DataArtsFabric

开发指南

文档版本 01

发布日期 2025-12-09

华为云计算技术有限公司

商标声明

和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 使用前必读

文档面向的读者对象

数据库开发指南重点面向数据库的设计者、应用程序开发人员或DBA，提供设计、构建、查询和维护数据仓库所需的信息。

作为数据库管理员和应用程序开发人员，至少需要了解以下知识：

- 操作系统知识。这是一切的基础。
- SQL语法。这是操作数据库的必备能力。

使用前提条件

使用本指南前，需要完成以下任务。

- 开通DataArtsFabric SQL服务。
- 使用SQL编辑器打开DataArtsFabric SQL服务。

关于上述任务的详细指导，请参见DataArtsFabric SQL服务《用户指南》。

SQL 语法文本格式约定

为了方便对语法使用的理解，在文档中对SQL语法文本按如下格式进行表述。

格式	意义
大写	语法关键字（语句中保持不变、必须照输的部分）采用大写表示。
小写	参数（语句中必须由实际值进行替代的部分）采用小写表示。
[ ]	表示用“[ ]”括起来的部分是可选的。
...	表示前面的元素可重复出现。
[x | y | ...]	表示从两个或多个选项中选取一个或者不选。
{ x | y | ... }	表示从两个或多个选项中选取一个。

格式	意义
[x | y | ...] [...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用空格分隔。
[x | y | ...] [,...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用逗号分隔。
{ x | y | ... } [...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间以空格分隔。
{ x | y | ... } [,...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间用逗号分隔。

2 设计数据库对象

2.1 了解 DataArtsFabricSQL 数据库对象

DataArtsFabric SQL是云原生数据库引擎，能让您借助LakeFormation（数据湖统一元数据管理引擎）和OBS（对象存储服务）的能力进行高效数据管理。

在DataArtsFabric SQL中，数据对象的创建、管理、销毁都与LakeFormation关联，LakeFormation为OBS中的数据提供了持久化的元数据存储。您可以使用DataArtsFabric SQL，快速，方便地操作LakeFormation的元数据。

了解更多LakeFormation，请参见**湖仓构建LakeFormation**。

DataArtsFabric SQL中数据库对象和LakeFormation数据对象具有一定的映射关系，如下所示：

表 2-1 DataArtsFabric SQL 与 LakeFormation 数据对象映射关系

DataArtsFabric SQL数据库对象	LakeFormation数据库对象	说明
Database	Catalog	DataArtsFabric SQL中，用户登录Database后，会自动连接到LakeFormation指定Catalog，用户不可以跨Catalog进行元数据访问。
Schema	Database	DataArtsFabric SQL中的Schema对应了LakeFormation的Database。
Table	Table	DataArtsFabric SQL和LakeFormation的表级映射关系是一致的。
IAM User	IAM User	DataArtsFabric SQL仅支持IAM用户访问，暂不支持其他类型用户访问。

说明

- DataArtsFabric SQL的Database对象无法创建、修改或删除。用户登录DataArtsFabric SQL后，自动连接到LakeFormation指定Catalog上，用户可以通过SQL语法便捷、快速地操作LakeFormation元数据和OBS数据。
- DataArtsFabric SQL当前仅支持IAM用户访问。

2.2 创建和管理 Schema

Schema又称作模式，从逻辑上组织一个数据库中的对象和数据。通过管理Schema，允许多个用户使用同一数据库而不相互干扰，同时便于将第三方应用添加到相应的Schema下而不引起冲突。

相同的数据库对象名称可以应用在同一数据库的不同Schema中，而没有冲突。例如，a_schema和b_schema都可以包含名为mytable的表。具有所需权限的用户可以访问数据库的多个Schema中的对象。

创建 Schema

使用CREATE SCHEMA命令来创建一个新的Schema。

```
CREATE SCHEMA myschema;
```

如果需要在模式中创建或者访问对象，其完整的对象名称由模式名称和具体的对象名称组成。中间由符号“.”隔开。例如：myschema.table。

设置 Schema 搜索路径

GUC参数search_path设置Schema的搜索顺序，参数取值形式为采用逗号分隔的Schema名称列表。如果创建对象时未指定目标Schema，则该对象会被添加到搜索路径中列出的第一个Schema中。当不同Schema中存在同名的对象时，查询对象未指定Schema的情况下，将从搜索路径中包含该对象的第一个Schema中返回对象。

- 使用SHOW命令查看当前搜索路径。

  ```
  SHOW SEARCH_PATH;
  ```

  search_path参数的默认值为：default_db。DataArtsFabric SQL会自动帮您创建一个默认Schema。
- 使用SET命令修改当前会话的默认Schema。例如，将搜索路径设置为myschema1、myschema2，首先搜索myschema1。

  ```
  SET SEARCH_PATH TO myschema1, myschema2;
  ```

使用 Schema

在特定Schema下创建对象或者访问特定Schema下的对象，需要使用有Schema修饰的对象名。名称包含Schema名以及对象名，之间用“.”号分开。

- 在myschema下创建mytable表。以schema_name.table_name格式创建表。

  ```
  CREATE TABLE myschema.mytable(id int, name varchar(20)) STORE AS ORC;
  ```
- 查询myschema下mytable表的所有数据。

  ```
  SELECT * FROM myschema.mytable;
   id | name
  ----+------
  (0 rows)
  ```

查看 Schema

- 使用current_schema函数查看当前Schema：
```
SHOW current_schema;
 current_schema
----------------

(1 row)
```
- 要查看所有Schema的列表，请执行。
```
SHOW SCHEMAS;
```
- 要查看某Schema详细信息，请执行。
```
DESCRIBE SCHEMA myschema;
```

Schema 的权限控制

默认情况下，用户只能访问属于自己的Schema中的数据库对象。如需要访问其他Schema的对象，则需赋予对应Schema的usage权限。

当前DataArtsFabric SQL不支持通过SQL执行权限相关操作。您可以通过LakeFormation界面执行权限相关操作，具体操作方法可以参见**LakeFormation数据权限概述**。

删除 Schema

- 使用DROP SCHEMA命令删除一个空的Schema（即该Schema下没有数据库对象）。
```
DROP SCHEMA IF EXISTS myschema;
```
- 默认情况下，删除一个Schema前，它必须为空。 要删除一个Schema及其包含的所有对象（表、数据、函数等），需要使用CASCADE关键字。
```
DROP SCHEMA myschema CASCADE;
```

2.3 创建和管理表

创建表

CREATE TABLE命令创建一个表，创建表时可以定义以下内容：

- 表的列及数据类型。
- 表分布的定义，即表的分布策略，它决定DataArtsFabric SQL数据库如何在片（Segment）之间划分数据。
- 表存储格式。
- 分区表定义。

示例：CREATE TABLE创建了一个表web_returns_p1，并以ORC文件格式存储数据。

```
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;
```

表类型

DataArtsFabric SQL支持两种表类型：托管表（Managed Table）和外表（External Table）。

类型	描述	适用场景
托管表（Managed Table）	数据和元数据均由DataArtsFabric SQL管理，用户可通过DataArtsFabric SQL管理元数据及数据文件。删除表时，表的数据和元数据均会被删除。表的数据路径必须为并行文件系统。	● 需要频繁更新的业务数据。 ● 需要防误删保障的关键数据。
外表（External Table）	DataArtsFabric SQL通过元数据或表模式读取指定位置的数据文件，用户不可通过DataArtsFabric SQL变更实际数据文件。删除表时，仅删除表的元数据，数据文件不会受到影响。	● 只读分析。 ● 基于文件路径的细粒度访问权限控制。

注意

对于托管表（Managed Table），数据文件仍存储在常规的文件系统（OBS并行文件系统）中，用户可以在不告知DataArtsFabric SQL的情况下对其进行变更。如果用户这样做，则违反了DataArtsFabric SQL对于托管表的期望和约定，可能会造成未定义的错误。

Managed Table与External Table使用示例

```
-- 创建Schema
CREATE SCHEMA sales_schema WITH LOCATION 'obs://bucket/catalog1/sales_schema/';

-- 创建托管表
CREATE TABLE region_sales_info(id int, item varchar(128), sale_date date) partition by (region
varchar(128)) store as parquet;

-- 创建外表指向托管表
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/';

-- 由托管表导入业务数据
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)

-- 外表初始无分区信息，无法查询到数据
SELECT * FROM readonly_region_sales_info;
 id | item | sale_date | region
----+------+-----------+--------
(0 rows)
```

```
-- 修复外表分区信息
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)

-- 删除外表 不会影响数据文件
DROP TABLE readonly_region_sales_info;
SELECT COUNT(1) from region_sales_info;
 count
-------
     4
(1 row)

-- 删除托管表，数据同时被删除
DROP TABLE region_sales_info;
```

表分布的定义

DataArtsFabric SQL支持的分布方式：哈希表（Hash）和轮询表（Roundrobin）。

策略	描述	适用场景	优势与劣势
哈希表（Hash）	数据以hash方式组织到n个桶（文件）中。	数据量较大的事实表。	• 在读数据时可以有效利用文件hash信息进行剪枝过滤，多表join场景下，相同hash的文件会发送到相同节点执行，加快计算速度。
轮询表（Roundrobin）	数据随机地组织到多个文件中。	数据量较大的事实表，且使用Hash分布时找不到合适的分布列。	• Roundrobin优点是保证了数据不会发生倾斜，从而提高了集群的空间利用率。 • 一般在大表无法找到合适的分布列时，定义为Roundrobin表，如果大表能够找到合适的分布列，优先选择性能更好的Hash分布。

选择分布列

采用Hash分布方式，需要为用户表指定一个分布列（distribute key）。当插入一条记录时，系统会根据分布列的值进行hash运算后，将数据存储在对应的DN中。

所以Hash分布列选取至关重要，需要满足以下原则：

1. **列值应比较离散**。例如，考虑选择表的主键为分布列，如在人员信息表中选择身份证号码为分布列。
2. **在满足第一条原则的情况下尽量不要选取存在常量filter的列**。例如，表dwcjk相关的部分查询中出现dwcjk的列zqdh存在常量的约束(例如zqdh=’000001’)，那么就应当尽量不用zqdh做分布列。
3. **在满足前两条原则的情况，考虑选择查询中的连接条件为分布列**，以便Join任务能够下推到DN中执行，且减少DN之间的通信数据量。

对于Hash分表策略，如果分布列选择不当，可能导致数据倾斜，查询时出现部分DN的I/O短板，从而影响整体查询性能。因此在采用Hash分表策略之后需对表的数据进行数据倾斜性检查，以确保数据在各个DN上是均匀分布的。

查看表数据

- 查询当前Schema下所有表的信息。
```
SHOW TABLES;
```
- 查询表的属性。
```
DESCRIBE  web_returns_p1;
```
- 执行如下命令查询表web_returns_p1的数据量。
```
SELECT count(*) FROM web_returns_p1;
```
- 执行如下命令查询表web_returns_p1的所有数据。
```
SELECT * FROM web_returns_p1;
```
- 执行如下命令只查询字段c_customer_sk的数据。
```
SELECT c_customer_sk FROM web_returns_p1;
```
- 执行如下命令过滤字段c_customer_sk的重复数据。
```
SELECT DISTINCT( c_customer_sk ) FROM web_returns_p1;
```
- 执行如下命令查询字段c_customer_sk为3869的所有数据。
```
SELECT * FROM web_returns_p1 WHERE c_customer_sk = 3869;
```
- 执行如下命令按照字段c_customer_sk进行排序。
```
SELECT * FROM web_returns_p1 ORDER BY c_customer_sk;
```

删除表数据

在使用表的过程中，可能需要删除表数据。

- 当前不支持通过DELETE命令删除数据。
- 如果执行如下命令，会删除表中所有的行。
```
TRUNCATE TABLE customer_t1;
```
- 删除创建的表。
```
DROP TABLE customer_t1;
```

2.4 表分区定义

分区表就是把逻辑上的一张表根据分区策略分成几张物理块库进行存储，这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。当进行条件查询时，系统只会扫描满足条件的分区，避免全表扫描，从而提升查询性能。

分区表的优势：

- 改善查询性能。对分区对象的查询可以仅搜索自己关心的分区，提高检索效率。
- 增强可用性。如果分区表的某个分区出现故障，表在其他分区的数据仍然可用。
- 提升可维护性。对于需要周期性删除的过期历史数据，可以通过drop/truncate分区的方式快速高效处理。

分区策略选择

当表有以下特征时，可以考虑使用表分区策略：

- 数据具有明显枚举类型字段。

  分区表需要根据有明显枚举类型字段进行表分区。例如按照日期、区域、数值等字段进行分区，时间字段是最常见的分区字段。

- 表数据量比较大。

  小表扫描本身耗时不大，分区表的性能收益不明显，因此只建议对大表采取分区策略。

对已有的表进行分区

表只能在创建时声明为分区表。 例如：

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

删除一个分区

使用ALTER TABLE语句从分区表中删除一个分区。例如，删除表my_tbl的分区。

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

查询分区

- 查询分区b = 10。

  ```
  SELECT * FROM my_tbl  where b = 10;
  ```

- 查看分区表信息。

  ```
  SHOW PARTITIONS my_tbl;
  DESCRIBE my_tbl;
  ```

删除分区表

使用DROP TABLE语句删除一个分区表。

```
DROP TABLE web_returns_p1;
```

3 查询数据

3.1 单表查询

示例表：

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

简单查询

通过SELECT ... FROM ... 语句从数据库中获取结果。

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

对结果进行筛选

通过WHERE语句对查询的结果进行过滤，找到想要查询的部分。

```
SELECT * FROM newproducts WHERE category='toys';
 product_id | product_name | category | quantity
```

```
------------+--------------+----------+----------
       1601 | telescope    | toys     |       80
       1666 | Frisbee      | toys     |      244
(2 rows)
```

对结果进行排序

使用ORDER BY语句可以让查询结果按照期望的方式进行排序。

```
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)
```

限制结果查询数量

如果需要查询只返回部分结果，可以使用LIMIT语句限制查询结果返回的记录数。

```
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)
```

聚合查询

可以通过使用GROUP BY语句配合聚合函数，构建一个聚合查询来关注数据的整体情况。

```
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)
```

3.2 多表连接查询

连接类型介绍

通过SQL完成各种复杂的查询，多表之间的连接是必不可少的。连接分为：内连接和外连接两大类，每大类中还可进行细分。

- 内连接：标准内连接（INNER JOIN），交叉连接（CROSS JOIN）和自然连接（NATURAL JOIN）。

- 外连接：左外连接（LEFT OUTER JOIN），右外连接（RIGHT OUTER JOIN）和全外连接（FULL JOIN）。

为了能更好的说明各种连接之间的区别，下面通过具体示例进行详细的阐述。

创建示例表student和math_score，并插入数据，设置enable_fast_query_shipping为off（默认为on）即查询优化器使用分布式框架；参数explain_perf_mode为pretty（默认值为pretty）指定explain的显示格式。

```
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);

SET enable_fast_query_shipping = off;
SET explain_perf_mode = pretty;
```

内连接

- 标准内连接（INNER JOIN）

  语法：

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

  说明：表示left_table和right_table中满足join_condition的行拼接在一起作为结果输出，不满足条件的元组不会输出。

  示例1：查询学生的数学成绩。

  ```
  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
    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)
```

- 交叉连接（CROSS JOIN）

  语法：

```
left_table CROSS JOIN right_table
```

  说明：表示left_table中所有行和right_table中的所有行分别进行连接，最终结果行数等于两边行数的乘积。又称笛卡尔积。

  示例2：学生表和数学成绩表的交叉连接。

```
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

 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)
```

- 自然连接（NATURAL JOIN）

  语法：

```
left_table NATURAL JOIN right_table
```

  说明：表示left_table和right_table中列名相同的列进行等值连接，且自动将同名列只保留一份。

  示例3：学生表和数学成绩表的自然连接，两表的同名列为id列，将按照id列进行等值连接。

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

外连接

- 左外连接（LEFT JOIN）

  语法：

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

说明：左外连接的结果集包括left_table的所有行，而不仅是连接列所匹配的行。如果left_table的某行在right_table中没有匹配行，则在相关联的结果集行中输出right_table的列均为空值。

示例4：学生表和数学成绩表进行左外连接，学生表中id为3的行在结果集中对应的右表数据用NULL填充。

```
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)
```

- 右外连接（RIGHT JOIN）

  语法：

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

  说明：与左外连接相反，右外连接的结果集包括right_table的所有行，而不仅是连接列所匹配的行。如果right_table的某行在left_table中没有匹配行，则在相关联的结果集行中left_table的列均为空值。

  示例5：学生表和数学成绩表进行右外连接，数学成绩表中id为6的行在结果集中对应的左表数据用NULL填充。

```
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)
```

对于右外连接，在join算子中却显示的是Left。这是因为，右外连接其实就是将左右表进行交互后的左外连接，所以数据库内部实现为了减少处理逻辑，会将右外连接转为左外连接。

- 全外连接（FULL JOIN）

  语法：

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

  说明：全外连接是左外连接和右外连接的综合。全外连接的结果集包括left_table和right_table的所有行，而不仅是连接列所匹配的行。如果left_table的某行在right_table中没有匹配行，则在相关联的结果集行中right_table的列均为空值。如果right_table的某行在left_table中没有匹配行，则在相关联的结果集行中left_table的列均为空值。

  示例6：学生表和数学成绩表进行全外连接，学生表中id为3的行在结果集中对应的右表数据用NULL填充，数学成绩表中id为6的行在结果集中对应的左表数据用NULL填充。

  ```
  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)
```

多表查询中 on 条件和 where 条件的区别

从上面各种连接语法中可见，除自然连接和交叉连接外，其他都需要有on条件（using在查询解析过程中会被转为on条件）来限制两表连接的结果。通常在查询的语句中也都会有where条件限制查询结果。这里说的on连接条件和where过滤条件是指不含可以下推到表上的过滤条件。on和where的区别是：

- on条件是两表连接的约束条件。
- where是对两表连接后产生的结果集再次进行过滤。

简单总结就是：on条件优先于where条件，在两表进行连接时被应用；生成两表连接结果集后，再应用where条件。

3.3 WITH 表达式

WITH表达式用于定义在大型查询中使用的辅助语句，这些辅助语句通常被称为公共表达式或CTE（即common table expr），可以理解为一个带名称的子查询，之后该子查询可以以其名称在查询中被多次引用。

WITH表达式中的辅助语句可以是SELECT，并且WITH子句本身也可以被附加到一个主语句中，主语句可以是SELECT、INSERT或DELETE。

WITH 中的 SELECT

在WITH子句中使用SELECT的相关信息。

语法格式

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

其中，with_query的语法为：

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

注意

- 显示指定MATERIALIZED时，将子查询执行一次，并将其结果集进行物化；指定NOT MATERIALIZED时，则将其子查询替换到主查询中的引用处。
- 每个CTE的AS语句指定的SQL语句，必须是可以返回查询结果的语句，目前只支持SELECT查询语句，暂不支持INSERT、UPDATE、DELETE、VALUES等其它数据修改语句。
- 单个WITH表达式表示一个SQL语句块中的CTE定义，可以同时定义多个CTE，每个CTE可以指定列名，也可以默认使用查询输出列的别名。例如：
  ```
  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;
  ```
  该语句中定义了两个CTE，s1和s2，其中s1指定了列名为a，b，s2未指定列名，则列名为输出列名x，y。
- 每个CTE可以在主查询中引用0次、1次或多次。
- 同一个语句块中不能出现同名的CTE，但不同语句块中可以出现同名的CTE，此时，语句中引用的CTE则是距离引用位置最近的语句块中的CTE。
- 由于SQL语句中可能包含多个SQL语句块，每个语句块都可以包含一个WITH表达式，每个WITH表达式中的CTE可以在当前语句块、当前语句块的后续CTE中，以及子层语句块中引用，但不能在父层语句块中引用。由于每个CTE的定义也是个语句块，因此也支持在该语句块中定义WITH表达式。

WITH中SELECT的基本价值是将复杂的查询分解称为简单的部分。示例如下：

```
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;
```

WITH子句定义了两个辅助语句regional_sales和top_regions，其中regional_sales的输出用在top_regions中而top_regions的输出用在主SELECT查询。这个例子可以不用WITH来书写，但是就必须要用两层嵌套的子SELECT，使得查询更长更难以维护。

3.4 SQL on Iceberg

3.4.1 Iceberg 简介

Iceberg是一个面向海量数据分析的开放表格式，是元数据和数据文件的一种组织方式，处于计算引擎和存储系统之间，旨在提供一种可扩展且可靠的方式来管理海量的数据表格。Iceberg的设计目标是提供一个可扩展、高性能、易于使用的表格管理解决方案，以满足现代分布式数据处理的需要。

本特性仅25.3.0及以上版本支持。

主要特点

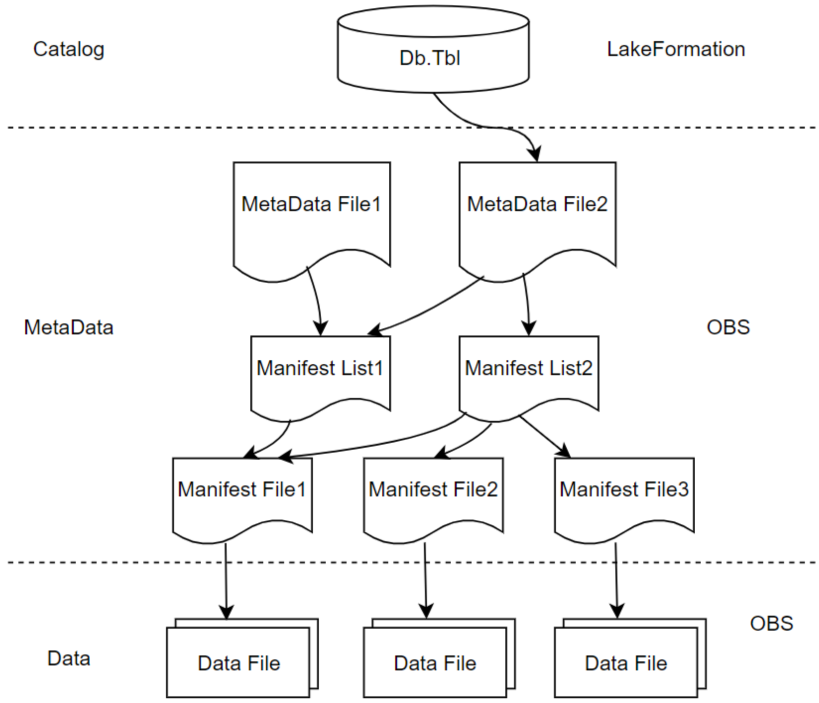
- **可扩展性**：Iceberg可轻松扩展到支持海量的数据表格和庞大的数据集合。
- **查询性能**：Iceberg的元数据管理和查询优化功能可以提高数据查询的性能。
- **数据版本管理**：Iceberg提供了可靠的数据版本管理功能，可以帮助用户对数据进行版本控制和回溯。
- **易于使用**：Iceberg提供了简单易用的API和命令行工具，使得用户可以轻松地创建、管理和查询数据表格。
- **灵活的分区策略**：Iceberg支持灵活的分区策略，可以根据不同的数据集合进行分区管理。
- **多版本数据支持**：Iceberg支持多版本数据，可以帮助用户对数据进行版本管理和回溯。
- **多种数据格式支持**：Iceberg支持多种数据格式，包括Parquet、ORC、Avro等。

基本概念

- Table（表格）：Iceberg的最基本的概念是Table，它是一个数据表格的抽象表示。Table包含了表格的元数据信息、数据存储位置、分区策略等信息。
- Partition（分区）：Partition是将Table中的数据按照指定的规则划分为多个子集的过程。Partition可以基于数据的某些特征进行划分，例如按照时间、地理位置、产品类型等进行分区。
- Metadata（元数据）：Iceberg的元数据是指描述Table中的数据结构、分区策略、数据版本等信息的数据。元数据存储在持久化的存储介质中，例如HDFS、S3等。
- Snapshot（快照）：Snapshot是指Table在某个时间点上的数据视图，它包含了Table当前版本的数据和元数据信息。
- Manifest（清单）：Manifest是指Table中数据文件的清单列表，它包含了每个数据文件的元数据信息（例如文件路径、大小、分区信息等）。

文件组织方式

如下图所示，Iceberg将数据分为元数据管理层、数据存储层。



- 元数据层：
  - metadata文件为json格式。存储当前版本的元数据信息，所有快照信息。
  - manifest list文件，即snapshot文件或清单列表文件，为avro格式。一次commit生成一个快照文件，每行存储一个manifest file的路径、其存储的数据文件的分区范围，增加删除了几个数据文件等信息，在查询时提供过滤信息，加快速度。
  - manifest文件，为avro格式。存储多个数据文件的信息列表，每行是一个数据文件的详细描述，包括状态、路径、分区信息、列级别的统计信息（最大最小值、空值数等）、文件大小以及文件里数据行数等。其中列级别的统计信息在扫描表数据时可过滤掉不必要的文件。
- 数据存储层：支持不同的文件格式，包括parquet、orc、avro。

3.4.2 Catalog 介绍

Iceberg Catalog是Iceberg的顶层组件，负责管理所有Iceberg表的元数据和元数据操作。 Catalog管理表的架构和元数据，提供了创建、查询和修改表的接口，是用户进行Iceberg表操作的入口点。用户可以通过它找到每个表当前元数据文件的位置，是读取和写入Iceberg表的关键组件。

当前DataArtsFabric SQL版本支持使用Hadoop Catalog作为Iceberg表的Catalog组件。

Hadoop Catalog

Hadoop Catalog不依赖外部系统，可以使用任意文件系统，它将当前表元数据文件路径记录在一个文件目录下。

因为Hadoop支持存算分离，底层的数据文件可以是HDFS或者是OBS对象系统等。

对Hadoop Catalog来讲，定位一个表的位置，只需要提供表的路径即可，因为表的元信息都存储在文件中。

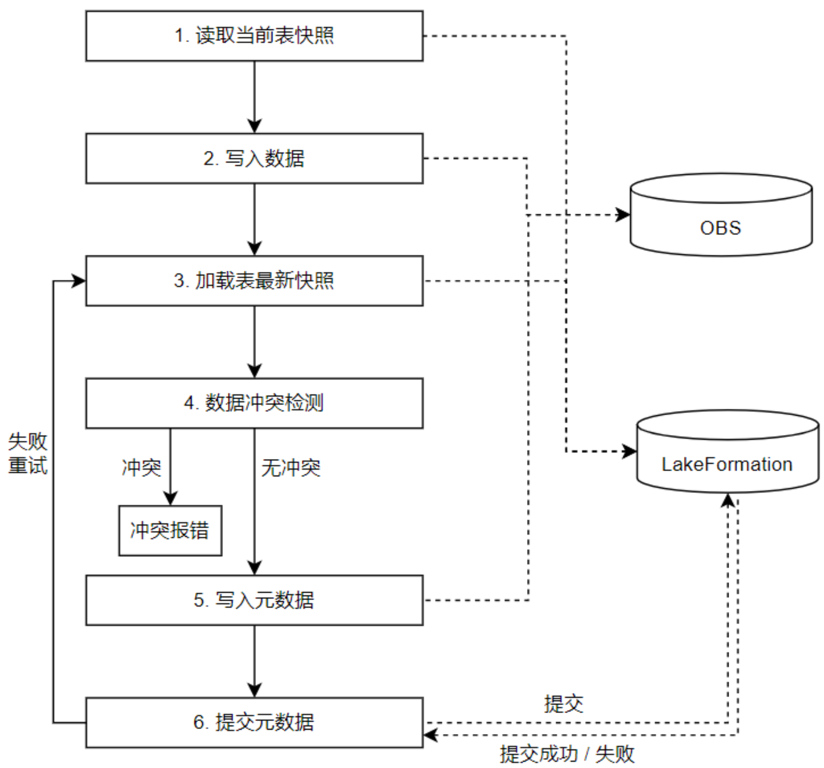
LakeFormation Catalog

LakeFormation Catalog依赖LakeFormation元数据服务，由LakeFormation管理最新的快照信息。

LakeFormation Catalog采用乐观锁（OCC）机制，保证了多租场景下并发写的数据一致性。

当前在DataArtsFabric SQL上创建的Iceberg表均为LakeFormation Catalog表。

下图简述了LakeFormation Catalog的处理并发提交流程。



1. 从LakeFormation上读取当前表快照信息，包括最新的MetaData File Path、SnapshotId等。
2. 依据当前快照，写入新的数据。
3. 加载最新快照。
4. 检测数据冲突，如果发生数据冲突，则语句执行失败，否则尝试事务提交。
5. 写入元数据，包括Manifest Files、Manifest List和MetaData File。
6. 将最新的MetaData File Path和SnapshotId等提交给LF。如果提交失败，则从步骤3重试。

3.4.3 使用 Iceberg 前准备

设置 GUC

本版本新增GUC参数**enable_meta_scan**，用于优化查询的性能。

该参数默认打开，但在表数据量很小的情况下，关闭可能比打开的时候查询性能更高，请基于实际情况打开或关闭该GUC。

代码示例如下：

```
SHOW enable_meta_scan;
SET enable_meta_scan=off;
SET enable_meta_scan=on;
或
SET enable_meta_scan=true;
SET enable_meta_scan=false;
```

3.4.4 创建、清空、删除 Iceberg 表

Iceberg表将数据存储在OBS上。DataArtsFabric SQL可以直接访问OBS上的Iceberg数据。

创建 Iceberg 表

通过CREATE TABLE语法创建Iceberg表。与其他格式的表相比，创建Iceberg表没有特别的参数需要指定，只需要指定STORE AS ICEBERG即可。具体语法请参见**CREATE TABLE**。

示例：

```
CREATE TABLE iceberg_ext(
col1 int,
col2 varchar(20),
...
)PARTITION BY (col3 bigint)
TABLEPROPERTIES (
'write.metadata.delete-after-commit.enabled'='true',
'write.metadata.previous-versions-max' = '10'
) STORE AS ICEBERG;
```

其中write.metadata.delete-after-commit.enabled控制是否在提交事务后删除旧版本元数据文件，默认情况为false。true的情况下，可以通过write.metadata.previous-versions-max设定保留多少个metadata。

使用这两个参数需要谨慎，虽然删除元数据文件可以节省存储空间，但需要确保不会影响数据的一致性和可用性。

清空 Iceberg 表

通过TRUNCATE TABLE语法清空表中的数据，具体语法请参见**TRUNCATE**。

示例：

```
TRUNCATE TABLE iceberg_ext;
```

删除 Iceberg 表

通过DROP TABLE语法删除Iceberg表，会同时删除表的元数据及数据（Managed表删除后可以从LakeFormation控制台恢复表的元数据及数据）。具体语法请参见**DROP TABLE**。

示例：

```
DROP TABLE iceberg_ext;
```

3.4.5 修改 Iceberg 表

在数据仓库管理中，用户经常需要对Iceberg表进行结构上的调整，如添加新列、修改现有列或调整表属性等，以适应业务变化。然而，如果操作不当，可能会导致数据不一致或性能下降的问题。如何确保在调整表结构时既高效又安全？目前支持对Iceberg表做以下操作：添加列、修改列、删除列、修改表属性、恢复表属性默认值，具体语法可参考**ALTER TABLE**。通过这些功能，用户可以灵活地管理Iceberg表，同时保证数据的完整性和系统的稳定性。

修改表中的列

通过ALTER TABLE语法的ADD COLUMN、RENAME COLUMN TO、ALTER COLUMN、DROP COLUMNS能力，可以对iceberg表的列进行加列、删列、重命名列、更新列和重新排序列等schema evolution操作。

须知

iceberg通过唯一编号fieldId为列进行编号，在写入数据时会将fieldId写入文件的schema中，在读取文件的schema时通过fieldId将元数据的列与文件中的列映射起来，能够实现schema evolution后的读取。对于非iceberg写入的文件，schema中不存在fieldId，读取文件时会以列名为依据进行映射，进行schema evolution操作后，如果存在复用列名的情况，可能会导致类型不匹配而读取失败或者读取到了以前的数据。

示例：

建iceberg分区表：

```
CREATE TABLE t_iceberg(a int, b array<struct<x:int, y:int>>) partition by (c int) store as iceberg;
```

- 添加列：

  iceberg表支持添加一列或者多列到表中或者复杂类型字段中，需要指定列名和类型，可以指定注释和位置。当添加到复杂类型字段中时，不能指定注释。

  ```
  -- 添加到表中
  alter table t_iceberg add column d int, e float comment 'this is e', f array<int> after a, g
  struct<x:int,y:int> comment 'this is g' before b;
  -- 添加到复杂类型字段中，需要使用element来指定array的子类型
  alter table t_iceberg add column b.element.z int after x, b.element.a before z;
  ```

- 重命名列：

  iceberg表支持对表列或者复杂类型字段进行重命名。

  ```
  alter table t_iceberg rename column a to newa;
  alter table t_iceberg rename column b.element.x to newx;
  ```

- 更新列：

  iceberg表支持更新表列的类型、注释和位置，支持更新复杂类型字段的类型和位置。

  ```
  alter table t_iceberg alter column a type bigint comment 'this is a' after c, c type bigint comment 'this is c' after a, b.element.y type bigint after first;
  ```

  当前仅支持从低精度类型更新到高精度类型：

  - 整数类型：从smallint更新到int或者bigint，从int更新到bigint。
  - 浮点类型：从float4更新到float8。
  - numeric类型：从低precision更新到高precision，不支持更新scale。
  - 字符串类型：varchar或者char的长度变长。

> ⚠ **注意**
>
> 浮点类型的存储是不精确的，从float4更新到float8后，数据的小数部分精度会变化，结果会发生变化。

- 删除列

  iceberg支持删除表列或者复杂类型字段。不能删除分区列，不能删除表的所有列或者复杂类型的所有字段。

  ```
  alter table t_iceberg drop columns (a, b.x);
  ```

修改表属性

通过ALTER TABLE语法的SET/UNSET TABLEPROPERTIES能力，可以对表的属性值进行修改。

示例：

```
ALTER TABLE iceberg_ext SET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled' = 'false');
ALTER TABLE iceberg_ext UNSET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled');
```

3.4.6 查询和写入 Iceberg 表

查询 Iceberg

可以查询Iceberg表中的数据。具体语法可参考**SELECT**。

注意事项：

当前仅支持查询最新全量数据。

示例：

```
SELECT * FROM iceberg_ext order by col1;
```

写入 Iceberg 表

可以向Iceberg表中添加一行或多行数据。具体语法可参考**INSERT**。

注意事项：

- 如果频繁插入小数据，可能会出现小文件问题，建议定期重写data/manifest文件以保障存储、查询效率。
- 当前仅支持数据文件以Parquet格式写入。

示例：

```
INSERT INTO iceberg_ext VALUES (1, 'a', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT OVERWRITE INTO iceberg_ext VALUES (1, 'b', 1.1234567, 100.11, '2025-03-03', '2025-03-03
16:00:00');
INSERT INTO iceberg_ext (col1, col2, col3, col4, col5, col6) SELECT col1, col2, col3, col4, col5, col6 FROM
iceberg_table WHERE col1 > 18;
```

删除 Iceberg 表数据

可以对Iceberg表中的一行或多行数据执行删除操作。具体语法可参考**DELETE**。

注意事项：

- 当前仅支持merge-on-read方式的delete操作。
- 如果有并发删除、更新时，delete时建议指定分区条件以减少并发冲突。
- 如果对分区表做整分区删除时，delete会通过标记元数据状态来加速删除操作，是否可以走优化可以通过explain delete查看关键字“Do speed up for delete”，如果可以走删除优化，删除返回的行数可能大于实际删除的行数。

示例：

```
DELETE FROM iceberg_ext where a = 1;
```

更新 Iceberg 表数据

可以对Iceberg表中的一行或多行数据执行更新操作。具体语法可参考**UPDATE**。

注意事项：

- 当前仅支持merge-on-read方式的delete操作。
- 如果有并发删除、更新时，update时建议指定分区条件以减少并发冲突。

示例：

```
UPDATE iceberg_ext set b = 1 where a = 1;
```

3.4.7 表达式分区

在处理大规模数据集时，数据分区是提高查询性能的关键技术之一。然而，传统的分区方式需要用户手动定义分区逻辑，这不仅增加了用户的负担，还可能导致分区策略不够灵活。表达式分区用于支持Iceberg的隐藏分区功能，其核心设计是将“分区逻辑”从用户视角隐藏，交给Iceberg元数据自动管理。如何确保数据分区既高效又灵活？通过使用表达式分区，用户可以无需关心复杂的分区逻辑，Iceberg将自动根据数据特征和查询模式优化分区策略，从而提高查询性能和数据管理效率。

约束限制

- 最多支持4个分区列。
- 不支持void表达式。
- 所有分区表达式不支持复杂数据类型。
- truncate/bucket表达式不支持二进制数据。
- INSERT INTO PARTITION功能仅支持identity表达式，不支持其他表达式。
- Iceberg约束：同一个表列，不能出现在year/month/day/hour两个时间表达式中。 但是，可以同时出现在时间和bucket/truncate表达式中。
- Iceberg约束：对分区键有默认命名规则，不允许与表的列名称与之冲突。
- 当前不支持分区演进功能。

支持的分区表达式

FabricSQL要支持分区表达式以及对应数据类型列表如下：

表 3-1 FabricSQL 分区支持的表达式

表达式名称	支持的数据类型	计算结果的类型	说明
identity	smallint, int, bigint, long, numeric, date, timestamp, text, varchar, bpchar	原数据类型	不支持float，double，boolean, bytea。
bucket[N]	smallint, int, bigint, long, numeric， date, timestamp text, bpchar, varchar	int	• 先计算输出数值的hash值，再求模。 计算结果为 [0, N)范围之内。 • 哈希算法采用的是MurmurHash3算法。
truncate[W]	smallint, int, bigint, long,numeric， text, bpchar, varchar	原数据类型	• 不支持bytea类型。 • 详情参见**分区表达式：truncate**。
year	date, timestamp	int	例：year('2025-10-10') 计算结果为 '2025'年。
month	date, timestamp	int	例：month('2025-10-10') 计算结果为'2025-10'月。
day	date, timestamp	int	例：day('2025-10-10') 计算结果为'2025-10-10'日。
hour	timestamp	int	例：hour('2025-10-10 10:10:10.000') 计算结果为'2025-10-10 10:00:00'时。
void	不支持	不支持	本质上是将所有数值映射到null值上。

所有表达式对于null值的处理结果必须返回null值。

identity表达式对输入的数值不做任何处理，直接输出原数值。

year表达式计算的是相对1970年、以年为单位的数值。month表达式计算的是相对1970-01-01、以月为单位的数值。day表达式计算的是相对1970-01-01、以天为单位的数值。hour表达式计算的是相对1970-01-01 00:00:00、以小时为单位的数值。Iceberg在记录分区值结果时，并不会直接记录整数数值，而是将它转换为可阅读的时间字符串。例如，Iceberg会将year('2025-10-10') 表达式计算出来的分区值记录为'2025'年，而不是整数55。

FabricSQL提供了SQL函数fq_iceberg_year()、fq_iceberg_month()、fq_iceberg_day()、fq_iceberg_hour()、fq_iceberg_truncate()、fq_iceberg_bucket()用于实现对应分区表达式year、month、day、hour、truncate、bucket的计算功能。

分区表达式：truncate

对于不同的数据类型，truncate表达式的计算方法不同，可以使用函数fq_iceberg_truncate() 进行运算求值。

对于整数类型，truncate(W, integer_value)返回的是不超过integer_value、且为W倍数的最大数值。 示例如下：

```
fabricsql=>select fq_iceberg_truncate (10, -2) ;
 fq_iceberg_truncate
---------------------
                 -10
(1 row)

fabricsql=> select fq_iceberg_truncate (10, 0) ;
 fq_iceberg_truncate
---------------------
                   0
(1 row)

fabricsql=> select fq_iceberg_truncate (10, 9) ;
 fq_iceberg_truncate
---------------------
                   0
(1 row)

fabricsql=> select fq_iceberg_truncate (10, 10) ;
 fq_iceberg_truncate
---------------------
                  10
(1 row)

fabricsql=> select fq_iceberg_truncate (10, 19) ;
 fq_iceberg_truncate
---------------------
                  10
(1 row)
```

对于字符串类型，truncate(W, string_value)返回的是string_value字符串中前缀长度为W的前缀子字符串。注意，长度以字符为单位。 示例如下：

```
fabricsql=> select *, length(t)  from fq_iceberg_truncate (2, '') as t;
 iceberg_truncate | length
------------------+--------
                  |      0
(1 row)

fabricsql=> select *, length(t)  from fq_iceberg_truncate (2, 'a') as t;
 iceberg_truncate | length
------------------+--------
 a                |      1
(1 row)

fabricsql=> select *, length(t)  from fq_iceberg_truncate (2, 'tt') as t;
 iceberg_truncate | length
------------------+--------
 tt               |      2
(1 row)

fabricsql=> select *, length(t)  from fq_iceberg_truncate (2, 'cow') as t;
 iceberg_truncate | length
------------------+--------
 co               |      2
(1 row)

fabricsql=> select *, length(t)  from fq_iceberg_truncate (2, '你好中国!') as t;
 iceberg_truncate | length
------------------+--------
 你好             |      2
(1 row)
```

对于numeric类型，truncate(W, num_value)返回的是不超过num_value、且小数点右移scale位后为W倍数的最大数值。示例如下：

```
fabricsql=> select fq_iceberg_truncate (50, 10.64::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.600
(1 row)

fabricsql=> select fq_iceberg_truncate (50, 10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.650
(1 row)

fabricsql=> select fq_iceberg_truncate (50, -10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
             -10.700
(1 row)
```

创建表达式分区表

CREATE TABLE语法详情参考**CREATE TABLE**。

创建一个表达式分区的Iceberg managed表SQL语句如下：

```
fabricsql=> set current_schema to test;
fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
CREATE TABLE
fabricsql=> 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)
```

在示例中， c3是一个identity分区键。此外，还创建了bucket、truncate、year表达式的三个分区键。Iceberg会对分区键进行内部命名，其命名规则如下表所述。

表 3-2 不同表达式分区键在 Iceberg 内部的默认命名规则

表达式	默认命名规则	样例
year(source_column)	<source_column>_year	year(ts) -> 'ts_year'
month(source_column)	<source_column>_month	month(ts) -> 'ts_month'
day(source_column)	<source_column>_day	day(ts) -> 'ts_day'
hour(source_column)	<source_column>_hour	hour(ts) -> 'ts_hour'
bucket(N, source_column)	<source_column>_bucket	bucket(ts) -> 'ts_bucket'
truncate(W, source_column)	<source_column>_trunc	truncate(ts) -> 'ts_trunc'
identity(source_column)	<source_column>	identity(ts) -> 'ts'

表的列名称不能够与分区键的名称冲突，否则会报错、建表失败。

```
fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val),  c3 int comment 'this is c3 field', year(ts), ts_year bigint ) store as iceberg;
ERROR:  column "ts_year" specified more than once
```

Iceberg不允许时间类型的同一列同时出现在year、month、day、hour多个表达式中，但是允许该列出现在truncate、bucket其他表达式中。

```
fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val),  c3 int comment 'this is c3 field', year(ts), month(ts) ) store as iceberg;
ERROR:  java.lang.IllegalArgumentException: Cannot add redundant partition: 1002: ts_year: year(4)
conflicts with 1003: ts_month: month(4).

fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val),  c3 int comment 'this is c3 field', year(ts), bucket(7, ts) ) store as iceberg;
CREATE TABLE
```

FabricSQL支持创建分区键的最大数量是4，否则会报错、建表失败。

```
fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
ERROR:  partitioned table should have no more than 4 partition keys
```

FabricSQL不支持将复杂类型作为分区键，否则会报错、建表失败。

```
fabricsql=> CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric)
partition by (bucket(16, val), truncate(10, id), c3 struct<c: text, d: int>, year(ts) ) store as iceberg;
ERROR:  Complex type cannot be served as a value-partitioning column.

fabricsql=> CREATE TABLE iceberg_table(a int , b struct<c: text, d: int>) PARTITION BY(bucket(5, b.c),
truncate(3, b.d)) store as ICEBERG;
ERROR:  syntax error at or near "."
LINE 1: ... struct<c: text, d: int>) PARTITION BY(bucket(5, b.c), trunc...
                                                              ^
```

FabricSQL支持创建表达式分区的Iceberg外表，需要满足额外条件：

- 与OBS上Iceberg表的分区键数量相同。
- 与OBS上Iceberg表的分区键定义完全一致，包括分区键的顺序、分区键的表达式、分区键操作的表列。

后续小节将以iceberg_test_part_expr表为样例，其表定义如下：

```
fabricsql=> set current_schema to test;
fabricsql=> CREATE TABLE iceberg_test_part_expr(id bigint, data text, category text, ts date) partition by (c3
int, truncate(4, data), year(ts)) store as iceberg;
CREATE TABLE
```

导入数据

实现表达式分区的Iceberg表支持数据的单条插入和批量导入功能。

```
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'he',           'shanxi', '2024-09-29', 20);
INSERT 1
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'helloworld',   'shanxi', '2023-06-29', 6);
INSERT 1
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'hella',        'shanxi', '2022-05-29', 60);
INSERT 1
fabricsql=> INSERT INTO iceberg_test_part_expr values(9, 'zhangmingmin', 'shanxi', '2025-04-29', 100);
INSERT 1
```

对于INSERT INTO PARTITION功能，FabricSQL仅支持所有分区键为identity类型的表，并不支持其他类型表达式的分区表。


```
fabricsql=> INSERT INTO iceberg_test_part_expr PARTITION(c3=1, data='hell', ts='2025') values (9,
'hellohello',   'shanxi', '2025-08-29 01:05:59.550463') ;
ERROR:  INSERT INTO PARTITION clause only support identity partition.
```

可以使用函数pg_get_partitions()来查看表的所有分区数据，示例如下：

```
fabricsql=> select partition_name from pg_get_partitions('iceberg_test_part_expr');
     partition_name
--------------------------
 c3=1/data=hell/ts=2025
 c3=20/data=he/ts=2024
 c3=100/data=zhan/ts=2025
 c3=6/data=hell/ts=2023
 c3=60/data=hell/ts=2022
(5 rows)
```

基于列表达式的查询优化

在支持表达式分区之后，FabricSQL可下推列表达式，从而进行静态分区剪枝的查询优化。 示例如下：

```
fabricsql=> explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr
where c3 = 1;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 id |                              operation                              | A-rows
----+------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                     |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr          |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      2

          Predicate Information (identified by plan id)
 ------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (c3 = 1)
        Pushdown Predicate Filter: (c3 = 1)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Filter: (c3 = 1)
        Pushdown Predicate Filter: (c3 = 1)
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 2)
        DN read from: direct
(19 rows)
```

可以看到， c3=1列表达式下推后，扫描的分区数量由6个裁减为2个。

基于 SQL 函数的查询优化

在实现表达式分区之后，FabricSQL支持白名单SQL函数的下推优化。当前主要支持的SQL函数白名单如下：

表 3-3 支持下推优化的 SQL 函数列表

分区表达式	SQL函数	下推条件
year	year(), date_trunc()	date_trunc()的时间单位是year。
month	date_trunc()	date_trunc()的时间单位是month。

分区表达式	SQL函数	下推条件
day	date_trunc()	date_trunc()的时间单位是day。
hour	date_trunc()	date_trunc()的时间单位是hour。
truncate	left(), substring()	substring()必须从字符串头开始算起。

字符串函数left()和substring()下推进行静态分区剪枝的示例如下：

```
fabricsql=> explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr
where substring(data, 1, 3) > 'hello';
                                    QUERY PLAN
------------------------------------------------------------------------------------------
 id |                              operation                              | A-rows
----+-------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                        |      1
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr         |      1
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      1

           Predicate Information (identified by plan id)
 --------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: ("substring"(data, 1, 3) > 'hello'::text)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 1)
        DN read from: direct
(16 rows)

fabricsql=> explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr
where left(data, 5) < 'hello';
                                    QUERY PLAN
------------------------------------------------------------------------------------------
 id |                              operation                              | A-rows
----+-------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                        |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr         |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      5

           Predicate Information (identified by plan id)
 --------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: ("left"(data, 5) < 'hello'::text)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 5)
        DN read from: direct
(16 rows)
```

可以看到，substring()函数下推后，分区数量由6个裁减到1个。 而left()函数下推后，分区数量由6个裁减到5个。

时间类型函数date_trunc()下推进行静态分区剪枝的示例如下：

```
fabricsql=> explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr
where date_trunc('year', ts) > '2023-12-30 00:00:00';
```

```
                                    QUERY PLAN
----------------------------------------------------------------------------------------------------
 id |                          operation                          | A-rows
----+-----------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                              |      4
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr         |      4
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      4

                     Predicate Information (identified by plan id)
----------------------------------------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (date_trunc('year'::text, ts) > '2023-12-30 00:00:00'::timestamp without time zone)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 4)
        DN read from: direct
(16 rows)

fabricsql=> explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr
where year(ts) < 2024;
                                QUERY PLAN
------------------------------------------------------------------------------------------
 id |                          operation                          | A-rows
----+-----------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                              |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr         |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      2

          Predicate Information (identified by plan id)
--------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (year(ts) < 2024)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 2)
        DN read from: direct
(16 rows)
```

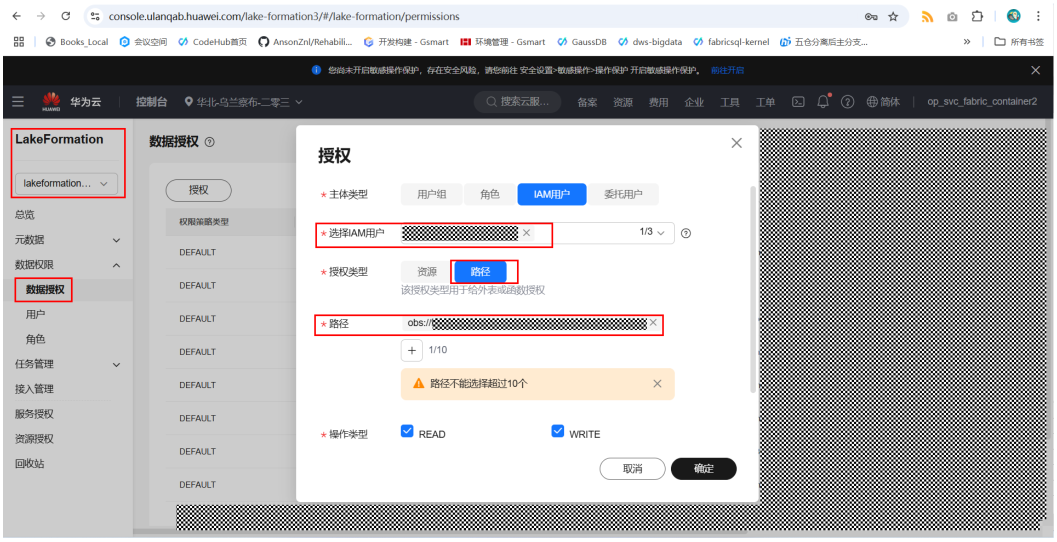
可以看到， date_trunc()函数下推后，分区数量由6个裁减到4个。 而year()函数下推后，分区数量由6个裁减到2个。

3.4.8 访问第三方引擎创建的 Iceberg 表

对于Spark产生的Iceberg表，DataArtsFabric SQL通过外表方式访问OBS上的Iceberg数据。

步骤1 获得对应OBS路径的访问权限。

登录华为云LakeFormation控制台，左侧选择“数据权限 > 数据授权”，单击“授权”，将文件所在路径的READ/WRITE授权给当前IAM用户。



步骤2 创建EXTERNAL TABLE。

```
CREATE EXTERNAL TABLE table_name [column_name type_name,...] location 'fullLocation' store as iceberg;
```

其中fullLocation是obs上Iceberg表所在路径的完整路径，如果指定列名与列类型，则需要列名与列类型与文件层面的元数据相同，如果不指定，则使用文件层面Iceberg元数据列名与列类型建表，ICEBERG文件层面数据类型与DataArtsFabric数据类型对应关系与需要指定的建表类型如下表所示。

表 3-4 DataArtsFabric 数据类型与 ICEBERG 数据类型匹配关系

类型名称	ICEBERG对应数据类型	DataArtsFabric对应建表类型
2字节整数	INTEGER	INTEGER
4字节整数	INTEGER	INTEGER
8字节整数	LONG	BIGINT
单精度浮点数	FLOAT	FLOAT4
双精度浮点数	DOUBLE	FLOAT8
科学数据类型	DECIMAL	DECIMAL[p,s]
日期类型	DATE	DATE
时间类型	TIMESTAMP	TIMESTAMP
BOOLEAN类型	BOOLEAN	BOOLEAN
Char类型	STRING	TEXT
VARCHAR类型	STRING	TEXT
字符串(文本大对象)	STRING	TEXT
数组类型	LIST	ARRAY<...>
STRUCT类型	STRUCT	STRUCT<...>

----结束

3.4.9 Iceberg 表服务

DataArtsFabric SQL提供一系列表服务，用户可以根据需要使用，例如清理旧快照，整理元数据、数据文件等，以提高存储、查询的效率。具体语法可参考**ICEBERG表服务函数**。

示例：

- 清理旧快照

```
select * from iceberg_expire_snapshots('iceberg_test','2025-05-15 14:12:00+08');
deleted_data_files_count | deleted_position_delete_files_count | deleted_equality_delete_files_count |
deleted_manifest_files_count | deleted_manifest_lists_count | deleted_statistics_files_count
--------------------------+----------------------------------+-------------------------------------
+-------------------------------+------------------------------+--------------------------------
0                              | 0                                          |
0                                           | 0                                         |
3                                  | 0
(1 row)
```

- 重写元数据文件

```
SELECT * from iceberg_rewrite_manifests('test1');
rewritten_manifests_count | added_manifests_count
--------------------------+----------------------
2                             | 1
(1 row)
```

- 清除孤立文件

```
SELECT * from iceberg_remove_orphanfiles('iceberg_test');
orphan_file_location
------------------------------------------------------------
obs://xxxx/test.txt
obs://xxxx/test2.txt
(2 rows)
```

- 重写数据文件

```
OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('rewrite-all' = 'true') WHERE TABLE_SC =
'Hainan';
INSERT 2
```

3.4.10 如何处理 Iceberg 表并发冲突

3.4.10.1 并发数据修改冲突

在并发写入Iceberg表时，通常会遇到两类冲突：并发修改数据冲突、乐观锁并发提交冲突。本文介绍并发数据修改冲突。

当出现以下并发作业时，可能会产生数据冲突，从而导致作业失败：

- 对同一分区数据并发执行DML操作，例如对同一分区执行Update/Delete/Insert Overwrite/Insert操作。
- Compaction与DML操作并发执行。

下文描述当前DataArtsFabric SQL支持的Iceberg操作类型可能会遇到的并发数据冲突。

- 当执行Insert Overwrite操作时，可能会出现以下冲突：
  - data数据冲突：如果为分区表，在检测数据冲突时，识别到insert overwrite涉及的分区有其他操作提交后产生的新的data文件，则检测为数据冲突发

生；如果为非分区表，在检测数据冲突时，识别到有任何其他操作提交后产生的新的data文件，则检测为数据冲突发生。

  – delete数据冲突：如果为分区表，在检测数据冲突时，识别到insert overwrite涉及的分区有其他操作删除了data文件、新增了delete文件，则检测为数据冲突发生；如果为非分区表，在检测数据冲突时，识别到有其他操作删除了data文件、新增了delete文件，则检测为数据冲突发生。

- 当执行Delete操作时，可能会出现以下冲突：

  data数据冲突：在检测数据冲突时，识别到delete file涉及的data file被其他操作删除了，则检测为数据冲突发生。

- 当执行Update操作时，可能会出现以下冲突：
  – data数据冲突：在检测数据冲突时，识别到delete file涉及的data file被其他操作删除了，则检测为数据冲突发生。
  – delete数据冲突：在检测数据冲突时，识别到有其他操作对update操作涉及的分区产生了新的delete file，则检测为数据冲突发生。

表 3-5 并发冲突矩阵

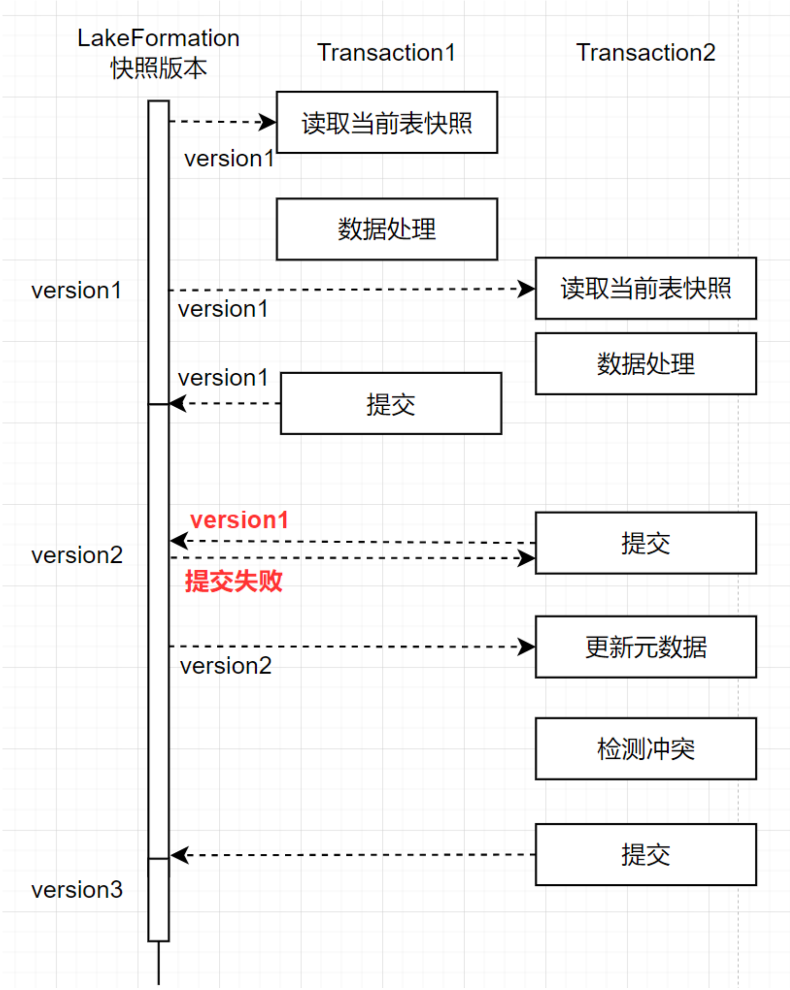
本次操作/已提交操作	insert	insert overwrite	delete
insert	无冲突	无冲突	无冲突
insert overwrite	可能会发生冲突	可能会发生冲突	可能会发生冲突
delete	无冲突	可能会发生冲突	无冲突
update	无冲突	可能会发生冲突	可能会发生冲突

说明

上述并发冲突矩阵是指，当本次操作开始后、提交前，有其他操作开始并完成了提交操作，则本次操作提交时是否会发生冲突。

3.4.10.2 乐观锁并发提交冲突

乐观锁并发提交冲突的产生原因：多个修改作业（包括Insert、Update、Delete、MergeInto以及表服务操作）同时在LakeFormation上提交导致的并发提交冲突。例如：



上图中，Transaction1和Transaction2分别为两个DML事务，Transaction1基于元数据版本version1进行数据处理和提交，并将元数据版本更新至version2，而Transaction2在Transaction1提交前已经基于元数据版本version1开始了数据处理，在首次提交时，LakeFormation拒绝提交，因为此时元数据已经更新为version2，需要由Transaction2更新元数据后，再次提交。

在这种场景下，如果并发量非常大，或者某个事务做冲突检测的时间过久，则会陷入“提交冲突->更新元数据->检测冲突->再次提交”的死循环，导致业务冲突报错。

为解决上述问题，Iceberg引入四个参数。

表 3-6 参数说明

参数名	说明	高并发场景下建议值	表服务场景下建议值
commit.retry.num-retries	最大重试次数	10	4
commit.retry.min-wait-ms	提交失败后，最小提交等待时间	100	1000
commit.retry.max-wait-ms	提交失败后，最大提交等待时间	10000	60000

参数名	说明	高并发场景下建议值	表服务场景下建议值
commit.retry.total-timeout-ms	提交过程最长执行时间。	1800000	1800000

4 JDBC 二次开发

4.1 开发规范

如果用户在APP的开发中，使用了连接池机制，那么需要遵循如下规范：

- 如果在连接中设置了GUC参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。
- 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

否则，连接池里面的连接就是有状态的，会对用户后续使用连接池进行操作的正确性带来影响。

4.2 获取 JDBC 驱动

您可以通过以下方式获取Java SDK。

使用Maven中央仓库和Maven工程下载安装Java SDK。

在Maven项目中添加以下依赖项到pom.xml文件：

```
<dependency>
    <groupId>com.huaweicloud.dws</groupId>
    <artifactId>huaweicloud-dws-jdbc</artifactId>
    <version>8.5.0.101</version>
</dependency>
```

4.3 基于 JDBC 开发

JDBC（Java Database Connectivity，java数据库连接）是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问接口，应用程序可基于它操作数据。DataArtsFabric SQL库提供了对JDBC 4.0特性的支持，需要使用JDK1.8及以上版本编译程序代码，不支持JDBC桥接ODBC方式。

4.3.1 JDBC 包与驱动类

JDBC 包

从管理控制台下载包名为dws_8.x.x_jdbc_driver.zip。

解压后有两个JDBC的驱动jar包：

- gsjdbc4.jar：与PostgreSQL保持兼容的驱动包，其中类名、类结构与PostgreSQL驱动完全一致，曾经运行于PostgreSQL的应用程序可以直接移植到当前系统使用。
- gsjdbc200.jar：如果同一JVM进程内需要同时访问PostgreSQL及DataArtsFabric SQL请使用此驱动包，它的主类名为“com.huawei.gauss200.jdbc.Driver”（即将“org.postgresql”替换为“com.huawei.gauss200.jdbc”）

驱动类

在创建数据库连接之前，需要加载数据库驱动类“org.postgresql.Driver”（对应包gsjdbc4.jar）或者“com.huawei.gauss200.jdbc.Driver”（对应gsjdbc200.jar）。

说明

由于DataArtsFabric SQL在JDBC的使用上与PG的使用方法保持兼容，所以同时在同一进程内使用两个JDBC的驱动的时候，可能会类名冲突。

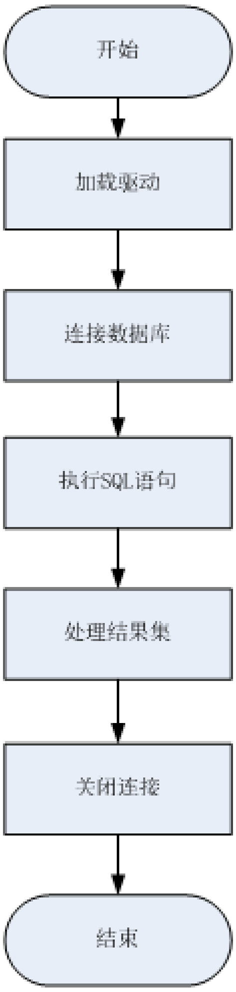
4.3.2 开发流程

在Java数据库连接（JDBC）开发中，遵循一定的流程可以帮助您高效地实现数据库操作。

一个基本的JDBC开发流程，通常包括建立数据库连接、执行SQL查询或更新、处理结果、关闭连接等关键步骤。

采用JDBC开发应用程序的流程图示例如下：

图 **4-1** 采用 JDBC 开发应用程序的流程



4.3.3 加载驱动

在创建数据库连接之前，需要先加载数据库驱动程序。

加载驱动有两种方法：

- 在代码中创建连接之前任意位置隐含装载：Class.forName("org.postgresql.Driver");
- 在JVM启动时参数传递：java -Djdbc.drivers=org.postgresql.Driver jdbctest

说明

- 上述jdbctest为测试用例程序的名称。
- 当使用gsjdbc200.jar时，上面的Driver类名相应修改为"com.huawei.gauss200.jdbc.Driver"

4.3.4 连接数据库

在创建数据库连接之后，才能使用它来执行SQL语句操作数据。

函数原型

JDBC提供了三个方法，用于创建数据库连接。

- DriverManager.getConnection(String url);
- DriverManager.getConnection(String url, Properties info);
- DriverManager.getConnection(String url, String user, String password);

参数

表 4-1 数据库连接参数

参数	描述
url	数据库连接描述符。格式如下： ● jdbc:fabricsql://fabric-ep.endpoint/catalog **说明** ● catalog为lakeformation中对应catalog名称。 ● fabric-ep.endpoint为DataArtsFabric服务终端节点。 ● 使用JDBC连接集群时集群链接地址只支持指定jdbc连接参数，不支持增加变量参数。
info	数据库连接属性。常用的属性如下： ● AccessKeyID：String类型。表示创建连接的用户访问密钥ID。 ● SecretAccessKey：String类型。表示创建连接的用户访问密钥。 ● securityToken：String类型，当使用临时AKSK访问时需要该参数。 **说明** 使用临时AKSK获取连接时，可能因为临时AKSK失效，导致连接中断。建议仅在测试场景使用。 ● workspaceId：String类型。当前环境所属workspace的ID。 ● endpointId：String类型。当前使用SQL端点的ID。 ● lakeformation_instance_id：String类型。用户所使用lakeformation实例ID。

示例

```
//以下用例以gsjdbc4.jar为例，如果要使用gsjdbc200.jar，请替换驱动类名（将代码中的“org.postgresql”替换成“com.huawei.gauss200.jdbc”）
//以下代码将获取数据库连接操作封装为一个接口，可通过给定用户名和密码来连接数据库。

public static Connection GetConnection() {
        //驱动类。
        String driver = "org.postgresql.Driver";
        //数据库连接描述符。
        String sourceURL = "jdbc:fabricsql://10.10.0.13:443/fabricsql_default";
        Connection conn = null;
        Properties properties = new Properties();

        try {
            //加载驱动。
            Class.forName(driver);
        } catch (ClassNotFoundException e ){
```

```
            e.printStackTrace();
            return null;
        }

        try {
            properties.setProperty("workspaceId", "");
            properties.setProperty("endpointId", "");
            properties.setProperty("lakeformation_instance_id", "");
            properties.setProperty("AccessKeyID", "");
            properties.setProperty("SecretAccessKey", "");
             //创建连接。
            conn = DriverManager.getConnection(sourceURL, properties);
            System.out.println("Connection succeed!");
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }
```

4.3.5 执行 SQL 语句

执行普通 SQL 语句

应用程序通过执行SQL语句来操作数据库的数据（不用传递参数的语句），需要按以下步骤执行：

步骤1 调用Connection的createStatement方法创建语句对象。

```
Statement stmt = con.createStatement();
```

步骤2 调用Statement的executeUpdate方法执行SQL语句。

```
stmt.executeUpdate("create table if not exists test_01(" +
    "p_partkey int4," +
    "p_name text," +
    "P_MFGR char(25)," +
    "P_BRAND char(25)," +
    "p_type text," +
    "p_size int4," +
    "P_CONTAINER char(10)," +
    "p_retailprice decimal(15,2)," +
    "p_comment text)" +
    "store as orc;");
```

步骤3 关闭语句对象。

```
stmt.close();
```

----结束

执行预编译 SQL 语句

预编译语句是只编译和优化一次，然后可以通过设置不同的参数值多次使用。由于已经预先编译好，后续使用会减少执行时间。因此，如果多次执行一条语句，请选择使用预编译语句。可以按以下步骤执行：

步骤1 调用Connection的prepareStatement方法创建预编译语句对象。

```
PreparedStatement pstmt = con.prepareStatement("select * from test_01 where p_partkey = ?");
```

步骤2 调用PreparedStatement的setShort设置参数。

```
pstmt.setString(1, "1502");
```

步骤3 调用PreparedStatement的executeUpdate方法执行预编译SQL语句。

```
ResultSet resultSet = pstmt.executeQuery();
```

步骤4 调用PreparedStatement的close方法关闭预编译语句对象。

```
pstmt.close();
```

----结束

执行批处理

用一条预处理语句处理多条相似的数据，数据库只创建一次执行计划，节省了语句的编译和优化时间。可以按如下步骤执行：

步骤1 调用Connection的prepareStatement方法创建预编译语句对象。

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO test_01 VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?);");
```

步骤2 针对每条数据都要调用setShort设置参数，以及调用addBatch确认该条设置完毕。

```
preparedStatement.setInt(1, 1);
preparedStatement.setString(2, "name");
preparedStatement.setString(3, "cata");
preparedStatement.setString(4, "test4");
preparedStatement.setString(5, "test5");
preparedStatement.setInt(6, 2);
preparedStatement.setString(7, "test7");
preparedStatement.setBigDecimal(8, new BigDecimal(502.1));
preparedStatement.setString(9, "test9");
preparedStatement.addBatch();
```

步骤3 调用PreparedStatement的executeBatch方法执行批处理。

```
int[] rowcount = pstmt.executeBatch();
```

步骤4 调用PreparedStatement的close方法关闭预编译语句对象。

```
pstmt.close();
```

----结束

4.3.6 处理结果集

在结果集中定位

ResultSet对象具有指向其当前数据行的光标。最初，光标被置于第一行之前。next方法将光标移动到下一行；因为该方法在ResultSet对象没有下一行时返回false，所以可以在while循环中使用它来迭代结果集。但对于可滚动的结果集，JDBC驱动程序提供更多的定位方法，使ResultSet指向特定的行。定位方法如**表4-2**所示。

表 4-2 在结果集中定位的方法

方法	描述
next()	把ResultSet向下移动一行。
previous()	把ResultSet向上移动一行。
beforeFirst()	把ResultSet定位到第一行之前。
afterLast()	把ResultSet定位到最后一行之后。
first()	把ResultSet定位到第一行。
last()	把ResultSet定位到最后一行。

方法	描述
absolute(int)	把ResultSet移动到参数指定的行数。

获取结果集中光标的位置

对于可滚动的结果集，可能会调用定位方法来改变光标的位置。JDBC驱动程序提供了获取结果集中光标所处位置的方法。获取光标位置的方法如**表4-3**所示。

表 4-3 获取结果集光标的位置

方法	描述
isFirst()	是否在一行。
isLast()	是否在最后一行。
isBeforeFirst()	是否在第一行之前。
isAfterLast()	是否在最后一行之后。
getRow()	获取当前在第几行。

获取结果集中的数据

ResultSet对象提供了丰富的方法，以获取结果集中的数据。获取数据常用的方法如**表4-4**所示，其他方法请参考JDK官方文档。

表 4-4 ResultSet 对象的常用方法

方法	描述
int getInt(int columnIndex)	按列标获取int型数据。
int getInt(String columnLabel)	按列名获取int型数据。
String getString(int columnIndex)	按列标获取String型数据。
String getString(String columnLabel)	按列名获取String型数据。
Date getDate(int columnIndex)	按列标获取Date型数据
Date getDate(String columnLabel)	按列名获取Date型数据。

4.3.7 关闭连接

在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。

关闭数据库连接可以直接调用其close方法即可。

如：conn.close()

4.3.8 示例：常用操作

以下演示基于JDBC开发的主要步骤。

涉及创建数据库、创建表、插入数据等。

代码示例

此示例将演示如何基于DataArtsFabric SQL提供的JDBC接口开发应用程序。

```
//DBtest.java

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;


public class DBTest {

  //创建数据库连接。
  public static Connection GetConnection() {
    String driver = "org.postgresql.Driver";
    String sourceURL = "jdbc:fabricsql://localhost:443/fabricsql_default";
    Connection conn = null;
    try {
      //加载数据库驱动。
      Class.forName(driver).newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }

    try {
      Properties properties = new Properties();
      properties.setProperty("workspaceId", "");
      properties.setProperty("endpointId", "");
      properties.setProperty("lakeformation_instance_id", "");
      properties.setProperty("AccessKeyID", "");
      properties.setProperty("SecretAccessKey", "");
      //创建数据库连接。
      conn = DriverManager.getConnection(sourceURL, properties);
      System.out.println("Connection succeed!");
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }

    return conn;
  };

  //执行普通SQL语句，创建newproducts表。
  public static void CreateTable(Connection conn) {
    Statement stmt = null;
    try {
      stmt = conn.createStatement();

      //执行普通SQL语句。
      int rc = stmt
          .executeUpdate("CREATE TABLE newproducts(product_id INTEGER,product_name
VARCHAR2(60),category VARCHAR2(60),quantity  INTEGER)STORE AS orc;");

      stmt.close();
    } catch (SQLException e) {
      if (stmt != null) {
        try {
```

```
          stmt.close();
        } catch (SQLException e1) {
          e1.printStackTrace();
        }
      }
      e.printStackTrace();
    }
  }

  //执行预处理语句，批量插入数据。
  public static void BatchInsertData(Connection conn) {
    PreparedStatement pst = null;

    try {
      //生成预处理语句。
      pst = conn.prepareStatement("INSERT INTO newproducts VALUES (?, ?, ?, ?);");
      for (int i = 0; i < 3; i++) {
        //添加参数。
        pst.setInt(1, i + 1);
        pst.setString(2, "name" + i);
        pst.setString(3, "cata" + i);
        pst.setInt(4, i + 1);
        pst.addBatch();
      }
      //执行批处理。
      pst.executeBatch();
      pst.close();
    } catch (SQLException e) {
      if (pst != null) {
        try {
          pst.close();
        } catch (SQLException e1) {
        e1.printStackTrace();
        }
      }
      e.printStackTrace();
    }
  }


  /**
   * 主程序，逐步调用各静态方法。
   * @param args
  */
  public static void main(String[] args) {
    //创建数据库连接。
    Connection conn = GetConnection();

    //创建表。
    CreateTable(conn);

    //批插数据。
    BatchInsertData(conn);

    //执行预编译语句，更新数据。
    ExecPreparedSQL(conn);

    //关闭数据库连接。
    try {
      conn.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }

  }

}
```

4.3.9 JDBC 接口参考

JDBC接口是一套提供给用户的API方法，本节将对部分常用接口做具体描述，如果涉及其他接口可参考JDK1.6（软件包）/JDBC4.0中相关内容。

4.3.9.1 java.sql.Connection

java.sql.Connection是数据库连接接口。

表 4-5 对 java.sql.Connection 接口的支持情况

方法名	返回值类型	支持JDBC 4
close()	void	Yes
commit()	void	Yes
createStatement()	Statement	Yes
getAutoCommit()	boolean	Yes
getClientInfo()	Properties	Yes
getClientInfo(String name)	String	Yes
getTransactionIsolation()	int	Yes
isClosed()	boolean	Yes
isReadOnly()	boolean	Yes
prepareStatement(String sql)	PreparedStatement	Yes
rollback()	void	Yes
setAutoCommit(boolean autoCommit)	void	Yes
setClientInfo(Properties properties)	void	Yes
setClientInfo(String name,String value)	void	Yes

4.3.9.2 java.sql.DatabaseMetaData

java.sql.DatabaseMetaData是数据库对象定义接口。

表 4-6 对 java.sql.DatabaseMetaData 的支持情况

方法名	返回值类型	支持JDBC 4
getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)	ResultSet	Yes
getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)	ResultSet	Yes
getTableTypes()	ResultSet	Yes
getUserName()	String	Yes
isReadOnly()	boolean	Yes
nullsAreSortedHigh()	boolean	Yes
nullsAreSortedLow()	boolean	Yes
nullsAreSortedAtStart()	boolean	Yes
nullsAreSortedAtEnd()	boolean	Yes
getDatabaseProductName()	String	Yes
getDatabaseProductVersion()	String	Yes
getDriverName()	String	Yes
getDriverVersion()	String	Yes
getDriverMajorVersion()	int	Yes
getDriverMinorVersion()	int	Yes
usesLocalFiles()	boolean	Yes
usesLocalFilePerTable()	boolean	Yes
supportsMixedCaseIdentifiers()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
supportsMixedCaseQuotedIdentifiers()	boolean	Yes
storesUpperCaseQuotedIdentifiers()	boolean	Yes
storesLowerCaseQuotedIdentifiers()	boolean	Yes

方法名	返回值类型	支持JDBC 4
storesMixedCaseQuotedIdentifiers()	boolean	Yes
supportsAlterTableWithAddColumn()	boolean	Yes
supportsAlterTableWithDropColumn()	boolean	Yes
supportsColumnAliasing()	boolean	Yes
nullPlusNonNullIsNull()	boolean	Yes
supportsConvert()	boolean	Yes
supportsConvert(int fromType, int toType)	boolean	Yes
supportsTableCorrelationNames()	boolean	Yes
supportsDifferentTableCorrelationNames()	boolean	Yes
supportsExpressionsInOrderBy()	boolean	Yes
supportsOrderByUnrelated()	boolean	Yes
supportsGroupBy()	boolean	Yes
supportsGroupByUnrelated()	boolean	Yes
supportsGroupByBeyondSelect()	boolean	Yes
supportsLikeEscapeClause()	boolean	Yes
supportsMultipleResultSets()	boolean	Yes
supportsMultipleTransactions()	boolean	Yes
supportsNonNullableColumns()	boolean	Yes
supportsMinimumSQLGrammar()	boolean	Yes
supportsCoreSQLGrammar()	boolean	Yes
supportsExtendedSQLGrammar()	boolean	Yes
supportsANSI92EntryLevelSQL()	boolean	Yes

方法名	返回值类型	支持JDBC 4
supportsANSI92IntermediateSQL()	boolean	Yes
supportsANSI92FullSQL()	boolean	Yes
supportsIntegrityEnhancementFacility()	boolean	Yes
supportsOuterJoins()	boolean	Yes
supportsFullOuterJoins()	boolean	Yes
supportsLimitedOuterJoins()	boolean	Yes
isCatalogAtStart()	boolean	Yes
supportsSchemasInDataManipulation()	boolean	Yes
supportsSavepoints()	boolean	Yes
supportsResultSetHoldability(int holdability)	boolean	Yes
getResultSetHoldability()	int	Yes
getDatabaseMajorVersion()	int	Yes
getDatabaseMinorVersion()	int	Yes
getJDBCMajorVersion()	int	Yes
getJDBCMinorVersion()	int	Yes

4.3.9.3 java.sql.Driver

java.sql.Driver是数据库驱动接口。

表 4-7 对 java.sql.Driver 的支持情况

方法名	返回值类型	支持JDBC 4
acceptsURL(String url)	boolean	Yes
connect(String url, Properties info)	Connection	Yes
jdbcCompliant()	boolean	Yes
getMajorVersion()	int	Yes
getMinorVersion()	int	Yes

4.3.9.4 java.sql.PreparedStatement

java.sql.PreparedStatement是预处理语句接口。

表 4-8 对 java.sql.PreparedStatement 的支持情况

方法名	返回值类型	支持JDBC 4
clearParameters()	void	Yes
execute()	boolean	Yes
executeQuery()	ResultSet	Yes
executeUpdate()	int	Yes
getMetaData()	ResultSetMetaData	Yes
setBoolean(int parameterIndex, boolean x)	void	Yes
setBigDecimal(int parameterIndex, BigDecimal x)	void	Yes
setByte(int parameterIndex, byte x)	void	Yes
setBytes(int parameterIndex, byte[] x)	void	Yes
setDate(int parameterIndex, Date x)	void	Yes
setDouble(int parameterIndex, double x)	void	Yes
setFloat(int parameterIndex, float x)	void	Yes
setInt(int parameterIndex, int x)	void	Yes
setLong(int parameterIndex, long x)	void	Yes
setNString(int parameterIndex, String value)	void	Yes
setShort(int parameterIndex, short x)	void	Yes

方法名	返回值类型	支持JDBC 4
setString(int parameterIndex, String x)	void	Yes
addBatch()	void	Yes
executeBatch()	int[]	Yes
clearBatch()	void	Yes

说明

- addBatch()、execute()必须在clearBatch()之后才能执行。
- 调用executeBatch()方法并不会清除batch。用户必须显式使用clearBatch()清除 。
- 在添加了一个batch的绑定变量后，用户如果想重用这些值(再次添加一个batch)，无需再次使用set*()方法 。
- 以下方法是从java.sql.Statement继承而来：close，execute，executeQuery，executeUpdate，getConnection，getResultSet，getUpdateCount，isClosed，setMaxRows, setFetchSize。

4.3.9.5 java.sql.ResultSet

java.sql.ResultSet是执行结果集接口。

表 4-9 对 java.sql.ResultSet 的支持情况

方法名	返回值类型	支持JDBC 4
findColumn(String columnLabel)	int	Yes
getBigDecimal(int columnIndex)	BigDecimal	Yes
getBigDecimal(String columnLabel)	BigDecimal	Yes
getBoolean(int columnIndex)	boolean	Yes
getBoolean(String columnLabel)	boolean	Yes
getByte(int columnIndex)	byte	Yes
getBytes(int columnIndex)	byte[]	Yes
getByte(String columnLabel)	byte	Yes

方法名	返回值类型	支持JDBC 4
getBytes(String columnLabel)	byte[]	Yes
getDate(int columnIndex)	Date	Yes
getDate(String columnLabel)	Date	Yes
getDouble(int columnIndex)	double	Yes
getDouble(String columnLabel)	double	Yes
getFloat(int columnIndex)	float	Yes
getFloat(String columnLabel)	float	Yes
getInt(int columnIndex)	int	Yes
getInt(String columnLabel)	int	Yes
getLong(int columnIndex)	long	Yes
getLong(String columnLabel)	long	Yes
getShort(int columnIndex)	short	Yes
getShort(String columnLabel)	short	Yes
getString(int columnIndex)	String	Yes
getString(String columnLabel)	String	Yes
getTime(int columnIndex)	Time	Yes
getTime(String columnLabel)	Time	Yes
getTimestamp(int columnIndex)	Timestamp	Yes
getTimestamp(String columnLabel)	Timestamp	Yes
isAfterLast()	boolean	Yes
isBeforeFirst()	boolean	Yes
isFirst()	boolean	Yes

方法名	返回值类型	支持JDBC 4
next()	boolean	Yes

说明

- 一个Statement不能有多个处于“open”状态的ResultSet。
- 用于遍历结果集(ResultSet)的游标(Cursor)在被提交后不能保持“open”的状态。

4.3.9.6 java.sql.ResultSetMetaData

java.sql.ResultSetMetaData是对ResultSet对象相关信息的具体描述。

表 4-10 对 java.sql.ResultSetMetaData 的支持情况

方法名	返回值类型	支持JDBC 4
getColumnCount()	int	Yes
getColumnName(int column)	String	Yes
getColumnType(int column)	int	Yes
getColumnTypeName(int column)	String	Yes

4.3.9.7 java.sql.Statement

java.sql.Statement是SQL语句接口。

表 4-11 对 java.sql.Statement 的支持情况

方法名	返回值类型	支持JDBC 4
close()	void	Yes
execute(String sql)	boolean	Yes
executeQuery(String sql)	ResultSet	Yes
executeUpdate(String sql)	int	Yes
getConnection()	Connection	Yes
getResultSet()	ResultSet	Yes
getQueryTimeout()	int	Yes
getUpdateCount()	int	Yes

方法名	返回值类型	支持JDBC 4
isClosed()	boolean	Yes
setQueryTimeout(int seconds)	void	Yes
setFetchSize(int rows)	void	Yes
cancel()	void	Yes

说明

通过setFetchSize可以减少结果集在客户端的内存占用情况。它的原理是通过将结果集打包成游标，然后分段处理，所以会加大数据库与客户端的通信量，会有性能损耗。

由于数据库游标是事务内有效，所以，在设置setFetchSize的同时，需要将连接设置为非自动提交模式，setAutoCommit(false)。同时在业务数据需要持久化到数据库中时，在连接上执行提交操作。

4.3.9.8 javax.sql.ConnectionPoolDataSource

javax.sql.ConnectionPoolDataSource是数据源连接池接口。

表 4-12 对 javax.sql.ConnectionPoolDataSource 的支持情况

方法名	返回值类型	支持JDBC 4
getLoginTimeout()	int	Yes
getLogWriter()	PrintWriter	Yes
getPooledConnection()	PooledConnection	Yes
getPooledConnection(String user,String password)	PooledConnection	Yes
setLoginTimeout(int seconds)	void	Yes
setLogWriter(PrintWriter out)	void	Yes

4.3.9.9 javax.sql.DataSource

javax.sql.DataSource是数据源接口。

表 4-13 对 javax.sql.DataSource 接口的支持情况

方法名	返回值类型	支持JDBC 4
getConneciton()	Connection	Yes
getConnection(String username,String password)	Connection	Yes
getLoginTimeout()	int	Yes
getLogWriter()	PrintWriter	Yes
setLoginTimeout(int seconds)	void	Yes
setLogWriter(PrintWriter out)	void	Yes

4.3.9.10 javax.sql.PooledConnection

javax.sql.PooledConnection是由连接池创建的连接接口。

表 4-14 对 javax.sql.PooledConnection 的支持情况

方法名	返回值类型	支持JDBC 4
addConnectionEventListener (ConnectionEventListener listener)	void	Yes
close()	void	Yes
getConnection()	Connection	Yes
removeConnectionEventListener (ConnectionEventListener listener)	void	Yes
addStatementEventListener (StatementEventListener listener)	void	Yes
removeStatementEventListener (StatementEventListener listener)	void	Yes

4.3.9.11 javax.naming.Context

javax.naming.Context是连接配置的上下文接口。

表 4-15 对 javax.naming.Context 的支持情况

方法名	返回值类型	支持JDBC 4
bind(Name name, Object obj)	void	Yes

方法名	返回值类型	支持JDBC 4
bind(String name, Object obj)	void	Yes
lookup(Name name)	Object	Yes
lookup(String name)	Object	Yes
rebind(Name name, Object obj)	void	Yes
rebind(String name, Object obj)	void	Yes
rename(Name oldName, Name newName)	void	Yes
rename(String oldName, String newName)	void	Yes
unbind(Name name)	void	Yes
unbind(String name)	void	Yes

4.3.9.12 javax.naming.spi.InitialContextFactory

javax.naming.spi.InitialContextFactory是初始连接上下文工厂接口。

表 4-16 对 javax.naming.spi.InitialContextFactory 的支持情况

方法名	返回值类型	支持JDBC 4
getInitialContext(Hashtable<?,?> environment)	Context	Yes

4.4 Java SDK

4.4.1 使用前须知

DataArtsFabric SQL Java软件开发包（DataArtsFabric SQL Java SDK，DataArtsFabric SQL Java Software Development Kit）是DataArtsFabric SQL提供的REST API的Java语言版本的封装。用户可以使用SDK提供的接口方法来使用DataArtsFabric SQL服务，以简化用户在使用中的开发步骤。

本部分介绍JavaSDK的版本变更，并提供SDK的安装说明

变更说明

表4-17展示了本SDK的版本变更情况，以及某些功能变化的说明。

表 4-17 Java SDK 版本变更说明

版本	变更类型	变更说明
2.0.0-r3	SDK首次发布	无

使用前须知：

- 使用前请确认您已经开通并有权限使用DataArtsFabric SQL服务，包括DataArtsFabric SQL会涉及的Lakeformation服务和OBS服务等。
- 请确认您已了解DataArtsFabric SQL中的一些基本概念，如工作空间（Workspace）、工作端点（Endpoint）、访问密钥（AK和SK）、会话（Session）等。

Java SDK 方法概览

表4-18总结了DataArtsFabric SQL Java SDK支持的接口方法及描述，每个接口方法的详细描述和示例代码请单击表中方法名称跳转至详情页查看。

表 4-18 Java SDK 方法概述

方法名称	方法	功能描述
创建Session	createSession	创建一个新会话，返回会话ID。
关闭Session	closeSession	关闭指定会话，释放相关资源。
同步执行SQL	syncQuery	执行SQL查询并同步等待结果，返回完整数据集。
异步执行SQL	asyncQuery	提交SQL查询后立即返回任务ID，结果通过异步方式获取。
取消查询	cancelExecute	终止正在执行的异步查询任务。
获取查询结果(通过OBS获取)	getStatementResult	获取异步查询的结果，结果集从OBS中读取。
获取查询结果(直接获取)	getStatementResultDirect	获取异步查询的结果，结果集由服务端直接返回。

兼容性说明

推荐使用的JDK版本：JDK 1.8及以上版本。

4.4.2 使用前准备

在使用JavaSDK访问DataArtsFabric SQL之前，您需要先完成服务环境的准备和开发环境的准备。服务环境准备包括准备账号和访问密钥，是使用SDK与DataArtsFabric SQL

云服务交互的必要条件。开发环境准备是指为了您能顺利完成SDK的安装、完成基于SDK的代码开发与运行，需要提前在本地完成开发环境的搭建，例如下载安装依赖软件、安装开发工具等。

准备华为账号

在使用DataArtsFabric SQL服务之前您需要一个华为账号，注册账号并开通华为云服务步骤请参见注册华为账号并开通华为云。

获取访问密钥

DataArtsFabric SQL SDK会使用您提供账号的AK和SK进行签名认证以访问服务，防止未经授权的用户访问并使用DataArtsFabric SQL资源和数据。访问密钥（Access Key ID/Secret Access Key，简称AK/SK）包含访问密钥ID（AK）和秘密访问密钥（SK）两部分，是您在华为云的长期身份凭证，您可以通过访问密钥对华为云API的请求进行签名。华为云通过AK识别访问用户的身份，通过SK对请求数据进行签名验证，用于确保请求的机密性、完整性和请求者身份的正确性。

访问密钥分为永久访问密钥（AK/SK）和临时访问密钥（AK/SK和SecurityToken）两种，获取永久访问密钥请参考**管理IAM用户访问密钥**，同时可以创建临时访问密钥，创建方式参考临时访问密钥。

说明

- 企业联邦用户不能创建访问密钥，但可以创建临时访问凭证（临时AK/SK和SecurityToken），具体内容请参见临时访问密钥。
- IAM提供的“安全设置”功能，适用于管理员管理IAM用户的访问密钥。在我的凭证中也可以管理访问密钥，我的凭证适用于所有用户在可以登录控制台的情况下，自行管理访问密钥。
- 账号和IAM用户的访问密钥是单独的身份凭证，即账号和IAM用户仅能使用自己的访问密钥进行API调用。

获取 Java SDK

您可以通过以下方式获取Java SDK。

- 使用Maven中央仓库和Maven工程下载安装Java SDK。

  在Maven项目中添加以下依赖项到pom.xml文件：

```
<dependency>
    <groupId>com.huaweicloud.dws</groupId>
    <artifactId>dws-rest-api</artifactId>
    <version>[最新版本]</version>
</dependency>
```

  其中[最新版本]替换为实际版本，参见**表4-17**。

4.4.3 客户端初始化

使用Java SDK工具访问DataArtsFabric SQL，需要用户初始化DataArtsFabric SQL客户端。用户可以使用永久AK/SK或临时AK/SK两种认证方式初始化客户端，示例代码如下：

前提条件

- 已参考使用前须知获取了对应权限。

- 已参考使用前准备配置了Java SDK并取得了AK/SK或者Token。

永久 AK/SK 认证方式样例代码

- 方法定义

```
DwsRestClient(String endpoint, String accessKey, String secretKey)
```

- 参数说明

参数名	是否必填	描述
endpoint	是	DataArtsFabric SQL提供的服务地址。
accessKey	是	接入键标识
secretKey	是	安全接入键

- 代码样例

```
// 从环境变量读取参数（推荐方式）
String endpoint = "example.com";
String accessKey = System.getenv("FABRICLSQL_ACCESS_KEY");
String secretKey = System.getenv("FABRICLSQL_SECRET_KEY");
// 验证参数是否为空（可选）
if (endpoint == null || accessKey == null || secretKey == null) {
    throw new IllegalArgumentException("Missing required environment variables");
}
DwsRestClient client = new DwsRestClient(endpoint, accessKey, secretKey);
```

说明

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

本示例以AK和SK保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量FABRICLSQL_ACCESS_KEY和FABRICLSQL_SECRET_KEY。

临时 AK/SK 认证方式样例代码

- 方法定义

```
DwsRestClient(String endpoint, String accessKey, String secretKey, String securityToken)
```

- 参数说明

参数名	是否必填	描述
endpoint	是	DataArtsFabric SQL提供的服务地址。
accessKey	是	临时接入键标识
secretKey	是	临时安全接入键
securityToken	是	临时访问密钥Token

- 代码样例

```
// 从环境变量读取参数（推荐方式）
String endpoint = "example.com";
String accessKey = System.getenv("FABRICLSQL_ACCESS_KEY");
```

```
String secretKey = System.getenv("FABRICLSQL_SECRET_KEY")
String securityToken = System.getenv("FABRICLSQL_SECURITY_TOKEN");
// 验证参数是否为空（可选）
if (endpoint == null || accessKey == null || secretKey == null || securityToken == null) {
    throw new IllegalArgumentException("Missing required environment variables");
}
DwsRestClient client = new DwsRestClient(endpoint, accessKey, secretKey, securityToken);
```

4.4.4 SDK 方法介绍

创建 Session

- 功能介绍

  Session表示一个连接，用户可以通过一个Session执行SQL请求，并通过该Session查询请求结果。该方法可以通过用户提供的工作空间ID和端点ID同步创建一个Session，可以设置Session连接的lakeformation实例信息和catalog名称。

- 方法定义

```
createSession(String workspaceId, CreateSessionRequest request)
createSession(String workspaceId, CreateSessionRequest request, int timeout)
createSession(String workspaceId, CreateSessionRequest request, Map<String, String>
headers)
createSession(String workspaceId, CreateSessionRequest request, Map<String, String>
headers, int timeout)
```

- 参数说明：

表 **4-19** createSession 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
request	CreateSessionRequest	必选	session请求信息。
timeout	int	可选	请求超时时间（默认为-1，不设置超时时间）。
headers	Map<String,String>	可选	HTTP请求头信息。

表 **4-20** CreateSessionRequest

参数名称	参数类型	是否必选	描述
endpointId	str	必选	执行端点ID。
lakeFormationConfig	LakeFormationConfig	必选	LakeFormation实例ID。

表 4-21 LakeFormationConfig

参数名称	参数类型	是否必选	描述
instanceId	String	必选	LakeFormation实例ID。
catalog	String	必选	LakeFormation Catalog名称

- 响应说明

表 4-22 响应参数

参数名称	参数类型	描述
sessionId	String	sessionId

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key （Ak）
        // - 第三个参数：Secret Key （Sk）
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);
```

关闭 Session

- 功能介绍

  该方法可以通过用户提供的工作空间id和sessionId关闭指定Session。

- 方法定义

```
closeSession(String workspaceId, String sessionId)
closeSession(String workspaceId, String sessionId, int timeout)
closeSession(String workspaceId, String sessionId, Map<String, String> headers)
closeSession(String workspaceId, String sessionId, Map<String, String> headers, int
timeout)
```

