AI DataLake

Aura Syntax Reference

Issue 01

Date 2026-11-09



HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2026. 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 Technologies Co., Ltd.

Trademarks and Permissions


and other Huawei trademarks are trademarks 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 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.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Before You Start..... 1

2 Development Process Using Aura SQL..... 3

3 DDL Syntax..... 7

3.1 Designing Database Objects..... 7
3.1.1 Creating and Managing Schemas..... 7
3.1.2 Creating and Managing Tables..... 9
3.1.3 Defining Table Partitions..... 11
3.2 SHOW..... 12
3.2.1 SHOW Conf..... 12
3.2.2 SHOW Schemas/Tables/Views/Partitions/Functions/Models..... 13
3.3 CREATE SCHEMA..... 14
3.4 DROP SCHEMA..... 15
3.5 CREATE TABLE..... 16
3.6 CREATE VIEW..... 26
3.7 CREATE FUNCTION..... 27
3.8 CREATE MODEL..... 30
3.9 DROP TABLE..... 31
3.10 DROP VIEW..... 31
3.11 DROP FUNCTION..... 32
3.12 DROP MODEL..... 32
3.13 ALTER TABLE..... 33
3.14 DESCRIBE..... 34
3.15 MSCK REPAIR TABLE..... 35
3.16 TRUNCATE..... 35
3.17 CREATE INDEX..... 36
3.18 DROP INDEX..... 42
3.19 SHOW INDEX..... 43
3.20 ALTER INDEX..... 43
3.21 CREATE ROLE..... 44
3.22 DROP ROLE..... 44
3.23 ALTER ROLE..... 44
3.24 ALTER USER..... 45

 Copyright © Huawei Technologies Co., Ltd.

3.25 ALTER USER GROUP 45
4 DML Syntax 46
4.1 Overview 46
4.2 EXPLAIN 46
4.3 INSERT 50
4.4 VALUES 54
4.5 DELETE 56
4.6 MERGE INTO 57
4.7 OPTIMIZE 60
4.8 UPDATE 62
5 DQL Syntax 65
5.1 SELECT 65
5.1.1 Basic Syntax 65
5.1.2 Table Query Examples 77
5.1.2.1 Single Table Query 77
5.1.2.2 Multi-Table Join Queries 79
5.2 WITH Expression 86
6 DCL Syntax 88
6.1 ANALYZE | ANALYSE 88
6.2 GRANT 89
6.3 REVOKE 90
7 Keywords 91
8 Data Type 121
8.1 Mapping Between Aura and LakeFormation Data Types 121
8.2 Numeric Types 124
8.3 Boolean Type 128
8.4 Character Types 128
8.5 Binary Type 131
8.6 Date/Time Types 132
8.7 Object Identifier Type 136
8.8 Implicit Conversion Support Range 138
8.9 Complex Types 138
9 Type Conversion 141
9.1 Overview 141
9.2 Operators 142
9.3 Functions 144
9.4 Value Storage 146
9.5 UNION, CASE, and Related Constructs 147
10 Expression 149
10.1 Simple Expressions 149

 Copyright © Huawei Technologies Co., Ltd.

10.2 Conditional Expressions 150
10.3 Subquery Expressions 155
10.4 Array Expressions 157
10.5 Row Expressions 159
11 Functions and Operators 161
11.1 Character Processing Functions and Operators 161
11.2 Binary String Functions and Operators 185
11.3 Bit String Functions and Operators 187
11.4 Numeric Operation Functions and Operators 188
11.4.1 Numeric Operators 189
11.4.2 Numeric Operation Functions 192
11.5 Time and Date Processing Functions and Operators 202
11.5.1 Time and Date Operators 202
11.5.2 Time and Date Functions 205
11.5.3 EXTRACT 222
11.5.4 date_part 226
11.5.5 date_format 227
11.5.6 time_format 229
11.6 Array Functions and Operators 232
11.6.1 Array Operators 232
11.6.2 Array Functions 235
11.7 Logical Operators 239
11.8 Comparison Operators 239
11.9 Pattern Matching Operators 240
11.10 Aggregate Functions 243
11.11 Window Functions 253
11.12 Type Conversion Functions 258
11.13 JSON/JSONB Functions and Operators 267
11.13.1 JSON/JSONB Operators 267
11.13.2 JSON/JSONB Functions 271
11.14 Conditional Expression Functions 289
11.15 Set Returning Functions 292
11.15.1 Series Generating Functions 292
11.16 System Information Functions 293
11.16.1 Session Information Functions 293
11.16.2 System Catalog Information Functions 294
11.16.3 System Function Information Functions 297
11.16.4 Status Information Functions 298
11.17 System Management Functions 300
11.17.1 Memory Management Functions 300
11.18 Database Object Functions 305
11.18.1 Database Object Size Functions 306

 Copyright © Huawei Technologies Co., Ltd.

11.19 Statistics Functions....306
11.20 Iceberg Table Service Functions....319
11.21 Large Model Inference Functions....335
12 User and Permissions Management....336
12.1 Aura Service Permissions....336
12.2 User Management....337
12.3 Permissions Management....337
12.3.1 Permission Introduction....337
12.3.2 Role Management....338
12.3.3 Object Authorization/Deauthorization Management....339
13 Performance Tuning....349
13.1 Overview....349
13.2 System Optimization....351
13.2.1 Tuning Database Parameters....351
13.2.2 SMP Parallel Execution....352
13.3 SQL Tuning....355
13.3.1 SQL Query Execution Process....355
13.3.2 SQL Execution Plan....357
13.3.3 Execution Plan Operator....367
13.3.4 SQL Execution Monitoring....371
13.3.5 SQL Tuning Process....382
13.3.6 Updating Statistics....383
13.3.7 Advanced SQL Tuning....384
13.3.7.1 Optimizing Subqueries....385
13.3.7.2 Optimizing Operators....392
13.3.7.3 SQL Statement Rewriting Rules....393
13.3.8 Configuring Optimizer Parameters....394
13.3.9 Hint-based Tuning....394
13.3.9.1 Plan Hint Optimization....394
13.3.9.2 Join Order Hints....396
13.3.9.3 Join Operation Hints....400
13.3.9.4 Rows Hints....400
13.3.9.5 Stream Operation Hints....401
13.3.9.6 Hints for Sublink Block Names....404
13.3.9.7 Hint That Disables Subquery Pull-up....405
13.3.9.8 Drive Hints....406
13.3.9.9 Hints for Configuration Parameters....407
13.3.9.10 Hints for UDF Fusion....409
13.3.9.11 Hint Errors, Conflicts, and Other Warnings....409
13.3.10 Execution Plan Operator Monitoring....412
13.4 SQL Tuning Examples....417
13.4.1 Case: Adjusting the GUC Parameter best_agg_plan....417

 Copyright © Huawei Technologies Co., Ltd.

14 GUC Parameters 420
14.1 Viewing and Setting GUC Parameters 420
14.2 Connection and Authentication 421
14.2.1 Security and Authentication 421
14.3 Query Planning 421
14.3.1 Optimizer Method Configuration 421
14.3.2 Other Optimizer Options 428
14.4 Default Settings of Client Connection 436
14.4.1 Statement Behavior 436
14.4.2 Locale and Formatting 444
14.5 Lock Management 447
14.6 Python UDF 449
14.7 Vector Search 450
14.8 Checkpoint-based Fault Recovery 453

 Copyright © Huawei Technologies Co., Ltd.

1 Before You Start

Before using Aura for data development, you should understand its core concepts and the two interaction modes it provides.

NOTE

This document applies only to AuraJobV2 and does not apply to AuraJob.

What Is Aura?

Aura is a high-performance, distributed computing engine designed for multimodal data processing. In the AI era, data is no longer limited to traditional structured tables. Unstructured data such as images, audio, video, and text plays an increasingly important role. The core design goal of Aura is to simplify multimodal data processing, enabling developers to use familiar Structured Query Language (SQL) or Python DataFrame APIs to perform complex data analysis tasks. In Aura, both SQL and DataFrame queries are compiled into optimized execution plans by the underlying runtime engine.

SQL Development and DataFrame Development

- SQL development: Standard and efficient

  SQL is a widely used language in data processing systems. In Aura, SQL is primarily intended for users familiar with data warehouse workloads.

  - **Key characteristics**: Declarative programming model, that is, you specify what to compute, not how to compute it.
  - **Typical use cases**: Fast data extraction, complex multi-table joins and aggregations, and integration with existing BI tools and systems.
  - **Aura advantages**: Aura supports most of the SQL:2016 standard out of the box, making it easy to learn and naturally suited for report development.

- DataFrame development: Flexible and programmatic

  DataFrame is a programming abstraction that represents data as a labeled two-dimensional structure, while enabling richer programmatic control over data processing logic. Aura's DataFrame APIs support both general structured data processing (similar to Ibis) and multimodal data processing. This allows developers to directly operate on image, audio, and text columns—for

 Copyright © Huawei Technologies Co., Ltd.

example, resizing images, extracting audio features, or computing text embeddings.

- **Key characteristics**: Supports both imperative and functional programming styles; supports method chaining; and allows seamless integration with Python's native control flow (loops, conditionals) and rich ecosystem (e.g., machine learning and visualization libraries).
- **Typical use cases**: Feature engineering, automated pipelines, and complex data transformation logic.

Choosing Between SQL and DataFrame for Development

In Aura, SQL and DataFrame are complementary rather than mutually exclusive.

- **When to use SQL:**
  - You are working with complex queries involving joins or nested subqueries.
  - You need readable and shareable analytical logic.
  - Your team is SQL-oriented.
  - You need integration with BI tools or downstream applications.
- **When to use DataFrame:**
  - You are performing exploratory data analysis and need to check intermediate results frequently.
  - You prefer Python programming with flexible control flow.
  - You require custom functions or complex transformation logic.
  - You plan to integrate with machine learning frameworks such as PyTorch or TensorFlow.

Aura supports seamless interoperability between SQL and DataFrame. For example, you can use DataFrame APIs to load and preprocess data, then run SQL for final aggregation and analysis. You can also call custom Python functions within SQL workflows.

 Copyright © Huawei Technologies Co., Ltd.

2 Development Process Using Aura SQL

After selecting the SQL development mode, you can use Aura to develop jobs according to the following process.

Step 1: Select a Table Storage Format

Before creating a table, you need to determine its storage format. Aura supports storage formats such as ORC, PARQUET, ICEBERG, and LANCE. Among these, ICEBERG and LANCE are the primary capabilities for data lake storage integration.

Table Type	Positioning	Key Capability
ORC	Default general format	Columnar storage and partitioning.
Parquet	General columnar format	Columnar storage and partitioning.
Iceberg	Open table format (lakehouse)	Massive data (PB-level), ACID transactions, snapshots, time travel, expression partitioning, and table-level ALTER operations.
Lance	AI-oriented columnar format	High-performance columnar storage, vector indexing, online query, and AI/embedding scenarios.
Text/CSV	Data import and export	Only basic import and export are supported. Advanced functions such as partitioning and ANALYZE are not supported. Therefore, this format is not recommended as a storage medium.

Apache Iceberg is an open table format designed for massive analytic datasets. It adds table functionality to compute engines through a high-performance table format and is used in exactly the same way as SQL tables.

Iceberg is designed for ultra-large tables and is widely deployed in production environments. A single table can contain tens of petabytes of data, and even at this scale, data can be read without a distributed SQL engine.

 Copyright © Huawei Technologies Co., Ltd.

For more information about SQL on Iceberg, see **Iceberg Functions**.

Lance is a high-performance columnar data storage format built on the Apache Arrow ecosystem. Developed primarily by the LanceDB team, it aims to achieve data access performance close to that of the Arrow in-memory format, while also supporting transaction management, indexing, and incremental updates.

Lance is suitable not only for offline batch processing and analysis, but also for online query, vector search, and AI applications. It is a new storage format designed for AI and data lake convergence scenarios.

For more information about SQL on Lance, see **Lance Functions**.

Step 2: Design Database Objects

Before creating objects, you are advised to design and plan database objects as follows:

- **Understand Aura database objects**: Be familiar with the database object types supported by Aura and their relationships.
- **Plan schemas**: Schemas are used to logically organize objects and data in a database, allowing multiple users to use the same database without interfering with each other.
- **Design table structures**: Determine the columns, types, and constraints of tables.
- **Design partitioning policies**: Plan partitioning modes for large tables to improve query performance.

For details, see **3.1 Designing Database Objects**.

Step 3: Configure Permissions

Before creating and accessing objects, you need to configure users, roles, and object permissions to ensure that users have the corresponding operation permissions:

- Understand the Aura permission model.
- Create and manage users.
- Authorize objects, cancel authorization, and manage roles.

Permission management is typically performed by administrators. Developers can use the **pg_show_*** series functions to query the permission status.

For details, see **12 User and Permissions Management**.

Step 4: Create Database Objects

After the design and authorization are complete, use DDL statements to create required database objects:

- Create a schema: **3.3 CREATE SCHEMA**.
- Create a table, including specifying the table format, partition, and table properties: **3.5 CREATE TABLE**.
- Create a view, function, model, and index: **3.6 CREATE VIEW**, **3.7 CREATE FUNCTION**, **3.8 CREATE MODEL** and **3.17 CREATE INDEX**.

 Copyright © Huawei Technologies Co., Ltd.

- Create a role: **3.21 CREATE ROLE**.

For details, see **3 DDL Syntax**.

Step 5: Operate Data

Use DML statements to add, delete, modify, merge, and optimize data in tables:

- Insert data: **4.3 INSERT**.
- Update data: **4.8 UPDATE**.
- Delete data: **4.5 DELETE**.
- Merge data (Iceberg tables): **4.6 MERGE INTO**.
- Optimize small files (Iceberg tables): **4.7 OPTIMIZE**.

For details, see **4 DML Syntax**.

Step 6: Query Data

Use DQL statements to query data:

- Single-table query: SELECT (filtering), WHERE (filtering), ORDER BY (sorting), LIMIT (limiting), and GROUP BY (aggregation).
- Multi-Table join query: INNER JOIN (INNER/CROSS/NATURAL) and OUTER JOIN (LEFT/RIGHT/FULL).
- Complex query: WITH expressions.

For details, see **5 DQL Syntax**.

Step 7: Use Functions and Data Types

During development, you need to understand the data type system and available functions of Aura.

- For details about data types and their mapping to LakeFormation, see **8 Data Type**.
- For details about the conversion rules between types, see **9 Type Conversion**.
- For details about how to compile expressions, see **10 Expression**.
- For details about functions and operators (character, number, time and date, JSON, aggregation, window, Iceberg table service, and foundation model inference), see **11 Functions and Operators**.
- For details about how to avoid using keywords in naming, see **7 Keywords**.

Step 8: Tune Performance

If the query performance does not meet the expectation, analyze the execution plan, locate the bottleneck, and adjust the engine behavior by setting GUC parameters.

- For details about the tuning process, execution monitoring, execution plan analysis, plan hints, and tuning cases, see **13 Performance Tuning**.
- For details about the values and functions of each parameter, see **14 GUC Parameters**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax Text Conventions

To facilitate the understanding of syntax usage, the syntax text in this document is presented in the following format:

During development, you are advised to use case-insensitive formats where possible. You are advised to use lowercase letters consistently for database object names (such as table names, column names, function names, field names, etc.). Avoid inconsistent casing for the same object in different locations to prevent errors such as "object not found" caused by identifier case sensitivity.

Format	Description
UPPERCASE	Keywords (parts of the statement that remain unchanged) are in uppercase.
lowercase	Parameters (parts of the statement to be replaced with actual values) are in lowercase.
[ ]	Content within square brackets ([ ]) is optional.
...	Preceding elements may repeat.
[x | y | ...]	Select one or none from two or more options.
{ x | y | ... }	Select one from two or more options.
[x | y | ...] [...]	Select multiple parameters or none. If multiple, separate them with spaces.
[x | y | ...] [,...]	Select multiple parameters or none. If multiple, separate them with commas (,).
{ x | y | ... } [...]	Select multiple parameters, at least one. If multiple, separate them with spaces.
{ x | y | ... } [,...]	Select multiple parameters, at least one. If multiple, separate them with commas (,).

 Copyright © Huawei Technologies Co., Ltd.

3 DDL Syntax

3.1 Designing Database Objects

3.1.1 Creating and Managing Schemas

A schema is the logical organization of objects and data in a database. Schema management allows multiple users to use the same database without interfering with each other. Third-party applications can be added to corresponding schemas to avoid conflicts.

The same database object name can be used in different schemas in a database without causing conflicts. For example, both **a_schema** and **b_schema** can contain a table named **mytable**. Users with required permissions can access objects across multiple schemas in a database.

Creating a Schema

Run **CREATE SCHEMA** to create a schema. For more information, see **3.3 CREATE SCHEMA**.

```
CREATE SCHEMA myschema;
```

To create or access an object within a schema, the full object name comprises the schema name followed by the specific object name, separated by a period (.), for example, **myschema.table**.

Setting the Schema Search Path

The GUC parameter **search_path** specifies the schema search sequence. The parameter value is a series of schema names separated by commas (,). If no schema is specified during object creation, the object will be added to the first schema displayed in the search path. If there are objects with the same name in different schemas and no schema is specified for an object query, the object will be returned from the first schema containing the object in the search path.

- Run the SHOW command to view the current search path.

  ```
  SHOW SEARCH_PATH;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Run the SET command to change the default schema of the current session. For example, if the search path is set to **myschema1**, **myschema2**, **myschema1** is searched first.
  ```
  SET SEARCH_PATH TO myschema1, myschema2;
  ```

Using a Schema

If you want to create or access an object in a specified schema, the object name must contain the schema name. To be specific, the name consists of a schema name and an object name, which are separated by a dot (.).

- Create a table **mytable** in **myschema**. Create a table in **schema_name.table_name** format.
  ```
  CREATE TABLE myschema.mytable(id int, name varchar(20)) STORE AS ORC;
  ```
- Query all data in the table **mytable** in **myschema**.
  ```
  SELECT * FROM myschema.mytable;
   id | name
  ----+------
  (0 rows)
  ```

Viewing a Schema

- Use the **current_schema** function to view the current schema.
  ```
  SHOW current_schema;
   current_schema
  ----------------

  (1 row)
  ```
- To view the list of all schemas, run the following command:
  ```
  SHOW SCHEMAS;
  ```
- To view the details of a schema, run the following command:
  ```
  DESCRIBE SCHEMA myschema;
  ```

Schema Permission Control

By default, a user can only access database objects in its own schema. To access objects in other schemas, the user must have the **usage** permission of the corresponding schema.

Aura currently does not support permission-related operations through SQL. You can perform permission-related operations on the LakeFormation console. For details, see **Configuring LakeFormation Metadata Permissions**.

Drop Schema

- Run the **DROP SCHEMA** command to delete an empty schema (no database objects in the schema).
  ```
  DROP SCHEMA IF EXISTS myschema;
  ```
- By default, a schema must be empty before being deleted. To delete a schema and all its objects (such as tables, data, and functions), use the **CASCADE** keyword.
  ```
  DROP SCHEMA myschema CASCADE;
  ```

 Copyright © Huawei Technologies Co., Ltd.

3.1.2 Creating and Managing Tables

Creating a Table

The **CREATE TABLE** command is used to create a table. When creating a table, you can define the following content:

- Columns and their data types.
- Table distribution definition, that is, the distribution strategy of the table, which determines how the Aura database divides data among segments.
- Table storage format.
- Partitioned table definition.

Example: **CREATE TABLE** creates a table named **web_returns_p1**, storing data in the ORC file format.

```
CREATE TABLE web_returns_p1
(
    wr_returned_date_sk       integer,
    wr_returned_time_sk       integer,
    wr_item_sk                integer,
    wr_refunded_customer_sk   integer
) store as ORC;
```

Table Types

Aura supports two table types: managed table and external table.

Type	Description	Use Case
Managed table	Both data and metadata are managed by Aura. Users can manage metadata and data files through Aura. Deleting a table removes both its data and metadata. The data path must be a parallel file system.	• Service data requiring frequent updates. • Critical data needing protection against accidental deletion.
External table	Aura reads data files from a specified location through metadata or table schemas. Users cannot modify the actual data files through Aura. Deleting a table only removes its metadata, leaving the data files unaffected.	• Read-only analysis. • Fine-grained access control based on file paths.

CAUTION

For managed tables, data files are still stored in a conventional file system (for example, OBS parallel file system). Users can alter these files without informing Aura. However, doing so violates the expectations and agreements of Aura regarding managed tables, potentially leading to undefined errors.

Copyright © Huawei Technologies Co., Ltd.

Examples of using managed tables and external tables

```
-- Create a schema.
CREATE SCHEMA sales_schema WITH LOCATION 'obs://bucket/catalog1/sales_schema/';

-- Switch the schema.
SET CURRENT_SCHEMA TO sales_schema;

-- Create a managed table.
CREATE TABLE region_sales_info(id int, item varchar(128), sale_date date) partition by (region
varchar(128)) store as parquet;

-- Create an external table to point to the managed table.
CREATE EXTERNAL TABLE readonly_region_sales_info(id int, item varchar(128), sale_date date) partition by
(region varchar(128)) store as parquet location 'obs://bucket/catalog1/sales_schema/region_sales_info/';

-- Import service data from the managed table.
INSERT INTO region_sales_info VALUES
(0, 'apple', '2001-01-01', 'Region A'),
(1, 'banana', '2001-01-02', 'Region B'),
(2, 'carrot', '2001-01-03', 'Region C'),
(3, 'desk', '2001-01-04', 'Region D');

SELECT * FROM region_sales_info order by 1;
 id |  item  | sale_date  |  region
----+--------+------------+----------
  0 | apple  | 2001-01-01 | Region A
  1 | banana | 2001-01-02 | Region B
  2 | carrot | 2001-01-03 | Region C
  3 | desk   | 2001-01-04 | Region D
(4 rows)

-- No data can be queried as the external table initially does not contain partition information.
SELECT * FROM readonly_region_sales_info;
 id | item | sale_date | region
----+------+-----------+--------
(0 rows)

-- Restore the partition information of the external table.
MSCK REPAIR TABLE readonly_region_sales_info;
SELECT * FROM readonly_region_sales_info order by 1;
 id |  item  | sale_date  |  region
----+--------+------------+----------
  0 | apple  | 2001-01-01 | Region A
  1 | banana | 2001-01-02 | Region B
  2 | carrot | 2001-01-03 | Region C
  3 | desk   | 2001-01-04 | Region D
(4 rows)

-- Drop the external table does not affect data files.
DROP TABLE readonly_region_sales_info;
SELECT COUNT(1) from region_sales_info;
 count
-------
     4
(1 row)

-- Dropping the managed table also deletes the data.
DROP TABLE region_sales_info;
```

Viewing Table Data

- Query information about all tables in the current schema.
  ```
  SHOW TABLES;
  ```
- Query the attributes of a table.
  ```
  DESCRIBE web_returns_p1;
  ```
- Query the data volume of the **web_returns_p1** table.
  ```
  SELECT count(*) FROM web_returns_p1;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Query all data from the **web_returns_p1** table.
  ```
  SELECT * FROM web_returns_p1;
  ```
- Query only the data of the **c_customer_sk** field.
  ```
  SELECT c_customer_sk FROM web_returns_p1;
  ```
- Filter out duplicate data of the **c_customer_sk** field.
  ```
  SELECT DISTINCT( c_customer_sk ) FROM web_returns_p1;
  ```
- Query all data whose **c_customer_sk** is **3869**.
  ```
  SELECT * FROM web_returns_p1 WHERE c_customer_sk = 3869;
  ```
- Sort data by the **c_customer_sk** field.
  ```
  SELECT * FROM web_returns_p1 ORDER BY c_customer_sk;
  ```

Deleting Table Data

When using a table, you may need to delete the table data.

- Currently, data cannot be deleted using the **DELETE** command.
- Executing the following command will delete all rows in the table.
  ```
  TRUNCATE TABLE customer_t1;
  ```
- Delete the created table.
  ```
  DROP TABLE customer_t1;
  ```

3.1.3 Defining Table Partitions

A partitioned table is a logical table that is divided into several physical blocks for storage based on a partitioning policy. The logical table is called a partitioned table, and the physical block is called a partition. A partitioned table is a logical table that does not store data. Data is stored on partitions. During conditional queries, the system scans only the partitions that meet the conditions, avoiding full table scanning and improving query performance.

Advantages of partitioned tables:

- Improved query performance. You can search in specific partitions, improving the search efficiency.
- Enhanced availability. If a partition is faulty, data in other partitions is still available.
- Improved maintainability. For expired historical data that needs to be periodically deleted, you can quickly delete it by dropping or truncate partitions.

Choosing to Partition a Table

You can choose to partition a table when the table has the following characteristics:

- The data contains fields with distinct enumerated types.

  Tables should be partitioned based on clearly defined enumerated fields such as date, region, or numerical values. Time-based fields are the most commonly used for partitioning.
- The table contains a large amount of data.

  Scanning small tables does not take much time, therefore the performance benefits of partitioning are not significant. Therefore, you are advised to partition only large tables.

 Copyright © Huawei Technologies Co., Ltd.

Partitioned Table Declaration

A table can be declared as a partitioned table only when it is created. For example:

```
CREATE TABLE my_tbl (x int) PARTITION BY (b int) STORE AS ORC;
INSERT INTO my_tbl VALUES (1,10);
```

Deleting a Partition

Run the **ALTER TABLE** statement to delete a partition from a partitioned table. For example, delete a partition from the **my_tbl** table.

```
ALTER TABLE my_tbl DROP PARTITIONS (b = 10);
```

Querying a Partition

- Query the partition b=10.
  ```
  SELECT * FROM my_tbl  where b = 10;
  ```
- View the information of the partitioned table.
  ```
  SHOW PARTITIONS my_tbl;
  DESCRIBE my_tbl;
  ```

Deleting a Partitioned Table

Run the **DROP TABLE** statement to delete a partitioned table.

```
DROP TABLE web_returns_p1;
```

3.2 SHOW

3.2.1 SHOW Conf

Function

Shows the current value of a runtime parameter. You can use the **SET** statement to set these parameters.

Precautions

Some parameters that can be viewed using **SHOW** are read-only. You can view but cannot set their values.

Syntax

```
SHOW
 {
   configuration_parameter |
   CURRENT_SCHEMA |
   TIME ZONE |

   SESSION AUTHORIZATION |
   ALL
 };
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- **configuration_parameter**

  Name of a runtime parameter.

  Range: You can view them by running the **SHOW ALL** command.

  NOTE

  Some parameters that can be viewed by running **SHOW ALL** cannot be set by running **SET**. For example, **max_datanodes**.

- **CURRENT_SCHEMA**

  Current schema.

- **TIME ZONE**

  Time zone.

- **SESSION AUTHORIZATION**

  User identifier of the current session.

- **ALL**

  All runtime parameters.

Examples

Show the value of **timezone**.

```
SHOW timezone;
```

Show the current setting of the **DateStyle** parameter.

```
SHOW DateStyle;
```

Show the current settings of all parameters.

```
SHOW ALL;
```

3.2.2 SHOW Schemas/Tables/Views/Partitions/Functions/Models

Function

SHOW is a syntax specific to Aura. This syntax is used to list sub-objects of a specified object in LakeFormation.

Precautions

None

Syntax

```
SHOW SCHEMAS;
SHOW TABLES IN schema_name;
SHOW PARTITIONS table_name;
SHOW VIEWS;
SHOW VIEWS IN schema_name;
SHOW FUNCTIONS;
SHOW FUNCTIONS IN schema_name;
SHOW MODELS;
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

None

Example

List all databases in the current catalog.

```
SHOW SCHEMAS;
```

List all tables in the **test_schema** database on LakeFormation.

```
SHOW TABLES IN test_schema;
```

List all partitions of the **test_table** table on LakeFormation.

```
SHOW PARTITIONS test_table;
```

List all views in the current database on LakeFormation.

```
SHOW VIEWS;
```

List all views in the **test_schema** database on LakeFormation.

```
SHOW VIEWS IN test_schema;
```

List all user-defined functions in the current database on LakeFormation.

```
SHOW FUNCTIONS;
```

List all functions in the **test_schema** database on LakeFormation.

```
SHOW FUNCTIONS IN test_schema;
```

List all models in the current catalog.

```
SHOW MODELS;
```

3.3 CREATE SCHEMA

Function

CREATE SCHEMA is specific to Aura. It is used to create a database with a specified name on LakeFormation.

Precautions

Aura does not have the cluster concept and does not store any metadata. All metadata is stored in LakeFormation. Therefore, after creating an external schema, you can log in to the LakeFormation console to view the created metadata.

Syntax

```
CREATE SCHEMA [ IF NOT EXISTS ] schema_name [ WITH LOCATION path ];
```

Parameter Description

- **schema_name**

  Database name of the target table.

 Copyright © Huawei Technologies Co., Ltd.

- **WITH LOCATION path**

  (Optional) Database storage directory. If no database directory is specified by the user, the system will automatically use the database directory with the same name as the storage directory of the database in the catalog directory.

- **path**

  Path for storing database files. When creating a schema, ensure that the target path does not contain other files.

Examples

Create a database directory.

```
CREATE SCHEMA test_schema;
```

Create a database directory and specify a storage directory for the database.

```
CREATE SCHEMA test_schema WITH LOCATION 'obs://xxxxx/yyyy/';
```

3.4 DROP SCHEMA

Function

DROP SCHEMA is specific to Aura. It is used to delete a database with a specified name from LakeFormation.

Precautions

If there is a large amount of user data, the statement execution may take a long time when schema data is deleted.

Syntax

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...] [ CASCADE | RESTRICT ];
```

Parameter Description

- **IF EXISTS**

  Reports a notice instead of an error if the specified schema does not exist.

- **schema_name**

  Name of the LakeFormation database.

- **CASCADE | RESTRICT**
  - **CASCADE**: Recursively deletes all objects under the database object from LakeFormation.
  - **RESTRICT**: Only the database object is deleted. If the database object contains any object, the deletion fails. This is the default action.

Examples

Delete the **test_schema** database. If the database has cascading objects, an error is reported.

```
DROP SCHEMA test_schema;
```

 Copyright © Huawei Technologies Co., Ltd.

Automatically deletes the **test_schema** database and all objects in the database.

```
DROP SCHEMA test_schema CASCADE;
```

3.5 CREATE TABLE

Function

CREATE TABLE is specific to Aura. It is used to create a table with a specified name on LakeFormation.

For details about expression-partitioned tables, see **Expression Partitioning**.

Precautions

- Columns appearing in **PARTITION BY** must not be described among the regular columns of the table. Partition columns are always positioned after the regular columns.
- **PARTITION BY** supports a maximum of four partition keys.
- **CLUSTERED BY** is applicable to ORC and Parquet file formats.
- **PARTITION BY** supports ORC, Parquet, Hudi, and Iceberg file formats.
- **ctid** and **xc_node_id** cannot be used as column names.
- For details about the main constraints on the currently implemented expression partitioning, see **Notes and Constraints**.

Syntax

```
CREATE [ EXTERNAL ] TABLE [ IF NOT EXISTS ] [schema_name.]table_name
[ ( col_name col_type [ COMMENT col_comment ] [, ... ] ) ]
[ COMMENT table_comment ]
[ PARTITION BY ( col_name col_type COMMENT col_comment | year(col_name) | month(col_name) |
day(col_name) | hour(col_name)| truncate(width, col_name) | bucket(bucket_num, col_name)[, ... ] ) ]
[ CLUSTERED BY (col_name [,...]) INTO ( bucket_num ) BUCKETS ]
[ TABLEPROPERTIES ( option_key = option_value [, ... ] ) ]
[ STORE AS table_format ]
[ LOCATION table_path ]
AS select_stmt;
```

Parameter Description

- **EXTERNAL**

  Specifying this keyword creates an EXTERNAL table. Omitting it defaults to creating a MANAGED table, where files, metadata, and statistical information are all managed by Aura.

- **IF NOT EXISTS**

  Issues a notice rather than throwing an error if a table with the same name already exists.

- **schema_name**

  The name of the database to which the table belongs. If no database name is specified, the table is created within the **current_schema**.

- **table_name**

  The name of the table to be created, which should not exceed 63 characters in length.

 Copyright © Huawei Technologies Co., Ltd.

- **col_name**

  The name of the column to be created in the table, also limited to 63 characters. The combined total of regular and partition columns must not exceed 5,000.

- **col_type**

  The data type of the column to be created, with supported ranges as follows:

Column Type	Can Be Declared As Partition Column	ORC Format	Parquet Format	Iceberg Format	Lance Format
SmallInt	√	√	√	√	√
Int	√	√	√	√	√
BigInt	√	√	√	√	√
Float	×	√	√	√	√
Double	×	√	√	√	√
Decimal	√	√	√	√	√
Numeric	√	√	√	√	√
Timestamp	√	√	√	√	√
Timestamptz	√	√	√	√	√
Date	√	√	√	√	√
Varchar	√	√	√	√	√
Char	√	√	√	√	√
Bool	×	√	√	√	√
Bytea	×	√	√	√	√
Text	√	√	√	√	√
Array	×	√	√	√	√
Struct	×	√	√	√	√

- **col_comment**

  Comment for the column, which can be any string.

- **table_comment**

 Copyright © Huawei Technologies Co., Ltd.

Comment for the table, which can also be any string.

- **bucket_num**

  Number of buckets.

- **option_key = option_value**

  Setting parameters at the table level, with supported parameter ranges as follows:

Table 3-1 Value range of the option_key parameter

option_key	option_value	Description	Applicable Scope
orc.compress	**zstd**, **snappy**, **lz4**, **uncompressed**	Compression method for ORC files.	ORC
orc.stripe.byte.size	An integer greater than or equal to 1,000,000	Stripe size of an ORC file	
parquet.compression	**zstd**, **snappy**, **lz4**, **uncompressed**	Compression method for Parquet files.	Parquet
parquet.compression-level	An integer ranging from –131,072 to 22	Compression mode for writing data to a Parquet table next time. The compression level can be specified only when the zstd algorithm is used. The value ranges from –131,072 to 22, and the default value is **1**.	
parquet.row.group.byte.size	An integer greater than or equal to 1,000,000	Row group size of a Parquet file	
parquet.page.size	Integer	Page size of a Parquet file	
parquet.dictionary.page.size	Integer	Dictionary size of a Parquet file	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
julian_adjust	**true**, **false**	Whether to convert to Julian date.	
parquet.bloom.filter.enabled.column.col	**true**, **false**	Whether to write the **col1** column to the Bloom filter index	
checkencoding	**high**, **low**, **no**	Whether to check character encoding.	ORC, Parquet
column_index_access	**true**, **false**	Matching method between table-defined columns and file columns during reading. Defaults to **true** for column index matching, **false** for column name matching.	
filesize	An integer ranging from 1 to 1024	Size of external table files generated.	
write.delete.mode	**copy-on-write**, **merge-on-read**	Mode setting for deletion: cow or mor.	Iceberg
write.update.mode	**copy-on-write**, **merge-on-read**	Mode setting for updates: cow or mor.	
write.merge.mode	**copy-on-write**, **merge-on-read**	Mode setting for merges: cow or mor.	
format-version	2 and 3	Iceberg table version	
write.parquet.compression-codec	**zstd**, **lz4**, **snappy**, **gzip**, **uncompressed**	Compression method for Parquet files.	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
write.parquet.compression-level	An integer ranging from –131,072 to 22	Compression level for writing the data in an Iceberg table to a Parquet file next time using the zstd algorithm. The compression level can be specified only when the zstd algorithm is used. The value ranges from –131,072 to 22, and the default value is **1**. If the behavior of this option is inconsistent with the default behavior of Iceberg, the formats that are not supported by Iceberg do not take effect by default, and the compression level can be specified for gzip.	
write.merge.isolation-level	**snapshot**, **serializable**	Isolation level for merge commands.	
write.metadata.delete-after-commit.enabled	**true**, **false**	Controls whether the oldest tracked version metadata file is deleted after committing.	
write.update.isolation-level	**snapshot**, **serializable**	Isolation level for update commands.	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
write.delete.isolation-level	**snapshot**, **serializable**	Isolation level for delete commands.	
write.metadata.previous-versions-max	An integer greater than 0	Number of older metadata files to retain.	
write.parquet.bloom-filter-enabled.column.col1	**true**, **false**	Whether to write the **col1** column to the bloom filter index.	
metafile_location	**metadata.json** path	**metadata.json** path specified for creating an external Iceberg table	
commit.retry.num-retries	An integer greater than 0	Number of retries upon a commit failure	
commit.retry.min-wait-ms	An integer greater than 0	Minimum wait time after a failed commit	
commit.retry.max-wait-ms	An integer greater than 0	Maximum time to wait before retrying a commit	
commit.retry.total-timeout-ms	An integer greater than 0	Total retry timeout period for a commit	
write.parquet.row-group-size-bytes	An integer greater than 0	Row group size of a Parquet file	
write.parquet.page-size-bytes	An integer greater than 0	Page size of a Parquet file	
write.parquet.page-row-limit	An integer greater than 0	Maximum number of rows on a Parquet file page	
write.parquet.dict-size-bytes	An integer greater than 0	Dictionary size of a Parquet file	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
write.parquet.bloom-filter-max-bytes	An integer greater than 0	Maximum number of bytes for a bloom filter bitset	
write.target-file-size-bytes	An integer greater than 0	Size of the Parquet file	
write.delete.target-file-size-bytes	An integer greater than 0	Size of the **delete** file	
write.format.default	parquet	Format of the data written to an Iceberg file. Only **parquet** is supported.	
header	**true**, **false**	Whether the CSV file includes a header row with column names.	CSV
separatorChar	Single-byte character (not **\r** or **\n**)	Field delimiter for CSV data files. Defaults to **,** if not set.	
quoteChar	Single-byte character. Must differ from the delimiter and null parameters.	Defines the character recognized as a quotation mark in CSV files. Default is double quotes (").	
escapeChar	Single-byte character	Escape character in CSV files. Defaults to the same value as quoteChar.	
field.delim	Single-byte character, which cannot be '\ \.abcdefghijklmnopqrstuvwxyz0123456789'	Field delimiter for TEXT data files. Defaults to **\t** if not set.	Text

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
serialization.null.format	Single-byte character	Null values in data files. Defaults to **\n**.	
escape.delim	Single-byte character	Escape character in TEXT files. Defaults to the same value as quoteChar.	
line.delim	Single-byte character	Line delimiter style in data files. Defaults to **\n**.	
drop.empty.lines	**true**, **false**	Whether to discard empty lines or single-line \n, \r, and \t in the text.	
fill_missing_fields	**true**, **false**	Whether to report an error when the number of columns in the result of splitting a line by field.delim is less than the number of columns.	
ignore_extra_data	**true**, **false**	Whether to report an error when the number of columns in the result of splitting a line by field.delim is greater than the number of columns.	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
skip.empty.files	**true**, **false**	Whether to read the JSON object whose content is '{}'. The default value is **false**.	JSON
nested.separator	Single-byte character, which cannot be a number sign (#).	Delimiter for nested JSON key concatenation. For example, if you set the delimiter to an underscore (_), **{"a":{"b":1}}** is mapped to column **a_b**, and the value of column **a_b** is **1**. By default, nested JSON read and write are not supported.	
ignore.nul.fields	**true**, **false**	Whether to skip writing keys with NULL values when writing a JSON object. The default value is **false**.	
max.rows.per.file	An integer from 1 to 9,223,372,036,854,775,807	Maximum number of rows that can be written to a file	Lance, Text, CSV, JSON
enable.blob	**true**, **false**	Whether to use BLOB for this column	Lance
arrow.fixed-size-list	An integer from 1 to 2,147,483,647	Number of items in each row for FixedSizeList	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
max.bytes.per.file	An integer from 1 to 9,223,372,036,854,775,807	Maximum number of bytes that can be written to a file	
lance.encoding.compression	A string, which can be **zstd**, **lz4**, or **fsst**	Compression mode for writing data to a Lance table next time. The compression level can be specified only when the zstd algorithm is used. The value ranges from 0 to 22, for example, **zstd:22**.	
lance.batch.readahead	An integer from 1 to 9,223,372,036,854,775,807	Number of batches to be prefetched	
lance.fragment.readahead	An integer from 1 to 9,223,372,036,854,775,807	Number of fragments to be prefetched	
lance.storage.version	A string, which can be **0.1**, **2.0**, **2.1**, or **2.2**	Version of the Lance storage format	
lance.block.size	An integer from 1 to 34,359,738,368	Minimum number of bytes in a single I/O request sent to the storage layer	
metadata.cache.size	An integer from 1 to 34,359,738,368	Size of the metadata cache, in bytes	
io.buffer.size	An integer from 1 to 34,359,738,368	Size of the I/O buffer, in bytes	

 Copyright © Huawei Technologies Co., Ltd.

option_key	option_value	Description	Applicable Scope
index.cache.size	An integer from 1 to 34,359,738,368	Size of the index cache, in bytes	

- **table_format**

  Table storage format. The ORC, Parquet, Iceberg, Text, JSON, CSV, and Lance formats are supported.

  CAUTION

  Text, JSON, and CSV support only basic data import and export and do not support advanced functions, such as partitioning and ANALYZE operations. Therefore, they are not recommended as data storage or processing media.

- **table_path**

  Table storage path. The value must be a valid OBS path. OBS buckets and parallel file systems are supported. If the path is an OBS bucket path, the table is read-only. Otherwise, the table can be read and written. If the table to be created is a managed table, the table storage path cannot be specified. The table storage path is specified by the system as the path with the same name as the table name in the schema path, and the path must be empty during table creation.

- **select_stmt**

  Query statement. External tables do not support CTAS statements.

Examples

Create an ORC table.

```
CREATE TABLE test_table1 (a int, b bool, c text) store as orc;
```

For details about how to create an Iceberg expression-partitioned table, see **Creating an Expression-Partitioned Tables**.

3.6 CREATE VIEW

Function

CREATE VIEW is used to create a view with a specified name on LakeFormation.

Syntax

```
CREATE [ OR REPLACE ] [schema_name.]view_name
[ ( col_name col_type [ COMMENT col_comment ] [, ... ] ) ]
[ COMMENT view_comment ]
AS select_stmt;
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- **OR REPLACE**

  If a view with the same name exists, it will create a view and overwrite the metadata of the old view.

- **schema_name**

  The name of the database to which the view belongs. If no database name is specified, the view is created within the **current_schema**.

- **view_name**

  Name of the view to be created, with a length that cannot exceed 63 characters.

- **view_comment**

  Comment information for the view, which can be specified as any string.

- **select_stmt**

  Query statement.

Examples

Create a view based on the ORC table.

```
CREATE TABLE test_table1 (a int, b bool, c text) store as orc;
CREATE VIEW v as select * from test_table1;
```

3.7 CREATE FUNCTION

Function

Creates a custom function.

Syntax

Syntax for creating a custom function

```
CREATE [AGGREGATE] FUNCTION function_name
   ( [  { argname  argtype } ]  [, ...] )
   [ RETURNS [SETOF] rettype | RETURNS TABLE (  { column_name column_type  }  [, ...] )]
   LANGUAGE lang_name
   [ { IMMUTABLE  | STABLE  | VOLATILE } ]
   RUNTIME_VERSION = { 'version' }
   RUNTIME_ENV = { 'env_json' }
   HANDLER = 'function_name'
   COMMENT = 'comment'
   [ STRICT ]
   [ PACKAGES = ( 'package_name==[version]' [ , ...] ) ]
   [ IMPORTS = ( 'obs_file_path' ) ]
   [ AS 'definition' ]
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 3-2 CREATE FUNCTION parameters

Parameter	Description	Value Range
function_name	Name of the function to be created (it can be schema-qualified).	A string, which must comply with the naming convention You are advised to specify a schema when creating a function. Otherwise, the function will be created under the default schema **default_db**. When calling a custom function, a schema must be specified. Otherwise, the system will call a built-in function.
argname	Name of a function parameter	A string, which must comply with the naming convention
argtype	Type of a function parameter	For details, see **Data Type Mapping**.
rettype	Data type of the function's return value	For details, see **Data Type Mapping**.
column_name	Column name	-
column_type	Column type	-
LANGUAGE lang_name	Name of the language used to implement the function Custom functions currently support only Python	-
AGGREGATE	Whether the function is an aggregate function	-
IMMUTABLE	Indicates that the function always returns the same result when given the same parameter values.	If the function's input parameter is a constant, the function's value is computed during the optimizer stage. This allows earlier access to the expression value, enabling more accurate cost estimation and generating an optimal execution plan.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Value Range
STABLE	Indicates that the function cannot modify the database. For the same parameter values, the function's return value remains unchanged within the same table scan, but may vary across different SQL statements.	-
VOLATILE	Indicates that the function's value can change within a single table scan, hence no optimizations are applied.	-
PACKAGES	Third-party Python packages required by the function runtime environment	-
RUNTIME_VERSION	Specific Python version for the function runtime environment. Currently, only the 3.11 series is supported.	-
RUNTIME_ENV	Third-party Python packages required by the function runtime environment and the installation mode (venv). This parameter will replace the **PACKAGES** parameter in the future.	-
HANDLER	Main function entry	-
IMPORTS	Path to the compressed package on OBS that the function depends on at runtime.	Only one path can be introduced in the **IMPORTS** clause, that is, one compressed package.
STRICT	Used to specify that the function always returns NULL if any of its parameters are NULL. Declaring this parameter means the function will not execute with NULL-valued parameters. Instead, it automatically returns a NULL result.	-

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Value Range
definition	Implementation of the function body	The default value is null, and the length cannot exceed 1,000 characters.
comment	Description of the function	-

3.8 CREATE MODEL

Function

Saves the meta information of a large model to LakeFormation. It is invoked by **ai_generate** in **11.21 Large Model Inference Functions**.

Constraints

It can only be invoked by remote large models.

Syntax

```
CREATE MODEL model_name
( remote_model_name = 'remote_model_name',base_url = 'base_url',
api_key = 'api_key'[, description = 'description'] );
```

Parameter Description

Table 3-3 CREATE MODEL parameters

Parameter	Description	Value Range
model_name	Name of the model to be created (it cannot be schema-qualified). It can only be invoked by Aura.	A string, which must comply with the naming convention
remote_model_name	Name of the model provided by the model vendor	String
base_url	URL provided by the model vendor	A string
api_key	Key required for accessing the remote model	A string
description	Custom description	The default value is null, and the length cannot exceed 2,048 characters.

 Copyright © Huawei Technologies Co., Ltd.

3.9 DROP TABLE

Function

DROP TABLE is an Aura syntax used to delete a table with a specified name from LakeFormation.

Precautions

When a managed table is deleted, the metadata, statistics, and data of the table are also deleted. If the data volume is large, the statement execution may take a long time. When an external table is deleted, only the metadata and statistics of the table are deleted, but the data is not deleted.

Syntax

```
DROP TABLE [IF EXISTS] table_name;
```

Parameter Description

- **IF EXISTS**

  Reports a notice instead of an error if the specified table does not exist.
- **table_name**

  Name of the LakeFormation table.

Examples

Delete the **test_table** table.

```
DROP TABLE test_table;
```

3.10 DROP VIEW

Function

DROP VIEW deletes a view with a specified name from LakeFormation.

Syntax

```
DROP VIEW [IF EXISTS] view_name;
```

Parameter Description

- **IF EXISTS**

  If the specified view does not exist, a message is displayed without any error message.
- **view_name**

  Name of the LakeFormation view.

 Copyright © Huawei Technologies Co., Ltd.

Examples

Delete the **test_view** view, but do not delete the tables on which the view depends.

```
DROP VIEW test_view;
```

3.11 DROP FUNCTION

Function

Deletes an existing function.

Precautions

- Built-in functions cannot be deleted. Only user-defined functions can be deleted.

Syntax

```
DROP FUNCTION [ IF EXISTS ] function_name;
```

Parameter Description

Table 3-4 DROP FUNCTION parameters

Parameter	Description	Value Range
IF EXISTS	Issues a notice instead of throwing an error if the specified function does not exist.	-
function_name	Name of the function to be deleted.	Existing function name.

Examples

Delete a user-defined function.

```
DROP FUNCTION your_schema.func_add;
```

3.12 DROP MODEL

Function

Deletes an existing model.

Syntax

```
DROP MODEL [ IF EXISTS ] model_name;
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 3-5 DROP MODEL parameters

Parameter	Description	Value Range
IF EXISTS	Issues a notice instead of throwing an error if the specified model does not exist.	-
model_name	Name of the model to be deleted	Name of an existing model

3.13 ALTER TABLE

Function

ALTER TABLE is a unique syntax within Aura. It modifies metadata information for tables on LakeFormation, enabling adjustments to table structures or attributes.

Constraints

- The **ADD COLUMNS**, **DROP COLUMNS**, **COLUMN RENAME**, and **ALTER COLUMN** syntaxes take effect only for Iceberg tables.
- Column position operations and complex type field operations apply exclusively to Iceberg tables.
- Only Iceberg tables support creating and deleting branches and tags.
- External tables do not support **ALTER TABLE**.

Syntax

```
ALTER TABLE table_name DROP PARTITIONS ( col_name = col_value [, ... ] ) [, ... ];
ALTER TABLE table_name ADD COLUMN col_name col_type [ COMMENT col_comment ] [ FIRST |  BEFORE col_name | AFTER col_name ] [, ... ];
ALTER TABLE table_name DROP COLUMNS ( col_name [, ... ] );
ALTER TABLE table_name RENAME COLUMN col_name TO col_name_new;
ALTER TABLE table_name ALTER COLUMN col_name [ TYPE col_type ] [ COMMENT col_comment ] [ FIRST |  BEFORE col_name | AFTER col_name ] [, ... ];
ALTER TABLE table_name RENAME TO table_name_new;
ALTER TABLE table_name SET TABLEPROPERTIES ( option_key = option_value [, ... ] );
ALTER TABLE table_name UNSET TABLEPROPERTIES ( option_key [, ... ] );
ALTER TABLE table_name UPDATE COLUMNS;
ALTER TABLE table_name ADD PARTITION COLUMN transform [, ...];
ALTER TABLE table_name DROP PARTITION COLUMN transform [, ...];
ALTER TABLE table_name SET PARTITION COLUMN transform [, ...];
ALTER TABLE table_name OWNER TO USER RoleId;
ALTER TABLE table_name OWNER TO USER GROUP RoleId;
ALTER TABLE table_name CREATE [OR REPLACE] BRANCH [IF NOT EXISTS] <branch_name>[AS OF VERSION <snapshot_id>][RETAIN <int> { DAYS | HOURS | MINUTES }][WITH SNAPSHOT RETENTION { minSnapshotsToKeep | maxSnapshotAge | minSnapshotsToKeep maxSnapshotAge }]
minSnapshotsToKeep ::= <int> SNAPSHOTS maxSnapshotAge ::= <int> { DAYS | HOURS | MINUTES }
ALTER TABLE table_name CREATE [OR REPLACE] TAG [IF NOT EXISTS] <tag_name>[AS OF VERSION <snapshot_id>][RETAIN <int> { DAYS | HOURS | MINUTES }]
ALTER TABLE table_name DROP { BRANCH [IF EXISTS] <branch_name> | TAG [IF EXISTS] <tag_name> }
The transform is as follows:
col_name | year(col_name) | month(col_name) | day(col_name) | hour(col_name)| truncate(width, col_name) | bucket(bucket_num, col_name)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

In the **ALTER TABLE** syntax, the parameter specifications are the same as those for table creation.

Examples

Add the **sales bigint** column to the **test_table** table.

```
ALTER TABLE test_table ADD COLUMN sales bigint COMMENT 'factory sales volume';
```

Rename the **sales** column in the **test_table** table to **my_sales**.

```
ALTER TABLE test_table RENAME COLUMN sales TO my_sales;
```

Change the type of the **my_sales** column in the **test_table** table to **String**.

```
ALTER TABLE test_table ALTER COLUMN my_sales TYPE text COMMENT 'factory sales volume';
```

Delete the **my_sales** column from the **test_table** table.

```
ALTER TABLE test_table DROP COLUMNS ( my_sales );
```

Rename the **test_table** table to **factory_info**.

```
ALTER TABLE test_table RENAME TO factory_info;
```

Enable the **hoodie.metadata.enable** parameter in the **factory_info** table.

```
ALTER TABLE factory_info SET TABLEPROPERTIES ( 'hoodie.metadata.enable' = 'true' );
```

Restore the parameters in the **factory_info** table.

```
ALTER TABLE factory_info UNSET TABLEPROPERTIES ( 'hoodie.metadata.enable' );
```

Synchronize the partition information of the **factory_info** table.

```
ALTER TABLE factory_info UPDATE COLUMNS;
```

3.14 DESCRIBE

Function

DESCRIBE is specific to Aura. This syntax is used to display details about a specified object in LakeFormation.

Syntax

```
DESCRIBE table_name;
DESCRIBE SCHEMA schema_name;
DESCRIBE view_name;
DESCRIBE FUNCTION function_name;
DESCRIBE MODEL model_name;
```

Example

Describe details about a target table.

```
DESCRIBE test_table;
```

Describe details about a target database.

```
DESCRIBE SCHEMA test_schema;
```

 Copyright © Huawei Technologies Co., Ltd.

Describe details about a target view.

```
DESCRIBE test_view;
```

Describe details about function **function_name** of **test_schema**.

```
DESCRIBE FUNCTION test_schema.function_name;
```

Describe details about a target model.

```
DESCRIBE model_name;
```

3.15 MSCK REPAIR TABLE

Function

MSCK REPAIR TABLE is specific to Aura. This syntax is used to synchronize partition information from the data catalog to the metadata storage engine.

Precautions

None.

Syntax

```
MSCK REPAIR TABLE table_name;
```

Parameter Description

None.

Examples

Synchronize partition information.

```
MSCK REPAIR TABLE table_name;
```

3.16 TRUNCATE

Function

TRUNCATE is specific to Aura. It is used to clear table data.

Precautions

External tables do not support **TRUNCATE**.

Iceberg tables do not support the **TRUNCATE PARTITION** syntax.

Syntax

```
TRUNCATE TABLE table_name [ PARTITIONS (col_name = col_value [, col_name = col_value ]) [, ... ] ];
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- **table_name**
  Name of the target table whose data is to be cleared.
- **PARTITIONS**
  If **PARTITIONS** is specified, only the data in the target partition is cleared.
- **col_name**
  Partition column name.
- **col_value**
  Partition data value

Examples

Delete the data of the table partition whose ID is 2.

```
TRUNCATE TABLE table_name PARTITIONS (id = 2);
```

3.17 CREATE INDEX

Function

CREATE INDEX can be used to create indexes for one or more columns of a foreign table.

An index is an object that creates a record for each value in the index column for quickly locating one or more rows in the foreign table.

Currently, Aura supports vector indexes for Lance foreign tables.

Constraints

- Currently, only vector indexes for Lance foreign tables are supported.
- The vector index algorithm can be any of {ivfflat, ivfpq, ivfrq, ivfsq, hnsw, hnswpq, hnswsq}. The supported distance functions include vector_l2_ops and vector_cosine_ops.
- Only one type of distance index can be created for each vector column, for example, vector_l2_ops|vector_cosine_ops. If an incorrect distance function is used, the query recall may be affected.
- You cannot create indexes with the same index name (**index_name**). To rebuild an index, use the ALTER INDEX ... REBUILD statement. To modify parameters, you need to delete the index (using DROP INDEX) and then use the new parameters to create an index (using CREATE INDEX). IF NOT EXISTS is recommended for preventing errors caused by the creation of indexes with the same name.
- NULL vectors and vectors with NULL values will not be indexed.
- The double type is not supported. You need to use a floating-point vector with a specified precision (array<float4> or array<float8>) and use TBLPROPERTIES to specify the dimension of the vector when creating a table, for example, **TBLPROPERTIES ('arrow.fixed-size-list.size.{nameOfThisVectorColumn}' = '128')**. This prevents other DDL operations (ALTER COLUMN) during index creation. float4/8 corresponds to float32/64.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
CREATE INDEX [ IF NOT EXISTS ] [index_name] ON table_name
[ USING access_method ]
( col_name opt_opclass [, ...] )
[ WITH ( option_key = option_value [, ... ] ) ];
```

Parameter Description

- **index_name**: index name
- **table_name**: name of the table for which an index is to be created
- **access_method**: index building algorithm, for example, IVFPQ for vector indexes
- **col_name**: name of the column for which an index is to be created
- **opt_opclass**: index column operator class. For example, if the cosine distance is used to create an index, the value is **vector_cosine_ops**.
- **option_key**: index parameter variable. For details, see the method of setting index parameters.
- **option_value**: index parameter value. For details, see the method of setting index parameters.

Table 3-6 Method of setting index parameters

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
ivfflat	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 4,096 or a value set using sqrt(num_rows) for large tables
	sample_rate	Number of vectors sampled for each center	with (sample_rate=?)	256, 1024, ... The table must contain at least **sample_rate** data records (a clustering bucket).
	max_iters	Number of iterations during k-means clustering	with (max_iters=?)	25, 50, ...
ivfpq	nlists, lists, or num_partitions	Number of clusters generated from the entire high-dimensional vector space	with (nlists=?)	num_rows // 4,096 or a value set using sqrt(num_rows) for large tables. In addition, the **lists/ num_sub_vectors % 8 =0** constraint is recommended for more efficient SIMD execution.

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
	num_sub_vectors	Number of sub-vectors a vector is split into during product quantization	with (num_sub_vectors=?)	dimension // 8, rounded to the nearest integer, for example, {4, 8, 16}. The hard constraint is as follows: num_sub_vectors < dim (number of column dimensions). When the value of dimension is less than or equal to 256, PQ is recommended.
	sample_rate	Number of vectors sampled during the calculation of the number of global clusters and quantization subvector cluster centers	with (sample_rate=?)	256, 1024, ...; Constraint: num_rows ≥ nlists x sample_rate^2
	max_iters	Number of iterations during the calculation of the number of global clusters and quantization subvector cluster centers	with (max_iters=?)	25, 50, ...
ivfrq	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 4,096 or a value set using sqrt(num_rows) for large tables
	num_bits	Number of quantization bits	with (bits=?)	Currently, only 1 bit is supported.

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
	sample_rate	Number of vectors sampled for each center	with (sample_rate=?)	256, 1024, ... The table must contain at least **sample_rate** data records (a clustering bucket).
	max_iters	Number of iterations during k-means clustering	with (max_iters=?)	25, 50, ...
ivfsq	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 4,096 or a value set using sqrt(num_rows) for large tables
	num_bits	Number of quantization bits	with (bits=?)	{4, 8 (default)}
	sample_rate	Number of vectors sampled for each center	with (sample_rate=?)	256, 1024, ... The table must contain at least **sample_rate** data records (a clustering bucket).
	max_iters	Number of iterations during k-means clustering	with (max_iters=?)	25, 50, ...
hnsw/ hnswflat	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 1,048,576 or a value set using sqrt(num_rows) for large tables
	m	Number of edges of each node in an HNSW graph	with (m=?)	default 20
	ef or ef_construction	Search breadth for graph creation	with (ef=?)	>=150

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
	sample_rate	Number of vectors sampled for each center	with (sample_rate=?)	256, 1024, ... The table must contain at least **sample_rate** data records (a clustering bucket).
	max_iters	Number of iterations during k-means clustering	with (max_iters=?)	25, 50, ...
hnswpq	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 1,048,576 or a value set using sqrt(num_rows) for large tables
	m	Number of edges of each node in an HNSW graph	with (m=?)	default 20
	ef or ef_construction	Search breadth for graph creation	with (ef=?)	>=150
	num_sub_vectors	Number of sub-vectors a vector is split into during product quantization	with (num_sub_vectors=?)	dimension // 8, rounded to the nearest integer, for example, {4, 8, 16} Hard constraint: num_sub_vectors < dim (number of column dimensions)

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
	sample_rate	Number of vectors sampled during the calculation of the number of global clusters and quantization subvector cluster centers	with (sample_rate=?)	256, 1024, ...; Constraint: num_rows ≥ nlists x sample_rate^2
	max_iters	Number of iterations during the calculation of the number of global clusters and quantization subvector cluster centers	with (max_iters=?)	25, 50, ...
hnswsq	nlists, lists, or num_partitions	Number of clustering buckets	with (nlists=?)	num_rows // 1,048,576 or a value set using sqrt(num_rows) for large tables
	m	Number of edges of each node in an HNSW graph	with (m=?)	default 20
	ef or ef_construction	Search breadth for graph creation	with (ef=?)	>=150
	num_bits	Number of quantization bits	with (bits=?)	{4, 8 (default)}

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Parameter (option_key)	Description	Usage	Reference Value (option_value)
	sample_rate	Number of vectors sampled during the calculation of the number of global clusters and quantization subvector cluster centers	with (sample_rate=?)	256, 1024, ...; Constraint: num_rows ≥ nlists x sample_rate^2
	max_iters	Number of iterations during the calculation of the number of global clusters and quantization subvector cluster centers	with (max_iters=?)	25, 50, ...

Example

Create an index for the vector column of the lance_vector table using IVFPQ.

```
create table lance_vector (vector array<float4>) TABLEPROPERTIES ('arrow.fixed-size-list.vector' = '16') store as lance;
CREATE INDEX ix ON lance_vector USING ivfpq (vector vector_cosine_ops) WITH (nlists=2, num_sub_vectors=2);
```

3.18 DROP INDEX

Function

DROP INDEX can be used to delete an index.

Constraints

Currently, it can only delete indexes of Lance foreign tables.

Syntax

```
DROP INDEX [ IF EXISTS ] index_name ON table_name;
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- **index_name**: index name
- **table_name**: name of the table for which an index is to be created

3.19 SHOW INDEX

Function

SHOW INDEX prints the indexes that have been created for a Lance table. The returned table contains three columns: table name, index name, and index column name.

If the table name is specified, only all indexes of the current Lance table are returned.

If no table name is specified, the indexes of all Lance tables in the current database are returned.

Constraints

Currently, it can only return indexes of Lance foreign tables.

Syntax

```
SHOW INDEX [table_name];
```

Parameter Description

table_name: target table name

3.20 ALTER INDEX

Function

ALTER INDEX can be used to rebuild or optimize an existing index.

Constraints

Currently, it can only optimize indexes of Lance foreign tables.

Syntax

```
ALTER INDEX [ IF EXISTS ] index_name ON table_name (OPTIMIZE);
```

Parameter Description

- **index_name**: index name
- **table_name**: name of the table for which an index is to be created

 Copyright © Huawei Technologies Co., Ltd.

3.21 CREATE ROLE

Function

CREATE ROLE is used to create a role in LakeFormation.

Syntax

```
CREATE ROLE role_name [ COMMENT comment_text ];
```

Parameter Description

- **role_name**: name of the role
- **comment_text**: description

3.22 DROP ROLE

Function

DROP ROLE is used to delete a role in LakeFormation.

Syntax

```
DROP ROLE [ IF EXISTS ] role_name;
```

Parameter Description

role_name: name of the role

3.23 ALTER ROLE

Function

ALTER ROLE is used to modify a role in LakeFormation.

Syntax

- Syntax 1
  ```
  ALTER ROLE role_name [ options ] name_list;
  ```
  options can be:
  ```
  | ADD USER/USER GROUP/AGENT
  | DROP USER/USER GROUP/AGENT
  ```
- Syntax 2
  ```
  ALTER ROLE role_name COMMENT comment_text;
  ```

Parameter Description

- **role_name**: name of the role
- **name_list**: list of users, user groups, and agencies

 Copyright © Huawei Technologies Co., Ltd.

- **comment_text**: description

3.24 ALTER USER

Function

ALTER USER is used to associate a user with roles in LakeFormation.

Syntax

```
ALTER USER role_name [ WITH ] ROLE role_list;
```

Parameter Description

- **role_name**: name of the user
- **role_list**: list of role names

3.25 ALTER USER GROUP

Function

ALTER USER GROUP is used to associate a user group with roles in LakeFormation.

Syntax

```
ALTER USER GROUP groupname [ WITH ] ROLE role_list;
```

Parameter Description

- **groupname**: user group name
- **role_list**: list of role names

 Copyright © Huawei Technologies Co., Ltd.

4 DML Syntax

4.1 Overview

Data Manipulation Language (DML) is used to perform operations on data in database tables, such as inserting, updating, querying, and deleting data.

Inserting Data

Inserting data is to add one or more records to a database table. For details, see **4.3 INSERT**.

Querying Data

The database query statement **SELECT** is used to search for information that meets conditions in a database. For details, see **5.1.1 Basic Syntax**.

4.2 EXPLAIN

Function

Shows the execution plan of a SQL statement.

The execution plan shows the scan methods used for the tables referenced by the SQL statement. If multiple tables are referenced, the execution plan also displays the JOIN algorithms employed.

The most critical part of the execution plan is the estimated execution cost of the statement, which represents the planner's estimation of the time required to execute the statement.

If the **ANALYZE** option is specified, the statement will be executed, and statistical data based on the actual runtime will be displayed, including the total time cost (in milliseconds) and the actual number of rows returned within each plan node. This is useful for assessing whether the planner's estimates align with reality.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

When the **ANALYZE** option is specified, the statement will be executed.

Syntax

- Display the execution plan of a SQL statement, supporting multiple options with no specific order required:

```
EXPLAIN [ (  option  [, ...] )  ] statement;
```

  Where the syntax for the **option** clause is:

```
ANALYZE [ boolean ] |
   ANALYSE [ 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 } |
   GENERIC [ boolean ]
   BLOCKNAME [boolean] |
   OUTLINE [boolean]
```

- Display the execution plan of a SQL statement, providing options in order:

```
EXPLAIN  { [  { ANALYZE  | ANALYSE  }  ] [ VERBOSE  ]  | PERFORMANCE  } statement;
```

- Display the information necessary to reproduce the execution plan of a SQL statement, typically used for troubleshooting. The **STATS** option must be used independently:

```
EXPLAIN ( STATS [ boolean ] ) statement;
```

- Display detailed timing information for each execution step of DDL statements.

```
EXPLAIN PERFORMANCE statement;
```

Parameter Description

- **statement**

  Specifies the SQL statement to analyze.

- **ANALYZE boolean | ANALYSE boolean**

  Displays the actual runtime and other statistical data.

  Range:

  – **TRUE** (default): Displays the actual runtime and other statistical data.

  – **FALSE**: Does not display.

- **VERBOSE boolean**

  Displays additional information about the plan.

  Range:

  – **TRUE** (default): Displays additional information.

  – **FALSE**: Does not display.

- **COSTS boolean**

  Includes the estimated total cost for each plan node, along with the estimated number of rows and the width of each row.

 Copyright © Huawei Technologies Co., Ltd.

Range:

- **TRUE** (default): Displays the estimated total cost and width.
- **FALSE**: Does not display.

- **CPU boolean**

  Prints CPU usage information.

  Range:

  - **TRUE** (default): Displays the CPU usage.
  - **FALSE**: Does not display.

- **DETAIL boolean**

  Prints information on DNs.

  Range:

  - **TRUE** (default): Prints information on DNs.
  - **FALSE**: Does not print.

- **NODES boolean**

  Prints information about the node where the query is executed.

  Range:

  - **TRUE** (default): Prints information about the execution node.
  - **FALSE**: Does not print.

- **NUM_NODES boolean**

  Prints the number of nodes involved in execution.

  Range:

  - **TRUE** (default): Prints the number of DNs.
  - **FALSE**: Does not print.

- **BUFFERS boolean**

  Includes information about buffer usage.

  Range:

  - **TRUE**: Displays buffer usage.
  - **FALSE** (default): Does not display.

- **TIMING boolean**

  Includes the actual startup time and the time spent on the output node.

  Range:

  - **TRUE** (default): Displays startup time and time spent on the output node.
  - **FALSE**: Does not display.

- **PLAN**

  Whether to store the execution plan in **plan_table**. When this option is enabled, the execution plan is stored in **PLAN_TABLE** and not printed to the current screen. Thus, this option cannot be used simultaneously with others when enabled.

  Range:

  - **ON** (default): The execution plan is stored in **plan_table** and is not printed to the current screen. If the plan is successfully stored, **EXPLAIN SUCCESS** is returned.

 Copyright © Huawei Technologies Co., Ltd.

  - **OFF**: The execution plan is not stored but is printed to the current screen.
- **FORMAT**

  Specifies the output format.

  Range: **TEXT**, **XML**, **JSON**, and **YAML**.

  Default Value: **TEXT**
- **GENERIC boolean**

  Displays the generic plan generated by replacing constants in the statement with parameters.

  Range:

  - **TRUE** (default): Displays the generic plan.
  - **FALSE**: Displays the custom plan.
- **BLOCKNAME boolean**

  Displays blockname information for operators.

  Range: Boolean

  - **TRUE**: Displays blockname information.
  - **FALSE**: Does not display blockname information.

  Default Value: **TRUE**
- **OUTLINE boolean**

  Displays outline information extracted from the plan.

  Range: Boolean

  - **TRUE**: Displays outline information.
  - **FALSE**: Does not display outline information.

  Default Value: **TRUE**
- **PERFORMANCE**

  When using this option, all relevant information during execution is printed.
- **STATS boolean**

  Prints information required for reproducing the execution plan of a SQL statement, including the object definition, statistics, and configuration parameters. This information is typically used for troubleshooting.

  Range:

  - **TRUE** (default): Displays information required for reproducing the execution plan of a SQL statement.
  - **FALSE**: Does not display.

Examples

Display the execution plan of a simple table query.

```
EXPLAIN SELECT * FROM tpcds.customer_address_p1;
```

Helpful Links

6.1 ANALYZE | ANALYSE

 Copyright © Huawei Technologies Co., Ltd.

4.3 INSERT

Function

Inserts one or more rows of data into a table.

Precautions

- To insert data into a table, you must have the **INSERT** permission on the table.
- To use the **RETURNING** clause, you must have the **SELECT** permission on the table.
- If using the **QUERY** clause to insert data rows from a query, you must also have the **SELECT** permission on the table used in the query.
- If using the **OVERWRITE** clause to insert data in overwrite mode, you must additionally have the **SELECT** and **TRUNCATE** permissions on the table.

CAUTION

For processing ORC and PARQUET table data, you are advised to use the **INSERT OVERWRITE** syntax. Example:

- For inserting data into a non-partitioned table, use **INSERT OVERWRITE INTO tablename select_clause**.
- For inserting data into a partitioned table, use **INSERT OVERWRITE INTO tablename PARTITION(partvalue [, partvalue]) select_clause**.

Currently, the **INSERT** syntax does not guarantee transactional integrity for ORC and PARQUET formats. Using the **INSERT** syntax writes new data incrementally, which may lead to:

- Dirty reads, where partially written data is accessed.
- Residual data remaining in case of exceptions during the write process, compromising data consistency.

Syntax

```
[ WITH  with_query [, ...] ]
INSERT [/*+ plan_hint */] [ OVERWRITE ] INTO table_name [ partition_clause ] [ AS alias ]
[ ( column_name [, ...] ) ] [FOR VERSION AS OF <branch_name>]
   {
VALUES {( { expression | DEFAULT } [, ...] ) }[, ...]
   | query }
   ;

where partition_clause can be:

PARTITION ( partition_key_value [, ...] )
```

Parameter Description

- **WITH with_query [, ...]**

 Copyright © Huawei Technologies Co., Ltd.

Declares one or more subqueries that can be referenced by name in the main query, acting as temporary tables.

Detailed format of **with_query**:

```
with_query_name [ ( column_name [, ...] ) ] AS
( select)
```

– **with_query_name** specifies the name of the result set generated by the subquery, which can be used to access the subquery's result set in the query.

– **column_name** specifies the column names displayed in the subquery result set.

– Each subquery can be a **SELECT** statement.

- **plan_hint clause**

  Appears after keywords in the form of /*+ */. Used to provide hints for optimizing the execution plan of a specific statement block. For details, see **Plan Hint Optimization**.

- **OVERWRITE**

  Indicates the overwrite insertion mode. When using this mode, existing data in the target table is cleared, retaining only the newly inserted data.

  OVERWRITE supports inserting specified columns. Other columns take default values or NULL if no default exists.

  NOTE

  - Do not concurrently execute **OVERWRITE** with real-time write operations like **INSERT INTO**, as it risks unintended clearing of real-time written data.
  - **OVERWRITE** is suitable for batch data import scenarios and is not recommended for inserting small amounts of data.
  - Do not perform concurrent **INSERT OVERWRITE** operations on the same table.
  - **OVERWRITE** allows overwriting external OBS tables, deleting original directory data files.
  - External tables support single-partition **OVERWRITE** and multi-value level partitioning. This function is only supported in version 9.1.0.100 or later.

    Example: INSERT OVERWRITE into table_name partition(p1_value, p2_value...) ...

- **table_name**

  Name of the target table where data is to be inserted.

  Range: an existing table name.

- **AS**

  Alias for the target table **table_name**. *alias* is the alias name.

- **column_name**

  Column name in the target table.

  – The column name can be modified by a subfield name or array subscript.

  – **Each column not present in the column list is filled with a system default value or a declared default value, or NULL if none is present. For example, if you insert data into some columns of a composite type, other columns will be NULL.**

  – The target column (**column_name**) can be sorted in order. If no column is listed, all columns are listed by default, and the order is the same as that of the table declaration.

 Copyright © Huawei Technologies Co., Ltd.

  - If only *N* columns are provided in the value clause and query, the target columns are the first *N* columns.
  - The values provided by the **value** clause and **query** are associated to the corresponding columns from left to right in the table.

  Range: an existing column name.

- FOR VERSION AS OF <branch_name>

  Writes data to a specified branch. **branch_name** indicates the branch name. Only Iceberg tables are supported.

- **partition_key_value**

  Partition key value.

  The values specified by the **PARTITION(partition_key_value [, ...])** clause can uniquely identify a partition.

  Range: value range of the partition key of the partition to be renamed.

  Note that all **partition_key** values here must be of the identity expression type.

- **expression**

  Assign a valid expression or value to the corresponding column:

  - When inserting single quotation marks into a column in a table, you need to use the single quotation marks for escape.
  - If the expression for inserting a row is not of the correct data type, the system attempts to convert the data type. If the conversion fails, the data fails to be inserted and the system returns an error message.

  Example:

```
CREATE TABLE tt01 (id int,content varchar(50)) store AS orc;

INSERT INTO tt01 values (1,'Jack say ''hello''');
INSERT 0 1
INSERT INTO tt01 values (2,'Rose do 50%');
INSERT 0 1
INSERT INTO tt01 values (3,'Lilei say ''world''');
INSERT 0 1
INSERT INTO tt01 values (4,'Hanmei do 100%');
INSERT 0 1

SELECT * FROM tt01;
 id |      content
----+-------------------
  3 | Lilei say 'world'
  4 | Hanmei do 100%
  1 | Jack say 'hello'
  2 | Rose do 50%
(4 rows)
```

- **DEFAULT**

  Default value of the corresponding field name. If no default value is available, the value is **NULL**.

- **query**

  A query statement (**SELECT** statement) that uses the query result as the inserted data.

Examples

Create the **reason_t1** table.

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE TABLE reason_t1
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NAME           VARCHAR(20)
) store AS orc;
```

Insert a record into the table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NAME) VALUES (1, 'S01', 'StudentA');
```

Insert a record into the table. The syntax is equivalent to the previous syntax.

```
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 multiple 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_NAME
----------+----------+------------
        1 |      S01 |   StudentA
        2 |      T01 |   TeacherA
        3 |      T02 |   TeacherB
(3 rows)
```

Use **INSERT OVERWRITE** to update data in a table, that is, insert data in overwrite mode.

```
INSERT OVERWRITE INTO reason_t1 values (4, 'S02', 'StudentB');
SELECT * FROM reason_t1 ORDER BY 1;
 TABLE_SK | TABLE_ID | TABLE_NAME
----------+----------+------------
        4 |      S02 |   StudentB
(1 row)
```

Insert data back into the **reason_t1** table.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1;
```

Specify the default values for independent columns.

```
INSERT INTO reason_t1 VALUES (5, 'S03', DEFAULT);
```

To insert some data in a table to another table, perform the **WITH** subquery to obtain the temporary table **temp_t**, and then insert all data in the temporary table **temp_t** to the **reason_t1** table.

```
WITH temp_t AS (SELECT * FROM reason_t1) INSERT INTO reason_t1 SELECT * FROM temp_t ORDER BY 1;
```

Insert data into a specified partition of a partitioned table.

