

云数据库 GaussDB

开发指南（分布式_V2.0-10.x）

文档版本	01
发布日期	2026-07-28



版权所有 © 华为云计算技术有限公司 2026。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

1 数据库系统概述.....	1
1.1 数据库逻辑结构图.....	1
1.2 数据查询请求处理过程.....	2
1.3 管理事务.....	3
2 数据库安全.....	6
2.1 用户及权限.....	6
2.1.1 默认权限机制.....	6
2.1.2 管理员.....	6
2.1.3 用户.....	8
2.1.4 角色.....	9
2.1.5 Schema.....	11
2.1.6 用户权限设置.....	12
2.1.7 行级访问控制.....	13
2.2 数据库审计.....	15
3 数据库使用入门.....	17
3.1 连接数据库.....	17
3.1.1 使用 gsql 连接.....	17
3.1.2 应用程序接口.....	19
3.2 操作数据库.....	19
3.2.1 创建数据库用户.....	19
3.2.2 管理数据库-非透明多写.....	20
3.2.3 管理表空间.....	23
3.2.4 管理表.....	26
3.2.4.1 创建表.....	26
3.2.4.2 向表中插入数据.....	26
3.2.4.3 更新表中数据.....	29
3.2.4.4 查看数据.....	31
3.2.4.5 删除表中数据.....	31
3.2.5 查看系统表.....	32
3.2.6 其他操作.....	34
3.2.6.1 管理 Schema.....	34
3.2.6.2 管理分区表.....	37

3.2.6.3 管理索引.....	41
3.2.6.4 管理视图.....	44
3.2.6.5 管理序列.....	45
3.2.6.6 管理定时任务.....	47
3.3 管理权限.....	50
3.4 使用存储过程.....	52
4 开发设计建议.....	57
4.1 概述.....	57
4.2 数据库设计规范.....	58
4.2.1 基本规范.....	58
4.2.2 部署规范.....	60
4.2.3 数据库对象命名规范.....	60
4.2.4 Database 和 Schema 设计规范.....	61
4.2.5 权限设计规范.....	63
4.2.6 字符集设计规范.....	63
4.2.7 表设计规范.....	64
4.2.8 字段设计规范.....	68
4.2.9 索引设计规范.....	71
4.2.10 函数/存储过程设计规范.....	72
4.2.11 约束设计.....	72
4.2.12 视图和关联表设计.....	73
4.3 数据库编程规范.....	73
4.3.1 GUC 参数编程规范.....	73
4.3.2 对象访问编程规范.....	73
4.3.3 WHERE.....	73
4.3.4 SELECT.....	75
4.3.5 INSERT.....	76
4.3.6 UPDATE.....	76
4.3.7 DELETE.....	76
4.3.8 关联查询.....	77
4.3.9 子查询.....	77
4.3.10 事务.....	78
4.3.11 SQL 编写.....	79
4.4 客户端编程规范.....	82
4.4.1 JDBC.....	82
4.4.2 连接控制规范.....	84
4.4.3 C/C++应用程序编译规范.....	84
5 应用程序开发教程.....	89
5.1 GaussDB 应用程序开发教程概述.....	89
5.2 开发规范.....	100
5.3 基于 JDBC 开发.....	100
5.3.1 开发流程.....	100

5.3.2 开发步骤.....	101
5.3.2.1 获取驱动 jar 包并配置 JDK 环境.....	101
5.3.2.2 连接数据库.....	105
5.3.2.2.1 连接介绍.....	105
5.3.2.2.2 以非加密方式连接.....	109
5.3.2.2.3 以 SSL 方式连接.....	110
5.3.2.2.4 以 UDS 方式连接.....	111
5.3.2.2.5 以国密 TLS 方式连接.....	112
5.3.2.2.6 连接参数参考.....	115
5.3.2.3 执行 SQL 语句.....	151
5.3.2.4 处理结果集.....	155
5.3.2.5 关闭数据库连接.....	158
5.3.3 典型应用开发示例.....	158
5.3.3.1 不同场景下配置连接参数.....	158
5.3.3.2 处理 SQL 异常.....	162
5.3.3.3 创建和调用存储过程.....	162
5.3.3.4 获取函数返回值.....	164
5.3.3.5 批量查询.....	166
5.3.3.6 使用流式读功能.....	167
5.3.3.7 应用层 SQL 重试.....	170
5.3.3.8 通过本地文件导入导出数据.....	174
5.3.3.9 逻辑复制（非透明多写）.....	175
5.3.3.10 监控数据库连接.....	183
5.3.3.11 使用 Array 类型.....	184
5.3.3.12 使用 Struct 类型.....	195
5.3.4 JDBC 接口参考.....	202
5.3.4.1 常用 Package 及接口.....	202
5.3.4.2 java.sql.Connection.....	204
5.3.4.3 java.sql.CallableStatement.....	211
5.3.4.4 java.sql.DatabaseMetaData.....	218
5.3.4.5 java.sql.Driver.....	227
5.3.4.6 java.sql.PreparedStatement.....	228
5.3.4.7 java.sql.ResultSet.....	232
5.3.4.8 java.sql.ResultSetMetaData.....	241
5.3.4.9 java.sql.Statement.....	242
5.3.4.10 java.sql.Clob.....	245
5.3.4.11 java.sql.Array.....	246
5.3.4.12 java.sql.Struct.....	248
5.3.4.13 java.sql.SQLException.....	249
5.3.4.14 java.io.IOException.....	250
5.3.4.15 javax.sql.ConnectionPoolDataSource.....	250
5.3.4.16 javax.sql.DataSource.....	250

5.3.4.17 javax.sql.PooledConnection.....	251
5.3.4.18 javax.naming.Context.....	251
5.3.4.19 CopyManager.....	252
5.3.4.20 PGReplicationConnection.....	253
5.3.4.21 PGReplicationStream.....	254
5.3.4.22 ChainedStreamBuilder.....	256
5.3.4.23 ChainedCommonStreamBuilder.....	256
5.3.4.24 PGObject.....	257
5.3.4.25 LogSequenceNumber.....	258
5.4 基于 ODBC 开发.....	259
5.4.1 开发流程.....	259
5.4.2 开发步骤.....	259
5.4.2.1 获取源码包、ODBC 包以及依赖库.....	259
5.4.2.2 连接数据库.....	261
5.4.2.2.1 Linux 系统下配置数据源.....	261
5.4.2.2.2 Windows 系统下配置数据源.....	265
5.4.2.2.3 连接数据库涉及的 API.....	269
5.4.2.3 执行 SQL 语句.....	270
5.4.2.4 处理结果集.....	271
5.4.2.5 关闭连接.....	272
5.4.3 典型应用开发示例.....	273
5.4.3.1 典型应用场景配置.....	273
5.4.3.2 获取和处理数据库中的数据.....	276
5.4.3.3 批量绑定.....	278
5.4.3.4 高性能绑定类型.....	283
5.4.3.5 开启连接池.....	290
5.4.4 ODBC 接口参考.....	293
5.4.4.1 SQLAllocEnv.....	293
5.4.4.2 SQLAllocConnect.....	293
5.4.4.3 SQLAllocHandle.....	293
5.4.4.4 SQLAllocStmt.....	295
5.4.4.5 SQLBindCol.....	295
5.4.4.6 SQLBindParameter.....	296
5.4.4.7 SQLColAttribute.....	297
5.4.4.8 SQLConnect.....	298
5.4.4.9 SQLDisconnect.....	299
5.4.4.10 SQLExecDirect.....	300
5.4.4.11 SQLExecute.....	301
5.4.4.12 SQLFetch.....	302
5.4.4.13 SQLFreeStmt.....	303
5.4.4.14 SQLFreeConnect.....	303
5.4.4.15 SQLFreeHandle.....	303

5.4.4.16 SQLFreeEnv.....	304
5.4.4.17 SQLPrepare.....	304
5.4.4.18 SQLGetData.....	305
5.4.4.19 SQLGetDiagRec.....	306
5.4.4.20 SQLSetConnectAttr.....	308
5.4.4.21 SQLSetEnvAttr.....	309
5.4.4.22 SQLSetStmtAttr.....	310
5.5 基于 libpq 开发.....	311
5.5.1 开发流程.....	311
5.5.2 开发步骤.....	312
5.5.2.1 获取发布包、依赖库和头文件.....	312
5.5.2.2 连接数据库.....	314
5.5.2.3 执行 SQL 语句.....	315
5.5.2.4 处理结果集.....	316
5.5.2.5 关闭连接.....	316
5.5.3 典型应用开发示例.....	316
5.5.3.1 数据库建连、执行 SQL 并返回结果.....	317
5.5.3.2 执行预备语句.....	319
5.5.3.3 绑定参数并返回二进制结果.....	321
5.5.4 libpq 接口参考.....	324
5.5.4.1 概述.....	324
5.5.4.2 数据库连接控制函数.....	325
5.5.4.2.1 PQconnectdbParams.....	325
5.5.4.2.2 PQconnectdb.....	325
5.5.4.2.3 PQconninfoParse.....	326
5.5.4.2.4 PQconnectStart.....	327
5.5.4.2.5 PQerrorMessage.....	327
5.5.4.2.6 PQsetdbLogin.....	327
5.5.4.2.7 PQfinish.....	328
5.5.4.2.8 PQreset.....	329
5.5.4.2.9 PQstatus.....	329
5.5.4.3 数据库执行语句函数.....	330
5.5.4.3.1 PQexec.....	330
5.5.4.3.2 PQprepare.....	332
5.5.4.3.3 PQclear.....	333
5.5.4.3.4 PQfnnumberEx.....	333
5.5.4.3.5 PQexecParams.....	334
5.5.4.3.6 PQexecParamsBatch.....	335
5.5.4.3.7 PQexecPrepared.....	336
5.5.4.3.8 PQexecPreparedBatch.....	337
5.5.4.4 异步命令处理函数.....	338
5.5.4.4.1 PQsendQuery.....	338

5.5.4.4.2 PQsendQueryParams.....	339
5.5.4.4.3 PQsendPrepare.....	340
5.5.4.4.4 PQsendQueryPrepared.....	340
5.5.4.4.5 PQflush.....	341
5.5.4.5 取消查询处理中函数.....	342
5.5.4.5.1 PQgetCancel.....	342
5.5.4.5.2 PQfreeCancel.....	343
5.5.4.5.3 PQcancel.....	343
5.5.4.6 数据库结果处理函数.....	344
5.5.4.6.1 PQgetvalue.....	344
5.5.4.6.2 PQnfields.....	344
5.5.4.6.3 PQntuples.....	345
5.5.4.6.4 PQfname.....	346
5.5.4.6.5 PQresultStatus.....	346
5.6 基于 Psycopg 开发.....	347
5.6.1 开发流程.....	347
5.6.2 开发步骤.....	348
5.6.2.1 环境准备.....	348
5.6.2.2 连接数据库.....	353
5.6.2.2.1 连接介绍.....	353
5.6.2.2.2 以非加密方式连接.....	353
5.6.2.2.3 以 SSL 方式连接.....	354
5.6.2.2.4 连接参数参考.....	355
5.6.2.3 执行 SQL 语句.....	359
5.6.2.4 处理结果集.....	360
5.6.2.5 关闭数据库连接.....	360
5.6.3 典型应用开发示例.....	361
5.6.3.1 数据库建连、执行 SQL 并返回结果.....	361
5.6.3.2 使用日志场景.....	362
5.6.4 Psycopg 接口参考.....	363
5.6.4.1 psycopg2.connect().....	363
5.6.4.2 connection.cursor().....	364
5.6.4.3 cursor.execute(query,vars_list).....	364
5.6.4.4 cursor.executemany(query,vars_list).....	365
5.6.4.5 connection.commit().....	366
5.6.4.6 connection.rollback().....	366
5.6.4.7 cursor.fetchone().....	367
5.6.4.8 cursor.fetchmany(size).....	367
5.6.4.9 cursor.fetchall().....	368
5.6.4.10 cursor.close().....	369
5.6.4.11 connection.close().....	369
5.7 基于 Go 驱动开发.....	370

5.7.1 开发流程.....	370
5.7.2 开发步骤.....	370
5.7.2.1 环境准备.....	370
5.7.2.2 连接数据库.....	373
5.7.2.2.1 以非加密方式连接.....	373
5.7.2.2.2 以 SSL 方式连接.....	376
5.7.2.2.3 连接参数参考.....	378
5.7.2.3 执行 SQL 语句.....	383
5.7.2.4 处理结果集.....	384
5.7.2.5 关闭数据库连接.....	385
5.7.3 典型应用开发示例.....	385
5.7.4 Go 接口参考.....	389
5.7.4.1 type DB.....	389
5.7.4.2 type Stmt.....	391
5.7.4.3 type Tx.....	392
5.7.4.4 type Rows.....	394
5.7.4.5 type Row.....	395
5.7.4.6 type ColumnType.....	395
5.7.4.7 type Result.....	396
5.8 基于 ecpg 开发.....	396
5.8.1 开发流程.....	396
5.8.2 准备 ecpg 开发相关组件.....	398
5.8.3 嵌入式 SQL-C 程序开发.....	399
5.8.3.1 核心概念简介.....	399
5.8.3.2 声明宿主变量.....	400
5.8.3.3 管理数据库连接.....	401
5.8.3.4 执行 SQL 语句.....	403
5.8.3.5 处理结果集.....	405
5.8.3.6 关闭连接.....	408
5.8.3.7 嵌入式 SQL 语法参考.....	408
5.8.3.7.1 ALLOCATE DESCRIPTOR.....	408
5.8.3.7.2 CONNECT.....	409
5.8.3.7.3 DEALLOCATE DESCRIPTOR.....	412
5.8.3.7.4 DECLARE.....	412
5.8.3.7.5 DESCRIBE.....	413
5.8.3.7.6 DISCONNECT.....	414
5.8.3.7.7 EXECUTE IMMEDIATE.....	415
5.8.3.7.8 GET DESCRIPTOR.....	415
5.8.3.7.9 OPEN.....	417
5.8.3.7.10 PREPARE.....	417
5.8.3.7.11 SET AUTOCOMMIT.....	418
5.8.3.7.12 SET CONNECTION.....	418

5.8.3.7.13 SET DESCRIPTOR.....	419
5.8.3.7.14 TYPE.....	419
5.8.3.7.15 VAR.....	421
5.8.3.7.16 WHENEVER.....	421
5.8.4 ecpg 预处理、编译、执行.....	423
5.8.5 典型应用开发示例.....	426
5.8.5.1 创建表、插入数据并查询.....	426
5.8.5.2 日期、时间戳与区间解析.....	428
5.8.5.3 解析、运算数值类型.....	431
5.8.6 ecpg 接口参考.....	436
5.8.6.1 概述.....	436
5.8.6.2 区间类型.....	437
5.8.6.3 数值类型.....	437
5.8.6.4 日期类型.....	440
5.8.6.5 时间戳类型.....	443
5.9 兼容性参考.....	444
5.10 附录.....	445
5.10.1 JDBC.....	445
5.10.1.1 数据类型映射关系.....	445
5.10.1.2 数据类型转换规则.....	446
5.10.1.3 日志管理.....	499
5.10.1.4 常见问题处理.....	503
5.10.1.4.1 batchMode 设置错误.....	503
5.10.1.4.2 Hibernate 框架插入数据开启校验时报错.....	504
5.10.1.4.3 使用 SSL 方式建连报错或阻塞.....	506
5.10.2 ODBC.....	507
5.10.2.1 配置参数说明.....	507
5.10.3 libpq.....	514
5.10.3.1 连接参数说明.....	514
5.10.4 ecpg.....	522
5.10.4.1 预备语句.....	522
5.10.4.2 数据类型和宿主变量.....	522
5.10.4.2.1 概述.....	522
5.10.4.2.2 声明段.....	523
5.10.4.2.3 检索查询.....	523
5.10.4.2.4 数据类型映射.....	524
5.10.4.2.5 处理字符串.....	525
5.10.4.2.6 使用非初级类型的宿主变量.....	525
5.10.4.2.7 访问特殊数据类型.....	527
5.10.4.2.8 处理非初级 SQL 数据类型.....	529
5.10.4.3 执行动态 SQL 语句.....	532
5.10.4.3.1 概述.....	532

5.10.4.3.2 执行没有结果集的语句.....	532
5.10.4.3.3 执行具有输入参数的语句.....	532
5.10.4.3.4 执行带有结果集的语句.....	532
5.10.4.4 错误处理.....	533
5.10.4.4.1 概述.....	533
5.10.4.4.2 设置回调.....	533
5.10.4.4.3 sqlca.....	534
5.10.4.4.4 SQLSTATE 与 SQLCODE.....	535
5.10.4.5 使用库函数.....	539
5.10.4.6 SQL 描述符区域.....	542
5.10.4.6.1 概述.....	542
5.10.4.6.2 命名 SQL 描述符区域.....	542
5.10.4.6.3 SQLDA.....	543
5.10.4.7 ecpg 与 Pro*C 兼容性对比.....	546
5.10.5 日志输出相关参数介绍.....	546
6 用户自定义函数.....	550
6.1 PL/SQL 语言函数.....	550
7 特性使用指导.....	551
7.1 自治事务.....	551
7.1.1 规格约束.....	551
7.1.2 存储过程支持自治事务.....	556
7.1.3 匿名块支持自治事务.....	557
7.1.4 函数支持自治事务.....	558
7.1.5 PACKAGE 支持自治事务.....	558
7.2 物化视图.....	560
7.2.1 全量物化视图.....	560
7.2.1.1 概述.....	560
7.2.1.2 支持和约束.....	560
7.2.1.3 使用.....	561
7.2.2 增量物化视图.....	562
7.2.2.1 概述.....	562
7.2.2.2 支持和约束.....	562
7.2.2.3 使用.....	563
7.3 设置密态等值查询.....	564
7.3.1 密态等值查询概述.....	564
7.3.2 使用 gsql 操作密态数据库.....	567
7.3.3 使用 JDBC 操作密态数据库.....	570
7.3.4 前向兼容与安全增强.....	575
7.3.5 密态支持函数/存储过程.....	578
7.4 透明数据加密.....	579
7.5 设置账本数据库.....	585
7.5.1 账本数据库概述.....	585

7.5.2 查看账本历史操作记录.....	588
7.5.3 校验账本数据一致性.....	589
7.5.4 归档账本数据库.....	590
7.5.5 修复账本数据库.....	591
7.6 基于标签的强制访问控制.....	594
7.6.1 定义安全标签.....	594
7.6.2 应用安全标签.....	595
7.6.3 基于标签的强制访问控制检查.....	595
7.7 逻辑复制.....	596
7.7.1 逻辑解码（非透明多写）.....	597
7.7.1.1 逻辑解码概述.....	597
7.7.1.2 逻辑解码选项.....	604
7.7.1.3 使用 SQL 函数接口进行逻辑解码.....	618
7.7.1.4 使用流式解码实现数据逻辑复制.....	618
7.7.1.5 逻辑解码支持 DDL.....	619
7.7.1.6 逻辑解码数据找回功能.....	628
7.8 分区表.....	629
7.8.1 表分区介绍.....	629
7.8.1.1 大容量数据库背景介绍.....	629
7.8.1.2 表分区技术.....	629
7.8.1.3 数据分区查找优化.....	630
7.8.1.4 数据分区运维管理.....	630
7.8.2 分区表介绍.....	631
7.8.2.1 基本概念.....	631
7.8.2.1.1 分区表（分区母表）.....	631
7.8.2.1.2 分区.....	633
7.8.2.1.3 分区键.....	633
7.8.2.2 分区策略.....	634
7.8.2.2.1 范围分区.....	634
7.8.2.2.2 哈希分区.....	636
7.8.2.2.3 列表分区.....	637
7.8.2.2.4 二级分区.....	638
7.8.2.2.5 分区表对导入操作的性能影响.....	641
7.8.2.3 分区基本使用.....	643
7.8.2.3.1 创建分区表.....	643
7.8.2.3.2 使用和管理分区表.....	645
7.8.2.3.3 分区表 DQL/DML.....	646
7.8.3 分区表查询优化.....	647
7.8.3.1 分区剪枝.....	648
7.8.3.1.1 分区表静态剪枝.....	648
7.8.3.1.2 分区表动态剪枝.....	652
7.8.3.2 分区算子执行优化.....	659

7.8.3.2.1 Partition Iterator 算子消除.....	659
7.8.3.2.2 Merge Append.....	660
7.8.3.2.3 Max/Min.....	662
7.8.3.2.4 动态剪枝性能优化.....	663
7.8.3.3 分区索引.....	664
7.8.3.4 分区表统计信息.....	667
7.8.3.4.1 级联收集统计信息.....	668
7.8.3.4.2 分区级统计信息.....	670
7.8.3.5 Partition-wise Join.....	674
7.8.3.5.1 非 SMP 场景下的 Partition-wise Join.....	674
7.8.3.5.2 SMP 场景下的 Full Partition-wise Join.....	676
7.8.3.5.3 SMP 场景下的 Partial Partition-wise Join.....	681
7.8.4 分区表运维管理.....	683
7.8.4.1 新增分区.....	684
7.8.4.1.1 范围分区表新增分区.....	684
7.8.4.1.2 列表分区表新增分区.....	685
7.8.4.1.3 二级分区表新增一级分区.....	685
7.8.4.1.4 二级分区表新增二级分区.....	686
7.8.4.2 删除分区.....	686
7.8.4.2.1 对一级分区表删除分区.....	686
7.8.4.2.2 对二级分区表删除一级分区.....	687
7.8.4.2.3 对二级分区表删除二级分区.....	687
7.8.4.3 交换分区.....	687
7.8.4.3.1 对一级分区表交换分区.....	688
7.8.4.3.2 对二级分区表交换二级分区.....	689
7.8.4.4 清空分区.....	689
7.8.4.4.1 对一级分区表清空分区.....	689
7.8.4.4.2 对二级分区表清空一级分区.....	690
7.8.4.4.3 对二级分区表清空二级分区.....	690
7.8.4.5 分割分区.....	690
7.8.4.5.1 对范围分区表分割分区.....	690
7.8.4.5.2 对列表分区表分割分区.....	691
7.8.4.5.3 对*-RANGE 二级分区表分割二级分区.....	692
7.8.4.5.4 对*-LIST 二级分区表分割二级分区.....	692
7.8.4.6 合并分区.....	693
7.8.4.6.1 对一级分区表合并分区.....	693
7.8.4.6.2 对二级分区表合并二级分区.....	693
7.8.4.7 移动分区.....	693
7.8.4.7.1 对一级分区表移动分区.....	693
7.8.4.7.2 对二级分区表移动二级分区.....	694
7.8.4.8 重命名分区.....	694
7.8.4.8.1 对一级分区表重命名分区.....	694

7.8.4.8.2 对二级分区表重命名一级分区.....	694
7.8.4.8.3 对二级分区表重命名二级分区.....	694
7.8.4.8.4 对 Local 索引重命名索引分区.....	694
7.8.4.9 分区表行迁移.....	694
7.8.4.10 分区表索引重建/不可用.....	695
7.8.4.10.1 索引重建/不可用.....	695
7.8.4.10.2 Local 索引分区重建/不可用.....	695
7.8.5 分区并发控制.....	696
7.8.5.1 常规锁设计.....	697
7.8.5.2 DQL/DML-DQL/DML 并发.....	698
7.8.5.3 DQL/DML-DDL 并发.....	698
7.8.6 分区表系统视图&DFX.....	701
7.8.6.1 分区表相关系统视图.....	701
7.8.6.2 分区表相关内置工具函数.....	702
7.9 存储引擎.....	705
7.9.1 存储引擎体系架构.....	705
7.9.1.1 存储引擎体系架构概述.....	705
7.9.1.1.1 静态编译架构.....	705
7.9.1.1.2 通用数据库服务层.....	706
7.9.1.2 设置存储引擎.....	707
7.9.1.3 存储引擎更新说明.....	708
7.9.1.3.1 GaussDB 内核 505 版本.....	708
7.9.1.3.2 GaussDB 内核 503 版本.....	708
7.9.1.3.3 GaussDB 内核 R2 版本.....	708
7.9.2 Astore 存储引擎.....	709
7.9.2.1 Astore 简介.....	709
7.9.3 Ustore 存储引擎.....	709
7.9.3.1 Ustore 简介.....	709
7.9.3.1.1 Ustore 特性与规格.....	710
7.9.3.1.2 使用 Ustore 进行测试.....	710
7.9.3.1.3 Ustore 的最佳实践.....	711
7.9.3.2 存储格式.....	714
7.9.3.2.1 RCR Uheap.....	714
7.9.3.2.2 UBTree.....	716
7.9.3.2.3 Undo.....	721
7.9.3.2.4 Enhanced Toast.....	722
7.9.3.3 Ustore 事务模型.....	726
7.9.3.3.1 事务提交.....	726
7.9.3.3.2 事务回滚.....	726
7.9.3.4 闪回恢复.....	727
7.9.3.4.1 闪回查询.....	727
7.9.3.4.2 闪回表.....	729

7.9.3.4.3 闪回 DROP/TRUNCATE.....	731
7.9.3.5 常用视图工具.....	739
7.9.3.6 常见问题及定位手段.....	742
7.9.3.6.1 snapshot too old.....	743
7.9.3.6.2 storage test error.....	744
7.9.3.6.3 备机读业务报错:"Index scan conflicts with replay. Try again later. The original maxoff is <number>, but now it is <number>.".....	744
7.9.3.6.4 长查询执行期间大量并发更新偶现写入性能下降.....	745
7.9.4 数据生命周期管理-OLTP 表压缩.....	746
7.9.4.1 特性简介.....	746
7.9.4.2 特性约束.....	746
7.9.4.3 特性规格.....	749
7.9.4.4 使用说明.....	752
7.9.4.5 维护窗口参数配置.....	756
7.9.4.6 运维 TIPS.....	758
7.10 内存管理.....	765
7.11 多线程架构.....	770
7.12 ABO 优化器.....	773
7.12.1 智能基数估计.....	773
7.12.1.1 特性概述.....	773
7.12.1.2 技术原理.....	774
7.12.1.3 操作指导.....	776
7.12.1.4 最佳实践.....	779
7.12.1.5 常见问题.....	780
7.12.2 自适应代价估计.....	781
7.12.2.1 特性概述.....	781
7.12.2.2 技术原理.....	783
7.12.2.3 操作指导.....	788
7.12.2.4 常见问题.....	793
7.13 CBO 优化器.....	793
7.13.1 特性概述.....	793
7.13.2 技术原理.....	794
7.13.2.1 统计信息.....	794
7.13.2.1.1 单列统计信息.....	794
7.13.2.1.2 多列统计信息.....	800
7.13.2.2 路径生成.....	807
7.13.2.2.1 扫描路径生成.....	809
7.13.2.2.2 连接路径生成.....	813
7.13.2.3 计划生成.....	816
7.13.2.3.1 代价估算.....	817
7.13.2.3.2 计划选择.....	819
7.13.3 操作指导.....	823
7.13.4 常见问题.....	824

7.14 AutoHint：智能 HINT 推荐.....	829
7.15 基于 DSL 的规则引擎.....	831
7.15.1 DSL 规则引擎整体流程.....	831
7.15.2 DSL 规则编写规范.....	843
7.15.2.1 DSL 规则语法组件介绍.....	844
7.15.2.2 DSL 规则示例与说明.....	887
7.15.3 DSL 规则引擎最佳实践.....	890
7.15.4 常见问题处理.....	892
7.16 Foreign Data Wrapper.....	893
7.16.1 file_fdw.....	893
7.17 GTM.....	895
7.17.1 特性概述.....	895
7.17.2 GTM 模式.....	897
7.17.2.1 GTM.....	897
7.17.2.1.1 GTM 模式简介.....	897
7.17.2.1.2 技术原理.....	898
7.17.2.2 GTM-Lite.....	898
7.17.2.2.1 GTM-Lite 模式简介.....	898
7.17.2.2.2 技术原理.....	898
7.17.2.3 GTM-Free.....	898
7.17.2.3.1 GTM-Free 模式简介.....	899
7.17.2.3.2 技术原理.....	900
7.17.3 操作指导.....	900
7.17.4 分布式最佳实践.....	900
7.17.5 常见问题.....	901
7.18 敏感数据发现.....	901
7.19 动态数据脱敏.....	902
7.20 HTAP 混合负载.....	903
7.20.1 概述.....	903
7.20.2 使用指导.....	905
7.20.2.1 参数设置.....	905
7.20.2.2 内存管控.....	905
7.20.2.3 DDL.....	908
7.20.2.3.1 创建和卸载 IMCV.....	908
7.20.2.3.2 其他 DDL.....	916
7.20.2.4 DML/DQL.....	919
7.20.2.4.1 DML.....	919
7.20.2.4.2 DQL.....	920
7.20.2.4.3 列回行混合扫描.....	921
7.20.2.5 混合负载透明路由.....	922
7.20.2.6 列存数据维护.....	926
7.20.2.7 备机能力.....	930

7.20.2.8 列存数据磁盘扩展.....	930
7.20.2.9 列存数据压缩.....	931
7.21 在线 DDL.....	932
7.22 bucket 分布表.....	935
7.22.1 hashbucket.....	935
7.22.1.1 创建 hashbucket 表.....	936
7.22.1.2 bucket 索引.....	937
7.22.1.2.1 bucket 索引推荐使用场景.....	937
7.22.1.3 hashbucket 表特性规格约束.....	938
7.22.1.4 hashbucket 剪枝.....	939
7.22.2 bucket 分布表最佳实践.....	941
7.22.2.1 设置 segment_buffers.....	941
7.22.2.2 bucket 分布表的空间预占用.....	942
7.23 失效重编译.....	944
7.23.1 特性概述.....	944
7.23.2 技术原理.....	945
7.23.3 操作指导.....	946
7.23.4 常见问题.....	949
7.24 存储过程性能定位方法.....	951
7.24.1 存储过程执行期间性能定位方法.....	951
7.24.1.1 pg_stat_activity 查询存储过程正在执行的语句.....	951
7.24.1.1.1 特性概述.....	952
7.24.1.1.2 技术原理.....	952
7.24.1.1.3 操作指导.....	953
7.24.1.2 trace 存储过程执行的语句.....	954
7.24.1.2.1 特性概述.....	954
7.24.1.2.2 技术原理.....	955
7.24.1.2.3 操作指导.....	956
7.24.2 存储过程执行完成后性能定位方法.....	957
7.24.2.1 存储过程慢 SQL.....	957
7.24.2.1.1 特性概述.....	957
7.24.2.1.2 技术原理.....	958
7.24.2.1.3 操作指导.....	959
7.24.3 常见问题.....	961
7.25 数据库连接驱动指南.....	962
7.25.1 JDBC.....	962
7.25.1.1 概述.....	962
7.25.1.2 技术原理.....	962
7.25.1.3 操作指导.....	965
7.25.1.3.1 开发流程.....	965
7.25.1.3.2 获取驱动 jar 包并配置 JDK 环境.....	965
7.25.1.3.3 连接数据库.....	968

7.25.1.3.4 执行 SQL 语句.....	969
7.25.1.3.5 处理结果集.....	971
7.25.1.3.6 关闭数据库连接.....	972
7.25.1.3.7 最佳实践.....	972
7.25.1.4 常见问题.....	972
7.25.1.4.1 batchMode 设置错误.....	972
7.25.1.4.2 Hibernate 框架插入数据开启校验时报错.....	973
7.25.1.4.3 使用 SSL 方式建连报错或阻塞.....	974
7.25.2 ODBC.....	975
7.25.2.1 概述.....	975
7.25.2.2 技术原理.....	975
7.25.2.3 操作指导.....	979
7.25.2.3.1 开发流程.....	979
7.25.2.3.2 获取源码包、ODBC 包以及依赖库.....	980
7.25.2.3.3 连接数据库.....	981
7.25.2.3.4 执行 SQL 语句.....	995
7.25.2.3.5 处理结果集.....	995
7.25.2.3.6 关闭连接.....	995
7.25.2.3.7 最佳实践.....	996
7.25.2.4 常见问题.....	996
7.26 基于 hashbucket 表的在线扩容.....	998
7.26.1 特性概述.....	998
7.26.2 技术原理.....	999
7.26.3 操作指导.....	1002
7.26.4 常见问题.....	1011
7.27 备机读.....	1012
7.27.1 特性概述.....	1012
7.27.2 技术原理.....	1013
7.27.2.1 三种回放模式及备机读.....	1013
7.27.2.2 分布式 GTM-FREE 备机读.....	1017
7.27.2.3 分布式 GTM-LITE 备机读.....	1019
7.27.2.4 分布式 GTM-LITE 容灾读.....	1024
7.27.3 操作指导.....	1027
7.27.3.1 环境规格与回放类型选择.....	1028
7.27.3.2 操作步骤.....	1031
7.27.3.3 注意事项.....	1033
7.27.4 常见问题.....	1033
7.28 抗过载.....	1039
7.28.1 SQL 限流.....	1039
7.28.1.1 特性概述.....	1039
7.28.1.2 技术原理.....	1041
7.28.1.3 操作指导.....	1041

7.28.1.4 常见问题.....	1043
7.28.2 内存抗过载.....	1043
7.28.2.1 特性概述.....	1043
7.28.2.2 技术原理.....	1044
7.28.2.3 操作指导.....	1045
7.28.2.4 常见问题.....	1045
7.28.3 线程池抗过载.....	1046
7.28.3.1 特性概述.....	1046
7.28.3.2 技术原理.....	1047
7.28.3.3 操作指导.....	1048
7.28.3.4 常见问题.....	1048
7.28.4 建连线程池.....	1049
7.28.4.1 特性概述.....	1049
7.28.4.2 技术原理.....	1049
7.28.4.3 操作指导.....	1050
7.28.5 并发场景支持抗过载逃生能力.....	1051
7.28.5.1 特性概述.....	1051
7.28.5.2 技术原理.....	1051
7.28.5.3 操作指导.....	1052
7.28.5.4 常见问题.....	1053
7.29 Unix Domain Socket 运维通道.....	1053
7.29.1 特性概述.....	1053
7.29.2 技术原理.....	1054
7.29.3 操作指导.....	1066
7.29.4 常见问题.....	1067
7.30 CM 仲裁及故障检测原理.....	1067
7.30.1 特性概述.....	1068
7.30.2 技术原理.....	1068
7.30.2.1 仲裁原理.....	1068
7.30.2.1.1 DN 仲裁.....	1068
7.30.2.1.2 CN 仲裁.....	1069
7.30.2.1.3 GTM 仲裁.....	1070
7.30.2.1.4 CMServer 仲裁.....	1071
7.30.2.1.5 ETCD 仲裁.....	1072
7.30.2.2 故障检测原理.....	1072
7.30.2.2.1 磁盘检测.....	1072
7.30.2.2.2 DN 僵死检测.....	1073
7.30.2.2.3 急救室能力.....	1073
7.30.2.2.4 硬件故障处理.....	1074
7.30.3 操作指导.....	1074
7.30.4 常见问题.....	1074
7.31 集群内与集群间倒换原理.....	1074

7.31.1 集群内倒换.....	1074
7.31.1.1 特性概述.....	1075
7.31.1.2 技术原理.....	1075
7.31.1.3 操作指导.....	1076
7.31.1.4 常见问题.....	1077
7.31.2 集群间倒换.....	1077
7.31.2.1 特性概述.....	1077
7.31.2.2 技术原理.....	1080
7.31.2.3 操作指导.....	1081
7.31.2.3.1 基于流式复制的异地容灾演练实践参考.....	1081
7.31.2.3.2 同城双中心高可用方案容灾演练实践参考.....	1085
7.31.2.4 常见问题.....	1088
7.31.2.4.1 两地三中心跨 Region 容灾计划内倒换 switchover 流程异常.....	1089
7.31.2.4.2 同城双中心计划内倒换 switchover 流程异常.....	1093
7.32 主备重建.....	1097
7.32.1 特性概述.....	1097
7.32.2 技术原理.....	1101
7.32.3 操作指导.....	1108
7.32.4 常见问题.....	1110
7.33 升级.....	1111
7.33.1 就地升级（非透明多写）.....	1111
7.33.1.1 特性概述.....	1111
7.33.1.2 技术原理.....	1115
7.33.1.3 操作指导.....	1115
7.33.1.4 常见问题.....	1118
7.33.2 灰度升级（非透明多写）.....	1118
7.33.2.1 特性概述.....	1118
7.33.2.2 技术原理.....	1123
7.33.2.3 操作指导.....	1123
7.33.2.4 常见问题.....	1127
7.33.3 滚动升级.....	1127
7.33.3.1 特性概述.....	1127
7.33.3.2 技术原理.....	1129
7.33.3.3 操作指导.....	1132
7.33.3.4 常见问题.....	1136
7.34 节点修复、替换.....	1136
7.34.1 特性概述.....	1136
7.34.2 技术原理.....	1137
7.34.3 操作指导.....	1138
7.34.4 常见问题.....	1143
7.35 计划缓存.....	1143
7.35.1 预编译计划缓存.....	1143

7.35.1.1 特性概述.....	1143
7.35.1.2 技术原理.....	1144
7.35.1.3 操作指导.....	1145
7.35.1.4 常见问题.....	1148
7.35.2 自动参数化和相同语句计划复用.....	1148
7.35.2.1 特性概述.....	1148
7.35.2.2 技术原理.....	1150
7.35.2.3 操作指导.....	1152
7.35.2.4 常见问题.....	1154
7.35.3 全局共享计划缓存.....	1156
7.35.3.1 特性概述.....	1156
7.35.3.2 技术原理.....	1157
7.35.3.3 操作指导.....	1160
7.35.3.4 常见问题.....	1162
7.35.4 计划缓存内存上限管控.....	1162
7.35.4.1 特性概述.....	1162
7.35.4.2 技术原理.....	1163
7.35.4.3 操作指导.....	1165
7.35.4.3.1 会话级计划缓存管控.....	1165
7.35.4.3.2 全局共享级计划缓存管控.....	1167
7.35.4.4 常见问题.....	1168
7.36 DATABASE LINK.....	1169
7.36.1 特性概述.....	1169
7.36.2 技术原理.....	1170
7.36.3 操作指导.....	1173
7.36.4 常见问题.....	1185
7.37 单节点表.....	1185
7.37.1 特性概述.....	1185
7.37.2 技术原理.....	1185
7.37.3 操作指导.....	1187
7.37.3.1 单节点表 DDL 操作.....	1187
7.37.3.2 单节点表在线转换操作.....	1190
7.37.3.3 单节点表 DML 操作.....	1192
7.37.4 常见问题.....	1197
7.38 ROWID 表.....	1197
7.38.1 特性概述.....	1197
7.38.2 使用指导.....	1198
7.38.2.1 DDL.....	1198
7.38.2.2 ROWID 类型.....	1199
7.38.2.3 DML.....	1200
8 基础概念.....	1238
8.1 数据库对象.....	1238

8.2 模式（Schema）	1242
8.3 约束（Constraint）	1244
8.4 序列（Sequence）	1247
8.5 系统列.....	1248
8.6 分区表.....	1248
8.7 存储引擎.....	1250
8.8 索引.....	1259
8.9 物化视图.....	1262
8.10 事务.....	1264
8.11 锁.....	1266
8.12 PL/SQL.....	1269
8.12.1 PL/SQL 简介.....	1269
8.12.2 游标（Cursor）	1269
8.12.3 存储过程（Procedure）	1272
8.12.4 函数（Function）	1274
8.12.5 触发器（Trigger）	1274
8.12.6 包（Package）	1276
8.13 闪回恢复.....	1278
8.14 VACUUM.....	1284
8.15 TOAST 表.....	1291
8.16 HTAP.....	1292
8.17 全密态数据库.....	1295
8.18 DDL.....	1299
8.19 预写式日志.....	1301
8.20 极致 RTO.....	1304
8.21 DCF.....	1306
8.22 CM.....	1310
8.23 调用堆栈.....	1312
8.24 等待事件.....	1315
8.25 备份恢复.....	1318
9 FAQ-非透明多写.....	1322
9.1 数据库系统概述.....	1322
9.1.1 GaussDB 支持哪些硬件平台？	1322
9.1.2 如何查看当前数据库的隔离级别？	1322
9.1.3 GaussDB 易用性如何？	1322
9.1.4 如何在 GaussDB 当前事务中建立保存点及删除保存点？	1323
9.1.5 事务的作用是什么？	1323
9.1.6 开启事务的方式有哪些？	1323
9.1.7 数据库事务正确执行的四个特性是什么？	1324
9.1.8 什么是脏读、幻读、不可重复读？	1324
9.1.9 什么是 MVCC？	1324
9.2 数据库连接管理.....	1325

9.2.1 如何查询数据库活跃连接数？	1325
9.2.2 通过客户端无法连接数据库的原因有哪些？	1325
9.2.3 GaussDB 支持哪些客户端接入认证方式？	1325
9.2.4 如何查看数据库应用连接数以及最大连接数？	1325
9.2.5 如何获取当前连接数据库的用户名？	1326
9.2.6 如何查看指定用户的会话连接数上限？	1326
9.3 数据库安全.....	1326
9.3.1 如何手动锁定和解锁用户？	1326
9.3.2 GaussDB 支持哪些安全管理？	1326
9.3.3 创建模型服务前，如何生成 OBS 传输密钥？	1327
9.3.4 动态脱敏策略如何生效？	1327
9.3.5 如何设置账号的有效期？	1327
9.3.6 GaussDB 如何开启审计策略，支持哪些内容的审计？	1327
9.3.7 如何查看某个用户有哪些表的权限？	1328
9.3.8 GaussDB 如何开启三权分立？	1328
9.3.9 GaussDB 如何创建只读用户？	1329
9.3.10 GaussDB 会话有几种状态？如何查询用户的会话状态？	1329
9.3.11 如何查询 SSL 证书的失效日期？	1330
9.3.12 如何配置开启密码复杂度校验？	1330
9.3.13 如何检查数据库初始化用户角色？	1330
9.3.14 哪些系统视图与审计信息有关？	1331
9.3.15 如何配置日志文件权限？	1331
9.3.16 如何配置数据库账号登录失败尝试次数？	1331
9.3.17 透明加密和全密态有什么区别？	1331
9.3.18 如何保证数据的安全性？	1331
9.3.19 GaussDB 如何创建弱口令？	1332
9.3.20 GaussDB 中模式、用户和角色如何理解？	1332
9.3.21 为什么 REVOKE 某个用户的 CONNECT ON DATABASE 权限后还可以登录数据库？	1332
9.3.22 如何查看数据库用户的创建时间？	1333
9.4 数据库使用入门.....	1333
9.4.1 GaussDB 的 DROP、TRUNCATE、DELETE 这三种删除数据的方式主要区别是什么？	1333
9.4.2 GaussDB 数据库一个汉字占几个字节？	1333
9.4.3 如何并行创建索引？	1333
9.4.4 如何查看 GaussDB KernelGaussDB 客户端与服务端字符编码类型？	1334
9.4.5 如何查看当前会话 ID 以及登录时间？.....	1334
9.4.6 如何查看某张表的创建时间？	1334
9.4.7 如何查看数据库参数值？	1334
9.4.8 如何查看外表信息？	1335
9.4.9 如何查看系统中正在运行的 SELECT 语句？	1335
9.4.10 如何查询表存在的索引？	1335
9.4.11 如何查询表的存储方式？	1336
9.4.12 如何查询表 and 数据库容量？	1336

9.4.13 如何查询会话数？	1336
9.4.14 如何查询数据库、表空间、表的大小？	1337
9.4.15 如何查询数据库运行时间？	1337
9.4.16 如何查询数据库最近一次重启时间？	1337
9.4.17 如何创建默认为当前时间的列？	1337
9.4.18 如何创建只读视图以及判断该视图是否是只读视图？	1337
9.4.19 如何关闭系统表结构修改配置？	1338
9.4.20 如何判断字段是否为 NULL？	1338
9.4.21 如何删除重复的行？	1338
9.4.22 如何使用 SQL 语句查看数据库的模式？	1339
9.4.23 如何通过 SQL 语言查询表、索引、视图的定义？	1339
9.4.24 数据分区都有哪些优点？	1339
9.4.25 唯一约束和唯一索引有什么区别？	1340
9.4.26 为什么查询中无法识别表名或者列名？	1340
9.4.27 以 pg_temp 或 pg_toast_temp 开头的 SCHEMA 是什么？	1341
9.4.28 在使用了字符串处理函数后查询就不使用索引，该如何处理？	1341
9.5 开发设计	1342
9.5.1 GaussDB 单表支持的最大列数？	1342
9.5.2 SQL 语言分为哪几类？	1342
9.5.3 触发器的作用是什么？	1342
9.5.4 函数和存储过程的区别是什么？	1342
9.5.5 如何查询当前数据库中有哪些触发器，以及触发器的定义？	1343
9.5.6 如何查询分区表分区信息和索引信息？	1344
9.5.7 如何查询一个表涉及到的所有触发器？	1344
9.5.8 数据库对象名称的长度是否有限制？	1344
9.5.9 选择数据库字符集时，应考虑哪些因素？	1344
9.6 其他	1344
9.6.1 如何检查磁盘类型？	1345
9.6.2 如何查询活跃元组、脏元组、脏元组比例？	1345
9.6.3 VACUUM 与 VACUUM FULL 的主要区别？	1345
9.6.4 GaussDB KernelGaussDB 中 pg_terminate_backend 和 pg_terminate_session 有什么区别？	1345
9.6.5 如何显示数据库文件信息？	1346
9.6.6 GaussDB KernelGaussDB Xlog 日志自动回收的频率是多久？	1347
9.6.7 什么是死元组？	1347
9.6.8 如何查询表的死元组？	1347
9.6.9 什么是 OID？	1347
9.6.10 什么是 UDF？	1347
9.6.11 存储过程嵌套调用的深度是多少？	1347
9.6.12 TOAST 是什么？	1347
9.6.13 沙箱的作用是什么？	1348
9.6.14 什么是向量数据库？	1348
9.6.15 什么是 GIN 索引？	1348

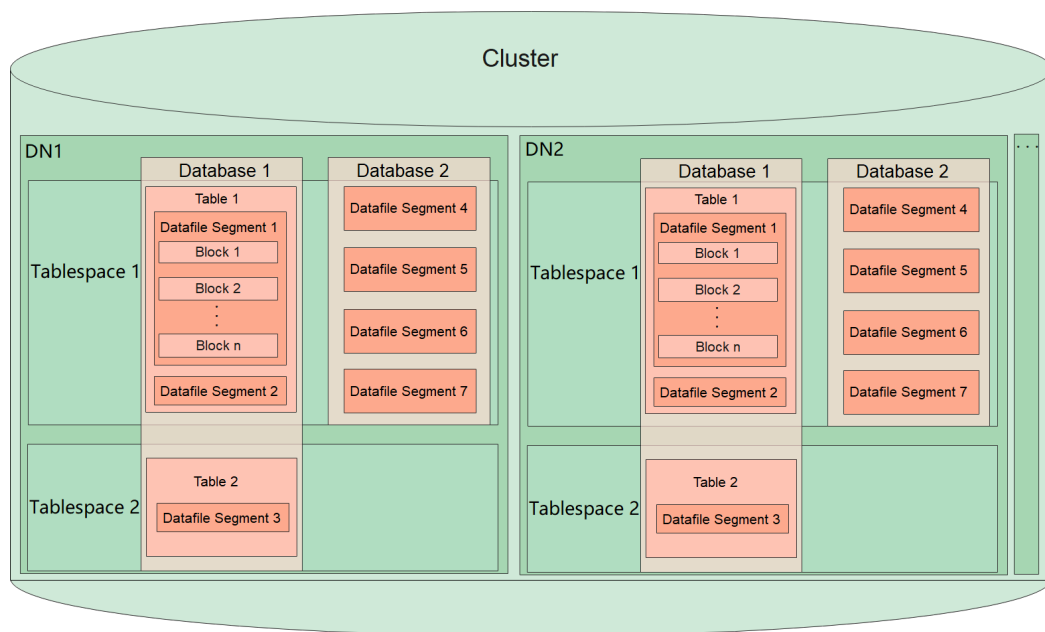
9.6.16 存储过程可以调用触发器吗？	1348
9.6.17 如何查询表中的随机几行？	1348
9.6.18 为什么 SEQUENCE 的编号存在间隙？	1349
9.6.19 如何解决 power()函数计算结果数值太大时的误差？	1349
9.6.20 什么是 CTID？	1350
9.6.21 如何查询表中某列第 N 大的数据？	1350
9.6.22 如何查询字段中包含非数字的数据？.....	1350
9.6.23 如何修改列的顺序？	1350
9.6.24 count(*)和 count(column)的区别是什么？	1351
9.6.25 GaussDB 都有哪些通配符，如何使用？	1351
9.6.26 LIMIT 2、LIMIT 2,3、LIMIT 2 OFFSET 3 的区别？.....	1352
9.6.27 空字符串和 NULL 的关系？	1352
9.6.28 日期/时间类型哪些带时区，哪些不带时区？.....	1352
9.6.29 如果建表时没有指定分布列，数据会怎么存储？	1352
9.6.30 什么是数据倾斜，如何检查？	1353
9.6.31 使用 LIKE 进行模糊查找时如何忽略大小写？	1354

1 数据库系统概述

1.1 数据库逻辑结构图

集群的每个DN（Data Node，数据节点）负责存储数据，其存储介质是磁盘，本节主要从逻辑视角介绍每个DN上有哪些对象，以及这些对象之间的关系。另外介绍数据在不同节点的分布方式。数据库逻辑结构如图1-1所示。

图 1-1 数据库逻辑结构图



说明

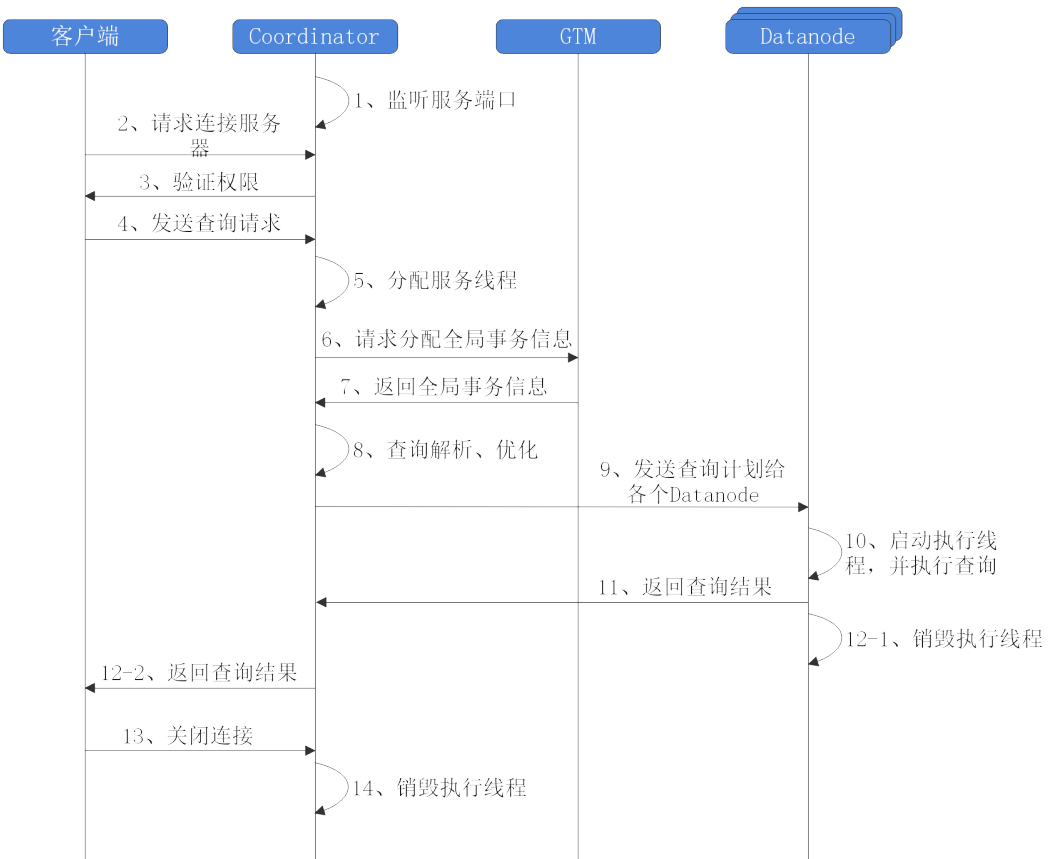
- Tablespace，即表空间，表空间是一个目录，集群中可以存在多个表空间，其中存储的是它所包含的数据库的各种物理文件。每个表空间可以对应多个Database。
- Database，即数据库，用于管理各类数据对象，各数据库间相互隔离。数据库管理的对象可分布在多个Tablespace上。
- Datafile Segment，即数据文件，通常每张表只对应一个数据文件。如果某张表的数据大于1GB，则会分为多个数据文件存储。
- Table，即表，每张表只能属于一个数据库，也只能对应到一个Tablespace。每张表对应的数据文件必须在同一个Tablespace中。
- Block，即数据块，是数据库管理的基本单位，默认大小为8KB。
- 数据在不同的DN上有四种分布方式，可以在建表的时候指定：REPLICATION、HASH、RANGE、LIST。

1.2 数据查询请求处理过程

GaussDB服务响应流程涉及到客户端、Coordinator、GTM、datanode等组件。

具体流程请参见图1-2。

图 1-2 GaussDB 服务响应流程



1.3 管理事务

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位。GaussDB数据库支持的事务控制命令有启动、设置、提交、回滚。GaussDB数据库支持的事务隔离级别有READ COMMITTED、REPEATABLE READ和SERIALIZABLE，SERIALIZABLE等价于REPEATABLE READ。

事务控制

以下是数据库支持的事务命令：

- **启动事务**
用户可以使用START TRANSACTION或者BEGIN语法启动事务。
- **设置事务**
用户可以使用SET TRANSACTION或者SET LOCAL TRANSACTION语法设置事务特性。
- **提交事务**
用户可以使用COMMIT或者END完成提交事务的功能，即提交事务的所有操作。
- **回滚事务**
回滚是在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销。

非透明多写特性下，事务相关详细操作请参见《参考》中“SQL参考 > SQL语法”下的“START TRANSACTION”、“BEGIN”、“SET TRANSACTION”、“COMMIT | END”和“ROLLBACK”章节。

事务隔离级别

事务的隔离级别控制多个事务在访问同一数据资源时对不同隔离级别数据的处理行为。

说明

在事务中第一个数据操作语句（SELECT、INSERT、DELETE、UPDATE、FETCH或COPY）执行之后，事务隔离级别不能再次设置。

- **READ COMMITTED**：读已提交隔离级别，事务只能读到已提交的数据而不会读到未提交的数据，这是缺省值。

实际上，SELECT查询会查看到在查询开始运行的瞬间该数据库的一个快照。不过，SELECT能查看到其自身所在事务中先前修改的执行结果。即使先前修改尚未提交。请注意，在同一个事务里两个相邻的SELECT命令可能会查看到不同的快照，因为其它事务会在第一个SELECT执行期间提交。

因为在读已提交模式里，每个新的命令都是从一个新的快照开始的，而这个快照包含所有到该时刻为止已提交的事务，因此同一事务中后面的命令将看到任何已提交的其它事务的效果。这里关注的问题是在单个命令里是否看到数据库里完全一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快，使用简单。不过，对于做复杂查询和更新的应用，可能需要保证数据库有比读已提交模式更加严格的一致性视图。

- **REPEATABLE READ**: 事务可重复读隔离级别，事务只能读到事务开始之前已提交的数据，不能读到未提交的数据以及事务执行期间其它并发事务提交的修改（但是，查询能查看到自身所在事务中先前修改的执行结果，即使先前修改尚未提交）。这个级别和读已提交是不一样的，因为可重复读事务中的查询看到的是事务开始时的快照，不是该事务内部当前查询开始时的快照，就是说，单个事务内部的SELECT命令总是查看到同样的数据，查看不到自身事务开始之后其他并发事务修改后提交的数据。使用该级别的应用必须准备好重试事务，因为可能会发生串行化失败。

REPEATABLE READ基于多版本快照实现，可能出现写偏斜的场景，如果需要避免该场景出现，请先对事务涉及的行进行SELECT FOR UPDATE操作。写偏斜的场景示例如下：

场景一：表a拥有id、value两个字段，类型均为int，插入两条数据，假定表a业务逻辑上需要满足两条数据value和小于等于10。并发开启两个事务，基于读取的值进行更新并修改相应值，修改后在事务内均满足value和小于等于10，提交后，最终value和等于12，违反a的业务逻辑假定。

```
gaussdb=# CREATE TABLE a(id int, value int);
CREATE TABLE
gaussdb=# INSERT INTO a VALUES(1,4);
INSERT 0 1
gaussdb=# INSERT INTO a VALUES(2,4);
INSERT 0 1
session1 :
gaussdb=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 4
2 | 4
(2 rows)
gaussdb=# UPDATE a SET value = 6 WHERE id = 1;
UPDATE 1
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 6
2 | 4
(2 rows)
session2:
gaussdb=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 4
2 | 4
(2 rows)
gaussdb=# UPDATE a SET value = 6 WHERE id = 2;
UPDATE 1
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 4
2 | 6
(2 rows)
session1:
gaussdb=# COMMIT;
COMMIT
session2:
gaussdb=# COMMIT;
COMMIT
gaussdb=# SELECT * FROM a;
id | value
----+-----
```

```
1 | 6
2 | 6
(2 rows)
```

场景二：表a拥有两个字段id、value，类型均为int，其中id为主键，并发主键删除插入，事务内可能读到两条主键的值，违反主键约束。

```
gaussdb=# CREATE TABLE a(id int PRIMARY KEY, value int);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "a_pkey" for table "a"
CREATE TABLE
gaussdb=# INSERT INTO a VALUES(1,10);
INSERT 0 1
session1:
gaussdb=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
gaussdb=# DELETE a WHERE id = 1;
DELETE 1
session2:
gaussdb=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 10
(1 row)
session1:
gaussdb=# COMMIT;
COMMIT
session2:
gaussdb=# INSERT INTO a VALUES(1, 100);
INSERT 0 1
gaussdb=# SELECT * FROM a;
id | value
----+-----
1 | 10
1 | 100
(2 rows)
```

- **SERIALIZABLE**：GaussDB目前功能上不支持此隔离级别，设置该隔离级别时，等价于REPEATABLE READ。

2 数据库安全

2.1 用户及权限

2.1.1 默认权限机制

数据库对象创建后，进行对象创建的用户就是该对象的所有者。集群安装后默认情况下，数据库系统管理员具有与对象所有者相同的权限。也就是说对象创建后，默认只有对象所有者或者系统管理员可以查询、修改和销毁对象，以及通过GRANT将对象的权限授予其他用户，请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。

为使其他用户能够使用对象，必须向用户或包含该用户的角色授予必要的权限。

GaussDB支持以下的权限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、CREATE、CONNECT、EXECUTE、USAGE、ALTER、DROP、COMMENT、INDEX和VACUUM。不同的权限与不同的对象类型关联。有关各权限的详细信息，请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。

要撤销已经授予的权限，请参见《参考》中“SQL参考 > SQL语法 > R > REVOKE”章节。对象所有者的权限（例如ALTER、DROP、COMMENT、INDEX、VACUUM、GRANT和REVOKE）是隐式拥有的，即只要拥有对象就可以执行对象所有者的隐式权限。对象所有者可以撤销自己的普通权限（SELECT、INSERT、UPDATE、DELETE），例如，设置表对自己以及其他用户只读，系统管理员用户除外。

系统表和系统视图要么只对系统管理员可见，要么对所有用户可见。标识了需要系统管理员权限的系统表和视图只有系统管理员可以查询。有关信息，请参见《参考》中“系统表和系统视图”章节。

数据库提供对象隔离的特性，对象隔离特性开启时，用户只能查看有权限访问的对象（表、视图、字段、函数），系统管理员不受影响。有关信息，请参见《参考》中“SQL参考 > SQL语法 > A > ALTER DATABASE”章节。

不建议用户修改系统表和系统视图的权限。

2.1.2 管理员

初始用户

集群安装过程中自动生成的账户称为初始用户。初始用户也是系统管理员、安全管理员、审计管理员、监控管理员、运维管理员和安全策略管理员，拥有系统的最高权

限，能够执行所有的操作。如果安装时不设置初始用户名称，则该账户与进行集群安装的操作系统用户同名。如果在安装集群时不设置初始用户的密码，安装完成后密码为空，在执行其他操作前需要通过gsq客户端修改初始用户的密码。如果初始用户密码为空，则除修改密码外，无法执行其他SQL操作以及升级、扩容、节点替换等操作。

说明

- 初始用户的OID为10，可以通过gs_roles视图查询。
- 初始用户会绕过所有权限检查。建议仅将初始用户作为DBA管理用途，而非业务应用。

系统管理员

系统管理员是指具有SYSADMIN属性的账户，默认安装情况下具有与对象所有者相同的权限，但不包括dbe_perf模式的对象权限。

要创建新的系统管理员，请以初始用户或者系统管理员用户身份连接数据库，并使用带SYSADMIN选项的语句进行设置。详情请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER sysadmin WITH SYSADMIN password '*****';
```

或者

```
gaussdb=# ALTER USER joe SYSADMIN;
```

ALTER USER时，要求用户已存在。

安全管理员

安全管理员是指具有CREATEROLE属性的账户，具有创建、修改、删除用户或角色的权限，和授予或者撤销任何非系统管理员、内置角色、永久用户、运维管理员的权限。

要创建新的安全管理员，三权分立关闭时，请以系统管理员或者安全管理员身份连接数据库。三权分立打开时，请以安全管理员身份连接数据库，并使用带CREATEROLE选项的语句进行设置。详情请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER createrole WITH CREATEROLE password '*****';
```

或者

```
gaussdb=# ALTER USER joe CREATEROLE;
```

ALTER USER时，要求用户已存在。

审计管理员

审计管理员是指具有AUDITADMIN属性的账户，具有查看和删除审计日志的权限。

要创建新的审计管理员，三权分立关闭时，请以系统管理员或者安全管理员身份连接数据库。三权分立打开时，只能以初始用户身份连接数据库，并使用带AUDITADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER auditadmin WITH AUDITADMIN password '*****';
```

或者

```
gaussdb=# ALTER USER joe AUDITADMIN;
```

ALTER USER时，要求用户已存在。

监控管理员

监控管理员是指具有MONADMIN属性的账户，具有查看dbe_perf模式下视图和函数的权限，亦可以对dbe_perf模式的对象权限进行授予或收回。

要创建新的监控管理员，请以系统管理员身份连接数据库，并使用带MONADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER monadmin WITH MONADMIN password '*****';
```

或者

```
gaussdb=# ALTER USER joe MONADMIN;
```

ALTER USER时，要求用户已存在。

运维管理员

运维管理员是指具有OPRADMIN属性的账户，具有使用Roach工具执行备份恢复的权限。

要创建新的运维管理员，请以初始用户身份连接数据库，并使用带OPRADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER opradmin WITH OPRADMIN password '*****';
```

或者

```
gaussdb=# ALTER USER joe OPRADMIN;
```

ALTER USER时，要求用户已存在。

安全策略管理员

安全策略管理员是指具有POLADMIN属性的账户，具有创建资源标签、脱敏策略和统一审计策略的权限。

要创建新的安全策略管理员，请以系统管理员用户身份连接数据库，并使用带POLADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
gaussdb=# CREATE USER poladmin WITH POLADMIN password '*****';
```

或者

```
gaussdb=# ALTER USER joe POLADMIN;
```

ALTER USER时，要求用户已存在。

2.1.3 用户

使用CREATE USER和ALTER USER可以创建和管理数据库用户。非透明多写特性下数据库集群包含一个或多个数据库。用户和角色在整个集群范围内是共享的，但是其数据

并不共享。即用户可以连接任何数据库，但当连接成功后，任何用户都只能访问连接请求里声明的数据库。

在用户登录GaussDB时会对其进行身份验证。用户可以拥有数据库和数据库对象（例如表），并且可以向用户和角色授予对这些对象的权限以控制谁可以访问哪一个对象。除系统管理员外，具有CREATEDB属性的用户可以创建数据库并授予对这些数据库的权限。

创建、修改和删除用户

- 使用CREATE USER语句创建用户，请参见《参考》中“SQL参考> SQL语法 > C> CREATE USER”章节。

例如：创建用户joe，并设置用户拥有CREATEDB属性。

```
gaussdb=# CREATE USER joe WITH CREATEDB PASSWORD '*****';  
CREATE ROLE
```

- 创建系统管理员，请使用带有SYSADMIN选项语句。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”章节。

```
gaussdb=# CREATE USER tom WITH SYSADMIN PASSWORD '*****';  
CREATE ROLE
```

- 删除现有用户，请参见《参考》中“SQL参考 > SQL语法 > D > DROP USER”章节。

```
gaussdb=# DROP USER tom;  
DROP ROLE
```

- 更改用户账户（例如，重命名用户或更改密码），请参见《参考》中“SQL参考 > SQL语法 > A > ALTER USER”章节。

```
gaussdb=# ALTER USER jim IDENTIFIED BY '*****' REPLACE '*****';  
ALTER ROLE
```

- 查看用户列表，请查询gs_user视图。请参见《参考》中“系统表和系统视图 > 系统视图 > 用户和权限管理 > GS_USER”章节。

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

- 查看用户属性，请查询系统表请查询pg_authid系统表。请参见《参考》中“系统表和系统视图 > 系统表 > 连接和认证 > PG_AUTHID”章节。

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

2.1.4 角色

通过GRANT将角色授予用户后，用户即具有了角色的所有权限。推荐使用角色进行高效权限分配。例如，可以为设计、开发和维护人员创建不同的角色，将角色GRANT给用户后，再向每个角色中的用户授予其工作所需数据的差异权限。在角色级别授予或撤销权限时，这些更改将作用到角色下的所有成员。

GaussDB提供了一个隐式定义的拥有所有角色的组PUBLIC，所有创建的用户和角色默认拥有PUBLIC拥有的权限。关于PUBLIC默认拥有的权限请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。要撤销或重新授予用户和角色对PUBLIC的权限，可通过在GRANT和REVOKE指定关键字PUBLIC实现。

要查看所有角色，请查询系统视图GS_ROLES，请参见《参考》中“系统表和系统视图 > 系统视图 > 用户和权限管理 > GS_ROLES”章节。

```
SELECT * FROM GS_ROLES;
```

创建、修改和删除角色

非三权分立时，只有系统管理员和具有CREATEROLE属性的用户才能创建、修改或删除角色。

- 创建角色，请参见《参考》中“SQL参考 > SQL语法 > C > CREATE ROLE”章节。
- 在现有角色中添加或删除用户，请参见《参考》中“SQL参考 > SQL语法 > A > ALTER ROLE”章节。
- 删除角色，请参见《参考》中“SQL参考 > SQL语法 > D > DROP ROLE”章节。DROP ROLE只会删除角色，并不会删除角色中的成员用户账户。

内置角色

GaussDB提供了一组默认角色，以gs_role_开头命名。它们提供对特定的、通常需要高权限的操作的访问，可以将这些角色GRANT给数据库内的其他用户或角色，让这些用户能够使用特定的功能。在授予这些角色时应当谨慎，以确保它们被用在需要的地方。内置角色允许的权限范围如表2-1所示。

表 2-1 内置角色权限描述

角色	权限描述
gs_role_signal_backend	具有调用函数pg_cancel_backend、pg_terminate_backend和pg_terminate_session来取消或终止其他会话的权限，或调用函数pg_terminate_active_session_socket来关闭活跃会话和客户端的socket连接，但不能操作属于初始用户和PERSISTENCE用户的会话。
gs_role_tablespace	具有创建表空间（tablespace）的权限。
gs_role_replication	具有调用逻辑复制相关函数的权限。
gs_role_account_lock	具有加解锁用户的权限，但不能加解锁初始用户和PERSISTENCE用户。
gs_role_pldebugger	具有执行dbe_pldebugger下调试函数的权限。
gs_role_public_dblink_drop	具有执行删除public database link对象的权限。
gs_role_public_dblink_alter	具有执行修改public database link对象的权限。
gs_role_seclabel	具有创建、删除和应用安全标签的权限。
gs_role_public_synonym_create	具有创建public同义词的权限。
gs_role_public_synonym_drop	具有删除public同义词的权限。

关于内置角色的管理有如下约束：

- 以gs_role_开头的角色名作为数据库的内置角色保留名，禁止新建以“gs_role_”开头的用户/角色/模式，也禁止将已有的用户/角色/模式重命名为以“gs_role_”开头。
- 禁止对内置角色进行ALTER和DROP操作。
- 内置角色默认没有LOGIN权限，不设预置密码。
- gsql元命令\du和\dg不显示内置角色的相关信息，但若显示指定了pattern为特定内置角色则会显示。
- 三权分立关闭时，初始用户、具有SYSADMIN权限的用户和具有内置角色ADMIN OPTION权限的用户有权对内置角色执行GRANT/REVOKE管理。三权分立打开时，初始用户和具有内置角色ADMIN OPTION权限的用户有权对内置角色执行GRANT/REVOKE管理。例如：

```
GRANT gs_role_signal_backend TO user1;  
REVOKE gs_role_signal_backend FROM user1;
```

2.1.5 Schema

Schema又称作模式。通过管理Schema，允许多个用户使用同一数据库而不相互干扰，可以将数据库对象组织成易于管理的逻辑组，同时便于将第三方应用添加到相应的Schema下而不引起冲突。

每个数据库包含一个或多个Schema。数据库中的每个Schema包含表和其他类型的对象。数据库创建初始，默认具有一个名为public的公共Schema，所有用户都拥有此Schema的usage权限。此外，每个数据库都包含一个pg_catalog Schema，它包含系统表 and 所有内置数据类型、函数和操作符。只有系统管理员和初始化用户可以在public和pg_catalog Schema下创建函数、存储过程和同义词对象，其他用户即使授予public和pg_catalog模式的create权限后也不可以创建上述三种对象。可以通过Schema分组数据库对象。Schema类似于操作系统目录，但Schema不能嵌套。默认只有初始化用户可以在pg_catalog模式下创建对象。

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

通过CREATE USER创建用户的同时，系统会在执行该命令的数据库中，为该用户创建一个同名的Schema。

数据库对象创建在数据库搜索路径中的第一个Schema内。默认情况下的第一个Schema及如何变更Schema顺序等更多信息，请参见[搜索路径](#)。

创建、修改和删除 Schema

- 创建Schema，请参见《参考》中“SQL参考 > SQL语法 > C > CREATE SCHEMA”章节。默认初始用户和系统管理员可以创建Schema，其他用户需要具备数据库的CREATE权限才可以在该数据库中创建Schema，授权方式请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”中将数据库的访问权限授予指定的用户或角色中的语法。
- 更改Schema名称或者所有者，请参见《参考》中“SQL参考 > SQL语法 > A > ALTER SCHEMA”章节。Schema所有者可以更改Schema。
- 删除Schema及其对象，请参见《参考》中“SQL参考 > SQL语法 > D > DROP SCHEMA”章节。Schema所有者可以删除Schema。
- 在Schema内创建表，请以schema_name.table_name格式创建表。不指定schema_name时，对象默认创建到[搜索路径](#)中的第一个Schema内。

- 查看Schema所有者，请对系统表PG_NAMESPACE和系统视图GS_USER执行如下关联查询。详情请参见《参考》中“系统表和系统视图 > 系统表 > 其他系统表 > PG_NAMESPACE”和《参考》中“系统表和系统视图 > 系统视图 > 用户和权限管理 > GS_USER”章节。语句中的schema_name请替换为实际要查找的Schema名称。

```
gaussdb=# SELECT s.nspname,u.username AS nspowner FROM pg_namespace s, gs_user u WHERE  
nspname='schema_name' AND s.nspowner = u.usesysid;
```

- 查看所有Schema的列表，请查询PG_NAMESPACE系统表。
- 查看属于某Schema下的表列表，请查询系统视图PG_TABLES。详细请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG_TABLES”章节。例如，以下查询会返回Schema pg_catalog中的表列表。

```
gaussdb=# SELECT distinct(tablename),schemaname FROM pg_tables where schemaname =  
'pg_catalog';
```

搜索路径

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

- 要查看当前搜索路径，请参见《参考》中“SQL参考 > SQL语法 > S > SHOW”章节。

```
gaussdb=# SHOW SEARCH_PATH;  
search_path  
-----  
"$user",public  
(1 row)
```

search_path参数的默认值为：“\$user”,public。\$user表示与当前会话用户名同名的Schema名，如果这样的模式不存在，\$user将被忽略。所以默认情况下，用户连接数据库后，如果数据库下存在同名Schema，则对象会添加到同名Schema下，否则对象被添加到Public Schema下。

- 更改当前会话的默认Schema，请使用SET命令。

执行如下命令将搜索路径设置为myschema, public，首先搜索myschema，然后搜索public。

```
gaussdb=# SET SEARCH_PATH TO myschema, public;  
SET
```

2.1.6 用户权限设置

- 给用户直接授予某对象的权限，请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。

将Schema中的表或者视图对象授权给其他用户或角色时，需要将表或视图所属Schema的USAGE权限同时授予该用户或角色。否则用户或角色将只能看到这些对象的名称，并不能实际进行对象访问。

例如，下面示例将Schema tpcds的权限授予用户joe后，将表tpcds.web_returns的select权限授予用户joe。

```
gaussdb=# GRANT USAGE ON SCHEMA tpcds TO joe;  
gaussdb=# GRANT SELECT ON TABLE tpcds.web_returns TO joe;
```

- 给用户指定角色，使用户继承角色所拥有的对象权限。

- 创建角色。

新建一个角色lily，同时给角色指定系统权限CREATEDB：

```
gaussdb=# CREATE ROLE lily WITH CREATEDB PASSWORD '*****';
```

- b. 给角色授予对象权限，请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。

例如，将模式tpcds的权限授予角色lily后，将表tpcds.web_returns的select权限授予角色lily。

```
gaussdb=# GRANT USAGE ON SCHEMA tpcds TO lily;  
gaussdb=# GRANT SELECT ON TABLE tpcds.web_returns TO lily;
```

- c. 将角色的权限授予用户。

```
gaussdb=# GRANT lily TO joe;
```

说明

当将角色的权限授予用户时，角色的属性并不会传递到用户。

- 回收用户权限，请参见《参考》中“SQL参考 > SQL语法 > R > REVOKE”章节。

2.1.7 行级访问控制

行级访问控制特性将数据库访问控制精确到数据表行级别，使数据库达到行级访问控制的能力。不同用户执行相同的SQL查询操作，读取到的结果是不同的。

用户可以在数据表创建行访问控制(Row Level Security)策略，该策略是指针对特定数据库用户、特定SQL操作生效的表达式。当数据库用户对数据表访问时，若SQL满足数据表特定的Row Level Security策略，在查询优化阶段将满足条件的表达式，按照属性(PERMISSIVE | RESTRICTIVE)类型，通过AND或OR方式拼接，应用到执行计划上。

行级访问控制的目的是控制表中行级数据可见性，通过在数据表上预定义Filter，在查询优化阶段将满足条件的表达式应用到执行计划上，影响最终的执行结果。当前受影响的SQL语句包括SELECT、UPDATE和DELETE。

例如：某表中汇总了不同用户的数据，但是不同用户只能查看自身相关的数据信息，不能查看其他用户的数据信息。

- 非透明多写特性下：

```
--创建用户alice, bob, peter  
gaussdb=# CREATE USER alice PASSWORD '*****';  
gaussdb=# CREATE USER bob PASSWORD '*****';  
gaussdb=# CREATE USER peter PASSWORD '*****';  
  
--创建表all_data，包含不同用户数据信息  
gaussdb=# CREATE TABLE all_data(id int, role varchar(100), data varchar(100));  
  
--向数据表插入数据  
gaussdb=# INSERT INTO all_data VALUES(1, 'alice', 'alice data');  
gaussdb=# INSERT INTO all_data VALUES(2, 'bob', 'bob data');  
gaussdb=# INSERT INTO all_data VALUES(3, 'peter', 'peter data');  
  
--将表all_data的读取权限赋予alice, bob和peter用户  
gaussdb=# GRANT SELECT ON all_data TO alice, bob, peter;  
  
--打开行访问控制策略开关  
gaussdb=# ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;  
  
--创建行访问控制策略，当前用户只能查看用户自身的数据  
gaussdb=# CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role =  
CURRENT_USER);  
  
--查看表详细信息  
gaussdb=# \d+ all_data  
Table "public.all_data"  
Column | Type | Modifiers | Storage | Stats target | Description
```

```
-----+-----+-----+-----+-----+-----+
id | integer | | plain | |
role | character varying(100) | | extended | |
data | character varying(100) | | extended | |
Row Level Security Policies:
  POLICY "all_data_rls" FOR ALL
  TO public
  USING (((role)::name = "current_user"()))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

--切换至用户alice，执行SQL"SELECT * FROM public.all_data"
gaussdb=# SELECT * FROM public.all_data;
id | role | data
-----+-----+-----
1 | alice | alice data
(1 row)

gaussdb=# EXPLAIN(COSTS OFF) SELECT * FROM public.all_data;
QUERY PLAN
-----
Streaming (type: GATHER)
  Node/s: All datanodes
  -> Seq Scan on all_data
      Filter: ((role)::name = 'alice'::name)
Notice: This query is influenced by row level security feature
(5 rows)

--切换至用户peter，执行SQL"SELECT * FROM public.all_data"
gaussdb=# SELECT * FROM public.all_data;
id | role | data
-----+-----+-----
3 | peter | peter data
(1 row)

gaussdb=# EXPLAIN(COSTS OFF) SELECT * FROM public.all_data;
QUERY PLAN
-----
Streaming (type: GATHER)
  Node/s: All datanodes
  -> Seq Scan on all_data
      Filter: ((role)::name = 'peter'::name)
Notice: This query is influenced by row level security feature
(5 rows)
```

须知

- PG_STATISTIC系统表和PG_STATISTIC_EXT系统表存储了统计对象的一些敏感信息，如高频值MCV。若创建行级访问控制后，将这两张系统表的查询权限授予普通用户，则普通用户仍然可以通过访问这两张系统表，获取统计对象里的敏感信息。
- PG_STATS、PG_EXT_STATS、PG_GTT_STATS为统计信息查询视图，校验用户对列的SELECT权限。若普通用户拥有SELECT权限且表上存在行级访问控制，可能查询到无权限查询但刚好出现在统计信息高频值里的部分数据（非整行数据，仅是某列的一个值，统计信息是按列收集计算的）。

2.2 数据库审计

背景信息

数据库审计功能对数据库系统的安全性至关重要。数据库审计管理员可以利用审计日志信息，重现导致数据库现状的一系列事件，找出非法操作的用户、时间和内容等。

关于审计功能，用户需要了解以下几点内容：

- 审计总开关GUC参数audit_enabled支持动态加载。在数据库运行期间修改该配置项的值会立即生效，无需重启数据库。默认值为on，表示开启审计功能。
- 除了审计总开关，各个审计项也有对应的开关。只有开关开启，对应的审计功能才能生效。
- 各审计项的开关支持动态加载。在数据库运行期间修改审计开关的值，不需要重启数据库便可生效。

目前，GaussDB支持以下审计项如表2-2所示。如需要修改具体的审计配置项，请联系管理员进行处理。参数类型及取值范围等详细信息请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 审计”章节。

表 2-2 配置审计项

配置项	描述
用户登录、注销审计	参数：audit_login_logout 默认值为7，表示开启用户登录、退出的审计功能。设置为0表示关闭用户登录、退出的审计功能。不推荐设置除0和7之外的值。
数据库启动、停止、恢复和切换审计	参数：audit_database_process 默认值为1，表示开启数据库启动、停止、恢复和切换的审计功能（透明特写特性不支持数据库启动、停止、恢复和切换审计）。
用户锁定和解锁审计	参数：audit_user_locked 默认值为1，表示开启审计用户锁定和解锁功能。
用户访问越权审计	参数：audit_user_violation 默认值为1，表示开启用户越权操作审计功能。
授权和回收权限审计	参数：audit_grant_revoke 默认值为1，表示开启审计用户权限授予和回收功能。
对用户操作进行全量审计	参数：full_audit_users 默认值为空字符串，表示采用默认配置，未配置全量审计用户。
不需要审计的客户端名称及IP地址	参数：no_audit_client 默认值为空字符串，表示采用默认配置，未将客户端及IP加入审计黑名单。

配置项	描述
数据库对象的CREATE，ALTER，DROP操作审计	参数：audit_system_object 非透明多写特性下：默认值为67121159，表示对DATABASE、SCHEMA、USER、NODE GROUP、SQLPatch这五类数据库对象的CREATE、ALTER、DROP操作进行审计。
具体表的INSERT、UPDATE和DELETE操作审计	参数：audit_dml_state 默认值为0，表示关闭具体表的DML操作（SELECT除外）审计功能。
SELECT操作审计	参数：audit_dml_state_select 默认值为0，表示关闭SELECT操作审计功能。
COPY审计	参数：audit_copy_exec 默认值为1，表示开启copy操作审计功能。
执行存储过程和自定义函数的审计	参数：audit_function_exec 默认值为0，表示不记录执行存储过程和自定义函数的审计日志。
执行白名单内的系统函数审计	参数：audit_system_function_exec 默认值为0，表示不记录执行系统函数的审计日志。
SET审计	参数：audit_set_parameter 默认值为0，表示不记录SET操作审计日志
事务ID记录	参数：audit_xid_info 默认值为0，表示关闭审计日志记录事务ID功能。
内部工具连接及操作审计、DN上对来自CN的登录或者退出登录审计	参数：audit_internal_event 非透明多写特性下：默认值为off，表示不对内部工具cm_agent、gs_clean和WDRXdb的登录或者退出登录及操作进行审计，DN上不对来自CN的登录或者退出登录进行审计。

说明

当使用EXPLAIN查看执行计划时，如记录审计日志，记录的审计类型由分析的具体SQL决定。

3 数据库使用入门

3.1 连接数据库

连接数据库的客户端工具包括gsq、应用程序接口（如ODBC和JDBC）。

- gsq是GaussDB自带的客户端工具。[使用gsq连接](#)数据库，可以交互式地输入、编辑、执行SQL语句。
- 用户可以使用标准的数据库[应用程序接口](#)（如ODBC和JDBC），开发基于GaussDB的应用程序。

3.1.1 使用 gsq 连接

gsq是GaussDB提供的在命令行下运行的数据库连接工具。此工具除了具备操作数据库的基本功能，还提供了若干高级特性，便于用户使用。

注意事项

缺省情况下，客户端连接数据库后处于空闲状态时会根据GUC参数session_timeout的默认值自动断开连接。如果要关闭超时设置，设置GUC参数session_timeout为0即可。

前提条件

已联系管理员获取连接信息。

远程连接数据库-非透明多写

步骤1 完成远程连接配置，请联系管理员处理。

步骤2 在客户端机器上，上传客户端工具包并配置gsq的执行环境变量。

1. 登录客户端机器。
2. 创建“/tmp/tools”目录（“/tmp/tools”目录仅为示例，也可创建其他目录）。
`mkdir /tmp/tools`
3. 获取软件安装包中的“GaussDB-Kernel_数据库版本号_操作系统版本号_64bit_Gsq.tar.gz”工具包并上传到“/tmp/tools”路径下。

📖 说明

- 软件包相对位置为安装时所放位置，根据实际情况填写。
 - 不同的操作系统，工具包文件名称会有差异。请根据实际的操作系统类型选择对应的工具包。
4. 解压文件。

```
cd /tmp/tools  
tar -zxvf GaussDB-Kernel_数据库版本号_操作系统版本号_64bit_Gsql.tar.gz
```
 5. 设置环境变量。
打开“~/.bashrc”文件。

```
vi ~/.bashrc
```


在其中输入如下内容后，使用“:wq!”命令保存并退出。

```
export PATH=/tmp/tools/bin:$PATH  
export LD_LIBRARY_PATH=/tmp/tools/lib:$LD_LIBRARY_PATH
```
 6. 使环境变量配置生效。

```
source ~/.bashrc
```

步骤3 连接数据库。

数据库安装完成后，会默认生成名称为postgres的数据库。

- 使用CN所在的服务器IP地址连接数据库。

```
gsql -d postgres -h 10.10.0.11 -U jack -p 8000
```


Password for user jack:

postgres为需要连接的数据库名称，10.10.0.11为CN所在的服务器IP地址，jack为登录数据库的用户名，8000为CN的端口号，上述示例中，连接的服务器的IP和端口也可以替换成DN服务器IP和端口。gsql支持使用域名代替IP地址。
- 连接成功后，系统显示类似如下信息：

```
gsql ((GaussDB Kernel XXX.X.XXX build 9235acb6) compiled at 2025-06-10 07:15:43 last mr 23922 release)  
Non-SSL connection (SSL connection is recommended when requiring high-security)  
Type "help" for help.
```


jack用户是管理员用户时，系统显示“DBNAME=#”。若使用普通用户身份登录和连接数据库，系统显示“DBNAME=>”。

“Non-SSL connection”表示未使用SSL方式连接数据库。如果需要高安全性时，请使用SSL连接。
- 使用IPv6地址连接数据库。

```
gsql -d postgres -h 2407:c182:14f0:594:er63:49c5:493q:594c -U jack -p 8000
```
- 3 CN节点连接数据库。

在多CN数据库实例环境中，当需要跳过故障CN节点时，可以使用gsql连接到正常状态的CN节点。

以3个CN的数据库实例为例，三个CN节点的IP:port分别是10.10.0.11:8000、10.10.0.12:8002和10.10.0.13:8004，执行以下命令。

```
gsql -d postgres -h 10.10.0.11,10.10.0.12,10.10.0.13 -U jack -p 8000,8002,8004
```


gsql会按从前往后的顺序依次连接三个IP:port，如果当前连接的IP:port不是正常状态的CN节点，则尝试连接下一个IP:port，直到成功连接到正常状态的CN节点为止。

可通过设置PGTARGETSESSIONATTRS环境变量的值来选择连接到不同类型的节点。

说明

- 上述示例中，jack用户为数据库初始化用户创建的普通用户。
- 默认禁止使用数据库初始化用户进行远程连接数据库，开启集群内部kerberos认证时，允许初始化用户在集群内部进行远程连接。

----结束

3.1.2 应用程序接口

用户可以使用标准的数据库应用程序接口（如ODBC、JDBC等），开发基于GaussDB的应用程序。

支持的应用程序接口

每个应用程序是一个独立的GaussDB开发项目。应用程序通过API与数据库进行交互，在避免了应用程序直接操作数据库系统的同时，增强了应用程序的可移植性、扩展性和可维护性。[表3-1](#)为GaussDB所支持的应用程序接口及其下载地址。

表 3-1 数据库应用程序接口

API	下载地址
ODBC	• Linux: 驱动程序: GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Odbc.tar.gz unixODBC源码包: https://gitee.com/src-openeuler/unixODBC/blob/openEuler-22.03-LTS-SP1/unixODBC-2.3.7.tar.gz • Windows: 驱动程序: GaussDB-Kernel_数据库版本号_Windows_Odbc.tar.gz
JDBC	• 驱动程序: GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Jdbc.tar.gz • 驱动类: com.huawei.gaussdb.jdbc.Driver

使用JDBC和ODBC接口连接数据库属远程连接，因此需要GaussDB已经做了支持远程连接的配置。相关操作请联系管理员处理。

更多支持的应用程序接口详细信息请参见[应用程序开发教程](#)。

3.2 操作数据库

本节描述使用数据库的基本操作。通过此节您可以完成创建数据库用户、创建数据库、创建表及向表中插入数据和查询表中数据等操作。不建议使用初始用户执行数据库对象的创建操作，否则可能导致业务用户无权查看或操作相关对象，推荐使用专门创建的业务用户来执行此类操作。

3.2.1 创建数据库用户

默认只有集群安装时创建的管理员用户可以访问初始数据库，您还可以创建其他数据库用户账号。


```
gaussdb=# CREATE USER joe WITH PASSWORD '*****';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE ROLE
```

如上创建了一个用户名为joe，密码为*****的用户。执行如下命令设置joe用户为系统管理员。

```
gaussdb=# GRANT ALL PRIVILEGES TO joe;
```

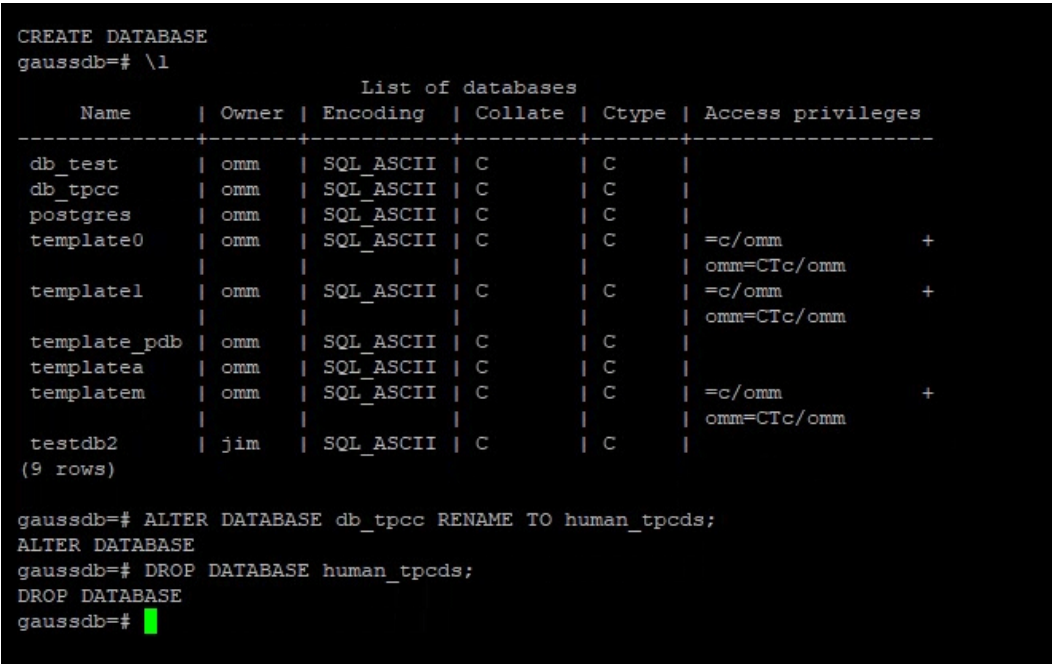
非透明多写特性下，使用GRANT命令进行相关权限设置，具体操作请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。

说明

关于数据库用户的更多信息请参见[用户及权限](#)。

3.2.2 管理数据库-非透明多写

图 3-1 创建和管理数据库



前提条件

用户需具备数据库创建权限或系统管理员权限方可执行数据库创建操作，具体权限授予方法请参见[用户及权限](#)。

背景信息

- 初始时，GaussDB包含四个模板数据库template0、template1、templatea、templatem，以及一个默认的用户数据库postgres。
- CREATE DATABASE实际上通过复制模板数据库来创建新数据库。默认复制template0。请避免使用客户端或其他方式连接及操作两个模板数据库。

说明

- 模板数据库中沒有用户表，可通过系统表PG_DATABASE查看模板数据库属性。
- 模板template0、templatea不允许用户连接；模板template1和templatem只允许数据库初始用户和系统管理员连接，普通用户无法连接。
- templatem为系统保留数据库名称，不允许用户以其为数据库名创建数据库。
- templatea、templatem不允许用户修改名称。
- 数据库系统中会有多个数据库，但是同一时刻客户端程序只能连接一个数据库。当前，不支持在不同的数据库之间进行相互查询（跨库查询或跨库事务）。
- 可以通过客户端工具gsqsl的-d参数指定目标数据库进行登录，也可以在客户端程序登录数据库以后通过\c命令进行数据库切换。

注意事项

如果数据库的编码为SQL_ASCII（可以通过“SHOW server_encoding”命令查看当前数据库存储编码），则在创建数据库对象时，如果对象名中含有多字节字符（例如中文），超过数据库对象名长度限制（63字节）的时候，数据库会将最后一个字节（而不是字符）截断，可能造成出现半个字符的情况。

针对这种情况，请遵循以下条件：

- 保证数据对象的名称不超过限定长度。
- 使用UTF8编码集为数据库的默认存储编码集（server_encoding）。
- 不要使用多字节字符作为对象名。
- 因为误操作导致在多字节字符的中间截断，从而导致无法删除数据库对象，如果出现这种现象，请使用截断前的数据库对象名进行删除操作，或将该对象从各个数据库节点的相应系统表中依次删掉。

操作步骤

步骤1 执行如下命令创建一个新的数据库db_tpcds。

```
gaussdb=# CREATE DATABASE db_tpcds;  
CREATE DATABASE  
gaussdb=# CREATE DATABASE db_tpcc1 TEMPLATE templatea;  
CREATE DATABASE
```

📖 说明

- 数据库名称遵循SQL标识符的一般规则。当前角色自动成为此新数据库的所有者。
- 如果一个数据库系统用于承载相互独立的用户和项目，建议把它们放在不同的数据库里。
- 如果项目或者用户是相互关联的，并且可以相互使用对方的资源，则应该把它们放在同一个数据库里，但可以规划在不同的模式中。模式只是一个纯粹的逻辑结构，某个模式的访问权限由权限系统模块控制。
- 创建数据库时，若数据库名称长度超过63字节，server端会对数据库名称进行截断，保留前63个字节，因此建议数据库名称长度不要超过63个字节。
- 数据库默认创建在pg_default表空间下。若要指定表空间，可以使用如下语句。

```
gaussdb=# CREATE DATABASE db_tpcds WITH TABLESPACE = hr_local;
CREATE DATABASE
```

其中hr_local为表空间名称，关于如何创建表空间，请参见[管理表空间](#)。
- 创建完db_tpcds数据库后，可以选择继续在默认的postgres数据库进行其他操作，也可以按如下方法退出postgres数据库，使用新用户连接到此数据库执行创建表等操作。

```
gaussdb=# \q
gsq! -d db_tpcds -p 8000 -U joe
Password for user joe:
gsq!((GaussDB Kernel XXX.XXX.XXX build f521c606) compiled at 2021-09-16 14:55:22 last mr 6385
release)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
```

步骤2 查看数据库。

- 执行\l元命令查看数据库系统的数据库列表。

```
gaussdb=# \l
```
- 执行如下命令通过系统表pg_database查询数据库列表。

```
gaussdb=# SELECT datname FROM pg_database;
```

步骤3 修改数据库。

用户可以使用如下命令修改数据库属性（比如：owner、名称和默认的配置属性）。

- 执行如下命令为数据库重新命名。

```
gaussdb=# ALTER DATABASE db_tpcds RENAME TO human_tpcds;
ALTER DATABASE
```

📖 说明

执行完参数设置后，需要手动执行CLEAN CONNECTION清理旧连接，否则可能存在节点间参数值不一致。

步骤4 删除数据库。

用户可通过DROP DATABASE命令删除数据库。该命令会移除数据库系统目录并永久删除磁盘上的数据文件。仅数据库所有者或者拥有系统管理员权限的用户可以执行删除操作。若存在活跃数据库连接，删除操作将失败，操作前请确保已连接到其他数据库。

执行如下命令删除数据库：

```
gaussdb=# DROP DATABASE human_tpcds;
DROP DATABASE
```

----结束

3.2.3 管理表空间

图 3-2 创建和管理表空间

```
gaussdb=# CREATE TABLESPACE tbs_t1 RELATIVE LOCATION 'tablespace/tbs_t1';
CREATE TABLESPACE
gaussdb=# CREATE TABLE foo(i int) TABLESPACE tbs_t1;
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'i' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# \db
          List of tablespaces
   Name   | Owner | Location
-----+-----+-----
pg_default | omm1  |
pg_global  | omm1  |
tbs_t1    | omm1  | tablespace/tbs_t1
ts_t1     | omm1  | tablespace/ts_t1
(4 rows)

gaussdb=# ALTER TABLESPACE tbs_t1 RENAME TO fspace;
ALTER TABLESPACE
gaussdb=# DROP TABLE foo;
DROP TABLE
gaussdb=# DROP TABLESPACE fspace;
DROP TABLESPACE
gaussdb=#
```

背景信息

非透明多写特性下：

通过使用表空间，管理员可以控制一个数据库安装的磁盘布局。这样有以下优点：

- 如果初始化数据库所在的分区或者表空间已满，又不能逻辑上扩展更多空间，可以在不同的分区上创建和使用表空间，直到系统重新配置空间。
- 表空间允许管理员根据数据库对象的使用模式安排数据位置，从而提高性能。
 - 一个频繁使用的索引可以存储在性能稳定且运算速度较快的磁盘上，比如一种固态设备。
 - 一个存储归档的数据，很少使用的或者对性能要求不高的表可以存储在一个运算速度较慢的磁盘上。
- 管理员通过表空间可以设置占用的磁盘空间，用以在和其他数据共用分区的时候，防止表空间占用相同分区上的其他空间。
- 表空间可以控制数据库数据占用的磁盘空间。当表空间所在磁盘的使用率达到 `cm_server` 参数 `datastorage_threshold_value_check` 时，数据库将被设置为只读模式，当磁盘使用率降到 `datastorage_threshold_value_check - datastorage_threshold_recover` 以下时，数据库将恢复到读写模式。CM的磁盘自动检查功能默认是开启的，若未开启，请联系管理员开启。
- 表空间对应于一个文件系统目录，比如“数据目录/pg_location/mount1/path1”是用户拥有读写权限的空目录，那么可以在这个目录下创建一个绝对路径表空间。

通过MAXSIZE进行表空间配额管理对并发插入性能可能会有30%左右的影响，MAXSIZE指定每个DN的配额大小，每个DN实际的表空间容量和配额误差在500MB以内。请根据实际情况确认是否需要设置表空间的最大值。

- GaussDB自带两个表空间：pg_default和pg_global。
 - 默认表空间pg_default：用来存储非共享系统表、用户表、用户表INDEX、临时表、临时表INDEX、内部临时表的默认表空间。对应存储目录为实例数据目录下的base目录。
 - 共享表空间pg_global：用来存放共享系统表的表空间。对应存储目录为实例数据目录下的global目录。

注意事项

非透明多写特性下：

- 在公有云场景下一般不建议用户使用自定义的表空间。

原因：用户自定义表空间通常配合主存（即默认表空间所在的存储设备，如磁盘）以外的其它存储介质使用，以隔离不同业务可以使用的I/O资源，而在公有云场景下，存储设备都是采用标准化的配置，无其它可用的存储介质，自定义表空间使用不当不利于系统长稳运行以及影响整体性能，因此建议使用默认表空间即可。
- 在主备环境下，创建表空间时不允许指定备机上权限不足的文件系统目录。

操作步骤

步骤1 创建表空间。

1. 执行如下命令创建用户为jack，密码为*****。

```
gaussdb=# CREATE USER jack IDENTIFIED BY '*****';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE ROLE
```
2. 执行如下命令创建表空间。

```
gaussdb=# CREATE TABLESPACE fastspace RELATIVE LOCATION 'my_tablespace/tablespace1';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLESPACE
```

其中“fastspace”为新创建的表空间，“CN和DN数据目录/pg_location/my_tablespace/tablespace1”是用户拥有读写权限的空目录。
3. 数据库系统管理员执行如下命令将“fastspace”表空间的访问权限授予数据用户jack。

```
gaussdb=# GRANT CREATE ON TABLESPACE fastspace TO jack;
```

当结果显示为如下信息，则表示授予成功。

```
GRANT
```

步骤2 在表空间中创建对象。

如果用户拥有表空间的CREATE权限，就可以在表空间上创建数据库对象，比如：表和索引等。

在表空间中创建对象有以下两种方式，以创建表为例。

- 执行如下命令在指定表空间创建表。

```
gaussdb=# CREATE TABLE foo(i int) TABLESPACE fastspace;
```

当结果显示为如下信息，则表示创建成功。

```
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'i' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

- 先使用SET default_tablespace设置默认表空间，再创建表。

--- 假设设置“fastspace”为默认表空间，然后创建表foo2。

```
gaussdb=# SET default_tablespace = 'fastspace';
```

```
SET
```

```
gaussdb=# CREATE TABLE foo2(i int);
```

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

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

```
CREATE TABLE
```

步骤3 查询表空间。

可通过以下两种方式查询表空间。

- 检查pg_tablespace系统表。如下命令可查到系统和用户定义的全部表空间。

