

云数据库 GaussDB

# 特性指南（分布式\_V2.0-8.x）

文档版本 01  
发布日期 2025-01-24



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 支持和约束	1
1.1.3 使用	2
1.2 增量物化视图	3
1.2.1 概述	3
1.2.2 支持和约束	3
1.2.3 使用	4
<b>2 设置密态等值查询</b>	<b>6</b>
2.1 密态等值查询概述	6
2.2 使用 gsql 操作密态数据库	9
2.3 使用 JDBC 操作密态数据库	11
2.4 前向兼容与安全增强	16
2.5 密态支持函数/存储过程	19
<b>3 设置账本数据库</b>	<b>22</b>
3.1 账本数据库概述	22
3.2 查看账本历史操作记录	24
3.3 校验账本数据一致性	26
3.4 归档账本数据库	27
3.5 修复账本数据库	28
<b>4 逻辑复制</b>	<b>31</b>
4.1 逻辑解码	31
4.1.1 逻辑解码概述	31
4.1.2 逻辑解码选项	37
4.1.3 使用 SQL 函数接口进行逻辑解码	45
4.1.4 使用流式解码实现数据逻辑复制	46
4.1.5 逻辑解码支持 DDL	46
4.1.6 逻辑解码数据找回功能	54
<b>5 分区表</b>	<b>56</b>
5.1 大容量数据库	56
5.1.1 大容量数据库背景介绍	56

5.1.2 表分区技术.....	56
5.1.3 数据分区查找优化.....	57
5.1.4 数据分区运维管理.....	57
5.2 分区表介绍.....	58
5.2.1 基本概念.....	58
5.2.1.1 分区表 (母表).....	58
5.2.1.2 分区 (分区子表、子分区).....	59
5.2.1.3 分区键.....	60
5.2.2 分区策略.....	60
5.2.2.1 范围分区.....	60
5.2.2.2 哈希分区.....	62
5.2.2.3 列表分区.....	63
5.2.2.4 分区表对导入操作的性能影响.....	63
5.2.3 分区基本使用.....	64
5.2.3.1 创建分区表.....	65
5.2.3.2 使用和管理分区表.....	66
5.2.3.3 分区表 DQL/DML.....	67
5.3 分区表查询优化.....	68
5.3.1 分区剪枝.....	68
5.3.1.1 分区表静态剪枝.....	69
5.3.1.2 分区表动态剪枝.....	73
5.3.1.2.1 PBE 动态剪枝.....	73
5.3.1.2.2 参数化路径动态剪枝.....	76
5.3.2 分区索引.....	80
5.3.3 分区表统计信息.....	83
5.3.3.1 级联收集统计信息.....	83
5.3.3.2 分区级统计信息.....	86
5.3.4 Partition-wise Join.....	90
5.3.4.1 非 SMP 场景下的 Partition-wise Join.....	90
5.3.4.2 SMP 场景下的 Full Partition-wise Join.....	92
5.4 分区表运维管理.....	94
5.4.1 新增分区.....	94
5.4.1.1 向范围分区表新增分区.....	95
5.4.1.2 向列表分区表新增分区.....	95
5.4.2 删除分区.....	95
5.4.3 交换分区.....	96
5.4.4 清空分区.....	97
5.4.5 分割分区.....	97
5.4.5.1 对范围分区表分割分区.....	98
5.4.5.2 对列表分区表分割分区.....	98
5.4.6 合并分区.....	99
5.4.7 移动分区.....	99

5.4.8 重命名分区.....	100
5.4.8.1 对分区表重命名分区.....	100
5.4.8.2 对 Local 索引重命名索引分区.....	100
5.4.9 分区表行迁移.....	100
5.4.10 分区表索引重建/不可用.....	101
5.4.10.1 索引重建/不可用.....	101
5.4.10.2 Local 索引分区重建/不可用.....	101
5.5 分区表系统视图&DFX.....	101
5.5.1 分区表相关系统视图.....	102
5.5.2 分区表相关内置工具函数.....	102
<b>6 存储引擎.....</b>	<b>106</b>
6.1 存储引擎体系架构.....	106
6.1.1 存储引擎体系架构概述.....	106
6.1.1.1 静态编译架构.....	106
6.1.1.2 通用数据库服务层.....	107
6.1.2 设置存储引擎.....	107
6.1.3 存储引擎更新说明.....	108
6.1.3.1 GaussDB 内核 505 版本.....	108
6.1.3.2 GaussDB 内核 503 版本.....	108
6.1.3.3 GaussDB 内核 R2 版本.....	109
6.2 Astore 存储引擎.....	109
6.2.1 Astore 简介.....	109
6.3 Ustore 存储引擎.....	110
6.3.1 Ustore 简介.....	110
6.3.1.1 Ustore 特性与规格.....	110
6.3.1.1.1 特性约束.....	110
6.3.1.1.2 存储规格.....	110
6.3.1.2 使用 Ustore 进行测试.....	111
6.3.1.3 Ustore 的最佳实践.....	112
6.3.1.3.1 怎么配置 init_td 大小.....	112
6.3.1.3.2 怎么配置 fillfactor 大小.....	113
6.3.1.3.3 在线校验功能.....	114
6.3.1.3.4 怎么配置回滚段大小.....	114
6.3.2 存储格式.....	115
6.3.2.1 RCR Uheap.....	115
6.3.2.1.1 RCR Uheap 多版本管理.....	115
6.3.2.1.2 RCR Uheap 可见性机制.....	115
6.3.2.1.3 RCR Uheap 空闲空间管理.....	116
6.3.2.2 UBTree.....	116
6.3.2.2.1 RCR UBTree.....	116
6.3.2.2.2 PCR UBTree.....	119
6.3.2.3 Undo.....	121

6.3.2.3.1 回滚段管理.....	121
6.3.2.3.2 文件组织结构.....	121
6.3.2.3.3 空间管理.....	121
6.3.2.4 Enhanced Toast.....	122
6.3.2.4.1 概述.....	122
6.3.2.4.2 Enhanced Toast 存储结构.....	122
6.3.2.4.3 Enhanced Toast 使用.....	122
6.3.2.4.4 Enhanced Toast 增删改查.....	122
6.3.2.4.5 Enhanced Toast 相关 DDL 操作.....	123
6.3.2.4.6 Enhanced Toast 运维管理.....	125
6.3.3 Ustore 事务模型.....	126
6.3.3.1 事务提交.....	126
6.3.3.2 事务回滚.....	126
6.3.4 闪回恢复.....	127
6.3.4.1 闪回查询.....	127
6.3.4.2 闪回表.....	129
6.3.4.3 闪回 DROP/TRUNCATE.....	131
6.3.5 常用视图工具.....	139
6.3.6 常见问题及定位手段.....	142
6.3.6.1 snapshot too old.....	142
6.3.6.1.1 长事务阻塞 Undo 空间回收.....	142
6.3.6.1.2 大量回滚事务拖慢 Undo 空间回收.....	143
6.3.6.2 storage test error.....	144
6.3.6.3 备机读业务报错:"UBTreeSearch::read_page has conflict with recovery, please try again later".....	144
6.3.6.4 长查询执行期间大量并发更新偶现写入性能下降.....	145
6.4 数据生命周期管理-OLTP 表压缩.....	146
6.4.1 特性简介.....	146
6.4.2 特性约束.....	146
6.4.3 特性规格.....	147
6.4.4 使用说明.....	147
6.4.5 维护窗口参数配置.....	152
6.4.6 运维 TIPS.....	153
<b>7 Foreign Data Wrapper.....</b>	<b>160</b>
7.1 file_fdw.....	160
<b>8 动态数据脱敏.....</b>	<b>162</b>

# 1 物化视图

物化视图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。物化视图常用的操作包括创建、查询、删除和刷新。

根据创建规则，物化视图分为全量物化视图和增量物化视图。全量物化视图只支持全量刷新；增量物化视图支持全量刷新和增量刷新两种方式。全量刷新会将基表中的数据全部重新刷入物化视图中，而增量刷新只会将两次刷新间隔期间的基表产生的增量数据刷入物化视图中。

目前Ustore引擎不支持创建、使用物化视图。

## 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刷新，而不支持做增量刷新。创建全量物化视图语法和CREATE TABLE AS语法一致（详情请参见《开发指南》中的“SQL参考 > SQL语法 > CREATE TABLE AS”章节），不支持对全量物化视图指定NodeGroup创建。全量物化视图的创建继承GTM-Free的相关约束。

### 1.1.2 支持和约束

#### 支持场景

- 通常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。

- Ustore引擎不支持全量物化视图的创建和使用。

## 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。

## 1.1.3 使用

### 语法格式

- 创建全量物化视图  
CREATE MATERIALIZED VIEW view\_name AS query;
- 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;

### 参数说明

- **view\_name**  
要创建的物化视图名。  
取值范围：字符串，要符合标识符的命名规范。
- **AS query**  
一个SELECT VALUES命令或者一个运行预备好的SELECT或VALUES查询的EXECUTE命令。

### 示例

```
-- 修改表的默认类型
gaussdb=# set enable_default_ustore_table=off;

-- 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，删除表
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE t1;
DROP TABLE
```

## 1.2 增量物化视图

### 1.2.1 概述

增量物化视图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的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 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。
- 对物化视图进行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，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。
- Ustore引擎不支持增量物化视图的创建和使用。

#### 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。

## 1.2.3 使用

### 语法格式

- **创建增量物化视图**  
CREATE INCREMENTAL MATERIALIZED VIEW view\_name AS query;
- **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;

### 参数说明

- **view\_name**  
要创建的物化视图名。  
取值范围：字符串，要符合标识符的命名规范。
- **AS query**  
一个SELECT VALUES命令或者一个运行预备好的SELECT或VALUES查询的EXECUTE命令。

### 示例

```
-- 修改表的默认类型
gaussdb=# SET enable_default_ustore_table=off;

-- 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, 删除表
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE t1;
DROP TABLE
```

# 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

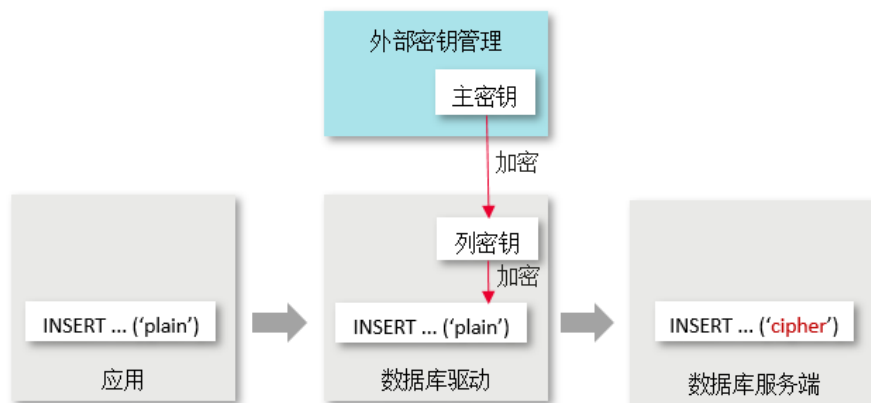
全密态数据库使用多级加密模型，加密模型中涉及3个对象：数据、列密钥和主密钥，以下是对3个对象的介绍：

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

#### 说明

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

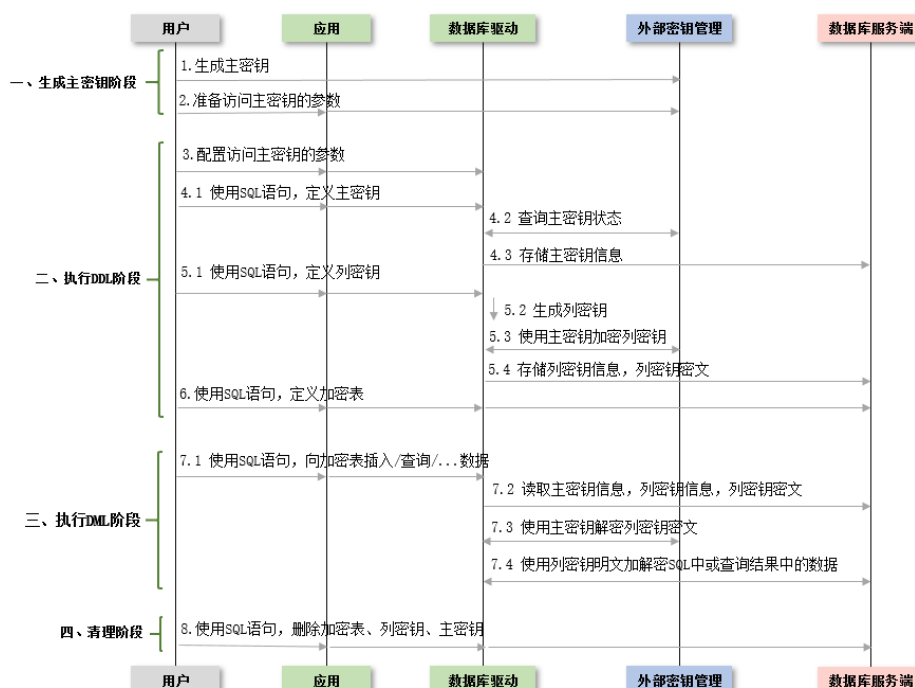
在使用全密态数据库的过程中，主要流程包括如下四个阶段，本节（2.1）介绍整体流程。[使用gsq操作密态数据库](#)、[使用JDBC操作密态数据库](#)章节介绍详细使用流程。

**一、生成主密钥阶段：**首先，用户需在华为云密钥服务中生成主密钥。生成主密钥后，需准备访问主密钥的参数，以供数据库使用。

**二、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。

**三、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。

**四、清理阶段：**依次删除加密表、列密钥和主密钥。



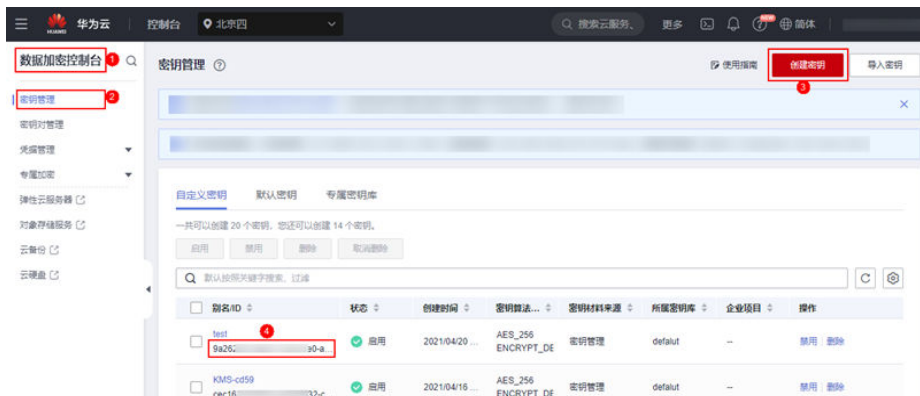
## 生成主密钥阶段

首次使用密态数据库时，需使用外部密钥管理服务生成至少一个主密钥，生成方式如下：

- 华为公有云场景
  - a. 登录账号：进入华为云官网，注册并登录账号。
  - b. 创建新用户：搜索并进入"身份认证服务"，在"用户"中，通过"创建用户"按钮创建一个IAM用户，设置IAM密码，并为IAM用户设置使用"数据加密服务"的权限。



- c. 登录新用户：重新回到登录页面，选择"IAM用户"登录方式，使用新上一步创建的IAM用户进行登录。后续操作均由该IAM用户完成。
- d. 创建主密钥：选择"密钥管理"功能，并通过"创建密钥"按钮创建至少1个密钥，即主密钥。
- e. 记住主密钥ID：成功创建主密钥后，每个主密钥都有1个密钥ID。在后续使用密态数据过程中，需配置主密钥ID，数据库驱动会通过Restful接口访问该主密钥。



在生成主密钥后，需为数据驱动准备访问主密钥的参数，比如IAM用户名、项目ID等参数。华为云支持两种身份认证方式，两种方式需要的参数个数与参数类型不同，选择其中一种方式即可。下述步骤介绍如何获取这些参数：

- 方式一 aksk认证
  - a. AK、SK：首先登录华为云“控制台”，单击右上角用户名，进入“我的凭证”，选择“访问密钥”，通过“新增访问密钥”创建AK与SK，创建完成后下载密钥（即AK与SK）。



- b. 项目ID：在华为云控制台中，单击右上角用户名，并进入“我的凭证”，单击“API凭证”即可找到“项目ID”。



- c. KMS服务器地址：<https://kms.项目.myhuaweicloud.com/v1.0/项目ID/kms>。

- 方式二 账号密码认证

- a. IAM用户名、账号名、项目、项目ID：在华为云控制台中，单击右上角用户名，并进入“我的凭证”，可看到下图所示页面，该页面可获取4个参数：IAM用户名、账号名、项目、项目ID。



- b. IAM服务器地址：<https://iam.项目.myhuaweicloud.com/v3/auth/tokens>。
- c. IAM用户密码：IAM用户名对应的密码。
- d. KMS服务器地址：<https://kms.项目.myhuaweicloud.com/v1.0/项目ID/kms>。

## 2.2 使用 gsql 操作密态数据库

### 执行 SQL 语句

执行本节的SQL语句前，请确保已提前生成主密钥，并确认访问主密钥的参数。

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 通过元命令设置访问主密钥的参数

注意：从keyType字符串开始，不要添加换行，不要添加空格，否则gsql工具无法识别完整参数。

华为云支持两种认证方式，两种认证方式的参数个数与参数类型不同，选择其中一种方式即可。

- **认证方式一 aksk认证**

```
gaussdb=# \key_info keyType=huawei_kms,kmsProjectId={项目ID},ak={AK},sk={SK}
```

参数获取：生成主密钥阶段介绍了如何获取相关参数：项目ID、AK、SK。

示例：\key\_info

```
keyType=huawei_kms,kmsProjectId=0b59929e8100268a2f22c01429802728,ak=XMAUMJY*****DFWLQW,sk=ga6rO8lx1Q4uB*****2gf80mulzUX
```

- **认证方式二 账号密码认证**

```
gaussdb=# \key_info keyType=huawei_kms,iamUrl={IAM服务器地址},iamUser={IAM用户名},iamPassword={IAM用户密码},iamDomain={账号名},kmsProject={项目}
```

参数获取：生成主密钥阶段介绍了如何获取相关参数：IAM服务器地址、IAM用户名、IAM用户密码、账号名、项目。

示例：\key\_info keyType=huawei\_kms,iamUrl=https://iam.example.com/v3/auth/

```
tokens,iamUser=test,iamPassword=*****,iamDomain=test_account,kmsProject=xxx
```

### 步骤3 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4 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5 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6 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 = '8000000001111111' 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7 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

### 配置 JDBC 驱动

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”及“应用程序开发教程 > 兼容性参考 > JDBC兼容性包”章节。

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

- 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。本章的Java代码示例默认使用gscejdbc.jar包。
- gaussdbjdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包没有打包密态数据库需要加载的加解密相关的依赖库，需要手动配置LD\_LIBRARY\_PATH环境变量。
- gaussdbjdbc-JRE7.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，在JDK1.7环境使用gaussdbjdbc-JRE7.jar包，此驱动包没有打包密态数据库需要加载的加解密相关的依赖库，需要手动配置LD\_LIBRARY\_PATH环境变量。

### 📖 说明

其他兼容性：密态数据库支持的JDBC驱动包还支持其他兼容性驱动包gsjdbc4.jar、opengaussjdbc.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”。

## 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。

- 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复制到该路径下，并在开启密态功能连接数据库的时候加载。
- 使用gaussdbjdbc.jar、gaussdbjdbc-JRE7.jar、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。

### ⚠️ 注意

全密态场景使用JDBC驱动包时需要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。

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会自动复制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。

## 执行 SQL 语句

执行本节的SQL语句前，请确保已提前生成主密钥，并确认访问主密钥的参数。

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。其中不设置enable\_ce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”。
// gaussdbjdbc.jar: 主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”。
// gaussdbjdbc-JRE7.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();

        // 设置访问主密钥的参数
        // 此处介绍2种方式，选择其中1种方式即可：
        // 认证方式一 aksk认证（生成主密钥阶段介绍了如何获取相关参数：项目ID、AK、SK）
        conn.setClientInfo("key_info", "keyType=huawei_kms, kmsProjectId={项目ID}, ak={AK}, sk={SK}");

        /* 示例:
        conn.setClientInfo("key_info",
            "keyType=huawei_kms,kmsProjectId=0b59929e8100268a2f22c01429802728," +
            "ak=XMAUMJY*****DFWLQW, sk=ga6rO8lx1Q4uB*****2gf80mulzUX,");
        */
        // 认证方式二 账号密码认证（生成主密钥阶段介绍了如何获取相关参数：IAM服务器地址、IAM用户名、IAM用户密码、账号名、项目）
        conn.setClientInfo("key_info", "keyType=huawei_kms," +
            "iamUrl={IAM服务器地址}," +
            "iamUser={IAM用户名}," +
            "iamPassword={IAM用户密码}," +
            "iamDomain={账号名}," +
            "kmsProject={项目}");
        /* 示例:
        conn.setClientInfo("key_info", "keyType=huawei_kms," +
            "iamUrl=https://iam.example.com/v3/auth/tokens," +
            "iamUser=test," +
            "iamPassword=*****," +
            "iamDomain=test_account," +
            "kmsProject=xxx");
        */
        // 定义主密钥：cmk1为主密钥名字，可自行取名
        // 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
        int rc = stmt.executeUpdate("CREATE CLIENT MASTER KEY lmgCMK1 WITH ( KEY_STORE = huawei_kms , KEY_PATH = '{KMS服务器地址}/{密钥ID}', ALGORITHM = AES_256);");

        /*
        KEY_PATH示例: https://kms.cn-north-4.myhuaweicloud.com/v1.0/0b59929e8100268a2f22c01429802728/kms/9a262917-8b31-41af-a1e0-a53235f32de9
        解释：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
        提示：KEY_PATH格式请参考：《开发指南》中“SQL参考 > SQL语法 > CREATE CLIENT
        */
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
MASTER KEY” 章节
*/
// 定义列加密密钥：列密钥由上一步创建的主密钥加密。详细语法参考：《开发指南》中“SQL参考
> SQL语法 > CREATE COLUMN ENCRYPTION KEY” 章节
int rc2 = stmt.executeUpdate("CREATE COLUMN ENCRYPTION KEY ImgCEK1 WITH VALUES
(CLIENT_MASTER_KEY = ImgCMK1, ALGORITHM = AES_256_GCM);");
// 定义加密表
int rc3 = stmt.executeUpdate("CREATE TABLE creditcard_info (id_number int, name varchar(50)
encrypted with (column_encryption_key = ImgCEK1, encryption_type = DETERMINISTIC),credit_card
varchar(19) encrypted with (column_encryption_key = ImgCEK1, encryption_type =
DETERMINISTIC));");
// 插入数据
int rc4 = stmt.executeUpdate("INSERT INTO creditcard_info VALUES
(1,'joe','6217986500001288393');");
// 查询加密表
ResultSet rs = null;
rs = stmt.executeQuery("select * from creditcard_info where name = 'joe';");
// 删除加密表
int rc5 = stmt.executeUpdate("DROP TABLE IF EXISTS creditcard_info;");
// 删除列加密密钥
int rc6 = stmt.executeUpdate("DROP COLUMN ENCRYPTION KEY IF EXISTS ImgCEK1;");
// 删除客户端主密钥
int rc7 = stmt.executeUpdate("DROP CLIENT MASTER KEY IF EXISTS ImgCMK1;");
// 关闭语句对象
stmt.close();
// 关闭连接
conn.close();
} catch (Exception e) {
    e.printStackTrace();
    return;
}
}
```

### 📖 说明

- 使用JDBC操作密态数据库时，一个数据库连接对象对应一个线程，否则，不同线程变更更可能导致冲突。
- 使用JDBC操作密态数据库时，不同connection对密态配置数据有变更，由客户端调用isvalid方法保证connection能够持有变更后的密态配置数据，此时需要保证参数refreshClientEncryption为1(默认值为1)，在单客户端操作密态数据场景下，refreshClientEncryption参数可以设置为0。
- 调用isValid方法刷新缓存示例

```
// 创建连接conn1
Connection conn1 = DriverManager.getConnection("url","user","password");
// 在另外一个连接conn2中创建客户端主密钥
...
// conn1通过调用isValid刷新缓存，刷新conn1密钥缓存
try {
    if (!conn1.isValid(60)) {
        System.out.println("isValid Failed for connection 1");
    }
} catch (SQLException e) {
    e.printStackTrace();
    return null;
}
```

## 执行密态等值密文解密

数据库连接接口PgConnection类型新增解密接口，可以对全密态数据库的密态等值密文进行解密。解密后返回其明文值，通过schema.table.column找到解文对应的密文列并返回其原始数据类型。

表 2-1 新增 com.huawei.gaussdb.jdbc.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 com.huawei.gaussdb.jdbc.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 com.huawei.gaussdb.jdbc.jdbc.PgConnection;
```

```
import com.huawei.gaussdb.jdbc.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 前向兼容与安全增强

### 前向兼容

在上文中，支持通过key\_info设置访问外部密钥管理的参数：

1. 使用gsql时，通过元命令\key\_info xxx设置。
2. 使用JDBC时，通过连接参数conn.setProperty(“key\_info”，“xxx”)设置。

为保持前向兼容，还支持通过环境变量等方式设置访问主密钥的参数。

### ⚠ 注意

第一次配置使用密态数据库时，可忽略下述方法。如果以前使用下述方法配置密态数据库，建议改用 'key\_info' 配置。

使用系统级环境变量配置的方式如下：

```
export HUAWEI_KMS_INFO='iamUrl=https://iam.{项目}.myhuaweicloud.com/v3/auth/tokens,iamUser={IAM用户名},iamPassword={IAM用户密码},iamDomain={账号名},kmsProject={项目}'  
# 该方法中操作系统日志可能会记录环境变量中的敏感信息，使用过程中注意及时清理。
```

还可通过标准库接口设置进程级环境变量，不同语言设置方法如下：

1. C/C++  
setenv("HIS\_KMS\_INFO", "xxx");
2. GO  
os.Setenv("HIS\_KMS\_INFO", "xxx");

## 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。本节介绍如何下载与配置CA证书。

### 配置方法

在key\_info参数的中，增加证书相关参数即可。

- 使用gsqldb时

```
gaussdb=# \key_info keyType=huawei_kms,iamUrl=https://iam.example.com/v3/auth/tokens,iamUser={IAM用户名},iamPassword={IAM用户密码},iamDomain={账号名},kmsProject={项目},iamCaCert=/路径/IAM的CA证书文件,kmsCaCert=/路径/KMS的CA证书文件
```

```
gaussdb=# \key_info keyType=huawei_kms,kmsProjectId={项目ID},ak={AK},sk={SK},kmsCaCert=/路径/KMS的CA证书文件
```

- 使用JDBC时

```
conn.setProperty("key_info", "keyType=huawei_kms," +  
"iamUrl=https://iam.{example}.com/v3/auth/tokens," +  
"iamUser={IAM用户名}," +  
"iamPassword={IAM用户密码}," +  
"iamDomain={账号名}," +  
"kmsProject={项目}," +  
"iamCaCert=/路径/IAM的CA证书文件," +  
"kmsCaCert=/路径/KMS的CA证书文件");
```