```
CREATE TABLE test_value_row(a int) PARTITION BY (d int) store AS orc;
INSERT OVERWRITE INTO test_value_row VALUES(55,51);
INSERT OVERWRITE INTO test_value_row VALUES(85,80);
INSERT OVERWRITE INTO test_value_row partition (d=80) VALUES(85);

ALTER TABLE test_value_row DROP PARTITIONS (d=51);
```

Optimization Suggestion

VALUES

When you use the insert statement to insert data in batches, you are advised to combine multiple records into one statement to improve data loading

 Copyright © Huawei Technologies Co., Ltd.

performance, for example, **INSERT INTO sections VALUES (30, 'Administration', 31, 1900),(40, 'Development', 35, 2000), (50, 'Development' , 60 , 2001)**.

4.4 VALUES

Function

Computes the value of one or more rows based on a given value expression. It is commonly used to generate a constant table within a larger command.

Precautions

- Do not use **VALUES** to return a very large number of result rows, as it may lead to memory exhaustion or poor performance. For the **INSERT INTO VALUES** syntax, you are advised to use it in scenarios involving small amounts of data insertion. For inserting large volumes of data, consider using other import methods provided by the product.
- If multiple rows are specified, each row must contain 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

- **expression**

  A constant or expression used to compute or insert into a specific location in the result table.

  In a **VALUES** list appearing at the top level of **INSERT**, expression can be replaced with **DEFAULT** to represent the default value of the destination column being inserted. However, **DEFAULT** cannot be used when **VALUES** appears in other contexts.

- **sort_expression**

  An expression or integer constant indicating how to sort the resulting rows.

- **ASC**

  Specifies sorting in ascending order.

- **DESC**

  Specifies sorting in descending order.

- **operator**

  A sorting operator.

- **count**

  Maximum number of rows to return.

- **start**

  Number of rows to skip before beginning to return rows.

 Copyright © Huawei Technologies Co., Ltd.

- **FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY**

  The **FETCH** clause limits the total number of rows returned from the first row of the query result, with the default value of count being **1**.

Examples

Create the **reason_t1** table.

```
CREATE TABLE reason_t1
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA         VARCHAR(20)
) store AS orc;
```

Insert a record into the table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

Insert a record into the table. The syntax is equivalent to the previous syntax.

```
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 multiple 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
       2 |      T01 |   TeacherA
       3 |      T02 |   TeacherB
(3 rows)
```

Use **INSERT OVERWRITE** to update data in a table, that is, insert data in overwrite mode.

```
INSERT OVERWRITE INTO reason_t1 values (4, 'S02', 'StudentB');
SELECT * FROM reason_t1 ORDER BY 1;
 TABLE_SK | TABLE_ID | TABLE_NA
----------+----------+------------
       4 |      S02 |   StudentB
(1 row)
```

Insert data back into the **reason_t1** table.

```
INSERT INTO reason_t1 SELECT * FROM reason_t1;
```

Specify the default values for independent columns.

```
INSERT INTO reason_t1 VALUES (5, 'S03', DEFAULT);
```

To insert some data in a table to another table, perform the **WITH** subquery to obtain the temporary table **temp_t**, and then insert all data in the temporary table **temp_t** to the **reason_t1** table.

```
WITH temp_t AS (SELECT * FROM reason_t1) INSERT INTO reason_t1 SELECT * FROM temp_t ORDER BY 1;
```

Insert data into a specified partition of a partitioned table.

```
CREATE TABLE test_value_row(a int)PARTITION BY (d int) store AS orc;
INSERT OVERWRITE INTO test_value_row  VALUES(55,51);
INSERT OVERWRITE INTO test_value_row  VALUES(85,80);
```

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER TABLE test_value_row DROP PARTITIONS (d=51);
```

4.5 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 are deleted, and only the table structure is retained.

NOTE

Currently, only Iceberg tables support the **DELETE** operation.

Precautions

To delete data from a table, you must have the **DELETE** permission on the table. You must also have the **SELECT** permission on the table referenced by the **USING** clause and the table read in the condition.

WARNING

Do not use **DELETE** to delete all data in a table. Consider using **TRUNCATE TABLE** instead.

Syntax

```
DELETE FROM table_name [ * ] [ [ AS ] alias ]
   [ USING using_list ]
   [ WHERE condition ]
   [FOR VERSION AS OF <branch_name>]
```

Parameter Description

- **table_name**: name of the target table. Range: an existing table name.
- **alias**: alias of the target table. Range: a string. It must comply with the naming convention.
- **using_list**: **USING** clause.
- **condition**: an expression that returns a boolean value, used to determine which rows need to be deleted.
- FOR VERSION AS OF <branch_name>: deletes data from a specified branch. **branch_name** indicates the branch name. Only Iceberg tables are supported.

Examples

Create the **reason_t2** table.

```
CREATE TABLE reason_t2
(
   TABLE_SK         INTEGER             ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA          VARCHAR(20)
```

 Copyright © Huawei Technologies Co., Ltd.

```
) STORE AS iceberg;

INSERT INTO reason_t2 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

Use the **WHERE** condition for deletion.

```
DELETE FROM reason_t2 WHERE TABLE_SK = 2;
DELETE FROM reason_t2 AS rt2 WHERE rt2.TABLE_SK = 2;
```

Use the **IN** syntax for deletion.

```
DELETE FROM reason_t2 WHERE TABLE_SK IN (1,3);
```

4.6 MERGE INTO

Scenario

In data processing and analysis scenarios, you often need to merge data from one source into another—for example, integrating incremental data into a main table. However, due to the lack of efficient single-statement support, users typically resort to combining multiple statements to achieve this functionality. This approach not only reduces processing efficiency but may also compromise data consistency and accuracy. Addressing these challenges requires ensuring both efficiency and reliability during data operations. To simplify this process, Aura introduces **MERGE INTO** syntax for Iceberg tables, enabling you to perform insert, update, or delete operations with a single statement. This enhancement streamlines data merging workflows, improving both efficiency and reliability.

Function

MERGE INTO matches data between target and source tables based on specified conditions. If a match is found, it performs an UPDATE or DELETE operation on the target table. If no match exists, it executes an INSERT. By consolidating UPDATE and INSERT operations into a single step, MERGE INTO enhances execution efficiency.

Precautions

- To execute **MERGE INTO**, you must have **UPDATE** and **INSERT** permissions on the target table and **SELECT** permission on the source table.
- The target table must be in Iceberg format.

Syntax

```
MERGE [/*+ plan_hint */] INTO table_name [ partition_clause ] [ [ AS ] alias ] [FOR VERSION AS OF
<branch_name>]
USING { { table_name | view_name } | subquery } [ partition_clause ] [ [ AS ] alias ]
ON ( condition )
{ when_matched_clause | when_not_matched_clause } [, ...]

where when_matched_clause can be:
WHEN MATCHED [AND condition] THEN
{ UPDATE SET { column_name = { expression | DEFAULT } | ( column_name [, ...] ) = ( { expression |
DEFAULT } [, ...] ) } [, ...] | DELETE }

and when_not_matched_clause can be:
WHEN NOT MATCHED [AND condition] THEN
INSERT { DEFAULT VALUES | [ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] }
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- plan_hint clause

  Appears after keywords in the form of /*+ */. Used to provide hints for optimizing the execution plan of a specific statement block. For details, see **Plan Hint Optimization**.

- INTO clause

  Defines the target table for updates or inserts:

  - **table_name**

    Name of the target table.

  - **partition_clause**

    Specifies partitions within the target table.

    The partition must already exist or correspond to valid partition key values.

  - **alias**

    Alias of the target table.

  - FOR VERSION AS OF <branch_name>

    Specifies the branch of the target table. **branch_name** indicates the branch name.

- USING clause

  Identifies the source table, which can be a table, view, or subquery.

  - **partition_clause**

    Specifies partitions within the source table.

    Only applicable if the source is a partitioned table. Views and subqueries do not support partitioning.

  - **alias**

    Alias of the source table.

- ON clause

  Defines the join condition between the target and source tables. Columns used in the join condition cannot be updated.

- WHEN MATCHED clause

  Executes **UPDATE** or **DELETE** operations when records in the source and target tables match based on the join condition.

  - Distribution columns cannot be updated. System tables and system columns are also not updatable.
  - An error will occur if a row is updated multiple times.

- WHEN NOT MATCHED clause

  The **WHEN NOT MATCHED** clause is used to **INSERT** data into the target table when no match is found between the source and target tables based on the given conditions.

  The **INSERT** clause does not support multiple **VALUES** statements.

  Both **WHEN MATCHED** and **WHEN NOT MATCHED** clauses can be reordered, omitted individually (but not both), or specified multiple times as needed.

 Copyright © Huawei Technologies Co., Ltd.

- DEFAULT

  Use this option to populate a field with its default value.
- AND condition

  Conditions for **WHEN MATCHED** and **WHEN NOT MATCHED** clauses ensure operations are performed only when the criteria are met. These conditions are optional.

Examples

Create the target table **products** and the source table **newproducts**, then insert sample data.

```
CREATE TABLE products
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
) STORE AS iceberg;

INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrncs'),
(1502, 'olympus is50', 'electrncs'),
(1600, 'play gym', 'toys'),
(1601, 'lamaze', 'toys'),
(1666, 'harry potter', 'dvd');

CREATE TABLE newproducts
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
) STORE AS iceberg;

INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrncs'),
(1601, 'lamaze', 'toys'),
(1666, 'harry potter', 'toys'),
(1700, 'wait interface', 'books');
```

Perform a **MERGE INTO** operation: Update rows where **product_id** matches, but only if the target table's **product_name** is not **'play gym'** and the source table's **product_name** is not **'lamaze'**. Delete rows where the source table's **product_name** is **'lamaze'**. Insert rows where **product_id** does not match and the source table's category is **'books'**.

```
MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED AND p.product_name != 'play gym' and np.product_name != 'lamaze' THEN
  UPDATE SET product_name = np.product_name, category = np.category
WHEN MATCHED AND np.product_name = 'lamaze' THEN
  DELETE
WHEN NOT MATCHED AND np.category = 'books' THEN
  INSERT VALUES (np.product_id, np.product_name, np.category)
;
```

Query results show updates: Rows with **product_id** values 1502 and 1666 were updated, row 1601 was deleted, and 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
       1666 | harry potter   | toys
```

 Copyright © Huawei Technologies Co., Ltd.

```
    1700 | wait interface | books
(5 rows)
```

4.7 OPTIMIZE

Function

Rewrites specified data in a target table.

NOTE

Currently, this syntax is only supported for Iceberg tables.

Precautions

- If a sort key is specified, data within files will be ordered after rewriting, but global ordering is not guaranteed.
- When defining rewrite conditions, you are advised to use partition-based criteria.

Syntax

```
OPTIMIZE table_name REWRITE DATA
   [ WITH OPTIONS (option_key = option_value [, ...]) ]
   [ WHERE condition ]
   [ ORDER BY target_list [ ASC | DESC ] ]
```

Parameter Description

- **table_name**: name of the target table. Range: an existing table name.
- **condition**: an expression that returns a boolean value, used to determine which files need to be rewritten.
- **target_list**: sort keys for data rewriting.
- **option_key**, **option_value**: configuration options for data rewriting.

Table 4-1 option_key, option_value descriptions

Parameter	Value	Description	Default Value
target-file-size-bytes	0–Max(uint64)	Size of rewritten files.	Matches **write.target-file-size-bytes**.
min-file-size-bytes	0–Max(uint64)	Original files smaller than this threshold are targeted.	75% of target-file-size-bytes
max-file-size-bytes	0–Max(uint64)	Original files larger than this threshold are targeted.	180% of target-file-size-bytes
min-input-files	0–Max(uint32)	Rewriting occurs if a partition has more target files than this value.	5

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Value	Description	Default Value
delete-file-threshold	0–Max(uint32)	Rewriting occurs if a partition has more delete files than this value.	INT_MAX
rewrite-all	**true**, **false**	Ignores all other conditions and rewrites all matching files.	false
max-file-group-size-bytes	0–Max(uint64)	Splits target files into multiple tasks if their total size exceeds this.	100 GB
output-spec-id	0~Max(int32)	Partition key definition ID used during data rewriting.	If not specified, the latest table definition's partition key ID is used.
use-starting-sequence-number	**true**, **false**	Uses the sequence number from the snapshot taken at the start of the data rewrite process.	true
remove_dangling_deletes	**true**, **false**	Deletes dangling delete files after rewriting. A delete file is considered "dangling" if it does not apply to any active data files. Enabling this feature generates an additional commit.	false

Examples

Create the **reason_t2** table.

```
CREATE TABLE reason_t2
(
   TABLE_SK         INTEGER             ,
   TABLE_ID         VARCHAR(20)         ,
   TABLE_NA          VARCHAR(20)
) PARTITION BY (TABLE_SC VARCHAR(20)) STORE AS iceberg;

INSERT INTO reason_t2 VALUES (1, 'S01', 'StudentA', 'Xiangbei'),(2, 'T01', 'TeacherA', 'Lingnan'),(3, 'T02',
'TeacherB', 'Hainan');
INSERT INTO reason_t2 VALUES (4, 'S02', 'StudentX', 'Xiangbei'),(5, 'T03', 'TeacherX', 'Lingnan'),(6, 'T04',
'TeacherY', 'Hainan');
```

Rewrite data in the TABLE_SC = 'Hainan' partition.

```
OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('rewrite-all' = 'true') WHERE TABLE_SC = 'Hainan';
```

Run the **ORDER BY** statement to view the data rewriting plan of the TABLE_SC = 'Lingnan' partition.

```
EXPLAIN VERBOSE OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('rewrite-all' = 'true') WHERE
TABLE_SC = 'Lingnan' ORDER BY 1;
```

 Copyright © Huawei Technologies Co., Ltd.

```
                                                    QUERY EXEC
INFO
-------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------
 id |                          operation                          | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Row Adapter                                             |   1000 |            |     178 | 73.68
  2 |   ->  Vector Insert on reason_t2                            |   1000 |            |     178 | 73.68
  3 |     ->  Vector Sort                                         |   1000 |            |     178 | 53.68
  4 |       ->  Partitioned Vector Foreign Scan on reason_t2      |   1000 |            |     178 | 1.35
                                        Predicate Information (identified by plan
id)

-------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------
  4 --Partitioned Vector Foreign Scan on reason_t2
      Server Type: lf
      Pruning results: (Manifests total: 3, Manifests left: 2, Files total: 4, Files dynamic prune 0, Files
dynamic RF prune: 0, Files left after pruning: 2)
      DN read from: direct
      replication: 2 files 0.00 MB
```

Configure **min-file-size-bytes** to rewrite all target files.

```
EXPLAIN VERBOSE OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('min-file-size-bytes' = '0')  ORDER
BY 1;
```

Here is an example of specifying the partition to write by setting **output-spec-id**:

```
ALTER TABLE reason_t2 ADD PARTITION COLUMN bucket(5, table_id) ;
select * from iceberg_partition_specs('reason_t2');
 spec_id | field_id |       name        | transform | source_id
---------+----------+-------------------+-----------+-----------
       0 |     1000 | table_sc          | identity  |         4
       1 |     1000 | table_sc          | identity  |         4
       1 |     1001 | table_id_bucket_5 | bucket[5] |         2
(3 rows)

OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('output-spec-id'='0')  ORDER BY 1;
```

4.8 UPDATE

Function

Updates data in a table. The **UPDATE** statement modifies specified column values in all rows that meet a given condition. The **WHERE** clause defines the condition, and fields listed in the **SET** clause are updated. Columns not included retain their original values.

NOTE

Currently, only Iceberg tables support this **UPDATE** operation.

Precautions

- To update data in a table, you must have **UPDATE** permissions on it. Additionally, you need **SELECT** permissions for any tables referenced in expressions or conditions within the query.
- Attempting to update the same row multiple times within a single SQL statement will result in an error.

 Copyright © Huawei Technologies Co., Ltd.

WARNING

Do not concurrently update the same partition of a table using **UPDATE**.

Syntax

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ * ] [ [ AS ] alias ] [FOR VERSION AS OF <branch_name>]
 SET {column_name = { expression | DEFAULT }
    |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] ) |sub_query }}[, ...]
    [ FROM from_list] [ WHERE condition ];

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

- **plan_hint**: specified as **/*+ */**, which provides hints for optimizing the execution plan of the statement block. For details, see **13.3.9 Hint-based Tuning**.
- **table_name**: name of the table to be updated, optionally qualified with its schema.
- **alias**: alias of the target table.
- FOR VERSION AS OF <branch_name>: branch of the target table.
- **column_name**: name of the column to modify.
  - You can reference the column using the table name followed by the column name. For example:
    ```
    UPDATE foo SET foo.col_name = 'a'
    ```
  - Alternatively, use the table's alias followed by the column name. For example:
    ```
    UPDATE foo AS f SET f.col_name = 'a'
    ```
- **expression**: A value or expression assigned to the column.
- **DEFAULT**: Populates the column with its default value. If no default value is available, the value is **NULL**.
- **sub_query**: A subquery used to update a table based on information from other tables within the same database. For details on the **SELECT** clause, refer to **5.1.1 Basic Syntax**.
- **from_list**: A list of table expressions allowing the use of columns from other tables in the **WHERE** condition. This is similar to declaring a table list in the **FROM** clause of a **SELECT** statement.

  Note: The target table must never appear in the **from_list**, except when performing a self-join (in which case it must appear under an alias).
- **condition**: A boolean expression determining which rows are updated. Only rows where the expression evaluates to **true** are updated.

Examples

Create the **reason_update** table.

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE TABLE reason_update
(
   TABLE_SK          INTEGER              ,
   TABLE_ID          VARCHAR(20)          ,
   TABLE_NA          VARCHAR(20)
) store as iceberg;
```

Insert data to the **reason_update** table.

```
INSERT INTO reason_update VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

Perform the UPDATE operation on the **reason_update** table.

```
UPDATE reason_update SET TABLE_NA = 'TeacherD' where TABLE_SK = 3;
```

 Copyright © Huawei Technologies Co., Ltd.

5 DQL Syntax

5.1 SELECT

5.1.1 Basic Syntax

Function

SELECT reads data from a table.

Acting like a filter layered over a database table, **SELECT** uses SQL keywords to extract the data needed from the data table.

Precautions

You must have query permissions on every column used in the **SELECT** command.

Syntax

```
[ WITH  with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] [time_travel_clause]]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ QUALIFY condition ]
[ { 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 ];
```

 Copyright © Huawei Technologies Co., Ltd.

NOTE

Aliases of expressions in **targetlist** can be used in conditions and expressions.

- Only referencing within the same level is allowed.
- Only aliases in **targetlist** can be referenced.
- Only subsequent expressions can reference preceding ones.
- Cannot contain **volatile** functions.
- Cannot include **Window** functions.
- Aliases cannot be referenced in **JOIN ON** conditions.
- An error will occur if there are multiple aliases to apply in **targetlist**.

- Subquery **with_query** is as follows:

```
with_query_name [ ( column_name [, ...] ) ]
    AS [ [ NOT ] MATERIALIZED ] ( {select | values | insert | update | delete} )
```

- Specified query **source from_item** is as follows:

```
{[ ONLY ] table_name | view_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 [, ...] ) ]}
```

- The **time_travel_clause** is as follows:

```
[FOR TIMESTAMP AS OF timestamp_expression] | [FOR VERSION AS OF snapshot_id]
```

- The **group** clause is as follows:

```
( )
| expression
| ( expression [, ...] )
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )
| GROUPING SETS ( grouping_element [, ...] )
```

- Specified partition **partition_clause** is as follows:

```
PARTITION { ( partition_name ) |
      FOR (  partition_value [, ...] ) }
```

NOTE

Specifying partitions is only applicable to regular tables.

- The sorting order **nlssort_expression_clause** is as follows:

```
NLSSORT ( column_name, ' NLS_SORT = { SCHINESE_PINYIN_M | generic_m_ci } ' )
```

- Simplified query syntax, functionally equivalent to **select * from table_name**.

```
TABLE { ONLY {(table_name)| table_name} | table_name [ * ]};
```

Parameter Description

- **WITH with_query [, ...]**

  Declares one or more subqueries that can be referenced by name in the main query, acting as temporary tables.

  Detailed format of **with_query** is: **with_query_name [(column_name [, ...])] AS [[NOT] MATERIALIZED] ({select | values | insert | update | delete})**.

  – **with_query_name** specifies the name of the result set generated by the subquery, which can be used to access the subquery's result set in the query.

  – By default, a **with_query** referenced multiple times by the main query is usually executed only once, and its result set is materialized for repeated

 Copyright © Huawei Technologies Co., Ltd.

queries by the main query. A **with_query** referenced once by the main query is not executed separately but replaces its subquery directly at the reference point in the main query, executing along with the main query. Explicitly specifying **[NOT] MATERIALIZED** can alter this default behavior:

- When **MATERIALIZED** is specified, the subquery is executed once, and its result set is materialized.
- When **NOT MATERIALIZED** is specified, the subquery is replaced at the reference point in the main query. **NOT MATERIALIZED** is ignored under the following circumstances:
  - The subquery contains **volatile** functions.
  - The subquery is an **INSERT**, **UPDATE**, or **DELETE** statement.
  - A **with_query2** referenced more than once refers to a self-referential **with_query1** from an outer layer, then **with_query2** cannot be replaced at the reference point.

    For example, in the following sample, **tmp2** is referenced twice. Since **tmp2** references the self-referential **tmp1** from an outer layer, **tmp2** is materialized even if **NOT MATERIALIZED** is specified.

```
with recursive tmp1(b) as (values(1)
union all
(with tmp2 as not materialized (select * from tmp1)
 select tt1.b + tt2.b from tmp2 tt1, tmp2 tt2))
 select * from tmp1;
```

– **column_name** specifies the column names displayed in the subquery's result set.

– Each subquery can be a **SELECT**, **VALUES**, **INSERT**, **UPDATE**, or **DELETE** statement.

- **plan_hint clause**

  It follows the format **/*+ */** and is placed immediately after the **SELECT** keyword. It provides hints for optimizing the execution plan of the **SELECT** statement block. For detailed usage instructions, refer to **13.3.9.1 Plan Hint Optimization**.

- **ALL**

  Declares that all rows meeting the conditions are returned. This is the default behavior, and the keyword can be omitted.

- **DISTINCT [ON (expression [, ...])]**

  Removes all duplicate rows from the result set of **SELECT**, ensuring each row in the result set is unique.

  ON (expression [, ...]) retains only the first row in the set of rows that yield the same result when evaluated against the given expressions.

  NOTE

  The **DISTINCT ON** expression is interpreted using the same rules as **ORDER BY**. Unless **ORDER BY** is used to ensure that the required row appears first, the first row is unpredictable.

- **SELECT list**

  Specifies the column names in the table to be queried, which can be partial columns or all columns (using the wildcard *).

 Copyright © Huawei Technologies Co., Ltd.

An alias can be assigned to an output column using the **AS output_name** clause, typically used for displaying the output column.

Column names can be expressed in the following forms:

- Manual input of column names, separating multiple columns with commas (,).
- Can be columns computed within the **FROM** clause.

- **FROM clause**

  Declares one or more source tables for **SELECT**.

  Elements involved in the **FROM** clause include:

  - table_name

    Table or view name, optionally prefixed with a schema name, for example, **schema_name.table_name**.

  - alias

    Assigns a temporary table alias to a table or complex table reference for subsequent query references.

    Aliases serve for abbreviation or disambiguation in self-joins. Providing an alias entirely conceals the actual table name.

  - column_alias

    Column alias.

  - PARTITION

    Queries data from a specific partition of a partitioned table.

  - partition_name

    Partition name.

  - partition_value

    Specifies the partition key value. When creating a partitioned table with multiple partition keys, the **PARTITION** clause can specify these key values to uniquely identify a partition.

  - subquery

    Subqueries may appear in the **FROM** clause, creating a temporary table to hold their outputs.

  - with_query_name

    WITH clauses can also serve as sources for the **FROM** clause, referenced by their **WITH** query names.

  - function_name

    Function name. Function calls can also appear in the **FROM** clause.

  - join_type

    The options are:

    - [INNER] JOIN

      Combines two **FROM** items using a **JOIN** clause. Parentheses can dictate nesting order. Absent them, **JOIN** nests left-to-right.

      Regardless, **JOIN** binds tighter than comma-separated **FROM** items.

    - LEFT [OUTER] JOIN

 Copyright © Huawei Technologies Co., Ltd.

Returns all rows satisfying the join condition from the Cartesian product, plus those rows from the left table that fail to match any row in the right table under the join condition. Left rows extend to the full width of the resulting table by padding NULLs in corresponding right table column positions. Note that the JOIN clause's condition applies only during matching computation. Outer conditions apply post-computation.

- RIGHT [OUTER] JOIN

  Yields all results of the inner join, augmented by each unmatched right row (left-padded with NULLs).

  Symbolic convenience since conversion to **LEFT OUTER JOIN** is possible by swapping left and right inputs.

- FULL [OUTER] JOIN

  Delivers all inner join results, supplemented by every unmatched left row (right-padded with NULLs) and every unmatched right row (left-padded with NULLs).

- CROSS JOIN

  Equivalent to **INNER JOIN ON (TRUE)**, which means no rows are removed by qualification. Such joins offer mere syntactic convenience, mirroring effects achievable via straightforward **FROM** and **WHERE** constructs.

  NOTE

  A join condition must be specified for the **INNER** and **OUTER** join types, namely one of **NATURAL ON**, **join_condition**, or **USING (join_column [, ...])**. However, they cannot appear in **CROSS JOIN**.

Among them, **CROSS JOIN** and **INNER JOIN** generate a simple Cartesian product, which is the same as the result of listing two items at the top level of **FROM**.

– ON join_condition

  Specifies the join condition, used to define which rows are matched in a join. Example: ON left_table.a = right_table.a

– USING(join_column[, ...])

  Specifies the abbreviation of **ON left_table.a = right_table.a AND left_table.b = right_table.b ...**. Corresponding columns must share the same name.

– NATURAL

  An abbreviation for a **USING** list containing all columns with identical names across the two tables.

– from item

  Name of the query source object used for joining.

– FOR TIMESTAMP AS OF timestamp_expression

  Time travel query, which queries the data of a specified object at a certain time point. **timestamp_expression** can be a specific time or a time expression. The value of the expression must be able to be calculated at the beginning of the query. **timestamp_expression** does not support subqueries, correlated references, and UDFs.

 Copyright © Huawei Technologies Co., Ltd.

- FOR VERSION AS OF <branch_name>|<tag_name>

  Retrieves data from a specified branch or tag.

- FOR VERSION AS OF snapshot_id

  Time travel query, which specifies the historical version of the object to be queried. **snapshot_id** indicates the historical version number. **snapshot_id** does not support subqueries, correlated references, and UDFs.

  NOTE

  Time travel query requires that the query object supports access to historical versions. Currently, Iceberg tables support performing Time travel queries.

  Example:

```
CREATE TABLE tt_iceberg (c1 int, c2 int) store AS iceberg;

INSERT INTO tt_iceberg values(1, 1),(1, 1),(1, 1);
INSERT INTO tt_iceberg values(2, 2),(2, 2);

-- Query the total number of current data rows.
SELECT count(1) FROM tt_iceberg FOR TIMESTAMP AS OF now();
 count
-------
     5
(1 row)

-- Query the total number of data rows 5 seconds ago.
SELECT count(1) FROM tt_iceberg FOR TIMESTAMP AS OF now() - interval '5s';
 count
-------
     5
(1 row)

-- Use iceberg_show_snapshots to query version information of the Iceberg table.
SELECT * FROM iceberg_show_snapshots('tt_iceberg'::regclass);
 sequence_number |     snapshot_id     |        create_time         |      parent_id      | operation
|                                                  manifestlist

|
summary

-----------------+---------------------+----------------------------+---------------------+-----------
+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------
+-----------------------------------------------------------------------------------------------------------------------
-----------------------------------
-------------------------------------------------------------------------------
               2 | 1539231080188128795 | 2025-07-11 10:54:51.319+00 | 3230380547366427570 |
append    | obs://mini-kernel/mini_kernel/llt/test/rhel_10_90_45_60/hudi_test_insert_p/tt_iceberg/
metadata/snap-1
539231080188128795-1-a0bab368-1cb3-4bb7-b24f-e05fef56617b.avro | {added-data-files=1,
added-records=2, added-files-size=700, changed-partition-count=1, total-records=5, total-files-
size=1400, total-data-files=2
, total-delete-files=0, total-position-deletes=0, total-equality-deletes=0}
               1 | 3230380547366427570 | 2025-07-11 10:54:39.264+00 |                   0 | append    |
obs://mini-kernel/mini_kernel/llt/test/rhel_10_90_45_60/hudi_test_insert_p/tt_iceberg/metadata/
snap-3
230380547366427570-1-c88145ca-e845-46b0-af91-bac79e5fef07.avro | {added-data-files=1,
added-records=3, added-files-size=700, changed-partition-count=1, total-records=3, total-files-
size=700, total-data-files=1,
 total-delete-files=0, total-position-deletes=0, total-equality-deletes=0}
(2 rows)

-- Query the total number of data rows of a specified version.
SELECT count(1) FROM tt_iceberg FOR VERSION AS OF '3230380547366427570';
 count
```

 Copyright © Huawei Technologies Co., Ltd.

```
-------
      3
(1 row)

-- Query the total number of data rows at the execution timestamp.
SELECT count(1) FROM tt_iceberg FOR TIMESTAMP AS OF '2025-07-11 10:54:39.264+00';
 count
-------
      3
(1 row)
```

- **WHERE clause**

  The **WHERE** clause forms a row selection expression to narrow the scope of the **SELECT** query. **condition** is any expression that returns a Boolean value. Rows that do not satisfy this condition will not be retrieved.

  NOTE

  For the **LIKE** operator in the **WHERE** clause, if special characters %, _, and \ need to be queried in **LIKE**, use backslashes (\) for escape.

  Example:

```
CREATE TABLE tt01 (id int,content varchar(50)) store AS orc;

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 ORDER BY id;
 id |      content
----+-------------------
  1 | Jack say 'hello'
  2 | Rose do 50%
  3 | Lilei say 'world'
  4 | Hanmei do 100%
(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)
```

- **GROUP BY clause**

  Groups query results by one or more columns. The results with the same value form a group.

  – ROLLUP ({ expression | (expression [, ...]) } [, ...])

    ROLLUP is used to calculate the standard aggregate value specified in **GROUP BY** for an ordered group column, create a high-level partial sum from right to left, and finally create a cumulative sum. A group can be considered as a set of groups. For example:

```
GROUP BY ROLLUP (a,b,c)
```

    Equivalent to:

```
GROUP BY GROUPING SETS((a,b,c), (a,b), (a), ( ))
```

    The elements in a **ROLLUP** clause can be separate columns or expressions, or lists enclosed in parentheses. If the lists are in

 Copyright © Huawei Technologies Co., Ltd.

parentheses, they must be considered as a whole when a grouping set is generated. For example:

```
GROUP BY ROLLUP ((a,b), (c,d))
```

Equivalent to:

```
GROUPING SETS ((a,b,c,d), (a,b), (c,d ), ( ))
```

– CUBE ({ expression | (expression [, ...]) } [, ...])

CUBE automatically groups and summarizes the fields listed in the **GROUP BY** clause. The result set contains all possible combinations of values in the dimension column and the aggregation values in the basic rows that match the combinations of these dimension values. It returns a row of summary information for each group. You can use **CUBE** to generate crosstab values. For example, if three expressions (n = 3) are provided in the **CUBE** clause, the operation result is a $2^n = 2^3 = 8$ groups. The rows grouped by the values of *n* expressions are called regular rows, and the remaining rows are called super-aggregated rows. For example:

```
GROUP BY CUBE (a,b,c)
```

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 separate columns or expressions, or lists enclosed in parentheses. If the lists are in parentheses, they must be considered as a whole when a grouping set is generated. For example:

```
GROUP BY CUBE (a, (b, c), d)
```

Equivalent to:

```
GROUP BY GROUPING SETS ((a,b,c,d), (a,b,c), (a,d), (b,c,d), (a), (b,c), (d), ())
```

– GROUPING SETS (grouping_element [, ...])

The **GROUPING SETS** clause is an extension of the **GROUP BY** clause. It allows you to specify multiple **GROUP BY** options. The option is used to define a group set. Each group 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 data groups that are not required. You can specify a data group for query as needed.

NOTE

If the expression of the **SELECT** list references ungrouped columns, an error is reported unless an aggregate function is used because multiple values may be returned for ungrouped columns.

- **HAVING clause**

  Used together with the **GROUP BY** clause to select specific groups. The **HAVING** clause compares some attributes of a group with a constant value. Only groups that satisfy the logical expression in the **HAVING** clause are extracted.

- **WINDOW clause**

  Typically formatted as **WINDOW window_name AS (window_definition) [, ...]**, where *window_name* is the name that can be referenced by subsequent window definitions. *window_definition* can take the following forms:

  [existing_window_name]

  [PARTITION BY expression [, ...]]

  [ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]

 Copyright © Huawei Technologies Co., Ltd.

[frame_clause]

frame_clause defines a window frame for the window function. Not all window functions depend on this frame, which consists of a set of related rows relative to the current query row. **frame_clause** can be structured as follows:

[RANGE | ROWS] frame_start

[RANGE | ROWS] BETWEEN frame_start AND frame_end

Options for **frame_start** and **frame_end**:

UNBOUNDED PRECEDING

value PRECEDING (unsupported by **RANGE**)

CURRENT ROW

value FOLLOWING (unsupported by **RANGE**)

UNBOUNDED FOLLOWING

- **QUALIFY clause**
  - Description: The **QUALIFY** clause filters result sets based on window function (WINDOW) expressions, similar to how the **HAVING** clause filters grouped results. It executes after window functions are computed, allowing you to apply filtering conditions using functions like **ROW_NUMBER()**, **RANK()**, or **SUM() OVER()**. Only rows where the **QUALIFY** expression evaluates to **true** are included in the final output.
  - Constraints: The **QUALIFY** clause must include at least one window function. For non-window function filtering conditions, use the **WHERE** clause instead.

  NOTICE

  The **QUALIFY** clause runs after window function calculations but before **DISTINCT**, **ORDER BY**, and **LIMIT** operations. The complete logical execution order is as follows: **FROM** → **WHERE** → **GROUP BY** → **HAVING** → **WINDOW** → **QUALIFY** → **DISTINCT** → **ORDER BY** → **LIMIT**. Ensure that **QUALIFY** safely references window function expressions such as **ROW_NUMBER()** or **SUM() OVER()**.

- **UNION clause**

  UNION computes the union of the sets of rows returned by multiple **SELECT** statements.

  Constraints for the **UNION** clause:

  - By default, **UNION** excludes duplicate rows unless the **ALL** clause is explicitly stated.
  - Multiple **UNION** operators within the same **SELECT** statement are evaluated from left to right unless overridden by parentheses.

  General syntax:

  select_statement UNION [ALL] select_statement

  - Here, **select_statement** can be any **SELECT** statement without **ORDER BY** or **LIMIT** clauses.

 Copyright © Huawei Technologies Co., Ltd.

  - If enclosed in parentheses, **ORDER BY** and **LIMIT** can be applied to sub-expressions.

- **INTERSECT clause**

  INTERSECT computes the intersection of the sets of rows returned by multiple **SELECT** statements, excluding duplicate records.

  Constraints for the **INTERSECT** clause:

  - Multiple **INTERSECT** operators within the same **SELECT** statement are evaluated from left to right unless overridden by parentheses.
  - When performing both **UNION** and **INTERSECT** operations on the results of multiple **SELECT** statements, **INTERSECT** operations are prioritized.

  General syntax:

  select_statement INTERSECT select_statement

- **EXCEPT clause**

  The **EXCEPT** clause has the following general form:

  select_statement EXCEPT [ALL] select_statement

  The **EXCEPT** operator computes the rows that exist in the output of the left **SELECT** statement but not in the output of the right **SELECT** statement.

  The result of **EXCEPT** does not contain any duplicate rows unless the **ALL** option is specified. When using **ALL**, a row that has *m* duplicates in the left table and *n* duplicates in the right table will appear max(m–n, 0) times in the result. Unless specified otherwise with parentheses, multiple **EXCEPT** operators within the same **SELECT** statement are evaluated from left to right. **EXCEPT** binds at the same level as **UNION**.

- **MINUS clause**

  Has the same functionality and usage as the **EXCEPT** clause.

- **ORDER BY clause**

  Sorts the data retrieved by the **SELECT** statement in ascending or descending order. For cases where the **ORDER BY** expression includes multiple columns:

  - Sorting is performed first based on the leftmost column. If the values in this column are identical, the next expression is used for comparison, and so forth.
  - If all declared expressions are identical, the results are returned in a random order.
  - Columns being sorted in **ORDER BY** must be included among the columns of the result set retrieved by the **SELECT** statement.

  NOTE

  - If **ORDER BY** is not specified, the data is returned in the order most quickly generated by the database system.
  - You may choose to append the keywords **ASC** (for ascending order) or **DESC** (for descending order) after any expression in the **ORDER BY** clause. If unspecified, **ASC** is used by default.
  - To support Chinese Pinyin sorting and case-insensitive sorting, you need to specify the encoding format as **UTF-8** or **GBK** during database initialization. The commands are as follows:

    initdb –E UTF8 –D ../data –locale=zh_CN.UTF-8 or **initdb –E GBK –D ../data –locale=zh_CN.GBK**

 Copyright © Huawei Technologies Co., Ltd.

- **[{ [LIMIT { count | ALL }] [OFFSET start [ROW | ROWS]] } | { LIMIT start, { count | ALL } }]**

  The **LIMIT** clause comprises two independent **LIMIT** clauses, an **OFFSET** clause, and a multi-parameter **LIMIT** clause:

  LIMIT { count | ALL }

  OFFSET start [ROW | ROWS]

  LIMIT start, { count | ALL }

  Here, **count** declares the maximum number of rows to return, while **start** indicates how many rows to skip before beginning to return rows. If both parameters are provided, **start** rows are skipped prior to returning up to **count** rows. A multi-parameter **LIMIT** clause cannot coexist with either a single-parameter **LIMIT** clause or an **OFFSET** clause.

- **FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY**

  If **count** is not specified, its default value is **1**. The **FETCH** clause restricts the total number of rows returned, starting from the first row of the query result.

- **PARTITION clause**

  Retrieves data from the corresponding partition in a partitioned table.

Examples

Query all **r_reason_sk** records in the **reason** table and remove duplicates.

```
SELECT DISTINCT(r_reason_sk) FROM reason;
```

Example of a **LIMIT** clause: Retrieve one record from the table.

```
SELECT * FROM reason LIMIT 1;
```

Example of a **LIMIT** clause: Retrieve the third record from the table.

```
SELECT * FROM reason LIMIT 1 OFFSET 2;
```

Example of a **LIMIT** clause: Retrieve the first two records from the table.

```
SELECT * FROM reason LIMIT 2;
```

Query all records and sort them in ascending alphabetical order.

```
SELECT r_reason_desc FROM reason ORDER BY r_reason_desc;
```

Example of a **GROUP BY** clause: Filter data based on query conditions and group the results.

```
SELECT r_reason_id, AVG(r_reason_sk) FROM reason GROUP BY r_reason_id HAVING AVG(r_reason_sk) > 25;
```

Example of a **GROUP BY** clause: Use a group by alias to group the results.

```
SELECT r_reason_id AS id FROM reason GROUP BY id;
```

Example of a **GROUP BY CUBE** clause: Filter data based on query conditions and summarize the grouped results.

```
SELECT r_reason_id,AVG(r_reason_sk) FROM reason GROUP BY CUBE(r_reason_id,r_reason_sk);
```

Example of a **GROUP BY GROUPING SETS** clause: Filter data based on query conditions and summarize the grouped results.

```
SELECT r_reason_id,AVG(r_reason_sk) FROM reason GROUP BY GROUPING
SETS((r_reason_id,r_reason_sk),r_reason_sk);
```

 Copyright © Huawei Technologies Co., Ltd.

Example of a **UNION** clause: Merge contents starting with W and N in the **r_reason_desc** column in the **reason** table.

```
SELECT r_reason_sk, reason.r_reason_desc
   FROM reason
   WHERE reason.r_reason_desc LIKE 'W%'
UNION
SELECT r_reason_sk, reason.r_reason_desc
   FROM reason
   WHERE reason.r_reason_desc LIKE 'N%';
```

Example of a **GROUP BY** clause: Group records in the **reason_p** table by **r_reason_id** and count the number of records in each group.

```
SELECT COUNT(*),r_reason_id FROM reason_p GROUP BY r_reason_id;
 count |   r_reason_id
-------+------------------
     2 | AAAAAAAACAAAAAAA
     5 | AAAAAAAABAAAAAAA
(2 rows)
```

Example of a **GROUP BY CUBE** clause: Filter data based on query conditions and summarize the grouped query results.

```
SELECT * FROM reason GROUP BY CUBE (r_reason_id,r_reason_sk,r_reason_desc);
```

Example of a **GROUP BY GROUPING SETS** clause: Filter data based on query conditions and summarize the grouped query results.

```
SELECT * FROM reason GROUP BY GROUPING SETS ((r_reason_id,r_reason_sk),r_reason_desc);
```

Example of a **HAVING** clause: Group records in the **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 reason_p GROUP BY r_reason_id HAVING c>2;
 c |   r_reason_id
---+------------------
 5 | AAAAAAAABAAAAAAA
(1 row)
```

Example of an **IN** clause: Group records in the **reason_p** table by **r_reason_id**, count the number of records in each group, and display only counts where **r_reason_id** is either **AAAAAAAABAAAAAAA** or **AAAAAAAADAAAAAAA**.

```
SELECT COUNT(*),r_reason_id FROM reason_p GROUP BY r_reason_id HAVING r_reason_id
IN('AAAAAAAABAAAAAAA','AAAAAAAADAAAAAAA');
count |   r_reason_id
-------+------------------
     5 | AAAAAAAABAAAAAAA
(1 row)
```

Example of an **INTERSECT** clause: Query information where **r_reason_id** equals **AAAAAAAABAAAAAAA** and **r_reason_sk** is smaller than **5**.

```
SELECT * FROM reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' INTERSECT SELECT * FROM reason_p
WHERE r_reason_sk<5;
   r_reason_id    |                                        r_reason_desc                                                
```

```
SELECT * FROM reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' EXCEPT SELECT * FROM reason_p
WHERE r_reason_sk<4;
  r_reason_id    |                                  r_reason_desc                                  | r_reason_sk
-----------------+-------------------------------------------------------------------------------------------------+-------------
 AAAAAAAABAAAAAAA | reason 5                                                                        |          10
 AAAAAAAABAAAAAAA | reason 3                                                                        |           4
 AAAAAAAABAAAAAAA | reason 2                                                                        |          10
 AAAAAAAABAAAAAAA | reason 4                                                                        |          10
(4 rows)
```

5.1.2 Table Query Examples

5.1.2.1 Single Table Query

Scenario

This section uses an example of a product inventory table to describe the basic usage of single table queries, including simple queries, conditional filtering, sorting, limiting the number of results, and aggregate queries.

Example Table Description

This section uses the product inventory table newproducts as an example. The table structure is as follows:

Table 5-1 Example table structure

Field	Type	Description
product_id	INTEGER	Product ID.
product_name	VARCHAR2(60)	Product name.
category	VARCHAR2(60)	Product category.
quantity	INTEGER	Inventory quantity.

Creating a Sample Table and Inserting Data

Run the following statements to create a sample table and insert data:

```
CREATE TABLE newproducts
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60),
quantity  INTEGER
)
STORE AS orc;


INSERT INTO newproducts VALUES (1502, 'earphones', 'electronics',150);
INSERT INTO newproducts VALUES (1601, 'telescope', 'toys',80);
INSERT INTO newproducts VALUES (1666, 'Frisbee', 'toys',244);
INSERT INTO newproducts VALUES (1700, 'interface', 'books',100);
INSERT INTO newproducts VALUES (2344, 'milklotion', 'skin care',320);
```

 Copyright © Huawei Technologies Co., Ltd.

```
INSERT INTO newproducts VALUES (3577, 'dumbbell', 'sports',550);
INSERT INTO newproducts VALUES (1210, 'necklace', 'jewels', 200);
```

Simple Query

Run the **SELECT... FROM...** statement to obtain results from the database.

```
SELECT category FROM newproducts;
  category
------------
 electronics
 toys
 sports
 jewels
 books
 skin care
 toys
(7 rows)
```

Filtering the Results

Run the **WHERE** statement to filter the query result and find the queried part.

```
SELECT * FROM newproducts WHERE category='toys';
 product_id | product_name | category | quantity
------------+--------------+----------+----------
       1601 | telescope    | toys     |       80
       1666 | Frisbee      | toys     |      244
(2 rows)
```

Sorting the Results

Run the **ORDER BY** statement to sort query results as desired.

```
SELECT product_id,product_name,category,quantity FROM newproducts ORDER BY quantity DESC;
 product_id | product_name |  category   | quantity
------------+--------------+-------------+----------
       3577 | dumbbell     | sports      |      550
       2344 | milklotion   | skin care   |      320
       1666 | Frisbee      | toys        |      244
       1210 | necklace     | jewels      |      200
       1502 | earphones    | electronics |      150
       1700 | interface    | books       |      100
       1601 | telescope    | toys        |       80
(7 rows)
```

Limiting the Number of Query Results

If you want the query to return only part of the result, you can use the **LIMIT** statement to limit the number of returned records.

```
SELECT product_id,product_name,category,quantity FROM newproducts ORDER BY quantity DESC limit 5;
 product_id | product_name |  category   | quantity
------------+--------------+-------------+----------
       3577 | dumbbell     | sports      |      550
       2344 | milklotion   | skin care   |      320
       1666 | Frisbee      | toys        |      244
       1210 | necklace     | jewels      |      200
       1502 | earphones    | electronics |      150
(5 rows)
```

 Copyright © Huawei Technologies Co., Ltd.

Aggregate Query

If you want to query data comprehensively, you can use the **GROUP BY** statement and aggregate functions to construct an aggregated query.

```
SELECT category, string_agg(quantity,',') FROM newproducts GROUP BY category;
  category   | string_agg
-------------+------------
 books       | 100
 electronics | 150
 skin care   | 320
 jewels      | 200
 toys        | 80,244
 sports      | 550
(6 rows)
```

5.1.2.2 Multi-Table Join Queries

What Is a Multi-Table Join Query?

In actual business scenarios, data is typically distributed across multiple tables. For example, student information is stored in the **student** table, and scores are stored in the **score** table. When you need to query the scores of each student, you must join the two tables. This is a multi-table join query.

A multi-table join query combines data from multiple tables based on a common join key (such as student ID) to return complete information across tables.

This section uses the student table and math score table as examples to introduce the basic usage of multi-table join queries, including:

- Inner join: inner join, cross join, and natural join.
- Outer join: left outer join, right outer join, and full join.
- The difference between the ON condition and the WHERE condition.

Example Table Description

This section uses two example tables.

- **student table** (student information table):

  Table 5-2 student table structure

Field	Type	Description
id	INTEGER	Student ID.
name	VARCHAR(50)	Student name.

- **math_score table** (math score table):

 Copyright © Huawei Technologies Co., Ltd.

Table 5-3 math_score table structure

Field	Type	Description
id	INTEGER	Student ID (associated with the student table).
score	INTEGER	Math score.

Creating a Table and Inserting Data

Run the following statements to create the sample tables **student** and **math_score** and insert data into them.

```
CREATE TABLE student(
  id INTEGER,
  name varchar(50)
)
STORE AS orc;

CREATE TABLE math_score(
  id INTEGER,
  score INTEGER
)
STORE AS orc;

INSERT INTO student VALUES(1, 'Tom');
INSERT INTO student VALUES(2, 'Lily');
INSERT INTO student VALUES(3, 'Tina');
INSERT INTO student VALUES(4, 'Perry');

INSERT INTO math_score VALUES(1, 80);
INSERT INTO math_score VALUES(2, 75);
INSERT INTO math_score VALUES(4, 95);
INSERT INTO math_score VALUES(6, NULL);
```

Inner Join

- Inner join

  Syntax:

  ```
  left_table [INNER] JOIN right_table [ ON join_condition | USING ( join_column )]
  ```

  Description: Rows that meet **join_condition** in both the left and right tables are joined and output. Tuples that do not meet **join_condition** are not output.

  Example: Query students' math scores.

  ```
  SELECT s.id, s.name, ms.score FROM student s JOIN math_score ms on s.id = ms.id;
   id | name  | score
  ----+-------+-------
    2 | Lily  |    75
    1 | Tom   |    80
    4 | Perry |    95
  (3 rows)


  EXPLAIN SELECT s.id, s.name, ms.score FROM student s JOIN math_score ms on s.id = ms.id;
                                      QUERY PLAN
  ------------------------------------------------------------------------------------------------
   id |                    operation                    | E-rows | E-memory | E-width | E-costs
  ----+-------------------------------------------------+--------+----------+---------+---------
    1 | ->  Row Adapter                                 |   5000 |          |     126 | 435.27
    2 |   ->  Vector Streaming (type: GATHER)           |   5000 |          |     126 | 435.27
    3 |     ->  Vector Sonic Hash Join (4,6)            |   5000 |          |     126 | 200.89
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
 4 |         ->  Vector Streaming(type: BROADCAST)    |   2000 |          |     122 | 104.64
 5 |           ->  Vector Foreign Scan on student s  |   1000 |          |     122 | 60.00
 6 |         ->  Vector Foreign Scan on math_score ms |   1000 |          |       8 | 60.00

Predicate Information (identified by plan id)
-----------------------------------------------
 3 --Vector Sonic Hash Join (4,6)
       Hash Cond: (s.id = ms.id)
 5 --Vector Foreign Scan on student s
       Server Type: lf
       Total files left: 4
 6 --Vector Foreign Scan on math_score ms
       Server Type: lf
       Total files left: 4

             ====== Query Summary =====
-------------------------------------------------------------
System available mem: 1992294KB
Query Max mem: 2097152KB
Query estimated mem: 2097152KB
LakeFormation request time: 290.913 ms, request count: 5
Total billed bytes: 0 bytes
Turbo Engine: true
(28 rows)
```

- Cross join

  Syntax:

  ```
  left_table CROSS JOIN right_table
  ```

  Description: Each row in the left table is joined with each row in the right table. The number of final rows is the product of the number of rows on both sides. The product is also called Cartesian product.

  Example: Cross join of the **student** table and the **math_score** table.

  ```
  SELECT s.id, s.name, ms.score FROM student s CROSS JOIN math_score ms order by id;
   id | name  | score
  ----+-------+-------
    1 | Tom   |    80
    1 | Tom   |    95
    1 | Tom   |    75
    1 | Tom   |
    2 | Lily  |    80
    2 | Lily  |    95
    2 | Lily  |    75
    2 | Lily  |
    3 | Tina  |    80
    3 | Tina  |    95
    3 | Tina  |    75
    3 | Tina  |
    4 | Perry |    80
    4 | Perry |    95
    4 | Perry |    75
    4 | Perry |
  (16 rows)

  EXPLAIN SELECT s.id, s.name, ms.score FROM student s CROSS JOIN math_score ms order by id;
                                        QUERY PLAN
  ------------------------------------------------------------------------------------------------------------------------
   id |                       operation                       | E-rows  |   E-memory    | E-width | E-costs
  ----+-------------------------------------------------------+---------+---------------+---------+----------
    1 | ->  Row Adapter                                       | 1000000 |               |     126 | 55463.56
    2 |   ->  Vector Streaming (type: GATHER)                 | 1000000 |               |     126 | 55463.56
    3 |     ->  Vector Sort                                   | 1000000 | 518MB(2043MB) |     126 | 54994.81
    4 |       ->  Vector Nest Loop (5,7)                      | 1000000 | 1MB           |     126 | 6415.89
    5 |         ->  Vector Streaming(type: BROADCAST)         |    2000 | 2MB           |     122 | 104.64
    6 |           ->  Vector Foreign Scan on student s        |    1000 | 1MB           |     122 | 60.00
    7 |         ->  Vector Materialize                        |    1000 | 16MB          |       4 | 62.50
    8 |           ->  Vector Foreign Scan on math_score ms    |    1000 | 1MB           |       4 | 60.00
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
Predicate Information (identified by plan id)
-----------------------------------------------
  6 --Vector Foreign Scan on student s
        Server Type: lf
        Total files left: 4
  8 --Vector Foreign Scan on math_score ms
        Server Type: lf
        Total files left: 4

              ====== Query Summary =====
-----------------------------------------------------------
System available mem: 1992294KB
Query Max mem: 2097152KB
Query estimated mem: 2097152KB
LakeFormation request time: 210.501 ms, request count: 5
Total billed bytes: 0 bytes
Turbo Engine: true
(28 rows)
```

- Natural join

  Syntax:

  ```
  left_table NATURAL JOIN right_table
  ```

  Description: Columns with the same name in left table and right table are joined by equi-join, and the columns with the same name are merged into one column.

  Example: Natural join of the **student** table and the **math_scores** table, where the columns with the same name in both tables are the **id** columns, will be joined on equal values of the **id** column.

  ```
  SELECT * FROM student s NATURAL JOIN math_score ms order by id;
   id | name  | score
  ----+-------+-------
    1 | Tom   |    80
    2 | Lily  |    75
    4 | Perry |    95
  (3 rows)

  EXPLAIN SELECT * FROM student s NATURAL JOIN math_score ms order by id;
                                        QUERY PLAN
  ------------------------------------------------------------------------------------------------------------
   id |                    operation                     | E-rows | E-memory | E-width | E-costs
  ----+--------------------------------------------------+--------+----------+---------+---------
    1 | ->  Row Adapter                                  |   5000 |          |     126 | 582.61
    2 |   ->  Vector Streaming (type: GATHER)            |   5000 |          |     126 | 582.61
    3 |     ->  Vector Sort                              |   5000 |          |     126 | 348.24
    4 |       ->  Vector Sonic Hash Join (5,7)           |   5000 |          |     126 | 200.89
    5 |         ->  Vector Streaming(type: BROADCAST)    |   2000 |          |     122 | 104.64
    6 |           ->  Vector Foreign Scan on student s   |   1000 |          |     122 | 60.00
    7 |         ->  Vector Foreign Scan on math_score ms |   1000 |          |       8 | 60.00

   Predicate Information (identified by plan id)
  -----------------------------------------------
    4 --Vector Sonic Hash Join (5,7)
          Hash Cond: (s.id = ms.id)
    6 --Vector Foreign Scan on student s
          Server Type: lf
          Total files left: 4
    7 --Vector Foreign Scan on math_score ms
          Server Type: lf
          Total files left: 4

                ====== Query Summary =====
  -----------------------------------------------------------
  System available mem: 1992294KB
  Query Max mem: 2097152KB
  Query estimated mem: 2097152KB
  LakeFormation request time: 377.624 ms, request count: 5
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
Total billed bytes: 0 bytes
Turbo Engine: true
(29 rows)
```

Outer Join

- Left Join

  Syntax:

  ```
  left_table LEFT [OUTER] JOIN right_table [ ON join_condition | USING ( join_column )]
  ```

  Description: The result set of a left outer join includes all rows of left_table, not only the joined rows. If a row in the left table does not match any row in the right table, the row will be **NULL** in the result set.

  Example: A left outer join is performed between the **student** table and the **math_scores** table. In the result set, for the row in the **student** table where the **id** is **3**, the corresponding data from the right table will be filled with **NULL**.

  ```
  SELECT s.id, s.name, ms.score FROM student s LEFT JOIN math_score ms on (s.id = ms.id) order by id;
   id | name  | score
  ----+-------+-------
    1 | Tom   |    80
    2 | Lily  |    75
    3 | Tina  |
    4 | Perry |    95
  (4 rows)

  EXPLAIN SELECT s.id, s.name, ms.score FROM student s LEFT JOIN math_score ms on (s.id = ms.id)
  order by id;
                                        QUERY PLAN
  ------------------------------------------------------------------------------------------------------------
    id |                       operation                       | E-rows | E-memory | E-width | E-costs
   ----+-------------------------------------------------------+--------+----------+---------+---------
    1 | ->  Row Adapter                                       |   5000 |          |     126 | 582.61
    2 |   ->  Vector Streaming (type: GATHER)                 |   5000 |          |     126 | 582.61
    3 |     ->  Vector Sort                                   |   5000 |          |     126 | 348.24
    4 |       ->  Vector Sonic Hash Right Join (5, 7)         |   5000 |          |     126 | 200.89
    5 |         ->  Vector Streaming(type: BROADCAST)         |   2000 |          |       8 | 104.64
    6 |           ->  Vector Foreign Scan on math_score ms    |   1000 |          |       8 | 60.00
    7 |         ->  Vector Foreign Scan on student s          |   1000 |          |     122 | 60.00

   Predicate Information (identified by plan id)
   -----------------------------------------------
    4 --Vector Sonic Hash Right Join (5, 7)
          Hash Cond: (ms.id = s.id)
    6 --Vector Foreign Scan on math_score ms
          Server Type: lf
          Total files left: 4
    7 --Vector Foreign Scan on student s
          Server Type: lf
          Total files left: 4

                ====== Query Summary =====
   ---------------------------------------------------------
   System available mem: 1992294KB
   Query Max mem: 2097152KB
   Query estimated mem: 2097152KB
   LakeFormation request time: 1253.487 ms, request count: 5
   Total billed bytes: 0 bytes
   Turbo Engine: true
   (29 rows)
  ```

- Right join

  Syntax:

  ```
  left_table RIGHT [OUTER] JOIN right_table [ ON join_condition | USING ( join_column )]
  ```

 Copyright © Huawei Technologies Co., Ltd.

Description: Contrary to left outer join, the result set of right outer join includes all rows of the right table, not only the rows that match the join column. If a row in the right table does not match any row in the left table, all columns in the left table in the associated result set row are null.

Example: A right outer join is performed between the **student** table and the **math_scores** table. In the result set, for the row in the **math_scores** table where the **id** is **6**, the corresponding data from the left table will be filled with **NULL**.

```
SELECT ms.id, s.name, ms.score FROM student s RIGHT JOIN math_score ms on (s.id = ms.id) order by id;
 id | name  | score
----+-------+-------
  1 | Tom   |    80
  2 | Lily  |    75
  4 | Perry |    95
  6 |       |
(4 rows)

EXPLAIN SELECT ms.id, s.name, ms.score FROM student s RIGHT JOIN math_score ms on (s.id = ms.id) order by id;
                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 id |                    operation                    | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                 |   5000 |          |     126 | 582.61
  2 |   ->  Vector Streaming (type: GATHER)           |   5000 |          |     126 | 582.61
  3 |     ->  Vector Sort                             |   5000 |          |     126 | 348.24
  4 |       ->  Vector Sonic Hash Right Join (5, 7)   |   5000 |          |     126 | 200.89
  5 |         ->  Vector Streaming(type: BROADCAST)   |   2000 |          |     122 | 104.64
  6 |           ->  Vector Foreign Scan on student s  |   1000 |          |     122 | 60.00
  7 |         ->  Vector Foreign Scan on math_score ms|   1000 |          |       8 | 60.00

 Predicate Information (identified by plan id)
 --------------------------------------------
   4 --Vector Sonic Hash Right Join (5, 7)
         Hash Cond: (s.id = ms.id)
   6 --Vector Foreign Scan on student s
         Server Type: lf
         Total files left: 4
   7 --Vector Foreign Scan on math_score ms
         Server Type: lf
         Total files left: 4

               ====== Query Summary =====
 ---------------------------------------------------------
 System available mem: 1992294KB
 Query Max mem: 2097152KB
 Query estimated mem: 2097152KB
 LakeFormation request time: 214.389 ms, request count: 5
 Total billed bytes: 0 bytes
 Turbo Engine: true
(29 rows)
```

In a right join, **Left** is displayed in the join operator. This is because a right join is actually the process replacing the left table with the right table then performing left join.

- Full join

  Syntax:

  ```
  left_table FULL [OUTER] JOIN right_table [ ON join_condition | USING ( join_column )]
  ```

  Description: A full join is a combination of a left outer join and a right outer join. The result set of a full outer join includes all rows of the left table and the right table, not just the joined rows. If a row in the left table does not match any row in the right table, all columns in the right table in the

 Copyright © Huawei Technologies Co., Ltd.

associated result set row are null. If a row in the right table does not match any row in the left table, all columns in the left table in the associated result set row are null.

Example: A full outer join is performed between the **student** table and the **math_scores** table. In the result set, for the row in the **student** table where the **id** is **3**, the corresponding data from the right table will be filled with **NULL**. For the row in the **math_scores** table where the **id** is **6**, the corresponding data from the left table will be filled with **NULL**.

```
SELECT s.id, s.name, ms.id, ms.score FROM student s FULL JOIN math_score ms ON (s.id = ms.id)
order by s.id;
 id | name  | id | score
----+-------+----+-------
  1 | Tom   |  1 |    80
  2 | Lily  |  2 |    75
  3 | Tina  |    |
  4 | Perry |  4 |    95
    |       |  6 |
(5 rows)

EXPLAIN SELECT s.id, s.name, ms.id, ms.score FROM student s FULL JOIN math_score ms ON (s.id =
ms.id) order by s.id;
                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 id |                       operation                       | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                       |   5000 |          |     130 | 734.79
  2 |   ->  Vector Streaming (type: GATHER)                 |   5000 |          |     130 | 734.79
  3 |     ->  Vector Sort                                   |   5000 |          |     130 | 434.86
  4 |       ->  Vector Sonic Hash Full Join (5, 7)          |   5000 |          |     130 | 243.95
  5 |         ->  Vector Streaming(type: REDISTRIBUTE)      |   1000 |          |     122 | 103.57
  6 |           ->  Vector Foreign Scan on student s        |   1000 |          |     122 | 60.00
  7 |         ->  Vector Streaming(type: REDISTRIBUTE)      |   1000 |          |       8 | 103.57
  8 |           ->  Vector Foreign Scan on math_score ms    |   1000 |          |       8 | 60.00

Predicate Information (identified by plan id)
-----------------------------------------------
  4 --Vector Sonic Hash Full Join (5, 7)
        Hash Cond: (s.id = ms.id)
  6 --Vector Foreign Scan on student s
        Server Type: lf
        Total files left: 4
  8 --Vector Foreign Scan on math_score ms
        Server Type: lf
        Total files left: 4

              ====== Query Summary =====
-------------------------------------------------------------
System available mem: 1992294KB
Query Max mem: 2097152KB
Query estimated mem: 2097152KB
LakeFormation request time: 206.619 ms, request count: 5
Total billed bytes: 0 bytes
Turbo Engine: true
(30 rows)
```

Differences Between the ON Condition and the WHERE Condition in Multi-Table Query

According to the preceding join syntax, except natural join and cross join, the **ON** condition (**USING** is converted to the **ON** condition during query parsing) is used on the join result of both the two tables. Generally, the **WHERE** condition is used in the query statement to restrict the query result. The **ON** join condition and

 Copyright © Huawei Technologies Co., Ltd.

WHERE filter condition do not contain conditions that can be pushed down to tables. The differences between **ON** and **WHERE** are as follows:

- The **ON** condition is used for joining two tables.
- **WHERE** is used to filter the result set.

To sum up, the **ON** condition is used when two tables are joined. After the join result set of two tables is generated, the **WHERE** condition is used.

5.2 WITH Expression

The WITH expression is used to define auxiliary statements used in large queries. These auxiliary statements are usually called common table expressions (CTEs), which can be understood as a named subquery. The subquery can be referenced multiple times by its name in the query.

The auxiliary statement in the **WITH** expression can be **SELECT**, and the **WITH** clause itself can be attached to a main statement.

SELECT in WITH

This section describes the usage of **SELECT** in a **WITH** clause.

Syntax

```
[WITH  with_query [,…] ] SELECT …
```

The syntax of **with_query** is as follows:

```
with_query_name [ ( column_name [, ...] ) ]
    AS [ [ NOT ] MATERIALIZED ] ( {select} )
```

 Copyright © Huawei Technologies Co., Ltd.

CAUTION

- If you use **MATERIALIZED**, the subquery runs once and its result set is saved. If you use **NOT MATERIALIZED**, the subquery is replaced with its reference in the main query.
- The SQL statement specified by the **AS** statement of each CTE must be a statement that can return query results. Currently, only the **SELECT** query statement is supported. Other data modification statements such as **INSERT**, **UPDATE**, **DELETE**, and **VALUES** are not yet supported.
- A **WITH** expression indicates the CTE definition in a SQL statement block. Multiple CTEs can be defined at the same time. You can specify column names for each CTE or use the aliases of the columns in the query output. Example:

  ```
  WITH s1(a, b) AS (SELECT x, y FROM t1), s2 AS (SELECT x, y FROM t2) SELECT * FROM s1 JOIN s2 ON s1.a=s2.x;
  ```

  This statement defines two CTEs: **s1** and **s2**. **s1** specifies the column names **a** and **b**, and **s2** does not specify the column names. Therefore, the column names are the output column names **x** and **y**.
- Each CTE can be referenced zero, one, or more times in the main query.
- CTEs with the same name cannot exist in the same statement block. If CTEs with the same name exist in different statement blocks, the CTE in the nearest statement block is referenced.
- An SQL statement may contain multiple SQL statement blocks. Each statement block can contain a **WITH** expression. The CTE in each **WITH** expression can be referenced in the current statement block, subsequent CTEs of the current statement block, and sub-layer statement blocks, however, it cannot be referenced in the parent statement block. The definition of each CTE is also a statement block. Therefore, a WITH expression can also be defined in the statement block.

The fundamental value of SELECT in WITH is to break down complex queries into simpler parts. An example is as follows: Example:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

The **WITH** clause defines two auxiliary statements: **regional_sales** and **top_regions**. The output of **regional_sales** is used in **top_regions**, and the output of **top_regions** is used in the main **SELECT** query. This example can be written without **WITH**. In that case, it must be written with a two-layer nested sub-SELECT statement, making the query longer and difficult to maintain.

 Copyright © Huawei Technologies Co., Ltd.

6 DCL Syntax

6.1 ANALYZE | ANALYSE

Function

Collects statistics about table contents in databases. The statistical results are stored in LakeFormation. The execution plan generator uses this statistical data to determine the most efficient execution plan.

Users capable of executing **ANALYZE** on a specific table include the table's owner and users granted read permissions on the table.

Precautions

ANALYZE scans data in the table and generates billing records.

Syntax

Collect statistics for the entire table.

```
{ ANALYZE | ANALYSE } table_name;
```

Collect statistics about a specific partition in a partitioned table.

```
{ ANALYZE | ANALYSE } table_name (predicate_condition);
```

Parameter Description

- **table_name**

  Name of the specific table to be analyzed (may include schema qualification).

  Range: an existing table name.

- **predicate_condition**

  Partitioning predicate condition.

  Range: correct partition definitions and corresponding partition values.

Examples

Update statistics for the **customer_info** table using **ANALYZE**:

 Copyright © Huawei Technologies Co., Ltd.

```
ANALYZE customer_info;
```

Update statistics for a specific partition in the **partition_customer_info** table:

```
CREATE TABLE partition_customer_info (a char(3),c date, d text) partition by (b bigint, month(c),
bucket(3,d)) store as iceberg;
-- Assume partitions exist: b=1, c_month=2024-05, d_bucket=5
ANALYZE partition_customer_info (b=1,month(c)='2024-05',bucket(3,d)=5);

CREATE TABLE partition_customer_info (a char(3),c date) partition by (b bigint, e text) store as parquet;
-- Assume partitions exist: b=1, e='abcd'
ANALYZE partition_customer_info (b=1,e='abcd');
-- Assume partitions exist: b=2, e=null
ANALYZE partition_customer_info (b=1,e is NULL);
```

6.2 GRANT

Function

GRANT is used to grant permissions in LakeFormation.

Syntax

```
GRANT privileges [ WITH GRANT ] ON privilege_target TO grantee_list;
```

WITH GRANT indicates cascading authorization, which means that the authorized user can grant the same permissions to other users.

Specifically:

- privileges can be:
  ```
  | [ READ, WRITE, DESCRIBE, EXECUTE, ALTER, DROP, INSERT, SELECT, UPDATE, DELETE ] [, ...]
  | ALL
  | ALL PRIVILEGE
  | ALL PRIVILEGES
  ```
- privilege_target can be:
  ```
  | DATABASE name_list
  | SCHEMA name_list
  | TABLE qualified_name_list
  | FUNCTION function_with_argtypes_list
  | PATH obs_path_list
  | ALL TABLES IN SCHEMA name_list
  | ALL FUNCTIONS IN SCHEMA name_list
  ```

  Specifically:

  - name_list is as follows:
    ```
    | name [, ...]
    ```
  - qualified_name_list is as follows:
    ```
    | qualified_name [, ...]
    ```
  - function_with_argtypes_list is as follows:
    ```
    | function_with_argtypes [, ...]
    ```
  - obs_path_list is as follows:
    ```
    | SCONST [, ...]
    ```
- grantee_list can be:
  ```
  | USER RoleId [, ...]
  | USER GROUP RoleId [, ...]
  | ROLE RoleId [, ...]
  ```

 Copyright © Huawei Technologies Co., Ltd.

6.3 REVOKE

Function

REVOKE is used to revoke permissions in LakeFormation.

Syntax

```
REVOKE privileges [ WITH GRANT ] ON privilege_target FROM grantee_list;
```

WITH GRANT indicates cascading authorization, which means that the authorized user can grant the same permissions to other users.

Specifically:

- privileges can be:
  ```
  | [ READ, WRITE, DESCRIBE, EXECUTE, ALTER, DROP, INSERT, SELECT, UPDATE, DELETE ] [, ...]
  | ALL
  | ALL PRIVILEGE
  | ALL PRIVILEGES
  ```
- privilege_target can be:
  ```
  | DATABASE name_list
  | SCHEMA name_list
  | TABLE qualified_name_list
  | FUNCTION function_with_argtypes_list
  | PATH obs_path_list
  | ALL TABLES IN SCHEMA name_list
  | ALL FUNCTIONS IN SCHEMA name_list
  ```
  Specifically:
  - name_list is as follows:
    ```
    | name [, ...]
    ```
  - qualified_name_list is as follows:
    ```
    | qualified_name [, ...]
    ```
  - function_with_argtypes_list is as follows:
    ```
    | function_with_argtypes [, ...]
    ```
  - obs_path_list is as follows:
    ```
    | SCONST [, ...]
    ```
- grantee_list can be:
  ```
  | USER RoleId [, ...]
  | USER GROUP RoleId [, ...]
  | ROLE RoleId [, ...]
  ```

 Copyright © Huawei Technologies Co., Ltd.

7 Keywords

SQL contains both reserved and non-reserved words. According to the standard, reserved words must not be used as other identifiers. Non-reserved words have special meanings only in specific contexts and can be used as identifiers in other contexts.

