangent.

Return type: double precision

Example:

```
SELECT cot(1);
cot
-----
0.642092615934331
(1 row)
```

degrees(double precision)

Description: Converts radians to angles.

Return type: double precision

Example:

```
SELECT degrees(0.5);
degrees
-----
28.6478897565412
(1 row)
```

div(y numeric, x numeric)

Description: Returns the integer part of y divided by x.

Return type: numeric

Example:

```
SELECT div(9,4);
div
----
2
(1 row)
```

exp(double precision or numeric)

Description: Returns the natural exponent.

Return type: same as the input

Example:

```
SELECT exp(1.0);
exp
-----
2.7182818284590452
(1 row)
```

floor(double precision or numeric)

Description: Returns the maximum integer less than or equal to a specified number.

Return type: same as the input

Example:

```
SELECT floor(-42.8);
floor
-----
-43
(1 row)
```

radians(double precision)

Description: Converts angles to radians.

Return type: double precision

Example:

```
SELECT radians(45.0);
radians
-----
0.785398163397448
(1 row)
```

random()

Description: Returns a random number between 0.0 and 1.0.

Return type: double precision

Example:

```
SELECT random();
       random
-----
0.824823560658842
(1 row)
```

rand()

Description: Returns a random number between 0.0 and 1.0. This function is compatible with MySQL. This function is supported by version 8.2.0 or later clusters.

Return type: double precision

Example:

```
SELECT rand();
       rand
-----
0.824823560658842
(1 row)
```

ln(double precision or numeric)

Description: Returns the natural logarithm.

Return type: same as the input

Example:

```
SELECT ln(2.0);
       ln
-----
0.6931471805599453
(1 row)
```

log(double precision or numeric)

Description: Returns the logarithm with base 10.

- In the ORA- or TD-compatible mode, this function returns the logarithm with base 10.
- In the MySQL-compatible mode, this function returns the natural logarithm.

Return type: same as the input

Example:

```
-- ORA-compatible mode
SELECT log(100.0);
       log
-----
2.0000000000000000
(1 row)
-- TD-compatible mode
```

```
SELECT log(100.0);
      log
-----
2.0000000000000000
(1 row)
-- MySQL-compatible mode
SELECT log(100.0);
      log
-----
4.6051701859880914
(1 row)
```

log(b numeric, x numeric)

Description: Returns the logarithm with base b.

Return type: numeric

Example:

```
SELECT log(2.0, 64.0);
      log
-----
6.0000000000000000
(1 row)
```

mod(x,y)

Description: Specifies the remainder of x/y (modulus). If x is 0, 0 is returned. If y is 0, x is returned.

Return type: same as the parameter type

Example:

```
SELECT mod(9,4);
      mod
-----
1
(1 row)
SELECT mod(9,0);
      mod
-----
9
(1 row)
```

pi()

Description: Returns constant π .

Return type: double precision

Example:

```
SELECT pi();
      pi
-----
3.14159265358979
(1 row)
```

power(a double precision, b double precision)

Description: Returns a raised to the power of b.

Return type: double precision

Example:

```
SELECT power(9.0, 3.0);
power
-----
729.0000000000000000
(1 row)
```

round(double precision or numeric)

Description: Returns the integer closest to the input parameter.

Return type: same as the input

Example:

```
SELECT round(42.4);
round
-----
42
(1 row)

SELECT round(42.6);
round
-----
43
(1 row)
```

NOTE

When the **round** function is invoked, the numeric type is rounded to zero. While on most computers, the real number and the double-precision number are rounded to the nearest even number.

round(v numeric, s int)

Description: Rounds to **s** decimal places.

Return type: numeric

Example:

```
SELECT round(42.4382, 2);
round
-----
42.44
(1 row)
```

setseed(double precision)

Description: Sets seed for the following random() invoking (between -1.0 and 1.0, inclusive).

Return type: void

Example:

```
SELECT setseed(0.54823);
setseed
-----
(1 row)
```

sign(double precision or numeric)

Description: Returns symbols of this parameter.

Return type: **-1** indicates a negative number, **0** indicates 0, and **1** indicates a positive number.

Example:

```
SELECT sign(-8.4);
sign
-----
-1
(1 row)
```

sin(x)

Description: Returns the sine.

Return type: double precision

Example:

```
SELECT sin(1.57079);
sin
-----
0.999999999979986
(1 row)
```

sqrt(double precision or numeric)

Description: Returns the square root.

Return type: same as the input

Example:

```
SELECT sqrt(2.0);
sqrt
-----
1.414213562373095
(1 row)
```

tan(x)

Description: Returns the tangent.

Return type: double precision

Example:

```
SELECT tan(20);
tan
-----
2.23716094422474
(1 row)
```

trunc(double precision or numeric)

Description: Returns the integer part of a number.

Return type: same as the input

Example:

```
SELECT trunc(42.8);
trunc
-----
42
(1 row)
```

trunc(v numeric, s int)

Description: Truncates a number to **s** decimal places.

Return type: numeric

Example:

```
SELECT trunc(42.4382, 2);
trunc
-----
42.43
(1 row)
```

truncate(v numeric, s int)

Description: Truncates **v** to **s** decimal places. In addition to any precision, **v** can also be an integer or float number. The return value type is the same as that of the input parameter **v**. If **s** is a negative number, the integer part is truncated. This function is supported by version 8.2.0 or later clusters.

Return type: numeric

Example:

```
SELECT trunc(42.4382, 2);
trunc
-----
42.43
(1 row)
```

width_bucket(operand numeric, b1 numeric, b2 numeric, count int)

Description: Sets the minimum value, maximum value, and number of groups in a group range, constructs a specified number of groups with the same size, and returns the ID of the group to which a specified field value belongs. **b1** is the minimum value of the group range, **b2** is the maximum value of the group range, and **count** is the number of groups.

Return type: integer

Example:

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
-----
3
(1 row)
```

width_bucket(operand double precision, b1 double precision, b2 double precision, count int)

Description: Sets the minimum value, maximum value, and number of groups in a group range, constructs a specified number of groups with the same size, and

returns the ID of the group to which a specified field value belongs. **b1** is the minimum value of the group range, **b2** is the maximum value of the group range, and **count** is the number of groups.

Return type: integer

Example:

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
-----
          3
(1 row)
```

7.5 Date and Time Processing Functions and Operators

7.5.1 Date and Time Operators

NOTICE

When you use date and time operators, add prefixes of explicit types to the operands to ensure that the operands can be properly parsed by the database.

For example, the operand in the following example does not have an explicit data type, and an error will occur.

```
SELECT date '2001-10-01' - '7' AS RESULT;
```

Table 7-2 Date and time operators

Oper ator	Example	Result:
+	date + integer → timestamp Add days to a date. SELECT date '2001-09-28' + integer '7' AS RESULT;	Add a date with an integer to obtain the time after 7 days. 2001-10-05 00:00:00

Operator	Example	Result:
+	date + interval → timestamp Add an interval to a date. <pre>SELECT date '2001-09-28' + interval '1 hour' AS RESULT; SELECT date '2021-02-28' + interval '1 month' AS RESULT; SELECT date '2021-01-31' + interval '1 month' AS RESULT;</pre>	Sum of the date parameter and the interval parameter Obtain the time after one hour. <pre>2001-09-28 01:00:00</pre> Obtain the time after one month. <pre>2021-02-28 00:00:00</pre> If the sum or subtraction results fall beyond the date range of a month, the result will be rounded to the last day of the month. <pre>2021-03-28 00:00:00</pre>
+	date + time → timestamp Add the time of a day to a date. <pre>SELECT date '2001-09-28' + time '03:00' AS RESULT;</pre>	Add a date and a time to obtain a specific time. <pre>2001-09-28 03:00:00</pre>
+	interval + interval → interval Add an interval. <pre>SELECT interval '1 day' + interval '1 hour' AS RESULT;</pre>	Add two intervals to obtain the sum. <pre>1 day 01:00:00</pre>
+	timestamp + interval → timestamp Add an interval to a timestamp. <pre>SELECT timestamp '2001-09-28 01:00' + interval '23 hours' AS RESULT;</pre>	Add a timestamp and an interval to obtain the time after 23 hours. <pre>2001-09-29 00:00:00</pre>
+	time + interval → time Add an interval to a time. <pre>SELECT time '01:00' + interval '3 hours' AS RESULT;</pre>	Add a time and an interval to obtain the time after three hours. <pre>04:00:00</pre>
-	date - date → integer Subtract a date from another to obtain the time difference. <pre>SELECT date '2001-10-01' - date '2001-09-28' AS RESULT;</pre>	Subtract a date from another to obtain the time difference. <pre>3 days</pre>
-	date - integer → timestamp Subtract days from a date. <pre>SELECT date '2001-10-01' - integer '7' AS RESULT;</pre>	Subtract an integer from a date to obtain the time difference between them. <pre>2001-09-24 00:00:00</pre>
-	date - interval → timestamp Subtract an interval from a date. <pre>SELECT date '2001-09-28' - interval '1 hour' AS RESULT;</pre>	Subtract an interval from a date to obtain the date and time difference. <pre>2001-09-27 23:00:00</pre>

Operator	Example	Result:
-	time - time → interval Subtract a time. SELECT time '05:00' - time '03:00' AS RESULT;	Subtract a time from another time to obtain the time difference. 02:00:00
-	time - interval → time Subtract an interval from a time. SELECT time '05:00' - interval '2 hours' AS RESULT;	Subtract an interval from a time to obtain the time difference. 03:00:00
-	timestamp - interval → timestamp Subtract an interval from a timestamp. SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT;	Subtract an interval from a timestamp to obtain the date and time difference. 2001-09-28 00:00:00
-	interval - interval → interval Subtract an interval. SELECT interval '1 day' - interval '1 hour' AS RESULT;	Subtract an interval from another interval to obtain the time difference. 23:00:00
-	timestamp - timestamp → interval Subtract a timestamp. SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT;	Subtract a timestamp from another timestamp to obtain the time difference. 1 day 15:00:00
-	select now() - interval → timestamp with time zone Obtain the time on the previous day. select now() - interval '1 day' AS RESULT;	Obtain the time on the previous day. 2025-07-29 10:54:29.116501+08
*	interval * double precision → interval Multiply an interval by a quantity. SELECT 900 * interval '1 second' AS RESULT; SELECT 21 * interval '1 day' AS RESULT; SELECT double precision '3.5' * interval '1 hour' AS RESULT;	Multiply the interval of 1 second by 900. 00:15:00 Multiply the interval of one day by 21. 21 days Multiply the interval of one hour by 3.5. 03:30:00
/	interval / double precision → interval SELECT interval '1 hour' / double precision '1.5' AS RESULT;	Divide an interval by a quantity to obtain a time segment. 00:40:00

7.5.2 Date and Time Functions

age(timestamp, timestamp)

Description: Subtracts arguments and returns the difference. If the result is negative, the returned result is also negative.

Return type: interval

Example:

```
SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
      age
-----
43 years 9 mons 27 days
(1 row)
```

age(timestamp)

Description: Subtracts the argument from the current date.

Return type: interval

Example:

```
SELECT age(timestamp '1957-06-13');
      age
-----
60 years 2 mons 18 days
(1 row)
```

adddate(date, interval | int)

Description: Returns the result of a given datetime plus the time interval of a specified unit. The default unit is day (when the second parameter is an integer).

Return type: timestamp

Example:

When the input parameter is of the text type:

```
select adddate('2020-11-13', 10);
      adddate
-----
2020-11-23
(1 row)

select adddate('2020-11-13', interval '1' month);
      adddate
-----
2020-12-13
(1 row)

select adddate('2020-11-13 12:15:16', interval '1' month);
      adddate
-----
2020-12-13 12:15:16
(1 row)

select adddate('2020-11-13', interval '1' minute);
      adddate
-----
```

```
2020-11-13 00:01:00  
(1 row)
```

When the input parameter is of the date type:

```
select adddate(current_date, 10);  
-----  
adddate  
-----  
2021-09-24  
(1 row)  
  
select adddate(date '2020-11-13', interval '1' month);  
-----  
adddate  
-----  
2020-12-13 00:00:00  
(1 row)
```

addtime(timestamp | time | text, interval | text)

Description: Returns the result of a given datetime plus the time interval of a specified unit. This function is supported by version 8.2.0 or later clusters.

Return type: same as the type of the first input parameter.

Example:

```
SELECT ADDTIME('2020-11-13 01:01:01', '23:59:59');  
-----  
addtime  
-----  
2020-11-14 01:01:00  
(1 row)
```

addtime(time, interval)

```
SELECT ADDTIME('20:1:1'::TIME, '20:59:59+8');  
-----  
addtime  
-----  
233:01:00  
(1 row)  
  
SELECT ADDTIME('20:1:1'::TIME, '8 20:59:59');  
-----  
addtime  
-----  
233:01:00  
(1 row)
```

addtime(text, interval)

```
SELECT ADDTIME('20:1:1', '20:59:59+8');  
-----  
addtime  
-----  
233:01:00  
(1 row)  
SELECT ADDTIME('20:1:1', '8 20:59:59');  
-----  
addtime  
-----  
233:01:00  
(1 row)
```

subdate(date, interval | int)

Description: Returns the result of a given datetime minus the time interval of a specified unit. The default unit is day (when the second parameter is an integer).

Return type: timestamp

Example:

When the input parameter is of the text type:

```
select subdate('2020-11-13', 10);
subdate
-----
2020-11-03
(1 row)

select subdate('2020-11-13', interval '2' month);
subdate
-----
2020-09-13
(1 row)

select subdate('2020-11-13 12:15:16', interval '1' month);
subdate
-----
2020-10-13 12:15:16
(1 row)

select subdate('2020-11-13', interval '2' minute);
subdate
-----
2020-11-12 23:58:00
(1 row)
```

When the input parameter is of the date type:

```
select subdate(current_date, 10);
subdate
-----
2021-09-05
(1 row)

select subdate(current_date, interval '1' month);
subdate
-----
2021-08-15 00:00:00
(1 row)
```

subtime(timestamp | time | text, interval | text)

Description: Returns the result of a given datetime minus the time interval of a specified unit. This function is supported by version 8.2.0 or later clusters.

Return type: same as the type of the first input parameter.

Example:

```
select subtime('2020-11-13 01:01:01', '23:59:59');
addtime
-----
2020-11-12 01:01:02
(1 row)
```

date_add(date, interval)

Description: Returns the result of a given datetime plus the time interval of a specified unit. It is equivalent to [adddate\(date, interval | int\)](#).

Return type: timestamp

date_sub(date, interval)

Description: Returns the result of a given datetime minus the time interval of a specified unit. It is equivalent to [subdate\(date, interval | int\)](#).

Return type: timestamp

timestampadd(field, numeric, timestamp)

Description: Adds an integer interval in the unit of **field** (the number of seconds can be a decimal) to a datetime expression. If the value is negative, the corresponding time interval is subtracted from the given datetime expression. The **field** can be **year**, **month**, **quarter**, **day**, **week**, **hour**, **minute**, **second**, or **microsecond**.

Return type: timestamp

Example:

```
select timestampadd(year, 1, timestamp '2020-2-29');
timestampadd
-----
2021-02-28 00:00:00
(1 row)

select timestampadd(second, 2.354156, timestamp '2020-11-13');
timestampadd
-----
2020-11-13 00:00:02.354156
(1 row)
```

timestampdiff(field, timestamp1, timestamp2)

Description: Subtracts **timestamp1** from **timestamp2** and returns the difference in the unit of **field**. If the difference is negative, this function returns it normally. The **field** can be **year**, **month**, **quarter**, **day**, **week**, **hour**, **minute**, **second**, or **microsecond**.

Return type: bigint

Example:

```
SELECT timestampdiff(day, timestamp '2001-02-01', timestamp '2003-05-01 12:05:55');
timestampdiff
-----
819
(1 row)
```

timediff(timestamp | time | text, timestamp | time | text)

Description: Subtracts a date from another date. If the difference is negative, this function returns it normally. The types of the two input parameters must be the same. This function is supported by version 8.2.0 or later clusters.

Return type: time

Example:

```
SELECT timediff('2022-7-5 1:1:1', '2021-7-5 1:1:1');
timediff
-----
```



```
8760:00:00
(1 row)
```

clock_timestamp()

Description: Returns the current timestamp of the real-time clock.

Return type: timestamp with time zone

Example:

```
SELECT clock_timestamp();
       clock_timestamp
-----
2017-09-01 16:57:36.636205+08
(1 row)
```

current_date

Description: Returns the current date.

Return type: date

Example:

```
SELECT current_date;
       date
-----
2017-09-01
(1 row)
```

curdate()

Description: Returns the current date. This function is compatible with MySQL. This function is supported by version 8.2.0 or later clusters.

Return type: date

Example:

```
SELECT curdate();
       curdate
-----
2022-09-19
(1 row)
```

current_time

Description: Returns the current time.

Return type: time with time zone

Example:

```
SELECT current_time;
       timetz
-----
16:58:07.086215+08
(1 row)
```

curtime([fsp])

Description: Returns the current time. **fsp** is an optional parameter. Its value is an integer, which indicates the precision of the returned result. This function is supported by version 8.2.0 or later clusters.

Return type: time with time zone

Example:

```
SELECT curtime();
      timetz
-----
16:58:07.086215+08
(1 row)
SELECT curtime(2);
      timetz
-----
16:58:07.08+08
(1 row)
```

current_timestamp

Description: Returns the current date and time.

Return type: timestamp with time zone

Example:

```
SELECT current_timestamp;
      pg_systimestamp
-----
2017-09-01 16:58:19.22173+08
(1 row)
```

convert_tz(timestamp, from_tz, to_tz)

Description: Converts the datetime value from the time zone provided by **from_tz** (text) to the time zone provided by **to_tz** (text), and returns the converted datetime. This function is supported by version 8.2.0 or later clusters.

Return type: timestamp without time zone

Example:

```
SELECT convert_tz('2018-12-25 13:25:00', '+02:00', '+08:00');
      convert_tz
-----
2018-12-25 19:25:00
(1 row)
SELECT convert_tz('2000-02-28 23:00:00', 'GMT', 'MET');
      convert_tz
-----
2000-02-29 00:00:00
(1 row)
```

datediff(date1, date2)

Description: Returns the number of days between two given dates.

Return type: integer

Example:

```
select datediff(date '2020-11-13', date '2012-10-16');
datediff
-----
      2950
(1 row)
```

date_part(text, timestamp)

Description: Obtains the precision specified by **text**.

This function is equivalent to **extract(field from timestamp)**.

Return type: double precision

Example:

```
SELECT date_part('hour', timestamp '2001-02-16 20:38:40');
date_part
-----
        20
(1 row)
```

date_part(text, interval)

Description: Obtains the precision specified by **text**. If the value is greater than 12, the remainder of the value divided by 12 is returned.

This function is equivalent to **extract(field from timestamp)**.

Return type: double precision

Example:

```
SELECT date_part('month', interval '2 years 3 months');
date_part
-----
         3
(1 row)
```

date_trunc(text, timestamp)

Description: Truncates a timestamp to the precision specified by **text**.

Return type: timestamp

Example:

```
SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40');
date_trunc
-----
2001-02-16 20:00:00
(1 row)

-- Obtain the last day of last year.
SELECT date_trunc('day', date_trunc('year', CURRENT_DATE)+ '-1');
date_trunc
-----
2022-12-31 00:00:00+08
(1 row)

-- Obtain the first day of this year.
SELECT date_trunc('year', CURRENT_DATE);
date_trunc
-----
2023-01-01 00:00:00+08
```

```
(1 row)

-- Obtain the first day of last year.
SELECT date_trunc('year',now() + '-1 year');
       date_trunc
-----
2022-01-01 00:00:00+08
(1 row)
```

trunc(timestamp)

Description: Truncates a timestamp to day.

Return type: timestamp

Example:

```
SELECT trunc(timestamp '2001-02-16
20:38:40');
       trunc
-----
2001-02-16 00:00:00
(1 row)
```

extract(field from timestamp)

Description: Obtains the value of **field** with the specified precision. For details about the valid values of **field**, see [EXTRACT](#).

Return type: double precision

Example:

```
SELECT extract(hour from timestamp '2001-02-16 20:38:40');
       date_part
-----
          20
(1 row)
```

extract(field from interval)

Description: Obtains the value of **field** with the specified precision. If the value is greater than 12, the remainder of the value divided by 12 is returned. For details about the valid values of **field**, see [EXTRACT](#).

Return type: double precision

Example:

```
SELECT extract(month from interval '2 years 3 months');
       date_part
-----
          3
(1 row)
```

day(date)

Description: Returns the number of days in the month in which **date** is located. This function is the same as the **dayofmonth** function.

Value range: 1 to 31

Return type: integer

Example:

```
select day('2020-06-28');
day
-----
28
(1 row)
```

dayofmonth(date)

Description: Returns the number of days in the month in which **date** is located.

Value range: 1 to 31

Return type: integer

Example:

```
select dayofmonth('2020-06-28');
dayofmonth
-----
28
(1 row)
```

dayofweek(date)

Description: Returns the week index corresponding to the given date, with Sunday as the start day of the week.

Value range: 1 to 7

Return type: integer

Example:

```
select dayofweek('2020-11-22');
dayofweek
-----
1
(1 row)
```

dayofyear(date)

Description: Returns the number of days of a given date in the current year.

Value range: 1 to 366

Return type: integer

Example:

```
select dayofyear('2020-02-29');
dayofyear
-----
60
(1 row)
```

hour(timestamp with time zone)

Description: Returns the hour value in a timestamp.

Return type: integer

Example:

```
SELECT hour(timestampz '2018-12-13 12:11:15+06');
hour
-----
14
(1 row)
```

isfinite(date)

Description: Checks whether a date is valid.

Return type: boolean

Example:

```
SELECT isfinite(date '2001-02-16');
isfinite
-----
t
(1 row)
SELECT isfinite(date 'infinity');
isfinite
-----
f
(1 row)
```

isfinite(timestamp)

Description: Checks whether a timestamp is valid.

Return type: boolean

Example:

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');
isfinite
-----
t
(1 row)
SELECT isfinite(timestamp 'infinity');
isfinite
-----
f
(1 row)
```

isfinite(interval)

Description: Checks whether an interval is valid.

Return type: boolean

Example:

```
SELECT isfinite(interval '4 hours');
isfinite
-----
t
(1 row)
```

justify_days(interval)

Description: Adjusts an interval so that 30-day time periods are represented as months.

Return type: interval

Example:

```
SELECT justify_days(interval '35 days');
justify_days
-----
1 mon 5 days
(1 row)
```

justify_hours(interval)

Description: Adjusts an interval so that 24-hour time periods are represented as days.

Return type: interval

Example:

```
SELECT JUSTIFY_HOURS(INTERVAL '27 HOURS');
justify_hours
-----
1 day 03:00:00
(1 row)
```

justify_interval(interval)

Description: Adjusts an interval using **justify_days** and **justify_hours**.

Return type: interval

Example:

```
SELECT JUSTIFY_INTERVAL(INTERVAL '1 MON -1 HOUR');
justify_interval
-----
29 days 23:00:00
(1 row)
```

localtime

Description: Returns the current time.

Return type: time

Example:

```
SELECT localtime AS RESULT;
result
-----
16:05:55.664681
(1 row)
```

localtimestamp

Description: Returns the current date and time.

Return type: timestamp

Example:

```
SELECT localtimestamp;
timestamp
-----
2017-09-01 17:03:30.781902
(1 row)
```

makedate(year, dayofyear)

Description: Returns a date value based on the given year and the number of days in a year.

Return type: date

Example:

```
select makedate(2020, 60);
makedate
-----
2020-02-29
(1 row)
```

maketime(hour, minute, second)

Description: Returns a value of the time type based on the given hour, minute, and second. The value of the time type in DWS ranges from 00:00:00 to 24:00:00. Therefore, this function is not available when the value of hour is greater than 24 or less than 0.

Return type: time

Example:

```
select maketime(12, 15, 30.12);
maketime
-----
12:15:30.12
(1 row)
```

microsecond(timestamp with time zone)

Description: Returns the microsecond value in the time.

Return type: integer

Example:

```
SELECT microsecond(timestampz '2018-12-13 12:11:15.123634+06');
microsecond
-----
123634
(1 row)
```

minute(timestamp with time zone)

Description: Returns the minute value in the time.

Return type: integer

Example:

```
SELECT minute(timestampz '2018-12-13 12:11:15+06');
minute
-----
11
(1 row)
```

month(date)

Description: Returns the month of a given datetime.

Return type: integer

Example:

```
select month('2020-11-30');
month
-----
11
(1 row)
```

now([fsp])

Description: Returns the start time of the transaction. The parameter determines the microsecond output precision. The default value is 6.

Return type: timestamp with time zone

Example:

```
SELECT now();
now
-----
2017-09-01 17:03:42.549426+08
(1 row)
SELECT now(3);
now
-----
2021-09-08 10:59:00.427+08
(1 row)
```

numtodsinterval(num, interval_unit)

Description: Converts a number to an interval. **num** is a numeric-typed number. **interval_unit** is a string in the following format: 'DAY' | 'HOUR' | 'MINUTE' | 'SECOND'

You can set the **IntervalStyle** parameter to **oracle** to be compatible with the interval output format of the function in the Oracle database.

Example:

```
SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
-----
100:00:00
(1 row)

SET intervalstyle = oracle;
SET
SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
-----
+0000000004 04:00:00.000000000
(1 row)
```

pg_sleep(seconds)

Description: Returns the delay of the server thread in seconds.

Return type: void

Example:

```
SELECT pg_sleep(10);
pg_sleep
```

```
-----  
(1 row)
```

period_add(P, N)

Description: Returns the date of a given period plus *N* months.

Return type: integer

Example:

```
select period_add(200801, 2);  
period_add  
-----  
200803  
(1 row)
```

period_diff(P1, P2)

Description: Returns the number of months between two given dates.

Return type: integer

```
select period_diff(200802, 200703);  
period_diff  
-----  
11  
(1 row)
```

quarter(date)

Description: Returns the quarter to which the date belongs.

Return type: integer

Example:

```
SELECT quarter(date '2018-12-13');  
quarter  
-----  
4  
(1 row)
```

second(timestamp with time zone)

Description: Returns the second in a timestamp.

Return type: integer

Example:

```
SELECT second(timestampz '2018-12-13 12:11:15+06');  
second  
-----  
15  
(1 row)
```

statement_timestamp()

Description: Returns the current date and time.

Return type: timestamp with time zone

Example:

```
SELECT statement_timestamp();
       statement_timestamp
-----
2017-09-01 17:04:39.119267+08
(1 row)
```

sysdate

Description: Returns the current date and time.

Return type: timestamp

Example:

```
SELECT sysdate;
       sysdate
-----
2017-09-01 17:04:49
(1 row)
```

timeofday()

Description: Current date and time (like **clock_timestamp**, but returned as a **text** string)

Return type: text

Example:

```
SELECT timeofday();
       timeofday
-----
Fri Sep 01 17:05:01.167506 2017 CST
(1 row)
```

transaction_timestamp()

Description: Returns the current date and time. This function is equivalent to **current_timestamp**.

Return type: timestamp with time zone

Example:

```
SELECT transaction_timestamp();
       transaction_timestamp
-----
2017-09-01 17:05:13.534454+08
(1 row)
```

from_unixtime(unix_timestamp[,format])

Description: Converts a Unix timestamp to the datetime type when the format string is set to the default value. If the format string is specified, this function converts the Unix timestamp to a string of a specified format.

Return type: timestamp (default format string) or text (specified format string)

Example:

```
SELECT from_unixtime(875996580);
       from_unixtime
-----
```

```
-----  
1997-10-04 20:23:00  
(1 row)  
SELECT from_unixtime(875996580, '%Y %D %M %h:%i:%s');  
       from_unixtime  
-----  
1997 5th October 04:23:00  
(1 row)
```

unix_timestamp([timestamp with time zone])

Description: Returns the number of seconds from '1970-01-01 00:00:00'UTC to the timestamp specified by the input parameter. If no parameter is input, this function obtains the number of seconds from '1970-01-01 00:00:00'UTC to the current time.

Return type: bigint (no parameter is input) or numeric (parameter is input)

Example:

```
SELECT unix_timestamp();  
       unix_timestamp  
-----  
1693906219  
(1 row)  
SELECT unix_timestamp('2018-09-08 12:11:13+06');  
       unix_timestamp  
-----  
1536387073.000000  
(1 row)
```

add_months(d,n)

Description: Adds a specified number of months to a date.

Return type: timestamp

Example:

```
SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11) FROM dual;  
       add_months  
-----  
2018-04-29 00:00:00  
(1 row)
```

last_day(d)

Description: Returns the last day of the month of a specified date.

- In the ORA- or TD-compatible mode, a timestamp is returned.
- In the MySQL-compatible mode, a date is returned.

Example:

```
select last_day(to_date('2017-01-01', 'YYYY-MM-DD')) AS cal_result;  
       cal_result  
-----  
2017-01-31 00:00:00  
(1 row)
```

next_day(x,y)

Description: Returns a specified day of the next week of a given date.

- In the ORA- or TD-compatible mode, a timestamp is returned.
- In the MySQL-compatible mode, a date is returned.

Example:

```
select next_day(timestamp '2017-05-25 00:00:00','Sunday')AS cal_result;
       cal_result
-----
2017-05-28 00:00:00
(1 row)
```

from_days(days)

Description: Returns the corresponding date value based on the given number of days.

Return type: date

Example:

```
select from_days(730669);
       from_days
-----
2000-07-03
(1 row)
```

to_days(timestamp)

Description: Returns the number of days from the first day of year 0 to a specified date.

Return type: integer

Example:

```
SELECT to_days(timestamp '2008-10-07');
       to_days
-----
       733687
(1 row)
```

7.5.3 EXTRACT

EXTRACT(*field* FROM *source*)

The **extract** function retrieves subcolumns such as year or hour from date/time values. **source** must be a value expression of type **timestamp**, **time**, or **interval**. (Expressions of type **date** are cast to **timestamp** and can therefore be used as well.) **field** is an identifier or string that selects what column to extract from the source value. The **extract** function returns values of type **double precision**. The following are valid field names:

century

Century

The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0. You go from **-1** century to **1** century.

Example:

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
date_part
-----
      20
(1 row)
```

day

- For **timestamp** values, the day (of the month) column (1–31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
      16
(1 row)
```

- For **interval** values, the number of days

```
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
date_part
-----
      40
(1 row)
```

decade

Year column divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
     200
(1 row)
```

dow

Day of the week as Sunday(0) to Saturday (6)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
       5
(1 row)
```

doy

Day of the year (1–365 or 366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
      47
(1 row)
```

epoch

- For **timestamp with time zone** values, the number of seconds since 1970-01-01 00:00:00 UTC (can be negative);
for **date** and **timestamp** values, the number of seconds since 1970-01-01 00:00:00 local time;
for **interval** values, the total number of seconds in the interval.

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
date_part
-----
```

```
982384720.12
(1 row)
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
date_part
-----
442800
(1 row)
```

- Way to convert an epoch value back to a timestamp

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second' AS RESULT;
result
-----
2001-02-17 12:38:40.12+08
(1 row)
```

hour

Hour column (0–23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
20
(1 row)
```

isodow

Day of the week (1–7)

Monday is 1 and Sunday is 7.

NOTE

This is identical to **dow** except for Sunday.

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
date_part
-----
7
(1 row)
```

isoyear

The ISO 8601 year that the date falls in (not applicable to intervals).

Each ISO year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. For details, see [week](#).

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
date_part
-----
2005
(1 row)
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
date_part
-----
2006
(1 row)
```

microseconds

The seconds column, including fractional parts, multiplied by 1,000,000

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
date_part
-----
28500000
(1 row)
```

millennium

Millennium

Years in the 1900s are in the second millennium. The third millennium started from January 1, 2001.

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
3
(1 row)
```

milliseconds

The seconds column, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
date_part
-----
28500
(1 row)
```

minute

Minutes column (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
38
(1 row)
```

month

For **timestamp** values, the number of the month within the year (1–12);

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
2
(1 row)
```

For **interval** values, the number of months, modulo 12 (0–11)

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
date_part
-----
1
(1 row)
```

quarter

Quarter of the year (1–4) that the date is in

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
1
(1 row)
```



```
-----  
      1  
(1 row)
```

second

Seconds column, including fractional parts (0–59)

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');  
date_part  
-----  
      28.5  
(1 row)
```

timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

```
SELECT EXTRACT(timezone FROM TIMETZ '17:12:28');  
date_part  
-----  
     28800  
(1 row)
```

timezone_hour

The hour component of the time zone offset

```
SELECT EXTRACT(timezone_hour FROM TIMETZ '17:12:28');  
date_part  
-----  
      8  
(1 row)
```

timezone_minute

The minute component of the time zone offset

```
SELECT EXTRACT(timezone_minute FROM TIMETZ '17:12:28');  
date_part  
-----  
      0  
(1 row)
```

week

The number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO-8601 week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

Because of this, it is possible for early January dates to be part of the 52nd or 53rd week of the previous year, and late December dates to be part of the 1st week of the next year. For example, **2005-01-01** is part of the 53rd week of year 2004, **2006-01-01** is part of the 52nd week of year 2005, and **2012-12-31** is part of the 1st week of year 2013. You are advised to use the columns **isoyear** and **week** together to ensure consistency.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');  
date_part  
-----
```

```
7  
(1 row)
```

year

Year column

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');  
date_part  
-----  
2001  
(1 row)
```

7.5.4 date_part

date_part

The **date_part** function is modeled on the traditional Ingres equivalent to the SQL-standard function **extract**:

`date_part('field', source)`

Note that the **field** must be a string, rather than a name. The valid field names are the same as those for **extract**. For details, see [EXTRACT](#).

Example:

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');  
date_part  
-----  
16  
(1 row)  
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');  
date_part  
-----  
4  
(1 row)
```

7.5.5 date_format

date_format(timestamp, fmt)

Converts a date into a string in the format specified by **fmt**.

Example:

```
SELECT date_format('2009-10-04 22:23:00', '%M %D %W');  
date_format  
-----  
October 4th Sunday  
(1 row)  
SELECT date_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');  
date_format  
-----  
2021-02-20 08:30:45  
(1 row)  
SELECT date_format('2021-02-20 18:10:15', '%r-%T');  
date_format  
-----  
06:10:15 PM-18:10:15  
(1 row)
```

Table 7-3 describes the patterns of date parameter values. They can be used for the **date_format**, **time_format**, **str_to_date**, **str_to_time**, and **from_unixtime** functions.

Table 7-3 Formats for the output string

Format	Description	Value
%a	Abbreviated week name	Sun...Sat
%b	Abbreviated month name	Jan...Dec
%c	Month	0...12
%D	Date with a suffix	0th, 1st, 2nd, 3rd, ...
%d	Day in a month (two digits)	00...31
%e	Day in a month	0...31
%f	Microsecond	000000...999999
%H	Hour, in 24-hour format	00...23
%h	Hour, in 12-hour format	01...12
%l	Hour, in 12-hour format, same as %h	01...12
%i	Minute	00...59
%j	Day in a year	001...366
%k	Hour, in 24-hour format, same as %H	0...23
%l	Hour, in 12-hour format, same as %h	1...12
%M	Month name	January...December
%m	Month (two digits)	00...12
%p	Morning and afternoon	AM PM
%r	Time, in 12-hour format	hh:mm:ss AM/PM
%S	Second	00...59
%s	Second, same as %S	00...59
%T	Time, in 24-hour format	hh:mm:ss
%U	Week (Sunday is the first day of a week.)	00...53
%u	Week (Monday is the first day of a week.)	00...53

Format	Description	Value
%V	Week (Sunday is the first day of a week). It is used together with %X.	01...53
%v	Week (Monday is the first day of a week). It is used together with %x.	01...53
%W	Week name	Sunday...Saturday
%w	Day of a week. The value is 0 for Sunday.	0...6
%X	Year (four digits). It is used together with %V. Sunday is the first day of a week.	-
%x	Year (four digits). It is used together with %v. Monday is the first day of a week.	-
%Y	Year (four digits)	-
%y	Year (two digits)	-
%%	Character '%'	Character '%'
%x	'x': any character apart from the preceding ones	Character 'x'

NOTICE

In the preceding table, %U, %u, %V, %v, %X, and %x are not supported currently.

7.5.6 time_format

time_format(time, fmt)

Description: The **time_format** function converts the **date** parameter into a string in the format specified by **fmt**. It is similar to the **date_format** function, but the format string can contain only hour, minute, second, and microsecond format specifiers. If other specifiers are contained, **NULL** or **0** is returned.

Return type: text

Example:

```
SELECT time_format('2009-10-04 22:23:00', '%M %D %W');  
time_format
```

(1 row)

```
SELECT time_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');
```

```
time_format
-----
0000-00-00 08:30:45
(1 row)
SELECT time_format('2021-02-20 18:10:15', '%r-%T');
time_format
-----
06:10:15 PM-18:10:15
(1 row)
```

NOTICE

time_format supports only time-related formats (%f, %H, %h, %l, %i, %k, %l, %p, %r, %S, %s, and %T) and does not support date-related formats. In other cases, **time_format** is output as common characters.

str_to_date(str, format)

Description: Converts a string of the date/time type to a value of the date type according to the provided display format.

Return type: timestamp

Example:

```
SELECT str_to_date('01,5,2021','%d,%m,%Y');
str_to_date
-----
2021-05-01 00:00:00
(1 row)
SELECT str_to_date('01,5,2021,09,30,17','%d,%m,%Y,%h,%i,%s');
str_to_date
-----
2021-05-01 09:30:17
(1 row)
```

For details about the input format types applicable to **str_to_date**, see [Table 7-3](#). Only input values of the date or date/time type can be converted. Use **str_to_time** when only values of the time type are input.

str_to_time(str, format)

Description: Converts a string of the time type to a value of the time type according to the provided display format.

Return type: time

Example:

```
SELECT str_to_time('09:30:17','%h:%i:%s');
str_to_time
-----
09:30:17
(1 row)
```

For details about the input format types applicable to **str_to_time**, see [Table 7-3](#). Only input values of the time type can be converted. Use **str_to_date** when values of the date or date/time type are input.

week(date[, mode])

Description: Returns the number of weeks in the year of the specified datetime. The default value is **0**.

Return type: integer

Table 7-4 Working principle of the mode in the week function

Sche ma	First Day of a Week	Week Range	Rule for Determining the First Week
0	Sun	0-53	Week of the first Sunday after New Year's Day
1	Mon	0-53	Week with four or more days after New Year's Day
2	Sun	1-53	Week of the first Sunday after New Year's Day
3	Mon	1-53	Week with four or more days after New Year's Day
4	Sun	0-53	Week with four or more days after New Year's Day
5	Mon	0-53	Week of the first Monday after New Year's Day
6	Sun	1-53	Week with four or more days after New Year's Day
7	Mon	1-53	Week of the first Monday after New Year's Day

Example:

```
select week('2018-01-01');
week
-----
0
(1 row)

select week('2018-01-01', 0);
week
-----
0
(1 row)

select week('2020-12-31', 1);
week
-----
53
(1 row)

select week('2020-12-31', 5);
week
-----
```

```
52  
(1 row)
```

weekday(date)

Description: Returns the week index corresponding to the given date, with Monday as the start day of the week.

Value range: 0 to 6

Return type: integer

Example:

```
select weekday('2020-11-06');  
weekday  
-----  
4  
(1 row)
```

weekofyear(date)

Description: Returns the number of weeks in the current year for the week of the given date. The value ranges from 1 to 53, which is equivalent to week(date, 3).

Return type: integer

Example:

```
select weekofyear('2020-12-30');  
weekofyear  
-----  
53  
(1 row)
```

year(date)

Description: Obtains the year of the **date**.

Return type: integer

Example:

```
select year('2020-11-13');  
year  
-----  
2020  
(1 row)
```

yearweek(date[, mode])

Description: Returns the year and the number of weeks in the current year for the given date. The number of weeks ranges from 1 to 53.

Return type: integer

Example:

```
select yearweek('2019-12-31');  
yearweek  
-----  
201952  
(1 row)
```

```
select yearweek('2019-1-1');
yearweek
-----
201852
(1 row)
```

7.6 SEQUENCE Functions

The sequence functions provide a simple method to ensure security of multiple users for users to obtain sequence values from sequence objects.

nextval(regclass)

Specifies an increasing sequence and returns a new value.

NOTE

- In order to avoid blocking concurrent transactions that retrieve values from the same sequence, the **nextval** operation does not undo its action. This means that once a value is obtained, it is considered as used and will not be given again. This applies even if the operation is part of a transaction that gets interrupted or if the calling query does not end up using the value. This means that aborted transactions may leave unused "holes" in the sequence of assigned values. Therefore, sequences in DWS cannot be used to obtain sequence without gaps.
- If the **nextval** function is pushed to DN, each DN will automatically connect to the GTM and requests the next value. For example, in the **INSERT INTO t1 SELECT xxx** statement, a column in table **t1** needs to invoke the **nextval** function. If maximum number of connections on the GTM is 8,192, this type of pushed statements occupies too many GTM connections. Therefore, the number of concurrent connections for these statements is limited to 7,000 divided by the number of cluster DNs. The other 1,192 connections are reserved for other statements.

Return type: bigint

The **nextval** function can be invoked in either of the following ways: (In example 2, the Oracle syntax is supported. Currently, the sequence name cannot contain a dot.)

Example 1:

```
SELECT nextval('seqDemo');
nextval
-----
2
(1 row)
```

Example 2:

```
SELECT seqDemo.nextval;
nextval
-----
2
(1 row)
```

currval(regclass)

Returns the last value of **nextval** for a specified sequence in the current session. If **nextval** has not been invoked for the specified sequence in the current session, an error is reported when **currval** is invoked. By default, **currval** is disabled. To enable

it, set **enable_beta_features** to **true**. After **currval** is enabled, **nextval** will not be pushed down.

Return type: bigint

The **currval** function can be invoked in either of the following ways: (In example 2, the Oracle syntax is supported. Currently, the sequence name cannot contain a dot.)

Example 1:

```
SELECT currval('seq1');
currval
-----
      2
(1 row)
```

Example 2:

```
SELECT seq1.currval seq1;
currval
-----
      2
(1 row)
```

lastval()

Returns the last value of **nextval** in the current session. This function is equivalent to **currval**, but **lastval** does not have a parameter. If **nextval** has not been invoked in the current session, an error is reported when **lastval** is invoked.

By default, **lastval** is disabled. To enable it, set **enable_beta_features** or **lastval_supported** to **true**. After **lastval** is enabled, **nextval** will not be pushed down.

Return type: bigint

For example:

```
SELECT lastval();
lastval
-----
      2
(1 row)
```

setval(regclass, bigint)

Sets the current value of a sequence.

Return type: bigint

For example:

```
SELECT setval('seqDemo',1);
setval
-----
      1
(1 row)
```

setval(regclass, bigint, boolean)

Sets the current value of a sequence and the is_called sign.

Return type: bigint

For example:

```
SELECT setval('seqDemo',1,true);
setval
-----
      1
(1 row)
```

NOTE

The current session and GTM will take effect immediately after **setval** is performed. If other sessions have buffered sequence values, **setval** will take effect only after the values are used up. Therefore, to prevent sequence value conflicts, you are advised to use **setval** with caution.

Because the sequence is non-transactional, changes made by **setval** will not be canceled when a transaction rolled back.

7.7 Array Functions and Operators

7.7.1 Array Operators

Array comparisons compare the array contents element-by-element, using the default B-tree comparison function for the element data type. In multidimensional arrays, the elements are accessed in row-major order. If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order.

=

Description: Specifies whether two arrays are equal.

Example:

```
SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT;
result
-----
      t
(1 row)
```

<>

Description: Specifies whether two arrays are not equal.

Example:

```
SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;
result
-----
      t
(1 row)
```

<

Description: Specifies whether an array is less than another.

Example:

```
SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;  
result  
-----  
t  
(1 row)
```

>

Description: Specifies whether an array is greater than another.

Example:

```
SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;  
result  
-----  
t  
(1 row)
```

<=

Description: Specifies whether an array is less than another.

Example:

```
SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;  
result  
-----  
t  
(1 row)
```

>=

Description: Specifies whether an array is greater than or equal to another.

Example:

```
SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;  
result  
-----  
t  
(1 row)
```

@>

Description: Specifies whether an array contains another.

Example:

```
SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;  
result  
-----  
t  
(1 row)
```

<@

Description: Specifies whether an array is contained in another.

Example:

```
SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;  
result  
-----
```

```
t  
(1 row)
```

&&

Description: Specifies whether an array overlaps another (have common elements).

Example:

```
SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;  
result  
-----  
t  
(1 row)
```

||

Description: Array-to-array concatenation

Example:

```
SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;  
result  
-----  
{1,2,3,4,5,6}  
(1 row)  
SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;  
result  
-----  
{{1,2,3},{4,5,6},{7,8,9}}  
(1 row)
```

||

Description: Element-to-array concatenation

Example:

```
SELECT 3 || ARRAY[4,5,6] AS RESULT;  
result  
-----  
{3,4,5,6}  
(1 row)
```

||

Description: Array-to-element concatenation

Example:

```
SELECT ARRAY[4,5,6] || 7 AS RESULT;  
result  
-----  
{4,5,6,7}  
(1 row)
```

7.7.2 Array Functions

array_append(anyarray, anyelement)

Description: Appends an element to the end of an array, and only supports dimension-1 arrays.

Return type: anyarray

Example:

```
SELECT array_append(ARRAY[1,2], 3) AS RESULT;  
result  
-----  
{1,2,3}  
(1 row)
```

array_prepend(anyelement, anyarray)

Description: Appends an element to the beginning of an array, and only supports dimension-1 arrays.

Return type: anyarray

Example:

```
SELECT array_prepend(1, ARRAY[2,3]) AS RESULT;  
result  
-----  
{1,2,3}  
(1 row)
```

array_cat(anyarray, anyarray)

Description: Concatenates two arrays, and supports multi-dimensional arrays.

Return type: anyarray

Example:

```
SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;  
result  
-----  
{1,2,3,4,5}  
(1 row)  
  
SELECT array_cat(ARRAY[[1,2],[4,5]], ARRAY[6,7]) AS RESULT;  
result  
-----  
{{1,2},{4,5},{6,7}}  
(1 row)
```

array_ndims(anyarray)

Description: Returns the number of dimensions of the array.

Return type: int

Example:

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;  
result  
-----  
2  
(1 row)
```

array_dims(anyarray)

Description: Returns a text representation of array's dimensions.

Return type: text

Example:

```
SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;  
result  
-----  
[1:2][1:3]  
(1 row)
```

array_length(anyarray, int)

Description: Returns the length of the requested array dimension.

Return type: int

Example:

```
SELECT array_length(array[1,2,3], 1) AS RESULT;  
result  
-----  
3  
(1 row)
```

array_lower(anyarray, int)

Description: Returns lower bound of the requested array dimension.

Return type: int

Example:

```
SELECT array_lower('[0:2]={1,2,3}':int[], 1) AS RESULT;  
result  
-----  
0  
(1 row)
```

array_upper(anyarray, int)

Description: Returns upper bound of the requested array dimension.

Return type: int

Example:

```
SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;  
result  
-----  
4  
(1 row)
```

array_to_string(anyarray, text [, text])

Description: Uses the first **text** as the new delimiter and the second **text** to replace **NULL** values.

Return type: text

Example:

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') AS RESULT;  
result  
-----  
1,2,3*,5  
(1 row)
```

 NOTE

In **string_to_array**, if the null-string parameter is omitted or NULL, none of the substrings of the input will be replaced by NULL.

In **array_to_string**, if the null-string parameter is omitted or NULL, any null elements in the array are simply skipped and not represented in the output string.

string_to_array(text, text [, text])

Description: Uses the second **text** as the new delimiter and the third **text** as the substring to be replaced by **NULL** values. A substring can be replaced by **NULL** values only when it is the same as the third **text**.

Return type: text[]

Example:

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
result
-----
{xx,NULL,zz}
(1 row)
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'y') AS RESULT;
result
-----
{xx,yy,zz}
(1 row)
```

 NOTE

In **string_to_array**, if the delimiter parameter is NULL, each character in the input string will become a separate element in the resulting array. If the delimiter is an empty string, then the entire input string is returned as a one-element array. Otherwise the input string is split at each occurrence of the delimiter string.

unnest(anyarray)

Description: Expands an array to a set of rows.

Return type: setof anyelement

Example:

```
SELECT unnest(ARRAY[1,2]) AS RESULT;
result
-----
1
2
(2 rows)
```

The **unnest** function is used together with the **string_to_array** array. To convert an array to columns, the statement first splits a string into arrays by comma, and then converts the arrays into columns.

```
SELECT unnest(string_to_array('a,b,c,d','')) AS RESULT;
result
-----
a
b
c
d
(4 rows)
```

interval(N, N1, N2, N3 ...)

Description: Searches for the last array index that is less than or equal to the target parameter **n** from the input integer array. If **n** is NULL, -1 is returned. The **interval()** function does not support the interval(N, N1) scenario. This function is supported by version 8.2.0 or later clusters.

Return type: int

Example:

```
SELECT INTERVAL(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11) AS RESULT;  
result  
-----  
11  
(1 row)
```

split(string, delim)

Description: Separates strings by a delimiter and returns an array. This function is supported by version 8.2.0 or later clusters.

Return type: text[]

Example:

```
SELECT SPLIT('a-b-c-d-e', '-') AS RESULT;  
result  
-----  
{a,b,c,d,e}  
(1 row)  
SELECT SPLIT('a-b-c-d-e', '-')[4] AS RESULT;  
result  
-----  
d  
(1 row)
```

7.8 Logical Operators

The usual logical operators include AND, OR, and NOT. SQL uses a three-valued logical system with true, false, and null, which represents "unknown". Their priorities are NOT > AND > OR.

Table 7-5 lists operation rules, where a and b represent logical expressions.

Table 7-5 Operation rules

a	b	a AND b Result	a OR b Result	NOT a Result
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE

a	b	a AND b Result	a OR b Result	NOT a Result
NULL	NULL	NULL	NULL	NULL

 NOTE

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

7.9 Comparison Operators

Comparison operators can be used for all data types. The return results are Boolean values.

All comparison operators are binary operators. Only data types that are the same or can be implicitly converted can be compared using comparison operators. Expressions such as "1<2<3" are invalid because Boolean values cannot be compared with 3.

Table 7-6 describes comparison operators provided by DWS.

Table 7-6 Comparison operators

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equality
<> or !=	Inequality

7.10 Pattern Matching Operators

There are three separate approaches to pattern matching provided by the database: the traditional SQL LIKE operator, the more recent SIMILAR TO operator, and POSIX-style regular expressions. Besides these basic operators, functions can be used to extract or replace matching substrings and to split a string at matching locations.

LIKE

Checks whether the string matches the pattern string following **LIKE**. If the string matches the supplied pattern, the LIKE expression returns true (the NOT LIKE

expression returns false). Otherwise, the LIKE expression returns false (the NOT LIKE expression returns true).

- Rule
 - a. This operator can succeed only when its pattern matches the entire string. If you want to match a sequence in any position within the string, the pattern must begin and end with a percent sign.
 - b. The underscore (_) represents (matching) any single character. Percentage (%) indicates the wildcard character of any string.
 - c. To match a literal underscore (_) or percent sign (%) without matching other characters, the respective character in pattern must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause.
 - d. To match the escape character itself, write two escape characters. For example: To write a **pattern** constant containing a backslash (\), you need to enter two backslashes in SQL statements.

NOTE

When **standard_conforming_strings** is set to **off**, any backslashes you write in literal string constants will need to be doubled. Therefore, writing a pattern matching a single backslash is actually going to write four backslashes in the statement. You can avoid this by selecting a different escape character by using ESCAPE, so that the backslash is no longer a special character of LIKE. But the backslash is still the special character of the character text analyzer, so you still need two backslashes. You can also select no escape character by writing **ESCAPE ''**. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

- e. The keyword ILIKE can be used instead of LIKE to make the match case-insensitive.
 - f. Operator ~~ is equivalent to LIKE, and operator ~~* corresponds to ILIKE.
- Examples

```
SELECT 'abc' LIKE 'abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE 'a%' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE '_b_' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE 'c' AS RESULT;
result
-----
f
(1 row)
```

SIMILAR TO

The **SIMILAR TO** operator determines whether to match a given string based on its own pattern and returns **true** or **false**. It is similar to LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression.

- Matching rule
 - a. Like LIKE, the SIMILAR TO operator returns true only if its pattern matches the given string. If you want to match a sequence in any position within the string, the pattern must begin and end with a percent sign.
 - b. The underscore (_) represents (matching) any single character. Percentage (%) indicates the wildcard character of any string.
 - c. SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

Table 7-7 Pattern matching metacharacter

Metacharacter	Description
	Specifies alternation (either of two alternatives).
*	Specifies repetition of the previous item zero or more times.
+	Specifies repetition of the previous item one or more times.
?	Specifies repetition of the previous item zero or one time.
{m}	Specifies repetition of the previous item exactly <i>m</i> times.
{m,}	Specifies repetition of the previous item <i>m</i> or more times.
{m,n}	Specifies repetition of the previous item at least <i>m</i> times and does not exceed <i>n</i> times.
()	Specifies that parentheses () can be used to group items into a single logical item.
[...]	Specifies a character class, just as in POSIX regular expressions.

- d. A preamble escape character disables the special meaning of any of these metacharacters. The rules for using escape characters are the same as those for LIKE.
- Precautions

If a large number of characters are repeatedly matched in the SIMILAR TO regular expression, the statement fails to be executed and error "invalid regular expression: regular expression is too complex" is reported due to the recursion size restriction. In this case, increase the value of [max_stack_depth](#).
 - Regular expression functions

The **substring(string from pattern for escape)** function can be used to intercept a substring that matches an SQL regular expression.

- Examples

```
SELECT 'abc' SIMILAR TO 'abc' AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 'abc' SIMILAR TO 'a' AS RESULT;  
result  
-----  
f  
(1 row)  
SELECT 'abc' SIMILAR TO '%(b|d)%' AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 'abc' SIMILAR TO '(b|c)%' AS RESULT;  
result  
-----  
f  
(1 row)
```

POSIX regular expressions

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). If a string is a member of a regular expression described by a regular expression, the string matches the regular expression. POSIX regular expressions provide a more powerful means for pattern matching than the LIKE and SIMILAR TO operators. [Table 7-8](#) lists all available operators for POSIX regular expression pattern matching.

Table 7-8 Regular expression match operators

Operator	Description	Example
~	Matches regular expression, which is case-sensitive.	'thomas' ~ '.*thomas.*'
~*	Matches regular expression, which is case-insensitive.	'thomas' ~* '.*Thomas.*'
! ~	Does not match regular expression, which is case-sensitive.	'thomas' !~ '.*Thomas.*'
! ~*	Does not match regular expression, which is case-insensitive.	'thomas' !~* '.*vadim.*'

- Matching rule

- Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

- b. Besides the metacharacters mentioned above, POSIX regular expressions also support the following pattern matching metacharacters:

Table 7-9 Pattern matching metacharacters

Metacharacter	Description
^	Specifies the match starting with a string.
\$	Specifies the match at the end of a string.
.	Matches any single character.

- Regular expression functions

POSIX regular expressions support the following functions:

- The **substring(string from pattern)** function provides a method for extracting a substring that matches the POSIX regular expression pattern.
- The **regexp_replace(string, pattern, replacement [, flags])** function provides the function of replacing the substring matching the POSIX regular expression pattern with the new text.
- The **regexp_matches(string text, pattern text [, flags text])** function returns a text array consisting of all captured substrings that match a POSIX regular expression pattern.
- The **regexp_split_to_table(string text, pattern text [, flags text])** function splits a string using a POSIX regular expression pattern as a delimiter.
- The **regexp_split_to_array(string text, pattern text [, flags text])** function behaves the same as **regexp_split_to_table**, except that **regexp_split_to_array** returns its result as an array of text.

 NOTE

The regular expression split functions ignore zero-length matches, which occur at the beginning or end of a string or after the previous match. This is contrary to the strict definition of regular expression matching. The latter is implemented by **regexp_matches**, but the former is usually the most commonly used behavior in practice.

- Examples

```
SELECT 'abc' ~ 'Abc' AS RESULT;
result
-----
f
(1 row)
SELECT 'abc' ~* 'Abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' !~ 'Abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' !~* 'Abc' AS RESULT;
result
```

```
-----  
f  
(1 row)  
SELECT 'abc' ~ '^a' AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 'abc' ~ '(b|d)' AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 'abc' ~ '^ (b|c)' AS RESULT;  
result  
-----  
f  
(1 row)
```

Although most regular expression searches can be executed quickly, the time and memory for regular expression processing can still be manually controlled. Avoid using regular expression search patterns from untrusted sources. If required, set a statement timeout limit. The search with the SIMILAR TO mode has the same security risks as the SIMILAR TO provides many capabilities that are the same as those of the POSIX- style regular expression. For better security with unreliable inputs, prefer the simpler LIKE search over complex alternatives.

7.11 Aggregate Functions

sum(expression)

Description: Sum of expression across all input values

Return type:

Generally, same as the argument data type. In the following cases, type conversion occurs:

- **BIGINT** for **SMALLINT** or **INT** arguments
- **NUMBER** for **BIGINT** arguments
- **DOUBLE PRECISION** for floating-point arguments

Example:

```
postgres=#SELECT SUM(ss_ext_tax) FROM tpeds.STORE_SALES;  
sum  
-----  
213267594.69  
(1 row)
```

max(expression)

Description: Specifies the maximum value of **expression** across all input values.

Argument types: any array, numeric, string, or date/time type

Return type: same as the argument type

Example:

```
postgres=#SELECT MAX(inv_quantity_on_hand) FROM tpcds.inventory;  
max  
-----  
1000000  
(1 row)
```

min(expression)

Description: Specifies the minimum value of **expression** across all input values.

Argument types: any array, numeric, string, or date/time type

Return type: same as the argument type

Example:

```
postgres=#SELECT MIN(inv_quantity_on_hand) FROM tpcds.inventory;  
min  
-----  
0  
(1 row)
```

avg(expression)

Description: Average (arithmetic mean) of all input values

Return type:

- **NUMBER** for any integer-type argument.
- **DOUBLE PRECISION** for floating-point parameters.
- Other values are the same as the input data type.

Example:

```
postgres=#SELECT AVG(inv_quantity_on_hand) FROM tpcds.inventory;  
avg  
-----  
500.0387129084044604  
(1 row)
```

median(expression)

Description: Median of all input values. Currently, only the numeric and interval types are supported. Null values are not used for calculation.

Return type:

- If all input values are integers, a median of the **NUMERIC** type is returned; otherwise, a median of the same type as the input values is returned.
- In the Teradata-compatible mode, if the input values are integers, the returned median is rounded to the nearest integer.

Example:

```
postgres=#SELECT MEDIAN(inv_quantity_on_hand) FROM tpcds.inventory;  
median  
-----  
500  
(1 row)
```

percentile_cont(const) within group(order by expression)

Description: returns a value corresponding to the specified percentile in the ordering, interpolating between adjacent input items if needed. Null values are not used for calculation.

Input: *const* indicates a number ranging from 0 to 1. Currently, only numeric and interval expressions are supported.

Return type:

- If all input values are integers, a median of the **NUMERIC** type is returned; otherwise, a median of the same type as the input values is returned.
- In the Teradata-compatible mode, if the input values are integers, the returned median is rounded to the nearest integer.

Example:

```
postgres=#SELECT percentile_cont(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----
2.2
(1 row)
postgres=#SELECT percentile_cont(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----
3.8
(1 row)
```

percentile_disc(const) within group(order by expression)

Description: returns the first input value whose position in the ordering equals or exceeds the specified percentile.

Input: *const* indicates a number ranging from 0 to 1. Currently, only numeric and interval expressions are supported. Null values are not used for calculation.

Return type: If all input values are integers, a median of the **NUMERIC** type is returned; otherwise, a median of the same type as the input values is returned.

Example:

```
postgres=#SELECT percentile_disc(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----
2
(1 row)
postgres=#SELECT percentile_disc(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----
4
(1 row)
```

count(expression)

Description: Number of input rows for which the value of expression is not null

Return type: bigint

Example:

```
postgres=#SELECT COUNT(inv_quantity_on_hand) FROM tpcds.inventory;
count
-----
11158087
(1 row)
```

count(*)

Description: Number of input rows

Return type: bigint

Example:

```
postgres=#SELECT COUNT(*) FROM tpcds.inventory;
count
-----
11745000
(1 row)
```

array_agg(expression)

Description: Input values, including nulls, concatenated into an array The input parameters of the function do not support the array format.

Return type: array of the argument type

Example:

Create the **employeeinfo** table and insert data into the table:

```
CREATE TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate date,deptno smallint);
INSERT INTO employeeinfo VALUES (7155, 'JACK', 'SALESMAN', '2018-12-01', 30);
INSERT INTO employeeinfo VALUES (7003, 'TOM', 'FINANCE', '2016-06-15', 20);
INSERT INTO employeeinfo VALUES (7357, 'MAX', 'SALESMAN', '2020-10-01', 30);

SELECT * FROM employeeinfo;
 empno | ename | job   | hiredate       | deptno
-----+-----+-----+-----+-----
 7155 | JACK | SALESMAN | 2018-12-01 00:00:00 |    30
 7357 | MAX  | SALESMAN | 2020-10-01 00:00:00 |    30
 7003 | TOM  | FINANCE  | 2016-06-15 00:00:00 |    20
(3 rows)
```

Query the names of all employees in the department whose ID is **30**:

```
SELECT array_agg(ename) FROM employeeinfo where deptno = 30;
array_agg
-----
{JACK,MAX}
(1 row)
```

Query all employees in the same department:

```
SELECT deptno, array_agg(ename) FROM employeeinfo group by deptno;
deptno | array_agg
-----+-----
    30 | {JACK,MAX}
    20 | {TOM}
(2 rows)

SELECT distinct array_agg(ename) OVER (PARTITION BY deptno) FROM employeeinfo;
array_agg
-----
```

```
{TOM}  
{JACK,MAX}  
(2 rows)
```

Query all department IDs and deduplicate them:

```
SELECT array_agg(distinct deptno) FROM employeeinfo group by deptno;  
array_agg  
-----  
{20}  
{30}  
(2 rows)
```

Sort the deduplicated department IDs in descending order:

```
SELECT array_agg(distinct deptno order by deptno desc) FROM employeeinfo;  
array_agg  
-----  
{30,20}  
(1 row)
```

string_agg(expression, delimiter)

Description: Input values concatenated into a string, separated by delimiter

Return type: same as the argument type

Example:

Query all employees in the same department based on the created table **employeeinfo**:

```
SELECT deptno, string_agg(ename,') FROM employeeinfo group by deptno;  
deptno | string_agg  
-----+-----  
30 | JACK,MAX  
20 | TOM  
(2 rows)
```

Query employees whose work IDs are smaller than 7156:

```
SELECT string_agg(ename,') FROM employeeinfo where empno < 7156;  
string_agg  
-----  
TOM,JACK  
(1 row)
```

listagg(expression [, delimiter]) WITHIN GROUP(ORDER BY order-list)

Description: Aggregation column data sorted according to the mode specified by **WITHIN GROUP**, and concatenated to a string using the specified delimiter

- **expression**: Mandatory. It specifies an aggregation column name or a column-based, valid expression. It does not support the **DISTINCT** keyword and the **VARIADIC** parameter.
- **delimiter**: Optional. It specifies a delimiter, which can be a string constant or a deterministic expression based on a group of columns. The default value is empty.
- **order-list**: Mandatory. It specifies the sorting mode in a group.

Return type: text

 NOTE

listagg is a column-to-row aggregation function, compatible with Oracle Database 11g Release 2. You can specify the **OVER** clause as a window function. When **listagg** is used as a window function, the **OVER** clause does not support the window sorting or framework of **ORDER BY**, so as to avoid ambiguity in **listagg** and **ORDER BY** of the **WITHIN GROUP** clause.

Example:

The aggregation column is of the text character set type:

```
postgres=#SELECT deptno, listagg(ename, ',') WITHIN GROUP(ORDER BY ename) AS employees FROM
emp GROUP BY deptno;
 deptno |          employees
-----+-----
      10 | CLARK,KING,MILLER
      20 | ADAMS,FORD,JONES,SCOTT,SMITH
      30 | ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
(3 rows)
```

The aggregation column is of the integer type:

```
postgres=#SELECT deptno, listagg(mgrno, ',') WITHIN GROUP(ORDER BY mgrno NULLS FIRST) AS mgrnos
FROM emp GROUP BY deptno;
 deptno |          mgrnos
-----+-----
      10 | 7782,7839
      20 | 7566,7566,7788,7839,7902
      30 | 7698,7698,7698,7698,7698,7839
(3 rows)
```

The aggregation column is of the floating point type:

```
postgres=#SELECT job, listagg(bonus, '($); ') WITHIN GROUP(ORDER BY bonus DESC) || '($)' AS bonus
FROM emp GROUP BY job;
 job |          bonus
-----+-----
CLERK | 10234.21($); 2000.80($); 1100.00($); 1000.22($)
PRESIDENT | 23011.88($)
ANALYST | 2002.12($); 1001.01($)
MANAGER | 10000.01($); 2399.50($); 999.10($)
SALESMAN | 1000.01($); 899.00($); 99.99($); 9.00($)
(5 rows)
```

The aggregation column is of the time type:

```
postgres=#SELECT deptno, listagg(hiredate, ', ') WITHIN GROUP(ORDER BY hiredate DESC) AS hiredates
FROM emp GROUP BY deptno;
 deptno |          hiredates
-----+-----
      10 | 1982-01-23 00:00:00, 1981-11-17 00:00:00, 1981-06-09 00:00:00
      20 | 2001-04-02 00:00:00, 1999-12-17 00:00:00, 1987-05-23 00:00:00, 1987-04-19 00:00:00, 1981-12-03
00:00:00
      30 | 2015-02-20 00:00:00, 2010-02-22 00:00:00, 1997-09-28 00:00:00, 1981-12-03 00:00:00, 1981-09-08
00:00:00, 1981-05-01 00:00:00
(3 rows)
```

The aggregation column is of the time interval type.

```
postgres=#SELECT deptno, listagg(vacationTime, ', ') WITHIN GROUP(ORDER BY vacationTime DESC) AS
vacationTime FROM emp GROUP BY deptno;
 deptno |          vacationtime
-----+-----
      10 | 1 year 30 days; 40 days; 10 days
      20 | 70 days; 36 days; 9 days; 5 days
```

```
30 | 1 year 1 mon; 2 mons 10 days; 30 days; 12 days 12:00:00; 4 days 06:00:00; 24:00:00
(3 rows)
```

By default, the delimiter is empty:

```
postgres=#SELECT deptno, listagg(job) WITHIN GROUP(ORDER BY job) AS jobs FROM emp GROUP BY
deptno;
deptno | jobs
-----+-----
10 | CLERKMANAGERPRESIDENT
20 | ANALYSTANALYSTCLERKCLERKMANAGER
30 | CLERKMANAGERSALESMANSALESMANSALESMANSALSMAN
(3 rows)
```

When **listagg** is used as a window function, the **OVER** clause does not support the window sorting of **ORDER BY**, and the **listagg** column is an ordered aggregation of the corresponding groups.

```
postgres=#SELECT deptno, mgrno, bonus, listagg(ename, ';' ) WITHIN GROUP(ORDER BY hiredate)
OVER(PARTITION BY deptno) AS employees FROM emp;
deptno | mgrno | bonus | employees
-----+-----+-----+-----
10 | 7839 | 10000.01 | CLARK; KING; MILLER
10 | 23011.88 | CLARK; KING; MILLER
10 | 7782 | 10234.21 | CLARK; KING; MILLER
20 | 7566 | 2002.12 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7566 | 1001.01 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7788 | 1100.00 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7902 | 2000.80 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7839 | 999.10 | FORD; SCOTT; ADAMS; SMITH; JONES
30 | 7839 | 2399.50 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 9.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 1000.22 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 99.99 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 1000.01 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 899.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
(14 rows)
```

group_concat(expression [ORDER BY {col_name | expr} [ASC | DESC]] [SEPARATOR str_val])

Description: concatenates the specified **str_val** delimiters used by column data into a string. The concatenation uses a sorting method that must be specified by the **ORDER BY** clause. **ORDER BY 1** is not allowed.

- **expression**: (mandatory) specifies a column name or a column-based valid expression. It does not support the **DISTINCT** keyword or the **VARIADIC** parameter.
- **str_val**: (optional) specifies a delimiter, which can be a string constant or a deterministic expression based on grouped columns. The default value indicates that commas (,) are used as delimiters.

Return type: text

NOTE

The group_concat function is supported only in 8.1.2 or later.

Example:

The default delimiter is a comma (,).

```
postgres=#SELECT group_concat(sname) FROM group_concat_test;
group_concat
```

```
-----  
ADAMS,FORD,JONES,KING,MILLER,SCOTT,SMITH  
(1 row)
```

Delimiters can be customized for the **group_concat** function.

```
postgres=#SELECT group_concat(sname separator ';') from group_concat_test;  
group_concat  
-----  
ADAMS;FORD;JONES;KING;MILLER;SCOTT;SMITH  
(1 row)
```

The **group_concat** function supports the **ORDER BY** clause, which concatenates column data in sequence.

```
postgres=#SELECT group_concat(sname order by snumber separator ';') FROM group_concat_test;  
group_concat  
-----  
MILLER;FORD;SCOTT;SMITH;KING;JONES;ADAMS  
(1 row)
```

covar_pop(Y, X)

Description: Overall covariance

Return type: double precision

Example:

```
postgres=# SELECT COVAR_POP(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk <  
1000;  
covar_pop  
-----  
829.749627587403  
(1 row)
```

covar_samp(Y, X)

Description: Sample covariance

Return type: double precision

Example:

```
postgres=#SELECT COVAR_SAMP(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk <  
1000;  
covar_samp  
-----  
830.052235037289  
(1 row)
```

stddev_pop(expression)

Description: Overall standard difference

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT STDDEV_POP(inv_quantity_on_hand) FROM tpods.inventory WHERE inv_warehouse_sk =  
1;  
stddev_pop  
-----  
289.224294957556  
(1 row)
```

stddev_samp(expression)

Description: Sample standard deviation of the input values

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT STDDEV_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk
= 1;
 stddev_samp
-----
289.224359757315
(1 row)
```

var_pop(expression)

Description: Population variance of the input values (square of the population standard deviation)

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT VAR_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
  var_pop
-----
83650.692793695475
(1 row)
```

var_samp(expression)

Description: Sample variance of the input values (square of the sample standard deviation)

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT VAR_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
  var_samp
-----
83650.730277028768
(1 row)
```

bit_and(expression)

Description: The bitwise AND of all non-null input values, or null if none

Return type: same as the argument type

Example:

```
postgres=#SELECT BIT_AND(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_and
-----
0
(1 row)
```

bit_or(expression)

Description: The bitwise OR of all non-null input values, or null if none

Return type: same as the argument type

Example:

```
postgres=#SELECT BIT_OR(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
bit_or
-----
1023
(1 row)
```

bool_and(expression)

Description: Its value is **true** if all input values are **true**, otherwise **false**.

Return type: Boolean

Example:

```
postgres=#SELECT bool_and(100 <2500);
bool_and
-----
t
(1 row)
```

bool_or(expression)

Description: Its value is **true** if at least one input value is **true**, otherwise **false**.

Return type: Boolean

Example:

```
postgres=#SELECT bool_or(100 <2500);
bool_or
-----
t
(1 row)
```

corr(Y, X)

Description: Correlation coefficient

Return type: double precision

Example:

```
postgres=#SELECT CORR(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
corr
-----
0.0381383624904186
(1 row)
```

every(expression)

Description: Equivalent to **bool_and**

Return type: Boolean

Example:

```
postgres=#SELECT every(100 <2500);
every
-----
```

```
t
(1 row)
```

rank(expression)

Description: The tuples in different groups are sorted non-consecutively by **expression**.

Return type: bigint

Example:

```
postgres=#SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq)
FROM tpceds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
 d_moy | d_fy_week_seq | rank
```

```
-----+-----+-----
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      2 |      8
1 |      2 |      8
1 |      2 |      8
1 |      2 |      8
1 |      2 |      8
1 |      2 |      8
1 |      2 |      8
1 |      3 |     15
1 |      3 |     15
1 |      3 |     15
1 |      3 |     15
1 |      3 |     15
1 |      3 |     15
1 |      3 |     15
1 |      4 |     22
1 |      4 |     22
1 |      4 |     22
1 |      4 |     22
1 |      4 |     22
1 |      4 |     22
1 |      4 |     22
1 |      5 |     29
1 |      5 |     29
2 |      5 |      1
2 |      5 |      1
2 |      5 |      1
2 |      5 |      1
2 |      5 |      1
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
2 |      6 |      6
(42 rows)
```

regr_avgx(Y, X)

Description: Average of the independent variable (**sum(X)/N**)

Return type: double precision

Example:


```
postgres=#SELECT REGR_AVGX(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_avgx
-----
578.606576740795
(1 row)
```

regr_avgy(Y, X)

Description: Average of the dependent variable ($\text{sum}(Y)/N$)

Return type: double precision

Example:

```
postgres=#SELECT REGR_AVGY(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_avgy
-----
50.0136711629602
(1 row)
```

regr_count(Y, X)

Description: Number of input rows in which both expressions are non-null

Return type: bigint

Example:

```
postgres=#SELECT REGR_COUNT(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_count
-----
2743
(1 row)
```

regr_intercept(Y, X)

Description: y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs

Return type: double precision

Example:

```
postgres=#SELECT REGR_INTERCEPT(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_intercept
-----
49.2040847848607
(1 row)
```

regr_r2(Y, X)

Description: Square of the correlation coefficient

Return type: double precision

Example:

```
postgres=#SELECT REGR_R2(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_r2
-----
```

```
0.00145453469345058
(1 row)
```

regr_slope(Y, X)

Description: Slope of the least-squares-fit linear equation determined by the (X, Y) pairs

Return type: double precision

Example:

```
postgres=#SELECT REGR_SLOPE(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
regr_slope
-----
0.00139920009665259
(1 row)
```

regr_sxx(Y, X)

Description: $\text{sum}(X^2) - \text{sum}(X)^2/N$ (sum of squares of the independent variables)

Return type: double precision

Example:

```
postgres=#SELECT REGR_SXX(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
regr_sxx
-----
1626645991.46135
(1 row)
```

regr_sxy(Y, X)

Description: $\text{sum}(X*Y) - \text{sum}(X) * \text{sum}(Y)/N$ ("sum of products" of independent times dependent variable)

Return type: double precision

Example:

```
postgres=#SELECT REGR_SXY(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
regr_sxy
-----
2276003.22847225
(1 row)
```

regr_syy(Y, X)

Description: $\text{sum}(Y^2) - \text{sum}(Y)^2/N$ ("sum of squares" of the dependent variable)

Return type: double precision

Example:

```
postgres=#SELECT REGR_SYY(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
regr_syy
-----
2189417.6547314
(1 row)
```

stddev(expression)

Description: Alias of **stddev_samp**

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT STDDEV(inv_quantity_on_hand) FROM tpeds.inventory WHERE inv_warehouse_sk = 1;
stddev
-----
289.224359757315
(1 row)
```

variance(expression)

Description: Alias of **var_samp**

Return type: **double precision** for floating-point arguments, otherwise **numeric**

Example:

```
postgres=#SELECT VARIANCE(inv_quantity_on_hand) FROM tpeds.inventory WHERE inv_warehouse_sk = 1;
variance
-----
83650.730277028768
(1 row)
```

checksum(expression)

Description: Returns the CHECKSUM value of all input values. This function can be used to check whether the data in the tables is the same before and after the backup, restoration, or migration of the DWS database. Before and after database backup, database restoration, or data migration, you need to manually run SQL commands to obtain the execution results. Compare the obtained execution results to check whether the data in the tables before and after the backup or migration is the same.

NOTE

- For large tables, the CHECKSUM function may take a long time.
- If the CHECKSUM values of two tables are different, it indicates that the contents of the two tables are different. Using the hash function in the CHECKSUM function may incur conflicts. There is low possibility that two tables with different contents may have the same CHECKSUM value. The same problem may occur when CHECKSUM is used for columns.
- If the time type is timestamp, timestamptz, or smalldatetime, ensure that the time zone settings are the same when calculating the CHECKSUM value.
- If the CHECKSUM value of a column is calculated and the column type can be changed to TEXT by default, set *expression* to the column name.
- If the CHECKSUM value of a column is calculated and the column type cannot be changed to TEXT by default, set *expression* to *Column name::TEXT*.
- If the CHECKSUM value of all columns is calculated, set *expression* to *Table name::TEXT*.

The following types of data can be converted into TEXT types by default: char, name, int8, int2, int1, int4, raw, pg_node_tree, float4, float8, bpchar, varchar, nvarchar2, date, timestamp, timestamptz, numeric, and smalldatetime. Other types need to be forcibly converted to TEXT.

Return type: numeric

Example:

The following shows the CHECKSUM value of a column that can be converted to the TEXT type by default:

```
postgres=#SELECT CHECKSUM(inv_quantity_on_hand) FROM tpcds.inventory;
checksum
-----
24417258945265247
(1 row)
```

CHECKSUM value of a column that cannot be converted to the TEXT type by default (Note that the CHECKSUM parameter is *column_name::TEXT*):

```
postgres=#SELECT CHECKSUM(inv_quantity_on_hand::TEXT) FROM tpcds.inventory;
checksum
-----
24417258945265247
(1 row)
```

The following shows the CHECKSUM value of all columns in a table. Note that the CHECKSUM parameter is set to *Table name::TEXT*. The table name is not modified by its schema.

```
postgres=#SELECT CHECKSUM(inventory::TEXT) FROM tpcds.inventory;
checksum
-----
25223696246875800
(1 row)
```

approx_count_distinct(col_name)

Description: Uses the HyperLogLog++ (HLL++) algorithm to estimate the cardinality (number of rows in a column after deduplication). This function is supported only by clusters of version 8.3.0 or later.

Input parameter description: **col_name** indicates the column whose cardinality needs to be estimated.

NOTE

You can adjust the error rate by setting the **approx_count_distinct_precision** parameter.

- The value range is [10,20]. The default value is 17. The theoretical error rate is 3%.
- This parameter indicates the number of buckets in the HyperLogLog++ (HLL++) algorithm. A larger value of this parameter indicates a larger number of buckets and a smaller theoretical error rate.
- Increasing this parameter value will increase the computing time and memory usage, but they will still be much lower than the cost of using a **COUNT DISTINCT** query. You are advised to use this function to replace the **COUNT DISTINCT** statement when the estimation cardinality is large.

Example:

```
CREATE TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate date,deptno
smallint) WITH (ORIENTATION = COLUMN);
INSERT INTO employeeinfo VALUES (7155, 'JACK', 'SALESMAN', '2018-12-01', 30);
INSERT INTO employeeinfo VALUES (7003, 'TOM', 'FINANCE', '2016-06-15', 20);
INSERT INTO employeeinfo VALUES (7357, 'MAX', 'SALESMAN', '2020-10-01', 30);

SELECT APPROX_COUNT_DISTINCT(empno) from employeeinfo;
approx_count_distinct
```

```
-----
              3
(1 row)

SELECT COUNT(DISTINCT empno) FROM employeeinfo GROUP BY ename;
count
-----
    1
    1
    1
(3 rows)
```

UNIQ(col_name)

Description: Aggregate function for calculating the number of non-duplicated values. Similar to **COUNT DISTINCT**, this function calculates the number of rows in a column after deduplication and returns a value indicating the rows without duplication. This function is supported only by clusters of version 8.3.0 or later.

Input parameter description: **col_name** indicates the column for which the number of rows after deduplication needs to be calculated. The value can be SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, TEXT, VARCHAR, TIMESTAMP, TIMESTAMPTZ, DATE, TIMETZ or UUID.

NOTE

- Currently, only DWS column-store tables support the **UNIQ** function.
- When the **UNIQ** function is used, the SQL statement must contain **GROUP BY**. To achieve better performance, the GROUP BY fields must be evenly distributed.
- To deduplicate three or more columns in an SQL statement, you can use the **UNIQ** function, which is more efficient than the **COUNT DISTINCT** function.
- Generally, the memory usage of the **UNIQ** function is lower than that of the **COUNT DISTINCT** function. If the memory usage exceeds the threshold when the **COUNT DISTINCT** function is used, you can use the **UNIQ** function to reduce the memory usage.

Example:

```
CREATE TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate date, deptno
smallint) WITH (ORIENTATION = COLUMN);
INSERT INTO employeeinfo VALUES (7155, 'JACK', 'SALESMAN', '2018-12-01', 30);
INSERT INTO employeeinfo VALUES (7003, 'TOM', 'FINANCE', '2016-06-15', 20);
INSERT INTO employeeinfo VALUES (7357, 'MAX', 'SALESMAN', '2020-10-01', 30);

SELECT UNIQ(deptno) FROM employeeinfo GROUP BY ename;
uniq
-----
    1
    1
    1
(3 rows)
```

7.12 Window Functions

Regular aggregate functions return a single value calculated from values in a row, or group all rows into a single output row. Window functions perform a calculation across a set of rows and return a value for each row.

A window function call represents the application of an aggregate-like function over some portion of the rows selected by a query. Therefore, aggregate functions

([Aggregate Functions](#)) can also be used as window functions. A window function can scan all rows and display the raw data and aggregation analysis results at the same time.

Precautions

- Column-store tables support only the window functions **rank (expression)** and **row_number (expression)** and the aggregate functions **sum**, **count**, **avg**, **min**, and **max**. Row-store tables do not have such restrictions.
- A single query can contain one or more window function expressions.
- Window functions can appear only in output columns. If you want to use the values of a window function for condition filtering, you need to nest the window function in the subquery and use the aliases of the window function expression at the outer layer for condition filtering. Example:

```
SELECT classid, id, score FROM (SELECT *, avg(score) OVER(PARTITION BY classid) as avg_score FROM score) WHERE score >= avg_score;
```
- In the query block where the window function is located, the **GROUP BY** expression can be used for grouping and deduplication. In this case, the **PARTITION BY** clause in the window function must be a subset of the **GROUP BY** expression to ensure that the window function performs calculation on the deduplication result. The expression of the **ORDER BY** clause must be a subset of the **GROUP BY** expression or an aggregate function of an aggregate operation. Example:

```
SELECT classid, rank() OVER(PARTITION BY classid ORDER BY sum(score)) as avg_score FROM score GROUP BY classid, id;
```

Syntax

A window function uses the **OVER** clause to define a window. The **OVER** clause is used for grouping data and sorting the elements in a group. Window functions are used for generating sequence numbers for the values in the group.

```
function_name ([expression [, expression ...]]) OVER ( window_definition )  
function_name ([expression [, expression ...]]) OVER window_name  
function_name ( * ) OVER ( window_definition )  
function_name ( * ) OVER window_name
```

window_definition is defined as follows:

```
[ existing_window_name ]  
[ PARTITION BY expression [, ...] ]  
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]  
[ frame_clause ]
```

The **PARTITION BY** option specifies that rows with the same **PARTITION BY** expression value are grouped.

The **ORDER BY** option is used to control the order in which the window function processes rows. **ORDER BY** must be followed by a column name. If **ORDER BY** is followed by a number, the number is processed as a constant and does not sort the target column.

frame_clause is defined as follows:

```
[ RANGE | ROWS ] frame_start  
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

When you need to specify a window to calculate the results of all rows in a group, you need to specify the start row and end row of the window range. The window

range supports the RANGE and ROWS modes. The ROWS mode specifies the window by the physical unit (row), and the RANGE mode specifies the window as the logical offset.

In **RANGE** and **ROWS**, you can use **BETWEEN** *frame_start* **AND** *frame_end* to specify the window's first and last rows. If only **frame_start** is specified, **frame_end** is **CURRENT ROW** by default.

The values of **frame_start** and **frame_end** are as follows:

- **CURRENT ROW**: The current row is used as the window frame's start or end point.
- ***N* PRECEDING**: The window frame starts from the *n*th row to the current row.
- **UNBOUNDED PRECEDING**: The window frame starts at the first row of the partition.
- ***N* FOLLOWING**: The window frame starts from the current row to the *n*th row.
- **UNBOUNDED FOLLOWING**: The window frame ends with the last row of the partition.

frame_start cannot be **UNBOUNDED FOLLOWING**, *frame_end* cannot be **UNBOUNDED PRECEDING**, and *frame_end* cannot be earlier than *frame_start*. For example, **RANGE BETWEEN CURRENT ROW AND *N* PRECEDING** is not allowed.

RANK()

Description: The **RANK** function is used for generating non-consecutive sequence numbers for the values in each group. The same values have the same rank value but with sequence numbers.

Return type: bigint

Example:

In the **score(id, classid, score)** table, the rows are student ID, class ID, and exam score.

Use the **RANK** function to sort student scores.

```
CREATE TABLE score(id int,classid int,score int);
INSERT INTO score VALUES(1,1,95),(2,2,95),(3,2,85),(4,1,70),(5,2,88),(6,1,70);
```

```
SELECT id, classid, score,RANK() OVER(ORDER BY score DESC) FROM score;
```

```
id | classid | score | rank
```

```
-----+-----+-----+-----
```

1	1	95	1
2	2	95	1
6	1	70	5
4	1	70	5
5	2	88	3
3	2	85	4

(6 rows)

ROW_NUMBER()

Description: The **ROW_NUMBER** function is used for generating consecutive sequence numbers for the values in each group. The same values have different sequence numbers.

Return type: bigint

Example:

```
SELECT id, classid, score, ROW_NUMBER() OVER(ORDER BY score DESC) FROM score ORDER BY score DESC;
```

id	classid	score	row_number
1	1	95	1
2	2	95	2
5	2	88	3
3	2	85	4
6	1	70	5
4	1	70	6

(6 rows)

DENSE_RANK()

Description: The **DENSE_RANK** function is used for generating consecutive sequence numbers for the values in each group. The same values have the same rank value number and the same sequence number.

Return type: bigint

Example:

```
SELECT id, classid, score, DENSE_RANK() OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	dense_rank
1	1	95	1
2	2	95	1
5	2	88	2
3	2	85	3
6	1	70	4
4	1	70	4

(6 rows)

PERCENT_RANK()

Description: The **PERCENT_RANK** function is used for generating corresponding sequence numbers for the values in each group. That is, the function calculates the value according to the formula $\text{Sequence number} = (\text{Rank} - 1) / (\text{Total rows} - 1)$. **Rank** is the corresponding sequence number generated based on the **RANK** function for the value and **Total rows** is the total number of elements in a group.

Return type: double precision

Example:

```
SELECT id, classid, score, PERCENT_RANK() OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	percent_rank
1	1	95	0
2	2	95	0
3	2	85	.6
4	1	70	.8
5	2	88	.4
6	1	70	.8

(6 rows)

CUME_DIST()

Description: The **CUME_DIST** function is used for generating accumulative distribution sequence numbers for the values in each group. That is, the function

calculates the value according to the following formula: Sequence number = Number of rows preceding or peer with current row/Total rows.

Return type: double precision

Example:

```
SELECT id,classid,score,CUME_DIST() OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	cume_dist
1	1	95	.3333333333333333
2	2	95	.3333333333333333
5	2	88	.5
3	2	85	.6666666666666667
4	1	70	1
6	1	70	1

(6 rows)

NTILE(num_buckets integer)

Description: The **NTILE** function is used for equally allocating sequential data sets to the buckets whose quantity is specified by **num_buckets** according to **num_buckets integer** and allocating the bucket number to each row. Divide the partition as equally as possible.

Return type: integer

Example:

```
SELECT id,classid,score,NTILE(3) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	ntile
1	1	95	1
2	2	95	1
5	2	88	2
3	2	85	2
4	1	70	3
6	1	70	3

(6 rows)

LAG(value any [, offset integer [, default any]])

Description: The **LAG** function is used for generating lag values for the corresponding values in each group. That is, the value of the row obtained by moving forward the row corresponding to the current value by **offset** (integer) is the sequence number. If the row does not exist after the moving, the result value is the default value. If omitted, **offset** defaults to **1** and **default** to **null**.

Return type: same as the parameter type

Example:

```
SELECT id,classid,score,LAG(id,3) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	lag
1	1	95	
2	2	95	
5	2	88	
3	2	85	1
4	1	70	2
6	1	70	5

(6 rows)

LEAD(value any [, offset integer [, default any]])

Description: The **LEAD** function is used for generating leading values for the corresponding values in each group. That is, the value of the row obtained by moving backward the row corresponding to the current value by **offset** (integer) is the sequence number. If the number of rows after the moving exceeds the total number for the current group, the result value is the default value. If omitted, **offset** defaults to **1** and **default** to **null**.

Return type: same as the parameter type

Example:

```
SELECT id,classid,score,LEAD(id,3) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	lead
1	1	95	3
2	2	95	4
5	2	88	6
3	2	85	
4	1	70	
6	1	70	

(6 rows)

FIRST_VALUE(value any)

Description: The **FIRST_VALUE** function is used for returning the first value of each group.

Return type: same as the parameter type

Example:

```
SELECT id,classid,score,FIRST_VALUE(id) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	first_value
1	1	95	1
2	2	95	1
5	2	88	1
3	2	85	1
4	1	70	1
6	1	70	1

(6 rows)

LAST_VALUE(value any)

Description: Returns the last value of each group.

Return type: same as the parameter type

The **LAST_VALUE** function supports the **IGNORE NULLS** syntax. This syntax returns the last value in a non-null window. If all values are **NULL**, **NULL** is returned. The format is as follows:

```
LAST_VALUE (expression) [IGNORE NULLS] OVER (window_definition)
```

Currently, only **ROWS between CURRENT ROW and UNBOUNDED FOLLOWING** and **ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT ROW** are supported.

Example:

```
SELECT id,classid,score,LAST_VALUE(id) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	last_value

1	1	95	2
2	2	95	2
5	2	88	5
3	2	85	3
4	1	70	6
6	1	70	6

(6 rows)

NTH_VALUE(value any, nth integer)

Description: The n th row for a group is the returned value. If the row does not exist, **NULL** is returned by default.

Return type: same as the parameter type

Example:

```
SELECT id,classid,score,NTH_VALUE(id,3) OVER(ORDER BY score DESC) FROM score;
```

id	classid	score	nth_value
1	1	95	
2	2	95	
5	2	88	5
3	2	85	5
4	1	70	5
6	1	70	5

(6 rows)

7.13 Type Conversion Functions

cast(x as y)

Description: Converts x into the type specified by y .

Example:

```
SELECT cast('22-oct-1997' as timestamp);
```

timestamp
1997-10-22 00:00:00

(1 row)

cast(x, y)

Description: Converts x into the type specified by y . This function is supported by version 8.2.0 or later clusters.

Example:

```
SELECT cast('22-oct-1997', timestamp);
```

timestamp
1997-10-22 00:00:00

(1 row)

try_cast(x as type)

Description: Converts x to a value of the given type. If the conversion fails and the current type is converted to a value allowed by DWS, **NULL** is returned. Otherwise, an error is raised. This function is supported by version 8.2.0 or later clusters.

Example:

```
SELECT try_cast('a' as int4);
      int4
-----
(1 row)
SELECT try_cast('22-oct-1997' as timestamp);
      timestamp
-----
1997-10-22 00:00:00
(1 row)
```

hextoraw(string)

Description: Converts a string in hexadecimal format into binary format.

Return type: raw

Example:

```
SELECT hextoraw('7D');
      hextoraw
-----
7D
(1 row)
```

numtoday(numeric)

Description: Converts values of the number type into the timestamp of the specified type.

Return type: timestamp

Example:

```
SELECT numtoday(2);
      numtoday
-----
2 days
(1 row)
```

pg_systimestamp()

Description: Obtains the system timestamp.

Return type: timestamp with time zone

Example:

```
SELECT pg_systimestamp();
      pg_systimestamp
-----
2015-10-14 11:21:28.317367+08
(1 row)
```

rawtohex(string)

Description: Converts a string in binary format into hexadecimal format.

The result is the ACSII code of the input characters in hexadecimal format.

Return type: varchar

Example:

```
SELECT rawtohex('1234567');
rawtohex
-----
31323334353637
(1 row)
```

to_char (datetime/interval [, fmt])

Description: Converts a DATETIME or INTERVAL value of the DATE/TIMESTAMP/TIMESTAMP WITH TIME ZONE/TIMESTAMP WITH LOCAL TIME ZONE type into the VARCHAR type according to the format specified by **fmt**.

- The optional parameter **fmt** includes the following types: date, time, week, quarter, and century. Each type has a unique template. The templates can be combined together. Common templates include: HH, MI, SS, YYYY, MM, and DD. For details, see [Table 7-11](#).
- A template may have a modification word. FM is a common modification word and is used to suppress the preceding zero or the following blank spaces.

Return type: varchar

Example:

```
SELECT to_char(current_timestamp,'HH12:MI:SS');
to_char
-----
10:19:26
(1 row)
SELECT to_char(current_timestamp,'FMHH12:FM MI:FMSS');
to_char
-----
10:19:46
(1 row)
```

to_char(double precision, text)

Description: Converts the values of the double-precision type into the strings in the specified format.

Return type: text

Example:

```
SELECT to_char(125.8::real, '999D99');
to_char
-----
125.80
(1 row)
```

to_char (integer/number[, fmt])

Descriptions: Converts an integer or a value in floating point format into a string in specified format.

- The optional parameter **fmt** can be the following types: decimal characters, grouping characters, positive/negative sign and currency sign. Each type has a unique template. The templates can be combined together. Common

templates include: 9, 0, millesimal sign (.), and decimal point (.). For details, see [Table 7-10](#).

- A template can have a modification word, similar to FM. However, FM does not suppress 0 which is output according to the template.
- Use the template X or x to convert an integer value into a string in hexadecimal format.

Return type: varchar

Example:

```
SELECT to_char(1485,'9,999');
to_char
-----
1,485
(1 row)
SELECT to_char( 1148.5,'9,999.999');
to_char
-----
1,148.500
(1 row)
SELECT to_char(148.5,'990999.909');
to_char
-----
0148.500
(1 row)
SELECT to_char(123,'XXX');
to_char
-----
7B
(1 row)
```

to_char(interval, text)

Description: Converts the values of the time interval type into the strings in the specified format.

Return type: text

Example:

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
to_char
-----
15:02:12
(1 row)
```

to_char(int, text)

Description: Converts the values of the integer type into the strings in the specified format.

Return type: text

Example:

```
SELECT to_char(125, '999');
to_char
-----
125
(1 row)
```

to_char(numeric, text)

Description: Converts the values of the numeric type into the strings in the specified format.

Return type: text

Example:

```
SELECT to_char(-125.8, '999D99S');
to_char
-----
125.80-
(1 row)
```

to_char(string)

Description: Converts the CHAR/VARCHAR/VARCHAR2/CLOB type into the VARCHAR type.

If this function is used to convert data of the CLOB type, and the value to be converted exceeds the value range of the target type, an error is returned.

Return type: varchar

Example:

```
SELECT to_char('01110');
to_char
-----
01110
(1 row)
```

to_char(timestamp, text)

Description: Converts the values of the timestamp type into the strings in the specified format.

Return type: text

Example:

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');
to_char
-----
10:55:59
(1 row)
```

to_clob(char/nchar/varchar/nvarchar/varchar2/nvarchar2/text/raw)

Description: Convert the RAW type or text character set type CHAR/NCHAR/VARCHAR/VARCHAR2/NVARCHAR2/TEXT into the CLOB type.

Return type: clob

Example:

```
SELECT to_clob('ABCDEF'::RAW(10));
to_clob
-----
ABCDEF
(1 row)
SELECT to_clob('hello111'::CHAR(15));
to_clob
-----
```

```
-----  
hello111  
(1 row)  
SELECT to_clob('gauss123'::NCHAR(10));  
to_clob  
-----  
gauss123  
(1 row)  
SELECT to_clob('gauss234'::VARCHAR(10));  
to_clob  
-----  
gauss234  
(1 row)  
SELECT to_clob('gauss345'::VARCHAR2(10));  
to_clob  
-----  
gauss345  
(1 row)  
SELECT to_clob('gauss456'::NVARCHAR2(10));  
to_clob  
-----  
gauss456  
(1 row)  
SELECT to_clob('World222!'::TEXT);  
to_clob  
-----  
World222!  
(1 row)
```

to_date(text)

Description: Converts values of the text type into the timestamp in the specified format.

Return type: timestamp

Example:

```
SELECT to_date('2015-08-14');  
to_date  
-----  
2015-08-14 00:00:00  
(1 row)
```

to_date(text, text)

Description: Converts the values of the string type into the dates in the specified format.

Return type: timestamp

Example:

```
SELECT to_date('05 Dec 2000', 'DD Mon YYYY');  
to_date  
-----  
2000-12-05 00:00:00  
(1 row)
```

to_date(string, fmt)

Description: Converts a string into a value of the DATE type according to the format specified by **fmt**. For details about the fmt format, see [Table 7-11](#).

This function cannot support the CLOB type directly. However, a parameter of the CLOB type can be converted using implicit conversion.

Return type: date

Example:

```
SELECT TO_DATE('05 Dec 2010','DD Mon YYYY');
       to_date
-----
2010-12-05 00:00:00
(1 row)
```

to_number (expr [, fmt])

Description: Converts **expr** into a value of the NUMBER type according to the specified format.

For details about the type conversion formats, see [Table 7-10](#).

If a hexadecimal string is converted into a decimal number, the hexadecimal string can include a maximum of 16 bytes if it is to be converted into a sign-free number.

During the conversion from a hexadecimal string to a decimal digit, the format string cannot have a character other than x or X. Otherwise, an error is reported.

Return type: number

Example:

```
SELECT to_number('12,454.8-', '99G999D9S');
       to_number
-----
-12454.8
(1 row)
```

to_number(text, text)

Description: Converts the values of the string type into the numbers in the specified format.

Return type: numeric

Example:

```
SELECT to_number('12,454.8-', '99G999D9S');
       to_number
-----
-12454.8
(1 row)
```

to_timestamp(double precision)

Description: Converts a UNIX century into a timestamp.

Return type: timestamp with time zone

Example:

```
SELECT to_timestamp(1284352323);
       to_timestamp
-----
```

```
2010-09-13 12:32:03+08
(1 row)
```

to_timestamp(string [,fmt])

Description: Converts a string into a value of the timestamp type according to the format specified by **fmt**. When **fmt** is not specified, perform the conversion according to the format specified by **nls_timestamp_format**. For details about the **fmt** format, see [Table 7-11](#).

In **to_timestamp** in DWS:

- If the input year *YYYY* is 0, an error will be reported.
- If the input year *YYYY*<0 to specify *SYYYY* in **fmt**, the year with the value of *n* (an absolute value) BC is output correctly.

Characters in the **fmt** must match the schema for formatting the data and time. Otherwise, an error is reported.

Return type: timestamp without time zone

Example:

```
SHOW nls_timestamp_format;
      nls_timestamp_format
-----
DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

SELECT to_timestamp('12-sep-2014');
      to_timestamp
-----
2014-09-12 00:00:00
(1 row)
SELECT to_timestamp('12-Sep-10 14:10:10.123000','DD-Mon-YY HH24:MI:SS.FF');
      to_timestamp
-----
2010-09-12 14:10:10.123
(1 row)
SELECT to_timestamp('-1','SYYYY');
      to_timestamp
-----
0001-01-01 00:00:00 BC
(1 row)
SELECT to_timestamp('98','RR');
      to_timestamp
-----
1998-01-01 00:00:00
(1 row)
SELECT to_timestamp('01','RR');
      to_timestamp
-----
2001-01-01 00:00:00
(1 row)
```

to_timestamp(text, text)

Description: Converts values of the string type into the timestamp of the specified type.

Return type: timestamp

Example:

```
SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
       to_timestamp
-----
2000-12-05 00:00:00
(1 row)
```

The following table describes the value formats of the **to_number** function.

Table 7-10 Template patterns for numeric formatting

Schema	Description
9	Value with specified digits
0	Values with leading zeros
Period (.)	Decimal point
Comma (,)	Group (thousand) separator
PR	Negative values in angle brackets
S	Sign anchored to number (uses locale)
L	Currency symbol (uses locale)
D	Decimal point (uses locale)
G	Group separator (uses locale)
MI	Minus sign in the specified position (if the number is less than 0)
PL	Plus sign in the specified position (if the number is greater than 0)
SG	Plus or minus sign in the specified position
RN	Roman numerals (the input values are between 1 and 3999)
TH or th	Ordinal number suffix
V	Shifts specified number of digits (decimal)

Table 7-11 describes the patterns of date and time values. They can be used for functions **to_date**, **to_timestamp**, and **to_char**, and the **nls_timestamp_format** parameter.

Table 7-11 Schemas for formatting date and time

Type	Schema	Description
Hour	HH	Number of hours in one day (01-12)
	HH12	Number of hours in one day (01-12)

Type	Schema	Description
	HH24	Number of hours in one day (00-23)
Minute	MI	Minute (00-59)
Second	SS	Second (00-59)
	FF	Microsecond (000000-999999)
	SSSSS	Second after midnight (0-86399)
Morning and afternoon	AM or A.M.	Morning identifier
	PM or P.M.	Afternoon identifier
Year	Y,YYY	Year with comma (with four digits or more)
	SYYYY	Year with four digits BC
	YYYY	Year (with four digits or more)
	YYY	Last three digits of a year
	YY	Last two digits of a year
	Y	Last one digit of a year
	IYYY	ISO year (with four digits or more)
	IYY	Last three digits of an ISO year
	IY	Last two digits of an ISO year
	I	Last one digit of an ISO year

Type	Schema	Description
	RR	Last two digits of a year (A year of the 20th century can be stored in the 21st century.) The password must comply with the following rules: • If the range of the input two-digit year is between 00 and 49: If the last two digits of the current year are between 00 and 49, the first two digits of the returned year are the same as the first two digits of the current year. If the last two digits of the current year are between 50 and 99, the first two digits of the returned year equal to the first two digits of the current year plus 1. • If the range of the input two-digit year is between 50 and 99: If the last two digits of the current year are between 00 and 49, the first two digits of the returned year equal to the first two digits of the current year minus 1. If the last two digits of the current year are between 50 and 99, the first two digits of the returned year are the same as the first two digits of the current year.
	RRRR	Capable of receiving a year with four digits or two digits. If there are 2 digits, the value is the same as the returned value of RR. If there are 4 digits, the value is the same as YYYY.
	• BC or B.C. • AD or A.D.	Era indicator Before Christ (BC) and After Christ (AD)
Month	MONTH	Full spelling of a month in uppercase (9 characters are filled in if the value is empty.)
	MON	Month in abbreviated format in uppercase (with three characters)
	MM	Month (01-12)
	RM	Month in Roman numerals (I-XII; I=JAN) and uppercase
Day	DAY	Full spelling of a date in uppercase (9 characters are filled in if the value is empty.)
	DY	Day in abbreviated format in uppercase (with three characters)
	DDD	Day in a year (001-366)

Type	Schema	Description
	DD	Day in a month (01-31)
	D	Day in a week (1-7. Sunday is 1.)
Week	W	Week in a month (1-5) (The first week starts from the first day of the month.)
	WW	Week in a year (1-53) (The first week starts from the first day of the year.)
	IW	Week in an ISO year (The first Thursday is in the first week.)
Century	CC	Century (with two digits) (The 21st century starts from 2001-01-01.)
Julian date	J	Julian date (starting from January 1 of 4712 BC)
Quarter	Q	Quarter

7.14 JSON/JSONB Functions and Operators

7.14.1 JSON/JSONB Operators

Table 7-12 Common JSON and JSONB Operators

Operator	Left Operand Type	Right Operand Type	Return Type	Description	Example
->	Array-json(b)	int	json(b)	Obtains the array-json element. If the subscript does not exist, NULL is returned.	<pre>SELECT ' [{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]::json->2; ?column? ----- {"c":"baz"} (1 row)</pre>
->	object-json(b)	text	json(b)	Obtains the value by a key. If no record exists, NULL is returned.	<pre>SELECT ' {"a":{"b":"foo"}}::json->'a'; ?column? ----- {"b":"foo"} (1 row)</pre>

Operator	Left Operand Type	Right Operand Type	Return Type	Description	Example
->>	Array-json(b)	int	text	Obtains the array-json element. If the subscript does not exist, NULL is returned.	<pre>SELECT '["a":"foo", {"b":"bar"}, {"c":"baz"}]::json->>2; ?column? ----- {"c":"baz"} (1 row)</pre>
->>	object-json(b)	text	text	Obtains the value by a key. If no record exists, NULL is returned.	<pre>SELECT '{"a":{"b":"foo"}}::json->>'a'; ?column? ----- {"b":"foo"} (1 row)</pre>
#>	container-json(b)	text[]	json	Obtains the JSON object in the specified path. If the path does not exist, NULL is returned. NOTE A DWS object identifier can end with a number sign (#). To avoid ambiguity during the parsing of a#>b, you need to add spaces in front of and behind the operator #>. Otherwise, a parsing error is reported.	<pre>SELECT '{"a":{"b":{"c":1}}}'::json #> '{a,b}'; ?column? ----- {"c":1} (1 row)</pre>
#>>	container-json(b)	text[]	text	Obtains the JSON object in the specified path. If the path does not exist, NULL is returned.	<pre>SELECT '{"a":{"b":{"c":1}}}'::json #>> '{a,b}'; ?column? ----- {"c":1} (1 row)</pre>

Table 7-13 Operators supported by jsonb

Operator	Right Oper and Type	Return Type	Description	Example
=	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":1}}}'::jsonb = '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- t (1 row)</pre>
<>	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":1}}}'::jsonb <> '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- f (1 row)</pre>
<	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":2}}}'::jsonb < '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- f (1 row)</pre>
>	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":2}}}'::jsonb > '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- t (1 row)</pre>
<=	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":2}}}'::jsonb <= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- f (1 row)</pre>
>=	jsonb	bool	Same as the jsonb_eq function, which compares the size of two jsonb files.	<pre>SELECT '{"a":{"b":{"c":2}}}'::jsonb >= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ----- t (1 row)</pre>
?	text	bool	Whether the string of the key or element exists at the top layer of the JSON value.	<pre>SELECT '{"a":1, "b":2}'::jsonb ? 'b'; ?column? ----- t (1 row)</pre>

Operator	Right Operator and Type	Return Type	Description	Example
?	text[]	bool	Whether any of these array strings exists as a top-layer key.	<pre>SELECT '{"a":1, "b":2, "c":3, "d":4}':jsonb ? '{a, b, e}':text[]; ?column? ----- t (1 row)</pre>
?&	text[]	bool	Whether all these array strings exist as top-layer keys.	<pre>SELECT '{"a":1, "b":2, "c":3, "d":4}':jsonb ?& '{a, b, c}':text[]; ?column? ----- t (1 row)</pre>
<@	jsonb	bool	Whether all items in the JSON file on the left exist at the top layer of the JSON file on the right.	<pre>SELECT '{"b":3}':jsonb <@ '{"a":{"b":{"c":2}}, "b":3}':jsonb; ?column? ----- t (1 row)</pre>
@>	jsonb	bool	Whether all items in the JSON file on the right exist at the top layer of the JSON file on the left.	<pre>SELECT '{"a":{"b":{"c":2}}, "b":3}':jsonb @> '{ "b":3}':jsonb; ?column? ----- t (1 row)</pre>
	jsonb	jsonb	Combines two JSONB objects into one.	<pre>SELECT '{"a":1, "b":2}':jsonb '{"c":3, "d":4}':jsonb; ?column? ----- {"a": 1, "b": 2, "c": 3, "d": 4} (1 row)</pre>
-	text	jsonb	Deletes a jsonb object and the specified key-value pair.	<pre>SELECT '{"a":1, "b":2}':jsonb - 'a'; ?column? ----- {"b": 2} (1 row)</pre>
-	text	jsonb	Deletes a jsonb object and the specified key-value pair.	<pre>SELECT '{"a":1, "b":2, "c":3, "d":4}':jsonb - '{a, b}':text[]; ?column? ----- {"c": 3, "d": 4} (1 row)</pre>
-	int	jsonb	Deletes the element corresponding to the subscript in the JSONB array.	<pre>SELECT '["a", "b", "c"]':jsonb - 2; ?column? ----- ["a", "b"] (1 row)</pre>

Operator	Right Operator and Type	Return Type	Description	Example
#-	text[]	jsonb	Deletes the key-value pair corresponding to the path in the JSONB object.	<pre>SELECT '{"a":{"b":{"c":{"d":1}}}, "e":2, "f":3}::jsonb #- "{a, b}::text[]; ?column? ----- {"a": {}, "e": 2, "f": 3} (1 row)</pre>

7.14.2 JSON/JSONB Functions

JSON/JSONB functions are used to generate JSON data (see [JSON Types](#)).

NOTE

Except the **array_to_json** and **row_to_json** functions, other JSON/JSONB functions and operators are supported only in 8.1.2 or later.

array_to_json(anyarray [, pretty_bool])

Description: Returns the array as JSON. A multi-dimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if **pretty_bool** is **true**.

Return type: json

Example:

```
postgres=#SELECT array_to_json('{{1,5},{99,100}}::int[]);
array_to_json
-----
[[1,5],[99,100]]
(1 row)
```

row_to_json(record [, pretty_bool])

Description: Returns the row as JSON. Line feeds will be added between level-1 elements if **pretty_bool** is **true**.

Return type: json

Example:

```
postgres=#SELECT row_to_json(row(1,'foo'));
row_to_json
-----
{"f1":1,"f2":"foo"}
(1 row)
```

json_agg(any)

Description: Aggregates values into a JSON array.

Return type: array-json

Example:

```
postgres=#SELECT * FROM classes;
name | score
-----+-----
A    |    2
A    |    3
D    |    5
D    |
(4 rows)

postgres=#SELECT name, json_agg(score) score FROM classes group by name order by name;
name |      score
-----+-----
A    | [2, 3]
D    | [5, null]
      | [null]
(3 rows)
```

json_object_agg(any, any)

Description: Aggregates values into a JSON object.

Return type: json

Example:

```
postgres=#SELECT * FROM classes;
name | score
-----+-----
A    |    2
A    |    3
D    |    5
D    |
(4 rows)

postgres=#SELECT json_object_agg(name, score) FROM classes group by name order by name;
      json_object_agg
-----+-----
{ "A" : 2, "A" : 3 }
{ "D" : 5, "D" : null }
(2 rows)
```

json_build_array(VARIADIC "any")

Description: Constructs a JSON array that may be of heterogeneous type from a variable parameter list.

Return type: json

Example:

```
postgres=#SELECT json_build_array(1,2,'3',4,5);
      json_build_array
-----+-----
[1, 2, "3", 4, 5]
(1 row)
```

json_build_object(VARIADIC "any")

Description: Constructs a JSON object from a variable parameter list. The parameter list consists of alternate keys and values. The number of input parameters must be an even number. Every two input parameters form a key-value pair. Note that the value of a key cannot be null.

Return type: json

Example:

```
postgres=#SELECT json_build_object('foo',1,'bar',2);
 json_build_object
-----
{"foo" : 1, "bar" : 2}
(1 row)
```

json_object(text[]), json_object(text[], text[])

Description: Constructs a JSON object from a text array.

This is an overloaded function. When the input parameter is a text array, the array length must be an even number, and members are considered alternate key-value pairs. When two text arrays are used, the first array is regarded as a key, and the second array is regarded as a value. The lengths of the two arrays must be the same. Note that the value of a key cannot be null.

Return type: json

Example:

```
postgres=#SELECT json_object('{a, 1, b, "def", c, 3.5}');
 json_object
-----
{"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

postgres=#SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');
 json_object
-----
{"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

postgres=#SELECT json_object('{a,b,"a b c"}', '{a,1,1}');
 json_object
-----
{"a" : "a", "b" : "1", "a b c" : "1"}
(1 row)
```

to_json(anyelement)

Description: Converts parameters to json.

Return type: json

Example:

```
postgres=#SELECT to_json('Fred said "Hi."::text);
 to_json
-----
"Fred said \"Hi.\""
(1 row)

-- Convert the column-store table json_tbl_2 to JSON:
postgres=# SELECT * FROM json_tbl_2;
 a | b
---+---
 1 | aaa
 1 | bbb
 2 | ccc
 2 | ddd
(4 rows)

postgres=# SELECT to_json(t.*) FROM json_tbl_2 t;
 to_json
-----
{"a":1,"b":"bbb"}
```

```
{ "a":2,"b":"ddd"}  
{ "a":1,"b":"aaa"}  
{ "a":2,"b":"ccc"}  
(4 rows)
```

json_strip_nulls(json)

Description: All object fields with null values are ignored, and other values remain unchanged.

Return type: json

Example:

```
postgres=#SELECT json_strip_nulls('{"f1":1,"f2":null},2,null,3');  
json_strip_nulls  
-----  
{"f1":1},2,null,3  
(1 row)
```

json_object_field(json, text)

Description: Same as the operator ->, which returns the value of a specified key in an object.

Return type: json

Example:

```
postgres=#SELECT json_object_field('{"a": {"b":"foo"}}','a');  
json_object_field  
-----  
{"b":"foo"}  
(1 row)
```

json_object_field_text(object-json, text)

Description: Same as the operator ->>, which returns the value of a specified key in an object.

Return type: text

Example:

```
postgres=#SELECT json_object_field_text('{"a": {"b":"foo"}}','a');  
json_object_field_text  
-----  
{"b":"foo"}  
(1 row)
```

json_array_element(array-json, integer)

Description: Same as the operator ->, which returns the element with the specified subscript in the array.

Return type: json

Example:

```
postgres=#SELECT json_array_element('[1,true,[1,[2,3]],null]',2);  
json_array_element  
-----
```

```
[1,[2,3]]  
(1 row)
```

json_array_element_text(array-json, integer)

Description: Same as the operator `->>`, which returns the element with the specified subscript in the array.

Return type: text

Example:

```
postgres=#SELECT json_array_element_text('[1,true,[1,[2,3]],null]',2);  
json_array_element_text  
-----  
[1,[2,3]]  
(1 row)
```

json_extract_path(json, VARIADIC text[])

Description: Same as the operator `#>`, which returns the JSON value of the path specified by `$2`.

Return type: json

Example:

```
postgres=#SELECT json_extract_path('{\"f2\":{\"f3\":1},\"f4\":{\"f5\":99,\"f6\":\"stringy\"}}', 'f4','f6');  
json_extract_path  
-----  
\"stringy\"  
(1 row)
```

json_extract_path_text(json, VARIADIC text[])

Description: Same as the operator `#>>`, which returns the text value of the path specified by `$2`.

Return type: text

Example:

```
postgres=#SELECT json_extract_path_text('{\"f2\":{\"f3\":1},\"f4\":{\"f5\":99,\"f6\":\"stringy\"}}', 'f4','f6');  
json_extract_path_text  
-----  
stringy  
(1 row)
```

json_array_elements(array-json)

Description: Splits an array. Each element returns a row.

Return type: json

Example:

```
postgres=#SELECT json_array_elements('[1,true,[1,[2,3]],null]');  
json_array_elements  
-----  
1  
true  
[1,[2,3]]  
null  
(4 rows)
```

json_array_elements_text(array-json)

Description: Splits an array. Each element returns a row.

Return type: text

Example:

```
postgres=#SELECT * FROM json_array_elements_text('1,true,[1,[2,3]],null');
 value
-----
 1
 true
[1,[2,3]]
(4 rows)
```

json_array_length(array-json)

Description: Returns the array length.

Return type: integer

Example:

```
postgres=#SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4,null]');
 json_array_length
-----
                6
(1 row)
```

json_object_keys(object-json)

Description: Returns all keys at the top layer of the object.

Return type: text

Example:

```
postgres=#SELECT json_object_keys('{ "f1": "abc", "f2": {"f3": "a", "f4": "b"}, "f1": "abcd"}');
 json_object_keys
-----
 f1
 f2
 f1
(3 rows)
```

json_each(object-json)

Description: Splits each key-value pair of an object into one row and two columns.

Return type: setof(key text, value json)

Example:

```
postgres=#SELECT * FROM json_each('{ "f1": [1,2,3], "f2": {"f3": 1}, "f4": null}');
 key | value
-----+-----
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  | null
(3 rows)
```

json_each_text(object-json)

Description: Splits each key-value pair of an object into one row and two columns.

Return type: setof(key text, value text)

Example:

```
postgres=#SELECT * FROM json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
key | value
-----+-----
f1  | [1,2,3]
f2  | {"f3":1}
f4  | 
(3 rows)
```

json_populate_record(anyelement, object-json [, bool])

Description: *\$1* must be a compound parameter. Each key-value in **object-json** is split. The key is used as the column name to match the column name in *\$1* and fill in the *\$1* format.

NOTE

To define a composite type for the input parameters of a JSON/JSONB function, you can use CREATE TYPE or CREATE TABLE. Here is an example:

```
CREATE TYPE jpop AS (a text, b INT, c timestamp);
CREATE TABLE jpop2(a text, b INT, c timestamp);
```

Return type: anyelement

Example:

```
postgres=#CREATE TYPE jpop AS (a text, b INT, c timestamp);
postgres=#SELECT * FROM json_populate_record(null::jpop, '{"a":"blurfl","x":43.2}');
a | b | c
-----+-----
blurfl |  | 
(1 row)
```

json_populate_recordset(anyelement, array-json [, bool])

Description: Performs the preceding operations on each element in the *\$2* array by referring to the **json_populate_record** and **jsonb_populate_record** functions. Therefore, each element in the *\$2* array must be of the **object-json** type.

Return type: setof anyelement

Example:

```
postgres=#CREATE TYPE jpop AS (a text, b INT, c timestamp);
postgres=#SELECT * FROM json_populate_recordset(null::jpop, '[{"a":1,"b":2},{\"a\":3,\"b\":4}]');
a | b | c
-----+-----
1 | 2 | 
3 | 4 | 
(2 rows)
```

json_to_record(object-json)

Description: Like all functions that return **record**, the caller must explicitly define the structure of the record using an **AS** clause. The key-value pair of **object-json** is

split and reassembled. The key is used as a column name to match and fill in the structure of the record specified by the **AS** clause.

Return type: record

Example:

```
postgres=#SELECT * FROM json_to_record('{\"a\":1,\"b\":\"foo\",\"c\":\"bar\"}::json) as x(a int, b text, d text);
 a | b | d
---+---+---
 1 | foo | 
(1 row)
```

json_to_recordset(array-json)

Description: Executes the preceding function on each element in the array by referring to the **json_to_record** function. Therefore, each element in the array must be **object-json**.

Return type: SETOF record

Example:

```
postgres=#SELECT * FROM json_to_recordset('[{\"a\":1,\"b\":{\"d\":\"foo\"},\"c\":true},{\"a\":2,\"c\":false,\"b\":{\"d\":\"bar\"}}]') AS x(a INT, b json, c BOOLEAN);
 a |      b      | c
---+-----+---
 1 | {\"d\":\"foo\"} | t
 2 | {\"d\":\"bar\"} | f
(2 rows)

postgres=#SELECT * FROM json_to_recordset('[{\"a\":1,\"b\":\"foo\",\"d\":false},{\"a\":2,\"b\":\"bar\",\"c\":true}]') AS x(a INT, b text, c BOOLEAN);
 a | b | c
---+---+---
 1 | foo | 
 2 | bar | t
(2 rows)
```

json_typeof(json)

Description: Checks the JSON type.

Return type: text

Example:

```
postgres=#SELECT value, json_typeof(value) from (values (json '123.4'), (json '\"foo\"'), (json 'true'), (json 'null'), (json '[1, 2, 3]'), (json '{\"x\":\"foo\", \"y\":123}'), (NULL::json)) as data(value);
 value | json_typeof
-----+-----
 123.4 | number
 \"foo\"  | string
 true  | boolean
 null  | null
 [1, 2, 3] | array
 {\"x\":\"foo\", \"y\":123} | object
(7 rows)
```

jsonb_object(text[])

Description: Constructs an **object-jsonb** from a text array. This is an overloaded function. When the input parameter is a text array, the array length must be an even number, and members are considered alternate key-value pairs.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_object('{a,1,b,2,3,NULL,"d e f","a b c"}');
          jsonb_object
-----
{"3": null, "a": "1", "b": "2", "d e f": "a b c"}
(1 row)
```

jsonb_object(text[], text[])

Description: When two text arrays are used, the first array is considered a key and the second array is considered a value. The lengths of the two arrays must be the same. Note that the value of a key cannot be null.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_object('{a,b,"a b c"}', '{a,1,1}');
          jsonb_object
-----
{"a": "a", "b": "1", "a b c": "1"}
(1 row)
```

to_jsonb(anyment)

Description: Converts other types to the corresponding jsonb type.

Return type: jsonb

Example:

```
postgres=#SELECT to_jsonb(1.1);
          to_jsonb
-----
1.1
(1 row)
```

jsonb_agg

Description: Aggregates jsonb objects into a jsonb array.

Return type: jsonb

Example:

```
postgres=#SELECT * FROM json_tbl_2;
 a | b
---+---
 1 | aaa
 1 | bbb
 2 | ccc
 2 | ddd
(4 rows)

postgres=#SELECT a, jsonb_agg(b) FROM json_tbl_2 GROUP BY a ORDER BY a;
 a | jsonb_agg
---+-----
 1 | ["aaa", "bbb"]
 2 | ["ccc", "ddd"]
(2 rows)
```

jsonb_object_agg

Description: Aggregates key-value pairs into a JSON object.

Return type: jsonb

Example:

```
postgres=#SELECT * FROM json_tbl_3;
 a | b | c
---+---+---
 1 | aaa | 10
 1 | bbb | 20
 2 | ccc | 30
 2 | ddd | 40
(4 rows)
postgres=#SELECT a, jsonb_object_agg(b, c) FROM json_tbl_3 GROUP BY a ORDER BY a;
 a | jsonb_object_agg
---+-----
 1 | {"aaa": 10, "bbb": 20}
 2 | {"ccc": 30, "ddd": 40}
(2 rows)
```

jsonb_build_array([VARIADIC "any"])

Description: Constructs a JSON array that may contain heterogeneous types from a variable parameter list.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_build_array('a',1,'b',1.2,'c',true,'d',null,'e',json '{"x": 3, "y": [1,2,3]}',"");
 jsonb_build_array
-----
["a", 1, "b", 1.2, "c", true, "d", null, "e", {"x": 3, "y": [1, 2, 3]}, null]
(1 row)
```

jsonb_build_object([VARIADIC "any"])

Description: Constructs a JSON object from a variable parameter list. The number of input parameters must be an even number. Every two input parameters form a key-value pair. Note that the value of a key cannot be null.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_build_object(1,2);
 jsonb_build_object
-----
{"1": 2}
(1 row)
```

jsonb_strip_nulls(jsonb)

Description: All object fields with null values are omitted. Other null values remain unchanged.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_strip_nulls('{"f1":1,"f2":null},2,null,3');
jsonb_strip_nulls
-----
[{"f1": 1}, 2, null, 3]
(1 row)
```

jsonb_object_field(jsonb, text)

Description: Same as the operator ->, which returns the value of a specified key in an object.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_object_field('{"a": {"b":"foo"}}', 'a');
jsonb_object_field
-----
{"b": "foo"}
(1 row)
```

jsonb_object_field_text(jsonb, text)

Description: Same as the operator ->>, which returns the value of a specified key in an object.

Return type: text

Example:

```
postgres=#SELECT jsonb_object_field_text('{"a": {"b":"foo"}}', 'a');
jsonb_object_field_text
-----
{"b": "foo"}
(1 row)
```

jsonb_array_element(array-jsonb, integer)

Description: Same as the operator ->, which returns the element with the specified subscript in the array.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_array_element('[1,true,[1,[2,3]],null]',2);
jsonb_array_element
-----
[1, [2, 3]]
(1 row)
```

jsonb_array_element_text(array-jsonb, integer)

Description: Same as the operator ->>, which returns the element with the specified subscript in the array.

Return type: text

Example:

```
postgres=#SELECT jsonb_array_element_text('[1,true,[1,[2,3]],null]',2);
jsonb_array_element_text
-----
```

```
[1, [2, 3]]  
(1 row)
```

jsonb_extract_path((jsonb, VARIADIC text[]))

Description: Same as the operator `#>`, which returns the value of the path specified by `$2`.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_extract_path('{ "f2":{"f3":1}, "f4":{"f5":99, "f6":"stringy"}}', 'f4','f6');  
jsonb_extract_path  
-----  
"stringy"  
(1 row)
```

jsonb_extract_path_text((jsonb, VARIADIC text[]))

Description: Same as the operator `#>>`, which returns the value of the path specified by `$2`.

Return type: text

Example:

```
postgres=#SELECT jsonb_extract_path_text('{ "f2":{"f3":1}, "f4":{"f5":99, "f6":"stringy"}}', 'f4','f6');  
jsonb_extract_path_text  
-----  
stringy  
(1 row)
```

jsonb_extract((jsonb, VARIADIC text[]))

Description: If you enter any object-jsonb or array-jsonb type, the value of the path specified by `$2` is returned. This function is supported only by clusters of version 9.1.0 or later.

Return type: SETOF jsonb

Example:

```
SELECT jsonb_extract('{ "f2":{"f3":1}, "f4":[{"f5":99}, {"f6":"stringy"}]}', 'f2','f3');  
jsonb_extract  
-----  
1  
(1 row)  
  
SELECT jsonb_extract('{ "f2":{"f3":1}, "f4":[{"f5":99}, {"f5":"stringy"}]}', 'f4','f5');  
jsonb_extract  
-----  
99  
"stringy"  
(2 rows)
```

jsonb_extract_text((jsonb, VARIADIC text[]))

Description: If you enter any object-jsonb or array-jsonb type, the value of the path specified by `$2` is returned. This function is supported only by clusters of version 9.1.0 or later.

Return type: SETOF text

Example:

```
SELECT jsonb_extract_text('{"f2":{"f3":1},"f4":[{"f5":99},{"f6":"stringy"}]}', 'f2', 'f3');
jsonb_extract_text
-----
1
(1 row)

SELECT jsonb_extract_text('{"f2":{"f3":1},"f4":[{"f5":99},{"f5":"stringy"}]}', 'f4', 'f5');
jsonb_extract_text
-----
99
stringy
(2 rows)
```

jsonb_array_elements(array-jsonb)

Description: Splits an array. Each element returns a row.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_array_elements('[1,true,[1,[2,3]],null]');
jsonb_array_elements
-----
1
true
[1, [2, 3]]
null
(4 rows)
```

jsonb_array_elements_text(array-jsonb)

Description: Splits an array. Each element returns a row.

Return type: text

Example:

```
postgres=#SELECT * FROM jsonb_array_elements_text('[1,true,[1,[2,3]],null]');
value
-----
1
true
[1, [2, 3]]
(4 rows)
```

jsonb_array_length(array-jsonb)

Description: Returns the array length.

Return type: integer

Example:

```
postgres=#SELECT jsonb_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4,null]');
jsonb_array_length
-----
6
(1 row)
```

jsonb_object_keys(object-jsonb)

Description: Returns all keys at the top layer of the object.

Return type: SETOF text

Example:

```
postgres=#SELECT jsonb_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}, "f1":"abcd"}');
jsonb_object_keys
-----
f1
f2
(2 rows)
```

jsonb_each(object-jsonb)

Description: Splits each key-value pair of an object into one row and two columns.

Return type: setof(key text, value jsonb)

Example:

```
postgres=#SELECT * FROM jsonb_each('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
key | value
-----+-----
f1  | [1, 2, 3]
f2  | {"f3": 1}
f4  | null
(3 rows)
```

jsonb_each_text(object-jsonb)

Description: Splits each key-value pair of an object into one row and two columns.

Return type: setof(key text, value text)

Example:

```
postgres=#SELECT * FROM jsonb_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
key | value
-----+-----
f1  | [1, 2, 3]
f2  | {"f3": 1}
f4  | 
(3 rows)
```

jsonb_populate_record(anyelement, object-jsonb [, bool])

Description: *\$1* must be a compound parameter. Each key-value in **object-json** is split. The key is used as the column name to match the column name in *\$1* and fill in the *\$1* format.

Return type: anyelement

Example:

```
postgres=#SELECT * FROM jsonb_populate_record(null::jpop,{'a':"blurfl","x":43.2});
a | b | c
-----+-----
blurfl | | 
(1 row)
```

jsonb_populate_recordset(anyelement, array-jsonb [, bool])

Description: Performs the preceding operations on each element in the $\$2$ array by referring to the **json_populate_record** and **jsonb_populate_record** functions. Therefore, each element in the $\$2$ array must be of the **object-json** type.

Return type: setof anyelement

Example:

```
postgres=#SELECT * FROM jsonb_populate_recordset(null::jpop, '["a":1,"b":2],{"a":3,"b":4}');
 a | b | c
---+---+---
 1 | 2 | 
 3 | 4 | 
(2 rows)
```

jsonb_to_record(object-json)

Description: Like all functions that return **record**, the caller must explicitly define the structure of the record using an **AS** clause. The key-value pair of **object-json** is split and reassembled. The key is used as a column name to match and fill in the structure of the record specified by the **AS** clause.

Return type: record

Example:

```
postgres=#SELECT * FROM jsonb_to_record('{"a":1,"b":"foo","c":"bar"}'::jsonb) as x(a int, b text, d text);
 a | b | d
---+---+---
 1 | foo | 
(1 row)
```

jsonb_to_recordset(array-json)

Description: Executes the preceding function on each element in the array by referring to the **jsonb_to_record** function. Therefore, each element in the array must be **object-jsonb**.

Return type: SETOF record

Example:

```
postgres=#SELECT * FROM jsonb_to_recordset('["a":1,"b":"foo","d":false],{"a":2,"b":"bar","c":true}')] AS x(a
INT, b text, c boolean);
 a | b | c
---+---+---
 1 | foo | 
 2 | bar | t
(2 rows)
```

jsonb_typeof(jsonb)

Description: Checks the JSONB type.

Return type: text

Example:

```
postgres=#SELECT jsonb_typeof(to_jsonb(1.1));
 jsonb_typeof
-----
```



```
number  
(1 row)
```

jsonb_ne(jsonb, jsonb)

Description: Same as the operator <>, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_ne('{"a":1, "b":2}::jsonb, '{"a":1, "b":3}::jsonb);  
jsonb_ne  
-----  
t  
(1 row)
```

jsonb_lt(jsonb, jsonb)

Description: Same as the operator <, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_lt('{"a":1, "b":2}::jsonb, '{"a":1, "b":3}::jsonb);  
jsonb_lt  
-----  
t  
(1 row)
```

jsonb_gt(jsonb, jsonb)

Description: Same as the operator >, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_gt('{"a":1, "b":2}::jsonb, '{"a":1, "b":3}::jsonb);  
jsonb_gt  
-----  
f  
(1 row)
```

jsonb_le(jsonb, jsonb)

Description: Same as the operator <=, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_le('["a", "b"]', '{"a":1, "b":2}');  
jsonb_le  
-----  
t  
(1 row)
```

jsonb_ge(jsonb, jsonb)

Description: Same as the operator >=, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_ge('["a", "b"]', '{"a":1, "b":2}');
jsonb_ge
-----
f
(1 row)
```

jsonb_eq(jsonb, jsonb)

Description: Same as the operator =, which compares two values.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_eq('["a", "b"]', '{"a":1, "b":2}');
jsonb_eq
-----
f
(1 row)
```

jsonb_cmp(jsonb, jsonb)

Description: Compares values. A positive value indicates greater than, a negative value indicates less than, and 0 indicates equal.

Return type: integer

Example:

```
postgres=#SELECT jsonb_cmp('["a", "b"]', '{"a":1, "b":2}');
jsonb_cmp
-----
-1
(1 row)
```

jsonb_exists(jsonb, text)

Description: Same as the operator ?, which determines whether all elements in the string array \$2 exist at the top layer of \$1 in the form of **key\elem\scalar**.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_exists('["1",2,3]', '1');
jsonb_exists
-----
t
(1 row)
```

jsonb_exists_any(jsonb, text[])

Description: Same as the operator ?|, which determines whether all elements in the string array \$2 exist at the top layer of \$1 in the form of **key\elem\scalar**.

Return type:

Example:

```
postgres=#SELECT jsonb_exists_any('["1","2",3]', '{1, 2, 4}');
jsonb_exists_any
-----
t
(1 row)
```

jsonb_exists_all(jsonb, text[])

Description: Same as the operator **?&**, which determines whether all elements in the string array *\$2* exist at the top layer of *\$1* in the form of **key\elem\scalar**.

Return type:

bool

Example:

```
postgres=#SELECT jsonb_exists_all('["1","2",3]', '{1, 2}');
jsonb_exists_all
-----
t
(1 row)
```

jsonb_contained(jsonb, jsonb)

Description: Same as the operator **<@**, which determines whether all elements in the string array *\$1* exist at the top layer of *\$2*.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_contained('[1,2,3]', '[1,2,3,4]');
jsonb_contained
-----
t
(1 row)
```

jsonb_contains(jsonb, jsonb)

Description: Same as the operator **@>**, which determines whether all elements in the string array *\$2* exist at the top layer of *\$1*.

Return type: Boolean

Example:

```
postgres=#SELECT jsonb_contains('{ "a":1, "b":2, "c":3 }::jsonb, '{ "a":1 }');
jsonb_contains
-----
t
(1 row)
```

jsonb_concat(jsonb, jsonb)

Description: Combines two JSONB objects into one.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_concat('{ "a":1, "b":2 }::jsonb, '{ "c":3, "d":4 }::jsonb');
jsonb_concat
```

```
-----  
{ "a": 1, "b": 2, "c": 3, "d": 4 }  
(1 row)
```

jsonb_delete(jsonb, text)

Description: Deletes the key-value pair corresponding to the key value in jsonb.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_delete('{ "a":1, "b":2 }'::jsonb, 'a');  
jsonb_delete  
-----  
{ "b": 2 }  
(1 row)
```

jsonb_delete_idx(jsonb, text)

Description: Deletes the element corresponding to an array subscript.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_delete_idx('[0,1,2,3,4]'::jsonb, 2);  
jsonb_delete_idx  
-----  
[0, 1, 3, 4]  
(1 row)
```

jsonb_delete_array(jsonb, VARIADIC text[])

Description: Deletes multiple elements from the jsonb array.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_delete_array(['a", "b", "c"]'::jsonb , 'a', 'b');  
jsonb_delete_array  
-----  
["c"]  
(1 row)
```

jsonb_delete_path(jsonb, text[])

Description: Deletes elements of a specified path from the jsonb array.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_delete_path('{ "a":{"b":{"c":1, "d":2}}, "e":3 }'::jsonb , array['a', 'b']);  
jsonb_delete_path  
-----  
{ "a": {}, "e": 3 }  
(1 row)
```

jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])

Description: Returns *target* with the section designated by *path* replaced by *new_value*, or with *new_value* added if **create_missing** is **true** (**true** by default) and the item designated by *path* does not exist. As with the path-oriented operators, negative integers that appear in *path* count from the end of JSON arrays.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_set('{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false);
      jsonb_set
-----
[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]
(1 row)
```

jsonb_pretty(jsonb)

Description: Returns in indented JSON text.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_pretty('{"a":{"b":{"c":1, "d":2}}, "e":3}::jsonb');
      jsonb_pretty
-----
{
  "a": {
    "b": {
      "c": 1,
      "d": 2
    }
  },
  "e": 3
}
(1 row)
```

jsonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])

Description: Returns the target where **new_value** is inserted. When the specified target is located within the **JSONB** array using the path, use the **insert_after** parameter to determine if **new_value** should be inserted after the target. The default value is **false**, meaning that the new value will be inserted after the target. If the target part specified by **path** is in a **JSONB** object, **new_value** is inserted only when the target part does not exist. Like path-oriented operators, a negative integer in **path** indicates that the count starts from the end of the **JSON** array.

Return type: jsonb

Example:

```
postgres=#SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');
      jsonb_insert
-----
{"a": [0, "new_value", 1, 2]}
(1 row)
```

ts_headline([config regconfig,] document jsonb, query tsquery [, options text])

Description: Highlights the jsonb search result.

Return type: jsonb

Example:

```
postgres=#SELECT ts_headline('english',
'[{ "id":9928,"user_id":4562,"user_name":"9LOHR4","create_time":"2021-06-22T16:28:16.504518+08:00"},
{ "id":9959,"user_id":5524,"user_name":"YID07D","create_time":"2021-06-22T16:28:16.557228+08:00"},
{ "id":9962,"user_id":7991,"user_name":"7C6QOM","create_time":"2021-06-22T16:28:16.56234+08:00"}]::jsonb,
to_tsquery('english', '9LOHR4'), 'StartSel = <, StopSel = >');

ts_headline
-----
[{"id": 9928, "user_id": 4562, "user_name": "<9LOHR4>", "create_time":
"2021-06-22T16:28:16.504518+08:00"}, {"id": 9959, "user_id": 5524, "user_name": "YID07D", "create_time":
"2021-06-22T16:28:16.557228+08:00"}, {"id": 9962, "user_id": 7991, "user_name": "7C6QOM",
"create_time": "2021-06-22T16:28:16.56234+08:00"}]
(1 row)
```

json_to_tsvector(config regconfig,] json, jsonb)

Description: Converts the json format to the tsvector file format that supports full-text search.

Return type: jsonb

Example:

```
postgres=#SELECT json_to_tsvector('{ "a":1, "b":2, "c":3 }::json, to_jsonb('key':text));

json_to_tsvector
-----
'b':2 'c':4
(1 row)
```

7.15 Security Functions

gs_password_deadline()

Description: Displays the time before the password of the current account expires. After the password expires, the system prompts the user to change the password. This parameter is related to the GUC parameter **password_effect_time**.

Return type: interval

Example:

```
SELECT gs_password_deadline();

gs_password_deadline
-----
83 days 17:44:32.196094
(1 row)
```

gs_password_expiration()

Description: Displays the time before the password of the current account expires. After the password expires, the user cannot log in to the database. This parameter is related to the DDL statement **PASSWORD EXPIRATION period** for creating a user. The return value of the function is greater than or equal to -1. If **PASSWORD EXPIRATION period** is not specified during user creation, the default value is -1, indicating that there is no expiration limit.

Return type: interval

Example:

```
SELECT gs_password_expiration();
gs_password_expiration
-----
29 days 23:59:49.731482
(1 row)
```

login_audit_messages(flag boolean)

Description: Queries login information about a login user.

Return type: tuple

Example:

- Checks the date, time, and IP address successfully authenticated during the last login.

```
SELECT * FROM login_audit_messages(true);
username | database | logintime      | type   | result | client_conninfo | session_id
-----+-----+-----+-----+-----+-----+-----
dbadmin  | gaussdb  | 2017-06-02 15:28:34+08 | login_success | ok    | gsqr@[local]    | 1663745305.140192538154752.coordinator1
(1 row)
```

- Checks the date, time, and IP address that failed to be authenticated during the last login.

```
SELECT * FROM login_audit_messages(false) ORDER BY logintime desc limit 1;
username | database | logintime      | type   | result | client_conninfo | session_id
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

- Checks the number of failed attempts, date, and time since the previous successful authentication.

```
SELECT * FROM login_audit_messages(false);
username | database | logintime      | type   | result | client_conninfo | session_id
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

login_audit_messages_pid(flag boolean)

Description: Queries login information about a login user. Different from **login_audit_messages**, this function queries login information based on **backendid**. Information about subsequent logins of the same user does not alter the query result of previous logins and cannot be found using this function.

Return type: tuple

Example:

- Checks the date, time, and IP address successfully authenticated during the last login.

```
SELECT * FROM login_audit_messages_pid(true);
username | database | logintime      | type  | result | client_conninfo | backendid |
session_id
-----+-----+-----+-----+-----+-----+-----
dbadmin  | postgres | 2017-06-02 15:28:34+08 | login_success | ok    | gsqr@[local] | 140311900702464 | 1663745305.140192538154752.coordinator1
(1 row)
```

- Checks the date, time, and IP address that failed to be authenticated during the last login.

```
SELECT * FROM login_audit_messages_pid(false) ORDER BY logintime desc limit 1;
username | database | logintime      | type  | result | client_conninfo | backendid |
session_id
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

- Checks the number of failed attempts, date, and time since the previous successful authentication.

```
SELECT * FROM login_audit_messages_pid(false);
username | database | logintime      | type  | result | client_conninfo | backendid |
session_id
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

pg_query_audit()

Description: Displays audit logs of the CN.

Return type: record

The following table describes return columns.

Table 7-14 Columns returned by the pg_query_audit() function

Column	Type	Description
begintime	timestamp with time zone	Operation start time
endtime	timestamp with time zone	Operation end time
operation_type	text	Operation type. For details, see Table 7-15 .
audit_type	text	Audit type. For details, see Table 7-16 .
result	text	Operation result
username	text	Name of the user who performs the operation
database	text	Database name
client_conninfo	text	Client connection information, that is, gsqr, JDBC, or ODBC.
object_name	text	Object name

Column	Type	Description
object_details	text	Columns involved in the statement and their types.
command_text	text	Command used to perform the operation
detail_info	text	Operation details
transaction_xid	text	Transaction ID
query_id	text	Query ID
node_name	text	Node name
session_id	text	ID of a session.
local_port	text	Local port
remote_port	text	Remote port
result_rows	text	Number of records returned after the statement is executed.
error_code	text	Error code returned when a statement execution error occurs.

Table 7-15 Operation types

Operation type	Description
audit_switch	Indicates that the operations of enabling and disabling the audit log function are audited.
login_logout	Indicates that user login and log-out operations are audited.
system	Indicates that the system startup, shutdown, and instance switchover operations are audited.
sql_parse	Indicates that SQL statement parsing operations are audited.
user_lock	Indicates that user locking and unlocking operations are audited.
grant_revoke	Indicates that user permission granting and revoking operations are audited.
violation	Indicates that user's access violation operations are audited.

Operation type	Description
ddl	Indicates that DDL operations are audited. DDL operations are controlled at a fine granularity based on operation objects. Therefore, audit_system_object is used to specify which objects' DDL operations should be audited. Note that the audit function is active whenever audit_system_object is configured, regardless of the ddl setting.
dml	Indicates that the DML operations are audited.
select	Indicates that the SELECT operations are audited.
internal_event	Indicates that internal incident operations are audited.
user_func	Indicates that operations related to user-defined functions, stored procedures, and anonymous blocks are audited. NOTE If a user-defined function or stored procedure contains a FETCH statement, the common_text field records the corresponding CURSOR content when the fetch statement is audited.
special_func	Indicates that special function invoking operations are audited. Special functions include pg_terminate_backend and pg_cancel_backend .
copy	Indicates that the COPY operations are audited.
set	Indicates that the SET operations are audited.
transaction	Indicates that transaction operations are audited.
vacuum	Indicates that the VACUUM operations are audited.
analyze	Indicates that the ANALYZE operations are audited.
cursor	Indicates that cursor operations are audited.
anonymous_block	Indicates that the anonymous block operations are audited.
explain	Indicates that the EXPLAIN operations are audited.
show	Indicates that the SHOW operations are audited.
lock_table	Indicates that table lock operations are audited.
comment	Indicates that the COMMENT operations are audited.
preparestmt	Indicates that the PREPARE , EXECUTE , and DEALLOCATE operations are audited.
cluster	Indicates that the CLUSTER operations are audited.
constraints	Indicates that the CONSTRAINTS operations are audited.

Operation type	Description
checkpoint	Indicates that the CHECKPOINT operations are audited.
barrier	Indicates that the BARRIER operations are audited.
cleanconn	Indicates that the CLEAN CONNECTION operations are audited.
seclabel	Indicates that security label operations are audited.
notify	Indicates that the notification operations are audited.
load	Indicates that the loading operations are audited.
discard	Indicates that all global temporary tables of the current session are audited.

Table 7-16 audit_type parameters

Parameter	Description
audit_open/ audit_close	Indicates that the audit type is operations enabling or disabling audit logs.
user_login/ user_logout	Indicates that the audit type is operations and users with successful login/logout.
system_start/ system_stop/ system_recover/ system_switch	Indicates that the audit type is system startup, shutdown, and instance switchover.
sql_wait/sql_parse	Indicates that the audit type is SQL statement parsing.
lock_user/unlock_user	Indicates that the audit type is successful user locking and unlocking.
grant_role/ revoke__role	Indicates that the audit type is user permission granting and revoking.
user_violation	Indicates that the audit type is unauthorized user access operations.
ddl_database_object	Indicates that successful DDL operations are audited. DDL operations are controlled at a fine granularity based on operation objects. So, audit_system_object is used to control the objects whose DDL operations are to be audited. (The audit function takes effect as long as audit_system_object is configured, no matter whether ddl is set.) For example, ddl_sequence indicates that the audit type is sequence-related operations.

Parameter	Description
dml_action_insert/ dml_action_delete/ dml_action_update/ dml_action_merge/ dml_action_select	Indicates that the audit type is DML operations such as INSERT , DELETE , UPDATE , and MERGE .
internal_event	Indicates that the audit type is internal events.
user_func	Indicates that the audit type is user-defined functions, stored procedures, or anonymous block operations. NOTE If a user-defined function or stored procedure contains a FETCH statement, the common_text field records the corresponding CURSOR content when the fetch statement is audited.
special_func	Indicates that the audit type is special function invocation. Special functions include pg_terminate_backend and pg_cancel_backend .
copy_to/copy_from	Indicates that the audit type is COPY operations.
set_parameter	Indicates that the audit type is SET operations.
trans_begin/ trans_commit/ trans_prepare/ trans_rollback_to/ trans_release/ trans_savepoint/ trans_commit_prepare / trans_rollback_prepare /trans_rollback	Indicates that the audit type is transaction-related operations.
vacuum/vacuum_full/ vacuum_merge	Indicates that the audit type is VACUUM operations.
analyze/analyze_verify	Indicates that the audit type is ANALYZE operations.
cursor_declare/ cursor_move/ cursor_fetch/ cursor_close	Indicates that the audit type is cursor-related operations.
codeblock_execute	Indicates that the audit type is anonymous blocks.
explain	Indicates that the audit type is EXPLAIN operations.
show	Indicates that the audit type is SHOW operations.
lock_table	Indicates that the audit type is table locking operations.
comment	Indicates that the audit type is COMMENT operations.

Parameter	Description
prepare/execute/deallocate	Indicates that the audit type is PREPARE , EXECUTE , or DEALLOCATE operations.
cluster	Indicates that the audit type is CLUSTER operations.
constraints	Indicates that the audit type is CONSTRAINTS operations.
checkpoint	Indicates that the audit type is CHECKPOINT operations.
barrier	Indicates that the audit type is BARRIER operations.
cleanconn	Indicates that the audit type is CLEAN CONNECTION operations.
seclabel	Indicates that the audit type is security label operations.
notify	Indicates that the audit type is notification operations.
load	Indicates that the audit type is loading operations.
discard	Indicates that the audit type is DISCARD operations.

pgxc_query_audit()

Description: Displays audit logs of all CNs.

Return type: record

The return fields of this function are the same as those of the **pg_query_audit** function.

pg_query_audit_details()

Description: Views audit logs and parses the **object_name** and **object_details** columns in audit logs from JSON format. This function is supported only by clusters of version 8.2.1.100 or later.

Return type: record

The following table describes return columns.

Table 7-17 Columns returned by the pg_query_audit() function

Column	Type	Description
begintime	timestamp with time zone	Operation start time
endtime	timestamp with time zone	Operation end time

Column	Type	Description
operation_type	text	Operation type. For details, see Table 7-15 .
audit_type	text	Audit type. For details, see Table 7-16 .
result	text	Operation result
username	text	Name of the user who performs the operation
database	text	Database name
client_conninfo	text	Client connection information, that is, gsql, JDBC, or ODBC.
transaction_xid	text	Transaction ID
query_id	text	Query ID
node_name	text	Node name
session_id	text	ID of a session.
local_port	text	Local port
remote_port	text	Remote port
object_name	text	Table name, function name, and view name.
column_name	text	Column name
type_of_use	text	Usage type of an object. The options are as follows: 1: reference only (This flag does not exist in actual use.) 2: accessed during execution (columns in the statement and columns accessed during parsing) 4: found in condition statements (excluding condition expressions and functions) 8: found in inner join 16: found in outer join 32: found in aggregation nodes (including distinct, group by, and aggregate functions) 64: found in full join The values in this column are displayed in overlay mode.

Column	Type	Description
use_type	text	Type parsed by type_of_use : 1 : Reference only 2 : Access 4 : Conditional 8 : Inner join 16 : Outer join 32 : Sum 64 : Full join
command_text	text	Command used to perform the operation

Example:

1. Query the column names of all objects and object types used in the statement.

```
SET audit_object_details = on;  
SELECT object_name,object_details,result_rows,error_code,command_text FROM  
pg_query_audit('2023-05-12-03 8:00:00','2023-05-12 22:55:00') where command_text like '%student%';
```

The query result is as follows:

object_name	object_details	result_rows	error_code	command_text
student				
		0		CREATE TABLE student(stuNo int, stuName TEXT);
studentscore				
		0		CREATE TABLE studentscore(stuNo int, stuscore int);
["public.student_view01","public.student","public.student"]				
		0		CREATE OR REPLACE VIEW student_view01 AS SELECT * FROM student t1 where t1.stuNo in (select stuNo from studentscore t2 where t1.stuNo = t2.stuNo);
["public.student_view01","public.student","public.studentscore"] {"public.student":{"stuno":"6"}, {"stuname":"2"}], "public.studentscore":{"stuno":"6"}}		0		SELECT * FROM student_view01

2. The **object_details** column in the query result displays the columns involved in the execution and their usage types. The columns are recorded in JSON format.

You can use the **pg_query_audit_details** function to parse the **object_name** and **object_details** columns.

```
SELECT database,object_name,column_name,type_of_use,use_type FROM  
pg_query_audit_details('2021-02-03 8:00:00','2024-02-03 22:55:00','current') where command_text like  
'%student%';
```

The query result is as follows:

database	object_name	column_name	type_of_use	use_type
gaussdb	student		2	Access
gaussdb			2	Access

```

gaussdb | studentcore | | 2 | Access
gaussdb | | | 2 | Access
gaussdb | public.student_view01 | | 2 | Access
gaussdb | public.studentcore | | 2 | Access
gaussdb | public.student | | 2 | Access
gaussdb | | | 2 | Access
gaussdb | public.student | stuno | 6 | Access,Conditional
gaussdb | public.student | stuname | 2 | Access
gaussdb | public.studentcore | stuno | 6 | Access,Conditional
gaussdb | public.student_view01 | | 2 | Access
gaussdb | public.student | | 2 | Access
gaussdb | public.studentcore | | 2 | Access
gaussdb | | | 2 | Access
(15 rows)

```

pg_delete_audit()

Description: Deletes audit logs generated in a specified period. This function has been deprecated in 8.1.3 and later cluster versions. Return type: void

NOTE

For database security concerns, this function is unavailable. If you call it, the following message is displayed: "ERROR: For security purposes, it is not allowed to manually delete audit logs."

create_audit_csv_foreign_table()

Description: Creates a partitioned foreign table for reading audit logs. You can invoke this function to generate the audit log foreign table **pgxc_audit_logs**. By reading **pgxc_audit_logs**, you can read audit logs stored on OBS. This function is supported only in clusters of 8.3.0 or later.

Input parameter:

- **obs_server**: OBS server name. The value is of the text type.
- **file_path**: OBS file path. The value is of the text type.

Return type: record

Example:

```

-- Create an OBS server.
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  address 'obs.example.xxx:xxx',
  encrypt 'true',
  access_key 'xxxxxxxxx',
  secret_access_key 'xxxxxxxxxxxxxxxxx',
  type 'obs' );
-- Invoke this function to generate a foreign table for audit logs.
SELECT * FROM pg_catalog.create_audit_csv_foreign_table('obs_server','/obs-audit/test/');
create_audit_csv_foreign_table
-----
t
(1 row)
-- Read audit logs stored on OBS.
SELECT * FROM pgxc_audit_logs where year=2023 and month=7 and date=24;
 year | month | date |   begintime   |   endtime   | operation_type | audit_type | result
-----+-----+-----+-----+-----+-----+-----+-----+-----
| username | database | client_conninfo | object_name | object_details | command_text |
detail_info | transaction_xid | query_id | node_name | session_id | local_port |
remote_port | result_rows | error_code
-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
-----

```



```

+-----+-----+-----+-----+-----+-----+-----+-----+
2023 | 7   | 24 | 2023-07-24 09:56:40.727+08 | 2023-07-24 09:56:42.215+08 | dml      |
dml_action_select | ok   | dbadmin | gaussdb | gsqa@[local] | public.pgxc_audit_logs |
select * f
rom pgxc_audit_logs where year=2023 and date=24 and month=7;

```

7.16 Conditional Expression Functions

coalesce(expr1, expr2, ..., exprn)

Description: Returns the first argument that is not **NULL** in the argument list.

COALESCE(expr1, expr2) is equivalent to **CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END**.

Examples:

```
SELECT coalesce(NULL,'hello');
 coalesce
-----
hello
(1 row)
```

 NOTE

- **NULL** is returned only if all parameters are **NULL**.
- This value is replaced by the default value when data is displayed.
- Like a CASE expression, COALESCE only evaluates the parameters that are needed to determine the result. That is, parameters to the right of the first non-null parameter are not evaluated.

decode(base_expr, compare1, value1, Compare2,value2, ... default)

Description: Compares base_expr with each compare(n) and returns value(n) if they are matched. If base_expr does not match each **compare(n)**, the default value is returned.

Examples:

```
SELECT decode('A','A',1,'B',2,0);
case
-----
1
(1 row)
```

if(bool_expr, expr1, expr2)

Description: Returns **expr1** or **expr2**. If the value of **bool_expr** is **true**, **expr1** is returned. Otherwise, **expr2** is returned.

This function is equivalent to **CASE WHEN bool_expr = true THEN expr1 ELSE expr2 END**.

Examples:

```
SELECT if(1 < 2, 'yes', 'no');
if
----
yes
(1 row)
```

 NOTE

expr1 and **expr2** can be of any type. For details about the available types, see [UNION, CASE, and Related Constructs](#).

ifnull(expr1, expr2)

Description: Returns **expr1** or **expr2**. If **expr1** is not **NULL**, **expr1** is returned. Otherwise, **expr2** is returned.

This function is logically equivalent to **CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END**.

Examples:

```
SELECT ifnull(NULL,'hello');
ifnull
-----
hello
(1 row)
```

 NOTE

expr1 and **expr2** can be of any type. For details about the available types, see [UNION, CASE, and Related Constructs](#).

isnull(expr)

Description: Checks whether **expr** is **NULL**. If it is **NULL**, **true** is returned. Otherwise, **false** is returned.

This function is logically equivalent to **expr IS NULL**.

Examples:

```
SELECT isnull(NULL), isnull('abc');
isnull | isnull
-----+-----
t      | f
(1 row)
```

nullif(expr1, expr2)

Description: Returns **NULL** or **expr1**. If **expr1** is equal to **expr2**, **NULL** is returned. Otherwise, **expr1** is returned.

nullif(expr1, expr2) is equivalent to **CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END**.

Examples:

```
SELECT nullif('hello','world');
nullif
-----
hello
(1 row)
```

Note:

Assume the two parameter data types are different:

- If implicit conversion exists between the two data types, implicitly convert the parameter of lower priority to this data type using the data type of higher

priority. If the conversion succeeds, computation is performed. Otherwise, an error is returned. For example:

```
SELECT nullif('1234'::VARCHAR,123::INT4);
nullif
-----
1234
(1 row)
SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
```

- If implicit conversion is not applied between two data types, an error is displayed. For example:

```
SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE);
ERROR:  operator does not exist: boolean = timestamp without time zone
LINE 1: SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE) FROM DUAL;
        ^
HINT:  No operator matches the given name and argument type(s). You might need to add explicit type casts.
```

nvl(expr1 , expr2)

Description: Returns **expr2** if **expr1** is **NULL**. If **expr1** is not **NULL**, **expr1** is returned.

Examples:

```
SELECT nvl('hello','world');
nvl
-----
hello
(1 row)
```

NOTE

Parameters **expr1** and **expr2** can be of any data type. If **expr1** and **expr2** are of different data types, NVL checks whether **expr2** can be implicitly converted to **expr1**. If it can, the **expr1** data type is returned. If **expr2** cannot be implicitly converted to **expr1** but **expr1** can be implicitly converted to **expr2**, the **expr2** data type is returned. If no implicit type conversion exists between the two parameters and the parameters are different data types, an error is reported.

sys_context('namespace' , 'parameter')

Description: Obtains and returns the parameter values of a specified **namespace**.

Return type: text

Examples:

```
SELECT sys_context('USERENV', 'CURRENT_SCHEMA');
sys_context
-----
public
(1 row)
```

The result varies according to the current actual schema.

NOTE

Currently, only the SYS_CONTEXT('USERENV','CURRENT_SCHEMA') and SYS_CONTEXT('USERENV','CURRENT_USER') formats are supported.

greatest(expr1 [, ...])

Description: Selects the largest value from a list of any number of expressions.

- In Oracle- or Teradata-compatible mode, the returned result is the maximum value of all non-null parameters.
- In MySQL-compatible mode, **null** is returned if an input parameter contains **null**.

Examples:

```
SELECT greatest(1*2,2-3,4-1);
greatest
-----
      3
(1 row)
SELECT greatest('ABC', 'BCD', 'CDE');
greatest
-----
CDE
(1 row)
```

least(expr1 [, ...])

Description: Selects the smallest value from a list of any number of expressions.

- In Oracle- or Teradata-compatible mode, the returned result is the minimum value of all non-null parameters.
- In MySQL-compatible mode, **null** is returned if an input parameter contains **null**.

Examples:

```
SELECT least(1*2,2-3,4-1);
least
-----
    -1
(1 row)
SELECT least('ABC','BCD','CDE');
least
-----
ABC
(1 row)
```

EMPTY_BLOB()

Description: Initiates a BLOB variable in an INSERT or an UPDATE statement to a NULL value.

Return type: BLOB

Examples:

```
-- Create a table:
CREATE TABLE blob_tb(b blob,id int) DISTRIBUTE BY REPLICATION;
-- Insert data:
INSERT INTO blob_tb VALUES (empty_blob(),1);
--Delete the table.
DROP TABLE blob_tb;
```

NOTE

The length obtained by using **DBMS.GETLENGTH** is 0.

7.17 Range Functions and Operators

7.17.1 Range Operators

=

Description: Equals

Example:

```
SELECT int4range(1,5) = '[1,4]':int4range AS RESULT;  
result  
-----  
t  
(1 row)
```

<>

Description: Does not equal to

Example:

```
SELECT numrange(1.1,2.2) <> numrange(1.1,2.3) AS RESULT;  
result  
-----  
t  
(1 row)
```

<

Description: Is less than

Example:

```
SELECT int4range(1,10) < int4range(2,3) AS RESULT;  
result  
-----  
t  
(1 row)
```

>

Description: Is greater than

Example:

```
SELECT int4range(1,10) > int4range(1,5) AS RESULT;  
result  
-----  
t  
(1 row)
```

<=

Description: Is less than or equals

Example:

```
SELECT numrange(1.1,2.2) <= numrange(1.1,2.2) AS RESULT;  
result  
-----  
t  
(1 row)
```

>=

Description: Is greater than or equals

Example:

```
SELECT numrange(1.1,2.2) >= numrange(1.1,2.0) AS RESULT;  
result  
-----  
t  
(1 row)
```

@>

Description: The object on the left includes the object on the right.

Example:

```
SELECT int4range(2,4) @> int4range(2,3) AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT '[2011-01-01,2011-03-01)>::tsrange @> '2011-01-10':::timestamp AS RESULT;  
result  
-----  
t  
(1 row)
```

<@

Description: The object on the right includes the object on the left.

Example:

```
SELECT int4range(2,4) <@ int4range(1,7) AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 42 <@ int4range(1,7) AS RESULT;  
result  
-----  
f  
(1 row)
```

&&

Description: Overlap (have points in common)

Example:

```
SELECT int8range(3,7) && int8range(4,12) AS RESULT;  
result  
-----  
t  
(1 row)
```

<<

Description: Strictly left of

Example:

```
SELECT int8range(1,10) << int8range(100,110) AS RESULT;  
result  
-----  
t  
(1 row)
```

>>

Description: Strictly right of

Example:

```
SELECT int8range(50,60) >> int8range(20,30) AS RESULT;  
result  
-----  
t  
(1 row)
```

&<

Description: Does not extend to the right of

Example:

```
SELECT int8range(1,20) &< int8range(18,20) AS RESULT;  
result  
-----  
t  
(1 row)
```

&>

Description: Does not extend to the left of

Example:

```
SELECT int8range(7,20) &> int8range(5,10) AS RESULT;  
result  
-----  
t  
(1 row)
```

-|-

Description: Is adjacent to

Example:

```
SELECT numrange(1.1,2.2) -|- numrange(2.2,3.3) AS RESULT;  
result  
-----  
t  
(1 row)
```

+

Description: Union

Example:

```
SELECT numrange(5,15) + numrange(10,20) AS RESULT;  
result  
-----  
[5,20)  
(1 row)
```

*

Description: Intersection

Example:

```
SELECT int8range(5,15) * int8range(10,20) AS RESULT;  
result  
-----  
[10,15)  
(1 row)
```

-

Description: Difference

Example:

```
SELECT int8range(5,15) - int8range(10,20) AS RESULT;  
result  
-----  
[5,10)  
(1 row)
```

NOTE

- The simple comparison operators <, >, <=, and >= compare the lower bounds first, and only if those are equal, compare the upper bounds.
- The <<, >>, and -|- operators always return false when an empty range is involved; that is, an empty range is not considered to be either before or after any other range.
- The union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges.

7.17.2 Range Functions

lower(anyrange)

Description: Lower bound of range

Return type: Range's element type

Example:

```
SELECT lower(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
1.1  
(1 row)
```

upper(anyrange)

Description: Upper bound of range

Return type: Range's element type

Example:

```
SELECT upper(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
2.2  
(1 row)
```

isempty(anyrange)

Description: Is the range empty?

Return type: boolean

Example:

```
SELECT isempty(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
f  
(1 row)
```

lower_inc(anyrange)

Description: Is the lower bound inclusive?

Return type: boolean

Example:

```
SELECT lower_inc(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
t  
(1 row)
```

upper_inc(anyrange)

Description: Is the upper bound inclusive?

Return type: boolean

Example:

```
SELECT upper_inc(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
f  
(1 row)
```

lower_inf(anyrange)

Description: Is the lower bound infinite?

Return type: boolean

Example:

```
SELECT lower_inf('(')::daterange AS RESULT;  
result  
-----  
t  
(1 row)
```

upper_inf(anyrange)

Description: Is the upper bound infinite?

Return type: boolean

Example:

```
SELECT upper_inf('(',')::daterange) AS RESULT;  
result  
-----  
t  
(1 row)
```

NOTE

The **lower** and **upper** functions return null if the range is empty or the requested bound is infinite. The **lower_inc**, **upper_inc**, **lower_inf**, and **upper_inf** functions all return false for an empty range.

7.18 Data Redaction Functions

Data redaction functions are used to mask and protect sensitive data. Generally, you are advised to bind these functions to the columns to be redacted based on the data redaction syntax, rather than use them directly on query statements.

mask_none(column_name)

Description: Masks no data (for internal tests only).

Return type: same as **column_name**

mask_full(column_name)

Description: Replaces all data with a fixed value. The fixed value varies depending on the data type of the redacted column.

Return type: same as **column_name**

mask_partial(column_name, mask_digital, mask_from[, mask_to])

Description: Replaces the digits from the **mask_from** to **mask_to** position in a number with the digit specified by **mask_digital**. The default value of **mask_to** can be used, which indicates that the digits from the **mask_from** position to the end of the number are replaced. **mask_digital** can only be a digit from 0 to 9.

Return type: same as **column_name**

mask_partial(column_name [, input_format, output_format], mask_char, mask_from[, mask_to])

Description: Replaces the digits from the **mask_from** to **mask_to** position in a string with the character specified by **mask_char** based on the given input and output formats.

Parameter description:

- **input_format**
The input format is a character string of V and F, whose length is the same as that of the data in the redacted column. Characters in positions corresponding to V may be masked, and characters in positions corresponding to F are skipped. The V character string specifies which characters are to be masked. The input and output formats apply to data with a fixed length, such as bank card numbers, ID card numbers, and phone numbers.
- **output_format**
The output format is a character string of V and any other character, whose length is the same as that of the data in the redacted column. V characters correspond to those in the **input_format**, and other characters correspond to the F characters in the **input_format**.
For parameters **input_format** and **output_format**, you can use their default values or set them to "". In this case, there is no requirement for the input or output format, and the whole string will be masked.
- **mask_char**
Masking character, which can be any one character, for example, an asterisk (*) or a number sign (#).
- **mask_from**
First character in the string that will be masked. The value must be greater than 0.
- **mask_to**
Last character in the string that will be masked. The default value can be used, which indicates that the character from the **mask_from** position to the last character of the string will be masked.

Return type: same as **column_name**

mask_partial(column_name, mask_field1, mask_value1, mask_field2, mask_value2, mask_field3, mask_value3)

Description: Masks a date or time based on three specified fields. If **mask_value** is -1, the corresponding **mask_field** is not masked. **mask_field** can be **month**, **day**, **year**, **hour**, **minute**, or **second**. The value range of each field must be within that of the actual time unit.

Return type: same as **column_name**

NOTE

Redaction functions are recommended if you want to create redaction policies.
For how to use data redaction functions, see [Data Redaction](#).

User-Defined Redaction Functions

You can use the PL/pgSQL language to customize redaction functions.

User-defined redaction functions must meet the following requirements:

- The return type must be the same as the data type of the redacted column.
- The functions can be pushed down.

- In addition to the redaction format, only one column can be specified in the argument list for data redaction.
- The functions only implement the formatting for specific data types and do not involve complex association operations with other table objects.

If either of the first two requirements is not met, an error will be reported when you create a redaction policy. If either of the last two requirements is not met, unexpected problems may occur in query execution results.

Deleting the bound user-defined data redaction function will cause an error during query execution due to an invalid redaction policy.

7.19 Bitmap Functions and Operators

7.19.1 Roaring Bitmap Operators

Starting from DWS 8.1.3, user profiling and precision marketing scenarios benefit from the utilization of efficient bitmap processing operators. This enhancement significantly enhances query performance.

=

Description: Compares two Roaring bitmaps to check whether they are equal.

Return type: bool

Example:

```
SELECT rb_build('{1,2,3}') = rb_build('{1,2,3}');
?column?
-----
t
(1 row)
SELECT rb_build('{2,3}') = rb_build('{1,2,3}');
?column?
-----
f
(1 row)
```

<>

Description: Compares two Roaring bitmaps to check whether they are unequal.

Return type: bool

Example:

```
SELECT rb_build('{1,2,3}') <> rb_build('{1,2,3}');
?column?
-----
f
(1 row)
SELECT rb_build('{2,3}') <> rb_build('{1,2,3}');
?column?
-----
t
(1 row)
```

&

Description: Calculates the intersection of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3}') & rb_build('{1,2,3}'));
rb_to_array
-----
{2,3}
(1 row)
```

|

Description: Calculates the union of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3}') | rb_build('{1,2,3}'));
rb_to_array
-----
{1,2,3}
(1 row)
```

|

Description: Calculates the result of adding an ID to a Roaring bitmap.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3}') | 4);
rb_to_array
-----
{2,3,4}
(1 row)
```

#

Description: XOR result of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3}') # rb_build('{1,2,3}'));
rb_to_array
-----
{1}
(1 row)
```

-

Description: Calculates the result set in the first Roaring bitmap but not in the second Roaring bitmap.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3,4}') - rb_build('{1,2,3}'));
rb_to_array
-----
{4}
(1 row)
```

-

Description: The result set of removing a specified ID from a Roaring bitmap.

Return type: RoaringBitmap

Example:

```
SELECT rb_to_array(rb_build('{2,3,4}') - 3);
rb_to_array
-----
{2,4}
(1 row)
```

@>

Description: Determines whether the Roaring bitmap before an operator contains the Roaring bitmap after the operator.

Return type: bool

Example:

```
SELECT rb_build('{2,3,4}') @> rb_build('{2,3}');
?column?
-----
t
(1 row)
SELECT rb_build('{2,3,4}') @> 4;
?column?
-----
t
(1 row)
```

<@

Description: Determines whether the Roaring bitmap before an operator is contained in the Roaring bitmap after the operator.

Return type: bool

Example:

```
SELECT 4 <@ rb_build('{2,3,4}');
?column?
-----
t
(1 row)
SELECT rb_build('{2,3,4}') <@ rb_build('{2,3}');
?column?
-----
f
(1 row)
```

&&

Description: If two Roaring bitmaps overlap, **true** is returned. Otherwise, **false** is returned.

Return type: bool

Example:

```
SELECT rb_build('{2,3,4}') && rb_build('{2,3}');
?column?
-----
t
(1 row)
SELECT rb_build('{2,3,4}') && rb_build('{7,8,9}');
?column?
-----
f
(1 row)
```

7.19.2 Bitmap Functions

Since 8.1.3, DWS supports efficient bitmap processing functions and operators, which can be used in user profiling and precision marketing, greatly improving query performance.

rb_build(array)

Description: Converts an int array to the RoaringBitmap type.

Return type: RoaringBitmap

Example:

```
SELECT rb_build('{1,2,3}');
rb_build
-----
\x3a300000010000000000020010000000010002000300
(1 row)
CREATE TABLE r_row (a int, b text, c roaringbitmap);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

INSERT INTO r_row values (1, 'a', rb_build('{1,2,3}'));
INSERT 0 1

SELECT * FROM r_row;
 a | b | c
-----+-----
 1 | a | \x3a300000010000000000020010000000010002000300
(1 row)

INSERT INTO r_row values (2, 'b', rb_build('{}'));
INSERT 0 1

SELECT * FROM r_row;
 a | b | c
-----+-----
 2 | b | \x3a3000000000000000
 1 | a | \x3a300000010000000000020010000000010002000300
(2 rows)
```

rb_iterate(roaringbitmap)

Description: Converts roaringbitmap data into int data and outputs the data in multiple lines.

Return type: record (int value in multiple rows)

Example:

```
postgres=#SELECT rb_iterate(c) FROM r_row;
rb_iterate
-----
1
2
3
(3 rows)
```

rb_to_array(roaringbitmap)

Description: Using rb_build reverse operation to convert roaringBitmap into an int array.

Return type: array

Example:

```
postgres=#SELECT rb_to_array(c) FROM r_row;
rb_to_array
-----
{1,2,3}
(1 row)
postgres=#SELECT rb_to_array("\x3a30000001000000000020010000000010002000300");
rb_to_array
-----
{1,2,3}
(1 row)
```

rb_and(roaringbitmap, roaringbitmap)

Description: Calculates the intersection of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_and(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-----
{2,3}
(1 row)
```

rb_or(roaringbitmap, roaringbitmap)

Description: Calculates the union of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_or(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-----
{1,2,3,4}
(1 row)
```


rb_xor(roaringbitmap, roaringbitmap)

Description: Calculates the XOR of two Roaring bitmaps.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_xor(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-----
{1,4}
(1 row)
```

rb_andnot(roaringbitmap, roaringbitmap)

Description: Sets in the first Roaring bitmap set but not in the second Roaring bitmap set.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_andnot(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-----
{1}
(1 row)
```

rb_cardinality(roaringbitmap)

Description: Calculates the cardinality of a Roaring bitmap.

Return type: int

Example:

```
postgres=#SELECT rb_cardinality(rb_build('{1,2,3}'));
rb_cardinality
-----
3
(1 row)
```

rb_and_cardinality(roaringbitmap, roaringbitmap)

Description: Calculates the cardinality of the intersection of two Roaring bitmaps.

Return type: int

Example:

```
postgres=#SELECT rb_and_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_and_cardinality
-----
2
(1 row)
```

rb_or_cardinality(roaringbitmap, roaringbitmap)

Description: Calculates the cardinality of the union of two Roaring bitmaps.

Return type: int

Example:

```
postgres=#SELECT rb_or_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_or_cardinality
-----
4
(1 row)
```

rb_xor_cardinality(roaringbitmap, roaringbitmap)

Description: Calculates the cardinality of two Roaring bitmaps after the XOR operation.

Return type: int

Example:

```
postgres=#SELECT rb_xor_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_xor_cardinality
-----
2
(1 row)
```

rb_andnot_cardinality(roaringbitmap, roaringbitmap)

Description: Calculates the cardinality of two Roaring bitmaps after ANDNOT operation.

Return type: int

Example:

```
postgres=#SELECT rb_andnot_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_andnot_cardinality
-----
1
(1 row)
```

rb_is_empty(roaringbitmap)

Description: Determines whether a Roaring bitmap is empty.

Return type: bool

Example:

```
postgres=#SELECT rb_is_empty(rb_build('{1,2,3}'));
rb_is_empty
-----
f
(1 row)
```

rb_equals(roaringbitmap, roaringbitmap)

Description: Determines whether two Roaring bitmaps are equal.

Return type: bool

Example:

```
postgres=#SELECT rb_equals(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_equals
-----
```

```
f  
(1 row)
```

rb_intersect(roaringbitmap, roaringbitmap)

Description: Determines whether two Roaring bitmaps are intersected.

Return type: bool

Example:

```
postgres=#SELECT rb_intersect(rb_build('{1,2,3}'), rb_build('{2,3,4}'));  
rb_intersect  
-----  
t  
(1 row)
```

rb_min(roaringbitmap)

Description: Returns the minimum value in a Roaring bitmap.

Return type: int

Example:

```
postgres=#SELECT rb_min(rb_build('{1,2,3}'));  
rb_min  
-----  
1  
(1 row)
```

rb_max(roaringbitmap)

Description: Returns the maximum value in a Roaring bitmap.

Return type: int

Example:

```
postgres=#SELECT rb_max(rb_build('{1,2,3}'));  
rb_max  
-----  
3  
(1 row)
```

rb_add(roaringbitmap, int)

Description: Adds an element to a Roaring bitmap.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_add(rb_build('{1,3}'), 2));  
rb_to_array  
-----  
{1,2,3}  
(1 row)
```

rb_added(int, roaringbitmap)

Description: Adds an element to a Roaring bitmap.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_added(2, rb_build('{1,3}')));
rb_to_array
-----
{1,2,3}
(1 row)
```

rb_contain(roaringbitmap,int)

Description: Determines whether a Roaring bitmap contains the specified element.

Return type: bool

Example:

```
postgres=#SELECT rb_contain(rb_build('{1,3}'), 2);
rb_contain
-----
f
(1 row)
```

rb_containedby(int,roaringbitmap)

Description: Determines whether the given element is included in a given Roaring bitmap.

Example:

```
postgres=#SELECT rb_containedby(2,rb_build('{1,3}'));
rb_containedby
-----
f
(1 row)
```

rb_contain_rb(roaringbitmap,roaringbitmap)

Description: Determines whether the first Roaring bitmap contains the second Roaring bitmap.

Return type: bool

Example:

```
postgres=#SELECT rb_contain_rb(rb_build('{1,3}'), rb_build('{2,3}'));
rb_contain_rb
-----
f
(1 row)
```

rb_containedby_rb(roaringbitmap,roaringbitmap)

Description: Determines whether the second Roaring bitmap contains the first Roaring bitmap.

Return type: bool

Example:

```
postgres=#SELECT rb_containedby_rb(rb_build('{1,3}'), rb_build('{2,3}'));
rb_containedby_rb
```

```
-----  
f  
(1 row)
```

rb_remove(roaringbitmap,int)

Description: Removes elements from a Roaring bitmap.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT rb_to_array(rb_remove(rb_build('{1,3}'),1));  
rb_to_array  
-----  
{3}  
(1 row)
```

rb_clear(roaringbitmap,int,int)

Description: Clears elements within a specified range from roaring bitmaps.

Return type: RoaringBitmap

Example:

```
postgres=#SELECT  
rb_to_array(rb_clear(rb_build('{1,2,3}'),1,2));  
rb_to_array  
-----  
{2,3}  
  
(1 row)
```

rb_flip(roaringbitmap,int,int)

Description: Reverses elements in a specified range.

Example:

```
postgres=#SELECT rb_to_array(rb_flip(rb_build('{1,2,3,7,9}'), 1,10));  
rb_to_array  
-----  
{4,5,6,8,10}  
(1 row)
```

rb_rank(roaringbitmap,int)

Description: Returns the cardinality of the set of values less than the specified value.

Return type: int

Example:

```
postgres=#SELECT rb_rank(rb_build('{1,10,100}'),99);  
rb_rank  
-----  
2  
(1 row)
```

7.19.3 Roaring Bitmap Aggregation Functions

Since 8.1.3, DWS supports efficient bitmap processing functions and operators, which can be used in user profiling and precision marketing, greatly improving query performance.

rb_build_agg(int)

Description: Aggregates int values in a group into a RoaringBitmap value.

Return type: RoaringBitmap

Example:

```
CREATE TABLE t1 (a int ,b int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
INSERT INTO t1 SELECT generate_series(1,10),generate_series(1,20,2);
INSERT 0 10
SELECT rb_iterate(rb_build_agg(b)) FROM t1;
rb_iterate
-----
1
3
5
7
9
11
13
15
17
19
(10 rows)
```

rb_and_agg(roaringbitmap)

Description: Aggregates data of the Roaring bitmaps in a group into a Roaring bitmap set based on the INTERSECT operation.

Example:

```
CREATE TABLE r1(a int ,b roaringbitmap);
INSERT INTO r1 SELECT a, rb_build_agg(b) FROM t1 GROUP BY a;
INSERT INTO t1 SELECT generate_series(1,10),generate_series(1,20,4);
INSERT INTO r1 SELECT a, rb_build_agg(b) FROM t1 GROUP BY a;
SELECT a, rb_to_array(rb_and_agg(b)) FROM r1 GROUP BY a ORDER BY a;
a | rb_to_array
-----+-----
1 | {1}
2 | {3}
3 | {5}
4 | {7}
5 | {9}
6 | {11}
7 | {13}
8 | {15}
9 | {17}
10 | {19}
(10 rows)
```

rb_or_agg(roaringbitmap)

Description: Combines Roaring bitmaps in a group into one Roaring bitmap based on the UNION logic.

Example:

```
SELECT a, rb_to_array(rb_or_agg(b)) FROM r1 GROUP BY a ORDER BY a;
a | rb_to_array
-----+-----
1 | {1}
2 | {3,5}
3 | {5,9}
4 | {7,13}
5 | {9,17}
6 | {1,11}
7 | {5,13}
8 | {9,15}
9 | {13,17}
10 | {17,19}
(10 rows)
```

rb_xor_agg(roaringbitmap)

Description: Combines Roaring bitmaps in a group into one Roaring bitmap based on the XOR logic.

Example:

```
SELECT a, rb_to_array(rb_xor_agg(b)) FROM r1 GROUP BY a ORDER BY a;
a | rb_to_array
-----+-----
1 | {}
2 | {5}
3 | {9}
4 | {13}
5 | {17}
6 | {1}
7 | {5}
8 | {9}
9 | {13}
10 | {17}
(10 rows)
```

rb_and_cardinality_agg(roaringbitmap)

Description: Cardinality of Roaring bitmaps in a group calculated based on INTERSECT.

Return type: int

Example:

```
SELECT a, rb_and_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
a | rb_and_cardinality_agg
-----+-----
1 | 1
2 | 1
3 | 1
4 | 1
5 | 1
6 | 1
7 | 1
8 | 1
9 | 1
```

10	1
(10 rows)	

rb_or_cardinality_agg(roaringbitmap)

Description: Cardinality of Roaring bitmaps in a group calculated based on UNION.

Example:

```
SELECT a, rb_or_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
```

a	rb_or_cardinality_agg

1	1
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2
(10 rows)	

rb_xor_cardinality_agg(roaringbitmap)

Description: Cardinality of Roaring bitmaps in a group calculated based on XOR.

Example:

```
SELECT a, rb_xor_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
```

a	rb_xor_cardinality_agg

1	0
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
(10 rows)	

7.19.4 Use Cases

Background

Currently, real-time precision marketing is required in the Internet, education, and gaming industries. User profiling enables user search based on combined criteria.
Example:

- Before launching a sales promotion, e-commerce companies need to send the promotion message to a selected batch of users with specific features.
- In education, exercise questions need to be pushed based on students' weaknesses.
- On search, video, and portal websites, different contents are pushed to users based on their interests.

These use cases have the following characteristics in common:

- The data volume is huge and the calculation workload is huge.
- There are a large number of users with a lot of labels and fields, occupying a large amount of storage space.
- The feature conditions for selection are diversified, and it is difficult to find a fixed index. If each field has an index, that will occupy too much storage space.
- High performance is required because real-time marketing requires response in seconds.
- Data update has high requirements on timeliness, and user profiles need to be updated in real time.

Roaring bitmaps in DWS can efficiently generate, compress, and parse bitmap data, and supports the most common bitmap aggregation operations (AND, OR, NOT, and XOR). This feature meets the requirements of real-time precision marketing and quick user selection in the case of a large amount of data with hundreds of millions of users and tens of millions of labels.

Using roaringbitmap

Assume that there is a web page browsing information table **userinfo**. The fields in the table are as follows:

```
CREATE TABLE userinfo
(userid int,
age int,
gender text,
salary int,
hobby text
)WITH (orientation=column);
```

The data in the **userinfo** table increases with the change of user information. For example, if a user has multiple hobby attributes, there will be multiple records in the **userinfo** table.

If a user wants to filter out males with income greater than CNY10,000, age greater than 30, and a hobby of phishing, and then push specific messages to these target groups.

The traditional method is to directly query the original table. The statement is as follows:

```
SELECT distinct userid FROM userinfo WHERE salary > 10000 AND age > 30 AND gender ='m' AND hobby
='fishing';
```

If the **userinfo** table contains a small amount of data, indexes are created in the **salary**, **age**, **gender**, and **hobby** columns to meet the query requirements. However, if the **userinfo** table contains a large amount of data and a large number of labels, the preceding statement cannot meet the requirements. The reasons are as follows:

- A large number of indexes need to be created.
- The count (distinct) performance is poor.

Roaring bitmaps perform better in this case.

1. Create a RoaringBitmap table.

```
CREATE TABLE userinfoet
( age int,
```

```
gender text,  
salary int,  
hobby text,  
userset roaringbitmap,  
PRIMARY KEY(age,gender,salary,hobby)  
)WITH (orientation=column);
```

2. All data in the **userinfo** table must be aggregated to the **userinfoaset** table through the aggregation of the label column. You can run the following command to aggregate all data. Or you can aggregate only incremental data. To aggregate only incremental data, a set of users with the same label are put in a table record. This can be implemented by using the UPSERT function. Frequent update operations may generate a large amount of dirty data. Therefore, you are advised to create the **userinfoaset** table as a row-store table to aggregate incremental data.

```
INSERT INTO userinfoaset  
SELECT age, gender, salary, hobby, rb_build_agg(userid)  
FROM  
userinfo  
GROUP BY age, gender, salary, hobby;
```

3. Query the **userinfoaset** table for the selected user information.

```
SELECT rb_iterate(rb_or_agg(userset)) FROM userinfoaset WHERE salary > 10000 AND age > 30 AND  
gender = 'm' AND hobby = 'fishing';
```

After data aggregation, the data volume of the table **userinfoaset** is much smaller than that of the source table, so the scanning performance of the base table is much faster. In addition, based on the advantages of Roaring bitmaps, the performance of calculating **rb_or_agg** and **rb_iterate** is better. Compared with the traditional method, the performance is significantly improved.

7.20 UUID Functions

UUID functions are used to generate UUID data (see [UUID Type](#)).

uuid_generate_v1()

Description: Generates a UUID sequence number.

Return type: UUID

Example:

```
SELECT uuid_generate_v1();  
      uuid_generate_v1  
-----  
c71ceaca-a175-11e9-a920-797ff7000001  
(1 row)
```

NOTE

The **uuid_generate_v1** function generates a UUID based on the time information, cluster node ID, and ID of the thread that generates the sequence. The UUID is globally unique in a single cluster, however, there is a possibility that the time information, cluster node IDs, thread IDs, and clock sequences of multiple clusters are the same at the same time. Therefore, there is a low probability that UUIDs generated among multiple clusters are duplicate.

uuid()

Description: Generates a UUID sequence number. This function is a MySQL compatibility function supported only by version 8.2.0 and later clusters.

Return type: UUID

Example:

```
SELECT uuid();
      uuid
-----
6327dc96-f0e7-0100-f2f2-6c9ff700fffe
(1 row)
```

NOTE

UUID is generated in the way the [uuid_generate_v1\(\)](#) function works. The function generates a UUID based on the time information, cluster node ID, and ID of the thread that generates the sequence. The UUID is globally unique in a single cluster, however, there is a possibility that the time information, cluster node IDs, thread IDs, and clock sequences of multiple clusters are the same at the same time. Therefore, there is a low probability that UUIDs generated among multiple clusters are duplicate.

sys_guid()

Description: Generates an Oracle GUID, which is similar to the UUID. This function is compatible with Oracle.

Return type: text

Example:

```
SELECT sys_guid();
      sys_guid
-----
4EBD3C74A17A11E9A1BF797FF7000001
(1 row)
```

NOTE

The data generation principle of the sys_guid function is the same as that of the uuid_generate_v1 function.

UUID Function Application Example

A UUID is globally unique. It can be used as a primary key or a distribution column in a data table. When [uuid_generate_v1\(\)](#) is used as the default value of the distribution column in a data table, data can be evenly distributed to each DN through hash distribution to prevent data skew.

NOTE

The obvious advantage of UUID is that it is globally unique and does not require a central node. UUIDs are generated independently on a single node. However, UUIDs occupy more storage space than INTs, the index efficiency is low, and the generated IDs are random, making them difficult to identify. Therefore, you need to select UUID or Sequence as the primary key of the data table based on the site requirements.

An example is as follows:

1. The INT type is used as the distribution column.

An example hash table **mytable01** is created, and the ints are used as the distribution column. After data is inserted, data skew occurs.