```
conn.setProperty("key_info", "keyType=huawei_kms, kmsProjectId={项目ID}, ak={AK}, sk={SK},  
kmsCaCert=/路径/KMS的CA证书文件");
```

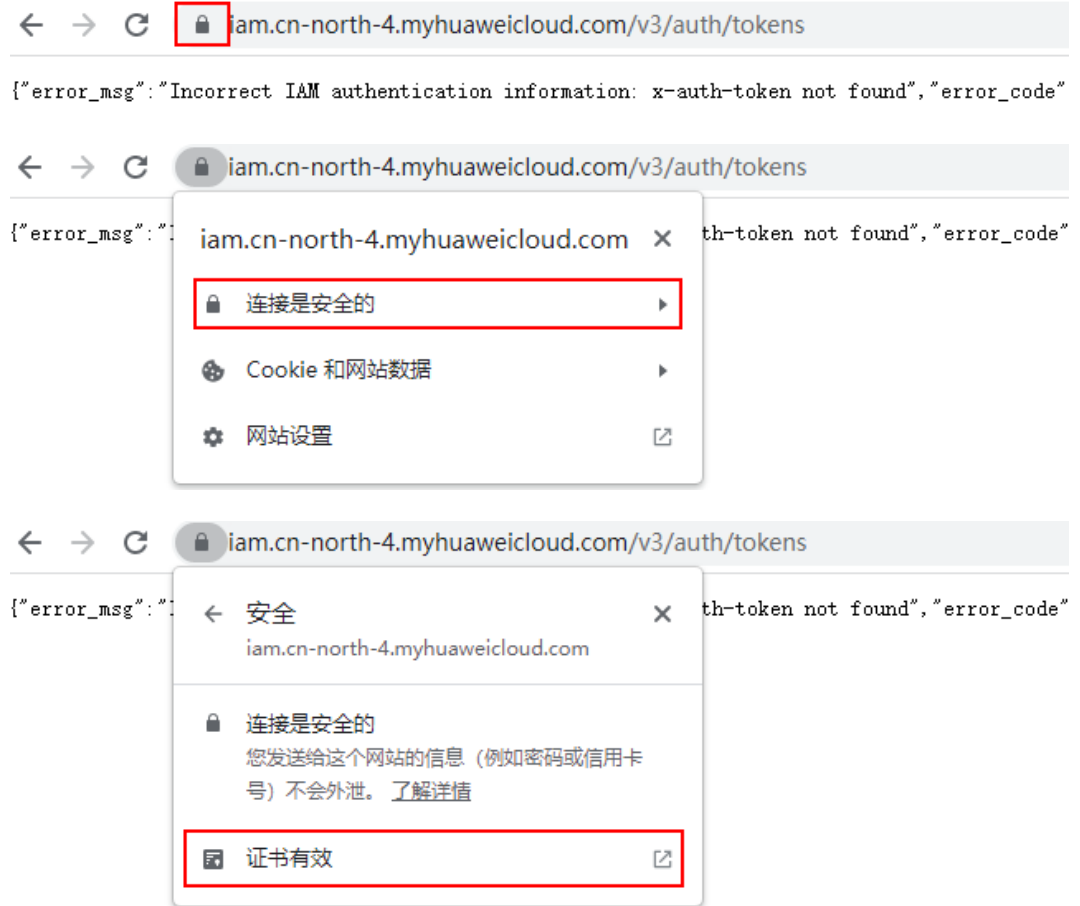
### 获取证书

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

由于使用restful接口访问密钥服务，当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，浏览器也早已提前自动下载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服务器地址和KMS服务器地址，地址获取方式参见：[生成主密钥阶段](#)。
- 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，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，详细步骤请参见[使用gsq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”所包含的查询是在执行过程编译，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中为变量授予默认值，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，需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（byteawithoutorderwithqualcol、byteawithoutordercol、\_byteawithoutorderwithqualcol、\_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 设置账本数据库

## 3.1 账本数据库概述

### 背景信息

账本数据库融合了区块链思想，将用户操作记录至两种历史表：用户历史表和全局区块表中。当用户创建防篡改用户表时，系统将自动为该表添加一个hash列来保存每行数据的hash摘要信息，同时在blockchain模式下会创建一张用户历史表来记录对应用户表中每条数据的变更行为；而用户对防篡改用户表的每一次修改行为将被记录到全局区块表中。由于历史表具有只可追加不可修改的特点，因此历史表记录串联起来便形成了用户对防篡改用户表的修改历史。

用户历史表命名和结构如下：

表 3-1 用户历史表 blockchain.<schemaname>\_<tablename>\_hist 所包含的字段

字段名	类型	描述
rec_num	bigint	行级修改操作在历史表中的执行序号。
hash_ins	hash16	INSERT或UPDATE操作插入的数据行的hash值。
hash_del	hash16	DELETE或UPDATE操作删除数据行的hash值。
pre_hash	hash32	当前用户历史表的数据整体摘要。

表 3-2 hash\_ins 与 hash\_del 场景对应关系

-	hash_ins	hash_del
INSERT	(√) 插入行的hash值。	空

-	hash_ins	hash_del
DELETE	空	(√) 删除行的hash值。
UPDATE	(√) 新插入数据的hash值。	(√) 删除前该行的hash值。

## 操作步骤

### 步骤1 创建防篡改模式。

例如，创建防篡改模式ledgernsp。

```
gaussdb=# CREATE SCHEMA ledgernsp WITH BLOCKCHAIN;
```

#### 📖 说明

如果需要创建防篡改模式或更改普通模式为防篡改模式，则需设置enable\_ledger参数为on。  
enable\_ledger默认参数为off。

### 步骤2 在防篡改模式下创建防篡改用户表。

例如，创建防篡改用户表ledgernsp.usertable。

```
gaussdb=# CREATE TABLE ledgernsp.usertable(id int, name text);
```

查看防篡改用户表结构及其对应的用户历史表结构。

```
gaussdb=# \d+ ledgernsp.usertable;
gaussdb=# \d+ blockchain.ledgernsp_usertable_hist;
```

执行结果如下：

```
gaussdb=# \d+ ledgernsp.usertable;
          Table "ledgernsp.usertable"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id     | integer |          | plain   |              |
 name   | text    |          | extended |              |
 hash_69dd43 | hash16 |          | plain   |              |
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no
History table name: ledgernsp_usertable_hist

gaussdb=# \d+ blockchain.ledgernsp_usertable_hist;
          Table "blockchain.ledgernsp_usertable_hist"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 rec_num | bigint |          | plain   |              |
 hash_ins | hash16 |          | plain   |              |
 hash_del | hash16 |          | plain   |              |
 pre_hash | hash32 |          | plain   |              |
Indexes:
 "gs_hist_69dd43_index" PRIMARY KEY, btree (rec_num int4_ops) TABLESPACE pg_default
Has OIDs: no
Distribute By: HASH(rec_num)
Location Nodes: ALL DATANODES
Options: internal_mask=263
```

### 说明

1. 防篡改模式下仅行存表为防篡改表，临时表、外表、unlog表及非行存表均无防篡改属性。
2. 防篡改表在创建时会自动增加一个用于校验的系统列，所以防篡改表单表最大列数为1599。

### 步骤3 修改防篡改用户表数据。

例如，对防篡改用户表执行INSERT、UPDATE、DELETE操作。

```
gaussdb=# INSERT INTO ledgernsp.usertable VALUES(1, 'alex'), (2, 'bob'), (3, 'peter');
INSERT 0 3
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name |      hash_69dd43
-----+-----+-----
  1 | alex | 1f2e543c580cb8c5
  2 | bob  | 8fcd74a8a6a4b484
  3 | peter| f51b4b1b12d0354b
(3 rows)

gaussdb=# UPDATE ledgernsp.usertable SET name = 'bob2' WHERE id = 2;
UPDATE 1
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name |      hash_69dd43
-----+-----+-----
  1 | alex | 1f2e543c580cb8c5
  2 | bob2 | 437761affbb7c605
  3 | peter| f51b4b1b12d0354b
(3 rows)

gaussdb=# DELETE FROM ledgernsp.usertable WHERE id = 3;
DELETE 1
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name |      hash_69dd43
-----+-----+-----
  1 | alex | 1f2e543c580cb8c5
  2 | bob2 | 437761affbb7c605
(2 rows)
```

### 步骤4 删除表和模式。

若要执行其他账本数据库章节的示例，请在执行完之后再执行当前步骤，否则请直接执行当前步骤。

```
gaussdb=# DROP TABLE ledgernsp.usertable;
DROP TABLE
gaussdb=# DROP SCHEMA ledgernsp;
DROP SCHEMA
```

----结束

## 3.2 查看账本历史操作记录

### 前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。
- 数据库各个CN节点全局区块表记录单独记录，全局区块表只能记录连接到当前CN执行的SQL操作。

## 背景信息

- 只有拥有AUDITADMIN属性的用户才可以查看账本历史操作记录。有关数据库用户及创建用户的办法请参见《开发指南》中“数据库安全 > 用户及权限 > 用户”章节。
- 查询全局区块表命令是直接查询gs\_global\_chain表，操作为：

```
SELECT * FROM gs_global_chain;
```

该表有10个字段，每个字段的含义请参见《开发指南》中“系统表和系统视图 > 系统表 > GS\_GLOBAL\_CHAIN”章节。
- 查询用户历史表的命令是直接查询BLOCKCHAIN模式下的用户历史表，操作为：例如用户表所在的模式为ledgernsp，表名为usertable，则对应的用户历史表名为blockchain.ledgernsp\_usertable\_hist：

```
SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

用户历史表有4个字段，每个字段的含义请参见表3-1。

### 说明

用户历史表的表名一般为blockchain.<schemaname>\_<tablename>\_hist形式。当防篡改用户表模式名或者表名过长导致前述方式生成的表名超出表名长度限制，则会采用blockchain.<schema\_oid>\_<table\_oid>\_hist的方式命名。

## 操作步骤

### 步骤1 查询全局区块表记录。

```
gaussdb=# SELECT * FROM gs_global_chain;
```

查询结果如下：

blocknum	dbname	username	starttime	releid	relnsp	relname	relhash
0	testdb	omm	2021-04-14 07:00:46.32757+08	16393	ledgernsp	usertable	a41714001181a294   6b5624e039e8aee36bf3e8295c75b40   insert into ledge rnspp.usertable values(1, 'alex'), (2, 'bob'), (3, 'peter');
1	testdb	omm	2021-04-14 07:01:19.767799+08	16393	ledgernsp	usertable	b3a9ed0755131181   328b48c4370faed930937869783c23e0   update ledgernsp. usertable set name = 'bob2' where id = 2;
2	testdb	omm	2021-04-14 07:01:29.896148+08	16393	ledgernsp	usertable	0ae4b4e4ed2fcab5   aa8f0a236357cac4e5bc1648a739f2ef   delete from ledge rnspp.usertable where id = 3;

该结果表明，用户omm连续执行了三条DML命令，包括INSERT、UPDATE和DELETE操作。

### 步骤2 查询历史表记录。

```
gaussdb=# SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

查询结果如下：

rec_num	hash_ins	hash_del	pre_hash
0	1f2e543c580cb8c5		e1b664970d925d09caa295abd38d9b35
1	8fcd74a8a6a4b484		dad3ed8939a141bf3682043891776b67
2	f51b4b1b12d0354b		53eb887fc7c4302402343c8914e43c69
3	437761affbb7c605	8fcd74a8a6a4b484	c2868c5b49550801d0dbbbaa77a83a10
4		f51b4b1b12d0354b	9c512619f6ffef38c098477933499fe3

(5 rows)

查询结果显示，用户omm对ledgernsp.usertable表插入了3条数据，更新了1条数据，随后删除了1行数据，最后剩余2行数据，hash值分别为1f2e543c580cb8c5和437761affbb7c605。

### 步骤3 查询用户表数据及校验列。

```
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable;
```

查询结果如下：

```
id | name | hash_69dd43
---+-----+-----
 1 | alex | 1f2e543c580cb8c5
 2 | bob2 | 437761affbb7c605
(2 rows)
```

查询结果显示，用户表中剩余2条数据，与2中的记录一致。

----结束

## 3.3 校验账本数据一致性

### 前提条件

数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。

### 背景信息

- 账本数据库校验功能目前提供两种校验接口，分别为：ledger\_hist\_check(text, text)和ledger\_gchain\_check(text, text)。普通用户调用校验接口，仅能校验自己有权访问的表。
- 校验防篡改用户表和用户历史表的接口为pg\_catalog.ledger\_hist\_check，操作为：  

```
SELECT pg_catalog.ledger_hist_check(schema_name text, table_name text);
```

如果校验通过，函数返回t，反之则提示失败原因并返回f。
- 校验防篡改用户表、用户历史表和全局区块表三者是否一致的接口为pg\_catalog.ledger\_gchain\_check，操作为：  

```
SELECT pg_catalog.ledger_gchain_check(schema_name text, table_name text);
```

如果校验通过，函数返回t，反之则提示失败原因并返回f。

### 操作步骤

#### 步骤1 校验防篡改用户表ledgernsp.usertable与其对应的历史表是否一致。

```
gaussdb=# SELECT pg_catalog.ledger_hist_check('ledgernsp', 'usertable');
```

查询结果如下：

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

该结果表明防篡改用户表和用户历史表中记录的结果能够一一对应，保持一致。

#### 步骤2 查询防篡改用户表ledgernsp.usertable与其对应的历史表以及全局区块表中关于该表的记录是否一致。

```
gaussdb=# SELECT pg_catalog.ledger_gchain_check('ledgernsp', 'usertable');
```



查询结果如下:

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

查询结果显示, 上述三表中关于ledgernsp.usertable的记录保持一致, 未发生篡改行为。

----结束

## 3.4 归档账本数据库

### 前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行, 并且对防篡改数据库执行了一系列增、删、改等操作, 保证在查询时段内有账本操作记录结果产生。
- 数据库已经正确配置审计文件的存储路径audit\_directory。

### 背景信息

- 账本数据库归档功能目前提供两种校验接口, 分别为: ledger\_hist\_archive(text, text)和ledger\_gchain\_archive(text, text)。账本数据库接口仅审计管理员可以调用。
- 归档用户历史表的接口为pg\_catalog.ledger\_hist\_archive, 表示归档当前DN的用户历史表数据。执行操作为:  

```
SELECT pg_catalog.ledger_hist_archive(schema_name text,table_name text);
```

如果归档成功, 函数返回t, 反之则提示失败原因并返回f。
- 归档全局区块表的接口为pg\_catalog.ledger\_gchain\_archive, 表示归档当前CN的全局历史表数据。执行操作为:  

```
SELECT pg_catalog.ledger_gchain_archive();
```

如果归档成功, 函数返回t, 反之则提示失败原因并返回f。

### 操作步骤

**步骤1** 使用EXECUTE DIRECT对某个DN节点进行归档操作。

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'select pg_catalog.ledger_hist_archive("ledgernsp", "usertable");'
```

查询结果如下:

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

用户历史表将归档为一条数据:

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'SELECT * FROM blockchain.ledgernsp_usertable_hist;'
```

rec_num	hash_ins	hash_del	pre_hash
3	e78e75b00d396899	8fcd74a8a6a4b484	fd61cb772033da297d10c4e658e898d7

(1 row)

该结果表明datanode1节点用户历史表导出成功。

## 步骤2 连接CN执行全局区块表导出操作。

```
gaussdb=# SELECT pg_catalog.ledger_gchain_archive();
```

查询结果如下：

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

全局历史表将以用户表为单位归档为N（用户表数量）条数据：

```
gaussdb=# SELECT * FROM gs_global_chain;
blocknum | dbname | username | starttime          | relid | relnsp | relname | relhash
| globalhash | txcommand
-----+-----+-----+-----+-----+-----+-----+-----
1 | testdb | libc    | 2021-05-10 19:59:38.619472+08 | 16388 | ledgernsp | usertable |
57c101076694b415 | be82f98ee68b2bc4e375f69209345406 | Archived.
(1 row)
```

该结果表明，当前coordinator节点全局区块表导出成功。

----结束

## 3.5 修复账本数据库

### 前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。

### 背景信息

- 当前的账本数据库机制为：全局区块表存储在CN端，各个CN数据独立。用户历史表存储在DN端，历史表记录的数据为所在DN防篡改表的数据变化。因此，在触发数据重分布时，可能导致防篡改表和用户历史表数据不一致，此时需要使用 `ledger_hist_repair(text, text)` 接口对指定DN节点的用户历史表进行修复，修复后当前DN节点调用历史表校验接口结果为true。在CN剔除、修复的场景下，可能导致全局区块表数据丢失或者与用户历史表不一致，此时需要使用 `ledger_gchain_repair(text, text)` 接口对整个集群范围内的全局区块表进行修复，修复后调用全局区块表校验接口结果为true。
- 修复用户历史表的接口为 `pg_catalog.ledger_hist_repair`，操作为：

```
SELECT pg_catalog.ledger_hist_repair(schema_name text,table_name text);
```

如果修复成功，函数返回修复过程中用户历史表hash的增量。  
注：对用户表执行闪回DROP时，可使用该函数恢复用户表和用户历史表名称，请参见[恢复用户表和用户历史表名称](#)。
- 修复全局区块表的接口为 `pg_catalog.ledger_gchain_repair`，操作为：

```
SELECT pg_catalog.ledger_gchain_repair(schema_name text,table_name text);
```

如果修复成功，函数返回修复过程中全局区块表中指定表的hash总和。

### 恢复用户表数据和全局区块表数据

以omm用户为例进行操作，步骤如下。

**步骤1** 以操作系统用户omm登录数据库主节点。

**步骤2** 使用EXECUTE DIRECT对某个DN节点进行历史表修复操作。

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'select pg_catalog.ledger_hist_repair("ledgernsp",  
"usertable");';
```

查询结果如下：

```
ledger_hist_repair  
-----  
84e8bfc3b974e9cf  
(1 row)
```

该结果表明datanode1节点用户历史表修复成功，修复造成的用户历史表hash增量为84e8bfc3b974e9cf。

**步骤3** 连接CN执行全局区块表修复操作。

```
gaussdb=# SELECT pg_catalog.ledger_gchain_repair('ledgernsp', 'usertable');
```

查询结果如下：

```
ledger_gchain_repair  
-----  
a41714001181a294  
(1 row)
```

该结果表明，当前集群全局区块表修复成功，且向当前CN节点插入一条修复数据，其hash值为a41714001181a294。

----结束

## 恢复用户表和用户历史表名称

已通过enable\_recyclebin参数和recyclebin\_retention\_time参数开启闪回DROP功能，恢复用户表和用户历史表名称。示例如下：

- DROP用户表，对用户表执行闪回DROP。使用ledger\_hist\_repair对用户表、用户历史表进行表名恢复。

-- 对用户表执行闪回drop，使用ledger\_hist\_repair对用户历史表进行表名恢复。

```
gaussdb=# CREATE TABLE ledgernsp.tab2(a int, b text);
```

NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'rec\_num' as the distribution column by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

```
CREATE TABLE
```

```
gaussdb=# DROP TABLE ledgernsp.tab2;
```

```
DROP TABLE
```

```
gaussdb=# SELECT rcyrelid, rcyname, rcyoriginname FROM gs_recyclebin;
```

```
rcyrelid | rcyname | rcyoriginname  
-----+-----+-----  
32838 | BIN$39B523388046$55C8400==$0 | tab2  
32846 | BIN$39B52338804E$55C90E8==$0 | gs_hist_tab2_index  
32843 | BIN$39B52338804B$55C96A0==$0 | ledgernsp_tab2_hist  
32841 | BIN$39B523388049$55C9EE0==$0 | pg_toast_32838  
(4 rows)
```

-- 对用户表执行闪回drop。

```
gaussdb=# TIMECAPSULE TABLE ledgernsp.tab2 TO BEFORE DROP;
```

```
TimeCapsule Table
```

-- 使用ledger\_hist\_repair恢复用户历史表表名。

```
gaussdb=# SELECT ledger_hist_repair('ledgernsp', 'tab2');
```

```
ledger_hist_repair
```

```
-----  
0000000000000000  
(1 row)
```

```

gaussdb=# \d+ ledgermsp.tab2;
          Table "ledgermsp.tab2"
  Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a      | integer |          | plain   |              |
 b      | text   |          | extended |              |
hash_1d2d14 | hash16 |          | plain   |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
History table name: ledgermsp_tab2_hist

-- 对用户表执行闪回drop, 使用ledger_hist_repair对用户表进行表名恢复。
gaussdb=# CREATE TABLE ledgermsp.tab3(a int, b text);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'rec_num' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# DROP TABLE ledgermsp.tab3;
DROP TABLE
gaussdb=# SELECT rcyrelid, rcyname, rcyoriginname FROM gs_recyclebin;
 rcyrelid |      rcyname      | rcyoriginname
-----+-----+-----
 32952 | BIN$80B6233880B8$FECFF98==$0 | tab3
 32960 | BIN$80B6233880C0$FED0C98==$0 | gs_hist_tab3_index
 32957 | BIN$80B6233880BD$FED1250==$0 | ledgermsp_tab3_hist
 32955 | BIN$80B6233880BB$FED1A00==$0 | pg_toast_32952
(4 rows)
-- 对用户历史表执行闪回drop。
gaussdb=# TIMECAPSULE TABLE blockchain.ledgermsp_tab3_hist TO BEFORE DROP;
TimeCapsule Table
-- 拿到回收站中用户表对应的rcyname, 使用ledger_hist_repair恢复用户表表名。
gaussdb=# SELECT ledger_hist_repair('ledgermsp', 'BIN$80B6233880B8$FECFF98==$0');
 ledger_hist_repair
-----
0000000000000000
(1 row)

gaussdb=# \d+ ledgermsp.tab3;
          Table "ledgermsp.tab3"
  Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a      | integer |          | plain   |              |
 b      | text   |          | extended |              |
hash_7a0c87 | hash16 |          | plain   |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
History table name: ledgermsp_tab3_hist

-- 删除表。
gaussdb=# DROP TABLE ledgermsp.tab2 PURGE;
DROP TABLE
gaussdb=# DROP TABLE ledgermsp.tab3 PURGE;
DROP TABLE

```

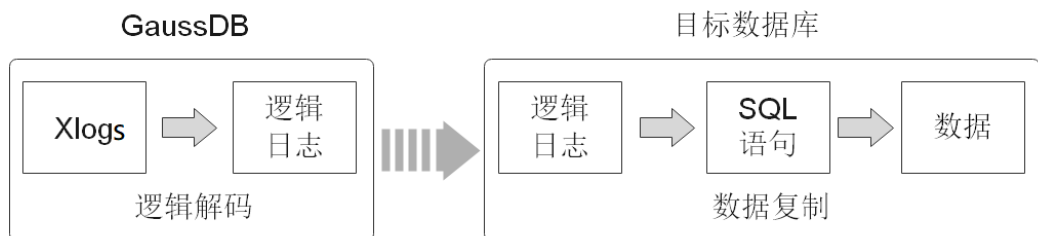
# 4 逻辑复制

GaussDB对数据复制能力的支持情况为：

支持通过数据迁移工具定期向异构数据库（如Oracle数据库等）进行数据同步，不具备实时数据复制能力，因此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。

基于上述情况，GaussDB提供了逻辑解码功能，通过反解xlog的方式生成逻辑日志。目标数据库解析逻辑日志以实时进行数据复制。具体如图4-1所示。逻辑复制降低了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，支持目标库进行数据同步期间的数据可读写，数据同步时延低。

图 4-1 逻辑复制



逻辑复制由两部分组成：逻辑解码和数据复制。逻辑解码会输出以事务为单位组织的逻辑日志。业务或数据库中间件将会对逻辑日志进行解析并最终实现数据复制。GaussDB当前只提供逻辑解码功能，因此本章节只涉及逻辑解码的说明。

## 4.1 逻辑解码

### 4.1.1 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可以使用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，这些更改可以在其他数据库上以它们在原数据库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CN或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>=每个节点所需的（物理流复制槽数+备份槽数+逻辑复制槽数）。

### 📖 说明

关于逻辑复制槽数，请按如下规则考虑：

- 一个逻辑复制槽只能解码一个Database的修改，如果需要解码多个Database，则需要创建多个逻辑复制槽。
- 如果需要多路逻辑复制同步给多个目标数据库，在源端数据库需要创建多个逻辑复制槽，每个逻辑复制槽对应一条逻辑复制链路。
- 同一个CN节点上最多支持同时开启10个分布式CN强一致解码任务，同一个DN节点上最多支持同时开启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，相关操作请参见[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。
- 仅限初始用户或拥有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

- 不支持数据页复制的DML解码。
- 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。
- 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）。

- 非M兼容库的浮点型数据、M兼容库的不带标度的float类型，解码结果显示精度extra\_float\_digits配置为3（该参数的含义参见GUC参数float\_shortest\_precision的具体信息）。
- 不支持解码PUBLIC SCHEMA的DDL操作。
- 支持M-Compatibility模式数据库的DML、DDL逻辑解码，解码数据类型如下：
  - 整型：TINYINT(M)、SMALLINT(M)、MEDIUMINT(M)、BIGINT(M)、INT(M)/INTEGER、BOOL/BOOLEAN。
  - 浮点型：FLOAT(M,D)、DOUBLE(M,D)。
  - 定点型：DECIMAL(M,D)、NUMERIC(M,N)。
  - BIT类型：BIT。
  - 字符串二进制类型：CHAR(N)、VARCHAR(N)、BINARY、VARBINARY。
  - 文本类型：TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT。
  - 日期类型：DATE、TIME、DATETIME、TIMESTAMP、YEAR。
  - 大对象类型：TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB。
  - 数据类型属性：UNSIGNED、ZEROFILL。
- M-Compatibility模式数据库，逻辑解码特殊约束如下：
  - CREATE TABLE、ALTER TABLE、DROP TABLE等语法中的 [partition\_options]、ENGINE、ROW\_FORMAT、algorithm\_option、lock\_option等选项在语法中无实际作用，此类语法不进行解码输出。
  - 对于ALTER SCHEMA、CREATE SCHEMA、DROP SCHEMA、ALTER DATABASE、CREATE DATABASE、DROP DATABASE语法中，因为M-Compatibility模式数据库下，DATABASE与Schema等价，所以ALTER DATABASE、CREATE DATABASE、DROP DATABASE会被解码为ALTER SCHEMA、CREATE SCHEMA、DROP SCHEMA。
  - 对于写法不同的字符设置语法，比如CHARACTER SET、CHAR SET、CHARSET都会被解码为CHARACTER SET。
  - 对于ALTER TABLE tbl\_name DROP {INDEX | KEY} index\_name语法，逻辑解码结果为DROP INDEX；对于删除主键 ALTER TABLE tbl\_name DROP {primary key | {index | key} index\_name} 语法会被解码为 ALTER TABLE tbl\_name DROP CONSTRAINT index\_name。
  - 对于ALTER TABLE tbl\_name ADD INDEX语法，逻辑解码结果为CREATE INDEX。
  - 在解码DML语句时，涉及插入时间类型带精度的（time、timestamp、datetime），按照最大精度6解码。
  - 对于支持二进制输入的数据类型（BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB），如果数据中包括'\0'会发生截断，仅支持并行解码BINARY格式输出（即解码任务参数decode-style='b'），串行解码、函数解码以及并行json、text格式输出均会发生截断。
- REPLACE语法为DML语法，REPLACE会被解码为对数据实际的操作，比如INSERT或DELETE+INSERT（存在主键或唯一键冲突）。
- COPY语法/LOAD语法为DML语法，会被解码为对数据实际的操作，比如多条INSERT。
- 逻辑复制槽名称必须小于64个字符，使用SQL函数创建或使用复制槽时，复制槽名称仅支持小写字母、数字以及\_、?、-和.字符，且不支持或..单独作为复制槽名称。调用JDBC API创建或使用复制槽时，复制槽名称仅支持小写字母、数字以及\_

字符（可以输入大写字母，但在内部会将其被转成小写字母进行处理，例如输入名称为MSLOT，实际名称是mslot）。

- 对多库的解码需要分别在库内创建流复制槽并开始解码，每个库的解码都需要单独扫描一遍日志。
- 不支持强切，强切后需要重新全量导出数据。
- 备机解码时，switchover和failover可能出现解码数据变多，需用户手动过滤。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。
- 备机解码时，如果需要删除数据库，需要业务保证数据完成迁移后，主机再删除逻辑复制槽对应的数据库。当前没有强制要求删除数据库之前一定要先删除库上的逻辑复制槽，可能出现当前库备机解码还未完成，主机删除数据库的情况。如果出现这种情况，当备机回放到删除数据库的动作，备机即使未完成当前库的所有解码任务，也会退出解码，如果再次重连，备机解码会因为已经删除数据库而无法启动，造成解码任务受损。
- 只支持CN和主DN创建删除复制槽。当删除的复制槽为最后一个复制槽时，删除完成后会产生告警"replicationSlotMinLSN is INVALID\_WAL\_REC\_PTR!!!"和"replicationSlotMaxLSN is INVALID\_WAL\_REC\_PTR!!!"。
- 不允许主备DN、多个备DN同时使用同一个复制槽解码，否则会产生数据不一致或者其他异常错误的情况。
- 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，用户需手动过滤。
- 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未启动长事务，启动长事务会阻塞逻辑复制槽的创建。
- 不支持全局临时表的DML解码。
- 不支持本地临时表的DML解码。
- SELECT INTO语句会解码创建目标表的DDL操作，数据插入的DML操作不解码。
- 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，具体配置方式请参考《开发指南》中“SQL参考 > SQL语法 > A > 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，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 > A > 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5万时，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7FFFFFFF/FFFFFFF。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7FFFFFFF/FFFFFFF），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。
- 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15秒再次重试解码，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，需在哪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），解码输出内容将出现被截断现象。
- 不支持无日志表的DML解码。
- 执行备份恢复操作时，实例恢复完成后会清理所有的逻辑复制槽，如有需要须重新建槽。
- 删除复制槽需要在删除数据库之前执行，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，否则会阻塞wal日志回收。