Table 7-1 SQL keywords

Keyword	Aura	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
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

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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	-	-
AUTHID	Reserved	-	-
AUTHINFO	Non-reserved	-	-
AUTHORIZATION	Reserved (can be a function or type)	Reserved	Reserved
AUTOEXTEND	Non-reserved	-	-
AUTOMAPPED	Non-reserved	-	-
AVG	-	Non-reserved	Reserved
BACKWARD	Non-reserved	-	-
BARRIER	Non-reserved	-	-
BEFORE	Non-reserved	Reserved	-
BEGIN	Non-reserved	Reserved	Reserved
BETWEEN	Non-reserved (cannot be a function or type)	Non-reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
BIGINT	Non-reserved (cannot be a function or type)	-	-
BINARY	Reserved (can be a function or type)	Reserved	-
BINARY_DOUBLE	Non-reserved (cannot be a function or type)	-	-
BINARY_INTEGER	Non-reserved (cannot be a function or type)	-	-
BIT	Non-reserved (cannot be a function or type)	Reserved	Reserved
BITVAR	-	Non-reserved	-
BIT_LENGTH	-	Non-reserved	Reserved
BLOB	Non-reserved	Reserved	-
BOOLEAN	Non-reserved (cannot be a function or type)	Reserved	-
BOTH	Reserved	Reserved	Reserved
BUCKETS	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
CASCADED	Non-reserved	Reserved	Reserved
CASE	Reserved	Reserved	Reserved
CAST	Reserved	Reserved	Reserved
CATALOG	Non-reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
CATALOG_NAME	-	Non-reserved	Non-reserved
CHAIN	Non-reserved	Non-reserved	-
CHAR	Non-reserved (cannot be a function or type)	Reserved	Reserved
CHARACTER	Non-reserved (cannot be a function or type)	Reserved	Reserved
CHARACTERISTICS	Non-reserved	-	-
CHARACTER_LENGTH	-	Non-reserved	Reserved
CHARACTER_SET_CATALOG	-	Non-reserved	Non-reserved
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 (cannot be a function or type)	Non-reserved	Reserved
COBOL	-	Non-reserved	Non-reserved
COLLATE	Reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
COLLATION	Reserved (can be a function or type)	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
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
COMPATIBLE_ILLEGAL_CHARS	Non-reserved	-	-
COMPLETE	Non-reserved	-	-
COMPRESS	Non-reserved	-	-
COMPLETION	-	Reserved	-
CONCURRENTLY	Reserved (can be a function or type)	-	-
CONDITION	-	-	-
CONDITION_NUMBER	-	Non-reserved	Non-reserved
CONFIGURATION	Non-reserved	-	-
CONNECT	-	Reserved	Reserved
CONNECTION	Non-reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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
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 (can be a function or type)	Reserved	Reserved
CSV	Non-reserved	-	-
CUBE	-	Reserved	-
CURRENT	Non-reserved	Reserved	Reserved
CURRENT_CATALOG	Reserved	-	-
CURRENT_DATE	Reserved	Reserved	Reserved
CURRENT_PATH	-	Reserved	-
CURRENT_ROLE	Reserved	Reserved	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
CURRENT_SCHEMA	Reserved (can be a function or type)	-	-
CURRENT_TIME	Reserved	Reserved	Reserved
CURRENT_TIMESTAMP	Reserved	Reserved	Reserved
CURRENT_USER	Reserved	Reserved	Reserved
CURSOR	Non-reserved	Reserved	Reserved
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	-	-
DATE	Non-reserved (cannot be a function or type)	Reserved	Reserved
DATETIME_INTERVAL_CODE	-	Non-reserved	Non-reserved
DATETIME_INTERVAL_PRECISION	-	Non-reserved	Non-reserved
DAY	Non-reserved	Reserved	Reserved
DBCOMPATIBILITY	Non-reserved	-	-
DEALLOCATE	Non-reserved	Reserved	Reserved
DEC	Non-reserved (cannot be a function or type)	Reserved	Reserved
DECIMAL	Non-reserved (cannot be a function or type)	Reserved	Reserved
DECLARE	Non-reserved	Reserved	Reserved
DECODE	Non-reserved (cannot be a function or type)	-	-
DEFAULT	Reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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	-	-
DELIMITERS	Non-reserved	-	-
DELTA	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	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
DOMAIN	Non-reserved	Reserved	Reserved
DOUBLE	Non-reserved	Reserved	Reserved
DROP	Non-reserved	Reserved	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

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
EXECUTE	Non-reserved	Reserved	Reserved
EXISTING	-	Non-reserved	-
EXISTS	Non-reserved (cannot be a function or type)	Non-reserved	Reserved
EXPIRATION	Non-reserved	-	-
EXPLAIN	Non-reserved	-	-
EXTENSION	Non-reserved	-	-
EXTERNAL	Non-reserved	Reserved	Reserved
EXTRACT	Non-reserved (cannot be a function or type)	Non-reserved	Reserved
FALSE	Reserved	Reserved	Reserved
FAMILY	Non-reserved	-	-
FAST	Non-reserved	-	-
FENCED	Non-reserved	-	-
FETCH	Reserved	Reserved	Reserved
FILEHEADER	Non-reserved	-	-
FILL_MISSING_FIELDS	Non-reserved	-	-
FINAL	-	Non-reserved	-
FIRST	Non-reserved	Reserved	Reserved
FIXED	Non-reserved	Reserved	Reserved
FLOAT	Non-reserved (cannot be a function or type)	Reserved	Reserved
FOLLOWING	Non-reserved	-	-
FOR	Reserved	Reserved	Reserved
FORCE	Non-reserved	-	-
FOREIGN	Reserved	Reserved	Reserved
FORMATTER	Non-reserved	-	-
FORTRAN	-	Non-reserved	Non-reserved
FORWARD	Non-reserved	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
FOUND	-	Reserved	Reserved
FREE	-	Reserved	-
FREEZE	Reserved (can be a function or type)	-	-
FROM	Reserved	Reserved	Reserved
FULL	Reserved (can be a function or type)	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 (cannot be a function or type)	-	-
GROUP	Reserved	Reserved	Reserved
GROUPING	-	Reserved	-
HANDLER	Non-reserved	-	-
HAVING	Reserved	Reserved	Reserved
HEADER	Non-reserved	-	-
HIERARCHY	-	Non-reserved	-
HOLD	Non-reserved	Non-reserved	-
HOST	-	Reserved	-
HOUR	Non-reserved	Reserved	Reserved
IDENTIFIED	Non-reserved	-	-
IDENTITY	Non-reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
IF	Non-reserved (cannot be a function or type)	-	-
IFNULL	Non-reserved (cannot be a function or type)	-	-
IGNORE	-	Reserved	-
IGNORE_EXTRA_DATA	Non-reserved	-	-
ILIKE	Reserved (can be a function or type)	-	-
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	-
INHERIT	Non-reserved	-	-
INHERITS	Non-reserved	-	-
INITIAL	Non-reserved	-	-
INITIALIZE	-	Reserved	-
INITIALLY	Reserved	Reserved	Reserved
INITRANS	Non-reserved	-	-
INLINE	Non-reserved	-	-
INNER	Reserved (can be a function or type)	Reserved	Reserved
INOUT	Non-reserved (cannot be a function or type)	Reserved	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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 (cannot be a function or type)	Reserved	Reserved
INTEGER	Non-reserved (cannot be a function or type)	Reserved	Reserved
INTERNAL	Reserved	-	-
INTERSECT	Reserved	Reserved	Reserved
INTERVAL	Non-reserved (cannot be a function or type)	Reserved	Reserved
INTO	Reserved	Reserved	Reserved
INVOKER	Non-reserved	Non-reserved	-
IS	Reserved	Reserved	Reserved
ISNULL	Non-reserved (cannot be a function or type)	-	-
ISOLATION	Non-reserved	Reserved	Reserved
ITERATE	-	Reserved	-
JOIN	Reserved (can be a function or type)	Reserved	Reserved
K	-	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	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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 (cannot be a function or type)	-	-
LEFT	Reserved (can be a function or type)	Reserved	Reserved
LENGTH	-	Non-reserved	Non-reserved
LESS	Reserved	Reserved	-
LEVEL	Non-reserved	Reserved	Reserved
LIKE	Reserved (can be a function or type)	Reserved	Reserved
LIMIT	Reserved	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	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
MAP	-	Reserved	-
MAPPING	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	-	-
MERGE	Non-reserved	-	-
MESSAGE_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_OCTET_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_TEXT	-	Non-reserved	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	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
MOVEMENT	Non-reserved	-	-
MUMPS	-	Non-reserved	Non-reserved
NAME	Non-reserved	Non-reserved	Non-reserved
NAMES	Non-reserved	Reserved	Reserved
NATIONAL	Non-reserved (cannot be a function or type)	Reserved	Reserved
NATURAL	Reserved (can be a function or type)	Reserved	Reserved
NCHAR	Non-reserved (cannot be a function or type)	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 (cannot be a function or type)	Reserved	-
NOT	Reserved	Reserved	Reserved
NOTHING	Non-reserved	-	-
NOTIFY	Non-reserved	-	-
NOTNULL	Reserved (can be a function or type)	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
NOWAIT	Non-reserved	-	-
NULL	Reserved	Reserved	Reserved
NULLABLE	-	Non-reserved	Non-reserved
NULLIF	Non-reserved (cannot be a function or type)	Non-reserved	Reserved
NULLS	Non-reserved	-	-
NUMBER	Non-reserved (cannot be a function or type)	Non-reserved	Non-reserved
NUMERIC	Non-reserved (cannot be a function or type)	Reserved	Reserved
NUMSTR	Non-reserved	-	-
NVARCHAR2	Non-reserved (cannot be a function or type)	-	-
NVL	Non-reserved (cannot be a function or type)	-	-
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	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
OPTIMIZE	Non-reserved	-	-
OPTION	Non-reserved	Reserved	Reserved
OPTIONS	Non-reserved	Non-reserved	-
OR	Reserved	Reserved	Reserved
ORDER	Reserved	Reserved	Reserved
ORDINALITY	-	Reserved	-
OUT	Non-reserved (cannot be a function or type)	Reserved	-
OUTER	Reserved (can be a function or type)	Reserved	Reserved
OUTPUT	-	Reserved	Reserved
OVER	Non-reserved	-	-
OVERLAPS	Reserved (can be a function or type)	Non-reserved	Reserved
OVERLAY	Non-reserved (cannot be a function or type)	Non-reserved	-
OVERRIDING	-	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	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
PARAMETER_SPECIFIC_SCHEMA	-	Non-reserved	-
PARSER	Non-reserved	-	-
PARTIAL	Non-reserved	Reserved	Reserved
PARTITION	Non-reserved	-	-
PARTITIONS	Non-reserved	-	-
PASCAL	-	Non-reserved	Non-reserved
PASSING	Non-reserved	-	-
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 (cannot be a function or type)	Non-reserved	Reserved
POSTFIX	-	Reserved	-
PRECEDING	Non-reserved	-	-
PRECISION	Non-reserved (cannot be a function or type)	Reserved	Reserved
PREFERRED	Non-reserved	-	-
PREFIX	Non-reserved	Reserved	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
PREORDER	-	Reserved	-
PREPARE	Non-reserved	Reserved	Reserved
PREPARED	Non-reserved	-	-
PRESERVE	Non-reserved	Reserved	Reserved
PRIMARY	Reserved	Reserved	Reserved
PRIOR	Non-reserved	Reserved	Reserved
PRIVATE	Non-reserved	-	-
PRIVILEGE	Non-reserved	-	-
PRIVILEGES	Non-reserved	Reserved	Reserved
PROCEDURAL	Non-reserved	-	-
PROCEDURE	Reserved	Reserved	Reserved
PROFILE	Non-reserved	-	-
PUBLIC	-	Reserved	Reserved
QUERY	Non-reserved	-	-
QUOTE	Non-reserved	-	-
RANGE	Non-reserved	-	-
RAW	Non-reserved	-	-
READ	Non-reserved	Reserved	Reserved
READS	-	Reserved	-
REAL	Non-reserved (cannot be a function or type)	Reserved	Reserved
REASSIGN	Non-reserved	-	-
REBUILD	Non-reserved	-	-
RECHECK	Non-reserved	-	-
RECURSIVE	Non-reserved	Reserved	-
REF	Non-reserved	Reserved	-
REFRESH	Non-reserved	-	-
REFERENCES	Reserved	Reserved	Reserved
REFERENCING	-	Reserved	-
REINDEX	Non-reserved	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
REJECT	Reserved	-	-
RELATIVE	Non-reserved	Reserved	Reserved
RELEASE	Non-reserved	-	-
RELOPTIONS	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
RIGHT	Reserved (can be a function or type)	Reserved	Reserved
ROLE	Non-reserved	Reserved	-
ROLLBACK	Non-reserved	Reserved	Reserved
ROLLUP	-	Reserved	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
ROUTINE	-	Reserved	-
ROUTINE_CATALOG	-	Non-reserved	-
ROUTINE_NAME	-	Non-reserved	-
ROUTINE_SCHEMA	-	Non-reserved	-
ROW	Non-reserved (cannot be a function or type)	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	-	-
SERIALIZABLE	Non-reserved	Non-reserved	Non-reserved
SERVER	Non-reserved	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
SERVER_NAME	-	Non-reserved	Non-reserved
SESSION	Non-reserved	Reserved	Reserved
SESSION_USER	Reserved	Reserved	Reserved
SET	Non-reserved	Reserved	Reserved
SETOF	Non-reserved (cannot be a function or type)	-	-
SETS	-	Reserved	-
SHARE	Non-reserved	-	-
SHIPPABLE	Non-reserved	-	-
SHOW	Non-reserved	-	-
SIMILAR	Reserved (can be a function or type)	Non-reserved	-
SIMPLE	Non-reserved	Non-reserved	-
SIZE	Non-reserved	Reserved	Reserved
SMALLDATETIME	Non-reserved (cannot be a function or type)	-	-
SMALLDATETIME_FORMAT	Non-reserved	-	-
SMALLINT	Non-reserved (cannot be a function or type)	Reserved	Reserved
SNAPSHOT	Non-reserved	-	-
SOME	Reserved	Reserved	Reserved
SOURCE	Non-reserved	Non-reserved	-
SPACE	-	Reserved	Reserved
SPECIFIC	-	Reserved	-
SPECIFICTYPE	-	Reserved	-
SPECIFIC_NAME	-	Non-reserved	-
SPILL	Non-reserved	-	-
SPLIT	Non-reserved	-	-
SQL	-	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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	-
SUBSTRING	Non-reserved (cannot be a function or type)	Non-reserved	Reserved
SUM	-	Non-reserved	Reserved
SUPERUSER	Non-reserved	-	-
SYMMETRIC	Reserved	Non-reserved	-
SYNONYM	Non-reserved	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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	-	-
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 (cannot be a function or type)	Reserved	Reserved
TIME_FORMAT	Non-reserved	-	-
TIMESTAMP	Non-reserved (cannot be a function or type)	Reserved	Reserved
TIMESTAMPADD	Non-reserved (cannot be a function or type)	-	-
TIMESTAMPDIFF	Non-reserved (cannot be a function or type)	-	-
TIMESTAMP_FORMAT	Non-reserved	-	-
TIMEZONE_HOUR	-	Reserved	Reserved
TIMEZONE_MINUTE	-	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
TINYINT	Non-reserved (cannot be a function or type)	-	-
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 (cannot be a function or type)	Reserved	-
TRIGGER	Non-reserved	Reserved	-
TRIGGER_CATALOG	-	Non-reserved	-
TRIGGER_NAME	-	Non-reserved	-
TRIGGER_SCHEMA	-	Non-reserved	-
TRIM	Non-reserved (cannot be a function or type)	Non-reserved	Reserved
TRUE	Reserved	Reserved	Reserved
TRUNCATE	Non-reserved	-	-
TRUSTED	Non-reserved	-	-
TRY_CAST	Non-reserved	-	-
TSTAG	Reserved. This field is only used in the stream data warehouse.	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
TSTIME	Reserved. This field is only used in the stream data warehouse.	-	-
TSFIELD	Reserved. This field is only used in the stream 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	-
UNTIL	Non-reserved	-	-
UNUSABLE	Non-reserved	-	-
UPDATE	Non-reserved	Reserved	Reserved
UPPER	-	Non-reserved	Reserved
USAGE	-	Reserved	Reserved
USER	Reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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 (cannot be a function or type)	Reserved	Reserved
VARCHAR	Non-reserved (cannot be a function or type)	Reserved	Reserved
VARCHAR2	Non-reserved (cannot be a function or type)	-	-
VARIABLE	-	Reserved	-
VARIADIC	Reserved	-	-
VARYING	Non-reserved	Reserved	Reserved
VCGROUP	Non-reserved	-	-
VERBOSE	Reserved (can be a function or type)	-	-
VERIFY	Non-reserved	-	-
VERSION	Non-reserved	-	-
VIEW	Non-reserved	Reserved	Reserved
VOLATILE	Non-reserved	-	-
WHEN	Reserved	Reserved	Reserved
WHENEVER	-	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
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 (cannot be a function or type)	-	-
XMLCONCAT	Non-reserved (cannot be a function or type)	-	-
XMLELEMENT	Non-reserved (cannot be a function or type)	-	-
XMLEXISTS	Non-reserved (cannot be a function or type)	-	-
XMLFOREST	Non-reserved (cannot be a function or type)	-	-
XMLNAMESPACES	Non-reserved (cannot be a function or type)	-	-
XMLPARSE	Non-reserved (cannot be a function or type)	-	-
XMLPI	Non-reserved (cannot be a function or type)	-	-

 Copyright © Huawei Technologies Co., Ltd.

Keyword	Aura	SQL:1999	SQL:1992
XMLROOT	Non-reserved (cannot be a function or type)	-	-
XMLSERIALIZE	Non-reserved (cannot be a function or type)	-	-
XMLTABLE	Non-reserved (cannot be a function or type)	-	-
YEAR	Non-reserved	Reserved	Reserved
YES	Non-reserved	-	-
ZONE	Non-reserved	Reserved	Reserved

 Copyright © Huawei Technologies Co., Ltd.

8 Data Type

8.1 Mapping Between Aura and LakeFormation Data Types

There is a clear mapping between the data types defined in Aura and those in LakeFormation, as shown in the table below.

Table 8-1 Mapping between Aura and LakeFormation data types

Data Type	Aura Data Type	Description	LakeFormation Data Type	Remarks
Numeric type	SMALLINT	Small integer, also called INT2.	smallint	–32,768 to +32,767
	TINYINT	Small integer, also called SMALLINT.	tinyint	–32,768 to +32,767
	INTEGER	Typical choice for integer, also called INT4.	int	–2,147,483,648 to +2,147,483,647
	BIGINT	Big integer, also called INT8.	bigint	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	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].	decimal	1 ≤ p ≤ 38, 0 ≤ s ≤ p; If **p** is unspecified, an error prompts exceeding the precision range.

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Aura Data Type	Description	LakeFormation Data Type	Remarks
	REAL,FLOAT4	Single-precision floating-point number, imprecise.	float	6-digit decimal precision.
	DOUBLE PRECISION, FLOAT8	Double-precision floating-point number, imprecise.	double	15-digit decimal precision.
	FLOAT[(p)]	Floating-point number, imprecise. The value range of precision (p) is [1, 53]. If the precision is not specified, DOUBLE PRECISION is used by default.	float/ double	• If 1 ≤ p < 25, the value is stored as a float. • If 25 ≤ p ≤ 53, the value is stored as a double.
	DEC[(p[,s])]	The value range of p (precision) is [1, 1000], and the value range of s (standard) is [0, p].	decimal	If 1 ≤ p ≤ 38 and 0 ≤ s ≤ p, an error is reported when p is not specified, indicating that the precision exceeds the allowed range.
	INTEGER[(p[,s])]	The value range of p (precision) is [1, 1000], and the value range of s (standard) is [0, p].	decimal	If 1 ≤ p ≤ 38 and 0 ≤ s ≤ p, an error is reported when p is not specified, indicating that the precision exceeds the allowed range.
Boolean type	BOOLEAN	Boolean: **true**, **false**, or **null**	boolean	-

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Aura Data Type	Description	LakeFormation Data Type	Remarks
Character type	CHAR(n),CHARACTER(n),NCHAR(n)	Fixed-length string. **n** indicates the character length, which is padded with spaces if insufficient. **n** is less than or equal to 255.	char	1 ≤ n ≤ 255. If **n** is not specified, the default value is **1**.
	VARCHAR(n),CHARACTER VARYING(n)	Variable-length string. **n** indicates the character length, which is less than or equal to 65,535.	varchar	1 ≤ n ≤ 65535. If **n** is not specified, the default value **65535** is used.
	Text	Variable-length string.	String	-
Date/Time type	DATE	Record date.	date	If the data includes time information, it is truncated to store only the date data.
	TIMESTAMP[(p)] [WITHOUT TIME ZONE]	Date and time without time zone information. **p** indicates precision after the decimal point. The value ranges from 0 to 6.	timestamp	The actual storage precision is fixed at 6 and is not affected by the value of **p**. If **p** is greater than 6, the precision is decreased to 6.
	TIMESTAMP[(p)] [WITH TIME ZONE]	Date and time (with a time zone). TIMESTAMP WITH TIME ZONE is also known as TIMESTAMPTZ. **p** indicates precision after the decimal point. The value ranges from 0 to 6.	timestamp with time zone	The actual storage precision is fixed at 6 and is not affected by the value of **p**. If **p** is greater than 6, the precision is decreased to 6.

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Aura Data Type	Description	LakeFormation Data Type	Remarks
Binary type	BYTEA	Variable-length binary string.	binary	-
Complex type	ARRAY<...>	Array type, with subscripts starting at 1, supporting multiple dimensions and nesting.	array	-
	STRUCT<...>	STRUCT type, supporting nesting.	struct	-

8.2 Numeric Types

Numeric types consist of integers of 1, 2, 4, or 8 bytes, floating-point numbers of 4 or 8 bytes, and optionally precise decimals.

For details about numeric operators and functions, see **11.4 Numeric Operation Functions and Operators**.

Aura supports integers, arbitrary precisions, floating point types, and serial integers.

Integer Types

The SMALLINT, INTEGER, and BIGINT types store the entire value (without the decimal part), that is, an integer. 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 8-2 Integer types

Column	Description	Storage Space	Range
SMALLINT	Small integer, also called INT2	2 bytes	-32,768 ~ +32,767

Copyright © Huawei Technologies Co., Ltd.

Column	Description	Storage Space	Range
INTEGER	Typical choice for integer, also called INT4	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 data of the SMALLINT, INTEGER, and BIGINT types.

```
CREATE TABLE int_type_t1
(
    a SMALLINT,
    b SMALLINT,
    c INTEGER,
    d BIGINT
) STORE AS orc;
```

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

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 8-3 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

 Copyright © Huawei Technologies Co., Ltd.

Examples:

Create a table with DECIMAL values.

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

Insert data.

```
INSERT INTO decimal_type_t1 VALUES(123456.122331);
INSERT INTO decimal_type_t1 VALUES(123456.452399);
```

View data.

```
SELECT * FROM decimal_type_t1;
   dt_col1
-------------
 123456.1223
 123456.4524
(2 rows)
```

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 8-4 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.

 Copyright © Huawei Technologies Co., Ltd.

Column	Description	Storage Space	Range
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_t1
(
FT_COL1 REAL,
FT_COL2 FLOAT4,
FT_COL3 DOUBLE PRECISION,
FT_COL4 FLOAT8,
FT_COL5 FLOAT,
FT_COL6 FLOAT(3),
FT_COL7 DEC(10,4),
FT_COL8 INTEGER(6,3)
) STORE AS orc;
```

Insert data.

```
INSERT INTO float_type_t1 VALUES (10.01, 10.3655, 123456.1234, 10.3214, 123.1237, 123.1237, 123.124, 125.123456);
```

View data.

```
SELECT * FROM float_type_t1;

 ft_col1 | ft_col2 |   ft_col3   | ft_col4 | ft_col5  | ft_col6 | ft_col7  | ft_col8
---------+---------+-------------+---------+----------+---------+----------+---------
   10.01 | 10.3655 | 123456.1234 | 10.3214 | 123.1237 | 123.124 | 123.1240 | 125.123
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

8.3 Boolean Type

Table 8-5 Boolean type

Type	Description	Storage Space	Value
BOOLEAN	Boolean type	1 byte.	• **true**: True. • **false**: False. • **null**: Unknown.

- Valid text literals for true: **TRUE**, **'t'**, **'true'**, **'y'**, **'yes'**, and **'1'**.
- Valid text literals for false: **FALSE**, **'f'**, **'false'**, **'n'**, **'no'**, and **'0'**.

Using **TRUE** and **FALSE** is more standardized (and SQL-compliant).

Example

Display boolean values with the letters **t** and **f**.

```
-- Create a table.
CREATE TABLE bool_type_t1
(
BT_COL1 BOOLEAN,
BT_COL2 TEXT
) STORE AS orc;

-- 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)

-- Drop the table.
DROP TABLE bool_type_t1;
```

8.4 Character Types

Table 8-6 lists the character types supported by Aura. For details about string operators and related built-in functions, see **11.1 Character Processing Functions and Operators**.

 Copyright © Huawei Technologies Co., Ltd.

Table 8-6 Character types

Name	Description	Length	Storage Space
CHAR(n) CHARACTER(n) NCHAR(n)	Fixed-length string. If the length is not reached, fill in spaces.	**n** indicates the character 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.
Text	Variable-length string.	-	The maximum size is 1,073,733,621 bytes (1 GB – 8,203 bytes).

NOTE

- In addition to the size limit per column, the total size of each tuple must not exceed 1,073,733,621 bytes (1 GB – 8,203 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 string. In Aura, operations of the fixed-length character type will use extra storage and memory resources.

There are two other fixed-length character types in Aura. They are displayed in **Table 8-7**.

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 regular 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. To avoid truncation at 64 bytes, the name type must be explicitly cast 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 8-7 Special character types

Name	Description	Storage Space
name	Internal type for object names	64 bytes

 Copyright © Huawei Technologies Co., Ltd.

Name	Description	Storage Space
char	Single-byte internal type	1 byte

Length

If a field is defined as **char(n)** or **varchar(n)**. **n** indicates the maximum length. No matter which type is used, the maximum length that can be set must not exceed 10485760 bytes (that is, 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)) STORE AS orc;
   ```
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)
   ```
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, it indicates the maximum storage capacity. If no length is specified, the field accommodates data of any size.

Example:

1. Create table **t2** and specify the character type of its columns.
   ```
   CREATE TABLE t2 (a char(5),b varchar(5)) STORE AS orc;
   ```
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**.

 Copyright © Huawei Technologies Co., Ltd.

```
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 conversion using the function, the actual byte length of column **a** is **3**.

```
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)
```

Empty Strings and NULL

Example:

```
SELECT  '' is null , null is null;
 isnull |  isnull
--------+----------
 f      | t
(1 row)
```

8.5 Binary Type

Table 8-8 lists the binary data types supported by Aura.

Table 8-8 Binary type

Type	Description	Storage Space
BYTEA	Variable-length binary string.	4 bytes plus the actual binary string. The maximum size is 1,073,733,621 bytes (1 GB – 8,203 bytes).

NOTE

In addition to the size limit per column, the total size of each tuple must not exceed 1 GB – 8,203 bytes.

Examples

```
-- Create a table.
CREATE TABLE binary_type_t1
(
BT_COL1 INTEGER,
BT_COL2 BYTEA
) STORE AS orc;

-- Insert data.
INSERT INTO binary_type_t1 VALUES(10, E'\\xDEADBEEF');
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Query data from the table.
SELECT * FROM binary_type_t1;
 bt_col1 |  bt_col2
---------+------------
      10 | \xdeadbeef
(1 row)

-- Drop the table.
DROP TABLE binary_type_t1;
```

8.6 Date/Time Types

Table 8-9 lists the date/time types supported by Aura. For the operators and built-in functions for these types, see **11.5 Time and Date Processing Functions and Operators**.

NOTE

If the time format of another database is different from that of Aura, modify the value of the **DateStyle** parameter to keep them consistent.

Table 8-9 Date/Time types

Type	Description	Storage Space
DATE	Record date	4 bytes
TIMESTAMP[(p)] or TIMESTAMP WITHOUT TIME ZONE[(p)]	Date and time. **p** indicates precision after the decimal point. The value ranges from 0 to 6.	8 bytes
TIMESTAMPTZ[(p)] or TIMESTAMP WITH TIME ZONE[(p)]	Date and time (with a time zone). **p** indicates precision after the decimal point. The value ranges from 0 to 6.	8 bytes

Example:

```
-- Create a table.
CREATE TABLE date_type_t1(
DA_COL1 DATE
) STORE AS orc;

-- Insert data.
INSERT INTO date_type_t1 VALUES (date '12-10-2010');

-- View data.
SELECT * FROM date_type_t1;
  da_col1
------------
 2010-12-10
(1 row)

-- Drop the table.
DROP TABLE date_type_t1;

-- Create a table.
```

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE TABLE timestamp_type_t1
(
TS_COL1 TIMESTAMP,
TS_COL2 TIMESTAMP(1) WITHOUT TIME ZONE,
TS_COL3 TIMESTAMP(6) WITHOUT TIME ZONE
) STORE AS orc;
-- Insert data.
INSERT INTO timestamp_type_t1 VALUES ('2024-11-15 21:21:21.2565455', '2024-11-15 21:21:21.2565455',
'2024-11-15 21:21:21.2565455');
-- View data.
SELECT * FROM timestamp_type_t1;
          ts_col1           |          ts_col2           |          ts_col3
----------------------------+----------------------------+----------------------------
 2024-11-15 21:21:21.256545 | 2024-11-15 21:21:21.256545 | 2024-11-15 21:21:21.256545
(1 row)

-- Drop the table.
DROP TABLE timestamp_type_t1;
```

Date Input

Dates and times can be entered in nearly any reasonable format, including ISO-8601, SQL-compatible, traditional POSTGRES, or others. The system allows you to customize the order of day, month, and year for date inputs. Setting the **DateStyle** parameter to **MDY** parses dates as month-day-year, **DMY** as day-month-year, and **YMD** as year-month-day.

Textual date entries must be enclosed in single quotes, with the following syntax:

type [(p)] 'value'

Here, **p** in the optional precision declaration is an integer indicating the fractional seconds digits. **Table 8-10** illustrates the input methods for the date type.

Table 8-10 Date input methods

Example	Description
1999-01-08	ISO 8601 format (preferred), always interpreted as January 8, 1999.
January 8, 1999	Unambiguous under any datestyle input mode.
1/8/1999	Ambiguous. Interpreted as January 8 in **MDY** mode, August 1 in **DMY** mode.
1/18/1999	Interpreted as January 18 in **MDY** mode, rejected otherwise.
01/02/03	• January 2, 2003 in **MDY** mode. • February 1, 2003 in **DMY** mode. • February 3, 2001 in **YMD** mode.
1999-Jan-08	Always January 8 regardless of mode.
Jan-08-1999	Always January 8 regardless of mode.
08-Jan-1999	Always January 8 regardless of mode.
99-Jan-08	January 8 in **YMD** mode, else an error.

 Copyright © Huawei Technologies Co., Ltd.

Example	Description
08-Jan-99	January 8. It is incorrect in **YMD** mode.
Jan-08-99	January 8. It is incorrect in **YMD** mode.
19990108	ISO 8601. Always January 8, 1999.
990108	ISO 8601. Always January 8, 1999.
1999.008	Year and day within the year.
J2451187	Julian day.
January 8, 99 BC	99 BC.

Example:

```
-- Create a table.
CREATE TABLE date_style_t1(DAT_COL1 DATE) STORE AS orc;

-- Insert data.
INSERT INTO date_style_t1 VALUES(date '12-10-2010');

-- View data.
SELECT  * FROM date_style_t1;
      dat_col1
---------------------
 2010-12-10 00:00:00
(1 row)

-- View the date format.
SHOW datestyle;
 DateStyle
-----------
 ISO, MDY
(1 row)

-- Set the date format.
SET datestyle='YMD';
SET

-- Insert data.
INSERT INTO date_style_t1 VALUES(date '2010-12-11');

-- View data.
SELECT  * FROM date_style_t1;
      dat_col1
---------------------
 2010-12-10
 2010-12-11
(2 rows)

-- Drop the table.
DROP TABLE date_style_t1;
```

Time

The supported time types are timestamp [(p)] without time zone and timestamp [(p)] with time zone. If only timestamp is written, it is equivalent to **timestamp without time zone**.

 Copyright © Huawei Technologies Co., Ltd.

If a time zone is specified in the input of the **time without time zone** type, the time zone is ignored.

For details about time input types, refer to **Table 8-11**. For details about time zone input types, refer to **Table 8-12**.

Table 8-11 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 the value.
4:05 PM	Same as 16:05. The 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	Abbreviated time zone.

Table 8-12 Time zone input

Example	Description
PST	Abbreviation (for Pacific Standard Time).
-8:00	Offset between ISO 8601 and PST.
-800	Offset between ISO 8601 and PST.
-8	Offset between ISO 8601 and PST.

Special Values

The special values supported by Aura are converted to common date/time values when being read. For details, see **Table 8-13**.

 Copyright © Huawei Technologies Co., Ltd.

Table 8-13 Special values

Input String	Applicable Type	Description
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix epoch)
infinity	timestamp	Later than any other timestamp
-infinity	timestamp	Earlier than any other timestamp
now	date, time, timestamp	Start time of the current transaction
today	date, timestamp	Today's midnight
tomorrow	date, timestamp	Tomorrow's midnight
yesterday	date, timestamp	Yesterday's midnight
allballs	time	00:00:00.00 UTC

8.7 Object Identifier Type

Aura internally uses object identifiers (OIDs) as primary keys for various system catalogs. The system does not add an OID system column to the table created by the user. The OID type indicates an object identifier.

Currently, the OID type is implemented as an unsigned four-byte integer. Therefore, you are advised not to use the OID column as the primary key in the created table.

Table 8-14 Object identifier types

Type	Reference	Description	Example
OID	-	Digital object identifier.	564182
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 parameter types.	*(integer,integer) or -(NONE,integer)
REGPROC	pg_proc	Function name.	sum
REGPROCEDURE	pg_proc	Function with parameter types.	sum(int4)
REGCLASS	pg_class	Relation name.	pg_type

 Copyright © Huawei Technologies Co., Ltd.

Type	Reference	Description	Example
REGTYPE	pg_type	Data type name.	integer

OID type: Used as a field in the database system catalog.

Example:

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

OID alias type **REGCLASS**: Mainly used for simplified lookup of object OID values.

Example:

```
postgres=# 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 | typname        |       19 |            -1
     1247 | typnamespace   |       26 |            -1
     1247 | typowner       |       26 |            -1
     1247 | typlen         |       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)
```

 Copyright © Huawei Technologies Co., Ltd.

8.8 Implicit Conversion Support Range

Currently, Aura supports implicit conversion of numeric types for tables stored in ORC or Parquet formats. This means that types with larger storage byte sizes are backward compatible with those having smaller storage byte sizes.

For example, if the actual storage is orc::SHORT (2 bytes), queries will function normally whether the table type is defined as SMALLINT, INT, or BIGINT. Below is the current support range for implicit conversions in Aura.

Table 8-15 Supported range for numeric type implicit conversions in Aura

Numeric Type Implicit Conversion in Aura	SMALLINT	INT	BIGINT	FLOAT4	FLOAT8	NUMERIC
orc::byte/ parquet::INT8 to	√	√	√	×	×	×
orc::short/ parquet::INT16 to	√	√	√	×	×	×
orc::int/ parquet::INT32 to	×	√	√	×	×	×
orc::long/ parquet::INT64 to	×	×	√	×	×	×
orc::float/parquet::32 to	×	×	×	√	×	×
orc::double/ parquet::64 to	×	×	×	×	√	×
orc::decimal/ parquet::decimal to	×	×	×	×	×	√

8.9 Complex Types

In Aura, Iceberg tables, Parquet tables, and ORC tables support reading and writing complex type columns, including both managed tables and external tables. Currently, only the struct and array types, along with their nested types, are supported.

Complex Type DDL

Currently, DDL only supports STRUCT and ARRAY types.

The definition syntax is as follows:

- **ARRAY**: Defined with the syntax ARRAY<data_type>.

 Copyright © Huawei Technologies Co., Ltd.

- **STRUCT**: Defined with the syntax STRUCT<col_name : data_type, ...>.

Here, **data_type** indicates the data type, which can be either a basic type or a complex type. For details about basic types, refer to **8.1 Mapping Between Aura and LakeFormation Data Types**.

DDL example:

```
-- Create a managed table containing complex types.
CREATE TABLE test_managed_table (
    name TEXT,
    address STRUCT<zip:INT, city:TEXT>,
    tags ARRAY<TEXT>,
    logs ARRAY<STRUCT<term:TEXT, time:TIMESTAMP>>
) store as parquet;

-- Create an external table containing complex types.
CREATE EXTERNAL TABLE test_external_table (
    name TEXT,
    address STRUCT<zip:INT, city:TEXT>,
    tags ARRAY<TEXT>,
    logs ARRAY<STRUCT<term:TEXT, time:TIMESTAMP>>
) store as orc location 'obs://test/test1';

-- After the table is created, you can view the full definition of LakeFormation.
DESCRIBE test_managed_table ;
DESCRIBE test_external_table ;
```

Complex Type DML

- STRUCT type: A STRUCT is constructed using **ROW(x,x,x,...)**, for example, **ROW(1,2,3,..)**. If the STRUCT contains more than one column, the ROW keyword can be omitted, such as (1,2,3,...).
- ARRAY type: An ARRAY is built using **ARRAY[x,x,x,..]**, for example, **ARRAY[1,2,3,...]**.
- Nested type: Nested types are constructed by applying the above rules sequentially. For example, a type like **struct<a:array<int>, b:array<int>>** would be represented as **ROW(ARRAY[1,2], ARRAY[3,4])**.

DML example:

```
-- Insert data.
INSERT INTO test_managed_table VALUES('Alice', (10001,'New York'), array['friend','colleague'],
array[('login','2024-01-01 00:00:00'),('login','2024-02-01 00:00:00')]);
INSERT INTO test_managed_table SELECT 'Bob', (90001,'Los Angeles')::STRUCT<zip:INT, city:TEXT>,
array['family'], array[('purchase','2024-02-15 00:00:00'),('purchase','2024-02-10
00:00:00')]::ARRAY<STRUCT<term:TEXT, time:TIMESTAMP>>;
INSERT INTO test_managed_table SELECT 'Charlie', (60601,'Chicago')::STRUCT<zip:INT, city:TEXT>,
array['friend','hobby'], array[NULL]::ARRAY<STRUCT<term:TEXT, time:TIMESTAMP>>;

-- View data.
SELECT * FROM test_managed_table ORDER BY name ;
  name   |        address        |        tags        |                                    logs
---------+-----------------------+--------------------
+-----------------------------------------------------------------------------------------
 Alice   | (10001,"New York")    | {friend,colleague} | {"(login,\"Mon Jan 01 00:00:00 2024\")","(login,\"Thu
Feb 01 00:00:00 2024\")"}
 Bob     | (90001,"Los Angeles") | {family}           | {"(purchase,\"Thu Feb 15 00:00:00
2024\")","(purchase,\"Sat Feb 10 00:00:00 2024\")"}
 Charlie | (60601,Chicago)       | {friend,hobby}     | {NULL}
(3 rows)
```

 Copyright © Huawei Technologies Co., Ltd.

Complex Type DQL

- The Struct type allows for direct retrieval of internal elements through point queries, where column names of the struct type should be enclosed in parentheses (()).
- Arrays function similarly to regular arrays, with specific elements accessed via indices. Note that indexing begins at 1.

DQL example:

```
-- Query complex data.
SELECT * FROM test_managed_table;

  name   |        address        |        tags        |                                    logs
---------+-----------------------+--------------------
+-----------------------------------------------------------------------------------------------
 Alice   | (10001,"New York")    | {friend,colleague} | {"(login,\"Mon Jan 01 00:00:00 2024\")","(login,\"Thu
Feb 01 00:00:00 2024\")"}
 Bob     | (90001,"Los Angeles") | {family}           | {"(purchase,\"Thu Feb 15 00:00:00
2024\")","(purchase,\"Sat Feb 10 00:00:00 2024\")"}
 Charlie | (60601,Chicago)       | {friend,hobby}     | {NULL}
(3 rows)

-- Extract struct elements.
SELECT (address).city, (address).zip FROM test_managed_table;

    city     |  zip
-------------+-------
 Los Angeles | 90001
 New York    | 10001
 Chicago     | 60601
(3 rows)

-- Extract array elements.
SELECT tags[1], tags[2] FROM test_managed_table;
  tags   |   tags
---------+----------
  friend | colleague
  family |
  friend | hobby
(3 rows)
```

 Copyright © Huawei Technologies Co., Ltd.

9 Type Conversion

9.1 Overview

Context

In SQL, each piece of data is associated with a data type that determines its behavior and usage. Aura provides an extensible data type system that is more versatile and flexible than other SQL implementations. Therefore, most type conversions in Aura are managed by common rules, which allow the use of mixed-type expressions.

The Aura scanner/analyzer only decomposes lexical elements into five basic categories: integers, floating point numbers, strings, identifiers, and keywords. Most non-numeric types are first represented as strings. The SQL language definition allows constant strings to be declared as specific types. Example:

```
SELECT text 'Origin' AS "label", '(0,0)' AS "value";
 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

The example is a text constant with the type text. If no type is explicitly declared for a string literal, the literal is initially defined as the unknown type.

In the Aura analyzer, there are four basic SQL structures that require independent type conversion rules:

- Function invocation

  Most SQL type systems are built on a rich set of functions. A function call can have one or more arguments. SQL allows function overloading. Therefore, the function to be called cannot be directly found based on the function name. The parser must select the correct function based on the parameter type provided by the function.

- Operator

  SQL allows prefix or suffix (unary) operators to be used on expressions, and allows binary operators (two parameters) to be used inside expressions. Like functions, operators can be overloaded. Therefore, the selection of operators depends on the parameter type.

 Copyright © Huawei Technologies Co., Ltd.

- Value storage

  The **INSERT** and **UPDATE** statements store the results of expressions into a table. The expression type in the statement must be the same as the target field type or can be converted to the target field type.

- UNION, CASE, and related constructs

  Because all query results in a union **SELECT** statement must be displayed in a column, the element types in each **SELECT** clause must match each other and be converted to a uniform type. Similarly, the result expression of a **CASE** construct must be converted to a unified type so that the entire case expression has a unified output type. The same requirement exists in the **ARRAY** construct and in the **GREATEST** and **LEAST** functions.

The system catalog **pg_cast** stores information about conversions between data types and how to perform these conversions. For details, see **PG_CAST**.

In the semantic analysis phase, the return value type of an expression is determined and an appropriate conversion behavior is selected. Basic data types, including **boolean**, **numeric**, **string**, **bitstring**, **datetime**, **timespan**, **geometric**, and **network**. Each type has one or more preferred types to solve the problem of type selection. Depending on the preferred type and available implicit transformations, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in an efficient way.

All type conversion rules are based on the following principles:

- Implicit conversions should never have strange or unpredictable output.
- If a query does not need implicit type conversion, the analyzer and executor should not perform more extra operations. That is, any query that matches the type and is well-formed should not consume more time in the parser and should not introduce any unnecessary implicit type conversion calls to the query.
- Additionally, if a query requires implicit conversion when calling a function, the interpreter should use the new function after the user defines a function with correct parameters.
- Empty strings (tabs, newlines, etc.) cannot be converted to numeric types.

9.2 Operators

Operator Type Resolution

1. Select the operators to be considered from the **pg_operator** system catalog. If an operator with the same parameter type and number of parameters can be found, the operator is the final operator. If multiple alternative operators are found, the most appropriate one is selected.
2. Search for the optimal match.
   a. Discard candidate operators for which the input types do not match and cannot be implicitly converted to match. The unknown text can be converted to any type in this case. If only one candidate operator is left, use it. Otherwise, go to the next step.
   b. Review all candidate operators and retain the operator that best matches the input type. Here, domains are treated as their base types. If no exact

 Copyright © Huawei Technologies Co., Ltd.

match exists, keep all candidates. If only one candidate remains, use it. Otherwise, proceed to the next step.

c. Examine all candidate operators and keep those that accept the most preferred type positions (within the input data type's type category) when type conversion is necessary. If no operator accepts the preferred type, retain all candidates. If only one candidate remains, use it. Otherwise, proceed to the next step.

d. If any input parameter is of the unknown type, inspect the type category of the corresponding parameter position among the remaining candidate operators. At each position capable of accepting the string type category, use the string type (as favoring strings is reasonable since unknown text resembles a string). Additionally, if all remaining candidate operators accept the same type category, choose that category. Otherwise, report an error (since making the correct choice requires more information). Now eliminate candidate operators that do not accept the chosen type category. Moreover, if any candidate accepts a preferred type within that category, exclude candidates that accept non-preferred types for that argument. If no operator can be kept, retain all candidates. If only one candidate remains, use it. Otherwise, proceed to the next step.

e. If both unknown and known parameters exist, and all known parameters share the same type, assume the unknown parameter also falls into that type, then verify which candidate operators accept this type at the unknown parameter position. If exactly one operator fits, use it. Otherwise, raise an error.

Examples

Example 1: Factorial operator type resolution. There is only one factorial operator (postfix !) in the system catalog, which takes **bigint** as its argument. The scanner assigns the initial type of bigint to the argument of the following query expression:

```
postgres=# SELECT 40 ! AS "40 factorial";

                  40 factorial
-------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

The analyzer performs type conversion on the parameter, rendering the query equivalent to:

```
postgres=# SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 2: String concatenation operator type analysis. A string-like syntax applies to both strings and complex extended types. Unspecified-type strings are matched against all potential candidate operators. An instance involving an untyped parameter follows:

```
postgres=# SELECT text 'abc' || 'def' AS "text and unknown";
 text and unknown
------------------
 abcdef
(1 row)
```

In this example, the analyzer seeks an operator where both parameters are of type text. Such an operator indeed exists, with both parameters being text.

 Copyright © Huawei Technologies Co., Ltd.

Below demonstrates concatenating two values of undefined types:

```
postgres=# SELECT 'abc' || 'def' AS "unspecified";
 unspecified
-------------
 abcdef
(1 row)
```

NOTE

As no type is specified in the query, there is no initial clue regarding the type here. Consequently, the parser scans through all candidate operators, discovering ones that accommodate either the string or bit-string type categories. Given the preference for the string type category, the preferred type text within this category is selected as the declared type for interpreting the unknown type text.

Example 3: Absolute value and negation operator type analysis. The Aura operator catalog has several entries for the prefix operator @. All the entries implement absolute-value operations for various numeric data types. Among them, one caters specifically to the **float8** type, deemed the preferred type within the numerical type category. Thus, upon encountering unknown inputs, Aura will use this type:

```
postgres=# SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Herein, prior to applying the chosen operator, the system implicitly transforms the unknown-type text into the **float8** type.

Example 4: Array containment operator type analysis. Consider a situation where the operator flanks a known type alongside an unknown type:

```
postgres=# SELECT array[1,2] <@ '{1,2,3}' as "is subset";
 is subset
-----------
 t
(1 row)
```

NOTE

Within Aura's **pg_operator** table, numerous entries correspond to the infix operator <@, yet only **array containment (anyarray <@ anyarray)** and **range containment (anyelement <@ anyrange)** permit an integer array on the left side. Since polymorphic pseudo-types are not regarded as preferred operator types, the resolver fails to disambiguate based thereon. Nonetheless, the ultimate guideline for **identifying the optimal match** stipulates presuming the unknown-type literal aligns with another known input type—that is, exclusively one of the pair of operators may align—hence opting for array containment. (Selecting range containment would trigger an error due to improper string formatting.)

9.3 Functions

Function Type Resolution

1. Select all possible functions from the system catalog **pg_proc**. If a function name without schema qualification is used, the function is assumed to be among those in the current search path. If a qualified function name is provided, only functions in the specified schema are considered.

   If multiple functions with different parameter types are found in the search path, an appropriate function is chosen from them.

 Copyright © Huawei Technologies Co., Ltd.

2. Look for a function that exactly matches the input parameter types. If one is found, use it. If the input parameters are all of the unknown type, no matching function is found.
3. If no exact match is found, check if the function is a special type conversion function.
4. Search for the optimal match.
   a. Discard candidate functions whose input types do not match and cannot be implicitly converted to match. **unknown** literals can be converted to anything in this case. If only one candidate remains, use it. Otherwise, proceed to the next step.
   b. Iterate through all candidate functions and retain those with the most precise input type matches. Here, domains are treated the same as their base types. If no function matches precisely, all candidates are kept. If only one candidate remains, use it. Otherwise, proceed to the next step.
   c. Iterate through all candidate functions and retain those that accept the most preferred type positions when type conversion is needed. If no function accepts the preferred type, all candidates are kept. If only one candidate remains, use it. Otherwise, proceed to the next step.
   d. If any input parameter is of the **unknown** type, examine the type category of the remaining candidate functions corresponding to the parameter position. At each position where the string type category is accepted, use the string type (this preference for strings is justified because **unknown** literals indeed resemble strings). Additionally, if all remaining candidates accept the same type category, choose that category. Otherwise, throw an error (since the correct choice cannot be determined without further information). Now eliminate candidates that do not accept the selected type category, and then remove candidates that accept a non-preferred type at that parameter position if any candidate accepts a preferred type within that category. If none of the candidates pass these tests, all candidates are kept. If only one candidate function satisfies the criteria, use it. Otherwise, proceed to the next step.
   e. If there are both **unknown** and known-type parameters, and all known-type parameters share the same type, assume the **unknown** parameters are also of that type, and verify which candidate function can accept this type at the unknown-parameter positions. If exactly one candidate fits, use it. Otherwise, an error occurs.

Examples

Example 1: Rounding function parameter type resolution. There is only one **round** function with two parameters (the first is **numeric**, the second is **integer**). Thus, the following query automatically converts the first parameter of **integer** type to **numeric**.

```
postgres=# SELECT round(4, 4);
 round
--------
 4.0000
(1 row)
```

It is actually transformed by the parser into:

```
postgres=# SELECT round(CAST (4 AS numeric), 4);
```

 Copyright © Huawei Technologies Co., Ltd.

Since numeric constants with decimal points are initially assigned the **numeric** type, the following query does not require type conversion and might be slightly more efficient:

```
postgres=# SELECT round(4.0, 4);
```

Example 2: Substring function type resolution. There are several **substr** functions, one of which takes **text** and **integer** types. When called with an untyped string constant, the system chooses the candidate function accepting the preferred type (**text** type) within the **string** type category.

```
postgres=# SELECT substr('1234', 3);
 substr
--------
     34
(1 row)
```

If the string is declared as **varchar** type, as if retrieved from a table, the parser attempts to convert it to **text** type:

```
postgres=# SELECT substr(varchar '1234', 3);
 substr
--------
     34
(1 row)
```

Post-transformation by the parser, it essentially becomes:

```
postgres=# SELECT substr(CAST (varchar '1234' AS text), 3);
```

NOTE

The parser knows from the **pg_cast** catalog that **text** and **varchar** are binary-compatible, meaning one can be passed to a function expecting the other without any physical conversion. Hence, in this scenario, no actual type conversion occurs.

Moreover, if the function is called with an **integer** parameter, the parser tries to convert it to **text** type:

```
postgres=# SELECT substr(1234, 3);
substr
--------
 34
(1 row)
```

Post-transformation by the parser, it essentially becomes:

```
postgres=# SELECT substr(CAST (1234 AS text), 3);
 substr
--------
     34
(1 row)
```

9.4 Value Storage

Data Type Resolution for Value Storage

1. Find an exact match with the target field.
2. Attempt to directly convert the expression to the target type. If there is a registered conversion function between these two types, invoke the conversion function directly. If the expression is of an unknown text type, the contents of the text string will be passed to the input conversion process of the target type.

 Copyright © Huawei Technologies Co., Ltd.

3. Check if the target type has a length conversion. A length conversion is a transformation from a type to itself. If one is found in the **pg_cast** catalog, apply it to the expression before storing it into the target column. This kind of conversion function always takes an extra parameter of type **integer**, receiving the **atttypmod** value of the target column (essentially its declared length; the interpretation of **atttypmod** varies across different data types), and might also take a third parameter of type **boolean** indicating whether the conversion is explicit or implicit. The conversion function is responsible for enforcing length-related semantics, such as length checks or truncations.

Examples

Conversion of the character storage type. For a statement where the target column is defined as **character(20)**, the following demonstrates that the length of the stored value is accurate:

```
CREATE TABLE x1
(
   customer_sk             integer,
   customer_id             char(20),
   first_name              char(6),
   last_name               char(8)
)
store AS orc;

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

Here is what truly occurs: Two **unknown** literals are by default resolved to **text**, enabling the **||** operator to resolve to text concatenation. Subsequently, the **text** result of the operation is transformed into **bpchar** (blank-padded character, the internal designation for the **character** type) to align with the target column type. Nonetheless, the transition from **text** to **bpchar** is binary-compatible, implying it is implicit and does not involve any actual function calls. Ultimately, the length conversion function **bpchar(bpchar, integer, boolean)** is located within the system catalog and implemented upon both the operation's result and the stored column length. Functions associated with this type execute necessary length verifications along with supplementary blank padding.

9.5 UNION, CASE, and Related Constructs

SQL UNION constructs match different data types and output a unified data type result set. Since all query results in a **SELECT UNION** statement must be displayed in a single column, the element types in each **SELECT** clause must match each other and be converted into a unified data type. Similarly, the result expression of a **CASE** construct must be converted to a unified type so that the entire case expression has a unified output type. The same requirement exists in the **ARRAY** construct and in the **GREATEST** and **LEAST** functions.

 Copyright © Huawei Technologies Co., Ltd.

Resolution for UNION, CASE, and Related Constructs

- If all inputs are of the same data type, excluding the **unknown** type (that is, when the input string literal's type is undeclared and initially treated as an unknown type), it resolves to the same data type as the input.
- If all inputs are of the **unknown** type, they resolve to the **text** type (the preferred type within the string category). Otherwise, **unknown** inputs are ignored.
- If the inputs do not belong to the same type category, the query fails (excluding the **unknown** type).
- If the input types belong to the same type category, the preferred type of that category is selected (unless the **union** operation chooses the type of the first branch as the selected type).

  NOTE

  In the **pg_type** system catalog, **typcategory** indicates the data type category, and **typispreferred** indicates whether the data type is the preferred one within its **typcategory**.
- Convert all inputs to the selected type (retaining the original length for strings). If there is no implicit conversion from the given input to the selected type, the conversion fails.
- If the input includes **json**, **txid_snapshot**, **sys_refcursor**, or geometric types, **union** operations cannot be performed.

 Copyright © Huawei Technologies Co., Ltd.

10 Expression

10.1 Simple Expressions

Logical Expressions

For details about the operators and calculation rules of logical expressions, refer to **11.7 Logical Operators**.

Comparison Expressions

Commonly used comparison operators can be found in **11.8 Comparison Operators**.

In addition to comparison operators, the following structures can also be used:

- 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 equivalent structures, though non-standard:

  expression ISNULL

  expression NOTNULL

NOTICE

Do not write **expression=NULL** or **expression<>(!=)NULL**, as **NULL** represents an unknown value, making it impossible to determine equality between two unknowns with such expressions.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
postgres=# SELECT 2 BETWEEN 1 AND 3 AS RESULT;
 result
----------
 t
(1 row)

postgres=# SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
 result
----------
 t
(1 row)

postgres=# SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
 result
----------
 f
(1 row)

postgres=# SELECT 2 < 1 OR 2 > 3 AS RESULT;
 result
----------
 f
(1 row)

postgres=# SELECT 2+2 IS NULL AS RESULT;
 result
----------
 f
(1 row)

postgres=# SELECT 2+2 IS NOT NULL AS RESULT;
 result
----------
 t
(1 row)

postgres=# SELECT 2+2 ISNULL AS RESULT;
 result
----------
 f
(1 row)

postgres=# SELECT 2+2 NOTNULL AS RESULT;
 result
----------
 t
(1 row)

postgres=# SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;
 result
----------
 t
(1 row)

postgres=# SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;
 result
----------
 f
(1 row)
```

10.2 Conditional Expressions

When executing SQL statements, conditional expressions can be used to filter data that meets specific criteria.

 Copyright © Huawei Technologies Co., Ltd.

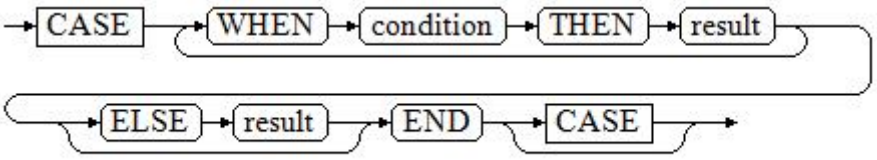
There are several main types of conditional expressions:

- CASE

  A **CASE** expression is a conditional expression, similar to the **CASE** statement found in other programming languages.

  Figure 10-1 shows the syntax of a **CASE** expression.

  Figure 10-1 case::=

  The **CASE** clause can be used within a valid expression. A **condition** is an expression that returns a Boolean data type.

  - If the result is **true**, the result of the **CASE** expression corresponds to the matching condition's result.
  - If the result is **false**, subsequent **WHEN** or **ELSE** clauses are processed similarly.
  - If none of the **WHEN** conditions are set to **true**, the result of the expression is determined by the **ELSE** clause's result. If the **ELSE** clause is omitted and no conditions match, the result is NULL.

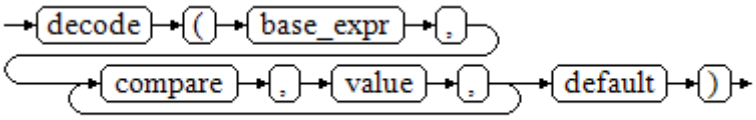
  Example:

```
postgres=# CREATE TABLE case_when_t1(CW_COL1 INT) store AS orc;

postgres=# INSERT INTO case_when_t1 VALUES (1), (2), (3);

postgres=# SELECT * FROM case_when_t1 order by cw_col1;
 cw_col1
---------
       1
       2
       3
(3 rows)

postgres=# SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN 'two'
ELSE 'other' END FROM case_when_t1 order by cw_col1;
 cw_col1 | case
---------+-------
       1 | one
       2 | two
       3 | other
(3 rows)

DROP TABLE case_when_t1;
```

- DECODE

  Figure 10-2 shows the syntax of a **DECODE** expression.

  Figure 10-2 decode::=

 Copyright © Huawei Technologies Co., Ltd.

This compares the **base_expr** with each **compare(n)**. If there is a match, it returns the corresponding **value(n)**. If no matches occur, it returns the default.

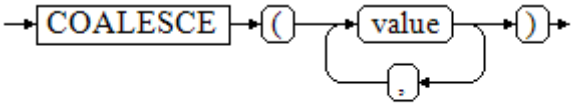
See **11.14 Conditional Expression Functions** for examples.

```
postgres=# SELECT DECODE('A','A',1,'B',2,0);
 case
------
    1
(1 row)
```

- COALESCE

  Figure 10-3 shows the syntax of a **COALESCE** expression.

  Figure 10-3 coalesce::=

  COALESCE returns the first non-NULL value among its parameters. If all parameters are NULL, it returns NULL. It is commonly used to substitute NULL with a default value during data display. Similar to a **CASE** expression, **COALESCE** only evaluates the necessary parameters to determine the result. Thus, parameters to the right of the first non-null one remain unevaluated.

  Example:

```
postgres=#
CREATE TABLE c_tabl(description varchar(10), short_description varchar(10), last_value varchar(10))
store AS orc;

postgres=# INSERT INTO c_tabl VALUES('abc', 'efg', '123');
postgres=# INSERT INTO c_tabl VALUES(NULL, 'efg', '123');

postgres=# INSERT INTO c_tabl VALUES(NULL, NULL, '123');

postgres=# SELECT description, short_description, last_value, COALESCE(description, short_description,
last_value) FROM 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)

postgres=# DROP TABLE c_tabl;
```

  If **description** is not NULL, the value of **description** is returned. Otherwise, the next parameter **short_description** is evaluated. If **short_description** is not NULL, the value of **short_description** is returned. Otherwise, the next parameter **last_value** is evaluated. If **last_value** is not NULL, the value of **last_value** is returned. Otherwise, **(none)** is returned.

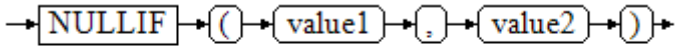
```
postgres=# SELECT COALESCE(NULL,'Hello World');
   coalesce
---------------
 Hello World
(1 row)
```

- NULLIF

  Figure 10-4 shows the syntax of a **NULLIF** expression.

 Copyright © Huawei Technologies Co., Ltd.

Figure 10-4 nullif::=



NULLIF returns NULL only if **value1** equals **value2**. Otherwise, it returns **value1**.

Example:

```
postgres=#
CREATE TABLE null_if_t1 (
    NI_VALUE1 VARCHAR(10),
    NI_VALUE2 VARCHAR(10)
) store AS orc;

postgres=# INSERT INTO null_if_t1 VALUES('abc', 'abc');
postgres=# INSERT INTO null_if_t1 VALUES('abc', 'efg');

postgres=# SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM null_if_t1 ORDER
BY 1, 2, 3;

 ni_value1 | ni_value2 | nullif
-----------+-----------+--------
 abc       | abc       |
 abc       | efg       | abc
(2 rows)
postgres=# DROP TABLE null_if_t1;
```

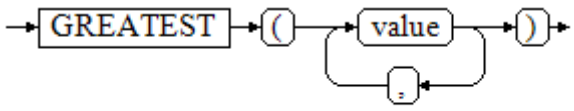
Returns **NULL** if **value1** equals **value2**. Otherwise, it returns **value1**.

```
postgres=# SELECT NULLIF('Hello','Hello World');
 nullif
--------
 Hello
(1 row)
```

- GREATEST (maximum value) and LEAST (minimum value)

  Figure 10-5 shows the syntax of a **GREATEST** expression.

  Figure 10-5 greatest::=

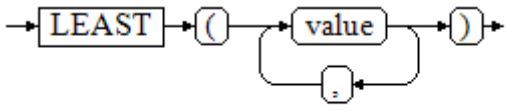
  

  It selects the largest value from a list of numeric expressions.

  ```
  postgres=# SELECT greatest(9000,155555,2.01);
   greatest
  ----------
    155555
  (1 row)
  ```

  Figure 10-6 shows the syntax of a **LEAST** expression.

  Figure 10-6 least::=

 Copyright © Huawei Technologies Co., Ltd.

It selects the smallest value from a list of numeric expressions.

All numeric expressions must be convertible to a common data type, which becomes the result type.

NULL values in the list are omitted. The result is **NULL** only if all expressions evaluate to **NULL**.

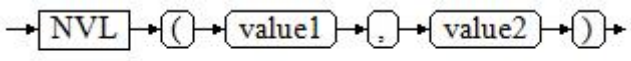
```
postgres=# SELECT least(9000,2);
 least
-------
     2
(1 row)
```

See **11.14 Conditional Expression Functions** for examples.

- NVL

  Figure 10-7 shows the syntax of an **NVL** expression.

  Figure 10-7 nvl::=

  If **value1** is NULL, **value2** is returned. If **value1** is not NULL, **value1** is returned.

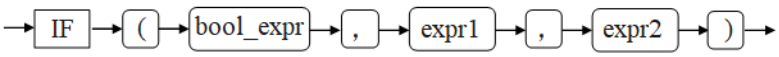
  Example:

  ```
  postgres=# SELECT nvl(null,1);
  nvl
  -----
   1
  (1 row)
  postgres=# SELECT nvl ('Hello World' ,1);
        nvl
  ---------------
   Hello World
  (1 row)
  ```

- IF

  Figure 10-8 shows the syntax of an **IF** expression.

  Figure 10-8 if::=

  If the value of **bool_expr** is **true**, **expr1** is returned. Otherwise, **expr2** is returned.

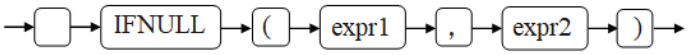
  See **11.14 Conditional Expression Functions** for examples.

- IFNULL

  Figure 10-9 shows the syntax of an **IFNULL** expression.

 Copyright © Huawei Technologies Co., Ltd.

Figure 10-9 ifnull::=



If **expr1** is not NULL, **expr1** is returned. Otherwise, **expr2** is returned.

See **11.14 Conditional Expression Functions** for examples.

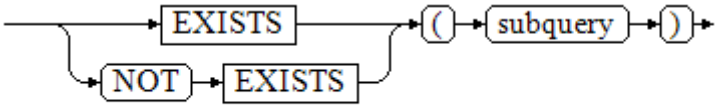
10.3 Subquery Expressions

Subquery expressions mainly include the following types:

- EXISTS/NOT EXISTS

  Figure 10-10 shows the syntax of an **EXISTS/NOT EXISTS** expression.

  Figure 10-10 EXISTS/NOT EXISTS::=

  The parameter of **EXISTS** is an arbitrary **SELECT** statement, or subquery. The system processes the subquery to determine whether it returns rows. If it returns at least one row, the result of **EXISTS** is **true**. If the subquery does not return any rows, the result of **EXISTS** is **false**.

  This subquery typically runs just enough to determine whether it can produce at least one row, rather than running to completion.

  Example:

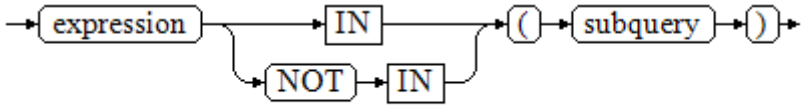
  ```
  postgres=# SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE EXISTS (SELECT d_dom FROM tpcds.date_dim WHERE d_dom = store_returns.sr_reason_sk and sr_customer_sk <10);
  sr_reason_sk | sr_customer_sk
  --------------+----------------
            13 |              2
            22 |              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 10-11 shows the syntax of an **IN/NOT IN** expression.

 Copyright © Huawei Technologies Co., Ltd.

Figure 10-11 IN/NOT IN::=



On the right is a parenthesized subquery that must return only one column. The left-hand expression computes and compares against each row of the subquery result. If any matching subquery row is found, the result of **IN** is **true**. If no matching row is found, the result is **false** (including the case where the subquery returns no rows).

NULLs in expressions or subquery rows follow the rules SQL uses for handling combinations of Boolean values and NULLs. If the corresponding columns of two rows are both equal and non-null, then the two rows are equal. If any corresponding column is unequal and non-null, then the two rows are unequal. Otherwise, the result is unknown (**NULL**). If every row's result is either unequal or NULL, and there is at least one NULL, then the result of **IN** is NULL.

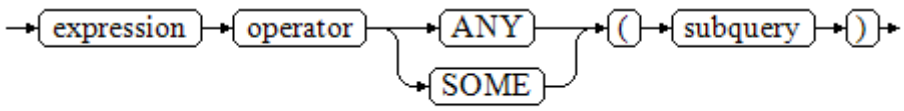
Example:

```
postgres=#  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 10-12 shows the syntax of an **ANY/SOME** expression.

  Figure 10-12 any/some::=



  On the right is a parenthesized subquery that must return only one column. The left-hand expression uses an operator to compute and compare against

 Copyright © Huawei Technologies Co., Ltd.

each row of the subquery result, which must be a Boolean value. If at least one **true** value is obtained, the result of **ANY** is **true**. If all results are **false**, the result is **false** (including the case where the subquery returns no rows). **SOME** is synonymous with **ANY**. **IN** and **ANY** can be used interchangeably.

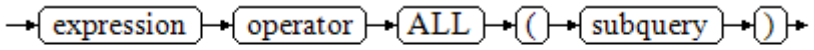
Example:

```
postgres=# 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 10-13 shows the syntax of an **ALL** expression.

  Figure 10-13 all::=

  On the right is a parenthesized subquery that must return only one column. The left-hand expression uses an operator to compute and compare against each row of the subquery result, which must be a Boolean value. If all results are **true**, the result of **ALL** is **true** (including the case where the subquery returns no rows). If at least one **false** value is obtained, the result is **false**.

  Example:

```
postgres=# 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)
```

10.4 Array Expressions

IN

expression **IN** *(value [, ...])*

The parentheses on the right contain a list of expressions. The result of the expression on the left is compared with the contents of the expression list. If the

 Copyright © Huawei Technologies Co., Ltd.

contents of the list match the result of the expression on the left, the result of **IN** is **true**. If there is no matching result, the result of **IN** is **false**.

Example:

```
postgres=# 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 satisfy the condition of the expression and at least one result from the expression list on the right is empty, then the result of **IN** is null instead of **false**. This handling aligns with the Boolean combination rules for SQL returning null values.

NOT IN

expression **NOT IN** *(value [, ...])*

The parentheses on the right contain a list of expressions. The result of the expression on the left is compared with the contents of the expression list. If none of the contents in the list matches the result of the expression on the left, the result of **NOT IN** is **true**. If there is a matching content, the result of **NOT IN** is **false**.

Example:

```
postgres=# SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
 result
----------
 t
(1 row)
```

NOTE

If the query returns a null result, or the expression list does not satisfy the condition of the expression and at least one result from the expression list on the right is null, then the result of **NOT IN** is null instead of **false**. This handling aligns with the Boolean combination rules for SQL returning null values.

Hint: In all cases, **X NOT IN Y** is equivalent to **NOT (X IN Y)**.

ANY/SOME (array)

expression operator **ANY** *(array expression)*

expression operator **SOME** *(array expression)*

```
postgres=# SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
postgres=# SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

The parentheses on the right contain an array expression, which must yield an array value. The result of the expression on the left is computed and compared

 Copyright © Huawei Technologies Co., Ltd.

using operators against each row's result of the array expression. The comparison result must be a Boolean value.

- If at least one comparison yields a **true** value, the result of **ANY** is **true**.
- If no comparison yields a **true** value, the result of **ANY** is **false**.

NOTE

If there is no **true** value in the results and the array expression produces at least one null value, the value of **ANY** is NULL instead of **false**. This handling aligns with the Boolean combination rules for SQL returning null values.

SOME is synonymous with **ANY**.

ALL (array)

expression operator **ALL** *(array expression)*

The parentheses on the right contain an array expression, which must yield an array value. The result of the expression on the left is computed and compared using operators against each row's result of the array expression. The comparison result must be a Boolean value.

- If all comparisons yield **true** values (including when the array contains no elements), the result of **ALL** is **true**.
- If any comparison yields a **false** value, the result of **ALL** is **false**.

If the array expression produces a NULL array, the result of **ALL** is NULL. If the value of the expression on the left is NULL, the result of **ALL** is typically also NULL (though some non-strict comparison operators might yield different results). Additionally, if the array expression on the right includes a null element and the comparison does not yield a **false** value, the result of **ALL** will be NULL (again, some non-strict comparison operators might yield different results) instead of **true**. This handling aligns with the Boolean combination rules for SQL returning null values.

```
postgres=# SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

10.5 Row Expressions

Syntax:

row_constructor operator row_constructor

Both sides are a row constructor. The values of two rows must have the same number of fields. Each row is compared. Operators such as =, <>, <, <=, and >= or a semantically equivalent operator can be used for row comparison.

The usage of <> differs slightly from other operators. If all fields of both rows are non-null and equal, the two rows are considered equal. If any field of the two rows is non-null and unequal, the two rows are considered unequal. Otherwise, the comparison result is unknown (null).

 Copyright © Huawei Technologies Co., Ltd.

For operators <, <=, >, and >=, elements in a row are compared from left to right until a pair of unequal elements or a pair of null elements is encountered. If there is at least one null value in the pair of elements, the comparison result is unknown (null). Otherwise, the comparison result of the pair of elements is the final result.

Example:

```
postgres=# SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;
 result
----------
 t
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

11 Functions and Operators

An operator can process one or more operands and can be placed before, after, or between operands. After the processing is complete, the processing result is returned.

Functions encapsulate service logic to implement specific functions. A function may or may not have parameters. A function has a return type. After the function is executed, the execution result is returned.

You can modify system functions. However, after the modification, the semantics of the system functions may change, which may cause system control disorder. In normal cases, you are not allowed to manually modify system functions.

11.1 Character Processing Functions and Operators

In everyday data processing tasks, you often need to perform various operations on strings, such as concatenation and pattern matching. However, when using Aura, you may face inefficiencies due to limited or unfamiliar string manipulation capabilities. How can we improve the efficiency and accuracy of string handling? Aura offers a range of character processing functions and operators designed for connecting strings with other strings or non-string values, as well as performing pattern matching. These tools effectively address these challenges, enhancing overall data processing performance.

bit_length(string)

Description: The bit length of a string.

Return type: integer.

Example:

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

 Copyright © Huawei Technologies Co., Ltd.

btrim(string text [, characters text])

Description: Removes the longest string containing only characters from **characters** (default is whitespace) from the beginning and end of **string**.

Return type: text.

Example:

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

char_length(string) or character_length(string)

Description: The number of characters in a string.

Return type: integer.

Example:

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

fq_iceberg_bucket(int,text)

Description: Calculates the bucket number for a specified string. The first int parameter defines the total number of buckets.

Return type: int.

Example:

```
select fq_iceberg_bucket(13, 'Hello, China!');
 fq_iceberg_bucket
-------------------
                 2
(1 row)
```

fq_iceberg_truncate(int,text)

Description: Generates a prefix substring of a specified length. The first int parameter sets the prefix length in characters, and the second text parameter is the input string.

Return type: text.

Example:

```
select *, length(t)  from fq_iceberg_truncate (2, 'Hello, China!') as t;
 iceberg_truncate | length
------------------+--------
 Hello            |      2
(1 row)
```

For more examples of fq_iceberg_truncate, see **Partition Expression: truncate**.

 Copyright © Huawei Technologies Co., Ltd.

instr(text,text,int,int)

Description: Position of the string matching function. The first int specifies the starting position for matching, and the second int specifies the number of matches.

Return type: integer.

Example:

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

lengthb(text/bpchar)

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

Return type: integer.

Example:

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

NOTE

- If there is a line break in the string, such as a string consisting of a line break and a space, the value of **LENGTH** and **LENGTHB** in Aura is **2**.
- In Aura, *n* in the CHAR(n) type indicates the number of characters. Therefore, for multibyte encoded character sets, the length returned by the **LENGTHB** function may be larger than *n*.

left(str text, n int)

Description: Returns the first *n* characters of a string.

When *n* is negative, all but the last */n/* characters are returned.

Return type: text.

Example:

```
SELECT left('abcde', 2);
 left
------
 ab
(1 row)
```

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

Description: Pads the **string** to the specified length using the **fill** character (a space by default). If the **string** is already longer than the **length**, its trailing part is truncated.

Return type: text.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT lpad('hi', 5, 'xyza');
 lpad
-------
 xyzhi
(1 row)
```

octet_length(string)

Description: Number of bytes in a string.

Return type: integer.

Example:

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

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

Description: Replaces substrings. **FROM int** indicates the character from which the first string is replaced. **For int** indicates the number of characters to be replaced in the first string.

Return type: text.

Example:

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

position(substring in string)

Description: Specifies the position of the substring in the string. If the string does not contain any substring, **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: Current client encoding name.

Return type: name.

Example:

 Copyright © Huawei Technologies Co., Ltd.

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

quote_ident(string text)

Description: Returns the identifier form (quotation marks are used as required) applicable to the SQL statement. Quotation marks are added only when necessary (the string contains non-identifier characters or characters that can be converted to uppercase or lowercase). The quotation marks embedded in the return value are written twice.

Return type: text.

Example:

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

quote_literal(string text)

Description: Returns the form (quotation marks are used as required) suitable for use as text in a SQL statement.

Return type: text.

Example:

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

If the following information is displayed, the text is escaped.

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

If the following information is displayed, the backslash is written twice.

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

If the parameter is NULL, a null value is returned. If the parameter is NULL, the **quote_nullable** function is recommended.

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

(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

quote_literal(value anyelement)

Description: Forcibly converts the given value to text and adds quotation marks as the text.

Return type: text.

Example:

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

If the following information is displayed, the fixed value is escaped.

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

If the following information is displayed, the backslash is written twice.

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

quote_nullable(string text)

Description: Returns a string in a SQL statement (quotation marks are used as required).

Return type: text.

Example:

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

If the following information is displayed, the text is escaped.

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

If the following information is displayed, the backslash is written twice.

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

If the parameter is NULL, a null value is returned.

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

 Copyright © Huawei Technologies Co., Ltd.

```
 NULL
(1 row)
```

quote_nullable(value anyelement)

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

Return type: text.

Example:

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

If the following information is displayed, the fixed value is escaped.

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

If the following information is displayed, the backslash is written twice.

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

If the parameter is NULL, a null value is returned.

```
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:

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

substring(string from *pattern*)

Description: Extracts substring matching POSIX regular expression. If there is no match, a null value is returned. Otherwise, the portion of the text that matches the pattern is returned.

Return type: text.

 Copyright © Huawei Technologies Co., Ltd.

Example:

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

NOTE

If the POSIX regular expression pattern contains any parentheses, the text that matches the first pair of subexpressions (corresponding to the first left parenthesis) is returned. If you want to use parentheses in an expression without causing this exception, you can add a pair of parentheses outside the entire expression.

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

Description: Returns all substrings that match the POSIX regular expression in **string**. If the pattern does not match, the function returns no rows. If the pattern contains no parenthesized subexpressions, each row returned is a single-element text array containing the substring matching the whole pattern. If the pattern contains a parenthetical subexpression, this function returns a text array whose *n*th element is the substring that matches the *n*th parenthetical subexpression of the pattern.

The **flags** parameter is optional and contains zero or more single-letter flags that change the behavior of the function. **i** indicates case-insensitive matching, and **g** indicates replacing each matched substring, not just the first one.

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)
```

 Copyright © Huawei Technologies Co., Ltd.

CAUTION

If there is no subquery and the **regexp_matches** function does not find a match, the data in the table will not be output. This is generally not the desired result and such usage should be avoided. You are advised 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 the **string** using POSIX regular expressions as delimiters. Similar to **regexp_split_to_table**, but **regexp_split_to_array** returns its results as a text array.

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 the **string** using POSIX regular expressions as delimiters. 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 start of the string) to the beginning of the current 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 includes zero or more single-character flags that alter the function's behavior. **i** indicates case-insensitive matching, and **g** indicates replacing each matched substring, not just the first one.

Return type: setof text.

Example:

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

 Copyright © Huawei Technologies Co., Ltd.

```
 world
(2 rows)
```

CAUTION

If there is no subquery and the **regexp_split_to_table** function does not find a match, the data in the table will not be output. This is generally not the desired result and such usage 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: Repeats the **string** a specified number of times.

Return type: text.

Example:

```
SELECT repeat('Pg', 4);
  repeat
----------
 PgPgPgPg
(1 row)
```

replace(string text, from text, to text)

Description: Replaces all instances of the substring **from** within the string with the substring **to**.

Return type: text.

Example:

```
SELECT replace('abcdefabcdef', 'cd', 'XXX');
    replace
----------------
 abXXXefabXXXef
(1 row)
```

reverse(str)

Description: Returns the string in reverse order.

Return type: text.

Example:

 Copyright © Huawei Technologies Co., Ltd.

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

right(str text, n int)

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

If n is negative, an empty string is returned.

Return type: text.

Example:

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

postgres=# SELECT right('abcde', -2);
 right
-------

(1 row)
```

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

Description: Pads the **string** to the specified length using the **fill** character (a space by default). If the **string** is already longer than the **length**, its trailing part is truncated.

Return type: text.

Example:

```
SELECT rpad('hi', 5, 'xy');
 rpad
-------
 hixyx
(1 row)
```

substrb(text,int,int)

Description: Extracts substrings. The first **int** indicates the start position of the extraction, and the second **int** indicates the number of bytes to be extracted.

Return type: text.

Example:

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

substrb(text,int)

Description: Extracts substrings. **int** indicates the start position of the extraction.

Return type: text.

 Copyright © Huawei Technologies Co., Ltd.

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: '||42 AS RESULT;
  result
-----------
 Value: 42
(1 row)
```

split_part(string text, delimiter text, field int)

Description: Splits **string** on **delimiter** and returns the **field**-th substring (based on the text of the first appeared delimiter).

Return type: text.

Example:

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

strpos(string, substring)

Description: Specifies the position of a substring. Same as **position(substring in string)**, but the parameter sequence is opposite.

Return type: integer.

Example:

```
SELECT strpos('source', 'rc');
 strpos
--------
```

 Copyright © Huawei Technologies Co., Ltd.

```
     4
(1 row)
```

to_hex(number int or bigint)

Description: Converts number to its hexadecimal representation.

Return type: text.

Example:

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

translate(string text, from text, to text)

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

Return type: text.

Example:

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

length(string)

Description: Gets the number of characters in string.

Return type: integer.

Example:

```
SELECT length('abcd');
 length
--------
      4
(1 row)
```

lengthb(string)

Description: Gets the number of bytes in **string**. The value depends on character sets. For instance, the number of returned bytes varies between GBK and UTF-8.

Return type: integer.

Example:

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

 Copyright © Huawei Technologies Co., Ltd.

substr(string,from)

Description:

Extracts substrings from **string**.

from indicates the start position of the extraction.

- When **from** is **0**, it is treated as **1**.
- When **from** is positive, all characters are extracted from **from** to the end.
- When **from** is negative, the last *n* characters of the **string** are extracted, where *n* is the absolute value of **from**.

Return type: varchar.

Example:

When **from** is positive:

```
SELECT substr('ABCDEF',2);
 substr
--------
 BCDEF
(1 row)
```

When **from** is negative:

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

substr(string,from,count)

Description:

Extracts substrings from **string**.

from indicates the start position of the extraction.

count indicates the length of the extracted substring.

- When **from** is **0**, it is treated as **1**.
- When **from** is positive, **count** characters are extracted starting from **from**.
- When **from** is negative, **count** characters are extracted starting from the *n*th to last character, where *n* is the absolute value of **from**.
- When **count** is less than 1, null is returned.

Return type: varchar.

Example:

When **from** is positive:

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

When **from** is negative:

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT substr('ABCDEF',-3,2);
 substr
--------
 DE
(1 row)
```

substrb(string,from)

Description: This function has the same functionality as **SUBSTR(string,from)**, but the calculation unit is bytes.

Return type: bytea.

Example:

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

substrb(string,from,count)

Description: This function has the same functionality as **SUBSTR(string,from,count)**, but the calculation unit is bytes.

Return type: bytea.

Example:

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

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

Definition: This function removes specified characters from the beginning, end, or both sides of a string. If no characters are specified, it defaults to removing whitespace (e.g., spaces, tabs).

Return type: varchar.

Example:

```
SELECT trim(BOTH 'x' FROM 'xTomxx');
 btrim
-------
 Tom
(1 row)
SELECT trim(LEADING 'x' FROM 'xTomxx');
 ltrim
-------
 Tomxx
(1 row)
SELECT trim(TRAILING 'x' FROM 'xTomxx');
 rtrim
-------
 xTom
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

rtrim(string [, characters])

Definition: This function removes specified characters from the end of a string. If no characters are specified, it defaults to removing whitespace (e.g., spaces, tabs).