```
gaussdb=# SELECT spcname FROM pg_tablespace;
```

- 使用gspl程序的元命令查询表空间。

```
gaussdb=# \db
```

步骤4 查询表空间使用率。

1. 查询表空间的当前使用情况。

```
gaussdb=# SELECT pg_tablespace_size('fastspace');
```

返回如下信息：

```
pg_tablespace_size
```

```
-----
```

```
2146304
```

```
(1 row)
```

其中2146304表示表空间的大小，单位为字节。

2. 非透明多写特性下，可使用如下方式计算表空间使用率。

表空间使用率=pg_tablespace_size/表空间所在目录的磁盘大小。

步骤5 修改表空间命名。

执行如下命令对表空间fastspace重命名为fspace。

```
gaussdb=# ALTER TABLESPACE fastspace RENAME TO fspace;
```

```
ALTER TABLESPACE
```

说明

仅作为表空间owner的用户或系统管理员有权修改表空间。

步骤6 删除表空间。

1. 执行如下命令删除用户jack。

```
gaussdb=# DROP USER jack CASCADE;
```

```
DROP ROLE
```

2. 执行如下命令删除表foo和foo2。

```
gaussdb=# DROP TABLE foo;
```

```
gaussdb=# DROP TABLE foo2;
```

当结果显示为如下信息，则表示删除成功。

```
DROP TABLE
```

3. 执行如下命令删除表空间fspace。

```
gaussdb=# DROP TABLESPACE fspace;
```

```
DROP TABLESPACE
```

说明

用户必须是表空间的owner或者系统管理员才能删除表空间。

----结束

3.2.4 管理表

3.2.4.1 创建表

背景信息

表是建立在数据库中的，在不同的数据库中可以存放相同的表，甚至可以通过使用模式在同一个数据库中创建相同名称的表。

创建表

执行如下命令创建表。

```
gaussdb=# CREATE TABLE customer_t1
(
  c_customer_sk      integer,
  c_customer_id      char(5),
  c_first_name       char(6),
  c_last_name        char(8)
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

其中c_customer_sk、c_customer_id、c_first_name和c_last_name是表的字段名，integer、char(5)、char(6)和char(8)分别是这四个字段的类型。

说明

- 默认情况下，新的数据库对象是创建在“\$user”模式下。关于模式的更多信息请参见[管理 Schema](#)。
- 除了创建的表以外，数据库还包含很多系统表。这些系统表包含集群安装信息以及GaussDB上运行的各种查询和进程的信息。可以通过查询系统表来收集有关数据库的信息。请参见[查看系统表](#)。
- 非透明多写特性下，关于创建表的更多信息请参见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE”章节。

3.2.4.2 向表中插入数据

在创建一个表后，表中并没有数据，在使用这个表之前，需要向表中插入数据。本小节介绍如何使用INSERT命令插入一行或多行数据，及从指定表插入数据。如果有大量数据需要批量导入表中请联系管理员处理。

背景信息

服务端与客户端使用不同的字符集时，两者字符集中单个字符的长度也会不同，客户端输入的字符串会以服务端字符集的格式进行处理，所以产生的最终结果可能会与预期不一致。

表 3-2 客户端和服务端设置字符集的输出结果对比

操作过程	服务端和客户端编码一致	服务端和客户端编码不一致
存入和取出过程中没有对字符串进行操作。	输出预期结果。	输出预期结果（客户端编码保持不变）。
存入取出过程对字符串有做一定的操作（如字符串函数操作）。	输出预期结果。	根据对字符串具体操作可能产生非预期结果。
存入过程中对超长字符串有截断处理。	输出预期结果。	字符集中字符编码长度是否一致，如果不一致可能会产生非预期的结果。

上述字符串函数操作与自动截断机制存在叠加影响风险。例如：在客户端与服务端字符集不一致的场景下，当同时存在字符串函数处理操作和自动截断机制时，系统将在完成字符串处理后继续执行二次截断，从而引发非预期结果。详细的示例请参见表 3-3。

说明

数据库DBCOMPATIBILITY设为兼容TD（Teradata）模式，且GUC参数td_compatible_truncation设置为on的情况下，才会对超长字符串进行截断。

执行如下命令建立示例中需要使用的表table1、table2。

```
gaussdb=# CREATE TABLE table1(id int, a char(6), b varchar(6),c varchar(6));
gaussdb=# CREATE TABLE table2(id int, a char(20), b varchar(20),c varchar(20));
```

表 3-3 示例

编号	1	2	3	4
服务端字符集	SQL_ASCII	SQL_ASCII	SQL_ASCII	SQL_ASCII
客户端字符集	UTF8	UTF8	UTF8	UTF8
是否启用自动截断	是	是	是	否
示例	gaussdb=# INSERT INTO table1 VALUES(1,reverse('123AAA78'),reverse('123AA78'),reverse('123AA78'));	gaussdb=# INSERT INTO table1 VALUES(2,reverse('123A78'),reverse('123A78'),reverse('123A78'));	gaussdb=# INSERT INTO table1 VALUES(3,'87A123','87A123','87A123');	gaussdb=# INSERT INTO table2 VALUES(1,reverse('123AA78'),reverse('123AA78'),reverse('123AA78')); gaussdb=# INSERT INTO table2 VALUES(2,reverse('123A78'),reverse('123A78'),reverse('123A78'));

结果	id a b c -----+----- +-----+----- 1 87 87 87	id a b c -----+----- +-----+----- 2 873 873 873	id a b c -----+-----+----- +----- 3 87A1 87A1 87A1	id a b c -----+----- +----- 1 87 321 87 321 87 321 2 87321 87321 87321
说明	字符串在服务端翻转后，并进行截断，由于服务端和客户端的字符集不一致，字符A在客户端由多个字节表示，结果产生异常。	字符串翻转后，又进行了自动截断，所以产生了非预期的效果。	由于字符串类型字段的长度是客户端字符编码长度的整数倍，所以截断后产生结果正常。	与示例1类似，多字节字符翻转之后不再表示原来的字符。

操作步骤

向表中插入数据前，请确保表已创建成功。创建表的步骤请参见[管理表](#)。

- 向表customer_t1中插入一行数据。
数据值是按照这些字段在表中出现的顺序列出的，并且用逗号分隔。通常数据值是文本（常量），但也允许使用表达式。
gaussdb=# INSERT INTO customer_t1(c_customer_sk, c_customer_id, c_first_name) VALUES (3769, 'hello', 'Grace');
如果用户已经知道表中字段的顺序，也可无需列出表中的字段。例如以下命令与上面的命令效果相同。
gaussdb=# INSERT INTO customer_t1 VALUES (3769, 'hello', 'Grace');
如果用户不知道所有字段的数值，可以忽略其中的一些。如果INSERT语句中，用户没有显式指定目标表的字段名字，那么VALUES子句中待插入的多个值，将按照列号和目标表的字段一一对应，即VALUES子句的第一个值对应目标表的第一列，VALUES子句的第二个值对应目标表的第二列，以此类推，没有VALUES数值对应的列自动填充缺省值或NULL。没有数值的字段将被填充为字段的缺省值。例如：
gaussdb=# INSERT INTO customer_t1 (c_customer_sk, c_first_name) VALUES (3769, 'Grace');

gaussdb=# INSERT INTO customer_t1 VALUES (3769, 'hello');
用户也可以对独立的字段或者整个行明确缺省值：
gaussdb=# INSERT INTO customer_t1 (c_customer_sk, c_customer_id, c_first_name) VALUES (3769, 'hello', DEFAULT);

gaussdb=# INSERT INTO customer_t1 DEFAULT VALUES;
 - 如果需要在表中插入多行，请执行如下命令：
gaussdb=# INSERT INTO customer_t1 (c_customer_sk, c_customer_id, c_first_name) VALUES
 (6885, 'maps', 'Joes'),
 (4321, 'tpcds', 'Lily'),
 (9527, 'world', 'James');
- 如果需要向表中插入多条数据，除此命令外，也可以多次执行插入一行数据命令实现。但是建议使用此命令以提升效率。

- 如果从指定表插入数据到当前表，例如在数据库中创建了一个表customer_t1的备份表customer_t2，现在需要将表customer_t1中的数据插入到表customer_t2中，则可以执行如下命令：

```
gaussdb=# CREATE TABLE customer_t2
(
    c_customer_sk      integer,
    c_customer_id      char(5),
    c_first_name       char(6),
    c_last_name        char(8)
);

gaussdb=# INSERT INTO customer_t2 SELECT * FROM customer_t1;
```

📖 说明

从指定表插入数据到当前表时，若指定表与当前表对应的字段数据类型之间不存在隐式转换，则这两种数据类型必须相同。

- 使用不带AS的表别名向表中插入数据。

```
gaussdb=# CREATE TABLE table_name(a int,b text,c timestamp,d float);
gaussdb=# INSERT INTO table_name alias_name VALUES(1,'ha','1997-01-01
10:00:00'::timestamp,1.234);
gaussdb=# INSERT INTO table_name alias_name(b,c,d) VALUES('ha','1997-01-01
10:00:00'::timestamp,1.234);
```

📖 说明

- INSERT不带AS的表别名不能为关键字（如SELECT、VALUE）或者表达式，别名要求符合标识符命名规则。
 - INSERT不带AS的表别名不支持INSERT INTO table_name alias_name(alias_name.col1,...,alias_name.coln) VALUES(xxx)格式。
 - INSERT的分区表不带AS的表别名不支持指定分区插入操作。
- 删除备份表。
gaussdb=# DROP TABLE customer_t2 CASCADE;

📖 说明

在删除表的时候，若当前需删除的表与其他表有依赖关系，需先删除关联的表，然后再删除当前表。

3.2.4.3 更新表中数据

修改已经存储在数据库中数据的行为叫做更新。用户可以更新单独一行，所有行或者指定的部分行。还可以独立更新每个字段，而其他字段则不受影响。

使用UPDATE命令更新现有行，需要提供以下三种信息：

- 表的名称和要更新的字段名
- 字段的新值
- 要更新的行

SQL通常不会为数据行提供唯一标识，因此更新操作需通过WHERE子句的条件表达式定位目标行。若表中定义主键，则可通过主键值精确指定待更新行；否则必须依赖业务逻辑条件组合进行更新。

建立表和插入数据的步骤请参见[创建表](#)和[向表中插入数据](#)。

需要将表customer_t1中c_customer_sk为9527的地域重新定义为9876。

```
gaussdb=# UPDATE customer_t1 SET c_customer_sk = 9876 WHERE c_customer_sk = 9527;
```

这里的表名称也可以使用模式名修饰，否则会从数据库的默认模式查询该表。SET后面紧跟字段和新的字段值。新的字段值不仅可以是常量，也可以是表达式。

比如，把所有c_customer_sk的值增加100。

```
gaussdb=# UPDATE customer_t1 SET c_customer_sk = c_customer_sk + 100;
```

在这里省略了WHERE子句，表示表中的所有行都要被更新。如果出现了WHERE子句，那么只有匹配其条件的行才会被更新。

在SET子句中的等号是一个赋值，而在WHERE子句中的等号是比较。WHERE条件不一定是相等比较，许多其他的操作符也可以使用。

用户可以在一个UPDATE命令中更新更多的字段，方法是在SET子句中列出更多赋值，比如：

```
gaussdb=# UPDATE customer_t1 SET c_customer_id = 'Admin', c_first_name = 'Local' WHERE c_customer_sk = 4421;
```

批量更新或删除数据后，会在数据文件中产生大量的删除标记，查询过程中标记删除的数据也是需要扫描的。故多次批量更新/删除后，标记删除的数据量过大会严重影响查询的性能。建议在批量更新/删除业务会反复执行的场景下，定期执行VACUUM FULL以保持查询性能。

批量更新表中数据

对包含数百万条记录的表进行批量更新时，直接执行更新操作会导致表锁定，显著影响其他操作的性能。为了避免这种情况，可以采取以下几种方法来优化批量更新操作：

- 分批更新

将大规模更新拆分为多个小批次，每次仅更新部分数据，从而缩短单次锁表的时间。

例如，原批量更新语句：UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';

优化方案：将批量更新按ID拆分为多条UPDATE语句，每次更新部分数据。

```
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id >= 1 and id <= 10000;
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id > 10000 and id <= 20000;
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id > 20000 and id <= 30000;
...
```

- 使用临时表

事务中存在复杂查询的情况下，可以将需要更新的记录先插入到临时表中，然后再从临时表中更新主表，避免长时间锁定这些数据行。

例如，原批量更新语句：UPDATE products SET price = price * 1.1 WHERE (size = 101 OR size = 102) AND category = 'Electronics' AND amount > 150;

优化方案：创建临时表存储符合条件的目标记录并进行更新，随后使用该临时表更新主表。

```
gaussdb=# CREATE TEMPORARY TABLE temp_products AS SELECT id, price * 1.1 as new_price from products WHERE (size = 101 OR size = 102) AND category = 'Electronics' AND amount > 150;
gaussdb=# UPDATE products p SET price = t.new_price FROM temp_products t where p.id = t.id;
gaussdb=# DROP TABLE temp_products;
```

- 使用并行更新

在多线程或多进程环境中，可以将更新操作并行化，多个线程同时更新不同的记录。这需要确保每个线程处理的记录不重叠。

例如，原批量更新语句：UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';

优化方案：根据ID将数据拆分为多个小批次，启动多个会话同时执行UPDATE操作，通过并发更新降低批量操作的总耗时。需确保各会话更新的元组无重叠。

session1:

```
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id >= 1 and id <= 10000;
```

session2:

```
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id > 10000 and id <= 30000;
```

session3:

```
gaussdb=# UPDATE products SET price = price * 1.1 WHERE category = 'Electronics' AND id > 30000;
```

3.2.4.4 查看数据

- 使用系统表pg_tables查询数据库所有表的信息。
gaussdb=# SELECT * FROM pg_tables;
- 使用gsql的\d+命令查询表的属性。
gaussdb=# \d+ customer_t1
- 执行如下命令查询表customer_t1的数据量。
gaussdb=# SELECT count(*) FROM customer_t1;
- 执行如下命令查询表customer_t1的所有数据。
gaussdb=# SELECT * FROM customer_t1;
- 执行如下命令只查询字段c_customer_sk的数据。
gaussdb=# SELECT c_customer_sk FROM customer_t1;
- 执行如下命令过滤字段c_customer_sk的重复数据。
gaussdb=# SELECT DISTINCT(c_customer_sk) FROM customer_t1;
- 执行如下命令查询字段c_customer_sk为3869的所有数据。
gaussdb=# SELECT * FROM customer_t1 WHERE c_customer_sk = 3869;
- 执行如下命令按照字段c_customer_sk进行排序。
gaussdb=# SELECT * FROM customer_t1 ORDER BY c_customer_sk;

3.2.4.5 删除表中数据

在使用表的过程中，可能会需要删除已过期的数据，删除数据必须从表中整行删除。

SQL不能直接访问独立的行，只能通过声明被删除行匹配的条件进行。如果表中有一个主键，用户可以指定准确的行。用户可以删除匹配条件的一组行或者一次删除表中的所有行。

执行DELETE命令删除行，如删除表customer_t1中所有c_customer_sk为3869的记录。

```
gaussdb=# DELETE FROM customer_t1 WHERE c_customer_sk = 3869;
```

执行如下命令，会删除表中所有的行。

```
gaussdb=# DELETE FROM customer_t1;
```

或：

```
gaussdb=# TRUNCATE TABLE customer_t1;
```

说明

全表删除的场景下，建议使用TRUNCATE，不建议使用DELETE。

删除创建的表。

```
gaussdb=# DROP TABLE customer_t1;
```

3.2.5 查看系统表

除了创建的表以外，数据库还包含很多系统表。这些系统表包含集群安装信息以及 GaussDB上运行的各种查询和进程的信息。可以通过查询系统表来收集有关数据库的信息。

系统表和系统视图（请参见《参考》中“系统表和系统视图”章节）中每个表的说明指出了表是对所有用户可见还是只对初始化用户可见。以初始化用户身份登录才能查询只对初始化用户可见的表。

GaussDB提供了以下类型的系统表和系统视图：

- 兼容PG的系统表和系统视图。这类系统表和系统视图具有PG前缀。
- GaussDB内核新增的系统表和系统视图。这类系统表和系统视图具有GS前缀。
- 兼容ORA数据库的系统表和系统视图。这类系统表和系统视图具有ALL、DBA、USER或PV前缀。

查看数据库中包含的表

在public Schema下新建五张表：

```
gaussdb=# CREATE TABLE public.search_table_t1(a int);  
CREATE TABLE  
gaussdb=# CREATE TABLE public.search_table_t2(b int);  
CREATE TABLE  
gaussdb=# CREATE TABLE public.search_table_t3(c int);  
CREATE TABLE  
gaussdb=# CREATE TABLE public.search_table_t4(d int);  
CREATE TABLE  
gaussdb=# CREATE TABLE public.search_table_t5(e int);  
CREATE TABLE
```

在PG_TABLES系统表中查看public Schema中包含的前缀为search_table的表：

```
gaussdb=# SELECT DISTINCT(tablename) FROM pg_tables WHERE SCHEMANAME = 'public' AND  
TABLENAME LIKE 'search_table%';
```

结果如下：

```
tablename  
-----  
search_table_t1  
search_table_t2  
search_table_t3  
search_table_t4  
search_table_t5  
(5 rows)
```

查看数据库用户

通过PG_USER可以查看数据库中所有用户的列表，还可以查看用户ID（USESYSID）和用户权限。

```
gaussdb=# SELECT * FROM gs_user;  
username | usesysid | usecreatedb | usesuper | usecatupd | userepl | passwd | valbegin | valuntil |  
respool  | parent  | spacelimit | useconfig | nodegroup | tempspacelimit | spillsp
```


3.2.6 其他操作

3.2.6.1 管理 Schema

图 3-3 创建和管理 Schema

```
commit 8868 last mr 17511 release)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

gaussdb=# CREATE SCHEMA myschema;
CREATE SCHEMA
gaussdb=# CREATE TABLE myschema.mytable(id int, name varchar(20));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribu
tion column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution c
olumn.
CREATE TABLE
gaussdb=# SET SEARCH_PATH TO myschema, public;
SET
gaussdb=# SHOW SEARCH_PATH;
      search_path
-----
myschema, public
(1 row)

gaussdb=# DROP SCHEMA myschema CASCADE;
NOTICE: drop cascades to table mytable
DROP SCHEMA
gaussdb=#
```

背景信息

Schema 又称作模式。通过管理 Schema，允许多个用户使用同一数据库而不相互干扰，可以将数据库对象组织成易于管理的逻辑组，同时便于将第三方应用添加到相应的 Schema 下而不引起冲突。管理 Schema 包括：创建 Schema、使用 Schema、删除 Schema、设置 Schema 的搜索路径以及 Schema 的权限控制。

注意事项

- 数据库集群包含一个或多个已命名数据库。用户和用户组在整个集群范围内是共享的，但是其数据并不共享。任何与服务器连接的用户都只能访问连接请求里声明的数据库。
- 一个数据库可以包含一个或多个已命名的 Schema，Schema 又包含表及其他数据库对象，包括数据类型、函数和操作符等。同一对象名可以在不同的 Schema 中使用而不会引起冲突。例如，schema1 和 schema2 都可以包含一个名为 mytable 的表。
- 和数据库不同，Schema 不是严格分离的。用户根据其对 Schema 的权限，可以访问所连接数据库的 Schema 中的对象。进行 Schema 权限管理首先需要对数据库的权限控制进行了解。
- 不能创建以 PG_ 为前缀的 Schema 名，该类 Schema 名为数据库系统预留的。
- 在每次创建新用户时，系统会在当前登录的数据库中为新用户创建一个同名 Schema。对于其他数据库，若需要同名 Schema，则需要用户手动创建。
- 通过未修饰的表名（名称中只含有表名，没有“Schema 名”）引用表时，系统会通过 search_path（搜索路径）来判断该表是哪一个 Schema 下的表。pg_temp 和

pg_catalog始终会作为搜索路径顺序中的前两位，无论二者是否出现在search_path中，或者出现在search_path中的任何位置。search_path（搜索路径）是一个Schema名列表，在其中找到的第一个表就是目标表，如果没有找到则报错。（某个表即使存在，如果它的Schema不在search_path中，依然会查找失败）在搜索路径中的第一个Schema称为“当前Schema”，它是搜索时查询的第一个Schema，同时在没有声明Schema名时，新创建的数据库对象会默认存放在该Schema下。

- 每个数据库都包含一个pg_catalog Schema，它包含系统表和所有内置数据类型、函数和操作符。pg_catalog是搜索路径中的一部分，始终在临时表所属的模式后面，并在search_path中所有模式的前面，即具有第二搜索优先级，以确保可以搜索到数据库内置对象。当用户需要创建与系统内置对象同名的自定义对象时，可以在该对象所属的Schema中执行操作，避免命名冲突。

创建 Schema

具体操作请参见CREATE SCHEMA语法（《参考》中“SQL参考 > SQL语法 > C > CREATE SCHEMA”）。

- 执行如下命令来创建一个Schema。

```
gaussdb=# CREATE SCHEMA myschema;
```

当结果显示为如下信息，则表示成功创建一个名为myschema的Schema。

```
CREATE SCHEMA
```

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

- 执行如下命令在创建Schema时指定owner。

```
gaussdb=# CREATE SCHEMA myschema AUTHORIZATION omm;
```

当结果显示为如下信息，则表示成功创建一个属于omm用户，名为myschema的Schema。

```
CREATE SCHEMA
```

使用 Schema

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

- 执行如下命令在myschema下创建mytable表。

```
gaussdb=# CREATE TABLE myschema.mytable(id int, name varchar(20));  
CREATE TABLE
```

- 执行如下命令查询myschema下mytable表的所有数据。

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

Schema 的搜索路径

通过设置search_path配置参数可以定义数据库查找对象的Schema搜索顺序。搜索路径中的第一个Schema将被设为当前默认Schema。若创建对象时未显式指定所属Schema，系统会自动将其创建至默认Schema中。

- 执行如下命令查看搜索路径。

```
gaussdb=# SHOW search_path;  
search_path  
-----
```



```
"$user",public  
(1 row)
```

- 执行如下命令将搜索路径设置为myschema, public，配置数据库的模式搜索顺序。系统将优先查找myschema，若未找到对应对象则继续搜索public。
gaussdb=# SET search_path TO myschema, public;
SET

Schema 的权限控制

默认情况下，用户仅能访问其拥有的Schema内的数据库对象。若要访问其他Schema中的对象，需由目标Schema所有者授予对应USAGE权限。

通过将模式的CREATE权限授予某用户，被授权用户就可以在此模式中创建对象。默认情况下，所有角色都拥有在public模式上的usage权限，但是普通用户没有在public模式上的CREATE权限。普通用户具备连接指定数据库并在其public模式中创建对象的权限时，将产生安全风险。若已授予普通用户public模式的CREATE权限，建议执行以下语句撤销该权限。

- 撤销PUBLIC在public模式下创建对象的权限，下面语句中第一个“public”是模式，第二个“public”指系统预定义的所有用户角色。

```
gaussdb=# REVOKE CREATE ON SCHEMA public FROM public;  
REVOKE
```

- 执行如下命令查看当前Schema。

```
gaussdb=# SELECT current_schema();  
current_schema  
-----  
myschema  
(1 row)
```

- 执行如下命令创建用户jack，并将myschema的usage权限授予用户jack。

```
gaussdb=# CREATE USER jack IDENTIFIED BY '*****';  
CREATE ROLE  
gaussdb=# GRANT USAGE ON SCHEMA myschema TO jack;  
GRANT
```

- 将用户jack对于myschema的usage权限收回。

```
gaussdb=# REVOKE USAGE ON SCHEMA myschema FROM jack;  
REVOKE
```

删除 Schema

具体操作请参见DROP SCHEMA语法（《参考》中“SQL参考 > SQL语法 > D > DROP SCHEMA”）。

- 当Schema为空时，即该Schema下没有数据库对象，使用DROP SCHEMA命令进行删除。例如删除名为nullschema的空Schema。

```
gaussdb=# DROP SCHEMA IF EXISTS nullschema;  
DROP SCHEMA
```

- 当Schema非空时，如果要删除一个Schema及其包含的所有对象，需要使用CASCADE关键字。例如删除myschema及该Schema下的所有对象。

```
gaussdb=# DROP SCHEMA myschema CASCADE;  
DROP SCHEMA
```

删除Schema后，需删除用户，可执行如下命令删除用户jack。

```
gaussdb=# DROP USER jack;  
DROP ROLE
```

3.2.6.2 管理分区表

背景信息

GaussDB数据库支持的分区表为范围分区表、列表分区表和哈希分区表。

- 范围分区表：将数据基于范围映射到每一个分区，这个范围是由创建分区表时指定的分区键决定的。这种分区方式被广泛应用，并且分区键经常采用日期，例如将销售数据按照月份进行分区。
- 列表分区表：将数据中包含的键值分别存储在不同的分区中，依次将数据映射到每一个分区，分区中包含的键值由创建分区表时指定。
- 哈希分区表：将数据根据内部哈希算法依次映射到每一个分区中，包含的分区个数由创建分区表时指定。

分区表和普通表相比具有以下优点：

- 改善查询性能：对分区对象的查询可以仅搜索与查询条件匹配的分区数据，提高检索效率。
- 增强可用性：如果分区表的某个分区出现故障，表在其他分区的数据仍然可用。
- 方便维护：如果分区表的某个分区出现故障，需要修复数据，只修复该分区即可。
- 均衡I/O：非透明多写特性下，可以把不同的分区映射到不同的磁盘以平衡I/O，改善整个系统性能。

普通表若要转成分区表，需要新建分区表，然后把普通表中的数据导入到新建的分区表中。因此在初始设计表时，请根据业务提前规划是否使用分区表。

操作步骤

本节示例中，将以创建RANGE分区为例给大家介绍相关操作步骤，其他分区示例请参见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE PARTITION”章节的示例。按照本节操作步骤示例操作前，要求用户已创建tpcds Schema。

示例一：使用默认表空间

- 创建分区表

```
gaussdb=# CREATE TABLE tpcds.customer_address
(
    ca_address_sk integer NOT NULL ,
    ca_address_id character(16) NOT NULL ,
    ca_street_number character(10) ,
    ca_street_name character varying(60) ,
    ca_street_type character(15) ,
    ca_suite_number character(10) ,
    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;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

- 插入数据

将表tpcds.customer_address的数据插入到表tpcds.web_returns_p2中。

在数据库中创建表tpcds.customer_address的备份表tpcds.web_returns_p2，将表tpcds.customer_address中的数据插入到表tpcds.web_returns_p2中，可以通过执行如下命令实现。

```
gaussdb=# CREATE TABLE tpcds.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;
CREATE TABLE
gaussdb=# INSERT INTO tpcds.web_returns_p2 SELECT * FROM tpcds.customer_address;
INSERT 0 0
```

- 修改分区表行迁移属性

```
gaussdb=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
ALTER TABLE
```

- 删除分区

删除分区P8。

```
gaussdb=# ALTER TABLE tpcds.web_returns_p2 DROP PARTITION P8;
ALTER TABLE
```

- 增加分区

增加分区P8，范围为 40000<=P8<MAXVALUE。

```
gaussdb=# ALTER TABLE tpcds.web_returns_p2 ADD PARTITION P8 VALUES LESS THAN (MAXVALUE);
ALTER TABLE
```

- 重命名分区

- 重命名分区P8为P_9。

```
gaussdb=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION P8 TO P_9;
ALTER TABLE
```

- 重命名分区P_9为P8。

```
gaussdb=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION FOR (40000) TO P8;  
ALTER TABLE
```

- 查询分区

查询分区P6。

```
gaussdb=# SELECT * FROM tpcds.web_returns_p2 PARTITION (P6);  
gaussdb=# SELECT * FROM tpcds.web_returns_p2 PARTITION FOR (35888);
```

- 删除分区表和表空间

```
gaussdb=# DROP TABLE tpcds.customer_address;  
DROP TABLE  
gaussdb=# DROP TABLE tpcds.web_returns_p2;  
DROP TABLE
```

示例二：使用用户自定义表空间

按照以下方式对范围分区表进行操作。

- 创建表空间

```
gaussdb=# CREATE TABLESPACE example1 RELATIVE LOCATION 'example1';  
gaussdb=# CREATE TABLESPACE example2 RELATIVE LOCATION 'example2';  
gaussdb=# CREATE TABLESPACE example3 RELATIVE LOCATION 'example3';  
gaussdb=# CREATE TABLESPACE example4 RELATIVE LOCATION 'example4';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLESPACE
```

- 创建分区表

```
gaussdb=# CREATE TABLE tpcds.customer_address  
(  
    ca_address_sk integer NOT NULL ,  
    ca_address_id character(16) NOT NULL ,  
    ca_street_number character(10) ,  
    ca_street_name character varying(60) ,  
    ca_street_type character(15) ,  
    ca_suite_number character(10) ,  
    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)  
)  
TABLESPACE example1  
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) TABLESPACE example2  
)  
ENABLE ROW MOVEMENT;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

- 插入数据

将表*tpcds.customer_address*的数据插入到表*tpcds.web_returns_p2*中。

例如在数据库中创建了一个表*tpcds.customer_address*的备份表

tpcds.web_returns_p2，现在需要将表*tpcds.customer_address*中的数据插入到表*tpcds.web_returns_p2*中，则可以执行如下命令。

```
gaussdb=# CREATE TABLE tpcds.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)
)
TABLESPACE example1
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) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
CREATE TABLE
gaussdb=# INSERT INTO tpcds.web_returns_p2 SELECT * FROM tpcds.customer_address;
INSERT 0 0

```

- 修改分区表行迁移属性

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
ALTER TABLE

```

- 删除分区

删除分区P8。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 DROP PARTITION P8;
ALTER TABLE

```

- 增加分区

增加分区P8，范围为 40000<= P8<MAXVALUE。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 ADD PARTITION P8 VALUES LESS THAN (MAXVALUE);
ALTER TABLE

```

- 重命名分区

- 重命名分区P8为P_9。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION P8 TO P_9;
ALTER TABLE

```

- 重命名分区P_9为P8。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION FOR (40000) TO P8;
ALTER TABLE

```

- 修改分区的表空间

- 修改分区P6的表空间为example3。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P6 TABLESPACE example3;
ALTER TABLE

```

- 修改分区P4的表空间为example4。

```

gaussdb=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P4 TABLESPACE example4;
ALTER TABLE

```

- 查询分区

查询分区P6。

```

gaussdb=# SELECT * FROM tpcds.web_returns_p2 PARTITION (P6);
gaussdb=# SELECT * FROM tpcds.web_returns_p2 PARTITION FOR (35888);

```

- 删除分区表和表空间

```
gaussdb=# DROP TABLE tpcds.customer_address;
DROP TABLE
gaussdb=# DROP TABLE tpcds.web_returns_p2;
DROP TABLE
gaussdb=# DROP TABLESPACE example1;
gaussdb=# DROP TABLESPACE example2;
gaussdb=# DROP TABLESPACE example3;
gaussdb=# DROP TABLESPACE example4;
DROP TABLESPACE
```

3.2.6.3 管理索引

背景信息

索引可以提高数据的访问速度，但同时也增加了插入、更新和删除操作的处理时间。因此，在创建索引前需评估是否创建索引，需决策索引字段的选择。需要分析应用程序的业务处理、数据使用、经常被用作查询的条件或者被要求排序的字段来确定是否建立索引。

索引建立在数据库表中的某些列上。因此，在创建索引时，需要评估目标列的选择标准。可以通过以下条件判断是否创建索引，选择创建索引的列。

- 在经常需要搜索查询的列上创建索引，可以加快搜索的速度。
- 在作为主键的列上创建索引，强制要求主键唯一、强制要求主键有序排列。
- 在经常使用连接的列上创建索引，可以加快连接的速度。
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的。
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- 在经常使用WHERE子句的列上创建索引，加快条件的判断速度。
- 为经常出现在关键字ORDER BY、GROUP BY和DISTINCT后面的字段建立索引。

说明

- 索引创建成功后，系统会自动判断何时引用索引。当系统认为使用索引比顺序扫描更快时，就会使用索引。
- 索引创建成功后，必须和表保持同步以保证能够准确地找到新数据，这样就增加了数据操作的负荷。因此请定期删除无用的索引。
- 分区表索引分为LOCAL索引与GLOBAL索引，一个LOCAL索引对应一个具体分区，而GLOBAL索引则对应整个分区表。
- 非透明多写特性下，在开启逻辑复制的场景下，如需创建包含系统列的主键索引，必须将该表的REPLICA IDENTITY属性设置为FULL或是使用USING INDEX指定不包含系统列的、唯一的、非局部的、不可延迟的、仅包括标记为NOT NULL的列的索引。

操作步骤

操作前请先创建分区表，步骤请参见[管理分区表](#)。

- 创建索引
 - 创建分区表索引tpcds_web_returns_p2_index1，不指定索引分区的名称。

```
gaussdb=# CREATE INDEX tpcds_web_returns_p2_index1 ON tpcds.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
```

- 创建分区表索引tpcds_web_returns_p2_index2，并指定索引分区的名称。

```
gaussdb=# CREATE INDEX tpcds_web_returns_p2_index2 ON tpcds.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
```

- 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.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.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.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';
```

- 执行如下命令查询指定索引的信息。

```
gaussdb=# \di+ tpcds.tpcds_web_returns_p2_index2
```

- 删除索引

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

```
gaussdb=# DROP INDEX tpcds.tpcds_web_returns_p2_index2;
```

当结果显示为如下信息，则表示删除成功。

```
DROP INDEX
```

索引介绍

GaussDB支持4种创建索引的方式，请参见[表3-4](#)。

表 3-4 索引方式

索引方式	描述
唯一索引	可用于约束索引属性值的唯一性，或者属性组合值的唯一性。如果一个表声明了唯一约束或者主键，则GaussDB自动在组成主键或唯一约束的字段上创建唯一索引（可能是多字段索引），以实现这些约束。目前，GaussDB只有B-Tree及UBTree可以创建唯一索引。
多字段索引	一个索引可以定义在表中的多个属性上。目前，GaussDB中的B-Tree支持多字段索引。
部分索引	建立在一个表的子集上的索引，这种索引方式只包含满足条件表达式的元组。
表达式索引	索引建立在一个函数或者从表中一个或多个属性计算出来的表达式上。表达式索引只有在查询时使用与创建时相同的表达式才会生效。

- 创建一个普通表

```
gaussdb=# CREATE TABLE tpcds.customer_address_bak AS TABLE tpcds.customer_address;  
INSERT 0 0
```

- 创建普通索引

如果对于tpcds.customer_address_bak表，需要经常进行以下查询。

```
gaussdb=# SELECT ca_address_sk FROM tpcds.customer_address_bak WHERE ca_address_sk=14888;
```

当数据库系统执行tpcds.customer_address_bak表的查询时，若采用全表扫描方式逐行匹配WHERE条件，在表数据量庞大但匹配行极少数（甚至为零）的场景下，该操作将产生显著性能瓶颈。若数据库系统在ca_address_sk属性上维护一个索引，用于快速定位匹配的元组，则数据库系统只需遍历少量索引节点即可获取匹配元组，从而大幅提升查询效率。同样，在数据库中进行更新和删除操作时，索引也可以提升这些操作的性能。

执行如下命令创建索引。

```
gaussdb=# CREATE INDEX index_wr_returned_date_sk ON tpcds.customer_address_bak  
(ca_address_sk);  
CREATE INDEX
```

- 创建唯一索引

在表tpcds.ship_mode_t1中的SM_SHIP_MODE_SK字段上创建唯一索引。

```
gaussdb=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON  
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

- 创建多字段索引

假如用户需要经常查询表tpcds.customer_address_bak中ca_address_sk是5050，且ca_street_number小于1000的记录，使用以下命令进行查询。

```
gaussdb=# SELECT ca_address_sk,ca_address_id FROM tpcds.customer_address_bak WHERE  
ca_address_sk = 5050 AND ca_street_number < 1000;
```

执行如下命令在字段ca_address_sk和ca_street_number上定义一个多字段索引。

```
gaussdb=# CREATE INDEX more_column_index ON  
tpcds.customer_address_bak(ca_address_sk,ca_street_number);  
CREATE INDEX
```

- 创建部分索引

如果只需要查询ca_address_sk为5050的记录，可以创建部分索引来提升查询效率。


```
gaussdb=# CREATE INDEX part_index ON tpcds.customer_address_bak(ca_address_sk) WHERE  
ca_address_sk = 5050;  
CREATE INDEX
```

- 创建表达式索引

假如经常需要查询ca_street_number小于1000的信息，执行如下命令进行查询。
gaussdb=# SELECT * FROM tpcds.customer_address_bak WHERE trunc(ca_street_number) < 1000;

可以为上述的查询创建表达式索引。

```
gaussdb=# CREATE INDEX para_index ON tpcds.customer_address_bak (trunc(ca_street_number));  
CREATE INDEX
```

- 删除tpcds.customer_address_bak表

```
gaussdb=# DROP TABLE tpcds.customer_address_bak;  
DROP TABLE
```

3.2.6.4 管理视图

图 3-4 创建和管理视图

```
gaussdb=# CREATE TABLE mytable (c1 int);  
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.  
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.  
CREATE TABLE  
gaussdb=# CREATE OR REPLACE VIEW myview AS SELECT * FROM mytable WHERE trunc(c1) > 1000;  
CREATE VIEW  
gaussdb=# SELECT * FROM myview;  
  c1  
----  
(0 rows)  
  
gaussdb=# \d+ myview  
View "public.myview"  
Column | Type  | Modifiers | Storage | Description  
-----+-----+-----+-----+-----  
c1      | integer |           | plain   |  
View definition:  
SELECT *  
FROM mytable  
WHERE trunc(mytable.c1::double precision) > 1000::double precision;  
  
gaussdb=# DROP VIEW myview;  
DROP VIEW  
gaussdb=# DROP TABLE mytable;
```

背景信息

当用户需要频繁查询多张表的特定字段组合时，可通过定义视图来避免重复编写查询语句。视图通过虚拟表抽象底层表结构，简化常用查询操作。

视图是一种逻辑表，与物理存储的基本表不同，其数据不实际驻留于磁盘。数据库仅保存视图的查询定义，实际数据始终保留在关联的基础表中。当基础表数据发生变更时，视图查询结果将动态同步这些变化。视图如同一个实时数据窗口，允许用户访问其关注的数据库镜像。每次访问视图时，系统都会重新执行其定义查询以获取最新数据。

说明

视图的优缺点详见基础概念[视图](#)。

管理视图

操作前请先创建表，步骤请参见[创建表](#)。

- 创建视图

执行如下命令创建新视图myview，其中mytable为已经创建的、包含名为c1整型字段的用户表。

```
gaussdb=# CREATE OR REPLACE VIEW myview AS SELECT * FROM mytable WHERE trunc(c1) > 1000;
CREATE VIEW
```

说明

CREATE VIEW中的OR REPLACE子句是可选的。若使用此子句，当视图已存在时，数据库系统将自动替换原有视图定义。

- 查询视图

执行如下命令查询myview视图。

```
gaussdb=# SELECT * FROM myview;
c1
----
(0 rows)
```

- 查看当前用户下的视图

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

- 查看所有视图

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

- 查看某视图的具体信息

执行如下命令查询myview视图的详细信息。

```
gaussdb=# \d+ myview
View "public.myview"
Column | Type   | Modifiers | Storage | Description
-----+-----+-----+-----+-----
c1      | integer |          | plain   |
View definition:
SELECT *
FROM mytable
WHERE trunc(mytable.c1::double precision) > 1000::double precision;
```

- 删除视图

执行如下命令删除myview视图。

```
gaussdb=# DROP VIEW myview;
DROP VIEW
```

3.2.6.5 管理序列

背景信息

序列Sequence是用来产生唯一整数的数据库对象。

声明了NO CYCLE时，序列的值是按照一定规则自增的整数。因为自增所以不重复，因此Sequence具有唯一标识性。这也是Sequence常被用作主键的原因。

通过序列使某字段成为唯一标识符的方法有两种：

- 声明字段的类型为序列整型，由数据库在后台自动创建一个对应的Sequence。

- 使用CREATE SEQUENCE语句创建自定义序列，然后在表字段的默认值中引用nextval('sequence_name')，即可实现自动生成唯一标识符。

注意事项

在非透明多写特性下，新序列值的产生是靠GTM维护的，默认情况下，每申请一个序列值都要向GTM发送一次申请，GTM在当前值的基础上加步长值作为产生的新值返回给调用者。GTM作为全局唯一的节点，势必成为性能的瓶颈，所以对于需要大量频繁产生序列号的操作，如使用Bulkload工具进行数据导入场景，是不建议产生默认序列值的。比如，在下面所示的场景中，INSERT INTO SELECT FROM语句的执行效率显著低于预期。

```
gaussdb=# CREATE SEQUENCE newSeq1;
gaussdb=# CREATE TABLE newT1
(
    id int NOT NULL DEFAULT nextval('newSeq1'),
    name text
);
gaussdb=# INSERT INTO newT1(name) SELECT name FROM T1;
```

可以提高性能的写法是（假设T1表导入newT1表中的数据为10000行）。

```
gaussdb=# INSERT INTO newT1(id, name) SELECT id,name FROM T1;
gaussdb=# SELECT SETVAL('newSeq1',10000);
```

说明

序列操作函数nextval()、setval()等均不支持回滚。另外setval设置的新值，会对当前会话的nextval立即生效，但对其他会话，如果定义了cache，不会立即生效，在用尽所有缓存的值后，其变动才被其他会话感知。所以为了避免产生重复值，要谨慎使用setval，设置的新值不能是已经产生的值或者在缓存中的值。

如果必须要在bulkload场景下产生默认序列值，则一定要为newSeq1定义足够大的cache，并且不要定义Maxvalue或者Minvalue。数据库会试图将nextval('sequence_name')的调用下推到Data Node以提高性能。目前GTM对并发的连接请求是有限制的，当Data Node很多时，将产生大量并发连接，这时一定要控制bulkload的并发数量，避免耗尽GTM的连接资源。如果目标表为复制表(DISTRIBUTE BY REPLICATION)时下推将不能进行。当数据量较大时，除了影响性能，空间也可能会剧烈膨胀，在导入结束后，需要用vacuum full来恢复。推荐采用如上建议，不要在bulkload的场景中产生默认序列值。

另外，序列创建后，在每个节点上都维护了一张单行表，存储序列的定义及当前值，但此当前值并非GTM上的当前值，只是保存本节点与GTM交互后的状态。如果其他节点也向GTM申请了新值，或者调用了Setval修改了序列的状态，不会刷新本节点的单行表，但因每次申请序列值是向GTM申请，所以对序列正确性没有影响。

操作步骤

声明字段类型为序列整型来定义标识符字段。例如：

```
gaussdb=# CREATE TABLE T1
(
    id serial,
    name text
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

创建序列，并通过nextval('sequence_name')函数指定为某一字段的默认值。在非透明多写特性下，这种方式更灵活，可以为序列定义cache，一次预申请多个序列值，减少与GTM的交互次数，来提高性能。

1. 创建序列。

```
gaussdb=# CREATE SEQUENCE seq1 CACHE 100;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE SEQUENCE
```

2. 指定创建的序列为某一字段的默认值，使该字段具有唯一标识属性。

```
gaussdb=# CREATE TABLE T2
(
  id int NOT NULL DEFAULT nextval('seq1'),
  name text
);
```

当结果显示为如下信息，则表示默认值指定成功。

```
CREATE TABLE
```

3. 指定序列与列的归属关系。

将序列和一个表的指定字段进行关联。这样，在删除该字段或其所在表的时候会自动删除已关联的序列。

```
gaussdb=# ALTER SEQUENCE seq1 OWNED BY T2.id;
```

当结果显示为如下信息，则表示指定成功。

```
ALTER SEQUENCE
```

说明

除了为序列指定CACHE，方法二所实现的功能基本与方法一类似。但是一旦定义CACHE，可能会导致序列值出现不连续现象（例如生成1、4、5等），并且不能保序。另外为某序列指定从属列后，从属列删除，对应的SEQUENCE也会被删除。虽然数据库并不限制序列只能为一列产生默认值，但建议不要多列共用同一个序列。

当前版本只支持在定义表的时候指定自增列，或者指定某列的默认值为nextval('seqname')，不支持在已有表中增加自增列或者增加默认值为nextval('seqname')的列。

3.2.6.6 管理定时任务

背景信息

当客户在使用数据库过程中，如果白天执行一些耗时比较长的任务（例如：统计数据汇总之类或从其他数据库同步数据的任务），会对正常的业务有性能影响，所以客户经常选择在晚上执行，无形中增加了客户的工作量。因此GaussDB数据库兼容ORA数据库中定时任务的功能，可以由客户创建定时任务，当任务时间点到达后可以自动触发任务的执行，从而可以减少客户运维的工作量。

GaussDB数据库兼容ORA定时任务功能主要通过DBE_SCHEDULER和DBE_TASK高级包提供的接口，可以实现定时任务的创建、任务到期自动执行、任务删除和修改任务属性（包括：任务id、任务的关闭开启、任务的触发时间、触发时间间隔和任务内容等）。推荐使用DBE_SCHEDULER接口，保证高可用性、高可靠性，并且支持更灵活的任务调度。接口说明、迁移指导示例请参见《参考》中“存储过程 > 高级包 > 二次封装接口（推荐）> DBE_SCHEDULER”章节。

注意事项

非透明多写特性下：

- PG_JOB系统表中next_run_date字段标识定时任务实际开始执行的时间，而不是计划任务开始的时间。
- 当前定时任务计算开始执行的时间依赖于上一次任务的实际执行时间，在不指定freq时会导致执行时间有较短时间的偏差。

```
gaussdb=# SELECT start_date, next_run_date, interval FROM pg_job WHERE job_name = 'job1';
 start_date      | next_run_date      | interval
-----+-----+-----
2024-12-03 15:24:11.94422 | 2024-12-03 15:26:12.197623 | interval'1 minute'
(1 row)
```

定时任务管理

步骤1 创建测试表。

```
gaussdb=# CREATE TABLE test(id int, time date);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

步骤2 创建自定义存储过程。

```
gaussdb=# CREATE OR REPLACE PROCEDURE PRC_JOB_1 ( ) AS
  N_NUM INTEGER := 1;
BEGIN
  FOR I IN 1..1000
  LOOP
    INSERT INTO test VALUES(I,SYSDATE);
  END LOOP;
END;
/
```

当结果显示为如下信息，则表示创建成功。

```
CREATE PROCEDURE
```

步骤3 创建任务。

- 新创建的任务（未指定job_id）表示每隔1分钟执行一次存储过程PRC_JOB_1。

```
gaussdb=# CALL db_task.submit('CALL public.prc_job_1();', sysdate, 'interval "1 minute"', :a);
id
----
1
(1 row)
```

- 指定job_id创建任务，其中job_id可用范围为1~32767。

```
gaussdb=# CALL db_task.id_submit(1,'CALL public.prc_job_1();', sysdate, 'interval "1 minute"');
id_submit
-----
1
(1 row)
```

步骤4 通过视图查看当前用户已创建的任务信息。

```
gaussdb=# SELECT job,dbname,start_date,last_date,this_date,next_date,broken,status,interval,failures,what
FROM my_jobs;
 job | dbname | start_date      | last_date      | this_date      | next_date      | broken | status | interval      | failures | what
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
1 | testdb | 2017-07-18 11:38:03 | 2017-07-18 13:53:03.607838 | 2017-07-18 13:53:03.607838 | 2017-07-18 13:54:03 | n      | s      | interval '1 minute' | 0 | call public.prc_job_1();
(1 row)
```

步骤5 停止任务。

```
gaussdb=# CALL db_task.finish(1,true);
finish
-----
```

```
(1 row)
```

步骤6 启动任务。

```
gaussdb=# CALL db_task.finish(1,false);
finish
-----
```

```
(1 row)
```

步骤7 修改任务属性。

- 修改JOB的next_time参数信息。

--修改Job1的next_time为1小时以后开始执行。

```
gaussdb=# CALL db_task.next_time(1, sysdate+1.0/24);
next_time
-----
```

```
(1 row)
```

- 修改JOB的interval参数信息。

--修改Job1的interval为每隔1小时执行一次。

```
gaussdb=# CALL db_task.interval(1,'sysdate + 1.0/24');
interval
-----
```

```
(1 row)
```

- 修改JOB的what参数信息。

--修改Job1的what为执行SQL语句“INSERT INTO public.test VALUES(333, sysdate+5);”。

```
gaussdb=# CALL db_task.content(1,'INSERT INTO public.test VALUES(333, sysdate+5);');
content
-----
```

```
(1 row)
```

- 同时修改JOB的next_date、interval、what等多个参数信息。

```
gaussdb=# CALL db_task.update(1, 'CALL public.prc_job_1();', sysdate, 'interval "1 minute"');
update
-----
```

```
(1 row)
```

步骤8 查看JOB执行情况。

当JOB自动执行时，如果JOB执行失败（即job_status状态值为'f'）时，请联系管理员查看gs_log的运行日志来查看JOB的失败信息。

日志信息如下所示，从失败信息（detail error msg）中可以查看失败的具体错误。

```
LOG: Execute Job Detail:
     job_id: 1
    what: call public.test();
 start_date: 2017-07-19 23:30:47.401818
 job_status: failed
detail error msg: relation "test" does not exist
 end_date: 2017-07-19 23:30:47.401818
next_run_date: 2017-07-19 23:30:56.855827
```

步骤9 删除JOB。

```
gaussdb=# CALL db_task.cancel(1);
cancel
-----
```

```
(1 row)
```

----结束

JOB 的权限控制

- 当创建一个JOB时，该JOB会和创建该JOB的数据库和用户绑定（即：pg_job系统表新增的JOB记录中的dbname和log_user）。
- DBA用户、系统管理员或该JOB的创建用户，可通过高级包接口remove、change、next_data、what、interval删除或修改JOB的参数信息。否则，会提示当前用户没有权限操作该JOB。
- 如果当前数据库是该JOB创建所属的数据库，那么连接到当前数据库上可以通过高级包接口cancel、update、next_data、content、interval删除或修改JOB的参数信息。
- 当删除JOB所属的数据库时，系统会关联删除该数据库从属的JOB记录。
- 当删除JOB所属的用户时，系统会关联删除该用户从属的JOB记录。

JOB 的并发控制管理

非透明多写特性下，用户可以通过配置GUC参数job_queue_processes调整并发同时执行的JOB数目。

- 当job_queue_processes为0时，表示不启用定时任务功能，任何JOB都不会被执行。
- 当job_queue_processes为大于0时，表示启用定时任务功能且系统能够并发处理的最大任务数。

由于并行运行的任务数太多会消耗更多的系统资源，因此需要设置系统并发处理的任务数，当前并发的任务数达到job_queue_processes时，且此时又有任务到期，那么这些任务本次得不到执行而延期到下一轮询周期。因此，建议用户需要根据每个任务的执行时长合理地设置任务的时间间隔（即submit接口中的interval参数），来避免由于任务执行时间太长而导致下个轮询周期无法正常执行。

注：对于不使用JOB的集群中，用户可以通过在集群安装初始化完成后，通过设置job_queue_processes为0来关闭JOB功能，减少系统资源的消耗。

出于定时任务的实时性考虑，对所有任务添加了缓存。如果执行时出现以下报错，原因是定时任务的后台线程和业务线程同时更新了同一个任务。此时可以通过重试定时任务相关接口以解决该问题。

ERROR: tuple concurrently updated

3.3 管理权限

设置或更改数据库用户或角色权限，可以使用DCL（Data Control Language，数据控制语言）语句。

背景信息

数据库对象创建后，创建该对象的用户即成为其所有者。数据库安装后，默认未开启三权分立，数据库系统管理员具有与对象所有者相同的权限。也就是说对象创建后，默认只有对象所有者或者系统管理员可以查询、修改和删除对象，以及通过GRANT将对象的权限授予其他用户。

为使其他用户能够使用对象，也可以向用户或包含该用户的角色授予必要的权限。

注意事项

不建议用户修改系统表和系统视图的权限。

管理对象权限

步骤1 以管理员用户root登录数据库。

步骤2 创建用户user1和用户user2。

```
gaussdb=# CREATE USER user1 password '*****';
gaussdb=# CREATE USER user2 password '*****';
```

步骤3 以用户user2登录数据库。

步骤4 创建用户表tbl，将用户user2的USAGE权限授予用户user1，将对表tbl进行SELECT的权限授予用户user1。

```
gaussdb=> CREATE TABLE user2.tbl(id int, name varchar(5));
CREATE TABLE
gaussdb=> GRANT USAGE ON SCHEMA user2 TO user1;
GRANT
gaussdb=> GRANT SELECT ON TABLE user2.tbl TO user1;
GRANT
```

步骤5 以用户user1登录数据库。

步骤6 用户user1查询用户表tbl成功，插入用户表tbl权限不足失败。

```
gaussdb=> SELECT * FROM user2.tbl;
 id | name
----+-----
(0 rows)

gaussdb=> INSERT INTO user2.tbl VALUES (1,'TOM');
ERROR: The operation permission on relation tbl is denied.
```

步骤7 以用户user2登录数据库。

步骤8 撤销用户user1对表tbl进行SELECT的权限，将对表tbl进行INSERT的权限授予用户user1。

```
gaussdb=> REVOKE SELECT ON TABLE user2.tbl FROM user1;
REVOKE
gaussdb=> GRANT INSERT ON TABLE user2.tbl TO user1;
GRANT
```

步骤9 以用户user1登录数据库。

步骤10 查询用户表tbl权限不足失败，插入用户表tbl成功。

```
gaussdb=> SELECT * FROM user2.tbl;
ERROR: The operation permission on relation tbl is denied.
gaussdb=> INSERT INTO user2.tbl VALUES (1,'TOM');
INSERT 0 1
```

步骤11 以管理员用户root登录数据库。

步骤12 清理环境。

```
gaussdb=# DROP USER user1, user2 CASCADE;
DROP ROLE
```

----结束

管理用户和角色

步骤1 以管理员用户root登录数据库。

步骤2 创建角色role1和角色role2。

```
gaussdb=# CREATE ROLE role1 WITH password 'xxxxxxx';
CREATE ROLE
```



```
gaussdb=# CREATE ROLE role2 WITH password 'xxxxxxx';  
CREATE ROLE
```

步骤3 修改角色role1具有CREATEDB和LOGIN属性，角色role2为默认权限。

```
gaussdb=# ALTER ROLE role1 WITH CREATEDB;  
ALTER ROLE  
gaussdb=# ALTER ROLE role1 WITH LOGIN;  
ALTER ROLE
```

步骤4 查看系统视图pg_roles获取角色的相关信息。

```
gaussdb=# SELECT oid, rolname, rolcreatedb, rolcanlogin FROM pg_roles WHERE rolname='role1' or  
rolname='role2';  
 oid | rolname | rolcreatedb | rolcanlogin  
-----+-----+-----+-----  
16750 | role1   | t           | t  
16778 | role2   | f           | f  
(2 rows)
```

步骤5 以用户role1登录数据库。

步骤6 通过以下命令创建数据库test_db，显示CREATE DATABASE表示创建成功。

```
gaussdb=# CREATE DATABASE test_db;  
CREATE PLUGGABLE DATABASE
```

步骤7 将角色role1权限授予角色role2，则角色role2继承role1的相关对象权限。

```
gaussdb=# GRANT role1 TO role2;  
GRANT ROLE
```

步骤8 查看系统表pg_auth_members获取角色成员关系。

```
gaussdb=# SELECT * FROM pg_auth_members;  
roleid | member | grantor | admin_option  
-----+-----+-----+-----  
16750 | 16778 | 16750 | f  
(1 row)
```

步骤9 以管理员用户root登录数据库。

步骤10 关闭并删除数据库test_db，删除角色role1和role2。

```
gaussdb=# ALTER DATABASE test_db CLOSE;  
ALTER PLUGGABLE DATABASE  
gaussdb=# DROP DATABASE test_db;  
DROP PLUGGABLE DATABASE  
gaussdb=# DROP ROLE role1,role2;  
DROP ROLE
```

----结束

3.4 使用存储过程

存储过程

商业规则和业务逻辑可以通过程序存储在GaussDB数据库中，这个程序就是存储过程。

存储过程是SQL和PL/SQL的组合。存储过程可以将执行商业规则的代码从应用程序中移动到数据库，从而实现代码的一次存储，供多个程序使用。

详细使用方法请参见《参考》中“存储过程”章节。

PACKAGE

PACKAGE是一组相关存储过程、函数、变量、常量和游标等PL/SQL程序的组合，具有面向对象的特点，可以对PL/SQL程序设计元素进行封装。PACKAGE中的函数具有统一性，创建、删除、修改都统一进行。

PACKAGE包含包头（Package Specification）和PACKAGE BODY两个部分，其中包头所包含的声明可以被外部函数、匿名块等访问；而在包体中包含的声明不能被外部函数、匿名块等访问，只能被包体内函数和存储过程等访问。

PL/SQL 标识符

PL/SQL子程序、PACKAGE、参数、变量、异常、游标都会有一个名称，即PL/SQL标识符。

标识符的命名需要遵守如下规范：

- 标识符需要为小写字母（a-z）、大写字母（A-Z）、下划线（_）、数字（0-9）或美元符号（\$）。
- 标识符必须以字母或下划线开头。
- 标识符的最小长度为1个字符，最大长度为63个字符。

PL/SQL 数据类型

数据类型是一组值的集合以及定义在这个值集上的一组操作。GaussDB数据库是由表的集合组成的，而各表中的列定义了该表，每一列都属于一种数据类型，GaussDB数据库根据数据类型有相应函数对其内容进行操作，例如：GaussDB数据库可对数值型数据进行加、减、乘、除等操作。

PL/SQL中使用SUBTYPE可以创建自己的子类型，其基类型可以是任何基础类型或用户自定义类型。

PL/SQL 基本结构

PL/SQL块中可以包含子块，子块可以位于PL/SQL中任何部分。PL/SQL块的结构如下：

- 声明部分：声明PL/SQL用到的变量、类型、游标、局部的存储过程和函数。
DECLARE

说明

不涉及变量声明时声明部分可以没有。

- 对匿名块来说，没有变量声明部分时，可以省去DECLARE关键字。
- 对存储过程来说，没有DECLARE关键字，AS关键字相当于DECLARE。即使没有变量声明的部分，AS关键字也必须保留。
- 执行部分：过程及SQL语句，程序的主要部分。必选。
BEGIN
- 执行异常部分：错误处理。可选。
EXCEPTION
- 结束。必选。
END;
/

须知

- 禁止在PL/SQL块中使用连续的Tab，连续的Tab可能会在使用gsql工具并带有“-r”参数执行PL/SQL块时导致异常。
- 只有最顶层匿名块需要/结束符。

PL/SQL块可以分为以下几类：

- 匿名块：只执行一次，在执行时编写，结果不会存储在数据库中。
- 程序：存储在数据库中的存储过程、函数、操作符和高级包等。当在数据库上建立好后，可以在其他程序中调用它们。

PL/SQL 基本语句

- 变量声明

变量通常在DECLARE部分声明，每个变量都需要指定类型。可以在声明变量时给变量赋初始值。

```
DECLARE
id    INTEGER; --声明变量
name  VARCHAR(20); --声明变量
age   INT := 20; --声明变量并赋值
weight NUMBER(5,2) := 60.5; --声明变量并赋值
address VARCHAR(20) := 'xian'; --声明变量并赋值
BEGIN
END;
/
ANONYMOUS BLOCK EXECUTE
```

- 赋值语句

变量赋值通常在BEGIN...END块中，使用:=或:=进行变量赋值。

```
DECLARE
id    INTEGER; --声明变量
name  VARCHAR(20); --声明变量
BEGIN
id := 1; --变量赋值
name := 'xiaohua'; --变量赋值
END;
/
ANONYMOUS BLOCK EXECUTE
```

PL/SQL 动态语句

动态语句允许程序执行期间动态的构造并执行SQL语句或PL/SQL块。

```
gaussdb=# CREATE SCHEMA use_procedure;
CREATE SCHEMA
gaussdb=# CREATE TABLE use_procedure.tb1 (a int);
CREATE TABLE
gaussdb=# INSERT INTO use_procedure.tb1 VALUES (generate_series(1, 10));
INSERT 0 10
gaussdb=# DECLARE
counts int;
BEGIN
EXECUTE IMMEDIATE 'select count(*) from use_procedure.tb1' INTO counts;
dbe_output.print_line(counts);
END;
/
10
ANONYMOUS BLOCK EXECUTE
gaussdb=# DROP SCHEMA use_procedure CASCADE;
```

```
NOTICE: drop cascades to table use_procedure.tb1
DROP SCHEMA
```

PL/SQL 控制语句

条件语句的主要作用判断参数或者语句是否满足已给定的条件，根据判定结果执行相应的操作。

```
gaussdb=# DECLARE
count int := 0;
BEGIN
LOOP
  IF count > 10 THEN
    EXIT;
  ELSE
    count := count + 1;
  END IF;
END LOOP;
END;
/
ANONYMOUS BLOCK EXECUTE
```

PL/SQL 事务

调用存储过程时会自动开启一个事务，在调用结束时自动提交或者发生异常时回滚。除了系统自动的事务控制外，也可以使用COMMIT/ROLLBACK来控制存储过程中的事务。在存储过程中调用COMMIT/ROLLBACK命令，将提交/回滚当前事务并自动开启一个新的事务，后续的所有操作都会在此新事务中运行。

```
gaussdb=# CREATE SCHEMA use_procedure;
CREATE SCHEMA
gaussdb=# CREATE TABLE use_procedure.tb1 (a int);
CREATE TABLE
gaussdb=# DECLARE
BEGIN
FOR i IN 1..10 LOOP
  INSERT INTO use_procedure.tb1 VALUES (i);
  IF i % 2 = 0 THEN
    COMMIT;
  ELSE
    ROLLBACK;
  END IF;
END LOOP;
END;
/
ANONYMOUS BLOCK EXECUTE
gaussdb=# DROP SCHEMA use_procedure CASCADE;
NOTICE: drop cascades to table use_procedure.tb1
DROP SCHEMA
```

PL/SQL 游标

为了处理SQL语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

```
gaussdb=# CREATE SCHEMA use_procedure;
CREATE SCHEMA
gaussdb=# CREATE TABLE use_procedure.tb1 (a int);
CREATE TABLE
gaussdb=# INSERT INTO use_procedure.tb1 VALUES (generate_series(1, 10));
INSERT 0 10
gaussdb=# DECLARE
CURSOR my_cur is SELECT a FROM use_procedure.tb1;
tb_a int;
BEGIN
```

```
OPEN my_cur;  
LOOP  
  FETCH my_cur into tb_a;  
  EXIT WHEN my_cur%NOTFOUND;  
END LOOP;  
CLOSE my_cur;  
END;  
/  
ANONYMOUS BLOCK EXECUTE  
gaussdb=# DROP SCHEMA use_procedure CASCADE;  
NOTICE: drop cascades to table use_procedure.tb1  
DROP SCHEMA
```

管理存储过程

- **创建存储过程**
gaussdb=# CREATE OR REPLACE PROCEDURE proc1() AS
DECLARE
id INT;
name VARCHAR(20);
BEGIN
id := 1;
name := 'xian';
END;
/
CREATE PROCEDURE
- **更改存储过程**
--修改存储过程名称
gaussdb=# ALTER PROCEDURE proc1() RENAME TO proc2;
ALTER FUNCTION
--用存储过程名重编译存储过程
gaussdb=# ALTER PROCEDURE proc2() COMPILE;
ALTER PROCEDURE
- **使用存储过程**
--直接调用
gaussdb=# CALL proc2();
proc2

(1 row)

--使用匿名块调用
gaussdb=# BEGIN
proc2();
END;
/
ANONYMOUS BLOCK EXECUTE
- **删除存储过程**
gaussdb=# DROP PROCEDURE proc2;
DROP PROCEDURE

4 开发设计建议

4.1 概述

简介

本规范以产品生命周期为主线，详细描述产品设计和开发流程过程中与数据库相关的设计和开发规范。

规范以提高可读性、代码质量为原则，强调实用性、可操作性，对数据库开发容易产生问题的地方做出了明确的规定。主要包括下列内容：

设计规范包括：数据库设计、性能设计。

编程规范包括：排版、命名、注释、语法、脚本、数据库编程、安装部署和安全规范。

同时，在必要时，对部分规范给出细则及具体的范例。

该开发规范仅作为参考，具体规范应结合应用技术架构及规划，由数据库使用者综合评估并实施。

术语约定

本规范采用以下的术语描述：

- **规格**：数据库规格，编程、设计必须遵守的原则，否则数据库将报错。
- **规则**：编程、设计强制遵守的原则。
- **建议**：编程、设计必须加以考虑的原则。
- **说明**：对规则或建议进行必要的解释。
- **示例**：对规则或建议进行正或反方面的举例。

适用范围

本规范适合人群包括：设计人员、开发人员、开发DBA、运维DBA、运维人员。

4.2 数据库设计规范

4.2.1 基本规范

- 【规则】数据库价值特性。

表 4-1 数据库价值特性推荐

特性分类	特性列表	说明
表类型	DISTRIBUTION表	数据分片。
	PARTITION表	数据分区。
存储引擎	行存储	按行顺序存储表，建议点查，增删改操作较多场景下使用。
事务	事务块	显式启动事务。
	单语句事务	不显式启动事务，单语句即为事务。
	分布式事务（弱一致）	GTM FREE模式。该模式不支持分布式事务的强一致性读取，但在SHARDING场景下可保证强一致。
部署	一主多备	支持DN分片，一个主DN，多个备机DN，保证数据备份和容灾。
安全	透明加密	支持数据表级别存储加密，上层业务无感知。
数据类型	整数类型	TINYINT、SMALLINT、INTEGER、BIGINT。
	任意精度类型	NUMERIC/DECIMAL。
	浮点类型	REAL/FLOAT4、DOUBLE PRECISION/FLOAT8、FLOAT。
	布尔类型	BOOLEAN。
	定长字符	CHAR(n)。
	变长字符	VARCHAR(n)、NVARCHAR2(n)、TEXT。
	时间类型	DATE、TIME、TIMETZ、TIMESTAMP、TIMESTAMPPTZ、SMALLDATETIME、INTERVAL、RELTIME、DATEA（仅在ORA兼容性生效）。
	二进制类型	BYTEA（变长二进制类型）。
	位串类型	BIT(n)、VARBIT(n)。
函数	字符处理函数	字符数据类型处理函数。
	二进制字符串函数	二进制字符类型处理函数。

特性分类	特性列表	说明
	数字操作函数	数值类型处理函数。
	时间和日期处理函数	时间和日期类型处理函数。
索引	主键/唯一索引	单列或多列主键/唯一索引。
	BTREE索引	索引类型。
扩容	在线扩容	扩节点和数据重分布过程中用户业务平滑过渡。

- 【规则】数据库使用最佳实践推荐。

表 4-2 数据库使用最佳实践推荐列表

序号	条目	建议值
1	集群最佳分片数（主DN数）	<256。
2	集群最佳长连接数	请参考对应硬件规格下GUC参数max_connections的默认配置。
3	单物理节点数据量最大值	16TB（具体容量请结合备份恢复规格目标确定）。
4	集群数据库活跃个数	1。
5	集群表和索引总个数	10000（单个schema下的表数量不建议超过200）。 说明 该值仅为建议值，实际需要根据客户的磁盘容量和业务来规划，理论最大值为 2^{32} 。
6	单表最佳字段个数	<50。
7	单表最佳索引个数	<5。
8	单表最佳复合索引个数	<3。
9	单复合索引包含最佳列数	<5。
10	单行最佳行宽	<2k。
11	单个字段最佳容量	• 非透明多写特性下：<10MB。
12	SQL语句最佳长度	<5k。
13	磁盘可用空间比例设置	• 非透明多写特性下：50%。

4.2.2 部署规范

【规则】安装数据库后，DBA应根据环境特性，合理配置数据库GUC参数。

GaussDB提供了多种修改GUC参数的方法，用户可以方便地针对数据库、用户、会话进行设置。

GUC参数配置，请联系管理员处理。

4.2.3 数据库对象命名规范

- 【规格】不使用保留或非保留关键字命名数据库对象。
可以使用如下命令查询数据库关键字，或者在《参考》中“SQL参考 > 关键字”章节中查看。

```
SELECT * FROM pg_get_keywords();
```
- 【规格】数据库对象命名长度不可超过63个字节。

表 4-3 数据库对象名称长度约束

对象类型	长度约束 (单位：字节)	备注
数据库名	<=63	-
表名	<=63	非透明多写特性下，若需要对建有全局二级索引（GSI）的表执行COPY、GDS数据导入操作，则表名长度不超过38个字节。如果超过该长度，COPY、GDS导入过程中创建的临时表，可能触发表名截断，进而引发数据导入流程中断，出现预期外的字符等问题。
表对象	<=63	如果超过该长度内核会对表名进行截断，从而造成实际名称和设置值不一致的现象，且在不同字符集下，可能造成字符被截断，出现预期外的字符。
非业务使用的临时表	<=63	例如：运营人员临时用作备份或临时进行数据采集用的中间表。 命名规则：tmp_表名缩写_创建人账号缩写_创建日期， 例如：tmp_user_ytw_160505。
表字段	<=63	字段命名使用实际含义的英文单词或简写。 例如表示bool类型的字段，命名规则：“is_”+描述。 如member表上表示为enabled的会员的列命名为is_enabled。
视图名及字段名	<=63	-
函数名及参数名	<=63	-

- 【规则】避免使用双引号括起来的字符串来定义数据库对象名称，除非必须限制数据库对象名称的大小写。
 - GaussDB默认不区分SQL中对象名称的大小写，假如同一数据库中同时存在“t_Table”“t_table”两张不同的表，这种情况下应使用双引号（""），如果不存在该情况，避免使用双引号。
 - 数据库对象名称大小写敏感会使定位问题难度增加。
- 【规则】标识符需满足以下要求：
 - 以字母或下划线开头，中间字符可以是字母、数字、下划线、\$、#。
 - 若标识符被双引号（""）包含或者MYSQL模式下被反引号（`）包含，则可以使用合法字符的任意组合，如"123gs_column"。
 - 标识符不区分大小写，只有被双引号包含或者MYSQL模式下被反引号（`）包含才区分大小写。
- 【规则】不支持gsq快捷命令（除\sf）查询反引号包含的对象名。
- 【建议】数据库对象命名风格务必保持统一。
 - 增量开发的业务系统或进行业务迁移的系统，建议遵守历史的命名风格。
 - 建议使用多个单词组成，以下划线分割。
 - 数据库对象名称建议能够望文知意，尽量避免使用自定义缩写（可以使用通用的术语缩写）进行命名。例如，在命名中可以使用具有实际业务含义的英文词汇或汉语拼音，但规则应该在集群范围内保持一致。
 - 变量名的关键是要具有描述性，即变量名要有一定的意义，变量名要有前缀标明该变量的类型。
- 【建议】表对象的命名应该可以表征该表的重要特征。例如，在表对象命名时区分该表是普通表、临时表还是非日志表：
 - 普通表名按照数据集的业务含义命名。
 - 临时表以“tmp_+后缀”命名。
 - 非日志表以“ul_+后缀”命名。
 - 外表以“f_+后缀”命名。
 - 不创建以redis_为前缀的数据库对象。
 - 不创建以pgxc_redistb为命名的数据库对象。
 - 不创建以mlog_和以matviewmap_为前缀的数据库对象。
 - 不创建以gs_role_为前缀的数据库对象。

4.2.4 Database 和 Schema 设计规范

- 【说明】创建Database时，需要重点关注字符集编码（ENCODING）和兼容性（DBCOMPATIBILITY）两个配置项。GaussDB支持TD、ORA、MYSQL、PG、M五种兼容模式，分别表示兼容Teradata语法、Oracle语法、MySQL语法、POSTGRES语法和M-Compatibility语法，不同兼容模式下的语法行为存在一定差异，默认为MYSQL兼容模式。
- 【说明】Database的owner默认拥有该Database下所有对象的所有权限，包括删除权限。删除权限影响较大，请谨慎使用。
- 【说明】如果该用户不具有sysadmin权限或者不是该Schema的owner，要访问Schema下的对象，需要同时给用户授予Schema的usage权限和对象的相应权限。
- 【说明】如果要在Schema下创建对象，需要授予操作用户该Schema的CREATE权限。

- 【说明】Schema的owner默认拥有该Schema下对象的所有权限，包括删除权限。删除权限影响较大，请谨慎使用。
- 【规格】使用JDBC客户端连接数据库时必须指明数据库名。具体格式为：
`jdbc:gaussdb://ip:port/database_name`

须知

JDBC实例一旦创建，无法进行数据库切换。

- 【规格】只在MYSQL（即`sql_compatibility = 'MYSQL'`）和M（即`sql_compatibility = 'M'`）兼容模式下的数据库支持不区分大小写的排序方式，其它数据库目前不支持不区分大小写的排序方式。
- 【规则】业务使用前必须由系统管理员为业务创建DATABASE、SCHEMA和USER，然后再赋予相关用户对应对象的权限。
- 【规则】业务使用前必须先创建业务数据库。禁止使用数据库安装后默认创建的postgres数据库存储业务数据，建议根据业务情况创建业务自己的数据库。
- 【规则】创建数据库时必须指定字符集为UTF8，创建数据库时必须选择与客户端统一的编码字符集。

为了满足全球化需求，数据库编码应能够存储与标识绝大多数的字符，因此推荐使用UTF8。GaussDB中的UTF8字符集与MySQL的UTF8MB4等价，能够支持emoji表情字符。

如果客户端的编码方式与数据库的编码方式不统一，需要转码，会带来性能问题，同时，针对同编码的内核优化无法触发，影响查询效率。

客户端的编码字符集需通过以下方式修改：

- 设置客户端连接参数，例如JDBC连接参数可通过在URL中追加`characterEncoding`和`allowEncodingChanges`参数。
`jdbc:gaussdb://ip:port/database_name?characterEncoding=utf8&allowEncodingChanges=true`
- 修改数据库GUC参数。
`SET client_encoding = 'UTF8';`

数据库的编码在CREATE DATABASE时进行设置。

```
CREATE DATABASE tester WITH ENCODING = 'UTF8';
```

须知

数据库一旦创建无法更改字符集。

- 【建议】一个集群内，用户自定义的Database数量推荐值为3个，不建议超过10个。用户自定义的Database数量过多会导致升级、备份等运维操作的效率降低。
- 【建议】实际用户环境中Schema数量不建议超过100个。当数据库中存在大量Schema时，会导致`gs_dump`等依赖Schema数量的操作性能变慢。
- 【建议】从便捷性和资源共享效率上考虑，建议使用SCHEMA进行业务隔离。数据库可以使用DATABASE和SCHEMA两种方式实现业务的隔离。区别在于：
 - DATABASE方式隔离的共享资源极少，可实现连接隔离、权限隔离等，隔离更加彻底。但DATABASE之间无法互相访问，JDBC建立连接时必须指明DATABASE，连接后无法切换DATABASE。
 - SCHEMA方式隔离的共用资源较多，可以通过GRANT与REVOKE语法便捷地控制不同用户对各SCHEMA及其下属对象的权限。

- 【建议】创建数据库时建议指定LC_CTYPE和LC_COLLATE，该参数将影响数据的排序顺序。

示例：

```
CREATE DATABASE SAMPLE_DB WITH LC_CTYPE = 'zh_CN.gbkl' LC_COLLATE = 'zh_CN.gbkl';
```

4.2.5 权限设计规范

- 【规则】数据库初始用户连接和访问数据库应作为DBA管理用途，不允许业务直接使用该数据库用户连接和访问数据库。

说明

数据库初始用户，即数据库的安装用户，与集群所属的操作系统用户同名。

- 【规则】数据库初始用户为业务创建用户和数据库，业务使用所创建的用户登录和访问数据库。
- 【规则】合理对角色和用户赋权，应使用最小化权限原则。
- 【建议】通过角色来管理权限。

使用角色管理权限，即在角色中配置权限，再将角色赋予用户。通过角色管理权限，更便于多用户、用户变更等场景下的权限管理。例如：

- 角色和用户为多对多关系，一个角色可以赋予多个用户，修改角色中的权限，被赋予角色的用户权限就可以同时更新。
- 删除用户时，不会影响到角色。
- 新建用户后，可以通过赋予角色快速获取所需权限。

- 【建议】在删除指定数据库时，应回收用户对该数据库的CONNECT权限，避免删除时仍然存在活跃的数据库连接而失败。
- 【建议】避免因权限使用不当导致权限被利用的风险。

部分场景下，用户操作不当可能会产生权限被利用的风险。例如：

- 当用户使用关联函数的方式创建非系统自带类型时，需要了解该类型定义及该类型所关联的函数。如果使用不当，可能会因为所关联的函数而产生权限被利用的风险。
- 通过GRANT授予用户使用表的权限时，如果用户使用不当，可能通过ALTER在表的默认值、约束增加表达式，通过创建索引在INDEX上增加表达式等操作，产生权限被利用的风险。
- 通过GRANT授予用户使用TRIGGER的权限时，如果用户使用不当，可能通过WHEN条件创建表达式，当触发器被触发时，存在权限被利用的风险。

4.2.6 字符集设计规范

- 【规格】目前除MYSQL或M模式数据库（即sql_compatibility = 'MYSQL'或'M'）外，仅支持对数据库定义字符集，不支持对表、字段等其他对象定义字符集。

在MYSQL或M模式数据库（即sql_compatibility = 'MYSQL'或'M'）中，表、字段等其他对象的字符集需要同数据库的字符集一致，否则会报错。

- 【建议】数据库和客户端均使用统一的UTF8字符集。
 - 客户端包括ODBC、JDBC、gsq等。
 - 推荐UTF8字符集原因见[Database和Schema设计规范](#)。

4.2.7 表设计规范

- 【规则】视图不建议嵌套。
 - 如果视图编写时使用了通配符，当被调用的视图新增或删除列时，视图将发生错误。
 - 视图嵌套可能因无法使用索引而导致执行效率低下，尽量使用带有索引的基表而不是视图上做关联操作。
- 【建议】视图定义中尽量避免排序操作。

ORDER BY子句在顶层视图上无效，如果必须对输出数据排序，请考虑在调用视图中使用ORDER BY。
- 【说明】非透明多写特性下，将表的扫描压力均匀分散在各个DN上。避免扫描压力集中在部分DN上，而导致性能瓶颈。例如，在事实表上使用等值过滤条件时，将会导致扫描压力不均匀。
- 【建议】尽量减少随机I/O。通过聚簇可以实现热数据的连续存储，将随机I/O转换为连续I/O，从而减少扫描的I/O代价。
- 【建议】非透明多写特性下，尽量避免数据shuffle。shuffle是指在物理上，数据从一个节点传输到另一个节点。shuffle占用了大量宝贵的网络资源，减少不必要的数据shuffle，可以减少网络压力，使数据的处理本地化，以提高集群的性能和可支持的并发度。通过对关联条件和分组条件的仔细设计，能够尽可能的减少不必要的数据shuffle。

选择分布方案

表的分布方式的选择如表4-4所示。

表 4-4 表的分布方式及使用场景

分布方式	描述	适用场景
Hash	表数据通过Hash方式散列到集群中的所有DN上。	数据量较大的事实表。
Replication	集群中每一个DN都有一份全量表数据。	维度表、数据量较小的事实表。
Range	表数据对指定列按照范围进行映射，分布到对应DN。	用户需要自定义分布规则的场景。
List	表数据对指定列按照具体值进行映射，分布到对应DN。	用户需要自定义分布规则的场景。

- 【说明】将表数据均匀分布在各个DN上。数据均匀分布，可以防止数据在部分DN上集中分布，从而导致因存储倾斜造成集群有效容量下降。通过选择合适的分布列，可以避免数据倾斜。
- 【规则】必须指定表分布（DISTRIBUTE BY），且表分布策略选择应符合以下原则。
 - 对于系统配置表、数据字典表等数据规模小于2000w且插入更新十分低频的表，要求采用REPLICATION分布。

```
CREATE TABLE t1 (contentId INT) DISTRIBUTE BY REPLICATION;
```

须知

慎用REPLICATION分布，该分布表会造成空间膨胀、DML性能下降等负面影响。

- 对于不满足条件1的表，即数据量较大，更新频率较高的表，必须进行数据分片，要求采用HASH分布策略，分布键必须是主键中的一个或多个字段。
CREATE TABLE t1 (contentId INT) DISTRIBUTE BY HASH (contentId);
- 对于小于2000w且多行插入更新、范围查询频率较高的表，建议不分片。由于目前暂不支持此类表（下个版本提供），需使用HASH分布表代替。

典型的分布表定义如下：

```
--定义一个表，表中每行存在所有DN中。
CREATE TABLE warehouse_d1
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)
  W_WAREHOUSE_SQ_FT   INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME       VARCHAR(60)
  W_STREET_TYPE       CHAR(15)
  W_SUITE_NUMBER      CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP              CHAR(10)
  W_COUNTRY           VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
)DISTRIBUTE BY REPLICATION;

--定义一个表，使用HASH分布。
CREATE TABLE warehouse_d2
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)
  W_WAREHOUSE_SQ_FT   INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME       VARCHAR(60)
  W_STREET_TYPE       CHAR(15)
  W_SUITE_NUMBER      CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP              CHAR(10)
  W_COUNTRY           VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2),
  CONSTRAINT W_CONSTR_KEY3 UNIQUE(W_WAREHOUSE_SK)
)DISTRIBUTE BY HASH(W_WAREHOUSE_SK);

--定义一个表，使用RANGE分布。
CREATE TABLE warehouse_d3
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)
  W_WAREHOUSE_SQ_FT   INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME       VARCHAR(60)
  W_STREET_TYPE       CHAR(15)
  W_SUITE_NUMBER      CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
```

```
W_ZIP          CHAR(10)          ,
W_COUNTRY      VARCHAR(20)        ,
W_GMT_OFFSET   DECIMAL(5,2)
)DISTRIBUTE BY RANGE(W_WAREHOUSE_ID)
(
  SLICE s1 VALUES LESS THAN (10) DATANODE dn1,
  SLICE s2 VALUES LESS THAN (20) DATANODE dn2,
  SLICE s3 VALUES LESS THAN (30) DATANODE dn3,
  SLICE s4 VALUES LESS THAN (MAXVALUE) DATANODE dn4
);

--定义一个表，使用LIST分布。
CREATE TABLE warehouse_d4
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)  ,
  W_WAREHOUSE_SQ_FT   INTEGER      ,
  W_STREET_NUMBER     CHAR(10)     ,
  W_STREET_NAME       VARCHAR(60)  ,
  W_STREET_TYPE       CHAR(15)     ,
  W_SUITE_NUMBER      CHAR(10)     ,
  W_CITY              VARCHAR(60)  ,
  W_COUNTY            VARCHAR(30)  ,
  W_STATE             CHAR(2)      ,
  W_ZIP              CHAR(10)      ,
  W_COUNTRY           VARCHAR(20)  ,
  W_GMT_OFFSET        DECIMAL(5,2)
)DISTRIBUTE BY LIST(W_COUNTRY)
(
  SLICE s1 VALUES ('USA') DATANODE dn1,
  SLICE s2 VALUES ('CANADA') DATANODE dn2,
  SLICE s3 VALUES ('UK') DATANODE dn3,
  SLICE s4 VALUES (DEFAULT) DATANODE dn4
);
```

选择分布键

分布表的分布键选择至关重要，如果分布键选择不当，可能会导致数据倾斜，从而导致查询时，I/O负载集中在部分DN上，影响整体查询性能。因此，在确定分布表的分布策略之后，需要对表数据进行倾斜性检查，以确保数据的均匀分布。分布键的选择一般需要遵循以下原则：

- 【建议】选作分布键的字段取值应该比较离散，以便数据能在各个DN上均匀分布。当单个字段无法满足离散条件时，可以考虑使用多个字段一起作为分布键。一般情况下，可以考虑选择表的主键作为分布键。例如，在人员信息表中选择证件号码作为分布键。
- 【建议】在满足第一条原则的情况下，尽量不要选取在查询中存在常量过滤条件的字段作为分布键。例如，在表dwcjk相关的查询中，字段zqdh存在常量过滤条件“zqdh='000001'”，那么尽量不选择zqdh字段作为分布键。
- 【建议】在满足前两条原则的情况，尽量选择查询中的关联条件为分布键。当关联条件作为分布键时，join任务的相关数据都分布在DN本地，将极大减少DN之间的数据流动代价。
- 【建议】分布键不建议超过3列，列数过多将带来较高的计算开销。
- 【规则】合理设计分布键，既考虑查询开发的便利性，又要考虑数据的均匀存储，避免数据倾斜和读热点。
 - a. 应使用取值较为离散的字段作为分布键，以便数据能够均匀分布到各个DN中。
 - b. 在满足条件1情况下，存在常量过滤的字段不建议成为分布键，否则会使所有的查询任务都会分发到唯一固定的DN上。

- c. 在满足条件1和2原则下，尽量选择查询中的关联条件作为分布键，这样可保证JOIN任务的相关数据分布在相同的DN上，减少DN间数据的流动代价。

表 4-5 常见的分布键及效果

分布键值	分布键分布均匀性
用户ID，应用程序中有许多用户。	好
状态代码，只有几个可用的状态代码。	差
项目创建日期，四舍五入至最近的时间段（例如：天、小时或分钟）。	差
设备ID，每个设备以相对类似的间隔访问数据。	好
设备ID，被跟踪的设备有很多，但到现在为止，其中某个设备比其他所有设备更加常用。	差

- 【规则】分布键使用的列长度不宜超过128，过长会带来较高的计算开销。
- 【规则】分布键值一旦插入不允许更新（UPDATE），如需更新需删除后插入。

选择分区方案

表 4-6 表的分区方式及使用场景

分区方式	描述
Range	表数据通过范围进行分区。
List	表数据通过指定列按照具体值进行分区。
Hash	表数据通过Hash散列方式进行分区。

当表中的数据量很大时，应当对表进行分区，一般需要遵循以下原则：

- 【说明】减少需要扫描的数据量。通过分区表的剪枝机制可以大幅减少数据的扫描量。
- 【建议】使用具有明显区间性的字段进行分区，比如日期、区域等字段上建立分区。
- 【建议】分区名称应当体现分区的数据特征。例如，关键字+区间特征。
- 【建议】将分区上边界的分区值定义为MAXVALUE，以防可能出现的数据溢出。
- 【建议】非透明多写特性下，避免分区和分布定义规则相同或相近，以防止部分DN上的分区无法映射到数据。

典型的分区表定义如下：

```
--设置DATE类型格式
SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
--创建Range分区表。
CREATE TABLE staffS_p1
(
```



```
staff_ID    NUMBER(6) not null,
FIRST_NAME  VARCHAR2(20),
LAST_NAME   VARCHAR2(25),
EMAIL       VARCHAR2(25),
PHONE_NUMBER VARCHAR2(20),
HIRE_DATE   DATE,
employment_ID VARCHAR2(10),
SALARY      NUMBER(8,2),
COMMISSION_PCT NUMBER(4,2),
MANAGER_ID   NUMBER(6),
section_ID   NUMBER(4)
)
PARTITION BY RANGE (HIRE_DATE)
(
  PARTITION HIRE_19950501 VALUES LESS THAN ('1995-05-01 00:00:00'),
  PARTITION HIRE_19950502 VALUES LESS THAN ('1995-05-02 00:00:00'),
  PARTITION HIRE_maxvalue VALUES LESS THAN (MAXVALUE)
);

--创建List分区表。
CREATE TABLE test_list (col1 int, col2 int)
partition by list(col1)
(
  partition p1 values (2000),
  partition p2 values (3000),
  partition p3 values (4000),
  partition p4 values (5000)
);

--创建Hash分区表。
CREATE TABLE test_hash (col1 int, col2 int)
partition by hash(col1)
(
  partition p1,
  partition p2
);
```

4.2.8 字段设计规范

- 【规则】字段设计应使用推荐类型。
字段设计需使用推荐字段。有些数据类型不推荐的原因是业务使用场景较少，未大规模商用。

表 4-7 数据库数据类型最佳实践

数据类型	说明	是否推荐
UUID	不同集群可能产生相同UUID	禁止
序列整型	即自增列，包括SMALLSERIAL、SERIAL、BIGSERIAL	禁止
整数类型	TINYINT、SMALLINT、INTEGER、BIGINT	推荐
任意精度类型	NUMERIC/DECIMAL	推荐
浮点类型	REAL/FLOAT4、DOUBLE PRECISION/FLOAT8、FLOAT	推荐
布尔类型	BOOLEAN	推荐
定长字符	CHAR(n)	推荐

数据类型	说明	是否推荐
变长字符	VARCHAR(n)、NVARCHAR2(n)	推荐
	TEXT/CLOB（字符大对象）	不推荐
时间类型	DATE、TIME、TIMESTAMP、SMALLDATETIME、INTERVAL、RELTIME、DATEA（仅在ORA兼容性生效）。	推荐
	TIMETZ、TIMESTAMP TZ	不推荐
二进制类型	BYTEA（变长二进制类型）	推荐
	BLOB（二进制大对象）、RAW（变长十六进制）	不推荐
位串类型	BIT(n)、VARBIT(n)	推荐
特殊字符类型	NAME、"CHAR"，通常供数据库系统内部使用	不推荐
JSON类型	JSON类型目前不支持操作符	不推荐
自定义类型	可用于定义枚举ENUM等类型	不推荐
HLL数据类型	建议直接使用HLL相关函数，减少性能影响	不推荐
货币类型	MONEY存储带有固定小数精度的货币金额	不推荐
几何类型	POINT、LSEG、BOX、PATH、POLYGON、CIRCLE	不推荐
网络地址类型	存储IPv4、IPv6和MAC地址数据类型	不推荐

- 【规则】尽量使用高效的数值类数据类型。在满足业务精度的情况下，选择的优先级从高到低依次为整数、浮点数、NUMERIC。
- 【规则】合理设置数值字段的数据类型，根据取值范围选择合适的数值类型，尽量少用NUMERIC/DECIMAL类型。
 - NUMERIC和DECIMAL等价，NUMERIC/DECIMAL数据类型操作对CPU消耗较高。
 - 当sql_compatibility='MYSQL'时，设置参数b_format_version='5.7'和参数b_format_dev_version='s1'后，TINYINT的范围是：-128 ~ +127。
 - UNSIGNED类型仅支持sql_compatibility='MYSQL'的数据库。

表 4-8 非透明多写特性下的数值类数据类型存储空间及取值范围

类型	存储空间	最小值	最大值
TINYINT [UNSIGNED]	1	0	255
SMALLINT [UNSIGNED]	2	带符号为-32768；无符号为0	带符号为32767；无符号为65535
INTEGER [UNSIGNED]	4	带符号为-2,147,483,648；无符号为0	带符号为2,147,483,647；无符号为4,294,967,295
BIGINT [UNSIGNED]	8	带符号为-9,223,372,036,854,775,808；无符号为0	带符号为9,223,372,036,854,775,807；无符号为18,446,744,073,709,551,615
REAL/FLOAT4	4	6位十进制数字精度	
DOUBLE PRECISION/ FLOAT8	8	15位十进制数字精度	

- 【规则】合理选用字符串数据类型。如果该字段输入确定为固定字符则使用定长字符类型，或需要自动补充空格，否则请使用变长字符类型VARCHAR。

典型的定长字段类型，例如“gender”字段，仅允许输入“f”或“m”一个字节长度的字符。这类字段建议使用定长数据类型（如CHAR(n)）。

如果不存在此特点，或者后续可能扩展需要输入更长的字符，请优先使用变长字符类型（如VARCHAR, TEXT），且不建议指定变长类型的长度。

原因如下：

 - 定长字段会对不够长度的输入数据补充空格，然后存入数据库中，产生不必要的存储空间浪费。
 - 如果定义为定长字符类型，后续扩展长度，需要对全表进行扫描重写，性能开销大，影响在线业务。
 - 对于指定固定长度的变长字段，每次插入时会检查是否长度越界，带来性能开销。
- 【规则】字符类型字段不应存储数字类型的数据。

如果对存储在字符类型字段中的数据进行数值计算，或者与数值进行比较操作（如置于过滤条件中），会带来不必要的数据类型转换的开销，同时该字段上的索引可能失效，影响查询性能。
- 【规则】字符类型字段不应存储时间或日期类数据。

如果对存储在字符类型字段中的数据与日期类数据进行计算或比较操作（如置于过滤条件中），会带来不必要的数据类型转换的开销，同时该字段上的索引可能失效，影响查询性能。
- 【规则】对于明确不存在NULL值的字段加上NOT NULL约束。

对于NOT NULL字段，优化器在某些场景下会进行特殊优化，可提升查询性能。

- 【规则】相关联字段的数据类型应保持一致。
在进行关联操作时，如果字段类型不一致，会带来数据类型转换开销。
- 【规则】大字段（例如varchar（1000）、varchar(4000)）不超过8个。
- 【建议】字段定义时建议同时创建COMMENT注释信息，以便于未来维护。
不同类型字段说明、取值范围及使用方法请参见《参考》中“SQL参考 > 数据类型”章节。
- 【建议】当多个表存在逻辑关系时，表示同一含义的字段应该使用相同的数据类型。
- 【建议】用于WHERE条件过滤和关联的字段都应设置NOT NULL约束。
对于NOT NULL字段，优化器在某些场景下会进行特殊优化，可较大提升查询性能。
- 【建议】不建议对表预留字段。大部分场景下可支持快速新增、删除表字段，或者修改字段的DEFAULT值。
新增列必须符合以下要求，否则会带来全表更新开销，影响在线业务。
 - 数据类型为以下类型中的一种：BOOL、BYTEA、SMALLINT、BIGINT、SMALLINT、INTEGER、NUMERIC、FLOAT、DOUBLE PRECISION、CHAR、VARCHAR、TEXT、TIMESTAMPTZ、TIMESTAMP、DATE、TIME、TIMETZ、INTERVAL、DATEA（仅在ORA兼容性生效）。
 - 新增列的DEFAULT值长度不超过128个字节。
 - 新增列DEFAULT值不包含volatile函数。
 - 新增列设置有DEFAULT值，且DEFAULT值不为NULL。如果不确定是否满足条件3，请联系GaussDB数据库技术人员进行评估。

4.2.9 索引设计规范

- 【规格】使用数据库索引实践推荐的索引类型。

说明

索引设计建议使用推荐类型。

表 4-9 数据库索引实践推荐

索引类型	说明	是否推荐
主键/唯一索引	单列或多列主键/唯一索引	推荐
表达式索引	索引列为表的一列或多列计算而来的一个函数或者标量表达式	限制使用
BTREE索引	索引构建类型	部分数据类型上推荐

- 【规则】组合索引字段个数不超过5个。
- 【规则】禁止总字符串长度超过200的组合索引。
- 【规则】索引（包括单列索引和复合索引）字段应为NOT NULL字段。
- 【规则】不建议单表创建多个唯一索引。

同时维护多个唯一索引的开销远大于维护一个多列唯一索引，如果业务逻辑上多个唯一索引，与一个多列唯一索引等价，应使用多列唯一索引。

- 【建议】同字段上创建索引的维护效率不同。数值类型字段优于字符类型及其他数据类型，因此对于考虑创建索引的ID、时间等字段，建议使用数值类型进行存储。
- 【建议】应考虑在关联列上创建索引。

支持HASH JOIN，但是当内表较小等RESCAN代价较低的情况下，仍然可能选择NESTLOOP JOIN来完成关联。如果通过EXPLAIN可以查看到NESTLOOP JOIN计划，则可以通过在关联列上创建索引，提高NESTLOOP JOIN效率。

4.2.10 函数/存储过程设计规范

- 【规则】禁止使用存储过程、触发器等实现业务逻辑，应该将这些逻辑都放到业务服务器上处理，避免对数据库产生逻辑依赖。
- 【规则】业务的数据库升级脚本中，禁止使用存储过程实现升级逻辑。
- 【规则】仅创建对固定入参有固定返回值的函数，函数必须设为IMMUTABLE和SHIPPABLE类型。

目前数据库支持三种类型的函数，分别是IMMUTABLE、STABLE、VOLATILE。

在非透明多写特性下，则在DN上执行。在此种场景下，所有DN上的数据将发送至某一个CN上进行计算，导致查询执行效率低下。

4.2.11 约束设计

DEFAULT 和 NULL 约束

- 【建议】如果能够从业务层面补全字段值，那么，不建议使用DEFAULT约束，避免数据加载时产生不符合预期的结果。
- 【建议】给明确不存在NULL值的字段加上NOT NULL约束，优化器会在特定场景下对其进行自动优化。
- 【建议】给可以显式命名的约束显式命名。除了NOT NULL和DEFAULT约束外，其他约束都可以显式命名。

唯一约束

- 【建议】从命名上明确标识唯一约束，例如，命名为“UNI+字段名”。

主键约束

- 【建议】从命名上明确标识主键约束，例如，将主键约束命名为“PK+字段名”。

检查约束

- 【建议】从命名上明确标识检查约束，例如，将检查约束命名为“CK+字段名”。

4.2.12 视图和关联表设计

视图设计

- 【建议】除非视图之间存在强依赖关系，否则不建议视图嵌套。
- 【建议】视图定义中尽量避免排序操作。

关联表设计

- 【建议】表之间的关联字段应该尽量少。
- 【建议】关联字段的数据类型应该保持一致。
- 【建议】关联字段在命名上，应该可以明显体现出关联关系。例如，采用同样名称来命名。

4.3 数据库编程规范

4.3.1 GUC 参数编程规范

【规则】客户端（如JDBC）应使用默认参数执行查询，如果需要修改会话级别的GUC参数，应谨慎评估。

通过ODBC或JDBC修改GUC参数时，需注意GUC参数仅会在当前connection中生效，特别是在连接池场景下，容易出现問題且問題定位困难。

如果在连接中必须进行GUC参数设置，那么在将连接归还给连接池之前，必须使用如下命令将连接的状态清空。

```
SET SESSION AUTHORIZATION DEFAULT;  
RESET ALL;
```

4.3.2 对象访问编程规范

- 【规则】用户在操作对象时，应该拥有该对象的操作权限。
权限说明请参考[用户及权限](#)章节，并遵循[权限设计规范](#)。
- 【规则】访问对象（表，函数等）时建议带上SCHEMA名称，即使用schemaname.tablename进行访问。
如果不追加SCHEMA名称前缀，会根据当前search_path中表空间列表，依次搜索所有表空间直到找到匹配的表，这会带来不必要的性能开销。

4.3.3 WHERE

- 【规则】禁止在WHERE条件中进行相同表字段的比较。
例如，如下语句应考虑合理性：

```
SELECT * FROM t1 WHERE col1 = col1;
```


应考虑修改为：

```
SELECT * FROM t1 WHERE col1 IS NOT NULL;
```
- 【规则】禁止WHERE条件涉及隐式数据类型转换。
数据库中进行隐式转换后可能导致无法使用所创建的索引。建议在开发过程中开启GUC参数check_implicit_conversions，并关闭enable_fast_query_shipping，以便检查查询语句中是否存在可能带来不良性能影响的隐式数据类型。

```
SET enable_fast_query_shipping = off;  
SET check_implicit_conversions = true;
```

由于隐式数据类型转换检测存在额外的开销，一旦查询语句开发完成后，请关闭check_implicit_conversions参数，并重置enable_fast_query_shipping。

示例：

- 如下代码不符合规范：

```
--t_tablename表的phonenummer字段为VARCHAR类型，而不是数值类型  
SELECT column1  
  INTO i_l_variable1  
  FROM t_tablename  
 WHERE phonenummer = 13512345678;
```

- 建议修改为：

```
--t_tablename表的phonenummer字段为VARCHAR类型，而不是数值类型  
SELECT column1  
  INTO i_l_variable1  
  FROM t_tablename  
 WHERE phonenummer = '13512345678';
```

- 【规则】禁止WHERE条件字段使用表达式或者函数。

对条件字段使用表达式或函数时，索引会失效，同时会对每一行数据进行计算，产生不必要的性能消耗。

示例：

- 如下代码不符合规范：

```
SELECT income FROM table WHERE abs(income) > ?;  
SELECT income FROM table WHERE income * 10 > ?;  
SELECT create_time  
  FROM table  
 WHERE date_format(create_time, '%Y%m%d %H:%i:%s') = '20090101 00:00:0';
```

- 应修改为：

```
SELECT income FROM table WHERE income > ? OR income < (-1) * ?;  
SELECT income FROM table WHERE income > ?/10;  
SELECT create_time  
  FROM table  
 WHERE create_time = str_to_date('20090101 00:00:0', '%Y%m%d %H:%i:%s');
```

- 【规则】表查询时，WHERE条件中应包含所有分布键字段等值查询条件，否则将在多个节点上进行查询，影响系统并发度和性能。
- 【规则】查询条件中与NULL做比较时，禁止使用“!= ”比较符，应使用IS NULL或IS NOT NULL。
- 【规则】查询条件中禁止对索引字段使用“!= ”比较符，避免索引失效。
- 【规则】查询条件的索引字段上禁止与NULL（IS NULL和IS NOT NULL）进行比较。
- 【规则】查询条件的索引字段上禁止使用NOT。
- 【规则】查询条件的索引字段上禁止使用NOT IN。
- 【规则】模糊查询LIKE语句，%不应放在首字符位置。
如果%放在首字符位置，将无法使用索引，会导致全表扫描。
- 【建议】WHERE条件中IN的候选子集不宜过大，建议不超过500。
 - 查询时，会对IN中每一条数据进行等值比较，开销较大。
 - 非透明多写特性下，如果包含的值为较为固定的值，应考虑创建REPLICATION表，并将候选数据写入表中，然后通过INNER JOIN来实现包含查询。
- 【建议】WHERE条件中IN的候选子集不为常量，而是表中的列时，建议改写为子查询。

在这种情况下，实际上是一个不等值的JOIN，会通过nestloop计划执行。在表过大时执行效率低下，建议修改为等值JOIN的子查询。例如：

- 如下代码不推荐使用：

```
SELECT col1, COALESCE(max(col2 - 1), 0)
FROM t1, t2
WHERE t1.col1 = ANY(VALUES(id1), (id2))
GROUP BY col1;
```

- 建议修改为：

```
SELECT col1, COALESCE(max(tmp), 0) FROM
(
  (
    SELECT col1, (col2-1) AS tmp
    FROM t1, t2
    WHERE t1.col1 = t2.id1 AND t1.col1 != t2.id2
  ) UNION ALL (
    SELECT col1, (col2-1) AS tmp
    FROM t1, t2
    WHERE t1.col1 = t2.id2
  )
) GROUP BY col1;
```

- 【建议】多使用等值操作，少使用非等值操作。

WHERE条件中的非等值条件（IN、BETWEEN、<、<=、>、>=）会导致后面的条件使用不了索引，因为不能同时用到两个范围条件。

4.3.4 SELECT

- 【规则】SELECT语句中禁用通配符字段“*”。

使用通配符字段查询表时，如果因业务或数据库升级导致表结构发生变化，可能出现与业务语句不兼容的情况。因此业务应指明所需查询的表字段名称，避免使用通配符。

- 【规则】带有LIMIT的查询语句中必须带有ORDER BY保证有序。

GaussDB作为一种分布式数据库，表数据将分布在多个DN上。如果SQL语句中只带有LIMIT，而不带有ORDER BY子句，数据库将会把网络传输较快的DN所发送的（符合查询要求的）结果作为最终结果输出到客户端。由于网络传输效率不同时刻可能发生改变，因此导致多次执行该SQL语句时，返回结果表现出不一致的情况。

- 【规则】避免对大字段（如VARCHAR(2000)）执行ORDER BY、DISTINCT、GROUP BY、UNION等会引起排序的操作。

此类操作将消耗大量的CPU和内存资源，执行效率低下。

- 【规则】禁止使用LOCK TABLE语句加锁，仅允许使用 SELECT .. FOR UPDATE语句。

LOCK TABLE提供多种锁级别，但如果对数据库原理和业务理解不足，误用表锁可能触发死锁，导致集群不可用。

- 【建议】避免在SELECT目标列中使用子查询，可能导致计划无法下推到DN执行，影响执行性能。

- 【建议】考虑使用UNION ALL，少使用UNION，注意考虑去重。

UNION ALL不去重，少了排序操作，速度相对UNION更快。如果没有去重的需求，优先使用UNION ALL。

- 【建议】避免频繁使用count()获取大表行数，该操作资源消耗较大，影响并行作业执行效率。

如果不需要实时的行数统计信息，可以尝试使用如下语句来获取表行数。

```
SELECT reltuples FROM pg_class WHERE relname = 'tablename';
```


pg_class中所记录的表行数信息只在对该表执行ANALYZE以后才会更新。目前ANALYZE有两种触发条件:

- 业务主动发送ANALYZE语句，例如：

```
--分析连接库中所有表
ANALYZE;
--分析指定表
ANALYZE tablename;
```
- 借助AUTO VACUUM机制，在每间隔一定时间或表的增删达到一定行数时触发。间隔时间和增删比例可通过GUC参数设置。

4.3.5 INSERT

- 【规格】INSERT ON DUPLICATE KEY UPDATE不支持对主键或唯一约束的列上执行UPDATE。
INSERT ON DUPLICATE KEY UPDATE的语义是对唯一约束冲突的行进行更新，这个过程中不应约束的值进行更新。
- 【规则】非透明多写特性下，禁止对存在多个唯一约束的表执行INSERT ON DUPLICATE KEY UPDATE。
表中存在多个唯一约束包括存在多个唯一索引，或既存在主键（PRIMARY KEY），又存在唯一索引（UNIQUE INDEX）两种情况。当存在多个唯一约束时，会默认检查所有的唯一约束条件，只要任何一个约束存在冲突，就会对冲突行进行更新，即可能更新多条记录，与业务预期不相符。业务应给予更加明确的插入更新条件。
- 【建议】对于批量插入的情况，建议使用executeBatch执行INSERT INTO VALUES (?), 执行效率将高于执行多条INSERT INTO VALUES(?)或INSERT INTO VALUES(?),..., (?)。

4.3.6 UPDATE

- 【规格】非透明多写特性下，在GTM-FREE模式下，不允许跨节点事务，因此更新HASH分布中数据表时WHERE条件中必须指定分布列等值过滤条件。
- 【规则】UPDATE语句中必须有WHERE子句，避免全表扫描。
- 【规则】不允许在UPDATE子句同时更新多个列时，被更新列同样是更新源。
同时更新多列，且更新源相同，在不同的数据库下行为不同，为了避免带来兼容性问题，业务层应避免上述操作。例如：

```
UPDATE table SET col1 = col2, col3 = col1 WHERE col1 = 1;
```


该语句在GaussDB中，col3的值为原col1的值；而MySQL中，col3的值为col2的值（因为col2的值被赋予了col1）。
- 【规则】UPDATE语句中禁止使用ORDER BY、GROUP BY子句，避免不必要的排序。
- 【建议】有主键/索引的表，更新时WHERE条件应结合主键/索引。

4.3.7 DELETE

- 【规格】非透明多写特性下，不支持DELETE语句中使用LIMIT。应使用WHERE条件明确需要更新的目标行。
- 【规格】非透明多写特性下，在GTM-FREE模式下，不允许跨节点事务，因此删除HASH分布表中数据时，必须在WHERE条件中指定分布列等值过滤条件。
- 【规则】DELETE语句中必须有WHERE子句，避免全表扫描。

- 【规则】DELETE语句中禁止使用ORDER BY、GROUP BY子句，避免不必要的排序。
- 【规则】如果需要清空一张表，建议使用TRUNCATE，而不是DELETE。
 - 非透明多写特性下，TRUNCATE会创建新的物理文件，并在事务结束时将原文件物理删除，清空磁盘空间。而DELETE会将表中数据进行标记，直到VACUUM FULL阶段才会真正清理磁盘空间。
- 【建议】DELETE有主键或索引的表，WHERE条件应结合主键或索引，提高执行效率。

4.3.8 关联查询

- 【规则】多表关联嵌套深度必须小于8。
关联嵌套过深，容易产生慢SQL，应从业务层考虑优化。
- 【规则】表关联查询必须明确指定各表的连接条件（ON），以避免产生笛卡尔积。
在MySQL中，JOIN与CROSS JOIN和INNER JOIN等价，但是在SQL标准中，JOIN仅与INNER JOIN等价，必须配合使用ON连接条件。
- 【规则】关联时，应该根据SQL标准指明连接方式，避免直接使用JOIN关键词，而是使用CROSS JOIN、INNER JOIN、LEFT JOIN、RIGHT JOIN等。
- 【规则】多表关联查询时，必须对表添加使用别名，保证语句逻辑清晰，便于维护。
- 【建议】不同字段的比较开销不同，关联字段应尽量使用比较效率高的字段类型。
 - 数值类型的比较效率远高于字符串类型。
 - 在数值类型中，整型效率高于NUMERIC和浮点类型。
- 【建议】关联字段应为相同数据类型，避免存在隐式类型转换影响执行效率。
- 【建议】少用嵌套子查询，尽量使用表关联，因为子查询会产生临时表，对SQL性能影响较大。
- 【建议】对于关联列上存在大量NULL值的情况，建议在WHERE条件中增加关联列IS NOT NULL的过滤条件，能够提升执行效率。
- 多表关联示例请参见《参考》中“SQL参考 > SQL语法 > S > SELECT > 连接查询”章节示例。

4.3.9 子查询

- 【规则】禁止一条SQL语句中，出现重复子查询语句。
- 【建议】避免在SELECT目标列中使用子查询，可能导致计划无法下推影响执行性能。
- 【建议】少用标量子查询。
标量子查询指结果为一个值，并且条件表达式为等值的子查询。不符合规范的语句示例如下：

```
SELECT * FROM t1 WHERE id = (SELECT id FROM t2 LIMIT 1);
```


上述语句建议业务拆分为两条SQL语句，先执行子查询。
- 【建议】子查询嵌套深度不建议超过两层。
由于子查询会带来临时表开销，过于复杂的查询应考虑从业务逻辑上进行优化。

4.3.10 事务

- 【规格】非透明多写特性下，在GTM-FREE模式下，不允许执行跨节点事务。

在GTM-FREE模式下，如果所执行的SQL语句包含跨节点事务，会报错：
Unsupport DML two phase commit under gtm free mode.