```
CREATE TABLE mytable01(a INT, b INT) DISTRIBUTE BY hash(a);
```

```
CREATE TABLE
```

```
INSERT INTO mytable01 VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable01 VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable01 VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable01 VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable01 VALUES(1, 10);
```

```
INSERT 0 1
```

```
SELECT * FROM mytable01;
```

```
 a | b
```

```
----+----
```

```
1 | 10
```

```
1 | 10
```

```
1 | 10
```

```
1 | 10
```

```
1 | 10
```

```
(5 rows)
```

```
SELECT table_skewness('mytable01');
```

```
 table_skewness
```

```
-----
```

```
("dn_6003_6004      ",5,100.000%)
```

```
("dn_6001_6002      ",0,0.000%)
```

```
("dn_6005_6006      ",0,0.000%)
```

```
(3 rows)
```

2. The UUIDs are used as the distribution column.

Create an example hash table **mytable02** and use the UUIDs as the distribution column. After data is inserted, data distribution is normal.

```
CREATE TABLE mytable02 (id UUID default uuid_generate_v1(), a INT, b INT) DISTRIBUTE BY hash(id);
```

```
CREATE TABLE
```

```
INSERT INTO mytable02(a, b) VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable02(a, b) VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable02(a, b) VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable02(a, b) VALUES(1, 10);
```

```
INSERT 0 1
```

```
INSERT INTO mytable02(a, b) VALUES(1, 10);
```

```
INSERT 0 1
```

```
SELECT * FROM mytable02;
```

```
 id | a | b
```

```
-----+-----+-----
```

```
63e45c14-cc74-0e00-e9aa-0a2c3fa0fffe | 1 | 10
```

```
63e45c1f-4d18-0700-e9ab-0a2c3fa0fffe | 1 | 10
```

```
63e45c26-f859-0b00-e9ad-0a2c3fa0fffe | 1 | 10
```

```
63e45c23-9e5d-0300-e9ac-0a2c3fa0fffe | 1 | 10
```

```
63e45c2a-5825-0600-e9ae-0a2c3fa0fffe | 1 | 10
```

```
(5 rows)
```

```
SELECT table_skewness('mytable02');
```

```
 table_skewness
```

```
-----
```

```
("dn_6001_6002      ",3,60.000%)
```

```
("dn_6003_6004      ",2,40.000%)
```

```
("dn_6005_6006      ",0,0.000%)
```

```
(3 rows)
```

7.21 Text Search Functions and Operators

7.21.1 Text Search Operators

@@

Description: Specifies whether the **tsvector**-typed words match the **tsquery**-typed words.

Example:

```
SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') AS RESULT;
result
-----
t
(1 row)
```

@@@

Description: Synonym for @@

Example:

```
SELECT to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat') AS RESULT;
result
-----
t
(1 row)
```

&&

Description: Performs the AND operation on two **tsquery**-typed words.

Example:

```
SELECT 'fat | rat'::tsquery && 'cat'::tsquery AS RESULT;
result
-----
( 'fat' | 'rat' ) & 'cat'
(1 row)
```

||

Description: Performs the OR operation on two **tsquery**-typed words.

Example:

```
SELECT 'fat | rat'::tsquery || 'cat'::tsquery AS RESULT;
result
-----
( 'fat' | 'rat' ) | 'cat'
(1 row)
SELECT 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector AS RESULT;
result
-----
'a':1 'b':2,5 'c':3 'd':4
(1 row)
```

!!Description: **NOT** a **tsquery**

Example:

```
SELECT !! 'cat'::tsquery AS RESULT;
result
-----
!'cat'
(1 row)
```

@>Description: Specifies whether a **tsquery**-typed word contains another **tsquery**-typed word.

Example:

```
SELECT 'cat'::tsquery @> 'cat & rat'::tsquery AS RESULT;
result
-----
f
(1 row)
```

<@Description: Specifies whether a **tsquery**-typed word is contained in another **tsquery**-typed word.

Example:

```
SELECT 'cat'::tsquery <@ 'cat & rat'::tsquery AS RESULT;
result
-----
t
(1 row)
```

In addition to the preceding operators, the ordinary B-tree comparison operators (including = and <) are defined for types **tsvector** and **tsquery**.

7.21.2 Text Search Functions

get_current_ts_config()

Description: Gets default text search configuration.

Return type: regconfig

Example:

```
SELECT get_current_ts_config();
get_current_ts_config
-----
english
(1 row)
```

length(tsvector)

Description: Number of lexemes in a **tsvector**-typed word.

Return type: integer

Example:

```
SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);
length
-----
      3
(1 row)
```

numnode(tsquery)

Description: Number of lexemes plus **tsquery** operators

Return type: integer

Example:

```
SELECT numnode('(fat & rat) | cat'::tsquery);
numnode
-----
      5
(1 row)
```

plainto_tsquery([config regconfig ,] query text)

Description: Generates **tsquery** lexemes without punctuation.

Return type: tsquery

Example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
plainto_tsquery
-----
'fat' & 'rat'
(1 row)
```

querytree(query tsquery)

Description: Gets indexable part of a **tsquery**.

Return type: text

Example:

```
SELECT querytree('foo & ! bar'::tsquery);
querytree
-----
'foo'
(1 row)
```

setweight(tsvector, "char")

Description: Assigns weight to each element of **tsvector**.

Return type: tsvector

Example:

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A');
setweight
-----
'cat':3A 'fat':2A,4A 'rat':5A
(1 row)
```

strip(tsvector)

Description: Removes positions and weights from **tsvector**.

Return type: tsvector

Example:

```
SELECT strip('fat:2,4 cat:3 rat:5A':tsvector);
strip
-----
'cat' 'fat' 'rat'
(1 row)
```

to_tsquery([config regconfig ,] query text)

Description: Normalizes words and converts them to **tsquery**.

Return type: tsquery

Example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
-----
'fat' & 'rat'
(1 row)
```

to_tsvector([config regconfig ,] document text)

Description: Reduces document text to **tsvector**.

Return type: tsvector

Example:

```
SELECT to_tsvector('english', 'The Fat Rats');
to_tsvector
-----
'fat':2 'rat':3
(1 row)
```

ts_headline([config regconfig,] document text, query tsquery [, options text])

Description: Highlights a query match.

Return type: text

Example:

```
SELECT ts_headline('x y z', 'z':tsquery);
ts_headline
-----
x y <b>z</b>
(1 row)
```

ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])

Description: Ranks document for query.

Return type: float4

Example:

```
SELECT ts_rank('hello world'::tsvector, 'world'::tsquery);
ts_rank
-----
.0607927
(1 row)
```

ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])

Description: Ranks document for query using cover density.

Return type: float4

Example:

```
SELECT ts_rank_cd('hello world'::tsvector, 'world'::tsquery);
ts_rank_cd
-----
0
(1 row)
```

ts_rewrite(query tsquery, target tsquery, substitute tsquery)

Description: Replaces **tsquery**-typed word.

Return type: tsquery

Example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery);
ts_rewrite
-----
'b' & ( 'foo' | 'bar' )
(1 row)
```

ts_rewrite(query tsquery, select text)

Description: Replaces **tsquery** data in the target with the result of a **SELECT** command.

Return type: tsquery

Example:

```
SELECT ts_rewrite('world'::tsquery, 'select "world"::tsquery, "hello"::tsquery');
ts_rewrite
-----
'hello'
(1 row)
```

7.21.3 Text Search Debugging Functions

ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])

Description: Tests a configuration.

Return type: SETOF record

Example:

```
SELECT ts_debug('english', 'The Brightest supernovae');
           ts_debug
-----
(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
(blank,"Space symbols"," ",{})
(asciiword,"Word, all ASCII",Brightest,{english_stem},english_stem,{brightest})
(blank,"Space symbols"," ",{})
(asciiword,"Word, all ASCII",supernovae,{english_stem},english_stem,{supernova})
(5 rows)
```

ts_lexize(dict regdictionary, token text)

Description: Tests a data dictionary.

Return type: text[]

Example:

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}
(1 row)
```

```
ts_parse(parser_name text, document text, OUT tokid integer, OUT token
text)
```

Description: Tests a parser.

Return type: SETOF record

Example:

```
SELECT ts_parse('default', 'foo - bar');
ts_parse
-----
(1,foo)
(12," ")
(12,"- ")
(1,bar)
(4 rows)
```

ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)

Description: Tests a parser.

Return type: SETOF record

Example:

```
SELECT ts_parse(3722, 'foo - bar');
ts_parse
-----
(1,foo)
(12," ")
(12," - ")
(1,bar)
(4 rows)
```

ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)

Description: Gets token types defined by parser.

Return type: SETOF record

Example:

```
SELECT ts_token_type('default');
       ts_token_type
-----
(1,asciiword,"Word, all ASCII")
(2,word,"Word, all letters")
(3,numword,"Word, letters and digits")
(4,email,"Email address")
(5,url,URL)
(6,host,Host)
(7,sfloat,"Scientific notation")
(8,version,"Version number")
(9,hword_numpart,"Hyphenated word part, letters and digits")
(10,hword_part,"Hyphenated word part, all letters")
(11,hword_asciipart,"Hyphenated word part, all ASCII")
(12,blank,"Space symbols")
(13,tag,"XML tag")
(14,protocol,"Protocol head")
(15,numhword,"Hyphenated word, letters and digits")
(16,asciihword,"Hyphenated word, all ASCII")
(17,hword,"Hyphenated word, all letters")
(18,url_path,"URL path")
(19,file,"File or path name")
(20,float,"Decimal notation")
(21,int,"Signed integer")
(22,uint,"Unsigned integer")
(23,entity,"XML entity")
(23 rows)
```

ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)

Description: Gets token types defined by parser.

Return type: SETOF record

Example:

```
SELECT ts_token_type(3722);
       ts_token_type
-----
(1,asciiword,"Word, all ASCII")
(2,word,"Word, all letters")
(3,numword,"Word, letters and digits")
(4,email,"Email address")
(5,url,URL)
(6,host,Host)
(7,sfloat,"Scientific notation")
(8,version,"Version number")
(9,hword_numpart,"Hyphenated word part, letters and digits")
(10,hword_part,"Hyphenated word part, all letters")
(11,hword_asciipart,"Hyphenated word part, all ASCII")
(12,blank,"Space symbols")
(13,tag,"XML tag")
(14,protocol,"Protocol head")
(15,numhword,"Hyphenated word, letters and digits")
(16,asciihword,"Hyphenated word, all ASCII")
(17,hword,"Hyphenated word, all letters")
```

```
(18,url_path,"URL path")
(19,file,"File or path name")
(20,float,"Decimal notation")
(21,int,"Signed integer")
(22,uint,"Unsigned integer")
(23,entity,"XML entity")
(23 rows)
```

ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)

Description: Gets statistics of a **tsvector** column.

Return type: SETOF record

Example:

```
SELECT ts_stat('select "hello world"::tsvector');
 ts_stat
-----
(world,1,1)
(hello,1,1)
(2 rows)
```

7.22 HLL Functions and Operators

7.22.1 HLL Operators

The HLL type supports the following operators:

Table 7-18 HLL Operators

HLL Operators	Description	Return Type	Example
=	Checks whether the values of hll and hll_hashval are equal.	bool	hll SELECT (hll_empty() hll_hash_integer(1)) = (hll_empty() hll_hash_integer(1)); ?column? ----- t (1 row) hll_hashval SELECT hll_hash_integer(1) = hll_hash_integer(1); ?column? ----- t (1 row)

HLL Operators	Description	Return Type	Example
<> or !=	Checks whether the values of hll and hll_hashval are not equal.	bool	hll SELECT (hll_empty() hll_hash_integer(1)) <> (hll_empty() hll_hash_integer(2)); ?column? ----- t (1 row) hll_hashval SELECT hll_hash_integer(1) <> hll_hash_integer(2); ?column? ----- t (1 row)
	Represents the functions of hll_add , hll_union , and hll_add_rev .	hll	hll_add SELECT hll_empty() hll_hash_integer(1); ?column? ----- \x128b7f8895a3f5af28cafe (1 row) hll_add_rev SELECT hll_hash_integer(1) hll_empty(); ?column? ----- \x128b7f8895a3f5af28cafe (1 row) hll_union SELECT (hll_empty() hll_hash_integer(1)) (hll_empty() hll_hash_integer(2)); ?column? ----- \x128b7f8895a3f5af28cafeda0ce907e4355b60 (1 row)
#	Calculates the distinct value of the hll , which is equivalent to the hll_cardinality function.	integer	SELECT #(hll_empty() hll_hash_integer(1)); ?column? ----- 1 (1 row)

7.22.2 Hash Functions

hll_hash_boolean(bool)

Description: Hashes data of the bool type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_boolean(FALSE);
       hll_hash_boolean
-----
5048724184180415669
(1 row)
```

hll_hash_boolean(bool, int32)

Description: Configures a hash seed (that is, change the hash policy) and hashes data of the bool type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_boolean(FALSE, 10);
       hll_hash_boolean
-----
391264977436098630
(1 row)
```

hll_hash_smallint(smallint)

Description: Hashes data of the smallint type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_smallint(100::smallint);
       hll_hash_smallint
-----
4631120266694327276
(1 row)
```

NOTE

If parameters with the same numeric value are hashed using different data types, the data will differ, because hash functions select different calculation policies for each type.

hll_hash_smallint(smallint, int32)

Description: Configures a hash seed (that is, change the hash policy) and hashes data of the smallint type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_smallint(100::smallint, 10);
       hll_hash_smallint
-----
8349353095166695771
(1 row)
```

hll_hash_integer(integer)

Description: Hashes data of the integer type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_integer(0);
       hll_hash_integer
-----
-3485513579396041028
(1 row)
```

hll_hash_integer(integer, int32)

Description: Hashes data of the integer type and configures a hash seed (that is, change the hash policy).

Return type: hll_hashval

Example:

```
SELECT hll_hash_integer(0, 10);
       hll_hash_integer
-----
183371090322255134
(1 row)
```

hll_hash_bigint(bigint)

Description: Hashes data of the bigint type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_bigint(100::bigint);
       hll_hash_bigint
-----
8349353095166695771
(1 row)
```

hll_hash_bigint(bigint, int32)

Description: Hashes data of the bigint type and configures a hash seed (that is, change the hash policy).

Return type: hll_hashval

Example:

```
SELECT hll_hash_bigint(100::bigint, 10);
       hll_hash_bigint
-----
4631120266694327276
(1 row)
```

hll_hash_bytea(bytea)

Description: Hashes data of the bytea type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_bytea(E'\\x');
       hll_hash_bytea
-----
0
(1 row)
```

hll_hash_bytea(bytea, int32)

Description: Hashes data of the bytea type and configures a hash seed (that is, change the hash policy).

Return type: hll_hashval

Example:

```
SELECT hll_hash_bytea(E'\\x', 10);
 hll_hash_bytea
-----
6574525721897061910
(1 row)
```

hll_hash_text(text)

Description: Hashes data of the text type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_text('AB');
 hll_hash_text
-----
5365230931951287672
(1 row)
```

hll_hash_text(text, int32)

Description: Hashes data of the text type and configures a hash seed (that is, change the hash policy).

Return type: hll_hashval

Example:

```
SELECT hll_hash_text('AB', 10);
 hll_hash_text
-----
7680762839921155903
(1 row)
```

hll_hash_any(anytype)

Description: Hashes data of any type.

Return type: hll_hashval

Example:

```
SELECT hll_hash_any(1);
 hll_hash_any
-----
-8604791237420463362
(1 row)

SELECT hll_hash_any('08:00:2b:01:02:03':macaddr);
 hll_hash_any
-----
-4883882473551067169
(1 row)
```


hll_hash_any(anytype, int32)

Description: Hashes data of any type and configures a hash seed (that is, change the hash policy).

Return type: hll_hashval

Example:

```
SELECT hll_hash_any(1, 10);
      hll_hash_any
-----
-1478847531811254870
(1 row)
```

hll_hashval_eq(hll_hashval, hll_hashval)

Description: Compares two pieces of data of the hll_hashval type to check whether they are the same.

Return type: bool

Example:

```
SELECT hll_hashval_eq(hll_hash_integer(1), hll_hash_integer(1));
      hll_hashval_eq
-----
t
(1 row)
```

hll_hashval_ne(hll_hashval, hll_hashval)

Description: Compares two pieces of data of the hll_hashval type to check whether they are different.

Return type: bool

Example:

```
SELECT hll_hashval_ne(hll_hash_integer(1), hll_hash_integer(1));
      hll_hashval_ne
-----
f
(1 row)
```

7.22.3 Precision Functions

HLL supports **explicit**, **sparse**, and **full** modes. **explicit** and **sparse** excel when the data scale is small, and barely produce errors in calculation results. When the number of distinct values increases, **full** becomes more suitable, but produces some errors. The following functions are used to view precision parameters in HLLs.

hll_schema_version(hll)

Description: Checks the schema version in the current HLL.

Return type: integer

Example:

```
SELECT hll_schema_version(hll_empty());
hll_schema_version
-----
1
(1 row)
```

hll_type(hll)

Description: Checks the type of the current HLL.

Return type: integer

Example:

```
SELECT hll_type(hll_empty());
hll_type
-----
1
(1 row)
```

hll_log2m(hll)

Description: Check the value of log2m of the current HLL. This value affects the error rate in calculating the number of distinct values by the HLL. The formula for calculating the error rate is as follows:

$$\pm 1.04 / \sqrt{2^{\log 2m}}$$

Return type: integer

Example:

```
SELECT hll_log2m(hll_empty());
hll_log2m
-----
11
(1 row)
```

hll_regwidth(hll)

Description: Checks the number of bits of buckets in a hll data structure.

Return type: integer

Example:

```
SELECT hll_regwidth(hll_empty());
hll_regwidth
-----
5
(1 row)
```

hll_expthresh(hll)

Description: Obtains the size of **expthresh** in the current HLL. An HLL usually switches from the **explicit** mode to the **sparse** mode and then to the **full** mode. This process is called the promotion hierarchy policy. You can change the value of **expthresh** to change the policy. For example, if **expthresh** is **0**, an HLL will skip the **explicit** mode and directly enter the **sparse** mode. If the value of **expthresh** is explicitly set to a value ranging from 1 to 7, this function returns $2^{\text{expthresh}}$.

Return type: record

Example:

```
SELECT hll_expthresh(hll_empty());
hll_expthresh
-----
(-1,160)
(1 row)

SELECT hll_expthresh(hll_empty(11,5,3));
hll_expthresh
-----
(8,8)
(1 row)
```

hll_sparseon(hll)

Description: Specifies whether to enable the **sparse** mode. **0** indicates **off** and **1** indicates **on**.

Return type: integer

Example:

```
SELECT hll_sparseon(hll_empty());
hll_sparseon
-----
1
(1 row)
```

7.22.4 Aggregate Functions

hll_add_agg(hll_hashval)

Description: Groups hashed data into HLL.

Return type: hll

Example:

1. Prepare data.

```
CREATE TABLE t_id(id int);
INSERT INTO t_id VALUES(generate_series(1,500));
CREATE TABLE t_data(a int, c text);
INSERT INTO t_data SELECT mod(id,2), id FROM t_id;
```
2. Create another table and specify an HLL column:

```
CREATE TABLE t_a_c_hll(a int, c hll);
```
3. Use **GROUP BY** on column **a** to group data, and insert the data to the HLL column:

```
INSERT INTO t_a_c_hll SELECT a, hll_add_agg(hll_hash_text(c)) FROM t_data GROUP BY a;
```
4. Calculate the number of distinct values for each group in the HLL column:

```
SELECT a, #c as cardinality FROM t_a_c_hll order by a;
a | cardinality
+-----+
0 | 250.741759091658
1 | 250.741759091658
(2 rows)
```

hll_add_agg(hll_hashval, int32 log2m)

Description: Groups hashed data into HLL and sets the **log2m** parameter. The parameter value ranges from 10 to 16.

Return type: hll

Example:

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), 10)) FROM t_data;  
hll_cardinality  
-----  
503.932348927339  
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth)

Description: Groups hashed data into HLL and sets the **log2m** and **regwidth** parameters in sequence. The value of **regwidth** ranges from 1 to 5.

Return type: hll

Example:

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1)) FROM t_data;  
hll_cardinality  
-----  
496.628982624022  
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh)

Description: Groups hashed data into **hll** and sets parameters **log2m**, **regwidth**, and **expthresh** in sequence. The value of **expthresh** is an integer ranging from -1 to 7. **expthresh** is used to specify the threshold for switching from the **explicit** mode to the **sparse** mode. -1 indicates the auto mode; 0 indicates that the **explicit** mode is skipped; a value from 1 to 7 indicates that the mode is switched when the number of distinct values reaches $2^{\text{expthresh}}$.

Return type: hll

Example:

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4)) FROM t_data;  
hll_cardinality  
-----  
496.628982624022  
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)

Description: Groups hashed data into **hll** and sets parameters **log2m**, **regwidth**, **expthresh**, and **sparseon** in sequence. The value of **sparseon** is 0 or 1.

Return type: hll

Example:

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4, 0)) FROM t_data;  
hll_cardinality  
-----  
496.628982624022  
(1 row)
```

hll_union_agg(hll)

Description: Perform the **UNION** operation on multiple pieces of data of the hll type to obtain one HLL.

Return type: hll

Example:

Perform the **UNION** operation on data of the HLL type in each group to obtain one HLL, and calculate the number of distinct values:

```
SELECT #hll_union_agg(c) as cardinality FROM t_a_c_hll;  
cardinality  
-----  
496.628982624022  
(1 row)
```

NOTE

To perform **UNION** on data in multiple HLLs, ensure that the HLLs have the same precision. Otherwise, **UNION** cannot be performed. This restriction also applies to the hll_union(hll, hll) function.

7.22.5 Functional Functions

hll_print(hll)

Description: Prints some debugging parameters of an HLL.

Return type: cstring

Example:

```
SELECT hll_print(hll_empty());  
hll_print  
-----  
EMPTY, nregs=2048, nbits=5, expthresh=-1(160), sparseon=1gongne  
(1 row)
```

hll_empty()

Description: Creates an empty HLL.

Return type: hll

Example:

```
SELECT hll_empty();  
hll_empty  
-----  
\x118b7f  
(1 row)
```

hll_empty(int32 log2m)

Description: Creates an empty HLL and sets the **log2m** parameter. The parameter value ranges from 10 to 16.

Return type: hll

Example:

```
SELECT hll_empty(10);
hll_empty
-----
\x118a7f
(1 row)
```

hll_empty(int32 log2m, int32 regwidth)

Description: Creates an empty HLL and sets the **log2m** and **regwidth** parameters in sequence. The value of **regwidth** ranges from 1 to 5.

Return type: hll

Example:

```
SELECT hll_empty(10, 4);
hll_empty
-----
\x116a7f
(1 row)
```

hll_empty(int32 log2m, int32 regwidth, int64 expthresh)

Description: Creates an empty HLL and sets the **log2m**, **regwidth**, and **expthresh** parameters. The value of **expthresh** is an integer ranging from -1 to 7. This parameter specifies the threshold for switching from the **explicit** mode to the **sparse** mode. -1 indicates the auto mode; 0 indicates that the **explicit** mode is skipped; a value from 1 to 7 indicates that the mode is switched when the number of distinct values reaches $2^{\text{expthresh}}$.

Return type: hll

Example:

```
SELECT hll_empty(10, 4, 7);
hll_empty
-----
\x116a48
(1 row)
```

hll_empty(int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)

Description: Creates an empty HLL and sets the **log2m**, **regwidth**, **expthresh**, and **sparseon** parameters. The value of **sparseon** is 0 or 1.

Return type: hll

Example:

```
SELECT hll_empty(10,4,7,0);
hll_empty
-----
\x116a08
(1 row)
```

hll_add(hll, hll_hashval)

Description: Adds hll_hashval to an HLL.

Return type: hll

Example:

```
SELECT hll_add(hll_empty(), hll_hash_integer(1));
       hll_add
-----
\x128b7f8895a3f5af28cafe
(1 row)
```

hll_add_rev(hll_hashval, hll)

Description: Adds hll_hashval to an HLL. This function works the same as hll_add, except that the positions of parameters are switched.

Return type: hll

Example:

```
SELECT hll_add_rev(hll_hash_integer(1), hll_empty());
       hll_add_rev
-----
\x128b7f8895a3f5af28cafe
(1 row)
```

hll_eq(hll, hll)

Description: Compares two HLLs to check whether they are the same.

Return type: bool

Example:

```
SELECT hll_eq(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_eq
-----
f
(1 row)
```

hll_ne(hll, hll)

Description: Compares two HLLs to check whether they are different.

Return type: bool

Example:

```
SELECT hll_ne(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_ne
-----
t
(1 row)
```

hll_cardinality(hll)

Description: Calculates the number of distinct values of an HLL.

Return type: integer

Example:

```
SELECT hll_cardinality(hll_empty() || hll_hash_integer(1));
       hll_cardinality
-----
1
(1 row)
```

hll_union(hll, hll)

Description: Performs the **UNION** operation on two HLL data structures to obtain one HLL.

Return type: hll

Example:

```
SELECT hll_union(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_union
-----
\x128b7f8895a3f5af28cafeda0ce907e4355b60
(1 row)
```

7.22.6 Built-in Functions

HyperLogLog (HLL) has a series of built-in functions for internal data processing. Generally, you are not advised to use these functions.

Table 7-19 HLL built-in functions

Function	Function
hll_in	Receives hll data in string format.
hll_out	Sends hll data in string format.
hll_recv	Receives hll data in bytea format.
hll_send	Sends hll data in bytea format.
hll_trans_in	Receives hll_trans_type data in string format.
hll_trans_out	Sends hll_trans_type data in string format.
hll_trans_recv	Receives hll_trans_type data in bytea format.
hll_trans_send	Sends hll_trans_type data in bytea format.
hll_tymod_in	Receives tymod data.
hll_tymod_out	Sends tymod data.
hll_hashval_in	Receives hll_hashval data.
hll_hashval_out	Sends hll_hashval data.
hll_add_trans0	Works similar to hll_add , and is used during the initial phase of DNs in distributed aggregation procedures.
hll_union_trans	Works similar to hll_union, and is used on the first phase of DNs in distributed aggregation operations.
hll_union_collect	Works similar to hll_union, and is used on the second phase of CNs in distributed aggregation operations to summarize the results of each DN.

Function	Function
hll_pack	Is used on the third phase of CNs in distributed aggregation operations to convert a user-defined type hll_trans_type to the hll type.
hll	Converts a hll type to another hll type. Input parameters can be specified.
hll_hashval	Converts the bigint type to the hll_hashval type.
hll_hashval_int4	Converts the int4 type to the hll_hashval type.

7.23 Set Returning Functions

7.23.1 Series Generating Functions

generate_series() returns a series-based set based on the specified start value (**start**), end value (**stop**), and step (**step**).

If **step** is a positive number and **start** is greater than **stop**, zero row is returned. If **step** is a negative number and **start** is less than **stop**, zero row is returned. If any input is NULL, zero rows are returned. If the value of **step** is **0**, an error is reported.

generate_series(start, stop)

Description: Generates a series of values, from **start** to **stop**, the default step is **1**.

Parameter type: int, bigint, or numeric

Return type: setof int, setof bigint, or setof numeric (same as the argument type)

Example:

```
SELECT * FROM generate_series(2,4);
generate_series
```

```
-----
      2
      3
      4
(3 rows)
```

```
SELECT * FROM generate_series(4,3);
generate_series
```

```
-----
(0 rows)
```

```
SELECT * FROM generate_series(1,NULL);
generate_series
```

```
-----
(0 rows)
```

generate_series(start, stop, step)

Description: Generates a series of values, from **start** to **stop** with a step size of **step**.

Parameter type: int, bigint, or numeric

Return type: setof int, setof bigint, or setof numeric (same as the argument type)

Example:

```
SELECT * FROM generate_series(5,1,-2);
generate_series
-----
         5
         3
         1
(3 rows)

SELECT * FROM generate_series(4,6,-5);
generate_series
-----
(0 rows)

SELECT * FROM generate_series(4,3,0);
ERROR:  step size cannot equal zero
```

generate_series(start, stop, step interval)

Description: Generates a series of values, from **start** to **stop** with a step size of **step**.

Parameter type: timestamp or timestamp with time zone

Return type: setof timestamp or setof timestamp with time zone (same as argument type)

Example:

```
-- this example relies on the date-plus-integer operator
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
-----
2017-06-02
2017-06-09
2017-06-16
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp, '2008-03-04 12:00', '10 hours');
generate_series
-----
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
2008-03-04 08:00:00
(9 rows)
```

7.23.2 Subscript Generating Functions

generate_subscripts(array anyarray, dim int)

Description: Generates a series comprising the given array's subscripts.

Return type: setof int

Example:

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
s
---
1
2
3
4
(4 rows)
```

generate_subscripts(array anyarray, dim int, reverse boolean)

Description: Generates a series comprising the given array's subscripts. When **reverse** is true, the series is returned in reverse order.

Return type: setof int

Example:

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1,TRUE) AS s;
s
---
4
3
2
1
(4 rows)
```

generate_subscripts is a function that generates the set of valid subscripts for the specified dimension of a given array. Zero rows are returned for arrays that do not have the requested dimension, or for NULL arrays (but valid subscripts are returned for NULL array elements). Example:

```
-- unnest a 2D array
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
SELECT $1[i][j]
  FROM generate_subscripts($1,1) g1(i),
       generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;

SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
unnest2
-----
1
2
3
4
(4 rows)
```

7.24 Geometric Functions and Operators

7.24.1 Geometric Operators

+

Description: The operator implements a panning function by adding the coordinates of the second point to each corresponding point of the first parameter.

Example:

```
SELECT box '((0,0),(1,1))' + point '(2.0,0)' AS RESULT;  
result  
-----  
(3,1),(2,0)  
(1 row)
```

-

Description: The operator implements a panning function by subtracting the coordinates of the second point from each corresponding point of the first parameter.

Example:

```
SELECT box '((0,0),(1,1))' - point '(2.0,0)' AS RESULT;  
result  
-----  
(-1,1),(-2,0)  
(1 row)
```

*

Description: Scaling out/rotation

Example:

```
SELECT box '((0,0),(1,1))' * point '(2.0,0)' AS RESULT;  
result  
-----  
(2,2),(0,0)  
(1 row)
```

/

Description: Scaling in/rotation

Example:

```
SELECT box '((0,0),(2,2))' / point '(2.0,0)' AS RESULT;  
result  
-----  
(1,1),(0,0)  
(1 row)
```

#

Description: Point or box of intersection

Example:

```
SELECT box'((1,-1),(-1,1))' # box'((1,1),(-1,-1))' AS RESULT;  
result  
-----  
(1,1),(-1,-1)  
(1 row)
```

#

Description: Number of paths or polygon vertices

Example:

```
SELECT # path'((1,0),(0,1),(-1,0))' AS RESULT;  
result
```

```
-----  
3  
(1 row)
```

@-@

Description: Length or circumference

Example:

```
SELECT @-@ path '((0,0),(1,0))' AS RESULT;  
result  
-----  
2  
(1 row)
```

@@

Description: Center of box

Example:

```
SELECT @@ circle '((0,0),10)' AS RESULT;  
result  
-----  
(0,0)  
(1 row)
```

##

Description: Closest point to first figure on second figure.

Example:

```
SELECT point '(0,0)' ## box '((2,0),(0,2))' AS RESULT;  
result  
-----  
(0,0)  
(1 row)
```

<->

Description: Distance between the two figures.

Example:

```
SELECT circle '((0,0),1)' <-> circle '((5,0),1)' AS RESULT;  
result  
-----  
3  
(1 row)
```

&&

Description: Overlaps? (One point in common makes this true.)

Example:

```
SELECT box '((0,0),(1,1))' && box '((0,0),(2,2))' AS RESULT;  
result  
-----  
t  
(1 row)
```

<<

Description: Is strictly left of (no common horizontal coordinate)?

Example:

```
SELECT circle '((0,0),1)' << circle '((5,0),1)' AS RESULT;
result
-----
t
(1 row)
```

>>

Description: Is strictly right of (no common horizontal coordinate)?

Example:

```
SELECT circle '((5,0),1)' >> circle '((0,0),1)' AS RESULT;
result
-----
t
(1 row)
```

&<

Description: Does not extend to the right of?

Example:

```
SELECT box '((0,0),(1,1))' &< box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```

&>

Description: Does not extend to the left of?

Example:

```
SELECT box '((0,0),(3,3))' &> box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```

<<|

Description: Is strictly below (no common horizontal coordinate)?

Example:

```
SELECT box '((0,0),(3,3))' <<| box '((3,4),(5,5))' AS RESULT;
result
-----
t
(1 row)
```

|>>

Description: Is strictly above (no common horizontal coordinate)?

Example:

```
SELECT box '((3,4),(5,5))' |>> box '((0,0),(3,3))' AS RESULT;  
result  
-----  
t  
(1 row)
```

&<|

Description: Does not extend above?

Example:

```
SELECT box '((0,0),(1,1))' &<| box '((0,0),(2,2))' AS RESULT;  
result  
-----  
t  
(1 row)
```

|&>

Description: Does not extend below?

Example:

```
SELECT box '((0,0),(3,3))' |&> box '((0,0),(2,2))' AS RESULT;  
result  
-----  
t  
(1 row)
```

<^

Description: Is below (allows touching)?

Example:

```
SELECT box '((0,0),(-3,-3))' <^ box '((0,0),(2,2))' AS RESULT;  
result  
-----  
t  
(1 row)
```

>^

Description: Is above (allows touching)?

Example:

```
SELECT box '((0,0),(2,2))' >^ box '((0,0),(-3,-3))' AS RESULT;  
result  
-----  
t  
(1 row)
```

?#

Description: Intersect?

Example:

```
SELECT lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' AS RESULT;  
result
```

```
-----  
t  
(1 row)
```

?-

Description: Is horizontal?

Example:

```
SELECT ?- lseg '((-1,0),(1,0))' AS RESULT;  
result  
-----  
t  
(1 row)
```

?-

Description: Are horizontally aligned?

Example:

```
SELECT point '(1,0)' ?- point '(0,0)' AS RESULT;  
result  
-----  
t  
(1 row)
```

?|

Description: Is vertical?

Example:

```
SELECT ?| lseg '((-1,0),(1,0))' AS RESULT;  
result  
-----  
f  
(1 row)
```

?|

Description: Are vertically aligned?

Example:

```
SELECT point '(0,1)' ?| point '(0,0)' AS RESULT;  
result  
-----  
t  
(1 row)
```

?-|

Description: Are perpendicular?

Example:

```
SELECT lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))' AS RESULT;  
result  
-----  
t  
(1 row)
```


?||

Description: Are parallel?

Example:

```
SELECT lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))' AS RESULT;
result
-----
t
(1 row)
```

@>

Description: Contains?

Example:

```
SELECT circle '((0,0),2)' @> point '(1,1)' AS RESULT;
result
-----
t
(1 row)
```

<@

Description: Contained in or on?

Example:

```
SELECT point '(1,1)' <@ circle '((0,0),2)' AS RESULT;
result
-----
t
(1 row)
```

~=

Description: Same as?

Example:

```
SELECT polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' AS RESULT;
result
-----
t
(1 row)
```

7.24.2 Geometric Functions

area(object)

Description: Area calculation

Return type: double precision

Example:

```
SELECT area(box '((0,0),(1,1))') AS RESULT;
result
-----
1
(1 row)
```

center(object)

Description: Figure center calculation

Return type: point

Example:

```
SELECT center(box '((0,0),(1,2))') AS RESULT;  
result  
-----  
(0.5,1)  
(1 row)
```

diameter(circle)

Description: Circle diameter calculation

Return type: double precision

Example:

```
SELECT diameter(circle '((0,0),2.0)') AS RESULT;  
result  
-----  
4  
(1 row)
```

height(box)

Description: Vertical size of box

Return type: double precision

Example:

```
SELECT height(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
1  
(1 row)
```

isclosed(path)

Description: A closed path?

Return type: boolean

Example:

```
SELECT isclosed(path '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----  
t  
(1 row)
```

isopen(path)

Description: An open path?

Return type: boolean

Example:

```
SELECT isopen(path '[(0,0),(1,1),(2,0)]') AS RESULT;  
result  
-----  
t  
(1 row)
```

length(object)

Description: Length calculation

Return type: double precision

Example:

```
SELECT length(path '((-1,0),(1,0))') AS RESULT;  
result  
-----  
4  
(1 row)
```

npoints(path)

Description: Number of points in path

Return type: int

Example:

```
SELECT npoints(path '[(0,0),(1,1),(2,0)]') AS RESULT;  
result  
-----  
3  
(1 row)
```

npoints(polygon)

Description: Number of points in polygon

Return type: int

Example:

```
SELECT npoints(polygon '((1,1),(0,0))') AS RESULT;  
result  
-----  
2  
(1 row)
```

pclose(path)

Description: Converts path to closed.

Return type: path

Example:

```
SELECT pclose(path '[(0,0),(1,1),(2,0)]') AS RESULT;  
result  
-----  
((0,0),(1,1),(2,0))  
(1 row)
```

popen(path)

Description: Converts path to open.

Return type: path

Example:

```
SELECT popen(path '((0,0),(1,1),(2,0)))' AS RESULT;  
result  
-----  
[(0,0),(1,1),(2,0)]  
(1 row)
```

radius(circle)

Description: Circle diameter calculation

Return type: double precision

Example:

```
SELECT radius(circle '((0,0),2.0)') AS RESULT;  
result  
-----  
2  
(1 row)
```

width(box)

Description: Horizontal size of box

Return type: double precision

Example:

```
SELECT width(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
1  
(1 row)
```

7.24.3 Geometric Type Conversion Functions

box(circle)

Description: Circle to box

Return type: box

Example:

```
SELECT box(circle '((0,0),2.0)') AS RESULT;  
result  
-----  
(1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)  
(1 row)
```

box(point, point)

Description: Points to box

Return type: box

Example:

```
SELECT box(point '(0,0)', point '(1,1)') AS RESULT;  
result  
-----  
(1,1),(0,0)  
(1 row)
```

box(polygon)

Description: Polygon to box

Return type: box

Example:

```
SELECT box(polygon '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----  
(2,1),(0,0)  
(1 row)
```

circle(box)

Description: Box to circle

Return type: circle

Example:

```
SELECT circle(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
<(0.5,0.5),0.707106781186548>  
(1 row)
```

circle(point, double precision)

Description: Center and radius to circle

Return type: circle

Example:

```
SELECT circle(point '(0,0)', 2.0) AS RESULT;  
result  
-----  
<(0,0),2>  
(1 row)
```

circle(polygon)

Description: Polygon to circle

Return type: circle

Example:

```
SELECT circle(polygon '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----  
<(1,0.3333333333333333),0.924950591148529>  
(1 row)
```

lseg(box)

Description: Box diagonal to line segment

Return type: lseg

Example:

```
SELECT lseg(box '((-1,0),(1,0))') AS RESULT;  
result  
-----  
[(1,0),(-1,0)]  
(1 row)
```

lseg(point, point)

Description: Points to line segment

Return type: lseg

Example:

```
SELECT lseg(point '(-1,0)', point '(1,0)') AS RESULT;  
result  
-----  
[(-1,0),(1,0)]  
(1 row)
```

path(polygon)

Description: Polygon to path

Return type: path

Example:

```
SELECT path(polygon '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----  
((0,0),(1,1),(2,0))  
(1 row)
```

point(double precision, double precision)

Description: Points

Return type: point

Example:

```
SELECT point(23.4, -44.5) AS RESULT;  
result  
-----  
(23.4,-44.5)  
(1 row)
```

point(box)

Description: Center of box

Return type: point

Example:

```
SELECT point(box '((-1,0),(1,0))') AS RESULT;  
result  
-----  
(0,0)  
(1 row)
```

point(circle)

Description: Center of circle

Return type: point

Example:

```
SELECT point(circle '((0,0),2.0)') AS RESULT;  
result  
-----  
(0,0)  
(1 row)
```

point(lseg)

Description: Center of line segment

Return type: point

Example:

```
SELECT point(lseg '((-1,0),(1,0))') AS RESULT;  
result  
-----  
(0,0)  
(1 row)
```

point(polygon)

Description: Center of polygon

Return type: point

Example:

```
SELECT point(polygon '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----  
(1,0.3333333333333333)  
(1 row)
```

polygon(box)

Description: Box to 4-point polygon

Return type: polygon

Example:

```
SELECT polygon(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
((0,0),(0,1),(1,1),(1,0))  
(1 row)
```

polygon(circle)

Description: Circle to 12-point polygon

Return type: polygon

Example:

```
SELECT polygon(circle '((0,0),2.0)') AS RESULT;
```

```
result
```

```
((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),(1,1.73205080756888),
(1.73205080756888,1),(2,2.44929359829471e-16),(1.73205080756888,-0.999999999999999),
(1,-1.73205080756888),(3.67394039744206e-16,-2),(-0.999999999999999,-1.73205080756888),
(-1.73205080756888,-1))
(1 row)
```

polygon(npts, circle)

Description: Circle to **npts**-point polygon

Return type: polygon

Example:

```
SELECT polygon(12, circle '((0,0),2.0)') AS RESULT;
```

```
result
```

```
((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),(1,1.73205080756888),
(1.73205080756888,1),(2,2.44929359829471e-16),(1.73205080756888,-0.999999999999999),
(1,-1.73205080756888),(3.67394039744206e-16,-2),(-0.999999999999999,-1.73205080756888),
(-1.73205080756888,-1))
(1 row)
```

polygon(path)

Description: Path to polygon

Return type: polygon

Example:

```
SELECT polygon(path '((0,0),(1,1),(2,0))') AS RESULT;
```

```
result
```

```
((0,0),(1,1),(2,0))
(1 row)
```

7.25 Network Address Functions and Operators

7.25.1 cidr and inet Operators

The operators <, <=, >, and >= test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any host part) and determine whether one network is identical to or a subnet of the other.

<

Description: Is less than

Example:

```
SELECT inet '192.168.1.5' < inet '192.168.1.6' AS RESULT;  
result  
-----  
t  
(1 row)
```

<=

Description: Is less than or equals

Example:

```
SELECT inet '192.168.1.5' <= inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```

=

Description: Equals

Example:

```
SELECT inet '192.168.1.5' = inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```

>=

Description: Is greater than or equals

Example:

```
SELECT inet '192.168.1.5' >= inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```

>

Description: Is greater than

Example:

```
SELECT inet '192.168.1.5' > inet '192.168.1.4' AS RESULT;  
result
```

```
-----  
t  
(1 row)
```

<>

Description: Does not equal to

Example:

```
SELECT inet '192.168.1.5' <> inet '192.168.1.4' AS RESULT;  
result  
-----  
t  
(1 row)
```

<<

Description: Is contained in

Example:

```
SELECT inet '192.168.1.5' << inet '192.168.1/24' AS RESULT;  
result  
-----  
t  
(1 row)
```

<<=

Description: Is contained in or equals

Example:

```
SELECT inet '192.168.1/24' <<= inet '192.168.1/24' AS RESULT;  
result  
-----  
t  
(1 row)
```

>>

Description: Contains

Example:

```
SELECT inet '192.168.1/24' >> inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```

>>=

Description: Contains or equals

Example:

```
SELECT inet '192.168.1/24' >>= inet '192.168.1/24' AS RESULT;  
result  
-----  
t  
(1 row)
```

~

Description: Bitwise NOT

Example:

```
SELECT ~ inet '192.168.1.6' AS RESULT;  
result  
-----  
63.87.254.249  
(1 row)
```

&

Description: The AND operation is performed on each bit of the two network addresses.

Example:

```
SELECT inet '192.168.1.6' & inet '10.0.0.0' AS RESULT;  
result  
-----  
0.0.0.0  
(1 row)
```

|

Description: The OR operation is performed on each bit of the two network addresses.

Example:

```
SELECT inet '192.168.1.6' | inet '10.0.0.0' AS RESULT;  
result  
-----  
202.168.1.6  
(1 row)
```

+

Description: Addition

Example:

```
SELECT inet '192.168.1.6' + 25 AS RESULT;  
result  
-----  
192.168.1.31  
(1 row)
```

-

Description: Subtraction

Example:

```
SELECT inet '192.168.1.43' - 36 AS RESULT;  
result  
-----  
192.168.1.7  
(1 row)  
SELECT inet '192.168.1.43' - inet '192.168.1.19' AS RESULT;  
result
```

```
-----  
24  
(1 row)
```

7.25.2 Network Address Functions

The **abbrev**, **host**, and **text** functions are primarily intended to offer alternative display formats.

Any **cidr** value can be cast to **inet** implicitly or explicitly; therefore, the functions shown above as operating on **inet** also work on **cidr** values. An **inet** value can be cast to **cidr**. After the conversion, any bits to the right of the subnet mask are silently zeroed to create a valid **cidr** value. In addition, you can cast a text string to **inet** or **cidr** using normal casting syntax. For example, **inet(expression)** or **colname::cidr**.

abbrev(inet)

Description: Abbreviated display format as text

Return type: text

Example:

```
SELECT abbrev(inet '10.1.0.0/16') AS RESULT;  
result  
-----  
10.1.0.0/16  
(1 row)
```

abbrev(cidr)

Description: Abbreviated display format as text

Return type: text

Example:

```
SELECT abbrev(cidr '10.1.0.0/16') AS RESULT;  
result  
-----  
10.1/16  
(1 row)
```

broadcast(inet)

Description: Broadcast address for network

Return type: inet

Example:

```
SELECT broadcast('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.255/24  
(1 row)
```

family(inet)

Description: Extracts family of address; **4** for IPv4, **6** for IPv6

Return type: int

Example:

```
SELECT family('::1') AS RESULT;  
result  
-----  
6  
(1 row)
```

host(inet)

Description: Extracts IP address as text.

Return type: text

Example:

```
SELECT host('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.5  
(1 row)
```

hostmask(inet)

Description: Constructs host mask for network.

Return type: inet

Example:

```
SELECT hostmask('192.168.23.20/30') AS RESULT;  
result  
-----  
0.0.0.3  
(1 row)
```

masklen(inet)

Description: Extracts subnet mask length.

Return type: int

Example:

```
SELECT masklen('192.168.1.5/24') AS RESULT;  
result  
-----  
24  
(1 row)
```

netmask(inet)

Description: Constructs a subnet mask for the network.

Return type: inet

Example:

```
SELECT netmask('192.168.1.5/24') AS RESULT;  
result  
-----
```

```
255.255.255.0  
(1 row)
```

network(inet)

Description: Extracts network part of address.

Return type: cidr

Example:

```
SELECT network('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.0/24  
(1 row)
```

set_masklen(inet, int)

Description: Sets subnet mask length for **inet** value.

Return type: inet

Example:

```
SELECT set_masklen('192.168.1.5/24', 16) AS RESULT;  
result  
-----  
192.168.1.5/16  
(1 row)
```

set_masklen(cidr, int)

Description: Sets subnet mask length for **cidr** value.

Return type: cidr

Example:

```
SELECT set_masklen('192.168.1.0/24::cidr', 16) AS RESULT;  
result  
-----  
192.168.0.0/16  
(1 row)
```

text(inet)

Description: Extracts IP address and subnet mask length as text.

Return type: text

Example:

```
SELECT text(inet '192.168.1.5') AS RESULT;  
result  
-----  
192.168.1.5/32  
(1 row)
```

trunc(macaddr)

The function **trunc(macaddr)** returns a MAC address with the last 3 bytes set to zero.

Description: Sets last 3 bytes to zero.

Return type: `macaddr`

Example:

```
SELECT trunc(macaddr '12:34:56:78:90:ab') AS RESULT;  
result  
-----  
12:34:56:00:00:00  
(1 row)
```

The **`macaddr`** type also supports the standard relational operators (such as `>` and `<=`) for lexicographical ordering, and the bitwise arithmetic operators (`~`, `&` and `|`) for NOT, AND and OR.

7.26 System Information Functions

7.26.1 Session Information Functions

`current_catalog`

Description: name of the current database (called **`catalog`** in the SQL standard), same as **`current_database()`**.

Return type: name

Example:

```
SELECT current_catalog;  
current_database  
-----  
gaussdb  
(1 row)
```

`current_database()`

Description: Name of the current database

Return type: name

Example:

```
SELECT current_database();  
current_database  
-----  
gaussdb  
(1 row)
```

`current_query()`

Description: Text of the currently executing query, as submitted by the client (might contain more than one statement)

Return type: text

Example:

```
SELECT current_query();  
current_query
```

```
-----  
SELECT current_query();  
(1 row)
```

current_schema[()]

Description: **current_schema** returns the first valid schema name in the search path. (If the search path is empty or contains no valid schema name, **NULL** is returned.) This is the schema that will be used for any tables or other named objects that are created without specifying a target schema.

Return type: name

Example:

```
SELECT current_schema();  
current_schema  
-----  
public  
(1 row)
```

current_schemas(boolean)

Description: **current_schemas(boolean)** returns an array of the names of all schemas presently in the search path. The Boolean option determines whether implicitly included system schemas such as **pg_catalog** are included in the returned search path.

Return type: name[]

Example:

```
SELECT current_schemas(true);  
current_schemas  
-----  
{pg_catalog,public}  
(1 row)
```

NOTE

The search path can be altered at run time. The command is:

```
SET search_path TO schema [, schema, ...]
```

current_user

Description: Username of current execution context. **current_user** is the identifier of the user whose permission needs to be checked. It is usually used to represent a session user, but this setting can be changed according to [SET ROLE](#). It also changes during the execution of functions with the attribute **SECURITY DEFINER**.

Return type: name

Example:

```
SELECT current_user;  
current_user  
-----  
dbadmin  
(1 row)
```


inet_client_addr()

Description: Displays the IP address of the currently connected client.

NOTE

- It is available only in remote connection mode.
- If the database is connected to the local PC, the value is empty.

Return type: inet

Example:

```
SELECT inet_client_addr();
inet_client_addr
-----
10.10.0.50
(1 row)
```

inet_client_port()

Description: Displays the port number of the currently connected client.

NOTE

It is available only in remote connection mode.

Return type: integer

Example:

```
SELECT inet_client_port();
inet_client_port
-----
33143
(1 row)
```

inet_server_addr()

Description: Displays the IP address of the current server.

NOTE

- It is available only in remote connection mode.
- If the database is connected to the local PC, the value is empty.

Return type: inet

Example:

```
SELECT inet_server_addr();
inet_server_addr
-----
10.10.0.13
(1 row)
```

inet_server_port()

Description: Displays the port of the current server. All these functions return NULL if the current connection is via a Unix-domain socket.

 NOTE

It is available only in remote connection mode.

Return type: integer

Example:

```
SELECT inet_server_port();
inet_server_port
-----
8000
(1 row)
```

pg_backend_pid()

Description: Process ID of the server process attached to the current session

Return type: integer

Example:

```
SELECT pg_backend_pid();
pg_backend_pid
-----
140229352617744
(1 row)
```

pg_conf_load_time()

Description: Configures load time. **pg_conf_load_time** returns the timestamp with time zone when the server configuration files were last loaded.

Return type: timestamp with time zone

Example:

```
SELECT pg_conf_load_time();
pg_conf_load_time
-----
2017-09-01 16:05:23.89868+08
(1 row)
```

pg_my_temp_schema()

Description: **pg_my_temp_schema** returns the OID of the current session's temporary schema, or zero if it has none (because it has not created any temporary tables). **pg_is_other_temp_schema** returns true if the given OID is the OID of another session's temporary schema.

Return type: OID

Example:

```
SELECT pg_my_temp_schema();
pg_my_temp_schema
-----
0
(1 row)
```

pg_is_other_temp_schema(oid)

Description: Whether the schema is the temporary schema of another session.

Return type: boolean

Example:

```
SELECT pg_is_other_temp_schema(25356);
       pg_is_other_temp_schema
-----
f
(1 row)
```

pg_postmaster_start_time()

Description: Specifies the start time of the database instance.

pg_postmaster_start_time returns the timestamp (with time zone) when the database instance was started.

Return type: timestamp with time zone

Example:

```
SELECT pg_postmaster_start_time();
       pg_postmaster_start_time
-----
2017-08-30 16:02:54.99854+08
(1 row)
```

pg_trigger_depth()

Description: Current nesting level of triggers

Return type: integer

Example:

```
SELECT pg_trigger_depth();
       pg_trigger_depth
-----
0
(1 row)
```

pgxc_version()

Description: Postgres-XC version information

Return type: text

Example:

```
SELECT pgxc_version();
       pgxc_version
-----
Postgres-XC 1.1 on x86_64-unknown-linux-gnu, based on PostgreSQL 9.2.4, compiled by g++ (GCC) 5.4.0, 64-bit
(1 row)
```

session_user

Description: Session user name **session_user** is usually the user who initiated the current database connection, but administrators can change this setting with [SET SESSION AUTHORIZATION](#).

Return type: name

Example:

```
SELECT session_user;  
session_user  
-----  
dbadmin  
(1 row)
```

user

Description: Is equivalent to **current_user**.

Return type: name

Example:

```
SELECT user;  
current_user  
-----  
dbadmin  
(1 row)
```

version()

Description: version information. **version** returns a string describing a server's version.

Return type: text

Example:

```
SELECT version();  
version  
-----  
PostgreSQL 9.2.4 gsql ((GaussDB 8.2.1 build 39137c2d) compiled at 2022-09-23 15:43:11 commit 3629 last  
mr 5138 release) on x86_64-unknown-linux-gnu, compiled by g++ (GCC) 5.4.0, 64-bit  
(1 row)
```

7.26.2 Access Privilege Inquiry Functions

has_any_column_privilege(user, table, privilege)

Description: Queries whether a specified user has permission for any column of table.

Return type: boolean

has_any_column_privilege(table, privilege)

Description: Queries whether the current user has permission for any column of table.

Return type: boolean

has_any_column_privilege checks whether a user can access any column of a table in a particular way. Its parameter possibilities are analogous to **has_table_privilege**, except that the desired access permission type must be some combination of SELECT, INSERT, UPDATE, or REFERENCES.

 NOTE

Note that having any of these permissions at the table level implicitly grants it for each column of the table, so **has_any_column_privilege** will always return **true** if **has_table_privilege** does for the same parameters. But **has_any_column_privilege** also succeeds if there is a column-level grant of the permission for at least one column.

has_column_privilege(user, table, column, privilege)

Description: Queries whether a specified user has permission for column.

Return type: boolean

has_column_privilege(table, column, privilege)

Description: Queries whether the current user has permission for column.

Return type: boolean

has_column_privilege checks whether a user can access a column in a particular way. Its argument possibilities are analogous to **has_table_privilege**, with the addition that the column can be specified either by name or attribute number. The desired access permission type must evaluate to some combination of **SELECT**, **INSERT**, **UPDATE**, or **REFERENCES**.

 NOTE

Note that having any of these permissions at the table level implicitly grants it for each column of the table.

has_database_privilege(user, database, privilege)

Description: Queries whether a specified user has permission for database.

Return type: boolean

has_database_privilege(database, privilege)

Description: Queries whether the current user has permission for database.

Return type: boolean

Note: **has_database_privilege** checks whether a user can access a database in a particular way. Its argument possibilities are analogous to **has_table_privilege**. The desired access permission type must evaluate to some combination of **CREATE**, **CONNECT**, **TEMPORARY**, or **TEMP** (which is equivalent to **TEMPORARY**).

has_foreign_data_wrapper_privilege(user, fdw, privilege)

Description: Queries whether a specified user has permission for foreign-data wrapper.

The **fdw** parameter indicates the name or ID of the foreign data wrapper.

Return type: boolean

has_foreign_data_wrapper_privilege(fdw, privilege)

Description: Queries whether the current user has permission for foreign-data wrapper.

Return type: boolean

Note: **has_foreign_data_wrapper_privilege** checks whether a user can access a foreign-data wrapper in a particular way. Its argument possibilities are analogous to **has_table_privilege**. The desired access permission type must evaluate to **USAGE**.

has_function_privilege(user, function, privilege)

Description: Queries whether a specified user has permission for function.

Return type: boolean

has_function_privilege(function, privilege)

Description: Queries whether the current user has permission for function.

Return type: boolean

Note: **has_function_privilege** checks whether a user can access a function in a particular way. Its argument possibilities are analogous to **has_table_privilege**. When a function is specified by a text string rather than by OID, the allowed input is the same as that for the **regprocedure** data type (see [Object Identifier Types](#)). The desired access permission type must evaluate to **EXECUTE**.

has_language_privilege(user, language, privilege)

Description: Queries whether a specified user has permission for language.

Return type: boolean

has_language_privilege(language, privilege)

Description: Queries whether the current user has permission for language.

Return type: boolean

Note: **has_language_privilege** checks whether a user can access a procedural language in a particular way. Its argument possibilities are analogous to **has_table_privilege**. The desired access permission type must evaluate to **USAGE**.

has_schema_privilege(user, schema, privilege)

Description: Queries whether a specified user has permission for schema.

Return type: boolean

has_schema_privilege(schema, privilege)

Description: Queries whether the current user has permission for schema.

Return type: boolean

Note: **has_schema_privilege** checks whether a user can access a schema in a particular way. Its argument possibilities are analogous to **has_table_privilege**. The desired access permission type must evaluate to some combination of **CREATE** or **USAGE**.

has_server_privilege(user, server, privilege)

Description: Queries whether a specified user has permission for foreign server.

Return type: boolean

has_server_privilege(server, privilege)

Description: Queries whether the current user has permission for foreign server.

Return type: boolean

Note: **has_server_privilege** checks whether a user can access a foreign server in a particular way. Its argument possibilities are analogous to **has_table_privilege**. The desired access permission type must evaluate to **USAGE**.

has_table_privilege(user, table, privilege)

Description: Queries whether a specified user has permission for table.

Return type: boolean

has_table_privilege(table, privilege)

Description: Queries whether the current user has permission for table.

Return type: boolean

has_table_privilege checks whether a user can access a table in a particular way. The user can be specified by name, by OID (**pg_authid.oid**), **public** to indicate the PUBLIC pseudo-role, or if the argument is omitted **current_user** is assumed. The table can be specified by name or by OID. When specifying by name, the name can be schema-qualified if necessary. The desired access permission type is specified by a text string, which must be one of the values **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**, **REFERENCES**, or **TRIGGER**. Optionally, **WITH GRANT OPTION** can be added to a permission type to test whether the permission is held with grant option. Also, multiple permission types can be listed separated by commas, in which case the result will be **true** if any of the listed permissions is held.

Example:

```
SELECT has_table_privilege('tpcds.web_site', 'select');
has_table_privilege
-----
t
(1 row)

SELECT has_table_privilege('dbadmin', 'tpcds.web_site', 'select,INSERT WITH GRANT OPTION ');
has_table_privilege
-----
t
(1 row)
```

pg_has_role(user, role, privilege)

Description: Queries whether a specified user has permission for role.

Return type: boolean

pg_has_role(role, privilege)

Description: Specifies whether the current user has permission for role.

Return type: boolean

Note: **pg_has_role** checks whether a user can access a role in a particular way. Its argument possibilities are analogous to **has_table_privilege**, except that **public** is not allowed as a user name. The desired access permission type must evaluate to some combination of **MEMBER** or **USAGE**. **MEMBER** denotes direct or indirect membership in the role (that is, the right to do **SET ROLE**), while **USAGE** denotes the permissions of the role are available without doing **SET ROLE**.

7.26.3 Schema Visibility Inquiry Functions

Schema visibility inquiry functions perform visibility checks on database objects. For functions and operators, an object in the search path is visible if there is no object of the same name and argument data type(s) earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require OIDs to identify the objects to be checked. If you want to test an object by name, it is convenient to use the OID alias types (**regclass**, **regtype**, **regprocedure**, **regoperator**, **regconfig**, or **regdictionary**).

For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. For example, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

pg_collation_is_visible(collation_oid)

Description: Queries whether the collation is visible in search path.

Return type: boolean

pg_conversion_is_visible(conversion_oid)

Description: Queries whether the conversion is visible in search path.

Return type: boolean

pg_function_is_visible(function_oid)

Description: Queries whether the function is visible in search path.

Return type: boolean

pg_opclass_is_visible(opclass_oid)

Description: Queries whether the operator class is visible in search path.

Return type: boolean

pg_operator_is_visible(operator_oid)

Description: Queries whether the operator is visible in search path.

Return type: boolean

pg_opfamily_is_visible(opclass_oid)

Description: Queries whether the operator family is visible in search path.

Return type: boolean

pg_table_is_visible(table_oid)

Description: Queries whether the table is visible in search path.

Return type: boolean

pg_ts_config_is_visible(config_oid)

Description: Queries whether the text search configuration is visible in search path.

Return type: boolean

pg_ts_dict_is_visible(dict_oid)

Description: Queries whether the text search dictionary is visible in search path.

Return type: boolean

pg_ts_parser_is_visible(parser_oid)

Description: Queries whether the text search parser is visible in search path.

Return type: boolean

pg_ts_template_is_visible(template_oid)

Description: Queries whether the text search template is visible in search path.

Return type: boolean

pg_type_is_visible(type_oid)

Description: Queries whether the type (or domain) is visible in search path.

Return type: boolean

7.26.4 System Catalog Information Functions

format_type(type_oid, typemod)

Description: Gets SQL name of a data type.

Return type: text

Note:

format_type returns the SQL name of a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known. Certain type modifiers are passed for data types with length limitations. The SQL name returned from **format_type** contains the length of the data type, which can be calculated by taking `sizeof(int32)` from actual storage length [actual storage len - `sizeof(int32)`] in the unit of bytes. 32-bit space is required to store the customized length set by users. So the actual storage length contains 4 bytes more than the customized length. In the following example, the SQL name returned from **format_type** is `character varying(6)`, indicating the length of `varchar` type is 6 bytes. So the actual storage length of `varchar` type is 10 bytes.

```
SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'), 10);
format_type
-----
character varying(6)
(1 row)
```

pg_check_authid(role_oid)

Description: Check whether a role name with given OID exists.

Return type: Boolean

pg_describe_object(catalog_id, object_id, object_sub_id)

Description: Gets description of a database object.

Return type: text

Note: **pg_describe_object** returns a description of a database object specified by catalog OID, object OID and a (possibly zero) sub-object ID. This is useful to determine the identity of an object as stored in the **pg_depend** catalog.

pg_get_constraintdef(constraint_oid)

Description: Gets definition of a constraint.

Return type: text

pg_get_constraintdef(constraint_oid, pretty_bool)

Description: Gets definition of a constraint.

Return type: text

Note: **pg_get_constraintdef** and **pg_get_indexdef** respectively reconstruct the creating command for a constraint and an index.

pg_get_expr(pg_node_tree, relation_oid)

Description: Decompiles internal form of an expression, assuming that any Vars in it refer to the relationship indicated by the second parameter.

Return type: text

pg_get_expr(pg_node_tree, relation_oid, pretty_bool)

Description: Decompiles internal form of an expression, assuming that any Vars in it refer to the relationship indicated by the second parameter.

Return type: text

Note: **pg_get_expr** decompiles the internal form of an individual expression, such as the default value for a column. It can be useful when examining the contents of system catalogs. If the expression might contain Vars, specify the OID of the relationship they refer to as the second parameter; if no Vars are expected, zero is sufficient.

pg_get_indexdef(index_oid)

Description: Gets **CREATE INDEX** command for index.

Return type: text

index_oid indicates the index OID, which can be queried in the **PG_STATIO_ALL_INDEXES** system view.

Example: Query the OID and CREATE INDEX command of **index ds_ship_mode_t1_index1**.

```
SELECT indexrelid FROM PG_STATIO_ALL_INDEXES WHERE indexrelname = 'ds_ship_mode_t1_index1';
indexrelid
-----
    136035
(1 row)
SELECT * FROM pg_get_indexdef(136035);
pg_get_indexdef
-----
CREATE INDEX ds_ship_mode_t1_index1 ON tpceds.ship_mode_t1 USING psort (sm_ship_mode_sk)
TABLESPACE pg_default
(1 row)
```

pg_get_indexdef(index_oid, column_no, pretty_bool)

Description: Gets **CREATE INDEX** command for index, or definition of just one index column when **column_no** is not zero.

Return type: text

```
SELECT * FROM pg_get_indexdef(136035,0,false);
pg_get_indexdef
-----
CREATE INDEX ds_ship_mode_t1_index1 ON tpceds.ship_mode_t1 USING psort (sm_ship_mode_sk)
TABLESPACE pg_default
(1 row)
SELECT * FROM pg_get_indexdef(136035,1,false);
pg_get_indexdef
-----
sm_ship_mode_sk
(1 row)
```

pg_get_keywords()

Description: Gets list of SQL keywords and their categories.

Return type: SETOF record

Note: **pg_get_keywords** returns a set of records describing the SQL keywords recognized by the server. The **word** column contains the keyword. The **catcode** column contains a category code: **U** for unreserved, **C** for column name, **T** for type or function name, or **R** for reserved. The **catdesc** column contains a possibly-localized string describing the category.

pg_get_ruledef(rule_oid)

Description: Gets **CREATE RULE** command for a rule.

Return type: text

pg_get_ruledef(rule_oid, pretty_bool)

Description: Gets **CREATE RULE** command for a rule.

Return type: text

pg_get_userbyid(role_oid)

Description: Gets role name with given OID.

Return type: name

Note: **pg_get_userbyid** extracts a role's name given its OID.

pg_get_viewdef(viewname text [, pretty bool [, fullflag bool]])

Description: gets underlying **SELECT** command for views.

Return type: text

Note:

- **pg_get_viewdef** reconstructs the **SELECT** query that defines a view. If the value of **pretty bool** is set to **true**, the display format is suitable for printing and more readable. The default value of **pretty bool** is **false**, and the display format is not readable. Use the default format for dump purposes whenever possible. The **pretty bool** parameter can be applied only to valid views.
- When **fullflag bool** is set to **true**, the complete definition of the view is displayed. The default value is **false**.

pg_get_viewdef(viewoid oid [, pretty bool [, fullflag bool]])

Description: gets underlying **SELECT** command for views.

Return type: text

pg_get_viewdef(view_oid, wrap_column_int)

Description: Gets underlying SELECT command for view, wrapping lines with columns as specified, printing is implied.

Return type: text

pg_get_tabledef(table_oid)

Description: Obtains the table definition based on the table OID.

Return type: text

Example: Obtain the OID of the table **customer_t2** from the system catalog **pg_class**, and then use this function to query the definition of **customer_t2** to obtain the table columns, storage mode (row-store or column-store), and table distribution mode configured for **customer_t2** when it is created.

```
select oid from pg_class where relname = 'customer_t2';
 oid
-----
17353
(1 row)

select * from pg_get_tabledef(17353);
 pg_get_tabledef
-----
SET search_path = dbadmin;          +
CREATE TABLE customer_t2 (        +
  state_id character(2),            +
  state_name character varying(40), +
  area_id numeric                   +
)                                    +
WITH (orientation=column, compression=low)+
DISTRIBUTE BY HASH(state_id)       +
TO GROUP group_version1;
(1 row)
```

pg_get_tabledef(table_name)

Description: Obtains the table definition based on the table name.

Return type: text

Remarks: **pg_get_tabledef** reconstructs the **CREATE** statement of the table definition, including the table definition, index information, and comments. Users need to create the dependent objects of the table, such as groups, schemas, tablespaces, and servers. The table definition does not include the statements for creating these dependent objects.

pg_get_tabledef(table_name/table_oid, forCreate boolean)

Description: Obtains the table definition based on the table name or OID. It is supported only by clusters of version 9.1.0.100 or later.

Return type: text

- If the **forCreate** parameter is **true**, the table definition obtained will exclude the **TO GROUP** clause.
- If the **forCreate** parameter is **false**, the table definition obtained will include the **TO GROUP** clause.

pg_get_tabledef(table_name/table_oid, forCreate boolean,withColComm boolean)

Description: Obtains the table definition based on the table name or OID. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: text

- If **withColComment** is **true**, the comment for a column field is displayed next to it,
- If **withColComment** is **false**, the comment for a column is shown at the end of the table definition.

Example:

```
SELECT pg_get_tabledef('person',false,true);
pg_get_tabledef
-----
SET search_path = public;          +
CREATE TABLE person (             +
    id integer COMMENT 'Student ID', +
    name character varying(25) COMMENT 'Name',+
    address text COMMENT 'Address'      +
)                                     +
WITH (orientation=row, compression=no) +
DISTRIBUTE BY ROUNDROBIN            +
TO GROUP group_version1;
(1 row)

SELECT pg_get_tabledef('person',false,false);
pg_get_tabledef
-----
SET search_path = public;          +
CREATE TABLE person (             +
    id integer,                    +
    name character varying(25),    +
    address text                   +
)                                     +
WITH (orientation=row, compression=no) +
DISTRIBUTE BY ROUNDROBIN            +
TO GROUP group_version1;            +
COMMENT ON COLUMN person.id IS 'Student ID'; +
COMMENT ON COLUMN person.name IS 'Name'; +
COMMENT ON COLUMN person.address IS 'Address';
(1 row)
```

pg_options_to_table(reloptions)

Description: Gets the set of storage option name/value pairs.

Return type: SETOF record

Note: **pg_options_to_table** returns the set of storage option name/value pairs (**option_name/option_value**) when passing **pg_class.reloptions** or **pg_attribute.attoptions**.

Example:

```
CREATE TABLE customer_test
(
    state_ID CHAR(2),
    state_NAME VARCHAR2(40),
    area_ID NUMBER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=middle);
```

```
SELECT pg_options_to_table(reloptions) FROM pg_class WHERE relname='customer_test';
pg_options_to_table
-----
(orientation,column)
(compression,middle)
(bucketnums,16384)
(colversion,2.0)
(enable_delta,false)
(5 rows)
```

pg_typeof(any)

Description: Gets the data type of any value.

Return type: regtype

Note:

pg_typeof returns the OID of the data type of the value that is passed to it. This can be helpful for troubleshooting or dynamically constructing SQL queries. The function is declared as returning **regtype**, which is an OID alias type (see [Object Identifier Types](#)). This means that it is the same as an OID for comparison purposes but displays as a type name.

Example:

```
SELECT pg_typeof(33);
pg_typeof
-----
integer
(1 row)

SELECT typen FROM pg_type WHERE oid = pg_typeof(33);
typen
-----
4
(1 row)
```

collation for (any)

Description: Gets the collation of the parameter.

Return type: text

Note:

The expression **collation for** returns the collation of the value that is passed to it.
Example:

```
SELECT collation for (description) FROM pg_description LIMIT 1;
pg_collation_for
-----
"default"
(1 row)
```

The value might be quoted and schema-qualified. If no collation is derived for the argument expression, then a null value is returned. If the parameter is not of a collatable data type, then an error is thrown.

getdistributekey(table_name)

Description: Gets a distribution column for a hash table.

Return type: text

Example:

```
SELECT getdistributekey('item');
getdistributekey
-----
i_item_sk
(1 row)
```

7.26.5 System Function Checking Function

pv_builtin_functions()

Description: Queries information about system functions.

Return type: record

pg_get_functiondef(func_oid)

Description: Gets definition of a function.

Return type: text

func_oid is the OID of the function, which can be queried in the **PG_PROC** system catalog.

Example: Query the OID and definition of the **justify_days** function.

```
SELECT oid FROM pg_proc WHERE proname = 'justify_days';
oid
-----
1295
(1 row)

SELECT * FROM pg_get_functiondef(1295);
headerlines | definition
-----+-----
4 | CREATE OR REPLACE FUNCTION pg_catalog.justify_days(interval)+
  | RETURNS interval +
  | LANGUAGE internal +
  | IMMUTABLE STRICT NOT FENCED NOT SHIPPABLE +
  | AS $function$interval_justify_days$function$ +
  |
(1 row)
```

Note: The query result returned by **pg_get_functiondef** is in the original text format of the stored procedure. The escape character (\) is used to facilitate subsequent parsing.

pg_get_function_arguments(func_oid)

Description: Gets argument list of function's definition (with default values).

Return type: text

Note: **pg_get_function_arguments** returns the argument list of a function, in the form it would need to appear in within **CREATE FUNCTION**.

pg_get_function_identity_arguments(func_oid)

Description: Gets argument list to identify a function (without default values).

Return type: text

Note: **pg_get_function_identity_arguments** returns the argument list necessary to identify a function, in the form it would need to appear in within **ALTER FUNCTION**. This form omits default values.

pg_get_function_result(func_oid)

Description: Gets **RETURNS** clause for function.

Return type: text

Note: **pg_get_function_result** returns the appropriate **RETURNS** clause for the function.

7.26.6 Comment Checking Functions

col_description(table_oid, column_number)

Description: Gets comment for a table column.

Return type: text

Note: **col_description** returns the comment for a table column, which is specified by the OID of its table and its column number.

Example: Query the **pg_class** system catalog to obtain the table OID, and query the INFORMATION_SCHEMA.COLUMNS system view to obtain **column_number**.

```
SELECT COLUMN_NAME,ORDINAL_POSITION FROM INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME = 't' AND COLUMN_NAME = 'a';
column_name | ordinal_position
-----+-----
a           | 1
(1 row)

SELECT oid FROM pg_class WHERE relname='t';
 oid
-----
44020
(1 row)

SELECT col_description(44020,1);
 col_description
-----
This is a test table.
(1 row)
```

obj_description(object_oid, catalog_name)

Description: Gets comment for a database object.

Return type: text

Note: The two-parameter form of **obj_description** returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, **obj_description(123456,'pg_class')** would retrieve the comment for the table with OID 123456. The one-parameter form of **obj_description** requires only the object OID.

obj_description cannot be used for table columns since columns do not have OIDs of their own.

obj_description(object_oid)

Description: Gets comment for a database object.

Return type: text

shobj_description(object_oid, catalog_name)

Description: Gets comment for a shared database object.

Return type: text

Note: The functions of **shobj_description** and **obj_description** are similar. The only difference is that **shobj_description** is used for shared objects. Some system catalogs are global to all databases within each cluster, and the comments for objects in them are stored globally as well.

7.26.7 Transaction IDs and Snapshots

The following functions provide server transaction information in an exportable form. The main use of these functions is to determine which transactions were committed between two snapshots.

pgxc_is_committed(transaction_id)

Description: Determines whether the given XID is committed or ignored. NULL indicates the unknown status (such as running, preparing, and freezing).

Return type: Boolean

txid_current()

Description: Gets current transaction ID.

Return type: bigint

txid_current_snapshot()

Description: Gets current snapshot.

Return type: txid_snapshot

txid_snapshot_xip(txid_snapshot)

Description: Gets in-progress transaction IDs in snapshot.

Return type: setof bigint

txid_snapshot_xmax(txid_snapshot)

Description: Gets **xmax** of snapshot.

Return type: bigint

txid_snapshot_xmin(txid_snapshot)

Description: Gets **xmin** of snapshot.

Return type: bigint

txid_visible_in_snapshot(bigint, txid_snapshot)

Description: Queries whether the transaction ID is visible in snapshot. (do not use with subtransaction ids)

Return type: boolean

The internal transaction ID type (**xid**) is 32 bits wide and wraps around every 4 billion transactions. **txid_snapshot**, the data type used by these functions, stores information about transaction ID visibility at a particular moment in time. [Table 7-20](#) describes its components.

Table 7-20 Snapshot components

Component	Description
xmin	Earliest transaction ID (txid) that is still active. All earlier transactions will either be committed and visible, or rolled back.
xmax	First as-yet-unassigned txid. All txids greater than or equal to this are not yet started as of the time of the snapshot, so they are invisible.
xip_list	Active txids at the time of the snapshot. The list includes only those active txids between xmin and xmax ; there might be active txids higher than xmax . A txid that is xmin ≤ txid < xmax and not in this list was already completed at the time of the snapshot, and is either visible or dead according to its commit status. The list does not include txids of subtransactions.

txid_snapshot's textual representation is **xmin:xmax:xip_list**.

For example: **10:20:10,14,15** means **xmin=10, xmax=20, xip_list=10, 14, 15**.

7.26.8 Computing Node Group Function

pv_compute_pool_workload()

Description: Load status of a computing Node Group.

Return type: void

Example:

```
SELECT * from pv_compute_pool_workload();
nodename | rpinuse | maxrp | nodestate
-----+-----+-----+-----
datanode1 | 0 | 1000 | normal
```

```
datanode2 | 0 | 1000 | normal  
(2 rows)
```

7.26.9 Lock Information Function

pgxc_get_lock_conflicts()

Description: Obtains information about conflicting locks in the cluster. When a lock is waiting for another lock or another lock is waiting for it, a lock conflict occurs.

Return type: SETOF record

Example:

```
SELECT * from pgxc_get_lock_conflicts();  
pgxc_get_lock_conflicts  
-----  
(relation,cn_5001,gaussdb,u1,test,,,,omm,648406,,,"<insufficient  
privilege>",140651661784832,ExclusiveLock,f,2024-10-29 17:24:13.425672+08,)  
(1 row)
```

7.27 System Management Functions

7.27.1 Configuration Settings Functions

Configuration setting functions are used for querying and modifying configuration parameters during running.

current_setting(setting_name)

Description: Specifies the current setting.

Return type: text

Note: **current_setting** obtains the current setting of **setting_name** by query. It is equivalent to the **SHOW** statement. For example:

```
SELECT current_setting('datestyle');  
current_setting  
-----  
ISO, MDY  
(1 row)
```

read_global_var(global_setting_name)

Description: Reads the current value of a global variable.

Return type: text

Note: **read_global_var** is used query the current value of **global_setting_name**. It is equivalent to the **SHOW** statement.

Example:

```
SET my.var = 1000;  
SELECT read_global_var('my.var');
```

```
read_global_var
-----
1000
(1 row)
```

NOTE

global_setting_name indicates a variable similar to **my.var**. The variable name contains decimal points and the left and right sides of the decimal points are not empty. Variables of this type are called global variables.

Global variables can be set only by running the **SET** command. Methods such as **gs_guc**, **ALTER DATABASE dbname SET paraname TO value**, and **ALTER USER username SET paraname TO value** are not supported. Global variables are visible to all users.

set_config(setting_name, new_value, is_local)

Description: Sets the parameter and returns a new value.

Return type: text

Note: **set_config** sets the parameter **setting_name** to **new_value**. If **is_local** is **true**, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use **false** instead. The function corresponds to the **SET** statement. For example:

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
-----
off
(1 row)
```

CAUTION

If **behavior_compat_options** is set to **DISABLE_SET_GLOBAL_VAR_ON_DATANODE**, you cannot use this function to set global variables on DNs.

7.27.2 Universal File Access Functions

Universal file access functions provide local access interfaces for files on a database server. Only files in the database cluster directory and the **log_directory** directory can be accessed. Use a relative path for files in the cluster directory, and a path matching the **log_directory** configuration setting for log files. Only database system administrators can use these functions.

pg_ls_dir(dirname text)

Description: Lists files in a directory.

Return type: setof text

Note: **pg_ls_dir** returns all the names in the specified directory, except the special entries "." and "..".

Examples:

```
SELECT pg_ls_dir('./');
       pg_ls_dir
-----
.postgresql.conf.swp
postgresql.conf
pg_tblspc
PG_VERSION
pg_ident.conf
core
server.crt
pg_serial
pg_twophase
postgresql.conf.lock
pg_stat_tmp
pg_notify
pg_subtrans
pg_ctl.lock
pg_xlog
pg_clog
base
pg_snapshots
postmaster.opts
postmaster.pid
server.key.rand
server.key.cipher
pg_multixact
pg_errorinfo
server.key
pg_hba.conf
pg_replslot
.pg_hba.conf.swp
cacert.pem
pg_hba.conf.lock
global
gaussdb.state
(32 rows)
```

pg_read_file(filename text, offset bigint, length bigint)

Description: Returns the content of a text file.

Return type: text

Note: **pg_read_file** returns part of a text file. It can return a maximum of *length* bytes from *offset*. The actual size of fetched data is less than *length* if the end of the file is reached first. If **offset** is negative, it is the length rolled back from the file end. If **offset** and **length** are omitted, the entire file is returned.

Examples:

```
SELECT pg_read_file('postmaster.pid',0,100);
       pg_read_file
-----
53078              +
/srv/BigData/hadoop/data1/coordinator+
1500022474         +
8000               +
/var/run/dws        +
localhost          +
2
(1 row)
```

pg_read_binary_file(filename text [, offset bigint, length bigint,missing_ok boolean])

Description: Returns the content of a binary file.

Return type: bytea

Note: **pg_read_binary_file** is similar to **pg_read_file**, except that the result is a **bytea** value; accordingly, no encoding checks are performed. In combination with the **convert_from** function, this function can be used to read a file in a specified encoding:

```
SELECT convert_from(pg_read_binary_file('filename'), 'UTF8');
```

pg_stat_file(filename text)

Description: Returns status information about a file.

Return type: record

Note: **pg_stat_file** returns a record containing the file size, last access timestamp, last modification timestamp, last file status change timestamp, and a **boolean** value indicating if it is a directory. Typical use cases are as follows:

```
SELECT * FROM pg_stat_file('filename');  
SELECT (pg_stat_file('filename')).modification;
```

Examples:

```
SELECT * FROM pg_stat_file('postmaster.pid');  
  
size |      access      |      modification      |      change  
| creation | isdir  
-----+-----  
117 | 2017-06-05 11:06:34+08 | 2017-06-01 17:18:08+08 | 2017-06-01 17:18:08+08  
| f  
(1 row)  
SELECT (pg_stat_file('postmaster.pid')).modification;  
-----  
2017-06-01 17:18:08+08  
(1 row)
```

7.27.3 Server Signaling Functions

Server signaling functions send control signals to other server processes. Only system administrators can use these functions.

pg_cancel_backend(pid int)

Description: Cancels a current query of the backend.

Return type: boolean

Note: **pg_cancel_backend** sends a query cancellation signal (**SIGINT**) to the backend process identified by **pid**. The PID of an active backend process can be found in the **pid** column of the **pg_stat_activity** view, or can be found by listing the database process using **ps** on the server.

Example:

```
SELECT pid FROM pg_stat_activity WHERE stmt_type = 'RESET';  
pid  
-----  
281471222065200  
(1 row)
```

```
SELECT pg_cancel_backend(281471222065200);
pg_cancel_backend
-----
t
(1 row)
```

pg_reload_conf()

Description: Notifies the server process to reload the configuration file.

Return type: boolean

Note: **pg_reload_conf** sends a **SIGHUP** signal to the server. As a result, all server processes reload their configuration files.

Example:

```
SELECT pg_reload_conf();
pg_reload_conf
-----
t
(1 row)
```

pg_rotate_logfile()

Description: Rotates the log files of the server.

Return type: boolean

Note: **pg_rotate_logfile** instructs the log file manager to immediately switch to a new output file. This function is valid only if the built-in log collector is running.

Example:

```
SELECT pg_rotate_logfile();
pg_rotate_logfile
-----
t
(1 row)
```

pg_terminate_backend(pid int)

Description: Terminates a backend thread.

Return type: boolean

Note: Each of these functions returns **true** if they are successful and **false** otherwise.

Example:

```
SELECT pid FROM pg_stat_activity;
 pid
-----
140657876268816
140433774061312
140433587902208
140433656592128
140433723717376
140433637189376
140433552770816
140433481983744
140433349310208
(9 rows)
```



```
SELECT pg_terminate_backend(140657876268816);
pg_terminate_backend
-----
t
(1 row)
```

pg_wlm_jump_queue(pid int)

Description: Moves a task to the top of the CN queue.

Return type: Boolean

Note: Each of these functions returns **true** if they are successful and **false** otherwise.

Example:

```
SELECT pid FROM pg_stat_activity WHERE stmt_type = 'RESET';
pid
-----
281471222065200
(1 row)

SELECT pg_wlm_jump_queue(281471222065200);
pg_wlm_jump_queue
-----
t
(1 row)
```

gs_wlm_switch_cgroup(pid int, cgroup text)

Description: Moves a job to other Cgroup to improve the job priority.

Return type: Boolean

Note: Each of these functions returns **true** if they are successful and **false** otherwise.

pg_cancel_query(queryId int)

Description: Cancels a current query of the backend. This function is supported in 8.1.2 or later.

Return type: boolean

Note: **pg_cancel_query** sends a query cancellation signal (**SIGINT**) to the backend process identified by **query_id**. The **query_id** of an active backend process can be found in the **query_id** column of the **pg_stat_activity** view.

Example:

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type = 'RESET';
query_id
-----
0
0
(2 rows)

SELECT pg_cancel_query(0);
pg_cancel_query
-----
f
(1 row)
```

pgxc_cancel_query(queryId int)

Description: Cancels the query that is being executed in the current cluster. This function is supported in 8.1.2 or later.

Return type: boolean

Note: If the queries of all nodes have been canceled, **true** is returned. Otherwise, **false** is returned.

Example:

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type = 'RESET';
query_id
-----
      0
      0
(2 rows)

SELECT pgxc_cancel_query(0);
pgxc_cancel_query
-----
f
(1 row)
```

pg_terminate_query(queryId int)

Description: Terminates a current query of the backend. This function is supported in 8.1.2 or later.

Return type: boolean

Example:

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type = 'RESET';
query_id
-----
      0
      0
(2 rows)

SELECT pg_terminate_query(0);
pg_terminate_query
-----
f
(1 row)
```

pgxc_terminate_query(queryId int)

Description: Terminates the query that is being executed in the current cluster. This function is supported in 8.1.2 or later.

Return type: boolean

Example:

```
SELECT query_id FROM pgxc_stat_activity;
query_id
-----
72339069014638631
(1 rows)

SELECT pgxc_terminate_query(72339069014638631);
pgxc_terminate_query
-----
```

```
t  
(1 row)
```

7.27.4 Snapshot Synchronization Functions

Snapshot synchronization functions save the current snapshot and return its identifier.

`create_wdr_snapshot()`

Description: Creates a performance data snapshot.

Return type: text

NOTE

- Only the database administrator **SYSADMIN** can execute this function.
- This function can be executed only on CNs. If it is executed on DN, the following message will be returned: "WDR snapshot can only be created on coordinator."
- Before executing this function, ensure that the value of **enable_wdr_snapshot** is **on**. If its value is **off**, the following message will be returned for this function: "WDR snapshot request can't be executed, because GUC parameter 'enable_wdr_snapshot' is off."
- If the snapshot thread is not started for some reason, for example, the node is restarted, the following message will be returned for this function: "WDR snapshot request can not be accepted, please retry later."
- If this function fails to be executed, the following message will be returned: "Cannot respond to WDR snapshot request."
- If this function is successfully executed, the following message will be returned: "WDR snapshot request has been submitted." This message indicates that the snapshot creation request has been sent to the background snapshot thread, but does not mean that the snapshot has been successfully created.

`kill_snapshot(scope cstring)`

Description: Kills the background snapshot thread. This function sends a command to the background snapshot thread and waits for the thread to stop.

The input parameter **scope** indicates the operation scope. Its value can be **local** or **global**.

- Value **local** indicates killing the snapshot thread on the current CN.
- Value **global** indicates killing the snapshot thread on the current CN as well as those on all the other CNs in the cluster.
- If any other value is passed, error message "Scope is invalid, use "local" or "global"." is displayed.
- The input parameter can be left empty, in which case the default value **local** will be used.

Return type: none

 NOTE

- Only the database administrator **SYSADMIN** can execute this function.
- This function can be executed only on CNs. If it is executed on DN, the following message will be returned: "kill_snapshot can only be executed on coordinator."
- Executing this function sends a kill signal to the background snapshot thread and waits for it to finish. If the snapshot thread is not killed within 100s, the error message "Kill snapshot thread failed" is displayed.

pg_export_snapshot()

Description: Saves the current snapshot and returns its identifier.

Return type: text

Note: **pg_export_snapshot** saves the current snapshot and returns a text string identifying the snapshot. This string must be passed to clients that want to import the snapshot. A snapshot can be imported when the **set transaction snapshot snapshot_id;** command is executed. Doing so is possible only when the transaction is set to the **REPEATABLE READ** isolation level. The output of the function cannot be used as the input of **set transaction snapshot**.

Example:

```
SELECT pg_export_snapshot();
pg_export_snapshot
-----
00000000000A7128-1
(1 row)
```

7.27.5 Advisory Lock Functions

Advisory lock functions manage advisory locks. These functions are only for internal use currently.

pg_advisory_lock(key bigint)

Description: Obtains an exclusive session-level advisory lock.

Return type: void

Note: **pg_advisory_lock** locks resources defined by an application. The resources can be identified using a 64-bit or two nonoverlapped 32-bit key values. If another session locks the resources, the function blocks the resources until they can be used. The lock is exclusive. Multiple locking requests are pushed into the stack. Therefore, if the same resource is locked three times, it must be unlocked three times so that it is released to another session.

pg_advisory_lock(key1 int, key2 int)

Description: Obtains an exclusive session-level advisory lock.

Return type: void

pg_advisory_lock_shared(key bigint)

Description: Obtains a shared session-level advisory lock.

Return type: void

pg_advisory_lock_shared(key1 int, key2 int)

Description: Obtains a shared session-level advisory lock.

Return type: void

Note: **pg_advisory_lock_shared** works in the same way as **pg_advisory_lock**, except the lock can be shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

pg_advisory_unlock(key bigint)

Description: Releases an exclusive session-level advisory lock.

Return type: boolean

pg_advisory_unlock(key1 int, key2 int)

Description: Releases an exclusive session-level advisory lock.

Return type: boolean

Note: **pg_advisory_unlock** releases the obtained exclusive advisory lock. If the release is successful, the function returns **true**. If the lock was not held, it will return **false**. In addition, a SQL warning will be reported by the server.

pg_advisory_unlock_shared(key bigint)

Description: Releases a shared session-level advisory lock.

Return type: boolean

pg_advisory_unlock_shared(key1 int, key2 int)

Description: Releases a shared session-level advisory lock.

Return type: boolean

Note: **pg_advisory_unlock_shared** works in the same way as **pg_advisory_unlock**, except it releases a shared session-level advisory lock.

pg_advisory_unlock_all()

Description: Releases all advisory locks owned by the current session.

Return type: void

Note: **pg_advisory_unlock_all** releases all advisory locks owned by the current session. The function is implicitly invoked when the session ends even if the client is abnormally disconnected.

pg_advisory_xact_lock(key bigint)

Description: Obtains an exclusive transaction-level advisory lock.

Return type: void

pg_advisory_xact_lock(key1 int, key2 int)

Description: Obtains an exclusive transaction-level advisory lock.

Return type: void

Note: **pg_advisory_xact_lock** works in the same way as **pg_advisory_lock**, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg_advisory_xact_lock_shared(key bigint)

Description: Obtains a shared transaction-level advisory lock.

Return type: void

pg_advisory_xact_lock_shared(key1 int, key2 int)

Description: Obtains a shared transaction-level advisory lock.

Return type: void

Note: **pg_advisory_xact_lock_shared** works in the same way as **pg_advisory_lock_shared**, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg_try_advisory_lock(key bigint)

Description: Obtains an exclusive session-level advisory lock if available.

Return type: boolean

Note: **pg_try_advisory_lock** is similar to **pg_advisory_lock**, except **pg_try_advisory_lock** does not block the resource until the resource is released. **pg_try_advisory_lock** either immediately obtains the lock and returns **true** or returns **false**, which indicates the lock cannot be performed currently.

pg_try_advisory_lock(key1 int, key2 int)

Description: Obtains an exclusive session-level advisory lock if available.

Return type: boolean

pg_try_advisory_lock_shared(key bigint)

Description: Obtains a shared session-level advisory lock if available.

Return type: boolean

pg_try_advisory_lock_shared(key1 int, key2 int)

Description: Obtains a shared session-level advisory lock if available.

Return type: boolean

Note: **pg_try_advisory_lock_shared** is similar to **pg_try_advisory_lock**, except **pg_try_advisory_lock_shared** attempts to obtain a shared lock instead of an exclusive lock.

pg_try_advisory_xact_lock(key bigint)

Description: Obtains an exclusive transaction-level advisory lock if available.

Return type: boolean

pg_try_advisory_xact_lock(key1 int, key2 int)

Description: Obtains an exclusive transaction-level advisory lock if available.

Return type: boolean

Note: **pg_try_advisory_xact_lock** works in the same way as **pg_try_advisory_lock**, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

pg_try_advisory_xact_lock_shared(key bigint)

Description: Obtains a shared transaction-level advisory lock if available.

Return type: boolean

pg_try_advisory_xact_lock_shared(key1 int, key2 int)

Description: Obtains a shared transaction-level advisory lock if available.

Return type: boolean

Note: **pg_try_advisory_xact_lock_shared** works in the same way as **pg_try_advisory_lock_shared**, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

7.27.6 Replication Functions

A replication function synchronizes logs and data between instances. It is a statistics or operation method provided by the system to implement HA.

NOTE

Replication functions except statistics queries are internal functions. You are not advised to use them directly.

pg_create_logical_replication_slot('slot_name', 'plugin_name')

Description: Creates a logical replication slot. This is no longer used in 9.1.0.200 and later versions. It is reserved for compatibility with earlier versions.

Parameter:

- **slot_name**
Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: _?-..

- plugin_name

Indicates the name of the plugin.

Value range: a string, supporting only **mppdb_decoding**

Return type: name, text

Note: The first return value is the slot name, and the second is the start LSN position for decoding in the logical replication slot.

pg_create_physical_replication_slot ('slot_name', isDummyStandby)

Description: Creates a physical replication slot.

Parameter:

- slot_name

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: _?.-

- isDummyStandby

Indicates whether the replication slot is the secondary one.

Value range: a boolean value, **true** or **false**

Return type: name, text

Note: The first return value is the slot name, and the second is the start LSN position for decoding in the physical replication slot.

pg_get_replication_slots()

Description: Displays information about all replication slots on the current DN.

Return type: record

The following information is returned:

Table 7-21 pg_get_replication_slots() fields

Field	Type	Description
slot_name	text	Replication slot name
plugin	name	Name of the output plug-in of the logical replication slot
slot_type	text	Replication slot type
datoid	oid	Replication slot's database OID
active	boolean	Whether the replication slot is active
xmin	xid	Transaction ID of the replication slot

Field	Type	Description
catalog_xmin	text	ID of the earliest-decoded transaction corresponding to the logical replication slot.
restart_lsn	text	Xlog file information on the replication slot.
dummy_standby	boolean	Indicates whether the replication slot is the secondary one.

Example:

```
SELECT * FROM pg_get_replication_slots();
 slot_name | plugin | slot_type | datoid | active | xmin | catalog_xmin | restart_lsn | dummy_standby
-----+-----+-----+-----+-----+-----+-----+-----+-----
gs_roach_common |      | physical |      | 0 | f |      | 602861775 | FFFFFFFF/FFFFFF | f
(1 row)
```

pg_drop_replication_slot('slot_name')

Description: Deletes a streaming replication slot.

Parameter:

- slot_name
Indicates the name of the streaming replication slot.
Value range: a string, supporting only letters, digits, and the following special characters: _?-.

Return type: void

pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

Description: Performs decoding but does not go to the next streaming replication slot. (The decoding result will be returned again on future calls.) This is no longer used in 9.1.0.200 and later versions. It is reserved for compatibility with earlier versions.

Parameter:

- slot_name
Indicates the name of the streaming replication slot.
Value range: a string, supporting only letters, digits, and the following special characters: _?-.
- LSN
Indicates a target LSN. Decoding is performed only when an LSN is less than or equal to this value.
Value range: a string, in the format of xlogid/xrecoff, for example, '1/2AAFC60' (If this parameter is set to **NULL**, the target LSN indicating the end position of decoding is not specified.)

- **upto_nchanges**

Indicates the number of decoded records (including the **begin** and **commit** timestamps). Assume that there are three transactions, which involve 3, 5, and 7 records, respectively. If **upto_nchanges** is **4**, 8 records of the first two transactions will be decoded. Specifically, decoding is stopped when the number of decoded records exceeds 4 after decoding in the first two transactions is finished.

Value range: a non-negative integer

 NOTE

If any of the **LSN** and **upto_nchanges** values are reached, decoding ends.

- **options** (optional)

- **include-xids**

Indicates whether the decoded **data** column contains XID information.

Valid value: **0** and **1**. The default value is **1**.

- **0**: The decoded **data** column does not contain XID information.

- **1**: The decoded **data** column contains XID information.

- **skip-empty-xacts**

Indicates whether to ignore empty transaction information during decoding.

Valid value: **0** and **1**. The default value is **0**.

- **0**: The empty transaction information is not ignored during decoding.

- **1**: The empty transaction information is ignored during decoding.

- **include-timestamp**

Indicates whether decoding information contains the **commit** timestamp.

Valid value: **0** and **1**. The default value is **0**.

- **0**: The decoding information does not contain the **commit** timestamp.

- **1**: The decoding information contains the **commit** timestamp.

Return type: text, uint, text

Note: The function returns the decoding result. Each decoding result contains three columns, corresponding to the above return types and indicating the LSN position, XID, and decoded content, respectively.

pg_logical_slot_get_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

Description: Performs decoding and goes to the next streaming replication slot. This is no longer used in 9.1.0.200 and later versions. It is reserved for compatibility with earlier versions.

Parameter: This function has the same parameters as **pg_logical_slot_peek_changes**. For details, see

`pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')`.

`pg_replication_slot_advance ('slot_name', 'LSN')`

Description: Directly goes to the streaming replication slot for a specified LSN, without outputting any decoding result.

Parameter:

- `slot_name`
Indicates the name of the streaming replication slot.
Value range: a string, supporting only letters, digits, and the following special characters: `_?-`.
- `LSN`
Indicates a target LSN. Next decoding will be performed only in transactions whose commission position is greater than this value. If an input LSN is smaller than the position recorded in the current streaming replication slot, the function directly returns. If the input LSN is greater than the LSN of the current physical log, the latter LSN will be directly used for decoding.
Value range: a string, in the format of `xlogid/xrecoff`

Return type: name, text

Note: A return result contains the slot name and LSN that is actually used for decoding.

`pg_stat_get_data_senders()`

Description: Displays statistics about replication sending threads on all data page on the current DN.

Return type: record

The following information is returned:

Table 7-22 `pg_stat_get_data_senders()` fields

Field	Type	Description
<code>pid</code>	<code>bigint</code>	Thread PID
<code>sender_pid</code>	<code>integer</code>	Current sender PID
<code>local_role</code>	<code>text</code>	Local role
<code>peer_role</code>	<code>text</code>	Peer role
<code>state</code>	<code>text</code>	Current sender's replication status
<code>catchup_start</code>	timestamp with time zone	Startup time of a catchup task
<code>catchup_end</code>	timestamp with time zone	End time of a catchup task

Field	Type	Description
queue_size	text	Data queue size
queue_lower_tail	text	Position of data queue tail 1
queue_header	text	Position of data queue header
queue_upper_tail	text	Position of data queue tail 2
send_position	text	Sending position of the sender
receive_position	text	Receiving position of the receiver
catchup_type	text	Catchup task type, full or incremental
catchup_bcm_filename	text	BCM file executed by the current catchup task
catchup_bcm_finished	integer	Number of BCM files completed by a catchup task
catchup_bcm_total	integer	Total number of BCM files to be operated by a catchup task
catchup_percent	text	Completion percentage of a catchup task
catchup_remaining_time	text	Estimated remaining time of a catchup task

pg_stat_get_wal_senders()

Description: Displays statistics about replication sending threads on all WALs on the current DN.

Return type: record

The following information is returned:

Table 7-23 pg_stat_get_wal_senders() fields

Field	Type	Description
pid	bigint	Thread PID
sender_pid	integer	Current sender PID
local_role	text	Local role
peer_role	text	Peer role
peer_state	text	Peer status
state	text	Current sender's replication status

Field	Type	Description
catchup_start	timestamp with time zone	Startup time of a catchup task
catchup_end	timestamp with time zone	End time of a catchup task
sender_sent_location	text	Location where the sender sends LSNs
sender_write_location	text	Location where the sender writes LSNs
sender_flush_location	text	Location where the sender flushes LSNs
sender_replay_location	text	Location where the sender replays LSNs
receiver_received_location	text	Location where the receiver receives LSNs
receiver_write_location	text	Location where the receiver writes LSNs
receiver_flush_location	text	Location where the receiver flushes LSNs
receiver_replay_location	text	Location where the receiver replays LSNs
sync_percent	text	Specifies the synchronization percentage.
sync_state	text	Synchronization state (asynchronous duplication, synchronous duplication, or potential synchronization)
sync_priority	integer	Priority of synchronous duplication (0 indicates asynchronization)
sync_most_available	text	Whether to block the active node when the synchronization on the standby node fails
channel	text	WALSender channel information

pg_stat_get_wal_receiver()

Description: Displays statistics about replication receiving threads on all WALs on the current DN.

Return type: record

The following information is returned:

Table 7-24 pg_stat_get_wal_receiver()

Field	Type	Description
receiver_pid	integer	Current receiver PID
local_role	text	Local role
peer_role	text	Peer role
peer_state	text	Peer status
state	text	Current receiver's replication status
sender_sent_location	text	Location where the sender sends LSNs
sender_write_location	text	Location where the sender writes LSNs
sender_flush_location	text	Location where the sender flushes LSNs
sender_replay_location	text	Location where the sender replays LSNs
receiver_received_location	text	Location where the receiver receives LSNs
receiver_write_location	text	Location where the receiver writes LSNs
receiver_flush_location	text	Location where the receiver flushes LSNs
receiver_replay_location	text	Location where the receiver replays LSNs
sync_percent	text	Specifies the synchronization percentage.
channel	text	WALReceiver channel information

pg_stat_get_stream_replications()

Description: Displays information about all replication statistics on the current DN.

Return type: record

The following information is returned:

Table 7-25 pg_stat_get_stream_replications()

Field	Type	Description
local_role	text	Local role

Field	Type	Description
static_connections	integer	Connection statistics
db_state	text	Database status
detail_information	text	Detail information

Example:

```
SELECT * FROM pg_stat_get_stream_replications();
local_role | static_connections | db_state | detail_information
-----+-----+-----+-----
Normal    |          0 | Normal  | Normal
(1 row)
```

pg_stat_xlog_space()

Description: Displays the Xlog space usage on the current DN.

Return type: record

The following information is returned:

Table 7-26 pg_stat_xlog_space()

Column	Type	Description
xlog_files	bigint	Number of all identified xlog files in the pg_xlog directory, excluding the backup and archive_status subdirectories.
xlog_size	bigint	Total size (MB) of all identified xlog files in the pg_xlog directory, excluding the backup and archive_status subdirectories.
other_size	bigint	Total size (MB) of files in the backup and archive_status subdirectories of the pg_xlog directory.

Example:

```
SELECT * FROM pg_stat_xlog_space();
xlog_files | xlog_size | other_size
-----+-----+-----
       79 |      1264 |          0
(1 row)
```

pgxc_stat_xlog_space()

Description: Displays the Xlog space usage on all active DN's.

Return type: record

The following information is returned:

Table 7-27 pgxc_stat_xlog_space()

Column	Type	Description
node_name	name	Node name
xlog_files	bigint	Number of all identified xlog files in the pg_xlog directory, excluding the backup and archive_status subdirectories.
xlog_size	bigint	Total size (MB) of all identified xlog files in the pg_xlog directory, excluding the backup and archive_status subdirectories.
other_size	bigint	Total size (MB) of files in the backup and archive_status subdirectories of the pg_xlog directory.

Example:

```
SELECT * FROM pgxc_stat_xlog_space();
 node_name | xlog_files | xlog_size | other_size
-----+-----+-----+-----
dn_6001_6002 | 73 | 1168 | 0
dn_6003_6004 | 73 | 1168 | 0
dn_6005_6006 | 73 | 1168 | 0
cn_5003 | 79 | 1264 | 0
cn_5001 | 72 | 1152 | 0
cn_5002 | 73 | 1168 | 0
(6 rows)
```

7.27.7 Resource Management Functions

This section describes the functions of the resource management module.

gs_switch_respool(query_id bigint, resource_pool_name name)

Description: Job resource pool switchover function is supported by cluster versions 9.1.0 and later.

The value of **resource_pool_name** must contain less than 64 characters. Upon calling this function, the system will decide whether to start a job in the queuing phase when switching to the new resource pool. If a job is running, the system will switch to the new resource pool to start the queuing job from the original resource pool. Additionally, the CPU control of the new resource pool will limit the queuing job.

This function supports resource pool switchover in both static and dynamic management and control modes, as well as resource pool switchover using short queries.

Return type: bool

CAUTION

- Only administrators or users with administrator permissions can use this function. Common users are not authorized to use it. To use this function, the cluster's CPU control function, specifically the **Cgroup** function, must be functioning normally. You can use the **pgxc_cgroup_reload_conf()** function to check whether **Cgroup** is normal. If the command output contains non-success information, rectify the fault.
- After the resource pool is switched, the function performs management and control based on the exception rules of the new resource pool.
- If a job is in the global concurrent queue, which means it is waiting in the global queue, it will not take effect right away after the resource pool is switched. At this point, the job has not entered the resource pool queue yet.
- 1. Job CPU control switchover supports CNs and DNs. For example, if resource pool A uses 40 cores and resource pool B uses 5 cores, switching a job from pool A to pool B will execute the job on the 5 cores of pool B. This results in slower execution but isolates the job, ensuring that jobs in pool A are unaffected, thus degrading the statement.
- When switching resource pools, use similar statements to perform batch switchovers. Ensure that the jobs being switched are in the same state, such as all queuing or all running. If switching between different types of statements, it is recommended to switch them twice. Switching them together may exceed the concurrent statement limit, causing abnormal job wake-ups.

```
SELECT gs_switch_respool (queryid,'xxx') FROM xxx where xxx;
```
- The restrictions on switching the statement resource pool are as follows:
 - If jobs from resource pool A are switched to pool B, jobs at the head of the queue in pool A will be woken up. If the concurrent job limit in pool B is reached, job A1 will not run immediately.
 - Switch job A2 from queuing in pool A to pool B. If the concurrent job limit in pool B is reached, job A2 will continue to queue. If not, job A2 will wake up and run immediately.
 - During resource pool switchover, only the same lane can be switched. For example, switching the fast lane of pool A will switch the fast lane of pool B. If the short query acceleration function is not enabled in pool B, the job runs directly regardless of its status in pool B.
- This function can be used only when all nodes are normal. If a node is faulty, the function fails to be executed.

Example:

While executing the statement with ID **74590868828368101**, the administrator successfully switches it to the **poolg1** resource pool.

```
CALL gs_switch_respool(74590868828368101,'poolg1');
gs_switch_respool
-----
t
(1 row)
```

gs_increase_except_num(unique_sql_id int8)

Description: Records job exception information. The input parameter must be greater than 0. If this function is invoked, the number of job exceptions will be

increased by 1 and the latest job exception time will be updated to the current time.

Return type: bool

Example:

```
SELECT gs_increase_except_num(111);
gs_increase_except_num
-----
t
(1 row)
```

gs_increase_except_num(unique_sql_id int8, except_num int4)

Description: Records job exception information. The input parameter must be greater than 0. If this function is invoked, the number of job exceptions will be increased by **except_num** and the latest job exception time will be updated to the current time.

Return type: bool

Example:

```
SELECT gs_increase_except_num(111, 4);
gs_increase_except_num
-----
t
(1 row)
```

gs_increase_except_num(unique_sql_id int8, except_num int4, except_time int8)

Description: Records job exception information. The input parameter must be greater than 0. If this function is invoked, the number of job exceptions will be increased by **except_num** and the latest job exception time will be updated to **except_time**, a timestamp. This function is mainly invoked internally.

Return type: bool

Example:

```
SELECT gs_increase_except_num(111, 4, 714623414421256);
gs_increase_except_num
-----
t
(1 row)
```

gs_append_blocklist(unique_sql_id int8)

Description: Adds a job to the blocklist and update the blocklist information in **GS_BLOCKLIST_QUERY**.

Return type: bool

Example:

```
SELECT gs_append_blocklist(111);
gs_append_blocklist
-----
t
(1 row)
```

gs_remove_blocklist(unique_sql_id int8)

Description: Removes a job from the blocklist. and update the blocklist information in **GS_BLOCKLIST_QUERY**.

Return type: bool

Example:

```
SELECT gs_remove_blocklist(111);  
gs_append_blocklist  
-----  
t  
(1 row)
```

gs_increase_sql_except_num(sql_hash text)

Description: Records job exception information. If this function is invoked, the number of job exceptions will be increased by 1 and the latest job exception time will be updated to the current time. This function is supported only by 9.1.0.200 and later cluster versions and is used for internal calling.

Return type: bool

Example:

```
SELECT gs_increase_sql_except_num('sql_7bf6381d2a5a456a1936027198ad8b12');  
gs_increase_sql_except_num  
-----  
t  
(1 row)
```

gs_increase_sql_except_num(sql_hash text, except_num int4)

Description: Records job exception information. If this function is invoked, the number of job exceptions will be increased by **except_num** and the latest job exception time will be updated to the current time. This function is supported only by 9.1.0.200 and later cluster versions and is used for internal calling.

Return type: bool

Example:

```
SELECT gs_increase_sql_except_num('sql_7bf6381d2a5a456a1936027198ad8b12', 4);  
gs_increase_sql_except_num  
-----  
t  
(1 row)
```

gs_increase_sql_except_num(sql_hash text, except_num int4, except_time int8)

Description: Records job exception information. If this function is invoked, the number of job exceptions will be increased by **except_num** and the latest job exception time will be updated to **except_time**. **except_time** functions as a timestamp here. This function is supported only by 9.1.0.200 and later cluster versions and is used for internal calling.

Return type: bool

Example:

```
SELECT gs_increase_sql_except_num('sql_7bf6381d2a5a456a1936027198ad8b12', 4, 714623414421256);
gs_increase_sql_except_num
-----
t
(1 row)
```

gs_append_blocklist(sql_hash text)

Description: Adds a job to the blocklist and updates the blocklist information in **GS_BLOCKLIST_SQL**. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: bool

Example:

```
SELECT gs_append_blocklist('sql_7bf6381d2a5a456a1936027198ad8b12');
gs_append_blocklist
-----
t
(1 row)
```

gs_remove_blocklist(sql_hash text)

Description: Removes a job from the blocklist and updates the blocklist information in **GS_BLOCKLIST_SQL**. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: bool

Example:

```
SELECT gs_remove_blocklist('sql_7bf6381d2a5a456a1936027198ad8b12');
gs_remove_blocklist
-----
t
(1 row)
```

gs_wlm_rebuild_except_rule_hash()

Description: Rebuilds the memory hash table of exception rules on the current node. The monitoring thread obtains the exception rule thresholds from the memory hash table in real time. When the hash table is abnormal, this function can be used to rebuild the table.

Return type: bool

Example:

```
SELECT gs_wlm_rebuild_except_rule_hash();
gs_wlm_rebuild_except_rule_hash
-----
t
(1 row)
```

gs_wlm_readjust_user_space(oid)

Description: This function calibrates the permanent storage space of a user. The input parameter is the user OID. If the input parameter is set to **0**, the permanent storage space of all users is calibrated.

Return type: text

Example:

```
SELECT gs_wlm_readjust_user_space(0);  
gs_wlm_readjust_user_space  
-----  
Exec Success  
(1 row)
```

pgxc_wlm_readjust_schema_space()

Description: This function calibrates the permanent storage space of a schema.

Return type: text

Example:

```
SELECT pgxc_wlm_readjust_schema_space();  
pgxc_wlm_readjust_schema_space  
-----  
Exec Success  
(1 row)
```

pgxc_wlm_readjust_relfilenode_size_table()

Description: Space statistics calibration function. It does not recreate the **PG_RELFILENODE_SIZE** system catalog but recalibrates the user and schema space.

NOTE

Due to transaction isolation between the calibration function and other services, the calibration function is invisible to other services that are being executed. As a result, the calibration function does not involve space changes of such services. To avoid space difference errors after calibration, you are advised to use the space calibration function to perform calibration when the space is stable.

Return type: text

Example:

```
SELECT pgxc_wlm_readjust_schema_space();  
pgxc_wlm_readjust_relfilenode_size_table  
-----  
Exec Success  
(1 row)
```

pgxc_wlm_readjust_relfilenode_size_table(integer)

Description: Space statistics calibration function

NOTE

Due to transaction isolation between the calibration function and other services, the calibration function is invisible to other services that are being executed. As a result, the calibration function does not involve space changes of such services. To avoid space difference errors after calibration, you are advised to use the space calibration function to perform calibration when the space is stable.

Parameter: The value ranges from 0 to 4. Different input parameter values indicate different calibration granularities.

- If the input parameter is set to **0** (default value), the **PG_RELFILENODE_SIZE** system catalog is not rebuilt and the user and schema space are recalibrated.

- If the input parameter is set to **1**, the **PG_RELFILENODE_SIZE** system catalog is rebuilt and the user and schema space are recalibrated.
- If the input parameter is set to **2**, the **PG_RELFILENODE_SIZE** system catalog is rebuilt.
- If the input parameter is set to **3**, the schema space is recalibrated.
- If the input parameter is set to **4**, the user space is recalibrated.

Return type: text

Example:

```
SELECT * FROM pgxc_wlm_readjust_relfilenode_size_table(1);
      result
-----
Exec success
(1 row)
```

pgxc_wlm_get_schema_space(cstring)

Description: Obtains the schema space of each instance in a specified logical cluster on the CN.

Return type: record

The following table describes return columns.

Column	Type	Description
schemaname	text	Schema name
schemaid	oid	Schema OID
databasename	text	Database name
databaseid	oid	Database OID
nodename	text	Instance name
nodegroup	text	Name of the node group
usedspace	bigint	Size of the used space
permSPACE	bigint	Upper limit of the space

Example:

```
SELECT * FROM pgxc_wlm_get_schema_space('group1');
 schemaname | schemaid | databasename | databaseid | nodename | nodegroup | usedspace | permSPACE
-----+-----+-----+-----+-----+-----+-----+-----
pg_catalog  |      11 | test1       | 16384 | datanode1 | installation | 9469952 | -1
public      |     220 | postgres   | 15253 | datanode1 | installation | 25280512 | -1
pg_toast    |      99 | test1       | 16384 | datanode1 | installation | 1859584 | -1
cstore      |     100 | test1       | 16384 | datanode1 | installation | 0 | -1
data_redis  |    18106 | postgres   | 15253 | datanode1 | installation | 655360 | -1
data_redis  |    18116 | test1       | 16384 | datanode1 | installation | 0 | -1
public      |     220 | test1       | 16384 | datanode1 | installation | 16384 | -1
dbms_om     |     3987 | postgres   | 15253 | datanode1 | installation | 0 | -1
dbms_job    |     3988 | postgres   | 15253 | datanode1 | installation | 0 | -1
```

dbms_om		3987		test1		16384		datanode1		installation		0		-1
dbms_job		3988		test1		16384		datanode1		installation		0		-1
sys		11693		postgres		15253		datanode1		installation		0		-1
sys		11693		test1		16384		datanode1		installation		0		-1
utl_file		14644		postgres		15253		datanode1		installation		0		-1
utl_raw		14669		postgres		15253		datanode1		installation		0		-1
dbms_sql		14674		postgres		15253		datanode1		installation		0		-1
dbms_output		14662		postgres		15253		datanode1		installation		0		-1
dbms_random		14666		postgres		15253		datanode1		installation		0		-1
dbms_lob		14701		postgres		15253		datanode1		installation		0		-1
information_schema		14300		postgres		15253		datanode1		installation		294912		-1
information_schema		14300		test1		16384		datanode1		installation		294912		-1
utl_file		14644		test1		16384		datanode1		installation		0		-1
dbms_output		14662		test1		16384		datanode1		installation		0		-1
dbms_random		14666		test1		16384		datanode1		installation		0		-1
utl_raw		14669		test1		16384		datanode1		installation		0		-1
dbms_sql		14674		test1		16384		datanode1		installation		0		-1
dbms_lob		14701		test1		16384		datanode1		installation		0		-1
pg_catalog		11		postgres		15253		datanode1		installation		13049856		-1
redisuser		16387		postgres		15253		datanode1		installation		630784		-1
pg_toast		99		postgres		15253		datanode1		installation		3080192		-1
cstore		100		postgres		15253		datanode1		installation		2408448		-1
pg_catalog		11		test1		16384		datanode2		installation		9469952		-1
public		2200		postgres		15253		datanode2		installation		25214976		-1
pg_toast		99		test1		16384		datanode2		installation		1859584		-1
cstore		100		test1		16384		datanode2		installation		0		-1
data_redis		18106		postgres		15253		datanode2		installation		655360		-1
data_redis		18116		test1		16384		datanode2		installation		0		-1
public		2200		test1		16384		datanode2		installation		16384		-1
dbms_om		3987		postgres		15253		datanode2		installation		0		-1
dbms_job		3988		postgres		15253		datanode2		installation		0		-1
dbms_om		3987		test1		16384		datanode2		installation		0		-1
dbms_job		3988		test1		16384		datanode2		installation		0		-1

pgxc_wlm_analyze_schema_space(cstring)

Description: Obtains the schema space of a specified logical cluster on the CN.

Return type: record

The following table describes return columns.

Column	Type	Description
schemaname	text	Schema name
databasename	text	Database name
nodegroup	text	Name of the node group
total_value	bigint	Total cluster space in the current schema
avg_value	bigint	Average space of instances in the current schema
skew_percent	integer	Skew ratio
extend_info	text	Extended information, including the maximum space of a single instance, minimum space of a single instance, and names of the instances with the maximum space or minimum space

Example:

```
SELECT * FROM pgxc_wlm_analyze_schema_space('group1');
  schemaname | databasename | nodegroup | total_value | avg_value | skew_percent |
extend_info
-----+-----+-----+-----+-----+-----+
pg_catalog   | test1        | installation | 56819712 | 9469952 | 0 | min:9469952
datanode1,max:9469952 datanode1
public       | postgres     | installation | 150495232 | 25082538 | 0 | min:24903680
datanode6,max:25280512 datanode1
pg_toast     | test1        | installation | 11157504 | 1859584 | 0 | min:1859584
datanode1,max:1859584 datanode1
cstore      | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
data_redis   | postgres     | installation | 1966080 | 327680 | 50 | min:0
datanode4,max:655360 datanode1
data_redis   | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
public       | test1        | installation | 98304 | 16384 | 0 | min:16384
datanode1,max:16384 datanode1
dbms_om      | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
dbms_job     | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
dbms_om      | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_job     | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
sys          | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
sys          | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
utl_file     | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
utl_raw      | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_sql     | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_output  | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
dbms_random  | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
dbms_lob     | postgres     | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
information_schema | postgres | installation | 1769472 | 294912 | 0 | min:294912
datanode1,max:294912 datanode1
information_schema | test1    | installation | 1769472 | 294912 | 0 | min:294912
datanode1,max:294912 datanode1
utl_file     | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_output  | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
dbms_random  | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0
datanode1
utl_raw      | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_sql     | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
dbms_lob     | test1        | installation | 0 | 0 | 0 | min:0 datanode1,max:0 datanode1
pg_catalog   | postgres     | installation | 75431936 | 12571989 | 3 | min:12124160
datanode4,max:13049856 datanode1
redisuser    | postgres     | installation | 1884160 | 314026 | 50 | min:16384
datanode4,max:630784 datanode1
pg_toast     | postgres     | installation | 17154048 | 2859008 | 7 | min:2637824
datanode4,max:3080192 datanode1
cstore      | postgres     | installation | 15294464 | 2549077 | 5 | min:2408448
datanode1,max:2703360 datanode6
(31 rows)
```

gs_wlm_set_queryband_action(cstring,cstring,int4)

Description: Sets the action and query order of **query_band**.

Return type: boolean

The following table describes the input parameters.

Name	Type	Description
qband	cstring	Query band key-value pair. The maximum length is 63 characters.
action	cstring	Action associated to a query band
order	int4	Query band query order. The default value is -1 .

Example:

```
SELECT * FROM gs_wlm_set_queryband_action('a=1','respool=p1');
gs_wlm_set_queryband_action
-----
t
(1 row)

SELECT * FROM gs_wlm_set_queryband_action('a=3','respool=p1;priority=rush',1);
gs_wlm_set_queryband_action
-----
t
(1 row)
```

gs_wlm_set_queryband_order(cstring,int4)

Description: Sets the **query_band** query order.

Return type: boolean

The following table describes the input parameters.

Name	Type	Description
qband	cstring	query_band key-value pairs
order	int4	Query band query order. The default value is -1 .

Example:

```
SELECT * FROM gs_wlm_set_queryband_order('a=1',2);
gs_wlm_set_queryband_order
-----
t
(1 row)
```

gs_wlm_get_queryband_action(cstring)

Description: Obtains the action and query order of **query_band**.

Return type: record

The following table describes return columns.

Column	Type	Description
qband	cstring	query_band key-value pairs
respool_id	Oid	OID of the resource pool associated with query_band
respool	text	Name of the resource pool associated with query_band
priority	text	Intra-queue priority associated with query_band
qborder	int4	query_band query order

Example:

```
SELECT * FROM gs_wlm_get_queryband_action('a=1');
qband | respool_id | respool | priority | qborder
-----+-----+-----+-----+-----
a=1   | 16388 | p1     | Medium   | -1
(1 row)
```

gs_cgroup_reload_conf()

Description: This function loads the Cgroup configuration file online on the current instance.

Return type: record

The following table describes return columns.

Column	Type	Description
node_name	text	Instance name
node_host	text	IP address of the node where the instance is located
result	text	Whether Cgroup online loading is successful

Example:

```
SELECT * FROM gs_cgroup_reload_conf();
node_name | node_host | result
-----+-----+-----
cn_5001   | 192.168.178.35 | success
```

pgxc_cgroup_reload_conf()

Description: This function loads the Cgroup configuration file online on all instances of the system.

Return type: record

The following table describes return columns.

Column	Type	Description
node_name	text	Instance name
node_host	text	IP address of the node where the instance is located
result	text	Whether Cgroup online loading is successful

Example:

```
SELECT * FROM pgxc_cgroup_reload_conf();
 node_name | node_host | result
-----+-----+-----
dn_6025_6026 | 192.168.178.177 | success
dn_6049_6050 | 192.168.179.79 | success
dn_6051_6052 | 192.168.179.79 | success
dn_6055_6056 | 192.168.179.79 | success
dn_6067_6068 | 192.168.181.57 | success
dn_6023_6024 | 192.168.178.39 | success
dn_6009_6010 | 192.168.181.21 | success
dn_6011_6012 | 192.168.181.21 | success
dn_6015_6016 | 192.168.181.21 | success
dn_6029_6030 | 192.168.178.177 | success
dn_6031_6032 | 192.168.178.177 | success
dn_6045_6046 | 192.168.179.45 | success
cn_5001      | 192.168.178.35 | success
cn_5003      | 192.168.178.39 | success
dn_6061_6062 | 192.168.181.179 | success
cn_5006      | 192.168.179.45 | success
cn_5004      | 192.168.178.177 | success
cn_5002      | 192.168.181.21 | success
cn_5005      | 192.168.178.187 | success
dn_6019_6020 | 192.168.178.39 | success
dn_6007_6008 | 192.168.178.35 | success
dn_6071_6072 | 192.168.181.57 | success
dn_6003_6004 | 192.168.178.35 | success
dn_6013_6014 | 192.168.181.21 | success
dn_6035_6036 | 192.168.178.187 | success
dn_6037_6038 | 192.168.178.187 | success
dn_6001_6002 | 192.168.178.35 | success
dn_6063_6064 | 192.168.181.179 | success
dn_6005_6006 | 192.168.178.35 | success
dn_6057_6058 | 192.168.181.179 | success
dn_6069_6070 | 192.168.181.57 | success
dn_6027_6028 | 192.168.178.177 | success
dn_6059_6060 | 192.168.181.179 | success
dn_6041_6042 | 192.168.179.45 | success
dn_6043_6044 | 192.168.179.45 | success
dn_6047_6048 | 192.168.179.45 | success
dn_6033_6034 | 192.168.178.187 | success
dn_6065_6066 | 192.168.181.57 | success
dn_6021_6022 | 192.168.178.39 | success
dn_6017_6018 | 192.168.178.39 | success
dn_6039_6040 | 192.168.178.187 | success
dn_6053_6054 | 192.168.179.79 | success
(42 rows)
```

pgxc_cgroup_reload_conf(text)

Description: This function loads the Cgroup configuration file online on a node. The input parameter is the IP address of the node.

Return type: record

The following table describes return columns.

Column	Type	Description
node_name	text	Instance name
node_host	text	IP address of the node where the instance is located
result	text	Whether Cgroup online loading is successful

Example:

```
SELECT * FROM pgxc_cgroup_reload_conf('192.168.178.35');
 node_name | node_host | result
-----+-----+-----
 cn_5001   | 192.168.178.35 | success
 dn_6007_6008 | 192.168.178.35 | success
 dn_6003_6004 | 192.168.178.35 | success
 dn_6001_6002 | 192.168.178.35 | success
 dn_6005_6006 | 192.168.178.35 | success
(5 rows)
```

gs_wlm_node_recover(boolean isForce)

Description: This function updates and restores job information and counts on the CCN in dynamic resource management mode. It can be executed only by administrators, and is usually used to restore a faulty CN after it was restarted. This function is called by the Cluster Manager (CM). Its usage is as follows:

- If this function is executed by CN, it instructs the CCN to clear job information and counts on the CN.
- If this function is executed by CCN, it resets job counts and obtains the latest slow lane job information from the CN.

Return type: bool

gs_wlm_node_clean(cstring nodename)

Description: On the CCN in dynamic resource management mode, clears the job information and counts of a specified CN. This function can be executed only by administrators, and is usually used to restore a faulty CN after it was restarted. This function is called by the Cluster Manager (CM). Generally, users are not advised to call it.

Return type: bool

pg_stat_get_wlm_node_resource_info(int4)

Description: This function displays the summary of all DN resources.

Return type: record

The following table describes return columns.

Column	Type	Description
min_mem_util	integer	Minimum memory usage of a DN
max_mem_util	integer	Maximum memory usage of a DN
min_cpu_util	integer	Minimum CPU usage of a DN
max_cpu_util	integer	Maximum CPU usage of a DN
min_io_util	integer	Minimum I/O usage of a DN
max_io_util	integer	Maximum I/O usage of a DN
phy_usemem_rate	integer	Maximum physical memory usage

pg_stat_get_workload_struct_info()

Description: Load management function for locating CCN queuing problems. This function is an internal function. To use this function, contact technical support.

Return type: record

pgxc_query_resource_info(query_id bigint)

Description: This function displays resource monitoring information about the statement with a specified query ID on all DNs. It is supported only by clusters of version 9.1.0.100 or later.

Return type: record

The following table describes return columns.

Column	Type	Description
node_name	text	Instance name, which contains only DNs.
user_id	oid	User ID.
queryid	bigint	Internal query ID used for statement execution.
used_mem	int	Memory used by the statement on the current DN. The unit is MB.
cpu_time	bigint	CPU time of a statement on the current DN. The unit is ms.
used_cpu	double	Number of CPUs used by the statement on the current DN.
spill_size	bigint	Amount of data spilled to disks on the current DN. The default value is 0. The unit is MB.
read_bytes	bigint	Number of logical read bytes used by the statement on the current DN. The unit is KB.

Column	Type	Description
write_bytes	bigint	Number of logical write bytes used by the statement on the current DN. The unit is KB.
read_count	bigint	Number of logical reads used by the statement on the current DN.
write_count	bigint	Number of logical writes used by the statement on the current DN.
read_speed	int	Logical read rate used by the statement on the current DN. The unit is KB/s.
write_speed	int	Logical write rate used by the statement on the current DN. The unit is KB/s.
curr_iops	int	I/O operations per second of the statement on the current DN. It is recorded as a count in a column-store table and as a count of 10,000 in a row-store table.
send_pkg	bigint	Total number of communication packages sent by a statement across all DNs.
recv_pkg	bigint	Total number of communication packages received by a statement across all DNs.
send_bytes	bigint	Total sent data of the statement stream, in byte.
recv_bytes	bigint	Total received data of the statement stream, in byte.
send_speed	int	Network sending rate of the statement on the current DN. The unit is KB/s.
recv_speed	int	Network receiving rate of the statement on the current DN. The unit is KB/s.

7.27.8 Memory Management Functions

This function is supported only by cluster versions 9.1.0 and later.

pg_shared_chunk_detail(contextname char(64))

Description: This function queries information about all chunks requested by the memory context in a specified shared memory.

The **contextname** parameter indicates the name of the memory context.

Before using this function, use the **pg_shared_chunk_dump(contextname char(64))** function to print related information to a file.

Return type: record

Example:

```
SELECT * FROM pg_shared_chunk_detail('pgstat file hash table');
   parent   | level | file_name | line_number | chunk_size | requested_number | total_size
-----+-----+-----+-----+-----+-----+-----
pgstat file context | 2 | dynahash.cpp | 158 | 2048 | 1 | 2048
pgstat file context | 2 | dynahash.cpp | 158 | 256 | 1 | 256
pgstat file context | 2 | dynahash.cpp | 158 | 4096 | 9 | 36864
pgstat file context | 2 | dynahash.cpp | 158 | 8192 | 4 | 32768
(4 rows)
```

pv_session_chunk_detail(tid int, contextname char(64))

Description: This function queries information about all chunks applied for by a memory context created by a specified thread.

The **tid** parameter indicates the thread ID, while the **contextname** parameter indicates the memory context name.

To use this function, you need to use the **pv_session_chunk_dump(tid int, contextname char(64))** function to print related information to a file.

Return type: record

Example:

```
SELECT * FROM pv_session_chunk_detail(140178810990936, 'Timezones');
   parent   | level | file_name | line_number | chunk_size | requested_number | total_size
-----+-----+-----+-----+-----+-----+-----
TopMemoryContext | 1 | dynahash.cpp | 158 | 1280 | 2 | 2560
TopMemoryContext | 1 | dynahash.cpp | 158 | 160 | 1 | 160
TopMemoryContext | 1 | dynahash.cpp | 158 | 2560 | 2 | 5120
TopMemoryContext | 1 | localtime.cpp | 1965 | 128 | 1 | 128
TopMemoryContext | 1 | localtime.cpp | 1965 | 448 | 1 | 448
(5 rows)
```

pg_shared_chunk_dump(contextname char(64))

Description: This function prints information about all chunks applied for by the memory context in the specified shared memory as a file.

The **contextname** parameter indicates the name of the memory context.

Return type: Boolean

Example:

```
SELECT * FROM pg_shared_chunk_dump('pgstat file hash table');
pg_shared_chunk_dump
-----
t
(1 row)
```

pv_session_chunk_dump(tid int, contextname char(64))

Description: This function prints information about all chunks applied for by a memory context created by a specified thread into a file.

The **tid** parameter indicates the thread ID, while the **contextname** parameter indicates the memory context name.

Return type: Boolean

Example:

```
SELECT * FROM pv_session_chunk_dump(140472797325280, 'Timezones');
pv_session_chunk_dump
-----
t
(1 row)
```

7.27.9 Other Functions

pgxc_pool_check()

Description: Checks whether the connection data buffered in the pool is consistent with **pgxc_node**. This parameter has been discarded in 8.3.0 and later cluster versions.

Return type: boolean

pgxc_pool_reload()

Description: Updates the connection information buffered in the pool.

Return type: boolean

pgxc_lock_for_backup()

Description: Locks the cluster before backup. Backup is performed to restore data on new nodes.

Return type: boolean

NOTE

pgxc_lock_for_backup locks a cluster before **gs_dump** or **gs_dumpall** is used to back up the cluster. After a cluster is locked, operations changing the system structure are not allowed. This function does not affect DML statements.

pg_pool_validate(clear boolean, co_node_name cstring)

Description: Clears invalid backend threads on a CN. (These backend threads hold invalid pooler connections to standby DN.)

Return type: record

pg_nodes_memory()

Description: queries the memory usage of all nodes.

Return type: record

table_skewness(text)

Description: queries the percentage of table data among all nodes.

Parameter: Indicates that the type of the name of the to-be-queried table is text.

Return type: record

table_skewness(table_name text, column_name text[, row_num text])

Description: Queries the proportion of column data distributed on each node based on the hash distribution rule. The results are sorted based on the data volumes of the nodes.

Parameters: **table_name** indicates a table name, **column_name** indicates a column name, and **row_num** indicates that all data in the current column is returned. The default value is **0**. A value other than **0** indicates the number of data records whose statistics are sampled. (Records are randomly sampled.)

Return type: record

Example:

Distribute data by hash based on the **a** column in the **tx** table. Seven records are distributed on DN 1, two records on DN 2, and one record on DN 0.

```
SELECT * FROM table_skewness('tx','a');
seqnum | num | ratio
-----+-----+-----
1      | 7   | 70.000%
2      | 2   | 20.000%
0      | 1   | 10.000%
(3 row)
```

table_data_skewness(data_row record, locatorType "char")

Description: Calculates the bucket distribution index for the records concatenated using the columns in a specified table.

Parameters: **data_row** indicates the record concatenated using columns in the specified table. **locatorType** indicates the distribution rule. You are advised to set **locatorType** to **H**, indicating hash distribution.

Return type: smallint

Example:

Calculates the bucket distribution index based on the hash distribution rule for the records combined concatenated using column **a** in the **tx** table.

```
select a, table_data_skewness(row(a), 'H') from tx;
a | table_data_skewness
--+-----
3 | 0
6 | 2
7 | 2
4 | 1
5 | 1
(5 rows)
```

table_distribution(schemaname text, tablename text)

Description: queries the storage space occupied by a specified table on each node.

Parameter: Indicates that the types of the schema name and table name for the table to be queried are both text.

Return type: record

 NOTE

- To query for the storage distribution of a specified table by using this function, you must have the **SELECT** permission for the table.
- The performance of **table_distribution** is better than that of **table_skewness**. Especially in a large cluster with a large amount of data, **table_distribution** is recommended.
- When you use **table_distribution** and want to view the space usage, you can use **dnsize** or **(sum(dnsize) over ())** to view the percentage.

table_distribution(regclass)

Description: queries the storage space occupied by a specified table on each node.

Parameter: indicates the name or OID of the table to be queried. The table name can be defined by the schema name. Parameter type: regclass

Return type: record

 NOTE

- To query for the storage distribution of a specified table by using this function, you must have the **SELECT** permission for the table.
- The performance of **table_distribution** is better than that of **table_skewness**. Especially in a large cluster with a large amount of data, **table_distribution** is recommended.
- When you use **table_distribution** and want to view the space usage, you can use **dnsize** or **(sum(dnsize) over ())** to view the percentage.

table_distribution()

Description: queries the storage distribution of all tables in the current database.

Return type: record

 NOTE

- This function involves the query for information about all tables in the database. To execute this function, you must have the administrator rights or rights of the preset role **gs_role_read_all_stats**.
- Based on the **table_distribution()** function, DWS provides the **PGXC_GET_TABLE_SKEWNESS** view as an alternative way to query for data skew. You are advised to use this view when the number of tables in the database is less than 10000.

gs_table_distribution(schemaname text, tablename text)

Description: queries the storage space occupied by a specified table on each node.

Return type: record

Table 7-28 Fields returned by the **gs_table_distribution(schemaname text, tablename text)** function

Name	Type	Description
schemaname	name	Schema name

Name	Type	Description
tablename	name	Table name
relkind	character	Type. • i: index • r: table
relpersistence	character	Type. • t: local temporary table • g: global temporary table • u: unlogged table • p: common table
nodename	name	Node name
dnsize	bigint	Storage space of the table on the node, in bytes.
sessionid	bigint	For a global temporary table, the session ID of each session is displayed.

 NOTE

- To query for the storage distribution of a specified table by using this function, you must have the **SELECT** permission for the table.
- This function is based on the physical file storage space records in the **PG_RELFILENODE_SIZE** system catalog. Ensure that the GUC parameters **use_workload_manager** and **enable_perm_space** are enabled.
- The **gs_table_distribution** function performs slower than the **table_distribution** function when querying a single table. However, when querying the entire database, the **gs_table_distribution** function shows significantly better performance. For large clusters with a substantial amount of data, it is recommended to use the **gs_table_distribution** function to query all tables in the database.
- For a global temporary table, the distribution of each session in the storage space is displayed. The **sessionid** column is used to distinguish sessions.

gs_table_distribution()

Description: quickly queries the storage distribution of all tables in the current database.

Return type: record

Table 7-29 Fields returned by the **gs_table_distribution()** function

Name	Type	Description
schemaname	name	Schema name
tablename	name	Table name

Name	Type	Description
relkind	character	Type of the table. i : index; r : table.
relpersistence	character	Type. • t: local temporary table • g: global temporary table • u: unlogged table • p: common table
nodename	name	Node name
dnsize	bigint	Storage space of the table on the node, in bytes.
sessionid	bigint	For a global temporary table, the session ID of each session is displayed.

 NOTE

- To query for the storage distribution of a specified table by using this function, you must have the **SELECT** permission for the table.
- This function is based on the physical file storage space records in the **PG_RELFILENODE_SIZE** system catalog. Ensure that the GUC parameters **use_workload_manager** and **enable_perm_space** are enabled.
- The **gs_table_distribution** function performs slower than the **table_distribution** function when querying a single table. However, when querying the entire database, the **gs_table_distribution** function shows significantly better performance. For large clusters with a substantial amount of data, it is recommended to use the **gs_table_distribution** function to query all tables in the database.
- Based on the **gs_table_distribution()** function, DWS 8.2.1 and later versions provide the **PGXC_WLM_TABLE_DISTRIBUTION_SKEWNESS** view for data skew query. You are advised to use this view when the number of tables in the database is small (less than 10,000).

check_foreign_key_constraint(schema.table)

Description: Checks whether data in a foreign key table meets foreign key constraints. This is supported only by clusters of version 9.1.0.200 or later.

Return type: text

Example:

```
set info_constraint_options = 'foreign_key';

CREATE TABLE ptt1 (a int,b int,e varchar(50),f varchar(50),PRIMARY KEY (a,b)) DISTRIBUTE BY HASH(a,b);
CREATE TABLE tt2 (a int , b int, c int, d int) DISTRIBUTE BY HASH(a,b);

select check_foreign_key_constraint('table_constraints.tt2');
check_foreign_key_constraint
-----
tt2 table has no foreign key constraint

ALTER TABLE tt2 ADD CONSTRAINT fk_tt1_a FOREIGN KEY (a,b) REFERENCES ptt1(a,b);
```

```
select check_foreign_key_constraint('table_constraints.tt2');
check_foreign_key_constraint
-----
tt2 satisfied all Foreign key constraints
```

plan_seed()

Description: Obtains the seed value of the previous query statement (internal use).

Return type: integer

pg_stat_get_env()

Description: Obtains the environment variable information about the current node.

Return type: record

pg_stat_get_thread()

Description: Provides information about the status of all threads under the current node.

Return type: record

pgxc_get_os_threads()

Description: Provides information about the status of threads under all normal nodes in a cluster.

Return type: record

pg_stat_get_sql_count()

Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE/MERGE INTO** statements executed by all users on the current node, response time, and the number of DDL, DML, and DCL statements.

Return type: record

pgxc_get_sql_count()

Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE/MERGE INTO** statements executed by all users on all nodes of the current cluster, response time, and the number of DDL, DML, and DCL statements.

Return type: record

pgxc_get_workload_sql_count()

Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE** statements executed in all workload Cgroup on all CNs of the current cluster and the number of DDL, DML, and DCL statements.

Return type: record

pgxc_get_workload_sql_elapse_time()

Description: Provides statistics on response time of **SELECT/UPDATE/INSERT/DELETE** statements executed in all workload Cgroup on all CNs of the current cluster.

Return type: record

get_instr_unique_sql()

Description: Provides information about Unique SQL statistics collected on the current node. If the node is a CN, the system returns the complete information about the Unique SQL statistics collected on the CN. That is, the system collects and summarizes the information about the Unique SQL statistics on other CNs and DNs. If the node is a DN, the Unique SQL statistics on the DN is returned. For details, see the **GS_INSTR_UNIQUE_SQL** view.

Return type: record

reset_instr_unique_sql(cstring, cstring, INT8)

Description: Clears collected Unique SQL statistics. The input parameters are described as follows:

- **GLOBAL/LOCAL**: Data is cleared from all nodes or the current node.
- **ALL/BY_USERID/BY_CNID/BY_GUC**: **ALL** indicates that all data is cleared. **BY_USERID/BY_CNID** indicates that data is cleared by **USERID** or **CNID**. **BY_GUC** indicates that the clearance operation is caused by the decrease of the value of the GUC parameter **instr_unique_sql_count**.
- The third parameter corresponds to the second parameter. The parameter is invalid for **ALL/BY_GUC**.

Return type: bool

pgxc_get_instr_unique_sql()

Description: Provides complete information about Unique SQL statistics collected on all CNs in a cluster. This function can be executed only on CNs.

Return type: record

get_instr_unique_sql_remote_cns()

Description: Provides complete information about Unique SQL statements collected on all CNs in the cluster, except the CN on which the function is being executed. This function can be executed only on CNs.

Return type: record

pgxc_get_node_env()

Description: Provides the environment variable information about all nodes in a cluster.

Return type: record

gs_switch_relfilenode()

Description: Exchanges meta information of two tables or partitions. (This is only used for the redistribution tool. An error message is displayed when the function is directly used by users).

Return type: integer

copy_error_log_create()

Description: Creates the error table (**public.pgxc_copy_error_log**) required for creating the **COPY FROM** error tolerance mechanism.

Return type: boolean

NOTE

- This function attempts to create the **public.pgxc_copy_error_log** table. For details about the table, see [Table 7-30](#).
- Create the B-tree index on the **relname** column and execute **REVOKE ALL on public.pgxc_copy_error_log FROM public** to manage permissions for the error table (the permissions are the same as those of the **COPY** statement).
- **public.pgxc_copy_error_log** is a row-store table. Therefore, this function can be executed and **COPY FROM** error tolerance is available only when row-store tables can be created in the cluster. After the GUC parameter **enable_hadoop_env** is enabled, row-based tables cannot be created in the cluster. The default value for DWS is **off**.
- Same as the error table and the **COPY** statement, the function requires **sysadmin** or higher permissions.
- If the **public.pgxc_copy_error_log** table or the **copy_error_log_relname_idx** index already exists before the function creates it, the function will report an error and roll back.

Table 7-30 Error table public.pgxc_copy_error_log

Column	Type	Description
relname	varchar	Table name in the form of <i>Schema name.Table name</i>
begintime	timestamp with time zone	Time when a data format error was reported
filename	varchar	Name of the source data file where a data format error occurs
rownum	bigint	Number of the row where a data format error occurs in a source data file
rawrecord	text	Raw record of a data format error in the source data file To prevent a field from being too long, the length of the field cannot exceed 1024 bytes.
detail	text	Error details

Column	Type	Description
columnname	varchar	Name of the column whose data format is incorrect in the data source file. Only 8.2.1.100 and later versions support this function.
errcode	varchar	Error code corresponding to the error information. The sqlstate error code is used. Only 8.2.1.100 and later versions support this function.
queryid	bigint	ID of the SQL statement for executing the Copy statement. It uniquely identifies an SQL statement. Only 8.2.1.100 and later versions support this function.

pv_compute_pool_workload()

Description: Provides the current load information about computing Node Groups on cloud.

Return type: record

pg_stat_get_status(tid, num_node_display)

Description: Queries for the blocking and waiting status of the backend threads and auxiliary threads in the current instance. For details about the returned results, see the PG_THREAD_WAIT_STATUS view. The input parameters are described as follows:

- **tid**: thread ID, which is of the bigint type. If this parameter is null, the waiting statuses of all backend threads and auxiliary threads are returned. Otherwise, only the waiting statuses of threads with the specified IDs are returned.
- **num_node_display**: integer type. Specifies the maximum number of waiting nodes displayed in the **wait_status** column for records whose waiting status is **wait node**.
 - If this parameter is left empty or set to a value less than or equal to **0**, only one waiting node is displayed.
 - If the value is greater than **20**, a maximum number of nodes can be displayed is **20**.
 - If the value is greater than **0** and less than or equal to **20**, the smaller value between **num_node_display** and the actual number of waiting nodes is displayed. Use the **SELECT * from pg_stat_get_status(NULL, 10)** query for example. If the number of waiting nodes is greater than **10**, the names of only 10 nodes are displayed randomly. If the number of waiting nodes is less than or equal to **10**, the names of all waiting nodes are displayed. If the number of waiting nodes is greater than the number of displayed nodes, the displayed node names are randomly selected.

Return type: record

pgxc_get_thread_wait_status(num_node_display)

Description: Queries for the call hierarchy between threads generated by all SQL statements on each node in a cluster, as well as the block waiting status of each thread. For details about the returned results, see the PGXC_THREAD_WAIT_STATUS view. The type and meaning of the input parameter **num_node_display** are the same as those of the **pg_stat_get_status** function.

Return type: record

pgxc_os_run_info()

Description: Obtains the running status of the operating system on each node in a cluster. For details about the returned results, see "System Catalogs > System Views > PV_OS_RUN_INFO" in the *Developer Guide*.

Return type: record

get_instr_wait_event()

Description: obtains the waiting status and events of the current instance. For details about the returned results, see "System Catalogs > System Views > GS_WAIT_EVENTS" in the *Developer Guide*. If the GUC parameter **enable_track_wait_event** is **off**, this function returns **0**.

Return type: record

pgxc_wait_events()

Description: queries statistics about waiting status and events on each node in a cluster. For details about the returned results, see "System Catalogs > System Views > PGXC_WAIT_EVENTS" in the *Developer Guide*. If the GUC parameter **enable_track_wait_event** is **off**, this function returns **0**.

Return type: record

pgxc_stat_bgwriter()

Description: queries statistics about backend write processes on each node in a cluster. For details about the returned results, see "System Catalogs > System Views > PG_STAT_BGWRITER" in the *Developer Guide*.

Return type: record

pgxc_stat_replication()

Description: queries information about the log synchronization status on each node in a cluster, such as the location where the logs are sent and received. For details about the returned results, see "System Catalogs > System Views > PG_STAT_REPLICATION" in the *Developer Guide*.

Return type: record

pgxc_replication_slots()

Description: queries the replication status on each DN in a cluster. For details about the returned results, see "System Catalogs > System Views > PG_REPLICATION_SLOTS" in the *Developer Guide*.

Return type: record

pgxc_settings()

Description: queries information about runtime parameters on each node in a cluster. For details about the returned results, see "System Catalogs > System Views > PG_SETTINGS" in the *Developer Guide*.

Return type: record

pgxc_instance_time()

Description: queries the running time statistics of each node in a cluster and the time consumed in each execution phase. For details about the returned results, see "System Catalogs > System Views > PV_INSTANCE_TIME" in the *Developer Guide*.

Return type: record

pg_stat_get_redo_stat()

Description: queries Xlog redo statistics on the current node. For details about the returned results, see "System Catalogs > System Views > PV_REDO_STAT" in the *Developer Guide*.

Return type: record

pgxc_redo_stat()

Description: queries the Xlog redo statistics of each node in a cluster. For details about the returned results, see "System Catalogs > System Views > PV_REDO_STAT" in the *Developer Guide*.

Return type: record

get_local_rel_iostat()

Description: Obtains the disk I/O statistics of the current instance. For details about the returned results, see "System Catalogs > System Views > GS_REL_IOSTAT" in the *Developer Guide*.

Return type: record

pgxc_rel_iostat()

Description: queries the disk I/O statistics on each node in a cluster. For details about the returned result, see "System Catalogs > System Views > GS_REL_IOSTAT" in the *Developer Guide*.

Return type: record

get_node_stat_reset_time()

Description: Obtains the time when statistics of the current instance were reset.

Return type: timestamptz

pgxc_node_stat_reset_time()

Description: queries the time when the statistics of each node in a cluster are reset. For details about the returned result, see "System Catalogs > System Views > GS_NODE_STAT_RESET_TIME" in the *Developer Guide*.

Return type: record

NOTE

When an instance is running, its statistics keep rising. In the following cases, the statistical values in the memory will be reset to 0:

- The instance is restarted or a cluster switchover occurs.
- The database is dropped.
- A reset operation is performed. For example, the statistics counter in the database is reset using the **pgstat_recv_resetcounter** function or the Unique SQL statements are cleared using the **reset_instr_unique_sql** function.

If any of the preceding events occurs, DWS will record the time when the statistics are reset. You can query the time using the **get_node_stat_reset_time** function.

pgxc_parallel_query(text, text)

Description: Runs a specified SQL query statement on a data instance of a specified type and returns the query result to the current CN. This function is supported in 8.1.2 or later.

The function has two parameters:

The first parameter specifies the instances on which the SQL statement is executed. Currently, the valid input parameters are **dn**, **datanode**, **cn**, **coordinator**, and **all**. Other values will cause function execution errors. **dn** and **datanode** indicate that the statement is executed on all DNs. **cn** and **coordinator** indicate that the statement is executed on all CNs. **all** indicates that the statement is executed on all CNs and DNs.

The second parameter specifies the verification of the objects queried by the SQL statement that is to be sent to a remote node for execution. User tables, distributed tables, and user-defined functions with multiple result sets are not supported.

Return type: record

 NOTE

- This function is only used by developers to efficiently collect the execution information or status of instances in a cluster. You are not advised to use it directly.
- This function contains multiple result sets, and the return data type is record. Therefore, you need to add the output column name and data type specified by the **AS** statement after the function call, as shown in the following:

```
SELECT * FROM pgxc_parallel_query('all', 'select node_name, db_name, thread_name, query_id, tid, lwtid, ptid, tlevel, smpid, wait_status, wait_event from pg_thread_wait_status') AS (node_name text, db_name text, thread_name text, query_id bigint, tid bigint, lwtid integer , ptid integer, tlevel integer , smpid integer, wait_status text, wait_event text);
```
- The data type of the output result of the SQL statement specified by the second parameter of the function must be the same as the data type specified by the **AS** statement. Otherwise, an error may be reported during execution due to type mismatch.
- The SQL statement specified by the second parameter of the function cannot trigger cross-node query. Otherwise, an error is reported.
- The SQL statement specified by the second parameter of the function can only be a **SELECT**, **UPDATE**, **DELETE**, or **INSERT** statement.
 - The **returning** statement is not supported.
 - The user who invokes the function must have the operation permission on the SQL objects.
 - For **INSERT** statements, **INSERT OVERWRITE**, **UPSERT**, and **INSERT INTO** are not supported.
 - The **UPDATE**, **DELETE**, and **INSERT** statements can be executed only by the initial user in in-place upgrade mode or by the administrator in redistribution mode. The number of records modified by the statements on each instance must be the same. Otherwise, an error will be reported during statement execution. The function outputs a column of values of the bigint type. These values indicate the number of records operated by the statement on each instance.

```
SELECT * FROM pgxc_parallel_query('cn', 'UPDATE pg_partition SET relpages = 0') AS (updated bigint);
```

create_wdr_snapshot()

Description: Creates a performance data snapshot.

Return type: text

 NOTE

- Only the database administrator **SYSADMIN** can execute this function.
- This function can be executed only on CNs. If it is executed on DN, the following message will be returned: "WDR snapshot can only be created on coordinator."
- Before executing this function, ensure that the value of **enable_wdr_snapshot** is **on**. If its value is **off**, the following message will be returned for this function: "WDR snapshot request cannot be executed, because GUC parameter 'enable_wdr_snapshot' is off."
- If the snapshot thread is not started for some reason, for example, the node is restarted, the following message will be returned for this function: "WDR snapshot request cannot be accepted, please retry later."
- If this function fails to be executed, the following message will be returned: "Cannot respond to WDR snapshot request."
- If this function is successfully executed, the following message will be returned: "WDR snapshot request has been submitted." This message indicates that the snapshot creation request has been sent to the background snapshot thread, but does not mean that the snapshot has been successfully created.

kill_snapshot(scope cstring)

Description: kills the background snapshot thread. This function sends a command to the background snapshot thread and waits for the thread to stop.

The input parameter **scope** indicates the operation scope. Its value can be **local** or **global**.

- Value **local** indicates killing the snapshot thread on the current CN.
- Value **global** indicates killing the snapshot thread on the current CN as well as those on all the other CNs in the cluster.
- If any other value is passed, error message "Scope is invalid, use "local" or "global"." is displayed.
- The input parameter can be left empty, in which case the default value **local** will be used.

Return type: none

NOTE

- Only the database administrator **SYSADMIN** can execute this function.
- This function can be executed only on CNs. If it is executed on DN, the following message will be returned: "kill_snapshot can only be executed on coordinator."
- Executing this function sends a kill signal to the background snapshot thread and waits for it to finish. If the snapshot thread is not killed within 100s, the error message "Kill snapshot thread failed" is displayed.

generate_wdr_report(begin_snap_id bigint, end_snap_id bigint, report_type cstring, report_scope cstring, node_name cstring)

Description: Creates a load analysis report.

The input parameters are described as follows:

- **begin_snap_id** and **end_snap_id**: IDs of the start and end snapshots, respectively. The IDs are of the bigint type. The value of **begin_snap_id** must be less than that of **end_snap_id**, and the time for the start and end snapshots cannot overlap. You can check whether the snapshot time overlaps by querying **select s1.end_ts < s2.start_ts from (select * from dbms_om.snapshot where snapshot_id=end_snap_id) as s2;** in the **dbms_om.snapshot** table. If **true** is returned, the snapshot time does not overlap. Otherwise, the snapshot time overlaps.
- **report_type**: report type. The value is a cstring and can be **summary**, **detail**, or **all**.
- **report_scope**: report scope. The value is a cstring and can be **cluster** or **node**.
- **node_name**: node name. The value is a cstring. If **report_scope** is **node**, the value of this parameter must be **pg_catalog**, which indicates the CN or DN name in the **node_name** column of the **pgxc_node** table.

Return type: text

 NOTE

- Only the database administrator **SYSADMIN** can execute this function.
- This function can be executed only on CNs. If it is executed on DN, the following message will be returned: "WDR report can only be created on coordinator."
- If the report is created successfully, message "Report %s has been generated" will be returned.
- The statistics cannot be reset between the time the start snapshot is taken and the time the end snapshot is taken. Otherwise, error message "Instance reset time is different" will be displayed. For details about the events that cause a statistics reset, see the [pgxc_node_stat_reset_time](#) function.

wdr_xdb_query(db_name text, snapshot_id bigint, view_name text)

Description: Queries a specified view in a specified database. The query results of some views vary depending on databases. For example, the **global_table_stat** view is used to query the statistics of a table. The results of querying this view vary because tables in different databases are different. The **wdr_xdb_query** function can access the database specified by **db_name** in the current connection and query the view specified by **view_name** in the database. The input parameters are described as follows:

- **db_name**: specifies the name of a database. The value is of the text type.
- **snapshot_id**: specifies the snapshot ID. The value is of the bigint type. For details, see section "Performance View Snapshot" in *Data Warehouse Service (DWS) Developer Guide*.
- **view_name**: specifies the name of a view. The value is of the text type. The view name must be in the following whitelist:
 - global_table_stat
 - global_table_change_stat
 - global_column_table_io_stat
 - global_row_table_io_stat

The return value type is record. The first column is **snapshot_id bigint**, and the second column is **db_name text**. The names, types, and sequences of other columns are the same as those of the views specified by **view_name**.

Example:

```
select snapshot_id, db_name, schemaname, relname, distribute_mode,
seq_scan ,seq_tuple_read ,index_scan ,index_tuple_read ,tuple_inserted
,tuple_updated ,tuple_deleted ,tuple_hot_updated ,live_tuples ,dead_tuples from
wdr_xdb_query('postgres'::text, 1, 'global_table_stat'::text) as i(snapshot_id bigint, db_name text,
schemaname name, relname name, distribute_mode char, seq_scan bigint, seq_tuple_read
bigint, index_scan bigint, index_tuple_read bigint, tuple_inserted bigint, tuple_updated bigint,
tuple_deleted bigint, tuple_hot_updated bigint, live_tuples bigint, dead_tuples bigint);
```

 NOTE

- This function is supported only in 8.1.2 or later.
- Only the database administrator **SYSADMIN** can execute this function.
- This function can be used to query only the views in the whitelist. If you use this function to query other views, the error message **Input view name is invalid.** will be displayed.

vac_fileclear_relation(oid)

Description: Forcibly clears VACUUM rewritten files in a specified column-store table to reclaim space.

Parameter: OID of a column-store table.

Return type: integer

NOTE

- Before using this function, set **colvacuum_threshold_scale_factor** and ensure that the files are cleared and space reclaimed only after the VACUUM process has rewritten the files of the specified column-store table.
- This function exclusively locks a specified column-store table.
- This function cannot be used after DDL delay is enabled.

vac_fileclear_all_relation()

Description: Forcibly clears VACUUM rewritten files in all specified column-store tables to reclaim space.

Return type: record

NOTE

This function cannot be used after DDL delay is enabled.

get_col_file_info(table_name)

Description: Queries the number of empty CU files and the total number of CU files in a specified column-store table. This function is supported only in cluster 8.2.0 and later versions.

Parameter: Name of a column-store table.

Return type: int

Fields in the returned value:

- **total_file_num int**: total number of CU files. The value ranges from **-1** to **INT_MAX**. **-1** indicates a failure, which could be caused by unsupported table types. The values in the range **0** to **INT_MAX** indicates the total number of files.
- **empty_file_num int**: number of empty CU files. The value ranges from **-1** to **INT_MAX**. **-1** indicates a failure, which could be caused by unsupported table types. The values in the range **0** to **INT_MAX** indicates the total number of empty files.

Example:

```
call get_col_file_info('t4');
total_file_num | empty_file_num
-----+-----
10 | 7
(1 row)
```

get_all_col_file_info()

Description: Queries the number of empty CU files and the total number of CU files in all column-store tables. This function is supported only in cluster 8.2.0 and later versions.

Return type: record

Fields in the returned value:

- **space_name text**: schema to which the column-store tables belong
- **table_name text**: name of a column-store table
- **total_file_num int**: total number of CU files. The value ranges from **-1** to **INT_MAX**. **-1** indicates a failure, which could be caused by unsupported table types. The values in the range **0** to **INT_MAX** indicates the total number of files.
- **empty_file_num int**: number of empty CU files. The value ranges from **-1** to **INT_MAX**. **-1** indicates a failure, which could be caused by unsupported table types. The values in the range **0** to **INT_MAX** indicates the total number of empty files.

Example:

```
call get_all_col_file_info();
space_name | table_name | total_file_num | empty_file_num
-----+-----+-----+-----
public    | t4        | 10             | 7
public    | t2        | 1              | 1
public    | t1        | 3              | 0
(3 rows)
```

pg_get_bucket_epoch(relid)

Description: Queries the **epoch** value of the **v3** table on the current node. To print the **epoch** values of all subpartitions in a partitioned table, you must provide the OID of the parent table. This function can be executed only in VW write scenarios. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Fields in the returned value:

- **relfilenode oid**: column-store **relfilenode** of a non-partitioned table. For a partitioned table, the value is **0**.
- **partfilenode oid**: **relfilenode** of a column-store partitioned table. For a non-partitioned table, the value is **0**.
- **epoch xid**: **epoch** value corresponding to the table on the current node.

Example:

```
select * from pg_get_bucket_epoch(2147483732);
relfilenode | partfilenode | epoch
-----+-----+-----
2137383742 | 0            | 1000001
(1 row)
```

pg_get_bucket_epoch(tablespaceid, databaseid, relfilenode)

Description: Queries the **epoch** value of the **v3** table on the current node. Enter the OID of the tablespace where the **v3** table is located, the OID of the database,

and the **relfilenode** corresponding to the table. This function can be executed in both VW write and read scenarios. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: xid

Fields in the returned value:

epoch xid: **epoch** value corresponding to the table on the current node.

Example:

```
select * from pg_get_bucket_epoch(2147483729, 16792, 2147483732);
epoch
-----
1000001
(1 row)
```

pgxc_get_bucket_epoch(tablespaceid, databaseid, relfilenode)

Description: Queries the **epoch** values of the **v3** table on all DN. Enter the OID of the tablespace where the **v3** table is located, the OID of the database, and the **relfilenode** corresponding to the table. This function can be executed in both VW write and read scenarios. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Fields in the returned value:

- **nodename text:** node name
- **epoch xid:** **epoch** value corresponding to the table on the current node.

Example:

```
select * from pgxc_get_bucket_epoch(2147483729, 16792, 2147483732);
nodename | epoch
-----+-----
datanode3 | 1000001
datanode1 |      0
datanode2 |      0
(1 row)
```

current_temp_schema()

Description: Returns the temporary schema of the current session. If the current session does not have a temporary schema, null is returned. This function is supported by clusters of version 8.2.1.220 or later.

Return type: name

Example:

```
select current_temp_schema();
current_temp_schema
-----
(1 row)
create temp table tmp1(a int,b int) distribute by hash(a);
CREATE TABLE
select current_temp_schema();
current_temp_schema
-----
```

```
pg_temp_coordinator1_2_2_140718823485088
(1 row)
```

get_volatile_pg_class()

Description: Obtains the **pg_class** metadata related to all volatile temporary tables in the current session. This parameter is supported by version 8.2.0 or later clusters.

Return type: record

Fields in the returned value:

- **oid**: OID of the volatile temporary table.
- Other fields: same as the fields (excluding hidden fields) in the **pg_class** **system** catalog.

get_volatile_pg_class(relname text)

Description: Obtains the **pg_class** metadata related to a specified volatile temporary table in the current session. This parameter is supported by version 8.2.0 or later clusters.

Parameter: name of the **volatile** temporary table in the current session.

Return type: record

Fields in the returned value:

- **oid**: OID of the volatile temporary table.
- Other fields: same as the fields (excluding hidden fields) in the **pg_class** **system** catalog.

Example:

```
SELECT * FROM get_volatile_pg_class('tx1');
 oid | relname | relnamespace | reltype | reloftype | relowner | relam | relfilenode | reltablespace | relpages |
 reltuples | relallvisible | reltoastrelid | reltoastidxid | reldeltarelid |
 reldeltaid | relcudescrid | relcudescidx | relhasindex | relisshared | relpersistence | relkind | relnatts |
 relchecks | relhasoids | relhaspkey | relhasrules | relhastriggers | relhassub
 class | relcmps | relhasclusterkey | relrowmovement | parttype | relfrozenxid | relacl |
 reloptions | relreplident | relfrozenxid64
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
16772 | tx1 | 16770 | 16774 | 0 | 10 | 0 | 16772 | 1665 | 0 | 0
| 0 | 16775 | 0 | 0 |
| 0 | 0 | 0 | f | f | v | r | 2 | 0 | f | f |
f | f | f | f | n | 11815 | | {orientation=row,compression=no} |
d | 11815
(1 row)
```

get_volatile_pg_attribute()

Description: Obtains the **pg_attribute** metadata related to all volatile temporary tables in the current session. This parameter is supported by version 8.2.0 or later clusters.

Return type: record

Fields in the returned value:

- **oid**: OID of the column.
- Other fields: same as the fields (excluding hidden fields) in the **pg_attribute** catalog.

get_volatile_pg_attribute(relname text, attrname text)

Description: Obtains the **pg_attribute** metadata related to a specified volatile temporary table in the current session. This parameter is supported by version 8.2.0 or later clusters.

Parameter:

- **relname**: table name (must be in the current session).
- **attrname**: column name.

Return type: record

Fields in the returned value:

- **oid**: OID of the column.
- Other fields: same as the fields (excluding hidden fields) in the **pg_attribute** catalog.

Example:

```
SELECT * FROM get_volatile_pg_attribute('tx1', 'b');
 attrelid | attname | atttypid | attstattarget | attlen | attnum | attndims | attcacheoff | atttypmod | attbyval |
 attstorage | attalign | attnotnull | atthasdef | attisdropped | attislocal |
 attcmprmode | attinhcount | attcollation | attacl | attoptions | attfdwoptions | attinitdefval | attkvtype
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 16772 | b       | f        |      25      |    -1   |    -1   |     2    |     0       |    -1     |    -1    | f         | x         | i         |
127   |         |          |      0       |    100  |         |         |         |         |         | 0         |          |          |
(1 row)
```

pg_get_publication_tables(pubname text)

Description: Returns the relid list of tables to be published based on the publication name. This function is supported by version 8.2.0.100 or later clusters.

Parameter: **pubname**

Return type: set of OID

Example:

```
SELECT * FROM pg_get_publication_tables('mypub');
 relid
-----
 16757
 16776
(2 rows)
```

pg_relation_is_publishable(relname regclass)

Description: Checks whether a table can be published. This function is supported by version 8.2.0.100 or later clusters.

Parameter: **relname**

Return type: Boolean

Example:

```
SELECT * FROM pg_relation_is_publishable('t1');
pg_relation_is_publishable
-----
t
(1 row)
```

get_col_cu_info(schema_name text, table_name text, row_count int8, dirty_percent int8)

Description: Queries the CU information of a column-store table. The CU information of each partition is collected separately. This function is supported by version 8.2.0.100 or later clusters.

Parameters: schema name (mandatory), table name (mandatory), threshold for the number of rows in a small CU (optional, 200 by default, ranging from 1 to 60000, and percentage threshold for deleting dirty CUs (optional, 70 by default, ranging from 1 to 100)

Return type: record

Fields in the returned value:

node_name: DN name.

part_name: partition name. This column is empty for a common table.

zero_size_cu_count: number of CUs whose **cuSize** is **0** and number of rows is less than or equal to **row_count**.

small_cu_count: number of CUs whose **cuSize** is **ALIGNOF_CUSIZE(8192)** and number of rows is less than or equal to **row_count**.

dirty_cu_count: number of CUs whose deadtupe percentage exceeds **dirty_percent** due to deletion.

total_cu_count: total number of CUs.

small_cu_size: total size of 8 KB CUs.

total_cu_size: total CU size.

Example:

```
SELECT * FROM get_col_cu_info('public','hs_part');
node_name | part_name | zero_size_cu_count | small_cu_count | dirty_cu_count | total_cu_count |
small_cu_size | total_cu_size
-----+-----+-----+-----+-----+-----+-----+-----
dn_1      | p1        | 3 | 0 | 0 | 3 | 0 bytes | 0 bytes
dn_1      | p2        | 3 | 0 | 0 | 3 | 0 bytes | 0 bytes
dn_1      | p3        | 3 | 0 | 0 | 3 | 0 bytes | 0 bytes
(3 rows)
```

```
SELECT * FROM get_col_cu_info('public','hs_part', 200, 90);
node_name | part_name | zero_size_cu_count | small_cu_count | dirty_cu_count | total_cu_count |
small_cu_size | total_cu_size
-----+-----+-----+-----+-----+-----+-----+-----
dn_1      | p1       |          3         |          0      |          0      |          3 | 0 bytes | 0 bytes
dn_1      | p2       |          3         |          0      |          0      |          3 | 0 bytes | 0 bytes
dn_1      | p3       |          3         |          0      |          0      |          3 | 0 bytes | 0 bytes
(3 rows)
```

get_col_file_vacuum_info(schema_name text, table_name text, force_get_rewritten_file_num bool)

Description: Queries the vacuum information of a column-store table. The vacuum information of each partition is collected separately. This function is supported by version 8.2.0.100 or later clusters.

Parameters: schema name (mandatory), table name (mandatory), and whether to forcibly obtain the precise number of files that can be cleared (mandatory, **false** by default)

Return type: record

Fields in the returned value:

node_name: DN name.

part_name: partition name. This column is empty for a common table.

total_file_num: total number of CU files.

rewritable_file_num: number of files that can be rewritten but have not been rewritten.

rewritten_file_num: number of files that have been rewritten but have not been cleared. The value is obtained from data in the memory. If the memory data is lost due to reasons such as restart, you can set **force_get_rewritten_file_num=true** to forcibly obtain the accurate number of files that can be cleared.

empty_file_num: number of cleared files.

Example:

```
SELECT * FROM get_col_file_vacuum_info('public','pa',false);
node_name | part_name | total_file_num | rewritable_file_num | rewritten_file_num | empty_file_num
-----+-----+-----+-----+-----+-----
datanode1 | pa1       |          1     |          0          |          0          |          0
datanode1 | pa2       |          1     |          0          |          0          |          0
datanode2 | pa1       |          1     |          0          |          0          |          0
datanode2 | pa2       |          1     |          0          |          0          |          0
datanode3 | pa1       |          1     |          0          |          0          |          0
datanode3 | pa2       |          1     |          0          |          0          |          0
(6 rows)
```

get_col_file_vacuum_info(schema_name text, table_name text, colvacuum_threshold_scale_factor int)

Description: Queries the vacuum information of a column-store table. The vacuum information of each partition is collected separately. This function is supported by version 8.2.0.100 or later clusters.

Parameters: schema name (mandatory), table name (mandatory), and **colvacuum_threshold_scale_factor** (mandatory. The value range is 0 to 100, indicating the ratio of dead tuples.)

Return type: record

Return value:

node_name: DN name.

part_name: partition name. This column is empty for a common table.

total_file_num: total number of CU files.

rewritable_file_num: number of files that can be rewritten but have not been rewritten.

rewritten_file_num: number of files that have been rewritten but have not been cleared. The value is obtained from data in the memory. If the memory data is lost due to reasons such as restart, you can set **force_get_rewritten_file_num=true** to forcibly obtain the accurate number of files that can be cleared.

empty_file_num: number of cleared files.

Example:

```
SELECT * FROM get_col_file_vacuum_info('public','pa',10);
```

node_name	part_name	total_file_num	rewritable_file_num	rewritten_file_num	empty_file_num
datanode1	pa1	1	0	0	0
datanode1	pa2	1	0	0	0
datanode2	pa1	1	0	0	0
datanode2	pa2	1	0	0	0
datanode3	pa1	1	0	0	0
datanode3	pa2	1	0	0	0

(6 rows)

get_all_col_cu_info(row_count int8)

Description: Queries the CU information of all column-store tables in the database. This function is supported by version 8.2.0.100 or later clusters.

Parameter: threshold for the number of rows in a small CU (optional, **200** by default, and ranging from **1** to **60000**)

Return type: record

Fields in the returned value:

node_name: DN name.

schema_name: schema name.

table_name: table name.

zero_size_cu_count: number of CUs whose **cuSize** is **0** and number of rows is less than or equal to **row_count**.

small_cu_count: number of CUs whose **cuSize** is **ALIGNOF_CUSIZE(8192)** and number of rows is less than or equal to **row_count**.

total_cu_count: total number of CUs.

small_cu_size: total size of 8 KB CUs.

total_cu_size: total CU size.

Example:

```
SELECT * FROM get_all_col_cu_info(200);
```

node_name	schema_name	table_name	zero_size_cu_count	small_cu_count	total_cu_count	small_cu_size	total_cu_size
datanode1	public	udi_48076	5	1	6	8192 bytes	8192 bytes
datanode1	public	udi_48077	5	1	6	8192 bytes	8192 bytes
datanode2	public	udi_48076	5	1	6	8192 bytes	8192 bytes
datanode2	public	udi_48077	5	1	6	8192 bytes	8192 bytes
datanode3	public	udi_48076	5	1	6	8192 bytes	8192 bytes
datanode3	public	udi_48077	5	1	6	8192 bytes	8192 bytes

(6 rows)

get_all_col_file_vacuum_info(force_get_rewritten_file_num bool)

Description: Queries the vacuum information of all column-store tables in the database. This function is supported by version 8.2.0.100 or later clusters.

Parameter: whether to forcibly obtain the accurate number of files that can be cleared (mandatory. It can be **true** or **false**.)

Return type: record

Fields in the returned value:

node_name: DN name.

schema_name: schema name.

table_name: table name.

total_file_num: total number of CU files.

rewritable_file_num: number of files that can be rewritten but have not been rewritten.

rewritten_file_num: number of files that have been rewritten but have not been cleared. The value is obtained from data in the memory. If the memory data is lost due to reasons such as restart, you can set **force_get_rewritten_file_num=true** to forcibly obtain the accurate number of files that can be cleared.

empty_file_num: number of cleared files.

Example:

```
SELECT * FROM get_all_col_file_vacuum_info(false);
```

node_name	schema_name	table_name	total_file_num	rewritable_file_num	rewritten_file_num	empty_file_num
datanode1	public	udi_57373	2	0	0	1
datanode1	public	udi_57374	2	0	0	1
datanode2	public	udi_57373	2	0	0	1
datanode2	public	udi_57374	2	0	0	1
datanode3	public	udi_57373	2	0	0	1
datanode3	public	udi_57374	2	0	0	1

show_tsc_info()

Description: Queries the TimeStamp-Counter (TSC) information obtained from the current database node. This function is supported by version 8.2.1 or later clusters.

Return type: record

Table 7-31 Parameter

Name	Type	Description
node_name	text	Node name
tsc_mult	bigint	TSC conversion multiplier
tsc_shift	bigint	TSC conversion shifts
tsc_frequency	float8	TSC frequency.
tsc_use_frequency	boolean	Indicates whether to use the TSC frequency for time conversion.
tsc_ready	boolean	Indicates whether the TSC frequency can be used for time conversion
tsc_scalar_error_info	text	Error information about obtaining TSC conversion information
tsc_freq_error_info	text	Error information about obtaining TSC frequency information

Example:

```
SELECT * FROM show_tsc_info();
 node_name | tsc_mult | tsc_shift | tsc_frequency | tsc_use_frequency | tsc_ready |
 tsc_scalar_error_info | tsc_freq_error_info
-----+-----+-----+-----+-----+-----+-----
 coordinator1 |          |          | 2400 | t          | t          | TSC scalar is not initialized |
```

get_tsc_info()

Description: Re-obtains the TimeStamp-Counter (TSC) information of the current database node. This function is supported by version 8.2.1 or later clusters.

Return type: record

Table 7-32 show_tsc_info() return columns

Column	Type	Description
node_name	text	Node name
tsc_mult	bigint	TSC conversion multiplier
tsc_shift	bigint	TSC conversion shifts

Column	Type	Description
tsc_frequency	float8	TSC frequency
tsc_use_frequency	boolean	Indicates whether to use the TSC frequency for time conversion.
tsc_ready	boolean	Indicates whether the TSC frequency can be used for time conversion
tsc_scalar_error_info	text	Error information about obtaining TSC conversion information
tsc_freq_error_info	text	Error information about obtaining TSC frequency information

Example:

```
SELECT * FROM get_tsc_info();
 node_name | tsc_mult | tsc_shift | tsc_frequency | tsc_use_frequency | tsc_ready |
 tsc_scalar_error_info | tsc_freq_error_info
-----+-----+-----+-----+-----+-----+-----
 coordinator1 |          |          | 2400 | t          | t          | TSC scalar is not initialized |
```

test_tsc_info(time float8, loops int)

Description: Tests the accuracy of the time converted using the TimeStamp-Counter (TSC) on the current node. This function is supported by version 8.2.1 or later clusters.

The input parameters are described as follows:

- **time**: indicates the test time difference (unit: s). The test duration must be less than or equal to 60s.
- **loops**: indicates the number of tests. The value ranges from 1 to 10.

Return type: record

Fields in the returned value:

- **id**: number of cycles.
- **real_time_diff**: time difference obtained using **gettimeofday** (unit: us).
- **est_time_scalar**: time difference (unit: μ s) converted using TSC conversion information.
- **est_time_frequency**: time difference (unit: μ s) converted using the TSC frequency.

Example:

```
SELECT * FROM test_tsc_info(0.01,10);
 id | real_time_diff | est_time_scalar | est_time_frequency
-----+-----+-----+-----
 1 | 10057 | 10056.9
 2 | 10057 | 10057.4816666667
 3 | 10056 | 10055.2841666667
 4 | 10054 | 10054.4908333333
```

5	10055	10054.2875
6	10055	10054.7483333333
7	10055	10054.4725
8	10054	10054.0766666667
9	10058	10058.1016666667
10	10057	10056.3733333333

(10 rows)

pgxc_recyclebin_purge

Description: This function traverses and clears objects in the recycle bin. It is supported only by 9.1.0.200 and later cluster versions.

Return type: void

Example:

```
SELECT * FROM pgxc_recyclebin_purge();
pgxc_recyclebin_purge
-----
(1 row)
```

7.28 Backup and Restoration Control Functions

7.28.1 Backup Control Functions

Backup control functions help online backup.

pg_create_restore_point(name text)

Description: Creates a named point for performing the restore operation (restricted to system administrators).

Return type: text

Note: **pg_create_restore_point** creates a named transaction log record that can be used as a restoration target, and returns the corresponding transaction log location. The given name can then be used with **recovery_target_name** to specify the point up to which restoration will proceed. Avoid creating multiple restoration points with the same name, since restoration will stop at the first one whose name matches the restoration target.

pg_current_xlog_location()

Description: Obtains the write position of the current transaction log.

Return type: text

Note: **pg_current_xlog_location** displays the write position of the current transaction log in the same format as those of the previous functions. Read-only operations do not require rights of the system administrator.

Example:

```
SELECT pg_current_xlog_location();
pg_current_xlog_location
-----
```

```
0/4310D5D8  
(1 row)
```

pg_current_xlog_insert_location()

Description: Obtains the insert position of the current transaction log.

Return type: text

Note: **pg_current_xlog_insert_location** displays the insert position of the current transaction log. The insertion point is the logical end of the transaction log at any instant, while the write location is the end of what has been written out from the server's internal buffers. The write position is the end that can be detected externally from the server. This operation can be performed to archive only some of completed transaction log files. The insert position is mainly used for commissioning the server. Read-only operations do not require rights of the system administrator.

Example:

```
SELECT pg_current_xlog_insert_location();  
pg_current_xlog_insert_location  
-----  
0/4316AD50  
(1 row)
```

pg_start_backup(label text [, fast boolean])

Description: Starts executing online backup (restricted to system administrators or replication roles).

Return type: text

Note: **pg_start_backup** receives a user-defined backup label (usually the name of the position where the backup dump file is stored). This function writes a backup label file to the data directory of the database cluster and then returns the starting position of backed up transaction logs in text mode.

```
SELECT pg_start_backup('label_goes_here');  
pg_start_backup  
-----  
0/3000020  
(1 row)
```

pg_stop_backup()

Description: Completes online backup (restricted to system administrators or replication roles).

Return type: text

Note: **pg_stop_backup** deletes the label file created by **pg_start_backup** and creates a backup history file in the transaction log archive area. The history file includes the label given to **pg_start_backup**, the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending transaction log location. After the ending position is calculated, the insert position of the current transaction log automatically goes ahead to the next transaction log file. This way, the ended transaction log file can be immediately archived so that backup is complete.

Example:

```
SELECT pg_stop_backup();
pg_stop_backup
-----
0/4401E610
(1 row)
```

pg_switch_xlog()

Description: Switches to a new transaction log file (restricted to system administrators).

Return type: text

Note: **pg_switch_xlog** moves to the next transaction log file so that the current log file can be archived (if continuous archive is used). The return value is the ending transaction log location + 1 within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switchover, **pg_switch_xlog** will do nothing but return the start location of the transaction log file currently in use.

Example:

```
SELECT pg_switch_xlog();
pg_switch_xlog
-----
0/45017978
(1 row)
```

pg_xlogfile_name(location text)

Description: Converts the position string in a transaction log to a file name.

Return type: text

Note: **pg_xlogfile_name** extracts only the transaction log file name. If the given transaction log position is the transaction log file border, a transaction log file name will be returned for both the two functions. This is usually the desired behavior for managing transaction log archiving, since the preceding file is the last one that currently needs to be archived.

pg_xlogfile_name_offset(location text)

Description: Converts the position string in a transaction log to a file name and returns the byte offset in the file.

Return type: text, integer

Note: **pg_xlogfile_name_offset** can extract transaction log file names and byte offsets from the returned results of the preceding functions. Example:

```
SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
NOTICE: pg_stop_backup cleanup done, waiting for required WAL segments to be archived
NOTICE: pg_stop_backup complete, all required WAL segments have been archived
  file_name      | file_offset
-----+-----
000000010000000000000003 |      272
(1 row)
```

pg_xlog_location_diff(location text, location text)

Description: **pg_xlog_location_diff** calculates the difference in bytes between two transaction log locations.

Return type: numeric

pg_cbm_tracked_location()

Description: Queries for the LSN location parsed by CBM.

Return type: text

pg_cbm_get_merged_file(startLSNArg text, endLSNArg text)

Description: Combines CBM files within the specified LSN range into one and returns the name of the combined file.

Return type: text

pg_cbm_get_changed_block(startLSNArg text, endLSNArg text)

Description: Combines CBM files within the specified LSN range into a table and return records of this table.

Return type: record

Note: The table columns include the start LSN, end LSN, tablespace OID, database OID, table relfilenode, table fork number, whether the table is deleted, whether the table is created, whether the table is truncated, number of pages in the truncated table, number of modified pages, and list of No. of modified pages.

pg_cbm_recycle_file(slotName name, targetLSNArg text)

Description: Deletes the CBM files that are no longer used and returns the first LSN after the deletion. If **slotName** is empty, **targetLSNArg** is used as the recycling point. During backup and DR, you need to specify a slot name due to parallelism. Record the **targetLSNArg** value of the task to the slot, traverse all backup slots, and find the smallest LSN as the recycling point.

Return type: text

pg_cbm_force_track(targetLSNArg text,timeOut int)

Description: Forcibly executes the CBM trace to the specified Xlog position and returns the Xlog position of the actual trace end point.

Return type: text

pg_enable_delay_ddl_recycle()

Description: Enables DDL delay and returns the Xlog position of the enabling point.

Return type: text

Example:

```
SELECT pg_enable_delay_ddl_recycle();
pg_enable_delay_ddl_recycle
-----
00000000/4607B250
(1 row)
```

pg_disable_delay_ddl_recycle(barrierLSNArg text, isForce bool)

Description: Disables DDL delay and returns the Xlog range where DDL delay takes effect.

Return type: record

pg_enable_delay_xlog_recycle()

Description: Enables Xlog recycle delay.

Return type: void

Example:

```
SELECT pg_enable_delay_xlog_recycle();
pg_enable_delay_xlog_recycle
-----
(1 row)
```

pg_disable_delay_xlog_recycle()

Description: Disables Xlog recycle delay.

Return type: void

Example:

```
SELECT pg_disable_delay_xlog_recycle();
pg_disable_delay_xlog_recycle
-----
(1 row)
```

pgxc_get_senders_catchup_time()

Description: Displays the catchup information of the currently active primary/standby instance sending thread on all DNs.

Return type: record

The following information is returned:

Table 7-33 pgxc_get_senders_catchup_time() columns

Column	Type	Description
node_name	text	Node name
lwpid	integer	Current sender lwpid
local_role	text	Local role

Column	Type	Description
peer_role	text	Peer role
state	text	Current sender's replication status
sender	text	Current sender type
catchup_start	timestamp with time zone	Startup time of a catchup task
catchup_end	timestamp with time zone	End time of a catchup task
catchup_type	text	Catchup task type, full or incremental
catchup_bcm_filename	text	BCM file executed by the current catchup task
catchup_bcm_finished	integer	Number of BCM files completed by a catchup task
catchup_bcm_total	integer	Total number of BCM files to be operated by a catchup task
catchup_percent	text	Completion percentage of a catchup task
catchup_remaining_time	text	Estimated remaining time of a catchup task

pgxc_lock_for_backup()

Description: Locks the cluster before backup. Backup is performed to restore data on new nodes.

Return type: boolean

NOTE

pgxc_lock_for_backup locks a cluster before **gs_dump** or **gs_dumpall** is used to back up the cluster. After a cluster is locked, operations changing the system structure are not allowed. This function does not affect DML statements.

Example:

```
SELECT pgxc_lock_for_backup();
pgxc_lock_for_backup
-----
t
(1 row)
```

7.28.2 Restoration Control Functions

Restoration control functions provide information about the status of standby nodes. These functions can be executed during restoration or when the database is running properly.

pg_is_in_recovery()

Description: Returns **true** if restoration is still in progress.

Return type: bool

Example:

```
SELECT pg_is_in_recovery();
 pg_is_in_recovery
-----
f
(1 row)
```

pg_last_xlog_receive_location()

Description: This function gets the last transaction log location received and synchronized to disk by streaming replication. While streaming replication is in progress, this will increase monotonically. If restoration has completed, then this value will remain static at the value of the last WAL record received and synchronized to disk during restoration. If streaming replication is disabled or if not yet started, the function return will return **NULL**.

Return type: text

Example:

```
SELECT pg_last_xlog_receive_location();
 pg_last_xlog_receive_location
-----
(1 row)
```

pg_last_xlog_replay_location()

Description: Gets last transaction log location replayed during restoration. When the recovery is still in progress, the transaction log keeps increasing. If restoration has completed, then this value will remain static at the value of the last WAL record received during that restoration. When the server has been started normally without restoration, the function returns **NULL**.

Return type: text

Example:

```
SELECT pg_last_xlog_replay_location();
 pg_last_xlog_replay_location
-----
0/2B16530
(1 row)
```

pg_last_xact_replay_timestamp()

Description: Gets the timestamp of last transaction replayed during restoration. This is the time to commit a transaction or abort a WAL record on the primary

node. If no transactions have been replayed during restoration, this function will return **NULL**. Otherwise, if restoration is still in progress, this will increase monotonically. If restoration has completed, then this value will remain static at the value of the last WAL record received during that restoration. If the server normally starts without manual intervention, this function will return **NULL**.

Return type: timestamp with time zone

Restoration control functions control restoration processes. These functions may be executed only during restoration.

Example:

```
SELECT pg_last_xact_replay_timestamp();
pg_last_xact_replay_timestamp
-----
2023-01-04 07:03:08.098024+00
(1 row)
```

pg_is_xlog_replay_paused()

Description: Returns **true** if restoration is paused.

Return type: bool

pg_xlog_replay_pause()

Description: Pauses restoration immediately.

Return type: void

pg_xlog_replay_resume()

Description: Restarts restoration if it was paused.

While restoration is paused, no further database changes are applied. In hot standby mode, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until restoration is resumed.

If streaming replication is disabled, the paused state may continue indefinitely without problem. While streaming replication is in progress, WAL records will continue to be received, which will eventually fill available disk space. This progress depends on the duration of the pause, the rate of WAL generation, and available disk space.

Return type: void

pg_xlog_replay_completion()

Description: Displays the progress of xlog redo on the current DN.

Return type: record

Example:

```
SELECT * FROM pg_xlog_replay_completion();
replay_start | replay_current | replay_end | replay_percent
-----+-----+-----+-----
```

```
0/2ACAB80 | 0/2B16530 | 0/4F62B090 | 0%  
(1 row)
```

The following information is returned:

Table 7-34 pg_xlog_replay_completion() columns

Column	Type	Description
replay_start	integer	Start LSN of xlog redo
replay_current	integer	LSN of the current replay of xlog redo
replay_end	integer	Maximum LSN that requires xlog redo
replay_percent	integer	Completion percentage of xlog redo

pg_data_sync_from_dummy_completion()

Description: Displays the progress of data page file synchronization during the failover on the current DN.

Return type: record

Example:

```
SELECT * FROM pg_data_sync_from_dummy_completion();  
start_index | current_index | total_index | sync_percent  
-----+-----+-----+-----  
0 | 0 | 0 | 100%  
(1 row)
```

The following information is returned:

Table 7-35 pg_data_sync_from_dummy_completion() columns

Column	Type	Description
start_index	integer	Start LSN of data page file synchronization
current_index	integer	Current LSN of data page file synchronization
total_index	integer	Maximum LSN of data page file synchronization
sync_percent	integer	Completion percentage of data page files

gs_roach_stop_backup(backupid text)

Description: Stops a backup started by the internal backup tool GaussRoach and returns the position where the current log is inserted. This function is similar to **pg_stop_backup**, but is more lightweight.

Return type: text

gs_roach_enable_delay_ddl_recycle(backupid name)

Description: Enables DDL delay and returns the log position of the enabling point. This function is similar to **pg_enable_delay_ddl_recycle**, but is more lightweight. In addition, this function allows you to enable DDL delay for multiple backups.

Return type: text

gs_roach_disable_delay_ddl_recycle(backupid text)

Description: Disables DDL delay, returns the logs for which DDL delay takes effect, and deletes the physical files of the column-store tables that have been deleted by the user. This function is similar to **pg_enable_delay_ddl_recycle**, but is more lightweight. In addition, this function allows you to disable DDL delay for multiple backups.

Return type: record

gs_roach_switch_xlog(request_ckpt bool)

Description: Switches the currently used log segment file and returns the position of the segment log. If the value of **request_ckpt** is **true**, a full check point is triggered.

Return type: text

pg_resume_bkp_flag(backupid name)

Description: This function resumes the **delay xlog** flag from a specified backup and returns **start_backup_flag boolean**, **to_delay boolean**, **ddl_delay_recycle_ptr text**, and **rewind_time text**.

Return type: record

pg_update_reloptions_in_newcluster

Description: This function writes the storage path in the file specified by the **obs-store-path-list** parameter to the **reloptions** field in the **pg_class** table during full restoration of V3 tables to a new cluster. This function is supported only by clusters of version 9.1.0 or later.

Return type: bool

7.29 Database Object Functions

7.29.1 Database Object Size Functions

Database object size functions calculate and return the actual disk space used by database objects in bytes.

pg_column_size(any)

Description: Specifies the number of bytes used to store a particular value (possibly compressed).

Return type: integer

Note: **pg_column_size** displays the space for storing an independent data value.