Return type: varchar.

Example:

```
SELECT rtrim('TRIMxxxx','x');
 rtrim
-------
 TRIM
(1 row)

SELECT rtrim('trimxxaaxxa', 'xa');
 rtrim
-------
 trim
(1 row)
```

ltrim(string [, characters])

Definition: This function removes specified characters from the beginning of a string. If no characters are specified, it defaults to removing whitespace (e.g., spaces, tabs).

Return type: varchar.

Example:

```
SELECT ltrim('xxxxTRIM','x');
 ltrim
-------
 TRIM
(1 row)

SELECT ltrim('axaxxxTRIM','xa');
 ltrim
-------
 TRIM
(1 row)
```

upper(string)

Description: Converts the string to uppercase.

Return type: varchar.

Example:

```
SELECT upper('tom');
 upper
-------
 TOM
(1 row)
```

lower(string)

Description: Converts the string to lowercase.

Return type: varchar.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT lower('TOM');
 lower
-------
 tom
(1 row)
```

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

Description: Pads the **string** to the specified length using the **fill** character (a space by default). If the **string** is already longer than the **length**, its trailing part is truncated.

length in Aura indicates the character length. A Chinese character counts as one character.

Return type: varchar.

Example:

```
SELECT rpad('hi',5,'xyza');
 rpad
-------
 hixyz
(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) of the **string**.

- When **position** is set to **0**, **0** is returned.
- If the value of **position** is a negative number, the system searches for the string from the last *n*th character. *n* is the absolute value of **position**.

This function uses characters as the calculation unit. For example, a Chinese character counts as 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: Queries and returns the value of the substring **position** that appears for the first time from the position specified by **position** (1 by default) of **string**. The calculation unit is character. If no substring exists in the **string**, **0** is returned.

 Copyright © Huawei Technologies Co., Ltd.

Return type: integer.

Example:

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

initcap(string)

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

Return type: text.

Example:

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

ascii(string <str>)

Description: ASCII code of the first character in the **string**.

Return type: integer.

Example:

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

replace(string varchar, search_string varchar, replacement_string varchar)

Description: Replaces all substrings **search_string** in **string** with substrings **replacement_string**.

Return type: varchar.

Example:

```
SELECT replace('jack and jue','j','bl');
    replace
----------------
 black and blue
(1 row)
```

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

Description: Adds a series of **repeat_strings** (a space by default) to the left of **string** to form a new string whose total length is *n*.

 Copyright © Huawei Technologies Co., Ltd.

If the length of the **string** is longer than the specified **length**, the function truncates the **string** and returns the string of the specified **length**.

Return type: varchar.

Example:

```
SELECT lpad('PAGE 1',15,'*.');
      lpad
-----------------
 *.*.*.*.*PAGE 1
(1 row)
SELECT lpad('hello world',5,'abcd');
 lpad
-------
 hello
(1 row)
```

concat(str1,str2)

Description: Concatenates and returns **str1** and **str2**.

If an input parameter is **NULL**, **NULL** is returned.

Return type: varchar.

Example:

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

chr(integer)

Description: Specifies the character of the ASCII code.

Return type: varchar.

Example:

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

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

Description: Replaces the substring that matches the POSIX regular expression. If no pattern is matched, the unmodified string is returned. If a match is found, the matching substring in the returned string is replaced with the **replacement** string.

The **replacement** string can contain **\n**, where **\n** ranges from 1 to 9, indicating that the substring of the *n*th parenthesized subexpression in the matching pattern within the string should be inserted. Additionally, it can contain **\&**, indicating that the substring matching the entire pattern should be inserted.

The optional **flags** parameter includes zero or more single-character flags that alter the function's behavior, as detailed in the following table.

 Copyright © Huawei Technologies Co., Ltd.

Table 11-1 flags parameter options

Option	Description
g	Replace every matched substring rather than just the first (by default, only the first matched substring is replaced).
B	By default, use the Henry Spencer regex library and its regex syntax. Specifying the B option prioritizes using the Boost.Regex library and its regex syntax. The following two scenarios will also automatically switch to using the Henry Spencer regex library and its regex syntax when the B option is specified: • Any combination of **p**, **q**, **w**, and **x** characters is included in the **flags**. • The **string** or **pattern** parameters contain multibyte characters.
b	Matches according to POSIX Basic Regular Expressions (BREs) matching rules.
c	Case-sensitive matching.
e	Matches according to the POSIX Extended Regular Expressions (EREs) matching rules. When neither **b** nor **e** is specified, if the Henry Spencer regex library is chosen, it follows the Advanced Regular Expressions (AREs), akin to Perl Compatible Regular Expressions (PCREs). If the Boost.Regex library is chosen, it adheres to PCREs.
i	Case-insensitive matching.
m	Newline-sensitive matching, synonymous with the **n** option.
n	Newline-sensitive matching. When this option is active, the newline character influences the matching of metacharacters (., ^, $, and [^).
p	Partially newline-sensitive matching. When this option is active, the newline character influences the matching of metacharacters (. and [^), partially compared to the **n** option.
q	Resets the regular expression to a double-quoted text string, treating all characters as literals.
s	Non-newline-sensitive matching.
t	Compact syntax (default). When this option is active, all characters are significant.
w	Inverse partially newline-sensitive matching. When this option is active, the newline character influences the matching of metacharacters (^ and $), inversely partially compared to the **n** option.
x	Extended syntax. Contrary to compact syntax, in extended syntax, whitespace characters in the regular expression are ignored. Whitespace characters include spaces, horizontal tabs, newlines, and any characters belonging to the space character set.

 Copyright © Huawei Technologies Co., Ltd.

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: Concatenates all but the first parameter with separators. The first parameter is used as the separator.

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 string **bytea** using the destination encoding specified by **dest_encoding**. **src_encoding** specifies the source encoding format. The **string** must be valid under this encoding.

Return type: bytea.

Example:

```
SELECT convert('text_in_utf8', 'UTF8', 'GBK');
          convert
-----------------------------
 \x746578745f696e5f75746638
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

NOTE

If the conversion rule from the source encoding format to the destination encoding format does not exist, the string is returned unchanged. For example, there is no conversion rule between GBK and LATIN1. Specific conversion rules can be obtained by querying the **pg_conversion** system catalog.

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
----------------------
 \x736f6d652074657874
(1 row)

db_latin1=# SELECT convert('some text', 'GBK', 'LATIN1');
      convert
----------------------
 \x736f6d652074657874
(1 row)
```

string [NOT] LIKE pattern [ESCAPE escape-character]

Description: Pattern matching function.

If the pattern contains neither percent signs nor underscores, it stands solely for itself, making LIKE behave like the equality operator. An underscore (_) in the pattern matches any single character, while a percent sign (%) matches no or multiple characters.

To match an underscore or percent sign literally, the respective character in the pattern must be preceded by an escape character. By default, the escape character is a backslash, but you may specify another using the **ESCAPE** clause. To match the escape character itself, use 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)
```

 Copyright © Huawei Technologies Co., Ltd.

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('ABC');
               md5
----------------------------------
 902fbdd2b1df0c4f70b4a5d23525e932
(1 row)
```

CONV(n, fromBase, toBase)

Description: Converts the given value or string into the target number system and outputs the result as a string. If the parameter contains NULL, NULL is returned. The value range is [-36, -2] and [2, 36].

Return type: text.

Example:

```
SELECT CONV(-1, 10, 16) as result;
      result
------------------
 FFFFFFFFFFFFFFFF
(1 row)
```

HEX(n)

Description: *n* can be of the int type or a string. Returns the hexadecimal string of *n*. If the parameter contains NULL, NULL is returned.

Return type: text.

Example:

```
SELECT HEX(255) as result;
 result
--------
   FF
(1 row)
SELECT HEX('abc') as result;
```

 Copyright © Huawei Technologies Co., Ltd.

```
 result
--------
 616263
(1 row)
```

UNHEX(n)

Description: Performs the reverse operation of **HEX(n)**. *n* can be of the int type or a string. Each pair of hexadecimal digits in a parameter is considered as a number and converted into the character represented by the number. If the parameter contains NULL, NULL is returned.

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 is 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 less than the second string according to the current classification order, **-1** is returned. Otherwise, **1** is returned. If the parameter contains NULL, NULL is 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 is returned.

Return type: text.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT BIN(16) as result;
 result
--------
 10000
(1 row)
```

11.2 Binary String Functions and Operators

SQL defines some binary string functions that use keywords rather than commas to separate arguments. Aura also provides the common syntax used for invoking functions.

octet_length(string)

Description: The number of bytes in a binary string.

Return type: integer.

Example:

```
postgres=# SELECT octet_length(E'jo\\000se'::bytea) AS RESULT;
 result
--------
      5
(1 row)
```

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

Description: Replaces a substring.

Return type: bytea

Example:

```
postgres=# 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: The position of a specific substring.

Return type: integer.

Example:

```
postgres=# SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea) AS RESULT;
 result
--------
      3
(1 row)
```

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

Description: Extracts a substring.

Return type: bytea

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT substring(E'Th\\000omas'::bytea from 2 for 3) AS RESULT;
  result
----------
 \x68006f
(1 row)
```

Extract time and get the hour value:

```
postgres=# SELECT substring('2022-07-18 24:38:15',12,2)AS RESULT;
 result
------------
 24
(1 row)
```

substring_index(string, delim, count)

Description: Performs a case-sensitive search for the delimiter and returns the substring preceding the *count*-th occurrence of the delimiter in the **string**. If **count** is negative, search backward from the end. If the parameter contains NULL, NULL is returned.

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**.

```
postgres=# SELECT SUBSTRING_INDEX('www.wWw.cloud.wWw.com', '.wWw.', 2) AS RESULT;
    result
---------------
 www.wWw.cloud
(1 row)
```

get_bit(string, offset)

Description: Extracts bits from a string.

Return type: integer.

Example:

```
postgres=# SELECT get_bit(E'Th\\000omas'::bytea, 45) AS RESULT;
 result
--------
      1
(1 row)
```

get_byte(string, offset)

Description: Extracts bytes from a string.

Return type: integer.

Example:

```
postgres=# SELECT get_byte(E'Th\\000omas'::bytea, 4) AS RESULT;
 result
--------
    109
(1 row)
```

set_bit(string,offset, newvalue)

Description: Sets bits in a string.

 Copyright © Huawei Technologies Co., Ltd.

Return type: bytea

Example:

```
postgres=# SELECT set_bit(E'Th\\000omas'::bytea, 45, 0) AS RESULT;
     result
------------------
 \x5468006f6d4173
(1 row)
```

set_byte(string,offset, newvalue)

Description: Sets bytes in a string.

Return type: bytea

Example:

```
postgres=# SELECT set_byte(E'Th\\000omas'::bytea, 4, 64) AS RESULT;
     result
------------------
 \x5468006f406173
(1 row)
```

11.3 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:

```
postgres=# SELECT B'10001' || B'011' AS RESULT;
  result
----------
 10001011
(1 row)
```

&

Description: AND operation between bit strings

For example:

```
postgres=# SELECT B'10001' & B'01101' AS RESULT;
 result
--------
 00001
(1 row)
```

|

Description: OR operation between bit strings

For example:

```
postgres=# SELECT B'10001' | B'01101' AS RESULT;
 result
```

 Copyright © Huawei Technologies Co., Ltd.

```
--------
 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:

```
postgres=# SELECT B'10001' # B'01101' AS RESULT;
 result
--------
 11100
(1 row)
```

~

Description: NOT operation between bit strings

For example:

```
postgres=# SELECT ~B'10001'AS RESULT;
 result
----------
 01110
(1 row)
```

<<

Description: Binary left shift.

For example:

```
postgres=# SELECT B'10001' << 3 AS RESULT;
 result
----------
 01000
(1 row)
```

>>

Description: binary right shift

For example:

```
postgres=# 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**.

11.4 Numeric Operation Functions and Operators

 Copyright © Huawei Technologies Co., Ltd.

11.4.1 Numeric Operators

+

Description: Addition

Example:

```
postgres=# SELECT 2+3 AS RESULT;
 result
--------
      5
(1 row)
```

-

Description: Subtraction

Example:

```
postgres=# SELECT 2-3 AS RESULT;
 result
--------
     -1
(1 row)
```

*****

Description: Multiplication

Example:

```
postgres=# SELECT 2*3 AS RESULT;
 result
--------
      6
(1 row)
```

/

Description: Division (Division operator does not round off.)

When the divisor is 0, the error message "division by zero" will be displayed.

Example:

```
postgres=# SELECT 4/2 AS RESULT;
 result
--------
      2
(1 row)
postgres=# SELECT 4/3 AS RESULT;
       result
------------------
 1.33333333333333
(1 row)
```

+/-

Description: Positive/Negative

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT -2 AS RESULT;
 result
--------
     -2
(1 row)
```

%

Description: Modulus (Remainder)

Note: An error is reported if the divisor is 0.

Example:

```
postgres=# SELECT 5%4 AS RESULT;
 result
--------
      1
(1 row)
```

@

Description: Absolute value.

Example:

```
postgres=# SELECT @ -5.0 AS RESULT;
 result
--------
    5.0
(1 row)
```

^

Description: XOR

In MySQL compatibility mode, it functions as an XOR.

Example:

```
postgres=# SELECT 2.0^3.0 AS RESULT;
       result
--------------------
 8.0000000000000000
(1 row)
```

|/

Description: Square root.

Example:

```
postgres=# SELECT |/ 25.0 AS RESULT;
 result
--------
      5
(1 row)
```

||/

Description: Cube root.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT ||/ 27.0 AS RESULT;
 result
--------
      3
(1 row)
```

!

Description: Factorial

Example:

```
postgres=# SELECT 5! AS RESULT;
 result
--------
    120
(1 row)
```

!!

Description: Factorial (prefix operator)

Example:

```
postgres=# SELECT !!5 AS RESULT;
 result
--------
    120
(1 row)
```

&

Description: Binary AND

Example:

```
postgres=# SELECT 91&15  AS RESULT;
 result
--------
     11
(1 row)
```

|

Description: Binary OR

Example:

```
postgres=# SELECT 32|3  AS RESULT;
 result
--------
     35
(1 row)
```

#

Description: Binary XOR

Example:

```
postgres=# SELECT 17#5  AS RESULT;
 result
--------
```

 Copyright © Huawei Technologies Co., Ltd.

```
    20
(1 row)
```

~

Description: Binary NOT

Example:

```
postgres=# SELECT ~1 AS RESULT;
 result
--------
    -2
(1 row)
```

<<

Description: Binary left shift

Example:

```
postgres=# SELECT 1<<4 AS RESULT;
 result
--------
    16
(1 row)
```

>>

Description: Binary right shift

Example:

```
postgres=# SELECT 8>>2 AS RESULT;
 result
--------
     2
(1 row)
```

11.4.2 Numeric Operation Functions

abs(x)

Description: Absolute value.

Return type: same as the input.

Example:

```
postgres=# SELECT abs(-17.4);
 abs
------
 17.4
(1 row)
```

acos(x)

Description: Arc cosine.

Return type: double precision.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT acos(-1);
       acos
------------------
 3.14159265358979
(1 row)
```

asin(x)

Description: Arc sine.

Return type: double precision.

Example:

```
postgres=# SELECT asin(0.5);
       asin
------------------
 0.523598775598299
(1 row)
```

atan(x)

Description: Arc tangent.

Return type: double precision.

Example:

```
postgres=# SELECT atan(1);
       atan
------------------
 0.785398163397448
(1 row)
```

atan2(y, x)

Description: Arc tangent of y/x.

Return type: double precision.

Example:

```
postgres=# SELECT atan2(2, 1);
      atan2
------------------
 1.10714871779409
(1 row)
```

bitand(integer, integer)

Description: Bitwise AND operation of two integers.

Return type: bigint.

Example:

```
postgres=# SELECT bitand(127, 63);
 bitand
--------
     63
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

cbrt(double precision)

Description: Cube root.

Return type: double precision.

Example:

```
postgres=# SELECT cbrt(27.0);
 cbrt
------
    3
(1 row)
```

ceil(double precision or numeric)

Description: Smallest integer not less than the parameter.

Return type: same as the input.

Example:

```
postgres=# SELECT ceil(-42.8);
 ceil
------
  -42
(1 row)
```

ceiling(double precision or numeric)

Description: Smallest integer not less than the parameter (alias for ceil).

Return type: same as the input.

Example:

```
postgres=# SELECT ceiling(-95.3);
 ceiling
---------
     -95
(1 row)
```

cos(x)

Description: Cosine.

Return type: double precision.

Example:

```
postgres=# SELECT cos(-3.1415927);
        cos
-------------------
 -0.999999999999999
(1 row)
```

cot(x)

Description: Cotangent.

Return type: double precision.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT cot(1);
       cot
------------------
 0.642092615934331
(1 row)
```

degrees(double precision)

Description: Converts radians to degrees.

Return type: double precision.

Example:

```
postgres=# SELECT degrees(0.5);
     degrees
------------------
 28.6478897565412
(1 row)
```

div(y numeric, x numeric)

Description: Integer quotient of y divided by x.

Return type: numeric.

Example:

```
postgres=# SELECT div(9,4);
 div
-----
   2
(1 row)
```

exp(double precision or numeric)

Description: Exponential.

Return type: same as the input.

Example:

```
postgres=# SELECT exp(1.0);
        exp
-------------------
 2.7182818284590452
(1 row)
```

floor(double precision or numeric)

Description: Largest integer not greater than the parameter.

Return type: same as the input.

Example:

```
postgres=# SELECT floor(-42.8);
 floor
-------
   -43
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

fq_iceberg_bucket(int, smallint/int/bigint)

Description: Calculates the bucket number for a specified integer. The first int parameter defines the total number of buckets.

Return type: int.

Example:

```
select fq_iceberg_bucket(13, 5211314);
 fq_iceberg_bucket
-------------------
                 5
(1 row)
```

fq_iceberg_bucket(int, numeric(p, s))

Description: Calculates the bucket number for a specified numeric data type. The first parameter defines the total number of buckets.

Return type: int.

Example:

```
select fq_iceberg_bucket(13, 1000.50::numeric(15,2));
 fq_iceberg_bucket
-------------------
                12
(1 row)
```

fq_iceberg_truncate(int W, smallint/int/bigint integer_value)

Description: Calculates the largest value that does not exceed **integer_value** and is a multiple of **W**.

Return type: Matches the data type of the second parameter.

Example:

```
select fq_iceberg_truncate (10, 19);
 fq_iceberg_truncate
---------------------
                  10
(1 row)
```

fq_iceberg_truncate(int W, numeric(p, s) num_value)

Description: Calculates the largest value that does not exceed **num_value**, where shifting the decimal point s places to the right results in a multiple of **W**.

Return type: numeric(p, s).

Example:

```
select fq_iceberg_truncate (50, 10.64::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.600
(1 row)
```

For more examples of fq_iceberg_truncate, see **Partition Expression: truncate**.

 Copyright © Huawei Technologies Co., Ltd.

radians(double precision)

Description: Converts degrees to radians.

Return type: double precision.

Example:

```
postgres=# SELECT radians(45.0);
     radians
------------------
 0.785398163397448
(1 row)
```

random()

Description: Random number ranging from 0.0 to 1.0.

Return type: double precision.

Example:

```
postgres=# SELECT random();
      random
------------------
 0.696101832669228
(1 row)
```

rand()

Description: Random number ranging from 0.0 to 1.0. This function is compatible with MySQL.

Return type: double precision.

Example:

```
postgres=# SELECT rand();
      rand
------------------
 0.930654047057033
(1 row)
```

ln(double precision or numeric)

Description: Natural logarithm.

Return type: same as the input.

Example:

```
postgres=# SELECT ln(2.0);
        ln
-------------------
 0.6931471805599453
(1 row)
```

log(double precision or numeric)

Description: common logarithm

Return type: same as the input.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
postgres=# SELECT log(100.0);
       log
--------------------
 4.6051701859880914
(1 row)
```

log(b numeric, x numeric)

Description: Logarithm with b as the base.

Return type: numeric.

Example:

```
postgres=# SELECT log(2.0, 64.0);
       log
--------------------
 6.0000000000000000
(1 row)
```

mod(x,y)

Description: Remainder (modulo) of x/y. If **x** is **0**, **0** is returned. If **y** is **0**, an error is reported.

Return type: same as the parameter type.

Example:

```
postgres=# SELECT mod(9,4);
 mod
-----
   1
(1 row)
postgres=# SELECT mod(9,0);
 ERROR:  Division by zero
```

pi()

Description: π constant.

Return type: double precision.

Example:

```
postgres=# SELECT pi();
       pi
------------------
 3.14159265358979
(1 row)
```

power(a double precision, b double precision)

Description: a raised to the power of b.

Return type: double precision.

Example:

```
postgres=# SELECT power(9.0, 3.0);
       power
----------------------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 729.0000000000000000
(1 row)
```

round(double precision or numeric)

Description: Closest integer to the input parameter.

Return type: same as the input.

Example:

```
postgres=# SELECT round(42.4);
 round
-------
    42
(1 row)

postgres=# SELECT round(42.6);
 round
-------
    43
(1 row)
```

NOTE

When calling the **round** function, numeric types will round towards zero, while (on most computers) real and double-precision types will result in the nearest even number.

round(v numeric, s int)

Description: Retains **s** decimal places, rounding the next digit.

Return type: numeric.

Example:

```
postgres=# SELECT round(42.4382, 2);
 round
-------
 42.44
(1 row)
```

setseed(double precision)

Description: Sets the seed for subsequent **random()** calls (from –1.0 to 1.0).

Return type: void.

Example:

```
postgres=# SELECT setseed(0.54823);
 setseed
---------

(1 row)
```

sign(double precision or numeric)

Description: Outputs the sign of this parameter.

Return type: **-1** indicates a negative number, **0** indicates 0, and **1** indicates a positive number.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT sign(-8.4);
 sign
------
   -1
(1 row)
```

sin(x)

Description: Sine.

Return type: double precision.

Example:

```
postgres=# SELECT sin(1.57079);
       sin
------------------
 0.999999999979986
(1 row)
```

sqrt(double precision or numeric)

Description: Square root.

Return type: same as the input.

Example:

```
postgres=# SELECT sqrt(2.0);
       sqrt
--------------------
 1.414213562373095
(1 row)
```

tan(x)

Description: Tangent.

Return type: double precision.

Example:

```
postgres=# SELECT tan(20);
       tan
------------------
 2.23716094422474
(1 row)
```

trunc(double precision or numeric)

Description: Truncates (takes the integer part).

Return type: same as the input.

Example:

```
postgres=# SELECT trunc(42.8);
 trunc
-------
    42
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

trunc(v numeric, s int)

Description: Truncates to **s** decimal places.

Return type: numeric.

Example:

```
postgres=# SELECT trunc(42.4382, 2);
 trunc
-------
 42.43
(1 row)
```

truncate(v numeric, s int)

Description: Truncates **v** to **s** decimal places. **v** can be of any precision, integer, or floating-point type. The return value type is the same as that of the input parameter **v**. When **s** is negative, the integer part is truncated.

Return type: numeric.

Example:

```
postgres=# SELECT truncate(42.4382, 2);
 truncate
-------
 42.4300
(1 row)
```

width_bucket(operand numeric, b1 numeric, b2 numeric, count int)

Description: Sets the minimum value, maximum value, and number of groups to construct a specified number of equally sized groups, returning the group number into which the specified field value falls. **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:

```
postgres=# 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 to construct a specified number of equally sized groups, returning the group number into which the specified field value falls. **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:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT width_bucket(5.35, 0.024, 10.06, 5);
 width_bucket
--------------
            3
(1 row)
```

11.5 Time and Date Processing Functions and Operators

11.5.1 Time and Date Operators

NOTICE

When using time and date operators, you should explicitly prefix the corresponding operands with type modifiers to ensure that the database parses the operands as intended, avoiding unexpected results.

For example, failing to specify the data type in the following query will cause an exception.

```
SELECT date '2001-10-01' - '7' AS RESULT;
```

Table 11-2 Time and date operators

Operator	Example

+	Add a date type parameter and an integer parameter to get the time after 7 days.
	`SELECT date '2001-09-28' + integer '7' AS RESULT;`
	`      result`
	`---------------------`
	` 2001-10-05`
	`(1 row)`

	Add a date type parameter and an interval parameter to get the time after 1 hour.
	`SELECT date '2001-09-28' + interval '1 hour' AS RESULT;`
	`      result`
	`---------------------`
	` 2001-09-28 01:00:00`
	`(1 row)`

 Copyright © Huawei Technologies Co., Ltd.

Operator	Example

	Add a date type parameter and an interval parameter to get the time span of 1 month.
	The date function adjusts the date range beyond the month to align with the last day of the respective month. It leaves the date range unchanged if it does not exceed the month.
	`SELECT date '2021-01-31' + interval '1 month' AS RESULT;`
	`result`
	`---`
	`2021-02-28 00:00:00`
	`(1 row)`
	`SELECT date '2021-02-28' + interval '1 month' AS RESULT;`
	`result`
	`---`
	`2021-03-28 00:00:00`
	`(1 row)`

	Add a date type parameter and a time type parameter to get a specific date and time result.
	`SELECT date '2001-09-28' + time '03:00' AS RESULT;`
	`result`
	`---`
	`2001-09-28 03:00:00`
	`(1 row)`

	Add interval parameters to get the sum of two time spans.
	`SELECT interval '1 day' + interval '1 hour' AS RESULT;`
	`result`
	`---`
	`1 day 01:00:00`
	`(1 row)`

	Add a timestamp type parameter and an interval parameter to get the time after 23 hours.
	`SELECT timestamp '2001-09-28 01:00' + interval '23 hours' AS RESULT;`
	`result`
	`---`
	`2001-09-29 00:00:00`
	`(1 row)`

	Add a time type parameter and an interval parameter to get the time after 3 hours.
	`SELECT time '01:00' + interval '3 hours' AS RESULT;`
	`result`
	`---`
	`04:00:00`
	`(1 row)`

-	Subtract date type parameters to get the time difference between two dates.
	`SELECT date '2001-10-01' - date '2001-09-28' AS RESULT;`
	`result`
	`---`
	`3`
	`(1 row)`

 Copyright © Huawei Technologies Co., Ltd.

Operator	Example

	Subtract an integer parameter from a date type parameter and return a timestamp type to get the time difference between them.
	`SELECT date '2001-10-01' - integer '7' AS RESULT;`
	`result`
	`---`
	`2001-09-24`
	`(1 row)`

	Subtract an interval parameter from a date type parameter to get the date and time difference.
	`SELECT date '2001-09-28' - interval '1 hour' AS RESULT;`
	`result`
	`---`
	`2001-09-27 23:00:00`
	`(1 row)`

	Subtract time type parameters to get the time difference between them.
	`SELECT time '05:00' - time '03:00' AS RESULT;`
	`result`
	`---`
	`02:00:00`
	`(1 row)`

	Subtract an interval parameter from a time type parameter to get the time difference between them.
	`SELECT time '05:00' - interval '2 hours' AS RESULT;`
	`result`
	`---`
	`03:00:00`
	`(1 row)`

	Subtract an interval from a timestamp to get the date and time difference between them.
	`SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT;`
	`result`
	`---`
	`2001-09-28 00:00:00`
	`(1 row)`

	Subtract interval parameters to get the time difference between them.
	`SELECT interval '1 day' - interval '1 hour' AS RESULT;`
	`result`
	`---`
	`23:00:00`
	`(1 row)`

	Subtract timestamp type parameters to get the date and time difference between them.
	`SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT;`
	`result`
	`---`
	`1 day 15:00:00`
	`(1 row)`

	Get the previous day from the current date.
	`select now() - interval '1 day'AS RESULT;`
	`result`
	`---`
	`2022-08-08 01:46:15.555406+00`
	`(1 row)`

 Copyright © Huawei Technologies Co., Ltd.

Operator	Example
*	Multiply a time span by a quantity. SELECT 900 * interval '1 second' AS RESULT; result ---------- 00:15:00 (1 row)
	SELECT 21 * interval '1 day' AS RESULT; result --------- 21 days (1 row)
	SELECT double precision '3.5' * interval '1 hour' AS RESULT; result ---------- 03:30:00 (1 row)
/	Divide a time span by a quantity to get a segment within a period of time. SELECT interval '1 hour' / double precision '1.5' AS RESULT; result ---------- 00:40:00 (1 row)

11.5.2 Time and Date Functions

age(timestamp, timestamp)

Description: Subtracts two parameters and returns the year, month, and day. If the subtraction result is negative, the function also returns a negative value.

Return type: interval.

Example:

```
postgres=# SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
          age
--------------------------
 43 years 9 mons 27 days
(1 row)
```

age(timestamp)

Description: Subtracts the current time from the parameter value.

Return type: interval.

Example:

```
postgres=# SELECT age(timestamp '1957-06-13');
          age
--------------------------
 60 years 2 mons 18 days
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

adddate(date, interval | int)

Description: Returns the result of a given datetime plus the time interval of a specified unit. The default unit (when the second parameter is an integer) is days.

Return type: timestamp.

Example:

When the input parameter is of the text type:

```
postgres=# select adddate('2020-11-13', 10);
  adddate
------------
 2020-11-23
(1 row)

postgres=# select adddate('2020-11-13', interval '1' month);
  adddate
------------
 2020-12-13
(1 row)

postgres=# select adddate('2020-11-13 12:15:16', interval '1' month);
       adddate
---------------------
 2020-12-13 12:15:16
(1 row)


postgres=# 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:

```
postgres=# select adddate(current_date, 10);
  adddate
------------
 2021-09-24
(1 row)

postgres=# 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 date/time plus a specified interval.

Return type: same as the type of the first input parameter.

Example:

```
postgres=# select addtime('2020-11-13 01:01:01', '23:59:59');
       addtime
---------------------
 2020-11-14 01:01:00
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

subdate(date, interval | int)

Description: Returns the result of subtracting a time interval of a specified unit from a given date and time. The default unit (when the second parameter is an integer) is days.

Return type: timestamp.

Example:

When the input parameter is of the text type:

```
postgres=# select subdate('2020-11-13', 10);
  subdate
------------
 2020-11-03
(1 row)

postgres=# select subdate('2020-11-13', interval '2' month);
  subdate
------------
 2020-09-13
(1 row)

postgres=# select subdate('2020-11-13 12:15:16', interval '1' month);
       subdate
---------------------
 2020-10-13 12:15:16
(1 row)

postgres=# 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:

```
postgres=# select subdate(current_date, 10);
  subdate
------------
 2021-09-05
(1 row)

postgres=# 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 date/time minus a specified interval.

Return type: same as the type of the first input parameter.

Example:

```
postgres=# select subtime('2020-11-13 01:01:01', '23:59:59');
       subtime
---------------------
 2020-11-12 01:01:02
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

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 time interval (seconds can include decimals) in the unit of field to a datetime expression. If the value is negative, the corresponding interval is subtracted from the given datetime expression. The **field** can be **year**, **month**, **quarter**, **day**, **week**, **hour**, **minute**, **second**, and **microsecond**.

Return type: timestamp.

Example:

```
postgres=# select timestampadd(year, 1, timestamp '2020-2-29');
    timestampadd
---------------------
 2021-02-28 00:00:00
(1 row)

postgres=# 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 **timestamp2** from **timestamp1** from two date parameters and returns a value in the unit of **field**. If the result is negative, the function returns a negative value. The **field** can be **year**, **month**, **quarter**, **day**, **week**, **hour**, **minute**, **second**, and **microsecond**.

Return type: bigint.

Example:

```
postgres=# 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 two date parameters. If the result is negative, the function returns a negative value. Both input parameters must be of the same type.

Return type: time.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
postgres=# SELECT timediff('2022-7-5 1:1:1', '2021-7-5 1:1:1');
   timediff
---------------
  8760:00:00
(1 row)
```

clock_timestamp()

Description: Current timestamp of the real-time clock.

Return type: timestamp with time zone.

Example:

```
postgres=# SELECT clock_timestamp();
        clock_timestamp
-------------------------------
 2017-09-01 16:57:36.636205+08
(1 row)
```

current_date

Description: Current date.

Return type: date.

Example:

```
postgres=# SELECT current_date;
    current_date
------------
 2017-09-01
(1 row)
```

curdate()

Description: Current date. This function is compatible with MySQL.

Return type: date.

Example:

```
postgres=# SELECT curdate();
   curdate
------------
 2022-09-19
(1 row)
```

current_time

Description: Current time.

Return type: time with time zone.

Example:

```
postgres=# SELECT current_time;
       current_time
--------------------
 16:48:27.59887
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

curtime([fsp])

Description: Current time. **fsp** is an optional parameter of integer type, specifying the fractional second precision of the return value.

Return type: time with time zone.

Example:

```
postgres=# SELECT curtime();
      curtime
--------------------
 16:46:40.759849
(1 row)
postgres=# SELECT curtime(2);
      curtime
--------------------
 16:47:13.9
(1 row)
```

current_timestamp

Description: Current date and time.

Return type: timestamp with time zone.

Example:

```
postgres=# SELECT current_timestamp;
      current_timestamp
-------------------------------
 2025-01-22 16:45:41.935586
(1 row)
```

convert_tz(timestamp, from_tz, to_tz)

Description: Converts the timestamp value from the time zone specified by **from_tz** (text) to the time zone specified by **to_tz** (text) and returns the converted time.

Return type: timestamp without time zone.

Example:

```
postgres=# SELECT convert_tz('2018-12-25 13:25:00', '+02:00', '+08:00');
   convert_tz
--------------------
 2018-12-25 19:25:00
(1 row)
postgres=# 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 the given dates.

Return type: integer.

Example:

```
postgres=# select datediff(date '2020-11-13', date '2012-10-16');
 datediff
```

 Copyright © Huawei Technologies Co., Ltd.

```
----------
    2950
(1 row)
```

date_part(text, timestamp)

Description: Obtains the precision specified by the **text** parameter. 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: Retrieves the precision specified by **text**. If the value exceeds 12, it takes the modulus with 12. 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 to the precision specified by **text**.

Return type: timestamp.

Example:

```
postgres=# SELECT date_trunc('hour', timestamp  '2001-02-16 20:38:40');
     date_trunc
---------------------
 2001-02-16 20:00:00
(1 row)

-- Get the last day of last year.
postgres=# SELECT date_trunc('day', date_trunc('year',CURRENT_DATE)+ '-1');
       date_trunc
-------------------------
 2022-12-31 00:00:00+08
(1 row)

-- Get the first day of this year.
SELECT date_trunc('year',CURRENT_DATE);
       date_trunc
-------------------------
 2023-01-01 00:00:00+08
(1 row)

-- Get the first day of last year.
SELECT date_trunc('year',now() + '-1 year');
       date_trunc
```

 Copyright © Huawei Technologies Co., Ltd.

```
------------------------
 2022-01-01 00:00:00+08
(1 row)
```

extract(field from timestamp)

Description: Retrieves the value of the specified precision for the **field**. Valid **field** values can be found in **11.5.3 EXTRACT**.

Return type: double precision.

Example:

```
postgres=# SELECT extract(hour from timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

extract(field from interval)

Description: Retrieves the value of the specified precision for the **field**. If the value exceeds 12, it takes the modulus with 12. Valid **field** values can be found in **11.5.3 EXTRACT**.

Return type: double precision.

Example:

```
postgres=# SELECT extract(month from interval '2 years 3 months');
 date_part
-----------
         3
(1 row)
```

day(date)

Description: Retrieves the number of days in the month for the given date, which is identical to the **dayofmonth** function.

Range: 1 to 31.

Return type: integer.

Example:

```
postgres=# select day('2020-06-28');
 day
-----
  28
(1 row)
```

dayofmonth(date)

Description: Retrieves the number of days in the month for the given **date**.

Range: 1 to 31.

Return type: integer.

Example:

```
postgres=# select dayofmonth('2020-06-28');
 dayofmonth
```

 Copyright © Huawei Technologies Co., Ltd.

```
------------
        28
(1 row)
```

dayofweek(date)

Description: Returns the weekday index for the given **date**, with Sunday being the first day of the week.

Range: 1 to 7.

Return type: integer.

Example:

```
postgres=# select dayofweek('2020-11-22');
 dayofweek
-----------
         1
(1 row)
```

dayofyear(date)

Description: Returns the number of days into the year for the given **date**.

Range: 1 to 366.

Return type: integer.

Example:

```
postgres=# select dayofyear('2020-02-29');
 dayofyear
-----------
        60
(1 row)
```

fq_iceberg_year(date/timestamp)

Description: Returns the year value of a given date relative to 1970.

Return type: integer.

Example:

```
select fq_iceberg_year('2025-10-10');
 fq_iceberg_year
-----------------
              55
(1 row)
```

fq_iceberg_month(date/timestamp)

Description: Returns the month value of a given date relative to January 1, 1970.

Return type: integer.

Example:

```
select fq_iceberg_month('2025-10-10');
 fq_iceberg_month
-----------------
             669
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

fq_iceberg_day(date/timestamp)

Description: Returns the day value of a given date relative to January 1, 1970.

Return type: integer.

Example:

```
select fq_iceberg_day('2025-10-10');
 fq_iceberg_day
----------------
          20371
(1 row)
```

fq_iceberg_hour(timestamp)

Description: Returns the hour value of a given timestamp relative to January 1, 1970, at 00:00:00.

Return type: integer.

Example:

```
select  fq_iceberg_hour ('2025-10-10 10:10:10.000') ;
 fq_iceberg_hour
-----------------
          488914
(1 row)
```

For more information about fq_iceberg_year, fq_iceberg_month, fq_iceberg_day, and fq_iceberg_hour, see **Supported Partition Expressions**.

fq_iceberg_bucket(int, date/timestamp)

Description: Calculates the bucket number for a specified time-based number. The first int parameter defines the total number of buckets.

Return type: int.

Example:

```
select fq_iceberg_bucket(13, '2025-10-11'::date);
 fq_iceberg_bucket
-------------------
                 8
(1 row)

select fq_iceberg_bucket(13, '2025-10-11 11:11:11.222'::timestamp);
 fq_iceberg_bucket
-------------------
                 1
(1 row)
```

hour(timestamp with time zone)

Description: Retrieves the hour value from the time.

Return type: integer.

Example:

```
postgres=# SELECT hour(timestamptz '2018-12-13 12:11:15+06');
 hour
------
```

 Copyright © Huawei Technologies Co., Ltd.

```
  14
(1 row)
```

isfinite(date)

Description: Checks if the date is finite.

Return type: boolean.

Example:

```
postgres=# SELECT isfinite(date '2001-02-16');
 isfinite
----------
 t
(1 row)
postgres=# SELECT isfinite(date 'infinity');
 isfinite
----------
 f
(1 row)
```

isfinite(timestamp)

Description: Determines if the time is finite.

Return type: boolean.

Example:

```
postgres=# SELECT isfinite(timestamp '2001-02-16 21:28:30');
 isfinite
----------
 t
(1 row)
postgres=# SELECT isfinite(timestamp 'infinity');
 isfinite
----------
 f
(1 row)
```

isfinite(interval)

Description: Verifies if the interval is finite.

Return type: boolean.

Example:

```
postgres=# SELECT isfinite(interval '4 hours');
 isfinite
----------
 t
(1 row)
```

justify_days(interval)

Description: Converts the time interval into months, assuming each month has 30 days.

Return type: interval.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT justify_days(interval '35 days');
 justify_days
--------------
 1 mon 5 days
(1 row)
```

justify_hours(interval)

Description: Converts the time interval into days, assuming each day has 24 hours.

Return type: interval.

Example:

```
postgres=# SELECT JUSTIFY_HOURS(INTERVAL '27 HOURS');
 justify_hours
---------------
 1 day 03:00:00
(1 row)
```

justify_interval(interval)

Description: Adjusts **interval** using both **justify_days** and **justify_hours**.

Return type: interval.

Example:

```
postgres=# SELECT JUSTIFY_INTERVAL(INTERVAL '1 MON -1 HOUR');
 justify_interval
------------------
 29 days 23:00:00
(1 row)
```

localtime

Description: Current local time.

Return type: timestamp.

Example:

```
postgres=# SELECT localtime AS RESULT;
     result
----------------
 2025-01-17 09:48:08.056091
(1 row)
```

localtimestamp

Description: Current local date and time.

Return type: timestamp.

Example:

```
postgres=# SELECT localtimestamp;
         localtimestamp
----------------------------
 2017-09-01 17:03:30.781902
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

makedate(year, dayofyear)

Description: Constructs a date value based on the provided year and day-of-year.

Return type: date.

Example:

```
postgres=# select makedate(2020, 60);
  makedate
------------
 2020-02-29
(1 row)
```

maketime(hour, minute, second)

Description: Creates a time value from the given hour, minute, and second. The value of the time type in Aura 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:

```
postgres=# select maketime(12, 15, 30.12);
  maketime
-------------
 12:15:30.12
(1 row)
```

microsecond(timestamp with time zone)

Description: Retrieves the microsecond value from the time.

Return type: integer.

Example:

```
postgres=# SELECT microsecond(timestamptz '2018-12-13 12:11:15.123634+06');
 microsecond
-------------
      123634
(1 row)
```

minute(timestamp with time zone)

Description: Obtains the minute value in the time.

Return type: integer.

Example:

```
postgres=# SELECT minute(timestamptz '2018-12-13 12:11:15+06');
 minute
--------
     11
(1 row)
```

month(date)

Description: Returns the month of a given datetime.

Return type: integer.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
postgres=# select month('2020-11-30');
 month
-------
    11
(1 row)
```

now([fsp])

Description: Start date and time of the current transaction. The parameter determines the microsecond output precision. The default value is **6**.

Return type: timestamp with time zone.

Example:

```
postgres=# SELECT now();
              now
-------------------------------
 2017-09-01 17:03:42.549426+08
(1 row)
postgres=# SELECT now(3);
             now
-----------------------------
 2021-09-08 10:59:00.427+08
(1 row)
```

numtodsinterval(num, interval_unit)

Description: Converts a number to the interval type. The value of **num** is a number of the numeric type, and the value of **interval_unit** is a string in the fixed format ('DAY' | 'HOUR' | 'MINUTE' | 'SECOND').

Example:

```
postgres=# SELECT numtodsinterval(100, 'HOUR');
 numtodsinterval
-----------------
 100:00:00
(1 row)
```

pg_sleep(seconds)

Description: Specifies the server thread execution delay, in seconds.

Return type: void.

Example:

```
postgres=# SELECT pg_sleep(10);
 pg_sleep
----------

(1 row)
```

period_add(P, N)

Description: Returns the date of the given period plus *N* months.

Return type: integer.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# 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.

```
postgres=# select period_diff(200802, 200703);
 period_diff
-------------
          11
(1 row)
```

quarter(date)

Description: Obtains the quarter the date belongs to.

Return type: integer.

Example:

```
postgres=# SELECT quarter(date '2018-12-13');
 quarter
---------
       4
(1 row)
```

second(timestamp with time zone)

Description: Obtains the second value of the time.

Return type: integer.

Example:

```
postgres=# SELECT second(timestamptz '2018-12-13 12:11:15+06');
 second
--------
     15
(1 row)
```

statement_timestamp()

Description: Current date and time.

Return type: timestamp with time zone.

Example:

```
postgres=# SELECT statement_timestamp();
      statement_timestamp
-------------------------------
 2017-09-01 17:04:39.119267+08
(1 row)
```

sysdate

Description: Current date and time.

 Copyright © Huawei Technologies Co., Ltd.

Return type: timestamp.

Example:

```
postgres=# SELECT sysdate;
      sysdate
--------------------
 2017-09-01 17:04:49
(1 row)
```

timeofday()

Description: Current date and time (like **clock_timestamp**, but **text** is returned).

Return type: text.

Example:

```
postgres=# SELECT timeofday();
              timeofday
--------------------------------------
 Fri Sep 01 17:05:01.167506 2017 CST
(1 row)
```

from_unixtime(unix_timestamp[,format])

Description: Converts the Unix timestamp to the date and time type when the format string is set to the default value. If the format string is specified, convert the Unix timestamp to a string of the specified format for output.

Return type: timestamp (default format string) or text (specified by the format string).

Example:

```
postgres=# SELECT from_unixtime(875996580);
   from_unixtime
--------------------
 1997-10-04 20:23:00
(1 row)
postgres=# 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: Obtains the number of seconds from 1970-01-01 00:00:00 UTC to the input parameter time. If there is no input parameter, the current time is used.

Return type: bigint (no input parameter) or numeric (with input parameters).

Example:

```
postgres=# SELECT unix_timestamp();
 unix_timestamp
----------------
     1693906219
(1 row)
postgres=# SELECT unix_timestamp('2018-09-08 12:11:13+06');
 unix_timestamp
----------------
     1536387073.000000
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

add_months(d,n)

Description: Calculates the time point **d** plus *n* months.

Return type: timestamp.

Example:

```
postgres=# SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11);
    add_months
---------------------
 2018-04-29
(1 row)
```

last_day(d)

Description: Calculates the time of the last day of the month at time point **d**.

The return value type is date.

Example:

```
postgres=# 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: Calculates the time of the next week y starting from time x.

The return value type is date.

Example:

```
postgres=# 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:

```
postgres=# select from_days(730669);
 from_days
------------
 2000-07-03
(1 row)
```

to_days(timestamp)

Description: Returns the number of days from year 0 to the input parameter date.

Return type: integer.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
postgres=# SELECT to_days(timestamp '2008-10-07');
 to_days
---------
  733687
(1 row)
```

11.5.3 EXTRACT

EXTRACT(*field* **FROM** *source***)**

The **extract** function retrieves subfields from date or time values, such as year and hour. The **source** must be a value expression of type **timestamp**, **time**, or **interval**. (An expression of type **date** is cast to **timestamp** and thus can also be used.) **field** is an identifier or string specifying the field to be retrieved from the source data. The **extract** function returns a value of type **double precision**. The possible values for **field** are as follows:

century

Century.

The first century begins at 0001-01-01 00:00:00 AD. This definition applies to all countries using the Gregorian calendar. There is no 0th century. It goes directly from the 1st century BC to the 1st century AD.

Example:

```
postgres=# SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
 date_part
-----------
        20
(1 row)
```

day

- If **source** is **timestamp**, it indicates the day of the month (1–31).

  ```
  postgres=# SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
   date_part
  -----------
         16
  (1 row)
  ```

- If **source** is **interval**, it indicates the number of days.

  ```
  postgres=# SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
   date_part
  -----------
         40
  (1 row)
  ```

decade

Year divided by 10.

```
postgres=# SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
       200
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

dow

Day of the week, ranging from Sunday (**0**) to Saturday (**6**).

```
postgres=# SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         5
(1 row)
```

doy

Day of the year (1 to 365 or 366).

```
postgres=# SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        47
(1 row)
```

epoch

- If **source** is **timestamp with time zone**, it indicates the number of seconds since 1970-01-01 00:00:00-00 UTC (result could be negative).

  If **source** is **date** and **timestamp**, it indicates the number of seconds since 1970-01-01 00:00:00-00 local time.

  If **source** is **interval**, it indicates the total number of seconds in the interval.

  ```
  postgres=# SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
  20:38:40.12-08');
    date_part
  --------------
   982384720.12
  (1 row)
  postgres=# SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
   date_part
  -----------
      442800
  (1 row)
  ```

- Method to convert an epoch value to a timestamp.

  ```
  postgres=# 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).

```
postgres=# 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.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

This is identical to **dow** except for Sunday.

```
postgres=# SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
 date_part
-----------
         7
(1 row)
```

isoyear

ISO 8601 standard year in the date (not applicable to intervals).

Every week starting on Monday containing January 4 falls within the ISO year, so the ISO year in early January or late December might differ from the Gregorian year. See the subsequent **week** description for more details.

```
postgres=# SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
 date_part
-----------
      2005
(1 row)
postgres=# SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
 date_part
-----------
      2006
(1 row)
```

microseconds

Seconds column (including fractional parts) multiplied by 1,000,000.

```
postgres=# SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
 date_part
-----------
  28500000
(1 row)
```

millennium

Millennium.

Years in the 20th century (19xx) fall within the second millennium. The third millennium began at midnight on January 1, 2001.

```
postgres=# SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         3
(1 row)
```

milliseconds

Seconds column (including fractional parts) multiplied by 1,000. Note that it includes whole seconds.

```
postgres=# SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
 date_part
-----------
     28500
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

minute

Minutes column (0–59).

```
postgres=# SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        38
(1 row)
```

month

If **source** is **timestamp**, it indicates the month of the year (1–12).

```
postgres=# SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         2
(1 row)
```

If **source** is **interval**, it indicates the number of months, then modulo 12 (0–11).

```
postgres=# SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
 date_part
-----------
         1
(1 row)
```

quarter

The quarter of the year in which the day falls (1–4).

```
postgres=# SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         1
(1 row)
```

second

Seconds column, including fractional parts (0–59).

```
postgres=# SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
 date_part
-----------
      28.5
(1 row)
```

timezone

Timezone offset from UTC in seconds. Positive values correspond to timezones east of UTC, negative values to those west of UTC.

```
SELECT EXTRACT(timezone FROM TIMETZ '17:12:28');
 date_part
-----------
   28800
(1 row)
```

timezone_hour

Hours portion of the timezone offset.

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT EXTRACT(timezone_hour FROM TIMETZ '17:12:28');
 date_part
-----------
         8
(1 row)
```

timezone_minute

Minutes portion of the timezone offset.

```
SELECT EXTRACT(timezone_minute FROM TIMETZ '17:12:28');
 date_part
-----------
         0
(1 row)
```

week

The week number of the year in which the day falls. ISO 8601 defines the first week of the year as containing January 4 (ISO-8601 weeks start on Monday). In other words, the first Thursday of the year falls in the first week.

Under ISO definitions, the first few days of January might belong to the 52nd or 53rd week of the previous year, and the last few days of December might belong to the first week of the following year. For example, 2005-01-01 was the 53rd week of 2004, while 2006-01-01 was the 52nd week of 2005, and 2012-12-31 was the first week of 2013. You are advised to use the **isoyear** column along with **week** to achieve consistent results.

```
postgres=# SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         7
(1 row)
```

year

Year column.

```
postgres=# SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
      2001
(1 row)
```

11.5.4 date_part

date_part

The **date_part** function is modeled on the traditional Ingres function (which is equivalent to the SQL standard function **EXTRACT**).

date_part('*field*', *source*)

Here, the **field** must be a string instead of a name. The valid field names are the same as those for **extract**. For details, see **11.5.3 EXTRACT**.

Example:

```
postgres=# SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
 date_part
```

 Copyright © Huawei Technologies Co., Ltd.

```
-----------
        16
(1 row)
postgres=# SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
 date_part
-----------
         4
(1 row)
```

11.5.5 date_format

date_format(timestamp, fmt)

The **date_format** function converts a date into a string in the format specified by **fmt**.

Example:

```
postgres=# SELECT date_format('2009-10-04 22:23:00', '%M %D %W');
    date_format
--------------------
 October 4th Sunday
(1 row)
postgres=# 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)
postgres=# SELECT date_format('2021-02-20 18:10:15', '%r-%T');
     date_format
----------------------
 06:10:15 PM-18:10:15
(1 row)
```

Table 11-3 describes the format types that can be used to format date parameters for output. These format types are applicable to the functions **date_format**, **time_format**, **str_to_date**, **str_to_time**, and **from_unixtime**.

Table 11-3 Output formats supported by date_format

Format	Description	Value
%a	Abbreviated week name	Sun...Sat
%b	Abbreviated month name	Jan...Dec
%c	Month	1...12
%D	Day of the month with English suffix	0th, 1st, 2nd, 3rd, ...
%d	Day of a month, two digits	01...31
%e	Day of a month	1...31
%f	Microseconds	000000...999999
%H	Hour, in 24-hour format	00...23
%h	Hour, in 12-hour format	01...12

 Copyright © Huawei Technologies Co., Ltd.

Format	Description	Value
%I	Hour, in 12-hour format, same as **%h**	01...12
%i	Minutes	00...59
%j	Day of 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	Full month name	January...December
%m	Month, two digits	01...12
%p	AM or PM	AM PM
%r	Time, in 12-hour format	hh:mm:ss AM/PM
%S	Seconds	00...59
%s	Seconds, same as **%S**	00...59
%T	Time, in 24-hour format	hh:mm:ss
%U	Week (00–53), Sunday is the first day of the week	00...53
%u	Week (00–53), Monday is the first day of the week	00...53
%V	Week (01–53), Sunday is the first day of the week, used with **%X**	01...53
%v	Week (01–53), Monday is the first day of the week, used with **%x**	01...53
%W	Full weekday name	Sunday...Saturday
%w	Day of the week, where Sunday is **0**	0...6
%X	Year where Sunday is the first day of the week, 4 digits, used with **%V**	-
%x	Year where Monday is the first day of the week, 4 digits, used with **%v**	-
%Y	Year, 4 digits	-
%y	Year, 2 digits	-

 Copyright © Huawei Technologies Co., Ltd.

Format	Description	Value
%%	Literal '%'	%
%*x*	'x', any character not listed above	x

NOTICE

Among the output formats supported by **date_format**, **%U**, **%u**, **%V**, **%v**, **%X**, and **%x** are not currently supported.

11.5.6 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 the format specifiers of hour, minute, second, and microsecond. If the format string contains other specifiers, **NULL** or **0** is returned.

Return type: text.

Example:

```
postgres=# SELECT time_format('2009-10-04 22:23:00', '%M %D %W');
   time_format
--------------------

(1 row)
postgres=# 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)
postgres=# 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, %I, %i, %k, %l, %p, %r, %S, %s, and %T). Date-related formats are not supported. Other cases are treated as ordinary characters.

str_to_date(str, format)

Description: Converts a string (str) in the date/time format to a value of the date type according to the provided display format (format).

Return type: timestamp.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
postgres=# SELECT str_to_date('01,5,2021','%d,%m,%Y');
     str_to_date
---------------------
 2021-05-01 00:00:00
(1 row)
postgres=# 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)
```

Refer to **Table 11-3** for the format types applicable to the formatted input of **str_to_date**. Here, only date format and date + time format inputs are supported. For scenarios involving only time format inputs, use **str_to_time**.

str_to_time(str, format)

Description: Converts a string (str) in the time format to a value of the time type according to the provided display format (format).

Return type: time.

Example:

```
postgres=# SELECT str_to_time('09:30:17','%h:%i:%s');
 str_to_time
-------------
 09:30:17
(1 row)
```

Refer to **Table 11-3** for the format types applicable to the formatted input of **str_to_time**. Here, only time format inputs are supported. For scenarios involving date format and date + time format inputs, use **str_to_date**.

week(date[, mode])

Description: Returns the number of weeks in the year of the specified date and time based on the mode. The default mode is **0**.

Return type: integer.

Table 11-4 Working principle of mode in the week function

Mode	First Day of the Week	Week Range	Rule for Determining the First Week
0	Sunday	0–53	Week containing the first Sunday after New Year's Day
1	Monday	0–53	Week with four or more days after New Year's Day
2	Sunday	1-53	Week containing the first Sunday after New Year's Day
3	Monday	1-53	Week with four or more days after New Year's Day

 Copyright © Huawei Technologies Co., Ltd.

Mode	First Day of the Week	Week Range	Rule for Determining the First Week
4	Sunday	0–53	Week with four or more days after New Year's Day
5	Monday	0–53	Week containing the first Monday after New Year's Day
6	Sunday	1-53	Week with four or more days after New Year's Day
7	Monday	1-53	Week containing the first Monday after New Year's Day

Example:

```
postgres=# select week('2018-01-01');
 week
------
    0
(1 row)

postgres=# select week('2018-01-01', 0);
 week
------
    0
(1 row)

postgres=# select week('2020-12-31', 1);
 week
------
   53
(1 row)

postgres=# select week('2020-12-31', 5);
 week
------
   52
(1 row)
```

weekday(date)

Description: Returns the weekday index corresponding to the given date, with Monday as the start of the week.

Range: 0 to 6.

Return type: integer.

Example:

```
postgres=# select weekday('2020-11-06');
 weekday
---------
       4
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

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:

```
postgres=# select weekofyear('2020-12-30');
 weekofyear
------------
         53
(1 row)
```

year(date)

Description: Retrieves the year of the **date**.

Return type: integer.

Example:

```
postgres=# 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 corresponding to the given date. The number of weeks ranges from [1, 53].

Return type: integer.

Example:

```
postgres=# select yearweek('2019-12-31');
 yearweek
----------
   201952
(1 row)

postgres=# select yearweek('2019-1-1');
 yearweek
----------
   201852
(1 row)
```

11.6 Array Functions and Operators

11.6.1 Array Operators

Array comparison uses the default B-tree comparison function to compare all elements one by one. Elements of multi-dimensional arrays are accessed in row-major order. If the contents of two arrays are identical but their dimensionalities differ, the primary determinant of the sorting order is the dimensionality.

 Copyright © Huawei Technologies Co., Ltd.

=

Description: Determines whether two arrays are equal.

Example:

```
postgres=# SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT;
 result
--------
 t
(1 row)
```

<>

Description: Determines whether two arrays are not equal.

Example:

```
postgres=# SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<

Description: Determines whether one array is less than another array.

Example:

```
postgres=# SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

>

Description: Determines whether one array is greater than another array.

Example:

```
postgres=# SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<=

Description: Determines whether one array is less than or equal to another array.

Example:

```
postgres=# SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;
 result
--------
 t
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

>=

Description: Determines whether one array is greater than or equal to another array.

Example:

```
postgres=# SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;
 result
--------
 t
(1 row)
```

@>

Description: Determines whether one array contains another array.

Example:

```
postgres=# SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;
 result
--------
 t
(1 row)
```

<@

Description: Determines whether one array is contained within another array.

Example:

```
postgres=# SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;
 result
--------
 t
(1 row)
```

&&

Description: Determines whether one array overlaps with another array (shares common elements).

Example:

```
postgres=# SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;
 result
--------
 t
(1 row)
```

||

- Description: Concatenates arrays.

  Example:

  ```
  postgres=# SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;
     result
  ---------------
   {1,2,3,4,5,6}
  (1 row)
  postgres=# SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;
            result
  ---------------------------
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
 {{1,2,3},{4,5,6},{7,8,9}}
(1 row)
```

- Description: Concatenates elements with an array.

  Example:

  ```
  postgres=# SELECT 3 || ARRAY[4,5,6] AS RESULT;
    result
  -----------
   {3,4,5,6}
  (1 row)
  ```

- Description: Concatenates arrays with elements.

  Example:

  ```
  postgres=# SELECT ARRAY[4,5,6] || 7 AS RESULT;
    result
  -----------
   {4,5,6,7}
  (1 row)
  ```

11.6.2 Array Functions

array_append(anyarray, anyelement)

Description: Appends an element to the end of an array and supports only one-dimensional arrays.

Return type: anyarray.

Example:

```
postgres=# 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 supports only one-dimensional arrays.

Return type: anyarray.

Example:

```
postgres=# 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:

```
postgres=# SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;
   result
-------------
```

 Copyright © Huawei Technologies Co., Ltd.

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

postgres=# 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 an array.

Return type: int.

Example:

```
postgres=# SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
 result
--------
      2
(1 row)
```

array_dims(anyarray)

Description: Returns a textual representation of the array's dimensions.

Return type: text.

Example:

```
postgres=# 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 specified array dimension.

Return type: int.

Example:

```
postgres=# SELECT array_length(array[1,2,3], 1) AS RESULT;
 result
--------
      3
(1 row)
```

array_lower(anyarray, int)

Description: Returns the lower bound of the specified array dimension.

Return type: int.

Example:

```
postgres=# SELECT array_lower('[0:2]={1,2,3}'::int[], 1) AS RESULT;
 result
--------
      0
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

array_upper(anyarray, int)

Description: Returns the upper bound of the specified array dimension.

Return type: int.

Example:

```
postgres=# 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 for the array and the second **text** to substitute for null values in the array.

Return type: text.

Example:

```
postgres=# 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 is NULL, substrings in the string without input content are replaced with NULL.

In **array_to_string**, if the null string parameter is omitted or is NULL, any null elements in the array are skipped during processing and do not appear in the output string.

string_to_array(text, text [, text])

Description: Uses the second **text** to specify a delimiter and the third optional **text** as a template for replacing NULL values. If a split substring exactly matches the third optional **text**, it is replaced with NULL.

Return type: text[].

Example:

```
postgres=# SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
    result
--------------
 {xx,NULL,zz}
(1 row)
postgres=# 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 becomes a separate element in the resulting array. If the delimiter is an empty string, the entire input string becomes a single-element array. Otherwise, the input string is split at each occurrence of the delimiter string.

 Copyright © Huawei Technologies Co., Ltd.

unnest(anyarray)

Description: Expands an array into a set of rows.

Return type: setof anyelement.

Example:

```
postgres=# SELECT unnest(ARRAY[1,2]) AS RESULT;
 result
--------
      1
      2
(2 rows)
```

The **unnest** function is used in conjunction with the **string_to_array** function. To transform an array into columns, first split the string into an array using commas, then expand the array 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 the input integer array and returns the last array index that is less than or equal to the target parameter **n**. If **n** is NULL, **-1** is returned. The **interval** function does not support the interval(N, N1) scenario.

Return type: int.

Example:

```
postgres=# 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: Splits the string according to the delimiter and returns an array.

Return type: text[].

Example:

```
postgres=# SELECT SPLIT('a-b-c-d-e', '-') AS RESULT;
   result
-------------
 {a,b,c,d,e}
(1 row)
postgres=# SELECT SPLIT('a-b-c-d-e', '-')[4] AS RESULT;
 result
--------
 d
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

11.7 Logical Operators

Common logical operators include **AND**, **OR**, and **NOT**. The operation results are **TRUE**, **FALSE**, and **NULL**, where **NULL** indicates unknown. The operation precedence order is: NOT > AND > OR.

For detailed calculation rules, refer to **Table 11-5**, where a and b represent logical expressions.

Table 11-5 Operation rules

a	b	Result of a AND b	Result of a OR b	Result of NOT a
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
NULL	NULL	NULL	NULL	NULL

NOTE

The operators AND and OR are commutative, meaning swapping the left and right operands does not affect the result.

11.8 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 11-6 describes the comparison operators provided by Aura.

Table 11-6 Comparison operators

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to

 Copyright © Huawei Technologies Co., Ltd.

Operators	Description
>=	Greater than or equal to
=	Equality
<> or !=	Inequality

11.9 Pattern Matching Operators

There are three separate approaches to pattern matching provided by the database: SQL LIKE 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. An underscore (_) represents (matches) any single character, and a percent sign (%) represents a wildcard for any string.
  c. To match a literal underscore (_) or percent sign (%), the corresponding character in **pattern** must be preceded by the escape character. The role of an escape character is to disable the special meaning of metacharacters. By default, the escape character is a backslash, but a different escape character can also be specified using the **ESCAPE** clause.
  d. To match the escape character itself, use 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.

 Copyright © Huawei Technologies Co., Ltd.

- Examples

```
postgres=# SELECT 'abc' LIKE 'abc' AS RESULT;
 result
------------
 t
(1 row)
postgres=# SELECT 'abc' LIKE 'a%' AS RESULT;
 result
------------
 t
(1 row)
postgres=# SELECT 'abc' LIKE '_b_' AS RESULT;
 result
------------
 t
(1 row)
postgres=# SELECT 'abc' LIKE '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 11-7** lists all available operators for POSIX regular expression pattern matching.

Table 11-7 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
  a. 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:

 Copyright © Huawei Technologies Co., Ltd.

Table 11-8 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 substrings that match the POSIX regular expression pattern.
  - The **regexp_replace(string, pattern, replacement [,flags])** function provides the capability to replace substrings that match the POSIX regular expression pattern with 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.
  - Similar to **regexp_split_to_table**, **regexp_split_to_array(string text, pattern text [, flags text])** is a regular expression separation function, but its result is returned as a text array.

    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

```
postgres=#  SELECT 'abc' ~ 'Abc' AS RESULT;
result
--------
 f
(1 row)
postgres=# SELECT 'abc' ~* 'Abc' AS RESULT;
 result
--------
 t
(1 row)
postgres=# SELECT 'abc' !~ 'Abc' AS RESULT;
 result
--------
 t
(1 row)
postgres=# SELECT 'abc'!~* 'Abc' AS RESULT;
 result
--------
 f
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT 'abc' ~ '^a' AS RESULT;
 result
--------
 t
(1 row)
postgres=# SELECT 'abc' ~ '(b|d)'AS RESULT;
 result
--------
 t
(1 row)
postgres=# 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. It is not recommended that you accept the regular expression search mode from the non-security mode source. If you must do this, you are advised to add the 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. The LIKE search is much simpler than the other two options. Therefore, it is more secure to accept the non-secure mode source search.

11.10 Aggregate Functions

sum(expression)

Description: Sum of expressions across all input rows.

Return type:

Generally, the input data type is the same as the output data type. However, type conversion occurs in the following cases:

- For **SMALLINT** or **INT** inputs, the output type is **BIGINT**.
- For **BIGINT** input, the output type is **NUMBER**.
- For floating-point input, the output type is **DOUBLE PRECISION**.

Example:

```
postgres=# SELECT SUM(ss_ext_tax) FROM tpcds.STORE_SALES;
  sum
--------------
 213267594.69
(1 row)
```

max(expression)

Description: Specifies the maximum value of **expression** across all input values.

Parameter type: any array, numeric, string, or date/time type.

Return type: same as the parameter type.

Example:

```
postgres=# SELECT MAX(inv_quantity_on_hand) FROM tpcds.inventory;
  max
```

 Copyright © Huawei Technologies Co., Ltd.

```
---------
 1000000
(1 row)
```

min(expression)

Description: Specifies the minimum value of **expression** across all input values.

Parameter type: any array, numeric, string, or date/time type.

Return type: same as the parameter 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:

- For any integer input, the result is of the **NUMBER** type.
- For any floating-point input, the result is of the **DOUBLE PRECISION** type.
- Others: same as the input data type.

Example:

```
postgres=# SELECT AVG(inv_quantity_on_hand) FROM tpcds.inventory;
        avg
----------------------
 500.0387129084044604
(1 row)
```

count(expression)

Description: Returns the number of rows in the table where the **expression** is not NULL.

Return type: bigint.

Example:

```
postgres=# SELECT COUNT(inv_quantity_on_hand) FROM tpcds.inventory;
  count
----------
 11158087
(1 row)
```

count(*)

Description: Returns the total number of records in the table.

Return type: bigint.

Example:

```
postgres=# SELECT COUNT(*) FROM tpcds.inventory;
  count
```

 Copyright © Huawei Technologies Co., Ltd.

```
----------
 11745000
(1 row)
```

array_agg(expression)

Description: Concatenates all input values (including nulls) into an array. Function inputs do not support array formats.

Return type: Array of the parameter type.

Example:

Create a table named **employeeinfo** and insert data:

```
CREATE EXTERNAL TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate
date,deptno smallint) store AS orc;
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 with ID **30**:

```
SELECT array_agg(ename) FROM employeeinfo where deptno = 30;
 array_agg
------------
 {JACK,MAX}
(1 row)
```

Query all employees belonging to 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 unique department IDs:

```
SELECT array_agg(distinct deptno) FROM employeeinfo group by deptno;
 array_agg
-----------
 {20}
 {30}
(2 rows)
```

string_agg(expression, delimiter)

Description: Joins input values into a single string separated by the specified delimiter.

 Copyright © Huawei Technologies Co., Ltd.

Return type: same as the parameter type.

Example:

Based on the created table **employeeinfo**, query all employees belonging to the same department:

```
SELECT deptno, string_agg(ename,',') FROM employeeinfo group by deptno;
 deptno | string_agg
--------+------------
     30 | JACK,MAX
     20 | TOM
(2 rows)
```

Query employees with employee IDs less than 7156:

```
SELECT string_agg(ename,',') FROM employeeinfo where empno < 7156;
 string_agg
------------
 TOM,JACK
(1 row)
```

covar_pop(Y, X)

Description: Population covariance.

Return type: double precision.

Example:

```
postgres=#  SELECT COVAR_POP(sr_fee, sr_net_loss) FROM tpcds.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 tpcds.store_returns WHERE sr_customer_sk <
1000;
    covar_samp
------------------
 830.052235037289
(1 row)
```

stddev_pop(expression)

Description: Population standard deviation.

Return type: **double precision** for floating-point inputs, and **numeric** for others.

Example:

```
postgres=# SELECT STDDEV_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk =
1;
    stddev_pop
------------------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 289.224294957556
(1 row)
```

stddev_samp(expression)

Description: Sample standard deviation.

Return type: **double precision** for floating-point inputs, and **numeric** for others.

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 (square of population standard deviation).

Return type: **double precision** for floating-point inputs, and **numeric** for others.

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 (square of sample standard deviation).

Return type: **double precision** for floating-point inputs, and **numeric** for others.

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: Bitwise AND of all non-NULL input values. NULL if all inputs are NULL.

Return type: same as the parameter type.

Example:

```
postgres=# SELECT BIT_AND(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_and
---------
       0
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

bit_or(expression)

Description: Bitwise OR of all non-NULL input values. NULL if all inputs are NULL.

Return type: same as the parameter 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: The value is **true** if all input values are **true**. Otherwise, the value is **false**.

Return type: Boolean.

Example:

```
postgres=# SELECT bool_and(100 <2500);
 bool_and
----------
 t
(1 row)
```

bool_or(expression)

Description: The value is **true** if at least one input value is **true**. Otherwise, the value is **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.

 Copyright © Huawei Technologies Co., Ltd.

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 tpcds.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 |             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
(42 rows)
```

regr_avgx(Y, X)

Description: Mean of independent variables (sum(X)/N).

 Copyright © Huawei Technologies Co., Ltd.

Return type: double precision.

Example:

```
postgres=# SELECT REGR_AVGX(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk <
1000;
    regr_avgx
------------------
 578.606576740795
(1 row)
```

regr_avgy(Y, X)

Description: Mean of dependent variables (sum(Y)/N).

Return type: double precision.

Example:

```
postgres=# SELECT REGR_AVGY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk <
1000;
    regr_avgy
------------------
 50.0136711629602
(1 row)
```

regr_count(Y, X)

Description: Number of input rows where both expressions are NOT NULL.

Return type: bigint.

Example:

```
postgres=# SELECT REGR_COUNT(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk <
1000;
 regr_count
------------
       2743
(1 row)
```

regr_intercept(Y, X)

Description: Fits all input points (X, Y) into a linear equation using least squares and return the Y-intercept.

Return type: double precision.

Example:

```
postgres=# SELECT REGR_INTERCEPT(sr_fee, sr_net_loss) FROM tpcds.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:

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# SELECT REGR_R2(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
      regr_r2
--------------------
 0.00145453469345058
(1 row)
```

regr_slope(Y, X)

Description: Fits all input points (X, Y) into a linear equation using the least square method and returns the slope of the straight line.

Return type: double precision.

Example:

```
postgres=# SELECT REGR_SLOPE(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk <
1000;
     regr_slope
--------------------
 0.00139920009665259
(1 row)
```

regr_sxx(Y, X)

Description: **sum(X^2) - sum(X)^2/N** (sum of squares of the independent variable).

Return type: double precision.

Example:

```
postgres=# SELECT REGR_SXX(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
     regr_sxx
------------------
 1626645991.46135
(1 row)
```

regr_sxy(Y, X)

Description: **sum(X*Y) - sum(X) * sum(Y)/N** (product of sums of the independent and dependent variables).

Return type: double precision.

Example:

```
postgres=# SELECT REGR_SXY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
     regr_sxy
------------------
 2276003.22847225
(1 row)
```

regr_syy(Y, X)

Description: **sum(Y^2) - 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 tpcds.store_returns WHERE sr_customer_sk < 1000;
    regr_syy
-----------------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 2189417.6547314
(1 row)
```

stddev(expression)

Description: Alias of **stddev_samp**.

Return type: **double precision** for floating-point inputs, and **numeric** for others.

Example:

```
postgres=# SELECT STDDEV(inv_quantity_on_hand) FROM tpcds.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 inputs, and **numeric** for others.

Example:

```
postgres=# SELECT VARIANCE(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
      variance
--------------------
 83650.730277028768
(1 row)
```

UNIQ(col_name)

Description: Aggregate function for calculating the number of non-duplicate values. Similar to **COUNT DISTINCT**, this function calculates the number of rows in a column after deduplication and returns a deduplicated value.

Input parameter description: **col_name** indicates the column whose number of rows after deduplication needs to be calculated. The SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, TEXT, VARCHAR, TIMESTAMP, TIMESTAMPTZ, DATE, TIMETZ and UUID types are supported.

NOTE

- 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.
- When the number of columns to be deduplicated in the SQL statement is greater than or equal to 3, you are advised to use the **UNIQ** function for deduplication to achieve a better effect than **COUNT DISTINCT**.
- 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 instead.

Example:

```
CREATE EXTERNAL TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate
date,deptno smallint) store AS orc;
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;
```

 Copyright © Huawei Technologies Co., Ltd.

```
 uniq
------
    1
    1
    1
(3 rows)
```

11.11 Window Functions

Ordinary aggregate functions can only compute results within a row or aggregate all rows into a single result. In contrast, window functions can perform calculations across rows and populate the result in each row.

By applying a window function to certain portions of the rows filtered through queries, functionality akin to the **11.10 Aggregate Functions** is achieved, allowing aggregate functions to also serve as window functions. Window functions can scan all rows, simultaneously displaying both raw data and aggregated analytical results.

Precautions

- Column-store tables currently only support the window functions **rank(expression)** and **row_number(expression)**, along with the aggregate functions **sum**, **count**, **avg**, **min**, and **max**. There are no such restrictions for row-store tables.
- A single query may include one or multiple window function expressions.
- Window functions can only appear in output columns. To use a window function's value for conditional filtering, it must be nested within a subquery, employing the alias of the window function expression at the outer level for filtering conditions. Example:
  ```
  SELECT classid, id, score FROM(SELECT *, avg(score) OVER(PARTITION BY classid) as avg_score FROM score) WHERE score >= avg_score;
  ```
- Query blocks containing window functions support de-duplication via the **GROUP BY** expression. It is required that the **PARTITION BY** clause in the window function be a subset of the **GROUP BY** expression to ensure the window function operates on the de-duplicated outcome of the **GROUP BY** column. Additionally, the expression in the **ORDER BY** clause should either be a subset of the **GROUP BY** expression or an aggregate function from an aggregation operation. Example:
  ```
  SELECT classid,rank() OVER(PARTITION BY classid ORDER BY sum(score)) as avg_score FROM score GROUP BY classid, id;
  ```

Syntax

Window functions use the specialized **OVER** clause to define the triggering scope of the window function. The **OVER** clause groups data and orders elements within those groups. Window functions assign serial numbers to values within a 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
```

Options for the **window_definition** clause include:

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
```

 Copyright © Huawei Technologies Co., Ltd.

```
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

PARTITION BY: Groups rows sharing identical **PARTITION BY** expression values.

ORDER BY: Dictates the processing sequence of rows by the window function. Following **ORDER BY**, a column name is mandatory. If a numeral follows, it is treated as a constant, rendering sorting ineffective on the target column.

For the **frame_clause** options:

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

Specifying a window to compute results for all rows within a group entails defining the starting and ending points of the window range. Supported modes are **RANGE** and **ROWS**, with **ROWS** delineating windows by physical units (rows) and **RANGE** designating them as logical offsets.

Within **RANGE** and **ROWS**, boundaries can be set using **BETWEEN frame_start AND frame_end**. If only **frame_start** is specified, **frame_end** is **CURRENT ROW** by default.

Options for **frame_start** and **frame_end**:

- **CURRENT ROW**: Refers to the present row.
- *N* **PRECEDING**: Indicates the *n*-th row prior to the current one.
- **UNBOUNDED PRECEDING**: Indicates the initial row of the current partition.
- *N* **FOLLOWING**: Indicates the *n*-th row subsequent to the current one.
- **UNBOUNDED FOLLOWING**: Indicates the final row of the current partition.

frame_start cannot be **UNBOUNDED FOLLOWING**, nor can *frame_end* be **UNBOUNDED PRECEDING**. Moreover, *frame_end* cannot be earlier than *frame_start*. For example, **RANGE BETWEEN CURRENT ROW AND** *N* **PRECEDING** is invalid.

