

云数据库 GaussDB

# 特性指南（分布式\_V2.0-3.x）

文档版本 01  
发布日期 2025-08-22



版权所有 © 华为云计算技术有限公司 2025。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

## 商标声明



HUAWEI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

|                       |           |
|-----------------------|-----------|
| <b>1 物化视图</b>         | <b>1</b>  |
| 1.1 全量物化视图            | 1         |
| 1.1.1 概述              | 1         |
| 1.1.2 使用              | 1         |
| 1.1.3 支持和约束           | 2         |
| 1.2 增量物化视图            | 2         |
| 1.2.1 概述              | 2         |
| 1.2.2 使用              | 3         |
| 1.2.3 支持和约束           | 4         |
| <b>2 设置密态等值查询</b>     | <b>5</b>  |
| 2.1 密态等值查询概述          | 5         |
| 2.2 使用 gsql 操作密态数据库   | 8         |
| 2.3 使用 JDBC 操作密态数据库   | 10        |
| 2.4 配置阶段安全增强          | 14        |
| 2.5 密态支持函数/存储过程       | 17        |
| <b>3 分区表</b>          | <b>19</b> |
| 3.1 大容量数据库            | 19        |
| 3.1.1 大容量数据库背景介绍      | 19        |
| 3.1.2 表分区技术           | 19        |
| 3.1.3 数据分区查找优化        | 20        |
| 3.1.4 数据分区运维管理        | 20        |
| 3.2 分区表介绍             | 21        |
| 3.2.1 基本概念            | 21        |
| 3.2.1.1 分区表 (母表)      | 21        |
| 3.2.1.2 分区 (分区子表、子分区) | 22        |
| 3.2.1.3 分区键           | 23        |
| 3.2.2 分区策略            | 23        |
| 3.2.2.1 范围分区          | 23        |
| 3.2.2.2 哈希分区          | 25        |
| 3.2.2.3 列表分区          | 26        |
| 3.2.2.4 分区表对导入操作的性能影响 | 27        |
| 3.2.3 分区基本使用          | 28        |

|                                    |           |
|------------------------------------|-----------|
| 3.2.3.1 创建分区表.....                 | 28        |
| 3.2.3.2 分区表 DML 查询语句.....          | 29        |
| 3.3 分区表查询优化.....                   | 31        |
| 3.3.1 分区剪枝.....                    | 31        |
| 3.3.1.1 分区表静态剪枝.....               | 31        |
| 3.3.1.2 分区表动态剪枝.....               | 35        |
| 3.3.1.2.1 PBE 动态剪枝.....            | 35        |
| 3.3.1.2.2 参数化路径动态剪枝.....           | 38        |
| 3.3.2 分区索引.....                    | 42        |
| 3.4 分区表运维管理.....                   | 45        |
| 3.4.1 新增分区.....                    | 46        |
| 3.4.1.1 向范围分区表新增分区.....            | 46        |
| 3.4.1.2 向列表分区表新增分区.....            | 46        |
| 3.4.2 删除分区.....                    | 46        |
| 3.4.3 交换分区.....                    | 47        |
| 3.4.4 清空分区.....                    | 48        |
| 3.4.5 分割分区.....                    | 49        |
| 3.4.5.1 对范围分区表分割分区.....            | 49        |
| 3.4.5.2 对列表分区表分割分区.....            | 50        |
| 3.4.6 合并分区.....                    | 50        |
| 3.4.7 移动分区.....                    | 51        |
| 3.4.8 重命名分区.....                   | 51        |
| 3.4.8.1 对分区表重命名分区.....             | 51        |
| 3.4.8.2 对 Local 索引重命名索引分区.....     | 51        |
| 3.4.9 分区表行迁移.....                  | 52        |
| 3.4.10 分区表索引重建/不可用.....            | 52        |
| 3.4.10.1 索引重建/不可用.....             | 52        |
| 3.4.10.2 Local 索引分区重建/不可用.....     | 53        |
| 3.5 分区表系统视图&DFX.....               | 53        |
| 3.5.1 分区表相关系统视图.....               | 53        |
| 3.5.2 分区表相关内置工具函数.....             | 54        |
| <b>4 存储引擎.....</b>                 | <b>57</b> |
| 4.1 存储引擎体系架构.....                  | 57        |
| 4.1.1 存储引擎体系架构概述.....              | 57        |
| 4.1.1.1 静态编译架构.....                | 57        |
| 4.1.1.2 通用数据库服务层.....              | 58        |
| 4.1.2 设置存储引擎.....                  | 58        |
| 4.1.3 存储引擎更新说明.....                | 59        |
| 4.1.3.1 GaussDB Kernel 503 版本..... | 59        |
| 4.1.3.2 GaussDB Kernel R2 版本.....  | 59        |
| 4.2 Astore 存储引擎.....               | 60        |
| 4.2.1 Astore 简介.....               | 60        |

|   |           |
|---|-----------|
| 4.3 Ustore 存储引擎.....  | 60        |
| 4.3.1 Ustore 简介.....  | 60        |
| 4.3.1.1 Ustore 特性与规格.....   | 61        |
| 4.3.1.1.1 特性约束.....   | 61        |
| 4.3.1.1.2 存储规格.....   | 61        |
| 4.3.1.2 使用 Ustore 进行测试.....   | 61        |
| 4.3.1.3 Ustore 的最佳实践.....   | 62        |
| 4.3.1.3.1 怎么配置 init_td 大小.....  | 62        |
| 4.3.1.3.2 怎么配置 fillfactor 大小.....   | 62        |
| 4.3.1.3.3 统计信息收集.....   | 63        |
| 4.3.1.3.4 在线校验功能.....   | 63        |
| 4.3.1.3.5 怎么配置回滚段大小.....  | 63        |
| 4.3.2 存储格式.....   | 64        |
| 4.3.2.1 Relation.....   | 64        |
| 4.3.2.1.1 PbRCR(Page base Row Consistency Read) Heap 多版本管理.....                                   | 65        |
| 4.3.2.1.2 PbPCR Heap 可见性机制.....   | 65        |
| 4.3.2.1.3 Heap 空间管理.....  | 65        |
| 4.3.2.2 Index.....  | 66        |
| 4.3.2.2.1 RCR(Row Consistency Read) UB-tree 多版本管理.....  | 66        |
| 4.3.2.2.2 RCR UB-tree 可见性机制.....  | 66        |
| 4.3.2.2.3 UB-tree 增删改查.....   | 67        |
| 4.3.2.2.4 UB-tree 空间管理.....   | 68        |
| 4.3.2.3 Undo.....   | 69        |
| 4.3.2.3.1 回滚段管理.....  | 69        |
| 4.3.2.3.2 文件组织结构.....   | 69        |
| 4.3.2.3.3 Undo 空间管理.....  | 69        |
| 4.3.3 Ustore 事务模型.....  | 69        |
| 4.3.3.1 事务提交.....   | 70        |
| 4.3.3.2 事务回滚.....   | 70        |
| 4.3.4 闪回恢复.....   | 70        |
| 4.3.4.1 闪回查询.....   | 71        |
| 4.3.4.2 闪回表.....  | 73        |
| 4.3.4.3 闪回 DROP/TRUNCATE.....   | 75        |
| 4.3.5 常用视图工具.....   | 83        |
| 4.3.6 常见问题及定位手段.....  | 86        |
| 4.3.6.1 snapshot too old.....   | 86        |
| 4.3.6.1.1 长事务阻塞 Undo 空间回收.....  | 86        |
| 4.3.6.1.2 大量回滚事务拖慢 Undo 空间回收.....   | 87        |
| 4.3.6.2 storage test error.....   | 87        |
| 4.3.6.3 备机读业务报错:"UBTreeSearch::read_page has conflict with recovery, please try again later"..... | 88        |
| <b>5 Foreign Data Wrapper.....</b>  | <b>90</b> |
| 5.1 file_fdw.....   | 90        |

|                              |           |
|------------------------------|-----------|
| <b>6 逻辑复制.....</b>           | <b>92</b> |
| 6.1 逻辑解码.....                | 92        |
| 6.1.1 逻辑解码概述.....            | 92        |
| 6.1.2 逻辑解码选项.....            | 96        |
| 6.1.3 使用 SQL 函数接口进行逻辑解码..... | 102       |
| 6.1.4 使用流式解码实现数据逻辑复制.....    | 103       |

# 1 物化视图

物化视图实际上就是一种特殊的物理表，物化视图是相对普通视图而言的。普通视图是虚拟表，应用的局限性较大，任何对视图的查询实际上都是转换为对SQL语句的查询，性能并没有实际上提高。而物化视图实际上就是存储SQL所执行语句的结果，起到缓存的效果。

## 1.1 全量物化视图

### 1.1.1 概述

全量物化视图仅支持对创建好的物化视图做全量更新，而不支持做增量更新。创建全量物化视图语法和CREATE TABLE AS语法一致，不支持对全量物化视图指定NodeGroup创建。

### 1.1.2 使用

#### 语法格式

- 创建全量物化视图  
`CREATE MATERIALIZED VIEW [ view_name ] AS { query_block };`
- 刷新全量物化视图  
`REFRESH MATERIALIZED VIEW [ view_name ];`
- 删除物化视图  
`DROP MATERIALIZED VIEW [ view_name ];`
- 查询物化视图  
`SELECT * FROM [ view_name ];`

#### 示例

```
-- 准备数据
CREATE TABLE t1(c1 int, c2 int);
INSERT INTO t1 VALUES(1, 1);
INSERT INTO t1 VALUES(2, 2);

-- 创建全量物化视图
gaussdb=# CREATE MATERIALIZED VIEW mv AS select count(*) from t1;
CREATE MATERIALIZED VIEW

-- 查询物化视图结果
```

```
gaussdb=# SELECT * FROM mv;
count
-----
      2
(1 row)

-- 再次向物化视图中基表插入数据
gaussdb=# INSERT INTO t1 VALUES(3, 3);

-- 对全量物化视图做全量刷新
gaussdb=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- 查询物化视图结果
gaussdb=# SELECT * FROM mv;
count
-----
      3
(1 row)

-- 删除物化视图
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

## 1.1.3 支持和约束

### 支持场景

- 大体上，全量物化视图所支持的查询范围与CREATE TABLE AS语句一致。
- 创建全量物化视图可以指定分布列。
- 可以在全量物化视图上创建索引。
- 支持analyze、explain。

### 不支持场景

- 全量物化视图不支持NodeGroup。
- 不可对物化视图做增删改操作，只支持查询语句。

### 约束

- 创建全量物化视图所使用的基表必须在所有DN上有定义，基表所属nodegroup必须为installation group。
- 全量物化视图的刷新、删除过程中会给基表加高级别锁，若物化视图的定义涉及多张表，需要注意业务逻辑，避免死锁产生。

## 1.2 增量物化视图

### 1.2.1 概述

增量物化视图顾名思义就是可以对物化视图增量刷新，需要用户手动执行语句，刷新物化视图在一段时间内的增量数据。与全量创建物化视图不同在于目前增量物化视图所支持场景较小，目前物化视图创建语句仅支持基表扫描语句或者UNION ALL语句。

## 1.2.2 使用

### 语法格式

- 创建增量物化视图  
CREATE INCREMENTAL MATERIALIZED VIEW [ view\_name ] AS { query\_block };
- 全量刷新物化视图  
REFRESH MATERIALIZED VIEW [ view\_name ];
- 增量刷新物化视图  
REFRESH INCREMENTAL MATERIALIZED VIEW [ view\_name ];
- 删除物化视图  
DROP MATERIALIZED VIEW [ view\_name ];
- 查询物化视图  
SELECT \* FROM [ view\_name ];

### 示例

```
-- 准备数据
CREATE TABLE t1(c1 int, c2 int);
INSERT INTO t1 VALUES(1, 1);
INSERT INTO t1 VALUES(2, 2);

-- 创建增量物化视图
gaussdb=# CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM t1;
CREATE MATERIALIZED VIEW

-- 插入数据
gaussdb=# INSERT INTO t1 VALUES(3, 3);
INSERT 0 1

-- 增量刷新物化视图
gaussdb=# REFRESH INCREMENTAL MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- 查询物化视图结果
gaussdb=# SELECT * FROM mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
(3 rows)

-- 插入数据
gaussdb=# INSERT INTO t1 VALUES(4, 4);
INSERT 0 1

-- 全量刷新物化视图
gaussdb=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- 查询物化视图结果
gaussdb=# select * from mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)

-- 删除物化视图
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

## 1.2.3 支持和约束

### 支持场景

- 单表查询语句
- 多个单表查询的UNION ALL
- 在物化视图上创建索引
- 对物化视图做Analyze操作
- 增量物化视图会继承基表NodeGroup创建（检查各个基表是否在同一个NodeGroup，并基于这个NodeGroup进行创建）。

### 不支持场景

- 物化视图中不支持带Stream计划，多表join连接计划以及subquery计划。
- 不支持WITH子句、GROUP BY子句、ORDER BY子句、LIMIT子句、WINDOW子句、DISTINCT算子、AGG算子以及不支持除UNION ALL外的子查询。
- 除少部分ALTER操作外，不支持对物化视图中基表做绝大多数DDL操作。
- 创建物化视图不可指定物化视图分布列。
- 不可对物化视图做增删改操作，只支持查询语句。
- 不支持用临时表/hashbucket/unlog/分区表创建物化视图，只支持hash分布表。
- 不支持物化视图嵌套创建（物化视图上创建物化视图）。
- 不支持UNLOGGED类型的物化视图，不支持WITH语法。

### 约束

- 物化视图定义如果为UNION ALL，则其中每个子查询需使用不同的基表，且各基表分布列相同。物化视图的分布列会自动推导且与各基表相同。
- 物化视图定义的列必须包含基表的所有分布列。
- 增量物化视图的创建、全量刷新、删除过程中会给基表加高级别锁，若物化视图的定义为UNION ALL，需要注意业务逻辑，避免死锁产生。

# 2 设置密态等值查询

## 2.1 密态等值查询概述

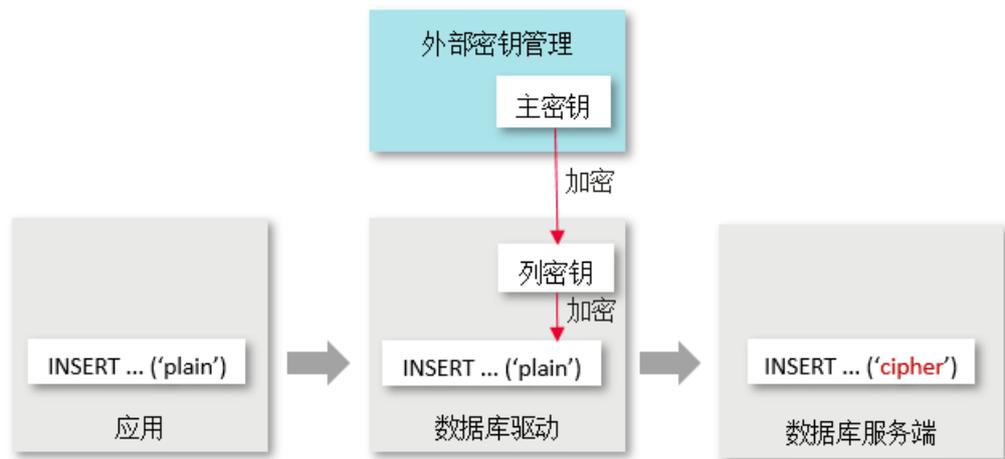
随着企业数据上云，数据的安全隐私保护面临越来越严重的挑战。密态数据库将解决数据整个生命周期中的隐私保护问题，涵盖网络传输、数据存储以及数据运行状态；更进一步，密态数据库可以实现云化场景下的数据隐私权限分离，即实现数据拥有者和实际数据管理者的数据读取能力分离。密态等值查询将优先解决密文数据的等值类查询问题。

### 加密模型

全密态数据库使用多级加密模型，加密模型中涉及3个对象：数据、列密钥和主密钥，以下是对3个对象的介绍：

- **数据：**
  - 指SQL语法中包含的数据，比如INSERT...VALUES ('data')语法中包含'data'。
  - 指从数据库服务端返回的查询结果，比如执行SELECT...语法返回的查询结果。

密态数据库会在驱动中对SQL语法中属于加密列的数据进行加密，对数据库服务端返回的属于加密列的查询结果进行解密。
- **列密钥：**数据由列密钥进行加密，列密钥由数据库驱动生成或由用户手动导入，列密钥密文存储在数据库服务端。
- **主密钥：**列密钥由主密钥加密，主密钥由外部密钥管理者生成并存储。数据库驱动会自动访问外部密钥管理者，以实现列密钥进行加解密。



## 整体流程

在使用全密态数据库的过程中，主要流程包括如下五个阶段，本节介绍整体流程，[使用gsq操作密态数据库](#)、[使用JDBC操作密态数据库](#)章节介绍详细使用流程。

**一、生成主密钥阶段：**首先，需在华为云密钥服务中，生成主密钥，并准备用于访问主密钥的参数，比如密钥ID、密钥服务账号等。

**二、配置阶段：**在数据库驱动侧，设置用于访问主密钥的参数。

**三、执行DDL阶段：**在本阶段，用户可使用密态数据库的密钥语法依次定义主密钥和列密钥，然后定义表并指定表中某列为加密列。定义主密钥和列密钥的过程中，需访问上一阶段生成的主密钥。

**四、执行DML阶段：**在创建加密表后，用户可直接执行包含但不限于INSERT、SELECT、UPDATE、DELETE等语法，数据库驱动会自动根据上一阶段的加密定义自动对加密列中的数据进行加解密。

**五、清理阶段：**依次删除加密表、列密钥和主密钥。

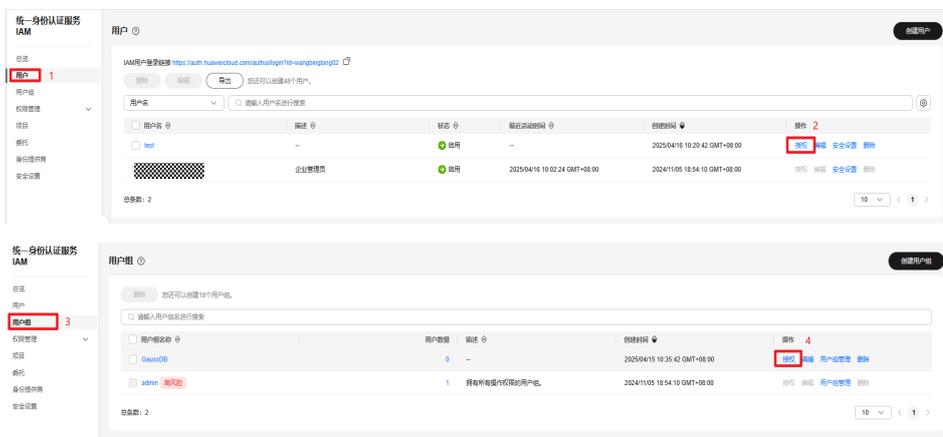


## 生成主密钥阶段

首次使用密态数据库需要执行准备阶段步骤，后续跳过该阶段即可。

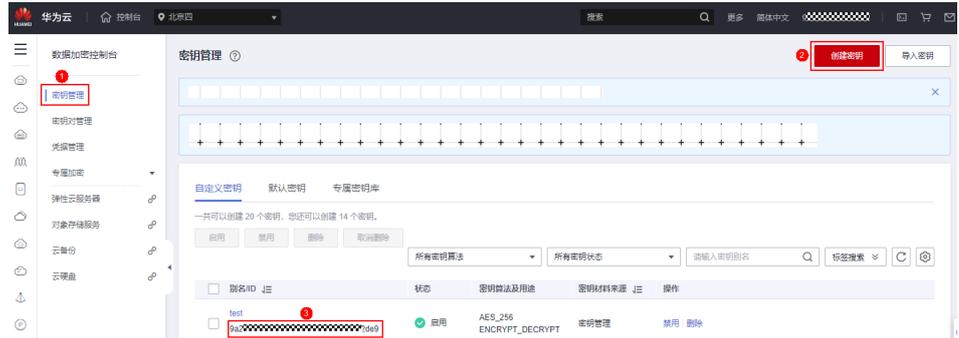
密态数据库支持使用不同的外部密钥来管理主密钥，根据场景选择其中一种即可。

- 华为云场景
  - a. 用户需先在打开华为云官网，注册账号，登录账号。
  - b. 在华为云中搜索“统一身份认证服务”，进入该服务，如图所示选择“用户”功能，并创建一个IAM用户，设置IAM密码，并为IAM用户关联一个“用户组”，然后对用户组授权使用“数据加密服务”权限。



- c. 接下来，请重新回到登录页面，登录方式选择为“IAM用户”，使用新创建的IAM用户进行登录。后续操作均由该IAM用户完成。
- d. 在华为云中搜索“数据加密服务”，进入该服务，如下图所示选择“密钥管理”功能，并通过“创建密钥”按钮创建密钥，密钥创建成功后，可看到每

个密钥都具有1个密钥ID。请记住该密钥ID，在后续执行DDL阶段中创建主密钥语法时，需使用该密钥ID。



- e. 本步生成的密钥即密态数据库中使用的**主密钥**，该密钥将由**华为云密钥管理服务**存储。以后执行与加解密相关的SQL语句时，数据库驱动会通过**华为云的restful接口**自动访问该密钥。

## 配置阶段

配置访问外部密钥的参数

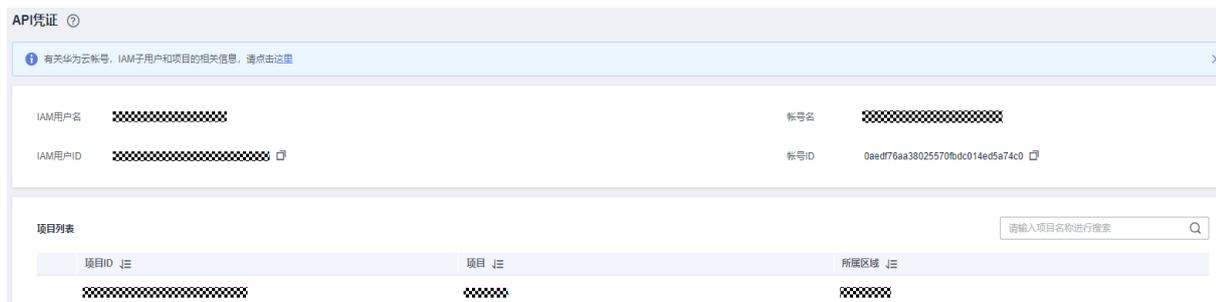
- 华为云场景

通过环境变量配置如下信息：

```
export HUAWEI_KMS_INFO='iamUrl=https://iam.{项目}.myhuaweicloud.com/v3/auth/tokens, iamUser={IAM用户名}, iamPassword={IAM用户密钥}, iamDomain={账号名}, kmsProject={项目}'
```

在**华为云控制台**中，点击右上角用户名，并进入“**我的凭证**”，可看到下图所示页面，该页面可获取上述所需参数：项目、IAM用户名、账号名。另外，请记住本页面的项目ID，在后续执行DDL阶段中创建主密钥语法时，需使用该项目ID。

图 2-1 华为云参数获取页面



# 示例

```
export HUAWEI_KMS_INFO='iamUrl=https://iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens, iamUser=test_user, iamPassword=*****, iamDomain=test_account, kmsProject=cn-north-4'
```

## 2.2 使用 gsql 操作密态数据库

### 执行 SQL 语句

在执行本节的SQL语句之前，请确保已完成前两阶段：**创建主密钥阶段**、**配置阶段**。

本节以完整的执行流程为例，介绍如何使用密态数据库语法，包括三个阶段：**使用DDL阶段**、**使用DML阶段**、**清理阶段**。

### 步骤1 连接数据库，并通过-C参数开启全密态开关

```
gsql -p PORT -d DATABASE -h HOST -U USER -W PASSWORD -r -C
```

### 步骤2 定义主密钥

在生成主密钥阶段，密钥服务已生成并存储主密钥，执行本语法只是将主密钥的相关信息存储在数据库中，方便以后访问。该语法详细格式参考：《开发指南》中“SQL参考 > SQL语法 > CREATE CLIENT MASTER KEY”章节。

```
gaussdb=# CREATE CLIENT MASTER KEY cmk1 WITH ( KEY_STORE = huawei_kms , KEY_PATH = '{KMS服务器地址}/{密钥ID}', ALGORITHM = AES_256);  
CREATE CLIENT MASTER KEY
```

- 参数获取：生成主密钥阶段介绍了如何获取如下参数：KMS服务器地址、密钥ID。

KEY\_PATH示例：<https://kms.cn-north-4.myhuaweicloud.com/v1.0/0b59929e8100268a2f22c01429802728/kms/9a262917-8b31-41af-a1e0-a53235f32de9>

### 步骤3 定义列密钥

列密钥由上一步定义的主密钥加密。详细语法参考：《开发指南》中“SQL参考 > SQL语法 > CREATE COLUMN ENCRYPTION KEY”章节。

```
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek1 WITH VALUES (CLIENT_MASTER_KEY = cmk1,  
ALGORITHM = AES_256_GCM);
```

### 步骤4 定义加密表

本示例中，通过语法指定表中name和credit\_card为加密列。

```
gaussdb=# CREATE TABLE creditcard_info (  
  id_number int,  
  name text encrypted with (column_encryption_key = cek1, encryption_type = DETERMINISTIC),  
  credit_card varchar(19) encrypted with (column_encryption_key = cek1, encryption_type =  
  DETERMINISTIC));  
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id_number' as the distribution column by  
default.  
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.  
CREATE TABLE
```

### 步骤5 对加密表进行其他操作

```
-- 向加密表写入数据。  
gaussdb=# INSERT INTO creditcard_info VALUES (1,'joe','6217986500001288393');  
INSERT 0 1  
gaussdb=# INSERT INTO creditcard_info VALUES (2,'joy','6219985678349800033');  
INSERT 0 1  
  
-- 从加密表中查询数据。  
gaussdb=# select * from creditcard_info where name = 'joe';  
 id_number | name | credit_card  
-----+-----+-----  
          1 | joe | 6217986500001288393  
  
-- 更新加密表中数据。  
gaussdb=# update creditcard_info set credit_card = '80000000011111111' where name = 'joy';  
UPDATE 1  
  
-- 向表中新增一列加密列。  
gaussdb=# ALTER TABLE creditcard_info ADD COLUMN age int ENCRYPTED WITH  
(COLUMN_ENCRYPTION_KEY = cek1, ENCRYPTION_TYPE = DETERMINISTIC);  
ALTER TABLE  
  
-- 从表中删除一列加密列。  
gaussdb=# ALTER TABLE creditcard_info DROP COLUMN age;  
ALTER TABLE  
  
-- 从系统表中查询主密钥信息。  
gaussdb=# SELECT * FROM gs_client_global_keys;
```

```
global_key_name | key_namespace | key_owner | key_acl | create_date
-----+-----+-----+-----+-----
cmk1           | 2200 | 10 | | 2021-04-21 11:04:00.656617
(1 rows)

-- 从系统表中查询列密钥信息。
gaussdb=# SELECT column_key_name,column_key_distributed_id,global_key_id,key_owner FROM
gs_column_keys;
column_key_name | column_key_distributed_id | global_key_id | key_owner
-----+-----+-----+-----
cek1           | 760411027 | 16392 | 10
(1 rows)

-- 查看表中列的元信息。
gaussdb=# \d creditcard_info
Table "public.creditcard_info"
Column | Type | Modifiers
-----+-----+-----
id_number | integer |
name | text | encrypted
credit_card | character varying | encrypted
```

## 步骤6 清理阶段

```
-- 删除加密表。
gaussdb=# DROP TABLE creditcard_info;
DROP TABLE

-- 删除列密钥。
gaussdb=# DROP COLUMN ENCRYPTION KEY cek1;
DROP COLUMN ENCRYPTION KEY

-- 删除主密钥。
gaussdb=# DROP CLIENT MASTER KEY cmk1;
DROP CLIENT MASTER KEY
```

----结束

## 2.3 使用 JDBC 操作密态数据库

### 获取 JDBC 驱动包

1. 获取JDBC驱动包，JDBC驱动获取及使用可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发”章节。

密态数据库支持的JDBC驱动包为gsjdbc4.jar、opengaussjdbc.jar、gscejdbc.jar。

- gsjdbc4.jar: 主类名为“org.postgresql.Driver”，数据库连接的url前缀为“jdbc:postgresql”。
- opengaussjdbc.jar: 主类名为“com.huawei.opengauss.jdbc.Driver”，数据库连接的url前缀为“jdbc:opengauss”。
- gscejdbc.jar（目前仅支持EulerOS操作系统）：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，密态场景推荐使用此驱动包。

2. 配置LD\_LIBRARY\_PATH

密态场景使用JDBC驱动包时，需要先设置环境变量LD\_LIBRARY\_PATH。例如：

```
export LD_LIBRARY_PATH=/tmp:$LD_LIBRARY_PATH
```

- 使用gscejdbc.jar驱动包时，gscejdbc.jar驱动包中密态数据库需要的依赖库会自动拷贝到该路径下，并在开启密态功能连接数据库的时候加载。

## 说明

使用gscejdbc.jar时，jvm加载class文件需要依赖系统的libstdc++库，若开启密态则gscejdbc.jar会自动拷贝密态数据库依赖的动态库（包括libstdc++库）到用户设定的LD\_LIBRARY\_PATH路径下。如果依赖库与现有系统库版本不匹配，则首次运行仅部署依赖库，再次调用后即可正常使用。