在这种情况下，如果某一条语句必须临时开启跨节点事务，需要在事务中先执行如下命令开启跨节点事务：

```
SET enable_twophase_commit = on;
```

并在事务结束前执行如下命令：

```
RESET enable_twophase_commit;
```

或者：

```
SET enable_twophase_commit = off;
```

- 【规格】大对象操作不支持事务。

说明

非透明多写特性下，大对象操作包括：创建删除DATABASE、ANALYZE、VACUUM。

- 【规则】通过JDBC客户端接入时，关闭autocommit参数，显式执行事务的COMMIT。

- 开启autocommit参数，会导致部分参数（如fetchsize）失效。
- 业务应明确业务逻辑，减少对数据库的依赖。

- 【规则】通过JDBC接入数据库时，避免拼接多条SQL为一条语句发送执行。

当多条语句拼接为一条语句，其中一条执行失败后会整体返回失败，不利于定位错误，建议拆分。

示例：

- 不符合规则语句

```
Connection conn = ....
try {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate("CREATE TABLE t1 (a int); DROP TABLE t1");
    } finally {
        stmt.close();
    }
    conn.commit();
} catch (Exception e) {
    conn.rollback();
} finally {
    conn.close();
}
```

- 建议拆分成两条语句分别发送：

```
Connection conn = ....
try {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate("CREATE TABLE t1 (a int)");
        stmt.executeUpdate("DROP TABLE t1");
    } finally {
        stmt.close();
    }
    conn.commit();
} catch (Exception e) {
    conn.rollback();
} finally {
    conn.close();
}
```

```
conn.close();  
}
```

4.3.11 SQL 编写

DDL

- 在GaussDB中，建议DDL（建表、COMMENT等）操作统一执行。在批处理作业中尽量避免DDL操作，避免大量并发事务对性能的影响。
- 在非日志表（unlogged table）使用完后，立即执行数据清理（TRUNCATE）操作。因为在异常场景下，GaussDB不保证非日志表(unlogged table)数据的安全性。
- 临时表和非日志表的存储方式建议和基表相同。
- 索引字段的总长度不超过50字节。否则，索引大小会膨胀比较严重，带来较大的存储开销，同时索引性能也会下降。
- 不要使用DROP...CASCADE方式删除对象，除非已经明确对象间的依赖关系，以免误删。
- 使用CREATE TABLE AS SELECT ... 语句创建表时，若涉及数据类型转换，请使用强制类型转换，以防隐式类型转换结果与预期不符。

详细DDL语法请参见《参考》中“SQL参考 > SQL语法 > DDL语法一览表”章节。

数据加载和卸载

- 在INSERT语句中显式设置插入的字段列表。例如：
INSERT INTO task(name,id,comment) VALUES ('task1','100','第100个任务');
- 在批量数据入库之后，或者数据增量达到一定阈值后，建议对表进行ANALYZE操作，防止统计信息不准确而导致的执行计划劣化。
- 如果要清理表中的所有数据，建议使用TRUNCATE TABLE方式，不要使用DELETE TABLE方式。DELETE TABLE方式删除性能差，且不会释放那些已经删除了的数据占用的磁盘空间。

类型转换

- 在需要数据类型转换（不同数据类型进行比较或转换）时，使用强制类型转换，以防隐式类型转换结果与预期不符。
- 在查询中，对常量要显式指定数据类型，不要试图依赖任何隐式的数据类型转换。
- 若sql_compatibility参数为ORA，在导入数据时，空字符串会自动转化为NULL。如果需要保留空字符串，则需sql_compatibility参数为TD。

详细数据类型介绍请参见《参考》中“SQL参考 > 数据类型”章节。

查询操作

- 除ETL程序外，应该尽量避免向客户端返回大量结果集的操作。如果结果集过大，应考虑业务设计是否合理。
- 使用事务方式执行DDL和DML操作。例如，TRUNCATE TABLE、UPDATE TABLE、DELETE TABLE、DROP TABLE等操作，一旦执行提交则无法恢复。对于这类操作，建议使用事务进行封装，必要时可以进行回滚。

- 在查询编写时，建议明确列出查询涉及的所有字段，不建议使用“SELECT *”语法。一方面基于性能考虑，尽量减少查询输出列，另一方面避免增删字段对前端业务兼容性的影响。
- 在访问表对象时带上Schema前缀，可以避免因Schema切换导致访问到非预期的表。
- 超过3张表或视图进行关联（特别是FULL JOIN）时，执行代价难以估算。建议使用WITH TABLE AS语句创建中间临时表的方式增加SQL语句的可读性。
- 尽量避免使用笛卡尔积和FULL JOIN。这些操作会造成结果集的急剧膨胀，同时其执行性能也会降低。
- NULL值的比较只能使用IS NULL或者IS NOT NULL的方式判断，其他任何形式的逻辑判断都返回NULL。例如：NULL<>NULL、NULL=NULL和NULL<>1返回结果都是NULL，而不是期望的布尔值。
- 需要统计表中所有记录数时，不要使用count(col)来替代count(*)。count(*)会统计NULL值（真实行数），而count(col)不会统计。
- 在执行count(col)时，将“值为NULL”的记录行计数为0。在执行sum(col)时，当所有记录都为NULL时，最终将返回NULL；当不全为NULL时，“值为NULL”的记录行将被计数为0。
- count(多个字段)时，多个字段名必须用圆括号括起来。例如，count((col1,col2,col3))。通过多字段统计行数时，即使所选字段都为NULL，该行也被计数，效果与count(*)一致。
- count(distinct col)用来计算该列不重复的非NULL的数量，NULL将不被计数。
- count(distinct (col1,col2,...))用来统计多列的唯一值数量，当所有统计字段都为NULL时，也会被计数，同时这些记录被认为是相同的。
- 使用连接操作符“||”替换concat函数进行字符串连接。因为concat函数的输出跟data type有关，生成执行计划时不能提前计算结果值，导致查询性能严重劣化。
- 使用下面表4-10替换now函数来获取当前时间。因为now函数生成的执行计划无法下推，导致查询性能严重劣化。

表 4-10 时间相关的宏

宏名称	描述	示例
CURRENT_DATE	获取当前日期，不包含时分秒。	<pre>gaussdb=# SELECT CURRENT_DATE; date ----- 2018-02-02 (1 row)</pre>
CURRENT_TIME	获取当前时间，不包含年月日。	<pre>gaussdb=# SELECT CURRENT_TIME; timetz ----- 00:39:34.633938+08 (1 row)</pre>
CURRENT_TIMESTAMP(n)	获取当前日期和时间，包含年月日时分秒及时区信息。 说明 n表示存储的毫秒位数。	<pre>gaussdb=# SELECT CURRENT_TIMESTAMP(6); timestampz ----- 2018-02-02 00:39:55.231689+08 (1 row)</pre>

- 尽量避免标量子查询语句的出现。标量子查询是出现在SELECT语句输出列表中的子查询，在下面例子中，“SELECT COUNT(*) FROM films f WHERE f.did = s.id”部分即为一个标量子查询语句。

```
SELECT id, (SELECT COUNT(*) FROM films f WHERE f.did = s.id) FROM staffs_p1 s;
```

标量子查询往往会导致查询性能严重劣化，在应用开发过程中，应当根据业务逻辑，对标量子查询进行等价转换，将其写为表关联。
- 在WHERE子句中，应当对过滤条件进行排序，把选择读较小（筛选出的记录数较少）的条件排在前面。
- WHERE子句中的过滤条件，尽量符合单边规则。即把字段名放在比较条件的一边，优化器在某些场景下会自动进行剪枝优化。形如col op expression，其中col为表的一个列，op为‘=’、‘>’的等比较操作符，expression为不含列名的表达式。例如，

```
SELECT id, from_image_id, from_person_id, from_video_id FROM face_data WHERE current_timestamp(6) - time < '1 days'::interval;
```

改写为：

```
SELECT id, from_image_id, from_person_id, from_video_id FROM face_data where time > current_timestamp(6) - '1 days'::interval;
```
- 尽量避免不必要的排序操作。排序需要耗费大量的内存及CPU，如果业务逻辑许可，可以组合使用ORDER BY和LIMIT，减小资源开销。GaussDB默认按照ASC & NULL LAST进行排序。
- 使用ORDER BY子句进行排序时，显式指定排序方式（ASC/DESC），NULL的排序方式（NULL FIRST/NULL LAST）。
- 不要单独依赖LIMIT子句返回特定顺序的结果集。如果返回部分特定结果集，可以将ORDER BY子句与LIMIT子句组合使用，必要时也可以使用OFFSET跳过特定结果。
- 在保障业务逻辑准确的情况下，建议尽量使用UNION ALL来代替UNION。
- 如果过滤条件只有OR表达式，可以将OR表达式转化为UNION ALL以提升性能。使用OR的SQL语句经常无法优化，导致执行速度变慢。例如，下面语句的转换。

```
SELECT * FROM scdc.pub_menu WHERE (cdp= 300 AND inline=301) OR (cdp= 301 AND inline=302) OR (cdp= 302 AND inline=301);
```

转换为：

```
SELECT * FROM scdc.pub_menu WHERE (cdp= 300 AND inline=301) union all SELECT * FROM scdc.pub_menu WHERE (cdp= 301 AND inline=302) union all SELECT * FROM tablename WHERE (cdp= 302 AND inline=301)
```
- 当IN(val1, val2, val3...)表达式中字段较多时，建议使用IN (VALUES (val1), (val2), (val3)...)语句进行替换。优化器会自动把IN约束转换为非关联子查询，从而提升查询性能。
- 在关联字段不存在NULL值的情况下，使用(NOT) EXISTS代替(NOT) IN。例如，在下面查询语句中，当T1.C1列不存在NULL值时，可以先为T1.C1字段添加NOT NULL约束，再进行改写。

```
SELECT * FROM T1 WHERE T1.C1 NOT IN (SELECT T2.C2 FROM T2);
```

可以改写为：

```
SELECT * FROM T1 WHERE NOT EXISTS (SELECT * FROM T2 WHERE T1.C1=T2.C2);
```

📖 说明

- 如果不能保证T1.C1列的值为NOT NULL的情况下，不能进行上述改写。
- 如果T1.C1为子查询的输出，要根据业务逻辑确认其输出是否为NOT NULL。
- 通过游标进行翻页查询，而不使用LIMIT OFFSET语法，避免多次执行带来的资源开销。游标必须在事务中使用，执行完后务必关闭游标并提交事务。

4.4 客户端编程规范

4.4.1 JDBC

- **【说明】**第三方工具通过JDBC连接GaussDB时，JDBC向GaussDB发起连接请求，会默认添加以下配置参数，详见JDBC代码ConnectionFactoryImpl类的实现。

```
params = {
    { "user", user },
    { "database", database },
    { "client_encoding", "UTF8" },
    { "DateStyle", "ISO" },
    { "extra_float_digits", "3" },
    { "TimeZone", createPostgresTimeZone() },
};
```

这些参数可能会导致JDBC客户端的行为与gsq客户端的行为不一致，例如，Date数据显示方式、浮点数精度表示、timezone显示。

如果实际期望和这些配置不符，建议在java连接设置代码中显式设定这些参数。

通过JDBC连接数据库时，会设置extra_float_digits=3，gsq中设置为extra_float_digits=0，可能会造成同一条数据在JDBC显示和gsq显示的精度不同。

M-Compatibility模式数据库下，extra_float_digits的默认值为0。

- **【规格】**JDBC实例必须指定数据库，一旦实例创建，无法切换数据库。
- **【规格】**单条SQL语句的长度不允许超过2G字节，业务应考虑通信成本，建议单条SQL语句不超过5K。
- **【规格】**目前仅支持对CREATE/ALTER TABLE中列的DEFAULT值进行参数化设置，其他DDL使用Prepare Execute执行方式进行参数设置无效。
- **【规则】**JDBC每条PreparedStatement语句中的参数不可超过32767个。
- **【规格】**连接参数fetchsize必须要在autocommit关闭情况下使用，否则fetchsize配置无效。
- **【规则】**使用默认GUC参数，避免通过JDBC发送SET请求修改GUC参数。详情请参见参考[GUC参数编程规范](#)。
- **【规则】**非透明多写特性下，数据库会根据GUC参数jdbc.version_num进行版本控制，禁止应用修改该参数。jdbc.version_num从503.1版本开始支持，jdbc.version_num规范如[表4-11](#)所示。

表 4-11 jdbc.version_num 参数规范

版本	jdbc.version_num	范围
505.2	502xx	50200-50299
505.1	501xx	50100-50199

版本	jdbc.version_num	范围
505.0	500xx	50000-50099
503.1	301xx	30100-30199

- 【规则】必须使用Prepare Execute方式执行查询语句，提高执行效率。
- 【规则】在同一个事务中，应逐条执行SQL语句，避免拼接多条SQL作为一条语句发送。详情请参见【规则】[通过JDBC接入数据库时，避免拼接多条SQL为一条语句发送执行](#)。
- 【规则】如果在连接中创建了临时表，那么在将连接归还给连接池之前，必须将临时表删除，避免业务出错。
- 【建议】对于精度敏感的场景，建议使用numeric类型。
- 【建议】通过JDBC连接数据库时，应该保证以下三个时区设置一致：
 - JDBC客户端所在主机的时区。
 - GaussDB集群所在主机的时区。
 - GaussDB集群配置过程中时区。

说明

时区设置相关的操作，请联系管理员。

- 【建议】在JDBC向GaussDB申请连接的代码中，建议显式开启autocommit。如果基于性能或者其它方面考虑，需要关闭autocommit时，需要应用程序保证事务的提交。例如，在指定的业务SQL执行完之后做显式提交，特别是客户端退出之前务必保证所有的事务已经提交。
- 【建议】推荐使用连接池限制应用程序的连接数。不建议每执行一条SQL就连接一次数据库。
- 【建议】在应用程序完成作业任务之后，应当及时断开和GaussDB的连接，释放资源。建议在任务中设置session超时时间参数。
- 【建议】使用JDBC连接池，在将连接释放给连接池前，需要执行以下操作，重置会话环境。否则，可能会因为历史会话信息导致对象冲突。
 - 如果在连接中设置了GUC参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。
 - 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。
- 【建议】在不使用ETL工具，数据入库实时性要求比较高的情况下，建议在开发应用程序时，使用GaussDB JDBC驱动的copyManger接口进行微批导入。
- 【建议】非透明多写特性下，建议设置连接参数autoBalance=true，开启CN负载均衡功能，并设置多个CN连接地址（使用逗号分隔）。
 - 一旦开启autoBalance，JDBC DRIVER会尝试将JDBC connection分配到不同的CN节点上。
 - 设置多个CN连接地址的目的是JDBC DRIVER在首次获取集群CN列表时，避免因所设置的CN节点故障而失败。
 - 一旦首次成功获取，便不会再依赖连接参数中指定的CN列表，而是根据实时获取的集群CN列表，每隔一段时间，选取连接其中有效的CN获取最新的CN列表。

- 【建议】建议设置连接参数batchMode=on，使用batch模式连接，提高执行性能。
- 【建议】应根据业务上层请求超时时间合理设置JDBC连接超时时间，避免作业完成或超长作业持续占用数据库资源。

📖 说明

超时参数包括loginTimeout、connectTimeout、socketTimeout等。

- loginTimeout：Integer类型。指建立数据库连接的等待时间。超时时间单位为秒。默认值为0，表示已禁用，timeout不发生。
- connectTimeout：Integer类型。用于连接服务器操作的超时值。如果连接到服务器花费的时间超过此值，则连接断开。超时时间单位为秒，默认值为0，表示已禁用，timeout不发生。
- socketTimeout：Integer类型。用于socket读取操作的超时值。如果从服务器读取所花费的时间超过此值，则连接关闭。超时时间单位为秒，默认值为0，表示已禁用，timeout不发生。
- cancelSignalTimeout：Integer类型。发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect超时”和“socket超时”。超时时间单位为秒，默认值为10秒。
- tcpKeepAlive：Boolean类型。启用或禁用TCP保活探测功能。默认值为false。

4.4.2 连接控制规范

【规则】建议维持静态连接池或控制建立连接频率，以避免频繁触发新建连接。如果CN或DN单节点的连接接入每秒超过以下建议次数，在未开启建连线程池时，可能会导致线程池和CPU使用率激增，从而显著影响业务性能；在开启建连线程池时会造成客户端建连时延的增加，并可能引起客户端建连超时报错。

- 8U/64G系统，每秒连接接入次数空载业务下不高于90，30% CPU业务负载下不高于50，50% CPU业务负载下不高于30，80%业务负载下不高于12。
- 16U/128G系统，每秒连接接入次数空载业务下不高于160，30% CPU业务负载下不高于90，50% CPU业务负载下不高于50，80%业务负载下不高于12。
- 32U/256G系统，每秒连接接入次数空载业务下不高于180，30% CPU业务负载下不高于150，50% CPU业务负载下不高于50，80%业务负载下不高于12。
- 64U/512G系统，每秒连接接入次数空载业务下不高于180，30% CPU业务负载下不高于170，50% CPU业务负载下不高于60，80%业务负载下不高于20。

4.4.3 C/C++应用程序编译规范

在C/C++应用程序开发过程中，编译器类型、语言标准及编译选项的差异，可能引入未定义行为、安全隐患和平台兼容性问题，进而影响系统稳定性和产品交付质量。本规范对C/C++应用程序的编译器选型、语言标准、编译参数、构建模式及相关管理要求作出统一规定，作为项目开发、集成、测试和交付过程中的基本遵循依据。

如果需要定位C/C++应用程序中出现的内存问题，可以开启ASan（AddressSanitizer）并解析ASan日志定位问题原因，详细请参考[ASan](#)以及[ASan日志解析](#)。

ASan

ASan（AddressSanitizer）是一款用于C/C++的内存错误检测器。以下内容介绍如何使用ASan日志查看内存问题。

ASan可以获取的内存错误信息，请参见[表4-12](#)。

表 4-12 Asan 报错信息关键字

关键字	含义
heap-buffer-overflow	堆越界
heap-use-after-free	释放后使用
double-free	重复释放
invalid-free	非法释放
stack-buffer-overflow	栈越界
stack-use-after-return	栈UAF（函数返回后仍然使用其栈内存）
global-buffer-overflow	全局区越界
alloc-dealloc-mismatch	分配释放不匹配
new-delete-type-mismatch	new/delete 错误
use-after-poison	访问红区
memory-leak	内存泄漏

说明

该工具支持x86、ARMCPU架构，支持Linux操作系统。

开启并生成 ASan 日志

以下步骤详细介绍如何开启ASan，并生成ASan日志：

步骤1 使用ASan，需要在编译C/C++程序时在编译命令中添加-fsanitize=address。

以最基础的编译命令举例：

```
gcc test.c -o test -fsanitize=address
```

步骤2 检查ASan环境变量是否设置。

```
echo $ASAN_OPTIONS
```

如果输出为空，执行**步骤3**。否则检查Asan环境变量是否含有目标参数，并且参数值设置正确。

步骤3 设置Asan环境变量。

执行以下命令设置ASan环境变量：

```
export ASAN_OPTIONS="unmap_shadow_on_exit=true:abort_on_error=true:log_path=[log_path]/  
[log_file_prefix]:alloc_dealloc_mismatch=1:fast_unwind_on_fatal=true:detect_stack_use_after_return=1:log_ex  
e_name=true"
```

说明

ASAN_OPTIONS可设置参数请参考。

- unmap_shadow_on_exit：在退出时显式地取消映射（大页）影子内存。默认为false。
- abort_on_error：在打印错误报告后，工具将调用abort()函数，而不是_exit()函数。默认为false。
- log_path：将日志写入“log_path.pid”。特殊值有“stdout”和“stderr”，默认为“stderr”。

其中环境变量中建议设置log_path，该参数为ASan日志输出路径，如不设置，无法获取具体的报错栈。[log_path]为日志文件目录，[log_file_prefix]为日志文件前缀。[log_path]和[log_file_prefix]均可由用户设置。

- alloc_dealloc_mismatch：用来检测“分配 / 释放方式不匹配”的内存错误，默认为0。
- fast_unwind_on_fatal：在发生致命错误时，使用基于快速帧指针的异常处理程序。默认为false。
- detect_stack_use_after_return：用于检测“函数返回后仍然使用其栈内存”的错误（Stack Use-After-Return，简称UAR），默认为0。
- log_exe_name：报告错误时提及可执行文件名称，并在日志中添加可执行文件名（如“log_path.exe_name.pid”）。

步骤4 运行编译后的文件。

```
./test
```

步骤5 查看ASan日志

执行结束后，ASan日志会生成在步骤2 ASan环境变量ASAN_OPTIONS中log_path设置的路径[log_path]中，日志格式为[log_file_prefix].[exe_name].[pid]。

以log_path设置为“/var/log”，log_file_prefix设置为“runlog”，exe_name为可执行文件名为test，运行test时的pid为1234为例，日志文件输出路径为：

```
/var/log/runlog.test.1234
```

----结束

ASan 日志解析

下面通过一段ASan日志分析日志结构：

```
=====
==602791==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x603000026904 at pc
0x000000400775 bp 0x7ffeba71f610 sp 0x7ffeba71f608
WRITE of size 4 at 0x603000026904 thread T0
#0 0x400774 in main (/home/test/workspace/mem+0x400774)
#1 0x7fb441858964 in __libc_start_main (/lib64/libc.so.6+0x22964)
#2 0x400658 (/home/test/workspace/mem+0x400658)

0x603000026904 is located 0 bytes to the right of 20-byte region [0x6030000268f0,0x603000026904)
allocated by thread T0 here:
#0 0x7fb4428d84d0 in __interceptor_malloc ../.././libsanitizer/asan/asan_malloc_linux.cc:62
#1 0x400731 in main (/home/test/workspace/mem+0x400731)
#2 0x7fb441858964 in __libc_start_main (/lib64/libc.so.6+0x22964)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/test/workspace/mem+0x400774) in main
Shadow bytes around the buggy address:
 0x0c067ffccd0: 00 00 00 00 fa fa 00 00 00 00 fa fa 00 00 00 fa
 0x0c067ffcce0: fa fa 00 00 00 00 fa fa 00 00 00 fa fa fd fd
 0x0c067ffccf0: fd fd fa fa fd fd fd fa fa 00 00 00 fa fa fa
 0x0c067ffcd00: fd fd fd fd fa fa fd fd fd fa fa fd fd fd
 0x0c067ffcd10: fa fa fd fd fd fa fa 00 00 00 fa fa fa 00 00
=>0x0c067ffcd20:[04]fa fa fa fa fa fa fa fa fa fa
 0x0c067ffcd30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

```
0x0c067ffcd40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067ffcd50: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067ffcd60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067ffcd70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable:          00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone:    fa
Freed heap region:    fd
Stack left redzone:    f1
Stack mid redzone:     f2
Stack right redzone:   f3
Stack after return:    f5
Stack use after scope: f8
Global redzone:        f9
Global init order:     f6
Poisoned by user:      f7
Container overflow:    fc
Array cookie:          ac
Intra object redzone:  bb
ASan internal:         fe
Left alloca redzone:   ca
Right alloca redzone:  cb
==602791==ABORTING
```

从上到下拆分日志依次解析：

日志头部信息：

```
==602791==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x603000026904 at pc
0x0000000400775 bp 0x7ffeba71f610 sp 0x7ffeba71f608
```

字段含义：

- ==602791==
进程 ID（PID）。ASan用这种格式标识输出来自哪个进程（多进程时作用明显）。
- ERROR:
日志级别/事件类型（通常是 ERROR）。
- AddressSanitizer
产生该报告的sanitizer名称。
- heap-buffer-overflow
错误类型关键字，详情请参考[表4-12](#)。
- on address 0x...
出问题的内存地址（被访问的目标地址）。
- at pc 0x...
程序计数器（PC）：发生访问时的指令地址（可用于符号化到函数/行号）。
- bp 0x... sp 0x...
寄存器快照：base pointer / stack pointer（用于定位栈帧、调试）。
- WRITE of size 4 at 0x603000026904 thread T0
主线程在向堆内存地址写入4字节数据时发生了非法访问，其中thread T0表示线程T0，即主线程。

访问动作摘要如下：

```
WRITE of size 4 at 0x603000026904 thread T0
```

字段含义：

- WRITE/READ
访问类型：写或读。
- of size 4
访问大小（字节数）。
- at 0x...
再次给出被访问地址（与首行一致）。
- thread T0
线程标识：T0通常是主线程；多线程程序会有T1/T2...

报错触发调用栈如下：

```
#0 0x400774 in main (/home/test/workspace/mem+0x400774)
#1 0x7fb441858964 in __libc_start_main (/lib64/libc.so.6+0x22964)
#2 0x400658 (/home/test/workspace/mem+0x400658)
```

地址与对象边界描述如下：

0x603000026904 is located 0 bytes to the right of 20-byte region [0x6030000268f0,0x603000026904)

- is located ...
目标地址相对某个已知内存块的相对位置（例如：在右边界外0字节、左边界外为多少字节、位于块内部偏移多少字节等）。
- xx-byte region
相关内存块的大小。
- [start, end)
该内存块的起止地址区间（左闭右开是常用表示法）。

分配/释放来源：

```
allocated by thread T0 here:
#0 0x... in __interceptor_malloc ....62
#1 0x... in main (...)
```

- allocated by thread T0 here
说明内存是在哪个线程、通过哪次分配得到的，并给出分配时栈回溯。
- #0 ... __interceptor_malloc
拦截器函数（interceptor）：ASan会把 malloc/free/new/delete等替换/包装成自己的版本，用来记录元数据与检测越界。
asan_malloc_linux.cc:62是ASan运行库源码位置（用于表明这是ASan的拦截实现）。
若是use-after-free，通常还会出现 freed by thread ... here: 的释放栈。

SUMMARY行：简要归纳（Summary）：

SUMMARY: AddressSanitizer: heap-buffer-overflow (/path/to/bin+0x400774) in main

通常包含：

- sanitizer名称。
- 错误类型。
- 一个“代表性位置”（模块+偏移、以及符号化后的函数名）。

简要归纳可以快速定位“主要出错点”。

5 应用程序开发教程

5.1 GaussDB 应用程序开发教程概述

GaussDB支持通过JDBC、ODBC、libpq、Psycopg、Go、ecpg等接口来连接数据库并进行操作。

说明

应用程序使用各驱动执行的SQL语句字符串中不能包含0字符，否则会导致执行失败。

JDBC

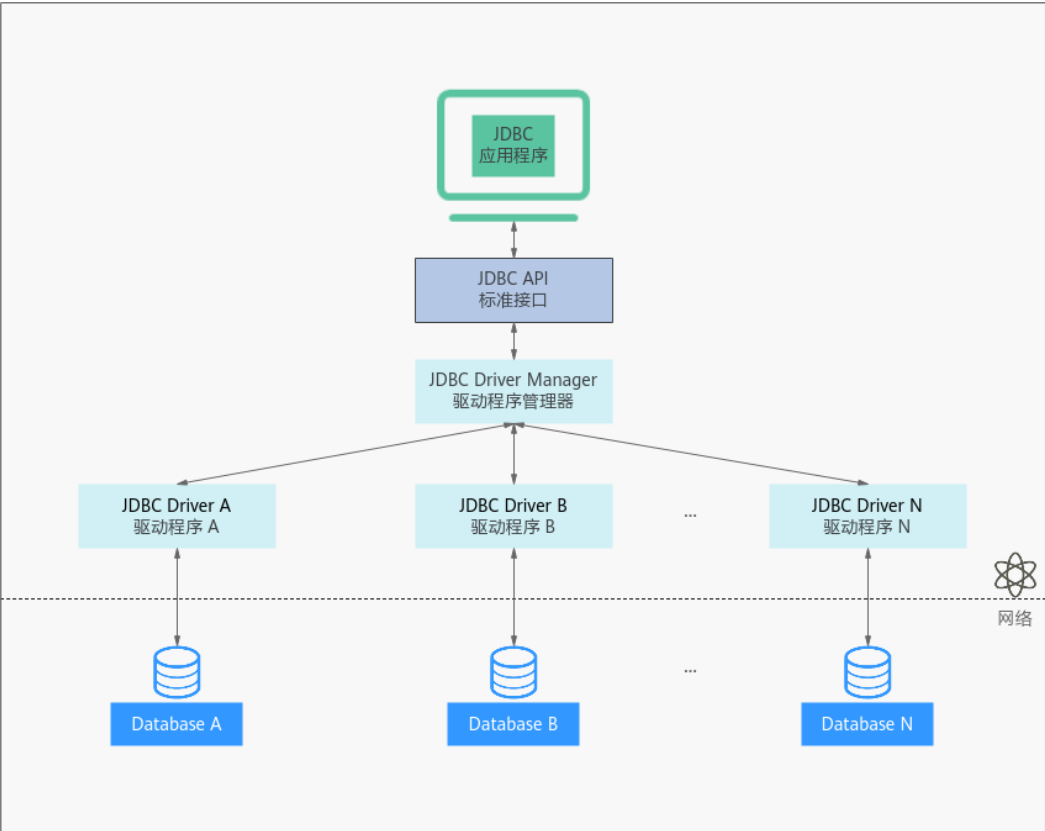
JDBC（Java Database Connectivity，Java数据库连接）是用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问接口。它允许Java应用程序通过SQL语句来执行数据库操作，包括查询、插入、更新和删除数据等。

JDBC架构主要包括以下组件：

- JDBC API：提供Java应用程序与数据库交互的标准接口（如Connection、Statement、ResultSet等）。
- 驱动程序管理器：加载和管理JDBC驱动程序，根据连接请求url匹配合适的驱动。
- 驱动程序：实现JDBC API的接口，将Java调用转换为数据库协议，发送SQL到数据库执行，并将结果返回到应用程序。

Java应用程序发起数据库操作请求（如查询、更新），通过JDBC API接口与JDBC驱动程序通信，而JDBC驱动程序负责加载和管理JDBC驱动程序。JDBC驱动程序则负责与特定的数据库通信，发送SQL到数据库。数据库执行SQL操作并返回结果集。JDBC的系统架构请参见图5-1。

图 5-1 JDBC 系统架构



以下是JDBC的一些关键特点和用法：

- 1. 数据库连接管理：JDBC允许应用程序建立与数据库的连接，并管理这些连接的生命周期。
- 2. SQL执行：使用JDBC可以执行SQL查询、更新、删除等操作。这通过在Java代码中构建SQL语句并将其发送到数据库来实现。
- 3. 事务管理：JDBC支持事务管理，可以通过JDBC API来启动、提交或回滚事务，确保数据库操作的一致性和完整性。
- 4. 异常处理：JDBC定义了一组异常类来处理数据库操作期间可能发生的各种异常情况，开发人员可以使用try-catch块来捕获和处理这些异常。
- 5. 批处理操作：JDBC提供了批处理功能，允许一次性执行多个SQL语句，从而提高数据库操作的效率。
- 6. 元数据访问：通过JDBC可以获取数据库的元数据信息，如表结构、列名、数据类型等，从而动态地构建SQL查询或根据数据库结构进行操作。

总的来说，JDBC提供了一个灵活且强大的桥梁，使Java应用程序能够与各种不同的数据库进行交互，从而实现数据的持久化和管理。GaussDB数据库提供了对JDBC 4.2特性的支持，需要使用JDK1.8、JDK17、JDK21版本（需使用对应版本的驱动包）编译程序代码，不支持JDBC桥接ODBC方式。

基于JDBC开发详细教程请参见[基于JDBC开发](#)。

ODBC

ODBC（Open Database Connectivity，开放式数据库连接）是由Microsoft基于Open Group与ISO/IEC的CLI（Call Level Interface）规范制定的数据库访问标准接口，使用

C/C++语言提供统一的API，使应用程序能够在不同数据库管理系统（DBMS）之间实现跨平台访问，而无需关注底层数据库类型或操作系统差异。

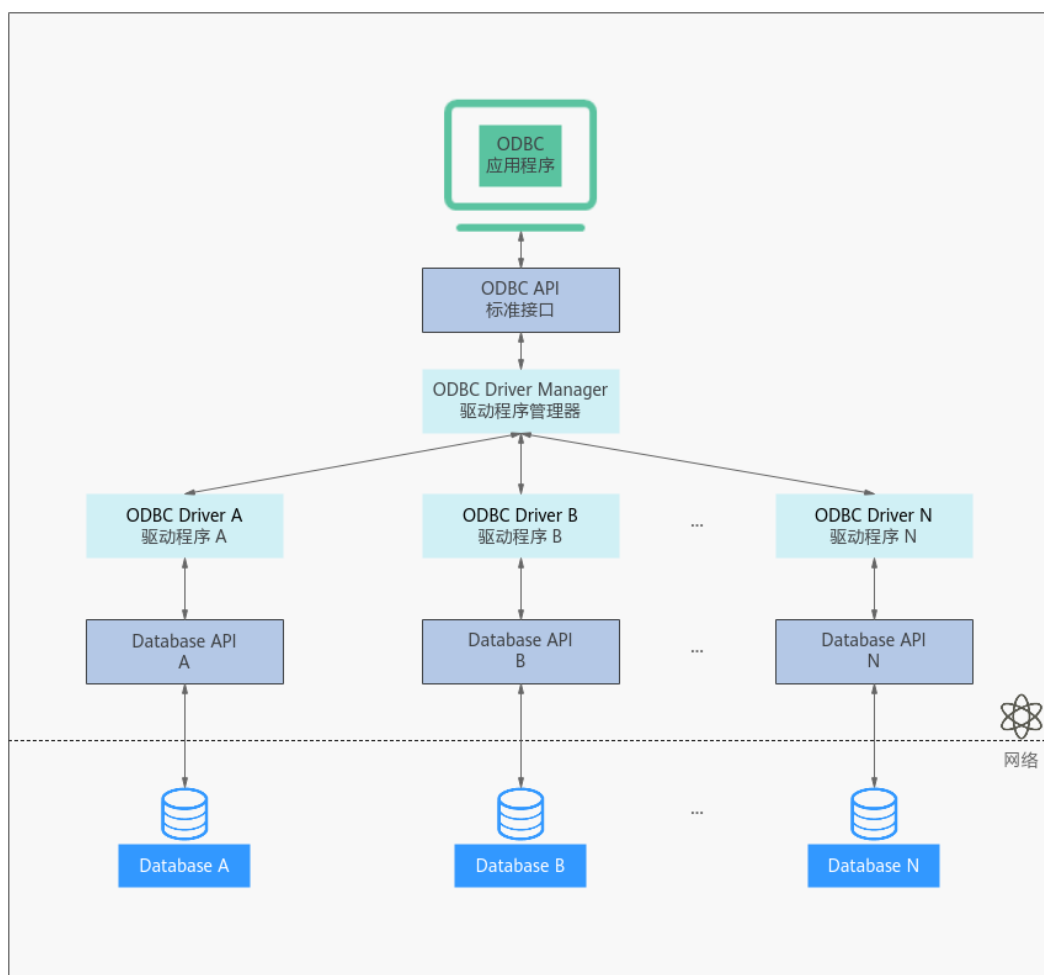
ODBC的核心目标是最大化互操作性：同一套应用程序源码可通过更换驱动程序访问不同的DBMS。应用程序发出的ODBC函数调用由驱动管理程序处理，并在运行期动态加载对应数据库的驱动程序。驱动程序负责将SQL请求转化为特定数据库能够识别的指令，并将执行结果返回给应用程序。这样的抽象机制类似打印机驱动屏蔽硬件差异，使ODBC成为数据库访问的通用层，从而增强了跨平台能力、可维护性和扩展性。

ODBC的架构包括三个主要组成部分：

- 应用程序：负责业务逻辑与发起ODBC调用。
- 驱动程序管理器：负责接收、分发并管理驱动程序。
- 驱动程序：处理ODBC函数调用，将SQL请求提交到特定数据源，并将结果返回到应用程序。

应用程序与驱动程序管理器之间通过标准ODBC API连接，驱动管理器再负责管理ODBC驱动的加载和执行流程。更直观的架构请参考图5-2。

图 5-2 ODBC 系统架构



作为数据库API标准，ODBC与具体DBMS、操作系统及编程语言均保持独立性。ODBC API基于Open Group与ISO/IEC的CLI规范，ODBC 3.x已完全实现该标准。GaussDB目前支持ODBC 3.5，并提供在Linux/Unix以及Windows环境下的驱动使用能力：

- Unix/Linux环境采用unixODBC-2.3.7作为驱动程序管理器。
- Windows系统采用操作系统默认内置的ODBC驱动程序管理器，可在“控制面板 > 管理工具 > 数据源（ODBC）”中进行管理。

Linux/Unix环境支持列表参见[表5-1](#)。

表 5-1 Linux/Unix 环境支持列表

操作系统	平台
EulerOS V2.0SP5	x86_64位
EulerOS V2.0SP9	ARM64位
EulerOS V2.0SP10	x86_64位
EulerOS V2.0SP10	ARM64位
Kylin V10	x86_64位
Kylin V10	ARM64位
UnionTech V20	x86_64位
UnionTech V20	ARM64位
Huawei Cloud EulerOS 2.0	x86_64位
Huawei Cloud EulerOS 2.0	ARM64位
Suse 12.5 说明 在Suse12.5上，默认没有libltdl动态库，需要安装libltdl动态库后， 才可以使用ODBC。	x86_64位
openEuler 22.03 LTS SP4	ARM64位
BCLinux-R8-U2	x86_64位
BCLINUX21.10	x86_64位
REDHAT 8.10	x86_64位
REDHAT 8.10	ARM64位
NFSChina Server 4.0	x86_64位
NFSChina Server 4.0	ARM64位
Ubuntu 22.04 LTS	x86_64位
Ubuntu 22.04 LTS	ARM64位
Suse15.7	x86_64位
Suse15.7	ARM64位
Huawei Cloud EulerOS 3.0	x86_64位
Huawei Cloud EulerOS 3.0	ARM64位

操作系统	平台
Kylin V11	ARM64位

Windows环境支持列表请参见[表5-2](#)。

表 5-2 Windows 环境支持列表

操作系统	平台
Windows 7	x86_32位
Windows 7	x86_64位
Windows Server 2008	x86_32位
Windows Server 2008	x86_64位

须知

在使用ODBC访问GaussDB时，除了标准ODBC行为外，还需特别注意GaussDB当前ODBC驱动的如下若干限制，这些限制可能影响部分SQL功能或数据类型的使用。

- ODBC不支持自定义类型，不支持在存储过程中使用自定义类型参数。
- 当数据库开启proc_outparam_override参数时（即启用GUC参数behavior_compat_options的proc_outparam_override配置项），ODBC无法正常调用带有out参数的存储过程。
- ODBC不支持容灾集群的备机读。
- 当表类型为HASHBUCKET表、SLICEBUCKET表和复制表时或者操作类型为SQL_UPDATE时，ODBC不支持使用可更新游标功能。
- ODBC驱动基于开源版本，在处理TINYINT、SMALLDATETIME、NVARCHAR2等类型时，可能出现类型不兼容的问题。

有关ODBC开发详细教程请参见[基于ODBC开发](#)。

libpq

libpq是GaussDB提供的C语言客户端接口库。是C/C++应用程序访问GaussDB数据库的主要开发接口。

libpq封装了客户端与数据库服务器之间的完整通信协议，为开发者提供了一组功能丰富、性能高效的函数接口。通过libpq，应用程序可以完成数据库连接的建立、管理、SQL语句的执行、结果集的处理、事务控制以及各种高级数据库操作。其采用轻量级设计，适合对性能、并发和资源占用有较高要求的业务场景。同时也是其他几个GaussDB应用程序接口的底层引擎，比如ODBC、Pscopg、ecpg等依赖的库文件。

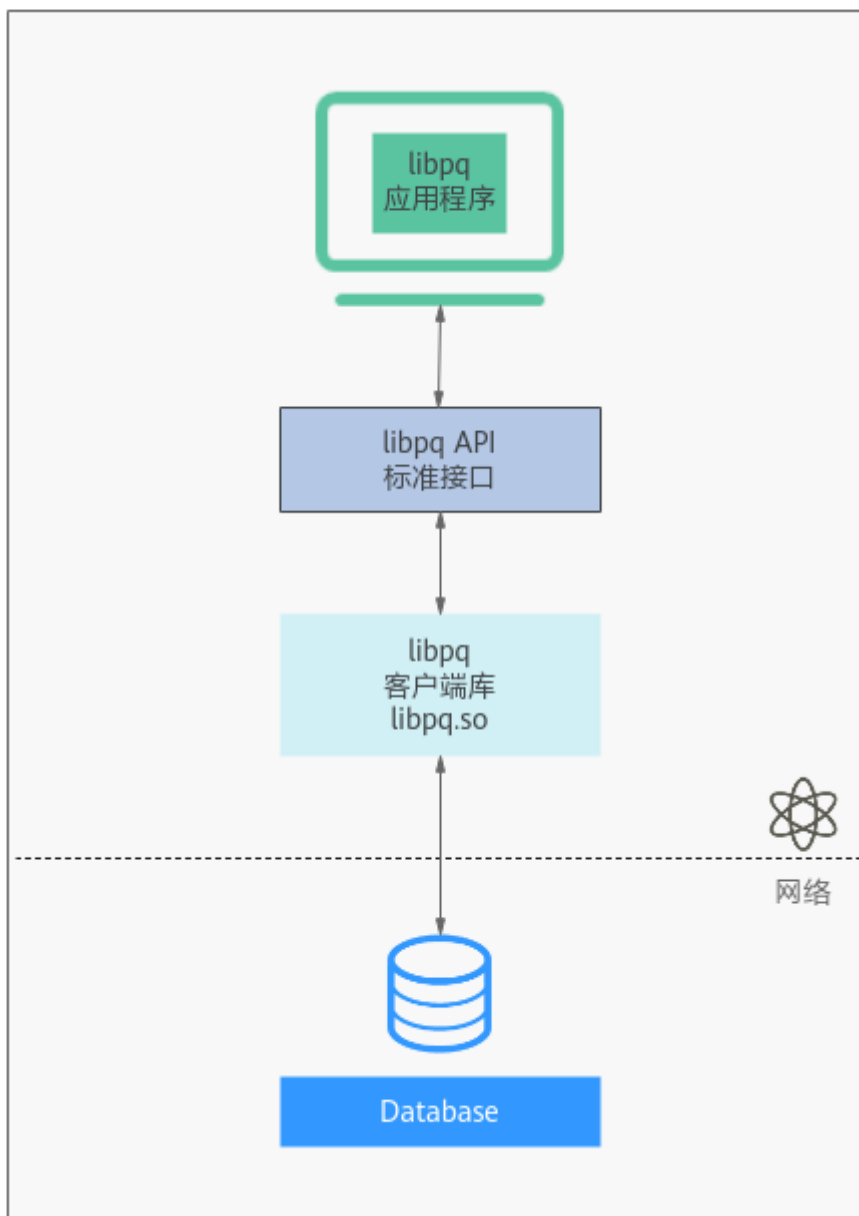
libpq的系统架构主要由以下2个部分组成：

应用程序：开发者编写的C/C++程序，通过调用libpq提供的API完成所有数据库操作。

libpq客户端库：核心接口库（libpq.so），负责API函数的实现、协议封装、连接状态维护、数据编解码以及错误处理。

libpq系统架构图如下图所示：

图 5-3 libpq 系统架构



libpq支持的操作系统平台与GaussDB内核所支持的操作系统平台保持一致，并随数据库内核版本更新而同步更新。

须知

libpq本身不是线程安全的。在多线程应用程序中，建议每个线程使用独立的连接，或通过外部连接池进行管理。

基于libpq开发详细教程请参见[基于libpq开发](#)。

Psycopg

Psycopg是一种用于执行SQL语句的PythonAPI，可以为GaussDB数据库提供统一访问接口，应用程序可基于它进行数据操作。Psycopg2是对libpq的封装，主要使用C语言实现，既高效又安全。它具有客户端游标和服务器端游标、异步通信和通知、支持“COPY TO/COPY FROM”功能。支持多种类型Python开箱即用，适配GaussDB数据类型。通过灵活的对象适配系统，可以扩展和定制适配。Psycopg2兼容Unicode。

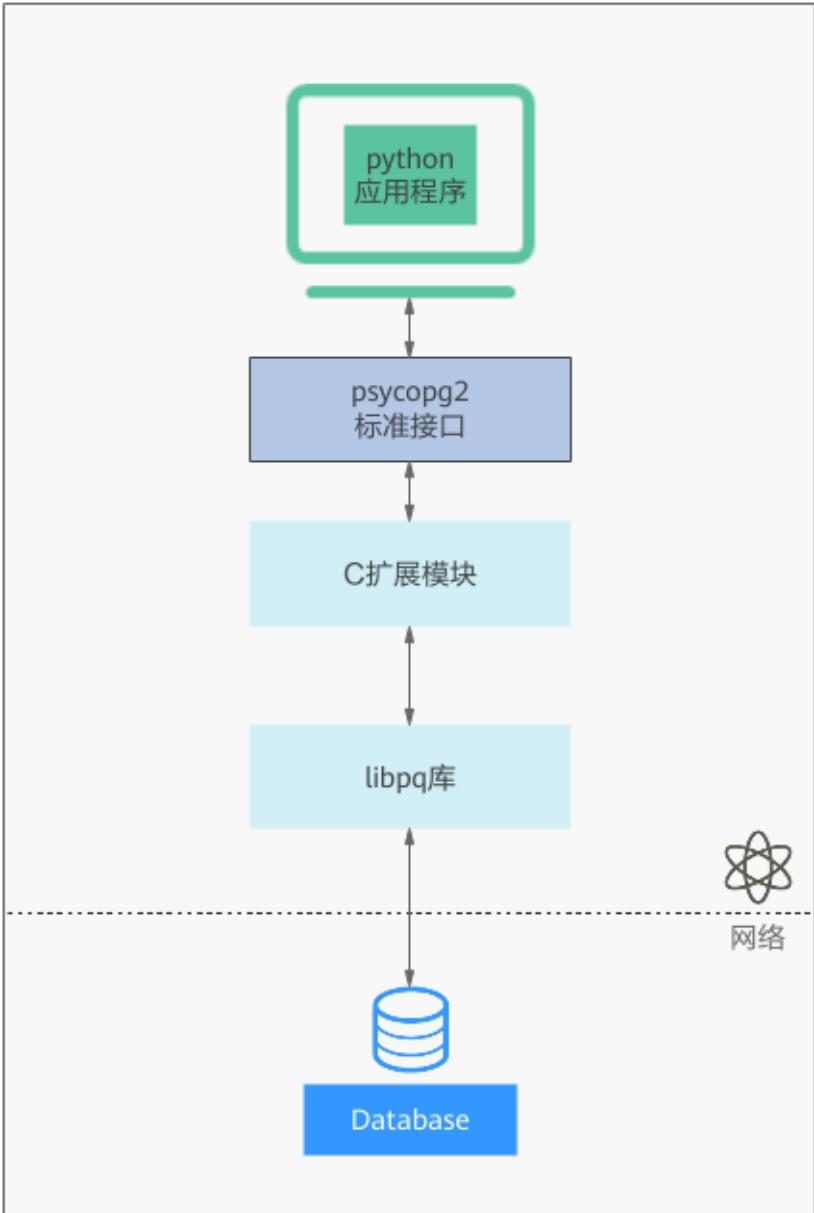
GaussDB数据库提供了对Psycopg2特性的支持，并且支持psycopg2通过SSL模式连接。

Psycopg系统架构主要由以下4个部分组成：

- 应用程序：负责业务逻辑。通过调用标准的Python对象和方法来发起数据库操作请求并接收返回的数据结果。
- psycopg2标准接口：Python数据库API规范（DB-API 2.0）的实现层。核心作用是把Python应用的请求（如执行SQL语句、处理查询结果）规范化，并作为纽带向下传递给C扩展模块。
- C扩展模块：负责将Python的数据类型和对象快速转换成底层的C数据结构，并直接调用libpq的函数。
- libpq库：底层C动态链接库，真正负责与数据库服务端建立TCP连接并收发网络协议包。

Psycopg系统架构如[图5-4](#)所示。

图 5-4 psycopg 系统架构



目前仅支持Linux/Unix环境，请参见表5-3。

表 5-3 Psycopg 支持平台

操作系统	平台	Python版本
EulerOS V2.0SP5	● x86_64位	3.11.4
EulerOS V2.0SP9	● ARM64位	3.11.4
EulerOS V2.0SP10	● ARM64位	3.11.4
EulerOS V2.0SP10	● x86_64位	3.11.4
openEuler 22.03 LTS SP4	● ARM64位	3.11.4

操作系统	平台	Python版本
Huawei Cloud EulerOS 2.0	- ARM64位 - x86_64位	3.11.4
Kylin v10	- ARM64位 - x86_64位	3.11.4
UnionTech20	- ARM64位 - x86_64位	3.11.4
Suse 12.5	x86_64位	3.11.4
BCLinux-R8-U2	x86_64位	3.11.4
REDHAT 8.10	- ARM64位 - x86_64位	3.11.4
BCLINUX 21.10	- ARM64位 - x86_64位	3.11.4
NFSChina Server 4.0	- ARM64位 - x86_64位	3.11.4
Ubuntu 22.04 LTS	- ARM64位 - x86_64位	3.11.4
Suse 15.7	- ARM64位 - x86_64位	3.11.4
Huawei Cloud EulerOS 3.0	- ARM64位 - x86_64位	3.11.4
Kylin v11	ARM64位	3.11.4

须知

psycpg2在编译过程中，会链接（link）GaussDB的openssl，GaussDB的openssl与操作系统自带的openssl可能不兼容。如果遇到不兼容现象，例如提示"version 'OPENSSL_1_1_1f' not found"，请使用环境变量LD_LIBRARY_PATH进行隔离，以避免混用操作系统自带的openssl与GaussDB依赖的openssl。

例如，在执行某个调用psycpg2的应用软件client.py时，将环境变量显性赋予该应用软件：

```
export LD_LIBRARY_PATH=/path/to/psycpg2/lib:$LD_LIBRARY_PATH python client.py
```

其中，/path/to/psycpg2/lib表示GaussDB依赖的openssl库所在目录，需根据文件实际存储路径修改。

基于Psycpg开发详细教程请参见[基于Psycpg开发](#)。

Go

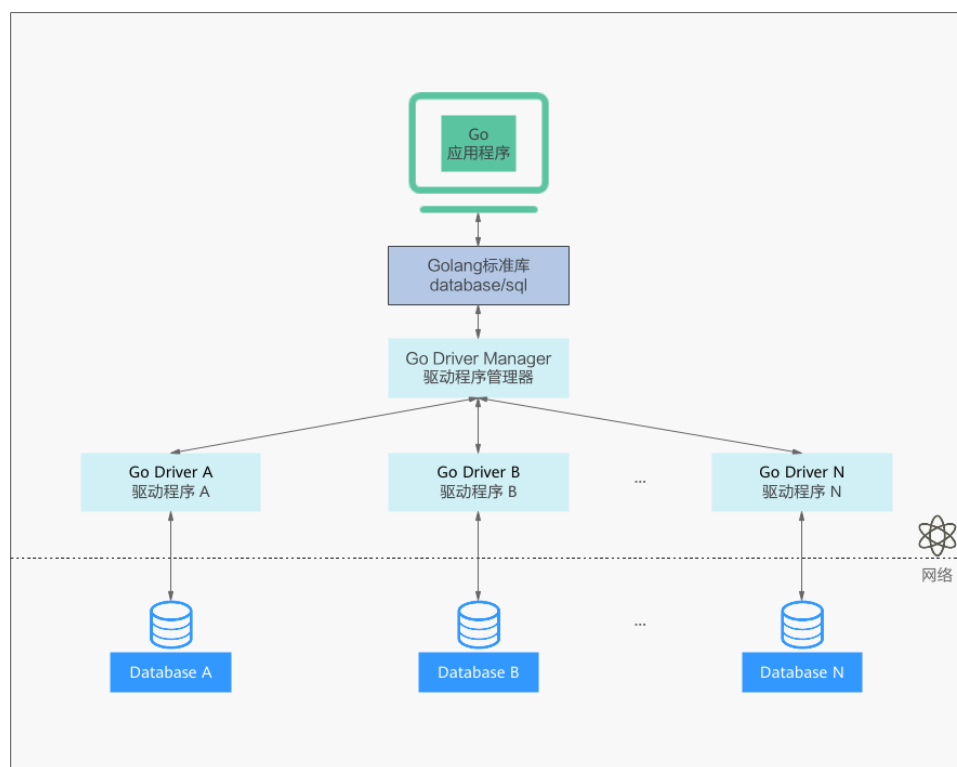
Go驱动是一个用于连接和操作GaussDB数据库的Go语言驱动。它提供了一些用于连接和操作GaussDB数据库的接口，可以用于执行查询、插入、更新和删除等操作。

Go架构主要包括以下组件：

- 应用程序：业务逻辑的载体，负责发起数据库操作请求（如SQL查询、事务处理等）。
- database/sql：Go标准库提供的抽象层，定义统一的数据库操作接口（如sql.Open、type DB等）。
- 驱动程序管理器：加载和管理Go驱动程序，根据调用database/sql中sql.Open接口的driverName参数以匹配相应驱动。
- 驱动程序：实现database/sql数据库操作接口，将Go调用转换为数据库协议，发送SQL到数据库执行，并将结果返回到应用程序。

Go系统架构如图5-5所示。

图 5-5 Go 系统架构



以下是Go驱动当前支持的部分功能及其用法：

1. 数据库连接：Go驱动支持通过db.Open连接数据库。连接串支持URL和DSN两种格式。
2. SQL执行与查询：Go驱动提供接口如db.Exec、db.Query等，可以用于SQL的执行与查询。
3. 事务管理：Go驱动支持Tx接口进行事务管理，其中实现了启动、提交、回滚事务等方法，确保数据库操作的一致性和完整性。

4. 元数据访问：Go驱动提供ColumnType接口用于查询数据库中列属性信息，包括是否可以为空、数据库类型名称、长度和小数位数等信息。
5. 可重用性：Go驱动提供Prepare方法用于预编译SQL，该SQL可以被多次执行，只需改变传入参数，提高数据库可重用性。

基于Go驱动开发详细教程请参见[基于Go驱动开发](#)。

ecpg

ecpg（embedded SQL C preprocessor for GaussDB）是一种用于C语言程序的嵌入式SQL预处理器，旨在为C语言程序提供直接集成SQL能力的标准化手段。ecpg允许开发者在C源代码中以EXEC SQL为前缀直接嵌入标准SQL语句。

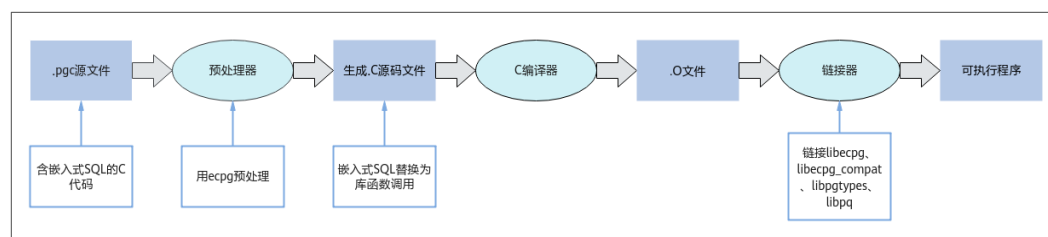
这种特殊代码形式通常如下：

```
EXEC SQL ...;
```

这些语句在语法上取代了一个C语句，可以出现在全局或者是具体函数内部。其核心价值在于通过预处理技术隐藏了底层通信及协议处理的复杂性，实现了宿主语言（C）与数据库语言（SQL）在语法层面的深度融合。

ecpg的实现机制遵循“预处理 -> 编译 -> 链接”的典型分层架构。其核心工作流如下图5-6所示：

图 5-6 ecpg 架构



如上图所示，一个嵌入式SQL程序由一种普通编程语言编写的代码（此处为C语言）和SQL命令共同组成。要构建该程序，源代码（*.pgc）首先通过嵌入式SQL预处理器，将源代码转换成一个普通C语言程序（*.C文件），然后再通过编译器处理，将嵌入式逻辑转化为.O文件。转换过的ecpg应用通过嵌入式SQL库（ecpglib）调用libpq库中的函数，与GaussDB服务器使用普通的前端/后端协议通信。

ecpg的核心优势如下：

- 开发效率：通过SQL与C的深度融合，消除繁琐的API调用逻辑，代码量更少，维护更直观。
- 数据安全：ecpg的预处理器能够自动处理C程序与数据库之间的数据交换逻辑，确保数据类型在传输过程中的精确映射。这种机制有效规避了手动处理数据转换时的潜在风险，并通过参数化处理天然抵御SQL注入攻击。
- 提前预警：不同于运行时才触发报错的传统API，ecpg预处理器能够对嵌入的SQL语句进行静态语法分析。这使得开发者在构建阶段即可识别大部分语法错误，从而降低了调试成本并提升了软件的鲁棒性。

ecpg程序在编写时需遵循特定的语法规则，以区分SQL逻辑与C语言逻辑：

- 嵌入式SQL语句遵循普通SQL代码的大小写敏感规则，并且允许嵌套的C语言代码风格注释（SQL标准的一部分）。

- 程序的C语言部分遵循C语言程序的标准，不支持嵌套注释。

基于ecpg开发详细教程请参见[基于ecpg开发](#)。

5.2 开发规范

如果用户在APP的开发过程中，使用了连接池机制，那么需要遵循如下规范：

- 如果在连接中设置了GUC参数，那么在将连接归还连接池之前，必须使用以下语句将连接的状态清空。

```
SET SESSION AUTHORIZATION DEFAULT;  
RESET ALL;
```

- 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

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

应用程序开发驱动兼容性说明如[表5-4](#)所示。

表 5-4 兼容性说明

驱动	兼容性说明
JDBC、Go、ODBC、libpq、Pycopg、ecpg	驱动前向兼容数据库，若需使用驱动与数据库同步增加的新特性，须升级数据库。

须知

- 原则上，兼容性参数应在创建数据库后就设置，不应在使用过程中来回切换。
- 涉及使用以下场景的特性需要配合将JDBC驱动进行升级。对于非透明多写特性，需要升级到503.1及以上的配套版本。
 1. 通过JDBC使用record、array、tableof嵌套自定义类型。
 2. 开启s2兼容性参数，设置sessiontimezone的合法性校验。
- 旧版本驱动连接新版本数据库的情况下，若涉及新特性可能导致异常行为，建议驱动和数据库使用配套版本。
- 在多线程环境下使用驱动：JDBC驱动程序线程不是安全的，无法保证连接上的方法同步。由调用者来同步对驱动程序的调用。

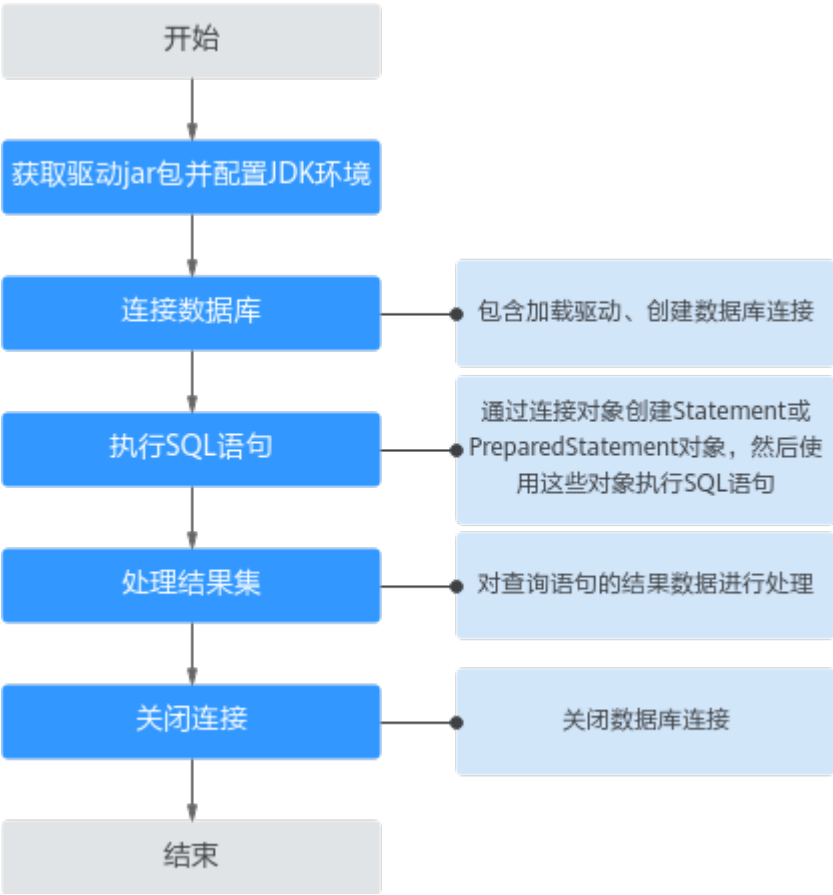
5.3 基于 JDBC 开发

5.3.1 开发流程

基于JDBC开发应用程序的流程，包含获取驱动jar包并配置JDK环境、连接数据库、执行SQL语句、处理结果集、关闭连接等几个部分。

基于JDBC开发应用程序的流程，请参见[图5-7](#)。

图 5-7 采用 JDBC 开发应用程序的流程



5.3.2 开发步骤

5.3.2.1 获取驱动 jar 包并配置 JDK 环境

环境准备包括获取驱动jar包、配置JDK环境。

获取驱动包

下载[表5-5](#)中的驱动包和驱动包校验包。

表 5-5 驱动包下载列表

版本	下载地址
V2.0-10.x	驱动包 驱动包校验包

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：

a. 上传软件包和软件包校验包到虚拟机的同一目录下。

b. 执行如下命令，校验软件包完整性。

```
F=$(echo DBS-GaussDB-driver*.tar.gz);echo "$(awk '{print $1}' "$F.sha256") $F" | sha256sum -c -
```

如果回显OK，则校验通过。例如：

```
DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz: OK
```

● Windows操作系统软件包完整性校验：

a. 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。

b. 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

■ {驱动包本地目录}：请根据实际下载目录进行替换。例如：C:\Users

■ {驱动包名}：请根据实际下载的驱动包名进行替换。例如：DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz

示例：certutil -hashfile C:\Users\DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz sha256

c. 将b获取到的Hash值和表5-5中获取到的驱动包校验包的Hash值进行比较。

■ 若一致则通过校验。

■ 若不一致，请重新下载驱动包，重复a~c进行校验。

获取驱动 jar 包

以下介绍了获取驱动jar包的步骤：

步骤1 解压获取到的驱动包。该包包含多种不同的驱动jar包，可根据后续步骤确定所需驱动jar包的文件路径。驱动jar包的文件层级关系参考以下结构。

DBS-GaussDB-driver_*/数据库部署形态/驱动类型/操作系统架构/发布包/驱动jar包

步骤2 选择数据库部署形态。

JDBC驱动包不区分部署形态，可选择Centralized/Distributed任何一种数据库部署形态对应的驱动jar包。

步骤3 选择驱动类型。

选择jdbc_driver.tar.gz，并解压。

步骤4 选择操作系统架构。

非透明多写特性下：

1. 非密态功能不依赖系统架构，可选择任何一种操作系统架构。

2. 密态功能需要严格选择对应系统架构的目录，可使用以下命令查看客户端系统架构。

```
uname -a
```

步骤5 选择发布包。

使用以下命令查看当前环境的JDK版本，然后选择与当前环境JDK适配的发布包。

```
java -version
```

- JDK8发布包中包含适配JDK1.8的驱动jar包。
- JDK17发布包中包含适配JDK17的驱动jar包。
- JDK21发布包中包含适配JDK21的驱动jar包。

步骤6 用户根据自身业务需求选择对应的驱动包。

说明

- 推荐用户在非密态场景下使用gaussdbjdbc.jar，在密态场景下使用gscejdbc.jar。opengaussjdbc-ce.jar、opengaussjdbc.jar、gsjdbc4.jar和gsjdbc200.jar后续将不再维护。
- 非密态场景下，gaussdbjdbc.jar兼容opengaussjdbc.jar、gsjdbc4.jar和gsjdbc200.jar，用户使用gaussdbjdbc.jar替换opengaussjdbc.jar、gsjdbc4.jar或gsjdbc200.jar时，无需修改jdbc相关业务代码。
- 密态场景下，gscejdbc.jar兼容opengaussjdbc-ce.jar，用户使用gscejdbc.jar替换opengaussjdbc-ce.jar时，无需修改jdbc相关业务代码。

解压发布包可获取不同兼容性的驱动jar包，除了gscejdbc.jar、opengaussjdbc-ce.jar之外的所有驱动jar包功能一致，只有Driver主类名、连接串前缀及内部类路径有所区别，具体参见表5-6。

表 5-6 驱动 jar 包介绍

驱动包	连接串前缀	Driver主类名	适用场景
gaussdbjdbc.jar	jdbc:gaussdb	com.huawei.gaussdb.jdbc.Driver	本章的Java代码示例默认使用gaussdbjdbc.jar包。非密态数据库场景下，优先推荐使用此驱动包。
opengaussjdbc.jar	jdbc:opengauss	com.huawei.opengauss.jdbc.Driver	遇到同一JVM进程内需要同时访问PG及GaussDB的场景，请使用此驱动包。
gsjdbc4.jar	jdbc:postgresql	org.postgresql.Driver	此驱动包适用于从PG数据库迁移业务的场景，驱动类和加载路径与迁移前保持一致，但接口支持情况不完全一致，未支持的接口需要业务侧进行调整。
gsjdbc200.jar	jdbc:gaussdb	com.huawei.gauss200.jdbc.Driver	该驱动包适用于从Gauss200迁移业务的场景，驱动类和加载路径与迁移前保持一致，但接口支持情况不完全一致，未支持的接口需要业务侧进行调整。其余与gsjdbc4.jar相同。
gscejdbc.jar	jdbc:gaussdb	com.huawei.gaussdb.jdbc.Driver	此驱动包包含密态数据库需要加载的加解密相关的依赖库，因此密态数据库场景下推荐使用此驱动包。目前仅支持Euler和HCE操作系统。

驱动包	连接串前缀	Driver主类名	适用场景
opengaussjdbc- ce.jar	jdbc:opengauss	com.huawei.opengauss.jdbc.Driver	该驱动包包含密态数据库需要加载的加解密相关的依赖库，url前缀为“jdbc:opengauss”的密态数据库场景下推荐使用此驱动包。目前仅支持Euler和HCE操作系统。

须知

- 不能使用GaussDB的驱动包操作PG数据库，虽然部分版本能成功连接，但部分接口行为与PG JDBC不兼容，可能导致未知错误。
- 不能使用PG的驱动包操作GaussDB数据库，虽然部分版本能成功连接，但部分接口行为与GaussDB JDBC不兼容，可能导致未知错误。
- 使用gscejdbc.jar驱动包时，需要先设置环境变量LD_LIBRARY_PATH，具体设置方式请参见[使用JDBC操作密态数据库](#)。

步骤7 在运行环境中添加驱动jar包。

例如在非密态数据库场景下，用户使用IDE进行开发，则在本地IDE添加gaussdbjdbc.jar包（优先推荐）。

----结束

配置 JDK 环境

客户端需配置JDK环境，JDK支持Windows、Linux等多种平台。本章节以配置JDK1.8和Windows11平台为例，给出具体步骤。

步骤1 在DOS窗口（Windows下的命令提示符）输入以下命令查看JDK版本。

```
java -version
```

确认是否已安装JDK1.8；如未安装，请从[官方网站](#)下载并安装。

步骤2 根据以下步骤配置系统环境变量。

1. 在“此电脑”单击右键，选择“属性”。
2. 在“系统”页面，选择“系统信息”，单击“高级系统设置”。
3. 在“系统属性”页面，“高级”页签上单击“环境变量”。
4. 在“环境变量”页面，在“系统变量”区域单击“新建”或“编辑”配置系统变量。变量说明如[表5-7](#)所示。

表 5-7 变量说明

变量名	操作	变量值
JAVA_HOME	- 若存在，则单击“编辑”。 - 若不存在，则单击“新建”。	JAVA的安装目录。 例如：C:\Program Files\Java\jdk1.8.0_131
Path	单击“编辑”。	- 若配置了JAVA_HOME，则在变量值的最前面加上： %JAVA_HOME%\bin - 若未配置JAVA_HOME，则在变量值的最前面加上JAVA安装的全路径： C:\Program Files\Java\jdk1.8.0_131\bin
CLASSPATH	单击“新建”。	%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar

----结束

5.3.2.2 连接数据库

5.3.2.2.1 连接介绍

在获取驱动jar包并配置JDK环境的情况下，通过JDBC连接数据库时需先加载驱动。本节主要介绍驱动加载方式、连接接口以及连接方式。

驱动加载方式

驱动加载有两种方式：

- 在代码中创建连接之前的任意位置装载：
`Class.forName("com.huawei.gaussdb.jdbc.Driver");`
- 在JVM启动时通过参数传递，“jdbctest”为测试用例程序的名称：
`java -Djdbc.drivers=com.huawei.gaussdb.jdbc.Driver jdbctest`

📖 说明

- 由于gsjdbc4.jar包与PG的JDBC驱动包类路径一致，若在同一进程内同时使用gsjdbc4.jar包和PG的JDBC驱动包，可能会导致类名冲突。
- 本版本JDBC不再支持IAM认证功能。
- GaussDB JDBC驱动主要提供以下增强特性：
 - 支持SHA256登录。
 - 支持SM3登录，需自行配置SM3加密所需插件。例如JDK8，可按下列示例在pom.xml中配置。

```
<dependency>  
<groupId>org.bouncycastle</groupId>  
<artifactId>bcprov-jdk18on</artifactId>  
<version>1.75</version>  
<scope>compile</scope>  
</dependency>
```
 - 支持对接实现SLF4J接口的第三方日志框架。
 - 支持连接级别的分布式负载均衡。
 - 支持容灾切换。

连接接口

JDBC提供了三种接口，用于创建数据库连接，请参见表5-8。

表 5-8 数据库连接接口

三 种 接 口	方法签名	在非透明多写特性下的相关描述
接 口 一	DriverManager.getConnection(String url)	此方式需在url中写入数据库用户名和密码，存在安全风险，因此不推荐使用。
接 口 二	DriverManager.getConnection(String url, String user, String password)	数据库用户名和密码单独传参，具体请参见采用接口二连接数据库（非透明多写）。
接 口 三	DriverManager.getConnection(String url, Properties info)	数据库用户名和密码通过info传参，具体请参见采用接口三连接数据库（非透明多写）。

关于url、info、user、password参数的描述，请参见表5-9。

表 5-9 数据库连接参数

参数	描述
url	使用gaussdbjdbc.jar驱动包连接数据库时的连接描述符。 host为服务器名称或IPv4时，格式如下： • jdbc:gaussdb: • jdbc:gaussdb:database • jdbc:gaussdb://host/database • jdbc:gaussdb://host:port/database • jdbc:gaussdb://host:port/database?param1=value1&param2=value2 • jdbc:gaussdb://host1:port1,host2:port2/database?param1=value1&param2=value2 host为IPv6时，格式如下： • jdbc:gaussdb: • jdbc:gaussdb:database • jdbc:gaussdb://host/database或jdbc:gaussdb://[host]/database • jdbc:gaussdb://[host]:port/database • jdbc:gaussdb://[host]:port/database?param1=value1&param2=value2 • jdbc:gaussdb://[host1]:port1,[host2]:port2/database?param1=value1&param2=value2 说明 • database为要连接的数据库名称。数据库名称缺省则与用户名一致。 • host在非透明多写特性下为数据库服务器名称或IP地址。同时支持IPv4和IPv6。在非透明多写特性下，由于安全原因，数据库CN禁止集群内部其他节点无认证接入。如果要在集群内部访问CN，请将JDBC程序部署在CN所在机器，host使用"127.0.0.1"。否则可能会上报“FATAL: Forbid remote connection with trust method!”错误。 建议业务系统单独部署在集群外部，否则可能会影响数据库运行性能。 缺省情况下，连接服务器为localhost。 • port在非透明多写特性下为数据库服务器端口。 缺省情况下，会尝试连接5432端口。 • 当host为IPv6且在url中指定port时，需要通过“[]”分隔IP，如：[IP]:port。 • param为参数名称，即数据库连接属性。参数可以配置在url中，以“?”开始配置，使用“=”给参数赋值，使用“&”分隔不同参数。 • value为参数值，即数据库连接属性值。 • 连接时需配置connectTimeout、socketTimeout，如果未配置，默认为0，即不会超时。在服务端与客户端出现网络故障时，客户端一直未收到服务端ACK确认报文，会启动超时重传机制，不断地进行重传。当重传次数达到默认的15次会报超时错误，会导致RTO很高。 • 使用JDBC标准接口建立连接时，请确保url格式的合法性。若url格式不合法，会导致连接失败，且报错信息中包含原始url字符串，可能造成敏感信息泄露。 • url中不推荐配置密码、凭证等敏感信息，建议通过Properties连接属性来设置敏感信息，详细操作请参见采用接口三连接数据库（非透明多写）。

参数	描述
info	数据库连接属性。全量参数请参见 连接参数参考 。 须知 url和info同时指定一个参数时，url传入的参数优先生效。
user	数据库用户名。
password	数据库用户的密码。

连接方式介绍

连接数据库的方式通常根据安全性、性能、可扩展性和实现复杂度等因素进行选择。各种连接方式的优缺点，如[表5-10](#)所示。

表 5-10 连接方式介绍

连接方式	优点	缺点	选择原则
非加密方式	- 简单易实现，无需额外的加密和解密操作。 - 性能较高，因为不需要进行加密/解密操作，减少了数据传输的开销。	- 安全性较低，数据在传输过程中容易被窃听或篡改。 - 不适合传输敏感信息，可能违反隐私保护法规。	只适用于内部网络或对安全性要求不高的场景。
SSL方式	- 提供了加密传输，数据在传输过程中得到保护，安全性较高。 - 支持客户端/服务器身份验证，增强了安全性。	- 配置和管理相对复杂，需要证书的颁发和更新。 - 在加密和解密大量数据时，可能会增加一些性能开销。	适用于对数据传输安全性要求较高的场景，如金融、医疗等行业。
UDS方式	- 适用于本地通信，不经过网络，传输速度快。 - 不需要额外的网络开销，不容易受到网络攻击。	仅限于在同一台主机上的进程间通信，无法用于远程连接。	适用于本地通信的场景，如本地服务和应用之间的数据交换。

连接方式	优点	缺点	选择原则
国密TLS方式	- 支持国密算法，符合国内对加密算法的法律要求。 - 提供了与标准TLS相当的安全性，适用于对加密标准有特殊要求的场景。	- 在国际范围内的应用可能受限，因为其他国家可能不支持国密算法。 - 可能存在性能开销，因为国密算法相对于传统的加密算法可能更为复杂。	适用于国内的特定行业，如金融等领域，对国内加密标准有法律要求的场景。

建立连接的首包介绍

驱动与数据库建立连接过程中，Socket（套接字）建立之后，由驱动以key value形式向数据库发送的第一个报文为首包。首包内容包括用户名（user）、数据库名（database）以及部分GUC参数信息（client_encoding、DateStyle、application_name、TimeZone、extra_float_digits、gaussdb.capacity_version、replication、search_path、enable_full_encryption、session_type、uppercase_attribute_name）。GUC参数的具体介绍，请参见《参考》中“数据库运行参数说明 > GUC参数说明”。

须知

- gaussdb.capacity_version参数表示驱动支持的兼容性能力集的版本号，用于驱动与数据库协商此次建立连接时使用的兼容性特性集合。该参数为内部参数。
- 数据库执行以下命令后，上述首包GUC参数的值将被更改为首包设置的值。
reset all;

5.3.2.2.2 以非加密方式连接

采用接口二连接数据库（非透明多写）

前置条件：在运行环境（如IDE）已添加gaussdbjdbc.jar包。

使用DriverManager.getConnection(String url, String user, String password)接口创建数据库连接的步骤如下：

- 步骤1** 导入java.sql.Connection和java.sql.DriverManager。
- java.sql.Connection是数据库连接接口，通过java.sql.DriverManager的getConnection()方法让应用程序连接到数据库。此外，用户需要根据实际的应用场景，再导入其他的接口和类，具体请参见[JDBC接口参考](#)。
- ```
import java.sql.Connection;
import java.sql.DriverManager;
```
- 步骤2** 指定数据库sourceURL（\$ip、\$port和database需要用户自行修改）、用户名和密码。
- 用户名和密码直接写到代码中存在较大安全风险，建议存放在环境变量中。
- ```
String sourceURL = "jdbc:gaussdb://$ip.$port/database";
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
```

步骤3 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
String driver = "com.huawei.gaussdb.jdbc.Driver";  
Class.forName(driver);
```

步骤4 创建数据库连接。

调用DriverManager.getConnection(String url, String user, String password)，进行数据库连接。

```
Connection conn = DriverManager.getConnection(sourceURL, userName, password);
```

----结束

采用接口三连接数据库（非透明多写）

前置条件：在运行环境（如IDE）已添加gaussdbjdbc.jar包。

使用DriverManager.getConnection(String url, Properties info)接口创建数据库连接的步骤如下：

步骤1 导入java.sql.Connection、java.sql.DriverManager和java.util.Properties。

java.util.Properties的setProperty()方法，用于设置Properties对象的属性值。此外，用户需要根据实际的应用场景，再导入其他的接口和类，具体请参见[JDBC接口参考](#)。

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.util.Properties;
```

步骤2 指定数据库sourceURL（\$ip、\$port和database需要用户自行修改）、用户名和密码。

用户名和密码直接写到代码中存在较大安全风险，建议存放在环境变量中。

```
String sourceURL = "jdbc:gaussdb://$ip.$port/database";  
String userName = System.getenv("EXAMPLE_USERNAME_ENV");  
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
```

步骤3 创建Properties对象，并将userName和password设置为该对象的属性值。

```
Properties info = new Properties();  
info.setProperty("user", userName);  
info.setProperty("password", password);
```

步骤4 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
String driver = "com.huawei.gaussdb.jdbc.Driver";  
Class.forName(driver);
```

步骤5 创建数据库连接。

调用DriverManager.getConnection(String url, Properties info)，进行数据库连接。

```
Connection conn = DriverManager.getConnection(sourceURL, info);
```

----结束

5.3.2.2.3 以 SSL 方式连接

用户通过JDBC连接GaussDB服务器时，可以通过开启SSL加密客户端和服务端之间的通信，为敏感数据在网络上的传输提供一种安全保障手段。本小节中，连接数据库采用接口三：DriverManager.getConnection(String url, Properties info)。

使用证书认证

前置条件：

- 在运行环境（如IDE）已添加gaussdbjdbc.jar包。
- 登录GaussDB管理控制台，在“实例管理”页面，单击实例名称进入“基本信息”页面，单击“SSL”处的[↓](#)，下载根证书或捆绑包，将根证书ca.pem放置在客户端。

使用客户端配置证书的方式连接数据库的命令如下：

步骤1 导入java.sql.Connection、java.sql.DriverManager、java.util.Properties。

此外，用户需要根据实际的应用场景，再导入其他的接口和类，详见[JDBC接口参考](#)。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
```

步骤2 指定数据库sourceURL（\$ip、\$port、database需要用户自行修改）、用户名和密码。

用户名和密码直接写到代码中有很大的安全风险，建议在环境变量中存放。

```
String sourceURL = "jdbc:gaussdb://$ip:$port/database";
Properties urlProps = new Properties();
urlProps.setProperty("user", System.getenv("EXAMPLE_USERNAME_ENV"));
urlProps.setProperty("password", System.getenv("EXAMPLE_PASSWORD_ENV"));
```

步骤3 设置ssl属性值为true，在客户端配置ca.pem根证书。

```
urlProps.setProperty("ssl", "true");
urlProps.setProperty("sslrootcert", "ca.pem");
```

步骤4 配置sslmode。

sslmode设置值：require、verify-ca、verify-full，参数介绍详见[sslmode](#)。客户根据应用场景选择一种即可。

```
urlProps.setProperty("sslmode", "require"); // 设置sslmode为require
// urlProps.setProperty("sslmode", "verify-ca"); // 设置sslmode为verify-ca
// urlProps.setProperty("sslmode", "verify-full"); // sslmode为verify-full（Linux下验证）
```

步骤5 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
Class.forName("com.huawei.gaussdb.jdbc.Driver");
```

步骤6 创建数据库连接。

调用DriverManager.getConnection(String url, Properties info)，进行数据库连接。
Connection conn = DriverManager.getConnection(sourceURL,urlProps);

----结束

5.3.2.2.4 以 UDS 方式连接

Unix Domain Socket（UDS）用于同一主机上不同进程间的数据交换，需添加unixsocket以获取套接字工厂。

前置条件：在运行环境（如IDE）添加gaussdbjdbc.jar、unixsocket-core-XXX.jar、unixsocket-common-XXX.jar、unixsocket-native-common-XXX.jar。XXX为版本号，后面三个jar包版本号需要一致。

以UDS方式连接数据库的步骤如下：

步骤1 导入java.sql.Connection、java.sql.DriverManager、java.util.Properties。

此外，用户需要根据实际的应用场景，再导入其他的接口和类，详见[JDBC接口参考](#)。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
```

步骤2 指定数据库的用户名和密码，并将其设置为Properties对象的属性值。

用户名和密码直接写到代码中存在较大安全风险，建议存放在环境变量中。

```
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
Properties properties = new Properties();
properties.setProperty("user", userName);
properties.setProperty("password", password);
```

步骤3 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
String driver = "com.huawei.gaussdb.jdbc.Driver";
Class.forName(driver);
```

步骤4 指定数据库的\$ip、\$port、database、socketFactory和socketFactoryArg。

\$ip和\$port需要用户自行修改，连接主机名database必须设置为“localhost”，socketFactory设置为org.newsclub.net.unix.AFUNIXSocketFactory\$FactoryArg，socketFactoryArg设置为[path-to-the-unix-socket]，实现以UDS方式连接数据库。socketFactory和socketFactoryArg参数的具体说明，详见[socketFactory](#)和[socketFactoryArg](#)。

```
Connection conn = DriverManager.getConnection("jdbc:gaussdb://$ip:$port/database?
socketFactory=org.newsclub.net.unix" +
".AFUNIXSocketFactory$FactoryArg&socketFactoryArg=[path-to-the-unix-socket]", properties);
System.out.println("Connection Successful!");
```

说明

socketFactoryArg参数需要根据真实路径进行配置，与GUC参数unix_socket_directory的值保持一致。

----结束

5.3.2.2.5 以国密 TLS 方式连接

用户通过JDBC连接GaussDB服务器时，可以通过开启国密TLS加密客户端和服务端之间的通信，为敏感数据在网络上的传输提供高强度的安全保障手段。以国密TLS方式连接数据库有两种方式：国密TLS单向认证、国密TLS双向认证。单向认证只需客户端验证服务端身份，而双向认证则需要客户端和服务端双方互相验证身份。

方式一：国密 TLS 单向认证

前置条件：

- 在运行环境（如IDE）已添加gaussdbjdbc.jar包。
- 用户已经获取服务端所需要的国密TLS证书和私钥文件，并完成服务端配置；同时已经获取客户端所需要的cacert.pem根证书。

📖 说明

- 关于证书生成和获取，请联系管理员。
- 服务端配置证书的具体操作，请联系管理员。

使用国密TLS单向认证的方式连接数据库的步骤如下：

步骤1 导入java.sql.Connection、java.sql.DriverManager、java.util.Properties。

此外，用户需要根据实际的应用场景，再导入其他的接口和类，详见[JDBC接口参考](#)。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
```

步骤2 指定数据库urls（\$ip、\$port、database需要用户自行修改）、用户名和密码。

从环境变量EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV获取用户名和密码，并将其设置为Properties对象的属性值。

```
String urls = "jdbc:gaussdb://$ip.$port/database";
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
Properties urlProps = new Properties();
urlProps.setProperty("user", userName);
urlProps.setProperty("password", password);
```

步骤3 设置ssl属性值为true，设置sslmode为verify-ca。

```
urlProps.setProperty("ssl", "true");
urlProps.setProperty("sslmode", "verify-ca");
```

步骤4 国密TLS单向认证：在客户端配置cacert.pem根证书，sslgmccipher仅允许配置为ECC_SM4_SM3。

sslrootcert和sslgmccipher参数的具体说明，详见[sslrootcert](#)和[sslgmccipher](#)。

```
urlProps.setProperty("sslrootcert", "cacert.pem");
urlProps.setProperty("sslgmccipher", "ECC_SM4_SM3");
```

步骤5 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
Class.forName("com.huawei.gaussdb.jdbc.Driver");
```

步骤6 创建数据库连接。

调用DriverManager.getConnection(String url, Properties info)，进行数据库连接。

```
Connection conn = DriverManager.getConnection(urls,urlProps);
```

----结束

方式二：国密 TLS 双向认证

前置条件：

- 在运行环境（如IDE）已添加gaussdbjdbc.jar包。
- 用户已经获取服务端所需要的国密TLS证书和私钥文件，并完成服务端配置；同时已经获取客户端所需要的client.key.pk8、client_enc.key.pk8、client.crt、client_enc.crt、cacert.pem证书与私钥文件。

📖 说明

- 关于证书生成和获取，请联系管理员。
- 服务端配置证书的具体操作，请联系管理员。

使用国密TLS双向认证的方式连接数据库的步骤如下：

步骤1 导入java.sql.Connection、java.sql.DriverManager、java.util.Properties。

此外，用户需要根据实际的应用场景，再导入其他的接口和类，详见[JDBC接口参考](#)。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
```

步骤2 指定数据库urls（\$ip、\$port、database需要用户自行修改）、用户名和密码。

从环境变量EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV获取用户名和密码，并将其设置为Properties对象的属性值。

```
String urls = "jdbc:gaussdb://$ip.$port/database";
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
Properties urlProps = new Properties();
urlProps.setProperty("user", userName);
urlProps.setProperty("password", password);
```

步骤3 设置ssl属性值为true，设置sslmode为verify-ca。

```
urlProps.setProperty("ssl", "true");
urlProps.setProperty("sslmode", "verify-ca");
```

步骤4 国密TLS双向认证：在客户端配置client.key.pk8、client_enc.key.pk8、client.crt、client_enc.crt、cacert.pem证书。sslsmciphper参数可以配置为ECC_SM4_SM3或者ECDHE_SM4_SM3，具体的参数说明详见[sslsmciphper](#)。

```
urlProps.setProperty("sslrootcert", "cacert.pem");
urlProps.setProperty("sslsmciphper", "ECDHE_SM4_SM3");
urlProps.setProperty("sslcert", "client.crt");
urlProps.setProperty("sslkey", "client.key.pk8");
urlProps.setProperty("sslencert", "client_enc.crt");
urlProps.setProperty("sslenckey", "client_enc.key.pk8");
```

步骤5 加载驱动。

执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
Class.forName("com.huawei.gaussdb.jdbc.Driver");
```

步骤6 创建数据库连接。

调用DriverManager.getConnection(String url, Properties info)，进行数据库连接。
Connection conn = DriverManager.getConnection(urls,urlProps);

----结束

 说明

- 使用JDBC客户端国密TLS功能
 - 需依赖于1.1或以上版本的BGMPProvider开源软件（bgmpprovider-1.X.X-jar-with-dependencies.jar）。
 - JDK版本需为8u302及以上。
- 由于国密系列算法具有原理严密、计算复杂、安全强度高的特点，因此配置国密TLS加密传输后，相比国际TLS可能出现明显的性能劣化。
- 将客户端密钥转换为PKCS8二进制格式需使用OpenSSL命令行。

5.3.2.2.6 连接参数参考

连接参数的名称对大小写敏感。将连接参数进行分类，具体如表5-11所示。

表 5-11 连接参数分类

类别	参数名称
认证	PGDBNAME、PGHOST、PGPORT、user、password、tenantName、tenantPassword
数据库连接	ApplicationName、protocolVersion、connectMode、loginTimeout、logUnclosedConnections、allowConnThreadPool
网络连接	socketFactory、socketFactoryArg、receiveBufferSize、sendBufferSize、tcpKeepAlive、connectTimeout、socketTimeout、socketTimeoutInConnecting、cancelSignalTimeout、cmNetworkTimeout
安全	ssl、sslmode、sslgmcipher、sslcert、sslkey、sslencert、sslenckey、sslrootcert、sslpassword、sslpasswordcallback、sslfactory、sslprivatekeyfactory、sslfactoryarg、sslhostnameverifier、TLSCiphersSupported、enable_ce、key_info、auto_sendtoken、refreshClientEncryption
结果集	fetchsize、defaultRowFetchSize、unknownLength、uppercaseAttributeName、stripTrailingZeros、useGsClobBlobClass、enableObjectTypeConv
数据类型	binaryTransfer、binaryTransferEnable、binaryTransferDisable、blobMode、yearIsDateType、zeroCharacterReplacement、use_boolean、setFloat、stringtype
日志及调试	logger、loggerLevel、loggerFile、printSqlInLog、actualErrorCodeForBatch、connectionExtraInfo、traceInterfaceClass
兼容性	driverInfoMode、oracleCompatible、mCompatible、pgCompatible、compatibilityTags、enableObjectTypeConv、assumeMinServerVersion、mCompatibleSetDateTime、mQuoteStringCompatible
性能	prepareThreshold、preparedStatementCacheQueries、preparedStatementCacheSizeMiB、databaseMetadataCacheFields、databaseMetadataCacheFieldsMiB、batchMode、rewriteBatchedInserts

类别	参数名称
可用性	currentSchema、enableStandbyRead、loadBalanceHosts、autoBalance、disasterToleranceCluster、usingEip、enableAutoMigrate、priorityServers、preferredServiceGroup、preferredNodeIds、backupServiceGroup、backupNodeIds、hostRecheckSeconds、ApplicationType、sessionType、execTargetNode
其他功能	allowEncodingChanges、characterEncoding、enableTimeZone、autosave、cleanupSavepoints、enableStreamingQuery、preferQueryMode、allowReadOnly、parseCandidatesByDomain、primaryDomains、priorityDomains、refreshDomainResolveTime、refreshCNIpListTime、statementTimeout、executeUpdateQueryable、dbMonitor、enableGaussArrayAndStruct、getColumnReturnNull、extendTablePrivilege、diffProcAndFunc