The **LAST_VALUE** function accommodates the **IGNORE NULLS** syntax, returning the last non-null value within its window. If all values are null, null is returned. Its specific format is:

```
LAST_VALUE (expression [IGNORE NULLS]) OVER (window_definition)
```

Currently, **IGNORE NULLS** is only compatible with the following window ranges: **ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING** and **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**.

RANK()

Description: The **RANK** function generates skip sequence numbers for values in each group. The same values have the same sequence number, but the same values occupy multiple sequence numbers.

Return type: bigint.

Example:

Given the **table score(id, classid, score)**, each row represents the student ID, class ID, and exam score.

 Copyright © Huawei Technologies Co., Ltd.

Use the **RANK** function to sort students' scores.

```
CREATE EXTERNAL 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 generates consecutive sequence numbers for 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 generates consecutive sequence numbers for values in each group. The same values have the same sequence number, and the same values occupy only one 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 generates a relative sequence number for the corresponding value in each group, which is calculated using the formula **(Rank – 1)/(Total rows – 1)**. Here, **Rank** is the corresponding sequence number

 Copyright © Huawei Technologies Co., Ltd.

generated based on the **RANK** function, and **total rows** is the total number of elements in the 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 generates cumulative distribution sequence numbers for the corresponding values in each group. This is calculated using the formula (Number of data rows less than or equal to the current value)/(Total number of rows in the group).

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 | .333333333333333
  2 |       2 |    95 | .333333333333333
  5 |       2 |    88 |               .5
  3 |       2 |    85 | .666666666666667
  4 |       1 |    70 |                1
  6 |       1 |    70 |                1
(6 rows)
```

NTILE(num_buckets integer)

Description: The **NTILE** function evenly distributes an ordered dataset into the number of buckets specified by **num_buckets** and assigns a bucket number to each row. The distribution should be as balanced 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 generates lagged values for the corresponding values in each group. It uses the value from the row obtained by shifting the

 Copyright © Huawei Technologies Co., Ltd.

current value's row back by offset positions as the sequence number. If the shifted row does not exist, the result defaults to the default value. If unspecified, **offset** defaults to **1** and **default** defaults 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 generates lead values for the corresponding values in each group. It uses the value from the row obtained by shifting the current value's row forward by offset positions as the sequence number. If the shifted row exceeds the total number of rows in the current group, the result defaults to the default value. If unspecified, **offset** defaults to **1** and **default** defaults 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 returns the first value in each group as the result.

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)
```

 Copyright © Huawei Technologies Co., Ltd.

LAST_VALUE(value any)

Description: The **LAST_VALUE** function returns the last value in each group as the result.

Return type: same as the parameter type.

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 **NTH_VALUE** function returns the *n*th row in the group as the result. If the row does not exist, it defaults to returning **NULL**.

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)
```

11.12 Type Conversion Functions

cast(x as y)

Description: A type conversion function that converts x into the type specified by y.

Example:

```
postgres=# SELECT cast('22-oct-1997' as timestamp);
      timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

cast(x, y)

Description: A type conversion function that converts x into the type specified by y.

Example:

```
postgres=# SELECT cast('22-oct-1997', timestamp);
      timestamp
```

 Copyright © Huawei Technologies Co., Ltd.

```
---------------------
 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 Aura, NULL is returned. Otherwise, an error is raised.

Example:

```
postgres=# SELECT try_cast('a' as int4);
     int4
---------------


(1 row)
postgres=# SELECT try_cast('22-oct-1997', timestamp);
     timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

to_char (datetime/interval [, fmt])

Description: Converts a **DATETIME** or **INTERVAL** value of the **DATE**, **TIMESTAMP**, **TIMESTAMP WITH TIME ZONE**, or **TIMESTAMP WITH LOCAL TIME ZONE** type to the **VARCHAR** type according to the format specified by **fmt**.

- Optional parameter **fmt** can include categories such as date, time, week, quarter, and century. Each category can have different templates, which can be reasonably combined. Common templates include HH, MM, SS, YYYY, MM, and DD. Refer to **Table 11-10** for details.
- Templates can have modifiers, with FM being a common one used to suppress leading zeros or trailing spaces.

Return type: varchar.

Example:

```
postgres=# SELECT to_char(current_timestamp,'HH12:MI:SS');
 to_char
----------
 10:19:26
(1 row)
postgres=# SELECT to_char(current_timestamp,'FMHH12:FMMI:FMSS');
 to_char
----------
 10:19:46
(1 row)
```

to_char(double precision, text)

Description: Converts double-precision values into strings in the specified format.

Return type: text.

Example:

```
postgres=# SELECT to_char(125.8::real, '999D99');
 to_char
---------
```

 Copyright © Huawei Technologies Co., Ltd.

```
  125.80
(1 row)
```

to_char (integer/number[, fmt])

Description: Converts integer or floating-point values into strings in the specified format.

- Optional parameter **fmt** can include categories such as decimal digits, group separators, plus/minus signs, and currency symbols. Each category can have different templates, which can be reasonably combined. Common templates include 9, 0, , (thousands separator), . (decimal point). Refer to **Table 11-9** for details.
- Templates can have modifiers like FM, but FM does not suppress zeros specified by the template 0.
- To convert integer values to their corresponding hexadecimal string representations, use the template **x** or **X**.

Return type: varchar.

Example:

```
postgres=# SELECT to_char(1485,'9,999');
 to_char
---------
  1,485
(1 row)
postgres=# SELECT to_char( 1148.5,'9,999.999');
  to_char
------------
  1,148.500
(1 row)
postgres=# SELECT to_char(148.5,'990999.909');
   to_char
-------------
    0148.500
(1 row)
postgres=# 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:

```
postgres=# SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
 to_char
----------
 15:02:12
(1 row)
```

to_char(int, text)

Description: Converts values of the integer type into strings in the specified format.

 Copyright © Huawei Technologies Co., Ltd.

Return type: text.

Example:

```
postgres=# 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:

```
postgres=# SELECT to_char(-125.8, '999D99S');
 to_char
---------
 125.80-
(1 row)
```

to_char (string)

Description: Converts the CHAR, VARCHAR, VARCHAR2, or CLOB type to the VARCHAR type.

If this function is used to convert the CLOB type and the value to be converted is not within the range of the target type, an error is returned.

Return type: varchar.

Example:

```
postgres=# 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:

```
postgres=# SELECT to_char(current_timestamp, 'HH12:MI:SS');
 to_char
----------
 10:55:59
(1 row)
```

to_date(text)

Description: Converts values of the text type into the timestamp in the specified format.

 Copyright © Huawei Technologies Co., Ltd.

Return type: timestamp.

Example:

```
postgres=# 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:

```
postgres=# 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 in the format specified by **fmt**. For details about the **fmt** format, see **Table 11-10**.

This function cannot directly support the CLOB type, but parameters of the CLOB type can be implicitly converted.

Return type: date.

Example:

```
postgres=# 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 in the specified format.

For details about the type conversion format, see **Table 11-9**.

When a hexadecimal string is converted into a decimal number, a maximum of 16 bytes of hexadecimal string can be converted into an unsigned number.

When a hexadecimal string is converted into a decimal number, the format string cannot contain other characters except x or X. Otherwise, an error is reported.

Return type: number.

Example:

```
postgres=# SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 -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:

```
postgres=# SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
 -12454.8
(1 row)
```

to_timestamp(double precision)

Description: Converts a UNIX epoch to a timestamp.

Return type: timestamp with time zone.

Example:

```
postgres=# 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 based on the format specified by **fmt**. If **fmt** is not specified, the format specified by **nls_timestamp_format** is used. For details about the **fmt** format, see **Table 11-10**.

In **to_timestamp** in Aura,

- If the input year YYYY is 0, the system reports an error.
- If the input year YYYY is less than 0 and SYYYY is specified in **fmt**, the year corresponding to the absolute value **n** BC is output correctly.

The characters in the **fmt** must match the date/time formatting mode. Otherwise, an error is reported.

Return type: timestamp without time zone.

Example:

```
postgres=# SHOW nls_timestamp_format;
    nls_timestamp_format
----------------------------
 DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

postgres=# SELECT to_timestamp('12-sep-2014');
    to_timestamp
---------------------
 2014-09-12 00:00:00
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

```
postgres=# 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)
postgres=# SELECT to_timestamp('-1','SYYYY');
      to_timestamp
------------------------
 0001-01-01 00:00:00 BC
(1 row)
postgres=# SELECT to_timestamp('98','RR');
    to_timestamp
---------------------
 1998-01-01 00:00:00
(1 row)
postgres=# SELECT to_timestamp('01','RR');
    to_timestamp
---------------------
 2001-01-01 00:00:00
(1 row)
```

to_timestamp(text, text)

Description: Converts a value of the string type to a timestamp in the specified format.

Return type: timestamp.

Example:

```
postgres=# SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
    to_timestamp
---------------------
 2000-12-05 00:00:00
(1 row)
```

Table 11-9 shows the template patterns that can be used to format numeric value, for the function **to_number**.

Table 11-9 Template patterns for numeric formatting

Pattern	Description
9	Value with the specified number of numeric value digits
0	Value with leading zeros
. (period)	Decimal point
, (comma)	Grouping (thousand) separator
PR	Negative value in angle brackets
S	Signed numeric value (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 number < 0)

 Copyright © Huawei Technologies Co., Ltd.

Pattern	Description
PL	Plus sign in the specified position (if number > 0)
SG	Plus/minus sign in the specified position
RN	Roman numeral (input between 1 and 3999)
TH or th	Ordinal suffix
V	Shift specified digits (decimal)

Table 11-10 shows the templates you can use to format date and time values. These patterns apply to the functions **to_date**, **to_timestamp**, **to_char**, and the parameter **nls_timestamp_format**.

Table 11-10 Patterns for date and time formatting

Category	Pattern	Description
Hour	HH	Hour of the day (01–12)
	HH12	Hour of the day (01–12)
	HH24	Hour of the day (00–23)
Minute	MI	Minute (00–59)
Second	SS	Second (00–59)
	FF	Microsecond (000000–999999)
	SSSSS	Second after midnight (0–86399)
AM or PM	AM or A.M.	AM indicator
	PM or P.M.	PM indicator
Year	Y,YYY	Year with comma (4 or more digits)
	SYYYY	Four-digit BC year
	YYYY	Year (4 or more digits)
	YYY	Last three digits of the year
	YY	Last two digits of the year
	Y	Last one digit of the year
	IYYY	ISO year (4 or more digits)
	IYY	Last three digits of the ISO year
	IY	Last two digits of the ISO year
	I	Last one digit of the ISO year

 Copyright © Huawei Technologies Co., Ltd.

Category	Pattern	Description
	RR	Last two digits of the year (can store 20th-century years in the 21st century) Rules are as follows: • Entered 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 are the first two digits of the current year plus 1. • Entered 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 are 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	Can accept 4-digit years or 2-digit years. If it is two digits, the value is the same as the return value of RR. If it is four digits, the value is the same as the return value of YYYY.
	• BC or B.C. • AD or A.D.	Era indicators. Before Christ (BC) and Anno Domini (AD).
Month	MONTH	Full-length uppercase month name (blank-padded to 9 characters)
	MON	Abbreviated uppercase month name (3 characters)
	MM	Month number (01–12)
	RM	Roman numeral month (I-XII; I=JAN) (uppercase)
Day	DAY	Full-length uppercase day name (blank-padded to 9 characters)
	DY	Abbreviated uppercase day name (3 characters)
	DDD	Day of the year (001–366)
	DD	Day of the month (01–31)

 Copyright © Huawei Technologies Co., Ltd.

Category	Pattern	Description
	D	Day of the week (1–7; Sunday is 1)
Week	W	Week of the month (1–5) (First week starts on the first day of the month)
	WW	Week of the year (1–53) (First week starts on the first day of the year)
	IW	ISO week of the year (First Thursday is in the first week)
Century	CC	Century (2 digits) (21st century starts from 2001-01-01)
Julian day	J	Julian day (days since January 1, 4712 BC)
Quarter	Q	Quarter

11.13 JSON/JSONB Functions and Operators

11.13.1 JSON/JSONB Operators

Table 11-11 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.	postgres=# SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2; ?column? ---------- {"c":"baz"} (1 row)
->	object-json(b)	text	json(b)	Obtains the value by a key. If no record exists, **NULL** is returned.	postgres=# SELECT '{"a": {"b":"foo"}}'::json->'a'; ?column? ---------- {"b":"foo"} (1 row)

 Copyright © Huawei Technologies Co., Ltd.

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.	postgres=# SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->>2; ?column? ---------- {"c":"baz"} (1 row)
->>	object-json(b)	text	text	Obtains the value by a key. If no record exists, **NULL** is returned.	postgres=# SELECT '{"a": {"b":"foo"}}'::json->>'a'; ?column? ---------- {"b":"foo"} (1 row)
#>	container-json(b)	text[]	json	Obtains the JSON object in the specified path. If the path does not exist, **NULL** is returned. **NOTE** An Aura object identifier can end with a number sign (#). To avoid ambiguity during the parsing of a#>b, you need to add spaces around the operator **#>**. Otherwise, an error will occur during parsing.	postgres=# SELECT '{"a":{"b": {"c":1}}}'::json #> '{a, b}'; ?column? ---------- {"c":1} (1 row)

 Copyright © Huawei Technologies Co., Ltd.

Operator	Left Operand Type	Right Operand Type	Return Type	Description	Example
#>>	container-json (b)	text[]	text	Obtains the JSON object in the specified path. If the path does not exist, **NULL** is returned.	postgres=# SELECT '{"a":{"b":{"c":1}}}'::json #>> '{a, b}'; ?column? ---------- {"c":1} (1 row)

Table 11-12 Operators supported by jsonb

Operator	Right Operand Type	Return Type	Description	Example
=	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":1}}}'::jsonb = '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
<>	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":1}}}'::jsonb <> '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
<	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":2}}}'::jsonb < '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
>	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":2}}}'::jsonb > '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
<=	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":2}}}'::jsonb <= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)

 Copyright © Huawei Technologies Co., Ltd.

Operator	Right Operand Type	Return Type	Description	Example
>=	jsonb	bool	Same as the **jsonb_eq** function, which compares the size of two jsonb files.	postgres=# SELECT '{"a":{"b":{"c":2}}}'::jsonb >= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
?	text	bool	Whether the string of the key or element exists at the top layer of the JSON value.	postgres=# SELECT '{"a":1, "b":2}'::jsonb ? 'b'; ?column? ---------- t (1 row)
?|	text[]	bool	Whether any of these array strings exists as a top-layer key.	postgres=# SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?| '{a, b, e}'::text[]; ?column? ---------- t (1 row)
?&	text[]	bool	Whether all these array strings exist as top-layer keys.	postgres=# SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?& '{a, b, c}'::text[]; ?column? ---------- t (1 row)
<@	jsonb	bool	Whether all items in the JSON file on the left exist at the top layer of the JSON file on the right.	postgres=# SELECT '{"b":3}'::jsonb <@ '{"a":{"b":{"c":2}}, "b":3}'::jsonb; ?column? ---------- t (1 row)
@>	jsonb	bool	Whether all items in the JSON file on the right exist at the top layer of the JSON file on the left.	postgres=# SELECT '{"a":{"b":{"c":2}}, "b":3}'::jsonb @> '{"b":3}'::jsonb; ?column? ---------- t (1 row)
||	jsonb	jsonb	Combines two JSONB objects into one.	postgres=# SELECT '{"a":1, "b":2}'::jsonb || '{"c":3, "d":4}'::jsonb; ?column? ---------- {"a": 1, "b": 2, "c": 3, "d": 4} (1 row)
-	text	jsonb	Deletes a jsonb object and the specified key-value pair.	postgres=# SELECT '{"a":1, "b":2}'::jsonb - 'a'; ?column? ---------- {"b": 2} (1 row)

 Copyright © Huawei Technologies Co., Ltd.

Operator	Right Operand Type	Return Type	Description	Example
-	text	jsonb	Deletes a jsonb object and the specified key-value pair.	postgres=# SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb - '{a, b}'::text[]; ?column? ------------------ {"c": 3, "d": 4} (1 row)
-	int	jsonb	Deletes the element corresponding to the subscript in the JSONB array.	postgres=# SELECT '["a", "b", "c"]'::jsonb - 2; ?column? ------------ ["a", "b"] (1 row)
#-	text[]	jsonb	Deletes the key-value pair corresponding to the path in the JSONB object.	postgres=# SELECT '{"a":{"b":{"c":{"d":1}}}, "e":2, "f":3}'::jsonb #- '{a, b}'::text[]; ?column? ---------------------------- {"a": {}, "e": 2, "f": 3} (1 row)

11.13.2 JSON/JSONB Functions

JSON/JSONB functions represent functions that can be used for JSON-type data.

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:

 Copyright © Huawei Technologies Co., Ltd.

```
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 possibly heterogeneously typed JSON array from a variadic 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 variadic parameter list. The parameter list consists of alternating keys and values. The number of input parameters must be even, with every two parameters forming a key-value pair. Note that the 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 is a single text array, its length must be even, and the members are treated as alternately occurring key/value pairs. When two text arrays are provided, the first array is treated as keys and the second as values. Both arrays must have the same length. Keys cannot be null.

Return type: json.

Example:

```
postgres=# SELECT json_object('{a, 1, b, "def", c, 3.5}');
              json_object
```

 Copyright © Huawei Technologies Co., Ltd.

```
----------------------------------------
 {"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)
```

json_strip_nulls(json)

Description: All object fields with null values are omitted, while 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: Similar to the operator **->**, it returns the value corresponding to the specified key in the 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: Similar to the operator **->>**, it returns the value corresponding to the specified key in the object.

Return type: text.

Example:

```
postgres=# SELECT json_object_field_text('{"a": {"b":"foo"}}','a');
 json_object_field_text
------------------------
 {"b":"foo"}
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

json_array_element(array-json, integer)

Description: Similar to the operator **->**, it returns the element at the specified index 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: Similar to the operator **->>**, it returns the element at the specified index 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 **#>**, it returns the JSON value of the path pointed to 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 **#>>**, it returns the text value of the path pointed to 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, returning each element as a row.

 Copyright © Huawei Technologies Co., Ltd.

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, returning each element as 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 top-level keys in 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 the object into one row with two columns.

 Copyright © Huawei Technologies Co., Ltd.

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 the object into one row with 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_to_record(object-json)

Description: As with all functions that return a record, the caller must explicitly define the structure of the record with an AS clause. It splits and reassembles the key-value pairs of **object-json**, using the key as the column name to match and fill 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 within the array by referring to the function **json_to_record**. Hence, every element in the array must be of the **object-json** type.

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)
```

 Copyright © Huawei Technologies Co., Ltd.

```
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_to_record_array(anyelement, array-json [, bool])

Description: **$1** must be a parameter of a composite type. It splits each key-value pair in **object-json**, using the key as the column name to match and find the corresponding column name in **$1**. The matched column must be of the TEXT type and will be filled into the format of **$1**.

Return type: text[].

Example:

```
CREATE TYPE jpop_array AS (a text, b text, c timestamp);
SELECT * FROM json_to_record_array(null::jpop_array , '{"a":1,"b":2}');
 json_to_record_array
----------------------
 {1,2,NULL}
(1 row)
```

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 is a single text array, its length must be even, and the members are treated as alternately occurring 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)
```

 Copyright © Huawei Technologies Co., Ltd.

jsonb_object(text[], text[])

Description: When two text arrays are used, the first array is considered keys and the second array is considered values. The lengths of the two arrays must be equal. Keys 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)
```

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)
```

 Copyright © Huawei Technologies Co., Ltd.

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 parameters must be even, with every two parameters forming a key-value pair. Note that keys 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: Omits all object fields with null values. 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: Similar to the operator **->**, it returns the value corresponding to the specified key in the object.

Return type: jsonb.

Example:

```
postgres=# SELECT jsonb_object_field('{"a": {"b":"foo"}}','a');
 jsonb_object_field
--------------------
 {"b": "foo"}
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

jsonb_object_field_text(jsonb, text)

Description: Similar to the operator **->>**, it returns the value corresponding to the specified key in the 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: Similar to the operator **->**, it returns the element at the specified index 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: Similar to the operator **->>**, it returns the element at the specified index 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: Equivalent to the operator **#>**, it returns the value of the path pointed to 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)
```

 Copyright © Huawei Technologies Co., Ltd.

jsonb_extract_path_text(jsonb, VARIADIC text[])

Description: Equivalent to the operator **#>>**, it returns the value of the path pointed to 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: Takes any **object-jsonb** or **array-jsonb** type as input and returns the value at the path specified by **$2**.

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: Takes any **object-jsonb** or **array-jsonb** type as input and returns the value at the path specified by **$2**.

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, returning each element as a row.

Return type: jsonb.

 Copyright © Huawei Technologies Co., Ltd.

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, returning each element as 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 top-level keys in 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 the object into one row with two columns.

Return type: setof(key text, value jsonb).

 Copyright © Huawei Technologies Co., Ltd.

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 the object into one row with 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_to_record(object-json)

Description: As with all functions that return a record, the caller must explicitly define the structure of the record with an **AS** clause. It splits and reassembles the key-value pairs of **object-json**, using the key as the column name to match and fill 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 of type **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)
```

 Copyright © Huawei Technologies Co., Ltd.

jsonb_to_record_array(anyelement, array-json [, bool])

Description: **$1** must be a parameter of a composite type. It splits each key-value pair in **object-jsonb**, using the key as the column name to match and find the corresponding column name in **$1**. The matched column must be of the TEXT type and will be filled into the format of **$1**.

Return type: text[].

Example:

```
postgres=# CREATE TYPE jpop_array AS (a text, b text, c timestamp);
postgres=# SELECT * FROM jsonb_to_record_array(null::jpop_array , '{"a":1,"b":2}');
 jsonb_to_record_array
-----------------------
 {1,2,NULL}
(1 row)
```

jsonb_ne(jsonb, jsonb)

Description: Same as the operator **<>**, which compares the size of 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 the size of 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 the size of 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 the size of two values.

 Copyright © Huawei Technologies Co., Ltd.

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 the size of 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 the size of 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 sizes. A positive number indicates greater than, a negative number indicates less than, and 0 indicates equality.

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 checks if the string **$2** exists at the top level of **$1** in the form of **key\elem\scalar**.

Return type: Boolean.

Example:

```
postgres=# SELECT jsonb_exists('["1",2,3]', '1');
 jsonb_exists
```

 Copyright © Huawei Technologies Co., Ltd.

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

jsonb_exists_any(jsonb, text[])

Description: Same as the operator **?|**, which checks if any element in the string array **$2** exists at the top level of **$1** in the form of **key\elem\scalar**.

Return type: Boolean

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 checks if all elements in the string array **$2** exist at the top level of **$1** in the form of **key\elem\scalar**.

Return type: Boolean

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 if all elements in **$1** exist at the top level 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 the value of **$1** contains all top-level elements of $2.

Return type: Boolean

Example:

```
postgres=# SELECT jsonb_contains('{"a":1, "b":2, "c":3}'::jsonb, '{"a":1}');
 jsonb_contains
------------------
 t
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

jsonb_concat(jsonb, jsonb)

Description: Concatenates two JSONB objects into one JSONB.

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 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 index.

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 a 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 at a specified path in a JSONB array.

Return type: jsonb.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
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**, where the part specified by **path** is replaced by **new_value**, or adds *new_value* if **create_missing** is **true** (**true** by default) and the item specified by **path** does not exist. As with path-oriented operators, negative integers in the **path** count from the end of the JSON array.

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 formatted JSON text with indentation.

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 **target** and inserts **new_value**. If the **target** part specified by **path** is in a JSONB array, **new_value** is inserted before the **target** or after it if **insert_after** is **true** (**false** by default). If the **target** part specified by **path** is in a JSONB object, **new_value** is inserted only if the **target** does not exist. As with path-oriented operators, negative integers in the **path** count 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
--------------------------------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 {"a": [0, "new_value", 1, 2]}
(1 row)
```

11.14 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:

```
postgres=# 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 no matches occur, it returns the default.

Examples:

```
postgres=# 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:

```
postgres=# SELECT if(1 < 2, 'yes', 'no');
 if
-----
 yes
(1 row)
```

NOTE

The **expr1** and **expr2** parameters can be of any type. For details about the return result type rules, refer to **9.5 UNION, CASE, and Related Constructs**.

 Copyright © Huawei Technologies Co., Ltd.

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:

```
postgres=# SELECT ifnull(NULL,'hello');
 ifnull
--------
 hello
(1 row)
```

NOTE

The **expr1** and **expr2** parameters can be of any type. For details about the return result type rules, refer to **9.5 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:

```
postgres=# 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:

```
postgres=# 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:

  ```
  postgres=# SELECT nullif('1234'::VARCHAR,123::INT4);
   nullif
  --------
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
  1234
(1 row)
postgres=# 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:

```
postgres=# 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:

```
postgres=# 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 **epr2** cannot be implicitly converted to **expr1** but **epr1** 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.

greatest(expr1 [, ...])

Description: Selects the largest value from a list of any number of expressions.

If an input parameter is **null**, **null** is returned.

Examples:

```
postgres=# SELECT greatest(1*2,2-3,4-1);
 greatest
----------
        3
(1 row)
postgres=# SELECT greatest('ABC', 'BCD', 'CDE');
 greatest
----------
 CDE
(1 row)
```

least(expr1 [, ...])

Description: Selects the smallest value from a list of any number of expressions.

If an input parameter is **null**, **null** is returned.

Examples:

```
postgres=# SELECT least(1*2,2-3,4-1);
 least
```

 Copyright © Huawei Technologies Co., Ltd.

```
-------
   -1
(1 row)
postgres=# SELECT least('ABC','BCD','CDE');
 least
--------
 ABC
(1 row)
```

11.15 Set Returning Functions

11.15.1 Series Generating Functions

generate_series() returns a series-based set based on the specified start value (**start**), end value (**stop**), and step (**step**).

The **generate_series()** function returns zero rows when the **step** parameter is positive and **start** is greater than **stop**. It also returns zero rows when the **step** parameter is negative and **start** is smaller than **stop**. 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
```

 Copyright © Huawei Technologies Co., Ltd.

```
-----------------
              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);
   dates
------------
 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)
```

11.16 System Information Functions

11.16.1 Session Information Functions

pg_backend_pid()

Description: Process ID of the service process connected to the current session.

Return type: integer.

Example:

```
postgres=# SELECT pg_backend_pid();
 pg_backend_pid
```

 Copyright © Huawei Technologies Co., Ltd.

```
-----------------
 140229352617744
(1 row)
```

user

Description: Equivalent to **current_user**.

Return type: name.

Example:

```
postgres=# SELECT user;
 current_user
--------------
 dbadmin
(1 row)
```

fq_metaserver_conninfo()

Description: Obtains the connection configuration of the metadata server.

Return type: record

Example:

```
postgres=# SELECT fq_metaserver_conninfo();

fq_metaserver_conninfo

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 (*********,fabrcsql_llt_serverless,*********.lakeformation.lakecat.com,*********,*********,obs.cn-north-5.myhuaweicloud.com,false,*********)
(1 row)
```

version()

Description: Version information. **version** returns a string describing the server's version information.

Return type: text.

Example:

```
postgres=# SELECT version();
                                                
```

format_type returns the SQL name based on the type OID of the data type and possible type modifiers. If the specific modifier is unknown, NULL is passed in the position of the type modifier. Generally, type modifiers are valid only for data types with length limitations. The SQL name returned by **format_type** contains the length value of the data type. The length value is the actual storage length len – sizeof (int32), in bytes. During data storage, 32-bit space is required to store the customized length information of the data type. That is, the actual storage length is four bytes longer than the customized length. In the following example, the SQL name returned by **format_type** is **character varying (6)**, where **6** indicates that the length value of the varchar type is 6 bytes. Therefore, the actual storage length of this type is 10 bytes.

```
SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'), 10);
    format_type
---------------------
 character varying(6)
(1 row)
```

pg_get_schemas(schemaPattern)

Description: Queries schemas in the current catalog. The **schemaPattern** parameter can be a schema name or a wildcard character represented by %. If **schemaPattern** is empty (using single quotes instead of parameters), it returns all schemas.

Return type: text.

```
select pg_get_schemas('');
    pg_get_schemas
----------------------
 test_orc
 test1_orc
 test_parquet
(3 row)

select pg_get_schemas('%orc');
    pg_get_schemas
----------------------
 test_orc
 test1_orc
(2 row)

select pg_get_schemas('test_orc');
    pg_get_schemas
----------------------
 test_orc
(1 row)
```

pg_get_tables(schemaPattern, tablePattern, tableTypes)

Description: Queries for information about tables that meet conditions in the current catalog. The **schemaPattern** parameter can be used to specify the schema name or the wildcard represented by %. The **tablePattern** parameter can be used to specify the table name or the wildcard represented by %. The **tableTypes** parameter can be used to specify the table type. The table types are **MANAGED_TABLE**, **EXTERNAL_TABLE**, **VIRTUAL_VIEW**, **MATERIALIZED_VIEW**, and **DICTIONARY_TABLE**. If these three parameters are empty (using single quotes to substitute the parameters), it returns all catalogs, schemas, table names, table types, and descriptions.

Return type: The row set consisting of the following fields is returned:

 Copyright © Huawei Technologies Co., Ltd.

- catalog_name
- schema_name
- table_name
- table_type
- description

```
select pg_get_tables('','','');
    pg_get_tables
----------------------
 (catalog,schema_parquet,table1,MANAGED_TABLE,"")
 (catalog,schema_parquet,table2,MANAGED_TABLE,"")
 (catalog,schema_orc,table1,MANAGED_TABLE,"")
(3 row)

select pg_get_tables('%parquet','','MANAGED_TABLE');
    pg_get_tables
----------------------
 (catalog,schema_parquet,table1,MANAGED_TABLE,"")
 (catalog,schema_parquet,table2,MANAGED_TABLE,"")
(2 row)
```

pg_get_columns(schemaPattern, tableNamePattern, columnNamePattern)

Description: Queries the column information that meets the conditions in the current catalog. The **schemaPattern** parameter specifies the schema name or the wildcard character %. The **tableNamePattern** parameter specifies the table name or the wildcard character %. The **columnNamePattern** parameter specifies the column name or the wildcard character %.

Return type: The row set consisting of the following fields is returned:

- catalog_name
- schema_name
- table_name
- column_name
- origintype
- atttypid
- atttypmod
- typename
- typtype
- attnum
- description
- attnotnull
- description

```
select pg_get_columns('schema_parquet','customer','');
    pg_get_columns
----------------------
 (catalog,schema_parquet,customer,c_custkey,bigint,20,-1,int8,b,1,,f)
 (catalog,schema_parquet,customer,c_name,string,25,-1,text,b,2,,f)
 (catalog,schema_parquet,customer,c_address,string,25,-1,text,b,3,,f)
 (catalog,schema_parquet,customer,c_nationkey,bigint,20,-1,int8,b,4,,f)
 (catalog,schema_parquet,customer,c_phone,"char(15)",1042,19,bpchar,b,5,,f)
 (catalog,schema_parquet,customer,c_acctbal,"decimal(15,2)",1700,983046,numeric,b,6,,f)
 (catalog,schema_parquet,customer,c_mktsegment,"char(10)",1042,14,bpchar,b,7,,f)
 (catalog,schema_parquet,customer,c_comment,string,25,-1,text,b,8,,f)
(8 row)
```

 Copyright © Huawei Technologies Co., Ltd.

pg_get_partition_keys(schema, table)

Description: Retrieves partition key information for a specified schema and table within the current catalog. For non-partitioned tables, it returns an empty result set.

Return type: A rowset containing the following fields:

- **table**: table name.
- **partition_key**: partition key information.

```
select * from pg_get_partition_keys('test', 'iceberg_table');
    table     |  partition_key
--------------+-----------------
 iceberg_table | bucket(16,val)
 iceberg_table | truncate(10,id)
 iceberg_table | c3
 iceberg_table | year(ts)
(4 rows)
```

pg_get_partitions(table)

Description: Queries data partition details for a specified table under the current schema in the catalog.

Return type: A rowset containing the following fields:

- **partition_name**: details of the data partition.
- **location**: storage location corresponding to the data partition.

```
select * from pg_get_partitions('iceberg_table');
     partition_name      |                                                 location
-------------------------
+-------------------------------------------------------------------------------------------------------------------------------
 c3=1/data=hell/ts=2025   | obs://test/1b974a89-32b1-4b6b-8e14-548bba1b5007/test1/
iceberg_test_part_expr/data/c3=1/data_trunc=hell/ts_year=2025/
 c3=20/data=he/ts=2024    | obs://test/1b974a89-32b1-4b6b-8e14-548bba1b5007/test1/
iceberg_test_part_expr/data/c3=20/data_trunc=he/ts_year=2024/
 c3=100/data=zhan/ts=2025 | obs://test/1b974a89-32b1-4b6b-8e14-548bba1b5007/test1/
iceberg_test_part_expr/data/c3=100/data_trunc=zhan/ts_year=2025/
 c3=6/data=hell/ts=2023   | obs://test/1b974a89-32b1-4b6b-8e14-548bba1b5007/test1/
iceberg_test_part_expr/data/c3=6/data_trunc=hell/ts_year=2023/
 c3=60/data=hell/ts=2022  | obs://test/1b974a89-32b1-4b6b-8e14-548bba1b5007/test1/
iceberg_test_part_expr/data/c3=60/data_trunc=hell/ts_year=2022/
(5 rows)
```

11.16.3 System Function Information Functions

pv_builtin_functions()

Description: Queries information about system built-in functions.

Return type: record

Example:

```
select * from pv_builtin_functions() limit 1;
 proname  | pronamespace | proowner | prolang | procost | prorows | provariadic | protransform | proisagg |
proiswindow | prosecdef | proleakproof | proisstrict | proretset | provolatile | pronargs | pronargdefaults | pror
ettype | proargtypes | proallargtypes | proargmodes | proargnames | proargdefaults |  prosrc  | probin |
proconfig | prodefaultargpos | fencedmode | proshippable | propackage | oid
----------+--------------+----------+---------+---------+---------+-------------+--------------+----------+-------------
+-----------+--------------+-------------+-----------+-------------+----------+-----------------+-----
-------+-------------+----------------+-------------+-------------+----------------+----------+--------+-----------
```

 Copyright © Huawei Technologies Co., Ltd.

```
+------------------+------------+--------------+------------+-----
 byteaout |          11 |      10 |      12 |       1 |      0 |          0 | -           | f       | f          | f        | f
| t          | f        | i          |        1 |             0 |
  2275 | 17        |             |           |           |             | byteaout |       |         |              | f
|            | f         |  31
(1 row)
```

11.16.4 Status Information Functions

pgxc_query_wait_status(tid)

Description: Displays the waiting status between execution unit threads of a specified session.

Return type: record

The returned information is as follows:

Table 11-13 pgxc_query_wait_status(tid) columns

Column	Type	Description
actor_name	text	Actor name.
actor_id	text	Actor ID.
actor_ip	text	Actor IP address.
db_name	text	Database name.
thread_name	text	Thread name.
query_id	bigint	Query ID.
tid	bigint	ID of the current thread.
lwtid	integer	Lightweight thread ID of the current thread.
ptid	integer	Parent thread of the streaming thread.
tlevel	integer	Level of the streaming thread.
smpid	integer	ID of a parallel thread.
wait_status	text	Waiting status of the current thread.
wait_event	text	If **wait_status** is **acquire lock**, **acquire lwlock**, or **wait io**, this column describes the specific lock, lightweight lock, and I/O information, respectively. Otherwise, this column is empty.

Example:

```
SELECT * FROM pgxc_query_wait_status(139775106588720);
  actor_name  |                 actor_id                 |  actor_ip   | db_name | thread_name |     query_id      |
tid       |  lwtid  | ptid | tlevel | smpid | wait_status | wait_event
```

 Copyright © Huawei Technologies Co., Ltd.

```
--------------+---------------------------------------------+-------------+---------+-------------+------------------
+-----------------+---------+------+--------+-------+-------------+-----------
-
 coordinator1 | runtime-c7010000-0000-4000-8bad-1509e41e3b67 | 10.42.XX.XX |          | gsql        |
72339069014638600 | 140406720987184 | 4194244 |      |      0 |     0 | none        |
(1 row)
```

yr_api_status()

Description: Displays statistics about YuanRong API calls during a session.

Return type: record

The function returns the following information:

Table 11-14 yr_api_status() columns

Column	Type	Description
actor_name	text	Actor name.
api	text	API name.
total_call	bigint	Total number of calls.
failed_call	bigint	Number of failed calls.
avg_call_during	bigint	Average API invocation duration. Unit: millisecond.
min_call_during	bigint	Minimum calling duration. Unit: millisecond.
max_call_during	bigint	Maximum calling duration. Unit: millisecond.
total_data	bigint	Total data volume. Unit: byte.
avg_data_rate	bigint	Average data processing rate. Unit: byte/ms.
min_data_rate	bigint	Minimum data processing rate. Unit: byte/ms.
max_data_rate	bigint	Maximum data processing rate. Unit: byte/ms.

Example:

```
SELECT * FROM yr_api_status();
  actor_name  |         api          | total_call | failed_call | avg_call_during | min_call_during | max_call_during |
total_data | avg_data_rate | min_data_rate | max_data_rate
--------------+----------------------+------------+-------------+-----------------+-----------------+-----------------
+------------+---------------+---------------+---------------
```

 Copyright © Huawei Technologies Co., Ltd.

```
 coordinator1 | YR::Invoke         |        12 |         0 |             |             |             |         |
|             |
 coordinator1 | YR::Get            |        11 |         0 |          13 |           1 |         134 |
|             |             |
 coordinator1 | YR::CreateProducer |         0 |         0 |             |             |             |
|             |             |
 coordinator1 | YR::Subscribe      |         0 |         0 |             |             |             |         |
|             |
 coordinator1 | YR::Send           |         0 |         0 |           0 |           0 |           0 |       0 |
0 |           0 |           0
 coordinator1 | YR::Receive        |         0 |         0 |           0 |           0 |           0 |       0 |
0 |           0 |           0
 coordinator1 | YR::KV::Set        |         0 |         0 |           0 |           0 |           0 |       0 |
0 |           0 |           0
 coordinator1 | YR::KV::Get        |         0 |         0 |           0 |           0 |           0 |       0 |
0 |           0 |           0
 coordinator1 | YR::KV::Del        |         0 |         0 |           0 |           0 |           0 |         |
|             |
(9 rows)
```

11.17 System Management Functions

11.17.1 Memory Management Functions

pgxc_total_memory_detail()

Description: Displays the memory usage of all backend actor nodes.

Return type: record

The function returns the following information:

Table 11-15 Output columns of pgxc_total_memory_detail()

Column	Type	Description
nodename	text	Node name.

Column	Type	Description
memorytype	text	Memory name. The options are: • **max_process_memory**: memory occupied by an Aura endpoint. • **process_used_memory**: memory occupied by an Aura process • **max_dynamic_memory**: maximum dynamic memory. • **dynamic_used_memory**: used dynamic memory. • **dynamic_peak_memory**: dynamic peak value of the memory. • **dynamic_used_shrctx**: maximum dynamic shared memory context. • **dynamic_peak_shrctx**: dynamic peak value of the shared memory context. • **max_shared_memory**: maximum shared memory. • **shared_used_memory**: used shared memory. • **max_cstore_memory**: maximum memory allowed by column storage. • **cstore_used_memory**: used memory of the column-store table. • **other_used_memory**: other used memory size. • **mmap_used_memory**: memory used by mmap.
memorymbytes	integer	Size of the used memory, in MB.

shared_memory_context(actor_name cstring)

Description: Queries statistics about shared memory contexts in a specified actor process.

Parameter: **actor_name**, indicating the actor name.

Return type: record

Example:

```
select * from shared_memory_context('coordinator1') order by totalsize desc limit 10;
             contextname             | level |              parent               | totalsize | freesize | usedsize
-------------------------------------+-------+-----------------------------------+-----------+----------+----------
 Query resource track memory context |     2 | Workload manager memory context |  67174520 |    64320 |
67110200
```

 Copyright © Huawei Technologies Co., Ltd.

```
 RawMemory                      |     1 | ProcessMemory                 |  23525840 |        0 | 23525840
 builtin proc name Lookup Table |     2 | builtin_procGlobalMemoryContext |    425984 |   261152 |
164832
 ProcessMemory                  |     0 |                               |    196608 |    76992 |   119616
 builtin pro Oid Lookup Table   |     2 | builtin_procGlobalMemoryContext |    196608 |    41664 |   154944
 PoolerMemoryContext            |     1 | ProcessMemory                 |     73728 |    56832 |    16896
 Node Pool                      |     3 | PoolerCoreContext             |     73640 |    25888 |    47752
 ThreadGroup Hash Table         |     1 | ProcessMemory                 |     65536 |    39936 |    25600
 Aio Sub Memory Context 6       |     2 | Aio Memory Context            |     65536 |    63872 |     1664
 Aio Sub Memory Context 5       |     2 | Aio Memory Context            |     65536 |    63872 |     1664
(10 rows)
```

shared_memory_context_chunk(actor_name cstring, contextname char(64))

Description: Queries information about all chunks applied for by the shared memory context named **contextname** of a specified actor.

Parameters:

- **actor_name**: actor name.
- **contextname**: memory context name.

Return type: record

Example:

```
select * from shared_memory_context_chunk('executor0_es_group', 'StreamInfoContext');
    parent     | level |      file_name       | line_number | chunk_size | request_count | total_size
---------------+-------+----------------------+-------------+------------+---------------+------------
 ProcessMemory |     1 | mcxt.cpp             |        1010 |       8192 |             1 |       8192
 ProcessMemory |     1 | streamThreadPool.cpp |          35 |       1024 |             1 |       1024
(2 rows)
```

pgxc_session_memory_detail()

Description: Displays the thread memory usage of the corresponding actor or all backend actor nodes by MemoryContext.

The memory context **TempSmallContextGroup** records the summary information of all memory contexts in the current thread where the **totalsize** column is fewer than 8,192 bytes, and the count of memory contexts is recorded in the **usedsize** column. Therefore, in the results, the **totalsize** and **freesize** columns in the **TempSmallContextGroup** memory context represent the aggregate sum of all memory contexts in the thread where **totalsize** is fewer than 8,192 bytes, and the **usedsize** column indicates the count of such memory contexts.

Parameter descriptions:

- **pgxc_session_memory_detail()**: no parameters, queries the statistical information of memory contexts for all threads in the CN and all DN processes.
- **pgxc_session_memory_detail(actor_name)**, with the input parameter **actor_name**, outputs the statistical information of memory contexts for all threads in the specified CN or DN process.

Return type: record

The function returns the following information:

 Copyright © Huawei Technologies Co., Ltd.

Table 11-16 pgxc_session_memory_detail columns

Column	Type	Description
sessid	text	Thread start time + thread ID (string format: timestamp.threadid).
sesstype	text	Thread name.
contextname	text	Memory context name.
level	smallint	Level of the current context within the overall memory context hierarchy.
parent	text	Parent memory context name.
totalsize	bigint	Total memory size of the current memory context, in bytes.
freesize	bigint	Total size of freed memory in the current memory context, in bytes.
usedsize	bigint	Total size of used memory in the current memory context, in bytes. For the **TempSmallContextGroup** memory context, this column indicates the count statistic.

session_memory_context_chunk(actor_name cstring, tid int64, contextname char(64))

Description: Queries information about all chunks allocated by a specific memory context created by a thread on a specified actor.

Parameter descriptions:

- **actor_name**: actor name
- **tid**: thread ID.
- **contextname**: memory context name.

Return type: record

Example:

```
select * from session_memory_context_chunk('executor0_es_group', 139736895414560, 'Node Pool') order
by total_size desc;
      parent       | level |  file_name   | line_number | chunk_size | request_count | total_size
-------------------+-------+--------------+-------------+------------+---------------+------------
 PoolerCoreContext |     3 | dynahash.cpp |         147 |       2048 |             1 |       2048
 PoolerCoreContext |     3 | dynahash.cpp |         147 |        256 |             1 |        256
 PoolerCoreContext |     3 | dynahash.cpp |         147 |      32584 |             1 |      32584
 PoolerCoreContext |     3 | dynahash.cpp |         147 |       4096 |             3 |      12288
(4 rows)
```

jemalloc_heap_stats(actor_name cstring)

Description: Queries the memory statistics provided by jemalloc.

Parameter: **actor_name**, indicating the actor name

 Copyright © Huawei Technologies Co., Ltd.

Return type: record

Example:

```
select * from jemalloc_heap_stats('coordinator1');
  actor_name  |
  stats
--------------
+-----------------------------------------------------------------------------------------------------------------------------------
---
-----------------------------------------------------------------------------------------------------------------------------------
----------------
 coordinator1 | ___ Begin jemalloc statistics ___

              | Version: "5.2.1-6-gd04ec97dec5e84ac24f32c29053aa94479402b6d"

              | Build-time option settings

              |   config.cache_oblivious: true

              |   config.debug: false

              |   config.fill: true

              |   config.lazy_lock: false

              |   config.malloc_conf:
"tcache:true,background_thread:true,percpu_arena:phycpu,dirty_decay_ms:1000,muzzy_decay_ms:1000"

              |   config.opt_safety_checks: false

              |   config.prof: false

              |   config.prof_libgcc: false

              |   config.prof_libunwind: false

              |   config.stats: true

              |   config.utrace: false
```

jemalloc_heap_profiling(actor_name cstring, type int)

Description: Samples memory allocations such as **malloc** in the Aura actor process and outputs the sampling results.

Table 11-17 jemalloc_heap_profiling parameters

Parameter	Type	Description
actor_name	String	Actor name
type	Integer	**1**: Enables memory tracing and records call stack information such as **malloc**. **0**: Disables memory tracing and returns the sampled call stack information such as **malloc**.

Return type:

- **actor_name**: actor name

 Copyright © Huawei Technologies Co., Ltd.

- **profiling**: memory allocation sampling information.

Notes:

- If the failure prompt reads "Memory profiling failed, check if $MALLOC_CONF contains 'prof:true'.", it means the environment variable **MALLOC_CONF=prof:true** was not set because the jemalloc library with memory sampling capability was not used. Add the **-j** option to the parameters when starting the CN using **gov_client**.
- If the failure prompt reads "Type %d is not supported. The valid range is 0-1.", it indicates an invalid user input. The correct values are **0** or **1**.

Outputting Memory Call Stack Information

Procedure:

Step 1 Output memory call stack information and generate trace files in the directory where the Aura process is located.

```
SELECT profiling FROM jemalloc_heap_profiling('coordinator1', 0);
```

Step 2 Save the query results as trace files.

Step 3 Use the jeprof tool provided by jemalloc to parse the 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

- Parsing memory call stack information requires analyzing the Aura source code, and the trace files should be sent back to development engineers for further analysis.
- Analyzing trace files requires the jeprof tool, which is generated by Jemalloc. Generating PDF call graphs depends on the Perl environment and requires installing the GraphViz tool compatible with the operating system.

----End

Examples

Run **gov_client -j** to restart the database.

Enable memory tracing during database runtime.

```
postgres=# SELECT profiling FROM jemalloc_heap_profiling('coordinator1', 1);
```

Disable memory tracing during database runtime.

```
postgres=# SELECT profiling FROM jemalloc_heap_profiling('coordinator1', 0);
```

Save the query results into trace files and export them in text or PDF format.

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 >prof.txt
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace file 1 > prof.pdf
```

11.18 Database Object Functions

 Copyright © Huawei Technologies Co., Ltd.

11.18.1 Database Object Size Functions

pg_table_size(regclass)

Description: Retrieves the table size and calculates the total size of all files in the table directory.

Return type: int64.

pg_table_files(regclass)

Description: Retrieves the table size and returns the paths and sizes of all files in the table directory.

Return type:

- **file_path**: file path, which is of the text type.
- **file_name**: file name, which is of the text type.
- **file_size**: file size, which is of the int64 type.
- **file_etag**: file etag, which is of the text type.
- **file_last_modify_time**: last file modification time, which is of the int type.

pg_size_pretty(bigint)

Description: Converts the byte-calculated size into a human-readable size.

Return type: text.

pg_size_pretty(numeric)

Description: Converts the numerically represented byte-calculated size into a human-readable size.

Return type: text.

Note: **pg_size_pretty** formats the results of other functions into a human-readable format, using KB, MB, GB, or TB as appropriate.

11.19 Statistics Functions

pg_stat_get_activity(integer)

Description: Returns a record of a background process with a specific PID. When the parameter is NULL, it returns records of all active background processes. The result is a subset of the **PG_STAT_ACTIVITY** view, excluding the **connection_info** column.

Return type: SETOF record.

gs_stack()

Description: Obtains the stack information of a CN or DN process. The input parameters **actor_name** and **tid** need to be obtained through views or other

 Copyright © Huawei Technologies Co., Ltd.

functions. **lwtid** is not supported. **gs_stack()** provides a more flexible and faster method for obtaining stacks than the **gdb** and **gstack** commands. It is recommended that administrators use this tool only during fault analysis and locating. It is not recommended that this tool be used as a routine monitoring tool.

Return value columns:

- **actor_name**: name of the actor where the thread is located.
- **tid**: thread ID.
- **lwtid**: lightweight thread ID.
- **stack**: stack information about the **tid** or **lwtid** thread.

The **gs_stack()** function can be used in the following ways:

- Method 1: If the input parameter of the **gs_stack()** function is empty, the stacks of all threads on the CNs and DNs are printed.

  NOTE

  Obtaining the running stacks of all threads on CNs and all DNs takes a long time, which affects normal service running. Therefore, you are advised not to use this function frequently.

```
select * from gs_stack();
  actor_name  |       tid       | lwtid
|



stack



--------------+-----------------+--------
+-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
----
----------------------------------
 coordinator1 | 106927505866816 | 186762 | GsStackBacktrace(int) +
0x1a


                          +
          |               |      |
0x7ffb45f6d1f0



                          +
          |               |      | __poll +
0x4f


                          +
          |               |      | poll +
0xa0


                          +
```

```
          |              |       | poll(pollfd*, unsigned long, int, void (*)()) +
0x19e

                          +
          |              |       | ServerLoop() +
0x570

                          +
          |              |       | PmStartupThreads() +
0x167

                          +
          |              |       | PostmasterMain(int, char**) +
0x22f

          |              |       |
 coordinator1 | 106927505998400 | 186773 | GsStackBacktrace(int) +
0x1a

                          +
          |              |       |
0x7ffb45f6d1f0

                          +
          |              |       |
0x7ffb45fb41aa

                          +
          |              |       | pthread_cond_clockwait +
0x1e2

                          +
          |              |       | std::__condvar::wait_until(std::mutex&, int, timespec&) +
0x39
```

- Method 2: Set the input parameter of the **gs_stack(actor_name)** function to the specified **actor_name**, indicating that the stacks of all threads on the actor named **actor_name** are printed.

  NOTE

  - **actor_name** must be the name of a CN or DN. Otherwise, the error message "invalid actor_name" is displayed.
  - The value of **actor_name** can be obtained from other views or functions (for example, the **pgxc_query_wait_status** function).

```
select * from gs_stack('coordinator1');
  actor_name  |       tid       | lwtid
|



stack



--------------+-----------------+--------
```

 Copyright © Huawei Technologies Co., Ltd.

```
+-----------------------
```

```
|

stack

--------------+-----------------+--------
+-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-----------------------------------------------
 coordinator1 | 106927505998400 | 186773 | GsStackBacktrace(int) +
0x1a

                                        +
          |             |       |
0x7ffb45f6d1f0

                                        +
          |             |       |
0x7ffb45fb41aa

                                        +
          |             |       | pthread_cond_clockwait +
0x1e2

                                        +
          |             |       | std::__condvar::wait_until(std::mutex&, int, timespec&) +
0x39

                                        +
          |             |

(1 row)
```

gs_wlm_session_statistics()

Description: Retrieves monitoring information for SQL statements currently being executed. It does not capture monitoring information for completed SQL statements. Each row in the returned results indicates one statement.

Fields returned by the function:

Column	Type	Description
session_id	text	Session ID of a statement.
statement_id	text	Statement ID.
start_time	timestamp with time zone	Start time of a statement.
duration	bigint	Statement running duration. Unit: ms.
estimate_total_time	bigint	Estimated statement running duration. Unit: ms.
estimate_left_time	bigint	Estimated remaining running duration of a statement. Unit: ms.

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Description
cn_actor_info	text	Serverless information of a CN.
dn_actor_info	text	Serverless information of a DN.
startup_coordinator_duration	bigint	Duration to start a CN. Unit: ms.
create_session_duration	bigint	Duration to create a session. Unit: ms.
enqueue_statement_duration	bigint	Duration to add a statement to a task queue. Unit: ms.
queue_duration	bigint	Queuing duration of a statement in a task queue. Unit: ms.
write_resultset_duration	bigint	Duration to write a statement to a result set. Unit: ms.
queryid	bigint	ID of a debug query.
schemaname	text	Schema name when a statement is executed.
query	text	Statement.
query_plan	text	Explain information.
tid	bigint	Thread ID.
parse_time	bigint	Time from parsing to execution start. Unit: ms.
estimate_memory	int	Estimated memory of a statement. Unit: MB.
min_dn_time	bigint	Minimum DN duration. Unit: ms.
max_dn_time	bigint	Maximum DN duration. Unit: ms.
average_dn_time	bigint	Average DN duration. Unit: ms.
dn_time_skew_percent	int	Skew percentage of DN duration. Unit: %.

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Description
dn_min_cpu_time	bigint	Minimum DN CPU duration. Unit: ms.
dn_max_cpu_time	bigint	Maximum DN CPU duration. Unit: ms.
dn_average_cpu_time	bigint	Average DN CPU duration. Unit: ms.
dn_total_cpu_time	bigint	Total DN CPU duration. Unit: ms.
dn_cpu_skew_percent	int	Skew percentage of DN CPU duration. Unit: %.
dn_min_peak_memory	int	Minimum peak memory of all DNs. Unit: MB.
dn_max_peak_memory	int	Maximum peak memory of all DNs. Unit: MB.
dn_average_peak_memory	int	Average peak memory of all DNs. Unit: MB.
dn_memory_skew_percent	int	Memory skew rate. Unit: %.
dn_spill_num	int	Number of DNs with disk spills.
dn_spill_count	int	Number of DNs with disk spills.
dn_min_spill_size	bigint	Minimum spill size. Unit: MB.
dn_max_spill_size	bigint	Maximum spill size. Unit: MB.
dn_average_spill_size	bigint	Average spill size. Unit: MB.
dn_spill_skew_percent	int	Spill skew rate. Unit: %.
dn_write_disk_total_size	bigint	Total size of data written to disks on a DN. Unit: MB.
dn_write_disk_max_size	bigint	Maximum amount of data written to disks on a DN. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Description
dn_write_disk_min_size	bigint	Minimum amount of data written to disks on a DN. Unit: MB.
dn_spill_obs_total_size	bigint	Total amount of data spilled to OBS by a DN. Unit: MB.
dn_spill_obs_max_size	bigint	Maximum amount of data spilled to OBS by a DN. Unit: MB.
dn_spill_obs_min_size	bigint	Minimum amount of data spilled to OBS by a DN. Unit: MB.
dn_write_obs_total_size	bigint	Total amount of data directly written to OBS by a DN. Unit: MB.
dn_write_obs_max_size	bigint	Maximum amount of data directly written to OBS by a DN. Unit: MB.
dn_write_obs_min_size	bigint	Minimum amount of data directly written to OBS by a DN. Unit: MB.
cn_spill_count	int	Number of spills from a CN.
cn_spill_size	bigint	Total size of data spilled from a CN. Unit: MB.
cn_spill_obs_size	bigint	Amount of data spilled to OBS by a CN. Unit: MB.
cn_write_obs_size	bigint	Amount of data directly written to OBS by a CN. Unit: MB.
cn_write_obs_disk_size	bigint	Amount of data spilled to local disks by a CN. Unit: MB.
disk_cache_hit_ratio	numeric	Disk cache hit rate.
disk_cache_disk_read_size	bigint	Read size of disk cache. Unit: byte.

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Description
disk_cache_disk_write_size	bigint	Write size of disk cache. Unit: byte.
disk_cache_remote_read_size	bigint	Remote read size of disk cache. Unit: byte.
disk_cache_remote_read_time	bigint	Count of remote reads from disk cache. Unit: times.
vfs_scan_bytes	bigint	Bytes scanned by the OBS file system. Unit: byte.
vfs_remote_read_bytes	bigint	Bytes remotely read by the OBS file system. Unit: byte.
disk_cache_load_time	bigint	Local load time of disk cache. Unit: µs.
disk_cache_conflict_count	bigint	Hash collision count of disk cache blocks. Unit: times.
disk_cache_error_count	bigint	Error count of disk cache. Unit: times.
disk_cache_error_code	bigint	Error code of disk cache.
obs_io_req_avg_rtt	bigint	Average round-trip time of OBS I/O requests. Unit: µs.
obs_io_req_avg_latency	bigint	Average latency of OBS I/O requests. Unit: µs.
obs_io_req_latency_gt_1s	bigint	Count of OBS I/O request delays exceeding 1s. Unit: times.
obs_io_req_latency_gt_10s	bigint	Count of OBS I/O request delays exceeding 10s. Unit: times.
obs_io_req_count	bigint	Number of OBS I/O requests. Unit: times.
obs_io_req_retry_count	bigint	Number of OBS I/O request retries. Unit: times.
obs_io_req_rate_limit_count	bigint	Rate limit count of OBS I/O requests. Unit: times.

 Copyright © Huawei Technologies Co., Ltd.

pg_get_actor_stat()

Description: Obtains statistics about some monitoring items, including creating a CN, starting a CN, creating a session, adding a statement to a queue, executing a statement, writing a result set, obtaining the execution status, and retrieving the execution result. The statistics are recorded each time one of these events occurs.

Return type:

Column	Type	Description
event_name	text	Name of a monitoring item.
object_name	text	Monitoring item ID.
start_time	timestamp with time zone	Start time of a monitoring item.
end_time	timestamp with time zone	End time of a monitoring item.
comment	text	Content of the comment.

obs_get_access_label()

Description: Obtains the permissions of a specified OBS path.

Return type:

Column	Type	Description
obs_path	text	OBS path
access_label	text	Permissions of the OBS path

read_external_table_file()

Description: Parses external table files in ORC or Parquet format. If a file cannot be queried using SELECT due to version differences or data producer issues, you can use this function for diagnosis and analysis.

The input parameters are as follows.

Column	Type	Description
tablename	text	Name of the target foreign table, which must be in Parquet, ORC, or Iceberg format

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Description
filename	text	Target file path, which must end with .orc or .parquet
target	text	Information of the file to be obtained. The value can be **meta/m** (metadata), **statistics/s** (statistics), **contents/c** (file content), **meta -v/m -v** (detailed metadata), or **statistics -i/s -i** (statistics including row indexes).
num	int	Used together with the third parameter to specify whether to obtain all (0) or part of the information

Return type: text

Example:

```
select * from read_external_table_file('test0508','/******/30f16dfe-a7dc-487b-a285-752aeaebde72.parquet','m
-v',0);
                                        read_external_table_file
-----------------------------------------------------------------------------------------------------------------
{                                                                                                               +
  "FileName": "/******/30f16dfe-a7dc-487b-a285-752aeaebde72.parquet",                                           +
  "Version": "2.6",                                                                                             +
  "CreatedBy": "parquet-cpp-arrow version 17.0.0",                                                              +
  "TotalRows": "1",                                                                                             +
  "NumberOfRowGroups": "1",                                                                                     +
  "NumberOfRealColumns": "1",                                                                                   +
  "NumberOfColumns": "1",                                                                                       +
  "Columns": [                                                                                                  +
    {                                                                                                           +
     "Id": "0",                                                                                                 +
     "Name": "a",                                                                                               +
     "PhysicalType": "INT32",                                                                                   +
     "ConvertedType": "INT_32",                                                                                 +
     "LogicalType": {"Type": "Int", "bitWidth": 32, "isSigned": true}                                           +
    }                                                                                                           +
   ],                                                                                                           +
  "RowGroups": [                                                                                                +
    {                                                                                                           +
      "Id": "0",                                                                                                +
      "Rows": "1",                                                                                              +
      "TotalBytes": "44",                                                                                       +
      "TotalCompressedBytes": "48",                                                                             +
      "FileOffset": "4",                                                                                        +
      "ColumnChunks": [                                                                                         +
        {"Id": "0",                                                                                             +
         "Values": "1",                                                                                         +
         "Compression": "SNAPPY",                                                                               +
         "Encodings": "PLAIN RLE RLE_DICTIONARY ",                                                              +
         "UncompressedSize": "44",                                                                              +
         "CompressedSize": "48",                                                                                +
        }                                                                                                       +
      ]                                                                                                         +
    }                                                                                                           +
  ]                                                                                                             +
}                                                                                                               +

(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

pg_get_statistics()

Description: Queries table statistics.

Input parameter description: table name (A schema can be specified.)

The returned value descriptions are as follows.

Column	Type	Description
column_name	text	Column name
null_fraction	float4	Proportion of NULL values. The value ranges from 0 to 1. For example, **0.25** indicates that 25% of the values in the column are **NULL**.
avg_width	int4	Average column width in bytes. For example, the width is 4 for int4 and depends on the content for text.
distinct	float4	Number of distinct values. A positive number indicates an absolute quantity, and a negative number indicates the proportion of the number of distinct values to the total number of rows. For example, –0.5 indicates that about 50% of the values are different from each other.
mcv_values	text	List of most common values. If no data is available, NULL is returned.
mcv_freqs	text	List of occurrence frequencies of the most common values. Each frequency matches a value in **mcv_values**. If no data is available, NULL is returned.
histogram_bounds	text	List of boundary values of a histogram. The column value range is divided into equal-frequency buckets. If no data is available, NULL is returned.

pg_get_partition_statistics()

Description: Queries partition statistics.

The input parameters are as follows.

 Copyright © Huawei Technologies Co., Ltd.

Column	Type	Mandatory	DefaultValue	Description
table_name	text	Yes	-	Name of a partitioned table (A schema can be specified.)
partition_values	ARRAY	Yes	-	Array of partition key values
spec_id	int8	No	0	Partition ID of an Iceberg table

The returned value descriptions are as follows.

Column	Type	Description
column_name	text	Column name
null_fraction	float4	Proportion of NULL values. The value ranges from 0 to 1. For example, **0.25** indicates that 25% of the values in the column are **NULL**.
avg_width	int4	Average column width in bytes. For example, the width is 4 for int4 and depends on the content for text.
distinct	float4	Number of distinct values. A positive number indicates an absolute quantity, and a negative number indicates the proportion of the number of distinct values to the total number of rows. For example, –0.5 indicates that about 50% of the values are different from each other.
mcv_values	text	List of most common values. If no data is available, NULL is returned.
mcv_freqs	text	List of occurrence frequencies of the most common values. Each frequency matches a value in **mcv_values**. If no data is available, NULL is returned.
histogram_bounds	text	List of boundary values of a histogram. The column value range is divided into equal-frequency buckets. If no data is available, NULL is returned.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
1. SELECT * FROM pg_get_statistics('my_table');

 column_name | null_fraction | avg_width | distinct | mcv_values  | mcv_freqs       | histogram_bounds
-------------+---------------+-----------+----------+-------------+-----------------+------------------
 id          |             0 |         4 |       -1 | NULL        | NULL            | 1, 50, 100
 name        |           0.1 |        16 |     -0.5 | Alice, Bob  | 0.3, 0.2        | NULL
 status      |             0 |         4 |        3 | active      | 0.6             | NULL

(1 row)

2. SELECT * FROM pg_get_partition_statistics('iceberg_table', ARRAY['2026-01-01', '3'], 0);

column_name | null_fraction | avg_width | distinct | mcv_values | mcv_freqs | histogram_bounds
-------------+---------------+-----------+----------+------------+-----------+------------------
 x           |             0 |         4 |       -1 | NULL       | NULL      | 1, 2, 3
 a           |             0 |        10 |     -0.8 | 123456789  | 0.4       | NULL
 b           |             0 |         4 |        1 | NULL       | NULL      | NULL

(1 row)
```

11.20 Iceberg Table Service Functions

iceberg_expire_snapshots

Description: Each commit in Iceberg generates a new snapshot while retaining old data and metadata for snapshot isolation and time travel. Expire snapshots can be used to clean up old snapshots that are no longer needed and data files only included in unnecessary snapshots to improve query and cost efficiency.

Notes:

- Clearly define the cleaning and retention policy for old snapshots. After executing the cleanup of old snapshots, the deleted old snapshot data will no longer be accessible through regular queries. If a running query relies on the soon-to-expire snapshot data, it may lead to query failure or inaccurate results.
- When executing **expire_snapshots**, conflicts may arise between conditions such as **retain_last**, **older_than**, and **snapshot_ids**. Iceberg does not strictly adhere to the specified conditions but instead aims to retain as many snapshots as possible.

Syntax:

```
iceberg_expire_snapshots('schema.table','older_than', 'clean_expired_metadata')
iceberg_expire_snapshots('schema.table','older_than', retain_last, 'snapshot_ids', 'clean_expired_metadata')
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table where operations are to be performed

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Default Value	Mandatory	Description
older_than	timestamp	5 days ago	Yes	Snapshots exceeding this time limit will be deleted. Note: You need to input the specific time zone to avoid misunderstandings. It can be specified at input or through **set timezone=xxx**.
retain_last	int	1	No	How many snapshots to retain.
snapshot_ids	String	None	No	Multiple snapshot IDs can be provided, separated by semicolons (;).
clean_expired_metadata	boolean	None	No	If this parameter is set to **true**, schemas and partition specifications that are no longer referenced will be additionally cleaned up.

Return values:

Parameter	Description
deleted_data_files_count	Number of deleted data files.
deleted_position_delete_files_count	Number of deleted position delete files.
deleted_equality_delete_files_count	Number of deleted equality delete files.
deleted_manifest_files_count	Number of deleted manifest files.
deleted_manifest_lists_count	Number of deleted snapshot files.
deleted_statistics_files_count	Number of deleted statistics files.

iceberg_rewrite_manifests

Description: The Iceberg table tracks each data file. The more data files, the more metadata stored in the snapshot file. **rewrite manifests** can re-write the table's manifest to optimize scan planning.

 Copyright © Huawei Technologies Co., Ltd.

Notes: This operation generates a new manifest file, potentially affecting ongoing queries and write operations, leading to metadata conflicts. Avoid concurrent execution with other operations.

Syntax:

```
iceberg_rewrite_manifests(table)
iceberg_rewrite_manifests(table, options)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table where operations are to be performed
options	array	None	No	Operational options. Example: **'{spec_id=2}'**

The **options** parameter supports the following option:

Parameter	Type	Description
spec_id	int32	Rewrites only the manifest file associated with the specified **spec_id**. Valid values can be queried from **iceberg_partition_specs**.

Return values:

Parameter	Description
rewritten_manifests_count	Number of rewritten manifest files.
added_manifests_count	Number of new manifest files.

iceberg_remove_orphanfiles

Description: Files not referenced by any Iceberg metadata file are termed orphaned files. Such files might result from failed transactions, unsuccessful concurrent commits, failures in old snapshot cleanup, or metadata cleanup errors, and can generally be deleted.

Notes: Ensure no concurrent write operations exist. During deletion, if there is a running write task, it might mistakenly delete files being generated (uncommitted temporary files).

Syntax:

 Copyright © Huawei Technologies Co., Ltd.

```
iceberg_remove_orphanfiles('schema.table')
iceberg_remove_orphanfiles('schema.table', 'older_than', 'location')
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table where operations are to be performed
older_than	timestamp	Three days ago	No	Orphan files older than this timestamp will be removed.
location	String	Table root directory	No	Only orphan files under this location will be removed.

Return values:

Parameter	Description
orphan_file_location	Path of the file identified as an orphaned file.

iceberg_add_files

Description: Add all files under the target folder to the Iceberg table.

This syntax recursively scans all files under the target folder and adds them to the Iceberg table.

All files must satisfy:

- Must be in Parquet format.
- Column names, types, and counts (including partition columns) in the Parquet file must match those in the target table.
- All data within a single Parquet file must belong to the same partition.

Syntax:

```
iceberg_add_files(table,foldername,iscopy)
```

Input parameters:

- **table**: table name, which is of the regclass type.
- **foldername**: folder path list, which is of the text type. Multiple paths are supported and are separated by commas (,).
- **iscopy**: whether to copy, which is of the Boolean type.

 Copyright © Huawei Technologies Co., Ltd.

Return value: whether the execution is successful, which is of the Boolean type.

iceberg_show_snapshots

Definition: Queries snapshot information for a specified Iceberg table.

Syntax:

```
iceberg_show_snapshots(table)
```

Input parameters: **table**: table name, which is of the regclass type.

Return values:

- **sequence_number**: snapshot sequence number, which is of the int64 type.
- **snapshot_id**: snapshot ID, which is of the int64 type.
- **create_time**: snapshot creation time, which is of the timestamptz type.
- **parent_id**: snapshot parent node, which is of the int64 type. If the value is **0**, no parent node exists.
- **operation**: snapshot operation type, which is of the text type.
- **manifestlist**: path to the manifest list associated with the snapshot, which is of the text type.
- **summary**: snapshot statistics, which is of the text type.

iceberg_manifests

Description: Queries all manifest files for a specified Iceberg table.

Syntax:

```
iceberg_manifests(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
content	An **int32** value indicating the file type in the manifest: **0** for data files (DATA), **1** for delete files (DELETES).
path	A **text** value indicating the full path of the manifest file in the storage system.
length	An **int64** value indicating the size of the manifest file in bytes.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
partition_spec_id	An **int32** value indicating the partition specification ID used by this manifest.
added_snapshot_id	An **int64** value indicating the snapshot ID that created this manifest.
added_data_files_count	An **int32** value indicating the number of new data files added in this manifest.
existing_data_files_count	An **int32** value indicating the number of existing data files in this manifest.
deleted_data_files_count	An **int32** value indicating the number of data files marked as deleted in this manifest.
added_delete_files_count	An **int32** value indicating the number of new delete files added in this manifest.
existing_delete_files_count	An **int32** value indicating the number of existing delete files in this manifest.
deleted_delete_files_count	An **int32** value indicating the number of delete files marked as deleted in this manifest.
partition_summaries	A **text** value containing a JSON array of partition summaries. Each element corresponds to a partition field and includes details such as **contains_null**, **contains_nan** (for floating-point types only), **lower_bound**, and **upper_bound**. These represent the range of values for the partition field within this manifest.

iceberg_manifest_entries

Description: Queries all manifest entry information for a specified Iceberg table. Each entry reflects the status of a data file in a specific snapshot.

Syntax:

```
iceberg_manifest_entries(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
status	An **int32** value indicating the file status: **0** for existing (EXISTING), **1** for added (ADDED), or **2** for deleted (DELETED).
snapshot_id	An **int64** value indicating the snapshot ID associated with this entry.
sequence_number	An **int64** value indicating the sequence number of the file, used to determine write order.
file_sequence_number	An **int64** value indicating the order in which the file was added within a single snapshot.
data_file	A **text** value containing complete metadata about the data file (e.g., path, format, partitions, size, record count, and column-level statistics).
readable_metrics	A **text** value presenting readable extended column-level statistics extracted from **data_file** (e.g., minimum, maximum, null counts) for easier inspection and debugging.

iceberg_partitions

Description: Queries partition statistics for a specified Iceberg table, aggregating metrics for all data files in each partition.

Syntax:

```
iceberg_partitions(table, small_file_threshold)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.
small_file_threshold	int32	128 (MB)	No	Threshold to classify files as "small."

Return values:

Parameter	Description
partition	A **text** value indicating the specific value of the partition column (matching the table's partition schema).
spec_id	An **int32** value indicating the ID of the partitioning specification used for this partition.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
record_count	An **int64** value indicating the total number of records across all active data files within the partition.
file_count	An **int32** value indicating the total number of active data files in the partition.
total_data_file_size_in_bytes	An **int64** value indicating the combined size (in bytes) of all active data files in the partition.
position_delete_record_count	An **int64** value indicating the total number of records marked as deleted by positional delete files in the partition.
position_delete_file_count	An **int32** value indicating the number of positional delete files in the partition.
equality_delete_record_count	An **int64** value indicating the total number of records marked as deleted by equality delete files in the partition.
equality_delete_file_count	An **int32** value indicating the number of equality delete files in the partition.
last_updated_at	A **timestamptz** value indicating the most recent update time for the partition (corresponding to the snapshot creation time).
updated_snapshot_id	An **int64** value indicating the snapshot ID associated with the latest update to the partition.
small_file_count	An **int64** value indicating the number of small files in the partition.

iceberg_partition_specs

Description: Queries the set of partition key definitions for all historical versions of a specified Iceberg table.

Syntax:

```
iceberg_partition_specs(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

 Copyright © Huawei Technologies Co., Ltd.

Return values:

Parameter	Description
spec_id	An **int32** value indicating the version ID of the partition key definition.
field_id	An **int32** value indicating the ID of the partition key.
name	A **text** value indicating the name of the partition key.
transform	A **text** description of the expression used for the partition key.
source_id	An **int32** value referencing the column ID from which the partition key is derived. This corresponds to the **field_id** in **iceberg_schemas**.

iceberg_schemas

Description: Queries the set of table definitions for all historical versions of a specified Iceberg table.

Syntax:

```
iceberg_schemas(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
schema_id	An **int32** value indicating the schema version ID.
field_id	An **int32** value indicating the column ID.
name	A **text** value indicating the column name.
type	A **text** value indicating the column type.

iceberg_history

Description: Queries historical version information for a specified Iceberg table.

Syntax:

 Copyright © Huawei Technologies Co., Ltd.

```
iceberg_history(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
made_current_at	A **timestamptz** value indicating when the snapshot was created.
snapshot_id	An **int64** value indicating the snapshot ID.
parent_id	An **int64** value indicating the parent snapshot ID.
is_current_ancestor	A Boolean value indicating whether this snapshot is an ancestor of the current latest snapshot.

iceberg_metadata_log_entries

Definition: Queries metadata file information for a specified Iceberg table.

Syntax:

```
iceberg_metadata_log_entries(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
timestamp	A **timestamptz** value indicating when the **metadata.json** file was generated.
file	A **string** value indicating the path to the **metadata.json** file.
latest_snapshot_id	An **int64** value indicating the latest snapshot ID at the time of the **metadata.json** creation.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
latest_schema_id	An **int64** value indicating the most recent schema (table structure) ID at the time of the **metadata.json** creation.
latest_sequence_number	An **int64** value indicating the latest sequence number for the table when the **metadata.json** file was created.

iceberg_files

Description: Queries information about all data files and delete files for a specified Iceberg table.

Syntax:

```
iceberg_files(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
content	An **int32** value indicating the file type.
file_path	A **string** value indicating the file path.
file_format	A **string** value indicating the file storage format.
spec_id	An **int64** value indicating the specification ID (**spec_id**) linked to the file.
record_count	An **int64** value indicating the number of rows within the file.
file_size_in_bytes	An **int64** value indicating the file size.
column_sizes	A **string** value indicating the storage sizes of each column.
value_counts	A **string** value indicating the number of non-null values for each column.
null_value_counts	A **string** value indicating the number of null values for each column.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
nan_value_counts	A **string** value indicating the number of not-a-number (NaN) values for each column.
lower_bounds	A **string** value defining the lower bounds of values for each column.
upper_bounds	A **string** value defining the upper bounds of values for each column.

iceberg_position_delete_files

Description: Queries position delete file information for a specified Iceberg table.

Syntax:

```
iceberg_position_delete_files(table)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
file_path	A **string** value indicating the path to the data files associated with the delete files.
pos	An **int64** value indicating the row number referenced in the delete file.
partition	A **string** value indicating the partition name.
spec_id	An **int64** value indicating the specification ID of the partition.
delete_file_path	A **string** value indicating the path to the delete files.

iceberg_refs

Description: Queries all branches/tags for a specified Iceberg table.

Syntax:

```
iceberg_refs(table)
```

 Copyright © Huawei Technologies Co., Ltd.

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Name of the table to query.

Return values:

Parameter	Description
name	A **string** value indicating the name.
type	A **string** value indicating whether it is a branch or tag.
snapshot_id	An **int64** value corresponding to the snapshot ID.
max_reference_age_in_ms	An **int64** value indicating the retention time in milliseconds.
min_snapshots_to_keep	An **int64** value defining the minimum number of snapshots to retain.
max_snapshot_age_in_ms	An **int64** value setting the maximum duration for retaining snapshots, in milliseconds.

iceberg_register_table

Description: Registers an existing **metadata.json** file with the current catalog.

Syntax:

```
select * from iceberg_register_table(table, file_path)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Registered table name
file_path	String	None	Yes	Path of the **metadata.json** file

Return values:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
snapshot_id	Snapshot ID of the newly registered table. The value type is int64.
record_count	Total number of rows in the newly registered table. The value type is int64.
total_data_files_count	Number of data files in the newly registered table. The value type is int64.

iceberg_rollback_to_snapshot

Description: Rolls back the main branch of a table to a specified snapshot.