- 使用opengaussjdbc.jar或gsjdbc4.jar时，需要同时解压包名为GaussDB-Kernel\_数据库版本号\_操作系统版本号\_64bit\_libpq.tar.gz的压缩包解压到指定目录，并将lib文件夹所在目录路径，添加至LD\_LIBRARY\_PATH环境变量中。

## 须知

全密态场景使用JDBC驱动包时需要有System.loadLibrary权限，以及环境变量LD\_LIBRARY\_PATH中第一优先路径的文件读写权限，建议使用独立目录作为全密态依赖库的存放路径。若在执行的时候指定java.library.path，需要与LD\_LIBRARY\_PATH的第一优先路径保持一致。

## 执行 SQL 语句

在执行本节的SQL语句之前，请确保已完成前两阶段：**准备阶段、配置阶段**。

本节以完整的执行流程为例，介绍如何使用密态数据库语法，包括三个阶段：使用DDL阶段、使用DML阶段、清理阶段。

JDBC开发中与非密态场景操作一致的部分请参考《开发指南》中“应用程序开发教程 > 基于JDBC开发”章节。

- 密态数据库连接参数

enable\_ce: String类型。其中enable\_ce=0表示不开启全密态开关，enable\_ce=1表示支持密态等值查询基本能力。

```
// 以下用例以gscejdbc.jar驱动为例，如果使用其他驱动包，仅需修改驱动类名和数据库连接的url前缀。  
// gsjdbc4.jar: 主类名为“org.postgresql.Driver”，数据库连接的url前缀为“jdbc:postgresql”。  
// opengaussjdbc.jar: 主类名为“com.huawei.opengauss.jdbc.Driver”，数据库连接的url前缀为“jdbc:opengauss”。  
// gscejdbc.jar: 主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”
```

```
public static void main(String[] args) {  
    // 驱动类。  
    String driver = "com.huawei.gaussdb.jdbc.Driver";  
    // 数据库连接描述符。enable_ce=1表示支持密态等值查询基本能力。  
    String sourceURL = "jdbc:gaussdb://127.0.0.1:8000/postgres?enable_ce=1";  
    // 在环境变量USER、PASSWORD分别配置用户名密码。  
    String username = System.getenv("USER");  
    String passwd = System.getenv("PASSWORD");  
    Connection conn = null;  
    try {  
        // 加载驱动  
        Class.forName(driver);  
        // 创建连接  
        conn = DriverManager.getConnection(sourceURL, username, passwd);  
        System.out.println("Connection succeed!");  
        // 创建语句对象  
        Statement stmt = conn.createStatement();  
  
        // 定义客户端主密钥  
        // KEY_PATH格式请参考：《开发指南》中“SQL参考 > SQL语法 > CREATE CLIENT MASTER KEY”  
        // 章节  
        // 华为云场景下，KEY_PATH中需使用项目ID与密钥ID，在准备阶段已介绍如何获取密钥ID，配置阶
```



表 2-1 新增 org.postgresql.jdbc.PgConnection 函数接口

| 方法名   | 返回值类型                    | 支持JDBC 4 |
|---|--------------------------|----------|
| decryptData(String ciphertext, Integer len, String schema, String table, String column) | ClientLogicDecryptResult | Yes      |

参数说明：

- **ciphertext**  
需要解密的密文。
- **len**  
密文长度。当取值小于实际密文长度时，解密失败。
- **schema**  
加密列所属schema名称。
- **table**  
加密列所属table名称。
- **column**  
加密列所属column名称。

#### 📖 说明

下列场景可以解密成功，但不推荐：

- 密文长度入参比实际密文长。
- schema.table.column指向其他加密列。此时将返回被指向的加密列的原始数据类型。

表 2-2 新增 org.postgresql.jdbc.clientlogic.ClientLogicDecryptResult 函数接口

| 方法名                | 返回值类型   | 描述                          | 支持JDBC4 |
|--------------------|---------|-----------------------------|---------|
| isFailed()         | Boolean | 解密是否失败，若失败返回True，否则返回False。 | Yes     |
| getErrMsg()        | String  | 获取错误信息。                     | Yes     |
| getPlaintext()     | String  | 获取解密后的明文。                   | Yes     |
| getPlaintextSize() | Integer | 获取解密后的明文长度。                 | Yes     |
| getOriginalType()  | String  | 获取加密列的原始数据类型。               | Yes     |

```
// 通过非密态连接、逻辑解码等其他方式获得密文后，可使用该接口对密文进行解密
import org.postgresql.jdbc.PgConnection;
import org.postgresql.jdbc.clientlogic.ClientLogicDecryptResult;
```

```
// conn为密态连接
// 调用密态PgConnection的decryptData方法对密文进行解密，通过列名称定位到该密文的所属加密列，并返回
// 其原始数据类型
ClientLogicDecryptResult decrypt_res = null;
decrypt_res = ((PgConnection)conn).decryptData(ciphertext, ciphertext.length(), schemaname_str,
        tablename_str, colname_str);
// 检查返回结果类解密成功与否，失败可获取报错信息，成功可获得明文及长度和原始数据类型
if (decrypt_res.isFailed()) {
    System.out.println(String.format("%s\n", decrypt_res.getErrMsg()));
} else {
    System.out.println(String.format("decrypted plaintext: %s size: %d type: %s\n", decrypt_res.getPlaintext(),
        decrypt_res.getPlaintextSize(), decrypt_res.getOriginalType()));
}
```

## 执行加密表的预编译 SQL 语句

```
// 调用Connection的prepareStatement方法创建预编译语句对象。
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO creditcard_info VALUES (?, ?, ?);");
// 调用PreparedStatement的setShort设置参数。
pstmt.setInt(1, 2);
pstmt.setString(2, "joy");
pstmt.setString(3, "6219985678349800033");
// 调用PreparedStatement的executeUpdate方法执行预编译SQL语句。
int rowcount = pstmt.executeUpdate();
// 调用PreparedStatement的close方法关闭预编译语句对象。
pstmt.close();
```

## 执行加密表的批处理操作

```
// 调用Connection的prepareStatement方法创建预编译语句对象。
Connection conn = DriverManager.getConnection("url","user","password");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO creditcard_info (id_number, name,
credit_card) VALUES (?,?);");
// 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。
int loopCount = 20;
for (int i = 1; i < loopCount + 1; ++i) {
    pstmt.setInt(1, i);
    pstmt.setString(2, "Name " + i);
    pstmt.setString(3, "CreditCard " + i);
    // Add row to the batch.
    pstmt.addBatch();
}
// 调用PreparedStatement的executeBatch方法执行批处理。
int[] rowcount = pstmt.executeBatch();
// 调用PreparedStatement的close方法关闭预编译语句对象。
pstmt.close();
```

## 2.4 配置阶段安全增强

### 安全地设置环境变量

环境变量HUAWEI\_KMS\_INFO中包含敏感信息，建议使用如下设置方式：

1. 设置临时环境变量：使用密态数据库时，通过export命令设置环境变量；使用完，即通过unset命令清理环境变量。该方法中操作系统日志可能会记录敏感信息，建议使用进程级环境变量或使用JDBC接口对connection连接参数进行设置。
2. 设置进程级环境变量：在应用程序代码中，通过编程接口设置环境变量，不同编程语言设置示例如下：
  - a. C/C++：setenv(name, value)。
  - b. Go：os.Setenv(name, value)。

- c. java暂不支持设置进程级环境变量，仅支持通过JDBC接口设置connection连接参数。

## 外部密钥服务的身份验证

当数据库驱动访问华为云密钥管理服务时，为避免攻击者伪装为密钥服务，在数据库驱动与密钥服务建立https连接的过程中，可通过CA证书验证密钥服务器的合法性。为此，需提前配置CA证书，如果未配置，将不会验证密钥服务的身份。配置方法如下：

华为云场景下，需在环境变量中增加如下参数：

```
export HUAWEI_KMS_INFO='其他参数, iamCaCert=路径/IAM的CA证书文件, kmsCaCert=路径/KMS的CA证书文件'
```

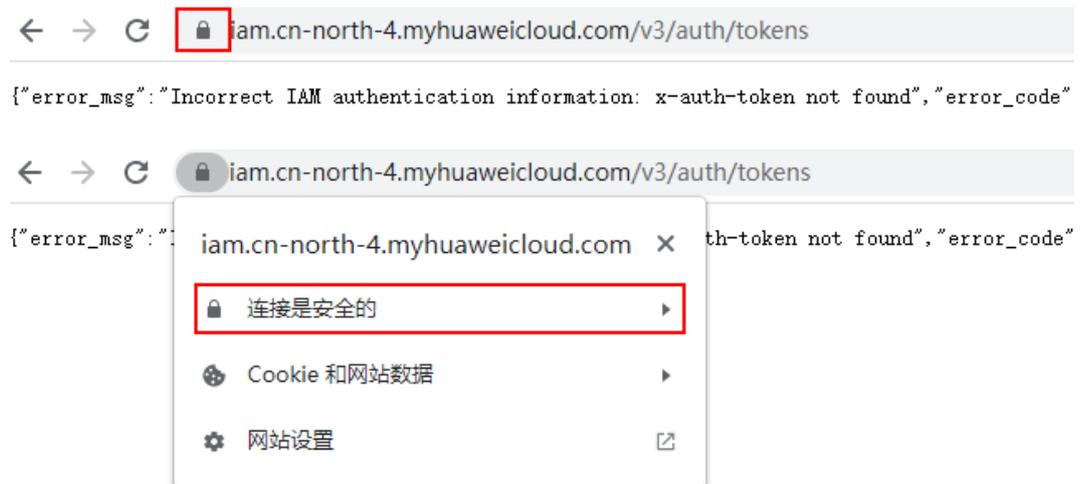
大部分浏览器均会自动下载网站对应的CA证书，并提供证书导出功能。虽然很多网站也提供自动下载CA证书的功能，但可能因本地环境中存在代理或网关，导致CA证书无法正常使用。所以，建议借助浏览器下载CA证书。下载方式如下：

### ⚠ 注意

由于我们使用restful接口访问密钥服务，当我们在浏览器输入接口对应的url时，可忽略下述**步骤2**中的失败页面，因为即使在失败的情况下，浏览器早已提前自动下载CA证书。

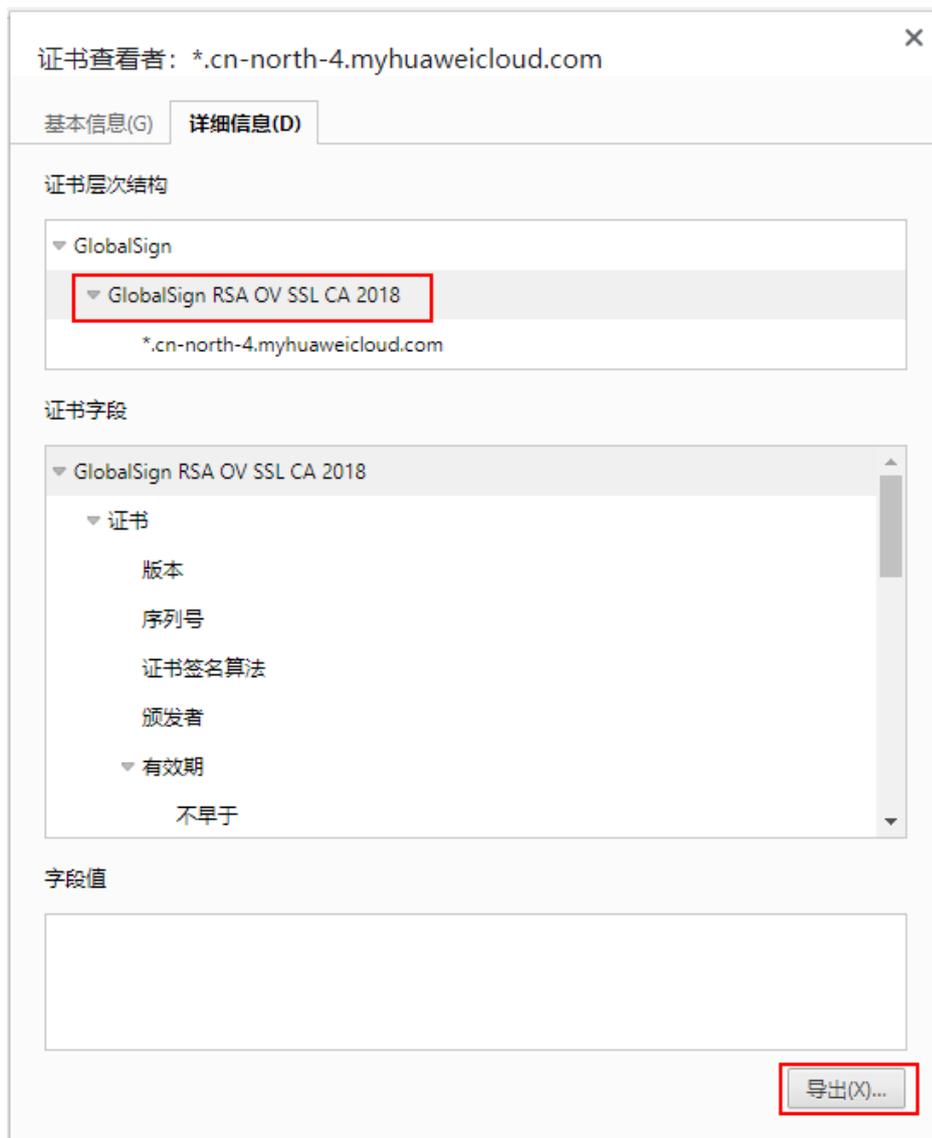
**步骤1** 输入域名：打开浏览器，在华为云场景中，分别输入IAM服务的域名：iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens与KMS的域名：kms.cn-north-4.myhuaweicloud.com/v1.0。

**步骤2** 查找证书：在每次输入域名后，找到SSL连接相关信息，单击后会发现证书，继续单击可查看证书内容。





**步骤3** 导出证书：在证书查看页面，可能会看到证书分为很多级，我们仅需要域名的上一级证书即可，选择该证书并单击导出，便可直接生成证书文件，即我们需要的证书文件。



**步骤4** 上传证书：将导出的证书上传至应用端，并配置到上述参数中即可。

----结束

## 2.5 密态支持函数/存储过程

密态支持函数/存储过程，当前版本只支持sql和plpgsql两种语言。由于密态支持存储过程中创建和执行函数/存储过程对用户是无感知的，因此语法和非密态无区别。

函数/存储过程语法参考《开发指南》中“用户自定义函数”章节和“存储过程”章节。

密态等值查询支持函数存储过程新增系统表gs\_encrypted\_proc，用于存储参数返回的原始数据类型。

系统表具体字段含义可参考《开发指南》中“系统表和系统视图 > 系统表 > GS\_ENCRYPTED\_PROC”章节。

### 创建并执行涉及加密列的函数/存储过程

**步骤1** 创建密钥，详细步骤请参考[使用gsql操作密态数据库](#)。

**步骤2** 创建加密表。

```
gaussdb=# CREATE TABLE creditcard_info (  
    id_number int,  
    name text,  
    credit_card varchar(19) encrypted with (column_encryption_key = cek1, encryption_type =  
    DETERMINISTIC)  
    ) with (orientation=row) distribute by hash(id_number);  
CREATE TABLE
```

**步骤3** 插入数据。

```
gaussdb=# insert into creditcard_info values(1, 'Avi', '1234567890123456');  
INSERT 0 1  
gaussdb=# insert into creditcard_info values(2, 'Eli', '2345678901234567');  
INSERT 0 1
```

**步骤4** 创建函数支持密态等值查询。

```
gaussdb=# CREATE FUNCTION f_encrypt_in_sql(val1 text, val2 varchar(19)) RETURNS text AS 'SELECT  
name from creditcard_info where name=$1 or credit_card=$2 LIMIT 1' LANGUAGE SQL;  
CREATE FUNCTION  
gaussdb=# CREATE FUNCTION f_encrypt_in_plpgsql (val1 text, val2 varchar(19), OUT c text) AS $$  
BEGIN  
    SELECT into c name from creditcard_info where name=$1 or credit_card = $2 LIMIT 1;  
END; $$  
LANGUAGE plpgsql;  
CREATE FUNCTION
```

**步骤5** 执行函数。

```
gaussdb=# SELECT f_encrypt_in_sql('Avi','1234567890123456');  
f_encrypt_in_sql  
-----  
Avi  
(1 row)  
  
gaussdb=# SELECT f_encrypt_in_plpgsql('Avi', val2=>'1234567890123456');  
f_encrypt_in_plpgsql  
-----  
Avi  
(1 row)
```

----结束

## 📖 说明

1. 函数/存储过程中的“执行动态查询语句”中的查询是在执行过程中编译，因此函数/存储过程中的表名、列名不能在创建阶段未知，输入参数不能用于表名、列名或以任何方式连接。
2. 函数/存储过程中的“执行动态查询语句”不支持EXECUTE 'query'中带有需要加密的数据值。
3. 在RETURNS、IN和OUT的参数中，不支持混合使用加密和非加密类型参数。虽然参数类型都是原始数据类型，但实际类型不同。
4. 在高级包接口中，如db\_output.print\_line()等在服务端打印输出的接口不会做解密操作，由于加密数据类型在强转成明文原始数据类型时会打印出该数据类型的默认值。
5. 当前版本函数/存储过程的LANGUAGE只支持SQL和plpgsql，不支持C和JAVA等其他过程语言。
6. 不支持在函数/存储过程中执行其他查询加密列的函数/存储过程。
7. 当前版本不支持default、DECLARE中为变量赋予默认值，且不支持对DECLARE中的返回值进行解密，用户可以在执行函数时用输入参数、输出参数来代替使用。
8. 不支持gs\_dump对涉及加密列的function进行备份。
9. 不支持在函数/存储过程中创建密钥。
10. 该版本密态函数/存储过程不支持触发器。
11. 密态等值查询函数/存储过程不支持对plpgsql语言对语法进行转义，对于语法主体带有引号的CREATE FUNCTION AS '语法主体'，可以用CREATE FUNCTION AS \$\$语法主体\$\$代替。
12. 不支持在密态等值查询函数/存储过程中执行修改加密列定义的操作，包括对创建加密表，添加加密列，由于执行函数是在服务端，客户端没法判断是否需要刷新缓存，得断开连接后或触发刷新客户端加密列缓存才可以对该列做加密操作。
13. 不支持使用密态数据类型（byteawithoutorderwithequalcol、byteawithoutordercol、\_byteawithoutorderwithequalcol、\_byteawithoutordercol）创建函数和存储过程。
14. 密态函数若返回值有加密类型，不支持返回不确定的行类型结果，如RETURN [SETOF] RECORD，可以使用返回可确定的行类型结果替代，如RETURN TABLE(columnname typename[,...])。
15. 密态支持函数在创建加密函数时会在系统表gs\_encrypted\_proc中添加参数对应的加密列的oid，因此删除表后重建同名表可能会使密态函数失效，需要重新创建密态函数。

# 3 分区表

本章节围绕分区表在大数据量场景下如何对保存的数据进行“查询优化”和“运维管理”出发，分六个章节以此对分区表使用上进行系统性说明，包含语义、原理、约束限制等方面。

## 3.1 大容量数据库

### 3.1.1 大容量数据库背景介绍

随着处理数据量的日益增长和使用场景的多样化，数据库越来越多地面对容量大、数据多样化的场景。在过去数据库业界发展的20多年时间里，数据量从最初的MB、GB级逐渐发展到现在的TB级，在如此数据大规模、数据多样化的客观背景下，数据库管理系统（DBMS）在数据查询、数据管理方面提出了更高的要求，客观上要求数据库能够支持多种优化查找策略和管理运维方式。

在计算机科学经典的算法中，人们通常使用分治法（Divide and Conquer）解决场景和规模较大的问题。其基本思想就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题直到最后子问题可以简单的直接求解，原问题的解可看成子问题的解的合并。对于大容量数据场景，数据库提供对数据进行“分治处理”的方式即分区，将逻辑数据库或其组成元素划分为不同的独立部分，每一个分区维护逻辑上存在相类似属性的数据，这样就把庞大的数据整体进行了切分，有利于数据的管理、查找和维护。

### 3.1.2 表分区技术

表分区技术（Table-Partitioning）通过将非常大的表或者索引从逻辑上切分为更小、更易管理的逻辑单元（分区），能够让用户对表查询、变更等语句操作具备更小的影响范围，能够让用户通过分区键（Partition Key）快速的定位到数据所在的分区，从而避免在数据库中对大表的全量扫描，能够在不同的分区上并发进行DDL、DML操作。从用户使用的角度来看，表分区技术主要有以下三个方面能力：

1. 提升大容量数据场景查询效率：由于表内数据按照分区键进行逻辑分区，查询结果可以通过访问分区的子集而不是整个表来实现。这种分区剪枝技术可以提供数量级的性能增益。
2. 降低运维与查询的并发操作影响：降低DML语句、DDL语句并发场景的相互影响，在对一些大数据量以时间维度进行分区的场景下会明显受益。例如，新数据分区进行入库、实时点查操作，老数据分区进行数据清洗、分区合并等运维性质操作。

- 提供大容量场景下灵活的数据运维管理方式：由于分区表从物理上对不同分区的数据做了表文件层面的隔离，每个分区可以具有单独的物理属性，如启用或禁用压缩、物理存储设置和表空间。同时它支持数据管理操作，如数据加载、索引创建和重建，以及分区级别的备份和恢复，而不是对整个表进行操作，从而减少了操作时间。

### 3.1.3 数据分区查找优化

分区表对数据查找方面的帮助主要体现在对分区键进行谓词查询场景，例如一张以月份Month作为分区键的表，如图3-1所示，如果以普通表的方式设计表结构则需要访问表全量的数据（Full Table Scan），如果以日期为分区键重新设计该表，那么原有的全表扫描会被优化成为分区扫描，当表内的数据量很大同时具有很长的历史周期时，由于扫描数据缩减所带来的性能提升会有非常明显的效果，如图3-2所示。

图 3-1 分区表示例图

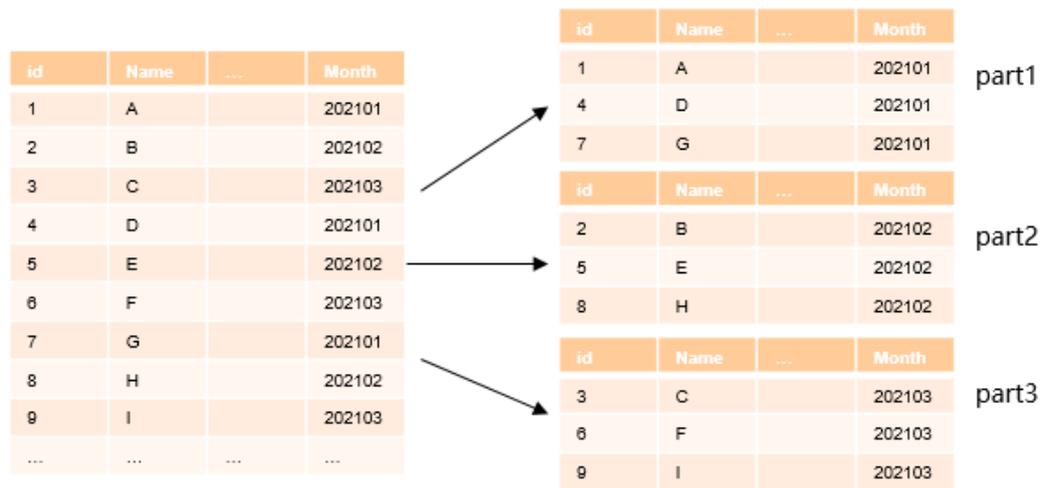
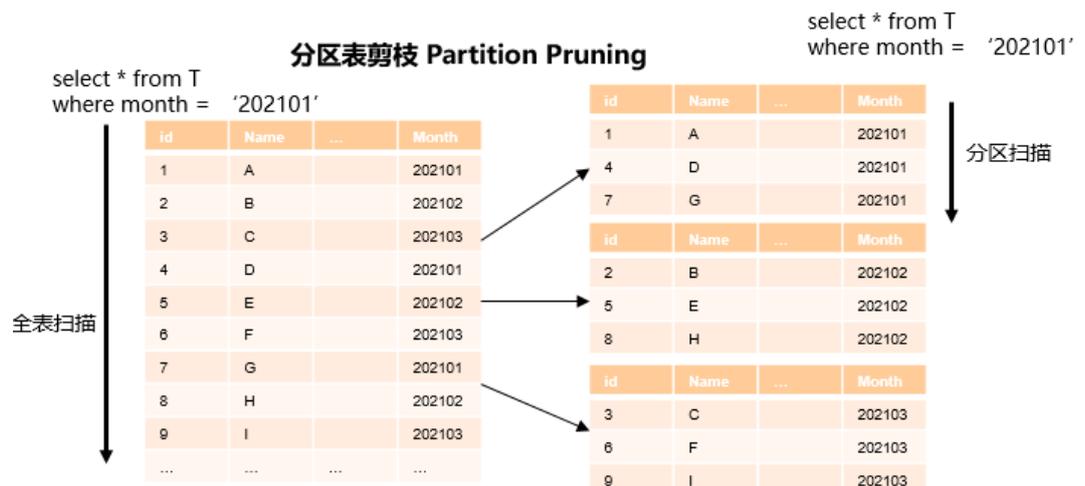


图 3-2 分区表剪枝示例图



### 3.1.4 数据分区运维管理

分区表技术为数据生命周期管理（Data Life Cycle Management, DLM）提供了灵活性的支持，数据生命周期管理是一组用于在数据的整个使用寿命中管理数据的过程和

策略。其中一个重要组成部分是确定在数据生命周期的任何时间点存储数据的最合适和最经济高效的介质：日常操作中使用的较新数据存储在最快速、可用性最高的存储层上，而不经常访问的较旧数据可能存储在成本较低、效率较低的存储层。较旧的数据也可能更新的频率较低，因此将数据压缩并存储为只读是有意义的。

分区表为实施DLM解决方案提供了理想的环境，通过不同分区使用不同表空间，最大限度在确保易用性的同时，实现了有效的数据生命周期的成本优化。这部分的设置由数据库运维人员在服务端设置操作完成，实际用户并不感知这一层面的优化设置，对用户而言逻辑上仍然是对同一张表的查询操作。此外不同分区可以分别实施备份、恢复、索引重建等运维性质的操作，能够对单个数据集不同子类进行分治操作，满足用户业务场景的差异化需求。

## 3.2 分区表介绍

分区表（Partitioned Table）指在单节点内对表数据内容按照分区键、以及围绕分区键的分区策略对表进行逻辑切分。从数据分区的角度来看是一种水平分区（horizontal partition）分区策略方式。分区表增强了数据库应用程序的性能、可管理性和可用性，并有助于降低存储大量数据的总体拥有成本。分区允许将表、索引和索引组织的表细分为更小的部分，使这些数据库对象能够在更精细的粒度级别上进行管理和访问。GaussDB提供了丰富的分区策略和扩展，以满足不同业务场景的需求。由于分区策略的实现完全由数据库内部实现，对用户是完全透明的，因此它几乎可以在实施分区表优化策略以后做平滑迁移，无需潜在耗费人力物力的应用程序更改。本章围绕GaussDB分区表的基本概念从以下几个方面展开介绍：

1. 分区表基本概念：从表分区的基本概念出发，介绍分区表的catalog存储方式以及内部对应原理。
2. 分区策略：从分区表所支持的基本类型出发，介绍各种分区模式下对应的特性以及能够达到的优化特点和效果。

### 3.2.1 基本概念

#### 3.2.1.1 分区表（母表）

实际对用户体现的表，用户对该表进行常规DML语句的增、删、查、改操作。通常使用在建表DDL语句显式的使用PARTITION BY语句进行定义，创建成功以后在pg\_class表中新增一个entry，并且parttype列内容为'p'，表明该entry为分区表的母表。分区母表通常是一个逻辑形态，对应的表文件并不存放数据。

示例1：t1\_hash为一个分区表，分区类型为hash：

```
gaussdb=# CREATE TABLE t1_hash (c1 INT, c2 INT, c3 INT)
PARTITION BY HASH(c1)
(
  PARTITION p0,
  PARTITION p1,
  PARTITION p2,
  PARTITION p3,
  PARTITION p4,
  PARTITION p5,
  PARTITION p6,
  PARTITION p7,
  PARTITION p8,
  PARTITION p9
);
gaussdb=# \d+ t1_hash
          Table "public.t1_hash"
```

```

Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1 | integer | | plain | | 
c2 | integer | | plain | | 
c3 | integer | | plain | | 
Partition By HASH(c1)
Number of partitions: 10 (View pg_partition to check each partition range.)
Distribute By: HASH(c1)
Location Nodes: ALL DATANODES
Has OIDs: no
Options: orientation=row, compression=no

--查询t1_hash分区类型
gaussdb=# SELECT relname, parttype FROM pg_class WHERE relname = 't1_hash';
relname | parttype
-----+-----
t1_hash | p
(1 row)

--删除t1_hash。
gaussdb=# DROP TABLE t1_hash;

```

### 3.2.1.2 分区（分区子表、子分区）

分区表中实际保存数据的表，对应的entry通常保存在pg\_partition中，各个子分区的parentid作为外键关联其分区母表在pg\_class表中的oid列。