认证

表 5-12 认证相关的连接参数

参数名称	参数说明	参数值
PGDBNAME	表示数据库名称（该参数在url中无需配置，系统将自动从Properties文件中解析）。	类型：String
PGHOST	在非透明多写特性下，表示主机IP地址（该参数在url中无需配置，系统将自动从Properties文件中解析），同时支持IPv4和IPv6。若配置多个CN，它们的IP和端口使用“:”分隔，并作为整体以逗号分隔其他CN。	类型：String
PGPORT	在非透明多写特性下，表示主机端口号（该参数在url中无需配置，系统将自动从Properties文件中解析）。	类型：Integer
user	表示创建连接的数据库用户。	类型：String
password	表示数据库用户的密码。	类型：String 须知 - 不推荐在url中配置该参数，存在敏感信息泄露的风险。 - 在非透明多写特性下，建议采用接口二的方式设置该参数进行数据库连接。或通过Properties连接属性来设置该参数，参见采用接口三连接数据库（非透明多写）。

数据库连接

表 5-13 数据库连接相关的连接参数

参数名称	参数说明	参数值
ApplicationName	表示正在连接的JDBC驱动名称。 非透明多写特性下，在CN上查询PGXC_STAT_ACTIVITY系统视图可以看到正在连接的客户端信息，其中application_name字段表示JDBC驱动名称。	类型：String 默认值：PostgreSQL JDBC Driver
protocolVersion	指定连接协议版本号，用于连接不同版本的数据库。 说明 设置protocolVersion时需确认数据库的类型和加密方式。 例如，用户要连接V5服务端且把该参数设置为3时，则需要确认或修改数据库的加密方式为MD5。修改加密方式为MD5，需要将password_encryption_type参数设置为1。随后使用对应加密方式的用户登录或者创建新用户，此时创建新用户的加密方式为MD5。同时修改pg_hba.conf，将客户端连接方式修改为MD5。 password_encryption_type参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 连接和认证 > 安全和认证”章节。	类型：Integer 取值范围： 1：表示连接V1服务端。 3：表示连接V5之前的服务端，或连接使用MD5加密方式的V5服务端。该方式存在安全风险，不建议配置该值。 350：表示连接V5之前的服务端。该方式存在安全风险，不建议配置该值。 351：表示连接支持MD5、SHA256及SM3加密方式的V5服务端，推荐配置该值。 默认值：351

参数名称	参数说明	参数值
loginTimeout	- 非透明多写特性下，表示建立数据库连接的等待时间。当url配置多IP时，若获取连接花费的时间超过此值，则连接失败，不再尝试后续IP。loginTimeout时间包括connectTimeout和socketTimeoutInConnecting，计算方式为： loginTimeout=(connectTimeout+socketTimeoutInConnecting)*节点数量。 说明 当CPU、内存、I/O负载中的任意一项接近100%时，会出现连接慢的现象，可能会导致连接时间超过阈值的问题，可通过以下方式问题进行排查： - 登录连接慢的物理机或通过管理工具查询资源负载：通过top命令等确认CPU使用率；通过free命令确认内存使用情况；通过iostat命令确认I/O负载。也可通过cm_agent中的监控日志，以及数据库运维平台中的监测记录进行检查。 - 针对短时间内大量慢查询导致的峰值负载场景，可通过[数据库服务器的端口号+1]端口连接，查询pg_stat_activity视图。针对慢查询，可以使用系统函数pg_terminate_backend(pid int)终止会话。 - 针对业务量长期超负载情况（无明显慢查询，或慢查询终止后但新的查询依然会变成慢查询），应考虑降低业务负载、增加数据库资源的方式进行优化。	类型：Integer 单位：s（秒） 有效取值范围：0 ~ 2147483647。0及负数表示已禁用，timeout不生效。 默认值：0 建议：配置后每次建立连接都会开启一个异步线程，在连接数较多的情况下可能会导致客户端压力增大，如果业务确认需要设置此参数，建议对该参数值进行调整。 - 在非透明多写特性下，建议调整为max(connectTimeout, socketTimeoutInConnecting) * 节点数，防止在网络异常且第n个IP为主的情况下无法成功连接。 说明 - 在非透明多写特性下，对于多IP配置，此参数设置值是尝试连接IP的总时间。若设置的值过小，可能会导致后面的IP无法连接的问题。例如配置了三个IP，如果loginTimeout为5s，尝试连接前两个IP总耗时5s且尚未成功建立连接，第三个IP将无法连接。此时会建立连接失败。

参数名称	参数说明	参数值
logUnclosedConnections	客户端可能由于未调用Connection对象的close()方法而泄漏Connection对象。设置为true时，JVM调用finalize()方法回收Connection对象会有日志记录。	类型：Boolean 取值范围： • true：表示如果用户没有调用close()方法，JVM调用finalize()方法回收Connection对象会有日志记录。 • false：表示如果用户没有调用close()方法，JVM调用finalize()方法回收Connection对象不会有日志记录。 默认值：false

网络连接

表 5-14 网络连接相关的连接参数

参数名称	参数说明	参数值
socketFactory	创建与服务器socket连接的类的名称。该类必须实现接口“javax.net.SocketFactory”，并定义无参或单String参数的构造函数。	类型：String
socketFactoryArg	表示上述socketFactory类构造函数的可选参数。如果不确定目标类的构造函数，则不推荐使用。	类型：String
receiveBufferSize	指定连接流上的SO_RCVBUF。	类型：Integer 单位：Byte 取值范围：-1 ~ 2147483647 默认值：-1。表示不设置缓冲区大小。
sendBufferSize	指定连接流上的SO_SNDBUF。	类型：Integer 单位：Byte 取值范围：-1 ~ 2147483647 默认值：-1。表示不设置缓冲区大小。

参数名称	参数说明	参数值
tcpKeepAlive	指定是否启用TCP保活探测功能。	类型：Boolean 取值范围： • true：表示启用TCP保活探测功能。 • false：表示禁用TCP保活探测功能。 默认值：false
connectTimeout	指定连接服务器操作系统的超时时间。如果连接到服务器操作系统花费的时间超过此值，则连接断开。当url配置多IP时，表示连接单个IP的超时时间。	类型：Integer 单位：s（秒） 取值范围：0 ~ 2147483。0表示已禁用，timeout不生效。 默认值：0
socketTimeout	指定socket读取操作的超时时间。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。推荐配置此参数，否则当数据库进程异常或服务端系统重启但未关闭socket时，客户端可能陷入长时间等待。建议依据业务可接受的SQL执行时间进行合理配置。 说明 当JDBC侧触发超时且连接关闭时，其下发给数据库侧正在运行的业务会被强制终止。业务强制终止的能力受GUC参数check_disconnect_query控制，设置为on表示支持该能力，设置为off表示不支持该能力。	类型：Integer 单位：s（秒） 有效取值范围：0 ~ 2147483。0及负数表示已禁用，timeout不生效。 默认值：0
socketTimeoutInConnecting	指定建立连接阶段socket读取操作的超时时间。在建立连接阶段，如果从服务器读取数据流超过此阈值，则尝试查找下一个节点建立连接。	类型：Integer 单位：s（秒） 有效取值范围：0 ~ 2147483。0及负数表示已禁用，timeout不生效。 默认值：5
cancelSignalTimeout	指定取消命令的“connect超时”和“socket超时”的超时时间。如果取消命令超过指定时间未响应，会中断该连接，减少占用客户端资源。	类型：Integer 单位：s（秒） 有效取值范围：0 ~ 2147483。0及负数表示已禁用，timeout不生效。 默认值：10

安全

表 5-15 安全相关的连接参数

参数名称	参数说明	参数值
ssl	表示是否以SSL方式连接数据库。设置为true时，支持使用NonValidatingFactory通道和证书两种SSL连接方式。	类型：Boolean 取值范围： • true：表示以SSL方式连接数据库。 • false：表示不以SSL方式连接数据库。 默认值：NULL（与false等效）
sslmode	表示SSL认证方式。	类型：String 取值范围：disable、allow、prefer、require、verify-ca、verify-full。 • disable：不使用SSL连接。 • allow：如果数据库服务器要求使用，则可以使用SSL连接，但不验证数据库服务器的真实性。 • prefer：如果数据库支持，那么首选使用SSL连接，但不验证数据库服务器的真实性。 • require：只尝试SSL连接，不会检查服务器证书是否由受信任的证书机构签发，且不会检查服务器主机名与证书中的主机名是否一致。 • verify-ca：只尝试SSL连接，并且验证服务器是否具有由可信任的证书机构签发的证书。 • verify-full：只尝试SSL连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。 默认值：NULL（与require等效）
sslgmcipher	指定国密TLS采用的加密算法套件。	类型：String 取值范围： • ECC_SM4_SM3：支持国密单向/双向认证。 • ECDHE_SM4_SM3：仅支持国密双向认证。 默认值：NULL

参数名称	参数说明	参数值
sslcert	指定客户端证书文件的完整路径。证书类型为End Entity。	类型：String 默认值：NULL（表示读取用户根目录的客户端证书文件）
sslkey	指定密钥文件的完整路径，使用时需将该密钥转换为DER格式。 openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt	类型：String 默认值：NULL（表示读取用户根目录的密钥文件）
sslencert	指定国密TLS客户端加密证书的完整路径，证书类型为End Entity。	类型：String 默认值：NULL（表示读取用户根目录的国密TLS客户端加密证书）
sslenckey	指定国密TLS密钥文件的完整路径，使用时需将该密钥转换为DER格式。 openssl pkcs8 -topk8 -outform DER -in client_enc.key -out client_enc.key.pk8 -nocrypt	类型：String 默认值：NULL（表示读取用户根目录的国密TLS密钥文件）
sslrootcert	表示SSL根证书的文件名。根证书的类型为CA。	类型：String 默认值：NULL（表示读取用户根目录的SSL根证书）
sslpassword	表示SSL密码。该参数提供给ConsoleCallbackHandler组件使用。	类型：String 默认值：NULL 须知 - 不允许在url中配置该参数，存在敏感信息泄露的风险。 - 建议通过Properties连接属性来设置该参数，参见采用接口三连接数据库（非透明多写）。
sslpasswordcallback	指定SSL密码提供者的完整类名。	类型：String 默认值：NULL（表示不开启此功能）
sslfactory	指定SSLSocketFactory在建立SSL连接时使用的类名。	类型：String 默认值：NULL（表示不开启此功能）

参数名称	参数说明	参数值
sslprivatekeyfactory	指定实现私钥解密方法的接口 com.huawei.gaussdb.jdbc.ssl.PrivateKeyFactory的实现类的完整限定类名。若未指定该参数，系统将首先尝试使用默认的JDK私钥解密算法。若无法解密则使用 com.huawei.gaussdb.jdbc.ssl.BouncyCastlePrivateKeyFactory，用户需提供bcpkix-jdk15on.jar包（建议版本1.65及以上）。	类型：String 默认值：NULL（表示会尝试使用默认的JDK私钥解密算法进行解密）
sslfactoryarg	表示sslfactory类构造函数的可选参数（不推荐使用）。	类型：String 默认值：NULL
sslhostnameverifier	表示主机名验证程序的类名。该参数实现 javax.net.ssl.HostnameVerifier接口。	类型：String 默认值： com.huawei.gaussdb.jdbc.ssl.PGjdbcHostnameVerifier
TLSCiphersSupported	指定支持的TLS加密套件列表。 说明 设置为不支持的TLS加密套件时，系统将抛出异常。	类型：String 默认值： TLS_DHE_RSA_WITH_AES_128_GCM_SHA256,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
enable_ce	表示密态等值查询功能的启用模式。	类型：Integer 取值范围： 1或3：表示JDBC支持密态等值查询基础功能。 - 其他值：表示不支持密态等值查询功能。 默认值：NULL（表示不支持密态等值查询功能）

参数名称	参数说明	参数值
key_info	与enable_ce参数协同使用，用于配置访问外部密钥管理服务的参数。该参数仅在密态数据库环境下生效。	类型：String 默认值：NULL 须知 - 不推荐在url中配置该参数，有敏感信息泄露的风险。 - 建议通过Properties连接属性来设置该参数，参见采用接口三连接数据库（非透明多写）。
auto_sendtoken	与enable_ce参数协同使用，仅在密态数据库参数enable_ce设置为3时生效。启用该参数后，JDBC客户端将在建立连接后自动执行与setClientInfo("send_token")功能相同的传输密钥操作。	类型：Boolean 默认值：false
refreshClientEncryption	表示密态数据库是否支持客户端缓存刷新功能。	类型：String 取值范围： 1或NULL：表示密态数据库支持客户端缓存刷新功能。 - 其他字符串：表示密态数据库不支持客户端缓存刷新功能。 默认值：NULL

结果集

表 5-16 结果集相关的连接参数

参数名称	参数说明	参数值
fetchsize	指定一次fetch在ResultSet中读取的行数，其功能与defaultRowFetchSize参数等效。	类型：Integer 有效取值范围：0 ~ 2147483647。0及负数表示一次fetch在ResultSet中读取所有行数。 默认值：0 建议：用户根据实际业务查询数据数量和客户端机器内存情况来配置此参数。设置fetchsize时要关闭自动提交模式（conn.setAutoCommit(false);），否则会导致fetchsize无法生效。

参数名称	参数说明	参数值
defaultRowFetchSize	指定一次fetch在ResultSet中读取的行数。限制每次访问数据库时读取的行数可以避免不必要的内存消耗，从而避免OutOfMemoryException。 须知 若同时设置fetchsize和defaultRowFetchSize，以fetchsize参数的值为准。	类型：Integer 取值范围：0 ~ 2147483647。0表示一次fetch在ResultSet中读取所有行数。 默认值：0
unknownLength	GaussDB某些数据类型（例如TEXT）没有明确数据长度，当通过java.sql.ResultSetMetaData#getColumnDisplaySize和java.sql.ResultSetMetaData#getPrecision等方法返回这些类型的数据时，此参数用于指定数据长度。	类型：Integer 取值范围：0 ~ 2147483647 默认值：2147483647
uppercaseAttributeName	指定将获取元数据的接口的查询结果转为大写或小写。主要适用场景为数据库中存储元数据全为小写，但要使用大写的元数据作为出参和入参。 涉及到的接口请参见： java.sql.DatabaseMetaData、 java.sql.ResultSetMetaData。 说明 uppercaseAttributeName参数开启后，如果数据库中有小写、大写和大小写混合的元数据，只能查询到小写部分的元数据，并以大写的形式输出。使用前请务必确认存储的元数据全为小写，以避免数据出错。	类型：String 取值范围： • true、upper：表示将元数据转为大写。 • false：表示以GUC参数uppercase_attribute_name配置为准。 • lower：表示将元数据转为小写。 默认值：false
stripTrailingZeros	指定是否去除numeric类型后的0，仅对java.sql.ResultSet#getObject(int columnIndex)生效。	类型：Boolean 取值范围： • true：表示去除numeric类型后的0。 • false：表示不去除numeric类型后的0。 默认值：false

参数名称	参数说明	参数值
useGsClobBlobClass	指定使用 java.sql.ResultSet#getObject方法获取Clob字段和Blob字段时返回某种类型的对象，以及使用 java.sql.ResultSetMetaData#getColumnName方法获取Clob字段和Blob字段时返回的类型名称。	类型： Boolean 取值范围： • true：表示使用 java.sql.ResultSet#getObject方法获取Clob字段时会返回PGClob类型的对象，获取Blob字段时会返回PGBlob类型的对象。使用 java.sql.ResultSetMetaData#getColumnName方法获取Clob字段的类型名称会返回 java.sql.Clob，获取Blob字段的类型名称时会返回java.sql.Blob。 • false：表示使用 java.sql.ResultSet#getObject方法获取Clob字段时会返回PGClob类型的对象，获取Blob字段时会返回byte[]类型的对象。使用 java.sql.ResultSetMetaData#getColumnName方法获取Clob字段的类型名称会返回 java.sql.CLOB，获取Blob字段的类型名称时会返回 java.sql.BLOB。 默认值： false
enableObjectTypeConversion	表示 java.sql.ResultSet#getObject(int, Class<?>)接口是否支持兼容类型转换。当前已支持数值类型互转、字符串类型互转和日期/时间类型互转。	类型： Boolean 取值范围： • true：表示支持兼容类型转换。 • false：表示不支持兼容类型转换。 默认值： false

数据类型

表 5-17 数据类型相关的连接参数

参数名称	参数说明	参数值
binaryTransfer	指定是否启用二进制格式发送和接收数据。	类型：Boolean 取值范围： • true：表示启用二进制格式传输数据。 • false：表示不启用二进制格式，而是使用text格式传输数据。 默认值：false
binaryTransferEnable	指定启用二进制传输的类型列表。以逗号分隔，例如：binaryTransferEnable=INT4_ARRAY,INT8_ARRAY。 可使用OID编号或者名称配置。例如：OID名称为BLOB，编号为88，可以进行如下配置： binaryTransferEnable=BLOB或binaryTransferEnable=88。 须知 如果binaryTransferDisable与binaryTransferEnable有相同的value，则会禁用。	类型：String 默认值：""（空字符串）
binaryTransferDisable	指定禁用二进制传输的类型列表。以逗号分隔，可使用OID编号或者名称配置。	类型：String 默认值：""（空字符串）
blobMode	指定setBinaryStream()方法为不同类型的数据赋值。	类型：String 取值范围： • on：表示为blob类型数据赋值。 • off：表示为bytea类型数据赋值。 默认值：on

参数名称	参数说明	参数值
yearIsDate Type	使用 java.sql.ResultSet#getObject 接口和 java.sql.ResultSet#getString接 口获取year数据类型字段时， 指定返回的数据对象是否以 Date类型呈现。	类型： Boolean 取值范围： • true：表示使用 java.sql.ResultSet#getObject方 法获取year数据类型字段时会返 回Date类型的对象，使用 java.sql.ResultSet#getString方法 获取year数据时会返回yyyy-mm- dd格式的日期字符串，使用 java.sql.ResultSet#getInt/ getLong/getShort方法获取数据 时会返回年份数值。 • false：表示使用 java.sql.ResultSet#getObject方 法获取year数据类型字段时会返 回Integer类型的对象，使用 java.sql.ResultSet#getString方法 获取year数据时会返回年份的字 符串，使用 java.sql.ResultSet#getInt/ getLong/getShort方法获取数据 时会返回年份数值。 默认值： true
use_boole an	表示extended模式下 setBoolean()方法绑定的Oid类 型。	类型： Boolean 取值范围： • true：表示绑定Boolean类型。 • false：表示绑定int2（ Smallint ） 类型。 默认值： false
setFloat	表示调用setFloat或者 setObject指定类型为float时， 传递给数据库的Oid是否为 float4的Oid。M- Compatibility模式数据库下不 受该参数影响。	类型： Boolean 取值范围： • true：表示调用setFloat或者 setObject指定类型为float时，传 递给数据库的Oid为float4。 • false：表示调用setFloat或者 setObject指定类型为float时，传 递给数据库的Oid为float8。 默认值： false

参数名称	参数说明	参数值
stringtype	指定调用 java.sql.PreparedStatement#setString方法时传给数据库的参数类型。	类型：String 取值范围： • varchar：表示将参数作为varchar类型发送给服务器。 • unspecified：表示参数发送到服务器时不指定类型，服务器将尝试推断适当的类型。 默认值：NULL（与varchar等效）

日志及调试

表 5-18 日志及调试相关的连接参数

参数名称	参数说明	参数值
logger	表示JDBC驱动要使用的日志输出框架。JDBC驱动支持对接用户应用程序使用的日志输出框架。	类型：String 取值范围： • Slf4JLogger：基于Slf4j-API的日志框架。目前支持的第三方日志输出框架只有基于Slf4j-API的日志框架。具体使用方式，请参见日志管理。 • JdkLogger：Java内置日志框架。若未显式配置，驱动将默认使用JdkLogger。
loggerLevel	指定日志记录级别。	类型：String 取值范围：目前支持四种级别，不区分大小写，取值如下。 • OFF：关闭所有日志输出。 • INFO：记录基本运行信息。 • DEBUG：记录调试信息（信息量大于INFO）。 • TRACE：记录最详细的调试信息（信息量最大）。 默认值：OFF
loggerFile	在开启连接监控功能时，该参数用于指定监控日志输出路径。仅限连接监控场景下配置该参数，其他场景下该参数不生效。	类型：String 默认值：NULL（表示在客户端运行程序目录下生成日志文件）

参数名称	参数说明	参数值
printSqlInLog	指定是否在异常信息或日志中输出SQL语句。	类型：Boolean 取值范围： • true：启用SQL语句输出。 • false：禁用SQL语句输出。 默认值：true
actualError CodeForBatch	executeBatch接口执行报错时，指定返回实际错误码或者返回固定错误码0。	类型：Boolean 取值范围： • true：返回实际错误码。 • false：返回固定错误码0。 默认值：false
connection ExtraInfo	指定是否上报驱动相关信息到数据库。	类型：Boolean 取值范围： • true：上报驱动部署路径、进程属主用户、url连接配置到数据库，信息将记录在connection_info参数中。 说明 • 非透明多写特性下，可通过PG_STAT_ACTIVITY和PGXC_STAT_ACTIVITY查看驱动相关信息。 • false：不进行信息上报。 默认值：false
tracerInterfaceClass	指定获取tracerId方法接口com.huawei.gaussdb.jdbc.log.Tracer的实现类完整限定名。	类型：String 默认值：NULL（表示不开启此功能）

兼容性

表 5-19 兼容性相关的连接参数

参数名称	参数说明	参数值
driverInfo Mode	表示驱动描述信息的输出模式。	类型：String 取值范围： • postgresql：表示输出 PostgreSQL 相关的驱动描述信息。 • gaussdb：表示输出 GaussDB 相关的驱动描述信息。 默认值：postgresql

参数名称	参数说明	参数值
oracleCompatible	指定驱动侧是否开启Oracle兼容特性。	类型：String 取值范围： • true：表示开启驱动侧所有的Oracle兼容tag（驱动升级后自动包含新版本新增的tag）。 • false：表示关闭驱动侧所有的Oracle兼容tag。 • tag：表示开启驱动侧部分的Oracle兼容特性，每个tag对应一个特定的Oracle兼容特性。当配置多个tag时，tag之间用逗号隔开，例如："tag1,tag2,tag3"。当前支持配置的tag包括： - getProcedureColumns：DatabaseMetaData#getProcedureColumns方法兼容Oracle数据库行为。当未指定catalog或schema时，默认不作为过滤条件。 - getCallableStatementResults：CallableStatement接口的getLong、getInt、getShort、getBytes方法兼容Oracle数据库行为。在注册出参类型为java.sql.Types#NUMERIC时，调用CallableStatement的getLong、getInt、getShort、getBytes方法，能够正常接收out的值，只有out数值超过java数值，数据类型取值范围会提示SQLException异常。 - batchInsertAffectedRows：reWriteBatchedInserts参数启用后，Statement#executeBatch方法才能兼容Oracle数据库行为。 - setDateTime：PBE（Prepare-Bind-Execute）模式下绑定Date、Time、Timestamp类型参数时，指定具体oid下发至数据库。 - blob：开启后，Blob类的接口兼容Oracle数据库行为。 - clob：开启后，Clob类的接口兼容Oracle数据库行为。

参数名称	参数说明	参数值
		– functionsCompatible：以下接口方法兼容Oracle数据库行为。 – ResultSet#wasNull、getBoolean、insertRow – DatabaseMetaData#getProcedureTerm – Blob#length、getBytes、position、setBytes、truncate、free – Clob#length、getSubString、getCharacterStream、getAsciiStream、position、setString、truncate、free – CallableStatement，请参见java.sql.CallableStatement。 默认值：false

参数名称	参数说明	参数值
mCompatible	指定驱动侧是否开启MySQL兼容特性。	类型：String 取值范围： • true：表示开启驱动侧所有的MySQL兼容tag（驱动升级后自动包含新版本新增的tag）。 • false：表示关闭驱动侧所有的MySQL兼容tag。 • tag：表示开启驱动侧部分的MySQL兼容特性，每个tag对应一个特定的MySQL兼容特性。当配置多个tag时，tag之间用逗号隔开，例如："tag1,tag2,tag3"。当前支持配置的tag包括： - DatetimeAsTimeStampType：使用java.sql.ResultSet#getObject方法获取datetime数据类型字段时会返回Timestamp类型的对象。 - blob：开启后，Blob类的接口兼容MySQL数据库行为。 - clob：开启后，Clob类的接口兼容MySQL数据库行为。 - functionsCompatible：以下接口方法兼容MySQL数据库行为。 - ResultSet#wasNull - DatabaseMetaData#getProcedureTerm - ResultSetMetaData#getColumnName - Blob#length、getBytes、setBytes、truncate、free - Clob#length、getSubString、getCharacterStream、getAsciiStream、position、setString、truncate、free 默认值：false

参数名称	参数说明	参数值
mCompatibleSetDate Time	连接MySQL兼容数据库时，绑定Date、Time、Timestamp类型参数，指定oid为具体值或为0下发至数据库。	类型： Boolean 取值范围： • true：绑定Date、Time、Timestamp类型参数，指定具体oid下发至数据库，与MySQL数据库的PBE模式（ useServerPrepStmts=true&cachePrepStmts=true ）保持兼容。 • false：绑定Date、Time、Timestamp类型参数，oid设置为0下发至数据库，由服务端自动推导类型，与MySQL数据库的默认Query模式保持兼容。 默认值： false
pgCompatible	设置JDBC对SQL语法解析方式是否兼容PG。	类型： Boolean 取值范围： • true：表示语法解析兼容PG，支持一个statement中包含多条SQL，并使用分号进行分隔的场景。开启后不支持执行带有分号的SQL语句。 • false：表示SQL语句解析方式采用默认行为（不兼容PG）。 默认值： false
compatibilityTags	将驱动的部分特性恢复到早期版本，以确保前向兼容。可配置相应tag，每个tag代表开启对应的前向兼容特性，不同tag之间用逗号隔开。目前支持的tag如下： typeMapInitNull：使用该tag后，兼容驱动早期版本的行为。 说明 一般不建议使用该参数，除非有明确的前向兼容问题。	类型： String 默认值： NULL
enableObjectTypeConv	表示 java.sql.ResultSet#getObject(int, Class<?>)方法是否支持兼容类型转换。当前已支持数值类型互转、字符串类型互转和日期/时间类型互转。	类型： Boolean 取值范围： • true：表示支持兼容类型转换。 • false：表示不支持兼容类型转换。 默认值： false

参数名称	参数说明	参数值
assumeMinServerVersion（废弃）	指定连接的服务器最低版本。 例如： assumeMinServerVersion高于或者等于9.0，可优化连接建立过程。客户端将不再发送将float精度设置为3的请求（默认维持精度等级2）。	类型：String 默认值：NULL
mQuoteStringCompatible	表示 java.sql.DatabaseMetaData#getIdentiferQuoteString()方法是否兼容MySQL。	类型：Boolean 取值范围： • true：兼容MySQL，若数据库的sql_mode参数包含ANSI_QUOTES选项，该方法返回双引号，否则返回反引号。 • false：不兼容MySQL，该方法返回双引号。 默认值：false

性能

表 5-20 性能相关的连接参数

参数名称	参数说明	参数值
prepareThreshold	表示PreparedStatement对象的执行次数阈值。达到该阈值后将复用服务端已解析的statement。 说明 - 开启该功能时，不建议使用prepareStatement方法执行涉及密码的语句（例如：CREATE USER user_name WITH PASSWORD '*****';）。因为当执行达到prepareThreshold指定次数时，数据库将缓存SQL语句，基于安全因素不会缓存密码，再次执行该prepareStatement方法时会报错“Password must contain at least 8 characters.”。 - 不推荐prepareThreshold设置为0或1。 - 设置为0时，每次执行都需要重新解析SQL，降低数据库性能。 - 设置为1时，即使仅执行一次也会缓存执行计划，造成不必要的内存资源消耗。	类型：Integer 有效取值范围：0 ~ 2147483647。0及负数表示禁用此功能。 默认值：5。表示同一PreparedStatement对象执行次数达到5次后，不再向服务端发送parse消息进行重复解析，而是复用服务端已解析的statement。
preparedStatementCacheQueries	表示每个连接中使用prepareStatement()创建的查询数量超过某一阈值时，将采用LRU（最近最少使用）策略淘汰查询缓存。	类型：Integer 有效取值范围：0 ~ 2147483647。0及负数表示禁用缓存。 默认值：256。表示每个连接中使用prepareStatement()创建的查询数量超过256时，将采用LRU（最近最少使用）策略淘汰查询缓存。 建议：根据业务需求调整该参数，需与prepareThreshold参数配合使用。
preparedStatementCacheSizeMiB	表示每个连接中使用prepareStatement()创建的查询占用缓存空间超过某一阈值时，将采用LRU（最近最少使用）策略淘汰查询缓存。	类型：Integer 单位：MB 有效取值范围：0 ~ 2147483647。0及负数表示禁用缓存。 默认值：5。表示每个连接中使用prepareStatement()创建的查询占用缓存空间超过5MB时，将采用LRU（最近最少使用）策略淘汰查询缓存。

参数名称	参数说明	参数值
databaseMetadataCacheFields	指定每个连接中缓存的数据库元数据字段数量的最大值。	类型：Integer 有效取值范围：0 ~ 2147483647。0及负数表示禁用缓存。 默认值：65536
databaseMetadataCacheFieldsMiB	指定每个连接中缓存的数据库元数据字段占用缓存空间的最大值。	类型：Integer 单位：MB 有效取值范围：0 ~ 2147483647。0及负数表示禁用缓存。 默认值：5
batchMode	表示是否使用batch模式连接。 说明 设置batchMode=on时，执行批量插入/更新/删除操作，每一列的数据类型以第一条数据指定的类型为准，若数据类型混用可能会导致报错或者插入的数据异常。	类型：String 取值范围： - on：表示开启batch模式，可以提升批量插入/更新/删除的性能。成功执行批量插入/更新/删除操作后，返回结果为[count, 0, 0...0]，数组第一个元素为批量插入/更新/删除的总行数。 - off：表示关闭batch模式。成功执行插入/更新/删除操作后，返回结果的数组各元素对应单条SQL语句执行的影响行数。 默认值：on
rewriteBatchedInserts	表示批量插入时，是否重写SQL语句。使用该参数时，要求设置batchMode=off。	类型：Boolean 取值范围： - true：表示批量插入时，可将插入语句合并为一条： <pre>INSERT INTO TABLE_NAME VALUES (values1, ..., valuesN), ..., (values1, ..., valuesN);</pre> - false：表示批量插入时，不重写SQL语句。 默认值：false

可用性

表 5-21 可用性相关的连接参数

参数名称	参数说明	参数值
currentSchema	指定当前连接的Schema。如果Schema名包含除字母、数字、下划线之外的特殊字符，需要对Schema名加引号，注意加引号后Schema名大小写敏感。如需配置多个Schema，要用逗号进行分隔。 例如： currentSchema=schema_a,"schema-b","schema/c"	类型：String 默认值：NULL（表示使用连接的用户名作为默认Schema）
enableStandbyRead	指定是否开启备机读模式。	类型：Boolean 取值范围： • true：表示启用。 • false：表示禁用。 默认值：false
loadBalanceHosts	指定是否使用负载均衡功能。 说明 • 该参数不可与autoBalance参数一起使用。	类型：Boolean 取值范围： • true：表示使用洗牌算法从候选主机中随机选择一个主机建立连接。 • false：表示顺序连接url中指定的多个主机。 默认值：false

参数名称	参数说明	参数值
autoBalance	- 在非透明多写特性下，指定负载均衡的策略。 须知 - 负载均衡是基于连接级别，不是基于事务级别。如果连接是长连接，并且连接上的负载不均衡，无法保证CN节点上的负载均衡。 - 该参数仅能在分布式场景下使用。 - 配置该参数后，除首次建立连接外，后续与数据库建立连接会依赖pgxc_node表中的IP和端口，具体请参见usingEip参数中的须知。 - 该参数不可与loadBalanceHosts参数一起使用。 - 如果在连接容灾集群时开启了负载均衡功能，那么查询请求会发送到主集群上，导致查询性能会受到影响。 - 在开启负载均衡时，url中可以配置浮动IP或数据IP，如果配置为浮动IP，系统会根据浮动IP获取对应的数据IP，通过获取的数据IP实现负载均衡。因此无论url中配置浮动IP或数据IP时，都需要确保数据IP网络连接正常，否则负载均衡功能将使用异常。	- 非透明多写特性 类型：String 取值范围： - true、balance或roundrobin：表示开启JDBC负载均衡功能，将应用程序的多个连接均衡到数据库集群中的各个可用CN。 例如： <pre>jdbc:gaussdb:// host1:port1,host2:port2/database? autoBalance=true</pre> JDBC会定期刷新整个集群的可用CN列表（CN列表中获取的host是数据IP），该刷新周期可以通过参数refreshCNipListTime进行配置，默认值为10s。比如获取到的列表为： <pre>host1:port1,host2:port2,host3:port3,host4:port4</pre> 在启用负载均衡时，host1和host2仅在首次连接时保证高可用性。之后，Driver将从host1、host2、host3和host4中依次选择可用CN来刷新可用CN列表。后续用户新建的连接将使用轮询（RoundRobin）算法从host1、host2、host3和host4中选择CN主机进行连接。 - priorityn：表示开启JDBC优先级负载均衡功能，将应用程序的多个连接首先均衡到url上配置的前n个中可用的CN数据库节点。当url上配置前n个节点全部不可用时，连接会随机分配到数据库集群中其他可用CN数据库节点。n为正整数，且小于url上配置的CN数量。当parseCandidatesByDomain设置为true时，priorityn的优先级配置不生效，这种场景下的优先级使用priorityDomains参数配置。使用priorityn进行负载均衡时，host不能使用IPv6。在多进程建立连接场景下，该策略会出现负载不均衡。

参数名称	参数说明	参数值
		例如： <pre>jdbc:gaussdb:// host1:port1,host2:port2,host3:port3,ho st4:port4/database? autoBalance=priority2</pre> JDBC将定期获取整个集群可用CN列表，比如获取到的列表为： <pre>host1:port1,host2:port2,host3:port3,ho st4:port4,host5:port5,host6:port6</pre> 其中host1和host2处于AZ1，host3和host4处于AZ2。Driver将优先从host1、host2中做负载均衡，host1和host2全部不可用时，才会从host3、host4、host5和host6中随机选择CN节点进行连接。 - shuffle：表示开启JDBC随机负载均衡功能，将应用程序的多个连接随机均衡到数据库集群中的各个可用CN。 例如： <pre>jdbc:gaussdb:// host1:port1,host2:port2,host3:port3/ database?autoBalance=shuffle</pre> JDBC将定期获取整个集群的可用CN列表，比如获取到的列表为： <pre>host1:port1,host2:port2,host3:port3,ho st4:port4</pre> host1、host2和host3仅在首次连接时保证高可用性，后续将使用洗牌（shuffle）算法在刷新后的CN列表中随机选择一个CN节点进行连接。 - specified：表示开启JDBC负载均衡功能，并且只在用户url中指定的节点上建立连接。 例如： <pre>jdbc:gaussdb:// host1:port1,host2:port2,host3:port3,ho st4:port4/database? autoBalance=specified&priorityServers =2</pre> JDBC 将识别用户指定的host1和host2为主集群中用于建立连接的节点，而host3和host4则作为备集群的连接节点。建立连接时，JDBC会首先在host1和host2上均匀分配连接。如果主集群中的所有节点

参数名称	参数说明	参数值
		（ host1和host2 ）均发生故障，JDBC不会尝试在剩余的集群节点上建立连接。当备节点升级为主节点后，JDBC会在 host3和host4上均匀分配连接。如果备集群中的所有节点（ host3和host4 ）均发生故障，JDBC同样不会尝试在剩余的集群节点上建立连接。 - shuffle：表示开启JDBC在指定节点上优先随机负载均衡功能，将应用程序的多个连接首先随机均衡到url上配置的前n个中可用的CN数据库节点。当url上配置前n个节点全部不可用时，连接会随机分配到数据库集群中其他可用CN数据库节点。n为正整数，且小于等于url上配置的CN数量。 当 parseCandidatesByDomain 设置为true时，shuffle的优先级配置不生效，这种场景下的优先级使用 priorityDomains 参数配置。使用shuffle进行负载均衡时，host不能使用IPv6。在多进程建立连接场景下，通过该策略可以实现指定优先节点的负载均衡连接。 例如： <pre>jdbc:gaussdb:// host1:port1,host2:port2,host3:port3,host4:port4/database? autoBalance=shuffle2</pre> JDBC将定期获取整个集群的可用CN列表，比如获取到的列表为： <pre>host1:port1,host2:port2,host3:port3,host4:port4,host5:port5,host6:port6</pre> Driver将优先从host1、host2中进行随机负载均衡，host1和host2全部不可用时，才会从host3、host4、host5和host6中随机选择CN主机进行连接。 - false：表示不开启JDBC负载均衡功能和优先级负载均衡功能。 默认值： false

参数名称	参数说明	参数值
disasterToleranceCluster	表示在开启负载均衡功能的场景下且JDBC连接数据库的url中只配置容灾集群节点时，判断负载均衡连接到主集群或连接到容灾集群。该参数不可与priorityServers参数共同使用。 说明 配置该参数时，连接串中应该只配置主集群或者容灾集群。	类型：Boolean 取值范围： - true：表示开启，会在容灾集群节点进行负载均衡。 - false：表示关闭，会在主集群节点间进行负载均衡。 默认值：false
usingEip	指定是否使用业务IP实现负载均衡。 须知 - 该参数需要配合autoBalance参数使用。 - 当usingEip=true时，除首次建立连接外，后续使用业务IP与数据库建立连接（对应pgxc_node表中node_host1和node_port1列）；当usingEip=false时，除首次建立连接外，后续使用数据IP与数据库建立连接（对应pgxc_node表中node_host和node_port列）。	类型：Boolean 取值范围： - true：表示使用业务IP实现负载均衡。 - false：表示使用数据IP实现负载均衡。 默认值：true
priorityServers	指定url中优先连接的节点数量。应用于流式容灾场景。该参数不可与disasterToleranceCluster参数共同使用。 示例： jdbc:gaussdb://host1:port1,host2:port2,host3:port3,host4:port4/database?priorityServers=2 表示host1与host2为主集群节点，host3与host4为容灾集群节点。	类型：Integer 取值范围：大于0且小于url中配置的CN数量。 默认值：NULL（表示不开启此功能）
hostRecheckSeconds	JDBC尝试连接主机后将记录主机状态（连接成功或连接失败、主节点或备节点）。在设定的hostRecheckSeconds时间内保持记录的主机状态可信，超时后记录的主机状态失效。	类型：Integer 单位：s（秒） 取值范围：0 ~ 2147483 默认值：10

参数名称	参数说明	参数值
ApplicationType	指定是否开启分布式写入和查询。该参数仅在数据库处于GTM-Free模式时有效。	类型：String 取值范围： - not_perfect_sharding_type：开启分布式写入和查询。 - perfect_sharding_type：默认禁止分布式写入和查询，只有在SQL文中加入/* multinode */才能执行分布式写入和查询。 默认值：not_perfect_sharding_type
sessionType	指定当前会话的类型，并生成对应的执行计划。	类型：String 取值范围： - normal：表示普通会话类型，根据数据库原有逻辑生成执行计划。 - singleNode：表示单节点会话类型，读写操作支持下推。单节点会话模式通过CN对指定目标DN进行1对1的会话。此模式需与参数execTargetNode同时使用，设置参数execTargetNode为目标DN节点名称，JDBC将优先连接目标DN对应的本地CN节点。 - globalReadOnly：表示跨节点只读会话类型。该模式下会正常生成分布式计划，但禁止分布式写操作。 默认值：normal
execTargetNode	表示单节点会话模式下对应的DN节点。此参数需与sessionType同时使用，且仅当sessionType设置为singleNode时生效。 说明 若sessionType设置为singleNode，但未指定execTargetNode，其效果等同于sessionType设置为normal。	类型：String 取值范围：当前集群内的所有DN节点名称。 默认值：NULL

其他功能

表 5-22 其他功能相关的连接参数

参数名称	参数说明	参数值
allowEncodingChanges	表示是否允许修改字符集类型，通常与characterEncoding参数配合使用，用“&”连接，格式示例如下： allowEncodingChanges=true&characterEncoding=UTF8	类型：Boolean 取值范围： • true：表示允许修改字符集类型。 • false：表示禁止修改字符集类型。 默认值：false
characterEncoding	表示字符集类型，需与allowEncodingChanges参数配合使用。当allowEncodingChanges为false时，characterEncoding仅允许设置为UTF8；当allowEncodingChanges为true时，characterEncoding可在支持范围内进行配置。 须知 • 配置GB18030_2022时，需确保数据库支持该字符集。 • 1.8.0_381及以上的JDK8版本支持GB18030 2022，低于该版本的JDK8仅支持GB18030 2000。若使用低版本JDK8并配置GB18030_2022，JDBC实际使用的字符集将降级为GB18030。 • 17.0.8及以上的JDK17版本支持GB18030 2022，低于该版本的JDK17仅支持GB18030 2000。若使用低版本JDK17并配置GB18030_2022，JDBC实际使用的字符集将降级为GB18030。	类型：String 取值范围：UTF8、GBK、LATIN1、GB18030、GB18030_2022、ZHS16GBK。 默认值：UTF8
enableTimeZone	指定是否启用客户端时区设置。	类型：Boolean 取值范围： • true：表示启用客户端时区设置（根据JVM时区指定数据库时区）。 • false：表示禁用客户端时区设置（使用数据库时区）。 默认值：true

参数名称	参数说明	参数值
autosave	事务内SQL语句执行失败时，指定驱动程序的处理策略。	类型：String 取值范围： - always：JDBC驱动程序在每次执行SQL语句之前设置一个保存点，并在失败时回滚到该保存点。 - never：不设置任何保存点。 - conservative：每次执行SQL语句之前都会设置保存点，仅在statement XXX无效等特定异常时回滚并重试。 默认值：never
cleanupSavepoints	指定是否自动释放JDBC创建的保存点（autosave为always或conservative时创建）。	类型：Boolean 取值范围： - true：自动释放JDBC创建的保存点。 - false：禁止此功能。 默认值：false
enableStreamingQuery	指定是否开启流式读功能。 说明 流式读功能：读取数据时，服务端将全部数据发送至客户端Socket缓冲区。缓冲区占满时暂停发送，有空余则继续向缓冲区中发送数据，同时JVM逐行从缓冲区中读取数据。 - 处理结果快，不会造成JVM内存溢出。 - 仅支持向后遍历，且在数据处理完成前或statement关闭前，当前连接无法执行其他操作。具体示例请参见使用流式读功能。	类型：Boolean 取值范围： - 设置为true，且使用statement.setFetchSize(Integer.MIN_VALUE)或者statement.enableStreamingResults()接口时，才能开启流式读功能。 - 设置为false，表示关闭流式读功能。 默认值：false

参数名称	参数说明	参数值
preferQueryMode	指定执行查询的模式。	类型：String 取值范围： • simple：使用简单查询将整个SQL语句（含参数）作为一个整体发送给数据库执行，无法缓存SQL语句。 • extended：使用扩展查询将SQL语句的解析（Parse）、参数绑定（Bind）和执行（Execute）分成多个步骤，仅对预编译语句（PreparedStatement）进行缓存，不缓存普通语句（Statement）。 • extendedForPrepared：对预编译语句（PreparedStatement）使用扩展查询，而对于普通语句（Statement）则使用简单查询，仅对预编译语句进行缓存，不缓存普通语句。 • extendedCacheEverything：使用扩展查询，同时缓存预编译语句（PreparedStatement）和普通语句（Statement）。 默认值：extended
allowReadOnly	指定是否允许开启只读模式。当allowReadOnly为true时，调用connection.setReadOnly(true)后，只读模式生效，无法修改数据；当allowReadOnly为false时，调用connection.setReadOnly(true)后，只读模式不生效，仍可以修改数据。	类型：Boolean 取值范围： • true：允许设置只读模式。 • false：禁止设置只读模式。 默认值：true
parseCandidatesByDomain	指定是否根据域名获取多个候选节点。	类型：Boolean 取值范围： • true：启用域名解析功能。开启后需将url中的host信息配置为“域名+端口”格式，驱动将基于域名获取IP并生成备节点集合，可配合autoBalance参数实现负载均衡功能。 • false：禁用域名解析功能。 默认值：false

参数名称	参数说明	参数值
primaryDomains	指定url中配置的前n个域名为主集群并优先连接。参数值大于0时表示启用该功能，需配合parseCandidatesByDomain参数为true时才能生效。将url中的域名拆分成两个分组，前一个分组为主集群，后一个分组为备集群。发生主备切换后，前一个分组为备集群，后一个分组为主集群。适用于流式容灾和Dorado双集群场景。 示例： jdbc:gaussdb://domain1:port1,domain2:port2,domain3:port3,domain4:port4/database?parseCandidatesByDomain=true&primaryDomains=2 domain1和domain2对应主集群，优先连domain1和domain2对应的IP。如果发生主备切换，将domain3和domain4标记为主集群，优先对domain3和domain4建立连接。	类型：Integer 取值范围：0 ~ (url配置的域名数量-1) 默认值：0（关闭此功能）
priorityDomains	指定url中配置的前n个域名为优先连接的域名。参数值大于0时表示启用该功能，需配合parseCandidatesByDomain参数为true时才能生效。若已配置primaryDomains，则此参数值需小于primaryDomains的配置值。 示例： jdbc:gaussdb://domain1:port1,domain2:port2,domain3:port3,domain4:port4/database?parseCandidatesByDomain=true&primaryDomains=2&priorityDomains=1 domain1和domain2对应主集群，但将优先连接domain1对应的IP。domain2是备用域名，仅在domain1不可连接时才会尝试连接domain2对应的IP。若发生主备切换，domain3和domain4将被标记为新的主集群，但将优先连接domain3对应的IP，仅在domain3不可连接时才会尝试连接domain4对应的IP。	类型：Integer 取值范围：0 ~ (url配置的域名数量-1) 默认值：0（关闭此功能）

参数名称	参数说明	参数值
refreshDomainResolveTime	指定域名解析结果的刷新周期，需配合parseCandidatesByDomain参数为true时才能生效。生效后会将url中的域名解析刷新时间设置为指定值，并在异步线程中定时更新域名解析结果。 须知 驱动仅通过应用所在环境的DNS服务获取域名解析结果，为保障域名绑定的变更被驱动及时感知，需应用所在环境的DNS服务保障域名绑定关系变更后能够及时生效。	类型：Integer 单位：s（秒） 取值范围：1 ~ 2147483647 默认值：10
refreshCNIPListTime	获取CN列表的缓存有效时间。JDBC在建立连接时会检测数据库集群中CN状态，在refreshCNIPListTime时间内，此状态被视为可信。超过该时间后状态失效，下次建立连接时会重新获取可用CN的IP列表。	类型：Integer 单位：s（秒） 取值范围：1 ~ 9999 默认值：10 建议：根据业务需求进行调整，并与autoBalance配合使用。
statementTimeout	指定statement执行的超时值。若statement执行时间超过该值，将取消当前执行操作。	类型：Integer 单位：ms（毫秒） 有效取值范围：0 ~ 2147483647。0及负数表示禁用超时功能。 默认值：0
executeUpdateQueryable	指定executeUpdate()方法是否允许执行DQL语句。 说明 建议通过executeQuery()方法执行DQL语句，因此不推荐启用此参数。	类型：Boolean 取值范围： • true：表示启用此功能。开启后，executeUpdate()方法执行DQL语句将返回-1，并可获取结果集。 • false：表示禁用此功能。 默认值：false

参数名称	参数说明	参数值
dbMonitor	指定是否开启JDBC连接监控功能。 说明 - 连接监控功能可监控以下指标：应用开启连接的次数、关闭连接的次数、连接异常断开的次数、数据库访问量、客户端CPU使用率、内存使用率、网络传输速率、网络时延、网络抖动及丢包率。 - 当dbMonitor设置为true、loggerLevel设置为debug、loggerFile设置为filePath时，可将客户端的连接监控数据输出到日志文件filePath中。具体配置示例，请参见监控数据库连接。	类型： Boolean 取值范围： - true：表示启用此功能。 - false：表示禁用此功能。 默认值： false
enableGaussArrayAndStruct	开启此功能后，会将投影列（或出参）为数组和集合类型的查询结果转换为GaussArray和GaussStruct对象。	类型： Boolean 取值范围： - true：表示启用此功能。 - false：表示禁用此功能。 默认值： false
getColumnReturnTypeScale	通过ResultSetMetaData#getColumns方法获取DECIMAL类型的小数位数时，设置是否将未指定小数位数的列返回null。	类型： Boolean 取值范围： - true：对于未指定小数位数的DECIMAL类型列，小数位数返回为null。 - false：对于未指定小数位数的DECIMAL类型列，小数位数返回为0。 默认值： false
extendTablePrivilege	通过PgDatabaseMetaData#getTablePrivileges方法获取用户对表的权限时，设置驱动行为与哪个开源数据库的驱动兼容。目前仅支持PostgreSQL数据库驱动的兼容。	类型： String 取值范围： PG：驱动行为与最新版本的PostgreSQL数据库的驱动兼容。 默认值： NULL

参数名称	参数说明	参数值
diffProcAndFunc	指定 DatabaseMetaData#getProcedures、DatabaseMetaData#getFunctions 方法是否区分存储过程和函数。	类型：Boolean 取值范围： • true：表示 DatabaseMetaData#getProcedures 只返回存储过程，DatabaseMetaData#getFunctions 只返回函数。 • false：表示 DatabaseMetaData#getProcedures、DatabaseMetaData#getFunctions 同时返回存储过程和函数。 默认值：false

5.3.2.3 执行 SQL 语句

应用程序通过执行 SQL 语句来操作数据库。本章节通过示例介绍 CREATE TABLE、INSERT、UPDATE、DELETE 和 SELECT 等操作。

- [执行普通 SQL 语句](#)
- [执行批量插入处理语句](#)
- [执行预处理更新语句](#)
- [执行删除语句](#)
- [执行查询语句](#)
- [调用存储过程](#)

执行普通 SQL 语句

前置条件：已连接数据库，连接对象为 conn。

执行普通 SQL 语句，创建 customer_t1 表的步骤如下：

步骤1 调用 Connection 接口的 createStatement 方法创建语句对象 stmt。

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

步骤2 调用 Statement 接口的 executeUpdate 方法执行 SQL 语句。

```
int rc = stmt.executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name VARCHAR(32));");
```

步骤3 调用 Statement 接口的 close 方法关闭语句对象 stmt。

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

----结束

📖 说明

- 数据库接收不在事务块中的执行请求，若包含多条语句时，将被打包成一个事务。事务块中不支持VACUUM操作。若任一语句执行失败，整个请求将被回滚。
- 使用Statement执行多条语句时，应以分号（;）作为各语句间的分隔符。存储过程、函数和匿名块不支持多条语句执行。当preferQueryMode=simple时，语句执行不进行逻辑解析，此场景下无法使用分号（;）作为多条语句间的分隔符。
- 斜杠（/）可作为创建存储过程、函数、匿名块、包体的结束符。当preferQueryMode=simple时，语句执行不进行逻辑解析，此场景下无法使用斜杠（/）作为结束符。
- JDBC会对prepareStatement中的SQL语句进行缓存，可能导致内存膨胀。若JVM内存较小，建议调整preparedStatementCacheSizeMiB或preparedStatementCacheQueries参数值。
- 在编写SQL语句时，若需添加注释，请避免注释内容紧贴SQL关键字。

执行批量插入处理语句

预处理语句仅编译和优化一次，可通过设置不同的参数值多次使用。由于已经预先编译完成，后续使用会减少执行时间。因此，对于需要重复执行的SQL操作，建议使用预处理语句。

前置条件：已通过[执行普通SQL语句](#)创建customer_t1表。

执行预处理语句，批量插入数据的步骤如下：

步骤1 调用Connection接口的prepareStatement方法创建预处理语句对象pst。

```
PreparedStatement pst = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?,?)");
```

步骤2 针对每条数据要调用对应接口设置参数，并调用addBatch方法将SQL语句添加到批处理中。

```
for (int i = 0; i < 3; i++) {  
    pst.setInt(1, i);  
    pst.setString(2, "data " + i);  
    pst.addBatch();  
}
```

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

```
pst.executeBatch();
```

步骤4 调用PreparedStatement接口的close方法关闭预处理语句对象pst。

```
pst.close();
```

----结束

📖 说明

在数据库批处理操作中，应遵循以下关键原则以确保性能：

1. 持续执行原则

- 批处理程序应保持持续执行，避免中途终止。
- 异常终止会显著影响数据库整体性能。

2. 事务管理优化

- 必须禁用自动提交功能。
`conn.setAutoCommit(false);`
- 采用分批提交策略而非单行提交。
- 根据数据量和系统资源确定合适的批量大小。

执行预处理更新语句

前置条件：已通过[执行批量插入处理语句](#)向customer_t1表插入数据。

执行预处理SQL语句对数据进行更新的步骤如下：

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

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE customer_t1 SET c_customer_name = ?  
WHERE c_customer_sk = 1");
```

步骤2 调用PreparedStatement接口的setString方法设置参数。

```
pstmt.setString(1, "new Data");
```

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

```
int rowcount = pstmt.executeUpdate();
```

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

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

----结束

须知

prepareStatement设置绑定参数后，最终会构建成B报文或U报文，并在下一步执行SQL语句时发送给服务端。但B报文或U报文存在最大长度限制（不能超过1023MB）。若一次绑定的数据过大，可能因报文过长导致异常。因此，prepareStatement设置绑定参数时需评估并控制数据大小，以避免超出报文上限。

执行删除语句

前置条件：已[执行预处理更新语句](#)。

执行删除语句的步骤如下：

步骤1 调用Connection接口的createStatement方法创建语句对象stmt。

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

步骤2 调用Statement接口的executeUpdate方法执行SQL语句，删除customer_t1表中c_customer_sk为0的记录。

```
int rd = stmt.executeUpdate("DELETE FROM customer_t1 WHERE c_customer_sk = 0;");
```

步骤3 调用Statement接口的close方法关闭语句对象stmt。

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

----结束

执行查询语句

前置条件：已[执行删除语句](#)。

执行查询语句的步骤如下：

步骤1 调用Connection接口的createStatement()方法创建语句对象stmts。

```
Statement stmts = conn.createStatement();
```

步骤2 调用Statement接口的executeQuery()方法执行SQL查询语句，并返回ResultSet。

```
ResultSet rs = stmt.executeQuery("SELECT * FROM customer_t1");
```

----结束

调用存储过程

GaussDB支持通过JDBC调用存储过程，前置条件是已连接数据库并且连接对象为conn。

创建存储过程的示例如下：

```
//在数据库中创建如下存储过程，它带有out参数。  
CREATE OR REPLACE PROCEDURE testproc  
(  
    psv_in1 in integer,  
    psv_in2 in integer,  
    psv_inout inout integer  
)  
AS  
BEGIN  
    psv_inout := psv_in1 + psv_in2 + psv_inout;  
END;  
/
```

调用存储过程的步骤如下：

步骤1 调用Connection的prepareCall方法创建调用语句对象cstmt。

```
CallableStatement cstmt = conn.prepareCall("{? = CALL testproc(?,?,?)}");
```

步骤2 调用CallableStatement的setInt方法设置参数。

```
cstmt.setInt(2, 50);  
cstmt.setInt(1, 20);  
cstmt.setInt(3, 90);
```

步骤3 调用CallableStatement的registerOutParameter方法注册输出参数。

```
cstmt.registerOutParameter(4, Types.INTEGER); //注册out类型的参数，类型为整型。
```

步骤4 调用CallableStatement的execute方法执行SQL语句。

```
cstmt.execute();
```

步骤5 调用CallableStatement的getInt方法获取out输出参数。

```
int out = cstmt.getInt(4);
```

步骤6 调用CallableStatement的close方法关闭调用语句对象cstmt。

```
cstmt.close();
```

----结束

说明

- 一些JDBC驱动程序提供命名参数的方法来设置参数。命名参数的方法允许根据名称而不是顺序来设置参数。若参数有默认值，则可以不指定参数值，而使用此参数的默认值。即使存储过程中参数的顺序发生了变更，也不必修改应用程序。目前GaussDB数据库的JDBC驱动程序不支持此方法。
- conn.prepareCall("{? = CALL testproc(?,?,?)}")，执行存储过程绑定参数时，可以按照占位符的顺序绑定参数并注册第一个参数为出参；也可以按照存储过程中的参数顺序绑定参数并注册第四个参数为出参，上述用例为此场景，注册第四个参数为出参。
- 当游标作为存储过程的返回值时，如果使用JDBC调用该存储过程，返回的游标将不可用。
- 存储过程不能和普通SQL在同一条语句中执行。
- 存储过程中inout类型参数必须注册为出参。

在语句中添加单分片执行语法

前置条件是已连接数据库，连接对象为conn，创建test表并插入数据。

步骤1 通过调用Connection对象的setClientInfo(String name,String value)方法来设置nodeName参数。

```
conn.setClientInfo("nodeName","datanode1");
```

步骤2 执行SQL语句，其中包括使用Statement对象的executeQuery(String sql)和execute(String sql)以及PreparedStatement对象的executeQuery()和execute()方法。

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM test");  
pstmt.execute();  
pstmt.executeQuery();  
Statement stmt=conn.createStatement();  
stmt.execute("SELECT * FROM test");  
stmt.executeQuery("SELECT * FROM test");
```

步骤3 关闭参数，将参数值设置为空字符串。

```
conn.setClientInfo("nodeName","");
```

----结束

须知

- 该功能基于单分片执行功能进行适配，所以使用前请确认使用的数据库是否支持单分片执行。
- nodeName参数开启后：
 - 执行SQL语句后需手动关闭参数，否则会对其他查询语句的执行产生影响。
 - 当前连接所有的语句都会到指定的DN上去执行。
 - PreparedStatement对象的缓存机制会受到影响，已经缓存的语句会被清空，之后执行的带单分片查询的语句都不再缓存，直到参数关闭后缓存功能恢复。
- 禁止多个线程对同一连接的nodeName参数进行修改。

5.3.2.4 处理结果集

执行SQL语句后，需要对结果集进行处理。本章节主要涵盖以下内容：设置结果集类型、在结果集中定位、获取结果集光标位置、获取结果集数据以及示例步骤。

设置结果集类型

不同类型的结果集具有不同的应用场景，应用程序需要根据实际情况选择合适的结果集类型。在执行SQL语句时，应先创建相应的语句对象，其中部分创建方法支持设置结果集类型。java.sql.Connection接口提供了三种方法，创建语句对象且支持设置结果集类型，如下所示：

```
//创建一个Statement对象，该对象将生成具有给定类型和并发性的ResultSet对象。  
createStatement(int resultSetType, int resultSetConcurrency);
```

```
//创建一个PreparedStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。  
prepareStatement(String sql, int resultSetType, int resultSetConcurrency);
```

```
//创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。  
prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```


表 5-23 结果集类型

参数	描述
resultSetType	表示结果集的类型，有以下三种： - ResultSet.TYPE_FORWARD_ONLY（默认值）：表示ResultSet光标只能向前移动。 - ResultSet.TYPE_SCROLL_SENSITIVE：修改后光标重新滚动到修改所在行，可以看到修改后的结果。 - ResultSet.TYPE_SCROLL_INSENSITIVE：修改后光标重新滚动到修改所在行，不能看到修改后的结果。 说明 ResultSet对象从数据库中获取了数据之后，即使类型是ResultSet.TYPE_SCROLL_SENSITIVE，也不会看到由其他事务在这之后引起的改变。调用ResultSet的refreshRow()方法，可获取当前光标所指的最新数据。
resultSetConcurrency	表示结果集的并发，有以下两种： - ResultSet.CONCUR_READ_ONLY（默认值）：只读结果集，不能直接通过结果集修改数据。 - ResultSet.CONCUR_UPDATEABLE：可改变的结果集。对于可滚动的结果集，可对结果集进行适当的改变。

在结果集中定位

ResultSet对象包含一个指向当前数据行的光标，初始状态位于结果集第一行之前。调用next()方法将光标移动至下一行，当已无下一行时，该方法返回false，因此可将其用于while循环中迭代结果集。对于可滚动结果集，JDBC驱动程序提供特定的定位方式，支持将光标移动至任意指定行。定位方法如表5-24所示。

表 5-24 在结果集中定位的方法

方法	描述
next()	把ResultSet光标向下移动一行。
previous()	把ResultSet光标向上移动一行。
beforeFirst()	把ResultSet光标定位到第一行之前。
afterLast()	把ResultSet光标定位到最后一行之后。
first()	把ResultSet光标定位到第一行。
last()	把ResultSet光标定位到最后一行。
absolute(int row)	把ResultSet光标移动到参数指定的行数。
relative(int rows)	rows为正数表示把ResultSet光标向下移动rows行，rows为负数表示把ResultSet光标向上移动（-rows）行。

获取结果集光标位置

对于可滚动的结果集，可调用定位方法来改变光标的位置。JDBC驱动程序提供了获取光标所在行的方法，如[表5-25](#)所示。

表 5-25 获取结果集光标位置

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

获取结果集数据

ResultSet对象提供了丰富的方法，以获取结果集中的数据。获取数据的常用方法如[表5-26](#)所示，其他方法请参见JDK官方文档。

表 5-26 获取结果集数据的常用方法

方法	描述
getInt(int columnIndex)	按列索引获取int型数据。
getInt(String columnLabel)	按列名获取int型数据。
getString(int columnIndex)	按列索引获取String型数据。
getString(String columnLabel)	按列名获取String型数据。
getDate(int columnIndex)	按列索引获取Date型数据。
getDate(String columnLabel)	按列名获取Date型数据。

示例步骤

[执行查询语句](#)后，对返回的结果集进行处理。

步骤1 使用while循环和next()方法逐行遍历结果集。

```
while (rs.next()) {  
    // 使用列索引获取值  
    int c_customer_sk = rs.getInt(1);    // 获取第一列的int型数据  
    String c_customer_name = rs.getString(2);    // 获取第二列的String型数据  
    System.out.println("c_customer_sk: " + c_customer_sk + ", c_customer_name: " + c_customer_name);  
}
```

步骤2 调用close()方法关闭资源。

```
rs.close(); // 关闭ResultSet对象  
stmts.close(); // 关闭Statement对象
```

----结束

5.3.2.5 关闭数据库连接

在完成相应的数据操作后，应调用close()方法关闭数据库连接。

```
conn.close();
```

📖 说明

Connection、Statement和ResultSet接口都提供close()方法，在使用完对象后应及时关闭它们。关闭Connection对象将间接关闭所有与之关联的Statement对象，关闭Statement对象则会间接关闭其对应的ResultSet对象。

5.3.3 典型应用开发示例

5.3.3.1 不同场景下配置连接参数

📖 说明

以下示例场景中node代表“host:port”，host为数据库服务器名称或IP地址，port为数据库服务器端口。

容灾场景

某客户有两套数据库集群，其中A集群为生产集群，B集群为容灾集群。当客户执行容灾切换时，A集群将降为容灾集群，B集群将升为生产集群。此时为了避免修改配置文件导致的应用重启或重新发版，客户可在初始配置文件时，即将A、B集群写入连接串中。此时在主集群不可连接时，驱动将尝试对容灾集群建立连接。例如A集群为{node1,node2,node3}。B集群为{node4,node5,node6}。

以Dorado双集群为例，url可参考如下命令进行配置，当主备切换后只会连接新的主集群。

```
jdbc:gaussdb://node1,node2,node3,node4,node5,node6/database?priorityServers=3
```

但需注意，以上配置在基于流式复制的异地容灾场景中，若对灾备集群实施灾备升主（failover），将导致双主集群同时运行，此时业务可能无法自动连接至预期的新主集群。针对此场景，可采用两种方案：

- 按照上述统一的url配置，但需通过额外管控手段在failover后对原主集群实施隔离。
- 将url按主备集群拆分为如下两条配置，在failover时由业务端主动切换连接。

```
jdbc:gaussdb://node1,node2,node3/database  
jdbc:gaussdb://node4,node5,node6/database
```

基于流式复制的异地容灾场景推荐使用容灾读业务方案：

主集群的url可参考如下命令进行配置：

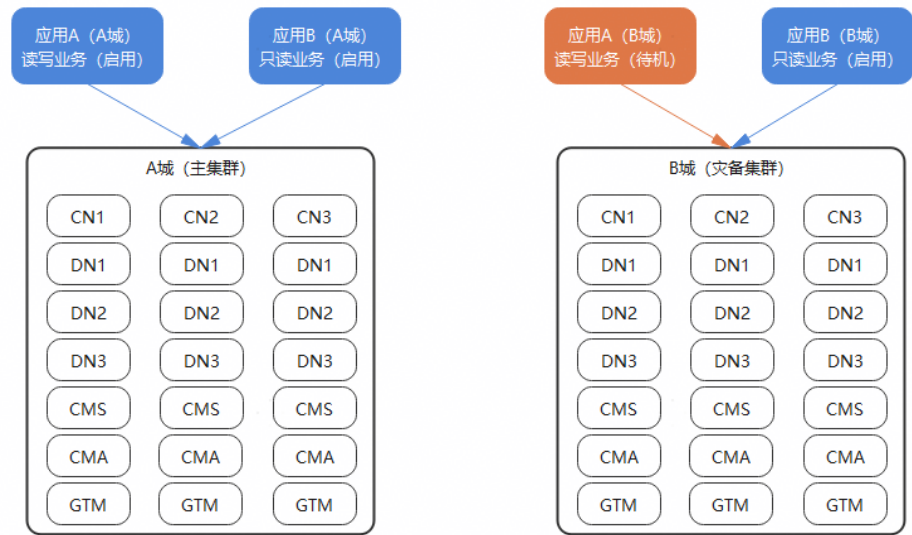
```
jdbc:gaussdb://node1,node2,node3/database?autoBalance=true&disasterToleranceCluster=true
```

灾备集群的url可参考如下命令进行配置：

```
jdbc:gaussdb://node4,node5,node6/database?autoBalance=true&disasterToleranceCluster=true
```

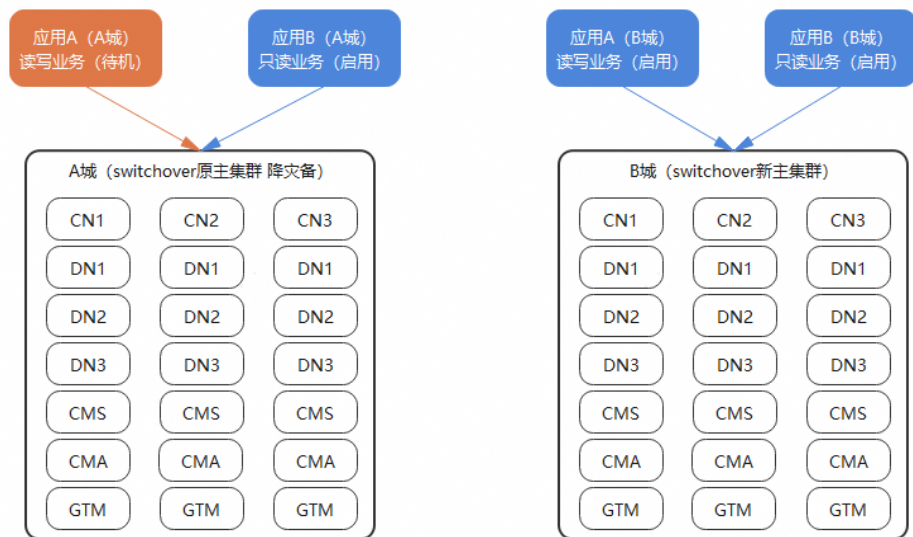
主集群和灾备集群应各配置一套读写业务与只读业务。读写应用A（A城）、只读应用B（A城）和只读应用B（B城）正常启用，读写应用A（B城）关闭业务流量。请参见图5-8。

图 5-8 正常运行时状态



当switchover发生时，数据库与应用分别进行容灾切换，读写应用A（A城）需先关闭业务流量，switchover成功后，读写应用A（B城）、只读应用B（B城）和只读应用B（A城）正常启用。请参见图5-9。

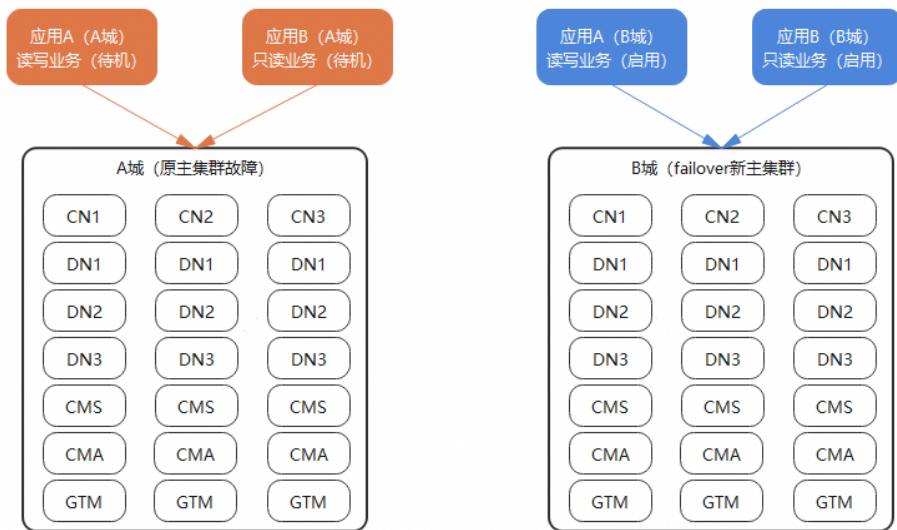
图 5-9 集群间 switchover 场景



当failover发生时，A城集群不可用，需先将读写应用A（A城）和只读应用B（A城）关闭业务流量以防止数据不一致问题，B城集群再执行升主failover操作。当B城集群

Normal后，读写应用A（B城）和只读应用B（B城）正常启用，读写应用A（A城）和只读应用B（A城）关闭业务流量。请参见图5-10。

图 5-10 failover 场景



负载均衡场景

某客户存在一套数据库集群，包含如下节点
{node1,node2,node3,node4,node5,node6,node7,node8,node9,node10,node11,node12}。

- 非透明多写特性下，客户在应用程序A中建立了120个长连接，并期望应用程序A上的连接可以均匀分布在集群各节点上，则url可参考如下命令进行配置。
jdbc:gaussdb://node1,node2,node3/database?autoBalance=true
- 非透明多写特性下，客户新开发了两个应用程序B、C，希望当前这三个应用程序均匀分布在指定节点，如应用程序A的连接分布在{node1,node2,node3,node4}，应用程序B的连接分布在{node5,node6,node7,node8}，应用程序C的连接分布在{node9,node10,node11,node12}，则url可参考如下命令进行配置（若url中前四个节点都不可连，应用会连接第五个节点，根据pgxc_node表中的信息，在集群中所有可用节点之间进行轮询连接）。
应用程序A：
jdbc:gaussdb://node1,node2,node3,node4,node5/database?autoBalance=priority4
应用程序B：
jdbc:gaussdb://node5,node6,node7,node8,node9/database?autoBalance=priority4
应用程序C：
jdbc:gaussdb://node9,node10,node11,node12,node13/database?autoBalance=priority4
- 非透明多写特性下，客户开发了一些应用程序，并且使用相同的连接配置串，同时期望各应用连接能较均匀分布在集群各节点上，则url可参考如下命令进行配置。
jdbc:gaussdb://node1,node2,node3,node4/database?autoBalance=shuffle
- 非透明多写特性下，客户开发了一些应用程序，想通过单节点会话模式建立连接，则url可参考如下命令进行配置。

```
jdbc:gaussdb://node1,node2,node3,node4/database?  
sessionType=singleNode&execTargetNode=dn_name
```

- 非透明多写特性下，客户不使用负载均衡功能，则url可参考如下命令进行配置。

```
jdbc:gaussdb://node1/database
```

或：

```
jdbc:gaussdb://node1/database?autoBalance=false
```

说明

非透明多写特性下，在开启autoBalance参数时，JDBC刷新可用CN列表的周期默认为10s，可使用refreshCNIPListTime进行设置，示例如下：

```
jdbc:gaussdb://node1,node2,node3,node4/database?autoBalance=true&refreshCNIPListTime=3
```

日志诊断场景

某客户在使用中出现数据导入慢或出现一些难以分析的异常报错时，可通过开启trace日志进行诊断，url可参考如下命令进行配置：

非透明多写特性

```
jdbc:gaussdb://node1/database?loggerLevel=trace
```

高性能场景

某客户对于相同SQL可能多次执行，仅是输入参数不同的情况，为了提升执行效率，可配置**prepareThreshold**参数，避免重复生成执行计划，url可参考如下命令进行配置：

非透明多写特性

```
jdbc:gaussdb://node1/database?prepareThreshold=5
```

某客户一次查询1000万行数据，为避免同时返回造成内存溢出，可使用**defaultRowFetchSize**，url可参考如下命令进行配置：

非透明多写特性

```
jdbc:gaussdb://node1/database?defaultRowFetchSize=50000
```

某客户需要批量插入1000万行数据，为提升效率，可使用**batchMode**，url可参考如下命令进行配置：

非透明多写特性

```
jdbc:gaussdb://node1/database?batchMode=on
```

大小写转换场景

在GaussDB中元数据默认存储为小写，如果从元数据默认存储为大写的数据库迁移至GaussDB，大写的元数据会变为小写。如果原业务中涉及到大写元数据的处理，可以开启uppercaseAttributeName参数，但是不建议通过这种方式来解决问题，建议通过修改业务代码来解决。如果一定要使用，请务必确认当前数据库中的元数据是否全为小写，以避免出现问题。url可参考如下命令进行配置：

非透明多写特性

```
jdbc:gaussdb://node1/database?uppercaseAttributeName=true
```

对于DatabaseMetaData中涉及的接口，按照入参直接调用即可，对于ResultSetMetaData中涉及的接口调用方法如下所示：

```
Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT * FROM test_supplier");  
ResultSetMetaData rsmd = rs.getMetaData();  
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
```



```
System.out.println(rsmd.getColumnLabel(i) + " " + rsmd.getColumnName(i));  
}
```

5.3.3.2 处理 SQL 异常

当JDBC操作（如执行SQL语句、处理结果集等）发生错误时，JDBC程序会抛出SQL异常。SQLException有getMessage()和printStackTrace()等常用方法，详细介绍请参见[java.sql.SQLException](#)。

以下代码展示了如何捕捉SQL异常。

```
try {  
    <some code>  
} catch(SQLException e) {  
    // System.out.println("exception: " + e.getMessage()); // 返回错误的描述信息  
    e.printStackTrace(); // 将当前异常的堆栈信息打印到标准错误流  
}
```

以使用错误的列索引为例，展示捕捉到SQL异常后的报错信息。若在处理结果集的[示例步骤](#)中，将rs.getString(2)写为rs.getString(3)。

- getMessage()方法使用示例

```
try {  
    while (rs.next()) {  
        int c_customer_sk = rs.getInt(1);  
        String c_customer_name = rs.getString(3); // 使用错误的列索引  
        System.out.println("c_customer_sk: " + c_customer_sk + ", c_customer_name: " +  
c_customer_name);  
    }  
} catch(SQLException e) {  
    System.out.println("exception: " + e.getMessage()); // 返回错误的描述信息  
}
```

运行程序后会出现以下信息：

```
exception: 栏位索引超过许可范围: 3, 栏位数: 2。
```

- printStackTrace()方法使用示例

```
try {  
    while (rs.next()) {  
        int c_customer_sk = rs.getInt(1);  
        String c_customer_name = rs.getString(3); // 使用错误的列索引  
        System.out.println("c_customer_sk: " + c_customer_sk + ", c_customer_name: " +  
c_customer_name);  
    }  
} catch(SQLException e) {  
    e.printStackTrace(); // 将当前异常的堆栈信息打印到标准错误流  
}
```

运行程序后会出现以下信息：

```
com.huawei.gaussdb.jdbc.util.PSQLException: 栏位索引超过许可范围: 3, 栏位数: 2。  
at com.huawei.gaussdb.jdbc.jdbc.PgResultSet.checkColumnIndex(PgResultSet.java:4005)  
at com.huawei.gaussdb.jdbc.jdbc.PgResultSet.checkResultSet(PgResultSet.java:4041)  
at com.huawei.gaussdb.jdbc.jdbc.PgResultSet.getString(PgResultSet.java:2754)  
at Customer.main(Customer.java:54)
```

5.3.3.3 创建和调用存储过程

本示例演示如何连接数据库、创建和调用存储过程。

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文  
存放，使用时解密），确保安全。  
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称  
请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
```

```
// $ip、$port、database需要用户自行修改。

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.CallableStatement;
import java.sql.Types;

public class DBTest {

    // 以非加密方式创建数据库连接。
    public static Connection GetConnection(String username, String passwd) {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        Connection conn = null;
        try {
            // 加载数据库驱动。
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    // 创建存储过程。
    public static void CreateCallable(Connection conn) {
        Statement stmt = null;
        try {
            stmt = conn.createStatement();
            // 创建存储过程，返回三个输入值的和。
            stmt.execute("CREATE OR REPLACE PROCEDURE TESTPROC \n" +
                "(\n" +
                "    psv_in1 in integer,\n" +
                "    psv_in2 in integer,\n" +
                "    psv_inout inout integer\n" +
                ")\n" +
                "as\n" +
                "begin\n" +
                "    psv_inout := psv_in1 + psv_in2 + psv_inout;\n" +
                "end;\n" +
                "/");
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }

    // 调用存储过程。
    public static void ExecCallableSQL(Connection conn) {
```



```
CallableStatement cstmt = null;
try {
    cstmt=conn.prepareCall("{? = CALL TESTPROC(?,?,?)}");
    cstmt.setInt(2, 50);
    cstmt.setInt(1, 20);
    cstmt.setInt(3, 90);
    cstmt.registerOutParameter(4, Types.INTEGER); //注册out类型的参数，类型为整型。
    cstmt.execute();
    int out = cstmt.getInt(4); //获取out参数。
    System.out.println("The CallableStatement TESTPROC returns:"+out);
    cstmt.close();
} catch (SQLException e) {
    if (cstmt != null) {
        try {
            cstmt.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
    e.printStackTrace();
}

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

    // 创建存储过程。
    CreateCallable(conn);

    // 调用存储过程。
    ExecCallableSQL(conn);

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

上述2个示例的运行结果为：

```
Connection succeed!
The CallableStatement TESTPROC returns:160
```

5.3.3.4 获取函数返回值

JDBC调用函数时获取返回值，以下示例展示返回值类型为bit和float8两种数据类型，其他数据类型可参考本示例。

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// $ip、$port、database需要用户自行修改。
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.CallableStatement;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Types;

public class Type {
    public static void main(String[] args) throws SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        Connection conn = null;

        try {
            // 加载数据库驱动。
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            // 以非加密方式创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
        }

        // 建表。
        String createsql = "CREATE TABLE IF NOT EXISTS t_bit(col_bit bit)";
        Statement stmt = conn.createStatement();
        stmt.execute(createsql);
        stmt.close();
        // bit类型使用示例，注意此处bit类型取值范围[0,1]。
        Statement st = conn.createStatement();
        String sqlstr = "CREATE OR REPLACE FUNCTION fun_1()\n" + "RETURNS bit AS $$\n"
            + "SELECT col_bit FROM t_bit LIMIT 1;\n" + "$$\n" + "LANGUAGE SQL;";
        st.execute(sqlstr);
        CallableStatement c = conn.prepareCall("{ ? = CALL fun_1() }");
        // 注册输出类型，位串类型。
        c.registerOutParameter(1, Types.BIT);
        c.execute();
        // 使用Boolean类型获取结果。
        System.out.println(c.getBoolean(1));

        // float8类型使用示例。
        st.execute("CREATE TABLE IF NOT EXISTS t_float(col1 float8)");
        PreparedStatement pstmt = conn.prepareStatement("INSERT INTO t_float VALUES(?)");
        pstmt.setDouble(1, 123456.123);
        pstmt.execute();
        pstmt.close();

        // 函数返回值为float8的使用示例。
        st.execute(
            "CREATE OR REPLACE FUNCTION func_float() " + "RETURN float8 " + "AS DECLARE " + "var1
float8; " + "BEGIN "
            + "SELECT col1 INTO var1 FROM t_float LIMIT 1; " + "RETURN var1; " + "END;");
        CallableStatement cs = conn.prepareCall("{ ? = CALL func_float() }");
        cs.registerOutParameter(1, Types.DOUBLE);
        cs.execute();
        System.out.println(cs.getDouble(1));
        st.close();

        // 关闭数据库连接。
        try {
```

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

上述2个示例的运行结果为：

```
Connection succeed!
false
123456.123
```

5.3.3.5 批量查询

此示例主要使用setFetchSize调整客户端内存使用，原理是通过数据库游标来分批获取服务端数据，但会加大网络交互，可能会损失部分性能。由于游标事务内有效，故需要先关闭自动提交事务，最后执行手动提交事务。

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。

// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

// \$ip、\$port、database需要用户自行修改。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;

public class Batch {
    public static void main(String[] args) throws SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        Connection conn = null;

        try {
            // 加载数据库驱动。
            Class.forName(driver).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            // 以非加密方式创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
        }

        // 关闭自动提交。
        conn.setAutoCommit(false);

        // 建表。
        Statement st = conn.createStatement();
        st.execute("CREATE TABLE mytable (cal1 int);");

        // 表中插入200行数据。
        PreparedStatement pstm = conn.prepareStatement("INSERT INTO mytable VALUES (?)");
        for (int i = 0; i < 200; i++) {
            pstm.setInt(1, i + 1);
        }
    }
}
```

```
        pstmt.addBatch();
    }
    pstmt.executeBatch();
    conn.commit();
    pstmt.close();

    // 打开游标，每次获取50行数据。
    st.setFetchSize(50);
    int fetchCount = 0;
    ResultSet rs = st.executeQuery("SELECT * FROM mytable");
    while (rs.next()) {
        fetchCount++;
    }
    System.out.println(fetchCount == 200);
    conn.commit();
    rs.close();

    // 关闭服务器游标。
    st.setFetchSize(0);
    fetchCount = 0;
    rs = st.executeQuery("SELECT * FROM mytable");
    while (rs.next()) {
        fetchCount++;
    }
    System.out.println(fetchCount == 200);
    conn.commit();
    rs.close();

    // 关闭语句对象、关闭数据库连接。
    st.close();
    conn.close();
}
}
```

上述2个示例的运行结果为：

```
Connection succeed!
true
true
```

执行完毕后可执行如下命令恢复自动提交事务。

```
conn.setAutoCommit(true);
```

5.3.3.6 使用流式读功能

流式读功能：读取数据时，服务端一次获取全部数据，发送到客户端socket缓冲区中，缓冲区占满则暂停，有空余则继续向缓冲区中发送数据，同时JVM逐行从缓冲区中读取数据。

优势是处理结果快，不会造成JVM内存溢出。劣势是只能向后遍历，数据处理完毕之前或者statement关闭之前，当前连接不能执行其他操作。

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// $ip、$port、database需要用户自行修改。

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class Stream {
    // 创建数据库连接。
    public static Connection getConnection(String username, String passwd) {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableStreamingQuery=true";
        Connection conn = null;
        try {
            //加载驱动。
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        try {
            //以非加密方式连接数据库。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        return conn;
    }

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

            //执行普通SQL语句。
            stmt.executeUpdate("DROP TABLE IF EXISTS t_user");
            stmt.executeUpdate("CREATE TABLE t_user(id int, name VARCHAR(20));");
            stmt.close();
        } catch (SQLException e) {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e1) {
                    e1.printStackTrace();
                }
            }
            e.printStackTrace();
        }
    }

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

        try {
            //生成预处理语句。
            pst = conn.prepareStatement("INSERT INTO t_user VALUES (?,?)");
            for (int i = 0; i < 20; i++) {
                //添加参数。
                pst.setInt(1, i + 1);
                pst.setString(2, "name " + (i + 1));
                pst.addBatch();
            }
            //执行批处理。
            pst.executeBatch();
            pst.close();
        } catch (SQLException e) {
            if (pst != null) {
                try {
                    pst.close();
                } catch (SQLException e1) {
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

```
    }  
    }  
    e.printStackTrace();  
  }  
}  
  
// 开启流式读，查询t_user表中内容。  
public static void StreamingQuery(Connection conn) {  
    PreparedStatement pst = null;  
    ResultSet resultSet = null;  
  
    try {  
        // 查询表t_user中的所有值。  
        pst = conn.prepareStatement("SELECT * FROM t_user ORDER BY id",  
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);  
        pst.setFetchSize(Integer.MIN_VALUE);  
        resultSet = pst.executeQuery();  
        while (resultSet.next()) {  
            System.out.println(resultSet.getInt(1));  
        }  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    } finally {  
        if (resultSet != null) {  
            try {  
                resultSet.close();  
            } catch (SQLException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
  
    if (pst != null) {  
        try {  
            pst.close();  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}  
  
public static void main(String[] args) throws Exception {  
    String userName = System.getenv("EXAMPLE_USERNAME_ENV");  
    String password = System.getenv("EXAMPLE_PASSWORD_ENV");  
  
    Connection conn = getConnection(userName, password);  
  
    CreateTable(conn);  
  
    BatchInsertData(conn);  
  
    StreamingQuery(conn);  
  
    //关闭数据库连接。  
    try {  
        conn.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

上述2个示例的运行结果为：

```
Connection succeed!  
1  
2  
3  
4  
5
```

```
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

须知

- 使用流式读功能时，结果集使用完之后，需要执行resultSet.close()或者statement.close()操作，否则当前连接将不可用。
- 创建Statement对象时，参数resultSetType和resultSetConcurrency的默认值分别为：TYPE_FORWARD_ONLY和CONCUR_READ_ONLY，如果修改该参数，会导致流式读功能失效。

5.3.3.7 应用层 SQL 重试

当主DN故障且10s未恢复时，GaussDB会自动将对应的备DN升主，使集群正常运行。备升主期间正在运行的作业会失败，备升主后启动的作业不会再受影响。如果要做到DN主备切换过程中，上层业务不感知，可参考此示例构建业务层SQL重试机制。

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。

// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

// \$ip、\$port、database需要用户自行修改。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class ExitHandler extends Thread {
    private Statement cancel_stmt = null;

    public ExitHandler(Statement stmt) {
        super("Exit Handler");
        this.cancel_stmt = stmt;
    }

    public void run() {
        System.out.println("exit handle");
        try {
            this.cancel_stmt.cancel();
        } catch (SQLException e) {
            System.out.println("cancel query failed.");
            e.printStackTrace();
        }
    }
}
```

```
public class SQLRetry {
    // 以非加密方式创建数据库连接。
    public static Connection GetConnection(String username, String passwd) {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        Connection conn = null;
        try {
            // 加载数据库驱动。
            Class.forName(driver).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    // 执行普通SQL语句，创建jdbc_test1表。
    public static void CreateTable(Connection conn) {
        Statement stmt = null;
        try {
            stmt = conn.createStatement();

            Runtime.getRuntime().addShutdownHook(new ExitHandler(stmt));

            // 执行普通SQL语句。
            int rc2 = stmt
                .executeUpdate("DROP TABLE if EXISTS jdbc_test1;");

            int rc1 = stmt
                .executeUpdate("CREATE TABLE jdbc_test1(col1 INTEGER, col2 VARCHAR(10));");

            stmt.close();
        } catch (SQLException e) {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e1) {
                    e1.printStackTrace();
                }
            }
            e.printStackTrace();
        }
    }

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

        try {
            // 生成预处理语句。
            pst = conn.prepareStatement("INSERT INTO jdbc_test1 VALUES (?,?)");
            for (int i = 0; i < 100; i++) {
                // 添加参数。
                pst.setInt(1, i);
                pst.setString(2, "data " + i);
                pst.addBatch();
            }
            // 执行批处理。
        }
    }
}
```



```
pst.executeBatch();
pst.close();
} catch (SQLException e) {
    if (pst != null) {
        try {
            pst.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
    e.printStackTrace();
}
}

// 执行预编译语句，查询数据。
private static boolean QueryRedo(Connection conn){
    PreparedStatement pstmt = null;
    boolean retValue = false;
    try {
        pstmt = conn
            .prepareStatement("SELECT col1 FROM jdbc_test1 WHERE col2 = ?");

        pstmt.setString(1, "data 10");
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            System.out.println("col1 = " + rs.getString("col1"));
        }
        rs.close();

        pstmt.close();
        retValue = true;
    } catch (SQLException e) {
        System.out.println("catch..... retValue " + retValue);
        if (pstmt != null) {
            try {
                pstmt.close();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
        }
        e.printStackTrace();
    }
}

System.out.println("finish.....");
return retValue;
}

// 查询语句，执行失败重试，重试次数可配置。
public static void ExecPreparedSQL(Connection conn) throws InterruptedException {
    int maxRetryTime = 10;
    int time = 0;
    String result = null;
    do {
        time++;
        try {
            System.out.println("time:" + time);
            boolean ret = QueryRedo(conn);
            if (ret == false){
                System.out.println("retry, time:" + time);
                Thread.sleep(10000);
                QueryRedo(conn);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    } while (null == result && time < maxRetryTime);
}
```

```
/**
 * 主程序，逐步调用各静态方法。
 * @param args
 * @throws InterruptedException
 */
public static void main(String[] args) throws InterruptedException {
    // 创建数据库连接。
    String userName = System.getenv("EXAMPLE_USERNAME_ENV");
    String password = System.getenv("EXAMPLE_PASSWORD_ENV");
    Connection conn = GetConnection(userName, password);

    // 创建表。
    CreateTable(conn);

    // 批量插入数据。
    BatchInsertData(conn);

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

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

上述2个示例的运行结果为：

```
Connection succeed!
time:1
col1 = 10
finish.....
time:2
col1 = 10
finish.....
time:3
col1 = 10
finish.....
time:4
col1 = 10
finish.....
time:5
col1 = 10
finish.....
time:6
col1 = 10
finish.....
time:7
col1 = 10
finish.....
time:8
col1 = 10
finish.....
time:9
col1 = 10
finish.....
time:10
col1 = 10
finish.....
exit handle
```

5.3.3.8 通过本地文件导入导出数据

在使用JAVA语言基于GaussDB数据库进行二次开发时，可以使用CopyManager接口，通过流方式将数据库中的数据导出到本地文件或者将本地文件导入数据库中，文件支持CSV、TEXT等格式。

代码运行的前提条件：在数据库中创建表migration_table和migration_table_1，并在migration_table表中插入数据。

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.sql.SQLException;
import com.huawei.gaussdb.jdbc.copy.CopyManager;
import com.huawei.gaussdb.jdbc.core.BaseConnection;

public class Copy {

    public static void main(String[] args) {
        String urls = new String("jdbc:gaussdb://$ip:$port/database"); //数据库URL
        String username = System.getenv("EXAMPLE_USERNAME_ENV"); //用户名
        String password = System.getenv("EXAMPLE_PASSWORD_ENV"); //密码
        String tablename = new String("migration_table"); //定义表信息
        String tablename1 = new String("migration_table_1"); //定义表信息
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        Connection conn = null;

        try {
            Class.forName(driver);
            conn = DriverManager.getConnection(urls, username, password); //以非加密方式连接数据库
        } catch (ClassNotFoundException e) {
            e.printStackTrace(System.out);
        } catch (SQLException e) {
            e.printStackTrace(System.out);
        }

        // 将SELECT * FROM migration_table查询结果导出到本地文件d:/data.txt中。
        try {
            copyToFile(conn, "d:/data.txt", "(SELECT * FROM migration_table)");
        } catch (SQLException e) {

            e.printStackTrace();
        } catch (IOException e) {

            e.printStackTrace();
        }
        //将d:/data.txt中的数据导入到migration_table_1中。
        try {
            copyFromFile(conn, "d:/data.txt", tablename1);
        } catch (SQLException e) {
            e.printStackTrace();
        } catch (IOException e) {

            e.printStackTrace();
        }

        // 将migration_table_1中的数据导出到本地文件d:/data1.txt中。
        try {
```

```
        copyToFile(conn, "d:/data1.txt", tablename1);
    } catch (SQLException e) {

        e.printStackTrace();
    } catch (IOException e) {

        e.printStackTrace();
    }
}

// 使用copyIn把数据从文件中导入数据库。
public static void copyFromFile(Connection connection, String filePath, String tableName)
    throws SQLException, IOException {

    FileInputStream fileInputStream = null;

    try {
        CopyManager copyManager = new CopyManager((BaseConnection) connection);
        fileInputStream = new FileInputStream(filePath);
        copyManager.copyIn("COPY " + tableName + " FROM STDIN", fileInputStream);
    } finally {
        if (fileInputStream != null) {
            try {
                fileInputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

// 使用copyOut把数据从数据库中导出到文件中。
public static void copyToFile(Connection connection, String filePath, String tableOrQuery)
    throws SQLException, IOException {

    FileOutputStream fileOutputStream = null;

    try {
        CopyManager copyManager = new CopyManager((BaseConnection) connection);
        fileOutputStream = new FileOutputStream(filePath);
        copyManager.copyOut("COPY " + tableOrQuery + " TO STDOUT", fileOutputStream);
    } finally {
        if (fileOutputStream != null) {
            try {
                fileOutputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

上述2个示例的运行结果为：本地d盘两个文件data.txt和data1.txt、数据库表migration_table_1，和数据库表migration_table数据相同。

5.3.3.9 逻辑复制（非透明多写）

本章节介绍JDBC逻辑解码的配置选项，并且通过示例演示JDBC接口使用逻辑复制功能。

须知

当前版本请使用直连DN解码，分布式强一致解码（CN解码）功能默认关闭，如有需要请联系华为工程师评估后打开enable-cn-decode参数。

配置选项

针对逻辑复制的配置选项，除了参考[逻辑解码选项](#)，还有专门给JDBC等流式解码工具增加的配置项，如下所示：

1. 解码线程并行度

通过配置选项parallel-decode-num，指定并行解码的Decoder线程数量。取值类型为int型，取值范围为1~20，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。当该选项配置为1时，禁止配置以下选项：解码格式选项decode-style、批量发送选项sending-batch和并行解码队列长度选项parallel-queue-size。

2. 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。除parallel_text_decoding、parallel_json_decoding、parallel_binary_decoding之外的其他插件类型情况下，decode-style默认值为'b'。该选项仅允许并行解码时设置，且二进制格式解码仅在并行解码场景下支持。对于json和text格式解码，在批量发送的解码结果中，每条解码语句的前4字节组成的uint32代表该条语句总字节数（不包含该uint32类型占用的4字节，0代表本批次解码结束），8字节uint64代表相应lsn（begin对应first_lsn，commit对应end_lsn，其他场景对应该条语句的lsn）。

说明

二进制格式编码规则如下所示：

1. 前4字节代表接下来到语句级别分隔符字母P（不含）或者该批次结束符F（不含）的解码结果的总字节数，该值如果为0代表本批次解码结束。
2. 接下来8字节uint64代表相应LSN（begin对应first_lsn，commit对应end_lsn，其他场景对应该条语句的lsn）。
3. 接下来1字节的字母有B/C/I/U/D，分别代表begin/commit/insert/update/delete。
4. 当第3步中的1字节的字母为B时：
 1. 接下来的8字节uint64代表CSN。
 2. 再接下来的8字节uint64代表first_lsn。
 3. 【可选】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. 【可选】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. 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. 当第3步中的1字节的字母为C时：
 1. 【可选】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. 【可选】接下来1字节字母如果为T，则代表后面的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. 当第3步中的1字节的字母为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. 【可选】接下来1字节字母如果为N代表为新元组，如果为O代表为旧元组，这里先发送新元组。
 1. 接下来的2字节uint16代表该元组需要解码的列数，记为attrnum。
 2. 以下流程重复attrnum次。
 1. 接下来2字节uint16代表列名的长度。
 2. 按照上述长度读取列名。
 3. 接下来4字节uint32代表当前列类型的OID。
 4. 接下来4字节uint32代表当前列的值（以字符串格式存储）的长度，如果为0xFFFFFFFF则表示NULL。如果为0xFFFFFFFF则表示该列为rowno系统列，实际rowno系统列长度为8字节；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，其中rowno系统列请按照8字节读取。
 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。

3. 限制仅备机解码

通过配置选项standby-connection，指定是否限制仅备机解码。其取值为Boolean型（可用0或1表示），取true（或1）代表限制仅允许连接备机解码，连接主机解码时会报错退出。取false（或0）时不做限制。默认值为false（0）。

4. 批量发送

通过配置选项sending-batch，指定是否批量发送。其取值范围为int型的0或1，取0表示逐条发送解码结果，取1表示解码结果累积到达1MB则批量发送解码结

果。默认值为0。该选项仅允许并行解码时设置。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。

5. 并行解码队列长度

通过配置选项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]，且必须为2的幂数，默认值为128。队列长度和解码过程的内存使用量正相关。

6. 逻辑解码内存阈值

逻辑解码内存阈值通过配置选项max-txn-in-memory指定单个事务解码中间结果缓存的内存阈值，单位为MB。并行解码模式下，该参数已废弃，不生效。串行解码模式下，取值范围为[0, 100]，默认值为0，表示不管控内存使用。

通过配置选项max-reorderbuffer-in-memory指定所有事务解码中间结果缓存的内存阈值，单位为GB。串行解码模式下，取值范围：[0, 100]。默认值为0，表示不管控内存使用。并行解码模式下，取值范围：[1, max_process_memory总量的50%]，默认值为1与max_process_memory/1048576*10%的较大值，其中1048576为kB到GB的单位转换，该参数已废弃，不生效。

并行解码通过配置选项max-logical-decode-memory指定所有事务解码中间结果缓存的内存阈值，单位为GB。取值范围：[6, 1024]，默认值为6。

当超过内存阈值时，解码过程将出现解码中间结果写临时文件的现象，影响逻辑解码的性能。临时文件的大小与事务修改的数据量成正比。如果临时文件过多，可能会导致磁盘进入只读状态的风险。

7. 逻辑解码表元信息内存阈值

逻辑解码表元信息通过配置选项desc-memory-limit指定逻辑解码任务维护的所有表元信息内存开销的上限，单位为MB，取值范围为[10, 1048576]，默认值为200。解码过程中，维护的表元信息总内存超过该阈值时，会触发表元信息的FIFO内存淘汰机制，清理部分内存。

8. 逻辑解码发送超时阈值

通过配置选项sender-timeout指定内核与客户端的心跳超时阈值。当该时间段内没有收到客户端任何消息，逻辑解码将主动停止，并断开和客户端的连接。单位为毫秒（ms），取值范围为[0, 2147483647]，默认值取决于GUC参数logical_sender_timeout配置。设置为0，表示逻辑解码不会主动断开和客户端的连接；如果设置过小，例如1ms，则可能存在解码任务中断的风险。

9. 逻辑解码用户黑名单选项

使用逻辑解码用户黑名单，逻辑解码结果将过滤黑名单中用户的事务操作。当前相关选项如下：

- a. exclude-userids：指定黑名单用户的OID，多个OID通过逗号分隔，不校验用户OID是否存在。同一个业务用户在不同DN的OID不一定相同。当前分布式直连DN逻辑解码需要传入业务用户在各DN上相应的OID，否则可能出现某些DN逻辑解码结果进行了过滤而某些DN节点未过滤的情况。
- b. exclude-users：指定黑名单用户名称，多个名称通过逗号分隔，通过dynamic-resolution设置是否动态解析识别用户名称。若解码报错用户不存在而出现中断，在确定日志产生时刻不存在对应的黑名单用户，可以通过配置dynamic-resolution为true或者从用户黑名单中删除报错用户名称来启动解码继续获取逻辑日志。
- c. dynamic-resolution：是否动态解析黑名单用户名称，默认值为true。设置为false时，当解码检测到黑名单exclude-users中用户不存在时，将报错并退出逻辑解码。设置为true时，当解码检测到黑名单exclude-users中用户不存在时继续解码。

10. 事务逻辑日志输出选项
- a. include-xids: 事务的BEGIN逻辑日志是否输出事务ID，默认值为true。

b. include-timestamp: 事务的BEGIN逻辑日志是否输出事务提交时间，默认值为false。

c. include-user: 事务的BEGIN逻辑日志是否输出事务的用户名称，默认值为false。事务的用户名称特指授权用户（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。
11. 逻辑复制槽名称
- withSlotName(String slotName): 用于唯一确定创建或解码的逻辑复制槽。

须知

- 输入的字符串如包含大写字母则会自动转化为小写字母。
- PGReplicationStream创建和使用的逻辑复制槽名称仅支持小写字母、数字以及“_”，长度最大为63。

12. 逻辑解码连接选项
- withStatusInterval: 用于调整超时时间，默认值为10000ms。

说明

备机解码在主机压力大时可能会导致连接超时关闭，通过配置withStatusInterval(10000,TimeUnit.MILLISECONDS)覆盖socketTimeout来解决连接超时关闭的问题。

13. 心跳日志输出选项
- enable-heartbeat: 是否输出心跳日志，默认值为false。

说明

- 以下对心跳日志进行解析，具体格式详见图5-11（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）。
- 二进制格式首先是字符'h'，表示消息是心跳日志。
 - 之后是心跳日志内容。第一个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；第二个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；第三个8字节uint64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。

图 5-11 心跳协议格式

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	uint64 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_decode_wal_time: XX"				'0'
Text/json+非批量发送	Char* "HeartBeat: latest_decode_lsn: XX, latest_flush_lsn: XX, latest_decode_wal_time: XX"						

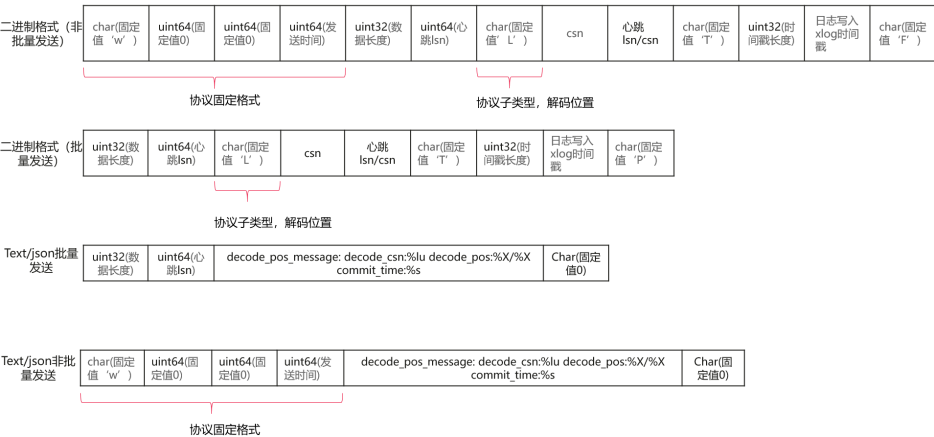
14. 逻辑解码控制参数，用于控制DDL的反解析流程以及输出形式：
- a. enable-ddl-decoding: 默认值为false，不开启DDL语句的逻辑解码。值为true时，开启DDL语句的逻辑解码。

- b. enable-ddl-json-format：默认值为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。
15. 位置日志输出选项
enable-decode-position：是否输出解码位置日志，默认值为false。

说明

- 以下对解码位置日志进行解析，具体格式详见图5-12。
- 二进制格式首先是字符'L'，表示消息是解码位置日志。
 - 之后是位置日志内容，依次为：第一个8字节uint64代表CSN，表示发送解码位置日志时对应的CSN值；第二个8字节uint64代表LSN或者CSN，当逻辑复制槽为lsn序时，表示发送解码位置日志时刻的WAL日志的位置，当复制槽为csn序时，表示发送解码位置日志时对应的CSN值；第三个8字节uint64代表时间戳（从1970年1月1日开始），表示调用gs_add_logical_decoding_position_xlog系统函数时产生的时间戳。
 - 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。

图 5-12 心跳协议格式



示例

须知

- 逻辑复制类PGReplicationStream为非线程安全类，禁止多线程使用同一个PGReplicationStream对象。
- PGReplicationStream创建和使用的逻辑复制槽名称仅支持小写字母、数字以及“_”，长度最大为63。

代码运行的前提条件：

- 添加JDBC用户机器IP到逻辑复制权限的白名单里。
- 将wal_level参数设置为logical，设置方法请联系管理员处理。
- 创建表t1和t2，并且对该表进行DDL或DML操作。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应加密存放，使用时解密），确保安全。  
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。  
// 如果采用DN解码、且表的分布方式为非Replication类型时，DN节点仅解码本地WAL日志，则需要连接所有的
```