- 参数说明：

表 4-23 closeSession 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
sessionId	String	必选	需要关闭session的sessionId。
timeout	int	可选	请求超时时间（默认为-1，不设置超时时间）。
headers	Map<String,String>	可选	Http请求头信息。

- 响应说明

  响应为空。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key (Ak)
        // - 第三个参数：Secret Key (Sk)
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);
client.closeSession(workspaceId, sessionId);
```

同步执行 SQL

- 功能介绍

  通过一个session同步下发SQL请求，并即时返回查询结果，用户需要提供工作空间ID和会话ID，并创建请求体信息。

- 方法定义

```
syncQuery(String workspaceId, StatementQuery query)
syncQuery(String workspaceId, StatementQuery query, int timeout)
syncQuery(String workspaceId, StatementQuery query, Map<String, String> headers)
```

```
syncQuery(String workspaceId, StatementQuery query, Map<String, String> headers, int timeout)
```

- 参数说明：

表 4-24 syncQuery 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
query	StatementQuery	必选	SQL请求类。
timeout	int	可选	请求超时时间。
headers	Map<String,String>	可选	HTTP请求头信息。

表 4-25 StatementQuery

参数名称	参数类型	是否必选	描述
statement	String	必选	下发SQL内容。
sessionId	String	必选	会话ID。
limit	int	可选	SQL limit限制。
bindings	List<List<String>>	可选	参数绑定。

- 响应体说明：返回请求响应体StatementResponse

表 4-26 StatementResponse

参数名称	参数类型	描述
status	int	状态。
sessionId	String	语句ID。
statementId	String	会话ID。
results	List<StatementResult>	查询结果。

表 4-27 StatementResult

参数名称	参数类型	描述
status	String	该条SQL的执行状态。

参数名称	参数类型	描述
statementId	String	语句ID。
numRows	int	语句总行数。
rowCount	int	该页总行数。
pageCount	int	总页数。
pageNo	int	当前页数，从数字1开始。
errCode	int	错误码。
message	String	错误信息。
resultSet	StatementResultSet	结果集。

表 **4-28** StatementResultSet

参数名称	参数类型	描述
columns	list[RowType]	列头数据。
rows	List<List<String>>	列数据。

说明

如果该请求执行失败或者查询本身不包含结果集，result_set中的columns和rows均为null。

表 **4-29** RowType

参数名称	参数类型	描述
name	String	列头名称。
tableId	String	表的ID。
columnId	String	列的ID。
format	int	格式。
type	int	类型。
size	int	大小。
typemod	int	typemod。

表 **4-30** StatementResponse 的 status 取值

状态值	描述
-1	执行失败（FAILED）。
0	执行成功（SUCCESSFUL）。
1	正在执行（RUNNING）。
2	指令执行完成，无返回值（EMPTY）。
3	正在排队等待执行（WAITING）。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key（Ak）
        // - 第三个参数：Secret Key （Sk）
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);

//执行查询
StatementQuery statementQuery = new StatementQuery();
statementQuery.setSessionId(sessionId);
statementQuery.setLimit(2);
statementQuery.setStatement("select 1");
StatementResponse statementResponse = client.syncQuery(workspaceId, statementQuery);
```

异步执行 SQL

- 功能介绍

  通过一个session异步下发SQL请求，并返回查询信息，供后续获取查询结果，用户需要提供工作空间ID和会话ID，并创建请求体信息。

- 方法定义

```
asyncQuery(String workspaceId, StatementQuery query)
asyncQuery(String workspaceId, StatementQuery query, int timeout)
asyncQuery(String workspaceId, StatementQuery query, Map<String, String> headers)
asyncQuery(String workspaceId, StatementQuery query, Map<String, String> headers, int
timeout)
```

- 参数说明：

表 4-31 asyncQuery 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
query	StatementQuery	必选	SQL请求类。
timeout	int	可选	请求超时时间。
headers	Map<String,String>	可选	HTTP请求头信息。

表 4-32 StatementQuery

参数名称	参数类型	是否必选	描述
statement	String	必选	下发SQL内容。
sessionId	String	必选	会话ID。
limit	int	可选	SQL limit限制。
bindings	List<List<String>>	可选	参数绑定。

- 响应体说明：返回请求响应体AsyncQueryResponse

表 4-33 AsyncQueryResponse

参数名称	参数类型	描述
sessionId	String	会话ID。
statementId	String	语句ID。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key (Ak)
        // - 第三个参数：Secret Key (Sk)
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
```

```
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);

//执行查询
StatementQuery statementQuery = new StatementQuery();
statementQuery.setSessionId(sessionId);
statementQuery.setLimit(2);
statementQuery.setStatement("select 1");
AsyncQueryResponse asyncQueryResponse = client.asyncQuery(workspaceId,
statementQuery, headers);
```

取消查询

- 功能介绍

  该方法可以通过用户提供的工作空间id、sessionId及statementId关闭指定Session。

- 方法定义

```
cancelExecute(String workspaceId, String sessionId, String statementId)
cancelExecute(String workspaceId, String sessionId, String statementId, Map<String,
String> headers)
cancelExecute(String workspaceId, String sessionId, String statementId, int timeout)
cancelExecute(String workspaceId, String sessionId, String statementId, Map<String,
String> headers, int timeout)
```

- 参数说明：

表 4-34 closeSession 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
sessionId	String	必选	对应的sessionId。
statementId	String	必选	需要终止查询的statementId
timeout	int	可选	请求超时时间(默认为-1，不设置超时时间)。
headers	Map<String,String>	可选	HTTP请求头信息。

- 响应说明

  响应为空。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key (Ak)
        // - 第三个参数：Secret Key (Sk)
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);

//执行查询
StatementQuery statementQuery = new StatementQuery();
statementQuery.setSessionId(sessionId);
statementQuery.setLimit(2);
statementQuery.setStatement("select 1");
AsyncQueryResponse asyncQueryResponse = client.asyncQuery(workspaceId,
statementQuery, headers);
```

获取查询结果（通过 OBS 获取）

- 功能介绍

  用户可以使用statementId及sessionId查询该语句的执行结果，并获取结果集。本接口方法是通过Java SDK访问OBS获取结果集并返回。

- 方法定义

```
getStatementResult(String workspaceId, String sessionId, String statementId, int pageNum)
getStatementResult(String workspaceId, String sessionId, String statementId, int
pageNum, int timeout)
getStatementResult(String workspaceId, String sessionId, String statementId, int
pageNum, Map<String, String> headers)
getStatementResult(String workspaceId, String sessionId, String statementId, int
pageNum, Map<String, String> headers, int timeout)
getStatementResult(String workspaceId, String sessionId, String statementId, int
pageNum, Map<String, String> headers, Region region)
getStatementResult(String workspaceId, String sessionId, String statementId, int
pageNum, Map<String, String> headers, int timeout, Region region)
```

- 参数说明：

表 4-35 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
sessionId	String	必选	会话ID。
statementId	String	必选	语句ID。
pageNum	int	必选	页码。
timeout	int	可选	请求超时时间。
headers	Map<String,String>	可选	HTTP请求头信息。
region	Region	可选	IAM认证Region，详情请参见**IAM Region说明**。

- 响应体说明

表 4-36 StatementResponse

参数名称	参数类型	描述
status	String	状态
statementId	String	语句ID
sessionId	int	会话ID。
results	List<StatementResult>	查询结果。

表 4-37 StatementResult

参数名称	参数类型	描述
status	String	该条SQL的执行状态。
statementId	String	语句ID。
numRows	int	语句总行数。
rowCount	int	该页总行数。
pageCount	int	总页数。
pageNo	int	当前页数，从数字1开始。

参数名称	参数类型	描述
errCode	int	错误码。
message	String	错误信息。
resultSet	StatementResultSet	结果集。

表 4-38 StatementResultSet

参数名称	参数类型	描述
columns	list[RowType]	列头数据。
rows	List<List<String>>	列数据。

说明

如果该请求执行失败或者查询本身不包含结果集，result_set中的columns和rows均为null。

表 4-39 RowType

参数名称	参数类型	描述
name	String	列头名称。
tableId	String	表的ID。
columnId	String	列的ID。
format	int	格式。
type	int	类型。
size	int	大小。
typemod	int	typemod。

表 4-40 StatementResponse 的 status 取值

状态值	描述
-1	执行失败（FAILED）。
0	执行成功（SUCCESSFUL）。
1	正在执行（RUNNING）。
2	指令执行完成，无返回值（EMPTY）。

状态值	描述
3	正在排队等待执行（WAITING）。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key（Ak）
        // - 第三个参数：Secret Key（Sk）
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);

//执行查询
StatementQuery statementQuery = new StatementQuery();
statementQuery.setSessionId(sessionId);
statementQuery.setLimit(2);
statementQuery.setStatement("select 1");
AsyncQueryResponse asyncQueryResponse = client.asyncQuery(workspaceId,
statementQuery);

StatementResponse statementResponse = client.getStatementResult(workspaceId,
sessionId, asyncQueryResponse.getStatementId(), 1);
while (statementResponse.getStatus() == StatementStatus.RUNNING.getValue()) {
    //轮询间隔
    TimeUnit.SECONDS.sleep(2);
    statementResponse = client.getStatementResult(workspaceId, sessionId,
asyncQueryResponse.getStatementId(), 1);
}
```

获取查询结果（直接获取）

- 功能介绍

  用户可以使用statementId及sessionId查询该语句的执行结果，并获取结果集。本接口方法是通过Java SDK访问服务端直接获取结果集并返回。

- 方法定义

```
StatementResponse getStatementResultDirect(String workspaceId, String sessionId, String
statementId, int pageNum, Map<String, String> headers, int timeout)
```

- 参数说明：

表 4-41 参数说明

参数名称	参数类型	是否必选	描述
workspaceId	String	必选	用户创建的DataArtsFabric的工作空间的ID。
sessionId	String	必选	会话ID。
statementId	String	必选	语句ID。
pageNum	int	必选	页码。
timeout	int	可选	请求超时时间。
headers	Map<String,String>	可选	HTTP请求头信息。

- 响应体说明

表 4-42 StatementResponse

参数名称	参数类型	描述
status	String	下发SQL内容。
statementId	String	会话ID。
sessionId	int	SQL limit限制。
results	List<StatementResult>	参数绑定。

表 4-43 StatementResult

参数名称	参数类型	描述
status	String	该条SQL的执行状态。
statementId	String	语句ID。
numRows	int	语句总行数。
rowCount	int	该页总行数。
pageCount	int	总页数。
pageNo	int	当前页数，从数字1开始。
errCode	int	错误码。
message	String	错误信息。
resultSet	StatementResultSet	结果集。

表 4-44 StatementResultSet

参数名称	参数类型	描述
columns	list[RowType]	列头数据。
rows	List<List<String>>	列数据。

说明

如果该请求执行失败或者查询本身不包含结果集，result_set中的columns和rows均为null。

表 4-45 RowType

参数名称	参数类型	描述
name	String	列头名称。
tableId	String	表的ID。
columnId	String	列的ID。
format	int	格式。
type	int	类型。
size	int	大小。
typemod	int	typemod。

表 4-46 StatementResponse 的 status 取值

状态值	描述
-1	执行失败（FAILED）。
0	执行成功（SUCCESSFUL）。
1	正在执行（RUNNING）。
2	指令执行完成，无返回值（EMPTY）。
3	正在排队等待执行（WAITING）。

- 示例代码

```
// 初始化DwsRestClient实例，传入必要的参数
        // 参数说明：
        // - 第一个参数：API的基础URL（endpoint）
        // - 第二个参数：Access Key (Ak)
        // - 第三个参数：Secret Key (Sk)
        // - 第四个参数：Security Token（可选，如果使用STS临时凭证）
DwsRestClient client = new DwsRestClient(
```

```
        "https://<your_endpoint_here>",
        "<your_access_key_here>",
        "<your_secret_key_here>",
        "<your_security_token_here>");
workspaceId = "<your_workspace_id_here>";
// 创建一个CreateSessionRequest对象，用于请求会话
CreateSessionRequest request = new CreateSessionRequest();
request.setEndpointId("<your_endpoint_id_here>");
// 创建LakeFormationConfig对象并设置相关配置
LakeFormationConfig lakeFormationConfig = new LakeFormationConfig();
lakeFormationConfig.setCatalog("<your_catalog_here>");
lakeFormationConfig.setInstanceId("<your_instance_id_here>");
request.setLakeFormationConfig(lakeFormationConfig);

// 调用client.createSession方法创建会话，并获取sessionId
String sessionId = client.createSession(workspaceId, request);

//执行查询
StatementQuery statementQuery = new StatementQuery();
statementQuery.setSessionId(sessionId);
statementQuery.setLimit(2);
statementQuery.setStatement("select 1");
AsyncQueryResponse asyncQueryResponse = client.asyncQuery(workspaceId,
statementQuery);
StatementResponse statementResponse = client.getStatementResultDirect(workspaceId,
sessionId, asyncQueryResponse.getStatementId(), 1,null,-1);
while (statementResponse.getStatus() == StatementStatus.RUNNING.getValue() ||
statementResponse.getStatus() == StatementStatus.WAITING.getValue()) {
    TimeUnit.SECONDS.sleep(2);
    statementResponse = client.getStatementResultDirect(workspaceId, sessionId,
asyncQueryResponse.getStatementId(), 1,null,-1);
}
```

5 性能调优

5.1 性能调优概述

数据库性能调优是指通过优化数据库系统的配置及SQL查询，以提高数据库性能和效率的过程。目的为消除性能瓶颈、减少响应时间、提高系统吞吐量和资源利用率，降低业务成本，从而提高系统稳定性，给用户带来更大的价值。

本章通过性能诊断、系统调优及SQL调优及常见SQL调优案例等性能调优的实际操作，为数据库性能调优人员提供全方位的指导。

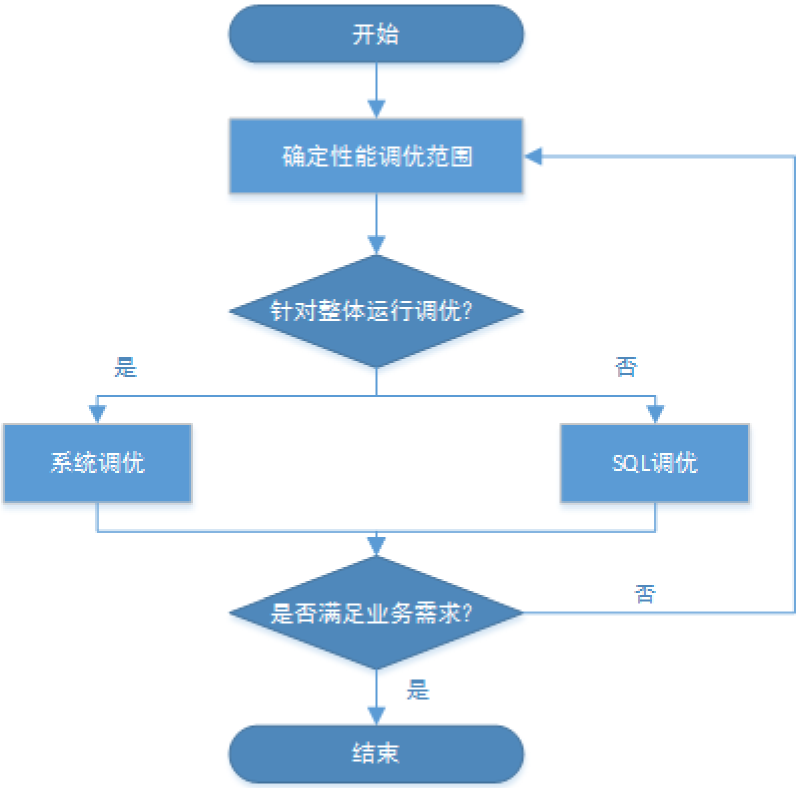
注意事项

数据库调优是一个复杂和细致的过程，需熟悉数据库系统的内部工作原理和相关技术。它需要综合考虑硬件、软件、查询、配置和数据结构等多个方面的因素，以达到最佳的性能和效率。因此，要求调优人员应对系统软件架构、软硬件配置、数据库配置参数、查询处理和数据库应用有广泛而深刻的理解。

调优流程

调优流程如**图5-1**所示。

图 5-1 DataArtsFabric SQL 性能调优流程



调优各阶段说明，如表5-1所示。

表 5-1 DataArtsFabric SQL 性能调优流程说明

阶段	描述
系统调优	进行系统级的调优，更充分地利用机器的CPU、内存、I/O和网络资源，提升查询的吞吐量。
SQL调优	审视业务所用SQL语句是否存在可优化空间，包括： ● 通过ANALYZE语句生成表统计信息：ANALYZE语句可收集与数据库中表内容相关的统计信息，统计结果存储在系统表PG_STATISTIC中。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。 ● 分析执行计划：EXPLAIN语句可显示SQL语句的执行计划，EXPLAIN PERFORMANCE语句可显示SQL语句中各算子的执行时间。 ● 查找问题根因并进行调优：通过分析执行计划，找到可能存在的原因，进行针对性的调优，通常为调整数据库级SQL调优参数。 ● 编写更优的SQL：介绍一些复杂查询中的中间临时数据缓存、结果集缓存、结果集合并等场景中的更优SQL语法。

5.2 系统调优

5.2.1 数据库系统参数调优

为了保证数据库尽可能高性能地运行，建议依据资源情况和业务实际进行数据库系统GUC参数的设置。本章节旨在介绍一些常用参数以及推荐配置，关于参数的详细设置方法请参考**查看和设置GUC参数**。

数据库内存相关参数

表 5-2 数据库内存相关参数

GUC参数	描述	建议
fq_runtime	设置查询使用的Actor的个数、CPU/内存配置。	● 该参数可以配置如下内容： – actor_size：每个actor的CPU核数和总内存 – actor：actor的个数 – dop：SMP并行度 可根据查询数据量的大小进行设置。
cstore_buffers	设置列存和OBS、HDFS外表列存格式（orc、parquet、carbondata）所使用的共享缓冲区的大小。	列存表使用cstore_buffers设置的共享缓冲区，几乎不用shared_buffers。因此在列存表为主的场景中，应减少shared_buffers，增加cstore_buffers。 OBS、HDFS外表使用cstore_buffers设置ORC、Parquet、Carbondata的元数据和数据的缓存，元数据缓存大小为cstore_buffers的1/4，最大不超过2GB，其余缓存空间为列存数据和外表列存格式数据共享使用。

5.2.2 SMP 并行执行

在复杂查询场景中，单个查询的执行较长，系统并发度低，通过SMP并行执行技术实现算子级的并行，能够有效减少查询执行时间，提升查询性能及资源利用率。

通过算子并行来提升性能，同时会占用更多的系统资源，包括CPU、内存、网络、I/O等等。本质上SMP是一种以资源换取时间的方式，在合适的场景以及资源充足的情况下，能够起到较好的性能提升效果；但是如果在不合适的场景下，或者资源不足的情况下，反而可能引起性能的劣化。同时，生成SMP需要考虑更多的候选计划，将会导致生成时间较长，相比串行场景也会引起性能的劣化。

DataArtsFabric SQL的SMP特性由GUC参数query_dop控制，该参数可设置用户自定义的查询并行度。

SMP 适用场景与限制

SMP适用场景：

- 支持并行的算子

  计划中存在以下算子支持并行：

  a. Scan：支持所有类型外表的扫描并行。
  b. Join：HashJoin、NestLoop
  c. Agg：HashAgg、SortAgg、PlainAgg、WindowAgg(只支持partition by，不支持order by)
  d. Stream：Redistribute、Broadcast
  e. 其他：Result、Subqueryscan、Unique、Material、Setop、Append、VectoRow、RowToVec

- SMP特有算子

  为了实现并行，新增了并行线程间的数据交换Stream算子供SMP特性使用。以下新增的算子可以看做Stream算子的子类：

  a. Local Gather：实现DN内部并行线程的数据汇总
  b. Local Redistribute：在DN内部各线程之间，按照分布键进行数据重分布
  c. Local Broadcast：将数据广播到DN内部的每个线程
  d. Local RoundRobin：在DN内部各线程之间实现数据轮询分发
  e. Split Redistribute：在集群跨DN的并行线程之间实现数据重分布
  f. Split Broadcast：将数据广播到集群所有DN的并行线程

  上述新增算子可以分为Local与非Local两类，Local类算子实现了DN内部并行线程间的数据交换，而非Local类算子实现了跨DN的并行线程间的数据交换。

- 示例说明

  以TPCH Q1的并行计划为例：

```
 id |                          operation                                           |
----+------------------------------------------------------------------------------
  1 | ->  Row Adapter                                                              |
  2 |   ->  Vector Streaming (type: GATHER)                                        |
  3 |     ->  Vector Sort                                                          |
  4 |       ->  Vector Sonic Hash Aggregate                                        |
  5 |         ->  Vector Streaming(type: REDISTRIBUTE dop: 1/4)                    |
  6 |           ->  Vector Sonic Hash Aggregate                                    |
  7 |             ->  Partitioned Vector Foreign Scan on tpch_parquet_1000x_partition_perf.lineitem |
```

  在这个计划中，实现了Foreign Scan以及HashAgg算子的并行，并且新增了Split Redistribute数据交换算子。

  其中5号算子为Split Redistribute算子，上面标有的“dop: 1/4”表明Split Redistribute的发送端和接收端线程的并行度分别为4和1，即下层的6号Hash Aggregate算子按照4并行度执行，而上层的1~4号算子按照串行执行。

  通过计划Stream算子上标明的dop信息即可看出各个算子的并行情况。

非适用场景：

1. 生成计划时间占比很高的短查询场景。
2. 不支持CN上的算子并行。
3. 不支持不能下推的查询并行执行。
4. 不支持子查询subplan的并行，以及包含子查询的算子并行。

资源对 SMP 性能的影响

SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，包括CPU、内存、I/O和网络带宽等资源的消耗都会出现明显的增长，而且随

着并行度的增大，资源消耗也随之增大。当上述资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致集群整体性能的劣化。SMP支持自适应特性，该特性会根据当前资源和查询特征，动态选取最优的并行度。下面对各种资源对SMP性能的影响情况分别进行说明：

- **CPU资源**

  在一般客户场景中，系统CPU利用率不高的情况下，利用SMP并行架构能够更充分地利用系统CPU资源，提升系统性能。但当数据库服务器的CPU核数较少，CPU利用率已经比较高的情况下，如果打开SMP并行，不仅性能提升不明显，反而可能因为多线程间的资源竞争而导致性能劣化。

- **内存资源**

  查询并行后会导致内存使用量的增长，但每个算子使用内存上限仍受到work_mem等参数的限制。假设work_mem为4GB，并行度为2，那么每个并行线程所分到的内存上限为2GB。在work_mem较小或者系统内存不充裕的情况下，使用SMP并行后，可能出现数据下盘，导致查询性能劣化的问题。

- **网络带宽资源**

  为了实现查询并行执行，会新增并行线程间的数据交换算子。对于Local类Stream算子，所需要进行数据交换的线程在同一个DN内，通过内存交换，不会增加网络负担。而非Local类算子，需要通过网络进行数据交换，因此会加重网络负担。当网络资源成为瓶颈的情况下，并行可能会导致一定程度的劣化。

- **I/O资源**

  要实现并行扫描必定会增加I/O的资源消耗，因此只有在I/O资源充足的情况下，并行扫描才能够提高扫描性能。

其他因素对 SMP 性能的影响

除了资源因素外，还有一些因素也会对SMP并行性能造成影响。例如分区表中分区数据不均，以及系统并发度等因素。

- **数据倾斜对SMP性能的影响**

  当数据中存在严重数据倾斜时，并行效果较差。例如某表join列上某个值的数据量远大于其他值，开启并行后，根据join列的值对该表数据做hash重分布，使得某个并行线程的数据量远多于其他线程，造成长尾问题，导致并行后效果差。

- **系统并发度对SMP性能的影响**

  SMP特性会增加资源的使用，而在高并发场景下资源剩余较少。所以，如果在高并发场景下，开启SMP并行，会导致各查询之间严重的资源竞争问题。一旦出现了资源竞争的现象，无论是CPU、I/O、内存或者网络资源，都会导致整体性能的下降。因此在高并发场景下，开启SMP经常不能达到性能提升的效果，甚至可能引起性能劣化。

SMP 相关参数配置建议

如果要打开SMP自适应功能，要设置query_dop=0。

SMP 配置方式

须知

系统的CPU、内存、I/O和网络带宽等资源充足。SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，当上述资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致性能的劣化。同时，SMP计划的生成时间较串行要长。因此，在短查询为主的TP类业务中，或者出现资源瓶颈的情况下，建议关闭SMP，即设置query_dop=1。

配置步骤：

1. 观察当前系统负载情况，如果系统资源充足（资源利用率小于50%），执行步骤2；否则退出。
2. 设置query_dop=1，利用explain打出执行计划，观察计划是否符合**SMP适用场景与限制**适用场景。如果符合，进入下一步。
3. 设置query_dop=-value，在考虑资源情况和计划特征基础上，限制dop选取的范围为[1,value]。
4. 设置query_dop=value，不考虑资源情况和计划特征，强制选取dop为1或value。
5. 在符合条件的查询语句执行前设置合适的query_dop值，在语句执行结束后关闭query_dop。例如，

```
SET query_dop = 0;
SELECT COUNT(*) FROM t1 GROUP BY a;
......
SET query_dop = 1;
```

说明

- 资源许可的情况下，并行度越高，性能提升效果越好。
- SMP并行度支持会话级设置，推荐客户在执行符合要求的查询前，打开smp，执行结束后，关闭smp。以免在业务峰值时，对业务造成冲击。
- SMP自适应（query_dop<=0）依赖资源管理，如果资源管理禁用（use_workload_manager为off），那么只会产生1或2并行度的计划。

5.3 SQL 调优

5.3.1 SQL 查询执行流程

SQL引擎从接收SQL语句到执行SQL语句需要经历的步骤如**图5-2**和**表5-3**所示。其中，红色字体部分为DBA可以介入实施调优的环节。

图 **5-2** SQL 引擎执行查询类 SQL 语句的流程

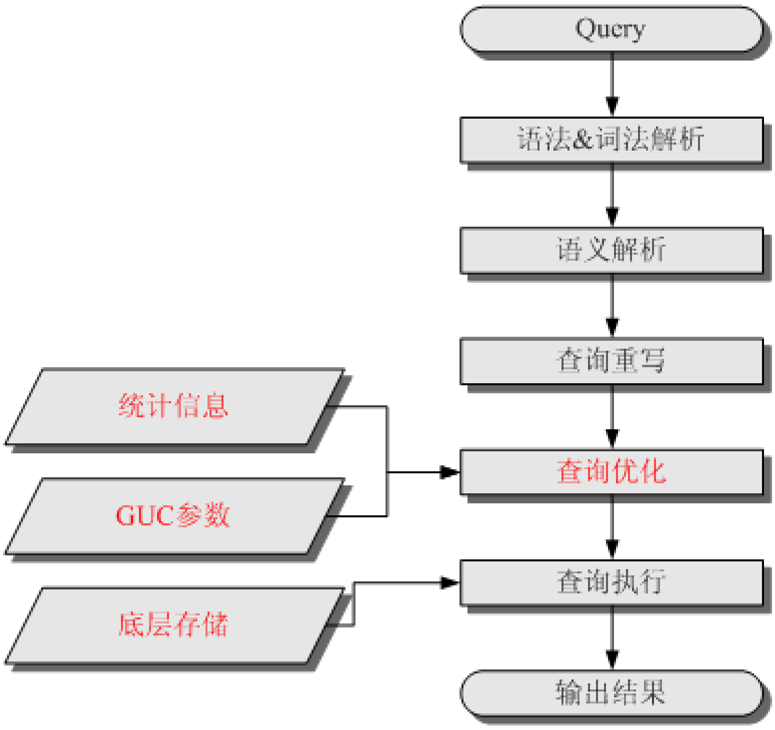


表 **5-3** SQL 引擎执行查询类 SQL 语句的步骤说明

步骤	说明
1、语法&词法解析	按照约定的SQL语句规则，把输入的SQL语句从字符串转化为格式化结构(Stmt)。
2、语义解析	将“语法&词法解析”输出的格式化结构转化为数据库可以识别的对象。
3、查询重写	根据规则把“语义解析”的输出等价转化为执行上更为优化的结构。
4、查询优化	根据“查询重写”的输出和数据库内部的统计信息规划SQL语句具体的执行方式，也就是执行计划。统计信息和GUC参数对查询优化（执行计划）的影响，请参见**调优手段之统计信息**和**调优手段之GUC参数**。
5、查询执行	根据“查询优化”规划的执行路径执行SQL查询语句。

调优手段之统计信息

DataArtsFabric SQL优化器是典型的基于代价的优化 (Cost-Based Optimization，简称CBO)。在这种优化器模型下，数据库根据表的元组数、字段宽度、NULL记录比率、distinct值、MCV值、HB值等表的特征值，以及一定的代价计算模型，计算出每一个执行步骤的不同执行方式的输出元组数和执行代价（cost），进而选出整体执行代价最小/首元组返回代价最小的执行方式进行执行。这些特征值就是统计信息。从上面描述可以看出统计信息是查询优化的核心输入，准确的统计信息将帮助优化器选择最合适的查询规划，一般来说通过ANALYZE语法收集整个表或者表的若干个字段的统

计信息，周期性地运行ANALYZE，或者在对表的大部分内容做了更改之后马上运行它是个好习惯。

调优手段之 GUC 参数

查询优化的主要目的是为查询语句选择高效的执行方式。

如下SQL语句:

```
SELECT count(1)
FROM customer inner join store_sales on (ss_customer_sk = c_customer_sk);
```

在执行customer inner join store_sales的时候，DataArtsFabric SQL支持Nested Loop和Hash Join两种不同的Join方式。优化器会根据表customer和表store_sales的统计信息估算结果集的大小以及每种join方式的执行代价，然后对比选出执行代价最小的执行计划。

正如前面所说，执行代价计算都是基于一定的模型和统计信息进行估算，当因为某些原因代价估算不能反映真实的cost的时候，就需要通过guc参数设置的方式让执行计划倾向更优规划。

调优手段之 SQL 重写

除了上述干预SQL引擎所生成执行计划的执行性能外，根据数据库的SQL执行机制以及大量的实践发现，有些场景下，在保证客户业务SQL逻辑的前提下，通过一定规则由DBA重写SQL语句，可以大幅度的提升SQL语句的性能。

这种调优场景对DBA的要求比较高，需要对客户业务有足够的了解，同时也需要扎实的SQL语句基本功，后续会介绍几个常见的SQL改写场景。

5.3.2 SQL 执行计划

SQL执行计划是一个节点树，显示DataArtsFabric SQL执行一条SQL语句时执行的详细步骤。

使用EXPLAIN命令可以查看优化器为每个查询生成的具体执行计划。EXPLAIN给每个执行节点都输出一行，显示基本的节点类型和优化器为执行这个节点预计的开销值。

执行计划显示信息

除了设置不同的执行计划显示格式外，还可以通过不同的EXPLAIN用法，显示不同详细程度的执行计划信息。常见有如下几种，关于更多用法请参见**EXPLAIN**说明。

- EXPLAIN *statement*： 只生成执行计划，不实际执行。其中statement代表SQL语句。
- EXPLAIN ANALYZE *statement*：生成执行计划，进行执行，并显示执行的概要信息。显示中加入了实际的运行时间统计，包括在每个规划节点内部花掉的总时间(以毫秒计)和它实际返回的行数。
- EXPLAIN PERFORMANCE *statement*：生成执行计划，进行执行，并显示执行期间的全部信息。

为了测量运行时在执行计划中每个节点的开销，EXPLAIN ANALYZE或EXPLAIN PERFORMANCE会在当前查询执行上增加性能分析的开销。在一个查询上运行EXPLAIN ANALYZE或EXPLAIN PERFORMANCE有时会比普通查询明显地花费更多的时间。超支的数量依赖于查询的本质和使用的平台。

因此，当定位SQL运行慢问题时，如果SQL长时间运行未结束，建议通过EXPLAIN命令查看执行计划，进行初步定位。如果SQL可以运行出来，则推荐使用EXPLAIN ANALYZE或EXPLAIN PERFORMANCE查看执行计划及其实际的运行信息，以便更精准地定位问题原因。

执行计划中的常见关键字说明：

- 表访问方式

  ForeignScan：全表顺序扫描。最基本的扫描算子，用于外表的顺序扫描，支持value分区剪枝。

- 表连接方式
  - Nested Loop

    嵌套循环，适用于被连接的数据子集较小的查询。在嵌套循环中，外表驱动内表，外表返回的每一行都要在内表中检索找到它匹配的行，因此整个查询返回的结果集不能太大（不能大于10000），要把返回子集较小的表作为外表，而且在内表的连接字段上建议要有索引。
  - （Sonic）Hash Join

    哈希连接，适用于数据量大的表的连接方式。优化器使用两个表中较小的表，利用连接键在内存中建立hash表，然后扫描较大的表并探测散列，找到与散列匹配的行。

- 运算符
  - sort

    对结果集进行排序。
  - filter

    EXPLAIN输出显示。WHERE子句被用作一个"filter"条件，附加到顺序扫描的计划节点上。这意味着规划器会为该节点扫描的每一行检查该条件，并仅输出满足条件的行。虽然预计的输出行数因WHERE子句而减少，但顺序扫描仍需访问所有10000行。因此，总体开销并未降低，反而因检查WHERE条件而增加了额外的CPU时间（具体为10000 × cpu_operator_cost）。
  - LIMIT

    LIMIT限定了执行结果的输出记录数。如果增加了LIMIT，将不会检索到所有行。

执行计划显示格式

DataArtsFabric SQL的执行计划显示格式层次清晰，计划包含了plan node id，性能分析简单直接。如图5-3。

图 5-3 pretty 格式执行计划示例

```
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 计划

DataArtsFabric SQL中使用Stream计划进行查询的执行：

CN根据原语句生成计划并将计划下发给DN进行执行，各DN执行过程中使用Stream算子进行数据交互。

现有表tt01和tt02定义如下：

```
CREATE TABLE tt01(c1 int, c2 int)

store as orc;

CREATE TABLE tt02(c1 int, c2 int)
store as orc;
```

两表JOIN，且连接条件包含非分布列，其DN间存在数据交换。此时对于tt02表，会在各DN进行基表扫描，扫描后会通过Redistribute Stream算子，按照JOIN条件中的tt02.c1进行哈希计算后重新发送给各DN，然后在各DN上做JOIN，最后汇总到CN。

图 5-4 stream 计划示例

```
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 详解

在SQL调优过程中经常需要执行EXPLAIN ANALYZE或EXPLAIN PERFORMANCE查看SQL语句实际执行信息，通过对比实际执行与优化器的估算之间的差别来为优化提供依据。EXPLAIN PERFORMANCE相对于EXPLAIN ANALYZE增加了每个DN上的执行信息。

以上一小节中的SQL查询语句为例：

```
SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c1;
```

执行EXPLAIN PERFORMANCE输出的显示执行信息分为以下6个部分：

1. 执行计划

图 5-5 explain performance 显示信息示例

```
 id |                   operation                   |        A-time        | A-rows | E-rows | E-distinct |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+-----------------------------------------------+----------------------+--------+--------+------------+----------------+----------+---------+---------+---------
  1 | ->  Row Adapter                               | 6205.676             |    100 |   5000 |            | 39KB           |          |         |      16 | 661.54
  2 |    ->  Vector Streaming (type: GATHER)        | 6205.436             |    100 |   5000 |            | 163KB          |          |         |      16 | 661.54
  3 |       ->  Vector Sonic Hash Join (4,6)        | [2427.736, 3033.795] |    100 |   5000 |            | [950KB, 950KB] | 16MB     |         |      16 | 173.74
  4 |          ->  Vector Streaming(type: BROADCAST) | [623.371, 1327.956] |    200 |   4000 | 200        | [72KB, 72KB]   | 2MB      |         |       8 | 119.99
  5 |             ->  Vector Foreign Scan on lmz.tt01 | [1765.576, 3857.581] | 100 |   1000 |            | [2MB, 2MB]     | 1MB      |         |       8 | 30.00
  6 |          ->  Vector Foreign Scan on lmz.tt02  | [1097.920, 2409.070] |    500 |   1000 | 200        | [2MB, 2MB]     | 1MB      |         |       8 | 30.00
```


以表格的形式将计划显示出来，包含有11个字段，分别是：id、operation、A-time、A-rows、E-rows、E-distinct、Peak Memory、E-memory、A-width、E-width和E-costs。字段含义如下表5-4。

表 5-4 执行字段说明

字段	描述
id	执行算子节点编号。
operation	具体的执行节点算子名称。 Vector前缀的算子是指向量化执行引擎算子，一般出现含有列存表的Query中。 Streaming是一个特殊的算子，它实现了分布式架构的核心数据shuffle功能，Streaming共有三种形态，分别对应了分布式结构下不同的数据shuffle功能： ● Streaming（type: GATHER）：作用是coordinator从DN收集数据。 ● Streaming(type: REDISTRIBUTE)：作用是DN根据选定的列把数据重分布到所有的DN。 ● Streaming(type: BROADCAST)：作用是把当前DN的数据广播给其他所有的DN。
A-time	各DN相应算子执行时间，一般DN上执行的算子的A-time是由[]括起来的两个值，分别表示此算子在所有DN上完成的最短时间和最长时间，包括下层算子执行时间。 注意：在整个计划中，除了叶子节点的执行时间是算子本身的执行时间，其余算子的执行时间均包含子节点的执行时间。
A-rows	表示相应算子输出的全局总行数。
E-rows	每个算子估算的输出行数。
E-distinct	表示hashjoin算子的distinct估计值。
Peak Memory	此算子在每个DN上执行时使用的内存峰值，[]中左侧为最小值，右侧为最大值。
E-memory	DN上每个算子估算的内存使用量，只有DN上执行的算子会显示。某些场景会在估算的内存使用量后使用括号显示该算子在内存资源充足下可以自动扩展的内存上限。
A-width	表示当前算子每行元组的实际宽度，仅对于重内存使用算子会显示，包括：(Vec)HashJoin、(Vec)HashAgg、(Vec)HashSetOp、(Vec)Sort、(Vec)Materialize算子等，其中(Vec)HashJoin计算的宽度是其右子树算子的宽度，会显示在其右子树上。
E-width	每个算子输出元组的估算宽度。

字段	描述
E-costs	每个算子估算的执行代价。 ● E-costs是优化器根据成本参数定义的单位来衡量的，习惯上以磁盘页面抓取为1个单位， 其它开销参数将参照它来设置。 ● 每个节点的开销（E-costs值）包括它的所有子节点的开销。 ● 开销只反映了优化器关心的东西，并没有把结果行传递给客户端的时间考虑进去。虽然这个时间可能在实际的总时间里占据相当重要的分量，但是被优化器忽略了，因为它无法通过修改规划来改变。

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
```

谓词过滤这部分主要显示的是对应执行算子节点的过滤条件，即在整个计划执行过程中不会变的信息，主要是一些join条件和一些filter信息。对于分区表，还会显示分区剪枝信息。

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: 1328.179, 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: 0.198; Stream Send time: 1.150; Data Serialize time: 0.068
        executor1_es_group Stream thread startup: 2.437;Stream pool init time: 0.169; 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
```

内存使用信息这部分显示的是整个计划中会将内存的使用情况打印出来的算子的内存使用信息，主要是Hash、Sort算子，包括算子峰值内存（peak memory）、优化器预估的内存（estimate memory）、控制内存（control memory）、估算内存使用（operator memory）、执行时实际宽度（width）、内存使用自动扩展次数（auto spread num）、是否提前下盘（early spilled）以及下盘信息，包括重复下盘次数（spill Time(s)）、内外表下盘分区数（inner/outer partition spill num）、下盘文件数（temp file num）、下盘数据量及最小和最大分区的下盘数据量（written disk IO [min, max] ）。其中sort算子不会显示具体的下盘文件数，仅在显示排序方法时显示Disk。下方是一个发生了下盘的算子的内存信息示例。

```
13 --Vector Sonic Hash Join (14,15)
       datanode1 Peak Memory: 34323KB, Control Memory: 12036KB
       datanode2 Peak Memory: 34323KB, Control Memory: 12036KB
       datanode3 Peak Memory: 34323KB, Control Memory: 12036KB
       datanode1 Stream thread startup: 2.761;Stream pool init time: 0.081; Stream Send time: 195.429, OS Kernel Send time: 96.284; Data Serialize time: 213.619
       datanode2 Stream thread startup: 4.035;Stream pool init time: 0.100; Stream Send time: 132.640, OS Kernel Send time: 80.149; Data Serialize time: 137.835
       datanode3 Stream thread startup: 1.779;Stream pool init time: 0.134; Stream Send time: 210.478, OS Kernel Send time: 102.706; Data Serialize time: 213.385
       datanode1 Partition Num: 96, Spill Times: 1
       datanode1 Inner Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 622367KB [4820KB 9761KB]
       datanode1 Outer Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 199125KB [1555KB 3113KB]
       datanode1 Hashjoin Build time: 5790.119, Probe time: 18041.761, Generate BF time: 575.776
       datanode1 Max Partition Memory Used : 0KB
       datanode2 Partition Num: 96, Spill Times: 1
       datanode2 Inner Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 622367KB [4820KB 9761KB]
       datanode2 Outer Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 129094KB [1009KB 2018KB]
       datanode2 Hashjoin Build time: 5662.444, Probe time: 15770.947, Generate BF time: 562.264
       datanode2 Max Partition Memory Used : 0KB
       datanode3 Partition Num: 96, Spill Times: 1
       datanode3 Inner Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 622367KB [4820KB 9761KB]
       datanode3 Outer Partition Spill Num: 96, Temp File Num: 96, Written Disk IO: 199125KB [1555KB 3113KB]
       datanode3 Hashjoin Build time: 5659.986, Probe time: 18134.547, Generate BF time: 563.589
       datanode3 Max Partition Memory Used : 0KB
```

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
```

这一部分显示的是每一个算子对应的输出目标列信息。

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)
```

这部分将各个算子的执行时间（如果包含过滤及投影也会显示对应的执行时间）、CPU、buffer的使用情况全部打印出来。

- 算子执行信息

```
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)
```

每个算子的执行信息都包含三个部分：

  - executor0_es_group/executor1_es_group表示具体执行的节点信息，括号中的信息是实际的执行信息。
  - actual time表示实际的执行时间，第一个数字表示执行时进入当前算子到输出第一条数据所花费的时间，第二个数字表示输出所有数据的总执行时间。

- rows表示当前算子输出数据行数。
- loops表示当前算子的执行次数。需要注意，对于分区表来说，每一个分区表的扫描就是一次完整的扫描操作，当切换到下一个分区的时候，又是一次新的扫描操作。

– CPU信息

```
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)
```

每个算子执行的过程都有CPU信息，其中cyc代表的是CPU的周期数，ex cyc表示的是当前算子的周期数，不包含其子节点；inc cyc是包含子节点的周期数；ex row是当前算子输出的数据行数；ex c/r则是ex cyc/ex row得到的每条数据所用的平均周期数。

– Buffer信息

```
executor0_es_group[worker 0] (Buffers: shared hit=1725 dirtied=117)
executor0_es_group[worker 1] (Buffers: shared hit=1723 dirtied=117)
```

buffers显示缓冲区信息，包括共享块和临时块的读和写。

共享块包含表和索引，临时块在排序和物化中使用的磁盘块。上层节点显示出来的块数据包含了其所有子节点使用的块数。

对于发生了下盘的算子，Buffer信息会展示下盘的数据量，“temp read”代表读取下盘的临时数据的次数，“written”代表写下盘的临时数据的次数，“written_size”代表下盘的数据量。

当关闭spill-to-OBS特性GUC开关enable_spill_to_remote_storage时，数据下盘到DN实例目录中，数据显示格式如下：

```
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)
```

当打开spill-to-OBS特性GUC开关enable_spill_to_remote_storage时，还会额外显示和该特性相关的下盘统计信息，示例如下图所示。其中，written_disk_size代表下盘到磁盘缓存的数据量；written_obs_size代表如磁盘缓存空间不足情况时直接下盘到obs的数据量；spill_obs_size代表的是磁盘缓存空间不足时，从磁盘缓存写回到obs的数据量。

```
executor11_es_group[worker 0] (Buffers: temp read=58600, written=58600, written_size=684962KB, written_disk_size=684962KB)
executor11_es_group[worker 1] (Buffers: temp read=49228, written=49228, written_size=574675KB, written_disk_size=574675KB, spill_obs_size=39497KB)
```

– 磁盘缓存信息

```
executor0_es_group[worker 0] (Disk Cache: hit=97178 miss=1 scanBytes=969.97MB remoteReadBytes=671.43MB loadTime=2568.4 openTime
=37.4 preadTime=375.2) (ReadAhead: parseMetaTime=448.7 submitBytes=959.01MB submitTime=4.3 waitTime=1340.8 waitCount=3
cancelCount=3 hitCount=1391 totalCount=1394 fabricCacheHitCount: L1Cache=717 L2Cache=3) (OBS I/O: count=258 averageRTT=262.3
averageLatency=202.9 latencyGt1s=2)
executor0_es_group[worker 1] (Disk Cache: hit=99601 scanBytes=993.30MB remoteReadBytes=684.00MB loadTime=2191.4 openTime=48.4
preadTime=385.1) (ReadAhead: parseMetaTime=305.0 submitBytes=982.41MB submitTime=4.5 waitTime=1103.8 waitCount=2 cancelCount=3
hitCount=1407 totalCount=1410 fabricCacheHitCount: L1Cache=729 L2Cache=2) (OBS I/O: count=264 averageRTT=215.2 averageLatency=
193.7 latencyGt1s=1)
executor10_es_group[worker 0] (Disk Cache: hit=118430 miss=1 scanBytes=1172.64MB remoteReadBytes=796.07MB loadTime=1841.4
openTime=35.8 preadTime=390.6) (ReadAhead: parseMetaTime=332.9 submitBytes=1161.68MB submitTime=4.5 waitTime=827.7 waitCount=6
cancelCount=7 hitCount=1609 totalCount=1616 fabricCacheHitCount: L1Cache=852 L2Cache=3) (OBS I/O: count=311 averageRTT=161.0
averageLatency=120.3)
```

- Disk Cache：表示磁盘缓存的命中信息和数据读取信息。
  - miss代表磁盘缓存未命中的次数。
  - hit表示磁盘缓存命中的次数。
  - disk_cache_error_code，error表示产生errorCode的次数。
  - scanBytes表示scan查询的数据量。
  - remoteReadBytes表示在OBS上读取的数据量。
  - loadTime表示从磁盘缓存加载数据的时间。
  - openTime表示打开磁盘缓存文件的时间。
  - preadTime表示从磁盘读取数据的时间。

注：为了提升OBS的效率会对相邻的请求块合并，或者因为请求写磁盘缓存的最小粒度是block(默认1M)，可能会使得scanBytes会小于remoteReadBytes。

- ReadAhead：表示数据预取的相关信息。
  - parseMetaTime表示预读时解析文件元数据的时间。
  - submitBytes表示预读的数据量。
  - submitTime表示数据预读请求的提交时间。
  - waitTime表示读取数据时命中执行中的数据预读请求并且等待预读请求完成的时间。
  - waitCount表示读取数据时命中执行中的数据预读请求的次数。
  - cancelCount表示读取数据时命中并撤销尚未开始执行的数据预读请求的次数。
  - hitCount表示读取数据时命中的已经完成的数据预读请求的次数。
  - fabricCacheHitCount L1Cache表示数据缓存在L1层命中的次数。
  - L2Cache表示数据缓存在L2层命中的次数。
- OBS I/O ： 表示OBS IO请求的详细信息。
  1) count表示OBS IO请求的总数量。
  2) averageRTT表示OBS IO请求的平均RTT(Round Trip Time，IO请求往返时间)，单位为μs。
  3) averageLatency表示OBS IO请求的平均延迟，单位为μs。
  4) latencyGt1s表示OBS IO请求延迟超过1s的数量。
  5) latencyGt10s表示OBS IO请求延迟超过10s的数量。
  6) retryCount表示OBS IO请求重试的总次数。
  7) rateLimitCount表示OBS IO请求被流控的总次数。

– 元戎接口调用信息

```
                         YR Api Call Information
------------------------------------------------------------------------------
executor0_es_group:
        YR::KV::Set(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Get(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Del(0, 0, 0ms, 0ms, 0ms)
executor1_es_group:
        YR::KV::Set(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Get(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Del(0, 0, 0ms, 0ms, 0ms)
Summary:
        YR::KV::Set(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Get(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
        YR::KV::Del(0, 0, 0ms, 0ms, 0ms, 0byte, 0byte/ms, 0byte/ms, 0byte/ms)
```

列出了所有executor actor上元戎接口调用统计信息，并在最后给出了总体信息。

每一行表示一个接口的统计信息，其字段的含义请参考函数yr_api_status()的字段。

– Query Summary

```
====== Query Summary =====

----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------
Datanode executor start time [executor5_es_group, executor11_es_group]: [83.846 ms,123.914 ms]
Datanode executor run time [executor15_es_group, executor5_es_group]: [54606.826 ms,54615.386 ms]
Datanode executor end time [executor0_es_group, executor20_es_group]: [0.047 ms,0.509 ms]
Remote query poll time: 55117.763 ms, Deserialze time: 0.032 ms
System available mem: 15937536KB
Query Max Assigned mem: 16777216KB
Query estimated mem: 16777216KB
Initial DOP: 4
Avail max core: 4.00
Final Max DOP: 4
LakeFormation request time: 118.362 ms, request count: 5
Coordinator executor start time: 14.874 ms
Coordinator executor run time: 55136.499 ms
Coordinator executor end time: 0.144 ms
Total network : 86kB
Parser runtime: 0.041 ms
YR utilize funtion analyse:
 Actor invoke runtime: 157.945 ms
 Actor Distribution Info: {10.42.3.93: [18]} {10.42.3.99: [17]} {10.42.0.156: [21]} {10.42.3.91: [16]} {10.42.3.101: [13]} {1
  {10.42.3.94: [4]} {10.42.0.157: [9]} {10.42.2.20: [3]} {10.42.3.97: [6]} {10.42.2.18: [2]} {10.42.3.95: [11]} {10.42.0.152:
  [22]} {10.42.0.153: [15]} {10.42.0.158: [14]} {10.42.3.100: [12]} {10.42.0.151: [7]} {10.42.0.159: [10]} {10.42.2.26: [19]}
 Actor Stream topo create time : [77.073 ms, 103.370 ms]
 Actor Consumer close time : [0.552 ms, 1.187 ms]
 Actor Stream Producer/Consumer count : [2304, 24]
Planner runtime: 340.622 ms
Query Id: 72057594037943640
Total runtime: 55492.182 ms
```

这一部分主要打印语句级执行信息，包括了各阶段执行时间、各个DN上初始化和结束阶段的最大最小执行时间、CN上的初始化、执行、结束阶段的时间，以及当前语句执行时系统可用内存、语句估算内存、并行度等信息。

- DN执行信息
  - DataNode executor start time：DN执行器开始时间，[min_node_name, max_node_name]：[min_time, max_time]。
  - DataNode executor run time：DN执行器运行时间，[min_node_name, max_node_name]：[min_time, max_time]。
  - DataNode executor end time：DN执行器结束时间，[min_node_name, max_node_name]：[min_time, max_time]。
- Remote query poll time：接收结果时用于poll等待的时间。
- 内存估算信息
  - System available mem：系统可用内存。
  - Query Max mem：查询最大内存。
  - Query estimated mem：语句估算内存。
- SMP自适应并行信息（仅开启SMP自适应时显示）
  - Initial DOP：计划生成的初始规划并行度。
  - Avail max core：可供该语句执行使用的CPU核数。
  - Final Max DOP：计划中算子的最大并行度。
- lakeformation访问信息：包括信息访问时间及次数。
- CN执行时间
  - Coordinator executor start time：CN执行器开始时间。
  - Coordinator executor run time：CN执行器运行时间。
  - Coordinator executor end time：CN执行器结束时间。
- Parser runtime：解析器运行时间。
- YR utilize function analyse：元戎调用函数分析，包括：
  - Actor invoke runtime：拉起actor的时间。

- Actor Distribution Info：拉起actor的IP及序号信息。
- Actor Stream topo create time：actor的stream启动时间。
- Actor Consumer close time：actor的consumer关闭时间。
- Actor Stream Producer/consumer count：actor的stream个数。

- Planner runtime：优化器执行时间。
- Query Id：查询ID。
- Total runtime：总执行时间。

须知

- A-rows和E-rows的差异体现了优化器估算和实际执行的偏差度。一般情况下两者偏差越大，则可以认为优化器生成的计划的越不可信，人工干预调优的必要性越大。
- A-time中的两个值偏差越大，表明此算子的计算偏斜(在不同DN上执行时间差异)越大，人工干预调优的必要性越大。一般来说，两个相邻的算子，上层算子的执行时间包含下层算子的执行时间，但如果上层算子为stream算子，由于各线程不存在驱动关系，上层算子执行时间可能小于下层算子的执行时间，即不存在包含关系。
- Max Query Peak Memory经常用来估算SQL语句耗费内存，也被用来作为SQL语句调优时运行态内存参数设置的重要依据。一般会以EXPLAIN ANALYZE或EXPLAIN PERFORMANCE的输出作为进一步调优的输入。

5.3.3 执行计划算子

算子介绍

SQL执行计划中每一个步骤为一个数据库运算符，也叫做一个执行算子。DataArtsFabric SQL中算子是基本的数据处理单元，合理地组合算子、优化算子的顺序和执行方式，可以提升数据的处理效率。

DataArtsFabric SQL算子可分为：扫描算子、控制算子、物化算子、连接算子、其他算子等。

扫描算子

扫描算子用来扫描表中的数据，每次获取一条元组作为上层节点的输入， 存在于查询计划树的叶子节点，它不仅可以扫描表，还可以扫描函数的结果集、链表结构、子查询结果集。常见的扫描算子如下表所示：

表 5-5 扫描算子

算子	含义	场景
VecForeignScan	顺序扫描	最基本的扫描算子，用于扫描外表。
VecSubqueryScan	子查询扫描	以另一个查询计划树（子计划）为扫描对象进行元组的扫描。

算子	含义	场景
FunctionScan	函数扫描	FROM function_name
ValuesScan	扫描values链表	对VALUES子句给出的元组集合进行扫描。

连接算子

连接算子对应了关系代数中的连接操作，以表t1 join t2为例，主要的集中连接类型如下：inner join、left join、right join、full join、semi join、 anti join，其实现方式包括Nestloop、HashJoin及MergeJoin。

表 5-6 连接算子

算子	含义	场景	实现特点
VecNestLoop	嵌套循环连接，暴力连接，对每一行都扫描内表。	Inner Join, Left Outer Join, Semi Join, Anti Join	适用于被连接的数据子集较小的查询。在嵌套循环中，外表驱动内表，外表返回的每一行都要在内表中检索找到它匹配的行，因此整个查询返回的结果集不能太大（不能大于10000），要把返回子集较小的表作为外表，而且在内表的连接字段上建议要有索引。
Vector Sonic HashJoin	哈希连接，内外表使用join列的hash值建立hash表，相同值的必在同一个hash桶。等值连接的连接两端必须为类型相同的等值连接，且支持hash散列。	Inner Join, Left Outer Join, Right Outer Join, Full Outer Join, Semi Join, Anti Join	哈希连接，适用于数据量大的表的连接方式。优化器使用两个表中较小的表，利用连接键在内存中建立hash表，然后扫描较大的表并探测散列，找到与散列匹配的行。Sonic和非Sonic的Hash Join的区别在于所使用hash表结构不同，不影响执行的结果集。

物化算子

物化算子是一类可缓存元组的节点。在执行过程中，很多扩展的物理操作符需要首先获取所有的元组才能进行操作（例如聚集函数操作、没有索引辅助的排序等），因此需要使用物化算子将元组缓存起来。

表 5-7 物化算子

算子	含义	场景
VecMaterial	物化	缓存子节点结果。

算子	含义	场景
VecSort	排序	ORDER BY子句，连接操作，分组操作，集合操作，配合Unique。
VecAgg	执行聚集函数	1. COUNT/SUM/AVG/MAX/MIN等聚集函数。 2. DISTINCT子句。 3. UNION去重。 4. GROUP BY子句。
VecWindowAgg	窗口函数	WINDOW子句。
VecSetOp	处理集合操作	INTERSECT/INTERSECT ALL，EXCEPT/EXCEPT ALL

控制算子

控制算子是一类用于处理特殊情况的节点，用于实现特殊的执行流程。

表 5-8 控制算子

算子	含义	场景
VecResult	直接进行计算	1. 不包含表扫描。 2. INSERT语句中只有一个VALUES子句。
VecModifyTable	INSERT/UPDATE/DELETE上层节点	INSERT/UPDATE/DELETE
VecAppend	追加	1. UNION(ALL)。 2. 继承表。
VecLimit	处理LIMIT子句	OFFSET … LIMIT …

其他算子

其他算子包括VecStream算子，以及RemoteQuery等算子。Stream算子主要有三种类型：Gather stream、Broadcast stream及Redistribute stream。

- Gather stream：每个源节点都将其数据发送给目标节点进行汇聚。
- Broadcast stream：由一个源节点将其数据发给N个目标节点进行运算。
- Redistribute stream：每个源节点将其数据根据连接条件计算Hash值，根据重新计算的Hash值进行分布，发给对应的目标节点。

表 5-9 其他算子

算子	含义	场景
VecStream	多节点数据交换	执行分布式查询计划，节点间存在数据交换。
RowToVec	行转列	行列混合场景。

5.3.4 SQL 执行监控

SQL 历史监控数据详解

SQL历史监控数据中记录了查询作业的各项资源使用情况（包括内存、下盘、CN和DN时长、OBS访问时间、LakeFormation访问时间等）以及SQL执行计划信息（EXPLAIN/EXPLAIN PERFORMANCE），且历史监控数据只有在SQL语句执行结束后才会生成。

SQL历史监控数据对外展示的字段如下：

表 5-10 SQL 历史监控数据对外展示的字段

字段名称	子字段名称	类型	描述
session_id	-	text	语句的Session ID。
statement_id	-	text	语句ID。
start_time	-	timestampt with zone	语句起始时间。
finish_time	-	timestampt with zone	语句上报时间/结束时间。
duration	-	bigint	语句运行时间。 单位：ms
cn_actor_info	-	text	CN的Serverless信息。
	actor_name	text	CN所在的Pod名称。
	actor_ip	text	CN所在的Pod IP。
	actor_id	text	CN在Pod中的名称。
dn_actor_info	-	text	DN的Serverless信息。
	actor_name	text	DN所在的Pod名称。
	actor_ip	text	DN所在的Pod ID。
	actor_id	text	DN在Pod中的名称。

字段名称	子字段名称	类型	描述
startup_coordinator_duration	-	bigint	启动CN时长。 单位：ms
create_session_duration	-	bigint	创建Session时长。 单位：ms
enqueue_statement_duration	-	bigint	语句加入任务队列时长。 单位：ms
queue_duration	-	bigint	语句在任务队列中排队时长。 单位：ms
write_resultset_duration	-	bigint	语句写结果集时长。 单位：ms
queryid	-	bigint	Debug Query的ID。
status	-	text	语句状态。
abort_info	-	text	报错信息。
schemaname	-	text	语句执行时的Schema名称。
query	-	text	语句。
query_plan	-	text	GUC参数resource_track_level为query时是explain信息，为perf时是explain performance信息。
tid	-	bigint	线程ID。
parse_time	-	bigint	从解析到执行开始之前的时间。 单位：ms
estimate_memory	-	int	语句估计内存。 单位：MB
dn_time	-	text	DN时长信息。
	min_dn_time	bigint	最小DN时长。 单位：ms
	max_dn_time	bigint	最大DN时长。 单位：ms
	average_dn_time	bigint	平均DN时长。 单位：ms
	dntime_skew_percent	int	DN时长偏斜百分比。 单位：%

字段名称	子字段名称	类型	描述
dn_cpu_time	-	text	DN CPU时长
	min_cpu_time	bigint	最小DN CPU时长。 单位：ms
	max_cpu_time	bigint	最大DN CPU时长。 单位：ms
	average_cpu_time	bigint	平均DN CPU时长。 单位：ms
	total_cpu_time	bigint	DN CPU总时长。 单位：ms
	cpu_skew_percent	int	DN CPU时长偏斜百分比。 单位：%
dn_peak_memory	-	text	DN峰值内存信息。
	min_peak_memory	int	所有DN最小峰值内存。 单位：MB
	max_peak_memory	int	所有DN最大峰值内存。 单位：MB
	average_peak_memory	int	所有DN平均峰值内存。 单位：MB
	memory_skew_percent	int	内存倾斜率。 单位：%
cn_spill_info	-	text	CN下盘信息。
	spill_count	bigint	CN下盘次数。
	spill_size	bigint	CN下盘大小。 单位：MB
	spill_obs_size	bigint	CN下盘OBS大小。 单位：MB
	write_obs_size	bigint	CN直写OBS大小。 单位：MB
	write_disk_size	bigint	CN写本地磁盘大小。 单位：MB
dn_spill_info	-	text	DN下盘信息。
	spill_dn_num	int	下盘DN数量。
	spill_count	int	DN下盘总次数

字段名称	子字段名称	类型	描述
	min_spill_size	bigint	最大下盘Size。 单位：MB
	max_spill_size	bigint	最小下盘Size。 单位：MB
	average_spill_size	bigint	平均下盘Size。 单位：MB
	spill_skew_percent	int	下盘倾斜率。 单位：%
	write_disk_total_size	bigint	DN下盘写本地磁盘总量。 单位：MB
	write_disk_max_size	bigint	DN最大写本地磁盘量。 单位：MB
	write_disk_min_size	bigint	DN最小写本地磁盘量。 单位：MB
	spill_obs_total_size	bigint	DN下盘写OBS总量。 单位：MB
	spill_obs_max_size	bigint	DN最大下盘写OBS量。 单位：MB
	spill_obs_min_size	bigint	DN最小下盘写OBS量。 单位：MB
	write_obs_total_size	bigint	DN直写OBS总量。 单位：MB
	write_obs_max_size	bigint	DN最大直写OBS量。 单位：MB
	write_obs_min_size	bigint	DN最小直写OBS量。 单位：MB
disk_cache_info	-	text	缓存信息。
	disk_cache_hit_ratio	numeric	磁盘缓存命中率。
	disk_cache_disk_read_size	bigint	磁盘缓存读取大小。 单位：B
	disk_cache_disk_write_size	bigint	磁盘缓存写入大小。 单位：B

字段名称	子字段名称	类型	描述
	disk_cache_remote_read_size	bigint	磁盘缓存远程读取大小。 单位：B
	disk_cache_remote_read_time	bigint	磁盘缓存远程读取次数。 单位：次
obs_info	-	text	OBS信息。
	vfs_scan_bytes	bigint	OBS文件系统扫描字节数。 单位：B
	vfs_remote_read_bytes	bigint	OBS 文件系统远程读取字节数。 单位：B
	disk_cache_load_time	bigint	磁盘缓存本地加载时间。 单位：us
	disk_cache_conflict_count	bigint	磁盘缓存块哈希冲突次数。 单位：次
	disk_cache_error_count	bigint	磁盘缓存错误次数。 单位：次
	disk_cache_error_code	bigint	磁盘缓存错误码。
	obs_io_req_avg_rtt	bigint	OBS IO请求平均往返时间。 单位：us
	obs_io_req_avg_latency	bigint	OBS IO请求平均延迟。 单位：us
	obs_io_req_latency_gt_1s	bigint	OBS IO请求延迟大于1秒的次数。 单位：次
	obs_io_req_latency_gt_10s	bigint	查询的OBS IO请求延迟大于10秒的次数。 单位：次
	obs_io_req_count	bigint	OBS IO请求次数。 单位：次
	obs_io_req_retry_count	bigint	OBS IO请求重试次数。 单位：次
	obs_io_req_rate_limit_count	bigint	OBS IO请求速率限制次数。 单位：次
yr_info	-	text	YR接口调用信息。

字段名称	子字段名称	类型	描述
	total_call	bigint	调用的总次数。
	failed_call	bigint	调用失败的次数。
	avg_call_during	bigint	平均调用时长。 单位：毫秒
	min_call_during	bigint	最小调用时长。 单位：毫秒
	max_call_during	bigint	最大调用时长。 单位：毫秒
	total_data	bigint	总数据量。 单位：字节
	avg_data_rate	bigint	平均数据处理速率。 单位：每毫秒字节数
	min_data_rate	bigint	最小数据处理速率。 单位：每毫秒字节数
	max_data_rate	bigint	最大数据处理速率。 单位：每毫秒字节数

当前SQL监控数据存放在结果集OBS桶中，和结果集文件夹同目录，可以在华为云网站中找到OBS服务打开，根据当前查询的结果集路径找到同目录下的对应SQL监控数据文件并下载查看。SQL监控数据文件打开后如下所示：

图 5-6 SQL 历史监控信息文件内容


```
{"session_id":"fa960ee8-a4ec-4f88-adbc-a184b7af8bea","statement_id":"bbafe11a-0a38-4556-8971-3031a7d0bda3","start_time":"2025-05-22T10:54:14.675626
+00:00","finish_time":"2025-05-22T10:54:21.715846+00:00","duration":7040,"cn_actor_info":"{\"actor_name\":\"function-agent-85b6fd7985-rxrbx\",
\"actor_ip\":\"10.42.0.107\",\"actor_id\":[\"runtime-551c57fd-3332-4000-8000-000007826e4c-6402e1492e4f\"]}","dn_actor_info":"
[{\"actor_name\":\"function-agent-85b6fd7985-rxrbx\",\"actor_ip\":\"10.42.0.107\",\"actor_id\":
[\"runtime-2b0c2a86-4c32-4000-8000-00007a1620ff-r-0-000000a40dcd\",\"runtime-2b0c2a86-4c32-4000-8000-00007a1620ff-r-1-fbcac2bb86cf\"]}]",
"create_coordinator_time":"2000-01-01T00:00:00.000201+00:00","startup_coordinator_duration":1604,"create_session_duration":16,
"enqueue_statement_duration":0,"queue_duration":0,"write_resultset_duration":3303,"queryid":72057594037927938,"status":"finished","abort_info":null,
"schemaname":"test_parquet_ra","query":"select * from lineitem_orc limit 100000;","query_plan":" id |
operation                          |  A-time   | A-rows | E-rows | E-distinct | Peak Memory | E-memory | A-width | E-width | E-costs \n----
+-----------------------------------------------------------------------------------+----------+--------+--------+-----------+-----------+-----------+---------
+---------+---------\n  1 | ->  Row Adapter                                                          | 2338.178 | 100000 |    1000 |           |
148KB      |          |          |     398 | 205.75  \n  2 |    ->  Vector Limit                                                              | 2274.792 | 100000 |
1000 |            | 8KB         |          |          |     398 | 205.75  \n  3 |       ->  Vector Streaming (type: GATHER)                                    |
2274.083 | 100000 |    1000 |           | 923KB       |          |          |     398 | 205.75  \n  4 |            ->  Vector
Limit                                          |          |        |    1000 |           |             | 1MB      |          |     398 | 60.00   \n
5 |               ->  Vector Foreign Scan on test_parquet_ra.lineitem_orc |           |        |    1000 |           |             |
1MB      |          |     398 | 60.00   \n\n     Predicate Information (identified by plan id)
\n-----------------------------------------------------------\n  5 --Vector Foreign Scan on test_parquet_ra.lineitem_orc\n        Server Type: lf\n
executor1_es_group: 85 files 2249.00 MB\n        executor0_es_group: 85 files 2276.00 MB\n        Total files left: 170\n\n    Memory Information
(identified by plan id)    \n-------------------------------------------------\n  1 --Row Adapter\n        Peak Memory: 148KB, Estimate Memory: 64MB\n
2 --Vector Limit\n        Peak Memory: 8KB\n  3 --Vector Streaming (type: GATHER)\n        Peak Memory:
923KB\n\n                                                                   Targetlist Information (identified by plan
id)
\n----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------\n  1 --Row Adapter\n        Output: l_orderkey, l_partkey, l_suppkey, l_linenumber,
l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode,
l_comment\n  2 --Vector Limit\n        Output: l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax,
l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment\n  3 --Vector Streaming (type: GATHER)
\n        Output: l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus,
l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment\n        Node/s: All datanodes\n  4 --Vector Limit\n        Output:
l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus, l_shipdate, l_commitdate,
l_receiptdate, l_shipinstruct, l_shipmode, l_comment\n  5 --Vector Foreign Scan on test_parquet_ra.lineitem_orc\n        Output: l_orderkey, l_partkey,
l_suppkey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate,
l_shipinstruct, l_shipmode, l_comment\n\n                    Datanode Information (identified by plan id)
\n--------------------------------------------------------------------------------\n  1 --Row Adapter\n        (actual time=2229.851..2338.178
rows=100000 loops=1)\n        (CPU: ex c/r=1281, ex row=100000, ex cyc=128148934, inc cyc=5132630640)\n  2 --Vector Limit\n        (actual time=2229.
520..2274.792 rows=100000 loops=1)\n        (CPU: ex c/r=16, ex row=100000, ex cyc=1637770, inc cyc=5004481706)\n  3 --Vector Streaming (type: GATHER)
\n        (actual time=2229.455..2274.083 rows=100000 loops=1)\n        (Buffers: 0)\n        (CPU: ex c/r=50028, ex row=100000, ex cyc=5002843936, inc
cyc=5002843936)\n\n                              ====== Query Summary =====
\n-----------------------------------------------------------------------------------------\nDatanode executor start time [executor0_es_group,
executor0_es_group]: [0.000 ms,0.000 ms]\nDatanode executor run time [executor0_es_group, executor0_es_group]: [0.000 ms,0.000 ms]\nDatanode executor
end time [executor0_es_group, executor0_es_group]: [0.000 ms,0.000 ms]\nRemote query poll time: 2208.760 ms, Deserialze time: 33.486 ms\nSystem
available mem: 1991680KB\nQuery Max Assigned mem: 2097152KB\nQuery estimated mem: 2048KB\nLakeFormation request time: 65.742 ms, request count:
3\nCoordinator executor start time: 17.876 ms\nCoordinator executor run time: 5539.416 ms\nCoordinator executor end time: 29.817 ms\nParser runtime: 0.
125 ms\nYR utilize funtion analyse:\n Actor invoke runtime: 170.444 ms\n Actor Distribution Info: {10.42.0.107: [0, 1]} \n Actor Stream topo create time
: [0.000 ms, 0.000 ms]\n Actor Consumer close time : [0.000 ms, 0.000 ms]\n Actor Stream Producer/Consumer count : [0, 0]\nPlanner runtime: 1232.148
ms\nQuery Id: 72057594037927938\nTotal runtime: 6819.523 ms","pid":139987815336336,"parse_time":1469,"estimate_memory":2,"dn_time":"
{\"min_dn_time\":1979,\"max_dn_time\":4927,\"average_dn_time\":3453,\"dntime_skew_percent\":29}","dn_cpu_time":"{\"min_cpu_time\":348,
\"max_cpu_time\":357,\"average_cpu_time\":352,\"total_cpu_time\":705,\"cpu_skew_percent\":1}","dn_peak_memory":"{\"min_peak_memory\":4,
\"max_peak_memory\":4,\"average_peak_memory\":4,\"memory_skew_percent\":0}","dn_spill_info":"{\"spill_info\":{\"spill_dn_num\":0,\"datanodes_num\":0},
\"min_spill_size\":0,\"max_spill_size\":0,\"average_spill_size\":0,\"spill_skew_percent\":0}","previous_billed_bytes":24454013,"disk_cache_info":"
{\"disk_cache_hit_ratio\":0.87735849056603776,\"disk_cache_disk_read_size\":19940890,\"disk_cache_disk_write_size\":0,
\"disk_cache_remote_read_size\":4513123,\"disk_cache_remote_read_time\":13}","obs_info":"{\"vfs_scan_bytes\":24454013,\"vfs_remote_read_bytes\":16081820,
\"preload_submit_time\":0,\"preload_wait_time\":3592674,\"preload_wait_count\":0,\"disk_cache_load_time\":7582599,\"disk_cache_conflict_count\":0,
\"disk_cache_error_count\":0,\"disk_cache_error_code\":0,\"obs_io_req_avg_rtt\":162879,\"obs_io_req_avg_latency\":480599,\"obs_io_req_latency_gt_1s\":7,
\"obs_io_req_latency_gt_10s\":0,\"obs_io_req_count\":25,\"obs_io_req_retry_count\":0,\"obs_io_req_rate_limit_count\":0}"}
```


SQL 实时监控数据详解

SQL实时监控数据中同样记录了查询作业的各项资源使用情况（包括内存、下盘、CN和DN时长、OBS访问时间、LakeFormation访问时间等）以及SQL执行计划信息（EXPLAIN/EXPLAIN PERFORMANCE），但SQL实时监控数据只能获取正在执行的语句的信息，对于执行结束的语句无法获取信息。

使用方式：select * from gs_wlm_session_statistics();

SQL实时监控数据对外展示的字段如下：

表 5-11 SQL 实时监控数据对外展示的字段

字段名称	类型	描述
session_id	text	语句的Session ID。
statement_id	text	语句ID。
start_time	timestampt with zone	语句起始时间。
duration	bigint	语句运行时间。 单位：ms
estimate_total_time	bigint	估计的语句运行时长。 单位：ms
estimate_left_time	bigint	估计语句剩余运行时长 单位：ms

字段名称	类型	描述
cn_actor_info	text	CN的Serverless信息。
dn_actor_info	text	DN的Serverless信息。
startup_coordinator_duration	bigint	启动CN时长。 单位：ms
create_session_duration	bigint	创建Session时长。 单位：ms
enqueue_statement_duration	bigint	语句加入任务队列时长。 单位：ms
queue_duration	bigint	语句在任务队列中排队时长。 单位：ms
write_resultset_duration	bigint	语句写结果集时长。 单位：ms
queryid	bigint	Debug Query的ID。
schemaname	text	语句执行时的Schema名称。
query	text	语句。
query_plan	text	explain信息
tid	bigint	线程ID。
parse_time	bigint	从解析到执行开始之前的时间。 单位：ms
estimate_memory	int	语句估计内存。 单位：MB
min_dn_time	bigint	最小DN时长。 单位：ms
max_dn_time	bigint	最大DN时长。 单位：ms
average_dn_time	bigint	平均DN时长。 单位：ms
dn_time_skew_percent	int	DN时长偏斜百分比。 单位：%
dn_min_cpu_time	bigint	最小DN CPU时长。 单位：ms
dn_max_cpu_time	bigint	最大DN CPU时长。 单位：ms

字段名称	类型	描述
dn_average_cpu_time	bigint	平均DN CPU时长。 单位：ms
dn_total_cpu_time	bigint	DN CPU总时长。 单位：ms
dn_cpu_skew_percent	int	DN CPU时长偏斜百分比。 单位：%
dn_min_peak_memory	int	所有DN最小峰值内存。 单位：MB
dn_max_peak_memory	int	所有DN最大峰值内存。 单位：MB
dn_average_peak_memory	int	所有DN平均峰值内存。 单位：MB
dn_memory_skew_percent	int	内存倾斜率。 单位：%
dn_spill_)num	int	下盘DN数量。
dn_spill_count	int	下盘DN次数总量。
dn_min_spill_size	bigint	最大下盘Size。 单位：MB
dn_max_spill_size	bigint	最小下盘Size。 单位：MB
dn_average_spill_size	bigint	平均下盘Size。 单位：MB
dn_spill_skew_percent	int	下盘倾斜率。 单位：%
dn_write_disk_total_size	bigint	dn下盘写磁盘总量。 单位：MB
dn_write_disk_max_size	bigint	dn最大下盘写磁盘量。 单位：MB
dn_write_disk_min_size	bigint	dn最小下盘写磁盘量。 单位：MB
dn_spill_obs_total_size	bigint	dn下盘写OBS总量。 单位：MB
dn_spill_obs_max_size	bigint	dn最大下盘写OBS量。 单位：MB

字段名称	类型	描述
dn_spill_obs_min_size	bigint	dn最小下盘写OBS量。 单位：MB
dn_write_obs_total_size	bigint	dn下盘直写OBS总量。 单位：MB
dn_write_obs_max_size	bigint	dn下盘最大直写OBS量。 单位：MB
dn_write_obs_min_size	bigint	dn下盘最小直写OBS量。 单位：MB
cn_spill_count	int	cn下盘次数。
cn_spill_size	bigint	cn下盘总量。 单位：MB
cn_spill_obs_size	bigint	cn下盘写OBS量。 单位：MB
cn_write_obs_size	bigint	cn下盘直写OBS量。 单位：MB
cn_write_obs_disk_size	bigint	cn下盘写本地磁盘量。 单位：MB
disk_cache_hit_ratio	numeric	磁盘缓存命中率。
disk_cache_disk_read_size	bigint	磁盘缓存读取大小。 单位：B
disk_cache_disk_write_size	bigint	磁盘缓存写入大小。 单位：B
disk_cache_remote_read_size	bigint	磁盘缓存远程读取大小。 单位：B
disk_cache_remote_read_time	bigint	磁盘缓存远程读取次数。 单位：次
vfs_scan_bytes	bigint	OBS文件系统扫描字节数。 单位：B
vfs_remote_read_bytes	bigint	OBS 文件系统远程读取字节数。 单位：B
disk_cache_load_time	bigint	磁盘缓存本地加载时间。 单位：us

字段名称	类型	描述
disk_cache_conflict_count	bigint	磁盘缓存块哈希冲突次数。 单位：次
disk_cache_error_count	bigint	磁盘缓存错误次数。 单位：次
disk_cache_error_code	bigint	磁盘缓存错误码。
obs_io_req_avg_rtt	bigint	OBS IO请求平均往返时间。 单位：us
obs_io_req_avg_latency	bigint	OBS IO请求平均延迟。 单位：us
obs_io_req_latency_gt_1s	bigint	OBS IO请求延迟大于1秒的次数。 单位：次
obs_io_req_latency_gt_10s	bigint	查询的OBS IO请求延迟大于10秒的次数。 单位：次
obs_io_req_count	bigint	OBS IO请求次数。 单位：次
obs_io_req_retry_count	bigint	OBS IO请求重试次数。 单位：次
obs_io_req_rate_limit_count	bigint	OBS IO请求速率限制次数。 单位：次

5.3.5 SQL 调优流程

对慢SQL语句进行分析，通常包括以下步骤：

操作步骤

步骤1 收集SQL中涉及到的所有表的统计信息。在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧会造成执行计划严重劣化，从而导致性能问题。从经验数据来看，10%左右性能问题是因为没有收集统计信息。具体请参见**更新统计信息**。

步骤2 通常情况下，有些SQL语句可以通过查询重写转换成等价的，或特定场景下等价的语句。重写后的语句比原语句更简单，且可以简化某些执行步骤达到提升性能的目的。查询重写方法在各个数据库中基本是通用的。**SQL语句改写规则**介绍了几种常用的通过改写SQL进行调优的方法。

步骤3 通过查看执行计划来查找原因。如果SQL长时间运行未结束，通过EXPLAIN命令查看执行计划，进行初步定位。如果SQL可以运行出来，则推荐使用EXPLAIN ANALYZE或EXPLAIN PERFORMANCE查看执行计划及实际运行情况，以便更精准地定位问题原因。有关执行计划的详细介绍请参见**SQL执行计划**。

步骤4 针对EXPLAIN或EXPLAIN PERFORMANCE信息，定位SQL慢的具体原因以及改进措施，具体参见**SQL调优进阶**。

步骤5 用户可以通过指定join顺序，join、stream、scan方法，指定结果行数，指定重分布过程中的倾斜信息等多个手段来进行执行计划的调优，以提升查询的性能。详细请参见**使用Plan Hint进行调优**。

步骤6 （可选）DataArtsFabric SQL支持在资源富足的情况下，通过算子并行来提升性能。详细请参见**SMP并行执行**。

----结束

5.3.6 更新统计信息

在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧会造成执行计划严重劣化，从而导致性能问题。

背景信息

ANALYZE语句可收集与数据库中表内容相关的统计信息，统计结果存储在系统表PG_STATISTIC中。查询优化器会使用这些统计数据，以生成最有效的执行计划。

建议在执行了大批量插入/删除操作后，例行对表或全库执行ANALYZE语句更新统计信息。目前默认收集统计信息的采样比例是30000行（即：guc参数default_statistics_target默认设置为100），如果表的总行数超过一定行数（大于1600000），建议设置guc参数default_statistics_target为-2，即按2%收集样本估算统计信息。

对于在批处理脚本或者存储过程中生成的中间表，也需要在完成数据生成之后显式地调用ANALYZE。

对于表中多个列有相关性且查询中有同时基于这些列的条件或分组操作的情况，可尝试收集多列统计信息，以便查询优化器可以更准确地估算行数，并生成更有效的执行计划。

生成统计信息

更新单个表的统计信息。

```
ANALYZE tablename;
```

须知

使用EXPLAIN查看各SQL的执行计划时，如果发现某个表SEQ SCAN的输出中rows=10，rows=10是系统给的默认值，有可能该表没有进行ANALYZE，需要对该表执行ANALYZE。

提升统计信息质量

ANALYZE是按照随机采样算法从表上采样，根据样本计算表数据特征。采样数可以通过配置参数default_statistics_target进行控制，default_statistics_target取值范围为-100~10000，默认值为100。

- 当default_statistics_target > 0时；采样的样本数为300*default_statistics_target，default_statistics_target取值越大，采样的样本也越大，样本占用的内存空间也越大，统计信息计算耗时也越长。
- 当default_statistics_target < 0时，采样的样本数为 (default_statistics_target)/100*表的总行数，default_statistics_target取值越小，采样的样本也越大。当default_statistics_target < 0时会把采样数据下盘，不存在样本占用的内存空间的问题，但是因为样本过大，计算耗时长的问题同样存在。

  default_statistics_target < 0时，实际采样数是 (default_statistics_target)/100*表的总行，所以又称之为百分比采样。

自动收集统计信息

当配置参数autoanalyze打开时，查询语句走到优化器发现表不存在统计信息或数据变化超过阈值时，会自动触发统计信息收集，以满足优化器的需求。

基于代价的优化器模型（CBO，cost base optimizer）中，统计信息决定了查询计划生成的好坏。因此，统计信息的及时有效很重要。

- 表级统计信息，存储在pg_class的relpages、reltuples中。
- 列级统计信息，存储在pg_statistics中，可以通过pg_stats视图查看。包括：NULL值比例，distinct值占比，高频值MCV，直方图histgram等。

收集条件：当数据量发生较大变化，默认是变化10%，认为数据特征已经有了变化，需要重新收集统计信息。

准确性

表 5-12 准确性

分类	说明
采样大小	可配置为按表大小自适应。由参数default_statistics_target控制。
采样随机性	• analyze_sample_mode参数设置新支持优化蓄水池和range采样随机性更优。 • random_function_version参数增强了随机数计算函数的随机性。
统计信息推算	enable_extrapolation_stats参数可以控制估算失真时，基于旧的统计信息自动推算更准确的统计信息。

统计信息查看

查看lake formation上的统计信息。

```
desc extended relation;
```

5.3.7 SQL 调优进阶

5.3.7.1 子查询调优

子查询背景介绍

应用程序通过SQL语句来操作数据库时会使用大量的子查询，这种写法比直接对两个表做连接操作在结构上和思路上更清晰，尤其是在一些比较复杂的查询语句中，子查询有更完整、更独立的语义，会使SQL对业务逻辑的表达更清晰更容易理解，因此得到了广泛的应用。

DataArtsFabric SQL根据子查询在SQL语句中的位置把子查询分成了子查询、子链接两种形式。

- 子查询SubQuery：对应于查询解析树中的范围表RangeTblEntry，更通俗一些指的是出现在FROM语句后面的独立的SELECT语句。
- 子链接SubLink：对应于查询解析树中的表达式，更通俗一些指的是出现在where/on子句、targetlist里面的语句。

  综上，对于查询解析树而言，SubQuery的本质是范围表，而SubLink的本质是表达式。针对SubLink场景而言，由于SubLink可以出现在约束条件、表达式中，按照DataArtsFabric SQL对sublink的实现，sublink可以分为以下几类：

  – exist_sublink：对应EXIST、NOT EXIST语句
  – any_sublink：对应op Any(select…)语句，其中OP可以是IN,<,>,=操作符
  – all_sublink：对应op ALL(select…)语句，其中OP可以是IN,<,>,=操作符
  – rowcompare_sublink：对应record op (select …)语句
  – expr_sublink：对应(SELECT with single targetlist item ...)语句
  – array_sublink：对应ARRAY(select…)语句
  – cte_sublink：对应with query(…)语句

  其中OLAP、HTAP场景中常用的sublink为exist_sublink、any_sublink，在DataArtsFabric SQL的优化引擎中对其应用场景做了优化（子链接提升），由于SQL语句中子查询的使用的灵活性，会带来SQL子查询过于复杂而造成的性能问题。子查询从大类上来看，分为非相关子查询和相关子查询：

  – **非相关子查询None-Correlated SubQuery**

    子查询的执行不依赖于外层父查询的任何属性值。这样子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解。

    例如：

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    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
```

  – **相关子查询Correlated-SubQuery**

子查询的执行依赖于外层父查询的一些属性值（如下列示例t2.c1 = t1.c1条件中的t1.c1）作为内层查询的一个AND-ed条件。这样的子查询不具备独立性，需要和外层查询按分组进行求解。

例如：

```
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
```

DataArtsFabric SQL 对 SubLink 的优化

针对SubLink的优化策略主要是让内层的子查询提升（pullup），能够和外表直接做关联查询，从而避免生成SubPlan+Broadcast内表的执行计划。判断子查询是否存在性能风险，可以通过explain查询语句查看Sublink的部分是否被转换成SubPlan+Broadcast的执行计划。

- **目前DataArtsFabric SQL支持的Sublink-Release场景**
  - IN-Sublink无相关条件
    - 不能包含上一层查询的表中的列（可以包含更高层查询表中的列）。
    - 不能包含易变函数。
  - Exist-Sublink包含相关条件

    Where子句中必须包含上一层查询的表中的列，子查询的其它部分不能含有上层查询的表中的列。其它限制如下：
    - 子查询必须有from子句。
    - 子查询不能含有with子句。
    - 子查询不能含有聚集函数。
    - 子查询里不能包含集合操作、排序、limit、windowagg、having操作。
    - 不能包含易变函数。
  - 包含聚集函数的等值相关子查询的提升

    子查询的where条件中必须含有来自上一层的列，而且此列必须和子查询本层涉及表中的列做相等判断，且这些条件必须用and连接。其它地方不能包含上层的列。其它限制条件如下：
    - 子查询中where条件包含的表达式（列名）必须是表中的列。
    - 子查询的Select关键字后，必须有且仅有一个输出列，此输出列必须是聚集函数（如max），并且聚集函数的参数（t2.c2）不能是来自外层表（t1）中的列。聚集函数不能是count。

下列示例可以提升：

```
select * from t1 where c1 >(
       select max(t2.c1) from t2 where t2.c1=t1.c1
);
```

下列示例不能提升，因为子查询没有聚集函数：

```
select * from t1 where c1 >(
       select  t2.c1 from t2 where t2.c1=t1.c1
);
```

下列示例不能提升，因为子查询有两个输出列：

```
select * from t1 where (c1,c2) >(
       select  max(t2.c1),min(t2.c2) from t2 where t2.c1=t1.c1
);
```

- 子查询必须是from子句。
- 子查询中不能有groupby、having、集合操作。
- 子查询只能是inner join。

  下列示例不能提升：

```
select * from t1 where c1 >(
       select max(t2.c1) from t2 full join t3 on (t2.c2=t3.c2) where t2.c1=t1.c1
);
```

- 子查询的targetlist中不能包含返回set的函数。
- 子查询的where条件中必须含有来自上一层的列，而且此列必须和子查询层涉及表中的列做相等判断，且这些条件必须用and连接。其它地方不能包含上层中的列。下列示例中的最内层子链接可以提升：

```
select * from t3 where t3.c1=(
        select t1.c1
        from t1 where c1 >(
                select max(t2.c1) from t2 where t2.c1=t1.c1
));
```

  基于上面的示例，再加一个条件，则不能提升，因为最内侧子查询引用了上层中的列。示例如下：

```
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

));
```

– 提升OR子句中的SubLink

当WHERE过滤条件中有OR连接的EXIST相关SubLink，

例如：

```
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);
```

将OR-ed连接的EXIST相关子查询OR子句的提升过程：

i. 提取where条件中，or子句中的opExpr。为：t1.a = (select avg(a) from t3 where t1.b = t3.b)
ii. 这个op操作中包含subquery，判断是否可以提升，如果可以提升，重写subquery为：select avg(a), t3.b from t3 group by t3.b，生成not null条件t3.b is not null，并将这个opexpr用这个not null条件替换。此时SQL变为：

```
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. 再次提取or子句中的exists sublink，exists (select * from t4 where t1.c = t4.c)，判断是否可以提升，如果可以提升，转换subquery为：select t4.c from t4 group by t4.c生成NotNull条件t4.c is not null提升查询，SQL变为：

```
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;
```

- **目前DataArtsFabric SQL不支持的Sublink-Release场景**

  除了以上场景之外都不支持Sublink提升，因此关联子查询会被计划成SubPlan+Broadcast的执行计划，当inner表的数据量较大时则会产生性能风险。

  如果相关子查询中跟外层的两张表做join，那么无法提升该子查询，需要通过将父SQL创建成with子句，然后再跟子查询中的表做相关子查询。

  例如：

```
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);
```

  改写为

```
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);
```

  – 出现在targetlist里的相关子查询无法提升(不含count)

    例如：

```
explain (costs off)
select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
from t1
where t1.c2 > 10;
```

    执行计划为：

```
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
```

    由于相关子查询出现在targetlist（查询返回列表）里，对于t1.c1=t2.c1不匹配的场景仍然需要输出值，因此使用left-outerjoin关联T1&T2确保t1.c1=t2.c1在不匹配时，子SSQ能够返回不匹配的补空值。

> **说明**
>
> SSQ和CSSQ的解释如下：
>
> - SSQ：ScalarSubQuery一般指返回1行1列scalar值的sublink，简称SSQ。
> - CSSQ：Correlated-ScalarSubQuery和SSQ相同不过是指包含相关条件的SSQ。

上述SQL语句可以改写为：

```
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;
```

改写后的执行计划为：

```
                                   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
```

可以看到出现在SSQ返回列表里的相关子查询SSQ，已经被提升成Right Join，从而避免当内表T2较大时出现SubPlan+Broadcast计划导致性能变差。

– 出现在targetlist里的相关子查询无法提升(带count)

例如：

```
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;
```

执行计划为：

```
                                          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
```

由于相关子查询出现在targetlist（查询返回列表）里，对于t1.c1=t2.c1不匹配的场景仍然需要输出值，因此使用left-outerjoin关联T1&T2确保t1.c1=t2.c1在不匹配时子SSQ能够返回不匹配的补空值，但是这里带了count语句及时在t1.c1=t2.t1不匹配时需要输出0，因此可以使用一个case-when NULL then 0 else count(*)来代替。

上述SQL语句可以改写为：

```
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;
```

改写后的执行计划为：

```
                                       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
```

– 相关条件为不等值场景

  例如：

```
select t1.c1, t1.c2
from t1
where t1.c1 = (select agg() from t2.c2 > t1.c2);
```

  对于非等值相关条件的SubLink目前无法提升，从语义上可以通过做2次join（一次CorrelationKey，一次rownum自关联）达到提升改写的目的。

  改写方案有两种：

  - 子查询改写方式

```
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改写方式

```
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;
```

须知

- 目前DataArtsFabric SQL尚无高效的实现表、中间结果集的全局唯一rowid因此目前此类场景很难改写，建议通过业务层进行规避，或者可以使用t1.xc_node_id + t1.ctid进行rowid关联，但是xc_node_id的重复率较高会导致join关联效率变低，而xc_node_id+ctid类型无法作为hashjoin的关联条件。
- 对于AGG类型为count(*)时需要进行CASE-WHEN对没有match的场景补0处理，非COUNT(*)场景NULL处理。
- CTE改写方式如果有sharescan支持性能上能够更优。

更多优化示例

示例：修改select语句，将子查询修改为和主表的join，或者修改为可以提升的subquery，但是在修改前后需要保证语义的正确性。

```
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
```

上面示例计划中存在一个subPlan，为了消除这个subPlan可以修改语句为：

```
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
```

从计划可以看出，subPlan消除了，计划变成了两个表的semi join，这样会大幅度提高执行效率。

5.3.7.2 算子级调优

一个查询语句要经过多个算子步骤才会输出最终的结果。由于个别算子耗时过长导致整体查询性能下降的情况比较常见。这些算子是整个查询的瓶颈算子。通用的优化手段是EXPLAIN ANALYZE/PERFORMANCE命令查看执行过程的瓶颈算子，然后进行针对性优化。

如下面的执行过程信息中，Hashagg算子的执行时间占总时间的：(66167-56217)/66878=14.8%，Foreign scan算子的执行时间占总时间的：56217/66878=84%，此处Foreign scan算子就是这个查询的瓶颈算子，在进行性能优化时应当优先考虑此算子的优化，如使用分区表。

```
---------------------------------------------------------------------------------------------------------------------------------------------------------
 id |                                  operation                                   |          A-time          |   A-rows   |   E-rows   |
----+------------------------------------------------------------------------------+--------------------------+------------+------------+
  1 | ->  Row Adapter                                                              | 66878.735                |          4 |          5 |
  2 |    ->  Vector Streaming (type: GATHER)                                       | 66878.719                |          4 |          5 |
  3 |       ->  Vector Sort                                                        | [66147.222, 66168.775]   |          4 |          5 |
  4 |          ->  Vector Sonic Hash Aggregate                                     | [66145.659, 66167.050]   |          4 |          6 |
  5 |             ->  Vector Streaming(type: REDISTRIBUTE dop: 1/4)                | [66143.236, 66165.115]   |        383 |        576 |
  6 |                ->  Vector Sonic Hash Aggregate                               | [30851.415, 66143.762]   |        383 |        576 |
  7 |                   ->  Partitioned Vector Foreign Scan on tpch_parquet_1000x_partition_perf.lineitem | [26009.452, 56217.193] | 5915582721 | 5918346062 |
```

同时，对于两个表的Join，如果数据量较大时，且选择了NestLoop，此时该算子性能会比较差。需要设置enable_nestloop=off，选择HashJoin，则性能可以得到较大提升。

5.3.7.3 SQL 语句改写规则

根据数据库的SQL执行机制以及大量的实践，总结发现：通过一定的规则调整SQL语句，在保证结果正确的基础上，能够提高SQL执行效率。如果遵守下列规则，能够大幅度提升业务查询效率。

- **使用union all代替union**

  union在合并两个集合时会执行去重操作，而union all则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用union all替代union以便提升性能。