示例1：t1\_hash为一个分区表：

```

gaussdb=# CREATE TABLE t1_hash (c1 INT, c2 INT, c3 INT)
PARTITION BY HASH(c1)
(
    PARTITION p0,
    PARTITION p1,
    PARTITION p2,
    PARTITION p3,
    PARTITION p4,
    PARTITION p5,
    PARTITION p6,
    PARTITION p7,
    PARTITION p8,
    PARTITION p9
);

--查询t1_hash分区类型
gaussdb=# SELECT oid, relname, parttype FROM pg_class WHERE relname = 't1_hash';
oid | relname | parttype
-----+-----+-----
16685 | t1_hash | p
(1 row)

--查询t1_hash的分区信息
gaussdb=# SELECT oid, relname, parttype, parentid FROM pg_partition WHERE parentid = 16685;
oid | relname | parttype | parentid
-----+-----+-----+-----
16688 | t1_hash | r | 16685
16689 | p0 | p | 16685
16690 | p1 | p | 16685
16691 | p2 | p | 16685
16692 | p3 | p | 16685
16693 | p4 | p | 16685
16694 | p5 | p | 16685
16695 | p6 | p | 16685
16696 | p7 | p | 16685
16697 | p8 | p | 16685
16698 | p9 | p | 16685
(11 rows)

```

```
--删除t1_hash  
gaussdb=# DROP TABLE t1_hash;
```

### 3.2.1.3 分区键

分区键由一个或多个列组成，分区键值结合对应分区方法能够唯一确定某一元组所在的分区，通常在建表时通过PARTITION BY语句指定：

```
CREATE TABLE table_name (...) PARTITION BY part_strategy (partition_key) (...)
```

#### 须知

范围分区表和列表分区表支持最多16列分区键；其他分区表只支持1列分区键。

## 3.2.2 分区策略

分区策略在使用DDL语句建表语句时通过PARTITION BY语句的语法指定，分区策略描述了在分区表中数据和分区路由映射规则。常见的分区类型有基于条件的Range分区、基于哈希散列函数的Hash分区、基于数据枚举的List列表分区：

```
CREATE TABLE table_name (...) PARTITION BY partition_strategy (partition_key) (...)
```

### 3.2.2.1 范围分区

范围分区（Range Partition）根据为每个分区建立的分区键的值范围将数据映射到分区。范围分区是生产系统中最常见的分区类型，通常在以时间维度（Date、Time Stamp）描述数据场景中使用。范围分区有两种语法格式，示例如下：

#### 1. VALUES LESS THAN的语法格式

对于从句是VALUE LESS THAN的语法格式，范围分区策略的分区键最多支持16列。

- 单列分区键示例如下：

```
gaussdb=# CREATE TABLE range_sales  
(  
  product_id INT4 NOT NULL,  
  customer_id INT4 NOT NULL,  
  time DATE,  
  channel_id CHAR(1),  
  type_id INT4,  
  quantity_sold NUMERIC(3),  
  amount_sold NUMERIC(10,2)  
)  
PARTITION BY RANGE (time)  
(  
  PARTITION date_202001 VALUES LESS THAN ('2020-02-01'),  
  PARTITION date_202002 VALUES LESS THAN ('2020-03-01'),  
  PARTITION date_202003 VALUES LESS THAN ('2020-04-01'),  
  PARTITION date_202004 VALUES LESS THAN ('2020-05-01')  
);
```

```
--清理示例  
gaussdb=# DROP TABLE range_sales;
```

其中date\_202002表示2020年2月的分区，将包含分区键值从2020年2月1日到2020年2月29日的数据。

每个分区都有一个VALUES LESS子句，用于指定分区的非包含上限。大于或等于该分区键的任何值都将添加到下一个分区。除第一个分区外，所有分区都具有由前一个分区的VALUES LESS子句指定的隐式下限。可以为最高分区

定义MAXVALUE关键字，MAXVALUE表示一个虚拟无限值，其排序高于分区键的任何其他可能值，包括空值。

- 多列分区键示例如下：

```
gaussdb=# CREATE TABLE range_sales_with_multiple_keys
(
  c1   INT4 NOT NULL,
  c2   INT4 NOT NULL,
  c3   CHAR(1)
)
PARTITION BY RANGE (c1,c2)
(
  PARTITION p1 VALUES LESS THAN (10,10),
  PARTITION p2 VALUES LESS THAN (10,20),
  PARTITION p3 VALUES LESS THAN (20,10)
);
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(9,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(9,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(9,21,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,15,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,21,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,21,'a');

gaussdb=# SELECT * FROM range_sales_with_multiple_keys PARTITION (p1);
 c1 | c2 | c3
----+----+----
  9 |  5 | a
  9 | 20 | a
  9 | 21 | a
 10 |  5 | a
(4 rows)

gaussdb=# SELECT * FROM range_sales_with_multiple_keys PARTITION (p2);
 c1 | c2 | c3
----+----+----
 10 | 15 | a
(1 row)

gaussdb=# SELECT * FROM range_sales_with_multiple_keys PARTITION (p3);
 c1 | c2 | c3
----+----+----
 10 | 20 | a
 10 | 21 | a
 11 |  5 | a
 11 | 20 | a
 11 | 21 | a
(5 rows)

--清理示例
gaussdb=# DROP TABLE range_sales_with_multiple_keys;
```

### 📖 说明

多列分区的分区规则如下：

1. 从第一列开始比较。
2. 如果插入的值当前列小于分区当前列边界值，则直接插入。
3. 如果插入的当前列等于分区当前列的边界值，则比较插入值的下一列与分区下一列边界值的大小。
4. 如果插入的当前列大于分区当前列的边界值，则换下一个分区进行比较。

## 2. START END语法格式

对于从句是START END语法格式，范围分区策略的分区键最多支持1列。

示例如下：

```
-- 创建表空间
gaussdb=# CREATE TABLESPACE startend_tbs1 LOCATION '/home/omm/startend_tbs1';
gaussdb=# CREATE TABLESPACE startend_tbs2 LOCATION '/home/omm/startend_tbs2';
gaussdb=# CREATE TABLESPACE startend_tbs3 LOCATION '/home/omm/startend_tbs3';
gaussdb=# CREATE TABLESPACE startend_tbs4 LOCATION '/home/omm/startend_tbs4';
-- 创建临时schema
gaussdb=# CREATE SCHEMA tpcds;
gaussdb=# SET CURRENT_SCHEMA TO tpcds;
-- 创建分区表，分区键是integer类型
gaussdb=# CREATE TABLE tpcds.startend_pt (c1 INT, c2 INT)
TABLESPACE startend_tbs1
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) TABLESPACE startend_tbs2,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) TABLESPACE startend_tbs3,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000) TABLESPACE startend_tbs4
)
ENABLE ROW MOVEMENT;

-- 查看分区表信息
gaussdb=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN pg_tablespace t ON
p.reltablespace=t.oid and p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
relname | boundaries | spcname
-----+-----+-----
p1_0 | {1} | startend_tbs2
p1_1 | {201} | startend_tbs2
p1_2 | {401} | startend_tbs2
p1_3 | {601} | startend_tbs2
p1_4 | {801} | startend_tbs2
p1_5 | {1000} | startend_tbs2
p2 | {2000} | startend_tbs1
p3 | {2500} | startend_tbs3
p4 | {3000} | startend_tbs1
p5_1 | {4000} | startend_tbs4
p5_2 | {5000} | startend_tbs4
startend_pt || startend_tbs1
(12 rows)

-- 清理示例
gaussdb=# DROP TABLE tpcds.startend_pt;
DROP TABLE
gaussdb=# DROP SCHEMA tpcds;
DROP SCHEMA
```

### 3.2.2.2 哈希分区

哈希分区（Hash Partition）基于对分区键使用哈希算法将数据映射到分区。使用的哈希算法为GaussDB内置哈希算法，在分区键取值范围不倾斜（no data skew）场景下，哈希算法在分区之间均匀分布行，使分区大小大致相同。因此哈希分区是实现分区间均匀分布数据的理想方法。哈希分区也是范围分区的一种易于使用的替代方法，尤其是当要分区的数据不是历史数据或没有明显的分区键时，示例如下：

```
gaussdb=# CREATE TABLE bmsql_order_line (
    ol_w_id    INTEGER NOT NULL,
    ol_d_id    INTEGER NOT NULL,
    ol_o_id    INTEGER NOT NULL,
    ol_number  INTEGER NOT NULL,
    ol_i_id    INTEGER NOT NULL,
    ol_delivery_d  TIMESTAMP,
    ol_amount  DECIMAL(6,2),
    ol_supply_w_id  INTEGER,
    ol_quantity  INTEGER,
    ol_dist_info  CHAR(24)
)
--预先定义100个分区
```

```
PARTITION BY HASH(ol_d_id)
(
  PARTITION p0,
  PARTITION p1,
  PARTITION p2,
  ...
  PARTITION p99
);
```

上述例子中，bmsql\_order\_line表的ol\_d\_id进行了分区，ol\_d\_id列是一个identifier性质的属性列，本身并不带有时间或者某一个特定维度上的区分。使用哈希分区策略来对其进行分表处理则是一个较为理想的选择，相比其他分区类型，除了预先确保分区键没有过多数据倾斜（某一、某几个值重复度高），只需要指定分区键和分区数即可创建分区，同时还能够确保每个分区的数据均匀，提升了分区表的易用性。

### 3.2.2.3 列表分区

列表分区（List Partition）能够通过在每个分区的描述中为分区键指定离散值列表来显式控制行如何映射到分区。列表分区的优势在于可以以枚举分区值方式对数据进行分区，可以对无序和不相关的数据集进行分组和组织。对于未定义在列表中的分区键值，可以使用默认分区（DEFAULT）来进行数据的保存，这样所有未映射到任何其他分区的行都不会生成错误。示例如下：

```
gaussdb=# CREATE TABLE bmsql_order_line (
  ol_w_id      INTEGER NOT NULL,
  ol_d_id      INTEGER NOT NULL,
  ol_o_id      INTEGER NOT NULL,
  ol_number    INTEGER NOT NULL,
  ol_i_id      INTEGER NOT NULL,
  ol_delivery_d  TIMESTAMP,
  ol_amount    DECIMAL(6,2),
  ol_supply_w_id  INTEGER,
  ol_quantity  INTEGER,
  ol_dist_info  CHAR(24)
)
PARTITION BY LIST(ol_d_id)
(
  PARTITION p0 VALUES (1,4,7),
  PARTITION p1 VALUES (2,5,8),
  PARTITION p2 VALUES (3,6,9),
  PARTITION p3 VALUES (DEFAULT)
);

--清理示例
gaussdb=# DROP TABLE bmsql_order_line;
```

上述例子和之前给出的哈希分区的例子类似，同样通过ol\_d\_id列进行分区，但是在List分区中直接通过对ol\_d\_id的可能取值范围进行限定，不在列表中的数据会进入p3分区（DEFAULT）。相比哈希分区，List列表分区对分区键的可控性更好，往往能够将目标数据保存在预想的分区中，但是如果列表值较多时在分区定义时变得麻烦，该情况下推荐使用Hash哈希分区。List、Hash分区往往都是处理无序、不相关的数据集进行分组和组织。

#### 注意

列表分区的分区键最多支持16列。如果分区键定义为1列，子分区定义时List列表中的枚举值不允许为NULL值；如果分区键定义为多列，子分区定义时List列表中的枚举值允许有NULL值。

### 3.2.2.4 分区表对导入操作的性能影响

在GaussDB内核实现中，分区表数据的插入的处理过程相比非分区表增加分区路由部分的开销，因从整体上分区表场景的数据插入开销主要看成：（1）heap-insert基表插入、（2）partition-routing分区路由两个部分，其中heap基表插入解决tuple入库对应heap表的问题并且该部分普通表和分区表共用，而分区路由部分解决分区路由即tuple元组插入到对应partRel的问题。

因此对数据插入优化的侧重点如下：

1. 分区表基表Heap表插入：
  - a. 算子底噪优化
  - b. heap数据插入
  - c. 索引插入build优化（带索引）
2. 分区表分区路由：
  - a. 路由查找算法逻辑优化
  - b. 路由底噪优化，包括分区表partRel句柄开启、新增的函数调用逻辑开销

#### 说明

分区路由的性能主要通过大数据量的单条INSERT语句体现，UPDATE场景内部包含了查找对应要更新的元组进行DELETE操作然后再进行INSERT，因此不如单条INSERT语句场景直接。

不同分区类型的路由算法逻辑如[表3-1](#)所示：

表 3-1 路由算法逻辑

| 分区方式                        | 路由算法复杂度     | 实现概述说明              |
|-----------------------------|-------------|---------------------|
| 范围分区 ( Range Partition )    | $O(\log N)$ | 基于二分binary-search实现 |
| 间隔分区 ( Interval Partition ) | $O(\log N)$ | 基于二分binary-search实现 |
| 哈希分区 ( Hash-Partition )     | $O(1)$      | 基于key-partOid哈希表实现  |
| 列表分区 ( List-Partition )     | $O(1)$      | 基于key-partOid哈希表实现  |

#### 注意

分区路由的主要处理逻辑根据导入数据元组的分区键计算其所在分区的过程，相比非分区表这部分为额外增加的开销，这部分开销在最终数据导入上的具体性能损失和服务器CPU处理能力、表宽度、磁盘/内存的实际容量相关，通常可以粗略认为：

- x86服务器场景下分区表相比普通表的导入性能会略低10%以内。
- ARM服务器场景下为20%，造成x86和ARM指向性能略微差异的主要原因是分区路由为in-memory计算强化场景，主流x86体系CPU在单核指令处理能力上略优于arm。

## 3.2.3 分区基本使用

### 3.2.3.1 创建分区表

#### 创建分区表

由于SQL语言功能强大和灵活多样性，SQL语法树通常比复杂，分区表同样如此，分区表的创建可以理解成在原有非分区表的基础上新增表分区属性，因此分区表的语法接口可以看成是对原有非分区表CREATE TABLE语句进行扩展PARTITION BY语句部分，同时指定分区相关的三个核元素：

1. 分区类型 ( partType )：描述分区表的分区策略，分别有RANGE/INTERVAL/LIST/HASH。
2. 分区键 ( partKey )：描述分区表的分区列，目前RANGE/LIST分区支持多列（不超过16列）分区键，HASH分区只支持单列分区。
3. 分区表达式 ( partExpr )：描述分区表的具体分区表方式，即键值与分区的对应映射关系。

这三部分重要元素在建表语句的Partition By Clause字句中体现，*PARTITION BY partType (partKey) ( partExpr[,partExpr]···)*。示例如下：

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
(
  [ /* 该部分继承于普通表的Create Table */
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
  | table_constraint
  | LIKE source_table [ like_option [ ... ] ] }, ... ]
)
[ WITH ( {storage_parameter = value} [ , ... ] ) ]
[ COMPRESS | NOCOMPRESS ]
[ TABLESPACE tablespace_name ]
/* 范围分区场景 */
PARTITION BY RANGE (partKey) [ INTERVAL ('interval_expr') [ STORE IN (tablespace_name [ , ... ] ) ] ] (
  partition_start_end_item [ , ... ]
  partition_less_then_item [ , ... ]
)
/* 列表分区场景 */
PARTITION BY LIST (partKey)
(
  PARTITION partition_name VALUES (list_values_clause) [ TABLESPACE tablespace_name [ , ... ] ]
  ...
)
/* 哈希分区场景 */
PARTITION BY HASH (partKey) (
  PARTITION partition_name [ TABLESPACE tablespace_name [ , ... ] ]
  ...
)
/* 开启/关闭分区表行迁移 */
[ { ENABLE | DISABLE } ROW MOVEMENT ];
```

规格约束：

1. Range/List分区最大支持16个分区键，Hash分区只支持1个分区键。
2. 除哈希分区外，分区键不能插入空值，否则DML语句会进行报错处理。唯一例外：Range分区表定义有MAXVALUE分区/List分区表定义有DEFAULT分区。
3. 分区数最大值为1048575个，可以满足大部分业务场景的诉求。但分区数增加会导致系统中文件数增加，影响系统的性能，一般对于单个表而言不建议分区数超过200。

## 修改分区属性

分区表和分区相关的部分属性可以使用类似非分区表的ALTER-TABLE命令进行分区属性修改，常用的分区属性修改语句包括：

1. 增加分区
2. 删除分区
3. 删除/清空分区数据
4. 切割分区
5. 合并分区
6. 移动分区
7. 交换分区
8. 重命名分区

以上常用的分区属性变更语句基于对普通表ALTER TABLE语句进行扩展，在使用方式上大部分使用方式类似，分区表属性变更的基本语法框架示例如下：

```
/* 基本alter table语法 */  
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
action [, ... ];
```

分区表ALTER TABLE语句使用方法请参考[分区表运维管理](#)、《开发指南》中“SQL参考 > SQL语法 > ALTER TABLE PARTITION”章节。

### 3.2.3.2 分区表 DML 查询语句

由于分区的实现完全体现在数据库内核中，用户对分区表的DQL/DML与非分区表相比，在语法上没有任何区别。

出于分区表的易用性考虑，GaussDB支持指定分区的查询操作，指定分区可以通过PARTITION (partname)或者PARTITION FOR (partvalue)来进行。指定分区执行DQL/DML时，若插入的数据不属于目标分区，则业务会产生报错；若查询的数据不属于目标分区，则会跳过该数据的处理。

指定分区DML支持以下几类语法：

1. 查询（SELECT）
2. 插入（INSERT）
3. 更新（UPDATE）
4. 删除（DELETE）
5. 插入或更新（UPSERT）
6. 合并（MERGE INTO）

下面给出了指定分区做DML的示例：

```
/* 创建分区表 list_02 */  
gaussdb=# CREATE TABLE IF NOT EXISTS list_02  
(  
  id INT,  
  role VARCHAR(100),  
  data VARCHAR(100)  
)  
PARTITION BY LIST (id)  
(  
  PARTITION p_list_2 VALUES(0,1,2,3,4,5,6,7,8,9),  
  PARTITION p_list_3 VALUES(10,11,12,13,14,15,16,17,18,19),  
  PARTITION p_list_4 VALUES( DEFAULT ),
```

```
PARTITION p_list_5 VALUES(20,21,22,23,24,25,26,27,28,29),
PARTITION p_list_6 VALUES(30,31,32,33,34,35,36,37,38,39),
PARTITION p_list_7 VALUES(40,41,42,43,44,45,46,47,48,49)
) ENABLE ROW MOVEMENT;
/* 导入数据 */
INSERT INTO list_02 VALUES(null, 'alice', 'alice data');
INSERT INTO list_02 VALUES(2, null, 'bob data');
INSERT INTO list_02 VALUES(null, null, 'peter data');

/* 对指定分区进行查询 */
-- 查询分区表全部数据
gaussdb=# SELECT * FROM list_02 ORDER BY data;
 id | role | data
-----+-----+-----
    | alice | alice data
  2 |    | bob data
    |    | peter data
(3 rows)
-- 查询分区p_list_2数据
gaussdb=# SELECT * FROM list_02 PARTITION (p_list_2) ORDER BY data;
 id | role | data
-----+-----+-----
  2 |    | bob data
(1 row)
-- 查询(100)所对应的分区的数据，即分区p_list_4
gaussdb=# SELECT * FROM list_02 PARTITION FOR (100) ORDER BY data;
 id | role | data
-----+-----+-----
    | alice | alice data
    |    | peter data
(2 rows)

/* 对指定分区做IUD */
-- 删除分区p_list_5中的全部数据
gaussdb=# DELETE FROM list_02 PARTITION (p_list_5);
-- 指定分区p_list_7插入数据，由于数据不符合该分区约束，插入报错
gaussdb=# INSERT INTO list_02 PARTITION (p_list_7) VALUES(null, 'cherry', 'cherry data');
ERROR: inserted partition key does not map to the table partition
-- 将分区值100所属分区，即分区p_list_4的数据进行更新
gaussdb=# UPDATE list_02 PARTITION FOR (100) SET data = "";

--UPSERT。
gaussdb=# INSERT INTO list_02 (id, role, data) VALUES (1, 'test', 'testdata') ON DUPLICATE KEY UPDATE
role = VALUES(role), data = VALUES(data);

--MERGE INTO。
gaussdb=# CREATE TABLE IF NOT EXISTS list_tmp
(
 id INT,
 role VARCHAR(100),
 data VARCHAR(100)
)
PARTITION BY LIST (id)
(
 PARTITION p_list_2 VALUES(0,1,2,3,4,5,6,7,8,9),
 PARTITION p_list_3 VALUES(10,11,12,13,14,15,16,17,18,19),
 PARTITION p_list_4 VALUES( DEFAULT ),
 PARTITION p_list_5 VALUES(20,21,22,23,24,25,26,27,28,29),
 PARTITION p_list_6 VALUES(30,31,32,33,34,35,36,37,38,39),
 PARTITION p_list_7 VALUES(40,41,42,43,44,45,46,47,48,49)) ENABLE ROW MOVEMENT;

gaussdb=# MERGE INTO list_tmp target
USING list_02 source
ON (target.id = source.id)
WHEN MATCHED THEN
 UPDATE SET target.data = source.data,
 target.role = source.role
WHEN NOT MATCHED THEN
 INSERT (id, role, data)
```

```
VALUES (source.id, source.role, source.data);  
--删除表。  
gaussdb=#  
DROP TABLE list_02;  
DROP TABLE list_tmp;
```

## 3.3 分区表查询优化

### 📖 说明

本小节示例对应explain\_perf\_mode参数值为normal。

### 3.3.1 分区剪枝

#### 3.3.1.1 分区表静态剪枝

对于检索条件中分区键上带有常数的分区表查询语句，在优化器阶段将对indexscan、bitmap indexscan、indexonlyscan等算子中包含的检索条件作为剪枝条件，完成分区的筛选。算子包含的检索条件中需要至少包含一个分区键字段，对于含有多个分区键的分区表，包含任意分区键子集即可。

静态剪枝支持范围如下所示：

1. 支持分区类型：范围分区、哈希分区、列表分区。
2. 支持表达式类型：比较表达式（<, <=, =, >=, >）、逻辑表达式、数组表达式。

### ⚠️ 注意

- 目前静态剪枝不支持子查询表达式。
- 为了支持分区表剪枝，在计划生成时会将分区键上的过滤条件强制转换为分区键类型，和隐式类型转换规则存在差异，可能导致相同条件在分区键上转换报错，非分区键上无报错。

- 静态剪枝支持的典型场景具体示例如下：

#### a. 比较表达式

```
--创建分区表  
gaussdb=# CREATE TABLE t1 (c1 int, c2 int)  
PARTITION BY RANGE (c1)  
(  
    PARTITION p1 VALUES LESS THAN(10),  
    PARTITION p2 VALUES LESS THAN(20),  
    PARTITION p3 VALUES LESS THAN(MAXVALUE)  
);  
gaussdb=# SET max_datanode_for_plan = 1;  
  
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1;  
QUERY PLAN  
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: datanode1  
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1
```

```
Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = 1)
Selected Partitions: 1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 < 1;
QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 < 1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 < 1
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 < 1)
Selected Partitions: 1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 > 11;
QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 > 11

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 > 11
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 2
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 > 11)
Selected Partitions: 2..3

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 IS NULL;
QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: datanode1
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 IS NULL

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 IS NULL
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 IS NULL)
Selected Partitions: 3
```

(15 rows)

b. 逻辑表达式

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 AND c2 = 2;  
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: datanode1  
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 AND c2 = 2  
  
Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 AND c2 = 2  
Datanode Name: datanode1  
Partition Iterator  
Output: c1, c2  
Iterations: 1  
-> Partitioned Seq Scan on public.t1  
Output: c1, c2  
Filter: ((t1.c1 = 1) AND (t1.c2 = 2))  
Selected Partitions: 1
```

(15 rows)

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 OR c1 = 2;  
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: All datanodes  
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 OR c1 = 2  
  
Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 OR c1 = 2  
Datanode Name: datanode1  
Partition Iterator  
Output: c1, c2  
Iterations: 1  
-> Partitioned Seq Scan on public.t1  
Output: c1, c2  
Filter: ((t1.c1 = 1) OR (t1.c1 = 2))  
Selected Partitions: 1
```

(15 rows)

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE NOT c1 = 1;  
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: All datanodes  
Remote query: SELECT c1, c2 FROM public.t1 WHERE NOT c1 = 1  
  
Remote SQL: SELECT c1, c2 FROM public.t1 WHERE NOT c1 = 1  
Datanode Name: datanode1  
Partition Iterator  
Output: c1, c2  
Iterations: 3  
-> Partitioned Seq Scan on public.t1  
Output: c1, c2  
Filter: (t1.c1 <> 1)  
Selected Partitions: 1..3
```

(15 rows)

c. 数组表达式

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 IN (1, 2, 3);  
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2
```

```
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
Selected Partitions: 1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ALL(ARRAY[1, 2, 3]);
                                QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ALL (ARRAY[1, 2, 3])

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ALL (ARRAY[1, 2, 3])
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 0
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = ALL ('{1,2,3}'::integer[]))
Selected Partitions: NONE

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ANY(ARRAY[1, 2, 3]);
                                QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
Datanode Name: datanode1
Partition Iterator
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
Selected Partitions: 1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = SOME(ARRAY[1, 2, 3]);
                                QUERY PLAN
-----
Data Node Scan
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
Datanode Name: datanode1
Partition Iterator
```

```
Output: c1, c2
Iterations: 1
-> Partitioned Seq Scan on public.t1
   Output: c1, c2
   Filter: (t1.c1 = ANY ('{1,2,3}':integer[]))
   Selected Partitions: 1

(15 rows)
```

- 静态剪枝不支持的典型场景具体示例如下：

a. 子查询表达式

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ALL(SELECT c2
FROM t1 WHERE c1 > 10);
          QUERY PLAN
-----
Streaming (type: GATHER)
  Output: public.t1.c1, public.t1.c2
  Node/s: All datanodes
  -> Partition Iterator
     Output: public.t1.c1, public.t1.c2
     Iterations: 3
     -> Partitioned Seq Scan on public.t1
        Output: public.t1.c1, public.t1.c2
        Distribute Key: public.t1.c1
        Filter: (SubPlan 1)
        Selected Partitions: 1..3
        SubPlan 1
        -> Materialize
           Output: public.t1.c2
           -> Streaming(type: BROADCAST)
              Output: public.t1.c2
              Spawn on: All datanodes
              Consumer Nodes: All datanodes
              -> Partition Iterator
                 Output: public.t1.c2
                 Iterations: 2
                 -> Partitioned Seq Scan on public.t1
                    Output: public.t1.c2
                    Distribute Key: public.t1.c1
                    Filter: (public.t1.c1 > 10)
                    Selected Partitions: 2..3

(26 rows)

--清理示例环境。
gaussdb=# DROP TABLE t1;
```

### 3.3.1.2 分区表动态剪枝

对于检索条件中存在带有变量的分区表查询语句，由于优化器阶段无法获取用户的绑定参数，因此优化器阶段仅能完成indexscan、bitmapindexscan、indexonlyscan等算子检索条件的解析，后续会在执行器阶段获得绑定参数后，完成分区筛选。算子包含的检索条件中需要至少包含一个分区键字段，对于含有多个分区键的分区表，包含任意分区键子集即可。目前分区表动态剪枝仅支持PBE（Prepare/Bind/Execute）场景和参数化路径场景。

#### 3.3.1.2.1 PBE 动态剪枝

PBE动态剪枝支持范围如下所示：

1. 支持分区类型：范围分区、哈希分区、列表分区。
2. 支持表达式类型：比较表达式（<, <=, =, >=, >）、逻辑表达式、数组表达式。
3. 支持部分隐式类型转换和函数：对于类型可以相互转换的场景和immutable函数可以支持PBE动态剪枝

**注意**

- PBE动态剪枝支持表达式、隐式转换、immutable函数和stable函数，不支持子查询表达式和volatile函数。对于stable函数，如to\_timestamp等类型转换函数，可能会受GUC参数变化，影响剪枝结果。为了保持性能优化，此情况可以通过analyze表重新生成gplan解决。
- 由于PBE动态剪枝是基于generic plan的剪枝，所以判断语句是否能PBE动态剪枝时，需要设置参数 plan\_cache\_mode = 'force\_generic\_plan'，排除custom plan的干扰。

- PBE动态剪枝支持的典型场景具体示例如下：

a. 比较表达式

```
--创建分区表
gaussdb=# CREATE TABLE t1 (c1 int, c2 int)
PARTITION BY RANGE (c1)
(
  PARTITION p1 VALUES LESS THAN(10),
  PARTITION p2 VALUES LESS THAN(20),
  PARTITION p3 VALUES LESS THAN(MAXVALUE)
);

gaussdb=# PREPARE p1(int) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p1(1);
QUERY PLAN

-----
Data Node Scan
  Output: t1.c1, t1.c2
  Node/s: datanode1
  Node expr: $1
  Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1
Datanode Name: datanode1
Partition Iterator
  Output: c1, c2
  Iterations: PART
  -> Partitioned Seq Scan on public.t1
      Output: c1, c2
      Filter: (t1.c1 = $1)
      Selected Partitions: 1 (pbe-pruning)

(16 rows)
```

b. 逻辑表达式

```
gaussdb=# PREPARE p2(INT, INT) AS SELECT * FROM t1 WHERE c1 = $1 AND c2 = $2;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p2(1, 2);
QUERY PLAN

-----
Data Node Scan
  Output: t1.c1, t1.c2
  Node/s: datanode1
  Node expr: $1
  Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1 AND c2 = $2

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1 AND c2 = $2
Datanode Name: datanode1
Partition Iterator
  Output: c1, c2
  Iterations: PART
  -> Partitioned Seq Scan on public.t1
      Output: c1, c2
      Filter: ((t1.c1 = $1) AND (t1.c2 = $2))
```

```
Selected Partitions: 1 (pbe-pruning)

(16 rows)

c. 类型转换触发隐式转换
gaussdb=# set plan_cache_mode = 'force_generic_plan';
gaussdb=# PREPARE p3(TEXT) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p3('12');
QUERY PLAN
-----
Data Node Scan
  Output: t1.c1, t1.c2
  Node/s: datanode1
  Node expr: $1
  Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1::bigint

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1::bigint
Datanode Name: datanode1
Partition Iterator
  Output: c1, c2
  Iterations: PART
  -> Partitioned Seq Scan on public.t1
      Output: c1, c2
      Filter: (t1.c1 = ($1)::bigint)
      Selected Partitions: 2 (pbe-pruning)

(16 rows)
```