```
postgres=#SELECT pg_column_size(1);
pg_column_size
-----
         4
(1 row)
```

pg_database_size(oid)

Description: Specifies the disk space used by the database with the specified OID.

Return type: bigint

pg_database_size(name)

Description: Specifies the disk space used by the database with the specified name.

Return type: bigint

Note: **pg_database_size** receives the OID or name of a database and returns the disk space used by the corresponding object.

Example:

```
postgres=#SELECT pg_database_size('gaussdb');
pg_database_size
-----
      51590112
(1 row)
```

pg_relation_size(oid)

Description: Specifies the disk space used by the table with a specified OID or index.

Return type: bigint

get_db_source_datasize()

Description: Estimates the total size of non-compressed data in the current database.

Return type: bigint

Note: (1) **ANALYZE** must be performed before this function is called. (2) Calculate the total size of non-compressed data by estimating the compression rate of column-store tables.

Example:

```
postgres=#analyze;
ANALYZE
postgres=#SELECT get_db_source_datasize();
```

```
get_db_source_datasize
-----
          35384925667
(1 row)
```

pg_relation_size(text)

Description: Specifies the disk space used by the table with a specified name or index. The table name can be schema-qualified.

Return type: bigint

pg_relation_size(relation regclass, fork text)

Description: Specifies the disk space used by the specified bifurcating tree ('main', 'fsm', or 'vm') of a certain table or index.

Return type: bigint

pg_relation_size(relation regclass)

Description: Is an abbreviation of **pg_relation_size(..., 'main')**.

Return type: bigint

Note: **pg_relation_size** returns the size in bytes of the table, index, or compressed table with the specified OID or name.

pg_partition_size(oid,oid)

Description: Specifies the disk space used by the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.

Return type: bigint

pg_partition_size(text, text)

Description: Specifies the disk space used by the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.

Return type: bigint

pg_partition_indexes_size(oid,oid)

Description: Specifies the disk space used by the index of the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.

Return type: bigint

pg_partition_indexes_size(text,text)

Description: Specifies the disk space used by the index of the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.

Return type: bigint

pg_indexes_size(regclass)

Description: Specifies the total disk space used by the index appended to the specified table.

Return type: bigint

pg_size_pretty(bigint)

Description: Converts the calculated byte size into a size readable to human beings.

Return type: text

pg_size_pretty(numeric)

Description: Converts the calculated byte size indicated by a numeral into a size readable to human beings.

Return type: text

Note: **pg_size_pretty** formats the results of other functions into a human-readable format. KB/MB/GB/TB can be used.

pg_table_size(regclass)

Description: Specifies the disk space used by the specified table, excluding indexes (but including TOAST, free space mapping, and visibility mapping).

Return type: bigint

pg_total_relation_size(oid)

Description: Specifies the disk space used by the table with a specified OID, including the index and the compressed data.

Return type: bigint

pg_total_relation_size(regclass)

Description: Specifies the total disk space used by the specified table, including all indexes and TOAST data.

Return type: bigint

pg_total_relation_size(text)

Description: This function specifies the disk space used by the table with the specified name, including the index and compressed data. The table name can be schema-qualified.

Return type: bigint

Note: **pg_total_relation_size** can specify the OID or name of a table or compressed table, and then return the data in bytes and the sizes of all related indexes and compressed tables.

pg_obs_file_size(regclass)

Description: This function obtains the CU file size, file name, and bucket ID stored on OBS by specifying the OID or name of a V3 column-store table. This function is supported only by clusters of version 9.1.0 or later.

Parameter: The input parameter can be the table OID or table name.

Return type: record

Table 7-36 Return fields

Field	Type	Description
bucketid	integer	ID of the bucket for storing CU files.
filename	text	CU file name.
size	bigint	CU file size, in bytes.

Example:

```
SELECT * FROM pg_obs_file_size('t3_col_part_dif_partition');
 bucketid | filename | size
-----+-----+-----
      9 | 23488119324673.0 | 512
     22 | 23488102760449.0 | 512
     21 | 23488119521281.0 | 512
     16 | 23488102662145.0 | 512
(4 rows)
```

pg_obs_file_size(text, text)

Description: This function specifies a partition of a column-store V3 table (primary table name or partitioned table) and returns the size, file name, and bucket ID of the CU file stored on OBS. This function is supported only by clusters of version 9.1.0 or later.

Parameter: The input parameter can be the table OID or table name. The partition name is used as the partition name.

Return type: record

Field	Type	Description
bucketid	integer	ID of the bucket for storing CU files.
filename	text	CU file name.
size	bigint	CU file size, in bytes.

7.29.2 Database Object Position Functions

pg_relation_filnode(relation regclass)

Description: Specifies the ID of a filnode with the specified relationship.

Return type: OID

Description: **pg_relation_filnode** receives the OID or name of a table, index, sequence, or compressed table, and returns the filnode number allocated to it. The filnode is the basic component of the file name used by the relationship. For most tables, the result is the same as that of **pg_class.relfilnode**. For the specified system directory, **relfilnode** is **0** and this function must be used to obtain the correct value. If a relationship that is not stored is transmitted, such as a view, this function returns **NULL**.

pg_relation_filepath(relation regclass)

Description: Specifies the name of a file path with the specified relationship.

Return type: text

Description: **pg_relation_filepath** is similar to **pg_relation_filnode**, except that **pg_relation_filepath** returns the whole file path name for the relationship (relative to the data directory **PGDATA** of the database cluster).

7.29.3 Partition Management Function

proc_add_partition (relname regclass, boundaries_interval interval)

Description: Adds partitions to a table with the automatic partition creation function enabled.

Return type: void

Note: When the function is executed, multiple partitions with time range as **boundaries_interval** are created based on the existing partition boundary until **new_part_boundary - now_time >= 29 * boundaries_interval**. Then, an extra partition is created to ensure that at least a new partition is created when the function is executed.

Example:

```
call proc_add_partition('my_schema.my_table', interval '1 day');
proc_add_partition
-----
(1 row)
```

proc_drop_partition (relname regclass, older_than interval)

Description: Deletes partitions from a table with the automatic partition deletion function enabled.

Return type: void

Note: When this function is executed, all partitions in the table are traversed and the partitions whose boundary is smaller than **now_time - older_than** are

deleted. If all partitions meet the deletion condition, the table is truncated with one partition kept.

Example:

```
call proc_drop_partition('my_schema.my_table', interval '7 day');
proc_drop_partition
-----
(1 row)
```

pg_partition_management_time(table_name, time)

Description: Modifies the execution time of the partition scheduling task that automatically adds or removes partitions. The default time is when the auto-increment partition table was created. This function is supported only by 8.3.0 and later versions.

Return value: **SUCCESS**, which indicates that the time is successfully modified.

Parameter:

- **table_name** indicates the name of the auto-increment partitioned table. The type is Name.
- **time** indicates the time when the data switching task is scheduled. The type is Time.

Example:

```
call pg_partition_management_time('partition_management_time','3:00');
pg_partition_management_time
-----
SUCCESS
(1 row)
```

7.29.4 Collation Version Function

pg_collation_actual_version (oid)

Description: Returns the actual version of the collation object currently installed in the operating system. Currently, this parameter is valid only for case_insensitive collations.

Return type: text

Example:

```
SELECT oid FROM pg_collation WHERE collname = 'case_insensitive';
oid
-----
3300
(1 row)

SELECT pg_collation_actual_version(3300);
pg_collation_actual_version
-----
153.14
(1 row)
```

7.29.5 Hot and Cold Table Functions

pg_obs_cold_refresh_time(table_name, time)

Description: Modifies the time when cold data in a hot and cold table is migrated to OBS. The default value is 00:00 every day.

table_name indicates the name of the hot and cold table, and the type is Name.
time indicates the time when the data switchover task is scheduled, and the type is Time.

Return value: **SUCCESS**, which indicates that the time is successfully modified.

Example:

```
SELECT * FROM pg_obs_cold_refresh_time('lifecycle_table', '06:30:00');
pg_obs_cold_refresh_time
-----
SUCCESS
(1 row)
```

pg_refresh_storage()

Description: Switches hot data to cold data on all hot and cold tables (in OBS).

Return type: int

Columns in the returned value:

- **success_count int**: indicates the number of tables that are successfully switched.
- **failed_count int**: indicates the number of tables that fail to be switched.

Example:

```
SELECT * FROM pg_refresh_storage();
success_count | failed_count
-----+-----
1 | 0
(1 row)
```

pg_lifecycle_table_data_distribute(table_name)

Description: Views the data distribution of a cold and hot table.

table_name indicates the table name and cannot be left blank.

Return value: record

Example: Multiple records are generated based on the number of nodes. The following example shows the data distribution in the **w1** table when there is only one DN node.

```
SELECT * FROM pg_catalog.pg_lifecycle_table_data_distribute('w1');
schemaname | tablename | nodename | hotpartition | coldpartition | switchablepartition | hotdatasize | colddatasize | switchabledatasize
-----+-----+-----+-----+-----+-----+-----+-----+-----
public | w1 | dn_1 | p2 | p1 | | 80 KB | 0 bytes | 0 bytes
(1 row)
```

pg_lifecycle_node_data_distribute()

Description: Views the data distribution of all hot and cold tables.

Return value: record

Example: There are two cold and hot tables in the database. The data distribution is as follows:

```
SELECT * FROM pg_catalog.pg_lifecycle_node_data_distribute();
 schemaname | tablename | nodename | hotpartition | coldpartition | switchablepartition | hotdatasize |
colddatasize | switchabledatasize
-----+-----+-----+-----+-----+-----+-----+-----
public      | w1        | dn_1     | p2           | p1           |                     | 81920      |
public      | w2        | dn_1     | p2           | p1           |                     | 81920      |
(2 rows)
```

refresh_hot_storage(relname text)

Description: Flushes all partition data of a specified cold and hot table to OBS. The returned value is the number of switched partitions. This function is supported only in 8.2.1.100 or later.

Parameter:

relname: table name (name of a cold and hot table. If the name of a non-cold and hot table is used, no error is reported and **0** is returned.)

Return type: integer

```
SELECT refresh_hot_storage('multi_temper_table');
 refresh_hot_storage
-----
4
(1 row)
```

refresh_hot_storage(relname text, partname text)

Description: Synchronizes partition data of a specified hot and cold table to OBS. The returned value is the number of switched partitions. This function is supported only in 8.2.1.100 or later.

Parameter:

- **relname:** table name (name of a cold and hot table. If the name of a non-cold and hot table is used, no error is reported and **0** is returned.)
- **partname:** partition name (specifies a partition name of a cold and hot table)

Return type: integer

```
SELECT refresh_hot_storage('multi_temper_table','p1');
 refresh_hot_storage
-----
1
(1 row)
```

reload_cold_partition(relname text)

Description: This function converts all cold partitions to hot ones. The returned value is the number of switched partitions. This function is supported only by 8.3.0 and later versions.

Parameter:

relname: table name (name of a cold and hot table. If the name of a non-cold and hot table is used, no error is reported and **0** is returned.)

Return type: integer

```
SELECT reload_cold_partition('multi_temper_table');
reload_cold_partition
-----
4
(1 row)
```

reload_cold_partition(relname text, partname text)

Description: This function converts specific cold partitions to hot ones. The returned value is the number of switched partitions. This function is supported only by 8.3.0 and later versions.

Parameter:

- **relname:** table name (name of a cold and hot table. If the name of a non-cold and hot table is used, no error is reported and **0** is returned.)
- **partname:** partition name (specifies a partition name of a cold and hot table)

Return type: integer

```
SELECT reload_cold_partition('multi_temper_table','p1');
reload_cold_partition
-----
1
(1 row)
```

7.30 Residual File Management Functions

7.30.1 Functions for Obtaining the Residual File List

pg_get_residualfiles()

Description: Obtains all residual file records of the current node. This function is an instance-level function and is irrelevant to the current database. It can run on any instance.

Return type: record

The following information is displayed:

Table 7-37 pg_get_residualfiles () return fields

Column	Type	Description
isverified	bool	Verified or not
isdeleted	bool	Deleted or not
dbname	text	Database name

Column	Type	Description
residualfile	text	Data file path
filepath	text	Residual file path
notes	text	Notes

Example:

```
SELECT * FROM pg_get_residualfiles();
isverified | isdeleted | dbname | residualfile | filepath | notes
-----+-----+-----+-----+-----+-----
f          | f         | db2    | base/49155/114691 | pgrf_20200908160211441546 |
f          | f         | db2    | base/49155/114694 | pgrf_20200908160211441546 |
f          | f         | db2    | base/49155/114696 | pgrf_20200908160211441546 |
(3 rows)
```

pgxc_get_residualfiles()

Description: Unified CN query function of pg_get_residualfiles() This function is a cluster-level function and is irrelevant to the current database. It runs on CNs.

Return type: record

The following information is displayed:

Table 7-38 pgxc_get_residualfiles () return fields

Column	Type	Description
nodename	text	Node name
isverified	bool	Verified or not
isdeleted	bool	Deleted or not
dbname	text	Database name
residualfile	text	Data file path
filepath	text	Residual file path
notes	text	Notes

Example:

```
SELECT * FROM pgxc_get_residualfiles();
nodename | isverified | isdeleted | dbname | residualfile | filepath | notes
-----+-----+-----+-----+-----+-----+-----
cn_5001  | f         | f         | postgres | base/15092/32803 | pgrf_20200910170129360401 |
dn_6001_6002 | f         | f         | db2    | base/49155/114691 | pgrf_20200908160211441546 |
dn_6001_6002 | f         | f         | db2    | base/49155/114694 | pgrf_20200908160211441546 |
dn_6001_6002 | f         | f         | db2    | base/49155/114696 | pgrf_20200908160211441546 |
(4 rows)
```

7.30.2 Functions for Scanning Residual Files

Only cluster 8.3.0 and later versions support the function of scanning residual files. All functions described in this section can be invoked only by the **sysadmin** role.

pg_scan_residualfiles()

Description: This function scans all residual file records in the database connected to the current node. It is an instance-level function and is related to the current database, and it can run on any instance.

- When it is executed on a CN, it scans the database of the CN and OBS for residual files.
- When it is executed on a DN, it scans the database of the DN for residual files.

Return type: record

The following information is displayed:

Table 7-39 Return fields

Field	Type	Description
pgscrfl	text	Local path of the metadata file that records residual file information.

Example:

```
SELECT * FROM pg_scan_residualfiles();
      pgscrfl
-----
pg_residualfiles/pgscrfl_meta_1663_16323_20231027143716005244
pg_residualfiles/pgscrfl_meta_2147484281_16323_20231027143716005713
(2 rows)
```

NOTE

The restrictions on using this function are as follows:

- Only the current database can be scanned for residual files. Currently, only the default tablespace, OBS tablespace, and user-defined tablespace are scanned.
- This function is a heavy-load operation and cannot be used concurrently on a single node. You should avoid using this function when the service is overloaded or the resource is scarce.
- Only hot partitions in cold-hot-partition-separated tables are supported. Cold partitions are not supported.
- Do not use this function during the upgrade observation period.
- Residual files in the deleted database cannot be scanned.
- Inconsistent cluster metadata information can cause this function incorrectly scan valid data files.

pgxc_scan_residualfiles(query_flag)

Description: Unified CN execution function of **pg_scan_residualfiles()** This function is a cluster-level function and is related to the current database. It runs on CNs.

Parameter description: **query_flag**. The parameter type is int, indicating the execution range. **1** indicates the CN, **2** indicates the primary DN, and **4** indicates the standby DN. The query union set can be obtained through the OR operation. For example, **1|2=3** indicates the CN and primary DN, and **1|2|4=7** indicates the CN, primary DN, and standby DN. The default value is **7**.

Return type: record

The command output is as follows.

Table 7-40 Return fields

Field	Type	Description
nodename	name	Node name.
instance_id	text	Instance name.
pgscrif	text	Local path of the metadata file that records residual file information.

Example:

```
SELECT * FROM pgxc_scan_residualfiles();
 node_name | instance_id | pgscrif
-----+-----+-----
dn_6001_6002 | dn_6001 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144103839354
dn_6001_6002 | dn_6001 | pg_residualfiles/pgscrif_meta_2147484281_16323_20231027144103839826
cn_5001 | cn_5001 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144103946217
dn_6007_6008 | dn_6008 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144104171311
(4 rows)

SELECT * FROM pgxc_scan_residualfiles(1);
 node_name | instance_id | pgscrif
-----+-----+-----
cn_5001 | cn_5001 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144421861628
(1 row)

SELECT * FROM pgxc_scan_residualfiles(2);
 node_name | instance_id | pgscrif
-----+-----+-----
dn_6001_6002 | dn_6001 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144424210395
dn_6001_6002 | dn_6001 | pg_residualfiles/pgscrif_meta_2147484281_16323_20231027144424210855
(2 rows)

SELECT * FROM pgxc_scan_residualfiles(4);
 node_name | instance_id | pgscrif
-----+-----+-----
dn_6007_6008 | dn_6008 | pg_residualfiles/pgscrif_meta_1663_16323_20231027144427492060
(1 row)
```

 NOTE

The restrictions on using this function are as follows:

- Only the current database can be scanned for residual files. Currently, only the default tablespace, OBS tablespace, and user-defined tablespace are scanned.
- This function is a heavy-load operation and cannot be used concurrently on a single node. You should avoid using this function when the service is overloaded or the resource is scarce.
- Only hot partitions in cold-hot-partition-separated tables are supported. Cold partitions are not supported.
- Do not use this function during the upgrade observation period.
- Residual files in the deleted database cannot be scanned.
- Inconsistent cluster metadata information can cause this function incorrectly scan valid data files.

pg_get_scan_residualfiles()

Description: Obtains all residual file records scanned on the current node. This function is an instance-level function and is irrelevant to the current database. It can run on any instance.

Return type: record

The command output is as follows.

Table 7-41 Return fields

Field	Type	Description
handled	bool	Whether residual files have been processed, moved, or modified.
dbname	text	Database name
residualfile	text	Path of the residual file
size	bigint	File size, in bytes. For residual files in the OBS path, the value of this parameter is 0 .
inode	bigint	Inode in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is 0 .
atime	timestampz	Access time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.
mtime	timestampz	Modify time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.
ctime	timestampz	Change time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.

Field	Type	Description
filepath	text	Local path of the metadata file that records residual file information, corresponding to the pgscrif_meta file.
notes	text	Notes

pgxc_get_scan_residualfiles(query_flag)

Description: Unified CN execution function of **pg_get_scan_residualfiles()** This function is a cluster-level function and is irrelevant to the current database. It runs on CNs.

The **query_flag** parameter is of the int type and indicates the execution range. **1** indicates the CN, **2** indicates the primary DN, and **4** indicates the standby DN. The query union set can be obtained through the OR operation. For example, **1|2=3** indicates the CN and primary DN, and **1|2|4=7** indicates the CN, primary DN, and standby DN. The default value is **7**.

Return type: record

The command output is as follows.

Table 7-42 Return fields

Field	Type	Description
nodename	name	Node name
instance_id	text	Instance name
handled	bool	Whether residual files have been processed, moved, or modified.
dbname	text	Database name
residualfile	text	Path of the residual file
size	bigint	File size, in bytes. For residual files in the OBS path, the value of this parameter is 0 .
inode	bigint	Inode in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is 0 .
atime	timestampz	Access time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.
mtime	timestampz	Modify time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.

Field	Type	Description
ctime	timestampz	Change time in the stat information of the residual file. For residual files in the OBS path, the value of this parameter is empty.
filepath	text	Local path of the metadata file that records residual file information, corresponding to the pgscrfl_meta file.
notes	text	Notes

pg_archive_scan_residualfiles()

Description: Archives all residual file records scanned on the database connected to the current node. This function is an instance-level function and is related to the current database. It can run on any instance.

Return type: record

The command output is as follows.

Table 7-43 Return fields

Field	Type	Description
archive	text	Path of the local folder after archiving. Residual files in the OBS path are archived in the corresponding OBS database directory.
count	bigint	Number of archived residual files.
size	bigint	Total size of archived residual files, in bytes.

Example:

```
SELECT * FROM pg_archive_scan_residualfiles();
      archive      | count | size
-----+-----+-----
pg_residualfiles/archive/pgscrfl_archive_20231027144613791801 |    4 | 81920
(1 row)
```

 NOTE

- This function can archive only the residual files recorded in the current login database. During archiving, the residual files are verified. The verification results are as follows:
 - Verification passed: After the verification is passed, the residual files are archived and marked as handled.
 - Verification failed: If the verification fails, the residual files are not archived and marked as handled.
 - Verification delayed: If the verification is delayed, it indicates that verification is not possible at the moment, and archiving is skipped. The verification delay is related to the transaction completion status and the standby node's redo progress.
- The actual archive directory and the corresponding tablespace are in the same file system. Deleting the tablespace also deletes the archived residual files.
- This function cannot be used after DDL delay is enabled.
- This function is a heavy-load operation and cannot be used concurrently on a single node. You should avoid using this function when the service is overloaded or the resource is scarce.

pgxc_archive_scan_residualfiles(query_flag)

Description: Unified CN execution function of **pg_archive_scan_residualfiles()**

This function is a cluster-level function and is related to the current database. It runs on CNs.

The **query_flag** parameter is of the int type and indicates the execution range. **1** indicates the CN, **2** indicates the primary DN, and **4** indicates the standby DN. The query union set can be obtained through the OR operation. For example, **1|2=3** indicates the CN and primary DN, and **1|2|4=7** indicates the CN, primary DN, and standby DN. The default value is **7**.

Return type: record

The command output is as follows.

Table 7-44 Return fields

Field	Type	Description
nodename	name	Node name.
instance_id	text	Instance name.
archive	text	Path of the local folder after archiving. Residual files in the OBS path are archived in the corresponding OBS database directory.
count	bigint	Number of archived residual files.
size	bigint	Total size of archived residual files, in bytes.

Example:

```
SELECT * FROM pgxc_archive_scan_residualfiles();
```

```
node_name | instance_id | archive | count | size
```

```
cn_5001 | cn_5001 | pg_residualfiles/archive/pgscrif_archive_20231027145050896440 | 1 | 40960  
dn_6007_6008 | dn_6008 | pg_residualfiles/archive/pgscrif_archive_20231027145051018138 | 1 | 24576  
(2 rows)
```

NOTE

- This function can archive only the residual files recorded in the current login database. During archiving, the residual files are verified. The verification results are as follows:
 - Verification passed: After the verification is passed, the residual files are archived and marked as handled.
 - Verification failed: If the verification fails, the residual files are not archived and marked as handled.
 - Verification delayed: If the verification is delayed, it indicates that verification is not possible at the moment, and archiving is skipped. The verification delay is related to the transaction completion status and the standby node's redo progress.
- The actual archive directory and the corresponding tablespace are in the same file system. Deleting the tablespace also deletes the archived residual files.
- This function cannot be used after DDL delay is enabled.
- This function is a heavy-load operation and cannot be used concurrently on a single node. You should avoid using this function when the service is overloaded or the resource is scarce.

pg_rm_scan_residualfiles_archive()

Description: Deletes all archived residual file records on the current node. This function is an instance-level function and is irrelevant to the current database. It can run on any instance.

Return type: record

The command output is as follows.

Table 7-45 Return fields

Field	Type	Description
count	bigint	Number of residual files that have been deleted from the archive. For residual files in the local path, the number of deleted files is counted. For residual files in the OBS path, the number of deleted table directories is counted.
size	bigint	Total size of residual files that have been deleted from the archive, in bytes. For residual files in the OBS path, the value of this parameter is 0 .

Example:

```
SELECT * FROM pg_rm_scan_residualfiles_archive();  
count | size  
-----+-----  
4 | 81920  
(1 row)
```

pgxc_rm_scan_residualfiles_archive(query_flag)

Description: Unified CN execution function of **pg_rm_scan_residualfiles_archive()**. This function is a cluster-level function and is irrelevant to the current database. It runs on CNs.

The **query_flag** parameter is of the int type and indicates the execution range. **1** indicates the CN, **2** indicates the primary DN, and **4** indicates the standby DN. The query union set can be obtained through the OR operation. For example, **1|2=3** indicates the CN and primary DN, and **1|2|4=7** indicates the CN, primary DN, and standby DN. The default value is **7**.

Return type: record

The command output is as follows.

Table 7-46 Return fields

Field	Type	Description
nodename	name	Node name.
instance_id	text	Instance name.
count	bigint	Number of residual files that have been deleted from the archive. For residual files in the local path, the number of deleted files is counted. For residual files in the OBS path, the number of deleted table directories is counted.
size	bigint	Total size of residual files that have been deleted from the archive, in bytes. For residual files in the OBS path, the value of this parameter is 0 .

Example:

```
SELECT * FROM pgxc_rm_scan_residualfiles_archive();
 node_name | instance_id | count | size
-----+-----+-----+-----
dn_6001_6002 | dn_6001 | 4 | 81920
cn_5001 | cn_5001 | 1 | 40960
dn_6007_6008 | dn_6008 | 1 | 24576
coordinator1 | coordinator1 | 1 | 0
(4 rows)
```

7.30.3 Functions for Verifying Residual Files

pg_verify_residualfiles(filepath)

Description: Verifies whether the file recorded in the parameter specified file is a residual file. This function is an instance-level function and is related to the current database. It can run on any instance.

Parameter type: text

Return type: bool

The following table describes return columns.

Table 7-47 pg_verify_residualfiles (filepath) return fields

Column	Type	Description
isverified	bool	Whether the verification is complete.

Example:

```
postgres=#SELECT * FROM pg_verify_residualfiles('pgrf_20200908160211441546');
isverified
-----
t
(1 row)
```

 NOTE

This function only verifies whether the recorded file is a residual file in the current database. If the recorded file is not in the current database, the verification is not applicable.

pg_verify_residualfiles()

Description: Verifies whether recorded files on all residual file lists of the current instance are residual files. This function is an instance-level function and is related to the current database. It can run on any instance.

Parameter type: none

Return type: record

The following table describes return columns.

Table 7-48 pg_verify_residualfiles () return fields

Column	Type	Description
result	bool	Whether the verification is complete.
filepath	text	Residual file path.
notes	text	Notes.

Example:

```
postgres=#SELECT * FROM pg_verify_residualfiles();
result |      filepath      | notes
-----+-----+-----
t      | pgrf_20200908160211441546 |
(1 row)
```

 NOTE

This function only verifies whether the recorded file is a residual file in the current database. If the recorded file is not in the current database, the verification is not applicable.

pgxc_verify_residualfiles()

Description: Unified CN query function of pg_verify_residualfiles() This function is a cluster-level function and is related to the current database. It runs on CNs.

Parameter type: none

Return type: record

The following table describes return columns.

Table 7-49 pgxc_verify_residualfiles () return fields

Column	Type	Description
nodename	text	Node name.
result	bool	Whether the verification is complete.
filepath	text	Residual file path.
notes	text	Notes.

Example:

```
postgres=#SELECT * FROM pgxc_verify_residualfiles();
 nodename | result |      filepath      | notes
-----+-----+-----+-----
  cn_5001 |    t   | pgrf_20200910170129360401 |
 dn_6001_6002 | t     | pgrf_20200908160211441546 |
(2 rows)
```

NOTE

This function only verifies whether the recorded file is a residual file in the current database. If the recorded file is not in the current database, the verification is not applicable.

pg_is_residualfiles(residualfile)

Description: Queries whether a specified **relfilenode** is a residual file in the current database. This function is an instance-level function and is related to the current database. It can run on any instance.

Parameter type: text

Return type: bool

The following table describes return columns.

Table 7-50 pg_is_residualfiles (residualfile) return fields

Column	Type	Description
result	bool	Whether a file is a residual file.

Example:

```
postgres=#SELECT * FROM pg_is_residualfiles('base/49155/114691');
result
-----
t
(1 row)
```

NOTE

This function only verifies whether the recorded file is a residual file in the current database. If the recorded file is not in the current database, it is verified as a residual file.

For example, the file **base/15092/14790** is not regarded as a residual file in a **gaussdb** database, but it is regarded as a residual file in other databases.

```
postgres=#SELECT * FROM pg_is_residualfiles('base/15092/14790');
result
-----
f
(1 row)

postgres=#\c db2
db2=# SELECT * FROM pg_is_residualfiles('base/15092/14790');
result
-----
t
(1 row)
```

7.30.4 Functions for Deleting Residual Files

pg_rm_residualfiles(filepath)

Description: Deletes files from a specified residual file list on the current instance. This function is an instance-level function and is irrelevant to the current database. It can run on any instance.

Parameter type: text

Return type: record

The following table describes return columns.

Table 7-51 pg_rm_residualfiles (filepath) return fields

Column	Type	Description
result	bool	Deletion completed or not

Example:

```
postgres=#SELECT * FROM pg_rm_residualfiles('pgrf_20200908160211441599');
result
-----
t
(1 row)
```


 NOTE

- Residual files can be deleted only after verification using the `pg_verify_residualfiles()` function.
- All verified files, regardless which database they are in, will be deleted.
- If all files recorded in the specified file have been deleted, the specified file will be removed and backed up in the `$PGDATA/pg_residualfile/backup` directory.

`pg_rm_residualfiles()`

Description: Deletes all files recorded on all residual file lists on the current instance. This function is an instance-level function and is irrelevant to the current database. It can run on any instance.

Parameter type: none

Return type: record

The following table describes return columns.

Table 7-52 `pg_rm_residualfiles ()` return fields

Column	Type	Description
result	bool	Deletion completed or not
filepath	text	Residual file path
notes	text	Notes

Example:

```
postgres=#SELECT * FROM pg_rm_residualfiles();
result |      filepath      | notes
-----+-----+-----
t      | pgrf_20200908160211441546 |
(1 row)
```

 NOTE

- Residual files can be deleted only after verification using the `pg_verify_residualfiles()` function.
- All verified files, regardless which database they are in, will be deleted.
- If all files recorded in the specified file have been deleted, the specified file will be removed and backed up in the `$PGDATA/pg_residualfile/backup` directory.

`pgxc_rm_residualfiles()`

Description: Unified CN query function of `pgxc_rm_residualfiles`. This function is a cluster-level function and is irrelevant to the current database. It runs on CNs.

Parameter type: none

Return type: record

The following table describes return columns.

Table 7-53 pgxc_rm_residualfiles () return fields

Column	Type	Description
nodename	text	Node name
result	bool	Deletion completed or not
filepath	text	Residual file path
notes	text	Notes

Example:

```
postgres=#SELECT * FROM pgxc_rm_residualfiles();
 nodename | result |      filepath      | notes
-----+-----+-----+-----
  cn_5001 |    t   | pgrf_20200910170129360401 |
dn_6001_6002 | t     | pgrf_20200908160211441546 |
(2 rows)
```

pgxc_clear_disk_cache()

Description: This function deletes all disk cache files. This function is supported only by clusters of version 9.0.2 or later.

Return type: void

Example:

```
SELECT pgxc_clear_disk_cache();
pgxc_clear_disk_cache
-----
(1 row)
```

7.30.5 Using Residual File Management Functions

Procedure

- Step 1** Call the **pgxc_get_residualfiles()** function to obtain the name of the database that has residual files.
- Step 2** Go to the databases where residual files exist and call the **pgxc_verify_residualfiles()** function to verify the residual files recorded in the current database.
- Step 3** Call the **pgxc_rm_residualfiles()** function to delete all the verified residual files.

----End

NOTE

The pgxc residual file management function only operates on the CN and the current primary DN, and does not verify or clear residual files on the standby DN. Therefore, after the primary DN is cleared, you need to clear residual files on the standby DN or build the standby DN in a timely manner. This prevents residual files on the standby DN from being copied back to the primary DN due to incremental build after a primary/standby switchover.

Examples

The following example uses two user-created databases, **db1** and **db2**.

```
postgres=# \l
```

Name	Owner	Encoding	Collate	Ctype	Access privileges
db1	fpi810	SQL_ASCII	C	C	
db2	fpi810	SQL_ASCII	C	C	
postgres	fpi810	SQL_ASCII	C	C	
template0	fpi810	SQL_ASCII	C	C	=c/fpi810 +
template1	fpi810	SQL_ASCII	C	C	fpi810=CTc/fpi810 +
					=c/fpi810 +
					fpi810=CTc/fpi810

(5 rows)

1. Run the following command to obtain all residual file records of the cluster on the CNs:

db1=# select * from pgxc_get_residualfiles() order by 4, 6; -- **order by** is optional.

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
```

nodename	isverified	isdeleted	dbname	residualfile	filepath	notes
dn_6001_6002	f	f	db1	base/16390/16395	pgrf_20200921153355979438	
dn_6001_6002	f	f	db1	base/16390/24592	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24579	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24582	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24584	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24585	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24588	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24589	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24591	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24576	pgrf_20200921153612088342	

(10 rows)

In the current cluster:

- Residual file records exist in the **db1** and **db2** databases on the **dn_6001_6002** node (active node instance).
 - Residual files are displayed in the **residualfile** column.
 - The **filepath** column lists the files that record residual files. These files are stored in the **pg_residualfiles** directory under the instance data directory.
2. Call the **pgxc_verify_residualfiles()** function to verify the **db1** database.
db1=# SELECT * FROM pgxc_verify_residualfiles();

```
postgres=# \c db1
```

Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "fpi810".

```
db1=# select * from pgxc_verify_residualfiles();
```

nodename	result	filepath	notes
dn_6001_6002	t	pgrf_20200921153355979438	
dn_6001_6002	t	pgrf_20200921153612088342	

(2 rows)

Verification functions are at the database level. Therefore, when a verification function is called in the **db1** database, it only verifies residual files in **db1**.

You can call the **get** function again to check whether the verification is complete.

```
db1=# SELECT * FROM pgxc_get_residualfiles() order by 4, 6;
```

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
```

nodename	isverified	isdeleted	dbname	residualfile	filepath	notes
dn_6001_6002	t	f	db1	base/16390/16395	pgrf_20200921153355979438	
dn_6001_6002	t	f	db1	base/16390/24592	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24579	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24582	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24584	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24585	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24588	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24589	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24591	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24576	pgrf_20200921153612088342	

(10 rows)

As shown in the preceding figure, the residual files in the **db1** database have been verified, and the residual files in the **db2** database are not verified.

3. Call the **pgxc_rm_residualfiles()** function to delete residual files.

```
db1=# SELECT * FROM pgxc_rm_residualfiles();
```

```
db1=# select * from pgxc_rm_residualfiles();
```

nodename	result	filepath	notes
dn_6001_6002	t	pgrf_20200921153355979438	
dn_6001_6002	t	pgrf_20200921153612088342	

(2 rows)

4. Call the **pgxc_get_residualfiles()** function again to check the deletion result.

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
```

nodename	isverified	isdeleted	dbname	residualfile	filepath	notes
dn_6001_6002	t	t	db1	base/16390/24592	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24579	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24582	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24584	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24576	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24588	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24589	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24591	pgrf_20200921153612088342	
dn_6001_6002	f	f	db2	base/16391/24585	pgrf_20200921153612088342	

(9 rows)

The result shows that the residual files in the **db1** database are deleted (**isdeleted** is marked as **t**) and the residual files in the **db2** database are not deleted.

In addition, nine query results are displayed. Compared with the previous query results, a record for the residual file ending with **9438** is missing. This is because the record file that records the residual file ending with **9438** contains only one record, which is deleted in step 3. If all residual files in a record file are deleted, the record file is also deleted. Deleted files are backed up in the **pg_residualfiles/backup** directory.

```
[fpi810@host-192-168-244-162 pg_residualfiles]$ ls  
backup pendingdeletesfile pgrf_20200921153612088342  
[fpi810@host-192-168-244-162 pg_residualfiles]$ ls backup/  
pgrf_20200921153355979438_bak
```

5. To delete files from the **db2** database, you need to call the **verify** function in the **db2** database and then call the **rm** function.
 - a. Go to the **db2** database and call the verification function.

```

db1=# \c db2
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db2" as user "fpi810".
db2=# select * from pgxc_verify_residualfiles();
  nodename | result |      filepath      | notes
-----+-----+-----+-----
  dn_6001_6002 | t      | pgrf_20200921153612088342 |
(1 row)

```

Query the verification result:

```

db2=# select * from pgxc_get_residualfiles() order by 4, 6;
  nodename | isverified | isdeleted | dbname | residualfile |      filepath      | notes
-----+-----+-----+-----+-----+-----+-----
  dn_6001_6002 | t          | f         | db1    | base/16390/24592 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24579 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24582 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24584 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24576 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24588 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24589 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24591 | pgrf_20200921153612088342 |
  dn_6001_6002 | t          | f         | db2    | base/16391/24585 | pgrf_20200921153612088342 |
(9 rows)

```

- b. Call the deletion function:

```

db2=# select * from pgxc_rm_residualfiles();
  nodename | result |      filepath      | notes
-----+-----+-----+-----
  dn_6001_6002 | t      | pgrf_20200921153612088342 |
(1 row)

```

- c. Query the deletion result:

```

db2=# select * from pgxc_get_residualfiles() order by 4, 6;
  nodename | isverified | isdeleted | dbname | residualfile | filepath | notes
-----+-----+-----+-----+-----+-----+-----
(0 rows)

```

All residual files recorded in the record file whose name ends with **8342** have been deleted, so the record file is deleted and backed up in the **backup** directory. As a result, no records are found.

```

[fpi810@host-192-168-244-162 pg_residualfiles]$ ls
backup pendingdeletesfile
[fpi810@host-192-168-244-162 pg_residualfiles]$ ls backup/
pgrf_20200921153355979438_bak pgrf_20200921153612088342_bak

```

7.31 Statistics Information Functions

Statistics information functions are divided into the following two categories: functions that access databases, using the OID of each table or index in a database to mark the database for which statistics are generated; functions that access servers, identified by the server process ID, whose value ranges from 1 to the number of currently active servers.

For a global temporary table, the input parameter of the function is the OID of the table in **pg_class**. The function returns the statistics of the current session. The results of querying the same global temporary table in different sessions may be different.

pg_stat_get_db_numbackends(oid)

Description: Obtains the number of active server threads of a specified database on the current instance.

Return type: integer

pg_stat_get_db_total_numbackends(oid)

Description: Obtains the total number of active server threads of a specified database on all CNs in a cluster (if this function is executed on a CN), or obtains the number of active server threads of a specified database on the current instance (if this function is executed on a DN).

Return type: integer

pg_stat_get_db_xact_commit(oid)

Description: Obtains the number of committed transactions in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_xact_commit(oid)

Description: Obtains the total number of committed transactions in a specified database on all CNs in a cluster (if this function is executed on a CN), or obtains the number of committed transactions in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_xact_rollback(oid)

Description: Obtains the number of rollback transactions in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_xact_rollback(oid)

Description: Obtains the total number of rollback transactions in a specified database on all CNs in a cluster (if this function is executed on a CN), or obtains the number of rollback transactions in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_blocks_fetched(oid)

Description: Obtains the number of disk block fetch requests in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_blocks_fetched(oid)

Description: Obtains the total number of disk block fetch requests in a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of disk block fetch requests in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_blocks_hit(oid)

Description: Obtains the number of requested disk blocks found in the cache in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_blocks_hit(oid)

Description: Obtains the total number of requested disk blocks found in the cache in a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of requested disk blocks found in the cache in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_tuples_returned(oid)

Description: This function obtains the number of tuples returned for a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_tuples_returned(oid)

Description: Obtains the total number of tuples returned for a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of tuples returned for a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_tuples_fetched(oid)

Description: Obtains the number of tuples read from a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_tuples_fetched(oid)

Description: Obtains the total number of tuples read from a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of tuples read from a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_tuples_inserted(oid)

Description: Obtains the number of tuples inserted into a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_tuples_inserted(oid)

Description: Obtains the total number of tuples inserted into a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of tuples inserted into a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_tuples_updated(oid)

Description: Obtains the number of updated tuples in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_tuples_updated(oid)

Description: Obtains the total number of updated tuples in a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of updated tuples in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_tuples_deleted(oid)

Description: Obtains the number of tuples deleted from a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_tuples_deleted(oid)

Description: Obtains the total number of tuples deleted from a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of tuples deleted from a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_conflict_lock(oid)

Description: Obtains the total number of conflicting locks in a specified database on all CNs and DNs in a cluster (if this function is executed on a CN), or obtains the number of conflicting locks in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_deadlocks(oid)

Description: Obtains the number of deadlocks in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_deadlocks(oid)

Description: Obtains the total number of deadlocks in a specified database on all CNs and DN in a cluster (if this function is executed on a CN), or obtains the number of deadlocks in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_conflict_all(oid)

Description: Obtains the number of conflict recoveries in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_conflict_all(oid)

Description: Obtains the total number of conflict recoveries in a specified database on all CNs and DN in a cluster (if this function is executed on a CN), or obtains the number of conflict recoveries in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_temp_files(oid)

Description: Obtains the number of temporary files created in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_temp_files(oid)

Description: Obtains the total number of temporary files created in a specified database on all DN in a cluster (if this function is executed on a CN), or obtains the number of temporary files created in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_temp_bytes(oid)

Description: Obtains the number of bytes of the temporary files created in a specified database on the current instance.

Return type: bigint

pg_stat_get_db_total_temp_bytes(oid)

Description: Obtains the total number of bytes of the temporary files created in a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the number of bytes of the temporary files created in a specified database on the current instance (if this function is executed on a DN).

Return type: bigint

pg_stat_get_db_blk_read_time(oid)

Description: Obtains the time required for reading data blocks from a specified database on the current instance.

Return type: double

pg_stat_get_db_total_blk_read_time(oid)

Description: Obtains the total time required for reading data blocks from a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the time required for reading data blocks from a specified database on the current instance (if this function is executed on a DN).

Return type: double

pg_stat_get_db_blk_write_time(oid)

Description: Obtains the time required for writing data blocks to a specified database on the current instance.

Return type: double

pg_stat_get_db_total_blk_write_time(oid)

Description: Obtains the total time required for writing data blocks to a specified database on all DNs in a cluster (if this function is executed on a CN), or obtains the time required for writing data blocks to a specified database on the current instance (if this function is executed on a DN).

Return type: double

pg_stat_get_numscans(oid)

Description: Number of sequential row scans done if parameters are in a table or number of index scans done if parameters are in an index

Return type: bigint

pg_stat_get_tuple()

Description: This function can be executed on both CNs and DNs. This function is supported only by version 8.1.3 or later clusters.

If no parameters are specified, this function queries the statistics of all system catalogs on CNs, the dirty page rate of the tables on each CN, the statistics of all

system catalogs and user catalogs on DNs, and the dirty page rate of the tables on each DN.

If the schema name and table name are specified, this function queries the statistics and dirty page rate of the specified table.

 NOTE

The statistics of this function depend on the **ANALYZE** operation. To obtain the most accurate information, perform the **ANALYZE** operation on the table first.

Return type: record

The following table describes return columns.

Column	Type	Description
nodename	text	Node name
tableid	oid	Table OID
partid	oid	Partition OID of the partitioned table
last_vacuum	timestamp with time zone	Time of the last manual VACUUM
last_autovacuum	timestamp with time zone	Time of the last AUTOVACUUM
last_analyze	timestamp with time zone	Time of the last manual ANALYZE
last_autoanalyze	timestamp with time zone	Time of the last AUTOANALYZE
vacuum_count	bigint	Number of VACUUM operations
autovacuum_count	bigint	Number of AUTOVACUUM operations
analyze_count	bigint	Number of ANALYZE operations
autoanalyze_count	bigint	Number of AUTOANALYZE_COUNT operations
n_tup_ins	bigint	Number of rows inserted
n_tup_upd	bigint	Number of rows updated
n_tup_del	bigint	Number of rows deleted
n_tup_hot_upd	bigint	Number of rows with HOT updates
n_tup_change	bigint	Number of changed rows after ANALYZE
n_live_tup	bigint	Estimated number of live rows
n_dead_tup	bigint	Estimated number of dead rows
dirty_rate	bigint	Dirty page rate of a single CN or DN

Column	Type	Description
last_data_changed	timestamp with time zone	Time when a table was last modified. This column is left empty in 8.2.0 and later versions. You are advised to query the value using pg_stat_get_last_data_changed_time(oid) .

pg_stat_get_tuples_returned(oid)

Description: Number of sequential row scans done if parameters are in a table or number of index entries returned if parameters are in an index

Return type: bigint

pg_stat_get_tuples_fetched(oid)

Description: Number of table rows fetched by bitmap scans if parameters are in a table,

or table rows fetched by simple index scan using the index if parameters are in an index

Return type: bigint

pg_stat_get_tuples_inserted(oid)

Description: Number of rows inserted into table

Return type: bigint

pg_stat_get_local_tuples_inserted(oid)

Description: Number of rows inserted into the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_tuples_updated(oid)

Description: Number of rows updated in table

Return type: bigint

pg_stat_get_local_tuples_updated(oid)

Description: Number of rows updated in the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_tuples_deleted(oid)

Description: Number of rows deleted from table

Return type: bigint

pg_stat_get_local_tuples_deleted(oid)

Description: Number of rows deleted from the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_tuples_changed(oid)

Description: Queries the function on a CN and returns the total number of inserted, updated, and deleted rows in the table since the last **ANALYZE** or **AUTOANALYZE** operation. Queries the function on a DN and returns the total number of inserted, updated, and deleted rows in the table since the last **ANALYZE** or **AUTOANALYZE** operation on the current node.

Return type: bigint

pg_stat_get_local_tuples_changed(oid)

Description: Number of inserted, updated, and deleted rows in the table since the last **ANALYZE** or **AUTOANALYZE** operation on the current node.

Return type: bigint

pg_stat_get_tuples_hot_updated(oid)

Description: Number of rows HOT-updated in table

Return type: bigint

pg_stat_get_local_tuples_hot_updated(oid)

Description: Number of rows with HOT updates in the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_live_tuples(oid)

Description: Number of live tuples in the table.

Return type: bigint

pg_stat_get_local_live_tuples(oid)

Description: Number of live tuples in the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_dead_tuples(oid)

Description: Number of dead tuples in the table.

Return type: bigint

pg_stat_get_local_dead_tuples(oid)

Description: Number of dead tuples in the table on the current node. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_blocks_fetched(oid)

Description: Number of disk block fetch requests for table or index

Return type: bigint

pg_stat_get_blocks_hit(oid)

Description: Number of disk block requests found in cache for table or index

Return type: bigint

pg_stat_get_partition_tuples_inserted(oid)

Description: Number of rows in the corresponding table partition

Return type: bigint

pg_stat_get_partition_tuples_updated(oid)

Description: Number of rows that have been updated in the corresponding table partition

Return type: bigint

pg_stat_get_partition_tuples_deleted(oid)

Description: Number of rows deleted from the corresponding table partition

Return type: bigint

pg_stat_get_partition_tuples_changed(oid)

Description: Total number of inserted, updated, and deleted rows after the table partition was last analyzed or autoanalyzed

Return type: bigint

pg_stat_get_partition_live_tuples(oid)

Description: Number of live rows in a table partition

Return type: bigint

pg_stat_get_partition_dead_tuples(oid)

Description: Number of dead rows in a table partition

Return type: bigint

pg_stat_get_xact_tuples_inserted(oid)

Description: Number of tuple inserted into the active subtransactions related to the table.

Return type: bigint

pg_stat_get_xact_tuples_deleted(oid)

Description: Number of deleted tuples in the active subtransactions related to a table

Return type: bigint

pg_stat_get_xact_tuples_hot_updated(oid)

Description: Number of hot updated tuples in the active subtransactions related to a table

Return type: bigint

pg_stat_get_xact_tuples_updated(oid)

Description: Number of updated tuples in the active subtransactions related to a table

Return type: bigint

pg_stat_get_xact_partition_tuples_inserted(oid)

Description: Number of inserted tuples in the active subtransactions related to a table partition

Return type: bigint

pg_stat_get_xact_partition_tuples_deleted(oid)

Description: Number of deleted tuples in the active subtransactions related to a table partition

Return type: bigint

pg_stat_get_xact_partition_tuples_hot_updated(oid)

Description: Number of hot updated tuples in the active subtransactions related to a table partition

Return type: bigint

pg_stat_get_xact_partition_tuples_updated(oid)

Description: Number of updated tuples in the active subtransactions related to a table partition

Return type: bigint

pg_stat_get_last_vacuum_time(oid)

Description: Last time when the autovacuum thread is manually started to clear a table

Return type: timestamptz

pg_stat_get_last_autovacuum_time(oid)

Description: Time of the last vacuum initiated by the autovacuum thread on this table

Return type: timestamptz

pg_stat_get_local_last_autovacuum_time(oid)

Description: Time of the last vacuum initiated by the autovacuum thread of the current node on this table. This function is supported only in 8.1.2 or later.

Return type: timestamptz

pg_stat_get_vacuum_count(oid)

Description: Number of times a table is manually cleared

Return type: bigint

pg_stat_get_autovacuum_count(oid)

Description: Number of times of vacuum initiated by the autovacuum thread on this table

Return type: bigint

pg_stat_get_local_autovacuum_count(oid)

Description: Number of times of vacuum initiated by the autovacuum thread of the current node on this table. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_last_analyze_time(oid)

Description: Last time when a table starts to be analyzed manually or by the autovacuum thread

Return type: timestamptz

pg_stat_get_last_autoanalyze_time(oid)

Description: Time of the last analysis initiated by the autovacuum thread on this table

Return type: timestamptz

pg_stat_get_local_last_autoanalyze_time(oid)

Description: Time of the last analysis initiated by the autovacuum thread of the current node on this table. This function is supported only in 8.1.2 or later.

Return type: timestamptz

pg_stat_get_analyze_count(oid)

Description: Number of times a table is manually analyzed

Return type: bigint

pg_stat_get_autoanalyze_count(oid)

Description: Number of times the autovacuum daemon analyzes a table

Return type: bigint

pg_stat_get_local_autoanalyze_count(oid)

Description: Number of times that the autovacuum daemon of the current node starts analysis on this table. This function is supported only in 8.1.2 or later.

Return type: bigint

pg_stat_get_local_analyze_status(oid)

Description: Specifies whether to analyze the status of a table on the current node. This parameter is valid only for CNs. This function is supported only in 8.1.2 or later.

- If the number of modified rows in the table exceeds the ANALYZE threshold (calculated based on `autovacuum_analyze_threshold` + `autovacuum_analyze_scale_factor` x `reltuples`, where **reltuples** is the estimated number of rows in the table recorded in `pg_class`), **Analyze needed** is returned.
- If the number of modified rows in the table does not exceed the threshold of **analyze**, the message **Analyze not needed** is returned.

Return type: text

pg_stat_get_runtime_relstats(oid)

Description: Returns memory table-level statistics generated by lightweight autoanalyze.

Return type: SETOF record

pg_stat_get_runtime_attstats(oid)

Description: Returns memory table- and column- level statistics generated by lightweight autoanalyze.

Return type: SETOF record

pg_stat_get_predicate_columns(oid)

Description: This function returns the predicate column information collected by a table. Only clusters of 9.1.0.100 and later versions support this function.

Return type: setof smallint

pg_stat_delete_predicate_columns(oid)

Description: This function clears the predicate column information of a table. Only clusters of 9.1.0.100 and later versions support this function.

Return type: void

pg_total_autovac_tuples(bool)

Description: Gets the tuple records related to **total autovac**, such as **nodename**, **nspname**, **relname**, and the IUD information of tuples.

Return type: SETOF record

pg_autovac_status(oid)

Description: Returns autovac information, such as **nodename**, **nspname**, **relname**, **analyze**, **vacuum**, thresholds of **analyze** and **vacuum**, and the number of analyzed or vacuumed tuples.

Return type: SETOF record

pg_autovac_timeout(oid)

Description: Returns the number of consecutive timeouts during the autovac operation on a table. If the table information is invalid or the node information is abnormal, **NULL** will be returned.

Return type: bigint

pg_autovac_coordinator(oid)

Description: Returns the name of the CN performing the autovac operation on a table. If the table information is invalid or the node information is abnormal, **NULL** will be returned.

Return type: text

pg_stat_get_all(regclass)

Description: This function retrieves a tuple record from the **pg_stat_object** table on the current CN. The record includes 29 columns, ranging from **databaseid** to

autovac_vacuum_count. The input parameter is the table OID. Only 8.2.1.210 and later cluster versions support this function.

Return type: SETOF record

Example: Query the last access time of the **t1** table in the public schema on the current node.

```
SELECT data_access_timestamp FROM pg_stat_get_all('public:t1::regclass');
data_access_timestamp
-----
2024-06-27 16:16:26.643132+08
(1 row)
```

pgxc_stat_get_all(relnamespace text, relname text)

Description: This function retrieves tuple records from the **pg_stat_object** table on all CNs. The record includes 29 columns, ranging from **databaseid** to **autovac_vacuum_count**. Only 8.2.1.210 and later cluster versions support this function.

Return type: SETOF record

Example: Query the last access time of the **t1** table in the public schema on all CNs.

```
SELECT relid,data_access_timestamp FROM pgxc_stat_get_all('public','t1');
 relid  | data_access_timestamp
-----+-----
2147516676 | 2000-01-01 08:00:00+08
2147516674 | 2000-01-01 08:00:00+08
2147516655 | 2024-06-27 16:16:26.643132+08
(3 rows)
```

pgxc_stat_get_last_data_access_timestamp(relnamespace text, relname text)

Description: This function returns the last access time of a specified table on all nodes. Clusters of 9.1.0.100 and later versions support this function.

Return type: timestamptz

Example: Query the last access time of table **t1** in the public schema on all nodes.

```
SELECT * FROM pgxc_stat_get_last_data_access_timestamp('public','t1');
pgxc_stat_get_last_data_access_timestamp
-----
2024-06-27 16:16:26.643132+08
(1 row)
```

pgxc_stat_flush_object_data()

Description: This function enables manual persistence, where each node writes statistics on all tables in local databases to the **pg_stat_object** system catalog. The statistics include 29 columns from **databaseid** to **autovac_vacuum_count** in **pg_stat_object**. Only the system administrator can execute this function. Clusters of 9.1.0.100 and later versions support this function.

Return type: Boolean

Example: The system administrator performs persistence.

```
SELECT pgxc_stat_flush_object_data();
pgxc_stat_flush_object_data
```

```
-----  
t  
(1 row)
```

 NOTE

This function cannot be executed concurrently.

pgxc_get_wlm_session_info_bytime(text, timestamp without time zone, timestamp without time zone, int)

Description: The query performance of the PGXC_WLM_SESSION_INFO view is poor if the view contains a large number of records. In this case, you are advised to use this function to filter the query. The input parameters are *time column* (**start_time** or **finish_time**), *start time*, *end time*, and *maximum number of records returned for each CN*. The return result is a subset of records in the **GS_WLM_SESSION_HISTORY** view.

Return type: SETOF record

pgxc_get_wlm_current_instance_info(text, int default null)

Description: Queries the current resource usage of each node in the cluster on the CN and reads the data that is not stored in the GS_WLM_INSTANCE_HISTORY system catalog in the memory. The input parameters are the node name (**ALL**, **C**, **D**, or *instance name*) and the maximum number of records returned by each node. The returned value is **GS_WLM_INSTANCE_HISTORY**.

Return type: SETOF record

pgxc_get_wlm_history_instance_info(text, TIMESTAMP, TIMESTAMP, int default null)

Description: Queries the historical resource usage of each cluster node on the CN node and reads data from the **GS_WLM_INSTANCE_HISTORY** system catalog. The input parameters are as follows: node name (**ALL**, **C**, **D**, or *instance name*), start time, end time, and maximum number of records returned for each instance. The returned value is **GS_WLM_INSTANCE_HISTORY**.

Return type: SETOF record

pg_stat_get_last_data_changed_time(oid)

Description: Returns the time when **INSERT**, **UPDATE**, **DELETE**, or **EXCHANGE/TRUNCATE/DROP PARTITION** was performed last time on a table. The data in the **last_data_changed** column of the PG_STAT_ALL_TABLES view is calculated by using this function. The performance of obtaining the last modification time by using the view is poor when the table has a large amount of data. In this case, you are advised to use the function.

Return type: timestamptz

pg_stat_set_last_data_changed_time(oid)

Description: Manually changes the time when **INSERT**, **UPDATE**, **DELETE**, or **EXCHANGE/TRUNCATE/DROP PARTITION** was performed last time.

Return type: void

pg_stat_set_last_data_changed_num(oid)

Description: Sets the history modification times of a table node.

Return type: void

pv_session_time()

Description: Collects statistics on the running time of each session thread on the current node and the time consumed in each execution phase.

Return type: record

pv_instance_time()

Description: Collects statistics on the running time of the current node and the time consumed in each execution phase.

Return type: record

pg_stat_get_activity(integer)

Description: Returns a record about the backend with the specified PID. A record for each active backend in the system is returned if **NULL** is specified. The return result is a subset of records (excluding the **connection_info** column) in the **PG_STAT_ACTIVITY** view.

Return type: SETOF record

pg_stat_get_activity_with_conninfo(integer)

Description: Returns a record about the backend with the specified PID. A record for each active backend in the system is returned if **NULL** is specified. The return result is a subset of records in the **PG_STAT_ACTIVITY** view.

Return type: SETOF record

pg_user_iostat(text)

Description: This function has been deprecated in version 8.1.2 and is invalid when used in queries in the current version. In cluster versions 8.2.0 and later, you can use the **PGXC_TOTAL_USER_RESOURCE_INFO** view to query the real-time resource consumption of users on instances.

Return type: record

Name	Type	Description
userid	oid	User ID.
min_curr_iops	int4	Minimum I/O of the current user across DNs.
max_curr_iops	int4	Maximum I/O of the current user across DNs.

Name	Type	Description
min_peak_ios	int4	Minimum peak I/O of the current user across DNs.
max_peak_ios	int4	Maximum peak I/O of the current user across DNs.
io_limits	int4	io_limits set for the resource pool specified by the user.
io_priority	text	io_priority set for the user.

pg_stat_get_function_calls(oid)

Description: Number of times the function has been called

Return type: bigint

pg_stat_get_function_total_time(oid)

Description: Gets the total wall-clock time spent on a function, in microseconds. The time spent on calling this function is included.

Return type: double precision

pg_stat_get_function_self_time(oid)

Description: Gets the time spent only on this function in the current transaction. The time spent on calling this function is not included.

Return type: double precision

pg_stat_get_backend_idset()

Description: Set of currently active server process numbers (from 1 to the number of active server processes)

Return type: setofinteger

pg_stat_get_backend_pid(integer)

Description: Thread ID of the given server thread

Return type: bigint

```
SELECT pg_stat_get_backend_pid(1);
pg_stat_get_backend_pid
-----
139706243217168
(1 row)
```

pg_stat_get_backend_dbid(integer)

Description: ID of the database connected to the given server process

Return type: OID

pg_stat_get_backend_userid(integer)

Description: User ID of the given server process

Return type: OID

pg_stat_get_backend_activity(integer)

Description: Active command of the given server process, but only if the current user is a system administrator or the same user as that of the session being queried and **track_activities** is on

Return type: text

pg_stat_get_backend_waiting(integer)

Description: True if the given server process is waiting for a lock, but only if the current user is a system administrator or the same user as that of the session being queried and **track_activities** is on

Return type: boolean

pg_stat_get_backend_activity_start(integer)

Description: The time at which the given server process's currently executing query was started, but only if the current user is a system administrator or the same user as that of the session being queried and **track_activities** is on

Return type: timestamp with time zone

pg_stat_get_backend_xact_start(integer)

Description: The time at which the given server process's currently executing transaction was started, but only if the current user is a system administrator or the same user as that of the session being queried and **track_activities** is on

Return type: timestamp with time zone

pg_stat_get_backend_start(integer)

Description: The time at which the given server process was started, or **NULL** if the current user is neither a system administrator nor the same user as that of the session being queried

Return type: timestamp with time zone

pg_stat_get_backend_client_addr(integer)

Description: IP address of the client connected to the given server process.

If the connection is over a Unix domain socket, or if the current user is neither a system administrator nor the same user as that of the session being queried, **NULL** will be returned.

Return type: inet

Note: An IP address used as an input parameter of this function cannot contain periods (.). For example, **192.168.100.128** should be written as **192168100128**.

pg_stat_get_backend_client_port(integer)

Description: TCP port number of the client connected to the given server process

If the connection is over a Unix domain socket, **-1** will be returned. If the current user is neither a system administrator nor the same user as that of the session being queried, **NULL** will be returned.

Return type: integer

pg_stat_get_bgwriter_timed_checkpoints()

Description: The number of times the background writer has started timed checkpoints (because the **checkpoint_timeout** time has expired)

Return type: bigint

pg_stat_get_bgwriter_requested_checkpoints()

Description: The number of times the background writer has started checkpoints based on requests from the backend because **checkpoint_segments** has been exceeded or the **CHECKPOINT** command has been executed

Return type: bigint

pg_stat_get_bgwriter_buf_written_checkpoints()

Description: The number of buffers written by the background writer during checkpoints

Return type: bigint

pg_stat_get_bgwriter_buf_written_clean()

Description: The number of buffers written by the background writer for routine cleaning of dirty pages

Return type: bigint

pg_stat_get_bgwriter_maxwritten_clean()

Description: The number of times the background writer has stopped its cleaning scan because it has written more buffers than specified in the **bgwriter_lru_maxpages** parameter

Return type: bigint

pg_stat_get_buf_written_backend()

Description: The number of buffers written by the backend because they needed to allocate a new buffer

Return type: bigint

pg_stat_get_buf_alloc()

Description: The total number of buffer allocations

Return type: bigint

pg_stat_clear_snapshot()

Description: Discards the current statistics snapshot.

Return type: void

pg_stat_reset()

Description: Resets all statistics counters for the current database to zero (requires system administrator permissions).

Return type: void

pg_stat_reset_shared(text)

Description: Resets all statistics counters for the current database in each node in a shared cluster to zero (requires system administrator permissions).

Return type: void

pg_stat_reset_single_table_counters(oid)

Description: Resets statistics for a single table or index in the current database to zero (requires system administrator permissions).

Return type: void

pg_stat_reset_single_function_counters(oid)

Description: Resets statistics for a single function in the current database to zero (requires system administrator permissions).

Return type: void

pg_stat_session_cu(int, int, int)

Description: Obtains the compression unit (CU) hit statistics of sessions running on the current node.

Return type: record

gs_get_stat_session_cu(text, int, int, int)

Description: Obtains the CU hit statistics of all sessions running in a cluster.

Return type: record

gs_get_stat_db_cu(text, text, bigint, bigint, bigint)

Description: Obtains the CU hit statistics of a database in a cluster.

Return type: record

pg_stat_get_cu_mem_hit(oid)

Description: Obtains the number of CU memory hits of a column storage table in the current database of the current node.

Return type: bigint

pg_stat_get_cu_hdd_sync(oid)

Description: Obtains the number of times CU is synchronously read from a disk by a column-store table in the current database of the current node.

Return type: bigint

pg_stat_get_cu_hdd_async(oid)

Description: Obtains the number of times CU is asynchronously read from a disk by a column-store table in the current database of the current node.

Return type: bigint

pg_stat_get_db_cu_mem_hit(oid)

Description: Obtains the CU memory hit in a database of the current node.

Return type: bigint

pg_stat_get_db_cu_hdd_sync(oid)

Description: Obtains the times CU is synchronously read from a disk by a database of the current node.

Return type: bigint

pg_stat_get_db_cu_hdd_async(oid)

Description: Obtains the times CU is asynchronously read from a disk by a database of the current node.

Return type: bigint

pgxc_fenced_udf_process()

Description: Shows the number of UDF Master and Work processes.

Return type: record

pgxc_terminate_all_fenced_udf_process()

Description: Kills all UDF Work processes.

Return type: bool

gs_all_nodegroup_control_group_info(text)

Description: Provides Cgroup information for all logical clusters. Before invoking this function, you need to specify the name of a logical cluster to be queried. For example, to query the Cgroup information of the **installation** logical cluster, run the following statement:

```
SELECT * FROM gs_all_nodegroup_control_group_info('installation');
```

Return type: record

The following table describes return columns.

Name	Type	Description
name	text	Name of a Cgroup
type	text	Type of the Cgroup
gid	bigint	Cgroup ID
classgid	bigint	ID of the Class Cgroup where a Workload Cgroup belongs
class	text	Class Cgroup
workload	text	Workload Cgroup
shares	bigint	CPU quota allocated to a Cgroup
limits	bigint	Limit of CPUs allocated to a Cgroup
wdlevel	bigint	Workload Cgroup level
cpucores	text	Usage of CPU cores in a Cgroup

gs_get_nodegroup_tablecount(name)

Description: Total number of user tables in all the databases in a logical cluster

Return type: integer

pgxc_max_datanode_size(name)

Description: Maximum disk space occupied by database files in all the DN's of a logical cluster. The unit is byte.

Return type: bigint

gs_check_logic_cluster_consistency()

Description: Checks whether the system information of all logical clusters in the system is consistent. If no record is returned, the information is consistent. Otherwise, the Node Group information on CNs and DN's in the logical cluster is

inconsistent. This function cannot be invoked during redistribution in a scale-in or scale-out.

Return type: record

gs_check_tables_distribution()

Description: Checks whether the user table distribution in the system is consistent. If no record is returned, table distribution is consistent. This function cannot be invoked during redistribution in a scale-in or scale-out.

Return type: record

pg_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)

Description: Obtains damage information about pages or CUs after the current node is started.

Return type: record

pgxc_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)

Description: Obtains damage information about pages or CUs after all the nodes in the cluster are started.

Return type: record

pg_stat_bad_block_clear()

Description: Deletes the page and CU damage information that is read and recorded on the node. (System administrator rights are required.)

Return type: void

pgxc_stat_bad_block_clear()

Description: Deletes the page and CU damage information that is read and recorded on all the nodes in the cluster. (System administrator rights are required.)

Return type: void

gs_respool_exception_info(pool text)

Description: Queries for the query rule of a specified resource pool.

Return type: record

gs_control_group_info(pool text)

Description: Queries for information about Cgroups associated with a resource pool.

Return type: record

The following information is displayed:

Attribute	Value	Description
name	class_a:workload_a1	Class name and workload name
class	class_a	Class Cgroup name
workload	workload_a1	Workload Cgroup name
type	DEFWD	Cgroup type (Top, CLASS, BAKWD, DEFWD, and TSWD)
gid	87	Cgroup ID
shares	30	Percentage of CPU resources to those on the parent node
limits	0	Percentage of CPU cores to those on the parent node
rate	0	Allocation ratio in Timeshare
cpucore	0-3	Number of CPU cores

gs_wlm_user_resource_info(name text)

Description: Queries for a user's resource quota and resource usage.

Return type: record

pgxc_stat_single_table(schema, tablename)

Description: Executed on CNs, with the schema name and table name passed. This function queries the statistics of a single table in the entire database and the dirty page rate of the table on each DN.

This function is supported only by version 8.1.3 or later clusters.

NOTE

The statistics of this function depend on the **ANALYZE** operation. To obtain the most accurate information, perform the **ANALYZE** operation on the table first.

Return type: record

The return value fields are the same as those of the [pg_stat_get_tuple\(\)](#) function.

```
SELECT * FROM pgxc_stat_single_table('public','t1');
nodename | tableid | partid | last_vacuum | last_autovacuum | last_analyze |
last_autoanalyze | vacuum_count | autovacuum_count | analyze_count | autoanalyze_count | n_tup_ins | n_
tup_upd | n_tup_del | n_tup_hot_upd | n_tup_change | n_live_tup | n_dead_tup | dirty_rate |
last_data_changed
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
datanode1 | 1270075 |      | 2000-01-01 08:00:00+08 | 2000-01-01 08:00:00+08 | 2023-01-09
09:38:43.220876+08 | 2000-01-01 08:00:00+08 |      0 |      0 |      1 |      0 |      0 |
```

```
0 | 0 | 0 | 0 | 0 | 0 | 0 |
(1 row)
```

pgxc_stat_wal_write()

Description: Records information about DN threads that interact with CNs, and collects statistics on WAL logs and data page import volume and rate. This function is executed on all CNs. This parameter is supported by version 8.2.0 or later clusters.

Return type: bigint

Fields in the returned value:

- **node_name**: instance name.
- **application_name**: application name.
- **query_start**: start time of the statement that is being executed.
- **datapage_write**: total size of data pages generated by the current query, in bytes
- **datapage_write_speed**: data page import speed, in bytes/s.
- **wal_write**: total size WAL logs imported by the current query, in bytes.
- **wal_write_speed**: WAL log import speed, in bytes/s.
- **total_datapage_write**: total size of data pages generated by the current thread, in bytes.
- **total_wal_write**: total size of WAL logs, in bytes.

```
SELECT * FROM pgxc_stat_wal_write();
 node_name | datid | pid | lwtid | usesysid | application_name | state | query_start
 | backend_start | query_id | datapage_write | datapage_write_speed | wal_write |
 wal_write_speed | total_datapage_write | total_wal_write
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 datanode1 | 15979 | 140328473442048 | 12408 | 10 | CalculateSpaceInfo | active | 2022-11-15
10:39:45.00219+08 | 2022-11-15 10:39:45.000918+08 | 0 | 0 | 0 | 70697
 | 0 | 0 | 0
 datanode1 | 0 | 140328075503360 | 12510 | 10 | WalSender to Secondary | idle
 | 2022-11-15 10:39:46.708557+08 | 0 | 0 | 210
 | 0 | 0 | 0
 datanode1 | 15979 | 140327896741632 | 13612 | 10 | coordinator1 | active | 2022-11-15
10:46:32.832548+08 | 2022-11-15 10:40:20.117516+08 | 72620543991349940 | 586579968 |
17425000 | 11743056 | 348000 | 5337505792 | 107245825
 datanode1 | 15979 | 140327583217408 | 13614 | 10 | coordinator1 | active | 2022-11-15
10:46:32.832548+08 | 72620543991349940 | 0 | 0 | 0
 | 0 | 0 | 0
 datanode1 | 15979 | 140327485175552 | 27914 | 10 | coordinator1 | active | 2022-11-15
10:47:06.493584+08 | 2022-11-15 10:47:06.489062+08 | 72620543991350020 | 0 | 0
 | 0 | 0 | 8675
(5 rows)
```

NOTE

When row-store data is imported in batches without indexes, the Xlogs related to logical new pages are generated during data page copy. If the volume of Xlogs is greater than the default value, flow control will be triggered.

pg_stat_wal_write()

Description: This function is executed on a CN or DN to record thread information on it and collect statistics on WAL logs and the volume and rate of imported data pages. The following uses a CN as an example.

This parameter is supported by version 8.2.0 or later clusters.

Return type: bigint

Fields in the returned value:

- **node_name**: instance name.
- **application_name**: application name.
- **query_start**: start time of the statement that is being executed.
- **datapage_write**: total size of data pages generated by the current query, in bytes
- **datapage_write_speed**: data page import speed, in bytes/s.
- **wal_write**: total size WAL logs imported by the current query, in bytes.
- **wal_write_speed**: WAL log import speed, in bytes/s.
- **total_datapage_write**: total size of data pages generated by the current thread, in bytes.
- **total_wal_write**: total size of WAL logs, in bytes.

SELECT * FROM pg_stat_wal_write();															
node_name	datid	pid	lwtid	usesysid	application_name	state	query_start	backend_start	query_id	datapage_write	datapage_write_speed	wal_write	wal_write_speed	total_datapage_write	total_wal_write
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----															
coordinator1	15979	1402341534	98368	25353	10 JobScheduler	active									
2022-11-15 11:57:54.396347+08				0	0		0								
0	0														
coordinator1	15979	1402341320	27136	25354	10 StatCollector	idle									
			0	0		0									0
0															
coordinator1	15979	1402340900	84096	25356	10 WDRSnapshot	idle	2022-11-15								
11:57:54.388948+08	2022-11-15 11:57:54.403055+08				0	0									0
0	0														
coordinator1	15979	1402340271	69536	25359	10 workload	active	2022-11-15								
11:57:54.404836+08	2022-11-15 11:57:54.389259+08	72620543991404472			0										0
1722112	0	0	0												
coordinator1	15979	1402340046	21056	25360	10 CalculateSpaceInfo	active	2022-11-15								
11:57:54.405677+08	2022-11-15 11:57:54.389847+08	72620543991349266			0										0
191326	0	0	0												
coordinator1	15979	1402339878	39744	25361	10 WorkloadMonitor	active	2022-11-15								
11:57:54.406263+08	2022-11-15 11:57:54.390373+08	72620543991404418			0										0
1183741	0	0	0												
coordinator1	15979	1402339710	58432	25362	10 WLMArbiter	active	2022-11-15								
11:57:54.406367+08	2022-11-15 11:57:54.390947+08				0	0									0
0	0														
coordinator1	15979	1402332286	66624	14415	10 gsql	active	2022-11-15								
21:19:32.200305+08	2022-11-15 21:09:38.916931+08	72620543991404422			0										0
30872	1000		0	0											
coordinator1	15979	1402331720	35328	15516	10 gsql	active	2022-11-15								
21:19:49.7877+08	2022-11-15 21:10:13.447312+08	72620543991404485			0										0
0	0		0												
coordinator1	0	1402342452	69248	25348	0 Background writer	idle									
2022-11-15 11:57:54.383651+08			0	0		0								0	
0	0														
coordinator1	0	1402342914	10688	25347	0 CheckPointer	idle									

```
2022-11-15 11:57:54.383231+08 |      0 |      0 |      |      0 |      0
|      0 |      0
coordinator1 | 0 | 140234228487936 | 25349 | 0 | Wal Writer | idle |      |
2022-11-15 11:57:54.384069+08 |      0 |      0 |      |      0 |      0
|      0 |      0
coordinator1 | 0 | 140234068584192 | 25357 | 0 | TwoPhase Cleaner | idle |      |
2022-11-15 11:57:54.388332+08 |      0 |      0 |      |      0 |      0
|      0 |      0
coordinator1 | 0 | 140234048657152 | 25358 | 0 | LWLock Monitor | idle |      |
2022-11-15 11:57:54.389239+08 |      0 |      0 |      |      0 |      0
|      0 |      0
coordinator1 | 0 | 140234194925312 | 25351 | 0 | CBM Writer | idle |      |
2022-11-15 11:57:54.38506+08 |      0 |      0 |      |      0 |      0
|      0 |      0
coordinator1 | 0 | 140234211706624 | 25350 | 0 | Tick service | idle |      |
2022-11-15 11:57:54.384136+08 |      0 |      0 |      |      0 |      0
|      0 |      0
(16 rows)
```

NOTE

When row-store data is imported in batches without indexes, the Xlogs related to logical new pages are generated during data page copy. If the volume of Xlogs is greater than the default value, flow control will be triggered.

gs_stack()

Description: Obtains the stack information of CN or DN processes. The input parameter **tid** needs to be obtained from views. **lwtid** is not supported. This function is supported by version 8.2.1 or later clusters.

gs_stack() provides a more flexible and quicker method to obtain the stack than **gdb** and **gstack** commands. It is recommended for administrators to use during problem analysis and locating. It is not recommended that this tool be used as a routine monitoring tool.

Fields in the returned value:

- **tid**: thread ID.
- **lwtid**: lightweight thread ID.
- **stack**: stack information corresponding to the **tid** or **lwtid** thread.

The **gs_stack()** function can be used in the following four ways:

- Method 1: If the input parameter value of the **gs_stack(tid)** function is 0, stacks of all threads in the CN or DN process connected to gsql are printed.

NOTE

- This statement can be executed by connecting to a CN or DN using gsql.
- The process of obtaining the running stacks of all threads in a CN or DN takes a long time, which affects normal service running. Therefore, you are advised not to run this command frequently.

```
SELECT * FROM gs_stack(0);
      tid      | lwtid |
stack
-----+-----
+-----+-----
140186894757888 | 95275 | __poll +
0x2d                                     +
      |      | ServerLoop() +
0x457                                     +
```


[illegible]

```

0x8bb      |      | AuxiliaryThreadUnderPm(AuxProcType) +
0x210      |      | SubPostmasterMain(tag_gs_thread_args*) +
0x7e0      |      | MainStarterThreadFunc(void*) +
0x6a       |      | ThreadStarterFunc(void*) +
0x66       |      | start_thread +
0xc5       |      | clone +
0x6d       |      |
140185903827032 | 95325 | __poll +
0x2d       |      |
0x11a      |      | GsWaiter::wait(void*, long) +
0x11a      |      | BackgroundWriterMain() +
0x541      |      | AuxiliaryThreadUnderPm(AuxProcType) +
0x210      |      | SubPostmasterMain(tag_gs_thread_args*) +
0x7e0      |      | MainStarterThreadFunc(void*) +
0x6a       |      | ThreadStarterFunc(void*) +
0x66       |      |

```

- Method 2: If the input parameter of the **gs_stack(tid)** function is set to a specified tid, the stack information of the tid thread in the CN or DN process connected using gsql is printed.

NOTE

- This statement can be executed by connecting to a CN or DN using gsql.
- tid** must be the ID of an existing thread on the connected CN or DN process. Otherwise, the error message "invalid thread id" is displayed.
- tid** is obtained from other views. **lwtid** is not supported.
- The process of obtaining the running stack of a specified thread in a specified CN or DN process takes a short time, which does not affect services. Therefore, this method is recommended.

```

SELECT * FROM gs_stack(140185903826656);
  tid  | lwtid |      stack
-----+-----+-----
140185903826656 | 95275 | __poll + 0x2d
      |      | GsWaiter::wait(void*, long) + 0x11a
      |      | CheckpointerMain() + 0x8bb
      |      | AuxiliaryThreadUnderPm(AuxProcType) + 0x210
      |      | SubPostmasterMain(tag_gs_thread_args*) + 0x7e0
      |      | MainStarterThreadFunc(void*) + 0x6a
      |      | ThreadStarterFunc(void*) + 0x66
      |      | start_thread + 0xc5
      |      | clone + 0x6d
(1 row)

```

- Method 3: If the **gs_stack()** function is set to **gs_stack('nodename', 0)**, stacks of all threads in the process specified by **nodename** are printed.

 NOTE

- The underlying implementation uses **execute direct on**. Therefore, you must use **gsql** to connect to the CN to run the command.
- The first parameter **nodename** must be enclosed in single quotation marks.
- The process of obtaining the running stacks of all threads in a CN or DN takes a long time, which affects normal service running. Therefore, you are advised not to run this command frequently.

```
SELECT * FROM gs_stack('datanode2', 0);
```

```
tid      | lwtid |
```

```
stack
```

```
-----+-----
```

```
+-----+-----
```

```
---
```

```
140634541242112 | 95442 | do_futex_wait +
```

```
0x5f
```

```
+
```

```
| | __new_sem_wait_slow +
```

```
0x57
```

```
+
```

```
| | sem_timedwait +
```

```
0x35
```

```
+
```

```
| | tagBinarySemaphore::timed_wait(int) +
```

```
0x4f
```

```
+
```

```
| | AuxiliaryThreadMain(void*) +
```

```
0x6d
```

```
+
```

```
| | LibcommThreadRoutine(void*) +
```

```
0x41
```

```
+
```

```
| | start_thread +
```

```
0xc5
```

```
+
```

```
| | clone +
```

```
0x6d
```

```
+
```

```
| |
```

```
140634549634816 | 95438 | epoll_wait +
```

```
0x33
```

```
+
```

```
| | LibcommEpollWait(int, int) +
```

```
0x38
```

```
+
```

```
| | RecvDataThreadMain(void*) +
```

```
0x8e
```

```
+
```

```
| | LibcommThreadRoutine(void*) +
```

```
0x41
```

```
+
```

```
| | start_thread +
```

```
0xc5
```

```
+
```

```
| | clone +
```

```
0x6d
```

```
+
```

```
| |
```

```
140634558027520 | 95432 | epoll_wait +
```

```
0x33
```

```
+
```

```
| | LibcommEpollWait(int, int) +
```

```
0x38
```

```
+
```

```
| | RecvDataThreadMain(void*) +
```

```
0x8e
```

```
+
```

```
| | LibcommThreadRoutine(void*) +
```

```
0x41
```

```
+
```

```
| | start_thread +
```

```
0xc5
```

```
+
```

```
| | clone +
```

```
0x6d
```

```
+
```

```
| |
```

```
140634566420224 | 95430 | epoll_wait +
```

```
0x33
```

```
+
```

```
| | LibcommEpollWait(int, int) +
```

```
0x38
```

```
+
```

```
| | RecvDataThreadMain(void*) +
```

```
0x8e
```

```
+
```

```
| | LibcommThreadRoutine(void*) +
```

```
0x41
```

```
+
```

```
| | start_thread +
```

```
0xc5
```

```
+
```

```

0x6d      |      | clone +
140634589476608 | 95426 | epoll_wait +
0x33      |      |
0x38      |      | LibcommEpollWait(int, int) +
0x8e      |      | RecvDataThreadMain(void*) +
0x41      |      | LibcommThreadRoutine(void*) +
0xc5      |      | start_thread +
0x6d      |      | clone +
140634597869312 | 95424 | do_futex_wait +
0x5f      |      | __new_sem_wait_slow +
0x57      |      | sem_timedwait +
0x35      |      | tagBinarySemaphore::timed_wait(int) +
0x4f      |      | SendDataWait() +
0x58      |      | SendDataThreadMain(void*) +
0xe4      |      | LibcommThreadRoutine(void*) +
0x41      |      | start_thread +
0xc5      |      | clone +
0x6d      |      |
140634606262016 | 95387 | do_futex_wait +
0x5f      |      | __new_sem_wait_slow +
0x57      |      | sem_timedwait +
0x35      |      | tagBinarySemaphore::timed_wait(int) +
0x4f      |      | SendDataWait() +
0x58      |      | SendDataThreadMain(void*) +
0xe4      |      |

```

- Method 4: If the **gs_stack()** function is set to **gs_stack('nodename', tid)**, stack information of the **tid** thread in the process specified by **nodename** are printed.

NOTE

- The underlying implementation uses **execute direct on**. Therefore, you must use **gsql** to connect to the CN to run the command.
- The first parameter **nodename** must be enclosed in single quotation marks.
- The second parameter must be a thread that exists in the process specified by **nodename**. Otherwise, the error message "invalid thread id" is displayed.
- tid** is obtained from other views. **lwtid** is not supported.
- The process of obtaining the running stack of a specified thread in a specified CN or DN process takes a short time, which does not affect services. Therefore, this method is recommended.

```

SELECT * FROM gs_stack('datanode2', 140634549634816);
tid      | lwtid | stack
-----+-----+-----

```

```
140634549634816 | 95438 | epoll_wait + 0x33      +
|               |      | LibcommEpollWait(int, int) + 0x38 +
|               |      | RecvDataThreadMain(void*) + 0x8e +
|               |      | LibcommThreadRoutine(void*) + 0x41+
|               |      | start_thread + 0xc5          +
|               |      | clone + 0x6d                +
(1 row)
```

pgxc_get_xlog_stats()

Description: Collects statistics on the number of Xlogs of different types from the time when each DN is started to the current time. This command is executed on CNs. This function is supported only by clusters of version 8.3.0 or later.

Return type: record

The following information is displayed:

Name	Type	Description
node_name	text	Node name
rmgr_name	text	Xlog main type
record_info	text	xlog info
record_count	bigint	Quantity
count_pct	text	Count (%)
record_size	bigint	Record size, in bytes.
record_pct	text	Percentage of the record size.
fpi_size	bigint	Full page image (FPI) size, in bytes.
fpi_pct	text	Percentage of the FPI size.
combined_size	bigint	Total size of Xlogs, in bytes.
combined_pct	text	Percentage of the total Xlog size.

NOTE

- This function collects statistics only on the number and size of Xlogs of different types since the last DN startup.
- The xlog size calculated by this function does not consider the xlog spilling. As a result, the xlog size is different from the actual xlog file size.

```
SELECT * FROM pgxc_get_xlog_stats();
node_name | rmgr_name | record_info | record_count | count_pct | record_size | record_pct | fpi_size | fpi_pct | combined_size | combined
_pct
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
datanode1 | XLOG      | checkpoint: shutdown | 0 | 0.00% | 0 | 0.00% | 0 |
```

0.00%	0	0.00%				
datanode1 XLOG	checkpoint: online	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	xlog no-op	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	nextOid	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	xlog switch	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	backup end	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	parameter change	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	restore point	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	full page writes	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	xlog recovery end	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 XLOG	page hint	1	0.12%	56	0.11%	8192
17.92%	8248	8.71%				
datanode1 XLOG	xlog fpi	0	0.00%	0	0.00%	0
0	0.00%					
datanode1 Transaction	commit	3	0.35%	246	0.50%	0
0.00%	246	0.26%				
datanode1 Transaction	prepare	1	0.12%	469	0.96%	0
0.00%	469	0.50%				
datanode1 Transaction	abort	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Transaction	commit prepared	1	0.12%	106	0.22%	0
0.00%	106	0.11%				
datanode1 Transaction	abort prepared	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Transaction	xid assignment xtop	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Transaction	commit compact	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Storage	file create	0	0.00%	0	0.00%	0
0	0.00%					
datanode1 Storage	file truncate	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Storage	vacuum clear file	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 CLOG	zero page	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 CLOG	truncate before	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Database	create db	1	0.12%	50	0.10%	0
0.00%	50	0.05%				
datanode1 Database	drop db	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Tablespace	create tablespace	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Tablespace	drop tablespace	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Tablespace	relative create	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 MultiXact	zero offsets page	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 MultiXact	zero members page	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 MultiXact	create multixact	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 RelMap	update relmap	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Standby	AccessExclusive locks	0	0.00%	0	0.00%	0
0.00%	0	0.00%				
datanode1 Standby	xlog running xacts	0	0.00%	0	0.00%	0
0.00%	0	0.00%				

datanode1 Standby	release AccessExclusive locks	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Standby	xlog standby csn	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	freeze	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Heap2	clean	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Heap2	page upgrade	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	cleanup info	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	visible	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Heap2	multi-insert	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	bcm	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	logical newpage rel	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	no repair page	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap2	multi-insert(init) base xid	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap	insert	280 32.67%	15964 32.63%	14314
31.31% 30278 31.99%				
datanode1 Heap	delete	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Heap	update	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap	base shift	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Heap	hot update	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap	new page	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap	lock type	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Heap	inplace	2 0.23%	116 0.24%	8164
17.86% 8280 8.75%				
datanode1 Heap	init page	2 0.23%	130 0.27%	92
0.20% 222 0.23%				
datanode1 Heap	insert(init) base xid	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Btree	insert leaf	562 65.58%	31484 64.35%	13352
29.21% 44836 47.37%				
datanode1 Btree	insert upper	1 0.12%	72 0.15%	148
0.32% 220 0.23%				
datanode1 Btree	insert meta	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Btree	split left	1 0.12%	78 0.16%	744 1.63%
822 0.87%				
datanode1 Btree	split right	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Btree	split left root	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Btree	split right root	1 0.12%	78 0.16%	672
1.47% 750 0.79%				
datanode1 Btree	delete	0 0.00%	0 0.00%	0 0.00%
0 0.00%				
datanode1 Btree	delete page	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Btree	delete page meta	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Btree	lev	1 0.12%	78 0.16%	40 0.09%
118 0.12%				
datanode1 Btree	delete page half	0 0.00%	0 0.00%	0
0.00% 0 0.00%				
datanode1 Btree	vacuum	0 0.00%	0 0.00%	0

0.00%		0 0.00%					
datanode1 Btree	reuse page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Create index			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Create ptree			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Insert item			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Gin	Page split			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Gin	Vacuum page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Delete page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Update metapage			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Insert new list page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Delete list pages			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gin	Vacuum data leaf page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gist	page update			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gist	page split			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Gist	Create index			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Gist	page delete			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Sequence	log			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 SPGist	create index			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	add leaf to page			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	move leafs			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	add node			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	prefix off			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 SPGist	picksplit			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 SPGist	vacuum leaf			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	vacuum root			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 SPGist	newest XID			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Slot	create slot			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Slot	advance slot			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Slot	drop slot			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Slot	check slot			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
datanode1 Heap3	new cid			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Heap3	heap rewrite			0 0.00%		0 0.00%	0
0.00%		0 0.00%					
datanode1 Barrier	barrier			0 0.00%		0 0.00%	0 0.00%
	0 0.00%						
(102 rows)							

pgxc_get_wal_speed()

Description: Obtains the WAL generation rate of each DN and the receive, write, flush, and redo rates of standby DNs. This function is supported only by clusters of version 8.3.0 or later.

Return type: record

The following information is displayed:

Name	Type	Description
node_name	text	Node name.
pid	bigint	PID of a thread.
local_role	text	Role of the current node
peer_role	text	Role of the recipient
peer_state	text	Status of the recipient
state	text	Transmission status
generate_speed	bigint	Generation rate of the primary DN wal, in byte/s.
receive_speed	bigint	Receiving rate of the standby DN. The unit is byte/s.
write_speed	bigint	Write speed of the standby DN. The unit is byte/s.
flush_speed	bigint	Disk flushing rate of the standby DN. The unit is byte/s.
redo_speed	bigint	Redo speed of the standby DN. The unit is byte/s.

```
SELECT * FROM pgxc_get_wal_speed();
node_name | pid | local_role | peer_role | peer_state | state | generate_speed | receive_speed |
write_speed | flush_speed | apply_speed
-----+-----+-----+-----+-----+-----+-----+-----+-----
datanode1 | 140240389260824 | Primary | Secondary | Normal | Streaming | 0 | 0
| 0 | 0 | 0
datanode1 | 140240389262192 | Primary | Standby | Normal | Streaming | 0 | 0
| 0 | 0 | 0
(2 rows)
```

pg_xlog_display_one_lsn(start_lsn)

Description: This command can be executed on CNs or DNs to completely parse Xlogs in the current location based on the start LSN. This function is supported only by clusters of version 8.3.0 or later.

Input parameter: **start_lsn**, indicating the start LSN. Ensure that the entered start LSN is the beginning of an Xlog.

Return type: record

The following information is displayed:

Name	Type	Description
node_name	text	Current instance name.
start_lsn	text	Start LSN.
end_lsn	text	End LSN.
startlsn	text	Start LSN of the Xlog.
endlsn	text	End LSN of the Xlog.
prelsn	text	Start LSN of the previous Xlog.
xid	xid	Xlog transaction ID.
datalen	int4	Xlog data length, in bytes.
totalen	int4	Xlog length, in bytes.
type	text	Xlog type.
desc	text	Xlog content.
blkref	text	Relfilenode associated with the Xlog.

Example:

```
SELECT * FROM pg_xlog_display_one_lsn('0/101CA00');
node_name | startlsn | endlsn | prelsn | xlog_tid | datalen | totalen | xlogtype |
xlogdescribe
          | blkref
-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----
datanode1 | 0/101CA00 | 0/101CA78 | 0/101C3F8 |      5 |    80 |   114 | Transaction | commit:
2023-10-19 22:21:38.617092+08; cs:0; inval msgs: catcache 11 ca
tcache 10 |
(1 row)
```

pg_xlogdump (tablename)

Description: Parses and filters Xlog files based on table names on CNs or DN. This function is supported only by clusters of version 8.3.0 or later.

Return type: record

The following information is displayed:

Name	Type	Description
node_name	text	Node name.
start_lsn	text	Start LSN.
end_lsn	text	End LSN.
startlsn	text	Start LSN of the Xlog.
endlsn	text	End LSN of the Xlog.
prelsn	text	Start LSN of the previous Xlog.
xid	xid	Xlog transaction ID.
datalen	int4	Xlog data length, in bytes.
totalen	int4	Xlog length, in bytes.
type	text	Xlog type.
desc	text	Xlog content.
blkref	text	Relfilenode associated with the Xlog.

NOTE

- The length of the **xlogdescribe** field varies depending on the Xlog type. The **pg_xlogdump()** function truncates the field and retains only the first 64 bytes.
- You can find the target Xlog and use the **pg_xlog_display_one_lsn()** function to view the complete Xlog content.

Example:

```
SELECT * FROM pg_xlogdump(pg_class);
 node_name | startlsn | endlsn | prelsn | xlog_tid | datalen | totalen | xlogtype | xlogdescribe
 |          |         |        |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+-----
 datanode1 | 0/2DFC660 | 0/2DFE1D0 | 0/2DFA610 |      0 |      2 |    7000 | Heap    | inplace: off 17 |
 blkrel #0: rel 1663/16324/15920, fork main, blk 2
 datanode1 | 0/2E02270 | 0/2E02D88 | 0/2E00220 |      0 |      2 |    2840 | Heap    | inplace: off 2 | blkrel
 #0: rel 1663/16324/15920, fork main, blk 0
 datanode1 | 0/2E06E60 | 0/2E06F68 | 0/2E06E28 |      0 |      2 |     264 | Heap    | inplace: off 10 | blkrel
 #0: rel 1663/16324/15920, fork main, blk 0
 datanode1 | 0/2E0FC08 | 0/2E0FD10 | 0/2E0EBB8 |      0 |      2 |     264 | Heap    | inplace: off 13 | blkrel
```

```
#0: rel 1663/16324/15920, fork main, blk 0
datanode1 | 0/2E12960 | 0/2E12A68 | 0/2E11960 | 0 | 2 | 264 | Heap | inplace: off 15 | blkrel
#0: rel 1663/16324/15920, fork main, blk 0
datanode1 | 0/2E12C88 | 0/2E12D90 | 0/2E12B78 | 0 | 2 | 264 | Heap | inplace: off 17 | blkrel
#0: rel 1663/16324/15920, fork main, blk 0
datanode1 | 0/2E13FF8 | 0/2E14118 | 0/2E13110 | 0 | 2 | 264 | Heap | inplace: off 18 | blkrel
#0: rel 1663/16324/15920, fork main, blk 0
datanode1 | 0/2E169F8 | 0/2E185E0 | 0/2E169B0 | 0 | 2 | 7120 | Heap | inplace: off 1 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
datanode1 | 0/2E188B8 | 0/2E189D0 | 0/2E185E0 | 0 | 2 | 280 | Heap | inplace: off 2 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
datanode1 | 0/2E18BF8 | 0/2E18CE0 | 0/2E189D0 | 0 | 2 | 232 | Heap | inplace: off 28 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
datanode1 | 0/2E18E80 | 0/2E18F88 | 0/2E18DC0 | 0 | 2 | 264 | Heap | inplace: off 4 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
datanode1 | 0/2E19160 | 0/2E19268 | 0/2E19088 | 0 | 2 | 264 | Heap | inplace: off 5 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
datanode1 | 0/2E19460 | 0/2E1B1D0 | 0/2E19388 | 0 | 2 | 7508 | Heap | inplace: off 16 | blkrel
#0: rel 1663/16324/15920, fork main, blk 6
datanode1 | 0/2E1B1D0 | 0/2E1B2E8 | 0/2E19460 | 0 | 2 | 280 | Heap | inplace: off 18 | blkrel
#0: rel 1663/16324/15920, fork main, blk 6
datanode1 | 0/2E1B2E8 | 0/2E1B3D0 | 0/2E1B1D0 | 0 | 2 | 232 | Heap | inplace: off 30 | blkrel
#0: rel 1663/16324/15920, fork main, blk 1
(15 rows)
```

pg_xlogdump (xid)

Description: Parses and filters Xlogs based on transaction IDs. This statement can be executed on CNs or DN. This function is supported only by clusters of version 8.3.0 or later.

Return type: record

The following information is displayed:

Name	Type	Description
node_name	text	Node name.
start_lsn	text	Start LSN.
end_lsn	text	End LSN.
startlsn	text	Start LSN of the Xlog.
endlsn	text	End LSN of the Xlog.
prelsn	text	Start LSN of the previous Xlog.
xid	xid	Xlog transaction ID.
datalen	int4	Xlog data length, in bytes.
totallen	int4	Xlog length, in bytes.
type	text	Xlog type.
desc	text	Xlog content.

Name	Type	Description
blkref	text	Relfilenode associated with the Xlog.

 NOTE

- The length of the **xlogdescribe** field varies depending on the Xlog type. The **pg_xlogdump()** function truncates the field and retains only the first 64 bytes.
- You can find the target Xlog and use the **pg_xlog_display_one_lsn()** function to view the complete Xlog content.

Example:

```
SELECT * FROM pg_xlogdump(1);
node_name | startlsn | endlsn | prelsn | xlog_tid | datalen | totalen | xlogtype | xlogdescribe
|         |         |         |         |          |         |         |          |
-----+-----+-----+-----+-----+-----+-----+-----+-----
datanode1 | 0/1000A0 | 0/10020F0 | 0/1000028 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/1255, fork fsm, blk 2
datanode1 | 0/10020F0 | 0/1004140 | 0/10000A0 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/1247, fork fsm, blk 2
datanode1 | 0/1004140 | 0/1004168 | 0/10020F0 | 1 | 4 | 38 | XLOG | nextOid: 18192 |
datanode1 | 0/1004168 | 0/10061B8 | 0/1004140 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2616, fork fsm, blk 2
datanode1 | 0/10061B8 | 0/1008208 | 0/1004168 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2602, fork fsm, blk 2
datanode1 | 0/1008208 | 0/100A258 | 0/10061B8 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2603, fork fsm, blk 2
datanode1 | 0/100A258 | 0/100C2A8 | 0/1008208 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2609, fork fsm, blk 2
datanode1 | 0/100C2A8 | 0/100E2F8 | 0/100A258 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2605, fork fsm, blk 2
datanode1 | 0/100E2F8 | 0/1010348 | 0/100C2A8 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/1259, fork fsm, blk 2
datanode1 | 0/1010348 | 0/1012398 | 0/100E2F8 | 1 | 0 | 8248 | XLOG | page hint | blkrel
#0: rel 1663/1/2610, fork fsm, blk 2
(10 rows)
```

pg_xlogdump (start_lsn, end_lsn)

Description: Parses Xlog files based on the start and end LSNs. This command can be executed on CNs or DN. This function is supported only by clusters of version 8.3.0 or later.

Input parameters: **start_lsn** indicates the start LSN, and **end_lsn** indicates the end LSN. You do not need to ensure that the start LSN is the first Xlog. The parsing starts from the first valid Xlog after the start LSN.

Return type: record

The following information is displayed:

Field	Type	Description
node_name	text	Node name.
start_lsn	text	Start LSN.

Field	Type	Description
end_lsn	text	End LSN.
startlsn	text	Start LSN of the Xlog.
endlsn	text	End LSN of the Xlog.
prelsn	text	Start LSN of the previous Xlog.
xid	xid	Xlog transaction ID.
datalen	int4	Xlog data length, in bytes.
totalen	int4	Xlog length, in bytes.
type	text	Xlog type.
desc	text	Xlog content.
blkref	text	Relfilenode associated with the Xlog.

NOTE

- The length of the **xlogdescribe** field varies depending on the Xlog type. The **pg_xlogdump()** function truncates the field and retains only the first 64 bytes.
- You can find the target Xlog and use the **pg_xlog_display_one_lsn()** function to view the complete Xlog content.

Example:

```
SELECT * FROM pg_xlogdump('0/101EF70','0/101F590');
 node_name | startlsn | endlsn | prelsn | xlog_tid | datalen | totalen | xlogtype | xlogdescribe
 |          |        |        |        |          |         |         |          |
-----+-----+-----+-----+-----+-----+-----+-----+-----
 datanode1 | 0/101EF70 | 0/101EFC0 | 0/101EF18 |      6 |      2 |      80 | Btree   | insert leaf: off 1 |
 blkrel #0: rel 1663/1/2674, fork main, blk 1
 datanode1 | 0/101EFC0 | 0/101F018 | 0/101EF70 |      6 |      3 |      88 | Heap    | insert: off 2 flags 0 |
 blkrel #0: rel 1663/1/2608, fork main, blk 0
 datanode1 | 0/101F018 | 0/101F068 | 0/101EFC0 |      6 |      2 |      80 | Btree   | insert leaf: off 1 |
 blkrel #0: rel 1663/1/2673, fork main, blk 1
 datanode1 | 0/101F068 | 0/101F0B8 | 0/101F018 |      6 |      2 |      80 | Btree   | insert leaf: off 2 |
 blkrel #0: rel 1663/1/2674, fork main, blk 1
 datanode1 | 0/101F0B8 | 0/101F110 | 0/101F068 |      6 |      3 |      88 | Heap    | insert: off 3 flags 0 |
 blkrel #0: rel 1663/1/2608, fork main, blk 0
 datanode1 | 0/101F110 | 0/101F160 | 0/101F0B8 |      6 |      2 |      80 | Btree   | insert leaf: off 1 |
 blkrel #0: rel 1663/1/2673, fork main, blk 1
 datanode1 | 0/101F160 | 0/101F1B0 | 0/101F110 |      6 |      2 |      80 | Btree   | insert leaf: off 3 |
 blkrel #0: rel 1663/1/2674, fork main, blk 1
 datanode1 | 0/101F1B0 | 0/101F208 | 0/101F160 |      6 |      3 |      88 | Heap    | insert: off 4 flags 0 |
 blkrel #0: rel 1663/1/2608, fork main, blk 0
 datanode1 | 0/101F208 | 0/101F258 | 0/101F1B0 |      6 |      2 |      80 | Btree   | insert leaf: off 1 |
 blkrel #0: rel 1663/1/2673, fork main, blk 1
 datanode1 | 0/101F258 | 0/101F2A8 | 0/101F208 |      6 |      2 |      80 | Btree   | insert leaf: off 4 |
 blkrel #0: rel 1663/1/2674, fork main, blk 1
 datanode1 | 0/101F2A8 | 0/101F300 | 0/101F258 |      6 |      3 |      88 | Heap    | insert: off 5 flags 0 |
 blkrel #0: rel 1663/1/2608, fork main, blk 0
```

```
datanode1 | 0/101F300 | 0/101F350 | 0/101F2A8 | 6 | 2 | 80 | Btree | insert leaf: off 1 |
blkrel #0: rel 1663/1/2673, fork main, blk 1
datanode1 | 0/101F350 | 0/101F3A0 | 0/101F300 | 6 | 2 | 80 | Btree | insert leaf: off 5 |
blkrel #0: rel 1663/1/2674, fork main, blk 1
datanode1 | 0/101F3A0 | 0/101F3F8 | 0/101F350 | 6 | 3 | 88 | Heap | insert: off 6 flags 0 |
blkrel #0: rel 1663/1/2608, fork main, blk 0
datanode1 | 0/101F3F8 | 0/101F448 | 0/101F3A0 | 6 | 2 | 80 | Btree | insert leaf: off 1 |
blkrel #0: rel 1663/1/2673, fork main, blk 1
datanode1 | 0/101F448 | 0/101F498 | 0/101F3F8 | 6 | 2 | 80 | Btree | insert leaf: off 5 |
blkrel #0: rel 1663/1/2674, fork main, blk 1
datanode1 | 0/101F498 | 0/101F4F0 | 0/101F448 | 6 | 3 | 88 | Heap | insert: off 7 flags 0 |
blkrel #0: rel 1663/1/2608, fork main, blk 0
datanode1 | 0/101F4F0 | 0/101F540 | 0/101F498 | 6 | 2 | 80 | Btree | insert leaf: off 1 |
blkrel #0: rel 1663/1/2673, fork main, blk 1
datanode1 | 0/101F540 | 0/101F590 | 0/101F4F0 | 6 | 2 | 80 | Btree | insert leaf: off 6 |
blkrel #0: rel 1663/1/2674, fork main, blk 1
(19 rows)
```

7.32 Trigger Functions

pg_get_triggerdef(oid)

Description: Obtains the definition information of a trigger.

Parameter: OID of the trigger to be queried

Return type: text

Example:

```
SELECT pg_get_triggerdef(oid) FROM pg_trigger;
          pg_get_triggerdef
-----
CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE
PROCEDURE tri_insert_func()
(1 row)
```

pg_get_triggerdef(oid, boolean)

Description: Obtains the definition information of a trigger.

Parameter: OID of the trigger to be queried and whether it is displayed in pretty mode

Return type: text

NOTE

The Boolean parameters take effect only when the WHEN condition is specified during trigger creation.

Example:

```
SELECT pg_get_triggerdef(oid,true) FROM pg_trigger;
          pg_get_triggerdef
-----
CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE
PROCEDURE tri_insert_func()
(1 row)

SELECT pg_get_triggerdef(oid,false) FROM pg_trigger;
          pg_get_triggerdef
-----
```

```
CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE  
PROCEDURE tri_insert_func()  
(1 row)
```

7.33 XML Functions

7.33.1 Generating XML Content

Expressions of functions and class functions in this section can be used to generate XML content from SQL data. This method is used to format query results into XML documents for processing in client applications.

XMLPARSE ({ DOCUMENT | CONTENT } *value*)

Description: Generates an XML value from character data.

Return type: XML

Example:

```
SELECT xmlparse(document '<foo>bar</foo>');  
xmlparse  
-----  
<foo>bar</foo>  
(1 row)
```

XMLSERIALIZE ({ DOCUMENT | CONTENT } *value AS type*)

Description: Generates a string from XML values.

Return type: type, which can be character, character varying, or text (or its alias)

Example:

```
SELECT xmlserialize(content 'good' AS CHAR(10));  
xmlserialize  
-----  
good  
(1 row)
```

xmlcomment(text)

Description: Creates an XML note that uses the specified text as the content. The text cannot contain (--) or end with a hyphen (-). If the parameter is null, the result is also null.

Return type: XML

Example:

```
SELECT xmlcomment('hello');  
xmlcomment  
-----  
<!--hello-->  
(1 row)
```


xmlconcat(xml[, ...])

Description: Concatenates a list of XML values into a single value. Null values are ignored. If all parameters are null, the result is also null.

Return type: XML

Example:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');
xmlconcat
-----
<abc/><bar>foo</bar>
(1 row)
```

Note: If XML declarations exist and they are the same XML version, the result will use the version. Otherwise, the result does not use any version. If all XML values have the **standalone** attribute whose status is **yes**, the **standalone** attribute in the result is **yes**. If at least one XML value's **standalone** attribute is **no**, the **standalone** attribute in the result is **no**. Otherwise, the result does not contain the **standalone** attribute.

Example:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');
xmlconcat
-----
<?xml version="1.1"?><foo/><bar/>
(1 row)
```

xmlelement(name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...])

Description: Generates an XML element with the given name, attribute, and content.

Return type: XML

Example:

```
SELECT xmlelement(name foo);
xmlelement
-----
<foo/>
(1 row)

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));
xmlelement
-----
<foo bar="xyz"/>
(1 row)

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');
xmlelement
-----
<foo bar="2023-08-16">content</foo>
(1 row)
```

Element and attribute names without valid XML names are escaped by replacing invalid characters with the sequence **_xHHHH_**, where **HHHH** is the Unicode code points of the character expressed in hexadecimal format. Example:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));
```

```
xmlelement
-----
<foo_x0024_bar a_x0026_b="xyz"/>
```

If the attribute value is a column reference, you do not need to specify an explicit attribute name, and the column name is used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is legal:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

But these are illegal:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

The element content, if specified, will be formatted based on its data type. If the content itself is of the XML type, a complex XML document will be constructed. Example:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),xmlelement(name
abc),xmlcomment('test'),xmlelement(name xyz));
xmlelement
-----
<foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Other types of content will be formatted into valid XML character data. This means that special characters `<`, `>`, and `&` will be converted to entities. Binary data (data type `bytea`) is represented as Base64 or hexadecimal code, depending on the configuration parameter `xmlbinary`.

xmlforest(content [AS name] [, ...])

Description: Generates an XML forest (sequence) of an element with a given name and content.

Return type: XML

Example:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);
xmlelement
-----
<foo>abc</foo><bar>123</bar>
(1 row)

SELECT xmlforest(table_name, column_name) FROM ALL_TAB_COLUMNS WHERE schema = 'pg_catalog';
xmlelement
-----
<table_name>pg_authid</table_name><column_name>rolsuper</column_name>
<table_name>pg_authid</table_name><column_name>rolinherit</column_name>
<table_name>pg_authid</table_name><column_name>rolcreatorole</column_name>
```

xmlpi(name target [, content])

Description: Creates an XML processing instruction. The content cannot contain the character sequence of `?>`.

Return type: XML

Example:

```
SELECT xmlpi(name php, 'echo "hello world";');
xmlpi
-----
```

```
-----  
<?php echo "hello world";?>  
(1 row)
```

xmlroot(xml, version text | no value [, standalone yes|no|no value])

Description: Modifies the attributes of the root node of an XML value. If a version is specified, it replaces the value in the version declaration of the root node. If a **standalone** value is specified, it replaces the **standalone** value in the root node.

Return type: XML

Example:

```
SELECT xmlroot(xmlparse(document '<?xml version="1.0" standalone="no"?><content>abc</content>'),  
              version '1.1', standalone yes);  
              xmlroot  
-----  
<?xml version="1.1" standalone="yes"?><content>abc</content>  
(1 row)
```

xmlagg(xml)

Description: The **xmlagg** function is an aggregate function that concatenates input values.

Return type: XML

Example:

```
CREATE TABLE test (y int, x xml);  
INSERT INTO test VALUES (1, '<foo>abc</foo>');  
INSERT INTO test VALUES (2, '<bar/>');  
  
SELECT xmlagg(x) FROM test;  
              xmlagg  
-----  
<bar/><foo>abc</foo>  
(1 row)
```

Add an **ORDER BY** clause to an aggregate call to determine the concatenation sequence. For example:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;  
              xmlagg  
-----  
<bar/><foo>abc</foo>  
(1 row)
```

7.33.2 XML Predicates

The functions in this section check the attributes of an XML value.

xml IS DOCUMENT

Description: IS DOCUMENT returns true if the XML value of the parameter is a correct XML document; if the XML document is incorrect, false is returned. If the parameter is null, a null value is returned.

Return type: bool

```
SELECT '<abc/>' is document;  
?column?
```

```
-----  
t  
(1 row)
```

xml IS NOT DOCUMENT

Description: Returns **true** if the XML value of the parameter is not a correct XML document. If the XML document is correct, **false** is returned. If the parameter is null, a null value is returned.

Return type: bool

```
SELECT 'abc' is document;  
?column?  
-----  
f  
(1 row)
```

XMLEXISTS(text PASSING [BY REF] xml [BY REF])

Description: If the **xpath** expression in the first parameter returns any node, the **XMLEXISTS** function returns true. Otherwise, the function returns false. (If any parameter is null, the result is null.) The BY REF clause is invalid and is used to maintain SQL compatibility.

Return type: bool

Example:

```
SELECT xmlexists('//*[text() = "TScity"]' PASSING BY REF '<towns><town>TScity</town><town>TOcity</town></towns>');  
xmlexists  
-----  
t  
(1 row)
```

xml_is_well_formed(text)

Description: Checks whether a text string is a well-formatted XML value and returns a Boolean result. If the **xmloption** parameter is set to **DOCUMENT**, the document is checked. If the **xmloption** parameter is set to **CONTENT**, the content is checked.

Return type: bool

Example:

```
SET xmloption TO DOCUMENT;  
SET  
  
SELECT xml_is_well_formed('<>');  
xml_is_well_formed  
-----  
f  
(1 row)  
  
SELECT xml_is_well_formed('<abc/>');  
xml_is_well_formed  
-----  
t  
(1 row)  
  
SET xmloption TO CONTENT;
```

```
SELECT xml_is_well_formed('abc');
xml_is_well_formed
-----
t
(1 row)
```

xml_is_well_formed_document(text)

Description: Checks whether a text string is a well-formatted text and returns a Boolean result.

Return type: bool

Example:

```
SELECT xml_is_well_formed_document('<test:foo xmlns:test="http://test.com/test">bar</test:foo>');
xml_is_well_formed
-----
t
(1 row)

SELECT xml_is_well_formed_document('<test:foo xmlns:test="http://test.com/test">bar</my:foo>');
xml_is_well_formed_document
-----
f
(1 row)
```

xml_is_well_formed_content(text)

Description: Checks whether a text string is a well-formatted content and returns a Boolean result.

Return type: bool

Example:

```
SELECT xml_is_well_formed_content('content');
xml_is_well_formed_content
-----
t
(1 row)

SELECT xml_is_well_formed_content('<content');
xml_is_well_formed_content
-----
f
(1 row)
```

7.33.3 Processing XML

To process values of the XML data type, DWS provides the **xpath** and **xpath_exists** functions, as well as the **XMLTABLE** table function.

xpath(xpath, xml [, nsarray])

Description: Returns an array of XML values corresponding to the set of nodes produced by the **xpath** expression. If the **xpath** expression returns a scalar value instead of a set of nodes, an array of individual elements is returned.

The second parameter **xml** must be a complete XML document, which must have a root node element.

The third parameter is optional and is an array mapping of a namespace. The array should be a two-dimensional text array, and the length of the second dimension should be 2. (It should be an array of arrays, each containing exactly two elements). The first element of each array item is the alias of the namespace name, and the second element is the namespace URI. The alias provided in this array does not have to be the same as the alias used in the XML document itself. In other words, in the context of both XML documents and **xpath** functions, aliases are local.

Return type: XML value array

Example:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>', ARRAY[ARRAY['my', 'http://example.com']]);
xpath
-----
{test}
(1 row)
```

Process the default (anonymous) namespace:

```
SELECT xpath('/mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>', ARRAY[ARRAY['mydefns', 'http://example.com']]);
xpath
-----
{test}
(1 row)
```

xpath_exists(xpath, xml [, nsarray])

Description: The **xpath_exists** function is a special form of the **xpath** function. This function does not return an XML value that satisfies the **xpath** function; it returns a Boolean value indicating whether the query is satisfied. This function is equivalent to the standard **XMLEXISTS** predicate, but it also provides support for a namespace mapping parameter.

Return type: bool

Example:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>', ARRAY[ARRAY['my', 'http://example.com']]);
xpath_exists
-----
t
(1 row)
```

xmltable

Description: Generates a table based on the input XML data, **XPath** expression, and column definition. An **xmltable** is similar to a function in syntax, but it can appear only as a table in the FROM clause of a query.

Return value: setof record

Syntax:

```
XMLTABLE ( [ XMLNAMESPACES ( namespace_uri AS namespace_name [ , ... ] ), ]
          row_expression PASSING [ BY { REF | VALUE } ]
          document_expression [ BY { REF | VALUE } ]
          COLUMNS name { type [ PATH column_expression ] [ DEFAULT default_expression ] [ NOT NULL | NULL ] | FOR ORDINALITY }
```

```
[, ...]  
)
```

Parameter:

- The optional XMLNAMESPACES clause is a comma-separated list of namespace definitions, where each **namespace_uri** is a text-type expression and each **namespace_name** is a simple identifier. XMLNAMESPACES specifies the XML namespaces used in the document and their aliases. The default namespace declaration is not supported.
- The mandatory parameter **row_expression** is an **XPath** 1.0 expression. This expression calculates the sequence of XML nodes based on the provided XML document **document_expression**. The sequence is the sequence of converting **xmltable** to output lines. If the **document_expression** value is **NULL** or an empty node set generated by **row_expression**, no line is returned.
- The **document_expression** parameter is used to input an XML document. The input document must be in the XML format. XML segment data or XML documents in incorrect formats are not supported. The BY REF and BY VALUE clauses do not take effect. They are used only to implement SQL standard compatibility.
- The COLUMNS clause specifies the column list definition in the output table. The column name and column data type are mandatory, and the path, default value, and whether the clause is empty are optional.
 - **column_expression** of a column is an **XPath** 1.0 expression used to calculate the value of the column extracted from the current row based on **row_expression**. If **column_expression** is not specified, the field name is used as an implicit path.
 - A column can be marked as **NOT NULL**. If **column_expression** in the **NOT NULL** column does not return any data, and there is no **DEFAULT** clause or the calculation result of **default_expression** is **NULL**, an error is reported.
 - The columns marked as **FOR ORDINALITY** are filled with row numbers starting from **1**. The sequence is the node sequence retrieved from the **row_expression** result set. A maximum of one column can be marked as **FOR ORDINALITY**.

NOTICE

XPath 1.0 does not specify the order for nodes, so the order in which results are returned depends on the order in which data is obtained.

Example:

```
SELECT * FROM XMLTABLE('/ROWS/ROW'  
PASSING '<ROWS><ROW id="1"><CITY_ID>1002a</CITY_ID><CITY_NAME>snowcity</CITY_NAME></  
ROW><ROW id="2"><CITY_ID>1003b</CITY_ID><CITY_NAME>icecity</CITY_NAME></ROW><ROW  
id="3"><CITY_ID>1004c</CITY_ID><CITY_NAME>windcity</CITY_NAME></ROW></ROWS>'  
COLUMNS id INT PATH '@id',  
ordinality FOR ORDINALITY,  
CITY_id TEXT PATH 'CITY_ID',CITY_name TEXT PATH 'CITY_NAME' NOT NULL);  
id | ordinality | city_id | city_name  
-----+-----+-----+-----  
1 | 1 | 1002a | snowcity  
2 | 2 | 1003b | icecity
```

```
3 |      3 | 1004c | windcity  
(3 rows)
```

7.33.4 Mapping a Table to XML

table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)

Description: Maps the contents of a table to XML values.

Return type: XML

table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

Description: Maps a relational table schema to an XML schema document.

Return type: XML

table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

Description: Maps a relational table to XML values and schema documents.

Return type: XML

query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)

Description: Maps the contents of an SQL query to XML values.

Return type: XML

query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

Description: Maps an SQL query into an XML schema document.

Return type: XML

query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

Description: Maps SQL queries to XML values and schema documents.

Return type: XML

cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text)

Description: Maps a cursor query to an XML value.

Return type: XML

cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)

Description: Maps a cursor query to an XML schema document.

Return type: XML

schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)

Description: Maps a table in a schema to an XML value.

Return type: XML

schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

Description: Maps a table in a schema to an XML schema document.

Return type: XML

schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

Description: Maps a table in a schema to an XML value and a schema document.

Return type: XML

database_to_xml(nulls boolean, tableforest boolean, targetns text)

Description: Maps a database table to an XML value.

Return type: XML

database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)

Description: Maps a database table to an XML schema document.

Return type: XML

database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)

Description: Maps database tables to XML values and schema documents.

Return type: XML

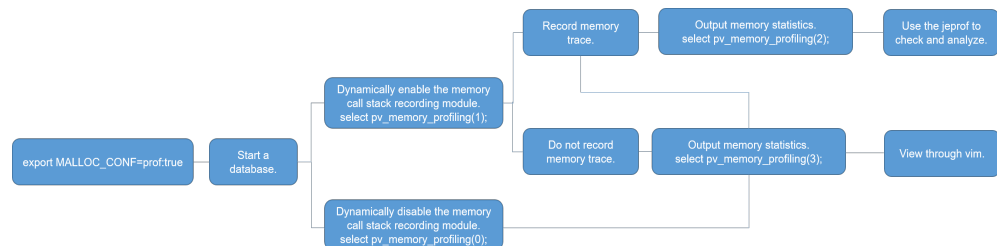
NOTE

The parameters for mapping a table to an XML value are described as follows:

- **tbl**: table name.
- **nulls**: indicates whether the output contains null values. If the value is **true**, the null value in the column is `<columnname xsi:nil="true"/>`. If the value is **false**, the columns containing null values are omitted from the output.
- **tableforest**: If this parameter is set to **true**, XML fragments are generated. If this parameter is set to **false**, XML files are generated.
- **targetns**: specifies the XML namespace of the desired result. If this parameter is not specified, an empty string is passed.
- **query**: SQL query statement
- **cursor**: cursor name
- **count**: amount of data obtained from the cursor
- **schema**: schema name

7.34 Call Stack Recording Functions

The **pv_memory_profiling(type int)** and environment variable **MALLOC_CONF** are used by DWS to control the enabling and disabling of the memory allocation call stack recording module and the output of the process-level memory call stack. The following figure illustrates the process.



MALLOC_CONF

The environment variable **MALLOC_CONF** is used to enable the monitoring module. It is in the `${BIGDATA_HOME}/mppdb/.mppdbgs_profile` file and is enabled by default. Note the following points:

- Restart the database after modifying this environment variable.
- If **om_monitor** is enabled in the cluster, restart the **om_monitor** process and then the database after setting this environment variable, so that the setting can take effect.
- This environment variable can be set on all servers in the cluster or on some servers where the module needs to be enabled. For the GaussDB process, each process determines whether to enable the module based on the **MALLOC_CONF** environment variable.

Commands for enabling and disabling **MALLOC_CONF**:

- Enabling the monitoring module:
`export MALLOC_CONF=prof:true`
- Disabling the monitoring module:
`export MALLOC_CONF=prof:false`

pv_memory_profiling (type int)

Parameter description: Controls the backtrace recording and output of memory allocation functions such as **malloc** in the gaussdb process.

Value range: a positive integer from 0 to 3.

Table 7-54 Values and descriptions of **pv_memory_profile**

pv_memory_profiling Value	Description
0	Disables the memory trace function and does not record information of call stacks such as malloc .
1	Enables the memory trace function to record information of call stacks such as malloc .
2	Outputs trace logs of call stacks such as malloc . • Output path: /proc/pid/cwd directory. <i>pid</i> indicates the ID of the GaussDB process. • Output log name format: jeprof.<pid>.*.heap, where <i>pid</i> indicates the ID of the GaussDB process and <i>*</i> indicates the unique sequence number of the output trace log, for example, jeprof.195473.0.u0.heap.
3	Outputs memory statistics. • Output path: /proc/pid/cwd directory. <i>pid</i> indicates the ID of the GaussDB process. • Log name format: <i>Node name + Process ID + Time + heap_stats + .out</i>. You can use vim to open the file.

Return type: Boolean

Note:

- If the function is called successfully, **true** is returned. Otherwise, **false** is returned.
- If **Memory profiling failed, check if \$MALLOC_CONF contain 'prof:true'**. is displayed, it indicates that the module is used when **MALLOC_CONF=prof:true** is not set. In this case, you need to set the environment variable.
- If **Type %d is not supported. The valid range is 0-3.** is displayed, the parameter value is incorrect. The correct values are **0**, **1**, **2**, and **3**.
- If "Memory profiling failed, input type is %d, failed number is %d." is displayed, contact technical support.

Outputting Memory Call Stack Information

Procedure:

Step 1 Execute the following statement to output the memory call stack information and output the **trace** file in the directory where the GaussDB process is located:

```
SELECT * FROM pv_memory_profiling(2);
```

Step 2 Use the jeprof tool provided by jemalloc to parse log information.

Method 1: Output in text format.

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 >prof.txt
```

Method 2: Export the report in PDF format.

```
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 > prof.pdf
```

NOTE

- To parse the memory call stack information, you need to use the GaussDB source code for analysis. You need to send the **trace** file to R&D engineers for analysis.
- To analyze the **trace** file, you need to use the jeprof tool, which is generated by jemalloc. The Perl environment is required for using the tool. To generate PDF calling diagrams, you need to install the GraphViz tool that matches the OS.

----End

Example

Log in as the system administrator, set environment variables, and start the database.

```
export MALLOC_CONF=prof:true
```

Disable the memory trace recording function when the database is running.

```
postgres=#SELECT pv_memory_profiling(0);
pv_memory_profiling
-----
t
(1 row)
```

Enable the memory trace recording function when the database is running.

```
postgres=#SELECT pv_memory_profiling(1);
pv_memory_profiling
-----
t
(1 row)
```

Output memory trace records.

```
postgres=#SELECT pv_memory_profiling(2);
pv_memory_profiling
-----
t
```

(1 row)

Generate the trace file in text or PDF format in the directory where the GaussDB process is located.

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 >prof.txt
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 > prof.pdf
```

Outputs memory statistics. Execute the following statement to generate the memory statistics file in the directory where the GaussDB process is located. The file can be directly read.

```
postgres=#SELECT pv_memory_profiling(3);
pv_memory_profiling
-----
t
(1 row)
```

7.35 Hudi System Functions

Hudi system functions are supported only by 9.1.0.100 and later cluster versions.

pg_show_custom_settings()

Description: Queries details about Hudi foreign table parameter settings.

Return type: SETOF record

Example:

```

SELECT * FROM pg_show_custom_settings();

```

name	setting	unit	category	short_desc
extra_desc	context	vartype	source	min_val max_val
enumvals	boot_val	reset_val	sourcefile	sourceline
-----+-----+-----				
+-----+-----+-----+-----+-----+-----+				
-----+-----+-----+-----				
hoodie.public.hudi_mor_ft.consume.ending.timestamp	20230404172329544			Customized Options
GUC placeholder variable		user	string	session
hoodie.public.hudi_mor_ft.consume.mode			incremental	
placeholder variable		user	string	session
hoodie.public.hudi_mor_ft.consume.start.timestamp	20230404172329543			Customized Options
GUC placeholder variable		user	string	session

(3 rows)

hudi_get_options(regclass)

Description: Queries the properties of a Hudi foreign table (hoodie.properties). It is represented by a key-value pair.

Return type: SETOF record

Example:

```

select * from hoodie_get_options('public.hudi_mor_ft');

```

key	value
hoodie.table.precombine.field	col_int
hoodie.datasource.write.drop.partition.columns	false
hoodie.table.partition.fields	
hoodie.table.type	MERGE_ON_READ
hoodie.archivelog.folder	archived
hoodie.compaction.payload.class	

```
org.apache.hudi.common.model.OverwriteWithLatestAvroPayload
hoodie.timeline.layout.version      | 1
hoodie.table.version                | 4
hoodie.table.recordkey.fields       | col_bigint
hoodie.database.name                | default
hoodie.datasource.write.partitionpath.urlencode | false
hoodie.table.name                   | lt_test_mor_014
hoodie.table.keygenerator.class      | org.apache.hudi.keygen.ComplexKeyGenerator
hoodie.datasource.write.hive_style_partitioning | true
hoodie.table.create.schema          |
{"type": "record", "name": "lt_test_mor_014_record", "namespace": "hoodie.lt_test_mor_014", "fields": [
  {"name": "_hoodie_commit_time", "type": ["string", "null"]},
  {"name": "_hoodie_commit_seqno", "type": ["string", "null"]},
  {"name": "_hoodie_record_key", "type": ["string", "null"]}, {"name": "_hoodie_partition_path", "type": ["string", "null"]},
  {"name": "_hoodie_file_name", "type": ["string", "null"]}, {"name": "col_bigint", "type": ["long", "null"]},
  {"name": "col_int", "type": ["int", "null"]}, {"name": "col_text", "type": ["string", "null"]}, {"name": "col_text2", "type": ["string", "null"]}]}
hoodie.table.checksum               | 515660817
(16 rows)
```

hoodi_get_max_commit(regclass)

Description: This function retrieves the most recent commit timestamp and data write time for the current Hudi external table.

Return type: record

Example:

```
SELECT * FROM hudi_get_max_commit('public.hudi_mor_ft');
 max_commit | write_time
-----+-----
20221207141822 | 2022-12-07 14:18:30+08
(1 row)
```

hoodi_get_commit(regclass, cstring, int)

Description: This function retrieves the timestamp and write time of the commit data for the current Hudi foreign table, starting from the specified commit and up to the *N*th commit. If the *N*th commit does not exist, the latest commit and the corresponding data write time are returned.

Return type: record

Example:

```
SELECT * FROM hudi_get_commit('public.hudi_mor_ft', '20230329174744657', 3);
 end_commit | write_time
-----+-----
20230329174808908 | 2023-08-31 15:43:08+08
(1 row)
```

hoodi_sync_task_submit(regclass, regclass)

Description: Submits an automatic Hudi synchronization task. The first input parameter is the synchronization target table, and the second input parameter is the Hudi foreign table. If the task is submitted successfully, the task ID is returned.

Return type: text

 NOTE

- The synchronization target table must contain a primary key, which must be the same as the value of **hudi recordkey**.
- If the hudi table contains the **precombine** field, the synchronization target table must also contain the corresponding field.
- If the synchronization target table contains only the primary key (no other fields except the primary key), the synchronization task cannot be submitted normally.
- The user must have the **INSERT** and **UPDATE** permissions on the target table and the **SELECT** permission on the Hudi foreign table. Otherwise, the synchronization task cannot be submitted.

Example:

```
SELECT hudi_sync_task_submit('public.hudi_sync_i','public.hudi_mor_ft');
       hudi_sync_task_submit
-----
6465efe2-3ea1-0b00-dde5-b57dfb30fffe
(1 row)
```

hudi_sync_task_submit(regclass, regclass, interval)

Description: The function is the same as that of **hudi_sync_task_submit(regclass, regclass)**. The difference is that, in this function, you can specify an input parameter of the interval type to specify the task scheduling period. The value ranges from 5 seconds to 24 hours. If the task is submitted successfully, the task ID is returned.

Return type: text

 NOTE

- The synchronization target table must contain a primary key, which must be the same as the value of **hudi recordkey**.
- If the hudi table contains the **precombine** field, the synchronization target table must also contain the corresponding field.
- If the synchronization target table contains only the primary key (no other fields except the primary key), the synchronization task cannot be submitted normally.
- The user must have the **INSERT** and **UPDATE** permissions on the target table and the **SELECT** permission on the Hudi foreign table. Otherwise, the synchronization task cannot be submitted.

Example:

```
SELECT hudi_sync_task_submit('public.hudi_sync_i','public.hudi_mor_ft','1 hour');
       hudi_sync_task_submit
-----
6465efe2-3ea1-0b00-dde5-b57dfb30fffe
(1 row)
```

hudi_sync_task_submit(regclass, regclass, text, text)

Description: The function is the same as that of **hudi_sync_task_submit(regclass, regclass)**. The difference is that you can specify two additional text input parameters to indicate the fields that you expect to be synchronized. The fields are separated by commas (,). Quotation marks and escape characters can be parsed. The number and sequence of fields in the two text parameters must be the same, because the synchronization will be performed based on the mapping between

the fields in the two text parameters. If the task is submitted successfully, the task ID is returned.

Return type: text

NOTE

- The synchronization target table must contain a primary key, which must be the same as the value of **hudi recordkey**.
- If the hudi table contains the **precombine** field, the synchronization target table must also contain the corresponding field.
- If the synchronization target table contains only the primary key (no other fields except the primary key), the synchronization task cannot be submitted normally.
- The user must have the **INSERT** and **UPDATE** permissions on the target table and the **SELECT** permission on the Hudi foreign table. Otherwise, the synchronization task cannot be submitted.

Example:

```
SELECT hudi_sync_task_submit('public.hudi_sync_i','public.hudi_mor_ft','_hoodie_commit_time, col_bigint, col_text', '_hoodie_commit_time, col_bigint, col_text');
       hudi_sync_task_submit
-----
646610bc-cdd1-0d00-d07d-b57e89a0fffe
(1 row)
```

hudi_sync_task_submit(regclass, regclass, text, text, interval)

Description: The function is the same as that of **hudi_sync_task_submit(regclass, regclass, text, text)**. The difference is that you can specify an additional input parameter of the interval type to specify the task scheduling period. The value ranges from 5 seconds to 24 hours.

Return type: text

NOTE

- The synchronization target table must contain a primary key, which must be the same as the value of **hudi recordkey**.
- If the hudi table contains the **precombine** field, the synchronization target table must also contain the corresponding field.
- If the synchronization target table contains only the primary key (no other fields except the primary key), the synchronization task cannot be submitted normally.
- The user must have the **INSERT** and **UPDATE** permissions on the target table and the **SELECT** permission on the Hudi foreign table. Otherwise, the synchronization task cannot be submitted.

Example:

```
SELECT hudi_sync_task_submit('public.hudi_sync_i','public.hudi_mor_ft','_hoodie_commit_time, col_bigint, col_text', '_hoodie_commit_time, col_bigint, col_text', '10 minute 30second');
       hudi_sync_task_submit
-----
646610bc-cdd1-0d00-d07d-b57e89a0fffe
(1 row)
```

hudi_show_sync_state()

Description: Obtains the synchronization status of the Hudi automatic synchronization task.

Return type: SETOF record

Example:

```
SELECT * FROM hudi_show_sync_state();
   target_tbl | source_ftbl |          payload_type          | precombine_key |
latest_commit |
-----+-----+-----+-----+
public.hudi_sync_i | public.hudi_mor_ft | org.apache.hudi.common.model.OverwriteWithLatestAvroPayload | |
col_int      | 20230511114021573 |
(1 row)
```

hudi_sync(regclass, regclass)

Description: Stored procedure, which is invoked by the Hudi automatic synchronization task. Tasks submitted using **pg_catalog.hudi_sync_task_submit(regclass, regclass)** will execute the stored procedure. Upon successful execution of the command, the number of synchronized rows and timestamp will be displayed.

Return type: text

Example:

```
CALL hudi_sync('public.hudi_sync_i', 'public.hudi_mor_ft');
NOTICE: execute full sync
CONTEXT: PL/pgSQL function hudi_sync(regclass,regclass) line 11 at RETURN
        hudi_sync
-----
sync 1 rows up to 20230511114021573.
(1 row)
```

hudi_sync_custom(regclass, regclass, text)

Description: A stored procedure, which is the entry for invoking the Hudi automatic synchronization task. Users can customize the mapping between the fields in the target table and those in the data source table. Tasks submitted using **pg_catalog.hudi_sync_task_submit(regclass, regclass, text, text)** will execute the stored procedure. In the preceding information, **text** is a JSON string, indicating the synchronization mapping between two table fields. Upon successful execution of the command, the number of synchronized rows and timestamp will be displayed.

Return type: text

Example:

```
CALL hudi_sync_custom('public.hudi_sync_i', 'public.hudi_mor_ft', '{"_hoodie_commit_time" :
"_hoodie_commit_time", "col_bigint" : "col_bigint", "col_text" : "col_text"}');
NOTICE: execute full sync
CONTEXT: PL/pgSQL function hudi_sync_custom(regclass,regclass,text) line 14 at RETURN
        hudi_sync_custom
-----
sync 1 rows up to 20230511114021573.
(1 row)
```

hudi_set_sync_commit(regclass, regclass, text)

Description: Sets the start timestamp of the first synchronization of the Hudi automatic synchronization task to prevent resynchronization. The first parameter is the synchronization target table, the second parameter is the Hudi foreign table,

and the third parameter is the expected synchronization start point. This function must be used before a synchronization task is submitted.

Return type: text

Example:

```
select hudi_set_sync_commit('public.hudi_sync_i', 'public.hudi_mor_ft', '20230511114021573');
NOTICE: set sync commit successfully, the next synchronization will start from 20230511114021573
CONTEXT: referenced column: hudi_set_sync_commit
-----
20230511114021573
(1 row)
```

NOTE

You must have the **INSERT** and **UPDATE** permissions on the target table and the **SELECT** permission on the Hudi foreign table. Otherwise, you cannot set the synchronization progress.

hudi_set_sync_commit(text, text)

Description: Sets the start timestamp of the next synchronization of a Hudi automatic synchronization task. You can use it to sync historical data again or to skip some data. The first parameter is the task ID, and the second parameter is the expected start time of the next synchronization. This function can be used only after a synchronization task is submitted. Before using this function, you need to pause the task.

Return type: text

Example:

```
select hudi_set_sync_commit('6524c8e3-aae9-0000-5a14-be8ec000fffe', '20230511114021573');
NOTICE: set sync commit successfully, the next synchronization will start from 20230511114021573
CONTEXT: referenced column: hudi_set_sync_commit
-----
20230511114021573
(1 row)
```

NOTE

- Only users who have the permission on the target task can invoke this function.
- Before invoking this function, ensure that the target task is paused and has been successfully executed at least once.

pg_task_show(text)

Description: Queries the information about the current automatic scheduling task. For a Hudi synchronization task, the input parameter should be **SQLonHudi**.

Return type: SETOF record

Example:

```
SELECT * FROM pg_task_show('SQLonHudi');
      task_id      |
-----+-----
what              |
interval | time_cons | start_time | end_time | parameter | task_rank | next_start_time |
| next_end_time | last_log | failure_times
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
64d257e9-1e9b-0d00-3ce3-7e61b5e0ffe | call pg_catalog.hudi_sync_custom('public.hudi_read_target',
'public.hudi_read101', '{"_hoodie_commit_seqno" : "_hoodie_commit_seqno", "id" : "id", "ts" : "ts",
"long_field" : "ts"}'); | SQLonHudi | 10 | f | '00:00:10' | | 2023-08-08 22:58:15.846903+08
| | | 5 | 2023-08-08 22:58:15.846903+08 | 2023-08-08 22:58:24.846903+08 |
| | 0
(1 row)
```

NOTE

The **last_log** and **failure_times** fields are used to record the status of the last task.

- The value of **last_log** is updated when the task is complete. If the task is successful, the content is cleared. If the task fails, the task failure log is recorded.
- The value of **failure_times** is updated at the end of each time window. It is set to **0** if the task is successful, incremented by 1 if the task fails, and unchanged if the task is not started in that round. This value helps identify when the first failure happens.

pg_task_remove(text)

Description: Deletes an automatic scheduling task. The input parameter is a task ID. The function returns the number of deleted tasks.

Return type: integer

Example:

```
SELECT pg_task_remove('64661705-8ada-0100-d07f-b57e89a0ffe');
pg_task_remove
-----
1
(1 row)
```

pg_task_pause(text)

Description: Suspend an automatic scheduling task. The input parameter is a task ID. The function returns the number of suspended tasks.

Return type: integer

Example:

```
SELECT pg_task_pause('64661705-8ada-0100-d07f-b57e89a0ffe');
pg_task_pause
-----
1
(1 row)
```

pg_task_resume(text)

Description: Resumes an automatic scheduling task. The input parameter is the ID of a suspended task. The function returns the number of resumed tasks.

Return type: integer

Example:

```
SELECT pg_task_resume('64661705-8ada-0100-d07f-b57e89a0ffe');
pg_task_resume
-----
```

```
1  
(1 row)
```

pg_task_reset_interval(text, interval)

Description: Modifies the scheduling period of a synchronization task. The first input parameter is **task_id**, and the second input parameter the scheduling period, of which the value ranges from 5 seconds to 24 hours. The function returns the number of tasks whose periods are modified.

Return type: integer

Example:

```
select pg_task_reset_interval('64bfd69c-a016-0000-120e-1e802978fffe', '10 hours 30 minutes');  
pg_task_reset_interval  
-----  
1  
(1 row)
```

7.36 Funnel and Retention Functions

Funnel and retention functions are supported only in cluster 8.3.0 and later versions.

Context

Both funnel functions and retention functions are widely used tools for analyzing user behavior in product and data analysis. They are particularly useful for product managers, data scientists, and marketing personnel. These functions assist in analyzing the customer journey, comprehending user churn and conversion problems, and assessing the product's ability to consistently attract and retain customers.

- Funnel analysis: examines each step a customer takes from initial contact to final purchase. It helps product teams and marketers understand how users interact with products or services, as well as the conversion and churn rates at each step. The funnel analysis involves designating stages of the customer's journey from first hearing about a product to making a purchase. The whole process may include visiting a website, registering an account, and purchasing a product.
- Retention analysis: focuses on the percentage of users who continue using a product after a specified period of time. It calculates the number of users who remain active after a certain time point, such as user registration, first order placement, or initial use. Common retention analysis functions include daily, weekly, and monthly retention rates, which help analyze whether users continue using the product after initially hearing about it.

WINDOW_FUNNEL

The **WINDOW_FUNNEL** function searches for an event chain in the sliding time window and counts the maximum number of consecutive events in the event chain.

Given a list of user-defined events, DWS finds the longest sequential match starting from the first event and returns the length of the match. Once the matching fails, the entire matching ends. Example:

Assume that the window is large enough:

- The condition events are c1, c2, and c3, but the user data is c1, c2, c3, and c4. So, c1, c2, and c3 are matched, and the return value of the function is 3.
- The condition events are c1, c2, and c3, but the user data is c4, c3, c2, and c1. So, c1 is matched, and the return value of the function is 1.
- The condition events are c1, c2, and c3, but the user data is c4 and c3. So, no event is matched, and the return value of the function is 0.

Function syntax

```
windowFunnel(window, mode, timestamp, cond1, cond2, ..., condN)
```

Input parameters

- **window**: bigint type. It indicates the size of the sliding time window. The unit is second.
- **mode**: text type. Currently, only the **Default** mode is supported. For other modes, an error is reported. In the **Default** mode, it matches as many events as possible, starting from the first event in the window.
- **timestamp**: time range when an event occurs. The **timestamp without time zone**, **timestamp with time zone**, **date**, **int** and **bigint** types are supported.
- **cond**: variable-length Boolean array. It indicates the condition. DWS supports only 1 to 32 conditions. If the number of conditions is not within the range, an error is reported.

Return values

level: int type. It indicates the maximum length of the event list that matches the condition.

Retention Functions

The retention function evaluates if an event meets each condition, starting from the first one.

Function syntax

```
retention(cond1, cond2, ..., cond32);
```

Input parameters

cond: variable-length Boolean array with a maximum length of 32 characters, indicating whether an event meets specific conditions. DWS supports only 1 to 32 conditions. If the number of conditions is not within the range, an error is reported.

Return values

retention condition: tinyint array type. It indicates the expression of the returned result, which is the same as the length of the input parameter **cond**. If the **cond1** and **condi** conditions are met, the *i*th value of the returned result is **1**. Otherwise, the *i*th value is **0**.

Other Retention-Related Functions

DWS supports functions **range_retention_count** and **range_retention_sum** as supplements to the retention function for better customer retention analysis.

- **range_retention_count**

range_retention_count calculates the retention rate of each user. This function returns an array, which can be used as the input parameter of the **range_retention_sum** function.

Function syntax

```
range_retention_count(is_first, is_active, dt, retention_interval, retention_granularity, output_format)
```

Input parameters

- **is_first**: indicates whether it is an initial behavior. The value is of the Boolean type. The value **true** indicates that it is an initial behavior, and the value **false** indicates that it is not an initial behavior.
- **is_active**: indicates whether it is a retention behavior. The value is of the Boolean type. The value **true** indicates that it is a retention behavior, and the value **false** indicates that it is not a retention behavior.
- **dt**: indicates the date when the behavior happens. The value is of the date type.
- **retention_interval**: indicates the storage interval. The value is of the array type. A maximum of 15 storage intervals are supported. For example, ARRAY[1,3,5,7,15,30].
- **retention_granularity**: indicates the retention granularity, which can be day, week, or month. The value is of the text type.
- **output_format**: indicates the output format. The value is of the text type and can be **normal** (default) or **expand** (daily retention details can be obtained).

Return value: BIGINT array of user retention information.

- **range_retention_sum**

range_retention_sum summarizes and calculates the daily (weekly/monthly) retention rate of all users.

Function syntax

```
range_retention_sum(range_retention_count_result)
```

Input parameter: return value of the **range_retention_count** function.

Return value: text array of user retention statistics.

Examples

Create the **funnel_test** table.

```
CREATE TABLE IF NOT EXISTS funnel_test
(
  user_id INT ,
  event_type TEXT,
  event_time TIMESTAMP,
  event_timez TIMESTAMP WITH TIME ZONE,
  event_time_int BIGINT
);
```

Insert data.

```
INSERT INTO funnel_test VALUES
(1,'Browse','2021-01-31 11:00:00', '2021-01-31 11:00:00+08', 10),
(1,'Click','2021-01-31 11:10:00', '2021-01-31 11:10:00+07', 20),
(1,'Order','2021-01-31 11:20:00', '2021-01-31 11:20:00+06', 30),
(1,'Pay','2021-01-31 11:30:00', '2021-01-31 11:30:00+05', 40),
(2,'Order','2021-01-31 11:00:00', '2021-01-31 11:00:00+08', 11),
(2,'Pay','2021-01-31 11:10:00', '2021-01-31 11:10:00+08', 12),
(1,'Browse','2021-01-31 11:00:00', '2021-01-31 11:00:00+01', 50),
(3,'Browse','2021-01-31 11:20:00', '2021-01-31 11:20:00-04', 30),
(3,'Click','2021-01-31 12:00:00', '2021-01-31 12:00:00-04', 80),
(4,'Browse','2021-01-31 11:50:00', '2021-01-31 11:50:00-01', 1000),
(4,'Pay','2021-01-31 12:00:00', '2021-01-31 12:00:00-02', 900),
(4,'Order','2021-01-31 12:00:00', '2021-01-31 12:00:00-03', 1001),
(4,'Click','2021-01-31 12:00:00', '2021-01-31 12:00:00-04', 1001),
(5,'Browse','2021-01-31 11:50:00', '2021-01-31 11:50:00+08', NULL),
(5,'Click','2021-01-31 12:00:00', '2021-01-31 12:00:00+08', 776),
(5,'Order','2021-01-31 11:10:00', '2021-01-31 11:10:00+08', 999),
(6,'Browse','2021-01-31 11:50:00', '2021-01-31 11:50:00+01', -1),
(6,'Click','2021-01-31 12:00:00', '2021-01-31 12:00:00+02', -2),
(6,'Order','2021-01-31 12:10:00', '2021-01-31 12:00:00+03', -3);
```

Calculate the funnel for each user. In the **level** column of the following command output, **0** means that there were no events in the window period and **1** means that there was one event in the window period.

```
SELECT
  user_id,
  windowFunnel(
    0, 'default', event_timez,
    event_type = 'Browse', event_type = 'Click', event_type = 'Order', event_type = 'Pay'
  ) AS level
FROM funnel_test
GROUP BY user_id
ORDER BY user_id;
```

user_id	level
1	1
2	0
3	1
4	1
5	1
6	1

(6 rows)

Calculate the funnel of each user and specify the length of the sliding time window as **NULL**. An error is reported.

```
SELECT
  user_id,
  windowFunnel(
    NULL, 'default', event_time,
    event_type = 'Browse', event_type = 'Click', event_type = 'Order', event_type = 'Pay'
  ) AS level
FROM funnel_test
GROUP BY user_id
ORDER BY user_id;
```

ERROR: Invalid parameter : window length or mode is null.

Calculate the funnel of each user and specify multiple conditions.

```
SELECT
  user_id,
  windowFunnel(
    40, 'default', date(event_time),
    true, true, false, true
  ) AS level
FROM funnel_test
GROUP BY user_id
```

```
ORDER by user_id;  
user_id | level
```

user_id	level
1	2
2	2
3	2
4	2
5	2
6	2

(6 rows)

Check the retention rate of each user in each event.

```
SELECT  
  user_id,  
  retention(  
    event_type = 'Browse', event_type = 'Click', event_type = 'Order', event_type = 'Pay'  
  ) AS r  
FROM funnel_test  
GROUP BY user_id  
ORDER BY user_id ASC;  
user_id | r
```

user_id	r
1	{1,1,1,1}
2	{0,0,0,0}
3	{1,1,0,0}
4	{1,1,1,1}
5	{1,1,1,0}
6	{1,1,1,0}

(6 rows)

Analyze the retention rate of each user in each event and set the first time point to **false**.

```
SELECT  
  user_id,  
  retention(  
    false, event_type = 'Browse', event_type = 'Click', event_type = 'Order', event_type = 'Pay'  
  ) AS r  
FROM funnel_test  
GROUP BY user_id  
ORDER BY user_id ASC;  
user_id | r
```

user_id	r
1	{0,0,0,0,0}
2	{0,0,0,0,0}
3	{0,0,0,0,0}
4	{0,0,0,0,0}
5	{0,0,0,0,0}
6	{0,0,0,0,0}

(6 rows)

Analyze the retention rate of all users in each event.

```
SELECT sum(r[1]), sum(r[2]), sum(r[3]), sum(r[4])  
FROM  
(  
  SELECT  
    retention(event_type = 'Browse', event_type = 'Click', event_type = 'Order', event_type = 'Pay') AS r  
  FROM funnel_test  
  GROUP BY user_id  
)  
sum | sum | sum | sum
```

sum	sum	sum	sum
5	5	4	2

(1 row)

Create the **retention_test** table.

```
CREATE TABLE retention_test(  
  uid INT,
```



```
event TEXT,  
event_time TIMESTAMP  
);
```

Insert data.

```
INSERT INTO retention_test VALUES  
(1, 'pay', '2024-05-01'),  
(1, 'login', '2024-05-01'),  
(1, 'pay', '2024-05-02'),  
(1, 'login', '2024-05-02'),  
(2, 'login', '2024-05-01'),  
(3, 'login', '2024-05-02'),  
(3, 'pay', '2024-05-03'),  
(3, 'pay', '2024-05-04');
```

Gather statistics on the retention rate of each user in the payment event after one or two days.

```
WITH retention_count_info AS (  
  SELECT  
    uid,  
    range_retention_count(event = 'login', event = 'pay',  
                          DATE(event_time), array[1, 2], 'day') AS info  
  FROM  
    retention_test  
  GROUP BY  
    uid  
) , retention_sum AS (  
  SELECT regexp_split_to_array(unnest(range_retention_sum(info)), ',') AS s  
  FROM retention_count_info  
) SELECT to_date(s[1]::int) AS login_date,  
        s[3]::numeric / s[2]::numeric AS retention_rate1,  
        s[4]::numeric / s[2]::numeric AS retention_rate2  
FROM retention_sum  
ORDER BY login_date;  
  login_date   | retention_rate1 | retention_rate2  
-----+-----  
2024-05-01 00:00:00 | .500000000000000000 | 0.000000000000000000  
2024-05-02 00:00:00 | .500000000000000000 | .500000000000000000  
(2 rows)
```

7.37 EXTERNAL SCHEMA System Functions

System functions related to EXTERNAL SCHEMA are supported only by clusters of version 8.3.0 or later.

pg_get_external_schema_table_options(text, text)

Description: Obtains the options of an external schema table.

Input parameter: The first input parameter is the external schema name, and the second input parameter is the table name.

Return type: SETOF record

Example:

```
SELECT * FROM pg_get_external_schema_table_options('ex_lf', 'test_lf');  
option_name | option_value  
-----+-----  
encoding    | utf8  
format      | parquet  
foldername  | /***/***/***  
(3 rows)
```

pg_get_external_schema_table_col(text, text)

Description: Obtains the column information of an external schema table.

Input parameter: The first input parameter is the external schema name, and the second input parameter is the table name.

Return type: SETOF record

Example:

```
SELECT * FROM pg_get_external_schema_table_col('ex_lf', 'test_lf');
 col_name | col_type | part_col
-----+-----+-----
field_smallint | smallint | f
field_int | int | f
field_integer | int | f
field_bigint | bigint | f
field_float | float | f
field_double | double | f
field_double_precision | double | f
field_decimal | decimal(10,0) | f
field_numeric | decimal(10,0) | f
field_timestamp | timestamp | f
field_date | date | f
field_varchar | varchar(5) | f
field_char | char(5) | f
field_boolean | boolean | f
field_string | string | f
(15 rows)
```

7.38 Storage-Compute Decoupling Functions

7.38.1 Data Sharing Functions

pgxc_group_add_subscription(src_vw_name, target_vw_name)

Description: In a storage-compute decoupling architecture, a KV subscription relationship is established between logical clusters (Virtual Warehouses, VWs). Once established, the KV cache of the consumer VW periodically synchronizes the SST file incrementally from the OBS cudesc snapshot directory of the producer to the local host. This allows tables in the producer VW to be queried across VWs in the consumer VM. The created subscription relationship can be found in the **pgxc_group_subscription** table. This function is supported only by clusters of version 9.0.3 or later.

Return type: void

The following information is displayed:

Table 7-55 Columns returned by pgxc_group_add_subscription(src_vw_name, target_vw_name)

Column	Type	Description
src_vw_name	text	Producer VW name, which is usually used as the VW to which data is written.

Column	Type	Description
target_vw_name	text	Consumer VW name, which is usually used as the VW for reading data.

Example:

```
SELECT pgxc_group_add_subscription('write_group', 'single_read_group');  
pgxc_group_add_subscription
```

(1 row)

pgxc_group_drop_subscription(src_vw_name, target_vw_name)

Description: To delete an established KV subscription relationship in a storage and compute separation architecture, remove the records from the **pgxc_group_subscription** table. After deletion, the consumer VW can no longer query tables in the producer VW across VWs. This function is supported only by clusters of version 9.0.3 or later.

Return type: void

The following information is displayed:

Table 7-56 Columns returned by pgxc_group_drop_subscription(src_vw_name, target_vw_name)

Column	Type	Description
src_vw_name	text	Producer VW name, which is usually used as the VW to which data is written.
target_vw_name	text	Consumer VW name, which is usually used as the VW for reading data.

Example:

```
SELECT pgxc_group_drop_subscription('write_group', 'single_read_group');  
pgxc_group_drop_subscription
```

(1 row)

7.38.2 Function for Obtaining File Information of Tablespaces in the Trash Directory

pgxc_tablespace_trash_info()

Description: Obtains the file statistics of the tablespaces whose filesystem is OBS in the trash directory in a storage-compute decoupling architecture. The information includes the number of files, total file size, and whether the

automatic cleanup lifecycle policy is enabled for the trash directory. This function is available only for clusters of version 9.1.0.222 or later.

Return type: record

Return columns:

Column	Type	Description
spcname	text	Tablespace name.
file_count	int	Number of tablespace-related files in the trash directory.
total_size	int	Total size of tablespace files in the trash directory.
lifecycle	text	Whether the expiration policy for lifecycle is added to the trash directory. This field does not take effect in the current version. • active: The policy is added. • inactive: The policy is not added.

Example:

```
select * from pgxc_tablespace_trash_info();
spcname | file_count | total_size | lifecycle
-----+-----+-----+-----
cu_obs_tbs|      0    |      0    | inactive
(1 row)
```

8 Expressions

8.1 Simple Expressions

Logical Expressions

[Logical Operators](#) lists the operators and calculation rules of logical expressions.

Comparative Expressions

[Comparison Operators](#) lists the common comparative operators.

In addition to comparative operators, you can also use the following sentence structure:

- BETWEEN operator
a BETWEEN x AND y is equivalent to **a >= x AND a <= y**.
a NOT BETWEEN x AND y is equivalent to **a < x OR a > y**.
- To check whether a value is null, use:
expression IS NULL
expression IS NOT NULL
or an equivalent (non-standard) sentence structure:
expression ISNULL
expression NOTNULL

NOTICE

Do not write **expression=NULL** or **expression<>(!=)NULL**, because **NULL** represents an unknown value, and these expressions cannot determine whether two unknown values are equal.

Examples

```
SELECT 2 BETWEEN 1 AND 3 AS RESULT;  
result
```

```
-----
t
(1 row)

SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
result
-----
t
(1 row)

SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
result
-----
f
(1 row)

SELECT 2 < 1 OR 2 > 3 AS RESULT;
result
-----
f
(1 row)

SELECT 2+2 IS NULL AS RESULT;
result
-----
f
(1 row)

SELECT 2+2 IS NOT NULL AS RESULT;
result
-----
t
(1 row)

SELECT 2+2 ISNULL AS RESULT;
result
-----
f
(1 row)

SELECT 2+2 NOTNULL AS RESULT;
result
-----
t
(1 row)

SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;
result
-----
t
(1 row)

SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;
result
-----
f
(1 row)
```

8.2 Conditional Expressions

Data that meets the requirements specified by conditional expressions are filtered during SQL statement execution.

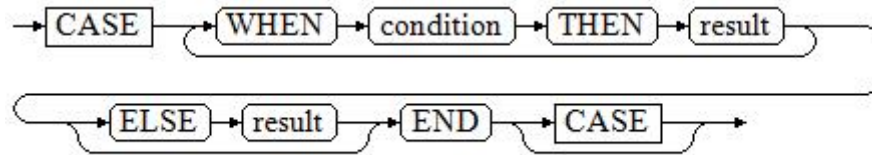
Conditional expressions include the following types:

- CASE

CASE expressions are similar to the **CASE** statements in other coding languages.

Figure 8-1 shows the syntax of a **CASE** expression.

Figure 8-1 case::=



A **CASE** clause can be used in a valid expression. **condition** is an expression that returns a value of Boolean type.

- If the result is **true**, the result of the **CASE** expression is the required result.
- If the result is false, the following **WHEN** or **ELSE** clauses are processed in the same way.
- If every **WHEN condition** is false, the result of the expression is the result of the **ELSE** clause. If the **ELSE** clause is omitted and has no match condition, the result is NULL.

CAUTION

- Keep the number of nested **case when** layers at 3 or fewer. The parser must expand all related fields during parsing. Deep recursion can lead to high memory usage.
- For complex **case when** statements, use a subquery instead. For details, see [Subquery Expressions](#).

Example:

```
CREATE TABLE tpcds.case_when_t1(CW_COL1 INT) DISTRIBUTE BY HASH (CW_COL1);

INSERT INTO tpcds.case_when_t1 VALUES (1), (2), (3);

SELECT * FROM tpcds.case_when_t1;
cw_col1
-----
3
1
2
(3 rows)

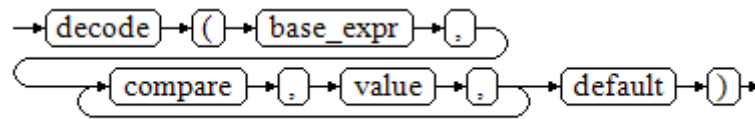
SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN 'two' ELSE 'other'
END FROM tpcds.case_when_t1;
cw_col1 | case
-----+-----
3 | other
1 | one
2 | two
(3 rows)

DROP TABLE tpcds.case_when_t1;
```

- DECODE

Figure 8-2 shows the syntax of a **DECODE** expression.

Figure 8-2 decode::=



Compare each following **compare(n)** with **base_expr**, **value(n)** is returned if a **compare(n)** matches the **base_expr** expression. If **base_expr** does not match each **compare(n)**, the default value is returned.

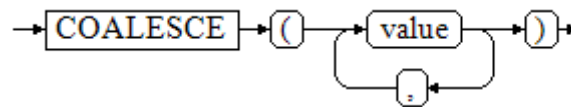
Conditional Expression Functions describes the examples.

```
SELECT DECODE('A','A',1,'B',2,0);
case
-----
1
(1 row)
```

- **COALESCE**

Figure 8-3 shows the syntax of a **COALESCE** expression.

Figure 8-3 coalesce::=



COALESCE returns its first non-NULL value. If all the arguments are NULL, return **NULL**. This value is replaced by the default value when data is displayed. Like a **CASE** expression, **COALESCE** only evaluates the parameters that are needed to determine the result. That is, parameters to the right of the first non-null parameter are not evaluated.

Example:

```
CREATE TABLE tpcds.c_tabl(description varchar(10), short_description varchar(10), last_value
varchar(10))
DISTRIBUTE BY HASH (last_value);

INSERT INTO tpcds.c_tabl VALUES('abc', 'efg', '123');
INSERT INTO tpcds.c_tabl VALUES(NULL, 'efg', '123');
INSERT INTO tpcds.c_tabl VALUES(NULL, NULL, '123');

SELECT description, short_description, last_value, COALESCE(description, short_description, last_value)
FROM tpcds.c_tabl ORDER BY 1, 2, 3, 4;
description | short_description | last_value | coalesce
-----+-----+-----+-----
abc         | efg              | 123       | abc
           | efg              | 123       | efg
           |                  | 123       | 123
(3 rows)

DROP TABLE tpcds.c_tabl;
```

If **description** is not **NULL**, the value of **description** is returned. Otherwise, parameter **short_description** is calculated. If **short_description** is not **NULL**,

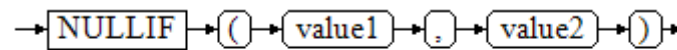
the value of **short_description** is returned. Otherwise, parameter **last_value** is calculated. If **last_value** is not **NULL**, the value of **last_value** is returned. Otherwise, **none** is returned.

```
SELECT COALESCE(NULL,'Hello World');
-----
coalesce
-----
Hello World
(1 row)
```

- **NULLIF**

Figure 8-4 shows the syntax of a **NULLIF** expression.

Figure 8-4 nullif::=



Only if **value1** is equal to **value2** can **NULLIF** return the **NULL** value. Otherwise, **value1** is returned.

Example:

```
CREATE TABLE tpcds.null_if_t1 (
  NI_VALUE1 VARCHAR(10),
  NI_VALUE2 VARCHAR(10)
) DISTRIBUTED BY HASH (NI_VALUE1);

INSERT INTO tpcds.null_if_t1 VALUES('abc', 'abc');
INSERT INTO tpcds.null_if_t1 VALUES('abc', 'efg');

SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM tpcds.null_if_t1 ORDER BY 1,
2, 3;

ni_value1 | ni_value2 | nullif
-----+-----+-----
abc      | abc      |
abc      | efg      | abc
(2 rows)
DROP TABLE tpcds.null_if_t1;
```

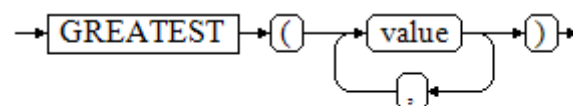
If **value1** is equal to **value2**, **NULL** is returned. Otherwise, **value1** is returned.

```
SELECT NULLIF('Hello','Hello World');
-----
nullif
-----
Hello
(1 row)
```

- **GREATEST** (maximum value) and **LEAST** (minimum value)

Figure 8-5 shows the syntax of a **GREATEST** expression.

Figure 8-5 greatest::=



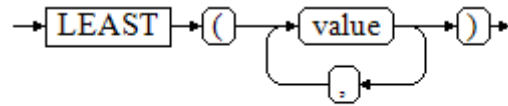
You can select the maximum value from any numerical expression list.

```
SELECT greatest(9000,155555,2.01);
-----
greatest
```

```
-----  
155555  
(1 row)
```

Figure 8-6 shows the syntax of a **LEAST** expression.

Figure 8-6 least::=



You can select the minimum value from any numerical expression list.

Each of the preceding numeric expressions can be converted into a common data type, which will be the data type of the result.

The NULL values in the list will be ignored. The result is **NULL** only if the results of all expressions are **NULL**.

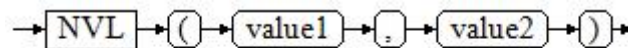
```
SELECT least(9000,2);  
least  
-----  
2  
(1 row)
```

Conditional Expression Functions describes the examples.

- NVL

Figure 8-7 shows the syntax of an **NVL** expression.

Figure 8-7 nvl::=



If the value of **value1** is **NULL**, **value2** is returned. Otherwise, **value1** is returned.

Example:

```
SELECT nvl(null,1);  
nvl  
-----  
1  
(1 row)  
SELECT nvl ('Hello World' ,1);  
nvl  
-----  
Hello World  
(1 row)
```

- IF

Figure 8-8 shows the syntax of an **IF** expression.

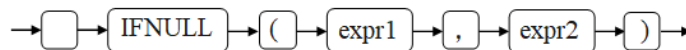
Figure 8-8 if::=

If the value of **bool_expr** is **true**, **expr1** is returned. Otherwise, **expr2** is returned.

[Conditional Expression Functions](#) describes the examples.

- IFNULL

[Figure 8-9](#) shows the syntax of an **IFNULL** expression.

Figure 8-9 ifnull::=

It returns **expr1** or **expr2**. If **expr1** is not **NULL**, **expr1** is returned. Otherwise, **expr2** is returned.

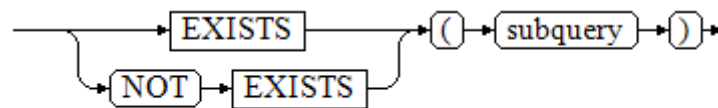
[Conditional Expression Functions](#) describes the examples.

8.3 Subquery Expressions

Subquery expressions include the following types:

- EXISTS/NOT EXISTS

[Figure 8-10](#) shows the syntax of an **EXISTS/NOT EXISTS** expression.

Figure 8-10 EXISTS/NOT EXISTS::=

The parameter of an **EXISTS** expression is an arbitrary **SELECT** statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of **EXISTS** is "true". If the subquery returns no rows, the result of **EXISTS** is "false".

The subquery will generally only be executed long enough to determine whether at least one row is returned, not all the way to completion.

Example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE EXISTS (SELECT d_dom FROM tpcds.date_dim WHERE d_dom = store_returns.sr_reason_sk and sr_customer_sk <10);
```

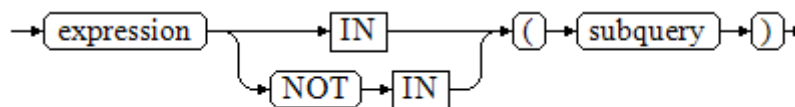
sr_reason_sk	sr_customer_sk
13	2
22	5
17	7
25	7
3	7
31	5
7	7
14	6
20	4
5	6
10	3
1	5
15	2
4	1
26	3

(15 rows)

- IN/NOT IN

Figure 8-11 shows the syntax of an **IN/NOT IN** expression.

Figure 8-11 IN/NOT IN::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **IN** is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

This is in accordance with SQL's normal rules for Boolean combinations of null values. If the columns corresponding to two rows equal and are not empty, the two rows are equal to each other. If any columns corresponding to the two rows do not equal and are not empty, the two rows are not equal to each other. Otherwise, the result is **NULL**. If there are no equal right-hand values and at least one right-hand row yields null, the result of **IN** will be null, not false.

Example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk IN (SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
```

sr_reason_sk	sr_customer_sk
10	3
26	3
22	5
31	5
1	5
32	5
32	5
4	1
15	2
13	2
33	4
20	4
33	8
5	6

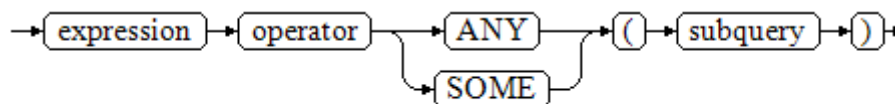
14	6
17	7
3	7
25	7
7	7

(19 rows)

- ANY/SOME

Figure 8-12 shows the syntax of an **ANY/SOME** expression.

Figure 8-12 any/some::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ANY** is "true" if any true result is obtained. The result is "false" if no true result is found (including the case where the subquery returns no rows). **SOME** is a synonym of **ANY**. **IN** can be equivalently replaced with **ANY**.

Example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk < ANY
(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
```

sr_reason_sk | sr_customer_sk

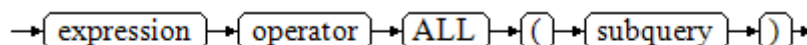
26	3
17	7
32	5
32	5
13	2
31	5
25	7
5	6
7	7
10	3
1	5
14	6
4	1
3	7
22	5
33	4
20	4
33	8
15	2

(19 rows)

- ALL

Figure 8-13 shows the syntax of an **ALL** expression.

Figure 8-13 all::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ALL** is "true" if all rows yield true (including the case where the subquery returns no rows). The result is "false" if any false result is found.

Example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk < all(SELECT
d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
-----+-----
(0 rows)
```

8.4 Array Expressions

IN

expression **IN** (*value* [, ...])

The parentheses on the right contain an expression list. The expression result on the left is compared with the content in the expression list. If the content in the list meets the expression result on the left, the result of **IN** is **true**. If no result meets the requirements, the result of **IN** is **false**.

Example:

```
SELECT 8000+500 IN (10000, 9000) AS RESULT;
result
-----
f
(1 row)
```

NOTE

If the expression result is null or the expression list does not meet the expression conditions and at least one empty value is returned for the expression list on the right, the result of **IN** is **null** rather than **false**. This method is consistent with the Boolean rules used when SQL statements return empty values.

NOT IN

expression **NOT IN** (*value* [, ...])

The parentheses on the right contain an expression list. The expression result on the left is compared with the content in the expression list. If the content in the list does not meet the expression result on the left, the result of **NOT IN** is **true**. If any content meets the expression result, the result of **NOT IN** is **false**.

Example:

```
SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
result
-----
t
(1 row)
```

 NOTE

If the query statement result is null or the expression list does not meet the expression conditions and at least one empty value is returned for the expression list on the right, the result of **NOT IN** is **null** rather than **false**. This method is consistent with the Boolean rules used when SQL statements return empty values.

In all situations, **X NOT IN Y** equals to **NOT(X IN Y)**.

ANY/SOME (array)

expression operator ANY (array expression)

expression operator SOME (array expression)

```
SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;  
result  
-----  
t  
(1 row)  
SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;  
result  
-----  
t  
(1 row)
```

The parentheses on the right contain an array expression, which must generate an array value. The result of the expression on the left uses operators to compute and compare the results in each row of the array expression. The comparison result must be a Boolean value.

- If at least one comparison result is true, the result of **ANY** is **true**.
- If no comparison result is true, the result of **ANY** is false.

 NOTE

If no comparison result is true and the array expression generates at least one null value, the value of **ANY** is NULL, rather than false. This method is consistent with the Boolean rules used when SQL statements return empty values.

SOME is a synonym of **ANY**.

ALL (array)

expression operator ALL (array expression)

The parentheses on the right contain an array expression, which must generate an array value. The result of the expression on the left uses operators to compute and compare the results in each row of the array expression. The comparison result must be a Boolean value.

- The result of **ALL** is "true" if all comparisons yield **true** (including the case where the array has zero elements).
- The result is **false** if any false result is found.

If the array expression yields a null array, the result of **ALL** will be null. If the left-hand expression yields null, the result of **ALL** is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of **ALL** will be null, not true (again, assuming a strict comparison operator).

This method is consistent with the Boolean rules used when SQL statements return empty values.

```
SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;  
result  
-----  
t  
(1 row)
```

8.5 Row Expressions

Syntax:

row_constructor operator row_constructor

Both sides of the row expression are row constructors. The values of both rows must have the same number of fields and they are compared with each other. The row comparison allows operators including =, <>, <, <=, and >= or a similar operator.

The use of operators =<> is slightly different from other operators. If all fields of two rows are not empty and equal, the two rows are equal. If any field in two rows is not empty and not equal, the two rows are not equal. Otherwise, the comparison result is null.

For operators <, <=, >, and >=, the fields in rows are compared from left to right until a pair of fields that are not equal or are empty are detected. If the pair of fields contains at least one null value, the comparison result is null. Otherwise, the comparison result of this pair of fields is the final result.

Example:

```
SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;  
result  
-----  
t  
(1 row)
```


9 Type Conversion

9.1 Overview

Context

SQL is a typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. DWS has an extensible type system that is more general and flexible than other SQL implementations. Hence, most type conversion behavior in DWS is governed by general rules. This allows the use of mixed-type expressions.

The DWS scanner/parser divides lexical elements into five fundamental categories: integers, floating-point numbers, strings, identifiers, and keywords. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with constant strings. For example, the query:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
label | value
-----+-----
Origin | (0,0)
(1 row)
```

has two literal constants, of type **text** and **point**. If a type is not specified for a string literal, then the placeholder type **unknown** is assigned initially.

There are four fundamental SQL constructs requiring distinct type conversion rules in the DWS parser:

- **Function calls**
Much of the SQL type system is built around a rich set of functions. Functions can have one or more arguments. Since SQL permits function overloading, the function name alone does not uniquely identify the function to be called. The parser must select the right function based on the data types of the supplied arguments.
- **Operators**
SQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions, operators can be overloaded, so the same problem of selecting the right operator exists.

- Value Storage

SQL **INSERT** and **UPDATE** statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

- **UNION, CASE, and related constructs**

Since all query results from a unionized **SELECT** statement must appear in a single set of columns, the types of the results of each **SELECT** clause must be matched up and converted to a uniform set. Similarly, the result expressions of a **CASE** construct must be converted to a common type so that the **CASE** expression as a whole has a known output type. The same holds for **ARRAY** constructs, and for the **GREATEST** and **LEAST** functions.

The system catalog **PG_CAST** stores information about the conversion between data types and how to perform the conversion. For details, see section "PG_CAST" in the *Data Warehouse Service (DWS) Developer Guide*.

The return type and conversion behavior of an expression are determined during semantic analysis. Data types are divided into several basic type categories, including **boolean**, **numeric**, **string**, **bitstring**, **datetime**, **timespan**, **geometric**, and **network**. Within each category there can be one or more preferred types, which are preferred when there is a choice of possible types. With careful selection of preferred types and available implicit casts, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in a useful way.

All type conversion rules are designed based on the following principles:

- Implicit conversions should never have surprising or unpredictable outcomes.
- There should be no extra overhead in the parser or executor if a query does not need implicit type conversion. That is, if a query is well-formed and the types already match, then the query should execute without spending extra time in the parser and without introducing unnecessary implicit conversion calls in the query.
- Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function.

Converting Empty Strings to Numeric Values in TD-Compatible Mode

- Different from the Oracle database, which processes an empty string as NULL, Teradata database converts an empty string to **0** by default. Therefore, when an empty string is queried, value **0** is found. Similarly, in TD-compatible mode, the empty string is converted to **0** of the corresponding numeric type by default. In addition, '-', '+', and ' ' are converted to **0** by default in TD-compatible mode, but an error is reported for a decimal point string. Example:

```
create table t1(no int,col varchar);
insert into t1 values(1,'');
insert into t1 values(2,null);
select * from t1 where col is null;
no | col
----+----
 2 |
(1 row)
select * from t1 where col="";
no | col
----+----
```

```
1 |  
(1 row)
```

- The method of converting an empty string into a numeric value in MySQL-compatible mode is the same as that in TD-compatible mode.

9.2 Operators

Operator Type Resolution

1. Select the operators to be considered from the **pg_operator** system catalog. Considered operators are those with the matching name and argument count. If the search path finds multiple available operators, only the most suitable one is considered.
2. Look for the best match.
 - a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. **unknown** literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
 - b. Run through all candidates and keep those with the most exact matches on input types. Domains are considered the same as their base type for this purpose. Keep all candidates if there are no exact matches. If only one candidate remains, use it; else continue to the next step.
 - c. Review all available candidate operators and select the one that supports the most preferred position types, which fall under the input data type's category, when type conversion is needed. Keep all candidates if none accepts preferred types. If only one candidate remains, use it; else continue to the next step.
 - d. If any input arguments are of **unknown** types, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survives these tests. If only one candidate remains, use it; else continue to the next step.
 - e. If there are both **unknown** and known-type arguments, and all the known-type arguments have the same type, assume that the **unknown** arguments are also of that type, and check which candidates can accept that type at the unknown-argument positions. If exactly one candidate passes this test, use it. Otherwise, an error is reported.

Examples

Example 1: factorial operator type resolution. There is only one factorial operator (postfix !) defined in the system catalog, and it takes an argument of type **bigint**. The scanner assigns an initial type of **bigint** to the argument in this query expression:

```
SELECT 40 ! AS "40 factorial";
      40 factorial
-----
815915283247897734345611269596115894272000000000
(1 row)
```

So the parser does a type conversion on the operand and the query is equivalent to:

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 2: string concatenation operator type resolution. A string-like syntax is used for working with string types and for working with complex extension types. Strings with unspecified type are matched with likely operator candidates. An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";
      text and unknown
-----
      abcdef
(1 row)
```

In this example, the parser looks for an operator whose parameters are of the text type. Such an operator is found.

Here is a concatenation of two values of unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";
      unspecified
-----
      abcdef
(1 row)
```

NOTE

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, **text**, is used as the specific type to resolve the unknown-type literals.

Example 3: absolute-value and negation operator type resolution. The DWS operator catalog has several entries for the prefix operator @. All the entries implement absolute-value operations for various numeric data types. One of these entries is for type **float8**, which is the preferred type in the numeric category. Therefore, DWS will use that entry when faced with an **unknown** input:

```
SELECT @ '-4.5' AS "abs";
      abs
-----
      4.5
(1 row)
```

Here the system has implicitly resolved the unknown-type literal as type **float8** before applying the chosen operator.

Example 4: array inclusion operator type resolution. The following is an example of resolving an operator with one known and one unknown input:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";
      is subset
-----
      t
(1 row)
```

 NOTE

In the **pg_operator** table of DWS, several entries correspond to the infix operator **<@**, but the only two that may accept an integer array on the left-hand side are array inclusion (**anyarray <@ anyarray**) and range inclusion (**anyelement <@ anyrange**). Because none of these polymorphic pseudo-types (see [Pseudo-Types](#)) is considered preferred, the parser cannot resolve the ambiguity on that basis. However, [2.e](#) tells it to assume that the unknown-type literal is of the same type as the other input, that is, integer array. Now only one of the two operators can match, so array inclusion is selected. (If you select range inclusion, an error will be reported because the string does not have the right format to be a range literal.)

9.3 Functions

Function Type Resolution

1. Select the functions to be considered from the **pg_proc** system catalog. If a non-schema-qualified function name was used, the functions in the current search path are considered. If a qualified function name was given, only functions in the specified schema are considered.
If the search path finds multiple functions of different argument types, a proper function in the path is considered.
2. Check for a function accepting exactly the input argument types. If the function exists, use it. Cases involving **unknown** will never find a match at this step.
3. If no exact match is found, see if the function call appears to be a special type conversion request.
4. Look for the best match.
 - a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. **unknown** literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
 - b. Run through all candidates and keep those with the most exact matches on input types. Domains are considered the same as their base type for this purpose. Keep all candidates if none has exact matches. If only one candidate remains, use it; else continue to the next step.
 - c. Run through all candidates and keep those that accept preferred types at the most positions where type conversion will be required. Keep all candidates if none accepts preferred types. If only one candidate remains, use it; else continue to the next step.
 - d. If any input arguments are of **unknown** types, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument.

Keep all candidates if none survives these tests. If only one candidate remains, use it; else continue to the next step.

- e. If there are both **unknown** and known-type arguments, and all the known-type arguments have the same type, assume that the **unknown** arguments are also of that type, and check which candidates can accept that type at the **unknown**-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Examples

Example 1: Use the rounding function argument type resolution as the first example. There is only one **round** function that takes two arguments; it takes a first argument of type **numeric** and a second argument of type **integer**. So the following query automatically converts the first argument of type **integer** to **numeric**:

```
SELECT round(4, 4);
round
-----
4.0000
(1 row)
```

That query is converted by the parser to:

```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type **numeric**, the following query will require no type conversion and therefore might be slightly more efficient:

```
SELECT round(4.0, 4);
```

Example 2: Use the substring function type resolution as the second example. There are several **substr** functions, one of which takes types **text** and **integer**. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category **string** (namely of type **text**).

```
SELECT substr('1234', 3);
substr
-----
34
(1 row)
```

If the string is declared to be of type **varchar**, as might be the case if it comes from a table, then the parser will try to convert it to become **text**:

```
SELECT substr(varchar '1234', 3);
substr
-----
34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

NOTE

The parser learns from the **pg_cast** catalog that **text** and **varchar** are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion is inserted in this case.

And, if the function is called with an argument of type **integer**, the parser will try to convert that to **text**:

```
SELECT substr(1234, 3);
substr
-----
34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (1234 AS text), 3);
substr
-----
34
(1 row)
```

9.4 Value Storage

Value Storage Type Resolution

1. Search for an exact match with the target column.
2. Try to convert the expression to the target type. This will succeed if there is a registered cast between the two types. If the expression is an unknown-type literal, the content of the literal string will be fed to the input conversion routine for the target type.
3. Check whether there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the **pg_cast** catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type **integer**. The parameter receives the destination column's **atttypmod** value (typically its declared length, although the interpretation of **atttypmod** varies for different data types), and may take a third boolean parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

Examples

Use the **character** storage type conversion as an example. For a target column declared as **character(20)** the following statement shows that the stored value is sized correctly:

```
CREATE TABLE x1
(
  customer_sk      integer,
  customer_id      char(20),
  first_name       char(6),
  last_name        char(8)
)
with (orientation = column,compression=middle)
distribute by hash (last_name);

INSERT INTO x1(customer_sk, customer_id, first_name) VALUES (3769, 'abcdef', 'Grace');

SELECT customer_id, octet_length(customer_id) FROM x1;
  customer_id | octet_length
-----+-----
  abcdef      |          20
(1 row)

DROP TABLE x1;
```

 NOTE

What has really happened here is that the two unknown literals are resolved to **text** by default, allowing the || operator to be resolved as **text** concatenation. Then the **text** result of the operator is converted to **bpchar** ("blank-padded char", the internal name of the **character** data type) to match the target column type. Since the conversion from **text** to **bpchar** is binary-coercible, this conversion does not insert any real function call. Finally, the sizing function **bpchar(bpchar, integer, boolean)** is found in the system catalog and used for the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

9.5 UNION, CASE, and Related Constructs

SQL UNION constructs must match up possibly dissimilar types to become a single result set. Since all query results from a **SELECT UNION** statement must appear in a single set of columns, the types of the results of each **SELECT** clause must be matched up and converted to a uniform set. Similarly, the result expressions of a **CASE** construct must be converted to a common type so that the **CASE** expression as a whole has a known output type. The same holds for **ARRAY** constructs, and for the **GREATEST** and **LEAST** functions.

Type Resolution for UNION, CASE, and Related Constructs

- If all inputs are of the same type, and it is not unknown, resolve as that type.
- If all inputs are of type **unknown**, resolve as type **text** (the preferred type of the string category). Otherwise, **unknown** inputs are ignored.
- If the entered values are not the same type, the query fails (except for the **unknown** type).
- When input types fall under the same category, the preferred type of that category is chosen, unless the union operation selects the type of the first branch as the chosen type.

 NOTE

typcategory in the **pg_type** system catalog indicates the data type category.
typispreferred indicates whether a type is preferred in **typcategory**.

- All the input is converted to the selected type. (The original length of a string is retained). Fail if there is not an implicit conversion from a given input to the selected type.
- If the input contains the json, txid_snapshot, sys_refcursor, or geometry type, **UNION** cannot be performed.

Type Resolution for CASE, COALESCE, IF, and IFNULL in TD-Compatible Mode

- If all inputs are of the same type, and it is not unknown, resolve as that type.
- If all inputs are of type **unknown**, resolve as type **text**.
- If inputs are of string type (including **unknown** which is resolved as type **text**) and digit type, resolve as the string type. If the inputs are not of the two types, fail.
- If the non-unknown inputs are all of the same type category, choose the input type which is a preferred type in that category, if there is one.
- Convert all inputs to the selected type. Fail if there is not an implicit conversion from a given input to the selected type.

Type Resolution for CASE, COALESCE, IF, and IFNULL in MySQL-Compatible Mode

- If all inputs are of the same type, and it is not unknown, resolve as that type.
- If all inputs are of type **unknown**, resolve as type **text**.
- If some inputs are of type **unknown** and the others are of a non-**unknown** type, resolve as that non-**unknown** type.
- If the inputs are of different non-**unknown** types, treat type **enum** as type **text** for comparison.
- If the non-**unknown** inputs are all of the same type, choose a preferred type, if there is one. If the inputs are of different types, resolve as type **text**.
- Convert all inputs to the selected type. Fail if there is not an implicit conversion from a given input to the selected type.

Examples

Example 1: Use type resolution with unknown types in a union as the first example. Here, the unknown-type literal **'b'** will be resolved to type **text**.

```
SELECT text 'a' AS "text" UNION SELECT 'b';
text
-----
a
b
(2 rows)
```

Example 2: Use type resolution in a simple union as the second example. The literal **1.2** is of type **numeric**, and the **integer** value **1** can be cast implicitly to **numeric**, so that type is used.