Syntax:

```
select * from iceberg_rollback_to_snapshot(table, snapshot_id)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table name
snapshot_id	int64	None	Yes	ID of the target snapshot

Return values:

Parameter	Description
previous_snapshot_id	Snapshot ID of the table before the rollback. The value type is int64.
current_snapshot_id	Snapshot ID of the table after the rollback. The value type is int64.

iceberg_rollback_to_timestamp

Description: Rolls back the main branch of a table to a specified timestamp.

Syntax:

```
select * from iceberg_rollback_to_timestamp(table, timestamp)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table name
timestamp	timestamptz	None	Yes	Target timestamp

Return values:

Parameter	Description
previous_snapshot_id	Snapshot ID of the table before the rollback. The value type is int64.
current_snapshot_id	Snapshot ID of the table after the rollback. The value type is int64.

iceberg_set_current_snapshot

Description: Sets the current snapshot of the main branch.

Syntax:

```
select * from iceberg_set_current_snapshot(table, snapshot_id)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table name
snapshot_id	int64	None	Yes	ID of the target snapshot

Return values:

Parameter	Description
previous_snapshot_id	Snapshot ID of the table before the snapshot is set. The value type is int64.
current_snapshot_id	Snapshot ID of the table after the snapshot is set. The value type is int64.

 Copyright © Huawei Technologies Co., Ltd.

iceberg_cherrypick_snapshot

Description: Extracts a snapshot to the main branch and creates another snapshot. The extracted snapshot cannot be an ancestor node of the main branch.

Syntax:

```
select * from iceberg_cherrypick_snapshot(table, snapshot_id)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	String	None	Yes	Table name
snapshot_id	int64	None	Yes	ID of the snapshot to be extracted

Return values:

Parameter	Description
previous_snapshot_id	Snapshot ID of the table before the snapshot extraction. The value type is int64.
current_snapshot_id	Snapshot ID of the table after the snapshot extraction. The value type is int64.

iceberg_fast_forward

Description: Fast-forwards the current snapshot of a branch to the latest snapshot of another branch.

Syntax:

```
select * from iceberg_fast_forward(table, from_branch, to_branch)
```

Input parameters:

Parameter	Type	Default Value	Mandatory	Description
table	string	None	Yes	Table name
from_branch	String	None	Yes	Current branch
to_branch	String	None	Yes	Target branch

Return values:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
branch_updated	Name of the updated branch. The value is a string.
previous_ref	Snapshot ID of the table before the snapshot forwarding. The value type is int64.
updated_ref	Snapshot ID of the table after the snapshot forwarding. The value type is int64.

11.21 Large Model Inference Functions

ai_generate(model_name, prompt[, model_hyper_parameter])

Description: Calls the remote model created in **3.8 CREATE MODEL** for inference.

Syntax:

```
ai_generate(model_name, prompt[, model_hyper_parameter])
```

Input parameters:

- **model_name**: name of the created model
- **prompt**: prompt sent to the remote model. The value can be **CONCAT(prompt_prefix, column_name)**, a string containing only text, or **column_name**.
- **model_hyper_parameter**: hyperparameter for remote model inference, for example, **'{"temperature": 0.3}'**

Return value: remote model inference result

Precautions: none

 Copyright © Huawei Technologies Co., Ltd.

12 User and Permissions Management

12.1 Aura Service Permissions

The Aura compute engine of AI DataLake is a Huawei Cloud service. Its user system complies with Huawei Cloud Identity and Access Management (IAM) and IAM Identity Center, and consists of users, user groups, and agencies. AI DataLake interconnects with LakeFormation (the unified metadata management service). The permission system of AI DataLake is integrated with LakeFormation's permissions management. LakeFormation service roles are recommended as permission sets for managing the permissions of AI DataLake data resources such as catalogs, databases, tables, models, and functions. These roles can be assigned to users, user groups, and agencies to grant them fine-grained data access permissions.

Table 12-1 Permission authentication services

Service	Function
IAM	User, user group, and agency management (creation, deletion, and authorization)
LakeFormation	Role management, authorization, permission query, and authentication

Data Control Language (DCL) is the core security capability of databases. It enables kernel-level permission control for databases, creation and management of users, user groups, and roles, and fine-grained object authorization and permission query. You can use SQL statements to perform all permission operations in the Aura database rather than on the LakeFormation console, achieving closed-loop permissions management.

 Copyright © Huawei Technologies Co., Ltd.

12.2 User Management

The Aura compute engine is a database query system that provides SQL statements for managing users, user groups, agencies, and roles.

User Management

- Query users.
  ```
  SELECT * FROM pg_show_users();
  ```
- Query user groups.
  ```
  SELECT * FROM pg_show_user_groups();
  ```
- Query agencies.
  ```
  SELECT * FROM pg_show_agents();
  ```
- Query roles.
  ```
  SELECT * FROM pg_show_roles();
  ```
- Query mappings between user objects.
  - Querying mappings between users.
    ```
    SELECT * FROM pg_show_mappings('USER', 'username');
    ```
  - Querying mappings between user groups.
    ```
    SELECT * FROM pg_show_mappings('GROUP', 'groupname');
    ```
  - Querying mappings between roles.
    ```
    SELECT * FROM pg_show_mappings('ROLE', 'rolename');
    ```
  - Querying mappings between agencies.
    ```
    SELECT * FROM pg_show_mappings('AGENT', 'agentname');
    ```

12.3 Permissions Management

12.3.1 Permission Introduction

Permissions are used to control whether a user is allowed to access database objects (such as tables, views, functions, and models) and perform operations (such as adding, deleting, modifying, querying, and creating database objects).

Permissions management in Aura falls into two categories:

- Data object permissions

  Grant permissions for a database object (table, view, column, database, function, or schema) to a role or user. The **GRANT** command grants specified database object permissions to one or more roles. These permissions are appended to those already granted, if any.
- User permissions

  Grant a role's permissions to one or more users. In this case, every role or user can be regarded as a set of one or more database permissions.

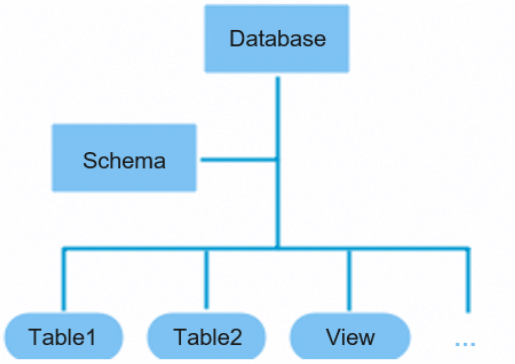
How It Works

Aura implements a hierarchical permission management on databases, schemas, and data objects.

 Copyright © Huawei Technologies Co., Ltd.

- Databases cannot communicate or share resources with each other. Their connections and permissions can be isolated. Users and roles are shared within the service, but data is not shared among databases. That is, a user can connect to any database, but can access only the database declared in the connection request.
- Schemas share more resources than databases do. User permissions on schemas and subordinate objects can be flexibly configured using the **GRANT** and **REVOKE** syntax. Each database has one or more schemas. Each schema contains various types of objects, such as tables and functions. To access the objects in a specified schema, a user must have required permissions.
- By default, only the object owner or system administrator can query, modify, and delete objects. To access a specific database object, such as Table1, other users must be granted the SELECT permission on Table1.

Figure 12-1 Permissions management



12.3.2 Role Management

Aura uses a typical role-based access control (RBAC) model to manage users, roles, and permissions.

A role is a set of permissions.

- Roles are assigned different permissions based on their responsibilities in the database system. A role is a set of database permissions and represents the behavior constraints of a user or user group.
- After a role is assigned to a user through **GRANT**, the user will have all the permissions of the role. Roles are recommended for efficient permissions assignment. For example, you can create different roles for designers, developers, and maintainers, assign the roles to users through **GRANT**, and grant specific data permissions to the users. When permissions are granted to or revoked from roles, the changes apply to all members of the roles.

Querying Roles

```
SELECT * FROM pg_show_roles();
```

Managing Roles

- Create a role.
  ```
  CREATE ROLE role_name [ COMMENT 'comment_text' ];
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Delete a role.
  ```
  DROP ROLE [ IF EXISTS ] role_name;
  ```
- Modify a role.
  ```
  ALTER ROLE role_name COMMENT 'modify COMMENT';
  ```
- Associate a role with users.
  - Change the role associated with a user.
    ```
    ALTER USER user1 WITH ROLE role1;
    ```
  - Change the users associated with a role.
    ```
    ALTER ROLE role1 ADD USER user1;
    ```
- Associate a role with user groups.
  - Change the role associated with a user group.
    ```
    ALTER USER GROUP group1 WITH ROLE role1;
    ```
  - Change the user groups associated with a role.
    ```
    ALTER ROLE role1 ADD USER GROUP group1;
    ```
- Associate a role with an agency.
  ```
  ALTER ROLE role1 ADD AGENT agent1;
  ```
- Remove association.
  - Disassociate a user with a role.
    ```
    ALTER ROLE role1 DROP USER user1;
    ```
  - Disassociate a user group with a role.
    ```
    ALTER ROLE role1 DROP USER GROUP group1;
    ```
  - Disassociate an agency with a role.
    ```
    ALTER ROLE role1 DROP AGENT agent1;
    ```

12.3.3 Object Authorization/Deauthorization Management

Permission Types Supported by Objects

- Permissions supported by objects

Object Type	READ	WRITE	DESCRIBE	EXECUTE	INSERT	SELECT	UPDATE	DELETE	ALTER	DROP
DATABASE	✓	✓	✓	-	-	-	-	-	✓	✓
SCHEMA	✓	✓	✓	-	-	-	-	-	✓	✓
TABLE	-	-	✓	-	✓	✓	✓	✓	✓	✓
FUNCTION	-	-	✓	✓	-	-	-	-	✓	✓
PATH	✓	✓	-	-	-	-	-	-	-	-

 Copyright © Huawei Technologies Co., Ltd.

- Permissions that can be granted to objects

Aura Object	Aura Permissions	LakeFormation Object	LakeFormation Permissions	Description
DATABASE	ALL	CATALOG	ALL	All permissions on a database object
	READ		LIST_DATABASE	Queries schemas in a database.
	WRITE		CREATE_DATABASE	Creates a schema in a database.
	DESCRIBE		DESCRIBE	Queries database metadata.
	ALTER		ALTER	Modifies database permissions. It is not supported currently.
	DROP		DROP	Deletes database permissions. It is not supported currently.
SCHEMA	ALL	DATABASE	ALL	All permissions on a schema object
	READ		LIST_TABLE LIST_FUNC	Queries tables in a schema. Queries functions in a schema.
	WRITE		CREATE_TABLE CREATE_FUNC	Creates a table in a schema. Creates a function in a schema.
	DESCRIBE		DESCRIBE	Queries schema metadata.
	ALTER		ALTER	Modifies schema permissions.
	DROP		DROP	Deletes schema permissions.
TABLE	ALL	TABLE	ALL	All permissions on a table object.
	DESCRIBE		DESCRIBE	Queries table metadata.
	INSERT		INSERT	Imports table data.

 Copyright © Huawei Technologies Co., Ltd.

Aura Object	Aura Permissions	LakeFormation Object	LakeFormation Permissions	Description
	SELECT		SELECT	Queries table data.
	UPDATE		UPDATE	Updates table data.
	DELETE		DELETE	Deletes table data.
	ALTER		ALTER	Modifies a table.
	DROP		DROP	Deletes a table.
FUNCTION	ALL	FUNCTION	ALL	All permissions on a function object
	DESCRIBE		DESCRIBE	Queries function metadata.
	EXECUTE		EXEC	Executes a function.
	ALTER		ALTER	Modifies a function.
	DROP		DROP	Deletes a function.
PATH	READ	URI	READ	Reads an OBS path.
	WRITE		WRITE	Writes an OBS path.

Notes and Constraints

Constraints on object authorization

- **Agency authorization restrictions**: Permissions cannot be directly granted to agencies.
- **Batch authorization restrictions**: Excessive authorization may cause LakeFormation to perform associated authorization on OBS paths.
- **Column permission restrictions**: Table column permissions cannot be managed. (It will be supported in later versions.)
- **Owner restrictions**: Only table owners can be changed.

Object Authorization Management

- Object authorization syntax

```
GRANT privileges [ WITH GRANT ] ON privilege_target TO grantee_list;
```

WITH GRANT indicates cascading authorization, which means that the authorized user can grant the same permissions to other users.

Specifically:

– **privileges** can be:

 Copyright © Huawei Technologies Co., Ltd.

```
| [ READ, WRITE, DESCRIBE, EXECUTE, ALTER, DROP, INSERT, SELECT, UPDATE, DELETE ] [, ...]
| ALL
| ALL PRIVILEGE
| ALL PRIVILEGES
```

  - **privilege_target** can be:

```
| DATABASE name_list
| SCHEMA name_list
| TABLE qualified_name_list
| FUNCTION function_with_argtypes_list
| PATH obs_path_list
| ALL TABLES IN SCHEMA name_list
| ALL FUNCTIONS IN SCHEMA name_list
```

    Specifically:

    - name_list is as follows:

```
| name [, ...]
```

    - qualified_name_list is as follows:

```
| qualified_name [, ...]
```

    - function_with_argtypes_list is as follows:

```
| function_with_argtypes [, ...]
```

    - obs_path_list is as follows:

```
| SCONST [, ...]
```

  - **grantee_list** can be:

```
| USER RoleId [, ...]
| USER GROUP RoleId [, ...]
| ROLE RoleId [, ...]
```

    Supported types of authorized objects:

    - USER *username*
    - USER GROUP *groupname*
    - ROLE *rolename*

- Database object authorization
  - Grant database permissions to a user.

```
GRANT READ [ WITH GRANT ] ON DATABASE database1 TO USER user1;
GRANT WRITE [ WITH GRANT ] ON DATABASE database1 TO USER user1;
GRANT DESCRIBE [ WITH GRANT ] ON DATABASE database1 TO USER user1;
GRANT ALTER [ WITH GRANT ] ON DATABASE database1 TO USER user1;
GRANT DROP  [ WITH GRANT ] ON DATABASE database1 TO USER user1;
GRANT ALL   [ WITH GRANT ] ON DATABASE database1 TO USER user1;
```

  - Grant database permissions to a user group.

```
GRANT READ  [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
GRANT WRITE [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
GRANT DESCRIBE [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
GRANT ALTER [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
GRANT DROP  [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
GRANT ALL   [ WITH GRANT ] ON DATABASE database1 TO USER GROUP group1;
```

  - Grant database permissions to a role.

```
GRANT READ  [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
GRANT WRITE [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
GRANT DESCRIBE [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
GRANT ALTER [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
GRANT DROP  [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
GRANT ALL   [ WITH GRANT ] ON DATABASE database1 TO ROLE role1;
```

- Schema object authorization
  - Grant schema permissions to a user.

 Copyright © Huawei Technologies Co., Ltd.

```
GRANT READ  [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
GRANT WRITE [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
GRANT DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
GRANT ALTER [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
GRANT DROP  [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
GRANT ALL   [ WITH GRANT ] ON SCHEMA schema1 TO USER user1;
```

  - Grant schema permissions to a user group.

```
GRANT READ  [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
GRANT WRITE [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
GRANT DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
GRANT ALTER [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
GRANT DROP  [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
GRANT ALL   [ WITH GRANT ] ON SCHEMA schema1 TO USER GROUP group1;
```

  - Grant schema permissions to a role.

```
GRANT READ  [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
GRANT WRITE [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
GRANT DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
GRANT ALTER [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
GRANT DROP  [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
GRANT ALL   [ WITH GRANT ] ON SCHEMA schema1 TO ROLE role1;
```

- Table object authorization
  - Grant table permissions to a user.

```
GRANT DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT INSERT [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT SELECT [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT UPDATE [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT DELETE [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT ALTER [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT DROP  [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
GRANT ALL   [ WITH GRANT ] ON TABLE s1.table1 TO USER user1;
```

    The following special authorization operations are also supported:

```
GRANT DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL  [ WITH GRANT ]  ON ALL
TABLES IN SCHEMA schema1 TO USER user1;
```

  - Grant table permissions to a user group.

```
GRANT DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT INSERT [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT SELECT [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT UPDATE [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT DELETE [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT ALTER [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT DROP  [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
GRANT ALL   [ WITH GRANT ] ON TABLE s1.table1 TO USER GROUP group1;
```

    The following special authorization operations are also supported:

```
GRANT DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL  [ WITH GRANT ] ON ALL
TABLES IN SCHEMA schema1 TO USER GROUP group1;
```

  - Grant table permissions to a role.

```
GRANT DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT INSERT [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT SELECT [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT UPDATE [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT DELETE [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT ALTER [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT DROP  [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
GRANT ALL   [ WITH GRANT ] ON TABLE s1.table1 TO ROLE role1;
```

    The following special authorization operations are also supported:

```
GRANT DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL  [ WITH GRANT ] ON ALL
TABLES IN SCHEMA schema1 TO ROLE role1;
```

- Function object authorization
  - Grant function permissions to a user.

```
GRANT DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER user1;
GRANT EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER user1;
```

 Copyright © Huawei Technologies Co., Ltd.

```
GRANT ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER user1;
GRANT DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER user1;
GRANT ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER user1;
```

The following special authorization operations are also supported:

```
GRANT DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
SCHEMA schema1 TO USER user1;
```

- Grant function permissions to a user group.

```
GRANT DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER GROUP group1;
GRANT EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER GROUP group1;
GRANT ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER GROUP group1;
GRANT DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER GROUP group1;
GRANT ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO USER GROUP group1;
```

The following special authorization operations are also supported:

```
GRANT DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
SCHEMA schema1 TO USER GROUP group1;
```

- Grant function permissions to a role.

```
GRANT DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO ROLE role1;
GRANT EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO ROLE role1;
GRANT ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO ROLE role1;
GRANT DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO ROLE role1;
GRANT ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) TO ROLE role1;
```

The following special authorization operations are also supported:

```
GRANT DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
SCHEMA schema1 TO ROLE role1;
```

- Path object authorization
  - Grant path permissions to a user.

```
GRANT READ [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER user1;
GRANT WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER user1;
GRANT ALL [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER user1;
```

  - Grant path permissions to a user group.

```
GRANT READ [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER GROUP group1;
GRANT WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER GROUP group1;
GRANT ALL [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO USER GROUP group1;
```

  - Grant path permissions to a role.

```
GRANT READ [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO ROLE role1;
GRANT WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO ROLE role1;
GRANT ALL [ WITH GRANT ] ON PATH 'obs://bucket/path1' TO ROLE role1;
```

Revoking Authorization

- Syntax for revoking authorization

```
REVOKE privileges [ WITH GRANT ] ON privilege_target FROM grantee_list;
```

WITH GRANT indicates cascading authorization, which means that the authorized user can grant the same permissions to other users.

Specifically:

- **privileges** can be:

```
| [ READ, WRITE, DESCRIBE, EXECUTE, ALTER, DROP, INSERT, SELECT, UPDATE, DELETE ] [, ...]
| ALL
| ALL PRIVILEGE
| ALL PRIVILEGES
```

- **privilege_target** can be:

```
| DATABASE name_list
| SCHEMA name_list
| TABLE qualified_name_list
| FUNCTION function_with_argtypes_list
| PATH obs_path_list
| ALL TABLES IN SCHEMA name_list
| ALL FUNCTIONS IN SCHEMA name_list
```

 Copyright © Huawei Technologies Co., Ltd.

Specifically:

- name_list is as follows:
  ```
  | name [, ...]
  ```
- qualified_name_list is as follows:
  ```
  | qualified_name [, ...]
  ```
- function_with_argtypes_list is as follows:
  ```
  | function_with_argtypes [, ...]
  ```
- obs_path_list is as follows:
  ```
  | SCONST [, ...]
  ```

– **grantee_list** can be:

```
| USER RoleId [, ...]
| USER GROUP RoleId [, ...]
| ROLE RoleId [, ...]
```

- Database object authorization revocation
  – Revoke database permissions from a user.
  ```
  REVOKE READ [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  REVOKE WRITE [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  REVOKE DESCRIBE [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  REVOKE ALTER [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  REVOKE DROP  [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  REVOKE ALL   [ WITH GRANT ] ON DATABASE database1 FROM USER user1;
  ```
  – Revoke database permissions from a user group.
  ```
  REVOKE READ  [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  REVOKE WRITE [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  REVOKE DESCRIBE [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  REVOKE ALTER [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  REVOKE DROP  [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  REVOKE ALL   [ WITH GRANT ] ON DATABASE database1 FROM USER GROUP group1;
  ```
  – Revoke database permissions from a role.
  ```
  REVOKE READ  [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  REVOKE WRITE [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  REVOKE DESCRIBE [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  REVOKE ALTER [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  REVOKE DROP  [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  REVOKE ALL   [ WITH GRANT ] ON DATABASE database1 FROM ROLE role1;
  ```
- Schema object authorization revocation
  – Revoke schema permissions from a user.
  ```
  REVOKE READ  [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  REVOKE WRITE [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  REVOKE DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  REVOKE ALTER [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  REVOKE DROP  [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  REVOKE ALL   [ WITH GRANT ] ON SCHEMA schema1 FROM USER user1;
  ```
  – Revoke schema permissions from a user group.
  ```
  REVOKE READ  [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  REVOKE WRITE [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  REVOKE DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  REVOKE ALTER [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  REVOKE DROP  [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  REVOKE ALL   [ WITH GRANT ] ON SCHEMA schema1 FROM USER GROUP group1;
  ```
  – Revoke schema permissions from a role.
  ```
  REVOKE READ  [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  REVOKE WRITE [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  REVOKE DESCRIBE [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  REVOKE ALTER [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  REVOKE DROP  [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  REVOKE ALL   [ WITH GRANT ] ON SCHEMA schema1 FROM ROLE role1;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Table object authorization revocation
  - Revoke table permissions from a user.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE INSERT [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE SELECT [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE UPDATE [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE DELETE [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE ALTER [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE DROP  [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    REVOKE ALL   [ WITH GRANT ] ON TABLE s1.table1 FROM USER user1;
    ```

    The following special operations are also supported:

    ```
    REVOKE DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL  [ WITH GRANT ]  ON ALL
    TABLES IN SCHEMA schema1 FROM USER user1;
    ```

  - Revoke table permissions from a user group.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE INSERT [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE SELECT [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE UPDATE [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE DELETE [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE ALTER [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE DROP  [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    REVOKE ALL   [ WITH GRANT ] ON TABLE s1.table1 FROM USER GROUP group1;
    ```

    The following special operations are also supported:

    ```
    REVOKE DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL
    TABLES IN SCHEMA schema1 FROM USER GROUP group1;
    ```

  - Revoke table permissions from a role.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE INSERT [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE SELECT [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE UPDATE [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE DELETE [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE ALTER [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE DROP  [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    REVOKE ALL   [ WITH GRANT ] ON TABLE s1.table1 FROM ROLE role1;
    ```

    The following special operations are also supported:

    ```
    REVOKE DESCRIBE,INSERT,SELECT,UPDATE,DELETE,ALTER,DROP,ALL  [ WITH GRANT ] ON ALL
    TABLES IN SCHEMA schema1 FROM ROLE role1;
    ```

- Function object authorization revocation
  - Revoke function permissions from a user.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER user1;
    REVOKE EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER user1;
    REVOKE ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER user1;
    REVOKE DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER user1;
    REVOKE ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER user1;
    ```

    The following special operations are also supported:

    ```
    REVOKE DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
    SCHEMA schema1 FROM USER user1;
    ```

  - Revoke function permissions from a user group.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER GROUP group1;
    REVOKE EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER GROUP group1;
    REVOKE ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER GROUP group1;
    REVOKE DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER GROUP group1;
    REVOKE ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM USER GROUP group1;
    ```

    The following special operations are also supported:

    ```
    REVOKE DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
    SCHEMA schema1 FROM USER GROUP group1;
    ```

  - Revoke function permissions from a role.

    ```
    REVOKE DESCRIBE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM ROLE role1;
    REVOKE EXECUTE [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM ROLE role1;
    ```

 Copyright © Huawei Technologies Co., Ltd.

```
REVOKE ALTER [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM ROLE role1;
REVOKE DROP  [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM ROLE role1;
REVOKE ALL   [ WITH GRANT ] ON FUNCTION func1(s1 int[]) FROM ROLE role1;
```

The following special operations are also supported:

```
REVOKE DESCRIBE,EXECUTE,ALTER,DROP,ALL   [ WITH GRANT ] ON ALL FUNCTIONS IN
SCHEMA schema1 FROM ROLE role1;
```

- Path object authorization revocation
  - Revoke path permissions from a user.
    ```
    REVOKE READ  [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER user1;
    REVOKE WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER user1;
    REVOKE ALL   [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER user1;
    ```
  - Revoke path permissions from a user group.
    ```
    REVOKE READ  [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER GROUP group1;
    REVOKE WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER GROUP group1;
    REVOKE ALL   [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM USER GROUP group1;
    ```
  - Revoke path permissions from a role.
    ```
    REVOKE READ  [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM ROLE role1;
    REVOKE WRITE [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM ROLE role1;
    REVOKE ALL   [ WITH GRANT ] ON PATH 'obs://bucket/path1' FROM ROLE role1;
    ```

Object Owner Management

- Change the table owner.
  ```
  ALTER TABLE table_name OWNER TO USER username;
  ALTER TABLE table_name OWNER TO USER GROUP groupname;
  ```
- Query the permissions granted to an object.
  - Query the permissions granted to a user.
    ```
    SELECT * FROM pg_show_grants('USER', 'username');
    ```
  - Query the permissions granted to a user group.
    ```
    SELECT * FROM pg_show_grants('GROUP', 'groupname');
    ```
  - Query the permissions granted to a role.
    ```
    SELECT * FROM pg_show_grants('ROLE', 'rolename');
    ```
  - Query the permissions granted to a database.
    ```
    SELECT * FROM pg_show_grants('DATABASE', 'databasename');
    ```
  - Query the permissions granted to a table.
    ```
    SELECT * FROM pg_show_grants('TABLE', 'schemaname.tablename');
    ```
  - Query the permissions granted to a function.
    ```
    SELECT * FROM pg_show_grants('FUNCTION', 'functionname');
    ```

Authorization/Authorization Revocation Examples

- Table authorization
  ```
  GRANT SELECT, INSERT ON TABLE schema1.table1 TO USER user1;
  GRANT ALL ON TABLE schema1.table1 TO ROLE role1;
  ```
- Schema authorization
  ```
  GRANT READ ON SCHEMA schema1 TO USER GROUP group1;
  ```
- Database authorization
  ```
  GRANT DESCRIBE ON DATABASE database1 TO USER user1;
  ```
- Function authorization
  ```
  GRANT EXECUTE ON FUNCTION func1(int[]) TO USER user1;
  ```
- OBS path authorization
  ```
  GRANT READ ON PATH 'obs://bucket/path1' TO ROLE role1;
  ```
- Batch authorization
  ```
  GRANT ALL ON ALL TABLES IN SCHEMA schema1 TO USER user1;
  GRANT ALL ON ALL FUNCTIONS IN SCHEMA schema1 TO ROLE role1;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Cascading authorization (Authorized users can grant permissions to other users.)
  ```
  GRANT SELECT ON TABLE t1 TO USER user1 WITH GRANT;
  ```
- Authorization revocation
  ```
  REVOKE SELECT ON TABLE schema1.table1 FROM USER user1;
  REVOKE ALL ON ALL TABLES IN SCHEMA schema1 FROM ROLE role1;
  ```

 Copyright © Huawei Technologies Co., Ltd.

13 Performance Tuning

13.1 Overview

Database performance tuning is the process of optimizing database system configuration and SQL queries to improve database performance and efficiency. The purpose includes eliminating performance bottlenecks, reducing response times, increasing throughput and resource utilization, cutting costs, and improving system stability.

This section provides comprehensive guidance for database administrators on performance diagnosis, system tuning, and SQL tuning, as well as practical examples of SQL tuning.

Precautions

Database tuning is a complex process that requires familiarity with the internal workings and related technologies of the database system. It requires a comprehensive consideration of various factors such as hardware, software, queries, configuration, and data structures to achieve optimal performance and efficiency. Consequently, tuning professionals must possess extensive and profound knowledge of system software architecture, hardware and software configurations, database configuration parameters, query processing, and database applications.

Performance Tuning Process

Figure 13-1 illustrates the performance tuning process.

 Copyright © Huawei Technologies Co., Ltd.

Figure 13-1 Aura performance tuning

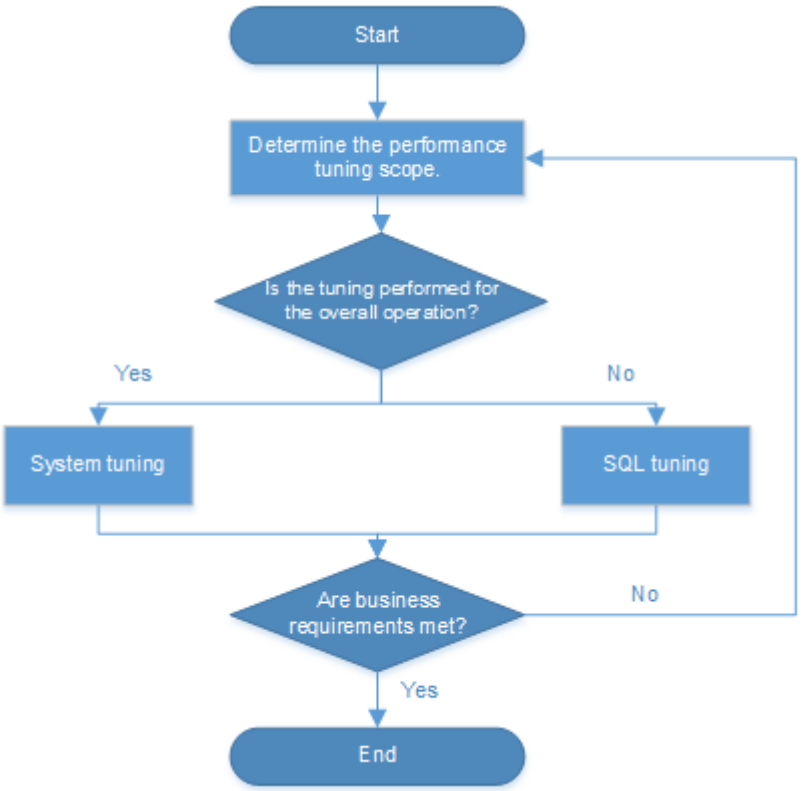


Table 13-1 gives a brief introduction to each phase of the performance tuning process.

Table 13-1 Aura performance tuning

Phase	Description
13.2 System Optimization	Conduct system-level tuning to more effectively utilize the node's CPU, memory, I/O, and networking resources, enhancing query throughput.

 Copyright © Huawei Technologies Co., Ltd.

Phase	Description
13.3 SQL Tuning	Analyze the SQL statements used and determine whether any optimization can be performed. Analysis of SQL statements comprises: • Generating table statistics using **ANALYZE**: The **ANALYZE** statement collects statistics about the database table content. Statistical results are stored in the system catalog **PG_STATISTIC**. The execution plan generator uses these statistics to determine which one is the most effective execution plan. • Analyzing the execution plan: The **EXPLAIN** statement displays the execution plan of SQL statements, and the **EXPLAIN PERFORMANCE** statement displays the execution time of each operator in SQL statements. • Identifying the root causes of issues: Identify possible causes by analyzing the execution plan and perform specific optimization by modifying database-level SQL optimization parameters. • Compiling better SQL statements: Compile better SQL statements in the scenarios, such as cache of intermediate and temporary data for complex queries, result set cache, and result set combination.

13.2 System Optimization

13.2.1 Tuning Database Parameters

To ensure high performance of the database, you are advised to configure GUC parameters based on available resources and the actual workloads. This section describes some of the common parameters and the recommended configurations for them. For more details, see **14.1 Viewing and Setting GUC Parameters**.

 Copyright © Huawei Technologies Co., Ltd.

Parameters Related to Database Memory

Table 13-2 Parameters related to database memory

GUC Parameter	Description	Suggestion
fq_runtime	Specifies the number of actors used for queries and their CPU/memory configurations.	• This parameter can configure the following: – **actor_size**: number of vCPUs and total memory per actor – **actor**: number of actors – **dop**: SMP degree of parallelism (DOP) Can be adjusted based on the volume of data being queried.

13.2.2 SMP Parallel Execution

Complex queries may take a long time. In a system with low concurrency support, this can be a problem. SMP is used to implement operator-level parallel execution, which can effectively speed up queries, improving query performance and resource utilization.

The SMP feature improves performance through operator parallelism but may drive more resource usage, including CPU, memory, network, and I/O. In essence, SMP is a method that trades resources for time, meaning it accelerates queries at the cost of additional resources. It improves system performance in appropriate scenarios and when resources are sufficient, but may also deteriorate performance if used inappropriately. Furthermore, compared with serial processing, SMP generates more candidate plans, which is more time-consuming and may hurt performance.

The SMP feature of Aura is controlled by the GUC parameter **query_dop**. Users use this parameter to specify an appropriate degree of query parallelism.

Application Scenarios and Constraints for SMP

Applicable Scenarios

- Operators supporting parallel processing are used.

  The execution plan contains the following operators:

  a. Scan: Parallel scanning for all types of external tables is supported.
  b. Join: HashJoin, NestLoop
  c. Agg: HashAgg, SortAgg, PlainAgg, and WindowAgg, which supports only **partition by**, and does not support **order by**.
  d. Stream: Redistribute, Broadcast
  e. Other: Result, Subqueryscan, Unique, Material, Setop, Append, VecToRow, RowToVec

 Copyright © Huawei Technologies Co., Ltd.

- SMP-unique operators are used.

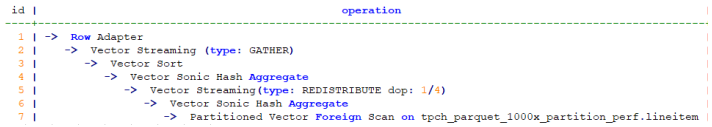
  To execute queries in parallel, Stream operators are added for data exchange for the SMP feature. These new operators can be considered as the subtypes of Stream operators.

  a. Local Gather aggregates data of parallel threads within a DN
  b. Local Redistribute redistributes data based on the distributed key across threads within a DN
  c. Local Broadcast broadcasts data to each thread within a DN.
  d. Local RoundRobin distributes data in polling mode across threads within a DN.
  e. Split Redistribute redistributes data across parallel threads on different DNs.
  f. Split Broadcast broadcasts data to all parallel DN threads.

  Among these operators, Local operators exchange data between parallel threads within a DN, and non-Local operators exchange data across DNs.

- Example

  The TPCH Q1 parallel plan is used as an example.

  In this plan, implement the Foreign Scan and HashAgg operator parallel, and adds the Split Redistribute data exchange operator.

  Operator 5 is the Split Redistribute operator. **dop: 1/4** indicates that the DOP of the sender and receiver threads of Split Redistribute is 1 and 4, respectively. That is, the lower-layer Hash Aggregate operator 6 is executed based on the DOP 4. The upper-layer operators 1 to 4 are executed in serial mode.

  You can view the parallelism situation of each operator in the dop information.

Non-Applicable Scenarios

1. Small queries are performed, where plan generation may account for a significant portion of the total query time.
2. Operators are processed on CNs.
3. Statements that cannot be pushed down are executed.
4. The **subplan** of a query and operators containing a subquery are executed.

Impact of Resource Availability on SMP Performance

The SMP architecture accelerates queries at the cost of additional resources. After the plan parallelism is executed, more resources are consumed, including the CPU, memory, I/O, and network bandwidth. As the DOP grows, the resource consumption also increases. If these resources become a bottleneck, SMP cannot improve performance. On the contrary, it may do exactly the opposite. Adaptive SMP is provided to dynamically select the optimal DOP for each query based on the resource usage and query requirements. The following information describes the situations that the SMP affects these resources:

 Copyright © Huawei Technologies Co., Ltd.

- **CPU resources**

  In a general customer scenario, the system CPU usage rate is not high. Using the SMP parallelism architecture will fully use the CPU resource to improve the system performance. If the number of CPU kernels of the database server is too small and the CPU usage is already high, enabling the SMP parallelism may deteriorate the system performance due to resource compete between multiple threads.

- **Memory resources**

  The query parallel causes memory usage growth, but the memory upper limit used by each operator is still restricted by **work_mem**. Assume that **work_mem** is 4 GB, and the DOP is 2, then the memory upper limit of each concurrent thread is 2 GB. When **work_mem** is small or the system memory is sufficient, running SMP parallelism may push data down to disks. As a result, the query performance deteriorates.

- **Network bandwidth resources**

  To execute queries in parallel, data exchange operators are added. Local stream operators exchange data between threads within a DN. Data is exchanged in memory, so it does not impact network performance. Non-local operators exchange data over the network and increase the network load. If the capacity of a network resource has already become a bottleneck, parallelism may hurt performance.

- **I/O resources**

  A parallel scan increases I/O resource consumption. It can improve performance only when I/O resources are sufficient.

Other Factors Impacting SMP Performance

Besides the resource factor, other factors may also impact SMP performance, such as uneven data distribution across tables and the degree of system parallelism.

- **Impact of data skew on SMP performance**

  Serious data skew deteriorates parallel execution performance. For example, if the data volume of a value in the join column is much more than that of other values, the data volume of a parallel thread will be much more than that of others after Hash-based data redistribution, resulting in the long-tail issue and poor parallelism performance.

- **Impact of system DOP on SMP performance**

  The SMP feature uses more resources, and unused resources are decreasing in a high concurrency scenario. Therefore, enabling the SMP parallelism will result in serious resource competition among queries. Once resource competition occurs, no matter the CPU, I/O, memory, or network resources, all of them will result in entire performance deterioration. In the high concurrency scenario, enabling the SMP will not improve the performance effect and even may cause performance deterioration.

Suggestions for SMP Parameter Settings

To enable the SMP adaptation function, set **query_dop** to **0**.

 Copyright © Huawei Technologies Co., Ltd.

SMP Configuration Procedure

NOTICE

The CPU, memory, I/O, and network bandwidth resources are sufficient. In essence, SMP is a method that trades resources for time. After the plan parallelism is executed, resource consumption increases. When these resources become a bottleneck, SMP may deteriorate, rather than improve performance. In addition, it takes a longer time to generate SMP plans than serial plans. Therefore, in TP services that mainly involve short queries or in case resources are insufficient, you are advised to disable SMP by setting **query_dop** to **1**.

Procedure:

Step 1 Observe the current system load situation. If the resource is sufficient (the resource usage ratio is smaller than 50%), perform **step 2**. Otherwise, exit this system.

Step 2 Set **query_dop** to **1** (default value). Use **explain** to generate an execution plan and check whether the plan can be used in scenarios described in **Application Scenarios and Constraints for SMP**. If the plan can be used, go to the next step.

Step 3 Set **query_dop=–***value*. The value range of the DOP is [1, *value*].

Step 4 Set **query_dop=***value*. The DOP is 1 or *value*.

Step 5 Before the query statement is executed, set **query_dop** to an appropriate value. After the statement is executed, set **query_dop** to **off**. Example:

```
SET query_dop = 0;
SELECT COUNT(*) FROM t1 GROUP BY a;
......
SET query_dop = 1;
```

NOTE

- If resources are sufficient, the higher the DOP, the better the performance improvement result.
- The SMP DOP supports a session level setting and you are advised to enable SMP before executing queries that meet the requirements. After the execution is complete, disable SMP. Otherwise, SMP may affect services during peak hours.
- SMP adaptability (**query_dop** ≤ 0) depends on resource management. If resource management is disabled (**use_workload_manager** is turned **off**), only plans with DOP of only 1 or 2 will be generated.

----End

13.3 SQL Tuning

13.3.1 SQL Query Execution Process

Figure 13-2 and **Table 13-3** show the process from receiving SQL statements to executing SQL statements by the SQL engine. The texts in red are steps where database administrators can optimize queries.

 Copyright © Huawei Technologies Co., Ltd.

Figure 13-2 Execution process of query-related SQL statements by the SQL engine

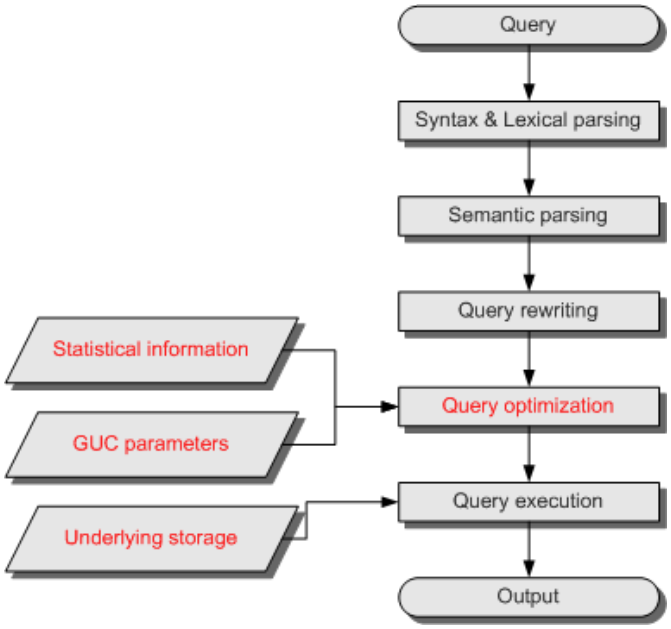


Table 13-3 Execution process of query-related SQL statements by the SQL engine

Step	Description
1. Perform syntax and lexical parsing.	Converts the input SQL statements from the string data type to the formatted structure stmt based on the specified SQL statement rules.
2. Perform semantic parsing.	Converts the formatted structure obtained from the previous step into objects that can be recognized by the database.
3. Rewrite the query statements.	Converts the output of the last step into the structure that optimizes the query execution.
4. Optimize the query.	Determines the execution mode of SQL statements (the execution plan) based on the result obtained from the last step and the internal database statistics. For details about the impact of statistics and GUC parameters on query optimization (execution plan), see **Optimizing Queries Using Statistics** and **Optimizing Queries Using GUC parameters**.
5. Perform the query.	Executes the SQL statements based on the execution path specified in the last step.

Optimizing Queries Using Statistics

The Aura optimizer is a typical cost-based optimization (CBO). The database uses the CBO to calculate the number of tuples and execution cost for each execution step in every execution plan. This calculation is based on factors such as the number of table tuples, column width, NULL record ratio, and characteristic values

 Copyright © Huawei Technologies Co., Ltd.

(such as distinct, MCV, and HB values) using specific cost calculation methods. The database then selects the execution plan with the lowest cost for overall execution or for returning the first tuple. These characteristic values are the statistics, which is the core for optimizing a query. Accurate statistics helps the optimizer select the most appropriate query plan. Generally, you can collect statistics of a table or that of some columns in a table using **ANALYZE**. You are advised to periodically execute **ANALYZE** or execute it immediately after you modified most contents in a table.

Optimizing Queries Using GUC parameters

Optimizing queries aims to select an efficient execution mode.

Take the following statement as an example:

```
SELECT count(1)
FROM customer inner join store_sales on (ss_customer_sk = c_customer_sk);
```

When executing **customer inner join store_sales**, Aura supports two different join methods: Nested Loop and Hash Join. The optimizer estimates the result set value and the execution cost under each join mode based on the statistics of the **customer** and **store_sales** tables and selects the execution plan that takes the lowest execution cost.

As described in the preceding content, the execution cost is calculated based on certain methods and statistics. If the actual execution cost cannot be accurately estimated, you need to optimize the execution plan by setting the GUC parameters.

Optimizing Queries by Rewriting SQL Statements

Besides the preceding methods that improve the performance of the execution plan generated by the SQL engine, database administrators can also enhance SQL statement performance by rewriting SQL statements while retaining the original service logic based on the execution mechanism of the database and abundant practical experience.

This requires that the system administrators know the customer business well and have professional knowledge of SQL statements.

13.3.2 SQL Execution Plan

An SQL execution plan is a node tree that displays the detailed steps performed when Aura executes an SQL statement.

You can run **EXPLAIN** to view the specific execution plan generated by the optimizer for each query. **EXPLAIN** outputs a line for each execution node, displaying the basic node type and the estimated cost value calculated by the optimizer for executing this node.

Execution Plan Display Information

In addition to setting different display formats for the execution plan, various **EXPLAIN** methods can also be used to show execution plan information at different levels of detail. Common ones include the following, and for more usage, refer to **4.2 EXPLAIN**.

 Copyright © Huawei Technologies Co., Ltd.

- **EXPLAIN** *statement*: only generates the execution plan without actual execution. Here, *statement* indicates the SQL statement.
- **EXPLAIN ANALYZE** *statement*: generates the execution plan, performs execution, and displays summary information of the execution. Actual runtime statistics are included in the display, such as the total time spent inside each planning node (in milliseconds) and the actual number of rows it returns.
- **EXPLAIN PERFORMANCE** *statement*: generates the execution plan, performs execution, and displays all information during the execution.

To measure the runtime cost of each node in the execution plan, **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** adds the overhead of performance analysis to the current query execution. Running **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** on a query may take significantly more time than a regular query. The extent of the excess depends on the nature of the query and the platform used.

Therefore, when troubleshooting slow SQL execution issues, if the SQL runs for a long time without completion, you are advised to use the **EXPLAIN** command to view the execution plan for preliminary diagnosis. If the SQL can be executed successfully, you are advised to run **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** to examine the execution plan and its actual runtime information for more precise identification of the issue's cause.

Description of common keywords in execution plans:

- Table access methods

  ForeignScan: full table sequential scan. The most basic scan operator, used for sequential scanning of external tables, supporting value partition pruning.
- Table join methods
  - Nested Loop

    Nested loops, suitable for queries where the connected data subsets are small. In nested loops, the outer table drives the inner table, and every row returned by the outer table must be searched in the inner table to find its matching row. Hence, the result set returned by the entire query should not be too large (not exceeding 10,000), and the table returning the smaller subset should serve as the outer table, with indexes preferably added to the connection fields of the inner table.
  - (Sonic) Hash Join

    Hash joins, appropriate for joining large datasets. The optimizer uses the smaller of the two tables to construct a hash table in memory using the join keys, then scans the larger table and probes the hash to find matches.
- Operators
  - sort

    Sorts the result sets.
  - filter

    EXPLAIN output displays. The **WHERE** clause is used as a filter condition and is attached to the plan node of a sequential scan. This means that the planner checks the condition for each row scanned by the node and outputs only the rows that meet the condition. Although the expected number of output rows is reduced due to the **WHERE** clause, sequential

 Copyright © Huawei Technologies Co., Ltd.

scans still need to access all 10,000 rows. Therefore, the overall cost is not reduced. Instead, additional CPU time (specifically, 10000 × **cpu_operator_cost**) is incurred due to checking the **WHERE** condition.

- LIMIT

  Limits the number of output records in the execution results. If **LIMIT** is applied, not all rows will be retrieved.

Execution Plan Display Format

The execution plan of Aura is displayed in a clear format. The plan contains the plan node ID, and the performance analysis is simple and direct. See **Figure 13-3**.

Figure 13-3 Example of an execution plan in pretty format

```
postgres=> explain select o_orderkey, count(*) from orders group by o_orderkey;
                                  QUERY PLAN
------------------------------------------------------------------------------------------------
 id |                    operation                    |  E-rows   | E-memory | E-width |  E-costs
----+-------------------------------------------------+-----------+----------+---------+------------
  1 | ->  Row Adapter                                 | 150000000 |          |      16 | 10903252.65
  2 |    ->  Vector Streaming (type: GATHER)          | 150000000 |          |      16 | 10903252.65
  3 |       ->  Vector Sonic Hash Aggregate           | 150000000 | 252MB    |      16 | 10902836.02
  4 |          ->  Vector Streaming(type: REDISTRIBUTE) | 150000000 | 2MB      |       8 | 8695322.50
  5 |             ->  Vector Foreign Scan on orders   | 150000000 | 1MB      |       8 | 7500010.00
```

Stream Plan

In Aura, the Stream plan is used to execute queries.

The CoordinatorNode (CN) generates a plan based on the original statement and delivers the plan to DNs for execution. During the execution of each DN, the Stream operator is used for data interaction.

The existing tables **tt01** and **tt02** are defined as follows:

```
CREATE TABLE tt01(c1 int, c2 int)

store as orc;

CREATE TABLE tt02(c1 int, c2 int)
store as orc;
```

Two tables are joined, the join condition contains non-distribution columns, and data is exchanged between DNs. In this case, the base table of the **tt02** table is scanned on each DN. After the scanning, the Redistribute Stream operator is used to perform hash calculation based on **tt02.c1** in the JOIN condition, and the result is sent to each DN. Then, JOIN is performed on each DN. Finally, the data is summarized to the CN.

 Copyright © Huawei Technologies Co., Ltd.

Figure 13-4 Example stream plan

```
postgres=> explain verbose select * from tt01, tt02 where tt01.c1=tt02.c1;
                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 id |                       operation                       | E-rows | E-distinct | E-memory | E-width | E-costs
----+-------------------------------------------------------+--------+------------+----------+---------+---------
  1 | ->  Row Adapter                                       |   5000 |            |          |      16 | 863.08
  2 |    ->  Vector Streaming (type: GATHER)                |   5000 |            |          |      16 | 863.08
  3 |       ->  Vector Sonic Hash Join (4,5)                |   5000 |            | 16MB     |      16 | 238.08
  4 |          ->  Vector Foreign Scan on lmz.tt01          |   1000 |        200 | 1MB      |       8 | 60.00
  5 |          ->  Vector Streaming(type: REDISTRIBUTE)     |   1000 |        200 | 2MB      |       8 | 150.40
  6 |             ->  Vector Foreign Scan on lmz.tt02       |   1000 |            | 1MB      |       8 | 60.00

                          Predicate Information (identified by plan id)
------------------------------------------------------------------------------------------------------------------
  3 --Vector Sonic Hash Join (4,5)
      Hash Cond: (tt01.c1 = tt02.c1)

 Targetlist Information (identified by plan id)
-------------------------------------------------
  1 --Row Adapter
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  2 --Vector Streaming (type: GATHER)
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
      Node/s: All datanodes
  3 --Vector Sonic Hash Join (4,5)
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  4 --Vector Foreign Scan on lmz.tt01
      Output: tt01.c1, tt01.c2
      Distribute Key: tt01.c1
  5 --Vector Streaming(type: REDISTRIBUTE)
      Output: tt02.c1, tt02.c2
      Distribute Key: tt02.c1
      Spawn on: All datanodes
      Consumer Nodes: All datanodes
  6 --Vector Foreign Scan on lmz.tt02
      Output: tt02.c1, tt02.c2
      Distribute Key: tt02.c2
```

EXPLAIN PERFORMANCE Description

During SQL optimization, you need to run **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** to view the actual execution information of SQL statements and compare the actual execution information with the optimizer's estimation to provide a basis for optimization. Compared with **EXPLAIN ANALYZE**, **EXPLAIN PERFORMANCE** adds execution information on each DN.

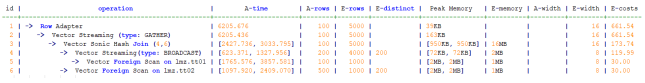
The SQL query statement in the preceding part is used as an example.

```
SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c1;
```

The output of **EXPLAIN PERFORMANCE** consists of the following parts:

1. Execution plan

Figure 13-5 Example explain performance displays



The plan is displayed in a table, including 11 columns: **id**, **operation**, **A-time**, **A-rows**, **E-rows**, **E-distinct**, **Peak Memory**, **E-memory**, **A-width**, **E-width**, and **E-costs**. Table 13-4 describes the columns.

Table 13-4 Execution column descriptions

Column	Description
id	ID of an execution operator node.

 Copyright © Huawei Technologies Co., Ltd.

Column	Description
operation	Name of an execution operator. The operator with the Vector prefix refers to a vectorized execution engine operator, which is generally used in a query containing a column-store table. Streaming is a special operator that implements the core data shuffle function of the distributed architecture. Streaming has three types, which correspond to different data shuffle functions. • **Streaming (type: GATHER)**: The coordinator collects data from DNs. • **Streaming(type: REDISTRIBUTE)**: redistributes data to all DNs based on the selected columns. • **Streaming (type: BROADCAST)**: broadcasts the data of the current DN to all other DNs.
A-time	Execution time of an operator on each DN. Generally, A-time of an operator executed on a DN is two values enclosed by square brackets ([]), indicating the shortest time and longest time for completing the operator on all DNs, including the execution time of lower-layer operators. Note: In the entire plan, the execution time of a leaf node is the execution time of the operator itself, and the execution time of other operators includes the execution time of its child nodes.
A-rows	Total number of global rows output by the corresponding operator.
E-rows	Number of output rows estimated by each operator.
E-distinct	Estimated distinct value of the hashjoin operator.
Peak Memory	Peak memory used by the operator on each DN. The left part of [] indicates the minimum value, and the right part indicates the maximum value.
E-memory	Estimated memory usage of each operator on a DN. Only operators executed on DNs are displayed. In certain scenarios, the upper limit of the memory that can be automatically expanded when the memory resources are sufficient is displayed in parentheses after the estimated memory usage.
A-width	Actual width of each tuple of the current operator. It is displayed only for heavy-memory operators, including (Vec)HashJoin, (Vec)HashAgg, (Vec) HashSetOp, (Vec)Sort, and (Vec) Materialize. The width calculated by (Vec) HashJoin is the width of the right subtree operator, is displayed in the right subtree.
E-width	Estimated width of the output tuple of each operator.

 Copyright © Huawei Technologies Co., Ltd.

Column	Description
E-costs	Estimated execution cost of each operator. • **E-costs** is measured by the optimizer based on the unit defined by the cost parameter. Generally, the disk page is fetched as one unit. Other cost parameters are set based on **E-costs**. • The cost (**E-costs** value) of each node includes the cost of all its child nodes. • The cost reflects only what the optimizer cares about and does not consider the time when the result row is transferred to the client. Although this time may be a significant component of the actual total time, it is ignored by the optimizer because it cannot be changed by modifying the plan.

2. Predicate Information (identified by plan id)

```
                Predicate Information (identified by plan id)
--------------------------------------------------------------------------------
  3 --Vector Sonic Hash Join (4,6)
      Hash Cond: (tt01.c1 = tt02.c1)
      Generate Bloom Filter On Expr: tt02.c1
      Generate Bloom Filter On Index: 0
      Generate Runtime Filter Type: GlobalBloomFilter
  5 --Vector Foreign Scan on lmz.tt01
      Server Type: lf
      Pruning results: (Files total: 10, Files left after metadata pruning: 10)
      DN read from: direct
      executor1_es_group: 5 tasks, basefile 5 files 2.00 MB, logfile 0 files 0.00 MB
      executor0_es_group: 5 tasks, basefile 5 files 2.00 MB, logfile 0 files 0.00 MB
      Filter By Bloom Filter On Expr: tt01.c1
      Filter By Bloom Filter On Index: 0
      Runtime Filter Source Type: Rule
  6 --Vector Foreign Scan on lmz.tt02
      Server Type: lf
      Pruning results: (Files total: 10, Files left after metadata pruning: 10)
      DN read from: direct
      executor1_es_group: 5 tasks, basefile 5 files 2.00 MB, logfile 0 files 0.00 MB
      executor0_es_group: 5 tasks, basefile 5 files 2.00 MB, logfile 0 files 0.00 MB
```

Predicate filtering displays the filter criteria of the corresponding execution operator node, that is, the information that does not change during the entire plan execution process, mainly some join conditions and filter information. For a partitioned table, partition pruning information is also displayed.

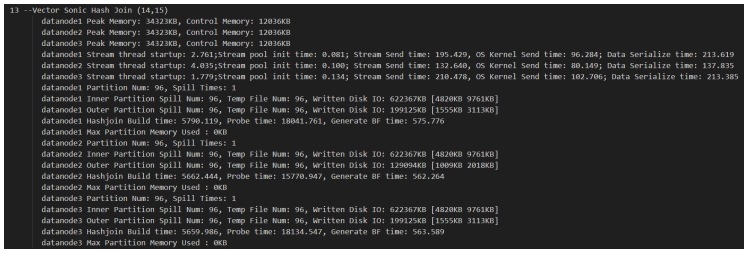
3. Memory Information (identified by plan id)

```
                                Memory Information (identified by plan id)
--------------------------------------------------------------------------------------------------------------------
Coordinator Query Peak Memory:
      Query Peak Memory: 6MB
DataNode Query Peak Memory
      executor0_es_group Query Peak Memory: 7MB
      executor1_es_group Query Peak Memory: 7MB
  1 --Row Adapter
      Peak Memory: 39KB, Estimate Memory: 64MB
  2 --Vector Streaming (type: GATHER)
      Peak Memory: 163KB
  3 --Vector Sonic Hash Join (4,6)
      executor0_es_group Peak Memory: 950KB, Estimate Memory: 16MB
      executor1_es_group Peak Memory: 950KB, Estimate Memory: 16MB
      executor0_es_group Stream Send time: 0.000; Data Serialize time: 0.023
      executor1_es_group Stream Send time: 0.000; Data Serialize time: 0.022
      executor0_es_group Memory Used : 245KB
      executor0_es_group Hashjoin Build time: 0.209, Probe time: 623.561, Generate BF time: 0.202, Generate BF size: 0.2KB
      executor1_es_group Memory Used : 245KB
      executor1_es_group Hashjoin Build time: 0.306, Probe time: 1328.175, Generate BF time: 0.415, Generate BF size: 0.2KB
  4 --Vector Streaming(type: BROADCAST)
      executor0_es_group Peak Memory: 72KB, Estimate Memory: 2MB
      executor1_es_group Peak Memory: 72KB, Estimate Memory: 2MB
      executor0_es_group Stream Network: 2kB, Network Poll Time: 0.000; Data Deserialize Time: 0.043
      executor1_es_group Stream Network: 2kB, Network Poll Time: 0.000; Data Deserialize Time: 0.041
  5 --Vector Foreign Scan on lmz.tt01
      executor0_es_group Peak Memory: 2284KB, Estimate Memory: 1024KB
      executor1_es_group Peak Memory: 2284KB, Estimate Memory: 1024KB
      executor0_es_group Stream thread startup: 3.149;Stream pool init time: 0.190; Stream Send time: 1.150; Data Serialize time: 0.068
      executor1_es_group Stream thread startup: 2.437;Stream pool init time: 0.185; Stream Send time: 1.037; Data Serialize time: 0.078
  6 --Vector Foreign Scan on lmz.tt02
      executor0_es_group Peak Memory: 2284KB, Estimate Memory: 1024KB
      executor1_es_group Peak Memory: 2284KB, Estimate Memory: 1024KB
```

The memory usage information is the memory usage information of the operators that will be printed in the entire plan. The memory usage information includes the peak memory of the operators, estimated memory of the optimizer, and control memory of the Hash and Sort operators, estimated memory usage (operator memory), actual width (width), number of automatic memory expansion times (auto spread num), whether to

 Copyright © Huawei Technologies Co., Ltd.

perform early spilled, and spill information (including spill time (s)), number of inner/outer partition spills (inner/outer partition spill num), number of spill files (temp file num), amount of data spilled, and minimum and maximum data volumes spilled (written disk I/O [min, max]). The Sort operator does not display the specific number of spill files. It only shows **Disk** when displaying the sorting method. Below is an example of the memory information of an operator that has been spilled.



4. Targetlist Information (identified by plan id)

```
Targetlist Information (identified by plan id)
------------------------------------------------------
  1 --Row Adapter
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  2 --Vector Streaming (type: GATHER)
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
      Node/s: All datanodes
  3 --Vector Sonic Hash Join (4,6)
      Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  4 --Vector Streaming(type: BROADCAST)
      Output: tt01.c1, tt01.c2
      Spawn on: All datanodes
      Consumer Nodes: All datanodes
  5 --Vector Foreign Scan on lmz.tt01
      Output: tt01.c1, tt01.c2
      Distribute Key: tt01.c1
  6 --Vector Foreign Scan on lmz.tt02
      Output: tt02.c1, tt02.c2
      Distribute Key: tt02.c2
```

This part displays the output target column information of each operator.

5. DataNode Information (identified by plan id)

```
1 --Row Adapter
    (actual time=6198.563..6205.676 rows=100 loops=1)
    (CPU: ex c/r=5473, ex row=100, ex cyc=547368, inc cyc=13652448830)
2 --Vector Streaming (type: GATHER)
    (actual time=6198.458..6205.436 rows=100 loops=1)
    (Buffers: 0)
    (CPU: ex c/r=136519014, ex row=100, ex cyc=13651901462, inc cyc=13651901462)
3 --Vector Sonic Hash Join (4,6)
    executor0_es_group (actual time=3023.398..3033.795 rows=51 loops=1)
    executor1_es_group (actual time=2170.712..2427.736 rows=49 loops=1)
    executor0_es_group (Buffers: 0)
    executor1_es_group (Buffers: 0)
    executor0_es_group (Disk Cache: hit=25 scanBytes=5.38MB remoteReadBytes=1.00MB loadTime=2379.9 (
    executor1_es_group (Disk Cache: hit=24 miss=1 scanBytes=5.38MB remoteReadBytes=3.41MB loadTime=
    executor0_es_group (CPU: ex c/r=8723, ex row=351, ex cyc=3061778, inc cyc=6674343040)
    executor1_es_group (CPU: ex c/r=11892, ex row=349, ex cyc=4150550, inc cyc=5341021034)
4 --Vector Streaming(type: BROADCAST)
    executor0_es_group (actual time=364.710..623.371 rows=100 loops=1)
    executor1_es_group (actual time=1071.030..1327.956 rows=100 loops=1)
    executor0_es_group (Buffers: 0)
    executor1_es_group (Buffers: 0)
    executor0_es_group (CPU: ex c/r=13714099, ex row=100, ex cyc=1371409942, inc cyc=1371409942)
    executor1_es_group (CPU: ex c/r=29214990, ex row=100, ex cyc=2921499012, inc cyc=2921499012)
```

This part prints out the execution time of each operator (if filtering and projection are included, their execution times are also displayed), CPU usage, and buffer usage.

- Operator execution information

```
3 --Vector Sonic Hash Join (4,6)
    executor0_es_group (actual time=3023.398..3033.795 rows=51 loops=1)
    executor1_es_group (actual time=2170.712..2427.736 rows=49 loops=1)
```

The execution information of each operator consists of three parts:

 Copyright © Huawei Technologies Co., Ltd.

- **executor0_es_group**/**executor1_es_group** indicates the node information where the command is executed. The information in parentheses is the actual execution information.
- **actual time** indicates the actual execution time. The first number indicates the time taken from entering the current operator to outputting the first data record, and the second number represents the total execution time of all data records.
- **rows** indicate the number of output rows of the current operator.
- **loops** indicate the number of executions of the current operator. Note that for a partitioned table, scanning each partition counts as a full scan operation. Switching to the next partition initiates a new scan operation.

– CPU information

```
executor0_es_group (CPU: ex c/r=8723, ex row=351, ex cyc=3061778, inc cyc=6674343040)
executor1_es_group (CPU: ex c/r=11892, ex row=349, ex cyc=4150550, inc cyc=5341021034)
```

Each operator's execution process includes CPU information, where **cyc** indicates the number of CPU cycles, **ex cyc** indicates the cycle count of the current operator excluding its child nodes, **inc cyc** indicates the cycle count including child nodes, **ex row** indicates the number of data rows output by the current operator, and **ex c/r** indicates the average number of cycles per data record calculated by **ex cyc** or **ex row**.

– Buffer information

```
executor0_es_group[worker 0] (Buffers: shared hit=1725 dirtied=117)
executor0_es_group[worker 1] (Buffers: shared hit=1723 dirtied=117)
```

buffers displays buffer information, including reads and writes on shared blocks and temporary blocks.

Shared blocks include tables and indexes, while temporary blocks are disk blocks used in sorting and materialization. The block data displayed at the higher-level node includes the number of blocks used by all its child nodes.

For operators that have been spilled, the buffer information shows the amount of data spilled. **temp read** indicates the number of times temporary data was read, **written** indicates the number of times temporary data was written to disk, and **written_size** indicates the amount of data spilled.

When the GUC switch **enable_spill_to_remote_storage** for the spill-to-OBS feature is turned off, data is spilled into the DN instance directory, with the data displayed in the following format:

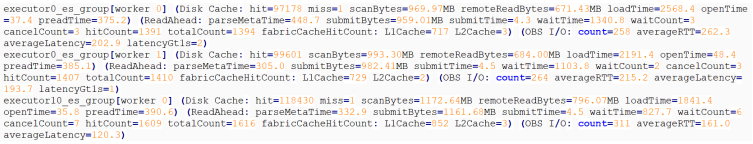
```
executor11_es_group[worker 0] (Buffers: temp read=417509, written=417509, written_size=572640KB)
executor11_es_group[worker 1] (Buffers: temp read=417540, written=417540, written_size=572513KB)
```

When the GUC switch **enable_spill_to_remote_storage** for the spill-to-OBS feature is turned on, additional spill-related statistical information is displayed, as shown in the example below. Among them, **written_disk_size** indicates the amount of data spilled to the disk cache. **written_obs_size** indicates the amount of data directly spilled to OBS when the disk cache space is insufficient. **spill_obs_size** indicates the amount of data written back to OBS from the disk cache when the disk cache space is insufficient.