- **join列增加非空过滤条件**

  如果join列上的NULL值较多，则可以加上is not null过滤条件，以实现数据的提前过滤，提高join效率。

- **选择hashagg。**

  查询中GROUP BY语句如果生成了groupagg+sort的plan性能会比较差，可以通过加大work_mem的方法生成hashagg的plan，因为不用排序而提高性能。

- **尝试将函数替换为case语句。**

  DataArtsFabric SQL函数调用性能较低，如果出现过多的函数调用导致性能下降很多，可以根据情况把可下推函数的函数改成CASE表达式。

- **避免对分区键使用函数或表达式运算。**

  对分区键使用函数或表达式运算会影响分区剪枝的应用，造成数据扫描量的增多。

- **尽量避免在where子句中使用!=或<>操作符、null值判断、or连接、参数隐式转换。**

- **对复杂SQL语句进行拆分。**

  对于过于复杂并且不易通过以上方法调整性能的SQL可以考虑拆分的方法，把SQL中某一部分拆分成独立的SQL并把执行结果存入临时表，拆分常见的场景包括但不限于：

  – 作业中多个SQL有同样的子查询，并且子查询数据量较大。
  – 函数（如subst，to_number）导致大数据量子查询选择度计算不准。
  – 多DN环境下对大表做broadcast的子查询。

5.3.8 优化器参数调整

本节将介绍影响DataArtsFabric SQL调优性能的关键CN配置参数，配置方法参见**查看和设置GUC参数**。

表 5-13 CN 配置参数

参数/参考值	描述
enable_nestloop=on	控制查询优化器对嵌套循环连接（Nest Loop Join）类型的使用。当设置为“on”后，优化器优先使用Nest Loop Join；当设置为“off”后，优化器在存在其他方法时将优先选择其他方法。 **说明** 如果只需要在当前数据库连接（即当前Session）中临时更改该参数值，则只需要在SQL语句中执行如下命令： `SET enable_nestloop to off;` 此参数默认设置为“on”，但实际调优中应根据情况选择是否关闭。一般情况下，在三种join方式（Nested Loop、Merge Join和Hash Join）里，Nested Loop性能较差，实际调优中可以选择关闭。
enable_hashagg=on	控制优化器对Hash聚集规划类型的使用。
enable_hashjoin=on	控制优化器对Hash连接规划类型的使用。
rewrite_rule	控制优化器是否启用特定组合的重写规则。

5.3.9 使用 Plan Hint 进行调优

5.3.9.1 Plan Hint 调优概述

Plan Hint为用户提供了直接影响执行计划生成的手段，用户可以通过指定join顺序，join、stream、scan方法，指定结果行数，指定重分布过程中的倾斜信息，指定配置参数的值等多个手段来进行执行计划的调优，以提升查询的性能。

功能描述

Plan Hint支持在SELECT、INSERT、UPDATE、MERGE、DELETE关键字后通过如下形式指定：

```
/*+ <plan hint> */
```

可以同时指定多个hint，之间使用空格分隔。hint只能hint当前层的计划，对于子查询计划的hint，需要在子查询对应的关键字后指定hint。

例如：

```
select /*+ <plan_hint1> <plan_hint2> */ * from t1, (select /*+ <plan_hint3> */ from t2) where 1=1;
```

其中<plan_hint1>，<plan_hint2>为外层查询的hint，<plan_hint3>为内层子查询的hint。

须知

如果在视图定义（CREATE VIEW）时指定hint，则在该视图每次被应用时会使用该hint。

当使用random plan功能（参数plan_mode_seed不为0）时，查询指定的plan hint不会被使用。

支持范围

当前版本Plan Hint支持的范围如下，后续版本会进行增强。

- 指定Join顺序的Hint - leading hint。
- 指定Join方式的Hint，仅支持除semi/anti join，unique plan之外的常用hint。
- 指定结果集行数的Hint。
- 指定Stream方式的Hint。
- 指定Scan方式的Hint，仅支持常用的tablescan，indexscan和indexonlyscan的hint。
- 指定子链接块名的Hint。
- 指定倾斜信息的Hint，仅支持Join与HashAgg的重分布过程倾斜。
- 指定Agg重分布列Hint。
- 指定子查询不提升的Hint。
- 指定配置参数值的Hint，仅支持部分配置参数，详见**配置参数的hint**。

注意事项

- 不支持Sort、Setop和Subplan的hint。
- 不支持对INSERT语句的目标表使用Hint。

示例

本章节使用同一个语句进行示例，便于Plan Hint支持的各方法做对比，示例语句及不带hint的原计划如下所示：

```
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
,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
```

5.3.9.2 Join 顺序的 Hint

功能描述

指明join的顺序，包括内外表顺序。

语法格式

- 单层圆括号()，仅指定join顺序，不指定内外表顺序。

```
leading(join_table_list)
leading(@block_name join_table_list)
```

- 双层圆括号(())，同时指定join顺序和内外表顺序，内外表顺序仅在最外层生效。

```
leading((join_table_list))
leading(@block_name (join_table_list))
```

- 单层方括号[]，同时指定[]这层的join顺序和内外表顺序。

```
leading[join_table_list]
leading[@block_name join_table_list]
```

- 单层圆括号()和单层方括号[]混合使用，同时指定join顺序和任意层内外表顺序。圆括号()这一层只指定join顺序，不指定内外表顺序，方括号[]这一层同时指定join顺序和内外表顺序。

```
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)]
```

须知

单层方括号[]可以和单层圆括号()混合使用，指定任意层的内外表顺序。不支持单层[]和双层(())一起使用。

参数说明

- **join_table_list**

  为表示表join顺序的hint字符串，可以包含当前层的任意个表（别名），或对于子查询提升的场景，也可以包含子查询的hint别名，同时任意表可以使用括号指定优先级，表之间使用空格分隔。

  join table list中指定的表需要满足以下要求，否则会报语义错误：

  - list中的表必须在当前层或提升的子查询中存在。
  - list中的表在当前层或提升的子查询中必须是唯一的。如果不唯一，需要使用不同的别名进行区分。
  - 同一个表只能在list里出现一次。
  - 如果表存在别名，则list中的表需要使用别名。

  说明

  - 表的语法格式如下：

    [schema.]table[@block_name]

    表名可以带schema，也可以带所在子查询语句块提升前的block_name。子查询语句块在优化器进行优化重写时发生提升，则该block_name会与leading中block_name不同。
  - 表如果存在别名，优先使用别名来表示该表。

- **block_name**

  语句块的block_name。表示该hint在block_name对应的子查询语句块中生效。

须知

- 默认为语句生成block_name。
- CN轻量化不生成block_name。
- create table as select、select into、select、insert、update、delete、merge语句生成block_name。
- block_name的命名规则：
  - 为select、insert、update、delete、merge语句自动生成blockname，block_name的命名格式分别为sel$n、ins$n、upd$n、del$n、mer$n，n从1开始计数，不同类型语句之间，计数不累加，同一类型语句之间计数累加。

    例如：

    ```
    INSERT INTO t SELECT * FROM t1 WHERE a1 IN (select * from t2);
    ```

    --------sel$2-------

    ----------------------sel$1---------------------

    -------------------------------ins$1---------------------------

  - 在优化器之前递归为每个语句块分配block_name。

    首先为当前语句块根据语句类型分配block_name，然后按照下面的顺序进行遍历，并为其中的语句块分配block_name：

    1. 遍历目标列
    2. 遍历merge语句的源表中的目标列
    3. 遍历merge语句中的action（update/insert）
    4. 遍历returning子句
    5. 遍历from中的join条件和where条件（join条件优先于where条件）
    6. 如果是集合操作，遍历集合的各个分支（union/intersect/except）
    7. 遍历having子句
    8. 遍历limit offset子句
    9. 遍历limit count子句
    10. 遍历cte
    11. 遍历from后的表
    12. 遍历upsert子句

  - 在优化器的重写阶段，由于fulljoin、cte inline、物化视图重写、inlist2join、or转换、multi count(distinct)、magicset、lazyagg、子查询/子链接提升等重写优化，都会构造新的子查询，此时会对新构造的子查询也应用上面分配block_name的递归处理，block_name的编号计数在之前的基础上进行累加。
  - 优化器重写阶段，发生子查询提升时，内层子查询中的表被提升到外层查询中，内层子查询被消除。此时，被提升的表可能会和外层查询中的表同名，所以在表中记录其原本所属block_name来区分来自不同查询块中的两个相同表。

例如：

leading(t1 t2 t3 t4 t5)表示：t1、t2、t3、t4、t5先join，五表join顺序及内外表不限。

leading((t1 t2 t3 t4 t5))表示：t1和t2先join，t2做内表；再和t3 join，t3做内表；再和t4 join，t4做内表；再和t5 join，t5做内表。

leading(t1 (t2 t3 t4) t5)表示：t2、t3、t4先join，内外表不限；再和t1, t5 join，内外表不限。

leading((t1 (t2 t3 t4) t5))表示：t2、t3、t4先join，内外表不限；在最外层，t1再和t2、t3、t4的join表join，t1为外表，再和t5 join，t5为内表。

leading((t1 (t2 t3) t4 t5)) leading((t3 t2))表示：t2、t3先join，t2做内表；然后再和t1 join，t2、t3的join表做内表；然后再依次跟t4、t5做join，t4、t5做内表。

leading[t1 [t2 t3]]等价于leading((t1 (t2 t3))) leading((t2 t3))。

leading(t1 [t2 t3])等价于leading(t1 t2 t3) leading((t2 t3))。

leading[@sel$1 t1@sel$1 [t2@sel$2 t3@sel$2]]表示：t2、t3位于子查询中，子查询被提升后，和t1进行join，其表示先t2和t3 join，t2为外表，t3为内表；然后再和t1 join，t1为外表，t2, t3的join表做内表。

示例

对示例中原语句使用如下hint:

```
explain
select /*+ leading((((((store_sales store) promotion) item) customer) ad2) store_returns) leading((store store_sales))*/ i_product_name product_name ...
```

该hint表示：表之间的join关系是：store_sales和store先join， store_sales做内表，然后依次跟promotion, item, customer, ad2, store_returns做join。生成计划如下所示：

```
 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
```

图中计划顶端warning的提示详见**Hint的错误、冲突及告警**的说明。

5.3.9.3 Join 方式的 Hint

功能描述

指明Join使用的方法，可以为Nested Loop，Hash Join和Merge Join。

语法格式

```
[no] nestloop|hashjoin|mergejoin([@block_name] table_list)
```

参数说明

- **no**表示hint的join方式不使用。
- **block_name**表示语句块的block_name，详细说明请参考**block_name**。

- **table_list**为表示hint表集合的字符串，该字符串中的表与**join_table_list**相同，只是中间不允许出现括号指定join的优先级。

例如：

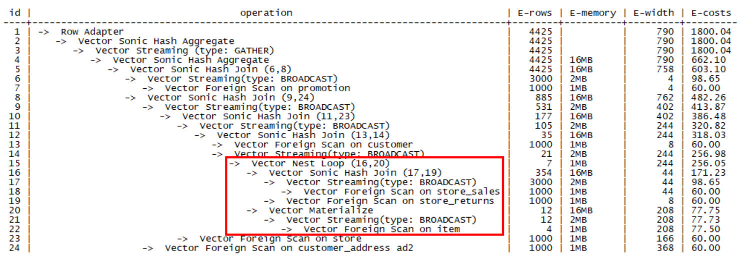
no nestloop(t1 t2 t3)表示：生成t1，t2，t3三表连接计划时，不使用nestloop。三表连接计划可能是t2 t3先join，再跟t1 join，或t1 t2先join，再跟t3 join。此hint只hint最后一次join的join方式，对于两表连接的方法不hint。如果需要，可以单独指定，例如：任意表均不允许nestloop连接，且希望t2 t3先join，则增加hint：no nestloop(t2 t3)。

示例

对示例中原语句使用如下hint:

```
explain
select /*+ nestloop(store_sales store_returns item) */ i_product_name product_name ...
```

该hint表示：生成store_sales，store_returns和item三表的结果集时，最后的两表关联使用nestloop。生成计划如下所示：



```
 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 | 77.50
 23 |                         ->  Vector Foreign Scan on store                    |   1000 | 1MB      |     166 | 60.00
 24 |                   ->  Vector Foreign Scan on customer_address ad2           |   1000 | 1MB      |     368 | 60.00
```

5.3.9.4 行数的 Hint

功能描述

指明中间结果集的大小，支持绝对值和相对值的hint。

语法格式

```
rows([@block_name] table_list #|+|-|* const)
```

参数说明

- **block_name**表示语句块的block_name，详细说明请参考**block_name**。
- **#,+,-,***，进行行数估算hint的四种操作符号。#表示直接使用后面的行数进行hint。+,-,*表示对原来估算的行数进行加、减、乘操作，运算后的行数最小值为1行。table_list为hint对应的单表或多表join结果集，与**Join方式的Hint**中table_list相同。
- **const**可以是任意非负数，支持科学计数法。

例如：

rows(t1 #5)表示：指定t1表的结果集为5行。

rows(t1 t2 t3 *1000)表示：指定t1, t2, t3 join完的结果集的行数乘以1000。

建议

- 推荐使用两个表*的hint。对于两个表的采用*操作符的hint，只要两个表出现在join的两端，都会触发hint。例如：设置hint为rows(t1 t2 * 3)，对于(t1 t3 t4)和(t2 t5 t6)join时，由于t1和t2出现在join的两端，所以其join的结果集也会应用该hint规则乘以3。
- rows hint支持在单表、多表、function table及subquery scan table的结果集上指定hint。

示例

对示例中原语句使用如下hint:

```
explain
select /*+ leading(store_sales store_returns) rows(store_sales store_returns *50) */ i_product_name
product_name ...
```

该hint表示：store_sales，store_returns关联的结果集估算行数在原估算行数基础上乘以50。生成计划如下所示：

```
 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
```

第11行算子的估算行数修正为17700行，原估算行数为354行。

5.3.9.5 Stream 方式的 Hint

功能描述

指明stream使用的方法，可以为broadcast和redistribute以及指定AGG重分布的分布键。

> 📖 **说明**
>
> 指定Agg重分布列Hint，仅8.1.3.100及以上集群版本支持。

语法格式

```
[no] broadcast | redistribute([@block_name] table_list) | redistribute ([@block_name] (*) (columns))
```

参数说明

- **no**表示hint的stream方式不使用，当指定AGG分布列的hint时指定no关键字无效。
- **block_name**表示语句块的block_name，详细说明请参考**block_name**。
- **table_list**为进行stream操作的单表或多表join结果集，见**参数说明**。
- 当指定分布列的hint时，*为固定写法，不支持指定表名。
- **columns**指定group by子句中的一个或者多个列，没有group by子句也可以指定distinct子句中的列。

说明

- 指定的分布列，需要用group by或distinct中的列序号或列名来表示，count（distinct）中的列只能通过列名指定。
- 对于多层的查询，可以在每层指定对应层的分布列hint，只在当前层生效。
- 指定的count(distinct)列仅针对生成双层hashagg的计划时才生效，否则指定的分布列无效。
- 指定了分布列，如果优化器估算后发现不需要重分布，则指定的分布列无效。

建议

- 通常优化器会根据统计信息选择一组不倾斜的分布键进行数据重分布。当默认选择的分布键有倾斜时，可以手动指定重分布的列，避免数据倾斜。
- 在选择分布键的时候，通常要根据数据分布特征选取一组distinct值比较高的列作为分布列，这样可以保证重分布后，数据均匀的分布到各个DN。
- 在编写好hint后，可以通过explain verbose+SQL打印执行计划，查看指定的分布键是否有效，如果指定的分布键无效会有warning提示。

示例

- 对示例中原语句使用如下hint：

```
explain
select /*+ no redistribute(store_sales store_returns item store) leading(((store_sales store_returns item store) customer)) */ i_product_name product_name ...
```

原计划中，(store_sales store_returns item store)和customer做join时，前者做了重分布，此hint表示禁止前者混合表做重分布，但仍然保持join顺序，则生成计划如下所示：

```
 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
```

- 指定Agg重分布的分布列。

```
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
```

通过下面的示例可以看到指定的group by列的后两列作为分布键。

```
                               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)
```

- 当语句中不包含group by子句时，指定distinct列作为重分布列。

```
explain (verbose on, costs off, nodes off)
select /*+ redistribute ((*) (3 1)) */ distinct a1, b1, c1 from t1;
```

```
                        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)
```

5.3.9.6 子链接块名的 hint

功能描述

指明子链接块的名称。

语法格式

```
blockname ([@block_name] table)
```

注意事项

- block_name hint仅在对应的子链接块提升时才会被上层查询使用。目前支持的子链接提升包括IN子链接提升、EXISTS子链接提升和包含Agg等值相关子链接提升。该hint通常会和前面章节提到的hint联合使用。
- 对于FROM关键字后的子查询，则需要使用子查询的别名进行hint，block_name hint不会被用到。
- 如果子链接中含有多个表，则提升后这些表可与外层表以任意优化顺序连接，hint也不会被用到。

参数说明

- **block_name**表示语句块的block_name，详细说明请参考**block_name**。
- **table**表示为该子链接块hint的别名的名称。

说明

- 表的语法格式如下：

  [schema.]table[@block_name]

  表名可以带schema，也可以带所在子查询语句块提升前的block_name。子查询语句块在优化器进行优化重写时发生提升，则该block_name会与leading中block_name不同。
- 表如果存在别名，优先使用别名来表示该表。

示例

```
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);
```

该hint表示：子链接的别名为tt，提升后与上层的store_sales表关联时使用nestloop。生成计划如下所示：

```
 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
```

5.3.9.7 指定子查询不提升的 hint

功能描述

优化器在对查询进行逻辑优化时通常会将可以提升的子查询提升到上层以避免嵌套执行，但对于某些场景，嵌套执行不会导致性能下降过多，而提升之后扩大了查询路径的搜索范围，可能导致性能变差。对于此类情况，可以使用no merge hint指定子查询不提升进行调试。大多数情况下不建议使用此hint。

语法格式

```
no merge[@block_name]
no merge ([@block_name1] subquery_name[@block_name2])
```

参数说明

- **block_name**表示语句块的block_name，详细说明请参考**block_name**。
- **subquery_name**为目标子查询名，亦可以是view或cte名，表示该子查询在逻辑优化时不会进行提升；当不指定subquery_name时，表示当前查询不提升。

示例

创建表t1、t2、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;
```

原语句为：

```
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
```

上述查询中，可以使用以下两种方式禁止子查询s1进行提升：

- 方式一：
  ```
  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;
  ```
- 方式二：
  ```
  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;
  ```

提升后效果：

```
 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
```

5.3.9.8 配置参数的 hint

功能描述

指明计划生成时配置参数的值，又称作guc hint。

注意事项

- 如果hint设置的配置参数在语句级别生效，则该hint必须写在顶层查询中，而不能写在子查询中。对于UNION、INTERSECT、EXCEPT和MINUS语句，可以将在语句级别的guc hint写在参与集合运算的任意一个SELECT子句上，该guc hint设置的配置参数会在参与集合运算的每个SELECT子句上生效。

- 子查询提升时，该子查询上的所有guc hint会被丢弃。
- 如果一个配置参数既被语句级别的guc hint设置，又被子查询级别的guc hint设置，则子查询级别的guc hint在对应的子查询中生效，语句级别的guc hint在语句的其它子查询中生效。

语法格式

```
set [global]([@block_name] guc_name guc_value)
```

参数说明

- **global**表示hint设置的配置参数在语句级别生效，不加global表示hint设置的配置参数在子查询级别生效，即仅在hint所在的子查询中生效，在该语句的其它子查询中不生效。
- **block_name**表示语句块的block_name，详细说明请参考**block_name**。
- **guc_name**表示hint指定的配置参数的名称。
- **guc_value**表示hint指定的配置参数的值。

guc hint当前仅支持部分配置参数，并且有些配置参数不支持在子查询级别设置，只能在语句级别设置，以下为支持的参数列表：

表 5-14 guc hint 支持的配置参数

配置参数名	是否支持在子查询级别设置
agg_redistribute_enhancement	是
best_agg_plan	是
cost_model_version	否
cost_param	否
enable_broadcast	是
enable_redistribute	是
enable_extrapolation_stats	是
enable_hashagg	是
enable_hashjoin	是
enable_nestloop	是
from_collapse_limit	是
join_collapse_limit	是
join_num_distinct	是
qual_num_distinct	是
query_dop	否
rewrite_rule	否

配置参数名	是否支持在子查询级别设置
skew_option	是

示例

对示例中原语句使用如下hint：

```
explain
select /*+ set global(query_dop 0) */ i_product_name product_name
...
```

该hint表示：在整个语句的计划生成过程中，将配置参数query_dop设置为0，即打开SMP自适应功能。

5.3.9.9 Hint 的错误、冲突及告警

Plan Hint的结果会体现在计划的变化上，可以通过explain来查看变化。

Hint中的错误不会影响语句的执行，只是不能生效，该错误会根据语句类型以不同方式提示用户。对于explain语句，hint的错误会以warning形式显示在界面上，对于非explain语句，会以debug1级别日志显示在日志中，关键字为PLANHINT。

hint 的错误类型

- 语法错误

  语法规则树归约失败，会报错，指出出错的位置。

  例如：hint关键字错误，leading hint或join hint指定2个表以下，其它hint未指定表等。一旦发现语法错误，则立即终止hint的解析，所以此时只有错误前面的解析完的hint有效。

  例如：

  ```
  leading((t1 t2)) nestloop(t1) rows(t1 t2 #10)
  ```

  nestloop(t1)存在语法错误，则终止解析，可用hint只有之前解析的leading((t1 t2))。

- 语义错误
  - 表不存在，存在多个，或在leading或join中出现多次，均会报语义错误。
  - scanhint中的index不存在，会报语义错误。
  - 另外，如果子查询提升后，同一层出现多个名称相同的表，且其中某个表需要被hint，hint会存在歧义，无法使用，需要为相同表增加别名规避。

- hint重复或冲突

  如果存在hint重复或冲突，只有第一个hint生效，其它hint均会失效，会给出提示。

  - hint重复是指，hint的方法及表名均相同。例如：nestloop(t1 t2) nestloop(t1 t2)。
  - hint冲突是指，table list一样的hint，存在不一样的hint，hint的冲突仅对于每一类hint方法检测冲突。

    例如：nestloop (t1 t2) hashjoin (t1 t2)，则后面与前面冲突，此时hashjoin的hint失效。注意：nestloop(t1 t2)和no mergejoin(t1 t2)不冲突。

须知

leading hint中的多个表会进行拆解。例如：leading ((t1 t2 t3))会拆解成：leading((t1 t2)) leading(((t1 t2) t3))，此时如果存在leading((t2 t1))，则两者冲突，后面的会被丢弃。（例外：指定内外表的hint如果与不指定内外表的hint重复，则始终丢弃不指定内外表的hint。）

- 子链接提升后hint失效

  子链接提升后的hint失效，会给出提示。通常出现在子链接中存在多个表连接的场景。提升后，子链接中的多个表不再作为一个整体出现在join中。

- 列类型不支持重分布
  - 对于skew hint来说，目的是为了进行重分布时的调优，所以当hint列的类型不支持重分布时，hint将无效。
- hint未被使用
  - 非等值join使用hashjoin hint或mergejoin hint
  - 不包含索引的表使用indexscan hint或indexonlyscan hint
  - 通常只有在索引列上使用过滤条件才会生成相应的索引路径，全表扫描将不会使用索引，因此使用indexscan hint或indexonlyscan hint将不会使用
  - indexonlyscan只有输出列仅包含索引列才会使用，否则指定时hint不会被使用
  - 多个表存在等值连接时，仅尝试有等值连接条件的表的连接，此时没有关联条件的表之间的路径将不会生成，所以指定相应的leading，join，rows hint将不使用，例如：t1 t2 t3表join，t1和t2，t2和t3有等值连接条件，则t1和t3不会优先连接，leading(t1 t3)不会被使用。
  - 生成stream计划时，如果表的分布列与join列相同，则不会生成redistribute的计划；如果不同，且另一表分布列与join列相同，只能生成redistribute的计划，不会生成broadcast的计划，指定相应的hint则不会被使用。
  - 对于AGG重分布列的hint，hint未被使用的可能原因如下：
    - 指定的分布键包含不支持重分布的数据类型。
    - 执行计划中不需要重分布。
    - 执行的分布键的序号有误。
    - 对于使用grouping sets子句和cube子句的AP函数，window agg中的分布键，不支持hint。

      说明

      指定Agg重分布列Hint，仅8.1.3.100及以上集群版本支持。
  - 如果子链接未被提升，则blockname hint不会被使用。
  - 对于skew hint，hint未被使用的可能原因如下：
    - 计划中不需要进行重分布。
    - hint指定的列为包含分布键。
    - hint指定倾斜信息有误或不完整，如对于join优化未指定值。

■ 倾斜优化的GUC参数处于关闭状态。

– 对于guc hint，hint未被使用的可能原因如下：

■ 配置参数不存在。

■ 配置参数不支持guc hint。

■ 配置参数的值无效。

■ 语句级别的guc hint没有被写在顶层查询中。

■ 子查询级别的guc hint设置的配置参数不支持在子查询级别设置。

■ guc hint所在的子查询被提升。

5.4 SQL 调优案例

5.4.1 案例：调整 GUC 参数 best_agg_plan

现象描述

t1的表定义为：

```
create table t1(a int, b int, c int) store as orc;
```

假设agg下层算子所输出结果集的分布列为setA，agg操作的group by列为setB，则在Stream框架下，Agg操作可以分为两个场景。

场景一：setA是setB的一个子集。

对于这种场景，直接对下层结果集进行汇聚的结果就是正确的汇聚结果，上层算子直接使用即可。

场景二：setA不是setB的一个子集。

对于这种场景，Stream执行框架分为如下三种计划形态：

hashagg+gather(redistribute)+hashagg

redistribute+hashagg(+gather)

hashagg+redistribute+hashagg(+gather)

DataArtsFabric SQL提供了guc参数best_agg_plan来干预执行计划，强制其生成上述对应的执行计划，此参数取值范围为0，1，2，3

- 取值为1时，强制生成第一种计划。
- 取值为2时，如果group by列可以重分布，强制生成第二种计划，否则生成第一种计划。
- 取值为3时，如果group by列可以重分布，强制生成第三种计划，否则生成第一种计划。
- 取值为0时，优化器会根据以上三种计划的估算代价选择最优的一种计划生成。

具体影响如下：

```
postgres=> set best_agg_plan=1;
SET
postgres=> 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)

postgres=> set best_agg_plan=2;
SET
postgres=> 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)

postgres=> set best_agg_plan=3;
SET
postgres=> 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)
```

总结

通常优化器总会选择最优的执行计划，但是众所周知代价估算，尤其是中间结果集的代价估算一般会有比较大的偏差，这种比较大的偏差就可能会导致agg的计算方式出现比较大的偏差，这时候就需要通过best_agg_plan进行agg计算模型的干预。

一般来说，当agg汇聚的收敛度很小时，即结果集的个数在agg之后并没有明显变少时（经验上以5倍为临界点），选择redistribute+hashagg执行方式，否则选择hashagg+redistribute+hashagg执行方式。

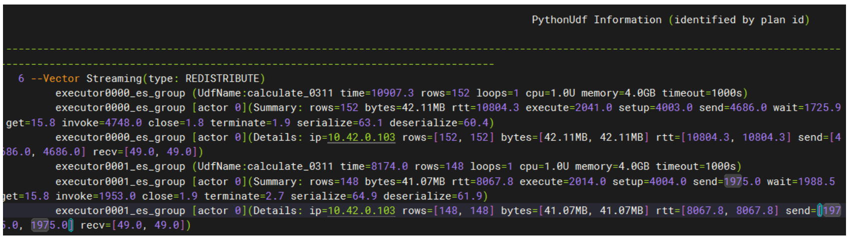
5.5 Python UDF 性能调优

DataArtsFabric SQL支持统计运行时的关键性能指标，根据性能指标，用户可以调整UDF运行时的资源规格和并行度。

UDF 性能监控

使用explain performance可以打印UDF Actor运行的性能指标。如**图5-7**所示，统计了函数calculate_0311在每个DN所对应的UDF Actor的性能情况。下文以图中三行性能数据为例进行说明。

图 5-7 Python UDF Actor 的性能指标统计



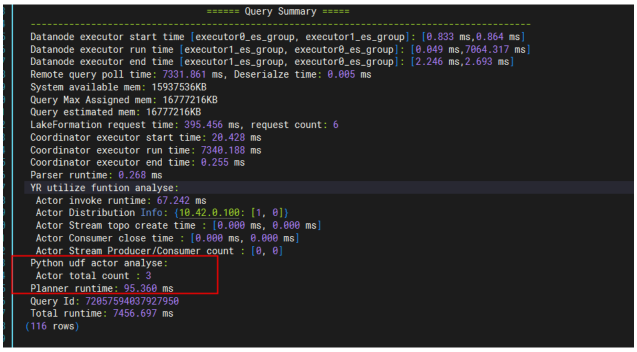
- executor0000_es_group (UdfName:calculate_0311 time=10907.3 rows=152 loops=1 cpu=1.0U memory=4.0GB timeout=1000s)中：
  – executor0_es_group指的是DN ID。
  – UdfName表示执行UDF的名称。
  – time描述该UDF总运行时间。
  – rows表示在该DN上处理的数据量。
  – loops表示调用UDF的次数。
  – cpu表示该UDF执行时actor所申请的cpu大小，单位为U。
  – memory表示该UDF执行时actor所申请的memory大小，单位为GB。
  – timeout表示该UDF执行时actor单次batch的最大执行时长，单位为秒。超时则会报错。

- executor0000_es_group [actor 0](Summary: rows=152 bytes=42.11MB rtt=10804.3 execute=2041.0 setup=4003.0 send=4686.0 wait=1725.9 get=15.8 invoke=4748.0 close=1.8 terminate=1.9 serialize=63.1 deserialize=60.4) 描述的是第0个UDF Actor的信息，具体如下：
  - rows表示该actor上处理的数据量。
  - bytes表示该actor处理的数据量大小。
  - rtt表示内核发送数据到actor，以及actor接收处理数据的端到端时间。
  - execute表示该actor实际计算的有效时间。
  - setup表示该actor在初始化后，准备环境和装载udf代码的时间。
  - send表示从内核发送数据到该actor的时间。
  - wait表示内核等待actor返回数据的等待时间。
  - get表示内核获取actor数据成功获取数据的时间。
  - Invoke表示拉起actor的时间。
  - close表示该actor关闭计算的时间。
  - terminate表示数据系统下线actor的耗时。
  - serialize表示对数据序列化的时间。
  - deserialize表示对数据反序列化的时间。
- executor0000_es_group [actor 0](Details: ip=10.42.0.103 rows=[152, 152] bytes=[42.11MB, 42.11MB] rtt=[10804.3, 10804.3] send=[4686.0, 4686.0] recv=[49.0, 49.0])统计信息中：
  - ip表示actor所在的ip地址。
  - rows表示actor执行的所有batch中最小行数和最大行数。
  - bytes表示actor执行的所有batch中数据大小最小值和最大值。
  - rtt表示actor执行的所有batch中端到端执行时间的最小值和最大值。
  - send表示actor执行的所有batch中从内核发送数据到该actor的最小值和最大值。
  - recv表示actor执行的所有batch中从python发送数据到内核接收的最小值和最大值。

如下图所示，Query Summary中统计了该查询中所有UDF Actor的汇总信息。

Actor total count表示当前query调用的所有actor数量。

图 5-8 Python UDF Actor 数据流传输信息总览



6 系统表

6.1 PG_AGGREGATE

PG_AGGREGATE系统表存储与聚集函数有关的信息。PG_AGGREGATE里的每条记录都是一条pg_proc里面的记录的扩展。PG_PROC记录承载该聚集的名字、输入和输出数据类型，以及其它一些和普通函数类似的信息。

表 6-1 PG_AGGREGATE 字段

名字	类型	引用	描述
aggfnoid	regproc	PG_PROC.oid	此聚集函数的PG_PROC OID。
aggtransfn	regproc	PG_PROC.oid	转换函数。
aggcollectfn	regproc	PG_PROC.oid	收集函数。
aggfinalfn	regproc	PG_PROC.oid	最终处理函数（如果没有则为0）。
aggsortop	oid	PG_OPERATOR.oid	关联排序操作符（如果没有则为0）。
aggtranstype	oid	PG_TYPE.oid	此聚集函数的内部转换（状态）数据的数据类型。
agginitval	text	-	转换状态的初始值。这是一个文本数据域，它包含初始值的外部字符串表现形式。如果数据域是null，则转换状态值从null开始。
agginitcollect	text	-	收集状态的初始值。这是一个文本数据域，它包含初始值的外部字符串表现形式。如果数据域是null，则收集状态值从null开始。

6.2 PG_ATTRIBUTE

PG_ATTRIBUTE系统表存储关于表字段的信息。

表 6-2 PG_ATTRIBUTE 字段

名称	类型	描述
attrelid	oid	该字段所属的表。
attname	name	字段名。
atttypid	oid	字段类型。
attstattarget	integer	控制ANALYZE为该字段设置的统计细节的级别。 ● 零值表示不收集统计信息。 ● 负数表示使用系统缺省的统计对象。 ● 正数值的确切信息是和数据类型相关的。 对于标量数据类型，ATTSTATTARGET既是要收集的“最常用数值”的目标数目，也是要创建的柱状图的目标数量。
attlen	smallint	是本字段类型pg_type.typlen的复制。
attnum	smallint	字段编号。
attndims	integer	如果该字段是数组，该值表示数组的维数，否则是0 。
attcacheoff	integer	在磁盘上总是-1 ，但是如果加载入内存中的行描述器中，它可能会被更新为缓冲在行中字段的偏移量。
atttypmod	integer	记录创建新表时支持的类型特定的数据（例如varchar字段的最大长度）。它传递给类型相关的输入和长度转换函数当做第三个参数。其值对那些不需要ATTTYPMOD的类型通常为-1。
attbyval	boolean	pg_type.typbyval字段值的复制。
attstorage	"char"	pg_type.typstorage字段值的复制。
attalign	"char"	pg_type.typalign字段值的复制。
attnotnull	boolean	代表一个非空约束。可以改变这个字段来打开或者关闭该约束。
atthasdef	boolean	该字段是否存在缺省值，此时它对应pg_attrdef表里实际定义此值的记录。
attisdropped	boolean	该字段是否已经被删除，不再有效。如果被删除，该字段物理上仍然存在表中，但会被分析器忽略，因此不能再通过SQL访问。

名称	类型	描述
attislocal	boolean	该字段是否局部定义在对象中。一个字段可以同时是局部定义和继承的。
attcmprmode	tinyint	对某一列指定压缩方式。压缩方式包括： ● ATT_CMPR_NOCOMPRESS ● ATT_CMPR_DELTA ● ATT_CMPR_DICTIONARY ● ATT_CMPR_PREFIX ● ATT_CMPR_NUMSTR
attinhcount	integer	该字段所拥有的直接父表的个数。如果一个字段的父表个数非零，则它就不能被删除或重命名。
attcollation	oid	对此列定义的校对列。
attacl	aclitem[]	列级访问权限控制。
attoptions	text[]	属性级可选项。
attfdwoptions	text[]	属性级外数据选项。
attinitdefval	bytea	存储了此列默认的值表达式。行存表的ADD COLUMN需要使用此字段。
attkvtype	tinyint	该字段的kv_type属性。取值如下： ● 0表示默认值，用于非时序表。 ● 1表示维度属性(TSTAG)，仅用于时序表。 ● 2表示指标属性(TSFIELD)，仅用于时序表。 ● 3时间属性(TSTIME)，仅用于时序表。

应用示例

查询指定表中包含的字段名和字段编号。t1和public分别替换为实际的表名和schema名称。

```
SELECT attname,attnum FROM pg_attribute WHERE attrelid=(SELECT pg_class.oid FROM pg_class JOIN
pg_namespace ON relnamespace=pg_namespace.oid WHERE relname='t1' and nspname='public') and
attnum>0;
     attname      | attnum
------------------+--------
 product_id       |      1
 product_name     |      2
 product_quantity |      3
(3 rows)
```

6.3 PG_AUTHID

PG_AUTHID系统表存储有关数据库认证标识符（角色）的信息。角色把“用户”的概念包含在内。一个用户实际上就是一个rolcanlogin标志被设置的角色。任何角色（不管rolcanlogin设置与否）都能够把其他角色作为成员。

在一个集群中只有一份pg_authid，不是每个数据库有一份。需要有系统管理员权限才可以访问此系统表。

表 **6-3** PG_AUTHID 字段

名称	类型	描述
oid	oid	行标识符（隐藏属性，必须明确选择才会显示）。
rolname	name	角色名称。
rolsuper	boolean	角色是否是拥有最高权限的初始系统管理员。
rolinherit	boolean	角色是否自动继承其所属角色的权限。
rolcreaterole	boolean	角色是否可以创建更多角色。
rolcreatedb	boolean	角色是否可以创建数据库。
rolcatupdate	boolean	角色是否可以直接更新系统表。只有usesysid=10的初始系统管理员拥有此权限。其他用户无法获得此权限。
rolcanlogin	boolean	角色是否可以登录，即该角色是否能够作为初始会话授权标识符。
rolreplication	boolean	角色是一个复制的角色（适配作用，没有实际的功能）。
rolauditadmin	boolean	审计用户。
rolsystemadmin	boolean	管理员用户。
rolconnlimit	integer	限制单个用户在单个CN上的最大并发连接数。-1表示没有限制。
rolpassword	text	口令(可能是加密的)，如果没有口令，则为NULL。
rolvalidbegin	timestamp with time zone	账户的有效开始时间，如果没有开始时间，则为NULL。
rolvaliduntil	timestamp with time zone	账户的有效结束时间，如果没有结束时间，则为NULL。
rolrespool	name	用户所能够使用的resource pool。

名称	类型	描述
roluseft	boolean	角色是否可以操作外表。
rolparentid	oid	用户所在组用户的OID。
roltabspace	Text	用户永久表存储空间限额。
rolkind	char	特殊用户种类，包括私有用户、逻辑集群管理员、普通用户。
rolnodegroup	oid	用户所关联的Node Group OID，该Node Group必须是逻辑集群。
roltempspace	Text	用户临时表存储空间限额。
rolspillspace	Text	用户算子落盘空间限额。
rolexcpdata	text	保留字段未使用。
rolpwdexpire	integer	用户口令过期时间，在口令未过期时，用户自己修改口令。过期后请管理员修改口令。-1表示没有过期时间限制。
rolpwdtime	timestamp with time zone	口令的创建时间。
roluuid	bigint	角色标识符。

6.4 PG_CLASS

PG_CLASS系统表存储数据库中所有内置系统表对象信息及其之间的关系。

表 6-4 PG_CLASS 字段

名称	类型	描述
oid	oid	行标识符（隐藏属性，必须明确选择才会显示）。
relname	name	表、索引、视图等对象的名称。
relnamespace	oid	包含该关系的命名空间的OID。
reltype	oid	与该表的行类型对应的数据类型（索引为零，因为索引没有pg_type记录）。
reloftype	oid	复合类型的OID，0表示其他类型。
relowner	oid	关系所有者。
relam	oid	如果行是索引，则就是所用的访问模式（B-tree，hash等）。
relfilenode	oid	该关系在磁盘上的文件的名称，如果没有则为0。

名称	类型	描述
reltablespace	oid	该关系存储所在的表空间。如果为0，则使用该数据库的缺省表空间。如果关系无磁盘文件，该字段无意义。
relpages	double precision	以页(大小为BLCKSZ)为单位的此表在磁盘上的大小，只是优化器使用的一个近似值。
reltuples	double precision	表中行的数目，只是优化器使用的一个估计值。
relallvisible	integer	被标识为全可见的表中的页数。此字段是优化器用来做SQL执行优化使用的。VACUUM、ANALYZE和一些DDL语句（例如，CREATE INDEX）会引起此字段更新。
reltoastrelid	oid	与此表关联的TOAST表的OID ，如果没有则为0。 TOAST表在一个从属表里“离线”存储大字段。
reltoastidxid	oid	对于TOAST表是它的索引的OID，如果不是TOAST表则为0。
reldeltarelid	oid	Delta表的OID。 Delta表附属于列存表。用于存储数据导入过程中的甩尾数据。
reldeltaidx	oid	Delta表的索引表OID。
relcudescrelid	oid	CU描述表的OID。 CU描述表（Desc表）附属于列存表。用于控制表目录中存储数据的可见性。
relcudescidx	oid	CU描述表的索引表OID。
relhasindex	boolean	如果对象是一个表且至少有（或者最近建有）一个索引，则为真。 由CREATE INDEX设置，但DROP INDEX不会立即将它清除。如果VACUUM进程检测一个表没有索引，会清理relhasindex字段，将relhasindex值设置为假。
relisshared	boolean	如果该表在整个集群中由所有数据库共享则为真。只有某些系统表（例如pg_database）是共享的。
relpersistence	"char"	• p表示永久表。 • u表示非日志表。 • t表示临时表。

名称	类型	描述
relkind	"char"	● r表示普通表。 ● i表示索引。 ● S表示序列。 ● v表示视图。 ● c表示复合类型。 ● t表示TOAST表。 ● f表示外表。 ● m表示物化视图。
relnatts	smallint	关系中用户字段数目（除了系统字段以外）。在pg_attribute里肯定有相同数目对应行。
relchecks	smallint	表上检查约束的数目。
relhasoids	boolean	如果为关系中每行都生成一个OID，则为真。
relhaspkey	boolean	如果该表有一个（或曾有）主键，则为真。
relhasrules	boolean	如果表有规则，则为真。
relhastriggers	boolean	如果表有（或曾有）触发器，则为真。
relhassubclass	boolean	如果表有（或曾有）任何继承的子表，则为真。
relcmprs	tinyint	表示是否启用表的压缩特性。需要特别注意，当且仅当批量插入才会触发压缩，普通的CRUD并不能够触发压缩。 ● 0表示其他不支持压缩的表（主要是指系统表，不支持压缩属性的修改操作）。 ● 1表示表数据的压缩特性为NOCOMPRESS或者无指定关键字。 ● 2表示表数据的压缩特性为COMPRESS。
relhasclusterkey	boolean	是否有局部聚簇存储。
relrowmovement	boolean	针对分区表进行update操作时，是否允许行迁移。 ● true：表示允许行迁移。 ● false：表示不允许行迁移。
parttype	"char"	表或者索引是否具有分区表的性质。 ● p表示带有分区表性质。 ● n表示没有分区表特性。 ● v表示该表为HDFS的Value分区表。

名称	类型	描述
relfrozenxid	xid32	该表中所有在此之间的事务ID已经被替换为一个固定的（"frozen"）事务ID。该字段用于跟踪表是否需要为了防止事务ID重叠（或者允许收缩pg_clog）而进行清理。如果该关系不是表则为0（InvalidTransactionId）。 为保持前向兼容，保留此字段，新增relfrozenxid64用于记录此信息。
reloptions	text[]	特定的访问方法选项，用"keyword=value"字符串形式表示。
relfrozenxid64	xid	该表中所有在此之前的事务ID已经被替换为一个固定的（"frozen"）事务ID。该字段用于跟踪表是否需要为了防止事务ID重叠（或者允许收缩pg_clog）而进行清理。如果该关系不是表则为0（InvalidTransactionId）。

6.5 PG_COLLATION

PG_COLLATION系统表描述可用的排序规则，本质上从一个SQL名字映射到操作系统本地类别。

表 6-5 PG_COLLATION 字段

名字	类型	引用	描述
oid	oid	-	行标识符（隐藏属性，必须明确选择才会显示）。
collname	name	-	排序规则名（每个命名空间和编码唯一）。
collnamespace	oid	PG_NAMESPACE.oid	包含该排序规则的命名空间的OID。
collowner	oid	PG_AUTHID.oid	排序规则的所有者。
collencoding	integer	-	排序规则可用的编码，如果适用于任意编码为-1。
collcollate	name	-	排序规则对象的LC_COLLATE。
collctype	name	-	排序规则对象的LC_CTYPE。

6.6 PG_DATABASE

PG_DATABASE系统表存储关于可用数据库的信息。

表 6-6 PG_DATABASE 字段

名称	类型	描述
datname	name	数据库名称。
datdba	oid	数据库所有者，通常为其创建者。
encoding	integer	数据库的字符编码方式。 pg_encoding_to_char()可以将此编号转换为编码名称。
datcollate	name	数据库使用的排序顺序。
datctype	name	数据库使用的字符分类。
datistemplate	boolean	是否允许作为模板数据库。
datallowconn	boolean	如果为假，则没有用户可以连接到这个数据库。这个字段用于保护template0数据库不被更改。
datconnlimit	integer	该数据库上允许的最大并发连接数，-1表示无限制。
datlastsysoid	oid	数据库里最后一个系统OID 。
datfrozenxid	xid32	用于跟踪该数据库是否需要为了防止事务ID重叠而进行清理。 为保持前向兼容，保留此字段，新增datfrozenxid64用于记录此信息。
dattablespace	oid	数据库的缺省表空间。
datcompatibility	name	数据库兼容模式。 ● ORA，表示兼容Oracle数据库。 ● TD，表示兼容Teradata数据库。 ● MySQL，表示兼容MySQL数据库。
datacl	aclitem[]	访问权限。
datfrozenxid64	xid	用于跟踪该数据库是否需要为了防止事务ID重叠而进行清理。

6.7 PG_FOREIGN_DATA_WRAPPER

PG_FOREIGN_DATA_WRAPPER系统表存储外部数据封装器定义。一个外部数据封装器是在外部服务器上驻留外部数据的机制，是可以访问的。

表 6-7 PG_FOREIGN_DATA_WRAPPER 字段

名字	类型	引用	描述
oid	oid	-	行标识符（隐藏属性，必须明确选择才会显示）。
fdwname	name	-	外部数据封装器名。
fdwowner	oid	PG_AUTHID.oid	外部数据封装器的所有者。
fdwhandler	oid	PG_PROC.oid	引用一个负责为外部数据封装器提供扩展例程的处理函数。如果没有提供处理函数则为0。
fdwvalidator	oid	PG_PROC.oid	引用一个验证器函数，这个验证器函数负责验证给予外部数据封装器的选项、外部服务器选项和使用外部数据封装器的用户映射的有效性。如果没有提供验证器函数则为0。
fdwacl	aclitem[]	-	访问权限。
fdwoptions	text[]	-	外部数据封装器指定选项，使用“keyword=value”格式的字符串。

6.8 PG_FOREIGN_SERVER

PG_FOREIGN_SERVER系统表存储外部服务器定义。一个外部服务器描述了一个外部数据源，例如一个远程服务器。外部服务器通过外部数据封装器访问。

表 6-8 PG_FOREIGN_SERVER 字段

名字	类型	引用	描述
oid	oid	-	行标识符（隐藏属性，必须明确选择才会显示）。
srvname	name	-	外部服务器名。
srvowner	oid	PG_AUTHID.oid	外部服务器的所有者。
srvfdw	oid	PG_FOREIGN_DATA_WRAPPER.oid	此外部服务器的外部数据封装器的OID。
srvtype	text	-	服务器的类型（可选）。
srvversion	text	-	服务器的版本（可选）。
srvacl	aclitem[]	-	访问权限。
srvoptions	text[]	-	外部服务器指定选项，使用“keyword=value”格式的字符串。

6.9 PG_NAMESPACE

PG_NAMESPACE系统表存储命名空间，即存储schema相关的信息。

表 6-9 PG_NAMESPACE 字段

名称	类型	描述
nspname	name	命名空间的名称。
nspowner	oid	命名空间的所有者。
nsptimeline	bigint	在DN上创建此命名空间时的时间线。此字段为内部使用，仅在DN上有效。
nspacl	aclitem[]	访问权限。具体请参见GRANT和REVOKE。
permspace	bigint	schema永久表空间限额。
usedspace	bigint	schema已用永久表空间大小。

6.10 PG_OPCLASS

PG_OPCLASS系统表定义索引访问方法操作符类。

每个操作符类为一种特定数据类型和一种特定索引访问方法定义索引字段的语义。一个操作符类本质上指定一个特定的操作符族适用于一个特定的可索引的字段数据类型。索引的字段实际可用的族中的操作符集是接受字段的数据类型作为它们的左边的输入的那个。

表 6-10 PG_OPCLASS 字段

名字	类型	引用	描述
oid	oid	-	行标识符（隐藏属性，必须明确选择才会显示）。
opcmethod	oid	PG_AM.oid	操作符类所属的索引访问方法。
opcname	name	-	操作符类的名称。
opcnamespace	oid	PG_NAMESPACE.oid	操作符类所属的命名空间。
opcowner	oid	PG_AUTHID.oid	操作符类所有者。
opcfamily	oid	PG_OPFAMILY.oid	包含此操作符类的操作符族。
opcintype	oid	PG_TYPE.oid	操作符类索引的数据类型。
opcdefault	boolean	-	如果操作符类是opcintype的缺省，则为真。

名字	类型	引用	描述
opckeytype	oid	PG_TYPE.oid	索引数据的类型，如果和opcintype相同则为0。

一个操作符类的opcmethod必须匹配包含它的操作符族的opfmethod。同样，对于任意给定的opcmethod和opcintype的组合，不能有超过一个PG_OPCLASS行有opcdefault为真。

6.11 PG_OPERATOR

PG_OPERATOR系统表存储有关操作符的信息。

表 6-11 PG_OPERATOR 字段

名字	类型	引用	描述
oid	oid	-	行标识符（隐藏属性，必须明确选择才会显示）。
oprname	name	-	操作符的名称。
oprnamespace	oid	PG_NAMESPACE.oid	包含此操作符的命名空间的OID。
oprowner	oid	PG_AUTHID.oid	操作符所有者。
oprkind	"char"	-	• b=infix =中缀（两边） • l=前缀（左边） • r=后缀（右边）
oprcanmerge	boolean	-	该操作符是否支持合并连接。
oprcanhash	boolean	-	该操作符是否支持Hash连接。
oprleft	oid	PG_TYPE.oid	左操作数的类型。
oprright	oid	PG_TYPE.oid	右操作数的类型。
oprresult	oid	PG_TYPE.oid	结果类型。
oprcom	oid	PG_OPERATOR.oid	此操作符的交换符（如果存在）。
oprnegate	oid	PG_OPERATOR.oid	此操作符的反转器（如果存在）。
oprcode	regproc	PG_PROC.oid	实现该操作符的函数。
oprrest	regproc	PG_PROC.oid	此操作符的约束选择性计算函数。

名字	类型	引用	描述
oprjoin	regproc	PG_PROC.oid	此操作符的连接选择性计算函数。

6.12 PG_PROC

PG_PROC系统表存储所有内置函数的信息。

表 6-12 PG_PROC 字段

名称	类型	描述
proname	name	函数名。
pronamespace	oid	此函数所在命名空间的OID。
proowner	oid	函数的所有者。
prolang	oid	实现语言或函数的调用接口。
procost	real	估计执行成本。
prorows	real	结果行估计数。
provariadic	oid	参数元素的数据类型。
protransform	regproc	此函数的简化调用方式。
proisagg	boolean	函数是否为聚集函数。
proiswindow	boolean	函数是否为窗口函数。
prosecdef	boolean	函数是否为一个安全定义器（例如，一个“setuid”函数）。
proleakproof	boolean	函数有无其他影响。如果函数没有对参数进行防泄露处理，则会抛出错误。
proisstrict	boolean	如果任意调用参数为空，函数是否返回空值。这种情况下函数实际上根本不会被调用。非“strict”的函数必须准备处理空值输入。
proretset	boolean	函数是否返回一个集合（即，指定数据类型的多个数值）。

名称	类型	描述
provolatile	"char"	说明该函数的结果是只依赖于它的输入参数，或者还会被外接因素影响。 ● i表示“不可变的”（immutable）函数，对于相同的输入总是输出相同的结果。 ● s表示“稳定的”（stable）函数，对于固定输入其结果在一次扫描里不变。 ● v表示“易变”（volatile）函数，其结果可能在任何时候都变化。
pronargs	smallint	参数个数。
pronargdefaults	smallint	有默认值的参数个数。
prorettype	oid	返回参数类型的OID。
proargtypes	oidvector	函数参数的数据类型的数组。数组里只包括输入参数（包括INOUT参数），因此也表现了函数的调用特征。
proallargtypes	oid[]	函数参数的数据类型的数组。数组里包括所有参数的类型（包括OUT和INOUT参数），如果所有参数都是IN参数，则这个字段就会为空。注意数组下标是以1为起点的，而因为历史原因，proargtypes的下标起点为0。
proargmodes	"char"[]	函数参数模式的数组。 ● i表示IN参数 ● o表示OUT参数 ● b表示INOUT参数 如果所有参数都是IN参数，则这个字段为空。注意此数组下标对应的是proallargtypes的位置，而不是proargtypes。
proargnames	text[]	函数参数的名字的数组。没有名字的参数在数组里设置为空字符串。如果没有一个参数有名字，这个字段为空。注意此数组的下标对应proallargtypes而不是proargtypes。
proargdefaults	pg_node_tree	默认值的表达式树。是PRONARGDEFAULTS元素的列表。
prosrc	text	描述函数或存储过程的定义。例如，对于解释型语言来说就是函数的源程序，或者一个链接符号，一个文件名，或者函数和存储过程创建时指定的其他任何函数体内容，具体取决于语言/调用习惯的实现。
probin	text	关于如何调用该函数的附加信息。同样，其含义也是和语言相关的。
proconfig	text[]	函数针对运行时配置变量的本地设置。

名称	类型	描述
proacl	aclitem[]	访问权限。具体请参见GRANT和REVOKE。
prodefaultargpos	int2vector	函数默认值的位置，不局限于能最后几个参数才可有默认值。
fencedmode	boolean	函数的执行模式，表示函数是在fence还是not fence模式下执行。如果是fence执行模式，函数的执行会在重新fork的进程中执行。默认值是fence。
proshippable	boolean	函数是否可以下推到DN上执行，默认值是false。 ● 对于IMMUTABLE类型的函数，函数始终可以下推到DN上执行。 ● 对于STABLE/VOLATILE类型的函数，仅当函数的属性是SHIPPABLE的时候，函数可以下推到DN执行。
propackage	boolean	该函数是否支持重载，主要针对Oracle风格的函数，默认值是false。

应用示例

查询指定函数的OID。例如，获取函数justify_days的OID为1295。

```
SELECT oid FROM pg_proc where proname ='justify_days';
 oid
------
 1295
(1 row)
```

查询指定函数是否为聚集函数。例如，查询justify_days函数为非聚集函数。

```
SELECT proisagg FROM pg_proc where proname ='justify_days';
 proisagg
----------
 f
(1 row)
```

6.13 PG_TYPE

PG_TYPE系统表存储数据类型的相关信息。

表 **6-13** PG_TYPE 字段

名称	类型	描述
typname	name	数据类型名称。
typnamespace	oid	此类型所在的命名空间的OID。

名称	类型	描述
typowner	oid	此类型的所有者。
typlen	smallint	对于定长类型是该类型内部表现形式的字节数。对于变长类型typlen为负值。 ● -1表示一种“变长”（有长度字属性的数据）。 ● -2表示一个以NULL结尾的C字符串。
typbyval	boolean	指定内部传递这个类型的数值时是传值还是传引用。如果该类型的TYPLEN不是1，2，4，8， TYPBYVAL最好为假。变长类型通常是传引用。即使TYPLEN允许传值，TYPBYVAL也可以为假。
typtype	"char"	● b表示基础类型。 ● c表示复合类型（例如一个表的行类型）。 ● e表示枚举类型。 ● p表示伪类型。 参见typrelid和typbasetype。
typcategory	"char"	数据类型的模糊分类，可用于解析器使用的数据转换依据。
typispreferred	boolean	如果为真，则数据符合TYPCATEGORY所指定的转换规则时进行转换。
typisdefined	boolean	如果定义了类型，则为真。如果是一种尚未定义的类型的占位符，则为假。如果为假，则除了该类型名称，命名空间和OID之外没有可依赖的对象。
typdelim	"char"	分析数组输入时，分隔两个此类型数值的字符。请注意，分隔符是与数组元素数据类型相关联，而不是与数组数据类型相关联。
typrelid	oid	如果是复合类型（请参见typtype），则此字段指向pg_class中定义该表的行。对于独立的复合类型，pg_class记录并不表示一个表，但是总需要它来查找该类型连接的pg_attribute记录。非复合类型为0。
typelem	oid	如果不为0，则它标识pg_type中的另一行。当前类型可以像一个产生类型为typelem的数组来描述。“true”数组类型是变长的（typlen= -1），但是某些定长（typlen > 0）类型也有非零的typelem（例如name和point）。如果一个定长类型有typelem，则其内部形式必须是typelem数据类型的某个数目的个数值，不能有其他数据。变长数组类型有一个该数组子过程定义的头（文件）。
typarray	oid	如果不为0，则表示在pg_type中有对应的类型记录。
typinput	regproc	输入转换函数（文本格式）。
typoutput	regproc	输出转换函数（文本格式）。

名称	类型	描述
typreceive	regproc	输入转换函数（二进制格式），如果没有则为0。
typsend	regproc	输出转换函数（二进制格式），如果没有则为0。
typmodin	regproc	输入类型修改符函数，如果为0，则不支持。
typmodout	regproc	输出类型修改符函数，如果为0，则不支持。
typanalyze	regproc	自定义的ANALYZE函数，如果使用标准函数，则为0。
typalign	"char"	当存储此类型的数值时要求的对齐方式。适用于磁盘存储以及该值在数据库中的大多数形式。如果数值是连续存储的，例如在磁盘上以完全的裸数据的形式存放时，则先在此类型的数据前填充空白，这样它就可以按照要求的边界存储。对齐引用是该序列中第一个数据的开头。可能的值包含： ● c = char对齐，即不需要对齐。 ● s = short对齐（在大多数机器上是2字节） ● i = int对齐（在大多数机器上是4字节） ● d = double对齐（在大多数机器上是8字节，但不一定是全部） **须知** 对于系统表里使用的类型，在pg_type里定义的尺寸和对齐方式要和编译器在表示表行的结构中布局方式保持一致。
typstorage	"char"	指明一个变长类型（那些有typlen = -1）是否准备好应付非常规值，以及对这种属性的类型的缺省策略是什么。可能的值包含： ● p：数值总是以简单方式存储。 ● e：数值可以存储在一个"次要"关系中（如果有该关系，请参见pg_class.reltoastrelid）。 ● m：数值可以以内联压缩方式存储。 ● x：数值可以以内联压缩方式或者在"次要"表里存储。 **须知** m域也可以移到从属表里存储，但只是最后的解决方法（首先移动e和x域）。
typenotnull	boolean	表示在某类型上的一个NOTNULL约束。目前只用于域。
typbasetype	oid	如果这是一个衍生类型（请参见typtype），则该标识作为这个类型的基础的类型。如果不是衍生类型则为零。
typtypmod	integer	域使用typtypmod记录要应用于其基础类型上的typmod（如果基础类型不使用typmod，则为-1）。如果此类型不是域，则为-1。

名称	类型	描述
typndims	integer	如果一个域是数组，则typndims是数组维数的数值（即typbasetype是一个数组类型；域的typelem将匹配基本类型的typelem）。除了数组类型的域以外的类型为0。
typcollation	oid	指定类型的排序规则。如果为0，则表示不支持排序规则。
typdefaultbin	pg_node_tree	如果不为NULL，则为该类型缺省表达式的nodeToString() 表现形式。目前这个字段只用于域。
typdefault	text	如果某类型没有相关缺省值，则为NULL。如果typdefaultbin不为NULL，则typdefault必须包含一个typdefaultbin代表的缺省表达式的人类可读版本。如果typdefaultbin为NULL但typdefault不为NULL，typdefault则是该类型缺省值的外部表现形式，可以将其输入到类型的转换器生成一个常量。

6.14 PGXC_CLASS

PGXC_CLASS系统表存储每张表的复制或分布信息。

表 **6-14** PGXC_CLASS 字段

名称	类型	描述
pcrelid	oid	表的OID。
pclocatortype	"char"	定位器类型。 ● H：hash ● M：Modulo ● N：Round Robin ● R：Replication
pchashalgorithm	smallint	使用哈希算法分布元组。
pchashbuckets	smallint	哈希容器的值。
pgroup	name	节点组名称。
redistributed	"char"	表已经完成重分布。
redis_order	integer	重分布的顺序。
pcattnum	int2vector	用作分布键的列标号。
nodeoids	oidvector_extend	表分布的节点OID列表。
options	text	系统内部保留字段，存储扩展状态信息。

6.15 PGXC_GROUP

PGXC_GROUP系统表存储节点组信息，在存算分离3.0版本中，每个逻辑集群节点组称为一个VW（Virtual Warehouse），而在存储KV层，每一个VW会和一个vgroup相对应。

表 6-15 PGXC_GROUP 字段

名称	类型	描述
group_name	name	节点组名称。
in_redistribution	"char"	是否需要重分布： ● n表示NodeGroup没有再进行重分布。 ● y表示NodeGroup是重分布过程中的源节点组。 ● t表示NodeGroup是重分布过程中的目的节点组。 ● s表示NodeGroup不需要重分布，重分布过程将跳过此节点组。
group_members	oidvector_extend	节点组的节点OID列表。
group_buckets	text	分布数据桶的集合。
is_installation	boolean	是否安装子集群。
group_acl	aclitem[]	访问权限。
group_kind	"char"	节点组类型： ● i表示安装节点组，包含所有DN节点。 ● n表示普通非逻辑集群节点组。 ● v表示逻辑集群节点组。 ● e表示弹性集群节点组 ● r表示复制表节点组，只能用于创建复制表，可以包含一个或多个逻辑集群节点组。
group_ckpt_csn	xid	节点组最近一次执行增量抽取的CSN。
vgroup_id	xid	节点组对应vgroup的ID标识。
vgroup_bucket_count	oid	节点组对应vgroup的桶数目。
group_ckpt_time	timestamp with time zone	节点组最近一次执行增量抽取的物理时间。

名称	类型	描述
apply_kv_duration	integer	节点组最近一次执行增量抽取中增量扫描耗时(单位为秒)。
ckpt_duration	integer	节点组最近一次执行增量抽取中checkpoint耗时（单位为秒）。

6.16 PGXC_NODE

PGXC_NODE系统表存储集群节点信息。

表 **6-16** PGXC_NODE 字段

名称	类型	描述
node_name	name	节点名称。
node_type	"char"	节点类型。 C：协调节点。 D：数据节点。
node_port	integer	节点的端口号。
node_host	name	节点的主机名称或者IP（如配置为虚拟IP，则为虚拟IP）。
node_port1	integer	复制节点的端口号。
node_host1	name	复制节点的主机名称或者IP（如配置为虚拟IP，则为虚拟IP）。
hostis_primary	boolean	表明当前节点是否发生主备切换。
nodeis_primary	boolean	在replication表下，是否优选当前节点作为优先执行的节点进行非查询操作。
nodeis_preferred	boolean	在replication表下，是否优选当前节点作为首选的节点进行查询。
node_id	integer	节点标识符。
sctp_port	integer	主节点使用TCP代理通信库或SCTP通信库的数据通道监听端口。
control_port	integer	主节点使用TCP代理通信库或SCTP通信库的控制通道监听端口。
sctp_port1	integer	备节点使用TCP代理通信库或SCTP通信库的数据通道监听端口。
control_port1	integer	备节点使用TCP代理通信库或SCTP通信库的控制通道监听端口。

名称	类型	描述
nodeis_central	boolean	当前节点为中心控制节点。

应用示例

查询集群的CN和DN信息：

```
SELECT * FROM pgxc_node;
  node_name   | node_type | node_port | node_host | node_port1 | node_host1 | hostis_primary | 
nodeis_primary | nodeis_preferred |    node_id
 | sctp_port | control_port | sctp_port1 | control_port1 | nodeis_central | read_only
--------------+-----------+-----------+-----------+------------+------------+----------------+----------------
+------------------+------------
-+-----------+--------------+------------+---------------+----------------+-----------
 datanode1    | D         |     55504 | localhost |      55504 | localhost  | t              | f              | f                | 
888802358
 |     55505 |        55507 |          0 |             0 | f              | f
 datanode2    | D         |     55508 | localhost |      55508 | localhost  | t              | f              | f                | 
-905831925
 |     55509 |        55511 |          0 |             0 | f              | f
 coordinator1 | C         |     55500 | localhost |      55500 | localhost  | t              | f              | f                | 
1938253334
 |         0 |            0 |          0 |             0 | t              | f
 datanode3    | D         |     55542 | localhost |      55542 | localhost  | t              | f              | f                | 
-1894792127
 |     57552 |        55544 |          0 |             0 | f              | t
 datanode4    | D         |     55546 | localhost |      55546 | localhost  | t              | f              | f                | 
-1307323892
 |     57808 |        55548 |          0 |             0 | f              | t
 datanode5    | D         |     55550 | localhost |      55550 | localhost  | t              | f              | f                | 
1797586929
 |     58064 |        55552 |          0 |             0 | f              | t
 datanode6    | D         |     55554 | localhost |      55554 | localhost  | t              | f              | f                | 
587455710
 |     58320 |        55556 |          0 |             0 | f              | t
 datanode7    | D         |     55558 | localhost |      55558 | localhost  | t              | f              | f                | 
-1685037427
 |     58576 |        55560 |          0 |             0 | f              | t
 datanode8    | D         |     55562 | localhost |      55562 | localhost  | t              | f              | f                | 
-993847320
 |     58832 |        55564 |          0 |             0 | f              | t
(9 rows)
```

7 GUC 参数

7.1 查看和设置 GUC 参数

为确保DataArtsFabric SQL的最优性能，用户可根据业务需求对数据库中的GUC参数进行调整。

数据库提供了许多运行参数，配置这些参数可以影响数据库系统的行为。在修改这些参数时请确保已了解对应参数对数据库的影响，否则可能会导致无法预料的结果。

注意事项

- 参数中如果取值范围为字符串，此字符串应遵循操作系统的路径和文件名命名规则。
- 取值范围最大值为INT_MAX的参数，此选项最大值跟所在的操作系统有关。
- 取值范围最大值为DBL_MAX的参数，此选项最大值跟所在的操作系统有关。

参数类型和值

- 所有的参数名称不区分大小写。 参数取值有整型、浮点型、字符串、布尔型和枚举型五类。

  布尔值可以是（on，off）、（true，false）、（yes，no）或者（1，0），且不区分大小写。
- 对于有单位的参数，在设置时请指定单位，否则将使用默认的单位。
  - 内存单位有：KB（千字节）、MB（兆字节）和GB（吉字节）。
  - 时间单位：ms（毫秒）、s（秒）、min（分钟）、h（小时）和d（天）。

查看和设置 GUC 参数

在DataArtsFabric SQL中，用户只能以SQL语句方式执行SET命令来设置GUC参数。具体格式如下

```
SET paramName TO paramValue;
```

例如：用户设置语句超时statement_timeout参数，可以在通过REST API接口发送SET statement_timeout TO 600 设置为10分钟。

如果用户要查看设置的GUC参数的值，可以通过SHOW语句来查看本Session设置的GUC参数值。例如：

```
SHOW statement_timeout;
```

语句可以查询设置的语句超时值。

7.2 连接和认证

7.2.1 安全和认证

session_timeout

参数说明：表明与服务器建立连接后，不进行任何操作的最长时间。

取值范围：整型，0-86400，最小单位为秒（s），0表示关闭超时设置。

默认值：10min

7.3 查询规划

7.3.1 优化器方法配置

以下配置参数提供了影响查询优化器选择查询规划的原始方法。如果优化器为特定的查询选择的缺省规划并不是最优的，可以通过使用这些配置参数强制优化器选择一个不同的规划来临时解决这个问题。更好的方法包括调节优化器开销常量、手动运行ANALYZE、增加配置参数default_statistics_target的值。

enable_bitmapscan

参数说明：控制优化器对位图扫描规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_hashagg

参数说明：控制优化器对Hash聚集规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_mixedagg

参数说明：控制优化器对Mixed Agg聚集规划类型的使用。（该参数仅8.2.0及以上集群版本支持）

取值范围：布尔型

- on表示使用，为符合条件的Grouping Sets语句（包括Rollup或Cube）生成Mixed Agg查询计划。
- off表示不使用。

默认值：on

须知

通常在大数量场景（单个DN表的数据量在100GB以上），采用Mixed Agg查询计划可以提升语句的执行性能。

不支持Mixed Agg的场景如下：

- GROUP BY子句里列的数据类型不支持哈希操作。
- 聚集函数中包含DISTINCT去重或ORDER BY排序。
- 使用GROUPING SETS子句时不包含空的分组。

enable_hashjoin

参数说明：控制优化器对Hash连接规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_indexscan

参数说明：控制优化器对索引扫描规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_indexonlyscan

参数说明：控制优化器对仅索引扫描规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_material

参数说明：控制优化器对实体化的使用。消除整个实体化是不可能的，但是可以关闭这个变量以防止优化器插入实体节点。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_mergejoin

参数说明：控制优化器对融合连接规划类型的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：off

enable_nestloop

参数说明：控制优化器对内表全表扫描嵌套循环连接规划类型的使用。完全消除嵌套循环连接是不可能的，但是关闭这个变量就会让优化器在存在其他方法的时候优先选择其他方法。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：off

left_join_estimation_enhancement

参数说明：控制left join的行数估计值是否采用优化后的值。该参数仅8.3.0.100及以上集群版本支持。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：off

enable_seqscan

参数说明：控制优化器对顺序扫描规划类型的使用。完全消除顺序扫描是不可能的，但是关闭这个变量会让优化器在存在其他方法的时候优先选择其他方法。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_sort

参数说明：控制优化器使用的排序步骤。完全消除明确的排序是不可能的，但是关闭这个变量可以让优化器在存在其他方法的时候优先选择其他方法。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

max_opt_sort_rows

参数说明：控制order by子句中最大优化的limit+offset行数。该参数仅8.3.0及以上集群版本支持。

取值范围：整型，0～INT_MAX

- 取值为0时：表示参数不生效。
- 取值为其他值时：表示order by子句中limit+offset行数小于该值时，优化生效，大于该值时，优化不生效。优化后耗时减小，但内存消耗可能增大。

默认值：0

enable_stream_concurrent_update

参数说明：控制优化器在并发更新场景下对stream的使用，该参数受限于**enable_stream_operator**参数。

取值范围：布尔型

- on表示允许优化器对update语句生成stream计划。
- off表示优化器对update语句仅能生成非stream计划。

默认值：on

enable_ctescan

参数说明：控制计划是否支持ctescan。

取值范围：布尔型

- on表示支持ctescan。
- off表示不支持ctescan。

默认值：on

enable_stream_operator

参数说明：控制优化器对stream的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_stream_recursive

参数说明：控制是否将with recursive关联查询下推DN分布式执行。

取值范围：布尔型

- on表示支持使用with recursive关联查询下推DN分布式执行。
- off表示不支持使用with recursive下推。

默认值：on

enable_value_redistribute

参数说明：控制是否开启生成value redistribute优化计划，8.2.0及以上集群版本中，该参数针对不带Partition by子句的rank、dense_rank、row_number是否生成value redistribute优化计划生效。

取值范围：布尔型

- on表示支持使用value redistribute生成优化计划。
- off表示不支持使用value redistribute生成优化计划。

默认值：off

max_recursive_times

参数说明：控制with recursive的最大迭代次数。

取值范围：整型，0～INT_MAX。

默认值：200

enable_vector_engine

参数说明：控制优化器对向量化执行引擎的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_broadcast

参数说明：控制优化器对stream代价估算时对broadcast分布方式的使用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_redistribute

参数说明：控制优化器对stream代价估算时对local redistribute和split redistribute分布方式的使用。该参数仅8.2.1.300及以上集群版本支持。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：on

enable_change_hjcost

参数说明：控制优化器在Hash Join代价估算路径选择时，是否使用将内表运行时代价排除在Hash Join节点运行时代价外的估算方式。如果使用，则有利于选择条数少，但运行代价大的表做内表。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：off

enable_fstream

参数说明：控制优化器下发语句时对stream的使用，该参数只限定于HDFS外表使用。

该参数现在已经废弃。为了保留前向兼容，可以设置成功，但是实际不起任何作用。

取值范围：布尔型

- on表示使用。
- off表示不使用。

默认值：off

enable_hashfilter

参数说明：控制是否允许对包含复制表(包括dual表和常量表)的计划生成hashfilter。该参数仅8.2.0及以上集群版本支持。

取值范围：布尔型

- on表示允许生成hashfilter。
- off表示任何情况下都不允许生成hashfilter。

默认值：on

best_agg_plan

参数说明：对于stream下的Agg操作，优化器会生成三种计划：

1. hashagg+gather(redistribute)+hashagg。
2. redistribute+hashagg(+gather)。
3. hashagg+redistribute+hashagg(+gather)。

本参数用于控制优化器生成哪种hashagg的计划。

取值范围：0，1，2，3

- 取值为1时，强制生成第一种计划。
- 取值为2时，如果group by列可以重分布，强制生成第二种计划，否则生成第一种计划。
- 取值为3时，如果group by列可以重分布，强制生成第三种计划，否则生成第一种计划。
- 取值为0时，优化器会根据以上三种计划的估算cost选择最优的一种计划生成。

默认值：0

turbo_engine_version

参数说明：对于建表指定turbo存储格式表（表属性中enable_turbo_store参数设置为on），且当查询不涉及merge join或sort agg算子时，执行器可走turbo执行引擎，执行器部分性能可获得成倍性能提升。

取值范围：0，1，2，3

- 取值为0时，表示turbo执行引擎关闭。
- 取值为1时，表示仅针对单表agg查询场景使用turbo执行引擎。
- 取值为2时，表示仅针对单表agg或多表join关联查询场景使用turbo执行引擎。
- 取值为3时，对于大多常用算子可使用turbo执行引擎加速，不支持算子如merge join，sort agg等算子。数据量较大且turbo_engine_version取值为3时，merge join，sort agg算子出现的情况较少，因此基本可以实现任意SQL语句的turbo执行引擎加速。

默认值：3

须知

跨VW场景暂不建议打开turbo执行引擎。

spill_compression

参数说明：控制执行器算子运行数据由于内存不足下盘时的压缩算法。该参数仅9.1.0.100及以上集群版本支持。

取值范围：枚举型

- 'lz4'，表示使用lz4压缩算法，对于下盘量较小场景性能更优，但占用更多存储空间。

- 'zstd'，表示使用zstd压缩算法，对于下盘量较大场景，IO为主要瓶颈，其性能更优，且存储空间约为lz4的2/3。

默认值：'lz4'

index_selectivity_cost

参数说明：控制列存表索引扫描时cbtree的cost计算（选择率>0.001），该参数仅8.2.1.1008.2.1.1及以上集群版本支持。

取值范围：浮点型，-1，0～1000，负数仅支持设置-1，且不允许设置其他负数。

- 取值为0时，不受索引选择率阈值0.001的影响。
- 取值为-1时，受disable_cost影响。
- 为其他值时，为cbtree cost计算的系数。

默认值：-1

index_cost_limit

参数说明：控制列存表索引扫描时cbtree的cost计算不生效阈值，该参数仅8.2.1.1008.2.1.1及以上集群版本支持。

取值范围：整型，0~2147483647。

- 取值为0时，表示参数不生效。
- 取值为其他值时，表行数少于该值时不受索引选择率阈值0.001的影响。

默认值：0

volatile_shipping_version

参数说明：控制volatile函数下推执行的范围。

取值范围：0，1，2，3

- 取值为3时，在2的基础上，扩展支持InlineCTE在只被引用一次时支持下推，扩展支持复制表UPSERT中使用包含volatile函数时的下推。
- 取值为2时，在1的基础上，扩展支持在复制CTE结果的目标列中包含volatile函数时的下推。
- 取值为1时，扩展支持nextval、uuid_generate_v1、sys_guid、uuid函数出现在语句目标列时的完全下推。
- 取值为0时，支持random类函数的完全下推，nextval、uuid_generate_v1函数仅在INSERT含有简单查询语句的部分下推。

默认值：3

agg_redistribute_enhancement

参数说明：当进行Agg操作时，如果包含多个group by列且均不为分布列，进行重分布时会选择某一group by列进行重分布。本参数控制选择重分布列的策略。

取值范围：布尔型

- on表示会选择估算distinct值最多的一个可重分布列作为重分布列。
- off表示会选择第一个可重分布列为重分布列。

默认值：off

enable_valuepartition_pruning

参数说明：是否对DFS分区表进行静态/动态优化。

取值范围：布尔型

- on表示对DFS分区表进行静态/动态优化。
- off表示不对DFS分区表进行静态/动态优化。

默认值：on

stream_multiple

参数说明：设置优化器计算Stream算子的开销时的加权。

在原代价模型的基础上，最终Stream代价将被乘以此加权参数。

取值范围：浮点型，0～10000。

默认值：1

> **须知**
>
> 此参数仅对Redistribute和Broadcast类型的Stream有效。

qrw_inlist2join_optmode

参数说明：控制是否使用inlist-to-join查询重写。

取值范围：字符串

- disable：关闭inlist2join查询重写。
- cost_base：基于代价的inlist2join查询重写。
- rule_base：基于规则的inlist2join查询重写，即强制使用inlist2join查询重写。
- 任意正整数：inlist2join查询重写阈值，即list内元素个数大于该阈值，进行inlist2join查询重写。

默认值：disable

enable_inlist_hashing

参数说明：控制是否使用inlist哈希优化。该参数仅9.1.0及以上集群版本支持。

取值范围：布尔型

- on表示开启inlist哈希优化。
- off表示关闭inlist哈希优化。

默认值：on

setop_optmode

参数说明：控制不带ALL选项的集合操作(UNION/EXCEPT/INTERSECT)语句的各个查询分支语句是否执行去重操作。该参数仅8.2.0及以上集群版本支持。

取值范围：枚举型

- disable：查询分支不执行去重操作。
- force：强制查询分支执行去重操作。
- cost：优化器在查询分支去重和不去重这两种执行方式中，选择代价比较小的执行方式。

默认值：cost

须知

此参数配置仅在SQL语句的执行计划满足以下条件时生效：

- SQL语句中的UNION/EXCEPT/INTERSECT操作不带ALL选项。
- 执行集合操作的各个查询分支在进行集合操作前执行过数据重分布动作。

skew_option

参数说明：控制是否使用优化策略。

取值范围：字符串

- off：关闭策略。
- normal：采用激进策略。对于不确定是否出现倾斜的场景，认为存在倾斜，并进行相应优化。
- lazy：采用保守策略。对于不确定是否出现倾斜场景，认为不存在倾斜，不进行优化。

默认值：normal

enable_expr_skew_optimization

参数说明：控制是否在倾斜优化策略中使用表达式统计信息。该参数仅9.1.0.100及以上集群版本支持。

取值范围：布尔型

- on表示在倾斜优化策略中使用表达式统计信息来判断是否存在倾斜。
- off表示在倾斜优化策略中不使用表达式统计信息来判断是否存在倾斜。

默认值：on

prefer_hashjoin_path

参数说明：控制是否优先生成hashjoin路径，以便通过将代价较大的其它路径预剪枝掉的方法来提升整体计划生成时间。该参数仅8.2.1及以上集群版本支持。

取值范围：布尔型

- on表示启用hashjoin路径提前生成的优化。
- off表示关闭hashjoin路径提前生成的优化。

默认值：on

enable_hashfilter_test

参数说明：该参数用于控制是否为基表扫描增加分布列的hashfilter，以便确认结果是否符合预期。同时，在数据插入时，控制是否进行DN准确性校验（即校验当前数据是否应该插入当前DN）。

取值范围：布尔型

- on表示为基表扫描添加分布列的hashfilter，并在数据插入时进行DN准确性校验。
- off表示不为基表扫描添加分布列的hashfilter，并在数据插入时不进行DN准确性校验。

默认值：off

须知

- 此参数仅对hash分布的表有效。
- 当该参数设置为on后，因在数据插入时会进行DN准确性校验，会影响数据插入性能。

enable_topk_optimization

参数说明：控制是否开启TopK排序优化。该参数仅9.1.0.200及以上集群版本支持。

取值范围：布尔型

- on表示开启TopK排序优化。
- off表示关闭TopK排序优化。

默认值：on

rewrite_rule

参数说明：标识开启的可选查询重写规则，在特定的场景中，通过此GUC参数对查询重写规则进行设置，使得查询效率最优。

此参数可以控制查询重写规则的组合，例如有多个重写规则：rule1、rule2、rule3、rule4。可以设置：

```
set rewrite_rule='on(rule1)';          --启用查询重写规则 rule1
set rewrite_rule='off(rule1)';          --关闭查询重写规则 rule1
set rewrite_rule='rule2,rule3';    --启用查询重写规则 rule2 和 rule3
set rewrite_rule=none;           --关闭所有可选查询重写规则
```

取值范围：字符串

表 7-1 取值说明

取值	作用
none	不使用任何可选查询重写规则。
lazyagg	使用Lazy Agg查询重写规则（消除子查询中的聚集运算）。
magicset	使用Magic Set查询重写规则（从主查询中下推条件到子查询）。
uniquecheck	使用Unique Check重写规则（允许目标列不含聚集函数的表达式子链接场景提升，需在子链接按关联列聚集后目标列值唯一才能开启，建议专业调优人员使用）。
disablerep	使用禁止复制表的子链接提升规则（针对复制表禁止子链接提升）。
notinopt	使用Not In查询重写规则（将Not In转换为更高效的Hash Join）。
or_conversion	使用OR查询重写规则（将包含OR的复杂谓词拆分或重写）。
projection_pushdown	使用Projection Pushdown查询重写规则。
plain_lazyagg	使用简单Lazy Agg查询重写规则（用于单个子查询和distinct操作）。
eager_magicset	使用Eager Magic Set查询重写规则（允许将带有group by或partition by的子查询中的qual条件下推）。
outer_join_quality_imply	使用外连接查询重写规则（允许在外连接中从外部表向内部表传递属性）。
inlist_merge	使用Inlist Merge查询重写规则（将同一列多个IN合并成一个IN(List)）。
casewhen_simplification	使用CASE WHEN查询重写优化（将case when..then..else..end进行常量折叠，去除不可达分支，简化计算）。
subquery_qual_pull_up	使用子查询上升查询重写优化（将子查询中的过滤条件提升到外层查询）。
join_elimination	使用Join Elimination查询重写规则（删除对最终结果没有影响的连接）。
not_distinct_from_opt	使用Not Distinct From查询重写规则（is not distinct from转换为=形式以支持hashjoin）。
foreign_agg_opt	使用部分聚合函数查询重写优化规则（对count(1)，max，min函数，通过文件footer查结果集）。
selfjoin_elimination	使用自连接查询优化重写规则（对exists、any子链接中两表自链接消除为单表聚合）。

默认值：magicset、or_conversion、projection_pushdown、plain_lazyagg、subquery_qual_pull_up、not_distinct_from_opt、foreign_agg_opt

cte_options

参数说明：用来配置计划能够选择ctescan的前提条件。

参数选项及取值：

表 7-2 参数选项及取值说明

参数名	取值	作用
pred_threshold	整型[0-50]，默认值为2。	设置单表使用cterscan的最小谓词数量。
max_mem	整型[32-2097151]（配置参数不可加单位，单位为MB），默认值为256。	设置最大估计内存，在引用计数小于ref_count_threshold时使用ctescan。
ref_count_threshold	整型[0-50]，默认值为4。	设置最小引用计数，在估算cte内存超过max_mem时，使用共享扫描。

示例：