DN节点并在所有的DN节点下创建逻辑复制槽、然后再进行数据汇总。如果只在单一节点执行，只会解码本节点的数据，导致无法接收全量日志。

// \$ip、\$port、database需要用户自行修改。

```
import com.huawei.gaussdb.jdbc.PGProperty;
import com.huawei.gaussdb.jdbc.PgConnection;
import com.huawei.gaussdb.jdbc.replication.LogSequenceNumber;
import com.huawei.gaussdb.jdbc.replication.PGReplicationStream;

import java.nio.ByteBuffer;
import java.sql.DriverManager;
import java.util.Properties;
import java.util.concurrent.TimeUnit;

public class LogicalReplicationDemo {
    private static PgConnection conn = null;
    public static void main(String[] args) {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        //此处配置数据库IP以及端口，这里的端口为haPort（High Availability Port，高可用端口），haPort通常
        //默认是所连接的DN端口号+1。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";

        //默认逻辑复制槽的名称是：replication_slot
        //测试模式：创建逻辑复制槽
        int TEST_MODE_CREATE_SLOT = 1;
        //测试模式：开启逻辑复制（前提条件是逻辑复制槽已经存在）
        int TEST_MODE_START_REPL = 2;
        //测试模式：删除逻辑复制槽
        int TEST_MODE_DROP_SLOT = 3;
        //开启不同的测试模式
        int testMode = TEST_MODE_START_REPL;

        try {
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            Properties properties = new Properties();
            PGProperty.USER.set(properties, System.getenv("EXAMPLE_USERNAME_ENV"));
            PGProperty.PASSWORD.set(properties, System.getenv("EXAMPLE_PASSWORD_ENV"));
            //对于逻辑复制，以下三个属性是必须配置项
            PGProperty.ASSUME_MIN_SERVER_VERSION.set(properties, "9.4");
            PGProperty.REPLICATION.set(properties, "database");
            PGProperty.PREFER_QUERY_MODE.set(properties, "simple");
            conn = (PgConnection) DriverManager.getConnection(sourceURL, properties);
            System.out.println("connection success!");

            if(testMode == TEST_MODE_CREATE_SLOT){
                conn.getReplicationAPI()
                    .createReplicationSlot()
                    .logical()
                    .withSlotName("replication_slot") //这里字符串如包含大写字母则会自动转化为小写字母
                    .withOutputPlugin("mppdb_decoding")
                    .make();
            } else if(testMode == TEST_MODE_START_REPL) {
                //开启此模式前需要创建复制槽
                LogSequenceNumber waitLSN = LogSequenceNumber.valueOf("6F/E3C53568"); //LSN需要用户根
                //据实际情况进行修改
                PGReplicationStream stream = conn
                    .getReplicationAPI()
                    .replicationStream()
                    .logical()
                    .withSlotName("replication_slot")
                    .withSlotOption("include-xids", true)
                    .withSlotOption("skip-empty-xacts", true)
                    // .withStartPosition(waitLSN) //解码日志开始输出的lsn位置
            }
        }
    }
}
```

```

.withSlotOption("parallel-decode-num", 10) //解码线程并行度
.withSlotOption("white-table-list", "public.t1,public.t2") //白名单列表
// .withSlotOption("standby-connection", true) //强制备机解码
.withSlotOption("decode-style", "t") //解码格式
.withSlotOption("sending-batch", 0) //批量发送解码结果
.withSlotOption("max-txn-in-memory", 100) //单个解码事务落盘内存阈值为100MB
.withSlotOption("max-reorderbuffer-in-memory", 2) //正在处理的解码事务落盘内存阈值为
2GB
// .withSlotOption("max-logical-decode-memory", 6) //直连DN并行解码正在处理的解码事务
落盘内存阈值为6GB
.withSlotOption("exclude-users", "userA") //不返回用户userA执行事务的逻辑日志
.withSlotOption("include-user", false) //事务BEGIN逻辑日志不携带用户名称
.withSlotOption("enable-heartbeat", true) // 开启心跳日志
.start();
while (true) {
    ByteBuffer byteBuffer = stream.readPending();

    if (byteBuffer == null) {
        TimeUnit.MILLISECONDS.sleep(10L);
        continue;
    }

    int offset = byteBuffer.arrayOffset();
    byte[] source = byteBuffer.array();
    int length = source.length - offset;
    System.out.println(new String(source, offset, length));

    //如果需要flush lsn，根据业务实际情况调用以下接口，每调用一次以下接口将触发数据库复制槽
    落盘逻辑，可能影响解码性能，建议每10s左右调用一次
    // LogSequenceNumber lastRecv = stream.getLastReceiveLSN();
    // stream.setFlushedLSN(lastRecv);
    // stream.forceUpdateStatus();

}
} else if (testMode == TEST_MODE_DROP_SLOT) {
    conn.getReplicationAPI()
        .dropReplicationSlot("replication_slot");
}
} catch (Exception e) {
    e.printStackTrace();
    return;
} finally {
    try {
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}
}
}

// 注意：如上代码不能直接读取二进制格式的日志，需按二进制格式编码规则读取。
```

text格式（即't'格式）解码结果示例如下：

```
BEGIN CSN: 2014 first_lsn: 0/2816A28
table public t1 INSERT: a[integer]:1 b[integer]:2 c[text]:'hello'
COMMIT XID: 15504
BEGIN CSN: 2015 first_lsn: 0/2816C20
table public t1 UPDATE: old-key: a[integer]:1 b[integer]:2 c[text]:'hello' new-tuple: a[integer]:1 b[integer]:5
c[text]:'hello'
COMMIT XID: 15505
BEGIN CSN: 2016 first_lsn: 0/2816D60
table public t1 DELETE: a[integer]:1 b[integer]:5 c[text]:'hello'
COMMIT XID: 15506
```

json格式（即"格式"）解码结果示例如下：

```
BEGIN CSN: 2014 first_lsn: 0/2816A28
{"table_name":"public.t1","op_type":"INSERT","columns_name":["a","b","c"],"columns_type":
["integer","integer","text"],"columns_val":["1","2","hello"],"old_keys_name":[],"old_keys_type":
[],"old_keys_val":[]}
COMMIT XID: 15504
```

```
BEGIN CSN: 2015 first_lsn: 0/2816C20
{"table_name":"public.t1","op_type":"UPDATE","columns_name":["a","b","c"],"columns_type":
["integer","integer","text"],"columns_val":["1","5","hello"],"old_keys_name":["a","b","c"],"old_keys_type":
["integer","integer","text"],"old_keys_val":["1","2","hello"]}
COMMIT XID: 15505
BEGIN CSN: 2016 first_lsn: 0/2816D60
{"table_name":"public.t1","op_type":"DELETE","columns_name":["a","b","c"],"columns_type":["integer","integer","text"],"columns_val":
["1","5","hello"],"old_keys_name":["a","b","c"],"old_keys_type":["integer","integer","text"],"old_keys_val":["1","5","hello"]}
COMMIT XID: 15506
```

5.3.3.10 监控数据库连接

连接监控功能支持监控JDBC端以下指标：应用开启连接的次数，关闭连接的次数，连接异常断开的次数，数据库访问量，客户端机器CPU的使用率，内存的使用率，上下行传输速率，应用端到数据库链路之间的网络时延、抖动、丢包率。此示例将演示如何使用JDBC驱动的连接监控功能。

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DBMonitorTest {

    public static void main(String[] args){
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String sourceURL
            = "jdbc:gaussdb://$ip.$port/database?
dbMonitor=true&loggerLevel=debug&loggerFile=dbMonitor.log";
        try {
            // 加载数据库驱动。
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
        }

        Connection conn = null;
        Statement statement = null;
        try {
            // 以非加密方式连接数据库。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");

            // 创建表。
            statement = conn.createStatement();
            String createTableQuery = "CREATE TABLE IF NOT EXISTS mytable (id INT PRIMARY KEY, name
VARCHAR(50))";
            statement.executeUpdate(createTableQuery);

            // 插入数据。
            String insertQuery = "INSERT INTO mytable (id, name) VALUES (1, 'John')";
            statement.executeUpdate(insertQuery);

            // 查询数据。
            String selectQuery = "SELECT * FROM mytable ";
            ResultSet resultSet = statement.executeQuery(selectQuery);
            while (resultSet.next()) {
```

```
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        System.out.println("id: " + id + ", name: " + name);
    }

    // 删除表。
    String dropTableQuery = "DROP TABLE IF EXISTS mytable";
    statement.executeUpdate(dropTableQuery);
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if (statement != null) {
            statement.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
}
```

上述2个示例的运行结果为：

```
Connection succeed!
id: 1, name: John
```

在日志文件dbMonitor.log中可以看到输出的连接监控信息，具体包括：

```
Nov 23, 2023 10:30:54 AM com.huawei.gaussdb.jdbc.qos.DataProcess saveQosResult
FINE: {
    "Destination host:port" : "localhost:8000",--服务器的IP和端口。
    "Delay" : "1.00 ms",--网络时延。
    "Jitter" : "0.04ms",--网络抖动。
    "Loss" : "0%",--网络丢包率。
    "DownloadSpeed" : "0.395Mbps",--网络下行速率。
    "UpLoadSpeed" : "0.498Mbps"--网络上行速率。
}

Nov 23, 2023 10:30:56 AM com.huawei.gaussdb.jdbc.CollectDBData saveCollectResult
FINE: {
    "openCount": "1",--应用开启数据库连接的次数。
    "closeCount": "1",--应用关闭数据库连接的次数。
    "abortedCount": "0",--连接异常断开的次数。
    "visitCount": "12",--应用对数据库的访问量。
    "cpuUsage": "20.39%",--客户端机器CPU的使用率。
    "memoryUsage": "98.32%",--客户端机器内存的使用率。
}
```

5.3.3.11 使用 Array 类型

以下示例将演示如何使用JDBC驱动的Array类型。

代码运行的前提条件：连接的数据库兼容模式为ORA，数据库版本需要大于等于503.0。

示例一：JDBC 驱动使用内核 table of 类型

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// $ip、$port、database需要用户自行修改。

import com.huawei.gaussdb.jdbc.jdbc.ArrayDescriptor;
import com.huawei.gaussdb.jdbc.jdbc.GaussArray;

import java.sql.*;

public class ArrayTest1 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE TYPE test_array is table of int");
        stmt.execute("CREATE OR REPLACE FUNCTION test_func RETURN test_array is\n" +
            "BEGIN\n" +
            "    RETURN test_array(123, 456, 789);\n" +
            "END;");

        stmt.execute("CREATE OR REPLACE PROCEDURE test_proc(v1 in test_array, v2 out test_array) is\n" +
            "BEGIN\n" +
            "    v2 := v1;\n" +
            "    v2(3) := 456;\n" +
            "END;");
    }

    /**
     * 清理数据库对象
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
    public static void cleanTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("DROP PROCEDURE test_proc");
        stmt.execute("DROP FUNCTION test_func");
    }
}
```

```
stmt.execute("DROP TYPE test_array");
}

/**
 * 使用array对象的相关接口
 *
 * @param array instance of Array
 * @throws SQLException if used array failed
 */
public static void testIntArray(Array array) throws SQLException {
    // 遍历打印元素数组
    Integer[] elements = (Integer[]) array.getArray();
    for (Integer element : elements) {
        System.out.println(element);
    }

    // 遍历打印部分元素
    Integer[] someElements = (Integer[]) array.getArray(2, 2);
    for (Integer element : someElements) {
        System.out.println(element);
    }

    // 打印元素类型名
    System.out.println(array.getBaseTypeName());

    // 打印元素的sqlType是否等于Types.INTEGER
    System.out.println(array.getBaseType() == Types.INTEGER);
}

/**
 * 构造array对象并使用其相关接口
 *
 * @param conn conn
 * @throws SQLException if create array failed
 */
public static void testConstructArray(Connection conn) throws SQLException {
    System.out.println("===== testConstructArray =====");
    String typeName = "test_array";
    ArrayDescriptor typeDesc = ArrayDescriptor.getDescriptor(typeName, conn);
    GaussArray array = new GaussArray(typeDesc, new Object[]{1, 2, 3});
    testIntArray(array);
}

/**
 * 获取function返回的array对象
 *
 * @param conn conn
 * @throws SQLException if an exception occurred
 */
public static void testReturnArrayParam(Connection conn) throws SQLException {
    System.out.println("===== testReturnArrayParam =====");

    // 执行存储过程，获取函数返回的array对象并执行Array相关接口
    PreparedStatement stmt = conn.prepareStatement("SELECT test_func()");
    ResultSet rs = stmt.executeQuery();
    rs.next();
    Array array = rs.getArray(1);
    testIntArray(array);
}

/**
 * 构造array对象并执行入参和出参
 *
 * @param conn conn
 * @throws SQLException if an exception occurred
 */
public static void testInputOutputArrayParam(Connection conn) throws SQLException {
    System.out.println("===== testInputOutputArrayParam =====");
    String typeName = "test_array";
```

```
ArrayDescriptor typeDesc = ArrayDescriptor.getDescriptor(typeName, conn);
GaussArray inArray = new GaussArray(typeDesc, new Object[]{1, 2, 3});

// 支持出参，需要数据库开启proc_outparam_override参数
Statement stmt = conn.createStatement();
stmt.execute("SET behavior_compat_options = 'proc_outparam_override'");

// 执行存储过程
CallableStatement cstmt = conn.prepareCall("{CALL test_proc(?, ?)}");
cstmt.setArray(1, inArray);
cstmt.registerOutParameter(2, Types.ARRAY, typeName);
cstmt.execute();

// 获取出参并执行Array相关接口
Array outArray = cstmt.getArray(2);
testIntArray(outArray);
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    // 构造数组对象
    testConstructArray(conn);

    // 数组返回值
    testReturnArrayParam(conn);

    // 数组入参和出参
    testInputOutputArrayParam(conn);

    // 删除测试对象
    cleanTestObject(conn);
}
}
```

上述2个示例的运行结果为：

```
===== testConstructArray =====
1
2
3
2
3
int4
true
===== testReturnArrayParam =====
123
456
789
456
789
int4
true
===== testInputOutputArrayParam =====
1
2
456
2
456
int4
true
```


示例二：Array 对象接口使用

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import com.huawei.gaussdb.jdbc.jdbc.ArrayDescriptor;
import com.huawei.gaussdb.jdbc.jdbc.GaussArray;

import java.sql.*;

public class ArrayTest2 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE TYPE test_array is table of int");
    }

    /**
     * 清理数据库对象
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
    public static void cleanTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("DROP TYPE test_array");
    }

    /**
     * 主程序，主流程。
     *
     * @param args args
     */
}
```

```
public static void main(String[] args) throws SQLException {
    // 创建数据库连接
    Connection conn = getConnection();

    // 创建前置测试对象
    prepareTestObject(conn);

    String typeName = conn.getSchema() + ".test_array";
    // 获取test_array类型的类型描述符
    ArrayDescriptor desc = ArrayDescriptor.getDescriptor(typeName, conn);

    // 根据类型描述符和元素数据创建GaussArray对象
    GaussArray array = new GaussArray(desc, new Object[]{1, 2, 3});

    // 获取array的类型描述符
    desc = array.getDescriptor();

    // 类型描述符相关接口使用, 非标准接口
    // 打印类型名是否等于$currentSchema.test_array
    System.out.println((conn.getSchema() + ".test_array").equals(desc.getSQLTypeName()));
    // 打印类型的sqlType是否等于Types.ARRAY
    System.out.println(desc.getSQLType() == Types.ARRAY);

    // array相关接口使用
    // 获取并遍历array的元素数组
    Integer[] elements = (Integer[]) array.getArray();
    for (Integer element : elements) {
        System.out.println(element);
    }

    // 获取并遍历array的部分元素数组
    elements = (Integer[]) array.getArray(1, 2);
    for (Integer element : elements) {
        System.out.println(element);
    }

    // 打印元素类型名
    System.out.println(array.getBaseTypeName());

    // 打印元素的sqlType是否等于Types.INTEGER
    System.out.println(array.getBaseType() == Types.INTEGER);

    // 删除测试对象
    cleanTestObject(conn);
}
```

上述2个示例的运行结果为：

```
true
true
1
2
3
1
2
int4
true
```

示例三：空元素转换

元素类型为字符类型时，空元素在传给数据库时会转换成null。

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
```

```
// $ip、$port、database需要用户自行修改。

import com.huawei.gaussdb.jdbc.jdbc.ArrayDescriptor;
import com.huawei.gaussdb.jdbc.jdbc.GaussArray;

import java.sql.*;

public class ArrayTest3 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE TYPE test_array is table of varchar(50)");
        stmt.execute("CREATE TABLE test_tab(c1 varchar(50))");
        stmt.execute("CREATE OR REPLACE PROCEDURE test_proc(v1 in test_array) is\n" +
            "BEGIN\n" +
            "    FOR i in 1..v1.count loop\n" +
            "        INSERT INTO test_tab VALUES(v1(i));\n" +
            "    END LOOP;\n" +
            "END;");
    }

    /**
     * 清理数据库对象
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
    public static void cleanTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("DROP PROCEDURE test_proc");
        stmt.execute("DROP TABLE test_tab");
        stmt.execute("DROP TYPE test_array");
    }

    /**
     * 将带有空元素的Array传给数据库
     *
     * @param conn conn
     */
}
```

```
* @throws SQLException if an exception occurred while executing the statement
*/
public static void inParamWithEmptyElement(Connection conn) throws SQLException {
    String typeName = "test_array";
    // 获取test_array类型的类型描述符
    ArrayDescriptor desc = ArrayDescriptor.getDescriptor(typeName, conn);
    // 根据类型描述符和元素数据创建GaussArray对象
    GaussArray array = new GaussArray(desc, new String[]{"", "", null});
    // 遍历并打印元素
    for (String s : (String[]) array.toArray()) {
        if (s == null) {
            System.out.println("s is null");
        } else {
            if (s.isEmpty()) {
                System.out.println("s is empty");
            }
        }
    }

    // 传入入参并执行存储过程
    CallableStatement cstmt = conn.prepareCall("{CALL test_proc(?)");
    cstmt.setObject(1, array, Types.ARRAY);
    cstmt.execute();

    // 查询结果
    PreparedStatement stmt = conn.prepareStatement("SELECT * FROM test_tab");
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        String s = rs.getString(1);
        if (s == null) {
            System.out.println("s is null");
        } else {
            if (s.isEmpty()) {
                System.out.println("s is empty");
            }
        }
    }
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    // 将带有空元素的array作为入参传给数据库
    inParamWithEmptyElement(conn);

    // 删除测试对象
    cleanTestObject(conn);
}
```

上述2个示例的运行结果为：

```
s is empty
s is empty
s is null
s is null
s is null
s is null
```

示例四：出参重载

实际注册的VARRAY类型与存储过程出参的VARRAY类型不一致，实际出参返回的实例类型为存储过程的出参类型。

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import com.huawei.gaussdb.jdbc.GaussArray;

import java.sql.*;

public class ArrayTest4 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE OR REPLACE PACKAGE test_pkg IS\n" +
            "    TYPE t1 IS varray(20) OF INT;\n" +
            "    TYPE t2 IS varray(20) OF VARCHAR(50);\n" +
            "    PROCEDURE proc_out_t1(v out t1);\n" +
            "END test_pkg;");

        stmt.execute("CREATE OR REPLACE PACKAGE BODY test_pkg IS\n" +
            "    PROCEDURE proc_out_t1(v out t1) IS\n" +
            "    BEGIN\n" +
            "        v := t1(1, 2, 3, 4, 5);\n" +
            "    END;\n" +
            "END test_pkg;");
    }

    /**
     * 清理数据库对象
     */
}
```

```
* @param conn conn
* @throws SQLException if an exception occurred while executing the statement
*/
public static void cleanTestObject(Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();
    stmt.execute("DROP PACKAGE test_pkg");
}

/**
 * 注册重载类型
 *
 * @param conn conn
 * @throws SQLException if an exception occurred while executing statement
 */
public static void testRegisterDiffType(Connection conn) throws SQLException {
    // 支持出参，需要数据库开启proc_outparam_override参数
    Statement stmt = conn.createStatement();
    stmt.execute("SET behavior_compat_options = 'proc_outparam_override'");
    CallableStatement cstmt = conn.prepareCall("{CALL test_pkg.proc_out_t1(?)}}");

    // 注册类型为t2
    cstmt.registerOutParameter(1, Types.ARRAY, "test_pkg.t2");
    cstmt.execute();

    // 实际获取类型为t1 getArray返回类型为Integer[]
    GaussArray array = (GaussArray)cstmt.getArray(1);
    // 遍历元素
    for (Integer element : (Integer[]) array.getArray()) {
        System.out.println(element);
    }
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    // 注册不同类型参数
    testRegisterDiffType(conn);

    // 删除测试对象
    cleanTestObject(conn);
}
}
```

上述2个示例的运行结果为：

```
1
2
3
4
5
```

须知

- 类型名称大小写敏感，且不支持类型名称有小数点的类型。
- 类型名称不支持同义词。
- 支持类型名称格式为schema.package.type、package.type、schema.type、type，若未传入schema名，默认按当前schema处理，若其中的名称包含小数点，需要用双引号包裹。
- 要想使用内置原生数组类型构造ArrayDescriptor，传入的类型名不能省略schema（当前schema下没有内置原生数组类型）。
- 元素支持的基础类型为：int2、int4、int8、float4、float8、numeric、bool、bpchar、varchar、nvarchar2、name、text、timestamp、timestampz、time、timetz、clob、bytea、blob。上述类型可能存在的别名，如：smallint、int、integer、bigint、number、float、boolean、char、varchar2等。
- 元素支持的自定义类型：数组、集合和record类型。
- JDBC开启URL参数enableGaussArrayAndStruct后，支持获取集合或数组类型的出参和返回值为Array对象。支持出参还需要数据库额外开启GUC参数behavior_compat_options = 'proc_outparam_override'。
- 不支持出参或返回值为多维数组；不支持出参或返回值的数组起始下标不为1。
- JDBC端不支持对类型修饰符进行校验，依赖于数据库校验，例如：
 1. 不支持元素长度校验，如table of varchar(5)类型，构造Array时传入字符串长度大于5，JDBC端不报错。
 2. 不支持元素精度校验，如table of numeric(3)类型，构造Array时传入数值位数大于3，JDBC端不报错。
 3. 不支持元素精度转换，如varray of numeric(3, 1)类型，构造Array时传入数值为2.55，JDBC端不会将其转换为2.6。
 4. 不支持varray长度校验，如varray(2) of int类型，构造Array时传入元素个数大于2，JDBC端不报错。
- 当传入数据库的array对象包含空元素时，数据库会将这些空元素转换为null。例如：当元素类型为字符类型时，如果元素为空，传给数据库时会转换为null；当元素类型为CLOB时，如果CLOB元素存储的字符串长度为0，传给数据库时会转换为null；当元素类型为BLOB时，如果BLOB元素存储的二进制数组长度为0，传给数据库时会转换为null；当元素类型为BYTEA时，如果BYTEA元素存储的二进制数组长度为0，传给数据库时会转换为null。

不推荐在构造的array对象中存储空元素，因为空元素在传给数据库时会转换为null，具体示例请参见[示例三：空元素转换](#)。

当数据库传给客户端的array对象包含空元素时，客户端也会将这些空元素转为null。
- 不同数组类型之间支持隐式转换，当实际注册的类型与存过出参类型不一致，实际出参返回的实例类型为存过的出参类型，可参考[示例四：出参重载](#)。
- Array接口的相关说明可参考GaussArray对象支持的标准接口下方的说明。
- ArrayDescriptor为非标准接口，ArrayDescriptor的getSQLType接口固定返回java.sql.Types.ARRAY。

ArrayDescriptor的getSQLTypeName接口返回的类型名可参考[数据类型映射关系](#)。

5.3.3.12 使用 Struct 类型

以下示例将演示如何使用JDBC驱动的Struct类型。

代码运行的前提条件：连接的数据库兼容模式为ORA，数据库版本需要大于等于503.0。

示例一：JDBC 驱动使用内核 record 类型

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import com.huawei.gaussdb.jdbc.jdbc.StructDescriptor;
import com.huawei.gaussdb.jdbc.jdbc.GaussStruct;

import java.sql.*;

public class StructTest1 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        // 支持出参，需要数据库开启proc_outparam_override参数
        stmt.execute("SET behavior_compat_options = 'proc_outparam_override'");
        // 创建测试对象
        stmt.execute("CREATE OR REPLACE PACKAGE test_pkg IS\n" +
            "    TYPE test_rec IS record(col1 int, col2 varchar(50));\n" +
            "    FUNCTION test_func RETURN test_rec;\n" +
            "    PROCEDURE test_proc(v1 in test_rec, v2 out test_rec);\n" +
            "END test_pkg;");
        stmt.execute("CREATE OR REPLACE PACKAGE BODY test_pkg IS\n" +
            "    FUNCTION test_func RETURN test_rec IS\n" +
```



```
        "    v test_rec;" +
        " BEGIN\n" +
        "    v.col1 := 123;\n" +
        "    v.col2 := 'abc';\n" +
        "    RETURN v;\n" +
        " END;\n" +
        " PROCEDURE test_proc(v1 in test_rec, v2 out test_rec) IS\n" +
        " BEGIN\n" +
        "    v2.col1 := v1.col1 + 1;\n" +
        "    v2.col2 := v1.col2 || 'd';\n" +
        " END;\n" +
        "END test_pkg;");
    }

    /**
     * 清理数据库对象
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
    public static void cleanTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("DROP PACKAGE test_pkg");
    }

    /**
     * 使用struct对象的相关接口
     *
     * @param struct instance of Struct
     * @throws SQLException if used Struct failed
     */
    public static void testStruct(Struct struct) throws SQLException {
        // 遍历打印元素
        Object[] attributes = struct.getAttributes();
        for (Object attribute : attributes) {
            System.out.println(attribute);
        }
    }

    /**
     * 构造Struct对象并使用其相关接口, 非标准构造方法
     *
     * @param conn conn
     * @throws SQLException if create Struct failed
     */
    public static void testConstructStruct1(Connection conn) throws SQLException {
        System.out.println("===== testConstructStruct1 =====");
        String typeName = "test_pkg.test_rec";
        StructDescriptor typeDesc = StructDescriptor.getDescriptor(typeName, conn);
        GaussStruct struct = new GaussStruct(typeDesc, new Object[]{666, "aaabbbccc"});
        testStruct(struct);
    }

    /**
     * 构造Struct对象并使用其相关接口, 标准接口构造
     *
     * @param conn conn
     * @throws SQLException if create Struct failed
     */
    public static void testConstructStruct2(Connection conn) throws SQLException {
        System.out.println("===== testConstructStruct2 =====");
        String typeName = "test_pkg.test_rec";
        Struct struct = conn.createStruct(typeName, new Object[]{666, "aaabbbccc"});
        testStruct(struct);
    }

    /**
     * 获取function返回的Struct对象
     *
     */
}
```

```
* @param conn conn
* @throws SQLException if an exception occurred
*/
public static void testReturnStructParam(Connection conn) throws SQLException {
    System.out.println("===== testReturnStructParam =====");

    // 执行存储过程，获取函数返回的Struct对象并执行Struct相关接口
    PreparedStatement stmt = conn.prepareStatement("SELECT test_pkg.test_func()");
    ResultSet rs = stmt.executeQuery();
    rs.next();
    Struct Struct = (Struct) rs.getObject(1);
    testStruct(Struct);
}

/**
 * 构造Struct对象并执行入参和出参
 *
 * @param conn conn
 * @throws SQLException if an exception occurred
 */
public static void testInputOutputStructParam(Connection conn) throws SQLException {
    System.out.println("===== testInputOutputStructParam =====");
    String typeName = "test_pkg.test_rec";
    StructDescriptor typeDesc = StructDescriptor.getDescriptor(typeName, conn);
    GaussStruct inStruct = new GaussStruct(typeDesc, new Object[]{123, "abc"});

    // 执行存储过程
    CallableStatement cstmt = conn.prepareCall("{CALL test_pkg.test_proc(?, ?)}");
    cstmt.setObject(1, inStruct);
    cstmt.registerOutParameter(2, Types.STRUCT, typeName);
    cstmt.execute();

    // 获取出参并执行Struct相关接口
    Struct outStruct = (Struct) cstmt.getObject(2);
    testStruct(outStruct);
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    // 构造Struct对象方式1
    testConstructStruct1(conn);

    // 构造Struct对象方式2
    testConstructStruct2(conn);

    // Struct返回值
    testReturnStructParam(conn);

    // Struct入参和出参
    testInputOutputStructParam(conn);

    // 删除测试对象
    cleanTestObject(conn);
}
}
```

上述2个示例的运行结果为：

```
===== testConstructStruct1 =====
666
```

```
aaabbbccc
===== testConstructStruct2 =====
666
aaabbbccc
===== testReturnStructParam =====
123
abc
===== testInputOutputStructParam =====
124
abcd
```

示例二：Struct 对象接口使用

非透明多写特性下的示例

// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// \$ip、\$port、database需要用户自行修改。

```
import com.huawei.gaussdb.jdbc.jdbc.GaussStruct;
import com.huawei.gaussdb.jdbc.jdbc.StructDescriptor;

import java.sql.*;

public class StructTest2 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE OR REPLACE PACKAGE test_pkg IS\n" +
            "    TYPE test_rec IS record(col1 int, col2 varchar(50));\n" +
            "END test_pkg;");
    }

    /**
     * 清理数据库对象
     */
}
```

```
* @param conn conn
* @throws SQLException if an exception occurred while executing the statement
*/
public static void cleanTestObject(Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();
    stmt.execute("DROP PACKAGE test_pkg");
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    String typeName = conn.getSchema() + ".test_pkg.test_rec";
    // 获取test_array类型的类型描述符
    StructDescriptor desc = StructDescriptor.getDescriptor(typeName, conn);

    // 根据类型描述符和元素数据创建GaussStruct对象
    GaussStruct struct = new GaussStruct(desc, new Object[]{123, "abc"});

    // 获取record的类型描述符
    desc = struct.getDescriptor();

    // 类型描述符相关接口使用, 非标准接口
    // 打印类型名是否等于$currentSchema.test_pkg.test_rec
    System.out.println(typeName.equals(desc.getSQLTypeName()));
    // 打印类型的sqlType是否等于Types.STRUCT
    System.out.println(desc.getSQLType() == Types.STRUCT);

    // Struct相关接口使用
    // 获取并遍历struct的元素
    Object[] attributes = struct.getAttributes();
    for (Object attribute : attributes) {
        System.out.println(attribute);
    }

    // 删除测试对象
    cleanTestObject(conn);
}
}
```

上述2个示例的运行结果为：

```
true
true
123
abc
```

示例三：空元素转换

元素类型为字符类型时，空元素在传给数据库时会转换成null。

非透明多写特性下的示例

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
// $ip、$port、database需要用户自行修改。
```

```
import java.sql.*;
```

```
public class StructTest3 {
    // 以非加密方式创建数据库连接
    public static Connection getConnection() {
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableGaussArrayAndStruct=true";
        Connection conn = null;
        try {
            // 加载数据库驱动
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        try {
            // 创建数据库连接
            conn = DriverManager.getConnection(sourceURL, username, passwd);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

        return conn;
    }

    /**
     * 创建前置数据库对象
     *
     * @param conn conn
     * @throws SQLException An exception occurred while executing the statement
     */
    public static void prepareTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("CREATE TABLE test_tab(c1 varchar(50))");

        stmt.execute("CREATE OR REPLACE PACKAGE test_pkg IS\n" +
            "    TYPE test_rec IS record(col1 varchar(50), col2 varchar(50), col3 varchar(50));\n" +
            "    PROCEDURE test_proc(v1 in test_rec);\n" +
            "END test_pkg;");

        stmt.execute("CREATE OR REPLACE PACKAGE BODY test_pkg IS\n" +
            "    PROCEDURE test_proc(v1 in test_rec) IS\n" +
            "        BEGIN\n" +
            "            INSERT INTO test_tab VALUES(v1.col1);\n" +
            "            INSERT INTO test_tab VALUES(v1.col2);\n" +
            "            INSERT INTO test_tab VALUES(v1.col3);\n" +
            "        END;\n" +
            "END test_pkg;");
    }

    /**
     * 清理数据库对象
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
    public static void cleanTestObject(Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();
        stmt.execute("DROP PACKAGE test_pkg");
        stmt.execute("DROP TABLE test_tab");
    }

    /**
     * 将带有空元素的Struct传给数据库
     *
     * @param conn conn
     * @throws SQLException if an exception occurred while executing the statement
     */
}
```

```
*/
public static void inParamWithEmptyElement(Connection conn) throws SQLException {
    String typeName = "test_pkg.test_rec";
    // 创建Struct对象
    Struct struct = conn.createStruct(typeName, new Object[]{"", "", null});
    // 遍历并打印元素
    for (Object attribute : struct.getAttributes()) {
        if (attribute == null) {
            System.out.println("attribute is null");
        } else {
            if (((String) attribute).isEmpty()) {
                System.out.println("attribute is empty");
            }
        }
    }
}

// 传入入参并执行存储过程
CallableStatement cstmt = conn.prepareCall("{CALL test_pkg.test_proc(?)}");
cstmt.setObject(1, struct, Types.STRUCT);
cstmt.execute();

// 查询结果
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM test_tab");
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    String s = rs.getString(1);
    if (s == null) {
        System.out.println("s is null");
    } else {
        if (s.isEmpty()) {
            System.out.println("s is empty");
        }
    }
}
}

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

    // 创建前置测试对象
    prepareTestObject(conn);

    // 将带有空元素的struct作为入参传给数据库
    inParamWithEmptyElement(conn);

    // 删除测试对象
    cleanTestObject(conn);
}
}
```

上述2个示例的运行结果为：

```
attribute is empty
attribute is empty
attribute is null
s is null
s is null
s is null
```

须知

- 类型名称大小写敏感，且不支持类型名称有小数点的类型。
- 类型名称不支持同义词。
- 支持类型名称格式为schema.package.type、package.type、schema.type、type，若未传入schema名，默认按当前schema处理，若其中的名称包含小数点，需要用双引号包裹。
- 元素支持的基础类型为：int2、int4、int8、float4、float8、numeric、bool、bpchar、varchar、nvarchar2、name、text、timestamp、timestampz、time、timetz、clob、bytea、blob。上述类型可能存在的别名，如：smallint、int、integer、bigint、number、float、boolean、char、varchar2等。
- 元素支持的自定义类型：数组、集合和record类型。
- JDBC开启URL参数enableGaussArrayAndStruct后，支持获取集合或数组类型的出参和返回值为Struct对象。支持出参还需要数据库额外开启GUC参数behavior_compat_options = 'proc_outparam_override'。
- JDBC端不支持对类型修饰符进行校验，依赖于数据库校验，例如：
 1. 不支持元素精度转换，如record(c1 numeric(3, 1), c2 numeric(3, 1))类型，构造Struct时传入的列元素数值为2.55，JDBC端不会将其转换为2.6。
 2. 不支持元素长度校验，如record(c1 varchar(5), c2 varchar(5))类型，构造Struct时传入字符串长度大于5，JDBC端不报错。
 3. 不支持not null校验，如record(c1 int not null, c2 int)类型，构造Struct时第一列元素为null，JDBC端不报错。
- 当传入数据库的struct对象包含空元素时，数据库会将这些空元素转换为null。例如：当元素类型为字符类型时，如果元素为空，传给数据库时会转换为null；当元素类型为CLOB时，如果CLOB元素存储的字符串长度为0，传给数据库时会转换为null；当元素类型为BLOB时，如果BLOB元素存储的二进制数组长度为0，传给数据库时会转换为null；当元素类型为BYTEA时，如果BYTEA元素存储的二进制数组长度为0，传给数据库时会转换为null。

不推荐在构造的struct对象中存储空元素，因为空元素在传给数据库时会转换为null。例如，元素为空时，会转换为null，具体示例可参考上述示例三。

当数据库传给客户端的struct对象包含空元素时，客户端也会将这些空元素转为null。
- Struct接口的相关说明可参考GaussStruct对象支持的标准接口下方的说明。
- StructDescriptor为非标准接口，StructDescriptor的getSQLType接口固定返回java.sql.Types.STRUCT。

StructDescriptor的getSQLTypeName接口返回的类型名可参考[数据类型映射关系](#)。

5.3.4 JDBC 接口参考

JDBC接口是一套提供给用户的API方法，本节将对部分常用接口做具体描述，若涉及其他接口可参考JDK1.8（软件包）/JDBC4.2中相关内容。

5.3.4.1 常用 Package 及接口

常用Package及接口的作用，如[表5-27](#)所示。

表 5-27 常用 Package 及接口

常用Package	作用	常用API接口
java.sql	java.sql包是JDBC最核心、最常用的Package。java.sql包提供了访问数据库的标准API，用户可使用API接口来连接数据库、执行SQL语句和处理结果。	● Connection：表示与数据库的连接。 ● DriverManager：用于管理JDBC驱动程序的加载和连接建立。 ● Statement：用于执行静态SQL语句并返回结果。 ● PreparedStatement：用于执行预编译的SQL语句。 ● CallableStatement：用于执行数据库存储过程。 ● ResultSet：用于处理从数据库查询返回的结果集。 ● ResultSetMetaData：描述ResultSet对象的相关信息。 ● DatabaseMetaData：描述数据库特定的信息。 ● SQLException：用于处理数据库操作中发生的异常。 ● Array、Blob、Clob、NClob：用于处理数据库中的大对象数据类型。
javax.sql	javax.sql包侧重于提供企业级服务，提供了更高级的数据库连接功能，例如连接池、分布式事务、行集（RowSet）等。	● DataSource：提供了一种更灵活的方式来获取数据库连接，支持连接池等高级功能。 ● ConnectionPoolDataSource：表示数据源连接池接口。 ● PooledConnection：表示由连接池创建的连接接口。 ● RowSet：表示可滚动的结果集接口。

常用Package	作用	常用API接口
java.util	java.util包是Java标准库中的一个核心工具包，提供了通用的数据结构、算法和实用工具所需的接口和类。	- Collection：定义了所有单列集合（List、Set及Queue）共有的基本操作。 - Map：表示键值对（Key-Value）的集合。 - Properties：用于处理键值对（key-value）的配置信息，尤其是以字符串为键和值的配置。
java.io	java.io包提供了处理输入/输出操作所需的所有接口和类。	- InputStream：字节输入流的顶层接口，用于按字节读取数据。 - OutputStream：字节输出流的顶层接口，用于按字节写出数据。 - Reader：字符输入流的顶层接口，用于按字符读取数据。 - Writer：字符输出流的顶层接口，用于按字符写出数据。 - IOException：用于表示输入/输出操作期间发生的错误或中断。
com.huawei.gaussdb.jdbc	GaussDB对JDBC的扩展。	- CopyManager：用于批量向数据库导入数据以及从数据库导出数据。 - PGReplicationConnection：用于执行逻辑复制相关的功能。 - PGReplicationStream：用于操作逻辑复制流。 - ChainedStreamBuilder：用于构建复制流。 - ChainedCommonStreamBuilder：用于为逻辑和物理复制指定通用参数。

5.3.4.2 java.sql.Connection

java.sql.Connection是数据库连接接口。

表 5-28 对 java.sql.Connection 接口的支持情况

方法名	描述	返回值类型	throws	支持 JDBC 4
abort(Executor executor)	终止打开的连接。	void	SQLException	Yes
clearWarnings()	清除为此Connection对象报告的所有警告。	void	SQLException	Yes
close()	立即释放此Connection对象的数据库和JDBC资源，而不是等待它们自动释放。	void	SQLException	Yes
commit()	使自上次提交/回滚以来所做的所有更改永久化，并释放此Connection对象当前持有的任何数据库锁。仅当禁用了自动提交模式时，才应使用此方法。	void	SQLException	Yes
createArrayOf(String typeName, Object[] elements)	用于创建Array对象的工厂方法。	Array	SQLException	Yes
createBlob()	构造一个实现Blob接口的对象。返回的对象最初不包含数据。Blob接口的setBinaryStream和setBytes方法可以用于向Blob添加数据。	Blob	SQLException	Yes
createClob()	构造一个实现Clob接口的对象。返回的对象最初不包含数据。Clob接口的setAsciiStream、setCharacterStream和setString方法可以用于向Clob添加数据。	Clob	SQLException	Yes
createSQLXML()	构造一个实现SQLXML接口的对象。返回的对象最初不包含数据。可以使用SQLXML接口的createXmlStreamWriter对象和setString方法向SQLXML对象添加数据。	SQLXML	SQLException	Yes
createStatement()	创建一个用于将SQL语句发送到数据库的语句对象。	Statement	SQLException	Yes
createStatement(int resultSetType, int resultSetConcurrency)	创建一个语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。	Statement	SQLException	Yes

方法名	描述	返回值类型	throws	支持 JDBC 4
createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。	Statement	SQLException	Yes
getAutoCommit()	检索此连接对象的当前自动提交模式。	boolean	SQLException	Yes
getCatalog()	检索此Connection对象的当前目录名称。	String	SQLException	Yes
getClientInfo()	返回一个列表，其中包含驱动程序支持的每个客户端信息属性的名称和当前值。如果客户端信息属性尚未设置且没有默认值，则该属性的值可能为NULL。 当前获取到的只有ApplicationName和ApplicationType信息。	Properties	SQLException	Yes
getClientInfo(String name)	返回由name指定的客户端信息属性的值。如果尚未设置指定的客户端信息属性，并且没有默认值，此方法可能会返回NULL。如果驱动程序不支持指定的客户端信息属性名称，此方法也将返回NULL。	String	SQLException	Yes
getHoldability()	检索使用此Connection对象创建的ResultSet对象的当前保持性。	int	SQLException	Yes
getMetaData()	检索包含有关此Connection对象表示连接的数据库的元数据的DatabaseMetaData对象。元数据包括有关数据库表、其支持的SQL语法、其存储过程、此连接的功能等的信息。	DatabaseMetaData	SQLException	Yes
getNetworkTimeout()	检索驱动程序等待数据库请求完成的毫秒数。如果超过限制，将引发SQLException。	int	SQLException	Yes
getSchema()	检索此Connection对象的当前架构名称。	String	SQLException	Yes
getTransactionIsolation()	检索此Connection对象的当前事务隔离级别。	int	SQLException	Yes

方法名	描述	返回值类型	throws	支持 JDBC 4
getTypeMap()	检索与此Connection对象关联的Map对象。除非应用程序添加了条目，否则返回的类型映射将为空。	Map<String,Class<?>>	SQLException	Yes
getWarnings()	检索此Connection对象上的调用报告的第一个警告。如果有多个警告，则后续警告将链接到第一个警告，并可以通过对先前检索的警告调用方法SQLWarning.getNextWarning来检索。	SQLWarning	SQLException	Yes
isClosed()	检索此Connection对象是否已关闭。如果已对连接调用了方法关闭，或者发生了某些重大错误，则连接将关闭。只有在调用Connection.Close方法之后调用此方法时，才保证返回true。	boolean	SQLException	Yes
isReadOnly()	检索此Connection对象是否处于只读模式。	boolean	SQLException	Yes
isValid(int timeout)	如果连接尚未关闭并且仍然有效，则返回true。驱动程序应提交对连接的查询，或使用其他机制，以肯定地验证连接在调用此方法时仍然有效。	boolean	SQLException	Yes
nativeSQL(String sql)	驱动程序可将JDBC SQL语句转换为系统的原生SQL语法再发送。此方法返回该驱动程序转换后的原生SQL语句。	String	SQLException	Yes
prepareCall(String sql)	创建用于调用数据库存储过程的CallableStatement对象。CallableStatement对象提供了用于设置其IN和OUT参数的方法，以及用于执行对存储过程调用的方法。	CallableStatement	SQLException	Yes
prepareCall(String sql, int resultSetType, int resultSetConcurrency)	创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的prepareCall方法相同，但它允许覆盖默认的结果集类型和并发。	CallableStatement	SQLException	Yes

方法名	描述	返回值类型	throws	支持 JDBC 4
prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的prepareCall方法相同，但它允许覆盖默认的结果集类型、结果集并发类型和保持性。	CallableStatement	SQLException	Yes
prepareStatement(String sql)	创建一个用于将参数化SQL语句发送到数据库的准备语句对象。	PreparedStatement	SQLException	Yes
prepareStatement(String sql, int autoGeneratedKeys)	创建一个默认的准备语句对象，该对象能够检索自动生成的键。给定的常量反馈驱动程序，使自动生成的键是否可供检索。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，则将忽略此参数。	PreparedStatement	SQLException	Yes
prepareStatement(String sql, int[] columnIndexes)	创建一个默认的准备语句对象，该对象能够返回由给定数组指定的自动生成的键。此数组包含目标表中包含应可用的自动生成键的列的索引。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，驱动程序将忽略数组。	PreparedStatement	SQLException	Yes
prepareStatement(String sql, int resultSetType, int resultSetConcurrency)	创建一个准备语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的准备语句方法相同，但它允许覆盖默认的结果集类型和并发。	PreparedStatement	SQLException	Yes
prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个准备语句对象，该对象将生成具有给定类型、并发性和保持性的ResultSet对象。	PreparedStatement	SQLException	Yes

方法名	描述	返回值类型	throws	支持 JDBC 4
prepareStatement(String sql, String[] columnNames)	创建一个默认的准备语句对象，该对象能够返回由给定数组指定的自动生成的键。此数组包含目标表中包含应返回的自动生成键的列的名称。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，驱动程序将忽略数组。	PreparedStatement	SQLException	Yes
releaseSavepoint(Savepoint savepoint)	从当前事务中删除指定的Savepoint和后续Savepoint对象。删除保存点后对保存点的任何引用都将引发SQLException。	void	SQLException	Yes
rollback()	撤销在当前事务中所做的所有更改，并释放此Connection对象当前持有的任何数据库锁。仅当禁用自动提交模式时，才应使用此方法。	void	SQLException	Yes
rollback(Savepoint savepoint)	撤销设置给定Savepoint对象后所做的所有更改。仅当禁用自动提交模式时，才应使用此方法。	void	SQLException	Yes
setAutoCommit(boolean autoCommit)	将此连接的自动提交模式设置为给定状态。如果连接处于自动提交模式，则其所有SQL语句都将作为单个事务执行和提交。否则，其SQL语句将分组为事务，这些事务由对方法提交或方法回滚的调用终止。默认情况下，新连接处于自动提交模式。	void	SQLException	Yes
setClientInfo(Properties properties)	设置连接的客户端信息属性的值。Properties对象包含要设置的客户端信息属性的名称和值。属性列表中包含的客户端信息属性集替换连接上的当前客户端信息属性集。如果当前在连接上设置的属性不在属性列表中，则将清除该属性。 当前只会将client中的ApplicationName和ApplicationType信息进行替换。	void	SQLClientInfoException	Yes
setClientInfo(String name, String value)	将name指定的客户端info属性的值设置为value指定的值。 当前只会将client中的ApplicationName和ApplicationType信息进行替换。	void	SQLClientInfoException	Yes

方法名	描述	返回值类型	throws	支持 JDBC 4
setHoldability(int holdability)	将使用此Connection对象创建的ResultSet对象的默认保持性更改为给定的保持性。ResultSet对象的默认保持性可以通过调用DatabaseMetaData.getResultSetHoldability来确定。	void	SQLException	Yes
setNetworkTimeout(Executor executor, int milliseconds)	设置连接或从连接创建的对象等待数据库回复任何一个请求的最长时间。如果任何请求都未应答，则等待方法将返回SQLException，并且从Connection创建的Connection或对象将标记为已关闭。除关闭、isCabled或Connection.isValid方法外，对对象的任何后续使用都将引发SQLException。	void	SQLException	Yes
setReadOnly(boolean readOnly)	将此连接置于只读模式，作为驱动程序提示，以启用数据库优化。	void	SQLException	Yes
setSavepoint()	在当前事务中创建一个未命名的保存点，并返回表示它的新保存点对象。	Savepoint	SQLException	Yes
setSavepoint(String name)	在当前事务中使用给定名称创建保存点，并返回表示它的新保存点对象。	Savepoint	SQLException	Yes
setSchema(String schema)	将给定的架构名称设置为访问。	void	SQLException	Yes
setTransactionIsolation(int level)	尝试将此Connection对象的事务隔离级别更改为给定的级别。接口Connection中定义的常量是可能的事务隔离级别。	void	SQLException	Yes
setTypeMap(Map<String,Class<?>> map)	安装给定的TypeMap对象作为此Connection对象的类型映射。类型映射将用于SQL结构化类型和不同类型的自定义映射。	void	SQLException	Yes

须知

- 接口内部默认使用自动提交模式，若通过setAutoCommit(false)关闭自动提交模式，将会导致后面执行的语句受到显式事务包裹，数据库中不支持事务中执行的语句不能在此模式下执行。
- 创建PreparedStatement类时，在SQL语句中使用RETURNING语句，RETURNING子句不生效。

5.3.4.3 java.sql.CallableStatement

java.sql.CallableStatement是存储过程执行接口。

表 5-29 对 java.sql.CallableStatement 的支持情况

方法名	返回值类型	支持JDBC4
getArray(int parameterIndex)	Array	Yes
getBigDecimal(int parameterIndex)	BigDecimal	Yes
getBlob(int parameterIndex)	Blob	Yes
getBoolean(int parameterIndex)	boolean	Yes
getByte(int parameterIndex)	byte	Yes
getBytes(int parameterIndex)	byte[]	Yes
getClob(int parameterIndex)	Clob	Yes
getDate(int parameterIndex)	Date	Yes
getDate(int parameterIndex, Calendar cal)	Date	Yes
getDouble(int parameterIndex)	double	Yes
getFloat(int parameterIndex)	float	Yes
getInt(int parameterIndex)	int	Yes
getLong(int parameterIndex)	long	Yes
getObject(int parameterIndex)	Object	Yes
getObject(int parameterIndex, Class<T> type)	Object	Yes
getRowId(int parameterIndex)	RowId	Yes

方法名	返回值类型	支持JDBC4
getShort(int parameterIndex)	short	Yes
getSQLXML(int parameterIndex)	SQLXML	Yes
getString(int parameterIndex)	String	Yes
getNString(int parameterIndex)	String	Yes
getTime(int parameterIndex)	Time	Yes
getTime(int parameterIndex, Calendar cal)	Time	Yes
getTimestamp(int parameterIndex)	Timestamp	Yes
getTimestamp(int parameterIndex, Calendar cal)	Timestamp	Yes
registerOutParameter(int parameterIndex, int type)	void	Yes
registerOutParameter(int parameterIndex, int sqlType, int type)	void	Yes
wasNull()	boolean	Yes

说明

- 不允许含有OUT参数的语句执行批量操作。
- 以下方法是从java.sql.Statement继承而来：close、execute、executeQuery、executeUpdate、getConnection、getResultSet、getUpdateCount、isClosed、setMaxRows、setFetchSize。
- 以下方法是从java.sql.PreparedStatement继承而来：addBatch、clearParameters、execute、executeQuery、executeUpdate、getMetaData、setBigDecimal、setBoolean、setByte、setBytes、setDate、setDouble、setFloat、setInt、setLong、setNull、setObject、setString、setTime、setTimestamp。
- registerOutParameter(int parameterIndex, int sqlType, int type)方法仅用于注册复合数据类型，其它类型不支持。

表 5-30 java.sql.CallableStatement 可支持的接口（仅在 ORA 兼容的模式支持，需要设置连接参数 oracleCompatible=functionsCompatible）

方法名	返回值类型	支持JDBC4
setNString(String parameterName, String value)	void	Yes
setClob(String parameterName, Reader reader, long length)	void	Yes
setClob(String parameterName, Reader reader)	void	Yes
setBlob(String parameterName, InputStream inputStream, long length)	void	Yes
setBlob(String parameterName, InputStream inputStream)	void	Yes
setBlob(String parameterName, Blob x)	void	Yes
setClob(String parameterName, Clob x)	void	Yes
setSQLXML(String parameterName, SQLXML xmlObject)	void	Yes
setNull(String parameterName, int sqlType)	void	Yes
setBoolean(String parameterName, boolean x)	void	Yes
setByte(String parameterName, byte x)	void	Yes
setShort(String parameterName, short x)	void	Yes
setInt(String parameterName, int x)	void	Yes
setLong(String parameterName, long x)	void	Yes

方法名	返回值类型	支持JDBC4
setFloat(String parameterName, float x)	void	Yes
setDouble(String parameterName, double x)	void	Yes
setBigDecimal(String parameterName, BigDecimal x)	void	Yes
setString(String parameterName, String x)	void	Yes
setBytes(String parameterName, byte[] x)	void	Yes
setDate(String parameterName, java.sql.Date x)	void	Yes
setTime(String parameterName, Time x)	void	Yes
setTimestamp(String parameterName, Timestamp x)	void	Yes
setObject(String parameterName, Object x, int targetSqlType, int scale)	void	Yes
setObject(String parameterName, Object x, int targetSqlType)	void	Yes
setObject(String parameterName, Object x)	void	Yes
setDate(String parameterName, java.sql.Date x, Calendar cal)	void	Yes
setTime(String parameterName, Time x, Calendar cal)	void	Yes

方法名	返回值类型	支持JDBC4
setTimestamp(String parameterName, Timestamp x, Calendar cal)	void	Yes
setNull(String parameterName, int sqlType, String typeName)	void	Yes
setRowId(String parameterName, RowId x)	void	Yes
getObject(String parameterName)	Object	Yes
getBoolean(String parameterName)	boolean	Yes
getByte(String parameterName)	byte	Yes
getShort(String parameterName)	short	Yes
getInt(String parameterName)	int	Yes
getLong(String parameterName)	long	Yes
getFloat(String parameterName)	float	Yes
getDouble(String parameterName)	double	Yes
getBytes(String parameterName)	byte[]	Yes
getArray(String parameterName)	Array	Yes
getDate(String parameterName)	java.sql.Date	Yes
getDate(String parameterName, Calendar cal)	java.sql.Date	Yes
getString(String parameterName)	String	Yes

方法名	返回值类型	支持JDBC4
getClob(String parameterName)	Clob	Yes
getSQLXML(String parameterName)	java.sql.SQLXML	Yes
getString(String parameterName)	String	Yes
getBigDecimal(String parameterName)	BigDecimal	Yes
getBlob(String parameterName)	Blob	Yes
getTimestamp(String parameterName)	Timestamp	Yes
getTimestamp(String parameterName, Calendar cal)	Timestamp	Yes
getRowId(String parameterName)	RowId	Yes
execute()	boolean	Yes
wasNull()	boolean	Yes
registerOutParameter(String parameterName, int sqlType)	void	Yes
registerOutParameter(String parameterName, int sqlType, int scale)	void	Yes
registerOutParameter(String parameterName, int sqlType, String typeName)	void	Yes

 说明

- 需要设置连接参数oracleCompatible=functionsCompatible。
- 不支持对于出参（包含返回值）有且仅有一个且类型是record的存储过程。
- 不支持注册为Types.NULL、Types.OTHER、Types.JAVA_OBJECT、Types.DISTINCT。
- 仅支持call语法，推荐使用索引的方式。
- CallableStatement的get方法类型转换能力与ResultSet相对应的get方法类型转换能力保持一致。
- registerOutParameter接口支持注册与存过定义参数类型不一致的sqlType。
 - proc_outparam_override参数未开启时，理论上所有sqlType都能注册（除了不支持的sqlType）；
 - proc_outparam_override参数开启时，注册sqlType会绑定类型oid，数据库内核会判断该类型是否支持隐式转换到定义的类型，若不支持隐式转换会报错。

getObject(Int parameterIndex)和getObject(String parameterName)获取的类型与返回值类型无关，与注册的sqlType有关。注册sqlType对应返回值类型，如表5-31所示。

表 5-31 注册 sqlType 对应返回值类型

sqlType	实际返回值类型
Types.BIT	boolean
Types.BOOLEAN	boolean
Types.TINYINT	byte
Types.SMALLINT	short
Types.INTEGER	int
Types.BIGINT	long
Types.REAL	float
Types.FLOAT	double
Types.DOUBLE	double
Types.DECIMAL	BigDecimal
Types.NUMERIC	BigDecimal
Types.CHAR	String
Types.VARCHAR	String
Types.LONGVARCHAR	String
Types.DATALINK	String
Types.NCHAR	String
Types.NVARCHAR	String
Types.LONGNVARCHAR	String

sqlType	实际返回值类型
Types.NCLOB	NClob(暂不支持)
Types.DATE	Date
Types.TIME	Time
Types.TIME_WITH_TIMEZONE	Time
Types.TIMESTAMP	Timestamp
Types.TIMESTAMP_WITH_TIMEZONE	Timestamp
Types.CLOB	Clob
Types.BLOB	Blob
Types.BINARY	byte[]
Types.VARBINARY	byte[]
Types.LONGVARBINARY	byte[]
Types.ARRAY	Array
Types.STRUCT	Record
Types.REF	Ref(暂不支持)
Types.ROWID	RowId
Types.SQLXML	SQLXML
Types.REF_CURSOR	ResultSet

5.3.4.4 java.sql.DatabaseMetaData

java.sql.DatabaseMetaData是数据库对象定义接口。

表 5-32 对 java.sql.DatabaseMetaData 的支持情况

方法名	返回值类型	支持JDBC4
allProceduresAreCallable()	boolean	Yes
allTablesAreSelectable()	boolean	Yes
autoCommitFailureClosesAllResultSets()	boolean	Yes
dataDefinitionCausesTransactionCommit()	boolean	Yes
dataDefinitionIgnoredInTransactions()	boolean	Yes

方法名	返回值类型	支持JDBC4
deletesAreDetected(int type)	boolean	Yes
doesMaxRowSizeIncludeBlobs()	boolean	Yes
generatedKeyAlwaysReturned()	boolean	Yes
getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)	ResultSet	Yes
getCatalogs()	ResultSet	Yes
getCatalogSeparator()	String	Yes
getCatalogTerm()	String	Yes
getClientInfoProperties()	ResultSet	Yes
getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)	ResultSet	Yes
getConnection()	Connection	Yes
getCrossReference(String parentCatalog, String parentSchema, String parentTable, String foreignCatalog, String foreignSchema, String foreignTable)	ResultSet	Yes
getDefaultTransactionIsolation()	int	Yes
getExportedKeys(String catalog, String schema, String table)	ResultSet	Yes
getExtraNameCharacters()	String	Yes
getFunctionColumns(String catalog, String schemaPattern, String functionNamePattern, String columnNamePattern)	ResultSet	Yes
getFunctions(String catalog, String schemaPattern, String functionNamePattern)	ResultSet	Yes

方法名	返回值类型	支持JDBC4
getIdentifierQuoteString()	String	Yes
getImportedKeys(String catalog, String schema, String table)	ResultSet	Yes
getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate)	ResultSet	Yes
getMaxBinaryLiteralLength()	int	Yes
getMaxCatalogNameLength()	int	Yes
getMaxCharLiteralLength()	int	Yes
getMaxColumnNameLength()	int	Yes
getMaxColumnsInGroupBy()	int	Yes
getMaxColumnsInIndex()	int	Yes
getMaxColumnsInOrderBy()	int	Yes
getMaxColumnsInSelect()	int	Yes
getMaxColumnsInTable()	int	Yes
getMaxConnections()	int	Yes
getMaxCursorNameLength()	int	Yes
getMaxIndexLength()	int	Yes
getMaxLogicalLobSize()	default long	Yes
getMaxProcedureNameLength()	int	Yes
getMaxRowSize()	int	Yes
getMaxSchemaNameLength()	int	Yes
getMaxStatementLength()	int	Yes
getMaxStatements()	int	Yes
getMaxTableNameLength()	int	Yes
getMaxTablesInSelect()	int	Yes
getMaxUserNameLength()	int	Yes
getNumericFunctions()	String	Yes

方法名	返回值类型	支持JDBC4
getPrimaryKeys(String catalog, String schema, String table)	ResultSet	Yes
getPartitionTablePrimaryKeys(String catalog, String schema, String table)	ResultSet	Yes
getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)	ResultSet	Yes
getProcedures(String catalog, String schemaPattern, String procedureNamePattern)	ResultSet	Yes
getProcedureTerm()	String	Yes
getSchemas()	ResultSet	Yes
getSchemas(String catalog, String schemaPattern)	ResultSet	Yes
getSchemaTerm()	String	Yes
getSearchStringEscape()	String	Yes
getSQLKeywords()	String	Yes
getSQLStateType()	int	Yes
getStringFunctions()	String	Yes
getSystemFunctions()	String	Yes
getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)	ResultSet	Yes
getTimeDateFunctions()	String	Yes
getTypeInfo()	ResultSet	Yes
getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)	ResultSet	Yes
getURL()	String	Yes

方法名	返回值类型	支持JDBC4
getVersionColumns(String catalog, String schema, String table)	ResultSet	Yes
insertsAreDetected(int type)	boolean	Yes
locatorsUpdateCopy()	boolean	Yes
othersDeletesAreVisible(int type)	boolean	Yes
othersInsertsAreVisible(int type)	boolean	Yes
othersUpdatesAreVisible(int type)	boolean	Yes
ownDeletesAreVisible(int type)	boolean	Yes
ownInsertsAreVisible(int type)	boolean	Yes
ownUpdatesAreVisible(int type)	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
storesMixedCaseIdentifiers()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
supportsBatchUpdates()	boolean	Yes
supportsCatalogsInDataManipulation()	boolean	Yes
supportsCatalogsInIndexDefinitions()	boolean	Yes
supportsCatalogsInPrivilegeDefinitions()	boolean	Yes
supportsCatalogsInProcedureCalls()	boolean	Yes
supportsCatalogsInTableDefinitions()	boolean	Yes
supportsCorrelatedSubqueries()	boolean	Yes
supportsDataDefinitionAndDataManipulationTransactions()	boolean	Yes

方法名	返回值类型	支持JDBC4
supportsDataManipulationTransactionsOnly()	boolean	Yes
supportsGetGeneratedKeys()	boolean	Yes
supportsMixedCaseIdentifiers()	boolean	Yes
supportsMultipleOpenResults()	boolean	Yes
supportsNamedParameters()	boolean	Yes
supportsOpenCursorsAcrossCommit()	boolean	Yes
supportsOpenCursorsAcrossRollback()	boolean	Yes
supportsOpenStatementsAcrossCommit()	boolean	Yes
supportsOpenStatementsAcrossRollback()	boolean	Yes
supportsPositionedDelete()	boolean	Yes
supportsPositionedUpdate()	boolean	Yes
supportsRefCursors()	boolean	Yes
supportsResultSetConcurrency(int type, int concurrency)	boolean	Yes
supportsResultSetType(int type)	boolean	Yes
supportsSchemasInIndexDefinitions()	boolean	Yes
supportsSchemasInPrivilegeDefinitions()	boolean	Yes
supportsSchemasInProcedureCalls()	boolean	Yes
supportsSchemasInTableDefinitions()	boolean	Yes
supportsSelectForUpdate()	boolean	Yes
supportsStatementPooling()	boolean	Yes
supportsStoredFunctionsUsingCallSyntax()	boolean	Yes
supportsStoredProcedures()	boolean	Yes

方法名	返回值类型	支持JDBC4
supportsSubqueriesInComparisons()	boolean	Yes
supportsSubqueriesInExists()	boolean	Yes
supportsSubqueriesInIns()	boolean	Yes
supportsSubqueriesInQuantifieds()	boolean	Yes
supportsTransactionIsolationLevel(int level)	boolean	Yes
supportsTransactions()	boolean	Yes
supportsUnion()	boolean	Yes
supportsUnionAll()	boolean	Yes
updatesAreDetected(int type)	boolean	Yes
getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)	ResultSet	Yes
getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)	ResultSet	Yes
getTableTypes()	ResultSet	Yes
getUserName()	String	Yes
isReadOnly()	boolean	Yes
nullsAreSortedHigh()	boolean	Yes
nullsAreSortedLow()	boolean	Yes
nullsAreSortedAtStart()	boolean	Yes
nullsAreSortedAtEnd()	boolean	Yes
getDatabaseProductName()	String	Yes
getDatabaseProductVersion()	String	Yes
getDriverName()	String	Yes
getDriverVersion()	String	Yes
getDriverMajorVersion()	int	Yes

方法名	返回值类型	支持JDBC4
getDriverMinorVersion()	int	Yes
usesLocalFiles()	boolean	Yes
usesLocalFilePerTable()	boolean	Yes
supportsMixedCaseIdentifiers()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
supportsMixedCaseQuotedIdentifiers()	boolean	Yes
storesUpperCaseQuotedIdentifiers()	boolean	Yes
storesLowerCaseQuotedIdentifiers()	boolean	Yes
storesMixedCaseQuotedIdentifiers()	boolean	Yes
supportsAlterTableWithAddColumn()	boolean	Yes
supportsAlterTableWithDropColumn()	boolean	Yes
supportsColumnAliasing()	boolean	Yes
nullPlusNonNullIsNull()	boolean	Yes
supportsConvert()	boolean	Yes
supportsConvert(int fromType, int toType)	boolean	Yes
supportsTableCorrelationNames()	boolean	Yes
supportsDifferentTableCorrelationNames()	boolean	Yes
supportsExpressionsInOrderBy()	boolean	Yes
supportsOrderByUnrelated()	boolean	Yes
supportsGroupBy()	boolean	Yes
supportsGroupByUnrelated()	boolean	Yes
supportsGroupByBeyondSelect()	boolean	Yes

方法名	返回值类型	支持JDBC4
supportsLikeEscapeClause()	boolean	Yes
supportsMultipleResultSets()	boolean	Yes
supportsMultipleTransactions()	boolean	Yes
supportsNonNullableColumns()	boolean	Yes
supportsMinimumSQLGrammar()	boolean	Yes
supportsCoreSQLGrammar()	boolean	Yes
supportsExtendedSQLGrammar()	boolean	Yes
supportsANSI92EntryLevelSQL()	boolean	Yes
supportsANSI92IntermediateSQL()	boolean	Yes
supportsANSI92FullSQL()	boolean	Yes
supportsIntegrityEnhancementFacility()	boolean	Yes
supportsOuterJoins()	boolean	Yes
supportsFullOuterJoins()	boolean	Yes
supportsLimitedOuterJoins()	boolean	Yes
isCatalogAtStart()	boolean	Yes
supportsSchemasInDataManipulation()	boolean	Yes
supportsSavepoints()	boolean	Yes
supportsResultSetHoldability(int holdability)	boolean	Yes
getResultSetHoldability()	int	Yes
getDatabaseMajorVersion()	int	Yes
getDatabaseMinorVersion()	int	Yes
getJDBCMinorVersion()	int	Yes
getJDBCMajorVersion()	int	Yes
getJDBCMinorVersion()	int	Yes

 说明

- uppercaseAttributeName为true时，以下方法会将查询结果转为大写，可转换范围与java中的toUpperCase方法一致。
 - public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern)
 - public ResultSet getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)
 - public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)
 - public ResultSet getSchemas(String catalog, String schemaPattern)
 - public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)
 - public ResultSet getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)
 - public ResultSet getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)
 - public ResultSet getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)
 - public ResultSet getPrimaryKeys(String catalog, String schema, String table)
 - protected ResultSet getImportedExportedKeys(String primaryCatalog, String primarySchema, String primaryTable, String foreignCatalog, String foreignSchema, String foreignTable)
 - public ResultSet getIndexInfo(String catalog, String schema, String tableName, boolean unique, boolean approximate)
 - public ResultSet getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)
 - public ResultSet getFunctions(String catalog, String schemaPattern, String functionNamePattern)
- getPartitionTablePrimaryKeys(String catalog, String schema, String table)方法用于获取分区表含全局索引的主键列，使用示例如下：

```
PgDatabaseMetaData dbmd = (PgDatabaseMetaData)conn.getMetaData();
dbmd.getPartitionTablePrimaryKeys("catalogName", "schemaName", "tableName");
```

5.3.4.5 java.sql.Driver

java.sql.Driver是数据库驱动接口。

表 5-33 对 java.sql.Driver 的支持情况

方法名	返回值类型	支持JDBC4
acceptsURL(String url)	boolean	Yes
connect(String url, Properties info)	Connection	Yes
jdbcCompliant()	boolean	Yes
getMajorVersion()	int	Yes
getMinorVersion()	int	Yes
getParentLogger()	Logger	Yes

方法名	返回值类型	支持JDBC4
getPropertyInfo(String url, Properties info)	DriverPropertyInfo[]	Yes

5.3.4.6 java.sql.PreparedStatement

java.sql.PreparedStatement是预处理语句接口。

表 5-34 对 java.sql.PreparedStatement 的支持情况

方法名	返回值类型	支持JDBC4
clearParameters()	void	Yes
execute()	boolean	Yes
executeQuery()	ResultSet	Yes
executeUpdate()	int	Yes
executeLargeUpdate()	long	No
getMetaData()	ResultSetMetaData	Yes
getParameterMetaData()	ParameterMetaData	Yes
setArray(int parameterIndex, Array x)	void	Yes
setAsciiStream(int parameterIndex, InputStream x, int length)	void	Yes
setAsciiStream(int parameterIndex, InputStream x, long length)	void	Yes
setAsciiStream(int parameterIndex, InputStream x)	void	Yes
setBinaryStream(int parameterIndex, InputStream x)	void	Yes
setBinaryStream(int parameterIndex, InputStream x, int length)	void	Yes

方法名	返回值类型	支持JDBC4
setBinaryStream(int parameterIndex, InputStream x, long length)	void	Yes
setBlob(int parameterIndex, InputStream inputStream)	void	Yes
setBlob(int parameterIndex, InputStream inputStream, long length)	void	Yes
setBlob(int parameterIndex, Blob x)	void	Yes
setCharacterStream(int parameterIndex, Reader reader)	void	Yes
setCharacterStream(int parameterIndex, Reader reader, int length)	void	Yes
setCharacterStream(int parameterIndex, Reader value, long length)	void	Yes
setClob(int parameterIndex, Reader reader)	void	Yes
setClob(int parameterIndex, Reader reader, long length)	void	Yes
setClob(int parameterIndex, Clob x)	void	Yes
setDate(int parameterIndex, Date x, Calendar cal)	void	Yes
setNull(int parameterIndex, int sqlType)	void	Yes
setNull(int parameterIndex, int sqlType, String typeName)	void	Yes

方法名	返回值类型	支持JDBC4
setObject(int parameterIndex, Object x)	void	Yes
setObject(int parameterIndex, Object x, int targetSqlType)	void	Yes
setObject(int parameterIndex, Object x, int targetSqlType, int scaleOrLength)	void	Yes
setObject(int parameterIndex, Object x, SQLType targetSqlType, int scaleOrLength)	void	Yes
setObject(int parameterIndex, Object x, SQLType targetSqlType)	void	Yes
setRowId(int parameterIndex, RowId x)	void	Yes
setSQLXML(int parameterIndex, SQLXML xmlObject)	void	Yes
setTime(int parameterIndex, Time x)	void	Yes
setTime(int parameterIndex, Time x, Calendar cal)	void	Yes
setTimestamp(int parameterIndex, Timestamp x)	void	Yes
setTimestamp(int parameterIndex, Timestamp x, Calendar cal)	void	Yes
setUnicodeStream(int parameterIndex, InputStream x, int length)	void	Yes

方法名	返回值类型	支持JDBC4
setURL(int parameterIndex, URL x)	void	Yes
setBoolean(int parameterIndex, boolean x)	void	Yes
setBigDecimal(int parameterIndex, BigDecimal x)	void	Yes
setByte(int parameterIndex, byte x)	void	Yes
setBytes(int parameterIndex, byte[] x)	void	Yes
setDate(int parameterIndex, Date x)	void	Yes
setDouble(int parameterIndex, double x)	void	Yes
setFloat(int parameterIndex, float x)	void	Yes
setInt(int parameterIndex, int x)	void	Yes
setLong(int parameterIndex, long x)	void	Yes
setShort(int parameterIndex, short x)	void	Yes
setString(int parameterIndex, String x)	void	Yes
setNString(int parameterIndex, String x)	void	Yes
addBatch()	void	Yes
executeBatch()	int[]	Yes

 说明

- addBatch()、execute()必须在clearBatch()之后才能执行。
- 调用executeBatch()方法并不会清除batch。用户必须显式使用clearBatch()清除。
- 在添加了一个batch的绑定变量后，用户若想重用这些值(再次添加一个batch)，无需再次使用set*()方法。
- 以下方法是从java.sql.Statement继承而来：close、execute、executeQuery、executeUpdate、getConnection、getResultSet、getUpdateCount、isClosed、setMaxRows、setFetchSize、enableStreamingResults。
- executeLargeUpdate()方法必须在JDBC4.2及以上版本使用。
- 执行SQL语句时报文有最大长度限制（不能超过1023MB），因此，使用setAsciiStream和setCharacterStream等方法时需评估和控制数据大小，避免出现因报文过长出现异常。
- executeBatch()方法在批量执行插入时，若任意一条数据发生异常，则整个Batch插入操作将全部回滚，不支持部分提交。

5.3.4.7 java.sql.ResultSet

java.sql.ResultSet是执行结果集接口。

表 5-35 对 java.sql.ResultSet 的支持情况

方法名	返回值类型	支持JDBC4
absolute(int row)	boolean	Yes
afterLast()	void	Yes
beforeFirst()	void	Yes
cancelRowUpdates()	void	Yes
clearWarnings()	void	Yes
close()	void	Yes
deleteRow()	void	Yes
findColumn(String columnLabel)	int	Yes
first()	boolean	Yes
getArray(int columnIndex)	Array	Yes
getArray(String columnLabel)	Array	Yes
getAsciiStream(int columnIndex)	InputStream	Yes
getAsciiStream(String columnLabel)	InputStream	Yes
getBigDecimal(int columnIndex)	BigDecimal	Yes

方法名	返回值类型	支持JDBC4
getBigDecimal(String columnLabel)	BigDecimal	Yes
getBinaryStream(int columnIndex)	InputStream	Yes
getBinaryStream(String columnLabel)	InputStream	Yes
getBlob(int columnIndex)	Blob	Yes
getBlob(String columnLabel)	Blob	Yes
getBoolean(int columnIndex)	boolean	Yes
getBoolean(String columnLabel)	boolean	Yes
getBytes(int columnIndex)	byte[]	Yes
getBytes(String columnLabel)	byte[]	Yes
getCharacterStream(int columnIndex)	Reader	Yes
getCharacterStream(String columnLabel)	Reader	Yes
getClob(int columnIndex)	Clob	Yes
getClob(String columnLabel)	Clob	Yes
getConcurrency()	int	Yes
getCursorName()	String	Yes
getDate(int columnIndex)	Date	Yes
getDate(int columnIndex, Calendar cal)	Date	Yes
getDate(String columnLabel)	Date	Yes
getDate(String columnLabel, Calendar cal)	Date	Yes

方法名	返回值类型	支持JDBC4
getDouble(int columnIndex)	double	Yes
getDouble(String columnLabel)	double	Yes
getFetchDirection()	int	Yes
getFetchSize()	int	Yes
getFloat(int columnIndex)	float	Yes
getFloat(String columnLabel)	float	Yes
getInt(int columnIndex)	int	Yes
getInt(String columnLabel)	int	Yes
getLong(int columnIndex)	long	Yes
getLong(String columnLabel)	long	Yes
getMetaData()	ResultSetMetaData	Yes
getObject(int columnIndex)	Object	Yes
getObject(int columnIndex, Class<T> type)	<T> T	Yes
getObject(int columnIndex, Map<String,Class<?>> map)	Object	Yes
getObject(String columnLabel)	Object	Yes
getObject(String columnLabel, Class<T> type)	<T> T	Yes
getObject(String columnLabel, Map<String,Class<?>> map)	Object	Yes
getRow()	int	Yes
getRowId(int columnIndex)	RowId	Yes

方法名	返回值类型	支持JDBC4
getRowId(String columnLabel)	RowId	Yes
getShort(int columnIndex)	short	Yes
getShort(String columnLabel)	short	Yes
getSQLXML(int columnIndex)	SQLXML	Yes
getSQLXML(String columnLabel)	SQLXML	Yes
getStatement()	Statement	Yes
getString(int columnIndex)	String	Yes
getString(String columnLabel)	String	Yes
getNString(int columnIndex)	String	Yes
getNString(String columnLabel)	String	Yes
getTime(int columnIndex)	Time	Yes
getTime(int columnIndex, Calendar cal)	Time	Yes
getTime(String columnLabel)	Time	Yes
getTime(String columnLabel, Calendar cal)	Time	Yes
getTimestamp(int columnIndex)	Timestamp	Yes
getTimestamp(int columnIndex, Calendar cal)	Timestamp	Yes
getTimestamp(String columnLabel)	Timestamp	Yes
getTimestamp(String columnLabel, Calendar cal)	Timestamp	Yes
getType()	int	Yes

方法名	返回值类型	支持JDBC4
getWarnings()	SQLWarning	Yes
insertRow()	void	Yes
isAfterLast()	boolean	Yes
isBeforeFirst()	boolean	Yes
isClosed()	boolean	Yes
isFirst()	boolean	Yes
isLast()	boolean	Yes
last()	boolean	Yes
moveToCurrentRow()	void	Yes
moveToInsertRow()	void	Yes
next()	boolean	Yes
previous()	boolean	Yes
refreshRow()	void	Yes
relative(int rows)	boolean	Yes
rowDeleted()	boolean	Yes
rowInserted()	boolean	Yes
rowUpdated()	boolean	Yes
setFetchDirection(int direction)	void	Yes
setFetchSize(int rows)	void	Yes
updateArray(int columnIndex, Array x)	void	Yes
updateArray(String columnLabel, Array x)	void	Yes
updateAsciiStream(int columnIndex, InputStream x)	void	Yes
updateAsciiStream(String columnLabel, InputStream x)	void	Yes
updateAsciiStream(int columnIndex, InputStream x, int length)	void	Yes

方法名	返回值类型	支持JDBC4
updateAsciiStream(String columnLabel, InputStream x, int length)	void	Yes
updateAsciiStream(int columnIndex, InputStream x)	void	Yes
updateAsciiStream(String columnLabel, InputStream x)	void	Yes
updateBigDecimal(int columnIndex, BigDecimal x)	void	Yes
updateBigDecimal(String columnLabel, BigDecimal x)	void	Yes
updateBinaryStream(int columnIndex, InputStream x)	void	Yes
updateBinaryStream(String columnLabel, InputStream x)	void	Yes
updateBinaryStream(int columnIndex, InputStream x, int length)	void	Yes
updateBinaryStream(String columnLabel, InputStream x, int length)	void	Yes
updateBinaryStream(int columnIndex, InputStream x, long length)	void	Yes
updateBinaryStream(String columnLabel, InputStream x, long length)	void	Yes
updateBlob(int columnIndex, Blob x)	void	Yes
updateBlob(String columnLabel, Blob x)	void	Yes

方法名	返回值类型	支持JDBC4
updateBlob(int columnIndex, InputStream x)	void	Yes
updateBlob(String columnLabel, InputStream x)	void	Yes
updateBlob(int columnIndex, InputStream x, long length)	void	Yes
updateBlob(String columnLabel, InputStream x, long length)	void	Yes
updateBoolean(int columnIndex, boolean x)	void	Yes
updateBoolean(String columnLabel, boolean x)	void	Yes
updateByte(int columnIndex, byte x)	void	Yes
updateByte(String columnLabel, byte x)	void	Yes
updateBytes(int columnIndex, byte[] x)	void	Yes
updateBytes(String columnLabel, byte[] x)	void	Yes
updateCharacterStream(int columnIndex, Reader x)	void	Yes
updateCharacterStream(String columnLabel, Reader x)	void	Yes
updateCharacterStream(int columnIndex, Reader x, int length)	void	Yes
updateCharacterStream(String columnLabel, Reader reader, int length)	void	Yes
updateCharacterStream(int columnIndex, Reader x, long length)	void	Yes

方法名	返回值类型	支持JDBC4
updateCharacterStream(String columnLabel, Reader x, long length)	void	Yes
updateClob(int columnIndex, Clob x)	void	Yes
updateClob(String columnLabel, Clob x)	void	Yes
updateClob(int columnIndex, Reader x)	void	Yes
updateClob(String columnLabel, Reader x)	void	Yes
updateClob(int columnIndex, Reader x, long length)	void	Yes
updateClob(String columnLabel, Reader x, long length)	void	Yes
updateDate(int columnIndex, Date x)	void	Yes
updateDate(String columnLabel, Date x)	void	Yes
updateDouble(int columnIndex, double x)	void	Yes
updateDouble(String columnLabel, double x)	void	Yes
updateFloat(int columnIndex, float x)	void	Yes
updateFloat(String columnLabel, float x)	void	Yes
updateInt(int columnIndex, int x)	void	Yes
updateInt(String columnLabel, int x)	void	Yes
updateLong(int columnIndex, long x)	void	Yes
updateLong(String columnLabel, long x)	void	Yes
updateNull(int columnIndex)	void	Yes

方法名	返回值类型	支持JDBC4
updateNull(String columnLabel)	void	Yes
updateObject(int columnIndex, Object x)	void	Yes
updateObject(int columnIndex, Object x, int scaleOrLength)	void	Yes
updateObject(String columnLabel, Object x)	void	Yes
updateObject(String columnLabel, Object x, int scaleOrLength)	void	Yes
updateRow()	void	Yes
updateRowId(int columnIndex, RowId x)	void	Yes
updateRowId(String columnLabel, RowId x)	void	Yes
updateShort(int columnIndex, short x)	void	Yes
updateShort(String columnLabel, short x)	void	Yes
updateSQLXML(int columnIndex, SQLXML xmlObject)	void	Yes
updateSQLXML(String columnLabel, SQLXML xmlObject)	void	Yes
updateString(int columnIndex, String x)	void	Yes
updateString(String columnLabel, String x)	void	Yes
updateTime(int columnIndex, Time x)	void	Yes
updateTime(String columnLabel, Time x)	void	Yes
updateTimestamp(int columnIndex, Timestamp x)	void	Yes

方法名	返回值类型	支持JDBC4
updateTimestamp(String columnLabel, Timestamp x)	void	Yes
wasNull()	boolean	Yes

说明

- 一个Statement不能有多个处于“open”状态的ResultSet。
- 用于遍历结果集（ResultSet）的游标（Cursor）在被提交后不能保持“open”的状态。
- 执行SQL语句时，报文有最大长度限制（不能超过1023MB）。因此使用updateAsciiStream、updateBlob、updateBinaryStream、updateClob、updateCharacterStream等方法时需评估和控制数据大小，避免因报文过长出现异常。
- updateRow()相关接口仅支持更新具有主键的表数据。

5.3.4.8 java.sql.ResultSetMetaData

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

表 5-36 对 java.sql.ResultSetMetaData 的支持情况

方法名	返回值类型	支持JDBC4
getCatalogName(int column)	String	Yes
getColumnClassName(int column)	String	Yes
getColumnCount()	int	Yes
getColumnDisplaySize(int column)	int	Yes
getColumnLabel(int column)	String	Yes
getColumnName(int column)	String	Yes
getColumnType(int column)	int	Yes
getColumnTypeName(int column)	String	Yes
getPrecision(int column)	int	Yes
getScale(int column)	int	Yes
getSchemaName(int column)	String	Yes
getTableName(int column)	String	Yes

方法名	返回值类型	支持JDBC4
isAutoIncrement(int column)	boolean	Yes
isCaseSensitive(int column)	boolean	Yes
isCurrency(int column)	boolean	Yes
isDefinitelyWritable(int column)	boolean	Yes
isNullable(int column)	int	Yes
isReadOnly(int column)	boolean	Yes
isSearchable(int column)	boolean	Yes
isSigned(int column)	boolean	Yes
isWritable(int column)	boolean	Yes

说明

uppercaseAttributeName为true时，下面接口会将查询结果转为大写，可转换范围为26个英文字母。

- public String getColumnName(int column)
- public String getColumnLabel(int column)

5.3.4.9 java.sql.Statement

java.sql.Statement是SQL语句接口。

表 5-37 对 java.sql.Statement 的支持情况

方法名	返回值类型	支持JDBC4
addBatch(String sql)	void	Yes
clearBatch()	void	Yes
clearWarnings()	void	Yes
close()	void	Yes
closeOnCompletion()	void	Yes
execute(String sql)	boolean	Yes
execute(String sql, int autoGeneratedKeys)	boolean	Yes
execute(String sql, int[] columnIndexes)	boolean	Yes

方法名	返回值类型	支持JDBC4
execute(String sql, String[] columnNames)	boolean	Yes
executeBatch()	boolean	Yes
executeQuery(String sql)	ResultSet	Yes
executeUpdate(String sql)	int	Yes
executeUpdate(String sql, int autoGeneratedKeys)	int	Yes
executeUpdate(String sql, int[] columnIndexes)	int	Yes
executeUpdate(String sql, String[] columnNames)	int	Yes
getConnection()	Connection	Yes
getFetchDirection()	int	Yes
getFetchSize()	int	Yes
getGeneratedKeys()	ResultSet	Yes
getMaxFieldSize()	int	Yes
getMaxRows()	int	Yes
getLargeMaxRows()	long	Yes
getMoreResults()	boolean	Yes
getMoreResults(int current)	boolean	Yes
getResultSet()	ResultSet	Yes
getResultSetConcurrency()	int	Yes
getResultSetHoldability()	int	Yes
getResultSetType()	int	Yes
getQueryTimeout()	int	Yes
getUpdateCount()	int	Yes
getWarnings()	SQLWarning	Yes

方法名	返回值类型	支持JDBC4
isClosed()	boolean	Yes
isCloseOnCompletion()	boolean	Yes
isPoolable()	boolean	Yes
setCursorName(String name)	void	Yes
setEscapeProcessing(boolean enable)	void	Yes
setFetchDirection(int direction)	void	Yes
setMaxFieldSize(int max)	void	Yes
setMaxRows(int max)	void	Yes
setLargeMaxRows(long max)	void	Yes
setPoolable(boolean poolable)	void	Yes
setQueryTimeout(int seconds)	void	Yes
setFetchSize(int rows)	void	Yes
cancel()	void	Yes
executeLargeUpdate(String sql)	long	No
getLargeUpdateCount()	long	No
executeLargeBatch()	long	No
executeLargeUpdate(String sql, int autoGeneratedKeys)	long	No
executeLargeUpdate(String sql, int[] columnIndexes)	long	No
executeLargeUpdate(String sql, String[] columnNames)	long	No
enableStreamingResults()	void	Yes

 说明

- 通过setFetchSize可以减少结果集在客户端的内存占用情况。它的原理是通过将结果集打包成游标，然后分段处理，所以会加大数据库与客户端的通信量，但会有性能损耗。
- 由于数据库游标是事务内有效，所以，在设置setFetchSize的同时，需要将连接设置为非自动提交模式，setAutoCommit(false)。同时在业务数据需要持久化到数据库中时，在连接上执行提交操作。
- LargeUpdate相关方法必须在JDBC4.2及以上版本使用。
- enableStreamingResults()是开启流式读的一个自定义接口，该接口间接调用setFetchSize(Integer.MIN_VALUE)。若要开启流式读功能，需要将url中的enableStreamingQuery设置为true，同时调用setFetchSize(Integer.MIN_VALUE)或者enableStreamingResults()。除开启流式读功能外，setFetchSize()的入参只能为正数或者0。
- 打开M兼容开关，即设置连接参数mCompatible为true时支持executeLargeUpdate方法，M兼容开关请参见[连接参数参考](#)。
- setLargeMaxRows方法暂不支持设置超过Integer.MAX_VALUE长度的参数。

5.3.4.10 java.sql.Clob

java.sql.Clob接口提供用于获取SQL字符大对象，访问SQL CLOB值。

表 5-38 java.sql.Clob 常用方法

方法名	返回值	描述	throws
length()	long	返回Clob字符数。	SQLException
getSubString(long pos, int length)	String	检索返回从pos开始最多包含length的连续字符串。	SQLException
getCharacterStream()	java.io. Reader	返回包含Clob值的字符 串流。	SQLException
getAsciiStream()	java.io. InputSt ream	返回包含Clob值的ASCII 流。 须知 目前不支持该方法。	SQLException
position(String searchstr, long start)	long	检索指定的字符串，搜索从start位置开始，返回检索的字符串出现的第一个位置。	SQLException
position(Clob searchstr, long start)	long	检索指定的Clob，搜索从start位置开始，返回检索的Clob出现的第一个位置。	SQLException
setString(long pos, String str)	int	在指定的pos位置开始，写入str到Clob中。	SQLException

方法名	返回值	描述	throws
setString(long pos, String str, int offset, int len)	int	在指定的pos位置开始，写入偏移offset量的长度为len的str对象到Clob中。	SQLException
setAsciiStream(long pos)	java.io.OutputStream	返回一个从pos位置开始，能写入字符到Clob对象中的ASCII流。 须知 目前不支持该方法。	SQLException
setCharacterStream(long pos)	java.io.Writer	返回一个从pos位置开始，能写入字符到Clob对象中的字符流。 须知 目前不支持该方法。	SQLException
truncate(long len)	void	截断len长度的Clob字符串。	SQLException
free()	void	释放Clob对象，并且释放资源。	SQLException
getCharacterStream(long pos, long length)	Reader	返回一个Reader包含部分Clob值的对象，该值以pos指定字符开头，长度为length。	SQLException

5.3.4.11 java.sql.Array

java.sql.Array接口是JDBC对SQL集合类型的映射，提供了将SQL数据以数组或ResultSet对象的形式传输到客户端的方法。

表 5-39 java.sql.Array 常用方法

方法名	返回值类型	throws	支持JDBC4
getBaseTypeName()	String	SQLException	Yes
getBaseType()	int	SQLException	Yes
getArray()	Object	SQLException	Yes
getArray(java.util.Map<String,Class<?>> map)	Object	SQLException	No
getArray(long index, int count)	Object	SQLException	Yes

方法名	返回值类型	throws	支持JDBC4
getArray(long index, int count, java.util.Map<String,Class<?>> map)	Object	SQLException	No
getResultSet()	ResultSet	SQLException	No
getResultSet(java.util.Map<String,Class<?>> map)	ResultSet	SQLException	No
getResultSet(long index, int count)	ResultSet	SQLException	No
getResultSet (long index, int count, java.util.Map<String,Class<?>> map)	ResultSet	SQLException	No
free()	void	SQLException	No

 说明

- `getArray`和`getArray(long index, int count)`接口获取的数组类型可参考[数据类型映射关系](#)的JAVA变量类型列。
例如：元素类型为`varchar`，`getArray`接口获取的对象类型为`String[]`。
特殊说明：元素类型为`int2/smallint`，元素存取为`Short`类型，`getArray`接口获取的对象类型为`Short[]`。
- `getBaseType`接口获取的`sqlType`可参考[数据类型映射关系](#)的JDBC类型索引列。
- `getBaseTypeName`获取的元素类型名可参考[数据类型映射关系](#)，其中集合类型、数组类型、`record`类型的类型名映射规则如下：
 1. 若为package里定义的类型，类型名通常格式为`schema.package.type`。
若`schema`、`package`、`type`中有任意名称不符合规则，则类型名格式为`"schema"."package"."type"`：
 - 只能包含字母、数字或下划线。
 - 第一个字符不能是数字。
 2. 若为schema里定义的类型，类型名通常格式为`schema.type`。
若`schema`、`type`中有任意名称不符合规则，则类型名格式为`"schema"."type"`：
 - 只能包含字母、数字或下划线。
 - 第一个字符不能是数字。
- 对于`getArray(long index, int count)`接口
 1. 起始元素的`index`为1；
 2. `index`的取值范围为1-2147483647，`index`的输入超范围时报错；
 3. `count`的取值范围为0-2147483647，`count`的输入超范围时报错；
 4. 若`index`大于元素个数，则返回空数组（长度为0的数组）；
 5. 若`count`超出按当前`index`可获取的最大元素的数量，则返回剩余元素组成的数组。

5.3.4.12 java.sql.Struct

`java.sql.Struct`接口是JDBC对SQL结构化类型的映射，`Struct`对象包含其所代表的SQL结构化类型的每个属性值，提供了将SQL数据以数组的形式传输到客户端的方法。

表 5-40 java.sql.Struct 常用方法

方法名	返回值类型	throws	支持情况
<code>getSQLTypeName()</code>	<code>String</code>	<code>SQLException</code>	Yes
<code>getAttributes()</code>	<code>Object[]</code>	<code>SQLException</code>	Yes
<code>getAttributes(java.util.Map<String, Class<?>> map)</code>	<code>Object[]</code>	<code>SQLException</code>	No

 说明

- `getAttribute`接口获取的对象类型为`Object[]`，返回的数组中每个元素的类型可参考[数据类型映射关系](#)的JAVA变量类型列。

例如：对于`record(c1 int, c2 varchar(5))`类型，`getAttribute`返回`Object[]`中第一个元素为`null`或者`Integer`类型数据，第二个元素为`null`或者`String`类型数据。

特殊说明：元素类型为`int2/smallint`，元素存取为`Short`类型。

- `getSQLTypeName`接口返回的类型名可参考[数据类型映射关系](#)，其中集合类型、数组类型、`record`类型的类型名映射规则如下。
 1. 若为`package`里定义的类型，类型名的格式通常为`schema.package.type`，须符合以下命名规则：
 - 只能包含字母、数字、下划线。
 - 第一个字符不能是数字。若`schema`、`package`、`type`中有任意名称不符合命名规则，则需要加引号，即使用`"schema"."package"."type"`格式。
 2. 若为`schema`里定义的类型，类型名的格式通常为`schema.type`，须符合以下命名规则：
 - 只能包含字母、数字、下划线。
 - 第一个字符不能是数字。若`schema`、`type`中有任意名称不符合命名规则，则需要加引号，即使用`"schema"."type"`格式。

5.3.4.13 java.sql.SQLException

`java.sql.SQLException`是JDBC API中用于处理数据库操作错误的核心异常类。当JDBC操作（如执行SQL语句、处理结果集等）发生错误时，JDBC程序会抛出此异常。

表 5-41 对 `java.sql.SQLException` 的支持情况

方法名	描述	返回值类型	支持JDBC4
<code>getMessage()</code>	返回错误的描述信息。	<code>String</code>	Yes
<code>getSQLState()</code>	返回SQL标准状态码。	<code>String</code>	Yes
<code>getErrorCode()</code>	返回特定的错误码。	<code>int</code>	Yes
<code>getCause()</code>	返回引发此异常的根本原因。	<code>Throwable</code>	Yes
<code>getNextException()</code>	获取与此异常关联的下一个 <code>SQLException</code> （用于处理异常链）。	<code>SQLException</code>	Yes
<code>setNextException(SQLException ex)</code>	将异常链接到当前异常的链尾。	<code>void</code>	Yes
<code>printStackTrace()</code>	将当前异常的堆栈信息打印到标准错误流。	<code>void</code>	Yes

5.3.4.14 java.io.IOException

java.io.IOException是Java I/O操作中的核心受检异常类，用于表示输入/输出操作期间发生的错误或中断。

表 5-42 对 java.io.IOException 的支持情况

方法名	描述	返回值类型	支持JDBC4
getMessage()	返回错误的详细描述信息。	String	Yes
getCause()	返回引发此异常的根本原因。	Throwable	Yes
printStackTrace()	将当前异常的堆栈信息打印到标准错误流。	void	Yes
initCause(Throwable cause)	设置导致此异常的原因（通常用于包装底层异常）。	Throwable	Yes
toString()	返回包含异常类名和错误信息的字符串。	String	Yes
getSuppressed()	获取所有被抑制的异常数组。	Throwable[]	Yes

5.3.4.15 javax.sql.ConnectionPoolDataSource

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

表 5-43 对 javax.sql.ConnectionPoolDataSource 的支持情况

方法名	返回值类型	支持JDBC4
getPooledConnection()	PooledConnection	Yes
getPooledConnection(String user,String password)	PooledConnection	Yes

5.3.4.16 javax.sql.DataSource

javax.sql.DataSource是数据源接口。

表 5-44 对 javax.sql.DataSource 接口的支持情况

方法名	返回值类型	支持JDBC4
getConnection()	Connection	Yes
getConnection(String username,String password)	Connection	Yes
getLoginTimeout()	int	Yes
getLogWriter()	PrintWriter	Yes
setLoginTimeout(int seconds)	void	Yes
setLogWriter(PrintWriter out)	void	Yes

5.3.4.17 javax.sql.PooledConnection

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

表 5-45 对 javax.sql.PooledConnection 的支持情况

方法名	返回值类型	支持JDBC4
addConnectionEventListener (ConnectionEventListener listener)	void	Yes
close()	void	Yes
getConnection()	Connection	Yes
removeConnectionEventListener (ConnectionEventListener listener)	void	Yes

5.3.4.18 javax.naming.Context

javax.naming.Context是连接配置的上下文接口。

表 5-46 对 javax.naming.Context 的支持情况

方法名	返回值类型	支持JDBC4
bind(Name name, Object obj)	void	Yes
bind(String name, Object obj)	void	Yes
lookup(Name name)	Object	Yes
lookup(String name)	Object	Yes

方法名	返回值类型	支持JDBC4
rebind(Name name, Object obj)	void	Yes
rebind(String name, Object obj)	void	Yes
rename(Name oldName, Name newName)	void	Yes
rename(String oldName, String newName)	void	Yes
unbind(Name name)	void	Yes
unbind(String name)	void	Yes

5.3.4.19 CopyManager

CopyManager是GaussDB JDBC驱动中提供的一个API接口类，用于批量向数据库导入数据以及从数据库导出数据。

CopyManager 的继承关系

CopyManager类位于com.huawei.gaussdb.jdbc.copy Package中，继承自java.lang.Object类，该类的声明如下：