- 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。
- 不支持全局二级索引，不支持分布列修改的DML解码。
- 容灾集群的备集群不支持通过SQL系统函数或工具进行逻辑解码。
- 不支持账本数据库功能，当前版本如果开启解码任务的数据库中有关于账本数据库的DML操作，则解码结果中会包含hash列，从而导致回放失败。
- 扩容场景下，若集群中创建有逻辑复制槽，会导致扩容失败，因此扩容前需要删除当前集群中已存在的逻辑复制槽。
- 不支持主键、外键、唯一约束中的信息约束（如not enforced等）选项，如果出现信息约束，则涉及信息约束的约束不解码，其他DDL照常解码。例如：“create table test(a int primary key not enforced);”语句会被解码为“create table test(a int);”。
- 使用CREATE TABLE语法创建表时不支持MURMURHASH选项的解码，若有MURMURHASH选项，将不解码该DDL语句。
- 仅支持wal\_level=logical的WAL日志解码，对于非logical的日志，串行解码（包括函数解码）输出结果中没有对应的值和类型，并行解码不输出其逻辑日志。
- 逻辑解码支持如下Xlog类型的DML：
  - Astore: insert、delete、update、multi-insert。
  - Ustore: insert、delete、update、multi-insert。
- wal\_level值为logical的情况下，Xlog日志数据量相比hot\_standby级别有一定膨胀。例如：在TPCC场景下，Astore的Xlog膨胀率约为11%，Ustore的Xlog膨胀率约为110%。
- 在大小写不敏感数据库中解码时，无论创建表名或用户名时使用的大写还是小写，解码选项白名单（white-table-list）或黑名单（exclude-users）中的表名或用户名都须使用小写。
- 逻辑复制槽会保护未解析XLog对应版本的系统表记录不被过早清理。当逻辑复制槽处于不活跃状态或者客户端消费过慢时，可能会对业务造成以下影响：
  - a. 如果业务中有大量的DDL，则会导致系统表历史版本过多，影响SQL命令执行效率。
  - b. 由于GaussDB数据页面管理的约束：同一页面中无法存放xid差值超过 $2^{32}$ 的两条记录。当系统表历史版本过多时，可能会阻塞DDL和DML等业务操作无法正常执行（错误码：GAUSS-21297）。

处理措施：

- a. 及时清理不再使用的逻辑复制槽。
  - b. 如果逻辑复制槽推进过慢，请参考《故障处理》中“常见故障界定与处理 > 逻辑解码常见问题 > 逻辑解码 > 逻辑复制槽异常/逻辑解码性能慢”章节进行处理。
  - c. 如果措施**b**未解决推进慢问题，则需要删除复制槽，重建逻辑解码任务。
- 从503.1.0之前的版本升级到最新版本，逻辑任务白名单校验规则优化在升级观察期不生效，与旧版本行为保持一致。

## SQL 函数解码性能

1. 在Benchmarksql-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%。

2. 在Benchmarksq1-5.0的100warehouse场景下，采用pg\_logical\_get\_area\_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。

3. 在Benchmarksq1-5.0的100warehouse场景下，采用pg\_logical\_slot\_get\_binary\_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2MB/s~3MB/s。
- 单次解码数据量256K行（对应约320MB~640MB日志），解码性能2MB/s~3MB/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\_binary\_changes + pg\_replication\_slot\_advance方式，解码性能相比采用pg\_logical\_slot\_get\_binary\_changes时要下降30%~50%。

## 流式解码性能

在并行解码的标准场景下（16核CPU、内存128GB、网络带宽 > 200MBps、表的列数为10~100、单行数据量0.1KB~1KB、DML操作以insert为主、不涉及落盘事务即单个事务中语句数量小于4096、parallel-decode-num为8、解码格式为't'且开启批量发送功能），连接DN解码性能（以Xlog消耗量为标准）不低于100MBps，连接CN解码性能不低于80MBps。为保证解码性能达标以及尽量降低对业务的影响，一台备机上应尽量仅建立一个并行解码连接，保证CPU、内存、带宽资源充足。

### 4.1.2 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。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：boolean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：boolean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：boolean型，针对并行解码场景默认值为false，针对SQL函数解码和串行解码场景默认值为true。
  - 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：boolean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。  
串行解码-取值范围：0~100的整型，默认值为0，即不开启此种管控。  
并行解码-取值范围：0~max\_process\_memory总量的25%，默认值为max\_process\_memory/4/1024，其中1024为kB到MB的单位转换，0表示不开启此条内存管控项。
- **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。  
串行解码-取值范围：0~100的整型，默认值为0，即不开启此种管控。  
并行解码-取值范围：0~max\_process\_memory总量的50%，默认值为max\_process\_memory/2/1048576，其中1048576为kB到GB的单位转换，0表示不开启此条内存管控项。

- desc-memory-limit  
内存管控参数，单位为MB，逻辑解码任务维护的表元信息总内存大于该值时，触发淘汰机制清理部分表元信息。  
取值范围：10~1024的整型，默认值为100。
- 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：boolean型，默认值为false。
  - false：设为false时，事务的BEGIN逻辑日志不输出事务的用户名。
  - true：设为true时，事务的BEGIN逻辑日志输出事务的用户名。
- exclude-userids:  
黑名单用户的OID参数，该参数只支持直连DN解码任务配置，分布式CN强一致解码不支持该参数。  
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。如果解码某条XLOG，且XLOG写入时，用户未创建，则认为用户不存在。  
取值范围：boolean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；当用户存在，黑名单功能正常过滤用户的操作。
  - true：设为true时，当解码观测到黑名单exclude-users中用户不存在时不报错，并正常解码；当用户存在，黑名单功能正常过滤用户的操作。
- standby-connection:  
仅流式解码设置，是否仅限制备机解码，因为CN没有备机，所以该参数在仅连接DN时支持。  
取值范围：boolean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。

#### 说明

如果主机资源使用率较大且业务对增量数据同步的实时性不敏感，建议进行备机解码；如果业务对增量数据同步的实时性要求高并且主机业务压力较小，建议使用主机解码。

- sender-timeout:  
仅流式解码设置，内核与客户端的心跳超时阈值。如果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。设置为0，表示逻辑解码不会主动断开与客户端的连接，如果设置过小，例如1ms，则可能存在解码任务中断风险。

- change-log-max-len:**  
逻辑日志缓存长度上限参数，单位为字节。仅连接DN的并行解码有效，分布式强一致解码、串行解码及SQL函数解码无效。如果单条解码结果长度超过上限，则会销毁重新分配大小为1024字节的内存并缓存。过长会增加内存占用，过短会频繁触发内存申请和释放的操作，不建议设置成小于1024的值。  
取值范围：1~65535，默认值为4096。
- max-decode-to-sender-cache-num:**  
并行解码日志的缓存条数阈值。仅连接DN的并行解码有效，分布式强一致解码、串行解码及SQL函数解码无效。缓存中的日志条数未超过这个阈值时，使用完毕的解码日志将置入缓存，否则直接释放。  
取值范围：1~65535，默认值为4096。
- enable-heartbeat:**  
仅流式解码时设置，代表是否输出心跳日志。  
取值范围：boolean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，此处说明并行解码场景心跳日志如何解析：二进制格式首先是字符'h'表示消息是心跳日志，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，代表是否使用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，代表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，用于跳过存储生成列的输出。对UPDATE和DELETE的旧元组无效，相应元组始终会输出存储生成列。分布式版本暂不支持存储生成列，此配置选项暂无实际影响。  
取值范围：boolean型，默认值为false/off。
    - true/on: 值为true/on时，不输出存储生成列的解码结果。
    - false/off: 设为false/off时，输出存储生成列的解码结果。虚拟生成列不受此参数控制，DML的解码结果始终不会输出虚拟生成列。
  - enable-ddl-decoding:  
逻辑解码控制参数，用于控制是否开启DDL语句的逻辑解码。  
取值范围：boolean型，默认值为false。
    - true: 值为true时，开启DDL语句的逻辑解码。
    - false: 值为false时，不开启DDL语句的逻辑解码。
  - enable-ddl-json-format:  
逻辑解码控制参数，用于控制DDL的反解析流程以及输出形式。  
取值范围：boolean型，默认值为false。
    - true: 值为true时，传送JSON格式的DDL反解析结果。
    - false: 设为false时，传送decode-style指定格式的DDL反解析结果。
  - timezone-is-utc:  
逻辑解码控制参数，用于控制时间类型（携带时区）数据的输出。如ORA、MYSQL兼容下的timestampz类型，M兼容的timestamp类型都表示带时区的时间类型，默认会输出当前数据库时区的时间，设置为true时，在解码时会强制输出为0时区的时间。该参数仅对流式解码有效，函数解码使用该参数会忽略不生效。  
取值范围：boolean型，默认值为false。

- true: 值为true时, 解码时间类型数据输出0时区的时间。
- false: 值为false时, 解码时间类型数据输出当前数据库时区的时间。
- decode-sequence:  
逻辑解码控制参数, 用来指定是否输出sequence值的变更日志的解码结果。  
取值范围: boolean型, 默认值为false。
  - true: 暂不支持设置。
  - false: 设为false时, 不输出sequence值的变更日志的解码结果。

#### 须知

解码选项decode-sequence当前仅允许设置为false, 设置为true会在启动解码时报错退出。

## 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, CN只会收集指定DN分片的复制表解码结果。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：boolean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boolean型。以文本格式输出解码结果。
  - 针对流式解码（仅连接DN时支持）：  
取值范围：boolean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:  
当enable-ddl-json-format参数值为true时，DDL的格式由enable-ddl-json-format控制，decode-style仅指定DML语句的解码格式；当enable-ddl-json-format参数值为false时，decode-style指定DML和DDL语句的解码格式。  
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：
  - 没有指定decode-style:  
针对复制槽插件类型为mppdb\_decoding、sql\_decoding，decode-style默认值为'b'即二进制格式解码。针对复制槽插件类型为parallel\_binary\_decoding、parallel\_json\_decoding、parallel\_text\_decoding，decode-style默认值分别为'b'、'j'、't'，解码格式分别为二进制格式、json格式、text格式。
  - 指定decode-style:  
按照指定的decode-style进行解码。

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）。

例如：以mppdb\_decoding插件为例，当decode-style为b类型时，以二进制格式解码，结果如下：

```
current_lsn: 0/CFE5C80 BEGIN CSN: 2357 first_lsn: 0/CFE5C80
current_lsn: 0/CFE5D40 INSERT INTO public.test1 new_tuple: {a[typid = 23]: "1", b[typid = 23]: "2"}
current_lsn: 0/CFE5E68 COMMIT xid: 78108
```

当decode-style为j类型时，以json格式解码，结果如下：

```
BEGIN CSN: 2358 first_lsn: 0/CFE6220
{"table_name": "public.test1", "op_type": "INSERT", "columns_name": ["a", "b"], "columns_type":
["integer", "integer"], "columns_val": ["3", "3"], "old_keys_name": [], "old_keys_type": [], "old_keys_val": []}
COMMIT XID: 78109
```

当decode-style为t类型时，以text格式解码，结果如下：

```
BEGIN CSN: 2359 first_lsn: 0/CFE64D0  
table public test1 INSERT: a[integer]:3 b[integer]:4  
COMMIT XID: 78110
```

## 📖 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代表本批次解码完成。
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。

### 4.1.3 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DN。

**步骤2** 使用如下命令通过DN端口连接数据库。

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

其中，user1为用户名，password为密码，gaussdb为需要连接的数据库名称，40000为数据库DN端口号，用户可根据实际情况替换。复制槽是建立在DN上的，因此需要通过DN端口连接数据库。

**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** 通过CN端口连接数据库，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** 参考**步骤2**连接DN，读取复制槽slot1解码结果，解码条数为4096。

#### 说明

逻辑解码选项请参见[逻辑解码选项](#)。

```
gaussdb=> SELECT * FROM pg_logical_slot_peek_changes('slot1', NULL, 4096);
location | xid | data
-----+-----+-----
+-----+-----+-----
-----+-----+-----
0/601C188 | 1010023 | BEGIN 1010023
0/601ED60 | 1010023 | COMMIT 1010023 (at 2023-09-14 16:03:51.394287+08) 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 (at 2023-09-14 16:03:57.239821+08) 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

### 4.1.4 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。

### 4.1.5 逻辑解码支持 DDL

GaussDB主机上正常执行DDL语句，通过逻辑解码工具可以获取到DDL语句。

表 4-1 具体支持的 DDL 类型

表	索引	自定义函数	自定义存储过程	触发器	Sequence	Schema	Comment
CREATE TABLE [PARTITION   AS   SUBPARTITION   LIKE] ALTER TABLE [PARTITION   SUBPARTITION] DROP TABLE RENAME TABLE TRUNCATE	CREATE INDEX ALTER INDEX DROP INDEX REINDEX	CREATE FUNCTION ALTER FUNCTION DROP FUNCTION	CREATE PROCEDURE ALTER PROCEDURE DROP PROCEDURE	CREATE TRIGGER ALTER TRIGGER DROP TRIGGER	CREATE SEQUENCE ALTER SEQUENCE DROP SEQUENCE	CREATE SCHEMA ALTER SCHEMA DROP SCHEMA	COMMENT

表 4-2 M-Compatibility 模式数据库支持的 DDL 类型

表	索引	Sequence	Schema	Comment on
ALTER TABLE	ALTER INDEX	ALTER SEQUENCE	ALTER SCHEMA	COMMENT ON
CREATE TABLE	CREATE INDEX	CREATE SEQUENCE	CREATE SCHEMA	
DROP TABLE	DROP INDEX	DROP SEQUENCE	DROP SCHEMA	
ALTER TABLE PARTITION	REINDEX			
CREATE TABLE PARTITION	ALTER TABLE DROP INDEX		ALTER DATABASE	
TRUNCATE	ALTER TABLE ADD INDEX		CREATE DATABASE	
			DROP DATABASE	

## 功能描述

数据库在执行DML的时候，存储引擎会生成对应的DML日志，用于进行恢复，对这些DML日志进行解码，即可还原对应的DML语句，生成逻辑日志。而对于DDL语句，数据库并不记录DDL原语句的日志，而是记录DDL语句涉及的系统表的DML日志。DDL种类多样、语法复杂，逻辑复制要支持DDL语句，通过这些系统表的DML日志来解码原DDL语句是非常困难的。新增DDL日志记录原DDL信息，并在解码时通过DDL日志可以得到DDL原语句。

在DDL语句执行过程中，SQL引擎解析器会对原语句进行语法、词法解析，并生成解析树（不同的DDL语法会生成不同类型的解析树，解析树中包含DDL语句的全部信息）。随后，执行器通过这些信息执行对应操作，生成、修改对应元信息。

本节通过新增DDL日志的方式，来支持逻辑解码DDL，其内容由解析器结果（解析树）以及执行器结果生成，并在执行器执行完成后生成该日志。

从语法树反解析出DDL，DDL反解析能够将DDL命令转换为JSON格式的语句，并提供必要的信息在目标位置重建DDL命令。与原始DDL命令字符串相比，使用DDL反解析的好处包括：

1. 解析出来的每个数据库对象都带有Schema，因此如果使用不同的search\_path，也不会有歧义。
2. 结构化的JSON和格式化的输出能支持异构数据库。如果用户使用的是不同的数据库版本，并且存在某些DDL语法差异，需要在应用之前解决这些差异。

反解析输出的结果是规范化后的形式，结果与用户输入等价，不保证完全相同，例如：

示例1：在函数体中没有单引号'时，函数体的分隔符\$\$会被解析为单引号'。

原始SQL语句：

```
CREATE FUNCTION func(a INT) RETURNS INT AS
$$
BEGIN
a:= a+1;
CREATE TABLE test(col1 INT);
```

```
INSERT INTO test VALUES(1);
DROP TABLE test;
RETURN a;
END;
$$
LANGUAGE plpgsql;
```

反解析结果:

```
CREATE FUNCTION public.func ( IN a pg_catalog.int4 ) RETURNS pg_catalog.int4 LANGUAGE plpgsql
VOLATILE CALLED ON NULL INPUT SECURITY INVOKER COST 100 AS '
BEGIN
a:= a+1;
CREATE TABLE test(col1 INT);
INSERT INTO test VALUES(1);
DROP TABLE test;
RETURN a;
END;
';
```

示例3: “ALTER INDEX "Alter\_Index\_Index" REBUILD PARTITION "CA\_ADDRESS\_SK\_index2"” 会被反解析为 “REINDEX INDEX public."Alter\_Index\_Index" PARTITION "CA\_ADDRESS\_SK\_index2"”。

示例4: 创建/修改范围分区表, START END语法格式均解码转化为LESS THAN语句:

```
gaussdb=# CREATE TABLE test_create_table_partition2 (c1 INT, c2 INT)
PARTITION BY RANGE (c2) (
PARTITION p1 START(1) END(1000) EVERY(200) ,
PARTITION p2 END(2000),
PARTITION p3 START(2000) END(2500),
PARTITION p4 START(2500),
PARTITION p5 START(3000) END(5000) EVERY(1000)
);
```

会被反解析为:

```
gaussdb=# CREATE TABLE test_create_table_partition2 (c1 INT, c2 INT)
PARTITION BY RANGE (c2) (
PARTITION p1_0 VALUES LESS THAN ('1'), PARTITION p1_1 VALUES LESS THAN ('201'), PARTITION p1_2
VALUES LESS THAN ('401'), PARTITION p1_3 VALUES LESS THAN ('601'), PARTITION p1_4 VALUES LESS
THAN ('801'), PARTITION p1_5 VALUES LESS THAN ('1000'),
PARTITION p2 VALUES LESS THAN ('2000'),
PARTITION p3 VALUES LESS THAN ('2500'),
PARTITION p4 VALUES LESS THAN ('3000'),
PARTITION p5_1 VALUES LESS THAN ('4000'),
PARTITION p5_2 VALUES LESS THAN ('5000')
);
```

示例5: 新增表的列字段时, 使用IF NOT EXISTS判断。

- 原始SQL语句:

```
gaussdb=# CREATE TABLE IF NOT EXISTS tb5 (c1 int,c2 int) with (ORIENTATION=ROW,
STORAGE_TYPE=USTORE);
gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN IF NOT EXISTS c2 char(5) after c1; -- 可解码。
TABLE中已有int型的列c2, 语句执行跳过, 反解析结果中c2列的类型仍为原来的类型。
```

反解析结果:

```
gaussdb=# ALTER TABLE IF EXISTS public.tb5 ADD COLUMN IF NOT EXISTS c2 pg_catalog.int4 AFTER
c1;
```

- 原始SQL语句:

```
gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN IF NOT EXISTS c2 char(5) after c1, ADD
COLUMN IF NOT EXISTS c3 char(5) after c1; -- 解码。新增列c3的反解析结果类型正确。
```

反解析结果:

```
gaussdb=# ALTER TABLE IF EXISTS public.tb5 ADD COLUMN IF NOT EXISTS c2 pg_catalog.int4 AFTER
c1, ADD COLUMN IF NOT EXISTS c3 pg_catalog.bpchar(5) AFTER c1;
```

- 原始SQL语句:

```
gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN c2 char(5) after c1, ADD COLUMN IF NOT
EXISTS c4 int after c1; --不解码, 语句执行错误。
```

示例6：定义数据库对象（DROP TABLE/INDEX/SEQUENCE）时，解码结果会添加IFEXISTS语法选项。

- 原始SQL语句：

```
gaussdb=# CREATE TABLE IF NOT EXISTS tb6 (c1 int,c2 int) with (ORIENTATION=ROW,  
STORAGE_TYPE=USTORE);  
gaussdb=# DROP TABLE tb6;
```

反解析结果：

```
gaussdb=# DROP TABLE IF EXISTS public.tb6 RESTRICT;
```

## 规格约束

- 逻辑解码支持DDL规格：
  - 纯DDL逻辑解码性能标准环境下约为100MB/S，DDL/DML混合事务逻辑解码性能标准环境下约为100MB/S。
  - 开启此功能后（设置wal\_level=logical且enable\_logical\_replication\_ddl=on），对DDL语句影响性能下降小于15%。
- 解码通用约束（串行和并行）：
  - 不支持解码本地临时对象的DDL操作。
  - 不支持FOREIGN TABLE场景的DDL解码。
  - alter table add column的default值不支持stable类型和volatile类型的函数；create table和alter table的column的check表达式不支持stable类型和volatile类型的函数；alter table如果有多条子语句，只要其中一条子语句存在上述两种情况，则该条alter table整条语句不反解析。

```
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b1 TIMESTAMP DEFAULT NOW(); -- 's' NOT DEPARSE  
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b2 INT DEFAULT RANDOM(); -- 'v' NOT DEPARSE  
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b3 INT DEFAULT ABS(1); -- 'i' DEPARSE
```
  - 不支持分布式CREATE MATERIALIZED VIEW的DDL解码。
  - 不支持CREATE/ALTER/DROP VIEW、COMMENT ON VIEW的DDL解码。
  - 不支持REINDEX DATABASE/SYSTEM的DDL解码。
  - 不支持视图上触发器相关的DDL解码。
  - 不支持CONCURRENTLY相关语句的DDL解码。
  - 创建对象时语句中存在IF NOT EXISTS时，如果对象已存在，则不进行解码。删除对象时语句中存在IF EXISTS时，如果对象不存在，则不进行解码。
  - 不对ALTER PACKAGE COMPILE语句进行解码，但会解码实例化内容中包含的DDL/DML语句。如果PACKAGE里没有DDL或DML部分的实例化内容，则alter package compile会被逻辑解码忽略。
  - 仅支持本版本之前版本的商用DDL语法，以下SQL语句不支持逻辑解码。

- 创建行存表，设置ILM策略。

原始SQL语句：

```
gaussdb=# CREATE TABLE IF NOT EXISTS tb3 (c1 int) with  
(storage_type=USTORE,ORIENTATION=ROW) ILM ADD POLICY ROW STORE COMPRESS  
ADVANCED ROW AFTER 7 day OF NO MODIFICATION;
```

反解析结果：

```
gaussdb=# CREATE TABLE IF NOT EXISTS public.tb3 (c1 pg_catalog.int4) WITH  
(storage_type = 'ustore', orientation = 'row', compression = 'no') NOCOMPRESS;
```

- B兼容模式下创建或修改表时，给表或列添加注释。

原始SQL语句：

```
gaussdb=# CREATE TABLE IF NOT EXISTS tb6 (c1 integer comment 'Mysql兼容注释语法')  
with (ORIENTATION=ROW, STORAGE_TYPE=USTORE);
```

反解析结果:

```
gaussdb=# CREATE TABLE IF NOT EXISTS public.tb6 (c1 pg_catalog.int4) WITH  
(storage_type = 'ustore', orientation = 'row', compression = 'no') NOCOMPRESS;
```

- 逻辑解码不支持DDL ( DCL ) /DML混合事务，混合事务中DDL之后的DML解码不支持。

-- 均不反解析，DCL为不支持语句故不解析，DML处于DCL之后也不反解析

```
gaussdb=# BEGIN;  
gaussdb=# GAIINT ALL PRIVILEGES to u01;  
gaussdb=# INSERT INTO test1(col1) values(1);  
gaussdb=# COMMIT;
```

-- 只反解析第一句和第三句SQL语句

```
gaussdb=# BEGIN;  
gaussdb=# CREATE TABLE mix_tran_t4(id int);  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# CREATE TABLE mix_tran_t5(id int);  
gaussdb=# COMMIT;
```

-- 只反解析第一句和第二句SQL语句

```
gaussdb=# BEGIN;  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# CREATE TABLE mix_tran_t6(id int);  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# COMMIT;
```

-- 全反解析

```
gaussdb=# BEGIN;  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# CREATE TABLE mix_tran_t7(id int);  
gaussdb=# CREATE TABLE mix_tran_t8(id int);  
gaussdb=# COMMIT;
```

-- 只反解析第一句和第三句SQL语句

```
gaussdb=# BEGIN;  
gaussdb=# CREATE TABLE mix_tran_t7(id int);  
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);  
gaussdb=# CREATE TABLE mix_tran_t8(id int);  
gaussdb=# COMMIT;
```

-- 全反解析

```
gaussdb=# BEGIN;  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# COMMIT;
```

-- 只反解析第一句SQL语句

```
gaussdb=# BEGIN;  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# COMMIT;
```

-- 只反解析第一句和第三句SQL语句

```
gaussdb=# BEGIN;  
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);  
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);  
gaussdb=# CREATE TABLE mix_tran_t9(id int);  
gaussdb=# COMMIT;
```

- 逻辑解码语句CREATE TABLE AS SELECT、SELECT INTO和CREATE TABLE AS仅能解码出CREATE TABLE语句，暂不支持解码INSERT语句。

### 说明

对于CTAS创建的表，仍会解码其ALTER和DROP语句。



示例:

原始SQL语句:

```
CREATE TABLE IF NOT EXISTS tb35_2 (c1 int) with (storage_type=USTORE,ORIENTATION=ROW);
INSERT INTO tb35_2 VALUES (6);
CREATE TABLE tb35_1 with (storage_type=USTORE,ORIENTATION=ROW) AS SELECT * FROM
tb35_2;
```

最后一句SQL语句反解析结果:

```
CREATE TABLE public.tb35_1 (c1 pg_catalog.int4) WITH (storage_type = 'ustore', orientation =
'row', compression = 'no') NOCOMPRESS;
```

- 执行存储过程/函数/高级包时,若其本身包含DDL/DML混合事务或者其本身与同事务内其他语句组成DDL/DML混合事务,则按照混合事务原则执行解码。
- 逻辑解码不支持账本数据库功能,创建账本数据库的DDL语句解码结果中会包含hash列。

■ 原始语句:

```
CREATE SCHEMA blockchain_schema WITH BLOCKCHAIN;
CREATE TABLE blockchain_schema.blockchain_table(mes int);
```

■ 解码结果:

```
CREATE SCHEMA blockchain_schema WITH BLOCKCHAIN;
CREATE TABLE blockchain_schema.blockchain_table (mes pg_catalog.int4, hash_a1d895
pg_catalog.hash16); -- 此语句无法在目标端回放。
```

需要在目标端手动关闭blockchain\_schema的防篡改属性后,才可以正常回放,此时目标端的blockchain\_table等同于一张普通表,再次执行DML命令可以正常回放。

SQL命令:

```
ALTER SCHEMA blockchain_schema WITHOUT BLOCKCHAIN;
CREATE TABLE blockchain_schema.blockchain_table (mes pg_catalog.int4, hash_a1d895
pg_catalog.hash16);
```

- 串行逻辑解码支持DDL特有约束:
  - sql\_decoding插件不支持json格式的DDL。

## 解码格式

- JSON格式

对于输入的DDL语句,SQL引擎解析器会通过语法、词法分析将其分解为解析树,解析树节点中包含了DDL的全部信息,并且执行器会根据解析树内容,执行系统元信息的修改。在执行器执行完成之后,便可以获取到DDL操作数据对象的search\_path。本特性在执行器执行成功之后,对解析树信息以及执行器结果进行反解析,以还原出DDL原语句的全部信息。反解析的方式可以分解整个DDL语句,以方便输出JSON格式的DDL,用以适配异构数据库场景。

CREATE TABLE语句在经过词法、语法分析之后,得到对应的CreateStmt解析树节点,节点中包含了表信息、列信息、分布式信息(DistributeBy结构体)、分区信息(PartitionState结构)等。通过反解析后,可输出的JSON格式如下:

```
{
  "JDDL": {
    "fmt": "CREATE %s TABLE %s %s %s %s",
    "with_clause": "WITH (%s)",
    "identity": "D %s",
    "table_elements": "%s",
    "object_name": "test_create_table_a",
    "schema_name": "public",
    "compression": "NOCOMPRESS",
    "persistence": "",
    "with_clause": "WITH (%s)",
    "with": {
      "label": "orientation",
      "value": "row",
      "label": "compression",
      "value": "no"
    },
    "if_not_exists": "",
    "table_elements": {
      "name": "a",
      "column_type": "int4",
      "typmod": "",
      "typarray": false,
      "type_name": "int4",
      "schema_name": "pg_catalog"
    }
  }
}
```

可以看到，JSON格式中包含对象的search\_path，其中的identity键标识schema为public，表名为test\_create\_table\_a，其中%{persistence}s对应的字段如下，此SQL语句不含此字段所以为空。

```
[ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ]
```

%{if\_not\_exists}s对应SQL语句中的字段，不含此字段所以为空：

```
[ IF NOT EXISTS ]
```

%{identity}D对应SQL语句中的字段：

```
table_name
```

%{table\_elements}s对应SQL语句中的字段：

```
( column_name data_type )
```

%{with\_clause}s对应SQL语句中的字段：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
```

%{compression}s对应SQL语句中的字段：

```
[ COMPRESS | NOCOMPRESS ]
```

- decode-style指定格式

输出的格式由decode-style参数控制，如当decode-style='j'时，输出格式如下：

```
{"TDDL":"CREATE TABLE public.test_create_table_a (a pg_catalog.int4) WITH (orientation = 'row', compression = 'no') NOCOMPRESS"}
```