```
set cte_options = "(pred_threshold:2),(max_mem:256),(ref_count_threshold:4)";
```

analyze_mode

参数说明：配置analyze选项。

取值范围：字符串类型。

- none：无特殊配置。
- auto：开启auto-analyze。
- file_sample：开启文件级采样。

默认值：none

7.3.2 其他优化器选项

default_statistics_target

参数说明：为没有用ALTER TABLE SET STATISTICS设置字段目标的表设置缺省统计目标。此参数设置为正数是代表统计信息的样本数量，为负数时，代表使用百分比的形式设置统计目标，负数转换为对应的百分比，即-5代表5%。采样时，会将default_statistics_target * 300作为随机抽样的大小，例如默认值为100时，会随机读取30000个页面再从中随机取30000条数据来完成随机抽样。

取值范围：浮点型，-100～10000。

须知

- 比默认值大的正数数值增加了ANALYZE所需的时间，但是可能会改善优化器的估计质量。
- 调整此参数可能存在性能劣化的风险，如果某个查询劣化，可以考虑
  - 恢复默认的统计信息。
  - 使用plan hint来调整到之前的查询计划。
- 当此guc参数设置为负数时，如果计算的采样样本数大于等于总数据量的2%，且用户表的数据量小于1600000时，ANALYZE所需时间相比guc参数为默认值的时间会有所增加。
- autoanalyze不支持临时表采样方式设置采样大小，采样过程使用参数默认值。
- 当强制使用内存方式计算统计信息时，采样大小受maintenance_work_mem参数限制。

默认值：100

from_collapse_limit

参数说明：根据生成的FROM列表的项数来判断优化器是否将把子查询合并到上层查询，如果FROM列表项个数小于等于该参数值，优化器会将子查询合并到上层查询。

取值范围：整型，1～INT_MAX。

须知

比默认值小的数值将降低规划时间，但是可能生成差的执行计划。

默认值：8

join_collapse_limit

参数说明：根据得出的列表项数来判断优化器是否执行把除FULL JOINS之外的JOIN构造重写到FROM列表中。

取值范围：整型，1～INT_MAX。

须知

- 设置为1会避免任何JOIN重排。这样就使得查询中指定的连接顺序就是实际的连接顺序。查询优化器并不是总能选取最优的连接顺序，高级用户可以选择暂时把这个变量设置为1，然后指定它们需要的连接顺序。
- 比默认值小的数值减少规划时间但也降低了执行计划的质量。

默认值：8

enable_bloom_filter

参数说明: 标识是否允许使用BloomFilter优化。

取值范围：布尔型

- on表示允许使用BloomFilter优化。
- off表示不允许使用BloomFilter优化。

默认值：on

须知

适用场景：外表侧同线程包含有HDFS内外表或列存表的HASH JOIN会触发Bloom Filter。

使用限制：

1. JOIN类型仅限于INNER JOIN、SEMI JOIN、RIGHT JOIN、RIGHT SEMI JOIN、RIGHT ANTI JOIN、RIGHT ANTI FULL JOIN。
2. JOIN内表侧关联条件：对于HDFS内外表不能为表达式；对于列存表可以为表达式，但仅限于非JOIN层计算的表达式。
3. JOIN外表侧关联条件必须为简单列关联。
4. JOIN内表侧与外表侧关联条件均为简单列关联时，计划层估算必须可以去除1/3以上的数据（仅针对HDFS内外表）。
5. JOIN不能包含null值关联。
6. 数据类型：
    - HDFS内外表字段类型支持SMALLINT、INTEGER、BIGINT、REAL/FLOAT4、DOUBLE PRECISION/FLOAT8、CHAR(n)/CHARACTER(n)/NCHAR(n)、VARCHAR(n)/CHARACTER VARYING(n)、CLOB、TEXT。
    - 列存表字段类型支持SMALLINT、INTEGER、BIGINT、OID、"char"、CHAR(n)/CHARACTER(n)/NCHAR(n)、VARCHAR(n)/CHARACTER VARYING(n)、NVARCHAR2(n)、CLOB、TEXT、DATE、TIME、TIMESTAMP、TIMESTAMPTZ，其中字符类型其排序规则必须指定为"C"。

enable_extrapolation_stats

参数说明: 标识是否允许基于历史统计信息使用推理估算的逻辑。使用该逻辑对于未及时收集统计信息的表可以增大估算准确的可能性，但也存在错误推理导致估算过大的可能性。

取值范围：布尔型

- on表示允许基于历史统计信息使用推理估算的逻辑。
- off表示不允许基于历史统计信息使用推理估算的逻辑。

默认值：off

query_dop

参数说明: 用户自定义的查询并行度。

取值范围：整型，-64-64

- [1,64]：打开固定SMP功能，系统会使用固定并行度。

- 0：打开SMP自适应功能，系统会根据资源情况和计划特征动态为每个查询选取[1,8]之间（x86平台），[1,64]之间（鲲鹏平台）的最优的并行
- [-64,-1]：打开SMP自适应功能，并限制自适应选取的最大并行度。

默认值：1

smp_thread_cost

参数说明：在SMP自适应场景下，规划线程并行度时，考虑每个SMP线程应承载的最小估算代价。当算子估算代价较小而并行度较大，使每个SMP线程计算的估算代价小于该阈值时，优化器会自动降低算子的并行度，以避免为低代价算子开启较高并行度浪费系统资源。

取值范围：浮点型，0-10000

- 0：关闭SMP线程的估算代价阈值限制，此时并行度的设置不受算子代价的影响。
- (0, 10000]：设置SMP线程的最小估算代价阈值，设置越大，则低代价算子越趋向于降低并行度。

默认值：1000

说明

当该参数不为0时，算子的估算代价过小会导致SMP并行度较低。当发现对应情况时，可将该参数设置为0，增大计算并行度。

plan_mode_seed

参数说明：该参数为调测参数，用于控制优化器通过动态规划算法进行代价估算的最优执行计划，或生成随机的计划。

参数类型：USERSET

取值范围：整型，-1~ 2147483647

- 0：通过动态规划算法进行代价估算的最优执行计划。
- -1：由优化器随机生成[1，2147483647]范围整型值的随机数，并根据随机数生成随机的执行计划。
- [1，2147483647]：由优化器根据指定随机数生成随机的执行计划。

默认值：0

须知

- 当该参数设置为随机执行计划模式时，优化器会生成不同的随机执行计划，该执行计划可能不是最优计划。因此在随机计划模式下，会对查询性能产生影响，所以建议在正常业务操作或运维过程中将该参数保持为默认值0。
- 当该参数不为0时，查询指定的plan hint不会生效。

fq_query_vdn

参数说明：控制任务拉起的actor个数。

取值范围：整型，[0, 256]

- 0：自适应actor个数。根据扫描和写入的代价动态选取actor个数。
- 1：使用单个actor，退化为单cn模式执行。
- 2-256：使用指定个数的actor执行。

默认值：2

fq_actor_adapt_type

参数说明：控制拉起actor自适应的方式。

取值范围：枚举类型

- default：不采用自适应的方式。actor默认规格为4u/16G。
- mem：进行actor规格自适应。保持cpu/mem为1：4的规格，cpu的范围为[1, 16]。

默认值：mem

说明

该参数仅在actor个数自适应的场景下生效，即fq_query_vdn=0。

fq_dynamic_actor_cost_threshold

参数说明：fq_query_vdn=0时，单cn代价的阈值。超过该阈值时拉起actor。

取值范围：浮点型，1~1e10

默认值：200000

说明

自适应时，选取的actor个数和代价之间不是简单的线性关系，而是梯度不断下降的过程，随着代价的增大，最终会趋向于256。

7.4 客户端连接缺省设置

7.4.1 语句行为

介绍SQL语句执行过程的相关默认参数。

search_path

参数说明：当一个被引用对象没有指定模式时，此参数设置模式搜索顺序。它的值由一个或多个模式名构成，不同的模式名用逗号隔开。

- 当前会话存放临时表的模式时，可以使用别名pg_temp将它列在搜索路径中，如'pg_temp，public' 。存放临时表的模式始终会作为第一个被搜索的对象，排在pg_catalog和search_path中所有模式的前面，即具有第一搜索优先级。建议用户不要在search_path中显式设置pg_temp。如果在search_path中指定了pg_temp，但不是在最前面，系统会提示设置无效，pg_temp仍被优先搜索。通

过使用别名pg_temp，系统只会在存放临时表的模式中搜索表、视图和数据类型这样的数据库对象，不会在里面搜索函数或运算符这样的数据库对象。

- 系统表所在的模式pg_catalog，总是排在search_path中指定的所有模式前面被搜索，即具有第二搜索优先级（pg_temp具有第一搜索优先级）。建议用户不要在search_path中显式设置pg_catalog。如果在search_path中指定了pg_catalog，但不是在最前面，系统会提示设置无效，pg_catalog仍被第二优先搜索。
- 当没有指定一个特定模式而创建一个对象时，它们被放置到以search_path为命名的第一个模式中。当搜索路径为空时，会报错误。
- 通过SQL函数current_schema可以检测当前搜索路径的有效值。这和检测search_path的值不尽相同，因为current_schema显示search_path中首位有效的模式名称。

取值范围：字符串

说明

- 设置为空串（''）的时候，系统会自动转换成一对双引号。
- 设置的内容中包含双引号，系统会认为是不安全字符，会将每个双引号转换成一对双引号。

默认值：default_db

current_schema

参数说明：设置当前的模式。

取值范围：字符串

默认值：default_db

statement_timeout

参数说明：当语句执行时间超过该参数设置的时间（从服务器收到命令时开始计时）时，该语句将会报错并退出执行。

取值范围：整型，0～2147483647，单位为毫秒（ms）。

默认值：0

bytea_output

参数说明：设置bytea类型值的输出格式。

取值范围：枚举型

- hex：将二进制数据编码为每字节2位十六进制数字。
- escape：传统化的PostgreSQL格式。采用以ASCII字符序列表示二进制串的方法，同时将那些无法表示成ASCII字符的二进制串转换成特殊的转义序列。

默认值：hex

xmlbinary

参数说明：设置二进制值是如何在XML中进行编码的。

取值范围：枚举型

- base64
- hex

默认值：base64

xmloption

参数说明：当XML和字符串值之间进行转换时，设置document或content是否是隐含的。

取值范围：枚举型

- document：表示HTML格式的文档。
- content：普通的字符串。

默认值：content

enable_disk_cache

参数说明：设置是否打开磁盘缓存和数据预读，当前数据预取仅对PARQUET/ORC格式文件有效，磁盘缓存对所有格式文件均有效。

取值范围：布尔型

- on：打开磁盘缓存和数据预取。
- off：关闭磁盘缓存和数据预取。

默认值：on

disk_cache_max_size

参数说明：用于设置磁盘缓存的总大小。

取值范围：整型，512MB~1PB

默认值：5GB

enable_aio_scheduler

参数说明：控制是否开启异步IO调度，该IO调度是异步读写的基础。

取值范围：布尔型

- on/true表示开启此IO调度开关。
- off/false表示关闭此IO调度开关。

默认值：on

runtime_filter_type

参数说明：标识使用的runtime filter类型。

取值范围：字符串

- all，表示应用所有的runtime filter。

- min_max，表示仅应用join场景下的runtime filter，且join场景仅会生成min_max过滤器。
- bloom_filter，表示仅应用join场景下的runtime filter，且满足条件后join会生成bloom filter进行过滤。
- global_filter：表示应用join场景下，跨dn节点的runtime filter，开启后支持runtime filter对不同dn上的数据生成min_max/bloom_filter过滤器。
- none，表示不使用runtime filter，此时仅对原版bloom filter生效场景具有过滤效果。

默认值：none

enable_meta_scan

参数说明：Iceberg查询时是否打开metaScan。

取值范围：bool

- on/true，打开metaScan，即在查询时由dn分布式获取待扫描文件列表。
- off/false，关闭metaScan，即在查询时在cn获取待扫描文件列表。

默认值：true

enable_spill_to_remote_storage

参数说明：控制是否打开spill-to-obs特性开关，为true时表示开启特性，下盘数据会被disk cache管控，并在空间不足时使用OBS作为逃生通道；为false时表示使用旧的下盘方式，直接写到本地EVS中。该参数存在参数依赖，当打开spill-to-obs特性开关enable_spill_to_remote_storage时，use_yr_as_block_cache_backend开关必须为false。

取值范围： bool

- on/true表示开启。
- off/false表示关闭。

默认值：true

注意

- 由于spill-to-obs特性依赖的元戎相关能力（如append buf）在25.3.0版本暂时无法提供，当前存在特性使用限制，当打开enable_spill_to_remote_storage时，diskcache的存储后端当前无法使用元戎数据系统，请设置use_yr_as_block_cache_backend开关为false后再启用spill-to-obs特性。
- use_yr_as_block_cache_backend为false时，意味着近计算缓存会直接使用DN实例下的目录来缓存数据，此时注意系统盘的空间问题， 此时建议将function-agent中存放DN实例的路径映射在大容量的物理EVS盘中，避免系统盘空间不足导致的pod被驱逐的问题。

staging_folder_expire_time

参数说明：控制ORC、PARQUET表临时文件目录残留后的自动清理周期，默认七天后自动清理。

取值范围： int64，取值范围12h~1year

默认值：604800

obs_result_format

参数说明：控制向OBS写入结果集文件时的格式，以及是否启动结果集文件压缩功能。

参数类型：USERSET

取值范围：int，取值范围0~3。

- 0：结果集文件格式为JSON，并且不对结果集文件进行压缩。
- 1：结果集文件格式为JSON，并且使用zstd算法对结果集文件进行压缩。
- 2：结果集文件格式为ARROW，并且不对结果集文件进行压缩。
- 3：结果集文件格式为ARROW，并且使用zstd算法对结果集文件进行压缩。

默认值：0

resource_track_level

参数说明：控制SQL监控数据中query_plan字段上报的信息类型，当前只对select、insert、delete、update、create table as语句生效。

参数类型：USERSET

取值范围：枚举型

- query：SQL监控数据的query_plan字段上报explain信息。
- perf：SQL监控数据的query_plan字段上报explain performance信息。

默认值：perf

filter_options

参数说明：标识使用的查询过滤增强选项。

取值范围：字符串，可以同时设置多个。

- parquet_pushdown：支持parquet表的or条件下推到存储层进行过滤。
- iceberg_pushdown：支持iceberg表的or条件下推到存储层进行过滤。
- dictionary：支持parquet表根据字典信息过滤rowgroup。
- partition_dynamic_pruning：支持动态分区剪枝。

默认值：空

log_min_messages

参数说明：控制写到服务器日志文件中的消息级别。

参数类型：USERSET

取值范围：枚举型，有效值有debug5、debug4、debug3、debug2、debug1、info、notice、warning、error、log、fatal、panic。每个级别包含其后所有的级别，级别越

靠后，发送到日志的消息越少。例如log在log_min_messages中的级别在error之后，意味这log级别时只能打印log、fatal、panic三种日志到服务器日志文件中。

默认值：warning

scan_max_batch_rows

参数说明：控制从数据表扫描时单个batch的最大行数。该参数当前只支持iceberg表、parquet表和orc表。

参数类型：USERSET

取值范围：int，取值范围为1~1000。

默认值：1000

7.4.2 区域和格式化

介绍时间格式设置的相关参数。

DateStyle

参数说明：设置日期和时间值的显示格式，以及有歧义的输入值的解析规则。

这个变量包含两个独立的部分：输出格式声明（ISO、Postgres、SQL、German）和输入输出的年/月/日顺序（DMY、MDY、YMD）。这两个可以独立设置或者一起设置。关键字Euro和European等价于DMY；关键字US、NonEuro、NonEuropean等价于MDY 。

取值范围：字符串

默认值：ISO, MDY

说明

gs_initdb会将这个参数初始化成与**lc_time**一致的值。

设置建议：优先推荐使用ISO格式。Postgres、SQL和German均采用字母缩写的形式来表示时区，例如“EST、WST、CST”等。

IntervalStyle

参数说明：设置区间值的显示格式。

取值范围：枚举型

- sql_standard表示产生与SQL标准规定匹配的输出。
- postgres表示产生与PostgreSQL 8.4版本相匹配的输出，当**DateStyle**参数被设为ISO时。
- postgres_verbose表示产生与PostgreSQL 8.4版本相匹配的输出，当**DateStyle**参数被设为non_ISO时。
- iso_8601表示产生与在ISO 8601中定义的“格式与代号”相匹配的输出。
- oracle表示产生于Oracle中与numtodsinterval函数相匹配的输出结果，详细请参考numtodsinterval。

须知

IntervalStyle参数也会影响不明确的间隔输入的说明。

默认值： postgres

TimeZone

参数说明： 设置显示和解释时间类型数值时使用的时区。

取值范围： 字符串

默认值： GMT

timezone_abbreviations

参数说明： 设置服务器接受的时区缩写值。

取值范围： 字符串

默认值： Default

说明

Default表示通用时区的缩写。但也有其他诸如 'Australia' 和 'India' 等用来定义特定的安装。

extra_float_digits

参数说明： 调整浮点值显示的数据位数，浮点类型包括float4、float8 以及几何数据类型。参数值加在标准的数据位数上（FLT_DIG或DBL_DIG中合适的）。

取值范围： 整型，-15～3

说明

- 设置为3，表示包括部分有效的数据位。对转储需要精确恢复的浮点数据尤其有用。
- 设置为负数，表示摒弃不需要的数据位。

默认值： 0

client_encoding

参数说明： 设置客户端的字符编码类型。

请根据前端业务的情况确定。尽量客户端编码和服务器端编码一致，提高效率。

取值范围： 兼容PostgreSQL所有的字符编码类型。其中UTF8表示使用数据库的字符编码类型。

说明

- 使用命令locale -a查看当前系统支持的区域和相应的编码格式，并可以选择进行设置。
- 默认情况下，gs_initdb会根据当前的系统环境初始化此参数，通过locale命令可以查看当前的配置环境。
- 参数建议保持默认值，不建议通过gs_guc工具或其他方式直接在postgresql.conf文件中设置client_encoding参数，即使设置也不会生效，以保证集群内部通信编码格式一致。

默认值： UTF8

推荐值： SQL_ASCII/UTF8

lc_monetary

参数说明： 设置货币值的显示格式，影响to_char之类的函数的输出。可接受的值是系统相关的。

取值范围： 字符串

说明

- 使用命令locale -a查看当前系统支持的区域和相应的编码格式，并可以选择进行设置。
- 默认情况下，gs_initdb会根据当前的系统环境初始化此参数，通过locale命令可以查看当前的配置环境。

默认值： en_US.UTF-8

lc_numeric

参数说明： 设置数值的显示格式，影响to_char之类的函数的输出。可接受的值是系统相关的。

取值范围： 字符串

说明

- 使用命令locale -a查看当前系统支持的区域和相应的编码格式，并可以选择进行设置。
- 默认情况下，gs_initdb会根据当前的系统环境初始化此参数，通过locale命令可以查看当前的配置环境。

默认值： en_US.UTF-8

lc_time

参数说明： 设置时间和区域的显示格式，影响to_char之类的函数的输出。可接受的值是系统相关的。

取值范围： 字符串

说明

- 使用命令locale -a查看当前系统支持的区域和相应的编码格式，并可以选择进行设置。
- 默认情况下，gs_initdb会根据当前的系统环境初始化此参数，通过locale命令可以查看当前的配置环境。

默认值： en_US.UTF-8

7.5 锁管理

在DataArtsFabric SQL中，使用LakeFormation中心锁来保证多计算节点并发写入同一张表时的正确性。本节介绍的参数主要管理LakeFormation中心锁的相关行为。

fq_lflock_wait_timeout

参数说明：控制获取锁的最长等待时间。当申请锁的等待时间超过设定值时，系统会报错。

取值范围：整型，0 ~ INT_MAX，单位为毫秒（ms）。

- 如果该参数的值等于0，表示至多允许申请一次，如果申请失败则直接报错，不会重试。

默认值：30s

fq_lflock_heartbeat_timeout

参数说明：设置锁心跳失败的最大超时时间。

- 为了维护LakeFormation中心锁的存活，系统会在持锁期间持续向LakeFormation发送锁心跳。当锁心跳发送失败时，系统不会立即报错而会不断重试，到最大超时时间时如果仍未成功则会报错。
- LakeFormation未收到锁心跳的自动放锁时间为60s，系统在正常情况下以30s的时间间隔向LakeFormation发送心跳。因此当此参数小于30s时，系统在第一次发送心跳失败时即会报错，当此参数大于60s时，系统在与LakeFormation长时间断连的情况下有可能丢锁。如果用户在系统与LakeFormation长时间断连的情况下仍不希望发生丢锁，请将此参数设置为小于60s，或自行保证不会在此参数设置的时间间隔内并发写入同一张表。

取值范围：整型，0 ~ INT_MAX，单位为毫秒（ms）。

默认值：5min

8 SQL 语法参考

8.1 关键字

SQL里有保留字和非保留字之分。根据标准，保留字绝不能用做其他标识符。非保留字只是在特定的环境里有特殊的含义，而在其他环境里是可以用做标识符的。

表 **8-1** SQL 关键字

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
ABORT	非保留	-	-
ABS	-	非保留	-
ABSOLUTE	非保留	保留	保留
ACCESS	非保留	-	-
ACCOUNT	非保留	-	-
ACTION	非保留	保留	保留
ADA	-	非保留	非保留
ADD	非保留	保留	保留
ADMIN	非保留	保留	-
AFTER	非保留	保留	-
AGGREGATE	非保留	保留	-
ALIAS	-	保留	-
ALL	保留	保留	保留
ALLOCATE	-	保留	保留
ALSO	非保留	-	-
ALTER	非保留	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
ALWAYS	非保留	-	-
ANALYSE	保留	-	-
ANALYZE	保留	-	-
AND	保留	保留	保留
ANY	保留	保留	保留
APP	非保留	-	-
ARE	-	保留	保留
ARRAY	保留	保留	-
AS	保留	保留	保留
ASC	保留	保留	保留
ASENSITIVE	-	非保留	-
ASSERTION	非保留	保留	保留
ASSIGNMENT	非保留	非保留	-
ASYMMETRIC	保留	非保留	-
AT	非保留	保留	保留
ATOMIC	-	非保留	-
ATTRIBUTE	非保留	-	-
AUTHID	保留	-	-
AUTHINFO	非保留	-	-
AUTHORIZATION	保留（可以是函数或类型）	保留	保留
AUTOEXTEND	非保留	-	-
AUTOMAPPED	非保留	-	-
AVG	-	非保留	保留
BACKWARD	非保留	-	-
BARRIER	非保留	-	-
BEFORE	非保留	保留	-
BEGIN	非保留	保留	保留
BETWEEN	非保留（不能是函数或类型）	非保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
BIGINT	非保留（不能是函数或类型）	-	-
BINARY	保留（可以是函数或类型）	保留	-
BINARY_DOUBLE	非保留（不能是函数或类型）	-	-
BINARY_INTEGER	非保留（不能是函数或类型）	-	-
BIT	非保留（不能是函数或类型）	保留	保留
BITVAR	-	非保留	-
BIT_LENGTH	-	非保留	保留
BLOB	非保留	保留	-
BOOLEAN	非保留（不能是函数或类型）	保留	-
BOTH	保留	保留	保留
BUCKETS	保留	-	-
BREADTH	-	保留	-
BY	非保留	保留	保留
C	-	非保留	非保留
CACHE	非保留	-	-
CALL	非保留	保留	-
CALLED	非保留	非保留	-
CARDINALITY	-	非保留	-
CASCADE	非保留	保留	保留
CASCADED	非保留	保留	保留
CASE	保留	保留	保留
CAST	保留	保留	保留
CATALOG	非保留	保留	保留
CATALOG_NAME	-	非保留	非保留
CHAIN	非保留	非保留	-
CHAR	非保留（不能是函数或类型）	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
CHARACTER	非保留（不能是函数或类型）	保留	保留
CHARACTERISTICS	非保留	-	-
CHARACTER_LENGTH	-	非保留	保留
CHARACTER_SET_CATALOG	-	非保留	非保留
CHARACTER_SET_NAME	-	非保留	非保留
CHARACTER_SET_SCHEMA	-	非保留	非保留
CHAR_LENGTH	-	非保留	保留
CHECK	保留	保留	保留
CHECKED	-	非保留	-
CHECKPOINT	非保留	-	-
CLASS	非保留	保留	-
CLEAN	非保留	-	-
CLASS_ORIGIN	-	非保留	非保留
CLOB	非保留	保留	-
CLOSE	非保留	保留	保留
CLUSTER	非保留	-	-
COALESCE	非保留（不能是函数或类型）	非保留	保留
COBOL	-	非保留	非保留
COLLATE	保留	保留	保留
COLLATION	保留（可以是函数或类型）	保留	保留
COLLATION_CATALOG	-	非保留	非保留
COLLATION_NAME	-	非保留	非保留
COLLATION_SCHEMA	-	非保留	非保留
COLUMN	保留	保留	保留
COLUMNS	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
COLUMN_NAME	-	非保留	非保留
COMMAND_FUNCTION	-	非保留	非保留
COMMAND_FUNCTION_CODE	-	非保留	-
COMMENT	非保留	-	-
COMMENTS	非保留	-	-
COMMIT	非保留	保留	保留
COMMITTED	非保留	非保留	非保留
COMPATIBLE_ILLEGAL_CHARS	非保留	-	-
COMPLETE	非保留	-	-
COMPRESS	非保留	-	-
COMPLETION	-	保留	-
CONCURRENTLY	保留（可以是函数或类型）	-	-
CONDITION	-	-	-
CONDITION_NUMBER	-	非保留	非保留
CONFIGURATION	非保留	-	-
CONNECT	-	保留	保留
CONNECTION	非保留	保留	保留
CONNECTION_NAME	-	非保留	非保留
CONSTRAINT	保留	保留	保留
CONSTRAINTS	非保留	保留	保留
CONSTRAINT_CATALOG	-	非保留	非保留
CONSTRAINT_NAME	-	非保留	非保留
CONSTRAINT_SCHEMA	-	非保留	非保留
CONSTRUCTOR	-	保留	-
CONTAINS	-	非保留	-
CONTENT	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
CONTINUE	非保留	保留	保留
CONVERSION	非保留	-	-
CONVERT	-	非保留	保留
COORDINATOR	非保留	-	-
COPY	非保留	-	-
CORRESPONDING	-	保留	保留
COST	非保留	-	-
COUNT	-	非保留	保留
CREATE	保留	保留	保留
CROSS	保留（可以是函数或类型）	保留	保留
CSV	非保留	-	-
CUBE	-	保留	-
CURRENT	非保留	保留	保留
CURRENT_CATALOG	保留	-	-
CURRENT_DATE	保留	保留	保留
CURRENT_PATH	-	保留	-
CURRENT_ROLE	保留	保留	-
CURRENT_SCHEMA	保留（可以是函数或类型）	-	-
CURRENT_TIME	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留
CURRENT_USER	保留	保留	保留
CURSOR	非保留	保留	保留
CURSOR_NAME	-	非保留	非保留
CYCLE	非保留	保留	-
DATA	非保留	保留	非保留
DATE_FORMAT	非保留	-	-
DATABASE	非保留	-	-
DATAFILE	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
DATE	非保留（不能是函数或类型）	保留	保留
DATETIME_INTERVAL_CODE	-	非保留	非保留
DATETIME_INTERVAL_PRECISION	-	非保留	非保留
DAY	非保留	保留	保留
DBCOMPATIBILITY	非保留	-	-
DEALLOCATE	非保留	保留	保留
DEC	非保留（不能是函数或类型）	保留	保留
DECIMAL	非保留（不能是函数或类型）	保留	保留
DECLARE	非保留	保留	保留
DECODE	非保留（不能是函数或类型）	-	-
DEFAULT	保留	保留	保留
DEFAULTS	非保留	-	-
DEFERRABLE	保留	保留	保留
DEFERRED	非保留	保留	保留
DEFINED	-	非保留	-
DEFINER	非保留	非保留	-
DELETE	非保留	保留	保留
DELIMITER	非保留	-	-
DELIMITERS	非保留	-	-
DELTA	非保留	-	-
DEPTH	-	保留	-
DEREF	-	保留	-
DESC	保留	保留	保留
DESCRIBE	-	保留	保留
DESCRIPTOR	-	保留	保留
DESTROY	-	保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
DESTRUCTOR	-	保留	-
DETERMINISTIC	非保留	保留	-
DIAGNOSTICS	-	保留	保留
DICTIONARY	非保留	保留	-
DIRECT	非保留	-	-
DIRECTORY	非保留	-	-
DISABLE	非保留	-	-
DISCARD	非保留	-	-
DISCONNECT	-	保留	保留
DISPATCH	-	非保留	-
DISTINCT	保留	保留	保留
DISTRIBUTE	非保留	-	-
DISTRIBUTION	非保留	-	-
DO	保留	-	-
DOCUMENT	非保留	-	-
DOMAIN	非保留	保留	保留
DOUBLE	非保留	保留	保留
DROP	非保留	保留	保留
DYNAMIC	-	保留	-
DYNAMIC_FUNCTION	-	非保留	非保留
DYNAMIC_FUNCTION_CODE	-	非保留	-
EACH	非保留	保留	-
ELASTIC	非保留	-	-
ELSE	保留	保留	保留
ENABLE	非保留	-	-
ENCODING	非保留	-	-
ENCRYPTED	非保留	-	-
END	保留	保留	保留
END-EXEC	-	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
ENFORCED	非保留	-	-
ENUM	非保留	-	-
EOL	非保留	-	-
EQUALS	-	保留	-
ERRORS	非保留	-	-
ESCAPE	非保留	保留	保留
ESCAPING	非保留	-	-
EVERY	非保留	保留	-
EXCEPT	保留	保留	保留
EXCEPTION	-	保留	保留
EXCHANGE	非保留	-	-
EXCLUDE	非保留	-	-
EXCLUDING	非保留	-	-
EXCLUSIVE	非保留	-	-
EXEC	-	保留	保留
EXECUTE	非保留	保留	保留
EXISTING	-	非保留	-
EXISTS	非保留（不能是函数或类型）	非保留	保留
EXPIRATION	非保留	-	-
EXPLAIN	非保留	-	-
EXTENSION	非保留	-	-
EXTERNAL	非保留	保留	保留
EXTRACT	非保留（不能是函数或类型）	非保留	保留
FALSE	保留	保留	保留
FAMILY	非保留	-	-
FAST	非保留	-	-
FENCED	非保留	-	-
FETCH	保留	保留	保留
FILEHEADER	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
FILL_MISSING_FIELDS	非保留	-	-
FINAL	-	非保留	-
FIRST	非保留	保留	保留
FIXED	非保留	保留	保留
FLOAT	非保留（不能是函数或类型）	保留	保留
FOLLOWING	非保留	-	-
FOR	保留	保留	保留
FORCE	非保留	-	-
FOREIGN	保留	保留	保留
FORMATTER	非保留	-	-
FORTRAN	-	非保留	非保留
FORWARD	非保留	-	-
FOUND	-	保留	保留
FREE	-	保留	-
FREEZE	保留（可以是函数或类型）	-	-
FROM	保留	保留	保留
FULL	保留（可以是函数或类型）	保留	保留
FUNCTION	非保留	保留	-
FUNCTIONS	非保留	-	-
G	-	非保留	-
GENERAL	-	保留	-
GENERATED	-	非保留	-
GET	-	保留	保留
GLOBAL	非保留	保留	保留
GO	-	保留	保留
GOTO	-	保留	保留
GRANT	保留	保留	保留
GRANTED	非保留	非保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
GREATEST	非保留（不能是函数或类型）	-	-
GROUP	保留	保留	保留
GROUPING	-	保留	-
HANDLER	非保留	-	-
HAVING	保留	保留	保留
HEADER	非保留	-	-
HIERARCHY	-	非保留	-
HOLD	非保留	非保留	-
HOST	-	保留	-
HOUR	非保留	保留	保留
IDENTIFIED	非保留	-	-
IDENTITY	非保留	保留	保留
IF	非保留（不能是函数或类型）	-	-
IFNULL	非保留（不能是函数或类型）	-	-
IGNORE	-	保留	-
IGNORE_EXTRA_DATA	非保留	-	-
ILIKE	保留（可以是函数或类型）	-	-
IMMEDIATE	非保留	保留	保留
IMMUTABLE	非保留	-	-
IMPLEMENTATION	-	非保留	-
IMPLICIT	非保留	-	-
IN	保留	保留	保留
INCLUDING	非保留	-	-
INCREMENT	非保留	-	-
INDEX	非保留	-	-
INDEXES	非保留	-	-
INDICATOR	-	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
INFIX	-	非保留	-
INHERIT	非保留	-	-
INHERITS	非保留	-	-
INITIAL	非保留	-	-
INITIALIZE	-	保留	-
INITIALLY	保留	保留	保留
INITRANS	非保留	-	-
INLINE	非保留	-	-
INNER	保留（可以是函数或类型）	保留	保留
INOUT	非保留（不能是函数或类型）	保留	-
INPUT	非保留	保留	保留
INSENSITIVE	非保留	非保留	保留
INSERT	非保留	保留	保留
INSTANCE	-	非保留	-
INSTANTIABLE	-	非保留	-
INSTEAD	非保留	-	-
INT	非保留（不能是函数或类型）	保留	保留
INTEGER	非保留（不能是函数或类型）	保留	保留
INTERNAL	保留	-	-
INTERSECT	保留	保留	保留
INTERVAL	非保留（不能是函数或类型）	保留	保留
INTO	保留	保留	保留
INVOKER	非保留	非保留	-
IS	保留	保留	保留
ISNULL	非保留（不能是函数或类型）	-	-
ISOLATION	非保留	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
ITERATE	-	保留	-
JOIN	保留（可以是函数或类型）	保留	保留
K	-	非保留	-
KEY	非保留	保留	保留
KEY_MEMBER	-	非保留	-
KEY_TYPE	-	非保留	-
LABEL	非保留	-	-
LANGUAGE	非保留	保留	保留
LARGE	非保留	保留	-
LAST	非保留	保留	保留
LATERAL	-	保留	-
LC_COLLATE	非保留	-	-
LC_CTYPE	非保留	-	-
LEADING	保留	保留	保留
LEAKPROOF	非保留	-	-
LEAST	非保留（不能是函数或类型）	-	-
LEFT	保留（可以是函数或类型）	保留	保留
LENGTH	-	非保留	非保留
LESS	保留	保留	-
LEVEL	非保留	保留	保留
LIKE	保留（可以是函数或类型）	保留	保留
LIMIT	保留	保留	-
LISTEN	非保留	-	-
LOAD	非保留	-	-
LOCAL	非保留	保留	保留
LOCALTIME	保留	保留	-
LOCALTIMESTAMP	保留	保留	-
LOCATION	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
LOCATOR	-	保留	-
LOCK	非保留	-	-
LOG	非保留	-	-
LOGGING	非保留	-	-
LOGIN	非保留	-	-
LOOP	非保留	-	-
LOWER	-	非保留	保留
M	-	非保留	-
MAP	-	保留	-
MAPPING	非保留	-	-
MATCH	非保留	保留	保留
MATCHED	非保留	-	-
MATERIALIZED	非保留	-	-
MAX	-	非保留	保留
MAXEXTENTS	非保留	-	-
MAXSIZE	非保留	-	-
MAXTRANS	非保留	-	-
MAXVALUE	保留	-	-
MERGE	非保留	-	-
MESSAGE_LENGTH	-	非保留	非保留
MESSAGE_OCTET_LENGTH	-	非保留	非保留
MESSAGE_TEXT	-	非保留	非保留
METHOD	-	非保留	-
MIN	-	非保留	保留
MINEXTENTS	非保留	-	-
MINUS	保留	-	-
MINUTE	非保留	保留	保留
MINVALUE	非保留	-	-
MOD	-	非保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
MODE	非保留	-	-
MODIFIES	-	保留	-
MODIFY	保留	保留	-
MODULE	-	保留	保留
MONTH	非保留	保留	保留
MORE	-	非保留	非保留
MOVE	非保留	-	-
MOVEMENT	非保留	-	-
MUMPS	-	非保留	非保留
NAME	非保留	非保留	非保留
NAMES	非保留	保留	保留
NATIONAL	非保留（不能是函数或类型）	保留	保留
NATURAL	保留（可以是函数或类型）	保留	保留
NCHAR	非保留（不能是函数或类型）	保留	保留
NCLOB	-	保留	-
NEW	-	保留	-
NEXT	非保留	保留	保留
NLSSORT	保留	-	-
NO	非保留	保留	保留
NOCOMPRESS	非保留	-	-
NOCYCLE	非保留	-	-
NODE	非保留	-	-
NOLOGGING	非保留	-	-
NOLOGIN	非保留	-	-
NOMAXVALUE	非保留	-	-
NOMINVALUE	非保留	-	-
NONE	非保留（不能是函数或类型）	保留	-
NOT	保留	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
NOTHING	非保留	-	-
NOTIFY	非保留	-	-
NOTNULL	保留（可以是函数或类型）	-	-
NOWAIT	非保留	-	-
NULL	保留	保留	保留
NULLABLE	-	非保留	非保留
NULLIF	非保留（不能是函数或类型）	非保留	保留
NULLS	非保留	-	-
NUMBER	非保留（不能是函数或类型）	非保留	非保留
NUMERIC	非保留（不能是函数或类型）	保留	保留
NUMSTR	非保留	-	-
NVARCHAR2	非保留（不能是函数或类型）	-	-
NVL	非保留（不能是函数或类型）	-	-
OBJECT	非保留	保留	-
OCTET_LENGTH	-	非保留	保留
OF	非保留	保留	保留
OFF	非保留	保留	-
OFFSET	保留	-	-
OIDS	非保留	-	-
OLD	-	保留	-
ON	保留	保留	保留
ONLY	保留	保留	保留
OPEN	-	保留	保留
OPERATION	-	保留	-
OPERATOR	非保留	-	-
OPTIMIZATION	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
OPTIMIZE	非保留	-	-
OPTION	非保留	保留	保留
OPTIONS	非保留	非保留	-
OR	保留	保留	保留
ORDER	保留	保留	保留
ORDINALITY	-	保留	-
OUT	非保留（不能是函数或类型）	保留	-
OUTER	保留（可以是函数或类型）	保留	保留
OUTPUT	-	保留	保留
OVER	非保留	-	-
OVERLAPS	保留（可以是函数或类型）	非保留	保留
OVERLAY	非保留（不能是函数或类型）	非保留	-
OVERRIDING	-	非保留	-
OWNED	非保留	-	-
OWNER	非保留	-	-
PACKAGE	非保留	-	-
PAD	-	保留	保留
PARAMETER	-	保留	-
PARAMETERS	-	保留	-
PARAMETER_MODE	-	非保留	-
PARAMETER_NAME	-	非保留	-
PARAMETER_ORDINAL_POSITION	-	非保留	-
PARAMETER_SPECIFIC_CATALOG	-	非保留	-
PARAMETER_SPECIFIC_NAME	-	非保留	-
PARAMETER_SPECIFIC_SCHEMA	-	非保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
PARSER	非保留	-	-
PARTIAL	非保留	保留	保留
PARTITION	非保留	-	-
PARTITIONS	非保留	-	-
PASCAL	-	非保留	非保留
PASSING	非保留	-	-
PASSWORD	非保留	-	-
PATH	-	保留	-
PCTFREE	非保留	-	-
PER	非保留	-	-
PERM	非保留	-	-
PERCENT	非保留	-	-
PERFORMANCE	保留	-	-
PLACING	保留	-	-
PLAN	保留	-	-
PLANS	非保留	-	-
PLI	-	非保留	非保留
POLICY	非保留	-	-
POOL	非保留	-	-
POSITION	非保留（不能是函数或类型）	非保留	保留
POSTFIX	-	保留	-
PRECEDING	非保留	-	-
PRECISION	非保留（不能是函数或类型）	保留	保留
PREFERRED	非保留	-	-
PREFIX	非保留	保留	-
PREORDER	-	保留	-
PREPARE	非保留	保留	保留
PREPARED	非保留	-	-
PRESERVE	非保留	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
PRIMARY	保留	保留	保留
PRIOR	非保留	保留	保留
PRIVATE	非保留	-	-
PRIVILEGE	非保留	-	-
PRIVILEGES	非保留	保留	保留
PROCEDURAL	非保留	-	-
PROCEDURE	保留	保留	保留
PROFILE	非保留	-	-
PUBLIC	-	保留	保留
QUERY	非保留	-	-
QUOTE	非保留	-	-
RANGE	非保留	-	-
RAW	非保留	-	-
READ	非保留	保留	保留
READS	-	保留	-
REAL	非保留（不能是函数或类型）	保留	保留
REASSIGN	非保留	-	-
REBUILD	非保留	-	-
RECHECK	非保留	-	-
RECURSIVE	非保留	保留	-
REF	非保留	保留	-
REFRESH	非保留	-	-
REFERENCES	保留	保留	保留
REFERENCING	-	保留	-
REINDEX	非保留	-	-
REJECT	保留	-	-
RELATIVE	非保留	保留	保留
RELEASE	非保留	-	-
RELOPTIONS	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
REMOTE	非保留	-	-
RENAME	非保留	-	-
REPEATABLE	非保留	非保留	非保留
REPLACE	非保留	-	-
REPLICA	非保留	-	-
RESET	非保留	-	-
RESIZE	非保留	-	-
RESOURCE	非保留	-	-
RESTART	非保留	-	-
RESTRICT	非保留	保留	保留
RESULT	-	保留	-
RETURN	非保留	保留	-
RETURNED_LENGTH	-	非保留	非保留
RETURNED_OCTET_LENGTH	-	非保留	非保留
RETURNED_SQLSTATE	-	非保留	非保留
RETURNING	保留	-	-
RETURNS	非保留	保留	-
REUSE	非保留	-	-
REVOKE	非保留	保留	保留
RIGHT	保留（可以是函数或类型）	保留	保留
ROLE	非保留	保留	-
ROLLBACK	非保留	保留	保留
ROLLUP	-	保留	-
ROUTINE	-	保留	-
ROUTINE_CATALOG	-	非保留	-
ROUTINE_NAME	-	非保留	-
ROUTINE_SCHEMA	-	非保留	-
ROW	非保留（不能是函数或类型）	保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
ROWS	非保留	保留	保留
ROW_COUNT	-	非保留	非保留
RULE	非保留	-	-
SAVEPOINT	非保留	保留	-
SCALE	-	非保留	非保留
SCHEMA	非保留	保留	保留
SCHEMA_NAME	-	非保留	非保留
SCOPE	-	保留	-
SCROLL	非保留	保留	保留
SEARCH	非保留	保留	-
SECOND	非保留	保留	保留
SECTION	-	保留	保留
SECURITY	非保留	非保留	-
SELECT	保留	保留	保留
SELF	-	非保留	-
SENSITIVE	-	非保留	-
SEPARATOR	非保留	-	-
SEQUENCE	非保留	保留	-
SEQUENCES	非保留	-	-
SERIALIZABLE	非保留	非保留	非保留
SERVER	非保留	-	-
SERVER_NAME	-	非保留	非保留
SESSION	非保留	保留	保留
SESSION_USER	保留	保留	保留
SET	非保留	保留	保留
SETOF	非保留（不能是函数或类型）	-	-
SETS	-	保留	-
SHARE	非保留	-	-
SHIPPABLE	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
SHOW	非保留	-	-
SIMILAR	保留（可以是函数或类型）	非保留	-
SIMPLE	非保留	非保留	-
SIZE	非保留	保留	保留
SMALLDATETIME	非保留（不能是函数或类型）	-	-
SMALLDATETIME_FORMAT	非保留	-	-
SMALLINT	非保留（不能是函数或类型）	保留	保留
SNAPSHOT	非保留	-	-
SOME	保留	保留	保留
SOURCE	非保留	非保留	-
SPACE	-	保留	保留
SPECIFIC	-	保留	-
SPECIFICTYPE	-	保留	-
SPECIFIC_NAME	-	非保留	-
SPILL	非保留	-	-
SPLIT	非保留	-	-
SQL	-	保留	保留
SQLCODE	-	-	保留
SQLERROR	-	-	保留
SQLEXCEPTION	-	保留	-
SQLSTATE	-	保留	保留
SQLWARNING	-	保留	-
STABLE	非保留	-	-
STANDALONE	非保留	-	-
START	非保留	保留	-
STATE	-	保留	-
STATEMENT	非保留	保留	-
STATEMENT_ID	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
STATIC	-	保留	-
STATISTICS	非保留	-	-
STDIN	非保留	-	-
STDOUT	非保留	-	-
STORAGE	非保留	-	-
STORE	非保留	-	-
STRICT	非保留	-	-
STRIP	非保留	-	-
STRUCTURE	-	保留	-
STYLE	-	非保留	-
SUBCLASS_ORIGIN	-	非保留	非保留
SUBLIST	-	非保留	-
SUBSTRING	非保留（不能是函数或类型）	非保留	保留
SUM	-	非保留	保留
SUPERUSER	非保留	-	-
SYMMETRIC	保留	非保留	-
SYNONYM	非保留	-	-
SYS_REFCURSOR	非保留	-	-
SYSDATE	保留	-	-
SYSID	非保留	-	-
SYSTEM	非保留	非保留	-
SYSTEM_USER	-	保留	保留
TABLE	保留	保留	保留
TABLES	非保留	-	-
TABLE_NAME	-	非保留	非保留
TEMP	非保留	-	-
TEMPLATE	非保留	-	-
TEMPORARY	非保留	保留	保留
TERMINATE	-	保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
TEXT	非保留	-	-
THAN	非保留	保留	-
THEN	保留	保留	保留
TIME	非保留（不能是函数或类型）	保留	保留
TIME_FORMAT	非保留	-	-
TIMESTAMP	非保留（不能是函数或类型）	保留	保留
TIMESTAMPADD	非保留（不能是函数或类型）	-	-
TIMESTAMPDIFF	非保留（不能是函数或类型）	-	-
TIMESTAMP_FORMAT	非保留	-	-
TIMEZONE_HOUR	-	保留	保留
TIMEZONE_MINUTE	-	保留	保留
TINYINT	非保留（不能是函数或类型）	-	-
TO	保留	保留	保留
TRAILING	保留	保留	保留
TRANSACTION	非保留	保留	保留
TRANSACTIONS_COMMITTED	-	非保留	-
TRANSACTIONS_ROLLED_BACK	-	非保留	-
TRANSACTION_ACTIVE	-	非保留	-
TRANSFORM	-	非保留	-
TRANSFORMS	-	非保留	-
TRANSLATE	-	非保留	保留
TRANSLATION	-	保留	保留
TREAT	非保留（不能是函数或类型）	保留	-
TRIGGER	非保留	保留	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
TRIGGER_CATALOG	-	非保留	-
TRIGGER_NAME	-	非保留	-
TRIGGER_SCHEMA	-	非保留	-
TRIM	非保留（不能是函数或类型）	非保留	保留
TRUE	保留	保留	保留
TRUNCATE	非保留	-	-
TRUSTED	非保留	-	-
TRY_CAST	非保留	-	-
TSTAG	保留，该字段仅在IoT数仓中使用	-	-
TSTIME	保留，该字段仅在IoT数仓中使用	-	-
TSFIELD	保留，该字段仅在IoT数仓中使用	-	-
TYPE	非保留	非保留	非保留
TYPES	非保留	-	-
UESCAPE	-	-	-
UNBOUNDED	非保留	-	-
UNCOMMITTED	非保留	非保留	非保留
UNDER	-	保留	-
UNENCRYPTED	非保留	-	-
UNION	保留	保留	保留
UNIQUE	保留	保留	保留
UNKNOWN	非保留	保留	保留
UNLIMITED	非保留	-	-
UNLISTEN	非保留	-	-
UNLOCK	非保留	-	-
UNLOGGED	非保留	-	-
UNNAMED	-	非保留	非保留
UNNEST	-	保留	-
UNTIL	非保留	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
UNUSABLE	非保留	-	-
UPDATE	非保留	保留	保留
UPPER	-	非保留	保留
USAGE	-	保留	保留
USER	保留	保留	保留
USER_DEFINED_TYPE_CATALOG	-	非保留	-
USER_DEFINED_TYPE_NAME	-	非保留	-
USER_DEFINED_TYPE_SCHEMA	-	非保留	-
USING	保留	保留	保留
VACUUM	非保留	-	-
VALID	非保留	-	-
VALIDATE	非保留	-	-
VALIDATION	非保留	-	-
VALIDATOR	非保留	-	-
VALUE	非保留	保留	保留
VALUES	非保留（不能是函数或类型）	保留	保留
VARCHAR	非保留（不能是函数或类型）	保留	保留
VARCHAR2	非保留（不能是函数或类型）	-	-
VARIABLE	-	保留	-
VARIADIC	保留	-	-
VARYING	非保留	保留	保留
VCGROUP	非保留	-	-
VERBOSE	保留（可以是函数或类型）	-	-
VERIFY	非保留	-	-
VERSION	非保留	-	-
VIEW	非保留	保留	保留

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
VOLATILE	非保留	-	-
WHEN	保留	保留	保留
WHENEVER	-	保留	保留
WHERE	保留	保留	保留
WHITESPACE	非保留	-	-
WINDOW	保留	-	-
WITH	保留	保留	保留
WITHIN	非保留	-	-
WITHOUT	非保留	保留	-
WORK	非保留	保留	保留
WORKLOAD	非保留	-	-
WRAPPER	非保留	-	-
WRITE	非保留	保留	保留
XML	非保留	-	-
XMLATTRIBUTES	非保留（不能是函数或类型）	-	-
XMLCONCAT	非保留（不能是函数或类型）	-	-
XMLELEMENT	非保留（不能是函数或类型）	-	-
XMLEXISTS	非保留（不能是函数或类型）	-	-
XMLFOREST	非保留（不能是函数或类型）	-	-
XMLNAMESPACES	非保留（不能是函数或类型）	-	-
XMLPARSE	非保留（不能是函数或类型）	-	-
XMLPI	非保留（不能是函数或类型）	-	-
XMLROOT	非保留（不能是函数或类型）	-	-
XMLSERIALIZE	非保留（不能是函数或类型）	-	-

关键字	DataArtsFabric SQL	SQL:1999	SQL:1992
XMLTABLE	非保留（不能是函数或类型）	-	-
YEAR	非保留	保留	保留
YES	非保留	-	-
ZONE	非保留	保留	保留

8.2 数据类型

8.2.1 与 LakeFormation 数据类型映射关系

DataArtsFabric SQL中定义的数据类型与LakeFormation中的类型存在明确的映射关系，如下表所示：

表 8-2 DataArtsFabric SQL 与 LakeFormation 数据类型映射关系

数据类型	DataArtsFabric SQL数据类型名称	描述	LakeFormation数据类型名称	备注
数值类型	SMALLINT	小范围整数，别名为INT2。	smallint	-32,768 ~ +32,767
	INTEGER	常用的整数，别名为INT4。	int	-2,147,483,648 ~ +2,147,483,647
	BIGINT	大范围的整数，别名为INT8。	bigint	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
	NUMERIC[(p[,s])]， DECIMAL[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	decimal	1≤p≤38, 0≤s≤p; p未指定时，报错提示超过精度范围。
	REAL,FLOAT4	单精度浮点数，不精准。	float	6位十进制数字精度。
	DOUBLE PRECISION, FLOAT8	双精度浮点数，不精准。	double	15位十进制数字精度。

数据类型	DataArtsFabric SQL数据类型名称	描述	LakeFormation数据类型名称	备注
	FLOAT[(p)]	浮点数，不精准。精度p取值范围为[1,53]。 如不指定精度，则默认为DOUBLE PRECISION表示。	float/ double	1≤p<25时，存储为float类型； 25≤p≤53时，存储为double类型。
	DEC[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	decimal	1≤p≤38, 0≤s≤p; p未指定时，报错提示超过精度范围。
	INTEGER[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	decimal	1≤p≤38, 0≤s≤p; p未指定时，报错提示超过精度范围。
布尔类型	BOOLEAN	布尔类型：true、false、null	boolean	-
字符类型	CHAR(n),CHARACTER(n),NCHAR(n)	定长字符串，n是指字符长度不足填充空格，n小于等于255。	char	1≤n≤255；n未指定时，默认为1。
	VARCHAR(n),CHARACTER VARYING(n)	变长字符串，n是指字符长度，n小于等于65535。	varchar	1≤n≤65535；n未指定时，默认为65535。
	TEXT	变长字符串。	string	-
日期/时间类型	DATE	记录日期。	date	如果数据包含时间信息，仅截断存储日期数据 。
	TIMESTAMP[(p)] [WITHOUT TIME ZONE]	日期和时间，不带时区信息。 p表示小数点后的精度，取值范围为0~6。	timestamp	实际存储精度固定为6，不受p大小影响; p＞6时，提示精度降为6。
二进制类型	BYTEA	变长的二进制字符串。	binary	-
复杂类型	ARRAY<...>	数组类型，下标从1开始，支持多维和嵌套。	array	-
	STRUCT<...>	STRUCT类型，支持嵌套。	struct	-

8.2.2 数值类型

数值类型也叫数字类型。由1、2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。

对应的数字操作符和相关函数，请参见数字操作函数和操作符。

DataArtsFabric SQL支持的数值类型按精度可以分为：整数类型，任意精度型，浮点类型和序列整型。

整数类型

SMALLINT、INTEGER和BIGINT类型存储整个数值（不带有小数部分），也就是整数。如果尝试存储超出范围以外的数值将会导致错误。

常用的类型是INTEGER，一般只有取值范围确定不超过SMALLINT的情况下，才会使用SMALLINT类型。而只有在INTEGER的范围不够的时候才使用BIGINT，因为前者相对快得多。

表 8-3 整数类型

名称	描述	存储空间	范围
SMALLINT	小范围整数，别名为INT2。	2字节	-32,768 ~ +32,767
INTEGER	常用的整数，别名为INT4。	4字节	-2,147,483,648 ~ +2,147,483,647
BIGINT	大范围的整数，别名为INT8。	8字节	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

示例：

创建带有SMALLINT、INTEGER、BIGINT类型数据的表。

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

插入数据。

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

查看数据。

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

任意精度型

使用Numeric/Decimal进行列定义时，建议指定该列的精度p（总位数）以及范围s（小数位数）。

如果数值的精度或者范围大于列的数据类型所声明的精度和范围，那么系统将会试图对这个值进行四舍五入。如果不能对数值进行四舍五入的处理来满足数据类型的限制，则会报错。

表 8-4 任意精度型

名称	描述	存储空间	范围
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。

示例：

创建带有DECIMAL数值类型的表。

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

插入数据。

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

查看数据。

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

浮点类型

浮点类型属于非精确，可变精度的数值类型。实际上，这些类型通常是对于二进制浮点算术（分别是单精度和双精度）的IEEE标准754的具体实现，在一定范围内由特定的处理器，操作系统和编译器所支持。

表 8-5 浮点类型

名称	描述	存储空间	范围
REAL, FLOAT4	单精度浮点数，不精准。	4字节	6位十进制数字精度。
DOUBLE PRECISION, FLOAT8	双精度浮点数，不精准。	8字节	1E-307~1E+308, 15位十进制数字精度。

名称	描述	存储空间	范围
FLOAT[(p)]	浮点数，不精准。精度p取值范围为[1,53]。 **说明** p为精度，表示总位数。	4字节或8字节	根据精度p不同选择REAL或DOUBLE PRECISION作为内部表示。如不指定精度，内部用DOUBLE PRECISION表示。
DEC[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。 **说明** p为总位数，s为小数位位数。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。
INTEGER[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。

示例：

创建带有浮点类型的表。

```
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;
```

插入数据。

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

查看数据。

```
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)
```

8.2.3 布尔类型

表 8-6 布尔类型

名称	描述	存储空间	取值
BOOLEAN	布尔类型	1字节。	• true：真 • false：假 • null：未知（unknown）

“真”值的有效文本值是：

TRUE、't'、'true'、'y'、'yes'、'1'。

“假”值的有效文本值是：

FALSE、'f'、'false'、'n'、'no'、'0'。

使用TRUE和FALSE是比较规范的用法（也是SQL兼容的用法）。

示例

显示用字母t和f输出boolean值。

```
--创建表。
CREATE TABLE bool_type_t1
(
BT_COL1 BOOLEAN,
BT_COL2 TEXT
) STORE AS orc;

--插入数据。
INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');
INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');

--查看数据。
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';
 bl_col1 | bt_col2
---------+---------
 t       | sic est
(1 row)

--删除表。
DROP TABLE bool_type_t1;
```

8.2.4 字符类型

DataArtsFabric SQL支持的字符类型请参见表8-7。字符串操作符和相关的内置函数请参见字符处理函数和操作符。

表 **8-7** 字符类型

名称	描述	长度	存储空间
CHAR(n) CHARACTER(n) NCHAR(n)	定长字符串，不足填充空格。	n是指字节长度，如不带精度n，默认精度为1。n小于10485761。	最大为10MB。
VARCHAR(n) CHARACTER VARYING(n)	变长字符串。	n是指字节长度，n小于10485761。	最大为10MB。
TEXT	变长字符串。	-	最大为1GB-8203B（即1073733621B）。

说明

- 除了每列的大小限制以外，每个元组的总大小也不可超过1GB-8023B（即1073733621B）。
- 对于字符串数据，建议使用变长字符串数据类型，并指定最大长度。请务必确保指定的最大长度大于需要存储的最大字符数，避免超出最大长度时出现字符截断现象。除非明确知道数据类型为固定长度字符串，否则，不建议使用CHAR(n)、NCHAR(n)、CHARACTER(n)等定长字符类型。在DataArtsFabric SQL中，定长字符类型的运算会存在额外的存储和内存开销。

在DataArtsFabric SQL里另外还有两种定长字符类型。在**表8-8**里显示。

其中，name类型只用在内部系统表中，作为存储标识符，该类型长度当前定为64字节（63可用字符加结束符）。不建议普通用户使用这种数据类型。name类型与其他数据类型进行对齐时（例如case when的多个分支中，其中一个分支返回name类型，其他类型返回text类型），可能会出现向name类型对齐，字符截断。如果不希望出现字符按照64位截断的情况，则需要将name类型强制类型转化为text类型。

类型"char"只用了一个字节的存储空间。它在系统内部主要用于系统表，主要作为简单化的枚举类型使用。

表 **8-8** 特殊字符类型

名称	描述	存储空间
name	用于对象名的内部类型。	64字节。
"char"	单字节内部类型。	1字节。

长度

如果把一个字段定义为char(n)或者varchar(n)， 代表该字段最大可容纳n个长度的数据。无论哪种类型，可设置的最大长度都不得超过10485760（即10MB）。

当数据长度超过指定的长度n时，会抛出错误"value too long"。也可通过指定数据类型，使超过长度的数据自动截断。

示例：

1. 创建表t1，指定其字段的字符类型。
```
CREATE TABLE t1(a char(5), b varchar(5)) STORE AS orc;
```
2. 向表t1插入数据时超过指定的字节长度报错。
```
INSERT INTO t1 VALUES('bookstore','123');
ERROR:  value too long for type character(5)
```
3. 向表t1插入数据并明确超过指定字节长度后自动截断。
```
INSERT INTO t1 VALUES('bookstore'::char(5),'12345678'::varchar(5));
INSERT 0 1

SELECT a,b FROM t1;
   a   |   b
-------+-------
 books | 12345
(1 row)
```

定长与变长

所有字符类型根据长度是否固定可以分为定长字符串与变长字符串两大类。

- 对于定长字符串，长度必须确定，如果不指定长度，则默认长度1；如果数据长度不足，会在尾部自动填充空格，用以存储和显示；但这部分填充的数据是无意义的，实际使用中会被忽略，如比较、排序或类型转换。
- 对于变长字符串，如果指定长度，则为最大可存储数据长度；如果不指定长度，则认为该字段支持任意长度。

示例：

1. 创建表t2，指定其字段的字符类型。
```
CREATE TABLE t2 (a char(5),b varchar(5)) STORE AS orc;
```
2. 向表t2插入数据并查询字段a的字节长度。因建表时指定a的字符类型为char(5)且是定长字符串，长度不足，填充空格，所以查询的字节长度为5。
```
INSERT INTO t2 VALUES('abc','abc');
INSERT 0 1

SELECT a,lengthb(a),b FROM t2;
   a   | lengthb |  b
-------+---------+-----
 abc   |       5 | abc
(1 row)
```
3. 用函数转换后查询字段a的实际字节长度为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)
```

空串与 NULL

TD与MySQL兼容模式下，区分空串与null。

- TD兼容模式下示例：
```
SELECT  '' is null , null is null;
 isnull |  isnull
```

```
--------+----------
 f      | t
(1 rows)
```

- MySQL兼容模式下示例：

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

8.2.5 二进制类型

DataArtsFabric SQL支持的二进制类型请参见**表8-9**。

表 8-9 二进制类型

名称	描述	存储空间
BYTEA	变长的二进制字符串	4字节加上实际的二进制字符串。最大为1G-8203B（即1073733621B）。

说明

除了每列的大小限制以外，每个元组的总大小也不可超过1G-8203字节。

示例

```
--创建表。
CREATE TABLE binary_type_t1
(
BT_COL1 INTEGER,
BT_COL2 BYTEA
) STORE AS orc;

--插入数据。
INSERT INTO binary_type_t1 VALUES(10, E'\\xDEADBEEF');

--查询表中的数据。
SELECT * FROM binary_type_t1;
 bt_col1 |  bt_col2
---------+------------
      10 | \xdeadbeef
(1 row)

--删除表。
DROP TABLE binary_type_t1;
```

8.2.6 日期/时间类型

DataArtsFabric SQL支持的日期/时间类型请参见**表8-10**。该类型的操作符和内置函数请参见**时间、日期处理函数和操作符**。

说明

如果其他的数据库时间格式和DataArtsFabric SQL的时间格式不一致，可通过修改配置参数**DateStyle**的值来保持一致。

表 8-10 日期/时间类型

名称	描述	存储空间
DATE	Oracle兼容模式下等价于timestamp(0)，记录日期和时间。 其他模式下，记录日期。	Oracle兼容模式下，占存储空间8字节 其他模式下，占存储空间4字节
TIMESTAMP[(p)] [WITHOUT TIME ZONE]	日期和时间。 p表示小数点后的精度，取值范围为0~6。	8字节

示例：

```
--创建表。
CREATE TABLE date_type_t1(
DA_COL1 DATE
) STORE AS orc;

--插入数据。
INSERT INTO date_type_t1 VALUES (date '12-10-2010');

--查看数据。
SELECT * FROM date_type_t1;
  da_col1
------------
 2010-12-10
(1 row)

--删除表。
DROP TABLE date_type_t1;

--创建表。
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;
--插入数据。
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');
--查看数据。
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)

--删除表。
DROP TABLE timestamp_type_t1;
```

日期输入

日期和时间的输入几乎可以是任何合理的格式，包括ISO-8601格式、SQL-兼容格式、传统POSTGRES格式或者其它的形式。系统支持按照日、月、年的顺序自定义日期输入。如果把DateStyle参数设置为MDY就按照“月-日-年”解析，设置为DMY就按照“日-月-年”解析，设置为YMD就按照“年-月-日”解析。

日期的文本输入需要加单引号包围，语法如下：

type [(p)] 'value'

可选的精度声明中的p是一个整数，表示在秒域中小数部分的位数。**表8-11**显示了date类型的输入方式。

表 8-11 日期输入方式

例子	描述
1999-01-08	ISO 8601格式（建议格式），任何方式下都是1999年1月8号。
January 8, 1999	在任何datestyle输入模式下都无歧义。
1/8/1999	有歧义，在MDY模式下是一月八号，在DMY模式下是八月一号。
1/18/1999	MDY模式下是一月十八日，其它模式下被拒绝。
01/02/03	● MDY模式下的2003年1月2日。 ● DMY模式下的2003年2月1日。 ● YMD模式下的2001年2月3日。
1999-Jan-08	任何模式下都是1月8日。
Jan-08-1999	任何模式下都是1月8日。
08-Jan-1999	任何模式下都是1月8日。
99-Jan-08	YMD模式下是1月8日，否则错误。
08-Jan-99	一月八日，除了在YMD模式下是错误的之外。
Jan-08-99	一月八日，除了在YMD模式下是错误的之外。
19990108	ISO 8601；任何模式下都是1999年1月8日。
990108	ISO 8601；任何模式下都是1999年1月8日。
1999.008	年和年里的第几天。
J2451187	儒略日。
January 8, 99 BC	公元前99年。

示例：

```
--创建表。
CREATE TABLE date_style_t1(DAT_COL1 DATE) STORE AS orc;

--插入数据。
INSERT INTO date_style_t1 VALUES(date '12-10-2010');

--查看数据。
SELECT  * FROM date_style_t1;
     dat_col1
---------------------
```

```
 2010-12-10 00:00:00
(1 row)

--查看日期格式。
SHOW datestyle;
 DateStyle
-----------
 ISO, MDY
(1 row)

--设置日期格式。
SET datestyle='YMD';
SET

--插入数据。
INSERT INTO date_style_t1 VALUES(date '2010-12-11');

--查看数据。
SELECT  * FROM date_style_t1;
      dat_col1
---------------------
 2010-12-10
 2010-12-11
(2 rows)

--删除表。
DROP TABLE date_style_t1;
```

时间

时间类型支持timestamp [(p)] without time zone。如果只写timestamp等效于timestamp without time zone。

如果在time without time zone类型的输入中声明了时区，则会忽略这个时区。

时间输入类型的详细信息请参见**表8-12**，时区输入类型的详细信息请参见**表8-13**。

表 8-12 时间输入

例子	描述
05:06.8	ISO 8601
4:05:06	ISO 8601
4:05	ISO 8601
40506	ISO 8601
4:05 AM	与04:05一样，AM不影响数值
4:05 PM	与16:05一样，输入小时数必须<= 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	缩写的时区

表 8-13 时区输入

例子	描述
PST	太平洋标准时间（Pacific Standard Time）
-8:00	ISO 8601与PST的偏移
-800	ISO 8601与PST的偏移
-8	ISO 8601与PST的偏移

特殊值

DataArtsFabric SQL支持几个特殊值，在读取的时候将被转换成普通的日期/时间值，请参考表8-14。

表 8-14 特殊值

输入字符串	适用类型	描述
epoch	date，timestamp	1970-01-01 00:00:00+00 （Unix系统零时）
infinity	timestamp	比任何其他时间戳都晚
-infinity	timestamp	比任何其他时间戳都早
now	date，time，timestamp	当前事务的开始时间
today	date，timestamp	今日午夜
tomorrow	date，timestamp	明日午夜
yesterday	date，timestamp	昨日午夜
allballs	time	00:00:00.00 UTC

8.2.7 对象标识符类型

DataArtsFabric SQL在内部使用对象标识符（OID）作为各种系统表的主键。系统不会给用户创建的表增加一个OID系统字段，OID类型代表一个对象标识符。

目前OID类型用一个四字节的无符号整数实现。因此不建议在创建的表中使用OID字段做主键。

表 8-15 对象标识符类型

名称	引用	描述	示例
OID	-	数字化的对象标识符。	564182

名称	引用	描述	示例
CID	-	命令标识符。它是系统字段cmin和cmax的数据类型。命令标识符是32位的量。	-
XID	-	事务标识符。它是系统字段xmin和xmax的数据类型。事务标识符也是32位的量。	-
TID	-	行标识符。它是系统表字段ctid的数据类型。行ID是一对数值（块号，块内的行索引），它标识该行在其所在表内的物理位置。	-
REGCONFIG	pg_ts_config	文本搜索配置。	english
REGDICTIONARY	pg_ts_dict	文本搜索字典。	simple
REGOPER	pg_operator	操作符名。	+
REGOPERATOR	pg_operator	带参数类型的操作符。	*(integer,integer)或-(NONE,integer)
REGPROC	pg_proc	函数名字。	sum
REGPROCEDURE	pg_proc	带参数类型的函数。	sum(int4)
REGCLASS	pg_class	关系名。	pg_type
REGTYPE	pg_type	数据类型名。	integer

OID类型：主要作为数据库系统表中字段使用。

示例：

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

OID别名类型REGCLASS：主要用于对象OID值的简化查找。

示例：

```
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)
```

8.2.8 隐式转换支持范围

DataArtsFabric SQL当前对于存储格式为orc或parquet的表类型，支持数值类型的隐式转换，即存储空间字节数多的类型向下兼容存储空间字节数少的类型。

例如实际存储为orc::SHORT(2字节)，建表类型为SMALLINT、INT、BIGINT时都可以正常查询，DataArtsFabric SQL当前支持的隐式转换范围如下表所示。

表 8-16 DataArtsFabric SQL 数值类型隐式转换支持范围

DataArtsFabric数值类型隐式转换	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	×	×	×	√	×	×

DataArtsFabric数值类型隐式转换	SMALLINT	INT	BIGINT	FLOAT4	FLOAT8	NUMERIC
orc::double/ parquet::64 to	×	×	×	×	√	×
orc::decimal/ parquet::decimal to	×	×	×	×	×	√

8.2.9 复杂类型

DataArtsFabric SQL支持从iceberg表和parquet文件中读写复杂类型列，orc文件仅支持读，仅支持struct和array类型及其嵌套类型。

复杂类型 DDL

当前DDL仅支持STRUCT和ARRAY类型。

定义语法如下：

- ARRAY：定义语法为ARRAY<data_type>。
- STRUCT：定义语法为STRUCT<col_name : data_type, ...>。

其中data_type表示数据类型，可以是基础类型和复杂类型，基础类型请参见**与LakeFormation数据类型映射关系**。

DDL示例：

```
-- 创建包含复杂类型的managed表
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;

-- 创建包含复杂类型的external表
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';

-- 建表后，可查看LakeFormation完整定义
DESCRIBE test_managed_table ;
DESCRIBE test_external_table ;
```

复杂类型 DML

- Struct类型数据构造方式通过ROW(x,x,x,...)，例如：ROW(1,2,3,..)；struct内列数超过1列时，row可省略，如：(1,2,3,...)。
- Array类型数据构造方式通过ARRAY[x,x,x,..]，例如：ARRAY[1,2,3,...]。
- 嵌套类型按照上述规则依次构造即可，如struct<a:array<int>, b:array<int>>类型对应的数据结构为：ROW(ARRAY[1,2], ARRAY[3,4])。

DML示例：

```
-- 插入数据
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>>;

-- 查看数据
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)
```

复杂类型 DQL

- Struct类型支持点查的方式直接获取内部元素，点查时表列名需用括号”()”包裹。
- Array的使用与一般数组类似，通过下标指定返回某个元素，需要注意的是下标起始位置索引为1。

DQL示例：

```
-- 查询复杂类型数据
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)

-- 提取struct元素
SELECT (address).city, (address).zip FROM test_managed_table;

    city     |  zip
-------------+-------
 Los Angeles | 90001
 New York    | 10001
 Chicago     | 60601
(3 rows)

-- 提取array元素
SELECT tags[1], tags[2] FROM test_managed_table;
  tags   |   tags
---------+----------
  friend | colleague
  family |
  friend | hobby
(3 rows)
```

复杂类型转 jsonb

当前复杂类型包括STRUCT和ARRAY支持通过JSON的方式读取，需设置GUC参数：

```
SET ENABLE_COMPLEX_TYPE=OFF;
```

用户在读取数据以后需要显式转换为对应的数据类型。

DQL示例（jsonb完整操作符和函数说明请参见**JSON/JSONB函数和操作符**）：

```
--  -> 通过键或索引提取 JSON 对象或数组的元素
SELECT name, address, tags, logs->0->'term' FROM test_managed_table;
  name   |                address                 |          tags          | ?column?
---------+--------------------------------------+------------------------+------------
 Alice   | {"zip": 10001, "city": "New York"}    | ["friend", "colleague"] | "login"
 Bob     | {"zip": 90001, "city": "Los Angeles"} | ["family"]             | "purchase"
 Charlie | {"zip": 60601, "city": "Chicago"}     | ["friend", "hobby"]     |
(3 rows)

--  ->> 提取元素并转为 text
SELECT name, address FROM test_managed_table WHERE address->>'zip' = '10001';
 name  |              address
-------+-----------------------------------
 Alice | {"zip": 10001, "city": "New York"}
(1 row)

--  @> 数组包含元素，左侧数组包含右侧数组内全部元素
SELECT name, tags FROM test_managed_table WHERE tags @> '["friend"]'::jsonb;
  name   |          tags
---------+------------------------
 Alice   | ["friend", "colleague"]
 Charlie | ["friend", "hobby"]
(2 rows)

--  类型显示转换，先使用 ->> 获取为字符，再进行类型转换
SELECT (logs->0->>'time')::TIMESTAMP FROM test_managed_table;
      timestamp
---------------------
 2024-01-01 00:00:00
 2024-02-15 00:00:00

(3 rows)
```

8.3 函数和操作符

8.3.1 字符处理函数和操作符

DataArtsFabric SQL提供的字符处理函数和操作符主要用于字符串与字符串、字符串与非字符串之间的连接，以及字符串的模式匹配操作。

bit_length(string)

描述：字符串的位数。

返回值类型：integer

示例：

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

btrim(string text [, characters text])

描述：从string开头和结尾删除只包含characters中字符（缺省是空白）的最长字符串。

返回值类型：text

示例：

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

char_length(string)或 character_length(string)

描述：字符串中的字符个数。

返回值类型：integer

示例：

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

fq_iceberg_bucket(int,text)

描述：计算指定字符串的bucket数值。其中，第一个int参数为bucket数量。

返回值类型：int

示例：

```
fabricsql => select fq_iceberg_bucket(13, '你好中国!');
 fq_iceberg_bucket
-------------------
                 2
(1 row)
```

fq_iceberg_truncate(int,text)

描述：计算指定长度的前缀子字符串。其中，第一个int参数为前缀长度，以字符为单位；第二个text参数为原字符串。

返回值类型：text

示例：

```
fabricsql => select *, length(t)  from fq_iceberg_truncate (2, '你好中国!') as t;
 iceberg_truncate | length
------------------+--------
 你好             |      2
(1 row)
```

更多fq_iceberg_truncate示例，请参考**分区表达式：truncate**。

instr(text,text,int,int)

描述：字符串匹配函数的位置，第一个int表示开始匹配起始位置，第二个int表示匹配的次数。

返回值类型：integer

示例：

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

lengthb(text/bpchar)

描述：获取指定字符串的字节数。

返回值类型：integer

示例：

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

说明

- 如果字符串中存在换行符，如字符串由一个换行符和一个空格组成，在DataArtsFabric SQL中LENGTH和LENGTHB的值为2。
- 对于CHAR(n) 类型，DataArtsFabric SQL中n是指字符个数。因此，对于多字节编码的字符集， LENGTHB函数返回的长度可能大于n。

left(str text, n int)

描述：返回字符串的前n个字符。

- ORA和TD兼容模式下，当n是负数时，返回除最后|n|个字符以外的所有字符。
- MySQL兼容模式下，当n是负数时，返回空串。

返回值类型：text

示例：

```
SELECT left('abcde', 2);
 left
------
 ab
(1 row)
```

length(string bytea, encoding name)

描述：指定encoding编码格式的string的字符数。在这个编码格式中，string必须是有效的。

返回值类型：integer

示例：

```
SELECT length('jose', 'UTF8');
 length
--------
      4
(1 row)
```

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

描述：通过填充字符fill（缺省时为空白），把string填充为length长度。如果string已经比length长则将其尾部截断。

返回值类型：text

示例：

```
SELECT lpad('hi', 5, 'xyza');
 lpad
-------
 xyzhi
(1 row)
```

octet_length(string)

描述：字符串中的字节数。

返回值类型：integer

示例：

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

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

描述：替换子字符串。FROM int表示从第一个string的第几个字符开始替换，for int表示替换第一个string的字符数目。

返回值类型：text

示例：

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

position(substring in string)

描述：指定子字符串在字符串中的位置。如果string中没有substring，则返回0。

返回值类型：integer

示例：

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

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

pg_client_encoding()

描述：当前客户端编码名称。

返回值类型：name

示例：

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

quote_ident(string text)

描述：返回适用于SQL语句的标识符形式（使用适当的引号进行界定）。只有在必要的时候才会添加引号（字符串包含非标识符字符或者会转换大小写的字符）。返回值中嵌入的引号都写了两次。

返回值类型：text

示例：

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

quote_literal(string text)

描述：返回适用于在SQL语句里当作文本使用的形式（使用适当的引号进行界定）。

返回值类型：text

示例：

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

如果出现如下写法，text文本将进行转义。

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

如果出现如下写法，反斜杠会写入两次。

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

如果参数为NULL，返回空。如果参数为NULL，通常使用函数quote_nullable更适用。

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

```
(1 row)
```

quote_literal(value anyelement)

描述：将给定的值强制转换为text，加上引号作为文本。

返回值类型：text

示例：

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

如果出现如下写法，定值将进行转义。

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

如果出现如下写法，反斜杠会写入两次。

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

quote_nullable(string text)

描述：返回适用于在SQL语句里当作字符串使用的形式（使用适当的引号进行界定）。

返回值类型：text

示例：

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

如果出现如下写法，text文本将进行转义。

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

如果出现如下写法，反斜杠会写入两次。

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

如果参数为NULL，返回NULL。

```
SELECT quote_nullable(NULL);
 quote_nullable
```

```
----------------
 NULL
(1 row)
```

quote_nullable(value anyelement)

描述：将给定的参数值转化为text，加上引号作为文本。

返回值类型：text

示例：

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

如果出现如下写法，定值将进行转义。

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

如果出现如下写法，反斜杠会写入两次。

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

如果参数为NULL，返回NULL。

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

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

描述：截取子字符串，from int表示从第几个字符开始截取，for int表示截取几个字节。

返回值类型：text

示例：

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

substring(string from *pattern*)

描述：截取匹配POSIX正则表达式的子字符串。如果没有匹配它返回空值，否则返回文本中匹配模式的那部分。

返回值类型：text

示例：

```
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)
```

说明

如果POSIX正则表达式模式包含任何圆括号，那么将返回匹配第一对子表达式（对应第一个左圆括号的）的文本。如果想在表达式里使用圆括号而又不想导致这个例外，那么可以在整个表达式外边加上一对圆括号。

substring(string from pattern for escape)

描述：截取匹配SQL正则表达式的子字符串。声明的模式必须匹配整个数据串，否则函数失败并返回空值。为了标识在成功的时候应该返回的模式部分，模式必须包含逃逸字符的两次出现，并且后面要跟上双引号（"）。匹配这两个标记之间的模式的文本将被返回。

返回值类型：text

示例：

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

rawcat(raw,raw)

描述：字符串拼接函数。

返回值类型：raw

示例：

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

regexp_like(text,text,text)

描述：正则表达式的模式匹配函数。

返回值类型：bool

示例：

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

```
 f
(1 row)
```

regexp_substr(text,text)

描述：正则表达式的抽取子串函数。与substr功能相似，正则表达式出现多个并列的括号时，也全部处理。

返回值类型：text

示例：

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

(1 row)
```

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

描述：返回string中所有匹配POSIX正则表达式的子字符串。如果pattern不匹配，该函数不返回行。如果模式不包含圆括号子表达式，则每一个被返回的行都是一个单一元素的文本数组，其中包括匹配整个模式的子串。如果模式包含圆括号子表达式，该函数返回一个文本数组，它的第n个元素是匹配模式的第n个圆括号子表达式的子串。

flags参数为可选参数，包含零个或多个改变函数行为的单字母标记。i表示进行大小写无关的匹配，g表示替换每一个匹配的子字符串而不仅仅是第一个。

须知

如果提供了最后一个参数，但参数值是空字符串（''），且数据库SQL兼容模式设置为ORA的情况下，会导致返回结果为空集。这是因为ORA兼容模式将''作为NULL处理，避免此类行为的方式有如下几种：

- 将数据库SQL兼容模式改为TD。
- 不提供最后一个参数，或最后一个参数不为空字符串。

返回值类型：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)
```

⚠ 注意

如果没有子查询，当regexp_matches函数没有匹配上时，不会输出表中的数据。这通常不是所需的返回结果，应避免这种写法，建议可使用regexp_substr函数来实现相同的功能。

```
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])

描述：用POSIX正则表达式作为分隔符，分隔string。和regexp_split_to_table相同，不过regexp_split_to_array会把它的结果以一个text数组的形式返回。

返回值类型：text[]

示例：

```
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])

描述：用POSIX正则表达式作为分隔符，分隔string。如果没有与pattern的匹配，该函数返回string。如果有至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。

flags参数包含零个或多个改变函数行为的单字母标记。i表示进行大小写无关的匹配，g表示替换每一个匹配的子字符串而不仅仅是第一个。

返回值类型：setof text

示例：

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


注意

如果没有子查询，当regexp_split_to_table函数没有匹配上时，不会输出表中的数据。这通常不是所需的返回结果，应避免这种写法。

```
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)

描述：将string重复number次。

返回值类型：text

示例：

```
SELECT repeat('Pg', 4);
  repeat
----------
 PgPgPgPg
(1 row)
```

replace(string text, from text, to text)

描述：把字符串string里出现的所有子字符串from的内容替换成子字符串to的内容。

返回值类型：text

示例：

```
SELECT replace('abcdefabcdef', 'cd', 'XXX');
    replace
----------------
 abXXXefabXXXef
(1 row)
```

reverse(str)

描述：返回颠倒的字符串。

返回值类型：text

示例：

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

right(str text, n int)

描述：返回字符串中的后n个字符。

- ORA和TD兼容模式下，当n是负数时，返回除前|n|个字符以外的所有字符。
- MySQL兼容模式下，当n是负数时，返回空串。

返回值类型：text

示例：

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

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

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

描述：使用填充字符fill（缺省时为空白），把string填充到length长度。如果string已经比length长则将其从尾部截断。

返回值类型：text

示例：

```
SELECT rpad('hi', 5, 'xy');
 rpad
-------
 hixyx
(1 row)
```

rtrim(string text [, characters text])

描述：从字符串string的结尾删除只包含characters中字符（缺省是个空白）的最长的字符串。

返回值类型：text

示例：

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

sys_context ('namespace' , 'parameter')

描述：获取并返回指定namespace下参数parameter的值。

返回值类型：text

示例：

```
SELECT SYS_CONTEXT ( 'postgres' , 'archive_mode');
 sys_context
```

```
-------------

(1 row)
```

substrb(text,int,int)

描述：提取子字符串，第一个int表示提取的起始位置，第二个表示提取几个字节。

返回值类型：text

示例：

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

substrb(text,int)

描述：提取子字符串，int表示提取的起始位置。

返回值类型：text

示例：

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

string || string

描述：连接字符串。

返回值类型：text

示例：

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

string || non-string 或 non-string || string

描述：连接字符串和非字符串。

返回值类型：text

示例：

```
SELECT 'Value: '||42 AS RESULT;
  result
-----------
 Value: 42
(1 row)
```

split_part(string text, delimiter text, field int)

描述：根据delimiter分隔string返回生成的第field个子字符串（从出现第一个delimiter的text为基础）。

返回值类型：text

示例：

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

strpos(string, substring)

描述：指定的子字符串的位置。和position(substring in string)一样，不过参数顺序相反。

返回值类型：integer

示例：

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

to_hex(number int or bigint)

描述：把number转换成十六进制表现形式。

返回值类型：text

示例：

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

translate(string text, from text, to text)

描述：把在string中包含的任何匹配from中字符的字符转化为对应的在to中的字符。如果from比to长，删掉在from中出现的额外的字符。

返回值类型：text

示例：

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

length(string)

描述：获取参数string中字符的数目。

返回值类型：integer

示例：

```
SELECT length('abcd');
 length
--------
      4
(1 row)
```

lengthb(string)

描述：获取参数string中字节的数目。与字符集有关，同样的中文字符，在GBK与UTF8中，返回的字节数不同。

返回值类型：integer

示例：

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

substr(string,from)

描述：

从参数string中抽取子字符串。

from表示抽取的起始位置。

- from为0时，按1处理。
- from为正数时，抽取从from到末尾的所有字符。
- from为负数时，抽取字符串的后n个字符，n为from的绝对值。

返回值类型：varchar

示例：

from为正数时：

```
SELECT substr('ABCDEF',2);
 substr
--------
 BCDEF
(1 row)
```

from为负数时：

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

substr(string,from,count)

描述：

从参数string中抽取子字符串。

from表示抽取的起始位置。

count表示抽取的子字符串长度。

- from为0时，按1处理。
- from为正数时，抽取从from开始的count个字符。
- from为负数时，抽取从倒数第n个开始的count个字符，n为from的绝对值。
- count小于1时，返回null。

返回值类型：varchar

示例：

from为正数时：

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

from为负数时：

```
SELECT substr('ABCDEF',-3,2);
 substr
--------
 DE
(1 row)
```

substrb(string,from)

描述：该函数和SUBSTR(string,from)函数功能一致，但是计算单位为字节。

返回值类型：bytea

示例：

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

substrb(string,from,count)

描述：该函数和SUBSTR(string,from,count)函数功能一致，但是计算单位为字节。

返回值类型：bytea

示例：

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

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

描述：从字符串string的开头、结尾或两边删除只包含characters中字符（缺省是一个空白）的最长的字符串。

返回值类型：varchar

示例：

```
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)
```

rtrim(string [, characters])

描述：从字符串string的结尾删除只包含characters中字符（缺省是个空白）的最长的字符串。

返回值类型：varchar

示例：

```
SELECT rtrim('TRIMxxxx','x');
 rtrim
-------
 TRIM
(1 row)
```

ltrim(string [, characters])

描述：从字符串string的开头删除只包含characters中字符（缺省是一个空白）的最长的字符串。

返回值类型：varchar

示例：

```
SELECT ltrim('xxxxTRIM','x');
 ltrim
-------
 TRIM
(1 row)
```

upper(string)

描述：把字符串转化为大写。

返回值类型：varchar

示例：

```
SELECT upper('tom');
 upper
-------
 TOM
(1 row)
```

ucase(string)

描述：把字符串转化为大写。

返回值类型：varchar

示例：

```
SELECT ucase('sam');
 ucase
-------
 SAM
(1 row)
```

lower(string)

描述：把字符串转化为小写。

返回值类型：varchar

示例：

```
SELECT lower('TOM');
 lower
-------
 tom
(1 row)
```

lcase(string)

描述：把字符串转化为小写。

返回值类型：varchar

示例：

```
SELECT lcase('SAM');
 lcase
-------
 sam
(1 row)
```

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

描述：使用填充字符fill（缺省时为空白），把string填充到length长度。如果string已经比length长则将其从尾部截断。

length参数在DataArtsFabric SQL中表示字符长度。一个汉字长度计算为一个字符。

返回值类型：varchar

示例：

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

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

描述：从字符串string的position（缺省时为1）所指的位置开始查找并返回第occurrence（缺省时为1）次出现子串substring的位置的值。

- 当position为0时，返回0。
- 当position为负数时，从字符串倒数第n个字符往前逆向搜索。n为position的绝对值。

本函数以字符为计算单位，如一个汉字为一个字符。

返回值类型：integer

示例：

```
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])

描述：从字符串string的position（缺省时为1）所指的位置开始查找并返回第一次出现子串substring位置的值。以字符为计算单位。当string中不存在substring时，返回0。

返回值类型：integer

示例：

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

initcap(string)

描述：将字符串中的每个单词的首字母转化为大写，其他字母转化为小写。

返回值类型：text

示例：

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

ascii(string)

描述：参数string的第一个字符的ASCII码。

返回值类型：integer

示例：

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

replace(string varchar, search_string varchar, replacement_string varchar)

描述：把字符串string中所有子字符串search_string替换成子字符串replacement_string。

返回值类型：varchar

示例：

```
SELECT replace('jack and jue','j','bl');
   replace
----------------
 black and blue
(1 row)
```

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

描述：在string的左侧添上一系列的repeat_string（缺省为空白）来组成一个总长度为n的新字符串。

如果string本身的长度比指定的长度length长，则本函数将把string截断并把前面长度为length的字符串内容返回。

返回值类型：varchar

示例：

```
SELECT lpad('PAGE 1',15,'*.');
      lpad
-----------------
 *.*.*.*.*PAGE 1
(1 row)
SELECT lpad('hello world',5,'abcd');
 lpad
-------
 hello
(1 row)
```

concat(str1,str2)

描述：将字符串str1和str2连接并返回。

- ORA和TD兼容模式下，返回结果为所有非NULL字符串的连接。
- MySQL兼容模式下，入参中存在NULL时，返回结果为NULL。

返回值类型：varchar

示例：

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

chr(integer)

描述：给出ASCII码的字符。

返回值类型：varchar

示例：

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

regexp_substr(source_char, pattern)

描述：正则表达式的抽取子串函数。

返回值类型：varchar

示例：

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

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

描述：替换匹配POSIX正则表达式的子字符串。 如果没有匹配pattern，那么返回不加修改的string串。 如果有匹配，则返回的string串里面的匹配子串将被replacement串替换掉。

replacement串可以包含\n， 其中\n是1到9， 表明string串里匹配模式里第n个圆括号子表达式的子串应该被插入， 并且它可以包含\&表示应该插入匹配整个模式的子串。

可选的flags参数包含零个或多个改变函数行为的单字母标记，见下表。

表 8-17 flags 参数的可选项

选项	描述
g	表示替换每一个匹配的子字符串而不仅仅是第一个（默认仅替换第一个匹配的子字符串）
B	默认情况下使用Henry Spencer正则库及其正则语法。指定B选项后，表示优先选用boost regex正则库及其正则语法。 以下两种情况在指定了B选项时，也会自动转换为选择Henry Spencer正则库及其正则语法： ● flags同时指定了p、q、w、x中的任意个字符。 ● string或pattern参数中含有多字节字符。
b	表示按照BRE(POSIX Basic Regular Expression)匹配模式的规则进行匹配。
c	大小写敏感匹配

选项	描述
e	表示按照ERE(POSIX Extended Regular Expression)匹配模式的规则进行匹配。当b和e都未指定时，如果选用的是Henry Spencer正则库，则按照ARE(Advanced Regular Expression，类似于Perl Regular Expression)匹配模式的规则进行匹配；如果选用的是boost regex正则库，则按照Perl Regular Expression匹配模式的规则进行匹配。
i	大小写不敏感匹配
m	换行敏感匹配，与选项n同义。
n	换行敏感匹配。此选项生效时，换行符影响元字符(.、^、$和[^)的匹配。
p	部分换行敏感匹配，此选项生效时，换行符影响元字符(.和[^)的匹配。部分是相对选项n而言。
q	重置正则表达式为加双引号的文本字符串，所有都是普通字符。
s	非换行敏感匹配。
t	紧凑语法（缺省）。该选项生效时，所有字符都很重要。
w	反部分换行敏感匹配，此选项生效时，换行符影响元字符(^和$)的匹配。部分是相对选项n而言。
x	扩展语法。与紧凑语法相对，在扩展的语法中，正则表达式中的空白字符被忽略。空白字符包括空格、水平制表符、新行、和任何属于space字符表的字符。

返回值类型：varchar

示例：

```
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" [, ...]])

描述：以第一个参数为分隔符，链接第二个以后的所有参数。

返回值类型：text

示例：

```
SELECT concat_ws(',', 'ABCDE', 2, NULL, 22);
 concat_ws
------------
 ABCDE,2,22
(1 row)
```

convert(string bytea, src_encoding name, dest_encoding name)

描述：以dest_encoding指定的目标编码方式转化字符串bytea。src_encoding指定源编码方式，在该编码下，string必须是合法的。

返回值类型：bytea

示例：

```
SELECT convert('text_in_utf8', 'UTF8', 'GBK');
          convert
----------------------------
 \x746578745f696e5f75746638
(1 row)
```

说明

如果源编码格式到目标编码格式的转化规则不存在，则字符串不进行任何转换直接返回，如GBK和LATIN1之间的转换规则是不存在的，具体转换规则可以通过查看系统表pg_conversion获得。

示例：

```
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)
```

convert_from(string bytea, src_encoding name)

描述：以数据库的编码方式转化字符串bytea。

src_encoding指定源编码方式，在该编码下，string必须是合法的。

返回值类型：text

示例：

```
SELECT convert_from('text_in_utf8', 'UTF8');
 convert_from
--------------
 text_in_utf8
(1 row)
SELECT convert_from('\x6461746162617365','gbk');
 convert_from
--------------
 database
(1 row)
```

convert_to(string text, dest_encoding name)

描述：将字符串转化为dest_encoding的编码格式。

返回值类型：bytea

示例：

```
SELECT convert_to('some text', 'UTF8');
      convert_to
----------------------
 \x736f6d652074657874
(1 row)
SELECT convert_to('database', 'gbk');
     convert_to
--------------------
 \x6461746162617365
(1 row)
```

string [NOT] LIKE pattern [ESCAPE escape-character]

描述：模式匹配函数。

如果pattern不包含百分号或者下划线，该模式只代表它本身，这时候LIKE的行为就像等号操作符。在pattern里的下划线（_）匹配任何单个字符；而一个百分号（%）匹配零或多个任何字符。

要匹配下划线或者百分号本身，在pattern里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜杠，但是用户可以用ESCAPE子句指定一个。要匹配逃逸字符本身，写两个逃逸字符。

返回值类型：boolean

示例：

```
SELECT 'AA_BBCC' LIKE '%A@_B%' ESCAPE '@' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'AA_BBCC' LIKE '%A@_B%' AS RESULT;
 result
--------
 f
(1 row)
SELECT 'AA@_BBCC' LIKE '%A@_B%' AS RESULT;
 result
--------
 t
(1 row)
```

REGEXP_LIKE(source_string, pattern [, match_parameter])

描述：正则表达式的模式匹配函数。

source_string为源字符串，pattern为正则表达式匹配模式。 match_parameter为匹配选项，可取值为：

- 'i'：大小写不敏感。
- 'c'：大小写敏感。
- 'n'：允许正则表达式元字符“.”匹配换行符。

- 'm'：将source_string视为多行。

如果忽略match_parameter选项，默认为大小写敏感，“.”不匹配换行符，source_string视为单行。

返回值类型：boolean

示例：

```
SELECT regexp_like('ABC', '[A-Z]');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('ABC', '[D-Z]');
 regexp_like
-------------
 f
(1 row)
SELECT regexp_like('abc', '[A-Z]','i');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('abc', '[A-Z]');
 regexp_like
-------------
 f
(1 row)
```

format(formatstr text [, str"any" [, ...]])

描述：格式化字符串。

返回值类型：text

示例：

```
SELECT format('Hello %s, %1$s', 'World');
      format
--------------------
 Hello World, World
(1 row)
```

md5(string)

描述：将string使用MD5加密，并以16进制数作为返回值。

说明

MD5的安全性较低，不建议使用。

返回值类型：text

示例：

```
SELECT md5('ABC');
              md5
----------------------------------
 902fbdd2b1df0c4f70b4a5d23525e932
(1 row)
```

decode(string text, format text)

描述：将二进制数据从文本数据中解码。

返回值类型：bytea

示例：

```
SELECT decode('ZGF0YWJhc2U=', 'base64');
    decode
--------------
 \x6461746162617365
(1 row)

SELECT convert_from('\x6461746162617365','utf-8');
 convert_from
--------------
 database
(1 row)
```

encode(data bytea, format text)

描述：将二进制数据编码为文本数据。

返回值类型：text

示例：

```
SELECT encode('database', 'base64');
  encode
----------
 ZGF0YWJhc2U=
(1 row)
```

CONV(n, fromBase, toBase)

描述： 将给定的数值或者字符串转换成目标进制，并按照字符串的形式输出结果。如果参数含有NULL值，返回NULL。进制表示范围为[-36, -2]&[2, 36]。

返回值类型：text

示例：

```
SELECT CONV(-1, 10, 16) as result;
      result
------------------
 FFFFFFFFFFFFFFFF
(1 row)
```

HEX(n)

描述：n可以是int类型也可以是字符串。返回n的十六进制字符串。如果参数含有NULL值，返回NULL。

返回值类型：text

示例：

```
SELECT HEX(255) as result;
 result
--------
   FF
(1 row)
SELECT HEX('abc') as result;
 result
--------
 616263
(1 row)
```

UNHEX(n)

描述：执行HEX(n)的反向操作，n可以是int类型也可以是字符串，将参数中的每一对十六进制数字理解为一个数字，并将其转化为该数字代表的字符。如果参数含有NULL值，返回NULL。该函数仅8.2.0及以上集群版本支持。

返回值类型：bytea

示例：

```
SELECT UNHEX('abc') as result;
 result
--------
 \x0abc
(1 row)
```

SPACE(n int)

描述：返回由n个空格组成的字符串。如果参数含有NULL值，返回NULL。

返回值类型：text

示例：

```
SELECT SPACE(2) as result;
 result
--------

(1 row)
```

STRCMP(text, text)

描述：比较两个字符串大小，如果所有的字符串均相同，则返回0，如果根据当前分类次序，第一个字符串小于第二个，则返回-1，其它情况返回1。如果参数含有NULL值，返回NULL。

返回值类型：text

示例：

```
SELECT STRCMP('AA', 'AA'), STRCMP('AA', 'AB'), STRCMP('AA', 'A');
STRCMP  |  STRCMP  |  STRCMP
------------------------------
    0  |    -1    |     1
(1 row)
```

BIN(n bigint)

描述：将bigint类型从十进制转换成二进制，并以字符串的形式返回结果值。如果参数含有NULL值，则返回NULL。

返回值类型：text

示例：

```
SELECT BIN(16) as result;
 result
--------
 10000
(1 row)
```

8.3.2 二进制字符串函数和操作符

SQL定义了一些二进制字符串函数，这些函数使用关键字而不是逗号来分隔参数。另外，DataArtsFabric SQL提供了函数调用所使用的常用语法。

octet_length(string)

描述：二进制字符串中的字节数。

返回值类型：integer

示例：

```
SELECT octet_length(E'jo\\000se'::bytea) AS RESULT;
 result
--------
      5
(1 row)
```

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

描述：替换子串。

返回值类型：bytea

示例：

```
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)

描述：特定子字符串的位置。

返回值类型：integer

示例：

```
SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea) AS RESULT;
 result
--------
      3
(1 row)
```

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

描述：截取子串。

返回值类型：bytea

示例：

```
SELECT substring(E'Th\\000omas'::bytea from 2 for 3) AS RESULT;
  result
----------
 \x68006f
(1 row)
```

截取时间，获取小时数：

```
SELECT substring('2022-07-18 24:38:15',12,2)AS RESULT;
 result
-----------
 24
(1 row)
```

trim([both] bytes from string)

描述：从string的开头和结尾删除只包含bytes中字节的最长字符串。

返回值类型：bytea

示例：

```
SELECT trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea) AS RESULT;
  result
----------
 \x546f6d
(1 row)
```

substring_index(string, delim, count)

描述：按照区分大小写匹配查找delimiter分隔符，返回string字符串中第count次出现delim分隔符之前的子串。如果count为负数，则从末尾向前查找delim分隔符。如果参数含有NULL值，返回NULL。该函数仅8.2.0及以上集群版本支持。

返回值类型：text

示例：按照区分大小写匹配查找delimiter分隔符".wWw."，返回string字符串"www.wWw.cloud.wWw.com"中第2次出现delimiter分隔符之前的子串"www.wWw.cloud"。

```
SELECT SUBSTRING_INDEX('www.wWw.cloud.wWw.com', '.wWw.', 2) AS RESULT;
    result
---------------
 www.wWw.cloud
(1 row)
```

btrim(string bytea,bytes bytea)

描述：从string的开头和结尾删除只包含bytes中字节的最长的字符串。

返回值类型：bytea

示例：

```
SELECT btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea) AS RESULT;
   result
------------
 \x7472696d
(1 row)
```

get_bit(string, offset)

描述：从字符串中抽取位。

返回值类型：integer

示例：

```
SELECT get_bit(E'Th\\000omas'::bytea, 45) AS RESULT;
 result
--------
```



```
     1
(1 row)
```

get_byte(string, offset)

描述：从字符串中抽取字节。

返回值类型：integer

示例：

```
SELECT get_byte(E'Th\\000omas'::bytea, 4) AS RESULT;
 result
--------
    109
(1 row)
```

set_bit(string,offset, newvalue)

描述：设置字符串中的位。

返回值类型：bytea

示例：

```
SELECT set_bit(E'Th\\000omas'::bytea, 45, 0) AS RESULT;
      result
------------------
 \x5468006f6d4173
(1 row)
```

set_byte(string,offset, newvalue)

描述：设置字符串中的字节。

返回值类型：bytea

示例：

```
SELECT set_byte(E'Th\\000omas'::bytea, 4, 64) AS RESULT;
      result
------------------
 \x5468006f406173
(1 row)
```

8.3.3 位串函数和操作符

除了常用的比较操作符之外，还可以使用以下的操作符。&，|和#的位串操作数必须等长。在位移的时候，保留原始的位串长度（并以0填充）。

||

描述：位串之间进行连接。

示例：

```
SELECT B'10001' || B'011' AS RESULT;
  result
----------
 10001011
(1 row)
```

&

描述：位串之间进行“与”操作。

示例：

```
SELECT B'10001' & B'01101' AS RESULT;
 result
--------
 00001
(1 row)
```

|

描述：位串之间进行“或”操作。

示例：

```
SELECT B'10001' | B'01101' AS RESULT;
 result
--------
 11101
(1 row)
```

#

描述：位串之间如果不一致进行“或”操作。如果两个位串中对应位置都为1或者0，则该位置返回为0。

示例：

```
SELECT B'10001' # B'01101' AS RESULT;
 result
--------
 11100
(1 row)
```

~

描述：位串之间进行“非”操作。

示例：

```
SELECT ~B'10001'AS RESULT;
 result
----------
 01110
(1 row)
```

<<

描述：位串进行左移操作。

示例：

```
SELECT B'10001' << 3 AS RESULT;
 result
----------
 01000
(1 row)
```

>>

描述：位串进行右移操作。

示例：

```
SELECT B'10001' >> 2 AS RESULT;
 result
----------
 00100
(1 row)
```

下列SQL标准函数除了可以用于字符串之外，也可以用于位串：length，bit_length，octet_length，position，substring，overlay。

下列的函数用于位串和二进制字符串：get_bit，set_bit。当用于位串时，函数位数从字符串的第一位（最左边）作为0位。

另外，在整数和bit之间可以相互转换。示例：

```
SELECT 44::bit(10) AS RESULT;
   result
------------
 0000101100
(1 row)

SELECT 44::bit(3) AS RESULT;
 result
--------
 100
(1 row)

SELECT cast(-44 as bit(12)) AS RESULT;
    result
--------------
 111111010100
(1 row)

SELECT '1110'::bit(4)::integer AS RESULT;
 result
--------
     14
(1 row)
```

说明

只是转换为“bit”的意思是转换成bit(1)，因此只会转换成整数的最低位。

8.3.4 数字操作函数和操作符

8.3.4.1 数字操作符

+

描述：加

示例：

```
SELECT 2+3 AS RESULT;
 result
--------
      5
(1 row)
```

-

描述：减

示例：

```
SELECT 2-3 AS RESULT;
 result
--------
     -1
(1 row)
```

*

描述：乘

示例：

```
SELECT 2*3 AS RESULT;
 result
--------
      6
(1 row)
```

/

描述：除（除法操作符不会取整）

除数为0时，结果为NULL（兼容Hive行为）。

示例：

```
SELECT 4/2 AS RESULT;
 result
--------
      2
(1 row)
SELECT 4/3 AS RESULT;
      result
------------------
 1.33333333333333
(1 row)
```

+/-

描述：正/负

示例：

```
SELECT -2 AS RESULT;
 result
--------
     -2
(1 row)
```

%

描述：模（求余）

注：模数为0时，结果为NULL（兼容hive行为）

示例：

```
SELECT 5%4 AS RESULT;
 result
--------
      1
(1 row)
```

@

描述：绝对值

示例：

```
SELECT @ -5.0 AS RESULT;
 result
--------
    5.0
(1 row)
```

^

描述：幂（指数运算）

MySQL兼容模式下，作用为异或。

示例：

```
SELECT 2.0^3.0 AS RESULT;
       result
--------------------
 8.0000000000000000
(1 row)
```

|/

描述：平方根

示例：

```
SELECT |/ 25.0 AS RESULT;
 result
--------
      5
(1 row)
```

||/

描述：立方根

示例：

```
SELECT ||/ 27.0 AS RESULT;
 result
--------
      3
(1 row)
```

!

描述：阶乘

示例：

```
SELECT 5! AS RESULT;
 result
--------
    120
(1 row)
```

!!