```
SELECT 1.2 AS "numeric" UNION SELECT 1;
numeric
-----
1
1.2
(2 rows)
```

Example 3: Use type resolution in a transposed union as the third example. Here, since type **real** cannot be implicitly cast to **integer**, but **integer** can be implicitly cast to **real**, the union result type is resolved as **real**.

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
real
-----
1
2.2
(2 rows)
```

Example 4: Use type resolution in the **COALESCE** function with input values of types **int** and **varchar** as the fourth example. Type resolution fails in ORA-compatible mode. The types are resolved as type **varchar** in TD-compatible mode, and as type **text** in MySQL-compatible mode.

```
-- Create the ora_db, td_db, and mysql_db databases by setting dbcompatibility to ORA, TD, and MySQL,
respectively:
CREATE DATABASE ora_db dbcompatibility = 'ORA';
CREATE DATABASE td_db dbcompatibility = 'TD';
CREATE DATABASE mysql_db dbcompatibility = 'MySQL';

-- Switch to the ora_db database:
\c ora_db
```

```
-- Create the t1 table:
ora_db=# CREATE TABLE t1(a int, b varchar(10));

-- Show the execution plan of a statement for querying the types int and varchar of input parameters for COALESCE:
ora_db=# EXPLAIN SELECT coalesce(a, b) FROM t1;
ERROR: COALESCE types integer and character varying cannot be matched
CONTEXT: referenced column: coalesce

-- Delete the table:
ora_db=# DROP TABLE t1;

-- Switch to the td_db database:
ora_db=# \c td_db

-- Create the t2 table:
td_db=# CREATE TABLE t2(a int, b varchar(10));

-- Show the execution plan of a statement for querying the types int and varchar of input parameters for COALESCE:
td_db=# EXPLAIN VERBOSE select coalesce(a, b) from t2;
QUERY PLAN
-----
id | operation | E-rows | E-distinct | E-width | E-costs
---+-----+-----+-----+-----+-----
1 | -> Data Node Scan on "__REMOTE_FQS_QUERY__" | 0 | | 0 | 0.00

Targetlist Information (identified by plan id)
-----
1 --Data Node Scan on "__REMOTE_FQS_QUERY__"
  Output: (COALESCE((t2.a)::character varying, t2.b))
  Node/s: All datanodes
  Remote query: SELECT COALESCE(a::character varying, b) AS "coalesce" FROM public.t2
(10 rows)

-- Delete the table:
td_db=# DROP TABLE t2;

-- Switch to the mysql_db database:
td_db=# \c mysql_db

-- Create the t3 table:
mysql_db=# CREATE TABLE t3(a int, b varchar(10));

-- Show the execution plan of a statement for querying the types int and varchar of input parameters for COALESCE:
mysql_db=# EXPLAIN VERBOSE select coalesce(a, b) from t3;
QUERY PLAN
-----
id | operation | E-rows | E-distinct | E-width | E-costs
---+-----+-----+-----+-----+-----
1 | -> Data Node Scan on "__REMOTE_FQS_QUERY__" | 0 | | 0 | 0.00

Targetlist Information (identified by plan id)
-----
1 --Data Node Scan on "__REMOTE_FQS_QUERY__"
  Output: (COALESCE((t3.a)::text, (t3.b)::text))
  Node/s: All datanodes
  Remote query: SELECT COALESCE(a::text, b::text) AS "coalesce" FROM public.t3
(10 rows)

-- Delete the table:
mysql_db=# DROP TABLE t3;

-- Switch to the gaussdb database.
mysql_db=# \c gaussdb

-- Delete the databases:
```

```
DROP DATABASE ora_db;  
DROP DATABASE td_db;  
DROP DATABASE mysql_db;
```

10 Full Text Search

10.1 Introduction

10.1.1 Full-Text Retrieval

Textual search operators have been used in databases for years. The DWS has ~, ~*, **LIKE**, and **ILIKE** operators for textual data types, but they lack many essential properties required by modern information systems. This problem can be solved by using indexes and dictionaries.

Text search lacks the following essential properties required by information systems:

- There is no linguistic support, even for English.
Regular expressions are not sufficient because they cannot easily handle derived words. For example, you might miss documents that contain **satisfies**, although you probably would like to find them when searching for **satisfy**. It is possible to use **OR** to search for multiple derived forms, but this is tedious and error-prone, because some words can have several thousand derivatives.
- They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
- They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be preprocessed and an index is saved for later rapid searching. Preprocessing includes:

- Parsing documents into tokens
It is useful to identify various classes of tokens, for example, numbers, words, complex words, and email addresses, so that they can be processed differently. In principle, token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes.
- Converting tokens into lexemes
A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often

involves removal of suffixes (such as **s** or **es** in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates stop words, which are words that are so common that they are useless for searching. (In short, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) DWS uses dictionaries to perform this step and provides various standard dictionaries.

- Storing preprocessed documents optimized for searching

For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes, it is often desirable to store positional information for proximity ranking. Therefore, a document that contains a more "dense" region of query words is assigned with a higher rank than the one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can define stop words that should not be indexed.

A data type **tsvector** is provided for storing preprocessed documents, along with a type **tsquery** for storing query conditions. For details, see [Text Search Types](#). For details about the functions and operators provided for the **tsvector** data type, see [Text Search Functions and Operators](#). The matching operator **@@** is the most important. For details, see [Basic Text Matching](#).

10.1.2 What Is a Document?

A document is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (keywords) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within DWS, a document is normally a textual column within a row of a database table, or possibly a combination (concatenation) of such columns, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```
SELECT d_dow || '-' || d_dom || '-' || d_fy_week_seq AS identify_serials FROM tpeds.date_dim WHERE  
d_fy_week_seq = 1;  
identify_serials
```

```
-----  
5-6-1  
0-8-1  
2-3-1  
3-4-1  
4-5-1  
1-2-1  
6-7-1  
(7 rows)
```

NOTICE

Actually, in these example queries, **coalesce** should be used to prevent a single **NULL** attribute from causing a **NULL** result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute

searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires system administrator permissions or special function support, so this is less convenient than keeping all the data inside the database. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed **tsvector** format. Searching and relevance-based ranking are performed entirely on the **tsvector** representation of a document. The original text is retrieved only when the document has been selected for display to a user. We therefore often speak of the **tsvector** as being the document, but it is only a compact representation of the full document.

10.1.3 Basic Text Matching

Full text search in DWS is based on the match operator **@@**, which returns **true** if a **tsvector** (document) matches a **tsquery** (query). It does not matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery AS RESULT;
result
-----
t
(1 row)
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector AS RESULT;
result
-----
f
(1 row)
```

As the above example suggests, a **tsquery** is not raw text, any more than a **tsvector** is. A **tsquery** contains search terms, which must be already-normalized lexemes, and may combine multiple terms using **AND**, **OR**, and **NOT** operators. For details, see [Text Search Types](#). There are functions **to_tsquery** and **plainto_tsquery** that are helpful in converting user-written text into a proper **tsquery**, for example by normalizing words appearing in the text. Similarly, **to_tsvector** is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat') AS RESULT;
result
-----
t
(1 row)
```

Observe that this match would not succeed if written as follows:

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat') AS RESULT;
result
-----
f
(1 row)
```

In the preceding match, no normalization of the word **rats** will occur. Therefore, **rats** does not match **rat**.

The **@@** operator also supports text input, allowing explicit conversion of a text string to **tsvector** or **tsquery** to be skipped in simple cases. The variants available are:

```
tsvector @@ tsquery
tsquery  @@ tsvector
```

```
text @@ tsquery  
text @@ text
```

The form **text @@ tsquery** is equivalent to **to_tsvector(text) @@ tsquery**. The form **text @@ text** is equivalent to **to_tsvector(text) @@ plainto_tsquery(text)**.

10.1.4 Configurations

Full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, for example, parse based on more than just white space. This functionality is controlled by text search configurations. DWS comes with predefined configurations for many languages, and you can easily create your own configurations. (The `\df` command of **gsql** shows all available configurations.)

During installation an appropriate configuration is selected and **default_text_search_config** is set accordingly in **postgresql.conf**. If you are using the same text search configuration for the entire cluster you can use the value in **postgresql.conf**. To use different configurations throughout the cluster but the same configuration within any one database, use `ALTER DATABASE ... SET`. Otherwise, you can set **default_text_search_config** in each session.

Each text search function that depends on a configuration has an optional argument, so that the configuration to use can be specified explicitly. **default_text_search_config** is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. DWS's text search facility provides the following types of configuration-related database objects:

- Text search parsers break documents into tokens and classify each token (for example, as words or numbers).
- Text search dictionaries convert tokens to normalized form and reject stop words.
- Text search templates provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- Text search configurations select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

10.2 Tables and Indexes

10.2.1 Searching a Table

It is possible to do a full text search without an index.

- A simple query to print each row that contains the word **science** in its **body** column is as follows:

```
DROP SCHEMA IF EXISTS tsearch CASCADE;  
  
CREATE SCHEMA tsearch;  
  
CREATE TABLE tsearch.pgweb(id int, body text, title text, last_mod_date date);  
  
INSERT INTO tsearch.pgweb VALUES(1, 'Philology is the study of words, especially the history and
```

```
development of the words in a particular language or group of languages.', 'Philology', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(2, 'Mathematics is the science that deals with the logic of shape,
quantity and arrangement.', 'Mathematics', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(3, 'Computer science is the study of processes that interact with
data and that can be represented as data in the form of programs.', 'Computer science', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(4, 'Chemistry is the scientific discipline involved with elements
and compounds composed of atoms, molecules and ions.', 'Chemistry', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(5, 'Geography is a field of science devoted to the study of the
lands, features, inhabitants, and phenomena of the Earth and planets.', 'Geography', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(6, 'History is a subject studied in schools, colleges, and
universities that deals with events that have happened in the past.', 'History', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(7, 'Medical science is the science of dealing with the
maintenance of health and the prevention and treatment of disease.', 'Medical science', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(8, 'Physics is one of the most fundamental scientific disciplines,
and its main goal is to understand how the universe behaves.', 'Physics', '2010-1-1');
```

```
SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector('english', body) @@
to_tsquery('english', 'science');
```

id	body	title

+-----+

2 Mathematics is the science that deals with the logic of shape, quantity and arrangement.	Mathematics	
3 Computer science is the study of processes that interact with data and that can be represented as data in the form of programs.	Computer science	
5 Geography is a field of science devoted to the study of the lands, features, inhabitants, and phenomena of the Earth and planets.	Geography	
7 Medical science is the science of dealing with the maintenance of health and the prevention and treatment of disease.	Medical science	

(4 rows)

This will also find related words, such as **science**, since all these are reduced to the same normalized lexeme.

The query above specifies that the **english** configuration is to be used to parse and normalize the strings. Alternatively, you can omit the configuration parameters, and use the configuration set by **default_text_search_config**.

```
SHOW default_text_search_config;
default_text_search_config
```

```
pg_catalog.english
(1 row)
```

```
SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector(body) @@ to_tsquery('science');
```

id	body	title

+-----+

2 Mathematics is the science that deals with the logic of shape, quantity and arrangement.	Mathematics	
3 Computer science is the study of processes that interact with data and that can be represented as data in the form of programs.	Computer science	
5 Geography is a field of science devoted to the study of the lands, features, inhabitants, and phenomena of the Earth and planets.	Geography	
7 Medical science is the science of dealing with the maintenance of health and the prevention and treatment of disease.	Medical science	

(4 rows)

- A more complex example to select the ten most recent documents that contain **treatment** and **science** in the **title** or **body** column is as follows:

```
SELECT title FROM tsearch.pgweb WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('treatment & science') ORDER BY last_mod_date DESC LIMIT 10;
```

```
title
```

```
-----
```

```
Medical science
```

```
(1 rows)
```

For clarity we omitted the **coalesce** function calls which would be needed to find rows that contain **NULL** in one of the two columns.

The preceding examples show queries without using indexes. Most applications will find this approach too slow. Therefore, practical use of text searching usually requires creating an index, except perhaps for occasional ad-hoc searches.

10.2.2 Creating an Index

You can create a **GIN** index to speed up text searches:

```
CREATE INDEX pgweb_idx_1 ON tsearch.pgweb USING gin(to_tsvector('english', body));
```

The `to_tsvector()` function accepts one or two arguments.

If the one-argument version of the index is used, the system will use the configuration specified by **default_text_search_config** by default.

To create an index, the two-argument version must be used, or the index content may be inconsistent. Only text search functions that specify a configuration name can be used in expression indexes. Index content is not affected by **default_text_search_config**, because different entries could contain **tsvectors** that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of **to_tsvector** was used in the index above, only a query reference that uses the two-argument version of **to_tsvector** with the same configuration name will use that index. That is, **WHERE to_tsvector('english', body) @@ 'a & b'** can use the index, but **WHERE to_tsvector(body) @@ 'a & b'** cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

More complex expression indexes can be set up when the configuration name of the index is specified by another column. For example:

```
CREATE INDEX pgweb_idx_2 ON tsearch.pgweb USING gin(to_tsvector('zhparser', body));
```

NOTE

In this example, `zhparser` supports only the UTF-8 or GBK database encoding format. If the `SQL_ASCII` encoding is used, an error will be reported.

where **body** is a column in the **pgweb** table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, for example, **WHERE to_tsvector(config_name, body) @@ 'a & b'** must match **to_tsvector** in the index.

Indexes can even concatenate columns:

```
CREATE INDEX pgweb_idx_3 ON tsearch.pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

Another approach is to create a separate **tsvector** column to hold the output of **to_tsvector**. This example is a concatenation of **title** and **body**, using **coalesce** to ensure that one column will still be indexed when the other is **NULL**:

```
ALTER TABLE tsearch.pgweb ADD COLUMN textsearchable_index_col tsvector;  
UPDATE tsearch.pgweb SET textsearchable_index_col = to_tsvector('english', coalesce(title,'') || ' ' ||  
coalesce(body,''));
```

Then, create a GIN index to speed up the search:

```
CREATE INDEX textsearch_idx_4 ON tsearch.pgweb USING gin(textsearchable_index_col);
```

Now you are ready to perform a fast full text search:

```
SELECT title  
FROM tsearch.pgweb  
WHERE textsearchable_index_col @@ to_tsquery('science & Computer')  
ORDER BY last_mod_date DESC  
LIMIT 10;
```

```
title  
-----  
Computer science  
  
(1 rows)
```

One advantage of the separate-column approach over an expression index is that it is unnecessary to explicitly specify the text search configuration in queries in order to use the index. As shown in the preceding example, the query can depend on **default_text_search_config**. Another advantage is that searches will be faster, since it will not be necessary to redo the **to_tsvector** calls to verify index matches. The expression-index approach is simpler to set up, however, and it requires less disk space since the **tsvector** representation is not stored explicitly.

10.2.3 Constraints on Index Use

The following is an example of using an index:

```
create table table1 (c_int int,c_bigint bigint,c_varchar varchar,c_text text) with(orientation=row);  
  
create text search configuration ts_conf_1(parser=POUND);  
create text search configuration ts_conf_2(parser=POUND) with(split_flag='%');  
  
set default_text_search_config='ts_conf_1';  
create index idx1 on table1 using gin(to_tsvector(c_text));  
  
set default_text_search_config='ts_conf_2';  
create index idx2 on table1 using gin(to_tsvector(c_text));  
  
select c_varchar,to_tsvector(c_varchar) from table1 where to_tsvector(c_text) @@ plainto_tsquery('#@.....  
&**) and to_tsvector(c_text) @@ plainto_tsquery('company') and c_varchar is not null order by 1 desc limit  
3;
```

In this example, **table1** has two GIN indexes created on the same column **c_text**, **idx1** and **idx2**, but these two indexes are created under different settings of **default_text_search_config**. Differences between this example and the scenario where one table has common indexes created on the same column are as follows:

- GIN indexes use different parsers (that is, different delimiters). In this case, the index data of **idx1** is different from that of **idx2**.

- In the specified scenario, the index data of multiple common indexes created on the same column is the same.

As a result, using **idx1** and **idx2** for the same query returns different results.

Constraints

Still use the above example. When:

- Multiple GIN indexes are created on the same column of the same table.
- The GIN indexes use different parsers (that is, different delimiters).
- The column is used in a query, and an index scan is used in the execution plan.

To avoid different query results caused by different GINs, ensure that only one GIN is available on a column of a physical table.

10.3 Controlling Text Search

10.3.1 Parsing Documents

DWS provides function **to_tsvector** for converting a document to the **tsvector** data type.

to_tsvector([*config regconfig*,] *document text*) returns **tsvector**

to_tsvector parses a textual document into tokens, reduces the tokens to lexemes, and returns a **tsvector**, which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
       to_tsvector
-----
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the preceding example we see that the resulting **tsvector** does not contain the words **a**, **on**, or **it**, the word **rats** became **rat**, and the punctuation sign (-) was ignored.

The **to_tsvector** function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries is consulted. where the list can vary depending on the token type. The first dictionary that recognizes the token emits one or more normalized lexemes to represent the token. For example:

- **rats** became **rat** because one of the dictionaries recognized that the word **rats** is a plural form of **rat**.
- Some words are recognized as stop words (see [Stop Words](#)), which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are **a**, **on**, and **it**.
- If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign (-) because there are in fact no dictionaries assigned for its token type (**space symbols**), meaning space tokens will never be indexed.

The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration. It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration **english** for the English language.

The function **setweight** can be used to label the entries of a **tsvector** with a given weight, where a weight is one of the letters **A**, **B**, **C**, or **D**. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because **to_tsvector(NULL)** will return **NULL**, you are advised to use **coalesce** whenever a column might be **NULL**. Here is the recommended method for creating a **tsvector** from a structured document:

```
CREATE TABLE tsearch.tt (id int, title text, keyword text, abstract text, body text, ti tsvector);

INSERT INTO tsearch.tt(id, title, keyword, abstract, body) VALUES (1, 'book', 'literature', 'Ancient poetry', 'Tang poem Song jambic verse');

UPDATE tsearch.tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A') ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');

DROP TABLE tsearch.tt;
```

Here we have used **setweight** to label the source of each lexeme in the finished **tsvector**, and then merged the labeled **tsvector** values using the **tsvector** concatenation operator **||**. For details about these operations, see [Manipulating tsvector](#).

10.3.2 Parsing Queries

DWS provides functions **to_tsquery** and **plainto_tsquery** for converting a query to the **tsquery** data type. **to_tsquery** offers access to more features than **plainto_tsquery**, but is less forgiving about its input.

to_tsquery([config regconfig,] querytext text) returns **tsquery**

to_tsquery creates a **tsquery** value from **querytext**, which must consist of single tokens separated by the Boolean operators **&** (AND), **|** (OR), and **!** (NOT). These operators can be grouped using parentheses. In other words, the input to **to_tsquery** must already follow the general rules for **tsquery** input, as described in [Text Search Types](#). The difference is that while basic **tsquery** input takes the tokens at face value, **to_tsquery** normalizes each token to a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
-----
'fat' & 'rat'
(1 row)
```

As in basic **tsquery** input, **weight(s)** can be attached to each lexeme to restrict it to match only **tsvector** lexemes of those **weight(s)**. For example:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
to_tsquery
-----
```

```
'fat' | 'rat':AB  
(1 row)
```

Also, the asterisk (*) can be attached to a lexeme to specify prefix matching:

```
SELECT to_tsquery('supern:*A & star:A*B');  
to_tsquery  
-----  
'supern':*A & 'star':*AB  
(1 row)
```

Such a lexeme will match any word having the specified string and weight in a **tsquery**.

plainto_tsquery([config regconfig,] querytext text) returns **tsquery**

plainto_tsquery transforms unformatted text **querytext** to **tsquery**. The text is parsed and normalized much as for **to_tsvector**, then the **&** (AND) Boolean operator is inserted between surviving words.

For example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');  
plainto_tsquery  
-----  
'fat' & 'rat'  
(1 row)
```

Note that **plainto_tsquery** cannot recognize Boolean operators, weight labels, or prefix-match labels in its input:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');  
plainto_tsquery  
-----  
'fat' & 'rat' & 'c'  
(1 row)
```

Here, all the input punctuation was discarded as being space symbols.

10.3.3 Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. DWS provides two predefined ranking functions. They take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and application-specific. Different applications might require additional information for ranking, for example, document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer]) returns float4

Ranks vectors based on the frequency of their matching lexemes.

ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer]) returns float4

This function requires positional information in its input. Therefore, it will not work on "stripped" **tsvector** values. It will always return zero.

For both these functions, the optional **weights** argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

```
{D-weight, C-weight, B-weight, A-weight}
```

If no **weights** are provided, then these defaults are used: {0.1, 0.2, 0.4, 1.0}

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size. For example, a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer **normalization** option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using a vertical bar (|) (for example, 2|4).

- 0 (the default) ignores the document length
- 1 divides the rank by (1 + Logarithm of the document length)
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by ts_rank_cd)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by (1 + Logarithm of the number of unique words in document)
- 32 divides the rank by (itself + 1)

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 (**rank/(rank+1)**) can be used to scale all ranks into the range zero to one. This is just a cosmetic change, and it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

Run the following statements in a database that uses the UTF-8 or GBK encoding:

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query) AS rank
FROM tsearch.pgweb, to_tsquery('science') query
WHERE query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
id | title | rank
---+-----+-----
11 | Philology | .2
2 | Mathematics | .1
12 | Geography | .1
13 | Computer science | .1
(4 rows)
```

This is the same example using normalized ranking:

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query, 32 /* rank/(rank+1) */) AS rank
FROM tsearch.pgweb, to_tsquery('science') query
WHERE query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
id | title | rank
-----+-----
11 | Philology | .166667
2 | Mathematics | .0909091
12 | Geography | .0909091
13 | Computer science | .0909091
(4 rows)
```

The following example sorts query by Chinese word segmentation:

```
CREATE TABLE tsearch.ts_zhparser(id int, body text);
INSERT INTO tsearch.ts_zhparser VALUES (1, 'sort');
INSERT INTO tsearch.ts_zhparser VALUES(2, 'sort query');
INSERT INTO tsearch.ts_zhparser VALUES(3, 'query sort');
-- Accurate match
SELECT id, body, ts_rank_cd (to_tsvector ('zhparser', body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery ('sort') query WHERE query @@ to_tsvector (body);
id | body | rank
-----+-----
1 | sort | .1
(1 row)

-- Fuzzy match
SELECT id, body, ts_rank_cd (to_tsvector ('zhparser', body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery ('sort') query WHERE query @@ to_tsvector ('zhparser', body);
id | body | rank
-----+-----
3 | query sort | .1
1 | sort | .1
2 | sort query | .1
(3 rows)
```

Ranking can be expensive since it requires consulting the **tsvector** of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

10.3.4 Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. DWS provides function **ts_headline** that implements this functionality.

ts_headline([**config regconfig**,] **document text**, **query tsquery** [, **options text**]) returns **text**

ts_headline accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by **config**. If **config** is omitted, the **default_text_search_config** configuration is used.

If an options string is specified it must consist of a comma-separated list of one or more **option=value** pairs. The available options are:

- **StartSel, StopSel**: The strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. You must double-quote these strings if they contain spaces or commas.
- **MaxWords, MinWords**: These numbers determine the longest and shortest headlines to output.

- **ShortWord**: Words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates common English articles.
- **HighlightAll**: Boolean flag. If **true** the whole document will be used as the headline, ignoring the preceding three parameters.
- **MaxFragments**: Maximum number of text excerpts or fragments to display. The default value of zero selects a non-fragment-oriented headline generation method. A value greater than zero selects fragment-based headline generation. This method finds text fragments with as many query words as possible and stretches those fragments around the query words. As a result query words are close to the middle of each fragment and have words on each side. Each fragment will be of at most **MaxWords** and words of length **ShortWord** or less are dropped at the start and end of each fragment. If not all query words are found in the document, then a single fragment of the first **MinWords** in the document will be displayed.
- **FragmentDelimiter**: When more than one fragment is displayed, the fragments will be separated by this string.

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>,  
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,  
MaxFragments=0, FragmentDelimiter=" ... "
```

For example:

```
SELECT ts_headline('english',  
'The most common type of search  
is to find all documents containing given query terms  
and return them in order of their similarity to the  
query.',  
to_tsquery('english', 'query & similarity'));  
      ts_headline  
-----  
containing given <b>query</b> terms  
and return them in order of their <b>similarity</b> to the  
<b>query</b>.  
(1 row)  
  
SELECT ts_headline('english',  
'The most common type of search  
is to find all documents containing given query terms  
and return them in order of their similarity to the  
query.',  
to_tsquery('english', 'query & similarity'),  
'StartSel = <, StopSel = >');  
      ts_headline  
-----  
containing given <query> terms  
and return them in order of their <similarity> to the  
<query>.  
(1 row)
```

ts_headline uses the original document, not a **tsvector** summary, so it can be slow and should be used with care.

10.4 Additional Features

10.4.1 Manipulating tsvector

DWS provides functions and operators that can be used to manipulate documents that are already in tsvector type.

- `tsvector || tsvector`

The tsvector concatenation operator returns a new tsvector which combines the lexemes and positional information of the two tsvectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand tsvector are offset by the largest position mentioned in the left-hand tsvector, so that the result is nearly equivalent to the result of performing **to_tsvector** on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the tsvector form, rather than concatenating text before applying **to_tsvector**, is that you can use different configurations to parse different sections of the document. Also, because the **setweight** function marks all lexemes of the given tsvector the same way, it is necessary to parse the text and do **setweight** before concatenating if you want to label different parts of the document with different weights.

- `setweight(vector tsvector, weight "char")` returns tsvector

setweight returns a copy of the input tsvector in which every position has been labeled with the given weight, either **A**, **B**, **C**, or **D**. (**D** is the default for new tsvectors and as such is not displayed on output.) These labels are retained when tsvectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

NOTICE

Note that weight labels apply to positions, not lexemes. If the input tsvector has been stripped of positions then **setweight** does nothing.

- `length(vector tsvector)` returns integer

Returns the number of lexemes stored in the vector.

- `strip(vector tsvector)` returns tsvector

Returns a tsvector which lists the same lexemes as the given tsvector, but which lacks any position or weight information. While the returned tsvector is much less useful than an unstripped tsvector for relevance ranking, it will usually be much smaller.

10.4.2 Manipulating Queries

DWS provides functions and operators that can be used to manipulate queries that are already in tsquery type.

- `tsquery && tsquery`

Returns the AND-combination of the two given tsqueries.

- `tsquery || tsquery`
Returns the OR-combination of the two given tsqueries.
- `!! tsquery`
Returns the negation (NOT) of the given tsquery.
- `numnode(query tsquery)` returns integer
Returns the number of nodes (lexemes plus operators) in a **tsquery**. This function is useful to determine if the query is meaningful (returns > 0), or contains only stop words (returns 0). For example:

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE: text-search query contains only stop words or doesn't contain lexemes, ignored
CONTEXT: referenced column: numnode
 numnode
-----
      0

SELECT numnode('foo & bar'::tsquery);
 numnode
-----
      3
```

- `querytree(query tsquery)` returns text
Returns the portion of a **tsquery** that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----
      T
(1 row)
```

10.4.3 Rewriting Queries

The **ts_rewrite** family of functions searches a given **tsquery** for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a **tsquery** specific version of substring replacement. A target and substitute combination can be thought of as a query rewrite rule. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (that is, new york, big apple, nyc, gotham) or narrow the search to direct the user to some hot topic.

- `ts_rewrite (query tsquery, target tsquery, substitute tsquery)` returns tsquery
This form of **ts_rewrite** simply applies a single rewrite rule: **target** is replaced by **substitute** wherever it appears in query. For example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
-----
 'b' & 'c'
```

- `ts_rewrite (query tsquery, select text)` returns tsquery
This form of **ts_rewrite** accepts a starting query and a SQL select command, which is given as a text string. The **select** must yield two columns of **tsquery** type. For each row of the select result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current **query** value.

 NOTE

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to **ORDER BY** some ordering key.

Consider a real-life astronomical example. We will expand query supernovae using table-driven rewriting rules:

```
CREATE TABLE tsearch.alias (id int, t tsquery, s tsquery);

INSERT INTO tsearch.alias VALUES(1, to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.alias');

      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' )
```

We can change the rewriting rules just by updating the table:

```
UPDATE tsearch.alias
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.alias');

      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' & '!nebula' )
```

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the **tsquery** type. In the example below, we select only those rules which might match the original query:

```
SELECT ts_rewrite('a & b':tsquery, 'SELECT t,s FROM tsearch.alias WHERE "a & b":tsquery @> t');

      ts_rewrite
-----
'b' & 'a'
(1 row)
DROP TABLE ts_rewrite;
```

10.4.4 Gathering Document Statistics

The function **ts_stat** is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlquery is a text value containing an SQL query which must return a single **tsvector** column. **ts_stat** executes the query and returns statistics about each distinct lexeme (word) contained in the **tsvector** data. The columns returned are:

- **word text**: the value of a lexeme.
- **ndoc integer**: number of documents (**tsvectors**) the word occurred in.
- **nentry integer**: total number of occurrences of the word.

If **weights** are supplied, only occurrences having one of those weights are counted. For example, to find the ten most frequent words in a document collection:

```
SELECT * FROM ts_stat('SELECT to_tsvector("english", sr_reason_sk) FROM tpccs.store_returns WHERE sr_customer_sk < 10') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;;
```

```
word | ndoc | nentry
-----+-----
32 | 2 | 2
33 | 2 | 2
1 | 1 | 1
10 | 1 | 1
13 | 1 | 1
14 | 1 | 1
15 | 1 | 1
17 | 1 | 1
20 | 1 | 1
22 | 1 | 1
(10 rows)
```

The same, but counting only word occurrences with weight **A** or **B**:

```
SELECT * FROM ts_stat('SELECT to_tsvector("english", sr_reason_sk) FROM tpccs.store_returns WHERE
sr_customer_sk < 10', 'a') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;
word | ndoc | nentry
-----+-----
(0 rows)
```

10.5 Parsers

Text search parsers are responsible for splitting raw document text into tokens and identifying each token's type, where the set of types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries.

Currently, DWS provides the following built-in parsers: `pg_catalog.default` for English configuration, and `pg_catalog.ngram`, `pg_catalog.zhparser`, and `pg_catalog.pound` for full text search in texts containing Chinese, or both Chinese and English.

The built-in parser is named **`pg_catalog.default`**. It recognizes 23 token types, shown in [Table 10-1](#).

Table 10-1 Default parser's token types

Alias	Description	Examples
<code>asciiword</code>	Word, all ASCII letters	elephant
<code>word</code>	Word, all letters	mañana
<code>numword</code>	Word, letters and digits	beta1
<code>asciihword</code>	Hyphenated word, all ASCII	up-to-date
<code>hword</code>	Hyphenated word, all letters	lógico-matemática
<code>numhword</code>	Hyphenated word, letters and digits	postgresql-beta1
<code>hword_ascii</code>	Hyphenated word part, all ASCII	postgresql in the context postgresql-beta1

Alias	Description	Examples
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática
hword_numpart	Hyphenated word part, letters and digits	beta1 in the context postgresql-beta1
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html, in the context of a URL
file	File or path name	/usr/local/foo.txt, if not within a URL
sfloat	Scientific notation	-1.23E+56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

Note: The parser's notion of a "letter" is determined by the database's locale setting, specifically **lc_ctype**. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types word and asciiword should be treated alike.

email does not support all valid email characters as defined by RFC 5322. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to identify overlapping tokens in the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```
SELECT alias, description, token FROM ts_debug('english','foo-bar-beta1');
  alias | description | token
-----+-----+-----
numhword | Hyphenated word, letters and digits | foo-bar-beta1
```

```
hword_asciipart | Hyphenated word part, all ASCII | foo
blank           | Space symbols | -
hword_asciipart | Hyphenated word part, all ASCII | bar
blank           | Space symbols | -
hword_numpart   | Hyphenated word part, letters and digits | beta1
```

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```
SELECT alias, description, token FROM ts_debug('english','http://example.com/stuff/index.html');
alias | description | token
-----+-----+-----
protocol | Protocol head | http://
url      | URL          | example.com/stuff/index.html
host     | Host         | example.com
url_path | URL path     | /stuff/index.html
```

N-gram is a mechanical word segmentation method, and applies to no semantic Chinese segmentation scenarios. The N-gram segmentation method ensures the completeness of the segmentation. However, to cover all the possibilities, it but adds unnecessary words to the index, resulting in a large number of index items. N-gram supports Chinese coding, including GBK and UTF-8. Six built-in token types are shown in [Table 10-2](#).

Table 10-2 Token types

Alias	Description
zh_words	chinese words
en_word	english word
numeric	numeric data
alnum	alnum string
grapsymbol	graphic symbol
multisymbol	multiple symbol

Zhparser is a dictionary-based semantic word segmentation method. The bottom-layer calls the Simple Chinese Word Segmentation (SCWS) algorithm (<https://github.com/hightman/scws>), which applies to Chinese segmentation scenarios. SCWS is a term frequency and dictionary-based mechanical Chinese words engine. It can split a whole paragraph Chinese text into words. The two Chinese coding formats, GBK and UTF-8, are supported. The 26 built-in token types are shown in [Table 10-3](#).

Table 10-3 Token types

Alias	Description
A	Adjective
B	Differentiation
C	Conjunction

Alias	Description
D	Adverb
E	Exclamation
F	Position
G	Lexeme
H	Preceding element
I	Idiom
J	Acronyms and abbreviations
K	Subsequent element
L	Common words
M	Numeral
N	Noun
O	Onomatopoeia
P	Preposition
Q	Quantifiers
R	Pronoun
S	Space
T	Time
U	Auxiliary word
V	Verb
W	Punctuation
X	Unknown
Y	Interjection
Z	Status words

Pound segments words in a fixed format. It is used to segment to-be-parsed nonsense Chinese and English words that are separated by fixed separators. It supports Chinese encoding (including GBK and UTF8) and English encoding (including ASCII). Pound has six pre-configured token types (as listed in [Table 10-4](#)) and supports five separators (as listed in [Table 10-5](#)). The default, the separator is #. Pound The maximum length of a token is 256 characters.

Table 10-4 Token types

Alias	Description
zh_words	chinese words
en_word	english word
numeric	numeric data
alnum	alnum string
grapsymbol	graphic symbol
multisymbol	multiple symbol

Table 10-5 Separator types

Delimiter	Description
@	Special character
#	Special character
\$	Special character
%	Special character
/	Special character

10.6 Dictionaries

10.6.1 Overview

A dictionary is used to define stop words, that is, words to be ignored in full-text retrieval.

A dictionary can also be used to normalize words so that different derived forms of the same word will match. A normalized word is called a lexeme.

In addition to improving retrieval quality, normalization and removal of stop words can reduce the size of the **tsvector** representation of a document, thereby improving performance. Normalization and removal of stop words do not always have linguistic meaning. Users can define normalization and removal rules in dictionary definition files based on application environments.

A dictionary is a program that receives a token as input and returns:

- An array of lexemes if the input token is known to the dictionary (note that one token can produce more than one lexeme).
- A single lexeme to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a filtering dictionary).

- An empty array if the input token is known to the dictionary but is a stop word.
- **NULL** if the dictionary does not recognize the token.

DWS provides predefined dictionaries for many languages and also provides five predefined dictionary templates, **Simple**, **Synonym**, **Thesaurus**, **Ispell**, and **Snowball**. These templates can be used to create new dictionaries with custom parameters.

When using full-text retrieval, you are advised to:

- In the text search configuration, configure a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until a dictionary recognizes it as a known word. If it is identified as a stop word, or no dictionary recognizes the token, it will be discarded and not indexed or searched for. Generally, the first dictionary that returns a non-**NULL** output determines the result, and any remaining dictionaries are not consulted. However, a filtering dictionary can replace the input token with a modified one, which is then passed to subsequent dictionaries.
- The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a **Snowball** stemmer dictionary or a **Simple** dictionary, which recognizes everything. In the following example, for an astronomy-specific search (**astro_en** configuration), you can configure the token type **asciiword** (ASCII word) with a **Synonym** dictionary of astronomical terms, a general English **Ispell** dictionary, and a **Snowball** English stemmer dictionary:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
ADD MAPPING FOR asciiword WITH astro_syn, english_ispell, english_stem;
```

A filtering dictionary can be placed anywhere in the list, except at the end where it would be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries.

10.6.2 Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. Each type of dictionaries treats stop words in different ways. For example, **Ispell** dictionaries first normalize words and then check the list of stop words, while **Snowball** dictionaries first check the list of stop words.

For example, every English text contains words like **a** and **the**, so it is useless to store them in an index. However, stop words affect the positions in **tsvector**, which in turn affect ranking.

```
SELECT to_tsvector('english','in the list of stop words');
       to_tsvector
-----
'list':3 'stop':5 'word':6
```

The missing positions 1, 2, and 4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
       ts_rank_cd
```

```
-----  
.05  
  
SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));  
ts_rank_cd  
-----  
.1
```

10.6.3 Simple Dictionary

A **Simple** dictionary operates by converting the input token to lower case and checking it against a list of stop words. If the token is found in the list, an empty array will be returned, causing the token to be discarded. If it is not found, the lower-cased form of the word is returned as the normalized lexeme. In addition, you can set **Accept to false** for **Simple** dictionaries (default: **true**) to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Precautions

- Most types of dictionaries rely on dictionary configuration files. The name of a configuration file can only be lowercase letters, digits, and underscores (_).
- A dictionary cannot be created in **pg_temp** mode.
- Dictionary configuration files must be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.
- Generally, a session will read a dictionary configuration file only once, when it is first used within the session. To modify a configuration file, run the **ALTER TEXT SEARCH DICTIONARY** statement to update and reload the file.

Procedure

Step 1 Create a **Simple** dictionary.

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (  
    TEMPLATE = pg_catalog.simple,  
    STOPWORDS = english  
);
```

english.stop is the full name of a file of stop words. For details about the syntax and parameters for creating a **Simple** dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

Step 2 Use the **Simple** dictionary.

```
SELECT ts_lexize('public.simple_dict','Yes');  
ts_lexize  
-----  
{yes}  
(1 row)  
  
SELECT ts_lexize('public.simple_dict','The');  
ts_lexize  
-----  
{}  
(1 row)
```

Step 3 Set **Accept=false** so that the **Simple** dictionary returns **NULL** instead of a lower-cased non-stop word.

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );  
SELECT ts_lexize('public.simple_dict','Yes');
```

```
ts_lexize
-----
(1 row)

SELECT ts_lexize('public.simple_dict','The');
ts_lexize
-----
{}
(1 row)
```

----End

10.6.4 Synonym Dictionary

A synonym dictionary is used to define, identify, and convert synonyms of tokens. Phrases are not supported (use the thesaurus dictionary in [Thesaurus Dictionary](#)).

Examples

- A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word "Paris" to "pari". It is enough to have a **Paris paris** line in the synonym dictionary and put it before the **english_stem** dictionary.

```
SELECT * FROM ts_debug('english', 'Paris');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}
(1 row)

CREATE TEXT SEARCH DICTIONARY my_synonym (
  TEMPLATE = synonym,
  SYNONYMS = my_synonyms,
  FILEPATH = 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=eu-west-101'
);

ALTER TEXT SEARCH CONFIGURATION english
  ALTER MAPPING FOR asciiword
  WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

ALTER TEXT SEARCH DICTIONARY my_synonym ( CASESENSITIVE=true);

SELECT * FROM ts_debug('english', 'Paris');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {pari}
(1 row)
```

The full name of the synonym dictionary file is **my_synonyms.syn**, and the dictionary is stored in 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=*eu-west-101*'. For details about the syntax and parameters for creating a synonym dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

- An asterisk (*) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in `to_tsvector()`, but when it is used in `to_tsquery()`, the result will be a query item with the prefix match marker (see [Manipulating Queries](#)).

Assume that the content in the dictionary file **synonym_sample.syn** is as follows:

```
postgres    pgsql
postgresql  pgsql
postgre     pgsql
gogle       googl
indices     index*
```

Create and use a dictionary.

```
CREATE TEXT SEARCH DICTIONARY syn (
  TEMPLATE = synonym,
  SYNONYMS = synonym_sample
);

SELECT ts_lexize('syn','indices');
ts_lexize
-----
{index}
(1 row)

CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);

ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;

SELECT to_tsvector('tst','indices');
to_tsvector
-----
'index':1
(1 row)

SELECT to_tsquery('tst','indices');
to_tsquery
-----
'index':*
(1 row)

SELECT 'indexes are very useful'::tsvector;
tsvector
-----
'are' 'indexes' 'useful' 'very'
(1 row)

SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
?column?
-----
t
(1 row)
```

10.6.5 Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that include relationships between words and phrases, such as broader terms (BT),

narrower terms (NT), preferred terms, non-preferred terms, and related terms. A thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. A thesaurus dictionary is an extension of the synonym dictionary with added phrase support.

Precautions

- A thesaurus dictionary has the capability to recognize phrases. Therefore, it must remember its state and interact with the parser to check whether it should handle the next token or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the **asciiword** token, then a thesaurus dictionary definition like **one 7** will not work because token type **uint** is not assigned to the thesaurus dictionary.
- Thesauruses are used during indexing. Any change in the thesaurus dictionary's parameters requires reindexing. For most other dictionary types, small changes such as adding or removing stop words does not force reindexing.

Procedure

Step 1 Create a TZ named **thesaurus_astro**.

thesaurus_astro is a simple astronomical TZ that defines two astronomical word combinations (word+synonym).

```
supernovae stars : sn  
crab nebulae : crab
```

Run the following statement to create the TZ:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (  
  TEMPLATE = thesaurus,  
  DictFile = thesaurus_astro,  
  Dictionary = pg_catalog.english_stem,  
  FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

The full name of the thesaurus dictionary file is **thesaurus_astro.ths**, and the dictionary is stored in the **obs://bucket_name/path accesskey=ak secretkey=sk region=rg** directory. **pg_catalog.english_stem** is the subdictionary (a **Snowball** English stemmer) used for input normalization. The subdictionary has its own configuration (for example, stop words), which is not shown here. For details about the syntax and parameters for creating a TZ, see [CREATE TEXT SEARCH DICTIONARY](#).

Step 2 Bind the TZ to the desired token types in the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION english  
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart  
  WITH thesaurus_astro, english_stem;
```

Step 3 Use the TZ.

- Test the TZ.

The **ts_lexize** function is not very useful for testing the TZ because the function processes its input as a single token. Instead, you can use the **plainto_tsquery**, **to_tsvector**, or **to_tsquery** function which will break their input strings into multiple tokens.

```
SELECT plainto_tsquery('english','supernova star');  
plainto_tsquery
```

```
-----
'sn'
(1 row)

SELECT to_tsvector('english','supernova star');
to_tsvector
-----
'sn':1
(1 row)

SELECT to_tsquery('english','"supernova star"');
to_tsquery
-----
'sn'
(1 row)
```

supernova star matches **supernovae stars** in **thesaurus_astro** because the **english_stem** stemmer of the Snowball type is specified in the **thesaurus_astro** definition. The stemmer removed **e** and **s**.

- To index the original phrase, include it in the right-hand part of the definition.
supernovae stars : sn supernovae stars

```
ALTER TEXT SEARCH DICTIONARY thesaurus_astro (
  DictFile = thesaurus_astro,
  FILEPATH = 'file:///home/dicts/');

SELECT plainto_tsquery('english','supernova star');
plainto_tsquery
-----
'sn' & 'supernova' & 'star'
(1 row)
```

----End

10.6.6 Ispell Dictionary

The Ispell dictionary template supports morphological dictionaries, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all declensions and conjugations of the search term **bank**, such as **banking**, **banked**, **banks**, **banks'**, and **bank's**.

DWS does not provide any predefined Ispell dictionaries or dictionary files. The **.dict** files and **.affix** files support multiple open-source dictionary formats, including **Ispell**, **MySpell**, and **Hunspell**.

Procedure

- Step 1** Obtain the dictionary definition file (.dict) and affix file (.affix).

You can use an open-source dictionary. The name extensions of the open-source dictionary may be **.aff** and **.dic**. In this case, you need to change them to **.affix** and **.dict**. In addition, for some dictionary files (for example, Norwegian dictionary files), you need to run the following commands to convert the character encoding to UTF-8:

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- Step 2** Create an Ispell dictionary.

```
CREATE TEXT SEARCH DICTIONARY norwegian_ispell (
  TEMPLATE = ispell,
  DictFile = nn_no,
  AffFile = nn_no,
```

```
FilePath = 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=eu-west-101'
);
```

The full name of the Ispell dictionary file is **nn_no.dict** and **nn_no.affix**, and the dictionary is stored in 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=eu-west-101'. For details about the syntax and parameters for creating an Ispell dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

Step 3 Use the Ispell dictionary to split compound words.

```
SELECT ts_lexize('norwegian_isspell', 'sjokoladefabrikk');
       ts_lexize
-----
{sjokolade,fabrikk}
(1 row)
```

MySpell does not support compound words. **Hunspell** supports compound words. DWS supports only the basic compound word operations of **Hunspell**. Generally, an Ispell dictionary recognizes a limited set of words, so they should be followed by another broader dictionary, for example, a Snowball dictionary, which recognizes everything.

----End

10.6.7 Snowball Dictionary

A **Snowball** dictionary is based on a project by Martin Porter and is used for stem analysis, providing stemming algorithms for many languages. DWS provides predefined **Snowball** dictionaries of many languages. You can query the **PG_TS_DICT** system catalog to view the predefined **Snowball** dictionaries and supported stemming algorithms.

Snowball dictionaries identify all inputs as recognized, whether simplified or not, so they should be placed at the end of the dictionary list. It is useless to place it before any other dictionary because a token will never pass it through to the next dictionary.

For details about the syntax of **Snowball** dictionaries, see [CREATE TEXT SEARCH DICTIONARY](#).

10.7 Configuration Examples

Text search configuration specifies the following components required for converting a document into a **tsvector**:

- A parser, decomposes a text into tokens.
- Dictionary list, converts each token into a lexeme.

Each time when the **to_tsvector** or **to_tsquery** function is invoked, a text search configuration is required to specify a processing procedure. The GUC parameter **default_text_search_config** specifies the default text search configuration, which will be used if the text search function does not explicitly specify a text search configuration.

DWS provides some predefined text search configurations. You can also create user-defined text search configurations. In addition, to facilitate the management of text search objects, multiple `gspl` meta-commands are provided to display

related information. For details, see "Meta-Command Reference" in the Data Warehouse Service (DWS) *Tool Guide*.

Procedure

- Step 1** Create a text search configuration **ts_conf** by copying the predefined text search configuration **english**.

```
CREATE TEXT SEARCH CONFIGURATION ts_conf ( COPY = pg_catalog.english );  
CREATE TEXT SEARCH CONFIGURATION
```

- Step 2** Create a **Synonym** dictionary.

Assume that the definition file **pg_dict.syn** of the **Synonym** dictionary contains the following contents:

```
postgres pg  
pgsql pg  
postgresql pg
```

Run the following statement to create the **Synonym** dictionary:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (  
    TEMPLATE = synonym,  
    SYNONYMS = pg_dict,  
    FILEPATH = 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=eu-west-101'  
);
```

- Step 3** Create an **Ispell** dictionary **english_ispell** (the dictionary definition file is from the open source dictionary).

```
CREATE TEXT SEARCH DICTIONARY english_ispell (  
    TEMPLATE = ispell,  
    DictFile = english,  
    AffFile = english,  
    StopWords = english,  
    FILEPATH = 'obs://bucket01/obs.example.com accesskey=xxxxx secretkey=xxxxx region=eu-west-101'  
);
```

- Step 4** Modify the text search configuration **ts_conf** and change the dictionary list for tokens of certain types. For details about token types, see [Parsers](#).

```
ALTER TEXT SEARCH CONFIGURATION ts_conf  
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,  
        word, hword, hword_part  
    WITH pg_dict, english_ispell, english_stem;
```

- Step 5** In the text search configuration, set non-index or set the search for tokens of certain types.

```
ALTER TEXT SEARCH CONFIGURATION ts_conf  
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

- Step 6** Use the text retrieval commissioning function **ts_debug()** to test the text search configuration **ts_conf**.

```
SELECT * FROM ts_debug('ts_conf', '  
PostgreSQL, the highly scalable, SQL compliant, open source object-relational  
database management system, is now undergoing beta testing of the next  
version of our software.  
' );
```

- Step 7** You can set the default text search configuration of the current session to **ts_conf**. This setting is valid only for the current session.

```
\dF+ ts_conf  
    Text search configuration "public.ts_conf"  
Parser: "pg_catalog.default"  
Token   | Dictionaries  
-----+-----
```



```
asciihword | pg_dict,english_ispell,english_stem
asciword   | pg_dict,english_ispell,english_stem
file       | simple
host       | simple
hword      | pg_dict,english_ispell,english_stem
hword_asciipart | pg_dict,english_ispell,english_stem
hword_numpart | simple
hword_part | pg_dict,english_ispell,english_stem
int        | simple
numhword   | simple
numword    | simple
uint       | simple
version    | simple
word       | pg_dict,english_ispell,english_stem
```

```
SET default_text_search_config = 'public.ts_conf';
SET
SHOW default_text_search_config;
default_text_search_config
-----
public.ts_conf
(1 row)
```

----End

10.8 Testing and Debugging Text Search

10.8.1 Testing a Configuration

The function `ts_debug` allows easy testing of a text search configuration.

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
returns setof record
```

ts_debug displays information about every token of document as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by **config**, or **default_text_search_config** if that argument is omitted.

ts_debug returns one row for each token identified in the text by the parser. The columns returned are:

- **alias text** — short name of the token type
- **description text** — description of the token type
- **token text** — text of the token
- **dictionaries regdictionary[]** — the dictionaries selected by the configuration for this token type
- **dictionary regdictionary**: the dictionary that recognized the token, or NULL if none did
- **lexemes text[]**: the lexeme(s) produced by the dictionary that recognized the token, or NULL if none did; an empty array ({}) means the token was recognized as a stop word

Here is a simple example:

```
SELECT * FROM ts_debug('english','a fat cat sat on a mat - it ate a fat rats');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | cat | {english_stem} | english_stem | {cat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | sat | {english_stem} | english_stem | {sat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | on | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | mat | {english_stem} | english_stem | {mat}
blank | Space symbols | | {} | | 
blank | Space symbols | - | {} | | 
asciiword | Word, all ASCII | it | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | ate | {english_stem} | english_stem | {ate}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | rats | {english_stem} | english_stem | {rat}
(24 rows)
```

10.8.2 Testing a Parser

The **ts_parse** function allows direct testing of a text search parser.

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts_parse parses the given **document** and returns a series of records, one for each token produced by parsing. Each record includes a **tokid** showing the assigned token type and a **token** which is the text of the token. For example:

```
SELECT * FROM ts_parse('default', '123 - a number');
tokid | token
-----+-----
22 | 123
12 | 
12 | -
1 | a
12 | 
1 | number
(6 rows)
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

ts_token_type returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer **tokid** that the parser uses to label a token of that type, the **alias** that names the token type in configuration commands, and a short description. For example:

```
SELECT * FROM ts_token_type('default');
tokid | alias | description
-----+-----+-----
1 | asciiword | Word, all ASCII
2 | word | Word, all letters
3 | numword | Word, letters and digits
4 | email | Email address
5 | url | URL
6 | host | Host
```

7	sfloat	Scientific notation
8	version	Version number
9	hword_numpart	Hyphenated word part, letters and digits
10	hword_part	Hyphenated word part, all letters
11	hword_asciipart	Hyphenated word part, all ASCII
12	blank	Space symbols
13	tag	XML tag
14	protocol	Protocol head
15	numhword	Hyphenated word, letters and digits
16	asciihword	Hyphenated word, all ASCII
17	hword	Hyphenated word, all letters
18	url_path	URL path
19	file	File or path name
20	float	Decimal notation
21	int	Signed integer
22	uint	Unsigned integer
23	entity	XML entity

(23 rows)

10.8.3 Testing a Dictionary

The **ts_lexize** function facilitates dictionary testing.

ts_lexize(dict regdictionary, token text) returns **text[]**: **ts_lexize** returns an array of lexemes if the input **token** is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or **NULL** if it is an unknown word.

For example:

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}

SELECT ts_lexize('english_stem', 'a');
ts_lexize
-----
{}
```

NOTICE

The **ts_lexize** function expects a single **token**, not text.

10.9 Limitations

The current limitations of DWS's full text search are:

- The length of each lexeme must be less than 2 KB.
- The length of a **tsvector** (lexemes + positions) must be less than 1 megabyte.
- Position values in **tsvector** must be greater than 0 and no more than 16,383.
- No more than 256 positions per lexeme. Excessive positions, if any, will be discarded.
- The number of nodes (lexemes + operators) in a tsquery must be less than 32,768.

11 System Operation

DWS runs SQL statements to perform different system operations, such as setting variables, displaying the execution plan, and collecting garbage data.

Setting Variables

For details about how to set various parameters for a session or transaction, see [SET](#).

Displaying the Execution Plan

For details about how to display the execution plan that DWS makes for SQL statements, see [EXPLAIN](#).

Specifying a Checkpoint in Transaction Logs

By default, WALs periodically specify checkpoints in a transaction log. **CHECKPOINT** forces an immediate checkpoint when the related command is issued, without waiting for a regular checkpoint scheduled by the system. For details, see [CHECKPOINT](#).

Collecting Unnecessary Data

For details about how to collect garbage data and analyze a database as required, For details, see [VACUUM](#).

Collecting statistics

For details about how to collect statistics on tables in databases, For details, see [ANALYZE | ANALYSE](#).

Setting the Constraint Check Mode for the Current Transaction

For details about how to set the constraint check mode for the current transaction, For details, see [SET CONSTRAINTS](#).

12 Transaction Management

DWS supports the ACID properties of database transactions. It provides the **READ COMMITTED** and **REPEATABLE READ** isolation levels of transactions.



WARNING

- In scenarios where the client explicitly starts a transaction or manually disables "autocommit", you must manually execute a commit command to commit the transaction.
 - You are advised to optimize the statements that take longer than 30 minutes to execute.
 - Avoid business operations that take longer than 2 hours to execute to prevent issues such as long transactions and long locks.
 - For more information about development design proposal, see [Development Design Proposal](#).
-

Concepts

- A transaction refers to an operation that consists of multiple steps, either all successful or all failed.
- A database transaction is a logical unit in the execution process of DBMS. It consists of a limited number of database operations in a certain sequence (generally all the operations between **BEGIN TRANSACTION** and **END TRANSACTION**). These operations are either fully completed or not performed at all.

Purposes

The purposes of database transactions are as follows:

- To provide a method for restoring the database operation sequence from a failure while keeping database consistency even in case of system failure.
- To provide isolation between programs accessing a database concurrently, so that they do not interfere one another in this process.

Transaction Execution Process

After a transaction is committed in DBMS, DBMS needs to ensure that all operations in the transaction are successfully completed and the results are permanently stored in the database. If an operation in the transaction fails, all the operations in the transaction must be rolled back to the status before the transaction is executed. Transactions run independently and do not interfere with each other or affect database running.

Transaction Properties

A transaction has atomicity, consistency, isolation, and durability (ACID) properties.

- **Atomicity:** All the operations in a transaction are inseparable. They are either fully completed or not executed at all.

For example, if account A transfers an amount of money to account B, \$500 are deducted from account A and \$500 added to account B. If the amount fails to be added to account B, it cannot be deducted from account A. If atomicity is not guaranteed, the account balances will be consistent.

- **Consistency:** A transaction can only bring the database from one valid state to another, maintaining database invariants. Any data written to the database must be valid according to all defined rules, including data accuracy, concatenation, and spontaneous execution of scheduled tasks.

For example, if account A transfers \$500 to account B, \$500 is deducted from account A and added to account B. The sum of the deducted amount (-\$500) and the added amount (+500) should be 0. The total account balance of accounts A and B remains unchanged, no matter whether the money is transferred.

- **Isolation:** The execution of a transaction cannot be interfered by other transactions. Isolation means that the operations inside a transaction and data used are isolated from other concurrent transactions. The concurrent transactions do not affect each other.

Transactions are often executed concurrently. Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Transaction isolation is divided into different levels, including **READ UNCOMMITTED**, **READ COMMITTED**, **REPEATABLE READ**, and **SERIALIZABLE**.

- **Durability:** Once a transaction is committed, its modifications are permanently saved to the database. Even if the system is faulty, committed modifications will not be lost.

Table 12-1 ACID usage

ACID	Purpose
Atomicity	Concurrency control and fault recovery
Consistency	SQL integrity constraints (primary key and foreign key constraints)

ACID	Purpose
Isolation	Concurrency control
Durability	Fault recovery

Common concurrency control technologies include lock-based and timestamp-based concurrency control. DWS uses the two-phase lock technology for DDL statements and uses multi-version concurrency control (MVCC) for DML statements. DWS databases fault recovery is based on WAL logs. MVCC mainly uses redo logs to ensure transaction read/write consistency.

Redo logs are critical components in database systems that ensure data durability and enable fault recovery. These logs record the modification operations—not the data before or after—performed by transactions. In the event of a failure, such as a system breakdown or crash, the database can replay these operations to restore data to a consistent state before the fault occurred.

Redo log recovery process: After a crash and subsequent restart, the database performs recovery using the redo log, which is known as Write-Ahead Log (WAL). It scans log entries from the last checkpoint—representing committed transactions that were not yet written to disk—and re-executes these operations to restore lost in-memory changes. Simultaneously, uncommitted transactions are rolled back using undo operations to reverse partial modifications. This ensures data is restored to a consistent and complete state, preserving durability of committed transactions and atomicity of incomplete ones. This mechanism is the core basis for databases to enable high reliability.

Isolation Levels

Isolation prevents data inconsistency during the execution of concurrent transactions. A transaction isolation level specifies how concurrent transactions process the same object.

In DWS, transaction isolation levels are controlled by the GUC parameter **transaction_isolation** or the **SET TRANSACTION** syntax. The following isolation levels are supported. The default isolation level is **READ COMMITTED**.

- **READ COMMITTED:** Only committed data is read.
- **READ UNCOMMITTED:** DWS does not support **READ UNCOMMITTED**. If **READ UNCOMMITTED** is set, **READ COMMITTED** is used instead.
- **REPEATABLE READ:** Only the data committed before transaction start is read. Uncommitted data or data committed in other concurrent transactions cannot be read.
- **SERIALIZABLE:** DWS does not support **SERIALIZABLE**. If **SERIALIZABLE** is set, **REPEATABLE READ** is used instead.

Transaction Control Syntax

- Starting a transaction

DWS starts a transaction using **START TRANSACTION** and **BEGIN**. For details, see **START TRANSACTION** and **BEGIN**.

- Setting a transaction
DWS sets a transaction using **SET TRANSACTION** or **SET LOCAL TRANSACTION**. For details, see [SET TRANSACTION](#).
 - Committing a transaction
DWS commits all operations of a transaction using **COMMIT** or **END**. For details, see [COMMIT | END](#).
 - Rolling back a transaction
If a fault occurs during a transaction and the transaction cannot proceed, the system performs rollback to cancel all the completed database operations related to the transaction. For details, see [ROLLBACK](#).
-  **NOTE**
- If an execution request (not in a transaction block) received in the database contains multiple statements, the statements will be packed into a transaction. If one of the statements fails, the entire request will be rolled back.
- Other operations on transactions
 - **SAVEPOINT** establishes a new savepoint within the current transaction. The transaction can be rolled back to the savepoint. You can roll back the commands executed after a savepoint but retain the commands executed before the savepoint. For details, see [SAVEPOINT](#).
 - **ROLLBACK TO SAVEPOINT** rolls back a transaction to a savepoint. It implicitly deletes all the savepoints established after that savepoint. For details, see [ROLLBACK TO SAVEPOINT](#).
 - **RELEASE SAVEPOINT** deletes a savepoint in a transaction. For details, see [RELEASE SAVEPOINT](#).

Transaction Example

A customer buys a \$100 item in a store using an e-payment account. At least two operations are involved: 1. \$100 is deducted from the customer's account. 2. \$100 is added to the store's account. In DBMS, the two operations must be both completed or not executed at all.

1. Create sample data.

Create an account balance table and insert data. (Assume the store's and the customer's accounts each have \$500.)

```
CREATE TABLE customer_info (  
  NAME VARCHAR(32) PRIMARY KEY,  
  MONEY INTEGER  
);  
INSERT INTO customer_info (name, money) VALUES ('buyer', 500), ('shop', 500);
```

The table data shows that the store and customer each have \$500.

```
SELECT * FROM customer_info;
```

name	money
buyer	500
shop	500
(2 rows)	

2. Simulate a successful transaction.

Deduct \$100 from the customer's account and add \$100 to the store's account.

```
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'buyer');
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'shop');
```

```
SELECT * FROM customer_info;
```

name	money
buyer	400
shop	600
(2 rows)	

3. Restore initial values.

```
UPDATE customer_info SET money=500;
select * from customer_info;
```

name	money
shop	500
buyer	500
(2 rows)	

4. Simulate a transaction failure.

\$100 is deducted from the customer's account but fails to be added to the store's account.

a. Deduct \$100 from the customer's account.

```
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'buyer');
```

b. The store finds a payment problem and terminates subsequent operations. An error is reported when the amount of money is added to the store's account. The execution of the following statement is terminated. (Only the store thinks that there is a problem with payment.)

```
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'shop');
```

c. Query the account balances. The consumer has paid \$100 but store does not receive it.

```
SELECT * FROM customer_info;
```

name	money
buyer	400
shop	500
(2 rows)	

Without ACID properties, the account balances will be incorrect once an error occurs during SQL statement execution..

Simulate the rollback of an abnormal database transaction.

1. Restore initial values.

```
UPDATE customer_info SET money=500;
```

2. Deduct \$100 from the customer's account.

```
BEGIN TRANSACTION;  
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM  
customer_info WHERE name = 'buyer');
```

3. An error is reported when the amount of money is added to the store's account. The execution of the following statement is terminated.

```
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM  
customer_info WHERE name = 'shop');
```

4. Roll back the transaction. All the completed database operations related to the transaction are canceled.

```
ERROR: syntax error at or near "shop"  
LINE 1: ...e IN (SELECT name FROM customer_info WHERE name = ''shop'');  
END TRANSACTION;  
ROLLBACK
```

5. Query the account balances. The query result shows that the account balances remain unchanged. If an error occurs during transaction execution, the database is rolled back to the state before the transaction starts. The integrity of the database is not damaged.

```
SELECT * FROM customer_info;
```

name	money
buyer	500
shop	500
(2 rows)	

Two-Phase Transaction

DWS uses the distributed shared nothing architecture. Table data is distributed on different nodes. One or more statements on the client may modify data on multiple nodes at the same time. In this case, a distributed transaction is generated. DWS uses two-phase commit transactions to ensure data consistency and atomicity in distributed transactions. Two-phase commit divides transaction commit into two phases, usually for transactions that contain write operations. When data is written to different nodes, the atomicity requirement of the transaction must be met, that is, either all data is committed or all data is rolled back.

Two-phase commit is not supported in the following scenarios:

- Explicit two-phase commit of **PREPARE TRANSACTION** is not supported.

```
BEGIN;  
PREPARE TRANSACTION 'p1';
```

```
ERROR: Explicit prepare transaction is not supported.
```

- The file mappings of system catalogs cannot be modified in a two-phase transaction.

```
REINDEX TABLE pg_class;
```

```
ERROR: cannot PREPARE a transaction that modified relation mapping.
```

- Transaction snapshots cannot be committed and exported in cross-node transactions.

```
BEGIN;  
CREATE TABLE t1(a int);  
SELECT pg_export_snapshot();  
END;
```

```
ERROR: cannot PREPARE a transaction that has exported snapshots.
```

13 DDL Syntax

13.1 DDL Syntax Overview

Data definition language (DDL) is used to define or modify an object in a database, such as a table, index, or view.

 NOTE

DWS does not support DDL if its CN is unavailable. For example, if a CN in the cluster is faulty, creating a database or table will fail.

Defining a Database

A database is the warehouse for organizing, storing, and managing data. Defining a database includes: creating a database, altering the database attributes, and dropping the database. The following table lists the related SQL statements.

Table 13-1 SQL statements for defining a database

Function	SQL Statement
Create a database	CREATE DATABASE
Alter database attributes	ALTER DATABASE
Delete a database	DROP DATABASE

Defining a Schema

A schema is the set of a group of database objects and is used to control the access to the database objects. The following table lists the related SQL statements.

Table 13-2 SQL statements for defining a schema

Function	SQL Statement
Create a schema	CREATE SCHEMA
Alter schema attributes	ALTER SCHEMA
Delete a schema	DROP SCHEMA

Defining a Table

A table is a special data structure in a database and is used to store data objects and the relationship between data objects. The following table lists the related SQL statements.

Table 13-3 SQL statements for defining a table

Function	SQL Statement
Create a table	CREATE TABLE
Alter table attributes	ALTER TABLE
Alter a table name	RENAME TABLE
Delete a table	DROP TABLE
Delete all the data from a table	TRUNCATE

Defining a Partitioned Table

A partitioned table is a special data structure in a database and is used to store data objects and the relationship between data objects. The following table lists the related SQL statements.

Table 13-4 SQL statements for defining a partitioned table

Function	SQL Statement
Create a partitioned table	CREATE TABLE PARTITION
Editing a partition	ALTER TABLE PARTITION
Alter partitioned table attributes	ALTER TABLE PARTITION
Delete a partition	ALTER TABLE PARTITION
Delete a partitioned table	DROP TABLE

Defining an Index

An index indicates the sequence of values in one or more columns in the database table. The database index is a data structure that improves the speed of data access to specific information in a database table. The following table lists the related SQL statements.

Table 13-5 SQL statements for defining an index

Function	SQL Statement
Create an index	CREATE INDEX
Alter index attributes	ALTER INDEX
Delete an index	DROP INDEX
Rebuild an index	REINDEX

Defining a Role

A role is used to manage rights. For database security, all management and operation rights can be assigned to different roles. The following table lists the related SQL statements.

Table 13-6 SQL statements for defining a role

Function	SQL Statement
Create a role	CREATE ROLE
Alter role attributes	ALTER ROLE
Delete a role	DROP ROLE

Defining a User

A user is used to log in to a database. Different rights can be assigned to users for managing data accesses and operations of users. The following table lists the related SQL statements.

Table 13-7 SQL statements for defining a user

Function	SQL Statement
Create a user	CREATE USER
Alter user attributes	ALTER USER
Delete a user	DROP USER

Defining a Redaction Policy

Data redaction is to protect sensitive data by masking or changing data. You can create a data redaction policy for a specific table object and specify the effective scope of the policy. You can also add, modify, and delete redaction columns. The following table lists the related SQL statements.

Table 13-8 SQL statements for managing redaction policies

Function	SQL Statement
Create a data redaction policy	CREATE REDACTION POLICY
Modify a data redaction policy applied to a specified table	ALTER REDACTION POLICY
Delete a data redaction policy applied to a specified table	DROP REDACTION POLICY

Defining Row-Level Access Control

Row-level access control policies control the visibility of rows in database tables. In this way, the same SQL query may return different results for different users. The following table lists the related SQL statements.

Table 13-9 SQL statements for row-level access control

Function	SQL Statement
Create a row-level access control policy	CREATE ROW LEVEL SECURITY POLICY
Modify an existing row-level access control policy	ALTER ROW LEVEL SECURITY POLICY
Delete a row-level access control policy from a table	DROP ROW LEVEL SECURITY POLICY

Defining a Stored Procedure

A stored procedure is a set of SQL statements for achieving specific functions and is stored in the database after compiling. Users can specify a name and provide parameters (if necessary) to execute the stored procedure. The following table lists the related SQL statements.

Table 13-10 SQL statements for defining a stored procedure

Function	SQL Statement
Create a stored procedure	CREATE PROCEDURE

Function	SQL Statement
Delete a stored procedure	DROP PROCEDURE

Defining a Function

In DWS, a function is similar to a stored procedure, which is a set of SQL statements. The function and stored procedure are used the same. The following table lists the related SQL statements.

Table 13-11 SQL statements for defining a function

Function	SQL Statement
Create a function	CREATE FUNCTION
Alter function attributes	ALTER FUNCTION
Delete a function	DROP FUNCTION

Defining a View

A view is a virtual table exported from one or several basic tables. The view is used to control data accesses for users. The following table lists the related SQL statements.

Table 13-12 SQL statements for defining a view

Function	SQL Statement
Create a view	CREATE VIEW
Delete a view	DROP VIEW

Defining a Cursor

To process SQL statements, the stored procedure process assigns a memory segment to store context association. Cursors are handles or pointers to context regions. With a cursor, the stored procedure can control alterations in context areas.

Table 13-13 SQL statements for defining a cursor

Function	SQL Statement
Create a cursor	CURSOR
Move a cursor	MOVE

Function	SQL Statement
Extract data from a cursor	FETCH
Close a cursor	CLOSE

Altering or Ending a Session

A session is a connection established between the user and the database. The following table lists the related SQL statements.

Table 13-14 SQL statements related to sessions

Function	SQL Statement
Alter a session	ALTER SESSION
End a session	ALTER SYSTEM KILL SESSION

Defining a Resource Pool

A resource pool is a system catalog used by the resource load management module to specify attributes related to resource management, such as Cgroups. The following table lists the related SQL statements.

Table 13-15 SQL statements for defining a resource pool

Function	SQL Statement
Create a resource pool	CREATE RESOURCE POOL
Change resource attributes	ALTER RESOURCE POOL
Delete a resource pool	DROP RESOURCE POOL

Defining Synonyms

A synonym is a special database object compatible with Oracle. It is used to store the mapping between a database object and another. Currently, only synonyms can be used to associate the following database objects: tables, views, functions, and stored procedures. The following table lists the related SQL statements.

Table 13-16 SQL statements for managing synonyms

Function	SQL Statement
Creating a synonym	CREATE SYNONYM
Modifying a synonym	ALTER SYNONYM

Function	SQL Statement
Deleting a synonym	DROP SYNONYM

Defining Text Search Configuration

A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching. The following table lists the related SQL statements.

Table 13-17 SQL statements for configuring text search

Function	SQL Statement
Create a text search configuration	CREATE TEXT SEARCH CONFIGURATION
Modify a text search configuration	ALTER TEXT SEARCH CONFIGURATION
Delete a text search configuration	DROP TEXT SEARCH CONFIGURATION

Defining a Full-text Retrieval Dictionary

A dictionary is used to identify and process specific words during full-text retrieval. Dictionaries are created by using predefined templates (defined in the **PG_TS_TEMPLATE** system catalog). Five types of dictionaries can be created, **Simple**, **Ispell**, **Synonym**, **Thesaurus**, and **Snowball**. Each type of dictionaries is used to handle different tasks. The following table lists the related SQL statements.

Table 13-18 SQL statements for a full-text search dictionary

Function	SQL Statement
Create a full-text retrieval dictionary	CREATE TEXT SEARCH DICTIONARY
Modify a full-text retrieval dictionary	ALTER TEXT SEARCH DICTIONARY
Delete a full-text retrieval dictionary	DROP TEXT SEARCH DICTIONARY

Defining Query Filtering Rules

You can create query filter rules to block slow SQL statements identified in advance. The following table lists the related SQL statements.

Table 13-19 SQL statements for querying filtering rules

Function	SQL Statement
Creating a Query Filtering Rule	CREATE BLOCK RULE
Modifying a Query Filtering Rule	ALTER BLOCK RULE
Deleting a Query Filtering Rule	DROP BLOCK RULE

13.2 ALTER BLOCK RULE

Function

This function modifies the attributes of a filtering rule, including the filtering rule name, bound client name, client IP address, user, and matching mode.

This syntax is supported only by clusters of 9.1.0.100 and later versions.

Precautions

- Only a user with the database owner permission or the **gs_role_block** role permission can run the **ALTER BLOCK RULE** statement. A system administrator has this permission by default.
- The **DEFAULT** syntax is used to reset an attribute, but it applies only to optional options.

Syntax

```
ALTER BLOCK RULE block_name
[ [ TO user_name@'host' ] | [ TO user_name ] | [ TO 'host' ] | [ TO DEFAULT ] ] |
[ FOR UPDATE | SELECT | INSERT | DELETE | MERGE | DEFAULT ] |
FILTER BY
{ SQL ( 'text' ) | TEMPLATE ( template_parameter = value ) }
[ WITH ( { with_parameter = value }, [ ... ] ) ];
ALTER BLOCK RULE block_name RENAME to new_block_name;
```

Parameter Description

Table 13-20 ALTER BLOCK RULE parameters

Parameter	Description	Value Range
block_name	Name of the query filtering rule whose attributes need to be modified.	A string, which must comply with the naming convention .
user_name	User to which the filtering rule applies.	A string that indicates a valid username.

Parameter	Description	Value Range
host	Client IP address to which the filtering rule applies.	A string that indicates a valid IP address.
SQL	SQL statement for querying the regular expression of the filtering rule.	A string that indicates a regular expression.
template_parameter	Template for filtering rule matching.	unique_sql_id/sql_hash . The value is a string with unique_sql_id as all digits.
with_parameter	Parameters for querying filter rule details.	• application_name: client name. • query_band • table_num: number of tables scanned by the statement. • partition_num: maximum number of partitions to be scanned by the operator. • estimate_row: estimated maximum number of table rows scanned by the operator. • resource_pool: name of the resource pool to be switched to. • max_active_num: maximum number of concurrent statements corresponding to the rule. • is_warning: whether an alarm should be generated or an error should be reported when a statement is intercepted.

Examples

Create a query rule **query_block**.

```
CREATE BLOCK RULE query_block FILTER BY SQL('update table_table set a=1');
```

Modify the attributes of the query rule **query_block**.

```
ALTER BLOCK RULE query_block TO user1@'192.168.x.x' FOR SELECT FILTER BY SQL('select * from table_name') WITH  
(application_name='gsqll',query_band='test1',table_num='2',partition_num='3',estimate_row='1000',resource_pool='rsp1',max_active_num='3',is_warning='off');
```

Rename the query rule **query_block**.

```
ALTER BLOCK RULE query_block RENAME TO query_block_new;
```

13.3 ALTER DATABASE

Function

This command is used to modify the attributes of a database, including the database name, owner, maximum number of connections, and object isolation attribute.

Precautions

- Only the owner of a database or a system administrator has the permission to run the **ALTER DATABASE** statement. Users other than system administrators may have the following permission constraints depending on the attributes to be modified:
 - To modify the database name, you must have the CREATEDB permission.
 - To modify a database owner, you must be a database owner and a member of the new owner, and have the CREATEDB permission.
 - To change the default tablespace, you must be a database owner or a system administrator, and must have the CREATE permission on the new tablespace. This statement physically migrates tables and indexes in a default tablespace to a new tablespace. Note that tables and indexes outside the default tablespace are not affected.
 - Only a database owner or a system administrator can modify GUC parameters for the database.
 - Only database owners and system administrators can modify the object isolation attribute of a database.
- You are not allowed to rename a database in use. To rename it, connect to another database.
- The compatibility mode of an existing database cannot be changed. You can only specify the compatibility mode when creating a database. For details, see [CREATE DATABASE](#).

Syntax

- Modify the maximum number of connections of the database.

```
ALTER DATABASE database_name  
[ [ WITH ] CONNECTION LIMIT connlimit ];
```

- Rename the database.

```
ALTER DATABASE database_name  
RENAME TO new_name;
```

NOTE

If the database contains OBS multi-temperature tables, the database name cannot be changed.

- Change the database owner.

```
ALTER DATABASE database_name  
OWNER TO new_owner;
```

- Change the default tablespace of the database.

```
ALTER DATABASE database_name  
SET TABLESPACE new_tablespace;
```

 NOTE

The current tablespaces cannot be changed to OBS tablespaces.

- Modify the session parameter value of the database.

```
ALTER DATABASE database_name  
SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```

- Reset the database configuration parameter.

```
ALTER DATABASE database_name RESET  
{ configuration_parameter | ALL };
```

- Modify the object isolation attribute of a database.

```
ALTER DATABASE database_name [ WITH ] { ENABLE | DISABLE } PRIVATE OBJECT;
```

 NOTE

- To modify the object isolation attribute of a database, the database must be connected. Otherwise, the modification will fail.
- For a new database, the object isolation attribute is disabled by default. After this attribute is enabled, common users can view only the objects (such as tables, functions, views, and columns) that they have the permission to access. This attribute does not take effect for administrators. After this attribute is enabled, administrators can still view all database objects.

Parameter Description

Table 13-21 ALTER DATABASE parameters

Parameter	Description	Value Range
database_name	Specifies the name of the database whose attributes are to be modified.	A string compliant with the identifier naming rules .
connlimit	Specifies the maximum number of concurrent connections that can be made to this database (excluding administrators' connections).	An integer ranging from 1 to 50 is recommended. The default value -1 indicates no restrictions.
new_name	Specifies the new name of a database.	A string compliant with the identifier naming rules .
new_owner	Specifies the new owner of a database.	A string that indicates a valid username.

Parameter	Description	Value Range
configuration_parameter value	Sets a specified database session parameter to a specified value.	The value should be a string. • DEFAULT indicates that the default settings of the system are used in a new session. • RESET indicates that the default settings of the system are used in a new session. • OFF indicates that the setting is disabled.
FROM CURRENT	Sets the value based on the database connected to the current session.	-
RESET configuration_parameter	Resets the specified database session parameter.	-
RESET ALL	Resets all database session parameters. The modified database session parameter values will take effect in the next session.	-

Examples

Create the **testdb** database and the **user1** user.

```
CREATE DATABASE testdb ENCODING 'UTF8' template = template0;  
DROP USER IF EXISTS user1;  
CREATE USER user1 PASSWORD '{Password}';
```

Set the number of connections of the **testdb** database to **10**.

```
ALTER DATABASE testdb CONNECTION LIMIT= 10;
```

Change the database name from **testdb** to **testdb_1**.

```
ALTER DATABASE testdb RENAME TO testdb_1;
```

Change the owner of the **testdb_1** database to **user1**.

```
ALTER DATABASE testdb_1 OWNER TO user1;
```

Set the tablespace of the **testdb_1** database to **PG_DEFAULT**.

```
ALTER DATABASE testdb_1 SET TABLESPACE PG_DEFAULT;
```

Disable the default index scan on the **testdb_1** database.

```
ALTER DATABASE testdb_1 SET enable_indexscan TO off;
```

Reset parameter **enable_indexscan**.

```
ALTER DATABASE testdb_1 RESET enable_indexscan;
```

Links

[CREATE DATABASE](#) and [DROP DATABASE](#)

13.4 ALTER EXCEPT RULE

Function

This syntax modifies an exception rule set. You can modify one or more specific rule thresholds in a rule set.

Precautions

None

Syntax

```
ALTER EXCEPT RULE except_rule_name
  WITH (
    | BLOCKTIME = VALUE,
    | CPUTIME = VALUE,
    | ELAPSEDTIME = VALUE,
    | CPUSKEWPERCENT = VALUE,
    | SPILLSIZE = VALUE,
    | BROADCASTSIZE = VALUE,
    | MEMSIZE = VALUE,
    | CPUAVGPERCENT = VALUE,
    | BANDWIDTH = VALUE,
    | ACTION = ['abort' | 'penalty']
  );
```

Parameter Description

Table 13-22 ALTER EXCEPT RULE parameters

Parameter	Description	Value Range
rule_name	Name of an exception rule set.	A string compliant with the identifier naming rules .
blocktime	Maximum duration of job queue blocking, in seconds.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
elapsedtime	Maximum job execution duration, in seconds.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
allcputime	Maximum CPU time used during job running, in seconds.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .

Parameter	Description	Value Range
cpuskewpercent	CPU usage skew during job execution, in percentage.	Numeric type. The value can be -1 or ranges from 1 to 100 .
cpuavgpercent	Average CPU usage during job execution, in percentage.	Numeric type. The value can be -1 or ranges from 1 to 100 .
spillsize	Maximum size of data spilled to disks during job execution, in MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
broadcastsize	Maximum broadcast size of a job, in MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
memsize	Maximum memory size used for job execution, in MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
bandwidth	Maximum network bandwidth that can be used for job execution, in MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .

Examples

Create an exception rule **except_rule1**.

```
CREATE EXCEPT RULE except_rule_1 WITH (blocktime=2000, spillsize=3000, action=abort);
```

Change the blocktime threshold of exception rule set **except_rule1** to **3000s** and the space for spilled data to **4000 MB**.

```
ALTER EXCEPT RULE except_rule_1 WITH (blocktime=3000, spillsize=4000);
```

Create a resource pool and bind it to exception rule set **except_rule_1**.

```
CREATE resource pool resource_pool_a1 WITH (except_rule='except_rule_1');
```

Unbind the exception rule set from the resource pool **resource_pool_a1**.

```
ALTER resource pool resource_pool_a1 WITH (except_rule='None');
```

Helpful Links

[CREATE EXCEPT RULE](#) and [DROP EXCEPT RULE](#)

13.5 ALTER EXTERNAL SCHEMA

Function

Modifies EXTERNAL SCHEMA. This syntax is supported only in 8.3.0 and later versions.

Syntax

Modify an external schema based on the specified name.

```
ALTER EXTERNAL SCHEMA schema_name
WITH [ SOURCE source_name ]
    [ DATABASE 'database_name' ]
    [ SERVER server_name ]
    [ CATALOG 'catalog_name' ]
    [ OPTIONS ( { option_name 'value ' } [, ...] ) ]
    [ METAADDRESS 'address']
    [ CONFIGURATION 'confpath'];
```

Parameter Description

Table 13-23 ALTER EXTERNAL SCHEMA parameters

Parameter	Description	Value Range
schema_name	Specifies the name of the external schema to be modified.	Name of an existing schema.
SOURCE	Specifies the type of the external metadata storage engine.	Currently, source_type can only be set to dli , lakeformation , or hive .
DATABASE	Specifies the database to be accessed by the external schema.	-
SERVER	You can associate an external schema with a foreign server to access external data.	An existing foreign server whose type is lf , obs , dli , or hdfs .
CATALOG	Catalog to be accessed in LakeFormation.	-
OPTIONS	Specifies parameters of foreign table data. This is supported only by 8.3.0 and later versions.	dli_project_id : project ID of DLI. You can obtain the project ID from the management console. This parameter can be set only when the server type is DLI.
METAADDRESS	Specifies the Hive Metastore communication port. This is supported only by 9.1.0 and later versions.	-

Parameter	Description	Value Range
CONFIGURATION	Specifies the path for storing Hive Metastore configuration files. This parameter is supported only by clusters of version 9.1.0 or later. If objects in the schema on the current search path are with the same name, specify the schemas different objects are in. You can run the SHOW SEARCH_PATH command to check the schemas on the current search path.	-

Examples

Modify the database and **FOREIGN SERVER** corresponding to the **EXTERNAL SCHEMA** whose name is **ex1**.

```
ALTER EXTERNAL SCHEMA ex1
  WITH DATABASE 'demo'
  SERVER my_server;
```

13.6 ALTER FOREIGN TABLE (GDS Import and Export)

Function

Modifies a GDS foreign table.

Precautions

None

Syntax

- Set the attributes of a foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name
  OPTIONS ( {[ ADD | SET | DROP ] option ['value']}[, ... ] );
```
- Set a new owner.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
  OWNER TO new_owner;
```
- Set the row-level access control switch for foreign tables (supported by cluster versions 9.1.0 and later).

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
  ENABLE | DISABLE | FORCE | NO FORCE ROW LEVEL SECURITY
```

Parameter Description

Table 13-24 ALTER FOREIGN TABLE (for GDS import and export) parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified index does not exist.	-
table_name	Specifies the name of an existing foreign table to be modified.	Name of an existing foreign table.
option	Specifies the name of the option to be modified.	See Parameter Description in CREATE FOREIGN TABLE for details.
value	Specifies the new value of option .	-
new_owner	Specifies the new owner of the foreign table.	A string that indicates a valid username.

Examples

Create a foreign table **customer_ft** to import data from GDS server 10.10.123.234 in TEXT format.

```
DROP FOREIGN TABLE IF EXISTS customer_ft;
CREATE FOREIGN TABLE customer_ft
(
  c_customer_sk      integer      ,
  c_customer_id      char(16)    ,
  c_current_demo_sk  integer      ,
  c_current_demo_sk  integer      ,
  c_current_addr_sk  integer      ,
  c_first_ship_date_sk integer      ,
  c_first_sales_date_sk integer      ,
  c_salutation       char(10)     ,
  c_first_name       char(20)     ,
  c_last_name        char(30)     ,
  c_preferred_cust_flag char(1)   ,
  c_birth_day        integer      ,
  c_birth_month      integer      ,
  c_birth_year       integer      ,
  c_birth_country    varchar(20)  ,
  c_login            char(13)     ,
  c_email_address     char(50)    ,
  c_last_review_date char(10)
)
SERVER gsmpp_server
OPTIONS
(
  location 'gsfs://10.10.123.234:5000/customer1*.dat',
  FORMAT 'TEXT' ,
  DELIMITER '|',
  encoding 'utf8',
  mode 'Normal')READ ONLY;
```

Modify the **customer_ft** attribute of the foreign table. Delete the **mode** option.

```
ALTER FOREIGN TABLE customer_ft options(drop mode);
```

Helpful Links

[CREATE FOREIGN TABLE \(for GDS Import and Export\)](#) and [DROP FOREIGN TABLE](#)

13.7 ALTER FOREIGN TABLE (for HDFS or OBS)

Function

Modifies an HDFS or OBS foreign table.

A foreign table is a virtual table within a database or big data platform that does not physically store data. Instead, it serves as a mapping to data residing in external storage systems, such as HDFS or OBS. This mapping is established using metadata, which includes the table structure, the location of the data, and its file format. It enables users to directly access or operate external data using standard SQL statements. A foreign table serves as a data proxy that logically maps data in external file systems to table structures that can be identified by databases.

Foreign tables are designed to break data silos. Foreign tables allow database users to access external system data in a familiar way through metadata mapping and export data to external systems. In this way, data can be efficiently collaborated, analyzed, and shared across systems, especially in big data scenarios, significantly reducing data movement costs and improving collaboration efficiency.

Precautions

None

Syntax

- Set the attributes of a foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name  
  OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] );
```
- Set the owner of the foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  OWNER TO new_owner;
```
- Update a foreign table column.

```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name  
  MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT NULL  
  [ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ... ] );
```
- Modify the columns of a foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  action [, ... ];
```

The **action** syntax is as follows:

```
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type  
| ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL  
| ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer  
| ALTER [ COLUMN ] column_name OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] )  
| MODIFY column_name data_type  
| MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]  
| MODIFY column_name [ CONSTRAINT constraint_name ] NULL
```

For details, see [ALTER TABLE](#).

- Add an informational constraint for a foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  ADD [ CONSTRAINT constraint_name ]  
    { PRIMARY KEY | UNIQUE } ( column_name )  
    [ NOT ENFORCED [ ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION ] |  
  ENFORCED ];
```

For details about the parameters for adding an informational constraint, see [Parameter Description](#).
- Remove a foreign table informational constraint.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  DROP CONSTRAINT constraint_name ;
```
- Set the row-level access control switch for foreign tables (supported by cluster versions 9.1.0 and later).

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  ENABLE | DISABLE | FORCE | NO FORCE ROW LEVEL SECURITY
```

Parameter Description

Table 13-25 ALTER FOREIGN TABLE (For HDFS or OBS) parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified foreign table does not exist.	-
tablename	Specifies the name of an existing foreign table to be modified.	Name of an existing foreign table.
new_owner	Specifies the new owner of the foreign table.	A string that indicates a valid username.
data_type	Specifies the new data type of an existing column.	A string.
constraint_name	Specifies the name of a constraint to be added or deleted.	A string.
column_name	Specifies an existing column name.	A string.

For how to modify other parameters in a foreign table, see [Parameter Description](#) in **ALTER TABLE**.

Examples

1. Create **HDFS_Server** and set the corresponding foreign data wrapper to **HDFS_FDW** or **DFS_FDW**.

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS (address  
'10.10.0.100:25000,10.10.0.101:25000',hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/  
hadoop',type'HDFS');
```

2. Create an HDFS foreign table that does not contain partition columns.

```
DROP FOREIGN TABLE IF EXISTS ft_region;
CREATE FOREIGN TABLE ft_region
(
  R_REGIONKEY INT4,
  R_NAME TEXT,
  R_COMMENT TEXT
)
SERVER
  hdfs_server
OPTIONS
(
  FORMAT 'orc',
  encoding 'utf8',
  FOLDERNAME '/user/hive/warehouse/gauss.db/region_orc11_64stripe/'
)
DISTRIBUTE BY
  roundrobin;
```

3. Change the type of the **r_name** column in the foreign table **ft_region** to **text**.
ALTER FOREIGN TABLE ft_region ALTER r_name TYPE TEXT;
4. Mark the **r_name** column of the foreign table **ft_region** as **not null**.
ALTER FOREIGN TABLE ft_region ALTER r_name SET NOT NULL;

Helpful Links

[CREATE FOREIGN TABLE \(SQL on OBS or Hadoop\)](#) and [DROP FOREIGN TABLE](#)

13.8 ALTER FUNCTION

Function

ALTER FUNCTION modifies the attributes of a customized function.

Precautions

- Only the function owner or a user granted the ALTER permission can run **ALTER FUNCTION**. System administrators have this permission by default.
- To change the function owner, you must be a direct or indirect member of the new owning role.
- To change the function owner, you must have CREATE permission on the function's schemas.
- To change a function's schema, you must also have the CREATE permission on the new schema.
- If a function involves operations on temporary tables, **ALTER FUNCTION** cannot be used.

Syntax

- Modify the additional parameter of the custom function.
ALTER FUNCTION function_name ([{ [argmode] [argname] argtype } [, ...]])
action [...] [RESTRICT];

The syntax of the **action** clause is as follows:

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
| {IMMUTABLE | STABLE | VOLATILE}
| {SHIPPABLE | NOT SHIPPABLE}
| {NOT FENCED | FENCED}
```

```
| {MASKED | NOT MASKED}  
| [ NOT ] LEAKPROOF  
| { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }  
| AUTHID { DEFINER | CURRENT_USER }  
| COST execution_cost  
| ROWS result_rows  
| SET configuration_parameter { { TO | = } { value | DEFAULT } } FROM CURRENT}  
| RESET {configuration_parameter | ALL}
```

- Rename a custom function. You can add the schema name before the new function name. However, you cannot change the schema name at the same time.

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    RENAME TO new_name;  
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    RENAME TO schema.new_name;
```

- Modify the owner of the custom function.

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    OWNER TO new_owner;
```

- Modify the schema of the custom function.

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    SET SCHEMA new_schema;
```

Parameter Description

Table 13-26 ALTER FUNCTION parameters

Parameter	Description	Value Range
function_name	Specifies the function name to be modified.	Name of an existing function.
argmode	Specifies the mode of an argument. It indicates whether a parameter is an input, output, or input/output parameter.	IN , OUT , or IN OUT . The default value is IN .
argname	Specifies the parameter name.	A string compliant with the identifier naming rules .
argtype	Specifies the parameter type of a function.	Valid types are listed in Data Types .
new_name	Specifies the new name of a function.	A string compliant with the identifier naming rules .
new_owner	Specifies the new owner of a function.	Valid username or role name.
new_schema	Specifies the new schema of a function.	Existing schema.

Parameter	Description	Value Range
CALLED ON NULL INPUT RETURNS NULL ON NULL INPUT STRICT	• CALLED ON NULL INPUT declares that some parameters of a function can be invoked in normal mode if the parameter values are NULL. By default, the usage is the same as specifying the parameters. • STRICT indicates that the function always returns NULL whenever any of its parameters are NULL. If this parameter is specified, the function is not called when there are NULL parameters; instead a NULL result is returned automatically. • The usage of RETURNS NULL ON NULL INPUT is the same as that of STRICT.	-
IMMUTABLE STABLE VOLATILE	• IMMUTABLE indicates that the function always returns the same result if the parameter values are the same. • STABLE indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same parameter values, but that its result varies by SQL statements. • VOLATILE indicates that the function value can change in one table scanning and no optimization is performed.	For details about IMMUTABLE, STABLE, and VOLATILE, see Three DWS Function Types Based on Behaviors .
SHIPPABLE NOT SHIPPABLE	Indicates whether the function can be pushed down to DNs for execution. • Functions of the IMMUTABLE type can always be pushed down to the DNs. • Functions of the STABLE or VOLATILE type can be pushed down to DNs only if their attribute is SHIPPABLE.	-

Parameter	Description	Value Range
MASKED NOT MASKED	Specifies whether the function performs masking when the GUC parameter enable_redactcol_computable is enabled. MASKED supports system function modification and cannot be used together with actions in other ALTER functions.	-
LEAKPROOF	Indicates that the function has no side effect and specifies that the parameter includes only the returned value. LEAKPROOF can be set only by the system administrator.	-
EXTERNAL (optional)	Used d for SQL compatibility. This function applies to all functions, including external ones.	-
SECURITY INVOKER/AUTHID CURREN_USER	Declares that the function will be executed according to the permission of the user that invokes it. By default, the usage is the same as specifying the parameters. SECURITY INVOKER and AUTHID CURREN_USER have the same functions.	-
SECURITY DEFINER/AUTHID DEFINER	Specifies that the function is to be executed with the permissions of the user that created it. By default, the usage is the same as specifying the parameters. The usage of AUTHID DEFINER is the same as that of SECURITY DEFINER .	-
COST execution_cost	A positive number giving the estimated execution cost for the function. The unit of execution_cost is cpu_operator_cost .	Positive
ROWS result_rows	Estimates the number of rows returned by the function. This is only allowed when the function is declared to return a set.	A positive number. The default value is 1000 .

Parameter	Description	Value Range
configuration_parameter	- value Sets a specified database session parameter to a specified value. DEFAULT or RESET indicates that the default settings of the system are used in a new session. OFF indicates that the setting is disabled. - Specifies the default value. - from current Uses the value of configuration_parameter of the current session.	A string

Examples

Create the **func_add_sql2** function for calculating the sum of two integers and get the results. If the input is **null**, **null** will be returned.

```
DROP FUNCTION IF EXISTS func_add_sql2;  
CREATE FUNCTION func_add_sql2(num1 integer, num2 integer) RETURN integer  
AS  
BEGIN  
  RETURN num1 + num2;  
END;  
/  
;
```

Alter the execution rule of the **func_add_sql2** function to **IMMUTABLE**. That is, the same result is returned if the parameter is not changed.

```
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) IMMUTABLE;
```

Change the name of the **func_add_sql2** function to **add_two_number**.

```
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) RENAME TO add_two_number;
```

Change the owner of the **add_two_number** function to dbadmin.

```
ALTER FUNCTION add_two_number(INTEGER, INTEGER) OWNER TO dbadmin;
```

Helpful Links

[CREATE FUNCTION](#) and [DROP FUNCTION](#)

13.9 ALTER GROUP

Function

This syntax modifies the attributes of a user group.

Precautions

ALTER GROUP is an alias for **ALTER ROLE**, and it is not a standard SQL command and not recommended. Users can use **ALTER ROLE** directly.

Syntax

- Add users to a group.

```
ALTER GROUP group_name  
  ADD USER user_name [, ... ];
```
- Remove users from a group.

```
ALTER GROUP group_name  
  DROP USER user_name [, ... ];
```
- Change the name of the group.

```
ALTER GROUP group_name  
  RENAME TO new_name;
```

Parameter Description

For details about more parameters, see [Parameter Description](#) in **ALTER ROLE**.

Table 13-27 ALTER GROUP parameters

Parameter	Description	Value Range
group_name	Specifies the name of the user group to be modified.	A string that indicates the name of an existing group.
user_name	Specifies the user to be added to or removed from the group.	Name of an existing user.
new_name	Specifies the new name of a group.	A string compliant with the identifier naming rules .

Helpful Links

[CREATE GROUP](#), [DROP GROUP](#), and [ALTER ROLE](#)

13.10 ALTER INDEX

Function

ALTER INDEX modifies the definition of an index.

Precautions

Only the owner of an index or a system administrator can run this statement.

Syntax

- Rename a table index. The new index name can be prefixed with the name of the schema where the original index is located, but the schema name cannot be changed at the same time.

```
ALTER INDEX [ IF EXISTS ] index_name  
    RENAME TO new_name;  
ALTER INDEX [ IF EXISTS ] index_name  
    RENAME TO schema.new_name;
```
- Modify the storage parameter of a table index.

```
ALTER INDEX [ IF EXISTS ] index_name  
    SET ( {storage_parameter = value} [, ... ] );
```
- Modify the status flag of an index.

```
ALTER INDEX [ IF EXISTS ] index_name  
    SET ( {invisible = value} [, ... ] );
```
- Reset the storage parameter of a table index.

```
ALTER INDEX [ IF EXISTS ] index_name  
    RESET ( storage_parameter [, ... ] );
```
- Set a table index or an index partition to be unavailable. **The syntax cannot be used for column-store tables.**

```
ALTER INDEX [ IF EXISTS ] index_name  
    [ MODIFY PARTITION index_partition_name ] UNUSABLE;
```
- Rebuild a table index or index partition.

```
ALTER INDEX index_name  
    REBUILD [ PARTITION index_partition_name ] [WITHOUT UNUSABLE];
```
- Rename an index partition. When **PG_OBJECT** record the last modification time of an index, this syntax cannot be recorded.

```
ALTER INDEX [ IF EXISTS ] index_name  
    RENAME PARTITION index_partition_name TO new_index_partition_name;
```
- Add and modify the index comment.

```
ALTER INDEX [ IF EXISTS ] index_name  
    COMMENT 'text';
```
- Delete the index comment.

```
ALTER INDEX [ IF EXISTS ] index_name  
    COMMENT '';  
ALTER INDEX [ IF EXISTS ] index_name  
    COMMENT NULL;
```

Parameter Description

Table 13-28 ALTER INDEX parameters

Parameter	Description	Value Range
IF EXISTS	If the specified index does not exist, a prompt instead of an error is returned.	-
RENAME TO	Renames an index. It has no impact on the stored data.	-

Parameter	Description	Value Range
SET ({ STORAGE_PARAMETER = value } [, ...])	Changes one or more index-method-specific storage parameters. This command does not change the index content right away. Depending on the settings, you might need to rebuild the index with REINDEX to get the desired result.	For details about the value range and usage, see Table 13-29 .
RESET ({ storage_parameter } [, ...])	Resets one or more index method-specific storage parameters of an index to default values. Similar to the SET statement, REINDEX may be used to completely update the index.	-
[MODIFY PARTITION index_partition_name] UNUSABLE	Sets the index on a table or index partition to be unavailable.	-
REBUILD [PARTITION index_partition_name]	Recreates the index on a table or index partition.	-
WITHOUT UNUSABLE	Indexes that are in an UNUSABLE state are not rebuilt when rebuilding indexes in tables or index partitions.	-
RENAME PARTITION	Renames an index partition.	-
COMMENT comment_text	Adds, modifies, or deletes index comments.	-
index_name	Specifies the index name to be modified.	Name of an existing index.
new_name	Specifies the new name for the index.	A string compliant with the identifier naming rules .
storage_parameter	Specifies the name of an index-method-specific parameter.	-

Parameter	Description	Value Range
invisible	Controls whether the optimizer generates index scan plans. The default value is OFF .	ON or OFF . • ON indicates that no index scan plan is generated. • OFF indicates that an index scan plan is generated.
value	Specifies the new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.	-
new_index_partition_name	Specifies the new name of the index partition.	A string compliant with the identifier naming rules .
index_partition_name	Specifies the name of the index partition.	A string compliant with the identifier naming rules .
comment_text	Specifies the comment of an index.	-

Table 13-29 WITH ({storage_parameter = value} [, ...]) parameter description

Storage Parameter	Description	Example
FILLFACTOR	This parameter is unavailable for GIN and PSORT indexes. It indicates the fill factor of an index and its value is a percentage between 10 to 100. Index pages are packed only to the indicated percentage and the remaining space is used for subsequent INSERT and UPDATE operations, thereby reducing page splits and improving performance. This parameter determines the percentage of space on index pages to be packed, which affects the database performance and storage space usage. You can adjust its value to reduce page splits caused by subsequent INSERT and UPDATE operations, optimizing the storage and performance of indexes. Impact of FILLFACTOR on index performance: ● Performance: When an index page is fully packed and a new record needs to be inserted, the page will be split, consuming more I/O resources and time. A lower fillfactor can reduce page splits and improve the INSERT and UPDATE performance, but will increase the storage space usage. A higher fillfactor can improve the storage space usage, but may cause more page splits and degrade the INSERT and UPDATE performance. ● Storage space: A lower fillfactor increases the free space of each data page. A higher fillfactor stores data more compactly, reducing disk space usage.	Change the fillfactor of an index to 60%. <pre>ALTER INDEX tpcds.ds_ship_mode_t1_index7 SET (FILLFACTOR = 60);</pre>

Storage Parameter	Description	Example
	How to configure FILLFACTOR: - For tables that are frequently updated, set FILLFACTOR to a small value, for example, 60 to 80, to reserve space and reduce page splits. - For tables that are rarely updated, set FILLFACTOR to a large value, for example, 90 to 100, to maximize storage space.	
INVISIBLE	Controls whether the optimizer generates index scan plans. Scenario: If you want to disable indexes temporarily during maintenance or test the impact of indexes on performance, you can use this parameter. Value range: ON indicates that no index scan plan is generated. OFF indicates that an index scan plan is generated. Default value: OFF	Not allow the optimizer to generate any index scan plan. <pre>ALTER INDEX tpcds.ds_ship_mode_t1_index9 SET (INVISIBLE = ON);</pre>
FASTUPDATE	This parameter is available only for GIN indexes. It controls whether to enable fast update for GIN indexes. Once enabled, data changes are temporarily stored in the pending list. Value range: ON or OFF Default value: ON	Enable fast update and set the maximum size of the pending list of the index to 8 MB. <pre>ALTER INDEX tpcds.ds_ship_mode_t1_index10 SET (FASTUPDATE = ON, GIN_PENDING_LIST_LIMIT = 8192);</pre>

Storage Parameter	Description	Example
GIN_PENDING_LIST_LIMIT	This parameter is supported only by GIN indexes. When fast update is enabled for GIN indexes, you can use this parameter to set the maximum size of the pending list of GIN indexes. The pending list is a data structure specific to GIN indexes and is used to temporarily store index update operations. When FASTUPDATE is enabled for a GIN index, new index items are not directly written to the main index structure, but are stored in the pending list. When conditions are met, the index items are merged into the main index in batches. Value range: 64 to <i>INT_MAX</i>. The unit is KB. Default value: determined by the value of gin_pending_list_limit (4 MB by default)	

Examples

Create a sample table named **tpcds.ship_mode_t1**.

```
DROP TABLE IF EXISTS tpcds.ship_mode_t1;  
CREATE TABLE tpcds.ship_mode_t1  
(  
    SM_SHIP_MODE_SK      INTEGER      NOT NULL,  
    SM_SHIP_MODE_ID      CHAR(16)      NOT NULL,  
    SM_TYPE              CHAR(30)      ,  
    SM_CODE              CHAR(10)      ,  
    SM_CARRIER          CHAR(20)      ,  
    SM_CONTRACT          CHAR(20)  
)  
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

Create a unique index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table.

```
DROP INDEX IF EXISTS ds_ship_mode_t1_index1;  
CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

Create an expression index on the **SM_CODE** column in the **tpcds.ship_mode_t1** table.

```
DROP INDEX IF EXISTS ds_ship_mode_t1_index2;  
CREATE INDEX ds_ship_mode_t1_index2 ON tpcds.ship_mode_t1(SUBSTR(SM_CODE,1,4));
```

Rename the **ds_ship_mode_t1_index1** index to **ds_ship_mode_t1_index5**.

```
ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO ds_ship_mode_t1_index5;
```

Set the **ds_ship_mode_t1_index2** index as unusable.

```
ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;
```

Rebuild the **ds_ship_mode_t1_index2** index.

```
ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;
```

Links

[CREATE INDEX](#), [DROP INDEX](#), and [REINDEX](#)

13.11 ALTER LARGE OBJECT

Function

This syntax modifies the definition of a large object. The only function of ALTER LARGE OBJECT is to assign a new owner.

Precautions

Only the owner of a large object can run this command. System administrators have this permission by default.

Syntax

```
ALTER LARGE OBJECT large_object_oid  
OWNER TO new_owner;
```

Parameter Description

Table 13-30 ALTER LARGE OBJECT parameters

Parameter	Description	Value Range
large_object_oid	OID of the large object to be modified	Name of an existing large object.
OWNER TO	Clause used to modify the owner of a large object	-
new_owner	Owner of a large object	Name of an existing user or role.

Examples

None.

13.12 ALTER MATERIALIZED VIEW

Function

Modifies the properties of a materialized view. This syntax is supported only by clusters of version 8.2.1.220 or later.

Precautions

To use ALTER MATERIALIZED VIEW, ensure that [enable_matview](#) is set to **on**.

Syntax

- Set whether to enable query rewrite for materialized views.

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] { materialized_view_name }  
[ ENABLE | DISABLE ] QUERY REWRITE;
```
- Specify how materialized views are refreshed. Currently, only **COMPLETE** is available, which rebuilds the entire materialized view from scratch, re-executing the defining query.

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] { materialized_view_name }  
REFRESH [ [DEFAULT] | [ RESTRICT ] | [ CASCADE FORWARD ] | [ CASCADE BACKWARD ] |  
[ CASCADE ALL ] ] [ COMPLETE ] [ ON DEMAND ] [ [ START WITH (timestampz) ] | [ EVERY  
(interval) ] ] ;
```

 - **REFRESH ON DEMAND** indicates that the data is manually refreshed as required.
 - **START WITH** specifies the first refresh time.
 - **EVERY** specifies the refresh interval. The value can be **MONTH**, **DAY**, **HOURL**, **MINUTE**, or **SECOND**.
- Change how materialized views refresh asynchronously in the background. This change applies only to materialized views that refresh this way. This syntax is supported only by clusters of version 9.1.0.200 or later. For details, see [REFRESH MATERIALIZED VIEW](#).

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] { materialized_view_name }  
REFRESH [DEFAULT] | [ RESTRICT ] | [ CASCADE FORWARD ] | [ CASCADE BACKWARD ] |  
[ CASCADE ALL ] ;
```
- Change the owner of the materialized view.

```
ALTER MATERIALIZED VIEW { materialized_view_name }  
OWNER TO new_owner;
```
- Allow for setting table properties of materialized views. This syntax is supported only by clusters of 9.1.0.200 and later versions.

```
ALTER MATERIALIZED VIEW { materialized_view_name }  
SET ( {storage_parameter = value} [, ... ] )  
| RESET ( storage_parameter [, ... ] )
```
- Rename a view.

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] materialized_view_name  
RENAME TO new_materialized_view_name;
```

Parameter Description

Table 13-31 ALTER MATERIALIZED VIEW parameters

Parameter	Description	Value Range
IF EXISTS	If the specified materialized view does not exist, a message instead of an error is returned.	-
materialized_view_name	Specifies the name of the materialized view to be modified.	Name of an existing materialized view.
ENABLE DISABLE QUERY REWRITE	Specifies whether to enable query rewrite.	Disabled by default. When ENABLE QUERY REWRITE is specified, you need to configure the GUC parameter mv_rewrite_rule to enable query rewrite of materialized views.

Parameter	Description	Value Range
REFRESH	Specifies how materialized views are refreshed. When the data in base tables changes, you need to run REFRESH MATERIALIZED VIEW to update the data in the materialized views.	- Refresh method: DEFAULT (default refresh) and RESTRICT (forcible refresh). You can specify the refresh method only for materialized views that are asynchronously refreshed in the background. - Refresh direction: When using materialized views in a cascading nested manner, the refresh direction can be specified as FORWARD, BACKWARD, or ALL. You can specify the refresh direction only for materialized views that are asynchronously refreshed in the background. For details, see REFRESH MATERIALIZED VIEW. - Currently, only COMPLETE is available, which rebuilds the entire materialized view from scratch, re-executing the defining query. - How to trigger a refresh: ON DEMAND: manual refresh on demand. START WITH (timestamp tz) EVERY (interval): scheduled refresh. START WITH specifies the first refresh time. EVERY specifies the refresh interval and its value can be MONTH, DAY, HOUR, MINUTE, or SECOND.

Parameter	Description	Value Range
storage_parameter	Allows for setting table properties of materialized views. Parameters such as mv_pck_column , bitmap_columns , enable_foreign_table_query_rewrite , excluded_inactive_tables , force_rewrite_timeout , and mv_analyze_mode can be set.	For details, see Parameter Description .
new_owner	Specifies the owner of a materialized view.	Name of an existing user.
new_materialized_view_name	Indicates the name of the new materialized view.	A string compliant with the identifier naming rules .

Examples

Enable query rewrite for a materialized view.

```
ALTER MATERIALIZED VIEW mv1 ENABLE QUERY REWRITE;  
NOTICE: REFRESH MATERIALIZED VIEW should be executed to enable query rewrite.  
ALTER MATERIALIZED VIEW
```

Modify the table properties of a materialized view.

```
ALTER MATERIALIZED VIEW mv1 SET (force_rewrite_timeout=100);  
ALTER MATERIALIZED VIEW mv1 SET (mv_pck_column='col1');  
ALTER MATERIALIZED VIEW mv1 SET (enable_foreign_table_query_rewrite = true);
```

Change the refresh time of the materialized view.

```
ALTER MATERIALIZED VIEW mv1 REFRESH START WITH('2025-01-01 15:15:15'::timestampz) EVERY  
(interval '60 s');
```

Helpful Links

[CREATE MATERIALIZED VIEW](#), [DROP MATERIALIZED VIEW](#), [REFRESH MATERIALIZED VIEW](#)

13.13 ALTER PUBLICATION

Function

ALTER PUBLICATION modifies the publication attributes.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.

- This statement can be used by the owner of a publication and the system administrator only.
- To alter the publication owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the database.
- In a publication with **FOR ALL TABLES**, the new publication owner must be the system administrator.
- An administrator can change the owner relationship of any publication.

Syntax

- Add objects to a publication.
`ALTER PUBLICATION name ADD publication_object [, ...]`
- Delete objects from a publication.
`ALTER PUBLICATION name DROP publication_object [, ...]`
- Replace the current object with a specified object.
`ALTER PUBLICATION name SET publication_object [, ...]`
- Set publication parameters. Retain the original values of the parameters that are not mentioned.
`ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )`
- Change the publication owner.
`ALTER PUBLICATION name OWNER TO new_owner`
- Rename the publication.
`ALTER PUBLICATION name RENAME TO new_name`

The syntax of using **publication_object** is as follows:

```
TABLE table_name [, ...]  
| ALL TABLES IN SCHEMA schema_name [, ... ]
```

Parameter Description

Table 13-32 ALTER PUBLICATION parameters

Parameter	Description	Value Range
name	Specifies the publication name to be modified.	-
table_name	Specifies the table name.	Name of an existing table.
schema_name	Specifies the schema name.	Name of an existing schema.
SET (publication_ parameter [= value] [, ...])	Modifies the publication parameters initially set by CREATE PUBLICATION .	For details about the parameters, see the Parameter Description in CREATE PUBLICATION .
new_owner	Specifies the new owner of the publication.	Name of an existing user.
new_name	Specifies the new name of the publication.	A string compliant with the identifier naming rules .

Examples

- Add a table to a publication.
`ALTER PUBLICATION mypublication ADD TABLE mydata2;`
- Deletes a schema from a publication.
`ALTER PUBLICATION mypublication DROP ALL TABLES IN SCHEMA myschema1;`
- Reset a publication object.
`ALTER PUBLICATION mypublication SET TABLE mydata2, ALL TABLES IN SCHEMA myschema2;`
- Change the publication owner.
`ALTER PUBLICATION mypublication OWNER TO user1;`
- Change the publication name.
`ALTER PUBLICATION mypublication RENAME TO mypublication1;`

Helpful Links

[CREATE PUBLICATION](#) and [DROP PUBLICATION](#)

13.14 ALTER REDACTION POLICY

Function

ALTER REDACTION POLICY modifies a data redaction policy applied to a specified table.

Precautions

Only the table object owner and users with the **gs_redaction_policy** preset role can modify the masking policy.

Syntax

- Modify the expression used for a redaction policy to take effect.
`ALTER REDACTION POLICY policy_name ON table_name WHEN (when_expression);`
- Enable or disable a redaction policy.
`ALTER REDACTION POLICY policy_name ON table_name ENABLE | DISABLE;`
- Rename a redaction policy.
`ALTER REDACTION POLICY policy_name ON table_name RENAME TO new_policy_name;`
- Add, modify, or delete a column on which the redaction policy is used.
`ALTER REDACTION POLICY policy_name ON table_name
action;`

There are several clauses of **action**:

```
[INHERIT] ADD COLUMN column_name WITH redaction_function_name ( [ argument [, ...] ] )  
| [INHERIT] MODIFY COLUMN column_name WITH redaction_function_name ( [ argument [, ...] ] )  
| DROP COLUMN column_name
```

Parameter Description

Table 13-33 ALTER REDACTION POLICY parameters

Parameter	Description	Value Range
policy_name	Specifies the name of the redaction policy to be modified.	Name of an existing masking policy.
table_name	Specifies the name of the table to which the redaction policy is applied.	Name of an existing table.
when_expression	Specifies the new expression used for the redaction policy to take effect.	-
ENABLE DISABLE	Specifies whether to enable or disable the current redaction policy. • ENABLE: makes the data masking policy of the table take effect again. • DISABLE: disables the masking policy applied to the table.	-
new_policy_name	Specifies the new name of the redaction policy.	A string compliant with the identifier naming rules .
column_name	Specifies the name of the table column to which the redaction policy is applied. • To add a column, use a column name that has not been bound to any redaction functions. • To modify a column, use the name of an existing column. • To delete a column, use the name of an existing column.	-
redaction_function_name	Specifies the name of a redaction function.	For details about the supported functions, see Data Redaction Functions .

Parameter	Description	Value Range
arguments	Specifies the list of arguments of the masking function. • MASK_NONE: indicates that no masking is performed. • MASK_FULL: indicates that all data is masked to a fixed value. • MASK_PARTIAL: indicates that data of the specified character type, numeric type, or time type is partially masked.	For details about the supported functions, see Data Redaction Functions.

Examples

Create a user named **test_role** and an example table named **emp**, and insert data into the table.

```
CREATE ROLE test_role PASSWORD '{Password}';
```

```
DROP TABLE IF EXISTS emp;  
CREATE TABLE emp(id int, name varchar(20), salary NUMERIC(10,2));  
INSERT INTO emp VALUES(1, 'July', 1230.10), (2, 'David', 999.99);
```

Define a masking policy **mask_emp** on the **emp** table that hides the **salary** column from the user **test_role**.

```
CREATE REDACTION POLICY mask_emp ON emp WHEN(current_user = 'test_role') ADD COLUMN salary  
WITH mask_full(salary);
```

Modify the expression for the data masking policy to take effect for the specified role. (If no user is specified, the policy takes effect for the current user by default.)

```
ALTER REDACTION POLICY mask_emp ON emp WHEN (pg_has_role(current_user, 'redact_role', 'member'));  
ALTER REDACTION POLICY mask_emp ON emp WHEN (pg_has_role('redact_role', 'member'));
```

Modify the expression for the masking policy to take effect for all users.

```
ALTER REDACTION POLICY mask_emp ON emp WHEN (1=1);
```

Disable the masking policy.

```
ALTER REDACTION POLICY mask_emp ON emp DISABLE;
```

Enable the masking policy again.

```
ALTER REDACTION POLICY mask_emp ON emp ENABLE;
```

Change the masking policy name to **mask_emp_new**.

```
ALTER REDACTION POLICY mask_emp ON emp RENAME TO mask_emp_new;
```

Add a column with the masking policy used.

```
ALTER REDACTION POLICY mask_emp_new ON emp ADD COLUMN name WITH mask_partial(name, '*', 1,  
length(name));
```

Modify the masking policy for the **name** column. Use the **MASK_FULL** function to redact all data in the **name** column.

```
ALTER REDACTION POLICY mask_emp_new ON emp MODIFY COLUMN name WITH mask_full(name);
```

Delete an existing column where the masking policy is used.

```
ALTER REDACTION POLICY mask_emp_new ON emp DROP COLUMN name;
```

Helpful Links

[CREATE REDACTION POLICY](#) and [DROP REDACTION POLICY](#)

13.15 ALTER RESOURCE POOL

Function

ALTER RESOURCE POOL changes the Cgroup of a resource pool.

Precautions

Users having the ALTER permission can modify resource pools.

Syntax

```
ALTER RESOURCE POOL pool_name  
  WITH ({MEM_PERCENT=pct | CONTROL_GROUP='group_name' | ACTIVE_STATEMENTS=stmt | MAX_DOP  
= dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='priority' |  
nodegroup='nodegroup_name' | except_rule='except_rule' | weight=bandwidth_weight |  
enable_concurrency_scaling=boolean}[ , ... ]);
```

Parameter Description

Table 13-34 ALTER RESOURCE POOL parameters

Parameter	Description	Value Range
pool_name	Specifies the name of the resource pool to be modified.	Name of an existing resource pool
group_name	Specifies the name of a Cgroup.	Name of an existing Cgroup
stmt	Specifies the maximum number of statements that can be concurrently executed in a resource pool.	Numeral type. The value ranges from -1 to INT_MAX.
dop	Specifies the maximum number of simple SQL statements that can be concurrently executed in a resource pool.	Numeral type. The value ranges from -1 to INT_MAX.

Parameter	Description	Value Range
memory_size	Specifies the estimated maximum memory for a resource pool.	Value range: a string • default: The memory limit is half of the resource pool memory. • unlimited: no limit. • Value range: 1 KB to 2047 GB.
mem_percent	Specifies the proportion of available resource pool memory to the total memory or group user memory.	The value of mem_percent for a common user is an integer ranging from 0 to 100. The default value is 0 .
io_limits	This parameter has been discarded in the 8.1.2 cluster version and is reserved for compatibility with earlier versions.	-
io_priority	This parameter has been discarded in the 8.1.2 cluster version and is reserved for compatibility with earlier versions.	-
except_rule	Specifies the exception rule set.	-
weight	Specifies the network bandwidth weight of a resource pool.	-
enable_concurrency_scaling	Specifies whether to enable the elastic concurrency expansion function. This function is supported only by clusters of version 9.1.0.100 or later. The default value is false .	true or false • true indicates that the elastic concurrent expansion function of jobs in the resource pool is enabled. • false indicates that the elastic concurrent expansion function of jobs in the resource pool is disabled.

 NOTE

- You can use double quotation marks (") or single quotation marks (') to set the Cgroup name **group_name**.
- The value of **group_name** is case-sensitive.
- When **group_name** is not specified, the default value "Medium" is used. It is the "Medium" Timeshare Cgroup of the DefaultClass Cgroup.
- If a database user specifies the Timeshare string (**Rush**, **High**, **Medium**, or **Low**) in the syntax, for example, if **control_group** is set to **High**, the resource pool will be associated with the **High** Timeshare Cgroup under **DefaultClass**.

Examples

Create an example resource pool **pool_test**, whose Cgroup is **Medium Timeshare Workload** under **DefaultClass**.

```
DROP RESOURCE POOL IF EXISTS pool_test;  
CREATE RESOURCE POOL pool_test;
```

Specify **High Timeshare Workload** under **DefaultClass** as the Cgroup for the resource pool **pool_test**.

```
ALTER RESOURCE POOL pool_test WITH (CONTROL_GROUP="High");
```

Disable the elastic concurrent expansion function for jobs in a specified resource pool.

```
ALTER RESOURCE POOL pool_test WITH (enable_concurrency_scaling=false);
```

Helpful Links

[CREATE RESOURCE POOL](#) and [DROP RESOURCE POOL](#)

13.16 ALTER ROLE

Function

ALTER ROLE changes the attributes of a role.

Precautions

None

Syntax

- Modifying the Rights of a Role

```
ALTER ROLE role_name [ [ WITH ] option [ ... ] ];
```

The **option** clause for granting rights is as follows:

```
{CREATEDB | NOCREATEDB}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {AUDITADMIN | NOAUDITADMIN}  
| {SYSADMIN | NOSYSADMIN}  
| {USEFT | NOUSEFT}  
| {LOGIN | NOLOGIN}
```

```
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCADMIN}  
| CONNECTION LIMIT connlimit  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'  
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY 'password' [ REPLACE 'old_password' ]  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }  
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |  
DISABLE }  
| VALID BEGIN 'timestamp'  
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| TEMP SPACE 'tmpspacelimit'  
| SPILL SPACE 'spillspacelimit'  
| NODE GROUP logic_cluster_name  
| ACCOUNT { LOCK | UNLOCK }  
| PGUSER  
| AUTHINFO 'authinfo'  
| PASSWORD EXPIRATION period
```

- Rename a role.

```
ALTER ROLE role_name  
  RENAME TO new_name;
```

- Set parameters for a role.

```
ALTER ROLE role_name [ IN DATABASE database_name ]  
  SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT};
```

- Reset parameters for a role.

```
ALTER ROLE role_name  
  [ IN DATABASE database_name ] RESET {configuration_parameter|ALL};
```

Parameter Description

Table 13-35 ALTER ROLE parameters

Parameter	Description	Value Range
role_name	Specifies the name of the role to be modified.	Name of an existing role.
new_name	Indicates the new name of a role.	A string compliant with the identifier naming rules .
CREATEDB NOCREATEDB	Defines a role's ability to create databases. If not specified, NOCREATEDB is the default value. A new role does not have the permission to create databases.	-
CREATEROLE NOCREATEROLE	Determines whether a role will be permitted to create new roles (that is, execute CREATE ROLE and CREATE USER). If not specified, NOCREATEROLE is the default value. A role with the CREATEROLE permission can also modify and delete other roles.	-

Parameter	Description	Value Range
INHERIT NOINHERIT	Determines whether the role can inherit permissions of its group. You are not advised to execute them.	-
AUDITADMIN NOAUDITADMIN	Determines whether a role has the audit and management attributes. If not specified, NOAUDITADMIN is the default.	-
SYSADMIN NOSYSADMIN	Determines whether a new role is a system administrator. Roles having the SYSADMIN attribute have the highest permission. If not specified, NOSYSADMIN is the default value.	-
USEFT NOUSEFT	Determines whether a new role can perform operations on foreign tables, such as creating, deleting, modifying, and reading/writing foreign tables. A new role does not have permissions for these operations. Default value NOUSEFT indicates that a new role cannot perform operations on foreign tables.	-
LOGIN NOLOGIN	Indicates whether a user can log in to the database. Only roles with the LOGIN attribute can log in to the database. A role having the LOGIN attribute can be thought of as a user. Value range: If not specified, NOLOGIN is the default.	-
REPLICATION NOREPLICATION	Determines whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the REPLICATION attribute is a highly privileged role, and should only be used on roles used for replication. If not specified, NOREPLICATION is the default.	-

Parameter	Description	Value Range
INDEPENDENT NOINDEPENDENT	Defines private and independent roles. For a role with the INDEPENDENT attribute, administrators' rights to control and access this role are separated. Specific rules are as follows: • Administrators have no rights to add, delete, query, modify, copy, or authorize the corresponding table objects without the authorization from the INDEPENDENT role. • Administrators have no rights to modify the inheritance relationship of the INDEPENDENT role without the authorization from this role. • Administrators have no rights to modify the owner of the table objects for the INDEPENDENT role. • Administrators have no rights to delete the INDEPENDENT attribute of the INDEPENDENT role. • Administrators have no rights to change the database password of the INDEPENDENT role. The INDEPENDENT role must manage its own password, which cannot be reset if lost. • The SYSADMIN attribute of a user cannot be changed to the INDEPENDENT attribute.	-
VCADMIN NOVCADMIN	Defines the role of a logical cluster administrator. A logical cluster administrator has the following more permissions than common users: • Create, modify, and delete resource pools in the associated logical cluster. • Grant the access permission for the associated logical cluster to other users or roles, or reclaim the access permission from those users or roles.	-

Parameter	Description	Value Range
CONNECTION LIMIT	Indicates the number of concurrent connections that a role can use on a single CN. To ensure the proper running of a cluster, the minimum value of CONNECTION LIMIT is the number of CNs in the cluster, because when a cluster runs ANALYZE on a CN, other CNs will connect to the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set CONNECTION LIMIT to 3 or a larger value.	Integers no less than -1. The default value -1 means no limit.

Parameter	Description	Value Range
ENCRYPTED UNENCRYPTED	Determines whether the password stored in the system will be encrypted. (If neither is specified, the password status is determined by password_encryption_type.) According to product security requirements, the password must be stored encrypted. Therefore, UNENCRYPTED is forbidden in DWS. If the password is encrypted using SHA256, it will be stored as it is, regardless of whether it is specified as ENCRYPTED or UNENCRYPTED. This is because the system cannot decrypt the specified encrypted password. This allows reloading of the encrypted password during dump/restore. - password Specifies the login password. - DISABLE By default, you can change your password unless it is disabled. Use this parameter to disable the password of a user. After the password of a user is disabled, the password will be deleted from the system. The user can connect to the database only through external authentication, for example, IAM authentication, Kerberos authentication, or LDAP authentication. Only administrators can enable or disable a password. Common users cannot disable the password of an initial user. To enable a password, run ALTER USER and specify the password.	The password must: - Contain at least eight characters. - Differ from the username or it in reverse order. - Contain at least three types of the following characters: uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), and special characters such as ~!@#\$%^&*()-_+=+ [{ }];;<.>/? (If you use a character that is not in the specified range, an alarm is generated, but the character can still be created.)
VALID BEGIN	Sets a date and time when the role's password becomes valid. If this clause is omitted, the password will be valid for all time.	-
VALID UNTIL	Sets a date and time after which the role's password is no longer valid. If this clause is omitted, the password will be valid for all time.	-

Parameter	Description	Value Range
RESOURCE POOL	Sets the name of resource pool used by a role, and the name belongs to the system catalog: pg_resource_pool .	-
USER GROUP 'groupuser'	Creates a sub-user.	-
PERM SPACE	Sets the storage space of the user permanent table. space_limit specifies the upper limit of the storage space of the permanent table.	space_limit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
TEMP SPACE	Sets the storage space of the user temporary table. tmpspacelimit specifies the storage space limit of the temporary table.	tmpspacelimit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
SPILL SPACE	Sets the operator disk flushing space of the user. spillspacelimit specifies the operator spilling space limit.	spillspacelimit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
NODE GROUP	Specifies the name of the logical cluster associated with a user. If the name contains uppercase characters or special characters, enclose the name with double quotation marks.	-
ACCOUNT LOCK ACCOUNT UNLOCK	● ACCOUNT LOCK: locks an account to forbid login to databases. ● ACCOUNT UNLOCK: unlocks an account to allow login to databases.	-

Parameter	Description	Value Range
PGUSER	This attribute is used to be compatible with open-source Postgres communication. An open-source Postgres client interface (Postgres 9.2.19 is recommended) can use a database user having this attribute to connect to the database. PGUSER of a role cannot be modified in the current version. For details, see PGUSER Parameter Setting Suggestions.	-
AUTHINFO 'authinfo'	Specifies the role authentication type. authinfo is the description character string, which is case sensitive. Only the LDAP type is supported. Its description character string is ldap. LDAP authentication is an external authentication mode. Therefore, PASSWORD DISABLE must be specified. • Additional information about LDAP authentication can be added to authinfo, for example, fulluser in LDAP authentication, which is equivalent to ldapprefix+username+ldapsuffix. If the content of authinfo is ldap, the role authentication type is LDAP. In this case, the ldapprefix and ldapsuffix information is provided by the corresponding record in the pg_hba.conf file. • When executing the ALTER ROLE command, users are not allowed to change the authentication type. Only LDAP users are allowed to modify LDAP attributes.	-
PASSWORD EXPIRATION period	Specifies the number of days before the login password of the role expires. You need to change the password before the password expires. If your login password expires, you cannot log in. Ask the administrator for a new password.	An integer ranging from -1 to 999. • Default value -1 indicates that there is no restriction. • 0 indicates that the login password expires immediately.

Parameter	Description	Value Range
IN DATABASE database_name	Modifies the parameters of a role in a specified database.	-
SET configuration_parameter	Sets parameters for a role. The session parameters modified using the ALTER ROLE command is only for a specific role and is valid in the next session triggered by the role. • DEFAULT clears the value of configuration_parameter. The value of the configuration_parameter parameter inherits the default value of the new session of the role. • FROM CURRENT uses the value of configuration_parameter of the current session.	Values of configuration_parameter and value are listed in SET .
RESET configuration_parameter/ALL	The effect of clearing the configuration_parameter value is the same as setting it to DEFAULT . ALL indicates that all parameter values are cleared.	-

PGUSER Parameter Setting Suggestions

The **PGUSER** attribute is for compatibility with the connection process. It does not handle incompatibilities due to kernel differences between the product and Postgres.

Users having the **PGUSER** attribute are authenticated in a way different from other users. Error information reported by the open-source client may cause the attribute to be enumerated. Therefore, you are advised to use a client of this product. Example:

```
# normaluser is a user that does not have the PGUSER attribute. psql is the Postgres client tool.
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

# pguser is a user having the PGUSER attribute.
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

Examples

Create example roles **r1**, **r2**, and **r3**.

```
CREATE ROLE r1 IDENTIFIED BY '{Password}';
CREATE ROLE r2 WITH LOGIN AUTHINFO 'ldapcn=r2,cn=user,dc=lework,dc=com' PASSWORD DISABLE;
CREATE ROLE r3 WITH LOGIN PASSWORD '{Password}' PASSWORD EXPIRATION 30;
```

Modify the login permission of role **r1**.

```
ALTER ROLE r1 login;
```

Change the password of role **r1**.

```
ALTER ROLE r1 IDENTIFIED BY '{new_Password}' REPLACE '{Password}';
```

Change the **manager** role to the system administrator.

```
ALTER ROLE r1 SYSADMIN;
```

Modify the **fulluser** information of the LDAP authentication role.

```
ALTER ROLE r2 WITH LOGIN AUTHINFO 'ldapcn=role2,cn=user2,dc=func,dc=com' PASSWORD DISABLE;
```

Change the validity period of the login password of the role to 90 days.

```
ALTER ROLE r3 PASSWORD EXPIRATION 90;
```

Links

[CREATE ROLE](#), [DROP ROLE](#), and [SET](#).

13.17 ALTER SUBSCRIPTION

Function

This syntax modifies the attributes of a subscription.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.
- Only the owner of a subscription can execute **ALTER SUBSCRIPTION**, and the new owner must be a system administrator.

Syntax

- Update the connection information of a subscription.

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo';
```
- Update the name of the publication on the publisher side.

```
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...];
```
- Enable a subscription.

```
ALTER SUBSCRIPTION name ENABLE;
```
- Disable a subscription.

```
ALTER SUBSCRIPTION name DISABLE;
```
- Set subscription parameters.

```
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ...] );
```
- Change the subscription owner.

```
ALTER SUBSCRIPTION name OWNER TO new_owner;
```
- Rename the subscription.

```
ALTER SUBSCRIPTION name RENAME TO new_name;
```

Parameter Description

Table 13-36 ALTER SUBSCRIPTION parameters

Parameter	Description	Value Range
name	Specifies the name of the subscription to be modified.	Name of an existing subscription.
CONNECTION 'conninfo'	Alters the connection attributes initially set by CREATE SUBSCRIPTION .	For details about the parameters, see Parameter Description in CREATE SUBSCRIPTION .
publication_name	Specifies the name of a new publication.	A string compliant with the identifier naming rules .
ENABLE	Enables a previously disabled subscription and starts logical replication at the end of a transaction.	-
DISABLE	Disables a running subscription and stops logical replication at the end of a transaction.	-
SET (publication_parameter [= value] [, ...])	Modifies the publication parameters initially set by CREATE PUBLICATION .	For details about the parameters, see the Parameter Description part in CREATE SUBSCRIPTION .
new_owner	Specifies the name new subscription owner.	Name of an existing user.
new_name	Specifies the new name of the subscription.	A string compliant with the identifier naming rules .

Examples

Modify subscription connection information.

```
ALTER SUBSCRIPTION mysub CONNECTION 'host=192.168.1.51 port=5432 user=foo dbname=foodb  
password=xxxx';
```

Helpful Links

[CREATE SUBSCRIPTION](#) and [DROP SUBSCRIPTION](#)

13.18 ALTER ROW LEVEL SECURITY POLICY

Function

ALTER ROW LEVEL SECURITY POLICY modifies an existing row-level access control policy, including the policy name and the users and expressions affected by the policy.

Precautions

Only the table owner or administrators can perform this operation.

Syntax

```
ALTER [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name RENAME TO  
new_policy_name
```

```
ALTER [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name  
[ TO { role_name | PUBLIC } [, ...] ]  
[ USING ( using_expression ) ]
```

Parameter Description

Table 13-37 ALTER ROW LEVEL SECURITY POLICY parameters

Parameter	Description	Value Range
policy_name	Specifies the name of a row-level access control policy to be modified.	A string compliant with the identifier naming rules.
table_name	Specifies the name of a table to which a row-level access control policy is applied.	Name of an existing table.
new_policy_name	Specifies the new name of a row-level access control policy.	-
role_name	Specifies names of users affected by a row-level access control policy will be applied. PUBLIC indicates that the row-level access control policy will affect all users.	Name of an existing role.
using_expression	Specifies an expression defined for a row-level access control policy. The return value is of the boolean type.	-

Examples

Enable row-level access control.

```
ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;
```

Change the name of the **all_data_rls** policy.

```
ALTER ROW LEVEL SECURITY POLICY all_data_rls ON all_data RENAME TO all_data_new_rls;
```

Change the users affected by the row-level access control policy.

```
ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data TO alice, bob;
```

Modify the expression defined for the access control policy.

```
ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data USING (id > 100 AND role =  
current_user);
```

Helpful Links

[CREATE ROW LEVEL SECURITY POLICY](#) and [DROP ROW LEVEL SECURITY POLICY](#)

13.19 ALTER SCHEMA

Function

ALTER SCHEMA changes the attributes of a schema.

Precautions

- Only the owner of a schema or users with the **ALTER** permission for the schema can run the **ALTER SCHEMA** statement. System administrators have this permission by default.
- To change the owner of a schema or its storage limit, a non-admin user must be directly or indirectly part of the new role and have **CREATE** permission on the database.

Syntax

- Rename a schema.

```
ALTER SCHEMA schema_name  
  RENAME TO new_name;
```
- Changes the owner of a schema.

```
ALTER SCHEMA schema_name  
  OWNER TO new_owner;
```
- Changes the storage space limit of the permanent table in the schema.

```
ALTER SCHEMA schema_name  
  WITH PERM SPACE 'space_limit';
```

Parameter Description

Table 13-38 ALTER SCHEMA parameters

Parameter	Description	Value Range
schema_name	Specifies the name of the schema to be modified.	Name of an existing schema.

Parameter	Description	Value Range
new_name	Specifies the new schema name.	A string compliant with the identifier naming rules .
new_owner	Specifies the new schema owner.	Name of an existing user or role.
WITH PERM SPACE 'space_limit'	The upper limit of the permanent table storage space of the schema.	A string consists of an integer and unit. The unit can be K/M/G/T/P. The parsed value is in kilobytes (K) and must stay within the 1 KB to 9,007,199,254,740,991 KB range.

Examples

Create an example schema **schema_test** and a user **user_a**.

```
CREATE SCHEMA schema_test;  
CREATE USER user_a PASSWORD '{Password}';
```

Rename **schema_test** to **schema_test1**.

```
ALTER SCHEMA schema_test RENAME TO schema_test1;
```

Change the owner of **schema_test1** to **user_a**.

```
ALTER SCHEMA schema_test1 OWNER TO user_a;
```

Helpful Links

[CREATE SCHEMA](#) and [DROP SCHEMA](#)

13.20 ALTER SEQUENCE

Function

Modifies the sequence definition.

Precautions

- You must be the owner of the sequence to use **ALTER SEQUENCE**.
- To change the owner, you must also be a direct or indirect member of the new role, and this role must have the **CREATE** permission on the sequence's schema.
- In the current version, you can modify only the owner, home column, and the maximum value. To modify other parameters, delete the sequence and create it again. Then, use the Setval function to restore original parameter values.
- **ALTER SEQUENCE MAXVALUE** cannot be used in transactions, functions, and stored procedures.

- After the maximum value of a sequence is changed, the cache of the sequence in all sessions is cleared.
- **ALTER SEQUENCE** blocks the invocation of **nextval**, **setval**, **currval**, and **lastval**.

Syntax

Change the maximum value or home column of the sequence.

```
ALTER SEQUENCE [ IF EXISTS ] name  
  [ MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE ]  
  [ CACHE cache ]  
  [ OWNED BY { table_name.column_name | NONE } ] ;
```

Change the owner of the sequence.

```
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner;
```

Parameter Description

Table 13-39 ALTER SEQUENCE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the sequence does not exist.	-
name	Specifies the name of the sequence to be modified.	Name of an existing sequence.
MAXVALUE maxvalue NO MAXVALUE	Specifies the maximum value of a sequence. If NO MAXVALUE is specified, the default value of the ascending sequence is $2^{63}-1$, and that of the descending sequence is -1 . NOMAXVALUE is equivalent to NO MAXVALUE .	-
CACHE cache	Allocates sequence numbers in advance and stores them in memory to speed up access. This parameter is supported only by clusters of version 8.2.1.100 or later.	• The minimum value of cachevalue is 1. (One value can be generated at a time, that is NOCACHE.) • If this parameter is not specified, the old cache value is retained.

Parameter	Description	Value Range
OWNED BY	Associates a sequence with a specified field in a table. In this way, the sequence will be deleted when you delete its associated field or the table where the field belongs. If the sequence has been associated with another table before you use this parameter, the new association will overwrite the old one. The associated table and sequence must be owned by the same user and in the same schema. If OWNED BY NONE is used, existing associations will be deleted.	-
new_owner	Specifies the username of the new owner.	Name of an existing user.

Examples

Create the sample sequence **seq_test** and sample table **t1**.

```
CREATE SEQUENCE seq_test START 101;  
DROP TABLE IF EXISTS t1;  
CREATE TABLE t1(c1 bigint default nextval('seq_test'));
```

Modify the maximum value of **serial** to **200**.

```
ALTER SEQUENCE seq_test MAXVALUE 200;
```

Change the home column of **seq_test** to **t1.c1**.

```
ALTER SEQUENCE seq_test OWNED BY t1.c1;
```

Helpful Links

[CREATE SEQUENCE](#) and [DROP SEQUENCE](#)

13.21 ALTER SERVER

Function

Adds, modifies, and deletes the definition of a foreign server.

Existing foreign servers can be queried from the **pg_foreign_server** system catalog.

Precautions

- Only the owner of the server has the permission to run the **ALTER SERVER** command. System administrators have this permission by default.
- To change the server owner, you must own the foreign server, belong directly or indirectly to the new owner's role, and have the **USAGE** permission on the server's foreign-data wrapper.

Syntax

- Change the parameters for an external server.

```
ALTER SERVER server_name [ VERSION 'new_version' ]  
[ OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] ) ];
```
- Change the owner of an external server.

```
ALTER SERVER server_name  
OWNER TO new_owner;
```
- Change the name of an external service.

```
ALTER SERVER server_name  
RENAME TO new_name;
```
- Refresh the HDFS configuration file.

```
ALTER SERVER server_name REFRESH OPTIONS;
```

Parameter Description

Table 13-40 ALTER SERVER parameters

Parameter	Description	Value Range
server_name	Specifies the name of the external service to be modified.	Name of an existing external server.
new_version	Specifies the new version of the external server.	-
OPTIONS	Specifies options for the server. ADD , SET , and DROP specify the operations to be performed. If the OPTIONS option is not specified, the ADD operation is performed by default. Currently, only SET is supported on an HDFS server. ADD and DROP are not supported. The syntax for SET and DROP operations is retained for later use.	option and value are corresponding operation parameters. For details, see Table 13-41 .
new_owner	Specifies the new owner of the external server.	Valid username.
new_name	Specifies the new name of the external server.	A string compliant with the identifier naming rules .

Parameter	Description	Value Range
REFRESH OPTIONS	Refreshes the HDFS configuration file cache information. This command is executed when the configuration file changes. If this command is not executed, an access error may occur. This parameter is used to update the HDFS server configuration cache and cannot be concurrently accessed by related HDFS foreign tables.	-

Table 13-41 Parameters for modifying **OPTIONS** supported by external servers

Parameter	Description	Value Range
address	Specifies the endpoint of the OBS service. Specifies the IP address and port number of the primary and standby nodes of the HDFS cluster. NOTE • address is mandatory for HDFS servers. Therefore, ADD and DROP operations are not supported. • address only supports IPv4 addresses in dot-decimal notation, and an address string cannot contain spaces. Groups of addresses are separated by commas (,). An IP address and a port number are separated by a colon (:). You are advised to configure two IP address and port pairs in an HDFS cluster. One is used as the socket address of the primary HDFS NameNode and another is used as that of the secondary HDFS NameNode. • If the server type is DLI, the address is the OBS address stored on DLI.	-
hdfscfgpath	Specifies the HDFS cluster configuration file. NOTE • If HDFS is in security mode, hdfscfgpath is mandatory. • If you set hdfscfgpath, you can only set one value for path.	-

Parameter	Description	Value Range
fed	Indicates that dfs_fdw is connected to HDFS in federation mode. The value rbf indicates that HDFS uses the Router-based Federation (RBF) mode. NOTE This parameter is supported in 8.1.2 or later. In 8.0.0, this parameter is supported only in 8.0.0.10 or later.	-
encrypt	Specifies whether data is encrypted. This parameter is available only when type is OBS . The default value is off .	• on indicates that data is encrypted. • off indicates that data is not encrypted.
access_key	Indicates the access key (AK) (obtained by users from the OBS page) used for the OBS access protocol. When you create a foreign table, its AK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is OBS .	-
secret_access_key	Indicates the secret access key (SK) (obtained by users from the OBS page) used for the OBS access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is OBS .	-
dli_address	Specifies the endpoint of the DLI service. This parameter is available only when type is DLI .	-
dli_access_key	Specifies the access key (AK) (obtained by users from the DLI console) used for the DLI access protocol. When you create a foreign table, its AK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is DLI .	-
dli_secret_access_key	Specifies the secret access key (SK) (obtained by users from the DLI console) used for the DLI access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is DLI .	-

Parameter	Description	Value Range
region	Indicates the IP address or domain name of the OBS server. This parameter is available only when type is OBS .	-
dbname	Specifies the database name of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection.	-
username	Specifies the username of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection.	-
password	Specifies the password of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection.	-
syncsrv	This parameter is used only for cross-cluster interconnection and indicates the GDS service used during data synchronization. The method for setting this parameter is the same as that for setting the location attribute of the GDS foreign table. This feature is supported only in 8.1.2 or later.	-

Examples

Create the **hdfs_server** external server, in which **hdfs_fdw** is the built-in foreign data wrapper.

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
(address '10.10.0.100:25000,10.10.0.101:25000',
 hdfsconfpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
 type 'HDFS'
);
```

Modify the **address** option of the external server **hdfs_server**.

```
ALTER SERVER hdfs_server OPTIONS ( SET address '10.10.0.110:25000,10.10.0.120:25000');
```

Modify the **hdfsconfpath** option of the external server **hdfs_server**.

```
ALTER SERVER hdfs_server OPTIONS ( SET hdfsconfpath '/opt/bigdata/hadoop');
```

Helpful Links

[CREATE SERVER](#), [DROP SERVER](#)

13.22 ALTER SESSION

Function

ALTER SESSION defines or modifies the conditions or parameters that affect the current session. Modified session parameters are kept until the current session is disconnected.

Precautions

- If the **START TRANSACTION** command is not executed before the **SET TRANSACTION** command, the transaction is ended instantly and the command does not take effect.
- You can use the transaction_mode(s) method declared in the **START TRANSACTION** command to avoid using the **SET TRANSACTION** command.

Syntax

- Set transaction parameters of a session.

```
ALTER SESSION SET [ SESSION CHARACTERISTICS AS ] TRANSACTION
  { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED } | { READ ONLY | READ
WRITE } } [, ...] ;
```
- Set other running parameters of a session.

```
ALTER SESSION SET
  {{config_parameter { { TO | = } { value | DEFAULT }
| FROM CURRENT }} | CURRENT_SCHEMA [ TO | = ] { schema | DEFAULT }
| TIME_ZONE time_zone
| SCHEMA schema
| NAMES encoding_name
| ROLE role_name PASSWORD 'password'
| SESSION AUTHORIZATION { role_name PASSWORD 'password' | DEFAULT }
| XML OPTION { DOCUMENT | CONTENT }
};
```

Parameter Description

Table 13-42 ALTER SESSION parameters

Parameter	Description	Value Range
SESSION	Indicates that the specified parameters take effect for the current session. This is the default value if neither SESSION nor LOCAL appears. If SET or SET SESSION is executed within a transaction that is later aborted, the effects of the SET command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another SET .	-
config_parameter	Indicates the configurable run-time parameters. You can use SHOW ALL to view available run-time parameters. NOTE Some parameters that viewed by SHOW ALL cannot be set by SET . For example, max_datanodes .	-
value	Indicates the new value of the config_parameter parameter.	This parameter can be specified as string constants, identifiers, numbers, or comma-separated lists of these. DEFAULT can be written to indicate resetting the parameter to its default value.
TIME_ZONE timezone	Indicates the local time zone for the current session.	A valid local time zone. The corresponding run-time parameter is TimeZone . The default value is PRC .
CURRENT_SCHEMA A	Indicates the current schema.	Name of an existing schema.
SCHEMA schema	Indicates the current schema. Here the schema is a string. Example: set schema 'public';	-

Parameter	Description	Value Range
NAMES encoding_name	Indicates the client character encoding name. This command is equivalent to set client_encoding to encoding_name .	A valid character encoding name. The run-time parameter corresponding to this option is client_encoding . The default encoding is UTF8 .
XML OPTION option	Indicates the XML resolution mode.	• CONTENT (default) • DOCUMENT

Examples

Create the **ds** schema.

```
CREATE SCHEMA ds;
```

Set the search path of the schema.

```
SET SEARCH_PATH TO ds, public;
```

Set the time/date type to the traditional postgres format (date before month).

```
SET DATESTYLE TO postgres, dmy;
```

Set the character code of the current session to UTF8.

```
ALTER SESSION SET NAMES 'UTF8';
```

Set the time zone to Berkeley of California.

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone to Italy.

```
SET TIME ZONE 'Europe/Rome';
```

Set the current schema.

```
ALTER SESSION SET CURRENT_SCHEMA TO tpcds;
```

Set **XML OPTION** to **DOCUMENT**.

```
ALTER SESSION SET XML OPTION DOCUMENT;
```

Create the role **joe**, and set the session role to **joe**.

```
CREATE ROLE joe WITH PASSWORD '{password}';  
ALTER SESSION SET SESSION AUTHORIZATION joe PASSWORD '{password}';
```

Switch to the default user.

```
ALTER SESSION SET SESSION AUTHORIZATION default;
```

Helpful Links

[SET](#)

13.23 ALTER SYNONYM

Function

ALTER SYNONYM is used to modify the attribute of a synonym.

Precautions

- Only the owner of the **SYNONYM** object can be changed.
- Only the system administrator and the owner of the **SYNONYM** object can change its owner information.
- The modifier must be a direct or indirect member of the new owner, and the new owner must have the **CREATE** permission on the schema to which the synonym belongs.

Syntax

```
ALTER SYNONYM synonym_name  
OWNER TO new_owner;
```

Parameter Description

Table 13-43 ALTER SYNONYM parameters

Parameter	Description	Value Range
synonym_name	Name of the synonym to be modified, which can include the schema name.	Name of an existing synonym.
OWNER TO	Clause used to modify the owner of a synonym.	-
new_owner	New owner of a synonym.	Valid username.

Examples

Create synonym **t1**.

```
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;
```

Create user **u1**.

```
CREATE USER u1 PASSWORD '{password}';
```

Change the owner of the synonym **t1** to **u1**.

```
ALTER SYNONYM t1 OWNER TO u1;
```

Helpful Links

[CREATE SYNONYM](#) and [DROP SYNONYM](#)

13.24 ALTER SYSTEM KILL SESSION

Function

ALTER SYSTEM KILL SESSION ends a session.

Precautions

None

Syntax

```
ALTER SYSTEM KILL SESSION 'session_sid, serial' [ IMMEDIATE ];
```

Parameter Description

Table 13-44 ALTER SYSTEM KILL SESSION parameters

Parameter	Description	Value Range
session_sid, serial	Specifies the SID and SERIAL of a session.	The SIDs and SERIALs of all sessions that can be queried from the system catalog V\$SESSION .
IMMEDIATE	Indicates that a session will be ended instantly after the command is executed.	-

Examples

Query session information.

```
SELECT sid,serial#,username FROM V$SESSION;
```

```
  sid      | serial# | username
-----+-----+-----
140131075880720 | 0 |
140131025549072 | 0 |
140131073779472 | 0 |
140131071678224 | 0 |
140131125774096 | 0 |
140131127875344 | 0 |
140131113629456 | 0 |
140131094742800 | 0 |
(8 rows)
```

End the session whose SID is 140131075880720.

```
ALTER SYSTEM KILL SESSION '140131075880720,0' IMMEDIATE;
```

13.25 ALTER TABLE

Function

ALTER TABLE is used to modify tables, including modifying table definitions, renaming tables, renaming specified columns in tables, renaming table constraints, setting table schemas, enabling or disabling row-level access control, and adding or updating multiple columns.

Only the owner of a table, a user granted with the ALTER permission for the table, or a system administrator has the permission to run the **ALTER TABLE** statement. To change the owner or schema of a table, you must be the table owner or a system administrator and be a direct or indirect member of the new role.

Precautions

Table 13-45 Precautions for ALTER TABLE

Constraint	Description
Storage method of tables	After a table is created, the parameter ORIENTATION (which specifies whether the table is a row-store or column-store table) cannot be modified.
Supported range of column-store tables	Column-store tables support PARTIAL CLUSTER KEY but do not support table-level foreign key constraints. In 8.1.1 or later, you can use the PRIMARY KEY constraint and table-level UNIQUE constraint during the column-store table creation.
Modification of column-store tables	• The NOT NULL constraint and PRIMARY KEY constraint can be added to existing column-store tables using ALTER. This constraint is supported only by clusters of version 8.2.0 or later. • If you delete the PRIMARY KEY constraint by specifying the constraint name, the NOT NULL constraint is not deleted. You can manually delete the NOT NULL constraint as needed.
Columns of column-store tables	The column constraints supported by column-store tables include NULL , NOT NULL , and DEFAULT constant values. Only the DEFAULT value can be modified and deleted (SET DEFAULT and DROP DEFAULT), and only the NOT NULL constraint can be deleted.

Constraint	Description
Operations supported by column-store tables	• ADD COLUMN is used to add a column. The types of new and modified columns should be the data types supported by column storage (for details, see Data Types). • ALTER TYPE is used to change the data type of a column. The USING option of ALTER TYPE supports only constant expressions and expressions involving the column. • SET STATISTICS is used to set the statistics collection target of a single column. • Table name can be modification. • DROP COLUMN is used to delete a column.
Modification of distribution columns	The data type of the distribution column in an existing table cannot be modified.
Row-store REPLICATION tables	A system column cannot be designated as a primary key in a row-store REPLICATION distributed table.
Auto-increment columns	Auto-increment columns cannot be added, or a column in which the DEFAULT value contains the nextval() expression cannot be added either.
Modification of OBS cold and hot tables	• The cold_tablespace and storage_policy parameters of ALTER RESET cannot be used in OBS hot or cold tables, and COLVERSION cannot be changed to 1.0 for such tables. • You can change a column-store table whose COLVERSION is 2.0 to an OBS hot or cold table. The COLD_TABLESPACE and STORAGE_POLICY parameters must be added. • You can use ALTER TABLE to change the values of STORAGE_POLICY for REOPTIONS. After the cold/hot switchover policy is changed, the cold/hot attribute of the existing cold data will not change. The new policy takes effect for the next cold/hot switchover.
V3 tables	COLVERSION cannot be modified for V3 tables (COLVERSION is 3.0), and non-V3 tables (COLVERSION is 2.0) cannot be switched to V3 tables.

Constraint	Description
Operations that trigger table rebuilding	When an ALTER TABLE operation is performed on a table, it triggers table rebuilding. During this process, data is dumped into a new data file. Once the process is complete, the original file is deleted. If the table is large, the process can use a significant amount of disk space. When the disk space is insufficient, exercise caution when performing the ALTER TABLE operation on large tables to prevent the cluster from being read-only. • Change the data type of a column. • Add columns (including the oid column) to a row-store table. • Modify COLVERSION for a column-store table. • Specify the DEFAULT constant values for a column added to a column-store table. If the DEFAULT values contain volatile functions or are not NULL and do not belong to a specific data type, ensure they are correctly defined.
Other	• When you modify the COLVERSION or enable_delta parameter of a column-store table, other ALTER operations cannot be performed. • You cannot specify a tablespace when running ALTER TABLE for an unlogged table. When you run ALTER TABLE for a non-unlogged table, do not specify pg_unlogged as the tablespace. • Distribution columns cannot be modified for global temporary tables.

 WARNING

- Avoid performing **ALTER TABLE**, **ALTER TABLE PARTITION**, **DROP PARTITION**, and **TRUNCATE** operations during peak hours to prevent long SQL statements from blocking these operations or SQL services.
- For more information about development and design specifications, see [Development and Design Proposal](#).

ALTER TABLE Syntax Format

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
action [ , ... ];
```

action can be the following modification operations:

- Change the data type of an existing column in the table. You can modify the null constraint, default value, and comment information at the same time. Only the type conversion of the same category (between values, strings, and time) is allowed.

```
ALTER TABLE [ IF EXISTS ] table_name  
MODIFY ( { column_name data_type | [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ] |  
[ CONSTRAINT constraint_name ] NULL | DEFAULT default_expr | COMMENT 'text' } [, ...] );
```

- Rename the table. Changing the table name does not affect the stored data. The new table name can be prefixed with the schema name of the original table, but the schema name cannot be changed at the same time.

```
ALTER TABLE [ IF EXISTS ] table_name  
RENAME TO new_table_name;  
ALTER TABLE [ IF EXISTS ] table_name  
RENAME TO schema.new_table_name;
```

- Rename the specified column in the table.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
RENAME [ COLUMN ] column_name TO new_column_name;
```

- Rename the constraint of the table.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
RENAME CONSTRAINT constraint_name TO new_constraint_name;
```

- Set the schema of the table.

```
ALTER TABLE [ IF EXISTS ] table_name  
SET SCHEMA new_schema;
```

NOTE

- Currently, **SET SCHEMA** can only set schemas to user schemas. It cannot set a schema to a system internal schema.
- The schema setting moves the table into another schema. Associated indexes and constraints owned by table columns are migrated as well. Currently, the schema for sequences cannot be changed. If the table has sequences, delete the sequences, and create them again or delete the ownership between the table and sequences. In this way, the table schema can be changed.
- To change the schema of a table, you must also have CREATE privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE permission on the table's schema. These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the table. However, a system administrator can alter ownership of any table anyway.
- Add columns.

```
ALTER TABLE [ IF EXISTS ] table_name  
ADD ( { column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint  
[ ... ] } [, ...] );
```
- Update columns.

```
ALTER TABLE [ IF EXISTS ] table_name  
MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT NULL  
[ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ...] );
```
- Delete an index from a specified table.

```
ALTER TABLE table_name DROP INDEX index_name;
```

NOTE

- All the actions except for **RENAME** and **SET SCHEMA** can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several columns or alter the type of several columns in a single command. This is useful with large tables, since only one pass over the table need be made.
- Adding a **CHECK** or **NOT NULL** constraint requires scanning the table to verify that existing rows meet the constraint.
- Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. Table rebuilding may take a significant amount of time for a large table; and will temporarily require as much as double the disk space.

Clauses of action Related to Table Operations

There are several clauses of **action**:

```
column_clause
| ADD table_constraint [ NOT VALID ]
| ADD table_constraint_using_index
| VALIDATE CONSTRAINT constraint_name
| DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
| CLUSTER ON index_name
| SET WITHOUT CLUSTER
| SET ( {storage_parameter = value} [, ... ] )
| RESET ( storage_parameter [, ... ] )
| OWNER TO new_owner
| SET TABLESPACE new_tablespace
| SET {COMPRESS|NOCOMPRESS}
| DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH ( column_name [,...] ) } }
| TO { GROUP groupname | NODE ( nodename [, ... ] ) }
| ADD NODE ( nodename [, ... ] )
| DELETE NODE ( nodename [, ... ] )
| DISABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE REPLICA TRIGGER trigger_name
| ENABLE ALWAYS TRIGGER trigger_name
| DISABLE ROW LEVEL SECURITY
| ENABLE ROW LEVEL SECURITY
| FORCE ROW LEVEL SECURITY
| NO FORCE ROW LEVEL SECURITY
| REFRESH STORAGE
```

- **ADD table_constraint [NOT VALID]**
Adds a new table constraint. When used with the **NOT VALID** option, this constraint is valid only for foreign keys and **CHECK** constraints. If the **NOT VALID** option is added to the constraint, the check on whether the existing records in the table meet the initial constraint is skipped.
- **ADD table_constraint_using_index**
Adds primary key constraint or unique constraint based on the unique index.
- **VALIDATE CONSTRAINT constraint_name**
Validates a foreign key or check constraint that was previously created as **NOT VALID**, by scanning the table to ensure there are no rows for which the constraint is not satisfied. Nothing happens if the constraint is already marked valid.
- **DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]**
Drops a table constraint.
- **CLUSTER ON index_name**
Selects the default index for future **CLUSTER** operations. It does not actually re-cluster the table.
- **SET WITHOUT CLUSTER**
Removes the most recently used **CLUSTER** index specification from the table. This operation affects future cluster operations that do not specify an index.
- **SET ({storage_parameter = value} [, ...])**
Changes one or more storage parameters for the table.
- **RESET (storage_parameter [, ...])**
Resets one or more storage parameters to their defaults. As with **SET**, a table rewrite might be needed to update the table entirely.

- **OWNER TO new_owner**

Changes the owner of the table, sequence, or view to the specified user.

- **SET {COMPRESS|NOCOMPRESS}**

Sets the compression feature of a table. The table compression feature affects only the storage mode of data inserted in a batch subsequently and does not affect storage of existing data. Setting the table compression feature will result in the fact that there are both compressed and uncompressed data in the table.

- **DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH (column_name [,...]) } }**

Changing a table's distribution mode will physically redistribute the table data based on the new distribution mode. After the distribution mode is changed, you are advised to manually run the **ANALYZE** statement to collect new statistics about the table.

 NOTE

- This operation is a major change operation, involving table distribution information modification and physical data redistribution. During the modification, services are blocked. After the modification, the original execution plan of services will change. Perform this operation according to the standard change process.
 - This operation is a resource-intensive operation. If you need to modify the distribution mode of large tables, perform the operation when the computing and storage resources are sufficient. Ensure that the remaining space of the entire cluster and the tablespace where the original table is located is sufficient to store a table that has the same size as the original table and is distributed in the new distribution mode.
- **TO { GROUP groupname | NODE (nodename [, ...]) }**
The syntax is only available in extended mode (when GUC parameter **enable_cluster_resize** is on). Exercise caution when enabling the mode. It is used for tools like internal dilatation tools. Common users should not use the mode.
 - **ADD NODE (nodename [, ...])**
It is only available for tools like internal dilatation. General users should not use the mode.
 - **DELETE NODE (nodename [, ...])**
It is only available for internal scale-in tools. Common users should not use the syntax.
 - **DISABLE TRIGGER [trigger_name | ALL | USER]**
Disables a single trigger specified by **trigger_name**, disables all triggers, or disables only user triggers (excluding internally generated constraint triggers, for example, deferrable unique constraint triggers and exclusion constraints triggers).

 NOTE

Exercise caution when using this function because data integrity cannot be ensured as expected if the triggers are not executed.

- **ENABLE TRIGGER [trigger_name | ALL | USER]**

Enables a single trigger specified by **trigger_name**, enables all triggers, or enables only user triggers.

- **ENABLE REPLICA TRIGGER trigger_name**
Determines that the trigger firing mechanism is affected by the configuration variable **session_replication_role**. When the replication role is **origin** (default value) or **local**, a simple trigger is fired.
When **ENABLE REPLICA** is configured for a trigger, it is fired only when the session is in **replica** mode.
- **ENABLE ALWAYS TRIGGER trigger_name**
Determines that all triggers are fired regardless of the current replication mode.
- **DISABLE/ENABLE ROW LEVEL SECURITY**
Enables or disables row-level access control for a table.
If row-level access control is enabled for a data table but no row-level access control policy is defined, the row-level access to the data table is not affected. If row-level access control for a table is disabled, the row-level access to the table is not affected even if a row-level access control policy has been defined. For details, see [CREATE ROW LEVEL SECURITY POLICY](#).
 **NOTE**
Row-level access control cannot be enabled for HDFS tables, foreign tables, and temporary tables.
- **NO FORCE/FORCE ROW LEVEL SECURITY**
Forcibly enables or disables row-level access control for a table.
By default, the table owner is not affected by the row-level access control feature. However, if row-level access control is forcibly enabled, the table owner (excluding system administrators) will be affected. System administrators are not affected by any row-level access control policies.
- **REFRESH STORAGE**
Changes the local hot partitions that meet the criteria specified in the **storage_policy** parameter of an OBS multi-temperature table to the cold partitions stored in the OBS.
For example, if **storage_policy** is set to **'LMT:10'** for an OBS hot or cold table when it is created, the partitions that are not updated within the last 10 days are switched to cold partitions in the OBS.

Clauses of column_clause Related to Column Operations

There are several clauses of **column_clause**:

```
ADD [ COLUMN ] column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint [ ... ] ]
| MODIFY [ COLUMN ] column_name data_type
| MODIFY [ COLUMN ] column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
| MODIFY [ COLUMN ] column_name [ CONSTRAINT constraint_name ] NULL
| MODIFY [ COLUMN ] column_name DEFAULT default_expr
| MODIFY [ COLUMN ] column_name ON UPDATE on_update_expr
| MODIFY [ COLUMN ] column_name COMMENT comment_text
| DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
| ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ USING expression ]
| ALTER [ COLUMN ] column_name { SET DEFAULT expression | DROP DEFAULT }
| ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
| ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
| ADD STATISTICS (( column_1_name, column_2_name [, ...] ))
| ADD { INDEX | UNIQUE [ INDEX ] } [ index_name ] ( { { column_name | ( expression ) } [ COLLATE
```

```
collation ] [ opclass ] [ ASC | DESC ] [ NULLS LAST ] } [, ... ] ) [ USING method ] [ NULLS [ NOT ] DISTINCT  
| NULLS IGNORE ] [ COMMENT 'text' ] LOCAL [ ( { PARTITION index_partition_name } [, ... ] ) ] [ WITH  
( { storage_parameter = value } [, ... ] ) ]  
| ADD { INDEX | UNIQUE [ INDEX ] } [ index_name ] ( { column_name | ( expression ) } [ COLLATE  
collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } } [, ... ] ) [ USING method ] [ NULLS [ NOT ]  
DISTINCT | NULLS IGNORE ] [ COMMENT 'text' ] [ WITH ( { storage_parameter = value } [, ... ] ) ] [ WHERE  
predicate ]  
| DROP { INDEX | KEY } index_name  
| CHANGE [ COLUMN ] old_column_name new_column_name data_type [ [ CONSTRAINT  
constraint_name ] NOT NULL [ ENABLE ] ]  
[ CONSTRAINT constraint_name ] NULL | DEFAULT default_expr | COMMENT 'text' ]  
| DELETE STATISTICS (( column_1_name, column_2_name [, ... ] ) )  
| ALTER [ COLUMN ] column_name SET ( { attribute_option = value } [, ... ] )  
| ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )  
| ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
```

- **ADD [COLUMN] column_name data_type [compress_mode] [COLLATE collation] [column_constraint [...]]**

Adds a column to a table. If a column is added with **ADD COLUMN**, all existing rows in the table are initialized with the column's default value (**NULL** if no **DEFAULT** clause is specified).

- **ADD ({ column_name data_type [compress_mode] } [, ...])**

Adds columns in the table.

- **MODIFY [COLUMN] column_name data_type**

Modifies the data type of an existing field in a table. Note that the data type of the distribution column cannot be modified.

- **MODIFY [COLUMN] column_name [CONSTRAINT constraint_name] NOT NULL [ENABLE]**

Adds a NOT NULL constraint to a column of a table. Currently, this clause is unavailable to column-store tables.

- **MODIFY [COLUMN] column_name [CONSTRAINT constraint_name] NULL**

Deletes the NOT NULL constraint to a certain column in the table.

- **MODIFY [COLUMN] column_name DEFAULT default_expr**

Changes the default value of the table.

- **MODIFY [COLUMN] column_name ON UPDATE on_update_expr**

Modifies the ON UPDATE expression of a specified column in a table. The column must be of the timestamp or timestampz type. If **on_update_expr** is **NULL**, the **ON UPDATE** clause is deleted.

- **MODIFY [COLUMN] column_name COMMENT comment_text**

Modifies the comment of the table.

- **DROP [COLUMN] [IF EXISTS] column_name [RESTRICT | CASCADE]**

Drops a column from a table. Index and constraint related to the column are automatically dropped. If an object not belonging to the table depends on the column, **CASCADE** must be specified, such as foreign key reference and view.

The **DROP COLUMN** form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a **NULL** value for the column. Therefore, column deletion takes a short period of time but does not immediately release the table space on the disks, because the space occupied by the deleted column is not reclaimed. The space will be reclaimed when **VACUUM** is executed.

- **ALTER [COLUMN] column_name [SET DATA] TYPE data_type [COLLATE collation] [USING expression]**

Change the data type of a field in the table. Only the type conversion of the same category (between values, character strings, and time) is allowed. Indexes and simple table constraints on the column will automatically use the new data type by reparsing the originally supplied expression.

ALTER TYPE requires an entire table be rewritten. This is an advantage sometimes, because it frees up unnecessary space from a table. For example, to reclaim the space occupied by a deleted column, the fastest method is to use the command.

```
ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
```

In this command, **anycol** indicates any column existing in the table and **anytype** indicates the type of the prototype of the column. **ALTER TYPE** does not change the table except that the table is forcibly rewritten. In this way, the data that is no longer used is deleted.

USING in **ALTER TYPE** can specify any expression involving the old values of the row; that is, it can refer to any columns other than the one being converted. This allows very general conversions to be done with the **ALTER TYPE** syntax. Because of this flexibility, the **USING** expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, **ALTER TYPE** might fail to convert the default even though a **USING** clause is supplied. In such cases, drop the default with **DROP DEFAULT**, perform the **ALTER TYPE**, and then use **SET DEFAULT** to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

- **ALTER [COLUMN] column_name { SET DEFAULT expression | DROP DEFAULT }**

Sets or removes the default value for a column. The default values only apply to subsequent **INSERT** commands; they do not cause rows already in the table to change. Defaults can also be created for views, in which case they are inserted into **INSERT** statements on the view before the view's **ON INSERT** rule is applied.

- **ALTER [COLUMN] column_name { SET | DROP } NOT NULL**

Changes whether a column is marked to allow **NULL** values or to reject **NULL** values. You can only use **SET NOT NULL** when the column contains no **NULL** values.

- **ALTER [COLUMN] column_name SET STATISTICS [PERCENT] integer**

Specifies the per-column statistics-gathering target for subsequent **ANALYZE** operations. The value ranges from 0 to 10000. Set it to -1 to revert to using the default system statistics target.

- **ADD { INDEX | UNIQUE [INDEX] } [index_name] ({ { column_name | (expression) } [COLLATE collation] [opclass] [ASC | DESC] [NULLS LAST] } [, ...]) [USING method] [NULLS [NOT] DISTINCT | NULLS IGNORE] [COMMENT 'text'] LOCAL [({ PARTITION index_partition_name } [, ...])] [WITH ({ storage_parameter = value } [, ...])]**

Create an index for the partitioned table. For details about the parameters, see [CREATE INDEX](#).

- **ADD { INDEX | UNIQUE [INDEX] } [index_name] ({ { column_name | (expression) } [COLLATE collation] [opclass] [ASC | DESC] [NULLS { FIRST | LAST }] } [, ...]) [USING method] [NULLS [NOT] DISTINCT | NULLS IGNORE] [COMMENT 'text'] [WITH ({storage_parameter = value} [, ...])] [WHERE predicate]**
Create an index on the table. For details about the parameters, see [CREATE INDEX](#).
- **DROP { INDEX | KEY } index_name**
Deletes an index from a table.
- **CHANGE [COLUMN] old_column_name new_column_name data_type [[CONSTRAINT constraint_name] NOT NULL [ENABLE] | [CONSTRAINT constraint_name] NULL | DEFAULT default_expr | COMMENT 'text']**
Modifies the column information in the table, such as column names and column field information.
- **{ADD | DELETE} STATISTICS ((column_1_name, column_2_name [, ...]))**
Adds or deletes the declaration of collecting multi-column statistics to collect multi-column statistics as needed when **ANALYZE** is performed for a table or a database. The statistics about a maximum of 32 columns can be collected at a time. You are not allowed to add or delete the declaration for system tables or foreign tables
- **ALTER [COLUMN] column_name SET ({attribute_option = value} [, ...])**
ALTER [COLUMN] column_name RESET (attribute_option [, ...])
Sets or resets per-attribute options.
The attribute option parameters are **n_distinct**, **n_distinct_inherited**, and **cstore_cu_sample_ratio**. **n_distinct** specifies and fixes the statistics of a table's distinct values. **n_distinct_inherited** specifies and inherits the distinct value statistics. **cstore_cu_sample_ratio** specifies the CU ratio for **ANALYZE** on a column-store table. Currently, the **n_distinct_inherited** parameter cannot be set or reset.
 - **n_distinct**
Sets the distinct value statistics of the column.
Value range: -1.0 to INT_MAX
Default value: **0**, indicating that this parameter is not set.
 - **n_distinct_inherited**
Sets the distinct value statistics of the column in an inherited table.
Value range: -1.0 to INT_MAX
Default value: **0**, indicating that this parameter is not set.
 - **cstore_cu_sample_ratio**
Specifies the expansion multiple in the calculation of CUs to be sampled during **ANALYZE** on a column-store table.
Value range: 1.0-10000.0
Default value: **1.0**
- **ALTER [COLUMN] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }**

Sets the storage mode for a column. This clause specifies whether this column is held inline or in a secondary TOAST table, and whether the data should be compressed. This statement can only be used for row-based tables. **SET STORAGE** only sets the strategy to be used for future table operations.

Options of Other Parameters

- **column_constraint** is as follows:
[CONSTRAINT constraint_name]
{ NOT NULL |
NULL |
CHECK (expression) |
DEFAULT default_expr |
UNIQUE [NULLS [NOT] DISTINCT | NULLS IGNORE] index_parameters |
PRIMARY KEY index_parameters }
[DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE]
- **compress_mode** of a column is as follows:
[DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS]
- **table_constraint_using_index** used to add the primary key constraint or unique constraint based on the unique index is as follows:
[CONSTRAINT constraint_name]
{ UNIQUE | PRIMARY KEY } USING INDEX index_name
[DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE]
- Add foreign key constraint **REFERENCES**.
[CONSTRAINT constraint_name]
FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])] }
- **table_constraint** is as follows:
[CONSTRAINT constraint_name]
{ CHECK (expression) |
UNIQUE [NULLS [NOT] DISTINCT | NULLS IGNORE] (column_name [, ...]) index_parameters |
PRIMARY KEY (column_name [, ...]) index_parameters }
[DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE]
- **index_parameters** is as follows:
[WITH ({storage_parameter = value} [, ...])]
[USING INDEX TABLESPACE tablespace_name]

Parameter Description

Table 13-46 ALTER TABLE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a prompt instead of an error if the table name does not exist.	-

Parameter	Description	Value Range
ONLY	Performs operations only on the specified table name. If ONLY is not specified: The table and all subtables will be modified. Note that you can only add or delete columns to or from a parent or child table, or change the type of a column. The parent table and its subsequent representatives must always have the same columns and types.	-
table_name	Specifies the name of the table to be modified. You can add the asterisk (*) option following the table name to specify that all subtables are scanned, which is the default operation.	Valid table name.
new_table_name	New name of a table.	A string compliant with the identifier naming rules .
column_name, column_1_name, and column_2_name	Specifies the name of an existing or new field.	Valid column name.
new_column_name	Specifies the new name of a column.	A string compliant with the identifier naming rules .
constraint_name	Specifies the name of a constraint.	Valid constraint name.
new_constraint_name	Specifies the new name of a table constraint.	A string compliant with the identifier naming rules .
schema_name	Specifies the schema name of a table.	Valid schema name.
new_schema	Specifies the new schema name.	Valid schema name.
index_name	Specifies the index name.	Valid index name.
new_owner	Specifies the name of the new owner of the table.	Valid username or role name.
new_tablespace	Specifies the name of the new tablespace to which the table belongs.	Valid tablespace name.

Parameter	Description	Value Range
data_type	Specifies the type of a new column or a new type of an existing column.	A string compliant with the identifier naming rules .
storage_parameter	Specifies the storage parameters of a table.	For details, see Table 13-47 .
compress_mode	Specifies the compression option of the table column. Currently, this parameter is valid only for row-store tables. The clause specifies the algorithm preferentially used by the column.	-
collation	Specifies the collation rule name of a column. The optional COLLATE clause specifies a collation for the new column; if omitted, the collation is the default for the new column.	-
USING expression	A USING clause specifies how to compute the new column value from the old; if omitted, the default conversion is an assignment cast from old data type to new. A USING clause must be provided if there is no implicit or assignment cast from the old to new type.	-
NOT NULL NULL	Sets whether the column allows null values.	-
integer	Specifies the constant value of an integer with a sign. When PERCENT is used, statistics are collected based on the percentage of table data.	0~100
attribute_option	Specifies an attribute option.	-

Parameter	Description	Value Range
PLAIN EXTERNAL EXTENDED MAIN	Specifies a column storage mode. • PLAIN must be used for fixed-length values (such as integers). It must be inline and uncompressed. • MAIN is for inline, compressible data. • EXTERNAL is for external, uncompressed data. Use of EXTERNAL will make substring operations on text and bytea values run faster, at the penalty of increased storage space. • EXTENDED is for external, compressed data. EXTENDED is the default for most data types that support non-PLAIN storage.	-
CHECK (expression)	New or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE succeed. If any row of an insert or update operation produces a FALSE result, an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns. Currently, CHECK expression does not include subqueries and cannot use variables apart from the current column.	-

Parameter	Description	Value Range
DEFAULT default_expr	Assigns a default data value for a column. The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is NULL. If a suffix operator, such as (!). is used in default_expr, enclose the operator in parentheses.	-
UNIQUE [NULLS [NOT] DISTINCT NULLS IGNORE] index_parameters UNIQUE (column_name [, ...]) [NULLS [NOT] DISTINCT NULLS IGNORE] index_parameters	The UNIQUE constraint specifies that a group of one or more columns of a table can contain only unique values. The [NULLS [NOT] DISTINCT NULLS IGNORE] field is used to specify how to process null values in the index column of the Unique index. For details, see Table 13-48. Default value: This parameter is left empty by default. NULL values can be inserted repeatedly.	-
PRIMARY KEY index_parameters PRIMARY KEY (column_name [, ...]) index_parameters	The primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values.	-

Parameter	Description	Value Range
DEFERRABLE NOT DEFERRABLE INITIALLY DEFERRED INITIALLY IMMEDIATE	Sets whether the constraint is deferrable. This option is unavailable to column-store tables. • DEFERRABLE: deferrable can be postponed until the end of the transaction using the SET CONSTRAINTS command. • NOT DEFERRABLE: checks immediately after the execution of each command. • INITIALLY IMMEDIATE: checks immediately after the execution of each statement. • INITIALLY DEFERRED: checks when the transaction ends.	-
WITH ({storage_parameter = value} [, ...])	Specifies an optional storage parameter for a table or an index.	-
COMPRESS NOCOMPRESS	• NOCOMPRESS: If the NOCOMPRESS keyword is specified, the existing compression feature of the table is not changed. • COMPRESS: If the COMPRESS keyword is specified, the table compression feature is triggered if tuples are inserted in a batch.	-
CASCADE	Automatically drops objects that depend on the dropped column or constraint (for example, views referencing the column).	-
RESTRICT	Refuses to drop the column or constraint if there are any dependent objects. This is the default behavior.	-

Parameter	Description	Value Range
cache_policy	Specifies the table cache policy. This parameter is supported only in storage-compute decoupled 3.0 clusters. Default value: ALL	• ALL: Hot cache is used for the entire table. • NONE: Cold cache is used for the entire table. • HPN: The first <i>N</i> partitions in a partitioned table use hot cache. The rest of the partitions use cold cache. • HPL: <i>P1, P2, ...</i> In a partitioned table, the specified partitions use hot cache. The rest of the partitions use cold cache. NOTE • For foreign tables and non-partitioned tables, only the ALL and NONE cache policies are supported. • Only range-partitioned and list-partitioned internal tables support HPN and HPL cache policies.

Table 13-47 storage_parameter parameters

Parameter Type	Parameter	Description	Value Range
Partition management parameters (newly added)	PERIOD	Sets the period for automatically creating partitions in partition management. NOTE • If this parameter is not configured when you create a table, you can run the set statements to configure this parameter and enable automatic partition creation. If this parameter has been configured before, you can run the set statements to modify this parameter. • You can run the reset command to disable the automatic partition creation function. However, if the automatic partition deletion function is enabled, the automatic partition creation function cannot be disabled.	Interval type. For details about the value range of PERIOD and the constraints on enabling this function, see the PERIOD parameter in WITH Clause Parameters.
	TTL	Sets the partition expiration time for automatically deleting partitions in partition management. NOTE • If this parameter is not configured when you create a table, you can run the set statements to configure this parameter and enable automatic partition deletion. If this parameter has been configured before, you can run the set statements to modify this parameter. • You can run the reset command to disable the automatic partition deletion.	Interval type. For details about the value range of TTL and restrictions on enabling this function, see the TTL parameters in WITH Clause Parameters.

Parameter Type	Parameter	Description	Value Range
Parameters related to column-store tables in Turbo storage format (newly added)	enable_turbo_store	Specifies whether the column-store table is in the turbo storage format. This is supported only by 9.1.0.100 and later cluster versions. NOTE - Common column-store tables in version 3.0 do not support the turbo storage format, while HStore Opt tables in version 3.0 only support the turbo storage format. - In version 2.0, there is no restriction on column-store tables.	-

The NULL value of the index column in the unique index is processed as follows:

Use the **[NULLS [NOT] DISTINCT | NULLS IGNORE]** field to set how null values are handled in the unique index column.

When the inserted data is compared with the original data in the table, the NULL value can be processed in any of the following ways:

- **NULLS DISTINCT:** NULL values are unequal and can be inserted repeatedly.
- **NULLS NOT DISTINCT:** NULL values are equal. If all index columns are NULL, NULL values cannot be inserted repeatedly. If some index columns are NULL, data can be inserted only when non-null values are different.
- **NULLS IGNORE:** NULL values are skipped during the equivalent comparison. If all index columns are NULL, NULL values can be inserted repeatedly. If some index columns are NULL, data can be inserted only when non-null values are different.

Table 13-48 Processing of NULL values in index columns in unique indexes

Constraint	All Index Columns Are NULL	Some Index Columns Are NULL.
NULLS DISTINCT	Can be inserted repeatedly.	Can be inserted repeatedly.
NULLS NOT DISTINCT	Cannot be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.

Constraint	All Index Columns Are NULL	Some Index Columns Are NULL.
NULLS IGNORE	Can be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.

Table Operation Examples

Create the sample table **customer**.

```
DROP TABLE IF EXISTS customer;  
CREATE TABLE customer  
(  
    C_CUSTKEY    BIGINT      ,  
    C_NAME      VARCHAR(25) ,  
    C_ADDRESS    VARCHAR(40) ,  
    C_NATIONKEY  INT         ,  
    C_PHONE      CHAR(15)    ,  
    C_ACCTBAL    DECIMAL(15,2)  
)  
DISTRIBUTE BY HASH(C_CUSTKEY);
```

Create the column-store table **customer_address**.

```
DROP TABLE IF EXISTS customer_address;  
CREATE TABLE customer_address  
(  
    ca_address_sk    INTEGER          NOT NULL ,  
    ca_address_id    CHARACTER(16)    NOT NULL ,  
    ca_street_number CHARACTER(10)      ,  
    ca_street_name   CHARACTER varying(60) ,  
    ca_street_type   CHARACTER(15)      ,  
    ca_suite_number  CHARACTER(10)  
)  
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)  
DISTRIBUTE BY HASH (ca_address_sk);
```

- Change the schema of a table.
Set the schema of the **customer** table to **myschema**.

```
ALTER TABLE customer SET SCHEMA myschema;
```
- Add a primary key constraint or unique constraint based on the unique index.
Create a unique index **customer_constraint1** for the customer table, add a primary key constraint based on the existing unique index, and rename the index.

```
CREATE UNIQUE INDEX customer_constraint1 ON customer(C_CUSTKEY);  
ALTER TABLE customer ADD CONSTRAINT customer_constraint2 PRIMARY KEY USING INDEX customer_constraint1;
```
- Rename a table constraint.

```
ALTER TABLE customer RENAME CONSTRAINT customer_constraint2 TO customer_constraint;
```
- Delete a table constraint.

```
ALTER TABLE customer DROP CONSTRAINT customer_constraint;
```
- Add a primary key constraint to a table.

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY(C_CUSTKEY);
```
- Add a partial cluster column to the column-store table **customer_address**.

```
ALTER TABLE customer_address ADD CONSTRAINT customer_address_cluster PARTIAL CLUSTER KEY(ca_address_sk);
```

- Delete the partial cluster column from the column-store table **customer_address**.
`ALTER TABLE customer_address DROP CONSTRAINT customer_address_cluster;`
- Modify the storage format of a column-store table.
`ALTER TABLE customer_address SET (COLVERSION = 1.0);`
- Change the distribution mode of a table.
`ALTER TABLE customer_address DISTRIBUTE BY REPLICATION;`
- Rename a table.
`ALTER TABLE CUSTOMER RENAME TO CUSTOMER_t;`
- Modify the table cache policy (supported only in storage-compute decoupled 3.0 clusters).
`ALTER TABLE orders SET (cache_policy = 'NONE');`

Index Operation Example

- Add an index to a table.
`ALTER TABLE customer ADD INDEX customer_index(C_CUSTKEY);`
- Delete a table index.
`ALTER TABLE customer DROP INDEX customer_index;`
`ALTER TABLE customer DROP KEY customer_index;`
- Adds an index to a column in a table.
`ALTER TABLE CUSTOMER ADD c_address_id varchar(20) CONSTRAINT ca_address_index CHECK (c_address_id > 0);`

Hot and Cold Table Operation Example

- Do a cold or hot switchover for a single table.
`ALTER TABLE cold_hot_table REFRESH STORAGE;`
- Change a column-store partitioned table to a hot and cold table.
`CREATE table test_1(id int,d_time date)
WITH(ORIENTATION=COLUMN)
DISTRIBUTE BY HASH (id)
PARTITION BY RANGE (d_time)
(PARTITION p1 START('2022-01-01') END('2022-01-31') EVERY(interval '1 day'))

ALTER TABLE test_1 SET (storage_policy = 'LMT:100');`

Column Operation Example

Create the sample table **warehouse_t**.

```
DROP TABLE IF EXISTS warehouse_t;  
CREATE TABLE warehouse_t  
(  
    W_WAREHOUSE_SK      INTEGER      NOT NULL,  
    W_WAREHOUSE_ID      CHAR(16)     NOT NULL,  
    W_WAREHOUSE_NAME     VARCHAR(20)  UNIQUE DEFERRABLE,  
    W_WAREHOUSE_SQ_FT    INTEGER      ,  
    W_COUNTY             VARCHAR(30)  ,  
    W_STATE              CHAR(2)      DEFAULT 'GA',  
    W_ZIP                CHAR(10)  
);
```

- Adds a column.
Adds a column to the **warehouse_t** table.
`ALTER TABLE warehouse_t ADD W_GOODS_CATEGORY int;`
Adds multiple columns to the **warehouse_t** table.
`ALTER TABLE warehouse_t ADD (W_COMMENT VARCHAR(117) NOT NULL, W_COUNT int);`
- Delete a column.

Delete a column from the **warehouse_t** table.

```
ALTER TABLE warehouse_t DROP COLUMN W_STATE;
```

- Rename a column.

Change the column name **W_ZIP** of the **warehouse_t** table to **new_W_ZIP**.

```
ALTER TABLE warehouse_t RENAME W_ZIP TO new_W_ZIP;
```

- Change the column name and column field information.

Change the column name **W_GOODS_CATEGORY** of the **warehouse_t** table to **W_GOODS_CATEGORY2**, change the data type of the **W_GOODS_CATEGORY2** column to **DECIMAL**, and set the column constraint to **NOT NULL**.

```
ALTER TABLE warehouse_t CHANGE W_GOODS_CATEGORY W_GOODS_CATEGORY2 DECIMAL NOT NULL COMMENT 'W_GOODS_CATEGORY';
```

- Change the comment of a column.

```
ALTER TABLE warehouse_t MODIFY W_GOODS_CATEGORY2 COMMENT 'W_GOODS_CATEGORY';
```

- Change the data type of a column.

Change the data type of the **warehouse_t** column and set the column constraint to **NOT NULL**.

```
ALTER TABLE warehouse_t MODIFY W_WAREHOUSE_SQ_FT varchar(20) NOT NULL;
```

- Add the **NOT NULL** constraint to a column.

```
ALTER TABLE warehouse_t ALTER COLUMN W_COUNTY SET NOT NULL;
```

- Add a primary key to the **warehouse_t** table.

```
ALTER TABLE warehouse_t ADD PRIMARY KEY(w_warehouse_name);
```

- Modify the **ON UPDATE** expression of a specified column in a table.

Add a timestamp column with the **ON UPDATE** expression to the **warehouse_t** table. The timestamp of the column is automatically updated when an **UPDATE** operation is performed on the table.

```
ALTER TABLE warehouse_t ADD COLUMN C_TIME timestamp on update current_timestamp;
```

Delete the timestamp column with the **ON UPDATE** expression from the **warehouse_t** table.