其中语句中也包含Schema名称。

## 接口设计

- 新增控制参数

a. 新增逻辑解码控制参数，用于控制DDL的反解析流程以及输出形式。可通过JDBC接口或者pg\_logical\_slot\_peek\_changes开启。

- enable-ddl-decoding：默认false，不开启DDL语句的逻辑解码；值为true时，开启DDL语句的逻辑解码。
- enable-ddl-json-format：默认false，传送TEXT格式的DDL反解析结果；值为true时，传送JSON格式的DDL反解析结果。

b. 新增GUC参数

- enable\_logical\_replication\_ddl：默认为ON，ON状态下，逻辑复制可支持DDL，否则，不支持DDL。只有当ON状态下，才会对DDL执行结果进行反解析，并生成DDL的WAL日志。否则，不反解析也不生成WAL日志。

enable\_logical\_replication\_ddl的开关日志，以证明是否是用户修改了该参数导致逻辑解码不支持DDL。

- 新增日志

新增DDL日志xl\_logical\_ddl\_message，其类型为RM\_LOGICALDDLMSG\_ID。其定义如下：

名称	类型	意义
db_id	OID	数据库ID
rel_id	OID	表ID
csn	CommitSeqNo	CSN快照

名称	类型	意义
cid	CommandId	Command ID
tag_type	NodeTag	DDL类型
message_size	Size	日志内容长度
filter_message_size	Size	日志中白名单过滤信息长度
message	char *	DDL内容

## 使用步骤

**步骤1** 逻辑解码特性需提前设置GUC参数wal\_level为logical，该参数需要重启生效。

```
gs_guc set -Z datanode -D $node_dir -c "wal_level = logical"
```

其中，\$node\_dir为数据库节点路径，用户可根据实际情况替换。

**步骤2** 以具有REPLICATION权限的用户登录GaussDB数据库主节点，使用如下命令连接数据库。

```
gsql -U user1 -W password -d db1 -p 16000 -r
```

其中，user1为用户名，password为密码，db1为需要连接的数据库名称，16000为数据库端口号，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    | 0/3764C788
(1 row)
```

**步骤4** 在数据库中创建Package。

```
gaussdb=# CREATE OR REPLACE PACKAGE ldp_pkg1 IS
  var1 int:=1; --公有变量
  var2 int:=2;
  PROCEDURE testpro1(var3 int); --公有存储过程，可以被外部调用
END ldp_pkg1;
/
```

**步骤5** 读取复制槽slot1解码结果，可通过JDBC接口或者pg\_logical\_slot\_peek\_changes推进复制槽。

### 说明

- 逻辑解码选项请参见[逻辑解码选项](#)和新增控制参数。
- 并行解码中，在JDBC接口中改变参数decode\_style可以决定解码格式：  
通过配置选项decode-style，指定解码格式。其取值为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。

```
gaussdb=# SELECT data FROM pg_logical_slot_peek_changes('slot1', NULL, NULL, 'enable-ddl-decoding',
'true', 'enable-ddl-json-format', 'false') WHERE data not like 'BEGIN%' AND data not like 'COMMIT%' AND
data not like '%dbe_pldeveloper.gs_source%';
```

data

```
----- {"TDDL":"CREATE OR REPLACE PACKAGE public.ldap_pkg1 AUTHID
```

```
CURRENT_USER IS var1 int:=1; --公有变量\n var2 int:=2;\nPROCEDURE testpro1(var3 int); --公有存储过程，可以被外部调用\nEND ldp_pkg1; \n /"}  
(1 row)
```

#### 步骤6 删除逻辑复制槽slot1，删除package ldp\_pkg1。

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

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

```
gaussdb=# DROP PACKAGE ldp_pkg1;  
NOTICE: drop cascades to function public.testpro1(integer)  
DROP PACKAGE
```

----结束

## 4.1.6 逻辑解码数据找回功能

逻辑解码对外提供单节点DML数据找回能力，从WAL日志是否归档的角度分为在线日志找回和归档日志找回：

- 在线数据找回：使用pg\_logical\_get\_area\_changes函数，可以在线找回相关DML数据，具体使用参考pg\_logical\_get\_area\_changes函数的使用方法。
- 归档数据找回：OM\_Agent通过pg\_logical\_get\_area\_changes函数提供OBS上归档日志的找回能力，OM\_Agent从OBS下载已经归档的日志到对应节点上，并创建与WAL日志文件前8位同名的文件夹。解码完成后，将找回的数据存入文件并返回路径。

### 约束

1. 解析的WAL日志级别为logical。
2. 数据表的复制标识必须为FULL，否则UPDATE和DELETE操作涉及到的被修改行不是全字段。
3. WAL日志记录的数据修改操作所对应的业务表，从找回起始位置到目前不能执行VACUUM FULL操作，否则该表VACUUM FULL之前的DML操作不会被数据找回。
4. WAL日志记录的数据修改操作所对应的业务表，从找回起始位置到目前，结构只能发生以下变化：
  - a. 业务表结构不发生变化，即无DDL操作。
  - b. 业务表在最后一列增加字段，且字段类型不能为带有默认值超过124字符长度的varchar类型。
  - c. 业务表删除某一行字段。
  - d. varchar字段增加原有长度。

其余表结构变更导致该表relfilenode发生变化的场景不支持，表结构变更前的DML操作不会被数据找回。

5. 每条WAL日志不能超过500MB。
6. 不支持扩容前的xlog日志数据找回。
7. 集群的每个分片都会生成一个结果文件，多个文件不会进行合并。
8. 仅支持归档数据找回，且需要开启归档，若数据尚未归档，则无法通过本接口找回。

9. OM\_Agent在下载之前会验证本地已用空间是否大于总空间的80%，如果大于则会报错（需要额外空间用于存放解码文件），报错信息为："no enough space left on device, available space must be greater than 20%"。
10. 下载失败或解码失败后，都会将下载的WAL日志文件进行清理，如果清理不成功，不会强制结束程序，只会把错误信息记录到DN的日志中。
11. 由用户传入的时间，起始时间不能超出系统表gs\_txn\_lsn\_time的最大时间，终止时间不能超过系统表gs\_txn\_lsn\_time的最小时间，否则将会报错。
12. 每次下载的xlog日志量不超过3GB。
13. 每次解码数据量不超过max\_process\_memory 的10%。
14. 不支持同一节点并发调用数据找回接口。

# 5 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，分六个章节对分区表使用进行系统性说明，包含语义、原理、约束限制等方面。

## 5.1 大容量数据库

### 5.1.1 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级数据量逐渐发展到现在的TB级数据量，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。

### 5.1.2 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，能够通过分区键（Partition Key）快速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。

### 5.1.3 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5-1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明显的效果，如图5-2所示。

图 5-1 分区表示例图

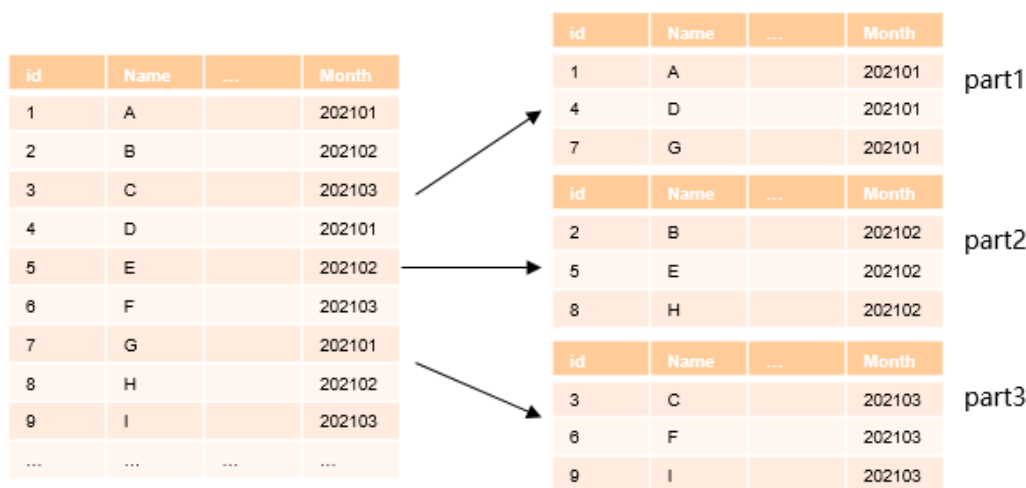
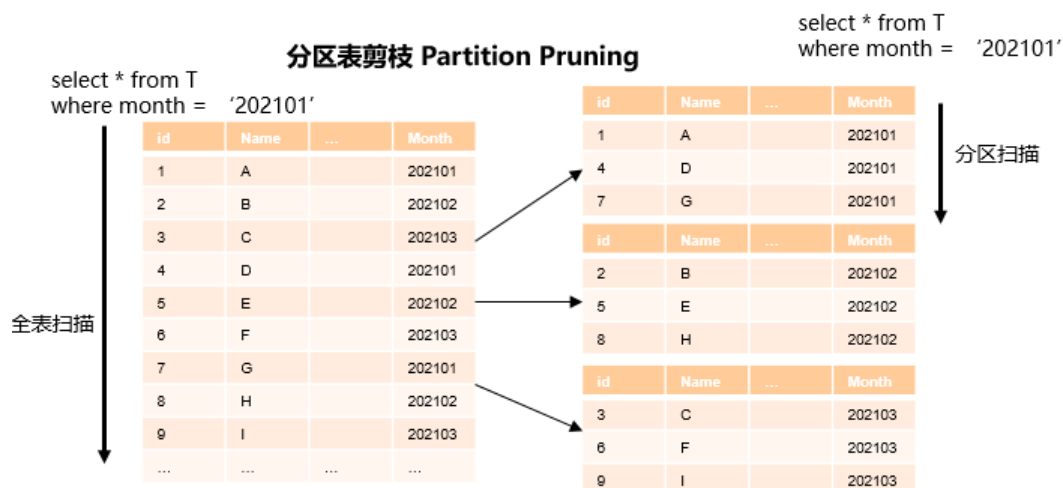


图 5-2 分区表剪枝示例图



### 5.1.4 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。

## 5.2 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）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。

### 5.2.1 基本概念

#### 5.2.1.1 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：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;

```

### 5.2.1.2 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：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;
```

### 5.2.1.3 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。

## 5.2.2 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) (...)
```

### 5.2.2.1 范围分区

范围分区（Range Partition）根据为每个分区建立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')  
);  
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)

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
```

### 5.2.2.2 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）的场景下，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。

### 5.2.2.3 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，往往能够准确的将目标数据保存在预想的分区中，但是如果列表值较多在分区定义时变得麻烦，该情况下推荐使用Hash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。

### 5.2.2.4 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5-1所示：

表 5-1 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
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。

## 5.2.3 分区基本使用

### 5.2.3.1 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/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子句中体现，*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) (
  { 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语句使用方法请参见[分区表运维管理](#)、《开发指南》中“SQL参考 > SQL语法 > ALTER TABLE PARTITION”章节。

### 5.2.3.2 使用和管理分区表

分区表支持大部分非分区表的相关功能，具体可以参考《开发指南》中常规表的各类操作语法相关资料。

除此之外，分区表还支持大量的分区级操作命令，包括分区级DQL/DML（如SELECT、INSERT、UPDATE、DELETE、UPSERT、MERGE INTO）、分区级DDL（如ADD、DROP、TRUNCATE、EXCHANGE、SPLIT、MERGE、MOVE、RENAME）、分区VACUUM/ANALYZE、分类分区索引等。相关命令使用方法请参见[分区表DQL/DML](#)、[分区索引](#)、[分区表运维管理](#)、以及《开发指南》中各个语法命令对应的章节。

分区级操作命令一般通过指定分区名或者分区值的方式进行，比如语法命令可能是如下情形：

```
sql_action [ t_name ] { PARTITION | SUBPARTITION } { p_name | (p_name) };  
sql_action [ t_name ] { PARTITION | SUBPARTITION } FOR (p_value);
```

通过指定分区名p\_name或指定分区值p\_value来定向操作某个特定分区，此时业务只会作用于对象分区，而不会影响其他任何分区。如果通过指定分区名p\_name来执行业务，数据库会匹配p\_name对应的分区，该分区不存在则业务抛出异常；如果通过指定分区值p\_value来执行业务，数据库会匹配p\_value值所属分区。

比如定义有如下的分区表：

```
gaussdb=# CREATE TABLE list_01  
(  
  id INT,  
  role VARCHAR(100),  
  data VARCHAR(100)  
)  
PARTITION BY LIST (id)  
(  
  PARTITION p_list_1 VALUES(0,1,2,3,4),  
  PARTITION p_list_2 VALUES(5,6,7,8,9),  
  PARTITION p_list_3 VALUES(DEFAULT)  
);  
  
-- 清理示例  
gaussdb=# DROP TABLE list_01;
```

指定分区业务中，PARTITION p\_list\_1与PARTITION FOR (4) 等价，为同一个分区；PARTITION p\_list\_3与PARTITION FOR (12) 等价，为同一个分区。



### 5.2.3.3 分区表 DQL/DML

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DQL/DML操作，指定分区可以通过PARTITION (partname)或者PARTITION FOR (partvalue)来进行。对于二级分区，可以通过SUBPARTITION(subpartname)或者SUBPARTITION FOR (subpartvalue)指定具体的二级分区。指定分区执行DQL/DML时，若插入的数据不属于目标分区，则业务报错；若查询的数据不属于目标分区，则跳过该数据的处理。

指定分区DQL/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 )

指定分区做DQL/DML的示例如下：

```
--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;
```

## 5.3 分区表查询优化

### 说明

本节示例对应explain\_perf\_mode参数值为normal。

### 5.3.1 分区剪枝

分区剪枝是GaussDB提供的一种分区表查询优化技术，数据库SQL引擎会根据查询条件，只扫描特定的部分分区。分区剪枝是自动触发的，当分区表查询条件符合剪枝场景时，会自动触发分区剪枝。根据剪枝阶段的不同，分区剪枝分为静态剪枝和动态剪枝，静态剪枝在优化器阶段进行，在生成计划之前，数据库已经知道需要访问的分区信息；动态剪枝在执行器阶段进行（执行开始/执行过程中），在生成计划时，数据库并不知道需要访问的分区信息，只是判断“可以进行分区剪枝”，具体的剪枝信息由执行器决定。

只有分区表页面扫描和Local索引扫描才会触发分区剪枝，Global索引没有分区的概念，不需要进行剪枝。

### 5.3.1.1 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，该操作与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，非分区键无报错的情况。

- 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
```

c. **数组表达式**

```
(15 rows)

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:

#### 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;
```

### 5.3.1.2 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。

#### 5.3.1.2.1 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。  
gaussdb=#  
--创建分区表

```
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;
```

### 5.3.1.2.2 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..3 (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..3 (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 = (lengthb(t3.c1)))
  -> 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, (lengthb(t3.c1))
    -> Streaming(type: REDISTRIBUTE)
      Output: t3.c1, t3.c2, (lengthb(t3.c1))
      Distribute Key: (lengthb(t3.c1))
      Spawn on: All datanodes
      Consumer Nodes: All datanodes
      -> Seq Scan on public.t3
        Output: t3.c1, t3.c2, lengthb(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;
```

### 5.3.2 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 5-3 Global Non-Partitioned Index

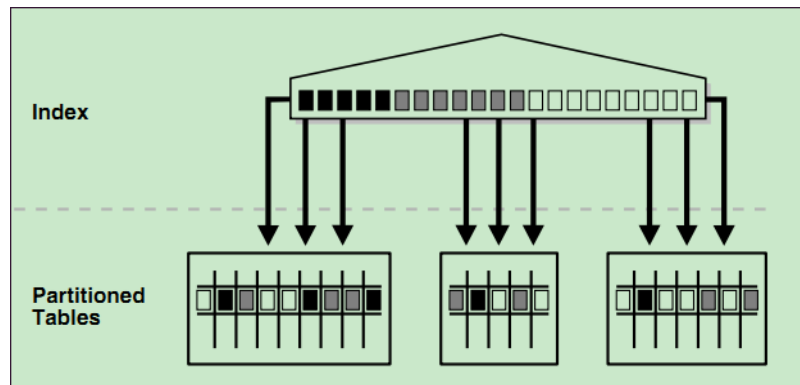


图 5-4 Global Partitioned Index

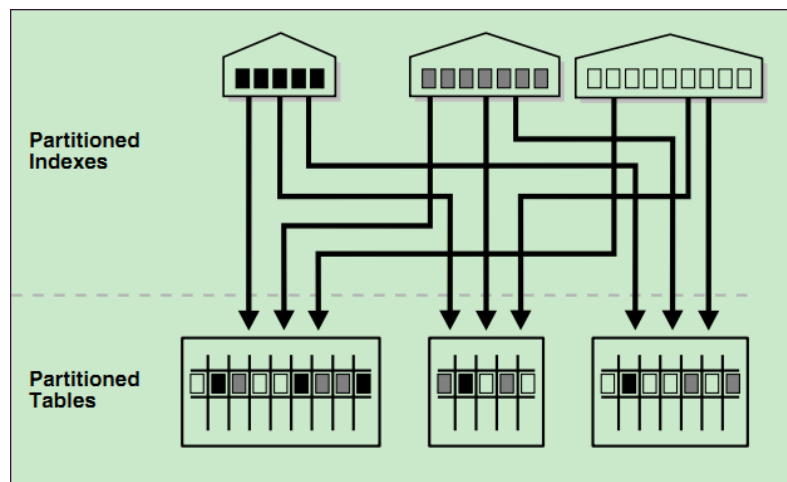
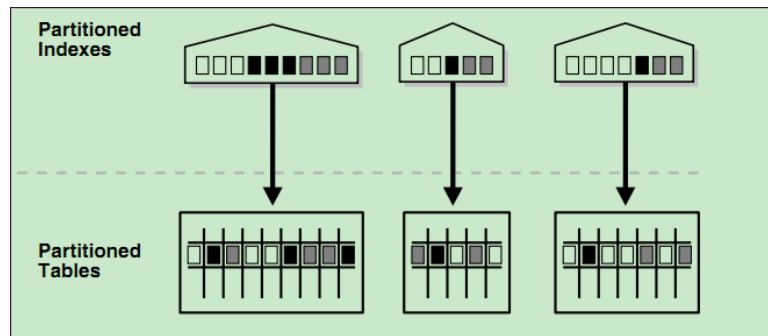


图 5-5 Local Partitioned Index



## 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。
- 在创建LOCAL索引时，可以通过FOR { partition\_name | ( partition\_value [, ...] ) }子句，指定在单个分区上创建LOCAL索引，此类索引在其他分区上不生效，后续新增的分区也不会自动创建该索引。需要注意的是，当前仅静态剪枝到单个分区的计划支持生成分类索引的查询路径。

### 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称。

```
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称。

```
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
```

- 创建分类分区索引。

指定分区名：

```
gaussdb=# CREATE INDEX tpcds_web_returns_for_p1 ON web_returns_p2 (ca_address_id) LOCAL(partition ind_part for p1);
```

指定分区键的值：

```
gaussdb=# CREATE INDEX tpcds_web_returns_for_p2 ON web_returns_p2 (ca_address_id) LOCAL(partition ind_part for (5000));
```

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;
```

### 5.3.3 分区表统计信息

对于分区表，支持收集分区级统计信息，相关统计信息可以在pg\_partition和pg\_statistic系统表以及pg\_stats和pg\_ext\_stats视图中查询。分区级统计信息适用于分区表进行静态剪枝后，分区表的扫描范围剪枝到单分区的场景下。分区级统计信息的支持范围为：分区级的page数和tuple数、单列统计信息、多列统计信息、表达式索引统计信息。

分区表统计信息有以下收集方式：

- 级联收集统计信息
- 指定具体单个分区收集统计信息

#### 5.3.3.1 级联收集统计信息

在ANALYZE | ANALYSE分区表时，系统会根据用户指定的或默认的PARTITION\_MODE，自动收集分区表中所有符合语义的分区级统计信息，PARTITION\_MODE的相关信息请参见《开发指南》中“SQL参考 > SQL语法 > ANALYZE | ANALYSE”中的PARTITION\_MODE参数。

#### 注意

- 当级联收集复制表、hashbucket表类型的分区表的统计信息，且PARTITION\_MODE为ALL时，其行为将转换为ALL COMPLETE模式。
- 分区级统计信息级联收集不支持default\_statistics\_target为负数的场景。

#### 示例

- 创建分区表并插入数据

```
gaussdb=# CREATE TABLE t1_range_int  
(  
  c1 INT,  
  c2 INT,  
  c3 INT,  
  c4 INT  
)  
PARTITION BY RANGE(c1)
```

```
(
PARTITION range_p00 VALUES LESS THAN(10),
PARTITION range_p01 VALUES LESS THAN(20),
PARTITION range_p02 VALUES LESS THAN(30),
PARTITION range_p03 VALUES LESS THAN(40),
PARTITION range_p04 VALUES LESS THAN(50)
);
gaussdb=# INSERT INTO t1_range_int SELECT v,v,v,v FROM generate_series(0, 49) AS v;
```

- 级联收集统计信息

```
gaussdb=# ANALYZE t1_range_int WITH ALL;
```

- 查看分区级统计信息

```
gaussdb=# SELECT relname, parttype, relpages, reltuples FROM pg_partition WHERE
parentid=(SELECT oid FROM pg_class WHERE relname='t1_range_int') ORDER BY relname;
```

relname	parttype	relpages	reltuples
range_p00	p	4	9
range_p01	p	7	17
range_p02	p	6	13
range_p03	p	2	5
range_p04	p	4	9
t1_range_int	r	0	0

(6 rows)

```
gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_stats WHERE
tablename='t1_range_int' ORDER BY tablename, partitionname, attname;
```

schemaname	tablename	partitionname	subpartitionname	attname	inherited	null_frac	avg_width	n_distinct	n_dndistinct	most_common_vals	most_common_freqs	histogram_bounds
public	t1_range_int	range_p00		c1	f	0	4	-1				
-1			{0,1,2,3,4,5,6,7,8,9}									
public	t1_range_int	range_p00		c2	f	0	4	-1				
-1			{0,1,2,3,4,5,6,7,8,9}									
public	t1_range_int	range_p00		c3	f	0	4	-1				
-1			{0,1,2,3,4,5,6,7,8,9}									
public	t1_range_int	range_p00		c4	f	0	4	-1				
-1			{0,1,2,3,4,5,6,7,8,9}									
public	t1_range_int	range_p01		c1	f	0	4	-1				
-1			{10,11,12,13,14,15,16,17,18,19}									
public	t1_range_int	range_p01		c2	f	0	4	-1				
-1			{10,11,12,13,14,15,16,17,18,19}									
public	t1_range_int	range_p01		c3	f	0	4	-1				
-1			{10,11,12,13,14,15,16,17,18,19}									
public	t1_range_int	range_p01		c4	f	0	4	-1				
-1			{10,11,12,13,14,15,16,17,18,19}									
public	t1_range_int	range_p02		c1	f	0	4	-1				
-1			{20,21,22,23,24,25,26,27,28,29}									
public	t1_range_int	range_p02		c2	f	0	4	-1				
-1			{20,21,22,23,24,25,26,27,28,29}									
public	t1_range_int	range_p02		c3	f	0	4	-1				
-1			{20,21,22,23,24,25,26,27,28,29}									
public	t1_range_int	range_p02		c4	f	0	4	-1				
-1			{20,21,22,23,24,25,26,27,28,29}									
public	t1_range_int	range_p03		c1	f	0	4	-1				
-1			{30,31,32,33,34,35,36,37,38,39}									
public	t1_range_int	range_p03		c2	f	0	4	-1				
-1			{30,31,32,33,34,35,36,37,38,39}									
public	t1_range_int	range_p03		c3	f	0	4	-1				
-1			{30,31,32,33,34,35,36,37,38,39}									
public	t1_range_int	range_p03		c4	f	0	4	-1				
-1			{30,31,32,33,34,35,36,37,38,39}									
public	t1_range_int	range_p04		c1	f	0	4	-1				

```

-1 | | | {40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | range_p04 | | c2 | f | 0 | 4 | -1 |
-1 | | | {40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | range_p04 | | c3 | f | 0 | 4 | -1 |
-1 | | | {40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | range_p04 | | c4 | f | 0 | 4 | -1 |
-1 | | | {40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | | | c1 | f | 0 | 4 | -1 | -1
| | |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | | | c2 | f | 0 | 4 | -1 | -1
| | |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | | | c3 | f | 0 | 4 | -1 | -1
| | |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
public | t1_range_int | | | c4 | f | 0 | 4 | -1 | -1
| | |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
(24 rows)

```

- 生成多列数据的分区级统计信息

```

gaussdb=# ALTER TABLE t1_range_int ADD STATISTICS ((c2, c3));
gaussdb=# ANALYZE t1_range_int WITH ALL;

```

- 查看多列数据的分区级统计信息

```

gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_ext_stats
WHERE tablename='t1_range_int' ORDER BY tablename,partitionname,attname;
schemaname | tablename | partitionname | subpartitionname | attname | inherited | null_frac |
avg_width | n_distinct | n_dndistinct | most_common_vals | most_common_freqs | histogram_bounds
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
public | t1_range_int | range_p00 | | | 2 3 | f | 0 | 8 | -1 |
-1 | | | |
public | t1_range_int | range_p01 | | | 2 3 | f | 0 | 8 | -1 |
-1 | | | |
public | t1_range_int | range_p02 | | | 2 3 | f | 0 | 8 | -1 |
-1 | | | |
public | t1_range_int | range_p03 | | | 2 3 | f | 0 | 8 | -1 |
-1 | | | |
public | t1_range_int | range_p04 | | | 2 3 | f | 0 | 8 | -1 |
-1 | | | |
public | t1_range_int | | | | 2 3 | f | 0 | 8 | -1 | -1
| | |
(6 rows)

```

- 创建表达式索引并生成对应的分区级统计信息

```

gaussdb=# CREATE INDEX t1_range_int_index ON t1_range_int(text(c1)) LOCAL;
gaussdb=# ANALYZE t1_range_int WITH ALL;

```

- 查看表达式索引的分区级统计信息

```

gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_stats WHERE
tablename='t1_range_int_index' ORDER BY tablename,partitionname,attname;
schemaname | tablename | partitionname | subpartitionname | attname | inherited |
null_frac | avg_width | n_distinct | n_dndistinct | most_common_vals | most_common_freqs
| histogram_bounds
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
public | t1_range_int_index | range_p00_text_idx | | text | f | 0 | 5
-1 | 0 | | | {0,1,2,3,4,5,6,7,8,9}

```

```
public | t1_range_int_index | range_p01_text_idx | | text | f | 0 | 6
| -1 | 0 | | | {10,11,12,13,14,15,16,17,18,19}
public | t1_range_int_index | range_p02_text_idx | | text | f | 0 | 6
| -1 | 0 | | | {20,21,22,23,24,25,26,27,28,29}
public | t1_range_int_index | range_p03_text_idx | | text | f | 0 | 6
| -1 | 0 | | | {30,31,32,33,34,35,36,37,38,39}
public | t1_range_int_index | range_p04_text_idx | | text | f | 0 | 6
| -1 | 0 | | | {40,41,42,43,44,45,46,47,48,49}
public | t1_range_int_index | | | text | f | 0 | 5 | -1
| 0 | | | |
{0,1,10,11,12,13,14,15,16,17,18,19,2,20,21,22,23,24,25,26,27,28,29,3,30,31,32,33,3
4,35,36,37,38,39,4,40,41,42,43,44,45,46,47,48,49,5,6,7,8,9}
(6 rows)
```

- 删除分区表

```
gaussdb=# DROP TABLE t1_range_int;
```

### 5.3.3.2 分区级统计信息

#### 指定单分区统计信息收集

当前分区表支持指定单分区统计信息收集，已收集统计信息的分区会在再次收集时自动更新维护。该功能适用于列表分区、哈希分区和范围分区。

```
gaussdb=# CREATE TABLE only_fisrt_part(id int,name varchar)PARTITION BY RANGE (id)
(PARTITION id11 VALUES LESS THAN (1000000),
PARTITION id22 VALUES LESS THAN (2000000),
PARTITION max_id1 VALUES LESS THAN (MAXVALUE));
```

```
gaussdb=# INSERT INTO only_fisrt_part SELECT generate_series(1,5000),'test';
```

```
gaussdb=# ANALYZE only_fisrt_part PARTITION (id11);
gaussdb=# ANALYZE only_fisrt_part PARTITION (id22);
gaussdb=# ANALYZE only_fisrt_part PARTITION (max_id1);
```

```
gaussdb=# SELECT relname, relpages, reltuples FROM pg_partition WHERE relname IN ('id11', 'id22',
'max_id1');
```

```
relname | relpages | reltuples
+-----+-----+-----+
id11 | 3400 | 5000
id22 | 0 | 0
max_id1 | 0 | 0
(3 rows)
```

```
gaussdb=# \x
gaussdb=# SELECT * FROM pg_stats WHERE tablename = 'only_fisrt_part' AND partitionname = 'id11';
-[ RECORD 1 ]-----
```