描述：阶乘（前缀操作符）

示例：

```
SELECT !!5 AS RESULT;
 result
--------
    120
(1 row)
```

&

描述：二进制AND

示例：

```
SELECT 91&15  AS RESULT;
 result
--------
     11
(1 row)
```

|

描述：二进制OR

示例：

```
SELECT 32|3  AS RESULT;
 result
--------
     35
(1 row)
```

#

描述：二进制XOR

示例：

```
SELECT 17#5  AS RESULT;
 result
--------
     20
(1 row)
```

~

描述：二进制NOT

示例：

```
SELECT ~1 AS RESULT;
 result
--------
     -2
(1 row)
```

<<

描述：二进制左移

示例：

```
SELECT 1<<4 AS RESULT;
 result
--------
     16
(1 row)
```

>>

描述：二进制右移

示例：

```
SELECT 8>>2 AS RESULT;
 result
--------
      2
(1 row)
```

8.3.4.2 数字操作函数

abs(x)

描述：绝对值。

返回值类型：和输入相同。

示例：

```
SELECT abs(-17.4);
 abs
------
 17.4
(1 row)
```

acos(x)

描述：反余弦。

返回值类型：double precision

示例：

```
SELECT acos(-1);
       acos
------------------
 3.14159265358979
(1 row)
```

asin(x)

描述：反正弦。

返回值类型：double precision

示例：

```
SELECT asin(0.5);
       asin
------------------
 0.523598775598299
(1 row)
```

atan(x)

描述：反正切。

返回值类型：double precision

示例：

```
SELECT atan(1);
      atan
-------------------
 0.785398163397448
(1 row)
```

atan2(y, x)

描述：y/x的反正切。

返回值类型：double precision

示例：

```
SELECT atan2(2, 1);
     atan2
-------------------
 1.10714871779409
(1 row)
```

bitand(integer, integer)

描述：计算两个数字与运算(&)的结果。

返回值类型：bigint

示例：

```
SELECT bitand(127, 63);
 bitand
--------
     63
(1 row)
```

cbrt(double precision)

描述：立方根。

返回值类型：double precision

示例：

```
SELECT cbrt(27.0);
 cbrt
------
    3
(1 row)
```

ceil(double precision or numeric)

描述：不小于参数的最小的整数。

返回值类型：与输入相同。

示例：

```
SELECT ceil(-42.8);
 ceil
------
  -42
(1 row)
```

ceiling(double precision or numeric)

描述：不小于参数的最小整数（ceil的别名）。

返回值类型：与输入相同。

示例：

```
SELECT ceiling(-95.3);
 ceiling
---------
     -95
(1 row)
```

cos(x)

描述：余弦。

返回值类型：double precision

示例：

```
SELECT cos(-3.1415927);
        cos
-------------------
 -0.999999999999999
(1 row)
```

cot(x)

描述：余切。

返回值类型：double precision

示例：

```
SELECT cot(1);
       cot
------------------
 0.642092615934331
(1 row)
```

degrees(double precision)

描述：把弧度转为角度。

返回值类型：double precision

示例：

```
SELECT degrees(0.5);
     degrees
------------------
 28.6478897565412
(1 row)
```


div(y numeric, x numeric)

描述：y除以x的商的整数部分。

返回值类型：numeric

示例：

```
SELECT div(9,4);
 div
-----
   2
(1 row)
```

exp(double precision or numeric)

描述：自然指数。

返回值类型：与输入相同。

示例：

```
SELECT exp(1.0);
        exp
--------------------
 2.7182818284590452
(1 row)
```

floor(double precision or numeric)

描述：不大于参数的最大整数。

返回值类型：与输入相同。

示例：

```
SELECT floor(-42.8);
 floor
-------
   -43
(1 row)
```

fq_iceberg_bucket(int, smallint/int/bigint)

描述：计算指定整数的bucket数值。其中，第一个int参数为bucket数量。

返回值类型：int

示例：

```
fabricsql => select fq_iceberg_bucket(13, 5211314);
 fq_iceberg_bucket
-------------------
                 5
(1 row)
```

fq_iceberg_bucket(int, numeric(p, s))

描述：计算指定numeric数据的bucket数值。其中，第一个参数为bucket数量。

返回值类型：int

示例：

```
fabricsql => 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)

描述：计算不超过integer_value、且为W倍数的最大数值。

返回值类型：与第二个参数的数据类型一致。

示例：

```
fabricsql => select fq_iceberg_truncate (10, 19);
 fq_iceberg_truncate
---------------------
                10
(1 row)
```

fq_iceberg_truncate(int W, numeric(p, s) num_value)

描述：计算不超过num_value、且小数点右移s位后为W倍数的最大数值。

返回值类型：numeric(p, s)

示例：

```
fabricsql => select fq_iceberg_truncate (50, 10.64::numeric(15,3));
 fq_iceberg_truncate
---------------------
            10.600
(1 row)
```

更多fq_iceberg_truncate示例，请参见**分区表达式：truncate**。

radians(double precision)

描述：把角度转为弧度。

返回值类型：double precision

示例：

```
SELECT radians(45.0);
     radians
------------------
 0.785398163397448
(1 row)
```

random()

描述：0.0到1.0之间的随机数。

返回值类型：double precision

示例：

```
SELECT random();
      random
------------------
 0.696101832669228
(1 row)
```

rand()

描述：0.0到1.0之间的随机数。此函数为Mysql兼容性函数。该函数仅8.2.0及以上集群版本支持。

返回值类型：double precision

示例：

```
SELECT rand();
     rand
-------------------
 0.930654047057033
(1 row)
```

ln(double precision or numeric)

描述：自然对数。

返回值类型：与输入相同。

示例：

```
SELECT ln(2.0);
        ln
--------------------
 0.6931471805599453
(1 row)
```

log(double precision or numeric)

描述：以10为底的对数。

- ORA和TD兼容模式下，表现为以10为底的对数。
- MySQL兼容模式下，表现为自然对数。

返回值类型：与输入相同。

示例：

```
-- ORA兼容模式
SELECT log(100.0);
        log
--------------------
 2.0000000000000000
(1 row)
-- TD兼容模式
SELECT log(100.0);
        log
--------------------
 2.0000000000000000
(1 row)
-- MySQL兼容模式
SELECT log(100.0);
        log
--------------------
 4.6051701859880914
(1 row)
```

log(b numeric, x numeric)

描述：以b为底的对数。

返回值类型：numeric

示例：

```
SELECT log(2.0, 64.0);
        log
--------------------
 6.0000000000000000
(1 row)
```

mod(x,y)

描述：x/y的余数（模）。如果x是0，则返回0。如果y是0，则返回x。

返回值类型：与参数类型相同。

示例：

```
SELECT mod(9,4);
 mod
-----
   1
(1 row)
SELECT mod(9,0);
 mod
-----
   9
(1 row)
```

pi()

描述："π"常量。

返回值类型：double precision

示例：

```
SELECT pi();
        pi
------------------
 3.14159265358979
(1 row)
```

power(a double precision, b double precision)

描述：a的b次幂。

返回值类型：double precision

示例：

```
SELECT power(9.0, 3.0);
        power
----------------------
 729.0000000000000000
(1 row)
```

round(double precision or numeric)

描述：离输入参数最近的整数。

返回值类型：与输入相同。

示例：

```
SELECT round(42.4);
 round
```

```
-------
    42
(1 row)

SELECT round(42.6);
 round
-------
    43
(1 row)
```

说明

当调用round函数时，数值类型将舍入零，而（在大多数计算机上）实数和双精度型，以最接近的偶数为结果。

round(v numeric, s int)

描述：保留小数点后s位，s后一位进行四舍五入。

返回值类型：numeric

示例：

```
SELECT round(42.4382, 2);
 round
-------
 42.44
(1 row)
```

setseed(double precision)

描述：为随后的random()调用设置种子(-1.0到1.0之间，包含边界值)。

返回值类型：void

示例：

```
SELECT setseed(0.54823);
 setseed
---------

(1 row)
```

sign(double precision or numeric)

描述：输出此参数的符号。

返回值类型：-1表示负数，0表示0，1表示正数。

示例：

```
SELECT sign(-8.4);
 sign
------
   -1
(1 row)
```

sin(x)

描述：正弦。

返回值类型：double precision

示例：

```
SELECT sin(1.57079);
       sin
------------------
 0.999999999979986
(1 row)
```

sqrt(double precision or numeric)

描述：平方根。

返回值类型：与输入相同。

示例：

```
SELECT sqrt(2.0);
       sqrt
-------------------
 1.414213562373095
(1 row)
```

tan(x)

描述：正切。

返回值类型：double precision

示例：

```
SELECT tan(20);
       tan
------------------
 2.23716094422474
(1 row)
```

trunc(double precision or numeric)

描述：截断（取整数部分）。

返回值类型：与输入相同。

示例：

```
SELECT trunc(42.8);
 trunc
-------
    42
(1 row)
```

trunc(v numeric, s int)

描述：截断为s位小数。

返回值类型：numeric

示例：

```
SELECT trunc(42.4382, 2);
 trunc
-------
 42.43
(1 row)
```

truncate(v numeric, s int)

描述：将v截断为s位小数。v除了可以是任意精度型，还可以是整型和浮点型，返回值与入参v类型相同。s为负数时截断整数部分。该函数仅8.2.0及以上集群版本支持。

返回值类型：numeric

示例：

```
SELECT truncate(42.4382, 2);
 truncate
-------
 42.4300
(1 row)
```

width_bucket(operand numeric, b1 numeric, b2 numeric, count int)

描述：设定分组范围的最小值、最大值和分组个数，构建指定个数的大小相同的分组，返回指定字段值落入的分组编号。b1为分组范围的最小值，b2为分组范围的最大值，count为分组的个数。

返回值类型：integer

示例：

```
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)

描述：设定分组范围的最小值、最大值和分组个数，构建指定个数的大小相同的分组，返回指定字段值落入的分组编号。b1为分组范围的最小值，b2为分组范围的最大值，count为分组的个数。

返回值类型：integer

示例：

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
 width_bucket
--------------
            3
(1 row)
```

8.3.5 时间、日期处理函数和操作符

8.3.5.1 时间/日期操作符

须知

用户在使用时间和日期操作符时，对应的操作数请使用明确的类型前缀修饰，以确保数据库在解析操作数的时候能够与用户预期一致，不会产生用户非预期的结果。

例如下面示例没有明确数据类型就会出现异常错误。

```
SELECT date '2001-10-01' - '7' AS RESULT;
```

表 8-18 时间和日期操作符

<table>
<tr><th>操作符</th><th>示例</th></tr>
<tr><td rowspan="5">+</td><td>date类型参数与integer参数相加，获取时间间隔为7天后的时间：<pre>SELECT date '2001-09-28' + integer '7' AS RESULT;
      result

 2001-10-05
(1 row)</pre></td></tr>
<tr><td>date类型参数与interval参数相加，获取时间间隔为1小时后的时间：<pre>SELECT date '2001-09-28' + interval '1 hour' AS RESULT;
      result

 2001-09-28 01:00:00
(1 row)</pre></td></tr>
<tr><td>date类型参数与interval参数相加，获取时间间隔为1个月的时间：
date函数对于日期相加减超过月份的日期范围，会对齐到对应月份最后一天，不超过则不处理。<pre>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)</pre></td></tr>
<tr><td>date类型参数与time类型参数相加，获取具体的日期和时间结果：<pre>SELECT date '2001-09-28' + time '03:00' AS RESULT;
      result

 2001-09-28 03:00:00
(1 row)</pre></td></tr>
<tr><td>interval参数相加，获取两个时间间隔之和：<pre>SELECT interval '1 day' + interval '1 hour' AS RESULT;
    result
----
 1 day 01:00:00
(1 row)</pre></td></tr>
</table>

<table>
<tr><th>操作符</th><th>示例</th></tr>
<tr><td></td><td>timestamp时间类型参数与interval参数相加，获取间隔23小时后的时间：<pre>SELECT timestamp '2001-09-28 01:00' + interval '23 hours' AS RESULT;
       result

 2001-09-29 00:00:00
(1 row)</pre></td></tr>
<tr><td></td><td>time类型参数与interval参数相加，获取间隔时间为3小时后的时间：<pre>SELECT time '01:00' + interval '3 hours' AS RESULT;
  result

 04:00:00
(1 row)</pre></td></tr>
<tr><td>-</td><td>date类型参数相减，获取两个日期的时间差：<pre>SELECT date '2001-10-01' - date '2001-09-28' AS RESULT;
 result

 3
(1 row)</pre></td></tr>
<tr><td></td><td>date类型参数与integer参数相减，返回timestamp类型，获取两者的时间差：<pre>SELECT date '2001-10-01' - integer '7' AS RESULT;
       result

 2001-09-24
(1 row)</pre></td></tr>
<tr><td></td><td>date类型参数与interval参数相减，获取两者的日期、时间差：<pre>SELECT date '2001-09-28' - interval '1 hour' AS RESULT;
       result

 2001-09-27 23:00:00
(1 row)</pre></td></tr>
<tr><td></td><td>time类型参数相减，获取两参数的时间差：<pre>SELECT time '05:00' - time '03:00' AS RESULT;
  result

 02:00:00
(1 row)</pre></td></tr>
<tr><td></td><td>time类型参数与interval相减，获取两参数的时间差：<pre>SELECT time '05:00' - interval '2 hours' AS RESULT;
  result

 03:00:00
(1 row)</pre></td></tr>
<tr><td></td><td>timestamp类型参数与interval相减，从时间戳中减去时间间隔，获取两者的日期时间差：<pre>SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT;
       result

 2001-09-28 00:00:00
(1 row)</pre></td></tr>
</table>

<table>
<tr><th>操作符</th><th>示例</th></tr>
<tr><td rowspan="3"></td><td>interval参数相减，获取两者的时间差：<pre>SELECT interval '1 day' - interval '1 hour' AS RESULT;
 result

 23:00:00
(1 row)</pre></td></tr>
<tr><td>timestamp类型参数相减，获取两者的日期时间差：<pre>SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT;
     result
----
 1 day 15:00:00
(1 row)</pre></td></tr>
<tr><td>获取当前日期的前一天：<pre>select now() - interval '1 day'AS RESULT;
           result

 2022-08-08 01:46:15.555406+00
(1 row)</pre></td></tr>
<tr><td rowspan="3">*</td><td>将时间间隔乘以数量：<pre>SELECT 900 * interval '1 second' AS RESULT;
 result

 00:15:00
(1 row)</pre></td></tr>
<tr><td><pre>SELECT 21 * interval '1 day' AS RESULT;
 result
----
 21 days
(1 row)</pre></td></tr>
<tr><td><pre>SELECT double precision '3.5' * interval '1 hour' AS RESULT;
 result

 03:30:00
(1 row)</pre></td></tr>
<tr><td>/</td><td>用时间间隔除以数量，获取一段时间中的某一段：<pre>SELECT interval '1 hour' / double precision '1.5' AS RESULT;
 result

 00:40:00
(1 row)</pre></td></tr>
</table>

8.3.5.2 时间/日期函数

age(timestamp, timestamp)

描述：将两个参数相减，并以年、月、日作为返回值。如果相减值为负，则函数返回亦为负。

返回值类型：interval

示例：

```
SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
           age
```

```
-------------------------
 43 years 9 mons 27 days
(1 row)
```

age(timestamp)

描述：当前时间和参数相减。

返回值类型：interval

示例：

```
SELECT age(timestamp '1957-06-13');
          age
-------------------------
 60 years 2 mons 18 days
(1 row)
```

adddate(date, interval | int)

描述：返回给定日期时间加上指定单位的时间间隔的结果。默认单位(即第二个参数为整型时)为天数。

返回值类型：timestamp

示例：

当入参为text类型时：

```
select adddate('2020-11-13', 10);
  adddate
------------
 2020-11-23
(1 row)

select adddate('2020-11-13', interval '1' month);
  adddate
------------
 2020-12-13
(1 row)

select adddate('2020-11-13 12:15:16', interval '1' month);
       adddate
---------------------
 2020-12-13 12:15:16
(1 row)


select adddate('2020-11-13', interval '1' minute);
       adddate
---------------------
 2020-11-13 00:01:00
(1 row)
```

当入参为date类型时：

```
select adddate(current_date, 10);
  adddate
------------
 2021-09-24
(1 row)

select adddate(date '2020-11-13', interval '1' month);
       adddate
---------------------
 2020-12-13 00:00:00
(1 row)
```


addtime(timestamp | time | text, interval | text)

描述：返回给定日期/时间加上指定时间间隔的结果。该函数仅8.2.0及以上集群版本支持。

返回值类型：与第一个入参类型相同。

示例：

```
select addtime('2020-11-13 01:01:01', '23:59:59');
       addtime
---------------------
 2020-11-14 01:01:00
(1 row)
```

subdate(date, interval | int)

描述：返回给定日期时间减去指定单位的时间间隔的结果; 默认单位(即第二个参数为整型时)为天数。

返回值类型：timestamp

示例：

当入参为text类型时：

```
select subdate('2020-11-13', 10);
  subdate
------------
 2020-11-03
(1 row)

select subdate('2020-11-13', interval '2' month);
  subdate
------------
 2020-09-13
(1 row)

select subdate('2020-11-13 12:15:16', interval '1' month);
       subdate
---------------------
 2020-10-13 12:15:16
(1 row)

select subdate('2020-11-13', interval '2' minute);
       subdate
---------------------
 2020-11-12 23:58:00
(1 row)
```

当入参为date类型时：

```
select subdate(current_date, 10);
  subdate
------------
 2021-09-05
(1 row)

select subdate(current_date, interval '1' month);
       subdate
---------------------
 2021-08-15 00:00:00
(1 row)
```

subtime(timestamp | time | text, interval | text)

描述：返回给定日期/时间减去指定时间间隔的结果。该函数仅8.2.0及以上集群版本支持。

返回值类型：与第一个入参类型相同。

示例：

```
select subtime('2020-11-13 01:01:01', '23:59:59');
      addtime
---------------------
 2020-11-12 01:01:02
(1 row)
```

date_add(date, interval)

描述：返回给定日期时间加上指定单位的时间间隔的结果。等效于**adddate(date, interval | int)**。

返回值类型：timestamp

date_sub(date, interval)

描述：返回给定日期时间减去指定单位的时间间隔的结果，等效于**subdate(date, interval | int)**。

返回值类型：timestamp

timestampadd(field, numeric, timestamp)

描述：将以单位field的整数时间间隔（秒数可以带小数）添加到日期时间表达式中。如果数值为负，则表示从给定的时间日期时间表达式中减去对应的时间间隔。field支持的参数为year，month，quarter，day，week，hour，minute，second和microsecond。

返回值类型：timestamp

示例：

```
select timestampadd(year, 1, timestamp '2020-2-29');
   timestampadd
---------------------
 2021-02-28 00:00:00
(1 row)

select timestampadd(second, 2.354156, timestamp '2020-11-13');
       timestampadd
----------------------------
 2020-11-13 00:00:02.354156
(1 row)
```

timestampdiff(field, timestamp1, timestamp2)

描述：将两个日期参数相减(timestamp2 - timestamp1)，并以单位field作为返回值。如果相减值为负，则函数返回值为负。field支持的参数为year、month、quarter、day、week、hour、minute、second和microsecond。

返回值类型：bigint

示例：

```
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)

描述：将两个日期参数相减。如果相减值为负，则函数返回值为负。两个入参类型要求一致。该函数仅8.2.0及以上集群版本支持。

返回值类型：time

示例：

```
SELECT timediff('2022-7-5 1:1:1', '2021-7-5 1:1:1');
   timediff
---------------
  8760:00:00
(1 row)
```

clock_timestamp()

描述：实时时钟的当前时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT clock_timestamp();
        clock_timestamp
-------------------------------
 2017-09-01 16:57:36.636205+08
(1 row)
```

current_date

描述：当前日期。

返回值类型：date

示例：

```
SELECT current_date;
    current_date
------------
 2017-09-01
(1 row)
```

curdate()

描述：当前日期。此函数为Mysql兼容性函数。该函数仅8.2.0及以上集群版本支持。

返回值类型：date

示例：

```
SELECT curdate();
   curdate
------------
 2022-09-19
(1 row)
```

current_time

描述：当前时间。

返回值类型：time with time zone

示例：

```
SELECT current_time;
      current_time
--------------------
 16:48:27.59887
(1 row)
```

curtime([fsp])

描述：当前时间。fsp为选填参数，参数类型为整型，表示指定返回的小数秒精度。该函数仅8.2.0及以上集群版本支持。

返回值类型：time with time zone

示例：

```
SELECT curtime();
      curtime
--------------------
 16:46:40.759849
(1 row)
SELECT curtime(2);
      curtime
--------------------
 16:47:13.9
(1 row)
```

current_timestamp

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
SELECT current_timestamp;
      current_timestamp
------------------------------
 2025-01-22 16:45:41.935586
(1 row)
```

convert_tz(timestamp, from_tz, to_tz)

描述：将时间日期值timestamp从*from_tz* (text)给出的时区转到*to_tz*(text)给出的时区，返回转换后的时区时间。该函数仅8.2.0及以上集群版本支持。

返回值类型：timestamp without time zone

示例：

```
SELECT convert_tz('2018-12-25 13:25:00', '+02:00', '+08:00');
   convert_tz
--------------------
 2018-12-25 19:25:00
(1 row)
SELECT convert_tz('2000-02-28 23:00:00', 'GMT', 'MET');
   convert_tz
```

```
--------------------
 2000-02-29 00:00:00
(1 row)
```

datediff(date1, date2)

描述：返回给定日期之间相差的天数值。

返回值类型：integer

示例：

```
select datediff(date '2020-11-13', date '2012-10-16');
 datediff
----------
     2950
(1 row)
```

date_part(text, timestamp)

描述：获取参数text指定的精度。

等效于extract(field from timestamp)。

返回值类型：double precision

示例：

```
SELECT date_part('hour', timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

date_part(text, interval)

描述：获取参数text指定的精度。如果大于12，则取与12的模。

等效于extract(field from timestamp)。

返回值类型：double precision

示例：

```
SELECT date_part('month', interval '2 years 3 months');
 date_part
-----------
         3
(1 row)
```

date_trunc(text, timestamp)

描述：截取到参数text指定的精度。

返回值类型：timestamp

示例：

```
SELECT date_trunc('hour', timestamp  '2001-02-16 20:38:40');
     date_trunc
---------------------
 2001-02-16 20:00:00
(1 row)
```

```
--获取去年的最后一天
SELECT date_trunc('day', date_trunc('year',CURRENT_DATE)+ '-1');
       date_trunc
------------------------
 2022-12-31 00:00:00+08
(1 row)

--获取今年的第一天
SELECT date_trunc('year',CURRENT_DATE);
       date_trunc
------------------------
 2023-01-01 00:00:00+08
(1 row)

--获取去年的第一天
SELECT date_trunc('year',now() + '-1 year');
       date_trunc
------------------------
 2022-01-01 00:00:00+08
(1 row)
```

trunc(timestamp)

描述：默认按天截取。

返回值类型：timestamp

示例：

```
SELECT trunc(timestamp  '2001-02-16
20:38:40');
                    trunc
---------------------
2001-02-16 00:00:00
(1 row)
```

extract(field from timestamp)

描述：获取field指定精度的值。field的有效值参见**EXTRACT**。

返回值类型：double precision

示例：

```
SELECT extract(hour from timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

extract(field from interval)

描述：获取field指定精度的值。如果大于12，则取与12的模。field的有效值参见**EXTRACT**。

返回值类型：double precision

示例：

```
SELECT extract(month from interval '2 years 3 months');
 date_part
-----------
         3
(1 row)
```

day(date)

描述：获取日期时间date所处月份中的天数，与dayofmonth函数相同。

取值范围：1~31

返回值类型：integer

示例：

```
select day('2020-06-28');
 day
-----
  28
(1 row)
```

dayofmonth(date)

描述：获取日期时间date所处月份中的天数。

取值范围：1~31

返回值类型：integer

示例：

```
select dayofmonth('2020-06-28');
 dayofmonth
------------
         28
(1 row)
```

dayofweek(date)

描述：返回给定日期date对应的星期索引，星期日作为一周的开始日。

取值范围：1~7

返回值类型：integer

示例：

```
select dayofweek('2020-11-22');
 dayofweek
-----------
         1
(1 row)
```

dayofyear(date)

描述：返回给定日期date在本年中的天数。

取值范围：1~366

返回值类型：integer

示例：

```
select dayofyear('2020-02-29');
 dayofyear
-----------
        60
(1 row)
```

fq_iceberg_year(date/timestamp)

描述：返回给定日期相对1970年、以年为单位的数值。

返回值类型：integer

示例：

```
fabricsql => select fq_iceberg_year('2025-10-10');
 fq_iceberg_year
-----------------
              55
(1 row)
```

fq_iceberg_month(date/timestamp)

描述：返回给定日期相对1970-01-01、以月为单位的数值。

返回值类型：integer

示例：

```
fabricsql => select fq_iceberg_month('2025-10-10');
 fq_iceberg_month
-----------------
             669
(1 row)
```

fq_iceberg_day(date/timestamp)

描述：返回给定日期相对1970-01-01、以天为单位的数值。

返回值类型：integer

示例：

```
fabricsql => select fq_iceberg_day('2025-10-10');
 fq_iceberg_day
----------------
         20371
(1 row)
```

fq_iceberg_hour(timestamp)

描述：返回给定日期相对1970-01-01 00:00:00、以小时为单位的数值。

返回值类型：integer

示例：

```
fabricsql => select  fq_iceberg_hour ('2025-10-10 10:10:10.000') ;
 fq_iceberg_hour
-----------------
         488914
(1 row)
```

关于fq_iceberg_year、fq_iceberg_month、fq_iceberg_day、fq_iceberg_hour更多信息，请参考支持的分区表达式。

fq_iceberg_bucket(int, date/timestamp)

描述：计算指定时间数值的bucket数值。其中，第一个int参数为bucket数量。

返回值类型：int

示例：

```
fabricsql => select fq_iceberg_bucket(13, '2025-10-11'::date);
 fq_iceberg_bucket
-------------------
                 8
(1 row)

fabricsql => select fq_iceberg_bucket(13, '2025-10-11 11:11:11.222'::timestamp);
 fq_iceberg_bucket
-------------------
                 1
(1 row)
```

hour(timestamp with time zone)

描述：获取时间中的小时值。

返回值类型：integer

示例：

```
SELECT hour(timestamptz '2018-12-13 12:11:15+06');
 hour
------
   14
(1 row)
```

isfinite(date)

描述：测试是否为有限日期。

返回值类型：boolean

示例：

```
SELECT isfinite(date '2001-02-16');
 isfinite
----------
 t
(1 row)
SELECT isfinite(date 'infinity');
 isfinite
----------
 f
(1 row)
```

isfinite(timestamp)

描述：测试判断是否为有限时间。

返回值类型：boolean

示例：

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');
 isfinite
----------
 t
(1 row)
SELECT isfinite(timestamp 'infinity');
 isfinite
----------
```

```
 f
(1 row)
```

isfinite(interval)

描述：测试是否为有限区间。

返回值类型：boolean

示例：

```
SELECT isfinite(interval '4 hours');
 isfinite
----------
 t
(1 row)
```

justify_days(interval)

描述：将时间间隔以30天为单位，表示为月。

返回值类型：interval

示例：

```
SELECT justify_days(interval '35 days');
 justify_days
--------------
 1 mon 5 days
(1 row)
```

justify_hours(interval)

描述：将时间间隔以24小时为单位，表示为天。

返回值类型：interval

示例：

```
SELECT JUSTIFY_HOURS(INTERVAL '27 HOURS');
 justify_hours
----------------
 1 day 03:00:00
(1 row)
```

justify_interval(interval)

描述：结合justify_days和justify_hours，调整interval。

返回值类型：interval

示例：

```
SELECT JUSTIFY_INTERVAL(INTERVAL '1 MON -1 HOUR');
 justify_interval
------------------
 29 days 23:00:00
(1 row)
```

localtime

描述：当前时间。

返回值类型：time

示例：

```
SELECT localtime AS RESULT;
     result
----------------
 2025-01-17 09:48:08.056091
(1 row)
```

localtimestamp

描述：当前日期及时间。

返回值类型：timestamp

示例：

```
SELECT localtimestamp;
         localtimestamp
----------------------------
 2017-09-01 17:03:30.781902
(1 row)
```

makedate(year, dayofyear)

描述：根据给定的年份和一年中的天数返回相对应的日期值。

返回值类型：date

示例：

```
select makedate(2020, 60);
  makedate
------------
 2020-02-29
(1 row)
```

maketime(hour, minute, second)

描述：根据所给的小时，分钟和秒数返回time类型的值。由于DataArtsFabric SQL中的time类型的取值范围为00:00:00到24:00:00, 故不支持hour大于24时和hour小于0时的场景。

返回值类型：time

示例：

```
select maketime(12, 15, 30.12);
  maketime
-------------
 12:15:30.12
(1 row)
```

microsecond(timestamp with time zone)

描述：获取时间中的微秒值。

返回值类型：integer

示例：

```
SELECT microsecond(timestamptz '2018-12-13 12:11:15.123634+06');
 microsecond
```

```
-------------
     123634
(1 row)
```

minute(timestamp with time zone)

描述：获取时间中的分钟值。

返回值类型：integer

示例：

```
SELECT minute(timestamptz '2018-12-13 12:11:15+06');
 minute
--------
    11
(1 row)
```

month(date)

描述：返回给定日期时间的月份。

返回值类型：integer

示例：

```
select month('2020-11-30');
 month
-------
   11
(1 row)
```

now([fsp])

描述：当前事务开始的日期及时间，参数确定微秒输出精度，缺省时为6。

返回值类型：timestamp with time zone

示例：

```
SELECT now();
              now
-------------------------------
 2017-09-01 17:03:42.549426+08
(1 row)
SELECT now(3);
            now
----------------------------
 2021-09-08 10:59:00.427+08
(1 row)
```

numtodsinterval(num, interval_unit)

描述：将数字转换为interval类型。num为numeric类型数字，interval_unit为固定格式字符串（'DAY' | 'HOUR' | 'MINUTE' | 'SECOND'）。

可以通过设置参数IntervalStyle为oracle，兼容该函数在Oracle中的interval输出格式。

示例：

```
SELECT numtodsinterval(100, 'HOUR');
 numtodsinterval
-----------------
 100:00:00
```

```
(1 row)

SET intervalstyle = oracle;
SET
SELECT numtodsinterval(100, 'HOUR');
       numtodsinterval
-------------------------------
 +000000004 04:00:00.000000000
(1 row)
```

pg_sleep(seconds)

描述：是服务器线程延迟时间，以秒计。

返回值类型：void

示例：

```
SELECT pg_sleep(10);
 pg_sleep
----------

(1 row)
```

period_add(P, N)

描述：返回给定时期加上N个月后的日期。

返回值类型：integer

示例：

```
select period_add(200801, 2);
 period_add
------------
     200803
(1 row)
```

period_diff(P1, P2)

描述：返回给定日期之间的月数差值。

返回值类型：integer

```
select period_diff(200802, 200703);
 period_diff
-------------
          11
(1 row)
```

quarter(date)

描述：获取日期date所属的季度。

返回值类型：integer

示例：

```
SELECT quarter(date '2018-12-13');
 quarter
---------
       4
(1 row)
```

second(timestamp with time zone)

描述：获取时间的秒数值。

返回值类型：integer

示例：

```
SELECT second(timestamptz '2018-12-13 12:11:15+06');
 second
--------
     15
(1 row)
```

statement_timestamp()

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
SELECT statement_timestamp();
      statement_timestamp
-------------------------------
 2017-09-01 17:04:39.119267+08
(1 row)
```

sysdate

描述：当前日期及时间。

返回值类型：timestamp

示例：

```
SELECT sysdate;
       sysdate
---------------------
 2017-09-01 17:04:49
(1 row)
```

timeofday()

描述：当前日期及时间（像clock_timestamp，但是返回时为text。）

返回值类型：text

示例：

```
SELECT timeofday();
              timeofday
-------------------------------------
 Fri Sep 01 17:05:01.167506 2017 CST
(1 row)
```

transaction_timestamp()

描述：当前日期及时间，与current_timestamp等效。

返回值类型：timestamp with time zone

示例：

```
SELECT transaction_timestamp();
     transaction_timestamp
-------------------------------
 2017-09-01 17:05:13.534454+08
(1 row)
```

from_unixtime(unix_timestamp[,format])

描述：格式串缺省时，将unix时间戳转换为日期时间类型输出。格式串指定时，将unix时间戳转换为指定格式的字符串输出。

返回值类型：timestamp（格式串缺省）/ text（格式串指定）

示例：

```
SELECT from_unixtime(875996580);
    from_unixtime
---------------------
 1997-10-04 20:23:00
(1 row)
SELECT from_unixtime(875996580, '%Y %D %M %h:%i:%s');
       from_unixtime
---------------------------
 1997 5th October 04:23:00
(1 row)
```

unix_timestamp([timestamp with time zone])

描述：获取从'1970-01-01 00:00:00'UTC到入参时间经历的秒数。无入参时，指定为当前时间。

返回值类型：bigint（无入参）/numeric（有入参）

示例：

```
SELECT unix_timestamp();
 unix_timestamp
----------------
     1693906219
(1 row)
SELECT unix_timestamp('2018-09-08 12:11:13+06');
 unix_timestamp
----------------
     1536387073.000000
(1 row)
```

add_months(d,n)

描述：用于计算时间点d再加上n个月的时间。

返回值类型：timestamp

示例：

```
SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11) FROM dual;
     add_months
---------------------
 2018-04-29
(1 row)
```

last_day(d)

描述：用于计算时间点d本月最后一天的时间。

- ORA和TD兼容模式下，返回值类型为timestamp。
- MySQL兼容模式下，返回值类型为date。

示例：

```
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)

描述：用于计算x时间开始的下一个星期y的时间。

- ORA和TD兼容模式下，返回值类型为timestamp。
- MySQL兼容模式下，返回值类型为date。

示例：

```
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)

描述：根据给定的天数，返回相对应的日期值。

返回值类型：date

示例：

```
select from_days(730669);
 from_days
------------
 2000-07-03
(1 row)
```

to_days(timestamp)

描述：返回自0年开始到入参日期的天数。

返回值类型：integer

示例：

```
SELECT to_days(timestamp '2008-10-07');
 to_days
---------
  733687
(1 row)
```

8.3.5.3 EXTRACT

EXTRACT(*field* **FROM** *source***)**

extract函数从日期或时间的数值里抽取子域，例如年、小时等。source必须是一个timestamp、time或interval类型的值表达式（类型为date的表达式转换为timestamp，因此也可以用）。field是一个标识符或者字符串，它指定从源数据中抽取的域。extract函数返回类型为double precision的数值。field的取值范围如下所示。

century

世纪。

第一个世纪从0001-01-01 00:00:00 AD开始。这个定义适用于所有使用阳历的国家。没有0世纪，直接从公元前1世纪到公元1世纪。

示例：

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
 date_part
-----------
        20
(1 row)
```

day

- 如果source为timestamp，表示月份里的日期（1-31）。
  ```
  SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
   date_part
  -----------
          16
  (1 row)
  ```
- 如果source为interval，表示天数。
  ```
  SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
   date_part
  -----------
          40
  (1 row)
  ```

decade

年份除以10。

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
       200
(1 row)
```

dow

每周的星期几，星期天（0）到星期六（6）。

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         5
(1 row)
```

doy

一年的第几天（1~365/366）。

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        47
(1 row)
```


epoch

- 如果source为timestamp with time zone，表示自1970-01-01 00:00:00-00 UTC以来的秒数（结果可能是负数）；

  如果source为date和timestamp，表示自1970-01-01 00:00:00-00当地时间以来的秒数；

  如果source为interval，表示时间间隔的总秒数。

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
  date_part
--------------
 982384720.12
(1 row)
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
 date_part
-----------
    442800
(1 row)
```

- 将epoch值转换为时间戳的方法。

```
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

小时域（0-23）。

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

isodow

一周的第几天（1-7）。

星期一为1，星期天为7。

说明

除了星期天外，都与dow相同。

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
 date_part
-----------
         7
(1 row)
```

isoyear

日期中的ISO 8601标准年（不适用于间隔）。

每个带有星期一开始的周中包含1月4日的ISO年，所以在年初的1月或12月下旬的ISO年可能会不同于阳历的年。详细信息请参见后续的week描述。

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
 date_part
-----------
      2005
(1 row)
```

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
 date_part
-----------
      2006
(1 row)
```

microseconds

秒域（包括小数部分）乘以1,000,000。

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
 date_part
-----------
  28500000
(1 row)
```

millennium

千年。

20世纪（19xx年）里面的年份在第二个千年里。第三个千年从2001年1月1日零时开始。

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         3
(1 row)
```

milliseconds

秒域（包括小数部分）乘以1000。请注意它包括完整的秒。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
 date_part
-----------
     28500
(1 row)
```

minute

分钟域（0-59）。

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        38
(1 row)
```

month

如果source为timestamp，表示一年里的月份数（1-12）。

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         2
(1 row)
```

如果source为interval，表示月的数目，然后对12取模（0-11）。

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
 date_part
-----------
```

```
        1
(1 row)
```

quarter

该天所在的该年的季度（1-4）。

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         1
(1 row)
```

second

秒域，包括小数部分（0-59）。

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
 date_part
-----------
      28.5
(1 row)
```

timezone

与UTC的时区偏移量，单位为秒。正数对应UTC东边的时区，负数对应UTC西边的时区。

```
SELECT EXTRACT(timezone FROM TIMETZ '17:12:28');
 date_part
-----------
     28800
(1 row)
```

timezone_hour

时区偏移量的小时部分。

```
SELECT EXTRACT(timezone_hour FROM TIMETZ '17:12:28');
 date_part
-----------
         8
(1 row)
```

timezone_minute

时区偏移量的分钟部分。

```
SELECT EXTRACT(timezone_minute FROM TIMETZ '17:12:28');
 date_part
-----------
         0
(1 row)
```

week

该天在所在的年份里是第几周。ISO 8601定义一年的第一周包含该年的一月四日（ISO-8601 的周从星期一开始）。换句话说，一年的第一个星期四在第一周。

在ISO定义里，一月的头几天可能是前一年的第52或者第53周，十二月的后几天可能是下一年第一周。例如，2005-01-01是2004年的第53周，而2006-01-01是2005年的

第52周，2012-12-31是2013年的第一周。建议isoyear字段和week一起使用以得到一致的结果。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         7
(1 row)
```

year

年份域。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
      2001
(1 row)
```

8.3.5.4 date_part

date_part

date_part函数是在传统的Ingres函数的基础上制作的（该函数等效于SQL标准函数extract）：

date_part('*field*', *source*)

这里的field参数必须是一个字符串，而不是一个名字。有效的field与extract一样，详细信息请参见**EXTRACT**。

示例：

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        16
(1 row)
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
 date_part
-----------
         4
(1 row)
```

8.3.5.5 date_format

date_format(timestamp, fmt)

date_format函数将日期参数按照fmt指定的格式转换为字符串。

示例：

```
SELECT date_format('2009-10-04 22:23:00', '%M %D %W');
    date_format
--------------------
 October 4th Sunday
(1 row)
SELECT date_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');
     date_format
---------------------
 2021-02-20 08:30:45
(1 row)
```

```
SELECT date_format('2021-02-20 18:10:15', '%r-%T');
     date_format
----------------------
 06:10:15 PM-18:10:15
(1 row)
```

表8-19显示了可以用于将日期参数格式化输出的格式类型，这些格式类型适用于函数date_format、time_format、str_to_date、str_to_time和from_unixtime。

表 8-19 date_format 支持的输出格式

格式	说明	取值
%a	缩写星期名	Sun...Sat
%b	缩写月份名	Jan...Dec
%c	月份	0...12
%D	带英文后缀的月份日期	0th, 1st, 2nd, 3rd, …
%d	一个月里的日，2位	00...31
%e	一个月里的日	0...31
%f	微秒	000000...999999
%H	小时，24小时制	00...23
%h	小时，12小时制	01...12
%I	小时，12小时制，同%h	01...12
%i	分钟	00...59
%j	一年里的日	001...366
%k	小时，24小时制，同%H	0...23
%l	小时，12小时制，同%h	1...12
%M	月份名	January...December
%m	月份，两位	00...12
%p	上下午	AM PM
%r	时间，12小时制	hh::mm::ss AM/PM
%S	秒	00...59
%s	秒，同%S	00...59
%T	时间，24小时制	hh::mm::ss
%U	周（00-53）星期日是一周的第一天	00...53
%u	周（00-53）星期一是一周的第一天	00...53

格式	说明	取值
%V	周 (01-53) 星期日是一周的第一天，与%X搭配使用	01...53
%v	周 (01-53) 星期一是一周的第一天，与%x搭配使用	01...53
%W	星期名	Sunday...Saturday
%w	一周的日，周日为0	0...6
%X	年份，其中的星期日是周的第一天，4 位，与%V搭配使用	-
%x	年份，其中的星期一是周的第一天，4 位，与%v搭配使用	-
%Y	年份，4位	-
%y	年份，2位	-
%%	字符'%'	字符'%'
%*x*	'*x*'，上述未列出的任意字符	字符'*x*'

须知

date_format支持的输出格式中，%U、%u、%V、%v、%X、%x暂不支持。

8.3.5.6 time_format

time_format(time, fmt)

描述：time_format函数将日期参数按照fmt指定的格式转换为字符串。与date_format函数类似，但格式字符串只能包含小时、分钟、秒和微秒的格式说明符，如果包含其他说明符则会返回NULL值或0。

返回值类型：text

示例：

```
SELECT time_format('2009-10-04 22:23:00', '%M %D %W');
    time_format
--------------------

(1 row)
SELECT time_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');
     time_format
---------------------
 0000-00-00 08:30:45
(1 row)
SELECT time_format('2021-02-20 18:10:15', '%r-%T');
     time_format
----------------------
 06:10:15 PM-18:10:15
(1 row)
```

须知

time_format仅支持时间相关的格式输出（%f、%H、%h、%I、%i、%k、%l、%p、%r、%S、%s、%T），不支持日期相关格式，其他情况处理为普通字符。

str_to_date(str, format)

描述：将日期/时间格式的字符串（str），按照所提供的显示格式（format）转换为日期类型的值。

返回值类型：timestamp

示例：

```
SELECT str_to_date('01,5,2021','%d,%m,%Y');
     str_to_date
---------------------
 2021-05-01 00:00:00
(1 row)
SELECT str_to_date('01,5,2021,09,30,17','%d,%m,%Y,%h,%i,%s');
     str_to_date
---------------------
 2021-05-01 09:30:17
(1 row)
```

适用于str_to_date的格式化输入的格式类型参考**表8-19**。这里仅支持“日期”格式、“日期+时间”格式的输入转换，对于仅“时间”格式的输入场景请使用str_to_time。

str_to_time(str, format)

描述：将时间格式的字符串（str），按照所提供的显示格式（format）转换为时间类型的值。

返回值类型：time

示例：

```
SELECT str_to_time('09:30:17','%h:%i:%s');
 str_to_time
-------------
 09:30:17
(1 row)
```

适用于str_to_time的格式化输入的格式类型参考**表8-19**，这里仅支持“时间”格式的输入转换，对于“日期”格式、“日期+时间”格式的输入场景请使用str_to_date。

week(date[, mode])

描述：根据模式返回指定日期时间所处年份中对应的周数，默认模式为0。

返回值类型：integer

表 8-20 week 函数中 mode 模式的工作原理

模式	一周的第一天	周数范围	第一周的判断规则
0	星期日	0-53	元旦后的第一个星期日所在周
1	星期一	0-53	元旦后有四天或者更多天所在周

模式	一周的第一天	周数范围	第一周的判断规则
2	星期日	1-53	元旦后的第一个星期日所在周
3	星期一	1-53	元旦后有四天或者更多天所在周
4	星期日	0-53	元旦后有四天或者更多天所在周
5	星期一	0-53	元旦后的第一个星期一所在周
6	星期日	1-53	元旦后有四天或者更多天所在周
7	星期一	1-53	元旦后的第一个星期一所在周

示例：

```
select week('2018-01-01');
 week
------
    0
(1 row)

select week('2018-01-01', 0);
 week
------
    0
(1 row)

select week('2020-12-31', 1);
 week
------
   53
(1 row)

select week('2020-12-31', 5);
 week
------
   52
(1 row)
```

weekday(date)

描述：返回给定日期date对应的星期索引，星期一作为一周的开始日。

取值范围：0~6

返回值类型：integer

示例：

```
select weekday('2020-11-06');
 weekday
---------
       4
(1 row)
```

weekofyear(date)

描述：返回给定日期date所在周在本年中对应的周数，取值范围为[1, 53]，等价于week(date, 3)。

返回值类型：integer

示例：

```
select weekofyear('2020-12-30');
 weekofyear
------------
         53
(1 row)
```

year(date)

描述：获取时间日期date所处的年份

返回值类型：integer

示例：

```
select year('2020-11-13');
 year
------
 2020
(1 row)
```

yearweek(date[, mode])

描述：返回给定日期date在本年中对应的年份和周数，周数范围为[1, 53]。

返回值类型：integer

示例：

```
select yearweek('2019-12-31');
 yearweek
----------
   201952
(1 row)

select yearweek('2019-1-1');
 yearweek
----------
   201852
(1 row)
```

8.3.6 数组函数和操作符

8.3.6.1 数组操作符

数组比较是使用默认的B-tree比较函数对所有元素逐一进行比较的。多维数组的元素按照行顺序进行访问。如果两个数组的内容相同但维数不等，决定排序顺序的首要因素是维数。

=

描述：两个数组是否相等。

示例：

```
SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT;
 result
--------
 t
(1 row)
```


<>

描述：两个数组是否不相等。

示例：

```
SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<

描述：一个数组是否小于另一个数组。

示例：

```
SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

>

描述：一个数组是否大于另一个数组。

示例：

```
SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<=

描述：一个数组是否小于或等于另一个数组。

示例：

```
SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;
 result
--------
 t
(1 row)
```

>=

描述：一个数组是否大于或等于另一个数组。

示例：

```
SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;
 result
--------
 t
(1 row)
```

@>

描述：一个数组是否包含另一个数组。

示例：

```
SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：一个数组是否被包含于另一个数组。

示例：

```
SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;
 result
--------
 t
(1 row)
```

&&

描述：一个数组是否和另一个数组重叠（有共同元素）。

示例：

```
SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;
 result
--------
 t
(1 row)
```

||

- 描述：数组与数组进行连接。

  示例：

  ```
  SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;
      result
  ---------------
   {1,2,3,4,5,6}
  (1 row)
  SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;
            result
  ---------------------------
   {{1,2,3},{4,5,6},{7,8,9}}
  (1 row)
  ```

- 描述：元素与数组进行连接。

  示例：

  ```
  SELECT 3 || ARRAY[4,5,6] AS RESULT;
    result
  -----------
   {3,4,5,6}
  (1 row)
  ```

- 描述：数组与元素进行连接。

  示例：

  ```
  SELECT ARRAY[4,5,6] || 7 AS RESULT;
    result
  -----------
   {4,5,6,7}
  (1 row)
  ```

8.3.6.2 数组函数

array_append(anyarray, anyelement)

描述：向数组末尾添加元素，只支持一维数组。

返回类型：anyarray

示例：

```
SELECT array_append(ARRAY[1,2], 3) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

array_prepend(anyelement, anyarray)

描述：向数组开头添加元素，只支持一维数组。

返回类型：anyarray

示例：

```
SELECT array_prepend(1, ARRAY[2,3]) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

array_cat(anyarray, anyarray)

描述：连接两个数组，支持多维数组。

返回类型：anyarray

示例：

```
SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;
   result
-------------
 {1,2,3,4,5}
(1 row)

SELECT array_cat(ARRAY[[1,2],[4,5]], ARRAY[6,7]) AS RESULT;
       result
---------------------
 {{1,2},{4,5},{6,7}}
(1 row)
```

array_ndims(anyarray)

描述：返回数组的维数。

返回类型：int

示例：

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
 result
--------
      2
(1 row)
```

array_dims(anyarray)

描述：返回数组维数的文本表示。

返回类型：text

示例：

```
SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
   result
------------
 [1:2][1:3]
(1 row)
```

array_length(anyarray, int)

描述：返回数组维度的长度。

返回类型：int

示例：

```
SELECT array_length(array[1,2,3], 1) AS RESULT;
 result
--------
      3
(1 row)
```

array_lower(anyarray, int)

描述：返回数组维数的下界。

返回类型：int

示例：

```
SELECT array_lower('[0:2]={1,2,3}'::int[], 1) AS RESULT;
 result
--------
      0
(1 row)
```

array_upper(anyarray, int)

描述：返回数组维数的上界。

返回类型：int

示例：

```
SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;
 result
--------
      4
(1 row)
```

array_to_string(anyarray, text [, text])

描述：使用第一个text作为数组的新分隔符，使用第二个text替换数组值为null的值。

返回类型：text

示例：

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') AS RESULT;
 result
-----------
 1,2,3,*,5
(1 row)
```

说明

在string_to_array中，如果省略null字符串参数或为NULL，将字符串中没有输入内容的子串替换为NULL。

在array_to_string中，如果省略null字符串参数或为NULL，运算中将跳过在数组中的任何null元素，并且不会在输出字符串中出现。

string_to_array(text, text [, text])

描述：使用第二个text指定分隔符，使用第三个可选的text作为NULL值替换模板，如果分隔后的子串与第三个可选的text完全匹配，则将其替换为NULL。

返回类型：text[]

示例：

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
   result
--------------
 {xx,NULL,zz}
(1 row)
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'y') AS RESULT;
  result
------------
 {xx,yy,zz}
(1 row)
```

说明

在string_to_array中，如果分隔符参数是NULL，输入字符串中的每个字符将在结果数组中变成一个独立的元素。如果分隔符是一个空白字符串，则整个输入的字符串将变为一个元素的数组。否则输入字符串将在每个分隔字符串处分开。

unnest(anyarray)

描述：扩大一个数组为一组行。

返回类型：setof anyelement

示例：

```
SELECT unnest(ARRAY[1,2]) AS RESULT;
 result
--------
      1
      2
(2 rows)
```

unnest函数配合string_to_array数组使用。数组转列，先将字符串按逗号分割成数组，然后再把数组转成列：

```
SELECT unnest(string_to_array('a,b,c,d',',')) AS RESULT;
 result
--------
 a
 b
 c
 d
(4 rows)
```

interval(N, N1, N2, N3 ...)

描述：从输入的整数数组中，查找返回最后一个小于等于目标参数n的数组索引。如果n为NULL，返回-1。interval函数不支持interval(N, N1)的场景。该函数仅8.2.0及以上集群版本支持。

返回类型：int

示例：

```
SELECT INTERVAL(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11) AS RESULT;
 result
--------
     11
(1 row)
```

split(string, delim)

描述：将字符串string按分隔符delimiter进行分隔，并返回数组。该函数仅8.2.0及以上集群版本支持。

返回类型：text[]

示例：

```
SELECT SPLIT('a-b-c-d-e', '-') AS RESULT;
   result
-------------
 {a,b,c,d,e}
(1 row)
SELECT SPLIT('a-b-c-d-e', '-')[4] AS RESULT;
 result
--------
 d
(1 row)
```

8.3.7 逻辑操作符

常用的逻辑操作符有AND、OR和NOT，其运算结果有三个值，分别为TRUE、FALSE和NULL，其中NULL代表未知。运算优先级顺序为：NOT>AND>OR。

运算规则请参见**表8-21**，表中的a和b代表逻辑表达式。

表 8-21 运算规则表

a	b	a AND b的结果	a OR b的结果	NOT a的结果
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

说明

操作符AND和OR具有交换性，即交换左右两个操作数，不影响其结果。

8.3.8 比较操作符

比较操作符可用于所有相关的数据类型，并返回布尔类型数值。

所有比较操作符都是双目操作符，被比较的两个数据类型必须是相同的数据类型或者是可以进行隐式转换的类型。例如1<2<3这样的表达式为非法的，因为布尔值和3之间不能做比较。

DataArtsFabric SQL提供的比较操作符请参见**表8-22**。

表 8-22 比较操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于
<> 或 !=	不等于

8.3.9 模式匹配操作符

数据库提供了三种实现模式匹配的方法：SQL LIKE操作符、SIMILAR TO操作符和POSIX-风格的正则表达式。除了这些基本的操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

LIKE

判断字符串是否能匹配上LIKE后的模式字符串。如果字符串与提供的模式匹配，则LIKE表达式返回为真（NOT LIKE表达式返回假），否则返回为假（NOT LIKE表达式返回真）。

- 匹配规则
  a. 此操作符只有在它的模式匹配整个串的时候才能成功。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。
  b. 下划线 （_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。
  c. 要匹配文本里的下划线（_）或者百分号（%），在提供的模式里相应字符必须前导逃逸字符。逃逸字符的作用是禁用元字符的特殊含义，缺省的逃逸字符是反斜线，也可以用ESCAPE子句指定一个不同的逃逸字符。
  d. 要匹配逃逸字符本身，需写两个逃逸字符。例如要写一个包含反斜线的模式常量，那就要在SQL语句里写两个反斜线。

📖 **说明**

参数standard_conforming_strings设置为off时，在文串常量中写的任何反斜线都需要被双写。因此写一个匹配单个反斜线的模式实际上要在语句里写四个反斜线。可通过用ESCAPE选择一个不同的逃逸字符来避免这种情况，这样反斜线就不再是LIKE的特殊字符了。但仍然是字符文本分析器的特殊字符，所以还是需要两个反斜线。也可通过写ESCAPE ''的方式不选择逃逸字符，这样可以有效地禁用逃逸机制，但是没有办法关闭下划线和百分号在模式中的特殊含义。

e. 关键字ILIKE可以用于替换LIKE，区别是LIKE大小写敏感，ILIKE大小写不敏感。

f. 操作符~~等效于LIKE，操作符~~*等效于ILIKE。

- 示例

```
SELECT 'abc' LIKE 'abc' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' LIKE 'a%' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' LIKE '_b_' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' LIKE 'c' AS RESULT;
 result
-----------
 f
(1 row)
```

SIMILAR TO

SIMILAR TO操作符根据自己的模式判断是否匹配给定串而返回真或者假。它和LIKE非常类似，只不过它使用SQL标准定义的正则表达式理解模式。

- 匹配规则

a. 和LIKE一样，SIMILAR TO操作符只有在它的模式匹配整个串的时候才返回真。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

b. 下划线（_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。

c. SIMILAR TO也支持下面这些从POSIX正则表达式借用的模式匹配元字符。

表 8-23 模式匹配元字符

元字符	含义
|	表示选择（两个候选之一）。
*	表示重复前面的项零次或更多次。
+	表示重复前面的项一次或更多次。
?	表示重复前面的项零次或一次。
{m}	表示重复前面的项刚好m次。

元字符	含义
{m,}	表示重复前面的项m次或更多次。
{m,n}	表示重复前面的项至少m次并且不超过n次。
()	把多个项组合成一个逻辑项。
[...]	声明一个字符类，就像POSIX正则表达式一样。

d. 前导逃逸字符可以禁止所有这些元字符的特殊含义。逃逸字符的使用规则和LIKE一样。

- 注意事项

  如果SIMILAR TO正则表达式重复匹配字符数量非常庞大，由于受递归大小限制，执行语句会失败并报错invalid regular expression: regular expression is too complex，可尝试调大GUC参数**max_stack_depth**。

- 正则表达式函数

  支持使用函数**substring(string from pattern for escape)**截取匹配SQL正则表达式的子字符串。

- 示例

```
SELECT 'abc' SIMILAR TO 'abc' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' SIMILAR TO 'a' AS RESULT;
 result
-----------
 f
(1 row)
SELECT 'abc' SIMILAR TO '%(b|d)%' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' SIMILAR TO '(b|c)%'  AS RESULT;
 result
-----------
 f
(1 row)
```

POSIX 正则表达式

正则表达式是一个字符序列，它是定义一个串集合 （一个正则集 ）的缩写。 如果一个串是正则表达式描述的正则集中的一员时， 那么就说这个串匹配该正则表达式。POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的含义。**表8-24**列出了所有可用于POSIX正则表达式模式匹配的操作符。

表 8-24 正则表达式匹配操作符

操作符	描述	例子
~	匹配正则表达式，大小写敏感	'thomas' ~ '.*thomas.*'

操作符	描述	例子
~*	匹配正则表达式，大小写不敏感	'thomas' ~* '.*Thomas.*'
！~	不匹配正则表达式，大小写敏感	'thomas' !~ '.*Thomas.*'
！~*	不匹配正则表达式，大小写不敏感	'thomas' !~* '.*vadim.*'

- 匹配规则
  a. 与LIKE不同，正则表达式允许匹配串里的任何位置，除非该正则表达式显式地挂接在串的开头或者结尾。
  b. 除了上文提到的元字符外， POSIX正则表达式还支持下表的模式匹配元字符。

表 8-25 模式匹配元字符

元字符	含义
^	表示串开头的匹配。
$	表示串末尾的匹配。
.	匹配任意单个字符。

- 正则表达式函数

  POSIX正则表达式支持下面函数。
  – **substring(string from pattern)**函数提供了抽取一个匹配POSIX正则表达式模式的子串的方法。
  – **regexp_replace(string, pattern, replacement [,flags])**函数提供了将匹配POSIX正则表达式模式的子串替换为新文本的功能。
  – **regexp_matches(string text, pattern text [, flags text])**函数返回一个文本数组，该数组由匹配一个POSIX正则表达式模式得到的所有被捕获子串构成。
  – **regexp_split_to_table(string text, pattern text [, flags text])**函数把一个POSIX正则表达式模式当作一个定界符来分离一个串。
  – **regexp_split_to_array(string text, pattern text [, flags text])**和regexp_split_to_table类似，是一个正则表达式分离函数，不过它的结果以一个text数组的形式返回。

  说明

  正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者由regexp_matches实现，但是通常前者是实际中最常用的行为。

- 示例

```
SELECT 'abc' ~ 'Abc' AS RESULT;
result
```

```
--------
 f
(1 row)
SELECT 'abc' ~* 'Abc' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc' !~ 'Abc' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc'!~* 'Abc' AS RESULT;
 result
--------
 f
(1 row)
SELECT 'abc' ~ '^a' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc' ~ '(b|d)'AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc' ~ '^(b|c)'AS RESULT;
 result
--------
 f
(1 row)
```

虽然大部分的正则表达式搜索都能很快地执行，但是正则表达式仍可能被人为地控制，通过任意长的时间和任意量的内存进行处理。不建议从非安全模式来源接受正则表达式搜索模式，如果必须这样做，建议加上语句超时限制。使用SIMILAR TO模式的搜索具有同样的安全性危险， 因为SIMILAR TO提供了很多和POSIX-风格正则表达式相同的能力。LIKE搜索比其他两种选项简单得多，因此在接受非安全模式来源搜索时要更安全些。

8.3.10 聚集函数

sum(expression)

描述：所有输入行的expression总和。

返回类型：

通常情况下输入数据类型和输出数据类型是相同的，但以下情况会发生类型转换：

- 对于SMALLINT或INT输入，输出类型为BIGINT。
- 对于BIGINT输入，输出类型为NUMBER 。
- 对于浮点数输入，输出类型为DOUBLE PRECISION。

示例：

```
SELECT SUM(ss_ext_tax) FROM tpcds.STORE_SALES;
  sum
--------------
 213267594.69
(1 row)
```

max(expression)

描述：所有输入行中expression的最大值。

参数类型：任意数组、数值、字符串、日期/时间类型。

返回类型：与参数数据类型相同。

示例：

```
SELECT MAX(inv_quantity_on_hand) FROM tpcds.inventory;
   max
---------
 1000000
(1 row)
```

min(expression)

描述：所有输入行中expression的最小值。

参数类型：任意数组、数值、字符串、日期/时间类型。

返回类型：与参数数据类型相同。

示例：

```
SELECT MIN(inv_quantity_on_hand) FROM tpcds.inventory;
 min
-----
   0
(1 row)
```

avg(expression)

描述：所有输入值的均值（算术平均）。

返回类型：

- 对于任何整数类型输入，结果都是NUMBER类型。
- 对于任何浮点输入，结果都是DOUBLE PRECISION类型。
- 其他，和输入数据类型相同。

示例：

```
SELECT AVG(inv_quantity_on_hand) FROM tpcds.inventory;
         avg
----------------------
 500.0387129084044604
(1 row)
```

median(expression)

描述：所有输入值的中位数值。当前只支持数值类型和interval类型。其中空值不参与计算。

返回类型：

- 对于任何整型数据输入，结果都是NUMERIC类型。否则与输入数据类型相同。
- Teradata兼容模式下，如果输入为整型，则返回的数据精度只有整数位。

示例：

```
SELECT MEDIAN(inv_quantity_on_hand) FROM tpcds.inventory;
 median
--------
    500
(1 row)
```

percentile_cont(const) within group(order by expression)

描述：返回一个对应于目标列排序中指定分位数的值，如有必要就在相邻的输入项之间插入值。其中空值不参与计算。

输入：const为在0-1之间的数值，expression当前只支持数值类型和interval类型。

返回类型：

- 对于任何整型数据输入，结果都是NUMERIC类型。否则与输入数据类型相同。
- Teradata兼容模式下，如果输入为整型，则返回的数据精度只有整数位。

示例：

```
SELECT percentile_cont(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----------------
2.2
(1 row)
SELECT percentile_cont(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----------------
3.8
(1 row)
```

percentile_disc(const) within group(order by expression)

描述：返回第一个在排序中位置等于或者超过指定分数的输入值。

输入：const为在0-1之间的数值，expression当前只支持数值类型和interval类型。其中空值不参与计算。

返回类型：对于任何整型数据输入，结果都是NUMERIC类型。否则，与输入数据类型相同。

示例：

```
SELECT percentile_disc(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----------------
2
(1 row)
SELECT percentile_disc(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----------------
4
(1 row)
```

count(expression)

描述：返回表中满足expression不为NULL的行数。

返回类型：BIGINT

示例：

```
SELECT COUNT(inv_quantity_on_hand) FROM tpcds.inventory;
  count
```

```
----------
 11158087
(1 row)
```

count(*)

描述：返回表中的记录行数。

返回类型：BIGINT

示例：

```
SELECT COUNT(*) FROM tpcds.inventory;
  count
----------
 11745000
(1 row)
```

array_agg(expression)

描述：将所有输入值（包括空）连接成一个数组。函数入参不支持数组形式。

返回类型：参数类型的数组。

示例：

创建表employeeinfo，并插入数据：

```
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)
```

查询部门编号为30的所有员工姓名：

```
SELECT array_agg(ename) FROM employeeinfo where deptno = 30;
 array_agg
------------
 {JACK,MAX}
(1 row)
```

查询属于同一个部门的所有员工：

```
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)
```

查询所有的部门编号且去重：

```
SELECT array_agg(distinct deptno) FROM employeeinfo group by deptno;
 array_agg
-----------
 {20}
 {30}
(2 rows)
```

查询所有的部门编号去重后按降序排列：

```
SELECT array_agg(distinct deptno order by deptno desc) FROM employeeinfo;
 array_agg
-----------
 {30,20}
(1 row)
```

string_agg(expression, delimiter)

描述：将输入值连接成为一个字符串，用分隔符分开。

返回类型：和参数数据类型相同。

示例：

基于创建的表employeeinfo，查询属于同一个部门的所有员工：

```
SELECT deptno, string_agg(ename,',') FROM employeeinfo group by deptno;
 deptno | string_agg
--------+------------
     30 | JACK,MAX
     20 | TOM
(2 rows)
```

查询工号小于7156的员工：

```
SELECT string_agg(ename,',') FROM employeeinfo where empno < 7156;
 string_agg
------------
 TOM,JACK
(1 row)
```

listagg(expression [, delimiter]) WITHIN GROUP(ORDER BY order-list)

描述：将聚集列数据按WITHIN GROUP指定的排序方式排列，并用delimiter指定的分隔符拼接成一个字符串。

- expression：必选。指定聚集列名或基于列的有效表达式，不支持DISTINCT关键字和VARIADIC参数。
- delimiter：可选。指定分隔符，可以是字符串常数或基于分组列的确定性表达式，缺省时表示分隔符为空。
- order-list：必选。指定分组内的排序方式。

返回类型：text

说明

listagg是兼容Oracle 11g2的列转行聚集函数，可以指定OVER子句用作窗口函数。为了避免与函数本身WITHIN GROUP子句的ORDER BY造成二义性，listagg用作窗口函数时，OVER子句不支持ORDER BY的窗口排序或窗口框架。

示例：

聚集列是文本字符集类型：

```
SELECT deptno, listagg(ename, ',') WITHIN GROUP(ORDER BY ename) AS employees FROM emp GROUP
BY deptno;
 deptno |              employees
--------+--------------------------------------
     10 | CLARK,KING,MILLER
     20 | ADAMS,FORD,JONES,SCOTT,SMITH
     30 | ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
(3 rows)
```

聚集列是整型：

```
SELECT deptno, listagg(mgrno, ',') WITHIN GROUP(ORDER BY mgrno NULLS FIRST) AS mgrnos FROM emp
GROUP BY deptno;
 deptno |             mgrnos
--------+-------------------------------
     10 | 7782,7839
     20 | 7566,7566,7788,7839,7902
     30 | 7698,7698,7698,7698,7698,7839
(3 rows)
```

聚集列是浮点类型：

```
SELECT job, listagg(bonus, '($); ') WITHIN GROUP(ORDER BY bonus DESC) || '($)' AS bonus FROM emp
GROUP BY job;
    job     |                      bonus
------------+-------------------------------------------------
 CLERK      | 10234.21($); 2000.80($); 1100.00($); 1000.22($)
 PRESIDENT  | 23011.88($)
 ANALYST    | 2002.12($); 1001.01($)
 MANAGER    | 10000.01($); 2399.50($); 999.10($)
 SALESMAN   | 1000.01($); 899.00($); 99.99($); 9.00($)
(5 rows)
```

聚集列是时间类型：

```
SELECT deptno, listagg(hiredate, ', ') WITHIN GROUP(ORDER BY hiredate DESC) AS hiredates FROM emp
GROUP BY deptno;
 deptno |                                                        hiredates
--------
+-----------------------------------------------------------------------------------------------------------------------------
--
     10 | 1982-01-23 00:00:00, 1981-11-17 00:00:00, 1981-06-09 00:00:00
     20 | 2001-04-02 00:00:00, 1999-12-17 00:00:00, 1987-05-23 00:00:00, 1987-04-19 00:00:00, 1981-12-03
00:00:00
     30 | 2015-02-20 00:00:00, 2010-02-22 00:00:00, 1997-09-28 00:00:00, 1981-12-03 00:00:00, 1981-09-08
00:00:00, 1981-05-01 00:00:00
(3 rows)
```

聚集列是时间间隔类型：

```
SELECT deptno, listagg(vacationTime, '; ') WITHIN GROUP(ORDER BY vacationTime DESC) AS vacationTime
FROM emp GROUP BY deptno;
 deptno |                                     vacationtime
--------+------------------------------------------------------------------------------------
     10 | 1 year 30 days; 40 days; 10 days
     20 | 70 days; 36 days; 9 days; 5 days
     30 | 1 year 1 mon; 2 mons 10 days; 30 days; 12 days 12:00:00; 4 days 06:00:00; 24:00:00
(3 rows)
```

分隔符缺省时，默认为空：

```
SELECT deptno, listagg(job) WITHIN GROUP(ORDER BY job) AS jobs FROM emp GROUP BY deptno;
 deptno |                     jobs
--------+----------------------------------------------
     10 | CLERKMANAGERPRESIDENT
     20 | ANALYSTANALYSTCLERKCLERKMANAGER
     30 | CLERKMANAGERSALESMANSALESMANSALESMANSALESMAN
(3 rows)
```

listagg作为窗口函数时，OVER子句不支持ORDER BY的窗口排序，listagg列为对应分组的有序聚集：

```
SELECT deptno, mgrno, bonus, listagg(ename,'; ') WITHIN GROUP(ORDER BY hiredate) OVER(PARTITION BY deptno) AS employees FROM emp;
 deptno | mgrno |  bonus   |                employees
--------+-------+----------+-------------------------------------------
     10 |  7839 | 10000.01 | CLARK; KING; MILLER
     10 |       | 23011.88 | CLARK; KING; MILLER
     10 |  7782 | 10234.21 | CLARK; KING; MILLER
     20 |  7566 |  2002.12 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7566 |  1001.01 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7788 |  1100.00 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7902 |  2000.80 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7839 |   999.10 | FORD; SCOTT; ADAMS; SMITH; JONES
     30 |  7839 |  2399.50 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |     9.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.22 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |    99.99 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.01 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |   899.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
(14 rows)
```

group_concat(expression [ORDER BY {col_name | expr} [ASC | DESC]] [SEPARATOR str_val])

描述：将列数据使用指定的str_val分隔符，按照ORDER BY子句指定的排序方式拼接成字符串，ORDER BY子句必须指定排序方式，不支持ORDER BY 1的写法。

- expression：必选，指定列名或基于列的有效表达式，不支持DISTINCT关键字和VARIADIC参数。
- str_val：可选，指定的分隔符，可以是字符串常数或基于分组列的确定性表达式。缺省时表示分隔符为逗号。

返回类型：text

说明

group_concat函数仅8.1.2及以上版本支持。

示例：

默认分隔符为逗号：

```
SELECT group_concat(sname) FROM group_concat_test;
              group_concat
-----------------------------------------
 ADAMS,FORD,JONES,KING,MILLER,SCOTT,SMITH
(1 row)
```

group_concat函数支持自定义分隔符：

```
SELECT group_concat(sname separator ';') from group_concat_test;
              group_concat
-----------------------------------------
 ADAMS;FORD;JONES;KING;MILLER;SCOTT;SMITH
(1 row)
```

group_concat函数支持ORDER BY子句，将列数据进行有序拼接：

```
SELECT group_concat(sname order by snumber separator ';') FROM group_concat_test;
              group_concat
-----------------------------------------
 MILLER;FORD;SCOTT;SMITH;KING;JONES;ADAMS
(1 row)
```

covar_pop(Y, X)

描述：总体协方差。

返回类型：double precision

示例：

```
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)

描述：样本协方差。

返回类型：double precision

示例：

```
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)

描述：总体标准差。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
   stddev_pop
------------------
 289.224294957556
(1 row)
```

stddev_samp(expression)

描述：样本标准差。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
  stddev_samp
------------------
 289.224359757315
(1 row)
```

var_pop(expression)

描述：总体方差（总体标准差的平方）。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：

```
SELECT VAR_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
     var_pop
--------------------
 83650.692793695475
(1 row)
```

var_samp(expression)

描述：样本方差（样本标准差的平方）。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：

```
SELECT VAR_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
     var_samp
--------------------
 83650.730277028768
(1 row)
```

bit_and(expression)

描述：所有非NULL输入值的按位与（AND），如果全部输入值皆为NULL，那么结果也为NULL 。

返回类型：和参数数据类型相同。

示例：

```
SELECT BIT_AND(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_and
---------
       0
(1 row)
```

bit_or(expression)

描述：所有非NULL输入值的按位或（OR），如果全部输入值皆为NULL，那么结果也为NULL。

返回类型：和参数数据类型相同

示例：

```
SELECT BIT_OR(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_or
--------
   1023
(1 row)
```

bool_and(expression)

描述：如果所有输入值都是真，则为真，否则为假。

返回类型：bool

示例：

```
SELECT bool_and(100 <2500);
 bool_and
```

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

bool_or(expression)

描述：如果所有输入值只要有一个为真，则为真，否则为假。

返回类型：bool

示例：

```
SELECT bool_or(100 <2500);
 bool_or
----------
 t
(1 row)
```

corr(Y, X)

描述：相关系数。

返回类型：double precision

示例：

```
SELECT CORR(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
       corr
--------------------
 0.0381383624904186
(1 row)
```

every(expression)

描述：等效于bool_and。

返回类型：bool

示例：

```
SELECT every(100 <2500);
 every
-------
 t
(1 row)
```

rank(expression)

描述：根据expression对不同组内的元组进行跳跃排序。

返回类型：bigint

示例：

```
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)

描述：自变量的平均值 (sum(X)/N)。

返回类型：double precision

示例：

```
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)

描述：因变量的平均值 (sum(Y)/N)。

返回类型：double precision

示例：

```
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)

描述：两个表达式都不为NULL的输入行数。

返回类型：bigint

示例：

```
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)

描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程，然后返回该直线的Y轴截距。

返回类型：double precision

示例：

```
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)

描述：相关系数的平方。

返回类型：double precision

示例：

```
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)

描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程， 然后返回该直线的斜率。

返回类型：double precision

示例：

```
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)

描述：sum(X^2) - sum(X)^2/N（自变量的“平方和”）。

返回类型：double precision

示例：

```
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)

描述：sum(X*Y) - sum(X) * sum(Y)/N （自变量和因变量的“乘方积”）。

返回类型：double precision

示例：

```
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)

描述：sum(Y^2) - sum(Y)^2/N（因变量的“平方和”）。

返回类型：double precision

示例：

```
SELECT REGR_SYY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    regr_syy
-----------------
 2189417.6547314
(1 row)
```

stddev(expression)

描述：stddev_samp的别名。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
      stddev
------------------
 289.224359757315
(1 row)
```

variance(expexpression,ression)

描述：var_samp的别名。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：

```
SELECT VARIANCE(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
      variance
--------------------
 83650.730277028768
(1 row)
```

checksum(expression)

描述：返回所有输入值的CHECKSUM值。使用该函数可以用来验证DataArtsFabric SQL数据库（不支持DataArtsFabric SQL之外的其他数据库）的备份恢复或者数据迁移操作前后表中的数据是否相同。在备份恢复或者数据迁移操作前后都需要用户通过手工执行SQL命令的方式获取执行结果，通过对比获取的执行结果判断操作前后表中的数据是否相同。

说明

- 对于大表，CHECKSUM函数可能会需要很长时间。
- 如果某两表的CHECKSUM值不同，则表明两表的内容是不同的。由于CHECKSUM函数中使用散列函数不能保证无冲突，因此两个不同内容的表可能会得到相同的CHECKSUM值，存在这种情况的可能性较小。对于列进行的CHECKSUM也存在相同的情况。
- 对于时间类型timestamp, timestamptz和smalldatetime，计算CHECKSUM值时请确保时区设置一致。

- 如果计算某列的CHECKSUM值，且该列类型可以默认转为TEXT类型，则expression为列名。
- 如果计算某列的CHECKSUM值，且该列类型不能默认转为TEXT类型，则expression为列名::TEXT。
- 如果计算所有列的CHECKSUM值，则expression为表名::TEXT。

可以默认转换为TEXT类型的类型包括：char, name, int8, int2, int1, int4, raw, pg_node_tree, float4, float8, bpchar, varchar, nvarchar2, date, timestamp, timestamptz, numeric, smalldatetime，其他类型需要强制转换为TEXT。

返回类型：numeric

示例：

表中可以默认转为TEXT类型的某列的CHECKSUM值：

```
SELECT CHECKSUM(inv_quantity_on_hand) FROM tpcds.inventory;
     checksum
-------------------
 24417258945265247
(1 row)
```

表中不能默认转为TEXT类型的某列的CHECKSUM值（注意此时CHECKSUM参数是列名::TEXT）：

```
SELECT CHECKSUM(inv_quantity_on_hand::TEXT) FROM tpcds.inventory;
     checksum
-------------------
 24417258945265247
(1 row)
```

表中所有列的CHECKSUM值。注意此时CHECKSUM参数是表名::TEXT，且表名前不加Schema：

```
SELECT CHECKSUM(inventory::TEXT) FROM tpcds.inventory;
     checksum
-------------------
 25223696246875800
(1 row)
```

approx_count_distinct(col_name)

描述：使用HyperLogLog++ (HLL++) 算法进行基数（某一列去重后的行数）的估算。该函数仅8.3.0及以上集群版本支持。

入参说明：col_name指需要估算基数的列。

📖 **说明**

可通过GUC参数approx_count_distinct_precision调整误差率。

- 参数取值范围为[10,20]，默认值为17，理论误差率为千分之三。
- 该参数表示HyperLogLog++ (HLL++)算法中分桶个数，参数越大时，分桶数则越大，理论误差率则越小。
- 该参数取值越大，相应的计算时间和内存资源开销越大，但依然远小于精确的count distinct语句对应的开销。推荐在估算基数较大时使用该函数替换count distinct。

示例：

```
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 APPROX_COUNT_DISTINCT(empno) from employeeinfo;
 approx_count_distinct
-----------------------
                     3
(1 row)

SELECT COUNT(DISTINCT empno) FROM employeeinfo GROUP BY ename;
 count
-------
     1
     1
     1
(3 rows)
```

UNIQ(col_name)

描述：计算非重复值个数的聚合函数，与COUNT DISTINCT类似，即计算某一列去重后的行数，结果返回一个去重值。该函数仅8.3.0及以上集群版本支持。

入参说明：col_name指需要计算去重后行数的列。支持SMALLINT、INTEGER、BIGINT、REAL、DOUBLE PRECISION、TEXT、VARCHAR、TIMESTAMP、TIMESTAMPTZ、DATE、TIMETZ、UUID类型。

📖 **说明**

- 使用UNIQ函数时，SQL需要包含GROUP BY，为了取得更好性能所选GROUP BY字段分布需要尽量均匀。
- 建议当SQL中需要去重的列大于等于3列时，使用UNIQ函数去重，以取得比COUNT DISTINCT更好的效果。
- 一般情况下UNIQ函数内存消耗低于COUNT DISTINCT，如果使用COUNT DISTINCT时遇到内存超限，可以使用UNIQ函数进行替换。

示例：

```
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;
 uniq
------
```



```
 1
 1
 1
(3 rows)
```

8.3.11 窗口函数

普通的聚集函数只能用来计算一行内的结果，或者把所有行聚集成一行结果。而窗口函数可以跨行计算，并且把结果填到每一行中。

通过查询筛选出的行的某些部分，窗口调用函数实现了类似于聚集函数的功能，所以聚集函数也可以作为窗口函数使用。 窗口函数可以扫描所有的行，并同时将原始数据和聚集分析结果同时显示出来。

注意事项

- 列存表目前只支持窗口函数rank(expression)和row_number(expression)，以及聚集函数的sum，count，avg，min和max，而行存表没有限制。
- 单个查询中可以包含一个或多个窗口函数表达式。
- 窗口函数仅能出现在输出列中。如果需要使用窗口函数的值进行条件过滤，需要将窗口函数嵌套在子查询中，在外层使用窗口函数表达式的别名进行条件过滤。例如：
  ```
  SELECT classid, id, score FROM(SELECT *, avg(score) OVER(PARTITION BY classid) as avg_score FROM score) WHERE score >= avg_score;
  ```
- 窗口函数所在查询块中支持使用GROUP BY表达式进行分组去重，但要求窗口函数中的PARTITION BY子句中必须是GROUP BY表达式的子集，以保证窗口函数在GROUP BY列去重后的结果上进行窗口运算，同时ORDER BY子句的表达式也需要是GROUP BY表达式的子集，或聚集运算的聚集函数。例如：
  ```
  SELECT classid,rank() OVER(PARTITION BY classid ORDER BY sum(score)) as avg_score FROM score GROUP BY classid, id;
  ```

语法格式

窗口函数需要特殊的关键字OVER语句来指定窗口触发窗口函数。OVER语句用于对数据进行分组，并对组内元素进行排序。窗口函数用于给组内的值生成序号：

```
function_name ([expression [, expression ... ]]) OVER ( window_definition )
function_name ([expression [, expression ... ]]) OVER window_name
function_name ( * ) OVER ( window_definition )
function_name ( * ) OVER window_name
```

其中window_definition子句option为：

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

PARTITION BY选项指定了将具有相同PARTITION BY表达式值的行分为一组。

ORDER BY选项用于控制窗口函数处理行的顺序。ORDER BY后面必须跟字段名，如果ORDER BY后面跟数字，该数字会被按照常量处理，对目标列没有起到排序的作用。

frame_clause子句option为：

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

当需要指定一个窗口对分组内所有行结果进行计算时，需要指定窗口区间开始的行和结束的行。窗口区间支持RANGE、ROWS两种模式，ROWS以物理单位（行）指定窗口，RANGE将窗口指定为逻辑偏移量。

RANGE、ROWS中可以使用BETWEEN frame_start AND frame_end指定边界可取值。如果仅指定frame_start，则frame_end默认为CURRENT ROW。

frame_start和frame_end取值为：

- CURRENT ROW，当前行。
- N PRECEDING，当前行向前第n行。
- UNBOUNDED PRECEDING，当前PARTITION的第1行。
- N FOLLOWING，当前行向后第n行。
- UNBOUNDED FOLLOWING，当前PARTITION的最后1行。

需要注意，frame_start不能为UNBOUNDED FOLLOWING，frame_end不能为UNBOUNDED PRECEDING，并且frame_end选项不能比上面取值中出现的frame_start选项早。例如RANGE BETWEEN CURRENT ROW AND N PRECEDING是不被允许的。

LAST_VALUE函数支持IGNORE NULLS语法，该语法返回非NULL窗口中的最后一个值，如果所有值都为NULL，则返回NULL，具体格式为：

```
LAST_VALUE (expression [IGNORE NULLS]) OVER (window_definition)
```

当前IGNORE NULLS仅支持ROWS between CURRENT ROW and UNBOUNDED FOLLOWING和ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT ROW两种窗口区间。

RANK()

描述：RANK函数为各组内值生成跳跃排序序号，其中相同的值具有相同序号，但相同值占用多个编号。

返回值类型：BIGINT

示例：

给定表score(id, classid, score)，每行表示学生id，所在班级id以及考试成绩。

使用RANK函数对学生成绩进行排序：

```
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()

描述：ROW_NUMBER函数为各组内值生成连续排序序号，其中相同的值其序号也不相同。

返回值类型：BIGINT

示例：

```
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()

描述：DENSE_RANK函数为各组内值生成连续排序序号，其中相同的值具有相同序号，相同值只占用一个编号。

返回值类型：BIGINT

示例：

```
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()

描述：PERCENT_RANK函数为各组内对应值生成相对序号，即根据公式 (rank - 1) / (total rows - 1)计算所得的值。其中rank为该值依据RANK函数所生成的对应序号，total rows为该分组内的总元素个数。

返回值类型：DOUBLE PRECISION

示例：

```
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()

描述：CUME_DIST函数为各组内对应值生成累积分布序号。即根据公式（小于等于当前值的数据行数）()/（该分组总行数total rows）()计算所得的相对序号。

返回值类型：DOUBLE PRECISION

示例：

```
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)

描述：NTILE函数根据num_buckets integer将有序的数据集合平均分配到num_buckets所指定数量的桶中，并将桶号分配给每一行。分配时应尽量做到平均分配。

返回值类型：INTEGER

示例：

```
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]])

描述：LAG函数为各组内对应值生成滞后值。即当前值对应的行数往前偏移offset位后所得行的value值作为序号。如果经过偏移后行数不存在，则对应结果取为default值。如果无指定，在默认情况下，offset取为1，default值取为NULL。

返回值类型：与参数数据类型相同。

示例：

```
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]])

描述：LEAD函数为各组内对应值生成提前值。即当前值对应的行数向后偏移offset位后所得行的value值作为序号。如果经过向后偏移后行数超过当前组内的总行数，则对应结果取为default值。如果无指定，在默认情况下，offset取为1，default值取为NULL。

返回值类型：与参数数据类型相同。

示例：

```
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)

描述：FIRST_VALUE函数取各组内的第一个值作为返回结果。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,FIRST_VALUE(id) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | first_value
----+---------+-------+-------------
  1 |       1 |    95 |           1
  2 |       2 |    95 |           1
  5 |       2 |    88 |           1
  3 |       2 |    85 |           1
  4 |       1 |    70 |           1
  6 |       1 |    70 |           1
(6 rows)
```

LAST_VALUE(value any)

描述：LAST_VALUE函数取各组内的最后一个值作为返回结果。

返回值类型：与参数数据类型相同。

示例：

```
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)

描述：NTH_VALUE函数返回该组内的第nth行作为结果。如果该行不存在，则默认返回NULL。

返回值类型：与参数数据类型相同。

示例：

```
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)
```

8.3.12 类型转换函数

cast(x as y)

描述：类型转换函数，将x转换成y指定的类型。

示例：

```
SELECT cast('22-oct-1997' as timestamp);
      timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

cast(x, y)

描述：类型转换函数，将x转换成y指定的类型。该函数仅8.2.0及以上集群版本支持。

示例：

```
SELECT cast('22-oct-1997', timestamp);
      timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

try_cast(x as type)

描述：将x转换成给定的type类型值，如果转换失败且当前类型转换为DataArtsFabric SQL允许的转换，则返回NULL，否则报错。该函数仅8.2.0及以上集群版本支持。

示例：

```
SELECT cast('a' as int4);
      int4
---------------

(1 row)
SELECT cast('22-oct-1997', timestamp);
      timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

hextoraw(string)

描述：将一个十六进制构成的字符串转换为二进制。

返回值类型：raw

示例：

```
SELECT hextoraw('7D');
 hextoraw
----------
 7D
(1 row)
```

numtoday(numeric)

描述：将数字类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT numtoday(2);
 numtoday
----------
 2 days
(1 row)
```

pg_systimestamp()

描述：获取系统时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT pg_systimestamp();
        pg_systimestamp
-------------------------------
 2015-10-14 11:21:28.317367+08
(1 row)
```

rawtohex(string)

描述：将一个二进制构成的字符串转换为十六进制的字符串。

结果为输入字符的ACSII码，以十六进制表示。

返回值类型：varchar

示例：

```
SELECT rawtohex('1234567');
    rawtohex
----------------
 31323334353637
(1 row)
```

to_char (datetime/interval [, fmt])

描述：将一个DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE或者TIMESTAMP WITH LOCAL TIME ZONE类型的DATETIME或者INTERVAL值按照fmt指定的格式转换为VARCHAR类型。

- 可选参数fmt可以为以下几类：日期、时间、星期、季度和世纪。每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：HH、MM、SS、YYYY、MM、DD，可参考**表8-27**。
- 模板可以有修饰词，常用的修饰词是FM，可以用来抑制前导的零或尾随的空白。

返回值类型：varchar

示例：

```
SELECT to_char(current_timestamp,'HH12:MI:SS');
 to_char
----------
```

```
 10:19:26
(1 row)
SELECT to_char(current_timestamp,'FMHH12:FMMI:FMSS');
 to_char
----------
 10:19:46
(1 row)
```

to_char(double precision, text)

描述：将双精度类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(125.8::real, '999D99');
 to_char
---------
  125.80
(1 row)
```

to_char (integer/number[, fmt])

描述：将一个整型或者浮点类型的值转换为指定格式的字符串。

- 可选参数fmt可以为以下几类：十进制字符、“分组”符、正负号和货币符号，每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：9、0、,（千分隔符）、.（小数点），可参考**表8-26**。
- 模板可以有类似FM的修饰词，但FM不抑制由模板0指定而输出的0。
- 要将整型类型的值转换成对应16进制值的字符串，使用模板“x”或“X”。

返回值类型：varchar

示例：

```
SELECT to_char(1485,'9,999');
 to_char
---------
  1,485
(1 row)
SELECT to_char( 1148.5,'9,999.999');
  to_char
------------
  1,148.500
(1 row)
SELECT to_char(148.5,'990999.909');
   to_char
-------------
    0148.500
(1 row)
SELECT to_char(123,'XXX');
 to_char
---------
   7B
(1 row)
```

to_char(interval, text)

描述：将时间间隔类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
 to_char
----------
 15:02:12
(1 row)
```

to_char(int, text)

描述：将整数类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(125, '999');
 to_char
---------
  125
(1 row)
```

to_char(numeric, text)

描述：将数字类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(-125.8, '999D99S');
 to_char
---------
 125.80-
(1 row)
```

to_char (string)

描述：将CHAR、VARCHAR、VARCHAR2、CLOB类型转换为VARCHAR类型。

如使用该函数对CLOB类型进行转换，且待转换CLOB类型的值超出目标类型的范围，则返回错误。

返回值类型：varchar

示例：

```
SELECT to_char('01110');
 to_char
---------
 01110
(1 row)
```

to_char(timestamp, text)

描述：将时间戳类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');
 to_char
----------
 10:55:59
(1 row)
```

to_clob(char/nchar/varchar/nvarchar/varchar2/nvarchar2/text/raw)

描述：将RAW类型或者文本字符集类型CHAR、NCHAR、VARCHAR、VARCHAR2、NVARCHAR2、TEXT转成CLOB类型。

返回值类型：clob

示例：

```
SELECT to_clob('ABCDEF'::RAW(10));
 to_clob
---------
 ABCDEF
(1 row)
SELECT to_clob('hello111'::CHAR(15));
 to_clob
----------
 hello111
(1 row)
SELECT to_clob('gauss123'::NCHAR(10));
 to_clob
----------
 gauss123
(1 row)
SELECT to_clob('gauss234'::VARCHAR(10));
 to_clob
----------
 gauss234
(1 row)
SELECT to_clob('gauss345'::VARCHAR2(10));
 to_clob
----------
 gauss345
(1 row)
SELECT to_clob('gauss456'::NVARCHAR2(10));
 to_clob
----------
 gauss456
(1 row)
SELECT to_clob('World222!'::TEXT);
  to_clob
-----------
 World222!
(1 row)
```

to_date(text)

描述：将文本类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT to_date('2015-08-14');
       to_date
---------------------
 2015-08-14 00:00:00
(1 row)
```

to_date(text, text)

描述：将字符串类型的值转换为指定格式的日期。

返回值类型：timestamp

示例：

```
SELECT to_date('05 Dec 2000', 'DD Mon YYYY');
       to_date
---------------------
 2000-12-05 00:00:00
(1 row)
```

to_date(string, fmt)

描述：将字符串string按fmt指定格式转化为DATE类型的值。fmt格式可参考**表8-27**。

该函数不能直接支持CLOB类型，但是CLOB类型的参数能够通过隐式转换实现。

返回值类型：date

示例：

```
SELECT TO_DATE('05 Dec 2010','DD Mon YYYY');
       to_date
---------------------
 2010-12-05 00:00:00
(1 row)
```

to_number (expr [, fmt])

描述：将expr按指定格式转换为一个NUMBER类型的值。

类型转换格式请参考**表8-26**。

转换十六进制字符串为十进制数字时，最多支持16个字节的十六进制字符串转换为无符号数。

转换十六进制字符串为十进制数字时，格式字符串中不允许出现除'x'或'X'以外的其他字符，否则报错。

返回值类型：number

示例：

```
SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

to_number(text, text)

描述：将字符串类型的值转换为指定格式的数字。

返回值类型：numeric

示例：

```
SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

to_timestamp(double precision)

描述：把UNIX纪元转换成时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT to_timestamp(1284352323);
      to_timestamp
------------------------
 2010-09-13 12:32:03+08
(1 row)
```

to_timestamp(string [,fmt])

描述：将字符串string按fmt指定的格式转换成时间戳类型的值。不指定fmt时，按参数nls_timestamp_format所指定的格式转换。fmt格式可参考**表8-27**。

DataArtsFabric SQL的to_timestamp中，

- 如果输入的年份YYYY=0，系统报错。
- 如果输入的年份YYYY<0，在fmt中指定SYYYY，则正确输出公元前绝对值n的年份。

fmt中出现的字符必须与日期/时间格式化的模式相匹配，否则报错。

返回值类型：timestamp without time zone

示例：

```
SHOW nls_timestamp_format;
    nls_timestamp_format
----------------------------
 DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

SELECT to_timestamp('12-sep-2014');
    to_timestamp
---------------------
 2014-09-12 00:00:00
(1 row)
SELECT to_timestamp('12-Sep-10 14:10:10.123000','DD-Mon-YY HH24:MI:SS.FF');
      to_timestamp
-------------------------
 2010-09-12 14:10:10.123
(1 row)
SELECT to_timestamp('-1','SYYYY');
      to_timestamp
------------------------
 0001-01-01 00:00:00 BC
(1 row)
SELECT to_timestamp('98','RR');
    to_timestamp
---------------------
 1998-01-01 00:00:00
(1 row)
SELECT to_timestamp('01','RR');
    to_timestamp
---------------------
 2001-01-01 00:00:00
(1 row)
```

to_timestamp(text, text)

描述：将字符串类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
   to_timestamp
---------------------
 2000-12-05 00:00:00
(1 row)
```

表8-26显示了可以用于格式化数值的模板模式，适用于函数to_number。

表 8-26 数值格式化的模板模式

模式	描述
9	带有指定数值位数的值
0	带前导零的值
.（句点）	小数点
,（逗号）	分组（千）分隔符
PR	尖括号内负值
S	带符号的数值（使用区域设置）
L	货币符号（使用区域设置）
D	小数点（使用区域设置）
G	分组分隔符（使用区域设置）
MI	在指明的位置的负号（如果数字<0）
PL	在指明的位置的正号（如果数字>0）
SG	在指明的位置的正/负号
RN	罗马数字（输入在1和3999之间）
TH或th	序数后缀
V	移动指定位（小数）

表8-27显示了可以用于格式化日期和时间值的模板，这些模式适用于函数to_date、to_timestamp、to_char和参数nls_timestamp_format。

表 8-27 用于日期/时间格式化的模式

类别	模式	描述
小时	HH	一天的小时数（01-12）
	HH12	一天的小时数（01-12）
	HH24	一天的小时数（00-23）
分钟	MI	分钟（00-59）
秒	SS	秒（00-59）

类别	模式	描述
	FF	微秒（000000-999999）
	SSSSS	午夜后的秒（0-86399）
上、下午	AM或A.M.	上午标识
	PM或P.M.	下午标识
年	Y,YYY	带逗号的年（4和更多位）
	SYYYY	公元前四位年
	YYYY	年（4和更多位）
	YYY	年的后三位
	YY	年的后两位
	Y	年的最后一位
	IYYY	ISO年（4位或更多位）
	IYY	ISO年的最后三位
	IY	ISO年的最后两位
	I	ISO年的最后一位
	RR	年的后两位（可在21世纪存储20世纪的年份） 规则如下： ● 输入的两位年份在00~49之间： 当前年份的后两位在00~49之间，返回值年份的前两位和当前年份的前两位相同； 当前年份的后两位在50~99之间，返回值年份的前两位是当前年份的前两位加1。 ● 输入的两位年份在50~99之间： 当前年份的后两位在00~49之间，返回值年份的前两位是当前年份的前两位减1； 当前年份的后两位在50~99之间，返回值年份的前两位和当前年份的前两位相同。
	RRRR	可接收4位年或两位年。如果是两位，则和RR的返回值相同，如果是四位，则和YYYY相同。
	● BC或B.C. ● AD或A.D.	纪元标识。BC（公元前），AD（公元后）。
月	MONTH	全长大写月份名（空白填充为9字符）
	MON	大写缩写月份名（3字符）
	MM	月份数（01-12）
	RM	罗马数字的月份（I-XII； I=JAN）（大写）

类别	模式	描述
天	DAY	全长大写日期名（空白填充为9字符）
	DY	缩写大写日期名（3字符）
	DDD	一年里的日（001-366）
	DD	一个月里的日（01-31）
	D	一周里的日（1-7；周日是 1）
周	W	一个月里的周数（1-5）（第一周从该月第一天开始）
	WW	一年里的周数（1-53）（第一周从该年的第一天开始）
	IW	ISO一年里的周数（第一个星期四在第一周里）
世纪	CC	世纪（2位）（21 世纪从 2001-01-01 开始）
儒略日	J	儒略日（自公元前 4712 年 1 月 1 日来的天数）
季度	Q	季度

8.3.13 JSON/JSONB 函数和操作符

8.3.13.1 JSON/JSONB 操作符

表 **8-28** json 和 jsonb 通用操作符

操作符	左操作数类型	右操作数类型	返回类型	描述	示例
->	Array-json(b)	int	json(b)	获得array-json元素。下标不存在返回空。	SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2; ?column? ------------- {"c":"baz"} (1 row)
->	object-json(b)	text	json(b)	通过键获得值。不存在则返回空。	SELECT '{"a":{"b":"foo"}}'::json->'a'; ?column? ------------- {"b":"foo"} (1 row)
->>	Array-json(b)	int	text	获得array-json元素。下标不存在返回空。	SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->>2; ?column? ------------- {"c":"baz"} (1 row)

操作符	左操作数类型	右操作数类型	返回类型	描述	示例
->>	object-json(b)	text	text	通过键获得值。不存在则返回空	SELECT '{"a":{"b":"foo"}}'::json->>'a'; ?column? ------------- {"b":"foo"} (1 row)
#>	container-json (b)	text[ ]	json	获取在指定路径的JSON对象，路径不存在则返回空。 **说明** DataArtsFabric SQL对象标识符支持以符号"#"结尾，为避免a#>b解析过程出现歧义，因此操作符"#>"前后需要增加空格，否则解析报错。	SELECT '{"a":{"b":{"c":1}}}'::json #> '{a, b}'; ?column? ---------- {"c":1} (1 row)
#>>	container-json (b)	text[ ]	text	获取在指定路径的JSON对象，路径不存在则返回空。	SELECT '{"a":{"b":{"c":1}}}'::json #>> '{a, b}'; ?column? ---------- {"c":1} (1 row)

表 **8-29** jsonb 支持的操作符

操作符	右操作类型	返回类型	描述	示例
=	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_eq	SELECT '{"a":{"b":{"c":1}}}'::jsonb = '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
<>	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_ne	SELECT '{"a":{"b":{"c":1}}}'::jsonb <> '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
<	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_lt	SELECT '{"a":{"b":{"c":2}}}'::jsonb < '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)

<table>
<tr><th>操作符</th><th>右操作类型</th><th>返回类型</th><th>描述</th><th>示例</th></tr>
<tr><td>></td><td>jsonb</td><td>bool</td><td>判断两个jsonb的大小关系。同函数jsonb_gt。</td><td>SELECT '{"a":{"b":{"c":2}}}'::jsonb > '{"a":{"b":{"c":1}}}'::jsonb;
 ?column?
----
 t
(1 row)</td></tr>
<tr><td><=</td><td>jsonb</td><td>bool</td><td>判断两个jsonb的大小关系。同函数jsonb_le。</td><td>SELECT '{"a":{"b":{"c":2}}}'::jsonb <= '{"a":{"b":{"c":1}}}'::jsonb;
 ?column?
----
 f
(1 row)</td></tr>
<tr><td>>=</td><td>jsonb</td><td>bool</td><td>判断两个jsonb的大小关系。同函数jsonb_ge。</td><td>SELECT '{"a":{"b":{"c":2}}}'::jsonb >= '{"a":{"b":{"c":1}}}'::jsonb;
 ?column?
----
 t
(1 row)</td></tr>
<tr><td>?</td><td>text</td><td>bool</td><td>键/元素的字符串是否存在JSON值的顶层</td><td>SELECT '{"a":1, "b":2}'::jsonb ? 'b';
 ?column?
----
 t
(1 row)</td></tr>
<tr><td>?|</td><td>text[]</td><td>bool</td><td>这些数组字符串中的任何一个是否作为顶层键存在。</td><td>SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?| '{a, b, e}'::text[];
 ?column?
----
 t
(1 row)</td></tr>
<tr><td>?&</td><td>text[]</td><td>bool</td><td>是否所有这些数组字符串都作为顶层键存在。</td><td>SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?& '{a, b, c}'::text[];
 ?column?
----
 t
(1 row)</td></tr>
<tr><td><@</td><td>jsonb</td><td>bool</td><td>左边的JSON的所有项是否全部存在于右边JSON的顶层。</td><td>SELECT '{"b":3}'::jsonb <@ '{"a":{"b":{"c":2}}, "b":3}'::jsonb;
 ?column?
----
 t
(1 row)</td></tr>
<tr><td>@></td><td>jsonb</td><td>bool</td><td>左边的JSON的顶层是否包含右边JSON的顶层所有项。</td><td>SELECT '{"a":{"b":{"c":2}}, "b":3}'::jsonb @> '{"b":3}'::jsonb;
 ?column?
----
 t
(1 row)</td></tr>
<tr><td>||</td><td>jsonb</td><td>jsonb</td><td>两个jsonb对象合并成一个。</td><td>SELECT '{"a":1, "b":2}'::jsonb || '{"c":3, "d":4}'::jsonb;
 ?column?