```
executor11_es_group[worker 0] (Buffers: temp read=58600, written=58600, written_size=804962KB, written_disk_size=804962KB)
executor11_es_group[worker 1] (Buffers: temp read=49228, written=49228, written_size=574675KB, written_disk_size=574675KB, spill_obs_size=39497KB)
```

 Copyright © Huawei Technologies Co., Ltd.

- Disk cache information



  - **Disk Cache**: indicates the hit information and data read information of the disk cache.
    - **miss**: indicates the number of disk cache misses.
    - **hit**: indicates the number of disk cache hits.
    - **disk_cache_error_code** and **error** indicate the number of times an error code is generated.
    - **scanBytes**: indicates the data volume queried by the scan.
    - **remoteReadBytes**: indicates the amount of data read from OBS.
    - **loadTime**: indicates the time taken to load data from the disk cache.
    - **openTime**: indicates the time taken to open the disk cache file.
    - **preadTime**: indicates the time taken to read data from the disk.

      Note: To improve the efficiency of OBS, adjacent request blocks are combined, or the minimum granularity of the request for writing data to the disk cache is block (1 MB by default). As a result, the value of **scanBytes** may be less than that of **remoteReadBytes**.
  - **ReadAhead**: indicates the information about data prefetch.
    - **parseMetaTime**: indicates the time taken to parse file metadata during prefetching.
    - **submitBytes** indicates the amount of prefetched data.
    - **submitTime** indicates the submission time of the data prefetch request.
    - **waitTime**: indicates the time spent waiting for the completion of a prefetch request when it is hit during data reading.
    - **waitCount**: indicates the number of times executing prefetch requests are hit during data reading.
    - **cancelCount**: indicates the number of times pending prefetch requests are hit and canceled during data reading.
    - **hitCount**: indicates the number of times completed prefetch requests are hit during data reading.
    - **diskCacheHitCount**: indicates the number of hits in the L1 layer data cache.
    - **L2Cache**: indicates the number of hits in the L2 layer data cache.
  - **OBS I/O**: details about an OBS I/O request.
    1) **count**: indicates the total number of OBS I/O requests.
    2) **averageRTT**: indicates the average round trip time (RTT) for OBS I/O requests, in microseconds.

 Copyright © Huawei Technologies Co., Ltd.

3) **averageLatency**: indicates the average latency for OBS I/O requests, in microseconds.
4) **latencyGt1s**: indicates the number of OBS I/O requests with latency exceeding 1 second.
5) **latencyGt10s** indicates the number of OBS I/O requests with latency exceeding 10 seconds.
6) **retryCount**: indicates the total number of retries for OBS I/O requests.
7) **rateLimitCount**: indicates the total number of times OBS I/O requests were rate-limited.

– Query Summary



This part displays statement-level execution information, including the execution time in each phase, maximum and minimum execution time in the initialization and end phases on each DN, initialization, execution, and end time on the CN, available system memory, estimated statement memory, and DOP during statement execution.

- DN execution information
  - **DataNode executor start time**: DN executor start time. The format is [min_node_name, max_node_name]: [min_time, max_time].
  - **DataNode executor run time**: DN executor running duration. The format is [min_node_name, max_node_name]: [min_time, max_time].
  - **DataNode executor end time**: DN executor end time. The format is [min_node_name, max_node_name]: [min_time, max_time].
- **Remote query poll time**: poll waiting time for receiving results.
- Memory estimation information
  - **System available mem**: available system memory.
  - **Query Max mem**: maximum query memory.
  - **Query estimated mem**: estimated statement memory.

 Copyright © Huawei Technologies Co., Ltd.

- SMP adaptive concurrency information (displayed only when SMP adaptation is enabled)
  - **Initial DOP**: DOP of the initial plan generated.
  - **Avail max core**: number of vCPUs available for statement execution.
  - **Final Max DOP**: maximum DOP of operators in the plan.
- LakeFormation access information: includes the information access time and frequency.
- CN execution time
  - **Coordinator executor start time**: CN executor start time.
  - **Coordinator executor run time**: CN executor running duration.
  - **Coordinator executor end time**: CN executor end time.
- **Parser runtime**: parser running duration.
- **Planner runtime**: optimizer execution duration.
- **Query Id**: query ID.
- **Total runtime**: total execution duration.

NOTICE

- The difference between **A-rows** and **E-rows** reflects the deviation between the optimizer's estimation and actual execution. Generally, the larger the deviation, the more unreliable the plan generated by the optimizer is, and the more necessary manual intervention becomes.
- The larger the discrepancy in the **A-time** values, the greater the computational skew (difference in execution times across different DNs) for this operator, increasing the necessity for manual tuning. Typically, for two adjacent operators, the execution time of the higher-level operator includes that of the lower-level one. However, if the higher-level operator is a stream operator, its execution time might be shorter than that of the lower-level operator due to the absence of thread dependencies, meaning there is no inclusive relationship.
- **Max Query Peak Memory** is frequently used to estimate the memory consumption of SQL statements and serves as a crucial reference for setting runtime memory parameters during SQL tuning. Usually, the output from **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** is used as input for further optimization.

13.3.3 Execution Plan Operator

Operator Introduction

In a SQL execution plan, each step indicates a database operator, also called an execution operator. In Aura, operators are the building blocks of data processing. By combining them effectively and optimizing their sequence and execution, you can significantly improve data processing efficiency.

 Copyright © Huawei Technologies Co., Ltd.

Aura provides scan operators, control operators, materialization operators, join operators, and other operators.

Scan Operators

A scan operator scans data in a table, processing one tuple at a time for the upper-layer node. It operates at the leaf node of the query plan tree and can scan tables, result sets, linked lists, and subquery results. The following table lists common scan operators.

Table 13-5 Scan operators

Operator	Description	Scenario
VecForeignScan	Sequential scanning	Basic scan operator, used to scan external tables.
VecSubqueryScan	Subquery scanning	Another query plan tree (subplan) is used as the scanning object to scan tuples.
FunctionScan	Function scanning	FROM function_name
ValuesScan	Values linked list scanning	It scans the given tuple set in VALUES clauses.

Join Operators

A join operator corresponds to a join operation in relational algebra. Taking joining **t1** table and **t2** table as an example, the main join types include inner join, left join, right join, full join, semi join, and anti join, with implementations such as Nestloop, HashJoin, and MergeJoin.

Table 13-6 Join operators

Operator	Description	Scenario	Implementation Feature
VecNestLoop	Nested loop join, which is a brute force approach. It scans the inner table for each row.	Inner Join, Left Outer Join, Semi Join, Anti Join	It is used for queries that have a smaller subset connected. In nested loops, the outer table drives the inner table, and every row returned by the outer table must be searched in the inner table to find its matching row. Hence, the result set returned by the entire query should not be too large (not exceeding 10,000), and the table returning the smaller subset should serve as the outer table, with indexes preferably added to the connection fields of the inner table.

 Copyright © Huawei Technologies Co., Ltd.

Operator	Description	Scenario	Implementation Feature
Vector Sonic HashJoin	Hash join: The inner and outer tables use the join column's hash value to create a hash table. Matching values are then stored in the same bucket. The two ends of an equal join must be of the same type and support hash.	Inner Join, Left Outer Join, Right Outer Join, Full Outer Join, Semi Join, Anti Join	Hash joins, appropriate for joining large datasets. The optimizer uses the smaller of the two tables to construct a hash table in memory using the join keys, then scans the larger table and probes the hash to find matches. While Sonic and non-Sonic hash joins have different internal structures, this does not impact the final result set.

Materialization Operators

Materialization operators are a class of nodes that can cache tuples. During execution, many extended physical operations can be performed only after all tuples are obtained, such as aggregation function operations and sorting without indexes. Therefore, materialization operators are required to cache all the tuples.

Table 13-7 Materialization operators

Operator	Description	Scenario
VecMaterial	Materialization	Caches the subnode result.
VecSort	Sorting	ORDER BY clause, which is used for join, group, and set operations and works with Unique.
VecAgg	Executes aggregate functions.	1. Aggregate functions such as COUNT, SUM, AVG, MAX, and MIN. 2. DISTINCT clause. 3. UNION deduplication. 4. GROUP BY clause.
VecWindowAgg	Window functions	WINDOW clause.
VecSetOp	Processing set operations	INTERSECT/INTERSECT ALL, EXCEPT/EXCEPT ALL

 Copyright © Huawei Technologies Co., Ltd.

Control Operators

Control operators are a type of node that handles exceptional scenarios and executes custom workflows.

Table 13-8 Control operators

Operator	Description	Scenario
VecResult	Performing calculation directly	1. Table scanning is not included. 2. The **INSERT** statement contains only one **VALUES** clause.
VecModifyTable	INSERT/UPDATE/DELETE upper-layer node	**INSERT**, **UPDATE**, and **DELETE**
VecAppend	Appending	1. **UNION(ALL)** 2. Table inheritance
VecLimit	Processing the LIMIT clause	**OFFSET ... LIMIT ...**

Other Operators

Other operators include VecStream and RemoteQuery. There are three types of Stream operators: Gather stream, Broadcast stream, and Redistribute stream.

- Gather stream: Each source node sends its data to the target node for aggregation.
- Broadcast stream: A source node sends its data to N target nodes for calculation.
- Redistribute stream: Each source node calculates the hash value of its data based on the join condition, distributes the data based on the hash value, and sends the data to the corresponding target node.

Table 13-9 Other Operators

Operator	Description	Scenario
VecStream	Multi-node data exchange	When a distributed query plan is executed, data is exchanged between nodes.
RowToVec	Rows-to-column conversion	Hybrid row-column.

 Copyright © Huawei Technologies Co., Ltd.

13.3.4 SQL Execution Monitoring

SQL Historical Monitoring Data

Historical SQL monitoring data records the resource usage (including memory, disk spilling, CN and DN duration, OBS access time, and LakeFormation access time) and SQL execution plan information (EXPLAIN/EXPLAIN PERFORMANCE). Historical monitoring data is generated only after the SQL statement is executed.

The fields displayed in the SQL historical monitoring data are as follows:

Table 13-10 Fields displayed in SQL historical monitoring data

Field Name	Subfield Name	Type	Description
session_id	-	text	Session ID of a statement.
statement_id	-	text	Statement ID.
start_time	-	timestamp with time zone	Start time of a statement.
finish_time	-	timestamp with time zone	Reported/End time of a statement.
duration	-	bigint	Statement running duration. Unit: ms.
cn_actor_info	-	text	Serverless information of a CN.
	actor_name	text	Name of the pod where a CN is.
	actor_ip	text	IP address of the pod where a CN is.
	actor_id	text	Name of a CN within a pod.
dn_actor_info	-	text	Serverless information of a DN.
	actor_name	text	Name of the pod where a DN is.
	actor_ip	text	ID of the pod where a DN is.
	actor_id	text	Name of a DN within a pod.
startup_coordinator_duration	-	bigint	Duration to start a CN. Unit: ms.
create_session_duration	-	bigint	Duration to create a session. Unit: ms.

 Copyright © Huawei Technologies Co., Ltd.

Field Name	Subfield Name	Type	Description
enqueue_statement_duration	-	bigint	Duration to add a statement to a task queue. Unit: ms.
queue_duration	-	bigint	Queuing duration of a statement in a task queue. Unit: ms.
write_resultset_duration	-	bigint	Duration to write a statement to a result set. Unit: ms.
queryid	-	bigint	ID of a debug query.
status	-	text	Statement status.
abort_info	-	text	Error message.
schemaname	-	text	Schema name when a statement is executed.
query	-	text	Statement.
query_plan	-	text	Explain information when GUC parameter **resource_track_level** is set to **query**. Explain performance information when set to **perf**.
tid	-	bigint	Thread ID.
parse_time	-	bigint	Time from parsing to execution start. Unit: ms.
estimate_memory	-	int	Estimated memory of a statement. Unit: MB.
dn_time	-	text	DN duration information.
	min_dn_time	bigint	Minimum DN duration. Unit: ms.
	max_dn_time	bigint	Maximum DN duration. Unit: ms.
	average_dn_time	bigint	Average DN duration. Unit: ms.
	dntime_skew_percent	int	Skew percentage of DN duration. Unit: %.
dn_cpu_time	-	text	DN CPU duration.

 Copyright © Huawei Technologies Co., Ltd.

Field Name	Subfield Name	Type	Description
	min_cpu_time	bigint	Minimum DN CPU duration. Unit: ms.
	max_cpu_time	bigint	Maximum DN CPU duration. Unit: ms.
	average_cpu_time	bigint	Average DN CPU duration. Unit: ms.
	total_cpu_time	bigint	Total DN CPU duration. Unit: ms.
	cpu_skew_percent	int	Skew percentage of DN CPU duration. Unit: %.
dn_peak_memory	-	text	DN peak memory information.
	min_peak_memory	int	Minimum peak memory of all DNs. Unit: MB.
	max_peak_memory	int	Maximum peak memory of all DNs. Unit: MB.
	average_peak_memory	int	Average peak memory of all DNs. Unit: MB.
	memory_skew_percent	int	Memory skew rate. Unit: %.
cn_spill_info	-	text	CN disk spill information.
	spill_count	bigint	Number of spills from a CN.
	spill_size	bigint	CN disk spill size. Unit: MB.
	spill_obs_size	bigint	Size of data spilled to OBS by a CN. Unit: MB.
	write_obs_size	bigint	Size of data directly written to OBS by a CN. Unit: MB.
	write_disk_size	bigint	Size of data written to local disks by a CN. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Field Name	Subfield Name	Type	Description
dn_spill_info	-	text	DN disk spill information.
	spill_dn_num	int	Number of DNs with disk spills.
	spill_count	int	Total number of DN disk spills.
	min_spill_size	bigint	Minimum spill size. Unit: MB.
	max_spill_size	bigint	Maximum spill size. Unit: MB.
	average_spill_size	bigint	Average spill size. Unit: MB.
	spill_skew_percent	int	Spill skew rate. Unit: %.
	write_disk_total_size	bigint	Total amount of data spilled to local disks by a DN. Unit: MB.
	write_disk_max_size	bigint	Maximum amount of data written to local disks by a DN. Unit: MB.
	write_disk_min_size	bigint	Minimum amount of data written to local disks by a DN. Unit: MB.
	spill_obs_total_size	bigint	Total amount of data spilled to OBS by a DN. Unit: MB.
	spill_obs_max_size	bigint	Maximum amount of data spilled to OBS by a DN. Unit: MB.
	spill_obs_min_size	bigint	Minimum amount of data spilled to OBS by a DN. Unit: MB.
	write_obs_total_size	bigint	Total amount of data directly written to OBS by a DN. Unit: MB.
	write_obs_max_size	bigint	Maximum amount of data directly written to OBS by a DN. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Field Name	Subfield Name	Type	Description
	write_obs_min_size	bigint	Minimum amount of data directly written to OBS by a DN. Unit: MB.
disk_cache_info	-	text	Cache information.
	disk_cache_hit_ratio	numeric	Disk cache hit rate.
	disk_cache_disk_read_size	bigint	Read size of disk cache. Unit: byte.
	disk_cache_disk_write_size	bigint	Write size of disk cache. Unit: byte.
	disk_cache_remote_read_size	bigint	Remote read size of disk cache. Unit: byte.
	disk_cache_remote_read_time	bigint	Count of remote reads from disk cache. Unit: times.
obs_info	-	text	OBS information.
	vfs_scan_bytes	bigint	Bytes scanned by the OBS file system. Unit: byte.
	vfs_remote_read_bytes	bigint	Bytes remotely read by the OBS file system. Unit: byte.
	disk_cache_load_time	bigint	Local load time of disk cache. Unit: μs.
	disk_cache_conflict_count	bigint	Hash collision count of disk cache blocks. Unit: times.
	disk_cache_error_count	bigint	Error count of disk cache. Unit: times.
	disk_cache_error_code	bigint	Error code of disk cache.
	obs_io_req_avg_rtt	bigint	Average round-trip time of OBS I/O requests. Unit: μs.

 Copyright © Huawei Technologies Co., Ltd.

Field Name	Subfield Name	Type	Description
	obs_io_req_avg_latency	bigint	Average latency of OBS I/O requests. Unit: µs.
	obs_io_req_latency_gt_1s	bigint	Count of OBS I/O request delays exceeding 1s. Unit: times.
	obs_io_req_latency_gt_10s	bigint	Count of OBS I/O request delays exceeding 10s. Unit: times.
	obs_io_req_count	bigint	Number of OBS I/O requests. Unit: times.
	obs_io_req_retry_count	bigint	Number of OBS I/O request retries. Unit: times.
	obs_io_req_rate_limit_count	bigint	Rate limit count of OBS I/O requests. Unit: times.
yr_info	-	text	YuanRong API invocation information.
	total_call	bigint	Total number of calls.
	failed_call	bigint	Number of failed calls.
	avg_call_during	bigint	Average API invocation duration. Unit: millisecond.
	min_call_during	bigint	Minimum calling duration. Unit: millisecond.
	max_call_during	bigint	Maximum calling duration. Unit: millisecond.
	total_data	bigint	Total data volume. Unit: byte.
	avg_data_rate	bigint	Average data processing rate. Unit: byte/ms.
	min_data_rate	bigint	Minimum data processing rate. Unit: byte/ms.
	max_data_rate	bigint	Maximum data processing rate. Unit: byte/ms.

 Copyright © Huawei Technologies Co., Ltd.

Current SQL monitoring data is stored in the OBS bucket of the result set, in the same directory as the result set folder. It can be accessed through the OBS service on the Huawei Cloud website. Based on the path of the current query's result set, locate the corresponding SQL monitoring data file in the same directory and download it for viewing. Upon opening the SQL monitoring data file, the following information is displayed:

Figure 13-6 Content of a historical SQL monitoring information file



SQL Real-Time Monitoring Data

Real-time SQL monitoring data also records the resource usage of query jobs (including memory, disk spillage, CN and DN durations, OBS access time, and LakeFormation access time) as well as SQL execution plan information (EXPLAIN/ EXPLAIN PERFORMANCE). However, the real-time SQL monitoring data can only retrieve information for statements currently being executed. It cannot obtain information for statements that have already completed execution.

Usage: select * from gs_wlm_session_statistics();

The fields displayed in the SQL real-time monitoring data are as follows:

Table 13-11 Fields displayed in SQL real-time monitoring data

Field	Type	Description
session_id	text	Session ID of a statement.
statement_id	text	Statement ID.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
start_time	timestamp with time zone	Start time of a statement.
duration	bigint	Statement running duration. Unit: ms.
estimate_total_time	bigint	Estimated statement running duration. Unit: ms.
estimate_left_time	bigint	Estimated remaining running duration of a statement Unit: ms.
cn_actor_info	text	Serverless information of a CN.
dn_actor_info	text	Serverless information of a DN.
startup_coordinator_duration	bigint	Duration to start a CN. Unit: ms.
create_session_duration	bigint	Duration to create a session. Unit: ms.
enqueue_statement_duration	bigint	Duration to add a statement to a task queue. Unit: ms.
queue_duration	bigint	Queuing duration of a statement in a task queue. Unit: ms.
write_resultset_duration	bigint	Duration to write a statement to a result set. Unit: ms.
queryid	bigint	ID of a debug query.
schemaname	text	Schema name when a statement is executed.
query	text	Statement.
query_plan	text	Explain information
tid	bigint	Thread ID.
parse_time	bigint	Time from parsing to execution start. Unit: ms.
estimate_memory	int	Estimated memory of a statement. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
min_dn_time	bigint	Minimum DN duration. Unit: ms.
max_dn_time	bigint	Maximum DN duration. Unit: ms.
average_dn_time	bigint	Average DN duration. Unit: ms.
dn_time_skew_percent	int	Skew percentage of DN duration. Unit: %.
dn_min_cpu_time	bigint	Minimum DN CPU duration. Unit: ms.
dn_max_cpu_time	bigint	Maximum DN CPU duration. Unit: ms.
dn_average_cpu_time	bigint	Average DN CPU duration. Unit: ms.
dn_total_cpu_time	bigint	Total DN CPU duration. Unit: ms.
dn_cpu_skew_percent	int	Skew percentage of DN CPU duration. Unit: %.
dn_min_peak_memory	int	Minimum peak memory of all DNs. Unit: MB.
dn_max_peak_memory	int	Maximum peak memory of all DNs. Unit: MB.
dn_average_peak_memory	int	Average peak memory of all DNs. Unit: MB.
dn_memory_skew_percent	int	Memory skew rate. Unit: %.
dn_spill_num	int	Number of DNs with disk spills.
dn_spill_count	int	Total number of DN spills.
dn_min_spill_size	bigint	Minimum spill size. Unit: MB.
dn_max_spill_size	bigint	Maximum spill size. Unit: MB.
dn_average_spill_size	bigint	Average spill size. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
dn_spill_skew_percent	int	Spill skew rate. Unit: %.
dn_write_disk_total_size	bigint	Total size of data written to disks by a DN. Unit: MB.
dn_write_disk_max_size	bigint	Maximum amount of data written to disks by a DN. Unit: MB.
dn_write_disk_min_size	bigint	Minimum amount of data written to disks by a DN. Unit: MB.
dn_spill_obs_total_size	bigint	Total amount of data spilled to OBS by a DN. Unit: MB.
dn_spill_obs_max_size	bigint	Maximum amount of data spilled to OBS by a DN. Unit: MB.
dn_spill_obs_min_size	bigint	Minimum amount of data spilled to OBS by a DN. Unit: MB.
dn_write_obs_total_size	bigint	Total amount of data directly written to OBS by a DN. Unit: MB.
dn_write_obs_max_size	bigint	Maximum amount of data directly written to OBS by a DN. Unit: MB.
dn_write_obs_min_size	bigint	Minimum amount of data directly written to OBS by a DN. Unit: MB.
cn_spill_count	int	Number of spills from a CN.
cn_spill_size	bigint	Total size of data spilled from a CN. Unit: MB.
cn_spill_obs_size	bigint	Amount of data spilled to OBS by a CN. Unit: MB.
cn_write_obs_size	bigint	Amount of data directly written to OBS by a CN. Unit: MB.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
cn_write_obs_disk_size	bigint	Amount of data spilled to local disks by a CN. Unit: MB.
disk_cache_hit_ratio	numeric	Disk cache hit rate.
disk_cache_disk_read_size	bigint	Read size of disk cache. Unit: byte.
disk_cache_disk_write_size	bigint	Write size of disk cache. Unit: byte.
disk_cache_remote_read_size	bigint	Remote read size of disk cache. Unit: byte.
disk_cache_remote_read_time	bigint	Count of remote reads from disk cache. Unit: times.
vfs_scan_bytes	bigint	Bytes scanned by the OBS file system. Unit: byte.
vfs_remote_read_bytes	bigint	Bytes remotely read by the OBS file system. Unit: byte.
disk_cache_load_time	bigint	Local load time of disk cache. Unit: μs.
disk_cache_conflict_count	bigint	Hash collision count of disk cache blocks. Unit: times.
disk_cache_error_count	bigint	Error count of disk cache. Unit: times.
disk_cache_error_code	bigint	Error code of disk cache.
obs_io_req_avg_rtt	bigint	Average round-trip time of OBS I/O requests. Unit: μs.
obs_io_req_avg_latency	bigint	Average latency of OBS I/O requests. Unit: μs.
obs_io_req_latency_gt_1s	bigint	Count of OBS I/O request delays exceeding 1s. Unit: times.
obs_io_req_latency_gt_10s	bigint	Count of OBS I/O request delays exceeding 10s. Unit: times.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
obs_io_req_count	bigint	Number of OBS I/O requests. Unit: times.
obs_io_req_retry_count	bigint	Number of OBS I/O request retries. Unit: times.
obs_io_req_rate_limit_count	bigint	Rate limit count of OBS I/O requests. Unit: times.

13.3.5 SQL Tuning Process

Analyze slow SQL statements by performing the following operations:

Procedure

Step 1 Collect all table statistics associated with the SQL statements. In a database, statistics serve as the raw data for the planner to generate execution plans. Failure to collect statistics or having outdated statistics can lead to severe degradation of execution plans, resulting in performance issues. According to the experience data, about 10% performance issues occurred because no statistics are collected. For details, see **13.3.6 Updating Statistics**.

Step 2 Generally, some SQL statements can be converted to its equivalent statements in all or certain scenarios by rewriting queries. SQL statements are simpler after they are rewritten. Some execution steps can be simplified to improve the performance. The query rewriting method is universal in all databases. **13.3.7.3 SQL Statement Rewriting Rules** introduces several common tuning methods through SQL statement rewriting.

Step 3 View the execution plan to find out the cause. If the SQL statements have been running for a long period of time and not ended, run the **EXPLAIN** command to view the execution plan and then locate the fault. If the SQL statement has been executed, run the **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** command to check the execution plan and actual running situation and then accurately locate the fault. For details about the execution plan, see **13.3.2 SQL Execution Plan**.

Step 4 For details about locating the specific causes of slow SQL execution and providing solutions based on the **EXPLAIN** or **EXPLAIN PERFORMANCE** information, see **13.3.7 Advanced SQL Tuning**.

Step 5 Specify a join order; join, stream, or scan operations; number of rows in a result; or redistribution skew information to optimize an execution plan, improving query performance. For details, see **13.3.9 Hint-based Tuning**.

Step 6 (Optional) Improve performance by using operators if resources are sufficient in Aura. For details, see **13.2.2 SMP Parallel Execution**.

----End

 Copyright © Huawei Technologies Co., Ltd.

13.3.6 Updating Statistics

In a database, statistics serve as the raw data for the planner to generate execution plans. Failure to collect statistics or having outdated statistics can lead to severe degradation of execution plans, resulting in performance issues.

Context

The **ANALYZE** statement gathers statistical information related to the content of tables within the database, with the results stored in the system catalog **PG_STATISTIC**. The query optimizer uses these statistics to produce the most efficient execution plan.

You are advised to routinely execute the **ANALYZE** statement on tables or the entire database following batch insert/delete operations to update statistics. Currently, the default sampling rate for collecting statistics is 30,000 rows (that is, the GUC parameter **default_statistics_target** is set to **100** by default). If the total row count of a table exceeds a certain threshold (greater than 1,600,000), you are advised to set **default_statistics_target** to **-2**, meaning statistics are estimated based on a 2% sample.

Intermediate tables generated using batch processing scripts or stored procedures should also explicitly call **ANALYZE** upon completion of data generation.

For scenarios where multiple columns in a table exhibit correlation and queries involve simultaneous conditional or group operations based on these columns, consider gathering multi-column statistics. This enables the query optimizer to more accurately estimate row counts and generate more effective execution plans.

Generating Statistics

Update statistics on a single table.

```
ANALYZE tablename;
```

NOTE

When examining the execution plan of various SQL statements using **EXPLAIN**, encountering **rows=10** in the **SEQ SCAN** output of a table suggests the default system-assigned value, potentially indicating the absence of **ANALYZE** execution on that table. Thus, you need to execute **ANALYZE** on the table.

Enhancing Statistical Quality

ANALYZE uses a random sampling algorithm to extract samples from a table, computing table data characteristics based on these samples. Sample size can be controlled through the configuration parameter **default_statistics_target**, which ranges from -100 to 10000, with a default value of **100**.

- When **default_statistics_target** > 0: The sample size equals 300 x **default_statistics_target**. Larger values result in bigger samples, increased memory usage, and prolonged computation times for statistics.
- When **default_statistics_target** is less than 0: The sample size corresponds to **default_statistics_target**/100 x Total table rows. Smaller values yield smaller samples. When **default_statistics_target** is less than 0, sampled data is written to disks, eliminating memory concerns but introducing extended computation durations due to substantial sample sizes.

 Copyright © Huawei Technologies Co., Ltd.

Negative **default_statistics_target** implies actual sampling equates to **default_statistics_target**/100 x Total number of table rows, hence termed percentage sampling.

Automatic Collection of Statistics

With the **autoanalyze** parameter enabled, when the optimizer detects missing statistics or significant data alterations exceeding thresholds during query optimization, automatic statistic collection is triggered to fulfill optimizer demands.

Within the cost-based optimizer (CBO) framework, statistics determine the efficacy of query plan generation, underscoring their timeliness and accuracy's paramount importance.

- Table-level statistics reside in **relpages** and **reltuples** of **pg_class**.
- Column-level statistics are housed in **pg_statistics**, accessible through the **pg_stats** view, including the percentage of NULL values, percentage of distinct values, high-frequency MCV values, and histograms.

Collection criteria: Significant data shifts—typically a 10% variation—prompt re-collection of statistics under the assumption that data characteristics have altered.

Accuracy

Table 13-12 Accuracy

Category	Description
Sampling size	Configurable to adapt according to table size, governed by the **default_statistics_target** parameter.
Sampling randomness	• The **analyze_sample_mode** parameter optimizes reservoir and range sampling randomness. • Enhance randomness of random number calculation with the **random_function_version** parameter.
Statistical estimation	The **enable_extrapolation_stats** parameter allows for automatic calculation of more accurate statistics based on old statistics during distortion estimation.

Viewing Statistics

For details, see **pg_get_statistics()** and **pg_get_partition_statistics()**.

13.3.7 Advanced SQL Tuning

 Copyright © Huawei Technologies Co., Ltd.

13.3.7.1 Optimizing Subqueries

Subquery Background

Applications use a significant amount of subqueries when performing operations on databases using SQL statements. This approach offers greater clarity in both structure and thought process compared to directly joining two tables. Particularly in more complex queries, subqueries provide more comprehensive and independent semantics, making SQL expressions of service logic clearer and easier to comprehend, hence their widespread adoption.

Subqueries are classified into subqueries and sublinks based on the location of subqueries in SQL statements.

- Subquery: corresponds to a scope table (RangeTblEntry) in the query parse tree. That is, a subquery is a **SELECT** statement following immediately after the **FROM** keyword.
- Sublink: corresponds to an expression in the query parsing tree. That is, a sublink is a statement in the **WHERE** or **ON** clause or in the target list.

  In summary, within the context of the query parse tree, a subquery fundamentally represents a range table, whereas a sublink embodies an expression. Sublinks can appear in constraints and expressions. Based on the implementation of sublinks in Aura, sublinks can be classified into the following types:

  - **exist_sublink**: corresponds to the **EXIST** and **NOT EXIST** statements.
  - **any_sublink**: corresponds to the *op* **Any(SELECT...)** statement. *op* can be the **IN**, **<**, **>**, or **=** operator.
  - **all_sublink**: corresponds to the *op* **ALL(SELECT...)** statement. *op* can be the **IN**, **<**, **>**, or **=** operator.
  - **rowcompare_sublink**: corresponds to the **RECORD** *op* **(SELECT...)** statement.
  - **expr_sublink**: corresponds to the **(SELECT** *with a single target list item...*) statement.
  - **array_sublink**: corresponds to the **ARRAY(SELECT...)** statement.
  - **cte_sublink**: corresponds to the **WITH(...)** query statement.

  Among them, the sublinks commonly used in OLAP and HTAP scenarios are **exist_sublink** and **any_sublink**. The Aura optimization engine has optimized their application scenarios (sublink promotion). However, due to the flexible usage of subqueries in SQL, overly complex subqueries can lead to performance issues. Broadly speaking, subqueries are categorized into non-correlated subqueries and correlated subqueries.

  - **Non-correlated subquery**

    The execution of a subquery is independent from any attribute of outer queries. In this way, a subquery can be executed before outer queries.

    Example:

    ```
    select t1.c1,t1.c2
    from t1
    where t1.c1 in (
       select c2
       from t2
    ```

 Copyright © Huawei Technologies Co., Ltd.

```
   where t2.c2 IN (2,3,4)
);
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 id |                    operation                        | E-rows | E-memory | E-width | E-costs
----+-----------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                     |      8 |          |       8 | 137.16
  2 |   ->  Vector Streaming (type: GATHER)               |      8 |          |       8 | 137.16
  3 |     ->  Vector Nest Loop Semi Join (4, 6)           |      8 | 1MB      |       8 | 129.16
  4 |       ->  Vector Streaming(type: REDISTRIBUTE)      |     15 | 2MB      |       8 | 64.20
  5 |         ->  Vector Foreign Scan on t1               |     15 | 1MB      |       8 | 63.75
  6 |       ->  Vector Materialize                        |     15 | 16MB     |       4 | 64.24
  7 |         ->  Vector Streaming(type: REDISTRIBUTE)    |     15 | 2MB      |       4 | 64.20
  8 |           ->  Vector Foreign Scan on t2             |     15 | 1MB      |       4 | 63.75

  Predicate Information (identified by plan id)
-------------------------------------------------
  3 --Vector Nest Loop Semi Join (4, 6)
        Join Filter: (t1.c1 = t2.c2)
  5 --Vector Foreign Scan on t1
        Filter: (c1 = ANY ('{2,3,4}'::integer[]))
        Server Type: lf
        Total files left: 0
  8 --Vector Foreign Scan on t2
        Filter: (c2 = ANY ('{2,3,4}'::integer[]))
        Server Type: lf
        Total files left: 0
```

– **Correlated subquery**

The execution of a subquery depends on some attributes of outer queries which are used as **AND** conditions of the subquery. In the following example, **t1.c1** in the **t2.c1 = t1.c1** condition is a dependent attribute. Such a subquery depends on outer queries and needs to be executed once for each outer query.

Example:

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
   select c2
   from t2
   where t2.c1 = t1.c1 AND t2.c2 in (2,3,4)
);
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 id |                    operation                        | E-rows | E-memory | E-width | E-costs
----+-----------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                     |    500 |          |       8 | 32454.80
  2 |   ->  Vector Streaming (type: GATHER)               |    500 |          |       8 | 32454.80
  3 |     ->  Vector Foreign Scan on t1                   |    500 | 1MB      |       8 | 32433.93
  4 |       ->  Row Adapter  [3, SubPlan 1]               |    135 | 1MB      |       8 | 64.41
  5 |         ->  Vector Result                           |    135 | 1MB      |       8 | 64.41
  6 |           ->  Vector Materialize                    |     45 | 16MB     |       8 | 64.41
  7 |             ->  Vector Streaming(type: BROADCAST)   |     45 | 2MB      |       8 | 64.33
  8 |               ->  Vector Foreign Scan on t2         |     15 | 1MB      |       8 | 63.75

  Predicate Information (identified by plan id)
-------------------------------------------------
  3 --Vector Foreign Scan on t1
        Filter: (SubPlan 1)
        Server Type: lf
        Total files left: 0
  5 --Vector Result
        Filter: (t2.c1 = t1.c1)
  8 --Vector Foreign Scan on t2
        Filter: (c2 = ANY ('{2,3,4}'::integer[]))
        Server Type: lf
        Total files left: 0
```

Aura SubLink Optimization

A subquery is pulled up to join with tables in outer queries, preventing the subquery from being converted into the combination of a subplan and broadcast. You can run the **EXPLAIN** statement to check whether a subquery is converted into the combination of a subplan and broadcast.

- **Sublink-release supported by Aura**
  - Pulling up the **IN** sublink
    - The subquery cannot contain columns in the outer query (columns in more outer queries are allowed).
    - The subquery cannot contain volatile functions.
  - Pulling up the **EXIST** sublink

    The **WHERE** clause must include columns from the table in the outer query. Other parts of the subquery must not contain columns from the table in the outer query. Additional restrictions are as follows:
    - The subquery must contain the **FROM** clause.
    - The subquery cannot contain the **WITH** clause.

 Copyright © Huawei Technologies Co., Ltd.

- The subquery cannot contain aggregate functions.
- The subquery cannot contain a **SET**, **SORT**, **LIMIT**, **WindowAgg**, or **HAVING** operation.
- The subquery cannot contain volatile functions.

– Pulling up an equivalent query containing aggregation functions

The **WHERE** condition of the subquery must contain a column from the outer query. Equivalence comparison must be performed between this column and related columns in tables of the subquery. These conditions must be connected using **AND**. No columns from the outer query may appear elsewhere. Additional restrictions are as follows:

- The expression in the **WHERE** condition of the subquery must be table columns.
- After the **SELECT** keyword of the subquery, there must be only one output column. The output column must be an aggregate function (for example, **MAX**), and the parameter (for example, **t2.c2**) of the aggregate function cannot be columns of a table (for example, **t1**) in outer queries. The aggregate function cannot be **COUNT**.

  For example, the following subquery can be pulled up:

  ```
  select * from t1 where c1 >(
      select max(t2.c1) from t2 where t2.c1=t1.c1
  );
  ```

  The following subquery cannot be pulled up because the subquery has no aggregation function.

  ```
  select * from t1 where c1 >(
      select  t2.c1 from t2 where t2.c1=t1.c1
  );
  ```

  The following subquery cannot be pulled up because the subquery has two output columns:

  ```
  select * from t1 where (c1,c2) >(
      select  max(t2.c1),min(t2.c2) from t2 where t2.c1=t1.c1
  );
  ```

- The subquery must be a **FROM** clause.
- The subquery cannot contain a **GROUP BY**, **HAVING**, or **SET** operation.
- The subquery can only be inner join.

  For example, the following subquery can be pulled up:

  ```
  select * from t1 where c1 >(
      select max(t2.c1) from t2 full join t3 on (t2.c2=t3.c2) where t2.c1=t1.c1
  );
  ```

- The target list of the subquery cannot contain the function that returns a set.
- The **WHERE** condition of the subquery must contain a column from the outer query. Equivalence comparison must be performed between this column and related columns in tables of the subquery. These conditions must be connected using **AND**. No columns from the outer query may appear elsewhere. For example, the following subquery can be pulled up:

 Copyright © Huawei Technologies Co., Ltd.

```
select * from t3 where t3.c1=(
      select t1.c1
      from t1 where c1 >(
            select max(t2.c1) from t2 where t2.c1=t1.c1
));
```

If another condition is added to the subquery in the previous example, the subquery cannot be pulled up because the subquery references to the column in the outer query. Example:

```
select * from t3 where t3.c1=(
      select t1.c1
      from t1 where c1 >(
            select max(t2.c1) from t2 where t2.c1=t1.c1 and t3.c1>t2.c2

));
```

– Pulling up a sublink in the **OR** clause

When the **WHERE** filter condition contains an **EXIST**-related sublink connected by **OR**, the example is as follows:

```
select a, c from t1
where t1.a = (select avg(a) from t3 where t1.b = t3.b) or
exists (select * from t4 where t1.c = t4.c);
```

The process of promoting the **EXIST**-related subquery (**OR** clause) connected by **OR** is as follows:

i. Extract the **opExpr** in the **OR** clause from the **WHERE** condition: **t1.a = (select avg(a) from t3 where t1.b = t3.b)**.

ii. If the identified **opExpr** contains a subquery that can be optimized, transform it into a more efficient form. Rewrite the subquery as: **SELECT AVG(a), t3.b FROM t3 GROUP BY t3.b**. Add a **NOT NULL** constraint (**t3.b IS NOT NULL**) to ensure data integrity. Finally, replace the original **opExpr** with this new condition. In this case, the SQL statement changes to:

```
select a, c
from t1 left join (select avg(a) avg, t3.b from t3 group by t3.b)  as t3 on (t1.a = avg and t1.b = t3.b)
where t3.b is not null or exists (select * from t4 where t1.c = t4.c);
```

iii. Extract the **EXISTS** sublink **exists (select * from t4 where t1.c = t4.c)** from the **OR** clause to check whether the sublink can be pulled up. If it can be pulled up, it is converted into **select t4.c from t4 group by t4.c**, generating the **NOT NULL** condition **t4.c is not null**. In this case, the SQL statement changes to:

```
select a, c
from t1 left join (select avg(a) avg, t3.b from t3 group by t3.b)  as t3 on (t1.a = avg and t1.b = t3.b)
left join (select t4.c from t4 group by t4.c) where t3.b is not null or t4.c is not null;
```

- **Sublink-release not supported by Aura**

Except the sublinks described above, all the other sublinks cannot be pulled up. In this case, a join subquery is planned as the combination of a subplan and broadcast. As a result, if tables in the subquery have a large amount of data, query performance may be poor.

If a correlated subquery joins with two tables in outer queries, the subquery cannot be pulled up. You need to change the parent query into a **WITH** clause and then perform the join.

Example:

```
select distinct t1.a, t2.a
from t1 left join t2 on t1.a=t2.a and not exists (select a,b from test1 where test1.a=t1.a and test1.b=t2.a);
```

 Copyright © Huawei Technologies Co., Ltd.

The parent query is changed into:

```
with temp as
(
      select * from (select t1.a as a, t2.a as b from t1 left join t2 on t1.a=t2.a)

)
select distinct a,b
from temp
where not exists (select a,b from test1 where temp.a=test1.a and temp.b=test1.b);
```

– The subquery (without **COUNT**) in the target list cannot be pulled up.

  Example:

```
explain (costs off)
select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
from t1
where t1.c2 > 10;
```

  The execution plan is as follows:

```
explain (costs off)
select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
from t1
where t1.c2 > 10;
```

```
                        QUERY PLAN
-----------------------------------------------------------------
  id |                       operation
 ----+-------------------------------------------------------
   1 | ->  Row Adapter
   2 |    ->  Vector Streaming (type: GATHER)
   3 |       ->  Vector Foreign Scan on t1
   4 |          ->  Row Adapter  [3, SubPlan 1]
   5 |             ->  Vector Result
   6 |                ->  Vector Materialize
   7 |                   ->  Vector Streaming(type: BROADCAST)
   8 |                      ->  Vector Foreign Scan on t2

 Predicate Information (identified by plan id)
 ---------------------------------------------
   3 --Vector Foreign Scan on t1
         Pushdown Predicate Filter: (c2 > 10)
         Server Type: lf
         Total files left: 0
   5 --Vector Result
         Filter: (t1.c1 = t2.c1)
   8 --Vector Foreign Scan on t2
         Server Type: lf
         Total files left: 0
```

  The correlated subquery is displayed in the target list (query return list). Values need to be returned even if the condition **t1.c1=t2.c1** is not met. Therefore, use a left outer join to join **t1** and **t2** so that the SSQ can return padding values when the condition **t1.c1=t2.c1** is not met.

  NOTE

  ScalarSubQuery (SSQ) and Correlated-ScalarSubQuery (CSSQ) are described as follows:

  - SSQ: a sublink that returns only a single row and column scalar value
  - CSSQ: an SSQ containing conditions

  The preceding SQL statement can be changed into:

```
with ssq as
(
   select t2.c1, t2.c2 from t2
)
select ssq.c2, t1.c2
from t1 left join ssq on t1.c1 = ssq.c1
where t1.c2 > 10;
```

  The execution plan after the change is as follows:

 Copyright © Huawei Technologies Co., Ltd.

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 id |                    operation                    | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                 |   1665 |          |       8 | 240.66
  2 |    ->  Vector Streaming (type: GATHER)          |   1665 |          |       8 | 240.66
  3 |       ->  Vector Sonic Hash Right Join (4, 6)   |   1665 | 16MB     |       8 | 171.28
  4 |          ->  Vector Streaming(type: BROADCAST)  |   3000 | 2MB      |       8 | 98.65
  5 |             ->  Vector Foreign Scan on t2       |   1000 | 1MB      |       8 | 60.00
  6 |          ->  Vector Foreign Scan on t1          |    333 | 1MB      |       8 | 62.50

Predicate Information (identified by plan id)
-----------------------------------------------
  3 --Vector Sonic Hash Right Join (4, 6)
        Hash Cond: (t2.c1 = t1.c1)
  5 --Vector Foreign Scan on t2
        Server Type: lf
        Total files left: 0
  6 --Vector Foreign Scan on t1
        Pushdown Predicate Filter: (c2 > 10)
        Server Type: lf
        Total files left: 0
```

In the preceding example, the SSQ is pulled up to right join, preventing poor performance caused by the combination of a subplan and broadcast when the table (**T2**) in the subquery is too large.

- The subquery (with **COUNT**) in the target list cannot be pulled up.

  Example:

  ```
  select (select count(*) from t2 where t2.c1=t1.c1) cnt, t1.c1, t3.c1
  from t1,t3
  where t1.c1=t3.c1 order by cnt, t1.c1;
  ```

  The execution plan is as follows:

  ```
                                                  QUERY PLAN
  -------------------------------------------------------------------------------------------------------
   id |                      operation                       | E-rows | E-memory | E-width | E-costs
  ----+------------------------------------------------------+--------+----------+---------+---------
    1 | ->  Row Adapter                                      |   5001 |          |      16 | 575.67
    2 |    ->  Vector Streaming (type: GATHER)               |   5001 |          |      16 | 575.67
    3 |       ->  Vector Sort                                |   5001 | 16MB     |      16 | 367.29
    4 |          ->  Vector Sonic Hash Join (5,13)           |   5000 | 16MB     |      16 | 273.91
    5 |             ->  Vector Streaming(type: BROADCAST)    |   3000 | 2MB      |      12 | 188.08
    6 |                ->  Vector Sonic Hash Left Join (7, 8)|   1000 | 16MB     |      12 | 149.43
    7 |                   ->  Vector Foreign Scan on t1      |   1000 | 1MB      |       4 | 60.00
    8 |                   ->  Vector Streaming(type: BROADCAST)|    600 | 2MB      |      12 | 82.35
    9 |                      ->  Vector Sonic Hash Aggregate |    200 | 16MB     |      12 | 70.25
   10 |                         ->  Vector Streaming(type: REDISTRIBUTE)|  522 | 2MB      |      12 | 68.03
   11 |                            ->  Vector Sonic Hash Aggregate|  522 | 16MB     |      12 | 63.41
   12 |                               ->  Vector Foreign Scan on t2|  1000 | 1MB      |       4 | 60.00
   13 |             ->  Vector Foreign Scan on t3            |   1000 | 1MB      |       4 | 60.00

  Predicate Information (identified by plan id)
  -----------------------------------------------
    4 --Vector Sonic Hash Join (5,13)
          Hash Cond: (t1.c1 = t3.c1)
    6 --Vector Sonic Hash Left Join (7, 8)
          Hash Cond: (t1.c1 = t2.c1)
    7 --Vector Foreign Scan on t1
          Server Type: lf
          Total files left: 0
   12 --Vector Foreign Scan on t2
          Server Type: lf
          Total files left: 0
   13 --Vector Foreign Scan on t3
          Server Type: lf
          Total files left: 0
  ```

  The correlated subquery appears in the target list (query return list). In the scenario where **t1.c1=t2.c1** does not match, the value still needs to be output. Therefore, left outer join is used to associate t1 and t2 to ensure that the sub SSQ can return the null value when **t1.c1=t2.c1** does not match. However, the **COUNT** statement is used, and **0** needs to be output even if **t1.c1=t2.c1** does not match. Therefore, you can use **case-when NULL then 0 else count(*)** instead.

  The preceding SQL statement can be changed into:

  ```
  with ssq as
  (
     select count(*) cnt, c1 from t2 group by c1
  )
  select case when
            ssq.cnt is null then 0
            else ssq.cnt
         end cnt, t1.c1, t3.c1
  from t1 left join ssq on ssq.c1 = t1.c1,t3
  where t1.c1 = t3.c1
  order by ssq.cnt, t1.c1;
  ```

  The execution plan after the change is as follows:

 Copyright © Huawei Technologies Co., Ltd.

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 id |                                operation                                 | E-rows | E-memory | E-width | E-costs
----+--------------------------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                                          |   5001 |          |      16 |  575.67
  2 |    ->  Vector Streaming (type: GATHER)                                   |   5001 |          |      16 |  575.67
  3 |       ->  Vector Sort                                                    |   5001 |  10MB    |      16 |  367.29
  4 |          ->  Vector Sonic Hash Join (5,13)                               |   5000 |  10MB    |      16 |  273.91
  5 |             ->  Vector Streaming(type: BROADCAST)                        |   3000 |  2MB     |      12 |  188.08
  6 |                ->  Vector Sonic Hash Left Join (7, 8)                    |   1000 |  10MB    |      12 |  149.43
  7 |                   ->  Vector Foreign Scan on t1                          |   1000 |  1MB     |       4 |   60.00
  8 |                   ->  Vector Streaming(type: BROADCAST)                  |    600 |  2MB     |      12 |   82.35
  9 |                      ->  Vector Sonic Hash Aggregate                     |    200 |  10MB    |      12 |   70.25
 10 |                         ->  Vector Streaming(type: REDISTRIBUTE)         |    522 |  2MB     |      12 |   68.03
 11 |                            ->  Vector Sonic Hash Aggregate               |    522 |  10MB    |      12 |   63.41
 12 |                               ->  Vector Foreign Scan on t2              |   1000 |  1MB     |       4 |   60.00
 13 |             ->  Vector Foreign Scan on t3                                |   1000 |  1MB     |       4 |   60.00

Predicate Information (identified by plan id)
-----------------------------------------------
  4 --Vector Sonic Hash Join (5,13)
        Hash Cond: (t1.c1 = t3.c1)
  6 --Vector Sonic Hash Left Join (7, 8)
        Hash Cond: (t1.c1 = t2.c1)
  7 --Vector Foreign Scan on t1
        Server Type: lf
        Total files left: 0
 12 --Vector Foreign Scan on t2
        Server Type: lf
        Total files left: 0
 13 --Vector Foreign Scan on t3
        Server Type: lf
        Total files left: 0
```

– Pulling up nonequivalent subqueries

  Example:

```
select t1.c1, t1.c2
from t1
where t1.c1 = (select agg() from t2.c2 > t1.c2);
```

  Nonequivalent subqueries cannot be pulled up. You can perform join twice (one CorrelationKey and one rownum self-join) to rewrite the statement.

  You can rewrite the statement in either of the following ways:

  - Subquery rewriting

```
select t1.c1, t1.c2
from t1, (
   select t1.rowid, agg() aggref
   from t1,t2
   where t1.c2 > t2.c2 group by t1.rowid
) dt /* derived table */
where t1.rowid = dt.rowid AND t1.c1 = dt.aggref;
```

  - CTE rewriting

```
WITH dt as
(
   select t1.rowid, agg() aggref
   from t1,t2
   where t1.c2 > t2.c2 group by t1.rowid
)
select t1.c1, t1.c2
from t1, dt
where t1.rowid = dt.rowid AND
t1.c1 = dt.aggref;
```

NOTICE

- Currently, Aura does not have an effective way to provide globally unique row IDs for tables and intermediate result sets. Therefore, the rewriting is difficult. It is recommended that this issue is avoided at the service layer or by using **t1.xc_node_id + t1.ctid** to associate row IDs. However, the high repetition rate of **xc_node_id** leads to low association efficiency, and **xc_node_id+ctid** cannot be used as the join condition of hash join.
- If the AGG type is **COUNT(*)**, **0** is used for data padding if **CASE-WHEN** is not matched. If the type is not **COUNT(*)**, **NULL** is used.
- CTE rewriting works better using ShareScan.

More Optimization Examples

Example: Modify the SELECT statement by changing the subquery to a JOIN relationship between the primary table and the parent query or modifying the

 Copyright © Huawei Technologies Co., Ltd.

subquery to improve the query performance. Ensure that the subquery to be used is semantically correct.

```
explain (costs off)select * from t1 where t1.c1 in (select t2.c1 from t2 where t1.c2 = t2.c2);
```

```
                            QUERY PLAN
------------------------------------------------------------------
 id |                        operation
----+-------------------------------------------------------------
  1 | ->  Row Adapter
  2 |    ->  Vector Streaming (type: GATHER)
  3 |       ->  Vector Foreign Scan on t1
  4 |          ->  Row Adapter  [3, SubPlan 1]
  5 |             ->  Vector Result
  6 |                ->  Vector Materialize
  7 |                   ->  Vector Streaming(type: BROADCAST)
  8 |                      ->  Vector Foreign Scan on t2

Predicate Information (identified by plan id)
-----------------------------------------------
  3 --Vector Foreign Scan on t1
        Filter: (SubPlan 1)
        Server Type: lf
        Total files left: 0
  5 --Vector Result
        Filter: (t1.c2 = t2.c2)
  8 --Vector Foreign Scan on t2
        Server Type: lf
        Total files left: 0
```

In the preceding example, a subplan exists in the plan. To delete the subplan, modify the statement as follows:

```
explain(costs off) select * from t1 where exists (select 1 from t2 where t1.c1 = t2.c1 and t1.c2 = t2.c2);
```

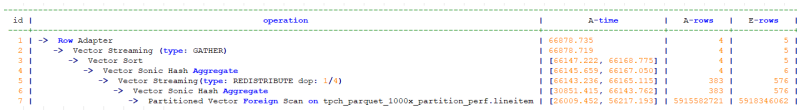
```
                        QUERY PLAN
-----------------------------------------------------------
 id |                      operation
----+------------------------------------------------------
  1 | ->  Row Adapter
  2 |    ->  Vector Streaming (type: GATHER)
  3 |       ->  Vector Sonic Hash Right Semi Join (4, 6)
  4 |          ->  Vector Streaming(type: BROADCAST)
  5 |             ->  Vector Foreign Scan on t2
  6 |          ->  Vector Foreign Scan on t1

    Predicate Information (identified by plan id)
-----------------------------------------------------
  3 --Vector Sonic Hash Right Semi Join (4, 6)
        Hash Cond: ((t2.c1 = t1.c1) AND (t2.c2 = t1.c2))
  5 --Vector Foreign Scan on t2
        Server Type: lf
        Total files left: 0
  6 --Vector Foreign Scan on t1
        Server Type: lf
        Total files left: 0
```

In this way, the subplan is replaced by the semi-join between the two tables, significantly improving the execution efficiency.

13.3.7.2 Optimizing Operators

A query statement requires passing through multiple operator steps before producing the final result. It is quite common for the overall query performance to degrade due to certain operators taking excessively long times. These operators become the bottlenecks of the entire query. To fix this, you can run the **EXPLAIN ANALYZE** or **PERFORMANCE** command to find the bottleneck.

As illustrated in the following execution details, the HashAgg operator's execution duration constitutes (66167 – 56217)/66878 = 14.8% of the total time, while the Foreign Scan operator takes up 56217/66878 = 84% of the total time. Here, the Foreign Scan operator emerges as the bottleneck for this particular query. When optimizing performance, priority should be given to enhancing this operator, such as using partitioned tables.



 Copyright © Huawei Technologies Co., Ltd.

Furthermore, when joining two tables with substantial data volumes, selecting NestLoop often leads to suboptimal performance from this operator. To significantly boost efficiency, you are advised to disable NestLoop by setting **enable_nestloop** to **off** and instead use HashJoin.

13.3.7.3 SQL Statement Rewriting Rules

Based on the database SQL execution mechanism and a large number of practices, summarize finds that: using rules of a certain SQL statement, on the basis of the so that the correct test result, which can improve the SQL execution efficiency. You can comply with these rules to greatly improve service query efficiency.

- Replacing **UNION** with **UNION ALL**

  UNION eliminates duplicate rows while merging two result sets but **UNION ALL** merges the two result sets without deduplication. Therefore, replace **UNION** with **UNION ALL** if you are sure that the two result sets do not contain duplicate rows based on the service logic.

- **Adding NOT NULL to the join column**

  If there are many **NULL** values in the join column, adding an **IS NOT NULL** filter condition allows for pre-filtering of data, thereby enhancing join efficiency.

- Use **hashagg**.

  If a plan involving groupAgg and SORT operations generated by the **GROUP BY** statement is poor in performance, you can set **work_mem** to a larger value to generate a **hashagg** plan, which does not require sorting and improves the performance.

- Replace functions with **CASE** statements

  The Aura performance greatly deteriorates if a large number of functions are called. In this case, you can modify the pushdown functions to **CASE** statements.

- **Do not use functions or expressions for partition keys.**

  Using functions or expressions for partition keys impacts partition pruning and leads to an increase in the volume of data scanned.

- Do not use **!=** or **<>** operators, **NULL**, **OR**, or implicit parameter conversion in **WHERE** clauses.

- **Split complex SQL statements.**

  You can split a SQL statement and save the execution result to a temporary table if the SQL statement is too complex to be optimized using the solutions above, including but not limited to the following scenarios:

  - The same subquery is involved in multiple SQL statements of a task and the subquery contains large amounts of data.
  - Functions such as **substr** and **to_number** cause incorrect measures for subqueries containing large amounts of data.
  - **BROADCAST** subqueries are performed on large tables in multi-DN environment.

 Copyright © Huawei Technologies Co., Ltd.

13.3.8 Configuring Optimizer Parameters

This section introduces key CN parameters that affect Aura performance tuning. For details about the parameter configuration method, see **14.1 Viewing and Setting GUC Parameters**.

Table 13-13 CN parameters

Parameter/ Reference Value	Description
enable_nestloop=on	Controls whether the query optimizer uses the nested loop join type. • When set to **on**, the optimizer prefers to use nested loop join. • When set to **off**, the optimizer prefers other methods when they are available. **NOTE** To temporarily change the value of this parameter in the current database connection (that is, the current session), run the following SQL statement: `SET enable_nestloop to off;` This parameter is set to **off** by default. However, in actual tuning, you should decide whether to disable it based on the specific situation. Generally, nested loop join has the poorest performance among the three **JOIN** methods (nested loop join, merge join, and hash join). You are advised to set this parameter to **off**.
enable_hashjoin=on	Specifies whether to enable the optimizer's use of Hash-join plan types.
rewrite_rule	Specifies whether the optimizer enables a specific rewriting rule.

13.3.9 Hint-based Tuning

13.3.9.1 Plan Hint Optimization

In plan hints, you can specify a join order, join, stream, and scan operations, the number of rows in a result, and redistribution skew information to tune an execution plan, improving query performance.

Function

Plan hints can be specified using the keywords such as **SELECT**, **INSERT**, **UPDATE**, **MERGE**, and **DELETE**, in the following format:

```
/*+ <plan hint> */
```

 Copyright © Huawei Technologies Co., Ltd.

You can specify multiple hints for a query plan and separate them by spaces. A hint specified for a query plan does not apply to its subquery plans. To specify a hint for a subquery, add the hint following the keyword of this subquery.

For example:

```
select /*+ <plan_hint1> <plan_hint2> */ * from t1, (select /*+ <plan_hint3> */ * from t2) where 1=1;
```

In the preceding command, <*plan_hint1*> and <*plan_hint2*> are the hints of a query, and <*plan_hint3*> is the hint of its subquery.

NOTE

- If a hint is specified in the view definition (**CREATE VIEW**), the hint will be used each time the view is applied.
- When the random plan feature is enabled (the parameter **plan_mode_seed** is not set to **0**), the plan hints specified in the query will not be used.

Supported Hints

Currently, the following hints are supported:

- Join order hints (**leading**)
- Join operation hints, excluding the **semi join**, **anti join**, and **unique plan** hints
- Rows hints
- Stream operation hints
- Scan operation hints, supporting only **tablescan**, **indexscan**, and **indexonlyscan**
- Sublink name hints
- Skew hints, supporting only the skew in the redistribution involving Join or HashAgg
- Hint used for **Agg** distribution columns
- Hint that disables subquery pull-up.
- Configuration parameter hints. For details about supported parameters, see **13.3.9.9 Hints for Configuration Parameters**.

Precautions

- **Sort**, **Setop**, and **Subplan** hints are not supported.
- Hints cannot be used for the target table of the **INSERT** statement.

Examples

The following is the original plan and is used for comparing with the optimized ones:

```
explain
select i_product_name product_name
,i_item_sk item_sk
,s_store_name store_name
,s_zip store_zip
,ad2.ca_street_number c_street_number
,ad2.ca_street_name c_street_name
,ad2.ca_city c_city
,ad2.ca_zip c_zip
```

 Copyright © Huawei Technologies Co., Ltd.

```
,count(*) cnt
,sum(ss_wholesale_cost) s1
,sum(ss_list_price) s2
,sum(ss_coupon_amt) s3
FROM   store_sales
,store_returns
,store
,customer
,promotion
,customer_address ad2
,item
WHERE  ss_store_sk = s_store_sk AND
ss_customer_sk = c_customer_sk AND
ss_item_sk = i_item_sk and
ss_item_sk = sr_item_sk and
ss_ticket_number = sr_ticket_number and
c_current_addr_sk = ad2.ca_address_sk and
ss_promo_sk = p_promo_sk and
i_color in ('maroon','burnished','dim','steel','navajo','chocolate') and
i_current_price between 35 and 35 + 10 and
i_current_price between 35 + 1 and 35 + 15
group by i_product_name
,i_item_sk
,s_store_name
,s_zip
,ad2.ca_street_number
,ad2.ca_street_name
,ad2.ca_city
,ad2.ca_zip
;
```

```
 id |                                operation                                | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                                         |   4425 |          |     790 | 1742.92
  2 |   ->  Vector Sonic Hash Aggregate                                       |   4425 |          |     790 | 1742.92
  3 |     ->  Vector Streaming (type: GATHER)                                 |   4425 |          |     790 | 1742.92
  4 |       ->  Vector Sonic Hash Aggregate                                   |   4425 | 16MB     |     790 | 604.98
  5 |         ->  Vector Sonic Hash Join (6,8)                                |   4425 | 16MB     |     758 | 545.98
  6 |           ->  Vector Streaming(type: BROADCAST)                         |   3000 | 2MB      |       4 | 98.65
  7 |             ->  Vector Foreign Scan on promotion                        |   1000 | 1MB      |       4 | 60.00
  8 |           ->  Vector Sonic Hash Join (9,23)                             |    885 | 16MB     |     762 | 425.15
  9 |             ->  Vector Streaming(type: BROADCAST)                       |    531 | 2MB      |     402 | 356.76
 10 |               ->  Vector Sonic Hash Join (11,12)                        |    177 | 16MB     |     402 | 329.36
 11 |                 ->  Vector Foreign Scan on store                        |   1000 | 1MB      |     166 | 60.00
 12 |                 ->  Vector Streaming(type: BROADCAST)                   |    105 | 2MB      |     244 | 267.44
 13 |                   ->  Vector Sonic Hash Join (14,15)                    |     35 | 16MB     |     244 | 264.66
 14 |                     ->  Vector Foreign Scan on customer                 |   1000 | 1MB      |       8 | 60.00
 15 |                     ->  Vector Streaming(type: BROADCAST)               |     21 | 2MB      |     244 | 203.60
 16 |                       ->  Vector Sonic Hash Join (17,18)                |      7 | 16MB     |     244 | 203.14
 17 |                         ->  Vector Foreign Scan on store_returns        |   1000 | 1MB      |       8 | 60.00
 18 |                         ->  Vector Streaming(type: BROADCAST)           |     60 | 2MB      |     252 | 141.01
 19 |                           ->  Vector Sonic Hash Join (20,21)            |     20 | 16MB     |     252 | 138.69
 20 |                             ->  Vector Foreign Scan on store_sales      |   1000 | 1MB      |      44 | 60.00
 21 |                             ->  Vector Streaming(type: BROADCAST)       |     12 | 2MB      |     208 | 77.73
 22 |                               ->  Vector Foreign Scan on item           |      4 | 1MB      |     208 | 77.50
 23 |             ->  Vector Foreign Scan on customer_address ad2             |   1000 | 1MB      |     368 | 60.00
```

13.3.9.2 Join Order Hints

Function

Theses hints specify the join order and outer/inner tables.

Syntax

- Single-layer parentheses (), specifying only the join order. The order of internal and foreign tables is not specified.
  ```
  leading(join_table_list)
  leading(@block_name join_table_list)
  ```
- Double parentheses (()), specifying the join order and outer/inner tables. The outer/inner tables are specified by the outermost parentheses.
  ```
  leading((join_table_list))
  leading(@block_name (join_table_list))
  ```
- Single-layer square brackets [], specifying the join order of [] and the sequence of internal and foreign tables.
  ```
  leading[join_table_list]
  leading[@block_name join_table_list]
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Combination of single-layer parentheses () and single-layer square brackets [], specifying the join order and the sequence of internal and foreign tables at any layer. The parentheses () specify only the join order, but not the sequence of internal and foreign tables. The square brackets [] specify both the join order and the sequence of internal and foreign tables.

```
leading(join_table_list1 [join_table_list2])
leading[join_table_list1 [join_table_list2]]
leading[join_table_list1 (join_table_list2)]
leading(@block_name join_table_list1 [join_table_list2])
leading[@block_name join_table_list1 [join_table_list2]]
leading[@block_name join_table_list1 (join_table_list2)]
```

NOTICE

Single-layer square brackets [] can be used together with single-layer parentheses () to specify the sequence of internal and foreign tables of any layer. Single-layer [] and double-layer () cannot be used together.

Parameter Description

- **join_table_list**

  Specifies the tables to be joined. The values can be table names or table aliases. If a subquery is pulled up, the value can also be the subquery alias. Separate the values with spaces. You can add parentheses to specify the join priorities of tables.

  To prevent semantic errors, tables in the list must meet the following requirements:

  - The tables must exist in the query or its subquery to be pulled up.
  - The table names must be unique in the query or subquery to be pulled up. If they are not, their aliases must be unique.
  - A table appears only once in the list.
  - An alias (if any) is used to represent a table.

  NOTE

  - The syntax format of tables is as follows:

    [schema.]table[@block_name]

    The table name can include the schema or the **block_name** prior to the lifting of the subquery statement block. If the subquery statement block is lifted during optimization and rewriting by the optimizer, this **block_name** will differ from the **block_name** in leading.
  - If a table has an alias, the alias is preferentially used to represent the table.

- **block_name**

  Specifies the block name of the statement block. It indicates that the hint takes effect in the subquery statement block corresponding to the block name.

  - By default, a block name is generated for a statement.
  - CN lightweight statements do not generate block names.
  - Block names can be generated for the CREATE TABLE AS SELECT, SELECT INTO, SELECT, INSERT, UPDATE, DELETE, and MERGE statements.

 Copyright © Huawei Technologies Co., Ltd.

- The naming rules for block names are as follows:
  - A block name is automatically generated for the SELECT, INSERT, UPDATE, DELETE, and MERGE statements. The naming format of a block name for these statements is seln, insn, updn, deln, and mer$n, respectively, where n starts from 1. The number of statements of different types is not accumulated, but the number of statements of the same type is accumulated.

    Example:

    ```
    INSERT INTO t SELECT * FROM t1 WHERE a1 IN (select * from t2);
    --------sel$2-------
    ----------------------sel$1----------------------
    -------------------------------ins$1--------------------------
    ```

  - Recursively assigns a block name to each statement block before the optimizer is used.

    First, assign block names to the existing statements block based on the statement type, then traverse the statement blocks in the following sequence, and assign block name to the statement blocks in the statement blocks:

    1) Traverse the target column.
    2) Traverse the target column in the source table of the MERGE statement.
    3) Traverse actions (update or insert) in the MERGE statement.
    4) Traverse the returning clause.
    5) Traverse the Join and Where conditions in From. (The join condition takes precedence over the Where condition.)
    6) For a set operation, traverse each branch of the set (UNION, INTERSECT, and EXCEPT).
    7) Traverse the HAVING clause.
    8) Traverse the LIMIT OFFSET clause.
    9) Traverse the LIMIT COUNT clause.
    10) Traverse CTE
    11) Traverse the table after From.
    12) Traverse the UPSERT clause.

  - In the rewriting phase of the optimizer, rewriting optimization is performed due to FUL LJOIN, cte inline, materialized view rewriting, INLIST2JOIN, OR conversion, multi count(distinct), Magic Set, lazyagg, and subquery/sublink promotion, a new subquery is constructed. In this case, the recursive processing during block name assignment is also applied to the newly constructed subquery. The number of block names is accumulated.
  - In the optimizer rewriting phase, when a subquery is promoted, the table in the inner subquery is promoted to the outer query, and the inner subquery is eliminated. In this case, the promoted table may have the same name as the table in the outer queries. Therefore, the block name to which the promoted table belongs is recorded in the

 Copyright © Huawei Technologies Co., Ltd.

table to distinguish two tables with the same name but are from different query blocks.

For example:

- **leading(t1 t2 t3 t4 t5)**: **t1**, **t2**, **t3**, **t4**, and **t5** are joined. The join order and outer/inner tables are not specified.
- **leading(t1 t2 t3 t4 t5)**: **t1**, **t2**, **t3**, **t4**, and **t5** are joined in sequence. The table on the right is used as the inner table in each join.
- **leading(t1 (t2 t3 t4) t5)**: First, **t2**, **t3**, and **t4** are joined and the outer/inner tables are not specified. Then, the result is joined with **t1** and **t5**, and the outer/inner tables are not specified.
- **leading(t1 (t2 t3 t4) t5)**: First, **t2**, **t3**, and **t4** are joined and the outer/inner tables are not specified. Then, the result is joined with **t1**, and **(t2 t3 t4)** is used as the inner table. Finally, the result is joined with **t5**, and **t5** is used as the inner table.
- **leading((t1 (t2 t3) t4 t5)) leading((t3 t2))**: First, **t2** and **t3** are joined and **t2** is used as the inner table. Then, the result is joined with **t1**, and **(t2 t3)** is used as the inner table. Finally, the result is joined with **t4** and then **t5**, and the table on the right in each join is used as the inner table.
- leading[t1 [t2 t3]] is equivalent to leading((t1 (t2 t3))) leading((t2 t3)).
- leading(t1 [t2 t3]) is equivalent to leading(t1 t2 t3) leading((t2 t3)).
- leading[@sel$1 t1@sel$1 [t2@sel$2 t3@sel$2]] indicates that t2 and t3 are located in the subquery. After the subquery is promoted, t2 and t3 are joined, and then the join table is joined to t1. Where t2 is a foreign table, t3 is an internal table, t1 is a foreign table. The join table of t2 and t3 is an internal table.

Examples

Hint the query plan in **Examples** as follows:

```
explain
select /*+ leading((((((store_sales store) promotion) item) customer) ad2) store_returns) leading((store
store_sales))*/ i_product_name product_name ...
```

First, **store_sales** and **store** are joined and **store_sales** is the inner table. Then, The result is joined with **promotion**, **item**, **customer**, **ad2**, and **store_returns** in sequence.

The optimized plan is as follows:

```
 id |                                operation                                | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                                         |   4425 |          |     790 | 2244.04
  2 |    ->  Vector Sonic Hash Aggregate                                      |   4425 |          |     790 | 2244.04
  3 |       ->  Vector Streaming (type: GATHER)                               |   4425 |          |     790 | 2244.04
  4 |          ->  Vector Sonic Hash Aggregate                                |   4425 | 16MB     |     790 | 1106.10
  5 |             ->  Vector Sonic Hash Join (6,22)                           |   4425 | 16MB     |     758 | 1024.99
  6 |                ->  Vector Sonic Hash Join (7,9)                         |  12500 | 16MB     |     766 | 863.42
  7 |                   ->  Vector Streaming(type: BROADCAST)                 |   3000 | 2MB      |     368 | 175.96
  8 |                      ->  Vector Foreign Scan on customer_address ad2    |   1000 | 1MB      |     368 | 60.00
  9 |                   ->  Vector Sonic Hash Join (10,12)                    |   2500 | 16MB     |     406 | 618.30
 10 |                      ->  Vector Streaming(type: BROADCAST)              |   3000 | 2MB      |       8 | 98.65
 11 |                         ->  Vector Foreign Scan on customer             |   1000 | 1MB      |       8 | 60.00
 12 |                      ->  Vector Sonic Hash Join (13,20)                 |    500 | 16MB     |     406 | 504.44
 13 |                         ->  Vector Sonic Hash Join (14,18)              |  25000 | 16MB     |     198 | 403.97
 14 |                            ->  Vector Sonic Hash Join (15,16)           |   5000 | 16MB     |     202 | 194.90
 15 |                               ->  Vector Foreign Scan on store          |   1000 | 1MB      |     166 | 60.00
 16 |                               ->  Vector Streaming(type: BROADCAST)     |   3000 | 2MB      |      44 | 98.65
 17 |                                  ->  Vector Foreign Scan on store_sales |   1000 | 1MB      |      44 | 60.00
 18 |                            ->  Vector Streaming(type: BROADCAST)        |   3000 | 2MB      |       4 | 98.65
 19 |                               ->  Vector Foreign Scan on promotion      |   1000 | 1MB      |       4 | 60.00
 20 |                         ->  Vector Streaming(type: BROADCAST)           |     12 | 2MB      |     208 | 77.73
 21 |                            ->  Vector Foreign Scan on item              |      4 | 1MB      |     208 | 77.50
 22 |                   ->  Vector Streaming(type: BROADCAST)                 |   3000 | 2MB      |       8 | 98.65
 23 |                      ->  Vector Foreign Scan on store_returns           |   1000 | 1MB      |       8 | 60.00
```

For details about the warning at the top of the plan, see **13.3.9.11 Hint Errors, Conflicts, and Other Warnings**.

 Copyright © Huawei Technologies Co., Ltd.

13.3.9.3 Join Operation Hints

Function

Specifies the join method. It can be nested loop join, hash join, or merge join.

Syntax

```
[no] nestloop|hashjoin|mergejoin([@block_name] table_list)
```

Parameter Description

- **no** indicates that the specified hint will not be used for a join.
- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- *table_list* is a string representing the hint table collection. The tables within this string match those in **join_table_list**, with the exception that parentheses for specifying join priorities are not allowed.

For example:

no nestloop(t1 t2 t3): **nestloop** is not used for joining **t1**, **t2**, and **t3**. The three tables may be joined in either of the two ways: Join **t2** and **t3**, and then **t1**; join **t1** and **t2**, and then **t3**. This hint takes effect only for the last join. If necessary, you can hint other joins. For example, you can add **no nestloop(t2 t3)** to join **t2** and **t3** first and to forbid the use of **nestloop**.

Examples

Hint the query plan in **Examples** as follows:

```
explain
select /*+ nestloop(store_sales store_returns item) */ i_product_name product_name ...
```

nestloop is used for the last join between **store_sales**, **store_returns**, and **item**. The optimized plan is as follows:

```
 id |                                operation                                | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------------------------------+--------+----------+---------+---------
  1 |  ->  Row Adapter                                                        |   4425 |          |     790 | 1800.04
  2 |    ->  Vector Sonic Hash Aggregate                                      |   4425 |          |     790 | 1800.04
  3 |      ->  Vector Streaming (type: GATHER)                                |   4425 |          |     790 | 1800.04
  4 |        ->  Vector Sonic Hash Aggregate                                  |   4425 |  16MB    |     790 | 662.10
  5 |          ->  Vector Sonic Hash Join (6,8)                               |   4425 |  16MB    |     758 | 603.10
  6 |            ->  Vector Streaming(type: BROADCAST)                        |   3000 |  2MB     |       4 | 98.65
  7 |              ->  Vector Foreign Scan on promotion                       |   1000 |  1MB     |       4 | 60.00
  8 |            ->  Vector Sonic Hash Join (9,24)                            |    885 |  16MB    |     762 | 482.26
  9 |              ->  Vector Streaming(type: BROADCAST)                      |    531 |  2MB     |     402 | 413.87
 10 |                ->  Vector Sonic Hash Join (11,23)                       |    177 |  16MB    |     402 | 386.48
 11 |                  ->  Vector Streaming(type: BROADCAST)                  |    105 |  2MB     |     244 | 320.82
 12 |                    ->  Vector Sonic Hash Join (13,14)                   |     35 |  16MB    |     244 | 318.03
 13 |                      ->  Vector Foreign Scan on customer                |   1000 |  1MB     |       8 | 60.00
 14 |                      ->  Vector Streaming(type: BROADCAST)              |     21 |  2MB     |     244 | 256.98
 15 |                        ->  Vector Nest Loop (16,20)                     |      7 |  1MB     |     244 | 256.05
 16 |                          ->  Vector Sonic Hash Join (17,19)             |    354 |  16MB    |      44 | 171.23
 17 |                            ->  Vector Streaming(type: BROADCAST)        |   3000 |  2MB     |      44 | 98.65
 18 |                              ->  Vector Foreign Scan on store_sales     |   1000 |  1MB     |      44 | 60.00
 19 |                            ->  Vector Foreign Scan on store_returns     |   1000 |  1MB     |       8 | 60.00
 20 |                          ->  Vector Materialize                         |     12 |  16MB    |     208 | 77.75
 21 |                            ->  Vector Streaming(type: BROADCAST)        |     12 |  2MB     |     208 | 77.73
 22 |                              ->  Vector Foreign Scan on item            |      4 |  1MB     |     208 | 77.30
 23 |                    ->  Vector Foreign Scan on store                     |   1000 |  1MB     |     166 | 60.00
 24 |                ->  Vector Foreign Scan on customer_address ad2          |   1000 |  1MB     |     368 | 60.00
```

13.3.9.4 Rows Hints

Function

These hints specify the number of rows in an intermediate result set. Both absolute values and relative values are supported.

Syntax

```
rows([@block_name] table_list #|+|-|* const)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- **#**,**+**,**-**, and ***** are operators used for hinting the estimation. **#** indicates that the original estimation is used without any calculation. **+**,**-**, and ***** indicate that the original estimation is calculated using these operators. The minimum calculation result is 1. *table_list* indicates the tables to be joined, which is the same as *table_list* in **13.3.9.3 Join Operation Hints**.
- *const* can be any non-negative number and supports scientific notation.

For example:

rows(t1 #5): The result set of **t1** is five rows.

rows(t1 t2 t3 *1000): Multiply the result set of joined **t1**, **t2**, and **t3** by 1000.

Suggestion

- The hint using ***** for two tables is recommended, because this hint will take effect for a join as long as the two tables appear on both sides of this join. For example, if the hint is **rows(t1 t2 * 3)**, the join result of **(t1 t3 t4)** and **(t2 t5 t6)** will be multiplied by 3 because **t1** and **t2** appear on both sides of the join.
- **rows** hints can be specified for the result sets of a single table, multiple tables, function tables, and subquery scan tables.

Examples

Hint the query plan in **Examples** as follows:

```
explain
select /*+ leading(store_sales store_returns) rows(store_sales store_returns *50) */ i_product_name
product_name ...
```

Multiply the result set of joined **store_sales** and **store_returns** by 50. The optimized plan is as follows:

```
 id |                           operation                           | E-rows |  E-memory   | E-width | E-costs
----+---------------------------------------------------------------+--------+-------------+---------+---------
  1 |  ->  Row Adapter                                              | 160000 |             |     790 | 7304.28
  2 |    ->  Vector Sonic Hash Aggregate                            | 160000 |             |     790 | 7304.28
  3 |      ->  Vector Streaming (type: GATHER)                      | 206007 |             |     790 | 7304.28
  4 |        ->  Vector Sonic Hash Aggregate                        | 206007 | 38MB(237MB) |     790 | 4732.92
  5 |          ->  Vector Sonic Hash Join (6,22)                    | 221250 | 16MB        |     758 | 1833.73
  6 |            ->  Vector Sonic Hash Join (7,20)                  |  44250 | 16MB        |     762 | 856.02
  7 |              ->  Vector Sonic Hash Join (8,12)                |   8850 | 16MB        |     604 | 532.90
  8 |                ->  Vector Sonic Hash Join (9,11)              |   5000 | 16MB        |     368 | 184.48
  9 |                  ->  Vector Streaming(type: BROADCAST)        |   3000 | 2MB         |       8 | 98.65
 10 |                    ->  Vector Foreign Scan on customer        |   1000 | 1MB         |       8 | 60.00
 11 |                  ->  Vector Foreign Scan on customer_address ad2 |   1000 | 1MB         |     368 | 60.00
 12 |                ->  Vector Streaming(type: BROADCAST)          |   1062 | 2MB         |     244 | 306.16
 13 |                  ->  Vector Sonic Hash Join (14,18)           |    354 | 16MB        |     244 | 265.07
 14 |                    -> Vector Sonic Hash Join (15,17)          |  17700 | 16MB        |      44 | 171.23
 15 |                      ->  Vector Streaming(type: BROADCAST)    |   3000 | 2MB         |      44 | 98.65
 16 |                        ->  Vector Foreign Scan on store_sales |   1000 | 1MB         |      44 | 60.00
 17 |                      ->  Vector Foreign Scan on store_returns |   1000 | 1MB         |       8 | 60.00
 18 |                    ->  Vector Streaming(type: BROADCAST)      |     12 | 2MB         |     208 | 77.73
 19 |                      ->  Vector Foreign Scan on item          |      4 | 1MB         |     208 | 77.50
 20 |                ->  Vector Streaming(type: BROADCAST)          |   3000 | 2MB         |     166 | 137.30
 21 |                  ->  Vector Foreign Scan on store             |   1000 | 1MB         |     166 | 60.00
 22 |              ->  Vector Streaming(type: BROADCAST)            |   3000 | 2MB         |       4 | 98.65
 23 |                ->  Vector Foreign Scan on promotion           |   1000 | 1MB         |       4 | 60.00
```

The estimated number of rows of the operator in row 11 is revised to 17,700, with the original estimate being 354 rows.

13.3.9.5 Stream Operation Hints

Function

Specifies the stream method, which can be broadcast, redistribute, or specifying the distribution key for **Agg** redistribution.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
[no] broadcast | redistribute([@block_name] table_list) | redistribute ([@block_name] (*) (columns))
```

Parameter Description

- **no** indicates that the hinted stream method is not used. When the hint is specified for the distribution columns in the **Agg** redistribution, **no** is invalid.
- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- **table_list** indicates the tables to be joined. For details, see **Parameter Description**.
- When hints are specified for distribution columns, the asterisk (*) is fixed and the table name cannot be specified.
- **columns** specifies one or more columns in the **GROUP BY** clause. When there are no **GROUP BY** clauses, it can specify the columns in the **DISTINCT** clause.

  NOTE

  - The specified distribution column must be specified using the column sequence number or column name in **group by** or **distinct**. The columns in **count(distinct)** can only be specified using column names.
  - For a multi-layer query, you can specify the distribution column hint at each layer. The hint takes effect only at the corresponding layer.
  - The column specified in **count(distinct)** takes effect only for two-level hashagg plans. Otherwise, the specified distribution column is invalid.
  - If the optimizer finds that redistribution is not required after estimation, the specified distribution column is invalid.

Tips

- Generally, the optimizer selects a group of non-skew distribution keys for data redistribution based on statistics. If the default distribution keys have data skew, you can manually specify the distribution columns to avoid data skew.
- When selecting a distribution key, select a group of columns with high distinct values as the distribution key based on data distribution features. In this way, data can be evenly distributed to each DN after redistribution.
- After writing hints, you can run **explain verbose** to print the execution plan and check whether the specified distribution key is valid. If the specified distribution key is invalid, a warning is displayed.

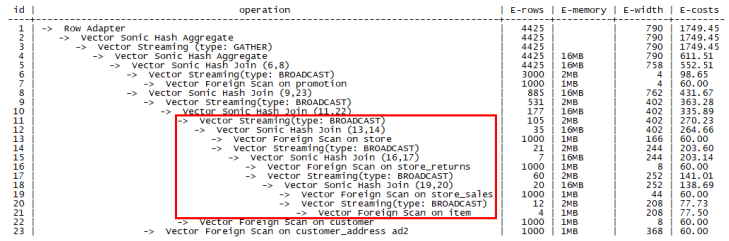
Examples

- Hint the query plan in **Examples** as follows:

  ```
  explain
  select /*+ no redistribute(store_sales store_returns item store) leading(((store_sales store_returns item store) customer)) */ i_product_name product_name ...
  ```

  In the original plan, the join result of **store_sales**, **store_returns**, **item**, and **store** is redistributed before it is joined with **customer**. After the hinting, the redistribution is disabled and the join order is retained. The optimized plan is as follows:

 Copyright © Huawei Technologies Co., Ltd.



```
 id |                                operation                                | E-rows | E-memory | E-width | E-costs
----+-------------------------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                                         |   4425 |          |     790 | 1749.45
  2 |   ->  Vector Sonic Hash Aggregate                                       |   4425 |          |     790 | 1749.45
  3 |     ->  Vector Streaming (type: GATHER)                                 |   4425 |          |     790 | 1749.45
  4 |       ->  Vector Sonic Hash Aggregate                                   |   4425 | 16MB     |     790 | 611.51
  5 |         ->  Vector Sonic Hash Join (6,8)                                |   4425 | 16MB     |     758 | 552.51
  6 |           ->  Vector Streaming(type: BROADCAST)                         |   3000 | 2MB      |       4 | 98.65
  7 |             ->  Vector Foreign Scan on promotion                        |   1000 | 1MB      |       4 | 60.00
  8 |           ->  Vector Sonic Hash Join (9,23)                             |    885 | 16MB     |     762 | 431.67
  9 |             ->  Vector Streaming(type: BROADCAST)                       |    531 | 2MB      |     402 | 363.28
 10 |               ->  Vector Sonic Hash Join (11,22)                        |    177 | 16MB     |     402 | 335.89
 11 |                 ->  Vector Streaming(type: BROADCAST)                   |    105 | 2MB      |     402 | 270.23
 12 |                   ->  Vector Sonic Hash Join (13,14)                    |     35 | 16MB     |     402 | 264.66
 13 |                     ->  Vector Foreign Scan on store                    |   1000 | 1MB      |     166 | 60.00
 14 |                     ->  Vector Streaming(type: BROADCAST)               |     21 | 2MB      |     244 | 203.60
 15 |                       ->  Vector Sonic Hash Join (16,17)                |      7 | 16MB     |     244 | 203.14
 16 |                         ->  Vector Foreign Scan on store_returns        |   1000 | 1MB      |       8 | 60.00
 17 |                         ->  Vector Streaming(type: BROADCAST)           |     60 | 2MB      |     252 | 141.01
 18 |                           ->  Vector Sonic Hash Join (19,20)            |     20 | 16MB     |     252 | 138.69
 19 |                             ->  Vector Foreign Scan on store_sales      |   1000 | 1MB      |      44 | 60.00
 20 |                             ->  Vector Streaming(type: BROADCAST)       |     12 | 2MB      |     208 | 77.73
 21 |                               ->  Vector Foreign Scan on item           |      4 | 1MB      |     208 | 77.50
 22 |                 ->  Vector Foreign Scan on customer                     |   1000 | 1MB      |       8 | 60.00
 23 |               ->  Vector Foreign Scan on customer_address ad2           |   1000 | 1MB      |     368 | 60.00
```

- Specifies the distribution columns for Agg redistribution.

  ```
  explain (verbose on, costs off, nodes off)
  select /*+ redistribute ((*) (2 3)) */ a1, b1, c1, count(c1)  from t1 group by a1, b1, c1 having count(c1) > 10 and sum(d1) > 100
  ```

  In the following example, the last two columns of the specified **GROUP BY** columns are used as distribution keys.

  ```
                            QUERY PLAN
  ------------------------------------------------------------------
    id |                 operation
   ----+-------------------------------------------
     1 | ->  Streaming (type: GATHER)
     2 |     ->  HashAggregate
     3 |       ->  Streaming(type: REDISTRIBUTE)
     4 |         ->  Seq Scan on public.t1

           Predicate Information (identified by plan id)
   ------------------------------------------------------------
     2 --HashAggregate
           Filter: ((count(t1.c1) > 10) AND (sum(t1.d1) > 100))

   Targetlist Information (identified by plan id)
   ----------------------------------------------
     1 --Streaming (type: GATHER)
           Output: a1, b1, c1, (count(c1))
     2 --HashAggregate
           Output: a1, b1, c1, count(c1)
           Group By Key: t1.a1, t1.b1, t1.c1
     3 --Streaming(type: REDISTRIBUTE)
           Output: a1, b1, c1, d1
           Distribute Key: b1, c1
     4 --Seq Scan on public.t1
           Output: a1, b1, c1, d1

      ====== Query Summary =====
   ---------------------------------
   System available mem: 24862720KB
   Query Max mem: 24862720KB
   Query estimated mem: 3138KB
  (30 rows)
  ```

- If the statement does not contain the **GROUP BY** clause, specify the distinct column as the distribution columns.