```
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
schemaname | public
tablename | only_fisrt_part
attname | name
inherited | f
null_frac | 0
avg_width | 5
n_distinct | 1
n_dndistinct | 0
most_common_vals | {test}
most_common_freqs | {1}
histogram_bounds |
correlation | 1
most_common_elems |
most_common_elem_freqs |
elem_count_histogram |
partitionname | id11
```



```
-> Partition Iterator (cost=0.00..1.18 rows=1 width=12)
    Iterations: 1
    -> Partitioned Seq Scan on only_first_part_two (cost=0.00..1.18 rows=1 width=12)
        Filter: (c2 = 0)
        Selected Partitions: 1
(7 rows)

gaussdb=# DROP TABLE only_fisrt_part_two;
```

## 通过改写分区子句剪枝优化

当没有分区级统计信息时，在优化器行数估算模块，通过在逻辑上对分区子句进行伪谓词的改写，利用改写后的伪谓词影响选择率的计算和整表的统计信息获取一个比较准确的行数估算值。

### 📖 说明

- 特性只作用于选择率的计算。
- 特性不支持二级分区。
- 特性只支持范围分区（range partition）、列表分区（list partition）。
- 对于范围分区，只支持单列分区键的改写，不支持多列分区键的改写。
- 对于列表分区，出于性能考虑，设置列表指定分区的枚举值个数的阈值为40个。
  - 当指定分区的列表枚举值个数超过40时，本特性不再适用。
  - 对于default分区，其列表枚举值个数是所有非default分区的枚举值个数的总和。

### 示例1：对于范围分区的改写

```
gaussdb=# CREATE TABLE test_int4_maxvalue(id INT, name VARCHAR)
PARTITION BY RANGE(id)
(
    PARTITION id1 VALUES LESS THAN(1000),
    PARTITION id2 VALUES LESS THAN(2000),
    PARTITION max_id VALUES LESS THAN(MAXVALUE)
);
gaussdb=# INSERT INTO test_int4_maxvalue SELECT GENERATE_SERIES(1,5000),'test';
gaussdb=# ANALYZE test_int4_maxvalue with global;

-- 查询指定分区id1
gaussdb=# EXPLAIN SELECT * FROM test_int4_maxvalue PARTITION(id1);
          QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
 Node/s: All datanodes

Remote SQL: SELECT id, name FROM public.test_int4_maxvalue PARTITION (id1)
Datanode Name: d1_datanode1
Partition Iterator (cost=0.00..7.91 rows=491 width=9)
 Iterations: 1
 -> Partitioned Seq Scan on test_int4_maxvalue (cost=0.00..7.91 rows=491 width=9)
    Selected Partitions: 1

Datanode Name: d1_datanode2
Partition Iterator (cost=0.00..8.08 rows=508 width=9)
 Iterations: 1
 -> Partitioned Seq Scan on test_int4_maxvalue (cost=0.00..8.08 rows=508 width=9)
    Selected Partitions: 1

(16 rows)

-- 查询指定分区max_id
gaussdb=# EXPLAIN SELECT * FROM test_int4_maxvalue PARTITION(max_id);
          QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
```

```
Node/s: All datanodes

Remote SQL: SELECT id, name FROM public.test_int4_maxvalue PARTITION (max_id)
Datanode Name: d1_datanode1
Partition Iterator (cost=0.00..24.46 rows=1546 width=9)
  Iterations: 1
  -> Partitioned Seq Scan on test_int4_maxvalue (cost=0.00..24.46 rows=1546 width=9)
      Selected Partitions: 3

Datanode Name: d1_datanode2
Partition Iterator (cost=0.00..23.55 rows=1455 width=9)
  Iterations: 1
  -> Partitioned Seq Scan on test_int4_maxvalue (cost=0.00..23.55 rows=1455 width=9)
      Selected Partitions: 3

(16 rows)

-- 删除分区表
gaussdb=# DROP TABLE test_int4_maxvalue;
```

### 示例2: 对于列表分区的改写

```
gaussdb=# CREATE TABLE test_default
(
  c1 INT,
  c2 BIGINT
)
PARTITION BY LIST(c2)
(
  PARTITION p_1 VALUES (10000, 20000),
  PARTITION p_2 VALUES (300000, 400000, 500000),
  PARTITION p_3 VALUES (DEFAULT)
);
gaussdb=# INSERT INTO test_default SELECT GENERATE_SERIES(1, 1000), 10000;
gaussdb=# INSERT INTO test_default SELECT GENERATE_SERIES(1001, 2000), 600000;
gaussdb=# ANALYZE test_default with global;

-- 查询指定分区p_1
gaussdb=# EXPLAIN SELECT * FROM test_default PARTITION(p_1);
          QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes

Remote SQL: SELECT c1, c2 FROM public.test_default PARTITION (p_1)
Datanode Name: d1_datanode1
Partition Iterator (cost=0.00..7.92 rows=492 width=12)
  Iterations: 1
  -> Partitioned Seq Scan on test_default (cost=0.00..7.92 rows=492 width=12)
      Selected Partitions: 1

Datanode Name: d1_datanode2
Partition Iterator (cost=0.00..8.08 rows=508 width=12)
  Iterations: 1
  -> Partitioned Seq Scan on test_default (cost=0.00..8.08 rows=508 width=12)
      Selected Partitions: 1

(16 rows)

-- 查询指定分区p_3
gaussdb=# EXPLAIN SELECT * FROM test_default PARTITION(p_3);
          QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes

Remote SQL: SELECT c1, c2 FROM public.test_default PARTITION (p_3)
Datanode Name: d1_datanode1
Partition Iterator (cost=0.00..8.24 rows=524 width=12)
  Iterations: 1
  -> Partitioned Seq Scan on test_default (cost=0.00..8.24 rows=524 width=12)
```

```
Selected Partitions: 3

Datanode Name: d1_datanode2
Partition Iterator (cost=0.00..7.76 rows=476 width=12)
  Iterations: 1
  -> Partitioned Seq Scan on test_default (cost=0.00..7.76 rows=476 width=12)
    Selected Partitions: 3

(16 rows)

-- 删除分区表
gaussdb=# DROP TABLE test_default;
```

## 5.3.4 Partition-wise Join

Partition-wise Join是一种分区级并行的优化技术，是指在符合一定条件的情况下，将两张表之间的Join，分解为两张表中对应的两个分区之间的Join。通过并发执行、减少数据通信量等方式，提升分区表的Join查询的性能。

Partition-wise Join分为SMP场景和非SMP场景。

### 5.3.4.1 非 SMP 场景下的 Partition-wise Join

在非SMP场景下，Partition-wise Join的路径是基于规则生成的，即只要符合条件，即可生成Partition-wise Join路径，而无需对比路径代价。其开关为GUC参数enable\_partitionwise。

## 使用规格

非SMP场景下的Partition-wise Join的使用规格：

- 只支持一级RANGE分区。
- 支持Hash Join、Nestloop Join、Merge Join。
- 只支持Inner Join。
- 需要设置query\_dop的值为1。
- 由于非SMP场景下的Partition-wise Join为规则选择，所以Partition-wise Join计划可能造成性能下降，需要用户自行决定是否启用。
- 仅支持FQS计划。

## 示例

```
--创建Range分区表。
gaussdb=# CREATE TABLE range_part (
gaussdb(#   a INTEGER,
gaussdb(#   b INTEGER,
gaussdb(#   c INTEGER
gaussdb(# ) PARTITION BY RANGE (a)
gaussdb-# (
gaussdb(# PARTITION range_part_p1 VALUES LESS THAN (10),
gaussdb(# PARTITION range_part_p2 VALUES LESS THAN (20),
gaussdb(# PARTITION range_part_p3 VALUES LESS THAN (30),
gaussdb(# PARTITION range_part_p4 VALUES LESS THAN (40)
gaussdb(# );

--使用FQS计划。
gaussdb=# SET enable_fast_query_shipping= ON;
SET

--设置query_dop为1，关闭SMP。
gaussdb=# SET query_dop = 1;
```



```
SET
--关闭非SMP场景下的Partition-wise Join开关。
gaussdb=# SET enable_partitionwise = off;
SET

--查看非Partition-wise Join执行计划。
gaussdb=# SET max_datanode_for_plan = 1;
SET
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part t1 INNER JOIN range_part t2 ON (t1.a = t2.a);
          QUERY PLAN
-----
Data Node Scan
  Node/s: All datanodes

Remote SQL: SELECT t1.a, t1.b, t1.c, t2.a, t2.b, t2.c FROM public.range_part t1 JOIN public.range_part t2 ON
t1.a = t2.a
Datanode Name: datanode1
Hash Join
  Hash Cond: (t1.a = t2.a)
  -> Partition Iterator
    Iterations: 4
    -> Partitioned Seq Scan on range_part t1
      Selected Partitions: 1..4
  -> Hash
    -> Partition Iterator
      Iterations: 4
    -> Partitioned Seq Scan on range_part t2
      Selected Partitions: 1..4

(17 rows)

--打开非SMP场景下的Partition-wise Join开关。
gaussdb=# SET enable_partitionwise = on;
SET

--查看非SMP场景下的Partition-wise Join计划。从执行计划中可以看到，Partition Iterator算子被提到了Hash
Join算子的上层。计算方式由原来的依次扫描完所有分区的数据之后再行Join，改为了每次扫描一对分区，进行
Join，再依次遍历下一个分区。
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part t1 INNER JOIN range_part t2 ON (t1.a = t2.a);
          QUERY PLAN
-----
Data Node Scan
  Node/s: All datanodes

Remote SQL: SELECT t1.a, t1.b, t1.c, t2.a, t2.b, t2.c FROM public.range_part t1 JOIN public.range_part t2 ON
t1.a = t2.a
Datanode Name: datanode1
Result
  -> Partition Iterator
    Iterations: 4
    -> Hash Join
      Hash Cond: (t1.a = t2.a)
      -> Partitioned Seq Scan on range_part t1
        Selected Partitions: 1..4
    -> Hash
      -> Partitioned Seq Scan on range_part t2
        Selected Partitions: 1..4

(16 rows)

-- 删除分区表
gaussdb=# DROP TABLE range_part;
```

### 5.3.4.2 SMP 场景下的 Full Partition-wise Join

SMP场景下的Partition-wise Join计划是基于代价选择的，在路径生成的过程中，会对比Partition-wise Join和非Partition-wise Join路径的估算代价，选择代价较低的路径。其开关参数为GUC参数enable\_smp\_partitionwise。

Full Partition-wise Join是指相互Join的两张表为分区策略完全相同的两张分区表，Full Partition-wise Join路径生成条件是两张表的分区键是一对相互匹配的Join key。

## 使用规格

SMP场景下的Full Partition-wise Join的使用规格：

- 支持一级HASH分区表和一级RANGE分区表。
- Hash分区表的分区策略完全相同是指分区键类型相同、分区数相同。
- Range分区表的分区策略完全相同是指分区键类型相同、分区数相同、分区键数量相同、每个分区的边界值相同。
- 仅支持Stream计划。
- 仅支持分区键和分布键完全一致的场景。
- 仅支持Join算子在单DN内完成计算，即Join算子的数据不跨节点。
- 支持Hash Join和Merge Join。
- 支持Seqscan、Indexscan、Indexonlyscan、Imcvscan。其中，对于Indexscan和Indexonlyscan，只支持分区Local索引，且索引类型为BTREE或UBTREE。
- 相关规格继承SMP规格，不支持SMP场景下的IUD操作。
- 需要开启SMP功能，且设置query\_dop的值大于1。

## 示例

```
--创建Hash分区表。
gaussdb=# CREATE TABLE hash_part
(
  a INTEGER,
  b INTEGER,
  c INTEGER
)
DISTRIBUTE BY HASH(a)
PARTITION BY HASH(a)
(
  PARTITION p1,
  PARTITION p2,
  PARTITION p3,
  PARTITION p4,
  PARTITION p5
);
CREATE TABLE

--使用Stream计划。
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET enable_stream_operator = on;
SET

--设置query_dop为5，开启SMP。
gaussdb=# SET query_dop = 5;
SET

--关闭SMP场景下的Partition-wise Join开关。
gaussdb=# SET enable_smp_partitionwise = off;
```

```
SET

--查看非Partition-wise Join的计划。从计划中可以看出，在通过Partition Iterator+Partitioned Seq Scan两层算子完成数据扫描之后，通过Streaming(type: LOCAL REDISTRIBUTE)算子对数据进行了一次重分布，用于保证Join算子中数据能够相互匹配。
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM hash_part t1, hash_part t2 WHERE t1.a = t2.a;
QUERY PLAN
-----
Streaming (type: GATHER)
Node/s: All datanodes
-> Streaming(type: LOCAL GATHER dop: 1/5)
   Spawn on: All datanodes
   -> Nested Loop
       Join Filter: (t1.a = t2.a)
       -> Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
           Spawn on: All datanodes
           -> Partition Iterator
               Iterations: 5
               -> Partitioned Seq Scan on hash_part t1
                   Selected Partitions: 1..5
           -> Materialize
               -> Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
                   Spawn on: All datanodes
                   -> Partition Iterator
                       Iterations: 5
                       -> Partitioned Seq Scan on hash_part t2
                           Selected Partitions: 1..5

(19 rows)

--打开SMP场景下的Partition-wise Join开关。
gaussdb=# SET enable_smp_partitionwise = on;
SET

--查看Partition-wise Join的执行计划。从计划中可以看出，Partition-wise Join计划消除了Streaming算子，即数据不再需要在线程之间重新分布，减少了数据搬运的开销，提升了Join操作的性能。
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM hash_part t1, hash_part t2 WHERE t1.a = t2.a;
QUERY PLAN
-----
Streaming (type: GATHER)
Node/s: All datanodes
-> Streaming(type: LOCAL GATHER dop: 1/5)
   Spawn on: All datanodes
   -> Hash Join (Partition-wise Join)
       Hash Cond: (t1.a = t2.a)
       -> Partition Iterator
           Iterations: 5
           -> Partitioned Seq Scan on hash_part t1
               Selected Partitions: 1..5
       -> Hash
           -> Partition Iterator
               Iterations: 5
               -> Partitioned Seq Scan on hash_part t2
                   Selected Partitions: 1..5

(15 rows)

-- 删除分区表
gaussdb=# DROP TABLE hash_part;
```

### 说明

仅在SMP场景下的Partition-wise Join计划中，Join算子右侧会有(Partition-wise Join)提示信息。非SMP场景无该提示信息。

## 5.4 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。

### 5.4.1 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。
- 新增分区不继承表上的分类索引属性。

### 5.4.1.1 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，请参见[对范围分区表分割分区](#)。

### 5.4.1.2 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。

## 5.4.2 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。
- 删除分区时，如果该分区上带有仅属于当前分区的分类索引时，则会级联删除分类  
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。

### 5.4.3 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。

#### 须知

- 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。
- 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;
```

## 5.4.4 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');
```

## 5.4.5 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。
- 分割的目标分区如果包含分类索引时，该分区不支持分割。
- 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。

### 5.4.5.1 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。

### 5.4.5.2 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。

## 5.4.6 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。
- 合并前的分区如果包含分类索引则不支持合并。

#### 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;
```

## 5.4.7 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;
```

## 5.4.8 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。

### 5.4.8.1 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;
```

### 5.4.8.2 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。

## 5.4.9 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';
```

## 5.4.10 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。

### 5.4.10.1 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;
```

可以通过如下命令重建索引range\_sales\_idx。

```
ALTER INDEX range_sales_idx REBUILD;
```

### 5.4.10.2 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;
```

## 5.5 分区表系统视图&DFX

## 5.5.1 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。

## 5.5.2 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, storage_type=USTORE,
segment=off)
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;
+
CREATE INDEX idx_range_a ON test_range_pt USING ubtree (a) LOCAL(PARTITION p1_a_idx,
PARTITION p2_a_idx, PARTITION p3_a_idx, PARTITION p4_a_idx, PARTITION p5_a_idx) WITH
(storage_type=USTORE) TABLESPACE pg_default;
(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_fileno(partition_oid)`获取到指定分区表的OID所对应的fileno。  
查看分区p1的fileno。

```
SELECT pg_partition_fileno(49294);
pg_partition_fileno
-----
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)
```

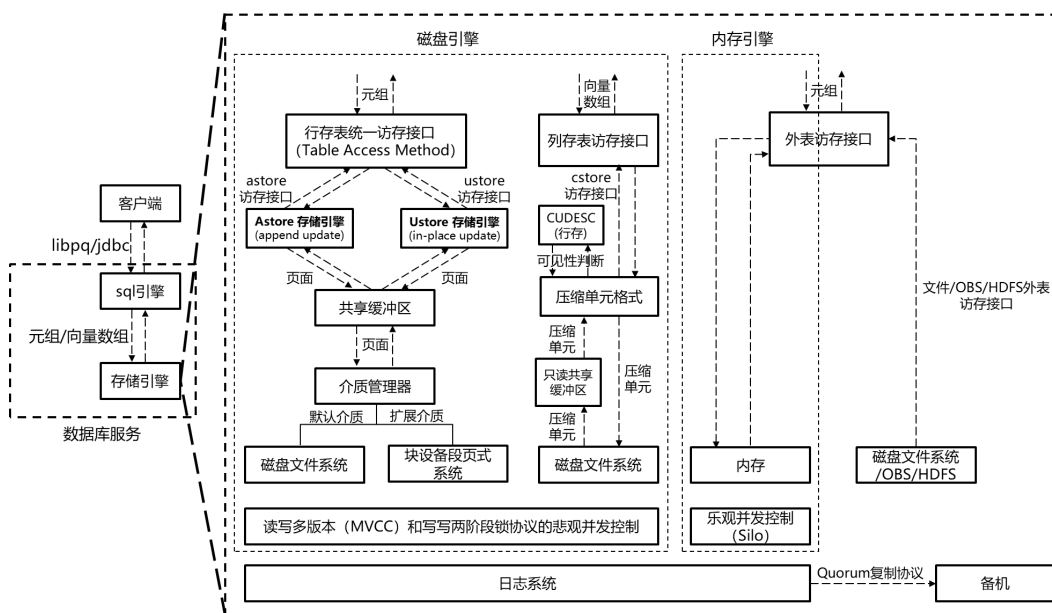
# 6 存储引擎

## 6.1 存储引擎体系架构

### 6.1.1 存储引擎体系架构概述

#### 6.1.1.1 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行存引擎可以选择基于Append update 的Astore或基于In-place update的Ustore。

### 6.1.1.2 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采用的是物理日志，其写入/传输/回放对存储引擎透明。

### 6.1.2 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, segment=off
```

```
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, segment=off

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 | test | table | z7ee88f3a | 0 bytes | {orientation=row,compression=no,storage_type=USTORE,segment=off} |
 public | test1 | table | z7ee88f3a | 0 bytes | {orientation=row,storage_type=ustore,compression=no,segment=off} |
 public | test2 | table | z7ee88f3a | 0 bytes | {orientation=row,storage_type=astore,compression=no} |
 public | test3 | table | z7ee88f3a | 16 kB | {orientation=column,storage_type=astore,compression=low} |
 public | test4 | table | z7ee88f3a | 0 bytes | {orientation=row,compression=no,storage_type=USTORE,segment=off} |
(5 rows)

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

gaussdb=# DROP TABLE test;
gaussdb=# DROP TABLE test1;
gaussdb=# DROP TABLE test2;
gaussdb=# DROP TABLE test4;
```

## 6.1.3 存储引擎更新说明

### 6.1.3.1 GaussDB 内核 505 版本

- Ustore支持柔性字段高效存储。
- Ustore支持Toast规模商用。
- Ustore增加页面恢复与逃生技术。
- Ustore支持SMP技术。

### 6.1.3.2 GaussDB 内核 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，Diagnose 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。

### 6.1.3.3 GaussDB 内核 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。

## 6.2 Astore 存储引擎

### 6.2.1 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。

---

#### 注意

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。
5. 支持回收站（闪回DROP、闪回Truncate）功能。

## 6.3 Ustore 存储引擎

### 6.3.1 Ustore 简介

Ustore（Unified Storage）是GaussDB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。

#### 使用 Ustore 的优势

- 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，空间膨胀可控。
- B-tree索引增加了事务信息，能够独立进行MVCC，增加了IndexOnlyScan的比例，大大减少回表次数。
- 不依赖Vacuum进行旧版本清理。独立的空间回收能力，索引与堆表解耦，可独立清理，IO平稳度更优。
- 大并发更新同一行的场景，相对于Astore的ROWID会偏移，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。
- 支持闪回功能。

---

#### 注意

Ustore DML在修改数据页面时，也需要同步生成Undo，因此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。

---

#### 6.3.1.1 Ustore 特性与规格

##### 6.3.1.1.1 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	×

##### 6.3.1.1.2 存储规格

1. 数据表最大列数不能超过1600列。

2. `init_td` (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`) 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。
3. 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 $8192 - \text{MAXALIGN}(56+128*26+4) = 4800\text{B}$ 。
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。

### 6.3.1.2 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
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, compression=no, segment=off
```

#### 删除Ustore表

```
gaussdb=# DROP TABLE ustore_table;
DROP TABLE
```

#### 为Ustore表创建索引

Ustore当前仅支持BTree类型的多版本索引。在一些场景中, 为了区别于Astore的BTree索引, 也会将Ustore表的多版本BTree索引称为UBTree (Ustore BTree, UBTree介绍详见UBTree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UBTree索引。

Ustore表不指定创建索引类型, 默认创建的是UBTree索引。

**注意**

UBTree索引分为RCR版本和PCR版本，默认创建RCR版本的UBTree。若在创建索引时with选项指定(index\_txntype=pcr)或者指定GUC的index\_txntype=pcr，则创建的是PCR版本的UBTree。

```
gaussdb=# CREATE TABLE test(a int);
CREATE TABLE
gaussdb=# CREATE INDEX UB_tree_index ON test(a);
CREATE INDEX
gaussdb=# \d+ test
Table "public.test"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
a      | integer |          | plain   |              |
Indexes:
    "ub_tree_index" ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off

--删除Ustore表索引。
gaussdb=# DROP TABLE test;
DROP TABLE
```

### 6.3.1.3 Ustore 的最佳实践

#### 6.3.1.3.1 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。wait available td是wait\_status的值之一，wait\_status表示当前线程的等待状态，包含等待状态详细信息。通过PG\_THREAD\_WAIT\_STATUS视图可以查询wait\_status的值（none表示没在等待任意事件，如果有等待事件即可看到对应wait available td的值），示例如下。init\_td的配置和详细描述参见《开发指南》的“SQL参考 > SQL语法 > CREATE TABLE”章节。init\_td查看和修改方法的具体示例如下：

```
gaussdb=# CREATE TABLE test1(name varchar) WITH(storage_type = ustore, init_td=2);
gaussdb=# \d+ test1
Table "public.test1"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
name   | character varying |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, init_td=2, compression=no, segment=off,
toast.storage_type=ustore, toast.toast_storage_type=enhanced_toast

gaussdb=# ALTER TABLE test1 SET(init_td=8);
```

```

gaussdb=# \d+ test1
                Table "public.test1"
  Column | Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 name   | character varying |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, compression=no, segment=off, init_td=8,
toast.storage_type=ustore, toast.toast_storage_type=enhanced_toast

gaussdb=# SELECT * FROM pg_thread_wait_status;
 node_name | db_name | thread_name | query_id | tid | sessionid | lwtid |
 psessionid | tlevel | smpid | wait_status | wait_event | locktag | lo
 ckmode | block_sessionid | global_sessionid
-----+-----+-----+-----+-----+-----+-----+-----
 sgnode   |        | PageWriter |          |    |          |      |
 |         | 0 | 0 | none | none |        |    |
 |         | 0:0#0
 sgnode   |        | PageWriter |          |    |          |      |
 |         | 0 | 0 | none | none |        |    |
 |         | 0:0#0
 sgnode   |        | PageWriter |          |    |          |      |
 |         | 0 | 0 | none | none |        |    |
 |         | 0:0#0
 sgnode   |        | PageWriter |          |    |          |      |
 |         | 0 | 0 | none | none |        |    |
 |         | 0:0#0
 sgnode   |        | PageWriter |          |    |          |      |
 |         | 0 | 0 | none | none |        |    |
 |         | 0:0#0

gaussdb=# DROP TABLE test1;
DROP TABLE
    
```

### 6.3.1.3.2 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的配置和详细描述参见《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非定长更新操作，建议为不调整页面填充率或者将页面填充率数值调整更小，以减少跨页更新带来的性能损耗。fillfactor查看和修改方法具体示例如下：

```

gaussdb=# CREATE TABLE test(a int) with(fillfactor=100);
gaussdb=# \d+ test
                Table "public.test"
  Column | Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a      | integer      |          | plain   |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, fillfactor=100, compression=no, storage_type=USTORE, segment=off

gaussdb=# ALTER TABLE test set(fillfactor=92);
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, segment=off, fillfactor=92