 {"a": 1, "b": 2, "c": 3, "d": 4}
(1 row)</td></tr>
</table>

操作符	右操作类型	返回类型	描述	示例
-	text	jsonb	删除jsonb对象删除指定的键值对。	SELECT '{"a":1, "b":2}'::jsonb - 'a'; ?column? ---------- {"b": 2} (1 row)
-	text	jsonb	删除jsonb对象删除指定的键值对。	SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb - '{a, b}'::text[]; ?column? ------------------ {"c": 3, "d": 4} (1 row)
-	int	jsonb	删除jsonb数组中下标对应的元素。	SELECT '["a", "b", "c"]'::jsonb - 2; ?column? ------------ ["a", "b"] (1 row)
#-	text[]	jsonb	删除jsonb对象中路径对应的键值对。	SELECT '{"a":{"b":{"c":{"d":1}}}, "e":2, "f":3}'::jsonb #- '{a, b}'::text[]; ?column? --------------------------- {"a": {}, "e": 2, "f": 3} (1 row)

8.3.13.2 JSON/JSONB 函数

JSON/JSONB函数表示可以用于JSON类型数据的函数。

说明

除函数array_to_json和row_to_json外，其余有关JSON/JSONB函数和操作符仅8.1.2及以上集群版本支持。

array_to_json(anyarray [, pretty_bool])

描述：返回JSON类型的数组。一个多维数组成为一个JSON数组的数组。如果pretty_bool设置为true，将在一维元素之间添加换行符。

返回类型：json

示例：

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);
array_to_json
------------------
[[1,5],[99,100]]
(1 row)
```

row_to_json(record [, pretty_bool])

描述：返回JSON类型的行。如果pretty_bool设置为true，将在第一级元素之间添加换行符。

返回类型：json

示例：

```
SELECT row_to_json(row(1,'foo'));
     row_to_json
---------------------
 {"f1":1,"f2":"foo"}
(1 row)
```

json_agg(any)

描述：将值聚集为json数组。

返回类型：array-json

示例：

```
SELECT * FROM classes;
 name | score
 -----+-------
 A    |     2
 A    |     3
 D    |     5
 D    |
(4 rows)
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)

描述：将值聚集为json对象。

返回类型：json

示例：

```
SELECT * FROM classes;
 name | score
 -----+-------
 A    |     2
 A    |     3
 D    |     5
 D    |
(4 rows)

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")

描述：从可变参数列表中构建一个可能是异构类型的JSON数组。

返回类型：json

示例：

```
SELECT json_build_array(1,2,'3',4,5);
 json_build_array
```

```
-------------------
 [1, 2, "3", 4, 5]
(1 row)
```

json_build_object(VARIADIC "any")

描述：从可变参数列表中构建JSON对象。参数列表由交替的键和值组成。其入参必须为偶数个，两两一组组成键值对。注意键不可为null。

返回类型：json

示例：

```
SELECT json_build_object('foo',1,'bar',2);
  json_build_object
------------------------
 {"foo" : 1, "bar" : 2}
(1 row)
```

json_object(text[])、json_object(text[], text[])

描述：从文本数组中构建JSON对象。

这是个重载函数，当入参为一个文本数组的时候，其数组长度必须为偶数，成员被当做交替出现的键/值对。两个文本数组的时候，第一个数组被视为键，第二个被视为值，两个数组长度必须相等。键不可为null。

返回类型：json

示例：

```
SELECT json_object('{a, 1, b, "def", c, 3.5}');
            json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');
            json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

SELECT json_object('{a,b,"a b c"}', '{a,1,1}');
              json_object
 ---------------------------------------
 {"a" : "a", "b" : "1", "a b c" : "1"}
(1 row)
```

to_json(anyelement)

描述：把参数转换为json。

返回类型：json

示例：

```
SELECT to_json('Fred said "Hi."'::text);
       to_json
---------------------
 "Fred said \"Hi.\""
(1 row)
——将json_tbl_2转换为json
postgres=# SELECT * FROM json_tbl_2;
  a |  b
```

```
---+-----
 1 | aaa
 1 | bbb
 2 | ccc
 2 | ddd
(4 rows)
postgres=# SELECT to_json(t.*) FROM json_tbl_2 t;
      to_json
-------------------
 {"a":1,"b":"bbb"}
 {"a":2,"b":"ddd"}
 {"a":1,"b":"aaa"}
 {"a":2,"b":"ccc"}
(4 rows)
```

json_strip_nulls(json)

描述：所有具有空值的对象字段被忽略，其他值保持不变。

返回类型：json

示例：

```
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)

描述：同操作符->, 返回对象中指定键对应的值。

返回类型：json

示例：

```
SELECT json_object_field('{"a": {"b":"foo"}}','a');
 json_object_field
-------------------
 {"b":"foo"}
(1 row)
```

json_object_field_text(object-json, text)

描述：同操作符->>, 返回对象中指定键对应的值。

返回类型：text

示例：

```
SELECT json_object_field_text('{"a": {"b":"foo"}}','a');
 json_object_field_text
------------------------
 {"b":"foo"}
(1 row)
```

json_array_element(array-json, integer)

描述：同操作符->, 返回数组中指定下标的元素。

返回类型：json

示例：

```
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)

描述：同操作符->>, 返回数组中指定下标的元素。

返回类型：text

示例：

```
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[])

描述：同操作符#>，返回$2所指路径的JSON值。

返回类型：json

示例：

```
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[])

描述：同操作符#>>，返回$2所指路径的text值。

返回类型：text

示例：

```
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)

描述：拆分数组，每一个元素返回一行。

返回类型：json

示例：

```
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)

描述：拆分数组，每一个元素返回一行。

返回类型：text

示例：

```
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)

描述：返回数组长度。

返回类型：integer

示例：

```
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)

描述：返回对象中顶层的所有键。

返回类型：text

示例：

```
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)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value json)

示例：

```
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)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value text)

示例：

```
SELECT * FROM json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |  value
-----+----------
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  |
(3 rows)
```

json_populate_record(anyelement, object-json [, bool])

描述：$1必须是一个复合类型的参数。将会把object-json里的每个对键值进行拆分，以键当做列名，与$1中的列名进行匹配查找，并填充到$1的格式中。

说明

JSON/JSONB函数中入参为复合类型时，可以使用CREATE EXTERNAL TABLE定义复合类型，例如：

```
CREATE EXTERNAL TABLE jpop2(a text, b INT, c timestamp);
```

返回类型：anyelement

示例：

```
SELECT * FROM json_populate_record(null::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl |   |
(1 row)
```

json_populate_recordset(anyelement, array-json [, bool])

描述：参考函数json_populate_record、jsonb_populate_record，对$2数组的每一个元素进行上述参数函数的操作，因此这也要求$2数组的每个元素都是object-json类型。

返回类型：setof anyelement

示例：

```
CREATE TYPE jpop AS (a text, b INT, c timestamp);
SELECT * FROM json_populate_recordset(null::jpop, '[{"a":1,"b":2},{"a":3,"b":4}]');
 a | b | c
---+---+---
 1 | 2 |
 3 | 4 |
(2 rows)
```

json_to_record(object-json)

描述：正如所有返回record的函数一样，调用者必须用一个AS子句显式地定义记录的结构。会将object-json的键值对进行拆分重组，把键当做列名，去匹配填充AS显示指定的记录的结构。

返回类型：record

示例：

```
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)

描述：参考函数json_to_record，对数组内个每个元素，执行上述函数的操作，因此这要求数组内的每个元素都得是object-json。

返回类型：setof record

示例：

```
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)

SELECT * FROM json_to_recordset('[{"a":1,"b":"foo","d":false},{"a":2,"b":"bar","c":true}]') AS x(a INT, b text, c BOOLEAN);
 a |  b  | c
---+-----+---
 1 | foo |
 2 | bar | t
(2 rows)
```

json_typeof(json)

描述：检测json类型。

返回类型：text

示例：

```
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[])

描述：从一个文本数组构造一个object-jsonb。这是个重载函数，当入参为一个文本数组的时候，其数组长度必须为偶数，成员被当做交替出现的键/值对。

返回类型：jsonb

示例：

```
SELECT jsonb_object('{a,1,b,2,3,NULL,"d e f","a b c"}');
                  jsonb_object
```

```
----------------------------------------------------
 {"3": null, "a": "1", "b": "2", "d e f": "a b c"}
(1 row)
```

jsonb_object(text[], text[])

描述：两个文本数组的时候，第一个数组认为是键，第二个认为是值，两个数组长度必须相等。键不可为null。

返回类型：jsonb

示例：

```
SELECT jsonb_object('{a,b,"a b c"}', '{a,1,1}');
             jsonb_object
------------------------------------
 {"a": "a", "b": "1", "a b c": "1"}
(1 row)
```

to_jsonb(anyment)

描述：将其他类型转换成对应的jsonb类型。

返回类型：jsonb

示例：

```
SELECT to_jsonb(1.1);
 to_jsonb
----------
 1.1
(1 row)
```

jsonb_agg

描述：将jsonb对象聚合成jsonb数组。

返回类型：jsonb

示例：

```
SELECT * FROM json_tbl_2;
 a |  b
---+-----
 1 | aaa
 1 | bbb
 2 | ccc
 2 | ddd
(4 rows)

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

描述：将键/值对聚集成一个JSON对象。

返回类型：jsonb

示例：

```
SELECT * FROM json_tbl_3;
 a |  b  | c
---+-----+----
 1 | aaa | 10
 1 | bbb | 20
 2 | ccc | 30
 2 | ddd | 40
(4 rows)
SELECT a, jsonb_object_agg(b, c) FROM json_tbl_3 GROUP BY a ORDER BY a;
 a |    jsonb_object_agg
---+------------------------
 1 | {"aaa": 10, "bbb": 20}
 2 | {"ccc": 30, "ddd": 40}
(2 rows)
```

jsonb_build_array([VARIADIC "any"])

描述：从一个可变参数列表构造一个可能包含异质类型的JSON数组。

返回类型：jsonb

示例：

```
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”])

描述：从一个可变参数列表构造出一个JSON对象，其入参必须为偶数个，两两一组组成键值对。注意键不可为null。

返回类型：jsonb

示例：

```
SELECT jsonb_build_object(1,2);
 jsonb_build_object
--------------------
 {"1": 2}
(1 row)
```

jsonb_strip_nulls(jsonb)

描述：所有具有空值的对象字段均被省略。其他空值保持不变。

返回类型：jsonb

示例：

```
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)

描述：同操作符->, 返回对象中指定键对应的值。

返回类型：jsonb

示例：

```
SELECT jsonb_object_field('{"a": {"b":"foo"}}','a');
 jsonb_object_field
--------------------
 {"b": "foo"}
(1 row)
```

jsonb_object_field_text(jsonb, text)

描述：同操作符->>, 返回对象中指定键对应的值。

返回类型：text

示例：

```
SELECT jsonb_object_field_text('{"a": {"b":"foo"}}','a');
 jsonb_object_field_text
-------------------------
 {"b": "foo"}
(1 row)
```

jsonb_array_element(array-jsonb, integer)

描述：同操作符->, 返回数组中指定下标的元素。

返回类型：jsonb

示例：

```
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)

描述：同操作符->>, 返回数组中指定下标的元素。

返回类型：text

示例：

```
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[])

描述：等价于操作符#>，返回$2所指路径的值。

返回类型：jsonb

示例：

```
SELECT jsonb_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
 jsonb_extract_path
--------------------
 "stringy"
(1 row)
```

jsonb_extract_path_text((jsonb, VARIADIC text[])

描述：等价于操作符#>>，返回$2所指路径的值。

返回类型：text

示例：

```
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[])

描述：输入任意的object-jsonb类型、array-jsonb类型，返回$2所指路径的值。该函数仅9.1.0及以上集群版本支持。

返回类型：SETOF jsonb

示例：

```
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[])

描述：输入任意的object-jsonb类型、array-jsonb类型，返回$2所指路径的值。该函数仅9.1.0及以上集群版本支持。

返回类型：SETOF text

示例：

```
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)

描述：拆分数组，每一个元素返回一行。

返回类型：jsonb

示例：

```
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)

描述：拆分数组，每一个元素返回一行。

返回类型：text

示例：

```
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)

描述：返回数组长度。

返回类型：integer

示例：

```
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)

描述：返回对象中顶层的所有键。

返回类型：SETOF text

示例：

```
SELECT jsonb_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}, "f1":"abcd"}');
 jsonb_object_keys
-------------------
 f1
 f2
(2 rows)
```

jsonb_each(object-jsonb)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value jsonb)

示例：

```
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)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value text)

示例：

```
SELECT * FROM jsonb_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |   value
-----+-----------
 f1  | [1, 2, 3]
 f2  | {"f3": 1}
 f4  |
(3 rows)
```

jsonb_populate_record(anyelement, object-jsonb [, bool])

描述：$1必须是一个复合类型的参数。将会把object-json里的每个对键值进行拆分，以键当做列名，与$1中的列名进行匹配查找，并填充到$1的格式中。

返回类型：anyelement

示例：

```
SELECT * FROM jsonb_populate_record(null::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl |   |
(1 row)
```

jsonb_populate_record_set(anyelement, array-jsonb [, bool])

描述：参考上述函数json_populate_record、jsonb_populate_record，对$2数组的每一个元素进行上述参数函数的操作，因此这也要求$2数组的每个元素都是object-json类型。

返回类型：setof anyelement

示例：

```
SELECT * FROM json_populate_recordset(null::jpop, '[{"a":1,"b":2},{"a":3,"b":4}]');
 a | b | c
---+---+---
 1 | 2 |
 3 | 4 |
(2 rows)
```

jsonb_to_record(object-json)

描述：正如所有返回record的函数一样，调用者必须用一个AS子句显式地定义记录的结构。会将object-json的键值对进行拆分重组，把键当做列名，去匹配填充AS显示指定的记录的结构。

返回类型：record

示例：

```
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)

描述：参考函数jsonb_to_record，对数组内个每个元素，执行上述函数的操作，因此这要求数组内的每个元素都得是object-jsonb。

返回类型：setof record

示例：

```
SELECT * FROM jsonb_to_recordset('[{"a":1,"b":"foo","d":false},{"a":2,"b":"bar","c":true}]') AS x(a INT, b text, c boolean);
 a |  b  | c
---+-----+---
 1 | foo |
 2 | bar | t
(2 rows)
```

jsonb_typeof(jsonb)

描述：检测jsonb类型。

返回类型：text

示例：

```
SELECT jsonb_typeof(to_jsonb(1.1));
 jsonb_typeof
--------------
 number
(1 row)
```

jsonb_ne(jsonb, jsonb)

描述：同操作符<>，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_ne('{"a":1, "b":2}'::jsonb, '{"a":1, "b":3}'::jsonb);
 jsonb_ne
----------
 t
(1 row)
```

jsonb_lt(jsonb, jsonb)

描述：同操作符<，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_lt('{"a":1, "b":2}'::jsonb, '{"a":1, "b":3}'::jsonb);
 jsonb_lt
----------
 t
(1 row)
```

jsonb_gt(jsonb, jsonb)

描述：同操作符>，比较两个值的大小。

返回类型：bool

示例：

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

jsonb_le(jsonb, jsonb)

描述：同操作符<=，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_le('["a", "b"]', '{"a":1, "b":2}');
 jsonb_le
----------
 t
(1 row)
```

jsonb_ge(jsonb, jsonb)

描述：同操作符>=，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_ge('["a", "b"]', '{"a":1, "b":2}');
 jsonb_ge
----------
 f
(1 row)
```

jsonb_eq(jsonb, jsonb)

描述：同操作符=，比较两个值的大小

返回类型：bool

示例：

```
SELECT jsonb_eq('["a", "b"]', '{"a":1, "b":2}');
 jsonb_eq
----------
 f
(1 row)
```

jsonb_cmp(jsonb, jsonb)

描述：比较大小，正数表示大于，负数表示小于，0表示相等。

返回类型：integer

示例：

```
SELECT jsonb_cmp('["a", "b"]', '{"a":1, "b":2}');
 jsonb_cmp
-----------
 -1
(1 row)
```

jsonb_exists(jsonb, text)

描述：同操作符?，字符串$2是否存在$1的顶层以key\elem\scalar的形式存在。

返回类型：bool

示例：

```
SELECT jsonb_exists('["1",2,3]', '1');
 jsonb_exists
--------------
 t
(1 row)
```

jsonb_exists_any(jsonb, text[])

描述：同操作符?|，字符串数组$2里面是否存在的元素，在$1的顶层以key\elem\scalar的形式存在。

返回类型：

示例：

```
SELECT jsonb_exists_any('["1","2",3]', '{1, 2, 4}');
 jsonb_exists_any
------------------
 t
(1 row)
```

jsonb_exists_all(jsonb, text[])

描述：同操作符?&，字符串数组$2里面是否所有的元素，都在$1的顶层以key\elem\scalar的形式存在。

返回类型：

bool

示例：

```
SELECT jsonb_exists_all('["1","2",3]', '{1, 2}');
 jsonb_exists_all
------------------
 t
(1 row)
```

jsonb_contained(jsonb, jsonb)

描述：同操作符<@, 判断$1中的所有元素是否在$2的顶层存在。

返回类型：bool

示例：

```
SELECT jsonb_contained('[1,2,3]', '[1,2,3,4]');
 jsonb_contained
-----------------
```

```
 t
(1 row)
```

jsonb_contains(jsonb, jsonb)

描述：同操作符@>, 判断$1中的顶层所有元素是否包含在$2的所有元素。

返回类型：bool

示例：

```
SELECT jsonb_contains('{"a":1, "b":2, "c":3}'::jsonb, '{"a":1}');
 jsonb_contains
-----------------
 t
(1 row)
```

jsonb_concat(jsonb, jsonb)

描述：连接两个jsonb对象为一个jsonb。

返回类型：jsonb

示例：

```
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)

描述：删除jsonb中的key值对应的键值对。

返回类型：jsonb

示例：

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

jsonb_delete_idx(jsonb, text)

描述：删除数组下标对应的元素。

返回类型：jsonb

示例：

```
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[])

描述：删除jsonb数组中的多个元素。

返回类型：jsonb

示例：

```
SELECT jsonb_delete_array('["a", "b", "c"]'::jsonb , 'a', 'b');
 jsonb_delete_array
--------------------
 ["c"]
(1 row)
```

jsonb_delete_path(jsonb, text[])

描述：删除jsonb数组中指定路径的元素。

返回类型：jsonb

示例：

```
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])

描述：返回target，用path指定的部分被new_value替换，或者如果create_missing为true（默认值为true）且path指定的项不存在，则添加new_value。与面向路径的运算符一样，path中出现的负整数从JSON数组的末尾开始计数。

返回类型：jsonb

示例：

```
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)

描述：以缩进的JSON文本形式返回。

返回类型：jsonb

示例：

```
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])

描述：返回target，并插入new_value。如果path指定的target部分位于JSONB数组中，则new_value将在目标之前或insert_after为true（默认值为false）之后插入。如果在JSONB对象中由path指定的target部分，则仅当target不存在时才插入new_value。与面向路径的运算符一样，path中出现的负整数从JSON数组的末尾开始计数。

返回类型：jsonb

示例：

```
SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');
        jsonb_insert
-------------------------------
 {"a": [0, "new_value", 1, 2]}
(1 row)
```

ts_headline([config regconfig,] document jsonb, query tsquery [, options text])

描述：高亮jsonb搜索结果。

返回类型：jsonb

示例：

```
SELECT ts_headline('english',
'[{"id":9928,"user_id":4562,"user_name":"9LOHR4","create_time":"2021-06-22T16:28:16.504518+08:00"},
{"id":9959,"user_id":5524,"user_name":"YID07D","create_time":"2021-06-22T16:28:16.557228+08:00"},
{"id":9962,"user_id":7991,"user_name":"7C6QOM","create_time":"2021-06-22T16:28:16.56234+08:00"}]'::json
b,
 to_tsquery('english', '9LOHR4'), 'StartSel = <, StopSel = >');

ts_headline

------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------
 [{"id": 9928, "user_id": 4562, "user_name": "<9LOHR4>", "create_time":
"2021-06-22T16:28:16.504518+08:00"}, {"id": 9959, "user_id": 5524, "user_name": "YID07D", "create_time":
"2021-06-22T16:28:16.557228+08:00"}, {"id": 9962, "user_id": 7991, "user_name": "7C6QOM",
"create_time": "2021-06-22T16:28:16.56234+08:00"}]
(1 row)
```

json_to_tsvector(config regconfig,] json, jsonb)

描述：将json格式转换为用于支持全文检索的文件格式tsvector。

返回类型：jsonb

示例：

```
SELECT json_to_tsvector('{"a":1, "b":2, "c":3}'::json, to_jsonb('key'::text));
 json_to_tsvector
------------------
 'b':2 'c':4
(1 row)
```

8.3.14 条件表达式函数

coalesce(expr1, expr2, ..., exprn)

描述：返回参数列表中第一个非NULL的参数值。

COALESCE(expr1, expr2) 等价于CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END。

示例：

```
SELECT coalesce(NULL,'hello');
 coalesce
----------
 hello
(1 row)
```

说明

- 如果表达式列表中的所有表达式都等于NULL，则本函数返回NULL。
- 它常用于在显示数据时用缺省值替换NULL。
- 和CASE表达式一样，COALESCE不会计算不需要用来判断结果的参数；即在第一个非空参数右边的参数不会被计算。

decode(base_expr, compare1, value1, Compare2,value2, … default)

描述：把base_expr与后面的每个compare(n) 进行比较，如果匹配返回相应的value(n)。如果没有发生匹配，则返回default。

示例：

```
SELECT decode('A','A',1,'B',2,0);
 case
------
 1
(1 row)
```

if(bool_expr, expr1, expr2)

描述：当bool_expr为true时，返回expr1，否则返回expr2。

if(bool_expr, expr1, expr2) 等价于CASE WHEN bool_expr = true THEN expr1 ELSE expr2 END。

示例：

```
SELECT if(1 < 2, 'yes', 'no');
 if
-----
 yes
(1 row)
```

说明

参数expr1和expr2可以为任意类型，返回结果类型规则请参考**UNION，CASE和相关构造**。

ifnull(expr1, expr2)

描述：当expr1不为NULL时，返回expr1，否则返回expr2。

ifnull(expr1, expr2) 逻辑上等价于CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END。

示例：

```
SELECT ifnull(NULL,'hello');
 ifnull
--------
 hello
(1 row)
```

说明

参数expr1和expr2可以为任意类型，返回结果类型规则请参考**UNION，CASE和相关构造**。

isnull(expr)

描述：当expr为NULL时，返回true，否则返回false。

isnull(expr) 逻辑上等价于expr IS NULL。

示例：

```
SELECT isnull(NULL), isnull('abc');
 isnull | isnull
--------+--------
 t      | f
(1 row)
```

nullif(expr1, expr2)

描述：当且仅当expr1和expr2相等时，NULLIF才返回NULL，否则它返回expr1。

nullif(expr1, expr2) 逻辑上等价于CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END。

示例：

```
SELECT nullif('hello','world');
 nullif
--------
 hello
(1 row)
```

备注：

如果两个参数的数据类型不同，则：

- 两种数据类型之间存在隐式转换，则以其中优先级较高的数据类型为基准将另一个参数隐式转换成该类型，转换成功则进行计算，转换失败则返回错误。如：
```
SELECT nullif('1234'::VARCHAR,123::INT4);
 nullif
--------
   1234
(1 row)
SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
```
- 两种数据类型之间不存在隐式转换，则返回错误 。如：
```
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)

描述：如果expr1为NULL则返回expr2。如果expr1非NULL，则返回expr1。

示例：

```
SELECT nvl('hello','world');
  nvl
-------
 hello
(1 row)
```

说明

参数expr1和expr2可以为任意类型，当NVL的两个参数不属于同类型时，看第二个参数是否可以向第一个参数进行隐式转换，如果可以则返回第一个参数类型。如果第二个参数不能向第一个参数进行隐式转换而第一个参数可以向第二个参数进行隐式转换，则返回第二个参数的类型。如果两个参数之间不存在隐式类型转换并且也不属于同一类型则报错。

greatest(expr1 [, ...])

描述：获取并返回参数列表中值最大的表达式的值。

- ORA和TD兼容模式下，返回结果为所有非null参数的最大值。
- MySQL兼容模式下，入参中存在null时，返回结果为null。

示例：

```
SELECT greatest(1*2,2-3,4-1);
 greatest
----------
        3
(1 row)
SELECT greatest('ABC', 'BCD', 'CDE');
 greatest
----------
 CDE
(1 row)
```

least(expr1 [, ...])

描述：获取并返回参数列表中值最小的表达式的值。

- ORA和TD兼容模式下，返回结果为所有非null参数的最小值。
- MySQL兼容模式下，入参中存在null时，返回结果为null。

示例：

```
SELECT least(1*2,2-3,4-1);
 least
-------
    -1
(1 row)
SELECT least('ABC','BCD','CDE');
 least
--------
 ABC
(1 row)
```

8.3.15 范围函数和操作符

8.3.15.1 范围操作符

=

描述：等于

示例：

```
SELECT int4range(1,5) = '[1,4]'::int4range AS RESULT;
 result
--------
 t
(1 row)
```

<>

描述：不等于

示例：

```
SELECT numrange(1.1,2.2) <> numrange(1.1,2.3) AS RESULT;
 result
--------
 t
(1 row)
```

<

描述：小于

示例：

```
SELECT int4range(1,10) < int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
```

>

描述：大于

示例：

```
SELECT int4range(1,10) > int4range(1,5) AS RESULT;
 result
--------
 t
(1 row)
```

<=

描述：小于或等于

示例：

```
SELECT numrange(1.1,2.2) <= numrange(1.1,2.2) AS RESULT;
 result
--------
 t
(1 row)
```

>=

描述：大于或等于

示例：

```
SELECT numrange(1.1,2.2) >= numrange(1.1,2.0) AS RESULT;
 result
--------
 t
(1 row)
```

@>

描述：包含范围、包含元素

示例：

```
SELECT int4range(2,4) @> int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
SELECT '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：范围包含于、元素包含于

示例：

```
SELECT int4range(2,4) <@ int4range(1,7) AS RESULT;
 result
--------
 t
(1 row)
SELECT 42 <@ int4range(1,7) AS RESULT;
 result
--------
 f
(1 row)
```

&&

描述：重叠（有共同点）

示例：

```
SELECT int8range(3,7) && int8range(4,12) AS RESULT;
 result
--------
 t
(1 row)
```

<<

描述：范围值是否比另一个范围值的最小值还小（没有交集）

示例：

```
SELECT int8range(1,10) << int8range(100,110) AS RESULT;
 result
--------
 t
(1 row)
```

>>

描述：范围值是否比另一个范围值的最大值还大（没有交集）

示例：

```
SELECT int8range(50,60) >> int8range(20,30) AS RESULT;
 result
--------
 t
(1 row)
```

&<

描述：范围值的最大值是否不超过另一个范围值的最大值。

示例：

```
SELECT int8range(1,20) &< int8range(18,20) AS RESULT;
 result
--------
 t
(1 row)
```

&>

描述：范围值的最小值是否不小于另一个范围值的最小值。

示例：

```
SELECT int8range(7,20) &> int8range(5,10) AS RESULT;
 result
--------
 t
(1 row)
```

-|-

描述：相邻

示例：

```
SELECT numrange(1.1,2.2) -|- numrange(2.2,3.3) AS RESULT;
 result
--------
 t
(1 row)
```

+

描述：并集

示例：

```
SELECT numrange(5,15) + numrange(10,20) AS RESULT;
 result
--------
```

```
 [5,20)
(1 row)
```

*

描述：交集

示例：

```
SELECT int8range(5,15) * int8range(10,20) AS RESULT;
 result
---------
 [10,15)
(1 row)
```

-

描述：差集

示例：

```
SELECT int8range(5,15) - int8range(10,20) AS RESULT;
 result
--------
 [5,10)
(1 row
```

说明

- 简单的比较操作符<，>，<=和>=先比较下界，只有下界相等时才比较上界。
- <<、>>和-|-操作符当包含空范围时也会返回false；也就是，不认为空范围在其他范围之前或之后。
- 并集和差集操作符的执行结果无法包含两个不相交的子范围。

8.3.15.2 范围函数

lower(anyrange)

描述：范围的下界

返回类型：范围元素类型

示例：

```
SELECT lower(numrange(1.1,2.2)) AS RESULT;
 result
--------
    1.1
(1 row)
```

upper(anyrange)

描述：范围的上界

返回类型：范围元素类型

示例：

```
SELECT upper(numrange(1.1,2.2)) AS RESULT;
 result
--------
```

```
   2.2
(1 row)
```

isempty(anyrange)

描述：范围是否为空

返回类型：boolean

示例：

```
SELECT isempty(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

lower_inc(anyrange)

描述：是否包含下界

返回类型：boolean

示例：

```
SELECT lower_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 t
(1 row)
```

upper_inc(anyrange)

描述：是否包含上界

返回类型：boolean

示例：

```
SELECT upper_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

lower_inf(anyrange)

描述：下界是否为无穷

返回类型：boolean

示例：

```
SELECT lower_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

upper_inf(anyrange)

描述：上界是否为无穷

返回类型：boolean

示例：

```
SELECT upper_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

说明

如果范围是空或者需要的界限是无穷的，lower和upper函数将返回null。lower_inc、upper_inc、lower_inf和upper_inf函数均对空范围返回false。

8.3.16 返回集合的函数

8.3.16.1 序列号生成函数

generate_series()函数根据指定的开始值(start)、结束值(stop)和步长(step)返回一个基于系列的集合。

generate_series()函数的入参中，当step是正数且start大于stop，则返回零行。相反，当step是负数且start小于stop，则返回零行。如果任何输入为NULL也会返回零行。如果step为零则会报错。

generate_series(start, stop)

描述：生成一个数值序列，从start到stop，步长默认为1。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

示例：

```
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)

描述：生成一个数值序列，从start到stop，步长为step。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

示例：

```
SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,6,-5);
 generate_series
-----------------
(0 rows)

SELECT * FROM generate_series(4,3,0);
ERROR:  step size cannot equal zero
```

generate_series(start, stop, step interval)

描述：生成一个数值序列，从start到stop，步长为step。

参数类型：timestamp或timestamp with time zone

返回值类型：setof timestamp或setof timestamp with time zone（与参数类型相同）

示例：

```
--这个示例应用于date-plus-integer操作符。
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)
```

8.3.16.2 下标生成函数

generate_subscripts(array anyarray, dim int)

描述：生成一系列包括给定数组的下标。

返回值类型：setof int

示例：

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
```

```
 1
 2
 3
 4
(4 rows)
```

generate_subscripts(array anyarray, dim int, reverse boolean)

描述：生成一系列包括给定数组的下标。当reverse为真时，该系列则以相反的顺序返回。

返回值类型：setof int

示例：

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1,TRUE) AS s;
 s
---
 4
 3
 2
 1
(4 rows)
```

8.3.17 系统信息函数

8.3.17.1 会话信息函数

pg_backend_pid()

描述：当前会话连接的服务进程的进程ID。

返回值类型：integer

示例：

```
SELECT pg_backend_pid();
 pg_backend_pid
-----------------
 140229352617744
(1 row)
```

pg_postmaster_start_time()

描述：服务器启动时间。pg_postmaster_start_time返回服务器启动时的timestamp with time zone。

返回值类型：timestamp with time zone

示例：

```
SELECT pg_postmaster_start_time();
   pg_postmaster_start_time
------------------------------
 2017-08-30 16:02:54.99854+08
(1 row)
```

session_user

描述：会话用户名。session_user通常是连接当前数据库的初始用户。

返回值类型：name

示例：

```
SELECT session_user;
 session_user
--------------
 dbadmin
(1 row)
```

user

描述：等价于current_user。

返回值类型：name

示例：

```
SELECT user;
 current_user
--------------
 dbadmin
(1 row)
```

version()

描述：版本信息。version返回一个描述服务器版本信息的字符串。

返回值类型：text

示例：

```
SELECT version();
                                                            version
---------------------------------------------------------------------------------------------------------------------------------------
 PostgreSQL 9.2.4 gsql ((GaussDB 8.2.1 build 39137c2d) compiled at 2022-09-23 15:43:11 commit 3629 last mr 5138 release) on x86_64-unknown-linux-gnu, compiled by g++ (GCC) 5.4.0, 64-bit
(1 row)
```

8.3.17.2 系统表信息函数

format_type(type_oid, typemod)

描述：获取数据类型的SQL名称。

返回类型：text

备注：

format_type通过数据类型的类型OID以及可能的类型修饰词，返回其SQL名称。如果不知道具体的修饰词，则在类型修饰词的位置传入NULL。类型修饰词一般只对有长度限制的数据类型有意义。format_type所返回的SQL名称中包含数据类型的长度值，其大小是：实际存储长度len - sizeof(int32)，单位字节。数据存储时需要32位的空间来存储用户对数据类型的自定义长度信息，即实际存储长度要比用户定义长度多4个字节。在下例中，format_type返回的SQL名称为“character varying(6)”，6表示varchar类型的长度值是6字节，因此该类型的实际存储长度为10字节。

```
SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'), 10);
     format_type
----------------------
 character varying(6)
(1 row)
```

pg_get_schemas(schemaPattern）

描述：查询当前catalog中的schema, 参数schemaPattern可以是具体schema名字或者用%表示的通配符。如果schemaPattern为空（使用单引号替代参数），则返回所有的schema。

返回类型：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)

描述：查询当前catalog中满足条件的表信息，参数schemaPattern可以指定schema名字或用%表示的通配符；参数tablePattern可以指定表名字或用%表示的通配符；tableTypes可以指定表类型。具体表类型为MANAGED_TABLE、EXTERNAL_TABLE、VIRTUAL_VIEW、 MATERIALIZED_VIEW、 DICTIONARY_TABLE这几种类型。这三个参数如果为空（使用单引号替代参数），则返回所有的catalog、schema、表名、表类型、描述信息。

返回类型：返回如下字段组成的行集。

- 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)

描述：查询当前catalog中满足条件的列信息。参数schemaPattern可以指定schema名字或用%表示的通配符；参数tableNamePattern可以指定表名字或用%表示的通配符；参数columnNamePattern可以指定列名字或用%表示的通配符。

返回类型：返回如下字段组成的行集

- 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)
```

pg_get_partition_keys(schema, table)

描述：查询当前catalog中指定schema名字、table名字的分区键信息。对于非分区表，则返回空集。

返回类型：返回如下字段组成的行集。

- table：表名字。
- partition_key：分区键信息。

```
fabricsql => 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)

描述：查询当前catalog中当前schema下指定table名字的数据分区信息。

返回类型：返回如下字段组成的行集：

- partition_name：数据分区信息。
- location：数据分区对应的存储位置信息。

```
fabricsql => 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)
```

8.3.17.3 系统函数信息函数

pv_builtin_functions()

描述：查询系统内置函数的信息。

返回类型：record

示例：

```
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
----------+--------------+----------+---------+---------+---------+-------------+--------------+----------+-------------
+-----------+--------------+-------------+-----------+------------+----------+-----------------+-----
-------+-------------+----------------+-------------+-------------+----------------+----------+--------+-----------
+------------------+------------+--------------+------------+-----
 byteaout |           11 |       10 |      12 |       1 |       0 |           0 | -            | f        | f           | f         | f
| t           | f         | i           |        1 |               0 |
  2275 | 17          |                |             |             |                | byteaout |        |           |                  | f
|              | f          |  31
(1 row)
```

8.3.17.4 状态信息函数

pgxc_query_wait_status(session_id)

描述：用于查看指定会话的执行单元线程之间的等待状态。

返回值类型：record

函数返回信息如下：

表 **8-30** pgxc_query_wait_status(session_id)字段

名称	类型	描述
actor_name	text	actor的名称。
actor_id	text	actor的id。
actor_ip	text	actor的ip。
db_name	text	数据库名称。
thread_name	text	线程名称。
query_id	bigint	查询id。
tid	bigint	当前线程的线程号。
lwtid	integer	当前线程的轻量级线程号。
ptid	integer	streaming线程的父线程。
tlevel	integer	streaming线程的层级。
smpid	integer	并行线程的id。
wait_status	text	当前线程的等待状态。
wait_event	text	如果wait_status是acquire lock、acquire lwlock、wait io三种类型，此列描述具体的锁、轻量级锁、IO的信息。否则为空。

示例：

```
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

--------------+--------------------------------------------+-------------+---------+-------------+-------------------
+-----------------+---------+------+--------+-------+-------------+-----------
-
 coordinator1 | runtime-c7010000-0000-4000-8bad-1509e41e3b67 | 10.42.0.240 |         | gsql        |
72339069014638600 | 140406720987184 | 4194244 |      |      0 |     0 | none        |
(1 row)
```

yr_api_status()

描述：用于查看会话过程中YR接口调用的统计信息。

返回值类型：record

函数返回信息如下：

表 **8-31** yr_api_status()字段

名称	类型	描述
actor_name	text	actor的名称。
api	text	接口名称。
total_call	bigint	调用的总次数。
failed_call	bigint	调用失败的次数。
avg_call_during	bigint	平均调用时长。 单位：毫秒
min_call_during	bigint	最小调用时长。 单位：毫秒
max_call_during	bigint	最大调用时长。 单位：毫秒
total_data	bigint	总数据量。 单位：字节
avg_data_rate	bigint	平均数据处理速率。 单位：每毫秒字节数
min_data_rate	bigint	最小数据处理速率。 单位：每毫秒字节数
max_data_rate	bigint	最大数据处理速率。 单位：每毫秒字节数

示例：

```
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
--------------+--------------------+------------+-------------+-----------------+-----------------+-----------------
+------------+---------------+---------------+---------------
 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)
```

8.3.18 系统管理函数

8.3.18.1 内存管理函数

内存管理函数仅25.5.0及以上集群版本支持。

pgxc_total_memory_detail()

描述：用于查看所有后端actor节点的内存使用情况。

返回值类型：record

函数返回信息如下：

表 8-32 pgxc_total_memory_detail()输出结果的字段

名称	类型	描述
nodename	text	节点名称。

名称	类型	描述
memorytype	text	内存的名称，包括以下几种： ● max_process_memory：DataArtsFabric SQL集群实例所占用的内存大小。 ● process_used_memory：DataArtsFabric SQL进程所使用的内存大小。 ● max_dynamic_memory：最大动态内存。 ● dynamic_used_memory：已使用的动态内存。 ● dynamic_peak_memory：内存的动态峰值。 ● dynamic_used_shrctx：最大动态共享内存上下文。 ● dynamic_peak_shrctx：共享内存上下文的动态峰值。 ● max_shared_memory：最大共享内存。 ● shared_used_memory：已使用的共享内存。 ● max_cstore_memory：列存所允许使用的最大内存。 ● cstore_used_memory：列存已使用的内存大小。 ● other_used_memory：其他已使用的内存大小。 ● mmap_used_memory：mmap使用的内存大小。
memorymbytes	integer	内存使用的大小，单位MB。

shared_memory_context(actor_name cstring)

描述：查询指定actor进程中共享内存上下文的统计信息。

参数：actor_name，表示actor名称。

返回值类型：record

示例：

```
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
 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))

描述：查询指定actor、名字为contextname的共享内存上下文申请的所有chunk信息。

参数：

- actor_name， 指定actor的名字
- contextname，表示内存上下文名称。

返回值类型：record

示例：

```
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()

描述：用于查看对应actor或者所有后端actor节点的线程内存使用情况，以MemoryContext维度来统计。

其中内存上下文“TempSmallContextGroup”，记录当前线程中所有内存上下文字段“totalsize”小于8192字节的信息汇总，并且内存上下文统计计数记录到“usedsize”字段中。所以在结果中，“TempSmallContextGroup”内存上下文中的“totalsize”和“freesize”是该线程中所有内存上下文“totalsize”小于8192字节的汇总总和，usedsize字段表示统计的内存上下文个数。

参数说明：

- pgxc_session_memory_detail()，无参数，查询CN和所有DN进程中所有线程的内存上下文的统计信息。
- pgxc_session_memory_detail(actor_name)，入参为actor_name，输出指定CN或DN进程中所有线程的内存上下文的统计信息。

返回值类型：record

函数返回信息如下：

表 8-33 pgxc_session_memory_detail 字段

名称	类型	描述
sessid	text	线程启动时间+线程标识（字符串信息为timestamp.threadid）。
sesstype	text	线程名称。
contextname	text	内存上下文名称。
level	smallint	当前上下文在整体内存上下文中的层级。
parent	text	父内存上下文名称。
totalsize	bigint	当前内存上下文的内存总数，单位Byte。
freesize	bigint	当前内存上下文中已释放的内存总数，单位Byte。
usedsize	bigint	当前内存上下文中已使用的内存总数，单位Byte；“TempSmallContextGroup”内存上下文中该字段含义为统计计数。

session_memory_context_chunk(actor_name cstring, tid int64, contextname char(64))

描述：查询指定actor上某个线程创建的某个内存上下文申请的所有chunk信息。

参数说明：

- actor_name， 表示actor名字。
- tid，表示线程ID。
- contextname，表示内存上下文名称。

返回值类型：record

示例：

```
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)

描述：查询jemalloc提供的内存统计信息。

参数：actor_name， 表示actor名字。

返回值类型：record

示例：

```
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)

功能说明：对DataArtsFabric SQL actor进程中malloc等内存分配进行采样，并输出采样结果。

表 8-34 jemalloc_heap_profiling 参数的含义

参数	类型	含义
actor_name	字符串	actor名字。
type	整数	1：开启内存trace，记录malloc等调用栈信息。 0：关闭内存trace，返回本次所记录的malloc等调用栈的采样信息。

返回值类型：

- actor_name：actor名字。
- profiling：内存分配采样信息。

备注：

- 如果失败提示信息为“Memory profiling failed, check if $MALLOC_CONF contain 'prof:true'.”，说明未设置环境变量MALLOC_CONF=prof:true，原因是

未使用带内存采样功能的jemalloc库。请在使用gov_client启动CN的参数中添加-j选项。

- 如果失败提示信息为“Type %d is not supported. The valid range is 0-1.”，说明用户输入参数错误，正确数值为0或1。

输出内存调用栈信息

操作步骤：

步骤1 输出内存调用栈信息，在DataArtsFabric SQL进程所在目录输出trace文件。

```
SELECT profiling FROM jemalloc_heap_profiling('coordinator1', 0);
```

步骤2 将查询结果另存为trace文件。

步骤3 使用jemalloc中提供的jeprof工具，解析日志信息。

方式1：以TEXT格式输出。

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 >prof.txt
```

方式2：以PDF格式输出。

```
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 > prof.pdf
```

说明

- 解析内存调用栈信息，需要依靠DataArtsFabric SQL源码进行分析，需要将trace文件返回给研发工程师进行分析。
- 分析trace文件，需要使用jeprof工具，该工具由Jemalloc生成。在常规使用中，需要依赖perl环境，如果需要生成pdf调用图，需要安装与操作系统匹配的GraphViz工具。

----结束

示例

使用gov_client -j重新启动数据库。

在数据库运行期间，打开内存trace记录功能：

```
SELECT profiling fromjemalloc_heap_profiling('coordinator1', 1);
```

在数据库运行期间，关闭内存trace记录功能：

```
SELECT profiling fromjemalloc_heap_profiling('coordinator1', 0);
```

将查询结果复制到trace文件中，以text格式或PDF格式输出：

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 >prof.txt
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 > prof.pdf
```

8.3.19 数据库对象函数

8.3.19.1 排序规则版本函数

pg_collation_actual_version (oid)

描述：返回当前安装在操作系统中的该排序规则对象的实际版本，目前仅对case_insensitive有效。

返回值类型：text

示例：

```
SELECT oid FROM pg_collation WHERE collname ='case_insensitive';
 oid
------
 3300
(1 row)

SELECT pg_collation_actual_version(3300);
 pg_collation_actual_version
-----------------------------
 153.14
(1 row)
```

8.3.19.2 数据库对象尺寸函数

pg_table_size(regclass)

描述：获取表大小，计算表目录下所有文件的总大小。

返回值类型：int64

pg_table_files(regclass)

描述：获取表大小，返回表目录下所有文件的路径和大小信息。

返回值类型：

- file_path：text类型，文件路径。
- file_name：text类型，文件名。
- file_size：int64类型，文件大小。
- file_etag：text类型，文件etag。
- file_last_modify_time：int类型，文件最后修改时间。

pg_size_pretty(bigint)

描述：把字节计算的尺寸转换成一个易读的尺寸。

返回值类型：text

pg_size_pretty(numeric)

描述：把用数值表示的字节计算的尺寸转换成一个易读的尺寸。

返回值类型：text

备注：pg_size_pretty用于把其他函数的结果格式化成一种易读的格式，可以根据情况使用KB 、MB 、GB 、TB。

8.3.20 统计信息函数

pg_stat_get_activity(integer)

描述：返回一个关于带有特殊PID的后台进程的记录信息，当参数为NULL时，则返回每个活动的后台进程的记录。返回结果是PG_STAT_ACTIVITY视图中的一个子集，不包含connection_info列。

返回值类型：setof record

gs_stack()

描述：获取CN或DN进程的堆栈信息，函数入参actor_name、tid均需要通过视图或其他函数进行获取，不支持lwtid。该函数仅25.5.0及以上集群版本支持。

gs_stack()提供了比gdb、gstack等命令更灵活、更快速的堆栈获得手段。仅建议管理员用户在问题分析定位过程中使用，不建议作为日常的监控工具使用。

返回值字段：

- actor_name：线程所在的actor名称。
- tid：线程ID。
- lwtid：轻量级线程ID。
- stack：tid或lwtid线程对应的堆栈信息。

gs_stack()函数有三种使用方式：

- 方式一：gs_stack()函数入参为空，表示打印CN和DN所有线程的堆栈。

  说明

  - 可通过gsql连接CN执行。
  - 获取CN和所有DN上所有线程的运行堆栈，运行时间长，对正常业务运行有影响，不建议频繁使用。

```
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
```

- 方式二：gs_stack(actor_name)函数入参值为指定的actor_name，表示打印名称为actor_name的actor上所有线程的堆栈。

  说明

  - 可通过gsql连接CN执行。
  - actor_name必须为CN或DN的名字，否则会报错“invalid actor_name”。
  - actor_name需通过其他视图或函数获取（例如函数pgxc_query_wait_status）。

```
select * from gs_stack('coordinator1');
  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
```

- 方式三：gs_stack(actor_name,tid)函数入参值为指定的actor_name和tid，表示打印名称为actor_name的actor上指定tid线程的堆栈。

说明

- 可通过gsql连接CN执行。
- actor_name必须为CN或DN的名字，否则会报错“invalid actorName”。
- actor_name和tid需通过其他视图或函数获取（例如函数pgxc_query_wait_status）。
- 获取指定CN或DN进程内指定线程的运行堆栈，运行时间短，对正常业务运行无影响，推荐使用。

```
select * from gs_stack('coordinator1',106927505998400);
  actor_name  |        tid        | lwtid
```

```
|
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()

描述：获取执行中SQL语句的监控信息，无法获取执行结束的SQL语句监控信息，返回结果每一行代表一个语句。

函数返回字段：

字段名称	类型	描述
session_id	text	语句的Session ID。
statement_id	text	语句ID。
start_time	timestampt with zone	语句起始时间。
duration	bigint	语句运行时间。 单位：ms
estimate_total_time	bigint	估计的语句运行时长。 单位：ms
estimate_left_time	bigint	估计语句剩余运行时长 单位：ms
cn_actor_info	text	CN的Serverless信息。

字段名称	类型	描述
dn_actor_info	text	DN的Serverless信息。
startup_coordinator_duration	bigint	启动CN时长。 单位：ms
create_session_duration	bigint	创建Session时长。 单位：ms
enqueue_statement_duration	bigint	语句加入任务队列时长。 单位：ms
queue_duration	bigint	语句在任务队列中排队时长。 单位：ms
write_resultset_duration	bigint	语句写结果集时长。 单位：ms
queryid	bigint	Debug Query的ID。
schemaname	text	语句执行时的Schema名称。
query	text	语句。
query_plan	text	explain信息
tid	bigint	线程ID。
parse_time	bigint	从解析到执行开始之前的时间。 单位：ms
estimate_memory	int	语句估计内存。 单位：MB
min_dn_time	bigint	最小DN时长。 单位：ms
max_dn_time	bigint	最大DN时长。 单位：ms
average_dn_time	bigint	平均DN时长。 单位：ms
dn_time_skew_percent	int	DN时长偏斜百分比。 单位：%
dn_min_cpu_time	bigint	最小DN CPU时长。 单位：ms
dn_max_cpu_time	bigint	最大DN CPU时长。 单位：ms

字段名称	类型	描述
dn_average_cpu_time	bigint	平均DN CPU时长。 单位：ms
dn_total_cpu_time	bigint	DN CPU总时长。 单位：ms
dn_cpu_skew_percent	int	DN CPU时长偏斜百分比。 单位：%
dn_min_peak_memory	int	所有DN最小峰值内存。 单位：MB
dn_max_peak_memory	int	所有DN最大峰值内存。 单位：MB
dn_average_peak_memory	int	所有DN平均峰值内存。 单位：MB
dn_memory_skew_percent	int	内存倾斜率。 单位：%
dn_spill_num	int	下盘DN数量
dn_spill_count	int	下盘DN次数总量
dn_min_spill_size	bigint	最大下盘Size。 单位：MB
dn_max_spill_size	bigint	最小下盘Size。 单位：MB
dn_average_spill_size	bigint	平均下盘Size。 单位：MB
dn_spill_skew_percent	int	下盘倾斜率。 单位：%
dn_write_disk_total_size	bigint	dn下盘写磁盘总量。 单位：MB
dn_write_disk_max_size	bigint	dn最大下盘写磁盘量。 单位：MB
dn_write_disk_min_size	bigint	dn最小下盘写磁盘量。 单位：MB
dn_spill_obs_total_size	bigint	dn下盘写OBS总量。 单位：MB
dn_spill_obs_max_size	bigint	dn最大下盘写OBS量。 单位：MB

字段名称	类型	描述
dn_spill_obs_min_size	bigint	dn最小下盘写OBS量。 单位：MB
dn_write_obs_total_size	bigint	dn下盘直写OBS总量。 单位：MB
dn_write_obs_max_size	bigint	dn下盘最大直写OBS量。 单位：MB
dn_write_obs_min_size	bigint	dn下盘最小直写OBS量。 单位：MB
cn_spill_count	int	cn下盘次数
cn_spill_size	bigint	cn下盘总量。 单位：MB
cn_spill_obs_size	bigint	cn下盘写OBS量。 单位：MB
cn_write_obs_size	bigint	cn下盘直写OBS量。 单位：MB
cn_write_obs_disk_size	bigint	cn下盘写本地磁盘量。 单位：MB
disk_cache_hit_ratio	numeric	磁盘缓存命中率。
disk_cache_disk_read_size	bigint	磁盘缓存读取大小。 单位：B
disk_cache_disk_write_size	bigint	磁盘缓存写入大小。 单位：B
disk_cache_remote_read_size	bigint	磁盘缓存远程读取大小。 单位：B
disk_cache_remote_read_time	bigint	磁盘缓存远程读取次数。 单位：次
vfs_scan_bytes	bigint	OBS 文件系统扫描字节数。 单位：B
vfs_remote_read_bytes	bigint	OBS 文件系统远程读取字节数。 单位：B
disk_cache_load_time	bigint	磁盘缓存本地加载时间。 单位：us

字段名称	类型	描述
disk_cache_conflict_count	bigint	磁盘缓存块哈希冲突次数。 单位：次
disk_cache_error_count	bigint	磁盘缓存错误次数。 单位：次
disk_cache_error_code	bigint	磁盘缓存错误码。
obs_io_req_avg_rtt	bigint	OBS IO请求平均往返时间。 单位：us
obs_io_req_avg_latency	bigint	OBS IO请求平均延迟。 单位：us
obs_io_req_latency_gt_1s	bigint	OBS IO请求延迟大于1秒的次数。 单位：次
obs_io_req_latency_gt_10s	bigint	查询的OBS IO请求延迟大于10秒的次数。 单位：次
obs_io_req_count	bigint	OBS IO请求次数。 单位：次
obs_io_req_count	bigint	OBS IO请求重试次数。 单位：次
obs_io_req_rate_limit_count	bigint	OBS IO请求速率限制次数。 单位：次

pg_get_actor_stat()

描述：获取一些监控事项的统计信息，监控事项包括创建CN，启动CN，创建会话，语句加入队列，执行语句，写结果集，获取执行状态，获取执行结果。每发生一次上述事项，就记录一次统计信息。

返回值类型：

字段名称	类型	描述
event_name	text	监控事项名称。
object_name	text	标识监控事项ID。
start_time	timestampt with zone	监控事项开始时间。
end_time	timestampt with zone	监控事项结束时间。

字段名称	类型	描述
comment	text	具体内容。

8.3.21 ICEBERG 表服务函数

iceberg_expire_snapshots

描述：Iceberg每个commit都会生成一个新快照，同时保留旧数据和元数据，以便进行快照隔离和time travel。expire snapshots可以用来清理不再需要的旧快照以及仅被不需要快照包含的数据文件，以提高查询、成本效率。

注意事项：

- 要明确旧快照的清理、保留策略。执行旧快照清理后，被删除的旧快照数据将无法再通过常规查询访问，如果有正在运行的查询依赖于即将过期的快照数据，可能会导致查询失败或结果不准确。
- 执行expire snapshots，retain_last、older_than、snapshot_ids可能会出现条件互相冲突的场景，iceberg不是严格按照传入条件处理的，会尽可能多地保留snapshots。

语法：

```
iceberg_expire_snapshots('schema.table','older_than')
iceberg_expire_snapshots('schema.table','older_than', retain_last, 'snapshot_ids')
```

入参说明：

名称	类型	默认值	是否必填	说明
table	string	无	是	要操作的表
older_than	timestamp	5天前	是	超过该时限的快照将被删除 注意：需要输入确定具体时区，避免误解。可以在输入时指定，或者通过set timezone=xxx指定。
retain_last	int	1	否	保留多少个snapshots。
snapshot_ids	string	无	否	以分号（;）分隔, 可以传入多个snapshot_id。

返回值：

名称	说明
deleted_data_files_count	被删除的数据文件数量。

名称	说明
deleted_position_delete_files_count	被删除的position delete文件数量。
deleted_equality_delete_files_count	被删除的equality delete文件数量。
deleted_manifest_files_count	被删除的清单文件数量。
deleted_manifest_lists_count	被删除的快照文件数量。
deleted_statistics_files_count	被删除的统计文件数量。

iceberg_rewrite_manifests

描述：Iceberg表跟踪每个数据文件，数据文件越多，存储在快照文件中的元数据就越多，rewrite manifests可以重写表的清单，优化扫描规划。

注意事项：该操作会生成新的清单文件，可能影响正在进行的查询和写入操作，导致元数据冲突，应避免与其他操作并发执行。

语法：

```
iceberg_rewrite_manifests('schema.table')
```

入参说明：

名称	类型	默认值	是否必填	说明
table	string	无	是	要操作的表。

返回值：

名称	说明
rewritten_manifests_count	被重写的清单文件数量。
added_manifests_count	新增的清单文件数量。

iceberg_remove_orphanfiles

描述：对于未被任何Iceberg元数据文件引用的文件，我们称之为孤立文件，这类文件可能由失败的事务、并发操作未能成功提交、旧快照清理失败、元数据清理失败等原因造成，通常可以删除。

注意事项：确保无并发写操作，删除过程中，如果有运行中的写入任务，可能会误删正在生成的文件（未提交的临时文件）。

语法：

```
iceberg_remove_orphanfiles('schema.table')
iceberg_remove_orphanfiles('schema.table', 'older_than', 'location')
```

入参说明：

名称	类型	默认值	是否必填	说明
table	string	无	是	要操作的表
older_than	timestamp	三天前	否	早于older_than的孤立文件将被删除。
location	string	表根目录	否	location下的孤立文件将被删除。

返回值：

名称	说明
orphan_file_location	被认定为孤立文件的文件路径。

iceberg_add_files

描述：将目标文件夹下的所有文件添加到iceberg表中。

该语法会递归扫描目标文件夹下的所有文件，并添加到iceberg表中。

所有文件必须满足：

- 必须为parquet文件格式。
- parquet文件中的列名、列类型、列个数（包括分区列）必须与目标表一致。
- 单个parquet文件中的所有数据必须隶属于同一个分区。

语法：

```
iceberg_add_files(table,foldername,iscopy)
```

入参说明：

- table：regclass类型，表名。

- foldername：text类型，文件夹路径列表，支持多个路径，以逗号分隔。
- iscopy：bool类型，是否拷贝。

返回值：bool，是否执行成功。

iceberg_show_snapshots

描述：查询指定ICEBERG表的快照信息。

语法：

```
iceberg_show_snapshots(table)
```

入参说明：table：regclass类型，表名。

返回值：

- sequence_number：int64类型，快照序号。
- snapshot_id：int64类型，快照ID。
- create_time：timestamptz类型，快照创建时间。
- parent_id：int64类型，快照的父亲节点，如果为0，表示无父亲节点。
- operation：text类型，本次快照的操作类型。
- manifestlist：text类型，快照对应的manifest list路径。
- summary：text类型，快照统计信息。

8.3.22 UDF 配置函数

update_udf_runtime_config

描述：用户可以执行内置函数指定某些UDF对应的Python UDF actor配置参数，或者指定所有UDF执行时的配置参数，包括actor使用的CPU核数，内存占用量和并行度。

语法：

```
update_udf_runtime_config(array['schema.func:memory=xxx,cpu=xxx,concurrency=xxx','default:memory=xxx,
cpu=xxx,concurrency=xxx'])
```

入参说明：

- schema：UDF函数所在的数据库名。
- func：UDF函数名。
- cpu：Python UDF actor所使用的CPU核数，单位为毫核。
- memory：Python UDF actor所使用的内存占用量，单位为MB。
- concurrency：Python UDF actor的并行度。

返回值：无

注意事项：如果UDF指定配置，则优先使用该配置。如果未指定，则使用'default'对应的配置，如果均未指定，则使用固定配置'cpu=500, memory=500, concurrency=1'。

8.4 表达式

8.4.1 简单表达式

逻辑表达式

逻辑表达式的操作符和运算规则，请参见逻辑操作符。

比较表达式

常用的比较操作符，请参见比较操作符。

除比较操作符外，还可以使用以下句式结构：

- BETWEEN操作符
  a BETWEEN x AND y等效于a >= x AND a <= y
  a NOT BETWEEN x AND y等效于a < x OR a > y
- 检查一个值是不是null，可使用：
  expression IS NULL
  expression IS NOT NULL
  或者与之等价的句式结构，但不是标准的：
  expression ISNULL
  expression NOTNULL

> **须知**
>
> 不要写expression=NULL或expression<>(!=)NULL，因为NULL代表一个未知的值，不能通过该表达式判断两个未知值是否相等。

示例

```
SELECT 2 BETWEEN 1 AND 3 AS RESULT;
 result
----------
 t
(1 row)

SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
 result
----------
 t
(1 row)

SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
 result
----------
 f
(1 row)

SELECT 2 < 1 OR 2 > 3 AS RESULT;
 result
----------
 f
(1 row)

SELECT 2+2 IS NULL AS RESULT;
 result
```



```
----------
 f
(1 row)

SELECT 2+2 IS NOT NULL AS RESULT;
 result
----------
 t
(1 row)

SELECT 2+2 ISNULL AS RESULT;
 result
----------
 f
(1 row)

SELECT 2+2 NOTNULL AS RESULT;
 result
----------
 t
(1 row)

SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;
 result
----------
 t
(1 row)

SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;
 result
----------
 f
(1 row)
```

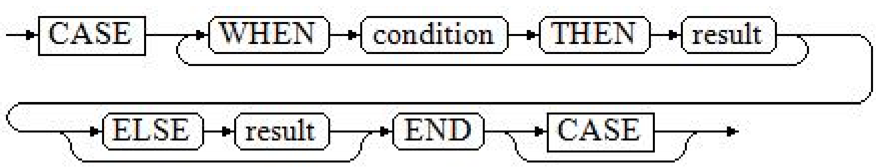
8.4.2 条件表达式

在执行SQL语句时，可通过条件表达式筛选出符合条件的数据。

条件表达式主要有以下几种：

- CASE
  CASE表达式是条件表达式，类似于其他编程语言中的CASE语句。
  CASE表达式的语法图请参考**图8-1**。

  图 8-1 case::=

  CASE子句可以用于合法的表达式中。condition是一个返回BOOLEAN数据类型的表达式：
  – 如果结果为真，CASE表达式的结果就是符合该条件所对应的result。
  – 如果结果为假，则以相同方式处理随后的WHEN或ELSE子句。
  – 如果各WHEN condition都不为真，表达式的结果就是在ELSE子句执行的result。如果省略了ELSE子句且没有匹配的条件，结果为NULL。

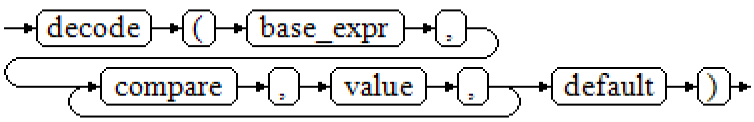
  示例：

```
CREATE TABLE case_when_t1(CW_COL1 INT) store AS orc;

INSERT INTO case_when_t1 VALUES (1), (2), (3);

SELECT * FROM case_when_t1;SELECT * FROM case_when_t1 order by cw_col1;
 cw_col1
---------
       1
       2
       3
(3 rows)

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;DROP TABLE case_when_t1;
```

- DECODE

  DECODE的语法图请参见**图8-2**。

  图 8-2 decode::=

  将表达式base_expr与后面的每个compare(n) 进行比较，如果匹配返回相应的value(n)。如果没有发生匹配，则返回default。

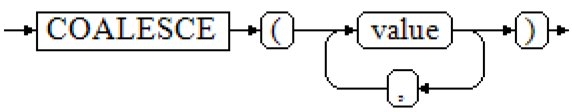
  示例请参见**条件表达式函数**。

```
SELECT DECODE('A','A',1,'B',2,0);
 case
------
    1
(1 row)
```

- COALESCE

  COALESCE的语法图请参见**图8-3**。

  图 8-3 coalesce::=

  COALESCE返回它的第一个非NULL的参数值。如果参数都为NULL，则返回NULL。它常用于在显示数据时用缺省值替换NULL。和CASE表达式一样，COALESCE只计算用来判断结果的参数，即在第一个非空参数右边的参数不会被计算。

  示例：

```
CREATE TABLE c_tabl(description varchar(10), short_description varchar(10), last_value varchar(10))
store AS orc;

INSERT INTO c_tabl VALUES('abc', 'efg', '123');
INSERT INTO c_tabl VALUES(NULL, 'efg', '123');

INSERT INTO c_tabl VALUES(NULL, NULL, '123');

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)

DROP TABLE c_tabl;
```

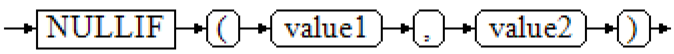
如果description不为NULL，则返回description的值，否则计算下一个参数short_description；如果short_description不为NULL，则返回short_description的值，否则计算下一个参数last_value；如果last_value不为NULL，则返回last_value的值，否则返回（none）。

```
SELECT COALESCE(NULL,'Hello World');
   coalesce
---------------
 Hello World
(1 row)
```

- NULLIF

  NULLIF的语法图请参见图8-4。

图 8-4 nullif::=



只有当value1和value2相等时，NULLIF才返回NULL。否则它返回value1。

示例：

```
CREATE TABLE null_if_t1 (
    NI_VALUE1 VARCHAR(10),
    NI_VALUE2 VARCHAR(10)
) store AS orc;

INSERT INTO null_if_t1 VALUES('abc', 'abc');
INSERT INTO null_if_t1 VALUES('abc', 'efg');

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)
DROP TABLE null_if_t1;
```

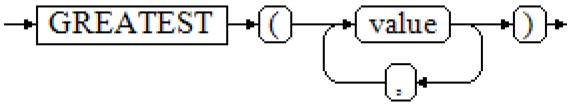
如果value1等于value2则返回NULL，否则返回value1。

```
SELECT NULLIF('Hello','Hello World');
 nullif
--------
 Hello
(1 row)
```

- GREATEST（最大值），LEAST（最小值）

  GREATEST的语法图请参见图8-5。

  图 8-5 greatest::=

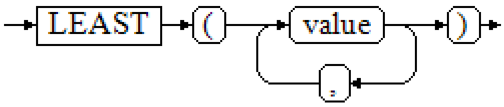


  从一个任意数字表达式的列表里选取最大的数值。

```
SELECT greatest(9000,155555,2.01);
 greatest
----------
   155555
(1 row)
```

  LEAST的语法图请参见图8-6。

  图 8-6 least::=



  从一个任意数字表达式的列表里选取最小的数值。

  以上的数字表达式必须都可以转换成一个普通的数据类型，该数据类型将是结果类型。

  列表中的NULL值将被忽略。只有所有表达式的结果都是NULL的时候，结果才是NULL。

```
SELECT least(9000,2);
 least
-------
     2
(1 row)
```

  示例请参见条件表达式函数。

- NVL

  NVL的语法图请参见图8-7。

  图 8-7 nvl::=

  如果value1为NULL则返回value2，如果value1非NULL，则返回value1。

  示例：

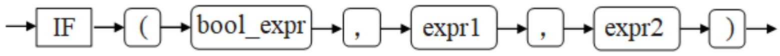
```
SELECT nvl(null,1);
nvl
-----
```

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

- IF

  IF的语法图请参见图8-8。

  图 8-8 if::=

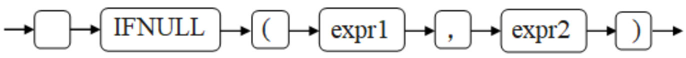
  当bool_expr为true时，返回expr1，否则返回expr2。

  示例请参见条件表达式函数。

- IFNULL

  IFNULL的语法图请参见图8-9。

  图 8-9 ifnull::=

  当expr1不为NULL时，返回expr1，否则返回expr2。

  示例请参见条件表达式函数。

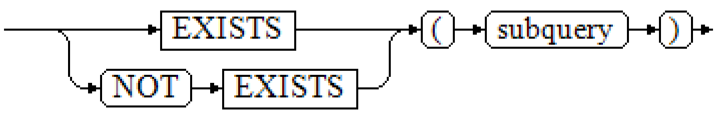
8.4.3 子查询表达式

子查询表达式主要有以下几种：

- EXISTS/NOT EXISTS

  EXISTS/NOT EXISTS的语法图请参见图8-10。

  图 8-10 EXISTS/NOT EXISTS::=

  EXISTS的参数是一个任意的SELECT语句，或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，则EXISTS结果就为"真"；如果子查询没有返回任何行， EXISTS的结果是"假"。

这个子查询通常只是运行到能判断它是否可以生成至少一行为止，而不是等到全部结束。

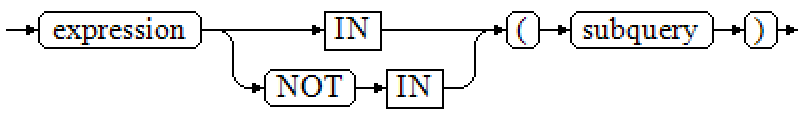
示例：

```
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

  IN/NOT IN的语法请参见图8-11。

图 8-11 IN/NOT IN::=



右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式对子查询结果的每一行进行一次计算和比较。如果找到任何相等的子查询行，则IN结果为"真"。如果没有找到任何相等行，则结果为"假"（包括子查询没有返回任何行的情况）。

表达式或子查询行里的NULL遵照SQL处理布尔值和NULL组合时的规则。如果两个行对应的字段都相等且非空，则这两行相等；如果任意对应字段不等且非空，则这两行不等；否则结果是未知（NULL）。如果每一行的结果都是不等或NULL ，并且至少有一个NULL ，则IN的结果是NULL 。

示例：

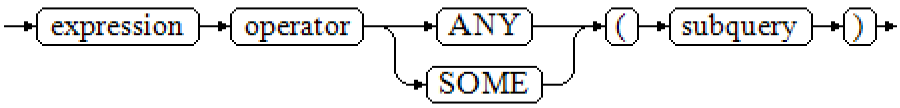
```
 SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk IN (SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          10 |              3
          26 |              3
          22 |              5
          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

  ANY/SOME的语法图请参见图8-12。

  图 8-12 any/some::=

  右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果至少获得一个真值，则ANY结果为“真”。如果全部获得假值，则结果是“假”（包括子查询没有返回任何行的情况）。SOME是ANY的同义词。IN与ANY可以等效替换 。

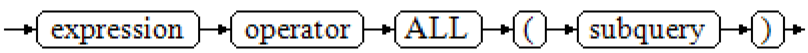
  示例：

```
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

  ALL的语法请参见图8-13。

  图 8-13 all::=

  右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如

果全部获得真值，ALL结果为"真"（包括子查询没有返回任何行的情况）。如果至少获得一个假值，则结果是"假"。

示例：

```
SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk < all(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
 sr_reason_sk | sr_customer_sk
--------------+----------------
(0 rows)
```

8.4.4 数组表达式

IN

expression **IN** *(value [, ...])*

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果列表中的内容符合左侧表达式的结果，则IN的结果为true。如果没有相符的结果，则IN的结果为false。

示例如下：

```
SELECT 8000+500 IN (10000, 9000) AS RESULT;
  result
----------
 f
(1 row)
```

说明

如果表达式结果为null，或者表达式列表不符合表达式的条件且右侧表达式列表返回结果至少一处为空，则IN的返回结果为null，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

NOT IN

expression **NOT IN** *(value [, ...])*

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果在列表中的内容没有符合左侧表达式结果的内容，则NOT IN的结果为true。如果有符合的内容，则NOT IN的结果为false。

示例如下：

```
SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
  result
----------
 t
(1 row)
```

说明

如果查询语句返回结果为空，或者表达式列表不符合表达式的条件且右侧表达式列表返回结果至少一处为空，则NOT IN的返回结果为null，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

提示：在所有情况下X NOT IN Y等价于NOT(X IN Y)。

ANY/SOME（array）

expression operator **ANY** *(array expression)*

expression operator **SOME** *(array expression)*

```
SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

- 如果对比结果至少获取一个真值，则ANY的结果为true。
- 如果对比结果没有真值，则ANY的结果为false。

说明

如果结果没有真值，并且数组表达式生成至少一个值为null，则ANY的值为NULL，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

SOME是ANY的同义词。

ALL (array)

expression operator **ALL** *(array expression)*

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

- 如果所有的比较结果都为真值（包括数组不含任何元素的情况），则ALL的结果为true。
- 如果结果有任一结果是false，则ALL的结果为false。

如果数组表达式产生一个NULL数组，则ALL的结果为NULL。如果左边表达式的值为NULL ，则ALL的结果通常也为NULL(某些不严格的比较操作符可能得到不同的结果)。另外，如果右边的数组表达式中包含null元素并且比较结果没有假值，则ALL的结果将是NULL(某些不严格的比较操作符可能得到不同的结果)， 而不是真。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

```
SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

8.4.5 行表达式

语法：

row_constructor operator row_constructor

两边都是一个行构造器，两行值必须具有相同数目的字段，每一行都进行比较，行比较允许使用=，<>，<，<=，>=等操作符，或其中一个相似的语义符。

=<>和别的操作符使用略有不同。如果两行值的所有字段都是非空并且相等，则认为两行是相等的；如果两行值的任意字段为非空并且不相等，则认为两行是不相等的；否则比较结果是未知的（null）。

对于<，<=，>，> =的情况下，行中元素从左到右依次比较，直到遇到一对不相等的元素或者一对为空的元素。如果这对元素中存在至少一个null值，则比较结果是未知的（null），否则这对元素的比较结果为最终的结果。

示例：

```
SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;
 result
----------
 t
(1 row)
```

8.5 类型转换

8.5.1 概述

背景信息

在SQL语言中，每个数据都与一个决定其行为和用法的数据类型相关。DataArtsFabric SQL提供一个可扩展的数据类型系统，该系统比其它SQL实现更具通用性和灵活性。因而，DataArtsFabric SQL中大多数类型转换是由通用规则来管理的，这种做法允许使用混合类型的表达式。

DataArtsFabric SQL扫描/分析器只将词法元素分解成五个基本种类：整数、浮点数、字符串、标识符和关键字。大多数非数字类型首先表现为字符串。SQL语言的定义允许将常量字符串声明为具体的类型。例，下面查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

示例中有两个文本常量，类型分别为text和point。如果没有为字符串文本声明类型，则该文本首先被定义成一个unknown类型。

在DataArtsFabric SQL分析器里，有四种基本的SQL结构需要独立的类型转换规则：

- 函数调用

  多数SQL类型系统是建筑在一套丰富的函数上的。函数调用可以有一个或多个参数。因为SQL允许函数重载，所以不能通过函数名直接找到要调用的函数，分析器必须根据函数提供的参数类型选择正确的函数。

- 操作符

  SQL允许在表达式上使用前缀或后缀（单目）操作符，也允许表达式内部使用双目操作符（两个参数）。像函数一样，操作符也可以被重载，因此操作符的选择也和函数一样取决于参数类型。

- 值存储

  INSERT和UPDATE语句将表达式结果存入表中。语句中的表达式类型必须和目标字段的类型一致或者可以转换为一致。

- UNION，CASE和相关构造

  因为联合SELECT语句中的所有查询结果必须在一列里显示出来，所以每个SELECT子句中的元素类型必须相互匹配并转换成一个统一类型。类似地，一个CASE构造的结果表达式必须转换成统一的类型，这样整个case表达式会有一个统一的输出类型。同样的要求也存在于ARRAY构造以及GREATEST和LEAST函数中。

系统表pg_cast存储了有关数据类型之间的转换关系以及如何执行这些转换的信息。详细信息请参见PG_CAST。

语义分析阶段会决定表达式的返回值类型并选择适当的转换行为。数据类型的基本类型分类，包括：boolean、numeric、string、bitstring、datetime、timespan、geometric和network。每种类型都有一种或多种首选类型用于解决类型选择的问题。根据首选类型和可用的隐含转换，就可能保证有歧义的表达式（那些有多个候选解析方案的）得到有效的方式解决。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐含转换决不能有奇怪的或不可预见的输出。
- 如果一个查询不需要隐含的类型转换，分析器和执行器不应该进行更多的额外操作。这就是说，任何一个类型匹配、格式清晰的查询不应该在分析器里耗费更多的时间，也不应该向查询中引入任何不必要的隐含类型转换调用。
- 另外，如果一个查询在调用某个函数时需要进行隐式转换，当用户定义了一个有正确参数的函数后，解释器应该选择使用新函数。

8.5.2 操作符

操作符类型解析

1. 从系统表pg_operator中选出要考虑的操作符。如果可以找到一个参数类型以及参数个数都一致的操作符，那么这个操作符就是最终使用的操作符。如果找到了多个备选的操作符，将从中选择一个最合适的。
2. 寻找最优匹配。
   a. 丢弃输入类型不匹配以及无法隐式转换成匹配的候选操作符。unknown文本在这种情况下可以转换成任何类型。如果只剩下一个候选操作符，则使用，否则继续下一步。
   b. 查看所有候选操作符，并保留输入类型最匹配的操作符。此时，域被看作和其基本类型相同。如果没有完全匹配的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。
   c. 查看所有候选操作符，保留需要类型转换时接受(属于输入数据类型的类型范畴的)首选类型位置最多的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。
   d. 如果有任何输入参数是unknown类型，请检查其余候选操作符对应参数位置的类型范畴。在每一个能够接受string类型范畴的位置使用string类型（这种偏向字符串的做法合理，因为unknown文本跟字符串相似）。另外，如果所有剩下的候选操作符都接受相同的类型范畴，则选择该类型范畴，否则会报错（因为在没有更多线索的条件下无法做出正确的选择）。现在丢弃不接受选定类型范畴的候选操作符。此外，如果有任意候选操作符接受该范畴中的首选类型，则丢弃该参数接受非首选类型的候选操作符。如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。
   e. 如果同时有unknown和已知参数，并且所有已知参数都是相同的类型，那么假设unknown参数也属于该类型，并检查哪些候选操作符在unknown参数位置接受该类型。如果只有一个操作符符合，则使用。否则，报错。

示例

示例1：阶乘操作符类型解析。在系统表中里只有一个阶乘操作符（后缀!），它以bigint作为参数。扫描器给下面查询表达式的参数赋予bigint的初始类型：

```
SELECT 40 ! AS "40 factorial";

                   40 factorial
--------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

分析器对参数做类型转换，查询等效于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

示例2：字符串连接操作符类型分析。一种字符串风格的语法既可以用于字符串也可以用于复杂的扩展类型。未声明类型的字符串将被所有可能的候选操作符匹配。有一个未声明的参数的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";
 text and unknown
------------------
 abcdef
(1 row)
```

本例中分析器寻找两个参数都是text的操作符。确实有这样的操作符，两个参数都是text类型。

下面是连接两个未声明类型的值：

```
SELECT 'abc' || 'def' AS "unspecified";
 unspecified
-------------
 abcdef
(1 row)
```

说明

因为查询中没有声明任何类型，所以本例中对类型没有任何初始提示。因此，分析器查找所有候选操作符，发现既存在接受字符串类型范畴的操作符也存在接受位串类型范畴的操作符。因为字符串类型范畴是首选，所以选择字符串类型范畴的首选类型text作为解析未知类型文本的声明类型。

示例3：绝对值和取反操作符类型分析。DataArtsFabric SQL操作符表里面有几条记录对应于前缀操作符@，它们都用于为各种数值类型实现绝对值操作。其中之一用于float8类型，它是数值类型范畴中的首选类型。因此，在面对unknown输入的时候，DataArtsFabric SQL会使用该类型：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

此处，系统在应用选定的操作符之前隐式的转换unknown类型的文字为float8类型。

示例4：数组包含操作符类型分析。以操作符两侧带有一个已知类型和一个未知类型的情况为例：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";
 is subset
-----------
 t
(1 row)
```

说明

DataArtsFabric SQL的pg_operator表中有多条记录对应于中缀操作符<@，但是只有“数组包含(anyarray <@ anyarray)”和“范围包含(anyelement <@ anyrange)”可在左侧接受一个整数数组。因为多态伪类型不被认为是首选操作符类型，因此解析器无法在此基础上解决歧义。不过，寻找最优匹配的最后一条解析规则又指出，假定未知类型的文字和另一个已知的输入类型相同，也就是只有两个运算符之一可以匹配，所以选择数组包含。（如果选择了范围包含，就会报错，因为字符串格式不正确不能作为范围。）

8.5.3 函数

函数类型解析

1. 从系统表pg_proc中选择所有可能被选到的函数。如果使用了一个不带模式修饰的函数名字，那么认为该函数是那些在当前搜索路径中的函数。如果给出一个带修饰的函数名，那么只考虑指定模式中的函数。

   如果搜索路径中找到了多个不同参数类型的函数。将从中选择一个合适的函数。
2. 查找和输入参数类型完全匹配的函数。如果找到一个，则用之。如果输入的实参类型都是unknown类型，则不会找到匹配的函数。
3. 如果未找到完全匹配，请查看该函数是否为一个特殊的类型转换函数。
4. 寻找最优匹配。
   a. 抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选函数。unknown文本在这种情况下可以转换成任何东西。如果只剩下一个候选项，则用之，否则继续下一步。
   b. 遍历所有候选函数，保留那些输入类型匹配最准确的。此时，域被看作和它们的基本类型相同。如果没有一个函数能准确匹配，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。
   c. 遍历所有候选函数，保留那些需要类型转换时接受首选类型位置最多的函数。如果没有接受首选类型的函数，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。
   d. 如果有任何输入参数是unknown类型，检查剩余的候选函数对应参数位置的类型范畴。在每一个能够接受字符串类型范畴的位置使用string类型（这种对字符串的偏爱是合适的，因为unknown文本确实像字符串）。另外，如果所有剩下的候选函数都接受相同的类型范畴，则选择该类型范畴，否则抛出一个错误（因为在没有更多线索的条件下无法做出正确的选择）。现在抛弃不接受选定的类型范畴的候选函数，然后，如果任意候选函数在那个范畴接受一个首选类型，则抛弃那些在该参数位置接受非首选类型的候选函数。如果没有一个候选符合这些测试则保留所有候选。如果只有一个候选函数符合，则使用它；否则，继续下一步。
   e. 如果同时有unknown和已知类型的参数，并且所有已知类型的参数有相同的类型，假设unknown参数也是这种类型，检查哪个候选函数可以在unknown参数位置接受这种类型。如果正好一个候选符合，那么使用它。否则，产生一个错误。

示例

示例1：圆整函数参数类型解析。只有一个round函数有两个参数（第一个是numeric，第二个是integer）。所以下面的查询自动把第一个类型为integer的参数转换成numeric类型。

```
SELECT round(4, 4);
 round
```

```
--------
 4.0000
(1 row)
```

实际上它被分析器转换成：

```
SELECT round(CAST (4 AS numeric), 4);
```

因为带小数点的数值常量初始时被赋予numeric类型，因此下面的查询将不需要类型转换，并且可能会略微高效一些：

```
SELECT round(4.0, 4);
```

示例2：子字符串函数类型解析。有好几个substr函数，其中一个接受text和integer类型。如果用一个未声明类型的字符串常量调用它，系统将选择接受string类型范畴的首选类型（也就是text类型）的候选函数。

```
SELECT substr('1234', 3);
 substr
--------
     34
(1 row)
```

如果该字符串声明为varchar类型，就像从表中取出来的数据一样，分析器将试着将其转换成text类型：

```
SELECT substr(varchar '1234', 3);
 substr
--------
     34
(1 row)
```

被分析器转换后实际上变成：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

说明

分析器从pg_cast表中了解到text和varchar是二进制兼容的，意思是说一个可以传递给接受另一个的函数而不需要做任何物理转换。因此，在这种情况下，实际上没有做任何类型转换。

而且，如果以integer为参数调用函数，分析器将试图将其转换成text类型：

```
SELECT substr(1234, 3);
substr
--------
 34
(1 row)
```

被分析器转换后实际上变成：

```
SELECT substr(CAST (1234 AS text), 3);
 substr
--------
     34
(1 row)
```

8.5.4 值存储

值存储数据类型解析

1. 查找与目标字段准确的匹配。
2. 试着将表达式直接转换成目标类型。如果已知这两种类型之间存在一个已登记的转换函数，那么直接调用该转换函数即可。如果表达式是一个未知类型文本，该文本字符串的内容将交给目标类型的输入转换过程。

3. 检查目标类型是否有长度转换。长度转换是一个从某类型到自身的转换。如果在pg_cast表里面找到一个，那么在存储到目标字段之前先在表达式上应用。这样的转换函数总是接受一个额外的类型为integer的参数，它接收目标字段的atttypmod值（实际上是其声明长度，atttypmod的解释随不同的数据类型而不同），并且它可能接受一个boolean类型的第三个参数，表示转换是显式的还是隐式的。转换函数负责施加那些长度相关的语义，例如长度检查或者截断。

示例

character存储类型转换。对一个目标列定义为character(20)的语句，下面的语句显示存储值的长度正确：

```
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;
```

说明

这里真正发生的事情是两个unknown文本缺省解析成text，这样就允许||操作符解析成text连接。然后操作符的text结果转换成bpchar("空白填充的字符型"， character类型内部名称)以匹配目标字段类型。不过，从text到bpchar的转换是二进制兼容的，这样的转换是隐含的并且实际上不做任何函数调用。最后，在系统表里找到长度转换函数bpchar(bpchar, integer, boolean)并且应用于该操作符的结果和存储的字段长。这个类型相关的函数执行所需的长度检查和额外的空白填充。

8.5.5 UNION，CASE 和相关构造

SQL UNION构造把不相同的数据类型进行匹配输出为统一的数据类型结果集。因为SELECT UNION语句中的所有查询结果必须在一列里显示出来，所以每个SELECT子句中的元素类型必须相互匹配并转换成一个统一的数据类型。类似地，一个CASE构造的结果表达式必须转换成统一的类型，这样整个case表达式会有一个统一的输出类型。同样的要求也存在于ARRAY构造以及GREATEST和LEAST函数中。

UNION，CASE 和相关构造解析

- 如果所有输入都是相同的数据类型，不包括unknown类型（即输入的字符串文本未声明类型，该文本首先被定义成一个未知类型），那么解析成所输入的相同数据类型。
- 如果所有输入都是unknown类型则解析成text类型（字符串类型范畴的首选类型）。否则，忽略unknown输入。
- 如果输入不属于同一个类型范畴，查询失败（unknown类型除外）。
- 如果输入类型是同一个类型范畴，则选择该类型范畴的首选类型（union操作会选择第一个分支的类型作为所选类型的情况除外）。

📖 说明

系统表pg_type中typcategory表示数据类型范畴， typispreferred表示是否是typcategory分类中的首选类型。

- 把所有输入转换为所选的类型（对于字符串保持原有长度）。如果从给定的输入到所选的类型没有隐式转换则失败。
- 如果输入中含json、txid_snapshot、sys_refcursor或几何类型，则不能进行union。

8.6 DDL 语法

8.6.1 SHOW

8.6.1.1 SHOW Conf

功能描述

SHOW将显示当前运行时参数的数值。可以使用SET语句来设置这些参数。

注意事项

SHOW可以查看的某些参数是只读的，可以查看但不能设置它们的值。

语法格式

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

   SESSION AUTHORIZATION |
   ALL
 };
```

参数说明

- **configuration_parameter**

  运行时参数的名称。

  取值范围：可以使用SHOW ALL命令查看运行时参数。

  📖 说明

  部分通过SHOW ALL查看的参数不能通过SET设置。如max_datanodes。

- **CURRENT_SCHEMA**

  当前模式。

- **TIME ZONE**

  时区。

- **SESSION AUTHORIZATION**

  当前会话的用户标识符。

- **ALL**
  所有运行时参数。

示例

显示timezone参数值：

```
SHOW timezone;
```

显示参数DateStyle的当前设置：

```
SHOW DateStyle;
```

显示所有参数的当前设置：

```
SHOW ALL;
```

8.6.1.2 SHOW Schemas/Tables/Views/Partitions/Functions

功能描述

SHOW为DataArtsFabric SQL服务下特有语法，该语法功能是列举LakeFormation上指定对象下的子对象信息。

注意事项

无。

语法格式

```
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;
```

参数说明

无。

示例

列举当前Catalog下的所有数据库。

```
SHOW SCHEMAS;
```

列举LakeFormation上test_schema数据库下面的所有表。

```
SHOW TABLES IN test_schema;
```

列举LakeFormation上test_table表的所有分区。

```
SHOW PARTITIONS test_table;
```

列举LakeFormation上当前数据库下面的所有视图。

```
SHOW VIEWS;
```

列举LakeFormation上test_schema数据库下面的所有视图。

```
SHOW VIEWS IN test_schema;
```

列举LakeFormation上当前数据库下面的所有自定义函数。

```
SHOW FUNCTIONS;
```

列举LakeFormation上test_schema数据库下面的所有函数。

```
SHOW FUNCTIONS IN test_schema;
```

8.6.2 CREATE SCHEMA

功能描述

CREATE SCHEMA为DataArtsFabric SQL服务下特有语法，该语法功能是在LakeFormation上创建指定名称的Database。

注意事项

DataArtsFabric SQL没有集群概念，不会存储任何元数据，所有元数据均存放于LakeFormation，因此用户在创建External Schema后，可以登录LakeFormation界面，查看创建的元数据。

语法格式

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

参数说明

- **schema_name**
  目标表的数据库名称。
- **WITH LOCATION path**
  可选参数，指定数据库存储目录。当用户未指定数据库目录时，则系统将自动在Catalog目录下，以同名数据库目录作为该数据库的存储目录。
- **path**
  数据库文件的存储路径，创建SCHEMA时要求目标路径内无其他文件。

示例

创建一个数据库目录：

```
CREATE SCHEMA test_schema;
```

创建一个数据库目录，并为该数据库指定存储目录：

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

8.6.3 DROP SCHEMA

功能描述

DROP SCHEMA为DataArtsFabric SQL服务下特有语法，该语法功能是在LakeFormation上删除指定名称的Database。

注意事项

在删除Schema数据时，如果用户数据量很大，可能会导致语句执行时间过长。

语法格式

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

参数说明

- **IF EXISTS**
  如果指定的模式不存在，发出一个notice而不是抛出一个错误。
- **schema_name**
  LakeFormation的数据库名字。
- **CASCADE | RESTRICT**
  - CASCADE：递归删除LakeFormation上该数据库对象下的所有对象。
  - RESTRICT：仅删除该数据库对象，如果数据库对象下包含任何对象，则删除失败（缺省行为）。

示例

删除数据库test_schema，如果数据库下有级联对象则会抛出错误。

```
DROP SCHEMA test_schema;
```

级联删除数据库test_schema以及该数据库下的所有对象。

```
DROP SCHEMA test_schema CASCADE;
```

8.6.4 CREATE TABLE

功能描述

CREATE TABLE为DataArtsFabric SQL服务下特有语法，该语法功能是在LakeFormation上创建指定名称的表。

关于表达式分区表的详细说明，请参考表达式分区。

注意事项

- PARTITION BY中出现的列不能出现在表的普通列描述中，分区列始终排在普通列的后面。
- PARTITION BY中支持的最大分区键数目是4。
- CLUSTERED BY语法仅适用于ORC、PARQUET表，不适用于Iceberg表。
- 不支持用户将ctid、xc_node_id作为列名。
- 当前实现的表达式分区的主要约束，请参考约束限制。

语法格式

```
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;
```

参数说明

- **EXTERNAL**

  如果指定该关键字，则创建EXTERNAL表；如果无该关键字，则默认创建MANAGED表，文件，元数据和统计信息都由DataArtsFabric SQL管理。

- **IF NOT EXISTS**

  如果存在同名表，则发出一个notice而不是抛出一个错误。

- **schema_name**

  表所属的数据库名，如果未指定数据库名时，则将在current_schema下建表。

- **table_name**

  创建表的表名，表名长度不可超过63个字符。

- **col_name**

  创建表的列名，列名长度不可超过63个字符。普通列和分区列的数量总和不可超过5000个。

- **col_type**

  创建表的列类型，列类型支持范围如下：

列类型	是否可以声明为分区列	是否支持ORC格式	是否支持PARQUET格式	是否支持Iceberg格式
SmallInt	√	√	√	√
Int	√	√	√	√
BigInt	√	√	√	√
Float	×	√	√	√
Double	×	√	√	√
Decimal	√	√	√	√
Numeric	√	√	√	√
Timestamp	√	√	√	√
Date	√	√	√	√
Varchar	√	√	√	√
Char	√	√	√	√
Bool	×	√	√	√
Bytea	×	√	√	√

列类型	是否可以声明为分区列	是否支持ORC格式	是否支持PARQUET格式	是否支持Iceberg格式
Text	√	√	√	√
Array	×	√	√	√
Struct	×	√	√	√

- **col_comment**
  列注释信息，可指定为任意字符串。
- **table_comment**
  表注释信息，可指定为任意字符串。
- **bucket_num**
  bucket个数。
- **option_key = option_value**
  表级别参数设置，支持参数范围如下：