- PBE动态剪枝不支持的典型场景具体示例如下:

```
a. 子查询表达式
gaussdb=# PREPARE p4(INT) AS SELECT * FROM t1 WHERE c1 = ALL(SELECT c2 FROM t1
WHERE c1 > $1);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p4(1);
QUERY PLAN
-----
Streaming (type: GATHER)
  Output: public.t1.c1, public.t1.c2
  Node/s: All datanodes
  -> Partition Iterator
      Output: public.t1.c1, public.t1.c2
      Iterations: 3
      -> Partitioned Seq Scan on public.t1
          Output: public.t1.c1, public.t1.c2
          Distribute Key: public.t1.c1
          Filter: (SubPlan 1)
          Selected Partitions: 1..3
          SubPlan 1
              -> Materialize
                  Output: public.t1.c2
                  -> Streaming(type: BROADCAST)
                      Output: public.t1.c2
                      Spawn on: All datanodes
                      Consumer Nodes: All datanodes
                      -> Partition Iterator
                          Output: public.t1.c2
                          Iterations: 3
                          -> Partitioned Seq Scan on public.t1
                              Output: public.t1.c2
                              Distribute Key: public.t1.c1
                              Filter: (public.t1.c1 > 1)
                              Selected Partitions: 1..3

(26 rows)
```

```
b. 类型转换无法直接触发隐式转换
gaussdb=# PREPARE p5(name) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p5('12');
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: All datanodes  
Remote query: SELECT c1, c2 FROM public.t1 WHERE c1::text = '12'::text  
  
Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1::text = '12'::text  
Datanode Name: datanode1  
Partition Iterator  
Output: c1, c2  
Iterations: 3  
-> Partitioned Seq Scan on public.t1  
Output: c1, c2  
Filter: ((t1.c1)::text = '12'::text)  
Selected Partitions: 1..3
```

(15 rows)

### c. stable/volatile函数

```
gaussdb=# create sequence seq;  
gaussdb=# PREPARE p6(TEXT) AS SELECT * FROM t1 WHERE c1 = currval($1);--volatile函数不支持剪枝  
PREPARE  
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p6('seq');  
QUERY PLAN
```

```
-----  
Data Node Scan  
Output: t1.c1, t1.c2  
Node/s: All datanodes  
Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE true  
Coordinator quals: ((t1.c1)::numeric = currval(('seq'::text)::regclass))  
  
Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE true  
Datanode Name: datanode1  
Partition Iterator  
Output: c1, c2  
Iterations: 3  
-> Partitioned Seq Scan on public.t1  
Output: c1, c2  
Selected Partitions: 1..3
```

(15 rows)

```
--清理示例环境。  
gaussdb=# DROP TABLE t1;
```

### 3.3.1.2.2 参数化路径动态剪枝

参数化路径动态剪枝支持范围如下所示：

1. 支持分区类型：范围分区、哈希分区、列表分区。
2. 支持算子类型：indexscan、indexonlyscan、bitmapscan。
3. 支持表达式类型：比较表达式（<, <=, =, >=, >）、逻辑表达式。

#### 注意

参数化路径动态剪枝不支持子查询表达式，不支持stable和volatile函数，不支持跨QueryBlock参数化路径，不支持BitmapOr, BitmapAnd算子。

- 参数化路径动态剪枝支持的典型场景具体示例如下：

#### a. 比较表达式

```
--创建分区表和索引  
gaussdb=# CREATE TABLE t1 (c1 INT, c2 INT)
```

```
PARTITION BY RANGE (c1)
(
  PARTITION p1 VALUES LESS THAN(10),
  PARTITION p2 VALUES LESS THAN(20),
  PARTITION p3 VALUES LESS THAN(MAXVALUE)
);
gaussdb=# CREATE TABLE t2 (c1 INT, c2 INT)
PARTITION BY RANGE (c1)
(
  PARTITION p1 VALUES LESS THAN(10),
  PARTITION p2 VALUES LESS THAN(20),
  PARTITION p3 VALUES LESS THAN(MAXVALUE)
);
gaussdb=# CREATE INDEX t1_c1 ON t1(c1) LOCAL;
gaussdb=# CREATE INDEX t2_c1 ON t2(c1) LOCAL;
gaussdb=# CREATE INDEX t1_c2 ON t1(c2) LOCAL;
gaussdb=# CREATE INDEX t2_c2 ON t2(c2) LOCAL;

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 = t2.c1;
QUERY
PLAN
-----
Data Node Scan
  Output: t2.c1, t2.c2, t1.c1, t1.c2
  Node/s: All datanodes
  Remote query: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1,
t1.c2 FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1

Remote SQL: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1, t1.c2
FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1
Datanode Name: datanode1
Nested Loop
  Output: t2.c1, t2.c2, t1.c1, t1.c2
  -> Partition Iterator
    Output: t2.c1, t2.c2
    Iterations: 3
    -> Partitioned Index Scan using t2_c1 on public.t2
      Output: t2.c1, t2.c2
      Selected Partitions: 1..3
  -> Partition Iterator
    Output: t1.c1, t1.c2
    Iterations: PART
    -> Partitioned Index Scan using t1_c1 on public.t1
      Output: t1.c1, t1.c2
      Index Cond: (t1.c1 = t2.c1)
      Selected Partitions: 1 (ppi-pruning)

(23 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 < t2.c1;
QUERY PLAN
-----
Streaming (type: GATHER)
  Output: t2.c1, t2.c2, t1.c1, t1.c2
  Node/s: All datanodes
  -> Nested Loop
    Output: t2.c1, t2.c2, t1.c1, t1.c2
    -> Streaming(type: BROADCAST)
      Output: t2.c1, t2.c2
      Spawn on: All datanodes
      Consumer Nodes: All datanodes
    -> Partition Iterator
      Output: t2.c1, t2.c2
      Iterations: 3
      -> Partitioned Seq Scan on public.t2
        Output: t2.c1, t2.c2
```

```
          Distribute Key: t2.c1
          Selected Partitions: 1..3
-> Partition Iterator
   Output: t1.c1, t1.c2
   Iterations: PART
-> Partitioned Index Scan using t1_c1 on public.t1
   Output: t1.c1, t1.c2
   Distribute Key: t1.c1
   Index Cond: (t1.c1 < t2.c1)
   Selected Partitions: 1 (ppi-pruning)
(24 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 < t2.c1;
          QUERY PLAN
```

```
-----
Streaming (type: GATHER)
  Output: t2.c1, t2.c2, t1.c1, t1.c2
  Node/s: All datanodes
-> Nested Loop
   Output: t2.c1, t2.c2, t1.c1, t1.c2
-> Streaming(type: BROADCAST)
   Output: t2.c1, t2.c2
   Spawn on: All datanodes
   Consumer Nodes: All datanodes
-> Partition Iterator
   Output: t2.c1, t2.c2
   Iterations: 3
   -> Partitioned Seq Scan on public.t2
       Output: t2.c1, t2.c2
       Distribute Key: t2.c1
       Selected Partitions: 1..3
-> Partition Iterator
   Output: t1.c1, t1.c2
   Iterations: PART
   -> Partitioned Index Scan using t1_c1 on public.t1
       Output: t1.c1, t1.c2
       Distribute Key: t1.c1
       Index Cond: (t1.c1 > t2.c1)
       Selected Partitions: 1..3 (ppi-pruning)
(24 rows)
```

b. 逻辑表达式

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 = t2.c1 AND t1.c2 = 2;
          QUERY
```

PLAN

```
-----
Data Node Scan
  Output: t2.c1, t2.c2, t1.c1, t1.c2
  Node/s: All datanodes
  Remote query: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1,
t1.c2 FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1 AND t1.c2 = 2

Remote SQL: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1, t1.c2
FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1 AND t1.c2 = 2
Datanode Name: datanode1
Nested Loop
  Output: t2.c1, t2.c2, t1.c1, t1.c2
-> Partition Iterator
   Output: t1.c1, t1.c2
   Iterations: 3
   -> Partitioned Index Scan using t1_c2 on public.t1
       Output: t1.c1, t1.c2
       Index Cond: (t1.c2 = 2)
       Selected Partitions: 1..3
-> Partition Iterator
   Output: t2.c1, t2.c2
   Iterations: PART
```

```
-> Partitioned Index Scan using t2_c1 on public.t2
Output: t2.c1, t2.c2
Index Cond: (t2.c1 = t1.c1)
Selected Partitions: 1..3 (ppi-pruning)
```

(24 rows)

- 参数化路径动态剪枝不支持的典型场景具体示例如下:

a. BitmapOr/BitmapAnd算子

```
gaussdb=# set enable_seqscan=off;
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) */ * FROM t2 JOIN
t1 ON t1.c1 = t2.c1 OR t1.c2 = 2;
WARNING: Statistics in some tables or columns(public.t2.c1, public.t1.c1, public.t1.c2) are not
collected.
```

HINT: Do analyze for them in order to generate optimized plan.

QUERY PLAN

```
-----
Streaming (type: GATHER)
Output: t2.c1, t2.c2, t1.c1, t1.c2
Node/s: All datanodes
-> Nested Loop
Output: t2.c1, t2.c2, t1.c1, t1.c2
-> Streaming(type: BROADCAST)
Output: t2.c1, t2.c2
Spawn on: All datanodes
Consumer Nodes: All datanodes
-> Partition Iterator
Output: t2.c1, t2.c2
Iterations: 3
-> Partitioned Seq Scan on public.t2
Output: t2.c1, t2.c2
Distribute Key: t2.c1
Selected Partitions: 1..3
-> Partition Iterator
Output: t1.c1, t1.c2
Iterations: 3
-> Partitioned Bitmap Heap Scan on public.t1
Output: t1.c1, t1.c2
Distribute Key: t1.c1
Recheck Cond: ((t1.c1 = t2.c1) OR (t1.c2 = 2))
Selected Partitions: 1..3
-> BitmapOr
-> Partitioned Bitmap Index Scan on t1_c1
Index Cond: (t1.c1 = t2.c1)
-> Partitioned Bitmap Index Scan on t1_c2
Index Cond: (t1.c2 = 2)
```

(29 rows)

b. 隐式转换

```
gaussdb=# CREATE TABLE t3(c1 TEXT, c2 INT);
CREATE TABLE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 JOIN t3 ON t1.c1 = t3.c1;
WARNING: Statistics in some tables or columns(public.t1.c1, public.t3.c1) are not collected.
```

HINT: Do analyze for them in order to generate optimized plan.

QUERY PLAN

```
-----
Streaming (type: GATHER)
Output: t1.c1, t1.c2, t3.c1, t3.c2
Node/s: All datanodes
-> Nested Loop
Output: t1.c1, t1.c2, t3.c1, t3.c2
Join Filter: (t1.c1 = ((t3.c1)::bigint))
-> Partition Iterator
Output: t1.c1, t1.c2
Iterations: 3
-> Partitioned Index Scan using t1_c1 on public.t1
Output: t1.c1, t1.c2
Distribute Key: t1.c1
Selected Partitions: 1..3
-> Materialize
```

```
Output: t3.c1, t3.c2, ((t3.c1)::bigint)
-> Streaming(type: REDISTRIBUTE)
Output: t3.c1, t3.c2, ((t3.c1)::bigint)
Distribute Key: ((t3.c1)::bigint)
Spawn on: All datanodes
Consumer Nodes: All datanodes
-> Seq Scan on public.t3
Output: t3.c1, t3.c2, t3.c1
Distribute Key: t3.c1

(23 rows)
```

c. 函数

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 JOIN t3 ON t1.c1 =
LENGTHB(t3.c1);
QUERY PLAN
-----
Nested Loop
Output: t1.c1, t1.c2, t3.c1, t3.c2
-> Seq Scan on public.t3
Output: t3.c1, t3.c2
-> Partition Iterator
Output: t1.c1, t1.c2
Iterations: 3
-> Partitioned Index Scan using t1_c1 on public.t1
Output: t1.c1, t1.c2
Index Cond: (t1.c1 = lengthb(t3.c1))
Selected Partitions: 1..3

(11 rows)

--清理示例环境。
gaussdb=# DROP TABLE t1;
gaussdb=# DROP TABLE t2;
gaussdb=# DROP TABLE t3;
```

### 3.3.2 分区索引

分区表上的索引共有三种类型:

1. Global Non-Partitioned Index
2. Global Partitioned Index
3. Local Partitioned Index

目前GaussDB支持Global Non-Partitioned Index和Local Partitioned Index类型索引。

图 3-3 Global Non-Partitioned Index

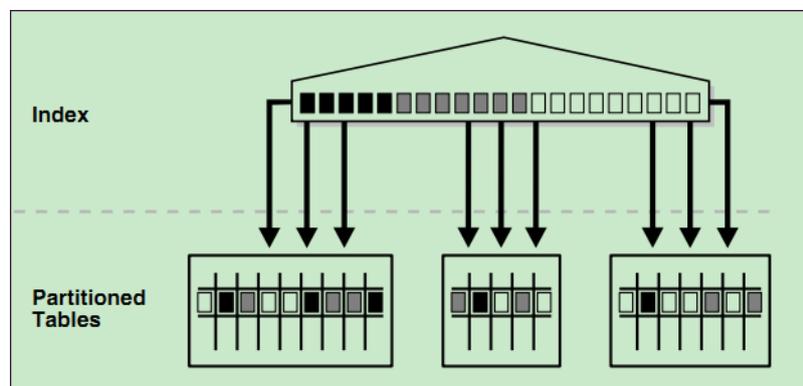


图 3-4 Global Partitioned Index

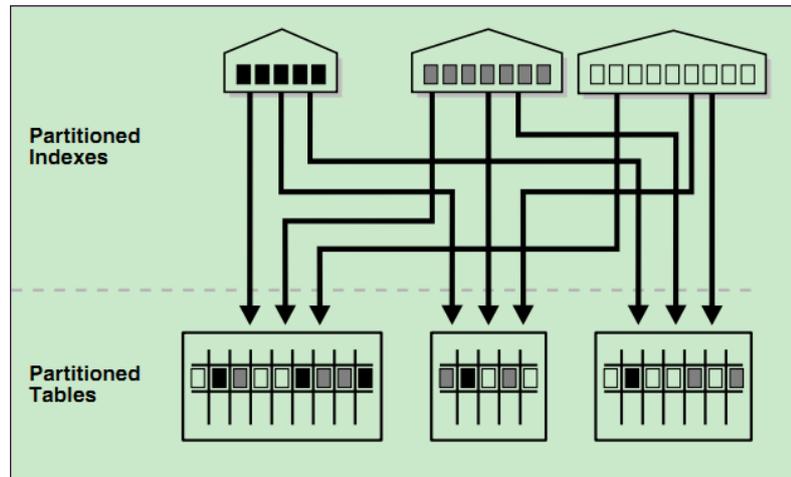
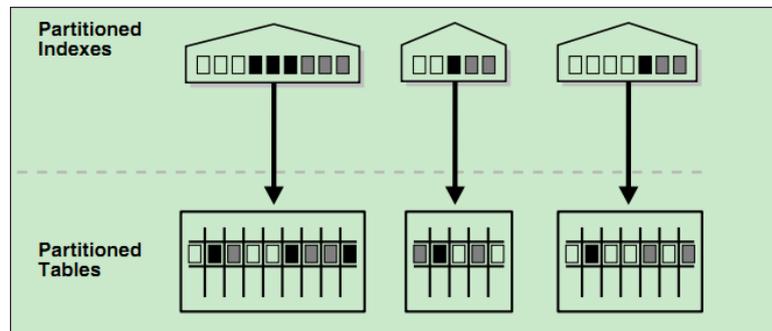


图 3-5 Local Partitioned Index



## 约束

- 分区表索引分为LOCAL索引与GLOBAL索引：LOCAL索引与某个具体分区绑定，而GLOBAL索引则对应整个分区表。
- 唯一约束和主键约束的约束键包含所有分区键则创建LOCAL索引，否则创建GLOBAL索引。

### 说明

当查询语句在查询数据涉及多个分区时，建议使用GLOBAL索引，反之建议使用LOCAL索引。但需要注意GLOBAL索引在分区维护语法中存在额外的开销。

## 示例

- 创建表  
gaussdb=# CREATE TABLE web\_returns\_p2  
(  
ca\_address\_sk INTEGER NOT NULL ,  
ca\_address\_id CHARACTER(16) NOT NULL ,  
ca\_street\_number CHARACTER(10) ,  
ca\_street\_name CHARACTER VARYING(60) ,  
ca\_street\_type CHARACTER(15) ,  
ca\_suite\_number CHARACTER(10) ,  
ca\_city CHARACTER VARYING(60) ,  
ca\_county CHARACTER VARYING(30) ,  
ca\_state CHARACTER(2) ,  
ca\_zip CHARACTER(10) ,

```
ca_country CHARACTER VARYING(20) ,
ca_gmt_offset NUMERIC(5,2) ,
ca_location_type CHARACTER(20)
)
PARTITION BY RANGE (ca_address_sk)
(
PARTITION P1 VALUES LESS THAN(5000),
PARTITION P2 VALUES LESS THAN(10000),
PARTITION P3 VALUES LESS THAN(15000),
PARTITION P4 VALUES LESS THAN(20000),
PARTITION P5 VALUES LESS THAN(25000),
PARTITION P6 VALUES LESS THAN(30000),
PARTITION P7 VALUES LESS THAN(40000),
PARTITION P8 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

- 创建索引

- 创建分区表LOCAL索引tpcds\_web\_returns\_p2\_index1，不指定索引分区的名

```
gaussdb=# CREATE INDEX tpcds_web_returns_p2_index1 ON web_returns_p2 (ca_address_id)
LOCAL;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE INDEX
```

- 创建分区表LOCAL索引tpcds\_web\_returns\_p2\_index2，并指定索引分区的名

```
gaussdb=# CREATE TABLESPACE example2 LOCATION '/home/omm/example2';
gaussdb=# CREATE TABLESPACE example3 LOCATION '/home/omm/example3';
gaussdb=# CREATE TABLESPACE example4 LOCATION '/home/omm/example4';
```

```
gaussdb=# CREATE INDEX tpcds_web_returns_p2_index2 ON web_returns_p2 (ca_address_sk)
LOCAL
```

```
(
PARTITION web_returns_p2_P1_index,
PARTITION web_returns_p2_P2_index TABLESPACE example3,
PARTITION web_returns_p2_P3_index TABLESPACE example4,
PARTITION web_returns_p2_P4_index,
PARTITION web_returns_p2_P5_index,
PARTITION web_returns_p2_P6_index,
PARTITION web_returns_p2_P7_index,
PARTITION web_returns_p2_P8_index
) TABLESPACE example2;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE INDEX
```

- 创建分区表GLOBAL索引tpcds\_web\_returns\_p2\_global\_index。

```
gaussdb=# CREATE INDEX tpcds_web_returns_p2_global_index ON web_returns_p2
(ca_street_number) GLOBAL;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE INDEX
```

- 修改索引分区的表空间

- 修改索引分区web\_returns\_p2\_P2\_index的表空间为example1。

```
gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 MOVE PARTITION
web_returns_p2_P2_index TABLESPACE example1;
```

当结果显示为如下信息，则表示修改成功。

```
ALTER INDEX
```

- 修改索引分区web\_returns\_p2\_P3\_index的表空间为example2。

```
gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 MOVE PARTITION
web_returns_p2_P3_index TABLESPACE example2;
```

当结果显示为如下信息，则表示修改成功。

```
ALTER INDEX
```

- 重命名索引分区
  - 执行如下命令对索引分区web\_returns\_p2\_P8\_index重命名web\_returns\_p2\_P8\_index\_new。

```
gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 RENAME PARTITION web_returns_p2_P8_index TO web_returns_p2_P8_index_new;
```

当结果显示为如下信息，则表示重命名成功。

```
ALTER INDEX
```
- 查询索引
  - 执行如下命令查询系统和用户定义的所有索引。

```
gaussdb=# SELECT RELNAME FROM PG_CLASS WHERE RELKIND='i' or RELKIND='I';
```
  - 执行如下命令查询指定索引的信息。

```
gaussdb=# \di+ tpcds_web_returns_p2_index2
```
- 删除索引

```
gaussdb=# DROP INDEX tpcds_web_returns_p2_index1;
```

当结果显示为如下信息，则表示删除成功。

```
DROP INDEX
```

清理示例

```
gaussdb=# DROP TABLE web_returns_p2;
```

## 3.4 分区表运维管理

分区表运维管理包括分区管理、分区表管理、分区索引管理和分区表业务并发支持等。

- 分区管理：也称分区级DDL，包括新增（Add）、删除（Drop）、交换（Exchange）、清空（Truncate）、分割（Split）、合并（Merge）、移动（Move）、重命名（Rename）共8种。

### 注意

- 对于哈希分区，涉及分区数的变更会导致数据re-shuffling，故当前GaussDB不支持导致Hash分区数变更的操作，包括新增（Add）、删除（Drop）、分割（Split）、合并（Merge）这4种。
- 涉及分区数据变更的操作会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，包括删除（Drop）、交换（Exchange）、清空（Truncate）、分割（Split）、合并（Merge）这5种。

### 说明

- 大部分分区DDL支持partition和partition for指定分区两种写法，前者需要指定分区名，后者需要指定分区定义范围内的任一分区值。比如假设分区part1的范围定义为[100, 200)，那么partition part1和partition for(150)这两种写法是等价的。
- 不同分区DDL的执行代价各不相同，由于在执行分区DDL过程中目标分区会被锁住，用户需要评估其代价以及对业务的影响。一般而言，分割（Split）、合并（Merge）的执行代价远大于其他分区DDL，与源分区的大小正相关；交换（Exchange）的代价主要源于Global索引的重建和validation校验；移动（Move）的代价限制于磁盘I/O；其余分区DDL的执行代价都很低。
- 分区表管理：除了继承普通表的功能外，还支持开启/关闭分区表行迁移的功能。
- 分区索引管理：支持用户设置索引/索引分区不可用，或者重建不可用的索引/索引分区，比如由于分区管理操作导致的Global索引失效场景。

- 分区表业务并发支持：分布式分区表的DDL操作会锁全表，不支持跨分区DDL-DQL/DML并发。

### 3.4.1 新增分区

用户可以在已建立的分区表中新增分区，来维护新业务的进行。当前各种分区表支持的分区上限为1048575，如果达到了上限则不能继续添加分区。同时需要考虑分区占用内存的开销，分区表使用内存大致为（分区数 \* 3 / 1024）MB，分区占用内存不允许大于local\_syscache\_threshold的值，同时还需要预留部分空间以供其他功能使用。

#### ⚠ 注意

- 新增分区不能作用于HASH分区上。

#### 3.4.1.1 向范围分区表新增分区

使用ALTER TABLE ADD PARTITION可以将分区添加到现有分区表的最后面，新增分区的上界值必须大于当前最后一个分区的上界值。

例如，对范围分区表range\_sales新增一个分区。

```
ALTER TABLE range_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01') TABLESPACE tb1;
```

#### 须知

当范围分区表有MAXVALUE分区时，无法新增分区。可以使用ALTER TABLE SPLIT PARTITION命令分割分区。分割分区同样适用于需要在现有分区表的前面/中间添加分区的情形，参考[对范围分区表分割分区](#)。

#### 3.4.1.2 向列表分区表新增分区

使用ALTER TABLE ADD PARTITION可以在列表分区表中新增分区，新增分区的枚举值不能与已有的任一个分区的枚举值重复。

例如，对列表分区表list\_sales新增一个分区。

```
ALTER TABLE list_sales ADD PARTITION channel5 VALUES ('X') TABLESPACE tb1;
```

#### 须知

当列表分区表有DEFAULT分区时，无法新增分区。可以使用ALTER TABLE SPLIT PARTITION命令分割分区。

### 3.4.2 删除分区

用户可以使用删除分区的命令来移除不需要的分区。删除分区可以通过指定分区名或者分区值来进行。

#### ⚠ 注意

- 删除分区不能作用于HASH分区上。
- 执行删除分区命令会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，或者用户自行重建Global索引。

使用ALTER TABLE DROP PARTITION可以删除指定分区表的任何一个分区，这个行为可以作用在范围分区表、列表分区表上。

例如，通过指定分区名删除范围分区表range\_sales的分区date\_202005，并更新Global索引。

```
ALTER TABLE range_sales DROP PARTITION date_202005 UPDATE GLOBAL INDEX;
```

或者，通过指定分区值来删除范围分区表range\_sales中'2020-05-08'所对应的分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

```
ALTER TABLE range_sales DROP PARTITION FOR ('2020-05-08');
```

#### 须知

- 当分区表只有一个分区时，不支持通过ALTER TABLE DROP PARTITION命令删除分区。
- 当分区表为哈希分区表时，不支持通过ALTER TABLE DROP PARTITION命令删除分区。

### 3.4.3 交换分区

用户可以使用交换分区的命令来将分区与普通表的数据进行交换。交换分区可以快速将数据导入/导出分区表，实现数据高效加载的目的。在业务迁移的场景，使用交换分区比常规导入会快很多。交换分区可以通过指定分区名或者分区值来进行。

#### ⚠ 注意

- 执行交换分区命令会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，或者用户自行重建Global索引。

#### 须知

- 执行交换分区时，可以申明WITH/WITHOUT VALIDATION，表明是否校验普通表数据满足目标分区的分区键约束规则（默认校验）。数据校验活动开销较大，如果能确保交换的数据属于目标分区，可以申明WITHOUT VALIDATION来提高交换性能。
- 可以申明WITH VALIDATION VERBOSE，此时数据库会校验普通表的每一行，将不满足目标分区的分区键约束规则的数据，插入到分区表的其他分区中，最后再进行普通表与目标分区的交换。

例如，给出如下分区定义和普通表exchange\_sales的数据分布，并将分区DATE\_202001和普通表exchange\_sales做交换，则根据申明子句的不同，存在以下三种行为：

- 申明WITHOUT VALIDATION，数据全部交换到分区DATE\_202001中，由于'2020-02-03', '2020-04-08'不满足分区DATE\_202001的范围约束，后续业务可能会出现异常。
- 申明WITH VALIDATION，由于'2020-02-03', '2020-04-08'不满足分区DATE\_202001的范围约束，数据库给出相应的报错。
- 申明WITH VALIDATION VERBOSE，数据库会将'2020-02-03'插入分区DATE\_202002，将'2020-04-08'插入分区DATE\_202004，再将剩下的数据交换到分区DATE\_202001中。

```
--分区定义
PARTITION DATE_202001 VALUES LESS THAN ('2020-02-01'),
PARTITION DATE_202002 VALUES LESS THAN ('2020-03-01'),
PARTITION DATE_202003 VALUES LESS THAN ('2020-04-01'),
PARTITION DATE_202004 VALUES LESS THAN ('2020-05-01')
-- exchange_sales的数据分布
('2020-01-15', '2020-01-17', '2020-01-23', '2020-02-03', '2020-04-08')
```

#### 警告

如果交换的数据不完全属于目标分区，请不要申明WITHOUT VALIDATION交换分区，否则会破坏分区约束规则，导致分区表后续DML业务结果异常。

进行交换的普通表和分区必须满足如下条件：

- 普通表和分区的列数目相同，对应列的信息严格一致。
- 普通表和分区的表压缩信息严格一致。
- 普通表索引和分区Local索引个数相同，且对应索引的信息严格一致。
- 普通表和分区的表约束个数相同，且对应表约束的信息严格一致。
- 普通表不可以是临时表。
- 普通表和分区表上不可以有动态数据脱敏，行访问控制约束。

使用ALTER TABLE EXCHANGE PARTITION可以对分区表交换分区。

例如，通过指定分区名将范围分区表range\_sales的分区date\_202001和普通表exchange\_sales进行交换，不进行分区键校验，并更新Global索引。

```
ALTER TABLE range_sales EXCHANGE PARTITION (date_202001) WITH TABLE exchange_sales WITHOUT VALIDATION UPDATE GLOBAL INDEX;
```

或者，通过指定分区值将范围分区表range\_sales中'2020-01-08'所对应的分区和普通表exchange\_sales进行交换，进行分区校验并将不满足目标分区约束的数据插入到分区表的其他分区中。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

```
ALTER TABLE range_sales EXCHANGE PARTITION FOR ('2020-01-08') WITH TABLE exchange_sales WITH VALIDATION VERBOSE;
```

### 3.4.4 清空分区

用户可以使用清空分区的命令来快速清空分区的数据。与删除分区功能类似，区别在于清空分区只会删除分区中的数据，分区的定义和物理文件都会保留。清空分区可以通过指定分区名或者分区值来进行。

#### ⚠ 注意

- 执行清空分区命令会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，或者用户自行重建Global索引。

使用ALTER TABLE TRUNCATE PARTITION可以清空指定分区表的任何一个分区。

例如，通过指定分区名清空范围分区表range\_sales的分区date\_202005，并更新Global索引。

```
ALTER TABLE range_sales TRUNCATE PARTITION date_202005 UPDATE GLOBAL INDEX;
```

或者，通过指定分区值来清空范围分区表range\_sales中'2020-05-08'所对应的分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