gaussdb=# DROP TABLE test;
DROP TABLE
```

### 6.3.1.3.3 在线校验功能

在线校验是Ustore特有的，在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UPAGE:UBTREE:UNDO三个模块校验。业务现网请保持开启，性能场景除外。

关闭：

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

打开：

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

### 6.3.1.3.4 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，输入有效单位为s,min,h,d。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的业务模型的取值：

使用Undo统计信息的系统函数gs\_stat\_undo，如果入参为false，输出undo\_space\_limit\_size、undo\_limit\_size\_per\_transaction、undo\_retention\_time参数的合理化建议。具体详细参数值参见《开发指南》的“SQL参考 > 函数和操作符 > undo系统函数”章节。

#### 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。

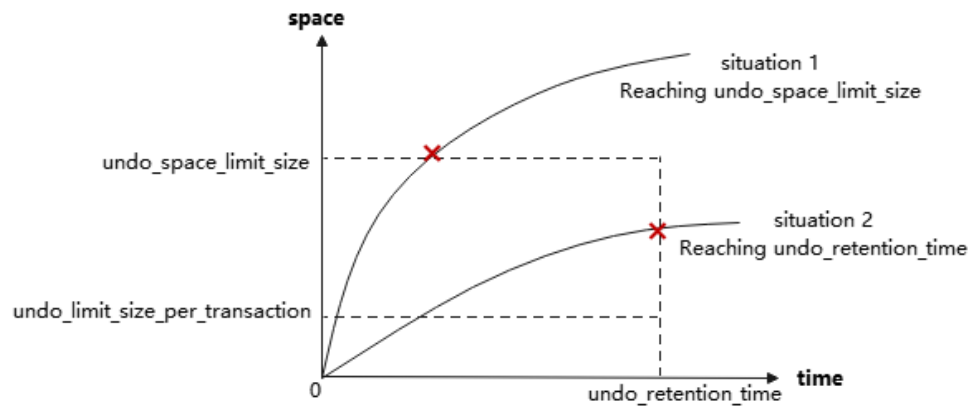
在磁盘空间允许的条件下，推荐undo\_space\_limit\_size设置翻倍。同时undo\_space\_limit\_size的取值越大则占用磁盘空间越大，可能降低性能。如果查询视图系统函数gs\_stat\_undo的curr\_used\_undo\_size发现不存在Undo空间膨胀，可以恢复为原值。

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。

#### 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。





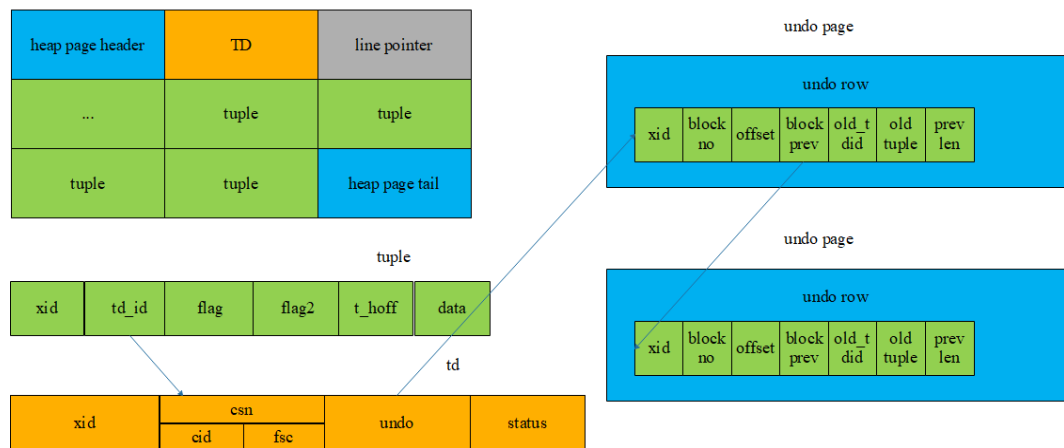
例如：Undo强制回收阈值参数undo\_space\_limit\_size设置为1GB，Undo旧版本保留时间undo\_retention\_time为900s。如果900s内产生的历史版本数据不足1GB\*0.8，则按照900s进行回收限制；否则按照1GB\*0.8进行回收限制。遇到该情况时，如果磁盘空闲空间充足，则上调undo\_space\_limit\_size，如果磁盘空闲空间紧缺，则下调undo\_retention\_time。

## 6.3.2 存储格式

### 6.3.2.1 RCR Uheap

#### 6.3.2.1.1 RCR Uheap 多版本管理

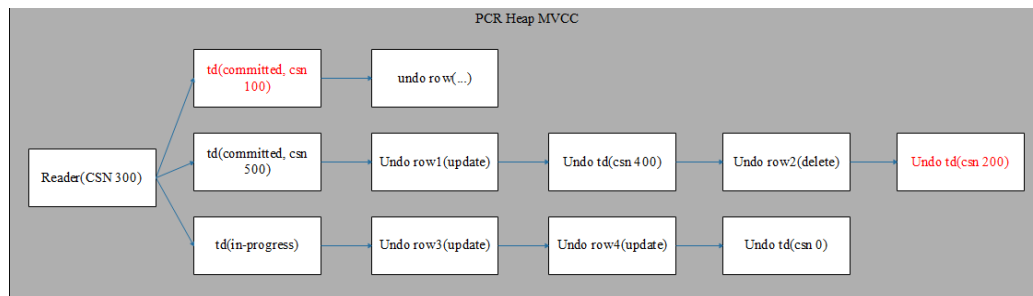
Ustore对其使用的heap做了如下重要的增强，简称Uheap。



Ustore RCR(Row Consistency Read)的多版本管理是基于数据行的行级多版本管理。不过Ustore将XID记录在了页面的TD(Transaction Directory)区域区别于常见的将XID存储在数据行上，节省了页面空间。事务修改记录时，会将历史数据记录到Undo Row中，在Tuple中的td\_id指向的TD槽上记录产生的Undo Row地址(zone\_id, block no, page offset)，并将新的数据覆盖写入页面。访问元组时，沿着版本链还原该元组，直到找到自己对应的版本。

#### 6.3.2.1.2 RCR Uheap 可见性机制

Ustore可见性判断是通过构建数据行的一致性版本获得的，老快照可通过Undo记录获取历史版本。



### 6.3.2.1.3 RCR Uheap 空闲空间管理

Ustore使用Free Space Map（FSM）文件记录了每个数据页的潜在空闲空间，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颗有三层节点的树，树的节点在逻辑上是堆顶关系。每次在FSM上查找时从根节点进行，一直查找到叶子节点，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。当用户执行插入类型的DML语句，类似Insert/Non-Inplace Update(新页面)/Multi Insert时，会查询FSM结构，寻找到一个可以插入当前记录的空间。用户执行完DML操作后会根据当前页面的潜在空闲空间与实际空闲空间的差值来决定是否将该页面的空闲空间刷新到FSM上。该差值越大，即潜在空间大于实际空间越多，则该页面被更新至FSM的几率越大。FSM上会记录数据页的潜在空闲空间，在用户执行插入操作找到一个页面时，如果该页面上的空闲空间较大则直接插入，否则如果潜在空间较大则对页面执行清理后插入。最后如果空间不够则重新搜索FSM结构或者拓展总页面数量。更新FSM结构主要有以下几个位置，DML、页面清理、vacuum、拓展页面、分区合并、页面扫描等。

### 6.3.2.2 UBTree

其使用的btree做了如下重要的增强，简称Ubtree。

- Ubtree索引增加了事务信息，能够独立进行MVCC；增加了IndexOnlyScan的比例，大大减少回表次数。
- 不依赖Vacuum进行旧版本清理。独立的空间回收能力，索引与堆表解耦，可独立清理，IO平稳度更优。

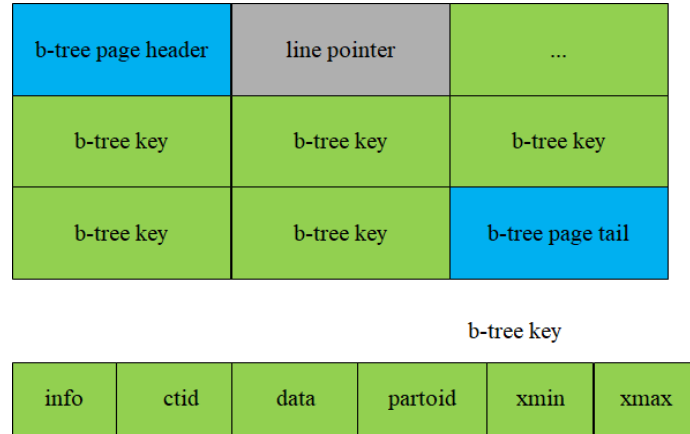
#### 6.3.2.2.1 RCR UBTree

### RCR UBTree 多版本管理

RCR(Row Consistency Read) btree 的多版本管理是基于数据行的行级多版本管理。将XID记录在了数据行上，会增加Key的大小，索引会有5-20%左右的膨胀。最新版本和历史版本均在btree上，索引没有记录Undo信息。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。索引分裂时，多版本信息随着key的迁移而迁移，如图6-1所示。

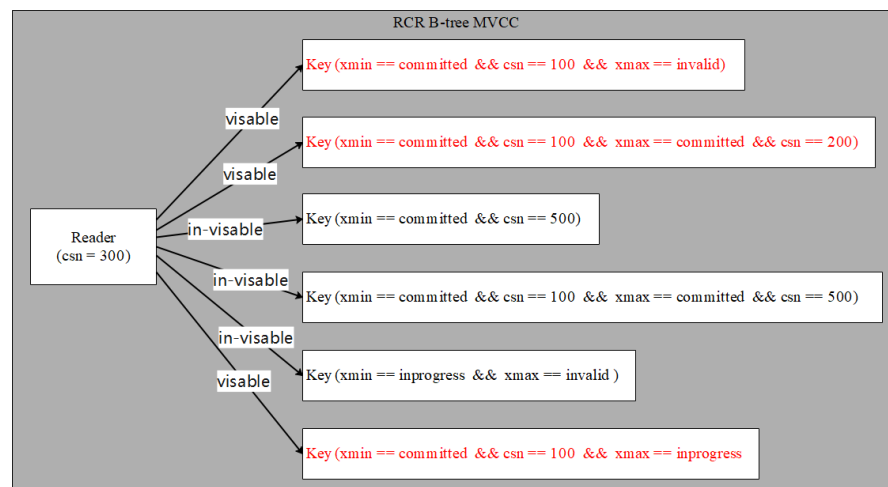
图 6-1 RCR UBTree 多版本管理



## RCR UBTree 可见性机制

RCR UBTree可见性判断是通过判断Key上xmin/xmax来确定的，与Astore堆表数据行上xmin/xmax作用类似，如图6-2所示。

图 6-2 RCR UBTree 可见性机制



## RCR UBTree 增删改查

- **Insert操作**: Ubtrees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**: Ubtrees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Ubtrees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，Ubtrees在数据发生更新时的处理如图6-3所示。

图 6-3 UBTREE 在数据发生更新时的处理

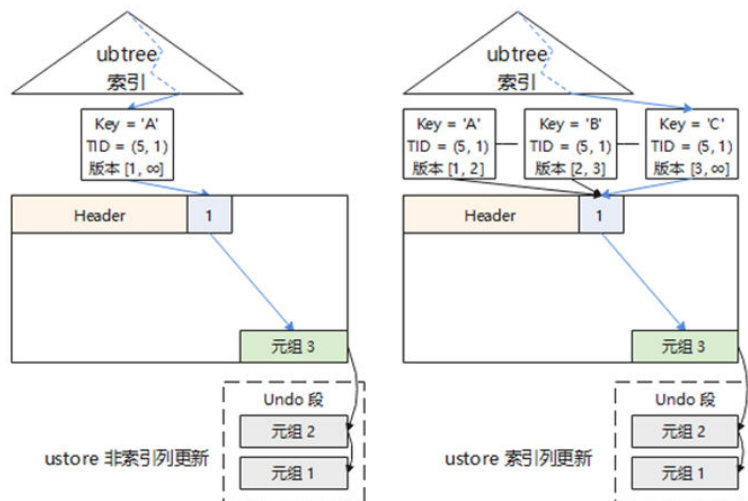


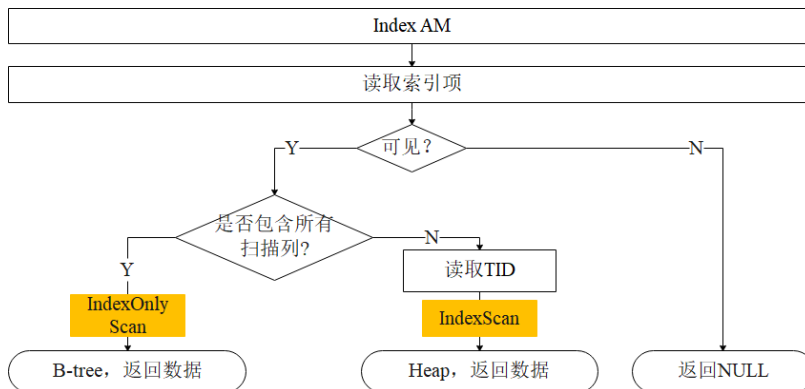
图6-3展示Ubtree在索引列和非索引列更新的差异：

- 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。
- 在索引列更新的情况下，Ubtree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，Ubtree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：

- 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。
- 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。如图4 对应的数据元组所示。

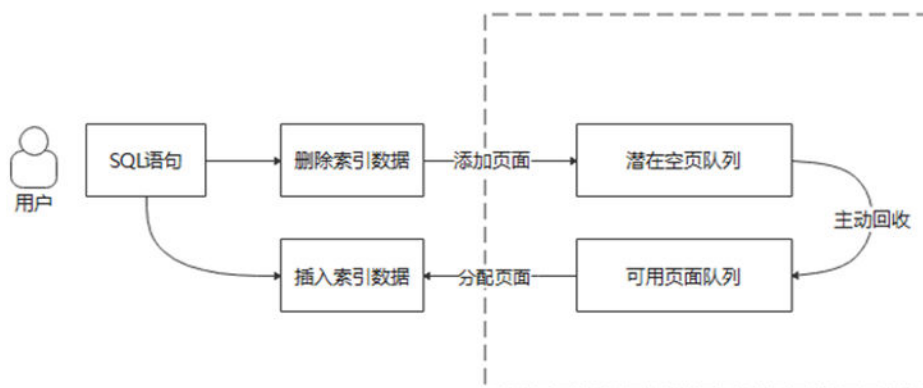
图 6-4 对应的数据元组



## RCR UBTree 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（UBTree 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。

图 6-5 索引页面在双循环队列间流动



如图6-5所示，索引页面在双循环队列间流动如下：

### 1. 索引空页流动到潜在队列

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. 潜在队列流动到可用队列

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，建议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. 可用队列流动到索引空页

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。

## 6.3.2.2.2 PCR UBTree

相比于RCR版本的UBTree，PCR版本的UBTree有以下特点。

- 索引元组的事务信息统一由TD槽进行管理。
- 增加了Undo操作，插入和删除前需要先写入Undo，事务abort时需要进行回滚操作。
- 支持闪回。

PCR UBTree通过在创建索引时with选项设置“index\_txntype=pcr”或者设置GUC参数“index\_txntype=pcr”进行创建。若没有显示指定with选项或者GUC，则默认创建

RCR版本的UBTree。当前PCR版本的UBTree不支持在线进行创建、极致RTO回放和备机读的功能。

注意，当前版本PCR索引在大数据量的回滚上耗时可能较长（回滚时间随数据量增长可能呈指数型增长，数据量太大可能会导致回滚未完成），回滚时间会在下个版本进行优化。以下是当前版本回滚时间的具体规格：

表 6-1 PCR 索引回滚时间的规格

类型/数据量	100	1000	1万	10万	100万
带PCR索引的回滚时间	0.692 ms	9.610 ms	544.678 ms	52,963.754 ms	89,440,029.048 ms
不带PCR索引的回滚时间	0.226 ms	0.916 ms	8.974 ms	94.903 ms	1206.177 ms
两者比值	3.06	10.49	60.70	558.08	74,151.66

## PCR UBTree 多版本管理

与RCR UBTree的区别是，PCR(Page Consistency Read) 的多版本管理是基于页面的多版本管理，所有元组的事务信息统一由TD槽进行管理。

## PCR UBTree 可见性机制

PCR UBTree可见性判断是通过把页面回滚到快照可见的时刻得到元组全部可见的页面完成的。

## PCR UBTree 增删改查

- **Insert操作**：操作与RCR UBTree基本一致，区别是：插入前需要先申请TD和写入Undo。
- **Delete操作**：操作与RCR UBTree基本一致，区别是：删除前需要先申请TD和写入Undo。
- **Update操作**：操作与RCR UBTree无区别，均转换为一条Delete操作和一条Insert操作。
- **Scan操作**：操作与RCR UBTree基本一致，区别是：查询操作需要将页面复制一个CR页面出来，将CR页面回滚到扫描快照可见的状态，从而整个页面的元组对于快照都是可见版本。

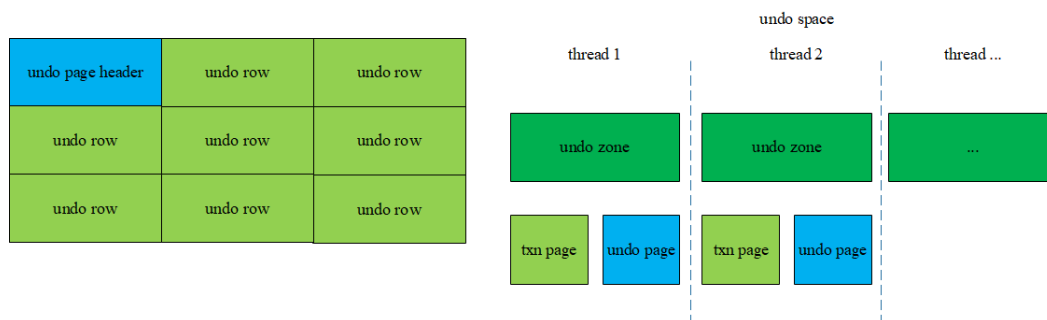
## PCR UBTree 空间管理

空间管理操作与RCR UBTree基本一致，区别是：PCR UBTree支持闪回，所以页面可回收的时间点由OldestXmin改成了GlobalRecycleXid。

### 6.3.2.3 Undo

历史版本数据集中存放在`$node_dir/undo`目录中，其中`$node_dir`为数据库节点路径，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。

#### 6.3.2.3.1 回滚段管理



1. 每个undo zone除了管理部分transaction page（用于存储事务回滚的元数据）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。

#### 6.3.2.3.2 文件组织结构

如需查询当前回滚段使用的存储方式是页式或段页式，可以查询系统表。当前仅支持页式。

示例：

```
gaussdb=# SELECT * FROM gs_global_config where name like '%undostorage%';
 name | value
-----+-----
undostorage | page
(1 row)
```

- 当回滚段使用的存储方式为页式：
  - txn page所在文件组织结构：  
`$node_dir/undo/{permanent|unlogged|temp}/$undo_zone_id.meta.$segno`
  - undo row所在文件组织结构：  
`$node_dir/undo/{permanent|unlogged|temp}/$undo_zone_id.$segno`

#### 6.3.2.3.3 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。



## 6.3.2.4 Enhanced Toast

### 6.3.2.4.1 概述

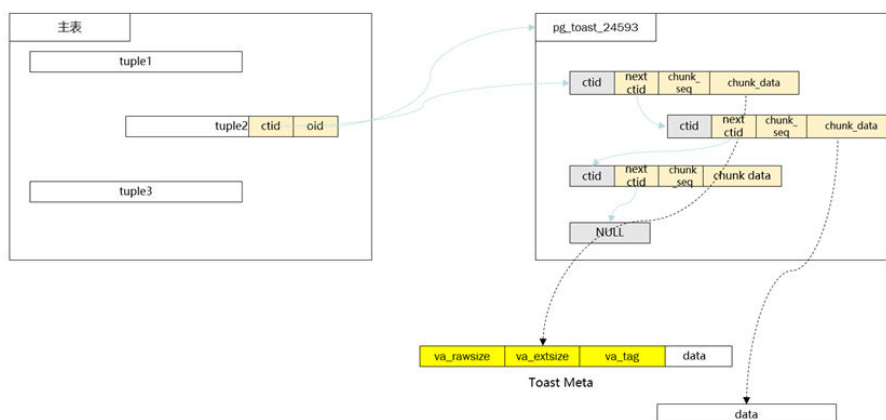
Enhanced Toast是一种用于处理超大字段的技术。首先，减少了Toast Pointer中的冗余信息，存储支持单表超长字段列数超过500列。其次，优化了主表与线外存储表之间的映射关系，无需通过pg\_toast\_index来存储主表数据与线外存储表数据的关系，降低了用户存储空间。最后，Enhanced Toast技术通过让分割数据自链接，消除了Oid分配的依赖，极大地加快了写入效率。

#### 说明

- Astore存储引擎不支持Enhanced Toast。
- 不支持对Enhanced Toast类型的线外存储表单独进行Vacuum Full操作。

### 6.3.2.4.2 Enhanced Toast 存储结构

Enhanced Toast技术使用自链接的方式来处理元组间的依赖关系。线外存储表把超长数据按照2K分割成链表块，主表的Toast Pointer指向线外存储表的对应数据链表头。这样极大简化了主表与线外存储表间的映射关系，有效的提升了数据写入与查询的性能。



### 6.3.2.4.3 Enhanced Toast 使用

新增的GUC参数enable\_enhance\_toast\_table用于控制线外存储结构。

“enable\_enhance\_toast\_table=on”表示使用Enhanced Toast线外存储表，

“enable\_enhance\_toast\_table=off”表示使用Toast线外存储表。

```
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "enable_enhance_toast_table=on"
```

### 6.3.2.4.4 Enhanced Toast 增删改查

**Insert操作：**触发Enhanced Toast的写入条件保持与原有Toast一致，除了数据写入时增加了数据间的链接信息之外，插入基本逻辑保持不变。

**Delete操作：**Enhanced Toast的数据删除流程不再依赖Toast数据索引，仅依靠数据间的链接信息将对应的数据进行遍历删除。

**Update操作：**Enhanced Toast的更新流程与原有Toast保持一致。



### 6.3.2.4.5 Enhanced Toast 相关 DDL 操作

#### Enhanced Toast表的创建

建表时指定Toast表的存储类型为Enhanced Toast或者Toast:

```
gaussdb=# CREATE TABLE test_toast (id int, content text) with(toast.toast_storage_type=toast);
CREATE TABLE
gaussdb=# \d+ test_toast
          Table "public.test_toast"
  Column | Type   | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id     | integer |          | plain   |              |
 content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=# DROP TABLE test_toast;
DROP TABLE
gaussdb=# CREATE TABLE test_toast (id int, content text) with(toast.toast_storage_type=enhanced_toast);
CREATE TABLE
gaussdb=# \d+ test_toast
          Table "public.test_toast"
  Column | Type   | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id     | integer |          | plain   |              |
 content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
gaussdb=# DROP TABLE test_toast;
DROP TABLE
```

建表时不指定线外存储表的类型，则创建线外存储表类型依赖于GUC参数 enable\_enhance\_toast\_table:

```
-- 根据“Enhanced Toast使用”章节打开GUC
gaussdb=# show enable_enhance_toast_table;
enable_enhance_toast_table
-----
on
(1 row)
gaussdb=# CREATE TABLE test_toast (id int, content text);
CREATE TABLE
gaussdb=# \d+ test_toast
          Table "public.test_toast"
  Column | Type   | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id     | integer |          | plain   |              |
 content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast

gaussdb=# DROP TABLE test_toast;
DROP TABLE
gaussdb=# SET enable_enhance_toast_table = off;
SET
gaussdb=# show enable_enhance_toast_table;
enable_enhance_toast_table
-----
off
(1 row)
```

```
gaussdb=# CREATE TABLE test_toast (id int, content text);
CREATE TABLE
gaussdb=# \d+ test_toast
          Table "public.test_toast"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
id     | integer |          | plain   |              |
content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=# DROP TABLE test_toast;
DROP TABLE
```

### 线外存储表结构的升级

当GUC参数“enable\_enhance\_toast\_table=on”时，线外存储表支持通过Vacuum Full操作将Toast升级为Enhanced Toast结构。

```
gaussdb=# CREATE TABLE test_toast (id int, content text);
CREATE TABLE
gaussdb=# \d+ test_toast
          Table "public.test_toast"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
id     | integer |          | plain   |              |
content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=# VACUUM FULL test_toast;
VACUUM
gaussdb=# \d+ test_toast
          Table "public.test_toast"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
id     | integer |          | plain   |              |
content | text   |          | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
gaussdb=# DROP TABLE test_toast;
DROP TABLE
```

### 分区表merge操作

支持将分区表的分区不同的线外存储表类型进行合并操作。

#### 说明

- 对于相同类型的线外存储分区，合并与原有逻辑保持一致，进行物理合并。
- 对于不同类型的线外存储分区，合并后的分区线外存储表为Enhanced Toast表，需要进行逻辑合并，性能劣于物理合并。

```
gaussdb=# CREATE TABLE test_partition_table(a int, b text)PARTITION BY range(a)(partition p1 values less
than (2000),partition p2 values less than (3000));
gaussdb=# SELECT relfilenode FROM pg_partition WHERE relname='p1';
relfilenode
-----
17529
(1 row)
```

```

gaussdb=# \d+ pg_toast.pg_toast_part_17529
TOAST table "pg_toast.pg_toast_part_17529"
  Column | Type | Storage
-----+-----+-----
 chunk_id | oid | plain
 chunk_seq | integer | plain
 chunk_data | bytea | plain
Options: storage_type=ustore, toast_storage_type=toast

gaussdb=# SELECT relfilenode FROM pg_partition WHERE relname='p2';
relfilenode
-----
      17528
(1 row)

gaussdb=# \d+ pg_toast.pg_toast_part_17528
TOAST table "pg_toast.pg_toast_part_17528"
  Column | Type | Storage
-----+-----+-----
 chunk_seq | integer | plain
 next_chunk | tid | plain
 chunk_data | bytea | plain
Options: storage_type=ustore, toast_storage_type=enhanced_toast
gaussdb=# ALTER TABLE test_partition_table MERGE PARTITIONS p1,p2 INTO partition p1_p2;
ALTER TABLE
gaussdb=# SELECT reltoastrelid::regclass FROM pg_partition where relname='p1_p2';
reltoastrelid
-----
pg_toast.pg_toast_part_17559
(1 row)
gaussdb=# \d+ pg_toast.pg_toast_part_17559
TOAST table "pg_toast.pg_toast_part_17559"
  Column | Type | Storage
-----+-----+-----
 chunk_seq | integer | plain
 next_chunk | tid | plain
 chunk_data | bytea | plain
Options: storage_type=ustore, toast_storage_type=enhanced_toast
gaussdb=# DROP TABLE test_partition_table;
DROP TABLE
    
```

### 6.3.2.4.6 Enhanced Toast 运维管理

通过gs\_parse\_page\_bypath解析主表中的ToastPointer信息。

```

gaussdb=# SELECT ctid,next_chunk,chunk_seq FROM pg_toast.pg_toast_part_17559;
 ctid | next_chunk | chunk_seq
-----+-----+-----
(0,1) | (0,0)      |         1
(0,2) | (0,1)      |         0
(0,3) | (0,0)      |         1
(0,4) | (0,3)      |         0
(4 rows)
gaussdb=# SELECT gs_parse_page_bypath((SELECT * FROM
pg_relation_filepath('test_toast')),0,'uheap',false);
gs_parse_page_bypath
-----
${data_dir}/gs_log/dump/1663_13113_17603_0.page
(1 row)
    
```

解析文件1663\_13113\_17603\_0.page中存储了ToastPointer的相关信息，具体如下：

```

Toast_Pointer:
column_index: 1
toast_relation_oid: 17608 --线外存储表OID信息
ctid: (4, 1) --线外存储数据链，头指针
bucket id: -1 --bucket id信息
column_index: 2
    
```

```
toast_relation_oid: 17608  
ctid: (2, 1)  
bucket id: -1
```

Enhanced Toast数据查询，通过直接查询到的Enhanced Toast表数据可以判断其链式结构的完整性。

```
gaussdb=# SELECT ctid,next_chunk,chunk_seq FROM pg_toast.pg_toast_part_17559;  
ctid | next_chunk | chunk_seq  
-----+-----+-----  
(0,1) | (0,0)      | 1  
(0,2) | (0,1)      | 0  
(0,3) | (0,0)      | 1  
(0,4) | (0,3)      | 0  
(4 rows)
```

### 6.3.3 Ustore 事务模型

GaussDB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/DDl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：事务提交中、事务提交、事务回滚、事务已冻结等。

#### 6.3.3.1 事务提交

针对隐式事务和显式事务，其提交策略如下所示：

1. 隐式事务。单条DML/DDl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。

#### 6.3.3.2 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、Ubtrees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、异步和页面级回滚等3种形式。

- **同步回滚**

有三种情况会触发事务的同步回滚：

- 事务块中的ROLLBACK关键字会触发同步回滚。
  - 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。
  - 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。
- **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。
  - **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才会进行父事务的回滚。

## 6.3.4 闪回恢复

闪回恢复功能是数据库恢复技术的一环，可以有选择性地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引擎只支持闪回DROP/TRUNCATE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。

### 6.3.4.1 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

整体方案分为三部分：旧版本保留、快照的维护和旧版本检索。旧版本保留：新增undo\_retention\_time配置参数，用来设置旧版本保留的时间，超过该时间的旧版本将被回收清理，若使用闪回查询则需将该参数设置为大于0的值，请联系管理员修改。

#### 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 [, ...] ) ]  
|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（需将undo\_retention\_time参数设置为大于0的值）：

```
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
```

### 6.3.4.2 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

整体方案分为三部分：旧版本保留、快照的维护和旧版本检索。旧版本保留：新增undo\_retention\_time配置参数，用来设置旧版本保留的时间，超过该时间的旧版本将被回收清理，请联系管理员修改。

## 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
```

### 6.3.4.3 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gaussdb.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。
  - 回收站中系统生成的对象名称是唯一的。因此，如果指定系统生成名称，那么数据库检索指定的对象。使用“select \* from gs\_recyclebin;”语句查看回收站中的内容。
  - 在指定了用户指定的名称且回收站中包含多个该名称的对象的情况下，数据库检索回收站中最近移动的对象，如果想要检索更早版本的表，可以执行如下操作：
    - 指定你想要检索的表的系统生成名称。
    - 执行TIMECAPSULE TABLE ... TO BEFORE DROP语句，直到你要检索的表。
  - 恢复DROP表时，只恢复基表名，其他子对象名均保持回收站对象名。用户可根据需要，执行DDL命令手工调整子对象名。
  - 回收站对象不支持DML、DCL、DDL等写操作，不支持DQL查询操作（后续支持）。
  - 闪回点和当前点之间，执行过修改表结构或影响物理结构的语句，闪回失败。执行过DDL的表进行闪回操作报错：“ERROR: The table definition of %s has been changed.”。涉及namespace、表名改变等操作的DDL执行闪回操作报错：ERROR: recycle object %s desired does not exist;
  - 开启enable\_recyclebin参数，启用回收站时，如果表上有truncate trigger、truncate表时，无法触发trigger。
- **RENAME TO**  
为从回收站中检索的表指定一个新名称。
- **TO BEFORE TRUNCATE**