```
ALTER TABLE warehouse_t MODIFY COLUMN C_TIME timestamp on update NULL;
```

Helpful Links

[CREATE TABLE](#), [12.101-RENAME TABLE](#), and [DROP TABLE](#)

FAQs

[How Do I Adjust GaussDB\(DWS\) Distribution Columns?](#)

[Error "invalid input syntax for xxx" Is Reported During Data Type Conversion](#)

[When Inserting Data Into a Table, An Error Is Reported: "duplicate key value violates unique constraint "%s""](#)

13.26 ALTER TABLE PARTITION

Function

ALTER TABLE PARTITION is used to modify table partitions, including adding, deleting, splitting, merging partitions, migrating data from ordinary tables to partitioned tables, and modifying partition attributes.

A partitioned table is a logical table that is divided into segments, called partitions whose data is stored on different physical blocks.

Common partitioning strategies define data ranges based on one or more columns. Each partition stores data within a range. These columns are called partition keys.

Currently, DWS row-store and column-store tables support range partitioning and list partitioning.

- Range partitioning

A table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. Each range has a dedicated partition for data storage.

- Core feature: Data is distributed by consecutive intervals. It is used in scenarios where data is ordered and retrieved by range.
- Scenario: Partition keys increase linearly or have continuous intervals. Data query is usually performed around a certain range, and new data naturally falls into a new range. For example, data about orders is partitioned by time range, and data about user levels is partitioned by value range.
- Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition. This is the most commonly used partitioning policy. Currently, range partitioning only allows the use of the range partitioning policy.

- List partitioning

List partitioning allocates records to partitions based on the key values in each partition. The key values do not overlap in different partitions. Each set of key values gets its own partition for storing related data. List partitioning is supported only by clusters of version 8.1.3 or later.

- Core feature: Data is divided based on discrete enumerated values, which are used as fixed classification category or high-frequency filter criteria.
- Scenario: Partition key values are fixed and discrete. Data query is usually based on a certain category, and key values do not increase irregularly. For example, the user information table is partitioned by province, and each province corresponds to a partition.
- In list partitioning, data is mapped to a created partition based on the partition key value. If the data can be mapped to, it is inserted into the specific partition. If it cannot be mapped to, error messages are returned.

Precautions

- The name of the partition to be added cannot be the same as that of an existing partition in the partition table.
- For a range partitioned table, the boundary value of the added partition must be the same type as the partition key of the partitioned table. The key value of the added partition must exceed the upper limit of the last partition.
- For a list partitioned table, if the DEFAULT partition has been defined, no new partition can be added.
- Unless otherwise specified, the syntax of range partitioned tables is the same as that of column-store partitioned tables.

- If the number of partitions in the target partitioned table has reached the maximum (32767), partitions cannot be added.
- If a partitioned table has only one partition, the partition cannot be deleted.
- When you run the **DROP PARTITION** command to delete a partition, the data in the partition is also deleted.
- Use **PARTITION FOR()** to choose partitions. The number of specified values in the brackets should be the same as the column number in customized partition, and they must be consistent.
- The **Value** partitioned table does not support the **Alter Partition** operation.
- For OBS hot and cold tables:
 - They do not support specifying the partition table's tablespace as the OBS tablespace for **MOVE**, **EXCHANGE**, **MERGE**, and **SPLIT** operations.
 - When an **ALTER** statement is executed, the data in the cold partition should stay in the cold partition, and the data in the hot partition should remain in the hot partition. It is not allowed to move cold partition data to the local tablespace.
 - Only the default tablespace is supported for cold partitions.
 - Cold and hot partitions cannot be merged.
 - Cold partition switching is not supported during the **EXCHANGE** operation.

**WARNING**

- Avoid performing **ALTER TABLE**, **ALTER TABLE PARTITION**, **DROP PARTITION**, and **TRUNCATE** operations during peak hours to prevent long SQL statements from blocking these operations or SQL services.
- For more information about development and design specifications, see [Development and Design Proposal](#).

Syntax

Modify the syntax of the table partition.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
action [, ... ];
```

action indicates the following clauses for maintaining partitions. For the partition continuity when multiple clauses are used for partition maintenance, DWS does **DROP PARTITION** and then **ADD PARTITION**, and finally runs the rest clauses in sequence.

- [modify_clause: Set Whether a Partition Index Is Usable](#)
- [rebuild_clause: Rebuild the Index of a Partition](#)
- [exchange_clause: Move the Data From an Ordinary Table to a Specified Partition](#)
- [row_clause: Enable or Disable Row Movement of a Partitioned Table](#)
- [merge_clause: Merge Partitions Into One](#)
- [split_clause: Split One Partition into Partitions](#)

- **add_clause sub-syntax: Add One or More Partitions to a Specified Partitioned Table**
- **drop_clause: Delete a Specified Partition or Multiple Partitions From a Partitioned Table**
- **truncate_partitioned_clause: Clear Data in a Table Partition**

ALTER TABLE PARTITION Main Clause Parameters

Table 13-49 ALTER TABLE PARTITION main clause parameters

Parameter	Description	Value Range
table_name	Specifies the name of the partitioned table to be modified.	Name of an existing partitioned table.
action [, ...]	Specifies clauses for maintaining partitions. For the partition continuity when multiple clauses are used for partition maintenance, DWS does DROP PARTITION and then ADD PARTITION , and finally runs the rest clauses in sequence.	For details, see the following clauses.
partition_name	Specifies the name of the partition to be modified.	Name of an existing partition.
partition_value	Specifies the key value of a partition. The value specified by PARTITION FOR (partition_value [, ...]) can uniquely identify a partition.	Value range of the partition key of the partition to be renamed.

modify_clause: Set Whether a Partition Index Is Usable

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
MODIFY PARTITION partition_name { UNUSABLE LOCAL INDEXES | REBUILD UNUSABLE LOCAL INDEXES }
```

Table 13-50 modify_clause parameters

Parameter	Description	Value Range
partition_name	Specifies the name of the partition to be modified.	Name of an existing partition.
UNUSABLE LOCAL INDEXES	Sets all the indexes unusable in the partition.	-
REBUILD UNUSABLE LOCAL INDEXES	Rebuilds all the indexes in the partition.	-

rebuild_clause: Rebuild the Index of a Partition

This syntax is supported only by clusters of version 8.3.0.100 or later.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
REBUILD PARTITION partition_name [ WITHOUT UNUSABLE ]
```

Table 13-51 rebuild_clause parameters

Parameter	Description	Value Range
WITHOUT UNUSABLE	Ignores indexes in the UNUSABLE state when indexes on a partition are rebuilt.	-

exchange_clause: Move the Data From an Ordinary Table to a Specified Partition

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
EXCHANGE PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) }  
  WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY ( ordinary_table_name )}  
  [ { WITH | WITHOUT } VALIDATION ] [ VERBOSE ]
```

Table 13-52 exchange_clause parameters

Parameter	Description	Value Range
partition_value	Specifies the key value of a partition. The value specified by PARTITION FOR (partition_value [, ...]) can uniquely identify a partition.	Value range of the partition key of the partition to be renamed.
ordinary_table_name	Specifies the name of the ordinary table whose data is to be migrated.	Name of an existing common table.

Parameter	Description	Value Range
{ WITH WITHOUT } VALIDATION	Checks whether the ordinary table data meets the specified partition key range of the partition to be exchanged. The default value is WITH. The check is time consuming, especially when the data volume is large. Therefore, use WITHOUT when you are sure that the current ordinary table data meets the partition key range of the partition to be exchanged.	• WITH: checks whether the ordinary table data meets the partition key range of the partition to be exchanged. If any data does not meet the required range, an error is reported. • WITHOUT: does not check whether the ordinary table data meets the partition key range of the partition to be exchanged.
VERBOSE	When VALIDATION is WITH, if the ordinary table contains data that is out of the partition key range, insert the data to the correct partition. If there is no correct partition where the data can be route to, an error is reported. CAUTION Only when VALIDATION is WITH, VERBOSE can be specified.	-

The ordinary table and the partitioned table whose data is to be exchanged must meet the following requirements:

- The number of columns of the ordinary table is the same as that of the partitioned table, and their information should be consistent, including the column name, data type, constraint, collation, storage parameter, compression, and data type of a deleted column.
- The compression information of the ordinary table and partitioned table should be consistent.
- The distribution column information of the ordinary table and the partitioned table should be consistent.
- The number and information of indexes of the ordinary table and the partitioned table should be consistent.
- The number and information of constraints of the ordinary table and the partitioned table should be consistent.
- The ordinary table cannot be a temporary table or unlogged table.

- Exchanging partitions between an ordinary table and a partitioned table requires them to be in the same logical cluster or node group to avoid slow data insertion. Otherwise, the exchange operation becomes a slow data copy process, especially with large tables.
- In online scale-out and redistribution scenarios, the exchange partition statement may interfere with the redistribution of ordinary tables and partitioned tables (if there are lock conflicts between the partition exchange and redistribution statement). Usually, the redistribution of ordinary tables and partitioned tables is retried twice after being interrupted, but if a table is exchanged too often, the redistribution may fail for multiple times. If the redistribution process of an ordinary table is interrupted by the partition exchange operation, the data has been replaced with the data in the original partitioned table during the redistribution retry. In this case, full redistribution will be performed again.
- If other columns following the last valid column in the partitioned table are deleted and the deleted columns are not considered, the partitioned table can be exchanged with the ordinary table as long as the columns of the two tables are the same.
- The table-level parameter **colversion** must be consistent between the column-store ordinary table and the column-store partitioned table. The colversion 2.0 and colversion 1.0 cannot be exchanged.

When the exchange is done, the data and tablespace of the ordinary table and partitioned table are exchanged. In this case, statistics about the ordinary table and the partitioned table become unreliable. Both tables should be analyzed again.

row_clause: Enable or Disable Row Movement of a Partitioned Table

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
{ ENABLE | DISABLE } ROW MOVEMENT
```

Table 13-53 row_clause parameters

Parameter	Description	Value Range
ENABLE/DISABLE ROW MOVEMENT	Specifies whether to relocate a row within a table. If the new value of a partition key in a row belongs to another partition, an error message is displayed or the row is moved to the corresponding partition. • ENABLE: The row is moved to the corresponding partition. If partition keys are frequently updated, enable this parameter. • DISABLE: The row is not moved to the corresponding partition. If partition keys are updated, an error is reported. If data cross-partition update is strictly prohibited, disable this parameter. The default value is DISABLE. CAUTION To enable cross-partition update, you need to enable row movement. However, if SELECT FOR UPDATE is executed concurrently to query the partitioned table, the query results may be inconsistent. Therefore, exercise caution when performing this operation.	The date column (partition key) of a partitioned table is partitioned by quarter into p_2023q1 and p_2023q2. The value 2023-02-15 of a row belongs to the partition p_2023q1 in the first quarter. After the value is updated to 2023-05-15, you can configure ROW MOVEMENT to determine whether to move the data in this row. • ENABLE: The row movement is enabled. The data is moved to the new partition p_2023q2. • DISABLE: The row movement is disabled. The system reports an error. For details, see Examples: Enabling and Disabling ROW MOVEMENT.

merge_clause: Merge Partitions Into One

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
MERGE PARTITIONS { partition_name } [, ...] INTO PARTITION partition_name
```

- The partition before the keyword **INTO** is called the source partition, and the partition after the **INTO** is called the target partition.
- The number of source partitions cannot be less than 2.
- The source partition name must be unique.
- The source partition cannot have unusable indexes. Otherwise, an error will be reported.
- The target partition name must either be the same as the name of the last source partition or different from all partition names of the table.

- The boundaries of the target partition are the union of the boundaries of all the source partitions.
- For a range partitioned table, all source partitions must have contiguous boundaries.
- For list partitioning, if the source partition contains a DEFAULT partition, the boundary of the target partition is also DEFAULT.

split_clause: Split One Partition into Partitions

split_clause specifies range partitioning.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) } { split_point_clause |  
no_split_point_clause }
```

- **split_point_clause:**

```
AT ( partition_value ) INTO ( PARTITION partition_name , PARTITION partition_name )
```

CAUTION

The size of split point should be in the range of splitting partition key. The split point can only split one partition into two.

-
- **no_split_point_clause:**

```
INTO { ( partition_less_than_item [, ...] ) | ( partition_start_end_item [, ...] ) }
```

CAUTION

- The first new partition key specified by **partition_less_than_item** must be larger than that of the former partition (if any), and the last partition key specified by **partition_less_than_item** must be equal to that of the splitting partition.
 - The start point (if any) of the first new partition specified by **partition_start_end_item** must be equal to the partition key (if any) of the previous partition. The end point (if any) of the last partition specified by **partition_start_end_item** must be equal to the partition key of the splitting partition.
 - **partition_less_than_item** supports a maximum of four partition keys and **partition_start_end_item** supports only one partition key. For details about the supported data types, see the **PARTITION BY RANGE(partition_key)** parameter in [Table 13-120](#).
 - **partition_less_than_item** and **partition_start_end_item** cannot be used in the same statement. There is no restriction on different SPLIT statements.
-
- **partition_less_than_item:**
PARTITION partition_name VALUES LESS THAN ({ partition_value | MAXVALUE } [, ...])
 - **partition_start_end_item** (For details about the restrictions, see the **partition_start_end_item** parameter in [Table 13-120](#)).
PARTITION partition_name {
 {START(partition_value) END (partition_value) EVERY (interval_value)} |

```
{START(partition_value) END ({partition_value | MAXVALUE})} |  
{START(partition_value)} |  
{END({partition_value | MAXVALUE})}  
}
```

split_clause specifies list partitioning.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) } { split_values_clause |  
split_no_values_clause }
```

- **split_values_clause** specifies a split point.

```
VALUES ( { (partition_value) [, ...] } | DEFAULT ) INTO ( PARTITION partition_name , PARTITION  
partition_name )
```

CAUTION

- If the source partition is not a **DEFAULT partition**, the boundary specified by the cut point is a non-null true subset of the source partition boundary. If the source partition is a **DEFAULT** partition, the boundary specified by the cut point cannot overlap with the boundaries of other non-**DEFAULT** partitions.
- The boundary specified by the split point is the boundary of the first partition after the keyword **INTO**. The difference between the boundary of the source partition and the specified boundary of the split point is the boundary of the second partition.
- If the source partition is the **DEFAULT** partition, the boundary of the second partition is still **DEFAULT**.

-
- **split_no_values_clause** specifies no split point is specified.

```
INTO ( list_partition_item [, ...], PARTITION partition_name )
```

CAUTION

- The syntax of **list_partition_item** is the same as that of specifying the partition when creating a list partitioned table, except that the boundary value in the partition definition cannot be **DEFAULT**.
 - Except for the last partition, the boundaries of other partitions must be explicitly defined. The defined boundary cannot be **DEFAULT** and must be a non-empty proper subset of the source partition boundary. The boundary of the last partition is the difference set between the source partition boundary and other partition boundaries, and the boundary of the last partition is empty (that is, the difference set cannot be empty).
 - If the source partition is a **DEFAULT** partition, the boundary of the last partition is **DEFAULT**.
-

add_clause sub-syntax: Add One or More Partitions to a Specified Partitioned Table

add_clause specifies range partitioning.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
ADD { partition_less_than_item... | partition_start_end_item }
```

 CAUTION

- The **partition_less_than_item** syntax can only be used for range partitioned tables. Otherwise, an error will be reported.
- The syntax of **partition_less_than_item** is the same as the syntax specifying partitions in creating a range partitioned table.
- If the boundary value of the last partition is a MAXVALUE, new partitions cannot be added. Otherwise, an error will be reported.

add_clause specifies list partitioning.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
ADD list_partition_item
```

 CAUTION

- The **list_partition_item** syntax can only be used for a list partitioned table. Otherwise, an error will be reported.
- The **list_partition_item** syntax is the same as the syntax for specifying partitions when creating a list partitioned table.
- If the current partitioned table contains DEFAULT partitions, no new partitions can be added. Otherwise, an error will be reported.

drop_clause: Delete a Specified Partition or Multiple Partitions From a Partitioned Table

Delete a specified partition.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
DROP PARTITION { partition_name | FOR ( partition_value [, ...] ) }
```

Delete multiple partitions. (This feature is supported by clusters of version 8.1.3.100 or later.)

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
DROP PARTITION { partition_name [, ...] }
```

truncate_partitioned_clause: Clear Data in a Table Partition

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
TRUNCATE PARTITION { partition_name | FOR ( partition_value [, ...] ) };
```

- **partition_value** indicates the partition key value. Multiple partition key values can be specified. Use commas (,) to separate multiple partition key values.
- When the **PARTITION FOR** clause is used, the entire partition where **partition_value** is located is cleared.

Syntax for Modifying a Partition Name

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
RENAME PARTITION { partition_name | FOR ( partition_value [, ...] ) } TO partition_new_name;
```

Table 13-54 Parameters for modifying a partition name

Parameter	Description	Value Range
partition_new_name	Specifies the new name of the partition.	A string compliant with the identifier naming rules .

Example: Create a Range Partitioned Table `customer_address`

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
  ca_address_sk    INTEGER          NOT NULL ,
  ca_address_id    CHARACTER(16)    NOT NULL ,
  ca_street_number CHARACTER(10)    ,
  ca_street_name   CHARACTER varying(60) ,
  ca_street_type   CHARACTER(15)    ,
  ca_suite_number  CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
  PARTITION P1 VALUES LESS THAN(100),
  PARTITION P2 VALUES LESS THAN(200),
  PARTITION P3 VALUES LESS THAN(300)
);
```

Example: Create a List Partitioned Table

```
DROP TABLE IF EXISTS data_list;
CREATE TABLE data_list(
  id int,
  time int,
  salary decimal(12,2)
)PARTITION BY LIST (time)(
  PARTITION P1 VALUES (202209),
  PARTITION P2 VALUES (202210,202208),
  PARTITION P3 VALUES (202211),
  PARTITION P4 VALUES (202212),
  PARTITION P5 VALUES (202301)
);
```

Example: Use `modify_clause` to Set Whether a Partitioned Index Is Usable

Create the local index **student_grade_index** for the partitioned table **customer_address** and set partition index names.

```
CREATE INDEX customer_address_index ON customer_address(ca_address_id) LOCAL
(
  PARTITION P1_index,
  PARTITION P2_index,
  PARTITION P3_index
);
```

Rebuild all indexes on partition **P1** in the partitioned table **customer_address**.

```
ALTER TABLE customer_address MODIFY PARTITION P1 REBUILD UNUSABLE LOCAL INDEXES;
```

Set all indexes in partition **P3** of the partitioned table **customer_address** to be unusable.

```
ALTER TABLE customer_address MODIFY PARTITION P3 UNUSABLE LOCAL INDEXES;
```

Example: Use `add_clause` to Add One or More Partitions to a Specified Partitioned Table

Add a partition to the range partitioned table **customer_address**.

```
ALTER TABLE customer_address ADD PARTITION P5 VALUES LESS THAN (500);
```

Add the following partitions to the range partitioned table **customer_address**:
[500, 600), [600, 700).

```
ALTER TABLE customer_address ADD PARTITION p6 START(500) END(700) EVERY(100);
```

Add the **MAXVALUE** partition **p7** to the range partitioned table **customer_address**.

```
ALTER TABLE customer_address ADD PARTITION p7 END(MAXVALUE);
```

Add partition **P6** to a list partitioned table.

```
ALTER TABLE data_list ADD PARTITION P6 VALUES (202302,202303);
```

Example: Use `split_clause` to Split a Partition into Multiple Partitions

Split partition **P7** in the range partitioned table **customer_address** at **800**.

```
ALTER TABLE customer_address SPLIT PARTITION P7 AT(800) INTO (PARTITION P6a,PARTITION P6b);
```

Split the partition at **400** in the range partitioned table **customer_address** into multiple partitions.

```
ALTER TABLE customer_address SPLIT PARTITION FOR(400) INTO(PARTITION p_part START(300) END(500) EVERY(100));
```

Split partition **P2** in the list partitioned table **data_list** into two partitions: **p2a** and **p2b**.

```
ALTER TABLE data_list SPLIT PARTITION P2 VALUES(202210) INTO (PARTITION p2a,PARTITION p2b);
```

Example: Use `exchange_clause` to Migrate Data from a Common Table to a Specified Partition

The following example demonstrates how to migrate data from table **math_grade** to partition **math** in partitioned table **student_grade**.

1. Create the partitioned table **student_grade**.

```
CREATE TABLE student_grade (  
    stu_name    char(5),  
    stu_no     integer,  
    grade      integer,  
    subject    varchar(30)  
)  
PARTITION BY LIST(subject)  
(  
    PARTITION gym VALUES('gymnastics'),  
    PARTITION phys VALUES('physics'),  
    PARTITION history VALUES('history'),  
    PARTITION math VALUES('math')  
);
```

2. Add data to the partitioned table **student_grade**.

```
INSERT INTO student_grade VALUES  
('Ann', 20220101, 75, 'gymnastics');
```



```
('Jack', 20220103, 60, 'math'),  
('Anna', 20220108, 56, 'history'),  
('Jann', 20220107, 82, 'physics'),  
('Molly', 20220104, 91, 'physics'),  
('Sam', 20220105, 72, 'math');
```

3. Query the records of partition **math** in **student_grade**.

```
SELECT * FROM student_grade PARTITION (math);
```

The query result is as follows:

stu_name	stu_no	grade	subject
Jack	20220103	60	math
Sam	20220105	72	math

(2 rows)

4. Create an ordinary table **math_grade** that matches the definition of the partitioned table **student_grade**.

```
CREATE TABLE math_grade  
(  
    stu_name    char(5),  
    stu_no      integer,  
    grade       integer,  
    subject     varchar(30)  
);
```

5. Insert data to the **math_grade** table. The data in the **student_grade** partitioned table conforms to the partition rule of partition **math**.

```
INSERT INTO math_grade VALUES  
('Ann', 20220101, 75, 'math'),  
('Jack', 20220103, 60, 'math'),  
('Anna', 20220108, 56, 'math'),  
('Jann', 20220107, 82, 'math');
```

6. Migrate data from the ordinary table **math_grade** to partition **math** in the partitioned table **student_grade**.

```
ALTER TABLE student_grade EXCHANGE PARTITION (math) WITH TABLE math_grade;
```

7. Query the **student_grade** partitioned table. The results show that data in the **math_grade** table has exchanged with data in the **math** partition of the **student_grade** table.

```
SELECT * FROM student_grade PARTITION (math);  
stu_name | stu_no | grade | subject  
-----+-----+-----+-----  
Anna    | 20220108 | 56 | math  
Jack    | 20220103 | 60 | math  
Ann     | 20220101 | 75 | math  
Jann    | 20220107 | 82 | math  
(4 rows)
```

8. Query the **math_grade** table. The result shows that the data stored in the **math** partition of the **student_grade** partitioned table has been exchanged to the **math_grade** table.

```
SELECT * FROM math_grade;  
stu_name | stu_no | grade | subject  
-----+-----+-----+-----  
Jack    | 20220103 | 60 | math  
Sam     | 20220105 | 72 | math  
(2 rows)
```

Example: Use `truncate_partitioned_clause` to Clear Data in a Table Partition

Clear the **p1** partition of the **customer_address** table.

```
ALTER TABLE customer_address TRUNCATE PARTITION p1;
```

Examples: Enabling and Disabling ROW MOVEMENT

If **ROW MOVEMENT** is enabled, the data of a row is moved to a new partition when the partition key of the row is updated to a new value and the new value belongs to another partition.

If a partition key is frequently updated, enabling **ROW MOVEMENT** may cause extra I/O overhead. To enable it, evaluate service requirements first.

Create a table partitioned by date and enable **ROW MOVEMENT**.

```
DROP TABLE IF EXISTS sale_data;
CREATE TABLE sales_data (
  sale_id INT,
  product_name VARCHAR(100),
  sale_date DATE,
  amount DECIMAL(10, 2)
)
PARTITION BY RANGE (sale_date) (
  PARTITION p_2023q1 VALUES LESS THAN ('2023-04-01'),
  PARTITION p_2023q2 VALUES LESS THAN ('2023-07-01'),
  PARTITION p_2023q3 VALUES LESS THAN ('2023-10-01'),
  PARTITION p_2023q4 VALUES LESS THAN ('2024-01-01')
)
ENABLE ROW MOVEMENT;
```

Insert test data.

```
INSERT INTO sales_data (sale_id, product_name, sale_date, amount)
VALUES
(1, 'Product A', '2023-02-15', 1000.00), -- It belongs to p_2023q1.
(2, 'Product B', '2023-05-20', 1500.00); -- It belongs to p_2023q2.
```

Query the data distribution of each partition.

```
SELECT tableoid::regclass AS partition, * FROM sales_data ORDER BY sale_id;
```

partition	sale_id	product_name	sale_date	amount
2147617615	1	Product A	2023-02-15 00:00:00	1000.00
2147617616	2	Product B	2023-05-20 00:00:00	1500.00
(2 rows)				

Update the partition key (which is moved from **p_2023q1** to **p_2023q2**).

```
UPDATE sales_data SET sale_date = '2023-05-01' WHERE sale_id = 1;
```

Query the data distribution of each partition again to confirm that the row has been moved.

```
SELECT tableoid::regclass AS partition, * FROM sales_data ORDER BY sale_id;
```

partition	sale_id	product_name	sale_date	amount
2147617616	1	Product A	2023-05-01 00:00:00	1000.00
2147617616	2	Product B	2023-05-20 00:00:00	1500.00
(2 rows)				

Disable **ROW MOVEMENT**.

```
ALTER TABLE sales_data DISABLE ROW MOVEMENT;
```

Modify the partition key again (which is move from **p_2023q2** back to **p_2023q2**).

```
UPDATE sales_data SET sale_date = '2023-02-15' WHERE sale_id = 1;
```

View the command output. It indicates that **ROW MOVEMENT** is disabled and the partition key cannot be updated.

```
gaussdb> UPDATE sales_data SET sale_date = '2023-02-15' WHERE sale_id = 1;  
ERROR:  dn_6003_6004: fail to update partitioned table "sales_data"  
DETAIL:  disable row movement
```

Example: Use merge_clause to Merge Multiple Partitions into One

Merge partitions **P2** and **P3** in the range partitioned table **customer_address** into one.

```
ALTER TABLE customer_address MERGE PARTITIONS P2, P3 INTO PARTITION P_M;
```

Example: Use drop_clause to Delete a Specified Partition From a Partitioned Table

Delete partitions **P6a** and **P6b** from partitioned table **customer_address**.

```
ALTER TABLE customer_address DROP PARTITION P6a, P6b;
```

Helpful Links

[CREATE TABLE PARTITION](#) and [DROP TABLE](#)

FAQs

- Practice: [Creating and Managing DWS Partitioned Tables](#)
- Practice: [Case: Reconstructing Partitioned Tables](#)
- ["inserted partition key does not map to any table partition" Is Reported When Data Is Inserted into a Partitioned Table](#)
- [Error upper boundary of adding partition MUST overtop last existing partition Is Reported When a New Partition Is Added to a Range Partitioned Table](#)

13.27 ALTER TEXT SEARCH CONFIGURATION

Function

This syntax modifies the definition of a text search configuration.

Users can update the mapping from string type to dictionary, change the configuration's name or owner, and adjust the search configuration parameters.

Precautions

- If a search configuration is referenced (to create an index), users are not allowed to modify it.
- To use **ALTER TEXT SEARCH CONFIGURATION**, you must be the owner of the configuration.

Syntax

- Add a mapping for text search.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```
- Modify the text search configuration dictionary.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary;
```
- Change the text search configuration mapping.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```
- Change the text search configuration dictionary.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING REPLACE old_dictionary WITH new_dictionary;
```
- Delete the mapping of the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name  
  DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ];
```
- Change the owner of the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner;
```
- Rename the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name;
```
- Rename the text search configuration schema.

```
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema;
```
- Modify the attributes of the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name SET ( { configuration_option = value } [, ...] );
```
- Reset the text search configuration attributes.

```
ALTER TEXT SEARCH CONFIGURATION name RESET ( {configuration_option} [, ...] );
```

NOTE

- The **ADD MAPPING FOR** form installs a list of dictionaries to be consulted for the specified token types; an error will be generated if there is already a mapping for any of the token types.
- The **ALTER MAPPING FOR** form removes existing mapping for those token types and then adds specified mappings.
- **ALTER MAPPING REPLACE ... WITH ...** and **ALTER MAPPING FOR... REPLACE ... WITH ...** options replace **old_dictionary** with **new_dictionary**. Note that only when **pg_ts_config_map** has tuples corresponding to **maptokentype** and **old_dictionary**, the update will succeed. If the update fails, no messages are returned.
- The **DROP MAPPING FOR** form deletes all dictionaries for the specified token types in the text search configuration. If **IF EXISTS** is not specified and the string type mapping specified by **DROP MAPPING FOR** does not exist in text search configuration, an error will occur in database.

Parameter Description

Table 13-55 ALTER TEXT SEARCH CONFIGURATION parameters

Parameter	Description	Value Range
name	Specifies the name of the text search configuration to be modified, which can include a schema.	-

Parameter	Description	Value Range
token_type	Specifies the name of the token type associated with the configured syntax parser.	For details, see Parsers .
dictionary_name	Specifies the name of a text search dictionary to be consulted for the specified token types. If multiple dictionaries are listed, they are consulted in the specified order.	-
old_dictionary	Specifies the name of the text search dictionary to be replaced in the mapping.	-
new_dictionary	Specifies the name of a text search dictionary to be substituted for old_dictionary .	-
new_owner	Specifies the new owner of the text search configuration.	Valid username.
new_name	Specifies the new name of the text search configuration.	A string compliant with the identifier naming rules .
new_schema	Specifies the new schema name of the text search configuration.	Valid schema name.
configuration_option	Specifies the text search configuration item.	For details, see CREATE TEXT SEARCH CONFIGURATION .
value	Specifies the value of text search configuration option.	-

Examples

Create a text search configuration named **ngram1**.

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS ngram1;  
CREATE TEXT SEARCH CONFIGURATION ngram1 (parser=ngram) WITH (gram_size = 2, grapsymbol_ignore  
= false);
```

Add a mapping to the text search type **ngram1**.

```
ALTER TEXT SEARCH CONFIGURATION ngram1 ADD MAPPING FOR multisymbol WITH simple;
```

Change the owner of text search configuration.

```
CREATE USER joe password '{Password}';  
ALTER TEXT SEARCH CONFIGURATION ngram1 OWNER TO joe;
```

Change the schema of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION ngram1 SET SCHEMA joe;
```

Rename a text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION joe.ngram1 RENAME TO ngram_1;
```

Delete type mapping.

```
ALTER TEXT SEARCH CONFIGURATION joe.ngram_1 DROP MAPPING IF EXISTS FOR multisymbol;
```

Create a text search configuration.

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS english_1;  
CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
```

Add a mapping for the text search. (Convert regular words to lowercase and performs stemming.)

```
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR word WITH simple,english_stem;
```

Add a mapping for the text search. (Stem email addresses in both English and French.)

```
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR email WITH english_stem,  
french_stem;
```

Modify the mapping of the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION english_1 ALTER MAPPING REPLACE french_stem with german_stem;
```

Query text search configurations.

```
SELECT b.cfname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname FROM pg_ts_config_map  
a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND a.mapdict=c.oid AND b.cfname='english_1'  
ORDER BY 1,2,3,4,5;
```

cfname	maptokentype	mapseqno	mapdict	dictname
english_1	2	1	3765	simple
english_1	2	2	14505	english_stem
english_1	4	1	14505	english_stem
english_1	4	2	14511	german_stem
(4 rows)				

Links

[CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION](#)

13.28 ALTER TEXT SEARCH DICTIONARY

Function

Modifies the definition of a full-text retrieval dictionary, including its parameters, name, owner, and schema.

Precautions

- **ALTER** is not supported by predefined dictionaries.

- Only the owner of a dictionary can do **ALTER** to the dictionary. System administrators have this permission by default.
- After a dictionary is created or modified, any modification to the user-defined dictionary definition file in the directory specified by **FilePath** will not affect the dictionary in the database. To make such modifications take effect in the dictionary in the database, run the **ALTER TEXT SEARCH DICTIONARY** statement to update the definition file of the dictionary.

Syntax

- Modify the dictionary definition.
`ALTER TEXT SEARCH DICTIONARY name ( option = value | option [, ...] );`
- Rename a dictionary.
`ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name;`
- Set the schema of a dictionary.
`ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema;`
- Change the owner of a dictionary.
`ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner;`

Parameter Description

Table 13-56 ALTER TEXT SEARCH DICTIONARY parameters

Parameter	Description	Value Range
name	Specifies the name of the full-text search dictionary to be modified, which can include a schema.	Name of a valid full-text search dictionary.
option	Specifies the parameter name of the full-text search dictionary specific option to be modified. Each type of dictionaries has a template containing their custom parameters. Parameters function in a way irrelevant to their setting sequence. • The TEMPLATE parameter in a dictionary cannot be modified. • To specify a dictionary, specify both the dictionary definition file path (FILEPATH) and the file name (the parameter varies based on dictionary types). • The name of a dictionary definition file can contain only lowercase letters, digits, and underscores (_).	For details about the parameters, see the option parameter in Table 13-128 .

Parameter	Description	Value Range
value	Specifies the new value of a parameter. If = and <i>value</i> are omitted, the previous settings of the parameter will be deleted and the default value will be used.	Corresponding option definition.
new_name	Specifies the new name of the full-text search dictionary.	A string compliant with the identifier naming rules .
new_owner	Specifies the new owner of the full-text search dictionary.	Valid username.
new_schema	Specifies the new schema of the full-text search dictionary.	Valid schema name.

Examples

Create a full-text search dictionary **my_dict**.

```
CREATE TEXT SEARCH DICTIONARY my_dict (  
  TEMPLATE = snowball,  
  Language = english,  
  StopWords = english  
);
```

Change the **Language** setting for the full-text search dictionary **my_dict** and remove the stop word definition.

```
ALTER TEXT SEARCH DICTIONARY my_dict (Language = dutch,StopWords);
```

Update the definition of the full-text search dictionary **my_dict** without changing any content.

```
ALTER TEXT SEARCH DICTIONARY my_dict (dummy);
```

Helpful Links

[CREATE TEXT SEARCH DICTIONARY](#) and [DROP TEXT SEARCH DICTIONARY](#)

13.29 ALTER TRIGGER

Function

ALTER TRIGGER modifies the definition of a trigger.

Precautions

Only the owner of a table where a trigger is created and system administrators can run the **ALTER TRIGGER** statement.

Syntax

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_name;
```

Parameter Description

Table 13-57 ALTER TRIGGER parameters

Parameter	Description	Value Range
trigger_name	Specifies the name of the trigger to be modified.	Valid trigger name.
table_name	Specifies the name of the table where the trigger to be modified is located.	Name of the valid table where the trigger is located.
new_name	Specifies the new trigger name.	A string compliant with the identifier naming rules .

Examples

Create sample tables **test_trigger_src_tbl** and **test_trigger_des_tbl**.

```
DROP TABLE IF EXISTS test_trigger_src_tbl;  
DROP TABLE IF EXISTS test_trigger_des_tbl;  
  
CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);  
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
```

Create the **tri_insert_func()** trigger function.

```
DROP FUNCTION IF EXISTS tri_insert_func;  
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS  
$$  
  DECLARE  
  BEGIN  
    INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);  
    RETURN NEW;  
  END  
$$ LANGUAGE PLPGSQL;
```

Create the **insert_trigger** trigger.

```
CREATE TRIGGER insert_trigger  
  BEFORE INSERT ON test_trigger_src_tbl  
  FOR EACH ROW  
  EXECUTE PROCEDURE tri_insert_func();
```

Modify the **insert_trigger** trigger.

```
ALTER TRIGGER insert_trigger ON test_trigger_src_tbl RENAME TO delete_trigger_renamed;
```

Disable the **delete_trigger_renamed** trigger.

```
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER delete_trigger_renamed;
```

Disable all triggers on the **test_trigger_src_tbl** table.

```
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER ALL;
```

Helpful Links

[CREATE TRIGGER](#), [DROP TRIGGER](#), and [ALTER TABLE](#)

13.30 ALTER TYPE

Function

ALTER TYPE modifies the definition of a type.

Precautions

- Only type owners can run **ALTER TYPE**.
- To modify the schema of a type, you must also have the **CREATE** permission for the new schema.
- To change the owner, you must belong directly or indirectly to the new owner's role. You also need the **CREATE** permission on the schema of the type.
- System administrators can change the ownership of any type as needed.
- To add an attribute or modify the type of an attribute, you must have the **USAGE** permission on the attribute type.

Syntax

- **Modify a type.**

```
ALTER TYPE name action [, ... ];  
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER };  
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ];  
ALTER TYPE name RENAME TO new_name;  
ALTER TYPE name SET SCHEMA new_schema;  
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } neighbor_enum_value ];  
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value;
```

where action is one of:

```
ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ];  
DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ];  
ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE | RESTRICT ];
```
- **Add a new attribute to a composite type.**

```
ALTER TYPE name ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ];
```
- **Delete an attribute from a composite type.**

```
ALTER TYPE name DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ];
```
- **Change the type of an attribute in a composite type.**

```
ALTER TYPE name ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE | RESTRICT ];
```

NOTE

ADD ATTRIBUTE, **DROP ATTRIBUTE**, and **ALTER ATTRIBUTE** can be combined for execution. For example, you can add multiple attributes to a command or change the types of multiple attributes.

- **Change the owner of a type.**

```
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER };
```

- Change the name of a type or the name of an attribute in a composite type.
`ALTER TYPE name RENAME TO new_name;`
`ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ];`
- Move a type to a new schema.
`ALTER TYPE name SET SCHEMA new_schema;`
- Add a new value to an enumerated type.
`ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } neighbor_enum_value ];`
- Change an enumerated value in the value list.
`ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value;`

Parameter Description

Table 13-58 ALTER TYPE parameters

Parameter	Description	Value Range
name	Specifies the name of an existing type to be modified, which can include a schema.	Valid type name.
new_name	Specifies the new name of a type.	A string.
new_owner	Specifies the username of the new owner of the type.	Valid username.
new_schema	Specifies the new schema of the type.	Valid schema name.
attribute_name	Specifies the name of the attribute to be added, changed, or deleted.	-
new_attribute_name	Specifies the new name of the property to be renamed.	-
data_type	Specifies the data type of the attribute to be added or the new type name of the attribute to be changed.	-
new_enum_value	Specifies the new tag value of the enumeration type.	A string. The value cannot be empty and contains a maximum of 64 bytes.
neighbor_enum_value	Specifies an existing enumerated value before or after which a new enumerated value will be added.	-

Parameter	Description	Value Range
existing_enum_value	Specifies the existing enumeration value to be renamed.	A string. The value cannot be empty and contains a maximum of 64 bytes.
CASCADE	Determines that the type to be modified, its associated records, and subtables that inherit the type will all be updated.	-
RESTRICT	Refuses the update if the type of the association to be updated is an updated type. RESTRICT is the default option.	-

Examples

Create an example composite type **test**, enumeration type **testdata**, and user **user_t**.

```
CREATE TYPE test AS (col1 int, col text);  
CREATE TYPE testdata AS ENUM ('create', 'modify', 'closed');  
CREATE USER user_t PASSWORD '{Password}';
```

Rename data type **test** to **test1**.

```
ALTER TYPE test RENAME TO test1;
```

Change the owner of the user-defined type **test1** to **user_t**.

```
ALTER TYPE test1 OWNER TO user_t;
```

Change the schema of the user-defined type **test1** to **user_t**.

```
ALTER TYPE test1 SET SCHEMA user_t;
```

Add a new attribute **col3** to the user-defined type **test1**.

```
ALTER TYPE user_t.test1 ADD ATTRIBUTE col3 int;
```

Add a tag value to the enumeration type **testdata**.

```
ALTER TYPE testdata ADD VALUE IF NOT EXISTS 'regress' BEFORE 'closed';
```

Rename a tag value of the enumeration type **testdata**.

```
ALTER TYPE testdata RENAME VALUE 'create' TO 'new';
```

Helpful Links

[CREATE TYPE](#) and [DROP TYPE](#)

13.31 ALTER USER

Function

ALTER USER modifies the attributes of a database user.

Precautions

Session parameters modified by **ALTER USER** apply to a specified user and take effect in the next session.

Syntax

- Modify user rights or other information.

```
ALTER USER user_name [ [ WITH ] option [ ... ] ];
```

The **option** clause is as follows:

```
{ CREATEDB | NOCREATEDB }
| { CREATEROLE | NOCREATEROLE }
| { INHERIT | NOINHERIT }
| { AUDITADMIN | NOAUDITADMIN }
| { SYSADMIN | NOSYSADMIN }
| { USEFT | NOUSEFT }
| { LOGIN | NOLOGIN }
| { REPLICATION | NOREPLICATION }
| { INDEPENDENT | NOINDEPENDENT }
| { VCADMIN | NOVCADMIN }
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |
DISABLE }
| VALID BEGIN 'timestamp'
| VALID UNTIL 'timestamp'
| RESOURCE POOL 'respool'
| USER GROUP 'groupuser'
| PERM SPACE 'spacelimit'
| TEMP SPACE 'tmpspacelimit'
| SPILL SPACE 'spillspacelimit'
| NODE GROUP logic_cluster_name
| ACCOUNT { LOCK | UNLOCK }
| PGUSER
| AUTHINFO 'authinfo'
| PASSWORD EXPIRATION period
```

- Change the user name.

```
ALTER USER user_name
  RENAME TO new_name;
```

- Change the value of a specified parameter associated with the user.

```
ALTER USER user_name
  SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```

- Reset the value of a specified parameter associated with the user.

```
ALTER USER user_name
  RESET { configuration_parameter | ALL };
```

Parameter Description

Table 13-59 ALTER USER parameters

Parameter	Description	Value Range/Default Value
user_name	Specifies the username to be changed.	Valid username.
new_name	Specifies the new name of a user.	A string compliant with the identifier naming rules .
CREATEDB NOCREATEDB	Controls whether a new user can create a database. A new user does not have the permission to create a database by default.	Default value: NOCREATEDB
CREATEROLE NOCREATEROLE	Controls whether a user can create new users or roles using CREATE ROLE and CREATE USER . A user with the CREATEROLE permission can also modify and delete other users.	Default value: NOCREATEROLE
INHERIT NOINHERIT	Controls whether a user can inherit the permissions of its group. You are not advised to use it.	-
AUDITADMIN NOAUDITADMIN	Defines whether a user has the audit administrator attribute.	Default value: NOAUDITADMIN
SYSADMIN NOSYSADMIN	Controls whether a user is a system administrator. A user with the SYSADMIN attribute has the highest permission in the system.	Default value: NOSYSADMIN
USEFT NOUSEFT	Controls whether a user can perform operations on foreign tables, such as creating, deleting, modifying, and reading/writing foreign tables. The default value is NOUSEFT , indicating that new users cannot perform operations on foreign tables by default. The new user does not have the permission to perform operations on foreign tables.	-

Parameter	Description	Value Range/Default Value
LOGIN NOLOGIN	Controls whether a user can log in to the database. Only users with the LOGIN attribute can log in to the database.	Default value: LOGIN
REPLICATION NOREPLICATION	Controls whether a user is allowed to initiate streaming replication or put the system in and out of backup mode. A user with the REPLICATION attribute is only used for replication.	Default value: NOREPLICATION
INDEPENDENT NOINDEPENDENT	Defines private and independent users. For a user with the INDEPENDENT attribute, administrators' rights to control and access this role are separated. Specific rules are as follows: • Administrators have no rights to add, delete, query, modify, copy, or authorize the corresponding table objects without the authorization from the INDEPENDENT user. • Without the authorization of the INDEPENDENT user, the administrator has no right to modify its inheritance relationship. • The administrator does not have the permission to change the owner of the table object of an INDEPENDENT user. • The administrator does not have the permission to remove the INDEPENDENT attribute of an INDEPENDENT user. • The administrator does not have the permission to change the database password of an INDEPENDENT user. An INDEPENDENT must manage its own password. If the password is lost, it cannot be reset. • The SYSADMIN attribute of a user cannot be changed to the INDEPENDENT attribute.	-

Parameter	Description	Value Range/Default Value
VCADMIN NOVCADMIN	Defines a logical cluster administrator. A logical cluster administrator has the following more permissions than common users: • Create, modify, and delete resource pools in the associated logical cluster. • Grant the access permission for the associated logical cluster to other users or roles, or reclaim the access permission from those users or roles.	-
CONNECTION LIMIT	Specifies the number of concurrent connections that can be used by a user on a single CN. To ensure that a cluster can run normally, the minimum value of CONNECTION LIMIT is the number of CNs in the cluster, because when you run ANALYZE on a CN of the cluster, other CNs will connect to the running CN for metadata synchronization. For example, if there are three CNs in a cluster, set CONNECTION LIMIT to 3 or a larger value.	Value range: integers no less than -1. Default value: -1, indicating there is no limit.

Parameter	Description	Value Range/Default Value
ENCRYPTED UNENCRYPTED	Determines whether the password stored in the system will be encrypted. (If neither is specified, the password status is determined by password_encryption_type.) According to product security requirements, the password must be stored encrypted. Therefore, UNENCRYPTED is forbidden in DWS. If the password is encrypted using SHA256, it will be stored as it is, regardless of whether it is specified as ENCRYPTED or UNENCRYPTED. This is because the system cannot decrypt the specified encrypted password. This allows reloading of the encrypted password during dump/restore. • password: login password. • DISABLE: Users can change their own passwords unless the passwords are disabled. You can use DISABLE to disable the password of a user. After the password of a user is disabled, the password will be deleted from the system. The user can connect to the database only through external authentication, for example, IAM authentication, Kerberos authentication, or LDAP authentication. Only administrators can enable or disable passwords. Common users cannot disable the password of an initial user. To enable a password, run ALTER USER and specify the password.	The password must: • Contain at least eight characters. • Differ from the username or it in reverse order. • Contain at least three types of the following characters: uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), and special characters such as ~!@#\$%^&*()-_+=\ {}];;<.>/? (If you use a character that is not in the specified range, an alarm is generated, but the character can still be created.)
VALID BEGIN	Sets the timestamp when a user takes effect. If this clause is omitted, there is no restriction on when the user takes effect.	-
VALID UNTIL	Sets an expiration timestamp for an account. After the specified time, the account cannot be used. If this clause is omitted, it means the account will never expire	-
RESOURCE POOL	Sets the name of the resource pool used by a user. The name is from the system catalog pg_resource_pool .	-

Parameter	Description	Value Range/Default Value
USER GROUP 'groupuser'	Creates a subuser.	-
PERM SPACE	Sets the storage space of the user permanent table. space_limit : specifies the upper limit of the storage space of the permanent table.	space_limit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
TEMP SPACE	Sets the storage space of the user temporary table. tmpspacelimit : specifies the storage space limit of the temporary table.	tmpspacelimit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
SPILL SPACE	Sets the operator disk flushing space of the user. spillspacelimit : specifies the operator spilling space limit.	spillspacelimit : A string consists of an integer and unit. The unit can be K/M/G/T/P. 0 indicates no limits.
NODE GROUP	Specifies the name of the logical cluster associated with a user. If the name contains uppercase characters or special characters, enclose the name with double quotation marks.	-
ACCOUNT LOCK ACCOUNT UNLOCK	Locks or unlocks an account. • ACCOUNT LOCK: locks an account to forbid login to databases. • ACCOUNT UNLOCK: unlocks an account to allow login to databases.	-
PGUSER	This attribute is used to be compatible with open-source Postgres communication. An open-source Postgres client interface (Postgres 9.2.19 is recommended) can use a database user having this attribute to connect to the database. PGUSER of a user cannot be modified in the current version. For details, see PGUSER Parameter Setting Suggestions .	-

Parameter	Description	Value Range/Default Value
AUTHINFO 'authinfo'	This attribute is used to specify the user authentication type. authinfo is the description character string, which is case sensitive. Only the LDAP type is supported. Its description character string is ldap. LDAP authentication is an external authentication mode. Therefore, PASSWORD DISABLE must be specified. • Additional information about LDAP authentication can be added to authinfo, for example, fulluser in LDAP authentication, which is equivalent to ldapprefix+username+ldapsuffix. If the content of authinfo is ldap, the user authentication type is LDAP. In this case, the ldapprefix and ldapsuffix information is provided by the corresponding record in the pg_hba.conf file. • When executing ALTER ROLE, users are not allowed to change the authentication type. Only LDAP users are allowed to modify LDAP attributes.	-
PASSWORD EXPIRATION period	Number of days before the login password of the role expires. The user needs to change the password in time before the login password expires. If the login password expires, the user cannot log in to the system. In this case, the user needs to ask the administrator to set a new login password.	The value is an integer ranging from -1 to 999. • Default value -1 indicates that there is no restriction. • 0 indicates that the login password expires immediately.

Parameter	Description	Value Range/Default Value
SET configuration_ parameter	Sets the session parameters of a user. The session parameters modified by running the ALTER USER command apply only to the specified user and are valid in the next session started by the user. • DEFAULT clears the value of configuration_parameter. The value of the configuration_parameter parameter inherits the default value of the new session of the role. • FROM CURRENT uses the value of configuration_parameter of the current session.	For details about the values of configuration_parameter and value , see SET .
RESET configuration_ parameter/ALL	The effect of clearing the configuration_parameter value is the same as setting it to DEFAULT. ALL indicates that all parameter values are cleared.	-

PGUSER Parameter Setting Suggestions

The **PGUSER** attribute is for compatibility with the connection process. It does not handle incompatibilities due to kernel differences between the product and Postgres.

Users with the **PGUSER** attribute have a different authentication method. Open-source clients might reveal the **PGUSER** attribute through error messages. You are advised to use the client provided with the product. Example:

```
# normaluser is a user that does not have the PGUSER attribute. psql is the Postgres client tool.
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

# pguser is a user having the PGUSER attribute.
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

Examples

Create example user **u1**.

```
DROP USER IF EXISTS u1 CASCADE;
CREATE USER u1 PASSWORD '{Password}';
```

Change the login password of user **u1**.

```
ALTER USER u1 IDENTIFIED BY '{new_Password}' REPLACE '{Password}';
```

Add the **CREATEROLE** permission to user **u1**.

```
ALTER USER u1 CREATEROLE;
```

Set the maximum number of database connections of user **u1** to **10**.

```
ALTER USER u1 with CONNECTION LIMIT 10;
```

Set the **enable_seqscan** parameter of the session associated with user **u1** to **on**. The setting takes effect in the next session.

```
ALTER USER u1 SET enable_seqscan TO on;
```

Reset the **enable_seqscan** parameter for user **u1**.

```
ALTER USER u1 RESET enable_seqscan;
```

Lock the **u1** account.

```
ALTER USER u1 ACCOUNT LOCK;
```

Links

[CREATE ROLE](#), [CREATE USER](#), and [DROP USER](#)

FAQs

[How Do I Unlock an Account?](#)

[An Error Message Is Displayed When a Common User Executes the Statement for Creating or Deleting a GDS or OBS Foreign Table, Indicating that the User Does Not Have the Permission or the Permission Is Insufficient](#)

13.32 ALTER VIEW

Function

ALTER VIEW modifies all auxiliary attributes of a view. (To modify the query definition of a view, use **CREATE OR REPLACE VIEW**.)

Precautions

- Only the view owner can run **ALTER VIEW**.
- To change a view's schema, you must also have the CREATE permission on the new schema.
- To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the view's schema.
- An administrator can change the owner relationship of any view.

Syntax

- Set the default value of the view column.

```
ALTER VIEW [ IF EXISTS ] view_name  
  ALTER [ COLUMN ] column_name SET DEFAULT expression;
```
- Remove the default value of the view column.

```
ALTER VIEW [ IF EXISTS ] view_name  
  ALTER [ COLUMN ] column_name DROP DEFAULT;
```
- Change the owner of a view.

```
ALTER VIEW [ IF EXISTS ] view_name  
OWNER TO new_owner;
```

- Renames a view. The new view name can be prefixed with the schema name of the original view. The schema name cannot be changed at the same time.

```
ALTER VIEW [ IF EXISTS ] view_name  
RENAME TO new_name;  
ALTER VIEW [ IF EXISTS ] view_name  
RENAME TO schema.new_name;
```

- Set the schema of the view.

```
ALTER VIEW [ IF EXISTS ] view_name  
SET SCHEMA new_schema;
```

- Set the options of the view.

```
ALTER VIEW [ IF EXISTS ] view_name  
SET ( { view_option_name [ = view_option_value ] } [, ... ] );
```

- Reset the options of the view.

```
ALTER VIEW [ IF EXISTS ] view_name  
RESET ( view_option_name [, ... ] );
```

- Rebuild the current view and its lower-layer and upper-layer dependent views.

```
ALTER VIEW [ IF EXISTS ] view_name  
REBUILD;
```

- Rebuild the current view and its lower-layer dependent views.

```
ALTER VIEW [ IF EXISTS ] ONLY view_name  
REBUILD;
```

Parameter Description

Table 13-60 ALTER VIEW parameters

Parameter	Description	Value Range
IF EXISTS	Sends a prompt instead of an error if the view does not exist.	-
view_name	Specifies the name of the view to be modified, which can include a schema.	Valid view name.
column_name	Indicates an optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.	Valid column name.
SET/DROP DEFAULT	Sets or deletes the default value of a column. Currently, this parameter does not take effect.	-
new_owner	Specifies the new owner of the view.	Valid username.
new_name	Specifies the new view name.	A string compliant with the identifier naming rules .
new_schema	Specifies the new schema of the view.	Valid schema name.

Parameter	Description	Value Range
view_option_name [= view_option_value]	This clause specifies optional parameters for a view. Currently, the only parameter supported by view_option_name is security_barrier , which should be enabled when a view is intended to provide row-level security.	security_barrier is of the boolean type. The options include: • TRUE • FALSE
REBUILD	This clause is used for view decoupling. You can use the saved original statement to rebuild views and restore the dependencies. In 9.1.0.200 and later cluster versions, the view decoupling function does not require view_independent to be enabled. If a view becomes invalid, it can simply be rebuilt.	For how to use REBUILD , see REBUILD Usage Description .
ONLY	Limits view rebuilding to only the selected view and its dependencies. This function does not require view_independent to be enabled. If a view becomes invalid, it can simply be rebuilt.	-

REBUILD Usage Description

- View rebuilding starts from the current view and updates all associated backward views. If the forward views on which the current view depends are also unavailable, automatic rebuilding is triggered.
- The temporary tables and views that have dependency relationships cannot be decoupled and dropped. However, you can perform the REBUILD operation on temporary views that do not have dependency relationships.
- View schema names and view names can be modified. The names of rebuilt view schemas or views are re-created based on the latest name, but the query operation retains the original definition.
- Only fields of the character, number, and time types in the base table can be modified. Only fields of the character, number, and time types in the base table can be modified. When a field is added to the base table, the view is not invalidated and the definition remains unchanged.
- Invalid views are exported as comments during backup. You need to manually restore the invalid views.

 NOTE

The upper-layer cascading views become invalid in the following scenarios:

- DROP TABLE/VIEW
- RENAME TABLE/VIEW
- ALTER TABLE DROP COLUMN
- ALTER TABLE CHANGE/ALTER COLUMN TYPE
- ALTER TABLE CHANGE/ALTER COLUMN NAME
- ALTER TABLE/VIEW NAMESPACE
- ALTER TABLE/VIEW RENAME

Examples

Create the sample view **myview**.

```
CREATE OR REPLACE VIEW myview AS  
SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';
```

Change the view name.

```
ALTER VIEW myview RENAME TO product_view;
```

Change the schema of a view.

```
ALTER VIEW product_view SET schema public;
```

Rebuild a view.

```
ALTER VIEW public.product_view REBUILD;
```

Rebuild a dependent view.

```
ALTER VIEW ONLY public.product_view REBUILD;
```

Helpful Links

[CREATE VIEW](#) and [DROP VIEW](#)

13.33 CLEAN CONNECTION

Function

CLEAN CONNECTION clears database connections when a database is abnormal. You may use this statement to delete a specific user's connections to a specified database.

Precautions

You must specify either **dbname** or **username**.

Syntax

```
CLEAN CONNECTION  
TO { COORDINATOR ( nodename [, ... ] ) | NODE ( nodename [, ... ] ) } ALL [ CHECK ] [ FORCE ] }  
[ FOR DATABASE dbname ]  
[ TO USER username ];
```


Parameter Description

Table 13-61 CLEAN CONNECTION parameters

Parameter	Description	Value Range
CHECK	This parameter can be specified only when the node list is specified as TO ALL . Setting this parameter will check whether a database is accessed by other sessions before its connections are cleared. If any sessions are detected before DROP DATABASE is executed, an error will be reported and the database will not be deleted.	-
FORCE	This parameter can be specified only when the node list is specified as TO ALL . Setting this parameter will send SIGTERM signals to all the threads related to the specified dbname and username and forcibly shut them down.	-
COORDINATOR (nodename [, ...]) NODE (nodename [, ...]) ALL	Deletes connections on a specified node. There are three scenarios: • Deletes connections to a specified CN. • Deletes connections to a specified DN. • Deletes connections to all CNs and DNs.	nodename is an existing node name.
dbname	Deletes connections to a specific database. If this parameter is not specified, connections to all databases will be deleted.	Valid database name.
username	Deletes connections of a specific user. If this parameter is not specified, connections of all users will be deleted.	Valid username.

Examples

Clean connections to nodes dn1 and dn2 for the **template1** database.

```
CLEAN CONNECTION TO NODE (dn1,dn2) FOR DATABASE template1;
```

Clean user **jack**'s connections to dn1.

```
CLEAN CONNECTION TO NODE (dn1) TO USER jack;
```

Delete all connections to the gaussdb database.

```
CLEAN CONNECTION TO ALL FORCE FOR DATABASE gaussdb;
```

13.34 CLOSE

Function

CLOSE frees the resources associated with a cursor.

Precautions

- After a cursor is closed, no subsequent operations are allowed on it.
- Cursors that are no longer used should be closed.
- Each non-held open cursor should be implicitly closed after the transaction that created the cursor is terminated with **COMMIT** or **ROLLBACK**.
- A holdable cursor is implicitly closed when the transaction that creates the cursor exits through **ROLLBACK**.
- If the transaction that creates a cursor is committed, the cursor remains open until an explicit **CLOSE** operation is performed or the client is disconnected.
- DWS does not have an explicit **OPEN** cursor statement. A cursor opens when you define it with the **CURSOR** command. You can see all available cursors by querying the **pg_cursors** system view.

Syntax

```
CLOSE { cursor_name | ALL };
```

Parameter Description

Table 13-62 CLOSE parameters

Parameter	Description	Value Range
cursor_name	Specifies the name of the cursor to be closed.	Name of an existing cursor.
ALL	Closes all open cursors.	-

Examples

Close a cursor.

```
CLOSE cursor1;
```

Links

[FETCH](#) and [MOVE](#)

13.35 CLUSTER

Function

This syntax clusters tables by their indexes.

- **CLUSTER** clusters the table specified by table name based on the index specified by index name. The index specified by index name must have been defined in the specified table.
- When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order.
- When a table is clustered, DWS records which index the table was clustered by. The form **CLUSTER table_name** reclusters the table using the same index used earlier. You can also use **SET WITHOUT CLUSTER** of **CLUSTER** or **ALTER TABLE** to set indexes for subsequent clustering operations or clear any previous settings.
- **CLUSTER** without any parameters reclusters all tables that have been clustered in the database owned by the current user. If **CLUSTER** is invoked by the system administrator, it indicates all tables that have been clustered.
- When a table is being clustered, an **ACCESS EXCLUSIVE** lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the clustering is complete.

Precautions

- **CLUSTER** cannot be executed in transactions.
- **CLUSTER** cannot be performed on HDFS tables.
- **CLUSTER** cannot be performed on global temporary tables.
- Performing a **CLUSTER** operation on a table triggers a rebuilding process where data is dumped into a new data file. Once the process is complete, the original file is deleted. Keep in mind that if the table is large, this process can consume a significant amount of disk space. When the disk space is insufficient, exercise caution when performing the **CLUSTER** operation on large tables to prevent the cluster from being read-only.
- In the storage-compute decoupled architecture, a message is displayed indicating that operations on the entire cluster database are not supported.
- Only row-store B-tree indexes support **CLUSTER**.
- In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using **CLUSTER**. If you request a range of index values from a table or if an index value matches multiple rows, **CLUSTER**

helps. It does this by storing related rows on the same page. This saves disk access time and speeds up queries.

- During clustering, the system creates a temporary copy of the table in the index sequence and a temporary copy of each index in the table. The disk must have enough space for both the table and its index.
- CLUSTER stores clustering information. You can manually cluster tables initially, then set up a periodic maintenance script. This allows tables to be automatically clustered regularly.
- Because the optimizer records statistics about the ordering of tables, it is advisable to run **ANALYZE** on the newly clustered table. Otherwise, the optimizer might select an incorrect query plan.

Syntax

- Cluster a table.
`CLUSTER [ VERBOSE ] table_name [ USING index_name ];`
- Cluster a partition.
`CLUSTER [ VERBOSE ] table_name PARTITION ( partition_name ) [ USING index_name ];`
- Cluster the table that has previously been clustered.
`CLUSTER [ VERBOSE ];`

Parameter Description

Table 13-63 CLUSTER parameters

Parameter	Description	Value Range
VERBOSE	Enables the display of progress messages.	-
table_name	Specifies the name of the table.	Name of an existing table.
index_name	Specifies the index name.	Name of an existing index.
partition_name	Specifies the partition name.	Name of an existing partition.

Examples

Create a partitioned table.

```
CREATE TABLE tpcds.inventory_p1
(
  INV_DATE_SK      INTEGER      NOT NULL,
  INV_ITEM_SK      INTEGER      NOT NULL,
  INV_WAREHOUSE_SK INTEGER      NOT NULL,
  INV_QUANTITY_ON_HAND INTEGER
)
DISTRIBUTE BY HASH(INV_ITEM_SK)
PARTITION BY RANGE(INV_DATE_SK)
(
  PARTITION P1 VALUES LESS THAN(2451179),
  PARTITION P2 VALUES LESS THAN(2451544),
```

```
PARTITION P3 VALUES LESS THAN(2451910),  
PARTITION P4 VALUES LESS THAN(2452275),  
PARTITION P5 VALUES LESS THAN(2452640),  
PARTITION P6 VALUES LESS THAN(2453005),  
PARTITION P7 VALUES LESS THAN(MAXVALUE)  
);
```

Create an index named **ds_inventory_p1_index1**.

```
CREATE INDEX ds_inventory_p1_index1 ON tpccs.inventory_p1 (INV_ITEM_SK) LOCAL;
```

Cluster the **tpccs.inventory_p1** table.

```
CLUSTER tpccs.inventory_p1 USING ds_inventory_p1_index1;
```

Cluster the **p3** partition.

```
CLUSTER tpccs.inventory_p1 PARTITION (p3) USING ds_inventory_p1_index1;
```

Cluster the tables that can be clustered in the database.

```
CLUSTER;
```

13.36 COMMENT

Function

COMMENT defines or changes the comment of an object.

Precautions

- Only one comment string is stored for each object. To modify a comment, issue a new **COMMENT** command for the same object. To remove a comment, write **NULL** at the location of the text string. Comments are automatically deleted when their objects are deleted.
- There is no security for viewing comments. Any user with database access can see all object comments. Comments on shared items like databases, roles, and tablespaces are visible to any user connected to any database in the cluster. Avoid adding sensitive security information to comments.
- For most objects, only the owner of the object can set comments. A role has no owner. So, only a system admin can use the **COMMENT ON ROLE** command for system admin roles. Roles with the **CREATE ROLE** permission can add comments to non-admin roles. Administrators can comment on anything.

Syntax

```
COMMENT ON  
{  
  AGGREGATE agg_name (agg_type [, ...] ) |  
  CAST (source_type AS target_type) |  
  COLLATION object_name |  
  COLUMN { table_name.column_name | view_name.column_name } |  
  CONSTRAINT constraint_name ON table_name |  
  CONVERSION object_name |  
  DATABASE object_name |  
  DOMAIN object_name |  
  EXTENSION object_name |  
  FOREIGN DATA WRAPPER object_name |  
  FOREIGN TABLE object_name |
```

```
FUNCTION function_name ( [ {[ argmode ] [ argname ] argtype} [, ...] ] ) |  
INDEX object_name |  
LARGE OBJECT large_object_oid |  
OPERATOR operator_name (left_type, right_type) |  
OPERATOR CLASS object_name USING index_method |  
OPERATOR FAMILY object_name USING index_method |  
[ PROCEDURAL ] LANGUAGE object_name |  
ROLE object_name |  
RULE rule_name ON table_name |  
SCHEMA object_name |  
SERVER object_name |  
TABLE object_name |  
TABLESPACE object_name |  
TEXT SEARCH CONFIGURATION object_name |  
TEXT SEARCH DICTIONARY object_name |  
TEXT SEARCH PARSER object_name |  
TEXT SEARCH TEMPLATE object_name |  
TYPE object_name |  
VIEW object_name  
}  
IS 'text';
```

Parameter Description

Table 13-64 COMMENT parameters

Parameter	Description	Value Range
agg_name	Specifies the name of the aggregate function to be commented out.	-
agg_type	Specifies the data types of the aggregation function parameters.	-
source_type	Specifies the name of the source data type of the cast.	-
target_type	Specifies the name of the target data type of the cast.	-
object_name	Specifies the name of an object.	-
table_name.column_name /view_name.column_name	Specifies the column whose comment is defined or modified. You can add the table name or view name as the prefix.	-
constraint_name	Specifies the table constraints whose comment is defined or modified.	Name of an existing table constraint.
table_name	Specifies the table name.	Name of an existing table.

Parameter	Description	Value Range
function_name	Specifies the function whose comment is defined or modified.	Use an existing function name.
argmode/argname/ argtype	Specifies the schema, name, and type of the function parameters.	-
large_object_oid	Specifies the OID of the large object whose comment is defined or modified.	-
operator_name	Specifies the name of the operator.	-
left_type/right_type	Specifies the data type of the operation parameter, which can be specified by a pattern. Write NONE for the missing argument of a prefix or postfix operator.	-
text	Specifies the new comment.	A string.

Examples

Create a sample table **CUSTOMER** and add comments to the **customer.c_customer_sk** column of the table.

```
CREATE TABLE IF NOT EXISTS CUSTOMER
(
  C_CUSTKEY BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY ,
  C_NAME   VARCHAR(25) ,
  C_ADDRESS VARCHAR(40) ,
  C_NATIONKEY INT ,
  C_PHONE   CHAR(15) ,
  C_ACCTBAL  DECIMAL(15,2)
)
DISTRIBUTE BY HASH(C_CUSTKEY);

COMMENT ON COLUMN customer.c_customer_sk IS 'Primary key of customer demographics table';
```

Create the **customer_details_view** view and add comments to the view.

```
CREATE OR REPLACE VIEW customer_details_view AS SELECT * FROM CUSTOMER;
COMMENT ON VIEW customer_details_view_v2 IS 'View of customer detail';
```

Add comments to the **customer** table.

```
COMMENT ON TABLE customer IS 'This is my table';
```

13.37 CREATE BARRIER

Function

Creates a barrier for cluster nodes. The barrier can be used for data restoration.

Precautions

Before creating a barrier, ensure that **gtm_backup_barrier** and **enable_cbm_tracking** are set to **on** for CNs and DN in the cluster.

Syntax

```
CREATE BARRIER [ barrier_name ] ;
```

Parameter Description

barrier_name

(Optional) Indicates the name of a barrier.

A string compliant with the [identifier naming rules](#).

Examples

Create a barrier without specifying its name.

```
CREATE BARRIER;
```

Create a barrier named **barrier1**.

```
CREATE BARRIER 'barrier1';
```

13.38 CREATE BLOCK RULE

Function

This function creates a filtering rule, including the filtering rule name, bound client name, client IP address, user, and matching mode.

This syntax is supported only by clusters of 9.1.0.100 and later versions.

Precautions

Only a user with the database owner permission or the **gs_role_block** role permission can run **CREATE BLOCK RULE**. A system administrator has this permission by default.

Syntax

```
CREATE BLOCK RULE [ IF NOT EXISTS ] block_name  
[ [ TO user_name@'host' ] | [ TO user_name ] | [ TO 'host' ] ] |  
[ FOR UPDATE | SELECT | INSERT | DELETE | MERGE ] |  
FILTER BY
```



```
{ SQL ( 'text' ) | TEMPLATE ( template_parameter = value ) }  
[ WITH ( { with_parameter = value }, [, ... ] ) ];
```

Parameter Description

Table 13-65 CREATE BLOCK RULE parameters

Parameter	Description	Value Range
block_name	Specifies the name of the filtering rule to be created.	A string compliant with the identifier naming rules .
user_name	Queries the users to which the filtering rule applies.	A string that indicates a valid username.
host	Queries the client IP address to which the filtering rule applies.	A string that indicates a valid IP address.
SQL	Indicates the SQL statement for querying the regular expression of the filtering rule.	A string that indicates a regular expression. The length of the statement or keyword matching the regular expression cannot exceed 1,024 characters.
template_parameter	Queries a filtering rule matching template.	unique_sql_id or sql_hash . The value is a string with unique_sql_id as all digits.

Parameter	Description	Value Range
with_parameter	Indicates the parameters for querying filter rule details.	The following parameters can be set together. The query that meets any of the parameters will be filtered. • application_name: client name. • query_band • table_num: number of tables scanned by the statement. • partition_num: maximum number of partitions to be scanned by the operator. • estimate_row: estimated maximum number of table rows scanned by the operator. • resource_pool: name of the resource pool to be switched to. • max_active_num: maximum number of concurrent statements corresponding to the rule. • is_warning: whether an alarm should be generated or an error should be reported when a statement is intercepted.

Examples

Create a query filtering rule named **query_block**.

```
CREATE BLOCK RULE query_block TO user1@'192.168.x.x' FOR SELECT FILTER BY SQL('select * from table_name')WITH(application_name='gsq',query_band='test1',table_num='2',partition_num='3',estimate_row='1000',resource_pool='rsp1',max_active_num='3',is_warning='off');
```

```
CREATE BLOCK RULE query_block FILTER BY TEMPLATE(unique_sql_id='1634655172');
```

```
CREATE BLOCK RULE query_block FILTER BY TEMPLATE(sql_hash='sql_c3d119fe636b9ef439b1f96c561c74ff');
```

13.39 CREATE DATABASE

Function

CREATE DATABASE creates a database.

DWS creates a database by copying a database template. DWS has two default template databases **template0** and **template1** and a default user database **gaussdb**. By default, a new database is created based on **template1**. The

character encoding of **template1** cannot be changed. If you want to change it, you can set the **template** parameter to **template0**.

Precautions

- A user that has the **CREATEDB** permission or a sysadmin can create a database.
- **CREATE DATABASE** cannot be executed within a transaction block.
- Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Syntax

```
CREATE DATABASE database_name
[ [ WITH ] { [ OWNER [=] user_name ] |
[ TEMPLATE [=] template ] |
[ ENCODING [=] encoding ] |
[ LC_COLLATE [=] lc_collate ] |
[ LC_CTYPE [=] lc_ctype ] |
[ DBCOMPATIBILITY [=] compatibility_type ] |
[ DBCOMPATIBILITY_BEHAVIOR [=] opt_compat_behavior ] |
[ CONNECTION LIMIT [=] conlimit ] }[...];
```

Parameter Description

Table 13-66 CREATE DATABASE parameters

Parameter	Description	Value Range
database_name	Name of the database to be created	A string, which must comply with the naming convention .
user_name	Specifies the owner of the new database. By default, the owner of the database is the current user.	An existing user name.
template	Specifies the name of the template used to create a database. If it is not specified, the system uses template1 by default.	Value range: template0 or template1 . • template0: The character set encoding can be specified if template0 is used. • template1 (default value): The character set encoding cannot be specified if template1 is used.

Parameter	Description	Value Range
encoding	Specifies the character set encoding used by the new database. The value can be a string (for example, SQL_ASCII) or an integer. By default, the encoding format of the template database is used. The encoding formats of the template databases template0 and template1 vary based on OS environments by default.	Common values: GBK , UTF8 , Latin1 , and SQL_ASCII .
lc_collate	Specifies the collation order to use in the new database. For example, this parameter can be set using <code>lc_collate = 'zh_CN.gbk'</code>. The use of this parameter affects the sort order applied to strings, for example, in queries with ORDER BY, as well as the order used in indexes on text columns. The default is to use the collation order of the template database. For other restrictions, see Character Set Encoding Description.	Valid collation order
lc_ctype	Specifies the character classification to use in the new database. For example, this parameter can be set using <code>lc_ctype = 'zh_CN.gbk'</code>. The use of this parameter affects the categorization of characters, for example, lower, upper and digit. The default is to use the character classification of the template database.	Valid character classification
compatibility_type	Specifies the compatible database type. If this parameter is not specified, the default value ORA is used.	The value can be ORA , TD , or MySQL , indicating Oracle, Teradata, or MySQL database, respectively.

Parameter	Description	Value Range
opt_compat_behavior	Specifies the compatibility behavior of the database. This parameter is supported only by clusters of version 9.1.0 or later. If this parameter is not specified, the default value NO_BEHAVIOR is used, indicating no special behavior.	The value can be td_rtrim or pg_char. • td_rtrim: removes the trailing part of a variable-length character string in Teradata compatibility mode. • pg_char: converts the varchar type to the nvarchar2 type in PostgreSQL compatibility mode.
connlimit	Number of concurrent connections to a database. The default value -1 means no limit. • This limit does not apply to system administrators. • To ensure that a cluster runs properly, the minimum value of CONNECTION LIMIT is the number of CNs in the cluster. because when a cluster runs ANALYZE on a CN, other CNs will connect to the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set CONNECTION LIMIT to 3 or a larger value.	An integer greater than or equal to -1

Character Set Encoding Description

- To view the character encoding of the current database, run the **show server_encoding;** command.
- To make your database compatible with most characters, you are advised to use the UTF8 encoding when creating a database.
- The character set encoding of the new database must be compatible with the language settings (**LC_COLLATE** and **LC_CTYPE**).
- If the language is **C** (or equivalently **POSIX**), then all encoding modes are allowed, but for other language settings only the encoding mode for character sets consistent with that of the locale will work properly.
- The encoding and locale settings must match those of the template database, except when template0 is used as template. This is because other databases might contain data that does not match the specified encoding, or might

contain indexes whose sort ordering is affected by **LC_COLLATE** and **LC_CTYPE**. Copying such data would result in a database that is corrupt according to the new settings. `template0`, however, is known to not contain any data or indexes that would be affected.

- Supported encoding depends on the environment. If the message "invalid locale name" is displayed, run the **locale -a** command to check the encoding set supported by the environment.
- When the specified character encoding set is **GBK**, some uncommon Chinese characters cannot be directly used as object names. This is because when the encoding range of the second byte of GBK is between 0x40 and 0x7E, the byte encoding overlaps with the ASCII character @A-Z[\]^_`a-z{[]}. @[\]^_?{} is an operator in the database. If it is directly used as an object name, a syntax error will be reported. For example, the GBK hexadecimal code is **0x8240**, and the second byte is **0x40**, which is the same as the ASCII character @. Therefore, the character cannot be used as an object name. If you really want to use it, you can avoid this problem by adding double quotation marks when creating and accessing objects.
- In the current version, the GBK character set supports the character €, which is represented as **0x80** in hexadecimal code. You can use the € character in the GBK library, and the GBK character set of DWS is compatible with the CP936 character set. Note that the GBK character set is approximately equal to the CP936 character set, but the GBK character set does not contain the definition of the character €.

Examples

Create database **music** using GBK (the local encoding type is also GBK).

```
CREATE DATABASE music ENCODING 'GBK' template = template0;
```

Create database **music2** and specify **jim** as its owner.

```
CREATE USER jim PASSWORD "{Password}";  
CREATE DATABASE music2 OWNER jim;
```

Create database **music3** using template **template0** and specify **jim** as its owner.

```
CREATE DATABASE music3 OWNER jim TEMPLATE template0;
```

Create a database compatible with Oracle and set the connection limit to **10**.

```
CREATE DATABASE ora_compatible_db DBCOMPATIBILITY 'ORA' CONNECTION LIMIT = 10;
```

Helpful Links

[ALTER DATABASE, DROP DATABASE](#)

FAQs

[How Do I View and Set the Character Set Encoding Format of a GaussDB\(DWS\) Database?](#)

[Query Results Are Inconsistent in Oracle, Teradata, and MySQL Compatibility Modes](#)

13.40 CREATE EXCEPT RULE

Function

This syntax is used to create an exception rule set. When creating an exception rule, you can specify the rule thresholds and operations following the triggering of an exception rule. Currently, only the **abort** operation is supported.

Precautions

None

Syntax

```
CREATE EXCEPT RULE except_rule_name
WITH (
  | BLOCKTIME = VALUE,
  | CPUTIME = VALUE,
  | ELAPSEDTIME = VALUE,
  | CPUSKEWPERCENT = VALUE,
  | SPILLSIZE = VALUE,
  | BROADCASTSIZE = VALUE,
  | MEMSIZE = VALUE,
  | CPUAVGPERCENT = VALUE,
  | BANDWIDTH = VALUE,
  | ACTION = ['abort' | 'penalty']
);
```

Parameter Description

Table 13-67 CREATE EXCEPT RULE parameters

Parameter	Description	Value Range
rule_name	Name of an exception rule set.	A string, which must comply with Identifier Naming Conventions .
blocktime	Maximum duration of job queue blocking, in seconds.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX. Set the value to -1, which indicates that the value is not limited.
elapsedtime	Maximum job execution duration, in seconds.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX.
allcputime	Maximum CPU time used during job running. The unit is second.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX.

Parameter	Description	Value Range
cpuske wpercen t	Average CPU usage during job execution. The unit is percentage.	Numeric type. The value can be -1 or ranges from 1 to 100 .
cpuavgp ercent	CPU usage skew during job execution. The unit is percentage.	Numeric type. The value can be -1 or ranges from 1 to 100 .
spillsize	Maximum size of data spilled to disks during job execution. The unit is MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
broadca stsize	Maximum broadcast size of a job. The unit is MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
memsiz e	Maximum memory size used for job execution. Unit: MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .
bandwi dth	Maximum bandwidth that can be used for job execution. Unit: MB.	Numeric type. The value can be -1 or ranges from 1 to INT64_MAX .

Examples

Create exception rule set **except_rule1** and set the **blocktime** threshold to **3000 seconds**, and spilling space to **4000 MB**.

```
CREATE EXCEPT RULE except_rule1 WITH (blocktime=3000, spillsize=4000, action=abort);
```

Create an exception rule set **except_rule2** and set the **memsize** threshold to **5000 MB**. The default operation following an exception is **abort**.

```
CREATE EXCEPT RULE except_rule2 WITH (memsize=3000);
```

Create a resource pool and bind it to exception rule set **except_rule3**.

```
CREATE resource pool resource_pool_a1 WITH (except_rule='except_rule3');
```

Helpful Links

[ALTER EXCEPT RULE](#), [DROP EXCEPT RULE](#)

13.41 CREATE EXTERNAL SCHEMA

Function

Creates an EXTERNAL schema. This syntax is supported only in 8.3.0 and later versions.

You can create an external schema to access the LakeFormation, DLI, and Hive MetaStore tables. You can use an external schema name as the prefix for access. If there is no schema name prefix, you can access the named objects in the current schema.

Precautions

- A user who has the CREATE permission on the current database can create a foreign schema.
- When creating a named object, do not use EXTERNAL SCHEMA as the prefix. Objects cannot be created in EXTERNAL SCHEMA.
- **CREATE EXTERNAL SCHEMA** does not support subcommands for creating objects in the new schema.

Syntax

- Create an external schema with a specified name.

```
CREATE EXTERNAL SCHEMA schema_name
  WITH SOURCE source_name
    DATABASE 'database_name'
    SERVER server_name
    [ CATALOG 'catalog_name' ]
    [ OPTIONS ( { option_name 'value' } [, ...] ) ]
    [ METAADDRESS 'address' ]
    [ CONFIGURATION 'confpath'];
```

Parameter Description

Table 13-68 CREATE EXTERNAL SCHEMA parameters

Parameter	Description	Value Range
schema_name	Name of an external schema. • The name must be unique. • The name cannot start with pg_.	A string, which must comply with Identifier Naming Conventions .
SOURCE	Type of the external metadata storage engine.	Currently, only dli , lakeformation , and hive are supported.

Parameter	Description	Value Range
DATABASE	Database corresponding to the external schema.	-
SERVER	Name of the external server associated with the external schema. External data can be accessed after the association.	Servers whose type is lf , obs , dli , or hdfs can be associated. For details, see CREATE SERVER .
CATALOG	Catalog to be accessed in LakeFormation. This parameter is mandatory only when server type is set to lf .	-
OPTIONS	Specifies the parameters for a foreign table are listed in the right column. This function is supported only by 8.3.0 and later versions.	dli_project_id: Specifies the project ID corresponding to DLI. You can obtain the project ID from the management console. This parameter is available only when the server type is DLI.
METASTORE ADDRESS	Hive Metastore communication interface. This parameter is supported only by 9.1.0 and later versions.	-
CONFIGURATION	Path for storing Hive Metastore configuration files. This parameter is supported only by 9.1.0 and later versions.	If objects in the schema on the current search path are with the same name, specify the schemas different objects are in. You can run the SHOW SEARCH_PATH command to check the schemas on the current search path.

Examples

- Read the LakeFormation table using the external schema.
 - Create **lf_server**. The corresponding foreign data wrapper is **DFS_FDW**.
 **NOTE**
For details about how to create **lf_server**, see section "Managing LakeFormation Data Sources" in *Data Warehouse Service (DWS) User Guide*.
 - Create an external schema, set **SOURCE** to **lakeformation**, and set the DLI server associated with the table to **lf_server**. In the following command, **DATABASE** is the LakeFormation database, and **CATALOG** is the LakeFormation catalog to be accessed. Replace them with the actual ones you use.

```
CREATE EXTERNAL SCHEMA ex_lf
WITH SOURCE lakeformation
DATABASE 'demo'
SERVER lf_server
CATALOG 'hive';
```
 - Role authorization
 - Query the current user.
 - `SELECT current_user;`
 - Create a role that matches the current role on the LakeFormation management plane and assign it the table access permissions.
 - Query data in the LakeFormation table using the external schema. **test_lf** indicates the LakeFormation table to be accessed.

```
SELECT COUNT(*) FROM ex_dli.test_lf;
count
-----
20
(1 row)
```
- Read the Hive Metastore table using the external schema. Only 9.1.0 and later versions support this function.
 - Create **obs/hdfs server**, with **DFS_FDW** as the foreign data wrapper.

```
CREATE SERVER hdfs_server
FOREIGN DATA WRAPPER HDFS_FDW
OPTIONS (
address '***.***.***.***:9000',
type'HDFS');

CREATE SERVER obs_server
FOREIGN DATA WRAPPER dfs_fdw
OPTIONS (
address 'obs.eu-west-101.myhuaweicloud.eu' ,
ACCESS_KEY 'access_key_value_to_be_replaced',
SECRET_ACCESS_KEY 'secret_access_key_value_to_be_replaced',
encrypt 'on',
type 'obs' );
```
 - Create an external schema. Set **SOURCE** to **hive** and the server associated with the table to **obs/hdfs server**. **DATABASE** indicates the HMS database, **METAADDRESS** indicates the Hive Metastore communication interface, and **CONFIGURATION** indicates the location where Hive Metastore configuration files are saved. Adjust these parameters to meet your site's requirements.

```
CREATE EXTERNAL SCHEMA ex_hms
WITH SOURCE source_type
```

```
DATABASE 'db_name'  
SERVER srv_name  
METAADDRESS 'address'  
CONFIGURATION 'confpath';
```

- Queries data in the HMS table using the external schema. **test_hms** indicates the HMS table to be accessed. Replace it as needed.

```
SELECT COUNT(*) FROM ex_hms.test_hms;  
count  
-----  
20  
(1 row)
```

Helpful Links

[ALTER EXTERNAL SCHEMA](#)

13.42 CREATE FOREIGN TABLE (for GDS Import and Export)

Function

CREATE FOREIGN TABLE creates a GDS foreign table in the current database for concurrent data import and export. The GDS foreign table can be read-only or write-only, used for concurrent data import and export, respectively. The foreign table is read-only by default.

GDS is a command line tool running on the Linux OS. It works with the foreign table mechanism to quickly import and export data. The GDS tool package needs to be installed on the server where the data source file is located. This server is called the data server or the GDS server.

For details about how to install and configure GDS, see [Installing, Configuring, and Starting GDS](#).

Precautions

- Only the system administrator **dbadmin** or a common user who has been granted the USEFT permission can perform foreign table operations.

```
ALTER USER user_name USEFT;
```
- The distribution mode of a GDS foreign table does not need to be explicitly specified. The default is **ROUNDROBIN**.
- All constraints (including column and row constraints) are invalid to the GDS foreign table.
- GDS supports the following file formats: TEXT, CSV, and FIXED.

Syntax

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name  
  ( [ { column_name type_name POSITION(offset,length) | LIKE source_table } [, ...] ] )  
  SERVER gsmpp_server  
  OPTIONS ( { option_name 'value' } [, ...] )  
  [ { WRITE ONLY | READ ONLY } ]  
  [ WITH error_table_name | LOG INTO error_table_name ]  
  [ REMOTE LOG 'name' ]  
  [ PER NODE REJECT LIMIT 'value' ]  
  [ TO { GROUP groupname | NODE ( nodename [, ...] ) } ] ;
```

Parameter Description

Table 13-69 CREATE FOREIGN TABLE (for GDS import and export) parameters

Parameter	Description	Value Range
IF NOT EXISTS	Sends a notice, but does not throw an error, if a table with the same name exists.	-
table_name	Specifies the name of the foreign table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
column_name	Specifies the name of a column in the foreign table.	
type_name	Specifies the data type of the column.	-
POSITION(offset, length)	Defining the location of each column in the data file in fixed length mode. offset is the start of the column in the source file, and length is the length of the column.	offset must be greater than 0 bytes, and its unit is byte. The length of each record must be less than or equal to 1 GB. By default, columns not in the file are replaced with null.
SERVER gsmpp_server	Specifies the server name of the foreign table. For the GDS foreign table, its server is created by initial database, which is gsmpp_server .	-
OPTIONS	Specifies parameters of foreign table data.	• Data format parameters. For details, see Table 13-70. • Error-tolerance parameters. For details, see Table 13-71. • Performance parameters. For details, see Table 13-72.

Parameter	Description	Value Range
READ ONLY	Specifies whether a foreign table is read-only. This parameter is available only for data import.	-

Table 13-70 OPTIONS parameters of the data format

Parameter	Description	Value Range
location	Specifies the data source location of the foreign table, which can be expressed through URLs. Separate URLs with vertical bars (). Currently, GDS can automatically create a directory defined by a foreign table during data export. If foreign table location specified as gsfs://192.168.0.91:5000/2019/09 for an export task, the 2019/09 subdirectory will be automatically created in the GDS data directory if it does not already exist. Manual creation of the directory specified in the foreign table is not necessary.	- For a read-only foreign table imported by GDS from a remote server in parallel, its URL must end with its corresponding schema or file name. (Read-only is the default file attribute.) For example: gsfs://192.168.0.90:5000/* or file:///data/data.txt or gsfs://192.168.0.90:5000/* gsfs://192.168.0.91:5000/*. - For a writable foreign table used for GDS to export data to a remote server in parallel, file names are not required in URLs. If the data source location is a remote URL, for example, gsfs://192.168.0.90:5000/, multiple data sources can be specified. If the number of exported data file locations is less than or equal to the number of DN, when you use the foreign table for export, data is evenly distributed to each data source location. If the number of exported data file locations is greater than the number of DN, when you export data, the data is evenly distributed to data source locations corresponding to the DN. Blank data files are created on the excess data source locations. - For a foreign table used for GDS to import data from a remote server in parallel, the number of URLs must be less than the number of DN, and URLs containing the same location cannot be used. - If the URL begins with gsfs://, data is imported and exported in encryption mode, and DOP cannot exceed 10. - During GDS export, the 2019/09 subdirectory in the gsfs://127.0.0.1:7789/2019/09/ directory specified by the location table is automatically created. - If file_type is set to pipe, GDS determines whether the target file to be imported or exported is a pipe file or a directory based on whether the last character in the URL is a slash (/). Example: - In gsfs://192.168.0.90:5000/a/b, GDS identifies b as a pipe file.

Parameter	Description	Value Range
		– In <code>gsfs://192.168.0.90:5000/a/b/</code>, GDS identifies b as a directory and creates a pipe file in the directory.
format	Specifies the format of the data source file in a foreign table.	Value range: CSV, TEXT, or FIXED. The default value is TEXT. • In CSV files, escape sequences are processed as common strings. Therefore, linefeeds are processed as data. • In TEXT files, escape sequences are processed as they are. Therefore, linefeeds are not processed as data. • The FIXED file can process newline characters in data columns efficiently, but cannot process special characters very well. NOTE • An escape sequence is a string starting with a backslash (\), including <code>\b</code> (backspace), <code>\f</code> (formfeed page break), <code>\n</code> (new line), <code>\r</code> (carriage return), <code>\t</code> (horizontal tab), <code>\v</code> (vertical tab), <code>\number</code> (octal number), and <code>\xnumber</code> (hexadecimal number). In TEXT files, strings are processed as they are. In files of other formats, strings are processed as data. • FIXED is defined as follows: (POSITION must be specified for each column when FIXED is used.) 1. The column length of each record is the same. 2. Spaces are used for column padding. Left padding is used for the numeric type and right padding is used for the char type. 3. No delimiters are used between columns.

Parameter	Description	Value Range
header	Specifies whether a data file contains a table header. header is available only for CSV and FIXED files. When data is imported, if header is on, the first row of the data file will be identified as title row and ignored. If header is off, the first row is identified as data. When data is exported, if header is on, fileheader must be specified. fileheader is used to specify the export header file format. If header is off, the exported file does not include a title row.	true/on or false/off . The default value is false or off .
fileheader	Specifies a file that defines the content in the header for exported data. The file contains one row of data description of each column. For example, to add a header in a file containing product information, define the file as follows: The information of products.\n	• This parameter is available only when header is on or true. The file must be prepared in advance. • In Remote mode, the definition file must be put to the working directory of GDS (the -d directory specified when starting the GDS). • The definition file can contain only one row of title information, and end with a newline character. Excess rows will be discarded. (Title information cannot contain newline character). • The length of the definition file including the newline character cannot exceed 1 MB.

Parameter	Description	Value Range
out_filename_prefix	Specifies the name prefix of the exported data file exported using GDS from a write-only foreign table. If file_type is set to pipe, the pipe file dbName_schemaName_foreignTableName.pipe is generated. If both out_filename_prefix and location specify a pipe name, the pipe name specified in location is used.	• The prefix of the specified file name must be valid and compliant with the restrictions of the file system in the physical environment where the GDS is deployed. Otherwise, the file will fail to be created. – The file name prefix can contain only lowercase letters, uppercase letters, digits, and underscores (_). – The prefix of the specified export file name cannot contain feature fields reserved for the Windows and Linux OS, including but not limited to: "con","aux","nul","prn","com0","com1","com2","com3","com4","com5","com6","com7","com8","com9","lpt0","lpt1","lpt2","lpt3","lpt4","lpt5","lpt6","lpt7","lpt8","lpt9" – The total length of the absolute path consisting of the exported file prefix, the path specified by GDS -d, .dat, or .pipe should be as required by the file system where GDS is deployed. – It is required that the prefix can be correctly parsed and identified by the receiver (including but not limited to the original database where it was exported) of the data file. Identify and modify the option that causes the file name resolution problem (if any). • To concurrently perform export jobs, do not use the same file name prefix for them. Otherwise, the exported files may overwrite each other or be lost in the OS or file system.

Parameter	Description	Value Range
delimiter	Specifies the column delimiter of data, and uses the default delimiter if it is not set. The default delimiter of TEXT is a tab and that of CSV is a comma (,). No delimiter is used in FIXED format.	The value of delimiter can be a multi-character delimiter whose length is less than or equal to 10 bytes. • A delimiter cannot be \r or \n. • A delimiter cannot be the same as the null value. The delimiter of CSV cannot be same as the quote value. • The delimiter for the TEXT format data cannot contain any of the following characters: \.abcdefghijklmnopqrstuvwxyz0123456789. • The data length of a single row should be less than 1 GB. If the delimiters are too long and there are too many rows, the length of valid data will be affected. • You are advised to use a multi-character, such as the combination of the dollar sign (\$), caret (^), the ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter. • For a multi-character delimiter, do not use the same characters, for example, ---.
quote	Specifies which characters in a CSV source data file will be identified as quotation marks. The default value is a double quotation mark (").	• The quote parameter cannot be the same as the delimiter or null parameter. • The quote parameter must be a single-byte character. • Invisible characters are recommended as quote values, such as 0x07, 0x08, and 0x1b.
escape	Specifies which characters in a CSV source data file are escape characters. Escape characters can only be single-byte characters. Default value: the same as the value of QUOTE	-

Parameter	Description	Value Range
null	Specifies the string that represents a null value.	• The default value is \N for the TEXT format. • The default value for the CSV format is an empty string without quotation marks. • The null value cannot be \r or \n. The maximum length is 100 characters. • The null value cannot be the same as the delimiter or quote parameter.
blank_number_str_to_null	Converts a variable number of spaces of the numeric type (INT1OID , INT2OID , INT4OID , INT8OID , FLOAT4OID , or FLOAT8OID) in a data file to NULL . This parameter is supported only by clusters of version 8.1.3, 8.2.1.236, 9.1.0, or later. For example, if this option is set to true , NULL is returned when null , null ' ' (one space) , or null ' ' (two spaces) is defined.	The value can be true or false . The default value is false , which means no conversion is done.
noescaping	Specifies in TEXT format, whether to escape the backslash (\) and its following characters.	noescaping is available only for the TEXT format. true/on or false/off . The default value is false or off .

Parameter	Description	Value Range
encoding	Specifies the encoding of a data file, that is, the encoding used to parse, check, and generate a data file. Its default value is the default client_encoding value of the current database. Before you import foreign tables, it is recommended that you set client_encoding to the file encoding format, or a format matching the character set of the file. Otherwise, unnecessary parsing and check errors may occur, leading to import errors, rollback, or even invalid data import. Before you import foreign tables, you are also advised to specify this parameter, because the export result using the default character set may not be what you expected. If this parameter is not specified when you create a foreign table, a warning message will be displayed on the client.	- Currently, GDS cannot parse or write in a file using multiple encoding formats during foreign table import or export. - Common encoding formats include UTF8, GBK, and GB18030. GB18030 has two versions: GB18030 and GB18030_2022. GB18030_2022 is the latest national standard in China prepared to support Chinese characters.

Parameter	Description	Value Range
dataencoding	Specifies the actual encoding of data in tables in the latin1 database. Converts data format during GDS data import and export. This parameter is supported by version 8.2.0 or later clusters.	GBK, GB18030, GB18030_2022, and UTF8. Typical scenarios: - To import the GB-encoded text data to the latin1 database and convert the data to UTF8. Set dataencoding to UTF8 and encoding to GB. - To export GB-encoded data from the latin1 database as a UTF8 data file, set dataencoding to GB and encoding to UTF8. CAUTION - This parameter can be used only in latin1 databases. - When this parameter is used, encoding must be GBK, GB18030, GB18030_2022, or UTF8.
mode	Specifies the data import policy during a specific data import process. DWS supports only the Normal mode.	Valid value: Normal (default): supports all file types (CSV, TEXT, FIXED). Enabling Gauss data service to help data import.
eol	Specifies the newline character style of the imported or exported data file.	Value range: multi-character newline characters within 10 bytes. Common newline characters include \r (0x0D), \n (0x0A), and \r\n (0x0D0A). Special newline characters include \$ and # . - The eol parameter supports only the TEXT format for data import and export and does not support the CSV or FIXED format for data import. For forward compatibility, the eol parameter can be set to 0x0D or 0x0D0A for data export in the CSV and FIXED formats. - If you want to export data in FIXED format as a single row, use the eol parameter and set it to an empty string ". However, it is important to note that this method of exporting data cannot be directly imported into the GaussDB(DWS) database. - The value of the eol parameter cannot be the same as that of DELIMITER or NULL. - The value of the eol parameter cannot contain digits, letters, or periods (.).

Parameter	Description	Value Range
conflict_delimiter	This parameter is generally used with the compatible_illegal_chars parameter. If a data file contains a truncated Chinese character, the truncated character and a delimiter will be encoded into another Chinese character due to inconsistent encoding between the foreign table and the database. As a result, the delimiter is masked and an error will be reported, indicating that there are missing fields. This parameter is used to avoid encoding a truncated character and a delimiter into another character.	Value range: true, on, false, and off. The default value is false or off. • If the parameter is set to true or on, encoding a truncated character and a delimiter into another character is allowed. • If the parameter is set to false or off, encoding a truncated character and a delimiter into another character is not allowed. This parameter is disabled by default. It is recommended that you disable this parameter, because encoding a truncated character and a delimiter into another character is rarely required. If the parameter is enabled, the scenario may be incorrectly identified and thereby causing incorrect information imported to the table.
file_type	Specifies the type of the file to be imported or exported.	Value options: normal, pipe, and interconn. normal is the default value. • If this parameter is set to normal, the file to be imported or exported is a common file. • If this parameter is set to pipe, the file to be imported or exported is a named pipe. • If the parameter is interconn, the current GDS foreign table is used for interconnected and compressed import and export. This parameter must be used together with the gds_compress parameter and is supported only by cluster 8.2.0 or later. CAUTION If the value of file_type is set to interconn, GDS must be upgraded to 8.2.0 or later. Otherwise, the error message "ERROR: un-support format" will be displayed.

Parameter	Description	Value Range
auto_create_pipe	Specifies whether the GDS process automatically creates a named pipe. When configuring auto_create_pipe, you need to set file_type to pipe or interconn. Otherwise, the foreign table cannot be created.	Value options: true, on, false, and off. The default value is true/on. • If this parameter is set to true or on, the GDS process is allowed to automatically create a named pipe. • If this parameter is set to false or off, you need to manually create a named pipe. • If auto_create_pipe is set to false and no pipe is specified during data import and export, the <i>database name_schema name_foreign table name.pipe</i> file will be opened. If a pipe has been specified, the specified pipe in the location will be opened. If the named pipe is not written by other programs or is not opened in write mode within the period specified by the pipe-timeout parameter, an error message is displayed indicating that the import or export task times out. If the file is not a pipe, an error is reported when the import or export task is executed. • If auto_create_pipe is set to true and no pipe file is specified during data import and export, the <i>database name_schema name_foreign table name.pipe</i> file will be opened. If the file is a common file, an error is reported when the file is imported or exported. If the file is a pipe, the system automatically deletes the file and re-creates the named pipe. • You can use the location parameter to specify the pipe during data export, for example, location'gsfs://127.0.0.1:7789/aa.pipe. When auto_create_pipe is set to true, GDS automatically creates the aa.pipe file in the data directory.

Parameter	Description	Value Range
del_pipe	Specifies whether to automatically delete the pipe file after the import or export task is complete. When configuring del_pipe, you need to set file_type to pipe or interconn. Otherwise, the foreign table cannot be created.	Value options: true or on; false or off. The default value is true or on. • If this parameter is set to true or on, the GDS process will automatically delete a named pipe file. • If this parameter is set to false or off, the GDS process will not delete a named pipe file.

Parameter	Description	Value Range
<code>gds_compress</code>	This parameter is used to reduce the network bandwidth for cluster interconnection when users use GDS pipe files. Currently, and only the Snappy algorithm is supported. This parameter is supported by version 8.2.0 or later clusters.	Value range: snappy. Currently, only the Snappy algorithm is supported, which means that data imported and exported through GDS pipes is compressed and decompressed using the Snappy algorithm. - When setting this parameter, file_type must be set to interconn and format to CSV. If this parameter is not set, by default, files are not compressed. Currently, this parameter can only be set to snappy. When using this function, ensure that the versions of the local cluster, remote cluster, and GDS are of the same version. - If the value of file_type is interconn, GDS must be upgraded to 8.2.0 or later. Otherwise, the error message "ERROR: unsupported format" is displayed. - This parameter reduces the network bandwidth between clusters and increases the cluster CPU load. You can determine whether to enable this parameter based on the site requirements. - When a GDS foreign table is used for interconnected data migration, to allow data to be compressed during migration, the GDS foreign table must support this parameter. You use interconnected foreign tables to compress and migrate data, or manually use a GDS foreign table to compress and migrate data at the network layer. Many parameters need to be set during manual procedure, you are advised to use interconnected foreign tables. - This parameter must be specified for foreign tables used for both export and import.
<code>preserve_blanks</code>	Specifies whether to retain the blank characters (including spaces, <code>\t</code> , <code>\v</code> , and <code>\f</code>) at the end of each column during fixed-length import. This parameter is supported by version 8.2.0.100 or later clusters.	Value range: true, on, false, and off. The default value is false or off. - If this parameter is set to true or on, the trailing <code>\t</code> is retained and pruning is not performed for column data of the BYTEAOID, CHAROID, NAMEOID, TEXTOID, BPCHAROID, VARCHAROID, NVARCHAR2OID or CSTRINGOID type. - If this parameter is set to false or off, blank characters (including spaces, <code>\t</code>, <code>\v</code>, and <code>\f</code>) at the end of all data types are pruned.

Parameter	Description	Value Range
fix	Specifies the length of fixed format data. The unit is byte. This syntax is available only for READ ONLY foreign tables.	Value range: Less than 1 GB , and greater than or equal to the total length specified by POSITION (The total length is the sum of offset and length in the last column of the table definition.)
out_fix_alignme nt	Specifies how the columns of types BYTEAOID, CHAROID, NAMEOID, TEXTOID, BPCHAROID, VARCHAROID, NVARCHAR2OID, and CSTRINGOID are aligned during fixed-length export.	Value range: align_left , align_right Default value: align_right CAUTION The bytea data type must be in hexadecimal format (for example, \XXXX) or octal format (for example, \XXX\XXX\XXX). The data to be imported must be left-aligned (that is, the column data starts with either of the two formats instead of spaces). Therefore, if the exported file needs to be imported using a GDS foreign table and the file data length is less than that specified by the foreign table formatter, the exported file must be left aligned. Otherwise, an error is reported during the import.
out_fix_num_ali gnment	Specifies how the columns of numeric types TINYINT, SMALLINT, INTEGER, BINARY_INTEGER, BIGINT, NUMERIC[(p[,s])], DECIMAL[(p[,s])], NUMBER[(p[,s])], REAL, FLOAT4, DOUBLE PRECISION, FLOAT8, FLOAT[(p)], BINARY_DOUBLE, DEC[(p[,s])], INTEGER[(p[,s])], SMALLSERIAL, SERIAL and BIGSERIAL are aligned during fixed-length export. This is supported only by clusters of version 9.1.0 or later.	Value range: align_left , align_right Default value: align_left

Parameter	Description	Value Range
date_format	Imports data of the DATE type. This syntax is available only for READ ONLY foreign tables.	Value range: any valid DATE value. For details, see Date and Time Processing Functions and Operators . CAUTION If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see timestamp_format below.
time_format	Imports data of the TIME type. This syntax is available only for READ ONLY foreign tables.	Value range: any valid TIME value. Time zones cannot be used. For details, see Date and Time Processing Functions and Operators .
timestamp_format	Imports data of the TIMESTAMP type. This syntax is available only for READ ONLY foreign tables.	Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
smalldatetime_format	Imports data of the SMALLDATETIME type. This syntax is available only for READ ONLY foreign tables.	Value range: any valid SMALLDATETIME value. For details, see Date and Time Processing Functions and Operators .

Table 13-71 OPTIONS fault tolerance parameters

Parameter	Description	Value Range
fill_missing_fields	Specifies whether to generate an error message when the last column in a row in the source file is lost during data import.	Value range: true , on , false , and off . The default value is false or off . - If this parameter is set to true or on and the last column of a data row in a data source file is lost, the column will be replaced with NULL and no error message will be generated. - If this parameter is set to false or off and the last column is missing, the following error information will be displayed: missing data for column "tt"

Parameter	Description	Value Range
ignore_extra_data	Specifies whether to ignore excessive columns when the number of data source files exceeds the number of foreign table columns. This parameter is available during data import.	Value range: true, on, false, and off. The default value is false or off. - If this parameter is set to true or on and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored. - If this parameter is set to false or off and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed: extra data after last expected column CAUTION If the newline character at the end of the row is lost, setting the parameter to true will ignore data in the next row.
reject_limit	Specifies the maximum number of data format errors allowed during a data import task. If the number of errors does not reach the maximum number, the data import task can still be executed. NOTE You are advised to replace this syntax with PER NODE REJECT LIMIT 'value'. Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. Once a non-data format error occurs, the whole data import process is stopped.	Value range: a positive integer or unlimited. Enclose positive integer values with single quotation marks (''). If this parameter is not specified, an error message is returned immediately.

Parameter	Description	Value Range
<code>compatible_illegal_chars</code>	Specifies whether to tolerate invalid characters during data import and export. This syntax is valid for READ ONLY and WRITE ONLY foreign tables. Only clusters of version 8.1.3.331 or later support export fault tolerance.	Value range: true, on, false, and off. The default value is false or off. • If the parameter is true or on, the database will accept invalid characters after converting them during import or export. • If this parameter is set to false or off, the import process will be stopped if there are invalid characters and an error occurs. • The rule of error tolerance when you import invalid characters is as follows: – <code>\0</code> is converted to a space. – Other illegal characters are converted to question marks. – If <code>compatible_illegal_chars</code> is set to true or on, the database will convert and accept the invalid characters. If NULL, DELIMITER, QUOTE, and ESCAPE are set to a spaces or question marks. Errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt user to modify parameter values that cause confusion, preventing import errors. – Enabling error tolerance for foreign table export will result in invalid characters being exported as question marks (?), which can lead to inconsistencies between the exported and original data when imported back into the GaussDB(DWS) database.
<code>replace_illegal_chars</code>	Error tolerance parameter for importing invalid characters. This parameter is used to replace <code>'\0'</code> with a specified string. This syntax is available only for READ ONLY foreign tables.	The restrictions on the string used to replace <code>'\0'</code> are as follows: • <code>compatible_illegal_chars</code> must be set to true or on. • <code>format</code> must be set to TEXT or CSV. • A maximum of 10 characters is allowed. • The string cannot be the same as the delimiter, or contain the delimiter, or contain the following characters: digits, letters, backslashes (<code>\</code>), and periods (<code>.</code>).

Parameter	Description	Value Range
WITH error_table_name	Data format errors during import are recorded in the table specified by error_table_name . You can query this table after the import to obtain error details. This parameter is available only after reject_limit is set. NOTE To be compatible with PostgreSQL open source interfaces, you are advised to replace this syntax with LOG INTO .	A string, which must comply with the naming convention.
LOG INTO error_table_name	Data format errors during import are recorded in the table specified by error_table_name . You can query this table after the import to obtain error details. NOTE This parameter is available only after PER NODE REJECT LIMIT is set.	A string, which must comply with the naming convention.
REMOTE LOG 'name'	The data format error information is saved as files in GDS. name is the prefix of the error data file.	-

Parameter	Description	Value Range
PER NODE REJECT LIMIT 'value'	This parameter specifies the allowed number of data format errors on each DN during data import. If the number of errors exceeds the specified value on any DN, data import fails, an error is reported, and the system exits data import. CAUTION This syntax specifies the error tolerance of a single node. Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. When a non-data format error occurs, the whole data scanning process is stopped.	An unlimited integer. If this parameter is not specified, an error message is returned immediately.

Table 13-72 OPTIONS performance parameters

Parameter	Description	Value Range
file_sequence	Concurrently imports data in parallel through GDS foreign tables, to improve single-file import performance. This parameter is only used for data import. The parameter format is file_sequence'<i>total number of shards-current shard</i>'. Example: file_sequence '3-1' indicates that the imported file is logically split into three shards and the data currently imported by the foreign table is the data on the first shard. file_sequence '3-2' indicates that the imported file is logically split into three shards and the data currently imported by the foreign table is the data on the second shard. file_sequence '3-3' indicates that the imported file is logically split into three shards and the data currently imported by the foreign table is the data on the third shard.	This parameter has the following constraints: • A file can be split to a maximum of 8 shards. • The number of currently imported shard should be less than or equal to the total number of split shards. • Only CSV and TXT files can be imported. When data is imported in parallel in CSV format, some shards fail to be imported in some scenarios because the CSV rules conflict with the GDS splitting logic. For details, see Exception Description When the file_sequence Parameter Is Used.

Examples

Create a foreign table **customer_ft** to import data from GDS server 10.10.123.234 in TEXT format:

```
CREATE FOREIGN TABLE customer_ft
(
  c_customer_sk      integer      ,
  c_customer_id      char(16)     ,
  c_current_demo_sk  integer      ,
  c_current_demo_sk  integer      ,
  c_current_addr_sk  integer      ,
  c_first_ship_to_date_sk integer   ,
  c_first_sales_date_sk integer   ,
  c_salutation       char(10)     ,
  c_first_name       char(20)     ,
  c_last_name        char(30)     ,
  c_preferred_cust_flag char(1)   ,
  c_birth_day        integer      ,
  c_birth_month      integer      ,
  c_birth_year       integer      ,
  c_birth_country    varchar(20)  ,
  c_login            char(13)     ,
  c_email_address    char(50)     ,
  c_last_review_date char(10)
)
  SERVER gsmpp_server
  OPTIONS
(
  location 'gsfs://10.10.123.234:5000/customer1*.dat',
  FORMAT 'TEXT' ,
  DELIMITER '|',
  encoding 'utf8',
  mode 'Normal')
READ ONLY;
```

Create a foreign table to import data from GDS servers 192.168.0.90 and 192.168.0.91 in TEXT format. Record errors that occur during data import in **foreign_HR_staffs_ft**. A maximum of two data format errors are allowed during the data import.

```
CREATE FOREIGN TABLE foreign_HR_staffs_ft
(
  staff_ID      NUMBER(6) ,
  FIRST_NAME    VARCHAR2(20),
  LAST_NAME     VARCHAR2(25),
  EMAIL         VARCHAR2(25),
  PHONE_NUMBER  VARCHAR2(20),
  HIRE_DATE     DATE,
  employment_ID VARCHAR2(10),
  SALARY        NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID    NUMBER(6),
  section_ID    NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'gsfs://192.168.0.90:5000/* | gsfs://192.168.0.91:5000/*', format
'TEXT', delimiter 'E'\x08', null '', reject_limit '2') WITH err_HR_staffs_ft;
```

Create a foreign table to import all files in the **input_data** directory in CSV format.

```
CREATE FOREIGN TABLE foreign_HR_staffs_ft1
(
  staff_ID      NUMBER(6) ,
  FIRST_NAME    VARCHAR2(20),
  LAST_NAME     VARCHAR2(25),
  EMAIL         VARCHAR2(25),
  PHONE_NUMBER  VARCHAR2(20),
  HIRE_DATE     DATE,
  employment_ID VARCHAR2(10),
```

```
SALARY      NUMBER(8,2),
COMMISSION_PCT NUMBER(2,2),
MANAGER_ID  NUMBER(6),
section_ID  NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'file:///input_data/', format 'csv', quote E'\x08', mode 'private',
delimiter ',') WITH err_HR_staffs_ft1;
```

Create a foreign table to export data to the **output_data** directory in CSV format.

```
CREATE FOREIGN TABLE foreign_HR_staffs_ft2
(
  staff_ID      NUMBER(6) ,
  FIRST_NAME    VARCHAR2(20),
  LAST_NAME     VARCHAR2(25),
  EMAIL         VARCHAR2(25),
  PHONE_NUMBER  VARCHAR2(20),
  HIRE_DATE     DATE,
  employment_ID VARCHAR2(10),
  SALARY        NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID    NUMBER(6),
  section_ID    NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'file:///output_data/', format 'csv', quote E'\x08', delimiter '|',
header 'on') WRITE ONLY;
```

Exception Description When the file_sequence Parameter Is Used

When data is imported in parallel in CSV format, some shards fail to be imported in the following scenario because the CSV rules conflict with the GDS splitting logic:

Scenario: A CSV file contains a newline character that is not escaped, the newline character is contained in the character specified by **quote**, and the data of this line is in the first row of the logical shard.

Example: If you import the **big.csv** file in parallel, the following information is displayed:

```
--id, username, address
10001,"customer1 name","Rose District"
10002,"customer2 name","
23 Road Rose
District NewCity"
10003,"customer3 name","NewCity"
```

After the file is split into two shards, the content of the first shard is as follows:

```
10001,"customer1 name","Rose District"
10002,"customer2 name","
23
```

The content of the second shard is as follows:

```
Road Rose
District NewCity"
10003,"customer3 name","NewCity"
```

The newline character after the first line of the second shard is contained between double quotation marks. As a result, GDS cannot determine whether the newline character is a newline character in the field or a separator in the line. Therefore, two data records on the first shard are successfully imported, but the second shard fails to be imported.

Helpful Links

[ALTER FOREIGN TABLE \(GDS Import and Export\), DROP FOREIGN TABLE](#)

[Installing, Configuring, and Starting GDS](#)

Best Practices: [Using GDS to Import Table Data from a Remote Server to a GaussDB\(DWS\) Cluster](#)

Troubleshooting: [Data Import and Export Faults with GDS](#)

13.43 CREATE FOREIGN TABLE (SQL on OBS or Hadoop)

Function

CREATE FOREIGN TABLE creates an HDFS or OBS foreign table in the current database to access structured data stored in the HDFS or OBS distributed cluster file system. You can also use the HDFS or OBS foreign table to export data in ORC and Parquet formats to HDFS or OBS.

A foreign table is a virtual table within a database or big data platform that does not physically store data. Instead, it serves as a mapping to data residing in external storage systems, such as HDFS or OBS. This mapping is established using metadata, which includes the table structure, the location of the data, and its file format. It enables users to directly access or operate external data using standard SQL statements. A foreign table serves as a data proxy that logically maps data in external file systems to table structures that can be identified by databases.

Foreign tables are designed to break data silos. Foreign tables allow database users to access external system data in a familiar way through metadata mapping and export data to external systems. In this way, data can be efficiently collaborated, analyzed, and shared across systems, especially in big data scenarios, significantly reducing data movement costs and improving collaboration efficiency.

When DWS accesses external data sources through foreign tables, the access performance varies according to data sources.

- Data stored in OBS: Data storage is decoupled from compute. The cluster storage cost is low, and storage capacity is not limited. Clusters can be deleted at any time. However, the compute performance depends on the OBS access performance and is lower than that of HDFS. OBS is recommended when data computing is infrequent.
- Data stored in HDFS: Data storage is not decoupled from compute. The cluster storage cost is high, and storage capacity is limited. The compute performance is high. You must export data before you delete clusters. HDFS is recommended for computing-intensive scenarios.

NOTE

The following read and write foreign tables are supported only in the storage-compute decoupling 3.0 version.

Precautions

- When using an OBS foreign table to access data in an OBS bucket, you need to ensure that your DWS cluster and OBS bucket are in the same region (for example, EU-Dublin).
- Only the system administrator **dbadmin** or a common user who has been granted the USEFT permission can perform foreign table operations.

```
ALTER USER user_name USEFT;
```
- The HDFS foreign table and OBS foreign table are classified into read-only, write-only, and read-write foreign tables. Read-only foreign tables are used for query. Write-only foreign tables can be used to export data from DWS to the distributed file system. Read-write foreign tables can be used for query and data export. Only 9.1.0.100 and later versions support read-write foreign tables.
- Supported formats:
 - Importing data: Only data in ORC, TEXT, CSV, CARBONDATA, PARQUET, and JSON formats can be imported for query.
 - Exporting data: For OBS foreign tables, data in ORC and PARQUET (supported only in storage-compute decoupling 3.0 version) can be exported. For HDFS foreign tables, data in ORC and PARQUET can be exported.
- Before using a foreign table, you need to manually create a foreign server, which is a virtual object configured in the database and used to define parameters (such as the address, protocol, and authentication information) for connecting to an external data source. Through the foreign servers, the local database can directly query and insert external data, just like operating local tables. (The syntax is similar, but may be restricted by performance or functions.) For details, see [CREATE SERVER](#).
- If the foreign data wrapper is set to **HDFS_FDW** or **DFS_FDW** when you manually create a foreign server, you need to configure the **DISTRIBUTE BY** clause to specify the distribution mode when creating a read-only foreign table.

Table 13-73 Read and write formats supported by OBS foreign tables

Data Type	DFS_FDW/HDFS_FDW		
	READ ONLY	WRITE ONLY	READ WRITE
-			
ORC	√	√	√
PARQUET	√	√	√
CARBONDATA	√	×	×
TEXT	√	×	×
CSV	√	×	×
JSON	√	×	×

Syntax

Create an HDFS foreign table.

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( [ { column_name type_name
  [ { [CONSTRAINT constraint_name] NULL |
    [CONSTRAINT constraint_name] NOT NULL |
    column_constraint [...] } ] |
  table_constraint [, ...] } [, ...] ) )
SERVER server_name
OPTIONS ( { option_name 'value ' } [, ...] )
[ {WRITE ONLY | READ ONLY | READ WRITE} ]
DISTRIBUTE BY {ROUNDROBIN | REPLICATION}

[ PARTITION BY ( column_name ) [ AUTOMAPPED ] ] ;
```

- **column_constraint** is as follows:
[CONSTRAINT constraint_name]
{PRIMARY KEY | UNIQUE}
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
- **table_constraint** is as follows:
[CONSTRAINT constraint_name]
{PRIMARY KEY | UNIQUE} (column_name)
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]

Parameter Description

Table 13-74 CREATE FOREIGN TABLE (SQL on OBS or Hadoop) parameters

Parameter	Description	Value Range
IF NOT EXISTS	Sends a notice, but does not throw an error, if a table with the same name exists.	-
table_name	Specifies the name of the foreign table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range
column_name	Specifies the name of a column in the foreign table. Columns are separated by commas (,).	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions. A JSON object consists of nested or parallel name-value pairs, which are irrelevant to the sequence. When data in JSON format is imported, the mapping between fields and values is determined based on the automatic mapping between field names and names of name-value pairs. You need to define proper field names. Otherwise, you may not get the expected result. The rules for automatic mapping between field names and names of name-value pairs are as follows: • If there are no nesting or arrays, the field names must be the same as the names of name-value pairs, case insensitive. • Use underscores (_) to concatenate two names to identify the nesting relationship. • A field name uses the number sign (#) and a decimal non-negative integer (n) to identify the nth element (starting from 0) of an array. For example, to import each element of the <code>{"A": "simple", "B": {"C": "nesting"}, "D": ["array", 2, {"E": "complicated"}]}</code> object, the field names in the foreign table must be defined as <code>a</code>, <code>b</code>, <code>b_c</code>, <code>d</code>, <code>d#0</code>, <code>d#1</code>, <code>d#2</code> and <code>d#2_e</code>. The sequence in which the fields are defined does not affect the import result.
type_name	Specifies the data type of the column.	-
constraint_name	Specifies the name of a constraint for the foreign table.	-

Parameter	Description	Value Range
{ NULL NOT NULL }	Specifies whether the column allows NULL. When you create a table, whether the data in HDFS is NULL or NOT NULL cannot be guaranteed. The consistency of data is guaranteed by users. Users must decide whether the column is NULL or NOT NULL. (The optimizer optimizes the NULL/NOT NULL and generates a better plan.)	-
SERVER server_name	Specifies the server name of the foreign table. Users can customize its name.	A string indicating an existing server. It must comply with the naming convention.
OPTION S	Specifies parameters of foreign table data.	The OPTIONS parameters to be specified vary depending on the scenario. The options are as follows: • Data format parameters. For details, see Table 13-75. • Error-tolerance parameters. For details, see Table 13-76. • Parameters that need to be specified only when the server type is DLI. For details, see Table 13-77. • Parameters that need to be specified only when format is set to hudi. For details, see OPTIONS parameters (only when format is set to hudi). • OBS foreign table. Different data source formats support different OPTIONS parameters. For details, see Table 13-79. • HDFS foreign table. Different data source formats support different OPTIONS parameters. For details, see Table 13-80.

Parameter	Description	Value Range
WRITE ONLY READ ONLY READ WRITE	Specifies the type of the foreign table to be created.	• WRITE ONLY creates a write-only HDFS/OBS foreign table. • READ ONLY creates a read-only HDFS/OBS foreign table. • READ WRITE creates a read/write HDFS/OBS foreign table. If the foreign table type is not specified, a read-only foreign table is created by default.
DISTRIBUTE BY ROUND ROBIN	Specifies ROUNDROBIN as the distribution mode for the HDFS/OBS foreign table. Each row of the table is sent to each DN in turn. Therefore, data is evenly distributed on each DN.	-
DISTRIBUTE BY REPLICATION	Specifies REPLICATION as the distribution mode for the HDFS foreign table. Each row in the table exists on all DNs. That is, full table data is stored on each DN.	-

Parameter	Description	Value Range
PARTITION BY (column_name) AUTOMAPPED	column_name specifies the partition column. AUTOMAPPED means the partition column specified by the HDFS partitioned foreign table is automatically mapped with the partition directory information in HDFS. The prerequisite is that the sequences of partition columns specified in the HDFS foreign table and in the directory are the same. This function is applicable only to read-only foreign tables.	• A write-only partitioned foreign table can have up to four levels of partitions, meaning that you can specify a maximum of four columns as partition columns. At least one column must be reserved as a non-partition column. • HDFS read-only foreign tables support partition tables in text, CSV, CarbonData, ORC, and Parquet formats. • HDFS write-only foreign tables support ORC and Parquet partitioned tables. • HDFS read and write foreign tables support ORC and Parquet partitioned tables. • OBS read-only, write-only, and read-write foreign tables support ORC and Parquet partitioned tables. • Columns of the floating point or Boolean type cannot be used as partition columns. • The maximum length of a partition field can be specified by the GUC parameter dfs_partition_directory_length. dfs_partition_directory_length. • A partition directory name is in the format <i>Partition column name=Partition column value</i>. Special characters in the name will be escaped. To ensure that the total length of the name after escaping does not exceed dfs_partition_directory_length, it is advisable to keep the name length before escaping less than or equal to $(\text{dfs\_partition\_directory\_length} + 1)/3$. • Do not use a column containing too many Chinese characters as a partition column. You may encounter errors when calculating the final partition directory name's length due to the different space requirements of Chinese and English characters. This is particularly true when the partition directory name exceeds the dfs_partition_directory_length length limit.
CONSTRAINT constraint_name	Specifies the name of informational constraint of the foreign table.	A string, which must comply with the naming convention.

Parameter	Description	Value Range
PRIMARY KEY	The primary key constraint specifies that one or more columns of a table must contain unique (non-duplicate) and non-null values. Only one primary key can be specified for a table.	-
UNIQUE	Specifies that a group of one or more columns of a table must contain unique values. For the purpose of a unique constraint, NULL is not considered equal.	-

Parameter	Description	Value Range
NOT ENFORCED	Specifies that the created constraint is an informational constraint, which is not forcibly verified by the database. This parameter is used with ENABLE QUERY OPTIMIZATION. • Informational rather than mandatory: You still need to specify a unique constraint for a column using PRIMARY KEY or UNIQUE to optimize the query execution plan. • No mandatory verification: The database does not check whether the data in the external data source actually meets constraint conditions (for example, whether duplicate values exist). • Performance optimization: The query optimizer is allowed to use an informational constraint to generate a more efficient execution plan. However, you need to ensure	For example, the primary key is declared, but the verification is not mandatory. <pre>CREATE FOREIGN TABLE hdfs_users (user_id INT PRIMARY KEY NOT ENFORCED, ...</pre>

Parameter	Description	Value Range
	that the data meets constraint conditions, or the query result may be incorrect. • Risk: If the actual data violates constraint conditions (for example, duplicate values exist), the query result may be incorrect (for example, the aggregation result is abnormal or redundant records are generated during JOIN operations). For more information, see Informational Constraints.	
ENFORCED	Specifies that the created constraint is an enforced constraint that is forcibly validated by the database. This parameter is reserved. Currently, DWS does not support ENFORCED .	-
PRIMARY KEY (column_name)	Specifies the informational constraint on column_name .	A string indicating an existing column_name. It must comply with the naming convention.

Parameter	Description	Value Range
ENABLE QUERY OPTIMIZATION	Enables the query optimizer to use informational constraints to generate a more efficient execution plan. This parameter is used together with NOT ENFORCED .	Create a foreign table, add a unique constraint to the p_partkey int column, do not enforce the constraint, and allow the optimizer to use the informational constraint to generate a better execution plan. CREATE FOREIGN TABLE ft_part (p_partkey int UNIQUE NOT ENFORCED ENABLE QUERY OPTIMIZATION, ...)
DISABLE QUERY OPTIMIZATION	Disables the query optimizer to use informational constraints to generate an efficient execution plan.	-

Table 13-75 OPTIONS parameters of the data format

Parameter	Description	Value Range
header	Specifies whether a data file contains a table header. header is available only for CSV files.	• true or on: The first row of the data file will be identified as the header and ignored during export. • false or off: The first row is identified as data. The default value is false or off .
quote	Specifies the quotation mark for the CSV format. The default value is a double quotation mark (").	The quote value cannot be the same as the delimiter or null value. The quote value must be a single-byte character. Invisible characters are recommended as quote values, such as 0x07, 0x08, and 0x1b.

Parameter	Description	Value Range
escape	Specifies an escape character for a CSV file. The value must be a single-byte character. In a database, an escape character is a special character used to escape other characters. It enables the database system to identify and correctly process characters that have special meanings and may conflict with SQL syntax (such as quotation marks and special characters) as common characters. Common escape characters include backslashes (\) and double quotation marks ("). The default value is a double quotation mark ("). If it is the same as the value of quote, it will be replaced with \0.	-

Parameter	Description	Value Range
location	Specifies the file path on OBS. This is an OBS foreign table parameter. The data sources of multiple buckets are separated by vertical bars (), for example, LOCATION 'obs://bucket1/folder/ obs://bucket2/'. The database scans all objects in the specified folders. When accessing a DLI multi-version table, you do not need to specify the location parameter.	-
format	Specifies the format of the source data file in a foreign table.	The HDFS foreign table can read data in ORC, TEXT, JSON, CSV, Parquet, and Hudi file formats when in READ ONLY mode. But if you are using the WRITE ONLY/READ WRITE foreign table, you can only work with data in ORC and Parquet file formats. OBS foreign tables allow you to read data in ORC, TEXT, JSON, CSV, CarbonData, Parquet, and Hudi file formats in read-only mode. But if you are using write-only mode, you can only work with data in ORC and Parquet file formats. CAUTION - Only JSON objects (embraced in {}) can be imported. JSON arrays (embraced in []) cannot be imported. However, arrays inside a JSON object can be imported. - Apache Hudi is a transactional data lake platform. In the current version, you can set the format parameter of the OBS foreign table to Hudi to access the corresponding Hudi table data sets. This version also provides the method of accessing the incremental or real-time view of the Hudi table.

Parameter	Description	Value Range
foldername	The directory of the data source file in the foreign table, that is, the corresponding file directory in HDFS or on OBS. This is mandatory for WRITE ONLY and READ WRITE writable foreign tables and optional for READ ONLY foreign tables. When accessing a DLI multi-version table, you do not need to specify the foldername parameter. • OBS foreign table. The format is <i>/OBS bucket name/folder name/</i>. Ensure that the OBS bucket and DWS cluster are in the same region. Cross-region access to OBS bucket data is not supported. • HDFS foreign table. The format is the directory where data files in the remote HDFS are stored. For an MRS cluster, log in to the MRS consoleMRS console, go to the cluster basic information page, click Files, select HDFS File	Example of an OBS foreign table <pre>OPTIONS (foldername '/obs-demo01/obs-dws/', format 'parquet', encoding 'utf8')</pre> Example of an HDFS foreign table: <pre>OPTIONS (foldername '/user/hive/warehouse/demo.db/ product_info_orc/', format 'orc', encoding 'utf8')</pre>

Parameter	Description	Value Range
	List , and find the directory where data is stored.	
encoding	It specifies the encoding format of a source data file in the foreign table. Its default value is utf8 . This parameter is optional.	-
totalrows	(Optional) Estimated number of rows in a table. This parameter is used only for OBS foreign tables. Because OBS may store many files, it is slow to analyze data. This parameter allows you to set an estimated value so that the optimizer can estimate the table size according to the value. Generally, query efficiency is high when the estimated value is close to the actual value.	-

Parameter	Description	Value Range
filenames	Data source files specified in the foreign table. Multiple files are separated by commas (,).	- You are advised to use the foldername parameter to specify the location of the data source. For a read-only foreign table, either filenames or foldername must be specified. For a write-only foreign table or read-write foreign table, only foldername can be specified. - If foldername is an absolute directory, it should be enclosed by slashes (/). Multiple paths are separated by commas (,). - When you query a partitioned table, data is pruned based on partition information, and data files that meet the requirement are queried. Pruning involves scanning HDFS directory contents many times. Therefore, do not use columns with low repetition as partition column. - OBS read-only foreign tables and read/write foreign tables are not supported.
delimiter	Specifies the column delimiter of data, and uses the default delimiter if it is not set. The default delimiter of TEXT is a tab.	The value of delimiter can be a multi-character delimiter whose length is less than or equal to 10 bytes. - A delimiter cannot be \r or \n. - A delimiter cannot be the same as the null parameter. - A separator cannot contain (\), (.), digits, or letters. - The data length of a single row should be less than 1 GB. A row that has many columns using long delimiters cannot contain much valid data. - You are advised to use a multi-character, such as the combination of the dollar sign (\$), caret (^), ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter. delimiter is available only for TEXT and CSV source data files.

Parameter	Description	Value Range
eol	Specifies the newline character style of the imported data file.	Multi-character newline characters are supported, but the newline character cannot exceed 10 bytes. Common newline characters include \r (0x0D), \n (0x0A), and \r\n (0x0D0A). Special newline characters include \$ and #. • The eol parameter applies only to TEXT files. • The value of the eol parameter cannot be the same as that of DELIMITER or NULL. • The value of the eol parameter cannot contain digits, letters, or periods (.).
null	Specifies the string that represents a null value.	The default value is \N for the TEXT format. • The null value cannot be \r or \n. The maximum length is 100 characters. • The null parameter cannot be the same as the delimiter. • null is available only for TEXT and CSV source data files.
noescaping	Specifies in TEXT format, whether to escape the backslash (\) and its following characters.	noescaping is available only for the TEXT format. true/on or false/off. The default value is false or off.
date_format	Specifies the DATE format for data import. This parameter is valid only for READ ONLY foreign tables.	The value must be in a valid DATE format. For details, see Date and Time Processing Functions and Operators. • If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see timestamp_format below. • date_format is available only for TEXT and CSV source data files.
time_format	Specifies the TIME format for data import. This parameter is valid only for READ ONLY foreign tables.	This parameter is available only for TEXT and CSV source data files. Value range: Valid TIME. Time zones cannot be used. For details, see Date and Time Processing Functions and Operators.

Parameter	Description	Value Range
timestamp_format	Specifies the TIMESTAMP format for data import. This parameter is valid only for READ ONLY foreign tables.	This parameter is available only for TEXT and CSV source data files. Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
smalldatetimelformat	Specifies the SMALLDATETIME format for data import. This parameter is valid only for READ ONLY foreign tables.	This parameter is available only for TEXT and CSV source data files. Value range: any valid SMALLDATETIME value. For details, see Date and Time Processing Functions and Operators .
dataencoding	This parameter specifies the data code of the data table to be exported when the database code is different from the data code of the data table. For example, the database code is Latin-1, but the data in the exported data table is in UTF-8 format. This parameter is optional. If this parameter is not specified, the database encoding format is used by default. This syntax is valid only for the WRITE ONLY/READ WRITE foreign table of HDFS.	The parameter is available only for HDFS foreign tables in ORC format with WRITE ONLY or READ WRITE. Value range: data code types supported by the database encoding

Parameter	Description	Value Range
filesize	Specifies the file size of a write-only foreign table. This parameter is optional. If this parameter is not specified, the file size in the distributed file system configuration is used by default. This syntax is valid only for WRITE ONLY/READ WRITE writable foreign tables.	The parameter is available only for HDFS foreign tables in ORC format with WRITE ONLY or READ WRITE. Value range: an integer ranging from 1 to 1024
compression	Specifies the file compression mode. This parameter is optional and is supported only by cluster versions 8.2.0 and later. For a write only/read-write foreign table, it specifies the compression mode for ORC files. For a READ ONLY foreign table, it specifies the compression mode for TEXT, CSV, or JSON files.	- For ORC files: the value can be zlib (default), snappy, and lz4. - For Parquet files: zlib, snappy (default), lz4, lz4_hadoop. - For TEXT, CSV, and JSON files: the value is gzip. CAUTION - In versions earlier than 8.2.0, the parameter cannot be specified for READ ONLY foreign tables. - The lz4_hadoop compression option is designed to work with the Hive LZ4 algorithm. By setting the compression format for a parquet foreign table, both the parquet foreign table and Hive can seamlessly exchange data.
version	Specifies the ORC version number. This parameter is optional. This syntax is available only for WRITE ONLY/READ WRITE foreign tables.	Value range: Only 0.12 is supported. The default value is 0.12 .

Parameter	Description	Value Range
julian_adjust	Specifies whether to correct the Julian day in the data. This parameter is optional and is supported only by cluster versions 8.3.0 and later. Spark 2.X uses the Julian day, and Spark 3.0 and later versions use the Proleptic Gregorian calendar. As a result, some dates before parquet 1582 are different. Correct the dates based on the site requirements.	The value can be true or false . Default value: true
cache_policy	Specifies the disk cache policy for foreign tables. This parameter is supported only in the storage-compute decoupling 3.0 version.	Value range: For foreign tables, this parameter can be set to ALL or NONE. ALL indicates that the hot cache in the disk cache is used, and NONE indicates that the cold cache is used. Hot cache occupies more space than cold cache and uses more complex replacement algorithms.

Table 13-76 OPTIONS fault tolerance parameters

Parameter	Description	Value Range
fill_missing_fields	Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading.	true/on or false/off . The default value is false or off . - If this parameter is set to true or on and the last column of a data row in a data source file is lost, the column is replaced with NULL and no error message will be generated. - If this parameter is set to false or off and the last column is missing, the following error information will be displayed: missing data for column "tt" CAUTION - Because SELECT COUNT(*) does not parse columns in text format, it does not report missing columns. fill_missing_fields is available only for TEXT and CSV source data files.
ignore_extra_data	Specifies whether to ignore excessive columns when the number of data source files exceeds the number of foreign table columns. This parameter is available only during data importing.	true/on or false/off . The default value is false or off . - If this parameter is set to true or on and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored. - If this parameter is set to false or off and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed: extra data after last expected column CAUTION - If the newline character at the end of the row is lost, setting the parameter to true will ignore data in the next row. - Because SELECT COUNT(*) does not parse columns in text format, it does not report missing columns. ignore_extra_data is available only for TEXT and CSV source data files.

Parameter	Description	Value Range
checkencoding	Specifies whether to check the character encoding.	Value range: no, low, and high. The default value is low. In TEXT format, the rule of error tolerance for invalid characters imported is as follows: • \0 is converted to a space. • Other invalid characters are converted to question marks. • Setting checkencoding to low enables invalid characters toleration. If NULL and DELIMITER are set to spaces or question marks (?), errors like "illegal chars conversion may confuse null 0x20" will be displayed, prompting you to modify parameters that may cause confusion and preventing importing errors. In ORC format, the rule of error tolerance for invalid characters imported is as follows: • If checkencoding is set to no, invalid characters are not checked during data import. If you confirm that there are no invalid characters, this option can shorten the import time. • If checkencoding is low, an imported field containing invalid characters will be replaced with a quotation mark string of the same length. • If checkencoding is high, data import stops when an invalid character is detected.

Parameter	Description	Value Range
force_mapping	Indicates the handling method used when no correct name-value pairs are matched for the foreign table columns in JSON format.	The options include true and false. Default value: true • If force_mapping is true, null is entered in the corresponding column. The meaning of null is the same as that defined in JSON. • If force_mapping is false, an error is reported, indicating that the column does not exist. CAUTION There are no restrictions on JSON objects. While the definition of foreign table fields must comply with DWS identifier specifications (such as length and character restrictions). This import method may cause exceptions. For example, a JSON name cannot be correctly identified or a field is repeatedly defined. You are advised to use the fault tolerance option force_mapping or JSON operators (for details, see JSON/JSONB Functions and Operators). For JSON format, SELECT COUNT(*) does not parse specific fields. Therefore, no error is reported when a field is missing or the format is incorrect.

Table 13-77 OPTIONS parameters (only when the server type is DLI)

Parameter	Description	Value Range
dli_project_id	Specifies the project ID corresponding to DLI. You can obtain the project ID from the management console. This parameter is available only when the server type is DLI. This parameter is supported only in 8.1.1 or later.	-
dli_database_name	Specifies the name of the database where the DLI multi-version table to be accessed is located. This parameter is available only when the server type is DLI. This parameter is supported only in 8.1.1 or later.	-
dli_table_name	Specifies the name of the DLI multi-version table to be accessed. This parameter is available only when the server type is DLI. This parameter is supported only in 8.1.1 or later.	-

Table 13-78 OPTIONS parameters (This parameter can be set only when format is set to Hudi. The following parameters are supported only in 9.1.0 or later.)

Parameter	Description	Value Range
auth_server	Specifies a server, which is used for user identity authentication when KMS is used to encrypt data in the Hudi foreign table. This parameter can be specified only when format is set to Hudi .	Value range: name of an available HDFS server. MRS provides the column-level encryption capability for Hudi data. RangerKMS is used to manage users and key IDs and control encryption and decryption permissions. You can specify the auth_server parameter when creating a foreign table to bind user RangerKMS to the table. In this way, you can be authorized and obtain keys to read encrypted data.
kms_url	Specifies the IP address and port number of the KMS service. This parameter can be specified only when format is set to hudi and auth_server is specified.	Value range: a valid string in ip:port format. Generally, if auth_server is specified, the IP address and port number of the KMS service can be automatically parsed. If this parameter is specified, the address is forcibly used to access the KMS service.
cow_improve	Optimization parameter for the COPY_ON_WRITE table. When this parameter is enabled, DNs replace the dws-bigdata component to directly read data from OBS. This parameter can be specified only when format is set to Hudi and cannot be specified together with auth_server .	Value range: true/on and false/off . The default value is false or off . NOTE This parameter is invalid for MERGE_ON_READ tables.

Table 13-79 OPTIONS supported by OBS foreign tables

Parameter	OBS										
	TEXT	CSV	JSON	ORC			CARBONDATA	PARQUET			HUDI
	READ ONLY	READ ONLY	READ ONLY	READ ONLY	WRITE ONLY	READWRITE	READ ONLY	READ ONLY	WRITE ONLY	READWRITE	READ ONLY
location	√	√	√	√	×	×	√	√	×	×	×
format	√	√	√	√	√	√	√	√	√	√	√
header	×	√	×	×	×	×	×	×	×	×	×
delimiter	√	√	×	×	×	×	×	×	×	×	×
quote	×	√	×	×	×	×	×	×	×	×	×
escape	×	√	×	×	×	×	×	×	×	×	×
null	√	√	×	×	×	×	×	×	×	×	×
noescaping	√	×	×	×	×	×	×	×	×	×	×
encoding	√	√	√	√	√	√	√	√	√	√	√
fill_missing_fields	√	√	×	×	×	×	×	×	×	×	×
ignore_extra_data	√	√	×	×	×	×	×	×	×	×	×
date_format	√	√	√	×	×	×	×	×	×	×	×

Parameter	OBS										
time_format	√	√	√	×	×	×	×	×	×	×	×
timestamp_format	√	√	√	×	×	×	×	×	×	×	×
smalldatetime_format	√	√	√	×	×	×	×	×	×	×	×
chunksizes	√	√	√	×	×	×	×	×	×	×	×
filenames	×	×	×	×	×	×	×	×	×	×	×
foldername	√	√	√	√	√	√	√	√	√	√	√
dataencoding	×	×	×	×	×	×	×	×	×	×	×
filesize	×	×	×	×	×	×	×	×	×	×	×
compression	√	√	√	×	√	√	×	×	√	√	×
version	×	×	×	×	√	√	×	×	×	×	×
checkencoding	√	√	√	√	×	√	√	√	√	√	√
total rows	√	√	√	√	×	√	×	×	×	×	×

Parameter	OBS										
force_mappi ng	x	x	√	x	x	x	x	x	x	x	x
auth_ server	x	x	x	x	x	x	x	x	x	x	√
kms_ url	x	x	x	x	x	x	x	x	x	x	√
cow_ impro ve	x	x	x	x	x	x	x	x	x	x	√
julia n_ad just	x	x	x	x	x	x	x	√	x	x	√

Table 13-80 OPTIONS supported by HDFS foreign tables

Parameter	HDFS								
-	TEXT	CSV	JSON	ORC			PARQUET		
	READ ONLY	READ ONLY	READ ONLY	READ ONLY	WRIT E ONLY	REA D WRIT E	READ ONLY	WRIT E ONLY	REA D WRI TE
locati on	x	x	x	x	x	x	x	x	x
forma t	√	√	√	√	√	√	√	√	√
heade r	x	√	x	x	x	x	x	x	x
delimi ter	√	√	x	x	x	x	x	x	x
quote	x	√	x	x	x	x	x	x	x
escap e	x	√	x	x	x	x	x	x	x
null	√	√	x	x	x	x	x	x	x

Parameter	HDFS								
noescaping	√	×	×	×	×	×	×	×	×
encoding	√	√	√	√	√	√	√	√	√
fill_missing_fields	√	√	×	×	×	×	×	×	×
ignore_extra_data	√	√	×	×	×	×	×	×	×
date_format	√	√	√	×	×	×	×	×	×
time_format	√	√	√	×	×	×	×	×	×
timestamp_format	√	√	√	×	×	×	×	×	×
small_datetime_format	√	√	√	×	×	×	×	×	×
chunk_size	√	√	√	×	×	×	×	×	×
filenames	√	√	√	√	×	×	√	×	×
folder_name	√	√	√	√	√	√	√	√	√
dataencoding	×	×	×	×	√	√	×	×	×
filesize	×	×	×	×	√	√	×	×	×
compression	√	√	√	×	√	√	×	√	√
version	×	×	×	×	√	√	×	×	×
checkencoding	√	√	√	√	√	√	√	√	√

Parameter	HDFS								
totalrows	x	x	x	x	x	x	x	x	x
force_mapping	x	x	√	x	x	x	x	x	x
julian_adjust	x	x	x	x	x	x	√	x	x

 NOTE

chunksize indicates the cache size of each OBS reading thread on a DN. The format can be text, CSV, or JSON. The default size is 4 MB.

Informational Constraints

In DWS, the use of data constraints depend on users. If users can make data sources strictly comply with certain constraints, the query on data with such constraints can be accelerated. Foreign tables do not support Index. Informational constraint is used for optimizing query plans.

The constraints of creating informational constraints for an HDFS foreign table are as follows:

- You can create an informational constraint only if the values in a NOT NULL column in your table are unique. Otherwise, the query result will be different from expected.
- Currently, the informational constraint of DWS supports only PRIMARY KEY and UNIQUE constraints.
- The informational constraints of DWS support the NOT ENFORCED attribute.
- UNIQUE informational constraints can be created for multiple columns in a table, but only one PRIMARY KEY constraint can be created in a table.
- Multiple informational constraints can be established in a column of a table (because the function that establishing a column or multiple constraints in a column is the same.) Therefore, you are not advised to set up multiple informational constraints in a column, and only one Primary Key type can be set up.
- Table-level COMMENT is not supported.
- Multi-column combination constraints are not supported.
- Different CNs in the same cluster cannot concurrently export data to the same write-only ORC foreign table.
- The catalog of a write-only foreign table in ORC format can only be used as the export catalog of a single foreign table of DWS. It cannot be used for multiple foreign tables, and other components cannot write other files to this catalog.

Example: Creating an HDFS Foreign Table

In HDFS, import the TPC-H benchmark test tables **part** and **region** using Hive. The path of the **part** table is **/user/hive/warehouse/partition.db/part_4**, and that of the **region** table is **/user/hive/warehouse/mppdb.db/region_orc11_64stripe/**.

1. Establish HDFS_Server, with HDFS_FDW or DFS_FDW as the foreign data wrapper.

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
(address '10.10.0.100:25000,10.10.0.101:25000',
hdfsconfigpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
type'HDFS');
```

NOTE

- The IP addresses and port numbers of HDFS NameNodes are specified in **OPTIONS**. For details about the port number, search for **dfs.namenode.rpc.port** in the MRS-HDFS service configuration. In this example the port number is 25000.
 - **10.10.0.100:25000,10.10.0.101:25000** indicates the IP addresses and port numbers of the primary and standby HDFS NameNodes. It is the recommended format. Two groups of parameters are separated by commas (,).
2. Create an HDFS foreign table. The HDFS server associated with the table is **hdfs_server**, the corresponding file format of the **ft_region** table on the HDFS server is **'orc'**, and the file directory in the HDFS file system is **'/user/hive/warehouse/mppdb.db/region_orc11_64stripe/'**.

- Create an HDFS foreign table without partition keys.

```
CREATE FOREIGN TABLE ft_region
(
    R_REGIONKEY INT4,
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/mppdb.db/region_orc11_64stripe/'
)
DISTRIBUTE BY
    roundrobin;
```

- Create an HDFS foreign table with partition keys.

```
CREATE FOREIGN TABLE ft_part
(
    p_partkey int,
    p_name text,
    p_mfgr text,
    p_brand text,
    p_type text,
    p_size int,
    p_container text,
    p_retailprice float8,
    p_comment text
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/partition.db/part_4'
)
DISTRIBUTE BY
```

```
roundrobin
PARTITION BY
(p_mfgr) AUTOMAPPED;
```

NOTE

DWS allows you to specify files using the keyword **filenames** or **foldername**. The latter is recommended. The key word **distribute** specifies the storage distribution mode of the **ft_region** table.

3. View the created server and foreign table.

```
SELECT * FROM pg_foreign_table WHERE ftrelid='ft_region'::regclass;
ftrelid | ftserver | ftwriteonly |          ftoptions
-----+-----+-----+-----
16510 | 16509 | f | {format=orc,foldername=/user/hive/warehouse/mppdb.db/
region_orc11_64stripe/}
(1 row)

select * from pg_foreign_table where ftrelid='ft_part'::regclass;
ftrelid | ftserver | ftwriteonly |          ftoptions
-----+-----+-----+-----
16513 | 16509 | f | {format=orc,foldername=/user/hive/warehouse/partition.db/part_4}
(1 row)
```

Example: Creating an HDFS Write-Only Foreign Table

Export data from the TPC-H benchmark test table **region** to the **/user/hive/warehouse/mppdb.db/region_orc/** directory of the HDFS file system through the HDFS write-only foreign table.

1. Create an HDFS foreign table. The corresponding foreign data wrapper is **HDFS_FDW** or **DFS_FDW**, which is the same as that in [Example 1](#).
2. Create a write-only HDFS foreign table.

```
CREATE FOREIGN TABLE ft_wo_region
(
  R_REGIONKEY INT4,
  R_NAME TEXT,
  R_COMMENT TEXT
)
SERVER
  hdfs_server
OPTIONS
(
  FORMAT 'orc',
  encoding 'utf8',
  FOLDERNAME '/user/hive/warehouse/mppdb.db/region_orc/'
)
WRITE ONLY;
```

3. Writes data to the HDFS file system through a write-only foreign table.
INSERT INTO ft_wo_region SELECT * FROM region;

Example: Creating an HDFS Write-Only Partitioned Foreign Table (Supported Only by Storage-Compute Decoupling 3.0 Version)

Export data from the TPC-H benchmark test table **region** table to the **/user/hive/warehouse/mppdb.db/region_orc/** directory of the HDFS file system through the write-only, multi-level partitioned HDFS foreign table.

1. Create an HDFS foreign table. The corresponding foreign data wrapper is **HDFS_FDW** or **DFS_FDW**, which is the same as that in [Example: Creating an HDFS Foreign Table](#).

2. Create a write-only foreign table for HDFS multi-level partitions.

```
CREATE FOREIGN TABLE ft_wo_region_partition
(
  R_REGIONKEY INT4,
  R_NAME TEXT,
  R_COMMENT TEXT
)
SERVER
  hdfs_server
OPTIONS
(
  FORMAT 'orc',
  encoding 'utf8',
  FOLDERNAME '/user/hive/warehouse/mppdb.db/region_orc/'
)
WRITE ONLY
PARTITION BY (R_REGIONKEY, R_NAME);
```

3. Writes data to the HDFS file system through a write-only foreign table.

```
INSERT INTO ft_wo_region_partition SELECT * FROM region;
```

Example: Creating an OBS Read-Only Foreign Table (Supported Only by Storage-Compute Decoupling 3.0 version)

Read data from the OBS file system through the foreign table **partsupp**.

1. Create an OBS server.

NOTICE

- Hard-coded or plaintext AK and SK are risky. For security purposes, encrypt your AK and SK and store them in the configuration file or environment variables.
- For details about the syntax for creating a server, see [CREATE SERVER](#).

```
CREATE SERVER obs_srv FOREIGN DATA WRAPPER dfs_fdw OPTIONS (
  address 'obs.eu-west-101.myhuaweicloud.eu',
  type 'obs',
  access_key 'xxx',
  secret_access_key 'xxx'
);
```

2. Create an OBS foreign table **partsupp** and use the cold cache in the disk cache.

```
CREATE FOREIGN TABLE PARTSUPP
(
  PS_PARTKEY    BIGINT NOT NULL,
  PS_SUPPKEY    BIGINT NOT NULL,
  PS_AVAILQTY   BIGINT NOT NULL,
  PS_SUPPLYCOST DECIMAL(15,2) NOT NULL,
  PS_COMMENT    VARCHAR(199) NOT NULL
) SERVER obs_srv options (
  encoding 'utf-8',
  format 'parquet',
  foldername 'xxx',
  cache_policy 'NONE'
) READ ONLY DISTRIBUTE BY ROUNDROBIN;
```

3. Read data from OBS through the foreign table **partsupp**.

```
SELECT * FROM PARTSUPP;
```

Example: Creating an HDFS Foreign Table Containing an Informational Constraint (NOT ENFORCED configured)

Perform operations on an HDFS foreign table that includes informational constraints.

- Create an HDFS foreign table with informational constraints.

```
CREATE FOREIGN TABLE ft_region (  
  R_REGIONKEY int,  
  R_NAME TEXT,  
  R_COMMENT TEXT  
  , primary key (R_REGIONKEY) not enforced)  
SERVER hdfs_server  
OPTIONS(format 'orc',  
  encoding 'utf8',  
  foldername '/user/hive/warehouse/mppdb.db/region_orc11_64stripe')  
DISTRIBUTE BY roundrobin;
```

- Check whether the region table has an informational constraint index:

```
SELECT relname,relhasindex FROM pg_class WHERE oid='ft_region'::regclass;
```

relname	relhasindex
ft_region	f

(1 row)

```
SELECT conname, contype, consoft, conopt, conindid, conkey FROM pg_constraint WHERE conname  
='ft_region_pkey';
```

conname	contype	consoft	conopt	conindid	conkey
ft_region_pkey	p	t	t	0	{1}

(1 row)

- Delete the informational constraint:

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT ft_region_pkey RESTRICT;
```

```
SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname  
='ft_region_pkey';
```

conname	contype	consoft	conindid	conkey

(0 rows)

- Add a unique informational constraint for the foreign table:

```
ALTER FOREIGN TABLE ft_region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT  
ENFORCED;
```

Delete the unique informational constraint:

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT constr_unique RESTRICT;
```

```
SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname  
='constr_unique';
```

conname	contype	consoft	conindid	conkey

(0 rows)

- Add a unique informational constraint for the foreign table:

```
ALTER FOREIGN TABLE ft_region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT  
ENFORCED disable query optimization;
```

```
SELECT relname,relhasindex FROM pg_class WHERE oid='ft_region'::regclass;
```

relname	relhasindex
ft_region	f

(1 row)

Delete the unique informational constraint:

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT constr_unique CASCADE;
```

Example: Reading OBS JSON Data Using an OBS Foreign Table

Read json data stored in OBS using a foreign table.

1. The following JSON files are on OBS. The JSON objects contain nesting and arrays. Some objects have lost columns, and some object names are duplicate.

```
{"A": "simple1", "B": {"C": "nesting1"}, "D": ["array", 2, {"E": "complicated"}]}
{"A": "simple2", "D": ["array", 2, {"E": "complicated"}]}
{"A": "simple3", "B": {"C": "nesting3"}, "D": ["array", 2, {"E": "complicated3"}]}
{"B": {"C": "nesting4"}, "A": "simple4", "D": ["array", 2, {"E": "complicated4"}]}
{"A": "simple5", "B": {"C": "nesting5"}, "D": ["array", 2, {"E": "complicated5"}]}
```

2. Create **obs_server**, with **DFS_FDW** as the foreign data wrapper.

```
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  ADDRESS 'obs.eu-west-101.myhuaweicloud.eu',
  ACCESS_KEY 'xxxxxxxxx',
  SECRET_ACCESS_KEY 'yyyyyyyyyyyyyy',
  TYPE 'OBS'
);
```

NOTE

- **ADDRESS** is the endpoint of OBS. Replace it with the actual endpoint. You can find the domain name by searching for the value of **regionCode** in the **region_map** file.
 - **ACCESS_KEY** and **SECRET_ACCESS_KEY** are access keys for the cloud account system. Replace the values as needed.
 - **TYPE** indicates the server type. Retain the value **OBS**.
3. Create the OBS foreign table **json_f** and define the column names. For example, **d#2_e** indicates that the column is object **e** nested in the 2nd element of array **d**. The OBS server associated with the table is **obs_server**. **foldername** indicates the data source directory of the foreign table, that is, the OBS directory.

NOTICE

// Hard-coded or plaintext AK and SK are risky. For security purposes, encrypt your AK and SK and store them in the configuration file or environment variables.

```
CREATE FOREIGN TABLE json_f (
  a VARCHAR(10),
  b_c TEXT,
  d#1 INTEGER,
  d#2_e VARCHAR(30)
)SERVER obs_server OPTIONS (
  foldername '/xxx/xxx/',
  format 'json',
  encoding 'utf8',
  force_mapping 'true'
)distribute by roundrobin;
```

4. Query the foreign table **json_f**. The fault tolerance parameter **force_mapping** is enabled by default. If a column is missing in a JSON object, NULL is filled in. If a JSON object name is duplicate, the last name prevails.

```
SELECT * FROM json_f;
  a | b_c | d#1 | d#2_e
-----+-----+-----+-----
simple1 | nesting1 | 2 | complicated1
simple2 | | 2 | complicated2
simple3 | nesting3 | 2 | complicated3
```

```
simple4 | nesting4 | 2 | complicated4  
repeat | nesting5 | 2 | complicated5  
(5 rows)
```

Example: Accessing Data on DLI Through a DLI Foreign Table

Read a DLI multi-version foreign table using a foreign table. Only DLI 8.1.1 and later support the multi-version foreign table example.

1. Create **dli_server**, with **DFS_FDW** as the foreign data wrapper.

```
CREATE SERVER dli_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (  
  ADDRESS 'obs.eu-west-101.myhuaweicloud.eu',  
  ACCESS_KEY 'xxxxxxxxx',  
  SECRET_ACCESS_KEY 'yyyyyyyyyyyyyy',  
  TYPE 'DLI',  
  DLI_ADDRESS 'dli.example.com',  
  DLI_ACCESS_KEY 'xxxxxxxxx',  
  DLI_SECRET_ACCESS_KEY 'yyyyyyyyyyyyyy'  
);
```

NOTE

- **ADDRESS** is the endpoint of OBS. **DLI_ADDRESS** is the endpoint of DLI. Replace it with the actual endpoint.
 - **ACCESS_KEY** and **SECRET_ACCESS_KEY** are access keys for the cloud account system to access OBS. Use the actual value.
 - **DLI_ACCESS_KEY** and **DLI_SECRET_ACCESS_KEY** are access keys for the cloud account system to access DLI. Use the actual value.
 - **TYPE** indicates the server type. Retain the value **DLI**.
2. Create the OBS foreign table **customer_address** for accessing DLI. The table does not contain partition columns, and the DLI server associated with the table is **dli_server**. Where, the **project_id** is **xxxxxxxxxxxxxxxxxx**, the **database_name** on DLI is **database123**, and the **table_name** of the table to be accessed is **table456**. Replace them based on the actual requirements.

NOTICE

// Hard-coded or plaintext AK and SK are risky. For security purposes, encrypt your AK and SK and store them in the configuration file or environment variables.

```
CREATE FOREIGN TABLE customer_address  
(  
  ca_address_sk          integer          not null,  
  ca_address_id          char(16)          not null,  
  ca_street_number       char(10)           ,  
  ca_street_name         varchar(60)        ,  
  ca_street_type         char(15)          ,  
  ca_suite_number        char(10)           ,  
  ca_city                varchar(60)        ,  
  ca_county              varchar(30)        ,  
  ca_state               char(2)           ,  
  ca_zip                 char(10)          ,  
  ca_country             varchar(20)        ,  
  ca_gmt_offset          decimal(36,33)     ,  
  ca_location_type       char(20)          ,  
)  
SERVER dli_server OPTIONS (  
  FORMAT 'ORC',  
  ENCODING 'utf8',
```

```
DLI_PROJECT_ID 'xxxxxxxxxxxxxx',  
DLI_DATABASE_NAME 'database123',  
DLI_TABLE_NAME 'table456'  
)  
DISTRIBUTE BY roundrobin;
```

3. Query data in a DLI multi-version table using a foreign table.

```
SELECT COUNT(*) FROM customer_address;  
count  
-----  
20  
(1 row)
```

Helpful Links

- For details about the syntax for modifying and deleting a foreign table, see [ALTER FOREIGN TABLE \(for HDFS or OBS\)](#) and [DROP FOREIGN TABLE](#).
- **More best practices:**
 - For details about how to use an HDFS foreign table to access the remote MRS data, see [Importing Table Data from MRS Hive to a GaussDB\(DWS\) Cluster](#).
 - For details about how to use an OBS foreign table, see [Transferring Data Between OBS Buckets and GaussDB\(DWS\) Clusters](#).
 - For details about how to use a DLI foreign table, see [Importing Table Data from DLI to a GaussDB\(DWS\) Cluster](#).

13.44 CREATE FOREIGN TABLE (for OBS Import and Export)

Function

CREATE FOREIGN TABLE creates an OBS foreign table in the current database for parallel data import and export of OBS data. You do not need to create an external server. The **gsmpp_server** created by the database by default can be used.

This syntax supports only TEXT and CSV data on OBS buckets. If you need to process ORC, CARBONDATA, and PARQUET data, see [CREATE FOREIGN TABLE \(SQL on OBS or Hadoop\)](#) and create an external server.

A foreign table is a virtual table within a database or big data platform that does not physically store data. Instead, it serves as a mapping to data residing in external storage systems, such as HDFS or OBS. This mapping is established using metadata, which includes the table structure, the location of the data, and its file format. It enables users to directly access or operate external data using standard SQL statements. A foreign table serves as a data proxy that logically maps data in external file systems to table structures that can be identified by databases.

Precautions

- Only the data in text and CSV formats is supported, and the OBS connection should be configured.

- When using an OBS foreign table to access data in an OBS bucket, you need to ensure that your DWS cluster and OBS bucket are in the same region (for example, EU-Dublin).
- Foreign tables are classified into read-only foreign tables (**READ ONLY**) and write-only foreign tables (**WRITE ONLY**). By default, foreign tables are read-only. When importing OBS data, you need to set the foreign table to **READ ONLY**. When exporting data to OBS, you need to set the foreign table to **WRITE ONLY**.
- Only the system administrator **dbadmin** or a common user who has been granted the USEFT permission can perform foreign table operations.
`ALTER USER user_name USEFT;`
- The distribution mode of an OBS foreign table does not need to be explicitly specified. The default mode is **ROUNDROBIN**.
- Only constraints in [Informational Constraints](#) take effect for the created foreign table.
- OBS buckets cannot contain Chinese paths.

Table 13-81 Read and write formats supported by OBS foreign tables

Data Type	DIST_FDW	
	READ ONLY	WRITE ONLY
-		
ORC	×	×
PARQUET	×	×
CARBONDATA	×	×
TEXT	√	√
CSV	√	√
JSON	×	×

Syntax

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name  
( { column_name type_name [column_constraint ]  
  | LIKE source_table | table_constraint [, ...]} [, ...] )  
SERVER server_name  
OPTIONS ( { option_name ' value ' } [, ...] )  
[ { WRITE ONLY | READ ONLY } ]  
[ WITH error_table_name | LOG INTO error_table_name ]  
[ PER NODE REJECT LIMIT 'value' ] ;
```

- **column_constraint** is as follows:
[CONSTRAINT constraint_name]
{PRIMARY KEY | UNIQUE}
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
- **table_constraint** is as follows:
[CONSTRAINT constraint_name]
{PRIMARY KEY | UNIQUE} (column_name)
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]

Parameter Description

Table 13-82 CREATE FOREIGN TABLE (for OBS import and export) parameters

Parameter	Description	Value Range or Example
IF NOT EXISTS	Sends a notice, but does not throw an error, if a table with the same name exists.	-
table_name	Specifies the name of the foreign table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
column_name	Specifies the name of a column in the foreign table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
type_name	Specifies the data type of the column.	-
SERVER server_name	Specifies the server name of the foreign table. For OBS foreign tables used for data import and export, you can use gsmpp_server created by the initial database by default or use a customized server.	• If a custom server is used, the foreign data wrapper should be dist_fdw. • For clusters of 8.2.0 and later versions, you can specify the following OBS access parameters in the customized dist_fdw server: access_key, secret_access_key, and security_token. If the preceding parameters are specified in the server, you do not need to specify them again in the foreign table.

Parameter	Description	Value Range or Example
OPTIONS	Specifies parameters of foreign table data.	• Data format parameters. For details, see Table 13-83. • Error-tolerance parameters. For details, see Table 13-84. • Performance parameters. For details, see Table 13-85.
READ ONLY	Specifies whether a foreign table is read-only. This parameter is available only for data import.	-
WRITE ONLY	Specifies whether a foreign table is write-only. This parameter is available only for data export.	-
WITH error_table_name	Data format errors during import are recorded in the table specified by error_table_name . You can query this table after the import to obtain error details. This parameter is available only after reject_limit is set.	A string, which must comply with the naming convention. CAUTION To be compatible with PostgreSQL open source interfaces, you are advised to replace this syntax with LOG INTO . When this parameter is specified, an error table is automatically created.
LOG INTO error_table_name	Data format errors during import are recorded in the table specified by error_table_name . You can query this table after the import to obtain error details.	A string, which must comply with the naming convention. • This parameter is available only after PER NODE REJECT LIMIT is set. • When this parameter is specified, an error table is automatically created.

Parameter	Description	Value Range or Example
PER NODE REJECT LIMIT 'value'	This parameter specifies the allowed number of data format errors on each DN during data import. If the number of errors exceeds the specified value on any DN, data import fails, an error is reported, and the system exits data import.	An unlimited integer. If this parameter is not specified, an error message is returned immediately. CAUTION This syntax specifies the error tolerance of a single node. Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. When a non-data format error occurs, the whole data import process stops.

Parameter	Description	Value Range or Example
NOT ENFORCED	Specifies that the created constraint is an informational constraint, which is not forcibly verified by the database. This option is used with ENABLE QUERY OPTIMIZATION. • Informational rather than mandatory: You still need to specify a unique constraint for a column using PRIMARY KEY or UNIQUE to optimize the query execution plan. • No mandatory verification: The database does not check whether the data in the external data source actually meets constraint conditions (for example, whether duplicate values exist). • Performance optimization: The query optimizer is allowed to use an informational constraint to generate a more efficient execution plan. However, you need to ensure that the data meets constraint conditions, or the query result may be incorrect. • Risk: If the actual data violates constraint conditions (for example, duplicate values exist), the query result may be incorrect (for example, the aggregation result is	For example, the primary key is declared, but the verification is not mandatory. <pre>CREATE FOREIGN TABLE hdfs_users (user_id INT PRIMARY KEY NOT ENFORCED, ...</pre>

Parameter	Description	Value Range or Example
	abnormal or redundant records are generated during JOIN operations). For more information, see Informational Constraints .	
ENFORCED	Specifies that the created constraint is an enforced constraint that is forcibly validated by the database. This parameter is reserved. Currently, DWS does not support ENFORCED .	-
PRIMARY KEY (column_name)	Specifies the informational constraint on column_name .	String, which must conform to the identifier naming conventions. The column name must exist.
ENABLE QUERY OPTIMIZATION	Enables the query optimizer to use informational constraints to generate a more efficient execution plan. This parameter is used together with NOT ENFORCED .	Create a foreign table, add a unique constraint to the p_partkey int column, do not enforce the constraint, and allow the optimizer to use the informational constraint to generate a better execution plan. CREATE FOREIGN TABLE ft_part (p_partkey int UNIQUE NOT ENFORCED ENABLE QUERY OPTIMIZATION, ...
DISABLE QUERY OPTIMIZATION	Disables the query optimizer to use informational constraints to generate an efficient execution plan.	-

Table 13-83 OPTIONS parameters of the data format

Parameter	Description	Value Range or Example
encrypt	Specifies whether HTTPS is enabled for data transfer. on enables HTTPS and off disables it (in this case, HTTP is used). The default value is off .	-
access_key	Indicates the access key (AK, obtained from the user information on the console) used for the OBS access protocol. When you create a foreign table, its AK value is not encrypted and saved to the metadata table of the database. The correctness of the parameter is not verified when a foreign table is created. For details about AKs, see Access Keys .	-
secret_access_key	Indicates the secret access key (SK, obtained from the user information on the console) used for the OBS access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database. The correctness of the parameter is not verified when a foreign table is created. For details about SKs, see Access Keys .	-

Parameter	Description	Value Range or Example
security_token	Corresponds to the SecurityToken value of the temporary security credential in IAM. A temporary AK, a temporary SK, and a temporary security token form a temporary security credential. The temporary security credential is valid for no more than 24 hours. This parameter is supported by version 8.2.0 or later clusters. CAUTION - This parameter is supported by version 8.2.0 or later clusters. - When this parameter is used, access_key and secret_access_key correspond to the temporary AK and SK, respectively.	-
chunksize	Specifies the cache read by each OBS thread on a DN. Its value range is 8 to 512 in the unit of MB. Its default value is 64.	-
location	Specifies the directory where OBS data is stored. The value can be described as a URL in the format of obs://OBS bucket name/folder name. Multiple URLs are separated by vertical bars (). Ensure that the OBS bucket and DWS cluster are in the same region. Cross-region access to OBS bucket data is not supported. For details about how to use this parameter, see Location Parameter Description.	Example: <pre>OPTIONS (encoding 'utf8', location 'obs://<obs_bucket_name>/traffic- data/gcxx', format 'text', ...</pre>

Parameter	Description	Value Range or Example
region	(Optional) specifies the value of regionCode, region information on the cloud. If the region parameter is explicitly specified, the value of region will be read. If the region parameter is not specified, the value of defaultRegion will be read.	Note the following when setting parameters for importing or exporting OBS foreign tables in TEXT or CSV format: • The location parameter is mandatory. The prefixes gsobs and obs indicate file locations on OBS. The gsobs prefix should be followed by <i>obs url</i>, <i>bucket</i>, and <i>prefix</i>. The obs prefix should be followed by <i>bucket</i> or <i>prefix</i>. • The data sources of multiple buckets are separated by vertical bars (), for example, LOCATION 'obs://bucket1/folder/ obs://bucket2/'. The database scans all objects in the specified folders.

Parameter	Description	Value Range or Example
format	Specifies the format of the source data file in a foreign table.	CSV or TEXT (default value). DWS only supports CSV and TEXT formats. • CSV (comma-separated format): – The CSV file can process linefeeds efficiently, but cannot process certain special characters very well. – A CSV file is composed of records that are separated as columns by delimiters. Each record shares the same column sequence. • TEXT (text format): Records are separated as columns by linefeed. The TEXT file can process special characters efficiently, but cannot process linefeeds well.
header	Specifies whether a file contains a header with the names of each column in the file.	When OBS exports data, this parameter cannot be set to true. Use the default value false, indicating that the first row of the exported data file is not the header. When data is imported, if header is on, the first row of the data file will be identified as title row and ignored. If header is off, the first row will be identified as a data row. Valid value: true, on, false, and off. The default value is false or off.

Parameter	Description	Value Range or Example
delimiter	Specifies the column delimiter of data, and uses the default delimiter if it is not set. The default delimiter of TEXT is a tab and that of CSV is a comma (,).	The value of delimiter can be a multi-character delimiter whose length is less than or equal to 10 bytes. • The delimiter of TEXT cannot be \r or \n. • A delimiter cannot be the same as the null value. The delimiter for the CSV format cannot be same as the quote value. • The separator of TEXT data cannot contain letters, digits, backslashes (\), and periods (.). • The data length of a single row should be less than 1 GB. A row that has many columns using long delimiters cannot contain much valid data. • You are advised to use a multi-character string, such as the combination of the dollar sign (\$), caret (^), and ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter.
quote	Specifies the quotation mark for the CSV format. The default value is a double quotation mark (").	• The quote value cannot be the same as the delimiter or null value. • The quote value must be a single-byte character. • Invisible characters are recommended as quote values, such as 0x07, 0x08, and 0x1b.

Parameter	Description	Value Range or Example
escape	Specifies an escape character for a CSV file. The value must be a single-byte character.	The default value is a double quotation mark ("). If the value is the same as the quote value, it will be replaced with \0.
null	Specifies how to represent a null value.	• The default value is \N for the TEXT format. • The default value for the CSV format is an empty string without quotation marks. • The null value cannot be \r or \n. The maximum length is 100 characters. • The null value cannot be the same as the delimiter or quote parameter.
noescaping	Specifies whether to escape the backslash (\) and its following characters in the TEXT format.	noescaping is available only for the TEXT format. true/on or false/off . The default value is false or off .

Parameter	Description	Value Range or Example
encoding	Specifies the encoding of a data file, that is, the encoding used to parse, check, and generate a data file. Its default value is the default client_encoding value of the current database.	Before you import foreign tables, it is recommended that you set client_encoding to the file encoding format, or a format matching the character set of the file. Otherwise, unnecessary parsing and check errors may occur, leading to import errors, rollback, or even invalid data import. Before exporting foreign tables, you are also advised to specify this parameter, because the export result using the default character set may not be what you expect. If this parameter is not specified when you create a foreign table, a warning message will be displayed on the client. CAUTION • Currently, OBS cannot parse a file using multiple character sets during foreign table import. • Currently, OBS cannot write a file using multiple character sets during foreign table export.

Parameter	Description	Value Range or Example
eol	Specifies the newline character style of the imported or exported data file.	Multi-character newline characters are supported, but the newline character cannot exceed 10 bytes. Common newline characters include <code>\r</code> (0x0D), <code>\n</code> (0x0A), and <code>\r\n</code> (0x0D0A). Special newline characters include <code>\$</code> and <code>#</code>. • The eol parameter supports only the TEXT format for data import and export. • The value of the eol parameter cannot be the same as that of DELIMITER or NULL. • The value of the eol parameter cannot contain digits, letters, or periods (.).
date_format	Specifies the DATE format for data import. This syntax is available only for READ ONLY foreign tables.	Valid DATE formats. For details, see Date and Time Processing Functions and Operators. NOTE If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see timestamp_format below.
time_format	Specifies the TIME format for data import. This syntax is available only for READ ONLY foreign tables.	Valid TIME formats. Time zones are not supported.
timestamp_format	Specifies the TIMESTAMP format for data import. This syntax is available only for READ ONLY foreign tables.	Valid TIMESTAMP formats. Time zones are not supported.
smalldatetime_format	Specifies the SMALLDATETIME format for data import. This syntax is available only for READ ONLY foreign tables.	Valid SMALLDATETIME formats.

Parameter	Description	Value Range or Example
bom	Indicates whether a CSV file contains the utf8 BOM.	This parameter is valid only when the foreign table is read-only and uses UTF8 code. Value range: true , on , false , and off Default value: false

Table 13-84 OPTIONS fault tolerance parameters

Parameter	Description	Value Range
fill_missing_fields	Specifies how to handle the problem that the last column of a row in the source file is lost during data import.	true/on or false/off . The default value is false or off . - If this parameter is set to true or on and the last column of a data row in a source data file is lost, the column will be replaced with NULL and no error message will be generated. - If this parameter is set to false or off and the last column of a data row in a source data file is lost, the following error information will be displayed: missing data for column "tt"
ignore_extra_data	Specifies whether to ignore excessive columns when the number of columns in a source data file exceeds that defined in the foreign table. This parameter is available only for data import.	true/on or false/off . The default value is false or off . - If this parameter is set to true or on and the number of source data files exceeds the number of foreign table columns, excessive columns will be ignored. - If this parameter is set to false or off and the number of source data files exceeds the number of foreign table columns, the following error information will be displayed: extra data after last expected column CAUTION If the linefeed at the end of a row is lost and this parameter is set to true , data in the next row will be ignored.

Parameter	Description	Value Range
reject_limit	Specifies the maximum number of data format errors allowed during the data import. If the number of data format errors does not reach the maximum, the data import is successful.	The value is an integer or unlimited. If this parameter is not specified, an error message is returned immediately. CAUTION You are advised to replace this syntax with PER NODE REJECT LIMIT 'value'. Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. Once a non-data format error occurs, the whole data import process is stopped.
force_save_err	Indicates whether to save the error information to the error table after the import exits due to an error.	true/on or false/off. The default value is false or off. This parameter is used together with reject_limit. Once this parameter is enabled: • If reject_limit is not specified, an error record will be retained in the error table. • If reject_limit is set to <i>N</i>, <i>N+1</i> error records will be retained in the error table.

Parameter	Description	Value Range
obs_null_file	Imports and exports empty files between GaussDB(DWS) and OBS. CAUTION - This parameter is supported only in 8.2.1 or later. - If obs_null_file is set to true or on and the export directory contains only the _SUCCESS empty file, the empty table can be exported repeatedly, while if obs_null_file is set to false or off, the empty table cannot be exported repeatedly. - If obs_null_file is set to true or on and files are imported from multiple buckets, an error is reported for the first path that the file does not exist.	true/on or false/off. The default value is false or off. If obs_null_file is set to true or on: - When an empty table is exported from GaussDB(DWS), an empty file named _SUCCESS is generated, indicating that the export is successful. When a non-empty table is exported, the original table and an empty file named _SUCCESS is generated. - When a file is imported to GaussDB(DWS), if the file does not exist or the path is incorrect, the following error information is displayed: No such file or directory: 'XXX'

Parameter	Description	Value Range
<code>compatible_illegal_chars</code>	Specifies whether to enable fault tolerance on invalid characters during data import. This syntax is available only for READ ONLY foreign tables.	true/on or false/off. The default value is false or off. - If this parameter is set to true or on, invalid characters are tolerated and imported to the database after conversion. - If this parameter is set to false or off and an error occurs when there are invalid characters, the import will be interrupted. - The rule of error tolerance when you import invalid characters is as follows: <code>\0</code> is converted to a space. - Other invalid characters are converted to question marks. - Setting compatible_illegal_chars to true or on enables toleration of invalid characters. If NULL, DELIMITER, QUOTE, and ESCAPE are set to spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt the user to modify parameters that may cause confusion, preventing importing errors. CAUTION On a Windows platform, if OBS reads data files using the TEXT format, 0x1A will be treated as an EOF symbol and a parsing error will occur. It is the implementation constraint of the Windows platform. Since OBS on a Windows platform does not support BINARY read, the data can be read by OBS on a Linux platform.

Table 13-85 OPTIONS performance parameters

Parameter	Description	Value Range
file_split_threshold	This parameter is used to optimize the performance of importing data in TEXT format. It specifies the lower limit of the logical block size of a file. If this parameter is specified, large files are split based on the actual file and DN status to improve the import concurrency. The purpose is to evenly distribute tasks on each DN. Therefore, this parameter can be used in scenarios where the number of files is less than the number of DNs or the file size is unbalanced.	0 to 2147483647 , in MB The default value is 0 , indicating that files are not split. • This parameter is supported only in 8.2.0 or later. • This parameter supports only READ ONLY foreign tables in TEXT format. • This parameter specifies the lower limit of the logical block size of a file. It does not specify a block size. For example, if the current file size is 1,024 MB and the number of DNs is 4, If the value of file_split_threshold is less than 256, the file is evenly divided into four blocks, and a 256 MB file import task is allocated to each DN. When file_split_threshold is set to 500, the file is split into 500 MB and 524 MB and allocated to two DNs because the block size cannot be less than 500 MB. This parameter is also applicable to multiple files. • Unless there are clear requirements for block sizes, you are advised to set this parameter to a small value, for example, 10. Otherwise, the concurrency may be affected.

Location Parameter Description

- The URL of a read-only foreign table (the default permission is read-only) can end with the path prefix or the full path of the target object in the format of **obs://Bucket/Prefix**. *Prefix* indicates the prefix of an object path, for example, **obs://mybucket/tpch/nation/**.
- If the **region** parameter is explicitly specified in **obs://Bucket/Prefix**, the value of **region** will be read. If the **region** parameter is not specified, the value of **defaultRegion** will be read.

- The URL of a writable foreign table does not need to contain a file name. You can specify only one data source location for a foreign table. The directory corresponding to the location must be created before you specify the location.
- URLs specified for a read-only foreign table must be different.
- Specify **location** when inserting data to a foreign table.
- Parameter **LOCATION** supports prefixes **gsobs** and **obs**, which are identified as OBS information. **LOCATION** should be followed by **gsobs**, *OBS URL*, and *Bucket*; or by **obs** and *Bucket*.

When importing and exporting data, you are advised to use the **location** parameter as follows:

- You are advised to specify a file name for **location** during data import. If you only specify an OBS bucket or directory, all text files in it will be imported. An error message will be reported if the data format is incorrect. If you set fault tolerance, a large amount of data may be imported to the fault-tolerant table.
- Multiple files in an OBS bucket can be imported at the same time. The matched files are imported based on the file name prefix.
For example, you can identify and import the following two files after specifying the prefix **mybucket/input_data/product_info** in **location**:
mybucket/input_data/product_info.0
mybucket/input_data/product_info.1
- If you specify a file name, for example, **1.csv**, then other files (like **1.csv1** or **1.csv22**) starting with **1.csv** in the bucket or directory where **1.csv** resides will be automatically imported. That is, **1.csv1** and **1.csv22** are automatically imported.
- To specify multiple URLs in OBS mode, separate URLs by using vertical bars (|). In **gsobs** mode, only one URL can be specified.
- During data export, a directory is generated for **location** by default. If you specify only a file name, the system automatically creates a directory whose name starts with the file name and then generates the file that stores the exported data. The file name is automatically generated by DWS.
- You can specify one path for **location** only during data export.

Examples

Create a foreign table named **OBS_ft** to import data in the .txt format from OBS to the **row_tbl** table.

CAUTION

// Hard-coded or plaintext AK and SK are risky. For security purposes, encrypt your AK and SK and store them in the configuration file or environment variables.

```
DROP FOREIGN TABLE IF EXISTS OBS_ft;

CREATE FOREIGN TABLE OBS_ft
( a int, b int)
SERVER gsmpp_server
OPTIONS
(location 'obs://gaussdbcheck/obs_ddl/test_case_data/txt_obs_informatonal_test001',
```

```
format 'text',
encoding 'utf8',
chunksize '32',
encrypt 'on',
ACCESS_KEY 'access_key_value_to_be_replaced',
SECRET_ACCESS_KEY 'secret_access_key_value_to_be_replaced',
delimiter E'\x08')
read only;

DROP TABLE row_tbl;

CREATE TABLE row_tbl( a int, b int);

INSERT INTO row_tbl select * from OBS_ft;
```

Helpful Links

[ALTER FOREIGN TABLE \(for HDFS or OBS\), DROP FOREIGN TABLE](#)

[Best practice: Using GaussDB\(DWS\) to Query Vehicle Routes at Traffic Checkpoints in Seconds](#)

Optimization

- delimiter
 - A delimiter cannot be `\r` or `\n`, or the same as the **null** value. The delimiter of CSV cannot be same as the **quote** value.
 - The data length of a single row should be less than 1 GB. A row that has many columns using long delimiters cannot contain much valid data.
 - You are advised to use a multi-character string, such as the combination of the dollar sign (\$), caret (^), and ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter.
- quote
 - The value must be a single-byte character. The **quote** value cannot be the same as the delimiter or **null** value.
 - Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.
- mode Normal
 - Supports all file types (including CSV, TEXT, and FIXED). To import data, you need to enable GDS on the data server.
- mode Shared
 - Supports the TEXT format. It does not require GDS, but all the user data has to be mounted to the same path of all the nodes through NFS.
- mode Private
 - Used in scenarios where user data has been stored under the same path as the local directory of DNs.

13.45 CREATE FUNCTION

Function

A function is a reusable code block that accepts input parameters, performs specific operations, and returns a value or result set. **CREATE FUNCTION** is used

to create a user-defined function (UDF) in a database, enabling other computing functions that are not supported by the built-in **system functions** of GaussDB(DWS) and covering complex computing scenarios specific to users.

The functions have the following features:

- They support input parameters and return results depending on specific conditions.
- You must declare the return value's data type. It outputs either a single value (scalar function) or a table (table-valued function).
- By wrapping complex logic into one function, you can call it like a built-in SQL function, enhancing code reuse.

Three DWS Function Types Based on Behaviors

SQL functions fall into three types based on their behaviors: immutable, stable, and volatile. **Table 13-86** shows their relationships.

Table 13-86 Function types

Function Type	Definition	Typical Use Case
immutable	An immutable function's output depends solely on its input parameters, ignoring external factors like database state, time, or transaction context. It consistently produces the same result when given identical inputs, regardless of when or where it is called. Additionally, it does not alter database data.	Mathematical calculations, hashing, and string operations—examples include ABS(x), POWER(a, b), and MD5(str).
stable	A stable function produces consistent results within a single query if the inputs remain unchanged. Its output depends on the database's current state, like table data or session variables, so results may vary across separate queries or transactions.	Tasks like filtering or aggregating live data, such as using MAX(col) or MIN(col), which stay constant during one scan.
volatile	Volatile functions are highly unpredictable. Their results can differ with every call or alter the database state by performing actions like writing data. Since their behavior cannot be predicted, the optimizer must recompute them on each execution.	Tasks like generating random numbers, retrieving timestamps, or executing DML operations. For instance, calling the RANDOM() function produces a new value every time.

You can view the **provolatile** column in the **PG_PROC** system catalog to check the function type. (**i** indicates immutable, **s** indicates stable, and **v** indicates volatile.)

Pushing Functions Down to DNs (SHIPPABLE and NOT SHIPPABLE)

The **CREATE FUNCTION** syntax uses **SHIPPABLE** and **NOT SHIPPABLE** to determine if the function runs on the data node (DN).

- Functions of the **IMMUTABLE** type can always be pushed down to the DNs.
- Functions of the **STABLE** or **VOLATILE** type can be pushed down to DNs only if their attribute is **SHIPPABLE**.

Exercise caution when defining the **SHIPPABLE** attribute for a function. **SHIPPABLE** means that the entire function will be pushed down to DNs for execution. If the attribute is incorrectly set, serious problems such as result errors may occur.

The **SHIPPABLE** attribute, like **IMMUTABLE**, comes with usage limits. It requires functions to exclude elements that prevent pushdown. Once pushed down to a DN, the function's calculations must rely solely on the DN's dataset.

Functions cannot be marked as IMMUTABLE if they:

1. Reference tables or views because their results may change when the data updates.
2. Call other **STABLE** or **VOLATILE** functions.
3. Include non-pushdown components, which clash with the **IMMUTABLE** rule. This often involves unsupported pushdown functions or syntax.
4. Perform aggregations needing **STREAM** plans (such as the **listagg** function), as these require computations on both DNs and CNs.

Functions cannot be marked as SHIPPABLE if they:

1. Reference a hash-distributed table.
2. Include non-pushdown components, functions, or syntax. For details, see [Optimizing Statement Pushdown](#)
3. Involve calculations with data spread across DNs, such as aggregations.

CAUTION

To optimize performance DWS, you need to define function attributes accurately. Incorrect settings can reduce query speed and produce inconsistent results.

Precautions

- The precision values (if any) of the parameters or return values of a function are not checked.
- Always define a clear schema for table operations in your functions. For instance, use **hr.table1** to point to a table in the **hr** schema. Without this, the function might not run correctly.
- **current_schema** and **search_path** specified by **SET** during function creation are invalid. The values of **search_path** and **current_schema** after function execution are the same as those before function execution.

- If a function has output parameters, the **SELECT** statement uses the default values of the output parameters when calling the function. When the **CALL** statement calls the function, it requires that the output parameter values are adapted to Oracle. When the **CALL** statement calls an overloaded **PACKAGE** function, it can use the default values of the output parameters. For details, see examples in [CALL](#).
- Only the functions compatible with PostgreSQL or those with the **PACKAGE** attribute can be overloaded. After **REPLACE** is specified, a new function is created instead of replacing a function if the number of parameters, parameter type, or return value is different.
- You can use the **SELECT** statement to specify different parameters using identical functions, but cannot use the **CALL** statement to call identical functions without the **PACKAGE** attribute because **CALL** aligns with Oracle syntax.
- When you create a function, you cannot insert other agg functions out of the avg function or other functions.
- In non-logical cluster mode, return values, parameters, and variables cannot be set to the tables of the Node Groups that are not installed in the system by default. The internal statements of SQL functions cannot be executed on such tables.
- In logical cluster mode, if return values and parameters of the function are user tables, all the tables must be in the same logical cluster. If the function body involves operations on multiple logical cluster tables, the function cannot be set to **IMMUTABLE** or **SHIPPABLE**, preventing the function from being pushed down to a DN.
- In logical cluster mode, the parameters and return values of the function cannot use the **%type** to reference a table column type. Otherwise, the function will fail to be created.
- By default, the permissions to execute new functions are granted to **PUBLIC**. For details, see [GRANT](#). You can revoke the default execution permissions from **PUBLIC** and grant them to other users as needed. To avoid the time window during which new functions can be accessed by all users, create functions in transactions and set function execution permissions.
- In a cluster with multiple CNs, the input or output parameters of a function cannot be set to the temporary table type. This is because the correct temporary schema cannot be obtained based on the table name when the function is created on a non-current CN. As a result, the accurate table type cannot be obtained.

Syntax

- Syntax (compatible with PostgreSQL) for creating a user-defined function:

```
CREATE [ OR REPLACE ] FUNCTION function_name
( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ] } [, ...] ] )
[ RETURNS rettype [ DETERMINISTIC ] | RETURNS TABLE ( { column_name column_type }
[, ...] ) ]
LANGUAGE lang_name
[
  { IMMUTABLE | STABLE | VOLATILE }
  | { SHIPPABLE | NOT SHIPPABLE }
  | WINDOW
  | [ NOT ] LEAKPROOF
  | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER |
```

```
AUTHID CURRENT_USER}
| {FENCED | NOT FENCED}
| {PACKAGE}

| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { {TO | =} value | FROM CURRENT }}
][...]
{
  AS 'definition'
  | AS 'obj_file', 'link_symbol'
}
```

- Oracle syntax of creating a customized function:

```
CREATE [ OR REPLACE ] FUNCTION function_name
( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ] } [, ...] ] )
RETURN rettype [ DETERMINISTIC ]
[
  {IMMUTABLE | STABLE | VOLATILE }
  | {SHIPPABLE | NOT SHIPPABLE}
  | {PACKAGE}
  | {FENCED | NOT FENCED}
  | [ NOT ] LEAKPROOF
  | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | [{ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER |
AUTHID DEFINER | AUTHID CURRENT_USER
}
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { {TO | =} value | FROM CURRENT

][...]