```
ALTER TABLE range_sales TRUNCATE PARTITION FOR ('2020-05-08');
```

## 3.4.5 分割分区

用户可以使用分割分区的命令来将一个分区分割为两个或多个新分区。当分区数据太大，或者需要对有MAXVALUE的范围分区/DEFAULT的列表分区新增分区时，可以考虑执行该操作。分割分区可以指定分割点将一个分区分割为两个新分区，也可以不指定分割点将一个分区分割为多个新分区。分割分区可以通过指定分区名或者分区值来进行。

#### ⚠ 注意

- 分割分区不能作用于哈希分区上。
- 执行分割分区命令会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，或者用户自行重建Global索引。

#### 须知

分割后的新分区，可以与源分区名字相同，比如将分区p1分割为p1,p2。但数据库不会将分割前后相同名的分区视为同一个分区。

### 3.4.5.1 对范围分区表分割分区

使用ALTER TABLE SPLIT PARTITION可以对范围分区表分割分区。

例如，假设范围分区表range\_sales的分区date\_202001定义范围为['2020-01-01', '2020-02-01')。可以指定分割点'2020-01-16'将分区date\_202001分割为两个分区，并更新Global索引。

```
ALTER TABLE range_sales SPLIT PARTITION date_202001 AT ('2020-01-16') INTO  
(  
    PARTITION date_202001_p1, --第一个分区上界是'2020-01-16'  
    PARTITION date_202001_p2 --第二个分区上界是'2020-02-01'  
) UPDATE GLOBAL INDEX;
```

或者，不指定分割点，将分区date\_202001分割为多个分区，并更新Global索引。

```
ALTER TABLE range_sales SPLIT PARTITION date_202001 INTO  
(  
    PARTITION date_202001_p1 VALUES LESS THAN ('2020-01-11'),  
    PARTITION date_202001_p2 VALUES LESS THAN ('2020-01-21'),
```

```
PARTITION date_202001_p3 --第三个分区上界是'2020-02-01'  
)UPDATE GLOBAL INDEX;
```

又或者，通过指定分区值而不是指定分区名来分割分区。

```
ALTER TABLE range_sales SPLIT PARTITION FOR ('2020-01-15') AT ('2020-01-16') INTO  
(  
    PARTITION date_202001_p1, --第一个分区上界是'2020-01-16'  
    PARTITION date_202001_p2 --第二个分区上界是'2020-02-01'  
) UPDATE GLOBAL INDEX;
```

#### 须知

若对MAXVALUE分区进行分割，前面几个分区不能申明MAXVALUE范围，最后一个分区会继承MAXVALUE分区范围。

### 3.4.5.2 对列表分区表分割分区

使用ALTER TABLE SPLIT PARTITION可以对列表分区表分割分区。

例如，假设列表分区表list\_sales的分区channel2定义范围为('6', '7', '8', '9')。可以指定分割点('6', '7')将分区channel2分割为两个分区，并更新Global索引。

```
ALTER TABLE list_sales SPLIT PARTITION channel2 VALUES ('6', '7') INTO  
(  
    PARTITION channel2_1, --第一个分区范围是('6', '7')  
    PARTITION channel2_2 --第二个分区范围是('8', '9')  
) UPDATE GLOBAL INDEX;
```

或者，不指定分割点，将分区channel2分割为多个分区，并更新Global索引。

```
ALTER TABLE list_sales SPLIT PARTITION channel2 INTO  
(  
    PARTITION channel2_1 VALUES ('6'),  
    PARTITION channel2_2 VALUES ('8'),  
    PARTITION channel2_3 --第三个分区范围是('7', '9')  
) UPDATE GLOBAL INDEX;
```

又或者，通过指定分区值而不是指定分区名来分割分区。

```
ALTER TABLE list_sales SPLIT PARTITION FOR ('6') VALUES ('6', '7') INTO  
(  
    PARTITION channel2_1, --第一个分区范围是('6', '7')  
    PARTITION channel2_2 --第二个分区范围是('8', '9')  
) UPDATE GLOBAL INDEX;
```

#### 注意

若对DEFAULT分区进行分割，前面几个分区不能申明DEFAULT范围，最后一个分区会继承DEFAULT分区范围。

### 3.4.6 合并分区

用户可以使用合并分区的命令来将多个分区合并为一个分区。合并分区只能通过指定分区名来进行，不支持指定分区值的写法。

#### ⚠ 注意

- 合并分区不能作用于哈希分区上。
- 执行合并分区命令会使得Global索引失效，可以通过UPDATE GLOBAL INDEX子句来同步更新Global索引，或者用户自行重建Global索引。

#### 须知

合并后的新分区，对于范围分区，可以与最后一个源分区名字相同，比如将p1,p2合并为p2；对于列表分区，可以与任一源分区名字相同，比如将p1,p2合并为p1。

如果新分区与源分区名字相同，数据库会将新分区视为对源分区的继承。

使用ALTER TABLE MERGE PARTITIONS可以将多个分区合并为一个分区。

例如，将范围分区表range\_sales的分区date\_202001和date\_202002合并为一个新的分区，并更新Global索引。

```
ALTER TABLE range_sales MERGE PARTITIONS date_202001, date_202002 INTO  
PARTITION date_2020_old UPDATE GLOBAL INDEX;
```

### 3.4.7 移动分区

用户可以使用移动分区的命令来将一个分区移动到新的表空间中。移动分区可以通过指定分区名或者分区值来进行。

使用ALTER TABLE MOVE PARTITION可以对分区表移动分区。

例如，通过指定分区名将范围分区表range\_sales的分区date\_202001移动到表空间tb1中。

```
ALTER TABLE range_sales MOVE PARTITION date_202001 TABLESPACE tb1;
```

或者，通过指定分区值将列表分区表list\_sales中'0'所对应的分区移动到表空间tb1中。

```
ALTER TABLE list_sales MOVE PARTITION FOR ('0') TABLESPACE tb1;
```

### 3.4.8 重命名分区

用户可以使用重命名分区的命令来将一个分区命名为新的名称。重命名分区可以通过指定分区名或者分区值来进行。

#### 3.4.8.1 对分区表重命名分区

使用ALTER TABLE RENAME PARTITION可以对分区表重命名分区。

例如，通过指定分区名将范围分区表range\_sales的分区date\_202001重命名。

```
ALTER TABLE range_sales RENAME PARTITION date_202001 TO date_202001_new;
```

或者，通过指定分区值将列表分区表list\_sales中'0'所对应的分区重命名。

```
ALTER TABLE list_sales RENAME PARTITION FOR ('0') TO channel_new;
```

#### 3.4.8.2 对 Local 索引重命名索引分区

使用ALTER INDEX RENAME PARTITION可以对Local索引重命名索引分区。具体方法与一级分区表重命名分区相同。

### 3.4.9 分区表行迁移

用户可以使用ALTER TABLE ENABLE/DISABLE ROW MOVEMENT来开启/关闭分区表行迁移。

开启行迁移时，允许通过更新操作将一个分区中的数据迁移到另一个分区中；关闭行迁移时，如果出现这种更新行为，则业务报错。

#### 须知

如果业务明确不允许对分区键所在列进行更新操作，建议关闭分区表行迁移。

例如，创建列表分区表，并开启分区表行迁移，此时可以跨分区更新分区键所在列；关闭分区表行迁移后，对分区键所在列进行跨分区更新会业务报错。

```
CREATE TABLE list_sales
(
  product_id INT4 NOT NULL,
  customer_id INT4 PRIMARY KEY,
  time_id DATE,
  channel_id CHAR(1),
  type_id INT4,
  quantity_sold NUMERIC(3),
  amount_sold NUMERIC(10,2)
)
PARTITION BY LIST (channel_id)
(
  PARTITION channel1 VALUES ('0', '1', '2'),
  PARTITION channel2 VALUES ('3', '4', '5'),
  PARTITION channel3 VALUES ('6', '7'),
  PARTITION channel4 VALUES ('8', '9')
) ENABLE ROW MOVEMENT;
INSERT INTO list_sales VALUES (153241,65143129,'2021-05-07','0',864134,89,34);
--跨分区更新成功，数据从分区channel1迁移到分区channel2
UPDATE list_sales SET channel_id = '3' WHERE channel_id = '0';
--关闭分区表行迁移
ALTER TABLE list_sales DISABLE ROW MOVEMENT;
--跨分区更新失败，报错fail to update partitioned table "list_sales"
UPDATE list_sales SET channel_id = '0' WHERE channel_id = '3';
--分区内更新依然成功
UPDATE list_sales SET channel_id = '4' WHERE channel_id = '3';
```

### 3.4.10 分区表索引重建/不可用

用户可以通过命令使得一个分区表索引或者一个索引分区不可用，此时该索引/索引分区不再维护；使用重建索引命令可以重建分区表索引，恢复索引的正常功能。

此外，部分分区级DDL操作也会使得Global索引失效，包括删除drop、交换exchange、清空truncate、分割split、合并merge，如果在DDL操作中带UPDATE GLOBAL INDEX子句，则会同步更新Global索引，否则需要用户自行重建索引。

#### 3.4.10.1 索引重建/不可用

使用ALTER INDEX可以设置索引是否可用。

例如，假设分区表range\_sales上存在索引range\_sales\_idx，可以通过如下命令设置其不可用。

```
ALTER INDEX range_sales_idx UNUSABLE;
```

可以使用如下命令重建索引range\_sales\_idx。

```
ALTER INDEX range_sales_idx REBUILD;
```

### 3.4.10.2 Local 索引分区重建/不可用

- 使用ALTER INDEX PARTITION可以设置Local索引分区是否可用。
- 使用ALTER TABLE MODIFY PARTITION可以设置分区表上指定分区的所有索引分区是否可用。

例如，假设分区表range\_sales上存在两张Local索引range\_sales\_idx1和range\_sales\_idx2，假设其在分区date\_202001上对应的索引分区名分别为range\_sales\_idx1\_part1和range\_sales\_idx2\_part1。

下面给出了维护分区表分区索引的语法：

- 可以通过如下命令设置分区date\_202001上的所有索引分区均不可用。  
`ALTER TABLE range_sales MODIFY PARTITION date_202001 UNUSABLE LOCAL INDEXES;`
- 或者通过如下命令单独设置分区date\_202001上的索引分区range\_sales\_idx1\_part1不可用。  
`ALTER INDEX range_sales_idx1 MODIFY PARTITION range_sales_idx1_part1 UNUSABLE;`
- 可以通过如下命令重建分区date\_202001上的所有索引分区。  
`ALTER TABLE range_sales MODIFY PARTITION date_202001 REBUILD UNUSABLE LOCAL INDEXES;`
- 或者通过如下命令单独重建分区date\_202001上的索引分区range\_sales\_idx1\_part1。  
`ALTER INDEX range_sales_idx1 REBUILD PARTITION range_sales_idx1_part1;`

## 3.5 分区表系统视图&DFX

### 3.5.1 分区表相关系统视图

分区表系统视图根据权限分为3类，具体字段信息请参考《开发指南》中“系统表和系统视图 > 系统视图”章节。

1. 所有分区视图：
  - ADM\_PART\_TABLES：所有分区表信息。
  - ADM\_TAB\_PARTITIONS：所有分区信息。
  - ADM\_PART\_INDEXES：所有Local索引信息。
  - ADM\_IND\_PARTITIONS：所有索引分区信息。
2. 当前用户可访问的视图：
  - DB\_PART\_TABLES：当前用户可访问的分区表信息。
  - DB\_TAB\_PARTITIONS：当前用户可访问的分区信息。
  - DB\_PART\_INDEXES：当前用户可访问的Local索引信息。
  - DB\_IND\_PARTITIONS：当前用户可访问的索引分区信息。
3. 当前用户拥有的视图：
  - MY\_PART\_TABLES：当前用户拥有的分区表信息。
  - MY\_TAB\_PARTITIONS：当前用户拥有的分区信息。
  - MY\_PART\_INDEXES：当前用户拥有的Local索引信息。
  - MY\_IND\_PARTITIONS：当前用户拥有的索引分区信息。

## 3.5.2 分区表相关内置工具函数

### 前置建表相关信息

- 前置建表:

```
CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE (a)
(
  PARTITION p1 VALUES LESS THAN (2000),
  PARTITION p2 VALUES LESS THAN (3000),
  partition p3 VALUES LESS THAN (4000),
  partition p4 VALUES LESS THAN (5000),
  partition p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;
```

- 查看分区表oid

```
SELECT oid FROM pg_class WHERE relname = 'test_range_pt';
oid
-----
49290
(1 row)
```

- 查看分区信息

```
SELECT oid,relname,parttype,parentid,boundaries FROM pg_partition WHERE parentid = 49290;
oid | relname | parttype | parentid | boundaries
-----+-----+-----+-----+-----
49293 | test_range_pt | r | 49290 |
49294 | p1 | p | 49290 | {2000}
49295 | p2 | p | 49290 | {3000}
49296 | p3 | p | 49290 | {4000}
49297 | p4 | p | 49290 | {5000}
49298 | p5 | p | 49290 | {NULL}
(6 rows)
```

- 创建索引

```
CREATE INDEX idx_range_a ON test_range_pt(a) LOCAL;
CREATE INDEX
--查看分区索引|oid
SELECT oid FROM pg_class WHERE relname = 'idx_range_a';
oid
-----
90250
(1 row)
```

- 查看索引分区信息

```
SELECT oid,relname,parttype,parentid,boundaries,indextblid FROM pg_partition WHERE parentid = 90250;
oid | relname | parttype | parentid | boundaries | indextblid
-----+-----+-----+-----+-----+-----
90255 | p5_a_idx | x | 90250 | | 49298
90254 | p4_a_idx | x | 90250 | | 49297
90253 | p3_a_idx | x | 90250 | | 49296
90252 | p2_a_idx | x | 90250 | | 49295
90251 | p1_a_idx | x | 90250 | | 49294
(5 rows)
```

### 工具函数示例

- pg\_get\_tabledef获取分区表的定义，入参可以为表的oid或者表名。

```
SELECT pg_get_tabledef('test_range_pt');
pg_get_tabledef
-----
SET search_path = public;
CREATE TABLE test_range_pt (
  a integer,
  b integer,
  c integer
)
```

```
WITH (orientation=row, compression=no)
PARTITION BY RANGE (a)
(
  PARTITION p1 VALUES LESS THAN (2000) TABLESPACE pg_default,
  PARTITION p2 VALUES LESS THAN (3000) TABLESPACE pg_default,
  PARTITION p3 VALUES LESS THAN (4000) TABLESPACE pg_default,
  PARTITION p4 VALUES LESS THAN (5000) TABLESPACE pg_default,
  PARTITION p5 VALUES LESS THAN (MAXVALUE) TABLESPACE pg_default+
)
ENABLE ROW MOVEMENT;
(1 row)
```

- pg\_stat\_get\_partition\_tuples\_hot\_updated返回给定分区id的分区热更新元组数的统计。

在分区p1中插入10条数据并更新，统计分区p1的热更新元组数。

```
INSERT INTO test_range_pt VALUES(generate_series(1,10),1,1);
INSERT 0 10
SELECT pg_stat_get_partition_tuples_hot_updated(49294);
pg_stat_get_partition_tuples_hot_updated
-----
0
(1 row)
UPDATE test_range_pt SET b = 2;
UPDATE 10
SELECT pg_stat_get_partition_tuples_hot_updated(49294);
pg_stat_get_partition_tuples_hot_updated
-----
10
(1 row)
```

- pg\_partition\_size(oid,oid)指定OID代表的分区使用的磁盘空间。其中，第一个oid为表的OID，第二个oid为分区的OID。

查看分区p1的磁盘空间。

```
SELECT pg_partition_size(49290, 49294);
pg_partition_size
-----
90112
(1 row)
```

- pg\_partition\_size(text, text) 指定名称的分区使用的磁盘空间。其中，第一个text为表名，第二个text为分区名。

查看分区p1的磁盘空间。

```
SELECT pg_partition_size('test_range_pt', 'p1');
pg_partition_size
-----
90112
(1 row)
```

- pg\_partition\_indexes\_size(oid,oid)指定OID代表的分区的索引使用的磁盘空间。其中，第一个oid为表的OID，第二个oid为分区的OID。

查看分区p1的索引分区磁盘空间。

```
SELECT pg_partition_indexes_size(49290, 49294);
pg_partition_indexes_size
-----
204800
(1 row)
```

- pg\_partition\_indexes\_size(text,text) 指定名称的分区的索引使用的磁盘空间。其中，第一个text为表名，第二个text为分区名。

查看分区p1的索引分区磁盘空间。

```
SELECT pg_partition_indexes_size('test_range_pt', 'p1');
pg_partition_indexes_size
-----
```

```
204800  
(1 row)
```

- `pg_partition_filenum(partition_oid)` 获取到指定分区表的oid所对应的filenum。  
查看分区p1的filenum。

```
SELECT pg_partition_filenum(49294);  
pg_partition_filenum  
-----  
49294  
(1 row)
```

- `pg_partition_filepath(partition_oid)` 指定分区的文件路径名。  
查看分区p1的文件路径。

```
SELECT pg_partition_filepath(49294);  
pg_partition_filepath  
-----  
base/16521/49294  
(1 row)
```

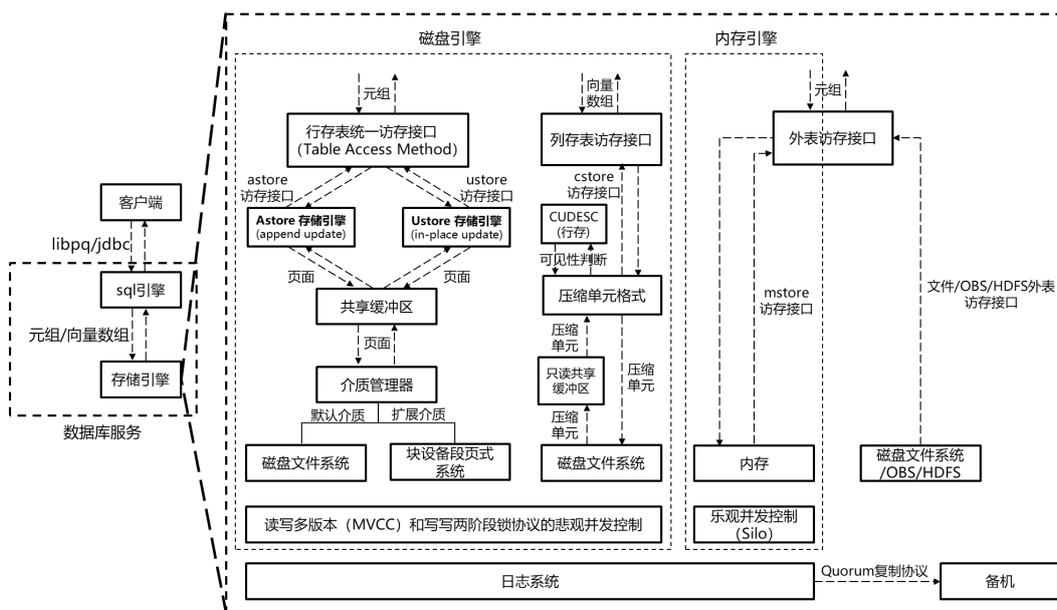
# 4 存储引擎

## 4.1 存储引擎体系架构

### 4.1.1 存储引擎体系架构概述

#### 4.1.1.1 静态编译架构

从整个数据库服务的组成构架来看，存储引擎向上对接SQL引擎，为SQL引擎提供或接收标准化的数据格式（元组或向量数组）；存储引擎向下对接存储介质，按照特定的数据组织方式，以页面、压缩单元（Compress Unit）或其他形式为单位，通过存储介质提供的特定接口，对存储介质中的数据完成读、写操作。GaussDB Kernel通过静态编译使数据库专业人员可以为特定的应用程序需求选择专用的存储引擎。为了减少对执行引擎的干扰，提供行存访问接口层TableAM，用来屏蔽底层行存引擎带来的差异，使得不同行存引擎可以分别独立演进。如下图所示。



在此基础之上，存储引擎通过日志系统提供数据的持久化和可靠性能力。通过并发控制（事务）系统保证同时执行的、多个读写操作之间的原子性、一致性和隔离性，通

过索引系统提供对特定数据的加速寻址和查询能力，通过主备复制系统提供整个数据库服务的高可用能力。

行存引擎主要面向OLTP ( OnLine Transaction Processing ) 类业务应用场景，适合高并发、小数据量的单点或小范围数据读写操作。行存引擎向上为SQL引擎提供元组形式的读写接口，向下以页面为单位通过可扩展的介质管理器对存储介质进行读写操作，并通过页面粒度的共享缓冲区来优化读写操作的效率。对于读写并发操作，采用多版本并发控制 ( MVCC, Multi-Version Concurrency Control )；对于写写并发操作，采用基于两阶段锁协议 ( 2PL, Two-Phase Locking ) 的悲观并发控制 ( PCC, Pessimistic Concurrency Control )。当前，行存引擎默认的介质管理器采用磁盘文件系统接口，后续可扩展支持块设备等其他类型的存储介质。GaussDB Kernel行存引擎可以选择基于Append update 的Astore或基于In-place update的Ustore。

#### 4.1.1.2 通用数据库服务层

从技术角度来看，存储引擎需要一些基础架构组件，主要包括：

**并发：**不同存储引擎选择正确的锁可以减少开销，从而提高整体性能。此外提供多版本并发控制或“快照”读取等功能。

**事务：**均需满足ACID的要求，提供事务状态查询等功能。

**内存缓存：**不同存储引擎在访问索引和数据时一般会对其进行缓存。缓存池允许直接从内存中处理经常使用的数据，从而加快了处理速度。

**检查点：**不同存储引擎一般都支持增量checkpoint/double write或全量checkpoint/full page write模式。应用可以根据不同条件进行选择增量或者全量，这个对存储引擎是透明的。

**日志：**GaussDB Kernel采用的是物理日志，其写入/传输/回放对存储引擎透明。

#### 4.1.2 设置存储引擎

存储引擎会对数据库整体效率和性能具有巨大影响，请根据实际需求选择适当的存储引擎。用户可使用WITH ( [ORIENTATION | STORAGE\_TYPE] [= value] [, ... ] )为表或索引指定一个可选的存储参数。参数的详细描述如下所示：

| ORIENTATION             | STORAGE_TYPE           |
|-------------------------|------------------------|
| ROW ( 缺省值 )：表的数据将以行式存储。 | [USTORE(缺省值) ASTORE 空] |

如果ORIENTATION指定为ROW，且STORAGE\_TYPE为空的情况下创建出的表类型取决于GUC参数enable\_default\_ustore\_table ( 取值为on/off，默认情况为on )：如果参数设置为on，创建出的表为Ustore类型；如果为off，创建出的表为Astore类型。

具体示例如下：

```
gaussdb=# CREATE TABLE TEST(a int);
gaussdb=# \d+ test
          Table "public.test"
  Column | Type   | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a      | integer |          | plain   |              |
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE
```

```
gaussdb=# CREATE TABLE TEST1(a int) with(orientation=row, storage_type=ustore);
gaussdb=# \d+ test1
Table "public.test1"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
a      | integer |          | plain   |              |
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no

gaussdb=# CREATE TABLE TEST2(a int) with(orientation=row, storage_type=astore);
gaussdb=# \d+ test2
Table "public.test2"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
a      | integer |          | plain   |              |
Has OIDs: no
Options: orientation=row, storage_type=astore, compression=no

gaussdb=# create table test4(a int) with(orientation=row);
gaussdb=# \d+
List of relations
Schema | Name | Type | Owner | Size | Storage | Description
-----+-----+-----+-----+-----+-----+-----
public | test4 | table | l30048445 | 0 bytes | {orientation=row,compression=no,storage_type=USTORE} |
(1 row)

gaussdb=# show enable_default_ustore_table;
enable_default_ustore_table
-----
on
(1 row)
```

## 4.1.3 存储引擎更新说明

### 4.1.3.1 GaussDB Kernel 503 版本

- Ustore适配分布式/并行查询/Global Temp Table/Vacuum full/列约束DEFERRABLE以及INITIALLY DEFERRED。
- Ustore增加在线重建索引。
- Ustore增加增强版本B-tree空页面估算，提升优化器代价估算准确度。
- Ustore增加存储引擎可靠性验证框架，Dignose Page/Page Verify。
- Ustore增强存储引擎相关的解析/检测/修复视图。
- Ustore增强基于WAL日志的定位能力，新增gs\_redo\_upage系统视图，支持对单页面的不断重放，获取并打印该页面的任何一个历史版本，加速页面损坏类问题的定位。
- Ustore扩展事务槽TD物理格式，为事务内空间复用做好铺垫。
- Ustore增加在线创建索引。
- Ustore适配闪回功能 ( for Ustore ) /极致RTO。

### 4.1.3.2 GaussDB Kernel R2 版本

- Ustore增加新的基于原位更新的行存储引擎Ustore，首次实现新、旧版本的记录的分离存储。
- Ustore增加回滚段模块。

- Ustore增加回滚过程，支持同步/异步/页内模式。
- Ustore增加支持事务的增强版本B-tree。
- Astore增加闪回功能，支持闪回表/闪回查询/闪回Drop/闪回Truncate。
- Ustore不支持的特性包括：分布式/并行查询/Table Sampling/Global Temp Table/在线创建/重建索引/极致RTO/Vacuum Full/列约束DEFERRABLE以及INITIALLY DEFERRED。

## 4.2 Astore 存储引擎

### 4.2.1 Astore 简介

Astore与Ustore的多版本实现最大的区别在于最新版本和历史版本是否分离存储。Astore不进行分离存储，而Ustore当前也只是分离了数据，索引本身没有分开。

#### 使用 Astore 的优势

1. Astore没有回滚段，而Ustore有回滚段。对于Ustore来说，回滚段是非常重要的，回滚段损坏，会导致数据丢失甚至数据库无法启动的严重问题；且Ustore恢复时同步需要Redo和Undo。由于Astore没有回滚段，旧数据都是记录在原先的文件中，所以当数据库异常crash后，恢复时，不会像Ustore数据库那样进行那么复杂的恢复。
2. 由于旧的数据是直接记录在数据文件中，而不是回滚段中，所以不会经常报Snapshot Too Old错误。
3. 回滚可以很快完成，因为回滚并不删除数据，但回滚时很复杂，在事务回滚时必须清理该事务所进行的修改，插入的记录要删除，更新的记录要更新回来，同时回滚的过程也会再次产生大量的Redo日志。
4. WAL日志要简单一些，仅需要记录数据文件的变化，不需要记录回滚段的变化。

## 4.3 Ustore 存储引擎

### 4.3.1 Ustore 简介

Ustore ( Unified Storage ) 是GaussDB Kernel推出的一款原位更新的存储引擎，其多版本的实现较Astore最大的区别在于最新版本和历史版本的数据是分离存储的，而索引当前还没有分离。Ustore目前已发展为GaussDB Kernel集中式形态的默认行存引擎。

#### 使用 Ustore 的优势

1. 最新版本和历史版本分离存储，相比Astore扫描范围小。去除Astore的HOT chain，非索引列/索引列更新，Heap均可原位更新，ROWID可保持不变。历史版本可批量回收，对最新版本空间膨胀友好。
2. 大并发更新同一行的场景，Ustore的原位更新机制保证了元组ROWID稳定，先到先得，更新时延相对稳定。
3. 不依赖Vacuum进行旧版本清理。Index与Heap解耦，可独立清理，IO平稳度较好。

#### 4. 支持闪回功能。

不过，Ustore DML除修改数据页面，同时也需要修改Undo，更新操作开销会稍大一些。此外单条Tuple扫描开销由于需要复制（Astore返回指针）也会大一些。

### 4.3.1.1 Ustore 特性与规格

#### 4.3.1.1.1 特性约束

| 类别   | 特性                      | 是否支持 |
|------|-------------------------|------|
| 事务   | Serializable            | ×    |
|      | 在事务块中对分区表执行DDL操作        | ×    |
| 可扩展性 | Hashbucket              | ×    |
| SQL  | Table sampling/物化视图/键值锁 | ×    |

#### 4.3.1.1.2 存储规格

1. 数据表最大列数不能超过1600列。
2. Ustore表（不含toast情况）最大Tuple长度不能超过（ $8192 - \text{MAXALIGN}(56 + \text{init\_td} * 26 + 4)$ ），其中MAXALIGN表示8字节对齐。当插入数据长度超过阈值时，用户会收到元组长度过长无法插入的报错。其中init\_td对于Tuple长度的影响如下：
  - 表init\_td数量为最小值2时，Tuple长度不能超过 $8192 - \text{MAXALIGN}(56+2*26+4) = 8080\text{B}$ 。
  - 表init\_td数量为默认值4时，Tuple长度不能超过 $8192 - \text{MAXALIGN}(56+4*26+4) = 8024\text{B}$ 。
  - 表init\_td数量为最大值128时，Tuple长度不能超过 $8192 - \text{MAXALIGN}(56+128*26+4) = 4800\text{B}$ 。
3. init\_td取值范围[2, 128]，默认值4。单页面支持的最大并发不超过128个。
4. 索引最大列数不能超过32列。全局分区索引最大列数不能超过31列。
5. 索引元组长度不能超过 $(8192 - \text{MAXALIGN}(28 + 3 * 4 + 3 * 10) - \text{MAXALIGN}(42))/3$ ，其中MAXALIGN表示8字节对齐。当插入数据长度超过阈值时，用户会收到索引元组长度过长无法插入的报错，其中索引页头为28B，行指针为4B，元组CTID+INFO标记位为10B，页尾为42B。
6. 回滚段容量最大支持16TB。

### 4.3.1.2 使用 Ustore 进行测试

#### 创建Ustore表

使用CREATE TABLE语句创建Ustore表。