闪回到TRUNCATE之前。

## 语法示例

```
-- PURGE TABLE table_name; --
--查看回收站
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychange | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
(0 rows)

gaussdb=# DROP TABLE IF EXISTS flashtest;
NOTICE: table "flashtest" does not exist, skipping
DROP TABLE
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychange | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+
(0 rows)
--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
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1, 'A');
INSERT 0 1
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 | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychange | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+
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 | 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 |
(2 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;
```





```
^
--闪回drop表
gaussdb=# TIMECAPSULE TABLE flashtest to before drop;
TimeCapsule Table
--查看表, 表被恢复到drop之前
gaussdb=# SELECT * FROM flashtest;
id | name
----+-----
   1 | A
(1 row)

--查看回收站, 回收站中的表被删除
gaussdb=# SELECT * FROM gs_recyclebin;
rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
| rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

--DROP表
gaussdb=# DROP TABLE IF EXISTS flashtest;
DROP TABLE
gaussdb=# SELECT * FROM flashtest;
ERROR: relation "flashtest" does not exist
LINE 1: SELECT * FROM flashtest;
^
--查看回收站, 表被放入回收站
gaussdb=# SELECT * FROM gs_recyclebin;
rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
| rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
18664 | 12737 | 18652 | BIN$31C14EB48DC$9B4E$0==$0 | flashtest | d | 0 |
| 79354845 | 2023-09-13 20:49:17.762977+08 | 79354753 |
79354753 | 2200 | 10 | 0 | 18652 | t | t | 226824 | 226824 |
18664 | 12737 | 18657 | BIN$31C14EB48E1$12E680A8==$0 | BIN$31C14EB48E1$12E45E00==$0 |
d | 3 | 79354845 | 2023-09-13 20:49:17.763271+08 | 79354753 |
79354753 | 99 | 10 | 0 | 18657 | f | f | 0 | 0 |
18664 | 12737 | 18655 | BIN$31C14EB48DF$12E68698==$0 | BIN$31C14EB48DF$12E46400==$0 |
d | 2 | 79354845 | 2023-09-13 20:49:17.763343+08 | 0 |
0 | 99 | 10 | 0 | 18655 | f | f | 226824 | 226824 |
(3 rows)

--闪回drop表, 表名用回收站中的rcyname
gaussdb=# TIMECAPSULE TABLE "BIN$31C14EB48DC$9B4E$0==$0" to before drop;
TimeCapsule Table
---查看回收站, 回收站中的表被删除
gaussdb=# SELECT * FROM gs_recyclebin;
rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
| rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

gaussdb=# SELECT * FROM flashtest;
id | name
----+-----
   1 | A
(1 row)

--DROP表
```

```
gaussdb=# DROP TABLE IF EXISTS flashtest;
DROP TABLE
----查看回收站，表被放入回收站
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname      | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcy
changeccsn | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcyanrestore | rcyanpurge |
 rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----
|
|
|
18667 | 12737 | 18652 | BIN$31C14EB48DC$9B8D$0==$0 | flashtest | d | 0
|
79354943 | 2023-09-13 20:52:14.525946+08 | 79354753 |
79354753 | 2200 | 10 | 0 | 18652 | t | t | 226824 | 226824 |
18667 | 12737 | 18657 | BIN$31C14EB48E1$1320B4F0==$0 | BIN$31C14EB48E1$12E680A8==$0 |
d | 3 | 79354943 | 2023-09-13 20:52:14.526319+08 | 79354753 |
79354753 | 99 | 10 | 0 | 18657 | f | f | 0 |
18667 | 12737 | 18655 | BIN$31C14EB48DF$1320BAE0==$0 | BIN$31C14EB48DF$12E68698==$0 |
d | 2 | 79354943 | 2023-09-13 20:52:14.526423+08 | 0 |
0 | 99 | 10 | 0 | 18655 | f | f | 226824 | 226824 |
(3 rows)

--查看表，表不存在
gaussdb=# SELECT * FROM flashtest;
ERROR: relation "flashtest" does not exist
LINE 1: SELECT * FROM flashtest;
          ^
--闪回drop表，并重命名表
gaussdb=# TIMECAPSULE TABLE flashtest to before drop rename to flashtest_rename;
TimeCapsule Table
--查看原表，表不存在
gaussdb=# SELECT * FROM flashtest;
ERROR: relation "flashtest" does not exist
LINE 1: SELECT * FROM flashtest;
          ^
--查看重命名后的表，表存在
gaussdb=# SELECT * FROM flashtest_rename;
 id | name
-----+-----
  1 | A
(1 row)

--查看回收站，回收站中的表被删除
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname      | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangeccsn | rcynamespace | rcyowner | rcytablespace |
 rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----
|
|
|
(0 rows)
--drop表
gaussdb=# DROP TABLE IF EXISTS flashtest_rename;
DROP TABLE
--清空回收站
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
--查看回收站，回收站被清空
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname      | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangeccsn | rcynamespace | rcyowner | rcytablespace |
 rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----
|
|
|
(0 rows)
```

```
-- TIMECAPSULE TABLE { table_name } TO BEFORE TRUNCATE --
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
--插入数据
gaussdb=# INSERT INTO flashtest VALUES(1, 'A');
INSERT 0 1
gaussdb=# SELECT * FROM flashtest;
 id | name
----+-----
  1 | A
(1 row)

--truncate表
gaussdb=# TRUNCATE TABLE flashtest;
TRUNCATE TABLE
--查看回收站, 表的数据被放入回收站
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname           | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangeecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rycanrestore | rycanpurge | rcyfrozenxid |
 rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
 18703 | 12737 | 18697 | BIN$31C14EB4909$9E4C$0==$0 | flashtest      | t          | 0 |
79356608 | 2023-09-13 21:24:42.819863+08 | 79356606 | 7935660
6 | 2200 | 10 | 0 | 18697 | t | t | 227927 | 227927 |
 18703 | 12737 | 18700 | BIN$31C14EB490C$132FE3F0==$0 | pg_toast_18697 | t          | 2 |
| 79356608 | 2023-09-13 21:24:42.820358+08 | 0 |
0 | 99 | 10 | 0 | 18700 | f | f | 227927 | 227927 |
(2 rows)

--查看表, 表中的数据为空
gaussdb=# SELECT * FROM flashtest;
 id | name
----+-----
(0 rows)

--闪回truncate表
gaussdb=# TIMECAPSULE TABLE flashtest to before truncate;
TimeCapsule Table
--查看表, 表中的数据被恢复
gaussdb=# SELECT * FROM flashtest;
 id | name
----+-----
  1 | A
(1 row)

--查看回收站
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname           | rcyoriginname | rcyoperation | rcytype |
 rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangeecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rycanrestore | rycanpurge | rcyfrozenxid |
 rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
 18703 | 12737 | 18700 | BIN$31C14EB490C$13300228==$0 | pg_toast_18697 | t          | 2 |
| 79356610 | 2023-09-13 21:24:42.872732+08 | 0 |
0 | 99 | 10 | 0 | 18706 | f | f | 0 | 227928 |
 18703 | 12737 | 18697 | BIN$31C14EB4909$9E4D$0==$0 | flashtest      | t          | 0 |
```





视图类型	类型	功能描述	使用场景	函数名称	
	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> <li>undo回收异常。</li> <li>回滚异常。</li> <li>日常巡检。</li> <li>校验报错。</li> <li>可见性判断异常。</li> <li>修改参数。</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。		gs_undo_dump_xid	
		用于解析指定UndoZone中所有 Transaction Slot信息。		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。		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。		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	
		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 ( 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。可以看做一次执行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 ( 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

视图类型	类型	功能描述	使用场景	函数名称
	预写日志 (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> <li>WAL写/刷盘hang住。</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。		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
	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	

视图类型	类型	功能描述	使用场景	函数名称
		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_m eta
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_tryr epair_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_by pa th
		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	

## 6.3.6 常见问题及定位手段

### 6.3.6.1 snapshot too old

查询SQL执行时间过长或者其他一些原因, 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。

#### 6.3.6.1.1 长事务阻塞 Undo 空间回收

##### 问题现象

- gs\_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 row)
1 | 280 | 248 | 17028 | 17028 | 17025 | 17028
```

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，则表明有长事务卡主了回收。

```
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;
```

```
tpcc=# select * from pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by xmin::text::bigint asc limit 5;
handle | gid | state | mode | xmin | vacuum | timeline | prepare_xid | pid | next_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277812763024 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277812763024 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277812763024 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277812763024 | 0
(5 rows)

Time: 1000.559 ms
tpcc=# select * from gs_txid_oldestxmin();
gs_txid_oldestxmin
-----
5576784113
(1 row)

Time: 2.935 ms
tpcc=# select * from txid_current();
txid_current
-----
55789826407
(1 row)

Time: 7.058 ms
tpcc=# select * from pg_stat_activity where pid = 281303148456336;
datid | datname | pid | sessionid | useysid | username | application_name | client_addr | client_hostname | client_port | backend_start | xact_start | query_start | state | resource_pool | query_id
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
| connection_info | | | | | | | | | | | | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
17205 | tpcc | 281303148456336 | 707 | 17281 | test | gsql | | | | | 60154 | 2023-03-13 18:00:40.17499140 | 2023-03-13 18:00:40.254716408 | 2023-03-13 18:00:40.254727408 | f | | active | default_pool | 14636698788954165
| select /tmp/tpccscan(part3) /v sum(c) count(*) from part union select sum(c) count(*) from part; | | driver_name: libpq, driver_version: GaussDB Kernel, SaaS 1.0 build 628fcd5d, compiled at 2023-03-13 08:05:43 commit 5208 (last nr 10563 release) | 3734947889 |
(1 row)

Time: 15592.269 ms
tpcc=#
```

## 处理方法

通过pg\_terminate\_session(pid, sessionid)终止长事务所在的会话（提醒：长事务无固定快速恢复手段，强制结束SQL语句为其中一种常用操作，属于高危操作，执行需谨慎，执行前需与业务及华为技术确认，避免造成业务失败或报错）。

### 6.3.6.1.2 大量回滚事务拖慢 Undo 空间回收

## 问题现象

使用gs\_async\_rollback\_xact\_status视图查看有大量的待回滚事务，且待回滚的事务数量维持不变或者持续增高。

```
SELECT * FROM gs_async_rollback_xact_status();
```

## 处理方法

调大异步回滚线程数量，调整方式有以下两种：

方式1：在gaussdb.conf中配置max\_undo\_workers，然后重启节点。

方式2：gs\_guc reload -Z NODE-TYPE [-N NODE-NAME] [-I INSTANCE-NAME] [-D DATADIR] -c max\_undo\_workers=100 重启实例。

### 6.3.6.2 storage test error

业务执行过程中，数据页、索引或者Undo页面发生变更后，该页面放锁之前会主动进行逻辑损坏检测，发现页面损坏问题后会输出包含“storage test error”关键字的日志信息到数据库运行日志（gs\_log文件），执行事务回滚，页面会恢复到修改前的状态。

#### 问题现象

gs\_log中打印“storage test error”关键字。

#### 处理方法

请联系华为技术支持解决。

### 6.3.6.3 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

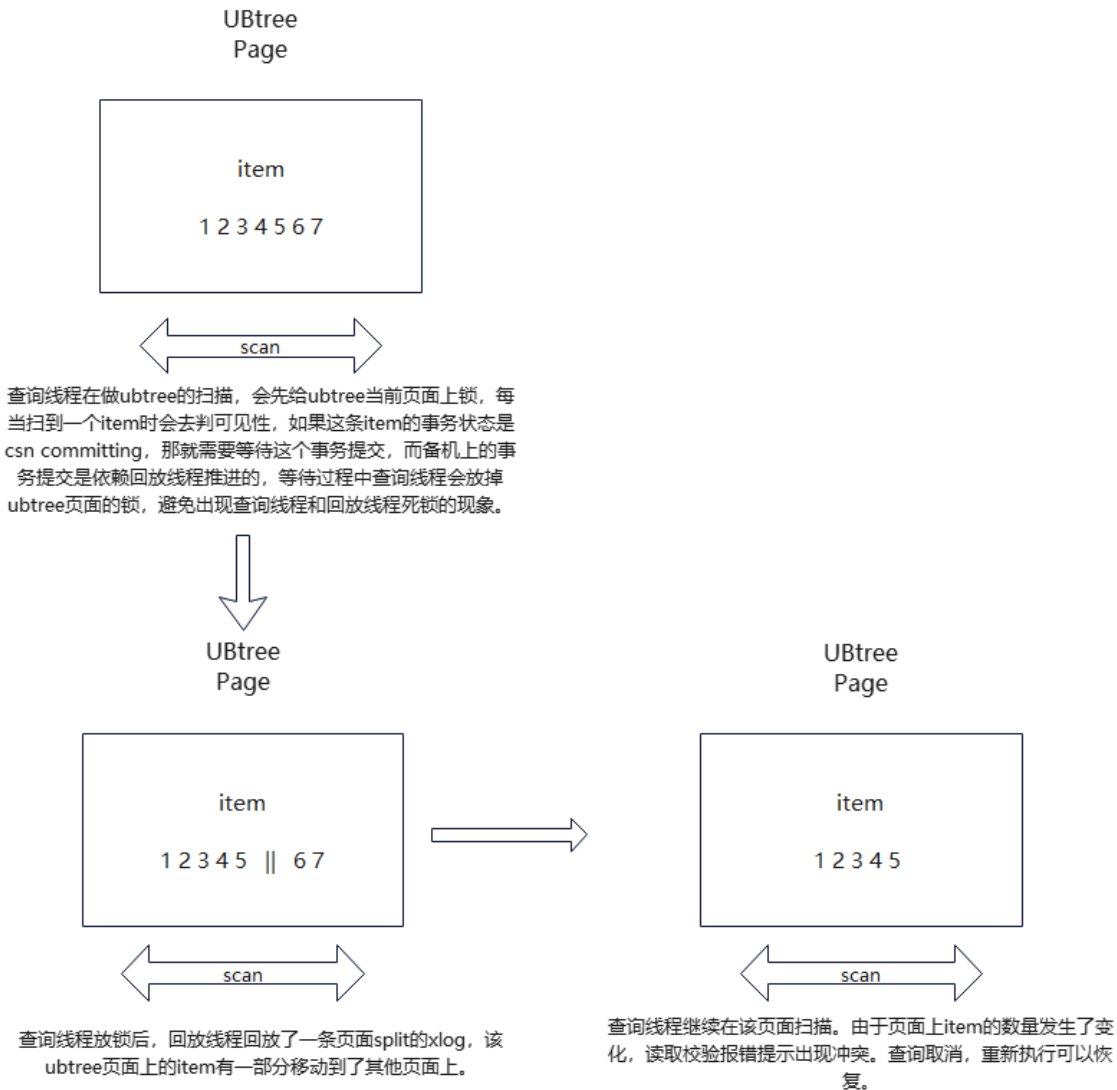
在开启并行回放或串行回放的情况下（查询GUC参数recovery\_parse\_workers和recovery\_max\_workers均是1为串行回放；recovery\_parse\_workers是1，recovery\_max\_workers大于1为并行回放），备机的查询线程在做索引扫描时，会先对索引页面加读锁，每当扫到一个元组时会去判可见性。如果该元组对应的事务处于committing状态，需要等待该事务提交后再判断。而备机上的事务提交是依赖日志回放线程推进的，这个过程中会对索引页面进行修改，因此需要加锁。查询线程在等待过程中会释放索引页面的锁，否则会出现查询线程等待回放线程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

#### 📖 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 6-6 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。

### 6.3.6.4 长查询执行期间大量并发更新偶现写入性能下降

#### 问题现象

执行全表扫描类型的长查询，扫描期间页面发生大量并发更新，部分DML写入性能较没有长查询时出现性能下降。

#### 问题分析

对于全表扫描场景下的长查询（例如持续两小时以上），在扫描到某个页面前，该页面发生大量集中并发更新（例如十万次以上更新），后续扫描到该页面时，需要访问

大量历史版本获取可见元组（MVCC机制），由于单页扫描期间持有页面读锁，若此时刚好需要写入该页面，写入会被阻塞，直到页面元组读取完成。

## 定位手段

1. 结合慢SQL告警、statement\_history视图等确认是否存在长查询和超时取消的DML语句。
2. 获取statement\_history中查询到的被取消DML的details信息，使用statement\_detail\_decode系统函数解析details字段，获取等待事件，如等待事件开销占比最高为BufferContentLock则大概率为本问题。

## 处理方法

事前预防：避免在高并发表上执行全表扫描类型长查询，建议长查询迁移到备机执行。

事中处理：结合慢SQL告警等确认是否触发此场景，可通过中断长查询避免对业务的持续影响。

# 6.4 数据生命周期管理-OLTP 表压缩

## 6.4.1 特性简介

OLTP表压缩是GaussDB高级压缩中的一个特性。基于全新的压缩算法、细粒度的自动冷热判定和支持块内压缩等技术创新，可以在提供合理压缩率的同时大幅度降低对业务的影响、增加后台调度、增加查询Job执行状态以及节约空间，能够在支持关键在线业务的容量控制中发挥重要价值。

## 6.4.2 特性约束

- 不支持系统表、内存表、全局临时表、本地临时表和序列表，不支持Ustore段页式表，不支持unlogged表，不支持压缩toast数据。
- 仅在ORA兼容模式、PG模式以及MYSQL兼容模式下有效。
- Ustore不支持编解码，压缩率低于Astore。
- 普通表开启压缩时，扩容空间预留需按照解压后的大小评估。
- HashBucket表不支持DBE\_HEAT\_MAP.ROW\_HEAT\_MAP和DBE\_COMPRESSION.GET\_COMPRESSION\_TYPE。
- 扩容期间不支持压缩调度。
- 扩容前请确认当前是否有正在执行的压缩任务，如果有的话，要么等待压缩任务结束，要么执行DBE\_ILM.STOP\_ILM或DBE\_ILM\_ADMIN.DISABLE\_ILM停掉，扩容完成后再执行DBE\_ILM\_ADMIN.ENABLE\_ILM开启。
- 由于当前版本新增了部分解压特性，会导致部分极小的表在判断压缩收益时不通过，从而不进行压缩。但此前没有部分解压的版本可以压缩。例如：单表单字段int1类型的数据从当前版本开始无法进行压缩。



### 6.4.3 特性规格

- TPCC只开启策略、不开调度对原有业务无影响。
- TPCC不开启压缩策略对原有业务无影响。
- TPCC.bmsql\_order\_line设置ILM策略（只识别完成派送的订单为冷行）不调度，TPmC劣化不高于2%（56核CPU370GB内存+3TB SSD硬盘，350GB SharedBuffer）。
- TPCC.bmsql\_order\_line设置ILM策略（只识别完成派送的订单为冷行）后台默认参数调度时，TPmC劣化不高于5%（56核CPU370GB内存+3TB SSD硬盘，350GB SharedBuffer）。
- 单线程ILM Job带宽约100MB/秒（56核CPU370GB内存+3TB SSD硬盘，350GB SharedBuffer）。  
度量方式：根据执行压缩的开始时间和结束时间以及压缩的页面个数计算带宽。
- get查询访问压缩数据比非压缩数据性能劣化，驱动侧不高于10%，plsql侧不高于15%（32MB SharedBuffer，6万页面数据）。
- multi-get查询访问压缩数据比非压缩数据性能劣化，驱动侧不高于30%，plsql侧不高于40%（32MB SharedBuffer，6万页面数据）。
- table-scan查询访问压缩数据比非压缩数据性能劣化，驱动侧不高于30%，plsql侧不高于40%（32MB SharedBuffer，6万页面数据）。
- TPCH.lineitem表压缩比（全冷行）不小于2:1。
- 对于TPC-C的Orderline表，以及TPC-H的Lineitem、Orders、Customer、Part表的测试表明，数值型字段较多时，压缩率高于LZ4和ZLIB；而文本型字段较多时，压缩率介于LZ类和LZ+Huffman组合类的压缩算法之间。

### 6.4.4 使用说明

#### 步骤1 执行如下命令开启压缩功能：

```
gaussdb=# ALTER DATABASE SET ilm = on;
```

检查当前数据库的public schema中是否存在gsilmpolicy\_seq和gsilmtask\_seq。

```
gaussdb=# \d
```

```
                List of relations
Schema | Name          | Type   | Owner | Storage
-----+-----+-----+-----+-----
public | gsilmpolicy_seq | sequence | omm |
public | gsilmtask_seq  | sequence | omm |
```

或者：

```
gaussdb=# SELECT a.oid, a.relname FROM pg_class a inner join pg_namespace b on a.relnamespace = b.oid
WHERE (a.relname = 'gsilmpolicy_seq' OR a.relname = 'gsilmtask_seq') AND b.nspname = 'public';
```

```
oid | relname
-----+-----
17002 | gsilmpolicy_seq
17004 | gsilmtask_seq
(2 rows)
```

生成异常会报warning：

```
WARNING: ILM sequences are already existed while initializing
```

#### 步骤2 为表添加压缩策略。

- 新建带策略的表：

```
gaussdb=# CREATE TABLE ilm_table_1 (col1 int, col2 text)
ilm add policy row store compress advanced row
after 3 days of no modification on (col1 < 1000);
```

- 为存量表添加策略:

```
gaussdb=# CREATE TABLE ilm_table_2 (col1 int, col2 text);
gaussdb=# ALTER TABLE ilm_table_2 ilm add policy row store
compress advanced row after 3 days of no modification;
```

- 检查策略视图中是否新增数据:

```
gaussdb=# SELECT * FROM gs_my_ilm_policies;

policy_name | policy_type | tablespace | enabled | deleted
-----+-----+-----+-----+-----
p1          | DATA MOVEMENT |          | YES    | NO
p2          | DATA MOVEMENT |          | YES    | NO
(2 rows)
```

- 检查策略详细信息视图中是否新增了符合刚刚设置的策略:

```
gaussdb=# SELECT * FROM gs_my_ilm_data_movement_policies;

policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type | condition_days | custom_function | policy_subtype | action_clause | tier_to
-----+-----+-----+-----+-----+-----+-----
p1          | COMPRESSION | ROW  | ADVANCED          |                |            |
LAST MODIFICATION
TIME | 3 |                |                |                |
p2          | COMPRESSION | ROW  | ADVANCED          |                |            |
LAST MODIFICATION
TIME | 3 |                |                |                |
(2 rows)
```

- 检查策略与目标表是否对应:

```
gaussdb=# SELECT * FROM gs_my_ilm_objects;

policy_name | object_owner | object_name | subobject_name | object_type | inherited_from |
tbs_inherited_from | enabled | deleted
-----+-----+-----+-----+-----+-----+-----
p1          | public      | ilm_table_1 |                | TABLE     | POLICY NOT INHERITED |
YES | NO
p2          | public      | ilm_table_2 |                | TABLE     | POLICY NOT INHERITED |
YES | NO
(2 rows)
```

### 步骤3 执行压缩评估。

- 手动执行压缩评估。



**注意**

为方便测试，本功能环境参数中提供POLICY\_TIME属性，决定时间条件以天为单位还是以秒为单位。通过下面语句调整：

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
```

插入随机数据用于测试：

```
gaussdb=# INSERT INTO ilm_table_1 select *, 'test_data' FROM generate_series(1, 10000);
```

```
gaussdb=# DECLARE
v_taskid number;
gaussdb=# BEGIN
DBE_ILM.EXECUTE_ILM(OWNER      => 'public',
OBJECT_NAME   => 'ilm_table_1',
TASK_ID       => v_taskid,
SUBOBJECT_NAME => NULL,
POLICY_NAME   => 'ALL POLICIES',
EXECUTION_MODE => 2);
RAISE INFO 'Task ID is:%', v_taskid;
```

```
gaussdb=# END;
/
```

如入参有误，会报对应的错误信息。无误则无输出（上述代码段添加了RAISE INFO语句打印当前task的id）。

```
INFO: Task ID is:1
```

检查task信息：

```
gaussdb=# SELECT * FROM gs_my_ilmtasks;
```

task_id	task_owner	state	creation_time	start_time	completion_time
1	omm	COMPLETED	2023-08-29 17:36:38.779555+08	2023-08-29 17:36:38.779555+08	2023-08-29 17:36:38.879485+08

(1 row)

检查评估结果：

```
gaussdb=# SELECT * FROM gs_my_ilmevaluationdetails;
```

task_id	policy_name	object_owner	object_name	subobject_name	object_type	selected_for_execution	job_name	comments
1	p1	public	ilm_table_1		TABLE	SELECTED FOR EXECUTION	ilmjob	

(1 row)

检查压缩job信息：

```
gaussdb=# SELECT * FROM gs_my_ilmresults;
```

task_id	job_name	job_state	start_time	completion_time	comments	statistics
1	ilmjob\$_postgres1	COMPLETED SUCCESSFULLY	2023-08-29 17:36:38.779555+08	2023-08-29 17:36:38.879485+08		SpaceSaving=0,BoundTime=0,LastBlkNum=0

(1 row)

- 触发后台自动调度评估。

使用初始用户登录template1数据库，创建维护窗口：

```
gaussdb=#
DECLARE
  V_HOUR INT := 22;
  V_MINUTE INT := 0;
  V_SECOND INT := 0;
  C_ADO_WINDOW_SCHEDULE_NAME TEXT := 'ado_window_schedule';
  C_ADO_WINDOW_PROGRAM_NAME TEXT := 'ado_window_program';
  C_MAINTENANCE_WINDOW_JOB_NAME TEXT := 'maintenance_window_job';
  V_MAINTENANCE_WINDOW_REPEAT TEXT;
  V_MAINTENANCE_WINDOW_START TIMESTAMPTZ;
  V_BE_SCHEDULE_ENABLE BOOL;
  V_MAINTENANCE_WINDOW_EXIST INT;
BEGIN
  SELECT COUNT(*) INTO V_MAINTENANCE_WINDOW_EXIST FROM PG_CATALOG.PG_JOB WHERE
  JOB_NAME = 'maintenance_window_job' AND DBNAME = 'template1';
  IF CURRENT_DATABASE() != 'template1' THEN
    RAISE EXCEPTION 'Create maintenance_window FAILED, current database is not tempalte1';
  END IF;
  IF V_MAINTENANCE_WINDOW_EXIST = 0 AND CURRENT_DATABASE() = 'template1' THEN
    SELECT
      CASE
        WHEN NOW() < CURRENT_DATE + INTERVAL '22 HOUR' THEN CURRENT_DATE +
        INTERVAL '22 HOUR'
        ELSE CURRENT_DATE + INTERVAL '1 DAY 22 HOUR'
      END INTO V_MAINTENANCE_WINDOW_START;
    --1. prepare for maintenance window schedule
    SELECT 'freq=daily;interval=1;byhour='||V_HOUR||';byminute='||V_MINUTE||';bysecond='||
    V_SECOND INTO V_MAINTENANCE_WINDOW_REPEAT;
```

```

BEGIN
  SELECT
    CASE
      WHEN VALUE = 1 THEN TRUE -- DBE_ILM_ADMIN.ILM_ENABLED
      ELSE FALSE
    END INTO V_BE_SCHEDULE_ENABLE
  FROM PG_CATALOG.GS_ILM_PARAM WHERE IDX = 7; -- DBE_ILM_ADMIN.ENABLED
EXCEPTION
  WHEN OTHERS THEN
    V_BE_SCHEDULE_ENABLE := FALSE;
END;
--2. Create ado window schedule
DBE_SCHEDULER.CREATE_SCHEDULE(
  SCHEDULE_NAME => C_ADO_WINDOW_SCHEDULE_NAME,
  START_DATE => '9999-01-01 00:00:01',
  REPEAT_INTERVAL => NULL,
  END_DATE => NULL,
  COMMENTS => 'ado window schedule');
--3. Create ado window program
DBE_SCHEDULER.CREATE_PROGRAM(
  PROGRAM_NAME => C_ADO_WINDOW_PROGRAM_NAME,
  PROGRAM_TYPE => 'plsql_block',
  PROGRAM_ACTION => 'call prvt_ilm.be_execute_ilm(0);',
  NUMBER_OF_ARGUMENTS => 0,
  ENABLED => TRUE,
  COMMENTS => NULL);
--4. Create maintenance window master job
DBE_SCHEDULER.CREATE_JOB(
  JOB_NAME => C_MAINTENANCE_WINDOW_JOB_NAME,
  START_DATE => V_MAINTENCE_WINDOW_START,
  REPEAT_INTERVAL => V_MAINTENCE_WINDOW_REPEAT,
  END_DATE => NULL,
  JOB_TYPE => 'STORED_PROCEDURE'::TEXT,
  JOB_ACTION => 'prvt_ilm.be_active_ado_window'::TEXT,
  NUMBER_OF_ARGUMENTS => 0,
  ENABLED => V_BE_SCHEDULE_ENABLE,
  AUTO_DROP => FALSE,
  COMMENTS => 'maintenance window job',
  destination_name=> 'CCN');
ELSE
  RAISE EXCEPTION 'CREATE ILM MAINTENANCE WINDOW FAILED';
END IF;
END;

```

自动调度提供若干参数用于调整：

gaussdb=# SELECT \* FROM gs\_adm\_ilmparameters;

name	value
EXECUTION_INTERVAL	15
RETENTION_TIME	30
ENABLED	1
POLICY_TIME	0
ABS_JOBLIMIT	10
JOB_SIZELIMIT	1024
WIND_DURATION	240
BLOCK_LIMITS	40
ENABLE_META_COMPRESSION	0
SAMPLE_MIN	10
SAMPLE_MAX	10
CONST_PRIO	40
CONST_THRESHOLD	90
EQVALUE_PRIO	60
EQVALUE_THRESHOLD	80
ENABLE_DELTA_ENCODE_SWITCH	1
LZ4_COMPRESSION_LEVEL	0
ENABLE_LZ4_PARTIAL_DECOMPRESSION	1

(18 rows)