{
  IS | AS
} plsql_body
/
```

Parameter Description

Table 13-87 CREATE FUNCTION parameters

Parameter	Description	Value Range
function_name	Specifies the name of the function to be created (optionally schema-qualified). CAUTION If the name of the function to be created is the same as that of a system function, you are advised to specify a schema. When invoking a user-defined function, you need to specify a schema. Otherwise, the system preferentially invokes the system function.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
argname	Specifies the name of a function parameter.	A string, which must comply with the naming convention.

Parameter	Description	Value Range
argmode	Specifies the mode of a function parameter.	IN, OUT, INOUT, or VARIADIC. • IN: input parameter. The default value is IN. • OUT: output parameter. Only this type can include VARIADIC. • INOUT: both input and output parameters. • OUT VARIADIC: output parameter, which is defined as an array type. CAUTION The parameters of OUT and INOUT cannot be used in function definition of RETURNS TABLE .
argtype	Data types of function parameters.	For details about common data types, see Data Types .
expression	Indicates the default expression of a parameter.	-
rettype	Indicates the return data type.	When there is OUT or IN OUT parameter, the RETURNS clause can be omitted. When this clause is present, the result type matches the specified output parameter. For multiple outputs, the result type becomes RECORD . With only one output, the result type aligns with that single parameter. The SETOF modifier indicates that the function will return a set of items, rather than a single item.
DETERMINISTIC	Adapted to Oracle SQL syntax. You are not advised to use it.	-
column_name	Specifies the column name.	-
column_type	Specifies the column type.	-

Parameter	Description	Value Range
definition	Specifies a string constant defining the function; the meaning depends on the language. It can be an internal function name, a path pointing to a target file, a SQL query, or text in a procedural language.	-
LANGUAGE lang_name	Indicates the name of the language that is used to implement the function. It can be SQL , internal , or the name of user-defined process language. To ensure downward compatibility, the name can use single quotation marks. Contents in single quotation marks must be capitalized.	-
WINDOW	Indicates that the function is a window function. The WINDOW attribute cannot be changed when the function definition is replaced. CAUTION For a user-defined window function, the value of LANGUAGE can only be internal , and the referenced internal function must be a window function.	-

Parameter	Description	Value Range
IMMUTABLE	Indicates that the function always returns the same result if the parameter values are the same.	If the input argument of the function is a constant, the function value is calculated at the optimizer stage. The advantage is that the expression value can be obtained as early as possible, so the cost estimation is more accurate and the execution plan generated is better. A user-defined IMMUTABLE function is automatically pushed down to DN for execution, which may cause potential risks. If a function is defined as IMMUTABLE but the function execution process is in fact not IMMUTABLE, serious problems such as result errors may occur. Therefore, exercise caution when defining the IMMUTABLE attribute for a function. Examples: 1. If a user-defined function references objects such as tables and views, the function cannot be defined as IMMUTABLE, because the function may return different results when the data in a referenced table changes. 2. Call other STABLE or VOLATILE functions. 3. Include non-pushdown components, which clash with the IMMUTABLE rule. Typical scenarios include functions and syntax that cannot be pushed down. 4. If a user-defined function contains an aggregation operation that will generate STREAM plans to complete the operation (meaning that DN and CN are involved for results calculation, such as the LISTAGG function), the function cannot be defined as IMMUTABLE.

Parameter	Description	Value Range
		To prevent possible problems, you can set behavior_compat_options to check_function_conflicts in the database to check definition conflicts. This method can identify the 1 and 2 scenarios described above.
STABLE	Indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same parameter values, but that its result varies by SQL statements.	-
VOLATILE	Indicates that the function value can change even within a single table scan, so no optimizations can be made.	-

Parameter	Description	Value Range
SHIPPABLE NOT SHIPPABLE	Indicates whether the function can be pushed down to DNs for execution.	- Functions of the IMMUTABLE type can always be pushed down to the DNs. - Functions of the STABLE or VOLATILE type can be pushed down to DNs only if their attribute is SHIPPABLE. Exercise caution when defining the SHIPPABLE attribute for a function. SHIPPABLE means that the entire function will be pushed down to DNs for execution. If the attribute is incorrectly set, serious problems such as result errors may occur. Similar to the IMMUTABLE attribute, the SHIPPABLE attribute has use restrictions. The function cannot contain factors that do not allow the function to be pushed down for execution. If a function is pushed down to a single DN for execution, the function's calculation logic will depend only on the data set of the DN. Examples: - If a function references a hash table, you cannot define the function as SHIPPABLE. - If a function contains factors, functions, or syntax that cannot be pushed down, the function cannot be defined as SHIPPABLE. For details, see Optimizing Statement Pushdown. - If a function's calculation process involves data across DNs, the function cannot be defined as SHIPPABLE. For example, some aggregation operations involve data across DNs.

Parameter	Description	Value Range
PACKAGE	Indicates whether the function can be overloaded. PostgreSQL-style functions can be overloaded, and this parameter is designed for Oracle-style functions.	• All PACKAGE and non-PACKAGE functions cannot be overloaded or replaced. • PACKAGE functions do not support parameters of the VARIADIC type. • The PACKAGE attribute of functions cannot be modified.
LEAKPROOF	Indicates that the function has no side effects. LEAKPROOF can be set only by the system administrator.	-
CALLED ON NULL INPUT	Declares that some parameters of the function can be invoked in normal mode if the parameter values are NULL . This parameter can be omitted.	-
RETURNS NULL ON NULL INPUT STRICT	Indicates that the function always returns NULL whenever any of its parameters are NULL . If this parameter is specified, the function is not executed when there are NULL parameters; instead a NULL result is returned automatically. The usage of RETURNS NULL ON NULL INPUT is the same as that of STRICT .	-
EXTERNAL	The keyword EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.	-

Parameter	Description	Value Range
SECURITY INVOKER AUTHID CURRENT_USER	Indicates that the function is to be executed with the permissions of the user that calls it. This parameter can be omitted. SECURITY INVOKER and AUTHID CURRENT_USER have the same functions.	-
SECURITY DEFINER AUTHID DEFINER	Specifies that the function is to be executed with the permissions of the user that created it. The usage of AUTHID DEFINER is the same as that of SECURITY DEFINER .	-
FENCED NOT FENCED	(Effective only for C functions) Specifies whether functions are executed in fenced mode. In NOT FENCED mode, a function is executed in a CN or DN process. In FENCED mode, a function is executed in a new fork process, which does not affect CN or DN processes.	Application scenarios: • Develop or debug a function in FENCED mode and execute it in NOT FENCED mode. This reduces the cost of the fork process and communication. • Perform complex OS operations, such as open a file, process signals and threads, in FENCED mode so that DWS running is not affected. • The default value is FENCED.
COST execution_cost	A positive number giving the estimated execution cost for the function. The unit of execution_cost is cpu_operator_cost .	Positive
ROWS result_rows	Estimates the number of rows returned by the function. This is only allowed when the function is declared to return a set.	Its value is a positive number. The default value is 1000 .

Parameter	Description	Value Range
configuration_parameter	Specifies the parameter value configured for the function.	• value Sets a specified database session parameter to a specified value. If the value is DEFAULT or RESET, the default setting is used in the new session. OFF closes the setting. Value range: a string – DEFAULT – OFF – RESET Specifies the default value. • from current Uses the value of configuration_parameter of the current session.
plsql_body	Indicates the PL/SQL stored procedure body. When the function is creating users, the log will record unencrypted passwords. You are not advised to do it.	-

Example: Calculate the Sum of Two Integers Using an SQL Function

Syntax:

```
CREATE OR REPLACE FUNCTION func_add_sql(  
integer,           --The first integer parameter is not named and is referenced by the location parameter  
$1.  
integer           --The second integer parameter is not named and is referenced by the location  
parameter $2.  
)  
RETURNS integer    --The return type of the function is integer.  
AS 'select $1 + $2;' --The function body adds the location parameters $1 and $2.  
LANGUAGE SQL       --The SQL language is used to implement this function.  
IMMUTABLE          --The function type is IMMUTABLE, which indicates that the same result is  
returned for the same input.  
RETURNS NULL ON NULL INPUT; --It is same as STRICT. If any parameter is NULL, the function  
returns NULL.
```

Return the sum of the two input parameters.

```
SELECT func_add_sql(8,3);
```

func_add_sql

11

Return null because the input parameter is null.

```
SELECT func_add_sql(Null,3);
```

func_add_sql

(Null)

Example: Increment an Integer, in PL/pgSQL

Syntax:

```
CREATE OR REPLACE FUNCTION func_increment_plsql(  
i integer          --The input parameter is of the integer type.  
) RETURNS integer  --The return value of the function is of the integer type.  
AS $$  
    BEGIN          --The function body begins.  
        RETURN i + 1;  --The return result is the sum of 1 and the input parameter.  
    END;  
$$ LANGUAGE plpgsql;  --The function is written in PL/pgSQL.
```

Return the integer auto-increment result.

```
SELECT func_increment_plsql(5);
```

func_increment_plsql

6

The input parameter is specified by the first column of the **numbers** table. After the function is used for calculation, the calculation result is returned to the second column.

```
DROP TABLE IF EXISTS numbers;  
CREATE TABLE numbers (value integer);  
INSERT INTO numbers VALUES (1), (2), (3), (4), (5);  
SELECT value, func_increment_plsql(value) AS incremented_value FROM numbers ORDER BY 1;
```

value	incremented_value
1	2
2	3
3	4
4	5
5	6

Example: Return a Record in PL/pgSQL

```
CREATE OR REPLACE FUNCTION compute(  
i int,          ---The input parameter is of the integer type.  
out result_1 bigint,  --The output result is 1 and the data type is large integer.  
out result_2 bigint  --The output result is 2 and the data type is large integer.  
)  
RETURNS SETOF RECORD  --A record containing more than one output parameter is returned.  
AS $$  
BEGIN  
    result_1 = i + 1;  -- It is used to calculate the first result value. The input value is increased by 1.  
    result_2 = i * 10; -- It is used to calculate the second result value. The input value is multiplied  
by 10.  
    RETURN next;      --It returns the current calculation result and then continue to execution  
(multiple rows can be returned).  
END;  
$$LANGUAGE plpgsql;  --The function is written in PL/pgSQL.
```

Return the following information when the function is called once.

```
SELECT * FROM compute(5);
```

result_1	result_2
6	50

Generate a sequence of 1, 2, and 3, and call the compute function for each value in the sequence.

```
SELECT i, (compute(i)).* --Call the compute function first, and then expand all columns returned by the function.
FROM generate_series(1, 3) --Generate a temporary sequence containing 1, 2, and 3 to assign values to i.
AS t(i); --Assign the result to temporary table t, in which i is the column name.
```

i	result_1	result_2
1	2	10
2	3	20
3	4	30

The data in table **test_data** is used. The **value** column in the table is used as the input parameter of the compute function. The calculation result is as follows:

```
DROP TABLE IF EXISTS test_data;
CREATE TABLE test_data (id serial, value int);
INSERT INTO test_data (value) VALUES (2), (4), (6);
```

```
SELECT id, value, (compute(value)).*
FROM test_data ORDER BY 1;
```

id	value	result_1	result_2
1	2	3	20
2	4	5	40
3	6	7	60

Example: Create an SQL Function that Returns Multiple Output Parameters

Syntax:

```
CREATE OR REPLACE FUNCTION func_dup_sql(
in int, --The input parameter is of the integer type.
out f1 int, --The first output parameter is of the integer type.
out f2 text) --The second output parameter is of the text type.
AS $$
SELECT $1, CAST($1 AS text) || 's text' -- Function body. The query returns two columns. The first column is the input parameter $1, and the second column is the parameter converted to text and concatenated with is text.
$$ LANGUAGE SQL;
```

Execute the function.

```
SELECT * FROM func_dup_sql(42);
```

f1	f2
42	42 is text

Example: Create an Overloaded Function with the PACKAGE Attribute

Define two overloaded functions named **package_func_overload** in **package**. The two functions have different meanings depending on the parameter type.

```
CREATE OR REPLACE FUNCTION package_func_overload(
col int,                --The first input parameter is of the integer type.
col2 int                --The second input parameter is of the integer type.
)
RETURN integer PACKAGE
AS
DECLARE
  col_type text;        -- Declare a local variable.
BEGIN
  col := 122;           --Set the first parameter to 122.
  dbms_output.put_line('two int parameters ' || col2);
  RETURN 0;             --Return the integer 0.
END;
/

CREATE OR REPLACE FUNCTION package_func_overload(
col int,
col2 smallint
)
RETURN integer PACKAGE
AS
DECLARE
  col_type text;
BEGIN
  col := 122;
  dbms_output.put_line('two smallint parameters ' || col2);
  RETURN 0;
END;
/
```

Function Pushdown Logic and Example

Pushdown logic for functions has two main aspects:

1. The system checks if the entire function can be pushed down based on its **SHIPPABLE** attribute.
2. If the full function is not pushed down, the optimizer automatically determines which individual SQL statements within the function can be executed separately.

Function-invoking Statements Without User Tables

When a function is called without referencing the user table, the statement runs on the CN, and the function body is not pushed down. Here is how it works:

1. First, execute the function. Since the statement does not include the user table, it always runs on the CN.
2. Next, execute the statements inside the function body. These can be pushed down if nothing prevents it (using FQS or STREAM plan frameworks).

**CAUTION**

- Calling a function alone (without user tables) with SQL join queries still produces accurate results.
- Pushdown only matters if the function call includes user tables directly.
- Execution plans within the function are not visible by default. Setting **enable_track_record_subsql** to **on** to log them.

Example: The function body is NOT SHIPPABLE, and the calling statement lacks the user table.

```
CREATE TABLE student (
  id integer,
  name character varying(50),
  age integer
)
WITH (orientation=row, compression=no)
DISTRIBUTE BY HASH(id);

CREATE TABLE stuscore(
  id integer,
  score integer
)
WITH (orientation=row, compression=no)
DISTRIBUTE BY HASH(id);

INSERT INTO student(id,name,age) VALUES(1,'bob',17),(3,'peter',18),(4,'david',15),(8,'alice',17);
INSERT INTO stuscore(id,score) VALUES(1,88),(3,68),(8,100),(4,90);

--Create a NOT SHIPPABLE function.
CREATE OR REPLACE FUNCTION get_student_id(name character varying)
  RETURNS TABLE(id integer)
  LANGUAGE plpgsql
  VOLATILE NOT FENCED NOT SHIPPABLE
AS $function$
DECLARE
BEGIN
  RETURN QUERY (SELECT id FROM mytest.student WHERE name=$1);
END $function$;
```

Check the execution plan for this function call.

```
EXPLAIN VERBOSE SELECT * FROM get_student_id('alice');
```

Since the user table is missing, the results do not indicate if the function was pushed down, as pushdown only matters when the user table is present.

QUERY PLAN					
id	operation	E-rows	E-distinct	E-width	E-costs
-----+-----+-----+-----+-----					
1	-> Function Scan on public.get_student_id	1000		4	10.25

Predicate Information (identified by plan id)					

1 --Function Scan on public.get_student_id					
Function Call: get_student_id('alice'::character varying)					

Targetlist Information (identified by plan id)					

1 --Function Scan on public.get_student_id					
Output: id					

Function-invoking Statements with User Tables

When a function-calling statement includes a user table, its pushdown capability depends on the function's SHIPPABLE attribute.

Example: The function attribute is NOT SHIPPABLE and the calling statement contains a user table.

Check the execution plan to confirm that the function cannot be pushed down.

```
EXPLAIN VERBOSE SELECT get_student_id('alice') FROM stuscore;
```

QUERY PLAN					
id	operation	E-rows E-distinct E-width E-costs			
-----+-----+-----+-----+-----					
1 ->	Data Node Scan on stuscore "_REMOTE_TABLE_QUERY_"	30000		0	157.35
SQL Diagnostic Information					

SQL is not plan-shipping					
reason: Function get_student_id() can not be shipped					

Example: The function attribute is SHIPPABLE and the calling statement contains a user table.

```
--Change the function attribute to SHIPPABLE.  
CREATE OR REPLACE FUNCTION get_student_id(name character varying)  
  RETURNS TABLE(id integer)  
  LANGUAGE plpgsql  
  VOLATILE NOT FENCED SHIPPABLE  
AS $function$  
DECLARE  
BEGIN  
  RETURN QUERY (SELECT id FROM mytest.student WHERE name=$1);  
END $function$;
```

Check the execution plan to confirm that the function can be pushed down.

```
EXPLAIN VERBOSE SELECT get_student_id('alice') FROM stuscore;
```

QUERY PLAN				
id	operation	E-rows	E-distinct	E-width E-costs
-----+-----+-----+-----+-----				
1	--> Data Node Scan on "__REMOTE_FQS_QUERY__"	0		0 0.00

Targetlist Information (identified by plan id)				

1 --Data Node Scan on "__REMOTE_FQS_QUERY__"				
Output: (get_student_id('alice'::character varying))				
Node/s: All datanodes (v3_logical, bucket:32)				
Remote query: SELECT public.get_student_id('alice'::character varying) AS get_student_id FROM public.stuscore				

Helpful Links

[ALTER FUNCTION](#), [DROP FUNCTION](#)

13.46 CREATE GROUP

Function

CREATE GROUP creates a user group.

Precautions

CREATE GROUP is an alias for **CREATE ROLE**, and it is not a standard SQL command and not recommended. Users can use **CREATE ROLE** directly.

Syntax

```
CREATE GROUP group_name [ [ WITH ] option [ ... ] ]  
[ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The syntax of optional **action** clause is as follows:

```
where option can be:  
{SYSADMIN | NOSYSADMIN}  
| {AUDITADMIN | NOAUDITADMIN}  
| {CREATEDB | NOCREATEDB}  
| {USEFT | NOUSEFT}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCAADMIN}  
| CONNECTION LIMIT connlimit  
| VALID BEGIN 'timestamp'
```

```
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| NODE GROUP logic_group_name  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN role_name [, ...]  
| USER role_name [, ...]  
| SYSID uid  
| DEFAULT TABLESPACE tablespace_name  
| PROFILE DEFAULT  
| PROFILE profile_name  
| PGUSER
```

Parameter Description

See [Parameter Description](#) in **CREATE ROLE**.

Helpful Links

[ALTER GROUP](#), [DROP GROUP](#), and [CREATE ROLE](#)

13.47 CREATE INDEX

Function

CREATE INDEX creates an index in a specified table.

Index Types

DWS supports the following types of indexes:

- B-tree indexes store data keys using a structure like the B+ tree, enabling fast index searches. B-tree indexes support comparison and range queries.
- GIN indexes are reverse indexes and can process values that contain multiple keys (for example, arrays).
- GiST indexes work with multidimensional data like geometry, geography, and collections.
- PSORT indexes are used to perform partial sort on column-store tables.

Notes:

- Row-store tables: B-tree (default), GIN, and GiST indexes
- Column-store tables: PSORT (default), B-tree, and GIN indexes

Partitioned tables have two index types: local and global.

- Local indexes (partitioned indexes) apply to individual partitions within a partitioned table. Each local index matches one partition and shares its count and range with the table. If a table partition changes, only that specific index partition requires maintenance.
- Global indexes span across all partitions, covering the entire table's data regardless of how the base table is partitioned. However, GaussDB(DWS) currently does not support global indexes.

Indexing Methods

Table 13-88 Indexing methods

Method	Description
Unique indexes	An index ensures the uniqueness of one or multiple attributes by acting as a constraint. When a table defines a unique constraint or primary key, DWS generates a unique index—possibly for multiple columns—on those specific columns to enforce this rule. In DWS, only B-tree supports creating unique indexes.
Composite indexes	A table's index can include multiple attributes. In DWS, B-tree allows multi-column indexing with up to 32 columns.
Partial indexes	Refers to an index that can be created for subsets of a table. This indexing method contains only tuples that meet condition expressions.
Expression index	An index can be built on a function or expression derived from one or more table columns. This type of index functions only if queries use the exact expression used during its creation.

Principles for Creating Indexes

Indexes are primarily used to enhance database performance (though inappropriate use can result in slower database performance). You are advised to create indexes on:

- Columns that are often queried
- Join conditions. For a query on joined columns, you are advised to create a composite index on the columns, for example, **select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b**. You can create a composite index on the **a** and **b** columns of table **t1**.
- Columns having filter criteria (especially scope criteria) of a **where** clause
- Columns that appear after **order by**, **group by**, and **distinct**.
- For point queries, create a B-tree index.

The index creation syntax on a partitioned table differs from that on an ordinary table. The partitioned table does not support concurrent index creation, partial index creation, or **NULL FIRST**.

Precautions

- Indexes consume storage and computing resources. Creating too many indexes has negative impact on database performance (especially the performance of data import. Therefore, you are advised to import the data before creating indexes). Create indexes only when they are necessary.
- All functions and operators used in an index definition must be immutable, that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time).

This restriction ensures that the behavior of the index is well-defined. When using a user-defined function on an index or in a **WHERE** statement, mark it as an immutable function.

- An index created on a partitioned table must include a partition column and all the partition keys.
- DWS can only create local indexes on partitioned tables.
- Column-store and HDFS tables support B-tree indexes. If the indexes are used, you cannot create expression and partial indexes.
- Column-store tables support creating unique indexes using B-tree indexes.
- Column-store and HDFS tables support psort indexes. If the psort indexes are used, you cannot create expression, partial, and unique indexes.
- Column-store tables support GIN indexes, rather than partial indexes and unique indexes. If GIN indexes are used, you can create expression indexes. However, an expression in this situation cannot contain empty splitters, empty columns, or multiple columns.
- Column-store indexes do not support **OR** query filter criteria and **inlist** operations.
- A primary key or unique index cannot be created for a round-robin table.
- Performing **CREATE INDEX** or **REINDEX** operations on a table triggers index rebuilding. During this process, data is dumped to a new data file, and once rebuilding is complete, the original file is deleted. For large tables, index rebuilding can consume a significant amount of disk space. Exercise caution when disk space is insufficient to prevent the cluster from becoming read-only.

**WARNING**

- For a table where a large amount of data needs to be added, deleted, or modified, it is recommended that the number of indexes be less than or equal to three. The maximum number of indexes is five.
- Do not perform the **CREATE INDEX** and **REINDEX** operations on large tables during peak hours.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

- Create an index on a table.

```
CREATE [ UNIQUE ] INDEX [ [IF NOT EXISTS] [ schema_name. ] index_name ] ON table_name
[ USING method ]
    ( ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
{ FIRST | LAST } ] }, ... )
    [ NULLS [ NOT ] DISTINCT | NULLS IGNORE ]
    [ COMMENT 'text' ]
    [ WITH ( {storage_parameter = value} [, ... ] ) ]

[ WHERE predicate ];
```
- Create an index for a partitioned table.

```
CREATE [ UNIQUE ] INDEX [ [IF NOT EXISTS] [ schema_name. ] index_name ] ON table_name
[ USING method ]
    ( ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
```

```
LAST ] ][, ... ] )  
[ NULLS [ NOT ] DISTINCT | NULLS IGNORE ]  
[ COMMENT 'text' ]  
LOCAL [ ( { PARTITION index_partition_name } [, ...] ) ]  
[ WITH ( { storage_parameter = value } [, ...] ) ]  
;
```

Parameter Description

Table 13-89 CREATE INDEX parameters

Parameter	Description	Value Range
UNIQUE	Causes the system to check for duplicate values in the table when the index is created (if data exists) and each time data is added. If the inserted or updated value causes duplicate records, an error is reported. Currently, only B-tree indexes of row-store tables and column-store tables support unique indexes.	-
IF NOT EXISTS	If IF NOT EXISTS is specified and no index with the same name exists, the index can be created. If an index with the same name already exists, the system will simply display a message stating that the index already exists and will not perform any additional operations during creation. When IF NOT EXISTS is used, the index name must be specified. This parameter is supported only by clusters of version 9.1.0 or later.	-
schema_name	Name of the schema where the index to be created is located. The specified schema name must be the same as the schema of the table.	-
index_name	Specifies the name of the index to be created. The schema of the index is the same as that of the table.	A string, which must comply with the identifier naming convention .

Parameter	Description	Value Range
table_name	Specifies the name of the table to be indexed (optionally schema-qualified).	Name of an existing table
USING method	Specifies the name of the index method to be used. - Row-based tables support the following index types: btree (default), gin, and gist. - Column-based tables support the following index types: psort (default), btree, and gin.	Value range: btree: B-tree indexes store data keys using a structure like the B+ tree, enabling fast index searches. B-tree indexes support comparison and range queries. gin: GIN indexes work like reverse lists and handle data with many elements, such as arrays. gist: GiST indexes work with multidimensional data like geometry, geography, and collections. psort: PSORT indexes are used to perform partial sort on column-store tables.
column_name	Specifies the name of a column of the table.	Multiple columns can be specified if the index method supports multi-column indexes. A maximum of 32 columns can be specified.

Parameter	Description	Value Range
expression	Specifies an expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax.	However, the parentheses can be omitted if the expression has the form of a function call. Expression can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on upper(col) would allow the clause WHERE upper(col) = 'JIM' to use an index. If an expression contains IS NULL, the index for this expression is invalid. In this case, you are advised to create a partial index.
COLLATE collation	Assigns a collation to the column (which must be of a collatable data type). If no collation is specified, the default collation is used.	-

Parameter	Description	Value Range
opclass	Specifies the name of an operator class.	Specifies an operator class for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type <code>int4</code> would use the <code>int4_ops</code> class; this operator class includes comparison functions for values of type <code>int4</code>. In practice, the default operator class for the column's data type is sufficient. The operator class applies to data with multiple sorts. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.
ASC	Indicates ascending sort order (default). This option is supported only by row storage.	-
DESC	Indicates descending sort order. This option is supported only by row storage.	-
NULLS FIRST	Specifies that nulls sort before not-null values. This is the default when DESC is specified.	-
NULLS LAST	Specifies that nulls sort after not-null values. This is the default when DESC is not specified.	-

Parameter	Description	Value Range
NULLS DISTINCT NULLS NOT DISTINCT NULLS IGNORE	Specifies how NULL values of index columns in a Unique index are processed.	Default value: This parameter is left empty by default. NULL values can be inserted repeatedly. When the inserted data is compared with the original data in the table, the NULL value can be processed in any of the following ways: • NULLS DISTINCT: Each NULL is treated as unique, allowing repeated insertions of NULLs. • NULLS NOT DISTINCT: All NULLs are seen as identical, so only one NULL is kept in the index, preventing repeated NULL insertions; only distinct non-NULL values can be added. • NULLS IGNORE: NULL values are skipped during the equivalent comparison. If all indexed columns are NULL, NULLs can be inserted repeatedly; if some columns are NULL, only new non-NULL values differing from existing ones can be added. The following table Table 13-91 lists the behaviors of the three processing modes.
COMMENT 'text'	Specifies the comment of an index.	-
WITH ({storage_parameter = value} [, ...])	Specifies the name of an index-method-specific storage parameter. Its value can be FILLFACTOR , INVISIBLE , FASTUPDATE , or GIN_PENDING_LIST_LIMIT .	For details about the value range and usage, see Table 13-90 .

Parameter	Description	Value Range
WHERE predicate	Creates a partial index. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table.	Predicate expression can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subquery and aggregate expressions are also forbidden in WHERE .
LOCAL	Local indexes (partitioned indexes) are created on each partition of a partitioned table. A local index corresponds to a partition of the partitioned table, and has the same number of partitions and the same partition range as the table. When the table partition changes, you only need to maintain the corresponding index partition.	-
PARTITION index_partition_name	Specifies the name of the index partition.	A string, which must comply with the naming convention.

Table 13-90 WITH ({storage_parameter = value} [, ...]) parameter

Storage Parameter	Description	Example
FILLFACTOR	This parameter is unavailable for GIN and PSORT indexes. It indicates the fill factor of an index and its value is a percentage between 10 and 100. Index pages are packed only to the indicated percentage and the remaining space is used for subsequent INSERT and UPDATE operations, thereby reducing page splits and improving performance. This parameter determines the percentage of space on index pages to be packed, which affects the database performance and storage space usage. You can adjust its value to reduce page splits caused by subsequent INSERT and UPDATE operations, optimizing the storage and performance of indexes. Impact of FILLFACTOR on index performance: ● Performance: When an index page is fully packed and a new record needs to be inserted, the page will be split, consuming more I/O resources and time. A lower fillfactor can reduce page splits and improve the INSERT and UPDATE performance, but will increase the storage space usage. A higher fillfactor can improve the storage space usage, but may cause more page splits and degrade the INSERT and UPDATE performance. ● Storage space: A lower fillfactor increases the free space of each data page. A higher fillfactor stores data more compactly, reducing disk space usage.	Create a specified B-tree index on the SM_SHIP_MODE_SK column in the tpcds.ship_mode_t1 table and set FILLFACTOR of the index to 70%. This means that each data page is only 70% full when initially packed. The remaining 30% space is reserved for future INSERT and UPDATE operations. <pre>CREATE INDEX ds_ship_mode_t1_index6 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK) WITH (FILLFACTOR = 70);</pre>

Storage Parameter	Description	Example
	How to configure FILLFACTOR: - For tables that are frequently updated, set FILLFACTOR to a small value, for example, 60 to 80, to reserve space and reduce page splits. - For tables that are rarely updated, set FILLFACTOR to a large value, for example, 90 to 100, to maximize storage space.	
INVISIBLE	Controls whether the optimizer generates index scan plans. Scenario: If you want to disable indexes temporarily during maintenance or test the impact of indexes on performance, you can use this parameter. Value range: ON indicates that no index scan plan is generated. OFF indicates that an index scan plan is generated. Default value: OFF	Create a specified B-tree index on the SM_SHIP_MODE_SK column in the tpcds.ship_mode_t1 table and set INVISIBLE to ON (ON indicates that optimizer does not generate the index scan plan.) <pre>CREATE INDEX ds_ship_mode_t1_index7 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK) WITH (INVISIBLE = ON);</pre>
FASTUPDATE	This parameter is available only for GIN indexes. It controls whether to enable fast update for GIN indexes. Once enabled, data changes are temporarily stored in the pending list. Value range: ON or OFF Default value: ON	Create a GIN index on the SM_SHIP_MODE_SK column in the tpcds.ship_mode_t1 table, enable fast update by configuring the FASTUPDATE parameter, and set the maximum pending list size of the index to 8 MB. <pre>CREATE INDEX ds_ship_mode_t1_index8 ON tpcds.ship_mode_t1 USING gin(SM_SHIP_MODE_SK) WITH (FASTUPDATE = ON, GIN_PENDING_LIST_LIMIT = 8192);</pre>

Storage Parameter	Description	Example
GIN_PENDING_LIST_LIMIT	This parameter is supported only by GIN indexes. When fast update is enabled for GIN indexes, you can use this parameter to set the maximum size of the pending list of GIN indexes. Pending List is a data structure specific to GIN indexes and is used to temporarily store index update operations. When the FASTUPDATE parameter is enabled for a GIN index, new index items are not directly written to the main index structure, but are stored in the pending list. When conditions are met, the index items are merged into the main index in batches. Value range: 64 to <i>INT_MAX</i>. The unit is KB. Default value: determined by the value of gin_pending_list_limit (4 MB by default).	

Table 13-91 Processing of NULL values in index columns in unique indexes

Constraint	All Index Columns Are NULL	Some Index Columns Are NULL.
NULLS DISTINCT	Can be inserted repeatedly.	Can be inserted repeatedly.
NULLS NOT DISTINCT	Cannot be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.
NULLS IGNORE	Can be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.

Example

- Create a sample table named **tpcds.ship_mode_t1**.

```
CREATE SCHEMA tpcds;  
DROP TABLE IF EXISTS tpcds.ship_mode_t1;  
CREATE TABLE tpcds.ship_mode_t1  
(  
    SM_SHIP_MODE_SK      INTEGER      NOT NULL,  
    SM_SHIP_MODE_ID     CHAR(16)     NOT NULL,  
    SM_TYPE              CHAR(30)     ,  
    SM_CODE              CHAR(10)     ,  
    SM_CARRIER          CHAR(20)     ,  
    SM_CONTRACT          CHAR(20)  
)  
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

Create a unique index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table.

```
CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

Create a unique index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table and specify how to process null values.

```
CREATE UNIQUE INDEX ds_ship_mode_t1_index5 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK)  
NULLS NOT DISTINCT;
```

Add comment to the index when creating an index on the **SM_SHIP_MODE_SK** column of table **tpcds.ship_mode_t1**.

```
CREATE INDEX ds_ship_mode_t1_index_comment ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK)  
COMMENT 'index';
```

Create a B-tree index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table.

```
CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK);
```

Create a specified B-tree index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table and set **FILLFACTOR** of the index to **70%**.

```
CREATE INDEX ds_ship_mode_t1_index6 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK)  
WITH (FILLFACTOR = 70);
```

Create a specified B-tree index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table and set **INVISIBLE** to **ON** (**ON** indicates that optimizer does not generate the index scan plan. The default value is **OFF**).

```
CREATE INDEX ds_ship_mode_t1_index7 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK)  
WITH (INVISIBLE = ON);
```

Create a GIN index on the **SM_SHIP_MODE_SK** column in the **tpcds.ship_mode_t1** table, enable fast update by configuring the **FASTUPDATE** parameter, and set the maximum pending list size of the index to 8 MB.

```
CREATE INDEX ds_ship_mode_t1_index8 ON tpcds.ship_mode_t1 USING gin(SM_SHIP_MODE_SK)  
WITH (FASTUPDATE = ON, GIN_PENDING_LIST_LIMIT = 8192);
```

Create an expression index on the **SM_CODE** column in the **tpcds.ship_mode_t1** table.

```
CREATE INDEX ds_ship_mode_t1_index2 ON tpcds.ship_mode_t1(SUBSTR(SM_CODE,1,4));
```

Create a partial index on the **SM_SHIP_MODE_SK** column where **SM_SHIP_MODE_SK** is greater than **10** in the **tpcds.ship_mode_t1** table.

```
CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK)  
WHERE SM_SHIP_MODE_SK>10;
```

- Create a sample table named **tpcds.customer_address_p1**.

```
CREATE TABLE tpcds.customer_address_p1  
(  
    CA_ADDRESS_SK      INTEGER      NOT NULL,  
    CA_ADDRESS_ID     CHAR(16)     NOT NULL,
```

```
CA_STREET_NUMBER    CHAR(10)          ,
CA_STREET_NAME      VARCHAR(60)       ,
CA_STREET_TYPE      CHAR(15)          ,
CA_SUITE_NUMBER     CHAR(10)          ,
CA_CITY             VARCHAR(60)       ,
CA_COUNTY           VARCHAR(30)       ,
CA_STATE            CHAR(2)           ,
CA_ZIP              CHAR(10)          ,
CA_COUNTRY           VARCHAR(20)      ,
CA_GMT_OFFSET        DECIMAL(5,2)     ,
CA_LOCATION_TYPE    CHAR(20)
)
DISTRIBUTE BY HASH(CA_ADDRESS_SK)
PARTITION BY RANGE(CA_ADDRESS_SK)
(
  PARTITION p1 VALUES LESS THAN (3000),
  PARTITION p2 VALUES LESS THAN (5000) ,
  PARTITION p3 VALUES LESS THAN (MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

Create the partitioned table index **ds_customer_address_p1_index1** with the name of the index partition not specified.

```
CREATE INDEX ds_customer_address_p1_index1 ON tpods.customer_address_p1(CA_ADDRESS_SK)
LOCAL;
```

Create the partitioned table index **ds_customer_address_p1_index2** with the name of the index partition specified.

```
CREATE INDEX ds_customer_address_p1_index2 ON tpods.customer_address_p1(CA_ADDRESS_SK)
LOCAL
(
  PARTITION CA_ADDRESS_SK_index1,
  PARTITION CA_ADDRESS_SK_index2,
  PARTITION CA_ADDRESS_SK_index3
)
;
```

Create the partitioned table index **ds_customer_address_p1_index_comment** and add index comments.

```
CREATE INDEX ds_customer_address_p1_index_comment ON
tpods.customer_address_p1(CA_ADDRESS_SK) COMMENT 'index' LOCAL
(
  PARTITION CA_ADDRESS_SK_index1,
  PARTITION CA_ADDRESS_SK_index2,
  PARTITION CA_ADDRESS_SK_index3
)
;
```

Application Differences Among NULLS DISTINCT, NULLS NOT DISTINCT, and NULLS IGNORE

Examples are provided below to illustrate the differences.

Example

Assume that there is a **users** table that contains ID and Email (which may be NULL, indicating that no email address is provided).

Scenario 1: NULLS DISTINCT (NULL can be inserted repeatedly) (default)

```
--Create a table.
DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id INT,
  email TEXT
```

```
)  
DISTRIBUTE BY HASH(email);  
--Create a unique index (NULLS DISTINCT by default).  
CREATE UNIQUE INDEX idx_email_unique ON users (email);  
--Insert data.  
INSERT INTO users (id, email) VALUES (1, NULL); -- Success  
INSERT INTO users (id, email) VALUES (2, NULL); -- Success (with each NULL treated as unique)  
INSERT INTO users (id, email) VALUES (3, 'a@example.com'); -- Success  
INSERT INTO users (id, email) VALUES (4, 'a@example.com'); -- Failure (duplicate non-NULL values)
```

```
error_code: DWS.S0010010  
error_msg: Constraint Violation. STATE: 23505, message: ERROR: dn_6005_6006: duplicate key value  
violates unique constraint "idx_email_unique"  
Detail: Key (email)=(a@example.com) already exists.
```

Scenario 2: NULLS NOT DISTINCT (NULL cannot be inserted repeatedly)

```
--Re-create a unique index and specify NULLS NOT DISTINCT.  
DROP TABLE IF EXISTS users;  
CREATE TABLE users (  
    id INT,  
    email TEXT  
)  
DISTRIBUTE BY HASH(email);  
CREATE UNIQUE INDEX idx_email_unique ON users (email) NULLS NOT DISTINCT;  
--Insert data.  
INSERT INTO users (id, email) VALUES (5, NULL); -- Success (the first NULL)  
INSERT INTO users (id, email) VALUES (6, NULL); -- Failure (the second NULL, violating the unique  
constraint)  
INSERT INTO users (id, email) VALUES (7, 'b@example.com'); -- Success  
INSERT INTO users (id, email) VALUES (8, 'b@example.com'); -- Failure (duplicate non-NULL values)
```

Scenario 3: NULLS IGNORE (NULL values are ignored.)

```
--Create an index and ignore NULL values.  
DROP TABLE IF EXISTS users;  
CREATE TABLE users (  
    id INT,  
    email TEXT  
)  
DISTRIBUTE BY HASH(email);  
CREATE UNIQUE INDEX idx_email_ignore ON users (email) NULLS IGNORE;  
--Insert data.  
INSERT INTO users (id, email) VALUES (9, NULL); -- Success (the first NULL)  
INSERT INTO users (id, email) VALUES (10, NULL); -- Success (NULL values are ignored.)  
INSERT INTO users (id, email) VALUES (11, 'c@example.com'); -- Success  
INSERT INTO users (id, email) VALUES (12, 'c@example.com'); -- Failure (duplicate non-NULL values)
```

Helpful Links

- For details about how to modify and delete indexes, see [ALTER INDEX](#) and [DROP INDEX](#).
- For more index cases, see [Creating and Managing Indexes](#).
- For more examples of GIN indexes, see [Best Practices of GIN Index](#).

FAQs

- [What Are the Differences Between GaussDB\(DWS\) Unique Constraints and Unique Indexes?](#)
- [Reindexing Fails](#)
- [When a User Specifies Only an Index Name to Modify the Index, A Message Indicating That the Index Does Not Exist Is Displayed](#)

- [GaussDB\(DWS\) Metadata Inconsistency - Abnormal Partition Index](#)
- [Why Is the Index Invalid During GaussDB\(DWS\) Query?](#)

13.48 CREATE MATERIALIZED VIEW

Function

CREATE MATERIALIZED VIEW creates a materialized view. This syntax is supported only by clusters of version 8.2.1.220 or later.

A materialized view is a special database object. It pre-computes complex query results and stores them in the database to accelerate queries.

When a large amount of data is frequently queried and analyzed, materialized views can be used to pre-calculate and store query results, significantly improving query performance.

Materialized View Permissions

1. To create a materialized view, you must have the **CREATE** permission on the schema and the **SELECT** permission on the base table or column.
2. To query a materialized view, you must have the **SELECT** permission on the materialized view.
3. To refresh a materialized view, you must have the **INSERT** permission on the view and the **SELECT** permission on the base table or column.
4. Other permissions:

Materialized views support fine-grained permissions such as **ANALYZE**, **VACUUM**, **ALTER**, and **DROP**.

Materialized views support **WITH GRANT OPTION**, which means that the recipient of the permission can in turn grant it to others.

5. Security:

Materialized views do not support higher-level security control. In scenarios where the **SELECT** permission is restricted, for example, a base table has row-level access control or masking policies or its owner is a private user, materialized views cannot be created.

If a materialized view already exists and an RLS or masking policy is added to the base table or the owner of the base table is changed to a private user, the materialized view can be queried but cannot be refreshed.

Precautions

- To use **CREATE MATERIALIZED VIEW**, ensure that [enable_matview](#) is set to **on**.
- The base table of a materialized view can be a row-store table, column-store table, HStore table, hot and cold table (supported by clusters of version 9.1.0.200 or later), partitioned table, specified partition, foreign table, or another materialized view.

- Temporary tables (including global temporary tables, volatile temporary tables, and common temporary tables) are not supported. Partitions cannot be specified for partitioned tables that are automatically created.
- The CTE in the definition of a materialized view cannot contain other DML statements except the **SELECT** statement.
- The base table of a materialized view cannot have masking policies and row-level access control, or belong to a private user. Masking policies and row-level access control may cause result set errors, while private users may cause problems such as materialized view refresh failures.
- Materialized views do not support the following storage types: foreign tables and time series tables.
- Materialized views do not allow **INSERT**, **UPDATE**, **MERGE INTO**, or **DELETE** operations.
- The result of each query to the materialized view saved to ensure that the query results are the same each time. After the **BUILD IMMEDIATE** or **REFRESH** operation, the correct result can be queried in the materialized view.
- A node group cannot be specified for a materialized view using syntax. A node group can be specified for the base table of a materialized view. The materialized view can inherit the node group information from the base table. The node groups of multiple base tables must be the same.

Syntax

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] materialized_view_name [ ( column_name [, ...] ) ]  
[ BUILD { DEFERRED | IMMEDIATE } ]  
[ REFRESH [ [DEFAULT] | [ RESTRICT ] | [ CASCADE FORWARD ] | [ CASCADE BACKWARD ] | [ CASCADE  
ALL ] ] [ COMPLETE ] [ ON DEMAND ] [ [ START WITH (timestampz) ] | [ EVERY (interval) ] ] ]  
[ { ENABLE | DISABLE } QUERY REWRITE ]  
[ WITH ( {storage_parameter = value} [, ...] ) ]  
[ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH ( column_name [,...] ) } } ]  
[ PARTITION BY {(column_name) | date_trunc('time_unit', column_name)} ]  
AS query;
```

Parameter Description

Table 13-92 CREATE MATERIALIZED VIEW parameters

Parameter	Description	Value Range/Default Value
BUILD DEFERRED IMMEDIATE	IMMEDIATE indicates that the latest data is included when the materialized view is created. DEFERRED indicates that data is included only when the materialized view is refreshed for the first time.	-

Parameter	Description	Value Range/Default Value
REFRESH	Specifies the refresh mode of a materialized view. After a materialized view is created, the data in the materialized view reflects only the status of the base table at the creation time. When the data in the base table changes, you need to refresh the materialized view (REFRESH MATERIALIZED VIEW) to update the data in the materialized view.	- Refresh method: The options include DEFAULT (default refresh) and RESTRICT (forcible refresh). You can only specify the refresh method for materialized views that are asynchronously refreshed in the background. - Refresh direction: When using materialized views in a cascading nested manner, the refresh direction can be specified as FORWARD, BACKWARD, or ALL. You can only specify the refresh direction for materialized views that are asynchronously refreshed in the background. For details, see REFRESH MATERIALIZED VIEW. - Currently, only the COMPLETE refresh mode is supported, which refresh full data in the materialized view. Execute the query statement defined in the materialized view to update the materialized view. - Refresh triggering mode. ON DEMAND: manual refresh on demand. START WITH (timestamp tz) EVERY (interval): automatic refresh at a scheduled time. START WITH specifies the first refresh time. EVERY specifies the automatic refresh interval and its value can be MONTH, DAY, hour, MINUTE, or SECOND.

Parameter	Description	Value Range/Default Value
ENABLE DISABLE QUERY REWRITE	Specifies whether to enable query rewrite. Query rewrite means that when a base table is queried and a materialized view is created for the base table, the database system automatically determines whether the pre-calculated result in the materialized view can be used to process the query. If a materialized view can be used, the pre-calculated result is directly read from the materialized view to accelerate the query.	Disabled by default. When ENABLE QUERY REWRITE is specified, you need to configure the GUC parameter mv_rewrite_rule to enable query rewrite of materialized views.
WITH	Specifies the parameters for creating a materialized view.	For details, see Table 13-93 .

Parameter	Description	Value Range/Default Value
DISTRIBUTE BY	Specifies how the table is distributed or replicated between DNs.	Value range: REPLICATION, ROUNDROBIN, or HASH • REPLICATION: Each row in the table exists on all DNs, that is, each DN has complete table data. • ROUNDROBIN: Each row in the table is sent to each DN in turn. Therefore, data is evenly distributed on each DN. This value is supported only in 8.1.2 or later. • HASH: Each row of the table will be placed into all the DNs based on the hash value of the specified column. Default value: determined by the default_distribution_mode parameter. NOTE When the materialized view is distributed in hash mode, data skew may occur. To check for data skews in materialized views, follow the same procedures used for detecting data skews in regular tables. For details, see "Checking for Data Skew" in the <i>Data Warehouse Service (DWS) Developer Guide</i>. If data skew is identified in a materialized view, perform data skew optimization at the storage layer by referring to "Optimizing Data Skew" in the <i>Data Warehouse Service (DWS) Developer Guide</i>.

Parameter	Description	Value Range/Default Value
PARTITION BY	Creates a partitioned materialized view. This parameter is supported only by clusters of version 9.1.0.200 or later. CAUTION • The partition key of a materialized view must match that of the queried table's partitioned table to ensure partition-based mapping for invalidation and refresh. • The partition key of a materialized view must be included in the outermost SQL statement of the query to enable partition key-based updates to the materialized view. • Materialized views support only range partitioning. • The queried table can be a composite partition. If multiple partition tables have the same partition conditions, the mapping between the queried table and the materialized view is automatically established. • Partition mapping of foreign tables is not supported because data changes in foreign tables cannot be detected. • Columns that generate null values in outer joins cannot be used as partition keys. • Merging and splitting of base table partitions are not supported.	• Proportional association: Specify the partition key of a partitioned table in the query as the partition key of the materialized view, creating a one-on-one mapping relationship. • Roll-up association: Use the date_trunc function to roll up the partition key of a partitioned table in the query, establishing a roll-up mapping relationship. The data type should be compatible with the date_trunc function.
AS query	Creates a materialized view based on the query result.	-

Table 13-93 WITH parameters

Parameter	Description	Value Range
ORIENTATION	Specifies the storage mode (row-store, column-store) for table data. This parameter cannot be modified once it is set.	ROW or COLUMN • ROW indicates that table data is stored in rows. ROW applies to OLTP service, which has many interactive transactions. An interaction involves many columns in the table. Using ROW can improve the efficiency. • COLUMN indicates that the data is stored in columns. Column storage applies to the data warehouse service, which has a large amount of aggregation computing, and involves a few column operations. Default value: ROW
enable_foreign_table_query_rewrite	Specifies whether to allow query rewriting on materialized views that contain foreign tables. This parameter must be used together with ENABLE QUERY REWRITE . The materialized view cannot detect the data changes in the foreign table. Specify this option if you want to enable query rewriting for materialized views that contain foreign tables.	• on: allows query rewriting on materialized views that contain foreign tables. • off: does not allow query rewriting on materialized views that contain foreign tables. Default value: off
bitmap_columns	The bitmap index is only applicable to the hstore_opt table. To use it, enable the table-level parameter enable_hstore_opt and set bitmap_columns to the specified column. This is supported only by clusters of version 9.1.0.200 or later.	-

Parameter	Description	Value Range
secondary_part_num	Specifies the number of level-2 partitions in a column-store table. This parameter applies only to H-Store column-store tables. This is supported only by clusters of version 9.1.0.200 or later.	Value range: 1 to 32 Default value: 8
enable_hstore_opt	When the enable_hstore_opt table-level parameter is enabled, the enable_hstore table-level parameter is also automatically enabled by default. This is supported only by clusters of version 9.1.0.200 or later.	true or false Default value: false
enable_turbo_store	Determines whether to create a turbo table (column-store tables). The parameter is only valid for column-store tables. This is supported only by clusters of version 9.1.0.200 or later.	on or off Default value: off
mv_analyze_mode	Determines the automatic analysis method for materialized views. This is supported only by clusters of version 9.1.0.200 or later.	• none: indicates that the materialized view does not automatically execute ANALYZE after being refreshed. • light: indicates that the materialized view performs light analysis after being refreshed. Default value: light

Parameter	Description	Value Range
mv_pck_column	Specifies a partial clustering storage for a materialized view. During the data import process to a column-store table, the data is partially sorted according to the specified column(s). This is supported only by clusters of version 9.1.0.200 or later. Enable mv_pck_column and set it to a specified column.	-
mv_support_function_type	Enables the use of function attributes in the query statements when creating materialized views. This is supported only by clusters of version 9.1.0.200 or later.	• stable: indicates that functions of the STABLE and IMMUTABLE types can be used in query statements. • volatile: indicates that functions of the VOLATILE, STABLE, and IMMUTABLE types can be used in query statements. Default value: empty
excluded_inactive_tables	Specifies that the materialized view will not be invalidated when there are data changes in the underlying base table. This is supported only by clusters of version 9.1.0.200 or later. Set excluded_inactive_tables to schemaName1.tableName1,schemaName2.tableName2 .	Default value: empty
force_rewrite_timeout	Allows query rewrite within a specified time interval after refresh, regardless of the freshness of the data in the materialized view. This is supported only by clusters of version 9.1.0.200 or later.	The unit is second. The default value is 0 .

Parameter	Description	Value Range
TABLESPACE tablespace_name	Specifies a tablespace for V3 storage. If default_tablespace is empty, the default tablespace of the database is used. This is supported only by clusters of version 9.1.0.200 or later.	-

Examples

Create a base table and insert data into the base table.

```
CREATE TABLE t1 (a int, b int) DISTRIBUTE BY HASH(a);  
INSERT INTO t1 SELECT x,x FROM generate_series(1,10) x;
```

Create a materialized view with the default option **BUILD IMMEDIATE**.

```
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM t1;
```

Create a materialized view in column-store mode.

```
CREATE MATERIALIZED VIEW mv2 WITH(orientation = column) AS SELECT * FROM t1;
```

Create a materialized view that is manually refreshed as required.

```
CREATE MATERIALIZED VIEW mv3 BUILD DEFERRED REFRESH ON DEMAND AS SELECT * FROM t1;
```

Create a materialized view with a scheduled refresh time.

```
CREATE MATERIALIZED VIEW mv4 BUILD DEFERRED REFRESH START WITH(trunc(sysdate)) EVERY  
(interval '1 day') AS SELECT * FROM t1;
```

Create a materialized view with a bitmap index.

```
CREATE MATERIALIZED VIEW mv1  
with (ORIENTATION = COLUMN, enable_hstore=true, enable_hstore_opt=on, bitmap_columns='col1') AS  
SELECT * FROM base_table;
```

Create a materialized view and specify the number of level-2 partitions in a column-store table.

```
CREATE MATERIALIZED VIEW mv  
WITH (ORIENTATION=COLUMN, ENABLE_HSTORE=ON, enable_hstore_opt=on, mv_pck_column='c3',  
secondary_part_column = 'c2', secondary_part_num = 8) AS SELECT * FROM base_table;
```

Create a materialized view and specify the PCK column for sorting.

```
CREATE MATERIALIZED VIEW mv  
WITH (ORIENTATION=COLUMN, ENABLE_HSTORE=ON, enable_hstore_opt=on, mv_pck_column='col3') AS  
SELECT * FROM base_table;
```

Create a materialized view and specify the analysis method.

```
CREATE MATERIALIZED VIEW mv1 enable query rewrite  
with(excluded_inactive_tables='matview_basic."T1",matview_basic."a=b"',mv_analyze_mode='none') as  
SELECT * FROM base_table;
```

Create a V3 materialized view.


```
CREATE MATERIALIZED VIEW mv1
with (orientation=column, enable_hstore=true, compression=low, enable_hstore_opt=on, COLVERSION =
3.0) TABLESPACE cu_obs_tbs distribute by hash(scope_name)
AS SELECT * FROM dicttbl_low;
```

Create a materialized view that contains a foreign table for query rewriting.

```
CREATE MATERIALIZED VIEW mv1 with (enable_foreign_table_query_rewrite = true) as SELECT * FROM
base_table;
```

Create a materialized view and specify that volatile functions can be used in the query statement.

```
CREATE MATERIALIZED VIEW mv_date with(mv_support_function_type = 'volatile') as select to_date(a)
from t_date;
```

Helpful Links

[ALTER MATERIALIZED VIEW](#), [DROP MATERIALIZED VIEW](#), [REFRESH MATERIALIZED VIEW](#)

13.49 CREATE PROCEDURE

Function

CREATE PROCEDURE creates a stored procedure.

Stored procedures are sets of precompiled SQL statements that are stored in a database and repeatedly executed.

Precautions

- When creating a stored procedure, you need to explicitly specify the schema (for example, **schema.table**) that the table object belongs to, or there maybe an execution exception.
- **current_schema** and **search_path** specified by **SET** during the stored procedure creation are invalid. **search_path** and **current_schema** before and after function execution should be the same.
- When creating a stored procedure, it is not allowed to nest other aggregate functions or system functions outside the avg function.
- The precision values of the parameters or return values of a stored procedure are not checked.
- In a cluster with multiple CNs, the input or output parameters of a stored procedure cannot be set to the temporary table type. When a stored procedure is created on a CN that is not currently connected, the correct temporary schema cannot be obtained based on the table name, resulting in an incorrect table type.
- A stored procedure can have multiple return values or no return value. After a stored procedure without a return value is invoked, the command output is empty.
- If a stored procedure has output parameters, the **SELECT** statement uses the default values of the output parameters when calling the procedure. When the **CALL** statement calls the stored procedure, it requires that the output parameter values are adapted to Oracle. When the **CALL** statement calls a

non-overloaded function, output parameters must be specified. When the **CALL** statement calls an overloaded PACKAGE function, it can use the default values of the output parameters. For details, see the examples in [CALL](#).

- A stored procedure with the PACKAGE attribute can use overloaded functions.
- Stored procedures cannot be submitted by segment.
- The content in CREATE FUNCTION also applies to CREATE PROCEDURE. For details, see [CREATE FUNCTION](#). After a stored procedure is created, the definition of **CREATE FUNCTION** is returned when the definition is queried.

Syntax

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
[ ( {[ argmode ] [ argname ] argtype [ { DEFAULT | := | = } expression ]}[,...]) ]
[
  { IMMUTABLE | STABLE | VOLATILE }
  | { SHIPPABLE | NOT SHIPPABLE }
  | { PACKAGE }
  | [ NOT ] LEAKPROOF
  | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER | AUTHID
CURRENT_USER}
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { [ TO | = ] value | FROM CURRENT }
][ ... ]
{ IS | AS }
plsql_body
/
```

Parameter Description

Table 13-94 CREATE PROCEDURE parameters

Parameter	Description	Value Range
OR REPLACE	Replaces the original definition when two stored procedures are with the same name.	-
procedure_name	Specifies the name of the stored procedure that is created (optionally with schema names).	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range
argmode	Specifies the mode of the parameter. OUT and INOUT cannot be used in procedure definition of RETURNS TABLE .	IN , OUT , INOUT , or VARIADIC . • IN (default value): input parameter. • OUT: output parameter. Only this type can include VARIADIC. • INOUT: both input and output parameters. • OUT VARIADIC: output parameter, which is defined as an array type.
argname	Specifies the name of the parameter. There is no strict requirement on the sequence of argname and argmode . You are advised to use argname , argmode , and argtype in sequence.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
argtype	Specifies the type of the parameter.	A valid data type.
IMMUTABLE STABLE VOLATILE	Specifies a constraint. Parameters here are similar to those of CREATE FUNCTION . For details, see CREATE FUNCTION .	-

Parameter	Description	Value Range
SHIPPABLE NOT SHIPPABLE	Specifies whether a stored procedure can be pushed down to DNs for execution. • IMMUTABLE stored procedures can always be pushed down to DNs for execution. • STABLE and VOLATILE stored procedures can be pushed down to DNs for execution only when the SHIPPABLE attribute is used. SHIPPABLE indicates that the entire stored procedure will be pushed down to DNs for execution. If SHIPPABLE is incorrectly set, serious problems such as incorrect results may occur. SHIPPABLE, like IMMUTABLE, comes with usage limits. It requires stored procedures to exclude elements that prevent pushdown. Once pushed down to a DN, the stored procedures must rely solely on the DN's dataset in the internal computing. Examples: 1. If a stored procedure references a hash table, you cannot define the procedure as SHIPPABLE. 2. If factors such as stored procedures and syntax cannot	-

Parameter	Description	Value Range
	be pushed down, the stored procedure cannot be defined as SHIPPABLE. For details, see Optimizing Statement Pushdown. 3. If a stored procedure's calculation process involves data across DNs, the procedure cannot be defined as SHIPPABLE. For example, some aggregation operations involve data across DNs.	
PACKAGE	Specifies whether the stored procedure can be overloaded. PostgreSQL-style stored procedures can be overloaded, and this parameter is designed for Oracle-style stored procedures. • All PACKAGE and non-PACKAGE stored procedures cannot be overloaded or replaced. • Stored procedures with the PACKAGE attribute functions do not support parameters in the VARIADIC mode. • The PACKAGE attribute of a stored procedure cannot be modified.	-

Parameter	Description	Value Range
LEAKPROOF	Specifies that the parameters of the stored procedure include only return values. LEAKPROOF can be set only by the system administrator.	-
CALLED ON NULL INPUT	Declares that some parameters of the stored procedure can be called in normal mode if the parameter values are NULL . It can be omitted.	-
RETURNS NULL ON NULL INPUT STRICT	STRICT specifies that the stored procedure always returns NULL whenever any of its parameters are NULL . If it is specified, the stored procedure is not executed when there are NULL parameters; instead a NULL result is returned automatically. The usage of RETURNS NULL ON NULL INPUT is the same as that of STRICT .	-
EXTERNAL	The keyword EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all stored procedures not only external ones.	-
SECURITY INVOKER AUTHID CURRENT_USER	Specifies that the stored procedure is to be executed with the permissions of the user that calls it. It can be omitted. SECURITY INVOKER and AUTHID CURRENT_USER have the same functions.	-

Parameter	Description	Value Range
SECURITY DEFINER AUTHID DEFINER	Specifies that the stored procedure will be executed with the permissions of the user who created it. AUTHID DEFINER and SECURITY DEFINER have the same functions.	-
COST execution_cost	Estimates the execution cost of the stored procedure. The unit of execution_cost is cpu_operator_cost .	Positive
ROWS result_rows	Estimates the number of rows returned by the stored procedure. This is only allowed when the stored procedure is declared to return a set.	Its value is a positive number. The default value is 1000 .
configuration_parameter	Specifies parameter values configured in the stored procedure. • value Sets a specified database session parameter to a specified value. Value range: a string Specifies the default value. – DEFAULT: The setting will be used in new sessions. – OFF: The setting is disabled. – RESET: The setting will be used in new sessions. • from current Uses the value of configuration_parameter of the current session.	-

Parameter	Description	Value Range
plsql_body	Specifies the PL/SQL stored procedure body. When you create a user, or perform other operations requiring password input in a stored procedure, the system catalog and csv log records the unencrypted password. You are advised not to perform such operations in the stored procedure body.	-

Examples

- Create the stored procedure **prc_add**, where **param1** is an input parameter and **param2** is an input and output parameter. The procedure calculates the sum of **param1** and **param2**, assigns the result to **param2**, and outputs the result.

```
CREATE OR REPLACE PROCEDURE prc_add
(
    param1 IN INTEGER,
    param2 IN OUT INTEGER
)
AS
BEGIN
    param2:= param1 + param2;
    dbms_output.put_line('result is: '||to_char(param2));
END;
/
```

Call the stored procedure **prc_add**.

```
CALL prc_add(2,3);
```

- Create the stored procedure **pro_variadic**. The parameter mode of **param1** is **VARIADIC**. The integer array is converted to a TEXT format and then output via **param2**.

```
CREATE OR REPLACE PROCEDURE pro_variadic (param1 VARIADIC int4[],param2 OUT TEXT)
AS
BEGIN
    param2:= param1::text;
END;
/
```

Execute the stored procedure **pro_variadic**.

```
SELECT pro_variadic(VARIADIC param1=> array[1,2,3,4]);
```

- Create the stored procedure **package_func_overload** with the PACKAGE attribute.

```
CREATE OR REPLACE PROCEDURE package_func_overload(col int, col2 out varchar)
PACKAGE
AS
DECLARE
    col_type text;
BEGIN
    col2 := '122';
```



```
dbms_output.put_line('two varchar parameters ' || col2);  
END;  
/
```

Helpful Links

For details about how to delete a stored procedure, see [DROP PROCEDURE](#). For details about how to call a stored procedure, see [CALL](#).

13.50 CREATE PUBLICATION

Function

CREATE PUBLICATION adds a new publication to the current database. The publication name must be different from the name of any existing publication in the current database. A publication is essentially the replication of data changes in a set of tables achieved by logical replication.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.
- If neither **FOR TABLE** nor **FOR ALL TABLES** is specified, a publication starts with a set of empty tables. Tables can be added later.
- Creating a publication does not start replication. It defines only one group and filtering logic for future subscribers. To create a publication, the caller must have CREATE permission on the current database.
- To add a table to a publication, the caller must have ownership of the table. To use FOR ALL TABLES and FOR ALL TABLES IN SCHEMA clauses, the caller must have system administrator permissions.
- Do not add a table to the same publication by using FOR TABLE and FOR ALL TABLES IN SCHEMA.

Syntax

```
CREATE PUBLICATION name  
[ FOR ALL TABLES  
| FOR publication_object [ , ... ] ]  
[ WITH ( publication_parameter [=value] [ , ... ] ) ] ;
```

The syntax of using **publication_object** is as follows:

```
TABLE table_name [ , ... ]  
| ALL TABLES IN SCHEMA schema_name [ , ... ]
```

Parameter Description

Table 13-95 CREATE PUBLICATION parameters

Parameter	Description	Value Range
name	Specifies the name of a new publication.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
FOR ALL TABLES	Marks a publication as replications of all fine-grained DR primary table changes in the database, including tables to be created.	-
FOR TABLE	Specifies the list of tables to be added to a publication. Only the fine-grained DR primary table can be a part of the publication.	-
table_name	Name of the table to be added to the publication, which can include the schema name.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
FOR ALL TABLES IN SCHEMA	Marks a publication as replications of all fine-grained DR primary table changes a specified schema list, including tables to be created.	-
schema_name	The name of the schema to be added to the publication.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
WITH (publication_parameter [=value] [, ...])	Specifies optional parameters for a publication. Parameters in the right column are supported.	publish: Specifies the DML operations that will be published to the subscriber. Value range: A string. Separate the operations by commas (,). The available operations are insert, update, delete, and truncate. By default, all actions are released. Therefore, the default value of this option is insert, update, delete, and truncate .

Examples

- Create sample table **tpcds.ship_mode_t1**.

```
CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK      INTEGER      NOT NULL,
    SM_SHIP_MODE_ID      CHAR(16)     NOT NULL,
    SM_TYPE              CHAR(30)      ,
    SM_CODE              CHAR(10)      ,
    SM_CARRIER          CHAR(20)      ,
    SM_CONTRACT          CHAR(20)
) WITH (ORIENTATION = COLUMN,enable_disaster_cstore='on')
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

Create sample table **tpcds.customer_address_p1**.

```
CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK        INTEGER      NOT NULL,
    CA_ADDRESS_ID        CHAR(16)     NOT NULL,
    CA_STREET_NUMBER     CHAR(10)      ,
    CA_STREET_NAME       VARCHAR(60)   ,
    CA_STREET_TYPE       CHAR(15)      ,
    CA_SUITE_NUMBER      CHAR(10)      ,
    CA_CITY              VARCHAR(60)    ,
    CA_COUNTY            VARCHAR(30)    ,
    CA_STATE             CHAR(2)        ,
    CA_ZIP              CHAR(10)       ,
    CA_COUNTRY           VARCHAR(20)    ,
    CA_GMT_OFFSET        DECIMAL(5,2)  ,
    CA_LOCATION_TYPE     CHAR(20)
) WITH (ORIENTATION = COLUMN,enable_disaster_cstore='on')
DISTRIBUTE BY HASH(CA_ADDRESS_SK);
```

Create sample schema **myschema1**.

```
CREATE SCHEMA myschema1;
```

Create sample schema **myschema2**.

```
CREATE SCHEMA myschema2.
```

- Create a publication for all changes of two tables and two schemas.

```
CREATE PUBLICATION mypublication FOR TABLE users, departments, ALL TABLES IN SCHEMA myschema1, myschema2;
```
- Create a publication for all changes in all tables.

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

Helpful Links

[ALTER PUBLICATION](#), [DROP PUBLICATION](#)

13.51 CREATE REDACTION POLICY

Function

CREATE REDACTION POLICY creates a data redaction policy for a table.

Precautions

- The data masking feature is controlled by **feature_support_options**. If error information "REDACTION POLICY is not yet supported, please add

enable_data_redaction into feature_support_options." or "Cannot use the feature of data redaction, please drop existed redaction policy or add enable_data_redaction into feature_support_options." is displayed, the data masking feature is not enabled. In this case, you need to contact technical support to set **feature_support_options**.

- Only the table object owner and users who have been granted the **gs_redaction_policy** preset role are authorized to create data masking policies.
- Data masking policies can be created for regular tables and non-EXTERNAL-SCHEMA foreign tables (such as OBS, HDFS, GDS, and foreign tables for coordinated analysis), but not for system catalogs, EXTERNAL SCHEMA foreign tables, temporary tables, UNLOGGED tables, views, or functions.
- Synonyms cannot be used to create redaction policies for ordinary table objects.
- Table objects and redaction policies have a one-to-one mapping relationship. A redaction policy is a collection of data redaction functions that can be applied to multiple columns in a table. You can set different redaction functions for different columns.
- A redaction policy is enabled by default upon its creation, that is, the **enable** parameter of the policy is **true** by default.
- Redaction policies do not take effect on users with the **sysadmin** permission. Data in the redacted columns is always visible to such users.
- Data redaction policies can be matched with specified roles.

Syntax

```
CREATE REDACTION POLICY [IF NOT EXISTS] policy_name ON table_name [ { BEFORE | AFTER }  
old_policy_name ]  
[ WHEN (when_expression) ]  
[ ADD COLUMN column_name WITH redaction_function_name ( [ argument [, ...] ] ) ] [, ... ];
```

Parameter Description

Table 13-96 CREATE REDACTION POLICY parameters

Parameter	Description	Value Range
policy_name	Specifies the name of a redaction policy.	-
table_name	Specifies the name of the table to which the redaction policy is applied.	-

Parameter	Description	Value Range
BEFORE AFTER	Specifies the relative location where the current policy is created. Generally, one masking policy is used for one table. The default value indicates that the current policy is created after the last candidate policy of the target table recorded in the current system catalog.	-

Parameter	Description	Value Range
WHEN (when_expression)	Specifies the expression used for the redaction policy to take effect. The redaction policy takes effect only when this expression is true.	When a query statement is querying a table where a redaction policy is enabled, the redacted data is invisible in the query only if the WHEN expression for the redaction policy is true. Generally, the WHEN clause is used to specify the users for which the redaction policy takes effect. The WHEN clause must comply with the following rules: 1. The expression can be a combination of multiple subexpressions connected by AND and/or OR. 2. Each subexpression supports only the =, <>, !=, >=, >, <=, and < operators. The left and right operand values can only be constant values or one of the following system constant values: SESSION_USER, CURRENT_USER, USER, CURRENT_ROLE, and CURRENT_SCHEMA system constants or the SYS_CONTEXT system function. 3. Each subexpression can be an IN or NOT IN expression. The value for the left operand can be any of the system constant values listed in rule 2, and each element in the array of the right operand must be a constant value. 4. Each subexpression can be a pg_has_role(user, role, privilege) system function. 5. If you want a redaction policy to be valid in all conditions, that is, you want it to take effect on all users (including the table owner), you are advised to use the

Parameter	Description	Value Range
		(1=1) expression to create this policy. 6. If the WHEN clause is not specified, the redaction policy is disabled by default. You need to manually specify a WHEN expression for the policy to take effect.
column_name	Specifies the name of the table column to which the redaction policy is applied.	-
redaction_function_name	Specifies the redaction function applied to the specified table column.	-
arguments	Specifies the list of arguments of the redaction function. You can use the built-in masking functions MASK_NONE , MASK_FULL , and MASK_PARTIAL , or create your own masking functions by using the C language or PL/pgSQL. For details, see Data Redaction Functions .	• MASK_NONE: indicates that no masking is performed. • MASK_FULL: indicates that all data is masked to a fixed value. • MASK_PARTIAL: indicates that partial masking is performed based on the specified character type, numeric type, or time type.

Examples

Create redaction policy for a specified user.

1. Create users **alice** and **matu**.

```
CREATE ROLE alice PASSWORD '{password}';  
CREATE ROLE matu PASSWORD '{password}';
```
2. Create the table object **emp** as user **alice**, and insert data into the table.

```
CREATE TABLE emp(id int, name varchar(20), salary NUMERIC(10,2));  
INSERT INTO emp VALUES(1, 'July', 1230.10), (2, 'David', 999.99);
```
3. Create the redaction policy **mask_emp** for the **emp** table as user **alice** and make the **salary** column invisible to user **matu**.

```
CREATE REDACTION POLICY mask_emp ON emp WHEN(current_user = 'matu') ADD COLUMN salary  
WITH mask_full(salary);
```
4. Grant the **SELECT** permission on the **emp** table to user **matu** as user **alice**.

```
GRANT SELECT ON emp TO matu;
```
5. Switch to user **matu**.

```
SET ROLE matu PASSWORD '{password}';
```

6. Query the **emp** table. Data in the **salary** column has been redacted.

```
SELECT * FROM emp;
```

Create redaction policy for the role.

1. Create the role **redact_role**.

```
CREATE ROLE redact_role PASSWORD '{password}';
```
2. Add users **matu** and **alice** to the role **redact_role**.

```
GRANT redact_role to matu,alice;
```
3. Create a table object **emp1** as the administrator and insert data.

```
CREATE TABLE emp1(id int, name varchar(20), salary NUMERIC(10,2));  
INSERT INTO emp1 VALUES(3, 'Rose', 2230.20), (4, 'Jack', 899.88);
```
4. Create a redaction policy **mask_emp1** for the table object **emp1** as the administrator to make the **salary** column invisible to role **redact_role**.

```
CREATE REDACTION POLICY mask_emp1 ON emp1 WHEN (pg_has_role(current_user, 'redact_role',  
'member')) ADD COLUMN salary WITH mask_full(salary);
```

If no user is specified, the current user (**current_user**) is used by default.

```
CREATE REDACTION POLICY mask_emp1 ON emp1 WHEN (pg_has_role('redact_role', 'member'))  
ADD COLUMN salary WITH mask_full(salary);
```
5. Grant the **SELECT** permission on the **emp1** table to user **matu** as the administrator.

```
GRANT SELECT ON emp1 TO matu;
```
6. Switch to user **matu**.

```
SET ROLE matu PASSWORD '{password}';
```
7. Query the **emp** table. Data in the **salary** column has been redacted.

```
SELECT * FROM emp1;
```

Helpful Links

[ALTER REDACTION POLICY](#), [DROP REDACTION POLICY](#)

13.52 CREATE ROW LEVEL SECURITY POLICY

Function

CREATE ROW LEVEL SECURITY POLICY creates a row-level access control policy for a table.

The policy takes effect only after row-level access control is enabled (by running **ALTER TABLE... ENABLE ROW LEVEL SECURITY** | **ALTER FOREIGN TABLE ... ENABLE ROW LEVEL SECURITY**).

Currently, row-level access control affects the read (**SELECT**, **UPDATE**, **DELETE**) of data tables and does not affect the write (**INSERT** and **MERGE INTO**) of data tables. The table owner or system administrators can create an expression in the **USING** clause. When the client reads the data table, the database server combines the expressions that meet the condition and applies it to the execution plan in the statement rewriting phase of a query. For each tuple in a data table, if the expression returns **TRUE**, the tuple is visible to the current user; if the expression returns **FALSE** or **NULL**, the tuple is invisible to the current user.

A row-level access control policy name is specific to a table. A data table cannot have row-level access control policies with the same name. Different data tables can have the same row-level access control policy.

Row-level access control policies can be applied to specified operations (**SELECT**, **UPDATE**, **DELETE**, and **ALL**). **ALL** indicates that **SELECT**, **UPDATE**, and **DELETE** will be affected. For a new row-level access control policy, the default value **ALL** will be used if you do not specify the operations that will be affected.

Row-level access control policies can be applied to a specified user (role) or to all users (**PUBLIC**). For a new row-level access control policy, the default value **PUBLIC** will be used if you do not specify the user that will be affected.

Precautions

- Row-level access control policies can be defined for row-store tables, row-store partitioned tables, column-store tables, column-store partitioned tables, replication tables, unlogged tables, hash tables, and non-EXTERNAL SCHEMA foreign tables.
- Row-level access control policies cannot be defined for HDFS tables, EXTERNAL SCHEMA foreign tables, and temporary tables.
- Row-level access control policies cannot be defined for views.
- A maximum of 100 row-level access control policies cannot be defined for a table.
- Users with administrator permissions, initial O&M users (Ruby), table owners, and members of the table owner role are not affected by row-level access control and can view full data of the table.
- Tables queried by using SQL statements, views, functions, and stored procedures are affected by row-level access control policies.
- The type of a column on which a row-level access control policy depends cannot be changed. For example, the following modifications are not supported:

```
ALTER TABLE public.all_data ALTER COLUMN role TYPE text;
```

Syntax

```
CREATE [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
[ AS { PERMISSIVE | RESTRICTIVE } ]
[ FOR { ALL | SELECT | UPDATE | DELETE } ]
[ TO { role_name | PUBLIC } [, ...] ]
USING ( using_expression )
```

Parameter Description

Table 13-97 CREATE ROW LEVEL SECURITY POLICY parameters

Parameter	Description	Value Range
policy_name	Specifies the name of a row-level access control policy to be created. The names of row-level access control policies for a table must be unique.	-

Parameter	Description	Value Range
table_name	Specifies the name of a table to which a row-level access control policy is applied.	-
PERMISSIVE	Specifies that the row-level access control policy is to be created as a permissive policy. For a given query, all applicable permissive policies are combined using the OR operator. Row-level access control policies are permissive by default.	-
RESTRICTIVE	Specifies that the row-level access control policy is to be created as a restrictive policy. For a given query, all applicable restrictive policies are combined using the AND operator. At least one permissive policy is required to grant access to data records. If only restrictive policies are used, no records will be accessible. When both permissive and restrictive policies are used, a record is accessible only when it passes at least one permissive policy and all restrictive policies.	-

Parameter	Description	Value Range
command	Specifies the SQL operations affected by a row-level access control policy, including ALL , SELECT , UPDATE , and DELETE . If this parameter is not specified, the default value ALL will be used, covering SELECT , UPDATE , and DELETE .	If <i>command</i> is set to SELECT, only tuple data that meets the condition (the return value of <i>using_expression</i> is TRUE) can be queried. The operations that are affected include SELECT, UPDATE.... RETURNING, and DELETE... RETURNING. If <i>command</i> is set to UPDATE, only tuple data that meets the condition (the return value of <i>using_expression</i> is TRUE) can be updated. The operations that are affected include UPDATE, UPDATE ... RETURNING, and SELECT ... FOR UPDATE/SHARE. If <i>command</i> is set to DELETE, only tuple data that meets the condition (the return value of <i>using_expression</i> is TRUE) can be deleted. The operations that are affected include DELETE and DELETE ... RETURNING. The following table Table 13-98 describes the relationship between row-level access control policies and SQL statements.
role_name	Specifies database users affected by a row-level access control policy. NOTICE System administrators are not affected by row access control.	If this parameter is not specified, the default value PUBLIC will be used, indicating that all database users will be affected. You can specify multiple affected database users.

Parameter	Description	Value Range
using_expression	Specifies an expression defined for a row-level access control policy (return type: boolean).	The expression cannot contain aggregate functions and window functions. In the statement rewriting phase of a query, if row-level access control for a data table is enabled, the expressions that meet the specified conditions will be added to the plan tree. The expression is calculated for each tuple in the data table. For SELECT , UPDATE , and DELETE , row data is visible to the current user only when the return value of the expression is TRUE . If the expression returns FALSE , the tuple is invisible to the current user. In this case, the user cannot view the tuple through the SELECT statement, update the tuple through the UPDATE statement, or delete the tuple through the DELETE statement.

Table 13-98 Relationship between row-level security policies and SQL statements

Command	SELECT/ALL Policy	UPDATE/ALL Policy	DELETE/ALL Policy
SELECT	Existing row	No	No
SELECT FOR UPDATE/SHARE	Existing row	Existing row	No
UPDATE	No	Existing row	No
UPDATE RETURNING	Existing row	Existing row	No
DELETE	No	No	Existing row
DELETE RETURNING	Existing row	No	Existing row

Example 1: Create a Row-level Access Control Policy That the Current User Can Only View Its Own Data.

1. Create users **alice** and **bob**.

```
CREATE ROLE alice PASSWORD '{password}';  
CREATE ROLE bob PASSWORD '{password}';
```
2. Create the data table **public.all_data**.

```
CREATE TABLE public.all_data(id int, role varchar(100), data varchar(100));
```
3. Insert data into the table.

```
INSERT INTO all_data VALUES(1, 'alice', 'alice data');  
INSERT INTO all_data VALUES(2, 'bob', 'bob data');  
INSERT INTO all_data VALUES(3, 'peter', 'peter data');
```
4. Grant the read permission for the **all_data** table to users **alice** and **bob**.

```
GRANT SELECT ON all_data TO alice, bob;
```
5. Enable row-level access control.

```
ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;
```
6. Create a row-level access control policy to specify that the current user can view only its own data.

```
CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role = CURRENT_USER);
```
7. View information about the **all_data** table.

```
\d+ all_data
```

Column	Type	Modifiers	Storage	Stats target	Description
id	integer		plain		
role	character varying(100)		extended		
data	character varying(100)		extended		

Row Level Security Policies:
POLICY "all_data_rls"
USING (((role)::name = "current_user"()))

Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

8. Run **SELECT**.

```
SELECT * FROM all_data;  
id | role | data  
---+-----  
1 | alice | alice data  
2 | bob | bob data  
3 | peter | peter data  
(3 rows)  
EXPLAIN(COSTS OFF) SELECT * FROM all_data;  
QUERY PLAN  
-----  
Streaming (type: GATHER)  
Node/s: All datanodes  
-> Seq Scan on all_data  
(3 rows)
```

9. Switch to the **alice** user.

```
set role alice password '{password}';
```

10. Perform the **SELECT** operation.

```
SELECT * FROM all_data;  
id | role | data  
---+-----  
1 | alice | alice data  
(1 row)  
EXPLAIN(COSTS OFF) SELECT * FROM all_data;  
QUERY PLAN  
-----  
Streaming (type: GATHER)
```

```
Node/s: All datanodes  
-> Seq Scan on all_data  
Filter: ((role)::name = 'alice'::name)  
Notice: This query is influenced by row level security feature  
(5 rows)
```

Example 2: Partition Permission Management Through Row-Level Control

1. Create user **alice**.

```
CREATE ROLE alice PASSWORD '{password1}';
```
2. Create the range partitioned table **web_returns_p1** and insert data into the table.

```
CREATE TABLE web_returns_p1  
(  
    wr_returned_date_sk    integer,  
    wr_returned_time_sk    integer,  
    wr_item_sk             integer NOT NULL,  
    wr_refunded_customer_sk integer  
)  
WITH (orientation = column)  
DISTRIBUTE BY HASH (wr_item_sk)  
PARTITION BY RANGE(wr_returned_date_sk)  
(  
    PARTITION p2016 START(800) END(830) EVERY(1)  
);  
  
INSERT INTO web_returns_p1 values (801,17,11,102);  
INSERT INTO web_returns_p1 values (802,18,12,103);
```
3. Grant the read permission on the **web_returns_p1** table to user **alice**.

```
GRANT SELECT ON web_returns_p1 TO alice;
```
4. Enable row-level access control.

```
ALTER TABLE web_returns_p1 ENABLE ROW LEVEL SECURITY;
```
5. Create row-level access control policy **web_returns_rsl**. In the command, **wr_returned_date_sk** is a partition name of the **web_returns_p1** partition table, and **801** is the partition value.

```
CREATE ROW LEVEL SECURITY POLICY web_returns_rsl ON web_returns_p1  
USING('wr_returned_date_sk' = '801');
```
6. Impose the row-level access control policy **web_returns_rsl** on user **alice**.

```
ALTER ROW LEVEL SECURITY POLICY web_returns_rsl ON web_returns_p1 TO alice;
```
7. Switch to the **alice** user.

```
set role alice password '{password1}';
```
8. Query the **web_returns_p1** table.

```
select * from web_returns_p1;
```

Helpful Links

[ALTER ROW LEVEL SECURITY POLICY, DROP ROW LEVEL SECURITY POLICY](#)

13.53 CREATE RESOURCE POOL

Function

CREATE RESOURCE POOL creates a resource pool and specifies the Cgroup for the resource pool.

Precautions

As long as the current user has **CREATE** permission, it can create a resource pool.

Syntax

```
CREATE RESOURCE POOL pool_name  
[WITH ({MEM_PERCENT=pct | CONTROL_GROUP="group_name" | ACTIVE_STATEMENTS=stmt |  
MAX_DOP = dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='priority' |  
nodegroup='nodegroup_name' | is_foreign = boolean | except_rule='except_rule' | weight=bandwidth_weight  
| enable_concurrency_scaling=boolean}[ , ... ])];
```

Parameter Description

Table 13-99 CREATE RESOURCE POOL parameters

Parameter	Description	Value Range
pool_name	Specifies the name of a resource pool. The name of a resource pool cannot be same as that of an existing resource pool.	A string, which must comply with the identifier naming convention .

Parameter	Description	Value Range
group_name	Specifies the name of a Cgroup.	A string, which must comply with the rules in the description. It specifies the created Cgroup. - You can use either double quotation marks ("") or single quotation mark (') in the syntax when setting the name of a Cgroup. - The value of group_name is case-sensitive. - If group_name is not specified, the string "Medium" will be used by default in the syntax, indicating the Medium Timeshare Cgroup under DefaultClass. - If an administrator specifies a Workload Cgroup under Class, for example, control_group set to class1:workload1, the resource pool will be associated with the workload1 Cgroup under class1. The level of Workload can also be specified. For example, control_group is set to class1:workload1:1. - If a database user specifies the Timeshare string (Rush, High, Medium, or Low) in the syntax, for example, if control_group is set to High, the resource pool will be associated with the High Timeshare Cgroup under DefaultClass. - In multi-tenant scenarios, the Cgroup associated with a group resource pool is a Class Cgroup, and that associated with a service resource pool is a Workload Cgroup. Additionally, switching Cgroups between

Parameter	Description	Value Range
		different resource pools is not allowed.
stmt	Specifies the maximum number of statements that can be concurrently executed in a resource pool.	Numeral type. The value ranges from -1 to <i>INT_MAX</i> .
dop	Specifies the maximum number of simple SQL statements that can be concurrently executed in a resource pool.	Numeral type. The value ranges from 1 to <i>INT_MAX</i> .
memory_size	Specifies the estimated maximum memory for a resource pool.	A string. • default: The memory limit is half of the resource pool memory. • unlimited: no limit. • The value is from 1 KB to 2047 GB.
mem_percent	Specifies the proportion of available resource pool memory to the total memory or group user memory.	• In multi-tenant scenarios, mem_percent of group users or service users ranges from 1 to 100. The default value is 20. • In common scenarios, mem_percent of common users ranges from 0 to 100. The default value is 0.
io_limits	This parameter has been discarded in 8.1.2 and is reserved for compatibility with earlier versions.	-
io_priority	This parameter has been discarded in 8.1.2 and is reserved for compatibility with earlier versions.	-
nodegroup	Specifies the name of a logical cluster where the resource pool is. The logical cluster must already exist.	If the logical cluster name contains uppercase letters or special characters or begins with a digit, enclose the name with double quotation marks in SQL statements.

Parameter	Description	Value Range
is_foreign	In logical cluster mode, the current resource pool is used to control the resources of common users who are not associated with the logical cluster specified by nodegroup .	• nodegroup must specify an existing logical cluster, and cannot be elastic_group or the default Node Group (group_version1), which is generated during cluster installation. • If is_foreign is set to true, the resource pool cannot be associated with users. That is, CREATE USER... RESOURCE POOL cannot be used to configure resource pools for users. The resource pool automatically checks whether the users are associated with its logical cluster. If they are not, they will be controlled by the resource pool when performing operations on DNs in the logical cluster.
except_rule	Name of an exception rule set	-
weight	Specifies the network bandwidth weight of the resource pool.	-
enable_concurrency_scaling	Specifies whether to enable the elastic concurrency expansion function. This function is supported only by clusters of version 9.1.0.100 or later.	• true indicates that the elastic concurrent expansion function of jobs in the resource pool is enabled. • false indicates that the elastic concurrent expansion function of jobs in the resource pool is disabled. The default value is false .

Examples

This example assumes that Cgroups have been created by users in advance.

Create a default resource pool, and associate it with the **Medium** Timeshare Workload Cgroup under **DefaultClass**.

```
CREATE RESOURCE POOL pool1;
```

Create a resource pool, and associate it with the **High** Timeshare Workload Cgroup under **DefaultClass**.

```
CREATE RESOURCE POOL pool2 WITH (CONTROL_GROUP="High");
```

Create a resource pool, and associate it with the **Low** Timeshare Workload Cgroup under **class1**.

```
CREATE RESOURCE POOL pool3 WITH (CONTROL_GROUP="class1:Low");
```

Create a resource pool, and associate it with the **wg1** Workload Cgroup under **class1**.

```
CREATE RESOURCE POOL pool4 WITH (CONTROL_GROUP="class1:wg1");
```

Create a resource pool, and associate it with the **wg2** Workload Cgroup under **class1**.

```
CREATE RESOURCE POOL pool5 WITH (CONTROL_GROUP="class1:wg2:3");
```

Create a resource pool and enable the elastic concurrent expansion function of the resource pool.

```
CREATE RESOURCE POOL pool6 WITH (enable_concurrency_scaling=true);
```

Helpful Links

[ALTER RESOURCE POOL](#), [DROP RESOURCE POOL](#)

13.54 CREATE ROLE

Function

Create a role.

A role is an entity that has own database objects and permissions. In different environments, a role can be considered a user, a group, or both.

Precautions

- **CREATE ROLE** adds a role to a database. The role does not have the login permission.
- Only the user who has the **CREATE ROLE** permission or a system administrator is allowed to create roles.

Syntax

```
CREATE ROLE role_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The syntax of role information configuration clause **option** is as follows:

```
{SYSADMIN | NOSYSADMIN}  
| {AUDITADMIN | NOAUDITADMIN}  
| {CREATEDB | NOCREATEDB}  
| {USEFT | NOUSEFT}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCAADMIN}  
| CONNECTION LIMIT connlimit  
| VALID BEGIN 'timestamp'
```

```
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| TEMP SPACE 'tmpspacelimit'  
| SPILL SPACE 'spillspacelimit'  
| NODE GROUP logic_cluster_name  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN rol e_name [, ...]  
| USER role_name [, ...]  
| SYSID uid  
| DEFAULT TABLESPACE tablespace_name  
| PROFILE DEFAULT  
| PROFILE profile_name  
| PGUSER  
| AUTHINFO 'authinfo'  
| PASSWORD EXPIRATION period
```

Parameter Description

Table 13-100 CREATE ROLE parameters

Parameter	Description	Value Range
role_name	Role name	A string. It must comply with the naming convention and can contain a maximum of 63 characters. For details, see Identifier Naming Conventions .
password	Specifies the login password.	A password must: - Contain at least eight characters. This is the default length. - Differ from the user name or the user name spelled backwards. - Contains at least three of the following four character types: uppercase letters, lowercase letters, digits, and special characters, including: ~!@#\$\$%^&*()-_+=\ [{}];,;<.>/?. If you use characters other than the four types, a warning is displayed, but you can still create the password. Value range: a string

Parameter	Description	Value Range
DISABLE	By default, you can change your password unless it is disabled. Use this parameter to disable the password of a user.	After the password of a user is disabled, the password will be deleted from the system. The user can connect to the database only through external authentication, for example, IAM, Kerberos, LDAP, or oneaccess. Only administrators can enable or disable a password. Common users cannot disable the password of an initial user. To enable a password, run ALTER USER and specify the password.
ENCRYPTED UNENCRYPTED	Determines whether the password stored in the system will be encrypted.	(If neither is specified, the password status is determined by password_encryption_type .) According to product security requirements, the password must be stored encrypted. Therefore, UNENCRYPTED is forbidden in DWS. If the password is encrypted using SHA256, it will be stored as it is, regardless of whether it is specified as ENCRYPTED or UNENCRYPTED . This is because the system cannot decrypt the specified encrypted password. This allows reloading of the encrypted password during dump/restore.
SYSADMIN NOSYSADMIN	Determines whether a new role is a system administrator. Roles having the SYSADMIN attribute have the highest permission.	The default value is NOSYSADMIN.
AUDITADMIN NOAUDITADMIN	Determines whether a role has the audit and management attributes.	The default value is NOAUDITADMIN.
CREATEDB NOCREATEDB	Defines a role's ability to create databases. A new role does not have the permission to create databases.	The default value is NOCREATEDB.

Parameter	Description	Value Range
USEFT NOUSEFT	Determines whether a new role can perform operations on foreign tables, such as creating, deleting, modifying, and reading/writing foreign tables. A new role does not have permissions for these operations.	The default value is NOUSEFT . NOTE In cluster 8.2.0 or later, if foreign_table_options of the GUC parameter security_enable_options is enabled, foreign table operations are permitted and the useft permission does not need to be granted to users.
CREATEROLE NOCREATEROLE	Determines whether a role will be permitted to create new roles (that is, execute CREATE ROLE and CREATE USER). A role with the CREATEROLE permission can also modify and delete other roles.	The default value is NOCREATEROLE.
INHERIT NOINHERIT	Determines whether a role "inherits" the permissions of roles it is a member of. You are not advised to execute them.	-
LOGIN NOLOGIN	Determines whether a role is allowed to log in to a database. A role having the LOGIN attribute can be thought of as a user.	The default value is NOLOGIN.
REPLICATION NOREPLICATION	Determines whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the REPLICATION attribute is a highly privileged role, and should only be used on roles used for replication.	The default value is NOREPLICATION.

Parameter	Description	Value Range
INDEPENDENT NOINDEPENDENT	It defines a private and independent role. For a role with the INDEPENDENT attribute, administrators' rights to control and access this role are separated.	The procedure is as follows: • Administrators have no rights to add, delete, query, modify, copy, or authorize the corresponding table objects without the authorization from the INDEPENDENT role. • Administrators have no rights to modify the inheritance relationship of the INDEPENDENT role without the authorization from this role. • Administrators have no rights to modify the owner of the table objects for the INDEPENDENT role. • Administrators have no rights to delete the INDEPENDENT attribute of the INDEPENDENT role. • Administrators have no rights to change the database password of the INDEPENDENT role. The INDEPENDENT role must manage its own password, which cannot be reset if lost. • The SYSADMIN attribute of a user cannot be changed to the INDEPENDENT attribute.

Parameter	Description	Value Range
VCADMIN NOVCADMIN	Defines the role of a logical cluster administrator. A logical cluster administrator has the following more permissions than common users: • Create, modify, and delete resource pools in the associated logical cluster. • Grant the access permission for the associated logical cluster to other users or roles, or reclaim the access permission from those users or roles.	-
CONNECTION LIMIT	Indicates how many concurrent connections the role can use on a single CN.	An integer that is greater than or equal to -1 . Default value: -1 , indicating there is no limit. NOTICE To ensure the proper running of a cluster, the minimum value of CONNECTION LIMIT is the number of CNs in the cluster, because when a cluster runs ANALYZE on a CN, other CNs will connect to the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set CONNECTION LIMIT to 3 or a larger value.
VALID BEGIN	Sets a date and time when the role's password becomes valid. If this clause is omitted, the password will be valid for all time.	-
VALID UNTIL	Sets a date and time after which the role's password is no longer valid. If this clause is omitted, the password will be valid for all time.	-

Parameter	Description	Value Range
RESOURCE POOL	Sets the name of resource pool used by the role, and the name belongs to the system catalog: pg_resource_pool .	-
USER GROUP 'groupuser'	Creates a sub-user.	-
PERM SPACE	Sets the storage space of the user permanent table.	space_limit : specifies the upper limit of the storage space of the permanent table. Value range: A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.
TEMP SPACE	Sets the storage space of the user temporary table.	tmpspacelimit : specifies the storage space limit of the temporary table. Value range: A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.
SPILL SPACE	Sets the operator disk flushing space of the user.	spillspacelimit : specifies the operator spilling space limit. Value range: A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.
NODE GROUP	Specifies the name of the logical cluster associated with a user. If the name contains uppercase characters or special characters, enclose the name with double quotation marks.	-
IN ROLE	Lists one or more existing roles whose permissions will be inherited by a new role. You are not advised to use them.	-
IN GROUP	Indicates an obsolete spelling of IN ROLE . You are not advised to use them.	-

Parameter	Description	Value Range
ROLE	Lists one or more existing roles which are automatically added as members of the new role.	-
ADMIN	Is similar to ROLE . However, the roles after ADMIN can grant rights of new roles to other roles.	-
USER	Indicates an obsolete spelling of the ROLE clause.	-
SYSID	The SYSID clause is ignored.	-
DEFAULT TABLESPACE	The DEFAULT TABLESPACE clause is ignored.	-
PROFILE	The PROFILE clause is ignored.	-
PGUSER	This attribute is used to be compatible with open-source Postgres communication. An open-source Postgres client interface (Postgres 9.2.19 is recommended) can use a database user having this attribute to connect to the database.	This attribute only ensures compatibility with the connection process. Incompatibility caused by kernel differences between this product and Postgres cannot be solved using this attribute. Users having the PGUSER attribute are authenticated in a way different from other users. Error information reported by the open-source client may cause the attribute to be enumerated. Therefore, you are advised to use a client of this product. Example: <pre># normaluser is a user that does not have the PGUSER attribute. psql is the Postgres client tool. pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser psql: authentication method 10 not supported</pre> # pguser is a user having the PGUSER attribute. pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser Password for user pguser:

Parameter	Description	Value Range
AUTHINFO 'authinfo'	This attribute is used to specify the role authentication type. authinfo is the description character string, which is case sensitive. Currently, only the LDAP and OneAccess type is supported. Its description character string is ldap and oneaccess . LDAP and OneAccess are external authentication modes. Therefore, PASSWORD DISABLE must be specified.	• Additional information about LDAP authentication can be added to authinfo, for example, fulluser in LDAP authentication, which is equivalent to ldapprefix+username+ldapsuffix. If the content of authinfo is ldap, the role authentication type is LDAP. In this case, the ldapprefix and ldapsuffix information is provided by the corresponding record in the pg_hba.conf file. • When the OneAccess authentication mode is supported, authinfo must contain oneaccessClientId and domain information in the 'oneaccessClientId=xxxx, domain=xxxx' format. oneaccessClientId indicates the ID of the oneaccess user and contains a maximum of 256 characters. domain indicates the access domain name of the oneaccess user and contains a maximum of 64 characters. • When executing the ALTER ROLE command, users are not allowed to change the authentication type. Only LDAP or OneAccess users are allowed to modify the LDAP or OneAccess attributes. • OneAccess is identified by internal code logic and does not need to be configured in pg_hba.conf. The authentication information is provided by authinfo.

Parameter	Description	Value Range
PASSWORD EXPIRATION period	Number of days before the login password of the role expires. The user needs to change the password in time before the login password expires. If the login password expires, the user cannot log in to the system. In this case, the user needs to ask the administrator to set a new login password.	An integer ranging from -1 to 999 . The default value is -1 , indicating that there is no restriction. The value 0 indicates that the login password expires immediately.

Examples

Create the role **manager**.

```
CREATE ROLE manager IDENTIFIED BY '{password}';
```

Create a role with a validity period from January 1, 2015 to January 1, 2026.

```
CREATE ROLE miriam WITH LOGIN PASSWORD '{password}' VALID BEGIN '2015-01-01' VALID UNTIL '2026-01-01';
```

Create a role. The authentication type is LDAP. Other LDAP authentication information is provided by **pg_hba.conf**:

```
CREATE ROLE role1 WITH LOGIN AUTHINFO 'ldap' PASSWORD DISABLE;
```

Create a role. The authentication type is LDAP. The **fulluser** information for LDAP authentication is specified during the role creation. In this case, LDAP is case sensitive and must be enclosed in single quotation marks.

```
CREATE ROLE role2 WITH LOGIN AUTHINFO 'ldapcn=role2,cn=user,dc=lework,dc=com' PASSWORD DISABLE;
```

Create a role and set the authentication type to **OneAccess**. The authinfo information of OneAccess authentication is specified during the role creation. In this case, **oneaccess** is case sensitive and must be enclosed in single quotation marks.

```
CREATE ROLE role3 WITH LOGIN AUTHINFO 'oneaccessClientId=AbCd123,domain=example.exampleoneaccess.com' PASSWORD DISABLE;
```

Create a role and set the validity period of its login password to 30 days.

```
CREATE ROLE role4 WITH LOGIN PASSWORD '{password}' PASSWORD EXPIRATION 30;
```

Helpful Links

[SET ROLE](#), [ALTER ROLE](#), [DROP ROLE](#), [GRANT](#), [REVOKE](#)

13.55 CREATE SCHEMA

Function

CREATE SCHEMA creates a schema.

Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). When creating named objects, you can also use the schema name as a prefix.

Optionally, **CREATE SCHEMA** can include sub-commands to create objects within the new schema. The sub-commands are treated essentially the same as separate commands issued after creating the schema. If the **AUTHORIZATION** clause is used, all the created objects are owned by this user.

Precautions

- As long as the current user has the CREATE permission, the user can create a schema.
- The owner of an object created by a system administrator in a schema with the same name as a common user is the common user, not the system administrator.

Syntax

- Create a schema based on a specified name:

```
CREATE SCHEMA schema_name  
[ AUTHORIZATION user_name ] [ WITH PERM SPACE 'space_limit' ] [ schema_element [ ... ] ];  
CREATE SCHEMA IF NOT EXISTS schema_name  
[ AUTHORIZATION user_name ] [ WITH PERM SPACE 'space_limit' ] ;
```
- Create a schema based on a user name:

```
CREATE SCHEMA AUTHORIZATION user_name [ WITH PERM SPACE 'space_limit' ] [ schema_element  
[ ... ] ];  
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION user_name [ WITH PERM SPACE 'space_limit' ] ;
```

Parameter Description

Table 13-101 CREATE SCHEMA parameters

Parameter	Description	Value Range
schema_name	Indicates the name of the schema to be created. NOTICE - The name must be unique, - and cannot start with pg_.	A string, which must comply with the identifier naming convention. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range
AUTHORIZATION user_name	Indicates the name of the user who will own this schema. If schema_name is not specified, user_name will be used as the schema name. In this case, user_name can only be a role name.	Name of an existing user or role.
WITH PERM SPACE 'space_limit'	Indicates the storage upper limit of the permanent table in the specified schema. If space_limit is not specified, the space is not limited.	A string consists of an integer and unit. The unit can be K/M/G/T/P currently. The unit of parsed value is K and cannot exceed the range that can be expressed in 64 bits, which is 1 KB to 9007199254740991 KB.
schema_element	Indicates an SQL statement defining an object to be created within the schema. Currently, only CREATE TABLE , CREATE VIEW , CREATE INDEX , CREATE PARTITION , and GRANT are accepted as clauses within CREATE SCHEMA . Objects created by sub-commands are owned by the user specified by AUTHORIZATION .	-

Parameter	Description	Value Range
IF NOT EXISTS	If IF NOT EXISTS is specified and no schema with the same name exists, a schema can be created. If a schema with the same name already exists during schema creation, the system will not report an error but will display a message indicating that the schema already exists and no further operations will be performed. When this parameter is used, the Schema_element subcommand cannot be included. The IF NOT EXISTS parameter is supported only in 9.1.0 and later versions.	If objects in the schema on the current search path are with the same name, specify the schemas different objects are in. You can run the SHOW SEARCH_PATH command to check the schemas on the current search path.

Examples

Create the role **role1**.

```
CREATE ROLE role1 IDENTIFIED BY '{password}';
```

-- Create the schema **role1** for the role **role1**. The owner of the **films** and **winners** tables created by the clause is **role1**.

```
CREATE SCHEMA AUTHORIZATION role1
  CREATE TABLE films (title text, release date, awards text[])
  CREATE VIEW winners AS
  SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Helpful Links

[ALTER SCHEMA](#), [DROP SCHEMA](#)

13.56 CREATE SEQUENCE

Function

CREATE SEQUENCE adds a sequence to the current database.

A sequence is a database object that generates a series of unique, ascending, or descending values. It is mainly used to generate the primary key or unique identifier in a table.

Precautions

- The user who creates a sequence is the owner of the sequence.
- If a schema name is specified, the sequence is created in the specified schema. Otherwise, the text search configuration is created in the current schema. The sequence name must be different from the names of other sequences, tables, indexes, views in the same schema.
- After the sequence is created, functions `nextval()` and `generate_series(1,N)` insert data to the table. Make sure that the number of times for invoking `nextval` is greater than or equal to `N+1`. Otherwise, errors will be reported because the number of times for invoking function `generate_series()` is `N+1`.
- A sequence cannot be created in the **template1** database.

Syntax

```
CREATE SEQUENCE [ IF NOT EXISTS ] name [ INCREMENT [ BY ] increment ]  
    [ MINVALUE minvalue | NO MINVALUE | NOMINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE |  
    NOMAXVALUE ]  
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE | NOCYCLE ]  
    [ OWNED BY { table_name.column_name | NONE } ];
```

Parameter Description

Table 13-102 CREATE SEQUENCE parameters

Parameter	Description	Value Range
IF NOT EXISTS	If IF NOT EXISTS is specified and a sequence with the same name does not exist, the sequence can be created successfully. If a sequence with the same name already exists during sequence creation, the system will display a message indicating that the sequence already exists and no further operations will be performed. No error will be reported. This parameter is supported only by 9.1.0 and later versions.	-
name	Specifies the name of the sequence to be created.	A string compliant with the identifier naming rules .

Parameter	Description	Value Range
increment	Specifies the step for a sequence. If the increment is a positive value, an increasing sequence is to be created. If the increment is a negative value, a decreasing sequence is to be created. The default value is 1 .	-
MINVALUE minvalue NO MINVALUE NOMINVALUE	Specifies the minimum value that can be generated by the sequence. If MINVALUE is not specified, or NO MINVALUE is specified, the default value of the ascending sequence is 1 , and that of the descending sequence is -2⁶³-1 . NOMINVALUE is equivalent to NO MINVALUE .	-
MAXVALUE maxvalue NO MAXVALUE NOMAXVALUE	Specifies the maximum value that can be generated by the sequence. If MAXVALUE is not declared or NO MAXVALUE is declared, the default value of the ascending sequence is 2⁶³-1 , and that of the descending sequence is -1 . NOMAXVALUE is equivalent to NO MAXVALUE .	-
start	Specifies the start value of the sequence. The default value for ascending sequences is minvalue and for descending sequences maxvalue .	-

Parameter	Description	Value Range
cache	Specifies the number of sequence numbers that are cached (pre-allocated) to the memory in a database for faster access. Within a cache period, the CN does not request a sequence number from the GTM. Instead, the CN uses the sequence number that is locally applied for in advance. In cluster versions 9.1.0.100 and later, the default value is specified by the GUC parameter default_sequence_cache. • In a newly installed cluster of 9.1.0.100 or later, the default value is 20. • If the cluster is upgraded from an earlier version to 9.1.0.100 or later, the default value is 1. Indicates that only one value can be generated at a time, that is, no cache.	• It is not recommended that you define cache, and maxvalue, and minvalue at the same time. The continuity of sequences cannot be ensured after cache is defined because unacknowledged sequences may be generated, wasting sequence number segments. • You are advised not to set a large value for cache (less than 100000000). Otherwise, it takes a long time to cache the sequence number (the first NEXTVAL in each cache period). Set a proper value for cache based on services to ensure quick access without wasting sequence numbers.

Parameter	Description	Value Range
CYCLE NOCYCLE	Recycles sequences after the number of sequences reaches maxvalue or minvalue. The default value is NO CYCLE, indicating that no new value can be generated after the sequence reaches the maximum or minimum value. NOCYCLE is equivalent to NO CYCLE. • If the sequence is defined as CYCLE, the sequence uniqueness cannot be ensured. • If NO CYCLE is specified, any invocation of nextval would return an error after the sequence reaches its maximum value.	-
OWNED BY	Associates a sequence with a specified column of a specific table. If the column or the table where the column is located is deleted, the sequence is automatically deleted. The associated table and sequence must be owned by the same user and in the same schema. If the default value is OWNED BY NONE, indicating that such association does not exist.	• OWNED BY only establishes the association between a table column and the sequence. The sequence is not created for this column. • You are not advised to use the sequence created using OWNED BY in other tables. If multiple tables need to share a sequence, the sequence must not belong to a specific table.

Examples

Create an ascending sequence named **serial**, which starts from 101:

```
CREATE SEQUENCE serial
START 101
CACHE 20;
```

Select the next number from the sequence:

```
SELECT nextval('serial');
nextval
-----
      101
```

Select the next number from the sequence:

```
SELECT nextval('serial');
nextval
-----
      102
```

Create a sequence associated with the table:

```
CREATE TABLE customer_address
(
  ca_address_sk      integer      not null,
  ca_address_id      char(16)     not null,
  ca_street_number   char(10)     ,
  ca_street_name     varchar(60)  ,
  ca_street_type     char(15)     ,
  ca_suite_number    char(10)     ,
  ca_city            varchar(60)  ,
  ca_county          varchar(30)  ,
  ca_state           char(2)      ,
  ca_zip             char(10)     ,
  ca_country         varchar(20)  ,
  ca_gmt_offset      decimal(5,2) ,
  ca_location_type   char(20)
);

CREATE SEQUENCE serial1
START 101
CACHE 20
OWNED BY customer_address.ca_address_sk;
```

Use **SERIAL** to create a serial table **serial_table** for primary key auto-increment.

```
CREATE TABLE serial_table(a int, b serial);
INSERT INTO serial_table (a) VALUES (1),(2),(3);
SELECT * FROM serial_table ORDER BY b;
a | b
---+---
1 | 1
2 | 2
3 | 3
(3 rows)
```

Helpful Links

[ALTER SEQUENCE](#), [DROP SEQUENCE](#)

13.57 CREATE SERVER

Function

CREATE SERVER creates an external server.

An external server stores the HDFS cluster information, OBS server information, DLI connection information, or other homogeneous cluster information.

Precautions

By default, only the system administrator can create a foreign server. Otherwise, creating a server requires USAGE permission on the foreign-data wrapper being used. The syntax is as follows:

```
GRANT USAGE ON FOREIGN DATA WRAPPER fdw_name TO username;
```

fdw_name is the name of **FOREIGN DATA WRAPPER**, and **username** is the user name of creating **SERVER**.

Syntax

```
CREATE SERVER server_name  
  FOREIGN DATA WRAPPER fdw_name  
  OPTIONS ( { option_name ' value ' } [, ...] ) ;
```

Parameter Description

Table 13-103 CREATE SERVER parameters

Parameter	Description	Value Range
server_name	Name of the foreign server to be created. The server name must be unique in a database.	A string, which must contain up to 63 characters and comply with the identifier naming conventions .
FOREIGN DATA WRAPPER fdw_name	Specifies the name of the foreign data wrapper.	fdw_name is the data wrapper created by the system during database initialization. - For an HDFS cluster, fdw_name can be hdfs_fdw or dfs_fdw. - For other homogeneous clusters, fdw_name is gc_fdw. For data import and export, the GDS foreign table uses gsmpp_server, and fdw_name is dist_fdw.
OPTIONS	Specifies the parameters of foreign servers.	For details about the supported parameters, see Table 13-104 .

Table 13-104 OPTIONS parameters

Parameter	Description	Value Range
address	Specifies the IP address of the OBS service endpoint or HDFS cluster.	OBS: Specifies the endpoint of the OBS service. HDFS: Specifies the IP address and port number of a NameNode (metadata node) in the HDFS cluster, or the IP address and port number of a CN in other homogeneous clusters. HDFS NameNodes are deployed in primary/secondary mode for HA. Add the addresses of the primary and secondary NameNodes to ADDRESS. When accessing HDFS, DWS dynamically checks for the primary NameNode. If the HDFS is in federation mode, you can add all router addresses to the address value. When DWS accesses the HDFS service, the system dynamically and randomly searches for the active router. NOTE • The address option must exist. In the cross-cluster interconnection scenario, only one address option can be set. • If the server type is DLI, the address is the OBS address stored on DLI. • If the HDFS is in federation mode, that is, fed 'rbf', address can be set to multiple groups of IP addresses and ports, corresponding to the address of the HDFS router.

Parameter	Description	Value Range
hdfscfgpath	This parameter is available only when type is HDFS. You can set the hdfscfgpath parameter to specify the HDFS configuration file path. DWS accesses the HDFS cluster based on the connection configuration mode and security mode specified in the HDFS configuration file stored in that path. If the HDFS cluster is connected in non-secure mode, data transmission encryption is not supported. If the address option is not specified, the address specified by hdfscfgpath in the configuration file is used by default.	-
fed	Indicates that dfs_fdw is connected to HDFS in federation mode.	The value rbf indicates that HDFS uses the RBF mode. NOTE This feature is supported in 8.1.2 or later.
encrypt	Specifies whether data is encrypted. This parameter is available only when type is OBS . The default value is off .	• on indicates that data is encrypted. • off indicates that data is not encrypted.
access_key	Specifies the access key (AK) (obtained by users from the OBS console) used for the OBS access protocol. When you create a foreign table, its AK value is encrypted and saved to the metadata table of the database.	• If FOREIGN DATA WRAPPER is set to dfs_fdw, this parameter can be set only when type is set to OBS. • For clusters of version 8.2.0 or later, this parameter can be specified when FOREIGN DATA WRAPPER is set to dist_fdw.

Parameter	Description	Value Range
secret_access_key	Indicates the secret access key (SK) (obtained by users from the OBS page) used for the OBS access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database.	• If FOREIGN DATA WRAPPER is set to dfs_fdw, this parameter can be set only when type is set to OBS. • For clusters of 8.2.0 or later, this parameter can be specified when FOREIGN DATA WRAPPER is set to dist_fdw.
security_token	Corresponds to the SecurityToken value of the temporary security credential in IAM. A temporary AK, a temporary SK, and a temporary security token form a temporary security credential. The temporary security credential is valid for no more than 24 hours. This parameter is supported by version 8.2.0 or later clusters.	• If FOREIGN DATA WRAPPER is set to dfs_fdw, this parameter can be set only when type is set to OBS. • For clusters of 8.2.0 or later, this parameter can be specified when FOREIGN DATA WRAPPER is set to dist_fdw. • When this parameter is used, access_key and secret_access_key correspond to the temporary AK and SK, respectively.
type	Specifies the dfs_fdw connection type.	• OBS indicates that OBS is connected. • HDFS indicates that HDFS is connected. • DLI indicates that DLI is connected.
dli_address	Specifies the endpoint of the DLI service. This parameter is available only when type is DLI .	-

Parameter	Description	Value Range
dli_access_key	Specifies the access key (AK) (obtained by users from the DLI console) used for the DLI access protocol. When you create a foreign table, its AK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is DLI .	-
dli_secret_access_key	Specifies the secret access key (SK) (obtained by users from the DLI console) used for the DLI access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database. This parameter is available only when type is DLI .	-
dbname	Specifies the database name of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection.	-
username	Specifies the username of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection.	-

Parameter	Description	Value Range
password	Specifies the password of a remote cluster to be connected. This parameter is used for collaborative analysis and cross-cluster interconnection. NOTE When an on-premises cluster is migrated to the cloud, the password in the server configuration exported from the on-premises cluster is in ciphertext. The encryption and decryption keys of the on-premises cluster are different from those of the cloud cluster. Therefore, if CREATE SERVER is executed on the cloud cluster, the execution fails and a decryption failure error is reported. In this case, you need to manually change the password in CREATE SERVER to a plaintext password.	-
syncsrv	This parameter is used only for cross-cluster interconnection and indicates the GDS service used during data synchronization. The method for setting this parameter is the same as that for setting the location attribute of the GDS foreign table.	-

Examples

Create the **hdfs_server** server, in which **hdfs_fdw** is the built-in foreign data wrapper. (The values of **address** and **hdfscfgpath** are for reference only. Change them as required.)

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
  (address '10.10.0.100:25000,10.10.0.101:25000',
   hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
   type 'HDFS'
  );
```

Create the **obs_server** server, in which **dfs_fdw** is the foreign-data wrapper.

```
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (  
  address 'obs.eu-west-101.myhuaweicloud.eu',  
  access_key 'xxxxxxx',  
  secret_access_key 'yyyyyyyyyyyyyy',  
  type 'obs'  
);
```

Create the **dli_server** server, in which **dfs_fdw** is the foreign-data wrapper.

```
CREATE SERVER dli_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (  
  address 'obs.eu-west-101.myhuaweicloud.eu',  
  access_key 'xxxxxxx',  
  secret_access_key 'yyyyyyyyyyyyyy',  
  type 'dli',  
  dli_address 'dli.example.com',  
  dli_access_key 'xxxxxxx',  
  dli_secret_access_key 'yyyyyyyyyyyyyy'  
);
```

Create another server in the homogeneous cluster, where **gc_fdw** is the foreign data wrapper in the database.

```
CREATE SERVER server_remote FOREIGN DATA WRAPPER GC_FDW OPTIONS  
  (address '10.10.0.100:25000,10.10.0.101:25000',  
  dbname 'test',  
  username 'test',  
  password 'xxxxxxx'  
);
```

Create a server whose **FOREIGN DATA WRAPPER** is **dist_fdw** for importing and exporting text data on OBS.

```
CREATE SERVER import_server FOREIGN DATA WRAPPER DIST_FDW OPTIONS  
(  
  access_key 'ak_string',  
  secret_access_key 'sk_string'  
);
```

Helpful Links

[ALTER SERVER](#), [DROP SERVER](#)

13.58 CREATE SUBSCRIPTION

Function

CREATE SUBSCRIPTION adds a new subscription to the current database. The subscription name must be different from a name of any existing subscription in the database. A subscription represents a replication for connecting to the publication side.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.
- A subscription can be created only by the system administrator.

Syntax

```
CREATE SUBSCRIPTION name  
  CONNECTION 'conninfo'  
  PUBLICATION publication_name [, ...]  
  [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

Parameter Description

Table 13-105 CREATE SUBSCRIPTION parameters

Parameter	Description	Value Range
name	Specifies the name of a subscription.	A string, which must comply with the identifier naming conventions .

Parameter	Description	Value Range
conninfo	Specifies the string for connecting to the publication side. Example: host=1.1.1.1,2.2.2.2 port=10000,20000 dbname=postgres user=repusr1 password=password_123	• host IP address of the publication side. You can specify the IP addresses of both the primary and standby nodes of the publication side. Separate the IP addresses with commas (,). • port The publication port cannot be the primary port. Instead, it must be the primary port number plus 1. Otherwise, the port number conflicts with the thread pool. You can specify the ports of both the primary and standby nodes of the publication side. Separate the ports with commas (,). CAUTION The number of hosts must be the same as that of ports. • dbname Specifies the database where a publication is deployed. • user and password Specifies the username and password used to connect to the publication side. The user has the system administrator permission (SYSADMIN) or O&M administrator permission (OPRADMIN).

Parameter	Description	Value Range
publication_name	Specifies the name of the publication you want to subscribe to on the publication side. A subscription corresponds to multiple publications.	-
WITH	Specifies the optional parameters for a subscription. The right column lists the supported parameters.	• enabled Specifies whether a subscription should be actively replicated, or whether it should be just set but not started. The value can be true or false. The default value is true. • create_slot Specifies whether a replication slot is created on the publisher. The value can be true or false. The default value is true. • slot_name Specifies the name of the replication slot. Value range: a string Default value: the subscription name

Examples

Create a subscription to tables of the publication **mypublication** on a remote server.

```
CREATE SUBSCRIPTION mysub
  CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb password=xxxx'
  PUBLICATION mypublication;
```

Helpful Links

[ALTER SUBSCRIPTION, DROP SUBSCRIPTION](#)

13.59 CREATE SYNONYM

Function

CREATE SYNONYM is used to create a synonym object. A synonym is an alias of a database object and is used to record the mapping between database object names. You can use synonyms to access associated database objects.

Precautions

- The user of a synonym should be its owner.
- If the schema name is specified, create a synonym in the specified schema. Otherwise create a synonym in the current schema.
- Database objects that can be accessed using synonyms include tables, views, functions, and stored procedures.
- To use synonyms, you must have the required permissions on associated objects.
- The following DML statements support synonyms: **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **EXPLAIN**, and **CALL**.
- The **CREATE SYNONYM** statement of an associated function or stored procedure cannot be used in a stored procedure. You are advised to use synonyms existing in the **pg_synonym** system catalog in the stored procedure.

Syntax

```
CREATE [ OR REPLACE ] SYNONYM synonym_name  
FOR object_name;
```

Parameter Description

Table 13-106 CREATE SYNONYM parameters

Parameter	Description	Value Range
synonym	Name of a synonym which is created (optionally with schema names)	A string that must comply with the rules in Identifier Naming Conventions .
object_name	Name of an object that is associated (optionally with schema names) NOTE object_name can be the name of an object that does not exist.	A string that must comply with the rules in Identifier Naming Conventions .

Examples

Create schema **ot**.

```
CREATE SCHEMA ot;
```

Create table **ot.t1** and its synonym **t1**.

```
CREATE TABLE ot.t1(id int, name varchar2(10)) DISTRIBUTE BY hash(id);  
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;
```

Use synonym **t1**.

```
SELECT * FROM t1;  
INSERT INTO t1 VALUES (1, 'ada'), (2, 'bob');  
UPDATE t1 SET t1.name = 'cici' WHERE t1.id = 2;
```

Create synonym **v1** and its associated view **ot.v_t1**.

```
CREATE SYNONYM v1 FOR ot.v_t1;  
CREATE VIEW ot.v_t1 AS SELECT * FROM ot.t1;
```

Use synonym **v1**.

```
SELECT * FROM v1;
```

Create overloaded function **ot.add** and its synonym **add**.

```
CREATE OR REPLACE FUNCTION ot.add(a integer, b integer) RETURNS integer AS  
$$  
SELECT $1 + $2  
$$  
LANGUAGE sql;  
  
CREATE OR REPLACE FUNCTION ot.add(a decimal(5,2), b decimal(5,2)) RETURNS decimal(5,2) AS  
$$  
SELECT $1 + $2  
$$  
LANGUAGE sql;  
  
CREATE OR REPLACE SYNONYM add FOR ot.add;
```

Use synonym **add**.

```
SELECT add(1,2);  
SELECT add(1.2,2.3);
```

Create stored procedure **ot.register** and its synonym **register**.

```
CREATE PROCEDURE ot.register(n_id integer, n_name varchar2(10))  
SECURITY INVOKER  
AS  
BEGIN  
    INSERT INTO ot.t1 VALUES(n_id, n_name);  
END;  
/  
  
CREATE OR REPLACE SYNONYM register FOR ot.register;
```

Use synonym **register** to invoke the stored procedure.

```
CALL register(3,'mia');
```

Helpful Links

[ALTER SYNONYM](#), [DROP SYNONYM](#)

13.60 CREATE TABLE

Function

Creates a new empty table in the current database.

This table is owned by the user who executes the command. However, if the system administrator creates a table in the schema with the same name as a common user, the owner of the table is the user (not the system administrator).

Precautions

Table 13-107 Precautions for CREATE TABLE

Constraint	Description
Number of column-store and HDFS partitioned tables that can be created	For better performance, keep this value at or below 1,000.
Data types supported by column-store tables	For details, see Data Types Supported by Column-Store Tables .
Primary key and unique constraint requirements	The primary key and unique constraints in a table must contain distribution columns.
Modification of distribution columns	The data type of the distribution column in an existing table cannot be modified.
Row-store REPLICATION tables	A system column cannot be designated as a primary key in a row-store REPLICATION distributed table.
Supported range of column-store tables	Column-store tables support the PARTIAL CLUSTER KEY and table-level primary key and unique constraints, but do not support table-level foreign key constraints.
Columns of column-store tables	Only the NULL , NOT NULL , and DEFAULT constant values are supported.
Hot and cold tables	- Hot and cold tables support only column-store partitioned tables and depend on the available OBS. - Hot and cold tables support only the default tablespace default_obs_tbs. If you need to add an OBS tablespace, contact technical support.

Constraint	Description
Delta tables supported by column-store tables	Whether column-store tables support a delta table is specified by the enable_delta parameter. The threshold for storing data into a delta table is specified by the deltarow_threshold parameter. Column-store tables with delta tables are not recommended. If delta tables are used, disk bloat and performance deterioration may occur due to insufficient merge time. Delta tables in column-store tables are no longer supported in versions later than 9.1.0.200 due to the preceding problems. HStore tables are used instead. For details about HStore tables, see "Hybrid Data Warehouse" in <i>Data Warehouse Service (DWS) Developer Guide</i> .
HStore tables	- When creating an HStore table, ensure that the database GUC parameter settings meet the following requirements: autovacuum is set to on. - The value of autovacuum_max_workers_hstore is greater than 0. - The value of autovacuum_max_workers is greater than that of autovacuum_max_workers_hstore.
V3 tables	- Once a table is created, it cannot be switched from a non-V3 table to a V3 table using the ALTER TABLE syntax. In other words, a non-V3 table (colversion is 2.0) cannot be converted to a V3 table (colversion is 3.0). - Delta tables and column-store level-2 partitions cannot be set for V3 tables (colversion is 3.0, storage-compute decoupling). - V3 tables cannot be configured as HStore tables, cold and hot tables, or time sequence tables. - Global temporary tables and temporary tables cannot be created for V3 tables. Any created temporary tables are automatically converted to temporary tables with colversion 2.0.
Other	If an error occurs during table creation, the system may not be able to clear the created, empty disk file automatically after the system is recovered. This problem rarely occurs and does not affect the normal running of the database system.

WARNING

- Do not specify a custom tablespace when creating an ordinary table.
- Do not specify the **COMPRESS** compression attribute when creating a row-store table.
- When creating a hash-distributed table object, ensure that data is evenly distributed. (For a table with more than 10 GB data, the skew rate must be less than 10%.)
- When creating a replication-distributed table object, ensure that there are less than 1 million rows in a table.
- For a large table (with more than 50 million rows of data) that contains the time field, ensure that the table is designed as a partitioned table and the partition interval is properly designed based on the query characteristics.
- For a table that has a large amount of data to be added, deleted, or modified, keep the number of indexes within 3 (maximum value: 5).
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
CREATE [ [ GLOBAL | LOCAL | VOLATILE ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]  
table_name  
  { ( { column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint [ ... ] ]  
    | table_constraint  
    | LIKE source_table [ like_option [ ... ] ] }  
    [, ... ] )  
    | LIKE source_table [ like_option [ ... ] ] }  
  [ WITH ( { storage_parameter = value } [, ... ] ) ]  
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]  
  [ COMPRESS | NOCOMPRESS ]  
  [ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH ( column_name [ ... ] ) } } ]  
  [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]  
  [ COMMENT [=] 'text' ];
```

- **column_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]  
{ NOT NULL |  
  NULL |  
  CHECK ( expression ) |  
  DEFAULT default_expr |  
  ON UPDATE on_update_expr |  
  COMMENT 'text' |  
  UNIQUE [ NULLS [NOT] DISTINCT | NULLS IGNORE ] index_parameters |  
  PRIMARY KEY index_parameters |  
  REFERENCES reftable [ ( refcolumn ) ] }  
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- **compress_mode** of a column is as follows:

```
{ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS }
```

- **table_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]  
{ CHECK ( expression ) |  
  UNIQUE [ NULLS [NOT] DISTINCT | NULLS IGNORE ] ( column_name [, ... ] ) index_parameters |  
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |  
  PARTIAL CLUSTER KEY ( column_name [, ... ] ) }  
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

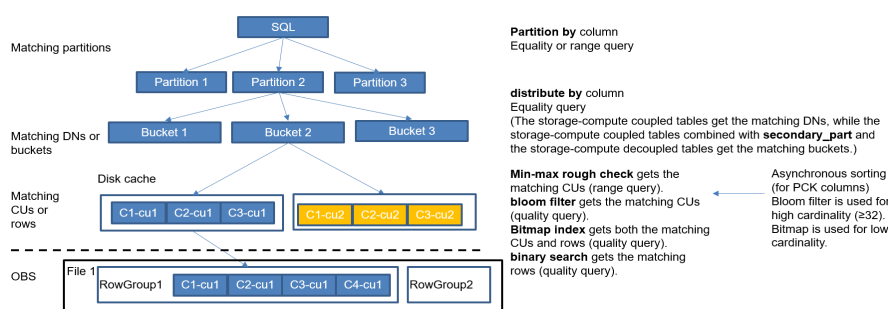
- **like_option** is as follows:

```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |  
PARTITION | REOPTIONS | DISTRIBUTION | DROPCOLUMNS | ALL }
```

- **index_parameters** is as follows:
[WITH ({storage_parameter = value} [, ...])]

Table Design Reference

DWS is compatible with the PostgreSQL ecosystem. Row storage and its B-tree index are similar to those of PostgreSQL. Column storage and its index are self-developed. When creating a table, it is crucial to choose the right storage method, distribution column, partition key, and index. This ensures efficient data access during SQL execution, reducing I/O consumption. The following figure illustrates the process from SQL statement initiation to data acquisition, helping you understand the function of each technical method for performance optimization.



1. When the SQL statement is executed, the partitioned table is optimized using the Partition Column to pinpoint the specific partition.
2. The Distribute Column is used in a distributed hash table to quickly identify the data shard where the data resides. The data shard is located on a DN in a storage-compute coupled architecture, while in a storage-compute decoupled architecture, it is located on a bucket.
3. In row-store mode, the B-tree index is used to quickly locate the data page. In column-store mode, the **min-max** index is used to quickly locate the CU data block that may contain relevant data. This index is particularly effective when filtering on the PCK column.
4. The system automatically maintains the **min-max** index for all columns in the column-store mode. There is no need for manual index definition. The **min-max** index is used for coarse filtering. CU data blocks meeting the min-max condition may not contain data rows that meet the filter condition. If a bitmap column is defined, the bitmap index can quickly locate the row number of data that meets the filter condition in the CU. For ordered CUs, binary search is also used to quickly locate the row number of data.
5. Column storage supports B-tree and GIN indexes, which can quickly locate the CU and row number of data that meets the conditions. However, due to high index maintenance costs, you are advised to use bitmap indexes instead, unless there are high performance requirements for point queries.

Table 13-108 lists the existing optimization methods of DWS.

Table 13-108 Optimization methods

No.	Method	Usage	Example SQL	Modifiable After Creation
1	String	1. The string type has slower performance compared to the fixed-length type, so it is not recommended for scenarios where the fixed-length type is more suitable. 2. If the specified length is less than 16, performance will be significantly improved.	-	Yes (The existing data can be rewritten.)
2	Numeric	Specifying precision for the numeric type is essential for improving performance. It is not advisable to use the numeric type without specifying precision.	-	Yes (The existing data can be rewritten.)
3	Partition by Column	1. This requires user-defined settings and is designed for partitioned tables. Pruning is possible using partition keys and partition-wise joins are supported. This method is suitable for equality and range queries. 2. Having more than 1000 partitions is not recommended, and it is advisable to limit the number of partition columns to two.	<pre>SELECT * FROM t1 WHERE t1.c1='p1';</pre>	No (You need to create a new table to make modifications.)
4	secondary_part_column	1. This requires user-defined settings and is applicable only to column-store tables and equality queries. 2. Specify a level-2 partition on the most commonly used equivalent filter.	<pre>SELECT * FROM t1 WHERE t1.c1='p1';</pre>	No (You need to create a new table to make modifications.)

No.	Method	Usage	Example SQL	Modifiable After Creation
5	Distribute by Column	This requires user-defined settings and is suitable for join fields that require frequent GROUP BY or multi-table joins. It reduces data shuffling through local joins and is ideal for equality queries.	<pre>SELECT * FROM t1 join t2 on t1.c3 = t2.c1;</pre>	No (You need to create a new table to make modifications.)
6	Bitmap column	Define the bitmap index (cardinality ≤ 32) or bloom filter (cardinality > 32) based on the repeated values in the CU. This method is applicable to equivalent queries of varchar or text type columns. It is advised to create indexes on columns involved in the WHERE condition.	<pre>SELECT * FROM t1 WHERE t1.c4='hello';</pre>	Yes (Modification does not rewrite existing data. Only the new data is affected.)
7	min-max index	1. The min-max index is automatically generated and can be used for both equality and range queries. 2. The min-max filtering effect depends on the data order. Specifying the PCK column enhances the filtering effect.	<pre>SELECT * FROM t1 WHERE c3 > 100 and c3 < 200;</pre>	Yes (The PCK columns can be modified. Modification does not rewrite existing data and only the new data is affected.)

No.	Method	Usage	Example SQL	Modifiable After Creation
8	Primary key (B-tree index)	1. UPSERT data import strongly depends on the primary key and needs to be customized. It is applicable to equality and range queries. We suggest limiting the number of columns to five or fewer. 2. If service requirements are met, it is better to use fixed-length type columns. During definition, place columns with more distinct values at the beginning.	<pre>SELECT * FROM t1 WHERE c3 > 100 and c3 < 200;</pre>	Yes (The index can be modified and re-created.)
9	GIN index	1. This requires user-defined settings and is suitable for multi-condition equality queries. Avoid using columns with more than 1 million distinct values. 2. It is recommended when the data volume after filtering is less than 1000. If the data volume remains large after filtering, it is not recommended.	<pre>SELECT * FROM t1 WHERE c1 = 100 and c3 = 200 and c2 = 105;</pre>	Yes (The index can be modified and re-created.)
10	Orientation=column/row	This method specifies whether a table is stored in rows or columns. Row-store tables cannot be compressed and are best suited for point queries and frequent updates. Column-store tables can be compressed and are ideal for analysis purposes.	-	No (You need to create a new table to make modifications.)

Parameter Description

Table 13-109 CREATE TABLE parameters

Parameter	Description	Value Range or Example
UNLOGGED	If UNLOGGED is specified, the created table is an unlogged table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, an unlogged table is automatically truncated after a crash or unclean shutdown, incurring data loss risks. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are not automatically logged as well.	Usage scenario: Unlogged tables do not ensure safe data. You can back up data before using unlogged tables; for example, you should back up the data before a system upgrade. Troubleshooting: If data is missing in the indexes of unlogged tables due to some unexpected operations such as an unclean shutdown, you should re-create the indexes with errors. Example: <pre>CREATE UNLOGGED TABLE table_unlog (col1 VARCHAR(50) , col2 TEXT);</pre>

Parameter	Description	Value Range or Example
	NOTICE 1. The UNLOGGED table uses no primary/standby mechanism. In the case of system faults or abnormal breakpoints, data loss may occur. Therefore, the UNLOGGED table cannot be used to store basic data. 2. Starting from version 9.1.0, UNLOGGED tables are automatically saved in the pg_unlogged tablespace and cannot be moved or assigned to other tablespaces. 3. After an earlier version is upgraded to 9.1.0, the UNLOGGED table created in the earlier version is still stored in the original tablespace. 4. If the instance restarts unexpectedly, the UNLOGGED table will be reset, which can impact the instance's recovery time objective (RTO). In version 9.1.0, switch_unlogged_tablespace.py can be used to move unlogged tables to optimize the recovery time objective (RTO). This script works together with the GUC parameter enable_unlogged_tablespace_compat.	

Parameter	Description	Value Range or Example
GLOBAL LOCAL VOLATILE	Specify the keywords GLOBAL, LOCAL, and VOLATILE before TEMP or TEMPORARY to create temporary tables with different attributes. Global temporary tables are supported only by 8.2.1.220 and later cluster versions. • If LOCAL is specified, a local temporary table is created. • If keyword GLOBAL is specified, the attributes of the temporary table depend on the GUC parameter enable_global_temp_table. The default value of this parameter is ON. If enable_global_temp_table is set to on, a global temporary table GLOBAL is created. If enable_global_temp_table is set to off, a local temporary table LOCAL is created. You can also specify keyword LOCAL to reach the same effect. • If VOLATILE is specified, a temporary volatile table is created. If default temptable_type is set to local, temporary tables created without keywords are local temporary tables. If default temptable_type is set to volatile, temporary tables created without keywords are volatile temporary tables.	Example: <pre>CREATE LOCAL TEMP TABLE temp_test (col1 INT, col2 DECIMAL(10,2));</pre>

Parameter	Description	Value Range or Example
TEMPORARY TEMP	If TEMP or TEMPORARY is specified, the created table is a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction. Therefore, apart from CN and other CN errors connected by the current session, you can still create and use temporary table in the current session. Temporary tables are created only in the current session. If a DDL statement involves operations on temporary tables, a DDL error will be generated. Therefore, you are not advised to perform operations on temporary tables in DDL statements. TEMP is equivalent to TEMPORARY.	For other restrictions of this parameter, see TEMPORARY TEMP Parameter Description .
IF NOT EXISTS	If IF NOT EXISTS is specified, a table will be created if there is no table using the specified name. If there is already a table using the specified name, no error will be reported. A message will be displayed indicating that the table already exists, and the database will skip table creation.	Example: Use IF NOT EXISTS to prevent duplicate table creation. <pre>CREATE TABLE IF NOT EXISTS test_t1 (col1 INT,col2 VARCHAR(30));</pre>

Parameter	Description	Value Range or Example
table_name	Specifies the name of the table to be created.	• The table name must start with a letter or underscore (_) and can contain up to 63 characters. Only letters, digits, underscores (_), dollar signs (\$), and number signs (#) are allowed. • Chinese table names are supported. The number of supported characters depends on the database's character set encoding. For details, see DWS Database Object Naming Rules. • You are not advised to use a keyword as a table name. For details, see Keyword. • A table name enclosed in double quotation marks can contain spaces and special characters. However, you are not advised to use these characters in a table name because they may make it difficult to reference and use. In addition, they may be processed differently under different database compatibility modes.
column_name	Specifies the name of a column to be created in the new table.	The column name can contain a maximum of 63 characters, including letters, digits, underscores (_), dollar signs (\$), and number signs (#). It must start with a letter or underscore (_).

Parameter	Description	Value Range or Example
data_type	Specifies the data type of the column.	In a database compatible with Teradata or MySQL syntax, if the data type of a column is set to DATE, the DATE type is returned. Otherwise, the TIMESTAMP type is returned.
compress_mode	Specifies the compress option of the table, only available for row-store table. The option specifies the algorithm preferentially used by table columns.	DELTA, PREFIX, DICTIONARY, NUMSTR, or NOCOMPRESS
COLLATE collation	Assigns a collation to the column (which must be of a collatable data type).	If no collation is specified, the default collation is used.
LIKE source_table [like_option ...]	Specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.	For details about LIKE constraints, see LIKE Parameter Description .

Parameter	Description	Value Range or Example
WITH	Specifies an optional storage parameter for a table or an index. NOTE Using Numeric of any precision to define column, specifies precision p and scale s. When precision and scale are not specified, the input will be displayed.	WITH contains many sub-parameters, which are classified as follows by scenario. You only need to pay attention to the parameters in specific scenarios. • For details about storage class parameters, see Table 13-110. • For details about parameters related to the partitioned table, see Table 13-111. • For details about the parameters related to the Delta table, see Table 13-112. • Parameters related to the Hstore table. For details, see Table 13-113. • For details about performance-related parameters, see Table 13-114.
ON COMMIT { PRESERVE ROWS DELETE ROWS }	ON COMMIT determines what to do when you commit a temporary table creation operation. Global temporary tables support only the PRESERVE ROWS option.	• PRESERVE ROWS (Default): No special action is taken at the ends of transactions. The temporary table and its table data are unchanged. • DELETE ROWS: All rows in the temporary table will be deleted at the end of each transaction block.

Parameter	Description	Value Range or Example
COMPRESS NOCOMPRESS	If you specify COMPRESS in the CREATE TABLE statement, the compression feature is triggered in the case of a bulk INSERT operation. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. If NOCOMPRESS is specified, the table is not compressed.	Default value: NOCOMPRESS , tuple data is not compressed before storage.
DISTRIBUTE BY	Specifies how the table is distributed or replicated between DNs. ● REPLICATION: Each row in the table exists on all DNs, that is, each DN has complete table data. ● ROUNDROBIN: Each row in the table is sent to each DN in turn. Therefore, data is evenly distributed on each DN. This value is supported only in 8.1.2 or later. ● HASH (column_name): Each row of the table will be placed into all the DNs based on the hash value of the specified column. For details about other restrictions, see DISTRIBUTE BY Parameter Description.	Example: Create a hash-distributed table. <pre>CREATE TABLE test_t2(col1 INT,col2 VARCHAR(30))DISTRIBUTE BY HASH(col1);</pre>

Parameter	Description	Value Range or Example
TO { GROUP groupname NODE (nodename [, ...]) }	TO GROUP specifies the Node Group in which the table is created. Currently, it cannot be used for HDFS tables. TO NODE is used for internal scale-out tools.	In logical cluster mode, if TO GROUP is not specified, the table is created in the node group associated with the logical cluster user by default. If the user, such as the administrator or a common user, does not manage the logical cluster, by default the table is created in the first logical cluster, which is the logical cluster with the smallest OID in pgxc_group. If the node group specified by TO GROUP is a replication table node group, the table is created on all CNs and DN, but the replication table data is distributed only on the DNs in the replication table node group. The Storage-compute decoupling 3.0 supports read-only logical clusters. If a user is not bound to any read-only logical clusters but sets TO GROUP to a logical cluster in a table creation statement, an error will be reported during table creation. If a user bound to a read-only logical cluster creates a table, the table will be created in the logical cluster specified by the GUC parameter default_storage_nodegroup. If default_storage_nodegroup is set to installation, tables will be created in the first logical cluster.

Parameter	Description	Value Range or Example
COMMENT [=] 'text'	The COMMENT clause can specify table comments during table creation.	Example: Add a comment for a table. <pre>CREATE TABLE table_user(user_id INT,username VARCHAR(50)) COMMENT 'user Information';</pre>
COMMENT 'text'	The COMMENT clause can specify a comment for a column.	Example: Add a comment for a specified column. <pre>CREATE TABLE table_users(user_id INT COMMENT 'employee id',username VARCHAR(50));</pre>
CONSTRAINT constraint_name	Specifies a name for a column or table constraint. The optional constraint clauses specify constraints that new or updated rows must satisfy for an insert or update operation to succeed. There are two ways to define constraints: • A column constraint is defined as part of a column definition, and it is bound to a particular column. • A table constraint is not bound to any particular columns but can apply to more than one column.	Example: Create the table_c_products table and define the primary key constraint. The primary key of the table consists of C_CUSTKEY and C_NAME . <pre>CREATE TABLE table_c_products (C_CUSTKEY BIGINT, C_NAME VARCHAR(25) , C_ADDRESS VARCHAR(40), C_NATIONKEY INT, C_PHONE CHAR(15) , C_ACCTBAL DECIMAL(15,2), CONSTRAINT C_CUSTKEY_KEY PRIMARY KEY(C_CUSTKEY,C_NAME)) DISTRIBUTE BY HASH(C_CUSTKEY,C_NAME);</pre>
NOT NULL	Indicates that the column is not allowed to contain NULL values.	-
NULL	The column is allowed to contain NULL values. This is the default setting. This clause is only provided for compatibility with non-standard SQL databases. You are advised not to use this clause.	-

Parameter	Description	Value Range or Example
CHECK (expression)	Specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. If any row of an insert or update operation produces a FALSE result, an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns.	<>NULL and !=NULL are invalid in an expression. Change them to IS NOT NULL.
DEFAULT default_expr	Assigns a default data value for a column.	The value can be any variable-free expressions (Subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is NULL.

Parameter	Description	Value Range or Example
ON UPDATE on_update_expr	The ON UPDATE clause specifies a timestamp function for a column. Ensure that the data type of the column for which the ON UPDATE clause specifies a timestamp function is timestamp or timestamptz. When an SQL statement containing the UPDATE operation is executed, this column is automatically updated to the time specified by the timestamp function.	on_update_expr supports only CURRENT_TIMESTAMP, CURRENT_TIME, CURRENT_DATE, LOCALTIME, and LOCALTIMESTAMP.

Parameter	Description	Value Range or Example
UNIQUE [NULLS [NOT] DISTINCT NULLS IGNORE] index_parameters UNIQUE [NULLS [NOT] DISTINCT NULLS IGNORE] (column_name [, ...]) index_parameters	Specifies that a group of one or more columns of a table can contain only unique values. The [NULLS [NOT] DISTINCT NULLS IGNORE] field is used to specify how to process null values in the index column of the Unique index.	Default value: This parameter is left empty by default. NULL values can be inserted repeatedly. When the inserted data is compared with the original data in the table, the NULL value can be processed in any of the following ways: • NULLS DISTINCT: NULL values are unequal and can be inserted repeatedly. • NULLS NOT DISTINCT: NULL values are equal. If all index columns are NULL, NULL values cannot be inserted repeatedly. If some index columns are NULL, data can be inserted only when non-null values are different. • NULLS IGNORE: NULL values are skipped during the equivalent comparison. If all index columns are NULL, NULL values can be inserted repeatedly. If some index columns are NULL, data can be inserted only when non-null values are different. The following table Table 13-116 lists the behaviors of the three processing modes. NOTE If DISTRIBUTE BY REPLICATION is not specified, the column table that contains only unique values must contain distribution columns.

Parameter	Description	Value Range or Example
PRIMARY KEY index_parameters PRIMARY KEY (column_name [, ...]) index_parameters	Specifies the primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values. Only one primary key can be specified for a table. CAUTION If DISTRIBUTE BY REPLICATION is not specified, the column set with a primary key constraint must contain distributed columns.	Example: Create the products table, set the data type of product_no to integer and set it as the primary key. <pre>CREATE TABLE products (product_no integer PRIMARY KEY, name text, price numeric) DISTRIBUTE BY HASH(product_no);</pre>
REFERENCES reftable [(refcolumn)]	The foreign key constraint requires that the group consisting of one or more columns in the new table should contain and match only the referenced column values in the referenced table. The referenced column should be the only column or primary key in the referenced table. DWS does not check foreign key constraints. When using foreign key constraints, you need to use the check_foreign_key_constraint function to check whether the data in the foreign key table meets the foreign key constraints.	-
DEFERRABLE NOT DEFERRABLE	Controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction using the SET CONSTRAINTS command.	NOT DEFERRABLE is the default value. Currently, only UNIQUE and PRIMARY KEY constraints of row-store tables accept this clause. All the other constraints are not deferrable.

Parameter	Description	Value Range or Example
PARTIAL CLUSTER KEY	Specifies a partial cluster key for storage. When importing data to a column-store table, you can perform local data sorting by specified columns (single or multiple).	Example: Create the products_new table and define the partial cluster key on the product_no column. <pre>CREATE TABLE products_new (product_no integer, name text, price numeric, PARTIAL CLUSTER KEY(product_no)) WITH (ORIENTATION = COLUMN);</pre>
INITIALLY IMMEDIATE INITIALLY DEFERRED	If a constraint is deferrable, this clause specifies the default time to check the constraint.	• If the constraint is INITIALLY IMMEDIATE (default value), it is checked after each statement. • If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. • The constraint check time can be altered using the SET CONSTRAINTS command.

Table 13-110 WITH parameters (storage-related)

Parameter	Description	Value Range
ORIENTATION	Specifies the storage mode (row-store, column-store) for table data. This parameter cannot be modified once it is set. - Row-store tables are suitable for OLTP workloads. There are many interactive transactions in such tables. An interaction involves multiple columns in a table. - Column-store tables are suitable for data warehouse workloads. There are a large amount of aggregation calculation and few column operations on these tables.	ROW indicates that table data is stored in rows. COLUMN indicates that the data is stored in columns. Default value: ROW (row-store) NOTE To create a column-store table, explicitly set orientation to column. When creating a local table in the storage-compute decoupling edition with all data stored on EVS disks, explicitly set colversion to 2.0.

Parameter	Description	Value Range
COMPRESSION	Specifies the compression level of the table data. It determines the compression ratio and time. Generally, the higher the level of compression, the higher the ratio, the longer the time, and the lower the level of compression, the lower the ratio, the shorter the time. The actual compression ratio depends on the distribution characteristics of loading table data.	This parameter is available only for column-store tables. Value: LOW (default value), MIDDLE, or HIGH. For details about the compression algorithms provided by DWS, see Table 13-115. NOTE - Currently, row-store table compression is not supported. - When migrating data from ORC to DWS column-store tables, low-level compression increases the backup size by 1.5 to 2 times compared to ORC, whereas high-level compression results in a similar backup size. It is crucial to take the conversion relationship into account while setting up a DWS cluster. - The middle compression of column-stores uses dictionary compression. For data not suitable for dictionary compression, the file size after middle compression may be greater than that of after low compression.
COMPRESSLEVEL	Specifies the compression level of the table data. It determines the compression ratio and time. This divides a compression level into sublevels, providing you with more choices for compression rate and duration. As the value becomes greater, the compression rate becomes higher and duration longer at the same compression level. The parameter is only valid for column-store tables.	0~3 Default value: 0

Parameter	Description	Value Range
MAX_BATCHROW	Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store tables.	10000 to 60000 Default value: 60,000
PARTIAL_CLUSTER_ROWS	Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store tables.	600000 to 2147483647 Default value: 4,200,000

Parameter	Description	Value Range
COLVERSION	Specifies the version of the column-store format. You can switch between different storage formats.	• 1.0: Each column in a column-store table is stored in a separate file. The file name is relfilenode.C1.0, relfilenode.C2.0, relfilenode.C3.0, or similar. • 2.0: All columns of a column-store table are combined and stored in a file. The file is named relfilenode.C1.0. • 3.0: Each column of a column-store table is stored in a file. The file is stored in the OBS file system and named C1_fileid.0. Default value: The default value for the storage-compute coupled version is 2.0, while for the storage-compute decoupling version, it is 3.0. The value of COLVERSION can only be set to 2.0 for OBS multi-temperature tables.

Parameter	Description	Value Range
		NOTE - For clusters of version 8.1.0, the default value of this parameter is 1.0. For clusters of version 8.1.1 or later, the default value of this parameter is 2.0. If the cluster version is upgraded from 8.1.0 to 8.1.1 or later, the default value of this parameter changes from 1.0 to 2.0. - When creating a column-store table, set COLVERSION to 2.0. Compared with the 1.0 storage format, the performance is significantly improved: - The time required for creating a column-store wide table is significantly reduced. - In the Roach data backup scenario, the backup time is significantly reduced. - The build and catch up time is greatly reduced. - The storage-compute decoupling version is compatible with all column-store versions. When creating a table, you need to explicitly set colversion to 1.0, 2.0, or 3.0. If colversion is set to 3.0, a table in decoupled storage and computing mode is created. If colversion is not explicitly specified, a column-store table of version 3.0 is created by default. When creating a table with decoupled storage and compute nodes, set colversion to 3.0 and set orientation to column. - V3 storage-compute decoupling tables do not support colversion switching using ALTER TABLE, for example, from 2.0 to 3.0.

Parameter	Description	Value Range
cache_policy (This parameter is supported only in 9.0.2 and later versions where the storage and compute nodes are decoupled.)	Specifies the cache mode of tables or partitioned tables (disks). If one of the following values is specified in the cache policy, hot cache is used. Otherwise, cold cache is used. Hot cache occupies more space than cold cache and uses more complex replacement policies.	• ALL: Hot cache is used for the entire table. • NONE: Cold cache is used for the entire table. • HPN: The first <i>N</i> partitions in a partitioned table use hot cache. The rest of the partitions use cold cache. • HPL: <i>P1, P2, ...</i> In a partitioned table, the specified partitions use hot cache. The rest of the partitions use cold cache. The default value is ALL. NOTE • For foreign tables and non-partitioned tables, only the ALL and NONE cache policies are supported. • Only range-partitioned and list-partitioned internal tables support HPN and HPL cache policies.
enable_disaster_cs_tore	Specifies whether fine-grained DR will be enabled for column-store tables. This parameter is available only for common column-store tables whose COLVERSION is 2.0. It is unavailable for HStore tables, row-store tables, temporary tables, unlogged tables, foreign tables, time series tables, cold and hot tables, materialized views, and delta tables, and cannot be enabled together with enable_hstore and enable_delta . This parameter is supported by version 8.2.0.100 or later clusters.	-

Parameter	Description	Value Range
<code>fine_disaster_table_role</code>	This parameter has been discarded in version 8.2.1 and is reserved for compatibility with earlier versions. This parameter is invalid in the current version. Specifies whether the fine-grained DR table will be set as a primary or secondary table. This parameter can be true only when the enable_disaster_cstore parameter has been set to true.	• primary: Specifies the primary fine-grained DR table. • standby: Specifies the standby fine-grained DR table.
<code>on_commit_preserve_rows</code>	It is similar to ON COMMIT { PRESERVE ROWS DELETE ROWS }. The two parameters cannot be specified at the same time. This parameter is used only for global temporary tables.	The default value is true .

Table 13-111 WITH parameters (related to partition tables)

Parameter	Description	Value Range
<code>TTL</code>	Schedules the partition deletion tasks in a partitioned table. By default, no partition deletion task is created.	1 hour ~ 100 years
<code>PERIOD</code>	Schedules the partition creation tasks in a partitioned table. If TTL has been configured, PERIOD cannot be greater than TTL .	1 hour ~ 100 years Default value: 1 day

Parameter	Description	Value Range
time_format	You can use auto-increment and decrement partitions for INT4 , INT8 , VARCHAR , and TEXT columns, which are commonly used for storing time-related data. The time_format option is applicable only when the partition key is INT4 , INT8 , VARCHAR , or TEXT and a period is specified. This is supported only by clusters of version 9.1.0.200 or later.	1. The value of time_format must comply with the PostgreSQL specifications, for example, yyyymmdd. 2. Restrictions on time_format differ by partition key type. VARCHAR/TEXT: • The precision can be accurate to seconds. • The value cannot contain letters, for example, month, am, and pm. • The time must be arranged in descending order, for example, year, month, day, hour, minute, and second. INT4/INT8: • The precision can be accurate to hours. • The value can contain only Y, M, D, and HH24. • The time must be arranged in descending order, for example, year, month, day, and hour. 3. ALTER restrictions: • The set operation is not supported. • When period is reset (indicating that automatic partitioning is disabled and a message is displayed), you can reset this option.

Table 13-112 WITH parameters (related to the Delta table)

Parameter	Description	Value Range
enable_delta	Specifies whether to enable delta tables in column-store tables. The parameter is only valid for column-store tables. If COLVERSION is set to 3.0 , enable_delta cannot be turned on because this parameter is not supported by V3 tables. Using column-store tables with delta tables is not recommended. This may cause disk bloat and performance deterioration due to delayed merge.	Default value: off
DELTAROW_THRESHOLD	Specifies the upper limit of to-be-imported rows for triggering the data import to a delta table when data is to be imported to a column-store table. This parameter takes effect only if the enable_delta table parameter is set to on . The parameter is only valid for column-store tables.	0 to 60000 Default value: 6000

Table 13-113 WITH parameters (related to Hstore tables)

Parameter	Description	Value Range
enable_hstore	Specifies whether an HStore table will be created (based on column-store tables). The parameter is only valid for column-store tables. This parameter is supported by version 8.2.0.100 or later clusters. If COLVERSION is set to 3.0 , enable_delta cannot be turned on because this parameter is not supported by V3 tables.	Default value: off NOTE If this parameter is enabled, the following GUC parameters must be set to ensure that H-Store tables are cleared. Set autovacuum to on , autovacuum_max_workers to 6 , and autovacuum_max_workers_hstore to 3 .

Parameter	Description	Value Range
enable_hstore_opt	If the enable_hstore_opt table-level parameter is enabled, the enable_hstore table-level parameter is also enabled by default. This parameter is supported only in cluster 8.3.0 and later versions.	Default value: false
secondary_part_column	Specifies the name of a level-2 partition column in a column-store table. Only one column can be specified as the level-2 partition column. This parameter applies only to HStore column-store tables. This parameter is supported only by clusters of version 8.3.0 or later.	• The column specified as a level-2 partition column cannot be deleted or modified. • The level-2 partition column can be specified only when a table is created. After a table is created, the level-2 partition column cannot be modified. • You are not advised to specify a distribution column as a level-2 partition column. • The level-2 partition column determines how the table is logically split into hash partitions on DNs, which enhances the query performance for that column.
secondary_part_num	Specifies the number of level-2 partitions in a column-store table. This parameter applies only to HStore column-store tables. This parameter is supported only by clusters of version 8.3.0 or later.	1~32 Default value: 8 NOTICE • This parameter can be specified only when secondary_part_column is specified. • The number of level-2 partitions can be specified only when a table is created and cannot be modified after the table is created. • You are not advised to change the default value, which may affect the import and query performance.

Parameter	Description	Value Range
bitmap_columns	The bitmap index is only applicable to the new HStore (hstore_opt) tables. To generate the bitmap index mapping, you need to enable the enable_hstore_opt table-level parameter and set bitmap_columns to Specified Columns . This parameter is supported only by clusters of version 8.3.0 or later.	-

Table 13-114 WITH parameters (other performance-related parameters)

Parameter	Description	Value Range
FILLFACTOR	The fillfactor of a table is a percentage between 10 and 100. 100 (complete packing) is the default value. When a smaller fillfactor is specified, INSERT operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The parameter has no meaning for column-store tables.	10~100

Parameter	Description	Value Range
analyze_mode	Specifies the mode of table-level auto-analyze.	• frozen: disables all ANALYZE operations (dynamic sampling can still be triggered when no statistics are collected). • backend: allows only ANALYZE triggered by AUTOVACUUM polling. • runtime: allows only runtime ANALYZE triggered by the optimizer. • all: Both backend and runtime AUTO-ANALYZE can be triggered. Default value: all
incremental_analyze	Specifies whether to enable the incremental analyze mode for partitioned tables. This parameter is valid only for partitioned tables and cannot be set for replicated tables. This is supported only by clusters of version 9.1.0.100 or later.	Default value: false

Parameter	Description	Value Range
enable_turbo_store	Determines whether to create a turbo table (column-store tables). The parameter is only valid for column-store tables. It is available only for clusters of versions 9.1.0.100 and later. NOTE - Common column-store tables in version 3.0 do not support the turbo storage format, while HStore Opt tables in version 3.0 only support the turbo storage format. - In version 2.0, there is no restriction on column-store tables. - Currently, turbo tables do not support time series tables, delta tables, or lightweight update.	Default value: off NOTE Turbo tables enhance the storage efficiency of numeric and varchar data types using the integers, leading to accelerated processing speeds for these types. - For enhanced performance, it is advised to define the numeric type with precision and scale (p, s). The storage allocation is as follows: For $p \leq 9$, 4-byte integers are used for storage. For $p \leq 18$, 8-byte integers are used. For $p \leq 37$, 16-byte integers are used. If $p > 37$ or remain unspecified, a variable-length format is used, which is less space-efficient and yields suboptimal performance. - If the maximum length (n) of the varchar type is less than or equal to 2 bytes, 2-byte integers are used for storage. If the maximum length (n) of the varchar type is less than or equal to 4 bytes, 4-byte integers are used for storage. If the maximum length (n) of the varchar type is less than or equal to 16 bytes, 16-byte integers are used for storage. Limit VARCHAR columns in GROUP BY or HASH JOIN operations to 16 bytes or less to enable faster integer-based storage and improve performance.

Parameter	Description	Value Range
SKIP_FPI_HINT	Indicates whether to skip the hint bits operation when the full-page writes (FPW) log needs to be written during sequential scanning.	Default value: false NOTE If SKIP_FPI_HINT is set to true and the checkpoint operation is performed on a table, no Xlog will be generated when the table is sequentially scanned. This applies to intermediate tables that are queried less frequently, reducing the size of Xlogs and improving query performance.
enable_column_autovacuum_garbage	Determines whether to enable CU rewriting logic for column-store tables using AUTOVACUUM. This parameter is supported only by clusters of version 8.2.1.100 or later. There is a low probability that an error is reported when lightweight UPDATE and AUTOVACUUM are executed concurrently. You can set the table-level parameter to off to avoid this problem.	Default value: true

Table 13-115 Compression algorithms for column-based storage

COMPRESSION	NUMERIC	STRING	INT
LOW	Delta compression + RLE compression	LZ4 compression	Delta compression (RLE is optional.)
MIDDLE	Delta compression + RLE compression + LZ4 compression	dict compression or LZ4 compression	Delta compression or LZ4 compression (RLE is optional)
HIGH	Delta compression + RLE compression + zlib compression	dict compression or zlib compression	Delta compression or zlib compression (RLE is optional)

Table 13-116 Processing of NULL values in index columns in unique indexes

Constraint	All Index Columns Are NULL	Some Index Columns Are NULL.
NULLS DISTINCT	Can be inserted repeatedly.	Can be inserted repeatedly.
NULLS NOT DISTINCT	Cannot be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.
NULLS IGNORE	Can be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.

TEMPORARY | TEMP Parameter Description

- Local or volatile temporary tables are visible to the current session through schema of the **pg_temp** start. Users should not delete schema started with **pg_temp**, **pg_toast_temp**.
- If **TEMPORARY** or **TEMP** is not specified when you create a table and the schema of the specified table starts with **pg_temp_**, the table is created as a temporary table.
- Similar to common tables, all metadata of local temporary tables is stored in system catalogs. Volatile temporary tables store table structure metadata in memory, except the schema metadata. Compared with local temporary tables, volatile temporary tables have the following constraints:
 - After a CN or DN is restarted, data in its memory will be lost, and accordingly, volatile temporary tables on it will become invalid.
 - Currently, volatile temporary tables do not support table structure modification, such as **ALTER** and **GRANT**.
 - Volatile temporary tables and local temporary tables share temporary schemas. Therefore, in the same session, the **VOLATILE** temporary table and local temporary table cannot have the same name.
 - Volatile temporary table information is not stored in system catalogs. Therefore, Volatile metadata cannot be queried by running DML statements in system catalogs.
 - Volatile temporary tables support only common row-store and column-store tables. Delta tables, time series tables, and cold and hot tables are not supported.
 - Views cannot be created based on volatile temporary tables.
 - A tablespace cannot be specified when a temporary table is created. The default tablespace of a volatile temporary table is **pg_volatile**.

- The following constraints cannot be specified when a volatile temporary table is created: CHECK, UNIQUE, PRIMARY KEY, TRIGGER, EXCLUDE, and PARTIAL CLUSTER.
- Similar to common tables, all metadata in global temporary tables is stored in system catalogs.
 - The difference between a global temporary table and a local temporary table is that in a global temporary table, the metadata is not deleted when a session exits, but the session data is deleted. Data of different sessions is independent but shares the metadata of the same global temporary table.
 - A global temporary table has a schema that is similar to a regular table, unlike a local or volatile temporary table which has a schema starting with **pg_temp**. This means that a global temporary table can have the same name as a local or volatile temporary table.
 - Global temporary tables support only common row-store and column-store tables. Delta tables, time series tables, and cold and hot tables are not supported.
 - Operations cannot be performed on global temporary tables of other logical clusters.

LIKE Parameter Description

The new table and the source table are decoupled after creation is complete. Changes to the source table will not be applied to the new table, and it is not possible to include data of the new table in scans of the source table.

Columns and constraints copied by **LIKE** are not merged with the same name. If the same name is specified explicitly or in another **LIKE** clause, an error is reported.

- The default expressions or the **ON UPDATE** expressions are copied from the source table to the new table only if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having default values **NULL**.
- The **CHECK** constraints are copied from the source table to the new table only when **INCLUDING CONSTRAINTS** is specified. Other types of constraints are never copied to the new table. **NOT NULL** constraints are always copied to the new table. These rules also apply to column constraints and table constraints.
- Any indexes on the source table will not be created on the new table, unless the **INCLUDING INDEXES** clause is specified.
- **STORAGE** settings for the copied column definitions are copied only if **INCLUDING STORAGE** is specified. The default behavior is to exclude **STORAGE** settings.
- If **INCLUDING COMMENTS** is specified, comments for the copied columns, constraints, and indexes are copied. The default behavior is to exclude comments.
- If **INCLUDING PARTITION** is specified, the partition definitions of the source table are copied to the new table, and the new table no longer uses the **PARTITION BY** clause. The default behavior is to exclude partition definition of the source table.

- If **INCLUDING REOPTIONS** is specified, the storage parameter (**WITH** clause of the source table) of the source table is copied to the new table. The default behavior is to exclude partition definition of the storage parameter of the source table.

 NOTE

PERIOD and **TTL** in the **WITH** clause are partition-related parameters. **LIKE INCLUDING REOPTIONS** will not be copied to the new table. To copy **LIKE INCLUDING REOPTIONS**, use **INCLUDING PARTITION**.

- If **INCLUDING DISTRIBUTION** is specified, the distribution information of the source table is copied to the new table, including distribution type and column, and the new table no longer use the **DISTRIBUTE BY** clause. The default behavior is to exclude distribution information of the source table.
- If **INCLUDING DROPCOLUMNS** is specified, the deleted column information in the source table is copied to the new table. By default, the deleted column information of the source table is not copied.
- **INCLUDING ALL** contains the meaning of **INCLUDING DEFAULTS**, **INCLUDING CONSTRAINTS**, **INCLUDING INDEXES**, **INCLUDING STORAGE**, **INCLUDING COMMENTS**, **INCLUDING PARTITION**, **INCLUDING REOPTIONS**, **INCLUDING DISTRIBUTION**, and **INCLUDING DROPCOLUMNS**.
- If **EXCLUDING** is specified, the specified parameters are not included.
- For an OBS multi-temperature table, all partitions of the new table are local hot partitions after **INCLUDING PARTITION** is specified.
- If **ALTER TABLE...ADD PARTITION** and **CREATE TABLE** are both executed for the source table, the number of partitions on different nodes may be inconsistent occasionally. As a result, an error is reported during the execution of **CREATE TABLE**. You are advised to execute **CREATE TABLE** again.

NOTICE

- If the source table contains a sequence with the **SERIAL**, **BIGSERIAL**, or **SMALLSERIAL** data type, or a column in the source table is a sequence by default and the sequence is created for this table by using **CREATE SEQUENCE... OWNED BY**, these sequences will not be copied to the new table, and another sequence specific to the new table will be created. This is different from earlier versions. To share a sequence between the source table and new table, create a shared sequence (do not use **OWNED BY**) and set a column in the source table to this sequence.
 - You are not advised to set a column in the source table to the sequence specific to another table especially when the table is distributed in specific node groups, because doing so may result in **CREATE TABLE ... LIKE** execution failures. In addition, doing so may cause the sequence to become invalid in the source sequence because the sequence will also be deleted from the source table when it is deleted from the table that the sequence is specific to. To share a sequence among multiple tables, you are advised to create a shared sequence for them.
-

DISTRIBUTE BY Parameter Description

Specifies how the table is distributed or replicated between DN.

Valid value:

- **REPLICATION**: Each row in the table exists on all DN, that is, each DN has complete table data.
- **ROUNDROBIN**: Each row in the table is sent to each DN in turn. Therefore, data is evenly distributed on each DN. This value is supported only in 8.1.2 or later.
- **HASH (column_name)**: Each row of the table will be placed into all the DN based on the hash value of the specified column.

NOTE

- When **DISTRIBUTE BY HASH (column_name)** is specified, the primary key and its unique index must contain the **column_name** column.
- When **DISTRIBUTE BY HASH (column_name)** in a referenced table is specified, the foreign key of the reference table must contain the **column_name** column.
- If **TO GROUP** is set to a replication table node group (supported in 8.1.2 or later), **DISTRIBUTE BY** must be set to **REPLICATION**. If **DISTRIBUTE BY** is not specified, the created table is automatically set as a replication table.

Default value: determined by the GUC parameter **default_distribution_mode**

- When **default_distribution_mode** is set to **roundrobin**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 - a. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 - b. If the primary key or unique constraint is not included during table creation, round-robin distribution is selected.
- When **default_distribution_mode** is set to **hash**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 - a. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 - b. If the primary key or unique constraint is not included during table creation but there are columns whose data types can be used as distribution columns, hash distribution is selected. The distribution column is the first column whose data type can be used as a distribution column.
 - c. If the primary key or unique constraint is not included during table creation and no column whose data type can be used as a distribution column exists, round-robin distribution is selected.

The following data types can be used as distribution columns:

- Integer types: **TINYINT**, **SMALLINT**, **INT**, **BIGINT**, and **NUMERIC/DECIMAL**
- Character types: **CHAR**, **BPCHAR**, **VARCHAR**, **VARCHAR2**, **NVARCHAR2**, and **TEXT**
- Date/time types: **DATE**, **TIME**, **TIMETZ**, **TIMESTAMP**, **TIMESTAMPTZ**, **INTERVAL**, and **SMALLDATETIME**

 NOTE

When you create a table, the choices of distribution keys and partition keys have major impact on SQL query performance. Therefore, choosing proper distribution column and partition key with strategies.

- Selecting an Appropriate Distribution Column

In the data distributed table using Hash, an appropriate distributed array should be used to distribute and store data on multiple DNs evenly, preventing data skew (uneven data distribution across several DNs). Determine the proper distribution column based on the following principles:

1. Determine whether data is skewed.

Connect to the database and run the following statements to check the number of tuples on each DN: Replace *tablename* with the actual name of the table to be analyzed.

```
SELECT a.count,b.node_name FROM (SELECT count(*) AS count,xc_node_id FROM
tablename GROUP BY xc_node_id) a, pgxc_node b WHERE a.xc_node_id=b.node_id ORDER
BY a.count DESC;
```

If tuple numbers vary greatly (several times or tenfold) in each DN, a data skew occurs. Change the data distribution key based on the following principles:

2. Run the ALTER TABLE statement to adjust the distribution column. The rules for selecting a distribution column are as follows:

The column value of the distribution column should be discrete so that data can be evenly distributed on each DN. For example, you are advised to select the primary key of a table as the distribution column, and the ID card number as the distribution column in a personnel information table.

With the above principles met, you can select join conditions as distribution keys so that join tasks can be pushed down to DNs, reducing the amount of data transferred between the DNs.

3. If a proper distribution column cannot be found to make data evenly distributed on each DN, you can use the **REPLICATION** or **ROUNDROBIN** data distribution mode. The **REPLICATION** data distribution mode stores complete data on each DN. Therefore, if a table is large and no proper distribution column can be found, the **ROUNDROBIN** data distribution mode is recommended. The **ROUNDROBIN** data distribution mode is supported in 8.1.2 or later.

- Selecting appropriate partition keys

In range partitioning, the table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. Each range has a dedicated partition for data storage.

Modify partition keys to make the query result stored in the same or least partitions (partition pruning). Obtaining consecutive I/O to improve the query performance.

In actual services, time is used to filter query objects. Therefore, you can use time as a partition key, and change the key value based on the total data volume and single data query volume.

Example: Creating a Storage-Compute Coupled Table

Create the **test_t4** table.

```
CREATE TABLE IF NOT EXISTS test_t4 ( ---Create a table when there are no test_t4 tables in the database.
col_id INT PRIMARY KEY, ---Define an INTEGER column and set it as the primary key.
col_number INT NOT NULL, ---Define an INTEGER column and set it to not null.
col_date DATE NOT NULL, --- Define a DATE column and set it to non-null.
col_price NUMERIC(10,2), ---Define a NUMERIC column. The maximum number of integer digits is 10,
and the maximum number of decimal places is 2.
col_status TEXT ---Define a TEXT column.
)
WITH (ORIENTATION = COLUMN,COLVERSION=2.0,COMPRESSION=middle) --- Set the storage method of
a table to column storage and the compression level to MIDDLE.
```

```
DISTRIBUTE BY HASH (col_id)      ---Use hash distribution.  
COMMENT 'order information for test_table4'; ---Comment a table.
```

Example: Creating a Storage-Compute Decoupled V3 Table

Create a V3 table (**test_t5**) with decoupled storage and compute (supported only in the storage-compute decoupling 3.0 version).

```
CREATE TABLE test_t5  
(  
  id integer not null,  
  data integer,  
  age integer  
)  
WITH (ORIENTATION = COLUMN, COLVERSION = 3.0)  
DISTRIBUTE BY ROUNDROBIN;
```

Specify the cache policy when creating a table (supported only in the storage-compute decoupling 3.0 version).

```
CREATE TABLE sports  
(  
  N_NATIONKEY INT NOT NULL  
  , N_NAME CHAR(25) NOT NULL  
  , N_REGIONKEY INT NOT NULL  
  , N_COMMENT VARCHAR(152)  
) WITH (orientation = column, colversion = 3.0, cache_policy = 'HPL: Balls, Basketball')  
tablespace cu_obs_tbs  
DISTRIBUTE BY ROUNDROBIN  
partition by list(N_NAME)  
(  
  partition Balls values ('Basketball', 'football', 'badminton'),  
  partition Athletics values ('High jump', 'long jump', 'javelin'),  
  partition Water_Sports values ('Surfing', 'diving', 'swimming'),  
  partition Shooting values ('air guns', 'Rifles', 'archery'),  
  partition rest values (DEFAULT)  
);
```

Example: Defining Constraints When Creating a Table

Define **PRIMARY KEY** constraints for one or more columns of a table.

```
CREATE TABLE CUSTOMER  
(  
  C_CUSTKEY BIGINT ,  
  C_NAME VARCHAR(25) ,  
  C_ADDRESS VARCHAR(40) ,  
  C_NATIONKEY INT ,  
  C_PHONE CHAR(15) ,  
  C_ACCTBAL DECIMAL(15,2) ,  
  CONSTRAINT C_CUSTKEY_KEY PRIMARY KEY(C_CUSTKEY,C_NAME)  
)  
DISTRIBUTE BY HASH(C_CUSTKEY,C_NAME);
```

Define a **UNIQUE** constraint for a table.

```
CREATE TABLE CUSTOMER  
(  
  C_CUSTKEY BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY,  
  C_NAME VARCHAR(25) ,  
  C_ADDRESS VARCHAR(40) ,  
  C_NATIONKEY INT ,  
  C_PHONE CHAR(15) ,  
  C_ACCTBAL DECIMAL(15,2)  
)  
DISTRIBUTE BY HASH(C_CUSTKEY);
```

Define a **CHECK** constraint for a table.

```
CREATE TABLE CUSTOMER
(
  C_CUSTKEY BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY,
  C_NAME VARCHAR(25) ,
  C_ADDRESS VARCHAR(40) ,
  C_NATIONKEY INT ,
  CONSTRAINT C_CUSTKEY_KEY2 CHECK(C_CUSTKEY > 0 AND C_NAME <> '')
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

Define a **CHECK** constraint for a column.

```
CREATE TABLE CUSTOMER
(
  C_CUSTKEY BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY ,
  C_NAME VARCHAR(25) ,
  C_ADDRESS VARCHAR(40) ,
  C_NATIONKEY INT NOT NULL CHECK (C_NATIONKEY > 0)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

Use **DEFAULT** to declare the default value **GA** for the column **col_STATE**.

```
CREATE TABLE test_t5
(
  col_SK INTEGER NOT NULL,
  col_ID CHAR(16) NOT NULL,
  colE_NAME VARCHAR(20) UNIQUE DEFERRABLE,
  col_WAREHOUSE INTEGER,
  col_COUNTY VARCHAR(30),
  col_STATE CHAR(2) DEFAULT 'GA',
  col_ZIP CHAR(10)
);
```

Example: Creating a Column-Store Table and Specifying the Compression Level

Create the column-store table **customer_address** and set the data compression level to **HIGH**.

```
CREATE TABLE customer_address
(
  ca_address_sk INTEGER NOT NULL ,
  ca_address_id CHARACTER(16) NOT NULL ,
  ca_street_number CHARACTER(10) ,
  ca_street_name CHARACTER varying(60) ,
  ca_street_type CHARACTER(15) ,
  ca_suite_number CHARACTER(10)
)
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)
DISTRIBUTE BY HASH (ca_address_sk);
```

Example: Using CREATE TABLE LIKE to Create a Table

Use CREATE TABLE LIKE to quickly copy the schema of the existing table **test_t6** and create the table **test_t7**.

```
CREATE TABLE IF NOT EXISTS test_t6
(
  col_id INT PRIMARY KEY,
  col_number INT NOT NULL,
  col_date DATE NOT NULL,
  col_price NUMERIC(10,2),
  col_status TEXT
)
WITH (ORIENTATION = COLUMN, COMPRESSION=middle)
```

```
DISTRIBUTE BY HASH (col_id);  
CREATE TABLE test_t7 (LIKE test_t6 INCLUDING ALL);
```

Helpful Links

[ALTER TABLE](#), [12.101-RENAME TABLE](#), and [DROP TABLE](#)

FAQs

[How Do I Adjust GaussDB\(DWS\) Distribution Columns?](#)

[How Will Data Be Stored in a GaussDB\(DWS\) Table If No Distribution Column Is Specified During Its Creation?](#)

13.61 CREATE TABLE AS

Function

CREATE TABLE AS creates a table based on the results of a query. It is applicable to data migration or replication scenarios.

CREATE TABLE AS creates a table and fills the table with **SELECT** query results. The table columns have the names and data types associated with the output columns of **SELECT**. Except that you can override the **SELECT** output column names by giving an explicit list of new column names.

CREATE TABLE AS queries once the source table and writes data in the new table. The query result view changes when the source table changes. In contrast, a view re-evaluates its defining **SELECT** statement whenever it is queried.

Precautions

- This command cannot be used to create a partitioned table.
- If an error occurs when you create a table, after the system is recovered, the system probably cannot automatically clear the created disk file whose size is not 0. This problem seldom occurs.
- The UNLOGGED table uses no primary/standby mechanism. In the case of system faults or abnormal breakpoints, data loss may occur. Therefore, the UNLOGGED table cannot be used to store basic data.
- The UNLOGGED table does not ensure safe data. Users can back up data before using the table; for example, users should back up the data before a system upgrade.

Syntax

```
CREATE [ [ GLOBAL | LOCAL | VOLATILE ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE table_name  
    [ (column_name [, ...] ) ]  
    [ WITH ( {storage_parameter = value} [, ...] ) ]  
    [ COMPRESS | NOCOMPRESS ]  
  
    [ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { [HASH] ( column_name ) } } ]  
  
    [ COMMENT [=] 'text' ]  
    AS query  
    [ WITH [ NO ] DATA ];
```

Parameter Description

Table 13-117 CREATE TABLE AS parameters

Parameter	Description	Value Range
GLOBAL LOCAL VOLATILE	Specifies the type of a temporary table.	For details, see Parameter Description .
TEMPORARY TEMP	If TEMP or TEMPORARY is specified, the created table is a temporary table.	-
UNLOGGED	Specifies that the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well. If data is missing in the indexes of unlogged tables due to some unexpected operations such as an unclean shutdown, users should re-create the indexes with errors.	Starting from version 9.1.0, UNLOGGED tables are automatically saved in the pg_unlogged tablespace and cannot be moved or assigned to other tablespaces. After an earlier version is upgraded to 9.1.0, the UNLOGGED table created in the earlier version is still stored in the original tablespace. If the instance restarts unexpectedly, the UNLOGGED table will be reset, which can impact the instance's recovery time objective (RTO). Starting from version 9.1.0, there is a script called switch_unlogged_tablespace.py that can move unlogged tables to optimize the recovery time objective (RTO). This script works together with the GUC parameter enable_unlogged_tablespace_compat.
table_name	Specifies the name of the table to be created.	A string, which must comply with the naming convention .
column_name	Specifies the name of a column to be created in the new table.	A string, which must comply with the naming convention .

Parameter	Description	Value Range
WITH (storage_parameter [= value] [, ...])	The WITH clause sets the storage options for a table or its index.	For details about the parameters, see Table 13-118 .
COMPRESS NOCOMPRESS	• If the keyword COMPRESS is specified, the compression feature is triggered when data is inserted into tables in batches. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. • If NOCOMPRESS is specified, the table is not compressed. Default value: NOCOMPRESS , tuple data is not compressed before storage.	-

Parameter	Description	Value Range
DISTRIBUTE BY	Specifies how the table is distributed or replicated between DNs. • REPLICATION: Each row in the table exists on all DNs, that is, each DN has complete table data. • ROUNDROBIN: Each row in the table is sent to each DN in sequence. This distribution policy prevents data skew. However, data distribution nodes are random. As a result, there is a higher probability that table redistribution is triggered during computing. This distribution policy is recommended for large tables with severe column skew. This value is supported only in 8.1.2 or later. • HASH (column_name): Each row of the table will be placed into all the DNs based on the hash value of the specified column. Default value: determined by the GUC parameter default_distribution_mode. For details about how to select the default value of DISTRIBUTE BY, see the notice below this table.	The following data types can be used as distribution columns: • Integer types: TINYINT, SMALLINT, INT, BIGINT, and NUMERIC/DECIMAL • Character types: CHAR, BPCHAR, VARCHAR, VARCHAR2, NVARCHAR2, and TEXT • Date/time types: DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPZ, INTERVAL, and SMALLDATETIME
COMMENT [=] 'text'	The COMMENT clause can specify table comments during table creation.	-

Parameter	Description	Value Range
AS query	Indicates a SELECT or VALUES command, or an EXECUTE command that runs a prepared SELECT , or VALUES query.	-
[WITH [NO] DATA]	Specifies whether the data produced by the query should be copied into the new table. By default, the data is copied. If the NO parameter is used, the data is not copied.	-

NOTICE

DISTRIBUTE BY parameter description:

- When **DISTRIBUTE BY HASH (column_name)** is specified, the primary key and its unique index must contain the **column_name** column.
 - When **DISTRIBUTE BY HASH (column_name)** in a referenced table is specified, the foreign key of the reference table must contain the **column_name** column.
 - When **default_distribution_mode** is set to **roundrobin**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 1. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 2. If the primary key or unique constraint is not included during table creation, round-robin distribution is selected.
 - When **default_distribution_mode** is set to **hash**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 1. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 2. If the primary key or unique constraint is not included during table creation but there are columns whose data types can be used as distribution columns, hash distribution is selected. The distribution column is the first column whose data type can be used as a distribution column.
 3. If the primary key or unique constraint is not included during table creation and no column whose data type can be used as a distribution column exists, round-robin distribution is selected.
-

Table 13-118 storage_parameter parameters

storage_parameter Parameter	Description	Value Range
FILLFACTOR	The fillfactor of a table is a percentage between 10 and 100. When a smaller fillfactor is specified, INSERT operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The default value is 100 (filled to capacity). The parameter is only valid for row-store tables.	10~100
ORIENTATION	Specifies the storage mode (row-store, column-store) for table data. This parameter cannot be modified once it is set. Default value: ROW (row-store)	• ROW indicates that table data is stored in rows. ROW applies to OLTP service, which has many interactive transactions. An interaction involves many columns in the table. Using ROW can improve the efficiency. • COLUMN indicates that the data is stored in columns. COLUMN applies to the data warehouse service, which has a large amount of aggregation computing, and involves a few column operations.

storage_parameter Parameter	Description	Value Range
COMPRESSION	Specifies the compression level of the table data. It determines the compression ratio and time. Generally, the higher the level of compression, the higher the ratio, the longer the time, and the lower the level of compression, the lower the ratio, the shorter the time. The actual compression ratio depends on the distribution characteristics of loading table data. NOTE Currently, row-store table compression is not supported.	This parameter is available only for column-store tables. Value: LOW (default value), MIDDLE, or HIGH.
MAX_BATCHROW	Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store tables. Default value: 60000	10000 to 60000
PARTIAL_CLUSTER_ROWS	Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store tables. Default value: 4,200,000	600000 to 2147483647
enable_delta	Specifies whether to enable delta tables in column-store tables. The parameter is only valid for column-store tables. Using column-store tables with delta tables is not recommended. This may cause disk bloat and performance deterioration due to delayed merge. Default value: off	on or off

storage_parameter Parameter	Description	Value Range
COLVERSION	Specifies the version of the column-store format. You can switch between different storage formats. Default value: 2.0 NOTE When creating a column-store table, set COLVERSION to 2.0. Compared with the 1.0 storage format, the performance is significantly improved: 1. The time required for creating a column-store wide table is significantly reduced. 2. In the Roach data backup scenario, the backup time is significantly reduced. 3. The build and catch up time is greatly reduced.	Valid value: • 1.0: Each column in a column-store table is stored in a separate file. The file name is relfilenode.C1.0, relfilenode.C2.0, relfilenode.C3.0, or similar. • 2.0: All columns of a column-store table are combined and stored in a file. The file is named relfilenode.C1.0.
SKIP_FPI_HINT	Indicates whether to skip the hint bits operation when the full-page writes (FPW) log needs to be written during sequential scanning. Default value: false NOTE If SKIP_FPI_HINT is set to true and the checkpoint operation is performed on a table, no Xlog will be generated when the table is sequentially scanned. This applies to intermediate tables that are queried less frequently, reducing the size of Xlogs and improving query performance.	true or false

Examples

Create the **customer** table.