```
public class CopyManager
extends Object
```

构造方法

```
public CopyManager(BaseConnection connection)
throws SQLException
```

常用方法

表 5-47 CopyManager 常用方法

方法名	返回值类型	描述	throws	支持JDBC4
copyIn(String sql)	CopyIn	-	SQLException	Yes
copyIn(String sql, InputStream from)	long	使用COPY FROM STDIN从InputStream中快速向数据库中的表加载数据。	SQLException,IOException	Yes

方法名	返回值类型	描述	throws	支持JDBC4
copyIn(String sql, InputStream from, int bufferSize)	long	使用COPY FROM STDIN从InputStream中快速向数据库中的表加载指定长度的数据。	SQLException, IOException	Yes
copyIn(String sql, Reader from)	long	使用COPY FROM STDIN从Reader中快速向数据库中的表加载数据。	SQLException, IOException	Yes
copyIn(String sql, Reader from, int bufferSize)	long	使用COPY FROM STDIN从Reader中快速向数据库中的表加载指定长度的数据。	SQLException, IOException	Yes
copyOut(String sql)	CopyOut	-	SQLException	Yes
copyOut(String sql, OutputStream to)	long	将一个COPY TO STDOUT的结果集从数据库发送到OutputStream类中。	SQLException, IOException	Yes
copyOut(String sql, Writer to)	long	将一个COPY TO STDOUT的结果集从数据库发送到Writer类中。	SQLException, IOException	Yes

5.3.4.20 PGReplicationConnection

PGReplicationConnection是GaussDB JDBC驱动中提供的一个API接口类，用于执行逻辑复制相关的功能。

PGReplicationConnection 的继承关系

PGReplicationConnection是逻辑复制的接口，非透明多写场景下的实现类是PGReplicationConnectionImpl，该类位于com.huawei.gaussdb.jdbc.replication Package中，该类的声明如下：

```
public class PGReplicationConnectionImpl implements PGReplicationConnection
```

构造方法

- public PGReplicationConnectionImpl(BaseConnection connection)
- public TenantReplicationConnectionImpl(BaseConnection connection)

常用方法

表 5-48 PGReplicationConnection 常用方法

方法名	返回值类型	描述	throws
createReplicationSlot()	ChainedCreateReplicationSlotBuilder	用于创建逻辑复制槽。 连接CN只能创建集群级（CSN序）逻辑复制槽，并在其他CN和主DN上创建同名复制槽，连接DN只能创建本地（LSN序）逻辑复制槽。若需要创建CN上的LSN序逻辑复制槽，或是DN上的CSN序逻辑复制槽，请参考逻辑复制SQL函数pg_create_logical_replication_slot。	-
dropReplicationSlot(String slotName)	void	用于删除逻辑复制槽。 连接CN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。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，则连接当前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。	SQLException, IOException
replicationStream()	ChainedStreamBuilder	用户开启逻辑复制。	-

5.3.4.21 PGReplicationStream

PGReplicationStream是GaussDB JDBC驱动中提供的一个API接口类，用于操作逻辑复制流。

PGReplicationStream 的继承关系

PGReplicationStream是逻辑复制的接口，非透明多写场景下的实现类是V3PGReplicationStream，该类位于com.huawei.gaussdb.jdbc.core.v3.replication Package中，该类的声明如下：

```
public class V3PGReplicationStream implements PGReplicationStream
```

构造方法

- public V3PGReplicationStream(CopyDual copyDual, LogSequenceNumber startLSN, long updateIntervalMs, ReplicationType replicationType)
- public MultiPrimaryReplicationStream(CopyDual copyDual, LogSequenceNumber startLSN, long updateIntervalMs, ReplicationType replicationType)

常用方法

表 5-49 PGReplicationStream 常用方法

方法名	返回值类型	描述	throws
close()	void	结束逻辑复制，并释放资源。	SQLException
forceUpdateStatus()	void	强制将上次接收、刷新和应用的 LSN 状态发送到后端。	SQLException
getLastAppliedLSN()	LogSequenceNumber	获取上次主机日志回放的LSN。	-
getLastFlushedLSN()	LogSequenceNumber	获取上次主机刷新的LSN，即当前逻辑解码推进的LSN。	-
getLastReceiveLSN()	LogSequenceNumber	获取上次接收的LSN（针对LSN序复制槽）或CSN（针对CSN序复制槽）。	-
isClosed()	boolean	复制流是否关闭。	-
read()	ByteBuffer	从后端读取下一条WAL记录。如果读取不到，该方法阻塞I/O读。	SQLException
readPending()	ByteBuffer	从后端读取下一条WAL记录。如果读取不到，该方法不阻塞I/O读。	SQLException
setAppliedLSN(LogSequenceNumber applied)	void	设置应用的LSN。	-

方法名	返回值类型	描述	throws
setFlushedLSN(LogSequenceNumber flushed)	void	设置刷新的LSN（针对LSN序复制槽）或CSN（针对CSN序复制槽），在下次更新时发送至后端，用于推进服务端LSN（针对LSN序复制槽）或CSN（针对CSN序复制槽）。	-

5.3.4.22 ChainedStreamBuilder

ChainedStreamBuilder是GaussDB JDBC驱动中提供的一个API接口类，用于构建复制流。本接口类仅在非透明多写特性下支持。

ChainedStreamBuilder 的继承关系

ChainedStreamBuilder是逻辑复制的接口，实现类是ReplicationStreamBuilder，该类位于com.huawei.gaussdb.jdbc.replication.fluent Package中，该类的声明如下：

```
public class ReplicationStreamBuilder implements ChainedStreamBuilder
```

构造方法

```
public ReplicationStreamBuilder(final BaseConnection connection)
```

常用方法

表 5-50 ReplicationStreamBuilder 常用方法

方法名	返回值类型	描述	throws
logical()	ChainedLogicalStreamBuilder	创建逻辑复制流。	-
physical()	ChainedPhysicalStreamBuilder	创建物理复制流。	-

5.3.4.23 ChainedCommonStreamBuilder

ChainedCommonStreamBuilder是GaussDB JDBC驱动中提供的一个API接口类，用于为逻辑和物理复制指定通用参数。本接口类仅在非透明多写特性下支持。

ChainedCommonStreamBuilder 的继承关系

ChainedCommonStreamBuilder是逻辑复制的接口，实现抽象类是AbstractCreateSlotBuilder，该类的继承类是LogicalCreateSlotBuilder，位于com.huawei.gaussdb.jdbc.replication.fluent.logical Package中，该类的声明如下：

```
public class LogicalCreateSlotBuilder
    extends AbstractCreateSlotBuilder<ChainedLogicalCreateSlotBuilder>
    implements ChainedLogicalCreateSlotBuilder
```

构造方法

```
public LogicalCreateSlotBuilder(BaseConnection connection)
```

常用方法

表 5-51 LogicalCreateSlotBuilder 常用方法

方法名	返回值类型	描述	throws
withSlotName(String slotName)	T	指定复制槽名。	-
withOutputPlugin(String outputPlugin)	ChainedLogicalCreateSlotBuilder	插件名称，当前支持 mppdb_decoding、sql_decoding、parallel_binary_decoding、parallel_json_decoding、parallel_text_decoding。 mppdb_decoding：一种解码的输出格式，设置后输出内容为JSON格式。输出的结果包含相关数据的属性信息和属性对应的值。sql_decoding、parallel_binary_decoding、parallel_json_decoding、parallel_text_decoding等根据插件格式不同，分别输出sql格式、二进制格式、json格式和text格式。	-
make()	void	在数据库中创建具有指定参数的插槽。	SQLException
self()	ChainedLogicalCreateSlotBuilder	返回 ChainedLogicalCreateSlotBuilder的实现。	-

5.3.4.24 PGobject

PGobject类用于表示非JDBC标准定义的类型。

表 5-52 PGObject 常用方法

方法名	返回值类型	描述	throws	支持 JDBC4
getStruct()	Object[]	获取复合类型子类型名，按创建顺序排序。	-	Yes
getValue()	String	获取复合类型字符串形式值。	-	Yes
getArrayValue()	String[]	获取复合类型数组形式值，以复合数据类型字段顺序排序。	-	Yes
getAttributes()	Object[]	获取复合类型对象数组形式值（如果组成字段类型为Table类型和Array类型则返回为PgArray，如果组成字段为复合类型则返回PGObject，其他类型返回字符串值）。	SQLException	Yes

5.3.4.25 LogSequenceNumber

LogSequenceNumber类是逻辑解码场景下的日志序列号，其内部的LogSequenceValue变量用于唯一确认一条SQL对应的编码后的日志。

非透明多写场景实现的子类为CommonLogSequenceValue。

表 5-53 LogSequenceNumber 常用方法

方法名	返回值类型	描述
ValueOf(long value)	LogSequenceNumber	通过输入的long类型的LSN值获取非透明多写场景下的LogSequenceNumber对象。
ValueOf(String strValue)	LogSequenceNumber	通过输入的字符串获取对应的LogSequenceNumber对象。
asLong()	long	获取日志序列中的LSN值。
asLongArray()	long	以数组的形式返回序列号，非透明多写场景下返回LSN值。
compareTo(LogSequenceNumber anotherLSN)	int	两个LogSequenceNumber内部的LogSequenceValue值作比较，在非透明多写场景下比较LSN的值。当前LogSequenceValue比输入的内部LogSequenceValue大则返回1，相等即返回0，否则返回-1。
equals(Object o)	boolean	非透明多写场景下lsn值相等则返回true。输入null、输入类不是LogSequenceNumber、或者类内部的LogSequenceValue不相同都返回false。

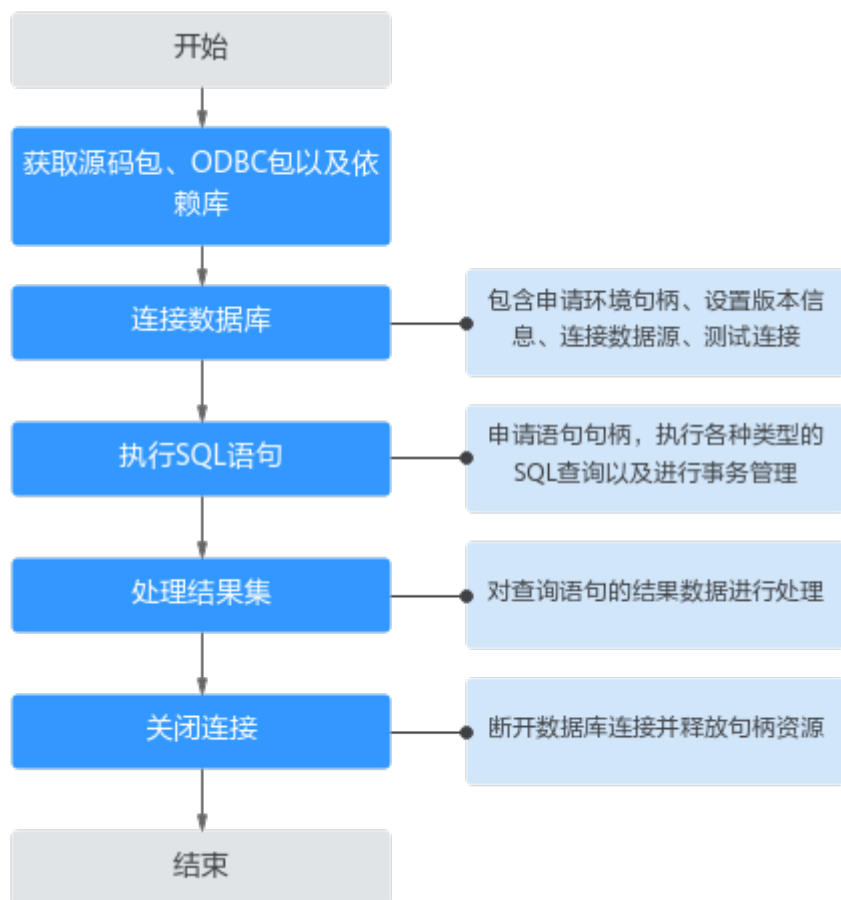
5.4 基于 ODBC 开发

5.4.1 开发流程

ODBC开发流程如图5-13所示。

其中包括获取源码包、ODBC包以及依赖库、连接数据库、执行SQL语句、处理结果集、关闭连接五个部分。

图 5-13 ODBC 开发流程



5.4.2 开发步骤

5.4.2.1 获取源码包、ODBC 包以及依赖库

基于ODBC开发用户需要获取unixODBC源码包、ODBC包以及依赖库。

获取源码包

- unixODBC源码包获取参考地址：<https://www.unixodbc.org/unixODBC-2.3.7.tar.gz>。

- MD5文件下载地址：<https://www.unixodbc.org/unixODBC-2.3.7.tar.gz.md5>。

下载后需使用MD5文件对unixODBC源码包进行完整性校验（查看MD5值，对比MD5值是否与源码包一致）。

获取驱动包

下载表5-54中的驱动包和驱动包校验包。

表 5-54 驱动包下载列表

版本	下载地址
V2.0-10.x	驱动包 驱动包校验包

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：
 - 上传软件包和软件包校验包到虚拟机的同一目录下。
 - 执行如下命令，校验软件包完整性。

```
F=$(echo DBS-GaussDB-driver*.tar.gz);echo "$(awk '{print $1}' "$F.sha256") $F" | sha256sum -c -
```

如果回显OK，则校验通过。例如：

```
DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz: OK
```
- Windows操作系统软件包完整性校验：
 - 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。
 - 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

 - {驱动包本地目录}：请根据实际下载目录进行替换。例如：C:\Users
 - {驱动包名}：请根据实际下载的驱动包名进行替换。例如：DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz示例：

```
certutil -hashfile C:\Users\DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz sha256
```
 - 将b获取到的Hash值和表5-54中获取到的驱动包校验包的Hash值进行比较。
 - 若一致则通过校验。
 - 若不一致，请重新下载驱动包，重复a~c进行校验。

获取 ODBC 包以及依赖库

ODBC包以及依赖库支持两种操作系统（Linux和Windows），请用户根据实际情况进行选择。

- Linux系统：
从驱动包中获取ODBC包，驱动包中提供多种数据库内核操作系统支持，用户可根据实际使用的数据库内核操作系统信息定位对应文件。驱动包的文件层级关系参考以下结构：DBS-GaussDB-driver_*/Distributed/驱动类型/操作系统信息（例如：Euler2.10_X86_64）。
压缩包名为GaussDB-Kernel_数据库版本号_Odbc_CPU架构信息_Distributed.tar.gz，将其解压后即可得到所需要的ODBC压缩包GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Odbc.tar.gz。
Linux环境下，开发应用程序需要使用unixODBC提供的头文件（sql.h、sqlext.h等）和库libodbc.so。这些头文件和库可从unixODBC-2.3.7源码包中获取。
- Windows系统：
从驱动包中获取ODBC包，驱动包中提供多种数据库内核操作系统支持，用户可根据实际使用的数据库内核操作系统信息定位对应文件。驱动包的文件层级关系参考以下结构：DBS-GaussDB-driver_*/Distributed/驱动类型/操作系统信息（例如：Euler2.10_X86_64）。
将文件中两个名称含有“Windows”的压缩包解压，即可得到GaussDB-Kernel_数据库版本号_Windows_X86_Odbc.tar.gz（32位）和GaussDB-Kernel_数据库版本号_Windows_X64_Odbc.tar.gz（64位）。
Windows环境下，开发应用程序所需的相关头文件和库文件由系统自身提供。

5.4.2.2 连接数据库

5.4.2.2.1 Linux 系统下配置数据源

在通过ODBC访问数据库之前，需要确保环境中已正确安装ODBC驱动及相关依赖。应用程序与数据库之间的通信依赖数据源（Data Source Name，DSN）配置，ODBC驱动程序管理器将根据DSN信息加载对应驱动、建立连接，并向应用程序提供统一的API接口。

本节说明如何在Linux系统下配置ODBC数据源（包含驱动配置文件与DSN配置文件的编辑要求），以及这些配置在连接流程中的作用。完成此步骤后，应用程序即可使用标准ODBC API调用数据库。

操作步骤

- 步骤1** 安装unixODBC（默认已获取unixODBC源码包）。若机器已安装了其他版本的unixODBC，可直接覆盖安装。

以unixODBC-2.3.7版本为例，在客户端执行如下命令安装unixODBC：

```
tar zxvf unixODBC-2.3.7.tar.gz
cd unixODBC-2.3.7

./configure --enable-gui=no #若在ARM架构服务器上编译，请追加一个configure参数：--build=aarch64-unknown-linux-gnu
make
#安装可能需要root权限
make install
```

 说明

- 目前不支持unixODBC-2.2.1版本。
- 默认安装至“/usr/local”目录下，生成数据源文件到“/usr/local/etc”目录下，库文件生成到“/usr/local/lib”目录。
- 通过编译带有--enable-fastvalidate=yes选项的unixODBC可提升性能，但可能会导致向ODBC API传递无效句柄的应用程序发生故障，而非返回SQL_INVALID_HANDLE错误。

步骤2 替换客户端GaussDB驱动程序。

解压GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Odbc.tar.gz，解压后会得到lib和odbc两个文件夹，在odbc文件夹中还会有一个lib文件夹，将/lib文件夹与/odbc/lib文件夹中的所有动态库都复制到“/usr/local/lib”目录下。

步骤3 配置数据源。

1. 配置ODBC驱动文件。

在“/usr/local/etc/odbcinst.ini”文件中追加以下内容：

```
[GaussMPP]
Driver64=/usr/local/lib/gsqlodbcw.so
setup=/usr/local/lib/gsqlodbcw.so
```

odbcinst.ini文件中的配置参数说明如表5-55所示：

表 5-55 odbcinst.ini 文件配置参数

参数	描述	示例
[DriverName]	驱动器名称，对应数据源DSN中的驱动名。	[GaussMPP]
Driver64	驱动动态库的路径。	Driver64=/usr/local/lib/gsqlodbcw.so
setup	驱动安装路径，与Driver64中动态库的路径一致。	setup=/usr/local/lib/gsqlodbcw.so

2. 配置数据源文件。

在“/usr/local/etc/odbc.ini”文件中追加以下内容：

```
[gaussdb]
Driver=GaussMPP
#数据库Server IP
Servername=127.0.0.1
#数据库名
Database=db1
#数据库用户名
Username=omm
#数据库用户密码
Password=
#数据库侦听端口
Port=8000
Sslmode=allow
```

当需要变更已有数据源的参数（如服务器IP、端口、数据库名称或认证信息）时，可直接编辑odbc.ini文件中对应的DSN（Data Source Name）条目并更新相关配置项。修改并保存后，新配置将在应用程序下一次建立连接时生效。

若不再使用某个数据源，可在odbc.ini文件中删除该DSN的完整配置段。删除后，依赖该DSN的应用程序将无法再通过其建立ODBC连接。

odbc.ini文件配置参数说明如表5-142所示：

步骤4 在客户端配置环境变量。

```
vim ~/.bashrc
```

在配置文件中追加以下内容：

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
export ODBCYSINI=/usr/local/etc
export ODBCINI=/usr/local/etc/odbc.ini
```

步骤5 执行如下命令使设置生效。

```
source ~/.bashrc
```

步骤6 测试连接。

安装后，/usr/bin目录下面会存放生成的二进制文件。

执行以下命令，其中“gaussdb”为数据源名称：

```
isql -v gaussdb
```

- 若显示如下信息，表明配置正确，连接成功。

```
+-----+
| Connected!                                |
|                                            |
| sql-statement                            |
| help [tablename]                        |
| quit                                    |
|                                            |
+-----+
```

- 若显示ERROR信息，则表明配置错误。请检查上述配置步骤是否正确执行。
- 若是集群环境，需要在所有机器上都复制配置一份unixODBC。

说明

目前通过ODBC连接数据库时，会如下设置内核参数：

```
SET extra_float_digits = 2;
SET DateStyle = 'ISO';
```

这些参数可能会导致ODBC客户端的行为与gsqldb客户端的行为不一致，例如，Date数据显示方式、浮点数精度表示。如果实际期望和这些配置不符，建议在ODBC应用代码中显式设定这些参数。

M-Compatibility模式数据库下，extra_float_digits的默认值为0。

----结束

常见问题处理

- [UnixODBC][Driver Manager]Can't open lib 'xxx/xxx/gsqlodbcw.so':file not found.

此问题的可能原因：

- odbcinst.ini文件中配置的路径不正确。

确认的方法：执行ls命令查询错误信息中的路径，以确保该gsqlodbcw.so文件存在，同时具有执行权限。

- gsqlodbcw.so的依赖库不存在，或者不在系统环境变量中。

确认的方法：执行ldd命令查询错误信息中的路径，如果是缺少libodbc.so.1等UnixODBC的库，那么按照“操作步骤”中的方法重新配置UnixODBC，并确保它的安装路径下的lib目录添加到了LD_LIBRARY_PATH中。如果重装仍无法解决，可以手动将数据库安装包下的unixodbc/lib下的内容复制到UnixODBC

的安装路径下的lib目录。如果是缺少其他库，请将ODBC驱动包中的lib目录添加到LD_LIBRARY_PATH中。如果缺少其他标准库，请自行安装。

- [UnixODBC]connect to server failed: no such file or directory

此问题的可能原因：

- 配置了错误的/不可达的数据库地址或者端口。

请检查数据源配置中的Servername及Port配置项。

- 服务器侦听不正确。

如果确认Servername及Port配置正确，请根据[操作步骤](#)中数据库服务器的相关配置，确保数据库侦听了合适的网卡及端口。

- 防火墙及网闸设备。

请确认防火墙设置，将数据库的通信端口添加到可信端口中。

如果有网闸设备，请确认相关的设置。

- [unixODBC]The password-stored method is not supported.

此问题的可能原因：

数据源中未配置Sslmode配置项。

请配置该选项至allow或以上选项。此配置的更多信息，请参见[表5-143](#)。

- Server common name "xxxx" does not match host name "xxxxx"

此问题的可能原因：

使用了SSL加密的“verify-full”选项，驱动程序会验证证书中的主机名与实际部署数据库的主机名是否一致。

碰到此问题可以使用“verify-ca”选项，不再校验主机名，或者重新生成一套与数据库所在主机名相同的服务端证书。

- Driver's SQLAllocHandle on SQL_HANDLE_DBC failed

此问题的可能原因：

可执行文件（比如UnixODBC的isql，以下都以isql为例）与数据库驱动（gsqldbwcw.so）依赖于不同的ODBC的库版本：libodbc.so.1或者libodbc.so.2。此问题可以通过如下方式确认：

```
ldd `which isql` | grep odbc  
ldd gsqlodbcw.so | grep odbc
```

这时，如果输出的libodbc.so最后的后缀数字不同或者指向不同的磁盘物理文件，那么基本就可以断定是此问题。isql与gsqldbwcw.so都会要求加载libodbc.so，这时如果它们加载的是不同的物理文件，便会导致两套完全同名的函数列表，同时出现在同一个可见域里（UnixODBC的libodbc.so.*的函数导出列表完全一致），产生冲突，无法加载数据库驱动。

确定一个要使用的UnixODBC，卸载另外一个（比如卸载库版本号为.so.2的UnixODBC），然后将剩下的.so.1的库，新建一个同名但是后缀为.so.2的软链接，便可解决此问题。

- FATAL: Forbid remote connection with trust method!

此问题的可能原因：

由于安全原因，数据库CN禁止集群内部其他节点无认证接入。

如果需要在集群内部访问CN，请将ODBC程序部署在CN所在机器，服务器地址使用“127.0.0.1”。建议业务系统单独部署在集群外部，否则可能会影响数据库运行性能。

- [unixODBC][Driver Manager]Invalid attribute value

此问题的可能原因：

在使用SQL on other GaussDB功能时出现此问题，可能与非推荐版本的unixODBC相关。

建议通过执行“odbcinst --version”命令验证环境中的unixODBC版本。

- authentication method 10 not supported.

使用开源客户端时出现此错误。可能原因：

数据库中用户密码仅以SHA256哈希值的形式存储，而不存储明文密码。

早期版本为了兼容某些开源客户端，同时存储了MD5哈希值和SHA256哈希值两种格式。当用户登录时，服务端会优先尝试使用客户端发送的校验方式（MD5或SHA256）进行验证对应的哈希值。

在从旧版本数据库升级到新版本时，由于哈希算法的单向不可逆特性，系统无法从已有的SHA256哈希值逆推出原始明文密码，也就无法再生成对应的MD5哈希值。因此升级完成后，数据库中只保留了SHA256哈希值，导致使用MD5校验方式的开源客户端认证失败，报此错误。

说明

MD5算法已被证明存在严重的安全缺陷，不再推荐使用。建议所有客户端逐步升级到支持SHA256或更强哈希算法的版本，以彻底消除兼容性与安全隐患。

该问题的解决方案包括：

- 更新用户密码（请参见《参考》中“SQL参考 > SQL语法 > A > ALTER USER”）。
- 新建用户并赋予和原用户同等的权限（请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”），使用新用户连接数据库。
- unsupported frontend protocol 3.51: server supports 1.0 to 3.0
此问题的可能原因：
目标数据库版本过低，或者目标数据库为开源数据库。
请使用与目标数据库版本匹配的数据库驱动连接目标数据库。
- FATAL: GSS authentication method is not allowed because XXXX user password is not disabled.
目标CN的gs_hba.conf文件中，配置了当前客户端IP使用“gss”方式来做认证。该认证算法不支持用作客户端的身份认证。请修改配置为“sha256”后重试。
- isql: error while loading shared libraries:xxx
当前环境缺少该动态库文件，需要自行安装对应的动态库。

5.4.2.2.2 Windows 系统下配置数据源

Windows操作系统自带ODBC数据源管理器，无需用户手动安装管理器即可直接进行配置。

操作步骤

步骤1 替换客户端GaussDB驱动程序。

根据需要，将包名为GaussDB-Kernel_数据库版本号_Windows_X64_Odbc.tar.gz的64位驱动或包名为GaussDB-Kernel_数据库版本号_Windows_X86_Odbc.tar.gz的32位驱动解压后，单击gsqldb.exe进行驱动安装。

步骤2 打开驱动管理器。

请使用ODBC版本对应的ODBC驱动管理器（如果使用64位ODBC驱动，必须使用64位的ODBC驱动管理器。假设操作系统安装盘符为C盘，如果是其他盘符，请对路径做相应修改）。

- 如果需要在64位操作系统使用32位ODBC驱动请使用：C:\Windows\SysWOW64\odbcad32.exe，请勿直接使用“控制面板 > 管理工具 > 数据源(ODBC)”。

 说明

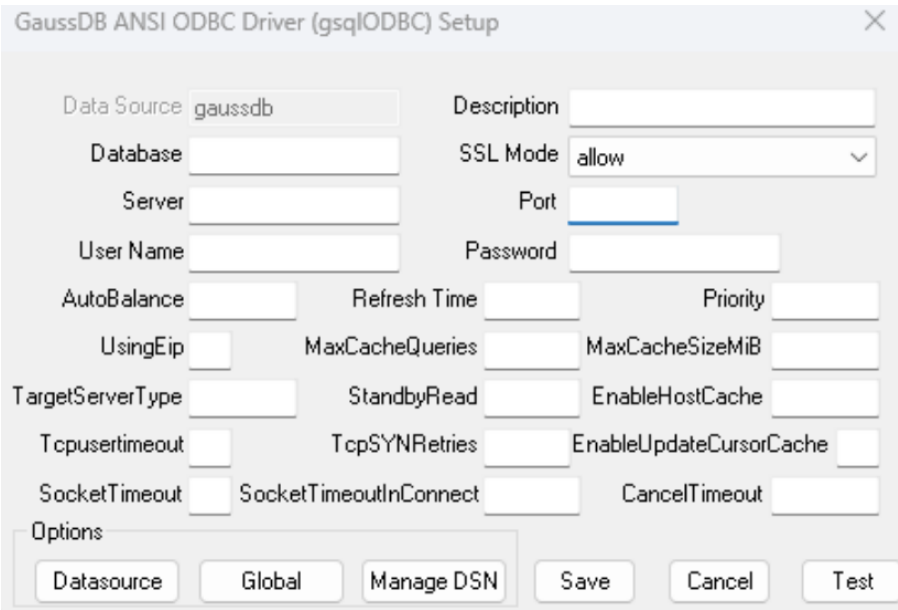
WOW64的全称是“Windows 32-bit on Windows 64-bit”，C:\Windows\SysWOW64\存放的是64位系统上的32位运行环境。而C:\Windows\System32\存放的是与操作系统一致的运行环境，具体的技术信息请查阅Windows的相关技术文档。

- 32位操作系统请使用：C:\Windows\System32\odbcad32.exe，或者单击“计算机 > 控制面板 > 管理工具 > 数据源(ODBC)”打开驱动管理器。
- 64位操作系统请使用：控制面板 > 管理工具 > 数据源(ODBC) 打开驱动管理。

步骤3 配置数据源。

在打开的驱动管理器上，选择“用户DSN > 添加 > GaussDB Unicode”，然后进行配置。如下图5-14所示：

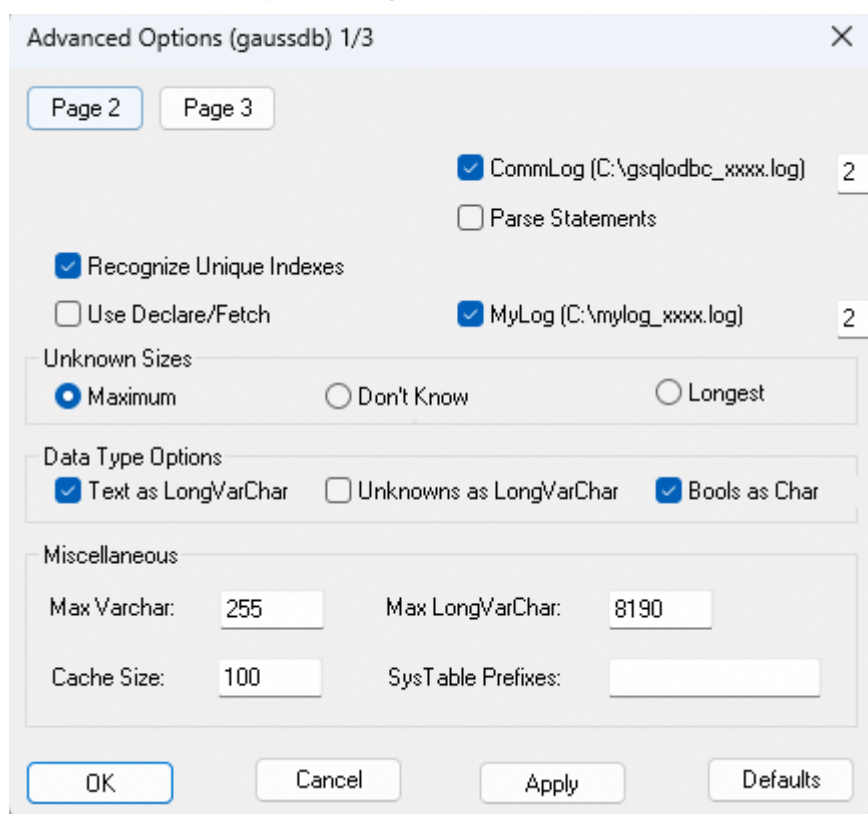
图 5-14 ODBC Driver Setup



参数说明请参见表5-142。

其中单击Datasource可以选择配置是否打印日志，单击后弹出对话框如下图5-15所示：

图 5-15 Advanced Options Page1



须知

此界面上配置的用户名及密码信息，将会被记录在Windows注册表中，再次连接数据库时不再需要输入认证信息。但是出于安全考虑，建议在单击“Save”按钮保存配置信息前，清空相关敏感信息，在使用ODBC的连接API时，再传入所需的用户名、密码信息。

步骤4 SSL模式。

如需使用SSL模式请将步骤3中设置窗口的“SSL Mode”选项调整至“require”。

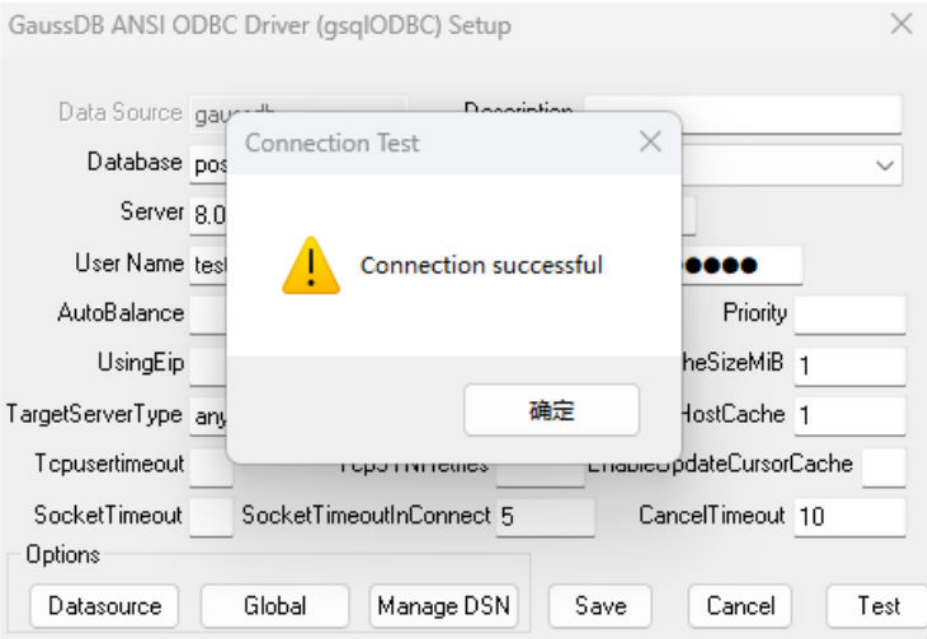
Sslmode的可选及其描述请参见表5-143。

步骤5 测试连接。

单击Test进行测试。

- 若显示信息如下，则表明配置正确，连接成功。

图 5-16 配置成功



- 若显示ERROR信息，则表明配置错误。请重新检查上述配置是否正确。

说明

目前通过ODBC连接数据库时，会设置如下参数：
SET extra_float_digits = 2;
SET DateStyle = 'ISO';

这些参数可能会导致ODBC客户端的行为与gsq客户端的行为不一致，例如，Date数据显示方式、浮点数精度表示。如果实际期望和这些配置不符，建议在ODBC应用代码中显式设定这些参数。

M-Compatibility模式数据库下，extra_float_digits的默认值为0。

----结束

常见问题处理

- connect to server failed: no such file or directory
此问题的可能原因：
 - 配置了错误的/不可达的数据库地址或者端口。
请检查数据源配置中的Server及Port配置项。
 - 服务器侦听不正确。
如果确认Server及Port配置正确，请根据[操作步骤](#)中数据库服务器的相关配置，确保数据库侦听了合适的网卡及端口。
 - 防火墙及网闸设备。
请确认防火墙设置，将数据库的通信端口添加到可信端口中。
如果有网闸设备，请确认相关的设置。
- The password-stored method is not supported.
此问题的可能原因：
数据源中未配置Sslmode配置项，请调整此项至allow或以上级别，允许SSL连接，此选项的更多说明，请参见[表5-143](#)。

- authentication method 10 not supported.
使用开源客户端时出现此错误。可能原因：
数据库中用户密码仅以SHA256哈希值的形式存储，而不存储明文密码。
早期版本为了兼容某些开源客户端，同时存储了MD5哈希值和SHA256哈希值两种格式。当用户登录时，服务端会优先尝试使用客户端发送的校验方式（MD5或SHA256）进行验证对应的哈希值。
在从旧版本数据库升级到新版本时，由于哈希算法的单向不可逆特性，系统无法从已有的SHA256哈希值逆推出原始明文密码，也就无法再生成对应的MD5哈希值。因此升级完成后，数据库中只保留了SHA256哈希值，导致使用MD5校验方式的开源客户端认证失败，报此错误。
数据库中的密码校验仅存储了SHA256格式哈希值，而开源客户端仅识别MD5校验，双方校验方法不匹配报错。

📖 说明

MD5算法已被证明存在严重的安全缺陷，不再推荐使用。建议所有客户端逐步升级到支持SHA256或更强哈希算法的版本，以彻底消除兼容性与安全隐患。
该问题的解决方案包括：
 - 更新用户密码（请参见《参考》中“SQL参考 > SQL语法 > A > ALTER USER”）。
 - 新建新用户并赋予和原用户同等的权限（请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”），使用新用户连接数据库。
- unsupported frontend protocol 3.51: server supports 1.0 to 3.0
此问题的可能原因：
目标数据库版本过低，或者目标数据库为开源数据库。
请使用与目标数据库版本匹配的数据库驱动连接目标数据库。
- FATAL: GSS authentication method is not allowed because XXXX user password is not disabled.
目标CN的gs_hba.conf文件中，配置了当前客户端IP使用“gss”方式来做认证。该认证算法不支持客户端身份认证。请修改配置为“sha256”后重试。

5.4.2.2.3 连接数据库涉及的 API

在完成数据源配置后，应用程序需要通过一组标准的ODBC API与数据库建立连接。该连接过程包含环境句柄与连接句柄的分配、属性设置以及实际连接动作，相关API说明如表5-56所示。ODBC驱动程序管理器在此过程中负责加载驱动、解析DSN参数，并初始化后续所有数据库交互所需的上下文。

表 5-56 相关 API 说明

功能	API
申请句柄资源	SQLAllocHandle ：申请句柄资源，可替代如下函数： ● SQLAllocEnv：申请环境句柄。 ● SQLAllocConnect：申请连接句柄。 ● SQLAllocStmt：申请语句句柄。
设置环境属性	SQLSetEnvAttr

功能	API
设置连接属性	SQLSetConnectAttr
连接数据库	SQLConnect

以开发源程序DBtest.c为例（完整示例请参考[获取和处理数据库中的数据](#)）：

```
// DBtest.c (compile with: libodbc.so)
// 程序头文件和全局变量请参考完整示例

// 申请环境句柄。
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    printf("Error AllocHandle\n");
    exit(0);
}

// 设置版本信息（环境属性）。
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

// 申请连接句柄 。
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
    exit(0);
}

// 获取用户名和用户密码。
char *userName;
userName = getenv("EXAMPLE_USERNAME_ENV");
char *password;
password = getenv("EXAMPLE_PASSWORD_ENV");

// 设置连接属性。
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT,(SQLPOINTER *)SQL_AUTOCOMMIT_ON, 0);

// 连接数据库，这里的userName与password分别表示连接数据库的用户名和用户密码。
// 如果odbc.ini文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦odbc.ini
// 权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
    (SQLCHAR*) userName, SQL_NTS, (SQLCHAR*) password, SQL_NTS);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    printf("Error SQLConnect %d\n",V_OD_erg);
    SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
    exit(0);
}
printf("Connected !\n");
```

5.4.2.3 执行 SQL 语句

为了执行SQL语句，ODBC提供一组专门的API，并在[表5-57](#)中集中列出了相关API并进行了说明。这些API覆盖了SQL的预处理、参数绑定、直接执行以及语句属性设置等功能，是应用程序与数据库进行指令交互的核心接口。

表 5-57 相关 API 说明

功能	API
设置语句属性	SQLSetStmtAttr
为执行SQL语句做准备	SQLPrepare
执行一条准备好的SQL语句	SQLExecute
绑定SQL语句的参数标志和缓冲区	SQLBindParameter
直接执行SQL语句	SQLExecDirect

 说明

- ODBC为应用程序与数据库的中心层，负责把应用程序发出的SQL指令传到数据库当中，自身并不解析SQL语法。故在应用程序中写入带有保密信息的SQL语句时（如明文密码），保密信息会被暴露在驱动日志中。
- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，如果其中有一个语句失败，那么整个请求都将会被回滚。
- 关闭事务自动提交后，需要显式手动提交事务，否则可能引发问题。例如，执行SELECT语句时会隐式开启事务，如果不手动提交，该隐式事务将无法正常结束，演变为长事务，导致一系列潜在问题。

示例如下（完整示例请参考[获取和处理数据库中的数据](#)）：

```
// 申请语句句柄。
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);

// 设置语句属性。
SQLSetStmtAttr(V_OD_hstmt, SQL_ATTR_QUERY_TIMEOUT, (SQLPOINTER *)3, 0);

// 直接执行SQL语句。
SQLExecDirect(V_OD_hstmt, "drop table IF EXISTS customer_t1", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name VARCHAR(32));", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "insert into customer_t1 values(25, 'li')", SQL_NTS);

// 准备执行。
SQLPrepare(V_OD_hstmt, "insert into customer_t1 values(?)", SQL_NTS);

// 绑定参数。
SQLBindParameter(V_OD_hstmt,1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
    &value, 0, NULL);

// 执行准备好的语句。
SQLExecute(V_OD_hstmt);    SQLExecDirect(V_OD_hstmt, "select c_customer_sk from customer_t1",
SQL_NTS);
```

5.4.2.4 处理结果集

ODBC处理结果集是从数据库获取数据并供应用程序进行处理的过程，包括：检索数据、展示、处理、传输以及业务逻辑实现等。相关API如[表5-58](#)所示。

表 5-58 相关 API 说明

功能	API
绑定缓冲区到结果集的列中	SQLBindCol
结果集中取行集	SQLFetch
返回结果集中某一列的数据	SQLGetData
获取结果集中列的描述信息	SQLColAttribute
查看最近一次操作错误信息	SQLGetDiagRec

示例如下（完整示例请参考[获取和处理数据库中的数据](#)）：

```
// 在执行完成SQL语句后，获取结果集某一列的属性。
SQLColAttribute(V_OD_hstmt,1,SQL_DESC_TYPE_NAME,typename,100,NULL,NULL);
printf("SQLColAttribute %s\n",typename);

// 绑定结果集。
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER*)&V_OD_buffer,150,
           (SQLLEN *)&V_OD_err);

// 通过SQLFetch取结果集中数据。
V_OD_erg=SQLFetch(V_OD_hstmt);

// 通过SQLGetData获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
    SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER*)&V_OD_id,0,NULL);
    printf("SQLGetData ----ID = %d\n",V_OD_id);
    V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");
```

5.4.2.5 关闭连接

在完成数据库操作后，需通过ODBC API正确关闭语句、连接和环境句柄，释放驱动及数据库端的资源。根据ODBC句柄管理机制，应用程序需按照“语句 → 连接 → 环境”的顺序依次释放，以避免资源泄漏或驱动状态异常。ODBC关闭数据库连接，释放资源相关API如表5-59所示。

表 5-59 相关 API 说明

功能	API
断开与数据源的连接	SQLDisconnect
释放句柄资源	SQLFreeHandle ：释放句柄资源，可替代如下函数： ● SQLFreeEnv：释放环境句柄 ● SQLFreeConnect：释放连接句柄 ● SQLFreeStmt：释放语句句柄

示例如下（完整示例请参考[获取和处理数据库中的数据](#)）：

```
// 断开数据源连接并释放句柄资源。  
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt);  
SQLDisconnect(V_OD_hdbc);  
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc);  
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
```

5.4.3 典型应用开发示例

5.4.3.1 典型应用场景配置

日志诊断场景

ODBC日志分为unixODBC驱动管理器日志和gsqLODBC驱动端日志。前者可以用于追踪应用程序API调用的执行状态，后者包含驱动底层实现的诊断（DFX）日志，用于帮助定位问题。

unixODBC日志需要在odbcinst.ini文件中配置：

```
[ODBC]  
Trace=Yes  
TraceFile=/path/to/odbctrace.log  
  
[GaussMPP]  
Driver64=/usr/local/lib/gsqlodbcw.so  
setup=/usr/local/lib/gsqlodbcw.so
```

gsqLODBC日志只需在odbc.ini加上如下内容：

```
[gaussdb]  
Driver=GaussMPP  
#数据库Server IP  
Servername=10.10.0.13  
...  
#打开驱动端debug日志，日志级别为TRACE  
Debug=5
```

说明

unixODBC日志将生成在TraceFile指定的路径下，gsqLODBC驱动生成的调试日志将存储在系统“/tmp/”目录下的“mylog_xxx.log”文件中。

负载均衡场景

当应用程序有大并发场景时可开启负载均衡：

- 负载均衡即为将并发连接随机分发到所有CN上，避免单个CN负载过大，达到提高性能的目的。
- 配置参数AutoBalance=1，开启负载均衡功能。
- 参数RefreshCNListTime=5可以选择性配置，默认刷新时间为10s。
- 参数Priority=1可以选择性配置，即为并发连接优先发送到配置文件中配置的CN上，当配置的CN全部不可连接时，连接才会被分发到剩余CN上。

示例场景：

集群环境有6个CN，CN1、CN2、CN3、CN4、CN5和CN6；配置文件配置4个CN，CN1、CN2、CN3和CN4。

odbc.ini配置文件示例：

```
[gaussdb]  
Driver=GaussMPP
```

```
#数据库Server IP。
Servername=10.145.130.26,10.145.130.27,10.145.130.28,10.145.130.29
#数据库名称。
Database=db1
#数据库用户名。
Username=omm
#数据库用户密码。
Password=
#数据库侦听端口。
Port=8000
Sslmode=allow
AutoBalance=1
RefreshCNListTime=3
Priority=1
```

当配置文件和集群环境如示例时，并发连接会随机、平均发送到CN1、CN2、CN3和CN4上，连接数均衡。当CN1、CN2、CN3和CN4全部不可用时，并发连接会随机、平均发送到CN5和CN6上。如果此时CN1、CN2、CN3和CN4中有CN重新可用时，连接则不会再发送到CN5和CN6上，而是重新发送到重新可用的CN上。

备机读场景

配置文件配置CN节点，设置StandbyRead=1开启分布式备机读模式。

odbc.ini配置文件示例：

```
[gaussdb]
Driver=GaussMPP
#数据库Server IP。
Servername=10.145.130.26,10.145.130.27,10.145.130.28,10.145.130.29
#数据库名称。
Database=db1
#数据库用户名称。
Username=omm
#数据库用户密码。
Password=
#数据库侦听端口。
Port=8000
#打开备机读模式。
StandbyRead=1
Sslmode=allow
#开启负载均衡模式。
AutoBalance=1
```

连接池场景

连接池允许应用程序重复使用预先建立的连接，无需每次都重新建立连接。连接一旦创建并放入连接池，应用程序即可重复利用，避免了重复执行完整的连接过程。

该池化连接机制的使用可以显著提升性能，尤其对于需要频繁建立和断开连接的中间层应用程序或网络连接的应用程序而言，性能提升尤为明显。

除了性能优势外，连接池架构还能够实现环境中的连接在单个进程中被多个组件共享的目的。这意味着，同一进程中的不同组件可以在不互相干扰的情况下共享连接池中的连接，进一步提高了系统的运行效率和资源利用率。

说明

在连接池中，打开的连接可能被多个用户重用，如果您的应用程序脚本会更改数据库连接的状态，可能会导致数据状态冲突或泄露，为了安全起见，建议谨慎评估使用连接池。

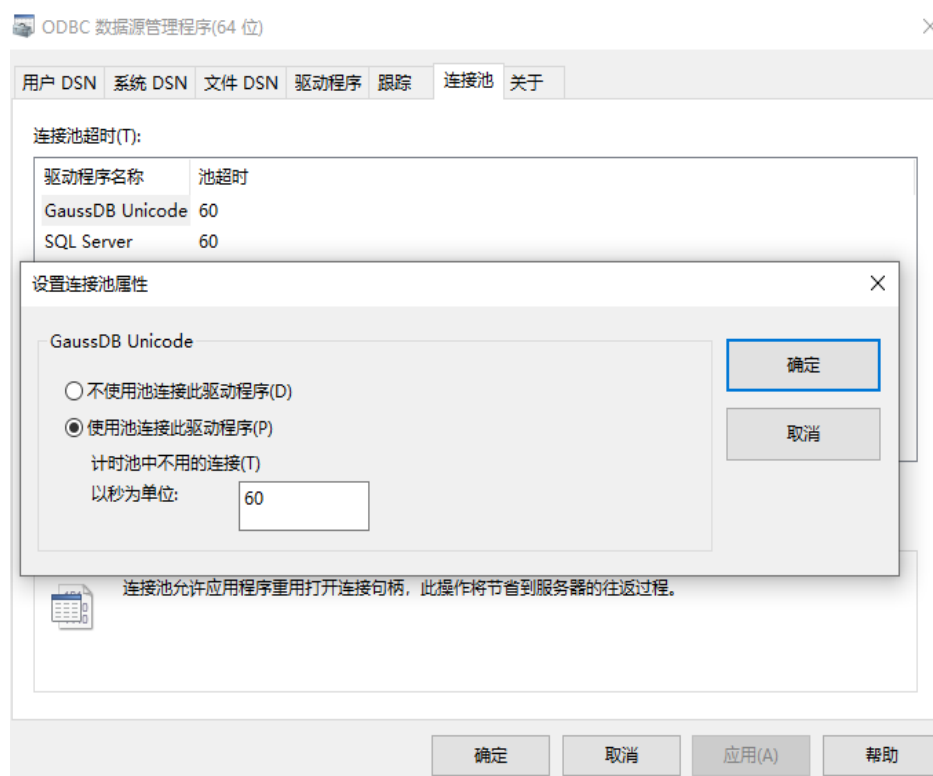
Linux场景配置

在odbcinst.ini配置文件中开启连接池，连接池相关参考配置如下：


```
[ODBC]
#开启连接池
Pooling=Yes
[GaussMPP]
#一个连接在连接池中未被重用则会被释放的计时，默认为0，开启连接池需要设置为大于0
CPOut=60
#该驱动下连接池存在的计时
CPTimeToLive=60
[GaussMPP2]
#关闭连接池
CPOut=0
```

Windows场景配置

在打开的驱动管理器上，选择“连接池”后双击“GaussDB Unicode”驱动名称，选择“使用池连接此驱动程序”[默认为60s]，该设置参数同Linux场景配置的CPOut参数，参考例图如下：



说明

1. 在应用程序中配置连接池参数，需要在环境句柄创建前通过调用SQLSetEnvAttr设置连接池参数，其中的环境句柄应设置为NULL，这使得SQL_ATTR_CONNECTION_POOLING成为进程级属性。目前在Windows上支持配置SQL_ATTR_CONNECTION_POOLING为以下两种值：
 - SQL_CP_OFF：默认参数设置，禁用连接池。
 - SQL_CP_ONE_PER_DRIVER：开启连接池，每个驱动支持一个连接池，驱动其中的所有连接共享一个池。
2. 应用程序调用SQLConnect或SQLDriverConnect时，会从连接池中提取连接，如果连接已超时，或者池中没有一个与请求匹配的连接，则会打开一个新连接，连接池对调用应用程序是透明的。
3. 应用程序调用SQLDisconnect时，连接将被放回连接池而非直接释放，供后续请求复用。
4. 在环境上调用SQLFreeHandle释放环境句柄前，应用程序成功为环境设置的所有环境属性都会持续存在。
5. 若应用程序的连接在一段时间内处于非活动状态（未在连接中使用），则会从池中删除该连接。连接池的大小仅受内存限制和服务器限制。

5.4.3.2 获取和处理数据库中的数据

此示例完整演示如何通过ODBC获取和处理GaussDB中的数据。

前提条件：数据源已配置成功。Linux系统请参考[Linux系统下配置数据源](#)；Windows系统请参考[Windows系统下配置数据源](#)。

说明

- Windows环境下ODBC应用代码可以使用MinGW（Minimalist GNU for Windows）编译器进行编译。编译命令如下：

```
gcc odbctest.c -o odbctest -lodb32
```

执行命令为：

```
./odbctest.exe
```

- 在Linux环境下ODBC应用代码可以使用GCC（GNU Compiler Collection）编译器进行编译。编译命令如下：

```
gcc odbctest.c -o odbctest -lodbcc
```

执行命令为：

```
./odbctest
```

如果编译找不到sql.h或者API接口，尝试手动连接unixodbc的头文件和动态库，即：

```
gcc -I /home/omm/unixodbc/include -L /home/omm/unixodbc/lib odbctest.c -o odbctest -lodbcc
```

```
// DBtest.c (compile with: libodbc.so)
```

```
// 本示例以用户名和密码保存在环境变量中为例，运行此示例前请先在本地环境中设置环境变量（环境变量名称请根据需求进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
```

```
#ifdef WIN32
```

```
#include <windows.h>
```

```
#endif
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <sql.h>
```

```
#include <sqltext.h>
```

```
SQLHENV      V_OD_Env;      // Handle ODBC environment
```

```
SQLHSTMT     V_OD_hstmt;    // Handle statement
```

```
SQLHDBC      V_OD_hdbc;    // Handle connection
```

```
char          typename[100];
```

```
SQLINTEGER   value = 100;
```

```
SQLINTEGER   V_OD_erg,V_OD_buffer,V_OD_err,V_OD_id;
```

```
int main(int argc,char *argv[])
```

```
{
```

```
// 1. 申请环境句柄。
```

```
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    printf("Error AllocHandle\n");
    exit(0);
}

// 2. 设置版本信息（环境属性）。
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

// 3. 申请连接句柄。
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
    exit(0);
}

// 获取用户名和用户密码。
char *userName;
userName = getenv("EXAMPLE_USERNAME_ENV");
char *password;
password = getenv("EXAMPLE_PASSWORD_ENV");

// 4. 设置连接属性。
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT,(SQLPOINTER *)SQL_AUTOCOMMIT_ON, 0);

// 5. 连接数据库，这里的userName与password分别表示连接数据库的用户名和用户密码。
// 如果odbc.ini文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦
odbc.ini权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
                      (SQLCHAR*) userName, SQL_NTS, (SQLCHAR*) password, SQL_NTS);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
    printf("Error SQLConnect %d\n",V_OD_erg);
    SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
    exit(0);
}
printf("Connected !\n");

// 6. 申请语句句柄。
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);

// 7. 设置语句属性。
SQLSetStmtAttr(V_OD_hstmt, SQL_ATTR_QUERY_TIMEOUT, (SQLPOINTER *)3,0);

// 8. 直接执行SQL语句。
SQLExecDirect(V_OD_hstmt, "drop table IF EXISTS customer_t1", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name
VARCHAR(32));", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "insert into customer_t1 values(25,'li')", SQL_NTS);

// 9. 准备执行。
SQLPrepare(V_OD_hstmt,"insert into customer_t1 values(?)",SQL_NTS);

// 10. 绑定参数。
SQLBindParameter(V_OD_hstmt,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,0,0,
                 &value,0,NULL);

// 11. 执行准备好的语句。
SQLExecute(V_OD_hstmt);
SQLExecDirect(V_OD_hstmt,"select c_customer_sk from customer_t1",SQL_NTS);

// 12. 获取结果集某一列的属性。
SQLColAttribute(V_OD_hstmt,1, SQL_DESC_TYPE_NAME,typename,100, NULL, NULL);
printf("SQLColAttribute %s\n",typename);

// 13. 绑定结果集。
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER)&V_OD_buffer,150,
```

```
(SQLLEN *)&V_OD_err);

// 14. 通过SQLFetch取结果集中数据。
V_OD_erg=SQLFetch(V_OD_hstmt);

// 15. 通过SQLGetData获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
    SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER)&V_OD_id,0,NULL);
    printf("SQLGetData ----ID = %d\n",V_OD_id);
    V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");

// 16. 断开数据源连接并释放句柄资源。
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt);
SQLDisconnect(V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
return(0);
}
```

运行结果如下：

```
Connected !
SQLColAttribute int4
SQLGetData ----ID = 25
SQLGetData ----ID = 100
Done!
```

5.4.3.3 批量绑定

前提条件：数据源已配置成功。Linux系统请参考[Linux系统下配置数据源](#)；Windows系统请参考[Windows系统下配置数据源](#)。

```
/*
*****
* 请在数据源中打开UseBatchProtocol，同时指定数据库中参数support_batch_bind为on。
* CHECK_ERROR的作用是检查并打印错误信息。
* 此示例将与用户交互式获取DSN、模拟的数据量，忽略的数据量，并将最终数据入库到test_odbc_batch_insert
  中。
*****
*/
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
void Exec(SQLHDBC hdbc, SQLCHAR* sql)
{
    SQLRETURN retcode;           // Return status
    SQLHSTMT hstmt = SQL_NULL_HSTMT; // Statement handle
    SQLCHAR loginfo[2048];

    // 分配语句句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
        return;
    }

    // 准备语句。
    retcode = SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS);
    sprintf((char*)loginfo, "SQLPrepare log: %s", (char*)sql);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed");
    }
}
```

```
        return;
    }

    // 执行语句。
    retcode = SQLExecute(hstmt);
    sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLExecute(hstmt) failed");
        return;
    }

    // 释放句柄。
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    sprintf((char*)loginfo, "SQLFreeHandle stmt log: %s", (char*)sql);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt) failed");
        return;
    }
}

int main ()
{
    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    long int batchCount = 1000; // 批量绑定的数据量
    SQLLEN rowsCount = 0;
    int ignoreCount = 0;        // 批量绑定的数据中，不要入库的数据量
    int i = 0;
    SQLRETURN retcode;
    SQLCHAR dsn[1024] = {'\0'};
    SQLCHAR loginfo[2048];
    do
    {
        if (ignoreCount > batchCount)
        {
            printf("ignoreCount(%d) should be less than batchCount(%d)\n", ignoreCount, batchCount);
        }
    }while(ignoreCount > batchCount);

    // 分配环境句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle failed");
        goto exit;
    }

    // 设置ODBC版本。
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER*)SQL_OV_ODBC3, 0);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLSetEnvAttr failed");
        goto exit;
    }

    // 分配连接。
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle failed");
        goto exit;
    }

    // 设置登录超时。
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);

    if (!SQL_SUCCEEDED(retcode)) {
```

```
printf("SQLSetConnectAttr failed");
goto exit;
}

// 设置自动提交。
retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT,
                             (SQLPOINTER)(1), 0);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLSetConnectAttr failed");
    goto exit;
}

// 连接到数据库。
sprintf(loginfo, "SQLConnect(DSN:%s)", dsn);
retcode = SQLConnect(hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
                     (SQLCHAR*) NULL, 0, NULL, 0);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLConnect failed");
    goto exit;
}

// 初始化表信息。
Exec(hdbc, "drop table if exists test_odbc_batch_insert");
Exec(hdbc, "create table test_odbc_batch_insert(id int primary key, col varchar2(50))");
// 下面的代码根据用户输入的数据量，构造出将要入库的数据：
{
    SQLRETURN retcode;
    SQLHSTMT hstmtinsrt = SQL_NULL_HSTMT;
    SQLCHAR *sql = NULL;
    SQLINTEGER *ids = NULL;
    SQLCHAR *cols = NULL;
    SQLLEN *bufLenIds = NULL;
    SQLLEN *bufLenCols = NULL;
    SQLUSMALLINT *operptr = NULL;
    SQLUSMALLINT *statusptr = NULL;
    SQLULEN process = 0;

    // 这里是按列构造，每个字段的内存连续存放在一起。
    ids = (SQLINTEGER*)malloc(sizeof(ids[0]) * batchCount);
    cols = (SQLCHAR*)malloc(sizeof(cols[0]) * batchCount * 50);

    // 这里是每个字段中，每一行数据的内存长度。
    bufLenIds = (SQLLEN*)malloc(sizeof(bufLenIds[0]) * batchCount);
    bufLenCols = (SQLLEN*)malloc(sizeof(bufLenCols[0]) * batchCount);

    // 该行是否需要被处理，SQL_PARAM_IGNORE 或 SQL_PARAM_PROCEED。
    operptr = (SQLUSMALLINT*)malloc(sizeof(operptr[0]) * batchCount);
    memset(operptr, 0, sizeof(operptr[0]) * batchCount);

    // 该行的处理结果。
    // 注：由于数据库中处理方式是同一语句隶属同一事务中，所以如果出错，那么待处理数据都将是出错的，
    并不会部分入库。
    statusptr = (SQLUSMALLINT*)malloc(sizeof(statusptr[0]) * batchCount);
    memset(statusptr, 88, sizeof(statusptr[0]) * batchCount);
    if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds)
    {
        fprintf(stderr, "FAILED:\tmalloc data memory failed\n");
        goto exit;
    }
    for (i = 0; i < batchCount; i++)
    {
        ids[i] = i;
        sprintf(cols + 50 * i, "column test value %d", i);
        bufLenIds[i] = sizeof(ids[i]);
        bufLenCols[i] = strlen(cols + 50 * i);
        operptr[i] = (i < ignoreCount) ? SQL_PARAM_IGNORE : SQL_PARAM_PROCEED;
    }
}
```

```
// 分配语句句柄。
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtinesrt);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLAllocHandle failed");
    goto exit;
}

// 准备语句。
sql = (SQLCHAR*)"insert into test_odbc_batch_insert values(?, ?)";
retcode = SQLPrepare(hstmtinesrt, (SQLCHAR*) sql, SQL_NTS);
sprintf((char*)loginfo, "SQLPrepare log: %s", (char*)sql);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLPrepare failed");
    goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)batchCount,
sizeof(batchCount));

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLSetStmtAttr failed");
    goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,
sizeof(ids[0]), 0,&(ids[0]), 0, bufLenIds);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLBindParameter failed");
    goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50,
cols, 50, bufLenCols);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLBindParameter failed");
    goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&process,
sizeof(process));

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLSetStmtAttr failed");
    goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_STATUS_PTR, (SQLPOINTER)statusptr,
sizeof(statusptr[0]) * batchCount);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLSetStmtAttr failed");
    goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_OPERATION_PTR, (SQLPOINTER)operptr,
sizeof(operptr[0]) * batchCount);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLSetStmtAttr failed");
    goto exit;
}
retcode = SQLExecute(hstmtinesrt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);

if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLExecute(hstmtinesrt) failed");
    goto exit;
}
retcode = SQLRowCount(hstmtinesrt, &rowsCount);

if (!SQL_SUCCEEDED(retcode)) {
```

```
        printf("SQLRowCount failed");
        goto exit;
    }
    if (rowCount != (batchCount - ignoreCount))
    {
        sprintf(loginfo, "(batchCount - ignoreCount)(%d) != rowCount(%d)", (batchCount -
ignoreCount), rowCount);

        if (!SQL_SUCCEEDED(retcode)) {
            printf("SQLExecute failed");
            goto exit;
        }
    }
    else
    {
        sprintf(loginfo, "(batchCount - ignoreCount)(%d) == rowCount(%d)", (batchCount -
ignoreCount), rowCount);

        if (!SQL_SUCCEEDED(retcode)) {
            printf("SQLExecute failed");
            goto exit;
        }
    }

    // 检查返回的行数。
    if (rowCount != process)
    {
        sprintf(loginfo, "process(%d) != rowCount(%d)", process, rowCount);

        if (!SQL_SUCCEEDED(retcode)) {
            printf("SQLExecute failed");
            goto exit;
        }
    }
    else
    {
        sprintf(loginfo, "process(%d) == rowCount(%d)", process, rowCount);

        if (!SQL_SUCCEEDED(retcode)) {
            printf("SQLExecute failed");
            goto exit;
        }
    }
    for (i = 0; i < batchCount; i++)
    {
        if (i < ignoreCount)
        {
            if (statusptr[i] != SQL_PARAM_UNUSED)
            {
                sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_UNUSED", i, statusptr[i]);

                if (!SQL_SUCCEEDED(retcode)) {
                    printf("SQLExecute failed");
                    goto exit;
                }
            }
        }
        else if (statusptr[i] != SQL_PARAM_SUCCESS)
        {
            sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_SUCCESS", i, statusptr[i]);

            if (!SQL_SUCCEEDED(retcode)) {
                printf("SQLExecute failed");
                goto exit;
            }
        }
    }
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmtinesrt);
    sprintf((char*)loginfo, "SQLFreeHandle hstmtinesrt");
```

```
        if (!SQL_SUCCEEDED(retcode)) {  
            printf("SQLFreeHandle failed");  
            goto exit;  
        }  
    }  
}  
exit:  
(void) printf ("\nComplete.\n");  
  
// 断开连接。  
if (hdbc != SQL_NULL_HDBC) {  
    SQLDisconnect(hdbc);  
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);  
}  
// 释放环境句柄。  
if (henv != SQL_NULL_HENV)  
    SQLFreeHandle(SQL_HANDLE_ENV, henv);  
return 0;  
}
```

运行结果如下：

Complete.

数据库中查询后显示部分结果如下：

id	col
0	column test value 0
1	column test value 1
2	column test value 2
3	column test value 3
4	column test value 4
5	column test value 5
6	column test value 6
7	column test value 7
8	column test value 8
.....	

5.4.3.4 高性能绑定类型

需要进行大量数据插入时，有如下建议供用户参考和设置：

- 需要设置批量绑定：在odbc.ini配置文件中设置“UseBatchProtocol=1”，并配置数据库GUC参数“support_batch_bind=on”。
- ODBC程序的绑定类型需与数据库中类型一致。
- 客户端字符集需与数据库字符集一致。
- 事务需切换为手动提交模式。

odbc.ini配置文件：

```
[gaussdb]  
Driver=GaussMPP  
#数据库Server IP  
Servername=10.10.0.13  
...  
#默认打开  
UseBatchProtocol=1  
#设置客户端字符编码，保证和server端一致  
ConnSettings=set client_encoding=UTF8
```

绑定类型示例：

前提条件：数据源已配置成功。Linux系统请参考[Linux系统下配置数据源](#)；Windows系统请参考[Windows系统下配置数据源](#)。


```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqltext.h>
#include <string.h>
#include <sys/time.h>

#define MESSAGE_BUFFER_LEN 128
SQLHANDLE h_env = NULL;
SQLHANDLE h_conn = NULL;
SQLHANDLE h_stmt = NULL;
void print_error()
{
    SQLCHAR Sqlstate[SQL_SQLSTATE_SIZE+1];
    SQLINTEGER NativeError;
    SQLCHAR MessageText[MESSAGE_BUFFER_LEN];
    SQLSMALLINT TextLength;
    SQLRETURN ret = SQL_ERROR;

    ret = SQLGetDiagRec(SQL_HANDLE_STMT, h_stmt, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
    if ( SQL_SUCCESS == ret)
    {
        printf("\n STMT ERROR-%05d %s", NativeError, MessageText);
        return;
    }

    ret = SQLGetDiagRec(SQL_HANDLE_DBC, h_conn, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
    if ( SQL_SUCCESS == ret)
    {
        printf("\n CONN ERROR-%05d %s", NativeError, MessageText);
        return;
    }

    ret = SQLGetDiagRec(SQL_HANDLE_ENV, h_env, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
    if ( SQL_SUCCESS == ret)
    {
        printf("\n ENV ERROR-%05d %s", NativeError, MessageText);
        return;
    }

    return;
}

/* 期盼函数返回SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS(func) \
{\
    SQLRETURN ret_value = (func);\
    if (SQL_SUCCESS != ret_value)\
    {\
        print_error();\
        printf("\n failed line = %u: expect SQL_SUCCESS, but ret = %d", __LINE__, ret_value);\
        return SQL_ERROR; \
    }\
}

/* 期盼函数返回SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS_I(i, func) \
{\
    SQLRETURN ret_value = (func);\
    if (SQL_SUCCESS != ret_value)\
    {\
        print_error();\
        printf("\n failed line = %u (i=%d): : expect SQL_SUCCESS, but ret = %d", __LINE__, (i), ret_value);\
    }\
}
```

```
        return SQL_ERROR; \
    }\
}

/* 期盼函数返回SQL_SUCCESS_WITH_INFO */
#define RETURN_IF_NOT_SUCCESS_INFO(func) \
{\
    SQLRETURN ret_value = (func);\
    if (SQL_SUCCESS_WITH_INFO != ret_value)\
    {\
        print_error();\
        printf("\n failed line = %u: expect SQL_SUCCESS_WITH_INFO, but ret = %d", __LINE__, ret_value);\
        return SQL_ERROR; \
    }\
}

/* 期盼数值相等 */
#define RETURN_IF_NOT(expect, value) \
if ((expect) != (value))\
{\
    printf("\n failed line = %u: expect = %u, but value = %u", __LINE__, (expect), (value)); \
    return SQL_ERROR;\
}

/* 期盼字符串相同 */
#define RETURN_IF_NOT_STRCMP_I(i, expect, value) \
if (( NULL == (expect) ) || (NULL == (value)))\
{\
    printf("\n failed line = %u (i=%u): input NULL pointer !", __LINE__, (i)); \
    return SQL_ERROR; \
}\
else if (0 != strcmp((expect), (value)))\
{\
    printf("\n failed line = %u (i=%u): expect = %s, but value = %s", __LINE__, (i), (expect), (value)); \
    return SQL_ERROR;\
}

/* prepare + execute SQL语句 */
int execute_cmd(SQLCHAR *sql)
{
    if ( NULL == sql )
    {
        return SQL_ERROR;
    }

    if ( SQL_SUCCESS != SQLPrepare(h_stmt, sql, SQL_NTS))
    {
        return SQL_ERROR;
    }

    if ( SQL_SUCCESS != SQLExecute(h_stmt))
    {
        return SQL_ERROR;
    }

    return SQL_SUCCESS;
}

/* execute + commit 句柄 */
int commit_exec()
{
    if ( SQL_SUCCESS != SQLExecute(h_stmt))
    {
        return SQL_ERROR;
    }

    /* 手动提交 */
    if ( SQL_SUCCESS != SQLEndTran(SQL_HANDLE_DBC, h_conn, SQL_COMMIT))
    {

```

```
        return SQL_ERROR;
    }

    return SQL_SUCCESS;
}

int begin_unit_test()
{
    SQLINTEGER    ret;

    /* 申请环境句柄 */
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &h_env);
    if ((SQL_SUCCESS != ret) && (SQL_SUCCESS_WITH_INFO != ret))
    {
        printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_ENV failed ! ret = %d", ret);
        return SQL_ERROR;
    }

    /* 进行连接前必须要先设置版本号 */
    if (SQL_SUCCESS != SQLSetEnvAttr(h_env, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3,
0))
    {
        print_error();
        printf("\n begin_unit_test::SQLSetEnvAttr SQL_ATTR_ODBC_VERSION failed ! ret = %d", ret);
        SQLFreeHandle(SQL_HANDLE_ENV, h_env);
        return SQL_ERROR;
    }

    /* 申请连接句柄 */
    ret = SQLAllocHandle(SQL_HANDLE_DBC, h_env, &h_conn);
    if (SQL_SUCCESS != ret)
    {
        print_error();
        printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_DBC failed ! ret = %d", ret);
        SQLFreeHandle(SQL_HANDLE_ENV, h_env);
        return SQL_ERROR;
    }

    /* 建立连接 */
    ret = SQLConnect(h_conn, (SQLCHAR*) "gaussdb", SQL_NTS,
                    (SQLCHAR*) NULL, 0, NULL, 0);
    if (SQL_SUCCESS != ret)
    {
        print_error();
        printf("\n begin_unit_test::SQLConnect failed ! ret = %d", ret);
        SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
        SQLFreeHandle(SQL_HANDLE_ENV, h_env);
        return SQL_ERROR;
    }

    /* 申请语句句柄 */
    ret = SQLAllocHandle(SQL_HANDLE_STMT, h_conn, &h_stmt);
    if (SQL_SUCCESS != ret)
    {
        print_error();
        printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_STMT failed ! ret = %d", ret);
        SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
        SQLFreeHandle(SQL_HANDLE_ENV, h_env);
        return SQL_ERROR;
    }

    return SQL_SUCCESS;
}

void end_unit_test()
{
    /* 释放语句句柄 */
    if (NULL != h_stmt)
    {

```

```
    SQLFreeHandle(SQL_HANDLE_STMT, h_stmt);
}

/* 释放连接句柄 */
if (NULL != h_conn)
{
    SQLDisconnect(h_conn);
    SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
}

/* 释放环境句柄 */
if (NULL != h_env)
{
    SQLFreeHandle(SQL_HANDLE_ENV, h_env);
}

return;
}

int main()
{
    /* begin test */
    if (begin_unit_test() != SQL_SUCCESS)
    {
        printf("\n begin_test_unit failed.");
        return SQL_ERROR;
    }
    /* 句柄配置同前面用例 */
    int i = 0;
    SQLCHAR* sql_drop = "drop table if exists test_bindnumber_001";
    SQLCHAR* sql_create = "create table test_bindnumber_001("
        "f4 number, f5 number(10, 2)"
        ")";
    SQLCHAR* sql_insert = "insert into test_bindnumber_001 values(?, ?)";
    SQLCHAR* sql_select = "select * from test_bindnumber_001";
    SQLLEN RowCount;
    SQL_NUMERIC_STRUCT st_number;
    SQLCHAR getValue[2][MESSAGE_BUFFER_LEN];

    /* step 1. 建表 */
    RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
    RETURN_IF_NOT_SUCCESS(execute_cmd(sql_create));

    /* step 2.1 通过SQL_NUMERIC_STRUCT结构绑定参数 */
    RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));

    /* 第一行: 1234.5678 */
    memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
    st_number.precision = 8;
    st_number.scale = 4;
    st_number.sign = 1;
    st_number.val[0] = 0x4E;
    st_number.val[1] = 0x61;
    st_number.val[2] = 0xBC;

    RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_NUMERIC,
        SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0, NULL));
    RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_NUMERIC,
        SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0, NULL));

    /* 关闭自动提交 */
    SQLSetConnectAttr(h_conn, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0);

    RETURN_IF_NOT_SUCCESS(commit_exec());
    RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
    RETURN_IF_NOT(1, RowCount);

    /* 第二行: 12345678 */
    memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
```

```
st_number.precision = 8;
st_number.scale = 0;
st_number.sign = 1;
st_number.val[0] = 0x4E;
st_number.val[1] = 0x61;
st_number.val[2] = 0xBC;

RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_NUMERIC,
SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_NUMERIC,
SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* 第三行: 12345678 */
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 0;
st_number.scale = 4;
st_number.sign = 1;
st_number.val[0] = 0x4E;
st_number.val[1] = 0x61;
st_number.val[2] = 0xBC;

RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_NUMERIC,
SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_NUMERIC,
SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.2 第四行通过SQL_C_CHAR字符串绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
SQLCHAR* szNumber = "1234.5678";
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.3 第五行通过SQL_C_FLOAT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
SQLREAL fNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_FLOAT,
SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_FLOAT,
SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.4 第六行通过SQL_C_DOUBLE绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
SQLDOUBLE dNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_DOUBLE,
SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_DOUBLE,
SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

SQLBIGINT bNumber1 = 0xFFFFFFFFFFFFFFFF;
SQLBIGINT bNumber2 = 12345;
```

```
/* step 2.5 第七行通过SQL_C_SBIGINT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_SBIGINT,
SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_SBIGINT,
SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.6 第八行通过SQL_C_UBIGINT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

SQLLEN lNumber1 = 0xFFFFFFFFFFFFFFFF;
SQLLEN lNumber2 = 12345;

/* step 2.7 第九行通过SQL_C_LONG绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.8 第十行通过SQL_C_ULONG绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG,
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_ULONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

SQLSMALLINT sNumber = 0xFFFF;

/* step 2.9 第十一行通过SQL_C_SHORT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.10 第十二行通过SQL_C_USHORT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

SQLCHAR cNumber = 0xFF;

/* step 2.11 第十三行通过SQL_C_TINYINT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
```

```
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_TINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_TINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* step 2.12 第十四行通过SQL_C_UTINYINT绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT, SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT, SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);

/* 用字符串类型统一进行期望 */
SQLCHAR* expectValue[14][2] = {"1234.5678", "1234.57"},
{"12345678", "12345678"},
{"0", "0"},
{"1234.5678", "1234.57"},
{"1234.5677", "1234.57"},
{"1234.5678", "1234.57"},
{"-1", "12345"},
{"18446744073709551615", "12345"},
{"-1", "12345"},
{"4294967295", "12345"},
{"-1", "-1"},
{"65535", "65535"},
{"-1", "-1"},
{"255", "255"},
};

RETURN_IF_NOT_SUCCESS(execute_cmd(sql_select));
while ( SQL_NO_DATA != SQLFetch(h_stmt))
{
    RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 1, SQL_C_CHAR, &getValue[0],
MESSAGE_BUFFER_LEN, NULL));
    RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 2, SQL_C_CHAR, &getValue[1],
MESSAGE_BUFFER_LEN, NULL));
    i++;
}
printf("\nComplete!\n");
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount));
RETURN_IF_NOT(1, RowCount);
SQLCloseCursor(h_stmt);
/* step final. 删除表还原环境 */
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
RETURN_IF_NOT_SUCCESS(commit_exec());
end_unit_test();
}
```

运行结果如下：

```
Complete!
```

5.4.3.5 开启连接池

前提条件：数据源已配置成功。Linux系统请参考[Linux系统下配置数据源](#)；Windows系统请参考[Windows系统下配置数据源](#)。

在Linux和Windows环境下开启连接池，对于需要频繁建立和断开连接的中间层应用程序可显著提升性能，参考配置请参见[连接池场景](#)，示例代码如下：

```
#ifdef WIN32
#include <windows.h>
```

```
#endif

#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
#include <sys/time.h>
#include <pthread.h>
#include <sqltypes.h>
#include <time.h>
SQLHENV env;
SQLHDBC conn;
struct timeval start, end;

#define CONN_COUNT 15000
#define CHECK_ERROR(retcode, str, handle, htype) \
    ({ \
        if (retcode != SQL_SUCCESS && retcode != SQL_SUCCESS_WITH_INFO) { \
            fprintf(stderr, "FAILED:\t"); \
            extract_error(str, handle, htype); \
            exit(-1); \
        } else { \
            printf("OK:\t%s\n", str); \
        } \
    })

void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;

    if (msg) {
        printf("%s\n", msg);
    }

    ret = SQLGetDiagRec(htype, handle, 1, sqlstate, &nativeerror, message, 256, &textlen);

    if (ret != SQL_ERROR) {
        printf("%s=%s\n", (CHAR *)sqlstate, (CHAR *)message);
    }
}

void extract_error(char *fn, SQLHANDLE handle, SQLSMALLINT type)
{
    SQLINTEGER i = 0;
    SQLINTEGER NativeError;
    SQLCHAR SQLState[7];
    SQLCHAR MessageText[256];
    SQLSMALLINT TextLength;
    SQLRETURN ret;

    fprintf(stderr, "The driver reported the following error %s\n", fn);
    if (NULL == handle)
        return;
    do {
        ret = SQLGetDiagRec(type, handle, ++i, SQLState, &NativeError, MessageText, sizeof(MessageText), &TextLength);
        if (SQL_SUCCEEDED(ret)) {
            printf("[SQLState:%s]:[%ldth error]:[NativeError:%ld]: %s\n",
                SQLState,
                (long)i,
                (long)NativeError,
                MessageText);
        }
    } while (ret == SQL_SUCCESS);
}
```



```
}

void InitializeEnvironment()
{
    /* Windows配置连接池参数，需要在分配环境句柄前完成，此处为开启连接池 */
    SQLSetEnvAttr(env, SQL_ATTR_CONNECTION_POOLING, (SQLINTEGER *)SQL_CP_ONE_PER_DRIVER, 0);

    /* 在Win上关闭连接池 */
    // SQLSetEnvAttr(env, SQL_ATTR_CONNECTION_POOLING, (SQLINTEGER *)SQL_CP_OFF, 0);

    // 分配环境句柄
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    // 配置ODBC版本
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3, 0);
    // 配置建连超时时间
    SQLSetConnectAttr(conn, SQL_LOGIN_TIMEOUT, (SQLPOINTER *)5, 0);
}

void test_connect()
{
    SQLRETURN ret;
    SQLCHAR str[1024];
    SQLSMALLINT strl;
    SQLCHAR dsn[1024];
    SQLINTEGER ulntVal;

    SQLAllocHandle(SQL_HANDLE_DBC, env, &conn);

    /* 根据场景需要调整连接串 */
    char *config = "Driver=GaussMPP;DSN=gaussdb;";
    ret = SQLSetConnectAttr(conn, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)(1), 0);

    ret = SQLDriverConnect(conn, 0, (SQLCHAR *)config, SQL_NTS, (SQLCHAR *)NULL, SQL_NTS, 0,
SQL_DRIVER_NOPROMPT);

    if (SQL_SUCCEEDED(ret)) {
        // printf("Connected\n");
    } else {
        print_diag("SQLDriverConnect failed.", SQL_HANDLE_DBC, conn);
        SQLFreeHandle(SQL_HANDLE_DBC, conn);
        SQLFreeHandle(SQL_HANDLE_ENV, env);
        exit(1);
    }

    /* 将连接放入连接池以重用连接 */
    if (conn != SQL_NULL_HDBC) {
        SQLDisconnect(conn);
        SQLFreeHandle(SQL_HANDLE_DBC, conn);
        conn = SQL_NULL_HDBC;
    }
}

int main()
{
    int count = 0;
    int timeuser;
    gettimeofday(&start, NULL);
    InitializeEnvironment();

    for (int i = 0; i < CONN_COUNT; i++) {
        test_connect();
        count++;
    }

    //释放环境句柄
    SQLFreeHandle(SQL_HANDLE_ENV, env);
    printf("Connection count: %d\n", count);
    gettimeofday(&end, NULL);
}
```

```
timeuser = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;  
printf("Connection time: %.3f s \n", (double)timeuser / 1000000);  
return 0;  
}
```

不同环境下结果不一，在连接池打开的情况下，本示例运行结果如下：

```
Connection count: 15000  
Connection time: 14.175 s
```

在连接池关闭的情况下，本示例运行结果如下：

```
Connection count: 15000  
Connection time: 691.768 s
```

说明

Windows环境和Linux环境的应用代码实现方式相同，但连接字符串需根据具体场景进行配置。建议在跨平台部署时注意数据库连接参数的适配性。

5.4.4 ODBC 接口参考

ODBC接口是一套提供给用户的API函数，本节将对部分常用接口做具体描述，若涉及其他接口可参考msdn（网址：[https://msdn.microsoft.com/en-us/library/windows/desktop/ms714177\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms714177(v=vs.85).aspx)）中ODBC Programmer's Reference项的相关内容。

5.4.4.1 SQLAllocEnv

在ODBC 3.x版本中，ODBC 2.x的函数SQLAllocEnv已被SQLAllocHandle代替。

有关详细信息请参见[SQLAllocHandle](#)。

5.4.4.2 SQLAllocConnect

在ODBC 3.x版本中，ODBC 2.x的函数SQLAllocConnect已被SQLAllocHandle代替。

有关详细信息请参见[SQLAllocHandle](#)。

5.4.4.3 SQLAllocHandle

功能描述

分配环境、连接、语句或描述符的句柄，它替代了ODBC 2.x函数SQLAllocEnv、SQLAllocConnect及SQLAllocStmt。

原型

```
SQLRETURN SQLAllocHandle(SQLSMALLINT HandleType,  
                          SQLHANDLE InputHandle,  
                          SQLHANDLE *OutputHandlePtr);
```

参数

表 5-60 SQLAllocHandle 参数

关键字	参数说明
HandleType	由SQLAllocHandle分配的句柄类型。必须为下列值之一： - SQL_HANDLE_ENV（环境句柄） - SQL_HANDLE_DBC（连接句柄） - SQL_HANDLE_STMT（语句句柄） - SQL_HANDLE_DESC（描述句柄） 申请句柄顺序为，先申请环境句柄，再申请连接句柄，最后申请语句句柄。后申请的句柄依赖它前面申请的句柄。
InputHandle	将要分配的新句柄的类型。InputHandle参数用于指定创建句柄的父句柄，以建立句柄之间的层次关系。不同类型的句柄有不同的层次关系，InputHandle参数用于指定这种关系。 - 如果HandleType为SQL_HANDLE_ENV，则该值为SQL_NULL_HANDLE，表示没有父句柄。 - 如果HandleType为SQL_HANDLE_DBC，则它一定是一个环境句柄，表示该连接句柄是在该环境下创建的。 - 如果HandleType为SQL_HANDLE_STMT或SQL_HANDLE_DESC，则它一定是一个连接句柄，表示该语句句柄是在该连接下创建的。
OutputHandlePtr	输出参数： OutputHandlePtr是一个指向SQLHANDLE类型的指针，用于返回分配的新句柄。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当分配的句柄并非环境句柄时，如果SQLAllocHandle返回的值为SQL_ERROR，则它会将OutputHandlePtr的值设置为SQL_NULL_HDBC、SQL_NULL_HSTMT或SQL_NULL_HDESC。之后，通过调用带有适当参数的[SQLGetDiagRec](#)，其中HandleType和Handle被设置为InputHandle的值，可得到相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.4 SQLAllocStmt

在ODBC 3.x版本中，ODBC 2.x的函数SQLAllocStmt已被SQLAllocHandle代替。
有关详细信息请参见[SQLAllocHandle](#)。

5.4.4.5 SQLBindCol

功能描述

将应用程序数据缓冲区绑定到结果集的列中。

原型

```
SQLRETURN SQLBindCol(SQLHSTMT StatementHandle,
                      SQLUSMALLINT ColumnNumber,
                      SQLSMALLINT TargetType,
                      SQLPOINTER TargetValuePtr,
                      SQLLEN BufferLength,
                      SQLLEN *StrLen_or_IndPtr);
```

参数

表 5-61 SQLBindCol 参数

关键字	参数说明
StatementHandle	语旬句柄。
ColumnNumber	要绑定结果集的列号。起始列号为0，以递增的顺序计算列号，第0列是书签列。若未设置书签页，则起始列号为1。
TargetType	缓冲区中C数据类型的标识符。
TargetValuePtr	输出参数： 指向与列绑定的数据缓冲区的指针。SQLFetch函数返回这个缓冲区中的数据。如果此参数为一个空指针，则StrLen_or_IndPtr是一个有效值。
BufferLength	TargetValuePtr指向缓冲区的长度，以字节为单位。
StrLen_or_IndPtr	输出参数： 缓冲区的长度或指示器指针。若为空值，则未使用任何长度或指示器值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLBindCol返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和

StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.6 SQLBindParameter

功能描述

将一条SQL语句中的一个参数标志和一个缓冲区绑定起来。

原型

```
SQLRETURN SQLBindParameter(SQLHSTMT StatementHandle,
                             SQLUSMALLINT ParameterNumber,
                             SQLSMALLINT InputOutputType,
                             SQLSMALLINT ValueType,
                             SQLSMALLINT ParameterType,
                             SQLULEN ColumnSize,
                             SQLSMALLINT DecimalDigits,
                             SQLPOINTER ParameterValuePtr,
                             SQLLEN BufferLength,
                             SQLLEN *StrLen_or_IndPtr);
```

参数

表 5-62 SQLBindParameter

关键词	参数说明
StatementHandle	语句句柄。
ParameterNumber	参数序号，起始为1，依次递增。
InputOutputType	输入输出参数类型。可以是SQL_PARAM_INPUT、SQL_PARAM_OUTPUT、SQL_PARAM_INPUT_OUTPUT。
ValueType	参数的C数据类型。可以是SQL_C_CHAR、SQL_C_LONG、SQL_C_DOUBLE等。
ParameterType	参数的SQL数据类型。可以是SQL_CHAR、SQL_INTEGER、SQL_DOUBLE等。
ColumnSize	相应参数标记的列或表达式的大小。
DecimalDigits	相应参数标记的列或表达式的十进制数字。
ParameterValuePtr	指向存储参数数据缓冲区的指针。
BufferLength	ParameterValuePtr指向缓冲区的长度，以字节为单位。
StrLen_or_IndPtr	缓冲区的长度或指示器指针。若为空值，则未使用任何长度或指示器值。

 说明

当绑定的字符类型数据中含'\0'字符且libpq的0字符参数supportZeroCharacter设置为1时，用户需要保证输入的字符串长度（StrLen_or_IndPtr）等于含'\0'字符的字符串的实际长度。当输入的字符串长度大于含'\0'字符的字符串实际长度时，此场景下若发生内存越界访问，会在libpq侧产生coredump；当输入的字符串长度小于含'\0'字符的字符串实际长度时，会按照用户输入的字符串长度对含'\0'字符的字符串进行截断。

当0字符参数supportZeroCharacter设置为1时，还需要保证客户端字符集和数据库字符集一致。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLBindParameter返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.7 SQLColAttribute

功能描述

返回结果集中一列的描述符信息。

原型

```
SQLRETURN SQLColAttribute(SQLHSTMT StatementHandle,
                           SQLUSMALLINT ColumnNumber,
                           SQLUSMALLINT FieldIdentifier,
                           SQLPOINTER CharacterAttributePtr,
                           SQLSMALLINT BufferLength,
                           SQLSMALLINT *StringLengthPtr,
                           SQLLEN *NumericAttributePtr);
```

参数

表 5-63 SQLColAttribute 参数

关键字	参数说明
StatementHandle	语旬句柄。
ColumnNumber	要检索字段的列号，起始为1，依次递增。
FieldIdentifier	IRD中ColumnNumber行的字段。

关键字	参数说明
CharacterAttributePtr	输出参数： 一个缓冲区指针，返回FieldIdentifier字段值。
BufferLength	- 如果FieldIdentifier是一个ODBC定义的字段，而且CharacterAttributePtr指向一个字符串或二进制缓冲区，则此参数为该缓冲区的长度。 - 如果FieldIdentifier是一个ODBC定义的字段，而且CharacterAttributePtr指向一个整数，则会忽略该字段。
StringLengthPtr	输出参数： 缓冲区指针，存放*CharacterAttributePtr中字符类型数据的字节总数，对于非字符类型，忽略BufferLength的值。
NumericAttributePtr	输出参数： 指向一个整型缓冲区的指针，返回IRD中ColumnNumber行FieldIdentifier字段的值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLColAttribute返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.8 SQLConnect

功能描述

在驱动程序和数据源之间建立连接。连接上数据源之后，可以通过连接句柄访问到所有有关连接数据源的信息，包括程序运行状态、事务处理状态和错误信息。

原型

```
SQLRETURN SQLConnect(SQLHDBC      ConnectionHandle,
                      SQLCHAR      *ServerName,
                      SQLSMALLINT   NameLength1,
                      SQLCHAR      *UserName,
                      SQLSMALLINT   NameLength2,
                      SQLCHAR      *Authentication,
                      SQLSMALLINT   NameLength3);
```

参数

表 5-64 SQLConnect 参数

关键字	参数说明
ConnectionHandle	连接句柄，通过SQLAllocHandle获得。
ServerName	要连接数据源的名称。
NameLength1	ServerName的长度。
UserName	数据源中数据库用户名。
NameLength2	UserName的长度。
Authentication	数据源中数据库用户密码。
NameLength3	Authentication的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当调用SQLConnect函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_DBC和ConnectionHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.9 SQLDisconnect

功能描述

关闭一个与特定连接句柄相关的连接。

原型

```
SQLRETURN SQLDisconnect(SQLHDBC ConnectionHandle);
```


参数

表 5-65 SQLDisconnect 参数

关键字	参数说明
ConnectionHandle	连接句柄，通过SQLAllocHandle获得。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当调用SQLDisconnect函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_DBC和ConnectionHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.10 SQLExecDirect

功能描述

使用参数的当前值，执行一条准备好的语句。对于一次只执行一条SQL语句，SQLExecDirect是最快的执行方式。

原型

```
SQLRETURN SQLExecDirect(SQLHSTMT StatementHandle,
                        SQLCHAR *StatementText,
                        SQLINTEGER TextLength);
```

参数

表 5-66 SQLExecDirect 参数

关键字	参数说明
StatementHandle	语句句柄，通过SQLAllocHandle获得。
StatementText	要执行的SQL语句，不支持一次执行多条语句。

关键字	参数说明
TextLength	StatementText的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_NEED_DATA：表示在执行SQL语句前没有提供足够的参数。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。
- SQL_NO_DATA：表示SQL语句不返回结果集。

注意事项

当调用SQLExecDirect函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用SQLGetDiagRec函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.11 SQLExecute

功能描述

如果语句中存在参数标记，SQLExecute函数使用参数标记参数的当前值，执行一条准备好的SQL语句。

原型

```
SQLRETURN SQLExecute(SQLHSTMT StatementHandle);
```

参数

表 5-67 SQLExecute 参数

关键字	参数说明
StatementHandle	要执行语句的句柄。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_NEED_DATA：表示在执行SQL语句前没有提供足够的参数。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_NO_DATA：表示SQL语句不返回结果集。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当SQLExecute函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，可通过调用[SQLGetDiagRec](#)函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.12 SQLFetch

功能描述

从结果集中取下一个行集的数据，并返回所有被绑定列的数据。

原型

```
SQLRETURN SQLFetch(SQLHSTMT StatementHandle);
```

参数

表 5-68 SQLFetch 参数

关键字	参数说明
StatementHandle	语句句柄，通过SQLAllocHandle获得。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_NO_DATA：表示SQL语句不返回结果集。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当调用SQLFetch函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用SQLGetDiagRec函数，并将HandleType和Handle参数设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.13 SQLFreeStmt

在ODBC 3.x版本中，ODBC 2.x的函数SQLFreeStmt已被SQLFreeHandle代替。
有关详细信息请参见[SQLFreeHandle](#)。

5.4.4.14 SQLFreeConnect

在ODBC 3.x版本中，ODBC 2.x的函数SQLFreeConnect已被SQLFreeHandle代替。
有关详细信息请参见[SQLFreeHandle](#)。

5.4.4.15 SQLFreeHandle

功能描述

释放与指定环境、连接、语句或描述符相关联的资源，它替代了ODBC 2.x函数SQLFreeEnv、SQLFreeConnect及SQLFreeStmt。

原型

```
SQLRETURN SQLFreeHandle(SQLSMALLINT HandleType,
                        SQLHANDLE Handle);
```

参数

表 5-69 SQLFreeHandle 参数

关键字	参数说明
HandleType	SQLFreeHandle要释放的句柄类型。必须为下列值之一： - SQL_HANDLE_ENV - SQL_HANDLE_DBC - SQL_HANDLE_STMT - SQL_HANDLE_DESC 如果HandleType不是其中之一，SQLFreeHandle返回SQL_INVALID_HANDLE。
Handle	要释放的句柄。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

如果SQLFreeHandle返回SQL_ERROR，句柄仍然有效。

示例

请参见：[典型应用开发示例](#)。

5.4.4.16 SQLFreeEnv

在ODBC 3.x版本中，ODBC 2.x的函数SQLFreeEnv已被SQLFreeHandle代替。
有关详细信息请参见[SQLFreeHandle](#)。

5.4.4.17 SQLPrepare

功能描述

准备一个将要使用的SQL语句。

需要注意的是，ODBC发送准备好的语句不支持内核复用计划，会导致每次执行都需要重新生成计划，CPU占用率高。如果业务对计划复用有需求，建议优先使用JDBC作为客户端。

原型

```
SQLRETURN SQLPrepare(SQLHSTMT StatementHandle,
                      SQLCHAR *StatementText,
                      SQLINTEGER TextLength);
```

参数

表 5-70 SQLPrepare 参数

关键字	参数说明
StatementHandle	语句句柄。
StatementText	SQL文本串。
TextLength	StatementText的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当SQLPrepare返回的值为SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用SQLGetDiagRec函数，并将HandleType和Handle参数分别设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.18 SQLGetData

功能描述

SQLGetData返回结果集中某一列的数据。可以多次调用它来部分检索不定长度的数据。

原型

```
SQLRETURN SQLGetData(SQLHSTMT StatementHandle,
                      SQLUSMALLINT Col_or_Param_Num,
                      SQLSMALLINT TargetType,
                      SQLPOINTER TargetValuePtr,
                      SQLLEN BufferLength,
                      SQLLEN *StrLen_or_IndPtr);
```

参数

表 5-71 SQLGetData 参数

关键字	参数说明
StatementHandle	语句句柄，通过SQLAllocHandle获得。
Col_or_Param_Nu m	要返回数据的列号。结果集的列按增序从1开始编号。书签列的列号为0。
TargetType	TargetValuePtr缓冲中的C数据类型的类型标识符。若TargetType为SQL_ASD_TYPE，驱动使用ARD中SQL_DESC_CONCISE_TYPE字段的类型标识符。若为SQL_C_DEFAULT，驱动根据源的SQL数据类型选择缺省的数据类型。
TargetValuePtr	输出参数： 指向返回数据所在缓冲区的指针。

关键字	参数说明
BufferLength	TargetValuePtr所指向缓冲区的长度。
StrLen_or_IndPtr	输出参数： 指向缓冲区的指针，在此缓冲区中返回长度或标识符的值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_NO_DATA：表示SQL语句不返回结果集。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当调用SQLGetData函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过调用SQLGetDiagRec函数，并将HandleType和Handle参数分别设置为SQL_HANDLE_STMT和StatementHandle，可得到一个相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.19 SQLGetDiagRec

功能描述

返回诊断记录的多个字段的当前值，其中诊断记录包含错误、警告及状态信息。

原型

```
SQLRETURN SQLGetDiagRec(SQLSMALLINT  HandleType,
                        SQLHANDLE      Handle,
                        SQLSMALLINT    RecNumber,
                        SQLCHAR         *SQLState,
                        SQLINTEGER      *NativeErrorPtr,
                        SQLCHAR         *MessageText,
                        SQLSMALLINT     BufferLength
                        SQLSMALLINT     *TextLengthPtr);
```

参数

表 5-72 SQLGetDiagRec 参数

关键字	参数说明
HandleType	句柄类型标识符，它说明诊断所要求的句柄类型。必须为下列值之一： • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
Handle	诊断数据结构的句柄，其类型由HandleType来指出。如果HandleType是SQL_HANDLE_ENV，Handle可以是共享的或非共享的环境句柄。
RecNumber	指出应用从查找信息的状态记录。状态记录从1开始编号。
SQLState	输出参数： 指向缓冲区的指针，该缓冲区存储着有关RecNumber的五字符的SQLSTATE码。
NativeErrorPtr	输出参数： 指向缓冲区的指针，该缓冲区存储着本地的错误码。
MessageText	指向缓冲区的指针，该缓冲区存储着诊断信息文本串。
BufferLength	MessageText的长度。
TextLengthPtr	输出参数： 指向缓冲区的指针，返回MessageText中的字节总数。如果返回字节数大于BufferLength，则MessageText中的诊断信息文本被截断成BufferLength减去NULL结尾字符的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

SQLGetDiagRec不发布自己的诊断记录。用下列返回值来报告自己的执行结果：

- SQL_SUCCESS：函数成功返回诊断信息。
- SQL_SUCCESS_WITH_INFO：MessageText太小以致不能容纳所请求的诊断信息。没有诊断记录生成。
- SQL_INVALID_HANDLE：由HandleType和Handle所指出的句柄是不合法句柄。
- SQL_ERROR：RecNumber小于等于0或BufferLength小于0。

如果调用ODBC函数返回SQL_ERROR或SQL_SUCCESS_WITH_INFO，可调用SQLGetDiagRec返回诊断信息SQLSTATE值，如[表2 SQLSTATE值](#)所示。

表 5-73 SQLSTATE 值

SQLSTATE	错误	描述
HY000	一般错误	未定义特定的SQLSTATE所产生的错误。
HY001	内存分配错误	驱动程序不能分配所需要的内存来支持函数的执行或完成。
HY008	取消操作	调用SQLCancel取消执行语句后，依然在StatementHandle上调用函数。
HY010	函数系列错误	在为执行中的所有数据参数或列发送数据前就调用了执行函数。
HY013	内存管理错误	不能处理函数调用，可能由当前内存条件差引起。
HYT01	连接超时	数据源响应请求之前，连接超时。
IM001	驱动程序不支持此函数	调用了StatementHandle相关的驱动程序不支持的函数。