```
gaussdb=# CREATE TABLE ustore_table(a INT PRIMARY KEY, b CHAR (20)) WITH (STORAGE_TYPE=USTORE);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "ustore_table_pkey" for table "ustore_table"
CREATE TABLE
gaussdb=# \d+ ustore_table
Table "public.ustore_table"
```

```

Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
a | integer | not null | plain | | 
b | character(20) | | extended | | 
Indexes:
"ustore_table_pkey" PRIMARY KEY, ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no
    
```

### 为Ustore表创建索引

Ustore当前仅支持B-tree类型的多版本索引，在一些场景中，为了区别于Astore的B-tree索引，也会将Ustore表的多版本B-tree索引称为UB-tree（Ustore B-tree，UB-tree介绍详见[Index](#)章节）。用户可以参照以下方式使用CREATE INDEX语句为Ustore表的“a”属性创建一个UB-tree索引。

Ustore表不指定创建索引类型，默认创建的是UB-tree索引：

```

gaussdb=# CREATE INDEX UB-tree_index ON ustore_table(a);
CREATE INDEX
gaussdb=# \d+ ustore_table
Table "public.ustore_table"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
a | integer | not null | plain | | 
b | character(20) | | extended | | 
Indexes:
"ustore_table_pkey" PRIMARY KEY, ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
"ubtree_index" ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no
    
```

## 4.3.1.3 Ustore 的最佳实践

### 4.3.1.3.1 怎么配置 init\_td 大小

TD（Transaction Directory，事务目录）是Ustore表独有的用于存储页面事务信息的结构，TD的数量决定该页面支持的最大并发数。在创建表或索引时可以指定初始的TD大小init\_td，默认值为4，即同时支持4个并发事务修改该页面，最大值为128。

用户需要结合业务并发度分析是否需要手动配置init\_td。另外也可以结合业务运行过程中“wait available td”等待事件出现的频率来分析是否需要调整，一般“wait available td”等于0，如果“wait available td”一直不为0，就存在等待TD的事件，此时建议增大init\_td再进行观察，反复几次，如果大于0的情况属于偶发，不建议调整，多余的TD槽位会占用更多的空间。推荐的增大的方法可以按照倍数进行测试，建议可从小到大尝试8、16、32、48、...、128，并观测对应的等待事件是否有明显减少，尽量取等待事件较少中init\_td数量最小的值作为默认值以节省空间。init\_td的配置和修改方法参见《开发指南》的“SQL参考 > SQL语法 > CREATE TABLE”章节。

### 4.3.1.3.2 怎么配置 fillfactor 大小

fillfactor是用于描述页面填充率的参数，该参数与页面能存放的元组数量、大小以及表的物理空间直接相关。Ustore表的默认页面填充率为92%，预留的8%空间用于更新的扩展，也可以用于TD列表的扩展空间。fillfactor的配置和修改方法参见《开发指南》的“SQL参考 > SQL语法 > CREATE TABLE”章节。

用户需要结合业务分析是否需要手动配置fillfactor。如果表数据导入后只有查询或定长更新操作，可将页面填充率调整为100%。如果数据导入后存在大量定长更新操作，建议为不调整页面填充率或者将页面填充率调整的更小，以减少跨页更新带来的性能损耗。

### 4.3.1.3.3 统计信息收集

Ustore的无效元组清理依赖于统计信息的准确性，关闭参数track\_counts以及track\_activities会造成空间膨胀，默认开启，请保持开启。性能场景除外。

打开:

```
gs_guc reload -Z datanode -N all -I all -c "track_counts=on;"  
gs_guc reload -Z datanode -N all -I all -c "track_activities=on;"
```

关闭:

```
gs_guc reload -Z datanode -N all -I all -c "track_counts=off;"  
gs_guc reload -Z datanode -N all -I all -c "track_activities=off;"
```

### 4.3.1.3.4 在线校验功能

在线校验是Ustore独创的运行过程中可以有效预防页面因编码逻辑错误导致的逻辑损坏，默认开启，业务现网请保持开启。性能场景除外。

关闭:

```
gs_guc reload -Z datanode -N all -I all -c "ustore_attr=";
```

打开:

```
gs_guc reload -Z datanode -N all -I all -c  
"ustore_attr="ustore_verify_level=fast;ustore_verify_module=upage:ubtree:undo"
```

### 4.3.1.3.5 怎么配置回滚段大小

一般情况下回滚段大小的参数使用默认值即可。为了达到最佳性能，部分场景下可调整回滚段大小的相关参数，具体场景与设置方法如下。

#### 1. 保留给定时间内的历史版本数据。

当使用闪回或者支撑问题定位时，通常希望保留更多历史版本数据，此时需要修改undo\_retention\_time。undo\_retention\_time默认值是0，取值范围为0~3天。

调整的推荐值为900s，需要注意的是，undo\_retention\_time的取值越大，对业务的影响除了Undo空间占用增多，也会造成数据空间膨胀，进一步影响数据扫描更新性能。当不使用闪回或者希望减少历史旧版本的磁盘空间占用时，需要将undo\_retention\_time调小来达到最佳性能。可以通过如下方法选择更适合自己业务模型的取值。

查询guc参数undo\_space\_limit\_size，查询视图gs\_stat\_undo，获取近期undo空间平均增长速度avg\_space\_increase\_speed与当前undo占用空间curr\_used\_undo\_size，计算undo\_retention\_time的建议值 $new\_val = 0.5 * (undo\_space\_limit\_size * 0.8 - curr\_used\_undo\_size) / avg\_space\_increase\_speed$ 。

#### 2. 保留给定空间大小的历史版本数据。

如果业务中存在长事务或大事务可能导致Undo空间膨胀时，需要将undo\_space\_limit\_size调大，undo\_space\_limit\_size默认值为256GB，取值范围为800MB~16TB。

在磁盘空间允许的条件下，推荐undo\_space\_limit\_size设置翻倍。同时undo\_space\_limit\_size的取值越大则占用磁盘空间越大，可能降低性能。如果查询gs\_stat\_undo()的curr\_used\_undo\_size发现不存在Undo空间膨胀，可以恢复为原值。

调整undo\_space\_limit\_size后可相应提高单事务平均占用undo空间undo\_limit\_size\_per\_transaction的取值，undo\_limit\_size\_per\_transaction取值

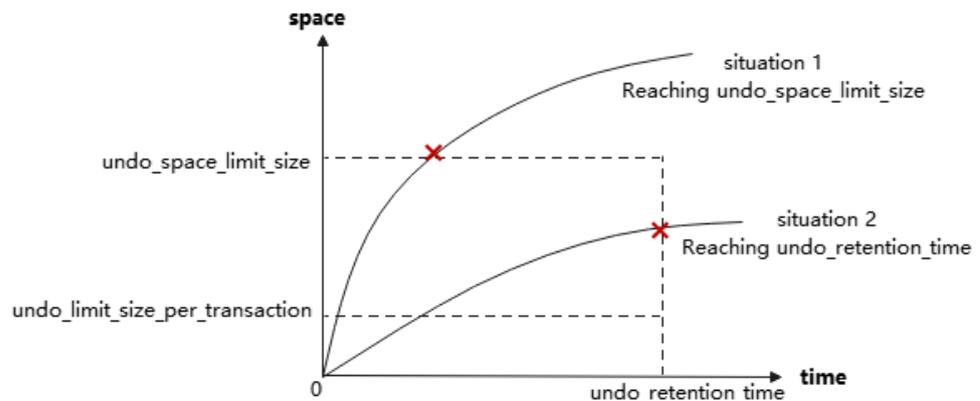
范围为2MB~16TB，默认值为32GB。设置时建议 `undo_limit_size_per_transaction` 不超过 `undo_space_limit_size`，即单事务Undo分配空间阈值不大于Undo总空间阈值。

为了更准确设置该参数来达到最佳性能，建议采用如下方式进行计算。

- `undo_space_limit_size`: 查询视图 `gs_stat_undo`，获取近期undo空间平均增长速度 `avg_space_increase_speed` 和 `curr_used_undo_size`，计算 `undo_space_limit_size` 的建议值  $new\_val = 86400 * 30 * avg\_space\_increase\_speed + curr\_used\_undo\_size$ 。
- `undo_limit_size_per_transaction`: 查询 `gs_stat_undo()`，获取单事务最大占用undo空间 `max_xact_space` (503.2版本中扩展该列)，建议该参数调整后不小于  $new\_val = 10 * max\_xact\_space$ 。

### 3. 历史版本的保留参数的调整优先级。

在 `undo_retention_time`、`undo_space_limit_size`、`undo_limit_size_per_transaction` 中，先触发的空间阈值会先进行约束限制。

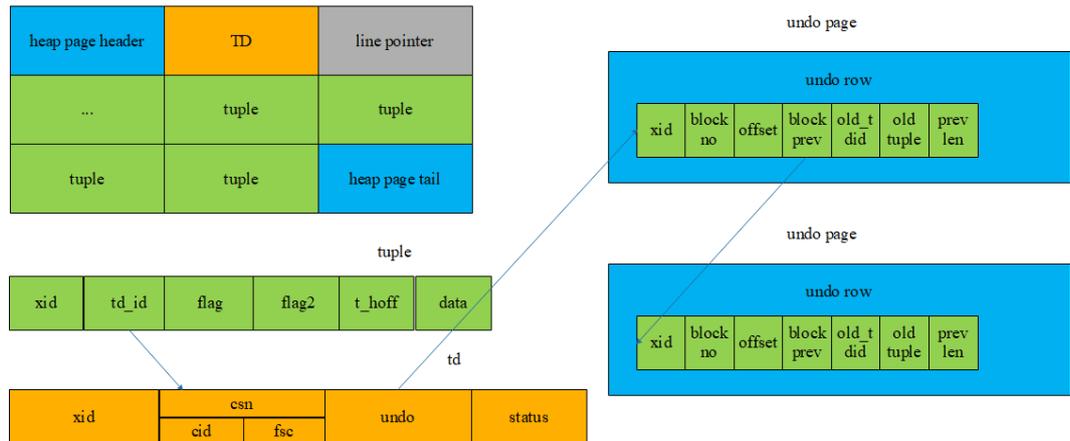


例如：Undo强制回收阈值参数 `undo_space_limit_size` 设置为1GB，Undo旧版本保留时间 `undo_retention_time` 为900s，如果900s内产生的历史版本数据不足  $1GB * 0.8$ ，则按照900s进行回收限制；否则按照  $1GB * 0.8$  进行回收限制。遇到该情况时，如果磁盘空闲空间充足，则上调 `undo_space_limit_size`，如果磁盘空闲空间紧缺，则下调 `undo_retention_time`。

## 4.3.2 存储格式

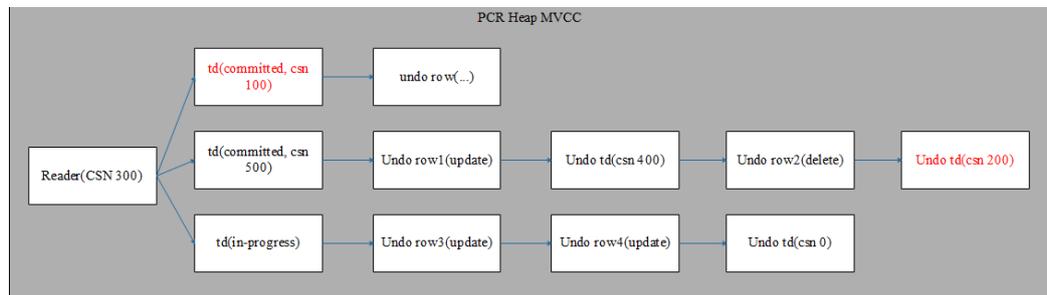
### 4.3.2.1 Relation

### 4.3.2.1.1 PbRCR(Page base Row Consistency Read) Heap 多版本管理



1. Heap的多版本管理是基于Tuple的行级多版本管理。
2. 事务修改记录时，会将历史数据记录到Undo Row中。
3. 在Tuple中的td\_id上记录产生的Undo Row地址(zone\_id, block no, page offset)。
4. 将新的数据覆盖写入Heap页面。
5. 每次对数据的修改都会产生Undo，同一记录的undo通过block prev串联。

### 4.3.2.1.2 PbPCR Heap 可见性机制



1. 当前仅支持行一致性读，后续支持CR页面构建以及页面一致性读，大大提升seqscan的效率。
2. 数据删除事务提交即可复用空间，无需等待oldestxmin的推进，空间利用率更高，老快照可通过Undo记录获取历史版本。

### 4.3.2.1.3 Heap 空间管理

Ustore使用Free Space Map (FSM) 文件记录了每个数据页的空闲空间，并且以树的结构组织起来。每当用户想要对某个表执行插入操作或者是非原位更新操作时，就会从该表对应的FSM中进行快速查找，查看当前FSM上记录的最大空闲空间是否可以满足插入所需的空间要求，如果满足则返回对应的blocknum用于执行插入操作，否则执行拓展页面逻辑。

每一个表或者分区对应的FSM结构存放在一个独立的FSM文件中，该FSM文件与表数据放在相同的目录下。例如，假设表t1对应的数据文件为32181，则其对应的FSM文件为32181\_fsm。FSM内部同样是以数据块的格式存储，这里称为FSM block，FSM block之间的逻辑结构组成了一棵有三层节点的树，树的节点在逻辑上是顶堆关系。每次在FSM上查找时从根节点进行，一直查找找到叶子节点，然后在叶子节点内搜索到

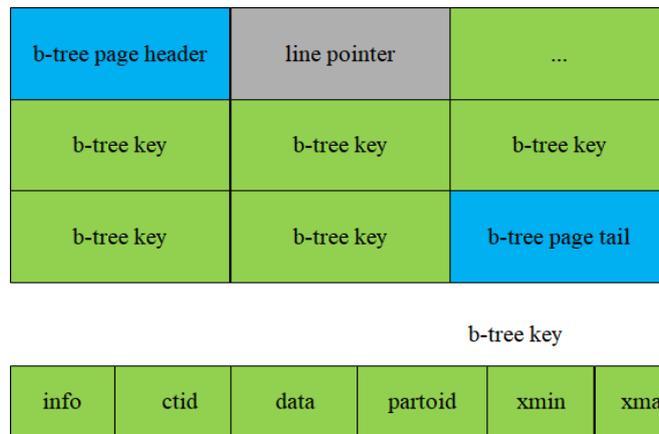
一个可用的页面并返回给业务用于执行后续操作。该结构不保证和数据页实际可用空间保持实时一致，会在DML的执行过程中进行维护。Ustore会在Auto Vacuum的过程中概率性对该FSM进行修复重建。

### 4.3.2.2 Index

UB-tree主要增强如下：

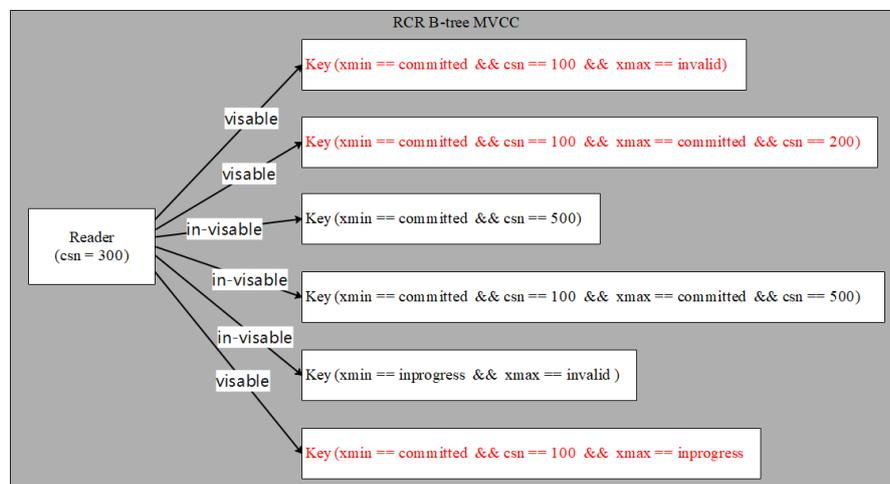
1. 增加了MVCC能力。
2. 增加了独立空页回收能力。

#### 4.3.2.2.1 RCR(Row Consistency Read) UB-tree 多版本管理



1. UB-tree的多版本管理采用基于Key的多版本管理，最新版本和历史版本均在UB-tree上。
2. 为了节省空间，xmin/xmax采用xid-base + delta的方式表示，64位的xid-base储存在页面上，元组上储存32位的delta。页面上xid-base也需要通过额外的逻辑进行维护。
3. UB-tree插入或者删除key时按照key + TID的顺序排列，索引列相同的元组按照对应元组的TID作为第二关键字进行排序。会将xmin、xmax追加到key的后面。
4. 索引分裂时，多版本信息随着key的迁移而迁移。

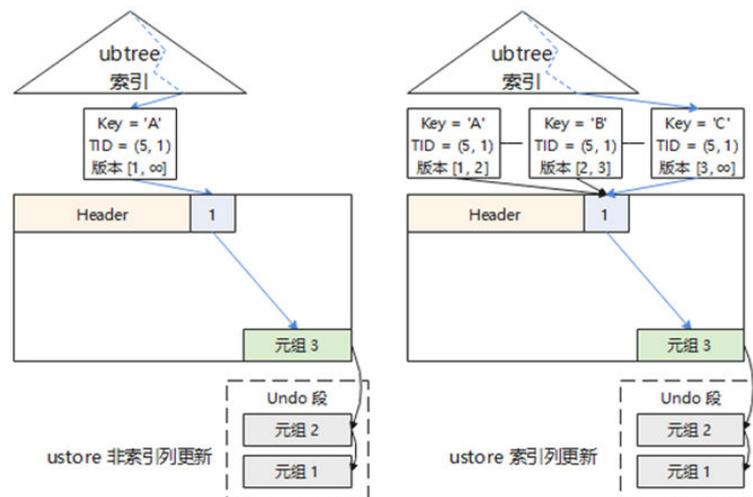
#### 4.3.2.2.2 RCR UB-tree 可见性机制



1. 支持索引数据的多版本管理及可见性检查，能够自主鉴别旧版本元组并进行回收，同时索引层的可见性检查使得索引扫描（Index Scan）及仅索引扫描（IndexOnly Scan）的概率大大提升。
2. 在索引插入操作之外，增加了索引删除操作，用于对被删除或修改的元组对应的索引元组进行标记。

#### 4.3.2.2.3 UB-tree 增删改查

- **Insert操作**：UB-tree的插入逻辑基本不变，只需增加索引插入时直接获取事务信息填写xmin字段。
- **Delete操作**：UB-tree额外增加了索引删除流程，索引删除主要步骤与插入相似，获取事务信息填写xmax字段（B-tree索引不维护版本信息，不需要删除操作），同时更新页面上的active\_tuple\_count，若active\_tuple\_count被减为0，则尝试页面回收。
- **Update操作**：对于Ustore而言，数据更新对UB-tree索引列的操作也与Astore有所不同，数据更新包含两种情况：索引列和非索引列更新，下图给出了UB-tree在数据发生更新时的处理。

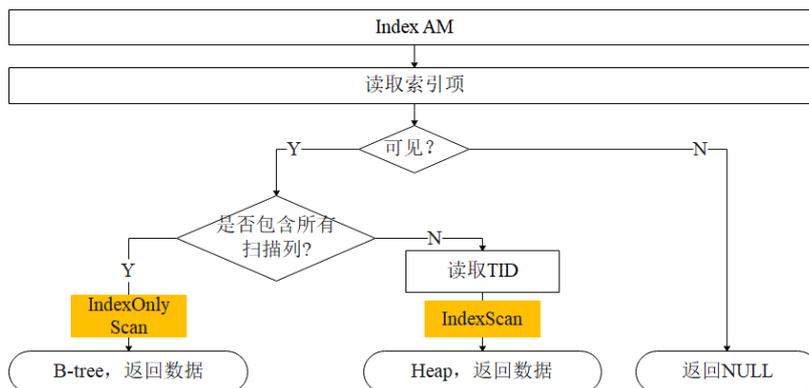


上图展示UB-tree在索引列和非索引列更新的差异：

- a. 在非索引列更新的情况下，索引不发生任何变化，index tuple仍指向第一次插入的data tuple，Uheap不会插入新的data tuple，而是修改当下data tuple并将历史数据存入Undo中。
  - b. 在索引列更新的情况下，UB-tree也会插入新的index tuple，但是会指向同一个data linepointer和同一个data tuple，扫描旧版本的数据则需要从Undo中读取。
- **Scan操作**：用户在读取数据时，可通过使用索引扫描加速，UB-tree支持索引数据的多版本管理及可见性检查，索引层的可见性检查使得索引扫描（Index Scan）及仅索引扫描（IndexOnly Scan）性能有所提升。

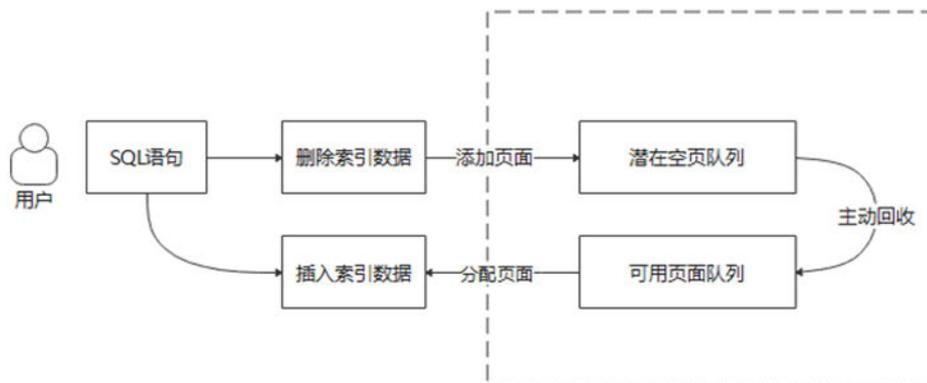
对于索引扫描：

- a. 若索引列包含所有扫描列（IndexOnly Scan），则通过扫描条件在索引上进行二分查找，找到符合条件元组即可返回数据。
- b. 若索引列不包含所有扫描列（Index Scan），则通过扫描条件在索引上进行二分查找，找到符合条件元组的TID，再通过TID到数据表上查找对应的数据元组。如下图所示。



#### 4.3.2.2.4 UB-tree 空间管理

当前Astore的索引依赖AutoVacuum和Free Space Map (FSM) 进行空间管理, 存在回收不及时的问题, 而Ustore的索引使用其特有的URQ (UB-tree Recycle Queue, 一种基于循环队列的数据结构, 即双循环队列), 对索引空闲空间进行管理。双循环队列是指有两个循环队列, 一个潜在空页队列, 另一个可用空页队列, 在DML过程中完成索引的空间管理, 能有效地缓解DML过程中造成的空间急剧膨胀问题。索引回收队列单独储存在B-tree索引对应的FSM文件中。



如上图所示, 索引页面在双循环队列间流动如下:

##### 1. 索引空页 > 潜在队列

索引页尾字段中记录了页面上活跃元组个数 (activeTupleCount), 在DML过程中, 删空一个页面的所有元组, 即activeTupleCount为零时会将索引页放入潜在队列中。

##### 2. 潜在队列 > 可用队列

潜在队列到可用队列的转化主要是达到一个潜在队列收支平衡以及可用队列在拿页时有页可拿的目的, 即当从可用队列拿出一个索引空页用完后, 最好能够从潜在队列转化至少一个索引页面到可用队列中, 以及每当潜在队列新加入一个索引页面时, 能从潜在队列中移除至少一个索引页插入可用队列中, 达到潜在队列的收支平衡, 以及可用队列有页可用的目的。

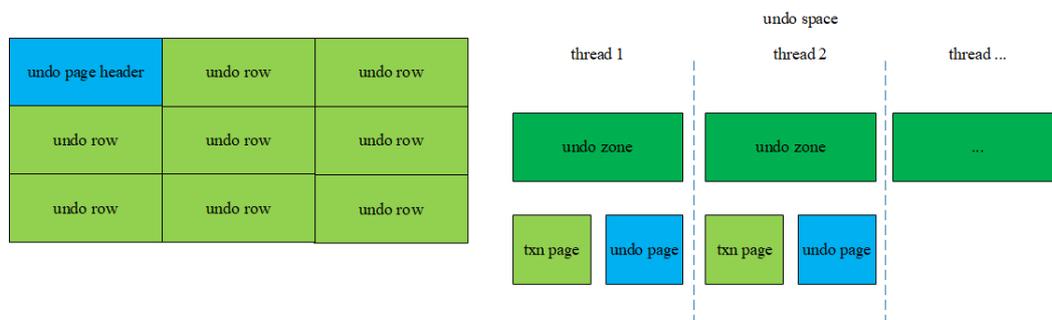
##### 3. 可用队列 > 索引空页

索引在分裂等获取一个索引空页面时, 会先从可用队列中进行查找是否有可以复用的索引页, 如果找到则直接进行复用, 没有可复用页面则进行物理扩页。

### 4.3.2.3 Undo

历史版本数据集中存放在\$GAUSS\_HOME/undo目录中，回滚段日志是与单个写事务关联的所有撤销日志的集合。支持permanent/unlogged/temp三种表类型。

#### 4.3.2.3.1 回滚段管理



1. 每个undo zone除了管理部分txn page外，还管理undo page。
2. Undo页面中存储undo row，对数据的修改会将历史版本记录到Undo中。
3. Undo记录也是数据，因此对Undo页面的修改同样会记录Redo。

#### 4.3.2.3.2 文件组织结构

txn page所在文件组织结构:

```
$GAUSS_HOME/undo/{permanent|unlogged|temp}/$undo_zone_id.meta.$segno
```

undo row所在文件组织结构:

```
$GAUSS_HOME/undo/{permanent|unlogged|temp}/$undo_zone_id.$segno
```

#### 4.3.2.3.3 Undo 空间管理

Undo子系统依赖后台回收线程进行空闲空间回收，负责主机上Undo模块的空间回收，备机通过回放Xlog进行回收。回收线程遍历使用中的undo zone，对该zone中的txn page扫描，依据xid从小到大的顺序进行遍历。回收已提交或者已回滚完成的事务，且该事务的提交时间应早于\$(current\_time-undo\_retention\_time)。对于遍历过程中需要回滚的事务，后台回收线程会为该事务添加异步回滚任务。

当数据库中存在运行时间长、修改数据量大的事务，或者开启闪回时间较长的时候，可能出现undo空间持续膨胀的情况。当undo占用空间接近undo\_space\_limit\_size时，就会触发强制回收。只要事务已提交或者已回滚完成，即使事务提交时间晚于\$(current\_time-undo\_retention\_time)，在这种情况下也可能被回收掉。

### 4.3.3 Ustore 事务模型

GaussDB Kernel事务基础:

1. 事务启动时不会自动分配XID，该事务中的第一条DML/DDI语句运行时才会真正为该事务分配XID。
2. 事务结束时，会产生代表事务提交状态的CLOG (Commit Log)，CLOG共有四种状态：事务运行中、事务提交、事务同步回滚、子事务提交。每个事务的CLOG状态位为2 bits，CLOG页面上每个字节可以表示四个事务的提交状态。
3. 事务结束时，还会产生代表事务提交顺序的CSN (Commit sequence number)，CSN为实例级变量，每个XID都有自己对应的唯一CSN。CSN可以标

记事务的以下状态：事务运行中、事务提交、事务同步回滚、事务正在提交、本事务为子事务、事务已冻结。

### 4.3.3.1 事务提交

1. 隐式事务。单条DML/DDI语句自动触发隐式事务，这种事务没有显式的事务块控制语句（START TRANSACTION/BEGIN/COMMIT/END），DML语句结束后自动提交。
2. 显式事务。显式事务由显式的START TRANSACTION/BEGIN语句控制事务的开始，由COMMIT/END语句控制事务的提交。

子事务必须存在于显式事务或存储过程中，由SAVEPOINT语句控制子事务开始，由RELEASE SAVEPOINT语句控制子事务结束。如果一个事务在提交时还存在未释放的子事务，该事务提交前会先执行子事务的提交，所有子事务提交完毕后才进行父事务的提交。

Ustore支持读已提交隔离级别。语句在执行开始时，获取当前系统的CSN作为当前语句的查询CSN。整个语句的可见结果由语句开始那一刻决定，不受后续其他事务修改影响。Ustore中read committed默认是保持一致性读的。Ustore也支持标准的2PC事务。

### 4.3.3.2 事务回滚

回滚是在事务运行的过程中发生了故障等异常情形下，事务不能继续执行，系统需要将事务中已完成的修改操作进行撤销。Astore、UB-tree没有回滚段，自然没有这个专门的回滚动作。Ustore为了性能考虑，它的回滚流程结合了同步、异步与页内即时回滚3种形式。

1. **同步回滚。**  
有三种情况会触发事务的同步回滚：
  - a. 事务块中的ROLLBACK关键字会触发同步回滚。
  - b. 事务运行过程中如果发生ERROR级别报错，此时的COMMIT关键字与ROLLBACK功能相同，也会触发同步回滚。
  - c. 事务运行过程中如果发生FATAL/PANIC级别报错，在线程退出前会尝试将该线程绑定的事务进行一次同步回滚。
2. **异步回滚。**同步回滚失败或者在系统宕机后再次重启时，会由Undo回收线程为未回滚完成的事务发起异步回滚任务，立即对外提供服务。由异步回滚任务发起线程undo launch负责拉起异步回滚工作线程undo worker，再由异步回滚工作线程实际执行回滚任务。undo launch线程最多可以同时拉起5个undo worker线程。
3. **页面级回滚。**当事务需要回滚但还未回滚到本页面时，如果其他事务需要复用该事务所占用的TD，就会在复用前对该事务在本页面的所有修改执行页面级回滚。页面级回滚只负责回滚事务在本页面的修改，不涉及其他页面。

Ustore子事务的回滚由ROLLBACK TO SAVEPOINT语句控制，子事务回滚后父事务可以继续运行，子事务的回滚不影响父事务的事务状态。如果一个事务在回滚时还存在未释放的子事务，该事务回滚前会先执行子事务的回滚，所有子事务回滚完毕后会进行父事务的回滚。

## 4.3.4 闪回恢复

闪回恢复功能是数据库恢复技术的一环，可以有选择性的撤销一个已提交事务的影响，将数据从人为不正确的操作中进行恢复。在采用闪回技术之前，只能通过备份恢复、PITR等手段找回已提交的数据库修改，恢复时长需要数分钟甚至数小时。采用闪

回技术后，通过闪回Drop和闪回Truncate恢复已提交的数据库Drop/Truncate的数据，只需要秒级，而且恢复时间和数据库大小无关。

#### 说明

- ASTORE引擎暂不支持闪回功能。
- 备机不支持闪回操作。
- 用户可以根据需要开启闪回功能，开启后会带来一定的性能劣化。

### 4.3.4.1 闪回查询

#### 背景信息

闪回查询可以查询过去某个时间点表的某个snapshot数据，这一特性可用于查看和逻辑重建意外删除或更改的受损数据。闪回查询基于MVCC多版本机制，通过检索查询旧版本，获取指定老版本数据。

#### 前提条件

undo\_retention\_time参数用于设置undo旧版本的保留时间。

#### 语法

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias [ , ... ] ) ] ]  
[ TABLESAMPLE sampling_method ( argument [ , ... ] ) [ REPEATABLE ( seed ) ] ]  
[ TIMECAPSULE { TIMESTAMP | CSN } expression ]  
{ ( 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 [ , ... ] ) ] }
```