表 8-35 option_key 参数支持范围

option_key	option_value	说明	适用范围
orc.compress	zlib, snappy, lz4, uncompressed	ORC文件压缩方式。	ORC
parquet.compression	zlib, snappy, lz4, uncompressed	PARQUET文件压缩方式。	PARQUET
julian_adjust	true, false	是否转换为Julian日期。	PARQUET
checkencoding	high, low, no	是否检查字符编码。	ORC、PARQUET
column_index_access	true, false	读取时表定义列和文件列匹配方式，默认true为列索引匹配，false为列名匹配。	ORC、PARQUET
filesize	1~1024的整数	生成外表文件大小。	ORC、PARQUET
write.delete.mode	copy-on-write, merge-on-read	设置delete时的模式：cow或mor。	Iceberg
write.update.mode	copy-on-write, merge-on-read	设置update时的模式：cow或mor。	

option_key	option_value	说明	适用范围
write.merge.mode	copy-on-write, merge-on-read	设置merge时的模式：cow或mor。	
write.parquet.compression-codec	zstd, lz4, snappy, gzip,uncompressed	parquet文件的压缩方式。	
write.merge.isolation-level	snapshot, serializable	merge命令的隔离级别。	
write.metadata.delete-after-commit.enabled	true, false	控制提交后是否删除最旧的跟踪版本元数据文件。	
write.update.isolation-level	snapshot, serializable	update命令的隔离级别。	
write.delete.isolation-level	snapshot, serializable	delete命令的隔离级别。	
write.metadata.previous-versions-max	大于0的整数	要保留的旧元数据文件的数量。	
write.parquet.bloom-filter-enabled.column.col1	true, false	col1列是否写入bloom filter索引。	

- **table_format**

  表存储格式，支持ORC、PARQUET、ICEBERG三种存储格式。
- **table_path**

  表存储路径，必须为合法OBS路径，支持OBS对象桶和并行文件系统。如果该路径为OBS对象桶路径，则该表只读，否则该表支持读写。如果创建表类型为托管表（Managed Table），则不允许指定表存储路径，表存储路径由系统指定为Schema路径下与表名同名路径，且要求该路径在建表时为空。
- **select_stmt**

  查询语句。外部表（External Table）不支持CTAS语句。

示例

创建ORC表：

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

创建iceberg表达式分区表示例，请参见创建表达式分区表。

8.6.5 CREATE VIEW

功能描述

CREATE VIEW语法功能是在LakeFormation上创建指定名称的视图。

语法格式

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

参数说明

- **OR REPLACE**
  如果存在同名视图，则将会创建新视图并覆盖老视图的元数据。
- **schema_name**
  视图所属的数据库名，如果未指定数据库名时，则将在current_schema下创建视图。
- **view_name**
  待创建的视图名，名称长度不可超过63个字符。
- **view_comment**
  视图注释信息，可指定为任意字符串。
- **select_stmt**
  查询语句。

示例

创建基于orc表的视图：

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

8.6.6 CREATE FUNCTION

功能描述

创建一个自定义函数。

语法格式

创建自定义函数语法。

```
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' }
    HANDLER = 'function_name'
    COMMENT = 'comment'
    [ STRICT ]
    [ PACKAGES = ( 'package_name==[version]' [ , ...] ) ]
```

```
[ IMPORTS = ( 'obs_file_path' ) ]
[ AS 'definition' ]
```

参数说明

表 8-36 CREATE FUNCTION 参数说明

参数	描述	取值范围
function_name	要创建的函数名字（可以用模式修饰）。	字符串，需符合标识符的命名规范。 创建函数时，建议指定schema，否则会在默认的schema default_db下创建函数。调用自定义函数时必须指定schema，否则系统会调用系统内置函数。
argname	函数参数的名字。	字符串，需符合标识符的命名规范。
argtype	函数参数的类型。	详情请参见**数据类型映射**。
rettype	函数返回值的数据类型。	详情请参见**数据类型映射**。
column_name	字段名称。	-
column_type	字段类型。	-
LANGUAGE lang_name	用于实现函数的语言的名字。自定义函数仅支持Python。	-
AGGREGATE	表示function是否为聚合函数。	-
IMMUTABLE	表示该函数在给出同样的参数值时总是返回同样的结果。	如果函数的入参是常量，会在优化器阶段计算该函数的值。优势是可以尽早获取表达式的值，从而能更准确地进行代价估算，生成的执行计划也更优。
STABLE	表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同SQL语句之间变化。	-
VOLATILE	表示该函数值可以在一次表扫描内改变，因此不会做任何优化。	-
PACKAGES	表示该函数运行时环境所依赖的Python三方包。	-

参数	描述	取值范围
RUNTIME_VERSION	表示函数运行时环境的Python的具体版本。当前仅支持指定为3.11系列。	-
HANDLER	表示函数的主函数入口。	-
IMPORTS	表示函数运行时所依赖在OBS的压缩包路径	IMPORTS子句中只能引入一个路径，即一个压缩包。
STRICT	STRICT用于指定如果函数的某个参数是NULL，此函数总是返回NULL。如果声明了这个参数，当有NULL值参数时该函数不会被执行；而只是自动返回一个NULL结果。	-
definition	函数体的具体实现。	默认为null，其长度不能超过1000字符。
comment	对函数的描述信息。	-

8.6.7 DROP TABLE

功能描述

DROP TABLE为DataArtsFabric SQL服务下语法，该语法功能是删除LakeFormation上指定名称的表。

注意事项

在删除管控表（MANAGED TABLE）时，会同时删除表元数据、统计信息和数据，如果用户数据量很大，可能会导致语句执行时间过长；删除外表（EXTERNAL TABLE）时，仅删除表的元数据及统计信息，不会删除数据

语法格式

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

参数说明

- **IF EXISTS**
  如果指定的表不存在，发出一个notice而不是抛出一个错误。
- **table_name**
  LakeFormation的表名字。

示例

删除表test_table：

```
DROP TABLE test_table;
```

8.6.8 DROP VIEW

功能描述

DROP VIEW的语法功能是删除LakeFormation上指定名称的视图。

语法格式

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

参数说明

- **IF EXISTS**
  如果指定的视图不存在，则发出一个提示而不会报错。
- **view_name**
  LakeFormation的视图名字。

示例

删除视图test_view，但不会删除视图所依赖的表：

```
DROP VIEW test_view;
```

8.6.9 DROP FUNCTION

功能描述

删除一个已存在的函数。

注意事项

- 无法删除内置函数，只支持删除自定义函数。

语法格式

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

参数说明

表 8-37 DROP FUNCTION 参数说明

参数	描述	取值范围
IF EXISTS	如果指定的外表不存在，则发出一个notice而不是抛出一个错误。	-
function_name	要删除的函数名字。	已存在的函数名。

示例

删除一个自定义函数

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

8.6.10 ALTER TABLE

功能描述

ALTER TABLE为DataArtsFabric SQL服务下特有语法，该语法功能是修改LakeFormation上表的元数据信息，用于调整数据表的结构或属性。

约束限制

- ADD COLUMNS、DROP COLUMNS、COLUMN RENAME、ALTER COLUMN语法仅对ICEBERG表生效，ORC、PARQUET表无法对列进行操作。
- 列位置操作、复杂类型字段操作仅对ICEBERG表生效。
- 外表（EXTERNAL TABLE）不支持ALTER TABLE。

语法格式

```
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;
```

参数说明

该ALTER TABLE语法中，参数规格与建表规格一致。

示例

在表test_table中添加sales bigint列：

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

将表test_table中sales列更名为my_sales：

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

将表test_table中my_sales列类型更换为String：

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

将表test_table中my_sales列删除：

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

将表test_table更名为factory_info：

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

将表factory_info中参数hoodie.metadata.enable打开：

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

将表factory_info中参数恢复默认值：

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

同步表factory_info分区信息：

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

8.6.11 DESCRIBE

功能描述

DESCRIBE为DataArtsFabric SQL服务下特有语法，该语法功能是显示LakeFormation上指定对象的详细信息。

注意事项

无。

语法格式

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

参数说明

无。

示例

描述目标表的详细信息：

```
DESCRIBE test_table;
```

描述目标数据库的详细信息：

```
DESCRIBE SCHEMA test_schema;
```

描述目标视图的详细信息：

```
DESCRIBE VIEW test_view;
```

描述test_schema的函数function_name的详细信息：

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

8.6.12 MSCK REPAIR TABLE

功能描述

MSCK REPAIR TABLE为DataArtsFabric SQL服务下特有语法，该语法功能是将数据目录上的分区信息同步到元数据存储引擎上。

注意事项

无。

语法格式

```
MSCK REPAIR TABLE table_name;
```

参数说明

无。

示例

同步分区信息：

```
MSCK REPAIR TABLE table_name;
```

8.6.13 TRUNCATE

功能描述

TRUNCATE为DataArtsFabric SQL服务下特有语法，该语法功能是将表数据清空。

注意事项

外表（EXTERNAL TABLE）不支持TRUNCATE。

Iceberg表不支持TRUNCATE PARTITION语法。

语法格式

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

参数说明

- **table_name**
  清空数据的目标表名。
- **PARTITIONS**
  如果指定PARTITIONS关键字，则只清空目标分区的数据
- **col_name**
  分区的列名
- **col_value**
  分区数据值

示例

删除表分区id=2的数据：

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

8.7 DML 语法

8.7.1 DML 语法一览表

DML（Data Manipulation Language数据操作语言），用于对数据库表中的数据进行操作。如：插入、更新、查询、删除。

插入数据

插入数据是往数据库表中添加一条或多条记录，请参考**INSERT**。

查询数据

数据库查询语句SELECT是用于在数据库中检索适合条件的信息，请参考**SELECT**。

8.7.2 EXPLAIN

功能描述

显示SQL语句的执行计划。

执行计划将显示SQL语句所引用的表采用的扫描方式。如果引用了多个表，执行计划还会显示使用的JOIN算法。

执行计划中最关键的部分是语句的预计执行开销，即计划生成器对执行该语句所需时间的预估。

如果指定了ANALYZE选项，则该语句会被执行，然后根据实际的运行结果显示统计数据，包括每个计划节点内时间总开销（毫秒为单位）和实际返回的总行数。这对于判断计划生成器是否接近现实非常有用。

注意事项

在指定ANALYZE选项时，语句会被执行。

语法格式

- 显示SQL语句的执行计划，支持多种选项，对选项顺序无要求：
  ```
  EXPLAIN [ (  option  [, ...] )  ] statement;
  ```
  其中选项option子句的语法为：
  ```
  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 }
  ```
- 显示SQL语句的执行计划，且要按顺序给出选项：
  ```
  EXPLAIN  { [  { ANALYZE  | ANALYSE  }  ] [ VERBOSE  ]  | PERFORMANCE  } statement;
  ```
- 显示复现SQL语句的执行计划所需的信息，通常用于定位问题。STATS选项必须单独使用：
  ```
  EXPLAIN ( STATS [ boolean ] ) statement;
  ```

- 显示DDL语句执行步骤的详细耗时信息（该语法仅9.1.0及以上集群版本支持）：
```
EXPLAIN PERFORMANCE statement;
```

参数说明

- **statement**

  指定要分析的SQL语句。

- **ANALYZE boolean | ANALYSE boolean**

  显示实际运行时间和其他统计数据。

  取值范围：

  – TRUE（缺省值）：显示实际运行时间和其他统计数据。

  – FALSE：不显示。

- **VERBOSE boolean**

  显示有关计划的额外信息。

  取值范围：

  – TRUE（缺省值）：显示额外信息。

  – FALSE：不显示。

- **COSTS boolean**

  包括每个规划节点的估计总成本，以及估计的行数和每行的宽度。

  取值范围：

  – TRUE（缺省值）：显示估计总成本和宽度。

  – FALSE：不显示。

- **CPU boolean**

  打印CPU的使用情况的信息。

  取值范围：

  – TRUE（缺省值）：显示CPU的使用情况。

  – FALSE：不显示。

- **DETAIL boolean**

  打印DN上的信息。

  取值范围：

  – TRUE（缺省值）：打印DN的信息。

  – FALSE：不打印。

- **NODES boolean**

  打印query执行的节点信息。

  取值范围：

  – TRUE（缺省值）：打印执行的节点的信息。

  – FALSE：不打印。

- **NUM_NODES boolean**

  打印执行中的节点的个数信息。

  取值范围：

  – TRUE（缺省值）：打印DN个数的信息。

– FALSE：不打印。

- **BUFFERS boolean**

  包括缓冲区的使用情况的信息。

  取值范围：

  – TRUE：显示缓冲区的使用情况。

  – FALSE（缺省值）：不显示。

- **TIMING boolean**

  包括实际的启动时间和花费在输出节点上的时间信息。

  取值范围：

  – TRUE（缺省值）：显示启动时间和花费在输出节点上的时间信息。

  – FALSE：不显示。

- **PLAN**

  是否将执行计划存储在plan_table中。当该选项开启时，会将执行计划存储在PLAN_TABLE中，不打印到当前屏幕，因此该选项为on时，不能与其他选项同时使用。

  取值范围：

  – ON（缺省值）：将执行计划存储在plan_table中，不打印到当前屏幕。执行成功返回EXPLAIN SUCCESS。

  – OFF：不存储执行计划，将执行计划打印到当前屏幕。

- **FORMAT**

  指定输出格式。

  取值范围：TEXT，XML，JSON和YAML。

  默认值：TEXT

- **PERFORMANCE**

  使用此选项时，即打印执行中的所有相关信息。

- **STATS boolean**

  打印复现SQL语句的执行计划所需的信息，包括对象定义、统计信息、配置参数等，通常用于定位问题。

  取值范围：

  – TRUE（缺省值）：显示复现SQL语句的执行计划所需的信息。

  – FALSE：不显示。

示例

修改explain_perf_mode为normal：

```
SET explain_perf_mode=normal;
```

显示表简单查询的执行计划：

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

相关链接

ANALYZE | ANALYSE

8.7.3 INSERT

功能描述

向表中添加一行或多行数据。

注意事项

- 只有拥有表INSERT权限的用户，才可以向表中插入数据。
- 如果使用RETURNING子句，用户必须要有该表的SELECT权限。
- 如果使用QUERY子句插入来自查询里的数据行，用户还需要拥有在查询里使用的表的SELECT权限。
- 如果使用OVERWRITE子句覆盖式插入数据，用户还需要拥有该表的SELECT和TRUNCATE权限。
- 当连接到TD兼容的数据库时，td_compatible_truncation参数设置为on时，将启用超长字符串自动截断功能，在后续的insert语句中（不包含外表的场景下），对目标表中char和varchar类型的列上插入超长字符串时，系统会自动按照目标表中相应列定义的最大长度对超长字符串进行截断。

  说明

  如果向字符集为字节类型编码（SQL_ASCII，LATIN1等）的数据库中插入多字节字符数据（如汉字等），且字符数据跨越截断位置，这种情况下，按照字节长度自动截断，自动截断后会在尾部产生非预期结果。如果用户有对于截断结果正确性的要求，建议用户采用UTF8等能够按照字符截断的输入字符集作为数据库的编码集。

注意

在处理ORC、PARQUET表数据时，建议使用INSERT OVERWRITE语法。例如：

- 当向非分区表中插入数据时，建议使用：INSERT OVERWRITE INTO tablename select_clause;
- 当向分区表中插入数据时，建议使用：INSERT OVERWRITE INTO tablename PARTITION(partvalue [, partvalue]) select_clause;

因为当前针对ORC、PARQUET格式下，INSERT语法无法保证事务。如果使用INSERT语法，新数据会增量式写入表中，写入过程中可能会出现：

- 脏读，读到正在写入的数据。
- 如果写入过程发生异常，可能出现数据残留，无法保证一致性。

语法格式

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

where partition_clause can be:

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

参数说明

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

  用于声明一个或多个可以在主查询中通过名字引用的子查询，相当于临时表。

  其中with_query的详细格式为：

```
with_query_name [ ( column_name [, ...] ) ] AS
( {select | values | insert | update | delete} )
```

  – with_query_name指定子查询生成的结果集名字，在查询中可使用该名称访问子查询的结果集。

  – column_name指定子查询结果集中显示的列名。

  – 每个子查询可以是SELECT，VALUES，INSERT，UPDATE或DELETE语句。

- **plan_hint子句**

  以/*+ */的形式在关键字后，用于对指定语句块生成的计划进行hint调优，详细用法请参见**使用Plan Hint进行调优**。

- **OVERWRITE**

  用于标识覆盖式插入模式，采用此方式执行后，目标表中原有数据将被清除，仅保留新插入的数据。

  OVERWRITE支持指定列插入的功能，其他列为默认值，如果无默认值则为NULL。

  须知

  - OVERWRITE不要和INSERT INTO这类实时写入的操作并发，否则实时写入数据有被意外清理的风险。
  - OVERWRITE适用于大批量数据导入场景，不建议用于少量数据的插入场景。
  - 避免对同一张表执行并发insert overwrite操作。
  - 如果集群正在扩缩容，且INSERT OVERWRITE的写入表需要执行数据重分布，则INSERT OVERWRITE会清除当前数据，并自动将插入的数据按扩缩容后的节点来进行数据分布。如果INSERT OVERWRITE和该表的数据重分布过程同时执行，INSERT OVERWRITE会中断该表的数据重分布过程。
  - OVERWRITE支持覆盖式插入OBS外表，此方式会删除原目录中数据文件。

- **table_name**

  要插入数据的目标表名。

  取值范围：已存在的表名。

- **AS**

  用于给目标表table_name指定别名。alias即为别名的名字。

- **column_name**

  目标表中的字段名：

  – 字段名可以有子字段名或者数组下标修饰。

  – **没有在字段列表中出现的每个字段，将由系统默认值，或者声明时的默认值填充，如果都没有则用NULL填充。例如，向一个复合类型中的某些字段插入数据，其他字段将是NULL。**

  – 目标字段（column_name）可以按顺序排列。如果没有列出任何字段，则默认全部字段，且顺序为表声明时的顺序。

– 如果value子句和query中只提供了N个字段，则目标字段为前N个字段。
– value子句和query提供的值在表中从左到右关联到对应列。

取值范围：已存在的字段名。

- **partition_key_value**

  分区键值。

  通过PARTITION(partition_key_value [, ...])子句指定的这一组值，可以唯一确定一个分区。

  取值范围：需要进行重命名分区的分区键的取值范围。

  注意，此处要求所有partition_key都必须是identity表达式的类型。

- **expression**

  赋予对应column的一个有效表达式或值：

  – 向表中字段插入单引号时需要使用单引号自身进行转义。
  – 如果插入行的表达式不是正确的数据类型，系统试图进行类型转换，如果转换不成功，则插入数据失败，系统返回错误信息。

  示例：

```
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**

  对应字段名的缺省值。如果没有缺省值，则为NULL。

- **query**

  一个查询语句（SELECT语句），将查询结果作为插入的数据。

示例

创建表reason_t1：

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

向表中插入一条记录：

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

向表中插入一条记录，和上一条语法等效：

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

向表中插入TABLE_SK小于1的记录：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

向表中插入多条记录：

```
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)
```

使用INSERT OVERWRITE更新表中的数据，即覆盖式插入数据：

```
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 rows)
```

将表reason_t1的数据插入到表reason_t1中：

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

对独立的字段明确缺省值：

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

将一个表中的部分数据插入到另一个表中：先通过WITH子查询得到一张临时表temp_t，然后将临时表temp_t中的所有数据插入另一张表reason_t1中：

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

向分区表的指定分区插入数据：

```
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 (80) VALUES(85,80);

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

8.7.4 VALUES

功能描述

根据给定的值表达式计算一个或一组行的值。它通常用于在一个较大的命令内生成一个“常数表”。

注意事项

- 应当避免使用VALUES返回数量非常大的结果行，否则可能会遭遇内存耗尽或者性能低下。尤其对于INSERT INTO VALUES语法，建议使用在小数据量插入的场景。对于大数据量的插入，建议使用产品提供的其它导入方式进行。
- 如果指定了多行，那么每一行都必须拥有相同的元素个数。

语法格式

```
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 ];
```

参数说明

- **expression**

  用于计算或插入结果表指定地点的常量或者表达式。

  在一个出现在INSERT顶层的VALUES列表中，expression可以被DEFAULT替换以表示插入目的字段的缺省值。除此以外，当VALUES出现在其他场合的时候是不能使用DEFAULT的。

- **sort_expression**

  一个表示如何排序结果行的表达式或者整数常量。

- **ASC**

  指定按照升序排列。

- **DESC**

  指定按照降序排列。

- **operator**

  一个排序操作符。

- **count**

  返回的最大行数。

- **start**

  开始返回行之前忽略的行数。

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

  FETCH子句限定返回查询结果从第一行开始的总行数，count的缺省值为1。

示例

创建表reason_t1：

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

向表中插入一条记录：

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

向表中插入一条记录，和上一条语法等效：

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

向表中插入TABLE_SK小于1的记录：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

向表中插入多条记录：

```
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)
```

使用INSERT OVERWRITE更新表中的数据，即覆盖式插入数据：

```
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 rows)
```

将表reason_t1的数据插入到表reason_t1中：

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

对独立的字段明确缺省值：

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

将一个表中的部分数据插入到另一个表中：先通过WITH子查询得到一张临时表temp_t，然后将临时表temp_t中的所有数据插入另一张表reason_t1中：

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

向分区表的指定分区插入数据：

```
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);

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

8.7.5 DELETE

功能描述

从指定的表里删除满足WHERE子句的行。如果WHERE子句不存在，将删除表中所有行，结果只保留表结构。

说明

目前仅Iceberg表支持Delete。

注意事项

要删除表中的数据，用户必须对它有DELETE权限。同样也必须有USING子句引用的表以及condition上读取的表的SELECT权限。

警告

避免使用DELETE删除全表数据，考虑使用TRUNCATE TABLE代替。

语法格式

```
DELETE FROM table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition ]
```


参数说明

- table_name：目标表的名字。取值范围：已存在的表名。
- alias：目标表的别名。取值范围：字符串，符合标识符命名规范。
- using_list：using子句。
- condition：一个返回boolean值的表达式，用于判断哪些行需要被删除。

示例

创建表reason_t2：

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

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

使用WHERE条件删除：

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

使用IN语法删除：

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

8.7.6 MERGE INTO

场景描述

在数据处理和分析的场景中，用户经常需要将一个数据源的数据合并到另一个数据源中，例如将增量数据合并到主表中。然而，在实际操作中，由于缺乏有效的单一语句支持，用户不得不使用多个语句组合来实现这一功能，这不仅降低了处理效率，还可能影响数据的一致性和准确性。如何在保证数据处理效率的同时，确保数据的一致性和准确性，成为了一个亟待解决的问题。为此，FabricSQL引入了对Iceberg表的MERGE INTO语法支持，允许用户通过单一语句实现数据的插入、更新或删除操作，从而简化了数据合并流程，提高了数据处理的效率和可靠性。

功能描述

MERGE INTO语句将目标表和源表中数据针对关联条件进行匹配，如果关联条件匹配时对目标表进行UPDATE或DELETE，不匹配时对目标表执行INSERT。MERGE INTO合并执行UPDATE和INSERT，避免多次执行以提高执行效率。

注意事项

- 执行MERGE INTO操作的用户需要同时拥有目标表的UPDATE和INSERT权限，以及源表的SELECT权限。
- MERGE INTO的目标表仅支持Iceberg格式。

语法格式

```
MERGE [/*+ plan_hint */] INTO table_name [ partition_clause ] [ [ AS ] alias ]
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 } [, ...] ) [, ...] }
```

参数说明

- plan_hint子句
  以/*+ */的形式在关键字后，用于对指定语句块生成的计划进行hint调优，详细用法请参见**使用Plan Hint进行调优**。
- INTO子句
  指定正在更新或插入的目标表。
  - **table_name**
    目标表的表名。
  - **partition_clause**
    用来指定目标表分区的子句。
    需要目标表已存在的分区，或者指定的分区键值对应的分区已存在。
  - **alias**
    目标表的别名。
- USING子句
  指定源表，源表可以是表、视图或子查询。
  - **partition_clause**
    用来指定源表分区的子句。
    只有当源表是分区表时可以指定分区，视图或者子查询不支持指定分区。
  - **alias**
    源表的别名。
- ON子句
  关联条件，用于指定目标表和源表的关联条件。不支持更新关联条件中的字段。
- WHEN MATCHED子句
  当源表和目标表中数据根据关联条件可以匹配上时，选择WHEN MATCHED子句进行UPDATE或DELETE操作。
  - 不支持更新分布列。不支持更新系统表、系统列。
  - 一行被多次更新时，会报错。
- WHEN NOT MATCHED子句
  当源表和目标表中数据根据关联条件无法匹配时，选择WHEN NOT MATCHED子句进行INSERT操作。
  不支持INSERT子句中包含多个VALUES。
  WHEN MATCHED和WHEN NOT MATCHED子句顺序可以交换，可以缺省其中一个，但不能同时缺省，可以同时指定多个WHEN MATCHED或WHEN NOT MATCHED子句。
- DEFAULT
  用对应字段的缺省值填充该字段。

- AND condition

  WHEN MATCHED子句和WHEN NOT MATCHED子句的条件，只有在条件满足时才进行操作，可缺省。

示例

创建目标表products和源表newproducts，并插入数据：

```
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');
```

执行MERGE INTO操作，仅当product_id字段匹配且目标表product_name字段非'play gym' 并且源表product_name字段非'lamaze'时UPDATE，源表product_name字段为'lamaze'时DELETE；product_id字段不匹配且源表category字段为'books'时INSERT。

```
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)
;
```

查询更新后的结果显示：目标表products的对应product_id字段1502、1666行相关数据已更新，1601行相关数据被删除，1700行为新插入的数据。

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

8.7.7 OPTIMIZE

功能描述

对目标表的指定数据进行重写。

说明

目前仅Iceberg表支持该语法。

注意事项

- 如果用户指定了排序键，数据重写后，文件内数据有序，而无法保证全局有序。
- 建议选择重写条件时，按照分区维度便捷重写条件。

语法格式

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

参数说明

- table_name：目标表的名字。取值范围：已存在的表名。
- condition：一个返回boolean值的表达式，用于判断哪些文件需要被重写。
- target_list：数据重写的排序键。
- option_key, option_value：数据重写的配置选项。

表 8-38 option_key, option_value 参数说明

参数	取值	说明	默认值
target-file-size-bytes	0~Max(uint64)	重写后的文件大小。	与表参数write.target-file-size-bytes保持一致
min-file-size-bytes	0~Max(uint64)	原始文件大小小于该阈值的文件会被当做目标文件。	75% * target-file-size-bytes
max-file-size-bytes	0~Max(uint64)	原始文件大小大于该阈值的文件会被当做目标文件。	180% * target-file-size-bytes
min-input-files	0~Max(uint32)	当单个分区内目标文件个数大于该值时，将会对该分区进行数据重写。	5
delete-file-threshold	0~Max(uint32)	当单个分区内delete文件个数大于该值时，将会对该分区进行数据重写。	INT_MAX

参数	取值	说明	默认值
rewrite-all	true, false	忽视以上所有条件，对所有满足用户条件的文件进行重写。	false
max-file-group-size-bytes	0~Max(uint64)	分区内目标文件总大小大于该阈值时，将该分区的目标文件拆分为多个task进行处理。	100GB

示例

创建表reason_t2：

```
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');
```

使用WHERE条件对TABLE_SC = 'Hainan'分区数据进行重写：

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

使用ORDER BY语句查看TABLE_SC = 'Lingnan'分区数据重写计划：

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

                                                    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
```

设置min-file-size-bytes以重写全部目标文件：

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

8.7.8 UPDATE

功能描述

更新表中的数据。UPDATE修改满足条件的所有行中指定的字段值，WHERE子句声明条件，SET子句指定的字段会被修改，没有出现的字段则保持它们的原值。

说明

目前仅Iceberg表支持Update。

注意事项

- 要修改表中的数据，用户必须对它有UPDATE权限。同样对expression或condition条件里涉及到的任何表要有SELECT权限。
- 试图在一个SQL语句中用多行数据更新同一行时会报错。

警告

避免UPDATE并发更新同一分区的数据。

语法格式

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
 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 [, ...] ]
```

参数说明

- plan_hint子句：以/*+ */的形式在关键字后，用于对指定语句块生成的计划进行hint调优，详细用法请参见**使用Plan Hint进行调优**。
- table_name：要更新的表名，可以使用模式修饰。
- alias：目标表的别名。
- column_name：要修改的字段名。
  - 支持使用目标表的表名加字段名来引用这个字段。例如：
    ```
    UPDATE foo SET foo.col_name = 'a'
    ```
  - 支持使用目标表的别名加字段名来引用这个字段。例如：
    ```
    UPDATE foo AS f SET f.col_name = 'a'
    ```
- expression：赋给字段的值或表达式。
- DEFAULT：用对应字段的缺省值填充该字段。如果没有缺省值，则为NULL。
- sub_query：子查询。使用同一数据库里其他表的信息来更新一个表可以使用子查询的方法。其中SELECT子句具体介绍请参考**SELECT**。

- from_list：一个表的表达式列表，允许在WHERE条件里使用其他表的字段。与在一个SELECT语句的FROM子句里声明表列表类似。

  注意：目标表绝对不能出现在from_list里，除非在使用一个自连接（此时它必须以from_list的别名出现）。
- condition：一个返回boolean类型结果的表达式。只有这个表达式返回true的行才会被更新。

示例

创建表reason_update：

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

向表reason_update中插入数据：

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

对表reason_update执行UPDATE操作：

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

8.8 DCL 语法

8.8.1 ANALYZE | ANALYSE

功能描述

用于收集有关数据库中表内容的统计信息，统计结果存储在LakeFormation上。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。

能够执行ANALYZE特定表的用户，包括表的所有者、被授予该表上读取权限的用户。

注意事项

ANALYZE会扫描全表数据，同时会产生计费。

语法格式

收集表的统计信息。

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

参数说明

table_name

需要分析的特定表的表名（可能会带模式名）。

取值范围：已有的表名。

示例

使用ANALYZE语句更新表customer_info统计信息：

```
ANALYZE customer_info;
```

8.9 DQL 语法

8.9.1 DQL 语法一览表

DQL（Data Query Language数据查询语言），用于从表或视图中获取数据。

查询

DataArtsFabric SQL提供了用于从表或视图中获取数据的语句。

具体信息，请参见**SELECT**。

8.9.2 SELECT

功能描述

SELECT用于从表中读取数据。

SELECT语句就像叠加在数据库表上的过滤器，利用SQL关键字从数据表中过滤出用户需要的数据。

注意事项

必须对每个在SELECT命令中使用的字段有查询权限。

语法格式

```
[ 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 )} [, ...] ]
[ { 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 ];
```

说明

condition和expression中可以使用targetlist中表达式的别名。

- 只能同一层引用。
- 只能引用targetlist中的别名。
- 只能是后面的表达式引用前面的表达式。
- 不能包含volatile函数。
- 不能包含Window function函数。
- 不支持在join on条件中引用别名。
- targetlist中有多个要应用的别名则报错。

- 其中子查询with_query为：
```
with_query_name [ ( column_name [, ...] ) ]
    AS [ [ NOT ] MATERIALIZED ] ( {select | values | insert | update | delete} )
```
- 其中指定查询源from_item为：
```
{[ 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 [, ...] ) ]}
```
- 其中time_travel_clause为：
```
[FOR TIMESTAMP AS OF timestamp_expression] | [FOR VERSION AS OF snapshot_id]
```
- 其中group子句为：
```
( )
| expression
| ( expression [, ...] )
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )
| GROUPING SETS ( grouping_element [, ...] )
```
- 其中指定分区partition_clause为：
```
PARTITION { ( partition_name ) |
      FOR (  partition_value [, ...] ) }
```
  说明

  指定分区只适合普通表。
- 其中设置排序方式nlssort_expression_clause为：
```
NLSSORT ( column_name, ' NLS_SORT = { SCHINESE_PINYIN_M | generic_m_ci } ' )
```
- 简化版查询语法，功能相当于select * from table_name。
```
TABLE { ONLY {(table_name)| table_name} | table_name [ * ]};
```

参数说明

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

  用于声明一个或多个可以在主查询中通过名字引用的子查询，相当于临时表。

  其中with_query的详细格式为：with_query_name [(column_name [, ...])] AS [[NOT] MATERIALIZED] ({select | values | insert | update | delete})

  – with_query_name指定子查询生成的结果集名字，在查询中可使用该名称访问子查询的结果集。
  – 默认情况下，被主查询多次引用的with_query通常只被执行一次，并将其结果集进行物化，供主查询多次查询其结果集；被主查询引用一次的with_query，则不再单独执行，而是将其子查询直接替换到主查询中的引用

处，随主查询一起执行。显示指定[NOT] MATERIALIZED，可改变默认行为：

  - 指定MATERIALIZED时，将子查询执行一次，并将其结果集进行物化。
  - 指定NOT MATERIALIZED时，则将其子查询替换到主查询中的引用处。以下几种情况会忽略NOT MATERIALIZED：
    - 子查询中含有volatile函数。
    - 子查询为INSERT/UPDATE/DELETE等语句。
    - 被引用次数大于1的with_query2引用了外层自引用的with_query1，则with_query2不能被替换到引用处。

      例如下面示例中，tmp2被引用了两次，tmp2因为引用了外层自引用的tmp1，所以即使tmp2指定了NOT MATERIALIZED也会被物化。
```
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指定子查询结果集中显示的列名。
- 每个子查询可以是SELECT，VALUES，INSERT，UPDATE或DELETE语句。

- **plan_hint子句**

  以/*+ */的形式在SELECT关键字后，用于对SELECT对应的语句块生成的计划进行hint调优，详细用法请参见**Plan Hint调优概述**。

- **ALL**

  声明返回所有符合条件的行，是默认行为，可以省略该关键字。

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

  从SELECT的结果集中删除所有重复的行，使结果集中的每行都是唯一的。

  ON (expression [, ...]) 只保留那些在给出的表达式上运算出相同结果的行集合中的第一行。

须知

DISTINCT ON表达式是使用与ORDER BY相同的规则进行解释的。除非使用了ORDER BY来保证需要的行首先出现，否则，"第一行" 是不可预测的。

- **SELECT列表**

  指定查询表中列名，可以是部分列或者是全部（使用通配符*表示）。

  通过使用子句AS output_name可以为输出字段取个别名，这个别名通常用于输出字段的显示。

  列名可以用下面几种形式表达：
  - 手动输入列名，多个列之间用英文逗号（,）分隔。
  - 可以是FROM子句里面计算出来的字段。

- **FROM子句**

  为SELECT声明一个或者多个源表。

  FROM子句涉及的元素如下所示。

– table_name

  表名或视图名，名称前可加上模式名，如：schema_name.table_name。

– alias

  给表或复杂的表引用起一个临时的表别名，以便被其余的查询引用。

  别名用于缩写或者在自连接中消除歧义。如果提供了别名，它就会完全隐藏表的实际名字。

– column_alias

  列别名

– PARTITION

  查询分区表的某个分区的数据。

– partition_name

  分区名。

– partition_value

  指定的分区键值。在创建分区表时，如果指定了多个分区键，可以通过PARTITION子句指定的这一组分区键的值，唯一确定一个分区。

– subquery

  FROM子句中可以出现子查询，创建一个临时表保存子查询的输出。

– with_query_name

  WITH子句同样可以作为FROM子句的源，可以通过WITH查询的名字对其进行引用。

– function_name

  函数名称。函数调用也可以出现在FROM子句中。

– join_type

  有5种类型，如下所示。

  ▪ [INNER] JOIN

    一个JOIN子句组合两个FROM项。可使用圆括弧以决定嵌套的顺序。如果没有圆括弧，JOIN从左向右嵌套。

    在任何情况下，JOIN都比逗号分隔的FROM项绑定得更紧。

  ▪ LEFT [OUTER] JOIN

    返回笛卡尔积中所有符合连接条件的行，再加上左表中通过连接条件没有匹配到右表行的那些行。这样，左边的行将扩展为生成表的全长，方法是在那些右表对应的字段位置填上NULL。请注意，只在计算匹配的时候，才使用JOIN子句的条件，外层的条件是在计算完毕之后施加的。

  ▪ RIGHT [OUTER] JOIN

    返回所有内连接的结果行，加上每个不匹配的右边行（左边用NULL扩展）。

    这只是一个符号上的方便，因为总是可以把它转换成一个LEFT OUTER JOIN，只要把左边和右边的输入互换位置即可。

  ▪ FULL [OUTER] JOIN

    返回所有内连接的结果行，加上每个不匹配的左边行（右边用NULL扩展），再加上每个不匹配的右边行（左边用NULL扩展）。

■ CROSS JOIN

CROSS JOIN等效于INNER JOIN ON（TRUE），即没有被条件删除的行。这种连接类型只是符号上的方便，因为它们与简单的FROM和WHERE的效果相同。

说明

必须为INNER和OUTER连接类型声明一个连接条件，即NATURAL ON，join_condition，USING (join_column [， ...]) 之一。但是它们不能出现在CROSS JOIN中。

其中CROSS JOIN和INNER JOIN生成一个简单的笛卡尔积，和在FROM的顶层列出两个项的结果相同。

– ON join_condition

连接条件，用于限定连接中的哪些行是匹配的。如：ON left_table.a = right_table.a。

– USING(join_column[，...])

ON left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。要求对应的列必须同名。

– NATURAL

NATURAL是具有相同名称的两个表的所有列的USING列表的简写。

– from item

用于连接的查询源对象的名称。

– FOR TIMESTAMP AS OF timestamp_expression

Time travel查询，指定查询对象在某一时刻的数据。timestamp_expression可以具体时间，或者时间表达式。必须可在查询开始计算表达式的值。timestamp_expression不支持子查询，相关引用，UDF。

– FOR VERSION AS OF snapshot_id

Time travel查询，指定查询对象的历史版本，snapshot_id即历史版本号。snapshot_id不支持子查询，相关引用，UDF。

说明

Time travel查询，需要查询对象支持访问历史版本。目前支持iceberg表执行Time travel查询

示例：

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

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

--查询当前的数据总行数
SELECT count(1) FROM tt_iecberg FOR TIMESTAMP AS OF now();
 count
-------
     5
(1 row)

--查询5s前的数据总行数
SELECT count(1) FROM tt_iecberg FOR TIMESTAMP AS OF now() - interval '5s';
 count
-------
     5
(1 row)
```

```
--使用iceberg_show_snapshots 查询iceberg表的版本信息
SELECT * FROM iceberg_show_snapshots('tt_iecberg'::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/w00818585/rhel_10_90_45_60/hudi_test_insert_p/
tt_iecberg/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/w00818585/rhel_10_90_45_60/hudi_test_insert_p/tt_iecberg/
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)

--查询指定版本的数据总行数
SELECT count(1) FROM tt_iecberg FOR VERSION AS OF '3230380547366427570';
 count
-------
     3
(1 row

--查询执行时间的数据总行数
SELECT count(1) FROM tt_iecberg FOR TIMESTAMP AS OF '2025-07-11 10:54:39.264+00'
 count
-------
     3
(1 row)
```

- **WHERE子句**

  WHERE子句构成一个行选择表达式，用来缩小SELECT查询的范围。condition是返回值为布尔型的任意表达式，任何不满足该条件的行都不会被检索。

  WHERE子句中可以通过指定"(+)"操作符的方法将表的连接关系转换为外连接。但是不建议用户使用这种用法，因为这并不是SQL的标准语法，在做平台迁移的时候可能面临语法兼容性的问题。同时，使用"(+)"有很多限制：

  a. "(+)"只能出现在where子句中。
  b. 如果from子句中已经有指定表连接关系，那么不能再在where子句中使用"(+)"。
  c. "(+)"只能作用在表或者视图的列上，不能作用在表达式上。
  d. 如果表A和表B有多个连接条件，那么必须在所有的连接条件中指定"(+)"，否则"(+)"将不会生效，表连接会转化成内连接，并且不给出任何提示信息。
  e. "(+)"作用的连接条件中的表不能跨查询或者子查询。如果"(+)"作用的表，不在当前查询或者子查询的from子句中，则会报错。如果"(+)"作用的对端的表不存在，则不报错，同时连接关系会转化为内连接。
  f. "(+)"作用的表达式不能直接通过"OR"连接。

g. 如果"(+)"作用的列是和一个常量的比较关系， 那么这个表达式会成为join条件的一部分。

h. 同一个表不能对应多个外表。

i. "(+)"只能出现"比较表达式"，"NOT表达式"，“ANY表达式”，“ALL表达式”，“IN表达式”，“NULLIF表达式”，“IS DISTINCT FROM表达式”，“IS OF”表达式。"(+)"不能出现在其他类型表达式中，并且这些表达式中不允许出现通过“AND”和“OR”连接的表达式。

j. "(+)"只能转化为左外连接或者右外连接，不能转化为全连接，即不能在一个表达式的两个表上同时指定"(+)"。

须知

对于WHERE子句的LIKE操作符，当LIKE中要查询特殊字符“%”、“_”、“\”的时候需要使用反斜杠“\”来进行转义。

示例：

```
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子句**

  将查询结果按某一列或多列的值分组，值相等的为一组。

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

    ROLLUP是计算一个有序的分组列在GROUP BY中指定的标准聚集值，然后从右到左进一步创建高层次的部分和，最后创建了累积和。一个分组能够看做一系列的分组集。例如：

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

    等价于：

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

    ROLLUP子句中的元素可以是单独的字段或表达式，也可以是使用括号包含的列表。如果是括号中的列表，产生分组集时它们必须作为一个整体。例如：

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

等价于：

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

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

CUBE是自动对group by子句中列出的字段进行分组汇总，结果集将包含维度列中各值的所有可能组合，以及与这些维度值组合相匹配的基础行中的聚合值。它会为每个分组返回一行汇总信息， 用户可以使用CUBE来产生交叉表值。例如，在CUBE子句中给出三个表达式（n = 3），运算结果为$2^n = 2^3 =$ 8组。 以n个表达式的值分组的行称为常规行，其余的行称为超级聚集行。例如：

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

等价于：

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

CUBE子句中的元素可以是单独的字段或表达式，也可以是使用括号包含的列表。如果是括号中的列表，产生分组集时它们必须作为一个整体。例如：

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

等价于：

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

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

GROUPING SETS子句是GROUP BY子句的进一步扩展，它可以使用户指定多个GROUP BY选项。选项用于定义分组集，每个分组集都需要包含在单独的括号中，空白的括号（()）表示将所有数据当作一个组处理。 这样做可以通过裁剪用户不需要的数据组来提高效率。 用户可以根据需要指定所需的数据组进行查询。

须知

如果SELECT列表的表达式中引用了那些没有分组的字段，则会报错，除非使用了聚集函数，因为对于未分组的字段，可能返回多个数值。

- **HAVING子句**

  与GROUP BY子句配合用来选择特殊的组。HAVING子句将组的一些属性与一个常数值比较，只有满足HAVING子句中的逻辑表达式的组才会被提取出来。

- **WINDOW子句**

  一般形式为WINDOW window_name AS (window_definition) [， ...]，window_name是可以被随后的窗口定义所引用的名称，window_definition可以是以下的形式：

  [existing_window_name]

  [PARTITION BY expression [, ...]]

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

  [frame_clause]

  frame_clause为窗函数定义一个窗口框架window frame，窗函数（并非所有）依赖于框架，window frame是当前查询行的一组相关行。frame_clause可以是以下的形式：

  [RANGE | ROWS] frame_start

  [RANGE | ROWS] BETWEEN frame_start AND frame_end frame_start和frame_end可以是：

UNBOUNDED PRECEDING

value PRECEDING（RANGE不支持）

CURRENT ROW

value FOLLOWING（RANGE不支持）

UNBOUNDED FOLLOWING

- **UNION子句**

  UNION计算多个SELECT语句返回行集合的并集。

  UNION子句有如下约束条件：

  – 除非声明了ALL子句，否则缺省的UNION结果不包含重复的行。

  – 同一个SELECT语句中的多个UNION操作符是从左向右计算的，除非用圆括弧进行了标识。

  一般表达式：

  select_statement UNION [ALL] select_statement

  – select_statement可以是任何没有ORDER BY、LIMIT子句的SELECT语句。

  – 如果用圆括弧包围，ORDER BY和LIMIT可以附着在子表达式里。

- **INTERSECT子句**

  INTERSECT计算多个SELECT语句返回行集合的交集，不含重复的记录。

  INTERSECT子句有如下约束条件：

  – 同一个SELECT语句中的多个INTERSECT操作符是从左向右计算的，除非用圆括弧进行了标识。

  – 当对多个SELECT语句的执行结果进行UNION和INTERSECT操作的时候，会优先处理INTERSECT。

  一般形式：

  select_statement INTERSECT select_statement

- **EXCEPT子句**

  EXCEPT子句有如下的通用形式：

  select_statement EXCEPT [ALL] select_statement

  EXCEPT操作符计算存在于左边SELECT语句的输出而不存在于右边SELECT语句输出的行。

  EXCEPT的结果不包含任何重复的行，除非声明了ALL选项。使用ALL时，一个在左边表中有m个重复而在右边表中有n个重复的行将在结果中出现max(m-n,0) 次。除非用圆括弧指明顺序，否则同一个SELECT语句中的多个EXCEPT操作符是从左向右计算的。EXCEPT和UNION的绑定级别相同。

- **MINUS子句**

  与EXCEPT子句具有相同的功能和用法。

- **ORDER BY子句**

  对SELECT语句检索得到的数据进行升序或降序排序。对于ORDER BY表达式中包含多列的情况：

  – 首先根据最左边的列进行排序，如果这一列的值相同，则根据下一个表达式进行比较，以此类推。

  – 如果对于所有声明的表达式都相同，则按随机顺序返回。

– ORDER BY中排序的列必须包括在SELECT语句所检索的结果集的列中。

须知

- 如果未指定ORDER BY，则按数据库系统最快生成的顺序返回。
- 可以选择在ORDER BY子句中的任何表达式之后添加关键字ASC（升序）或DESC（降序）。如果未指定，则默认使用ASC。
- 如果要支持中文拼音排序和不区分大小写排序，需要在初始化数据库时指定编码格式为UTF-8或GBK。 命令如下:

  initdb –E UTF8 –D ../data –locale=zh_CN.UTF-8或initdb –E GBK –D ../data –locale=zh_CN.GBK。

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

  LIMIT子句由两个独立的Limit子句、Offset子句和一个多参Limit子句构成:

  LIMIT { count | ALL }

  OFFSET start [ROW | ROWS]

  LIMIT start, { count | ALL }

  其中，count声明返回的最大行数，而start声明开始返回行之前忽略的行数。如果这两个参数都指定了，会在开始计算count个返回行之前先跳过start行。多参Limit子句不可和单参的Limit子句或Offset子句共同出现。

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

  如果不指定count，默认值为1，FETCH子句限定返回查询结果从第一行开始的总行数。

- **NLS_SORT**

  指定某字段按照特殊方式排序。目前仅支持中文拼音格式排序和不区分大小写排序。

  取值范围:

  – SCHINESE_PINYIN_M，按照中文拼音排序（目前只支持GBK字符集内的一级汉字排序）。如果要支持此排序方式，在创建数据库时需要指定编码格式为“GBK”，否则排序无效。

  – generic_m_ci，不区分大小写排序。

- **PARTITION子句**

  查询某个分区表中相应分区的数据。

示例

查询reason表的所有r_reason_sk记录，且去除重复。

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

LIMIT子句示例：获取表中一条记录。

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

LIMIT子句示例：获取表中第三条记录。

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

LIMIT子句示例：获取表中前两条记录。

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

查询所有记录，且按字母升序排列。

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

GROUP BY子句示例：根据查询条件过滤，并对结果进行分组。

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

GROUP BY子句示例：通过group by别名来对结果进行分组。

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

GROUP BY CUBE子句示例：根据查询条件过滤，并对结果进行分组汇总。

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

GROUP BY GROUPING SETS子句示例:根据查询条件过滤，并对结果进行分组汇总。

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

UNION子句示例：将表reason里r_reason_desc字段中的内容以W开头和以N开头的进行合并。

```
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%';
```

NLS_SORT子句示例：中文拼音排序。

```
CREATE TABLE stu_pinyin_info (id bigint, name text) store AS orc;
INSERT INTO stu_pinyin_info VALUES (1, '雷锋'),(2, '石传祥');
SELECT * FROM stu_pinyin_info ORDER BY NLSSORT (name, 'NLS_SORT = SCHINESE_PINYIN_M' );
 id |  name
----+--------
  1 | 雷锋
  2 | 石传祥
(2 rows)
```

不区分大小写排序:

```
CREATE TABLE stu_icase_info (id bigint, name text) store AS orc;
INSERT INTO stu_icase_info VALUES (1, 'aaaa'),(2, 'AAAA');
SELECT * FROM stu_icase_info ORDER BY NLSSORT (name, 'NLS_SORT = generic_m_ci');
 id | name
----+------
  1 | aaaa
  2 | AAAA
(2 rows)
```

创建分区表reason_p，并插入数据，再从reason_p的表分区P_05_BEFORE中获取数据。

```
CREATE TABLE reason_p
(
  r_reason_id character(16),
  r_reason_desc character(100)
)
PARTITION BY (r_reason_sk integer) store AS orc;


INSERT INTO reason_p values('AAAAAAAABAAAAAAA','reason 1',3),('AAAAAAAABAAAAAAA','reason 2',10),
('AAAAAAAABAAAAAAA','reason 3',4),('AAAAAAAABAAAAAAA','reason 4',10),
('AAAAAAAABAAAAAAA','reason 5',10),('AAAAAAAACAAAAAAA','reason 6',20),
('AAAAAAAACAAAAAAA','reason 7',30);
```

```
SELECT * FROM reason_p WHERE r_reason_sk=4;
  r_reason_id    |                                          r_reason_desc                                          | r_reason_sk
------------------+--------------------------------------------------------------------------------------------------------
+-------------
 AAAAAAAABAAAAAAA | reason 3                                                                                         |           4
(1 row)
——查询分区列等于10的行数：
SELECT count(*) FROM reason_p WHERE r_reason_sk=10;
 count
--------
     3
(1 row)
```

GROUP BY子句示例：按r_reason_id分组统计reason_p表中的记录数。

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

GROUP BY CUBE子句示例：根据查询条件过滤，并对查询结果分组汇总。

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

GROUP BY GROUPING SETS子句示例：根据查询条件过滤，并对查询结果分组汇总。

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

HAVING子句示例：按r_reason_id分组统计reason_p表中的记录，并只显示r_reason_id个数大于2的信息。

```
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)
```

IN子句示例：按r_reason_id分组统计reason_p表中的r_reason_id个数，并只显示r_reason_id值为AAAAAAAABAAAAAAA或AAAAAAAADAAAAAAA的个数。

```
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)
```

INTERSECT子句示例：查询r_reason_id等于AAAAAAAABAAAAAAA，并且r_reason_sk小于5的信息。

```
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                                          | r_reason_sk
------------------+--------------------------------------------------------------------------------------------------------
+-------------
 AAAAAAAABAAAAAAA | reason 1                                                                                         |           3
 AAAAAAAABAAAAAAA | reason 3                                                                                         |           4
(2 rows)
```

EXCEPT子句示例：查询r_reason_id等于AAAAAAAABAAAAAAA，并且去除r_reason_sk小于4的信息。

```
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)
```

通过在where子句中指定"(+)"来实现左连接。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk  =
t2.c_customer_sk(+)
order by 1 desc limit 1;
 sr_item_sk | c_customer_id
------------+---------------
      18000 |
(1 row)
```

通过在where子句中指定"(+)"来实现右连接。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+)  =
t2.c_customer_sk
order by 1 desc limit 1;
 sr_item_sk |  c_customer_id
------------+------------------
            | AAAAAAAAJNGEBAAA
(1 row)
```

通过在where子句中指定"(+)"来实现左连接，并且增加连接条件。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk  =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1 order by 1  limit 1;
 sr_item_sk | c_customer_id
------------+---------------
          1 |
(1 row)
```

不支持在where子句中指定"(+)"的同时使用内层嵌套AND/OR的表达式。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where not(t1.sr_customer_sk  =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1);
ERROR:  Operator "(+)" can not be used in nesting expression.
LINE 1: ...tomer_id from store_returns t1, customer t2 where
not(t1.sr_...
```

where子句在不支持表达式宏指定"(+)"会报错。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where (t1.sr_customer_sk  =
t2.c_customer_sk(+))::bool;
ERROR:  Operator "(+)" can only be used in common expression.
```

where子句在表达式的两边都指定"(+)"会报错。

```
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+)  =
t2.c_customer_sk(+);
ERROR:  Operator "(+)" can't be specified on more than one relation in one join condition
HINT:  "t1", "t2"...are specified Operator "(+)" in one condition.
```

9 多模态 AI 数据湖

9.1 多模态数据处理

9.1.1 概述

在AI场景中，数据科学家经常面临多模态数据（如文本、图像、音频、视频、点云等）的管理与分析挑战，这些数据通常分散在不同的数据源中，导致数据整合和处理效率低下。为了解决这一问题，如何实现跨模态数据的高效统一管理与分析？FabricSQL多模态AI数据湖服务应运而生，它是一站式的Serverless服务平台，专为管理、处理、分析多模态数据类型而设计，旨在为AI提供高质量、多维度的数据支持。其核心目标是通过整合异构数据源，实现跨模态数据的统一存储、处理和智能化分析，从而赋能AI模型的训练、推理及应用。此外，Fabric Data是基于Fabric平台孵化的一套Dataset开发接口，适合数据科学家、数据分析师的使用习惯，以直观、声明式的方式管理、操作、分析数据。

核心能力

- 多模态数据统一处理：支持结构化、半结构化和非结构化数据的原生处理，提供统一的数据处理能力。
- 强类型系统：通过强类型系统保障类型安全，减少调试成本，提高开发效率。
- 函数式编程接口：提供简单易用的函数式编程接口，如map、flatMap等，作为API的“第一公民”。
- Lazy执行与高性能引擎：采用Lazy执行模式，结合FabricSQL分布式执行引擎，提供卓越的性能表现。
- 与AI工具链的深度集成：与主流AI工具链（如TensorFlow、PyTorch等）实现深度集成和无缝对接，支持端到端的AI开发流程。

9.1.2 快速开始

安装 FabricData SDK

- 在线安装：

  通过华为源使用pip进行安装：


```
pip install huawei_fabric_data --trusted-host pypi.cloudartifact.dgg.dragon.tools.huawei.com -i https://
pypi.cloudartifact.dgg.dragon.tools.huawei.com/artifactory/cbu-pypi-public/simple/
```

- 离线安装：

  下载SDK，使用以下命令安装SDK包。

  ```
  pip install huawei_fabric_data-0.1.0-py3-none-any.whl
  ```

  python版本推荐使用3.11。

使用 Fabric Data

步骤1 创建会话，入参请参考**表9-1**。

```
import ibis  # 导入ibis依赖
from fabric_data.multimodal import ai_lake
import logging
import os
con = ai_lake.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"),
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
    fabric_workspace_id=os.getenv("fabric_workspace_id"),
    lf_catalog_name=os.getenv("lf_catalog_name"),
    lf_instance_id=os.getenv("lf_instance_id"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
    use_single_cn_mode=True,  # 表示是否开启单CN模式
    logging_level=logging.INFO, # 设置日志级别
)
```

步骤2 加载数据集。

```
target_database = "test"
dataset_name = "test_data"
ds = con.load_dataset(dataset_name, database=target_database)
ds.show()
   a  b  c
0  1  3  1
```

步骤3 数据加工。

```
import ibis.expr.datatypes as dt
import fabric_data as fabric
func_signature=fabric.Signature(
     parameters=[
         fabric.Parameter(name="start", annotation=int),
         fabric.Parameter(name="endnum", annotation=int),
         fabric.Parameter(name="step", annotation=int)
     ],
     return_annotation=dt.Struct({"col1": int}),
)
@fabric.udf.python(database=target_database, signature=func_signature)
def py_generate_series(start, endnum, step):
    current = start
    while current <= endnum:
        yield {'col1':current}
        current += step
ds = ds.flat_map(
   fn=py_generate_series,
   on=[ds.a, ds.b, ds.c],
   as_col='new_col',
   dpu=0.33
)
ds.show()
   a  b  c  new_col
0  1  3  1    1
1  1  3  1    2
2  1  3  1    3
```

步骤4 关闭会话。

```
con.close()
```

----结束

表 **9-1** connect()接口参数信息

参数	含义	类型	默认值	备注
fabric_endpoint	指定连接的端点。	str | None	None	更多信息，请参见**地区和终端节点**。
fabric_endpoint_id	指定连接的端点ID。	str | None	None	更多信息，请参见**查询endpoint_id**。
fabric_workspace_id	指定连接的工作空间ID。	str | None	None	更多信息，请参见**获取工作空间ID**。
lf_instance_id	指定连接的LakeFormation服务实例ID。	str | None	None	-
lf_catalog_name	指定连接的目录。	str | None	None	-
access_key	指定连接的访问密钥ID。	str | None	None	更多信息，请参见**获取ak/sk**。
secret_key	指定连接的访问密钥。	str | None	None	-
security_token	指定连接的安全令牌（仅在上述访问密钥ID，访问密钥是临时的情况下需要）。	str | None	None	-
logging_level	日志级别。	int	logging.INFO	-
use_single_cn_mode	表示是否开启单CN模式。	bool	False	-
verify	指定连接的认证。	bool	False	-
default_database	指定连接的数据库名称。	str | None	None	-

9.1.3 用户自定义函数

9.1.3.1 UDF 概述

当FabricSQL的内置函数无法满足用户业务场景时，用户可以使用自定义函数来封装业务逻辑，UDF的具体能力请参考自定义函数。

在FabricData SDK中支持了Scalar UDF、Class UDF、Vectorized UDF、UDTF、UDAF等不同类别UDF的注册和使用。

表 9-2 UDF 类型介绍

类型	输入	输出	描述
Scalar UDF（标量UDF）	单行输入，多参数，例如(a, b, c)	单行输出，一个标量，例如int/float/string/date	最常见形式，每次被调用时只处理一行数据。可以做简单计算、格式化、清洗、转换。例如：字符串清洗、数学计算、日期转换。类比Python的普通函数。
Class UDF（类UDF）	与Scalar UDF相同：单行输入	单行输出	是对Scalar UDF的OOP封装方式，适合有内部缓存、初始化参数、模型对象等情况。例如需要在函数内部加载一次模型、正则、配置。数据库只是在内部初始化一次，不需要每次调用都构建资源。
Vectorized UDF（向量化UDF）	一次性输入一批行，类型为pandas.Series/pyarrow.ChunkedArray	一次性输出同等数量行的向量化结构	性能最优的单行转换方式，用于批量处理。适合：向量化数学计算、文本批量处理、embedding处理、NLP分词。具有SIMD/批量执行优势，常用于Arrow、Pandas。
UDTF（User Defined Table Function，表函数）	单行输入	多行输出	一个输入可能产出多个输出行。类似Python的yield。用于拆分、展开、解包结构化数据。例如：拆分JSON数组、展开多模态数据、音频/视频分片、tokenize、爆炸式展开。
UDAF（User Defined Aggregate Function，聚合函数）	多行输入，可能来自多个分区	单个输出	对一组数据做聚合的函数：累加、统计、聚类、embedding聚合。具备accumulate、merge、finish生命周期，支持分布式执行。

UDF 注册方式

注册UDF总体上来说分为两种方式：显式注册和隐式注册。

表 9-3 注册 UDF 方式

注册方式	含义	是否依赖会话对象	是否侵入式添加注册逻辑	适用场景	参考
显式注册	代码中明确指定UDF的注册信息	是	是	如果用户希望明确控制注册时间，允许侵入式添加注册逻辑，或对同一个Backend连接下的Scalar UDF注册和使用分离有要求。	**UDF显式注册语法**
隐式注册	运行时自动发现并注册UDF	否	否	如果用户希望无侵入式地注册Scalar UDF，且对同一个Backend连接下的Scalar UDF注册和使用分离无要求。	**UDF隐式注册语法**

UDF 注册类型

不管是显式注册还是隐式注册，对于不同的UDF注册类型，其含义有所不同，详情参见下表：

表 9-4 UDF 注册类型的含义

UDF注册类型	含义	是否向量化	适用场景与特点	参考
python	将一个原始的Python函数或者类注册进数据库中。	否	逐行处理数据，适用于简单或特定的计算，但性能较低。	**Python/Pyarrow/Pandas UDF注册参数**
builtin	获得数据库已存在的函数的句柄，无实际注册的操作。	否	直接调用数据库后端已存在的函数，适用于利用数据库原生功能的场景。	**Builtin UDF注册参数**
pyarrow	将一个接受pyarrow.ChunkedArray作为输入和输出的Python函数或类注册进数据库中。	是	利用Pandas的矢量化操作，适用于需要在Python层进行复杂数据处理的场景。	**Python/Pyarrow/Pandas UDF注册参数**

UDF注册类型	含义	是否向量化	适用场景与特点	参考
pandas	将一个接受pandas.Series作为输入和输出的Python函数或类注册进数据库中。	是	利用PyArrow的高性能计算能力，适用于需要处理大型数据集或进行高效计算的场景。	**Python/Pyarrow/Pandas UDF注册参数**

对于Scalar UDF，目前支持全部上述4种注册类型。

对于UDAF、UDTF，目前仅支持上述的python、builtin这2种注册类型。

UDF 运行时的额外参数

不管是显式注册还是隐式注册，对于所有的UDF类型，都支持在使用时通过with_arguments方法传入额外参数。这些参数包括了concurrency、min_concurrency、max_concurrency、timeout、dpu、apu，可以为UDF的执行提供细粒度的资源分配、并发度控制、时间上限等。

具体用法见**UDF WITH ARGUMENTS使用语法**。

9.1.3.2 Scalar UDF

Scalar UDF是最基本的行处理函数， 它接受零个或多个输入参数，并对这一行数据进行操作，最终返回一行结果。

通常Scalar UDF适用于数学运算、复杂类型转换和自定义格式化等行式处理的场景。

Scalar UDF的行为与内置函数（COS(), MAX(), MIN()）非常相似，但其具体实现由用户自定义。

示例

1. 建立连接。
```
import ibis
import fabric_data as fabric
from fabric_data.udf import RegisterType

con = ibis.fabric.connect(...)
```
2. 定义UDF。
```
def add_udf(lhs: int, rhs: int) -> int:
    return lhs + rhs

def calculate_product(price: float, quantity: int) -> float:
    return price * quantity

con.create_scalar_function(
    add_udf,
    input_type=fabric.udf.InputType.PYTHON,
    database=db,
```

```
    comment="To test create_scalar_function"
)
```

3. 注册UDF。
```
# 显式，直接注册UDF
udf = con.udf.python.register(calculate_product, database="your-database", register_type=RegisterType.STAGED)
# 或者从文件注册UDF
udf = con.udf.python.register_from_file("your-current-file-path", "calculate_product", database="your-database", register_type=RegisterType.STAGED)
```
4. 使用UDF并执行。
```
# 使用UDF
t = con.table(name="your-table", schema="your-schema")
expression = t.select(udf(t.price, t.quantity).name("sum column"))
print(expression.execute())
```

9.1.3.3 Class UDF

Class UDF基于面向对象编程范式实现，主要目的是为了创建有状态的处理逻辑。函数要求实现__init__, __call__等方法。

Class UDF 注册

Class UDF注册传入的是Python类，对于__init__、__call__、 __del__ 3个实例方法有特殊的含义认定，详情请参见下表；其他的Python类和实例方法不做限制，用户可以任意添加。

表 9-5 注册 Python Class 时的特殊实例方法

Python Class方法	是否必须	参数	含义	适用场景
__init__(self, *args, **kwargs)	否	通过with_arguments方法传入参数，只允许传入标量Scalar。	UDF的构造方法。	初始化UDF属性（如保存参数、打开文件、建立连接等）。
__call__(self, *args, **kwargs)	是	通过UDF算子传入参数，可以传入标量Scalar和列名Column。	UDF的业务逻辑方法入口。	UDF处理数据、执行计算。
__del__(self)	否	不支持传入参数。	UDF的析构方法。	UDF资源清理（如关闭文件、断开网络连接等）。

示例

```
import ibis
import fabric_data as fabric
from fabric_data.udf import RegisterType

class Hello:
    def __init__(self, firstname: str, lastname: str):
        self.firstname = firstname
        self.lastname = lastname

    def __call__(self, location: str) -> str:
        return f"Hello{self.firstname} {self.lastname}! Welcome to {location}"

con = ibis.fabric.connect(...)
```

```
# 显式，直接注册UDF
udf = con.udf.python.register(Hello, database="your-database", register_type=RegisterType.STAGED)
# 或者从文件注册UDF
udf = con.udf.python.register_from_file("your-current-file-path", "Hello", database="your-database",
register_type=RegisterType.STAGED)

# 使用Class UDF
t = con.table(name="your-table", schema="your-schema")
expression = t.select(udf(t.city).with_arguments(firstname="Ethan", lastname="Carter"))

print(expression.execute())
```

9.1.3.4 Vectorized UDF

Vectorized UDF是向量化执行的函数，是为解决传统行式UDF性能瓶颈而设计的高效函数。入参、出参通常为PyArrow或者Pandas类型。

示例

下文提供两个示例展示如何使用Vectorized UDF。

- 示例一：使用Pyarrow进行向量化加速。

```
import fabric_data as fabric
from fabric_data.udf import RegisterType

import pyarrow.compute as pc

# 隐式注册UDF
@fabric.udf.pyarrow(database="your-database", register_type=RegisterType.STAGED)
def calculate_product(prices: fabric.PyarrowVector[float], quantities: fabric.PyarrowVector[int]) ->
fabric.PyarrowVector[float]:
    return fabric.PyarrowVector[float](pc.multiply(prices, quantities))

# 使用UDF
con = ibis.fabric.connect(...)
t = con.table("your-table", database="your-database")
expression = t.select(calculate_product(t.price, t.quantity).name("product column"))
print(expression.execute())
```

- 示例二：使用Pandas进行向量化加速。

```
import fabric_data as fabric
from fabric_data.udf import RegisterType

import pandas as pd

# 隐式注册UDF
@fabric.udf.pandas(database="your-database", register_type=RegisterType.STAGED)
def calculate_product(prices: fabric.PandasVector[float], quantities: fabric.PandasVector[int]) ->
fabric.PandasVector[float]:
    return fabric.PandasVector[float](prices * quantities, dtype=pd.Float64Dtype())

# 使用UDF
con = ibis.fabric.connect(...)
t = con.table("your-table", database="your-database")
expression = t.select(calculate_product(t.price, t.quantity).name("product column"))
print(expression.execute())
```

9.1.3.5 UDTF

UDTF是输入一行返回多行的函数，适用于典型的多模态数据处理场景，包括视频/音频切帧，样本生成等。UDTF具体的定义和使用约束请参考**UDTF**。

示例

以下示例展示UDTF的注册和使用的流程。

1. 创建会话。
```
import ibis
import fabric_data as fabric
import logging
# create session
con = ai_lake.connect(...)
```
2. 定义并注册UDTF。
```
# function definition
def py_generate_series(start, endnum, step):
    current = start
    while current <= endnum:
        yield {'col1':current}
        current += step

# register udtf
import ibis.expr.datatypes as dt
func_signature=fabric.Signature(
    parameters=[
        fabric.Parameter(name="start", annotation=int),
        fabric.Parameter(name="endnum", annotation=int),
        fabric.Parameter(name="step", annotation=int)
    ],
    return_annotation=dt.Struct({"col1": int}),
)
target_database = 'test'
con.create_table_function(py_generate_series, database=target_database, signature=func_signature)
```
3. 调用UDTF。
```
# use udtf
ds = con.load_dataset("demo_table", database=target_database)
py_generate_series_handler = con.get_function("py_generate_series", database=target_database)
ds = ds.flat_map(
   fn=py_generate_series_handler,
   on=[ds.a, ds.b, ds.c],
   as_col='udtf_col'
)

# trigger executing
res = ds.execute()
print(res)
| a     | b     | c     | udtf_col |
| ----- | ----- | ----- | -------- |
| int64 | int64 | int64 | int64    |
| ----- | ----- | ----- | -------- |
| 1     | 4     | 1     | 1        |
| 1     | 4     | 1     | 2        |
| 1     | 4     | 1     | 3        |
| 1     | 4     | 1     | 4        |
```

9.1.3.6 UDAF

UDAF是输入多行返回一组聚合值的函数，适用于典型的多模态数据处理场景，包括视频/音频/图像的统计特征聚合，多模态标签总结等。UDAF具体的定义和使用约束请参考**UDAF**。

UDAF 注册

UDAF注册传入的必须是Python类，对于__init__、aggregate_state、accumulate、merge、finish、 __del__ 6个实例方法有特殊的含义认定，详情请参见下表；其他的Python类和实例方法不做限制，用户可以任意添加。

表 9-6 注册 UDAF 时的特殊实例方法

Python Class方法	是否必须	参数	含义	适用场景
__init__(self, *args, **kwargs)	否	通过with_arguments方法传入参数，只允许传入标量Scalar。	UDAF的构造方法。	初始化UDAF属性（如保存参数、打开文件、建立连接等）。
aggregate_state(self)	是	无。	可序列化的聚合中间态，用于跨分区传递。	accumulate与merge间共享状态。
accumulate(self, *args, **kwargs)	是	通过UDAF算子传入参数，可以传入标量Scalar和列名Column。	增量更新聚合中间态。	正常数据扫过阶段。
merge(self, other_state)	是	来自其它分区的中间态。	合并其它分区状态到当前中间态。	分布式reduce阶段，多分区聚合结果归并。
finish(self)	是	无。	输出最终聚合结果。	聚合收尾。
__del__(self)	否	不支持传入参数。	UDAF的析构方法。	UDAF资源清理（如关闭文件、断开网络连接等）。

示例

```
import ibis
import fabric_data as fabric
import datetime as dt

# create session
con = ai_lake.connect(...)

# UDAF definition
class PythonVWAP:
    """A generic UDAF that computes VWAP (volume-weighted average price) and total quantity."""

    def __init__(
        self,
        *,
        symbol: Optional[str] = None,
        start_date: Optional[dt.date] = None,
        end_date: Optional[dt.date] = None,
        min_qty: float = 0.0,
        round_digits: Optional[int] = None,
        json_indent: Optional[int] = None,
        **kwargs: Any,
    ):
        self._symbol = symbol
        self._start = start_date
        self._end = end_date
```

```
        self._min_qty = float(min_qty)
        self._round = round_digits
        self._json_indent = json_indent
        self._agg_state = _AggState()

    @property
    def aggregate_state(self) -> _AggState:
        return self._agg_state

    def _date_in_range(self, d: Optional[dt.date]) -> bool:
        if d is None:
            return not (self._start or self._end)
        if self._start and d < self._start:
            return False
        if self._end and d > self._end:
            return False
        return True

    def accumulate(
        self,
        quantity: Optional[float],
        price: Optional[float],
        date: Optional[dt.date] = None,
        symbol: Optional[str] = None,
    ) -> None:
        st = self._agg_state
        st.rows_seen += 1

        if quantity is None or price is None:
            return
        try:
            q = float(quantity)
            p = float(price)
        except (TypeError, ValueError):
            return
        if q <= 0 or p <= 0 or q < self._min_qty:
            return
        if self._symbol is not None and symbol is not None and symbol != self._symbol:
            return
        if not self._date_in_range(date):
            return

        st.sum_px_qty += p * q
        st.sum_qty += q
        st.rows_used += 1

    def merge(self, other_agg_state: _AggState) -> None:
        st = self._agg_state
        st.sum_px_qty += other_agg_state.sum_px_qty
        st.sum_qty += other_agg_state.sum_qty
        st.rows_seen += other_agg_state.rows_seen
        st.rows_used += other_agg_state.rows_used

    def finish(self) -> str:
        st = self._agg_state
        if st.sum_qty <= 0:
            payload: Dict[str, Optional[float | int]] = {
                "vwap": None,
                "total_quantity": 0.0,
                "rows_seen": st.rows_seen,
                "rows_used": st.rows_used,
            }
        else:
            v = st.sum_px_qty / st.sum_qty
            if self._round is not None:
                v = round(v, self._round)
            payload = {
                "vwap": v,
                "total_quantity": st.sum_qty,
```

```
                "rows_seen": st.rows_seen,
                "rows_used": st.rows_used,
            }

        # Return JSON *string*
        return json.dumps(
            payload,
            indent=self._json_indent,
            ensure_ascii=False,
            sort_keys=True,
            separators=None if self._json_indent is not None else (",", ":"),
        )

# register udaf
con.create_agg_function(
    PythonVWAP,
    database="test",
    signature=fabric.Signature(
        parameters=[
            fabric.Parameter(name="quantity", annotation=int),
            fabric.Parameter(name="price", annotation=float),
            fabric.Parameter(name="date", annotation=dt.date),
            fabric.Parameter(name="symbol", annotation=str),
        ],
        return_annotation=str,
    ),
    volatility=fabric.udf.VolatilityType.IMMUTABLE,
    strict=False,
)

# use udaf
ds = con.load_dataset("stock_table", database="test")
VMAP_handler = con.get_function("PythonVWAP", database="test")

from fabric_data.multimodal.function import AggregateFnBuilder
udaf_builder = AggregateFnBuilder(
    fn=VMAP_handler,
    on=[ds.quantity, ds.price, ds.data, ds.symbol],
    as_col="VWAP_column",
    num_dpus=0.5,
)

ds = ds.aggregate(udaf_builder, by=[ds.symbol])

# trigger executing
res = ds.execute()

df = res.copy()
# VWAP_column is a JSON string; expand to columns
parsed = df["VWAP_column"].apply(json.loads).apply(pd.Series)
df = pd.concat([df.drop(columns=["VWAP_column"]), parsed], axis=1)

print(df)
| symbol | vwap   | total_quantity | rows_seen | rows_used |
|--------|--------|----------------|-----------|-----------|
| string | float64| float64        | int64     | int64     |
|--------|--------|----------------|-----------|-----------|
| AAPL   | 188.42 | 10000.0        | 120       | 90        |
| MSFT   | 321.15 |  8000.0        | 110       | 85        |
```

9.1.4 多模态数据类型

9.1.4.1 概述

Fabric Data支持通过UDF处理图片（PNG/JPG）、音频（WAV/MP3）和视频（MP4/AVI）等多模态数据。在使用UDF处理数据之前，需要将多模态数据保存到Fabric表中。下文以图片为例演示如何导入数据到Fabric中：

1. 准备图片类型数据。

   a. 将图片数据读出后写入parquet文件中。

   ```
   import pyarrow as pa
   import pandas as pd
   data = {"img": [
       {'filename': "image.png", 'format': 'png', 'height': 1, 'width': 2},
       ]
   }
   with open("image.png", 'rb') as file:
       data["img"][0]["data"] = file.read()
   df = pd.DataFrame(data)
   schema = pa.schema([('img', pa.struct([('filename', pa.string()), ('format', pa.string()), ('height', 
   pa.int64()), ('width', pa.int64()), ('data', pa.binary())]))])
   table = pa.Table.from_pandas(df, schema=schema)
   pa.parquet.write_table(table, "image_type.parquet")
   ```

   b. 将写入数据后的parquet文件上传到OBS中。

2. 创建包含图片类型的表，指定location为上一步的OBS路径。

```
import os
from fabric_data.multimodal import ai_lake
from fabric_data.multimodal.types import image
# Set the target database name
target_database = "multimodal_lake"

import logging
con = ai_lake.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"),
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
    fabric_workspace_id=os.getenv("fabric_workspace_id"),
    lf_catalog_name=os.getenv("lf_catalog_name"),
    lf_instance_id=os.getenv("lf_instance_id"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
    default_database=target_database,
    use_single_cn_mode=True,
    logging_level=logging.WARN,
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"))
con.create_table("image_table", schema={"img": image.Image}, external=True, location="obs://
image_type")
```

9.1.4.2 图片

图片类型内部使用pillow库处理图片，可以通过获取PIL Image对象后通过PIL库处理图片。 支持处理的图片格式：JPEG、PNG、BMP。

表 9-7 属性（Properties）

属性名	类型	说明
filename	str	图像的文件名或路径（如果从文件加载）。
format	str	图像格式（如PNG、JPEG）。
height	int	图像高度（像素）。

属性名	类型	说明
width	int	图像宽度（像素）。
data	bytes	图像数据。
pil_image	PIL_Image.Image	动态属性：返回Pillow格式的图像对象（自动从data解码或缓存）。

表 9-8 方法（Methods）

方法名	参数	返回值	说明
__init__	filename: Union[str, os.PathLike] = None data: Union[str, PIL.Image] = None format: Optional[str] = None	None	构造方法：从文件（filename）或数据（data）加载图像，支持bytes或Pillow对象。
to_numpy	—	np.ndarray	将图像转换为NumPy数组（float32类型）。
resize	new_shape: tuple[int, int]	Image	返回调整尺寸后的新图像。
show	—	None	调用Pillow的默认图像查看器显示图像。
to_thumbnail	size: tuple[int, int] = (100, 100)	Image	生成指定尺寸的缩略图。
from_struct（类方法）	obj: dict（需包含filename、data、format等字段）	Image	从结构化字典创建Image对象。

9.1.4.3 视频

视频内部使用decord和opencv处理数据。

- 支持的格式：mp4、avi、mov、mkv、flv、wmv、m4v、3gp。
- 支持的编码：MPEG-4、MPEG-2、VP8、VP9、AV1、MS-MPEG-4。

视频对象的属性和方法如下所示：

表 **9-9** 属性（Attributes）

属性名	类型	说明
filename	str	视频文件的路径名（如果从文件加载）。
format	str	视频格式（如mp4、avi）。
data	bytes	视频数据。

表 **9-10** 方法（Methods）

方法名	参数	返回值	说明
__init__	filename: Union[str, os.PathLike] = None,data: Union[str, bytes] = None	None	构造函数：从文件或二进制数据加载视频。
video_reader (属性)	—	decord.VideoReader	动态属性：返回decord的视频读取器对象（延迟初始化）。
truncate	start_time: float（开始时间，秒） end_time: float（结束时间，秒） shape: Optional[tuple[int, int]]（目标分辨率） fps: float = None（目标帧率） codec: str = "avc1"（编码格式） format: str = "mp4"（输出格式）	Video	截取视频片段，支持调整分辨率、帧率和编码格式，返回新Video对象。
from_struct (类方法)	obj: dict（需包含filename或data字段）	Video	从字典结构（如JSON）创建Video对象。

9.1.4.4 音频

音频内部处理使用soundfile库处理数据。支持的音频格式：mp3、aiff、flac、wav。音频对象的属性和方法如下所示：

表 9-11 属性（Properties）

属性名	类型	说明
filename	str	音频文件的路径或文件名（如果从文件加载）。
format	str	音频格式（如WAV、MP3）。
data	bytes	音频数据。
sound_file	sf.SoundFile	动态属性：返回soundfile库的音频对象（自动从data解码或缓存）。

表 9-12 方法（Methods）

方法名	参数	返回值	说明
__init__	filename: Union[str, os.PathLike] = None data: Union[str, bytes, sf.SoundFile] = None	None	构造方法：从文件（filename）或数据（data）加载音频，支持二进制数据或soundfile对象。
truncate	start_time: float = 0 end_time: Optional[float] = None	Audio	截取音频片段（时间单位为秒），返回新Audio对象。
convert_format	format: str subtype: Optional[str] = None	Audio	转换音频格式（如WAV转MP3），返回新Audio对象。
write	filename: str	None	将音频写入指定文件（自动推断格式）。

9.1.5 AI Dataset&Table

9.1.5.1 接口总览

Fabric Data主要包含数据处理、数据管理、输入/输出、辅助工具、触发执行五个部分：

- 数据处理：支持丰富的变换操作，如map()、flat_map()、filter()、join()、groupby()等，可对样本进行特征工程和结构化处理。
- 数据管理：提供对数据集的增删改查能力，包括insert()、update()、delete()以及索引创建与删除，支持数据生命周期管理。
- 输入/输出：具备向主流湖仓格式，如Parquet、Iceberg、Data Formation写入数据的能力，实现与数据湖无缝集成。
- 辅助工具：提供schema()、columns()和explain_plan()等元信息查询与执行计划分析功能，便于调试与优化。
- 触发执行：通过execute()、limit()、take()等方法触发实际计算并获取结果，支持懒加载与按需执行。

表 9-13

操作类型	接口	描述
Dataset - 数据处理 (Data Processing)	**map**	对单行输入数据应用一对一函数映射
	map_batchs	对批量输入数据应用一对一函数映射
	flat_map	对单行输入数据应用一对多函数映射
	filter	执行过滤条件，保留满足条件的行
	join	多数据集关联
	order_by	排序
	aggregate	对整个数据集进行聚合
	groupby	数据集分组
	min	计算指定列的最小值
	max	计算指定列的最大值
	mean	计算指定列的均值
	unique	获取指定列的唯一值列表
	select_columns	选取指定列
	add_column	添加新列
	drop_columns	移除特定列
	rename_columns	重命名列
Table - 数据管理 (Data Management)	**insert**	插入数据
	delete	删除数据
	update	更新数据

操作类型	接口	描述
输入/输出(Input/Output)	**write_parquet**	写入数据到parquet表
	write_iceberg	写入数据到iceberg表
辅助工具(Utility)	**schema**	数据集的schema
	columns	数据集列名列表
	count	返回数据集的行数
	explain_plan	打印执行计划
	explain_performance	执行并打印详细计划
	stats	查看执行的统计信息（查询需事先执行）
触发执行(Action)	**show**	触发执行，并展示结果
	execute	触发执行，返回结果
	limit	输出最多limit行记录
	take	触发执行，返回单行迭代器
	take_batch	触发执行，返回批量迭代器

9.1.6 最佳实践

9.1.6.1 语音识别案例

本案例介绍如何定义Vectorized Scalar UDF来进行语音识别、如何定义UDAF来进行聚合统计和可视化。

步骤1 从fabric_data中引入高阶类型（图片、音频、视频）：

```
from fabric_data.multimodal.types.image import Image, display_image
from fabric_data.multimodal.types.audio import Audio
from fabric_data.multimodal.types.vectorized import PandasVector
```

步骤2 定义Class Vectorized Scalar UDF来进行语音识别，这里使用到外部包transformers、torch：

```
class SpeechRecognitionUDF:
    def __init__(self, model_path: str) -> None:
        import os
        from transformers import pipeline
        import torch

        model_dir = os.path.abspath(model_path)
        self._asr = pipeline(
            "automatic-speech-recognition",
            model=model_dir,
            device=0 if torch.cuda.is_available() else -1,
        )
```

```
    def __call__(self, audios: PandasVector[Audio]) -> PandasVector[str]:
        return PandasVector[str](audios.map(
            lambda audio: self._asr(audio.sound_file.read(dtype="float32"))["text"]
        ))
```

步骤3 定义UDAF来进行TF-IDF词频统计和词云图片生成，这里使用到外部包wordcloud、matplotlib：

```
class SpeakerWordsUDAF:
    def __init__(self, top_n: int = 40):
        from typing import List
        self._texts: List[str] = []
        self._top_n = top_n
        self._chapter_id = None

        # tiny built-in stopword list (extend as needed)
        self._stopwords = {
            "the", "a", "an", "and", "or", "but",
            "also", "about", "this", "that", "these", "those",
            "his", "her", "its", "their", "our", "your",
            "of", "to", "in", "on", "for", "at", "by", "with",
            "from", "as", "is", "are", "was", "were", "be", "been",
            "it", "he", "she", "they", "we", "you", "i",
            "not", "do", "did", "does", "had", "have", "has",
            "how", "what", "when", "where", "which", "who", "whom",
            "would", "could", "should", "will", "can", "may",
        }

    @property
    def aggregate_state(self):
        return {
            "texts": self._texts,
            "chapter_id": self._chapter_id,
        }

    def accumulate(self, chapter_id: int, hyp_text: str):
        if hyp_text:
            self._texts.append(hyp_text)
        if chapter_id:
            self._chapter_id = chapter_id

    def merge(self, other_state):
        if other_texts := other_state.get("texts"):
            self._texts.extend(other_texts)
        if other_chapterid := other_state.get("chapter_id"):
            if self._chapter_id:
                # for one SpeakerWordsUDAF instance, it should process texts with same chapter_id
                assert self._chapter_id == other_chapterid
            else:
                # CN get chapter_id from DN
                self._chapter_id = other_chapterid

    def _tokenize(self, text: str):
        # simple whitespace tokenizer + basic cleaning
        tokens = []
        for t in text.split():
            t = t.strip().lower()
            # drop short tokens and non-alpha tokens
            if len(t) < 3:
                continue
            if not t.isalpha():
                continue
            if t in self._stopwords:
                continue
            tokens.append(t)
        return tokens

    def _generate_wordcloud(self, frequencies: dict) -> Image:
        from io import BytesIO
```

```
        import matplotlib.pyplot as plt
        from wordcloud import WordCloud
        try:
            # Create figure
            fig, ax = plt.subplots(figsize=(10, 5))

            # Generate WordCloud
            wc = WordCloud(width=800, height=400, background_color="white") \
                .generate_from_frequencies(frequencies)

            # Draw
            ax.imshow(wc, interpolation='bilinear')
            ax.axis("off")
            ax.set_title(f"WordCloud for Chapter {self._chapter_id}", fontsize=18, pad=20)

            # Save to in-memory bytes
            buf = BytesIO()
            fig.savefig(buf, format="png", bbox_inches="tight")
            buf.seek(0)

            return Image(data=buf.getvalue(), format="PNG")
        except Exception as e:
            # You may log or re-raise here
            raise
        finally:
            # Cleanup: ALWAYS close the figure
            if fig is not None:
                plt.close(fig)

    def finish(self) -> Image:
        import math
        if not self._texts:
            return None

        # 1) tokenize all docs
        docs = [self._tokenize(txt) for txt in self._texts]
        num_docs = len(docs)

        # 2) document frequency
        df = {}
        for tokens in docs:
            for tok in set(tokens):
                df[tok] = df.get(tok, 0) + 1

        # 3) tf-idf per doc → aggregate
        tfidf_sums = {}
        tfidf_counts = {}

        for tokens in docs:
            if not tokens:
                continue
            # term freq in this doc
            tf = {}
            for tok in tokens:
                tf[tok] = tf.get(tok, 0) + 1

            doc_len = len(tokens)
            for tok, f in tf.items():
                tf_val = f / doc_len
                # idf: log(N/df) + 1
                idf_val = math.log(num_docs / df[tok]) + 1.0
                tfidf = tf_val * idf_val

                tfidf_sums[tok] = tfidf_sums.get(tok, 0.0) + tfidf
                tfidf_counts[tok] = tfidf_counts.get(tok, 0) + 1

        # 4) average tf-idf per term
        avg_tfidf = {
            tok: (tfidf_sums[tok] / tfidf_counts[tok])
```

```
            for tok in tfidf_sums
        }

        # 5) sort & top-N
        sorted_terms = sorted(
            avg_tfidf.items(),
            key=lambda kv: kv[1],
            reverse=True
        )[: self._top_n]

        out = {word: float(score) for word, score in sorted_terms}
        return self._generate_wordcloud(out)
```

步骤4 建立远程连接，这里用到外部包huawei-fabricsql-connectorapi：

```
from contextlib import contextmanager
from fabric_data.multimodal import ai_lake
import os

@contextmanager
def create_connect():
    conn = ai_lake.connect(
        fabric_endpoint="172.16.31.57:60001",
        fabric_endpoint_id="8a708bbf-f862-4c53-9622-4c59b44524a8",
        fabric_workspace_id="ca319048-b07c-498c-97df-8f20b8ce0a52",
        lf_catalog_name="mini_kernel2",
        lf_instance_id="e662709e-6223-44a8-b254-d84cf71842de",
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database="multimodal_lake",
        use_single_cn_mode=True,
    )
    print(f"Establish ai_lake.connect: {conn}")
    try:
        yield conn
    finally:
        # release resource
        conn.close()
```

步骤5 准备Parquet表：

```
with create_connect() as conn:
    from fabric_data.ibis.backends.sql.datatype import advance_dtype
    conn.drop_table("librispeech_asr", database="multimodal_lake")
    table = conn.create_table(
        "librispeech_asr",
        schema={
            "file": str,
            "audio": advance_dtype(Audio),
            "text": str,
            "speaker_id": int,
            "chapter_id": int,
            "id": str,
        },
        table_format="parquet",
        external=True,
        if_not_exists=True,
        location="obs://mini-kernel/mini_kernel2/multimodal-test/librispeech_asr",
        database="multimodal_lake",
    )
    print(table)
```

步骤6 注册Scalar UDF，UDAF：

```
with create_connect() as conn:
    conn.set_function_staging_workspace(
        obs_server=os.getenv("obs_server"),
        obs_bucket_name="mini-kernel",
        obs_directory_base="mini_kernel2/multimodal-test",
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
    )
```

```
    print("finish set_function_staging_workspace")

    conn.delete_function("SpeechRecognitionUDF", database="multimodal_lake")
    conn.create_scalar_function(
        SpeechRecognitionUDF,
        database="multimodal_lake",
        packages=["transformers", "torch"],
        imports=["./facebook_wav2vec2_base_960h"]
    )

    conn.delete_function("SpeakerWordsUDAF", database="multimodal_lake")
    conn.create_agg_function(
        SpeakerWordsUDAF,
        database="multimodal_lake",
        packages=["wordcloud", "matplotlib"]
    )
```

步骤7 以Fabric Data API形式使用Scalar UDF, UDAF，得到最后的分组统计词云图片：

```
with create_connect() as conn:
    # load dataset
    ds = conn.load_dataset("librispeech_asr", database="multimodal_lake")
    ds.show(limit=1)
    ds = ds.limit(1)

    # use Scalar UDF for ASR
    udf = conn.get_function("SpeechRecognitionUDF", database="multimodal_lake")

    ds = ds.map_batches(
        fn=udf,
        on=[ds.audio],
        as_col="hyp_text",
        num_dpus=0.5,
        concurrency=1,
        model_path="./fabric_data/examples/facebook_wav2vec2_base_960h",
    )
    ds.show(limit=1)

    # use UDAF for summary
    udaf = conn.get_function("SpeakerWordsUDAF", database="multimodal_lake")

    from fabric_data.multimodal.function import AggregateFnBuilder
    udaf_builder = AggregateFnBuilder(
        fn=udaf,
        on=[ds.chapter_id, ds.hyp_text],
        as_col="tf_idf_image",
        num_dpus=0.5,
    )

    summary = ds.aggregate(
        udaf_builder,
        by=ds.chapter_id,
    )
    summary.show(limit=1)

    df = summary.execute()
    print(df)
    display_image(df["tf_idf_image"].tolist())
```

----结束

9.1.6.2 视频帧描述与摘要生成示例

本示例将介绍如何使用华为Fabric数据湖API实现以下功能：

- 使用UDTF（用户定义表函数）提取关键视频帧并生成对应的帧描述。
- 应用UDAF（用户定义聚合函数）将多个帧描述综合成连贯的视频摘要。

环境准备

本示例需要以下Python包：

```
huawei-fabric-data
huawei-fabricsql-connectorapi
pipeline
torch
transformers
```

操作步骤

1. 创建服务器连接。
```
import os
from fabric_data.multimodal import ai_lake
# Set the target database name
target_database = "multimodal_lake"

import logging
con = ai_lake.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"),
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
    fabric_workspace_id=os.getenv("fabric_workspace_id"),
    lf_catalog_name=os.getenv("lf_catalog_name"),
    lf_instance_id=os.getenv("lf_instance_id"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
    default_database=target_database,
    use_single_cn_mode=True,
    logging_level=logging.WARNING,
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"))
```
2. 创建Fabric表。
```
from fabric_data.multimodal.types import video
# Create a video table where the data is stored in Parquet files on OBS.
con.create_table("yt_vid_type", database=target_database, table_format="parquet", schema={"vio": video.Video},
                 external=True,
                 location="obs://mini-kernel/dataframe_test/yt_video_type", if_not_exists=True)
print(con.describe_table("yt_video_type", database=target_database))
```
3. 定义视频帧提取处理类。
```
import io
from PIL import Image
from fabric_data.multimodal.types import video
class VideoFrameExtractor:
    def __init__(self, target_frames=12):
        self.target_frames = target_frames
    def display_key_frames(self, vio: video.Video):
        from fabric_data.multimodal.types import image
        frames, timestamps, duration = self.extract_key_frames(vio)
        images = []

        for index, frame in enumerate(frames):
            pil_img = Image.fromarray(frame).convert("RGB")
            buffer = io.BytesIO()
            pil_img.save(buffer, format="JPEG")
            img = image.Image(filename=timestamps[index], data=buffer.getvalue())
            images.append(img.to_thumbnail((200, 200)))
        image.display_image(images)
    def extract_key_frames(self, vio: video.Video):
        try:
            vr = vio.video_reader
```

```
            total_frames = len(vr)
            fps = vr.get_avg_fps()
            duration = total_frames / fps
            # Select the sampling strategy based on video length.
            if duration <= 30:
                frame_indices = self._uniform_sampling(total_frames, min(8, self.target_frames))
            elif duration <= 180:
                frame_indices = self._keyframe_sampling(total_frames, self.target_frames)
            else:
                frame_indices = self._adaptive_sampling(total_frames, fps, self.target_frames)
            frames = vr.get_batch(frame_indices).asnumpy()
            timestamps = [idx / fps for idx in frame_indices]
            return frames, timestamps, duration
        except Exception as e:
            return [], [], 0

    def _uniform_sampling(self, total_frames, num_frames):
        """Sampling at uniform intervals."""
        step = max(1, total_frames // num_frames)
        return list(range(0, total_frames, step))[:num_frames]
    def _keyframe_sampling(self, total_frames, num_frames):
        """Key Frame Sampling - Focus on sampling the beginning, middle, and end."""
        key_points = [
            0,
            total_frames // 4,
            total_frames // 2,
            3 * total_frames // 4,
            total_frames - 1
        ]
        # Insert intermediate frames.
        additional = num_frames - len(key_points)
        if additional > 0:
            step = total_frames // (additional + 1)
            additional_indices = [i * step for i in range(1, additional + 1)]
            key_points.extend(additional_indices)
        return sorted(key_points)[:num_frames]
    def _adaptive_sampling(self, total_frames, fps, num_frames):
        """Adaptive Sampling - Sampling by Time Segments"""
        segments = 6
        frames_per_segment = max(1, num_frames // segments)
        indices = []
        for i in range(segments):
            start = (i * total_frames) // segments
            end = ((i + 1) * total_frames) // segments
            # Sample within each paragraph.
            if frames_per_segment > 0:
                step = max(1, (end - start) // frames_per_segment)
                segment_indices = [start + j * step for j in range(frames_per_segment)
                                   if start + j * step < end]
                indices.extend(segment_indices)
        return indices[:num_frames]

t = con.load_dataset("yt_video_type", database=target_database)
df = t.limit(1).select_columns(t.vio).execute()
frame_extractor = VideoFrameExtractor()
for row in df.itertuples():
    print(frame_extractor.display_key_frames(row[1]))
```

4. 定义视频帧描述处理类。

   本示例使用Hugging Face的**Salesforce/blip-image-captioning-base**模型作为图像到文本生成模型。

   由于数据库中的Python UDF目前不提供可靠的网络连接和下载功能，建议将原始模型下载到本地并通过OBS上传压缩文件（如.zip）。

   可以通过以下Python脚本在本地机器上下载Salesforce/blip-image-captioning-base：

```
from huggingface_hub import snapshot_download
local_dir = "blip-image-captioning-base"
```

```
snapshot_download(
    repo_id="Salesforce/blip-image-captioning-base",
    local_dir=local_dir,
    local_dir_use_symlinks=False,
)
import json
import torch
from transformers import BlipProcessor, BlipForConditionalGeneration, pipeline
class VideoContentAnalyzer:
    def __init__(self, model_name="./fabric_data/examples/blip-image-captioning-base",
target_frames=12):
        self.extractor = VideoFrameExtractor(target_frames)
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.processor = BlipProcessor.from_pretrained(model_name)
        self.model = BlipForConditionalGeneration.from_pretrained(
            model_name,
            torch_dtype=torch.float16 if self.device == "cuda" else torch.float32
        ).to(self.device)
        self.model.eval()
    def __call__(self, vio: video.Video):
        frames, _, _ = self.extractor.extract_key_frames(vio)
        frame_analyses = self.analyze_frames(frames)
        for analyse in frame_analyses:
            yield {"description": json.dumps(analyse)}
    def analyze_frames(self, frames):
        """Analyze the video frame content."""
        frame_analyses = []
        for i, frame in enumerate(frames):
            try:
                # Generate frame description.
                frame_description = self._generate_frame_description(frame)
                # Analyze content features.
                content_features = self._analyze_content_features(frame_description)
                frame_analyses.append({
                    'frame_index': i,
                    'description': frame_description,
                    'content_type': content_features['content_type'],
                    'key_objects': content_features['key_objects'],
                    'scene_context': content_features['scene_context']
                })
            except Exception as e:
                # Current frame processing failed; skipping to next frames.
                continue

        return frame_analyses
    def _generate_frame_description(self, frame):
        pil_image = Image.fromarray(frame).convert("RGB")
        """Generate a single-frame description."""
        inputs = self.processor(images=pil_image, return_tensors="pt").to(self.device)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_length=50,
                num_beams=5,
                early_stopping=True
            )
        description = self.processor.decode(outputs[0], skip_special_tokens=True)
        return description.strip()
    def _analyze_content_features(self, description):
        description_lower = description.lower()
        content_categories = {
            'educational': ['tutorial', 'lesson', 'explain', 'teach', 'learn', 'education'],
            'entertainment': ['funny', 'comedy', 'entertainment', 'show', 'performance'],
            'nature': ['outdoor', 'nature', 'landscape', 'mountain', 'forest', 'animal'],
            'sports': ['sport', 'game', 'player', 'team', 'match', 'competition'],
            'food': ['food', 'cooking', 'recipe', 'meal', 'restaurant', 'kitchen'],
            'technology': ['computer', 'tech', 'device', 'electronic', 'software'],
            'people': ['person', 'people', 'man', 'woman', 'child', 'group']
        }
```

```
        detected_categories = []
        for category, keywords in content_categories.items():
            if any(keyword in description_lower for keyword in keywords):
                detected_categories.append(category)
        common_objects = [
            'person', 'people', 'man', 'woman', 'child', 'car', 'building',
            'tree', 'house', 'street', 'water', 'sky', 'food', 'animal'
        ]
        key_objects = [obj for obj in common_objects if obj in description_lower]
        if any(word in description_lower for word in ['indoors', 'inside', 'room']):
            scene_context = 'indoor'
        elif any(word in description_lower for word in ['outdoors', 'outside', 'park', 'street']):
            scene_context = 'outdoor'
        else:
            scene_context = 'unknown'

        return {
            'content_type': detected_categories[:2] if detected_categories else ['general'],
            'key_objects': key_objects[:5],
            'scene_context': scene_context        }
```

5. 创建UDTF处理视频数据。

```
import ibis.expr.datatypes as dt
import fabric_data as fabric
signature = fabric.Signature(
    parameters=[
        fabric.Parameter(name="vio", annotation=video.Video),
    ],
    return_annotation=dt.Struct({"description": str}),
)
try:
    #con.delete_function("VideoContentAnalyzer", database=target_database)
    video_content_analyzer = con.get_function("VideoContentAnalyzer", database=target_database)
except Exception as e:
    con.create_table_function(VideoContentAnalyzer, database=target_database, signature=signature,
imports=("blip-image-captioning-base",))     video_content_analyzer =
con.get_function("VideoContentAnalyzer", database=target_database)
```

6. 执行UDTF生成视频帧描述。

```
t = t.limit(1)
descriptions = t.select_columns(video_content_analyzer(t.vio).with_arguments(model_name="./
fabric_data/examples/blip-image-captioning-base", target_frames=12).name("descriptions"))
df = descriptions.execute()
import pandas as pd
pd.set_option('display.width', None)
pd.set_option('max_colwidth', None)print(df)
```

7. 定义UDAF从帧描述生成视频摘要。

   以下示例使用Hugging Face的**google-t5/t5-small**模型作为文本到文本生成模型。

   由于数据库中的Python UDF目前不提供可靠的网络连接和下载功能，建议将原始模型下载到本地并通过OBS上传压缩文件（如.zip）。

```
from collections import Counter
import typing
class VideoSummaryGenerator:
    """Video Summary Generator"""

    def __init__(self, model):
        # Initialize the text summarization model.
        self.summarizer = pipeline(
            "summarization",
            model=model,
            device=0 if torch.cuda.is_available() else -1
        )
    def generate_summary(self, frame_analyses):
        """Generate a video summary"""
        all_descriptions = [analysis['description'] for analysis in frame_analyses]
        if not all_descriptions:
```

```
            return "Unable to analyze the video content."