示例

请参见：[典型应用开发示例](#)。

5.4.4.20 SQLSetConnectAttr

功能描述

设置控制连接各方面的属性。

原型

```
SQLRETURN SQLSetConnectAttr(SQLHDBC ConnectionHandle,
                             SQLINTEGER Attribute,
                             SQLPOINTER ValuePtr,
                             SQLINTEGER StringLength);
```

参数

表 5-74 SQLSetConnectAttr 参数

关键字	参数说明
ConnectionHandle	连接句柄。
Attribute	设置属性。

关键字	参数说明
ValuePtr	指向对应Attribute的值。依赖于Attribute的值，ValuePtr是32位无符号整型值，或指向以空结束的字符串，二进制缓冲区，或者驱动定义值。如果ValuePtr参数是驱动程序指定值，ValuePtr可能是有符号的整数。
StringLength	如果ValuePtr指向字符串或二进制缓冲区，则这个参数是*ValuePtr长度，如果ValuePtr指向整型，忽略StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLSetConnectAttr的返回值为SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过借助SQL_HANDLE_DBC的HandleType和ConnectionHandle的Handle，调用[SQLGetDiagRec](#)可得到相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.21 SQLSetEnvAttr

功能描述

设置控制环境各方面的属性。

原型

```
SQLRETURN SQLSetEnvAttr(SQLHENV EnvironmentHandle,
                        SQLINTEGER Attribute,
                        SQLPOINTER ValuePtr,
                        SQLINTEGER StringLength);
```

参数

表 5-75 SQLSetEnvAttr 参数

关键字	参数说明
EnvironmentHandle	环境句柄。

关键字	参数说明
Attribute	需设置的环境属性，可为如下值： - SQL_ATTR_ODBC_VERSION：指定ODBC版本。 - SQL_CONNECTION_POOLING：连接池属性。 - SQL_OUTPUT_NTS：指明驱动器返回字符串的形式。
ValuePtr	指向对应Attribute的值。依赖于Attribute的值，ValuePtr可能是32位整型值，或为以空结束的字符串。
StringLength	如果ValuePtr指向字符串或二进制缓冲区，则这个参数是*ValuePtr长度，如果ValuePtr指向整型，忽略StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLSetEnvAttr的返回值为SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过借助SQL_HANDLE_ENV的HandleType和EnvironmentHandle的Handle，调用[SQLGetDiagRec](#)可得到相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

5.4.4.22 SQLSetStmtAttr

功能描述

设置相关语句的属性。

原型

```
SQLRETURN SQLSetStmtAttr(SQLHSTMT StatementHandle,
                           SQLINTEGER Attribute,
                           SQLPOINTER ValuePtr,
                           SQLINTEGER StringLength);
```

参数

表 5-76 SQLSetStmtAttr 参数

关键字	参数说明
StatementHandle	语句子柄。
Attribute	需设置的属性。
ValuePtr	指向对应Attribute的值。依赖于Attribute的值，ValuePtr可能是32位无符号整型值，或指向以空结束的字符串，二进制缓冲区，或者驱动定义值。如果ValuePtr参数是驱动程序指定值，ValuePtr可能是有符号的整数。
StringLength	如果ValuePtr指向字符串或二进制缓冲区，则这个参数是*ValuePtr长度，如果ValuePtr指向整型，忽略StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他API的返回值同理。

注意事项

当SQLSetStmtAttr的返回值为SQL_ERROR或SQL_SUCCESS_WITH_INFO时，通过借助SQL_HANDLE_STMT的HandleType和StatementHandle的Handle，调用[SQLGetDiagRec](#)可得到相关的SQLSTATE值，通过SQLSTATE值可以查出调用此函数的具体信息。

示例

请参见：[典型应用开发示例](#)。

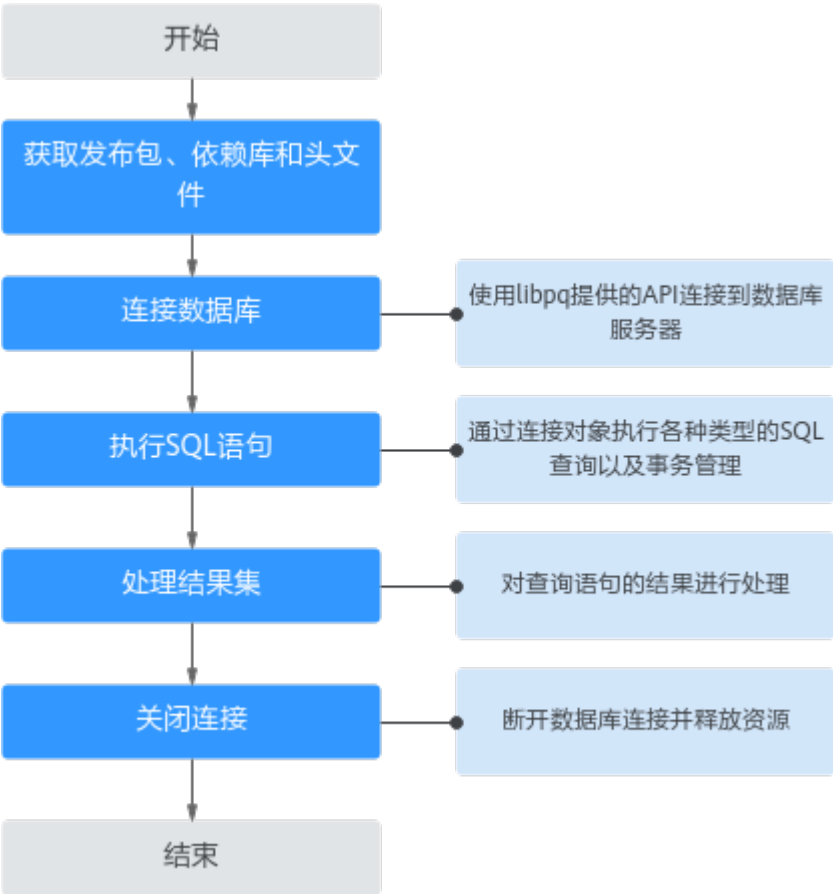
5.5 基于 libpq 开发

5.5.1 开发流程

libpq开发流程如[图5-17](#)所示。

其中包含获取发布包、依赖库和头文件、连接数据库、执行SQL语句、处理结果集、关闭连接五个部分。

图 5-17 libpq 开发流程图



5.5.2 开发步骤

5.5.2.1 获取发布包、依赖库和头文件

获取驱动包

下载表5-77中的驱动包和驱动包校验包。

表 5-77 驱动包下载列表

版本	下载地址
V2.0-10.x	驱动包 驱动包校验包

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：
 - a. 上传软件包和软件包校验包到虚拟机的同一目录下。

- b. 执行如下命令，校验软件包完整性。

```
F=$(echo DBS-GaussDB-driver*.tar.gz);echo "$(awk '{print $1}' "$F.sha256") $F" | sha256sum -c -
```

如果回显OK，则校验通过。例如：

```
DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz: OK
```

- Windows操作系统软件包完整性校验：

- 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。
- 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

- {驱动包本地目录}：请根据实际下载目录进行替换。例如：C:\Users
- {驱动包名}：请根据实际下载的驱动包名进行替换。例如：DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz

示例：certutil -hashfile C:\Users\DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz sha256

- 将**b**获取到的Hash值和**表5-77**中获取到的驱动包校验包的Hash值进行比较。
 - 若一致则通过校验。
 - 若不一致，请重新下载驱动包，重复**a~c**进行校验。

libpq依赖的库和头文件从驱动包中获取，驱动包中提供多种数据库内核操作系统支持，用户可根据实际使用的数据库内核操作系统信息定位对应文件。驱动包的文件层级关系参考以下结构：/Distributed/驱动类型/操作系统信息（例如：Euler2.10_X86_64）。该文件中包含2个压缩文件：GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Libpq.tar.gz、GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Libpq_Static.tar.gz。

Libpq_Static驱动包与Libpq驱动包在功能上一致，用户可根据实际应用场景选择适合的libpq.so库，区别在于：

- Libpq驱动包中的libpq.so动态链接openssl库(libssl.so、libcrypto.so)。
- Libpq_Static驱动包中的libpq.so静态链接openssl库，以解决在部分客户场景中与操作系统openssl版本不一致导致冲突的问题。

将以上压缩包解压后即可得到所需要的include、lib文件。其中include文件夹中的头文件为所需的头文件，lib文件夹中为所需的libpq库文件。使用libpq的程序必须包括头文件“libpq-fe.h”并且必须与lib目录下的libpq库进行连接。

说明

除“libpq-fe.h”外，include文件夹下默认还存在头文件“postgres_ext.h”、“gs_thread.h”、“gs_threadlocal.h”，这三个头文件是“libpq-fe.h”的依赖文件。

libpq开发需要在Linux操作系统下进行。推荐使用以下编译环境：

- 操作系统：推荐与数据库内核版本保持一致。
- 编译器：推荐GCC 7.3或更高版本。
- 构建工具：make、Cmake（推荐使用CMake 3.10及以上版本进行构建）。

在开始编译前，请确保上述工具已正确安装并可通过命令行正常调用。

5.5.2.2 连接数据库

连接到数据库是使用libpq开发应用程序的第一步。此时用户可以使用**PQconnectdb**或**PQsetdbLogin**函数来建立与数据库服务器的连接。

连接所用的连接串即conninfo有以下两种形式：

1. 所需的连接参数直接写为[name=value][...]的形式，多个参数之间使用空格隔开，例如：
"dbname=%s port=%s host=%s application_name=%s connect_timeout=%d sslmode=%s user=%s password=%s connectMode=%s tenantName=%s tenantPassword=%s"
2. 连接串写为postgresql://[userspec@][hostspec][:dbspec][?paramspec]的形式，其中userspec = user[:password]，hostspec = [host][:port][,...]，paramspec = name=value[&...]，例如：
"postgresql://HOST?dbname=postgres&port=PORT&user=USER&password=*****"

以开发源程序testlibpq.c为例（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

```
/*
 * 说明： testlibpq.c源程序，提供libpq基本且常见的使用场景。
 * 使用libpq提供的PQconnectdb、PQexec、PQntuples、PQfinish等接口实现数据库建连，执行sql，获取返回结果以及资源清理。
 * 头文件，自定义函数请参考完整示例。
 */

/* 此处user、passwd等变量应从环境变量或配置文件读取，环境变量需用户自己按需配置；非环境变量情况下可直接赋值字符串 */
const char conninfo[1024];
PGconn *conn;
PGresult *res;
int nFields;
int i,j;
char *passwd = getenv("EXAMPLE_PASSWD_ENV");
char *port = getenv("EXAMPLE_PORT_ENV");
char *host = getenv("EXAMPLE_HOST_ENV");
char *username = getenv("EXAMPLE_USERNAME_ENV");
char *dbname = getenv("EXAMPLE_DBNAME_ENV");

/*
 *tenant_conn_mode = getenv("EXAMPLE_TENANT_MODE_ENV");
 *tenant_username = getenv("EXAMPLE_TENANT_USERNAME_ENV");
 *tenant_passwd = getenv("EXAMPLE_TENANT_PASSWD_ENV");
 */

/*
 * 用户在命令行上提供了conninfo字符串的值时使用该值
 * 否则环境变量或者所有其它连接参数
 * 都使用缺省值。
 */
if (argc > 1)
    strcpy(conninfo, argv[1]);
else
    sprintf(conninfo,
        "dbname=%s port=%s host=%s application_name=test connect_timeout=5 sslmode=allow user=%s password=%s",
        dbname, port, host, username, passwd);

/* 连接数据库 */
conn = PQconnectdb(conninfo);

/* 检查后端连接成功建立 */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
        PQerrorMessage(conn));
    exit_nicely(conn);
}
```

注意事项：

- 敏感信息（密码等）不建议硬编码，应从环境变量、配置文件或密钥管理系统获取；
- 建立连接后立即通过PQstatus判断连接状态，并处理错误，提升程序健壮性。

5.5.2.3 执行 SQL 语句

通过连接对象，使用PQexec函数执行SQL查询语句。可以执行各种类型的SQL查询，例如SELECT查询、插入数据、更新数据、删除数据等。但如果同时执行多个SQL语句作为一个事务，应该使用事务控制功能，例如执行BEGIN、COMMIT、ROLLBACK 等SQL语句来控制事务的开始、提交和回滚。同时也要注意执行SQL查询后的错误处理。

示例如下（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

```
/*
 * 连接成功后
 * 测试实例涉及游标的使用时候必须使用事务块
 * 把全部放在一个 "select * from pg_database"
 * PQexec() 里，过于简单，不推荐使用
 */

/* 开始一个事务块 */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * 在结果不需要的时候PQclear PGresult，以避免内存泄漏
 */
PQclear(res);

/*
 * 从系统表 pg_database（数据库的系统目录）里抓取数据
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
```

注意事项：

- 所有执行函数返回的PGresult对象必须及时通过PQclear释放，否则会导致内存泄漏；
- 生产环境强烈推荐使用参数化查询或预处理语句，避免直接拼接SQL字符串；
- 避免重复释放：同一个PGresult只能释放一次。

5.5.2.4 处理结果集

libpq提供**PQnfields**、**PQntuples**、**PQfname**等函数来帮助用户对执行SELECT查询后的结果进行适当的解析和处理。主要步骤如下：

1. 判断执行结果状态；
2. 获取结果集行数和列数；
3. 遍历并提取数据；
4. 释放PGresult对象（必须执行）。

示例如下（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

```
/* 打印属性名称 */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n\n");

/* 打印行 */
for (i = 0; i < PQntuples(res); i++)
{
    for (j = 0; j < nFields; j++)
        printf("%-15s", PQgetvalue(res, i, j));
    printf("\n");
}

/* 释放结果对象的内存以避免内存泄漏 */
PQclear(res);
```

5.5.2.5 关闭连接

libpq通常使用**PQfinish**函数来关闭与数据库的连接。如果应用程序建立了多个连接，则需要确保每个连接都被正确关闭。

示例如下（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

```
/* 关闭入口 ... 不用检查错误 ... */
res = PQexec(conn, "CLOSE myportal");
PQclear(res);

/* 结束事务 */
res = PQexec(conn, "END");
PQclear(res);

/* 关闭数据库连接并清理 */
PQfinish(conn);
```

注意事项：

- 无论连接成功、失败还是发生异常，都必须确保PQfinish被调用，否则会导致连接句柄泄漏，长期运行可能耗尽系统文件描述符或数据库连接数；
- 先释放所有PGresult对象（调用PQclear），再关闭连接（调用PQfinish）；
- 多线程环境建议每个线程应使用独立的PGconn对象，关闭时需确保线程安全。

5.5.3 典型应用开发示例

5.5.3.1 数据库建连、执行 SQL 并返回结果

说明

gcc编译libpq源程序，需要通过-l *directory*选项，提供头文件的安装位置（有些时候编译器会查找缺省的目录，因此可以忽略这些选项）。如：

```
gcc -I (头文件所在目录) -L (libpq库所在目录) -o testlibpq testlibpq.c -lpq
```

执行命令为：

```
./testlibpq
```

如果要使用制作文件(makefile)，向CPPFLAGS、LDFLAGS、LIBS变量中增加如下选项：

```
CPPFLAGS += -I (头文件所在目录)
```

```
LDFLAGS += -L (libpq库所在目录)
```

```
LIBS += -lpq
```

例如：

```
CPPFLAGS += -I$(GAUSSHOME)/include/libpq
```

```
LDFLAGS += -L$(GAUSSHOME)/lib
```

非透明多写特性的示例如下：

```
/*
 * testlibpq.c
 * 说明： testlibpq.c源程序，提供libpq基本且常见的使用场景。
 * 使用libpq提供的PQconnectdb、PQexec、PQntuples、PQfinish等接口实现数据库建连，执行sql，获取返回结果以及资源清理。
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <string.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int main(int argc, char **argv)
{
    /* 此处user、passwd等变量应从环境变量或配置文件读取，环境变量需用户自己按需配置；非环境变量情况下可直接赋值字符串 */
    const char conninfo[1024];
    PGconn *conn;
    PGresult *res;
    int nFields;
    int i,j;
    char *passwd = getenv("EXAMPLE_PASSWD_ENV");
    char *port = getenv("EXAMPLE_PORT_ENV");
    char *host = getenv("EXAMPLE_HOST_ENV");
    char *username = getenv("EXAMPLE_USERNAME_ENV");
    char *dbname = getenv("EXAMPLE_DBNAME_ENV");

    /*
     * 用户在命令行上提供了conninfo字符串的值时使用该值
     * 否则环境变量或者所有其它连接参数
     * 都使用缺省值。
     */
    if (argc > 1)
        strcpy(conninfo, argv[1]);
    else
        sprintf(conninfo,
            "dbname=%s port=%s host=%s application_name=test connect_timeout=5 sslmode=allow user=%s password=%s",
            dbname, port, host, username, passwd);

    /* 连接数据库 */
```

```
conn = PQconnectdb(conninfo);

/* 检查后端连接成功建立 */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/*
 * 连接成功后
 * 测试实例涉及游标的使用时候必须使用事务块
 * 把全部放在一个 "select * from pg_database"
 * PQexec() 里，过于简单，不推荐使用
 */

/* 开始一个事务块 */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * 在结果不需要的时候PQclear PGresult，以避免内存泄漏
 */
PQclear(res);

/*
 * 从系统表 pg_database（数据库的系统目录）里抓取数据
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* 打印属性名称 */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n\n");

/* 打印行 */
for (i = 0; i < PQntuples(res); i++)
{
    for (j = 0; j < nFields; j++)
        printf("%-15s", PQgetvalue(res, i, j));
    printf("\n");
}

/* 释放结果对象的内存以避免内存泄漏 */
PQclear(res);

/* 关闭入口 ... 不用检查错误 ... */
```

```
res = PQexec(conn, "CLOSE myportal");
PQclear(res);

/* 结束事务 */
res = PQexec(conn, "END");
PQclear(res);

/* 关闭数据库连接并清理 */
PQfinish(conn);

return 0;
}
```

示例运行结果如下，其中“user_name”表示数据库管理员用户名，根据具体使用环境会发生变化：

datname	datdba	encoding	datcollate	datctype	datistemplate	datallowconn	
datconlimit	datlastsysoid	datfrozenxid	dattablespace	datcompatibilitydatacl	datfrozenxid64		
datminmxid	dattimezone	dattype					
template_pdb	10	7	en_US.UTF-8	en_US.UTF-8	t	t	-1
12837	0	1663	A	3	2	PRC	P
templatea	10	7	en_US.UTF-8	en_US.UTF-8	t	f	-1
12837	0	1663	A	{=c/user_name,user_name=CTc/user_name}	41372		
2	PRC	D					
template1	10	7	en_US.UTF-8	en_US.UTF-8	t	t	-1
12837	0	1663	A	{=c/user_name,user_name=CTc/user_name}	40414		
2	PRC	D					
templatem	10	7	en_US.UTF-8	en_US.UTF-8	t	t	-1
12837	0	1663	M	{=c/user_name,user_name=CTc/user_name}	55146		
2	PRC	D					
template0	10	7	en_US.UTF-8	en_US.UTF-8	t	f	-1
12837	0	1663	A	{=c/user_name,user_name=CTc/user_name}	39935		
2	PRC	D					
postgres	10	7	en_US.UTF-8	en_US.UTF-8	f	t	-1
12837	0	1663	A	40893	2	PRC	D

示例运行结果如下，根据具体使用环境会发生变化：

datname	datdba	encoding	datcollate	datctype	datistemplate	datallowconn
datconnlimit	datlastsysoid	datfrozenxid	dattablespace	datcompatibilitydatacl	datfrozenxid64	
datminmxid	dattimezone	dattype				
postgres	10	7	en_US.UTF-8	en_US.UTF-8	f	t
12837	0	1663	A	40893	2	PRC
						D

5.5.3.2 执行预备语句

以下示例创建一个给定参数的预备语句，用于PQexecPrepared执行预备语句：

非透明多写特性的示例如下：

```
/*
 * testlibpq2.c 测试PQprepare
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <string.h>
int main(int argc, char * argv[])
{
    /* 此处user、passwd等变量应从环境变量或配置文件读取，环境变量需用户自己按需配置；非环境变量情况下
    可直接赋值字符串 */
    PGconn *conn;
    PGresult *res;
    ConnStatusType pgstatus;
    char connstr[1024];
    char cmd_sql[2048];
    int nParams = 0;
```

```
int paramLengths[5];
int paramFormats[5];
Oid paramTypes[5];
char * paramValues[5];
int i, cnt;
char cid[32];
int k;
char *passwd = getenv("EXAMPLE_PASSWD_ENV");
char *port = getenv("EXAMPLE_PORT_ENV");
char *hostaddr = getenv("EXAMPLE_HOST_ENV");
char *username = getenv("EXAMPLE_USERNAME_ENV");
char *dbname = getenv("EXAMPLE_DBNAME_ENV");

/* PQconnectdb连接数据库, 详细的连接信息为connstr */
sprintf(connstr,
        "hostaddr=%s dbname=%s port=%s user=%s password=%s",
        hostaddr, dbname, port, username, passwd);

conn = PQconnectdb(connstr);
pgstatus = PQstatus(conn);
if (pgstatus == CONNECTION_OK)
{
    printf("Connect database success!\n");
}
else
{
    printf("Connect database fail:%s\n", PQerrorMessage(conn));
    return -1;
}

/* 创建表t01 */
res = PQexec(conn, "DROP TABLE IF EXISTS t01;CREATE TABLE t01(a int, b int);INSERT INTO t01
values(1, 23);");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    printf("Command failed: %s.\n", PQerrorMessage(conn));
    PQfinish(conn);
    return -1;
}

/* cmd_sql查询 */
sprintf(cmd_sql, "SELECT b FROM t01 WHERE a = $1");
/* cmd_sql中$1对应的参数 */
paramTypes[0] = 23;
/* PQprepare创建一个给定参数的预备语句 */
res = PQprepare(conn,
                "pre_name",
                cmd_sql,
                1,
                paramTypes);
if( PQresultStatus(res) != PGRES_COMMAND_OK )
{
    printf("Failed to prepare SQL : %s\n: %s\n",cmd_sql, PQerrorMessage(conn));
    PQfinish(conn);
    return -1;
}
PQclear(res);
paramValues[0] = cid;
for (k=0; k<2; k++)
{
    sprintf(cid, "%d", 1);
    paramLengths[0] = 6;
    paramFormats[0] = 0;
    /* 执行预备语句 */
    res = PQexecPrepared(conn,
                        "pre_name",
                        1,
                        paramValues,
                        paramLengths,
```

```
        paramFormats,
        0);
if( (PQresultStatus(res) != PGRES_COMMAND_OK ) && (PQresultStatus(res) != PGRES_TUPLES_OK))
{
    printf("%s\n", PQerrorMessage(conn));
    PQclear(res);
    PQfinish(conn);
    return -1;
}
cnt = PQntuples(res);
printf("return %d rows\n", cnt);
for (i=0; i<cnt; i++)
{
    printf("row %d: %s\n", i, PQgetvalue(res, i, 0));
}
PQclear(res);
}
/* 执行结束，关闭连接 */
PQfinish(conn);
return 0;
}
```

上述2个示例运行结果如下：

```
Connect database success!
NOTICE: table "t01" does not exist, skipping
return 1 rows
row 0: 23
return 1 rows
row 0: 23
```

5.5.3.3 绑定参数并返回二进制结果

以下示例执行一个绑定参数的命令，并以二进制格式请求查询结果。

```
CREATE TABLE test1 (i int4, t text);
INSERT INTO test1 values (2, 'ho there');
```

用以上命令填充数据库后，运行如下示例：

非透明多写特性的示例如下：

```
/*
 * testlibpq3.c
 * 测试PQexecParams
 *
 * 期望的输出如下
 *
 * tuple 0: got
 * i = (4 bytes) 2
 * t = (8 bytes) 'ho there'
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <libpq-fe.h>

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
/*
 * 这个函数打印查询结果，这些结果是二进制格式，从上面的
 * 注释里面创建的表中抓取出来的
 */
static void show_binary_results(PGresult *res)
{
    int i;
    int i_fnum, t_fnum;

    /* 使用 PQfnumber 来避免对结果中的字段顺序进行假设 */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");

    for (i = 0; i < PQntuples(res); i++)
    {
        char *iptr;
        char *tptr;
        int ival;

        /* 获取字段值（忽略可能为空的可能） */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);

        /*
         * INT4 的二进制表现形式是网络字节序
         * 建议转换成本地字节序
         */
        ival = ntohl(*(uint32_t *) iptr);

        /*
         * TEXT 的二进制表现形式是文本，因此libpq能够给它附加一个字节零
         * 把它看做 C 字符串
         */

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
            PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
            PQgetlength(res, i, t_fnum), tptr);
        printf("\n\n");
    }
}

int main(int argc, char **argv)
{
    /* 此处user、passwd等变量应从环境变量或配置文件读取，环境变量需用户自己按需配置；非环境变量情况下
    可直接赋值字符串 */
    const char conninfo[1024];
    PGconn *conn;
    PGresult *res;
    const char *paramValues[1];
    int paramLengths[1];
    int paramFormats[1];
    uint32_t binaryIntVal;
    char *passwd = getenv("EXAMPLE_PASSWD_ENV");
    char *port = getenv("EXAMPLE_PORT_ENV");
    char *hostaddr = getenv("EXAMPLE_HOST_ENV");
    char *username = getenv("EXAMPLE_USERNAME_ENV");
    char *dbname = getenv("EXAMPLE_DBNAME_ENV");

    /*
     * 如果用户在命令行上提供了参数，
     * 那么使用该值为conninfo 字符串；否则
     * 使用环境变量或者缺省值。
     */
    if (argc > 1)
        strcpy(conninfo, argv[1]);
    else
```

```
    sprintf(conninfo,
        "dbname=%s port=%s host=%s application_name=test connect_timeout=5 sslmode=allow user=%s"
        "password=%s",
        dbname, port, hostaddr, username, passwd);

/* 和数据库建立连接 */
conn = PQconnectdb(conninfo);

/* 检查与服务器的连接是否成功建立 */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
        PQerrorMessage(conn));
    exit_nicely(conn);
}

res = PQexec(conn, "drop table if exists test1;CREATE TABLE test1 (i int4, t text);");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

PQclear(res);

res = PQexec(conn, "INSERT INTO test1 values (2, 'ho there');");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

PQclear(res);

/* 把整数值 "2" 转换成网络字节序 */
binaryIntVal = htonl((uint32_t) 2);

/* 为 PQexecParams 设置参数数组 */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1; /* 二进制 */

/* PQexecParams执行一个绑定参数的命令 */
res = PQexecParams(conn,
    "SELECT * FROM test1 WHERE i = $1::int4",
    1, /* 一个参数 */
    NULL, /* 让后端推导参数类型 */
    paramValues,
    paramLengths,
    paramFormats,
    1); /* 要求二进制结果 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* 显示二进制结果 */
show_binary_results(res);

PQclear(res);

/* 关闭与数据库的连接并清理 */
PQfinish(conn);
```



```
    return 0;  
}
```

上述2个示例运行结果如下：

```
tuple 0: got  
i = (4 bytes) 2  
t = (8 bytes) 'ho there'
```

5.5.4 libpq 接口参考

须知

当前不支持通过libpq调用PQfn接口。

5.5.4.1 概述

libpq接口参考提供给用户一套数据库C语言客户端开发所需的核心API函数。这些函数按功能分类组织，涵盖了从连接建立、SQL执行、异步处理到结果集处理的完整开发流程。

开发者可根据实际需求查阅对应分类，快速定位所需函数。当前不支持通过libpq调用PQfn接口；本节将对部分常用接口做具体描述，若涉及其他接口可参考[链接](#)。libpq接口参考共分为以下6个主要部分：

- **数据库连接控制函数：**

数据库连接控制函数控制与数据库服务器的连接。一个应用程序一次可以与多个服务器建立连接，例如一个客户端连接多个数据库的场景。支持每个连接都是用一个从函数PQconnectdb、PQconnectdbParams或PQsetdbLogin获得的PGconn对象表示。也可以支持PQconnectStart接口结合异步PQconnectPoll轮询方式获得连接对象。注意，这些函数总是返回一个非空的对象指针，除非内存分配失败，会返回一个空的指针。连接建立的接口保存在PGconn对象中，可以调用PQstatus函数来检查返回值查看连接是否成功。由于PGconn对象将SSL上下文存储在线程本地，所以PGconn的释放线程应与申请线程一致。

- **数据库执行语句函数：**

与数据库服务器的连接成功建立，便可以使用该章节描述的函数执行SQL查询和命令。是libpq最常用的功能接口。

- **异步命令处理函数：**

异步命令处理函数用于实现非阻塞的SQL执行，适用于高并发、低延迟或需要同时处理多个命令的场景。该类函数允许应用程序在不阻塞当前线程的情况下发送SQL命令、轮询命令执行状态等。

- **取消查询处理中函数：**

取消查询处理中函数用于在查询执行过程中主动中断正在运行的SQL语句。该类函数主要用于长时间查询的超时控制、用户主动取消操作、异常情况下的快速中断等。

- **数据库结果处理函数：**

数据库结果处理函数用于解析和提取PQexec或异步执行返回的PGresult结果集。该类函数提供行数、列数获取、字段值提取与空值判断等功能。

以上各部分函数相互配合，共同构成libpq功能体系。

5.5.4.2 数据库连接控制函数

5.5.4.2.1 PQconnectdbParams

功能描述

与数据库服务器建立一个新的连接。

原型

```
PGconn* PQconnectdbParams(const char* const* keywords, const char* const* values, int expand_dbname);
```

参数

表 5-78 PQconnectdbParams 参数

关键字	参数说明
keywords	定义为一个字符串的数组，每个都成为一个关键字。
values	给每个关键字一个值。
expand_dbname	当expand_dbname非零时，允许将dbname的关键字值看做一个连接字符串。只有第一个出现的dbname是这样，随后的dbname值作为纯数据库名处理。

返回值

PGconn *：指向包含连接的对象指针，内存在函数内部申请。

注意事项

该函数用从两个NULL结束的数组中的参数打开一个新的数据库连接。与 PQsetdbLogin不同，该函数可以不必更换函数签名（名字）就可以扩展参数集，所以建议应用程序中使用该函数（或者类似的非阻塞变种PQconnectStartParams和 PQconnectPoll）。

5.5.4.2.2 PQconnectdb

功能描述

与数据库服务器建立一个新的连接。

原型

```
PGconn* PQconnectdb(const char* conninfo);
```

参数

表 5-79 PQconnectdb 参数

关键字	参数说明
conninfo	连接字符串，字符串中的字段请参见 连接参数说明 章节。

返回值

PGconn *：指向包含连接的对象指针，内存在函数内部申请。

注意事项

- 该函数用conninfo传入的参数打开一个新的连接。
- 传入的参数可以为空，表明使用所有缺省的参数，或者可以包含一个或更多个用空白间隔的参数设置，或者还可以包含一个URL。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.2.3 PQconninfoParse

功能描述

根据连接，返回已解析的连接选项。

原型

```
PQconninfoOption* PQconninfoParse(const char* conninfo, char** errmsg);
```

参数

表 5-80

关键字	参数说明
conninfo	被传递的字符串。可以为空，这样将会使用默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个URL。
errmsg	错误信息。

返回值

PQconninfoOption类型指针。

5.5.4.2.4 PQconnectStart

功能描述

与数据库服务器建立一次非阻塞的连接。

原型

```
PGconn* PQconnectStart(const char* conninfo);
```

参数

表 5-81

关键字	参数说明
conninfo	连接信息字符串。可以为空，这样将会使用默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个URL。

返回值

PGconn类型指针。

5.5.4.2.5 PQerrorMessage

功能描述

返回连接上的错误信息。

原型

```
char* PQerrorMessage(const PGconn* conn);
```

参数

表 5-82

关键字	参数说明
conn	连接句柄。

返回值

char类型指针。

5.5.4.2.6 PQsetdbLogin

功能描述

与数据库服务器建立一个新的连接。

原型

```
PGconn* PQsetdbLogin(const char* pghost, const char* pgport, const char* pgoptions, const char* pgtty,
const char* dbName, const char* login, const char* pwd);
```

参数

表 5-83 PQsetdbLogin 参数

关键字	参数说明
pghost	要连接的主机名，请参见 连接参数说明 描述的host字段。
pgport	主机服务器的端口号，请参见 连接参数说明 描述的port字段。
pgoptions	添加命令行选项以在运行时发送到服务器，请参见 连接参数说明 描述的options字段。
pgtty	该参数声明服务器日志的输出方向，当前设置不生效。
dbName	要连接的数据库名，请参见 连接参数说明 描述的dbname字段。
login	要连接的用户名，请参见 连接参数说明 描述的user字段。
pwd	如果服务器要求密码认证，所用的密码，请参见 连接参数说明 描述的password字段。

返回值

PGconn *: 指向包含连接的对象指针，内存在函数内部申请。

注意事项

- 该函数为PQconnectdb前身，参数个数固定，未定义参数被调用时使用缺省值，若需要给固定参数设置缺省值，则可赋值NULL或者空字符串。
- 若dbName中包含“=”或连接URL的有效前缀，则该dbName被看做一个conninfo字符串并传递至PQconnectdb中，其余参数与PQconnectdbParams保持一致。

5.5.4.2.7 PQfinish

功能描述

关闭与服务器的连接，同时释放被PGconn对象使用的存储器。

原型

```
void PQfinish(PGconn* conn);
```

参数

表 5-84 PQfinish 参数

关键字	参数说明
conn	指向包含连接的对象指针。

注意事项

若PQstatus判断服务器连接尝试失败，应用程序调用PQfinish释放被PGconn对象使用的存储器，PQfinish调用后PGconn指针不可再次使用。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.2.8 PQreset

功能描述

重置与服务器的通讯端口。

原型

```
void PQreset(PGconn* conn);
```

参数

表 5-85 PQreset 参数

关键字	参数说明
conn	指向包含连接的对象指针。

注意事项

此函数将关闭与服务器的连接并且试图与同一个服务器重建新的连接，并使用所有前面使用过的参数。该函数在连接异常后进行故障恢复时很有用。

5.5.4.2.9 PQstatus

功能描述

返回连接的状态。

原型

```
ConnStatusType PQstatus(const PGconn* conn);
```

参数

表 5-86 PQstatus 参数

关键字	参数说明
conn	指向包含连接的对象指针。

返回值

ConnStatusType：连接状态的枚举，包括：

CONNECTION_STARTED 等待进行连接。
CONNECTION_MADE 连接成功；等待发送。
CONNECTION_AWAITING_RESPONSE 等待来自服务器的响应。
CONNECTION_AUTH_OK 已收到认证；等待后端启动结束。
CONNECTION_SSL_STARTUP 协商SSL加密。
CONNECTION_SETENV 协商环境驱动的参数设置。
CONNECTION_OK 连接正常。
CONNECTION_BAD 连接故障。

注意事项

状态可以是多个值之一。但是，在异步连接过程之外只能看到其中两个：CONNECTION_OK和CONNECTION_BAD。与数据库的良好连接状态为CONNECTION_OK，与数据库连接失败状态为CONNECTION_BAD。通常，“正常”状态将一直保持到PQfinish，但通信失败可能会导致状态过早变为CONNECTION_BAD。在这种情况下，应用程序可以尝试通过调用进行恢复PQreset。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.3 数据库执行语句函数

5.5.4.3.1 PQexec

功能描述

向服务器提交一条命令并等待结果。

原型

```
PGresult* PQexec(PGconn* conn, const char* query);
```

参数

表 5-87 PQexec 参数

关键字	参数说明
conn	指向包含连接的对象指针。
query	需要执行的查询字符串。

返回值

PGresult：包含查询结果的对象指针。

注意事项

应该调用PQresultStatus函数来检查任何错误的返回值（包括空指针的值，在这种情况下它将返回PGRES_FATAL_ERROR）。使用PQerrorMessage获取有关错误的更多信息。

须知

命令字符串可以包括多个SQL命令（用分号分隔）。在一个PQexec调用中发送的多个查询是在一个事务里处理的，除非在查询字符串里有明确的BEGIN/COMMIT命令把整个字符串分隔成多个事务。请注意，返回的PGresult结构只描述字符串里执行的最后一条命令的结果，如果有一个命令失败，那么字符串处理的过程就会停止，并且返回的PGresult会描述错误条件。

PQexec函数对普通的同步应用里提交命令已经足够使用。但是它却有几个缺陷，而这些缺陷可能对某些用户很重要。

- PQexec等待命令结束，而应用可能还有其它的工作要做（比如维护用户界面等），此时并不希望PQexec阻塞应用。
- 因为客户端应用在等待结果的时候是处于挂起状态的，所以应用很难判断它是否该尝试结束正在进行的命令。
- PQexec只能返回一个PGresult结构。如果提交的命令字符串包含多个SQL命令，除了最后一个PGresult以外都会被PQexec丢弃。
- PQexec总是收集命令的整个结果，将其缓存在一个PGresult中。虽然这为应用简化了错误处理逻辑，但是对于包含多行的结果是不切实际的。

不想受到这些限制的应用可以改用如下函数，也是构造PQexec的函数：

PQsendQuery和PQgetResult。PQsendQueryParams、PQsendPrepare、PQsendQueryPrepared也可以和PQgetResult一起使用。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.3.2 PQprepare

功能描述

用给定的参数提交请求，创建一个预备语句，然后等待结束。

原型

```
PGresult* PQprepare(PGconn* conn, const char* stmtName, const char* query, int nParams, const Oid* paramTypes);
```

参数

表 5-88 PQprepare 参数

关键字	参数说明
conn	指向包含连接的对象指针。
stmtName	需要执行的prepare名称。
query	需要执行的查询字符串。
nParams	参数个数。
paramTypes	声明参数类型的数组。

返回值

PGresult：包含查询结果的对象指针。

注意事项

- 该函数创建一个为PQexecPrepared执行用的预备语句，支持命令的重复执行，不需要每次都进行解析和规划。
- PQprepare仅在协议3.0及以后的连接中支持，使用协议2.0时，PQprepare将失败。
- 该函数从查询字符串创建一个名为stmtName的预备语句，该查询字符串必须包含一个SQL命令。stmtName可以是""来创建一个未命名的语句，在这种情况下，任何预先存在的未命名的语句都将被自动替换，否则，如果在当前会话中已经定义了语句名称，那么就是一个错误。如果使用了任何参数，那么在查询中将它们称为\$1,\$2等。nParams是在paramTypes[]数组中预先指定类型的参数的数量。（当nParams为0时，数组指针可以为NULL），paramTypes[]通过OID指定要分配给参数符号的数据类型。如果paramTypes为NULL，或者数组中的任何特定元素为零，服务器将按照对非类型化字面字符串的相同方式为参数符号分配数据类型。另外，查询可以使用数字高于nParams的参数符号，还将推断这些符号的数据类型。

须知

通过执行SQLPREPARE语句，还可以创建与PQexecPrepared一起使用的预备语句。此外，虽然没有用于删除预备语句的libpq函数，但是SQL DEALLOCATE语句可用于此目的。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.3.3 PQclear

功能描述

释放与PGresult相关联的存储空间，任何不再需要的查询结果都应该用PQclear释放。

原型

```
void PQclear(PGresult* res);
```

参数

表 5-89 PQclear 参数

关键字	参数说明
res	包含查询结果的对象指针。

注意事项

PGresult不会自动释放，当提交新的查询时它并不消失，甚至断开连接后也不会。要删除它，必须调用PQclear，否则会有内存泄漏。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.3.4 PQfnumberEx

功能描述

返回与给定列名相关联的列号。列号从0开始。

原型

```
int PQfnumberEx(const PGresult* res, const char* field_name, bool case_sensitive);
```

参数

表 5-90 PQfnumberEx 参数

关键字	参数说明
res	操作结果句柄。
field_name	列名。
case_sensitive	列名是否大小写敏感。

返回值

int型整数。

5.5.4.3.5 PQexecParams

功能描述

执行一个绑定参数的命令。

原型

```
PGresult* PQexecParams(PGconn* conn,
    const char* command,
    int nParams,
    const Oid* paramTypes,
    const char* const* paramValues,
    const int* paramLengths,
    const int* paramFormats,
    int resultFormat);
```

参数

表 5-91 PQexecParams 参数

关键字	参数说明
conn	连接句柄。
command	SQL文本串。
nParams	绑定参数的个数。
paramTypes	绑定参数的类型。
paramValues	绑定参数的值。

关键字	参数说明
paramLengths	参数长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数格式（文本或二进制）。
resultFormat	返回结果格式（文本或二进制）。

返回值

PGresult类型指针。

5.5.4.3.6 PQexecParamsBatch

功能描述

执行一个批量绑定参数的命令。

原型

```
PGresult* PQexecParamsBatch(PGconn* conn,
                             const char* command,
                             int nParams,
                             int nBatch,
                             const Oid* paramTypes,
                             const char* const* paramValues,
                             const int* paramLengths,
                             const int* paramFormats,
                             int resultFormat);
```

参数

表 5-92 PQexecParamsBatch 参数

关键字	参数说明
conn	连接句柄。
command	SQL文本串。
nParams	绑定参数的个数。
nBatch	批量操作数。
paramTypes	绑定参数的类型。
paramValues	绑定参数的值。

关键字	参数说明
paramLengths	参数长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数格式（文本或二进制）。
resultFormat	返回结果格式（文本或二进制）。

返回值

PGresult类型指针。

5.5.4.3.7 PQexecPrepared

功能描述

发送一个请求来用给定参数执行一个预备语句，并且等待结果。

原型

```
PGresult* PQexecPrepared(PGconn* conn,
    const char* stmtName,
    int nParams,
    const char* const* paramValues,
    const int* paramLengths,
    const int* paramFormats,
    int resultFormat);
```

参数

表 5-93 PQexecPrepared 参数

关键字	参数说明
conn	连接句柄。
stmtName	stmt名称，可以用""或者NULL来引用未命名语句，否则它必须是一个现有预备语句的名字。
nParams	参数个数。
paramValues	参数的实际值。

关键字	参数说明
paramLengths	参数的实际数据长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数的格式（文本或二进制）。
resultFormat	结果的格式（文本或二进制）。

 说明

当绑定的字符类型数据中含'\0'字符且libpq的0字符参数supportZeroCharacter设置为1时，用户需要保证输入的字符串长度（paramLengths）等于含'\0'字符的字符串的实际长度。当输入的字符串长度大于含'\0'字符的字符串实际长度时，此场景下若发生内存越界访问，会在libpq侧产生coredump；当输入的字符串长度小于含'\0'字符的字符串实际长度时，会按照用户输入的字符串长度对含'\0'字符的字符串进行截断。

当0字符参数supportZeroCharacter设置为1时，还需要保证客户端字符集和数据库字符集一致。

返回值

PGresult类型指针。

5.5.4.3.8 PQexecPreparedBatch

功能描述

发送一个请求来用给定的批量参数执行一个预备语句，并且等待结果。

原型

```
PGresult* PQexecPreparedBatch(PGconn* conn,
                              const char* stmtName,
                              int nParams,
                              int nBatchCount,
                              const char* const* paramValues,
                              const int* paramLengths,
                              const int* paramFormats,
                              int resultFormat);
```

参数

表 5-94 PQexecPreparedBatch 参数

关键字	参数说明
conn	连接句柄。

关键字	参数说明
stmtName	<i>stmt</i> 名称，可以用""或者NULL来引用未命名语句，否则它必须是一个现有预备语句的名字。
nParams	参数个数。
nBatchCount	批量数。
paramValues	参数的实际值。
paramLengths	参数的实际数据长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数的格式（文本或二进制）。
resultFormat	结果的格式（文本或二进制）。

返回值

PGresult类型指针。

5.5.4.4 异步命令处理函数

5.5.4.4.1 PQsendQuery

功能描述

向服务器提交一个命令而不等待结果。如果查询成功发送则返回1，否则返回0。

原型

```
int PQsendQuery(PGconn* conn, const char* query);
```

参数

表 5-95 PQsendQuery 参数

关键字	参数说明
conn	指向包含连接的对象指针。
query	需要执行的查询字符串。

返回值

int：执行结果为1表示成功，0表示失败，失败原因存到conn->errorMessage中。

注意事项

在成功调用PQsendQuery后，调用PQgetResult一次或者多次获取结果。PQgetResult返回空指针表示命令已执行完成，否则不能再次调用PQsendQuery（在同一连接上）。

5.5.4.4.2 PQsendQueryParams

功能描述

给服务器提交一个命令和分隔的参数，而不等待结果。

原型

```
int PQsendQueryParams(PGconn* conn, const char* command, int nParams, const Oid* paramTypes,
    const char* const* paramValues, const int* paramLengths, const int* paramFormats, int resultFormat);
```

参数

表 5-96 PQsendQueryParams 参数

关键字	参数说明
conn	指向包含连接的对象指针。
command	需要执行的查询字符串。
nParams	参数个数。
paramTypes	参数类型。
paramValues	参数值。
paramLengths	参数长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数格式。
resultFormat	结果的格式。

返回值

int：执行结果为1表示成功，0表示失败，失败原因存到conn->errorMessage中。

注意事项

该函数等效于PQsendQuery，只是查询参数可以和查询字符串分开声明。函数的参数处理和PQexecParams类似，它不能在2.0版本的协议连接上工作，并且它只允许在查询字符串里出现一条命令。

5.5.4.4.3 PQsendPrepare

功能描述

发送一个请求，创建一个给定参数的预备语句，而不等待结束。

原型

```
int PQsendPrepare(PGconn* conn, const char* stmtName, const char* query, int nParams, const Oid* paramTypes);
```

参数

表 5-97 PQsendPrepare 参数

关键字	参数说明
conn	指向包含连接的对象指针。
stmtName	需要执行的prepare名称。
query	需要执行的查询字符串。
nParams	参数个数。
paramTypes	声明参数类型的数组。

返回值

int：执行结果为1表示成功，0表示失败，失败原因存到conn->errorMessage中。

注意事项

该函数为PQprepare的异步版本：如果能够分派请求，则返回1，否则返回0。调用成功后，调用PQgetResult判断服务端是否成功创建了preparedStatement。函数的参数与PQprepare一样处理。与PQprepare一样，它也不能在2.0协议的连接上工作。

5.5.4.4.4 PQsendQueryPrepared

功能描述

发送一个请求执行带有给出参数的预备语句，不等待结果。

原型

```
int PQsendQueryPrepared(PGconn* conn, const char* stmtName, int nParams, const char* const* paramValues, const int* paramLengths, const int* paramFormats, int resultFormat);
```

参数

表 5-98 PQsendQueryPrepared 参数

关键字	参数说明
conn	指向包含连接信息的对象指针。
stmtName	需要执行的prepare名称。
nParams	参数个数。
paramValues	参数值。
paramLengths	参数长度。 须知 下列情况中，参数长度强制使用paramLengths中对应的值： - 参数格式为二进制。 - 参数格式为文本且连接参数force_no_truncation或者supportZeroCharacter的值为1。 此时paramLengths不允许为空，并且参数长度的正确性须由调用者保证。
paramFormats	参数格式。
resultFormat	结果的格式。

返回值

int：执行结果为1表示成功，0表示失败，失败原因存到conn->errorMessage中。

注意事项

该函数类似于PQsendQueryParams，但是要执行的命令是通过命名一个预先准备的语句来指定的，而不是提供一个查询字符串。该函数的参数与PQexecPrepared一样处理。和PQexecPrepared一样，它也不能在2.0协议的连接上工作。

5.5.4.4.5 PQflush

功能描述

尝试将任何排队的输出数据刷新到服务器。

原型

```
int PQflush(PGconn* conn);
```

参数

表 5-99 PQflush 参数

关键字	参数说明
conn	指向包含连接信息的对象指针。

返回值

int: 如果成功（或者如果发送队列为空），则返回0；如果由于某种原因失败，则返回-1；如果发送队列中的所有数据都发送失败，则返回1。（此情况只有在连接为非阻塞时才能发生），失败原因存到conn->errorMessage中。

注意事项

在非阻塞连接上发送任何命令或数据之后，调用PQflush。如果返回1，则等待套接字变为读或写就绪。如果为写就绪状态，则再次调用PQflush。如果已经读到，调用PQconsumeInput，然后再次调用PQflush。重复，直到PQflush返回0。（必要检查读就绪并使用PQconsumeInput耗尽输入，因为服务器可能会阻止尝试向客户端发送数据（例如NOTICE消息），并且在客户端读取它的数据之前不会读取客户端的数据。）一旦PQflush返回0，等待套接字准备好，然后按照上面描述读取响应。

5.5.4.5 取消查询处理中函数

5.5.4.5.1 PQgetCancel

功能描述

创建一个数据结构，其中包含取消通过特定数据库连接发出的命令所需的信息。

原型

```
PGcancel* PQgetCancel(PGconn* conn);
```

参数

表 5-100 PQgetCancel 参数

关键字	参数说明
conn	指向包含连接信息的对象指针。

返回值

PGcancel: 指向包含cancel信息对象的指针。

注意事项

PQgetCancel创建一个给定PGconn连接对象的PGcancel对象。如果给定的conn是NULL或无效连接，它将返回NULL。PGcancel对象是一个不透明的结构，应用程序不能直接访问它，它只能传递给PQcancel或PQfreeCancel。

5.5.4.5.2 PQfreeCancel

功能描述

释放PQgetCancel创建的数据结构。

原型

```
void PQfreeCancel(PGcancel * cancel);
```

参数

表 5-101 PQfreeCancel 参数

关键字	参数说明
cancel	指向包含cancel信息的对象指针。

注意事项

PQfreeCancel释放一个由前面的PQgetCancel创建的数据对象。

5.5.4.5.3 PQcancel

功能描述

要求服务器放弃处理当前命令。

原型

```
int PQcancel(PGcancel* cancel, char* errbuf, int errbufsize);
```

参数

表 5-102 PQcancel 参数

关键字	参数说明
cancel	指向包含cancel信息的对象指针。
errbuf	出错时保存错误信息的buffer。
errbufsize	保存错误信息的buffer大小。

返回值

int：执行结果为1表示成功，0表示失败，失败原因存到errbuf中。

注意事项

- 成功发送并不保证请求将产生任何效果。如果取消有效，当前命令将提前终止并返回错误结果。如果取消失败（例如，因为服务器已经处理完命令），无返回结果。
- 如果errbuf是信号处理程序中的局部变量，则可以安全地从信号处理程序中调用PQcancel。就PQcancel而言，PGcancel对象是只读的，因此它也可以从一个线程中调用，这个线程与操作PGconn对象线程是分离的。

5.5.4.6 数据库结果处理函数

5.5.4.6.1 PQgetvalue

功能描述

返回一个PGresult的一行的单一阈值。行和列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

原型

```
char* PQgetvalue(const PGresult* res, int tup_num, int field_num);
```

参数

表 5-103 PQgetvalue 参数

关键字	参数说明
res	操作结果句柄。
tup_num	行数。
field_num	列数。

返回值

对于文本格式的数据，PQgetvalue返回的值是该阈值的一种空值结束的字符串表示。

对于二进制格式的数据，该值是由该数据类型的typsend和typreceive函数决定的二进制表示。

如果该阈值为空，则返回一个空串。

5.5.4.6.2 PQnfields

功能描述

返回查询结果中每一行的列（域）数。

原型

```
int PQnfields(const PGresult* res);
```

参数

表 5-104 PQnfields 参数

关键字	参数说明
res	操作结果句柄。

返回值

int类型数字。

5.5.4.6.3 PQntuples

功能描述

返回查询结果中的行（元组）数。

原型

```
int PQntuples(const PGresult* res);
```

参数

表 5-105 PQntuples 参数

关键字	参数说明
res	操作结果句柄。

须知

PQntuples返回一个整数结果，在 32 位操作系统上大型的结果集可能使返回值溢出。

返回值

int类型数字。

示例

请参见[典型应用开发示例](#)章节。

5.5.4.6.4 PQfname

功能描述

返回与给定列号相关联的列名。列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

原型

```
char* PQfname(const PGresult* res, int field_num);
```

参数

表 5-106 PQfname 参数

关键字	参数说明
res	操作结果句柄。
field_num	列数。

返回值

char类型指针。

5.5.4.6.5 PQresultStatus

功能描述

返回命令的结果状态。

原型

```
ExecStatusType PQresultStatus(const PGresult* res);
```

参数

表 5-107 PQresultStatus 参数

关键字	参数说明
res	包含查询结果的对象指针。

返回值

PQresultStatus：命令执行结果的枚举，包括：

PQresultStatus可以返回下面数值之一：
PGRES_EMPTY_QUERY
发送给服务器的字符串是空的。

PGRES_COMMAND_OK
成功完成一个不返回数据的命令。

PGRES_TUPLES_OK
成功执行一个返回数据的查询（比如SELECT或者SHOW）。

PGRES_COPY_OUT
（从服务器）Copy Out（复制出）数据传输开始。

PGRES_COPY_IN
Copy In（复制入）（到服务器）数据传输开始。

PGRES_BAD_RESPONSE
服务器的响应无法理解。

PGRES_NONFATAL_ERROR
发生了一个非致命错误（通知或者警告）。

PGRES_FATAL_ERROR
发生了一个致命错误。

PGRES_COPY_BOTH
复制入/出（到和从服务器）数据传输开始。这个特性当前只用于流复制，所以这个状态不会在普通应用中发生。

PGRES_SINGLE_TUPLE
PGresult包含一个来自当前命令的结果元组。这个状态只在查询选择了单行模式时发生。

注意事项

- 请注意，恰好检索到零行的SELECT命令仍然显示PGRES_TUPLES_OK。PGRES_COMMAND_OK用于永远不能返回行的命令（插入或更新，不带返回子句等）。PGRES_EMPTY_QUERY响应可能表明客户端软件存在bug。
- 状态为PGRES_NONFATAL_ERROR的结果永远不会由PQexec或其他查询执行函数直接返回，此类结果将传递给通知处理程序。

示例

请参见[典型应用开发示例](#)章节。

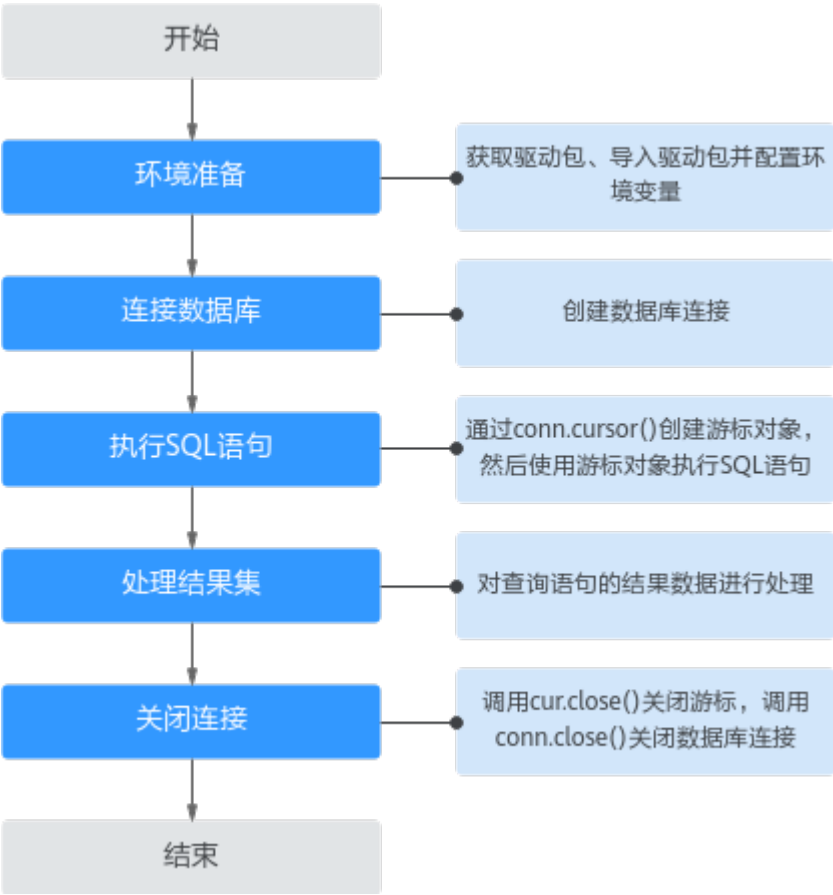
5.6 基于 Psycopg 开发

5.6.1 开发流程

Psycopg2开发应用程序的流程可分为环境准备、连接数据库、执行SQL语句、处理结果集、关闭连接等几个部分。

开发流程如[图5-18](#)所示。

图 5-18 Psycopg2 开发应用程序的流程



5.6.2 开发步骤

5.6.2.1 环境准备

获取驱动包

下载[表5-108](#)中的驱动包和驱动包校验包。

表 5-108 驱动包下载列表

版本	下载地址
V2.0-10.x	驱动包 驱动包校验包

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：
 - a. 上传软件包和软件包校验包到虚拟机的同一目录下。

- b. 执行如下命令，校验软件包完整性。

```
F=$(echo DBS-GaussDB-driver*.tar.gz);echo "$(awk '{print $1}' "$F.sha256") $F" | sha256sum -c -
```

如果回显OK，则校验通过。例如：

```
DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz: OK
```

- Windows操作系统软件包完整性校验：

- a. 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。
- b. 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

- {驱动包本地目录}**：请根据实际下载目录进行替换。例如：C:\Users

- {驱动包名}**：请根据实际下载的驱动包名进行替换。例如：DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz

示例：certutil -hashfile C:\Users\DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz sha256

- c. 将**b**获取到的Hash值和**表5-108**中获取到的驱动包校验包的Hash值进行比较。

- 若一致则通过校验。
- 若不一致，请重新下载驱动包，重复**a~c**进行校验。

获取 Python 驱动包

以下介绍了获取Python驱动包的步骤：

- 步骤1** 解压获取到的驱动包。该包包含多种不同的驱动包，可根据后续步骤确定所需驱动包的文件路径。驱动包的文件层级关系参考以下结构。

```
DBS-GaussDB-driver_*/数据库部署形态/驱动类型/操作系统架构/发布包/驱动包
```

- 步骤2** 选择数据库部署形态。

分布式部署形态选择Distributed。

- 步骤3** 选择驱动类型。

选择python_driver.tar.gz，并解压。

- 步骤4** 选择操作系统架构。

用户需根据客户端所在机器操作系统来选择对应操作系统架构的驱动包。

- 步骤5** 选择Python发布包。

- 步骤6** 解压Python发布包，则可获取两种格式的驱动包，分别为tar.gz格式和whl格式。这两种格式的驱动包功能相同，用户根据自身业务需求选择对应的驱动包。

- 非透明多写驱动包格式如下：

```
GaussDB-Kernel_数据库版本号_Python_CPU架构_Python版本_Distributed.tar.gz  
> GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Python.tar.gz  
> GaussDB-Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl
```

示例：

```
GaussDB-Kernel_507.0.0.B071_Python_X86_Py3.11_Distributed.tar.gz  
> GaussDB-Kernel_507.0.0_Euler_64bit_Python.tar.gz  
> GaussDB-Kernel_507_0_0-2.9.10-py311-none-linux_x86_64.whl
```

说明

- tar.gz格式驱动包，对其进行解压后包含两个文件夹：
 - psycpg2：表示python驱动的源码文件。
 - lib：表示python驱动的依赖库。
- whl格式驱动包安装完成后，在对应python环境的site-packages目录中生成以下两个文件夹：
 - psycpg2：表示python驱动的源码文件。
 - gaussdb_psycpg2.lib：表示python驱动的依赖库。
- whl格式驱动包命名规则说明：
 - abiTag：表示{abi tag}的信息，标记应用程序二进制接口（ABI）的标签（如：cp311或none）。
 - 支持的平台：表示{platform tag}的信息，标记支持的平台（如：linux_x86_64表示在Linux x86_64位系统上，any表示平台无关）。
- whl格式驱动包使用约束：安装whl格式python驱动包时，需保证环境上的python版本与whl格式驱动包名中的“Python版本”保持一致。
- 目前python驱动包仅支持python3.11版本。

----结束

安装前检查

安装psycpg2 Python驱动前，需检查环境上是否已安装开源psycpg2 Python驱动，如果已安装开源psycpg2 Python驱动则需要清理操作，否则会影响Python驱动的使用。请按以下步骤进行检查操作：

步骤1 切换至root用户，执行以下命令检查环境中是否已安装psycpg2 Python驱动。

```
ll $(python3 -c 'import site; print(site.getsitepackages()[0])') | grep psycpg2
```

如果存在输出对应的目录，则执行[下一步](#)，否则结束检查。

步骤2 执行以下命令，清理已经安装的Psycpg2驱动文件。

```
rm -rf $(python3 -c 'import site; print(site.getsitepackages()[0])')/psycpg2*
```

----结束

导入 tar.gz 格式驱动包并配置环境变量

步骤1 将“GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Python.tar.gz”驱动包导入到环境root根目录。

步骤2 解压驱动包。

```
tar zxvf xxxx_Python.tar.gz
```

步骤3 切换至root用户，将psycpg2驱动源码文件复制到python安装目录下的site-packages文件夹下。

```
su root  
cp psycpg2 $(python3 -c 'import site; print(site.getsitepackages()[0])') -r
```

须知

`$(python3 -c 'import site; print(site.getsitepackages()[0])')`命令的作用是获取python用于安装第三方包的site-packages目录主路径。当出现目录不存在的报错时，可通过`python3 -c "import site; print(site.getsitepackages())"`命令获取python site-packages目录的路径列表，从中找一个有效路径替代。

步骤4 修改psycpg2目录权限为755。

```
chmod 755 $(python3 -c 'import site; print(site.getsitepackages()[0])')/psycpg2 -R
```

步骤5 将psycpg2目录配置到环境变量PYTHONPATH中，并使之生效。

```
export PYTHONPATH=$(python3 -c 'import site; print(site.getsitepackages()[0])'):$PYTHONPATH
```

步骤6 对于非数据库用户，需要将解压后的lib目录的绝对路径[path-to-lib]配置到环境变量LD_LIBRARY_PATH。

```
export LD_LIBRARY_PATH=[path-to-lib]:$LD_LIBRARY_PATH
```

说明

对于数据库用户，则无需手动配置LD_LIBRARY_PATH。

----结束

导入 whl 格式驱动包并配置环境变量

须知

安装whl格式驱动包时，如果数据库已安装过“GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl”驱动包且未通过“pip3 uninstall”或“python3 -m pip uninstall”命令卸载驱动包，建议先通过“pip3 uninstall”或“python3 -m pip uninstall”命令卸载对应的驱动包。具体操作请参见[方式二](#)。

步骤1 将“GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl”驱动包导入到环境root根目录。

步骤2 切换至root用户，安装whl格式驱动包。

```
su root
```

环境支持pip命令时，执行如下命令：

```
pip3 install GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl
```

环境不支持pip命令时，执行如下命令：

```
python3 -m pip install GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl
```

步骤3 （可选）执行如下命令为非root用户赋予使用python驱动的权限。

```
chmod 755 $(python3 -c 'import site; print(site.getsitepackages()[0])')/psycpg2 -R
```

须知

`$(python3 -c 'import site; print(site.getsitepackages()[0])')`命令的作用是获取python用于安装第三方包的site-packages目录主路径。当出现目录不存在的报错时，可通过`python3 -c "import site; print(site.getsitepackages())"`命令获取python site-packages目录的路径列表，从中找一个有效路径替代。

步骤4 配置环境变量。

- 对于非数据库用户，需要将安装whl格式驱动包后的lib目录配置到环境变量LD_LIBRARY_PATH中。

```
lib_path=$(python3 -c 'import site; print(site.getsitepackages()[0]+'/gaussdb_psycomp2.lib'))
export LD_LIBRARY_PATH=$lib_path:$LD_LIBRARY_PATH
```

- 对于安装数据库用户，无需手动配置环境变量LD_LIBRARY_PATH。

----结束

（可选）卸载驱动包

步骤1 切换至root用户。

```
su root
```

步骤2 执行如下命令卸载python驱动。

- 方式一：推荐通过tar.gz格式驱动包安装python驱动方式时使用，通过whl格式驱动包安装python驱动也可使用，但推荐使用[方式二](#)。

```
site_package_path=$(python3 -c 'import site; print(site.getsitepackages()[0])')
rm -rf $site_package_path/psycpg2
```

须知

`site_package_path=$(python3 -c 'import site; print(site.getsitepackages()[0])')`命令的作用是获取python用于安装第三方包的site-packages目录主路径。当出现目录不存在的报错时，可通过`python3 -c "import site; print(site.getsitepackages())"`命令获取python site-packages目录的路径列表，从中找一个有效路径替代。

- 卸载whl格式驱动包时执行以下命令：

```
rm -rf $site_package_path/gaussdb_psycomp2.lib $site_package_path/GaussDB_Kernel_*.dist-info
```

- 卸载tar.gz格式驱动包时执行以下命令：

```
rm -rf [path-to-lib]
```

其中，[path-to-lib]表示安装tar.gz格式驱动包时，用户存放lib目录的位置。

- 方式二：仅适用于通过whl格式驱动包安装python驱动的方式时使用。

- 若通过pip install方式安装whl格式驱动包时，建议执行如下命令：

```
pip3 uninstall GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl
```

- 若通过python3 -m pip install方式安装whl格式驱动包时，建议执行如下命令：

```
python3 -m pip uninstall GaussDB_Kernel_数据库版本号-psycpg2版本号-Python版本-abiTag-支持的平台.whl
```

----结束

5.6.2.2 连接数据库

须知

- Python驱动版本≥2.9.10时，客户端将不再支持部分编码集（如ZHS16GBK、GB18030_2022），此时：
 - 连接参数client_encoding不可设置为这些编码集。
 - 无法连接使用这些编码集的数据库。
- 执行以下命令可以查看Python驱动版本号。

```
python3 -c "import psycopg2;print(psycopg2.__version__)"
```

5.6.2.2.1 连接介绍

通过psycopg2.connect函数建立数据库连接。函数的各参数说明，请参见[psycopg2.connect\(\)](#)。使用psycopg2.connect函数连接数据库时，可以使用dsn参数传入一个字符串，也可以使用**kwargs传入多个关键字参数。

- 方式1：psycopg2.connect(dsn)。该方式使用dsn参数传入一个字符串（包含数据库名称、用户名、密码、服务器地址、端口号），具体步骤请参见[使用dsn参数的连接方式](#)。
- 方式2：psycopg2.connect(**kwargs)。该方式使用**kwargs传入多个关键字参数（数据库名称、用户名、密码、服务器地址、端口号），具体步骤请参见[使用**kwargs的连接方式](#)。

5.6.2.2.2 以非加密方式连接

使用 dsn 参数的连接方式

调用psycopg2.connect(dsn)创建数据库连接，步骤如下：

步骤1 导入psycopg2和os模块。

```
import psycopg2
import os
```

步骤2 指定用户名和密码。建议在环境变量中存放用户名和密码，因为直接写到代码中有安全风险。

```
userName = os.getenv('EXAMPLE_USERNAME_ENV')
passWord = os.getenv('EXAMPLE_PASSWORD_ENV')
```

步骤3 指定host和port。建议在环境变量中存放host和port。

```
host= os.getenv('EXAMPLE_HOST_ENV')
port= os.getenv('EXAMPLE_PORT_ENV')
```

步骤4 调用psycopg2.connect(dsn)创建数据库连接。

- 非透明多写特性下，用户需要指定数据库名称、服务器地址以及端口号。

```
conn = psycopg2.connect(f'dbname={database} user={userName} password={passWord} host={host} port={port}')
```

----结束

使用**kwargs 的连接方式

调用psycopg2.connect(**kwargs)创建数据库连接，步骤如下：

步骤1 导入psycopg2和os模块。

```
import psycopg2
import os
```

步骤2 指定用户名和密码。建议在环境变量中存放用户名和密码，因为直接写到代码中有安全风险。

```
userName = os.getenv('EXAMPLE_USERNAME_ENV')
passWord = os.getenv('EXAMPLE_PASSWORD_ENV')
```

步骤3 指定host和port。建议在环境变量中存放host和port。

```
host = os.getenv('EXAMPLE_HOST_ENV')
port = os.getenv('EXAMPLE_PORT_ENV')
```

步骤4 调用psycopg2.connect(**kwargs)创建数据库连接。

- 非透明多写特性下，用户需要指定数据库名称、服务器地址以及端口号。
conn = psycopg2.connect(dbname="database", user=userName, password=passWord, host=host, port=port)

----结束

5.6.2.2.3 以 SSL 方式连接

通过psycopg2连接GaussDB数据库时，可以通过开启SSL加密客户端和服务端之间的通讯，为敏感数据传输提供一种安全保障手段。

前置条件

登录GaussDB管理控制台，在“实例管理”页面，单击实例名称进入“基本信息”页面，单击“SSL”处的 [↓](#)，下载根证书或捆绑包，将根证书ca.pem放置在客户端。

操作步骤

步骤1 导入psycopg2、ssl和os模块。

```
import psycopg2
import ssl
import os
```

步骤2 指定用户名和密码。建议在环境变量中存放用户名和密码，因为直接写到代码中有安全风险。

```
userName = os.getenv('EXAMPLE_USERNAME_ENV')
passWord = os.getenv('EXAMPLE_PASSWORD_ENV')
```

步骤3 指定host和port。建议在环境变量中存放host和port。

```
host = os.getenv('EXAMPLE_HOST_ENV')
port = os.getenv('EXAMPLE_PORT_ENV')
```

步骤4 sslmode设置为require、verify-ca、verify-full，就会启用SSL加密，客户根据应用场景选择一种即可。具体介绍请参见[表5-109](#)。

```
# 设置sslmode为require。
sslmode = "require"
# 设置sslmode为verify-ca。
sslmode = "verify-ca"
# 设置sslmode为verify-full。
sslmode = "verify-full"
```


表 5-109 sslmode 的可选项及其描述

sslmode	是否会启用SSL加密	描述
disable	否	只尝试非SSL连接。
allow	可能	首先尝试非SSL连接，如果连接失败，再尝试SSL连接。
prefer	可能	首先尝试SSL连接，如果连接失败，将尝试非SSL连接。
require	是	只尝试SSL连接。如果存在CA文件，则按设置成verify-ca的方式验证。
verify-ca	是	只尝试SSL连接，并且验证服务器是否具有由可信任的证书机构签发的证书。
verify-full	是	只尝试SSL连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。

步骤5 以SSL方式创建数据库连接。

- 非透明多写特性下，调用psycopg2.connect(**kwargs)，用户需要指定数据库名称、服务器地址、端口号，同时配置ca.pem根证书。
`conn = psycopg2.connect(dbname="database", user=username, password=password, host=host, port=port, sslmode=sslmode, sslrootcert="ca.pem")`

----结束

须知

- 以SSL方式连接数据库，建议使用python解释器通过生成动态链接库（.so）文件的方式编译，可通过如下步骤确认python解释器的编译方式。
 - 在python解释器命令行中输入import ssl，导入SSL。
 - 执行ps ux查询python解释器运行的pid（假设pid为*****）。
 - 在shell命令行中执行pmap -p ***** | grep ssl，查看返回结果中是否包含libssl.so的相关路径。如果有，则python解释器为动态链接方式编译。
- 若python解释器不是通过生成动态链接库（.so）文件的方式编译，则需要获取GaussDB-Kernel_数据库版本号_操作系统信息_64bit_Libpq_Static.tar.gz，解压后拷贝lib目录里的libpq.so.5，替换Python的lib目录里的libpq.so.5。

5.6.2.2.4 连接参数参考

常用的连接参数，如表5-110所示。

表 5-110 连接参数说明

参数名称	参数说明	参数值
dbname	数据库名称。	默认值：当前操作系统用户名。
user	用户名。	默认值：当前操作系统用户名。
password	密码。	默认值：NULL
host	指定要连接的主机名或者数据库IP地址。 非透明多写特性下：可指定多个主机名或IP地址，主机名或IP地址间以逗号隔开，默认为UNIX socket类型。	默认值：NULL
port	连接端口号。 非透明多写特性下，该参数为主机服务器的端口号。当host为多个IP地址时，如果端口号相同，则指定一个端口号即可；如果端口号不同，则端口号需要与IP地址一一对应，以逗号分隔。	默认值：5432
sslmode	指定SSL认证方式。	取值范围：disable、allow、prefer、require、verify-ca、verify-full。各取值的详细含义，请参见 表5-109 。 默认值：prefer
sslcert	SSL连接时，指定客户端证书路径。	默认值：NULL（表示读取用户根目录的客户端证书文件）
sslkey	SSL连接时，指定客户端密钥路径。	默认值：NULL（表示读取用户根目录的密钥文件）
sslrootcert	SSL连接时，指定根证书路径。	默认值：NULL（表示读取用户根目录的SSL根证书）
sslcompression	SSL连接时发送的数据是否被压缩。若不是以SSL方式连接，则忽略此参数。	取值范围： 0：表示SSL连接时发送的数据不被压缩。 1：表示SSL连接时发送的数据被压缩。 默认值：1

参数名称	参数说明	参数值
sslctrl	指定SSL证书吊销列表（CRL，Certificate Revocation List）的文件路径。若服务端证书存在于已配置的CRL中，客户端在SSL握手阶段将拒绝认证并终止连接。	默认值：NULL
hostaddr	数据库IP地址。只能指定IP地址来连接数据库，而不能通过主机名解析。 须知 同时指定host和hostaddr时，会优先使用hostaddr。	默认值：NULL
connect_timeout	连接的最大等待时间。建议设置连接超时值不小于2秒。	单位：s（秒） 默认值：0，表示不超时。
client_encoding	客户端编码格式。	默认值：建立连接时服务端返回的客户端编码格式。
application_name	设置连接数据库的应用程序名称。	默认值：NULL
fallback_application_name	设置连接数据库的备用应用名称。当未显式设置application_name参数时，则会自动回退到fallback_application_name参数值，即将fallback_application_name参数值作为连接的默认应用名称。	默认值：NULL
keepalives	控制客户端与数据库连接的TCP Keepalive机制是否启用。TCP保活机制主要用于检测长时间没有数据交换的连接是否仍然有效，通常用在基于IP的网络通信中。 须知 - TCP保活机制通过定期发送TCP探测包检测连接状态。Keepalive相关的参数有keepalives_idle、keepalives_interval、keepalives_count。 - 空闲时间阈值：若连接空闲超过keepalives_idle秒，系统会发送首个Keepalive探测包。 - 后续探测：若未收到响应，会按keepalives_interval间隔继续发送探测包，最多尝试keepalives_count次。 - 连接状态：若所有探测均失败，连接将被标记为断开，Psycopg会抛出异常或尝试重连。 - 而在Unix域套接字这种本地进程间通信中，由于没有网络延迟和连接中断的问题，所以不需要TCP保活机制来检查连接状态。因此通过Unix域套接字进行连接，则忽略Keepalive相关参数。	取值范围： 1：表示启用。 0：表示禁用。 默认值：1

参数名称	参数说明	参数值
keepalives_idle	连接空闲后，首次发送TCP Keepalive探测包的等待时间。也就是当数据库连接空闲时间（从最后一次数据传输开始计算空闲时间）超过keepalives_idle设定值后，客户端会开始向服务端发送TCP Keepalive探测包。	单位：s（秒） 默认值：0，表示使用系统TCP配置值。
keepalives_interval	客户端未收到服务端ACK响应，重试探测的时间间隔。	单位：s（秒） 默认值：0，表示使用系统TCP配置值。
keepalives_count	指定发送TCP Keepalive探测包的次数。也就是允许连续探测失败的次数，超过则判定连接断开。	默认值：0，表示使用系统TCP配置值。
options	指定在连接开始时发送到服务器的命令行选项。例如，将该参数设置为-c geqo=off表示设置会话的geqo参数值为off。	默认值：NULL
replication	这个选项决定是否该连接应该使用复制协议而不是普通协议。这是PostgreSQL的复制连接以及pg_basebackup之类的工具在内部使用的协议，但也可以被第三方应用使用。	取值范围：支持以下参数值，与大小写无关。 • true、on、yes、1、reply：用于集群内的连接，连接进入到物理复制模式。 • standby_cluster、hadr_main_standby、hadr_standby_cn：用于集群间的连接，连接进入到物理复制模式。 • database：连接进入到逻辑复制模式，连接到dbname参数指定的数据库。 • false、off、no、0、NULL（缺省值）：该连接是一个常规连接，这是默认行为。 默认值：NULL 须知 在物理或者逻辑复制模式中，仅能使用简单查询协议。
requirepeer	指定服务器操作系统的用户名。	默认值：NULL

参数名称	参数说明	参数值
target_session_attrs	设定连接的主机的类型。仅当主机类型与设定值匹配时，才能连接成功。此参数仅在指定多个IP地址时生效。	取值范围： - any：可对所有类型的主机进行连接。 - read-write：仅对可读写的主机进行连接。 - read-only：仅对只读的主机进行连接。 - primary：仅对主备系统中的主节点进行连接。 - standby：仅对主备系统中的备节点进行连接。 - prefer-standby：首先尝试与主备系统中的备节点进行连接。如果host列表的所有备节点都连接失败，则尝试以any模式进行连接。 默认值：primary
tcp_user_timeout	在支持tcp_user_timeout套接字选项的操作系统中，此参数用于设置传输数据在TCP连接被内核强制关闭前允许保持未确认状态的最长时间。当使用Unix域套接字建立连接时，此参数不生效。	单位：s（秒） 默认值：0，表示使用系统TCP配置值。
rw_timeout	设置客户端连接的读写超时时间。 当libpq侧触发超时且连接关闭时，libpq侧下发给数据库侧正在运行的业务会被强制终止。该能力受GUC参数check_disconnect_query控制，设置为on表示支持该能力，设置为off表示不支持该能力。	单位：s（秒） 默认值：0，表示不设置超时时间。

说明

- Psycopg2连接能力基于底层libpq库实现。在负载均衡能力方面存在以下差异：
 - 非透明多写特性下：由于libpq不支持负载均衡机制，因此Psycopg2也不支持负载均衡。

5.6.2.3 执行 SQL 语句

说明

在Psycopg2中，当数据库创建游标对象时默认开启事务，该游标下执行的SQL命令都在该事务中。该事务提交和回滚由connection对象处理。

应用程序通过执行SQL语句来操作数据库，可以执行各种类型的SQL语句，例如创建表、插入数据、更新数据、删除数据、查询等。`execute()`方法用于执行单行SQL语句，`executemany()`方法用于执行多行SQL语句。

以下给出`execute()`和`executemany()`方法使用的一个示例（完整示例请参考[数据库建连、执行SQL并返回结果](#)）。前置条件是已经连接数据库，连接对象为`conn`。

步骤1 调用`cursor()`方法创建游标对象。

```
cur=conn.cursor()
```

步骤2 调用`execute()`方法创建student表。

```
cur.execute("CREATE TABLE student(id integer,name varchar,gender varchar)")
```

步骤3 调用`execute()`方法逐个插入数据。

```
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(1,'Aspirin','M')) # 使用%s作为占位符，执行时参数值会自动替换%s占位符。
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(2,'Taxol','F'))
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(3,'Dixheral','M'))
```

步骤4 调用`executemany()`方法批量插入数据。

```
stus = ((4,'John','M'),(5,'Alice','F'),(6,'Peter','M'))
cur.executemany("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",stus)
```

步骤5 调用`execute()`方法将id为1的学生姓名更新为'Aspirin Updated'。

```
cur.execute("UPDATE student SET name=%s WHERE id=%s", ('Aspirin Updated', 1))
```

步骤6 调用`execute()`方法删除id为2的学生记录。

```
cur.execute("DELETE FROM student WHERE id=%s", (2,))
```

步骤7 调用`execute()`方法查询结果集。

```
cur.execute("SELECT * FROM student")
```

----结束

5.6.2.4 处理结果集

执行SELECT查询语句后，需要从结果集中获取数据，可以使用`fetchone()`方法获取单行数据，使用`fetchmany(size)`方法获取size行数据，或者使用`fetchall()`方法获取查询结果的所有行。示例如下（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

调用`execute()`方法查询结果集后，使用`fetchall()`方法获取查询结果的所有行。

```
results=cur.fetchall()
print (results)
```

说明

对于数据库特有数据类型，如tinyint类型，查询结果中相应字段为字符串形式。

5.6.2.5 关闭数据库连接

数据库操作完成后，需要调用`cur.close()`方法关闭游标，调用`conn.close()`方法关闭数据库连接。

假设连接对象为`conn`，游标对象为`cur`。示例如下（完整示例请参考[数据库建连、执行SQL并返回结果](#)）：

步骤1 调用`cur.close()`方法关闭游标。

```
cur.close()
```

步骤2 调用conn.close()方法关闭数据库连接。

```
conn.close()
```

----结束

须知

此方法关闭数据库连接，并不会自动调用commit()方法提交当前事务。如果要将当前事务操作（如INSERT、UPDATE、DELETE）永久保存到数据库中，需要在执行close()关闭连接之前先调用commit()提交当前事务。

5.6.3 典型应用开发示例

5.6.3.1 数据库建连、执行 SQL 并返回结果

非透明多写特性下行行为结果

本示例演示在Python驱动下连接数据库、建表、插入数据并返回结果的常规操作。

```
# 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在环境中设置环境变量
EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
# 用户需要提供dbname="database"数据库名、port=数据库端口号、host="localhost"表示数据库在本地机器上
运行。
# 关于证书等文件的获取，请登录GaussDB管理控制台，在“实例管理”页面，单击实例名称进入“基本信息”
页面，单击“SSL”处的下载图标，下载根证书或捆绑包，并将根证书ca.pem放置在客户端。

import psycopg2
import os
import ssl

# 从环境变量中获取host和port。
host = "localhost"
port = os.getenv('EXAMPLE_PORT_ENV')

# 从环境变量中获取用户名和密码。
userName = os.getenv('EXAMPLE_USERNAME_ENV')
passWord = os.getenv('EXAMPLE_PASSWORD_ENV')

# 以非加密方式连接数据库。
conn = psycopg2.connect(dbname="database", user=userName, password=passWord, host=host, port=port)

# 用户需要使用SSL方式连接数据库，则采用本行代码。
# conn = psycopg2.connect(dbname="database", user=userName, password=passWord, host=host,
port=port,sslmode="verify-ca", sslrootcert="ca.pem")

# 创建游标对象。
cur=conn.cursor()
# 创建表。
cur.execute("CREATE TABLE student(id integer,name varchar,gender varchar)")

# 插入数据。
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(1,'Aspirin','M'))
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(2,'Taxol','F'))
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(3,'Dixerhal','M'))

# 批量插入数据。
stus = ((4,'John','M'),(5,'Alice','F'),(6,'Peter','M'))
cur.executemany("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",stus)

# 更新数据。
cur.execute("UPDATE student SET name=%s WHERE id=%s", ('Aspirin Updated', 1))
```

```
# 删除数据。
cur.execute("DELETE FROM student WHERE id=%s", (2,))

# 获取结果。
cur.execute('SELECT * FROM student')
results=cur.fetchall()
print (results)

# 提交操作。
conn.commit()

# 插入一条数据。
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s)",(7,'Lucy','F'))

# 回退操作。
conn.rollback()

# 关闭游标和连接。
cur.close()
conn.close()
```

示例运行结果如下：

```
[(1, 'Aspirin Updated', 'M'), (3, 'Dixheral', 'M'), (4, 'John', 'M'), (5, 'Alice', 'F'), (6, 'Peter', 'M')]
```

5.6.3.2 使用日志场景

本示例演示在Python驱动下使用日志的场景。

```
# 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在环境中设置环境变量
EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
# 用户需要提供"database"、$port。"localhost"表示数据库在本地机器上运行。

import logging
import psycopg2
from psycopg2.extras import LoggingConnection
import os

# 从环境变量中获取用户名和密码。
userName = os.getenv('EXAMPLE_USERNAME_ENV')
passWord = os.getenv('EXAMPLE_PASSWORD_ENV')

logging.basicConfig(level=logging.DEBUG) # 日志级别。
logger = logging.getLogger(__name__)

db_settings = {
    "user": userName,
    "password": passWord,
    "host": "localhost",
    "dbname": "database",
    "port": $port
}

# LoggingConnection默认记录所有SQL，可自行实现filter过滤不需要的或敏感的SQL，下面给出了过滤
password的相关SQL。
class SelfLoggingConnection(LoggingConnection):

    def filter(self, msg, curs):
        if db_settings['password'] in msg.decode():
            return b'queries containing the password will not be recorded'
        return msg

# 创建连接对象。
conn = psycopg2.connect(connection_factory=SelfLoggingConnection, **db_settings)
conn.initialize(logger)
cur = conn.cursor() # 创建游标对象。
cur.execute("select 1;")
```

```
# 关闭游标和连接。  
cur.close()  
conn.close()
```

示例运行结果如下：

```
DEBUG: __main__:b'select 1;'
```

须知

- LoggingConnection默认记录所有SQL信息，且不会对敏感信息进行脱敏，可通过filter函数自行定义输出的SQL内容。
- 日志功能是psycopg2为了方便开发者显性调试全量SQL而提供的额外功能，默认情况下不需要使用。该功能会在psycopg2执行SQL语句前打印SQL语句，但是，需要在debug日志级别下才会输出。该功能不是默认功能，只在特殊需要时才使用，没有特别需求时不建议使用。详情参考：<https://www.psycopg.org/docs/extras.html?highlight=loggingconnection>。

5.6.4 Psycopg 接口参考

5.6.4.1 psycopg2.connect()

功能描述

创建数据库连接，并返回connection对象。

原型

```
connect(dsn, connection_factory, cursor_factory, **kwargs)
```

参数

表 5-111 psycopg2.connect 参数

关键字	参数说明
dsn	数据源名称 (Data Source Name)，它是一个字符串，包含了数据库连接所需的信息，如数据库名称、服务器地址、端口号、用户名、密码等。默认为None。
connection_factory	提供给创建自定义游标的简便方法。默认为None。
cursor_factory	使用自定义游标。默认为None。
**kwargs	是一个可变数量的关键字参数，允许传递其他参数给连接函数。常用的连接参数请参见 连接参数参考 。

返回值

connection对象（连接数据库实例的对象）。

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.2 connection.cursor()

功能描述

创建一个数据库游标对象，用于在数据库中执行SQL命令、管理查询结果和控制事务。

原型

```
cursor(name, cursor_factory, scrollable, withhold)
```

参数

表 5-112 connection.cursor 参数

关键字	参数说明
name	cursor名称，默认为None。
cursor_factory	用于创造非标准cursor，默认为None。
scrollable	设置SCROLL选项，默认为None。
withhold	设置HOLD选项，默认为False。

返回值

cursor对象。

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.3 cursor.execute(query,vars_list)

功能描述

执行被参数化的SQL语句。psycopg2模块支持用%s标志的占位符。

原型

```
cursor.execute(query,vars_list)
```

参数

表 5-113 cursor.execute 参数

关键字	参数说明
query	待执行的SQL语句。
vars_list	变量列表，用于替换SQL语句中的%s占位符。

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.4 cursor.executemany(query,vars_list)

功能描述

执行多次相同SQL语句，特别适用于需要批量插入数据的情况。

原型

```
cursor.executemany(query,vars_list)
```

参数

表 5-114 cursor.executemany 参数

关键字	参数说明
query	待执行的SQL语句。
vars_list	包含多个元组或字典的列表。每个元组或字典对应一次SQL语句的执行，其中的元素将被用来替换SQL语句中的%s占位符。

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.5 connection.commit()

功能描述

提交当前事务，将所有未保存的数据库操作（如INSERT、UPDATE、DELETE）永久保存到数据库中，并结束当前事务。以下是详细说明：

- 持久化更改：自上次提交或回滚后，所有通过该连接执行的数据修改操作会被永久写入数据库。
- 结束事务：提交后，当前事务终止。后续操作将属于新事务，新事务需再次提交才能生效。

须知

默认情况下，Psycopg2在执行第一个命令之前打开一个事务，若不调用commit()，关闭连接时事务会自动回滚，所有数据修改都将丢失。

原型

```
connection.commit()
```

参数

无

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.6 connection.rollback()

功能描述

回滚当前事务，撤销所有未提交的更改，确保数据的一致性。

原型

```
connection.rollback()
```

参数

无

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.7 cursor.fetchone()

功能描述

- 1. 获取单行数据：执行SQL查询后，调用fetchone()会返回结果集的下一行数据（以元组形式），未读取数据时首次调用返回第一行。
- 2. 无数据时返回None：当所有行均被读取后，再次调用会返回None，表示结果集已耗尽。
- 3. 游标控制：每次调用fetchone()会使游标向前移动一行，按顺序遍历结果。

原型

```
cursor.fetchone()
```

参数

无

返回值

单个元组，为结果集的第一条结果。如果结果集中没有数据时，则返回None。

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.8 cursor.fetchmany(size)

功能描述

执行SQL查询后，调用fetchmany(size)会从游标的未读取结果中返回指定数量的行。每次调用会从上一次结束的位置继续读取，直到结果集耗尽。

原型

```
cursor.fetchmany(size)
```

参数

表 5-115 cursor.fetchmany 参数

关键字	参数说明
size	整型，表示每次获取的行数。

返回值

返回“最多size行”，但可能更少。具体返回结果，如[表5-116](#)所示。

表 5-116 不同场景的返回值

场景	返回结果
游标的未读取结果集剩余行数大于等于size。	返回size行。
游标的未读取结果集剩余行数小于size。	返回剩余全部。
游标的未读取结果集已无数据。	返回空列表。

示例

无

5.6.4.9 cursor.fetchall()

功能描述

获取查询结果的所有（剩余）行，并将它们作为元组列表返回，适用于结果集较小的情况。

- 场景1：假设有一个查询返回了5条记录，调用cursor.fetchall()会得到一个包含这5个元组的列表。每个元组对应一行数据，字段按查询顺序排列。
- 场景2：假设有一个查询返回了5条记录，先调用fetchone()会返回结果集的第一行数据，游标向前移动一行。然后再调用cursor.fetchall()，则会将结果集剩余4行数据作为元组列表返回。

须知

- 游标状态：调用fetchall()后，游标将处于结果集末尾，再次调用会返回空列表。
- 内存风险：大数据量时需谨慎使用，避免内存溢出。

原型

```
cursor.fetchall()
```

参数

无

返回值

元组列表，为结果集的所有结果。如果结果集没有数据时，则返回空列表。

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.10 cursor.close()

功能描述

关闭当前连接的游标。

原型

```
cursor.close()
```

参数

无

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.6.4.11 connection.close()

功能描述

关闭数据库连接。

须知

此方法关闭数据库连接，并不自动调用commit()。如果只是关闭数据库连接而不调用commit()方法，那么所有更改将会丢失。

原型

```
connection.close()
```

参数

无

返回值

无

示例

请参见[数据库建连、执行SQL并返回结果](#)。

5.7 基于 Go 驱动开发

5.7.1 开发流程

Go驱动应用开发流程依赖于Go原生的database/sql包和GaussDB开发的driver驱动，database/sql包提供用户使用的标准接口，GaussDB实现需要被database/sql包调用的接口。

Go开发应用程序的流程可分为环境准备、连接数据库、执行SQL语句、处理结果集、关闭连接，开发流程如图5-19所示。

图 5-19 Go 开发应用程序的流程图



5.7.2 开发步骤

5.7.2.1 环境准备

获取驱动包

下载表5-117中的驱动包和驱动包校验包。

表 5-117 驱动包下载列表

版本	下载地址
V2.0-10.x	驱动包 驱动包校验包

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：

- 上传软件包和软件包校验包到虚拟机的同一目录下。
- 执行如下命令，校验软件包完整性。

```
F=$(echo DBS-GaussDB-driver*.tar.gz);echo "$(awk '{print $1}' "$F.sha256") $F" | sha256sum -c
```

如果回显OK，则校验通过。例如：

```
DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz: OK
```

- Windows操作系统软件包完整性校验：

- 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。
- 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

- {驱动包本地目录}**：请根据实际下载目录进行替换。例如：C:\Users

- {驱动包名}**：请根据实际下载的驱动包名进行替换。例如：DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz

示例：certutil -hashfile C:\Users\DBS-GaussDB-driver_507.0_20260629130806.133858239.tar.gz sha256

- 将**b**获取到的Hash值和**表5-117**中获取到的驱动包校验包的Hash值进行比较。
 - 若一致则通过校验。
 - 若不一致，请重新下载驱动包，重复**a~c**进行校验。

获取 Go 驱动包

步骤1 解压获取到的驱动包。驱动包的文件层级关系参考以下结构：DBS-GaussDB-driver_*/数据库部署形态/驱动类型/操作系统架构/发布包/驱动包

步骤2 选择数据库部署形态。

驱动包不区分部署形态，可选择Centralized/Distributed任意一种数据库部署形态对应的驱动包。

步骤3 选择驱动类型。

选择go_driver.tar.gz，并解压。

步骤4 选择操作系统架构。

1. 非密态功能不依赖系统架构，可选择任意一种操作系统架构。
2. 密态功能需要严格选择对应系统架构的目录，可使用以下命令查看客户端系统架构。

```
uname -a
```

步骤5 选择go发布包。

步骤6 解压go发布包，则可获取Go驱动源码包：openGauss-connector-go-pq。

----结束

配置环境变量

- GO111MODULE：用户使用在线导入的方式安装Go驱动时需要设置GO111MODULE为on。

```
export GO111MODULE=on
```

说明

如果不希望进行go mod工程的改造，需将GO111MODULE设置为off，并手动下载依赖包。依赖包与驱动根目录和业务代码保持同级。

- GOPROXY：用户使用在线导入时需配置包含Go驱动包的路径[path-to-goproxy]。

```
export GOPROXY=[path-to-goproxy]
```
- GOPATH：Go驱动依赖包存放路径[path-to-gopath]。

```
export GOPATH=[path-to-gopath]
```
- Go版本要求在1.13或以上，可以执行以下命令查看Go环境变量GOVERSION的配置值。

```
go env
```
- 用户根据场景需要，配置其他的Go环境变量。

下载依赖包

步骤1 Go Modules开启时（即GO111MODULE为on），可进入Go驱动代码根路径，执行以下命令下载驱动代码所需相关依赖。

```
go mod tidy
```

步骤2 完成**步骤1**后，在go.mod文件中添加以下一行，替换驱动包依赖模块路径。

```
replace [Go驱动模块名] => [Go驱动包本地路径]
```

说明

- [Go驱动模块名]，根据实际使用Go驱动模块名修改。
- [Go驱动包本地路径]，根据实际Go驱动本地路径修改。

以代码xxx.go中引用Go驱动为例：

```
// xxx.go
package main
import (
    _ "openGauss-connector-go-pq"
)
```

go.mod文件添加以下内容：

```
replace openGauss-connector-go-pq => [path-to-driver]/openGauss-connector-go-pq
```

说明

- 示例xxx.go中“openGauss-connector-go-pq”为驱动模块名，用户可根据自身需求修改。
- go.mod中[path-to-driver]/openGauss-connector-go-pq为本地Go驱动包路径，该行将xxx.go中引用的Go驱动模块名openGauss-connector-go-pq指向本地Go驱动[path-to-driver]/openGauss-connector-go-pq。

----结束

须知

- 通过go mod tidy下载相关依赖时可能会下载某个依赖的低版本，如果依赖的低版本存在漏洞，可以通过更改go.mod文件中对应依赖的版本号，更新依赖到漏洞修复后的版本来规避风险。
- 使用go 1.13版本或以上即可，runtime运行库需要更新至1.18版本或以上。

5.7.2.2 连接数据库

5.7.2.2.1 以非加密方式连接

使用Go驱动开发时，配置连接参数并生成连接串，再调用Golang中database/sql的sql.Open()接口创建一个新的SQL数据库连接对象，sql.Open()并不会立即建立数据库连接，实际建立连接是在第一次进行数据库操作（如查询或db.Ping()）时才建立。可以通过调用Go sql的标准接口db.Ping()检查数据库连接是否能成功建立，并在有需要时建立连接（例如刚调用sql.Open()后，db.Ping()会触发首次连接尝试）。

sql.Open()函数原型

Go驱动提供了如下的方法用于创建一个数据库连接对象。

```
func Open(driverName, dataSourceName string) (*DB, error)
```

参数说明：

- driverName为驱动名称。数据库驱动的可选项有：
 - “gaussdb”：连接GaussDB数据库时，需要传入驱动名称“gaussdb”。本章节示例，数据库驱动名称均使用“gaussdb”。
 - 其他Go驱动名称，请参见[Go驱动兼容](#)。
- dataSourceName为连接字符串，支持DSN和URL两种：
 - DSN格式：key1 = value1 key2 = value2 ...，每组关键字间使用空格隔开，等号左右的空格是可选的。
 - URL格式：driverName://[userspec@][hostspec]/[dbname][?paramspec]。其中，driverName为驱动名称，数据库的驱动名称使用“gaussdb”。userspec表示user[:password]，需要注意的是使用URL进行连接时，密码中不可包含URL串中的分隔符。如果密码中包含分隔符的话，建议采用DSN格式。另外，不允许使用初始化用户进行远程登录。hostspec表示[host][:port][...]。dbname为数据库名称。paramspec表示name=value[&...]。

须知

- 在DSN格式中，对于多个IP地址的场景：
 - 当num(ip) = num(port)时，ip和port是一一对应匹配。
 - 当num(ip) > num(port)时，无法匹配到port的ip均与第一个port匹配。例如，host = ip1, ip2, ip3 port = port1, port2的匹配情况为ip1:port1, ip2:port2, ip3:port1。
 - 当num(ip) < num(port)时，则多余的port被舍弃，即使用不到。例如host = ip1, ip2, ip3 port = port1, port2, port3, port4的匹配情况为ip1:port1, ip2:port2, ip3:port3。
- 在URL格式中，对于多个IP地址的场景：
 - URL串中ip:port必须成对出现，即num(ip) = num(port)，并以逗号隔开。例如，gaussdb://user:password@ip1:port1, ip2:port2, ip3:port3/gaussdb。
 - URL串中仅包含多个IP地址，port由环境变量指定或采用默认值5432。例如gaussdb://user:password@ip1, ip2, ip3/gaussdb并设置环境变量PGPORT = "port1, port2"，其匹配情况为ip1:port1, ip2:port2, ip3:port1。未设置环境变量的匹配情况为ip1:5432, ip2:5432, ip3:5432。
- 若输入IP地址为IPv6格式，则URL格式连接串中需在IP两边增加'[]'，例如：gaussdb://user:password@[ip1]:port1, [ip2]:port2, [ip3]:port3/gaussdb。

db.Ping()函数原型

Golang的database/sql中提供了DB对象绑定的接口Ping，测试连接数据库对象创建后是否连接成功，以下为接口定义。

```
func (db *DB) Ping() error
```

用户可以使用db.Ping()函数测试是否成功连接数据库，通过以下代码可以更清楚了解如何使用db.Ping()函数。

```
err = db.Ping()
if err != nil {
    log.Fatal("连接失败:", err)
    return
}
log.Println("成功连接到数据库")
```

当err为nil时表示成功建连且没有报错信息，上述代码则会输出“成功连接到数据库”。err不为nil表示建立数据库连接失败，并接收错误信息，用户可以通过log.Fatal输出错误信息err。例如，建立数据库连接失败时，db.Ping()返回错误信息“connection refused”，此时err接收该错误信息，调用log.Fatal("连接失败:", err)则会输出“连接失败:connection refused”。

方式一

通过DSN/URL连接串配置连接参数。例如，用户自行设置IP地址、端口号、用户名和密码对应的[GOXX]环境变量，并在业务代码中通过os.Getenv()方法获取对应变量值进行DSN/URL连接串的拼接，步骤如下：

步骤1 设置连接串拼接所需的环境变量。

```
export GOHOSTIP="XXX.XXX.XXX.XXX"
export GOPORT=XXX
```

```
export GOUSRNAME="XXX"
export GOPASSWD="XXXXXXX"
```

步骤2 引用Go驱动模块以及其他依赖标准库。

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "database/sql"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "os"
    "log"
)
```

说明

- database/sql: Golang标准库，提供通用的数据库操作接口，定义与数据库交互的核心方法（如 Query(), Exec(), Prepare() 等）。
- _ "[go驱动模块名]": Go驱动模块。
- os: Golang标准库，提供与操作系统交互的功能（文件操作、环境变量）。本示例中os包用于从环境变量获取参数值，驱动不依赖此包。
- log: Golang标准库，提供日志记录功能，常用于输出错误信息或调试日志。本示例中log包用于打印日志，驱动不依赖此包。

步骤3 调用os.Getenv()方法从环境变量获取IP地址、端口号、用户名、密码。