```
DROP TABLE IF EXISTS customer;  
CREATE TABLE customer  
(  
    C_CUSTKEY    BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY ,  
    C_NAME      VARCHAR(25) ,  
    C_ADDRESS   VARCHAR(40) ,  
    C_NATIONKEY INT NOT NULL CHECK (C_NATIONKEY > 0)
```

```
)  
DISTRIBUTE BY HASH(C_CUSTKEY);
```

Create the **store_returns_t1** table and insert numbers that are greater than 4795 in the **C_CUSTKEY** column of the **CUSTOMER** table:

```
CREATE TABLE store_returns_t1 AS SELECT * FROM CUSTOMER WHERE C_CUSTKEY > 4795;
```

Copy **store_returns_t1** to create the **store_returns_t2** table:

```
CREATE TABLE store_returns_t2 AS table store_returns_t1;
```

Helpful Links

[CREATE TABLE, SELECT](#)

13.62 CREATE TABLE PARTITION

Function

CREATE TABLE PARTITION creates a partitioned table. A partitioned table is a logical table that is divided into segments, called partitions whose data is stored on different physical blocks.

Common partitioning strategies define data ranges based on one or more columns. Each partition stores data within a range. These columns are called partition keys.

Currently, DWS row-store and column-store tables support range partitioning and list partitioning.

- Range partitioning

A table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. Each range has a dedicated partition for data storage.

- Core feature: Data is distributed by consecutive intervals. It is used in scenarios where data is ordered and retrieved by range.
- Scenario: Partition keys increase linearly or have continuous intervals. Data query is usually performed around a certain range, and new data naturally falls into a new range. For example, data about orders is partitioned by time range, and data about user levels is partitioned by value range.
- Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition. This is the most commonly used partitioning policy. Currently, range partitioning only allows the use of the range partitioning policy.

- List partitioning

List partitioning allocates records to partitions based on the key values in each partition. The key values do not overlap in different partitions. Each set of key values gets its own partition for storing related data. List partitioning is supported only by clusters of version 8.1.3 or later.

- Core feature: Data is divided based on discrete enumerated values, which are used as fixed classification category or high-frequency filter criteria.

- Scenario: Partition key values are fixed and discrete. Data query is usually based on a certain category, and key values do not increase irregularly. For example, the user information table is partitioned by province, and each province corresponds to a partition.
- In list partitioning, data is mapped to a created partition based on the partition key value. If the data can be mapped to, it is inserted into the specific partition. If it cannot be mapped to, error messages are returned.

Major Differences Between Range Partitioning and List Partitioning

Table 13-119 Range partitioning and list partitioning

Dimension	Range Partitioning	List Partitioning
Data distribution	Continuous range (time and value)	Discrete enumerations values (classification and status)
Query optimization	Range filter (BETWEEN, >, <)	Exact match (= and IN)
Scenario	Ranges need to be added or deleted regularly (for example, rolling by month).	Fixed categories need to be maintained (such as regions and types).
Typical Case	Log, time series, and numeric archiving	Regional division, status isolation, and multi-tenant data

Advantages of Partitioning

- The performance of some types of queries can be greatly improved, especially when rows with high access rates in a table are located in a single partition or a few partitions. Partitioning narrows the range of data search and improves data access efficiency.
- When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of reads scattered across the whole table.
- Bulk loads and deletion can be performed by adding or removing partitions, if that requirement is planned into the partitioning design. It also entirely avoids the **VACUUM** overhead caused by a bulk **DELETE** (only for range partitioning).

Precautions

- A single table can have up to 32,767 partitions with a total of 32,767 boundary values. For better performance, keep this value at or below 1,000.
- A partitioned table supports unique and primary key constraints. The constraint keys of these constraints contain all partition keys.

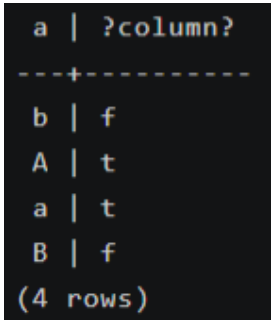
Syntax

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
(
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
  | table_constraint
  | LIKE source_table [ like_option [...] ] }, ... ]
)
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ COMPRESS | NOCOMPRESS ]
[ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { [ HASH ] ( column_name ) } } ]
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
PARTITION BY {
  {VALUES (partition_key)} |
  {RANGE (partition_key) ( partition_less_than_item [, ... ] )} |
  {RANGE (partition_key) ( partition_start_end_item [, ... ] )} |
  {LIST (partition_key) (list_partition_item [, ...])}
} [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

CREATE TABLE PARTITION Main Clause Parameters

Table 13-120 CREATE TABLE PARTITION main clause parameters

Parameter	Description	Value Range or Example
IF NOT EXISTS	Does not throw an error if a table with the same name exists. A notice is issued in this case.	-
partition_table_name	Name of the partitioned table	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
column_name	Specifies the name of a column to be created in the new table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
data_type	Specifies the data type of the column.	-

Parameter	Description	Value Range or Example
COLLATE collation	The COLLATE clause defines the collation rule of a column. The supported data types include char, varchar, text, nchar, and nvarchar. The collation rule significantly impacts the sequence of query results, particularly when using SELECT ... ORDER BY. If no collation rule is specified, the default collation rule C (case-sensitive) is used. Common collation rules: • C: The collation uses the byte order of the binary code and is sensitive to case, accents, and special characters. Data can indeed be compared based on its ASCII code values. • case_insensitive: The collation is case-insensitive. It is supported by clusters of version 8.1.3.x or later. • en_US.utf8: The collation uses the American English with a UTF-8 encoding and is case insensitive. You can query collation rules supported by DWS from the PG_COLLATION system catalog. <pre>SELECT * FROM pg_collation;</pre> To check the default collation rule of the current database, run the following statement: <pre>SELECT pg_database.datcollate AS current_collation FROM pg_catalog.pg_database WHERE pg_database.datname = pg_catalog.current_database();</pre>	A case-insensitive collation rule for the values of a column <pre>CREATE TABLE t1 (a text collate case_insensitive); INSERT INTO t1 values('a'),('A'), ('b'),('B'); SELECT a, a='a' FROM t1;</pre> The following figure shows that A and a are equivalent. 

Parameter	Description	Value Range or Example
CONSTRAINT constraint_name	Specifies a name for a column or table constraint. The optional constraint clauses specify constraints that new or updated rows must satisfy for an insert or update operation to succeed. There are two ways to define constraints: • A column constraint is defined as part of a column definition, and it is bound to a particular column. • A table constraint is not bound to any particular columns but can apply to more than one column.	For details about the supported constraints, see Table 13-121 .
LIKE source_table [like_option ...]	Specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.	For details about the supported constraints, see LIKE Parameters .
WITH	Specifies an optional storage parameter for a table or an index.	For details about the supported WITH parameters, see Table 13-122 .
COMPRESS / NOCOMPRESS	Specifies the keyword COMPRESS during the creation of a table, so that the compression feature is triggered in the case of a bulk INSERT operation. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. If NOCOMPRESS is specified, the table is not compressed.	Default value: NOCOMPRESS , tuple data is not compressed before storage.

Parameter	Description	Value Range or Example
DISTRIBUTE BY	Specifies how the table is distributed or replicated between DNs.	• REPLICATION: Each row in the table exists on all DNs, that is, each DN has complete table data. • ROUNDROBIN: Each row in the table is sent to each DN in turn. Therefore, data is evenly distributed on each DN. This value is supported only in 8.1.2 or later. • HASH (column_name): Each row of the table will be placed into all the DNs based on the hash value of the specified column. For details about other restrictions, see DISTRIBUTE BY Parameter Description. CAUTION • When DISTRIBUTE BY HASH (column_name) is specified, the primary key and its unique index must contain the column_name column. • When DISTRIBUTE BY HASH (column_name) in a referenced table is specified, the foreign key of the reference table must contain the column_name column.
TO { GROUP groupname NODE (nodename [, ...]) }	TO GROUP specifies the Node Group in which the table is created. Currently, it cannot be used for HDFS tables. TO NODE is used for internal scale-out tools.	-

Parameter	Description	Value Range or Example
{ ENABLE DISABLE } ROW MOVEMENT	Specifies whether to relocate a row within a table. If the new value of a partition key in a row belongs to another partition, an error message is displayed or the row is moved to the corresponding partition. • ENABLE: The row is moved to the corresponding partition. If partition keys are frequently updated, enable this parameter. • DISABLE: The row is not moved to the corresponding partition. If partition keys are updated, an error is reported. If data cross-partition update is strictly prohibited, disable this parameter. The default value is DISABLE. Suggestions: • Scenario: To enable row movement, you need to change partition key values through UPDATE operations and can accept its impact on performance. • Performance: Enabling row movement will increase the overhead of UPDATE operations. Exercise caution when enabling this parameter in high-concurrency scenarios. • Alternative solution: If you need to frequently adjust the partition ownership, delete the original row and then insert a new row, or exchange partitions.	The date column (partition key) of a partitioned table is partitioned by quarter into p_2023q1 and p_2023q2. The value 2023-02-15 of a row belongs to the partition p_2023q1 in the first quarter. After the value is updated to 2023-05-15, you can configure ROW MOVEMENT to determine whether to move the data in this row. • ENABLE: The row movement is enabled. The data is moved to the new partition p_2023q2. • DISABLE: The row movement is disabled. The system reports an error. For details about examples, see Examples: Enabling and Disabling ROW MOVEMENT.

Parameter	Description	Value Range or Example
	CAUTION To enable cross-partition update, you need to enable row movement. However, if SELECT FOR UPDATE is executed concurrently to query the partitioned table, the query results may be inconsistent. Therefore, exercise caution when performing this operation.	

CONSTRAINT constraint_name Parameter

```
{ NOT NULL |  
  NULL |  
  CHECK ( expression ) |  
  DEFAULT default_expr |  
  UNIQUE [ NULLS [NOT] DISTINCT | NULLS IGNORE ] index_parameters |  
  PRIMARY KEY index_parameters }  
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Table 13-121 CONSTRAINT constraint_name parameters

Parameter	Description	Value Range or Example
NOT NULL	Specifies that the column is not allowed to contain NULL values.	-
NULL	Specifies that the column can store NULL values. This is the default. This clause is only provided for compatibility with non-standard SQL databases. You are advised not to use this clause.	-

Parameter	Description	Value Range or Example
CHECK (condition) [NO INHERIT]	Specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed, or an error is reported and the execution fails. A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns. A constraint marked with NO INHERIT will not propagate to child tables.	Single-column constraints: The price must be a positive number. <pre>CREATE TABLE products (product_id INT, price DECIMAL(10,2) CHECK (price > 0), discount DECIMAL(3,2));</pre> Insert a data record where the price is a negative number. <pre>INSERT INTO products (product_id, price, discount) VALUES (2, -10.00, 1.1);</pre> The constraint is violated, so an error is reported and the insertion fails. <pre>ERROR: dn_6003_6004: new row for relation "products" violates check constraint "products_price_check"</pre> Table constraints (on multiple columns): Create an inventory table with two columns available and reserved. One column indicates available quantity and the other column indicates reserved quantity. The available quantity must be greater than or equal to the reserved quantity. <pre>CREATE TABLE inventory (item_id INT PRIMARY KEY, quantity_on_hand INT, quantity_reserved INT, CHECK (quantity_on_hand >= quantity_reserved));</pre>

Parameter	Description	Value Range or Example
DEFAULT default_expr	Assigns a default data value for a column. The value can be any variable-free expressions (Subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is NULL.	-

Parameter	Description	Value Range or Example
UNIQUE [NULLS [NOT] DISTINCT NULLS IGNORE] index_parameters UNIQUE [NULLS [NOT] DISTINCT NULLS IGNORE] (column_name [, ...]) index_parameters	Specifies that a group of one or more columns of a table can contain only unique values. The [NULLS [NOT] DISTINCT NULLS IGNORE] field is used to specify how to process null values in the index column of the Unique index.	Default value: NULL can be inserted repeatedly. When the inserted data is compared with the original data in the table, the NULL value can be processed in any of the following ways: • NULLS DISTINCT: Each NULL value is regarded as a different value, that is, NULL values can be inserted repeatedly. • NULLS NOT DISTINCT: All NULL values are regarded as the same value. That is, only one NULL value entry is retained in the index. NULL values cannot be inserted repeatedly. Only distinct non-NULL values can be inserted. • NULLS IGNORE: NULL values are skipped during the equivalent comparison. If all index columns are NULL, NULL values can be inserted repeatedly. If some index columns are NULL, data can be inserted only when Not NULL values are different. Table 13-124 lists the behaviors of the three processing modes. For details, see Application Differences Among NULLS DISTINCT, NULLS NOT DISTINCT, and NULLS IGNORE.

Parameter	Description	Value Range or Example
		CAUTION If DISTRIBUTE BY REPLICATION is not specified, the column table that contains only unique values must contain distribution columns.
PRIMARY KEY index_parameters PRIMARY KEY (column_name [, ...]) index_parameters	Specifies the primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values. Only one primary key can be specified for a table. CAUTION If DISTRIBUTE BY REPLICATION is not specified, the column set with a primary key constraint must contain distributed columns.	-
DEFERRABLE NOT DEFERRABLE	Sets whether the constraint can be deferrable. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction using the SET CONSTRAINTS command. NOT DEFERRABLE is the default value. Currently, only UNIQUE and PRIMARY KEY constraints of row-store tables accept this clause. All the other constraints are not deferrable.	-

Parameter	Description	Value Range or Example
INITIALLY IMMEDIATE INITIALLY DEFERRED	If a constraint is deferrable, this clause specifies the default time to check the constraint. • If the constraint is INITIALLY IMMEDIATE (default value), it is checked after each statement. • If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered using the SET CONSTRAINTS command.	-

table_constraint Parameters

Table constraints are similar to that of column constraints. For details, see [CONSTRAINT constraint_name Parameter](#).

```
{ CHECK ( expression ) |  
  UNIQUE [ NULLS [NOT] DISTINCT | NULLS IGNORE ] ( column_name [, ... ] ) index_parameters |  
  PRIMARY KEY ( column_name [, ... ] ) index_parameters }  
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

LIKE Parameters

like_option is as follows:

```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |  
  REOPTIONS | DISTRIBUTION | ALL }
```

The new table and original table are decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Default expressions for the copied column definitions will only be copied if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having default values **NULL**.

NOT NULL constraints are always copied to the new table. **CHECK** constraints will only be copied if **INCLUDING CONSTRAINTS** is specified; other types of constraints will never be copied. These rules also apply to column constraints and table constraints.

Columns and constraints copied by **LIKE** are not merged with the same name. If the same name is specified explicitly or in another **LIKE** clause, an error is reported.

- Any indexes on the source table will not be created on the new table, unless the **INCLUDING INDEXES** clause is specified.

- **STORAGE** settings for the copied column definitions will only be copied if **INCLUDING STORAGE** is specified. The default behavior is to exclude **STORAGE** settings.
- Comments for the copied columns, constraints, and indexes will only be copied if **INCLUDING COMMENTS** is specified. The default behavior is to exclude comments.
- If **INCLUDING REOPTIONS** is specified, the new table will copy the storage parameter (**WITH** clause of the source table) of the source table. The default behavior is to exclude partition definition of the storage parameter of the source table.
- If **INCLUDING DISTRIBUTION** is specified, the new table will copy the distribution information of the source table, including distribution type and column, and the new table cannot use **DISTRIBUTE BY** clause. The default behavior is to exclude distribution information of the source table.
- **INCLUDING ALL** is an abbreviated form of **INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS INCLUDING REOPTIONS INCLUDING DISTRIBUTION**.

WITH Clause Parameters

Table 13-122 WITH parameters

Parameter	Description	Value Range
FILLFACTOR	The fillfactor of a table is a percentage between 10 and 100. 100 (complete packing) is the default value. When a smaller fillfactor is specified, INSERT operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The parameter has no meaning for column-store tables.	10~100

Parameter	Description	Value Range
ORIENTATION	Determines the storage mode of the data in the table. CAUTION orientation cannot be modified.	• COLUMN: The data will be stored in columns. • ROW (default value): The data will be stored in rows. • ORC: The data of the table will be stored in ORC format (only HDFS table).
COMPRESSION	This parameter is available only for column-store tables. Value: LOW (default value), MIDDLE , or HIGH . NOTE Currently, row-store table compression is not supported.	-
MAX_BATCHROW	Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store tables.	10000 to 60000 Default value: 60000
PARTIAL_CLUSTER_ROWS	Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store tables.	The value is no less than 100,000. The value is the multiple of MAX_BATCHROW .
enable_delta	Specifies whether to enable delta tables in column-store tables. The parameter is only valid for column-store tables. Using column-store tables with delta tables is not recommended. This may cause disk bloat and performance deterioration due to delayed merge.	Default value: off
DELTAROW_THRESHOLD	A reserved parameter. The parameter is only valid for column-store tables.	0 to 60000 Default value: 6000

Parameter	Description	Value Range
COLD_TABLESPACE	Specifies the OBS tablespace for the cold partitions in a hot or cold table. This parameter is available only to column-store partitioned tables and cannot be modified. It must be used together with STORAGE_POLICY . The parameter STORAGE_POLICY can be left unconfigured. In this case, the default value default_obs_tbs is used.	Valid OBS tablespace name.

Parameter	Description	Value Range
STORAGE_POLICY	Specifies the rule for switching between hot and cold partitions. This parameter is used only for multi-temperature tables. This parameter must be used together with COLD_TABLESPACE .	Value range: <i>Cold and hot switchover policy name</i>. <i>Cold and hot switchover threshold</i>. Currently, only LMT and HPN policies are supported. LMT indicates that the switchover is performed based on the last update time of partitions. HPN indicates the switchover is performed based on a fixed number of reserved hot partitions. • LMT:<i>[day]</i>: Switch the hot partition data that is not updated in the last <i>[day]</i> days to the OBS tablespace as cold partition data. <i>[day]</i> is an integer ranging from 0 to 36500, in days. • HPN: <i>[hot_partition_num]</i>: <i>[hot_partition_num]</i> indicates the number of hot partitions (with data) to be retained. The rule is to find the maximum sequence ID of the partitions with data. The partitions without data whose sequence ID is greater than the maximum sequence ID are hot partitions, and <i>[hot_partition_num]</i> partitions are retained as hot partitions in descending order according to the sequence ID. A partition whose sequence ID is smaller than the minimum sequence ID of the

Parameter	Description	Value Range
		retained hot partition is a cold partition. During hot and cold partition switchover, data needs to be migrated to the OBS tablespace. <i>[hot_partition_num]</i> is an integer ranging from 0 to 1600. CAUTION For a LIST partition, you are advised to use the HPN policy with caution. Otherwise, the new partition may not be a hot partition.
ENABLE_MULTI_T EMP_DISKCACHE	This parameter improves the performance of tables that separate cold and hot data. It is only applicable to tables that support this feature and is supported in versions 9.1.0 and later.	on or off Default value: on/true

Parameter	Description	Value Range
PERIOD	Specifies the period of automatically creating partitions and enables the automatic partition creation function. Only row-store and column-store range partitioned tables, time series tables, and cold and hot tables are supported. The partition key must be unique and its type can only be TIMESTAMP[(p)] [WITHOUT TIME ZONE], TIMESTAMP[(p)] [WITH TIME ZONE] or DATE. MAXVALUE partitions are not supported. The value of (nowTime - boundaryTime)/ PERIOD must be less than the upper limit of the number of partitions, where nowTime indicates the current time and boundaryTime indicates the earliest partition boundary time. It cannot be used on midrange computers and acceleration clusters.	1 hour ~ 100 years CAUTION • In a database compatible with Teradata or MySQL, if the partition key type is DATE, PERIOD cannot be less than 1 day. • If PERIOD is set when a partitioned table is created, you can specify only the partition key. Two default partitions are created during table creation. The time ranges of the two default partitions are both PERIOD. The boundary time of the first default partition is the first hour, day, week, month, or year past the current time. The time unit is selected based on the maximum unit of PERIOD. The boundary time of the second default partition is the boundary time of the first partition plus PERIOD. Assume that the current time is 2022-02-17 16:32:45, and the boundary of the first default partition is described in Table 13-125. For more information about the default partitions, see Example: Creating a Partitioned Table with Automatic Partition Management and No Specified Partitions.

Parameter	Description	Value Range
TTL	Specifies the partition expiration time in partition management and enables the automatic partition deletion function. This parameter cannot be set separately. You must set PERIOD in advance or at the same time. The value of this parameter must be greater than or equal to that of PERIOD .	1 hour ~ 100 years CAUTION PERIOD indicates that data is partitioned by time period. The partition size may affect the query performance. The proc_add_partition (relname regclass, boundaries_interval interval) function is automatically invoked to create a partition after each period. Time To Live (TTL) specifies the data storage period of the table. The data that exceeds the TTL period will be cleared. To do this, the function in proc_drop_partition (relname regclass, older_than interval) is automatically invoked based on the period. The PERIOD and TTL values are of the Interval type, for example, 1 hour, 1 day, 1 week, 1 month, 1 year, and 1 month 2 day 3 hour .

Parameter	Description	Value Range
COLVERSION	Specifies the version of the column-store format. Switching between different storage formats is supported. However, the storage format of a partitioned table cannot be switched.	• 1.0: Each column in a column-store table is stored in a separate file. The file name is relfilenode.C1.0, relfilenode.C2.0, relfilenode.C3.0, or similar. • 2.0: All columns of a column-store table are combined and stored in a file. The file is named relfilenode.C1.0. Default value: 2.0 The value of COLVERSION can only be set to 2.0 for OBS multi-temperature tables. CAUTION When creating a column-store table, set COLVERSION to 2.0 . Compared with the 1.0 storage format, the performance is significantly improved: 1. The time required for creating a column-store wide table is significantly reduced. 2. In the Roach data backup scenario, the backup time is significantly reduced. 3. The build and catch up time is greatly reduced.
SKIP_FPI_HINT	Indicates whether to skip the hint bits operation when the full-page writes (FPW) log needs to be written during sequential scanning.	Default value: false CAUTION If SKIP_FPI_HINT is set to true and the checkpoint operation is performed on a table, no Xlog will be generated when the table is sequentially scanned. This applies to intermediate tables that are queried less frequently, reducing the size of Xlogs and improving query performance.

PARTITION BY Parameters

- **partition_less_than_item** is as follows:
`PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE } )`
- **partition_start_end_item** is as follows:
`PARTITION partition_name {
 {START(partition_value) END (partition_value) EVERY (interval_value)} |
 {START(partition_value) END ({partition_value | MAXVALUE})} |
 {START(partition_value)} |
 {END({partition_value | MAXVALUE})}
}`
- **list_partition_item**:
`PARTITION partition_name VALUES ( { (partition_value) [, ...] | DEFAULT } )`

Table 13-123 PARTITION BY Parameters

Parameter	Description	Value Range
PARTITION BY RANGE(partition_key)	The syntax specifying range partitioning. partition_key indicates the name of a partition key.	- Assume that the VALUES LESS THAN syntax is used. CAUTION In this case, a maximum of four partition keys are supported, and the partition key must be a column name. If there are multiple partition keys, a column name can appear only once, and two adjacent partition keys must be separated by a comma (,). Data types supported by the partition keys are as follows: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE PRECISION, CHARACTER VARYING(n), VARCHAR(n), CHARACTER(n), CHAR(n), CHARACTER, CHAR, TEXT, NVARCHAR2, NAME, TIMESTAMP[(p)] [WITHOUT TIME ZONE], TIMESTAMP[(p)] [WITH TIME ZONE], and DATE. - Assume that the START END syntax is used. CAUTION In this case, only one partition key is supported. Data types supported by the partition key are as follows: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE PRECISION, TIMESTAMP[(p)] [WITHOUT TIME

Parameter	Description	Value Range
		ZONE], TIMESTAMP[(p)] [WITH TIME ZONE], and DATE.
PARTITION BY LIST (partition_key,...)	The syntax specifying list partitioning. partition_key indicates the name of a partition key.	In list partitioning, partition keys support the following data types: TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC/DECIMAL, TEXT, NVARCHAR2, VARCHAR(n), CHAR, BPCHAR, TIME, TIME WITH TIMEZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, DATE, INTERVAL and SMALLDATETIME. CAUTION In list partitioning, a partition key has a maximum of four columns.

Parameter	Description	Value Range
partition_less_than_item	Partition definition syntax in range partitioning. partition_name is the name of a range partition. partition_value is the upper limit of range partition, and the value depends on the type of partition_key. MAXVALUE can specify the upper boundary of a range partition, and it is commonly used to specify the upper boundary of the last range partition. <pre>PARTITION partition_name VALUES LESS THAN ({ partition_value DEFAULT })</pre> CAUTION • Upper boundaries must be specified for each partition. • The types of upper boundaries must be the same as those of partition keys. • In a partition list, partitions are arranged in ascending order of upper boundary values. Therefore, a partition with a certain upper boundary value is placed before another partition with a larger upper boundary value. • If a partition key consists of multiple columns, the columns are used for partitioning in sequence. The first column is preferred to be used for partitioning. If the values of the first columns are the same, the second column is used. The subsequent columns are used in the same manner.	-

Parameter	Description	Value Range
partition_start_end_item	The syntax of using the start value and interval value to define a range partition. <pre>PARTITION partition_name {START (partition_value) END (partition_value) EVERY (interval_value)} {START (partition_value) END (partition_value) MAXVALUE} {START(partition_value) {END (partition_value) MAXVALUE}}</pre>	The following table describes the parameters. • partition_name: name or name prefix of a range partition. It is the name prefix only in the following cases (assuming that partition_name is p1): – If START+END+EVERY is used, the names of partitions will be defined as p1_1, p1_2, and the like. For example, if PARTITION p1 START(1) END(4) EVERY(1) is defined, the generated partitions are [1, 2), [2, 3), and [3, 4), and their names are p1_1, p1_2, and p1_3. In this case, p1 is a name prefix. – If the defined statement is in the first place and has START specified, the range (MINVALUE, START) will be automatically used as the first actual partition, and its name will be p1_0. The other partitions are then named p1_1, p1_2, and the like. For example, if PARTITION p1 START(1), PARTITION p2 START(2) is defined, generated partitions are (MINVALUE, 1), [1, 2), and [2, MAXVALUE), and their names will be p1_0, p1_1, and p2.

Parameter	Description	Value Range
		In this case, p1 is a name prefix and p2 is a partition name. MINVALUE means the minimum value. • partition_value: start point value or end point value of a range partition. The value depends on partition_key and cannot be MAXVALUE. • interval_value: width of each partition for dividing the [START, END) range. It cannot be MAXVALUE. If the value of (END – START) divided by EVERY has a remainder, the width of only the last partition is less than the value of EVERY. • MAXVALUE: maximum value. It is usually used to set the upper boundary for the last range partition.

Parameter	Description	Value Range
		CAUTION - If the defined statement is in the first place and has START specified, the range (MINVALUE, START) will be automatically used as the first actual partition. - The START END syntax must comply with the following rules: - The value of START (if any, same for the following situations) in each partition_start_end_item must be smaller than that of END. - In two adjacent partition_start_end_item statements, the value of the first END must be equal to that of the second START. - The value of EVERY in each partition_start_end_item must be a positive number (in ascending order) and must be smaller than END minus START. - Each partition includes the start value (unless it is MINVALUE) and excludes the end value. The format is as follows: [Start value, end value). - Partitions created by the same partition_start_end_item belong to the same tablespace. - If partition_name is a name prefix of a partition, the length must not exceed 57 bytes. If there are more than 57 bytes, the prefix will be automatically truncated. - When creating or modifying a partitioned table, ensure that the total number of partitions in the table does not exceed the maximum value (32767). - In statements for creating partitioned

Parameter	Description	Value Range
list_partition_item	Partition definition syntax in list partitioning. partition_name indicates the partition name. partition_value is an enumerated value of the list partition boundary. The value depends on the type of partition_key. DEFAULT indicates the default partition boundary. <code>PARTITION partition_name VALUES ({ (partition_value) [, ...] DEFAULT })</code>	The following conventions and constraints apply to list partitioned tables: • The partition whose boundary value is DEFAULT is the default partition. • Each list partitioned table can have only one DEFAULT partition. • The number of partitions in a partitioned table cannot exceed 32767, and the number of boundary values of all partitions cannot exceed 32767. • Regardless of the number of partition keys, the boundary of the DEFAULT partition can only be DEFAULT. • If a partition key consists of multiple columns, each partition_value must contain the values of all partition keys. If a partition key contains only one column, the parentheses on both sides of partition_value can be omitted. For details, see Example: Creating a List Partitioned Table with Multiple Partition Keys. • If the partition key consists of multiple columns, compare the values in the columns one by one. If a value is different from another value, regardless of their columns, they are different values.

Parameter	Description	Value Range
		- Each value of partition_value must be unique. - When data is inserted, if its partition key and value falls into the boundary of a non-DEFAULT partition, the data is written to the partition. Otherwise, the data is written to the DEFAULT partition.

Table 13-124 Processing of NULL values in index columns in unique indexes

Constraint	All Index Columns Are NULL	Some Index Columns Are NULL
NULLS DISTINCT	Can be inserted repeatedly.	Can be inserted repeatedly.
NULLS NOT DISTINCT	Cannot be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.
NULLS IGNORE	Can be inserted repeatedly.	Cannot be inserted if the non-null values are equal. Can be inserted if the non-null values are not equal.

Table 13-125 Partition boundaries

period	Maximum PERIOD Unit	Boundary of First Default Partition
1hour	Hour	2022-02-17 17:00:00
1day	Day	2022-02-18 00:00:00
1month	Month	2022-03-01 00:00:00
13month	Year	2023-01-01 00:00:00

DISTRIBUTE BY Parameter Description

Default value: determined by the GUC parameter **default_distribution_mode**

- When **default_distribution_mode** is set to **roundrobin**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 - a. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 - b. If the primary key or unique constraint is not included during table creation, round-robin distribution is selected.
- When **default_distribution_mode** is set to **hash**, the default value of **DISTRIBUTE BY** is selected according to the following rules:
 - a. If the primary key or unique constraint is included during table creation, hash distribution is selected. The distribution column is the column corresponding to the primary key or unique constraint.
 - b. If the primary key or unique constraint is not included during table creation but there are columns whose data types can be used as distribution columns, hash distribution is selected. The distribution column is the first column whose data type can be used as a distribution column.
 - c. If the primary key or unique constraint is not included during table creation and no column whose data type can be used as a distribution column exists, round-robin distribution is selected.

The following data types can be used as distribution columns:

- Integer types: **TINYINT**, **SMALLINT**, **INT**, **BIGINT**, and **NUMERIC/DECIMAL**
- Character types: **CHAR**, **BPCHAR**, **VARCHAR**, **VARCHAR2**, **NVARCHAR2**, and **TEXT**
- Date/time types: **DATE**, **TIME**, **TIMETZ**, **TIMESTAMP**, **TIMESTAMPTZ**, **INTERVAL**, and **SMALLDATETIME**

Example: Using the LESS THAN Syntax to Create a Range Partitioned Table

The range partitioned table **customer_address** has four partitions and their partition keys are of the integer type. The ranges of the partitions are as follows: **ca_address_sk < 2450815**, **2450815 <= ca_address_sk < 2451179**, **2451179 <= ca_address_sk < 2451544**, **2451544 <= ca_address_sk**.

```
CREATE TABLE customer_address
(
  ca_address_sk    INTEGER          NOT NULL ,
  ca_address_id    CHARACTER(16)    NOT NULL ,
  ca_street_number CHARACTER(10)      ,
  ca_street_name   CHARACTER varying(60) ,
  ca_street_type   CHARACTER(15)      ,
  ca_suite_number  CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
  PARTITION P1 VALUES LESS THAN(2450815),
  PARTITION P2 VALUES LESS THAN(2451179),
  PARTITION P3 VALUES LESS THAN(2451544),
  PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

View the information of the partitioned table.

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='customer_address'::regclass ORDER BY 1;
```

relname	boundaries
customer_address	
p1	{2450815}
p2	{2451179}
p3	{2451544}
p4	{NULL}

(5 rows)

Query the number of rows in the **P1** partition.

```
SELECT count(*) FROM customer_address PARTITION (P1);  
SELECT count(*) FROM customer_address PARTITION FOR (2450815);
```

Examples: Enabling and Disabling ROW MOVEMENT

If **ROW MOVEMENT** is enabled, the data of a row is moved to a new partition when the partition key of the row is updated to a new value and the new value belongs to another partition.

If a partition key is frequently updated, enabling **ROW MOVEMENT** may cause extra I/O overhead. To enable it, evaluate service requirements first.

Create a table partitioned by date and enable **ROW MOVEMENT**.

```
DROP TABLE IF EXISTS sale_data;  
CREATE TABLE sales_data (  
    sale_id INT,  
    product_name VARCHAR(100),  
    sale_date DATE,  
    amount DECIMAL(10, 2)  
)  
PARTITION BY RANGE (sale_date) (  
    PARTITION p_2023q1 VALUES LESS THAN ('2023-04-01'),  
    PARTITION p_2023q2 VALUES LESS THAN ('2023-07-01'),  
    PARTITION p_2023q3 VALUES LESS THAN ('2023-10-01'),  
    PARTITION p_2023q4 VALUES LESS THAN ('2024-01-01')  
)  
ENABLE ROW MOVEMENT;
```

Insert test data.

```
INSERT INTO sales_data (sale_id, product_name, sale_date, amount)  
VALUES  
(1, 'Product A', '2023-02-15', 1000.00), -- It belongs to p_2023q1.  
(2, 'Product B', '2023-05-20', 1500.00); -- It belongs to p_2023q2.
```

Query the data distribution of each partition.

```
SELECT tableoid::regclass AS partition, * FROM sales_data ORDER BY sale_id;
```

partition	sale_id	product_name	sale_date	amount
2147617615	1	Product A	2023-02-15 00:00:00	1000.00
2147617616	2	Product B	2023-05-20 00:00:00	1500.00

(2 rows)

Update the partition key (which is moved from **p_2023q1** to **p_2023q2**).

```
UPDATE sales_data SET sale_date = '2023-05-01' WHERE sale_id = 1;
```

Query the data distribution of each partition again to confirm that the row has been moved.

```
SELECT tableoid::regclass AS partition, * FROM sales_data ORDER BY sale_id;
```

partition	sale_id	product_name	sale_date	amount
2147617616	1	Product A	2023-05-01 00:00:00	1000.00
2147617616	2	Product B	2023-05-20 00:00:00	1500.00

(2 rows)

Disable **ROW MOVEMENT**.

```
ALTER TABLE sales_data DISABLE ROW MOVEMENT;
```

Modify the partition key again (which is move from **p_2023q2** back to **p_2023q2**).

```
UPDATE sales_data SET sale_date = '2023-02-15' WHERE sale_id = 1;
```

View the command output. It indicates that **ROW MOVEMENT** is disabled and the partition key cannot be updated.

```
gaussdb=> UPDATE sales_data SET sale_date = '2023-02-15' WHERE sale_id = 1;  
ERROR:  dn_6003_6004: fail to update partitioned table "sales_data"  
DETAIL:  disable row movement
```

Example: Using the START END Syntax to Create a Column-store Range Partitioned Table

```
CREATE TABLE customer_address_SE  
(  
    ca_address_sk    INTEGER          NOT NULL ,  
    ca_address_id    CHARACTER(16)    NOT NULL ,  
    ca_street_number CHARACTER(10)      ,  
    ca_street_name   CHARACTER varying(60) ,  
    ca_street_type   CHARACTER(15)      ,  
    ca_suite_number  CHARACTER(10)  
)  
WITH (ORIENTATION = COLUMN)  
DISTRIBUTE BY HASH (ca_address_sk)  
PARTITION BY RANGE(ca_address_sk)  
(  
    PARTITION p1 START(1) END(1000) EVERY(200),  
    PARTITION p2 END(2000),  
    PARTITION p3 START(2000) END(5000)  
);
```

View the information of the partitioned table.

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='customer_address_SE'::regclass ORDER  
BY 1;  
 relname | boundaries  
-----+-----  
customer_address_se |  
p1_0          | {1}  
p1_1          | {201}  
p1_2          | {401}  
p1_3          | {601}  
p1_4          | {801}  
p1_5          | {1000}  
p2            | {2000}  
p3            | {5000}  
(9 rows)
```

Example: Using the START END Syntax to Create and Modify a Range Partitioned Table

Create a temporary schema.

```
CREATE SCHEMA tpcds;  
SET CURRENT_SCHEMA TO tpcds;
```

Create a partitioned table with the partition key of the integer type.

```
CREATE TABLE tpcds.startend_pt (c1 INT, c2 INT)  
DISTRIBUTE BY HASH (c1)  
PARTITION BY RANGE (c2) (  
    PARTITION p1 START(1) END(1000) EVERY(200),  
    PARTITION p2 END(2000),  
    PARTITION p3 START(2000) END(2500),  
    PARTITION p4 START(2500),  
    PARTITION p5 START(3000) END(5000) EVERY(1000)  
)  
ENABLE ROW MOVEMENT;
```

View the information of the partitioned table.

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='tpcds.startend_pt'::regclass ORDER BY  
1;
```

relname	boundaries
p1_0	{1}
p1_1	{201}
p1_2	{401}
p1_3	{601}
p1_4	{801}
p1_5	{1000}
p2	{2000}
p3	{2500}
p4	{3000}
p5_1	{4000}
p5_2	{5000}
tpcds.startend_pt	

(12 rows)

Import data and check the data volume in the partition.

```
INSERT INTO tpcds.startend_pt VALUES (GENERATE_SERIES(0, 4999), GENERATE_SERIES(0, 4999));  
SELECT COUNT(*) FROM tpcds.startend_pt PARTITION FOR (0);
```

count
1

(1 row)

```
SELECT COUNT(*) FROM tpcds.startend_pt PARTITION (p3);
```

count
500

(1 row)

View the information of the partitioned table.

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='tpcds.startend_pt'::regclass ORDER BY  
1;
```

relname	boundaries
p1_0	{1}
p1_1	{201}
p1_2	{401}
p1_3	{601}
p1_4	{801}
p1_5	{1000}
p2	{2000}
p3	{2500}
p4	{3000}
p5_1	{4000}
p6_1	{5300}
p6_2	{5600}
p6_3	{5900}

```
p71      | {6000}  
q1_1     | {4250}  
q1_2     | {4500}  
q1_3     | {4750}  
q1_4     | {5000}  
tpcds.startend_pt |  
(19 rows)
```

Example: Creating a List Partitioned Table with Partition Keys

```
CREATE TABLE data_list  
(  
    id int,  
    time int,  
    sarlay decimal(12,2)  
)  
PARTITION BY LIST (time)  
(  
    PARTITION P1 VALUES (202209),  
    PARTITION P2 VALUES (202210,202208),  
    PARTITION P3 VALUES (202211),  
    PARTITION P4 VALUES (202212),  
    PARTITION P5 VALUES (202301)  
);
```

Example: Creating a List Partitioned Table with Multiple Partition Keys

A partitioned table has two partition keys: **period** and **city**.

```
CREATE TABLE sales_info  
(  
    sale_time timestamptz,  
    period    int,  
    city      text,  
    price     numeric(10,2),  
    remark    varchar2(100)  
)  
DISTRIBUTE BY HASH(sale_time)  
PARTITION BY LIST (period, city)  
(  
    PARTITION north_2022 VALUES (('202201', 'north1'), ('202202', 'north2')),  
    PARTITION south_2022 VALUES (('202201', 'south1'), ('202202', 'south2'), ('202203', 'south2')),  
    PARTITION rest VALUES (DEFAULT)  
);
```

Example: Creating a Partitioned Table with Automatic Partition Management and No Specified Partitions

Set **PERIOD** to **1 day** and partition key to **time**.

```
CREATE TABLE time_part  
(  
    id integer,  
    time timestamp  
) with (PERIOD='1 day')  
partition by range(time);
```

Two default partitions are created during table creation. The boundary time of the first default partition is the start of the day later than the current time, that is, 2022-12-13 00:00:00. The boundary time of the second default partition is the boundary time of the first partition plus PERIOD, that is, 2022-12-13 00:00:00+1 day=2022-12-14 00:00:00.

```
SELECT now();  
       now  
-----
```

```
2022-12-12 20:41:21.603172+08
(1 row)

SELECT relname, boundaries FROM pg_partition p where p.parentid='time_part'::regclass ORDER BY 1;
 relname | boundaries
-----+-----
default_part_1 | {"2022-12-13 00:00:00"}
default_part_2 | {"2022-12-14 00:00:00"}
time_part |
(3 rows)
```

Example: Creating a Partitioned Table with Automatic Partition Management and Specified Partitions

```
CREATE TABLE CPU(
  id integer,
  idle numeric,
  IO numeric,
  scope text,
  IP text,
  time timestamp
) with (TTL='7 days',PERIOD='1 day')
partition by range(time)
(
  PARTITION P1 VALUES LESS THAN('2022-01-05 16:32:45'),
  PARTITION P2 VALUES LESS THAN('2022-01-06 16:56:12')
);
```

Example: Creating a Partitioned Table by Month

Create partitioned table **customer_address**, which contains 13 partitions and their partition keys are of the date type.

```
CREATE TABLE customer_address
(
  ca_address_sk integer NOT NULL,
  ca_address_date date NOT NULL
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE (ca_address_date)
(
  PARTITION p202001 VALUES LESS THAN('20200101'),
  PARTITION p202002 VALUES LESS THAN('20200201'),
  PARTITION p202003 VALUES LESS THAN('20200301'),
  PARTITION p202004 VALUES LESS THAN('20200401'),
  PARTITION p202005 VALUES LESS THAN('20200501'),
  PARTITION p202006 VALUES LESS THAN('20200601'),
  PARTITION p202007 VALUES LESS THAN('20200701'),
  PARTITION p202008 VALUES LESS THAN('20200801'),
  PARTITION p202009 VALUES LESS THAN('20200901'),
  PARTITION p202010 VALUES LESS THAN('20201001'),
  PARTITION p202011 VALUES LESS THAN('20201101'),
  PARTITION p202012 VALUES LESS THAN('20201201'),
  PARTITION p202013 VALUES LESS THAN(MAXVALUE)
);
```

Insert data.

```
INSERT INTO customer_address values('1','20200215');
INSERT INTO customer_address values('7','20200805');
INSERT INTO customer_address values('9','20201111');
INSERT INTO customer_address values('4','20201231');
```

Query a partition.

```
SELECT * FROM customer_address PARTITION(p202009);
ca_address_sk | ca_address_date
-----+-----
```

```
7 | 2020-08-05 00:00:00
(1 row)
```

Example: Using the START END Syntax to Create a Partitioned Table with Multiple Partitions at a Time

- Create a partitioned table **day_part**. Each day is a partition, and the partition key is a date.

```
CREATE table day_part(id int,d_time date)
DISTRIBUTE BY HASH (id)
PARTITION BY RANGE (d_time)
(PARTITION p1 START('2022-01-01') END('2022-01-31') EVERY(interval '1 day'));
ALTER TABLE day_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
```

- Create a partitioned table **week_part**, seven days as a partition, and the partition key is a date.

```
CREATE table week_part(id int,w_time date)
DISTRIBUTE BY HASH (id)
PARTITION BY RANGE (w_time)
(PARTITION p1 START('2021-01-01') END('2022-01-01') EVERY(interval '7 day'));
ALTER TABLE week_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
```

- Create the partitioned table **month_part**, each month as a partition, and the partition key is a date.

```
CREATE table month_part(id int,m_time date)
DISTRIBUTE BY HASH (id)
PARTITION BY RANGE (m_time)
(PARTITION p1 START('2021-01-01') END('2022-01-01') EVERY(interval '1 month'));
ALTER TABLE month_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
```

Example: Creating a Hot or Cold Table

Only a column-store partitioned table is supported. Use the default OBS tablespace. Set LMT to 30 for cold and hot switchover rules.

```
CREATE TABLE cold_hot_table
(
  W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)   ,
  W_STREET_NUMBER      CHAR(10)      ,
  W_STREET_NAME        VARCHAR(60)   ,
  W_STREET_ID          CHAR(15)      ,
  W_SUITE_NUMBER       CHAR(10)
)
WITH (ORIENTATION = COLUMN, storage_policy = 'LMT:30')
DISTRIBUTE BY HASH (W_WAREHOUSE_ID)
PARTITION BY RANGE(W_STREET_ID)
(
  PARTITION P1 VALUES LESS THAN(100000),
  PARTITION P2 VALUES LESS THAN(200000),
  PARTITION P3 VALUES LESS THAN(300000),
  PARTITION P4 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

Helpful Links

[ALTER TABLE PARTITION, DROP TABLE](#)

FAQs

- Practice: [Creating and Managing DWS Partitioned Tables](#)
- Practice: [Case: Reconstructing Partitioned Tables](#)

- "inserted partition key does not map to any table partition" Is Reported When Data Is Inserted into a Partitioned Table
- Error upper boundary of adding partition MUST overtop last existing partition Is Reported When a New Partition Is Added to a Range Partitioned Table

13.63 CREATE TEXT SEARCH CONFIGURATION

Function

CREATE TEXT SEARCH CONFIGURATION creates a text search configuration. A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching.

Precautions

- If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words. Subsequent **ALTER TEXT SEARCH CONFIGURATION** commands must be used to create mappings to make the configuration useful. If **COPY** option is specified, the parser, mapping and configuration option of text search configuration is copied automatically.
- If a schema name is specified, the text search configuration is created in the specified schema. Otherwise, the text search configuration is created in the current schema.
- The user who defines a text search configuration becomes its owner.
- **PARSER** and **COPY** options are mutually exclusive, because when an existing configuration is copied, its parser configuration is also copied.

Syntax

```
CREATE TEXT SEARCH CONFIGURATION name
( PARSER = parser_name | COPY = source_config )
[ WITH ( {configuration_option = value} [, ...] ) ];
```

Parameter Description

Table 13-126 CREATE TEXT SEARCH CONFIGURATION parameters

Parameter	Description	Value Range
name	Specifies the name (optionally schema-qualified) of the text search configuration to be created.	A string, which must comply with the naming convention .
parser_name	Specifies the name of the text search parser to use for this configuration.	-

Parameter	Description	Value Range
source_config	Specifies the name of an existing text search configuration to copy.	-
configuration_option	Specifies the configuration parameter of text search configuration is mainly for the parser executed by parser_name or contained by source_config .	Currently, the default, ngram, and zhparser parsers are supported. • The parser of the default type does not have the corresponding configuration_option. • For details about the configuration_option of the ngram and zhparser parsers, see Table 13-127.

Table 13-127 Configuration parameters for ngram and zhparser parsers

Parser	Parameters for adding an account	Description	Value Range
ngram	gram_size	Length of word segmentation	Integer, 1 to 4 Default value: 2
	punctuation_ignore	Whether to ignore punctuations	• true (default value): Ignore punctuations. • false: Do not ignore punctuations.
	graphsymbol_ignore	Whether to ignore graphical characters	• true: Ignore graphical characters. • false (default value): Do not ignore graphical characters.
zhparser	punctuation_ignore	Whether to ignore special characters including punctuations (\r and \n will not be ignored) in the word segmentation result	• true (default value): Ignore all the special characters including punctuations. • false: Do not ignore all the special characters including punctuations.

Parser	Parameters for adding an account	Description	Value Range
	seg_with_duality	Whether to aggregate segments with duality	• true: Aggregate segments with duality. • false (default value): Do not aggregate segments with duality.
	multi_short	Whether to execute long words composite divide	• true (default value): Execute long words composite divide. • false: Do not execute long words composite divide.
	multi_duality	Whether to aggregate segments in long words with duality	• true: Aggregate segments in long words with duality. • false (default value): Do not aggregate segments in long words with duality.
	multi_zmain	Whether to display key single words individually	• true: Display key single words individually. • false (default value): Do not display key single words individually.
	multi_zall	Whether to display all single words individually	• true: Display all single words individually. • false (default value): Do not display all single words individually.

Examples

Create a text search configuration named **ngram1** using the ngram parser.

```
CREATE TEXT SEARCH CONFIGURATION ngram1 (parser=ngram) WITH (gram_size = 2, grapsymbol_ignore = false);
```

Create a text search configuration named **ngram2** and copy all parameters of the existing configuration **ngram1**.

```
CREATE TEXT SEARCH CONFIGURATION ngram2 (copy=ngram1);
```

Create a text search configuration named **english_1** and use the default text parser.

```
CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
```

Helpful Links

[ALTER TEXT SEARCH CONFIGURATION](#), [DROP TEXT SEARCH CONFIGURATION](#)

13.64 CREATE TEXT SEARCH DICTIONARY

Function

CREATE TEXT SEARCH DICTIONARY creates a full-text retrieval dictionary. A full-text search dictionary is used to identify and process particular words during full-text retrieval.

Dictionaries are created by using predefined templates (defined in the **PG_TS_TEMPLATE** system catalog). Five types of dictionaries can be created, **Simple**, **Ispell**, **Synonym**, **Thesaurus**, and **Snowball**. Each type of dictionaries is used to handle different tasks.

Precautions

- A user with the **SYSADMIN** permission can create a dictionary. Then, the user automatically becomes the owner of the dictionary.
- A dictionary cannot be created in **pg_temp** mode.
- After a dictionary is created or modified, any modification to the user-defined dictionary definition file will not affect the dictionary in the database. To make such modifications take effect in the dictionary in the database, run the **ALTER** statement to update the definition file of the dictionary.
- The **TEMPLATE** parameter in a dictionary cannot be modified.
- To specify a dictionary, specify both the dictionary definition file path (**FILEPATH**) and the file name (the parameter varies based on dictionary types).
- The name of a dictionary definition file can contain only lowercase letters, digits, and underscores (_).
- The predefined dictionary file is stored in the **\$GAUSSHOME/share/postgresql/tsearch_data** directory.

Syntax

```
CREATE TEXT SEARCH DICTIONARY name (  
    TEMPLATE = template  
    [, option = value [, ... ]]  
);
```

Parameter Description

Table 13-128 CREATE TEXT SEARCH DICTIONARY parameters

Parameter	Description	Value Range
name	Specifies the name (optionally schema-qualified) of the full-text search dictionary to be created.	A string, which must comply with the naming convention .
template	Specifies a template name.	Templates (Simple , Synonym , Thesaurus , Ispell , and Snowball) defined in the PG_TS_TEMPLATE system catalog
option	Specifies the parameter name of a specific option for a full-text search dictionary. Each type of dictionaries has a template containing their custom parameters. Parameters function in a way irrelevant to their setting sequence.	For details, see Table 13-129 .
value	Specifies a parameter value. If the value is not an identifier or a number, enclose it with single quotation marks ('). You can also enclose identifiers and numbers with single quotation marks.	-

Table 13-129 Option parameters of Simple, Ispell, Synonym, Thesaurus, and Snowball

option	Name of the option	Description
Parameters for a Simple dictionary	STOPWORD S	Specifies the name of a file listing stop words. The default file name extension is .stop. For example, if the value of STOPWORDS is french , the actual file name is french.stop . In the file, each line defines a stop word. Dictionaries will ignore blank lines and spaces in the file and convert stop-word phrases into lowercase.

option	Name of the option	Description
	ACCEPT	Specifies whether to accept a non-stop word as recognized. The default value is true. If ACCEPT=true is set for a Simple dictionary, no token will be passed to subsequent dictionaries. In this case, you are advised to place the Simple dictionary at the end of the dictionary list. If ACCEPT=false is set, you are advised to place the Simple dictionary before at least one dictionary in the list.

option	Name of the option	Description
	FILEPATH	Specifies the directory for storing the stop word file. The stop word file can be stored locally or on the OBS server. If the file is stored locally, the directory format is 'file://absolute_path'. If the file is stored on the OBS server, the directory format is 'obs://bucket/path accesskey=ak secretkey=sk region=region_name'. The directory must be enclosed in single quotation marks (''). The default value is the directory where predefined dictionary files are located. Both the FILEPATH and STOPWORDS parameters need to be specified. To create a dictionary using the stop word file on the OBS server, perform the following steps: 1. Upload the stop word file to the OBS server. For example, upload the french.stop file to the gaussdb bucket on the OBS server obsv3.sa-fb-1.externaldemo.com. The URL is https://gaussdb.obsv3.sa-fb-1.externaldemo.com/french.stop. For details about how to upload the file and query the URL, see <i>Object Storage Service (OBS) User Guide</i>. 2. Add "<i>region_name</i>": "<i>obs domain</i>" to the \$GAUSSHOME/etc/region_map file. <i>region_name</i> can be a string consisting of uppercase letters, lowercase letters, digits, slashes (/), or underscores (_). <i>obs domain</i> indicates the domain name of the OBS server. For example, if <i>region_name</i> is set to rg, region_map is as follows: "rg": "obsv3.sa-fb-1.externaldemo.com". NOTICE <i>region_name</i> and <i>obs domain</i> are enclosed in double quotation marks. There is no space on the left of the colon and one space on the right of the colon. 3. Run the CREATE TEXT SEARCH DICTIONARY command to create a dictionary. The command is as follows: <pre>CREATE TEXT SEARCH DICTIONARY french_dict (TEMPLATE = pg_catalog.simple, STOPWORDS = french, FILEPATH = 'obs://gaussdb accesskey=xxx secretkey=yyy region=rg');</pre> The french.stop file is stored in the root directory of the gaussdb bucket. Therefore, the <i>path</i> is empty.

option	Name of the option	Description
Parameters for a Synonym dictionary	SYNONYM	Specifies the name of the definition file for a Synonym dictionary. The default file name extension is .syn. The file is a list of synonyms. Each line is in the format of <i>token synonym</i> , that is, token and its synonym separated by a space.
	CASESENSITIVE	Specifies whether tokens and their synonyms are case sensitive. The default value is false , indicating that tokens and synonyms in dictionary files will be converted into lowercase. If this parameter is set to true , they will not be converted into lowercase.
	FILEPATH	Specifies the directory for storing Synonym dictionary files. The directory can be a local directory or an OBS directory. The default value is the directory where predefined dictionary files are located. The directory format and the process of creating a Synonym dictionary using a file on the OBS server are the same as those of the Simple dictionary.
Parameters for a Thesaurus dictionary	DICTFILE	Specifies the name of a dictionary definition file. The default file name extension is .ths. The file is a list of synonyms. Each line is in the format of <i>sample words : indexed words</i> . The colon (:) is used as a separator between a phrase and its substitute word. If multiple sample words are matched, the TZ selects the longest one.

option	Name of the option	Description
	DICTIONARY	Specifies the name of a subdictionary used for word normalization. This parameter is mandatory and only one subdictionary name can be specified. The specified subdictionary must exist. It is used to identify and normalize input text before phrase matching. If an input word cannot be recognized by the subdictionary, an error will be reported. In this case, remove the word or update the subdictionary to make the word recognizable. In addition, an asterisk (*) can be placed at the beginning of an indexed word to skip the application of a subdictionary on it, but all sample words must be recognizable by the subdictionary. If the sample words defined in the dictionary file contain stop words defined in the subdictionary, use question marks (?) to replace them. Assume that a and the are stop words defined in the subdictionary. ? one ? two : swsw a one the two and the one a two will be matched and output as swsw.
	FILEPATH	Specifies the directory for storing dictionary definition files. The directory can be a local directory or an OBS directory. The default value is the directory where predefined dictionary files are located. The directory format and the process of creating a Synonym dictionary using a file on the OBS server are the same as those of the Simple dictionary.
Parameters for an Ispell dictionary	DICTFILE	Specifies the name of a dictionary definition file. The default file name extension is .dict.
	AFFFILE	Specifies the name of an affix file. The default file name extension is .affix.
	STOPWORDS	Specifies the name of a file listing stop words. The default file name extension is .stop. The file content format is the same as that of the file for a Simple dictionary.

option	Name of the option	Description
	FILEPATH	Specifies the directory for storing dictionary files. The directory can be a local directory or an OBS directory. The default value is the directory where predefined dictionary files are located. The directory format and the process of creating a Synonym dictionary using a file on the OBS server are the same as those of the Simple dictionary.
Parameters for a Snowball dictionary	LANGUAGE	Specifies the name of a language whose stemming algorithm will be used. According to spelling rules in the language, the algorithm normalizes the variants of an input word into a basic word or a stem.
	STOPWORD S	Specifies the name of a file listing stop words. The default file name extension is .stop. The file content format is the same as that of the file for a Simple dictionary.
	FILEPATH	Specifies the directory for storing dictionary definition files. The directory can be a local directory or an OBS directory. The default value is the directory where predefined dictionary files are located. Both the FILEPATH and STOPWORDS parameters need to be specified. The directory format and the process of creating a Snowball dictionary using a file on the OBS server are the same as those of the Simple dictionary.

Examples

Create an **Ispell** dictionary **english_ispell** (the dictionary definition file is from the open source dictionary).

```
CREATE TEXT SEARCH DICTIONARY english_ispell (  
  TEMPLATE = ispell,  
  DictFile = english,  
  AffFile = english,  
  StopWords = english,  
  FilePath = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

See examples in [Configuration Examples](#).

Helpful Links

[ALTER TEXT SEARCH DICTIONARY](#), [CREATE TEXT SEARCH DICTIONARY](#)

13.65 CREATE TRIGGER

Function

Creates a trigger. The trigger will be associated with a specified table or view, and will execute a specified function when certain events occur.

Precautions

- Currently, triggers can be created only on ordinary row-store tables, instead of on column-store tables, temporary tables, or unlogged tables.
- If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- A trigger works only on one table. There is no limit on the number of triggers that can be created. However, more triggers on a table consume more performance.
- Triggers are usually used for data association and synchronization between multiple tables. SQL execution performance is greatly affected. Therefore, you are advised not to use this statement when a large amount of data needs to be synchronized and performance requirements are high.
- When a trigger meets the following conditions, the trigger statement and trigger itself can be pushed together down to a DN for execution, improving the trigger execution performance:
 - **enable_trigger_shipping** and **enable_fast_query_shipping** are both enabled. (This is the default configuration.)
 - The trigger function used by the source table is a PL/pgSQL function (recommended).
 - The source and target tables have the same type and number of distribution keys, are both row-store tables, and belong to the same Node Group.
 - The **INSERT**, **UPDATE**, or **DELETE** statement on the source table contains an expression about equality comparison between all the distribution keys and the *NEW* or *OLD* variable.
 - The **INSERT**, **UPDATE**, or **DELETE** statement on the source table can be pushed down without a trigger.
 - There are only six types of triggers, specified by **INSERT/UPDATE/DELETE**, **AFTER/BEFORE**, and **FOR EACH ROW**, on the source table, and all the triggers can be pushed down.
- The following details trigger types:
 - **INSTEAD OF** triggers must be marked as **FOR EACH ROW** and can be defined only on views.
 - **BEFORE** and **AFTER** triggers on a view must be marked as **FOR EACH STATEMENT**.
 - **TRUNCATE** triggers must be marked as **FOR EACH STATEMENT**.

Syntax

```
CREATE [ CONSTRAINT ] TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }  
ON table_name  
[ FROM referenced_table_name ]  
{ NOT DEFERRABLE | [ DEFERRABLE ] } { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }  
[ FOR [ EACH ] { ROW | STATEMENT } ]  
[ WHEN ( condition ) ]  
EXECUTE PROCEDURE function_name ( arguments );
```

Events include:

```
INSERT  
UPDATE [ OF column_name [, ... ] ]  
DELETE  
TRUNCATE
```

Parameter Description

Table 13-130 CREATE TRIGGER parameters

Parameter	Description	Value Range
CONSTRAINT	Creates a constraint trigger, that is, a trigger is used as a constraint. Such a trigger is similar to a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS . Constraint triggers must be AFTER ROW triggers.	-
trigger_name	Specifies the name of the trigger to be created. The name cannot be schema-qualified because the trigger inherits the schema of its table. In addition, triggers on the same table cannot be named the same. For a constraint trigger, this is also the name to use when you modify the trigger's behavior using SET CONSTRAINTS .	A string, which must comply with the naming convention .
BEFORE	Specifies that a trigger function is called before the trigger event.	-
AFTER	Specifies that a trigger function is called after the trigger event. A constraint trigger can only be specified as AFTER .	-
INSTEAD OF	Specifies that a trigger function directly replaces the trigger event.	-

Parameter	Description	Value Range
event	Event for starting a trigger.	Values are INSERT , UPDATE , DELETE , and TRUNCATE . You can also specify multiple trigger events through OR . For UPDATE events, use the following syntax to specify a list of columns: UPDATE OF column_name1 [, column_name2 ...] The trigger will only fire if at least one of the listed columns is mentioned as a target of the UPDATE statement. INSTEAD OF UPDATE events do not support lists of columns.
table_name	Specifies the name of the table where a trigger needs to be created.	Name of an existing table in the database.
referenced_table_name	Specifies the name of another table referenced by a constraint. This parameter can be specified only for constraint triggers. It does not support foreign key constraints and is not recommended for general use.	Name of an existing table in the database.
DEFERRABLE NOT DEFERRABLE	Specifies the start time of the trigger. It can only be specified for constraint triggers. Controls whether the constraint can be deferred. For details, see CREATE TABLE .	-
INITIALLY IMMEDIATE INITIALLY DEFERRED	If a constraint is deferrable, the two clauses specify the default time to check the constraint, and can be specified only for constraint triggers. For details, see CREATE TABLE .	-

Parameter	Description	Value Range
FOR EACH ROW FOR EACH STATEMENT	Specifies the frequency of firing a trigger. • FOR EACH ROW indicates that the trigger should be fired once for every row affected by the trigger event. • FOR EACH STATEMENT indicates that the trigger should be fired just once per SQL statement. If this parameter is not specified, the default value FOR EACH STATEMENT will be used. Constraint triggers can only be specified as FOR EACH ROW.	-
condition	Specifies a Boolean expression that determines whether a trigger function will actually be executed. If WHEN is specified, the function will be called only when condition returns true. • In FOR EACH ROW triggers, the WHEN condition can reference the columns of old or new row values by writing OLD.column_name or NEW.column_name, respectively. Note that INSERT triggers cannot reference OLD and DELETE triggers cannot reference NEW. • INSTEAD OF triggers do not support WHEN conditions. • WHEN expressions cannot contain subqueries. • For constraint triggers, evaluation of the WHEN condition is not deferred, but occurs immediately after the update operation is performed. If the condition does not return true, the trigger will not be queued for deferred execution.	-
function_name	Specifies a user-defined function, which must be declared as taking no parameters and returning data of the trigger type. This function is executed when a trigger fires.	-

Parameter	Description	Value Range
arguments	Specifies an optional, comma-separated list of parameters to be provided to a function when a trigger is executed. Parameters are literal string constants. Simple names and numeric constants can also be included, but they will all be converted to strings. Check descriptions of the implementation language of a trigger function to find out how these parameters are accessed within the function.	-

Table 13-131 Types of triggers supported on tables and views

Trigger Timing	Trigger Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	Not supported	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	Not supported	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	Not supported
	TRUNCATE	Not supported	Not supported

Table 13-132 Special variables in the functions PL/pgSQL triggers

Variable	Description
NEW	New tuple for INSERT/UPDATE operations. This variable is NULL for DELETE operations.
OLD	Old tuple for UPDATE/DELETE operations. This variable is NULL for INSERT operations.
TG_NAME	Trigger name
TG_WHEN	Trigger timing (BEFORE/AFTER/INSTEAD OF)

Variable	Description
TG_LEVEL	Trigger frequency (ROW/STATEMENT)
TG_OP	Trigger event (INSERT/UPDATE/DELETE/TRUNCATE)
TG_RELID	OID of the table where a trigger is located
TG_RELNAME	Name of the table where a trigger is located. (This variable is now discarded and is replaced by TG_TABLE_NAME .)
TG_TABLE_NAME	Name of the table where a trigger is located.
TG_TABLE_SCHEMA	Schema information of the table where a trigger is located
TG_NARGS	Number of parameters for a trigger function
TG_ARGV[]	List of parameters for a trigger function

Examples

Create a source table and a trigger table:

```
CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);  
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
```

Create the trigger function **tri_insert_func()**:

```
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS  
$$  
DECLARE  
BEGIN  
    INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);  
    RETURN NEW;  
END  
$$ LANGUAGE PLPGSQL;
```

Create the trigger function **tri_update_func()**:

```
CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS  
$$  
DECLARE  
BEGIN  
    UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE id1=OLD.id1;  
    RETURN OLD;  
END  
$$ LANGUAGE PLPGSQL;
```

Create the trigger function **tri_delete_func()**:

```
CREATE OR REPLACE FUNCTION tri_delete_func() RETURNS TRIGGER AS  
$$  
DECLARE  
BEGIN
```



```
DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;  
RETURN OLD;  
END  
$$ LANGUAGE PLPGSQL;
```

Create an **INSERT** trigger:

```
CREATE TRIGGER insert_trigger  
BEFORE INSERT ON test_trigger_src_tbl  
FOR EACH ROW  
EXECUTE PROCEDURE tri_insert_func();
```

Create an **UPDATE** trigger:

```
CREATE TRIGGER update_trigger  
AFTER UPDATE ON test_trigger_src_tbl  
FOR EACH ROW  
EXECUTE PROCEDURE tri_update_func();
```

Create a **DELETE** trigger:

```
CREATE TRIGGER delete_trigger  
BEFORE DELETE ON test_trigger_src_tbl  
FOR EACH ROW  
EXECUTE PROCEDURE tri_delete_func();
```

Helpful Links

[ALTER TRIGGER](#), [DROP TRIGGER](#), [ALTER TABLE](#)

13.66 CREATE TYPE

Function

CREATE TYPE defines new data types in the current database. The user who defines a new data type becomes its owner. Types are designed only for row-store tables.

Four types of data can be created by using **CREATE TYPE**: composite data, base data, a shell data, and enumerated data.

- Composite types

A composite type is specified by a list of attribute names and data types. If the data type of an attribute is collatable, the attribute's collation rule can also be specified. A composite type is essentially the same as the row type of a table. However, using **CREATE TYPE** avoids the need to create an actual table when only a type needs to be defined. In addition, a standalone composite type is useful, for example, as the parameter or return type of a function.

To create a composite type, you must have the **USAGE** permission for all its attribute types.

- Base types

You can customize a new base type (scalar type). Generally, functions required for base types must be coded in C or another low-level language.

- Shell types

With this parameter, you can create a shell type, which is a type name without a definition. To create a shell type, use **CREATE TYPE** with only the

type name. Shell types are needed as forward references when base types are created.

- Enumerated types

An enumerated type is a list of enumerated values. Each value is a non-empty string with the maximum length of 64 bytes.

Precautions

- If a schema name is given, the type will be created in the specified schema. Otherwise, it will be created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, the type name must also be distinct from the name of any existing table in the same schema.)
- When creating a base type, you can place parameters in any order. The **input_function** and **output_function** parameters are mandatory, and other parameters are optional.
- The **category** and **preferred** parameters can be used to help determine which implicit cast excels in ambiguous situations. Each data type belongs to a category named by a single ASCII character, and each type is either preferred or not within its category. If this rule is helpful in resolving overloaded functions or operators, the parser will prefer casting to preferred types (but only from other types within the same category). For types that have no implicit casts to or from any other types, it is sufficient to leave these parameters at their default values. However, for a group of types that have implicit casts, mark them all as belonging to a category and select one or two of the most general types as being preferred within the category. The **category** parameter is helpful in adding a user-defined type to an existing built-in category, such as the numeric or string type. However, you can also create new entirely-user-defined type categories. Select any ASCII character other than an uppercase letter to name such a category.
- **type_modifier_input_function** and **type_modifier_output_function** are needed if a type supports modifiers, that is, optional constraints attached to a type declaration, such as `char(5)` or `numeric(30,2)`. DWS allows user-defined types to take one or more simple constants or identifiers as modifiers. However, this information must be capable of being packed into a single non-negative integer value for storage in system catalogs. Declared modifiers are passed to **type_modifier_input_function** in the cstring array format. The parameter must check values for validity, throwing an error if they are wrong. If they are correct, the parameter will return a single non-negative integer value, which will be stored as `typmod` in a column. If the type does not have **type_modifier_input_function**, type modifiers will be rejected. **type_modifier_output_function** converts the internal integer `typmod` value back to a correct format for user display. It must return a cstring value, which is the exact string appending to the type name. For example, a numeric function may return `(30,2)`. If the default display format is enclosing a stored `typmod` integer value in parentheses, you can omit **type_modifier_output_function**.
- Input and output functions can be declared to have the results or parameters of a new type because they have to be created before the new type is created. The new type should first be defined as a shell type, which is a placeholder type that has no attributes except a name and an owner. This can be done by delivering the **CREATE TYPE** *name* statement, with no additional parameters.

Then, the C I/O functions can be defined as referencing the shell type. Finally, **CREATE TYPE** with a full definition replaces the shell type with a complete, valid type definition. After that, the new type can be used normally.

Syntax

```
CREATE TYPE name AS
  ( [ attribute_name data_type [ COLLATE collation ] [ , ... ] ] )

CREATE TYPE name (
  INPUT = input_function,
  OUTPUT = output_function
  [ , RECEIVE = receive_function ]
  [ , SEND = send_function ]
  [ , TYPMOD_IN = type_modifier_input_function ]
  [ , TYPMOD_OUT = type_modifier_output_function ]
  [ , ANALYZE = analyze_function ]
  [ , INTERNALLENGTH = { internallength | VARIABLE } ]
  [ , PASSEDBYVALUE ]
  [ , ALIGNMENT = alignment ]
  [ , STORAGE = storage ]
  [ , LIKE = like_type ]
  [ , CATEGORY = category ]
  [ , PREFERRED = preferred ]
  [ , DEFAULT = default ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
  [ , COLLATABLE = collatable ]
)

CREATE TYPE name

CREATE TYPE name AS ENUM
  ( [ 'label' [ , ... ] ] )
```

Parameter Description

Table 13-133 CREATE TYPE parameters

Parameter	Description	Value Range
name	Specifies the name (optionally schema-qualified) of the type to be created.	It must comply with the naming convention .
attribute_name	Specifies the name of an attribute (column) for the composite type.	-
data_type	Specifies the name of an existing data type to become a column of the composite type.	-

Parameter	Description	Value Range
collation	Specifies the name of an existing collation associated with a column of the composite type.	-
input_function	Specifies the name of a function that converts data from the external text format of a type to its internal format. An input function can be declared as taking one parameter of the cstring type or taking three parameters of the cstring, oid, and integer types. • The cstring-type parameter is the input text as a C string. • The oid-type parameter is the OID of the type (except for array types, where the parameter is the element type OID of an array type). • The integer-type parameter is typmod of the destination column, if known (-1 will be passed if not known). An input function must return a value of the data type itself. Generally, an input function must be declared as STRICT. If it is not, it will be called with a NULL parameter coming first when the system reads a NULL input value. In this case, the function must still return NULL unless an error raises. (This mechanism is designed for supporting domain input functions, which may need to reject NULL input values.)	-
output_function	Specifies the name of a function that converts data from the internal format of a type to its external text format. An output function must be declared as taking one parameter of a new data type. It must return data of the cstring type. Output functions are not invoked for NULL values.	-

Parameter	Description	Value Range
receive_function (optional)	Specifies the name of a function that converts data from the external binary format of a type to its internal format. If this function is not used, the type cannot participate in binary input. It costs lower to convert the binary format to the internal format, more portable. (For example, the standard integer data types use the network byte order as an external binary representation, whereas the internal representation is in the machine's native byte order.) This function should perform adequate checks to ensure a valid value. Also, this function can be declared as taking one parameter of the internal type or taking three parameters of the internal, oid, and integer types. • The internal-type parameter is a pointer to a StringInfo buffer holding received byte strings. • The oid- and integer-type parameters are the same as those of the text input function. A receive function must return a value of the data type itself. Generally, a receive function must be declared as STRICT. If it is not, it will be called with a NULL parameter coming first when the system reads a NULL input value. In this case, the function must still return NULL unless an error raises. (This mechanism is designed for supporting domain receive functions, which may need to reject NULL input values.)	-
send_function (optional)	Specifies the name of a function that converts data from the internal format of a type to its external binary format. If this function is not used, the type cannot participate in binary output. A send function must be declared as taking one parameter of a new data type. It must return data of the bytea type. Send functions are not invoked for NULL values.	-
type_modifier_input_function (optional)	Specifies the name of a function that converts an array of modifiers for a type to its internal format.	-

Parameter	Description	Value Range
type_modifier_output_function (optional)	Specifies the name of a function that converts the internal format of modifiers for a type to its external text format.	-
analyze_function (optional)	Specifies the name of a function that performs statistical analysis for a data type. By default, if there is a default B-tree operator class for a type, ANALYZE will attempt to gather statistics by using the "equals" and "less-than" operators of the type. This behavior is inappropriate for non-scalar types, and can be overridden by specifying a custom analysis function. The analysis function must be declared to take one parameter of the internal type and return a boolean result.	-
internallength (optional)	Specifies a numeric constant for specifying the length in bytes of the internal representation of a new type. By default, it is variable-length. Although the details of the new type's internal representation are only known to I/O functions and other functions that you create to work with the type, there are still some attributes of the internal representation that must be declared to DWS. The most important one is internallength . Base data types can be fixed-length (when internallength is a positive integer) or variable-length (when internallength is set to VARIABLE ; internally, this is represented by setting typlen to -1). The internal representation of all variable-length types must start with a 4-byte integer. internallength defines the total length.	-
PASSEDBYVALUE (optional)	Specifies that values of a data type are passed by value, rather than by reference. Types passed by value must be fixed-length, and their internal representation cannot be larger than the size of the Datum type (4 bytes on some machines, and 8 bytes on others).	-
alignment (optional)	Specifies the storage alignment required for a data type. It supports values char , int2 , int4 , and double . The default value is int4 . The allowed values equate to alignment on 1-, 2-, 4-, or 8-byte boundaries. Note that variable-length types must have an alignment of at least 4 since they must contain an int4 value as their first component.	-

Parameter	Description	Value Range
storage (optional)	Specifies the storage strategy for a data type. It supports values plain, external, extended, and main. The default value is plain. • plain specifies that data of a type will always be stored in-line and not compressed. (Only plain is allowed for fixed-length types.) • extended specifies that the system will first try to compress a long data value and will then move the value out of the main table row if it is still too long. • external allows a value to be moved out of the main table, but the system will not try to compress it. • main allows for compression, but discourages moving a value out of the main table. (Data items with this storage strategy might still be moved out of the main table if there is no other way to make a row fit. However, they will be kept in the main table preferentially over extended and external items.) All storage values except plain imply that the functions of the data type can handle values that have been toasted. A specified value determines the default TOAST storage strategy for columns of data types that support TOAST. Users can select different strategies for individual columns using ALTER TABLE SET STORAGE.	-
like_type (optional)	Specifies the name of an existing data type that has the same representation as a new type. The values of internallength, passedbyvalue, alignment, and storage are copied from this type, unless they are overridden by explicit specifications elsewhere in the CREATE TYPE command. Specifying representation in this way is especially useful when the low-level implementation of a new type references an existing type.	-

Parameter	Description	Value Range
category (optional)	Specifies the category code (a single ASCII character) for a type. The default value is U, indicating the user-defined type. You can also choose other ASCII characters to create custom categories.	-
preferred (optional)	Specifies whether a type is preferred within its type category. If it is, the value will be TRUE , else FALSE . The default value is FALSE . Be cautious when creating a new preferred type within an existing type category because this could cause great changes in behavior.	-
default (optional)	Specifies the default value for a data type. If this parameter is omitted, the default value will be NULL. A default value can be specified if you expect the columns of a data type to default to something other than the NULL value. You can also specify a default value using the DEFAULT keyword. (Such a default value can be overridden by an explicit DEFAULT clause attached to a particular column.)	-
element (optional)	Specifies the type of an array element when an array type is created. For example, to define an array of 4-byte integers (int4), set ELEMENT to int4 .	-
delimiter (optional)	Specifies the delimiter character to be used between values in arrays made of a type. delimiter can be set to a specific character. The default delimiter is a comma (.). Note that a delimiter is associated with the array element type, instead of the array type itself.	-
collatable (optional)	Specifies whether a type's operations can use collation information. If they can, the value will be TRUE, else FALSE (default). If collatable is TRUE, column definitions and expressions of a type may carry collation information by using the COLLATE clause. It is the implementations of functions operating on the type that actually use the collation information. This use cannot be achieved merely by marking the type collatable.	-

Parameter	Description	Value Range
label (optional)	Specifies a text label associated with an enumerated value. It is a non-empty string of up to 64 characters.	-

Examples

Example 1: Create a composite type, create a table, insert data, and make a query.

```
CREATE TYPE compfoo AS (f1 int, f2 text);
CREATE TABLE t1_compfoo(a int, b compfoo);
CREATE TABLE t2_compfoo(a int, b compfoo);
INSERT INTO t1_compfoo values(1,(1,'demo'));
INSERT INTO t2_compfoo select * from t1_compfoo;
SELECT (b).f1 FROM t1_compfoo;
SELECT * FROM t1_compfoo t1 join t2_compfoo t2 on (t1.b).f1=(t1.b).f1;
```

Example 2: Create an enumeration type and use it in the table definition.

```
CREATE TYPE bugstatus AS ENUM ('create', 'modify', 'closed');
CREATE TABLE customer (name text,current_bugstatus bugstatus);
INSERT INTO customer VALUES ('type','create');
SELECT * FROM customer WHERE current_bugstatus = 'create';
```

Example 3: Compile a .so file and create the shell type.

```
CREATE TYPE complex;
```

This statement creates a placeholder for the type to be created, which can then be referenced when defining its I/O functions. Now you can define an I/O function. Note that the function must be declared in NOT FENCED mode when it is created.

```
CREATE FUNCTION
complex_in(cstring)
  RETURNS complex
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_out(complex)
  RETURNS cstring
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_rcv(internal)
  RETURNS complex
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_send(complex)
  RETURNS bytea
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;
```

Provide a complete definition of the data type.

```
CREATE TYPE complex (
internallength = 16,
```

```
input = complex_in,  
output = complex_out,  
receive = complex_rcv,  
send = complex_send,  
alignment = double  
);
```

The C functions corresponding to the input, output, receive, and send functions are defined as follows:

```
-- Define a structure body Complex:  
typedef struct Complex {  
    double    x;  
    double    y;  
} Complex;  
  
-- Define an input function:  
PG_FUNCTION_INFO_V1(complex_in);  
  
Datum  
complex_in(PG_FUNCTION_ARGS)  
{  
    char      *str = PG_GETARG_CSTRING(0);  
    double    x,  
             y;  
    Complex   *result;  
  
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)  
        ereport(ERROR,  
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),  
                 errmsg("invalid input syntax for complex: \"%s\"",  
                        str)));  
  
    result = (Complex *) palloc(sizeof(Complex));  
    result->x = x;  
    result->y = y;  
    PG_RETURN_POINTER(result);  
}  
  
-- Define an output function:  
PG_FUNCTION_INFO_V1(complex_out);  
  
Datum  
complex_out(PG_FUNCTION_ARGS)  
{  
    Complex   *complex = (Complex *) PG_GETARG_POINTER(0);  
    char      *result;  
  
    result = (char *) palloc(100);  
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);  
    PG_RETURN_CSTRING(result);  
}  
  
-- Define a receive function:  
PG_FUNCTION_INFO_V1(complex_rcv);  
  
Datum  
complex_rcv(PG_FUNCTION_ARGS)  
{  
    StringInfo buf = (StringInfo) PG_GETARG_POINTER(0);  
    Complex   *result;  
  
    result = (Complex *) palloc(sizeof(Complex));  
    result->x = pq_getmsgfloat8(buf);  
    result->y = pq_getmsgfloat8(buf);  
    PG_RETURN_POINTER(result);  
}  
  
-- Define a send function:  
PG_FUNCTION_INFO_V1(complex_send);
```

```
Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Helpful Links

[ALTER TYPE](#), [DROP TYPE](#)

13.67 CREATE USER

Function

Create a database user.

Precautions

- A user created using the **CREATE USER** statement has the **LOGIN** permission by default.
- A schema named after the user is automatically created in the database where the statement is executed, but not in other databases. You can run the **CREATE SCHEMA** statement to create such a schema for the user in other databases.
- The owner of an object created by a system administrator in a schema with the same name as a common user is the common user, not a system administrator.
- Users who are not system administrators and are granted with all schema permissions cannot create objects in a schema named after a user, unless the users are granted with the role permissions of that schema. For details, see [After the all Permission Is Granted to the Schema of a User, the Error Message "ERROR: current user does not have privilege to role tom" Persists During Table Creation](#).

CAUTION

- Avoid having all business operations run under a single database user. Instead, plan different database users according to the business modules.
 - For better access control of different business modules, it is better to use multiple users and permissions instead of depending on the system administrator user to run business operations.
 - For more information about development and design proposal, see [Development and Design Proposal](#).
-

Syntax

```
CREATE USER user_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The **option** clause is used for setting information including permissions and attributes.

```
{SYSADMIN | NOSYSADMIN}  
| {AUDITADMIN | NOAUDITADMIN}  
| {CREATEDB | NOCREATEDB}  
| {USEFT | NOUSEFT}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCADMIN}  
| CONNECTION LIMIT connlimit  
| VALID BEGIN 'timestamp'  
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| TEMP SPACE 'tmpspacelimit'  
| SPILL SPACE 'spillspacelimit'  
| NODE GROUP logic_cluster_name  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN role_name [, ...]  
| USER role_name [, ...]  
| SYSID uid  
| DEFAULT TABLESPACE tablespace_name  
| PROFILE DEFAULT  
| PROFILE profile_name  
| PGUSER  
| AUTHINFO 'authinfo'  
| PASSWORD EXPIRATION period
```

Parameter Description

Table 13-134 CREATE USER parameters

Parameter	Description	Value Range/ Default Value
user_name	Name of the user to be created.	A string. It must comply with the naming conventions and can contain a maximum of 63 characters. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range/ Default Value
password	Specifies the login password. A password must: • Contain at least eight characters. This is the default length. • Differ from the user name or the user name spelled backwards. • Contains at least three of the following four character types: uppercase letters, lowercase letters, digits, and special characters, including: ~!@#\$%^&*()-_+=\ {}];;<.>/. If you use characters other than the four types, a warning is displayed, but you can still create the password.	A string
DISABLE	By default, you can change your password unless it is disabled. Use this parameter to disable the password of a user. After the password of a user is disabled, the password will be deleted from the system. The user can connect to the database only through external authentication, for example, IAM authentication, Kerberos authentication, or LDAP authentication. Only administrators can enable or disable a password. Common users cannot disable the password of an initial user. To enable a password, run ALTER USER and specify the password.	-
ENCRYPTED UNENCRYPTED	Determines whether the password stored in the system will be encrypted. (If neither is specified, the password status is determined by password_encryption_type.) According to product security requirements, the password must be stored encrypted. Therefore, UNENCRYPTED is forbidden in DWS. If the password is encrypted using SHA256, it will be stored as it is, regardless of whether it is specified as ENCRYPTED or UNENCRYPTED. This is because the system cannot decrypt the specified encrypted password. This allows reloading of the encrypted password during dump/restore.	-

Parameter	Description	Value Range/ Default Value
SYSADMIN NOSYSADMIN	Determines whether a new user is a system administrator. A user with the SYSADMIN attribute has the highest permission in the system.	Default value: NOSYSADMIN
AUDITADMIN NOAUDITADMIN	Defines whether a user has the audit administrator attribute.	Default value: NOAUDITADMIN
CREATEDB NOCREATEDB	Controls whether a new user can create a database. A new user does not have the permission to create a database by default.	Default value: NOCREATEDB
USEFT NOUSEFT	Determines whether a new role can perform operations on foreign tables, such as creating, deleting, modifying, and reading/writing foreign tables. The default value is NOUSEFT , indicating that new users cannot perform operations on foreign tables by default. The new user does not have the permission to perform operations on foreign tables.	-
CREATEROLE NOCREATEROLE	Controls whether a user can create a role or user (that is, execute CREATE ROLE and CREATE USER). A user with the CREATEROLE permission can also modify and delete other users or roles.	Value range: If not specified, NOCREATEROLE is the default.
INHERIT NOINHERIT	Controls whether a user can inherit the permissions of its group. You are not advised to use them.	-
LOGIN NOLOGIN	Controls whether a user can log in to the database. Only users with the LOGIN attribute can log in to the database.	Default value: LOGIN
REPLICATION NOREPLICATION	Controls whether a user is allowed to initiate streaming replication or put the system in and out of backup mode. A user with the REPLICATION attribute is only used for replication.	Default value: NOREPLICATION

Parameter	Description	Value Range/ Default Value
INDEPENDENT NOINDEPENDENT	Defines private and independent users. For a user with the INDEPENDENT attribute, administrators' rights to control and access this role are separated. Specific rules are as follows: - Administrators have no rights to add, delete, query, modify, copy, or authorize the corresponding table objects without the authorization from the INDEPENDENT user. - Without the authorization of the INDEPENDENT user, the administrator has no right to modify its inheritance relationship. - The administrator does not have the permission to change the owner of the table object of an INDEPENDENT user. - The administrator does not have the permission to remove the INDEPENDENT attribute of an INDEPENDENT user. - The administrator does not have the permission to change the database password of an INDEPENDENT user. An INDEPENDENT must manage its own password. If the password is lost, it cannot be reset. - The SYSADMIN attribute of a user cannot be changed to the INDEPENDENT attribute.	-
VCADMIN NOVCADMIN	Defines a logical cluster administrator. A logical cluster administrator has the following more permissions than common users: - Create, modify, and delete resource pools in the associated logical cluster. - Grant the access permission for the associated logical cluster to other users or roles, or reclaim the access permission from those users or roles.	-

Parameter	Description	Value Range/ Default Value
CONNECTION LIMIT	Specifies the number of concurrent connections that can be used by a user on a single CN. To ensure that a cluster runs properly, the minimum value of connlimit is the number of CNs in the cluster, because when a cluster runs ANALYZE on a CN, other CNs will connect to the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set CONNECTION LIMIT to 3 or a larger value.	Integers no less than -1. The default value -1 means no limit.
VALID BEGIN	Sets the timestamp when a user takes effect. If this clause is omitted, there is no restriction on when the user takes effect.	-
VALID UNTIL	Sets an expiration timestamp for an account. After the specified time, the account cannot be used. If this clause is omitted, there is no restriction on when the user expires.	-
RESOURCE POOL	Sets the name of resource pool used by a user, and the name belongs to the system catalog: pg_resource_pool .	-
USER GROUP 'groupuser'	Create a sub-user.	-
PERM SPACE	Sets the storage space of the user permanent table. space_limit: specifies the upper limit of the storage space of the permanent table.	space_limit: A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.
TEMP SPACE	Sets the storage space of the user temporary table. tmpspacelimit: specifies the storage space limit of the temporary table.	tmpspacelimit: A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.

Parameter	Description	Value Range/ Default Value
SPILL SPACE	Sets the operator disk flushing space of the user. spillspacelimit : specifies the operator spilling space limit.	spillspacelimit : A string consists of an integer and unit. The unit can be K/M/G/T/P currently. 0 indicates no limits.
NODE GROUP	Specifies the name of the logical cluster associated with a user. If the name contains uppercase characters or special characters, enclose the name with double quotation marks.	-
IN ROLE	The new user immediately has the permissions of users listed in the IN ROLE clause. You are not advised to use them.	-
IN GROUP	Indicates an obsolete spelling of IN ROLE . You are not advised to execute them.	-
ROLE	The ROLE clause lists one or more existing users. They are automatically added as members of the new user and have all the permissions of the new user.	-
ADMIN	The ADMIN clause is similar to the ROLE clause. The difference is that the user after ADMIN can grant the permissions of the new user to other users.	-
USER	Indicates an obsolete spelling of the ROLE clause.	-
SYSID	The SYSID clause is ignored.	-
DEFAULT TABLESPACE	The DEFAULT TABLESPACE clause is ignored.	-
PROFILE	The PROFILE clause is ignored.	-

Parameter	Description	Value Range/ Default Value
PGUSER	This attribute is used to be compatible with open-source Postgres communication. An open-source Postgres client interface (Postgres 9.2.19 is recommended) can use a database user having this attribute to connect to the database. This attribute only ensures compatibility with the connection process. Incompatibility caused by kernel differences between this product and Postgres cannot be solved using this attribute.	-
AUTHINFO 'authinfo'	This attribute is used to specify the user authentication type. authinfo is the description character string, which is case sensitive. Only the LDAP type is supported. Its description character string is ldap. LDAP authentication is an external authentication mode. Therefore, PASSWORD DISABLE must be specified. • Additional information about LDAP authentication can be added to authinfo, for example, fulluser in LDAP authentication, which is equivalent to ldapprefix+username+ldapsuffix. If authinfo is ldap, the user authentication type is LDAP. In this case, the ldapprefix and ldapsuffix information is provided by the corresponding record in the pg_hba.conf file. • When executing the ALTER ROLE command, users are not allowed to change the authentication type. Only LDAP users are allowed to modify LDAP attributes.	-

Parameter	Description	Value Range/ Default Value
PASSWORD EXPIRATION period	Number of days before the login password of the role expires. The user needs to change the password in time before the login password expires. If the login password expires, the user cannot log in to the system. In this case, the user needs to ask the administrator to set a new login password.	The value is an integer ranging from -1 to 999. • The default value -1 indicates no restrictions. • 0: indicates that the login password expires immediately.

Examples

Create user **new_u1**.

```
CREATE USER new_u1 PASSWORD '{password}';
```

The following statements are equivalent to the above.

```
CREATE USER new_u1 IDENTIFIED BY '{password}';
```

For a user having the **CREATE DATABASE** permission, specify the **CREATEDB** keyword.

```
CREATE USER new_u1 CREATEDB PASSWORD '{password}';
```

Helpful Links

[ALTER USER](#), [CREATE ROLE](#), [DROP USER](#)

FAQs

[How Do I Check the Creation Time of a GaussDB\(DWS\) Database User?](#)

13.68 CREATE VIEW

Function

CREATE VIEW creates a view.

A view is a virtual table, not a base table. A database only stores the definition of a view and does not store its data. The data is still stored in the original base table. If data in the base table changes, the data in the view changes accordingly. In this sense, a view is like a window through which users can know their interested data and data changes in the database.

Precautions

None

Syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW view_name [ ( column_name [ , ... ] ) ]  
[ WITH ( {view_option_name [= view_option_value]} [ , ... ] ) ]  
AS query;
```

NOTE

- You can use **WITH (security_barriers)** to create a relatively secure view. This prevents attackers from printing hidden base table data by using the **RAISE** statement of low-cost functions.
- When the **view_independent** GUC parameter is enabled, columns can be deleted from common views. Note that if a column-level constraint exists, the corresponding column cannot be deleted.
- Duplicate CTE names are not supported when the view decoupling function is enabled.

Example:

```
CREATE TABLE t1(a1 INT, b1 INT);  
CREATE TABLE t2(a2 INT, b2 INT, c2 INT);  
CREATE OR REPLACE VIEW v1 AS WITH tmp AS (SELECT * FROM t2) ,tmp1 AS (SELECT b2,c2  
FROM tmp WHERE b2 = (WITH RECURSIVE tmp(aa, bb) AS (SELECT a1,b1 FROM t1) SELECT bb  
FROM tmp WHERE aa = c2)) SELECT c2 FROM tmp1;
```

Parameter Description

Table 13-135 CREATE VIEW parameters

Parameter	Description	Value Range
OR REPLACE	If a view with the same name already exists, redefine the view.	-
TEMP TEMPORARY	Creates a temporary view.	-
view_name	Specifies the name of a view to be created. It is optionally schema-qualified.	A string, which must comply with the naming convention .
column_name	Specifies an optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.	A string, which must comply with the naming convention .

Parameter	Description	Value Range
WITH ({view_option_name [= view_option_value]} [, ...])	Specifies the optional parameters for a view. Currently, the only parameter supported by view_option_name is security_barrier , which should be enabled when a view is intended to provide row-level security.	If security_barrier is of the boolean type, the options are as follows: • TRUE • FALSE
query	A SELECT or VALUES statement which will provide the columns and rows of the view.	-

Examples

Create a view consisting of columns whose **spcname** is **pg_default**:

```
CREATE VIEW myView AS  
SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';
```

Run the following command to redefine the existing view **myView** and create a view consisting of columns whose **spcname** is **pg_global**:

```
CREATE OR REPLACE VIEW myView AS  
SELECT * FROM pg_tablespace WHERE spcname = 'pg_global';
```

Create a view consisting of rows with **age** smaller than 12:

```
DROP TABLE IF EXISTS customer;  
  
CREATE TABLE customer  
(id int, name varchar(20) , age int)  
with (orientation = column,COMPRESSION=MIDDLE)  
distribute by hash(id);  
  
INSERT INTO customer VALUES (1,'lily',10),(2, 'lucy',12),(3,'lilei',15);  
  
CREATE VIEW customer_details_view_v1 AS  
SELECT * FROM customer  
WHERE age < 12;
```

Updatable Views

After the **enable_view_update** parameter is enabled, the **INSERT**, **UPDATE**, **DELETE**, and **MERGE INTO** statements can be used to update simple views. Only 8.1.2 or later supports updates using the **MERGE INTO** statement.

Views that meet all the following conditions can be updated:

- The FROM clause in the view definition contains only one common table, which cannot be a system table, foreign table, DFS table, delta table, TOAST table, or error table.
- The view contains updatable columns, which are simple references to the updatable columns of the base table.

- The view definition does not contain the WITH, DISTINCT, GROUP BY, ORDER BY, FOR UPDATE, FOR SHARE, HAVING, TABLESAMPLE, LIMIT or OFFSET clause.
- The view definition does not contain the UNION, INTERSECT, or EXCEPT operation.
- The selection list of the view definition does not contain aggregate functions, window functions, or functions that return collections.
- Ensure that there is no trigger whose trigger occasion is **INSTEAD OF** in the view for **INSERT**, **UPDATE**, and **DELETE** statements. For the **MERGE INTO** statement, neither the view nor the underlying table can have triggers.
- The view definition does not contain sublinks.
- The view definition does not contain functions whose attribute is **VOLATILE**. The values of such functions can be changed during a table scan.
- The view definition does not set an alias for the column where the distribution key of the table resides, or name a common column as the distribution key column.
- When the RETURNING clause is used in the view update operation, columns in the view definition only come from the base table.

If the definition of the updatable view contains a WHERE condition, the condition restricts the UPDATE and DELETE statements from modifying rows on the base table. If the WHERE condition is not met after the UPDATE statement is executed, the updated rows cannot be queried in the view. Similarly, If the WHERE condition is not met after the INSERT statement is executed, the inserted data cannot be queried in the view. To insert, update, or delete data in a view, you must have the corresponding permission on the view and tables.

Helpful Links

[ALTER VIEW](#), [DROP VIEW](#)

13.69 CURSOR

Function

This statement is used to create a cursor and retrieve specified rows of data from a query.

To process SQL statements, the stored procedure process assigns a memory segment to store context association. Cursors are handles or pointers to context regions. With cursors, stored procedures can control alterations in context regions.

Precautions

- **CURSOR** is used only in transaction blocks.
- Generally, **CURSOR** and **SELECT** both have text returns. Since data is stored in binary format in the system, the system needs to convert the data from the binary format to the text format. If data is returned in text format, the client-end application needs to convert the data back to a binary format for processing. **FETCH** implements conversion between binary data and text data.

- Use a binary cursor unless necessary, since a text cursor occupies larger storage space than a binary cursor. A binary cursor returns internal binary data, which is easier to operate. To return data in text format, it is advisable to retrieve data in text format, therefore reducing workload at the client end. For example, the value 1 in an integer column of a query is returned as a character string 1 if a default cursor is used, but is returned as a 4-byte binary value (big-endian) if a binary cursor is used.

Syntax

```
CURSOR cursor_name  
[ BINARY ] [ NO SCROLL ] [ { WITH | WITHOUT } HOLD ]  
FOR query ;
```

Parameter Description

Table 13-136 CURSOR parameters

Parameter	Description	Value Range
cursor_name	Specifies the name of the cursor to be created.	A string, which must comply with the naming convention .
BINARY	Specifies that data retrieved by the cursor will be returned in binary format, not in text format.	-
NO SCROLL	Specifies the mode of data retrieval by the cursor. • NO SCROLL: If NO SCROLL is specified, backward fetches will be rejected. • Not stated: The system automatically determines whether the cursor can be used for backward fetches based on the execution plan.	-
WITH HOLD WITHOUT HOLD	Specifies whether the cursor can still be used after the cursor creation event. • WITH HOLD indicates that the cursor can still be used. • WITHOUT HOLD indicates that the cursor cannot be used. (default setting)	-
query	The SELECT or VALUES clause specifies the row to return the cursor value.	Use the SELECT or VALUES clause.

Examples

Start a transaction.

```
START TRANSACTION;
```

Create a cursor named **cursor1**:

```
CURSOR cursor1 FOR SELECT * FROM tpceds.customer_address ORDER BY 1;
```

create a cursor named **cursor2**:

```
CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
```

Usage of the **WITH HOLD** cursor:

1. Set up the **WITH HOLD** cursor.

```
DECLARE cursor3 CURSOR WITH HOLD FOR SELECT * FROM tpceds.customer_address ORDER BY 1;
```
2. Fetch the first two rows from cursor3.

```
FETCH FORWARD 2 FROM cursor3;
```
3. End the transaction.

```
END;
```
4. Fetch the next row from cursor3.

```
FETCH FORWARD 1 FROM cursor3;
```
5. Close the cursor.

```
CLOSE cursor3;
```

Helpful Links

[FETCH](#)

13.70 DISCARD

Function

Releases internal resources related to database sessions.

The **DISCARD** command is used to reset the status of some or all sessions. Different **DISCARD** clauses release different types of resources. The **DISCARD ALL** command releases all temporary resources related to the current session and resets them to the initial state.

Precautions

- After the **DISCARD VOLATILE { TEMPORARY | TEMP }** statement is executed, all volatile temporary table resources in the current session will be cleared. However, the statement cannot clear a single volatile temporary table resource.
- If a global temporary table occupies resources in a session, you need to run the **DISCARD** command to clear the resources of all sessions before performing DDL operations.
- After **DISCARD ALL** is executed successfully, schemas starting with **pg_temp** and **pg_toast_temp** are also deleted.
- **DISCARD ALL** cannot be executed in a transaction.

Syntax

```
DISCARD {{ GLOBAL { TEMPORARY | TEMP } [ TABLE table_name ] } |  
         { VOLATILE { TEMPORARY | TEMP } } |  
         { ALL | TEMP | TEMPORARY | PLANS | SEQUENCES }}
```

Parameter Description

Table 13-137 DISCARD parameters

Parameter	Description
VOLATILE { TEMPORARY TEMP }	Releases resources related to the VOLATILE temporary table in the current session.
GLOBAL { TEMPORARY TEMP } [TABLE table_name]	- Run the DISCARD GLOBAL TEMP command to release resources related to the global temporary table in the current session. DISCARD GLOBAL TEMP TABLE table_name releases resources of a specified global temporary table in the current session.
TEMP TEMPORARY	Releases resources related to all temporary tables in the current session, including volatile and global temporary tables.
PLANS	Releases all cached query plans in the current session and forces them to be replanned when related PREPARE statements are used next time.
SEQUENCES	Discards all cached sequence-related states, including currval()/lastval() information and any pre-allocated sequence values that have not been returned through nextval() .
ALL	Releases all temporary resources related to the current session and resets them to their initial state. This has almost the same effect as executing the following statement sequence: SET SESSION AUTHORIZATION DEFAULT; RESET ALL; DEALLOCATE ALL; CLOSE ALL; UNLISTEN *; SELECT pg_advisory_unlock_all(); DISCARD PLANS; DISCARD SEQUENCES; DISCARD TEMP;

Examples

DISCARD global temporary tables

```
CREATE GLOBAL TEMP TABLE t_global_temp(a int,b int);
```

NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

```
CREATE TABLE
INSERT INTO t_global_temp VALUES(1,1),(2,2);
INSERT 0 2

DROP TABLE t_global_temp;
ERROR: can not DROP TABLE when global temp table "t_global_temp" is in use

DISCARD GLOBAL TEMP TABLE t_global_temp;

DROP TABLE t_global_temp;
DROP TABLE
```

The **DISCARD VOLATILE** command clears all resources related to volatile temporary tables in the current session.

```
CREATE VOLATILE TEMP TABLE TX1(A INT) DISTRIBUTE BY HASH(A);
CREATE TABLE
CREATE VOLATILE TEMP TABLE TX2(A INT) DISTRIBUTE BY HASH(A);
CREATE TABLE

SELECT * FROM TX1;
 a
---
(0 rows)
SELECT * FROM TX2;
 a
---
(0 rows)

DISCARD VOLATILE TEMP;

SELECT * FROM TX1;
ERROR: relation "tx1" does not exist
LINE 1: SELECT * FROM TX1;
          ^
SELECT * FROM TX2;
ERROR: relation "tx2" does not exist
LINE 1: SELECT * FROM TX2;
```

After **DISCARD TEMP** is run, all temporary table resources in the current session are cleared.

```
CREATE GLOBAL TEMP TABLE t_global_temp(a int,b int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
INSERT INTO t_global_temp VALUES(1,1),(2,2);
INSERT 0 2

CREATE VOLATILE TEMP TABLE t_volatile_temp(a int,b int);
CREATE TEMP TABLE t_temp(a int,b int);

DISCARD TEMP;

SELECT * FROM t_global_temp;
 a | b
---+---
(0 rows)

SELECT * FROM t_volatile_temp;
ERROR: relation "t_volatile_temp" does not exist
LINE 1: select * from t_volatile_temp;

SELECT * FROM t_temp;
ERROR: relation "t_temp" does not exist
LINE 1: select * from t_temp;
```

13.71 DROP BLOCK RULE

Function

Delete a query filtering rule.

This syntax is supported only by clusters of 9.1.0.100 and later versions.

Precautions

Only a user with the database owner permission or the **gs_role_block** role permission can run the **DROP BLOCK RULE** command. A system administrator has this permission by default.

Syntax

```
DROP BLOCK RULE [ IF EXISTS ] block_name;
```

Parameter Description

Table 13-138 DROP BLOCK RULE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified query filtering rule does not exist.	-
block_name	Name of the query filtering rule to be deleted.	The value must be an existing filter rule name.

Examples

Delete a query filtering rule named **query_block**.

```
DROP BLOCK RULE query_block;  
DROP BLOCK RULE IF EXISTS query_block;
```

13.72 DROP DATABASE

Function

Delete the database.

Precautions

- Only the owner of a database or a system administrator has the permission to run the **DROP DATABASE** command.
- DROP DATABASE** does not take effect for the three preinstalled system databases (**gaussdb**, **TEMPLATE0**, and **TEMPLATE1**) because they are

protected. To check databases in the current service, run the `\l` command of `gsql`.

- This command cannot be run while the database to be deleted is associated with a user. You can check the current database connections in the `v$session` view.
- **DROP DATABASE** cannot be run inside a transaction block.
- If **DROP DATABASE** fails to be run and is rolled back, run **DROP DATABASE IF EXISTS**.
- If a "database is being accessed by other users" error is displayed when you run **DROP DATABASE**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.

** DANGER**

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE, USER/ROLE, SCHEMA, TABLE, VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
 - For more information about development and design proposal, see [Development and Design Proposal](#).
-

Syntax

```
DROP DATABASE [ IF EXISTS ] database_name ;
```

Parameter Description

Table 13-139 Parameter Description

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified database does not exist.	-
database_name	Specifies the name of the database to be deleted.	Specifies an existing database name.

Examples

Delete the database named **music**.

```
DROP DATABASE music;
```

Links

[CREATE DATABASE, ALTER DATABASE](#)

FAQs

If you see this error while running the **DROP DATABASE** command, another user is connected to the database.

database "mysql_compatible_db" is being accessed by other users Detail: There are 2 other sessions using the database.

Disconnect all users from the database.

```
CLEAN CONNECTION TO ALL FORCE FOR DATABASE mysql_compatible_db;
```

Retry the **DROP DATABASE** command—it should work. If it does not, perform **CLEAN CONNECTION** and attempt again.

```
DROP DATABASE mysql_compatible_db;
```

13.73 DROP EXCEPT RULE

Function

This syntax deletes a specified exception rule set.

Precautions

Only the system administrator can perform the **DROP EXCEPT RULE** operation.

Syntax

```
DROP EXCEPT RULE [ IF EXISTS ]  
rule_name ;
```

Parameter Description

Table 13-140 DROP EXCEPT RULE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified exception rule does not exist.	-
rule_name	Indicates the name of the exception rule set.	Name of an existing exception rule set.

Examples

Remove the rule set named **except_rule1**:

```
DROP EXCEPT RULE except_rule1;
```

Remove the rule set named **except_rule2** if it exists:

```
DROP EXCEPT RULE IF EXISTS except_rule2;
```

Links

[CREATE EXCEPT RULE, ALTER EXCEPT RULE](#)

13.74 DROP FOREIGN TABLE

Function

DROP FOREIGN TABLE deletes a specified foreign table.

Precautions

DROP FOREIGN TABLE forcibly deletes a specified table. After a table is deleted, any indexes that exist for the table will be deleted. The functions and stored procedures used in this table cannot be run.

Syntax

```
DROP FOREIGN TABLE [ IF EXISTS ]  
table_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-141 DROP FOREIGN TABLE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a message instead of an error if the specified foreign table does not exist.	-
table_name	Specifies the name of the foreign table to be deleted.	An existing table name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes all objects (such as views) that depend on the foreign table to be deleted. • RESTRICT: refuses to delete the foreign table if any objects depend on it. This is the default behaviour.

Examples

Delete the foreign table named **customer_ft**:

```
DROP FOREIGN TABLE customer_ft;
```

Helpful Links

[ALTER FOREIGN TABLE \(GDS Import and Export\)](#), [ALTER FOREIGN TABLE \(for HDFS or OBS\)](#), [CREATE FOREIGN TABLE \(for GDS Import and Export\)](#), [CREATE FOREIGN TABLE \(SQL on OBS or Hadoop\)](#)

13.75 DROP FUNCTION

Function

DROP FUNCTION deletes an existing function.

Precautions

- To delete an overloaded function, you must specify the function's parameter (argument) type. For non-overloaded functions, you can delete them by just specifying the function name.
- If a function involves operations on temporary tables, the function cannot be deleted by running **DROP FUNCTION**.

Syntax

```
DROP FUNCTION [ IF EXISTS ] function_name  
[ ( [ { [ argmode ] [ argname ] argtype } [, ...] ] ) [ CASCADE | RESTRICT ] ];
```

Parameter Description

Table 13-142 DROP FUNCTION parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified table does not exist.	-
function_name	Specifies the name of the function to be deleted.	Use an existing function name.
argmode	Specifies the mode of a function parameter.	-
argname	Specifies the name of a function parameter.	-
argtype	Specifies the data type of a function parameter.	-

Parameter	Description	Value Range
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes all objects that depend on the function to be deleted (such as operators). • RESTRICT: refuses to delete the function if any objects depend on it. This is the default.

Examples

Delete the function **add_two_number**.

```
DROP FUNCTION add_two_number;
```

Helpful Links

[ALTER FUNCTION](#), [CREATE FUNCTION](#)

13.76 DROP GROUP

Function

DROP GROUP deletes a user group.

DROP GROUP is the alias for **DROP ROLE**.

Precautions

DROP GROUP is the internal interface encapsulated in the **gs_om** tool. You are not advised to use this interface, because doing so affects the cluster.

Syntax

```
DROP GROUP [ IF EXISTS ] group_name [, ...];
```

Parameter Description

See [Parameter Description](#) in **DROP ROLE**.

Helpful Links

[CREATE GROUP](#), [ALTER GROUP](#), [DROP ROLE](#)

13.77 DROP INDEX

Function

DROP INDEX deletes an index.

Precautions

Only the owner of an index or a system administrator can run **DROP INDEX** command.

Syntax

```
DROP INDEX [ CONCURRENTLY ][ IF EXISTS ]  
index_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-143 DROP INDEX parameters

Parameter	Description	Value Range
CONCURRENTLY	Deletes an index without locking concurrent selections, inserts, updates, and deletes on the index table. A normal DROP INDEX obtains an exclusive lock on the table to prevent other access until the index is deleted. With this option, the command waits until the conflicting transaction is complete. NOTE Note that only one index name can be specified and the CASCADE option is not supported. (Therefore, indexes that support UNIQUE or PRIMARY KEY constraints cannot be deleted in this way.) Regular DROP INDEX commands can be executed within a transaction block, but cannot be executed in DROP INDEX CONCURRENTLY mode.	-
IF EXISTS	Sends a notice instead of an error if the specified index does not exist.	-
index_name	Specifies the name of the index to be deleted.	An existing index.

Parameter	Description	Value Range
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes all objects that depend on the index to be deleted. • RESTRICT (default): refuses to delete the index if any objects depend on it.

Examples

Delete the **ds_ship_mode_t1_index2** index.

```
DROP INDEX tpcds.ds_ship_mode_t1_index2;
```

Helpful Links

[ALTER INDEX](#), [CREATE INDEX](#)

13.78 DROP MATERIALIZED VIEW

Function

Deletes a materialized view. This syntax is supported only by clusters of version 8.2.1.220 or later.

Precautions

None

Syntax

```
DROP MATERIALIZED VIEW [ IF EXISTS ]  
{[schema.]materialized_view_name} [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-144 DROP MATERIALIZED VIEW parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified materialized view does not exist.	-
materialized_view_name	Name of the materialized view to be deleted.	Name of an existing materialized view.

Parameter	Description	Value Range
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes all objects that are contained in the schema to be deleted. • RESTRICT: refuses to delete the schema that contains any objects. This is the default.

Examples

Delete a materialized view.

```
DROP MATERIALIZED VIEW mv1;
```

Helpful Links

[ALTER MATERIALIZED VIEW](#), [CREATE MATERIALIZED VIEW](#), [REFRESH MATERIALIZED VIEW](#)

13.79 DROP OWNED

Function

DROP OWNED deletes the database objects of a database role.

Precautions

The role's permissions on all the database objects in the current database and shared objects (databases and tablespaces) are revoked.

Syntax

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-145 DROP OWNED parameters

Parameter	Description	Value Range
name	Name of the role whose objects are to be deleted and whose permissions are to be revoked.	An existing role name.

Parameter	Description	Value Range
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes objects that depend on the affected objects. • RESTRICT (default): refuses to delete objects with dependent objects.

Example

Remove all database objects owned by role **u1**.

```
DROP OWNED BY u1;
```

13.80 DROP PUBLICATION

Function

DROP PUBLICATION deletes a publication.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.
- This statement can be used by the publication owner and system administrators only.

Syntax

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Parameter Description

Table 13-146 DROP PUBLICATION parameters

Parameter	Description	Value Range
IF EXISTS	If the specified publication does not exist, no error is thrown. Instead, a notification is reported indicating that the publication does not exist.	-
name	Specifies the name of the publication you want to delete.	An existing publication.
CASCADE RESTRICT	Currently, these keywords do not work because there is no dependency on publications.	-

Examples

Delete a publication.
`DROP PUBLICATION mypublication;`

Helpful Links

[ALTER PUBLICATION](#), [CREATE PUBLICATION](#)

13.81 DROP REDACTION POLICY

Function

DROP REDACTION POLICY deletes a data masking policy applied to a specified table.

Precautions

Only the table owner and users granted the **gs_redaction_policy** preset role have the permission to delete data masking policies.

Syntax

```
DROP REDACTION POLICY [ IF EXISTS ] policy_name ON table_name;
```

Parameter Description

Table 13-147 DROP REDACTION POLICY parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of throwing an error if the redaction policy to be deleted does not exist.	-
policy_name	Specifies the name of the masking policy to be deleted.	An existing name of the masking policy.
table_name	Specifies the name of the table with the masking policy to be deleted.	An existing name of the table to which the data masking policy is applied.

Examples

Delete the masking policy **mask_emp** from table **emp**.

```
DROP REDACTION POLICY mask_emp ON emp;
```

Helpful Links

[ALTER REDACTION POLICY](#), [CREATE REDACTION POLICY](#)

13.82 DROP ROW LEVEL SECURITY POLICY

Function

Deletes a row-level access control policy from a table.

Precautions

Only the table owner or administrators can delete a row-level access control policy from the table.

Syntax

```
DROP [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name [ CASCADE | RESTRICT ]
```

Parameter Description

Table 13-148 DROP ROW LEVEL SECURITY POLICY parameters

Parameter	Description	Value Range
IF EXISTS	Reports a notice instead of an error if the specified row-level access control policy does not exist.	-
policy_name	Specifies the name of a row-level access control policy to be deleted.	An existing name of a row-level access control policy.
table_name	Specifies the name of a table to which a row-level access control policy to be deleted belongs.	An existing name of the data table to which a row-level access control policy belongs.
CASCADE/ RESTRICT	The two parameters are used only for syntax compatibility. No objects depend on access control policies and thereby CASCADE is equivalent to RESTRICT .	-

Examples

Delete the row-level access control policy **all_data_rls** from table **all_data**.

```
DROP ROW LEVEL SECURITY POLICY all_data_rls ON all_data;
```

Helpful Links

[ALTER ROW LEVEL SECURITY POLICY](#), [CREATE ROW LEVEL SECURITY POLICY](#)

13.83 DROP PROCEDURE

Function

DROP PROCEDURE deletes an existing stored procedure.

Precautions

None.

Syntax

```
DROP PROCEDURE [ IF EXISTS ] procedure_name ;
```

Parameter Description

Table 13-149 DROP PROCEDURE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the stored procedure does not exist.	-
procedure_name	Specifies the name of the stored procedure to be deleted.	An existing name of a stored procedure.

Examples

Delete a stored procedure.

```
DROP PROCEDURE prc_add;
```

Helpful Links

[CREATE PROCEDURE](#)

13.84 DROP RESOURCE POOL

Function

DROP RESOURCE POOL deletes a resource pool.

NOTE

If a role has been associated with a resource pool, the resource pool cannot be deleted.

Precautions

The user must have the DROP permission in order to delete a resource pool.

Syntax

```
DROP RESOURCE POOL [ IF EXISTS ] pool_name;
```

Parameter Description

Table 13-150 DROP RESOURCE POOL parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified resource pool does not exist.	-
pool_name	Specifies the name of a created resource pool.	A string, which must comply with the naming convention.

Example

Delete resource pool **pool**.

```
DROP RESOURCE POOL pool;
```

Helpful Links

[ALTER RESOURCE POOL](#), [CREATE RESOURCE POOL](#)

13.85 DROP ROLE

Function

DROP ROLE deletes a specified role.

Precautions

If a "role is being used by other users" error is displayed when you run **DROP ROLE**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.



DANGER

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE**, **USER/ROLE**, **SCHEMA**, **TABLE**, **VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
DROP ROLE [ IF EXISTS ] role_name [, ...];
```

Parameter Description

Table 13-151 DROP ROLE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified role does not exist.	-
role_name	Specifies the name of the role to be deleted.	An existing role.

Examples

Delete **manager** role.

```
DROP ROLE manager;
```

Helpful Links

[CREATE ROLE](#), [ALTER ROLE](#), [SET ROLE](#)

13.86 DROP SCHEMA

Function

DROP SCHEMA deletes a schema in a database.

Precautions

- Only the owner of a schema or a user granted with the DROP permission for the schema or a system administrator has the permission to execute the **Drop SCHEMA** statement.

** DANGER**

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE**, **USER/ROLE**, **SCHEMA**, **TABLE**, **VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
 - For more information about development and design proposal, see [Development and Design Proposal](#).
-

Syntax

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-152 DROP SCHEMA parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified schema does not exist.	-
schema_name	Specifies the name of the schema to be deleted.	An existing schema name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes all objects that are contained in the schema to be deleted. • RESTRICT: refuses to delete the schema that contains any objects. This is the default.

NOTICE

- Do not delete the schemas with the beginning of **pg_temp** or **pg_toast_temp**. They are internal system schemas, and deleting them may cause unexpected errors.
- A user cannot delete the schema in use. To delete the schema in use, switch to another schema.

Example

Delete **ds_new** schema.

```
DROP SCHEMA ds_new;
```

Links

[ALTER SCHEMA](#), [CREATE SCHEMA](#)

13.87 DROP SEQUENCE

Function

DROP SEQUENCE deletes a sequence from the current database.

Precautions

Only a sequence owner or a system administrator can delete a sequence.

Syntax

```
DROP SEQUENCE [ IF EXISTS ] {[schema.]sequence_name} [ , ... ] [ CASCADE | RESTRICT ] ;
```

Parameter Description

Table 13-153 DROP SEQUENCE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified sequence does not exist.	-
name	Specifies the name of the sequence to be deleted.	Specifies an existing sequence name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	● CASCADE: Automatically deletes objects that depend on the sequence to be deleted. ● RESTRICT: Refuses to delete the sequence if any objects depend on it. This is the default action.

Examples

Delete sequence **serial**.

```
DROP SEQUENCE serial;
```

Helpful Links

[ALTER SEQUENCE](#), [CREATE SEQUENCE](#)

13.88 DROP SERVER

Function

DROP SERVER deletes an existing data server.

Precautions

Only the server owner can delete a server.

Syntax

```
DROP SERVER [ IF EXISTS ] server_name [ {CASCADE | RESTRICT} ] ;
```

Parameter Description

Table 13-154 DROP SERVER parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified server does not exist.	-
server_name	Specifies the name of the server to be deleted.	An existing server name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically drops objects that depend on the server to be deleted. • RESTRICT (default): refuses to delete the server if any objects depend on it.

Examples

Delete **hdfs_server** server.

```
DROP SERVER hdfs_server;
```

Helpful Links

[ALTER SERVER](#), [CREATE SERVER](#)

13.89 DROP SUBSCRIPTION

Function

DROP SUBSCRIPTION deletes a subscription.

Precautions

- This statement is supported by version 8.2.0.100 or later clusters.
- A subscription can be deleted by the system administrator only.

Syntax

```
DROP SUBSCRIPTION [ IF EXISTS ] name
```

Parameter Description

Table 13-155 DROP SUBSCRIPTION parameters

Parameter	Description	Value Range
IF EXISTS	If the specified subscription does not exist, no error is thrown. Instead, a notification is reported indicating that the subscription does not exist.	-
name	Specifies the name of the subscription you want to delete.	An existing subscription.

Examples

Delete a subscription.
DROP SUBSCRIPTION mysub;

Helpful Links

[ALTER SUBSCRIPTION](#), [CREATE SUBSCRIPTION](#)

13.90 DROP SYNONYM

Function

DROP SYNONYM is used to delete a synonym object.

Precautions

Only a synonym owner or a system administrator can run the **DROP SYNONYM** command.

Syntax

```
DROP SYNONYM [ IF EXISTS ] synonym_name [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-156 DROP SYNONYM parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of reporting an error if the specified synonym does not exist.	-

Parameter	Description	Value Range
synonym_name	Name of a synonym which is deleted (optionally with schema names)	An existing synonym name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes objects (such as views) that depend on the synonym to be deleted. • RESTRICT: refuses to delete the synonym if any objects depend on it. This is the default.

Examples

Delete a synonym.

```
DROP SYNONYM t1;
```

Helpful Links

[ALTER SYNONYM](#) and [CREATE SYNONYM](#)

13.91 DROP TABLE

Function

DROP TABLE deletes a specified table.

Precautions

- **DROP TABLE** forcibly deletes a specified table. After a table is deleted, any indexes that exist for the table will be deleted; any functions or stored procedures that use this table cannot be run. Deleting a partitioned table also deletes all partitions in the table.
- Only the table owner, schema owner, or a user granted with the DROP permission can run **DROP TABLE** on a table. A system administrator has this permission by default. To delete all the rows in a table but retain the table definition, use **TRUNCATE** or **DELETE**.

** DANGER**

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE, USER/ROLE, SCHEMA, TABLE, VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
DROP TABLE [ IF EXISTS ]  
{ [schema.]table_name } [, ...] [ CASCADE | RESTRICT ] ;
```

Parameter Description

Table 13-157 DROP TABLE parameters

Parameter	Description	Value Range
IF EXISTS	Reports a notice instead of an error if the specified table does not exist.	-
schema	Name of the schema to which the table to be deleted belongs.	-
table_name	Specifies the name of the table to be deleted.	An existing table name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: automatically deletes objects (such as views) that depend on the table to be deleted. • RESTRICT (default): refuses to delete the table if any objects depend on it. This is the default.
PURGE	If the Purge parameter is specified, data is directly deleted regardless of whether the recycle bin feature is enabled. This is supported only by clusters of version 9.1.0.200 or later.	-

Parameter	Description	Value Range
KEEP ttl HOUR	In V3 tables, enabling the recycle bin feature and setting a TTL allows physical files to be moved to the recycle bin for potential restoration. However, once the TTL expires, the background process automatically clears the objects in the recycle bin. This is supported only by clusters of version 9.1.0.200 or later. NOTICE You are advised to use this parameter in scenarios where the DROP and TRUNCATE operations are not frequently performed.	-

Examples

Delete **warehouse_t1** table.

```
DROP TABLE tpcds.warehouse_t1;
```

Helpful Links

[ALTER TABLE](#), [12.101-RENAME TABLE](#), and [CREATE TABLE](#)

13.92 DROP TEXT SEARCH CONFIGURATION

Function

DROP TEXT SEARCH CONFIGURATION deletes an existing text search configuration.

Precautions

To run the **DROP TEXT SEARCH CONFIGURATION** command, you must be the owner of the text search configuration.

Syntax

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```


Parameter Description

Table 13-158 DROP TEXT SEARCH CONFIGURATION parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified text search configuration does not exist.	-
name	Specifies the name (optionally schema-qualified) of a text search configuration to be deleted.	An existing text search configuration name.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	● CASCADE: Automatically deletes objects that depend on the text search configuration to be deleted. ● RESTRICT: Refuses to delete the text search configuration if any objects depend on it. This is the default. This is the default.

Examples

Delete the text search configuration **ngram1**.

```
DROP TEXT SEARCH CONFIGURATION ngram1;
```

Helpful Links

[ALTER TEXT SEARCH CONFIGURATION](#), [CREATE TEXT SEARCH CONFIGURATION](#)

13.93 DROP TEXT SEARCH DICTIONARY

Function

DROP TEXT SEARCH DICTIONARY deletes a full-text retrieval dictionary.

Precautions

- **DROP** is not supported by predefined dictionaries.
- Only the owner of a dictionary can do **DROP** to the dictionary. System administrators have this permission by default.
- Execute **DROP...CASCADE** only when necessary because this operation will delete the text search configuration that uses this dictionary.

Syntax

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Parameter Description

Table 13-159 DROP TEXT TEXT SEARCH DICTIONARY parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified full-text search dictionary does not exist.	-
name	Specifies the name of a dictionary to be deleted. (If you do not specify a schema name, the dictionary in the current schema will be deleted by default.)	Name of an existing dictionary.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: Automatically deletes dependent objects of a dictionary and then deletes all dependent objects of these objects in sequence. If any text search configuration that uses the dictionary exists, DROP execution will fail. You can add CASCADE to delete all text search configurations and dictionaries that use the dictionary. • RESTRICT: Rejects the deletion of a dictionary if any object depends on the dictionary. This is the default.

Examples

Delete **english** dictionary.

```
DROP TEXT SEARCH DICTIONARY english;
```

Helpful Links

[ALTER TEXT SEARCH DICTIONARY](#), [CREATE TEXT SEARCH DICTIONARY](#)

13.94 DROP TRIGGER

Function

DROP TRIGGER deletes a trigger.

Precautions

Only the owner of a trigger and system administrators can run the **DROP TRIGGER** statement.

Syntax

```
DROP TRIGGER [ IF EXISTS ] trigger_name ON table_name [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-160 DROP TRIGGER parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified trigger does not exist.	-
trigger_name	Specifies the name of the trigger to be deleted.	An existing trigger.
table_name	Specifies the name of the table where the trigger to be deleted is located.	An existing table having a trigger
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: Deletes objects that depend on the trigger. • RESTRICT: Refuses to delete the trigger if any objects depend on it. This is the default.

Examples

Delete trigger **insert_trigger**.

```
DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
```

Helpful Links

[CREATE TRIGGER](#), [ALTER TRIGGER](#), [ALTER TABLE](#)

13.95 DROP TYPE

Function

DROP TYPE deletes a user-defined data type. Only the type owner has permission to run this statement.

Syntax

```
DROP TYPE [ IF EXISTS ] name [ , ... ] [ CASCADE | RESTRICT ]
```

Parameter Description

Table 13-161 DROP TYPE parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified type does not exist.	-
name	Specifies the name of the type to be deleted (schema-qualified).	Specifies an existing domain type.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: Deletes objects (such as columns, functions, and operators) that depend on the type. • RESTRICT: Refuses to delete the type if any objects depend on it. This is the default.

Examples

Delete **compfoo** type.

```
DROP TYPE compfoo cascade;
```

Helpful Links

[ALTER TYPE](#), [CREATE TYPE](#)

13.96 DROP USER

Function

Deleting a user will also delete the schema having the same name as the user.

Precautions

- **CASCADE** is used to delete objects (excluding databases) that depend on the user. **CASCADE** cannot delete locked objects unless the locked objects are unlocked or the processes that lock the objects are stopped.
- When deleting a user in the database, if the object that the user depends on is in another database or the object of the dependent user is another database, you need to manually delete the dependent objects in other databases or delete the dependent database. Then, delete the user. Cross-database cascading deletion cannot be performed.
- In a multi-tenant scenario, the service user will also be deleted when you delete a user group. If the specified **CASCADE** concatenation is deleted, **CASCADE** will be specified upon the deletion of the service user. If you fail to delete a user, an error is reported, and you cannot delete other users either.
- If the error table specified by the GDS foreign table created by user A is under the schema of user B, user B cannot be deleted by specifying the **CASCADE** keyword in **DROP USER**.
- If a "role is being used by other users" error is displayed when you run **DROP USER**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.



DANGER

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE, USER/ROLE, SCHEMA, TABLE, VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
DROP USER [ IF EXISTS ] user_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-162 DROP USER parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified user does not exist.	-
user_name	Specifies the name of a user to be deleted.	Name of an existing user.

Parameter	Description	Value Range
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed. NOTE In DWS, the <code>postgresql.conf</code> file contains the <code>enable_kill_query</code> parameter. This parameter affects the action of deleting user objects using CASCADE. • If <code>enable_kill_query</code> is on and CASCADE is used to delete user objects, the processes will be automatically killed and the user will be deleted at the same time. • If <code>enable_kill_query</code> is off and CASCADE is used to delete user objects, the user will be deleted after the processes are automatically killed.	• CASCADE: automatically deletes all objects (such as tables) that depend on the user to be deleted. When a user is deleted in CASCADE mode, objects owner by the user and the user's permissions for objects will be deleted. • RESTRICT: refuses to delete the user if any objects depend on it. This is the default.

Examples

Delete user **jim**.

```
DROP USER jim CASCADE;
```

Helpful Links

[ALTER USER](#), [CREATE USER](#)

13.97 DROP VIEW

Function

DROP VIEW forcibly deletes an existing view in a database.

Precautions

Only a view owner or a system administrator can run **DROP VIEW** command.



DANGER

- Be cautious when using **DROP OBJECT** (e.g., **DATABASE**, **USER/ROLE**, **SCHEMA**, **TABLE**, **VIEW**) as it may cause data loss, especially with **CASCADE** deletions. Always back up data before proceeding.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
DROP VIEW [ IF EXISTS ] view_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

Table 13-163 DROP VIEW parameters

Parameter	Description	Value Range
IF EXISTS	Sends a notice instead of an error if the specified view does not exist.	-
view_name	Specifies the name of the view to be deleted.	An existing view.
CASCADE RESTRICT	Specifies how to process related data in the dependent object when a delete operation is performed.	• CASCADE: deletes objects (such as other views) that depend on a view to be deleted. • RESTRICT: refuses to delete the view if any objects depend on it. This is the default.

Examples

Delete **myView** view.

```
DROP VIEW myView;
```

Delete **customer_details_view_v2** view.

```
DROP VIEW public.customer_details_view_v2;
```

Helpful Links

[ALTER VIEW](#), [CREATE VIEW](#)

13.98 FETCH

Function

FETCH retrieves data using a previously-created cursor.

A cursor has an associated position, which is used by **FETCH**. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result.

- When created, a cursor is positioned before the first row.
- After fetching some rows, the cursor is positioned on the row most recently retrieved.

- If **FETCH** runs off the end of the available rows then the cursor is left positioned after the last row, or before the first row if fetching backward.
- **FETCH ALL** or **FETCH BACKWARD ALL** will always leave the cursor positioned after the last row or before the first row.

Precautions

- If **NO SCROLL** is defined for the cursor, a backward fetch like **FETCH BACKWARD** is not allowed.
- The forms **NEXT**, **PRIOR**, **FIRST**, **LAST**, **ABSOLUTE**, and **RELATIVE** appropriately fetch a record after moving the cursor. If the cursor is already after the last row before being moved, an empty result is returned, and the cursor is left positioned before the first row (backward fetch) or after the last row (forward fetch) as appropriate.
- The forms using **FORWARD** and **BACKWARD** retrieve the indicated number of rows moving in the forward or backward direction, leaving the cursor positioned on the last-returned row (or after (backward fetch)/before (forward fetch) all rows, if the count exceeds the number of rows available).
- **RELATIVE 0**, **FORWARD 0**, and **BACKWARD 0** all request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row, in which case, no row is returned.
- If the cursor of **FETCH** involves a column-store table, backward fetches like **BACKWARD**, **PRIOR**, and **FIRST** are not supported.

Syntax

```
FETCH [ direction { FROM | IN } ] cursor_name;
```

The **direction** clause specifies optional parameters.

```
NEXT
| PRIOR
| FIRST
| LAST
| ABSOLUTE count
| RELATIVE count
| count
| ALL
| FORWARD
| FORWARD count
| FORWARD ALL
| BACKWARD
| BACKWARD count
| BACKWARD ALL
```

Parameter Description

Table 13-164 FETCH parameters

Parameter	Description	Value Range
direction_clause	Defines the fetch direction.	For details, see Table 13-165 .

Parameter	Description	Value Range
{ FROM IN } cursor_name	Specifies the cursor name using the keyword FROM or IN .	Name of the cursor

Table 13-165 Value range of direction_clause

Value Range	Description	Remarks
NEXT (default value)	Fetches the next row.	-
PRIOR	Fetches the prior row.	-
FIRST	Fetches the first row of the query (same as ABSOLUTE 1).	-
LAST	Fetches the last row of the query (same as ABSOLUTE -1).	-
ABSOLUTE count	Fetches the (count)'th row of the query. ABSOLUTE fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway.	count is a possibly-signed integer constant: • If count is a positive integer, fetches the (count)'th row of the query, starting from the first row. If count is less than the current cursor position, a rewind operation is required, which is currently not supported. • If count is a negative value or 0, a backward scanning is required, which is currently not supported.

Value Range	Description	Remarks
RELATIVE count	Fetches the (count)'th succeeding row, or the abs(count)'th prior row if count is negative.	count is a possibly-signed integer constant: • If count is a positive integer, fetches the (count)'th succeeding row. • If count is a negative value, a backward scanning is required, which is currently not supported. • RELATIVE 0 fetches the current row.
count	Fetches the next count rows (same as FORWARD count).	-
ALL	Fetches all remaining rows (same as FORWARD ALL).	-
FORWARD	Fetches the next row (same as NEXT).	-
FORWARD count	Fetches the next or prior count rows (same as RELATIVE count).	-
FORWARD ALL	Fetches all remaining rows.	-
BACKWARD	Fetches the prior row (same as PRIOR).	-
BACKWARD count	Fetches the prior count rows (scanning backwards).	count is a possibly-signed integer constant: • If count is a positive integer, fetches the (count)'th prior row. • If count is a negative integer, fetches the abs(count)'th succeeding row. • BACKWARD 0 re-fetches the current row.
BACKWARD ALL	Fetches all prior rows (scanning backwards).	-

Examples

Example 1: Run the **SELECT** statement to read a table using a cursor.

Set up the **cursor1** cursor.

```
CURSOR cursor1 FOR SELECT * FROM tpods.customer_address ORDER BY 1;
```

Fetch the first three rows from **cursor1**.

```
FETCH FORWARD 3 FROM cursor1;
```

Example 2: Use a cursor to read the content in the **VALUES** clause.

Set up the cursor **cursor2**.

```
CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
```

Fetch the first two rows from **cursor2**.

```
FETCH FORWARD 2 FROM cursor2;
```

Helpful Links

[CLOSE](#), [MOVE](#)

13.99 MOVE

Function

MOVE repositions a cursor without retrieving any data. **MOVE** works exactly like the [FETCH](#) command, except it only repositions the cursor and does not return rows.

Precautions

None

Syntax

```
MOVE [ direction [ FROM | IN ] ] cursor_name;
```

The **direction** clause specifies optional parameters.

```
NEXT  
| PRIOR  
| FIRST  
| LAST  
| ABSOLUTE count  
| RELATIVE count  
| count  
| ALL  
| FORWARD  
| FORWARD count  
| FORWARD ALL  
| BACKWARD  
| BACKWARD count  
| BACKWARD ALL
```

Parameter Description

MOVE command parameters are the same as **FETCH** command parameters. For details, see [Parameter Description](#) in **FETCH**.

 NOTE

On successful completion, a **MOVE** command returns a command tag of the form **MOVE count**. The **count** is the number of rows that a **FETCH** command with the same parameters would have returned (possibly zero).

Examples

Create table **reason** and insert data into it.

```
DROP TABLE IF EXISTS reason;
CREATE TABLE reason
(
  a  int primary key,
  b  int,
  c  int
);
INSERT INTO reason VALUES (1, 2, 3);
```

Start a transaction.

```
START TRANSACTION;
```

Define **cursor1** cursor.

```
CURSOR cursor1 FOR SELECT * FROM tpeds.reason;
```

Skip the first three rows of **cursor1**.

```
MOVE FORWARD 3 FROM cursor1;
```

Fetch the first four rows from **cursor1**.

```
FETCH 4 FROM cursor1;
```

Close a cursor.

```
CLOSE cursor1;
```

End the transaction.

```
END;
```

Helpful Links

[CLOSE](#), [FETCH](#)

13.100 REFRESH MATERIALIZED VIEW

Function

REFRESH MATERIALIZED VIEW refreshes materialized views. This syntax is supported only by clusters of version 8.2.1.220 or later.

The refresh mode is specified by the REFRESH parameter in the [CREATE MATERIALIZED VIEW](#) syntax. Currently, full refresh and scheduled refresh are supported.

Precautions

The refresh operation blocks the DML operations on the base table.

Syntax

```
REFRESH MATERIALIZED VIEW  
{[schema.]materialized_view_name} [ [DEFAULT] | [RESTRICT] | [CASCADE FORWARD] | [CASCADE  
BACKWARD] | [CASCADE ALL] ]
```

Parameter Description

Table 13-166 REFRESH MATERIALIZED VIEW parameters

Parameter	Description	Value Range
materialize d_view_na me	Specifies the name of the materialized view to be refreshed.	-
DEFAULT	Specifies default refresh. It means that a single materialized view is not refreshed only until the base table has changed.	-
RESTRICT	Specifies forced refresh. It means that a materialized view is refreshed regardless of whether the base table has changed.	-
CASCADE FORWARD	Indicates a cascading refresh in the upward direction, starts by refreshing the current materialized view, and then refreshes the nested materialized views that depend it. The default refresh method is used.	-
CASCADE BACKWARD	Specifies cascading refresh in the downward direction. The system refreshes the lowest-level materialized view that the current materialized view relies on, then continues to refresh the nested materialized views that depend on the lowest-level one, and finally refreshes the current materialized view itself. The default refresh method is used.	-
CASCADE ALL	Specifies all cascading refreshes. First, cascading refreshes are performed downward, and then cascading refreshes are performed upward.	-

Examples

Refresh a materialized view.

```
REFRESH MATERIALIZED VIEW mv1;
```

Perform a cascading refresh in the downward direction for materialized views.

```
CREATE TABLE t1(a int,b int);
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM t1 where a > 0;
CREATE MATERIALIZED VIEW mv2 AS SELECT * FROM mv1;
INSERT INTO t1 values(1,1),(2,2);
REFRESH MATERIALIZED VIEW mv2 cascade backward; -- After refreshing mv2, the data in mv1 was also updated.
```

```
SELECT * FROM mv1 ORDER BY 1,2;
a | b
---+---
1 | 1
2 | 2
(2 rows)
```

```
SELECT * FROM mv2 ORDER BY 1,2;
a | b
---+---
1 | 1
2 | 2
(2 rows)
```

Helpful Links

[CREATE MATERIALIZED VIEW](#), [ALTER MATERIALIZED VIEW](#), [DROP MATERIALIZED VIEW](#)

13.101 REINDEX

Function

REINDEX rebuilds an index using the data stored in the index's table, replacing the old copy of the index.

There are several scenarios in which **REINDEX** can be used:

- An index has become corrupted, and no longer contains valid data.
- An index has become "bloated", that is, it contains many empty or nearly-empty pages.
- You have altered a storage parameter (such as fillfactor) for an index, and wish to ensure that the change has taken full effect.

An index build with the **CONCURRENTLY** option failed, leaving an "invalid" index.

Precautions

Index reconstruction of the **REINDEX DATABASE** or **SYSTEM** type cannot be performed in transaction blocks.

In the storage-compute decoupling architecture, REINDEX full database indexes are not supported.

**WARNING**

- Do not perform the **CREATE INDEX** and **REINDEX** operations on large tables during peak hours.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

- Recreate indexes for the entire database or system catalogs.
`REINDEX { DATABASE | SYSTEM } name [ FORCE ];`
- Recreate an ordinary index or an index on an ordinary table.
`REINDEX { INDEX | TABLE } name [ FORCE | WITHOUT UNUSABLE ]`
- Recreate an index partition or an index on a partition.
`REINDEX { | TABLE } name
PARTITION partition_name [FORCE | WITHOUT UNUSABLE];`

Parameter Description

Table 13-167 REINDEX parameters

Parameter	Description	Value Range
INDEX	Recreates the specified index.	-
TABLE	Recreates all indexes of the specified table. If the table has a secondary TOAST table, that is reindexed as well.	-
DATABASE	Recreates all indexes within the current database.	-
SYSTEM	Recreates all indexes on system catalogs within the current database. Indexes on user tables are not processed.	-
name	Name of the specific index, table, or database to be reindexed. Index and table names can be schema-qualified. NOTE REINDEX DATABASE and SYSTEM can create indexes for only the current database. Therefore, name must be the same as the current database name.	-
FORCE	This is an obsolete option. It is ignored if specified.	-

Parameter	Description	Value Range
WITHOUT UNUSABLE	If this parameter is specified, the UNUSABLE indexes or UNUSABLE index partitions on the object is ignored during REINDEX . This parameter is supported only by 9.1.0.100 and later versions.	-
partition_name	Specifies the name of the partition or index partition to be reindexed.	• If it is REINDEX INDEX, specify the name of an index partition. • If it is REINDEX TABLE, specify the name of a partition.

NOTICE

Index reconstruction of the **REINDEX DATABASE** or **SYSTEM** type cannot be performed in transaction blocks.

Examples

Rebuild a single index.

```
REINDEX INDEX tpcds.tpcds_customer_index1;
```

Rebuild all indexes on the **tpcds.customer_t1** table.

```
REINDEX TABLE tpcds.customer_t1;
```

Rebuild the partition index of partition **P1** in the partitioned table **customer_address**.

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
  ca_address_sk    INTEGER          NOT NULL ,
  ca_address_id    CHARACTER(16)    NOT NULL ,
  ca_street_number CHARACTER(10)
  , ca_street_name  CHARACTER varying(60)
  , ca_street_type  CHARACTER(15)
  , ca_suite_number CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
  PARTITION P1 VALUES LESS THAN(2450815),
  PARTITION P2 VALUES LESS THAN(2451179),
  PARTITION P3 VALUES LESS THAN(2451544),
  PARTITION P4 VALUES LESS THAN(MAXVALUE)
);

CREATE INDEX customer_address_index on customer_address(CA_ADDRESS_SK) LOCAL;

REINDEX TABLE customer_address PARTITION P1;
```


13.102 RENAME TABLE

Function

RENAME TABLE renames a specified table.

Precautions

RENAME TABLE has the same function as the following command:

```
ALTER TABLE table_name RENAME to new_table_name
```

Syntax

```
RENAME TABLE  
{[schema.]table_name TO new_table_name} [, ...];
```

Parameter Description

Table 13-168 RENAME TABLE parameters

Parameter	Description	Value Range
schema	Specifies the schema name.	Valid schema name
table_name	Specifies the name of the table to be modified.	Name of an existing table
new_table_name	Specifies the new table name.	A string that must comply with the rules in Identifier Naming Conventions .

Examples

Create a column-store table and specify the storage format and compression mode.

```
DROP TABLE IF EXISTS customer_address;  
CREATE TABLE customer_address  
(  
  ca_address_sk    INTEGER          NOT NULL ,  
  ca_address_id    CHARACTER(16)    NOT NULL ,  
  ca_street_number CHARACTER(10)      ,  
  ca_street_name   CHARACTER varying(60) ,  
  ca_street_type   CHARACTER(15)      ,  
  ca_suite_number  CHARACTER(10)  
)  
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)  
DISTRIBUTE BY HASH (ca_address_sk);
```

Rename a table.

```
RENAME TABLE customer_address TO new_customer_address;
```

Helpful Links

[CREATE TABLE](#), [ALTER TABLE](#), [DROP TABLE](#)

13.103 RESET

Function

RESET restores run-time parameters to their default values. The default values are parameter default values complied in the **postgresql.conf** configuration file.

RESET is an alternative spelling for:

SET configuration_parameter TO DEFAULT

Precautions

RESET and **SET** have the same transaction behavior. Their impact will be rolled back.

Syntax

```
RESET {configuration_parameter | CURRENT_SCHEMA | TIME_ZONE | TRANSACTION ISOLATION LEVEL |  
SESSION AUTHORIZATION | ALL };
```

Parameter Description

Table 13-169 RESET parameters

Parameter	Description	Value Range
configuration_parameter	Specifies the name of a settable run-time parameter.	You can view run-time parameters by running the SHOW ALL command. NOTE Some parameters that viewed by SHOW ALL cannot be set by SET . For example, max_datanodes .
CURRENT_SCHEMA	Specifies the current schema.	-
TIME_ZONE	Specifies the time zone.	-
TRANSACTION ISOLATION LEVEL	Specifies the transaction isolation level.	-
SESSION AUTHORIZATION	Specifies the session authorization.	-

Parameter	Description	Value Range
ALL	Specifies all runtime parameters.	-

Examples

Reset **timezone** to the default value.

```
RESET timezone;
```

Set all parameters to their default values.

```
RESET ALL;
```

Helpful Links

[SET](#), [SHOW](#)

13.104 SET

Function

SET modifies a run-time parameter.

Precautions

Most run-time parameters can be modified by executing **SET**. Some parameters cannot be modified after a server or session starts.

Syntax

- Set the system time zone.

```
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT };
```
- Set the schema of the table.

```
SET [ SESSION | LOCAL ]  
  { CURRENT_SCHEMA { TO | = } { schema | DEFAULT }  
  | SCHEMA 'schema' };
```
- Set client encoding.

```
SET [ SESSION | LOCAL ] NAMES encoding_name;
```
- Set XML parsing mode.

```
SET [ SESSION | LOCAL ] XML OPTION { DOCUMENT | CONTENT };
```
- Set other running parameters.

```
SET [ LOCAL | SESSION ]  
  { { config_parameter { { TO | = } { value | DEFAULT }  
    | FROM CURRENT } } };
```

Parameter Description

Table 13-170 SET parameters

Parameter	Description	Value Range
SESSION	Indicates that the specified parameters take effect for the current session. This is the default value if neither SESSION nor LOCAL appears. If SET or SET SESSION is executed within a transaction that is later aborted, the effects of the SET command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another SET.	-
LOCAL	Indicates that the specified parameters take effect for the current transaction. After COMMIT or ROLLBACK, the session-level setting takes effect again. The effects of SET LOCAL last only till the end of the current transaction, whether committed or not. A special case is SET followed by SET LOCAL within a single transaction: the SET LOCAL value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the SET value will take effect.	-
TIME_ZONE timezone	Indicates the local time zone for the current session.	A valid local time zone. The corresponding run-time parameter is TimeZone . The default value is PRC .
CURRENT_SCHEMA schema	Indicates the current schema.	Specifies an existing schema name.
SCHEMA schema	Indicates the current schema. Here the schema is a string. Example: set schema 'public';	-

Parameter	Description	Value Range
NAMES encoding_name	Indicates the client character encoding name. This command is equivalent to set client_encoding to encoding_name .	A valid character encoding name. The run-time parameter corresponding to this option is client_encoding . The default encoding is UTF8 .
XML OPTION option	Indicates the XML resolution mode.	CONTENT (default) and DOCUMENT
config_parameter	Indicates the configurable run-time parameters. You can use SHOW ALL to view available run-time parameters. NOTE Some parameters that viewed by SHOW ALL cannot be set by SET . For example, max_datanodes .	-
value	Indicates the new value of the config_parameter parameter. This parameter can be specified as string constants, identifiers, numbers, or comma-separated lists of these. DEFAULT can be written to indicate resetting the parameter to its default value.	-

Examples

Configure the search path of the **tpcds** schema.

```
SET search_path TO tpcds, public;
```

Set the date style to the traditional POSTGRES style (date placed before month).

```
SET datestyle TO postgres;
```

Helpful Links

[RESET](#), [SHOW](#)

13.105 SET CONSTRAINTS

Function

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction.

IMMEDIATE constraints are checked at the end of each statement. **DEFERRED** constraints are not checked until transaction commit. Each constraint has its own **IMMEDIATE** or **DEFERRED** mode.

Upon creation, a constraint is given one of three characteristics **DEFERRABLE INITIALLY DEFERRED**, **DEFERRABLE INITIALLY IMMEDIATE**, or **NOT DEFERRABLE**. The third class is always **IMMEDIATE** and is not affected by the **SET CONSTRAINTS** command. The first two classes start every transaction in specified modes, but its behaviors can be changed within a transaction by **SET CONSTRAINTS**.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If multiple constraints match a name, the name is affected by all of these constraints. **SET CONSTRAINTS ALL** changes the modes of all deferrable constraints.

When **SET CONSTRAINTS** changes the mode of a constraint from **DEFERRED** to **IMMEDIATE**, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the **SET CONSTRAINTS** command. If any such constraint is violated, the **SET CONSTRAINTS** fails (and does not change the constraint mode). Therefore, **SET CONSTRAINTS** can be used to force checking of constraints to occur at a specific point in a transaction.

Only foreign key constraints are affected by this setting. Check and unique constraints are always checked immediately when a row is inserted or modified.

Precautions

SET CONSTRAINTS sets the behavior of constraint checking only within the current transaction. Therefore, if you execute this command outside of a transaction block (**START TRANSACTION/COMMIT** pair), it will not appear to have any effect.

Syntax

```
SET CONSTRAINTS { ALL | { name } [, ...] } { DEFERRED | IMMEDIATE } ;
```

Parameter Description

Table 13-171 SET CONSTRAINTS parameters

Parameter	Description	Value Range
name	Specifies the constraint name.	An existing constraint name, which can be found in the system catalog pg_constraint .
ALL	Indicates all constraints.	-
DEFERRED	Indicates that constraints are not checked until transaction commit.	-

Parameter	Description	Value Range
IMMEDIATE	Indicates that constraints are checked at the end of each statement.	-

Examples

Set that constraints are checked when a transaction is committed.

```
SET CONSTRAINTS ALL DEFERRED;
```

13.106 SET ROLE

Function

SET ROLE sets the current user identifier of the current session.

Precautions

- Users of the current session must be members of specified **rolename**, but the system administrator can choose any roles.
- Executing this command may add rights of a user or restrict rights of a user. If the role of a session user has the **INHERITS** attribute, it automatically has all rights of roles that **SET ROLE** enables the role to be. In this case, **SET ROLE** physically deletes all rights directly granted to session users and rights of its belonging roles and only leaves rights of the specified roles. If the role of the session user has the **NOINHERITS** attribute, **SET ROLE** deletes rights directly granted to the session user and obtains rights of the specified role.

Syntax

- Set the current user identifier of the current session.

```
SET [ SESSION | LOCAL ] ROLE role_name PASSWORD '{password}';
```
- Reset the current user identifier to that of the current session.

```
RESET ROLE;
```

Parameter Description

Table 13-172 SET ROLE parameters

Parameter	Description	Value Range
SESSION	Specifies that the command takes effect only for the current session. This parameter is used by default.	The value is a string and must comply with the rules in Identifier Naming Conventions .

Parameter	Description	Value Range
LOCALE	Indicates that the specified command takes effect only for the current transaction.	-
role_name	Specifies the role name.	The value is a string and must comply with the rules in Identifier Naming Conventions .
password	Specifies the password of a role. It must comply with the password convention.	-
RESET ROLE	Resets the current user identifier.	-

Examples

Create the role **manager**.

```
CREATE ROLE paul IDENTIFIED BY '{Password}';
```

Set the current user to **paul**.

```
SET ROLE paul PASSWORD '{password}';
```

View the current session user and the current user.

```
SELECT SESSION_USER, CURRENT_USER;
```

Reset the current user.

```
RESET role;
```

13.107 SET SESSION AUTHORIZATION

Function

SET SESSION AUTHORIZATION sets the session user identifier and the current user identifier of the current SQL session to a specified user.

Precautions

The session identifier can be changed only when the initial session user has the system administrator rights. Otherwise, the system supports the command only when the authenticated user name is specified.

Syntax

- SET SESSION AUTHORIZATION** sets the session user identifier and the current user identifier of the current session.

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION role_name PASSWORD '{password}';
```


- Reset the identifiers of the session and current users to the initially authenticated user names.

```
{SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT  
| RESET SESSION AUTHORIZATION};
```

Parameter Description

Table 13-173 SET SESSION AUTHORIZATION parameters

Parameter	Description	Value Range
SESSION	Indicates that the specified parameters take effect for the current session.	-
LOCALE	Indicates that the specified command takes effect only for the current transaction.	-
role_name	Specifies the role name.	An existing role name
password	Specifies the password of a role. It must comply with the password convention.	-
DEFAULT	Reset the identifiers of the session and current users to the initially authenticated user names.	-

Examples

Set the current user to **paul**.

```
SET SESSION AUTHORIZATION paul password '{password};
```

View the current session user and the current user.

```
SELECT SESSION_USER, CURRENT_USER;
```

Reset the current user.

```
RESET SESSION AUTHORIZATION;
```

Helpful Links

[SET ROLE](#)

13.108 SHOW

Function

SHOW shows the current value of a run-time parameter. You can use the **SET** statement to set these parameters.

Precautions

Some parameters that can be viewed by **SHOW** are read-only. You can view but cannot modify their values.

Syntax

```
SHOW
{
  configuration_parameter |
  CURRENT_SCHEMA |
  TIME_ZONE |
  TRANSACTION ISOLATION LEVEL |
  SESSION AUTHORIZATION |
  ALL
};
```

Parameter Description

See [Parameter Description](#) in **RESET**.

Examples

Show the value of **timezone**.

```
SHOW timezone;
```

Show the current setting of the **DateStyle** parameter.

```
SHOW DateStyle;
```

Show the current setting of all parameters.

```
SHOW ALL;
```

Helpful Links

[SET](#), [RESET](#)

13.109 TRUNCATE

Function

TRUNCATE quickly removes all rows from a database table.

It has the same effect as an unqualified **DELETE**, but since it does not actually scan the table, it is faster. This is most effective on large tables.

Differences Among TRUNCATE TABLE, DELETE TABLE, and DROP TABLE

Table 13-174 Differences among TRUNCATE TABLE, DELETE TABLE, and DROP TABLE

Dimension	TRUNCATE TABLE	DELETE TABLE	DROP TABLE
Syntax	DDL	DML	DDL

Dimension	TRUNCATE TABLE	DELETE TABLE	DROP TABLE
Deleted content	All data in the table is deleted, but the table schema is not deleted.	Only the content is deleted and the definition is not deleted.	Content and definition are deleted.
Space release	The space is released.	The space is not released.	The space is released.
Execution Speed	Fast Data is deleted by releasing the data pages used to store table data. Only the page release is recorded in the transaction log. Less system and transaction log resources are used.	Slow Each time a row is deleted, a record is generated for each deleted row in the transaction log.	Fastest
Scenario	You need to quickly delete data from a table while retaining the table schema, and the data volume is large.	You need to delete data based on specific conditions and the data volume is under control.	You need to delete the entire table, including the table schema and data.

Precautions

- Exercise caution when running the **TRUNCATE TABLE** statement. Before running this statement, ensure that the table data can be deleted or has been backed up. After you run the **TRUNCATE TABLE** statement to delete table data, the data cannot be restored.
- The **TRUNCATE** operation on global temporary tables only truncates data of the current session. Data of other sessions is not affected.
- In the storage-compute decoupling architecture, it is not possible to perform the **TRUNCATE** operation on common tables and temporary tables simultaneously.



WARNING

- Avoid performing **ALTER TABLE**, **ALTER TABLE PARTITION**, **DROP PARTITION**, and **TRUNCATE** operations during peak hours to prevent long SQL statements from blocking these operations or SQL services.
 - For more information about development and design specifications, see [Development and Design Proposal](#).
-

Syntax

TRUNCATE empties a table or set of tables.

```
TRUNCATE [ TABLE ] [ ONLY ] { [[database_name.]schema_name.]table_name [ * ] } [, ... ]  
[ CONTINUE IDENTITY ] [ CASCADE | RESTRICT ] ;
```

Parameter Description

Table 13-175 TRUNCATE parameters

Parameter	Description	Value Range
ONLY	Specifies the range of tables to be cleared. This parameter is reserved only for compatibility with PostgreSQL. GaussDB(DWS) does not support inherited tables.	- If ONLY is specified, only the specified table is cleared. - If ONLY is not specified, the specified table and all its inherited tables (if any) are cleared.
database_name	Specifies the name of the database where the table to be cleared is located.	An existing database name.
schema_name	Specifies the schema name of the table to be cleared.	An existing schema name.
table_name	Specifies the name of the table to be cleared (which can be schema-qualified).	An existing table name.
CONTINUE IDENTITY	Does not change the values of sequences.	This parameter is set to the default value.
CASCADE RESTRICT	(Optional) Specifies the method of clearing tables that have foreign key references.	CASCADE: automatically truncates all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to CASCADE. RESTRICT: refuses to truncate if any of the tables have foreign-key references from tables that are not listed in the command. The default value is RESTRICT.

Examples

1. Creates the sample table **test_t** and insert data into it.

```
CREATE TABLE test_t
(
  col_id INT PRIMARY KEY,
  col_number INT NOT NULL,
  col_date DATE NOT NULL,
  col_price NUMERIC (10,2),
  col_status TEXT
)
WITH (ORIENTATION = COLUMN)
DISTRIBUTE BY HASH (col_id);

INSERT INTO test_t VALUES
(01,300,'2025-03-01',99.20,'sold'),
(02,400,'2025-04-01',95.50,'sold'),
(03,450,'2025-05-01',420.50,'sold'),
(04,100,'2025-06-01',100.85,'restock');
```

2. View data in the **test_t** table.

```
SELECT * FROM test_t;
```

col_id	col_number	col_date	col_price	col_status
1	300	2025-03-01 00:00:00	99.20	sold
2	400	2025-04-01 00:00:00	95.50	sold
3	450	2025-05-01 00:00:00	420.50	sold
4	100	2025-06-01 00:00:00	100.85	restock

(4 rows)

3. Clear the **test_t** table.

```
TRUNCATE test_t;
```

If the following information is displayed, the table is cleared successfully.

```
TRUNCATE TABLE
```

4. View the table definition. The TRUNCATE TABLE operation clears the table but retains the table definition.

```
SELECT * FROM pg_get_tabledef('public.test_t');
pg_get_tabledef
```

SET search_path = public;	+
CREATE TABLE test_t (	+
col_id integer NOT NULL,	+
col_number integer NOT NULL,	+
col_date timestamp(0) without time zone NOT NULL,	+
col_price numeric(10,2),	+
col_status text	+
)	+
WITH (orientation=column, compression=middle, colversion=2.0, enable_delta=false)+	+
DISTRIBUTE BY HASH(col_id)	+
;	

(1 row)

13.110 VACUUM

Function

VACUUM reclaims storage space occupied by tables or B-tree indexes. In normal database operation, rows that have been deleted are not physically removed from their table; they remain present until a **VACUUM** is done. Therefore, it is necessary to execute **VACUUM** periodically, especially on frequently-updated tables.

Differences between VACUUM and VACUUM FULL

The **VACUUM** command comes in two types: **VACUUM** and **VACUUM FULL**. **VACUUM** performs a simpler process known as **LAZY VACUUM** compared to **VACUUM FULL**. The table below highlights the key differences between **LAZY VACUUM** and **VACUUM FULL**.

Table 13-176 Differences between VACUUM and VACUUM FULL

Item	VACUUM	VACUUM FULL
Space cleanup	When you delete a record at the end of the table, its space is freed and given back to the operating system. For records elsewhere, their space becomes available for reuse.	Regardless of location, all deleted data eventually frees up physical space for the OS. New data insertion requires allocating fresh disk pages.
Lock type	A Level-4 lock (ShareUpdateExclusive-Lock) allows concurrent operations.	A Level-8 lock (AccessExclusiveLock) locks the table entirely, pausing all other actions during its execution.
Physical space	Not released.	Released.
Transaction ID	Not reclaimed.	Reclaimed.
Resource consumption	The execution requires little resources and can be performed periodically.	The execution requires significant resources. Make sure the database's disk usage nears its limit before starting. Also, carry out this task when there is less data to handle.
Effect	Table performance will be slightly improved.	Table performance is greatly improved.

Auto Cleanup (AUTOVACUUM)

The **VACUUM** operation frees up space, enhancing query performance. DWS offers automatic cleanup, which you can activate under specific conditions as needed. You can configure this on the console by referring to [Managing O&M Plans](#)

Precautions

- With no table specified, **VACUUM** processes all the tables that the current user has permission to vacuum in the current database. With a table specified, **VACUUM** processes only that table.
- To perform **VACUUM** on a table, you must be the owner of the table, granted with the **VACUUM** permission on the table, or granted with the

gs_role_vacuum_any role. By default, a system administrator has this permission. However, database owners are allowed to **VACUUM** all tables in their databases, except shared catalogs. (The restriction for shared catalogs means that a true database-wide **VACUUM** can only be executed by the system administrator). **VACUUM** skips over any tables that the calling user does not have the permission to vacuum.

- **VACUUM** cannot be executed inside a transaction block.
- It is recommended that active production databases be vacuumed frequently (at least nightly), in order to remove dead rows. **VACUUM ANALYZE** executes a **VACUUM** operation and then an **ANALYZE** operation for each selected table. After adding or deleting a large number of rows, it might be a good idea to execute the **VACUUM ANALYZE** command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the optimizer to make better choices in planning queries.
- **VACUUM** causes a substantial increase in I/O traffic, which might cause poor performance for other active sessions. Therefore, it is sometimes advisable to use the cost-based **VACUUM** delay feature.
- When **VERBOSE** is specified, **VACUUM** prints progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.
- When the option list is surrounded by parentheses, the options can be written in any order. If there are no brackets, the options must be given in the order displayed in the syntax.
- During **VACUUM** on column-store tables, four internal operations occur: **VACUUM** on the primary table, **VACUUM** on the Desc table of the primary table, **VACUUM** on the delta table of the primary table, and migration of data from the delta table to the primary table. **VACUUM** does not reclaim the storage space of the delta table. To reclaim it, do **VACUUM DELTAMERGE** to the column-store table. By default, the **VACUUM** primary table operation is enabled. You can disable this function by setting the **colvacuum_threshold_scale_factor** parameter.
- The **VACUUM** operation on column-store primary tables does not support temporary tables, multi-temperature tables, and time series tables.
- This operation does not immediately free up space. To reclaim space right away, execute the **vac_fileclear_relation** function after running **VACUUM**; this will place an exclusive lock on the chosen table.
- Plain **VACUUM** (without **FULL**) recycles space and makes it available for reuse. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. **VACUUM FULL** executes wider processing, including moving rows across blocks to compress tables so they occupy minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.
- **FULL** is recommended only in special scenarios. For example, you wish to physically narrow the table to decrease the occupied disk space after deleting most rows of a table. **VACUUM FULL** usually shrinks more table size than **VACUUM**. If the physical space usage does not decrease after you run the command, check whether there are other active transactions (that have started before you delete data transactions and not ended before you run

VACUUM FULL). If there are such transactions, run this command again when the transactions quit.

- **VACUUM** and **VACUUM FULL** clear deleted tuples after the delay specified by **vacuum_defer_cleanup_age**.
- Running **VACUUM FULL** on system catalogs can only be done when the database is offline. Otherwise, table locks, exceptions, and errors may occur.
- Running **VACUUM FULL** on a column-store partitioned table locks the table and its partitions.
- A distributed deadlock may occur when **VACUUM FULL** and DML statements run concurrently in the following scenarios:
 - **VACUUM FULL** on subpartitions and **INSERT/UPDATE/DELETE** on the primary table
 - **VACUUM FULL** on a full table and **SELECT** on a full table or **SELECT** on subpartitions
- In the storage-compute decoupled architecture, a message is displayed indicating that the full database **VACUUM**, **VACUUM FULL**, **VACUUM ANALYZE**, or **VACUUM DELTAMERGE** is not supported.

**WARNING**

- When a **VACUUM FULL** operation is performed on a table, it triggers table rebuilding. During this rebuilding process, data is dumped into a new data file. Once the rebuilding is complete, the original file is deleted. However, it is important to note that if the table is large, the rebuilding process can consume a significant amount of disk space. When the disk space is insufficient, exercise caution when performing the **VACUUM FULL** operation on large tables to prevent the cluster from being read-only.
- Execute **VACUUM FULL** on tables with a high dirty page rate and small CU ratio exceeding 25% at regular intervals. Perform **VACUUM FULL** on common tables during low-traffic hours and on system tables when they are offline.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

- Reclaim space and update statistics. The keyword sequence must be given in the order in which the syntax is displayed.

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | {ANALYZE | ANALYSE } } [,...] ) ]  
[ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) ];
```
- Reclaim space, without updating statistics information.

```
VACUUM [ FULL [COMPACT] ] [ FREEZE ] [ VERBOSE ] [ table_name ] [ PARTITION  
( partition_name ) ];
```
- Reclaim space and update statistics information, with a specific order of keywords required.

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] { ANALYZE | ANALYSE } [ VERBOSE ]  
[ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) ];
```
- For HDFS and column-store tables, migrate data from the delta table to the primary table. The **partition_name** parameter is supported only by 8.2.1.200 and later cluster versions.

```
VACUUM DELTAMERGE [ table_name ][partition_name];
```


- For HDFS tables, delete the empty value partition directory of HDFS table in HDFS storage.
`VACUUM HDFS DIRECTORY [ table_name ];`

Parameter Description

Table 13-177 VACUUM parameters

Parameter	Description	Value Range
FULL	Selects the parameter, which can reclaim more space, but takes much longer and exclusively locks the table. FULL options can also contain the COMPACT parameter, which is only used for the HDFS table. Specifying the COMPACT parameter improves VACUUM FULL operation performance. NOTICE COMPACT and PARTITION cannot be used at the same time. - Using FULL will cause statistics missing. To collect statistics, add the keyword ANALYZE to VACUUM FULL.	-

Parameter	Description	Value Range
FREEZE	Resolves the transaction ID wraparound issue by default. Using FREEZE sets vacuum_freeze_min_age to 0 during VACUUM. NOTICE Since GaussDB(DWS) employs 64-bit XIDs, the wraparound problem does not occur theoretically. Avoid using VACUUM FREEZE unless necessary.	-
VERBOSE	Prints a detailed vacuum activity report for each table.	-
ANALYZE ANALYSE	Updates statistics used by the optimizer to determine the most efficient way to execute a query.	-
table_name	Indicates the name (optionally schema-qualified) of a specific table to vacuum.	Defaults are all tables in the current database.
column_name	Indicates the name of a specific field to analyze.	Defaults are all columns.

Parameter	Description	Value Range
PARTITION	Specifies that the table to be cleared is a partitioned table. HDFS tables do not support the PARTITION parameter. NOTE If the PARTITION and COMPACT parameters are used at the same time, the following error message is displayed: COMPACT can not be used with PARTITION.	-
partition_name	Indicates the partition name of a specific table to vacuum.	Defaults are all partitions.

Parameter	Description	Value Range
DELTAMERGE	For HDFS tables and column-store tables, migrate data in the delta table of HDFS tables or column-store tables to the primary table. If the data volume of the delta table is less than 60,000 rows, the data will not be migrated. Otherwise, the data will be migrated to HDFS, and the delta table will be cleared by TRUNCATE . For a column-store table, this operation always transfers all data in the delta table to the CU.	-

Parameter	Description	Value Range
	NOTE The following DFX functions are provided to return the data storage in the delta table of a column-store table (for an HDFS table, it can be returned by EXPLAIN ANALYZE): - pgxc_get_delta_info(TEXT): The input parameter is a column-store table name. The delta table information on each node is collected and displayed, including the number of active tuples, table size, and maximum block ID. - get_delta_info(TEXT): The input parameter is a column-store table name. The system summarizes the results returned from pgxc_get_delta_info and returns the total number of active tuples, total table size, and maximum block ID in the delta table. When querying delta information about a temporary table, you need to specify the schema of the temporary table. Otherwise, an error is reported, indicating that the table cannot be found.	

Parameter	Description	Value Range
HDFS DIRECTORY	Deletes the empty value partition directory of HDFS table in HDFS storage for HDFS table.	-

Examples

Delete all tables in the current database.

```
VACUUM;
```

Reclaim the space of partition **P2** of the **tpcds.web_returns_p1** table without updating statistics.

```
VACUUM FULL tpcds.web_returns_p1 PARTITION(P2);
```

Reclaim the space of the **tpcds.web_returns_p1** table and update statistics.

```
VACUUM FULL ANALYZE tpcds.web_returns_p1;
```

Delete all tables in the current database and collect statistics about the query optimizer.

```
VACUUM ANALYZE;
```

Delete only the **reason** table.

```
VACUUM (VERBOSE, ANALYZE) tpcds.reason;
```

Perform the **DELTAMERGE** operation on the column-store table **table_delta**.

```
VACUUM DELTAMERGE tpcds.table_delta;
```

Perform the **DELTAMERGE** operation only on partition **p1** of the column-store table **table_delta**.

```
VACUUM DELTAMERGE tpcds.table_delta partition(p1);
```

14 DML Syntax

14.1 DML Syntax Overview

Data Manipulation Language (DML) is used to perform operations on data in database tables, such as inserting, updating, querying, or deleting data.

Insert Data

Inserting data refers to adding one or multiple records to a database table. For details, see [INSERT](#).

Updating Data

Modifying data refers to modifying one or multiple records in a database table. For details, see [UPDATE](#).

Querying Data

The database query statement **SELECT** is used to search required information in a database. For details, see [SELECT](#).

Deleting Data

For details about how to delete data that meets specified conditions from a table, see [DELETE](#).

Copying Data

DWS provides a statement for copying data between tables and files. For details, see [COPY](#).

Locking a Table

DWS provides multiple lock modes to control concurrent accesses to table data. For details, see [LOCK](#).

Run the following statement to invoke the function:

DWS provides three statements for invoking functions. These statements are the same in the syntax structure. For details, see [CALL](#).

14.2 CALL

Function

CALL calls defined functions or stored procedures.

Precautions

If the name of a user-defined function is the same as that of a system function, you need to specify a schema when invoking the user-defined function. Otherwise, the system preferentially invokes the system function.

Syntax

```
CALL [schema.] {func_name| procedure_name} ( param_expr );
```

Parameter Description

Table 14-1 CALL parameters

Parameter	Description	Value Range
schema	Specifies the name of the schema to which a function or stored procedure belongs.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
func_name	Specifies the name of the function or stored procedure to be called.	Use an existing function name.

Parameter	Description	Value Range
param_expr	Specifies a list of parameters in the function. You can use ":@" or "=>" to separate a parameter name from its value. By doing so, parameters can be arranged in any order. If the parameter list contains only parameter values, the sequence of the parameter values must be the same as that defined in the function or stored procedure. NOTE The parameters include input parameters (whose name and type are separated by IN) and output parameters (whose name and type are separated by OUT). When you run the CALL statement to call a function or stored procedure, the parameter list must contain an output parameter for non-overloaded functions. You can set the output parameter to a variable or any constant. For details, see Examples. For an overloaded package function, the parameter list can have no output parameter, but the function may not be found. If an output parameter is contained, it must be a constant.	Names of existing function parameters or stored procedure parameters.

Examples

Create the **func_add_sql** function to compute the sum of two integers and return the result:

```
CREATE FUNCTION func_add_sql(num1 integer, num2 integer) RETURN integer
AS
BEGIN
RETURN num1 + num2;
END;
/
```

Transfer based on parameter values:

```
CALL func_add_sql(1, 3);
```

Transfer based on the naming flags:

```
CALL func_add_sql(num1 => 1, num2 => 3);
CALL func_add_sql(num2 := 2, num1 := 3);
```

Delete a function:

```
DROP FUNCTION func_add_sql;
```

Create a function with output parameters:

```
CREATE FUNCTION func_increment_sql(num1 IN integer, num2 IN integer, res OUT integer)
RETURN integer
AS
BEGIN
res := num1 + num2;
END;
/
```

Set output parameters to constants:

```
CALL func_increment_sql(1,2,1);
```

Set output parameters to variables:

```
DECLARE
res int;
BEGIN
func_increment_sql(1, 2, res);
dbms_output.put_line(res);
END;
/
```

Create overloaded functions:

```
create or replace procedure package_func_overload(col int, col2 out int) package
as
declare
col_type text;
begin
col := 122;
dbms_output.put_line('two out parameters ' || col2);
end;
/
create or replace procedure package_func_overload(col int, col2 out varchar)
package
as
declare
col_type text;
begin
col2 := '122';
dbms_output.put_line('two varchar parameters ' || col2);
end;
/
```

Call a function:

```
call package_func_overload(1, 'test');
call package_func_overload(1, 1);
```

Delete a function:

```
DROP FUNCTION func_increment_sql;
```

14.3 COPY

Function

COPY copies data between tables and files. **COPY FROM** copies data from a file to a table. **COPY TO** copies data from a table to a file.

Use **COPY FROM** and **COPY TO** for limited local data transfers with low concurrency.

NOTICE

- Since GaussDB(DWS) is a fully managed cloud service, direct access to backend nodes is not available, making the **COPY** syntax unusable in this environment. Instead, employ the **\COPY** command via the gsql client; note that other tools like SQL editors do not support it.
- For better results, save your data files on OBS and use OBS foreign tables for importing into the database.

Precautions

- **COPY** applies to only tables and does not apply to views.
- To insert data to a table, you must have the permission to insert data.
- If a list of columns is specified, **COPY** will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, **COPY FROM** will insert the default values for those columns.
- If a source data file is specified, the file must be accessible from the server. If **STDIN** is specified, data is transmitted between the client and server. Separate columns by pressing **Tab**. Enter **\.** in a new line to indicate the end of input.
- If the number of columns in a row of the data file is smaller or larger than the expected number, **COPY FROM** displays an error message.
- A backslash and a period (**\.**) indicate the end of data. The end identifier is not required for reading data from a file and is required for copying data between client applications.
- In **COPY FROM**, **"\N"** indicates an empty character string, and **"\\N"** indicates the actual data **"\N"**.
- **COPY FROM** does not support pre-processing of data during data import, for example, expression calculation and default value filling. If you need to perform pre-processing during data import, you need to import the data to a temporary table, and then run SQL statements to insert data to the table using expression or function operations. However, this method may cause I/O expansion, deteriorating data import performance.
- Transactions will be rolled back when data format errors occur during **COPY FROM** execution. In this case, error information is insufficient so you cannot easily locate the incorrect data from a large amount of raw data.
- You can use **COPY TO** to export data in TEXT or CSV format to OBS, but cannot use **COPY FROM** to import data from OBS. For CSV files that are exported to OBS and contain utf8 BOM, you are advised to use OBS to import them. Otherwise, the BOM field may fail to be identified.

Syntax

- Copy the data from a file to a table:

```
COPY table_name [ ( column_name [, ...] ) ]  
FROM { 'filename' | STDIN }  
[ [ USING ] DELIMITERS 'delimiters' ]  
[ WITHOUT ESCAPING ]  
[ LOG ERRORS ]  
[ LOG ERRORS data ]  
[ REJECT LIMIT 'limit' ]  
[ [ WITH ] ( option [, ...] ) ]  
| copy_option  
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option  
[ ...] ]];
```

NOTE

In the SQL syntax, **FIXED FORMATTER ({ column_name(offset, length) } [, ...])**, and **[(option [, ...]) | copy_option [...]]** can be in any sequence.

- Copy the data from a table to a file:

```
COPY table_name [ ( column_name [, ...] ) ]  
TO { 'filename' | STDOUT }  
[ [ USING ] DELIMITERS 'delimiters' ]
```

```
[ WITHOUT ESCAPING ]
[ [ WITH ] ( option [, ...] ) ]
| copy_option
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[ ...] ]];

COPY query
TO { 'filename' | STDOUT }
[ WITHOUT ESCAPING ]
[ [ WITH ] ( option [, ...] ) ]
| copy_option
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[ ...] ]];
```

NOTE

- The syntax constraints of **COPY TO** are as follows:
 - (**query**) is incompatible with **[USING] DELIMITER**. If the data of **COPY TO** comes from a query result, **COPY TO** cannot specify **[USING] DELIMITERS**.
- Use spaces to separate **copy_option** following **FIXED FORMATTER**.
- **copy_option** is the native parameter, while **option** is the parameter imported by a compatible foreign table.
- In the SQL syntax, **FIXED, FORMATTER ({ column_name(offset, length) } [, ...])**, and **[(option [, ...]) | copy_option [...]]** can be in any sequence.

Parameter Description

Table 14-2 COPY parameters

Parameter	Description	Value Range
query	Copies the query result.	A SELECT or VALUES command in parentheses.
table_name	Specifies the name (optionally schema-qualified) of an existing table.	An existing table name.
column_name	Specifies the list of optional fields to be copied.	If the column list is not specified, all columns in the table will be copied.
STDIN	Indicates that the input comes from the client application.	-
STDOUT	Specifies that output goes to the client application.	-

Parameter	Description	Value Range
FIXED	Fixes column length. - When the column length is fixed, DELIMITER, NULL, and CSV cannot be specified. - When FIXED is specified, BINARY, CSV, and TEXT cannot be specified by option or copy_option.	The definition of fixed length: - The column length of each record is the same. - Spaces are added to short columns. Digit type columns must be left-aligned, and character columns must be right-aligned. - No delimiters are used between columns.
[USING] DELIMITER 'delimiters'	The string that separates columns within each row (line) of the file, and it cannot be larger than 10 bytes.	The delimiter cannot include any of the following characters: \\abcdefghijklmnopqrstuvwxyz0123456789 The default value is a tab character in text format and a comma in CSV format.
WITHOUT ESCAPING	Specifies that the backslash (\\) and its following characters are not escaped in text format.	Only the text format is supported.

Parameter	Description	Value Range
LOG ERRORS	If this parameter is specified, the error tolerance mechanism for data type errors in the COPY FROM statement is enabled. Row errors are recorded in the public.pgxc_copy_error_log table in the database for future reference. NOTICE The restrictions of this error tolerance parameter are as follows: • This error tolerance mechanism captures only the data type errors (DATA_EXCEPTION) that occur during data parsing of COPY FROM on a CN. Other errors, such as network errors between CNs and DN or expression conversion errors on DNs, are not captured. • Before enabling error tolerance for COPY FROM for the first time in a database, check whether the public.pgxc_copy_error_log table exists. If it does not, call the <code>copy_error_log_create()</code> function to create it. If it does, copy its data elsewhere and call the <code>copy_error_log_create()</code> function to create the table. For details about columns in the public.pgxc_copy_error_log table, see Table 7-30. • While a COPY FROM statement with specified LOG ERRORS is being executed, if public.pgxc_copy_error_log does not exist or does not have the table definitions compliant with the predefined in <code>copy_error_log_create()</code>, an error will be reported. Ensure that the error table is created using the <code>copy_error_log_create()</code> function. Otherwise, COPY FROM statements with error tolerance may fail to be run. • If existing error tolerance parameters (for example, IGNORE_EXTRA_DATA) of the COPY statement are enabled, the error of the corresponding type will be processed as specified by the parameters and no error will be reported. Therefore, the error table does not contain such error data. • The coverage scope of this error tolerance mechanism is the same as that of a GDS foreign table. You are advised to filter query results based on table names or the timestamp of marking the start of COPY FROM statement execution. For how to process error data, see the section about handling error tables.	A value set while data is imported using COPY FROM .

Parameter	Description	Value Range
LOG ERRORS DATA	The differences between LOG ERRORS DATA and LOG ERRORS are as follows: • LOG ERRORS DATA records error row data (rawrecord). • When this parameter is used, if the error content is too complex, the error may fail to be written into the fault tolerance table. As a result, the task fails.	-
REJECT LIMIT 'limit'	Used with the LOG ERROR parameter to set the upper limit of the tolerated errors in the COPY FROM statement. If the number of errors exceeds the limit, later errors will be reported based on the original mechanism. NOTE Different from the GDS error tolerance mechanism, in the error tolerance mechanism described in the description of LOG ERRORS , the count of REJECT LIMIT is calculated based on the number of data parsing errors on the CN where the COPY FROM statement is run, not based on the number of errors on each DN.	A positive integer (1 to INTMAX) or unlimited Default value: • If LOG ERRORS is not specified, an error is reported. • If LOG ERRORS is specified, the default value is 0.
FORMATTER	Defines the position of each column in the data file in the column(offset,length) format when the fixed length mode is used (that is, the FIXED parameter is specified).	• The value of offset must be larger than 0. The unit is byte. • The value of length must be larger than 0. The unit is byte. • The total length of all columns must be less than 1 GB. • Replace columns that are not in the file with NULL.
OPTION { option_name 'value ' }	Specifies all types of parameters of a compatible foreign table.	For details, see Table 14-3 .

Parameter	Description	Value Range
COPY_OPTION { option_name ' value ' }	Specifies all types of native parameters of COPY .	For details, see Table 14-4 .

Table 14-3 OPTION parameters

Parameter	Description	Value Range
FORMAT	Specifies the format of the source data file in the foreign table.	CSV, text, fixed, and binary files are supported. • The CSV file can process newline characters efficiently, but cannot process certain special characters well. • The TEXT file can process special characters efficiently, but cannot process newline character well. • The FIXED file can process newline characters in data columns efficiently, but cannot process special characters well. • All data in the BINARY file is stored/read as binary format rather than as text. It is faster than the text and CSV formats, but a binary-format file is less portable. Default value: TEXT
OIDS	Copies the OID for each row. NOTE An error is raised if OIDs are specified for a table that does not have OIDs, or in the case of copying a query.	• true/on • false/off Default value: false

Parameter	Description	Value Range
DELIMITER	Specifies the character that separates columns within each row (line) of the file. • A delimiter cannot be \r or \n. • A delimiter cannot be the same as null. The delimiter for CSV cannot be same as quote. • The delimiter for the TEXT format data cannot contain lowercase letters, digits, or dot (.). • The data length of a single row should be less than 1 GB. If the delimiters are too long and there are too many rows, the length of valid data will be affected. • You are advised to use multi-characters and invisible characters for delimiters. For example, you can use multi-characters (such as \$^&) and invisible characters (such as 0x07, 0x08, and 0x1b). • For a multi-character delimiter, do not use the same characters, for example, ---. NOTE	The value of delimiter can be a multi-character delimiter whose length is less than or equal to 10 bytes. Default value: • A tab character in TEXT format • A comma (,) in CSV format • No delimiter in FIXED format
NULL	Specifies how to represent a null value.	• The null value cannot be \r or \n. The maximum length is 100 characters. • The null value cannot be the same as the delimiter or quote parameter. Default value: • An empty string without quotation marks in CSV format • \N in TEXT format

Parameter	Description	Value Range
HEADER	Specifies whether a file contains a header with the names of each column in the file. header is available only for CSV and FIXED files. - When data is imported, if header is on, the first row of the data file will be identified as title row and ignored. If header is off, the first row is identified as data. - When data is exported, if header is on, fileheader must be specified. If header is off, the exported file does not include a title row.	- true/on - false/off Default value: false
QUOTE	Specifies the quote character used when a data value is referenced in a CSV file.	Default value: double quotation mark ("). - The quote parameter cannot be the same as the delimiter or null parameter. - The quote parameter must be a single one-byte character. - Invisible characters are recommended as quote values, such as 0x07, 0x08, and 0x1b.
ESCAPE	Specifies the escape character in CSV format. The escape character can only be a single-byte character.	Single-byte character. Default value: double quotation mark (")

Parameter	Description	Value Range
EOL 'newline_character'	Specifies the newline character style of the imported or exported data file.	Multi-character newline characters are supported, but the newline character cannot exceed 10 bytes. Common newline characters include <code>\r</code> (0x0D), <code>\n</code> (0x0A), and <code>\r\n</code> (0x0D0A). Special newline characters include <code>\$</code> and <code>#</code>. • The EOL parameter supports only the TEXT format for data import and export and does not support the CSV or FIXED format for data import. For forward compatibility, the EOL parameter can be set to 0x0D or 0x0D0A for data export in the CSV and FIXED formats. • The value of the EOL parameter cannot be the same as that of DELIMITER or NULL. • The EOL parameter value cannot contain lowercase letters, digits, or dot (.).
FORCE_QUOTE { (column_name [, ...]) * }	Forces quotation marks to be used for all non-null values in each specified column in CSV COPY TO mode. NULL values are not quoted.	Existing fields.
FORCE_NOT_NULL (column_name [, ...])	Assigns a value to a specified column in CSV COPY FROM mode.	Existing fields.

Parameter	Description	Value Range
ENCODING	Specifies the encoding format of a data file.	The default value is the current database encoding format. Common encoding formats include UTF8, GBK, and GB18030. GB18030 has two versions: GB18030 and GB18030_2022. GB18030_2022 is the latest national standard in China prepared to support Chinese characters.
IGNORE_EXT RA_DATA	When the number of data source files exceeds the number of foreign table columns, whether ignoring excessive columns at the end of the row. This parameter is available only during data importing.	- When this parameter is true or on and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored. - If the parameter is set to false or off, and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed: extra data after last expected column Default value: false NOTICE If the newline character at the end of the row is lost, setting the parameter to true will ignore data in the next row.

Parameter	Description	Value Range
COMPATIBLE_ILLEGAL_CHARS	Specifies the error tolerance parameter for invalid characters. This parameter is valid only for COPY FROM import. This parameter does not support the binary format. The "cannot specify bulkload compatibility options in BINARY mode" error will be reported. The rule of error tolerance when you import invalid characters is as follows: • \0 is converted to a space. • Other invalid characters are converted to question marks. • Setting compatible_illegal_chars to true/on enables toleration of invalid characters. If NULL, DELIMITER, QUOTE, and ESCAPE are set to spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt the user to modify parameters that may cause confusion, preventing importing errors.	• When the parameter is true or on, invalid characters are tolerated and imported to the database after conversion. • If the parameter is false or off, and an error occurs when there are invalid characters, the import will be interrupted. Default value: false or off
FILL_MISSING_FIELDS	Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading.	• true/on: The value of the last field is set to NULL and no error message is reported. • false/off: The following error message is reported: missing data for column "tt" Default value: false

Parameter	Description	Value Range
DATE_FORMAT AT	Imports data of the DATE type. It is valid only for COPY FROM import. This parameter does not support the binary format. The "cannot specify bulkload compatibility options in BINARY mode" error will be reported.	The value must be in a valid DATE format. For details, see Date and Time Processing Functions and Operators . NOTICE In databases configured for Oracle compatibility (with DBCOMPATIBILITY set to ORA during table creation), the DATE type functions as TIMESTAMP. For more information about the syntax compatibility differences, see Syntax Compatibility Differences Among Oracle, Teradata, and MySQL . For details, see timestamp_format below.
TIME_FORMAT AT	Imports data of the TIME type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid TIME formats. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
TIMESTAMP_FORMAT	Imports data of the TIMESTAMP type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid TIMESTAMP formats. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
SMALLDATE TIME_FORMAT AT	Imports data of the SMALLDATETIME type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid SMALLDATETIME formats. For details, see Date and Time Processing Functions and Operators .

Parameter	Description	Value Range
SERVER	Specifies the OBS server, which takes effect only when COPY TO is used. filename is a path on OBS, indicating that data is exported to OBS.	Name of the created OBS server.
BOM	Specifies whether to add the UTF-8 BOM field to the exported CSV file. This parameter is valid only when COPY TO is executed and the SERVER parameter is set to a valid value. The exported file must be encoded in UTF-8 format.	• true/on • false/off Default value: false
MAXROW	Specifies the maximum number of lines in an exported file. A new file is generated if the number of lines exceeds the maximum. This parameter is valid only when COPY TO is executed and the SERVER parameter is set to a valid value. When HEADER is set to true , MAXROW must be greater than 1. It must be specified together with FILEPREFIX .	1–2147483647
FILEPREFIX	Specifies the prefix of an export file name. This parameter is valid only when COPY TO is executed and the SERVER parameter is set to a valid value. It must be specified together with MAXROW .	A valid string that does not start or end with a slash (/).
PRESERVE_BLANKS	Specifies whether to retain the blank characters (including spaces, \t, \v, and \f) at the end of each column during fixed-length import. This parameter is supported only by clusters of 8.2.0.100 or later.	• true/on: indicates that the trailing \t is retained and the column data of the BYTEAOID, CHAROID, NAMEOID, TEXTOID, BPCHAROID, VARCHAROID, NVARCHAR2OID and CSTRINGOID types is not pruned. • false/off: prunes trailing white space characters (including spaces, \t, \v, and \f) of all data types. Default value: false/off .

Table 14-4 COPY_OPTION parameters

Parameter	Description	Value Range
OIDS	Copies the OID for each row. An error is raised if OIDs are specified for a table that does not have OIDs, or in the case of copying a query.	-
NULL null_string	Specifies the string that represents a null value. NOTICE When using COPY FROM , any data item that matches this string will be stored as a NULL value, so you should make sure that you use the same string as you used with COPY TO .	• The null value cannot be \r or \n. The maximum length is 100 characters. • The null value cannot be the same as the delimiter or quote parameter. Default value: • \N in TEXT format • An empty string without quotation marks in CSV format
HEADER	Specifies whether a file contains a header with the names of each column in the file. header is available only for CSV and FIXED files.	• When data is imported, if header is on, the first row of the data file will be identified as title row and ignored. If header is off, the first row is identified as data. • When data is exported, if header is on, fileheader must be specified. If header is off, the exported file does not include a title row.

Parameter	Description	Value Range
FILEHEADER	Specifies a file that defines the content in the header for exported data. The file contains data description of each column. NOTICE • This parameter is available only when header is on or true. • fileheader specifies an absolute path. • The file can contain only one row of header information, and ends with a linefeed. Excess rows will be discarded. (Header information cannot contain linefeeds.) • The length of the file including the linefeed cannot exceed 1 MB.	-
FREEZE	Sets the COPY loaded data row as frozen, like these data have executed VACUUM FREEZE. This is a performance option of initial data loading. The data will be frozen only when the following three requirements are met: • The table being loaded has been created or truncated in the current subtransaction before copying. • There are no cursors open in the current transaction. • There are no original snapshots in the current transaction. NOTICE When COPY is completed, all other sessions will immediately recognize the data. This violates the general principle of MVCC visibility and leads to potential risks.	-
FORCE NOT NULL column_name [, ...]	Assigns a value to a specified column in CSV COPY FROM mode.	Existing fields.
FORCE QUOTE { column_name [, ...] * }	Forces quotation marks to be used for all non-null values in each specified column in CSV COPY TO mode. NULL values are not enclosed in quotation marks.	Existing fields.