语法树中“TIMECAPSULE {TIMESTAMP | CSN} expression”为闪回功能新增表达方式，其中TIMECAPSULE表示使用闪回功能，TIMESTAMP以及CSN表示闪回功能使用具体时间点信息或使用CSN（commit sequence number）信息。

#### 参数说明

- **TIMESTAMP**
  - 指要查询某个表在TIMESTAMP这个时间点上的数据，TIMESTAMP指一个具体的历史时间。
- **CSN**
  - 指要查询整个数据库逻辑提交序下某个CSN点的数据，CSN指一个具体逻辑提交时间点，数据库中的CSN为写一致性点，每个CSN代表整个数据库的一个一致性点，查询某个CSN下的数据代表SQL查询数据库在该一致性点的相关数据。

备注：使用时间点进行闪回时，可能会有3s的误差。想要闪回到精确的操作点，需要使用CSN进行闪回。GTM-Free模式下没有全局一致性csn点，暂时不支持以csn的方式进行闪回。

#### 使用示例

- 示例：

```
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
NOTICE: table "flashtest" does not exist, skipping
DROP TABLE
--创建表flashtest
gaussdb=# CREATE TABLE "public".flashtest (col1 INT,col2 TEXT) with(storage_type=ustore);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'col1' as the distribution column by
default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
--查询csn
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
 int8in
-----
 79351682
 79351682
 79351682
 79351682
 79351682
 79351682
(6 rows)
--查询当前时间戳
gaussdb=# select now();
      now
-----
2023-09-13 19:35:26.011986+08
(1 row)
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
   3 | INSERT3
   1 | INSERT1
   2 | INSERT2
   4 | INSERT4
   5 | INSERT5
   6 | INSERT6
(6 rows)
--闪回查询某个csn处的表
gaussdb=# SELECT * FROM flashtest TIMECAPSULE CSN 79351682;
 col1 | col2
-----+-----
(0 rows)
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
   1 | INSERT1
   2 | INSERT2
   4 | INSERT4
   5 | INSERT5
   3 | INSERT3
   6 | INSERT6
(6 rows)
--闪回查询某个时间戳处的表
gaussdb=# SELECT * FROM flashtest TIMECAPSULE TIMESTAMP '2023-09-13 19:35:26.011986';
 col1 | col2
-----+-----
(0 rows)
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
   1 | INSERT1
   2 | INSERT2
   4 | INSERT4
   5 | INSERT5
   3 | INSERT3
   6 | INSERT6
```

```
(6 rows)
--闪回查询某个时间戳处的表
gaussdb=# SELECT * FROM flashtest TIMECAPSULE TIMESTAMP to_timestamp ('2023-09-13
19:35:26.011986', 'YYYY-MM-DD HH24:MI:SS.FF');
 col1 | col2
-----+-----
(0 rows)
--闪回查询某个csn处的表，并对表进行重命名
gaussdb=# SELECT * FROM flashtest AS ft TIMECAPSULE CSN 79351682;
 col1 | col2
-----+-----
(0 rows)
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

### 4.3.4.2 闪回表

#### 背景信息

闪回表可以将表恢复至特定时间点，当逻辑损坏仅限于一个或一组表，而不是整个数据库时，此特性可以快速恢复表的数据。闪回表基于MVCC多版本机制，通过删除指定时间点和该时间点之后的增量数据，并找回指定时间点和当前时间点删除的数据，实现表级数据还原。

#### 前提条件

`undo_retention_time`参数用于设置undo旧版本的保留时间。

#### 语法

```
TIMECAPSULE TABLE table_name TO { TIMESTAMP | CSN } expression
```

#### 使用示例

```
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
NOTICE: table "flashtest" does not exist, skipping
DROP TABLE
--创建表
gaussdb=# CREATE TABLE "public".flashtest (col1 INT,col2 TEXT) with(storage_type=ustore);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'col1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
--查询csn
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
 int8in
-----
79352065
79352065
79352065
79352065
79352065
79352065
(6 rows)
--查询当前的时间戳
gaussdb=# select now();
 now
-----
2023-09-13 19:46:34.102863+08
(1 row)
--查看表flashtest
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
(0 rows)
```

```
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
 3 | INSERT3
 1 | INSERT1
 2 | INSERT2
 4 | INSERT4
 5 | INSERT5
 6 | INSERT6
(6 rows)
--闪回表至特定csn
gaussdb=# TIMECAPSULE TABLE flashtest TO CSN 79352065;
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
(0 rows)
gaussdb=# select now();
      now
-----
2023-09-13 19:52:21.551028+08
(1 row)
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
 3 | INSERT3
 6 | INSERT6
 1 | INSERT1
 2 | INSERT2
 4 | INSERT4
 5 | INSERT5
(6 rows)
--闪回表至特定的时间戳
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP to_timestamp ('2023-09-13 19:52:21.551028',
'YYYY-MM-DD HH24:MI:SS.FF');
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
(0 rows)
gaussdb=# select now();
      now
-----
2023-09-13 19:54:00.641506+08
(1 row)
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
 3 | INSERT3
 6 | INSERT6
 1 | INSERT1
 2 | INSERT2
 4 | INSERT4
 5 | INSERT5
(6 rows)
--闪回表至特定的时间戳
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP '2023-09-13 19:54:00.641506';
```

```
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
(0 rows)
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

### 4.3.4.3 闪回 DROP/TRUNCATE

#### 背景信息

- 闪回DROP：可以恢复意外删除的表，从回收站（recyclebin）中恢复被删除的表及其附属结构如索引、表约束等。闪回drop是基于回收站机制，通过还原回收站中记录的表的物理文件，实现已drop表的恢复。
- 闪回TRUNCATE：可以恢复误操作或意外被进行truncate的表，从回收站中恢复被truncate的表及索引的物理数据。闪回truncate基于回收站机制，通过还原回收站中记录的表的物理文件，实现已truncate表的恢复。

#### 前提条件

- 开启enable\_recyclebin参数（GUC参数在postgresql.conf文件修改），启用回收站，请联系管理员修改。
- recyclebin\_retention\_time参数用于设置回收站对象保留时间，超过该时间的回收站对象将被自动清理，请联系管理员修改。

#### 相关语法

- 删除表  
DROP TABLE table\_name [PURGE]
- 清理回收站对象  
PURGE { TABLE { table\_name }  
| INDEX { index\_name }  
| RECYCLEBIN  
}
- 闪回被删除的表  
TIMECAPSULE TABLE { table\_name } TO BEFORE DROP [RENAME TO new\_tablename]
- 截断表  
TRUNCATE TABLE { table\_name } [ PURGE ]
- 闪回截断的表  
TIMECAPSULE TABLE { table\_name } TO BEFORE TRUNCATE

#### 参数说明

- DROP/TRUNCATE TABLE table\_name PURGE
  - 默认将表数据放入回收站中，PURGE直接清理。
- PURGE RECYCLEBIN
  - 表示清理回收站对象。
- TO BEFORE DROP  
使用这个子句检索回收站中已删除的表及其子对象。  
可以指定原始用户指定的表的名称，或对对象删除时数据库分配的系统生成名称。



```

gaussdb=# select * from flashtest;
 id | name
----+-----
  1 | A
(1 row)
--DROP表flashtest
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
--查看回收站，删除的表被放入回收站
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcybid | rcyrelid |      rcyname      | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangeecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid |
 rcyfrozenxid64
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
      18591 | 12737 | 18585 | BIN$31C14EB4899$9737$0==$0 | flashtest      | d          | 0 |
79352606 | 2023-09-13 20:01:28.640664+08 | 79352595 | 7935259
5 |      2200 | 10 | 0 | 18585 | t          | t          | 225492 | 225492
|      18591 | 12737 | 18590 | BIN$31C14EB489E$12D1B978==$0 | pg_toast_18585_index | d          | 3
|      79352606 | 2023-09-13 20:01:28.64093+08 | 79352595 | 7935259
5 |      99 | 10 | 0 | 18590 | f          | f          | 0 | 0
|      18591 | 12737 | 18588 | BIN$31C14EB489C$12D1BF60==$0 | pg_toast_18585 | d          | 2
|      79352606 | 2023-09-13 20:01:28.641018+08 | 0 |
0 |      99 | 10 | 0 | 18588 | f          | f          | 225492 | 225492
(3 rows)
--查看表flashtest，表不存在
gaussdb=# select * from flashtest;
ERROR: relation "flashtest" does not exist
LINE 1: select * from flashtest;
          ^
--PURGE表，将回收站中的表删除
gaussdb=# PURGE TABLE flashtest;
PURGE TABLE
--查看回收站，回收站中的表被删除
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcybid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychangeecs | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
(0 rows)

-- PURGE INDEX index_name; --
gaussdb=# drop table if EXISTS flashtest;
NOTICE: table "flashtest" does not exist, skipping
DROP TABLE
--创建表flashtest
gaussdb=# create table if not EXISTS flashtest(id int, name text) with (storage_type = ustore);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
--为表flashtest创建索引flashtest_index
gaussdb=# create index flashtest_index on flashtest(id);
CREATE INDEX
--查看flashtest表的基本信息
gaussdb=# \d+ flashtest
          Table "public.flashtest"
  Column | Type  | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----
 id      | integer |          | plain   |              |
 name    | text   |          | extended |              |
Indexes:
    "flashtest_index" ubtree (id) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Distribute By: HASH(id)

```











(0 rows)

### 4.3.5 常用视图工具

| 视图类型  | 类型          | 功能描述  | 使用场景  | 函数名称                       |
|---|-------------|---|---|----------------------------|
| 解析  | 全类型         | 用于解析指定表页面，并返回存放解析内容的路径。                                   | <ul style="list-style-type: none"> <li>查看页面信息。</li> <li>查看元组（非用户数据）信息。</li> <li>页面或者元组损坏。</li> <li>元组可见性问题。</li> <li>校验报错问题。</li> </ul> | gs_parse_page_by_path      |
|   | 索引回收队列（URQ） | 用于解析UB-tree索引回收队列关键信息。                                    | <ul style="list-style-type: none"> <li>UB-tree索引空间膨胀。</li> <li>UB-tree索引空间回收异常。</li> <li>校验报错问题。</li> </ul>                             | gs_urq_dump_stat           |
|   | 回滚段（Undo）   | 用于解析指定Undo Record的内容，不包含旧版本元组的数据。                         | <ul style="list-style-type: none"> <li>undo空间膨胀。</li> </ul>   | gs_undo_dump_record        |
|   |             | 用于解析指定事务生成的所有Undo Record，不包含旧版本元组的数据。                     | <ul style="list-style-type: none"> <li>undo回收异常。</li> </ul>   | gs_undo_dump_xid           |
|   |             | 用于解析指定UndoZone中所有Transaction Slot信息。                      | <ul style="list-style-type: none"> <li>回滚异常。</li> <li>日常巡检。</li> </ul>  | gs_undo_translot_dump_slot |
|   |             | 用于解析指定事务对应Transaction Slot信息，包括事务XID和该事务生成的Undo Record范围。 | <ul style="list-style-type: none"> <li>校验报错。</li> <li>可见性判断异常。</li> </ul>   | gs_undo_translot_dump_xid  |
|   |             | 用于解析指定Undo Zone的元信息，显示Undo Record和Transaction Slot指针使用情况。 | <ul style="list-style-type: none"> <li>修改参数。</li> </ul>   | gs_undo_meta_dump_zone     |
|   |             | 用于解析指定Undo Zone对应Undo Space的元信息，显示Undo Record文件使用情况。      |   | gs_undo_meta_dump_spaces   |
| 用于解析指定Undo Zone对应Slot Space的元信息，显示Transaction Slot文件使用情况。 |             | gs_undo_meta_dump_slot                                    |   |                            |

| 视图类型 | 类型            | 功能描述  | 使用场景   | 函数名称                            |
|------|---------------|---|--|---------------------------------|
|      |               | 用于解析数据页和数据页上数据的所有历史版本，并返回存放解析内容的路径。   |  | gs_undo_dump_parsepage_mv       |
|      | 预写日志<br>(WAL) | 用于解析指定LSN范围内的XLOG日志，并返回存放解析内容的路径。可以通过pg_current_xlog_location()获取当前XLOG位置。  | <ul style="list-style-type: none"> <li>WAL日志出错。</li> <li>日志回放出错。</li> <li>页面损坏。</li> </ul>       | gs_xlogdump_lsn                 |
|      |               | 用于解析指定XID的XLOG日志，并返回存放解析内容的路径。可以通过txid_current()获取当前事务ID。   |  | gs_xlogdump_xid                 |
|      |               | 用于解析指定表页面对应的日志，并返回存放解析内容的路径。  |  | gs_xlogdump_tablepath           |
|      |               | 用于解析指定表页面和表页面对应的日志，并返回存放解析内容的路径。可以看作一次执行gs_parse_page_bypath和gs_xlogdump_tablepath。该函数执行的前置条件是表文件存在。如果想查看已删除的表的相关日志，请直接调用gs_xlogdump_tablepath。 |  | gs_xlogdump_parsepage_tablepath |
| 统计   | 回滚段<br>(Undo) | 用于显示Undo模块的统计信息，包括Undo Zone使用情况、Undo链使用情况、Undo模块文件创建删除情况和Undo模块参数设置推荐值。   | <ul style="list-style-type: none"> <li>Undo空间膨胀。</li> <li>Undo资源监控。</li> </ul>                   | gs_stat_undo                    |
|      | 预写日志<br>(WAL) | 用于统计预写日志(WAL)写盘时的内存状态表内容。   | <ul style="list-style-type: none"> <li>WAL写/刷盘监控。</li> </ul>                                     | gs_stat_wal_entrytable          |
|      |               | 用于统计预写日志(WAL)刷盘状态、位置统计信息。   | <ul style="list-style-type: none"> <li>WAL写/刷盘hang住。</li> </ul>                                  | gs_walwriter_flush_position     |
|      |               | 用于统计预写日志(WAL)写刷盘次数频率、数据量以及刷盘文件统计信息。   |  | gs_walwriter_flush_stat         |
| 校验   | 堆表/索引         | 用于离线校验表或者索引文件磁盘页面数据是否异常。  | <ul style="list-style-type: none"> <li>页面损坏或者元组损坏。</li> <li>可见性问题。</li> <li>日志回放出错问题。</li> </ul> | ANALYZE VERIFY                  |
|      |               | 用于校验当前实例当前库物理文件是否存在丢失。  | 文件丢失。  | gs_verify_data_file             |

| 视图类型 | 类型           | 功能描述                                    | 使用场景   | 函数名称                         |
|------|--------------|---|--|------------------------------|
|      | 索引回收队列（URQ）  | 用于校验UB-tree索引回收队列（潜在队列/可用队列/单页面）数据是否异常。 | <ul style="list-style-type: none"> <li>• UB-tree索引空间膨胀。</li> <li>• UB-tree索引空间回收异常。</li> </ul>                                   | gs_verify_urq                |
|      | 回滚段（Undo）    | 用于离线校验Undo Record数据是否存在异常。              | <ul style="list-style-type: none"> <li>• Undo Record异常或者损坏。</li> <li>• 可见性问题。</li> <li>• 回滚出错或者异常。</li> </ul>                    | gs_verify_undo_record        |
|      |              | 用于离线校验Transaction Slot数据是否存在异常。         | <ul style="list-style-type: none"> <li>• Undo Record异常或者损坏。</li> <li>• 可见性问题。</li> <li>• 回滚出错或者异常。</li> </ul>                    | gs_verify_undo_slot          |
|      |              | 用于离线校验Undo元信息数据是否存在异常。                  | <ul style="list-style-type: none"> <li>• 因Undo meta引起的节点无法启动问题。</li> <li>• Undo空间回收异常。</li> <li>• Snapshot too old问题。</li> </ul> | gs_verify_undo_meta          |
| 修复   | 堆表/索引/Undo文件 | 用于基于备机修复主机丢失的物理文件。                      | 堆表/索引/Undo文件丢失。  | gs_repair_file               |
|      | 堆表/索引/Undo页面 | 用于校验并基于备机修复主机受损页面。                      | 堆表/索引/Undo页面损坏。  | gs_verify_and_tryrepair_page |
|      |              | 用于基于备机页面直接修复主机页面。                       |  | gs_repair_page               |
|      |              | 用于基于偏移量对页面的备份进行字节修改。                    |  | gs_edit_page_bypath          |

| 视图类型 | 类型           | 功能描述                               | 使用场景           | 函数名称                   |
|------|--------------|------------------------------------|----------------|------------------------|
|      |              | 用于将修改后的页面覆盖写入到目标页面。                |                | gs_repair_page_by_path |
|      | 回滚段 (Undo)   | 用于重建Undo元信息，如果校验发现Undo元信息没有问题则不重建。 | Undo元信息异常或者损坏。 | gs_repair_undo_by_zone |
|      | 索引回收队列 (URQ) | 用于重建UB-tree索引回收队列。                 | 索引回收队列异常或者损坏。  | gs_repair_urq          |

## 4.3.6 常见问题及定位手段

### 4.3.6.1 snapshot too old

查询SQL执行时间过长或者其它一些原因，Undo无法保存太久的历史数据就可能因为历史版本被强制回收报错。一般情况下需要扩容回滚段空间，但具体问题需要具体分析。

#### 4.3.6.1.1 长事务阻塞 Undo 空间回收

##### 问题现象

1. pg\_log中打印如下错误：  
snapshot too old! the undo record has been forcibly discarded  
xid xxx, the undo size xxx of the transaction exceeds the threshold xxx. trans\_undo\_threshold\_size xxx,undo\_space\_limit\_size xxx.

在真实报错信息中，上文中的xxx为实际数据。

2. global\_recycle\_xid (Undo子系统的全局回收事务XID) 长时间不发生变化。

```
gaussdb=# select * from gs_undo_meta_dump_slot(1,-1);
 zone_id | allocate | recycle | frozen_xid | global_frozen_xid | recycle_xid | global_recycle_xid
-----+-----+-----+-----+-----+-----+-----
      1 | 280     | 248     | 17028      | 17028              | 17025      | 17028
(1 row)
```

3. pg\_running\_xacts与pg\_stat\_activity视图查询存在长事务，阻塞oldestxmin和global\_recycle\_xid推进。如果pg\_running\_xacts中查询活跃事务的xmin和gs\_txid\_oldestxmin相等，且通过pid查询pg\_stat\_activity查询线程执行语句时间过长，则表明有长事务卡住被回收。

```
select * from pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by
xmin::text::bigint asc limit 5;
select * from gs_txid_oldestxmin();
select * from pg_stat_activity where pid = 长事务所在线程PID;
```



### 4.3.6.3 备机读业务报错:"UBTreeSearch::read\_page has conflict with recovery, please try again later"

#### 问题现象

业务在使用备机读时，出现报错（错误码43244），错误信息中包含“UBTreeSearch::read\_page has conflict with recovery, please try again later”关键字。

#### 问题分析

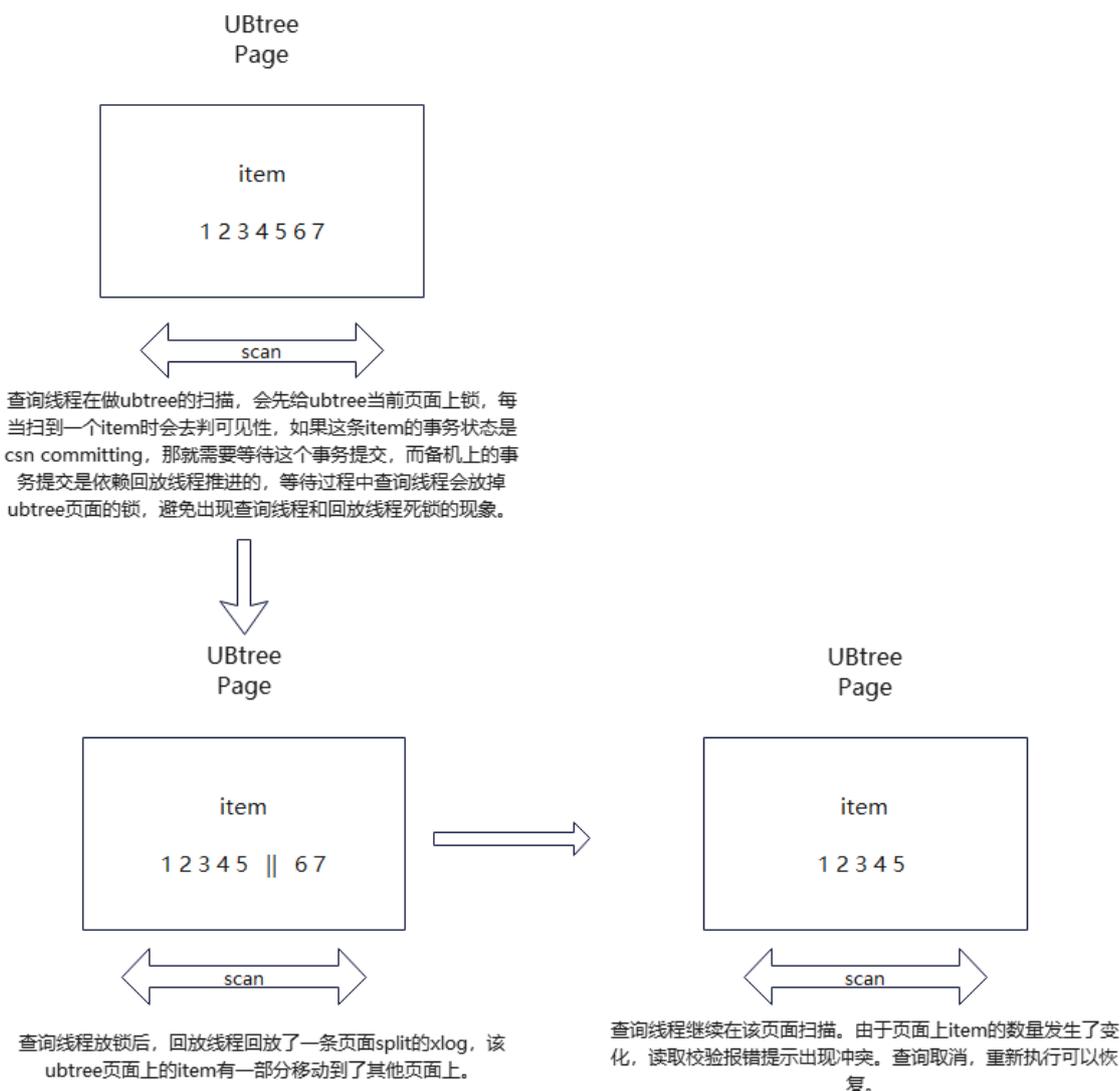
在开启并行回放或串行回放的情况下（查询GUC参数recovery\_parse\_workers和recovery\_max\_workers均是1为串行回放；recovery\_parse\_workers是1，recovery\_max\_workers大于1为并行回放），备机的查询线程在做索引扫描时，会先对索引页面加读锁，每当扫到一个元组时会去判可见性，如果该元组对应的事务处于committing状态，需要等待该事务提交后再判断。而备机上的事务提交是依赖日志回放线程推进的，这个过程中会对索引页面进行修改，因此需要加锁。查询线程在等待过程中会释放索引页面的锁，否则会出现查询线程等待回放线程进行事务提交，而回放线程在等待查询线程释放锁。

该报错仅出现在查询与回放都需要访问同一个索引页面的场景下，查询线程在释放锁并等待事务结束过程中，访问的页面出现被修改的情况。

#### 📖 说明

- 备机查询在扫到committing状态的元组时，需要等待事务提交是因为事务提交的顺序与产生日志的顺序可能是乱序的，例如主机上tx\_1的事务比tx\_2先提交，而备机上tx\_1的commit日志在tx\_2的commit日志之后回放，按照事务提交顺序来看tx\_1对tx\_2应当是可见的，所以需要等待事务提交。
- 备机查询在扫描索引页面时，发现页面元组数量（包含死元组）发生变化后不可重试，是因为在扫描时可能为正向或反向扫描，而举例来说页面发生分裂后一部分元组移动到右页面，在反向扫描的情况下即使重试只能向左扫描读取，无法再保证结果的正确性，并且由于无法分辨发生分裂或者插入，所以不可重试。

图 4-1 问题分析



## 处理方法

出现报错时, 建议重试查询。另外建议选择非频繁更新的索引字段、采用软删除的方式 (物理删除操作在业务低谷期执行), 可以降低出现该报错的概率。

# 5 Foreign Data Wrapper

GaussDB的FDW（Foreign Data Wrapper）可以实现各个GaussDB数据库及远程服务器（包括数据库、文件系统）之间的跨库操作。目前支持的外部数据封装器类型包括file\_fdw。

## 5.1 file\_fdw

file\_fdw模块提供了外部数据封装器file\_fdw，可以用来在服务器的文件系统中访问数据文件。数据文件必须是COPY FROM可读的格式，具体请参见《开发指南》中“SQL参考 > SQL语法 > COPY”章节。使用file\_fdw访问的数据文件是当前可读的，不支持对该数据文件的写入操作。

当前GaussDB会默认编译file\_fdw，initdb的时候会在pg\_catalog schema中创建该插件。

file\_fdw对应的server和外表只允许数据库的初始用户或开启运维模式时的运维管理员创建。

使用file\_fdw创建的外部表可以有如下选项：

- filename  
指定要读取的文件，必需的参数，且必须是一个绝对路径名。
- format  
远端server的文件格式，支持text/csv/binary三种格式，和COPY语句的FORMAT选项相同。
- header  
指定的文件是否有标题行，与COPY语句的HEADER选项相同。
- delimiter  
指定文件的分隔符，与COPY的DELIMITER选项相同。
- quote  
指定文件的引用字符，与COPY的QUOTE选项相同。
- escape  
指定文件的转义字符，与COPY的ESCAPE选项相同。
- null  
指定文件的null字符串，与COPY的NULL选项相同。

- encoding  
指定文件的编码，与COPY的ENCODING选项相同。
- force\_not\_null  
这是一个布尔选项。如果为真，则声明字段的值不应该匹配空字符串（也就是，文件级别null选项）。与COPY的FORCE\_NOT\_NULL选项里的字段相同。

#### 说明

- file\_fdw不支持COPY的OIDS和 FORCE\_QUOTE选项。
- 这些选项只能为外部表或外部表的字段声明，不是file\_fdw的选项，也不是使用file\_fdw的服务器或用户映射的选项。
- 修改表级别的选项需要系统管理员权限。因为安全原因，只有系统管理员能够决定读取的文件。
- 对于一个使用file\_fdw的外部表，EXPLAIN可显示要读取的文件名和文件大小（单位：字节）。指定了COSTS OFF关键字之后，不显示文件大小。

## 使用 file\_fdw

- 创建服务器对象：CREATE SERVER
- 创建用户映射：CREATE USER MAPPING
- 删除用户映射：DROP USER MAPPING
- 删除服务器对象：DROP SERVER

## 注意事项

- 使用file\_fdw需要指定要读取的文件，请先准备好该文件，并授予数据库对该文件的读取权限。
- 不支持DROP EXTENSION file\_fdw操作。

# 6 逻辑复制

逻辑复制分为逻辑解码与数据复制两部分。逻辑解码提取事务级逻辑日志，由业务或数据库中间件解析后完成数据复制。GaussDB数据库支持通过数据迁移工具定期向异构数据库同步数据，不具备实时数据复制能力，不足以支撑与异构数据库间并网运行实时数据同步的诉求。因此，GaussDB数据库提供逻辑解码功能，通过解析Xlog生成逻辑日志，供目标数据库实时解析实现数据复制。降低对目标数据库的形态限制，支持异构数据库、同构异形数据库对数据的同步；支持目标数据库进行数据同步期间的数据可读写，实现数据同步低时延。实现逻辑如图6-1所示，本章节仅介绍逻辑解码。