- EXECUTION\_INTERVAL：自动调度任务执行间隔，默认每15分钟执行一次。

- RETENTION\_TIME: 历史压缩任务记录清理间隔，默认每30天清理一次。
- ENABLED: 当前自动调度启用情况，默认为开启。
- POLICY\_TIME: 策略评估的时间单位，测试使用。默认以天为单位。
- ABS\_JOBLIMIT: 单次评估生成压缩任务数量上限，默认为10个。
- JOB\_SIZELIMIT: 单个压缩任务的IO上限，默认为1GB。
- WIND\_DURATION: 单次维护窗口的持续时间。
- BLOCK\_LIMITS: 控制实例级的行存压缩速率上限，默认是40，取值范围是0到10000（0表示不限制），单位是block/ms，表示每毫秒最多压缩多少个block。速率上限计算方法： $BLOCK\_LIMITS * 1000 * BLOCKSIZE$ ，以默认值40为例，其速率上限为： $40 * 1000 * 8KB = 320000KB/s$ 。
- ENABLE\_META\_COMPRESSION: 是否开启header压缩，默认为0，取值范围为0（关闭）和1（开启）。

### 说明

设置此参数为1时，对于单行数据较短的表，压缩率会有一定提升，但是访问压缩行的性能会有较大幅度的下降。若数据库多是单行数据较长的表，不建议开启此参数。

- SAMPLE\_MIN: 常量编码和等值编码采样步长最小值，默认为10，取值范围[1, 100]，支持小数输入，小数会自动向下取整。
- SAMPLE\_MAX: 常量编码和等值编码采样步长最大值，默认为10，取值范围[1, 100]，支持小数输入，小数会自动向下取整。
- CONST\_PRIO: 常量编码优先级，默认为40，取值范围[0, 100]，100表示关闭常量编码，支持小数输入，小数会自动向下取整。
- CONST\_THRESHOLD: 常量编码阈值，默认为90，取值范围[1, 100]，表示一列常量值的占比超过该阈值时进行常量编码，支持小数输入，小数会自动向下取整。
- EQVALUE\_PRIO: 等值编码优先级，默认为60，取值范围[0, 100]，100表示关闭等值编码，支持小数输入，小数会自动向下取整。
- EQVALUE\_THRESHOLD: 等值编码阈值，默认为80，取值范围[1, 100]，表示两列数据的等值比例超过该阈值时进行等值编码，支持小数输入，小数会自动向下取整。
- ENABLE\_DELTA\_ENCODE\_SWITCH: 差值编码开关，默认为1，支持小数输入，0表示关闭，1表示开启，小数会自动向下取整。
- LZ4\_COMPRESSION\_LEVEL: lz4压缩等级，默认为0，取值范围[0, 16]，支持小数输入，小数会自动向下取整。
- ENABLE\_LZ4\_PARTIAL\_DECOMPRESSION: 部分解压开关，默认为1，支持小数输入，0表示关闭，1表示开启，小数会自动向下取整。

以上参数均可通过DBE\_ILM\_ADMIN.CUSTOMIZE\_ILM()接口调整。

维护窗口默认每天晚上22:00（北京时间）开启，可通过DBE\_SCHEDULER提供的接口SET\_ATTRIBUTE进行设置：

```
\c template1
CALL DBE_ILM_ADMIN.DISABLE_ILM();
CALL DBE_ILM_ADMIN.ENABLE_ILM();
DECLARE
  newtime timestampz := CLOCK_TIMESTAMP() + to_interval('2 seconds');
BEGIN
  DBE_SCHEDULER.set_attribute(
    name      => 'maintenance_window_job',
    attribute => 'start_date',
    value     => TO_CHAR(newtime, 'YYYY-MM-DD HH24:MI:SS')
  );
```

```
END;  
/
```

----结束

## 6.4.5 维护窗口参数配置

- RETENTION\_TIME: 评估与压缩记录的保留时长，单位天，默认值30。用户可根据自己存储容量自行调节。
- EXECUTION\_INTERVAL: 评估任务的执行频率，单位分钟，默认值15。用户可根据自己维护窗口期间业务与资源情况调节。该参数与ABS\_JOBLIMIT相互影响。单日单线程最大可产生的I/O为WIND\_DURATION/EXECUTION\_INTERVAL\*JOB\_SIZELIMIT。
- JOB\_SIZELIMIT: 控制单个压缩Job可以处理的最大字节数，单位兆，默认值1024。压缩带宽约为100MB/秒，每个压缩Job限制I/O为1GB时，最多10秒完成。用户可根据自己业务闲时情况以及需要压缩的数据量自行调节。
- ABS\_JOBLIMIT: 控制一次评估最多生成多少个压缩Job。用户可根据自己设置策略的分区及表数量自己调节。建议最大不超过10，可以使用“select count(\*) from gs\_adm\_illobjects where enabled = true”命令查询。
- POLICY\_TIME: 控制判定冷行的条件单位是天还是秒，秒仅用来做测试用。取值为：ILM\_POLICY\_IN\_SECONDS或ILM\_POLICY\_IN\_DAYS（默认值）。
- WIND\_DURATION: 维护窗口持续时长，单位分钟，默认240分钟（4小时）。维护窗口默认从北京时间22点开始持续240分钟，用户可根据自己业务闲时情况自行调节。
- BLOCK\_LIMITS: 控制实例级的行存压缩速率上限，默认是40，取值范围是0到10000(0表示不限制)，单位是block/ms，表示每毫秒最多压缩多少个block。速率上限计算方法： $BLOCK\_LIMITS * 1000 * BLOCKSIZE$ ，以默认值40为例，其速率上限为： $40 * 1000 * 8KB = 320000KB/s$ 。
- ENABLE\_META\_COMPRESSION: 是否开启header压缩，默认为0，取值范围为0（关闭）和1（开启）。用户可根据自己的实际情况来进行开启或关闭。
- SAMPLE\_MIN: 常量编码和等值编码采样步长最小值，默认为10，取值范围[1, 100]，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- SAMPLE\_MAX: 常量编码和等值编码采样步长最大值，默认为10，取值范围[1, 100]，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- CONST\_PRIO: 常量编码优先级，默认为40，取值范围[0, 100]，100表示关闭常量编码，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- CONST\_THRESHOLD: 常量编码阈值，默认为90，取值范围[1, 100]，表示一列常量值的占比超过该阈值时进行常量编码，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- EQVALUE\_PRIO: 等值编码优先级，默认为60，取值范围[0, 100]，100表示关闭等值编码，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- EQVALUE\_THRESHOLD: 等值编码阈值，默认为80，取值范围[1, 100]，表示两列数据的等值比例超过该阈值时进行等值编码，支持小数输入，小数会自动向下取整。用户可根据自己的实际情况来设置具体值。

- **ENABLE\_DELTA\_ENCODE\_SWITCH**: 差值编码开关, 默认为1, 支持小数输入, 0表示关闭, 1表示开启, 小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- **LZ4\_COMPRESSION\_LEVEL**: lz4压缩等级, 默认为0, 取值范围[0, 16], 支持小数输入, 小数会自动向下取整。用户可根据自己的实际情况来设置具体值。
- **ENABLE\_LZ4\_PARTIAL\_DECOMPRESSION**: 部分解压开关, 默认为1, 支持小数输入, 0表示关闭, 1表示开启, 小数会自动向下取整。用户可根据自己的实际情况来进行开启或关闭。

示例分析:

```
EXECUTION_INTERVAL: 15  
JOB_SIZELIMIT: 10240  
WIND_DURATION: 240  
BLOCK_LIMITS: 0
```

此配置下单表分区在一个维护窗口期间可完成 $240/15 * 10240\text{MB} = 160\text{GB}$ 数据的评估压缩。压缩带宽为100MB/秒, 实际压缩仅耗时 $160\text{GB} / (100\text{MB}/\text{秒}) = 27$ 分钟。其他时间对业务无影响。用户可根据自己业务闲时可支配给压缩的时长来调节参数。

## 6.4.6 运维 TIPS

### 1. 手动触发一次压缩 (示例中一次压缩102400MB)。

#### a. 给表加上冷热分离策略:

```
gaussdb=# DROP TABLE IF EXISTS ILM_TABLE;  
gaussdb=# CREATE TABLE ILM_TABLE(a int);  
gaussdb=# ALTER TABLE ILM_TABLE ILM ADD POLICY ROW STORE COMPRESS ADVANCED  
ROW AFTER 3 MONTHS OF NO MODIFICATION;
```

#### b. 手动触发压缩:

```
DECLARE  
v_taskid number;  
BEGIN  
DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);  
DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 102400);  
DBE_ILM.EXECUTE_ILM(OWNER => '$schema_name',  
OBJECT_NAME => 'ilm_table',  
TASK_ID => v_taskid,  
SUBOBJECT_NAME => NULL,  
POLICY_NAME => 'ALL POLICIES',  
EXECUTION_MODE => 2);  
RAISE INFO 'Task ID is:', v_taskid;  
END;  
/
```

#### c. 查看压缩JOB是否完成, 可以看到具体的执行信息:

```
gaussdb=# SELECT * FROM gs_adm_ilmresults ORDER BY task_id desc;
```

task_id	job_name	start_time	completion_time
17267	ilmjob\$_2	2023-03-29 08:11:25	2023-03-29 08:11:25

SpaceSaving=453048, BoundTime=1680145883, LastBlkNum=128

### 2. 手动停止压缩。

```
gaussdb=# DBE_ILM.STOP_ILM (task_id => V_TASK, p_drop_running_Jobs => FALSE, p_Jobname =>  
V_JOBNAME);
```

表 6-2 DBE\_ILM.STOP\_ILM 输入参数

名称	描述
TASK_ID	指定待停止ADO task的描述符ID。
P_DROP_RUNNING_JOBS	是否停止正在执行中的任务，true为强制停止，false为不停止正在执行的任务。
P_JOBNAME	标识待停止的特定JobName，通过GS_MY_ILMEVALUATIONDETAILS视图可以查询。

3. 为表生成策略及后台调度压缩任务。

a. 给表加上冷热分离策略：

```
gaussdb=# DROP TABLE IF EXISTS ILM_TABLE;
gaussdb=# CREATE TABLE ILM_TABLE(a int);
gaussdb=# ALTER TABLE ILM_TABLE ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 3 MONTHS OF NO MODIFICATION;
```

b. 设置ILM执行相关参数：

```
BEGIN
DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
DBE_ILM_ADMIN.CUSTOMIZE_ILM(12, 10);
DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 512);
END;
/
```

c. 开启后台的定时调度：

```
gaussdb=# CALL DBE_ILM_ADMIN.DISABLE_ILM();
gaussdb=# CALL DBE_ILM_ADMIN.ENABLE_ILM();
```

d. 用户可以根据需要，调用DBE\_SCHEDULER.set\_attribute设置后台维护窗口的开启时间。当前默认22:00开启。

4. 设置ILM执行相关参数。

控制ADO的条件单位是天还是秒，秒仅用来做测试用。取值为：  
ILM\_POLICY\_IN\_SECONDS = 1或ILM\_POLICY\_IN\_DAYS = 0（默认值）：

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
```

控制一次ADO Task最多生成多少个ADO Job。取值范围大于等于0小于等于2147483647的整数或浮点数，作用时向下取整：

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(12, 10);
```

ADO Task的执行频率，单位分钟，默认值15。取值范围大于等于1小于等于2147483647的整数或浮点数，作用时向下取整：

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
```

控制单个ADO Job可以处理的最大字节数，单位兆。取值范围大于等于0小于等于2147483647的整数或浮点数，作用时向下取整：

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 512);
```

5. 评估一张表是否适合压缩及评估压缩后带来多少收益。

```
DBE_COMPRESSION.GET_COMPRESSION_RATIO(
SCRATCHTBSNAME IN VARCHAR2,
OWNNAME       IN VARCHAR2,
OBJNAME       IN VARCHAR2,
SUBOBJNAME    IN VARCHAR2,
COMPTYPE      IN NUMBER,
BLKCNT_CMP    OUT INTEGER,
BLKCNT_UNCMP  OUT INTEGER,
ROW_CMP       OUT INTEGER,
ROW_UNCMP     OUT INTEGER,
CMP_RATIO     OUT NUMBER,
```



```
COMPTYPE_STR OUT VARCHAR2,
SAMPLE_RATIO IN NUMBER DEFAULT 20,
OBJTYPE IN INTEGER DEFAULT 1)
```

**表 6-3** DBE\_COMPRESSION.GET\_COMPRESSION\_RATIO 输入参数

名称	描述
SCRATCHTBSNAME	数据对象所属表空间。
OWNNAME	数据对象所有者 (所属模式)。
OBJNAME	数据对象名称。
SUBOBJNAME	数据子对象名称。
COMPTYPE	<ul style="list-style-type: none"> <li>• 1: 未压缩</li> <li>• 2: 高级压缩</li> </ul>
SAMPLE_RATIO	采样比例, 输入为0-100的整数或浮点数, 对应为百分之N的采样比例。默认为20, 即对20%的行数进行采样。
OBJTYPE	对象类型, 支持: 1: 表对象

**表 6-4** DBE\_COMPRESSION.GET\_COMPRESSION\_RATIO 输出参数

名称	描述
BLKCNT_CMP	样本被压缩后占用的块数。
BLKCNT_UNCMP	样本未压缩占用的块数。
ROW_CMP	样本被压缩后单个块内可容纳的行数。
ROW_UNCMP	样本未被压缩时单个数据块可容纳的行数。
CMP_RATIO	压缩比, blkcnt_uncmp除以blkcnt_cmp。
COMPTYPE_STR	描述压缩类型的字符串。

**示例:**

```
gaussdb=# ALTER DATABASE set ilm = on;
gaussdb=# CREATE user user1 IDENTIFIED BY '*****';
gaussdb=# CREATE user user2 IDENTIFIED BY '*****';
gaussdb=# SET ROLE user1 PASSWORD '*****';
gaussdb=# CREATE TABLE TEST_DATA (ORDER_ID INT, GOODS_NAME TEXT, CREATE_TIME
TIMESTAMP)
ILM ADD POLICY ROW STORE COMPRESS ADVANCED ROW AFTER 1 DAYS OF NO MODIFICATION;
INSERT INTO TEST_DATA VALUES (1, '零食大礼包A', NOW());

DECLARE
o_blkcnt_cmp integer;
o_blkcnt_uncmp integer;
o_row_cmp integer;
```

```

o_row_uncmp    integer;
o_cmp_ratio    number;
o_comptype_str varchar2;
begin
dbe_compression.get_compression_ratio(
  SCRATCHTBSNAME => NULL,
  OWNNAME        => 'user1',
  OBJNAME        => 'test_data',
  SUBOBJNAME     => NULL,
  COMPTYPE       => 2,
  BLKCNT_CMP     => o_blkcnt_cmp,
  BLKCNT_UNCMP  => o_blkcnt_uncmp,
  ROW_CMP        => o_row_cmp,
  ROW_UNCMP     => o_row_uncmp,
  CMP_RATIO      => o_cmp_ratio,
  COMPTYPE_STR  => o_comptype_str,
  SAMPLE_RATIO  => 100,
  OBJTYPE       => 1);
RAISE INFO 'Number of blocks used by the compressed sample of the object      : %', o_blkcnt_cmp;
RAISE INFO 'Number of blocks used by the uncompressed sample of the object    : %',
o_blkcnt_uncmp;
RAISE INFO 'Number of rows in a block in compressed sample of the object      : %', o_row_cmp;
RAISE INFO 'Number of rows in a block in uncompressed sample of the object    : %', o_row_uncmp;
RAISE INFO 'Estimated Compression Ratio of Sample                          : %', o_cmp_ratio;
RAISE INFO 'Compression Type                                               : %', o_comptype_str;
end;
/
INFO: Number of blocks used by the compressed sample of the object      : 0
INFO: Number of blocks used by the uncompressed sample of the object    : 0
INFO: Number of rows in a block in compressed sample of the object      : 0
INFO: Number of rows in a block in uncompressed sample of the object    : 0
INFO: Estimated Compression Ratio of Sample                          : 1
INFO: Compression Type                                               : Compress Advanced

```

6. 查询每一行的最后修改时间。

```

DBE_HEAT_MAP.ROW_HEAT_MAP(
  OWNER          IN VARCHAR2,
  SEGMENT_NAME  IN VARCHAR2,
  PARTITION_NAME IN VARCHAR2  DEFAULT NULL,
  CTID          IN TEXT,
  V_DEBUG       IN BOOL       DEFAULT FALSE);

```

表 6-5 DBE\_HEAT\_MAP.ROW\_HEAT\_MAP 输入参数

名称	描述
OWNER	数据对象所属Schema。
SEGMENT_NAME	数据对象名称。
PARTITION_NAME	数据对象分区名，可选参数，默认为 NULL。
CTID	目标行的ctid，即block_id或row_id。
V_DEBUG	debug调试，增加日志打印。

表 6-6 DBE\_HEAT\_MAP.ROW\_HEAT\_MAP 输出参数

名称	描述
OWNER	数据对象的所有者。
SEGMENT_NAME	数据对象名称。

名称	描述
PARTITION_NAME	数据对象分区名称，可选参数。
TABLESPACE_NAME	数据所属的表空间名称。
FILE_ID	行所属的绝对文件ID。
RELATIVE_FNO	行所属的相对文件ID（GaussDB中无此逻辑，因此取值同上）。
CTID	行的ctid，即block_id或row_id。
WRITETIME	行的最后修改时间。

示例：

```
gaussdb=# ALTER DATABASE set ilm = on;
gaussdb=# CREATE Schema HEAT_MAP_DATA;
gaussdb=# SET current_schema=HEAT_MAP_DATA;

gaussdb=# CREATE TABLESPACE example1 RELATIVE LOCATION 'tablespace1';
gaussdb=# CREATE TABLE HEAT_MAP_DATA.heat_map_table(id INT, value TEXT) TABLESPACE
example1;
gaussdb=# INSERT INTO HEAT_MAP_DATA.heat_map_table VALUES (1, 'test_data_row_1');

gaussdb=# SELECT * from DBE_HEAT_MAP.ROW_HEAT_MAP(
owner      => 'heat_map_data',
segment_name => 'heat_map_table',
partition_name => NULL,
ctid       => '(0,1)');
owner | segment_name | partition_name | tablespace_name | file_id | relative_fno | ctid |
writetime
-----+-----+-----+-----+-----+-----+-----+-----
heat_map_data | heat_map_table |                | example1        | 17291 | 17291 | (0,1) |
(1 row)
```

7. 查询ILM调度与执行的相关环境参数。

```
gaussdb=# SELECT * FROM GS_ADM_ILMPARAMETERS;
name | value
-----+-----
EXECUTION_INTERVAL | 15
RETENTION_TIME | 30
ENABLED | 1
POLICY_TIME | 0
ABS_JOBLIMIT | 10
JOB_SIZELIMIT | 1024
WIND_DURATION | 240
BLOCK_LIMITS | 40
ENABLE_META_COMPRESSION | 0
SAMPLE_MIN | 10
SAMPLE_MAX | 10
CONST_PRIO | 40
CONST_THRESHOLD | 90
EQVALUE_PRIO | 60
EQVALUE_THRESHOLD | 80
ENABLE_DELTA_ENCODE_SWITCH | 1
LZ4_COMPRESSION_LEVEL | 0
ENABLE_LZ4_PARTIAL_DECOMPRESSION | 1
(18 rows)
```

8. 查询ILM策略的概要信息，包含策略名称、类型、启用禁用状态、删除状态。

```
gaussdb=# SELECT * FROM GS_ADM_ILMPOLICIES;
policy_name | policy_type | tablespace | enabled | deleted
-----+-----+-----+-----+-----
```

```
p1      | DATA MOVEMENT |      | YES | NO
```

```
gaussdb=# SELECT * FROM GS_MY_ILMPOLICIES;
policy_name | policy_type | tablespace | enabled | deleted
```

```
-----+-----+-----+-----+-----
p1      | DATA MOVEMENT |      | YES | NO
```

9. 查询ILM策略的数据移动概要信息，包含策略名称、动作类型、条件等。

```
gaussdb=# SELECT * FROM GS_ADM_ILMDATAMOVEMENTPOLICIES;
policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type | condition_days | custom_function | policy_subtype | action_clause | tier_to
```

```
-----+-----+-----+-----+-----+-----+-----
p1      | COMPRESSION | ROW | ADVANCED |      |      |
TIME | 90 |      |      |      |      |
```

```
gaussdb=# SELECT * FROM GS_MY_ILMDATAMOVEMENTPOLICIES;
policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type | condition_days | custom_function | policy_subtype | action_clause | tier_to
```

```
-----+-----+-----+-----+-----+-----+-----
p1      | COMPRESSION | ROW | ADVANCED |      |      |
TIME | 90 |      |      |      |      |
(1 row)
```

10. 查询所有存在ILM策略应用的数据对象与相应策略的概要信息，包含策略名称、数据对象名称、策略的来源、策略的启用删除状态。

```
gaussdb=# SELECT * FROM GS_ADM_ILMOBJECTS;
policy_name | object_owner | object_name | subobject_name | object_type | inherited_from |
tbs_inherited_from | enabled | deleted
```

```
-----+-----+-----+-----+-----+-----+-----
p1      | public | lineitem |      | TABLE | POLICY NOT INHERITED |
YES | NO
```

```
gaussdb=# SELECT * FROM GS_MY_ILMOBJECTS;
policy_name | object_owner | object_name | subobject_name | object_type | inherited_from |
tbs_inherited_from | enabled | deleted
```

```
-----+-----+-----+-----+-----+-----+-----
p1      | public | lineitem |      | TABLE | POLICY NOT INHERITED |
YES | NO
```

11. 查询ADO Task的概要信息，包含Task ID，Task Owner，状态以及时间信息。

```
gaussdb=# SELECT * FROM GS_ADM_ILMTASKS;
task_id | task_owner | state | creation_time | start_time |
completion_time
```

```
-----+-----+-----+-----+-----+-----+-----
1 | omm | COMPLETED | 2023-10-16 12:03:55.113296+08 | 2023-10-16 12:03:55.113296+08 |
2023-10-16 12:03:56.326864+08
(1 row)
```

```
gaussdb=# SELECT * FROM GS_MY_ILMTASKS;
task_id | task_owner | state | creation_time | start_time |
completion_time
```

```
-----+-----+-----+-----+-----+-----+-----
1 | omm | COMPLETED | 2023-10-16 12:03:55.113296+08 | 2023-10-16 12:03:55.113296+08 |
2023-10-16 12:03:56.326864+08
(1 row)
```

12. 查询ADO Task的评估详情信息，包含Task ID，策略信息、对象信息、评估结果以及ADO JOB名称。

```
gaussdb=# SELECT * FROM GS_ADM_ILMEVALUATIONDETAILS;
task_id | policy_name | object_owner | object_name | subobject_name | object_type |
selected_for_execution | job_name | comments
```

```

+-----+-----+
-
  1 | p2      | public  | ilm_table_1 |      | TABLE  | SELECTED FOR EXECUTION | ilmjob
$ _postgres1 |
(1 row)

gaussdb=# SELECT * FROM GS_MY_ILMEVALUATIONDETAILS;
 task_id | policy_name | object_owner | object_name | subobject_name | object_type |
selected_for_execution | job_name    | comments
-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
-
  1 | p2      | public  | ilm_table_1 |      | TABLE  | SELECTED FOR EXECUTION | ilmjob
$ _postgres1 |
(1 row)

```

13. 查询ADO JOB的执行详情信息，包含Task ID，JOB名称、JOB状态、JOB时间信息等。

```

gaussdb=# SELECT * FROM GS_ADM_ILMRESULTS;
 task_id | job_name  | job_state  | start_time  | completion_time  |
comments | statistics
-----+-----+-----+-----+-----+-----+
+-----+-----+
-
  1 | ilmjob$ _postgres1 | COMPLETED SUCCESSFULLY | 2023-10-16 12:03:56.290176+08 |
2023-10-16 12:03:56.319829+08 |      | SpaceSaving=0,BoundTime=1697429033,LastBlkNum=40
(1 row)

gaussdb=# SELECT * FROM GS_MY_ILMRESULTS;
 task_id | job_name | job_state | start_time | completion_time | comments | statistics
-----+-----+-----+-----+-----+-----+
(0 rows)
 task_id | job_name  | job_state  | start_time  | completion_time  |
comments | statistics
-----+-----+-----+-----+-----+
+-----+-----+
-
  1 | ilmjob$ _postgres1 | COMPLETED SUCCESSFULLY | 2023-10-16 12:03:56.290176+08 |
2023-10-16 12:03:56.319829+08 |      | SpaceSaving=0,BoundTime=1697429033,LastBlkNum=40
(1 row)

```

# 7 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。

## 7.1 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、系统管理员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。
- 创建外表：CREATE FOREIGN TABLE。

#### 说明

- 外表的表结构需要与指定的文件的数据保持一致。
  - 对外表做查询操作，写操作不被允许。
- 删除外表：DROP FOREIGN TABLE。
- 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。

# 8 动态数据脱敏

数据脱敏是行之有效的数据库隐私保护方案之一，可以在一定程度上限制非授权用户对隐私数据的窥探。动态数据脱敏机制是一种通过定制化制定脱敏策略从而实现对隐私数据保护的一种技术，可以有效地在保留原始数据的前提下解决非授权用户对敏感信息的访问问题。当管理员指定待脱敏对象和定制数据脱敏策略后，用户所查询的数据库资源如果关联到对应的脱敏策略时，则会根据用户身份和脱敏策略进行数据脱敏，从而限制非授权用户对隐私数据的访问。

## 查看动态数据脱敏基本配置

**步骤1** 设置并查看动态数据脱敏功能是否已开启。

```
gs_guc reload -Z coordinator -N all -I all -c "enable_security_policy=on"
```

enable\_security\_policy取值为on时表示开启，取值为off是表示关闭。

```
gaussdb=# SHOW enable_security_policy;
enable_security_policy
-----
on
(1 row)
```

----结束

## 创建脱敏策略

**步骤1** 创建数据表。

--创建一个表tb\_for\_masking。

```
gaussdb=# CREATE TABLE tb_for_masking(idx int, col1 text, col2 text, col3 text, col4 text, col5 text, col6 text, col7 text,col8 text);
```

--给表tb\_for\_masking插入数据。

```
gaussdb=# INSERT INTO tb_for_masking VALUES(1, '9876543210', 'usr321usr', 'abc@example.com', 'abc@example.com', '1234-4567-7890-0123', 'abcdef 123456 ui 323 jsfd321 j3k2l3', '4880-9898-4545-2525', 'this is a llt case');
```

--查看数据。

```
gaussdb=# SELECT * FROM tb_for_masking;
```

```
idx | col1 | col2 | col3 | col4 | col5 | col6
-----+-----+-----+-----+-----+-----+-----
1 | 9876543210 | usr321usr | abc@example.com | abc@example.com | 1234-4567-7890-0123 | abcdef
```

```
123456 ui 323 jsfd321 j
```

```
3k2l3 | 4880-9898-4545-2525 | this is a llt case
```

```
(1 row)
```



## 步骤2 创建资源标签。

```
--创建资源标签标记敏感列col1。  
gaussdb=# CREATE RESOURCE LABEL mask_lb1 ADD COLUMN(tb_for_masking.col1);  
CREATE RESOURCE LABEL  
gaussdb=# CREATE RESOURCE LABEL mask_lb5 ADD COLUMN(tb_for_masking.col5);  
CREATE RESOURCE LABEL
```

## 步骤3 创建脱敏策略。

该语法详细格式参考：《开发指南》中“SQL参考 > SQL语法 > CREATE MASKING POLICY”章节。

```
--创建一个名为maskpol1的脱敏策略。  
gaussdb=# CREATE MASKING POLICY maskpol1 maskall ON LABEL(mask_lb1);  
CREATE MASKING POLICY
```

动态数据脱敏配置脱敏策略时，对用户创建的自定义函数进行支持适配。

```
gaussdb=# create or replace function msk_creditcard(col text) returns TEXT as $$  
declare  
    result TEXT;  
begin  
    result := overlay(col placing 'xxxx-xxxx' from 6);  
    return result;  
end;  
$$ language plpgsql security DEFINER;  
CREATE FUNCTION  
--创建一个名为maskpol5的脱敏策略。  
gaussdb=# CREATE MASKING POLICY maskpol5 msk_creditcard ON LABEL(mask_lb5);  
CREATE MASKING POLICY
```

## 步骤4 查新tb\_for\_masking表的脱敏列数据。

```
--访问tb_for_masking表，col1列触发脱敏策略。  
gaussdb=# SELECT col1 FROM tb_for_masking;  
col1  
-----  
xxxxxxxxxx  
(1 row)  
--访问tb_for_masking表，col5列触发脱敏策略。  
gaussdb=# SELECT col5 FROM tb_for_masking;  
col5  
-----  
1234-xxxx-xxxx-0123  
(1 row)
```

## 步骤5 清理数据。

```
--删除脱敏策略。  
gaussdb=# DROP MASKING POLICY maskpol1, maskpol5;  
DROP MASKING POLICY  
--删除资源标签。  
gaussdb=# DROP RESOURCE LABEL mask_lb1, mask_lb5;  
DROP RESOURCE LABEL  
--删除表tb_for_masking。  
gaussdb=# DROP TABLE tb_for_masking;  
DROP TABLE
```

---结束