        # Analyze the characteristics of the video content.
        content_analysis = self._analyze_video_content(frame_analyses)
        # Generate a summary
        summary = self._generate_detailed_summary(all_descriptions, content_analysis)
        return summary
    def _analyze_video_content(self, frame_analyses):
        """Analyze the overall video content."""
        all_descriptions = ' '.join([analysis['description'] for analysis in frame_analyses]).lower()
        # Extract the key information
        content_types = []
        all_objects = []
        for analysis in frame_analyses:
            content_types.extend(analysis['content_type'])
            all_objects.extend(analysis['key_objects'])
        # Calculate frequency
        common_content_types = [item for item, count in Counter(content_types).most_common(2)]
        common_objects = [item for item, count in Counter(all_objects).most_common(5)]
        return {
            'primary_categories': common_content_types,
            'dominant_objects': common_objects,
            'total_frames_analyzed': len(frame_analyses)
        }
    def _create_description_text(self, descriptions, content_analysis):
        """Create descriptive text."""
        desc_counter = Counter(descriptions)
        representative_descs = [desc for desc, count in desc_counter.most_common(3)]
        text = "Video Content Analysis:\n"
        text += f"Primary Categories: {', '.join(content_analysis['primary_categories'])}\n"
        text += f"Key Objects: {', '.join(content_analysis['dominant_objects'][:3])}\n"
        text += "Representative Scenes:\n"
        for i, desc in enumerate(representative_descs, 1):
            text += f"{i}. {desc}\n"

        return text
    def _generate_concise_summary(self, description_text, content_analysis):
        """Generate a concise summary"""
        primary_category = content_analysis['primary_categories'][0] if
content_analysis['primary_categories'] else "General content"
        main_objects = ', '.join(content_analysis['dominant_objects'][:2])
        templates = [
            f"This is a {primary_category} video that includes elements like {main_objects}.",
            f"The video presents a {primary_category} scenario, with its focus on {main_objects}.",
            f"The content centers on {primary_category}, highlighting elements such as {main_objects}."
        ]
        import random
        return random.choice(templates)
    def _generate_detailed_summary(self, description_text, content_analysis):
        # Use the summary model to generate a more detailed description.
        try:
            summary = self.summarizer(
                description_text,
                max_length=150,
                min_length=50,
                do_sample=False
            )[0]['summary_text']
            return summary
        except Exception as e:
            # Failed to generate detailed information, switched to producing concise summary.
            return self._generate_concise_summary(description_text, content_analysis)
    def _generate_engaging_summary(self, description_text, content_analysis):
        """Generate an engaging summar"""
        primary_category = content_analysis['primary_categories'][0] if
content_analysis['primary_categories'] else "Awesome"
        main_objects = '、'.join(content_analysis['dominant_objects'][:2])
        engaging_templates = [
            f"□ Don't miss out! This {primary_category} video takes you deep into the fascinating world
of {main_objects}!",
```

```
            f"□ A spectacular showcase! A visual feast exploring the unique charm of {main_objects} in a
{primary_category} setting!",
            f"□ Watch now! This video perfectly captures the {primary_category} moment of
{main_objects}!"
        ]
        import random
        return random.choice(engaging_templates)
class VideoSummarySystem:
    """Video Summary Generation System"""

    def __init__(self, model="./fabric_data/examples/t5-small"):
        self.summary_generator = VideoSummaryGenerator(model)
        self.frame_descriptions = []
    def accumulate(self, description: str) -> None:
        self.frame_descriptions.append(json.loads(description))
    def finish(self) -> str:
        summary = self.summary_generator.generate_summary(self.frame_descriptions)
        result = {
            'video_summary': summary,
            'frames_analyzed': len(self.frame_descriptions),
            'content_categories': list(set(
                cat for analysis in self.frame_descriptions
                for cat in analysis['content_type']
            )),
            'key_objects': list(set(
                obj for analysis in self.frame_descriptions
                for obj in analysis['key_objects']
            ))[:8],
            'processing_details': {
                'total_frames': len(self.frame_descriptions),
                'successful_analyses': len(self.frame_descriptions)
            }
        }
        return json.dumps(result)
    @property
    def aggregate_state(self) -> typing.Dict[str, typing.Any]:
        return {
            "frame_descriptions": self.frame_descriptions,
        }
    def merge(self, other_state: typing.Dict[str, typing.Any]) -> None:        self.frame_descriptions +=
other_state.get("frame_descriptions")
```

8. 执行UDAF生成视频描述。

```
from fabric_data.multimodal.function import AggregateFnBuilder
con.delete_function("VideoSummarySystem", database=target_database)
# Create a UDAF to generate video descriptions from key frame descriptions.
con.create_agg_function(
    VideoSummarySystem,
    database=target_database,
    imports=("t5-small",))
video_summary = con.get_function("VideoSummarySystem", database=target_database)
agg_builder = AggregateFnBuilder(
    fn=video_summary,
    on=[descriptions.descriptions.description],
    as_col="describe",
    num_dpus=0.5,
    constructor_kwargs={"model": "./fabric_data/examples/t5-small"}
)
df = descriptions.aggregate(agg_builder).execute()
for row in df.itertuples():    print(row[1])
```

9. 关闭连接。

```
con.close()
```

9.1.6.3 多模态数据类型使用示例

图片使用示例

1. 准备图片类型数据。
   a. 将图片数据读出后写入parquet文件中。
   ```
   import pyarrow as pa
   import pandas as pd
   data = {"img": [
       {'filename': "image.png", 'format': 'png', 'height': 1, 'width': 2},
       ]
   }
   with open("image.png", 'rb') as file:
       data["img"][0]["data"] = file.read()
   df = pd.DataFrame(data)
   schema = pa.schema([('img', pa.struct([('filename', pa.string()), ('format', pa.string()), ('height',
   pa.int64()), ('width', pa.int64()), ('data', pa.binary())]))])
   table = pa.Table.from_pandas(df, schema=schema)pyarrow.parquet.write_table(table,
   "image_type.parquet")
   ```
   b. 将写入数据后的parquet文件上传到OBS中。
2. 创建包含图片类型的表，指定location为上一步的OBS路径。
   ```
   import os
   from fabric_data.multimodal import ai_lake
   from fabric_data.multimodal.types import image
   # Set the target database name
   target_database = "multimodal_lake"

   import logging
   con = ai_lake.connect(
       fabric_endpoint=os.getenv("fabric_endpoint"),
       fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
       fabric_workspace_id=os.getenv("fabric_workspace_id"),
       lf_catalog_name=os.getenv("lf_catalog_name"),
       lf_instance_id=os.getenv("lf_instance_id"),
       access_key=os.getenv("access_key"),
       secret_key=os.getenv("secret_key"),
       default_database=target_database,
       use_single_cn_mode=True,
       logging_level=logging.WARNING,
   )
   con.set_function_staging_workspace(
       obs_directory_base=os.getenv("obs_directory_base"),
       obs_bucket_name=os.getenv("obs_bucket_name"),
       obs_server=os.getenv("obs_server"),
       access_key=os.getenv("access_key"),
       secret_key=os.getenv("secret_key"))
   con.create_table("image_table", schema={"img": image.Image}, external=True, location="obs://
   image_type")
   ```
3. 使用udf处理图片。
   ```
   from fabric_data.multimodal.types import image
   from fabric_data.ibis.expr.operations.udf import RegisterType
   @fabric_data.udf.python(register_type=RegisterType.STAGED, database="database")
   def udf(img: image.Image) -> image.Image:
       new_img = img.resize((10, 10))
       return new_img
   t = con.load_dataset("image_table", database=target_database)
   df = t.select(img=udf(t.img)).execute()
   df["img"][0].show()
   ```

音频使用示例

1. 准备音频类型数据。
   a. 将音频数据读出后写入parquet文件中。

```
import pyarrow as pa
import pandas as pd
data = {"audio": [
    {'filename': "audio.mp3", 'format': 'mp3', 'data': "", 'subtype': None},
]
}
import base64
with open("audio.mp3", 'rb') as file:
    data["audio"][0]["data"] = file.read()
df = pd.DataFrame(data)
schema = pa.schema([('audio', pa.struct([('filename', pa.string()), ('format', pa.string()),
('subtype', pa.int64()), ('data', pa.binary())]))])
table = pa.Table.from_pandas(df, schema=schema)
pyarrow.parquet.write_table(table, "audio_type.parquet")
```

b. 将写入数据后的parquet文件上传到OBS中。

2. 创建包含音频数据的表，指定location为上一步的OBS路径。

```
import os
from fabric_data.multimodal import ai_lake
from fabric_data.multimodal.types import audio
# Set the target database name
target_database = "multimodal_lake"

import logging
con = ai_lake.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"),
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
    fabric_workspace_id=os.getenv("fabric_workspace_id"),
    lf_catalog_name=os.getenv("lf_catalog_name"),
    lf_instance_id=os.getenv("lf_instance_id"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
    default_database=target_database,
    use_single_cn_mode=True,
    logging_level=logging.WARNING,
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"))
con.create_table("audio_table", schema={"audio": audio.Audio}, external=True, location="obs://
audio_type")
```

3. 使用udf处理音频。

```
from fabric_data.multimodal.types import audio
from fabric_data.ibis.expr.operations.udf import RegisterType
@fabric_data.udf.python(register_type=RegisterType.STAGED, database="database")
def udf(aio: audio.Audio) -> audio.Audio:
    new_aio = aio.truncate(start_time=0, end_time=10)
    return new_aio
t = con.load_dataset("audio_table", database=target_database)
df = t.select(aio=udf(t.audio)).execute()
```

视频使用示例

1. 准备视频类型数据。

a. 将视频数据读出后写入parquet文件中。

```
import pyarrow as pa
import pandas as pd
data = {"video": [
    {'filename': "video.mp4", 'format': 'mp4'},
]
}
import base64
with open("video.mp4", 'rb') as file:
```

```
    data["video"][0]["data"] = file.read()
df = pd.DataFrame(data)
schema = pa.schema([('video', pa.struct([('filename', pa.string()), ('format', pa.string()), ('data',
pa.binary())]))])
table = pa.Table.from_pandas(df, schema=schema)
pyarrow.parquet.write_table(table, "video_type.parquet")
```

b. 将写入数据后的parquet文件上传到OBS中。

2. 创建包含视频数据的表，指定location为上一步的OBS路径。

```
import os
from fabric_data.multimodal import ai_lake
from fabric_data.multimodal.types import video
# Set the target database name
target_database = "multimodal_lake"

import logging
con = ai_lake.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"),
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"),
    fabric_workspace_id=os.getenv("fabric_workspace_id"),
    lf_catalog_name=os.getenv("lf_catalog_name"),
    lf_instance_id=os.getenv("lf_instance_id"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
    default_database=target_database,
    use_single_cn_mode=True,
    logging_level=logging.WARNING,
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"))
con.create_table("video_table", schema={"video": video.Video}, external=True, location="obs://
video_type")
```

3. 使用udf处理视频：

```
from fabric_data.multimodal.types import video
from fabric_data.ibis.expr.operations.udf import RegisterType

@fabric_data.udf.python(register_type=RegisterType.STAGED, database="database")
def udf(vio: video.Video) -> video.Video:
    new_vio = vio.truncate(0, 5, (500,500))
    return new_vio
t = con.table("video_table")
df = t.select(vio=udf(t.video)).execute()
```

9.1.7 接口参考

9.1.7.1 Dataset APIs

columns

描述：获取Dataset的所有列的列名集合。

输入参数：无

返回值类型：tuple[str, ...]

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
```

```
...     ak='*'
...     sk='*'
...
...
... )
>>> ds = conn.load_dataset("demo_table_10", database='my_schema')
>>> ds.columns
('species',
  'island',
  'bill_length_mm',
  'bill_depth_mm',  'flipper_length_mm')
```

add_column(*exprs, **mutations)

描述：给一个Dataset添加列。

输入参数：

- exprs (Union[Sequence[Expr], None])： 要添加为列的命名表达式序列。
- mutations(Value)： 使用关键字参数的命名表达式。

返回值类型：已添加新列的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_6", database='my_schema')
>>> ds
| species | bill_length_mm |
| ------- | -------------- |
| string  | float64        |
| ------- | -------------- |
| Adelie  | 40.0           |
| Adelie  | 41.0           |
| Adelie  | 42.0           |
| Adelie  | 43.0           |

#Add a new column from a per-element expression
>>> ds.add_column(new_col=ds.bill_length_mm + 1)
| species | bill_length_mm | new_col  |
| ------- | -------------- | -------- |
| string  | float64        | float64  |
| ------- | -------------- | -------- |
| Adelie  | 40.0           | 41.0     |
| Adelie  | 41.0           | 42.0     |
| Adelie  | 42.0           | 43.0     |
| Adelie  | 43.0           | 44.0     |
```

drop_columns(*fields)

描述：删除Dataset的列。

输入参数：

fields(str)： 被删除列的列名。

返回值类型：删除列后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_12", database='my_schema')
>>> ds
| species | island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| ------- | --------- | -------------- | ------------- | ----------------- |
| string  | string    | float64        | float64       | int64             |
| ------- | --------- | -------------- | ------------- | ----------------- |
| Adelie  | Torgersen | 40.0           | 22.0          | 180               |
| Adelie  | Torgersen | 41.0           | 19.0          | 183               |
| Adelie  | Torgersen | 42.0           | 20.0          | 190               |
| Adelie  | Torgersen | 43.0           | 23.0          | 185               |

# Drop one or more columns
>>> ds.drop_columns("species")
| island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| --------- | -------------- | ------------- | ----------------- |
| string    | float64        | float64       | int64             |
| --------- | -------------- | ------------- | ----------------- |
| Torgersen | 40.0           | 22.0          | 180               |
| Torgersen | 41.0           | 19.0          | 183               |
| Torgersen | 42.0           | 20.0          | 190               |
| Torgersen | 43.0           | 23.0          | 185               |
```

aggregate(metrics=(), /, *, by=(), having=(), **kwargs)

描述：通过指定分组字段，用一组聚合函数对数据集进行聚合。

输入参数：

- metrics(Union[Sequence[AggregateFnBuilder], None])：聚合表达式，这类表达式可以是任何能生成标量值的表达式，其中包括聚合函数，例如sum。
- by(Union[Sequence[Value], None])：分组表达式，一般设为Dataset的列名集合。
- having(Union[Sequence[BooleanValue], None])：聚合后过滤器。其数据维度应该与metrics相同，且having（子句）的输出类型为布尔型。
- kwargs：命名的聚合表达式。

返回值类型：聚合后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_13", database='my_schema')
>>> ds
| fruit  | price |
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
>>> ds.aggregate(
...     by=["fruit"],
...     total_cost=ds.price.sum(),
```

```
...     avg_cost=ds.price.mean(),
...     having=ds.price.sum() < 0.5,
... ).order_by("fruit")
| fruit  | total_cost | avg_cost |
| ------ | ---------- | -------- |
| string | float64    | float64  |
| ------ | ---------- | -------- |
| banana | 0.25       | 0.25     |
| orange | 0.33       | 0.33     |
```

count()

描述：计算该Dataset的总行数。

输入参数：无

返回值类型：int

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_11", database='my_schema')
>>> ds
| a     |
| ----- |
| string|
| ----- |
| foo   |
| bar   |
| baz   |
>>> ds.count()
 3
```

execute(*, limit='default', params=None, **kwargs)

描述：执行表达式。

输入参数：

- limit(Union[int, str, None])：限定表达式执行结果的总行数。
- params(Union[Mapping[Value, Any], None])：标量参数表达式到值的映射。
- kwargs：关键字参数

返回值类型：Union[pandas.DataFrame, pandas.Series, Any]

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_8", database='my_schema')
>>> ds.execute()
     a     b
```

```
0    1    c
1    1    a
2    2    a
[3 rows x 2 columns]
```

filter(expr=None, fn=None, on=None, num_dpus=None, num_apus=None, apu_model=None, concurrency=None, **fn_constructor_kwargs)

描述：过滤掉不满足条件的行。

输入参数：

- expr(Union[list[ir.BooleanValue], Sequence[ir.BooleanValue], IfAnyAll, None])：表达式字符串，需为有效的Python表达式。
- on(Union[str, list[str]])： 用于应用函数的列。
- fn(Union[str, Callable, None])： 可调用的函数，或则函数名称。
- num_dpus(Union[float, None])：数据处理单元的数量，CPU与内存之间存在固定比例1:4。
- num_apus(Union[float, None])：AI处理单元的数量，这是针对NPU/GPU的配置选项。
- apu_model(Union[str, None])：AI处理单元型号，例如NPU/910B、GPU/A100等。
- concurrency(Union[int, Tuple[int, int], None])： 函数执行时的并发度。
- fn_constructor_kwargs：构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：过滤掉不满足条件的行后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_3", database='my_schema')
>>> ds = ds.filter(expr=[ds.a < 2])
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |

# Filtered by User-defined functions
# my_schema.id_filter：inputting an int value, returning false if value is larger than 2.
>>> ds = conn.load_dataset("demo_table_2", database='my_schema')
>>> ds = ds.filter(
...     fn='my_schema.id_filter',
...     on='a',
... )
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |
```


flat_map(fn, on, as_col=None, num_dpus=None, num_apus=None, apu_model=None, concurrency=None, **fn_constructor_kwargs)

描述：将给定的函数应用于每一行，并对结果进行展平。

输入参数：

- fn (Union[str, Callable])：可调用的函数或其名称，例如myschema.myfunction。
- on (Union[str, list[str],])： 要应用函数的列或列的集合。
- as_col (Union[str, list[str], None])：输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
- num_dpus(Union[float, None])：数据处理单元的数量，CPU与内存之间存在固定比例1:4。
- num_apus(Union[float, None])：AI处理单元的数量，这是针对NPU/GPU的配置选项。
- apu_model(Union[str, None])：AI处理单元型号，例如NPU/910B、GPU/A100等。
- concurrency(Union[int, Tuple[int, int], None])： 函数执行时的并发度。
- fn_constructor_kwargs：构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：

Dataset： 调用函数后包含名为as_col的新列的数据集。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_2", database='my_schema')
>>> ds.execute()
      a    b    c
0     1    4    1
[1 rows x 3 columns]

>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
...     on=['a', 'b', 'c'],
...     as_col=['outCol'])
>>> ds
| a     | b     | c     | outCol |
| ----- | ----- | ----- | ------ |
| int64 | int64 | int64 | int64  |
| ----- | ----- | ----- | ------ |
| 1     | 4     | 1     | 1      |
| 1     | 4     | 1     | 2      |
| 1     | 4     | 1     | 3      |
| 1     | 4     | 1     | 4      |

# Access table columns using the ds.attribute (dot notation) style
>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
...     on=[ds.a, ds.b, ds.c],
...     as_col='udtf_col')
>>> ds
| a     | b     | c     | udtf_col |
```

```
| ----- | ----- | ----- | -------- |
| int64 | int64 | int64 | int64    |
| ----- | ----- | ----- | -------- |
| 1     | 4     | 1     | 1        |
| 1     | 4     | 1     | 2        |
| 1     | 4     | 1     | 3        |
| 1     | 4     | 1     | 4        |
```

map_batchs(fn, on, as_col=None, num_dpus=None, num_apus=None, apu_model=None, concurrency=None, **fn_constructor_kwargs)

描述：将给定的函数应用于批量数据。

输入参数：

- fn（Union[str, Callable]）：可调用的函数或其名称，例如myschema.myfunction。
- on（Union[str, list[str],]）：要应用函数的列或列的集合。
- as_col (Union[str, list[str], None])：输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
- num_dpus(Union[float, None])：数据处理单元的数量，CPU与内存之间存在固定比例1:4。
- num_apus(Union[float, None])：AI处理单元的数量，这是针对NPU/GPU的配置选项。
- apu_model(Union[str, None])：AI处理单元型号，例如NPU/910B、GPU/A100等。
- concurrency(Union[int, Tuple[int, int], None])：函数执行时的并发度。
- fn_constructor_kwargs：构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：

Dataset：调用函数后包含名为as_col的新列的数据集。

groupby(*by, **key_exprs)

描述：创建一个分组的Dataset。

输入参数：

- by（Union[str, Iterable[str], None]）：分组表达式。
- key_exprs（Union[str, Iterable[str]]）：命名的分组表达式。

返回值类型：一个分组后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_14", database='my_schema')
>>> ds
| fruit  | price |
| ------ | ----- |
| string | float |
```

```
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
```

join(ds, /, on=(), *, join_type='inner', left_suffix='left_', right_suffix='right_')

描述：在两个Dataset之间执行join操作。

输入参数：

- ds (Dataset)：右侧要连接的数据集。
- on (Union[str, Sequence[Union[str, BooleanColumn, Literal[True], Literal[False], tuple[str, str]]]])： 连接条件，详见示例。
- join_type (JoinKind)： 连接方法，如 "inner" 或 "left"。
- left_suffix (str)： 用于重命名左侧表中重叠列的格式字符串后缀（例如 "left_"）。
- right_suffix (str)： 用于重命名右侧表中重叠列的格式字符串后缀（例如 "right_"）。

返回值类型：join后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds1 = conn.load_dataset("demo_table_15", database='my_schema')
>>> ds2 = conn.load_dataset("demo_table_16", database='my_schema')

# Equality inner join on the shared id column. Note the _right suffix added to all overlapping columns from
the right # table (in this case only the "id" column).
>>> ds1.join(ds2, "id", join_type="inner")
|   id   |  fruit  |  price  |     color     |
| ------ | ------- | ------- | ------------- |
| int64  | string  | float64 | string        |
| ------ | ------- | ------- | ------------- |
|      2 |   apple |    0.25 | Yellow        |
|      3 |  banana |    0.33 | Orange        |

# Explicit equality join using the default join_type value of "inner". Note join_type there is no _right suffix
added # to the movieId column since this is an inner join and the movieId column is part of the join
condition.
>>> ds1.join(ds2, ds1.id == ds2.id)
|   id   |  fruit  |  price  |     color     |
|--------|---------|---------|---------------|
| int64  | string  | float64 | string        |
|--------|---------|---------|---------------|
|      2 |   apple |    0.25 | Yellow        |
|      3 |  banana |    0.33 | Orange        |
```

map(fn, on, as_col=None, num_dpus=None, num_apus=None, apu_model=None, concurrency=None, **fn_constructor_kwargs)

描述：对当前Dataset的每一行应用给定函数。

输入参数：

- fn (Union[str, Callable])：可调用函数或其名称，如myschema.myfunction。
- on (Union[str, list[str],])：要应用函数的列或列的集合。
- as_col (Union[str, list[str], None])：输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
- num_dpus(Union[float, None])：数据处理单元的数量，CPU与内存之间存在固定比例1:4。
- num_apus(Union[float, None])：AI处理单元的数量，这是针对NPU/GPU的配置选项。
- apu_model(Union[str, None])：AI处理单元型号，例如NPU/910B、GPU/A100等。
- concurrency(Union[int, Tuple[int, int], None])：函数执行时的并发度。
- fn_constructor_kwargs：构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型: Dataset：调用函数后包含新列（列名为as_col）的数据集。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table", database='my_schema')
>>> ds = ds.map(
...     fn='my_schema.multiply_2',
...     on=['col_a'],
...     as_col='col_c'
... )
>>> ds
| col_a | col_b | col_c |
|-------|-------|-------|
| int64 | string | int64 |
|-------|-------|-------|
|     1 | c     | 2     |
|     1 | a     | 2     |
|     2 | a     | 4     |
```

order_by(key, descending=False)

描述：根据一个或多个表达式对Dataset进行排序。

输入参数：

- key (Union[str, List[str], None])：用于排序的表达式。
- descending (Union[bool, List[bool]])：是否按降序排序（DESC），默认为False。

返回值类型：排序后的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
```

```
>>> ds = conn.load_dataset("demo_table_15", database='my_schema')
>>> ds
| a     | b      | c     |
|-------|--------|-------|
| 3     | a      | 4     |
| 2     | B      | 6     |
| 1     | c      | 5     |
| 3     | D      | 7     |

# Sort by b. Default is ascending. Note how capital letters come before lowercase
>>> ds.order_by("b")
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 2     | B      | 6     |
| 3     | D      | 7     |
| 3     | a      | 4     |
| 1     | c      | 5     |

# Sort in descending order
>>> ds.order_by(ds.b.desc())
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 1     | c      | 5     |
| 3     | a      | 4     |
| 3     | D      | 7     |
| 2     | B      | 6     |

# Sort by multiple columns/expressions
>>> ds.order_by(["a", ds.c.desc()])
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 1     | c      | 5     |
| 2     | B      | 6     |
| 3     | D      | 7     |
| 3     | a      | 4     |
```

rename_columns(names, /, **substitutions)

描述：重命名Dataset中的列。

输入参数：

- names(Mapping[str, str])： 映射（字典），以 {新列名: 旧列名} 的形式指定重命名规则，未在映射中出现的列将保留原名。
- substitutions(str)： 显式重命名的列，以关键字参数形式指定，格式为新名=旧名。

返回值类型：重命名后的新Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_8", database='my_schema')
>>> ds
| a     | b      |
```

```
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.rename_columns({"a": "rename_col"})
| rename_col | b      |
|------------|--------|
| int64      | string |
|------------|--------|
| 1          | c      |
| 1          | a      |
| 2          | a      |
```

schema()

描述：返回Dataset的schema。

输入参数：无

返回值类型：Schema对象。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_7", database='my_schema')
>>> ds.schema()
ibis.Schema {
  species            string
  island             string
  bill_length_mm     float64
  bill_depth_mm      float64
  flipper_length_mm  int64
}
```

select_columns(*exprs, **named_exprs)

描述：从Dataset的列中选取指定的列。

输入参数：

- exprs(Union[Value, str, Iterable[Union[Value, str]]])：列表达式、列名字符串，或由列表达式和列名字符串组成的列表。
- named_exprs(Union[Value, str])：列表达式（支持命名参数形式）。

返回值类型：包含选定列的Dataset。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_5", database='my_schema')
>>> ds.select_columns("species", "bill_length_mm")
```

```
| species    | bill_length_mm |
|------------|----------------|
| string     | float64        |
|------------|----------------|
| Adelie     | 40.0           |
| Adelie     | 41.0           |
| Adelie     | 42.0           |
| Adelie     | 43.0           |

# Selection by zero-indexed column position
>>> ds.select_columns(ds[0], ds[4])
| species    | flipper_length_mm |
|------------|-------------------|
| string     | int64             |
|------------|-------------------|
| Adelie     | 180               |
| Adelie     | 183               |
| Adelie     | 190               |
| Adelie     | 185               |
```

show(limit=10)

描述：显示查询结果生成的数据。

输入参数：

limit(int)：限制输出的行数，默认为10。

返回值类型：None

take(limit)

描述：从Dataset中返回最多limit行数据，该方法适用于数据检查和预览。

输入参数：

- limit (int)：要返回的最大行数。

返回值类型: list[dict[str, Any]]：包含数据集中最多limit行的列表，每行以字典形式表示。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_9", database='my_schema')
>>> ds.take(3)
[{'a': 1}, {'a': 1}, {'a': 2}]
```

write_iceberg(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None)

描述：将数据写入指定的Iceberg表中。

输入参数：

- target_name (str)：目标表的名称。
- database (str)：表所属的数据库。

- create_if_not_exists (bool)： 如果为True，当表不存在时将自动创建该表。
- overwrite (bool)：如果为True，将覆盖表中现有的数据。
- is_external (bool)： 如果为True，将表视为外部表。
- schema（可选，任意类型）： 表的结构定义，可选参数。

返回值类型：None

write_orc(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None)

描述：将数据写入指定的ORC表中。

输入参数：

- target_name (str)： 目标表的名称。
- database (str)： 表所属的数据库。
- create_if_not_exists (bool)： 如果为True，当表不存在时将自动创建该表。
- overwrite (bool)： 如果为True，将覆盖表中现有的数据。
- is_external (bool)： 如果为True，将表视为外部表。
- schema (可选，任意类型)： 表的结构定义，可选参数。

返回值类型：None

write_parquet(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None)

描述：将数据写入指定的parquet表中。

输入参数：

- target_name (str)： 目标表的名称。
- database (str)： 表所属的数据库。
- create_if_not_exists (bool)： 如果为True，当表不存在时将自动创建该表。
- overwrite (bool)： 如果为True，将覆盖表中现有的数据。
- is_external (bool)： 如果为True，将表视为外部表。
- schema（可选，任意类型）： 表的结构定义，可选参数。

返回值类型：None

max(on=None)

描述：计算Dataset中指定列的最大值。

输入参数：

on (Union[str, list[str], None])： 指定要计算最大值的列。

返回值类型：Dataset

min(on=None)

描述：计算Dataset中指定列的最小值。

输入参数：

on（Union[str, list[str], None]）：指定要计算最小值的列。

返回值类型：Dataset

mean(on=None)

描述：计算Dataset中指定列的均值。

输入参数：

on（Union[str, list[str], None]）：指定要计算均值的列。

返回值类型：Dataset

unique(col)

描述：计算Dataset中指定列的唯一值集合。

输入参数：

col（str）：指定要计算唯一值的列。

返回值类型：list

limit(n, /, *, offset=0)

描述：从当前Dataset中选择行，起始位置为偏移量。

输入参数：

- n(Union[int, None])：要包含的行数。如果为None，则从偏移量开始选择整个数据集。
- offset(int)：起始跳过的行数。

返回值类型：

Dataset：从偏移量开始的n行数据。

示例：

```
>>> from fabric_data.multimodal import ai_lake
>>> conn = ai_lake.connect(
...     endpoint='*',
...     ak='*'
...     sk='*'
...     ...
... )
>>> ds = conn.load_dataset("demo_table_4", database='my_schema')
>>> ds
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.limit(2)
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
```

```
| 1     | c      |
| 1     | a      |
| 2     | a      |
# use None with offset to slice starting from a particular row
>>> ds.limit(None, offset=1)
| a      | b      |
|--------|--------|
| int64  | string |
|--------|--------|
| 1      | a      |
| 2      | a      |
```

take_batch(batch_size=20, *, batch_format='default')

描述： 从Dataset中返回最多batch_size行数据。

输入参数：

- batch_size (int) ： batch包含的行数（默认为20）。
- batch_format (Optional[str])：batch的格式。
  - "default"返回pyarrow.Table。
  - "pandas"返回pandas.DataFrame。
  - "numpy"返回字典形式的numpy数组。

返回值类型：按指定格式输出数据集合。

9.1.7.2 Table APIs

insert(data, overwrite=False)

描述：用于将Dataset插入到数据库表中。

输入参数：

- data (dataset.Dataset)：要插入的Dataset对象，必须是fabric_data.multimodal.dataset.Dataset类型。
- overwrite (bool)：是否覆盖已存在的数据，默认为False。

返回值类型：None

show(limit=10)

描述：用于显示表的前N行数据。

输入参数：

limit(int) - 限制输出的行数，默认为10。

返回值类型：None

to_dataset()

描述：将当前表对象转换为Dataset对象。

输入参数：无

返回值类型：Dataset

add_columns(new_cols)

描述：向表中添加一个或多个新列。

输入参数：

new_cols (Dict[str, str])： 要添加的列的字典，键为列名（ str ），值为列类型（ str ）。

返回值类型：None

drop_columns(*columns)

描述：用于从表中删除一个或多个列。

输入参数：

*columns (str)：要删除的列名，可以是多个字符串参数。

返回值类型：None

alter_column_type(column_name, new_type)

描述：修改表中指定列的数据类型。

输入参数：

- column_name (str)： 要修改类型的列名。
- new_type (str)： 新的数据类型名称。

返回值类型：None

rename_column(column_name, new_name)

描述：重命名表中的列名。

输入参数：

- column_name (str)： 当前列名。
- new_type (str)： 新列名。

返回值类型：None

delete(where)

描述：从表中删除满足指定条件的数据。

输入参数：

where (str) - SQL WHERE子句：用于指定删除条件。

返回值类型：None

update(values_expr, where)

描述：更新表中满足指定条件的行数据。

输入参数：

- values_expr (Dict[str, ValueExpression])： 键为列名（str），值为用于更新该列的表达式（ValueExpression）。
- where (Union[ir.BooleanValue, list[ir.BooleanValue], IfAnyAll, None])： 用于过滤要更新的行的条件。如果为None，则更新所有行。

返回值类型：None

9.1.7.3 User-Defined Function APIs

UDF 显式注册语法

显式注册的含义是用户需要手动在Python代码中侵入式添加注册逻辑代码，需要用户使用backend...register/register_from_file来实现，调用即注册。显式注册依赖于已经获得backend会话对象才能进行。

推荐使用显式注册的场景：如果用户希望明确控制注册时间，允许侵入式添加注册逻辑，或对同一个Backend连接下的UDF注册和使用分离有要求。

一个典型的场景是1个开发团队负责UDF的注册，多个团队负责UDF的使用，注册团队和使用团队之间的Python脚本不互通。

表 9-14 UDF 显式注册语法

UDF类型	UDF类型（二级）	注册类型（三级）	代码入口	参考
udf, udaf, udtf	python	直接注册	backend.[udf | udaf | udtf].python.register(<注册函数>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
		从文件注册	backend.[udf | udaf | udtf].python.register_from_file(<文件路径>, <函数名>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
	builtin	直接注册	backend.[udf | udaf | udtf].builtin.register(<注册函数>, <注册参数>)	**Builtin UDF注册参数**
		从文件注册	backend.[udf | udaf | udtf].builtin.register_from_file(<文件路径>, <函数名>, <注册参数>)	**Builtin UDF注册参数**
	pyarrow	直接注册	backend.udf.pyarrow.register(<注册函数>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
		从文件注册	backend.udf.pyarrow.register_from_file(<文件路径>, <函数名>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
	pandas	直接注册	backend.udf.pandas.register(<注册函数>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**

UDF类型	UDF类型（二级）	注册类型（三级）	代码入口	参考
		从文件注册	backend.udf.pandas.register_from_file(<文件路径>, <函数名>, <注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**

UDF 隐式注册语法

隐式注册的含义是依赖Python运行时自动发现并注册UDF。用户不需要在Python代码中侵入式添加注册逻辑代码，而是使用@装饰器修饰原始Python函数，然后在DataFrame中使用被装饰的原始Python函数的标识符，即可完成注册。隐式注册在使用@装饰器时不需要获得backend会话对象，后续在ibis DataFrame中获取backend会话对象。

推荐使用隐式注册的场景：如果用户希望无侵入式地注册UDF，且对同一个Backend连接下的UDF注册和使用分离无要求。

一个典型的场景是1个用户的Python脚本中直接写了整个UDF的注册和使用全流程。

表 9-15 UDF 隐式注册语法

UDF类型	UDF类型（二级）	代码入口	参考
udf, udaf, udtf	python	@fabric.[udf | udaf | udtf].python(<注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
	builtin	@fabric.[udf | udaf | udtf].builtin(<注册参数>)	**Builtin UDF注册参数**
	pyarrow	@fabric.udf.pyarrow(<注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**
	pandas	@fabric.udf.pandas(<注册参数>)	**Python/Pyarrow/Pandas UDF注册参数**

对于隐式注册，实际注册动作的发生时间根据DataFrame操作模式Lazy、Eager有所区别。

参考上文提及的**ibis官方文档**，对于DataFrame的操作分为Eager、Lazy两种模式，由配置项ibis.options.interactive控制，默认为False，即所有的DataFrame都默认是Lazy模式。对于两种DataFrame执行模式，UDF注册的发生时间不同，详情说明如下：

表 9-16 DataFrame 执行模式

ibis.options.interactive	DataFrame执行模式	UDF注册时间	UDF使用时间
False	Lazy	整个DataFrame调用execute方法时	整个DataFrame调用execute方法时
True	Eager	第一次在DataFrame中使用	每一次在DataFrame中使用

Python/Pyarrow/Pandas UDF 注册参数

注册Python/Pyarrow/Pandas UDF的作用是将一个原始的Python函数或类注册进数据库中。

不管是显式注册还是隐式注册，不管是Scalar UDF、UDAF、UDTF，对于注册Python、Pyarrow、Pandas类型的UDF，目前都接受用户传入以下参数：

表 9-17 Python/Pyarrow/Pandas UDF 注册参数

注册参数	含义	类型	默认值
name	指定UDF实际数据库中存储名称。	str | None	None
database	指定UDF所在的LakeFormation数据库。	str | None	None
fn	指定UDF原始的Python函数。	Callable	None
signature	指定UDF函数签名和返回值类型。	fabric_data.ibis.common.annotations.Signature | None	None
replace（目前不可用）	指定UDF是否支持就地修改。	bool	False
temporary（目前不可用）	指定UDF是否会话级别的生命周期。	bool	False
if_not_exist（目前不可用）	指定UDF是否跳过已存在报错。	bool	False
strict	指定UDF是否自动过滤NULL值。	bool	True

注册参数	含义	类型	默认值
volatility	指定UDF稳定性。	VolatilityType.VOLATILE | VolatilityType.STABLE | VolatilityType.IMMUTABLE	VolatilityType.VOLATILE
runtime_version（目前不可用）	指定UDF执行的Python版本。	str	sys.version_info
imports	指定UDF依赖的外部代码文件。	List[str]	None
packages	指定UDF依赖的Python模块。	List[Union[str, module]]	None
register_type	指定UDF的注册形式。	RegisterType.INLINE | RegisterType.STAGED	RegisterType.INLINE
comment	指定UDF的用户注释。	str | None	None

Python/Pyarrow/Pandas UDF 注册参数注意事项

- 对于import参数，只支持用户传入当前Python函数或类所在的.py文件同级目录或子目录下的文件路径。
- 对于fn参数，如果fn不在当前注册UDF的.py文件中，那么需要同时在imports参数中添加fn定义的文件路径，例如：

```
from process import outer

con = ibis.fabric.connect(...)

# 注册UDAF
udf = con.udaf.python.register(
    outer(), #从外部引入的fn
    imports=["process.py"] #为fn添加文件路径
)
```

- 对于signature参数，目前允许用户传入，用户传入的优先级高于自动推断。当用户不传入时，进入参数/返回值类型自动推断逻辑，详情请参见**signature参数的类型推断**。
  - 对于注册Pyarrow UDF，无论用户是否传入signature参数，都需要依赖PyarrowVector，例如：

```
import fabric_data as fabric
import pyarrow as pa
import pyarrow.compute as pc

# === 用户传入signature，需要依赖PyarrowVector ===
def calculate_sum(
    prices: pa.ChunkedArray,
    quantities: pa.ChunkedArray,
) -> pa.ChunkedArray:
    return pc.multiply(prices, quantities)

con = ibis.fabric.connect(...)

# 注册UDF
udf = con.udf.pyarrow.register(
```

```
    fn=calculate_sum
    signature=fabric.Signature(
        parameters=[
            fabric.Parameter(name="price", annotation=fabric.PyarrowVector[float]),
            fabric.Parameter(name="quantity", annotation=fabric.PyarrowVector[int]),
        ],
        return_annotation=fabric.PyarrowVector[float],
    ),
)

# === 用户不传入signature，也需要依赖PyarrowVector ===
def calculate_sum(
    prices: fabric.PyarrowVector[float],
    quantities: fabric.PyarrowVector[int],
) -> fabric.PyarrowVector[float]:
    return fabric.PyarrowVector[float](pc.multiply(prices, quantities))

con = ibis.fabric.connect(...)

# 注册UDF
udf = con.udf.pyarrow.register(
    fn=calculate_sum
)
```

– 对于注册Pandas UDF，无论用户是否传入signature参数，都需要依赖PandasVector，例如：

```
import fabric_data as fabric
import pandas as pd

# === 用户传入signature，需要依赖PandasVector ===
def calculate_sum(
    prices: pd.Series,
    quantities: pd.Series,
) -> pd.Series:
    return pd.Series(prices * quantities, dtype=pd.Float64Dtype())

con = ibis.fabric.connect(...)

# 注册UDF
udf = con.udf.pandas.register(
    fn=calculate_sum
    signature=fabric.Signature(
        parameters=[
            fabric.Parameter(name="price", annotation=fabric.PandasVector[float]),
            fabric.Parameter(name="quantity", annotation=fabric.PandasVector[int]),
        ],
        return_annotation=fabric.PandasVector[float],
    ),
)

# === 用户不传入signature，也需要依赖PandasVector ===
def calculate_sum(
    prices: fabric.PandasVector[float],
    quantities: fabric.PandasVector[int],
) -> fabric.PandasVector[float]:
    return fabric.PandasVector[float](prices * quantities, dtype=pd.Float64Dtype())

con = ibis.fabric.connect(...)

# 注册UDF
udf = con.udf.pandas.register(
    fn=calculate_sum
)
```

- 对于volatility参数，3个枚举类型的含义是：
  – VolatilityType.VOLATILE：函数结果可能在任何时候都变化。
  – VolatilityType.STABLE：函数对于固定输入其结果在一次扫描里不变。
  – VolatilityType.IMMUTABLE：函数对于相同的输入总是输出相同的结果。

volatility参数不会影响函数下推执行，对于IMMUTABLE/STABLE/VOLATILE类型的函数，Python UDF函数都可以下推到DN执行。

- 对于packages参数，如果用户没有指定：
  – 注册Pyarrow UDF会自动使用后端环境的pyarrow版本。
  – 注册Pandas UDF会自动使用后端环境的pandas版本。

Builtin UDF 注册参数

注册Builtin UDF的作用是获得数据库已存在的函数的句柄，无实际注册的操作。

不管是显式注册还是隐式注册，不管是Scalar UDF、UDAF、UDTF，对于注册Builtin类型的UDF，目前都接受用户传入以下参数：

表 9-18 Builtin UDF 注册参数

注册参数	含义	类型	默认值
name	指定UDF实际数据库存储名称	str | None	None
database	指定UDF所在的LakeFormation数据库	str | None	None
fn	指定UDF原始的Python函数	Callable	None
signature	指定UDF函数签名和返回值类型	ibis.common.annotations.Signature | None	None

Builtin UDF 注册参数注意事项

对于signature参数，目前允许用户传入，用户传入的优先级高于自动推断。当用户不传入时，进入参数/返回值类型自动推断逻辑，详情请参见**signature参数的类型推断**。

signature 参数类型的推断

对于signature参数，允许用户传入参数/返回值类型，也允许用户不传入。

- 如果用户传入signature参数，不需要原始Python函数使用类型注解（type hints）语法，此时可以支持及时操作式的注册UDF。
- 如果用户不传入signature参数，推荐对原始Python函数使用类型注解（type hints）语法，此时不能支持及时操作式的注册UDF。

两种方式的区别总结如下：

表 9-19 signature 参数说明

signature参数	含义	需要原始Python函数使用类型注解语法	支持REPL及时操作
用户不传值	自动推断（推荐）	否，但是推荐使用	否
用户传值	指定传值	否	是

此处的及时操作指的是读取–求值–输出循环（Read–Eval–Print Loop, REPL），通常见于Python用户交互式终端（Terminal）中。

类型注解（type hints）语法由Python 3.5引入（PEP 484），在函数定义中，类型注解通过在参数名后加冒号（:）和类型，以及在参数列表末尾使用箭头（->）指定返回类型来实现，示例如下：

```
def greet(name: str) -> str:
    return f"Hello, {name}"
from typing import List, Dict, Optional

def process_data(data: List[int]) -> Dict[str, Optional[int]]:
    return {"max": max(data) if data else None}
```

对于Python/Pyarrow/Pandas UDF，注册时要求强数据类型，所有的参数/返回值都需要指定类型。如果用户不能通过原始Python函数的类型注解语法来指明，那么就需要用户主动使用signature参数指定ibis DataType。

对于Builtin UDF，注册时不要求强数据类型（因为数据库中已经注册了该UDF函数）。如果用户不能通过原始Python函数的类型注解语法来指明，那么推荐用户只写出参数名称，不写出类型；如果用户后续用到Builtin UDF的返回值（非Top SELECT UDF），那么需要指明函数返回值类型，必要时需要用户主动使用signature参数指定ibis DataType，如果不需要（Top SELECT UDF）则允许用户不写出函数返回值类型。

对于用户不传入signature参数，依赖自动推断时的总结如下：

表 9-20 signature 自动推断

注册UDF类型	参数类型	返回值类型
Python/Pyarrow/Pandas UDF	需要类型注解（type hints）语法指定。	需要类型注解（type hints）语法指定。
Builtin UDF	可以只写出参数名称，不写出类型。	后续使用返回值时需要类型注解（type hints）语法指定，否则不需要。

对于用户不传入signature参数，自动推断的情况，底层实现原理是inspect.signature。目前，接受用户传入以下参数/返回值类型：

表 9-21 接受的参数/返回值类型

Python类型	ibis DataType类型	对应DataArtsFabric SQL类型
DataType	DataType	-
type(None)	null	NULL
bool	Boolean	BOOLEAN
bytes	Binary	BYTEA
str	String	TEXT
numbers.Integral	Int64	BIGINT
numbers.Real	Float64	DOUBLE PRECISION
decimal.Decimal	Decimal	DECIMAL
datetime.datetime	Timestamp	TIMESTAMP/ TIMESTAMPTZ
datetime.date	Date	TIMESTAMP
datetime.time	Time	TIME
datetime.timedelta	Interval	INTERVAL
uuid.UUID	UUID	UUID
class	Struct	STRUCT
typing.Sequence, typing.Array	Array	ARRAY
typing.Mapping, typing.Map	Map	HSTORE
fabric_data.PyarrowVector[T]	T	T
fabric_data.PandasVector[T]	T	T

注意事项：

- Python内置int类型属于numbers.Integral的子类。
- Python内置float类型属于numbers.Real的子类。

上表中没有列出的Python类型，都是目前暂时不支持的自动转化类型。

对于用户不传入signature参数，同时也没有写出Python类型注解（type hints）语法的参数/返回值，目前自动推断采取如下的方式处理：

表 9-22 特殊参数类型处理

参数类型	生成匹配Pattern	Pattern效果
POSITIONAL_ONLY, KEYWORD_ONLY, POSITIONAL_OR_KEYWORD	ValueOf(None)	免除__signature__.validate。
VAR_POSITIONAL	TupleOf(pattern=pattern)	for-loop执行pattern。
VAR_KEYWORD	DictOf(key_pattern=InstanceOf(str), value_pattern=pattern)	for-loop执行pattern。
Return	ValueOf(Unknown)	提供UnknowScaclar, UnknownColumn作为UDF返回值向上返回。

inspect.signature对于参数类型（Parameter.kind）的分类如下：

表 9-23 inspect.signature 参数类型

参数类型	含义	示例代码	满足条件的参数
POSITIONAL_ONLY	仅限位置参数。	def func(a, /, b): pass	a
KEYWORD_ONLY	仅限关键字参数。	def func(a, *, b): pass	b
POSITIONAL_OR_KEYWORD	位置或关键字参数。	def func(a, b): pass	a, b
VAR_POSITIONAL	可变位置参数。	def func(*args): pass	args
VAR_KEYWORD	可变关键字参数。	def func(**kwargs): pass	kwargs

UDF 直接操作语法

在注册和使用分离的场景下，为使用者提供Scalar UDF、UDAF、UDTF的直接操作语法，使用者只需要知道UDF名称（name）、所在的数据库名称（database），就可以直接操作使用UDF。以下操作依赖Backend会话对象的udf属性进行。

signature(name, database=None)

描述：从后端DB中返回UDF的函数签名和返回值类型。

输入参数：

- name（str）- UDF函数名称。
- database（str）- UDF函数所属的数据库名称。

返回值类型：fabric_data.ibis.common.annotations.Signature - 已注册的UDF函数签名和返回值类型。

get(name, database=None)

描述：从后端DB中返回UDF函数。

输入参数：

- name（str）- UDF函数名称。
- database（str）- UDF函数所属的数据库名称。

返回值类型：Callable[..., ibis.expr.types.Value] - 已注册的UDF函数。

names(database=None)

描述：从后端DB中返回所有的UDF函数名称。

输入参数：

- database（str）： UDF函数所属的数据库名称。

返回值类型：List[str] - 所有已注册的UDF函数名称。

unregister(name, database=None)

描述：从后端DB中删除指定的UDF函数。

输入参数：

- database（str）： UDF函数所属的数据库名称。

返回值类型：None

UDF WITH ARGUMENTS 使用语法

不管是使用**UDF显式注册语法**，**UDF隐式注册语法**后返回的UDF算子，还是通过**UDF直接操作语法**返回的已注册UDF，目前都支持用户通过with_arguments方法传入参数，参数总体来说分为两类：

- 特殊含义的参数名，这些参数用来配置UDF运行时的资源、并发和执行上限时间，例如concurrency、timeout、dpu、apu，具体情况可见**UDF运行时配置列表**。所有的UDF类型目前都支持用户使用with_arguments传入配置参数。
- 一般的参数名，这些参数用来配置UDF初始化时的状态，只有Class UDF、Class UDTF、UDAF这些用户在__init__方法中定义了可选参数的情况下可以传入Scalar标量值。实现一次__init__初始化内部状态，后续多次使用。

表 9-24 WITH ARGUMENTS 语法适用情况

参数名	适用UDF类型	含义
特殊含义的参数名，如concurrency、timeout、dpu、apu	所有UDF类型，包括Scalar UDF、Class UDF、Vectorized UDF、UDTF、UDAF	配置UDF运行时的资源、并发和执行上限时间。
一般的参数名，由Python Class的__init__方法定义	Class UDF类型，包括Class UDF、Class UDTF、UDAF	配置UDF初始化状态，适合有内部缓存、初始化参数、模型对象等情况。

所有经过with_arguments方法传入的参数值都是Scalar标量，所有参数值整体作为一个**kwargs（Python字典类型）传入。

9.1.7.4 Util APIs

create_agg_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, comment=None, **kwargs)

描述：注册一个用户自定义聚合函数（UDAF）。

输入参数：

- udf (Callable)： 要注册的用户自定义聚合函数。
- input_type (InputType)： 函数的输入类型，可选值包括PYTHON、BUILTIN、PYARROW、PANDAS。
- database (str)： 函数所属的数据库名称。
- name (Optional[str])： 函数的名称，默认为None。
- signature (Optional[Signature])： 函数的签名定义，可选。
- imports (Optional[Union[str, Iterable[str]]])： 函数所需的导入模块，可以是单个字符串或字符串列表。
- packages (Optional[Union[str, Iterable[str]]])： 函数所需的依赖包，可以是单个包名或包名列表。
- comment (Optional[str])： 函数的注释说明，可选。
- **kwargs：其他可选的关键字参数。

返回值类型：

Callable： 注册后的函数。

create_scalar_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, imports=None, packages=None, comment=None, **kwargs)

描述：注册一个新的标量用户自定义函数（Scalar UDF）。

输入参数：

- udf (Callable)： 要注册的用户自定义标量函数。
- input_type (InputType)： 函数的输入类型，可选值包括PYTHON、BUILTIN、PYARROW、PANDAS。
- database (str)： 函数所属的数据库名称。
- name (Optional[str])： 函数的名称，默认为None。
- imports (Optional[Union[str, Iterable[str]]])： 函数所需的导入模块，可以是单个字符串或字符串列表。
- packages (Optional[Union[str, Iterable[str]]])： 函数所需的依赖包，可以是单个包名或包名列表。
- comment (Optional[str])： 函数的注释说明，可选。
- **kwargs：其他可选的关键字参数。

返回值类型：tuple[str, ...]。

create_table(name, /, *, schema, table_format, database=None, temp=False, external=False, overwrite=False, if_not_exists=False, partition_by=None, table_properties=None, location=None)

描述：在指定数据库中创建一个新表。

输入参数：

- name (str) - 表的名称。
- schema (ibis.Schema)： 表的定义。
- table_format (str)： 表的存储格式。
- database (Optional[str])： 要创建表的数据库名称。如果未指定，则使用默认数据库。
- temp (bool)： 如果为True，则创建临时表。
- external (bool)：如果为True，则创建外部表。
- overwrite (bool)： 如果为True，则在表已存在时覆盖原表。
- if_not_exists (bool)：如果为True，则仅在表不存在时创建表。
- partition_by (Optional[ibis.Schema])： 用于对表进行分区的架构。
- table_properties (Optional[dict])： 表的其他属性。
- location (Optional[str]： 表数据存储的位置。

返回值类型: Dataset。

create_table_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, comment=None, **kwargs)

描述：注册一个新的用户自定义表值函数（UDTF）。

输入参数：

- udf (Callable)： 要注册的用户自定义表值函数。
- input_type (InputType)： 函数的输入类型，可选值包括PYTHON、BUILTIN。

- database (str)： 函数所属的数据库名称。
- name (Optional[str])： 函数的名称，默认为None。
- signature (Optional[Signature])： 函数的签名定义，可选。
- imports (Optional[Union[str, Iterable[str]]])： 函数所需的导入模块，可以是单个字符串或字符串列表。
- packages (Optional[Union[str, Iterable[str]]])： 函数所需的依赖包，可以是单个包名或包名列表。
- comment (Optional[str])： 函数的注释说明，可选。
- **kwargs：其他可选的关键字参数。

返回值类型：

Callable： 注册后的函数。

delete_function(name, /, *, database=None, if_it_exists=False)

描述：从指定数据库中删除一个用户自定义函数。

输入参数：

- name (str)： 要删除的函数名称。
- database (Optional[str])： 要删除函数的数据库名称。如果未指定，则使用默认数据库。
- if_exists (bool)： 如果为True，则仅在函数存在时执行删除操作，避免因函数不存在而引发错误。

返回值类型：str。

delete_table(name, /, *, database=None, if_it_exists=False)

描述：从数据库中删除指定的Dataset。

输入参数：

- name (str)： 要删除的表的名称。
- database (Optional[str])： 要删除表的数据库名称。如果未提供，则使用默认数据库。
- if_it_exists (bool)： 如果设置为True，则仅在表存在时删除，避免因不存在而报错。

返回值类型： str。

describe_function(name, /, *, database=None)

描述：描述指定数据库中的函数。

输入参数：

- name (str)： 函数名称。
- database (Optional[str])： 函数所在的数据库名称。

返回值类型：str，函数的元数据信息。

describe_table(name, /, *, database=None)

描述：描述指定数据库中的表。

输入参数：

- name (str)： 函数名称。
- database (Optional[str])： 表所在的数据库名称。

返回值类型：str，表的元数据信息。

drop_table(name, *, database=None, force=False)

描述：从指定数据库中删除一个表。

输入参数：

- name (str)： 要删除的表的名称。
- database (Optional[str])： 要删除表的数据库名称。如果未提供，则使用默认数据库。
- force (bool)： 如果设置为True，则即使表包含数据也会强制删除。默认为False。

返回值类型： None。

get_function(name, *, database=None)

描述：从指定或默认数据库中获取一个用户自定义函数。

输入参数：

- func_name (str)： 要获取的用户自定义函数的名称。
- database (Optional[str]： 要获取函数的数据库名称。如果未提供，则使用默认数据库。

返回值类型： Callable，可调用的函数。

list_functions(database=None)

描述：列出指定数据库或默认数据库中的所有用户自定义函数。

输入参数：

database (Optional[str])： 要列出函数的数据库名称。如果未提供，则使用默认数据库。

返回值类型：pandas.DataFrame。

list_tables(database=None)

描述：列出指定数据库或默认数据库中的所有表。

输入参数：

database（Optional[str]）： 要列出表的数据库名称。如果为None，则使用默认数据库。

返回值类型：list[tuple]，包含表名称的元组列表。

load_dataset(name, *, database=None)

描述：从指定数据库加载数据集。

输入参数：

- name（str）： 要加载的Dataset的名称。
- database（Optional[str]）： 要加载Dataset的数据库名称。如果未提供，则使用默认数据库。

返回值类型：Dataset。

open_table(name, *, database=None)

描述：打开指定数据库中的表。

输入参数：

- name（str）： 要打开的表的名称。
- database（Optional[str]）： 包含该表的数据库名称。如果未提供，则使用默认数据库。

返回值类型 ： Table。

set_function_staging_workspace(*, obs_server, obs_bucket_name, obs_directory_base, access_key, secret_key, security_token=None)

描述：配置函数部署所需的临时工作区，包括设置必要的凭证和存储信息。

输入参数：

- obs_server（str）： OBS服务的地址。
- obs_bucket_name（str）： 要使用的OBS桶的名称。
- obs_directory_base（str）： OBS中用于存储工作区数据的基目录路径。
- access_key（str）： 用于身份验证的访问密钥。
- secret_key（str）： 用于身份验证的私密密钥。
- security_token（Optional[str]）： 用于身份验证的安全令牌，如果未提供则默认为None。

返回值类型 ： None。

sql(query, **kwargs)

描述：用于执行SQL查询语句，并返回查询结果。

输入参数：

- query（str）： 要执行的SQL查询语句。
- **kwargs（Any）： 可变关键字参数，用于传递查询的额外参数。

返回值类型 ： Any，返回SQL查询的结果。

explain_plan(verbose=True)

描述：生成作业执行计划。

输入参数：

verbose(bool) ： 如果为True，提供查询计划的详细说明；如果为false，则仅提供简单的执行计划。

返回值类型：str。

explain_performance()

描述：执行查询并生成计划，计划中包含执行过程中详细的性能指标。

输入参数：无。

返回值类型：str，详细的执行计划。

stats()

描述：获取当前会话的所有历史作业的执行记录。

输入参数：无。

返回值类型 ： list。

9.1.7.5 DataFrame APIs

本章节提供了类Pandas的Python DataFrame SDK，方便用户使用Python编写数据处理作业；同时利用DataArtsFabric SQL内核高效的计算能力，为数据科学家、AI工程师等提供了易用、高效的数据处理能力。

本特性基于Ibis Python DataFrame开源框架实现，将Ibis前端框架与DataArtsFabric SQL引擎对接。用户可基于熟悉的Ibis DataFrame API编写数据处理脚本，Ibis框架将Python API翻译为DataArtsFabric SQL引擎可执行的SQL语句并下发，从而实现计算逻辑在DataArtsFabric SQL引擎的高效处理。支持ibis的api使用，具体ibis api请参见**Ibis官方文档**。

快速开始

以下代码使用ibis库连接DataArtsFabric数据湖并执行数据查询，将结果转换为DataFrame格式的基本语法。

示例仅供参考，请您根据实际情况进行修改。

关于Ibis更详细的用法，请参见**Ibis官方文档**。

```
import ibis  # 导入ibis依赖
import logging
import os
con = ibis.fabric.connect(
    fabric_endpoint=os.getenv("fabric_endpoint"), # 指定服务的区域，区域查询请参见地区和终端节点
    fabric_endpoint_id=os.getenv("fabric_endpoint_id"), # 查询endpoint_id
    fabric_workspace_id=os.getenv("fabric_workspace_id"), # 获取workspace_id
    lf_catalog_name=os.getenv("lf_catalog_name"), # 连接指定的Catalog
```

```
    lf_instance_id=os.getenv("lf_instance_id"), # LakeFormation服务的实例ID，详情请参见获取LakeFormation实例id
    access_key=os.getenv("access_key"), # 获取ak/sk
    secret_key=os.getenv("secret_key"),
    use_single_cn_mode=True, # 表示是否开启单CN模式
    logging_level=logging.INFO, # 设置日志级别
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"), # obs中udf的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"), # obs的名称
    obs_server=os.getenv("obs_server"), # obs访问地址,详情请参见终端节点（Endpoint）和访问域名
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key")
)
t = con.table("table_name", database="db") # 通过连接到后端获取table表信息，建立表对象
t.select("cola") # 查询表字段
df = t.execute() # 将DataFrame转为SQL，传输到后端执行，并且返回Pandas DataFrame格式的结果
```

不带 UDF 的 DF 示例

下文以tpch的query1为例，展示DataFrame的用法。

查询SQL为：

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM
    lineitem
WHERE
    l_shipdate <= CAST('1998-09-02' AS date)
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,    l_linestatus;
```

对应的DataFrame逻辑如下：

```
import ibis # 导入ibis依赖
con = ibis.fabric.connect(...)
t = con.table("lineitem", database="tpch") # 通过连接到后端获取table表信息，建立表对象
q = t.filter(t.l_shipdate <= add_date("1998-12-01", dd=-90))
discount_price = t.l_extendedprice * (1 - t.l_discount)
charge = discount_price * (1 + t.l_tax)
q = q.group_by(["l_returnflag", "l_linestatus"])
q = q.aggregate(
    sum_qty=t.l_quantity.sum(),
    sum_base_price=t.l_extendedprice.sum(),
    sum_disc_price=discount_price.sum(),
    sum_charge=charge.sum(),
    avg_qty=t.l_quantity.mean(),
    avg_price=t.l_extendedprice.mean(),
    avg_disc=t.l_discount.mean(),
    count_order=lambda t: t.count(),
)
q = q.order_by(["l_returnflag", "l_linestatus"])
sql = q.compile() # 将DataFrame编译为sql字符串df = q.execute() # 执行表达式并且返回结果集
```

带 Scalar UDF 的 DF 示例

场景描述

在AI数据工程中，数据预处理是一个关键步骤，通常需要对存储在数据库中的数据进行复杂的清洗、转换和特征工程操作。然而，传统的数据预处理逻辑往往在数据库外部通过Python脚本实现，这会导致大量数据在数据库和Python环境之间传输，不仅增加了计算开销，还无法充分利用数据库的分布式计算能力。通过在数据库内核中实现Python UDF，并在ibis-fabric中增加Python UDF显式，隐式调用接口，可以将数据预处理逻辑直接嵌入到数据库查询中，减少数据传输，提升整体处理效率。

对于复杂业务逻辑，例如使用机器学习模型对特征数据表进行按行过滤。传统函数式UDF定义方式无法有效支持模型初始化和资源管理需求，因为模型超参数和资源（如模型文件加载）需要在函数执行前一次性配置，而函数式UDF无法高效管理此类初始化逻辑和资源释放操作。通过以Class形式定义Python UDF，用户可以在__init__方法中配置模型超参数和初始化资源（如加载模型），并在__call__方法中执行推理逻辑，同时通过__del__方法释放资源（如关闭文件或连接），从而实现高效的资源管理和业务逻辑处理。

约束限制

- Python UDF功能约束限制如下：
  - 仅支持Python语言编写的用户自定义函数。
  - Python环境依赖管理由用户自行负责，需确保各依赖库版本兼容。
  - 运行时环境要求Python版本为3.11。
- Python Class UDF功能约束限制如下：
  - 类必须定义__init__方法，且其参数名称需与调用时的参数名称保持一致。
  - 类必须定义__call__成员方法，该方法为UDF的主函数入口。
  - 类可按需定义__del__成员方法，且该方法不支持传入参数。
  - 调用class型UDF时，__init__方法的参数仅支持传入常量，不支持传入表的数据列或其他表达式。

结合DataFrame使用Scalar UDF是推荐的标准用法，此时整个Scalar UDF的外部必须要包围DataFrame的SELECT方法。

经过注册后返回的值是DataFrame中的一个UDF算子。此时，该算子可以被多个DataFrame表达式多次调用，示例如下：

```
import ibis
import ibis_fabric as fabric
from ibis_fabric.udf import RegisterType

def transform_json(ts: float, msg: str) -> str:
    import json
    from car_bu.parser_core import ParseObjects, dict_to_object
    if msg == '0_msg':
        return json.dumps({"time_stamp": 0.0, "msg": {}})
    else:
        d = dict_to_object(json.loads(msg))
        return json.dumps({"time_stamp": ts/10, "msg": ParseObjects(d)})

con = ibis.fabric.connect(...)

# 显式注册transform_json
transform_json_udf = con.udf.python.register(
    transform_json,
    database="your-database",
    imports=['car_bu/parser_core.py'],
```

```
    packages=['json'],
    register_type=RegisterType.OBS
)

# 结合DataFrame的SELECT方法，第一次使用transform_json
t = con.table("your-table", database="your-database")
expression = t.select(transform_json_udf(t.ts, t.msg).name("json column"))
df = expression.execute()

# 结合DataFrame的SELECT方法，第二次使用transform_json
t = con.table("your-table", database="your-database")
filtered = t.filter(...)
local_view = filtered.select(...).mutate(...)
span_base = local_view.select(...).filter(...)
span_part2 = ibis.memtable(...)
union_part = span_base.union(span_part2)
window = ibis.window(...)
final_span = union_part.union(...).order_by(...).select(...)
result = final_span.mutate(...).select(...).filter(...).order_by(...)
result = result.select(transform_json_udf(result.ts, result.msg).name("json column"))
df = result.execute()
```

对于Python类的Scalar Class UDF，使用方式和Python函数的Scalar UDF稍有不同：UDF算子接受__call__方法参数；with_arguments接受__init__方法参数。完整示例如下：

```
import abc
import ibis
import ibis_fabric as fabric
from ibis_fabric.udf import RegisterType
from sentencepiece import SentencePieceProcessor
from typing import Sequence


class SPManager(Base):
    def __init__(self, model_file: str, *, bos: bool = False, eos: bool = False, reverse: bool = False) -> None:
        sp = SentencePieceProcessor(model_file=model_file)
        opts = ":".join(opt for opt, flag in (("bos", bos), ("eos", eos), ("reverse", reverse)) if flag)
        sp.set_encode_extra_options(opts)
        self.spm = sp

    def __call__(self, input_text: str) -> Sequence[str]:
        return self.spm.encode(input_text, out_type=str)

con = ibis.fabric.connect(...)

# 显式注册SPManager
sentencepiece_udf = con.udf.python.register(
    SPManager,
    database="your-database",
    imports=['test_model.model'],
    packages=['sentencepiece'],
    register_type=RegisterType.STAGED
)

# 结合DataFrame的SELECT方法，使用SPManager
t = con.table("your-table", database="your-database")
expression = t.select(sentencepiece_udf(t.data).with_arguments(model_file="test_model.model", bos=True,
eos=True).name("pieces column"))
df = expression.execute()
```

带 UDAF 的 DF 示例

结合DataFrame使用UDAF是推荐的标准用法，此时整个UDAF的外部必须要包围DataFrame的aggregate方法。

UDAF只能出现在SELECT、ORDER BY、HAVING 3种从句中。

经过注册后返回的值是DataFrame中的一个UDF算子。此时，该算子可以被多个DataFrame表达式多次调用，with_arguments接受__init__方法参数，示例如下：

```
import ibis
import ibis_fabric as fabric
from ibis_fabric.udf import RegisterType

import typing
import json

class PythonTopK:
    def __init__(self, k: int):
        # 用户在 with_arguments 里传入的参数
        self.k = k
        self._agg_state = []

    @property
    def aggregate_state(self) -> typing.Sequence[int]:
        # 聚合中间态（当前 top k 列表）
        return self._agg_state

    def accumulate(self, value: int) -> None:
        # 累积阶段：逐行更新 top k
        if value is not None:
            self._agg_state.append(value)
            self._agg_state.sort(reverse=True)
            self._agg_state = self._agg_state[: self.k]

    def merge(self, other_state: typing.Sequence[int]) -> None:
        # 合并阶段：跨分区合并 top k
        merged = self._agg_state + other_state
        merged.sort(reverse=True)
        self._agg_state = merged[: self.k]

    def finish(self) -> str:
        # 收尾阶段：返回最终 top k 结果，以JSON字符串形式返回
        return json.dumps(self._agg_state)

con = ibis.fabric.connect(...)

# 显式注册PythonTopK
topk_udf = con.udf.python.register(
    PythonTopK,
    packages=["json"],
    database="your-database",
    register_type=RegisterType.STAGED
)

t = con.table("your-table", database="your-database")

# 结合DataFrame的aggregate方法，第一次使用PythonTopK，UDAF在SELECT
expression = t.aggregate(
    top_values=topk_udaf(t.amount).with_arguments(k=3),
    by=[t.category],
)
df = expression.execute()

# 结合DataFrame的aggregate方法，第二次使用PythonTopK，UDAF在ORDER BY
g = t.aggregate(
    sort_key=topk_udaf(t.amount).with_arguments(k=10),
    by=[t.category],
)
expression = g.order_by(g.sort_key.desc())
df = expression.execute()

# 结合DataFrame的aggregate方法，第三次使用PythonTopK，UDAF在HAVING
expression = t.group_by(t.category)
     .aggregate(top_values=topk_udaf(t.amount).with_arguments(k=5))
```

```
    .filter(lambda s: s.top_values.isnull() == False)
df = expression.execute()
```

直接使用带 Scalar UDF 的 DF 示例

场景描述

在大数据处理场景中，用户在使用DataFrame进行数据处理时，常常需要通过用户自定义函数（UDF）来实现复杂的数据计算逻辑。然而，当前系统中UDF的注册与调用耦合在一起，用户无法在注册后单独查看或删除已注册的UDF，这在团队协作开发或动态管理UDF时带来了诸多不便。为了解决这一问题，本次需求通过新增Backend.udf系列API，支持用户在运行时动态查看、调用和删除UDF，从而提升UDF管理的灵活性和开发效率。

约束限制

UDF直接调用/查看/删除功能约束限制如下：

用户必须先获得Backend（Fabric）连接，才能调用Backend UDF Registery API。

对于负责类型的支持依赖fabric内核对复杂类型的支持。

在注册和使用分离的场景下，对于UDF的使用者，允许用户通过Fabric Backend提供的Backend.udf接口直接获得、查看、删除Scalar UDF，示例如下：

```
import ibis
import fabric_data as fabric

con = ibis.fabric.connect(...)

# 查看数据库中已有的UDF列表
udfs = con.udf.names(database="your-database")

if "transform_json" in udfs:
    # 直接获得UDF，确认数据库中已有transform_json函数
    transform_json_udf = con.udf.get(name="transform_json", database="your-database")
    # 结合DataFrame的SELECT方法，使用transform_json
    expression = t.select(transform_json_udf(t.ts, t.msg).name("json column"))
    df = expression.execute()
    # 删除UDF
    con.udf.unregister("transform_json", database="your-database")

if "SPManager" in udfs:
    # 直接获得UDF，确认数据库中已有SPManager类
    sentencepiece_udf = con.udf.get(name="SPManager", database="your-database")
    # 结合DataFrame的SELECT方法，使用SPManager
    expression = t.select(sentencepiece_udf(t.data).with_arguments(model_file="test_model.model",
bos=True, eos=True).name("pieces column"))
    df = expression.execute()
    # 删除UDF
    con.udf.unregister("SPManager", database="your-database")
```

直接使用带 UDAF 的 DF 示例

在注册和使用分离的场景下，对于UDAF的使用者，允许用户通过Fabric Backend提供的Backend.udaf接口直接获得、查看、删除UDAF，示例如下：

```
import ibis
import fabric_data as fabric

con = ibis.fabric.connect(...)

# 查看数据库中已有的UDAF列表
udfs = con.udaf.names(database="your-database")

if "pythontopk" in udfs:
```

```
# 直接获得UDAF，确认数据库中已有pythontopk函数
topk_udaf = con.udaf.get(name="pythontopk", database="your-database")
# 结合DataFrame的aggregate方法，使用PythonTopK
expression = t.aggregate(
    top_values=topk_udaf(t.amount).with_arguments(k=3),
    by=[t.category],
)
df = expression.execute()
# 删除UDF
con.udaf.unregister("pythontopk", database="your-database")
```

DataFrame create_table 函数使用说明

create_table用于在DataArtsFabric创建一个表，函数签名如下：

```
def create_table(
    self,
    name: str,
    obj: Optional[Union[ir.Table, pd.DataFrame, pa.Table, pl.DataFrame, pl.LazyFrame]] = None,
    *,
    schema: Optional[ibis.Schema] = None,
    database: Optional[str] = None,
    temp: bool = False,
    external: bool = False,
    overwrite: bool = False,
    partition_by: Optional[ibis.Schema] = None,
    table_properties: Optional[dict] = None,
    store: Optional[str] = None,
    location: Optional[str] = None
)
```

表 9-25 参数说明

参数名称	类型	是否必须	说明
name	str	是	要创建的表名。
obj	ir.Table| pd.DataFrame| pa.Table| pl.DataFrame| pl.LazyFrame	否	用于填充表格的数据；必须至少指定obj或schema之一（当前不支持在创建表的时候插入数据）。
schema	sch.SchemaLike	否	要创建的表的架构；必须至少指定obj或schema之一。
database	str	否	创建表的数据库的名称；如果未传递，则使用当前数据库。
temp	bool	否	是否创建为临时表。默认为False。
external	bool	否	是否创建为外表。默认为False。
overwrite	bool	否	如果为True，表已存在则替换表。默认为False（当前不支持覆盖表）。

参数名称	类型	是否必须	说明
partition_by	sch.SchemaLike	否	指定分区列，分区列中出现的列不能出现在表的普通列描述中。
table_properties	dict	否	表级别可选参数设置，支持参数范围如下参考**表8-35**。
store	str	否	表存储格式，支持ORC、PARQUET、HUDI、ICEBERG四种存储格式。
location	str	否	表存储路径，必须为合法OBS路径，支持OBS对象桶和并行文件系统。 ● 如果该路径为OBS对象桶路径，则该表只读，否则该表支持读写。 ● 如果创建表类型为托管表（Managed Table），则不允许指定表存储路径，表存储路径由系统指定为Schema路径下与表名同名路径，且要求该路径在建表时为空。

10 函数参考

10.1 自定义函数

10.1.1 DataArtsFabric SQL UDF 概述

用户自定义函数（User-Defined Function，简称UDF）是指由用户根据特定需求自定义的函数，用于扩展数据库或数据处理系统的功能。UDF可以执行各种操作，如数据转换、复杂计算、数据聚合等，这些操作可能超出了系统提供的内置函数的能力。

UDF的主要用途包括：

- 扩展功能：UDF允许用户扩展数据库或数据处理系统的功能，执行内置函数不支持的操作。
- 复杂计算：UDF可用于执行复杂的数学或逻辑计算，这些计算可能涉及多个步骤和条件。
- 数据转换：UDF可用于数据清洗和转换，如格式化日期、转换字符串或编码数据。
- 业务逻辑封装：UDF可将业务逻辑封装在函数中，使得数据处理更加模块化和可重用。
- 性能优化：在某些情况下，通过UDF将复杂的计算逻辑移到数据库服务器上执行，可以减少数据传输和提高性能。

从类型上来分，UDF包括以下三种主要类型：

表 10-1 UDF 类型

UDF类型	输入值	返回值	描述
标量函数（Scalar UDF，一般简称UDF）	一行数据	一个标量值	Scalar UDF是一种用户自定义函数。Scalar UDF输入是数据库中一行数据，输出是一个标量值；对于每一行的输入数据处理是独立的，不依赖于其他行的数据。这类函数适用于需要对数据进行特定计算或转换的场景。本特性引入的Scalar UDF允许用户使用Python的语法在数据库中对数据进行计算或转换。
聚合函数（Aggregate UDF，简称UDAF）	多行数据	一个聚合值	UDAF是一种聚合类型的用户自定义函数。UDAF的输入是一组数据行，而输出是对整组数据进行聚合后得到的单个结果值。UDAF通常用于分组统计、聚合计算等场景。
表值函数（Table-Valued UDF，简称UDTF）	一行数据	多行结果	UDTF是一种以表作为输出的用户自定义函数，其输入是一行数据，输出则是一个包含多行多列的表结构。UDTF通常用于行生成、数据展开、多值解析等场景

约束限制

- UDF编程语言仅支持Python。
- UDF的参数不支持设置默认值，如果创建函数时参数或返回值带有精度，不进行精度检测。
- 不支持函数重载。

10.1.2 UDF 开发（Python）

10.1.2.1 约束限制

Python UDF即使用Python语言编写的自定义函数，当DataArtsFabric SQL提供的内置函数无法支撑客户的业务实现时， 客户可以参考本文中的开发流程及使用示例，自行编写代码逻辑创建Python UDF，以满足多样化业务需求。

使用Python UDF，有以下限制：

- UDF运行时环境的Python版本仅支持3.11系列，且建议版本使用3.11.9
- 函数体和代码包归档路径，两者必须定义一个。

10.1.2.2 数据类型映射

Python和DataArtsFabric SQL数据类型的映射关系如下表所示：

表 10-2 Python 和 DataArtsFabric SQL 数据类型的映射关系

Python类型	DataArtsFabric SQL类型
int	BIGINT
bool	BOOLEAN
float	DOUBLE PRECISION
str	TEXT
bytes	BYTEA
datetime.date	DATE
datetime.time	TIME
datetime.datetime	TIMESTAMP
decimal.Decimal	NUMERIC

10.1.2.3 代码归档包的组织结构

当用户的业务场景比较复杂，UDF所涉及的代码较多时，推荐用户以文件压缩包的形式注册函数，将UDF依赖的所有相关代码文件统一归档到一个压缩包里，上传至OBS后，在创建函数时指定压缩包的存储路径。压缩包里代码文件结构要求如下：

- 必须包含文件"{FUNCTION_NAME}_source.py"，该文件认为是函数的主入口文件。且文件内容需要使用CLOUDPICKLE序列化为十六进制。
- 主函数需要import的模块所属的代码文件，需要正确组织结构。
- 压缩包格式仅支持zip。

压缩包示例如下：

```
my_udf.zip/
├── my_udf_source.py
├── my_helper.py
└── other/
    └── utils.py
```

10.1.2.4 开发注意事项

- DataArtsFabric SQL提供了三种注册UDF的方式：**ibis-fabric SDK显示注册、隐式注册**以及**SQL的DDL注册**，推荐用户使用ibis-fabric sdk显示注册，该方式有以下两个优势：
  - SDK提供的注册接口已完全遵守DataArtsFabric SQL DDL的使用规范，只要正确调用接口即可，能避免违反约束规范而导致UDF不能正常使用的问题。
  - SDK提供了更多的约束检查，例如Python版本的检测，自动整理UDF代码包等。

- Python UDF的三方依赖包列表需要保证lib间的兼容性，并建议所有依赖包显示指定稳定的版本。用户本地调试UDF时，建议使用华为源所提供的三方包来搭建调试环境，以避免UDF代码在本地环境可运行，而在DataArtsFabric SQL里运行时出现版本无法安装、版本不兼容等问题。
- 用户在准备压缩包，使用CLOUDPICKLE序列化主函数时，确保本地Python版本为3.11.9且cloudpickle版本为3.0.0，以避免DataArtsFabric SQL运行环境里因为版本不一致导致函数体解析失败的问题。
- 用户只能使用OBS并行文件系统的桶，来存储UDF代码压缩包，且需要在LakeFormation上赋予IAM用户读的权限。

10.1.2.5 WITH ARGUMENTS 语法

WITH ARGUMENTS语法扩展了现有用户定义函数（UDF）的功能，主要作用是允许用户在UDF运行时设置初始状态。通过这一功能，用户可以：

- 配置运行时资源和并发度：用户可以指定UDF运行时所需的资源（如内存、CPU等）以及并发执行的实例数量，从而优化性能和资源利用率。
- 设置类的初始化参数：对于以类（class）形式定义的UDF，用户可以通过WITH ARGUMENTS语法传递初始化构造参数，从而在UDF实例化时设置特定的初始状态。

使用注意事项如下：

- WITH ARGUMENTS支持几个固定的参数配置，具体的参数信息请参考**表10-3**。对于class的初始化参数名称不能与固定参数同名，参数名称是大小写敏感的。
- 入参列表仅支持常量。
- 以class形式定义的UDF函数体，用户按需定义class的__init__和__del__成员方法，但必须定义__call__成员方法，该方法为主函数入口。
- min_concurrency和max_concurrency需要同时设置，设置成功后表示开启自适应并发度。

表 10-3 UDF 运行时配置列表

参数名	参数描述	默认值
concurrency	UDF的运行并发度，取值范围[1,1024]。	1
min_concurrency	UDF的运行最小并发度，取值范围[1,1024]。	0
max_concurrency	UDF的运行最大并发度，取值范围[1,1024]。	0
timeout	UDF执行的超时时间，取值范围[1,3600]，单位为s。	1000
dpu	UDF运行时单个actor的资源规格，例如dpu=1，表示规格为1U4G，其取值可以是介于0.33到1.0之间的分数值，也可以是1到32间的整数值，需要注意的是，系统会将输入值四舍五入，保留两位小数。	1

参数名	参数描述	默认值
apu	指定异构资源的类型和规格，例如"NPU/.+/1"	null

10.1.2.6 自适应并行

场景描述

在AI数据工程中，面对大量数据处理的场景，需要通过并行调用actor来提升数据处理的效率，进行分布式计算。但固定并行度依赖于调参经验以及多次测试反馈，为提升用户体验和减少调参时间，提供自适应并行来提高使用UDF的易用度。

约束限制

功能约束限制如下：

- 执行UDF函数时必须指定min_concurrency和max_concurrency有效值。
- 如果当前batch的行数小于计算出的并行度，实际拉起的actor数将等于行数。否则实际拉起的actor数将等于当前计算的并行度。

当执行UDF时，同时设置min_concurrency和max_concurrency参数后，将会开启自适应并行功能。

UDF以min_concurrency并行度开始执行所涉及的数据batch。每一轮batch执行时，都会计算UDF有效计算时长占总时间的占比。当每次batch执行中，UDF有效计算占比达到80%，当前并行度将会按照2的步长增加并行度，直至最大增加至max_concurrency并行度后，后续不变化。当某一轮占比不到80%时，则下一轮并行度将保持不变。

开启自适应并行特性的UDF，各个actor由于先后执行数据的批次不同，因此统计信息存在较大差别。对于自适应并行场景下的统计信息查看，请参考**Python UDF性能调优**。

10.1.2.7 Scalar UDF

开发Python Scalar UDF，总体流程如下：

步骤1 根据用户的业务编写和调试Python代码。

开发UDF代码，除了业务本身代码以外，代码中还需要包含如下信息：

- 导入模块：如果使用ibis-fabric sdk开发UDF，则需要安装和导入ibis相关模块。
- 函数的主入口Handler：自定义代码中，必须清晰定义函数的主入口，Handler方法是唯一的。
- 函数签名：Handler方法的定义中，需要显示指定输入参数和返回值的数据类型。
- 其它：如果使用ibis-fabric sdk注册函数，则需要导入ibis-fabric模块的注册接口，并编写注册的代码；如果使用SQL语法注册函数，则需要编写序列化Handler的代码。

以实现一个两数求和的Python UDF为例，代码文件结构如下：

```
my_udf/
├── udf_main.py
```

```
├── register.py
├── serialize.py
└── other/
     └── util.py
```

代码示例如下：

```
# udf_main.py包含了UDF的主函数，即Handler
from other.util import printLog
import numpy as np
def pysum(arg1: int, arg2: int) -> int:
    printLog(arg1, arg2)
    return np.sum(np.array([arg1, arg2]))

# util.py是函数的依赖文件
def printLog(a,b):
    print(f"computing the sum of {a} and {b}")

# register.py
import huawei-ibis-fabric as fabric
import ibis   # 导入ibis依赖
con = ibis.fabric.connect(...)
from udf_main import pysum
udf = con.udf.python.register(pysum, imports=["other/util.py"], packages=["numpy"])

# serialize.py 序列化handler的代码如下
import cloudpickle
import pickle
from udf_main import pysum

if __name__ == "__main__":
    with open("pysum_source.py", "w") as file:
        my_bytes = cloudpickle.dumps(pysum, protocol=pickle.HIGHEST_PROTOCOL).hex()
        file.write(my_bytes)
```

步骤2 注册Python UDF。注册Python UDF有以下两种方式：

- 方式一：使用**ibis-fabric sdk显示注册**（推荐）。

  ibis-fabric显式注册是用户指定ibis的backend为Fabric，通过register/register_from_file接口来实现，调用即注册。以上述步骤1的例子为例，在与udf_main.py同级目录下新增register.py，详细流程及代码片段如下：

  a. 使用ibis-fabric创建会话。

  ```
  import huawei-ibis-fabric as fabric
  import ibis   # 导入ibis依赖
  con = ibis.fabric.connect(...)  # 创建会话
  ```

  b. （可选方案1）使用register接口注册，其中fn=pysum，是必填参数，表示UDF的主函数；其他参数均可选，请根据业务需要选填，例如该例中，imports表示UDF所依赖的文件。packages表示UDF所依赖的三方库。

  ```
  from udf_main import pysum
  udf = con.udf.python.register(pysum, imports=["other/util.py"], packages=["numpy"])
  ```

  c. （可选方案2）使用register_from_file接口注册，其中file_path="udf_main.py"，表示主函数所在的文件路径；func_name="pysum"，表示主函数名称。file_path和func_name这两个参数必填，其他参数请根据业务需要选填。

  ```
  udf = con.udf.python.register_from_file("udf_main.py", "pysum", imports=["other/util.py"],
  packages=["numpy"])
  ```

  您可以按需选择可选方案1或可选方案2，ibis-fabric会自动根据注册接口的入参将UDF所依赖的文件全部打包到一个压缩包中，并上传至OBS桶里。压缩包文件结构如下：

  ```
  pysum.zip/
  ├── pysum_source.py  # 该文件的内容，是使用cloudpickle将UDF主函数体序列化后并编码为十六进制。
  └── other/
       └── util.py  # 该文件是UDF所依赖的文件，打包时会保留其与主函数所在文件的相对目录结构。
  ```

- 方式二：使用任意DataArtsFabric的客户端工具连接DataArtsFabric服务，下发UDF的DDL注册。

  该方式是直接使用DataArtsFabric SQL提供的UDF语法，通过编写DDL的形式注册，使用时请遵守具体的**使用规范**。

  a. 参照步骤1中的示例，参考**代码归档包的组织结构**，将主函数序列化后，将pysum_source.py和依赖的相关文件一起打包，压缩包结构和上述方式一中的压缩包一致。

  b. 参考UDF**语法格式**，编写UDF的DDL。对于函数的入参和返回值类型参考**数据类型映射**，根据函数本身来定义。

```
CREATE FUNCTION "your_schema".pysum(arg1 bigint, arg2 bigint)
RETURNS bigint
LANGUAGE PYTHON
RUNTIME_VERSION = '3.11'
HANDLER = 'pysum'
imports = ('https://test_bucket.obs.cn-north-7.huawei.com/calculate_0/calculate_0.zip')
packages=('numpy');
```

步骤3 查看Python UDF。

DataArtsFabric SQL提供show functions查看特定schema下的所有函数，且提供了describe function function_name方法查看特定函数的详细信息，具体的使用方式请参考**SHOW**和**DESCRIBE**。

步骤4 调用Python UDF。

使用DataArtsFabric任意客户端，下发查询即可，例如：

```
select "your_schema".pysum(3,4);
```

步骤5 查看Python UDF性能指标及调优。

DataArtsFabric SQL支持了对Python UDF运行时的性能监控，以及提供了接口，可配置UDF运行时的资源规格和并发度，具体使用方式请参考**Python UDF性能调优**。

----结束

10.1.2.8 UDTF

UDTF整体开发流程可参考Scalar UDF，需要注意以下几点：

- Handler目前仅支持返回迭代器，推荐使用yield。
- UDTF不支持开启并发和向量化计算。
- 当Handler出现死循环，无限输出结果时，FabricSQL会根据超时时间自动关闭程序，默认超时时间为1000s。

示例

1. 创建UDTF：

```
CREATE FUNCTION py_generate_series(start int, endnum int, step int)
RETURNS TABLE(a int)
LANGUAGE python
RUNTIME_VERSION='3.11.2'
HANDLER='generate_series'
as $$
def generate_series(start, endnum, step):
    current = start
    while current <= endnum:
        yield current
        current += step
$$;
```

2. 执行UDTF：
```
SELECT py_generate_series(1,4,1);
 py_generate_series
----------------------
                    1
                    2
                    3
                    4
(4 rows)
```

10.1.2.9 UDAF

UDAF整体开发流程可以参考Scalar UDF，需要注意以下几点：

- Handler指定的必须是Python Class，且Python Class中必须包括accumulate、aggregate_state、merge、finish 4个方法。具体的方法含义见**表1 注册UDAF时的特殊实例方法**。
- UDAF要求用户实现的accumulate、aggregate_state、merge方法是无副作用的（side-effect free），即这些方法的返回结果只取决于输入参数和聚合状态本身，而不应依赖于方法被调用的次数、顺序或外部环境。
- UDAF的finish方法不论是单CN模式，还是多DN模式执行，对于每个分组（由GROUP BY从句指定）都只会在全局调用1次。
- UDAF要求用户在定义非STRICT时，能够在accumulate方法中正确处理NULL值输入，即Python中的None值。
- UDAF不支持作为窗口函数（window function）使用，即不支持在OVER从句中使用UDAF。
- UDAF不支持开启并发和向量化计算。

示例

1. 创建UDAF：
```
CREATE AGGREGATE FUNCTION pysum(x INT)
RETURNS INT
STRICT
LANGUAGE PYTHON
RUNTIME_VERSION='3.11.2'
HANDLER = 'CalculateSum'
AS $$
import typing
class CalculateSum:
    def __init__(self) -> None:
        self.sum = 0

    def accumulate(self, x: int) -> None:
        self.sum += x

    def finish(self) -> int:
        return self.sum

    @property
    def aggregate_state(self) -> typing.Dict[str, typing.Any]:
        return {
            "sum": self.sum,
        }

    def merge(self, other_state: typing.Dict[str, typing.Any]) -> None:
        self.sum += other_state.get("sum")
$$;
```
2. 执行UDAF：
```
SELECT pysum(x) FROM UNNEST(ARRAY[1, 2, NULL, 3]) AS x;
 pysum
```

```
-------
     6
(1 row)
```