Parameter	Description	Value Range
BINARY	The binary format option causes all data to be stored/read as binary format rather than as text. In binary mode, you cannot declare DELIMITER , NULL , or CSV . After the binary type is specified, you cannot use option or copy_option to specify the CSV, fixed, and text types.	-
CSV	Enables the CSV mode. After the CSV type is specified, you cannot use option or copy_option to specify the binary, fixed, or text type.	-
QUOTE [AS] 'quote_character'	Specifies the quote character for a CSV file. • The quote parameter cannot be the same as the delimiter or null parameter. • The quote parameter must be a single one-byte character. • Invisible characters are recommended as quote values, such as 0x07, 0x08, and 0x1b.	Default value: double quotation mark (")
ESCAPE [AS] 'escape_character'	Specifies an escape character for a CSV file. The value must be a single-byte character.	The default value is a double quotation mark ("). If it is the same as the value of quote , it will be replaced with \0.

Parameter	Description	Value Range
EOL 'newline_character'	Specifies the newline character style of the imported or exported data file.	Multi-character newline characters are supported, but the newline character cannot exceed 10 bytes. Common newline characters include <code>\r</code> (0x0D), <code>\n</code> (0x0A), and <code>\r\n</code> (0x0D0A). Special newline characters include <code>\$</code> and <code>#</code>. • The EOL parameter supports only the TEXT format for data import and export. For forward compatibility, the EOL parameter can be set to 0x0D or 0x0D0A for data export in the CSV and FIXED formats. • The value of the EOL parameter cannot be the same as that of DELIMITER or NULL. • The EOL parameter value cannot contain lowercase letters, digits, or dot (.).
ENCODING 'encoding_name'	Specifies the name of a file encoding format.	Valid encoding formats. Default value: current encoding format.
IGNORE_EXT RA_DATA	When the number of data source files exceeds the number of foreign table columns, excess columns at the end of the row are ignored. This parameter is available only during data importing. If you do not use this parameter, and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed: extra data after last expected column	-

Parameter	Description	Value Range
COMPATIBLE_ILLEGAL_CHARS	Specifies error tolerance for invalid characters during importing. Invalid characters are converted before importing. No error is reported and the import is not interrupted. The binary format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. It is valid only for COPY FROM import. If this parameter is not used, an error is reported when an invalid character is encountered during the import, and the import is interrupted. The rule of error tolerance when you import invalid characters is as follows: • \0 is converted to a space. • Other invalid characters are converted to question marks. • Setting compatible_illegal_chars to true/on enables toleration of invalid characters. If NULL, DELIMITER, QUOTE, and ESCAPE are set to spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt the user to modify parameters that may cause confusion, preventing importing errors.	-
FILL_MISSING_FIELDS	Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading. NOTICE Do not specify this option. If this option is enabled, the parser will ignore the data problem during CN data parsing (that is, the fault tolerance range of the COPY error table) and report an error again on the DN. As a result, the COPY error table (with LOG ERRORS REJECT LIMIT enabled) cannot capture such data exceptions.	• true/on • false/off Default value: false or off

Parameter	Description	Value Range
DATE_FORM AT 'date_format_string'	Imports data of the DATE type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. This parameter is valid only for COPY FROM import. NOTE In databases configured for Oracle compatibility (with DBCOMPATIBILITY set to ORA during table creation), the DATE type functions as TIMESTAMP. For more information about the syntax compatibility differences, see Syntax Compatibility Differences Among Oracle, Teradata, and MySQL. For details, see timestamp_format below.	Valid DATE formats. For details, see Date and Time Processing Functions and Operators .
TIME_FORM AT 'time_format_string'	Imports data of the TIME type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid TIME formats. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
TIMESTAMP_FORMAT 'timestamp_format_string'	Imports data of the TIMESTAMP type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid TIMESTAMP formats. Time zones are not supported. For details, see Date and Time Processing Functions and Operators .
SMALLDATE TIME_FORM AT 'smalldatetime_format_string'	Imports data of the SMALLDATETIME type. This parameter is valid only for COPY FROM import. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur.	Valid SMALLDATETIME formats. For details, see Date and Time Processing Functions and Operators .

Parameter	Description	Value Range
PRESERVE_BLANKS	Specifies whether to retain the blank characters (including spaces, \t, \v, and \f) at the end of each column during fixed-length import. This parameter is supported by clusters of version 8.2.0.100 or later.	• true/on: indicates that the trailing \t is retained and the column data of the BYTEAOID, CHAROID, NAMEOID, TEXTOID, BPCHAROID, VARCHAROID, NVARCHAR2OID and CSTRINGOID types is not pruned. • false/off: prunes trailing white space characters (including spaces, \t, \v, and \f) of all data types. Default value: false/off .

The following special backslash sequences are recognized by **COPY FROM**:

Table 14-5 Special backslash sequences

Backslash Sequence	Description
\b	Backslash (ASCII 8)
\f	Form feed (ASCII 12)
\n	Newline character (ASCII 10)
\r	Carriage return character (ASCII 13)
\t	Tab (ASCII 9)
\v	Vertical tab (ASCII 11)
\digits	Backslash followed by one to three octal digits specifies the ASCII value is the character with that numeric code.
\xdigits	Backslash followed by an x and one or two hex digits specifies the character with that numeric code.

Example (supported only by the gsql client)

Copy data from the **tpcds.ship_mode** file to the **/home/omm/ds_ship_mode.dat** file:

```
\COPY tpcds.ship_mode TO '/home/omm/ds_ship_mode.dat';
```

Copy the query result from the **tpcds.ship_mode** file to the **/home/omm/ds_ship_mode.dat** file.

```
\COPY (SELECT * FROM tpcds.ship_mode WHERE a <> null) TO '/home/omm/ds_ship_mode.dat';
```

Write **tpcds.ship_mode** as output to **stdout**:

```
\COPY tpcds.ship_mode TO stdout;
```

Create the **tpcds.ship_mode_t1** table:

```
CREATE TABLE tpcds.ship_mode_t1
(
  SM_SHIP_MODE_SK      INTEGER      NOT NULL,
  SM_SHIP_MODE_ID      CHAR(16)     NOT NULL,
  SM_TYPE              CHAR(30)      ,
  SM_CODE              CHAR(10)      ,
  SM_CARRIER          CHAR(20)      ,
  SM_CONTRACT          CHAR(20)
)
WITH (ORIENTATION = COLUMN, COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

Copy data from **stdin** to the **tpcds.ship_mode_t1** table:

```
\COPY tpcds.ship_mode_t1 FROM stdin;
```

Copy data from the **/home/omm/ds_ship_mode.dat** file to the **tpcds.ship_mode_t1** table:

```
\COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat';
```

Copy data from the **/home/omm/ds_ship_mode.dat** file to the **tpcds.ship_mode_t1** table, with the import format set to TEXT (**format 'text'**), the delimiter set to **\t** (delimiter **E'\t'**), excessive columns ignored (**ignore_extra_data 'true'**), and characters not escaped (**noescaping 'true'**).

```
\COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' WITH(format 'text', delimiter E'\t',
ignore_extra_data 'true', noescaping 'true');
```

Copy data from the **/home/omm/ds_ship_mode.dat** file to the **tpcds.ship_mode_t1** table, with the import format set to FIXED, fixed-length format specified (**FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20), SM_CONTRACT(82,20))**), excessive columns ignored (**ignore_extra_data**), and headers included (**header**).

```
\COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' FIXED
FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10),
SM_CARRIER(61,20), SM_CONTRACT(82,20)) header ignore_extra_data;
```

Copy data from the **/home/omm/ds_ship_mode.dat** file to the **tpcds.ship_mode_t1** table, with the import format set to FIXED, fixed-length format specified (**FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20), SM_CONTRACT(82,20))**), excessive columns ignored (**ignore_extra_data**), headers included (**header**), and trailing **\t** retained.

```
\COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' (FORMAT 'fixed', FORMATTER
```

```
(SM_SHIP_MODE_SK(0,2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10),  
SM_CARRIER(61,20), SM_CONTRACT(82,20)), PRESERVE_BLANKS'true', HEADER 'true',  
IGNORE_EXTRA_DATA 'true');
```

Export **tpcds.ship_mode_t1** as a text file **ds_ship_mode.dat** in the OBS directory /**bucket/path/**. You need to specify the server options that contain OBS access information:

```
COPY tpcds.ship_mode_t1 TO '/bucket/path/ds_ship_mode.dat' WITH (format 'text', encoding 'utf8', server  
'obs_server');
```

Export **tpcds.ship_mode_t1** as a CSV file in the OBS directory /**bucket/path/**. You need to specify the server options that contain OBS access information. The file contains the title line and BOM header. A single file can contain a maximum of 1000 lines. If the number of lines exceeds 1000, a new file is generated. The user-defined file name prefix is **justprefix**:

```
COPY (select * from tpcds.ship_mode_t1 where SM_SHIP_MODE_SK=1060) TO '/bucket/path/' WITH (format  
'csv', header 'on', encoding 'utf8', server 'obs_server', bom 'on', maxrow '1000', fileprefix 'justprefix');
```

Delete the **tpcds.ship_mode_t1** table:

```
DROP TABLE tpcds.ship_mode_t1;
```

FAQs

"ERROR: invalid byte sequence for encoding 'UTF8': 0x00" Is Reported When Data Is Imported to GaussDB(DWS) Using COPY FROM

14.4 DELETE

Function

Deletes rows that satisfy the **WHERE** clause from the specified table. If the **WHERE** clause does not exist, all rows in the table will be deleted. The result is a valid, but an empty table.

Precautions

- You must have the **DELETE** permission on the table to delete from it, as well as the **SELECT** permission for any table in the **USING** clause or whose values are read in the **condition**.
- For replication tables, **DELETE** can be performed only in the following two scenarios:
 - Scenarios with primary key constraints.
 - Scenario where the execution plan can be pushed down.
- For column-store tables, the **RETURNING** clause is currently not supported.



WARNING

- Avoid using **UPDATE** or **DELETE** to modify or delete a large volume of data and use **TRUNCATE PARTITION** or **DROP PARTITION** instead.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]  
DELETE [/*+ plan_hint */] FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]  
[ PARTITION ( partition_name ) | PARTITION FOR ( partition_key_value [, ...] ) ]  
[ USING using_list ]  
[ WHERE condition | WHERE CURRENT OF cursor_name ]  
[ RETURNING { * | { output_expr [ [ AS ] output_name ] } [, ...] } ];
```

Parameter Description

Table 14-6 DELETE parameters

Parameter	Description	Value Range
WITH [RECURSIVE] with_query [, ...]	The WITH clause allows you to specify one or more subqueries that can be referenced by name in the primary query, equal to temporary table. If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. The format of with_query is as follows: with_query_name [(column_name [, ...])] AS ({select values insert update delete}) -- with_query_name specifies the name of the result set generated by a subquery. Such names can be used to access the result sets of column_name specifies the column name displayed in the subquery result set. Each subquery can be a SELECT, VALUES, INSERT, UPDATE or DELETE statement.	-
plan_hint clause	Following the keyword in the /*+ */ format, hints are used to optimize the plan generated by a specified statement block. For details, see Hint-based Tuning .	-
ONLY	If ONLY is specified, only the specified table is deleted. If ONLY is not specified, the specified table and all its inherited tables are deleted. This parameter is reserved only for compatibility with PostgreSQL. GaussDB(DWS) does not support inherited tables.	-
table_name	Specifies the name (optionally schema-qualified) of a target table.	An existing table name.

Parameter	Description	Value Range
alias	Specifies the alias for the target table.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
partition_name	Specifies the name of a partition. Only clusters of version 8.2.1 or later support this option.	An existing partition name.
partition_key_value	Specifies the key value of a partition. The value specified by PARTITION FOR (partition_key_value [, ...]) can uniquely identify a partition.	Value range of the partition key for the partition to be renamed.
using_list	Specifies the USING clause.	-
condition	Specifies an expression that returns a value of type boolean. Only rows for which this expression returns true will be deleted.	-
WHERE CURRENT OF cursor_name	Not supported currently. Only syntax interface is provided.	-
output_expr	Specifies an expression to be computed and returned by the DELETE command after each row is deleted. The expression can use any column names of the table. Write * to return all columns.	-
output_name	Specifies a name to use for a returned column.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Examples

Create the **tpcds.customer_address_bak** table:

```
CREATE TABLE tpcds.customer_address_bak AS TABLE tpcds.customer_address;
```

Delete employees whose **ca_address_sk** is less than **14888** in the **tpcds.customer_address_bak** table:

```
DELETE FROM tpcds.customer_address_bak WHERE ca_address_sk < 14888;
```

Delete the employees whose **ca_address_sk** is **14891**, **14893**, and **14895** from **tpcds.customer_address_bak**:

```
DELETE FROM tpcds.customer_address_bak WHERE ca_address_sk in (14891,14893,14895);
```

Delete all data in the **tpcds.customer_address_bak** table:

```
DELETE FROM tpcds.customer_address_bak;
```

Use a subquery (to delete the row-store table **tpcds.warehouse_t30**) to obtain a temporary table **temp_t**, and then query all data in the temporary table **temp_t**:

```
WITH temp_t AS (DELETE FROM tpcds.warehouse_t30 RETURNING *) SELECT * FROM temp_t ORDER BY 1;
```

Delete partition **p1** from the partitioned table **test_range_row**:

```
CREATE TABLE test_range_row(a int, d int)
DISTRIBUTE BY hash(a) PARTITION BY RANGE(d)
(
    PARTITION p1 values LESS THAN (60),
    PARTITION p2 values LESS THAN (75),
    PARTITION p3 values LESS THAN (90),
    PARTITION p4 VALUES LESS THAN (maxvalue)
);
INSERT OVERWRITE INTO test_range_row PARTITION(p1) VALUES(55,51);
INSERT OVERWRITE INTO test_range_row PARTITION(p3) VALUES(85,80);

DELETE FROM test_range_row PARTITION(p1);
```

14.5 EXPLAIN

Function

EXPLAIN outputs the execution plan of SQL statements. You can analyze the execution plan to locate slow SQL statements and optimize SQL statements.

The execution plan shows how the tables referenced by the SQL statement will be scanned, for example, by plain sequential scan or index scan. If multiple tables are referenced, the execution plan also shows what join algorithms will be used to bring together the required rows from each input table. The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement.

The **ANALYZE** option (**EXPLAIN ANALYZE**) causes the statement to be actually executed, not only planned. The actual execution time of each step is displayed. This helps you verify the accuracy of the execution plan and evaluate the actual performance.

For details about SQL tuning, see [SQL Tuning](#).

Syntax

- Display the execution plan of an SQL statement, which supports multiple options and has no requirements for the order of options.

```
EXPLAIN [ ( option [, ...] ) ] statement;
```

The syntax of the **option** clause is as follows:

```
ANALYZE [ boolean ] |
ANALYZE [ boolean ] |
VERBOSE [ boolean ] |
COSTS [ boolean ] |
```

```
CPU [ boolean ] |  
DETAIL [ boolean ] |  
NODES [ boolean ] |  
NUM_NODES [ boolean ] |  
BUFFERS [ boolean ] |  
TIMING [ boolean ] |  
PLAN [ boolean ] |  
FORMAT { TEXT | XML | JSON | YAML } |  
BLOCKNAME [boolean] |  
OUTLINE [boolean] |  
WARMUP |  
WARMUP HOT
```

- Display the execution plan of an SQL statement, where options are in order.
EXPLAIN { [{ ANALYZE | ANALYSE }] [VERBOSE] | PERFORMANCE } statement;
- Display information required for reproducing the execution plan of an SQL statement. The information is usually used for fault locating. The **STATS** option must be used independently.
EXPLAIN (STATS [boolean]) statement;
- Display the detailed time consumption of the DDL statement execution. This syntax is supported only by clusters of version 9.1.0 or later.
EXPLAIN PERFORMANCE statement;

DDL statement such as CREATE, CREATE INDEX, DROP, VACUUM FULL, ANALYZE, and COPY are supported.
- Perform pre-query and cache the pre-queried data to the local disk for higher query speed. This syntax is supported only by clusters of 9.1.0.200 and later versions.
EXPLAIN WARMUP statement;
EXPLAIN WARMUP HOT statement;

Parameter Description

Table 14-7 EXPLAIN parameters

Parameter	Description	Value Range
statement	Specifies the SQL statement to be analyzed.	-

Parameter	Description	Value Range
ANALYZE boolean ANALYZE boolean	The ANALYZE option causes the statement to be executed, not only planned. Then actual runtime statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. To use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, CREATE TABLE AS, or EXECUTE statement without letting the command affect your data, you can place it in a transaction and roll back it after the analysis is complete. <pre>START TRANSACTION; EXPLAIN ANALYZE ...; ROLLBACK;</pre>	• TRUE: displays the actual run times and other statistics. • FALSE: No display. Default value: TRUE
VERBOSE boolean	Displays additional information about the query plan. Additional information includes the output column list for each node in the plan tree, schema-qualify table and function names, always label variables in expressions with their range table alias, and always print the name of each trigger for which statistics are displayed.	• TRUE: The information is displayed. • FALSE: No display. Default value: TRUE
COSTS boolean	Displays the estimated total cost of each plan node, as well as the estimated number of rows and the estimated width of each row.	• TRUE: The information is displayed. • FALSE: No display. Default value: TRUE
CPU boolean	Displays CPU usage.	• TRUE: The CPU usage is displayed. • FALSE: No display. Default value: TRUE
DETAIL boolean	Displays DN information. NOTE For clusters of 8.2.1 or later, when Detail is enabled in EXPLAIN, the time required for comparing skew values is displayed in the execution plan.	• TRUE: prints DN information. • FALSE: No display. Default value: TRUE

Parameter	Description	Value Range
NODES boolean	Prints information about the nodes executed by query.	• TRUE: prints information about executed nodes. • FALSE: No display. Default value: TRUE
NUM_NODES boolean	Prints the quantity of executing nodes.	• TRUE: prints the number of DNs. • FALSE: No display. Default value: TRUE
BUFFERS boolean	Displays information about the buffer usage. The buffer information includes the number of shared blocks hit, read, and updated, the number of local blocks hit, read, and updated, the number of temp blocks hit, read, and updated. Shared blocks contain data from regular tables and indexes, local blocks contain data from temporary tables and indexes, and temporary blocks contain short-term working data used in sorts, hashes.	• TRUE: displays information on buffer usage. • FALSE: No display. Default value: FALSE
TIMING boolean	Displays the actual startup time and time spent in each node in the output.	• TRUE: displays the information. • FALSE: No display. Default value: TRUE
PLAN boolean	Specifies whether to store the execution plan in the plan_table table. When this option is enabled, the execution plan is stored in PLAN_TABLE and is not displayed on the current screen. Therefore, this option cannot be used together with other options when it is enabled.	• ON: The execution plan is stored in plan_table and is not displayed on the current screen. If the plan is stored successfully, EXPLAIN SUCCESS is returned. • OFF: The execution plan is not stored but is printed on the screen. Default value: ON

Parameter	Description	Value Range
FORMAT	Specifies the output format. The output content of each format is the same. XML, JSON, and YAML are easier for programs to parse SQL statements.	Text, XML, JSON, and YAML formats are supported. Default value: TEXT
BLOCKNAME boolean	Displays the blockname information of an operator. The blockname information is printed only when explain_perf_mode is set to pretty .	• TRUE: displays blockname information. • FALSE: displays no blockname information. Default value: TRUE
OUTLINE boolean	Displays outline information extracted from the plan. The outline is printed only when explain_perf_mode is set to pretty .	• TRUE: displays outline information. • FALSE: displays no outline information. Default value: TRUE
PERFORMANCE	This option prints all relevant information in execution.	-
STATS boolean	Specifies whether to display information required for reproducing the execution plan of an SQL statement, including the object definition, statistics, and configuration parameters. The information is usually used for fault locating.	• TRUE: display information required for reproducing the execution plan of an SQL statement. • FALSE: No display. Default value: TRUE
WARMUP	Processes the queried data in the sequence of A1in > A1out > Am. This is supported only by clusters of version 9.1.0.200 or later.	-
WARMUP HOT	Directly adds the queried data to the Am column. This is supported only by clusters of version 9.1.0.200 or later.	-

Examples

Create the **tpcds.customer_address_p1** table.

```
CREATE TABLE tpcds.customer_address_p1 AS TABLE tpcds.customer_address;
```

Change the value of **explain_perf_mode** to **normal**.

```
SET explain_perf_mode=normal;
```

Display an execution plan for simple queries in the table.

```
EXPLAIN SELECT * FROM tpcds.customer_address_p1;
      QUERY PLAN
-----
Data Node Scan on "__REMOTE_FQS_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes
(2 rows)
```

Generate an execution plan in JSON format (assume **explain_perf_mode** is set to **normal**).

```
EXPLAIN(FORMAT JSON) SELECT * FROM tpcds.customer_address_p1;
      QUERY PLAN
-----
[
  {
    "Plan": {
      "Node Type": "Data Node Scan",
      "RemoteQuery name": "__REMOTE_FQS_QUERY__",
      "Alias": "__REMOTE_FQS_QUERY__",
      "Startup Cost": 0.00,
      "Total Cost": 0.00,
      "Plan Rows": 0,
      "Plan Width": 0,
      "Nodes": "All datanodes"
    }
  }
]
(1 row)
```

If there is an index and we use a query with an indexable **WHERE** condition, **EXPLAIN** might show a different plan.

```
EXPLAIN SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
      QUERY PLAN
-----
Data Node Scan on "__REMOTE_LIGHT_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Node/s: datanode2
(2 rows)
```

Generate an execution plan in YAML format (assume **explain_perf_mode** is set to **normal**).

```
EXPLAIN(FORMAT YAML) SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
      QUERY PLAN
-----
- Plan:
  Node Type: "Data Node Scan"
  RemoteQuery name: "__REMOTE_LIGHT_QUERY__"
  Alias: "__REMOTE_LIGHT_QUERY__"
  Startup Cost: 0.00
  Total Cost: 0.00
  Plan Rows: 0
  Plan Width: 0
  Nodes: "datanode2"
(1 row)
```

Here is an example of an execution plan with cost estimates suppressed.

```
EXPLAIN(COSTS FALSE)SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
      QUERY PLAN
-----
Data Node Scan on "__REMOTE_LIGHT_QUERY__"
Node/s: datanode2
(2 rows)
```

Here is an example of an execution plan for a query that uses an aggregate function.


```
EXPLAIN SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1 WHERE ca_address_sk<10000;  
QUERY PLAN
```

```
-----  
Aggregate (cost=18.19..14.32 rows=1 width=4)  
-> Streaming (type: GATHER) (cost=18.19..14.32 rows=3 width=4)  
    Node/s: All datanodes  
    -> Aggregate (cost=14.19..14.20 rows=3 width=4)  
        -> Seq Scan on customer_address_p1 (cost=0.00..14.18 rows=10 width=4)  
            Filter: (ca_address_sk < 10000)  
(6 rows)
```

-- Delete the **tpcds.customer_address_p1** table.

```
DROP TABLE tpcds.customer_address_p1;
```

Run the **EXPLAIN PERFORMANCE** command for the **ANALYZE** statement.

```
EXPLAIN PERFORMANCE ANALYZE t2_dist_row;  
QUERY EXEC INFO
```

```
-----  
lock FirstCN:  
    coordinator1: actual time=0.240 loops=1  
estimate rows: actual time=[datanode3 0.000, datanode1 0.001]  
    coordinator1: actual time=0.000 loops=1  
    datanode1: actual time=0.001 loops=1  
    datanode2: actual time=0.000 loops=1  
    datanode3: actual time=0.000 loops=1  
sample rows: actual time=[datanode1 5.109, coordinator1 119.838]  
    coordinator1: actual time=119.838 loops=1  
    datanode1: actual time=5.109 loops=1  
    datanode2: actual time=5.621 loops=1  
    datanode3: actual time=5.342 loops=1  
fetch global stats:  
    coordinator1: actual time=8.501 loops=1  
calc stats: actual time=[datanode3 80.794, datanode2 109.155]  
    coordinator1: actual time=97.452 loops=1  
    datanode1: actual time=94.375 loops=1  
    datanode2: actual time=109.155 loops=1  
    datanode3: actual time=80.794 loops=1  
calc column stats: actual time=[datanode2 0.938, datanode3 9.811]  
    coordinator1: actual time=5.162 loops=2  
    datanode1: actual time=1.453 loops=2  
    datanode2: actual time=0.938 loops=2  
    datanode3: actual time=9.811 loops=2  
calc index stats: actual time=[datanode3 12.392, coordinator1 36.113]  
    coordinator1: actual time=36.113 loops=1  
    datanode1: actual time=15.933 loops=1  
    datanode2: actual time=13.419 loops=1  
    datanode3: actual time=12.392 loops=1  
calc expr stats: actual time=[datanode3 41.665, datanode2 78.442]  
    coordinator1: actual time=55.608 loops=1  
    datanode1: actual time=63.179 loops=1  
    datanode2: actual time=78.442 loops=1  
    datanode3: actual time=41.665 loops=1  
sync stats:  
    coordinator1: actual time=7.906 loops=1  
  
General Tracks  
CN build CN connection:  
    coordinator1: actual time=0.002 loops=1  
CN build DN connection:  
    coordinator1: actual time=0.070 loops=1  
-> execute ddl on other CN:  
    coordinator1: actual time=0.001 loops=1  
-> execute ddl on other DN:  
    coordinator1: actual time=0.000 loops=1  
Query Id: 72902018968225366  
Total runtime: 242.211 ms  
(48 rows)
```

Show the blockname of a plan.

```
EXPLAIN (BLOCKNAME ON) SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1 WHERE  
ca_address_sk<10000;
```

QUERY PLAN

id	operation	E-rows	E-memory	E-width	E-costs
1	-> Aggregate	1		12	16.14
2	-> Streaming (type: GATHER)	2		12	16.14
3	-> Aggregate	2	1MB	12	10.14
4	-> Seq Scan on customer_address_p1	7	1MB		4 10.12

Predicate Information (identified by plan id)

```
4 --Seq Scan on customer_address_p1  
  Filter: (ca_address_sk < 10000)
```

Query Block Name / Object Alias (identified by plan id)

```
1 - sel$1  
4 - sel$1 / customer_address_p1@"sel$1"
```

===== Query Summary =====

```
System available mem: 4710400KB  
Query Max mem: 4710400KB  
Query estimated mem: 2048KB  
(22 rows)
```

Show the outline of a plan.

```
EXPLAIN (OUTLINE ON) SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1 WHERE  
ca_address_sk<10000;
```

QUERY PLAN

id	operation	E-rows	E-memory	E-width	E-costs
1	-> Aggregate	1		12	16.14
2	-> Streaming (type: GATHER)	2		12	16.14
3	-> Aggregate	2	1MB	12	10.14
4	-> Seq Scan on customer_address_p1	7	1MB		4 10.12

Predicate Information (identified by plan id)

```
4 --Seq Scan on customer_address_p1  
  Filter: (ca_address_sk < 10000)
```

Outline Data

```
/*+  
  begin_outline_data  
  TableScan(@"sel$1" tpcds.customer_address_p1@"sel$1")  
  end_outline_data  
*/
```

===== Query Summary =====

```
System available mem: 4710400KB  
Query Max mem: 4710400KB  
Query estimated mem: 2048KB  
(25 rows)
```

Helpful Links

[ANALYZE](#) | [ANALYZE](#)

14.6 EXPLAIN PLAN

Function

You can run the **EXPLAIN PLAN** statement to save the information about an execution plan to the **PLAN_TABLE** table. Different from the **EXPLAIN** statement, **EXPLAIN PLAN** only stores plan information and does not print it on the screen.

Syntax

```
EXPLAIN PLAN  
[ SET STATEMENT_ID = string ]  
FOR statement ;
```

Parameter Description

Table 14-8 EXPLAIN PLAN parameters

Parameter	Description
PLAN	Stores plan information in PLAN_TABLE . If the storing is successful, EXPLAIN SUCCESS is returned.
STATEMENT_ID	Tags a query. The tag information will be stored in PLAN_TABLE . NOTE If the EXPLAIN PLAN statement does not contain SET STATEMENT_ID , the value of STATEMENT_ID is empty by default. In addition, the value of STATEMENT_ID cannot exceed 30 bytes. Otherwise, an error will be reported.

Precautions

- **EXPLAIN PLAN** cannot be executed on DNs.
- Plan information cannot be collected for SQL statements that failed to be executed.
- Data in **PLAN_TABLE** is in a session-level life cycle. Sessions are isolated from users and thereby users can view data of only the current session and current user.
- **PLAN_TABLE** cannot be joined with GDS foreign tables.
- For a query that cannot be pushed down, object information cannot be collected and only such information as **REMOTE_QUERY** and **CTE** can be collected. For details, see [Example 2](#).

Example 1

You can perform the following steps to collect execution plans of SQL statements by running **EXPLAIN PLAN**:

Step 1 Import TPC-H sample data. For details, see [Importing Sample Data](#).

Step 2 Run the **EXPLAIN PLAN** statement.

NOTE

After the **EXPLAIN PLAN** statement is executed, plan information is automatically stored in **PLAN_TABLE**. **INSERT**, **UPDATE**, and **ANALYZE** cannot be performed on **PLAN_TABLE**.

For details about **PLAN_TABLE**, see the **PLAN_TABLE** system view.

```
explain plan set statement_id='TPCH-Q4' for
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= '1993-07-01'::date
and o_orderdate < '1993-07-01'::date + interval '3 month'
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
```

Step 3 Query **PLAN_TABLE**.

```
SELECT * FROM PLAN_TABLE;
```

statement_id	plan_id	id	operation	options	object_name	object_type	object_owner	projection
TPCH-Q4	16781167	1	ROW ADAPTER					ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*))
TPCH-Q4	16781167	2	VECTOR SORT					ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*))
TPCH-Q4	16781167	3	VECTOR AGGREGATE	HASHED				ORDERS.O_ORDERPRIORITY, PG_CATALOG.COUNT(*)
TPCH-Q4	16781167	4	VECTOR STREAMING	GATHER				ORDERS.O_ORDERPRIORITY, (COUNT(*))
TPCH-Q4	16781167	5	VECTOR AGGREGATE	HASHED				ORDERS.O_ORDERPRIORITY, COUNT(*)
TPCH-Q4	16781167	6	VECTOR NESTED LOOPS	SEMI				ORDERS.O_ORDERPRIORITY
TPCH-Q4	16781167	7	TABLE ACCESS	CSTORE SCAN	ORDERS	TABLE	TPCH	ORDERS.O_ORDERKEY
TPCH-Q4	16781167	8	VECTOR MATERIALIZE					LINEITEM.L_ORDERKEY
TPCH-Q4	16781167	9	TABLE ACCESS	CSTORE SCAN	LINEITEM	TABLE	TPCH	LINEITEM.L_ORDERKEY

Step 4 Delete data from **PLAN_TABLE**.

```
DELETE FROM PLAN_TABLE WHERE xxx;
```

----End

Example 2

For a query that cannot be pushed down, only such information as **REMOTE_QUERY** and **CTE** can be collected from **PLAN_TABLE** after **EXPLAIN PLAN** is executed.

The optimizer generates a plan for pushing down statements. In this case, only **REMOTE_QUERY** can be collected.

```
explain plan set statement_id = 'test remote query' for
select
current_user
from
customer;
```

Query **PLAN_TABLE**.

```
SELECT * FROM PLAN_TABLE;
```

statement_id	plan_id	id	operation	options	object_name	object_type	object_owner	projection
test remote query	29360133	1	NESTED LOOPS	CARTESIAN				'apple'::name
test remote query	29360133	2	DATA NODE SCAN		customer	REMOTE_QUERY		
test remote query	29360133	3	DATA NODE SCAN		customer_address	REMOTE_QUERY		

(3 rows)

14.7 LOCK

Function

LOCK TABLE obtains a table-level lock.

When the lock for commands referencing a table is automatically acquired, DWS always uses the lock mode with minimum constraints. Use **LOCK** if users need a more strict lock mode. For example, suppose an application runs a transaction at the **Read Committed** isolation level and needs to ensure that data in a table remains stable in the duration of the transaction. To achieve this, you could obtain **SHARE** lock mode over the table before the query. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data. It is because the **SHARE** lock mode conflicts with the **ROW EXCLUSIVE** lock acquired by writers, and your **LOCK TABLE name IN SHARE MODE** statement will wait until any concurrent holders of **ROW EXCLUSIVE** mode locks commit or roll back. Therefore, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

Precautions

- **LOCK TABLE** is useless outside a transaction block: the lock would remain held only to the completion of the statement. If **LOCK TABLE** is out of any transaction block, an error is reported.
- If no lock mode is specified, then **ACCESS EXCLUSIVE**, the most restrictive mode, is used.
- **LOCK TABLE ... IN ACCESS SHARE MODE** requires the **SELECT** permission on the target table. All other forms of **LOCK** require table-level **UPDATE** and/or the **DELETE** permission.
- There is no **UNLOCK TABLE** command. Locks are always released at transaction end.
- **LOCK TABLE** only deals with table-level locks, and so the mode names involving **ROW** are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, **ROW EXCLUSIVE** mode is a shareable table lock. Keep in mind that all the lock modes have identical semantics so far as **LOCK TABLE** is concerned, differing only in the rules about which modes conflict with which. For details about the rules, see [Lock Levels and Conflicts](#).

Syntax

```
LOCK [ TABLE ] {[ ONLY ] name [, ...] | {name [ * ]} [, ...]}  
  [ IN {ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE  
ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE | UPDATE EXCLUSIVE} MODE ]  
  [ NOWAIT ] [LOCAL COORDINATOR ONLY];
```

Lock Levels and Conflicts

DWS uses lock modes to control how concurrent transactions access resources, ensuring data consistency and preventing loss.

[Table 14-9](#) shows the eight main lock modes and their potential conflicts.

Table 14-9 Conflicts between lock modes

Lock Mode	Lock Level	Lock Usage	Conflict
AccessShareLock	1	SELECT statement, allowing other transactions to read data.	8
RowShareLock	2	SELECT FOR UPDATE or FOR SHARE, allowing other transactions to read but preventing writes.	7 8
RowExclusiveLock	3	INSERT, UPDATE, and DELETE statements prevent other transactions from reading and writing.	5 6 7 8
ShareUpdateExclusiveLock	4	VACUUM (non-FULL), ANALYZE, CREATE INDEX CONCURRENTLY, and COMMENT ON statements allows other transactions to read data but blocks writes.	4 5 6 7 8
ShareLock	5	CREATE INDEX (non-CONCURRENTLY) prevents other transactions from reading and writing.	3 4 6 7 8
ShareRowExclusiveLock	6	Similar to RowExclusiveLock, this lock lets you read data safely by using ROW SELECT...FOR UPDATE . But it permits RowShareLock, allowing others to read the row but not update or delete it.	3 4 5 6 7 8
ExclusiveLock	7	This lock prevents RowShareLock or SELECT... FOR UPDATE operations.	2 3 4 5 6 7 8
AccessExclusiveLock	8	The lock stops other transactions from reading and writing data during ALTER TABLE , DROP TABLE , or VACUUM FULL operations.	1 2 3 4 5 6 7 8

Parameter Description

Table 14-10 LOCK parameters

Parameter	Description	Value Range
name	The name (optionally schema-qualified) of an existing table to lock. The tables are locked one-by-one in the order specified in the LOCK TABLE command.	An existing table name.
ONLY	If ONLY is specified, only the specified table is deleted. If ONLY is not specified, the specified table and all its inherited tables are deleted. This parameter is reserved only for compatibility with PostgreSQL. GaussDB(DWS) does not support inherited tables.	-
ACCESS SHARE	ACCESS SHARE allows only read operations on a table. In general, any SQL statements that only read a table and do not modify it will acquire this lock mode. The SELECT command acquires a lock of this mode on referenced tables.	-
ROW SHARE	ROW SHARE allows concurrent read of a table but does not allow any other operations on the table. SELECT FOR UPDATE and SELECT FOR SHARE automatically acquire the ROW SHARE lock on the target table and add the ACCESS SHARE lock to other referenced tables except FOR SHARE and FOR UPDATE .	-
ROW EXCLUSIVE	Like ROW SHARE , ROW EXCLUSIVE allows concurrent read of a table and modification of data in the table. UPDATE , DELETE , and INSERT automatically acquire this lock on the target table and add the ACCESS SHARE lock to other referenced tables. Generally, all commands that modify table data acquire the ROW EXCLUSIVE lock for tables.	-

Parameter	Description	Value Range
SHARE UPDATE EXCLUSIVE	This mode protects a table against concurrent schema changes and VACUUM runs. Acquired by VACUUM (without FULL), ANALYZE and CREATE INDEX CONCURRENTLY statements.	-
SHARE	SHARE allows concurrent queries of a table but does not allow modification of the table. Acquired by CREATE INDEX (without CONCURRENTLY).	-
SHARE ROW EXCLUSIVE	SHARE ROW EXCLUSIVE protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time. No SQL statements automatically acquire this lock mode.	-
EXCLUSIVE	EXCLUSIVE allows concurrent queries of the target table but does not allow any other operations. This mode allows only concurrent ACCESS SHARE locks; that is, only reads from the table can proceed in parallel with a transaction holding this lock mode. No SQL statements automatically acquire this lock mode on user tables. However, it will be acquired on some system tables in case of some operations.	-
ACCESS EXCLUSIVE	This mode guarantees that the holder is the only transaction accessing the table in any way. Acquired by the ALTER TABLE , DROP TABLE , TRUNCATE , REINDEX , CLUSTER , and VACUUM FULL commands.	This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Parameter	Description	Value Range
UPDATE EXCLUSIVE	The UPDATE EXCLUSIVE lock allows concurrent (AUTO) VACUUM and (AUTO) ANALYZE, but does not allow concurrent (AUTO) VACUUM. NOTE - This parameter is supported only by clusters of version 8.2.1.200 or later. - The UPDATE EXCLUSIVE lock is used only in the VACUUM syntax.	-
NOWAIT	Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified locks cannot be acquired immediately without waiting, the transaction is aborted. If NOWAIT is not specified, LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released.	-
LOCAL COORDINATOR ONLY	Specifies that LOCK TABLE is executed only on the CN that receives the current session request and is not delivered to other CNs or all DN. This is used only for metadata operations to improve efficiency. NOTICE - This parameter is supported by version 8.2.0.100 or later clusters. - Currently, only the ACCESS SHARE lock mode is supported. If other lock modes are used, an error will be reported.	-

Examples

Obtain a **SHARE** lock on a primary key table when going to perform inserts into a foreign key table.

```
START TRANSACTION;

LOCK TABLE tpceds.reason IN SHARE MODE;

SELECT r_reason_desc FROM tpceds.reason WHERE r_reason_sk=5;
r_reason_desc
-----
Parts missing
(1 row)

COMMIT;
```

Obtain a **SHARE ROW EXCLUSIVE** lock on a primary key table when performing a delete operation.

```
CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;

START TRANSACTION;

LOCK TABLE tpcds.reason_t1 IN SHARE ROW EXCLUSIVE MODE;

DELETE FROM tpcds.reason_t1 WHERE r_reason_desc IN(SELECT r_reason_desc FROM tpcds.reason_t1
WHERE r_reason_sk < 6 );

DELETE FROM tpcds.reason_t1 WHERE r_reason_sk = 7;

COMMIT;
```

Delete the **tpcds.reason_t1** table.

```
DROP TABLE tpcds.reason_t1;
```

Add the **ACCESS SHARE** lock to the table and set the lock scope to the current CN.

```
BEGIN;
LOCK TABLE lock_test IN ACCESS SHARE MODE LOCAL COORDINATOR ONLY;
SELECT pg_get_tabledef('lock_test');
END;
```

14.8 MERGE INTO

Function

MERGE INTO matches data in a source table with that in a target table based on a join condition. If data matches, **UPDATE** will be executed on the target table. Otherwise, **INSERT** will be executed. **MERGE INTO** handles both **UPDATE** and **INSERT** operations within a single statement, potentially improving performance.

Precautions

- To run **MERGE INTO**, you must have the **UPDATE** and **INSERT** permissions for the target table, as well as the **SELECT** permission for the source table.
- **PREPARE** is not supported.
- **MERGE INTO** cannot be executed during redistribution.
- **MERGE INTO** cannot be executed for target tables that contain triggers.
- When executing **MERGE INTO** for the round-robin table, you are advised to disable the GUC parameter **allow_concurrent_tuple_update**, or some **MERGE INTO** statements are not supported.

Syntax

```
MERGE [/*+ plan_hint */] INTO table_name [ partition_clause ] [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ partition_clause ] [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
    UPDATE SET { column_name = { expression | DEFAULT } |
      ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
```

```
INSERT { DEFAULT VALUES |  
[ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] [ WHERE condition ] }  
];
```

where partition_clause can be:

```
PARTITION ( partition_name ) | PARTITION FOR ( partition_key_value [, ...] )
```

Parameter Description

Table 14-11 MERGE INTO parameters

Parameter	Description	Value Range
plan_hint clause	Following the keyword in the /*+ */ format, hints are used to optimize the plan generated by a specified statement block. For details, see Hint-based Tuning .	-
INTO clause	Specifies the target table that is being updated or has data being inserted. • table_name Specifies the name of the target table. • partition_clause Specifies a partition in the target table. This parameter is supported only by clusters of version 9.1.0 or later. Partitions can be specified only if the target table is a partitioned one. The specified partition name must already exist in the target table or the partition specified by the partition key already exists. • alias Specifies the alias for the target table.	Valid table name.

Parameter	Description	Value Range
USING clause	Specifies the source table, which can be a table, view, or subquery. • partition_clause Specifies a partition in the source table. This parameter is supported only by clusters of version 9.1.0 or later. Partitions can be specified only if the source table is a partitioned one. Partitions cannot be specified for views or subqueries. The specified partition name must already exist in the source table or the partition specified by the partition key already exists. • alias Specifies the alias for the target table.	Valid source table name.
ON clause	Specifies the condition used to match data between the source and target tables. • Columns in the condition cannot be updated. • The ON association condition can be ctid, xc_node_id, or tableoid.	-

Parameter	Description	Value Range
WHEN MATCHED clause	Performs UPDATE if data in the source table matches that in the target table based on the condition. • Distribution keys cannot be updated. • System catalogs and system columns cannot be updated. • With a partition specified in the target table, if the UPDATE causes the partition keys to change: – If the target table is a column-store table, the error message "The inserted partition key is not mapped to the specified 'partition_name' partition" is displayed. – If the target table is a row-store table: If ENABLE ROW MOVEMENT is set for the target table, no error will be reported, the updated data will belong to the new partition. If Disable ROW MOVEMENT is set for the target table, the error message "fail to update partitioned table 'table_name'" will be displayed.	-

Parameter	Description	Value Range
WHEN NOT MATCHED clause	Specifies that the INSERT operation is performed if data in the source table does not match that in the target table based on the condition. • An INSERT clause cannot contain multiple VALUES. • The order of WHEN MATCHED and WHEN NOT MATCHED clauses can be reversed. One of them can be used by default, but they cannot be both used at one time. Two WHEN MATCHED or WHEN NOT MATCHED clauses cannot be specified at the same time. • If a partition in the target table is specified and more data beyond the scope of the partition is inserted, the error message "The inserted partition key is not mapped to the specified 'partition_name' partition" will be displayed.	-
DEFAULT	Sets the column to its default value.	The value is NULL if no specified default value has been assigned to it.
WHERE condition	Specifies the conditions for the UPDATE and INSERT clauses. The two clauses will be executed only when the conditions are met. This parameter can be left empty. System columns cannot be referenced in WHERE condition.	-

Examples

Create the target table **products** and source table **newproducts**, and insert data to them.

```
CREATE TABLE products
(
  product_id INTEGER,
  product_name VARCHAR2(60),
  category VARCHAR2(60)
);

INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrncs');
INSERT INTO products VALUES (1502, 'olympus is50', 'electrncs');
INSERT INTO products VALUES (1600, 'play gym', 'toys');
INSERT INTO products VALUES (1601, 'lamaze', 'toys');
```

```
INSERT INTO products VALUES (1666, 'harry potter', 'dvd');

CREATE TABLE newproducts
(
  product_id INTEGER,
  product_name VARCHAR2(60),
  category VARCHAR2(60)
);

INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrnics');
INSERT INTO newproducts VALUES (1601, 'lamaze', 'toys');
INSERT INTO newproducts VALUES (1666, 'harry potter', 'toys');
INSERT INTO newproducts VALUES (1700, 'wait interface', 'books');
```

Run **MERGE INTO**. If the **product_id** field is matched and the **product_name** field in the target table is not **play gym**, UPDATE is executed. If the **product_id** field is not matched and the **category** field in the source table is **books**, INSERT is executed.

```
MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED THEN
  UPDATE SET p.product_name = np.product_name, p.category = np.category WHERE p.product_name !=
  'play gym'
WHEN NOT MATCHED THEN
  INSERT VALUES (np.product_id, np.product_name, np.category) WHERE np.category = 'books';
```

View the query result. The data in **product_id** of rows 1502 and 1666 in the **products** table was updated, and the data in row 1700 was newly inserted.

```
SELECT * FROM products ORDER BY product_id;
product_id | product_name | category
-----+-----+-----
1501 | vivitar 35mm | electrncs
1502 | olympus camera | electrncs
1600 | play gym | toys
1601 | lamaze | toys
1666 | harry potter | toys
1700 | wait interface | books
(6 rows)
```

FAQs

[Error "unable to get a stable set of rows in the source table"](#)

14.9 INSERT and UPSERT

14.9.1 INSERT

Function

INSERT inserts new rows into a table.

Precautions

- For how to use INSERT, see [INSERT Operation](#).
- You must have the **INSERT** permission on a table in order to insert into it.
- Use of the **RETURNING** clause requires the **SELECT** permission on all columns mentioned in **RETURNING**.

- If you use the **query** clause to insert rows from a query, you of course need to have the **SELECT** permission on any table or column used in the query.
- Using the **OVERWRITE** clause requires both **SELECT** and **TRUNCATE** permissions on the table since it deletes existing data before inserting new records.
- When you connect to a database compatible to Teradata and **td_compatible_truncation** is **on**, a long character string will be automatically truncated. If later **INSERT** statements (not involving foreign tables) insert long strings to columns of char- and varchar-typed columns in the target table, the system will truncate the long strings to ensure no strings exceed the maximum length defined in the target table.

NOTICE

If inserting multi-byte character data (such as Chinese characters) to database with the character set byte encoding (SQL_ASCII, LATIN1), and the character data crosses the truncation position, the string is truncated based on its bytes instead of characters. Unexpected result will occur in tail after the truncation. If you want correct truncation result, you are advised to adopt encoding set such as UTF8, which has no character data crossing the truncation position.

Syntax

```
[ WITH [ RECURSIVE ] with_query [ , ... ] ]  
INSERT [ /*+ plan_hint */ ] [ IGNORE | OVERWRITE ] INTO table_name [ partition_clause ] [ AS alias ]  
[ ( column_name [ , ... ] ) ]  
  { DEFAULT VALUES  
  | VALUES { ( { expression | DEFAULT } [ , ... ] ) } [ , ... ]  
  | query }  
  [ ON DUPLICATE KEY duplicate_action | ON CONFLICT [ conflict_target ] conflict_action ]  
  [ RETURNING { * | { output_expression [ [ AS ] output_name ] } [ , ... ] } ] ;
```

where partition_clause can be:
PARTITION (partition_name)
| PARTITION FOR (partition_key_value [, ...])

and duplicate_action can be:

```
UPDATE { column_name = { expression | DEFAULT } |  
      ( column_name [ , ... ] ) = ( { expression | DEFAULT } [ , ... ] )  
      } [ , ... ]
```

and conflict_target can be one of:

```
( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [ , ... ] ) [ WHERE  
index_predicate ]  
ON CONSTRAINT constraint_name
```

and conflict_action is one of:

```
DO NOTHING  
DO UPDATE SET { column_name = { expression | DEFAULT } |  
              ( column_name [ , ... ] ) = ( { expression | DEFAULT } [ , ... ] )  
              } [ , ... ]  
[ WHERE condition ]
```


Parameter Description

Table 14-12 INSERT parameters

Parameter	Description	Value Range or Example
WITH [RECURSIVE] with_query [, ...]	The WITH clause allows you to specify one or more subqueries that can be referenced by name in the primary query, equal to temporary table. If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. The format of with_query is as follows: with_query_name [(column_name [, ...])] AS ({select values insert update delete}) with_query_name specifies the name of the result set generated by a subquery. Such names can be used to access the result sets of subqueries in a query. column_name specifies the column name displayed in the subquery result set. Each subquery can be a SELECT, VALUES, INSERT, UPDATE or DELETE statement.	Generating Temporary Tables with WITH and Inserting Data into Another Table
plan_hint clause	Following the keyword in the /+ * / format, hints are used to optimize the plan generated by a specified statement block. For details, see Hint-based Tuning .	-
IGNORE	Specifies that the data that duplicates an existing primary key or unique key value will be ignored. For details, see UPSERT.	Using IGNORE to Ignore Primary Key Conflicts

Parameter	Description	Value Range or Example
OVERWRITE	Specifies the overwrite mode. After this mode is used, the original data is cleared and only the newly inserted data exists. This parameter is available only for clusters of version 8.1.1 or later. You can specify the columns on which OVERWRITE takes effect, and the other columns will keep their original data. If a column has no original data, its value is NULL. INSERT OVERWRITE constraint: • Do not perform OVERWRITE and INSERT INTO operations at the same time. Otherwise, data written in real time may be unexpectedly cleared. • INSERT OVERWRITE is unavailable for replication tables. OVERWRITE applies to the scenario where a large amount of data is imported. You are not advised to use OVERWRITE to insert a small amount of data. • Do not concurrently perform insert overwrite operations on the same table. Otherwise, an error similar to "tuple concurrently updated." will be reported. • If the cluster is being scaled out and data redistribution is required for the table where INSERT OVERWRITE is performed, INSERT OVERWRITE clears the current data and automatically distributes the inserted data to the new nodes after scale-out. If INSERT OVERWRITE and the data redistribution of the table are performed at the same time, INSERT OVERWRITE will interrupt the data redistribution of the table. • OVERWRITE allows you to insert data into an OBS foreign table in overwrite mode, where the data in the original directory will be overwritten. • The foreign table supports OVERWRITE single-partition and multi-level Value partitioning. This function is supported only in 9.1.0.100 and later versions.	-

Parameter	Description	Value Range or Example
	Example: INSERT OVERWRITE into table_name partition for (p1_value, p2_value...) ... When INSERT OVERWRITE and ALTER TABLE...ADD PARTITION are both executed on a given table, the number of partitions on different nodes may be inconsistent. As a result, an error is reported during the execution of INSERT OVERWRITE. You are advised to run INSERT OVERWRITE again.	
table_name	Specifies the name of the target table.	An existing table name.
AS	Specifies an alias for the target table <i>table_name</i> . <i>alias</i> indicates the alias name.	-
column_name	Specifies the name of a column in a table. - The column name can be qualified with a subfield name or array subscript, if needed. - Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or NULL if there is none. (Inserting into only some fields of a composite column leaves the other fields null.) - The target column names column_name can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order. - The target columns are the first N column names, if there are only N columns supplied by the value clause or query. - The values supplied by the value clause or query are associated with the explicit or implicit column list left-to-right.	An existing column name.
partition_name	Specifies the name of a partition.	Name of an existing partition.

Parameter	Description	Value Range or Example
partition_key_value	Specifies the key value of a partition. The value specified by PARTITION FOR (partition_key_value [, ...]) can uniquely identify a partition. Multiple partition key values can be specified. Use commas (,) to separate multiple partition key values.	Value range of the partition key of the partition to be renamed.
expression	Specifies an expression or a value to assign to the corresponding column. • If single-quotation marks are inserted in a column, the single-quotation marks need to be used for escape. • If the expression for any column is not of the correct data type, automatic type conversion will be attempted. If the attempt fails, data insertion fails and the system returns an error message.	Inserting an Expression or Value into a Column
DEFAULT	Specifies the default value of the corresponding field name.	The value is NULL if no specified default value has been assigned to it. For details, see Using the Default Value of a Column When Writing Data .
query	Specifies a query statement (SELECT statement) that uses the query result as the inserted data.	Inserting Query Results into a Table
ON DUPLICATE KEY	Specifies that the data that duplicates an existing primary key or unique key value will be updated. <i>duplicate_action</i> specifies the columns and data to be updated. For details, see UPSERT .	Inserting Data with ON DUPLICATE KEY for Handling Primary Key/Unique Conflicts

Parameter	Description	Value Range or Example
ON CONFLICT	Specifies that the data that duplicates an existing primary key or unique key value will be ignored or updated. <i>conflict_target</i> specifies the column name <i>index_column_name</i>, expression <i>index_expression</i> that contains multiple column names, or constraint name <i>constraint_name</i>. It is used to infer whether there is a unique index from the column name, the expression that contains multiple column names, or the constraint name. <i>index_column_name</i> and <i>index_expression</i> must comply with the index column format of CREATE INDEX. <i>conflict_action</i> specifies the policy to be executed upon a primary key or unique constraint conflict. There are two available actions: • DO NOTHING: Ignore the conflict. • DO UPDATE SET: Update data upon a conflict. The columns and data to be updated must be specified. For details, see UPSERT.	Examples
RETURNING	Returns the inserted rows. The syntax of the RETURNING list is identical to that of the output list of SELECT .	Using RETURNING to Return Insertion Results
output_expression	An expression used to calculate the output of the INSERT command after each row is inserted.	The expression can use any column names of the table. Write * to return all columns of the inserted row(s).
output_name	Specifies a name to use for a returned column.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Inserting One or More Data Records

Create the **reason_t1** table.

```
DROP TABLE IF EXISTS reason_t1;  
CREATE TABLE reason_t1  
(  
    TABLE_SK    INTEGER  
    TABLE_ID    VARCHAR(20)  
    TABLE_NA    VARCHAR(20)  
);
```

Insert one data record to the table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

Insert one data record into the table. This command is equivalent to the last one.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA');
```

Insert records whose **TABLE_SK** is less than **1** into the table.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

Insert records into the table.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');  
SELECT * FROM reason_t1 ORDER BY 1;
```

table_sk	table_id	table_na
1	S01	StudentA
1	S01	StudentA
1	S01	StudentA
2	T01	TeacherA
3	T02	TeacherB

Inserting Query Results into a Table

Insert data back into the **reason_t1** table.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1;
```

Using RETURNING to Return Insertion Results

Insert table data and return the values of **table_id** and **table_na** corresponding to the inserted data.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 RETURNING table_id, table_na;
```

table_id	table_na
T01	TeacherA
T02	TeacherB
S01	StudentA
T02	TeacherB
T01	TeacherA

Insert data and return all column values of the inserted data.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 RETURNING *;
```

table_sk	table_id	table_na
2	T01	TeacherA
3	T02	TeacherB
2	T01	TeacherA
3	T02	TeacherB
1	S01	StudentA

Insert data and return the value of the renamed column **my_id**.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA') RETURNING table_id AS my_id;
```

my_id
S01

Using INSERT OVERWRITE to Insert Data in Overwriting Mode

Use INSERT OVERWRITE to update data in a table, that is, insert data in overwriting mode (clear the data in the target table first).

```
INSERT OVERWRITE INTO reason_t1 values (5, 'S02', 'StudentB');  
SELECT * FROM reason_t1 ORDER BY 1;
```

table_sk	table_id	table_na
5	S02	StudentB

Using the Default Value of a Column When Writing Data

```
DROP TABLE IF EXISTS products;  
CREATE TABLE products (
```

```
id INT PRIMARY KEY,  
name VARCHAR(50),  
price DECIMAL(10, 2) DEFAULT 0.00  
);
```

Insert a data record. If the value of the **price** column is not defined, **0.00** is used by default.

```
INSERT INTO products (id, name)VALUES (201, 'phone'); -- The price column uses the default value 0.00.
```

Insert a data record and explicitly use the default value.

```
INSERT INTO products (id, name, price)VALUES (202, 'pad', DEFAULT); -- Explicitly use the default value.  
SELECT * FROM product;
```

id	name	price
201	phone	0.00
202	pad	0.00

Generating Temporary Tables with WITH and Inserting Data into Another Table

Insert some data in a table to another table: Use the **WITH** subquery to obtain a temporary table **temp_t**, and then insert all data in **temp_t** to another table **reason_t1**.

```
WITH temp_t AS (SELECT * FROM reason_t1) INSERT INTO reason_t1 SELECT * FROM temp_t ORDER BY 1;
```

Using IGNORE to Ignore Primary Key Conflicts

```
DROP TABLE IF EXISTS users;  
CREATE TABLE users (  
id INT PRIMARY KEY,  
username VARCHAR(50)  
)  
DISTRIBUTE BY HASH (id);
```

Insert two records whose primary keys are both **1**. A conflict occurs and the operation cannot be performed.

```
INSERT INTO users (id, username)  
VALUES (1, 'john_doe'),  
(2, 'jane_smith'),  
(1, 'duplicate_user'); --Primary key conflict. Insertion failed.
```

```
error_code: DWS.S0010010  
error_msg: Constraint Violation. STATE: 23505, message: ERROR: dn_6005_6006: duplicate key value violates  
unique constraint "users_pkey"  
Detail: Key (id)=(1) already exists.
```

Use **IGNORE** to ignore the conflict. The conflicting data is ignored and skipped.

```
INSERT IGNORE INTO users (id, username)  
VALUES (1, 'john_doe'),  
(2, 'jane_smith'),  
(1, 'duplicate_user'); -- Conflicting data will be ignored.  
SELECT * FROM users;
```


id	username
1	john_doe
2	jane_smith

Inserting Data with ON DUPLICATE KEY for Handling Primary Key/Unique Conflicts

```
DROP TABLE IF EXISTS user_logins;  
CREATE TABLE user_logins  
(user_id int primary key,  
login_count int)  
DISTRIBUTE BY HASH (user_id);
```

Insert two records. The primary key of the second record conflicts with that of the first record. The insertion fails.

```
INSERT INTO user_logins VALUES (1, 1);  
INSERT INTO user_logins VALUES (1, 1); --Primary key conflict. Insertion failed.  
SELECT * FROM user_logins;
```

user_id	login_count
1	1

If a duplicate key exists, increment the **login_count** field by 1 using the **ON DUPLICATE KEY** clause.

```
INSERT INTO user_logins VALUES (1, 1) ON DUPLICATE KEY UPDATE login_count = login_count + 1;  
SELECT * FROM user_logins;
```

user_id	login_count
1	2

Inserting an Expression or Value into a Column

```
CREATE TABLE tt01 (id int,content varchar(50));
```

```
INSERT INTO tt01 values (1,'Jack say "hello"'); --Insert "Jack say'hello'" and add single quotation marks to escape.  
INSERT INTO tt01 values (2,'Rose do 50%');  
INSERT INTO tt01 values (3,'Lilei say "world"'); --Insert "Lilei say'world'" and add single quotation marks to escape.  
INSERT INTO tt01 values (4,'Hanmei do 100%');  
  
SELECT * FROM tt01;
```

id	content
3	Lilei say 'world'
2	Rose do 50%
1	Jack say 'hello'
4	Hanmei do 100%

Inserting Data into a Specified Partition of a Partitioned Table

Insert data into a specified partition of a partitioned table.

```
CREATE TABLE test_range_row(a int, d int)
DISTRIBUTE BY hash(a) PARTITION BY RANGE(d)
(
  PARTITION p1 values LESS THAN (60),
  PARTITION p2 values LESS THAN (75),
  PARTITION p3 values LESS THAN (90),
  PARTITION p4 VALUES LESS THAN (maxvalue)
);
INSERT OVERWRITE INTO test_range_row PARTITION(p1) VALUES(55,51);
INSERT OVERWRITE INTO test_range_row PARTITION(p3) VALUES(85,80);

DELETE FROM test_range_row PARTITION(p1);
```

Insert data into a partition of a partitioned table.

```
CREATE FOREIGN TABLE test_ft(c1 text, c2 text, c3 text, c4 text)
PARTITION BY (c1,c2,c3) AUTOMAPPED;
```

Insert data into the level-1 (p1_1) partition.

```
INSERT OVERWRITE INTO test_ft PARTITION FOR(p1_1) VALUES(p1_1,2,3,4);
```

Insert data into the level-2 (p1_2, p2_2) partition.

```
INSERT OVERWRITE INTO test_ft PARTITION FOR(p1_2,p2_2) VALUES(p1_2,p2_2,3,4);
```

Insert data into the level-3 (p1_3,p2_3,p3_3) partition.

```
INSERT OVERWRITE INTO test_ft PARTITION FOR(p1_3,p2_3,p3_3) VALUES(p1_3,p2_3,p3_3,4);
```

FAQs

- ["inserted partition key does not map to any table partition" Is Reported When Data Is Inserted into a Partitioned Table](#)
- ["value too long for type character varying" Is Displayed When VARCHAR\(n\) Stores Chinese Characters](#)

14.9.2 UPSERT

Function

UPSERT inserts rows into a table. When a primary key or unique constraint conflict occurs, it can choose to update the conflicting data or ignore the conflict.

NOTICE

The **UPSERT** syntax is supported only in 8.1.1 and later.

Precautions



WARNING

- Do not concurrently update the same column-store table using the **UPDATE** and **UPSERT** statements.
 - For more information about development and design proposal, see [Development and Design Proposal](#).
-
- Permission requirements: Only users with the **INSERT** or **UPDATE** permission on a table can run the **UPSERT** statement to insert data to or update data in the table.
 - Index requirements: The **UPSERT** statement of updating data upon conflict can be executed only when the target table contains a primary key or unique index. The **UPSERT** statement of updating data upon conflict cannot be executed if no unique indexes are available. You can execute the statement only after the indexes are rebuilt.
 - Other requirements:
 - The **WITH** clause cannot be used at the same time.
 - **INSERT OVERWRITE** cannot be used at the same time.
 - The **UPDATE** clause and its **WHERE** clause do not support subqueries.
 - **VALUES(colname)** in the **UPDATE** clause does not support outer nested functions. That is, the usage similar to **sqrt(VALUES(colname))** is not supported. To support this function, use the **EXCLUDED.colname** syntax.
 - **INSERT INTO ON CONFLICT(...) DO UPDATE** must contain **conflict_target**. That is, a column or constraint name must be specified.
 - The distribution column cannot be updated. (Exception: Update is allowed if the distribution key is the same as the updated value.)

```
CREATE TABLE t1(dist_key int PRIMARY KEY, a int, b int);
INSERT INTO t1 VALUES(1,2,3) ON CONFLICT(dist_key) DO UPDATE SET dist_key =
EXCLUDED.dist_key, a = EXCLUDED.a + 1;
INSERT INTO t1 VALUES(1,2,3) ON CONFLICT(dist_key) DO UPDATE SET dist_key = dist_key, a =
EXCLUDED.a + 1;
```
 - The **UPSERT** statement cannot be executed on the target table that contains a trigger (with the **INSERT** or **UPDATE** trigger event).
 - The **UPSERT** statement is not supported for updatable views.
 - The **UPDATE** clause, the **WHERE** clause of **UPDATE**, and the index condition expression should not contain [functions that cannot be pushed down](#).
 - Unique indexes cannot be deferred.
 - When performing the update operation of **UPSERT** using **INSERT INTO SELECT**, pay attention to the query result sequence of **SELECT**. In a distributed environment, if the **ORDER BY** statement is not used, the sequence of returned results may be different each time the same **SELECT** statement is executed. As a result, the execution result of the **UPSERT** statement does not meet the expectation.
 - Multiple updates are not supported. When conflicting data groups exist, you will see an error like "INSERT ON CONFLICT DO UPDATE command

cannot affect row a second time." This does not happen if the query uses a PGXC plan. For details, see [FAQs](#).

- Impact on performance:
 - When UPSERT is executed on column-store tables, enable DELTA tables to avoid small CUs from being generated. A large number of small CUs may cause tables to bloat or lead to poor query performance.
 - **UPSERT**, **UPDATE**, and **DELETE** operations cannot be concurrently performed because they need to wait for the CU lock. This problem cannot be solved even if the DELTA table is enabled. To execute **UPSERT**, **UPDATE**, and **DELETE** operations concurrently on column-store tables, use HStore tables.
- Deadlock: A distributed deadlock may occur, resulting in query hanging. When many **UPSERT** commands run together in a transaction or batch via JDBC with **setAutoCommit(false)**, deadlocks can happen if several tasks try to update the same row simultaneously. This occurs because different threads on various nodes might update rows in varying orders. To solve this issue, you can:
 - a. Decrease the value of the GUC parameter **lockwait_timeout** (default value: 20 minutes). A distributed deadlock error will be reported after waiting for *the value of lockwait_timeout*. You can decrease the value of this parameter to reduce the service waiting time caused by a deadlock.
 - b. Only import data with the same primary key through one database connection. You can run **UPSERT** statements at the same time.
 - c. Ensure that only one UPSERT statement is executed in each transaction. In this way, **UPSERT** statements can be executed concurrently.
 - d. Run all **UPSERT** operations one at a time in a single thread; avoid running them simultaneously.

Method 1 reduces waiting time but does not fix deadlocks. When using the **UPSERT** statement in services, lower this parameter's value. Methods 2, 3, and 4 resolve deadlocks, with method 2 being preferred due to its superior performance.

Syntax

For details, see [Syntax](#) of **INSERT**. The following table describes the syntax of **UPSERT**.

The first syntax does not specify an index. The system checks for conflicts on all primary keys or unique indexes. If a conflict exists, the system ignores or updates the corresponding data.

The second syntax is to specify an index. The primary key or unique index is inferred from the column name, the expression that contains column names, or the constraint name specified in the **ON CONFLICT** clause.

- Scenario 1: If inserting data causes a conflict, the system updates the current data with the new information.
 - Conflict update without specifying an index:
`INSERT INTO ON DUPLICATE KEY UPDATE`
 - Conflict update with an index specified:
`INSERT INTO ON CONFLICT(...) DO UPDATE SET`
`INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO UPDATE SET`

- Scenario 2: If a data insertion causes a conflict, the system skips the operation.
 - Conflict ignorance without specifying an index:
INSERT IGNORE
INSERT INTO ON CONFLICT DO NOTHING
 - Conflict ignorance with an index specified:
INSERT INTO ON CONFLICT(...) DO NOTHING
INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO NOTHING

Table 14-13 UPSERT parameters

Parameter	Description	Value Range or Example
ON CONFLICT (...)	The system determines the primary key or unique index by analyzing the given column or constraint names for a specific index's syntax. You can specify a single column name or multiple column names by using an expression, for example, (column1, column2, column3). collation and opclass can be specified when you create an index. Therefore, you can also specify them after the column name for index inference. COLLATE collation specifies the collation of a column, and opclass specifies the name of the operator class. For details, see CREATE INDEX. When inferring the unique index from an expression that includes multiple column names, the system checks whether there is a unique index that exactly contains all the column names specified by conflict_target. • If collation and opclass are not specified, a match is considered found as long as a column has the same name as the specified single column or multiple columns have the same names as those specified by the column expression (regardless of the values of collation and opclass specified for the index column). • If collation and opclass are specified, their values must also match the collation and opclass of the index.	-

Parameter	Description	Value Range or Example
UPDATE clause	The UPDATE clause can use VALUES(colname) or EXCLUDED.colname to reference inserted data. EXCLUDED indicates the data row that should be excluded due to conflict. For details about the syntax example, see Examples .	-
WHERE clause	- The WHERE clause is used to determine whether a specified condition is met when data conflict occurs. If yes, update the conflicting data. Otherwise, ignore the conflict. - Only syntax 2 of Update Data Upon Conflict can specify the WHERE clause, that is, INSERT INTO ON CONFLICT(...) DO UPDATE SET WHERE.	-

Examples

Create table **reason_t2** and insert data into it.

```
DROP TABLE IF EXISTS reason_t2;
CREATE TABLE reason_t2
(
  a  int primary key,
  b  int,
  c  int
);

INSERT INTO reason_t2 VALUES (1, 2, 3);
SELECT * FROM reason_t2 ORDER BY 1;
a | b | c
---+---+---
1 | 2 | 3
(1 rows)
```

Insert two data records into the table **reason_t2**. One data record conflicts and the other does not. Conflicting data is ignored, and non-conflicting data is inserted.

```
INSERT INTO reason_t2 VALUES (1, 4, 5),(2, 6, 7) ON CONFLICT(a) DO NOTHING;
SELECT * FROM reason_t2 ORDER BY 1;
a | b | c
---+---+---
1 | 2 | 3
2 | 6 | 7
(2 rows)
```

Insert two data records into the table **reason_t2**. One data record conflicts and the other does not. Update conflicting information and add new data where there is no conflict. **EXCLUDED.b** and **EXCLUDED.c** refer to columns **b** and **c** for the inserted values, which are 4 and 5 in this case.

```
INSERT INTO reason_t2 VALUES (1, 4, 5),(3, 8, 9) ON CONFLICT(a) DO UPDATE SET b = EXCLUDED.b, c = EXCLUDED.c;
```

```
SELECT * FROM reason_t2 ORDER BY 1;
a | b | c
---+---+---
1 | 4 | 5
2 | 6 | 7
3 | 8 | 9
(3 rows)
```

Filter the updated rows.

```
INSERT INTO reason_t2 VALUES (2, 7, 8) ON CONFLICT (a) DO UPDATE SET b = excluded.b, c = excluded.c
WHERE reason_t2.c = 7;
SELECT * FROM reason_t2 ORDER BY 1;
a | b | c
---+---+---
1 | 4 | 5
2 | 7 | 8
3 | 8 | 9
(3 rows)
```

Insert data into the table **reason_t**. Update the conflicting data and adjust the mapping. That is, update column c to column b and column b to column c.

```
INSERT INTO reason_t2 VALUES (1, 2, 3) ON CONFLICT (a) DO UPDATE SET b = excluded.c, c = excluded.b;
SELECT * FROM reason_t2 ORDER BY 1;
a | b | c
---+---+---
1 | 3 | 2
2 | 7 | 8
3 | 8 | 9
(3 rows)
```

FAQs

Multiple updates are not supported. When conflicting data groups exist, you will see an error like "INSERT ON CONFLICT DO UPDATE command cannot affect row a second time." This does not happen if the query uses a PGXC plan. Below is an example.

```
CREATE TABLE t1(id int PRIMARY KEY, a int, b int);
SET explain_perf_mode=pretty;
-- Use the stream query plan:
EXPLAIN (COSTS OFF) INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a =
EXCLUDED.a + 1;
          QUERY PLAN
-----
id |          operation
---+-----
1 | -> Streaming (type: GATHER)
2 | -> Insert on t1
3 | -> Streaming(type: REDISTRIBUTE)
4 | -> Values Scan on ""VALUES""
Predicate Information (identified by plan id)
-----
2 --Insert on t1
   Conflict Resolution: UPDATE
   Conflict Arbiter Indexes: t1_pkey
===== Query Summary =====
-----
System available mem: 819200KB
Query Max mem: 819200KB
Query estimated mem: 3104KB
(18 rows)
INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a = EXCLUDED.a + 1;
ERROR: dn_xxxx: INSERT ON CONFLICT DO UPDATE command cannot affect row a second time
HINT: Ensure that no rows proposed for insertion within the same command have duplicate constrained
values.
-- Disable the stream plan and generate a PGXC plan:
```

```
set enable_stream_operator = off;
EXPLAIN (COSTS OFF) INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a =
EXCLUDED.a + 1;
      QUERY PLAN
-----
id |      operation
---+-----
 1 | -> Insert on t1
 2 | -> Values Scan on "VALUES*"
Predicate Information (identified by plan id)
-----
 1 --Insert on t1
    Conflict Resolution: UPDATE
    Conflict Arbiter Indexes: t1_pkey
    Node expr: id
(11 rows)
INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a = EXCLUDED.a + 1;
INSERT 0 2
```

14.10 UPDATE

Function

UPDATE updates data in a table. **UPDATE** changes the values of the specified columns in all rows that satisfy the condition. The **WHERE** clause clarifies conditions. The columns to be modified need be mentioned in the **SET** clause; columns not explicitly modified retain their previous values.

Precautions

- For how to use UPDATE, see [UPDATE and DELETE Operations](#).
- Permission requirements: To modify a table, you must have the UPDATE permission on the table and the SELECT permission on any table involved in the expression or condition.
- The distribution column of a table cannot be modified.
- You are not advised to create a table that needs to be frequently updated as a replication table.
- Constraints on column-store tables:
 - Column-store tables do not support the **RETURNING** clause.
 - Column-store tables do not support non-deterministic update. If you update data in one row with multiple rows of data in a column-store table, an error is reported.
 - Memory space that records update operations in column-store tables is not reclaimed. You need to clean it by executing **VACUUM FULL table_name**.
 - Column-store tables support lightweight **UPDATE** operations, which only rewrite the updated columns to reduce space usage. To enable column-store lightweight **UPDATE**, configure **enable_light_colupdate**. It is disabled by default.
 - Column-store lightweight **UPDATE** is unavailable and automatically changes to the regular **UPDATE** operation in the following scenarios: updating an index column, updating a primary key column, updating a partition column, updating a PCK column, and online scaling.

- There is a low probability that an error is reported when column-store lightweight **UPDATE** and backend column-store **AUTOVACUUM** coexist. To avoid such an issue, you can run **ALTER TABLE** to set the table-level parameter **enable_column_autovacuum_garbage** to **off** (**off** means that the backend column-store **AUTOVACUUM** of tables is disabled).

Syntax

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
[ PARTITION ( partition_name ) | PARTITION FOR ( partition_key_value [, ...] ) ]
SET {column_name = { expression | DEFAULT }
| ( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] ) |sub_query }}[, ...]
[ FROM from_list ] [ WHERE condition ]
[ RETURNING {*
| {output_expression [ [ AS ] output_name ]} [, ...] }];
```

where sub_query can be:

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
```

Parameter Description

Table 14-14 UPDATE parameters

Parameter	Description	Example
plan_hint clause	Following the keyword in the /*+ */ format, hints are used to optimize the plan generated by a specified statement block. For details, see Hint-based Tuning .	-
table_name	Specifies the name (optionally schema-qualified) of the table to be updated.	test.table1 (test is the schema name).
[AS] alias	Specifies a short alias for the table for easy reference. Its value is a string, which must comply with the naming convention. For details, see Identifier Naming Conventions .	Update the employees table with alias e : UPDATE employees AS e SET e.salary = e.salary * 1.1 WHERE e.department = 'technology';
partition_name	Specifies the partition name. If this parameter is specified, the data corresponding to the partition name will be updated. It is available for range partitioned tables, list partitioned tables, and other named partitioned tables. Only clusters of version 8.2.1 or later support this option.	Update data in the partition p1 : UPDATE test_range_row PARTITION(p1) SET d = 41;

Parameter	Description	Example
partition_key_value	Specifies the partition key value. If this parameter is specified, the data corresponding to the partition key value will be updated The value specified by PARTITION FOR (partition_key_value [, ...]) can uniquely identify a partition.	Update the value of b to 20 for the partition with key 41: <pre>UPDATE test_range_row PARTITION FOR (41) SET b = 20;</pre>
column_name	Renames a column. You can refer to this column by specifying the table name and column name of the target table. Example: <pre>UPDATE foo SET foo.col_name = 'GaussDB';</pre> You can refer to this column by specifying the target table alias and the column name. Example: <pre>UPDATE foo AS f SET f.col_name = 'GaussDB';</pre>	-
expression	An expression or value to assign to the column.	Discount the product price by 20%: <pre>UPDATE products SET price = price * 0.8 WHERE product_id = 101;</pre>
DEFAULT	Sets the column to its default value. The value is NULL if no specified default value has been assigned to it.	Reset the status and login times of inactive users: <pre>UPDATE users SET status = DEFAULT, login_count = DEFAULT WHERE last_login < CURRENT_DATE - INTERVAL '30 days';</pre>
sub_query	Specifies the subquery, which is used to update the target table based on the query results of other tables in the same database. For details about clauses in the SELECT statement, see SELECT.	Update the average salary of a department using a subquery. <pre>UPDATE departments d SET avg_salary = (SELECT AVG(salary) FROM employees e WHERE e.dept_id = d.dept_id);</pre>

Parameter	Description	Example
from_list	Specifies a table expression, that allows the columns from other tables to appear in the WHERE condition and update the target table. This is similar to the list of tables that can be specified in the FROM clause of a SELECT statement. NOTICE Note that the target table must not appear in the from_list, unless you intend a self-join (in which case it must appear with an alias in the from_list).	Example: Use the FROM from_list Clause to Preferentially Process Orders of Customers with High VIP Levels
condition	Specifies an expression that follows WHERE and returns a result of the boolean type. Only rows for which this expression returns true are updated.	Update the values of the students whose math scores are lower than 80: <pre>UPDATE student_scores SET math_score = math_score + 5, update_time = NOW() WHERE math_score < 80;</pre>
output_expression	Specifies an expression to be computed and returned by the UPDATE command after each row is updated. Value range: any table and table columns listed in FROM. The asterisk (*) indicates that all columns are returned.	Update and return the stock information before and after the update: <pre>UPDATE inventory SET stock = stock - 10 WHERE category = 'Electronics' RETURNING item_id AS "Product ID", name AS "Product name", stock AS "New stock", stock + 10 AS "Old stock";</pre>
output_name	Specifies the name of the column returned by RETURNING to optimize the output. Its value is a string, which must comply with the naming convention. For details, see Identifier Naming Conventions.	-

Examples: Common UPDATE Scenarios

Create a sample table and insert data into it.

```
DROP TABLE IF EXISTS products;
CREATE TABLE products (
  product_id INT PRIMARY KEY,    --Product ID
```

```
name VARCHAR(100),           --Product name
price DECIMAL(10,2),         --Product price
stock INT,                   --Product stock
last_updated TIMESTAMP       --Data update time
);
```

```
INSERT INTO products VALUES
(101, 'Phone', 2999.00, 50, NULL),
(102, 'PC', 5999.00, 30, NULL),
(103, 'PAD', 1999.00, 20, NULL);
```

Increase the price of all products by 20% (using a statement without **WHERE** to update the values of all records).

```
UPDATE products SET price = price * 1.2 , last_updated = NOW();
SELECT * FROM products ORDER BY 1;
```

product_id	name	price	stock	last_updated
101	Phone	3598.80	50	2025-05-27 21:44:13.759399
102	PC	7198.80	30	2025-05-27 21:44:13.759399
103	PAD	2398.80	20	2025-05-27 21:44:13.759399

Increase the price of the PAD by 500 (using a statement with **WHERE** to update the specified data).

```
UPDATE products SET price = price + 500 , last_updated = NOW()
WHERE name = 'PAD';
SELECT * FROM products ORDER BY 1;
```

product_id	name	price	stock	last_updated
101	Phone	3598.80	50	2025-05-27 21:44:13.759399
102	PC	7198.80	30	2025-05-27 21:44:13.759399
103	PAD	2898.80	20	2025-05-27 21:47:20.690807

Discount the price of products whose stock is greater than 40 by 20% (using a statement with **WHERE** to update data in a specified range).

```
UPDATE products SET price = price * 0.8 , last_updated = NOW()
WHERE stock > 40;
SELECT * FROM products ORDER BY 1;
```

product_id	name	price	stock	last_updated
101	Phone	2879.04	50	2025-05-27 21:53:31.996656
102	PC	7198.80	30	2025-05-27 21:44:13.759399
103	PAD	2898.80	20	2025-05-27 21:47:20.690807

Adjust the price and stock of the PC (Specifying multiple values in **SET**)

```
UPDATE products SET price = price - 1000 , stock = stock - 10, last_updated = NOW()
WHERE name = 'PC';
SELECT * FROM products ORDER BY 1;
```

product_id	name	price	stock	last_updated
101	Phone	2879.04	50	2025-05-27 21:53:31.996656
102	PC	6198.80	20	2025-05-27 22:00:07.851013
103	PAD	2898.80	20	2025-05-27 21:47:20.690807

Example: Update a Specified Partition of a Partitioned Table

Create a sample table and import data into it.

```
DROP TABLE IF EXISTS test_range_row;
CREATE TABLE test_range_row(a int, b int,d int)
DISTRIBUTE BY hash(a) PARTITION BY RANGE(d)
(
  PARTITION p1 values LESS THAN (60),
  PARTITION p2 values LESS THAN (75),
  PARTITION p3 values LESS THAN (90),
  PARTITION p4 VALUES LESS THAN (maxvalue)
);

INSERT INTO test_range_row PARTITION(p1) VALUES(55,20,51);
INSERT INTO test_range_row PARTITION(p1) VALUES(54,18,59);
INSERT INTO test_range_row PARTITION(p2) VALUES(15,18,63);
INSERT INTO test_range_row PARTITION(p3) VALUES(12,21,80);
INSERT INTO test_range_row PARTITION(p3) VALUES(83,21,85);
INSERT INTO test_range_row PARTITION(p4) VALUES(36,21,97);

SELECT * FROM test_range_row ORDER BY d;
```

a	b	d
55	20	51
54	18	59
15	18	63
12	21	80
83	21	85
36	21	97

Total Records: 6

Change the values of column **d** to **41** in partition **p1**.

```
UPDATE test_range_row PARTITION(p1) SET d = 41;
SELECT * FROM test_range_row ORDER BY d;
```

a	b	d
54	18	41
55	20	41
15	18	63
12	21	80
83	21	85
36	21	97

Total Records: 6

According to the preceding result, the partition is **p1**, where the values of column **d** are less than **60**. The values of column **d** in partition **p1** are changed to **41**.

Update the values of column **b** to **20** in the partition where the partition key value is 41.

```
UPDATE test_range_row PARTITION FOR (41) SET b = 20;  
SELECT * FROM test_range_row ORDER BY d;
```

a	b	d
54	20	41
55	20	41
15	18	63
12	21	80
83	21	85
36	21	97

Total Records: 6

Example: Use the FROM from_list Clause to Preferentially Process Orders of Customers with High VIP Levels

The target table needs to be updated. The orders table needs to be updated, but the **FROM** clause needs to be used to associate the **customer ID** field in the **customers** table to obtain customer VIP levels.

Create the customer information table and order table, which contain the following fields:

```
DROP TABLE IF EXISTS customers;
CREATE TABLE customers (          --Customer information table to be created
  customer_id INT PRIMARY KEY,    -- Customer ID
  name VARCHAR(100),             --Customer name
  vip_level INT                   --VIP level
);
DROP TABLE IF EXISTS orders;
CREATE TABLE orders (            --Order table to be created
  order_id INT PRIMARY KEY,       --Order ID
  customer_id INT,               --Customer ID
  amount DECIMAL(10,2),          --Consumption amount
  status VARCHAR(20)             --Order status
);
```

Insert test data and ensure the order status of customers is pending (It means that the orders of the three customers are not processed.)

```
INSERT INTO customers VALUES
(1,'Tom', 1)
(2,'Jack', 3),
(3,'Tony', 2);
INSERT INTO orders VALUES
(1001, 1, 500.00, 'pending'),
(1002, 2, 1200.00, 'pending'),
(1003, 3, 800.00, 'pending');
SELECT * FROM orders;
```

order_id	customer_id	amount	status
1002	2	1200.00	pending
1001	1	500.00	pending
1003	3	800.00	pending

Total Records: 3

Orders of customers with high VIP levels need be prioritized when there are too many orders. **FROM from_list** is used to associate the customer ID in the customer table to obtain the customer VIP level. In this example, orders of customers with VIP level 3 or higher need to be processed first.

```
UPDATE orders o
SET status = 'priority'    --Set the order status to priority.
FROM customers c
WHERE o.customer_id = c.customer_id AND c.vip_level >= 3; --Obtain the customers whose VIP levels are
                                                             higher than 3 from the customer table.
SELECT * FROM orders;
```

order_id	customer_id	amount	status
1002	2	1200.00	priority
1001	1	500.00	pending
1003	3	800.00	pending

Total Records: 3

Example: Use DEFAULT to Reset the Status of an Inactive User

Create a user table (with DEFAULT) and insert data.

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
  user_id INT PRIMARY KEY,      --User ID
  username VARCHAR(50),        --Username
  login_count INT DEFAULT 0,    -- Number of login times. The default value is 0.
  last_login TIMESTAMP,        --Last login time
  status VARCHAR(20) DEFAULT 'active'; --User status. The default value is active.
)

INSERT INTO users (user_id, username, login_count, last_login) VALUES
(1, 'user1', 5, '2023-06-01 10:00:00'),
(2, 'user2', 3, '2023-06-15 14:30:00');
SELECT * FROM users;
```

user_id	username	login_count	last_login	status
2	user2	3	2023-06-15 14:30:00	active
1	user1	5	2023-06-01 10:00:00	active

Total Records: 2

Reset the status and login times of inactive users:

```
UPDATE users SET
status = DEFAULT,      -- Reset the status to active
login_count = DEFAULT -- Reset the login times to 0
WHERE last_login < CURRENT_DATE - INTERVAL '30 days';
SELECT * FROM users;
```

user_id	username	login_count	last_login	status
1	user1	0	2023-06-01 10:00:00	active
2	user2	0	2023-06-15 14:30:00	active

FAQs

- [Error "abort transaction due to concurrent update" Is Reported During SQL Execution](#)

- [How Do I Determine Whether UPDATE or DELETE Has Been Executed on a Table?](#)
- ["ERROR: Non-deterministic UPDATE" Is Reported During Update](#)

14.11 VALUES

Function

VALUES computes a row or a set of rows based on given values. It is most commonly used to generate a constant table within a large command.

Precautions

- **VALUES** lists with large numbers of rows should be avoided, as you might encounter out-of-memory failures or poor performance. **VALUES** appearing within **INSERT** is a special case, because the desired column types are known from the **INSERT**'s target table, and need not be inferred by scanning the **VALUES** list. In this case, **VALUE** can handle larger lists than are practical in other contexts.
- If more than one row is specified, all the rows must have the same number of elements.

Syntax

```
VALUES {( expression [, ...] )} [, ...]  
[ ORDER BY { sort_expression [ ASC | DESC | USING operator ] } [, ...] ]  
[ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]  
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ];
```

Parameter Description

Table 14-15 VALUES parameters

Parameter	Description	Value Range
expression	Specifies a constant or expression to compute and insert at the indicated place in the resulting table or set of rows. In a VALUES list appearing at the top level of an INSERT , an expression can be replaced by DEFAULT to indicate that the destination column's default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.	-
sort_expression	Specifies an expression or integer constant indicating how to sort the result rows.	-
ASC	Indicates ascending sort order.	-

Parameter	Description	Value Range
DESC	Indicates descending sort order.	-
operator	Specifies a sorting operator.	-
count	Specifies the maximum number of rows to return.	-
start	Specifies the number of rows to skip before starting to return rows.	-
FETCH { FIRST NEXT } [count] { ROW ROWS } ONLY	The FETCH clause restricts the total number of rows starting from the first row of the return query result.	The default value is 1 .

Inserting One or More Data Records

Create the **reason_t1** table.

```
DROP TABLE IF EXISTS reason_t1;  
CREATE TABLE reason_t1  
(  
    TABLE_SK    INTEGER    ,  
    TABLE_ID    VARCHAR(20) ,  
    TABLE_NA    VARCHAR(20)  
);
```

Insert one data record to the table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

Insert one data record into the table. This command is equivalent to the last one.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA');
```

Insert records whose **TABLE_SK** is less than **1** into the table.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

Insert records into the table.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');  
SELECT * FROM reason_t1 ORDER BY 1;
```

table_sk	table_id	table_na
1	S01	StudentA
1	S01	StudentA
1	S01	StudentA
2	T01	TeacherA
3	T02	TeacherB

15 DCL Syntax

15.1 DCL Syntax Overview

Data control language (DCL) is used to set or modify database users or role rights.

Authorization

DWS provides a statement for granting rights to data objects and roles. For details, see [GRANT](#).

Revoking Rights

DWS provides a statement for revoking rights. For details, see [REVOKE](#).

Setting Default Rights

DWS allows users to set rights for objects that will be created. For details, see [ALTER DEFAULT PRIVILEGES](#).

15.2 ALTER DEFAULT PRIVILEGES

Function

ALTER DEFAULT PRIVILEGES allows you to set the permissions that will be used for objects to be created. It does not affect permissions assigned to existing objects.

A user can modify only the default permissions of objects created by the user or the role to which the user belongs. These permissions can be set globally (all objects created in the database) or for objects in a specified schema.

To view information about the default permissions of database users, query the system catalog **PG_DEFAULT_ACL**.

Precautions

- Only the permissions for tables (including views), sequences, functions, and types (including domains) can be altered.
- The VACUUM, DROP, and ALTER permissions on foreign tables cannot be granted to users.

Syntax

```
ALTER DEFAULT PRIVILEGES
[ FOR { ROLE | USER } target_role [, ...] ]
[ IN SCHEMA schema_name [, ...] ]
abbreviated_grant_or_revoke;
```

The **abbreviated_grant_or_revoke** clause is used to specify the objects for which permissions are granted or revoked. The options include:

- grant_on_tables_clause
- grant_on_functions_clause
- grant_on_types_clause
- grant_on_sequences_clause
- revoke_on_tables_clause
- revoke_on_functions_clause
- revoke_on_types_clause
- revoke_on_sequences_clause

grant_on_tables_clause grants permissions on tables.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE | ANALYSE
| VACUUM | ALTER | DROP }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```

grant_on_functions_clause grants permissions on functions.

```
GRANT { { EXECUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
ON FUNCTIONS
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```

grant_on_types_clause grants permissions on types.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON TYPES
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```

grant_on_sequences_clause grants permissions on sequences.

```
GRANT { { USAGE | SELECT | UPDATE }
[, ...] | ALL [ PRIVILEGES ] }
ON SEQUENCES
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```

revoke_on_tables_clause revokes permissions on tables.

```
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE | ANALYSE |
VACUUM | ALTER | DROP }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

revoke_on_functions_clause revokes permissions on functions.

```
REVOKE [ GRANT OPTION FOR ]  
  { { EXECUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }  
  ON FUNCTIONS  
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
  [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

revoke_on_types_clause revokes permissions on types.

```
REVOKE [ GRANT OPTION FOR ]  
  { USAGE | ALL [ PRIVILEGES ] }  
  ON TYPES  
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
  [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

revoke_on_sequences_clause revokes permissions on sequences.

```
REVOKE [ GRANT OPTION FOR ]  
  { { USAGE | SELECT | UPDATE }  
    [, ...] | ALL [ PRIVILEGES ] }  
  ON SEQUENCES  
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
  [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

Parameter Description

Table 15-1 ALTER DEFAULT PRIVILEGES parameters

Parameter	Description	Value Range
target_role	Name of an existing role. If FOR ROLE/USER is omitted, the current role or user is assumed. target_role must have the CREATE permissions for schema_name . You can use the has_schema_privilege function to check whether a role or user has the CREATE permission on a schema. <pre>SELECT a.rolname, n.nspname FROM pg_authid as a, pg_namespace as n WHERE has_schema_privilege(a.oid, n.oid, 'CREATE');</pre>	Name of an existing role.
schema_name	Name of an existing schema. If a schema name is specified, the default permissions of all objects created in this schema will be modified. If IN SCHEMA is omitted, global permissions are modified.	Name of an existing schema.
role_name	Name of an existing role whose permissions are to be granted or revoked. NOTE If you want to delete a role that is granted with default permissions, you must revoke the default permissions or use DROP OWNED BY to delete the default permission records of the role.	Name of an existing role.

Examples

- Grant the **SELECT** permission on all the tables (and views) in **tpcds** to every user.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT SELECT ON TABLES TO PUBLIC;
```

- Grant the **INSERT** permission on all the tables in **tpcds** to user **jack**.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT INSERT ON TABLES TO jack;
```

- Revoke the preceding permissions.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE SELECT ON TABLES FROM PUBLIC;  
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE INSERT ON TABLES FROM jack;
```

- Assume that there are two users **test1** and **test2**. If you require that user **test2** can query tables created by user **test1**, execute the following statements:

- Grant the schema permission of user **test1** to **test2**.

```
GRANT usage, create ON SCHEMA test1 TO test2;
```

- Grant user **test2** the permission to query tables of user **test1**.

```
ALTER DEFAULT PRIVILEGES FOR USER test1 IN SCHEMA test1 GRANT SELECT ON tables TO test2;
```

- Create a table as user **test1**.

```
SET ROLE test1 password '{password}';  
CREATE TABLE test3( a int, b int);
```

- Enable the **test2** user to perform the query.

```
SET ROLE test2 password '{password}';  
SELECT * FROM test1.test3;  
a | b  
---+---  
(0 rows)
```

Helpful Links

[GRANT](#), [REVOKE](#)

15.3 ANALYZE | ANALYSE

Function

This syntax gathers statistics on table content within a database. These statistics include both table-level and column-level details. Table-level data is kept in **PG_CLASS**, while column-level details are stored in **PG_STATISTIC**. The query planner relies on this information to choose the best execution strategy.

Its key purposes are:

- Optimizing query performance: supplies data distribution insights to the query optimizer for generating efficient plans.
- Enhancing plan accuracy: helps the optimizer better estimate the cost of various query paths.
- Selecting indexes: assists the optimizer in deciding if and which indexes to use.

Scenarios Requiring the ANALYZE Operation

- Significant data changes**

Significant data changes, such as importing, updating, or deleting large amounts of data, alter row counts and data characteristics, requiring updated table statistics.

- **Newly added data querying**

Daily updates to a service table with fresh data may lead to inaccurate queries if statistics are not refreshed promptly. The following tables have the characteristics of this scenario:

- Service tables that grow over time.
- Service tables that have daily data updates.
- Tables with fresh data that needs to be processed and analyzed.

Typically, after importing data, users process and analyze it using filters on partitioned tables.

```
passtime > '2020-01-19 00:00:00' AND passtime < '2020-01-20 00:00:00'
```

Without running **ANALYZE** after importing new data, the optimizer finds that the **passtime** filter range exceeds the recorded statistical limits. This leads to an incorrect row estimate, showing only one tuple matches the condition where **passtime** falls between 2020-01-19 00:00:00 and 2020-01-20 00:00:00.

Precautions

- If no parameters are specified, **ANALYZE** analyzes each table and partitioned table in the database. You can also specify **table_name**, **column**, and **partition_name** to limit the analysis to a specified table, column, or partitioned table.
- Users who can execute **ANALYZE** on a specific table include the owner of the table, owner of the database where the table is, users with the **ANALYZE** permission on the table, users who are granted the **gs_role_analyze_any** role, and users with the **SYSADMIN** attribute.
- To collect statistics using percentage sampling, you must have the **ANALYZE** and **SELECT** permissions.
- If the **enable_analyze_partition** parameter is enabled and the table-level **incremental_analyze** parameter is set for a partitioned table, the **ANALYZE** statement is executed on partitions lacking statistics or with data changes. The statistics of the partition main table are then generated by combining the partition statistics.
- Only cluster versions 8.1.1 and later support using anonymous blocks, transaction blocks, functions, or stored procedures to perform the **ANALYZE** operation on an unsharded table.
- However, for analyzing an entire database, the **ANALYZE** operation of each table is in different transactions. Therefore, the current version does not support the **ANALYZE** execution for the entire database in anonymous blocks, transaction blocks, functions, or stored procedures.
- Statistics updates of PG_CLASS related columns cannot be rolled back.
- The **ANALYZE** statistics of global temporary tables in each session are independent. Each session has its own statistics.
- **ANALYZE** and **ANALYZE VERIFY** are used to check whether data files of common tables (row-store and column-store tables) in a database are damaged. Currently, this function does not support HDFS tables.

- Most **ANALYZE VERIFY** operations are used for abnormal scenario detection, and require a release version. Remote read is not triggered in the **ANALYZE VERIFY** scenario. Therefore, the remote read parameter does not take effect. If the system detects that the page is damaged due to an error in the key system table, the system reports an error and stops the detection.

**WARNING**

- If more than 10% of a table's records are added or modified at once, explicitly execute the **ANALYZE** operation.
- For more information about development and design proposal, see [Development and Design Proposal](#).

Syntax

- Collect statistics information about a table.

```
{ ANALYZE | ANALYSE } [ { { VERBOSE | LIGHT | FORCE | PREDICATE } }  
[ table_name [ ( column_name [, ...] ) ] ] ;
```
- Collect statistics about a partition.

```
{ ANALYZE | ANALYSE } [ { { VERBOSE | LIGHT | FORCE } }  
[ table_name [ ( column_name [, ...] ) ] ]  
PARTITION ( partition_name ) ;
```

 - Ordinary partitioned tables support the syntax for collecting statistics on a specific partition, but not the actual function of it. **ANALYZE** on a specified partition will cause a warning message.
 - Temporary sampling tables cannot be used to collect partition statistics.
 - Multi-column statistics and expression statistics on partitions are not supported.
- Collect statistics about a foreign table.

```
{ ANALYZE | ANALYSE } [ VERBOSE ]  
{ foreign_table_name | FOREIGN TABLES } ;
```
- Collect statistics about multiple columns.

```
{ ANALYZE | ANALYSE } [ VERBOSE ]  
table_name (( column_1_name, column_2_name [, ...] )) ;
```

 - To sample data in percentage, set **default_statistics_target** to a negative number.
 - The statistics about a maximum of 32 columns can be collected at a time.
 - You are not allowed to collect statistics about multiple columns in system catalogs, global temporary tables, or HDFS foreign tables.
- Check the data files in the current database.

```
{ ANALYZE | ANALYSE } VERIFY { FAST | COMPLETE } ;
```

 - All operations on the database are supported. Because many tables are involved, you are advised to save the result in redirection mode: **gsql -d database -p port -f "verify.sql"> verify_warning.txt 2>&1**.
 - HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.
 - Only the NOTICE message is displayed for external tables. The detection of internal tables is included in the external tables that depend on them, but it is not shown externally.

- This command can be used to process tolerant errors. The assert operation in a debug version may cause the core to fail to execute commands. Therefore, you are advised to perform this operation in a release version.
- If a key system table is damaged during a full database operation, an error is reported and the operation stops.
- Check data files of tables and indexes.
`{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name[index_name] [CASCADE];`
 - You can perform operations on common tables and index tables, but cannot perform **CASCADE** operations on index tables. The reason is that **CASCADE** is used to process all index tables of the primary table. When the index table is checked separately, **CASCADE** is not required.
 - HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.
 - When the primary table is checked, the internal tables of the primary table, such as the **toast** table and **cudesc** table, are also checked.
 - When the system displays a message indicating that the index table is damaged, you are advised to run the **reindex** command to recreate the index.
- Check the data files of the table partition.
`{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name PARTITION {(partition_name)} [CASCADE];`
 - You can detect a single partition of a table, but cannot perform the **CASCADE** operation on index tables.
 - HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.

Parameter Description

Table 15-2 ANALYZE | ANALYSE parameters

Parameter	Description	Value Range
VERBOSE	After VERBOSE is enabled, ANALYZE sends progress information, indicating the table being processed. Statistics about the table are also printed.	-
LIGHT	In LIGHT mode, the system stores table statistics in memory instead of writing them to the system catalog. Executing this command locks the table at a certain level. Specify this option within parentheses, like ANALYZE (LIGHT) t1 .	-
FORCE	In FORCE mode, you can override locks on table statistics and run ANALYZE . Add this option in parentheses like this: ANALYZE (FORCE) t1 .	-

Parameter	Description	Value Range
PREDICATE	PREDICATE indicates that only the statistics of the currently identified predicate column are calculated. This parameter is supported only by clusters of version 9.1.0.100 or later. You can set the analyze_predicate_column_threshold parameter to determine whether the ANALYZE operation on predicate columns is supported. A value of 0 indicates that the ANALYZE operation is disabled. A value greater than 0 indicates that the ANALYZE operation is enabled only for tables whose number of columns is greater than or equal to this value. Use this option in parentheses, for example, ANALYZE (PREDICATE) xxx. Predicate information is collected in the query parsing phase. Dynamic sampling also supports predicate column sampling.	-
table_name	Specifies the name (possibly schema-qualified) of a specific table to analyze. If omitted, all regular tables (but not foreign tables) in the current database are analyzed. Currently, you can use ANALYZE to collect statistics from row-store tables, column-store tables, HDFS tables, ORC- and CARBONDATA-formatted OBS foreign tables, and foreign tables for collaborative analysis.	The table name can contain a maximum of 63 characters, including letters, digits, underscores (_), dollar signs (\$), and number signs (#). It must start with a letter or underscore (_).
column_name	Specifies the name of a specific column to analyze. All columns are analyzed by default.	Existing column name (format: column_1_name, column_2_name, column_3_name....)
partition_name	Analyzes the statistics of the partitioned table. Currently the partitioned table supports the syntax of analyzing a partitioned table, but does not execute this syntax.	A string, which must comply with the naming convention of the partition name.
foreign_table_name	Specifies the name (possibly schema-qualified) of a specific table to analyze. The data of the table is stored in HDFS.	A string, which must comply with the naming convention of foreign tables.

Parameter	Description	Value Range
FOREIGN TABLES	Analyzes HDFS foreign tables stored in HDFS and accessible to the current user.	A string, which must comply with the naming rules of HDFS foreign tables.
index_name	Name of the index table to be analyzed, which may contain the schema name.	A string, which must comply with the naming rules of index tables.
FAST COMPLETE	Verifies row-store tables and column-store tables in different modes.	• Row-store tables – In FAST mode, the CRC and page header of the row-store table are verified. If the verification fails, an alarm is generated. – In COMPLETE mode, the pointer and tuple of the row-store table are parsed and verified. • Column-store tables – In FAST mode, the CRC and magic of the column-store table are verified. If the verification fails, an alarm is generated. – In COMPLETE mode, CUs of column-store tables are parsed and verified.
CASCADE	In CASCADE mode, all indexes of the current table are checked.	-

Examples

- Do **ANALYZE** to update statistics in the **customer_info** table:
`ANALYZE customer_info;`
- Do **ANALYZE VERBOSE** to update statistics and display table information in the **customer_info** table:
`ANALYZE VERBOSE customer_info;`
INFO: analyzing "cstore.pg_delta_3394584009"(cn_5002 pid=53078)
INFO: analyzing "public.customer_info"(cn_5002 pid=53078)
INFO: analyzing "public.customer_info" inheritance tree(cn_5002 pid=53078)
`ANALYZE`
- Update the statistics of the **name** and **email** columns in the **customer_info** table.
`ANALYZE customer_info (name, email);`
- Update the statistics of the **customer_info** table in lightweight mode.
`ANALYZE (LIGHT) customer_info;`
- Update the statistics of the **customer_info** table in forcible mode, even if the table is locked.
`ANALYZE (FORCE) customer_info;`
- Update only the statistics of predicate columns. Only the predicate columns collected so far are sampled. The statistics are used after the query service is stable.
`ANALYZE (PREDICATE) customer_info;`

You can use the `pg_stat_get_predicate_columns` function to query the predicate columns of a table. The table name in the following example is **customer_info**. Replace it with the actual table name.

```
SELECT pr.attnum,pa.attname FROM  
pg_catalog.pg_stat_get_predicate_columns('customer_info'::regclass) pr LEFT JOIN pg_attribute pa ON  
pa.attrelid='customer_info'::regclass AND pa.attnum = pr.attnum;
```

15.4 DEALLOCATE

Function

Deallocates a previously prepared statement. If a prepared statement is not explicitly deleted, it is deleted at the end of the session.

For details about prepared statements, see [PREPARE](#).

Precautions

None

Syntax

```
DEALLOCATE [ PREPARE ] { name | ALL };
```

Parameter Description

Table 15-3 DEALLOCATE parameters

Parameter	Description	Value Range
PREPARE	Creates a prepared statement. This keyword is optional and does not affect operations.	-
name	Specifies the name of the prepared statement to be deleted.	A string, which must comply with the naming rules of prepared statements.
ALL	Deallocates all prepared statements.	A string, which must comply with the prepared statement format.

Examples

None

15.5 DO

Function

DO executes an anonymous code block.

A code block is a function body without parameters. Its return type is **void**. It is analyzed and executed at the same time.

Precautions

- Before using a programming language, install it in the current database using **CREATE LANGUAGE**. If no language is specified, **plpgsql** is installed by default.
- To use an untrusted language, you must be a system administrator or have the USAGE permission for programming languages.

Syntax

```
DO [ LANGUAGE lang_name ] code;
```

Parameter Description

Table 15-4 DO parameters

Parameter	Description	Value Range
lang_name	Name of the programming language used to parse code.	Program language that complies with the specifications. The default language is PL/pgSQL.
code	Programming language code that can be executed.	A string.

Examples

Grant user **webuser** all the operation permissions on views in the **tpcds** schema:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT c.relname,n.nsname FROM pg_class c,pg_namespace n
        WHERE c.relnamespace = n.oid AND n.nsname = 'tpcds' AND relkind IN ('r','v')
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO
webuser';
    END LOOP;
END$$;
```

15.6 EXECUTE

Function

EXECUTE executes a prepared statement. Because a prepared statement exists only in the lifetime of a session, the prepared statement must be created by an earlier **PREPARE** statement in the current session.

Precautions

If the **PREPARE** statement creating the prepared statement declares certain parameters, the parameter set transferred to the **EXECUTE** statement must be compatible. Otherwise, an error occurs.

Syntax

```
EXECUTE name [ ( parameter [, ...] ) ] ;
```

Parameter Description

Table 15-5 EXECUTE parameters

Parameter	Description	Value Range
name	Specifies the name of the statement to be executed.	A string, which must comply with the naming rules of prepared statements.
parameter	Specifies a parameter of the prepared statement. It must be an expression that generates a value compatible with the data type specified when the prepared statement is created.	A string, which is an expression of a value that is compatible with the data type of the specified parameter.

Examples

Create and run a prepared statement for the **INSERT** statement:

```
PREPARE insert_reason(integer,character(16),character(100)) AS INSERT INTO tpcds.reason_t1  
VALUES($1,$2,$3);  
EXECUTE insert_reason(52, 'AAAAAAAADDDAAAAAA', 'reason 52');
```

15.7 EXECUTE DIRECT

Function

EXECUTE DIRECT executes an SQL statement on a specified node. Generally, the cluster automatically allocates an SQL statement to proper nodes. **EXECUTE DIRECT** is mainly used for database maintenance and testing.

Precautions

- Only a system administrator can run the **EXECUTE DIRECT** statement.
- To ensure data consistency across nodes, only the **SELECT** statement can be used. Transaction statements, DDL, and DML cannot be used.
- When the AVG aggregation calculation is performed on the specified DN using such statements, the result set is returned in array, for example, {4,2}. The result of sum is 4, and that of count is 2.
- Do not run the **SELECT** statement on nodes where CNs reside because user table data is not stored there.
- **EXECUTE DIRECT** cannot be nested. If the inner SQL statement to be executed is also **EXECUTE DIRECT**, run only the bottom-layer **EXECUTE DIRECT** statement.

Syntax

```
EXECUTE DIRECT ON ( nodename [, ... ] ) query ;
```

Parameter Description

Table 15-6 EXECUTE DIRECT parameters

Parameter	Description	Value Range
nodename	Node name	A string that indicates the name of an existing node.
query	Statement to be executed	A string that indicates a SQL statement that can be executed.

Examples

Query records in table **tpcds.customer_address** on the dn_6001_6002 node:

```
EXECUTE DIRECT ON(dn_6001_6002) 'select count(*) from tpcds.customer_address';
count
-----
16922
(1 row)
```

15.8 GRANT

Functions and Precautions

GRANT grants permissions to roles and users.

GRANT is used in the following scenarios:

Granting System Permissions (SYSADMIN, CREATEDB, CREATEROLE, AUDITADMIN, and LOGIN) to Roles or Users

System permissions are also called user attributes, including **SYSADMIN**, **CREATEDB**, **CREATEROLE**, **AUDITADMIN**, and **LOGIN**.

Table 15-7 System permissions

System Permission	Description	Can Be Granted or Revoked
SYSADMIN	Allows you to view special system catalogs.	Yes
CREATEDB	Allows you to creates databases.	No
CREATEROLE	Allow you to create users and roles.	

System Permission	Description	Can Be Granted or Revoked
AUDITADMIN	Allows you to view audit logs.	
LOGIN	Allows you to connect to databases.	

System permissions are generally specified using **CREATE ROLE** or **ALTER ROLE** syntax.

The **SYSADMIN** permission can be granted and revoked using **GRANT ALL PRIVILEGE** and **REVOKE ALL PRIVILEGE**, respectively.

System permissions cannot be inherited by a user from a role, and cannot be granted using **PUBLIC**.

Granting Database Objects (Tables, Views, Specified Columns, Databases, Functions, and Schemas) to Roles or Users

Precautions

- Do not perform **GRANT** on a table partition. Otherwise, an alarm will be generated.
- To rename an existing object, you must own the object and have the **CREATE** permission on the schema of the object.
- When you grant table or view rights to other users, you also need to grant the **USAGE** permission for the schema that the tables and views belong to. Without this permission, the users granted with the table or view rights can only see the object names, but cannot access them.
- **GRANT** grants specified database object permissions to one or more roles. These permissions are appended to those already granted, if any.
- The key word **PUBLIC** indicates that the permissions are to be granted to all roles, including those that might be created later. **PUBLIC** can be regarded as an implicitly defined group including all roles. Any particular role will have the sum of permissions granted directly to it using **GRANT**, permissions granted to any role it is presently a member of, and permissions granted to **PUBLIC**.
- If **WITH GRANT OPTION** is specified, the recipient of a permission can in turn grant it to others. This option cannot be granted to **PUBLIC**. Only DWS supports this operation.
- DWS grants the permissions for objects of certain types to **PUBLIC**. By default, permissions for tables, table columns, sequences, external data sources, external servers, schemas, and tablespace are not granted to **PUBLIC**. However, permissions for the following objects are granted to **PUBLIC**: **CONNECT** and **CREATE TEMP TABLE** permissions for databases, **EXECUTE** permission for functions, and **USAGE** permission for languages and data types (including domains). An object owner can revoke the default permissions granted to **PUBLIC** and grant permissions to other users as needed. For security purposes, you are advised to create an object and set permissions for

it in the same transaction so that other users do not have time windows to use the object. In addition, you can run the **ALTER DEFAULT PRIVILEGES** statement to modify the initial default permissions.

Granting the Permissions of a Role or User to Others

Grant a role's or user's permissions to one or more roles or users. In this case, every role or user can be regarded as a set of one or more database permissions.

- If **WITH ADMIN OPTION** is specified, the member can in turn grant permissions in the role to others, and revoke permissions in the role as well.
- If a role or user granted with certain permissions is changed or revoked, the permissions inherited from the role or user also change.
- A database administrator can grant permissions to and revoke them from any role or user.
- Roles having **CREATEROLE** permission can grant or revoke membership in any role that is not an administrator.

Permissions

Table 15-8 GRANT permissions

Parameter	Description
SELECT	Allows SELECT from any column, or the specific columns listed, of the specified table, view, or sequence.
INSERT	Allows INSERT of a new row into the specified table.
UPDATE	Allows UPDATE of any column, or the specific columns listed, of the specified table. SELECT ... FOR UPDATE and SELECT ... FOR SHARE also require this permission on at least one column, in addition to the SELECT permission.
DELETE	Allows DELETE of a row from the specified table.
TRUNCATE	Allows TRUNCATE on the specified table.
REFERENCES	To create a foreign key constraint, it is necessary to have this permission on both the referencing and referenced columns.
TRIGGER	To create a trigger, you must have the TRIGGER permission on the table or view.
ANALYZE ANALYSE	To perform the ANALYZE ANALYSE operation on a table to collect statistics data, you must have the ANALYZE ANALYSE permission on the table.

Parameter	Description
CREATE	- For databases, allows new schemas to be created within the database. - For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have the CREATE permission on the schema of the object. - For sub-clusters, allows tables to be created.
CONNECT	Allows the user to connect to the specified database.
TEMPORARY TEMP	Allows temporary tables to be created when the specified database is used.
EXECUTE	Allows the use of the specified function and the use of any operators that are implemented on top of the function.
USAGE	- Procedural languages: It allows the use of the specified language for the creation of functions in that language. - Schemas: It allows access to objects contained in the specified schema. Without this permission, it is still possible to see the object names. - Sequences: It allows the use of the nextval function. - Sub-clusters: It allows users who can access objects contained in the specified schema to access tables in a specified sub-cluster.
COMPUTE	Allows users to perform elastic computing in a computing sub-cluster that they have the compute permission on.
ALTER	Allows you to modify tables, schemas, or functions.
DROP	Allows you to delete tables, schemas, or functions.
VACUUM	Allows VACUUM to be executed on tables.
ALL PRIVILEGES	Grants all of the available permissions at once. Only system administrators have permission to run GRANT ALL PRIVILEGES .

Parameter	Description
WITH GRANT OPTION	If WITH GRANT OPTION is specified, the recipient of a permission can in turn grant it to others. This option cannot be granted to PUBLIC . • WITH GRANT OPTION cannot be used with NODE GROUP. • When using WITH GRANT OPTION, ensure that grant_with_grant_option is set for security_enable_options.
WITH ADMIN OPTION	Specifies whether permission transfer is allowed. If WITH ADMIN OPTION is specified, members of a role can grant membership of the role to others.

Syntax

- Grant the table or view access permission to a specified role or user. Do not perform **GRANT** on a table partition. Otherwise, an alarm will be generated.
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE | ANALYZE | VACUUM | ALTER | DROP } [, ...]
| ALL [PRIVILEGES] }
ON { [TABLE] table_name [, ...]
| ALL TABLES IN SCHEMA schema_name [, ...] }
TO { [GROUP] role_name | PUBLIC } [, ...]
[WITH GRANT OPTION];

CAUTION

Do not perform concurrent GRANT or REVOKE operations (including **ON TABLE** and **ON ALL TABLES IN SCHEMA**) on a given table or view, or do not perform both GRANT or REVOKE and DDL operations on a given table or view. Otherwise, an error similar to "tuple concurrently updated." is reported.

- Grant the column access permission to a specified role or user.
GRANT { { SELECT | INSERT | UPDATE | REFERENCES } (column_name [, ...]) } [, ...]
| ALL [PRIVILEGES] (column_name [, ...]) }
ON [TABLE] table_name [, ...]
TO { [GROUP] role_name | PUBLIC } [, ...]
[WITH GRANT OPTION];
- Grant the schema access permission to a specified role or user.
GRANT { { CREATE | USAGE | ALTER | DROP } [, ...] | ALL [PRIVILEGES] }
ON SCHEMA schema_name [, ...]
TO { [GROUP] role_name | PUBLIC } [, ...]
[WITH GRANT OPTION];

CAUTION

When you grant table or view rights to other users, you also need to grant the USAGE permission for the schema that the tables and views belong to. Without this permission, the users granted with the table or view rights can only see the object names, but cannot access them.

- Grant the database access permission to a specified role or user.

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]  
      | ALL [ PRIVILEGES ] }  
ON DATABASE database_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant a role's permissions to other users or roles.

```
GRANT role_name [, ...]  
TO role_name [, ...]  
[ WITH ADMIN OPTION ];
```

- Grant all permissions to a specified role.

```
GRANT ALL { PRIVILEGES | PRIVILEGE }  
TO role_name;
```

- Grant the domain access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON DOMAIN domain_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

NOTE

The current version does not support granting the domain access permission.

- Grant the external data source access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON FOREIGN DATA WRAPPER fdw_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant the external server access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON FOREIGN SERVER server_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant the function access permission to a specified role or user.

```
GRANT { { EXECUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }  
ON { FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type } [, ...] ) } [, ...]  
    | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant the procedural language access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON LANGUAGE lang_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

NOTE

The current version does not support granting the procedural language access permission.

- Grant the large object access permission to a specified role or user.

```
GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }  
ON LARGE OBJECT loid [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

NOTE

The current version does not support granting the large object access permission.

- Grant the sequence access permission to a specified role or user.

```
GRANT { { SELECT | UPDATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }  
ON { SEQUENCE sequence_name [, ...]  
    | ALL SEQUENCES IN SCHEMA schema_name [, ...] }  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant the sub-cluster access permission to a specified role or user. Common users cannot perform **GRANT** or **REVOKE** operations on node groups.

```
GRANT { CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }  
ON NODE GROUP group_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

- Grant the type access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON TYPE type_name [, ...]  
TO { [ GROUP ] role_name | PUBLIC } [, ...]  
[ WITH GRANT OPTION ];
```

 NOTE

The current version does not support granting the type access permission.

Parameter Description

Table 15-9 GRANT parameters

Parameter	Description	Value Range
role_name	Specifies an existing user name. To obtain the value, see PG_USER .	A string, which must comply with the naming convention. Enter a maximum of 63 characters.
table_name	Specifies an existing table name. To obtain the value, see PG_TABLES .	A string of no more than 63 characters, starting with a letter or underscore (_). The value can contain letters, digits, underscores (_), dollar signs (\$), and number signs (#).
column_name	Specifies an existing column name. To obtain the value, run: <pre>SELECT * FROM PG_GET_TABLEDEF('table name');</pre>	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
schema_name	Specifies an existing schema name. To obtain the value, see PG_NAMESPACE .	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
database_name	Specifies an existing database name. To obtain the value, PG_DATABASE .	A string that complies with the naming convention, which includes uppercase and lowercase letters, digits, underscores (_), and dollar signs (\$). However, it cannot begin with a digit or dollar sign (\$).

Parameter	Description	Value Range
function_name	Specifies an existing function name. To obtain the value, PG_PROC .	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
sequence_name	Specifies a sequence schema name. To obtain the value, view PG_SEQUENCES (supported only by clusters of version 9.1.0 or later).	A string that contains only letters (both uppercase and lowercase), special characters #, \$, and digits.
domain_name	Specifies an existing domain type.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
fdw_name	Specifies an existing foreign data wrapper name.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
lang_name	Specifies an existing language name.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
type_name	Specifies an existing type name.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
group_name	Specifies an existing sub-cluster name.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
argmode	Specifies the parameter mode of a function.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
arg_name	Specifies the parameter name of a function.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
arg_type	Specifies the parameter type of a function.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range
loid	Specifies the identifier of the large object that contains this page.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Examples

- **Grant system permissions to a user or role.**
 - Grant all available permissions of user **sysadmin** to user **joe**:

```
GRANT ALL PRIVILEGES TO joe;
```

Afterward, user **joe** has the sysadmin permissions.
- **Grant object permissions to a user or role.**
 - Grant the SELECT permission on the **tpcds.reason** table to user **joe**:

```
GRANT SELECT ON TABLE tpcds.reason TO joe;
```
 - Grant all permissions of the **tpcds.reason** table to user **kim**.

```
GRANT ALL PRIVILEGES ON tpcds.reason TO kim;
```

After the granting succeeds, user **kim** has all the permissions of the **tpcds.reason** table, including the add, delete, modify, and query permissions.
 - Grant the permission to use the **tpcds** schema to user **joe**.

```
GRANT USAGE ON SCHEMA tpcds TO joe;
```

After the authorization is successful, user **joe** has the **USAGE** permission of the schema and can access the objects contained in the schema.
 - Grant the query permission for the **r_reason_sk**, **r_reason_id**, and **r_reason_desc** columns and the update permission for the **r_reason_desc** column in the **tpcds.reason** table to user **joe**:

```
GRANT select (r_reason_sk,r_reason_id,r_reason_desc),update (r_reason_desc) ON tpcds.reason TO joe;
```

After the granting succeeds, user **joe** immediately has the query permission of the **r_reason_sk** and **r_reason_id** columns in the **tpcds.reason** table.

```
GRANT select (r_reason_sk, r_reason_id) ON tpcds.reason TO joe ;
```
 - Grant the **EXECUTE** permission of the **func_add_sql** function to user **joe**.

```
CREATE FUNCTION func_add_sql(f1 integer,f2 integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
GRANT EXECUTE ON FUNCTION func_add_sql(integer, integer) TO joe;
```
 - Grant the **UPDATE** permission of the sequence **serial** to user **joe**.

```
GRANT UPDATE ON SEQUENCE serial TO joe;
```
 - Grant user **joe** the database connection permission of gaussdb and the permission to create schemas in gaussdb.

```
GRANT create,connect on database gaussdb TO joe ;
```
 - Grant the **tpcds** schema access permission and object creation permission to this role, but do not enable it to grant these permissions to others:

```
GRANT USAGE,CREATE ON SCHEMA tpcds TO tpcds_manager;
```
- **Grant the permissions of a user or role to other users or roles.**

- Grant the permissions of user **joe** to user **manager**, and allow **manager** to grant these permissions to others:
`GRANT joe TO manager WITH ADMIN OPTION;`
- Grant the permissions of user **manager** to user **senior_manager**:
`GRANT manager TO senior_manager;`

Helpful Links

[REVOKE, ALTER DEFAULT PRIVILEGES](#)

FAQs

- [How Do I Grant Table Permissions to a Specified GaussDB\(DWS\) User?](#)
- [How Do I Grant the Permissions of a Schema to a Specified GaussDB\(DWS\) User?](#)
- [How Do I Create a GaussDB\(DWS\) Database Read-Only User?](#)
- [How Do I View the Table Permissions of a GaussDB\(DWS\) User?](#)
- [After the Permission for Querying Tables in a Schema Is Granted to a User, the User Still Cannot Query the Tables](#)
- [How Do I Revoke the Permission of a User If grant select on table t1 to public Has Been Executed on a Table](#)
- [An Error Message Is Displayed When a Common User Executes the Statement for Creating or Deleting a GDS or OBS Foreign Table, Indicating that the User Does Not Have the Permission or the Permission Is Insufficient](#)
- [After the all Permission Is Granted to the Schema of a User, the Error Message "ERROR: current user does not have privilege to role tom" Persists During Table Creation](#)
- [An Error Message Is Reported During Statement Execution, Indicating that the User Does Not Have the Required Permission](#)

15.9 PREPARE

Function

PREPARE creates a prepared statement.

A prepared statement is a performance optimizing object on the server. When the **PREPARE** statement is executed, the specified query is parsed, analyzed, and rewritten. When the **EXECUTE** is executed, the prepared statement is planned and executed. This avoids repetitive parsing and analysis. After the **PREPARE** statement is created, it exists throughout the database session. Once it is created (even if in a transaction block), it will not be deleted when a transaction is rolled back. It can only be deleted by explicitly invoking **DEALLOCATE** or automatically deleted when the session ends.

Precautions

None

Syntax

```
PREPARE name [ ( data_type [, ...] ) ] AS statement;
```

Parameter Description

Table 15-10 PREPARE parameters

Parameter	Description	Value Range
name	Name of a prepared statement. It must be unique in the session.	A string
data_type	Data type of the parameter.	-
statement	SQL statement that can be executed.	Any SELECT INSERT, UPDATE, DELETE, or VALUES statement.

Examples

Create and run a prepared statement for the **INSERT** statement:

```
PREPARE insert_reason(integer,character(16),character(100)) AS INSERT INTO tpccs.reason_t1  
VALUES($1,$2,$3);  
EXECUTE insert_reason(52, 'AAAAAAAADDDAAAAA', 'reason 52');
```

Helpful Links

[DEALLOCATE](#)

15.10 REASSIGN OWNED

Function

REASSIGN OWNED changes the owner of a database.

REASSIGN OWNED requires that the system change owners of all the database objects owned by **old_role** to **new_role**.

Precautions

- **REASSIGN OWNED** is often executed before deleting a rule. Because objects in other databases are not affected, you usually need to run this command in each database that contains the objects owned by the role to be deleted.
- You must have the permissions on the original and target roles to execute it.
- The resource management module does not monitor the data switch of the syntax. You need to call **select gs_wlm_readjust_user_space(0)** to manually calibrate the monitoring data.

Syntax

```
REASSIGN OWNED BY old_role [, ...] TO new_role;
```

Parameter Description

Table 15-11 REASSIGN OWNED parameters

Parameter	Description	Value Range
old_role	Role name of the old owner.	A string of no more than 63 characters. Enter a string starting with a letter or underscore (_) and containing letters, digits, underscores (_), and dollar signs (\$).
new_role	Name of the new role that will become the object owner.	A string of no more than 63 characters. Enter a string starting with a letter or underscore (_) and containing letters, digits, underscores (_), and dollar signs (\$).

Examples

Reassign all database objects owned by the **joe** and **jack** roles to **admin**:

```
REASSIGN OWNED BY joe, jack TO admin;
```

15.11 REVOKE

Function

REVOKE revokes rights from one or more roles.

Precautions

If a non-owner user of an object attempts to **REVOKE** rights on the object, the command is executed based on the following rules:

- If the user has no right whatsoever on the object, the command will fail outright.
- If some permissions are available, the command proceeds, but it revokes only those rights for which the user has grant options.
- The **REVOKE ALL PRIVILEGES** forms will issue an error message if no grant options are held, while the other forms will issue a warning if grant options for any of the rights named in the command are not held.
- Do not perform **REVOKE** to a table partition. Performing **REVOKE** to a partitioned table incurs an alarm.

Syntax

- Revoke the permission of specified table and view.

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE |
    ANALYZE | VACUUM | ALTER | DROP }[, ...]
  | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
    | ALL TABLES IN SCHEMA schema_name [, ...] }
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```



Do not perform concurrent GRANT or REVOKE operations (including **ON TABLE** and **ON ALL TABLES IN SCHEMA**) on a given table or view, or do not perform both GRANT or REVOKE and DDL operations on a given table or view. Otherwise, an error similar to "tuple concurrently updated." is reported.

- Revoke the permission of specified fields on the table.

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )[, ...]
  | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
  ON [ TABLE ] table_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission of a specified database.

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
  | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission of a specified function.

```
REVOKE [ GRANT OPTION FOR ]
  { { EXECUTE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
  ON { FUNCTION {function_name ( [ { [ argmode ] [ arg_name ] arg_type } [, ...] ) } [, ...]
    | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission of a specified large object.

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
  ON LARGE OBJECT loid [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission on a specified sequence.

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | UPDATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
  ON SEQUENCE sequence_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission of a specified schema.

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | USAGE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- Revoke the permission of a specified sub-cluster.

```
REVOKE [ GRANT OPTION FOR ]
  { CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }
```

```
ON NODE GROUP group_name [, ...]  
FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
```

- Revoke the permission of roles based on roles.

```
REVOKE [ ADMIN OPTION FOR ]  
role_name [, ...] FROM role_name [, ...]  
[ CASCADE | RESTRICT ];
```

- Revoke all permissions of a specified role.

```
REVOKE ALL { PRIVILEGES | PRIVILEGE } FROM role_name;
```

Parameter Description

The keyword **PUBLIC** indicates an implicitly defined group that has all roles. For details about the **REVOKE** permission and related parameters, see [Parameter Description](#) in "GRANT".

NOTE

- Permissions of a role include the permissions directly granted to the role, permissions inherited from the parent role, and permissions granted to **PUBLIC**. Therefore, revoking the **SELECT** permission for an object from **PUBLIC** does not necessarily mean that the **SELECT** permission for the object has been revoked from all roles, because the **SELECT** permission directly granted to roles and inherited from parent roles still remains. Similarly, if the **SELECT** permission is revoked from a user and the **PUBLIC** user still has the **SELECT** permission, the user can still use the **SELECT** permission.
- If user A holds the **UPDATE** rights on a table and the **WITH GRANT OPTION** and has granted them to user B, the rights that user B holds are called dependent rights. If the rights or the grant option held by user A is revoked, the dependent rights still exist. Those dependent rights are also revoked if **CASCADE** is specified.
- A user can only revoke rights that were granted directly by that user. If, for example, user A has granted a right with grant option (**WITH ADMIN OPTION**) to user B, and user B has in turned granted it to user C, then user A cannot revoke the right directly from C. However, user A can revoke the grant option held by user B and use **CASCADE**. In this manner, the rights held by user C are automatically revoked. For another example, if both user A and user B have granted the same right to C, A can revoke his own grant but not B's grant, so C will still effectively have the right.
- If the role executing **REVOKE** holds rights indirectly via more than one role membership path, it is unspecified which containing role will be used to execute the command. In such cases, it is best practice to use **SET ROLE** to become the specific role you want to do the **REVOKE** as, and then execute **REVOKE**. Failure to do so may lead to deleting rights not intended to delete, or not deleting any rights at all.

Examples

Revoke all permissions of user **joe**.

```
REVOKE ALL PRIVILEGES FROM joe;
```

Revoke the permissions granted in a specified schema.

```
REVOKE USAGE,CREATE ON SCHEMA tpceds FROM tpceds_manager;
```

Revoke the **CONNECT** privilege from user **joe**.

```
REVOKE CONNECT ON DATABASE mydatabase FROM joe;
```

Revoke the membership of role **admins** from user **joe**.

```
REVOKE admins FROM joe;
```

Revoke all the privileges of user **joe** for the **myView** view.

```
REVOKE ALL PRIVILEGES ON myView FROM joe;
```

Revoke the insert permission for the **customer_t1** table from **public**.

```
REVOKE INSERT ON customer_t1 FROM PUBLIC;
```

Revoke user **joe**'s permission for the **tpcds** schema.

```
REVOKE USAGE ON SCHEMA tpcds FROM joe;
```

Revoke the query permissions for **r_reason_sk** and **r_reason_id** in the **tpcds.reason** table from user **joe**.

```
REVOKE select (r_reason_sk, r_reason_id) ON tpcds.reason FROM joe;
```

Revoke user **joe**'s execute permission on function **func_add_sql**.

```
REVOKE execute ON FUNCTION func_add_sql(integer, integer) FROM joe CASCADE;
```

Links

[GRANT](#)

16 DQL Syntax

16.1 DQL Syntax Overview

Data Query Language (DQL) can obtain data from tables or views.

Query

DWS provides statements for obtaining data from tables or views. For details, see [SELECT](#).

Defining a New Table Based on Query Results

DWS provides a statement for creating a table based on query results and inserting the queried data into the table. For details, see [SELECT INTO](#).

16.2 SELECT

Function

Retrieves data from a table or view.

Serving as an overlaid filter for a database table, **SELECT** using SQL keywords retrieves required data from data tables.

Precautions

- For details about how to use **SELECT**, see [SELECT Operation](#).
- Using **SELECT** can join HDFS and ordinary tables, but cannot join ordinary and GDS foreign tables. That is, a **SELECT** statement cannot contain both ordinary and GDS foreign tables.
- You must have the **SELECT** permission on each column used in the **SELECT** command. If **FOR UPDATE** or **FOR SHARE** is used, the **UPDATE** permission is also required.

Syntax

The syntax format of the main clause is shown below. The syntax formats of the FROM and GROUP BY clauses are defined separately. For details about the FROM and GROUP BY clauses, see [FROM Clause Parameters](#) and [GROUP BY Parameters](#).

NOTE

SELECT supports the simplified query syntax, which is equivalent to **select * from table_name**.

```
TABLE { ONLY {(table_name)| table_name} | table_name [ * ]};
```

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause } [ NULLS { FIRST | LAST } ]} [, ...] ]
[ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

NOTICE

In **condition** and **expression**, you can use the aliases of expressions in the output column in compliance with the following rules:

- Only aliases at the same level can be referenced.
- Only aliases in the output column can be referenced.
- A prior expression is referenced only in a subsequent expression.
- The **volatile** function cannot be used.
- The **Window** function cannot be used.
- Do not reference an alias in the **join on** condition.
- An error is reported if the output column contains multiple referenced aliases.

SELECT Main Clause Parameters

Table 16-1 SELECT parameters

Parameter	Description	Value Range or Example
WITH [RECURSIVE] with_query [, ...]	Specifies one or more subqueries (CTE) that can be referenced by name in the primary query. The subqueries act as temporary tables. If a given subquery result needs to be referenced multiple times, use WITH to simplify the code logic and improve the coding efficiency. If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. The format of with_query: <pre>with_query_name [(column_name [, ...])] AS [[NOT] MATERIALIZED] ({select values insert update delete})</pre>	• with_query_name specifies the name of the result set generated by the subquery. You can use this name to access the result set of the subquery. • [NOT] MATERIALIZED specifies this parameter to change the default behavior. – If MATERIALIZED is specified, the subquery is executed once, and the result set is materialized for the primary query to query the result set for multiple times. – If NOT MATERIALIZED is specified, with_query is referenced only once by the primary query and is not executed separately. Instead,

Parameter	Description	Value Range or Example
		with_query is substituted with its reference in the primary query and executed alongside it. - NOT MATERIALIZED is ignored in the following cases: - The subquery contains volatile functions. - The subquery is a SELECT or VALUES statement containing FOR UPDATE or FOR SHARE. - The subquery is an INSERT, UPDATE, or DELETE statement. - RECURSIVE is specified for with_query. - If with_query2 is

Parameter	Description	Value Range or Example
		referenced more than once and it references with_query1, which referenced itself in the outer layer, with_query2 cannot take the place where it can be referenced. For example, in the following example, tmp2 is referenced twice. Because tmp2 references tmp1 which referenced itself in the outer layer, tmp2 will be materialized even if NOT MATERIALIZED is specified. with recursive tmp1(b) as (values(1) union all

Parameter	Description	Value Range or Example
		<pre>(with tmp2 as not materialized (select * from tmp1) select tt1.b + tt2.b from tmp2 tt1, tmp2 tt2)) select * from tmp1;</pre> • column_name specifies a column name displayed in the subquery result set. • Each subquery can be a SELECT, VALUES, INSERT, UPDATE or DELETE statement.
plan_hint clause	This clause, in the format /*+ */ , comes after the SELECT keyword. It is used to enhance the plan of a SELECT statement block. For details, see Hint-based Tuning .	-
ALL	Specifies that all rows that meet the conditions are returned. This is the default behavior and can be omitted.	-
DISTINCT [ON (expression [, ...])]	Removes all duplicate rows from the SELECT result to ensure that each row is unique. ON (expression [, ...]) is only reserved for the first row among all the rows with the same result calculated using given expressions. NOTICE DISTINCT ON expression is explained with the same rule of ORDER BY . Unless you use ORDER BY to guarantee that the required row appears first, you cannot know what the first row is.	-

Parameter	Description	Value Range or Example
SELECT list	Indicates columns to be queried. Some or all columns (using wildcard character *) can be queried. You may use the AS output_name clause to give an alias for an output column. The alias is used for the displaying of the output column.	Column names may be either of: - Manually input column names which are spaced using commas (,). - Fields computed in the FROM clause.
FROM clause	Indicates one or more source tables for SELECT . For details, see Table 16-2 .	-
WHERE clause	The WHERE clause forms an expression for row selection to narrow down the query range of SELECT .	LIKE operator - The (+) operator is not recommended. For details, see Restrictions on the WHERE Clause .
GROUP BY clause	Condenses query results into a single row or selected rows that share the same values for the grouped expressions.	For details, see Table 16-4 .
HAVING clause	Filters group rows created by GROUP BY . It is similar to WHERE , but is usually used with GROUP BY to filter out special groups. The HAVING clause compares some attributes of groups with a constant. Only groups that match the logical expression in the HAVING clause are extracted.	-

Parameter	Description	Value Range or Example
WINDOW clause	Defines a window function. The format is WINDOW <i>window_name</i> AS (<i>window_definition</i>) [, ...]. <i>window_name</i> indicates a name that can be referenced from subsequent window definitions. <i>window_definition</i> indicates the window rule, which can be expressed in the following format: <pre>[existing_window_name] [PARTITION BY expression [, ...]] [ORDER BY expression [ASC DESC USING operator] [NULLS { FIRST LAST }] [, ...]] [frame_clause]</pre> frame_clause defines a window frame for the window function. The window function (not all window functions) depends on window frame and window frame is a set of relevant rows of the current query row. frame_clause can be expressed in the following forms: <pre>[RANGE ROWS] frame_start [RANGE ROWS] BETWEEN frame_start AND frame_end</pre> frame_start and frame_end can be expressed in the following forms: <pre>UNBOUNDED PRECEDING value PRECEDING (not supported for RANGE) CURRENT ROW value FOLLOWING (not supported for RANGE) UNBOUNDED FOLLOWING</pre> NOTICE For the query of column storage table, only row_number window function is supported, frame_clause is not supported.	-

Parameter	Description	Value Range or Example
UNION clause	Computes the set union of the rows returned by the involved SELECT statements. The UNION clause has the following constraints: • By default, the result of UNION does not contain any duplicate rows unless the ALL option is specified. • Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise specified by parentheses. • FOR UPDATE cannot be specified either for a UNION result or for any input of a UNION. General expression: <pre>select_statement UNION [ALL] select_statement</pre> • select_statement can be any SELECT statement without an ORDER BY, LIMIT, or FOR UPDATE clause. • ORDER BY and LIMIT in parentheses can be attached in a sub-expression.	-
INTERSECT clause	Computes the set intersection of rows returned by the involved SELECT statements. The result of INTERSECT does not contain any duplicate rows. The INTERSECT clause has the following constraints: • Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless otherwise specified by parentheses. • Processing INTERSECT preferentially when UNION and INTERSECT operations are executed for results of multiple SELECT statements. General format: <pre>select_statement INTERSECT select_statement</pre> select_statement can be any SELECT statement without a FOR UPDATE clause.	-

Parameter	Description	Value Range or Example
EXCEPT clause	The EXCEPT operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one. EXCEPT clause has the following common form: <pre>select_statement EXCEPT [ALL] select_statement</pre> • select_statement can be any SELECT statement without a FOR UPDATE clause. • The result of EXCEPT does not contain any duplicate rows unless the ALL option is specified. To execute ALL, a row that has m duplicates in the left table and n duplicates in the right table will appear $\text{MAX}(m-n, 0)$ times in the result set. • Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. EXCEPT binds at the same level as UNION. • Currently, FOR UPDATE cannot be specified either for an EXCEPT result or for any input of an EXCEPT.	-
MINUS clause	Has the same function and syntax as EXCEPT clause.	-

Parameter	Description	Value Range or Example
ORDER BY clause	Sorts data retrieved by SELECT in descending or ascending order. If the ORDER BY expression contains multiple columns: • If two columns are equal according to the leftmost expression, they are compared according to the next expression and so on. • If they are equal according to all specified expressions, they are returned in an implementation-dependent order. • Columns sorted by ORDER BY must be contained in the result retrieved by SELECT. NOTICE • If ORDER BY is not specified, the query results are returned following the generation sequence in the database system. • You can add the keyword ASC (in ascending order) or DESC (in descending order) next to any expression in the ORDER BY clause. If the keyword is not specified, ASC is used by default. • To sort query results by case-insensitive Chinese pinyin, set the encoding mode to UTF-8 or GBK during database creation.	-
nlsort_expressi on_clause	Sets the sorting mode. The format is as follows: <pre>NLSORT (column_name, ' NLS_SORT = { SCHINESE_PINYIN_M generic_m_ci } ')</pre> NLS_SORT: indicates a field to be ordered in a special mode. Currently, only the Chinese Pinyin order and case insensitive order are supported. Values: • SCHINESE_PINYIN_M: Chinese characters are sorted by pinyin. Currently, only level-1 Chinese characters in the GBK character set can be sorted. To use this sort method, specify GBK as the encoding format when you create the database. If you do not do so, this value is invalid. • generic_m_ci, case-insensitive order.	• Example: Use NLSSORT to Sort Query Results Without Case Sensitivity

Parameter	Description	Value Range or Example
[{ [LIMIT { count ALL }] [OFFSET start [ROW ROWS]] } { LIMIT start, { count ALL } }]	The LIMIT clause consists of two independent LIMIT clauses, an OFFSET clause, and a LIMIT clause with multiple parameters. LIMIT { count ALL } OFFSET start [ROW ROWS] LIMIT start, { count ALL } count in the clauses specifies the maximum number of rows to return, while start specifies the number of rows to skip before starting to return rows. When both are specified, start rows are skipped before starting to count the count rows to be returned. A multi-parameter LIMIT clause cannot be used together with a single-parameter LIMIT or OFFSET clause. Example: LIMIT 10 OFFSET 20; -- Skip the first 20 rows and return the next 10 rows, that is, return rows 21 to 30. LIMIT 5, 10; -- Skip the first five lines and returns the last 10 rows, that is, returns the rows 6 to 16.	-
FETCH { FIRST NEXT } [count] { ROW ROWS } ONLY	The FETCH clause restricts the total number of rows starting from the first row of the return query result.	If count is not specified, the default value 1 is used.

Parameter	Description	Value Range or Example
FOR UPDATE SHARE clause	The FOR UPDATE clause locks the selected rows in a SELECT query to prevent other transactions from modifying or deleting them until the current transaction is completed. This means that any transactions attempting to execute UPDATE, DELETE, or SELECT FOR UPDATE on these rows will be blocked until the current transaction is finished. NOWAIT option: prevents an operation from waiting for other transactions to commit. If the selected row cannot be locked immediately, an error is reported when SELECT FOR UPDATE NOWAIT is executed. • If specified tables are named in FOR UPDATE/SHARE, then only rows coming from those tables are locked; any other tables used in SELECT are simply read as usual. Otherwise, locking all tables in the command. • If FOR UPDATE or FOR SHARE is applied to a view or sub-query, it affects all tables used in the view or sub-query. • Multiple FOR UPDATE and FOR SHARE clauses can be written if it is necessary to specify different locking behaviors for different tables. • If the same table is mentioned (or implicitly affected) by both FOR UPDATE and FOR SHARE clauses, it is processed as FOR UPDATE. Similarly, a table is processed as NOWAIT if that is specified in any of the clauses affecting it. • FOR SHARE or FOR UPDATE: requires a shared lock instead of an exclusive lock for each retrieved row. A share lock blocks other transaction from performing UPDATE, DELETE, or SELECT FOR UPDATE on these rows, but it does not prevent them from performing SELECT FOR SHARE.	-

Parameter	Description	Value Range or Example
	NOTICE - For SQL statements containing FOR UPDATE or FOR SHARE, their execution plans will be pushed down to DNs. If the pushdown fails, an error will be reported. - The query of column storage table does not support FOR UPDATE or FOR SHARE.	

FROM Clause Parameters

FROM from_item clause format:

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]  
[( select ) [ AS ] alias [ ( column_alias [, ...] ) ] ]  
[with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ] ]  
[function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ] ]  
[function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] ) ]  
[from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ] ] }
```

Table 16-2 FROM clause parameters

Parameter	Description	Value Range or Example
table_name	Indicates the name (optionally schema-qualified) of an existing table or view, for example, schema_name.table_name .	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
alias	Gives a temporary alias to a table to facilitate the quotation by other queries. An alias is used for brevity or to eliminate ambiguity for self-joins. When an alias is provided, it completely hides the actual name of the table or function.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
column_alias	Specifies the column alias.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .

Parameter	Description	Value Range or Example
partition_clause	Queries data in the specified partition of a partitioned table. The format is as follows: <pre>PARTITION { (partition_name) FOR (partition_value [, ...]) }</pre> • partition_name: indicates the partition name. • partition_value: indicates the partition key value. If there are many partition keys, use the PARTITION FOR clause to specify the value of the only partition key you want to use. NOTICE Partitions can be specified only for ordinary tables.	Examples: <pre>SELECT * FROM tpcds.reason_p PARTITION (P_05_BEFORE);</pre>
subquery	Performs a subquery in the FROM clause. A temporary table is created to save subquery results.	For example, assign an alias e to a subquery result and use it as a FROM data source. <pre>SELECT d.department_name, e.avg_salary FROM (SELECT department_id, AVG(salary) AS avg_salary FROM employees GROUP BY department_id) AS e JOIN departments d ON e.department_id = d.id;</pre>
with_query_name	WITH clause can also be the source of FROM clause and can be referenced with the name queried by executing WITH.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions. For example, use WITH cte and reference cte in a FROM clause. <pre>WITH cte AS (SELECT id, name FROM users) SELECT * FROM cte AS user_info;</pre>

Parameter	Description	Value Range or Example
function_name	Specifies the function name. Function calls can appear in the FROM clause.	A string, which must comply with the naming convention. For details, see Identifier Naming Conventions .
join_type	Specifies the JOIN type.	Supported JOIN types: • [INNER] JOIN • LEFT [OUTER] JOIN • RIGHT [OUTER] JOIN • FULL [OUTER] JOIN • CROSS JOIN For details, see Table 16-3 .
ON join_condition	A join condition to define which rows have matches in joins. Example: ON left_table.a = right_table.a	-
USING(join_column[, ...])	ON left_table.a = right_table.a AND left_table.b = right_table.b ... abbreviation. Corresponding columns must have the same name.	-
NATURAL	NATURAL is a shorthand for a USING list that mentions all columns in the two tables that have the same names.	-
from item	Specifies the name of the query source object connected.	-

Table 16-3 Five types of join_type

Parameter	Description
[INNER] JOIN	A JOIN clause consists of two FROM items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOIN nests left-to-right. In any case, JOIN binds more tightly than the commas separating FROM items.

Parameter	Description
LEFT [OUTER] JOIN	Returns all rows in the qualified Cartesian product (all combined rows that pass its join condition), and pluses one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting NULL values for the right-hand columns. Note that only the JOIN clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.
RIGHT [OUTER] JOIN	Returns all the joined rows, plus one row for each unmatched right-hand row (extended with NULL on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.
FULL [OUTER] JOIN	Returns all the joined rows, pluses one row for each unmatched left-hand row (extended with NULL on the right), and pluses one row for each unmatched right-hand row (extended with NULL on the left).
CROSS JOIN	CROSS JOIN is equivalent to INNER JOIN ON (TRUE) , which means no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you could not do with plain FROM and WHERE . NOTE - For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL ON, join_condition, or USING (join_column [, ...]). For CROSS JOIN, none of these clauses can appear. CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two items at the top level of FROM.

GROUP BY Parameters

The **group** clause is as follows:

```
(  
| expression  
| ( expression [, ...] )  
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )  
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )  
| GROUPING SETS ( grouping_element [, ...] )
```


Table 16-4 GROUP By parameters

Parameter	Description	Examples
ROLLUP ({ expression (expression [, ...]) } [, ...])	ROLLUP calculates the standard aggregation value specified by an ordered grouping column in GROUP BY, creates a high-level partial sum from right to left, and finally creates a cumulative sum. A group can be regarded as a series of grouping sets. Example: GROUP BY ROLLUP (a,b,c) The preceding condition expression is equivalent to: GROUP BY GROUPING SETS((a,b,c), (a,b), (a), ()) The elements in the ROLLUP clause can be independent fields or expressions, or a list contained in parentheses. If it is a list in parentheses, they must be a whole when the grouping set is generated. Example: GROUP BY ROLLUP ((a,b), (c,d)) The preceding condition expression is equivalent to: GROUPING SETS ((a,b,c,d), (a,b), (c,d), ())	Examples
CUBE ({ expression (expression [, ...]) } [, ...])	A CUBE grouping is an extension to the GROUP BY clause that creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). CUBE can be understood as all dimensions, that is, all combinations of columns in the aggregation key. In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions. For example, given three expressions ($n=3$) in the CUBE clause, the operation results in $2^n = 2^3 = 8$ groupings. Rows grouped on the values of n expressions are called regular rows, and the rest are called superaggregate rows. Example: GROUP BY CUBE (a,b,c) The preceding condition expression is equivalent to: GROUP BY GROUPING SETS((a,b,c), (a,b), (a,c), (b,c), (a), (b), (c), ()) The elements in the CUBE clause can be independent fields or expressions, or a list contained in parentheses. If it is a list in parentheses, they must be a whole when the grouping set is generated. Example: GROUP BY CUBE (a, (b, c), d) The preceding condition expression is equivalent to: GROUP BY GROUPING SETS ((a,b,c,d), (a,b,c), (a), ())	Examples

Parameter	Description	Examples
GROUPING SETS (grouping_element [, ...])	GROUPING SETS is another extension to the GROUP BY clause. It allows users to specify multiple GROUP BY clauses. The option is used to define a grouping set. Each grouping set needs to be included in a separate parenthesis. A blank parenthesis (()) indicates that all data is processed as a group. This improves efficiency by trimming away unnecessary data. You can specify the required data group for query. NOTICE If the SELECT list expression quotes some ungrouped fields and no aggregate function is used, an error is displayed. This is because multiple values may be returned for ungrouped fields.	Examples

Restrictions on the WHERE Clause

- For the **WHERE** clause, if a special character %, _, or \ is queried in the **LIKE** operator, use the backslash (\) for escape.

```
CREATE TABLE tt01 (id int,content varchar(50));

INSERT INTO tt01 values (1,'Jack say "hello"');
INSERT INTO tt01 values (2,'Rose do 50%');
INSERT INTO tt01 values (3,'Lilei say "world"');
INSERT INTO tt01 values (4,'Hanmei do 100%');

SELECT * FROM tt01;
id | content
---+-----
3 | Lilei say 'world'
4 | Hanmei do 100%
1 | Jack say 'hello'
2 | Rose do 50%
(4 rows)

SELECT * FROM tt01 WHERE content like '%"he%';
id | content
---+-----
1 | Jack say 'hello'
(1 row)

SELECT * FROM tt01 WHERE content like '%50\%';
id | content
---+-----
2 | Rose do 50%
(1 row)
```

- In the **WHERE** clause, you can use the operator (+) to convert a table join to an outer join. However, this method is not recommended because it is not the standard SQL syntax and may raise syntax compatibility issues during platform migration. The restrictions on using the plus sign (+) are as follows:
 - It can appear only in the **WHERE** clause.
 - If a table join has been specified in the **FROM** clause, the operator (+) cannot be used in the **WHERE** clause.
 - The operator (+) can work only on columns of tables or views, instead of on expressions.

- d. If table A and table B have multiple join conditions, the operator (+) must be specified in all the conditions. Otherwise, the operator (+) will not take effect, and the table join will be converted into an inner join without any prompt information.
- e. Tables specified in a join condition where the operator (+) works cannot cross queries or subqueries. If tables where the operator (+) works are not in the **FROM** clause of the current query or subquery, an error will be reported. If a peer table for the operator (+) does not exist, no error will be reported and the table join will be converted into an inner join.
- f. Expressions where the operator (+) is used cannot be directly connected through **OR**.
- g. If a column where the operator (+) works is compared with a constant, the expression becomes a part of the join condition.
- h. A table cannot have multiple foreign tables.
- i. The operator (+) can appear only in the following expressions: comparison, **NOT**, **ANY**, **ALL**, **IN**, **NULLIF**, **IS DISTINCT FROM**, and **IS OF** expressions. It is not allowed in other types of expressions. In addition, **AND** and **OR** are not allowed in these expressions.
- j. The operator (+) can be used to convert a table join only to a left or right outer join, instead of a full join. That is, the operator (+) cannot be specified on both tables of an expression.

Example: Use WITH to Implement Subqueries

Obtain the **temp_t** temporary table by a subquery and query all records in this table.

```
WITH temp_t(name,isdba) AS (SELECT username,usesuper FROM pg_user) SELECT * FROM temp_t;
```

Explicitly specify **MATERIALIZED** for the **with_query** named **temp_t**, and then query all data in the **temp_t** table.

```
WITH temp_t(name,isdba) AS MATERIALIZED (SELECT username,usesuper FROM pg_user) SELECT * FROM temp_t;
```

Explicitly specify **NOT MATERIALIZED** for the **with_query** named **temp_t**, and then query all data in the **temp_t** table.

```
WITH temp_t(name,isdba) AS NOT MATERIALIZED (SELECT username,usesuper FROM pg_user)
SELECT * FROM temp_t t1 WHERE name LIKE 'A%'
UNION ALL
SELECT * FROM temp_t t2 WHERE name LIKE 'B%';
```

Example: Other Common SELECT Scenarios

Query all the **r_reason_sk** records in the **tpcds.reason** table and de-duplicate them.

```
SELECT DISTINCT(r_reason_sk) FROM tpcds.reason;
```

Example of a **LIMIT** clause: Obtain a record from the table.

```
SELECT * FROM tpcds.reason LIMIT 1;
```

Example of a **LIMIT** clause: Obtain the third record from the table.

```
SELECT * FROM tpcds.reason LIMIT 1 OFFSET 2;
```

Example of a **LIMIT** clause: Obtain the first two records from a table.

```
SELECT * FROM tpcds.reason LIMIT 2;
```

Query all records and sort them in alphabetic order.

```
SELECT r_reason_desc FROM tpcds.reason ORDER BY r_reason_desc;
```

Use table aliases to obtain data from the **pg_user** and **pg_user_status** tables.

```
SELECT a.username,b.locktime FROM pg_user a,pg_user_status b WHERE a.usesysid=b.roloid;
```

Example of the **FULL JOIN** clause: Join data in the **pg_user** and **pg_user_status** tables.

```
SELECT a.username,b.locktime,a.usesuper FROM pg_user a FULL JOIN pg_user_status b on a.usesysid=b.roloid;
```

Example of the **GROUP BY** clause: Filter data based on query conditions, and group the results.

```
SELECT r_reason_id, AVG(r_reason_sk) FROM tpcds.reason GROUP BY r_reason_id HAVING AVG(r_reason_sk) > 25;
```

Example of the **GROUP BY** clause: Group the results by alias.

```
SELECT r_reason_id AS id FROM tpcds.reason GROUP BY id;
```

Example of the **UNION** clause: Merge the names started with W and N in the **r_reason_desc** column in the **tpcds.reason** table.

```
SELECT r_reason_sk, tpcds.reason.r_reason_desc
FROM tpcds.reason
WHERE tpcds.reason.r_reason_desc LIKE 'W%'
UNION
SELECT r_reason_sk, tpcds.reason.r_reason_desc
FROM tpcds.reason
WHERE tpcds.reason.r_reason_desc LIKE 'N%';
```

Example: Use WITH RECURSIVE to Query the Hierarchical Relationship of a Three-Level Department in a Company

Take the recursion of three levels of departments in a company as an example. In a table that contains level-1, level-2, and level-3 departments, use the keyword **RECURSIVE** to query all level-2 and level-3 departments of a department (for example, the technology department).

Prepare data:

```
DROP TABLE IF EXISTS departments;
CREATE TABLE departments
(id varchar(10),
pid varchar(10),
dept_name varchar(50));
INSERT INTO departments VALUES('0',null,'Head office');
INSERT INTO departments VALUES('1','0','Technology'); -- Level-1 department
INSERT INTO departments VALUES('2','0','Marketing');
INSERT INTO departments VALUES('3','0','HR');
INSERT INTO departments VALUES('4','0','Business');
INSERT INTO departments VALUES('5','0','Maintenance');
INSERT INTO departments VALUES('1-1','1','R&D Center'); -- Level-2 department
INSERT INTO departments VALUES('1-2','1','Pre-Research');
INSERT INTO departments VALUES('1-3','1','Network');
INSERT INTO departments VALUES('1-4','1','Test');
INSERT INTO departments VALUES('1-5','1','Architecture');
```

```
INSERT INTO departments VALUES('1-1-1','1-1','Frontend'); -- Level-3 department
INSERT INTO departments VALUES('1-1-2','1-1','Backend');
INSERT INTO departments VALUES('1-1-3','1-1','Design');
INSERT INTO departments VALUES('2-1','2','Branding');
INSERT INTO departments VALUES('2-2','2','Pricing');
INSERT INTO departments VALUES('2-3','2','Training');
```

Query the original data of the table. In the command output, it is difficult to determine the relationship of the three levels of departments.

```
SELECT * FROM departments;
```

id	pid	dept_name
1	0	Technology
4	0	Business
1-2	1	Pre-Research
1-5	1	Architecture
1-1-3	1-1	Design
2-3	2	Training
0	(Null)	Head office
3	0	HR

In this case, use the keyword **WITH RECURSIVE** to query all level-2 and level-3 departments under the technology department.

```
WITH RECURSIVE t1 AS ( -- Define a common expression named t1.
SELECT id,pid,dept_name FROM departments WHERE id='1' -- Initialize the recursive root node: Query the
record whose ID is 1, that is, the technology department.
UNION ALL --Recursively combine results (duplicate items are retained).
SELECT --Recursive part: query data layer by layer.
t2.id, --Subnode ID, that is, the name of each level-1 department.
t2.pid, --Parent node ID, that is, the head office.
t2.dept_name --Subnode name.
FROM departments t2
INNER JOIN t1 ON t2.pid = t1.id --Connect the recursive result through the parent node ID.
)
SELECT id,dept_name FROM t1 ORDER BY id; --(Final query) All level-2 and level-3 departments are
displayed by ID.
```

id	dept_name
1	Technology
1-1	R&D Center
1-1-1	Frontend
1-1-2	Backend
1-1-3	Design
1-2	Pre-Research
1-3	Network
1-4	Test
1-5	Architecture

View that the preceding query results are not intuitive. To display the department of each level, run the following command:

```
WITH RECURSIVE t1 AS (  
  SELECT --Initialize the root node of recursion: Query the record whose ID is 1, that is, the technology  
    department, and convert data type from record to text.  
    id,  
    pid,  
    dept_name::text -- Explicitly convert the data type to text (prepared for subsequent path concatenation).  
  FROM departments  
  WHERE id='1'  
  
  UNION ALL  
  
  SELECT --Recursive part: concatenate the level path.  
    t2.id,  
    t2.pid,  
    (t1.dept_name||'-'>||t2.dept_name)::text AS dept_name --Concatenate the parent path and the current  
    node name, for example, Technology -> R&D Center.  
  FROM departments t2  
  INNER JOIN t1 ON t2.pid = t1.id )  
  
SELECT id,dept_name FROM t1 ORDER BY id; --(Final query) Display the result with the level path
```

id	dept_name
1	Technology
1-1	Technology->R&D Center
1-1-1	Technology->R&D Center->Frontend
1-1-2	Technology->R&D Center->Backend
1-1-3	Technology->R&D Center->Design
1-2	Technology->Pre-Research
1-3	Technology->Network
1-4	Technology->Test
1-5	Technology->Architecture

Example: Use NLSSORT to Sort Query Results Without Case Sensitivity

Case-insensitive order:

```
CREATE TABLE stu_icode_info (id bigint, name text) DISTRIBUTE BY REPLICATION;  
INSERT INTO stu_icode_info VALUES (1, 'aaaa'), (2, 'AAAA');  
SELECT * FROM stu_icode_info ORDER BY NLSSORT (name, 'NLS_SORT = generic_m_ci');  
id | name  
----+-----  
1 | aaaa  
2 | AAAA  
(2 rows)
```

Example: Query Data in a Partitioned Table

Create the partitioned table **tpcds.reason_p**, insert data, and obtain data from the **P_05_BEFORE** partition of the table.

```
CREATE TABLE tpcds.reason_p  
(  
  r_reason_sk integer,  
  r_reason_id character(16),  
  r_reason_desc character(100)  
)  
PARTITION BY RANGE (r_reason_sk)  
(  
  partition P_05_BEFORE values less than (05),  
  partition P_15 values less than (15),  
  partition P_25 values less than (25),  
  partition P_35 values less than (35),  
  partition P_45_AFTER values less than (MAXVALUE)  
);  
  
INSERT INTO tpcds.reason_p values(3,'AAAAAAAAABAAAAAAAA','reason 1'),  
(10,'AAAAAAAAABAAAAAAAA','reason 2'),(4,'AAAAAAAAABAAAAAAAA','reason 3'),  
(10,'AAAAAAAAABAAAAAAAA','reason 4'),(10,'AAAAAAAAABAAAAAAAA','reason 5'),
```

```
(20,'AAAAAAAACAAAAAAA','reason 6'),(30,'AAAAAAAACAAAAAAA','reason 7');
```

```
SELECT * FROM tpcds.reason_p PARTITION (P_05_BEFORE);  
r_reason_sk | r_reason_id | r_reason_desc
```

```
-----+-----+-----  
4 | AAAAAAABAAAAAAA | reason 3  
3 | AAAAAAABAAAAAAA | reason 1  
(2 rows)
```

—Query the number of rows in partition **P_15**:

```
SELECT count(*) FROM tpcds.reason_p PARTITION (P_15);  
count  
-----  
3  
(1 row)
```

Example of the **GROUP BY** clause: Group records in the **tpcds.reason_p** table by **r_reason_id**, and count the number of records in each group.

```
SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id;  
count | r_reason_id  
-----+-----  
2 | AAAAAAABAAAAAAA  
5 | AAAAAAABAAAAAAA  
(2 rows)
```

Example of the **HAVING** clause: Group records in the **tpcds.reason_p** table by **r_reason_id**, count the number of records in each group, and display only values whose number of **r_reason_id** is greater than 2.

```
SELECT COUNT(*) c,r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING c>2;  
c | r_reason_id  
-----+-----  
5 | AAAAAAABAAAAAAA  
(1 row)
```

Example of the **IN** clause: Group records in the **tpcds.reason_p** table by **r_reason_id**, count the number of records in each group, and display only the numbers of records whose **r_reason_id** is **AAAAAAAABAAAAAAA** or **AAAAAAAADAAAAAAA**.

```
SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING r_reason_id  
IN('AAAAAAAABAAAAAAA','AAAAAAAADAAAAAAA');  
count | r_reason_id  
-----+-----  
5 | AAAAAAABAAAAAAA  
(1 row)
```

Example of the **INTERSECT** clause: Query records whose **r_reason_id** is **AAAAAAAABAAAAAAA** and whose **r_reason_sk** is smaller than 5.

```
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' INTERSECT SELECT * FROM  
tpcds.reason_p WHERE r_reason_sk<5;  
r_reason_sk | r_reason_id | r_reason_desc  
-----+-----+-----  
4 | AAAAAAABAAAAAAA | reason 3  
3 | AAAAAAABAAAAAAA | reason 1  
(2 rows)
```

Example of the **EXCEPT** clause: Query records whose **r_reason_id** is **AAAAAAAABAAAAAAA** and whose **r_reason_sk** is greater than or equal to 4.

```
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' EXCEPT SELECT * FROM  
tpcds.reason_p WHERE r_reason_sk<4;  
r_reason_sk | r_reason_id | r_reason_desc  
-----+-----+-----  
10 | AAAAAAABAAAAAAA | reason 2  
10 | AAAAAAABAAAAAAA | reason 5  
10 | AAAAAAABAAAAAAA | reason 4
```



```
4 | AAAAAAAAAAAAAAAAAA | reason 3
(4 rows)
```

Specify the operator (+) in the **WHERE** clause to indicate a left join.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk =
t2.c_customer_sk(+)
order by 1 desc limit 1;
sr_item_sk | c_customer_id
-----+-----
18000 |
(1 row)
```

Specify the operator (+) in the **WHERE** clause to indicate a right join.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+) =
t2.c_customer_sk
order by 1 desc limit 1;
sr_item_sk | c_customer_id
-----+-----
| AAAAAAAJINGEBAAA
(1 row)
```

Specify the operator (+) in the **WHERE** clause to indicate a left join and add a join condition.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1 order by 1 limit 1;
sr_item_sk | c_customer_id
-----+-----
1 |
(1 row)
```

If the operator (+) is specified in the **WHERE** clause, do not use expressions connected through **AND/OR**.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where not(t1.sr_customer_sk =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1);
ERROR: Operator "+" can not be used in nesting expression.
LINE 1: ...tomer_id from store_returns t1, customer t2 where
not(t1.sr_...
```

If the operator (+) is specified in the **WHERE** clause which does not support expression macros, an error will be reported.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where (t1.sr_customer_sk =
t2.c_customer_sk(+))::bool;
ERROR: Operator "+" can only be used in common expression.
```

If the operator (+) is specified on both sides of an expression in the **WHERE** clause, an error will be reported.

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+) =
t2.c_customer_sk(+);
ERROR: Operator "+" can't be specified on more than one relation in one join condition
HINT: "t1", "t2"...are specified Operator "+" in one condition.
```

Example: Use GROUP BY for a Query (ROLLUP, CUBE, and GROUPING SETS as Part of GROUP BY)

In some report scenarios, data is often grouped using GROUP BY. In the following example, obtain the number of employees in level-2 departments under a level-1 department.

```
DROP TABLE IF EXISTS emp;
CREATE TABLE emp(
```

```
id int primary key, --Employee ID
name text,          --Employee name
dep_1 text,         --Level-1 department
dep_2 text          --Level-2 department
);
```

```
INSERT INTO emp VALUES
(01, 'liwei', 'Product', 'SRE'),
(02, 'liming', 'Product', 'U1'),
(03, 'hanlei', 'Product', 'U1'),
(04, 'hanmeimei', 'Product', 'SRE'),
(05, 'dingjun', 'Product', 'SRE'),
(06, 'huxue', 'Product', 'E1'),
(07, 'liuyi', 'Product', 'E1'),
(08, 'luhua', 'Product', 'E1'),
(09, 'yangjing', 'Product', 'U1'),
(10, 'binbin', 'Product', 'U1'),
(11, 'jim', 'Product', 'SRE'),
(12, 'leo', 'Product', 'SER');
```

```
SELECT COUNT(*), dep_2 FROM emp GROUP BY dep_2; --Count the number of employees in level-2
departments.
```

count	dep_2
1	SER
3	E1
4	SRE
4	U1

In common statistical report services, the total number of employees in a level-1 department (that is, the sum of employees in level-2 departments) needs to be calculated. In this case, **ROLLUP** can be used to ensure that there is an additional row of total data in the result, as shown in the following command output. In the following figure, **count** is **12** and **dep_2** is **Null**, which indicate that the total number of employees in the level-1 department is 12.

```
SELECT COUNT(*), dep_2 FROM emp GROUP BY ROLLUP(dep_2) ORDER BY 1;
```

count	dep_2
1	SER
3	E1
4	U1
4	SRE
12	(Null)

The following describes how to use ROLLUP, CUBE, and GROUPING SETS.

- **ROLLUP:**

Create **customer** table.

```
DROP TABLE IF EXISTS customer;
CREATE TABLE customer
(
  c_id char(16) not null,      --Customer ID
  c_name char(20),            --Customer name
  c_age integer,              --Age
  c_country varchar(20),      --Country
  c_class char(10),           --Customer level
  c_gender text,              --Gender
  c_balance numeric );        --Balance

INSERT INTO customer VALUES(1, 'tom', '20', 'Region A', '1', 'male', 300);
INSERT INTO customer VALUES(2, 'jack', '30', 'Region B', '1', 'male', 100);
INSERT INTO customer VALUES(3, 'rose', '40', 'Region C', '1', 'female', 200);
INSERT INTO customer VALUES(4, 'Frank', '60', 'Region D', '1', 'male', 100);
INSERT INTO customer VALUES(5, 'Leon', '20', 'Region A', '2', 'male', 200);
INSERT INTO customer VALUES(6, 'Lucy', '20', 'Region A', '1', 'female', 500);
```

ROLLUP generates subtotals and grand totals across multiple levels of aggregation within a single query. It starts from the most detailed level of aggregation and progressively rolls up to less detailed levels, ultimately producing a grand total.

```
-- Group by c_country and c_class, then group by c_country, and finally group by the entire table.
SELECT c_country, c_class, sum(c_balance) FROM customer GROUP BY ROLLUP(c_country, c_class)
ORDER BY 1,2,3;
```

c_country	c_class	sum
Region A	1	800
Region A	2	200
Region A	(Null)	1000
Region B	1	100
Region B	(Null)	100
Region C	1	200
Region C	(Null)	200
Region D	1	100
Region D	(Null)	100
(Null)	(Null)	1400

Total Records: 10

The preceding statement is equivalent to:

```
SELECT c_country, c_class, SUM(c_balance) FROM customer GROUP BY c_country, c_class UNION ALL
SELECT c_country, null, SUM(c_balance) FROM customer GROUP BY c_country
UNION ALL
SELECT null, null, SUM(c_balance) FROM customer ORDER BY 1,2,3;
```

- **CUBE:**

CUBE can be understood as all dimensions, that is, all combinations of columns in the aggregation key.

```
-- Perform GROUP BY on (c_country,c_class), (c_country), and (c_class) in sequence, and on the entire table.  
SELECT c_country, c_class, SUM(c_balance) FROM customer GROUP BY CUBE(c_country, c_class)  
ORDER BY 1,2,3;
```

c_country	c_class	sum
Region A	1	800
Region A	2	200
Region A	(Null)	1000
Region B	1	100
Region B	(Null)	100
Region C	1	200
Region C	(Null)	200
Region D	1	100
Region D	(Null)	100
(Null)	1	1200
(Null)	2	200
(Null)	(Null)	1400

Total Records: 12

The preceding statement is equivalent to:

```
SELECT c_country, c_class, SUM(c_balance) FROM customer GROUP BY c_country, c_class  
union all  
SELECT c_country, null, SUM(c_balance) FROM customer GROUP BY c_country  
union all  
SELECT null, null, SUM(c_balance) FROM customer  
union all  
SELECT NULL, c_class, SUM(c_balance) FROM customer GROUP BY c_class order by 1,2,3;
```

- **GROUPING SETS:**

GROUPING SETS is different from **ROLLUP** and **CUBE**. It extracts some records from the results of **ROLLUP** and **CUBE**.

```
-- Perform GROUP BY on (c_class) and (c_country) separately.  
SELECT c_country, c_class, SUM(c_balance) FROM customer GROUP BY GROUPING SETS(c_country,  
c_class) ORDER BY 1,2,3;
```

c_country	c_class	sum
Region A	(Null)	1000
Region B	(Null)	100
Region C	(Null)	200
Region D	(Null)	100
(Null)	1	1200
(Null)	2	200

Total Records: 6

The preceding statement is equivalent to:

```
SELECT c_country, null, SUM(c_balance) FROM customer GROUP BY c_country
UNION ALL
SELECT null, c_class, sum(c_balance) FROM customer GROUP BY c_class ORDER BY 1,2,3;
```

16.3 SELECT INTO

Function

SELECT INTO defines a new table based on a query result and insert data obtained by query to the new table.

Different from **SELECT**, data found by **SELECT INTO** is not returned to the client. The table columns have the same names and data types as the output columns of the **SELECT**.

Precautions

CREATE TABLE AS provides functions similar to **SELECT INTO** in functions and provides a superset of functions provided by **SELECT INTO**. You are advised to replace **SELECT INTO** with **CREATE TABLE AS** because **SELECT INTO** cannot be used in stored procedures and **SELECT INTO (column)** cannot receive blank rows.

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    { * | {expression [ [ AS ] output_name ] [, ...] }
    INTO [ [ GLOBAL | LOCAL | VOLATILE ] { TEMPORARY | TEMP } | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW {window_name AS ( window_definition )} [, ...] ]
    [ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
    [ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause } [ NULLS { FIRST |
LAST } ]} [, ...] ]
    [ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] | { LIMIT start, { count | ALL } } ]
```

```
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]  
[ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

Parameter Description

**INTO [[GLOBAL | LOCAL | VOLATILE] { TEMPORARY | TEMP } | UNLOGGED]
[TABLE] new_table**

[GLOBAL | LOCAL | VOLATILE] { TEMPORARY | TEMP } specifies the type of a temporary table. For details, see the description of GLOBAL | LOCAL | VOLATILE in [Parameter Description](#).

UNLOGGED indicates that the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well.

new_table specifies the name of the new table.

NOTE

For details about other **SELECT INTO** parameters, see [SELECT Main Clause Parameters](#) in **SELECT**.

Examples

Add values whose **TABLE_SK** is less than 3 in the **reason_t** table to the new table.

```
CREATE TABLE reason_t  
(  
    TABLE_SK    INTEGER  
    TABLE_ID    VARCHAR(20)  
    TABLE_NA    VARCHAR(20)  
);  
INSERT INTO reason_t VALUES (1, 'S01', 'StudentA'), (2, 'T01', 'TeacherA'), (3, 'T02', 'TeacherB'), (3, 'S02', 'StudentB');  
  
SELECT * INTO reason_t_bck FROM reason_t WHERE TABLE_SK < 3;  
INSERT 0 2
```

Helpful Links

[SELECT](#)

17 TCL Syntax

17.1 TCL Syntax Overview

Transaction Control Language (TCL) controls the time and effect of database transactions and monitors the database.

Commit

DWS uses the COMMIT or END statement to commit transactions. For details, see [COMMIT | END](#).

Setting a Savepoint

DWS creates a new savepoint in the current transaction. For details, see [SAVEPOINT](#).

Rollback

DWS rolls back the current transaction to the last committed state. For details, see [ROLLBACK](#).

17.2 ABORT

Function

ABORT rolls back the current transaction and cancels the changes in the transaction.

This command is equivalent to [ROLLBACK](#), and is present only for historical reasons. Now **ROLLBACK** is recommended.

Precautions

ABORT has no impact outside a transaction, but will provoke a warning.

Syntax

```
ABORT [ WORK | TRANSACTION ] ;
```

Parameter Description

WORK | TRANSACTION

This keyword is optional and does not affect the **ABORT** operation.

Examples

Abort a transaction. Performed update operations will be undone.

```
ABORT;
```

Helpful Links

[SET TRANSACTION](#), [COMMIT | END](#), [ROLLBACK](#)

17.3 BEGIN

Function

BEGIN may be used to initiate an anonymous block or a single transaction. This section describes the syntax of **BEGIN** used to initiate an anonymous block. For details about the **BEGIN** syntax that initiates transactions, see [START TRANSACTION](#).

An anonymous block is a structure that can dynamically create and execute stored procedure code instead of permanently storing code as a database object in the database.

Precautions

None

Syntax

- Enable an anonymous block:

```
[DECLARE [declare_statements]]  
BEGIN  
execution_statements  
END;  
/
```
- -- Start a transaction:

```
BEGIN [ WORK | TRANSACTION ]  
[  
  {  
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE  
    READ }  
    | { READ WRITE | READ ONLY }  
  } [ ...]  
];
```


Parameter Description

Table 17-1 BEGIN parameters

Parameter	Description	Value Range
declare_statements	Declares a variable, including its name and type, for example, sales_cnt int .	-
execution_statements	Specifies the statement to be executed in an anonymous block.	Use an existing function name.

Examples

- Start a transaction block.
`BEGIN;`
- Start a transaction block at the REPEATABLE READ isolation level.
`BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
- Generate a string using an anonymous block.
`BEGIN
dbms_output.put_line('Hello');
END;
/`

Helpful Links

[START TRANSACTION](#)

17.4 CHECKPOINT

Function

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to a disk.

CHECKPOINT forces a transaction log checkpoint. By default, WALs periodically specify checkpoints in a transaction log. You may use **gs_guc** to specify run-time parameters **checkpoint_segments** and **checkpoint_timeout** to adjust the atomized checkpoint intervals.

Precautions

- Only a system administrator has the permission to call **CHECKPOINT**.
- CHECKPOINT** forces an immediate checkpoint when the related command is issued, without waiting for a regular checkpoint scheduled by the system.

Syntax

```
CHECKPOINT;
```

Parameter Description

None

Examples

Set a checkpoint:

```
CHECKPOINT;
```

17.5 COMMIT | END

Function

COMMIT or **END** commits all operations of a transaction.

Precautions

Only the transaction creators or system administrators can run the **COMMIT** command. The creation and commit operations must be in different sessions.

Syntax

```
{ COMMIT | END } [ WORK | TRANSACTION ] ;
```

Parameter Description

Table 17-2 COMMIT | END parameters

Parameter	Description
COMMIT END	Commits the current transaction and make all changes made by the transaction become visible to others.
WORK TRANSACTION	The keyword is optional and only improves readability, with no effect on transaction commitment.

Examples

Commit the transaction to make all changes permanent:

```
COMMIT;
```

Helpful Links

[ROLLBACK](#)

17.6 COMMIT PREPARED

Function

COMMIT PREPARED commits a prepared two-phase transaction.

Precautions

- The function is only available in maintenance mode (when GUC parameter **xc_maintenance_mode** is **on**). Exercise caution when enabling the mode. It is used by maintenance engineers for troubleshooting. Common users should not use the mode.
- Only the transaction creators or system administrators can run the **COMMIT** command. The creation and commit operations must be in different sessions.
- The transaction function is maintained automatically by the database, and should be not visible to users.

Syntax

```
COMMIT PREPARED transaction_id ;  
COMMIT PREPARED transaction_id WITH CSN;
```

Parameter Description

Table 17-3 COMMIT PREPARED parameters

Parameter	Description
transaction_id	Identifier of the transaction to be committed. The identifier cannot be the same as any identifier used by the current prepared transaction.
CSN(commit sequence number)	Sequence number of the transaction to be committed. The value is a 64-bit ascending unsigned number.

Helpful Links

[PREPARE TRANSACTION](#), [ROLLBACK PREPARED](#)

17.7 PREPARE TRANSACTION

Function

PREPARE TRANSACTION prepares the current transaction for two-phase commit.

After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a high probability that

it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with **COMMIT PREPARED** or **ROLLBACK PREPARED**, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, **PREPARE TRANSACTION** is not unlike a **ROLLBACK** command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the **PREPARE TRANSACTION** command fails for any reason, it becomes a **ROLLBACK** and the current transaction is canceled.

Precautions

- The transaction function is maintained automatically by the database, and should be not visible to users.
- When running the **PREPARE TRANSACTION** command, increasing the value of **max_prepared_transactions** in configuration file **postgresql.conf**. You are advised to set **max_prepared_transactions** to a value not less than that of **max_connections** so that one pending prepared transaction is available for each session.

Syntax

```
PREPARE TRANSACTION transaction_id;
```

Parameter Description

transaction_id

An arbitrary identifier that later identifies this transaction for **COMMIT PREPARED** or **ROLLBACK PREPARED**. The identifier must be different from those for current prepared transactions.

Value range: The identifier must be written as a string literal, and must be less than 200 bytes long.

Helpful Links

[COMMIT PREPARED](#), [ROLLBACK PREPARED](#)

17.8 SAVEPOINT

Function

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that rolls back all commands that are executed after the savepoint was established, restoring the transaction state to what it was at the time of the savepoint.

Precautions

- Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint. Use **RELEASE SAVEPOINT** to destroy a savepoint but keep the effects of the commands executed after the savepoint was established.
- Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.
- **SAVEPOINT** cannot be used for functions, anonymous blocks, or stored procedures.
- In the case of an unexpected termination of a distributed thread or process caused by a node or connection failure, or of an error caused by the inconsistency between source and destination table structures in a COPY FROM operation, the transaction cannot be rolled back to the established savepoint. Instead, the entire transaction will be rolled back.
- According to the SQL standard, a savepoint is destroyed automatically when another savepoint with the same name is established. In DWS, old savepoints are kept, though only the most recent one will be used for rollback or release. Releasing the newer savepoint with **RELEASE SAVEPOINT** will cause the older one to again become accessible to **ROLLBACK TO SAVEPOINT** and **RELEASE SAVEPOINT**. Except for this, **SAVEPOINT** is fully SQL conforming.

Syntax

```
SAVEPOINT savepoint_name;
```

Parameter Description

savepoint_name

Specifies the name of a new savepoint.

Examples

- Create a savepoint and undo all commands executed after the savepoint is created:

```
START TRANSACTION;
INSERT INTO table1 VALUES (1);
SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (3);
COMMIT;
```

Query the table content, which should contain 1 and 3 but not 2, because 2 has been rolled back.
- Create and then destroy a savepoint.

```
START TRANSACTION;
INSERT INTO table1 VALUES (3);
SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (4);
RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Query the table content, which should contain both 3 and 4.

Helpful Links

[RELEASE SAVEPOINT, ROLLBACK TO SAVEPOINT](#)

17.9 SET TRANSACTION

Function

SET TRANSACTION sets the characteristics of the current transaction. It has no effect on any subsequent transactions. Available transaction characteristics include the transaction separation level and transaction access mode (read/write or read only).

Precautions

None

Syntax

Set the isolation level and access mode of the transaction.

```
{ SET [ LOCAL ] TRANSACTION|SET SESSION CHARACTERISTICS AS TRANSACTION }  
  { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }  
    | { READ WRITE | READ ONLY } } [, ...]
```

Parameter Description

Table 17-4 SET TRANSACTION parameters

Parameter	Description	Value Range
LOCAL	Indicates that the specified command takes effect only for the current transaction.	-
SESSION	Indicates that the specified command takes effect only for the current session.	A string, which must comply with the naming convention.

Parameter	Description	Value Range
ISOLATION LEVEL	Specifies the transaction isolation level that determines the data that a transaction can view if other concurrent transactions exist. NOTE The isolation level of a transaction cannot be reset after the first clause (INSERT, DELETE, UPDATE, FETCH, COPY) for modifying data is executed in the transaction.	• READ COMMITTED: Only committed data is read. The default value is READ COMMITTED. • READ UNCOMMITTED: DWS does not support READ UNCOMMITTED. If READ UNCOMMITTED is set, READ COMMITTED is executed instead. • REPEATABLE READ: Only the data committed before transaction start is read. Uncommitted data or data committed in other concurrent transactions cannot be read. • SERIALIZABLE: DWS does not support SERIALIZABLE. If SERIALIZABLE is set, REPEATABLE READ is executed instead.
READ WRITE READ ONLY	Specifies the transaction access mode.	Read-write mode or read-only mode.

Examples

Set the isolation level of the current transaction to **READ COMMITTED** and the access mode to **READ ONLY**:

```
START TRANSACTION;  
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;  
COMMIT;
```

17.10 START TRANSACTION

Function

START TRANSACTION starts a transaction. If the isolation level, read/write mode, or deferrable mode is specified, a new transaction will have those characteristics. You can also specify them using [SET TRANSACTION](#).

Precautions

None

Syntax

Format 1: START TRANSACTION

```
START TRANSACTION  
[
```

```
{  
  ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }  
  | { READ WRITE | READ ONLY }  
} [, ...]  
};
```

Format 2: BEGIN

```
BEGIN [ WORK | TRANSACTION ]  
[  
  {  
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }  
    | { READ WRITE | READ ONLY }  
  } [, ...]  
];
```

Parameter Description

Table 17-5 START TRANSACTION parameters

Parameter	Description	Value Range
WORK TRANSACTION	Optional keyword in the BEGIN format, which does not affect the operation.	-
ISOLATION LEVEL	Transaction isolation level. This parameter determines the data that a transaction can view if other concurrent transactions exist. NOTE The isolation level of a transaction cannot be reset after the first clause (INSERT , DELETE , UPDATE , FETCH , COPY) for modifying data is executed in the transaction.	• READ COMMITTED: Only committed data is read. The default value is READ COMMITTED. • READ UNCOMMITTED: DWS does not support READ UNCOMMITTED. If READ UNCOMMITTED is set, READ COMMITTED is executed instead. • REPEATABLE READ: Only the data committed before transaction start is read. Uncommitted data or data committed in other concurrent transactions cannot be read. • SERIALIZABLE: DWS does not support SERIALIZABLE. If SERIALIZABLE is set, REPEATABLE READ is executed instead.
READ WRITE READ ONLY	Transaction access mode.	Read-write mode or read-only mode.

Examples

- Start a transaction in default mode.

```
START TRANSACTION;  
SELECT * FROM tpceds.reason;  
END;
```
- Start a transaction with the isolation level being **READ COMMITTED** and the access mode being **READ WRITE**:


```
START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;  
SELECT * FROM tpeds.reason;  
COMMIT;
```

Helpful Links

[COMMIT](#) | [END](#), [ROLLBACK](#), [SET TRANSACTION](#)

17.11 ROLLBACK

Function

Rolls back the current transaction and backs out all updates in the transaction.

ROLLBACK backs out of all changes that a transaction makes to a database if the transaction fails to be executed due to a fault.

Precautions

If a **ROLLBACK** statement is executed out of a transaction, no error occurs, but a warning information is displayed.

Syntax

```
ROLLBACK [ WORK | TRANSACTION ];
```

Parameter Description

WORK | TRANSACTION

This keyword is optional and only improves readability, with no effect on the **ROLLBACK** operation.

Examples

Undo all changes in the current transaction:

```
ROLLBACK;
```

Helpful Links

[COMMIT](#) | [END](#)

17.12 RELEASE SAVEPOINT

Function

RELEASE SAVEPOINT destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. To do that, use **ROLLBACK TO SAVEPOINT**.

Destroying a savepoint when it is no longer needed allows the system to reclaim some resources earlier than transaction end.

RELEASE SAVEPOINT also destroys all savepoints that were established after the named savepoint was established.

Precautions

- Specifying a savepoint name that was not previously defined causes an error.
- It is not possible to release a savepoint when the transaction is in an aborted state.
- If multiple savepoints have the same name, only the one that was most recently defined is released.

Syntax

```
RELEASE [ SAVEPOINT ] savepoint_name;
```

Parameter Description

savepoint_name

Specifies the name of the savepoint you want to destroy.

Examples

Create and then destroy a savepoint:

```
BEGIN;  
  INSERT INTO tpods.table1 VALUES (3);  
  SAVEPOINT my_savepoint;  
  INSERT INTO tpods.table1 VALUES (4);  
  RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

Helpful Links

[SAVEPOINT](#), [ROLLBACK TO SAVEPOINT](#)

17.13 ROLLBACK PREPARED

Function

ROLLBACK PREPARED cancels a transaction ready for two-phase committing.

Precautions

- The function is only available in maintenance mode (when GUC parameter **xc_maintenance_mode** is **on**). Exercise caution when enabling the mode. It is used by maintenance engineers for troubleshooting. Common users should not use the mode.
- Only the user that initiates a transaction or the system administrator can roll back the transaction.
- The transaction function is maintained automatically by the database, and should be not visible to users.

Syntax

```
ROLLBACK PREPARED transaction_id ;
```

Parameter Description

transaction_id

Specifies the identifier of the transaction to be submitted. The identifier must be different from those for current prepared transactions.

Helpful Links

[COMMIT PREPARED](#), [PREPARE TRANSACTION](#)

17.14 ROLLBACK TO SAVEPOINT

Function

ROLLBACK TO SAVEPOINT rolls back to a savepoint. It implicitly destroys all savepoints that were established after the named savepoint.

Rolls back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

Precautions

- Specifying a savepoint name that has not been established is an error.
- Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a **FETCH** or **MOVE** command inside a savepoint that is later rolled back, the cursor remains at the position that **FETCH** left it pointing to (that is, the cursor motion caused by **FETCH** is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a cannot-execute state, so while the transaction can be restored using **ROLLBACK TO SAVEPOINT**, the cursor can no longer be used.
- Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint. Use **RELEASE SAVEPOINT** to destroy a savepoint but keep the effects of the commands executed after the savepoint was established.

Syntax

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name;
```

Parameter Description

savepoint_name

Rolls back the name of the specified savepoint.

Examples

Undo the effects of the commands executed after my_savepoint was established.

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by savepoint rollback:

```
BEGIN;  
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;  
SAVEPOINT foo;  
FETCH 1 FROM foo;  
?column?  
-----  
1  
ROLLBACK TO SAVEPOINT foo;  
FETCH 1 FROM foo;  
?column?  
-----  
2  
COMMIT;
```

Helpful Links

[SAVEPOINT](#), [RELEASE SAVEPOINT](#)