  ```
  explain (verbose on, costs off, nodes off)
  select /*+ redistribute ((*) (3 1)) */ distinct a1, b1, c1 from t1;
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
                    QUERY PLAN
-------------------------------------------------
  id |                operation
 ----+-----------------------------------------
   1 | ->  Streaming (type: GATHER)
   2 |     ->  HashAggregate
   3 |        ->  Streaming(type: REDISTRIBUTE)
   4 |           ->  Seq Scan on public.t1

 Targetlist Information (identified by plan id)
 -------------------------------------------------
   1 --Streaming (type: GATHER)
         Output: a1, b1, c1
   2 --HashAggregate
         Output: a1, b1, c1
         Group By Key: t1.a1, t1.b1, t1.c1
   3 --Streaming(type: REDISTRIBUTE)
         Output: a1, b1, c1
         Distribute Key: c1, a1
   4 --Seq Scan on public.t1
         Output: a1, b1, c1

    ====== Query Summary =====
 ---------------------------------
 System available mem: 24862720KB
 Query Max mem: 24862720KB
 Query estimated mem: 3136KB
(25 rows)
```

13.3.9.6 Hints for Sublink Block Names

Function

These hints specify the name of a sublink block.

Syntax

```
blockname ([@block_name] table)
```

Precautions

- This block name hint is used by an outer query only when a sublink is pulled up. Currently, only the **Agg** equivalent join, **IN**, and **EXISTS** sublinks can be pulled up. This hint is usually used together with the hints described in the previous sections.
- The subquery after the **FROM** keyword is hinted by using the subquery alias. In this case, **block_name hint** becomes invalid.
- If a sublink contains multiple tables, the tables will be joined with the outer-query tables in a random sequence after the sublink is pulled up. In this case, **blockname** also becomes invalid.

Parameter Description

- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- *table* indicates the name you have specified for a sublink block.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- The syntax format of tables is as follows:

  [schema.]table[@block_name]

  The table name can include the schema or the **block_name** prior to the lifting of the subquery statement block. If the subquery statement block is lifted during optimization and rewriting by the optimizer, this **block_name** will differ from the **block_name** in leading.
- If a table has an alias, the alias is preferentially used to represent the table.

Examples

```
explain select /*+nestloop(store_sales tt) */ * from store_sales where ss_item_sk in (select /*
+blockname(tt)*/ i_item_sk from item group by 1);
```

tt indicates the sublink block name. After being pulled up, the sublink is joined with the outer-query table **store_sales** by using **nestloop**. The optimized plan is as follows:

```
 id |                       operation                        | E-rows | E-memory | E-width | E-costs
----+--------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                        |    499 |          |     140 | 264.13
  2 |    ->  Vector Streaming (type: GATHER)                 |    499 |          |     140 | 264.13
  3 |       ->  Vector Nest Loop (4,7)                       |    500 | 1MB      |     140 | 223.32
  4 |          ->  Vector Sonic Hash Aggregate               |    201 | 16MB     |       4 | 38.33
  5 |             ->  Vector Streaming(type: REDISTRIBUTE)   |   1000 | 2MB      |       4 | 37.39
  6 |                ->  Vector Foreign Scan on item         |   1000 | 1MB      |       4 | 30.00
  7 |          ->  Vector Materialize                        |   1000 | 16MB     |     140 | 64.07
  8 |             ->  Vector Streaming(type: REDISTRIBUTE)   |   1000 | 2MB      |     140 | 62.87
  9 |                ->  Vector Foreign Scan on store_sales  |   1000 | 1MB      |     140 | 30.00
```

13.3.9.7 Hint That Disables Subquery Pull-up

Function

To optimize query logic, the optimizer usually pulls up subqueries for execution. However, sometimes the pulled up subqueries do not run much faster than others, and may even be slower due to enlarged search scope. In this case, you can specify the **no merge** hint to disable pull-up. This hint is not recommended in most cases.

Syntax

```
no merge[@block_name]
no merge ([@block_name1] subquery_name[@block_name2])
```

Parameter Description

- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- **subquery_name** indicates the name of a subquery. It can also be a view or CTE name. The specified subquery will not be unnested during logic optimization. If **subquery_name** is not specified, the current query will not be unnested.

Examples

Create tables **t1**, **t2**, and **t3**.

 Copyright © Huawei Technologies Co., Ltd.

```
create table t1(a1 int,b1 int,c1 int,d1 int) store as orc;
create table t2(a2 int,b2 int,c2 int,d2 int) store as orc;
create table t3(a3 int,b3 int,c3 int,d3 int) store as orc;
```

The original statement is as follows:

```
explain select * from t3, (select a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
```

```
 id |                     operation                      | E-rows | E-memory | E-width | E-costs
----+----------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                    |  25001 |          |      32 | 936.75
  2 |    ->  Vector Streaming (type: GATHER)             |  25001 |          |      32 | 936.75
  3 |       ->  Vector Sonic Hash Join (4,9)             |  25000 | 16MB     |      32 | 244.33
  4 |          ->  Vector Sonic Hash Join (5,7)          |   5000 | 16MB     |      28 | 108.44
  5 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      16 | 48.10
  6 |                ->  Vector Foreign Scan on t3       |   1000 | 1MB      |      16 | 30.00
  7 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      12 | 48.10
  8 |                ->  Vector Foreign Scan on t2       |   1000 | 1MB      |      12 | 30.00
  9 |          ->  Vector Streaming(type: BROADCAST)     |   3000 | 2MB      |       8 | 71.21
 10 |             ->  Vector Foreign Scan on t1          |   1000 | 1MB      |       8 | 30.00
```

In this query, you can use the following methods to disable the pull-up of subquery **s1**:

- Method 1:
  ```
  explain select /*+ no merge(s1) */ * from t3, (select a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
  ```
- Method 2:
  ```
  explain select * from t3, (select /*+ no merge */ a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
  ```

The following figure shows the result after the no merge hint is used.

```
 id |                     operation                      | E-rows | E-memory | E-width | E-costs
----+----------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                    |   5001 |          |      32 | 551.37
  2 |    ->  Vector Streaming (type: GATHER)             |   5001 |          |      32 | 551.37
  3 |       ->  Vector Sonic Hash Join (4,6)             |   5000 | 16MB     |      32 | 232.56
  4 |          ->  Vector Streaming(type: BROADCAST)     |   6000 | 2MB      |      16 | 71.21
  5 |             ->  Vector Foreign Scan on t3          |   1000 | 1MB      |      16 | 30.00
  6 |          ->  Vector Sonic Hash Join (7,9)          |   5001 | 16MB     |      16 | 108.44
  7 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |       8 | 48.10
  8 |                ->  Vector Foreign Scan on t1       |   1000 | 1MB      |       8 | 30.00
  9 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      12 | 48.10
 10 |                ->  Vector Foreign Scan on t2       |   1000 | 1MB      |      12 | 30.00
```

13.3.9.8 Drive Hints

Function

During query plan generation, the optimizer uses dynamic programming or genetic algorithms to enumerate potential join paths and selects the optimal one. As the number of join tables increases, the search space may expand excessively. For such cases, a drive hint can be used to designate the fact table in the query, reducing the scope of the path search space through heuristic search. This hint takes effect only when the GUC parameter **join_search_mode** is set to **heuristic**.

Syntax

```
[no] drive (table)
```

Parameter Description

- **no**: indicates that the table specified by the hint is not a drive table.
- **table**: indicates the table to be hinted. Only one table can be specified. If the table has an alias, the alias should be prioritized for the hint.

 Copyright © Huawei Technologies Co., Ltd.

Examples

Create tables **t1**, **t2**, and **t3**.

```
create table t1(a1 int,b1 int,c1 int,d1 int) store as orc;
create table t2(a2 int,b2 int,c2 int,d2 int) store as orc;
create table t3(a3 int,b3 int,c3 int,d3 int) store as orc;
```

The original statement is as follows:

```
explain select * from t1,t2,t3 where t1.b1=t2.b2 and t1.c1=t3.c3 and t2.d2=t3.d3;
```

```
 id |                      operation                       | E-rows | E-memory | E-width | E-costs
----+------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                      |   1768 |          |      48 | 321.86
  2 |    ->  Vector Streaming (type: GATHER)               |   1768 |          |      48 | 321.86
  3 |       ->  Vector Sonic Hash Join (4,6)               |   1768 | 16MB     |      48 | 205.89
  4 |          ->  Vector Streaming(type: BROADCAST)       |   6000 | 2MB      |      16 | 71.21
  5 |             ->  Vector Foreign Scan on t3            |   1000 | 1MB      |      16 | 30.00
  6 |          ->  Vector Sonic Hash Join (7,9)            |   5000 | 16MB     |      32 | 108.44
  7 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      16 | 48.10
  8 |                ->  Vector Foreign Scan on t1         |   1000 | 1MB      |      16 | 30.00
  9 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      16 | 48.10
 10 |                ->  Vector Foreign Scan on t2         |   1000 | 1MB      |      16 | 30.00
```

In the preceding query, specifying the **t3** table as the driving table allows it to be prioritized in joining with other tables, reducing the search paths for plan generation and altering the join order. The addition of the hint results in the following display:

```
explain select /*+ drive(t3) */ * from t1, t2, t3 where t1.b1=t2.b2 and t1.c1=t3.c3 and t2.d2=t3.d3;
```

Result after hints are used:

```
 id |                      operation                       | E-rows | E-memory | E-width | E-costs
----+------------------------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                                      |   1768 |          |      48 | 321.86
  2 |    ->  Vector Streaming (type: GATHER)               |   1768 |          |      48 | 321.86
  3 |       ->  Vector Sonic Hash Join (4,6)               |   1768 | 16MB     |      48 | 205.89
  4 |          ->  Vector Streaming(type: BROADCAST)       |   6000 | 2MB      |      16 | 71.21
  5 |             ->  Vector Foreign Scan on t2            |   1000 | 1MB      |      16 | 30.00
  6 |          ->  Vector Sonic Hash Join (7,9)            |   5000 | 16MB     |      32 | 108.44
  7 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      16 | 48.10
  8 |                ->  Vector Foreign Scan on t3(DRIVE)  |   1000 | 1MB      |      16 | 30.00
  9 |             ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |      16 | 48.10
 10 |                ->  Vector Foreign Scan on t1         |   1000 | 1MB      |      16 | 30.00
```

CAUTION

When the GUC parameter **join_search_mode** is set to **heuristic** and the number of join tables exceeds **from_collapse_limit**, the optimizer automatically identifies the drive table and displays it in the plan. If the drive table identified by the optimizer is incorrect, the **no drive** hint can be used for correction.

13.3.9.9 Hints for Configuration Parameters

Function

A hint, or a GUC hint, specifies the value of a configuration parameter when a plan is generated.

Precautions

- If the configuration parameter set by a hint takes effect at the statement level, the hint must be written in the top-level query instead of the subquery. For **UNION**, **INTERSECT**, **EXCEPT**, and **MINUS** statements, statement-level GUC hints can be written on any **SELECT** clause that participates in set

 Copyright © Huawei Technologies Co., Ltd.

calculation, the configuration parameters set by this GUC hint take effect for each **SELECT** clause involved in set calculation.

- When a subquery is lifted, all GUC hints on that subquery are discarded.
- If a configuration parameter is set by both a statement-level GUC hint and a subquery-level GUC hint, the subquery-level GUC hint takes effect in the corresponding subquery, statement-level GUC hints take effect in other subqueries of the statement.

Syntax

```
set [global]([@block_name] guc_name guc_value)
```

Parameter Description

- **global** indicates that the parameter set by a hint takes effect at the statement level. If **global** is not specified, the parameter takes effect only in the subquery where the hint is.
- **block_name** indicates the block name of the statement block. For details, see **block_name**.
- **guc_name** indicates the name of the configuration parameter specified by a hint.
- **guc_value** indicates the value of a configuration parameter specified by a hint.

Currently, GUC hints support only certain configuration parameters, and some configuration parameters cannot be set at the subquery level but only at the statement level. Below is the list of supported parameters.

Table 13-14 Configuration parameters supported by GUC hints

Parameter	Configurable at Subquery Level
agg_redistribute_enhancement	Yes
best_agg_plan	Yes
cost_model_version	No
cost_param	No
enable_broadcast	Yes
enable_redistribute	Yes
enable_extrapolation_stats	Yes
enable_hashjoin	Yes
enable_nestloop	Yes
from_collapse_limit	Yes
join_collapse_limit	Yes
join_num_distinct	Yes

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Configurable at Subquery Level
qual_num_distinct	Yes
query_dop	No
rewrite_rule	No
skew_option	Yes

Examples

Hint the query plan in **Examples** as follows:

```
explain
select /*+ set global(query_dop 0) */ i_product_name product_name
...
```

This hint indicates that the **query_dop** parameter is set to **0** during the plan generation of the entire statement, meaning SMP adaptation is enabled.

13.3.9.10 Hints for UDF Fusion

Function

When **sql_opt_options='on(udf_fusion)'** is set, the UDF fusion optimization is enabled. Nested UDFs can be fused into one. When a UDF has multiple UDFs as inputs, only one of them can be selected for fusion. You can use hints to specify two UDFs to be fused or specify a UDF that does not participate in fusion.

Syntax

```
[no] udffusion(udf_name udf_name)
```

Parameter Description

- **no**: indicates that the UDF corresponding to the hint cannot be merged. If multiple UDFs are specified in the brackets, each pair of UDFs cannot be merged. If only one UDF is specified, this UDF cannot be merged with any other UDF.
- **udf_name**: indicates the UDF specified by the hint. You can specify only the function name or specify **schema_name.udf_name**. In the former case, the current schema is used as the schema of the UDF.

Example

```
select /*+ udffusion(udf1 udf3) */ udf1(udf2(c1), udf3(c2)) from t1
```

13.3.9.11 Hint Errors, Conflicts, and Other Warnings

Plan hints change an execution plan. You can run **EXPLAIN** to view the changes.

Hints containing errors are invalid and do not affect statement execution. The errors will be displayed in different ways based on statement types. Hint errors in

 Copyright © Huawei Technologies Co., Ltd.

an **EXPLAIN** statement are displayed as a warning on the interface. Hint errors in other statements will be recorded in debug1-level logs containing the **PLANHINT** keyword.

Hint Error Types

- Syntax errors.

  An error will be reported if the syntax tree fails to be reduced. The No. of the row generating an error is displayed in the error details.

  For example, the hint keyword is incorrect, no table or only one table is specified in the **leading** or **join** hint, or no tables are specified in other hints. The parsing of a hint is terminated immediately after a syntax error is detected. Only the hints that have been parsed successfully are valid.

  For example:

  ```
  leading((t1 t2)) nestloop(t1) rows(t1 t2 #10)
  ```

  The syntax of **nestloop(t1)** is wrong and its parsing is terminated. Only **leading(t1 t2)** that has been successfully parsed before **nestloop(t1)** is valid.

- Semantic errors.
  - An error will be reported if the specified tables do not exist, multiple tables are found based on the hint setting, or a table is used more than once in the **leading** or **join** hint.
  - An error will be reported if the index specified in a scan hint does not exist.
  - If multiple tables with the same name exist after a subquery is pulled up and some of them need to be hinted, add aliases for them to avoid name duplication.

- Duplicated or conflicted hints.

  If hint duplication or conflicts occur, only the first hint takes effect. A message will be displayed to describe the situation.
  - Hint duplication indicates that a hint is used more than once in the same query, for example, **nestloop(t1 t2) nestloop(t1 t2)**.
  - A hint conflict indicates that the functions of two hints with the same table list conflict with each other.

    For example, if **nestloop (t1 t2) hashjoin (t1 t2)** is used, **hashjoin (t1 t2)** becomes invalid. **nestloop(t1 t2)** does not conflict with **no mergejoin(t1 t2)**.

    NOTICE

    Multiple tables in the **leading** hint are split. For example, **leading ((t1 t2 t3))** is split into **leading((t1 t2)) leading(((t1 t2) t3))**. If **leading ((t2 t1))** exists, the two values conflict and the subsequent values are discarded. If two hints use duplicated table lists and only one of them has the specified outer/inner table, the one without a specified outer/inner table becomes invalid.

- A hint becomes invalid after a sublink is pulled up.

 Copyright © Huawei Technologies Co., Ltd.

In this case, a message will be displayed. Generally, such invalidation occurs if a sublink contains multiple tables to be joined, because the table list in the sublink becomes invalid after the sublink is pulled up.

- Unsupported column types.
  - Skew hints are specified to optimize redistribution. They will be invalid if their corresponding columns do not support redistribution.
- Specified hints are not used.
  - If **hashjoin** or **mergejoin** is specified for non-equivalent joins, it will not be used.
  - If **indexscan** or **indexonlyscan** is specified for a table that does not have an index, it will not be used.
  - If **indexscan hint** or **indexonlyscan** is specified for a full-table scan or for a scan whose filtering conditions are not set on index columns, it will not be used.
  - The specified **indexonlyscan** hint is used only when the output column contains only indexes.
  - In equivalent joins, only the joins containing equivalence conditions are valid. Therefore, the **leading**, **join**, and **rows** hints specified for the joins without an equivalence condition will not be used. For example, **t1**, **t2**, and **t3** are to be joined, and the join between **t1** and **t3** does not contain an equivalence condition. In this case, **leading(t1 t3)** will not be used.
  - To generate a streaming plan, if the distribution key of a table is the same as its join key, **redistribute** specified for this table will not be used. If the distribution key and join key are different for this table but the same for the other table in the join, **redistribute** specified for this table will be used but **broadcast** will not.
  - If a hint for an **Agg** distribution column is not used, the possible causes are as follows:
    - The specified distribution key contains data types that do not support redistribution.
    - Redistribution is not required in the execution plan.
    - Wrong distribution key sequence numbers are executed.
    - For AP functions that use the GROUPING SETS and CUBE clauses, hints are not supported for distribution keys in window aggregate functions .
  - If no sublink is pulled up, the specified **blockname** hint will not be used.
  - For unused skew hints, the possible causes are:
    - The plan does not require redistribution.
    - The columns specified by hints contain distribution keys.
    - Skew information specified in hints is incorrect or incomplete, for example, no value is specified for join optimization.
    - Skew optimization is disabled by GUC parameters.

 Copyright © Huawei Technologies Co., Ltd.

- The possible causes of the GUC hint not being used are as follows:
  - The parameter to be configured does not exist.
  - The parameter to be configured does not support GUC hints.
  - The value of the parameter to be configured is invalid.
  - The statement-level GUC hint is not written in the top-level query.
  - The parameter to be configured in the subquery-level GUC hint cannot be set at the subquery level.
  - The subquery where the GUC hint is located is pulled up.

13.3.10 Execution Plan Operator Monitoring

Historical Monitoring Data of Execution Plan Operators

The historical monitoring data of execution plan operators includes the fine-grained execution progress of each operator in a query job, including the start time, execution duration, number of processed rows, memory usage, and the amount of data spilled to disks. The historical data is collected and reported every 5 seconds.

The following table lists the fields in the historical monitoring data of execution plan operators.

Table 13-15 Fields in the historical monitoring data of execution plan operators

Field	Type	Description
session_id	text	Session ID of a statement
statement_id	text	Statement ID
endpoint_id	text	Endpoint ID
update_time	timestamp with time zone	Current report time
query_id	text	debug_query_id, which is applicable to CNs
plan_node_id	int	Unique ID of a node in the execution plan
plan_node_name	text	Name of a node in the execution plan
start_time	timestamp	Start time
duration	bigint	Execution duration
status	text	Current status

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
query_dop	int	Query parallelism degree
estimated_rows	bigint	Estimated number of rows
tuple_processed	bigint	Number of rows that have been processed
min_peak_memory	int	Minimum peak memory used
max_peak_memory	int	Maximum peak memory used
average_peak_memory	int	Average peak memory used
memory_skew_percent	int	Memory usage skew percentage
min_spill_size	int	Minimum amount of data spilled to disks
max_spill_size	int	Maximum amount of data spilled to disks
average_spill_size	int	Average amount of data spilled to disks
spill_skew_percent	int	Skew percentage of spilled data
min_cpu_time	bigint	Minimum CPU time of a node
max_cpu_time	bigint	Maximum CPU time of a node
total_cpu_time	bigint	Total CPU time of all nodes
cpu_skew_percent	bigint	CPU usage skew percentage
warning	text	Warning information displayed during execution
parent_id	int	Parent node ID
exec_count	int	Number of times the node has been executed
progress	text	Execution progress, in percentage

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
min_net_size	bigint	Minimum amount of data transmitted over the network
max_net_size	bigint	Maximum amount of data transmitted over the network
total_net_size	bigint	Total amount of data transmitted over the network
min_read_size	bigint	Minimum amount of data read by a node
max_read_size	bigint	Maximum amount of data read by a node
total_read_size	bigint	Total amount of data read by all nodes
min_write_size	bigint	Minimum amount of data written by a node
max_write_size	bigint	Maximum amount of data written by a node
total_write_size	bigint	Total amount of data written by all nodes

Real-Time Monitoring Data of Execution Plan Operators

The real-time monitoring data of execution plan operators includes the fine-grained execution progress of each operator in a query job, including the start time, execution duration, number of processed rows, memory usage, and the amount of data spilled to disks. The background thread samples data every 5 seconds. The real-time monitoring data only includes information about the operators that are being executed during the latest sampling. Information about the statements that have been completed cannot be obtained.

Usage method:

```
select * from pgxc_wlm_operator_statistics(query_id);
```

The following table lists the input parameter.

 Copyright © Huawei Technologies Co., Ltd.

Table 13-16 Input parameter

Parameter	Type	Description
query_id	text	Query ID of a statement If the value is **0**, information about all queries is returned.

The fields displayed in the real-time monitoring data of execution plan operators are as follows.

Table 13-17 Fields in the real-time monitoring data of execution plan operators

Field	Type	Description
session_id	text	Session ID of a statement
statement_id	text	Statement ID
username	text	Username
query_id	text	debug_query_id, which is applicable to CNs
tid	bigint	Thread ID
plan_node_id	int	Unique ID of a node in the execution plan
plan_node_name	text	Name of a node in the execution plan
start_time	timestamp	Start time
duration	bigint	Execution duration
status	text	Current status
query_dop	int	Query parallelism degree
estimated_rows	bigint	Estimated number of rows
tuple_processed	bigint	Number of rows that have been processed
min_peak_memory	int	Minimum peak memory used
max_peak_memory	int	Maximum peak memory used

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
average_peak_memory	int	Average peak memory used
memory_skew_percent	int	Memory usage skew percentage
min_spill_size	int	Minimum amount of data spilled to disks
max_spill_size	int	Maximum amount of data spilled to disks
average_spill_size	int	Average amount of data spilled to disks
spill_skew_percent	int	Skew percentage of spilled data
min_cpu_time	bigint	Minimum CPU time of a node
max_cpu_time	bigint	Maximum CPU time of a node
total_cpu_time	bigint	Total CPU time of all nodes
cpu_skew_percent	bigint	CPU usage skew percentage
warning	text	Warning information displayed during execution
parent_id	int	Parent node ID
exec_count	int	Number of times the node has been executed
progress	text	Execution progress, in percentage
min_net_size	bigint	Minimum amount of data transmitted over the network
max_net_size	bigint	Maximum amount of data transmitted over the network
total_net_size	bigint	Total amount of data transmitted over the network
min_read_size	bigint	Minimum amount of data read by a node

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Description
max_read_size	bigint	Maximum amount of data read by a node
total_read_size	bigint	Total amount of data read by all nodes
min_write_size	bigint	Minimum amount of data written by a node
max_write_size	bigint	Maximum amount of data written by a node
total_write_size	bigint	Total amount of data written by all nodes

13.4 SQL Tuning Examples

13.4.1 Case: Adjusting the GUC Parameter best_agg_plan

Symptom

The t1 table is defined as follows:

```
create table t1(a int, b int, c int) store as orc;
```

Assume that the distribution column of the result set provided by the agg lower-layer operator is setA, and the group by column of the agg operation is setB, the agg operations can be performed in two scenarios in the stream framework.

Scenario 1: setA is a subset of setB.

In this scenario, the aggregation result of the lower-layer result set is the correct result, which can be directly used by the upper-layer operator.

Scenario 2: setA is not a subset of setB.

In this scenario, the Stream execution framework is classified into the following three plans:

hashagg+gather(redistribute)+hashagg

redistribute+hashagg(+gather)

hashagg+redistribute+hashagg(+gather)

Aura offers the GUC parameter **best_agg_plan** to intervene in the execution plan, forcing it to generate the corresponding execution plan mentioned above. The options are **0**, **1**, **2**, and **3**.

- The value **1** forces the generation of the first plan.
- The value **2** forces the generation of the second plan if the group by columns can be redistributed. Otherwise, it generates the first plan.

 Copyright © Huawei Technologies Co., Ltd.

- The value **3** forces the generation of the third plan if the group by columns can be redistributed. Otherwise, it generates the first plan.
- The value **0** allows the optimizer to select the best plan based on the estimated costs of the above three plans.

Possible impacts are as follows:

```
set best_agg_plan=1;
SET
explain select a, count(1) from t1 group by a;
QUERY PLAN
-------------------------------------------------------------------------------------------
id |                operation                 | E-rows | E-memory | E-width | E-costs
----+------------------------------------------+--------+----------+---------+---------
1 | ->  Row Adapter                          |    200 |          |      12 | 85.16
2 |    ->  Vector Sonic Hash Aggregate       |    200 |          |      12 | 85.16
3 |       ->  Vector Streaming (type: GATHER) |    522 |          |      12 | 85.16
4 |          ->  Vector Sonic Hash Aggregate |    522 | 16MB     |      12 | 63.41
5 |             ->  Vector Foreign Scan on t1 |   1000 | 1MB      |       4 | 60.00

Predicate Information (identified by plan id)
-----------------------------------------------
5 --Vector Foreign Scan on t1
Server Type: lf
Total files left: 0

====== Query Summary =====
------------------------------
System available mem: 262144KB
Query Max mem: 262144KB
Query estimated mem: 1872KB
(19 rows)

set best_agg_plan=2;
SET
explain select a, count(1) from t1 group by a;
QUERY PLAN
-------------------------------------------------------------------------------------------------
id |                   operation                   | E-rows | E-memory | E-width | E-costs
----+-----------------------------------------------+--------+----------+---------+---------
1 | ->  Row Adapter                               |    201 |          |      12 | 78.84
2 |    ->  Vector Streaming (type: GATHER)        |    201 |          |      12 | 78.84
3 |       ->  Vector Sonic Hash Aggregate         |    201 | 16MB     |      12 | 70.47
4 |          ->  Vector Streaming(type: REDISTRIBUTE) |   1000 | 2MB      |       4 | 67.39
5 |             ->  Vector Foreign Scan on t1     |   1000 | 1MB      |       4 | 60.00

Predicate Information (identified by plan id)
-----------------------------------------------
5 --Vector Foreign Scan on t1
Server Type: lf
Total files left: 0

====== Query Summary =====
------------------------------
System available mem: 262144KB
Query Max mem: 262144KB
Query estimated mem: 3965KB
(19 rows)

set best_agg_plan=3;
SET
explain select a, count(1) from t1 group by a;
QUERY PLAN
-------------------------------------------------------------------------------------------------
id |                   operation                   | E-rows | E-memory | E-width | E-costs
----+-----------------------------------------------+--------+----------+---------+---------
1 | ->  Row Adapter                               |    200 |          |      12 | 82.25
2 |    ->  Vector Streaming (type: GATHER)        |    200 |          |      12 | 82.25
```

```
3 |       ->  Vector Sonic Hash Aggregate             |    200 | 16MB     |      12 | 70.25
4 |         ->  Vector Streaming(type: REDISTRIBUTE) |    522 | 2MB      |      12 | 68.03
5 |           ->  Vector Sonic Hash Aggregate         |    522 | 16MB     |      12 | 63.41
6 |             ->  Vector Foreign Scan on t1         |   1000 | 1MB      |       4 | 60.00

Predicate Information (identified by plan id)
-----------------------------------------------
6 --Vector Foreign Scan on t1
Server Type: lf
Total files left: 0

====== Query Summary =====
------------------------------
System available mem: 262144KB
Query Max mem: 262144KB
Query estimated mem: 3968KB
(20 rows)
```

Summary

Generally, the optimizer chooses an optimal execution plan, but the cost estimation, especially that of the intermediate result set, has large deviations, which may result in large deviations in agg calculation. In this case, you need to use best_agg_plan to adjust the agg calculation model.

When the aggregation convergence ratio is very small, that is, the number of result sets does not become small obviously after the agg operation (5 times is a critical point), you can select the redistribute+hashagg or hashagg+redistribute +hashagg execution mode.

 Copyright © Huawei Technologies Co., Ltd.

14 GUC Parameters

14.1 Viewing and Setting GUC Parameters

To ensure the optimal performance of Aura, you can adjust the GUC parameters in the database.

The database offers numerous operation parameters. Configuring these can impact the behavior of the database system. When modifying these settings, make sure you comprehend their impact on the database to avoid unforeseen results.

Precautions

- If the value range of a parameter is a string, the string should follow the OS's rules for path and file name naming conventions.
- For parameters with a maximum value of **INT_MAX**, the maximum value varies depending on the OS.
- For parameters with a maximum value of **DBL_MAX**, the maximum value varies depending on the OS.

Parameter Types and Values

- All parameter names are case-insensitive. Parameter values can be integer, float, string, boolean, or enum types.

  Boolean values can be (**on**/**off**), (**true**/**false**), (**yes**/**no**) or (**1**/**0**), and are not case-sensitive.

  Strings are classified into regular strings and JSON strings:

  Regular string: The GUC value is a string.

  JSON string: The GUC value is a JSON string that contains multiple key-value pairs. Each key represents a parameter, and the corresponding value represents the parameter value.
- For parameters with units, always specify the unit when setting them. Otherwise, default units will apply.
  - Memory units include: KB, MB, and GB.
  - Time units: ms, s, min, h, and d.

 Copyright © Huawei Technologies Co., Ltd.

Viewing and Setting GUC Parameters

In Aura, you can set GUC parameters only by running the SET command via SQL statements. The syntax is as follows:

```
SET paramName TO paramValue;
```

For example, to set the **statement_timeout** parameter, you can use the REST API to send **SET statement_timeout TO 60000**, thereby setting it to 10 minutes.

To check the current settings of GUC parameters within a session, use the SHOW statement like so:

```
SHOW statement_timeout;
```

This allows you to query the currently configured statement timeout duration.

14.2 Connection and Authentication

14.2.1 Security and Authentication

This section describes parameters related to the security authentication modes of the client and server.

Parameter Description

Table 14-1 Security and authentication parameters

Parameter	Type	Description
session_timeout	Integer	**Definition**: Specifies the maximum duration for which no operation is performed after a connection is established with the server. **Range**: An integer ranging from 0 to 86400. The minimum unit is seconds. **0** means to disable the timeout. **Default Value**: 10 min
hang_timeout	Integer	**Definition**: Specifies the timeout period for kernel hang detection. **Range**: An integer ranging from 0 to 2147483647. The minimum unit is minute. A value of **0** disables the hang detection feature. **Default Value**: 5 min

14.3 Query Planning

14.3.1 Optimizer Method Configuration

The following configuration parameters provide a basic way to influence the query optimizer's choice of execution plans. If the default plan chosen by the optimizer

 Copyright © Huawei Technologies Co., Ltd.

for a specific query is not optimal, these configuration parameters can be used to force the optimizer to select a different plan as a temporary solution. Better methods include tuning the optimizer cost constants, manually running **ANALYZE**, and increasing the value of the **default_statistics_target** parameter.

Parameter Description

Table 14-2 Optimizer method configuration parameters

Parameter	Type	Description
enable_hashjoin	Boolean	**Definition**: Controls whether the optimizer uses hash join plans. **Range**: Boolean. ● **on**: It is used. ● **off**: It is not used. **Default Value**: **on**.
enable_nestloop	Boolean	**Definition**: Controls whether the optimizer uses the nested loop join plan type for full table scans of internal tables. It is impossible to completely eliminate nested loop joins, but disabling this variable will cause the optimizer to prefer other methods when they are available. **Range**: Boolean. ● **on**: It is used. ● **off**: It is not used. **Default Value**: **off**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
best_agg_plan	Integer	**Definition**: For the Agg operation in the stream, the optimizer generates three plans: 1. hashagg+gather(redistribute)+hashagg 2. redistribute+hashagg(+gather) 3. hashagg+redistribute+hashagg(+gather) This parameter controls which hash aggregation plan the optimizer generates. **Range**: An integer, which can be **0**, **1**, **2**, or **3**. ● The value **1** forces the generation of the first plan. ● The value **2** forces the generation of the second plan if the group by columns can be redistributed. Otherwise, it generates the first plan. ● The value **3** forces the generation of the third plan if the group by columns can be redistributed. Otherwise, it generates the first plan. ● The value **0** allows the optimizer to select the best plan based on the estimated costs of the above three plans. **Default Value**: **0**.
turbo_engine_version	Integer	**Definition**: Specifies whether to use the turbo execution engine. If the turbo storage format is specified for a table (the **enable_turbo_store** parameter in the table attribute is set to **on**) and the query does not involve the merge join or sort agg operator, the turbo execution engine can be used to multiply the performance of the executor. **Range**: An integer, which can be **0**, **1**, **2**, or **3**. ● The value **0** disables the turbo execution engine. ● The value **1** enables the turbo execution engine only for single-table aggregate queries. ● The value **2** enables the turbo execution engine only for single-table aggregate or multi-table join queries. ● If the value is **3**, the turbo execution engine can be used to accelerate the execution of most common operators, but not supported by some operators such as merge join and sort agg. When the data volume is large and the value of **turbo_engine_version** is **3**, the merge join and sort agg operators are seldom used. Therefore, the turbo execution engine can be used to accelerate the execution of almost all SQL statements. **Default Value**: **3**. **NOTE** You are advised not to enable the turbo execution engine in cross-VW scenarios.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
agg_redistribute_enhancement	Boolean	**Definition**: Specifies the policy for selecting a redistribution column when multiple GROUP BY columns are involved in the Agg operation and none of them is a distribution key. This parameter is used to control the policy for selecting a redistribution column. **Range**: Boolean. • **on** indicates that the redistribution column with the largest estimated distinct value is selected as the redistribution column. • **off** indicates the first column that can be redistributed for redistribution. **Default Value**: **off**.
skew_option	String	**Definition**: Controls whether to use the optimization policy. **Range**: A string. • **off**: Disables the policy. • **normal**: Uses the radical policy. For scenarios where skew occurrence is uncertain, assumes skew exists and performs corresponding optimizations. • **lazy**: Uses a conservative policy. If it is uncertain whether skew occurs, the system considers that no skew occurs and does not perform optimization. **Default Value**: **normal**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description

rewrite_rule	String	**Definition**: Specifies the enabled optional query rewriting rules. In specific scenarios, this GUC parameter is used to set query rewriting rules to achieve optimal query efficiency.
		This parameter can be used to control the combination of query rewriting rules. For example, there are multiple rewriting rules: rule1, rule2, rule3, and rule4. You can set this parameter as follows:
		`set rewrite_rule='on(rule1)';  -- Enable the query rewriting rule rule1.`
		`set rewrite_rule='off(rule1)';  -- Disable the query rewriting rule rule1.`
		`set rewrite_rule='rule2,rule3';  -- Enable the query rewriting rules rule2 and rule3.`
		`set rewrite_rule=none;  -- Disable all optional query rewriting rules.`
		Range: A string, which can be a combination of multiple optional query rewriting rules, including:
		● **none**: No optional query rewriting rules are used.
		● **lazyagg**: The Lazy Agg query rewriting rule is used to eliminate aggregation in subqueries.
		● **magicset**: The Magic Set query rewriting rule is used (conditions are pushed down from the main query to subqueries).
		● **uniquecheck**: The Unique Check query rewriting rule is used (expression sublinks without aggregate functions in target columns can be pulled up. This rule can be enabled only when the target column values are unique after the sublinks are aggregated by the associated columns. It is recommended that this rule be used by professional tuning personnel).
		● **disablerep**: The rule for forbidding sublink pull-up for replicated tables is used.
		● **notinopt**: The Not In query rewriting rule is used (Not In is converted to a more efficient hash join).
		● **or_conversion**: The OR query rewriting rule is used (complex predicates containing OR are split or rewritten).
		● **projection_pushdown**: The projection pushdown query rewriting rule is used.
		● **plain_lazyagg**: The plain lazyagg query rewriting rule is used (for single subqueries and distinct operations).
		● **eager_magicset**: The eager magic set query rewriting rule is used (qual conditions can be pushed down to subqueries with GROUP BY or PARTITION BY).
		● **outer_join_quality_imply**: The outer join query rewriting rule is used (attributes can be transferred

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
		from the outer table to the inner table in an outer join). • **inlist_merge**: The inlist merge query rewriting rule is used (multiple INs in the same column are combined into one IN(List)). • **casewhen_simplification**: The CASE WHEN query rewriting rule is used (constant folding is performed on the CASE WHEN..THEN...ELSE...END statement, and unreachable branches are removed to simplify calculation). • **subquery_qual_pull_up**: The subquery pull-up query rewriting rule is used (filter conditions in subqueries are pulled up to the outer query). • **join_elimination**: The join elimination query rewriting rule is used (joins that do not affect the final result are deleted). • **not_distinct_from_opt**: The NOT DISTINCT FROM query rewriting rule is used (IS NOT DISTINCT FROM is converted to = to support hash join). • **foreign_agg_opt**: The query rewriting rule for partial aggregate functions is used (the result set is queried through the file footer for the count(1), max, and min functions). • **selfjoin_elimination**: The self-join query rewriting rule is used (the self-join of two tables in the EXISTS and ANY sublinks is eliminated to a single-table aggregation). • **remove_groupby**: The remove groupby query rewriting rule is used (redundant grouping columns are deleted). • **sum_opt**: The SUM optimization query rewriting rule is used (extracting common expressions from SUM to reduce the number of SUM aggregations). **Default Value**: **magicset**, **or_conversion**, **projection_pushdown**, **plain_lazyagg**, **subquery_qual_pull_up**, **not_distinct_from_opt**, **foreign_agg_opt**, **selfjoin_elimination**, **remove_groupby**, or **sum_opt**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
cte_options	String	**Definition**: Specifies the prerequisites for selecting CteScan in a plan. The parameter value is a string of comma-separated key-value pairs, with each pair in the format of (*key:value*). For details about the values and functions of each parameter, see the table of **cte_options** parameters and values. **Range**: A string of key-value pairs. The sub-items include **pred_threshold**, **max_mem**, and **ref_count_threshold**. For details about the value ranges of each sub-item, see the table of **cte_options** parameters and values. **Default Value**: **(pred_threshold:2),(max_mem:256),(ref_count_threshold:4)**.
analyze_mode	String	**Definition**: Specifies the ANALYZE options. This parameter can be used to set a combination of multiple optimization options. For details, see the description of **rewrite_rule**. **Range**: A string. ● **none**: No special configuration is required. ● **auto**: Enable auto-analyze. ● **file_sample**: Enable file-level sampling. ● **partition**: Enable partition-level ANALYZE. **Default Value**: **none**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
sql_opt_options	String	**Definition**: Configures the optimization options for generating SQL plans. This parameter can be used to set a combination of multiple optimization options. For details, see the description of **rewrite_rule**. **Range**: A string. • **none**: No special configuration is required. • **cte_scan**: Enables shared scan for **WITH** clause. • **common_expr_reuse**: Enables expression reuse for non-result-set expressions. • **project_set**: Enables expression reuse for result-set functions and optimizes execution logic using **project_set**. • **udf_split**: Splits nested user-defined functions into different operators and uses the stream operator to execute the functions concurrently. • **async_udf**: Enables asynchronous, concurrent execution of multiple UDFs at the same level. • **iceberg_bucket_stream_opt**: Supports bucket aware joins for Iceberg bucket partitioned tables, reducing stream overhead. • **udf_fusion**: Nested UDFs, such as udf1(udf2), can be combined into a single UDF and executed within one actor. • **objectref_transfer**: Data of the bytea type can be stored in the distributed shared memory and transferred between operators as ObjectRef. • **udf_variadic_input**: A UDF can accept multiple independent ObjectRef parameters, with a maximum of 32 ObjectRef parameters supported. • **arrow_column_convert**: Vectorized conversion between Arrow and Turbo. **Default Value**: **cte_scan,iceberg_bucket_stream_opt**.

14.3.2 Other Optimizer Options

The quality of a query plan directly affects the SQL execution performance. Settings such as statistics sampling, planning constraints, degree of parallelism, and inter-node communication mode affect the selection of query plans. This section describes other optimizer options, including the statistics sampling target (**default_statistics_target**), planning collapse limits (**from_collapse_limit** and **join_collapse_limit**), SMP and actor parallelism (**query_dop** and **fq_query_vdn**), and inter-node communication mode (**enable_libcomm**). You can adjust these parameters based on the data scale and service load to improve the accuracy and execution efficiency of query plans.

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 14-3 Other optimizer options

Parameter	Type	Description
default_statistics_target	Float	**Definition**: Specifies the default statistics target for tables that have not had a statistics target set for any column with **ALTER TABLE SET STATISTICS**. If this parameter is set to a positive number, it indicates the number of samples for statistics. If this parameter is set to a negative number, it indicates that the statistics target is set in percentage. The negative number is converted to the corresponding percentage. For example, **-5** indicates 5%. During sampling, the value of **default_statistics_target** multiplied by 300 is used as the size of random sampling. For example, if the default value is **100**, 30,000 pages are randomly read and 30,000 data records are randomly selected from the pages to complete random sampling. **Range**: A floating point number ranging from –100 to 10,000. **NOTE** • A larger positive number than the default value increases the time required to do **ANALYZE**, but might improve the quality of the optimizer's estimates. • Adjusting this parameter may cause performance deterioration. If a query is degraded, you can restore the default statistics or use plan hints to adjust the query plan to the previous one. • When this GUC parameter is set to a negative value, if the computed sample size is greater than or equal to 2% of the total data volume and the table contains fewer than 1,600,000 rows, the **ANALYZE** duration may exceed that of the default GUC parameter setting. • **AUTOANALYZE** does not support customizing the sampling size for temporary tables. The default value is used instead. • If statistics are forcibly calculated in-memory, the sampling size is constrained by the **maintenance_work_mem** parameter. **Default Value**: **100**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
from_collapse_limit	Integer	**Definition**: Specifies whether the optimizer merges subqueries into upper-level queries based on the number of items in the generated FROM list. If the number of items in the FROM list is less than or equal to this value, the optimizer merges subqueries into upper-level queries. **Range**: An integer, in the range [1, INT_MAX]. **NOTE** Smaller values reduce planning time but may lead to inferior execution plans. **Default Value**: **8**.
join_collapse_limit	Integer	**Definition**: Specifies whether the optimizer rewrites the **JOIN** constructs except **FULL JOINS** into the **FROM** list based on the number of items in the list. **Range**: An integer, in the range [1, INT_MAX]. **NOTE** • Setting this parameter to **1** prevents any JOIN reordering. In this way, the join order specified in the query is the actual join order. The query optimizer does not always select the optimal join order. Advanced users can temporarily set this variable to **1** and then specify the required join order. • Smaller values reduce planning time but degrade the quality of execution plans. **Default Value**: **8**.
enable_libcomm	Boolean	**Definition**: Specifies whether DNs communicate with each other through libcomm. **Range**: Boolean. • **on**: libcomm is used for communication between DNs. • **off**: The stream is used for communication between DNs. **Default Value**: **off**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
enable_bloom_filter	Boolean	**Definition**: Specifies whether BloomFilter optimization is allowed. This function is applicable to scenarios where a **HASH JOIN** on the foreign table side, which includes HDFS internal/foreign tables or column-store tables in the same thread, triggers the Bloom Filter optimization. **Constraints**: 1. The JOIN type can only be **INNER JOIN**, **SEMI JOIN**, **RIGHT JOIN**, **RIGHT SEMI JOIN**, **RIGHT ANTI JOIN**, or **RIGHT ANTI FULL JOIN**. 2. Join condition on the inner table side of JOIN: The join condition cannot be an expression for HDFS internal and foreign tables. The join condition can be an expression for column-store tables, but only for non-JOIN layer calculation. 3. The association condition of the JOIN foreign table must be simple column association. 4. If the association conditions of both the internal and external tables of JOIN are simple column association, more than 1/3 of the data can be removed from the estimation at the plan layer (only for HDFS internal and external tables). 5. The JOIN cannot contain null value associations. 6. Data types: ● The HDFS internal and external table field types support **SMALLINT**, **INTEGER**, **BIGINT**, **REAL**/**FLOAT4**, **DOUBLE PRECISION**/**FLOAT8**, **CHAR(n)**/**CHARACTER(n)**/**NCHAR(n)**, **VARCHAR(n)**/**CHARACTER VARYING(n)**, **CLOB**, and **TEXT**. ● The column-store table supports the **SMALLINT**, **INTEGER**, **BIGINT**, **OID**, "**char**", **CHAR(n)**/**CHARACTER(n)**/**NCHAR(n)**, **VARCHAR(n)**/**CHARACTER VARYING(n)**, **NVARCHAR2(n)**, **CLOB**, **TEXT**, **DATE**, **TIME**, **TIMESTAMP**, and **TIMESTAMPTZ** field types. The collation of the character type must be **C**. **Range**: Boolean. ● **on**: The BloomFilter optimization can be used. ● **off**: The BloomFilter optimization cannot be used. **Default Value**: None.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
enable_extrapolation_stats	Boolean	**Definition**: Specifies whether to allow the use of inference-based estimation logic based on historical statistics. Using this logic can increase the possibility of accurate estimation for tables whose statistics are not collected in time. However, there is also a possibility that the estimation is too large due to incorrect inference. **Range**: Boolean. ● **on**: The inference-based estimation logic is allowed to be used based on historical statistics. ● **off**: The inference-based estimation logic is not allowed to be used based on historical statistics. **Default Value**: **off**.
query_dop	Integer	**Definition**: Specifies the user-defined query parallelism. **Range**: An integer, in the range [–64, 64]. ● [–64, –1]: SMP adaptation is enabled, and the maximum degree of parallelism that can be dynamically selected is limited. ● **0**: SMP adaptation is enabled. The system dynamically selects the optimal degree of parallelism for each query based on resource conditions and plan characteristics, within the range of [1, 8] on x86 platforms and [1, 64] on Kunpeng platforms. ● [1, 64]: Fixed SMP is enabled, and the system will use the specified degree of parallelism. **Default Value**: **1**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
smp_thread_cost	Float	**Definition**: Specifies the minimum estimated cost that each SMP thread should carry when planning the degree of parallelism in the SMP adaptive scenario. If the estimated cost of an operator is low but the degree of parallelism is high, causing the estimated cost calculated by each SMP thread to be less than this threshold, the optimizer automatically reduces the degree of parallelism of the operator. This prevents system resources from being wasted by enabling a high degree of parallelism for low-cost operators. **Range**: A floating point number, in the range [0, 10000]. ● **0**: The estimated cost threshold of the SMP thread is disabled. In this case, the DOP is not affected by the operator cost. ● (0, 10000]: Minimum estimated cost threshold of SMP threads. A larger value indicates that the DOP of low-cost operators tends to decrease. **NOTE** If this parameter is not set to **0** and the estimated cost of an operator is too small, the degree of parallelism in SMP is low. In this case, you can set this parameter to **0** to increase the degree of parallelism. **Default Value**: **1000**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
plan_mode_seed	Integer	**Definition**: This parameter is a debug parameter. It controls whether the optimizer uses the dynamic programming algorithm to estimate the optimal execution plan or generates a random plan. Parameter type: USERSET. **Range**: An integer, in the range [–1, 2147483647]. ● **0**: Optimal execution plan whose cost is estimated using the dynamic planning algorithm. ● **–1**: The optimizer randomly generates a random integer ranging from 1 to 2147483647 and generates a random execution plan based on the random number. ● [1, 2147483647]: The optimizer generates a random execution plan based on the specified random number. **NOTE** ● When this parameter is set to the random plan mode, the optimizer generates different random execution plans, which may not be the optimal plan. Therefore, the random plan mode affects the query performance. You are advised to set this parameter to the default value **0** during normal service operations or O&M. ● If this parameter is not set to **0**, the specified plan hint does not take effect. **Default Value**: **0**.
fq_query_vdn	Integer	**Definition**: Specifies the number of actors that can be started for a task. **Range**: An integer, in the range [0, 256]. ● **0**: The number of actors is adaptively determined. The number of actors is dynamically selected based on the scanning and write costs. ● **1**: A single actor is used, and the execution mode is degraded to the single CN mode. ● 2–256: Use a specified number of actors for execution. **Default Value**: **2**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
fq_actor_adapt_type	Enum	**Definition**: Specifies the mode in which actors are started adaptively. **Range**: Enumerated values. • **default**: The adaptive mode is not used. The default specifications of the actor are 4 vCPUs and 16 GB memory. • **mem**: The actor specifications are automatically adapted. The ratio of CPU to memory is 1:4, and the CPU range is [1, 16]. **NOTE** This parameter only takes effect when actor count adaptation is enabled (that is, fq_query_vdn = 0). **Default Value**: **mem**.
fq_dynamic_actor_cost_threshold	Float	**Definition**: Specifies the threshold of the single-CN cost when **fq_query_vdn** is set to **0**. If the threshold is exceeded, the actor is started. **Range**: A floating point number, in the range [1, 1e10]. **Default Value**: **200000**. **NOTE** During adaptation, the number of selected actors and the cost are not in a simple linear relationship. Instead, they are in a process of gradient descent. As the cost increases, the number of selected actors tends to 256.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
objref_transfer_options	String	**Definition**: Specifies the settings related to big data transfer. The parameter value is a string of comma-separated key-value pairs, with each pair in the format of (*key*:*value*). This parameter takes effect only when **objectref_transfer** is set in **sql_opt_options**. **Range**: A string of key-value pairs. • **release_batch_threshold**: The value is an integer in the range [1, INT_MAX]. The default value is **16**. This parameter controls the number of batches that the consumer accumulates before notifying the producer to release ObjectRef. • **width_threshold**: The value is an integer in the range [0, INT_MAX] (the unit cannot be added to the value, and the unit is byte). The default value is **0**. This parameter controls the number of bytes that the estimated column width exceeds. The default value is **0**, indicating that this parameter does not take effect. The bytea column is not controlled by the second parameter. • **row_serialize**: The value is of Boolean type. The default value is **off**. This parameter controls whether to accumulate batches and send them in serialized mode for streams except broadcast and roundrobin that is sent by batch. **off**/**false** indicates that streams are serialized and sent in batches. **on**/**true** indicates that streams are serialized and sent row by row. **Default Value**: **(release_batch_threshold:16), (width_threshold:0)**. **Example**: `set objref_transfer_options = "(release_batch_threshold:16), (width_threshold:0)";`

14.4 Default Settings of Client Connection

14.4.1 Statement Behavior

The object search order, statement execution timeout control, data cache and I/O scheduling, result set output format, and log level during SQL statement execution all affect query efficiency and the convenience of fault locating. This section describes the GUC parameters that control the execution of SQL statements, including **search_path**, **statement_timeout**, disk cache and asynchronous I/O, runtime filter, result set format, and log level. You can adjust these parameters

 Copyright © Huawei Technologies Co., Ltd.

based on service characteristics to optimize query performance and meet monitoring and troubleshooting requirements.

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 14-4 Statement behavior parameters

Parameter	Type	Description
search_path	String	**Definition**: Specifies the order in which schemas are searched when a referenced object does not specify a schema. Its value consists of one or more schema names, with different schema names separated by commas. ● When the current session stores temporary tables in a schema, you can use the alias **pg_temp** to list it in the search path, for example, **pg_temp, public**. The schema that stores temporary tables is always searched first, before all schemas in **pg_catalog** and **search_path**, that is, it has the highest search priority. You are advised not to explicitly set **pg_temp** in **search_path**. If **pg_temp** is specified in **search_path** but not at the beginning, the system will display a message indicating that the setting is invalid, and **pg_temp** will still be searched first. By using the alias **pg_temp**, the system only searches for database objects such as tables, views, and data types in the schema that stores temporary tables, and does not search for database objects such as functions or operators in the schema. ● The schema **pg_catalog**, where system catalogs reside, is always searched before all schemas specified in **search_path**, that is, it has the second-highest search priority (with **pg_temp** having the highest). You are advised not to explicitly set **pg_catalog** in **search_path**. If **pg_catalog** is specified in **search_path** but not at the beginning, the system will display a message indicating that the setting is invalid, and **pg_catalog** will still be searched with the second-highest priority. ● If no particular schema is specified when an object is created, it is placed in the first named schema in **search_path**. If the search path is empty, an error is reported. ● The effective value of the current search path can be detected using the SQL function **current_schema**. This differs from checking the value of **search_path**, because **current_schema** displays the first valid schema name in **search_path**. **Range**: A string. ● Setting it to an empty string ('') causes the system to convert it to a pair of double quotes.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
		• Including double quotes in the setting is considered unsafe. The system converts each double quote into a pair of double quotes. **Default Value**: None.
current_schema	String	**Definition**: Specifies the current schema. **Range**: A string. **Default Value**: **default_db**.
statement_timeout	Integer	**Definition**: Specifies the statement execution timeout. If the time that a statement is executed exceeds the value of this parameter (starting from the time when the server receives the command), an error is reported and the statement exits the execution. **Range**: An integer ranging from 0 to 2147483647. The unit is ms. **Default Value**: **0**.
bytea_output	Enum	**Definition**: Specifies the output format of values of the bytea type. **Range**: Enumerated values. - **hex**: Encodes binary data into two hexadecimal digits for each byte. - **escape**: Traditional PostgreSQL format. It uses an ASCII character sequence to represent a binary string and converts binary strings that cannot be represented as ASCII characters into special escape sequences. **Default Value**: **hex**.
enable_disk_cache	Boolean	**Definition**: Specifies whether to enable disk cache and data prefetching. Currently, data prefetching is valid only for files in Parquet or ORC format, and disk cache is valid for files in all formats. **Range**: Boolean. - **on**: Enables disk cache and data prefetching. - **off**: Disables disk cache and data prefetching. **Default Value**: **on**.
disk_cache_max_size	Integer	**Definition**: Specifies the total size of the disk cache. **Range**: An integer ranging from 512 MB to 1 PB. **Default Value**: 5 GB.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
enable_aio_scheduler	Boolean	**Definition**: Specifies whether to enable asynchronous I/O scheduling, which is the basis of asynchronous read and write. **Range**: • **on** or **true** indicates that I/O scheduling is enabled. • **off** or **false** indicates that I/O scheduling is disabled. **Default Value**: **on**.
runtime_filter_type	String	**Definition**: Specifies the type of the runtime filter to be used. **Range**: A string. • **all**: All runtime filters are applied. • **min_max**: Only the runtime filter in join scenarios is applied, and only a **min_max** filter is generated in such scenarios. • **bloom_filter**: Only the runtime filter in join scenarios is applied, and after meeting certain conditions, a **bloom_filter** is generated for filtering. • **global_filter**: Cross-DN node runtime filter in join scenarios. Once enabled, it supports generating **min_max** or **bloom_filter** filters for data on different DNs. • **topn_filter**: The runtime filter in the TopN sorting (ORDER BY... LIMIT) scenario is applied. After this parameter is enabled, runtime filters can generate filters for excluding invalid data based on the real-time data of the TopN. • **none**: No runtime filter is used. Only the original **bloom_filter** remains effective for filtering. **Default Value**: **min_max**, **bloom_filter**, **global_filter**, or **topn_filter**.
enable_meta_scan	Boolean	**Definition**: Specifies whether to enable metaScan during Iceberg query. **Range** • **on** or **true**: Enables metaScan. That is, the DN obtains the list of files to be scanned in a distributed manner during the query. • **off** or **false**: Disables metaScan. That is, the CN obtains the list of files to be scanned during the query. **Default Value**: **true**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
enable_spill_to_remote_storage	Boolean	**Definition**: Specifies whether to enable the spill-to-obs feature. If this parameter is set to **true**, the feature is enabled. In this case, the data spilled to disks is managed and controlled by the disk cache, and OBS is used as the emergency channel when the space is insufficient. If this parameter is set to **false**, the old spill-to-disk mode is used, and data is directly written to the local EVS disk. This parameter has a parameter dependency. When the spill-to-obs feature is enabled (**enable_spill_to_remote_storage** is set to **true**), the **use_yr_as_block_cache_backend** parameter must be set to **false**. **Range** ● **on** or **true**: Enables the feature. ● **off** or **false**: Disables the feature. **Default Value**: **true**.
enable_remote_cache	Boolean	**Definition**: Specifies whether to use the Cache service as the cross-statement cache for SQL statements. If this parameter is set to **true**, the Cache service is used. If this parameter is set to **false**, the cross-statement cache is not used. Parameter type: POSTMASTER. **Range**: Boolean. **Default Value**: **false**.
staging_folder_expire_time	Integer	**Definition**: Specifies the interval for automatically clearing residual temporary files in ORC and Parquet tables. By default, the residual files are automatically cleared seven days later. **Range**: An int64 ranging from 12 hours to 1 year. **Default Value**: **604800**.
obs_result_format	Integer	**Definition**: Specifies the format of the result set file written to OBS and whether to enable the result set file compression function. Parameter type: USERSET. **Range** ● **0**: The result set file is in JSON format and is not compressed. ● **1**: The result set file is in JSON format and is compressed using the zstd algorithm. ● **2**: The result set file is in ARROW format and is not compressed. ● **3**: The result set file is in ARROW format and is compressed using the zstd algorithm. **Default Value**: **0**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
resource_track_level	Enum	**Definition**: Specifies the type of information reported in the **query_plan** field of SQL monitoring data. Currently, this parameter takes effect only for the **SELECT**, **INSERT**, **DELETE**, **UPDATE**, and **CREATE TABLE AS** statements. Parameter type: USERSET. **Range**: Enumerated values. • **query**: The **query_plan** field in the SQL monitoring data reports the explain information. • **perf**: The **query_plan** field in the SQL monitoring data reports the explain performance information. • **operator**: Collects and reports operator-level monitoring information in real time based on **perf**. **Default Value**: **perf**.
external_partition_pruning_method	Enum	**Definition**: Specifies the ORC and Parquet table partition reading mode. Parameter type: USERSET. **Range**: Enumerated values. • **partition_list**: The partition list is obtained from LakeFormation. Ensure that the partition information on LakeFormation is correct. You can use the **show partitions table_name** syntax to query the partition list. • **partition_folder**: The partition list is obtained from OBS by listing file directories. Ensure that the data directory on OBS is organized in the format of key1=value1/key2=value2. • **url_filter**: The partition list is obtained from LakeFormation, and the query conditions are pushed down to LakeFormation as much as possible. **Default Value**: **partition_list**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
reader_opt_options	String	**Definition**: Specifies the enhanced query filtering option. **Range**: A string. Multiple values can be set at the same time. • **parquet_pushdown**: Supports pushing down **OR** conditions of Parquet tables to the storage layer for filtering. • **iceberg_pushdown**: Supports pushing down **OR** conditions of Iceberg tables to the storage layer for filtering. • **dictionary**: Supports filtering rowgroups of Parquet tables based on dictionary information. • **partition_dynamic_pruning**: Supports dynamic partition pruning. • **remove_duplicate_filter**: Removes redundant Iceberg filter criteria. • **complex_type_pushdown**: Supports pushdown of complex conditions. **Default Value**: **parquet_pushdown**, **iceberg_pushdown**, **partition_dynamic_pruning**, **dictionary**, **complex_type_pushdown**, **remove_duplicate_filter**, **page_index**, **parquet_native_decoder**, **dict_filter_row**, or **balance_file_size**.
log_min_messages	Enum	**Definition**: Controls the level of messages to be written to the server log. Parameter type: USERSET. **Range**: Enumerated values. Options: **debug5**, **debug4**, **debug3**, **debug2**, **debug1**, **info**, **notice**, **warning**, **error**, **log**, **fatal**, and **panic**. Each level includes all subsequent levels. The higher the level, the fewer messages are sent to the log. For example, if the log level in **log_min_messages** is after **error**, only **log**, **fatal**, and **panic** logs will be printed to the server log file. **Default Value**: **warning**.
enable_topsql_to_collector	Boolean	**Definition**: Specifies whether to start the background resident thread for collecting and reporting performance metrics. **Range**: Boolean. • **on** or **true**: Enables the feature. • **off** or **false**: Disables the feature. **Default Value**: **false**.

 Copyright © Huawei Technologies Co., Ltd.

14.4.2 Locale and Formatting

The display format, time zone, character encoding, and currency and numeric locale formats vary depending on service scenarios. A unified format setting helps ensure the accuracy and readability of data display. This section describes GUC parameters related to the display format, including DateStyle, IntervalStyle, TimeZone, client_encoding, and lc_*.

Parameter Description

Table 14-5 Locale and formatting parameters

Parameter	Type	Description
DateStyle	String	**Definition**: Specifies the display format for date and time values, and the rules for interpreting ambiguous date input values. This parameter consists of two independent parts: the output format model (ISO, Postgres, SQL, or German) and the input/output order of year/month/day (DMY, MDY, or YMD). The two parts can be set separately or together. The keywords **Euro** and **European** are synonyms for **DMY**. The keywords **US**, **NonEuro**, and **NonEuropean** are synonyms for **MDY**. The ISO format is recommended. Postgres, SQL, and German use abbreviations to represent time zones, such as **EST**, **WST**, and **CST**. **NOTE** **gs_initdb** initializes this parameter to the same value as that of **lc_time**. **Range**: A string. **Default Value**: **ISO, MDY**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
IntervalStyle	Enum	**Definition**: Specifies the display format of interval values. **Range**: Enumerated values. • **sql_standard** generates output compliant with the SQL standard. • **postgres** indicates that the output is formatted to match PostgreSQL 8.4 when the DateStyle parameter is set to **ISO**. • **postgres_verbose** indicates that the output is formatted to match PostgreSQL 8.4 when the DateStyle parameter is set to **non_ISO**. • **iso_8601** indicates that the output is formatted to match the "format and code" defined in ISO 8601. • **oracle** outputs results corresponding to the numtodsinterval function in Oracle databases. Refer to **numtodsinterval** for details. **NOTE** The **IntervalStyle** parameter also affects the interpretation of ambiguous interval input. **Default Value**: **postgres**.
TimeZone	String	**Definition**: Specifies the time zone used for displaying and interpreting time stamps. **Range**: A string. **Default Value**: **GMT**.
timezone_abbreviations	String	**Definition** Specifies the list of time zone abbreviations that are accepted by the server. **Range**: A string. **Default Value**: **Default**. **NOTE** **Default** indicates the abbreviation for the general time zone. However, there are other abbreviations such as **'Australia'** and **'India'** that are used to define specific installations.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
extra_float_digits	Integer	**Definition**: Specifies the number of digits displayed for floating-point values, including float4, float8, and geometric data types. The parameter value is added to the standard number of digits (**FLT_DIG** or **DBL_DIG** as appropriate). **Range**: -15 to 3. **NOTE** • Setting it to **3** includes some significant data bits. This is particularly useful for dumping floating-point data that needs precise restoration. • Setting it to a negative value discards insignificant data bits. **Default Value**: **0**.
client_encoding	String	**Definition**: Specifies the character encoding type on the client. Set this parameter based on the frontend services. Keeping the client and server encodings consistent improves efficiency. **Range**: All character encoding types compatible with PostgreSQL. **UTF8** indicates that the database's character encoding type is used. **NOTE** • Run the **locale -a** command to view the locales and corresponding encoding formats supported by the current system and configure accordingly. • By default, **gs_initdb** initializes this parameter based on the current system environment. You can run the **locale** command to view the current configuration environment. • You are advised to keep the default value. Do not modify the **client_encoding** parameter directly in **postgresql.conf** using tools like **gs_guc**, as changes will not take effect to maintain consistent internal communication encoding. **Default Value**: **UTF8**.
lc_monetary	String	**Definition**: Specifies the display format for monetary values. It affects the output of functions such as **to_char**. Acceptable values are system-dependent. **Range**: A string. **NOTE** • Run the **locale -a** command to view the locales and corresponding encoding formats supported by the current system and configure accordingly. • By default, **gs_initdb** initializes this parameter based on the current system environment. You can run the **locale** command to view the current configuration environment. **Default Value**: **en_US.UTF-8**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
lc_numeric	String	**Definition** Specifies the display format for numbers. It affects the output of functions such as **to_char**. Acceptable values are system-dependent. **Range**: A string. **NOTE** • Run the **locale -a** command to view the locales and corresponding encoding formats supported by the current system and configure accordingly. • By default, **gs_initdb** initializes this parameter based on the current system environment. You can run the **locale** command to view the current configuration environment. **Default Value**: **en_US.UTF-8**.
lc_time	String	**Definition**: Specifies the display format of time and region. It affects the output of functions such as **to_char**. Acceptable values are system-dependent. **Range**: A string. **NOTE** • Run the **locale -a** command to view the locales and corresponding encoding formats supported by the current system and configure accordingly. • By default, **gs_initdb** initializes this parameter based on the current system environment. You can run the **locale** command to view the current configuration environment. **Default Value**: **en_US.UTF-8**.

14.5 Lock Management

In Aura, LakeFormation central locks are used to ensure correctness when multiple compute nodes concurrently write to the same table. This section introduces parameters primarily governing the behavior of LakeFormation central locks.

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 14-6 Lock management parameters

Parameter	Type	Description
fq_lflock_wait_timeout	Integer	**Definition**: Controls the maximum waiting time for obtaining a lock. If the waiting time for obtaining a lock exceeds the specified value, the system reports an error. **Range**: An integer ranging from 0 to INT_MAX, in milliseconds. If the value is **0**, the lock can be requested only once. If the request fails, the system reports an error and does not try again. **Default Value**: 30s.
fq_lflock_heartbeat_timeout	Integer	**Definition**: Specifies the maximum timeout interval for lock heartbeat failures. ● To maintain the survival of the LakeFormation central lock, the system continuously sends lock heartbeats to LakeFormation during the lock holding period. When the lock heartbeat fails to be sent, the system does not report an error immediately but retries continuously. If the lock heartbeat still fails to be sent within the maximum timeout interval, the system reports an error. ● The automatic lock release time of LakeFormation is 60s when it does not receive lock heartbeats. In normal cases, the system sends heartbeats to LakeFormation at an interval of 30s. Therefore, if this parameter is set to a value less than 30s, the system reports an error when the first heartbeat fails to be sent. If this parameter is set to a value greater than 60s, the system may lose the lock when it is disconnected from LakeFormation for a long time. If you do not want to lose the lock when the system is disconnected from LakeFormation for a long time, set this parameter to a value less than 60s, or ensure that concurrent writes to the same table are not performed within the interval specified by this parameter. **Range**: An integer ranging from 0 to INT_MAX, in milliseconds. **Default Value**: 5 min.

 Copyright © Huawei Technologies Co., Ltd.

14.6 Python UDF

This section describes GUC parameters for Python UDFs.

Parameter Description

Table 14-7 Python UDF parameters

Parameter	Type	Description
python_udf_spec_options	String	**Definition**: Specifies the default values of common parameters in WITH ARGUMENTS during Python UDF execution. If a parameter is not specified in WITH ARGUMENTS, the GUC value set by **python_udf_spec_options** is used by default. The priority of parameters explicitly specified in WITH ARGUMENTS is higher than that of the GUC value. • **set python_udf_spec_options = '(key1:value1),(key2:value2)'** indicates that the value of a specified key is added or updated based on the existing settings. If a key already exists, the value of the key is overwritten. If the key does not exist, the key is added. • **set python_udf_spec_options = ''** or **reset python_udf_spec_options** indicates that all existing settings are cleared. **Range**: A string, in the format of comma-separated (*key*:*value*) pairs. Currently, six parameters are supported: **timeout**, **cpu**, **memory**, **arch**, **apu**, and **concurrency**. For details, see **WITH ARGUMENTS Syntax**. **Default Value**: Empty ('').
enable_udf_global_actor_pool	Boolean	**Definition**: Specifies whether to enable the global Python UDF actor pool. After this function is enabled, Python UDF actors are managed by the global CN instead of each DN. You need to enable **udf_split** first (by setting **sql_opt_options='on(udf_split)'**). Otherwise, this parameter does not take effect. **Range**: Boolean (**on**/**off**). **Default Value**: **off**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
enable_udf_scheduler	Boolean	**Definition**: Specifies whether to enable the global dynamic scheduling policy for Python UDFs. After this function is enabled, the system dynamically schedules Python UDF actors based on the running time of different Python UDFs in statements, reducing the execution bottleneck of a single UDF. This parameter takes effect only when **udf_split** (by setting **sql_opt_options='on(udf_split)'**) and **enable_udf_global_actor_pool** are enabled and the statement does not contain multi-branch operations such as JOIN. **Range**: Boolean (**on**/**off**). **Default Value**: **off**.
enable_udf_log_trace	Boolean	**Definition**: Specifies whether to enable the Python UDF log traceability function. After this function is enabled, the system automatically injects the SESSIONID/STATEMENTID tag to all logs generated during UDF execution (print, logging, loguru, and warnings). **Range**: Boolean (**on**/**off**). **Default Value**: **on**.

14.7 Vector Search

The recall rate and query latency of vector search are affected by the number of detected partitions, reordering factor, TopK pruning policy, and runtime resources. You need to balance the two factors in your service. This section describes the GUC parameters related to Lance vector search, including query policy parameters (**lance_options**), runtime behavior parameters (**lance_runtime_options**), and runtime cache parameters (**lance_cache_options**). You can adjust these parameters based on the data scale and query performance requirements to improve the recall rate and control the query latency.

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 14-8 Vector search parameters

Parameter	Type	Description
lance_options	String	**Definition**: Sets parameters related to Lance vector retrieval, including the number of probed partitions, reordering factor, and distributed Top-K pruning policy. The parameter format is a comma-separated key-value pair, and each key-value pair is represented in the format of (*key*:*value*). **Range**: Key-value pair string. The value range of each subitem is as follows: • **nprobes**: The value is an integer ranging from 1 to INT_MAX. The default value is **1024**. It indicates the number of IVF partitions to be probed during vector index query. A larger value indicates more partitions to be scanned, higher recall rate, and longer query latency. • **refine_factor**: The value is an integer ranging from 0 to INT_MAX. The default value is **0**. It indicates the reordering factor. The size of the candidate set returned by the index scan is the product of this factor and the Top-K value. The system calculates the exact distance of the candidate set, sorts the candidate set, and obtains the Top-K value. A larger value indicates a higher recall rate and higher compute overhead. • **large_topk_threshold**: The value is an integer ranging from 0 to INT_MAX. The default value is **0**. It indicates the K value threshold for triggering distributed Top-K pruning. When the **LIMIT K** value of a query is greater than or equal to the threshold, the **per_shard_topk** policy is enabled. The CN calculates the number of candidates to be returned by each DN and pushes down the calculation result. If the value is **0**, this policy is disabled. • **global_topk_budget**: The value is a float ranging from 1.0 to 100.0. The default value is **3.0**. It indicates the budget coefficient of distributed Top-K pruning, which is used together with **large_topk_threshold**. Calculation formula: **perShardTopk=ceil(global_topk_budget × K / active_dn_count)**, where **active_dn_count** is the number of active DNs. When **active_dn_count** is **3.0**, **perShardTopk** equals **K**, which is consistent with the recall rate when pruning is not enabled.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
		Default Value: **(nprobes:1024), (refine_factor:0), (large_topk_threshold:0), (global_topk_budget:3.0)**. **Example**: `set lance_options='(nprobes:10), (refine_factor:0), (large_topk_threshold:0), (global_topk_budget:3.0)'`
lance_runtime_options	String	**Definition**: Sets parameters related to the running behavior of Lance, including the number of CPU threads, number of I/O threads, and memory pool size. The parameter format is a comma-separated key-value pair, and each key-value pair is represented in the format of (*key*:*value*). **lance_runtime_options** is a Postmaster-level parameter. It must be configured before the Aura server starts. Modifying this parameter using the **SET** command at runtime is not supported. **Range**: A string of key-value pairs. The value range of each sub-item is as follows: ● **cpu_threads**: Specifies the size of the thread pool for CPU-intensive tasks (such as decompression, encoding, and vector calculation). The value is an integer ranging from 1 to INT_MAX. There is no default value. ● **io_threads**: Controls the degree of parallelism for I/O operations, which is used for the number of concurrent read and write operations on object storage. The value is an integer ranging from 1 to INT_MAX. There is no default value. ● **mem_pool_size**: Specifies the memory pool size (in bytes) of the DataFusion execution engine, which is used for memory management during query execution, especially for operations that support spilling to disks (such as sorting and aggregation). The value is an integer ranging from 1 to INT64_MAX. There is no default value. **Default Value**: None. **Example**: `set lance_runtime_options='(cpu_threads:8), (io_threads:64), (mem_pool_size:1048576)'`

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
lance_cache_options	String	**Definition**: Sets parameters related to the cache of the running Lance, including whether to enable the internal thread pool, thread pool size, thread pool queue size, and read/write timeout interval of the thread pool. The parameter value is a string of key-value pairs separated by commas (,), with each key-value pair in the format of (*key*:*value*). This parameter takes effect only when **enable_remote_cache** is set to **on**. **Range**: A string of key-value pairs. The sub-items are as follows: • **cache_pool_support**: Specifies whether to enable the cache thread pool when Lance is running. The value is an integer, which can be **0** or **1**. The default value is **1**. • **cache_pool_size**: Controls the size of the cache thread pool. The value is an integer ranging from 1 to INT_MAX. The default value is **16**. • **cache_pool_queue_depth**: Controls the queue size of each thread in the cache thread pool. The value is an integer ranging from 1 to INT_MAX. The default value is **4**. • **cache_pool_timeout**: Controls the read/write timeout interval of each thread in the cache thread pool. If the timeout interval is reached, the I/O request of OBS is directly initiated. The value is an integer ranging from 1 to INT_MAX. The default value is **30**. **Default Value**: (**cache_pool_support**: **1**), (**cache_pool_size**: **16**), (**cache_pool_queue_depth**: **4**), (**cache_pool_timeout**: **30**). **Example**: `set lance_cache_options = "(cache_pool_size:32),(cache_pool_queue_depth:8)";`

14.8 Checkpoint-based Fault Recovery

When AI DataLake processes multimodal data, operations such as accessing OBS or calling models during execution may cause access exceptions. You can enable checkpoint-based fault recovery to improve the query success rate.

Notes and Constraints

- Checkpoint-based fault recovery is enabled only for read-only narrow-dependency queries (SELECT) and is disabled for DML statements (INSERT/UPDATE/DELETE) to avoid side effects such as sequence number increment and data modification during retries.

 Copyright © Huawei Technologies Co., Ltd.

- The retry granularity is at the DN level (the files (1 to *n*) processed by the DN are retried as a whole). As a result, duplicate query results may exist.
- Only the query model is supported. The session model is not supported.
- By default, **enable_meta_scan** is enabled for Iceberg tables. The query plan includes data redistribution between DNs, which breaks the narrow dependency of the original statement, causing checkpoint-based fault recovery to not take effect.

Parameter Description

Table 14-9 Checkpoint-based fault recovery parameters

Parameter	Type	Description
checkpoint_resume_max_retries	Integer	**Definition**: Specifies the maximum number of checkpoint-based fault recovery retries on the CN after a DN crashes due to an error. After this function is enabled, the CN automatically intercepts the DN failure, rebuilds the DN, and resumes the execution. The CN retries the execution of *N* files on the DN at the file granularity. In addition, the number of retries is counted independently for each DN. In the multi-DN scenario, the retry of each DN does not affect that of other DNs. This parameter takes effect only for SELECT queries with narrow dependency. **NOTE** Narrow dependency means that each parent partition is used by at most one child partition. **Range**: An integer, in the range [0, 10]. ● **0**: Checkpoint-based fault recovery is disabled. After a DN crashes, the CN directly reports an error to the user and the query is terminated. ● [1, 10]: Checkpoint-based fault recovery is enabled. After a DN actor crashes, the CN automatically intercepts the failure, stops the old actor, creates a new actor, rebinds the connection, and resumes the execution. The query is retried for a maximum of the specified number of times. If the number of retries exceeds the specified value, the query is terminated and the original DN error is reported. **Default Value**: **0**.

Example

Set the maximum number of checkpoint-based fault recovery retries on the CN to **3** after a DN error or crash occurs.

```
SET checkpoint_resume_max_retries = 3;
SELECT udf(col) FROM table;
```

 Copyright © Huawei Technologies Co., Ltd.