图 6-1 逻辑复制



## 6.1 逻辑解码

### 6.1.1 逻辑解码概述

#### 功能描述

逻辑解码为逻辑复制提供事务解码的基础能力，GaussDB使用SQL函数接口进行逻辑解码。此方法调用方便，不需使用工具，对接外部工具接口也比较清晰，不需要额外适配。

由于逻辑日志是以事务为单位的，在事务提交后才能输出，且逻辑解码是由用户驱动的；因此为了防止事务开始时的Xlog被系统回收，或所需的事务信息被VACUUM回收，GaussDB新增了逻辑复制槽，用于阻塞Xlog的回收。

一个逻辑复制槽表示一个更改流，这些更改可以在其它集群上以它们在原集群上产生的顺序重新执行。每个逻辑复制槽都由其对应逻辑日志的获取者维护。如果处于流式解码中的逻辑复制槽所在库不存在业务，则该复制槽会依照其他库的日志位置来推

进。活跃状态的LSN序逻辑复制槽在处理到活跃事务快照日志时可以根据当前日志的LSN推进复制槽；活跃状态的CSN序逻辑复制槽在处理到虚拟事务日志时可以根据当前日志的CSN推进复制槽。

## 前提条件

- 逻辑日志目前从DN中抽取，如果进行逻辑复制，应使用SSL连接，因此需要保证相应节点上的GUC参数`ssl`设置为`on`。

### 📖 说明

为避免安全风险，请保证启用SSL连接。

- 设置GUC参数`wal_level`为`logical`。
- 设置GUC参数`max_replication_slots`>=每个DN节点所需的（物理流复制槽数+备份槽数+逻辑复制槽数）。

### 📖 说明

- 每个逻辑复制槽仅解码单个数据库的修改；如需解码多个数据库，需创建多个逻辑复制槽。
- 多路逻辑复制同步时，源端数据库需为每条逻辑复制链路创建独立逻辑复制槽。
- 单个CN节点最多同时运行10个分布式强一致解码任务；单个DN节点最多同时运行20个解码任务。
- 用户需要通过DN端口连接数据库，才可以直接使用SQL函数接口进行逻辑解码操作，相关操作参考章节[使用SQL函数接口进行逻辑解码](#)。如果使用CN端口连接数据库，则需要通过EXECUTE DIRECT ON (datanode\_name) 'statement'语句来执行SQL函数。
- 仅限初始用户和拥有REPLICATION权限的用户进行操作。三权分立关闭时数据库管理员可进行逻辑复制操作，三权分立开启时不允许数据库管理员进行逻辑复制操作。

## 注意事项

- 逻辑解码不支持DDL。
- 在执行特定的DDL语句（如普通表truncate或分区表exchange）时，可能造成解码数据丢失。
- 不支持数据页复制的DML解码。
- 当执行DDL语句（如alter table）后，该DDL语句前尚未解码的物理日志可能会丢失。
- 逻辑复制不支持集群在线扩容。在线扩容前，需要删除已存在的逻辑复制槽，扩容完成后重新创建。
- 单条元组大小不超过1GB，考虑解码结果可能大于插入数据，因此建议单条元组大小不超过500MB。
- 不支持压缩表的DML语句解码。
- GaussDB支持解码的数据类型为：INTEGER、BIGINT、SMALLINT、TINYINT、SERIAL、SMALLSERIAL、BIGSERIAL、FLOAT、DOUBLE PRECISION、BOOLEAN、BIT(n)、BIT VARYING(n)、DATE、TIME[WITHOUT TIME ZONE]、TIMESTAMP[WITHOUT TIME ZONE]、CHAR(n)、VARCHAR(n)、TEXT、CLOB（解码成TEXT格式）。
- 逻辑复制槽名称必须小于64个字符，仅支持小写字母、数字以及“\_”、“?”、“-”、“.”字符，且不支持“-”或“..”单独作为复制槽名称。

- 当逻辑复制槽所在数据库被删除后，这些复制槽变为不可用状态，需要用户手动删除。
- 对多个数据库的解码需要分别在数据库内创建流复制槽并开始解码，每个数据库的解码都需要单独扫一遍日志。
- 逻辑解码时不支持强切，强切后需要重新全量导出数据。
- 在事务中执行DDL语句后，该DDL语句与之后的语句不会被解码。
- 备机解码时，switchover和failover可能导致解码数据增多，需手动过滤。Quorum协议下，switchover和failover选择升主的备机，需要与当前主机日志同步。
- 不允许主备，多个备机同时使用同一个复制槽解码，否则会产生数据不一致的情况。
- 只支持CN和主DN创建删除复制槽。
- 数据库故障重启或逻辑复制进程重启后，解码数据可能存在重复，用户需自己过滤。
- 计算机内核故障后，解码可能存在乱码，需手动或自动过滤。
- 请确保在创建逻辑复制槽过程中长事务未启动，启动长事务会阻塞逻辑复制槽的创建。如果创建复制槽时因为长事务阻塞，可通过执行SQL函数 `pg_terminate_backend(创建该复制槽线程id)` 来手动停止创建。
- 不支持interval partition表复制。
- 为解析某个Astore表的UPDATE和DELETE语句，需为此表配置REPLICA IDENTITY属性，在此表无主键时需要配置为FULL，配置方式请参考《开发指南》中“SQL参考 > SQL语法 > ALTER TABLE”章节中“**REPLICA IDENTITY { DEFAULT | USING INDEX index\_name | FULL | NOTHING }**”字段。
- 禁止在使用逻辑复制槽时在其他节点对该复制槽进行操作，删除复制槽的操作需在该复制槽停止解码后执行。
- 基于目标库可能需要源库的系统状态信息考虑，逻辑解码仅自动过滤模式'pg\_catalog'和'pg\_toast'下OID小于16384的系统表的逻辑日志。若目标库不需要复制其他相关系统表的内容，则逻辑日志回放过程中需要对相关系统表进行过滤。
- 在开启逻辑复制的场景下，如需创建包含系统列的主键索引，必须将该表的REPLICA IDENTITY属性设置为FULL或是使用USING INDEX指定不包含系统列的、唯一的、非局部的、不可延迟的、仅包括标记为NOT NULL的列的索引。
- 对于扩容或升级前已存在的复制表场景，需要对复制表手动配置logical\_repl\_node属性或RESET为默认值，配置方式请参考《开发指南》中“SQL参考 > SQL语法 > ALTER TABLE”章节中“**storage\_parameter**”参数的使用说明，以及“logical\_repl\_node”属性相关说明。
- 若一个事务的子事务过多导致落盘文件过多，退出解码时需执行SQL函数 `pg_terminate_backend(逻辑解码的walsender线程id)`来手动停止解码，而且退出时延增加约为1分钟/30万个子事务。因此在开启逻辑解码时，若一个事务的子事务数量达到5万，会打印一条WARNING日志。
- 当逻辑复制槽处于非活跃状态，且设置GUC参数enable\_xlog\_prune=on、enable\_logicalrepl\_xlog\_prune=on、max\_size\_for\_xlog\_retention为非零值，且备份槽或逻辑复制槽导致保留日志段数已超过GUC参数wal\_keep\_segments同时其他复制槽并未导致更多的保留日志段数时，如果max\_size\_for\_xlog\_retention大于0且当前逻辑复制槽导致保留日志的段数（每段日志大小为16MB）超过max\_size\_for\_xlog\_retention，或者max\_size\_for\_xlog\_retention小于0且磁盘使用率达到(-max\_size\_for\_xlog\_retention)/100，当前逻辑复制槽会强制失效，其

restart\_lsn将被设置为FFFFFFFF/FFFFFFFF。该状态的逻辑复制槽不参与阻塞日志回收或系统表历史版本的回收，但仍占用复制槽的限制数量，需要手动删除。

- 备机解码启动后，向主机发送复制槽推进指令后会占用主机上对应的逻辑复制槽（即标识为活跃状态）。在此之前主机上对应逻辑复制槽为非活跃状态，此状态下如果满足逻辑复制槽强制失效条件则会被标记为失效（即restart\_lsn将被设置为FFFFFFFF/FFFFFFFF），备机将无法推进主机复制槽，且备机回放完成复制槽失效日志后当前复制槽的备机解码断开后将无法重连。
- 不活跃的逻辑复制槽将阻塞WAL日志回收和系统表元组历史版本清理，导致磁盘日志堆积和系统表扫描性能下降，因此不再使用的逻辑复制槽请及时清理。需要特别注意，在升级提交之前观察期内使用DN扩展IP连接DN创建的逻辑复制槽，在升级回滚之前务必手动清理，否则随着DN扩展IP特性回滚无法直连DN清理。
- 分布式强一致逻辑解码（连接CN解码）仅支持GTM-Lite分布式部署及流式解码，不支持CN连接备DN进行解码、SQL逻辑解码函数、在线扩容、全局索引。
- 针对分布式强一致逻辑解码（连接CN解码）功能，CN高可用由业务负责切换。
- CN上的CSN序逻辑复制槽仅起到占位作用，不随着逻辑解码的进行而推进，同时也不会阻塞日志回收，因此此类复制槽xmin、catalog\_xmin、restart\_lsn、confirmed\_flush和confirmed\_csn字段均不显示。
- 通过协议连接CN创建逻辑复制槽仅支持CSN序复制槽，通过协议连接DN创建逻辑复制槽仅支持LSN序复制槽。
- 针对分布式解码，对于故障报错或者手动停止解码客户端等场景，需等待15s再次重试解码，如有复制槽占用则需通过执行SQL函数pg\_terminate\_backend(占用该复制槽线程id)来手动解除复制槽占用。
- 在CN上创建复制槽失败报错后，需要在CN上进行复制槽删除操作，然后在CN上重新创建复制槽。
- 在CN上删除逻辑复制槽时，若为LSN序逻辑复制槽，则仅删除当前节点复制槽，其他节点同名逻辑复制槽不受影响；否则只要其他节点有残留同名CSN序逻辑复制槽，执行删除时不会因为某些节点不存在复制槽而报错，同时所有节点的同名复制槽会被成功删除；如果任何节点均不存在该复制槽，则报错。
- 在CN上创建CSN序逻辑复制槽时，某些节点如残留同名LSN序逻辑复制槽，需在这些节点执行删除残留复制槽的操作。否则会在除当前CN节点外，其他不存在同名复制槽的CN和主DN节点上创建CSN序逻辑复制槽。
- 如果当前CN节点残留LSN序逻辑复制槽，同时其他某些节点上残留同名CSN序逻辑复制槽，则在当前CN节点上执行删除复制槽操作仅会删除本地LSN序逻辑复制槽，待删除完成再次执行删除操作方可删除其他节点的同名复制槽。
- 解码使用JSON格式输出时不支持数据列包含特殊字符（如'\0'空字符），解码输出列内容将出现被截断现象。
- 当同一事务产生大量需要落盘的子事务时，同时打开的文件句柄可能会超限，需将GUC参数max\_files\_per\_process配置成大于子事务数量上限的两倍。
- sql\_decoding将UPDATE语句解码为DELETE+INSERT操作。
- 在海量子事务场景下，逻辑解码内存控制机制会根据用户设置的阈值，将逻辑日志落盘，落盘文件数量与子事务数量正相关，磁盘读写速率会成为逻辑解码瓶颈；存在海量子事务场景时需要关注落盘时延：当数据远端存储时，读时延开销指数上升，写时延倍数上升，需要业务层面避免产生海量子事务。

建议如下：

- a. 排查使用子事务场景，确认业务子事务数量。
- b. 修改max-changes-in-memory、max-reorderbuffer-in-memory、max-txn-in-memory等参数，提高临时文件生成阈值。

- c. 根据业务实际情况裁剪子事务数量，建议不超过1000个子事务。

## SQL 函数解码性能

在Benchmarksq1-5.0的100warehouse场景下，采用pg\_logical\_slot\_get\_changes时：

- 单次解码数据量4K行（对应约5MB~10MB日志），解码性能0.3MB/s~0.5MB/s。
- 单次解码数据量32K行（对应约40MB~80MB日志），解码性能3MB/s~5MB/s。
- 单次解码数据量256K行（对应约320MB~640MB日志），解码性能3MB/s~5MB/s。
- 单次解码数据量再增大，解码性能无明显提升。

如果采用pg\_logical\_slot\_peek\_changes + pg\_replication\_slot\_advance方式，解码性能相比采用pg\_logical\_slot\_get\_changes时要下降30%~50%。

### 6.1.2 逻辑解码选项

逻辑解码选项可以用来为本次逻辑解码提供限制或额外功能，如“解码结果是否包含事务号”、“解码时是否忽略空事务”等等。对于具体配置方法，SQL函数解码请参考《开发指南》中“SQL参考 > 函数和操作符 > 系统管理函数 > 逻辑复制函数”章节中函数pg\_logical\_slot\_peek\_changes的可选入参'options\_name'和'options\_value'，JDBC流式解码请参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 示例：逻辑复制代码示例”章节中示例代码中函数withSlotOption的使用方法。

#### 通用选项

串行解码和并行解码均可配置，但可能无效，请参考相关选项详细说明。

- include-xids：  
解码出的data列是否包含xid信息。  
取值范围：bool型，默认值为true。
  - false：设为false时，解码出的data列不包含xid信息。
  - true：设为true时，解码出的data列包含xid信息。
- skip-empty-xacts：  
解码时是否忽略空事务信息。  
取值范围：bool型，默认值为false。
  - false：设为false时，解码时不忽略空事务信息。
  - true：设为true时，解码时会忽略空事务信息。
- include-timestamp：  
解码信息是否包含commit时间戳。  
取值范围：bool型，默认值为false。
  - false：设为false时，解码信息不包含commit时间戳。
  - true：设为true时，解码信息包含commit时间戳。
- only-local：  
是否仅解码本地日志。  
取值范围：bool型，默认值为true。

- false: 设为false时, 解码非本地日志和本地日志。
- true: 设为true时, 仅解码本地日志。
- white-table-list:  
白名单参数, 包含需要进行解码的Schema和表名。  
取值范围: 包含白名单中表名的字符串, 不同的表以','为分隔符进行隔离; 使用'\*'来模糊匹配所有情况; Schema名和表名间以'.'分隔, 不允许存在任意空白符。例如:  

```
select * from pg_logical_slot_peek_changes('slot1', NULL, 4096, 'white-table-list', 'public.t1,public.t2,*t3,my_schema.*');
```
- max-txn-in-memory:  
内存管控参数, 单位为MB, 单个事务占用内存大于该值即进行落盘。  
取值范围: 0~100的整型, 默认值为0, 即不开启此种管控。
- max-reorderbuffer-in-memory  
内存管控参数, 单位为GB, 拼接-发送线程中正在拼接的事务总内存 (包含缓存) 大于该值则对当前解码事务进行落盘。  
取值范围: 0~100的整型, 默认值为0, 即不开启此种管控。
- include-user:  
事务的BEGIN逻辑日志是否输出事务的用户名。事务的用户名特指授权用户 (执行事务对应会话的登录用户), 它在事务的整个执行过程中不会发生变化。  
取值范围: bool型, 默认值为false。
  - false: 设为false时, 事物的BEGIN逻辑日志不输出事务的用户名。
  - true: 设为true时, 事物的BEGIN逻辑日志输出事务的用户名。
- exclude-userids:  
黑名单用户的OID参数。只支持SQL函数解码时使用, 不支持启动逻辑解码任务时指定。  
取值范围: 字符串类型, 指定黑名单用户的OID, 多个OID通过','分隔, 不校验用户OID是否存在。
- exclude-users:  
黑名单用户的名称列表。  
取值范围: 字符串类型, 指定黑名单用户名, 通过','分隔, 不校验用户名是否存在。
- dynamic-resolution:  
是否动态解析黑名单用户名。  
取值范围: bool型, 默认值为true。
  - false: 设为false时, 当解码观测到黑名单exclude-users中用户不存在时将会报错并退出逻辑解码。
  - true: 设为true时, 当解码观测到黑名单exclude-users中用户不存在时继续解码。
- standby-connection:  
**仅流式解码设置**, 是否仅限制备机解码。只支持SQL函数解码时使用, 不支持启动逻辑解码任务时指定。  
取值范围: bool型, 默认值为false。
  - true: 设为true时, 仅允许连接备机解码, 连接主机解码时会报错退出。

- false: 设为false时, 不做限制, 允许连接主机或备机解码。
- sender-timeout:
 

**仅流式解码设置**, GaussDB与客户端的心跳超时阈值。当该时间段内没有收到客户端任何消息, 逻辑解码将主动停止, 并断开和客户端的连接。单位为毫秒 (ms)。

取值范围: 0~2147483647的int型, 默认值取决于GUC参数 logical\_sender\_timeout的配置值。
- enable-heartbeat:
 

**仅流式解码时设置**, 是否输出心跳日志。

取值范围: bool型, 默认值为false。

  - true: 设为true时, 输出心跳日志。
  - false: 设为false时, 不输出心跳日志。

### 📖 说明

若开启心跳日志选项, 此处说明心跳日志如何解析: 二进制格式首先是字符'h'表示是消息是心跳日志, 之后是心跳日志内容, 分别是8字节uint64, 直连DN解码场景代表LSN, 表示发送心跳逻辑日志时读取的WAL日志结束位置, 而在分布式强一致解码场景为CSN, 表示发送心跳逻辑日志时已发送的解码日志事务CSN; 8字节uint64, 直连DN解码场景代表LSN, 表示发送心跳逻辑日志时刻已经落盘的WAL日志的位置, 而在分布式强一致解码场景为CSN, 表示集群下一个提交事务将获得的CSN; 8字节int64代表时间戳 (从1970年1月1日开始), 表示最新解码到的事务日志或检查点日志的产生时间戳。关于消息结束符: 如果是二进制格式则为字符'F', 如果格式为TEXT或者JSON且为批量发送则结束符为0, 否则没有结束符。消息内容采用大端字节序进行数据传输。具体格式见下图 (考虑到前向兼容性, 相关部分仍保留着LSN的命名方式, 实际含义依具体场景而定):

|                   |   |            |   |                          |                         |                          |     |
|-------------------|---|------------|---|--------------------------|-------------------------|--------------------------|-----|
| 二进制格式(批量发送与非批量发送) | uint32 len  | uint64 lsn | 'h'   | uint64 latest_decode_lsn | uint64 latest_flush_lsn | int64 latest_decode_time | 'F' |
| text/json+批量发送    | uint32 len  | uint64 lsn | char* "HeartBeat: latest_decode_lsn: XX, latest_flush_lsn: XX, latest_decoded_wal_time: XX" |                          |                         |                          | '0' |
| text/json+非批量     | char* "HeartBeat: latest_decode_lsn: XX, latest_flush_lsn: XX, latest_decoded_wal_time: XX" |            |   |                          |                         |                          |     |

- parallel-decode-num:
 

**仅流式解码设置有效**, 并行解码的Decoder线程数量; 系统函数调用场景下此选项无效, 仅校验取值范围。

取值范围: 1~20的int型, 取1表示按照原有的串行逻辑进行解码, 取其余值即为开启并行解码, 默认值为1。

### 须知

当parallel-decode-num不配置 (即为默认值1) 或显式配置为1时, 下述“并行解码”中的选项不可配置。

- output-order:
 

**仅流式解码设置有效**, 是否使用CSN顺序输出解码结果; 系统函数调用场景下此选项无效, 仅校验取值范围。

取值范围: 0或1的int型, 默认值为0。

  - 0: 设为0时, 解码结果按照事务的COMMIT LSN排序, 当且仅当解码复制槽的confirmed\_csn列值为0 (即不显示) 时可使用该方式, 否则报错。
  - 1: 设为1时, 解码结果按照事务的CSN排序, 当且仅当解码复制槽的confirmed\_csn列值为非零值时可使用该方式, 否则报错。

### 须知

- 当output-order不配置（即为默认值0，按照COMMIT LSN排序）或显式配置为0时，下述“分布式强一致解码”中的选项不可配置。
  - 在流式解码场景，DN收到来自CN的逻辑解码连接时，output-order选项失效，默认采用CSN序解码。
- 
- auto-advance:  
**仅流式解码设置有效**，是否允许自主推进逻辑复制槽。  
取值范围：boolean型，默认值为false。
    - true：设为true时，在已发送日志都被确认推进且没有待发送事务时，推进逻辑复制槽到当前解码位置。
    - false：设为false时，完全交由复制业务调用日志确认接口推进逻辑复制槽。
  - skip-generated-columns:  
逻辑解码控制参数，用于跳过生成列的输出。对UPDATE和DELETE的旧元组无效，相应元组始终会输出生成列。分布式版本暂不支持生成列，此配置选项暂无实际影响。只支持SQL函数解码时使用，不支持启动逻辑解码任务时指定。  
取值范围：boolean型，默认值为false。
    - true：表示不输出生成列的解码结果。
    - false：表示输出生成列的解码结果。

## 分布式强一致解码

- logical-receiver-num:  
**仅流式解码设置有效**，分布式解码启动的logical\_receiver的数量，系统函数调用场景下此选项无效，仅校验取值范围。  
取值范围：1~20的int型，默认值为1。当该值被设置为比当前集群分片数更大时，将被修改为分片数。
- slice-id:  
仅连接DN解码时设置，指定当前DN所在的分片号，用于复制表解码。  
取值范围：0~8192的int型，默认值为-1，即不指定分片号，但在解码到复制表时会报错。

### 注意

该配置选项在尝试连接DN使用CSN序逻辑复制槽（confirmed\_csn为非0值的复制槽）进行解码时使用，用来表示自己的分片号（即第几个分片，第一个分片则输入0），如果不设置该参数（即使用默认值-1）在解码到复制表时将会报错。此参数用于使用连接CN的分布式解码时，CN从DN收集解码结果时使用。不建议在此场景下手动连接DN解码。

- start-position:  
**仅连接DN设置**，主要功能为过滤掉小于指定CSN对应的事务，以及针对指定的CSN对应的事务，过滤掉小于指定LSN的日志，且**指定CSN对应事务的BEGIN日志一定被过滤掉**。

取值范围：字符串类型，可以解析为以 '/' 分隔，左右两侧分别为代表 CSN 和 LSN 的两个 uint64 类型。

### 注意

该配置选项用于 CN 解码时，CN 建立与 DN 的连接后发送解码请求时使用此配置选项过滤可能已经被接收过的日志。不建议在此场景下手动连接 DN 解码使用此参数。

## 串行解码

- force-binary:  
是否以二进制格式输出解码结果，针对不同场景呈现不同行为。
  - 针对系统函数 pg\_logical\_slot\_get\_binary\_changes 和 pg\_logical\_slot\_peek\_binary\_changes:  
取值范围：bool 型，默认值为 false。此值无实际意义，均以二进制格式输出解码结果。
  - 针对系统函数 pg\_logical\_slot\_get\_changes、pg\_logical\_slot\_peek\_changes 和 pg\_logical\_get\_area\_changes:  
取值范围：仅取 false 值的 bool 型。以文本格式输出解码结果。
  - 针对流式解码:  
取值范围：bool 型，默认值为 false。此值无实际意义，均以文本格式输出解码结果。

## 并行解码

以下配置选项仅限流式解码设置。

- decode-style:  
指定解码格式。  
取值范围：char 型的字符 'j'、't' 或 'b'，分别代表 JSON 格式，TEXT 格式及二进制格式。默认值为 'b' 即二进制格式解码。  
对于 JSON 格式和 TEXT 格式解码，开启批量发送选项时的解码结果中，每条解码语句的前 4 字节组成的 uint32 代表该条语句总字节数（不包含该 uint32 类型占用的 4 字节，0 代表本批次解码结束），8 字节 uint64 代表相应 lsn（begin 对应 first\_lsn，commit 对应 end\_lsn，其他场景对应该条语句的 lsn）。

## 说明

二进制格式编码规则如下所示：

1. 前4字节代表接下来到语句级别分隔符字母P（不含）或者该批次结束符F（不含）的解码结果的总字节数，该值如果为0代表本批次解码结束。
  2. 接下来8字节uint64代表相应lsn（begin对应first\_lsn，commit对应end\_lsn，其他场景对应该条语句的lsn）。
  3. 接下来1字节的字母有5种B/C/I/U/D，分别代表begin/commit/insert/update/delete。
  4. 第3步字母为B时：
    1. 接下来的8字节uint64代表CSN。
    2. 接下来的8字节uint64代表first\_lsn。
    3. 【该部分为可选项】接下来的1字节字母如果为T，则代表后面4字节uint32表示该事务commit时间戳长度，再后面等同于该长度的字符为时间戳字符串。
    4. 【该部分为可选项】接下来的1字节字母如果为N，则代表后面4字节uint32表示该事务用户名的长度，再后面等同于该长度的字符为事务的用户名字。
    5. 因为之后仍可能有解码语句，接下来会有1字节字母P或F作为语句间的分隔符，P代表本批次仍有解码的语句，F代表本批次完成。
  5. 第3步字母为C时：
    1. 【该部分为可选项】接下来1字节字母如果为X，则代表后面的8字节uint64表示xid。
    2. 【该部分为可选项】接下来的1字节字母如果为T，则代表后面4字节uint32表示时间戳长度，再后面等同于该长度的字符为时间戳字符串。
    3. 因为批量发送日志时，一个COMMIT日志解码之后可能仍有其他事务的解码结果，接下来的1字节字母如果为P则表示该批次仍需解码，如果为F则表示该批次解码结束。
  6. 第3步字母为I/U/D时：
    1. 接下来的2字节uint16代表Schema名的长度。
    2. 按照上述长度读取Schema名。
    3. 接下来的2字节uint16代表table名的长度。
    4. 按照上述长度读取table名。
    5. 【该部分为可选项】接下来1字节字母如果为N代表为新元组，如果为O代表为旧元组，这里先发送新元组。
      1. 接下来的2字节uint16代表该元组需要解码的列数，记为attrnum。
      2. 以下流程重复attrnum次。
        1. 接下来2字节uint16代表列名的长度。
        2. 按照上述长度读取列名。
        3. 接下来4字节uint32代表当前列类型的Oid。
        4. 接下来4字节uint32代表当前列的值（以字符串格式存储）的长度，如果为0xFFFFFFFF则表示NULL，如果为0则表示长度为0的字符串。
        5. 按照上述长度读取列值。
    6. 因为之后仍可能有解码语句，接下来的1字节字母如果为P则表示该批次仍需解码，如果为F则表示该批次解码结束。
- sending-batch：

指定是否批量发送。

取值范围：0或1的int型，默认值为0。

    - 0：设为0时，表示逐条发送解码结果。
    - 1：设为1时，表示解码结果累积到达1MB则批量发送解码结果。

开启批量发送的场景中，当解码格式为'j'或't'时，在原来的每条解码语句之前会附加一个uint32类型，表示本条解码结果长度（长度不包含当前的uint32类型），以及一个uint64类型，表示当前解码结果对应的lsn。

### 须知

在CSN序解码（即output-order设置为1）场景下，批量发送仅限于单个事务内（即如果一个事务有多条较小的语句会采用批量发送），即不会使用批量发送功能在同一批次里发送多个事务，且BEGIN和COMMIT语句不会批量发送。

- parallel-queue-size：  
指定并行逻辑解码线程间进行交互的队列长度。  
取值范围：2~1024的int型，且必须为2的整数幂，默认值为128。  
队列长度和解码过程的内存使用量正相关。

## 6.1.3 使用 SQL 函数接口进行逻辑解码

GaussDB可以通过调用SQL函数，进行创建、删除、推进逻辑复制槽，获取解码后的事务日志。

### 操作步骤

**步骤1** 以具有REPLICATION权限的用户登录GaussDB集群任一主机。

**步骤2** 使用如下命令通过CN端口连接数据库。

```
gsql -U user1 -d gaussdb -p 40000 -r
```

其中，user1为用户名，gaussdb为需要连接的数据库名称，40000为数据库CN端口号，用户可根据实际情况替换。

**步骤3** 创建名称为slot1的逻辑复制槽。

```
gaussdb=# SELECT * FROM pg_create_logical_replication_slot('slot1', 'mppdb_decoding');
slotname | xlog_position
-----+-----
slot1    | 0/601C150
(1 row)
```

**步骤4** 在数据库中创建表t，并向表t中插入数据。

```
gaussdb=# CREATE TABLE t(a int PRIMARY KEY, b int);
gaussdb=# INSERT INTO t VALUES(3,3);
```

**步骤5** 读取所有DN上复制槽slot1解码结果，解码条数为4096。

### 📖 说明

逻辑解码选项可参考[逻辑解码选项](#)。

```
gaussdb=# EXECUTE DIRECT ON DATANODES 'SELECT * FROM pg_logical_slot_peek_changes("slot1", NULL, 4096);';
location | xid | data
-----+-----
+-----+-----
-----+-----
0/601C188 | 1010023 | BEGIN 1010023
0/601ED60 | 1010023 | COMMIT 1010023 CSN 1010022
0/601ED60 | 1010024 | BEGIN 1010024
0/601ED60 | 1010024 | {"table_name":"public.t","op_type":"INSERT","columns_name":
["a","b"],"columns_type":["integer","integer"],"columns_val":["3","3"],"old_keys_name":["old_keys_type":
[],"old_keys_val":[]]}
0/601EED8 | 1010024 | COMMIT 1010024 CSN 1010023
(5 rows)
```

**步骤6** 删除逻辑复制槽slot1。

```
gaussdb=# SELECT * FROM pg_drop_replication_slot('slot1');
pg_drop_replication_slot
-----
(1 row)
```

----结束

## 6.1.4 使用流式解码实现数据逻辑复制

第三方复制工具通过流式逻辑解码从GaussDB抽取逻辑日志后到对端数据库回放。对于使用JDBC连接数据库的复制工具，具体代码请参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 示例：逻辑复制代码示例”章节。