```
hostip := os.Getenv("GOHOSTIP")
port := os.Getenv("GOPORT")
username := os.Getenv("GOUSRNAME")
passwd := os.Getenv("GOPASSWD")
```

步骤4 采用DSN格式或URL格式定义连接字符串，调用sql.Open() 函数创建数据库连接对象。

```
str := "host=" + hostip + " port=" + port + " user=" + username + " password=" + passwd + " dbname=gaussdb sslmode=disable" //DSN连接串。用户需要自行设置dbname和sslmode关键字的取值。
// str := "gaussdb://" + username + ":" + passwd + "@" + hostip + ":" + port + "/" + "gaussdb? sslmode=disable" //URL连接串。用户需要自行设置数据库名称和sslmode的取值。
db, err := sql.Open("gaussdb", str) //创建数据库连接对象，但不会立即建立数据库连接。
if err != nil {
    log.Fatal(err) //执行报错时，使用log.Fatal方法输出错误信息。
}
```

步骤5 调用db.Ping()验证数据库建连是否成功。

```
err = db.Ping()
if err != nil {
    log.Fatal(err) //err不为nil表明连接失败，则会执行log.Fatal输出连接失败的原因。
}
```

----结束

说明

步骤2-步骤5为示例中的业务代码。

方式二

部分连接参数通过环境变量[PGXX]设置参数值，并且这部分参数在DSN/URL连接串中不进行配置。例如DSN/URL连接串配置用户名和密码，通过环境变量PGHOST、PGPORT、PGDATABASE、PGXX分别指定IP地址、端口号、数据库名、其他参数，步骤如下：

步骤1 设置环境变量。

```
export PGHOST="XXX.XXX.XXX.XXX"
export PGPORT=XXX
```

```
export PGDATABASE="XXX"
export PGSSLMODE="disable"
export GOUSRNAME="XXX"
export GOPASSWD="XXXXXXX"
```

步骤2 引用Go驱动模块以及其他依赖标准库。

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "database/sql"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "os"
    "log"
)
```

说明

- database/sql: Golang标准库，提供通用的数据库操作接口，定义与数据库交互的核心方法（如 Query(), Exec(), Prepare() 等）。
- _ "[go驱动模块名]": Go驱动模块，通过[此步骤](#)将Go驱动模块指向本地Go驱动目录。
- os: Golang标准库，提供与操作系统交互的功能（文件操作、环境变量）。本用例中os包用于从环境变量获取参数值，驱动不依赖此包。
- log: Golang标准库，提供日志记录功能，常用于输出错误信息或调试日志。本用例中log包用于打印日志，驱动不依赖此包。

步骤3 调用os.Getenv()方法获取用户名、密码。

```
usrname := os.Getenv("GOUSRNAME")
passwd := os.Getenv("GOPASSWD")
```

步骤4 采用DSN格式或URL格式定义连接字符串，并调用sql.Open() 函数创建数据库连接对象。

```
str := "user=" + username + " password=" + passwd //DSN连接串。
// str := "gaussdb://" + username + ":" + passwd //URL连接串。
db, err := sql.Open("gaussdb", str) //创建数据库连接对象，但不会立即建立数据库连接。
if err != nil {
    log.Fatal(err) //执行报错时，使用log.Fatal方法输出错误信息。
}
```

步骤5 调用db.Ping()验证数据库建连是否成功。

```
err = db.Ping()
if err != nil {
    log.Fatal(err) //err不为nil表明连接失败，则会执行log.Fatal输出连接失败的原因。
}
```

----结束

说明

步骤2-步骤5为示例中的业务代码。

5.7.2.2.2 以 SSL 方式连接

应用程序通过Go驱动连接GaussDB数据库时，可以开启SSL加密客户端和服务端之间的通讯。当开启SSL加密模式时，Go驱动默认执行TLS 1.3标准协议，支持SSL加密模式最低为TLS 1.2标准协议。本小节主要介绍如何采用SSL方式对客户端进行配置。

前提条件

已经获取服务端所需要的证书和私钥文件，并完成服务端配置；同时已经获取客户端所需要的client.crt客户端证书、cacert.pem根证书、client.key客户端私钥文件。

说明

关于证书生成和获取，请联系管理员。服务端配置证书的具体操作，请联系管理员。

使用 require 和 verify-ca 验证连接数据库

步骤1 设置环境变量。

```
export GOHOSTIP="XXX.XXX.XXX.XXX"
export GOPORT=XXX
export GOUSRNAME="XXX"
export GOPASSWD="XXXXXXX"
export GOSSLPASSWD="XXXXXXX"
```

步骤2 导入连接所需的依赖包。

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "database/sql"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "os"
    "log"
)
```

步骤3 调用os.Getenv()方法从环境变量获取IP地址、端口号、用户名、用户密码、SSL密码短语。

```
hostip := os.Getenv("GOHOSTIP")
port := os.Getenv("GOPORT")
username := os.Getenv("GOUSRNAME")
passwd := os.Getenv("GOPASSWD")
sslpasswd := os.Getenv("GOSSLPASSWD")
```

步骤4 采用DSN格式定义连接字符串。

```
dsnStr := "host=" + hostip + " port=" + port + " user=" + username + " password=" + passwd + "
dbname=gaussdb sslcert=certs/client.crt sslkey=certs/client.key sslpassword=" + sslpasswd
```

步骤5 配置sslmode。

```
parameters := []string{
    "sslmode=require",
    "sslmode=verify-ca sslrootcert=certs/cacert.pem",
}
```

步骤6 启用SSL加密的方式，分别使用require和verify-ca验证连接数据库。

```
for _, param := range parameters {
    db, err := sql.Open("gaussdb", dsnStr+param)
    if err != nil {
        log.Fatal(err)
    }
    err = db.Ping() //验证是否成功连接数据库。
    if err != nil {
        log.Fatal(err)
    }
}
```

----结束

说明

步骤2-步骤6为示例中的业务代码。

使用 RSA 和 ECDSA 两种证书连接数据库

步骤1 设置环境变量。

```
export GOHOSTIP="XXX.XXX.XXX.XXX"
export GOPORT=XXX
export GOUSRNAME="XXX"
export GOPASSWD="XXXXXXX"
export GOSSLPASSWD="XXXXXXX"
```

步骤2 导入连接所需的依赖包。

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "database/sql"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "os"
    "log"
)
```

步骤3 调用os.Getenv()方法从环境变量获取IP地址、端口号、用户名、用户密码、SSL密码短语。

```
hostip := os.Getenv("GOHOSTIP")
port := os.Getenv("GOPORT")
username := os.Getenv("GOUSRNAME")
passwd := os.Getenv("GOPASSWD")
sslpasswd := os.Getenv("GOSSLPASSWD")
```

步骤4 采用DSN格式定义连接字符串。

```
dsnStr := "host=" + hostip + " port=" + port + " user=" + username + " password=" + passwd + "
dbname=gaussdb sslcert=certs/client.crt sslkey=certs/client.key sslpassword=" + sslpasswd
```

步骤5 配置SSL相关参数。

```
connStrs := []string{
    " sslmode=verify-ca sslcert=certs/client_rsa.crt sslkey=certs/client_rsa.key sslpassword=" + sslpasswd + "
    sslrootcert=certs/cacert_rsa.pem",
    " sslmode=verify-ca sslcert=certs/client_ecdsa.crt sslkey=certs/client_ecdsa.key sslpassword=" + sslpasswd + "
    sslrootcert=certs/cacert_ecdsa.pem",
}
```

步骤6 分别使用RSA和ECDSA两种证书来连接数据库。

```
for _, connStr := range connStrs {
    db, err := sql.Open("gaussdb", dsnStr+connStr)
    if err != nil {
        log.Fatal(err)
    }
    err = db.Ping() //验证是否成功连接数据库。
    if err != nil {
        log.Fatal(err)
    }
}
```

----结束

说明

步骤2-步骤6为示例中的业务代码。

5.7.2.2.3 连接参数参考

常用连接参数

Go驱动的常用连接参数，如[表5-118](#)所示。

表 5-118 数据库连接参数

参数名称	参数说明
host	主机IP地址，默认值为localhost。也可通过环境变量PGHOST来指定。
port	主机服务器的端口号，默认值为5432。也可通过环境变量PGPORT来指定。
dbname	数据库名，也可通过环境变量PGDATABASE来指定。
user	用户名，也可通过环境变量PGUSER来指定。
password	密码。
connect_timeout	指定连接服务器操作的超时时间，也可通过环境变量PGCONNECT_TIMEOUT来指定。默认值为0，表示没有超时。
sslmode	启用SSL加密的方式，也可通过环境变量PGSSLMODE来指定。默认值为prefer。 参数取值范围： • disable：不使用SSL安全连接。 • allow：如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。 • prefer：如果数据库支持，那么首选使用SSL安全加密连接，但不验证数据库服务器的真实性。 • require：必须使用SSL安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。 • verify-ca：必须使用SSL安全连接，并验证服务器是否具有由可信任的证书机构签发的证书。 • verify-full：必须使用SSL安全连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。
sslkey	SSL连接时，指定客户端密钥路径。也可通过设置环境变量PGSSLKEY来指定。
sslcert	SSL连接时，指定客户端证书路径。也可通过设置环境变量PGSSLCERT来指定。
sslrootcert	SSL连接时，指定根证书路径。也可通过设置环境变量PGSSLROOTCERT来指定。
sslcrl	指定SSL证书吊销列表（CRL，Certificate Revocation List）的文件路径。也可通过设置环境变量PGSSLCRL来指定。若服务端证书存在于已配置的CRL中，客户端在SSL握手阶段将拒绝认证并终止连接。
sslpassword	指定解密SSL密钥文件的密码短语。启用此参数时，表明sslkey文件是通过加密方式存储的，当前sslkey支持DES/AES加密方式。 说明 DES加密算法安全性低，存在安全风险，建议使用更安全的加密算法。

参数名称	参数说明
disable_prepared_binary_result	表示此连接在从预准备语句接收查询结果时，是否使用二进制格式。该参数为字符串类型，仅用于调试。默认值为no。 取值范围： - yes：表示此连接在从预准备语句接收查询结果时不使用二进制格式。 - no：表示此连接在从预准备语句接收查询结果时使用二进制格式。
binary_parameters	表示连接期间是否以二进制形式发送[]byte。该参数属于字符串类型。默认值为no。 取值范围： - yes：表示以二进制形式发送[]byte。建议值为yes，绑定参数按照[]byte绑定，可以减少内部类型转换。 - no：表示不以二进制形式发送[]byte。
target_session_attrs	指定数据库的连接类型，该参数用于识别主备节点，也可通过环境变量PGTARGETSESSIONATTRS来指定。默认值为any，该参数共有六种取值：any、master、slave、preferSlave、read-write、read-only。 - any：尝试连接URL连接串中的任何一个数据节点。 - master：尝试连接到URL连接串中的主节点，如果找不到就抛出异常。 - slave：尝试连接到URL连接串中的备节点，如果找不到就抛出异常。 - preferSlave：尝试连接到URL连接串中的备数据节点。若没有可用的备数据节点，则连接到主数据节点。 - read-write：仅对可读写的主机进行连接。 - read-only：仅对只读的主机进行连接。
loggerLevel	指定日志级别，打印相关日志信息，也可通过环境变量PGLOGGERLEVEL来指定。 支持trace、debug、info、warn、error、none，级别从高到低。默认值为error。 说明 日志或报错信息中若含有安全相关的关键字（PASSWORD、GS_ENCRYPT、IDENTIFIED等），Go驱动会使用*****代替相关信息。
application_name	标识连接到数据库的应用程序名称。默认值为go-driver，该参数不建议用户配置。
RuntimeParams	设置连接会话时默认运行的USERSET类型GUC参数的值。例如参数search_path、application_name、statement_timeout等，可以在连接字符串中进行配置。

参数名称	参数说明
autoBalance	分布式环境下，指定负载均衡的策略，为字符串类型。默认值为false，参数取值共有六种：true、balance、roundrobin、shuffle、priorityn、false。 • true、balance或roundrobin：表示开启go驱动sql负载均衡功能，将应用程序的多个连接均衡到数据库集群中各个可用的CN。 例如： gaussdb://user:password@host1:port1,host2:port2/database? autoBalance=true Driver将定期刷新整个集群可用CN列表，该刷新周期可使用参数recheckTime进行配置，默认值为10s。比如获取到的列表为： host1:port1,host2:port2,host3:port3,host4:port4 在启用负载均衡时，host1和host2仅在首次连接时保证高可用性。之后，Driver将从host3、host4、host1、host2中依次选择可用的CN来刷新可用CN列表，后续用户调用的Connector.Connect将使用RoundRobin算法从host1、host2、host3、host4选取CN节点进行连接。 • priorityn：表示开启Driver优先级负载均衡功能，将应用程序的多个连接首先均衡到url上配置的前n个中可用的CN数据库节点，当url上配置前n个节点全部不可用时，连接会随机分配到数据库集群中其他可用CN数据库节点。n为正整数，且小于url上配置的CN数量。 例如： gaussdb://user:password@host1:port1,host2:port2,host3:port3/database? autoBalance=priority2 Driver将定期获取整个集群可用CN列表，比如获取到的列表为： host1:port1,host2:port2,host3:port3,host4:port4,host5:port5,host6:port6 其中host1和host2处于AZ1，host3和host4处于AZ2。Driver将从优先从host1，host2中做负载均衡，host1和host2全部不可用才从host3、host4、host5、host6中随机选择CN节点进行连接。 • shuffle：表示开启Driver随机负载均衡功能，将应用程序的多个连接随机均衡到数据库集群中的各个可用CN。 例如： gaussdb://user:password@host1:port1,host2:port2,host3:port3/database? autoBalance=shuffle Driver将定期获取整个集群的可用CN列表，比如获取到的列表为： host1:port1,host2:port2,host3:port3,host4:port4 host1、host2和host3仅在首次连接时保证高可用性，后续将使用洗牌（shuffle）算法在刷新后的CN列表中随机选择一个CN节点进行连接。 • false：表示不开启Driver负载均衡功能和优先级负载均衡功能。

参数名称	参数说明
	说明 - 负载均衡是基于连接级别，不是基于事务级别。如果连接是长连接，并且连接上的负载不均衡，无法保证CN节点上的负载均衡。 - 负载均衡仅能在分布式场景下使用。
recheckTime	表示定期检测数据库集群中CN状态，获取可用CN的IP列表的时间间隔。该参数为integer类型，单位为s，取值范围为5 ~ 60，默认为10s。
usingEip	指定是否使用业务IP实现负载均衡。该参数为字符串类型，默认值为yes。 - yes：表示使用业务IP（对应pgxc_node表中node_host1和node_port1列）实现负载均衡。 - no：表示使用数据IP（对应pgxc_node表中node_host和node_port列）实现负载均衡。
tcp_syn_retries	设置TCP连接建立阶段的最大重试次数。若未配置此参数，系统将采用设备默认的重试次数。建议通过以下公式确保合理配置： 重试次数 × 单次重试间隔 < connect_timeout。 说明 单次重试间隔为系统底层实现值，通常无需手动配置。
socketTimeout	设置读写超时时间，如果业务语句执行或者从网络读取数据流超过该值，连接中断。 说明 该参数限制单条SQL语句最长的执行时间，单语句执行超过该值则会中断重连。建议根据业务特征进行配置，如果未配置，默认为0，即不会超时。
timeZone	设置该参数后，建连时将数据库GUC参数TimeZone设为该参数值，查询不带时区的timestamp/date数据时，返回结果会自动补充GUC参数TimeZone指定的时区。该参数属于字符串类型。 取值范围：与数据库GUC参数TimeZone的取值范围一致。GUC参数TimeZone的取值范围，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 客户端连接缺省设置 > 区域和格式化”。

连接参数配置值的生效优先级

Go驱动可以通过以下方式获取连接参数配置值，分别为：

- DSN/URL连接串（显式指定）
- 环境变量[PGXX]（部分参数可通过环境变量来指定，详见[表5-118](#)）
- 默认值

连接参数配置值的生效优先级：DSN/URL连接串 > 环境变量[PGXX] > 默认值。以配置连接参数中的端口号为例：

- 创建数据库连接对象时，端口号优先从DSN或URL连接串获取port参数的配置值。若连接串有配置port参数，则以该配置值作为端口号创建数据库连接对象。

2. 若连接串没有配置port参数，则Go驱动会进一步检查是否配置环境变量PGPORT。若环境变量PGPORT对应值不为空，则以该配置值作为端口号创建数据库连接对象。
3. 若连接串和环境变量都没有配置，则以默认值5432作为端口号创建数据库连接对象。

说明

- [PGXX]为环境变量名，实际环境变量名参考[表5-118](#)中参数说明部分。
- 若选择使用[PGXX]环境变量中的值进行数据库连接对象创建，建议优先检查[PGXX]对应的值是否为预期值。
- 环境变量可以通过export PGXX="value"进行配置。[PGXX]为[表5-118](#)的环境变量，value为配置值。以PGTARGETSESSIONATTRS为例：
export PGTARGETSESSIONATTRS="master"
- 连接参数配置参考[方式一](#)或[方式二](#)。

建立连接的首包介绍

驱动与数据库建立连接过程中，Socket（套接字）建立之后，由驱动以key value形式向数据库发送的第一个报文为首包。首包内容包括用户名（user）、数据库名（database）以及部分GUC参数信息（enable_full_encryption、application_name、RuntimeParams）。其中RuntimeParams指代数据库支持的USERSET类型GUC参数，而非参数名，具体介绍可以参考[RuntimeParams](#)。GUC参数的具体介绍，请参见《参考》中“数据库运行参数说明 > GUC参数说明”。

须知

数据库执行以下命令后，上述首包GUC参数的值将被更改为首包设置的值。
reset all;

5.7.2.3 执行 SQL 语句

[type DB](#)的Exec类方法用于执行不返回结果集的SQL语句，比如CREATE TABLE、INSERT、UPDATE、DELETE等DDL和DML语句。Query类方法用于执行SELECT查询语句。

Exec 类方法

以下给出Exec方法使用的一个示例（完整示例请参考[示例一](#)）。

前提条件：已经连接数据库，连接对象为db。

步骤1 定义包含4个SQL语句的字符串切片。

```
sqls := []string{
    "DROP TABLE IF EXISTS testExec",
    "CREATE TABLE testExec(f1 int, f2 varchar(20), f3 number, f4 timestampz, f5 boolean)",
    "INSERT INTO testExec VALUES(1, 'abcdefg', 123.3, '2022-02-08 10:30:43.31 +08', true)",
    "INSERT INTO testExec VALUES(:f1, :f2, :f3, :f4, :f5)",
}
```

步骤2 定义数据切片。

```
intF1 := []int{2, 3, 4, 5, 6}
intF2 := []string{"hello world", "华为", "北京", "nanjing", "研究所"}
intF3 := []float64{641.43, 431.54, 5423.52, 665537.63, 6503.1}
```

```
intF4 := []time.Time{
    time.Date(2022, 2, 8, 10, 35, 43, 623431, time.Local),
    time.Date(2022, 2, 10, 19, 11, 54, 353431, time.Local),
    time.Date(2022, 2, 12, 6, 11, 15, 636431, time.Local),
    time.Date(2022, 2, 14, 4, 51, 22, 747653, time.Local),
    time.Date(2022, 2, 16, 13, 45, 55, 674636, time.Local),
}
intF5 := []bool{false, true, false, true, true}
```

步骤3 通过循环结构和range关键字遍历sqls切片，调用Exec方法执行不返回结果集的SQL语句。

```
for _, s := range sqls {
    if strings.Contains(s, ":f") { //判断是否参数化SQL。
        for i, _ := range intF1 { //遍历数据索引。
            _, err := db.Exec(s, intF1[i], intF2[i], intF3[i], intF4[i], intF5[i]) //执行参数化插入。
            if err != nil {
                log.Fatal(err)
            }
        }
    } else { //非参数化SQL。
        _, err = db.Exec(s) //直接执行DDL。
        if err != nil {
            log.Fatal(err)
        }
    }
}
```

----结束

Query 类方法

调用Exec方法向testExec表插入数据后，可以调用Query类方法执行返回结果集的SQL语句，QueryRow(query string)只返回一个数据行，Query(query string)返回多个数据行。

以下给出Query类方法使用的示例（完整示例请参考[示例一](#)）。

- 调用QueryRow方法查询单行数据。
row := db.QueryRow("SELECT * FROM testExec")
- 调用Query方法执行带条件的查询，返回多个数据行。
rows, err := db.Query("SELECT * FROM testExec WHERE f1 > :1", 1)
if err != nil {
 log.Fatal(err)
}

5.7.2.4 处理结果集

执行SELECT语句查询数据后，可以使用type Rows接口和type Row接口对数据结果集进行处理。type Row接口用于处理单行结果，提供(r *Row)Scan(dest ...interface{})方法将数据扫描到指定的变量。type Rows接口用于处理多行结果，提供遍历方法(rs *Rows)Next()和数据扫描方法(rs *Rows)Scan(dest ...interface{})进行逐行扫描。

示例如下（完整示例请参考[典型应用开发示例](#)的示例一）：

调用QueryRow方法查询单行数据后，使用type Row接口的Scan方法将这一行数据扫描到指定的变量。

步骤1 定义变量，用于接收数据结果集。

```
var f1 int
var f2 string
var f3 float64
var f4 time.Time
var f5 bool
```

步骤2 使用type Row接口的Scan方法将单行数据扫描到指定的变量。

```
err = row.Scan(&f1, &f2, &f3, &f4, &f5)
if err != nil {
    log.Fatal(err)
} else {
    fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
}
```

----结束

调用Query方法返回多行数据后，使用for循环和type Rows接口的Next和Scan方法逐行将数据扫描到指定的变量。

步骤1 定义变量，用于接收数据结果集。

```
var f1 int
var f2 string
var f3 float64
var f4 time.Time
var f5 bool
```

步骤2 使用for循环和type Rows接口的Next和Scan方法逐行将数据扫描到指定的变量。

```
for rows.Next() {
    err = rows.Scan(&f1, &f2, &f3, &f4, &f5)
    if err != nil {
        log.Fatal(err)
    } else {
        fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
    }
}
```

----结束

5.7.2.5 关闭数据库连接

操作完成后，可以使用(db *DB)Close()方法关闭数据库连接。

假设连接句柄为db，则关闭数据库连接的操作如下：

```
db.Close()
```

5.7.3 典型应用开发示例

本章节的示例演示连接数据库、建表、插入数据并返回结果的常规操作。示例一以单ip:port为例，示例二以多ip:port为例。

示例一

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "database/sql"
    "fmt"
    "os"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "log"
    "strings"
    "time"
)

// 以下代码以单ip:port为例，本示例以IP地址、端口号、用户名、密码保存在环境变量中为例，运行本示例前请
// 先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）。
func main() {
    hostip := os.Getenv("GOHOSTIP") //GOHOSTIP为写入环境变量的IP地址。
    port := os.Getenv("GOPORT")    //GOPORT为写入环境变量的端口号。
```

```
username := os.Getenv("GOUSRNAME") //GOUSRNAME为写入环境变量的用户名。
passwd := os.Getenv("GOPASSWD") //GOPASSWD为写入环境变量的密码。
str := "host=" + hostip + " port=" + port + " user=" + username + " password=" + passwd + "
dbname=gaussdb sslmode=disable" //DSN连接串。用户需要自行设置dbname和sslmode关键字的取值。
//str := "gaussdb://" + username + ":" + passwd + "@" + hostip + ":" + port + "/"gaussdb?
sslmode=disable" //URL连接串。用户需要自行设置数据库名称和sslmode的取值。
db, err := sql.Open("gaussdb", str)
if err != nil {
    log.Fatal(err) //执行出错时，使用标准库log的Fatal方法输出错误信息,并同时调用os.Exit(1)终止程序。
}
defer db.Close() //延迟执行db.Close(), 确保在数据库操作完毕后关闭连接。
err = db.Ping() //验证是否成功连接数据库。
if err != nil {
    log.Fatal(err)
}

//定义包含4个SQL语句的字符串切片。
sqls := []string{
    "DROP TABLE IF EXISTS testExec", //若testExec表存在则删除。
    "CREATE TABLE testExec(f1 int, f2 varchar(20), f3 number, f4 timestampz, f5 boolean)", //创建testExec
    "INSERT INTO testExec VALUES(1, 'abcdefg', 123.3, '2022-02-08 10:30:43.31 +08', true)", //插入一条数
    "INSERT INTO testExec VALUES(:f1, :f2, :f3, :f4, :f5)", //参数化插入语句。
}

//数据切片定义。
intF1 := []int{2, 3, 4, 5, 6}
intF2 := []string{"hello world", "华为", "北京", "nanjing", "研究所"}
intF3 := []float64{641.43, 431.54, 5423.52, 665537.63, 6503.1}
intF4 := []time.Time{
    time.Date(2022, 2, 8, 10, 35, 43, 623431, time.Local),
    time.Date(2022, 2, 10, 19, 11, 54, 353431, time.Local),
    time.Date(2022, 2, 12, 6, 11, 15, 636431, time.Local),
    time.Date(2022, 2, 14, 4, 51, 22, 747653, time.Local),
    time.Date(2022, 2, 16, 13, 45, 55, 674636, time.Local),
}
intF5 := []bool{false, true, false, true, true}

//循环结构，使用range关键字遍历sqls切片。
for _, s := range sqls {
    if strings.Contains(s, ":f") { //判断是否参数化SQL。
        for i, _ := range intF1 { //遍历数据索引。
            _, err := db.Exec(s, intF1[i], intF2[i], intF3[i], intF4[i], intF5[i]) //执行参数化插入。
            if err != nil {
                log.Fatal(err)
            }
        }
    } else { //非参数化SQL。
        _, err = db.Exec(s) //直接执行DDL。
        if err != nil {
            log.Fatal(err)
        }
    }
}

var f1 int
var f2 string
var f3 float64
var f4 time.Time
var f5 bool
row := db.QueryRow("SELECT * FROM testExec") //查询单行数据。
err = row.Scan(&f1, &f2, &f3, &f4, &f5) //将这一行数据扫描到指定的变量。
if err != nil {
    log.Fatal(err)
} else {
    fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
}
```

```
rows, err := db.Query("SELECT * FROM testExec WHERE f1 > :1", 1) //执行带条件的查询。
if err != nil {
    log.Fatal(err)
}
defer rows.Close() //使用defer确保rows在函数结束时被关闭。
//使用for循环和type Rows接口的Next和Scan方法逐行将数据扫描到指定的变量。
for rows.Next() {
    err = rows.Scan(&f1, &f2, &f3, &f4, &f5)
    if err != nil {
        log.Fatal(err)
    } else {
        fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
    }
}
}
```

示例运行结果如下：

```
f1:1, f2:abcdefg, f3:123.3, f4:2022-02-08 10:30:43.31 +0800 CST, f5:true
f1:2, f2:hello world, f3:641.43, f4:2022-02-08 10:35:43.000623 +0800 CST, f5:false
f1:3, f2:华为, f3:431.54, f4:2022-02-10 19:11:54.000353 +0800 CST, f5:true
f1:4, f2:北京, f3:5423.52, f4:2022-02-12 06:11:15.000636 +0800 CST, f5:false
f1:5, f2:nanjing, f3:665537.63, f4:2022-02-14 04:51:22.000748 +0800 CST, f5:true
f1:6, f2:研究所, f3:6503.1, f4:2022-02-16 13:45:55.000675 +0800 CST, f5:true
```

示例二

```
package main
//依赖包根据环境中依赖包路径设置。
import (
    "context"
    "database/sql"
    "fmt"
    "os"
    _ [go驱动模块名] //请根据go驱动实际模块名替换。
    "log"
    "strings"
    "time"
)
// 以下代码以多ip:port为例，本示例以IP地址、端口号、用户名、密码保存在环境变量中为例，运行本示例前请
// 先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）。
func main() {
    ctx := context.Background()
    ctx2SecondTimeout, cancelFunc2SecondTimeout := context.WithTimeout(ctx, 2*time.Second)
    defer cancelFunc2SecondTimeout()
    hostip1 := os.Getenv("GOHOSTIP1") //GOHOSTIP1为写入环境变量的IP地址。
    hostip2 := os.Getenv("GOHOSTIP2") //GOHOSTIP2为写入环境变量的IP地址。
    hostip3 := os.Getenv("GOHOSTIP3") //GOHOSTIP3为写入环境变量的IP地址。
    port1 := os.Getenv("GOPORT1") //GOPORT1为写入环境变量的端口号。
    port2 := os.Getenv("GOPORT2") //GOPORT2为写入环境变量的端口号。
    username := os.Getenv("GOUSRNAME") //GOUSRNAME为写入环境变量的用户名。
    passwd := os.Getenv("GOPASSWD") //GOPASSWD为写入环境变量的密码。

    str := "host="+hostip1+","+hostip2+","+hostip3+" port="+port1+","+port2+" user="+username+"
password="+passwd+" dbname=gaussdb sslmode=disable" //DSN连接串。用户需要自行设置dbname和
sslmode关键字的取值。
    //str := "gaussdb://" + username + ":" + passwd + "@" + hostip1 + ":" + port1 + "," + hostip2 + ":" + port2 + "," + hostip3 + "/"
    //gaussdb?sslmode=disable" //URL连接串。用户需要自行设置数据库名称和sslmode的取值。
    db, err := sql.Open("gaussdb", str)
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close() //延迟执行db.Close()，确保在数据库操作完毕后关闭连接。

    // Ping database connection with 2 second timeout
    err = db.PingContext(ctx2SecondTimeout) //验证是否成功连接数据库。
    if err != nil {
        log.Fatal(err)
    }
}
```



```
//定义包含4个SQL语句的字符串切片。
sqls := []string{
    "DROP TABLE IF EXISTS testExecContext",
    "CREATE TABLE testExecContext(f1 int, f2 varchar(20), f3 number, f4 timestampz, f5 boolean)", //建表。
    "INSERT INTO testExecContext VALUES(1, 'abcdefg', 123.3, '2022-02-08 10:30:43.31 +08', true)", //插入一条数据。
    "INSERT INTO testExecContext VALUES(:f1, :f2, :f3, :f4, :f5)", //参数化插入语句。
}

//数据切片定义。
intF1 := []int{2, 3, 4, 5, 6}
intF2 := []string{"hello world", "华为", "北京2022冬奥会", "nanjing", "研究所"}
intF3 := []float64{641.43, 431.54, 5423.52, 665537.63, 6503.1}
intF4 := []time.Time{
    time.Date(2022, 2, 8, 10, 35, 43, 623431, time.Local),
    time.Date(2022, 2, 10, 19, 11, 54, 353431, time.Local),
    time.Date(2022, 2, 12, 6, 11, 15, 636431, time.Local),
    time.Date(2022, 2, 14, 4, 51, 22, 747653, time.Local),
    time.Date(2022, 2, 16, 13, 45, 55, 674636, time.Local),
}
intF5 := []bool{false, true, false, true, true}

//循环结构，使用range关键字遍历sqls切片。
for _, s := range sqls {
    if strings.Contains(s, ":") { //判断是否参数化SQL。
        for i, _ := range intF1 { //遍历数据索引。
            _, err := db.ExecContext(ctx2SecondTimeout, s, intF1[i], intF2[i], intF3[i], intF4[i], intF5[i]) //执行参数化插入。
            if err != nil {
                log.Fatal(err)
            }
        }
    } else { //非参数化SQL。
        _, err = db.ExecContext(ctx2SecondTimeout, s) //直接执行DDL。
        if err != nil {
            log.Fatal(err)
        }
    }
}

var f1 int
var f2 string
var f3 float64
var f4 time.Time
var f5 bool
row := db.QueryRowContext(ctx2SecondTimeout, "SELECT * FROM testExecContext") //查询单行数据。
err = row.Scan(&f1, &f2, &f3, &f4, &f5) //将这一行数据扫描到指定的变量。
if err != nil {
    log.Fatal(err)
} else {
    fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
}

rows, err := db.QueryContext(ctx2SecondTimeout, "SELECT * FROM testExecContext WHERE f1 > :1", 1)
if err != nil {
    log.Fatal(err)
}
defer rows.Close() //使用defer确保rows在函数结束时被关闭。

//使用for循环和type Rows接口的Next和Scan方法逐行将数据扫描到指定的变量。
for rows.Next() {
    err = rows.Scan(&f1, &f2, &f3, &f4, &f5)
    if err != nil {
        log.Fatal(err)
    } else {
        fmt.Printf("f1:%v, f2:%v, f3:%v, f4:%v, f5:%v\n", f1, f2, f3, f4, f5)
    }
}
```

```
}  
}
```

示例运行结果如下：

```
f1:1, f2:abcdefg, f3:123.3, f4:2022-02-08 10:30:43.31 +0800 CST, f5:true  
f1:2, f2:hello world, f3:641.43, f4:2022-02-08 10:35:43.000623 +0800 CST, f5:false  
f1:3, f2:华为, f3:431.54, f4:2022-02-10 19:11:54.000353 +0800 CST, f5:true  
f1:4, f2:北京2022冬奥会, f3:5423.52, f4:2022-02-12 06:11:15.000636 +0800 CST, f5:false  
f1:5, f2:nanjing, f3:665537.63, f4:2022-02-14 04:51:22.000748 +0800 CST, f5:true  
f1:6, f2:研究所, f3:6503.1, f4:2022-02-16 13:45:55.000675 +0800 CST, f5:true
```

5.7.4 Go 接口参考

5.7.4.1 type DB

type DB接口是database/sql包中的核心接口之一，它定义了一系列的方法，用于执行SQL查询、事务管理等操作，如表5-119所示。

表 5-119 type DB 接口的常用方法

方法	描述	返回值类型
(db *DB)Begin()	开启一个事务，事务的隔离级别由驱动决定。	*Tx, error
(db *DB)BeginTx(ctx context.Context, opts *TxOptions)	开启一个给定事务隔离级别的事务，给定的上下文会一直使用到事务提交或回滚为止。若上下文被取消，那么database/sql包将会对事务进行回滚。	*Tx, error
(db *DB)Close()	关闭数据库并释放所有已打开的资源。	error
(db *DB)Exec(query string, args ...interface{})	执行一个不返回数据行的操作。	Result, error
(db *DB)ExecContext(ctx context.Context, query string, args ...interface{})	在给定上下文中，执行一个不返回数据行的操作。	Result, error
(db *DB)Ping()	检查数据库连接是否仍然有效，并在有需要时建立一个连接。	error
(db *DB)PingContext(ctx context.Context)	在给定上下文中，检查数据库连接是否仍然有效，并在有需要时建立一个连接。	error
(db *DB)Prepare(query string)	为以后的查询或执行创建一个预处理语句。	*Stmt, error
(db *DB)PrepareContext(ctx context.Context, query string)	在给定的上下文中，为以后的查询或执行创建一个预处理语句。	*Stmt, error

方法	描述	返回值类型
(db *DB)Query(query string, args ...interface{})	执行一个查询并返回多个数据行。	*Rows, error
(db *DB)QueryContext(ctx context.Context, query string, args ...interface{})	在给定的上下文中，执行一个查询并返回多个数据行。	*Rows, error
(db *DB)QueryRow(query string, args ...interface{})	执行一个只返回一个数据行的查询。	*Row
(db *DB)QueryRowContext(ctx context.Context, query string, args ...interface{})	在给定上下文中，执行一个只返回一个数据行的查询。	*Row

参数说明

表 5-120 常用方法的参数说明

参数	参数说明
ctx	表示给定的上下文。
query	被执行的SQL语句。
args	被执行SQL语句需要绑定的参数。支持按位置绑定和按名称绑定，参见如下 示例 。
opts	事务隔离级别和事务访问模式。其中事务隔离级别（opts.Isolation）支持范围为sql.LevelReadUncommitted、sql.LevelReadCommitted、sql.LevelRepeatableRead、sql.LevelSerializable。事务访问模式（opts.ReadOnly）支持范围为true（read only）和false（read write）。

须知

- Query类方法Query()、QueryContext()、QueryRow()、QueryRowContext()通常用于查询语句，如SELECT语句。操作语句使用Exec()类方法执行，若非查询语句通过Query类方法执行，则执行结果可能与预期不符，因此不建议使用Query类方法执行非查询语句，例如UPDATE、INSERT等。
- 使用Query类方法执行查询语句的结果需要通过type Rows中Next()方法获取，若不通过Next()方法获取，可能会产生不可预期的错误。

示例

```
// 前置条件是已经连接数据库，连接对象为db。  
// 建表
```

```
_, err = db.Exec("CREATE TABLE test_bound(id int, name text)")

// 按位置绑定
_, err = db.Exec("INSERT INTO test_bound(id, name) VALUES(:1, :2)", 1, "张三")
if err != nil {
    log.Fatal(err)
}

// 按名称绑定
_, err = db.Exec("INSERT INTO test_bound(id, name) VALUES(:id, :name)", sql.Named("id", 1),
sql.Named("name", "张三"))
if err != nil {
    log.Fatal(err)
}
```

5.7.4.2 type Stmt

type Stmt接口主要用于预编译SQL语句，常用方法如表5-121所示。

表 5-121 type Stmt 接口的常用方法

方法	描述	返回值类型
(s *Stmt)Close()	关闭给定的预处理语句。	error
(s *Stmt)Exec(args ...interface{})	使用给定的参数执行预处理语句，并返回一个Result值。支持PBE特性。 PBE，即Prepare-Bind-Execute，是发送和执行查询的一种方式，DN可通过复杂查询协议接收PBE报文执行语句。 说明 - 预编译语句的占位符可以为“\$”也可以是“?”。 - 预编译语句的占位符数量由数据库限制，当表字段超过数据库限制或者与当前表字段数目不匹配时，由服务端返回错误。 - PBE特性支持增、删、改操作，批量操作时U报文最大长度限制为1023MB。超出长度限制会报错： bind message length XXX too long. This can be caused by very large or incorrect length specifications on InputStream parameters. - 插入一条数据的时候，使用PBE对比单插接口（conn.simpleExec，简单执行语句）有大幅的性能下降，建议优先使用单插接口而不是PBE。 - 驱动底层错误处理重构，PBE性能对比原来下降幅度在5%以内。	Result, error
(s *Stmt)ExecContext(ctx context.Context, args ...interface{})	在给定的上下文中，使用给定的参数执行预处理语句，并返回一个Result值。	Result, error

(s *Stmt)Query(args ...interface{})	使用给定的参数执行预处理语句，并以 *Rows形式返回查询结果。	*Rows, error
(s *Stmt)QueryContext(ctx context.Context, args ...interface{})	在给定的上下文中，使用给定的参数执行预处理语句，并以 *Rows形式返回查询结果。	*Rows, error
(s *Stmt)QueryRow(args ...interface{})	使用给定的参数执行预处理语句，并返回一个 *Row作为结果。	*Row
(s *Stmt)QueryRowContext (ctx context.Context, args ...interface{})	在给定的上下文中，使用给定的参数执行预处理语句，并返回一个 *Row作为结果。	*Row

参数说明

表 5-122 常用方法的参数说明

参数	参数说明
ctx	表示给定的上下文。
args	被执行SQL语句需要绑定的参数。支持按位置绑定和按名称绑定，参见 示例 。

须知

1.

Query类方法Query()、QueryContext()、QueryRow()、QueryRowContext()通常用于查询语句，如SELECT语句。操作语句使用Exec()类方法执行，若非查询语句通过Query类方法执行，则执行结果可能与预期不符，因此不建议使用Query类方法执行非查询语句，例如UPDATE、INSERT等。
2.

使用Query类方法执行查询语句的结果需要通过type Rows中Next()方法获取，若不通过Next()方法获取，可能会产生不可预期的错误。

5.7.4.3 type Tx

type Tx接口定义了一系列方法，用于管理数据库事务，常用方法如[表5-123](#)所示。

表 5-123 type Tx 接口的常用方法

方法	描述	返回值类型
(tx *Tx)Commit()	提交事务。	error

(tx *Tx)Exec(query string, args ...interface{})	执行一个不返回数据行的操作。	Result, error
(tx *Tx)ExecContext(ctx context.Context, query string, args ...interface{})	在给定上下文中，执行一个不返回数据行的操作。	Result, error
(tx *Tx)Prepare(query string)	为以后的查询或执行创建一个预处理语句。返回的语句将在事务中执行，并且一旦事务被提交或回滚就不能再使用。	*Stmt, error
(tx *Tx)PrepareContext(ctx context.Context, query string)	为以后的查询或执行创建一个预处理语句。返回的语句将在事务中执行，并且一旦事务被提交或回滚就不能再使用。 给定的上下文将用于预备阶段，而不是事务执行阶段。该方法返回的语句将在事务上下文中执行。	*Stmt, error
(tx *Tx)Query(query string, args ...interface{})	执行一个返回数据行的查询。	*Rows, error
(tx *Tx)QueryContext(ctx context.Context, query string, args ...interface{})	在给定上下文中，执行一个返回数据行的查询。	*Rows, error
(tx *Tx)QueryRow(query string, args ...interface{})	执行一个只返回一个数据行的查询。	*Row
(tx *Tx)QueryRowContext(ctx context.Context, query string, args ...interface{})	在给定上下文中，执行一个只返回一个数据行的查询。	*Row
(tx *Tx)Rollback()	事务回滚。	error
(tx *Tx)Stmt(stmt *Stmt)	为已有的语句返回一个事务专用的预处理语句。 示例： str, err := db.Prepare("INSERT INTO t1 VALUES(:1, :2)") tx, err := db.Begin() res, err := tx.Stmt(str).Exec(1, "aaa")	*Stmt
(tx *Tx)StmtContext(ctx context.Context, stmt *Stmt)	在给定上下文中，为已有的语句返回一个事务专用的预处理语句。	*Stmt

参数说明

表 5-124 常用方法的参数说明

参数	参数说明
ctx	表示给定的上下文。
query	被执行的SQL语句。
args	被执行SQL语句需要绑定的参数。支持按位置绑定和按名称绑定，参见 示例 。
stmt	已有的预处理语句，一般指prepared语句返回的预处理语句。

须知

- Query类方法Query()、QueryContext()、QueryRow()、QueryRowContext()通常用于查询语句，如SELECT语句。操作语句使用Exec()类方法执行，若非查询语句通过Query类方法执行，则执行结果可能与预期不符，因此不建议使用Query类方法执行非查询语句，例如UPDATE、INSERT等。
- 使用Query类方法执行查询语句的结果需要通过[type Rows](#)中Next()方法获取，若不通过Next()方法获取，可能会产生不可预期的错误。

5.7.4.4 type Rows

type Rows接口用于处理多行结果，提供遍历和数据扫描等方法，如[表5-125](#)所示。

表 5-125 type Rows 接口的常用方法

方法	描述	返回值类型
(rs *Rows)Close()	关闭Rows，停止对数据集的迭代。	error
(rs *Rows)ColumnTypes()	返回列类型。	[]*ColumnType, error
(rs *Rows)Columns()	返回列名。	[]string, error
(rs *Rows)Err()	返回迭代过程中出现的任何错误。	error
(rs *Rows)Next()	为Scan方法准备好下一个待读取的数据行。如果有下个数据行的结果集，则返回true，否则返回false。	boolean
(rs *Rows)Scan(dest ...interface{})	将当前被遍历的数据行的各个列复制到dest指向的值中。	error
(rs *Rows)NextResultSet()	判断是否有进一步的结果集。	boolean

参数说明

表 5-126 常用方法的参数说明

参数	参数说明
dest	查询列需要被复制到该参数指向的值。

5.7.4.5 type Row

type Row接口用于处理单行结果，提供数据扫描等方法，如表5-127所示。

表 5-127 type Row 接口的常用方法

方法	描述	返回值类型
(r *Row)Scan(dest ...interface{})	将当前数据行中的列复制到dest指向的值中。	error
(r *Row)Err()	返回执行过程中出现的错误。	error

参数说明

表 5-128 常用方法的参数说明

参数	参数说明
dest	查询列需要被复制到该参数指向的值。

5.7.4.6 type ColumnType

type ColumnType接口主要用于描述数据库表中的列类型信息，常用方法如表5-129所示。

表 5-129 type ColumnType 接口的常用方法

方法	描述	返回值类型
(ci *ColumnType)DatabaseTypeName()	返回列类型的数据库系统名称。返回空字符串表示该驱动类型名字并未被支持。	error
(ci *ColumnType)DecimalSize()	返回小数类型的范围和精度。返回值ok的值为false时，说明给定的类型不可用或者不支持。	precision, scale int64, ok boolean

(ci *ColumnType)Length()	返回数据列类型长度。返回值ok的值为false时，说明给定的类型是非可变长度的类型。	length int64, ok boolean
(ci *ColumnType)ScanType()	返回一种Go类型，该类型能够在Rows.scan进行扫描时使用。	reflect.Type
(ci *ColumnType)Name()	返回数据列的名字。	string

5.7.4.7 type Result

type Result接口主要用于表示执行SQL语句后的结果，常用方法如[表5-130](#)所示。

表 5-130 type Result 接口的常用方法

方法	描述	返回值类型
(res Result)RowsAffected()	获取受影响的行数。在执行UPDATE、DELETE等修改操作时非常有用，可以帮助用户了解有多少行被成功修改。	int64, error

5.8 基于 ecpg 开发

5.8.1 开发流程

ecpg整体开发流程图如[图5-20](#)所示。

包括准备ecpg开发相关组件、嵌入式SQL-C程序开发、预处理嵌入式SQL-C程序、编译C程序、执行C程序。

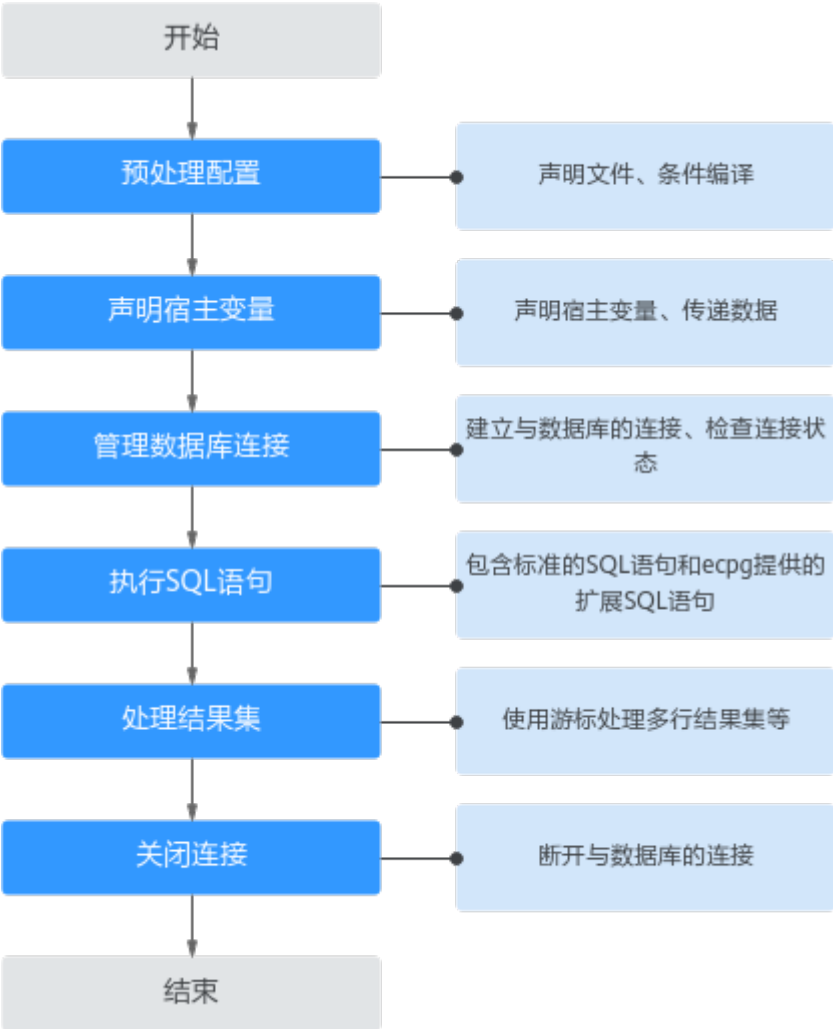
图 5-20 ecpg 整体开发流程



其中嵌入式SQL-C程序开发流程图如图5-21所示。

包括预处理配置、声明宿主变量、管理数据库连接、执行SQL语句、处理结果集、关闭连接。

图 5-21 嵌入式 SQL-C 程序开发流程



5.8.2 准备 ecpg 开发相关组件

进行ecpg应用程序开发前，需确保开发环境已正确配置GaussDB客户端组件及编译工具链。

ecpg的开发依赖于特定的预处理器、动态链接库及头文件。核心组件的使用路径及编译引用规则如下表所示：

表 5-131 ecpg 核心组件路径和编译引用规则

组件分类	核心文件/库名	使用路径	编译/引用参数	功能
预处理器	ecpg	\$GAUSSH OME/bin	无	将.pgc源码文件转化为标准 .c文件的二进制工具

组件分类	核心文件/库名	使用路径	编译/引用参数	功能
通信与控制库	libecpg.so、 libecpg.so.6和 libecpg.so.6.4	\$GAUSSH OME/lib	-lecp	提供数据库连接管理、SQL语句执行及事务控制的底层实现。
类型处理库	libpgtypes.so、 libpgtypes.so.6和 libpgtypes.so.6.4	\$GAUSSH OME/lib	-lpgtypes	提供数值、日期、时间戳、区间等复杂数据类型的运算与转换能力。
依赖头文件	sqlca.h、sqllda.h等	\$GAUSSH OME/ include/ ecpg	-I \$GAUSSH OME/ include/ ecpg	提供预编译期所需的宏定义、数据结构声明。

编译预处理工具准备：

- ecpg开发需要GCC（GNU Compiler Collection）编译器进行编译，建议使用7.3.0版本。
- ecpg客户端工具支持的操作系统与CPU架构（x86/ARM）与服务端发布的支持矩阵保持一致。

已知问题和解决方法：

动态库运行期寻址：操作系统默认不扫描GaussDB库目录。运行编译后的二进制文件前，必须将\$GAUSSHOME/lib注入系统环境变量LD_LIBRARY_PATH中。

5.8.3 嵌入式 SQL-C 程序开发

5.8.3.1 核心概念简介

嵌入式SQL-C程序开发，是指利用ecpg预编译器，在标准的C语言代码中直接插入SQL语句的编程方法。通过这种方式，开发者可以在单一源文件中，结合C语言的过程控制能力与SQL的数据处理能力，高效构建数据库应用程序。

在开始编写具体的嵌入式代码之前，了解以下基础概念将有助于您更好地组织程序：

C 预处理器与 ecpg 预编译器

在嵌入式开发中，您的源码（.pgc）实际上要经历两道串行且独立的预处理工序。理解它们的分工至关重要：

1. ecpg预编译器作为第一道工序运行。它专门扫描源代码，提取所有以EXEC SQL引导的语句，并将其转换为调用底层库的标准C函数，最终生成一个临时.c文件。
2. 标准的C预处理器（包含在C编译器中）才开始接手这个生成的.c文件。它负责处理原生的C指令，如展开#include头文件和#define宏定义。

关于ecpg预编译器的详细功能，可以参考[ecpg预处理、编译、执行](#)。

静态 SQL 和动态 SQL

大多数应用程序设计用于处理静态SQL语句。在这种情况下，您在编译前就已经明确知道要执行哪些SQL命令、操作哪些表和列。然而，某些程序（如通用查询工具）可能直到运行时才能确定具体的SQL结构。动态SQL就是一种允许程序在运行时动态构建、接受并执行SQL语句的高级技术。

宿主变量与数据类型

宿主变量是在C程序中声明的变量，用于在应用程序和数据库之间传递真实数据。例如，您可以将数据库查询的结果直接提取并存入宿主变量中，或者将宿主变量中的值作为参数插入到数据库表中。它是连接C语言数据空间与SQL语句执行的桥梁。

由于C语言的原生数据类型与数据库的SQL数据类型存在差异，宿主变量的使用强依赖于类型映射机制。开发者不仅需要利用宿主变量提取或插入数据，还必须遵循严格的映射规则，将数据库的特殊类型（如时间戳、字符串）安全地转换为C语言的初级或非初级（如结构体、数组）数据类型。

游标（Cursors）

数据库的查询操作（SELECT）通常会返回多行数据，而C语言的普通变量在特定时刻只能保存单个值。游标就是用来解决这一差异的机制。通过声明游标，您的程序可以在结果集上打开一个“数据窗口”，然后按顺序逐行提取（FETCH）数据并进行处理。

描述符区域（SQLDA）

当您使用最复杂的动态SQL技术时（即在编写代码时不仅不知道具体的SQL语句，甚至连查询会返回多少列、是什么数据类型都不知道），普通的宿主变量将无法满足需求。此时需要使用描述符区域（SQLDA）。它是一个高级的内存数据结构，允许程序在运行时动态地解析、分配和存储这些未知结构的数据。

错误处理机制

嵌入式SQL语句在执行后，可能会成功、失败或产生警告。ecpg提供了两种主要的处理机制：SQLCA（SQL通信区）是一个全局数据结构，它向程序传递运行时的具体状态反馈（如错误码、受影响的行数），程序可根据反馈执行不同操作；WHENEVER语句则允许您指定当系统检测到错误或警告时要自动执行的动作，例如跳转到特定代码段、调用函数或直接退出程序。

5.8.3.2 声明宿主变量

宿主变量是C程序与SQL交互的桥梁，必须在特定代码块中声明。

语法如下：

声明区域以下面的代码开始：
EXEC SQL BEGIN DECLARE SECTION;

以下面的代码结束：
EXEC SQL END DECLARE SECTION;

在此之间，必须有常规的C语言变量声明，比如：

```
int x = 4;  
char foo[16], bar[16];
```

须知

- 标记段代码开始和结束之间声明的宿主变量类型必须为当前支持的数据类型，请参见表5-145。
- 对于简单的单一变量，您可以省略完整的声明段包裹，通过指令前缀进行隐式声明：EXEC SQL int i = 4。
- 不在SQL命令里使用的变量可以在特殊的声明段外面声明。只有在声明区域内声明的变量才能在SQL语句中使用。
- 若宿主变量涉及自定义的结构体或联合体，其完整的类型定义也必须包裹在声明段中，否则ecpg预编译器将无法解析其底层内存结构。

5.8.3.3 管理数据库连接

连接数据库

使用如下语句连接数据库：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

target可以通过如下方法声明，斜体部分为变量，请根据实际情况进行修改：

- *dbname[@hostname][:port]*
- *tcp:gaussdb://hostname[:port]/dbname[?options]*
- *unix:gaussdb://hostname[:port]/dbname[?options]*

ecpg支持IPv6格式，需要在hostname两端添加方括号“[]”。

IPv6格式示例如下：

```
db1@[::1]
db1@[::1]:5432
tcp:gaussdb://[::1]/db1
tcp:gaussdb://[::1]:5432/db1
unix:gaussdb://[::1]/db1
unix:gaussdb://[::1]:5432/db1
```

说明

目前ecpg支持的IPv6格式如下：

- 简写大写格式
- 简写小写格式
- 非简写大写格式
- 非简写小写格式

IPv6的回环地址为::1或者0:0:0:0:0:0:1，因此Unix协议在IPv6下只支持这两种格式。

声明连接用户名的方法如下：

- *username/password*
- *username SQLIDENTIFIED BY password*
- *username USING password*

如上所述，参数username以及password可以是一个SQL标识符、一个SQL字符串或一个对字符变量的引用。

connection-name表示连接名，如果一个程序仅使用一个连接，可以省略connection-name；若使用多个连接，则最近打开的连接将作为当前连接。

说明

- 若连接语句中指定了ip-port，则必须指定username/password，该规则由内核通信认证所决定。若不指定ip-port，则通过本地\$PGPORT(UDS协议)进行通信。
- 若客户连接时使用SSL安全协议，则需要使用tcp:gaussdb://hostname[:port][/dbname][?options]连接格式，在options选项中配置sslmode=disable/require。

简单示例如下：

```
#include <stdlib.h>
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:gaussdb://sql.mydomain.com/mydb AS myconnection USER username;

EXEC SQL BEGIN DECLARE SECTION;

/* 此处target、user、passwd应从环境变量或配置文件读取，环境变量需用户自己按需配置；非环境变量情况下可直接赋值字符串 */
const char *target = getenv("EXAMPLE_TARGET_ENV");
const char *user = getenv("EXAMPLE_USERNAME_ENV");
const char *passwd = getenv("EXAMPLE_PASSWD_ENV");
EXEC SQL END DECLARE SECTION;
...

EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* 或者 EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

完整使用示例，请参见[CONNECT](#)中的“连接语法使用示例”。

说明

- 连接目标的格式未在SQL标准中说明，因此若要开发可移植的应用，可使用上述最后一个例子的方法将连接目标字符串封装在某个变量里。
- ecpg兼容性请参见[ecpg兼容](#)。

管理连接

默认情况下，嵌入式SQL-C程序中的语句均在当前连接（即最近创建的连接）上执行。当应用需要并发处理多个数据库实例时，可通过以下两种方式进行连接路由：

- 方法1：为每个SQL语句明确指定一个连接：
EXEC SQL AT connection-name SELECT ...;
适用于应用程序以混合顺序使用多个连接的情况。
- 方法2：执行一个语句来切换连接：
EXEC SQL SET CONNECTION connection-name;
适用于多个语句在同一个连接上执行的场景。

管理连接示例如下：

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
char dbname[1024];
EXEC SQL END DECLARE SECTION;

int main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
```

```
/* 这个查询将在最近连接的数据库 "testdb3" 中执行 */
EXEC SQL SELECT current_database() INTO :dbname;
printf("current=%s (should be testdb3)\n", dbname);

/* 使用 "AT" 在 "testdb2" 中运行一个查询 */
EXEC SQL AT con2 SELECT current_database() INTO :dbname;
printf("current=%s (should be testdb2)\n", dbname);

/* 切换当前连接到 "testdb1" */
EXEC SQL SET CONNECTION con1;

EXEC SQL SELECT current_database() INTO :dbname;
printf("current=%s (should be testdb1)\n", dbname);

EXEC SQL DISCONNECT ALL;
return 0;
}
```

运行结果如下：

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

说明

- 多线程模式下不支持不同线程使用同一连接名，每个线程连接名唯一。
- 连接的建立和关闭需要在同一进程或线程中进行。
- 如果应用程序创建多个执行线程，它们不能共享同一个连接，必须明确控制对连接的访问（比如利用互斥量）或者每个线程使用一个唯一 连接。

5.8.3.4 执行 SQL 语句

嵌入式SQL-C程序支持直接执行静态SQL语句（如SELECT、INSERT），也支持执行动态SQL语句。动态SQL语句详情请参考[执行动态SQL语句](#)。本节主要介绍执行静态SQL语句。

静态SQL语句执行步骤如下：

步骤1 创建表结构：

```
EXEC SQL CREATE TABLE foo (a int, b varchar);
```

步骤2 插入单行记录：

```
EXEC SQL INSERT INTO foo VALUES (5, 'abc');
```

步骤3 删除指定记录：

```
EXEC SQL DELETE FROM foo WHERE a = 5;
```

步骤4 更新表数据：

```
EXEC SQL UPDATE foo SET b = 'gdp' WHERE a = 7;
```

步骤5 单行查询数据：

```
EXEC SQL SELECT a INTO :var_a FROM foo WHERE b = 'def';
```

通过INTO关键字将查询结果直接映射到绑定的宿主变量中。

----结束

完整使用示例如下：

前提条件：需要提前创建testdb数据库。

```
#include <stdlib.h>
#include <string.h>
```



```
#include <stdio.h>

int main ()
{
    ECPGdebug (1, stderr);

    EXEC SQL BEGIN DECLARE SECTION;
    int var_a;
    EXEC SQL END DECLARE SECTION;

    /* 提前创建testdb */
    EXEC SQL CONNECT TO testdb;

    /* 创建一个表 */
    EXEC SQL CREATE TABLE foo (a int, b varchar);

    /* 插入数据 */
    EXEC SQL INSERT INTO foo VALUES (5, 'abc');
    EXEC SQL INSERT INTO foo VALUES (6, 'def');
    EXEC SQL INSERT INTO foo VALUES (7, 'ghi');

    /* 删除一行 */
    EXEC SQL DELETE FROM foo WHERE a = 5;

    /* 更新表数据 */
    EXEC SQL UPDATE foo SET b = 'gdp' WHERE a = 7;

    /* 单行查询表数据 */
    EXEC SQL SELECT a INTO :var_a FROM foo WHERE b = 'def';

    /* 打印查询结果 */
    printf("select res is %d\n", var_a);
    EXEC SQL DISCONNECT;
    return 0;
}
```

运行结果如下（[39841]是线程ID，随机生成）：

```
[39841]: ECPGdebug: set to 1
[39841]: ECPGconnect: opening database testdb on <DEFAULT> port <DEFAULT>
[39841]: ecpg_execute on line 17: query: create table foo ( a int , b varchar ); with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 17: using PQexec
[39841]: ecpg_execute on line 17: OK: CREATE TABLE
[39841]: ecpg_execute on line 20: query: insert into foo values ( 5 , 'abc' ); with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 20: using PQexec
[39841]: ecpg_execute on line 20: OK: INSERT 0 1
[39841]: ecpg_execute on line 21: query: insert into foo values ( 6 , 'def' ); with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 21: using PQexec
[39841]: ecpg_execute on line 21: OK: INSERT 0 1
[39841]: ecpg_execute on line 22: query: insert into foo values ( 7 , 'ghi' ); with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 22: using PQexec
[39841]: ecpg_execute on line 22: OK: INSERT 0 1
[39841]: ecpg_execute on line 25: query: delete from foo where a = 5; with 0 parameter(s) on connection
testdb
[39841]: ecpg_execute on line 25: using PQexec
[39841]: ecpg_execute on line 25: OK: DELETE 1
[39841]: ecpg_execute on line 28: query: update foo set b = 'gdp' where a = 7; with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 28: using PQexec
[39841]: ecpg_execute on line 28: OK: UPDATE 1
[39841]: ecpg_execute on line 31: query: select a from foo where b = 'def'; with 0 parameter(s) on
connection testdb
[39841]: ecpg_execute on line 31: using PQexec
[39841]: ecpg_execute on line 31: correctly got 1 tuples with 1 fields
[39841]: ecpg_get_data on line 31: RESULT: 6 offset: 4; array: no
```

```
select res is 6  
[39841]: ecpg_finish: connection testdb closed
```

事务管理

在执行SQL语句的过程中，通常包括事务管理。事务（Transaction）是数据库操作的基本单位，用于确保一组SQL语句要么全部成功执行，要么全部失败回滚。事务的四大特性（ACID）包括：

- 原子性（Atomicity）：事务中的所有操作要么全部完成，要么全部不完成。
- 一致性（Consistency）：事务执行前后，数据库状态保持一致。
- 隔离性（Isolation）：多个事务并发执行时，彼此互不干扰。
- 持久性（Durability）：事务提交后，对数据库的修改是永久性的。

在ecpg缺省模式下，语句只有在EXEC SQL COMMIT发出的时候才被提交。嵌入的SQL接口支持事务的自动提交（通过EXEC SQL SET AUTOCOMMIT TO ON语句设置自动提交）。在自动提交模式下，每条命令都是自动提交的，除非它们被包围在一个明确的事务块里。自动提交模式可以用EXEC SQL SET AUTOCOMMIT TO OFF语句关闭。

常见事务管理命令如下：

- EXEC SQL COMMIT：提交正在进行的事务。
- EXEC SQL ROLLBACK：回滚正在进行的事务。
- EXEC SQL SET AUTOCOMMIT TO ON：启动自动提交模式。
- EXEC SQL SET AUTOCOMMIT TO OFF：关闭自动提交模式，缺省模式。

5.8.3.5 处理结果集

- 返回单行结果的SELECT语句可以直接使用EXEC SQL执行，请参见[执行SQL语句](#)章节。

示例：

```
/* 首先建立一个表并插入数据 */  
EXEC SQL CREATE TABLE test_table (number1 integer, number2 integer);  
EXEC SQL INSERT INTO test_table (number1, number2) VALUES (2, 1);  
  
/* 查询结果为单行,:num 为宿主变量 */  
EXEC SQL SELECT number1 INTO :num FROM test_table WHERE number2 = 1;
```

- 若要处理多行结果集，则必须使用游标，如何使用游标请参考[使用游标](#)。

示例：

```
/* 首先建立一个表并插入数据 */  
EXEC SQL CREATE TABLE test_table (number1 integer, number2 integer);  
EXEC SQL INSERT INTO test_table (number1, number2) VALUES (2, 1);  
EXEC SQL INSERT INTO test_table (number1, number2) VALUES (3, 1);  
EXEC SQL INSERT INTO test_table (number1, number2) VALUES (4, 1);  
EXEC SQL INSERT INTO test_table (number1, number2) VALUES (5, 1);  
  
/* 定义宿主变量 */  
EXEC SQL BEGIN DECLARE SECTION;  
int v1;  
int v2;  
EXEC SQL END DECLARE SECTION;  
  
/* 声明游标 */  
EXEC SQL DECLARE test_bar CURSOR FOR SELECT number1, number2 FROM test_table ORDER BY  
number1;  
  
/* 打开游标 */  
EXEC SQL OPEN test_bar;
```

```
/* 当游标到达结果集末尾时跳出循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

/* 获取查询结果集 */
while(1)
{
    EXEC SQL FETCH NEXT FROM test_bar INTO :v1, :v2;
    printf("number1 = %d, number2 = %d\n",v1,v2);
}

/* 关闭游标 */
EXEC SQL CLOSE test_bar;
```

使用游标

使用游标可以检索出多行的结果集，应用程序必须声明一个游标并且从游标中抓取每一行数据。

步骤1 声明一个游标：

```
EXEC SQL DECLARE c CURSOR FOR SELECT * FROM tb1;
```

步骤2 打开游标：

```
EXEC SQL OPEN c;
```

步骤3 从游标中抓取一行数据：

```
EXEC SQL FETCH 1 IN c INTO :a, :str;
```

步骤4 关闭游标：

```
EXEC SQL CLOSE c;
```

----结束

说明

在开启GUC参数enable_ecpg_cursor_duplicate_operation（默认开启）的情况下，使用ecpg连接ORA兼容数据库时，允许通过如下方式重复打开/关闭游标：

```
EXEC SQL OPEN c;
EXEC SQL OPEN c;
EXEC SQL CLOSE c;
EXEC SQL CLOSE c;
```

更多游标的使用细节请参见[DECLARE](#)，关于FETCH命令的细节请参见《参考》中“SQL参考 > SQL语法 > F > FETCH”章节。

完整使用示例：

前提条件：需要提前创建testdb数据库。

```
#include <string.h>
#include <stdlib.h>

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    int *a = NULL;
    char *str = NULL;
    EXEC SQL END DECLARE SECTION;

    int count = 0;

    /* 提前创建testdb */
    EXEC SQL CONNECT TO testdb ;
    EXEC SQL SET AUTOCOMMIT TO OFF;
    EXEC SQL BEGIN;
```

```
EXEC SQL DROP TABLE IF EXISTS tb1;
EXEC SQL CREATE TABLE tb1 (id int, info text);
EXEC SQL INSERT INTO tb1 (id, info) SELECT generate_series(1, 100000), 'test';
EXEC SQL SELECT COUNT(*) INTO :a FROM tb1;
printf ("a is %d\n", *a);
EXEC SQL COMMIT;

/* 定义游标 */
EXEC SQL DECLARE c CURSOR FOR SELECT * FROM tb1;

/* 打开游标 */
EXEC SQL OPEN c;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
while(1) {
    /* 抓取数据 */
    EXEC SQL FETCH 1 IN c INTO :a, :str;
    count++;
    if (count == 100000) {
        printf("Fetch res: a is %d, str is %s", *a, str);
    }
}

/* 关闭游标 */
EXEC SQL CLOSE c;
EXEC SQL SET AUTOCOMMIT TO ON;
EXEC SQL DROP TABLE tb1;
EXEC SQL DISCONNECT;

ECPGfree_auto_mem();
return 0;
}
```

运行结果如下：

```
a is 100000
Fetch res: a is 100000, str is test
```

当cursor指向表的某一行时，可以使用**WHERE CURRENT OF cursor_name**语法更新或删除cursor当前指向的行。使用限制及约束请参见《参考》中“SQL参考 > SQL语法 > U > UPDATE”章节对此语法介绍。

完整使用示例：

```
#include <string.h>
#include <stdlib.h>

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    int va;
    int vb;
    EXEC SQL END DECLARE SECTION;

    int count = 0;

    /* 提前创建testdb */
    EXEC SQL CONNECT TO testdb;
    EXEC SQL SET AUTOCOMMIT TO OFF;
    EXEC SQL BEGIN;
    EXEC SQL DROP TABLE IF EXISTS t1;
    EXEC SQL CREATE TABLE t1 (c1 INT, c2 INT);
    EXEC SQL INSERT INTO t1 VALUES(generate_series(1, 10000), generate_series(1, 10000));
    EXEC SQL COMMIT;

    /* 定义游标 */
    EXEC SQL DECLARE cur1 CURSOR FOR SELECT * FROM t1 WHERE c1 < 100 FOR UPDATE;

    /* 打开游标 */
    EXEC SQL OPEN cur1;
```

```
/* 抓取数据 */
EXEC SQL FETCH 1 IN cur1 INTO :va, :vb;
printf("c1: %d, c2: %d\n", va, vb);

/* 使用 WHERE CURRENT OF 删除当前行 */
EXEC SQL DELETE FROM t1 WHERE CURRENT OF cur1;

/* 继续抓取数据 */
EXEC SQL FETCH 1 IN cur1 INTO :va, :vb;
EXEC SQL FETCH 1 IN cur1 INTO :va, :vb;
printf("c1: %d, c2: %d\n", va, vb);

/* 使用 WHERE CURRENT OF 更新当前行 */
EXEC SQL UPDATE t1 SET c2 = 21 WHERE CURRENT OF cur1;

/* 查询更新后的数据 */
EXEC SQL SELECT c2 INTO :vb FROM t1 WHERE c1 = :va;
printf("c1: %d, c2: %d\n", va, vb);

/* 关闭游标 */
EXEC SQL CLOSE cur1;

/* 恢复自动提交并清理 */
EXEC SQL SET AUTOCOMMIT TO ON;
EXEC SQL DROP TABLE t1;
EXEC SQL DISCONNECT;

ECPGfree_auto_mem();
return 0;
}
```

运行结果如下：

```
c1: 1, c2: 1
c1: 3, c2: 3
c1: 3, c2: 21
```

5.8.3.6 关闭连接

业务逻辑处理完毕后，必须显式关闭数据库连接以释放底层资源，防止服务端连接泄露。

使用如下语句关闭连接：

```
EXEC SQL DISCONNECT [connection];
```

根据不同的并发或多实例管理需求，[connection]参数可采用以下四种方式进行声明：

- connection-name（指定连接名）：精确关闭已命名的特定数据库连接。
- default（缺省）：关闭默认建立的未命名连接。
- current（当前）：关闭程序中最近一次处于活跃状态的连接。
- all（所有）：一次性安全清理当前程序建立的所有GaussDB会话连接。

5.8.3.7 嵌入式 SQL 语法参考

5.8.3.7.1 ALLOCATE DESCRIPTOR

功能描述

分配一个新命名的SQL描述符区域。

语法格式

```
ALLOCATE DESCRIPTOR name
```

参数说明

name

SQL描述符名称。大小写敏感，是一个SQL标识或者一个宿主变量。

示例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

相关链接

[DEALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#), [SET DESCRIPTOR](#)

5.8.3.7.2 CONNECT

功能描述

在客户端和SQL服务器之间建立连接。

语法格式

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
```

参数说明

- **connection_target**

以下形式指定连接的目标服务器：

- [*database_name*] [@*host*] [:*port*]：通过TCP/IP连接。
- *unix:gaussdb://host* [:*port*] / [*database_name*] [?
connection_option]：通过Unix域套接字连接。
- *tcp:gaussdb://host* [:*port*] / [*database_name*] [?*connection_option*]：通过TCP/IP连接。
- SQL string constant：包含上述三种形式之一的值。

说明

其他connection_target参数介绍请参见[ecpg兼容](#)。

- **connection_name**

用于该连接的一个可选标识符，可以在其他命令中引用它。可以是一个SQL标识符或者一个宿主变量。

- **connection_user**

用于数据库连接的用户名/密码。

以下方式选择一种使用：

- user_name/password*
- user_name* SQLIDENTIFIED BY *password*
- user_name* USING *password*

此参数也能指定用户名和密码。

用户名和密码可以是SQL标识符、字符串常量或者宿主变量。

说明

上述参数中斜体部分为变量，请根据实际情况进行修改。

示例

指定连接参数变体的示例：

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main USER connectuser USING "connectdb";
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO tcp:gaussdb://localhost/connectdb USER connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:gaussdb://localhost:$PORT/connectdb USER connectuser SQLIDENTIFIED BY
connectpw;
EXEC SQL CONNECT TO unix:gaussdb://localhost/connectdb USER connectuser SQLIDENTIFIED BY
"connectpw";
EXEC SQL CONNECT TO unix:gaussdb://localhost/connectdb USER connectuser USING "connectpw";
```

连接语法使用示例：

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    /* 宿主变量定义，定义连接串所需的database、password等字段，实际值应从环境变量或配置文件读取，环
    境变量需用户自己按需配置；非环境变量情况下可直接赋值字符串。*/
    EXEC SQL BEGIN DECLARE SECTION;
    char* db = (char*)malloc((sizeof(char) * 200));
    char* pw = (char*)malloc((sizeof(char) * 200));
    EXEC SQL END DECLARE SECTION;
    char* db_env = getenv("EXAMPLE_DATABASENAME_ENV");
    char* pw_env = getenv("EXAMPLE_PASSWD_ENV");
    strcpy(db, db_env);
    strcpy(pw, pw_env);

    /* 连接语句涉及数据库、用户、密码。需提前创建好并有相关操作权限。*/

    /* 连接方式：EXEC SQL CONNECT TO [ database_name ][ @host ][ :port ] [ USER connection_user ] */
    /* case1: 使用默认的本地连接方式，连接数据库为gaussdb库。*/
    EXEC SQL CONNECT TO gaussdb;

    /* case2: 使用默认的本地连接方式，连接数据库为gaussdb库，连接别名为conn1。*/
    EXEC SQL CONNECT TO gaussdb AS conn1;

    /* case3: 使用ip+port方式（localhost数据库监听的本地地址，$PORT为数据库监听端口），连接数据库为
    connectdb库，指定数据库别名，指定用户密码。$PORT换成真实的端口号*/
    EXEC SQL CONNECT TO connectdb@localhost:$PORT AS conn2 USER connectuser USING :pw;

    /* case4: 使用ip+port方式（127.0.0.1数据库监听的本地地址，$PORT为数据库监听端口），连接数据库为
    connectdb库，指定数据库别名，指定用户密码。*/
    EXEC SQL CONNECT TO connectdb@127.0.0.1:$PORT AS conn3 USER connectuser SQLIDENTIFIED BY :pw;

    /* case5: 关闭数据库连接 */
    EXEC SQL DISCONNECT gaussdb;
    EXEC SQL DISCONNECT conn1;
    EXEC SQL DISCONNECT conn2;
    EXEC SQL DISCONNECT conn3;
```

```
/* 连接方式: EXEC SQL CONNECT TO <tcp|unix>:gaussdb://host [ :port ]/[ database_name ][ ?
connection_option ] */
/* case1: 通过宿主变量pw、db方式, 替换url变量。*/
strcpy(db, "tcp:gaussdb://localhost/connectdb");
EXEC SQL CONNECT TO :db USER connectuser USING :pw;

/* case2: 其中127.0.0.1为数据库侦听ip,connectdb为数据库database。*/
EXEC SQL CONNECT TO tcp:gaussdb://127.0.0.1/connectdb AS conn4 USER connectuser USING :pw;

/* case3:其中127.0.0.1为数据库侦听ip,connectdb为数据库database,connect_timeout=14为连接串配置参数。*/
EXEC SQL CONNECT TO tcp:gaussdb://localhost/connectdb?connect_timeout=14 AS conn5 USER
connectuser SQLIDENTIFIED BY :pw;

/* case4: 关闭所有连接 */
EXEC SQL CLOSE ALL;

/* 连接数据库, 并执行业务 */
EXEC SQL CONNECT TO tcp:gaussdb://127.0.0.1/connectdb AS conn4 USER connectuser USING :pw;
EXEC SQL SET AUTOCOMMIT = ON;
EXEC SQL CREATE TABLE t1(a INT);
EXEC SQL INSERT INTO t1 VALUES(1), (2);
EXEC SQL SELECT a FROM t1 WHERE a > 1;
EXEC SQL DROP TABLE t1;
EXEC SQL DISCONNECT CURRENT;
free(pw);
free(db);

return 0;
}
```

使用宿主变量指定连接参数的示例:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    /* 此处dbname、user、pwd应从环境变量或配置文件读取, 环境变量需用户自己按需配置; 非环境变量情况
    下可直接赋值字符串 */
    char *dbname = getenv("EXAMPLE_DBNAME_ENV"); /* 数据库名 */
    char *user = getenv("EXAMPLE_USERNAME_ENV"); /* 连接用户名 */
    char *pwd = getenv("EXAMPLE_PASSWD_ENV"); /* 密码 */
    char *connection = "tcp:gaussdb://localhost:$PORT/testdb"; /* 连接字符串 $PORT换成真实的端口号 */
    char ver[256]; /* 存储版本字符串的缓冲区 */
    EXEC SQL END DECLARE SECTION;

    /* 开启调试日志 */
    ECPGdebug(1, stderr);

    /* 连接到数据库 */
    EXEC SQL CONNECT TO :dbname;
    /* 设置搜索路径 */
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);
    EXEC SQL COMMIT;
    /* 查询数据库版本 */
    EXEC SQL SELECT version() INTO :ver;
    /* 断开连接 */
    EXEC SQL DISCONNECT;

    /* 打印数据库版本 */
    printf("version: %s\n", ver);

    /* 使用连接字符串连接到数据库 */
    EXEC SQL CONNECT TO :connection USER :user USING :pwd;
    /* 设置搜索路径 */
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);
    EXEC SQL COMMIT;
    /* 查询数据库版本 */
}
```



```
EXEC SQL SELECT version() INTO :ver;  
/* 断开连接 */  
EXEC SQL DISCONNECT;  
  
/* 打印数据库版本 */  
printf("version: %s\n", ver);  
  
return 0;  
}
```

运行结果如下（根据本地环境结果会不一致，此处运行结果仅供参考）：

```
version: gaussdb (GaussDB Kernel 506.0.0 build 7cc7ccbb) compiled at 2025-02-24 09:17:26 commit 10340  
last mr 22441 debug  
version: gaussdb (GaussDB Kernel 506.0.0 build 7cc7ccbb) compiled at 2025-02-24 09:17:26 commit 10340  
last mr 22441 debug
```

相关链接

[DISCONNECT](#)，[SET CONNECTION](#)

5.8.3.7.3 DEALLOCATE DESCRIPTOR

功能描述

释放SQL描述符区域。

语法格式

```
DEALLOCATE DESCRIPTOR name
```

参数说明

name

SQL描述符名称。大小写敏感，是一个SQL标识符或一个宿主变量。

示例

```
DEALLOCATE DESCRIPTOR mydesc;
```

相关链接

[ALLOCATE DESCRIPTOR](#)，[GET DESCRIPTOR](#)，[SET DESCRIPTOR](#)

5.8.3.7.4 DECLARE

功能描述

声明一个游标用于迭代预备语句的结果集。该命令与内核SQL命令的DECLARE在语义上略有不同，后者执行查询并准备结果集以便检索，而嵌入式SQL命令只是将一个名称声明为“循环变量”并用于在查询的结果集上迭代，实际在使用OPEN命令打开游标时执行。

语法格式

```
DECLARE cursor_name [ BINARY ] [ NO SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR  
prepared_name  
DECLARE cursor_name [ BINARY ] [ NO SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

参数说明

- **cursor_name**
游标名称，大小写敏感。可以是一个SQL标识符或者一个宿主变量。
- **prepared_name**
预备查询的名称。可以是一个SQL标识符或者一个宿主变量。
- **query**
提供游标要返回的行的SELECT命令。

说明

游标选项的含义请参见《参考》中“SQL参考 > SQL语法 > D > DECLARE”章节。

示例

声明用于查询的游标示例：

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;  
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

声明用于预备语句的游标示例：

```
EXEC SQL PREPARE stmt1 AS SELECT version();  
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

相关链接

[OPEN](#)

5.8.3.7.5 DESCRIBE

功能描述

检索预备语句中包含的结果列的元数据信息。

语法格式

```
DESCRIBE [ OUTPUT ] prepared_name USING SQL DESCRIPTOR descriptor_name  
DESCRIBE [ OUTPUT ] prepared_name INTO SQL DESCRIPTOR descriptor_name  
DESCRIBE [ OUTPUT ] prepared_name INTO sqllda_name
```

参数说明

- **prepared_name**
预备语句名称。可以是一个SQL标识符或者宿主变量。
- **descriptor_name**
描述符名称。大小写敏感，可以是一个SQL标识符或者一个宿主变量。
- **sqllda_name**
SQLDA变量名称，详细使用请参见[SQLDA](#)。

示例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;  
EXEC SQL PREPARE stmt1 FROM :sql_stmt;  
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

```
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;  
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

相关链接

[ALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#)

5.8.3.7.6 DISCONNECT

功能描述

关闭一个或所有的数据库连接。

语法格式

```
DISCONNECT connection_name  
DISCONNECT [ CURRENT ]  
DISCONNECT DEFAULT  
DISCONNECT ALL
```

参数说明

- **connection_name**
由CONNECT命令建立的数据库连接名称。
- **CURRENT**
关闭当前连接，它可以是最近打开的连接或者是由SET CONNECTION命令设置的连接。如果没有参数被传入给DISCONNECT命令，当前链接则会作为默认值被关闭。
- **DEFAULT**
关闭默认连接。
- **ALL**
关闭所有打开的连接。

示例

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main(void)  
{  
    /* 需要提前创建testdb库 */  
    EXEC SQL CONNECT TO testdb AS DEFAULT;  
    EXEC SQL CONNECT TO testdb AS con1;  
    EXEC SQL CONNECT TO testdb AS con2;  
    EXEC SQL CONNECT TO testdb AS con3;  
    EXEC SQL DISCONNECT CURRENT; /* 关闭 con3 */  
    EXEC SQL DISCONNECT DEFAULT; /* 关闭 DEFAULT */  
    EXEC SQL DISCONNECT ALL; /* 关闭 con2 以及 con1 */  
    return 0;  
}
```

相关链接

[CONNECT](#), [SET CONNECTION](#)

5.8.3.7.7 EXECUTE IMMEDIATE

功能描述

预备并且执行动态指定的SQL语句，不检索结果行。

语法格式

```
EXECUTE IMMEDIATE string
```

参数说明

string

包含要被执行的SQL语句的C字符串或者宿主变量。

示例

使用EXECUTE IMMEDIATE和名为command的宿主变量执行INSERT语句：

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: "r1"', 1, 'f');  
EXEC SQL EXECUTE IMMEDIATE :command;
```

5.8.3.7.8 GET DESCRIPTOR

功能描述

检索查询结果集的信息，并且将它存储到宿主变量中。在使用该命令将信息传递给宿主语言变量之前通常使用FETCH或者SELECT填充标识符区域。该命令有两种形式：

- 检索描述符的头部项，适用于全面查看结果集。
- 列号作为附加参数，检索特定列的信息。

语法格式

```
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item [, ... ]  
GET DESCRIPTOR descriptor_name:cvariable = descriptor_header_item [, ... ]
```

参数说明

- **descriptor_name**
描述符名称。
- **column_number**
关于被检索的列数信息。计数从1开始。
- **descriptor_item**
标记识别检索列的信息项。
- **cvariable**
宿主变量将接收从描述符区域检索的数据。
- **descriptor_header_item**
标识要检索哪一个头部信息项。当前仅支持用于得到结果集中列数的COUNT。

示例

检索结果集中的列数：

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

在第一列中检索数据长度：

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

检索作为字符串第二列的数据主体：

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

执行SELECT current_database();并且显示列数、列数据长度和列数据的完整过程示例：

前提条件：需要提前创建testdb数据库。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    int d_count = 0;
    char d_data[1024] = {0};
    int d_returned_octet_length = 0;
    EXEC SQL END DECLARE SECTION;

    /* 需要提前创建testdb */
    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);
    EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* 描述、打开一个游标，并且分配一个描述符给该游标 */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* 得到全部列的数量 */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count = %d\n", d_count);

    /* 得到一个返回列的长度 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* 将返回的列取出成一个字符串 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data = %s\n", d_data);

    /* 关闭 */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

运行结果如下：

```
d_count          = 1
d_returned_octet_length = 6
d_data           = testdb
```

相关链接

[ALLOCATE DESCRIPTOR](#)，[DEALLOCATE DESCRIPTOR](#)，[SET DESCRIPTOR](#)

5.8.3.7.9 OPEN

功能描述

打开一个游标，并将实际值选择性地绑定到游标声明中的占位符。该游标必须事先使用DECLARE命令声明过。执行OPEN命令会在服务器上执行查询。

语法格式

```
OPEN cursor_name  
OPEN cursor_name USING value [, ... ]  
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

参数说明

- **cursor_name**
被打开的游标的名称。可以是一个SQL标识符或者一个宿主变量。
- **value**
被绑定到游标中一个占位符的值。可以是一个SQL常量、一个宿主变量或者一个带有指示符的宿主变量。
- **descriptor_name**
要绑定到游标中占位符的值的描述符名称。可以是一个SQL标识符或者一个宿主变量。

示例

```
EXEC SQL OPEN a;  
EXEC SQL OPEN d USING 1, 'test';  
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;  
EXEC SQL OPEN :curname1;
```

相关链接

[DECLARE](#)

5.8.3.7.10 PREPARE

功能描述

准备用于执行的语句。

语法格式

```
PREPARE name FROM string
```

参数说明

- **name**
预备查询标识符。
- **string**
动态构造的SQL语句，可以是C字符串或宿主变量。

示例

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";
EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;
EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

说明

ecpg提供的PREPARE语句不直接等同于内核提供的PREPARE语法，举例如下：

GaussDB语法：

```
PREPARE name [ ( data_type [, ...] ) ] AS statement;
```

嵌入式SQL语句：

```
EXEC SQL PREPARE I(int, int) AS INSERT INTO T VALUES ( $1, $2 );
EXEC SQL EXECUTE I(1, 2);
```

执行上述语句出现报错时，提示too few arguments on。

利用动态SQL语法规则解决上述问题：

```
EXEC SQL PREPARE I AS INSERT INTO T VALUES ( $1, $2 );
EXEC SQL EXECUTE I using 1, 2;
```

5.8.3.7.11 SET AUTOCOMMIT

功能描述

设置当前数据库会话的自动提交行为。默认情况下，嵌入式SQL程序不自动提交，因此需要显式地发出COMMIT。该命令可以把会话改成自动提交模式，使每一个单独的语句都会被隐式提交。

语法格式

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

5.8.3.7.12 SET CONNECTION

功能描述

设置一个数据库连接。

语法格式

```
SET CONNECTION { TO | = } connection_name
```

参数说明

connection_name

由CONNECT命令建立的数据库连接名称。

示例

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

相关链接

[CONNECT](#)，[DISCONNECT](#)

5.8.3.7.13 SET DESCRIPTOR

功能描述

在SQL描述符区域中设置信息，描述符区域通常用于绑定预备查询执行中的参数。该命令有两种功能：

- 用于描述符头部，它独立于特定的数据。
- 为由数字标识的特定数据赋值。

语法格式

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]  
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

参数说明

- **descriptor_name**
描述符名称。
- **descriptor_header_item**
标记识别设置的头部信息项。目前仅支持COUNT设置描述符项数。
- **number**
设置的描述符项数。计数从1开始。
- **descriptor_item**
标记识别在描述符中的项的信息。当前仅支持DATA、TYPE和LENGTH。
- **value**
存储到描述符项中的值。可以是SQL常量或者宿主变量。

示例

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;  
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;  
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;  
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';  
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

相关链接

[ALLOCATE DESCRIPTOR](#), [DEALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#)

5.8.3.7.14 TYPE

功能描述

定义一个新的数据类型。当运行带有-c选项的ecpg时，仅标识该命令。

语法格式

```
TYPE type_name IS ctype
```

参数说明

- **type_name**

数据类型名称。

- **ctype**
C语言数据类型。

示例

```
EXEC SQL TYPE customer IS
struct
{
    varchar name[50];
    int    phone;
};

EXEC SQL TYPE cust_ind IS
struct ind
{
    short  name_ind;
    short  phone_ind;
};

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZE];
EXEC SQL TYPE string IS char[11];
```

使用EXEC SQL TYPE的示例

前提条件：示例在ecpg预处理阶段需要添加-c参数，并且需要提前创建testdb数据库。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

EXEC SQL WHENEVER SQLERROR SQLPRINT;
EXEC SQL TYPE tt IS
struct
{
    varchar v[256];
    int    i;
};
EXEC SQL TYPE tt_ind IS
struct ind {
    short  v_ind;
    short  i_ind;
};

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
    EXEC SQL END DECLARE SECTION;

    /* 连接到数据库 */
    EXEC SQL CONNECT TO testdb AS con1;

    /* 查询当前数据库名称和一个固定值（256），并将结果存入结构体和指示符变量 */
    EXEC SQL SELECT current_database(), 256 INTO :t_ind LIMIT 1;

    /* 打印结果 */
    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);
}
```

```
/* 断开数据库连接 */  
EXEC SQL DISCONNECT con1;  
  
return 0;  
}
```

运行结果如下：

```
t.v = testdb  
t.i = 256  
t_ind.v_ind = 0  
t_ind.i_ind = 0
```

5.8.3.7.15 VAR

功能描述

将新的C语言数据类型分配给宿主变量。宿主变量必须预先在声明段声明。

说明

- 对于VAR的用法需要谨慎。使用VAR语句后数据类型的变化可能会导致内存地址无效，从而导致数据变量无效，出现无法成功赋值的场景。
- 若在宿主变量声明段中确定好数据类型，则无须使用VAR语句。

语法格式

```
VAR varname IS ctype
```

参数说明

- **varname**
C语言变量名称。
- **ctype**
C语言类型说明。

示例

```
EXEC SQL BEGIN DECLARE SECTION;  
short a;  
EXEC SQL END DECLARE SECTION;  
EXEC SQL VAR a IS int;
```

5.8.3.7.16 WHENEVER

功能描述

定义一个行为，它会在SQL执行异常时（行未找到、SQL告警或错误）被调用。

语法格式

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

参数说明

- **SQLERROR**
当SQL语句执行期间发生错误时，调用指定操作。

- **SQLWARNING**

当SQL语句执行期间发生告警时，调用指定操作。

- **NOT FOUND**

当SQL语句检索或者影响为零行时，调用指定操作。

说明

action表示需要执行的操作，详情请参考[设置回调](#)。

示例

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR SQLCALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

使用WHENEVER NOT FOUND BREAK来处理结果集的循环，参考如下完整示例：

前提条件：需要提前创建testdb数据库。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;
    EXEC SQL BEGIN DECLARE SECTION;
    char* d1;
    char* d2;
    EXEC SQL END DECLARE SECTION;

    /* 当到达结果集末尾时，跳出循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        exec sql get descriptor d value 1 :d1=DATA;
        exec sql get descriptor d value 2 :d2=DATA;
        printf("d1 is %s,%s\n", d1, d2) ;
    }
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;
    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

运行结果如下：

```
d1 is testdb,hoge
```

5.8.4 ecpg 预处理、编译、执行

前提条件

已准备嵌入式SQL-C源程序，以.pgc为后缀名，ecpg负责将其转换成可被编译器编译的C语言程序。

生成的C语言程序使用gcc命令编译为可执行文件，运行该可执行文件实现客户端程序访问数据库。

文件包含

嵌入式SQL-C程序开发中，一般需要包含以下几个头文件：

```
#include <stdio.h>
#include <stdlib.h>
EXEC SQL INCLUDE sqlca;
```

其中，sqlca是SQL通信区（SQL Communication Area），用于存储SQL语句执行后的状态信息（如成功、错误码）。关于sqlca详情请参考[sqlca](#)。

如需将外部文件包含到嵌入SQL程序中，可使用如下语句：

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

说明

- ecpg预处理器按照如下顺序搜索文件：
 - 当前目录
 - /usr/local/include
 - GaussDB的目录，在编译时定义
 - /usr/include
- EXEC SQL INCLUDE "filename"语句仅搜索当前目录。
- 在每一个目录中，预处理器首先按照给定的文件名搜索，若没找到则会追加.h到文件名后重试（除非指定的文件名已有该后缀）。
- 文件名是大小写敏感的。

如果使用C语法引入自定义头文件，且头文件中包含ecpg的语法时，可以按照以下示例方式进行预处理：

- a.pgc文件：

```
#include "a.h"
```
- a.h文件：

```
#ifndef _A_H
#define _A_H
EXEC SQL BEGIN DECLARE SECTION;
char sDbName[50];
char sUser[50];
char sPwd[50];
EXEC SQL END DECLARE SECTION;
#endif
```
- 编译方式：
 - 使用cpp预处理，进行头文件展开，生成新的pgc文件：

```
cpp -I $GAUSSHOME/include/ecpg -I $GAUSSHOME/include -I $GAUSSHOME/include/gaussdb/
server/ -E -C a.pgc a.tmp.d
```

```
cpp -I $GAUSSHOMe/include/ecpg -I $GAUSSHOMe/include -I $GAUSSHOMe/include/gaussdb/  
server/ -P a.tmp.d a.tmp.pgc
```

b. **ecpg预处理:**

```
ecpg -I $GAUSSHOMe/include/ecpg -o a.tmp.c a.tmp.pgc
```

c. **gcc编译:**

```
gcc -g -I $GAUSSHOMe/include/ecpg -I $GAUSSHOMe/include -I $GAUSSHOMe/include/  
gaussdb/server/ -L $GAUSSHOMe/lib a.tmp.c -o a -lecpg -lrt -lpq -lpqtypes -lpthread
```

说明

- ecpg引入自定义头文件时，不要同时使用c语言的include方式引入和ecpg语法引入，避免出现可能重复展开头文件的问题。
- 使用cpp命令时，连接串中不能直接使用tcp和unix协议；如果需要此两种形式的连接串，使用宿主变量（宿主变量详情请参考[数据类型和宿主变量](#)）定义。

条件编译

ecpg提供了ifdef、ifndef、else、elif和endif条件编译指令。在预处理时，按照不同的条件去编译程序的不同部分，使用时，需要添加EXEC SQL前缀关键字。

示例如下：

```
EXEC SQL ifndef TZVAR;  
EXEC SQL SET TIMEZONE TO 'GMT';  
EXEC SQL elif TZNAME;  
EXEC SQL SET TIMEZONE TO TZNAME;  
EXEC SQL else;  
EXEC SQL SET TIMEZONE TO TZVAR;  
EXEC SQL endif;
```

define 和 undef 指令

嵌入式SQL具有类似于C语言中#define的指令：

```
EXEC SQL DEFINE name;  
EXEC SQL DEFINE name value;  
EXEC SQL UNDEF name;
```

示例如下：

```
/* 定义名称 */  
EXEC SQL DEFINE HAVE_FEATURE;  
  
/* 定义常量 */  
EXEC SQL DEFINE MYNUMBER 12;  
EXEC SQL DEFINE MYSTRING 'abc';  
  
/* 使用 UNDEF 移除定义 */  
EXEC SQL UNDEF MYNUMBER;
```

在嵌入式SQL程序中，既可以使用C语言标准的#define和#undef，也可以使用专为SQL设计的EXEC SQL DEFINE。两者的主要区别在于定义值的计算时机：C语言的#define在编译阶段由C预处理器处理并进行文本替换；而EXEC SQL DEFINE则在ecpg预处理阶段被计算和替换。这意味着，EXEC SQL DEFINE的定义会在SQL代码被处理之前生效，从而影响后续的SQL语句生成。如下示例，ecpg对其进行替换并且编译器不会解析名为MYNUMBER的任何名称或标识符：

```
EXEC SQL DEFINE MYNUMBER 12;  
...  
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

须知

不能把#define用于一个将要在嵌入式SQL查询中使用的变量，因为在这种情况下嵌入式SQL预编译器无法识别该声明。

预处理

ecpg预处理命令如下（以test.pgc文件为例）：

```
ecpg -I $GAUSSHOMe/include/ecpg -o test.c test.pgc
```

ecpg预处理的参数选项如下：

```
ecpg [OPTION]...
```

其中OPTION参数选项如下：

- -o OUTFILE：预处理嵌入式SQL-C程序将结果写入OUTFILE文件，OUTFILE文件为C语言文件（test.c）。
- -I DIRECTORY：头文件的搜索路径。
- -c：预处理嵌入式SQL-C程序自动生成C语言文件。
- --version：查看ecpg当前版本。
- -C MODE：指定预处理兼容模式，例如传入“ORA”可开启ORACLE兼容模式。
- -r OPTION：指定运行时的行为。OPTION支持以下值：
 - no_indicator：允许使用特殊值来表示空值。
 - prepare：强制在执行所有语句之前先进行PREPARE操作。
 - questionmarks：允许使用问号“？”作为占位符。
 - with_hold：对于未指定HOLD关键字创建的游标，当指定该选项后，游标行为默认为WITH HOLD（该功能需要ecpg和内核版本一致）。

编译

编译经过预处理的嵌入式SQL-C程序命令如下：

```
gcc -I $GAUSSHOMe/include/ecpg -I $GAUSSHOMe/include -I $GAUSSHOMe/include/gaussdb/server/ -L $GAUSSHOMe/lib test.c -o test -lecpg -lrt -lpq -lpqtypes -lpthread
```

执行

执行命令如下：

```
./test
```

须知

- ecpg作为编译预处理工具，若在预处理或编译过程中出现找不到头文件或者函数实现的报错信息，可以根据需要指定头文件，或者连接动态库。
- 使用ecpg开发应用程序时，所依赖的其他动态库和头文件通常位于\$GAUSSHOMe/include/libpq、\$GAUSSHOMe/include目录下。
- 编译过程中常见的动态库依赖：-lpq、-lpq_ce、-lpthread。若开发过程中需要使用libpq通信库，则需要连接-lpq和-lpq_ce。若开发过程中需要使用多线程连接，则需要连接-lpthread。

5.8.5 典型应用开发示例

5.8.5.1 创建表、插入数据并查询

前提条件：需要提前创建testdb数据库。

须知

在数据库连接过程中，若连接语句中指定了ip-port，则必须指定username/password，该规则由GaussDB内核通信认证所决定。若不指定ip-port，则通过本地\$PGPORT（UDS协议）进行通信。

以下介绍几种常用连接数据库的方式（详情请参考[管理数据库连接](#)，[CONNECT](#)）：

```
/* 连接语句涉及数据库、用户、密码。需提前创建好并有相关操作权限。*/
```

```
/* 连接方式：EXEC SQL CONNECT TO [ database_name ] [ @host ] [ :port ] [ USER connection_user ] */  
/* case1: 使用默认的本地连接方式，连接数据库为gaussdb库。*/  
EXEC SQL CONNECT TO gaussdb;
```

```
/* case2: 使用默认的本地连接方式，连接数据库为gaussdb库，连接别名为conn1。*/  
EXEC SQL CONNECT TO gaussdb AS conn1;
```

```
/* case3: 使用ip+port方式（localhost数据库监听的本地地址，$PORT为数据库监听端口），连接数据库为connectdb库，指定数据库别名，指定用户密码。$PORT换成真实的端口号。*/  
EXEC SQL CONNECT TO connectdb@localhost:$PORT AS conn2 USER connectuser USING :pw;
```

```
/* case4: 使用ip+port方式（127.0.0.1数据库监听的本地地址，$PORT为数据库监听端口），连接数据库为connectdb库，指定数据库别名，指定用户密码。*/  
EXEC SQL CONNECT TO connectdb@127.0.0.1:$PORT AS conn3 USER connectuser SQLIDENTIFIED BY :pw;
```

此示例演示如何使用ecpg创建表、插入数据并查询数据。

```
#include <locale.h>  
#include <string.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
EXEC SQL WHENEVER SQLERROR SQLPRINT;  
EXEC SQL INCLUDE sqlca;  
  
int main(void)  
{  
    EXEC SQL BEGIN DECLARE SECTION;  
    char *temp_str = (char *)malloc(11);  
    EXEC SQL END DECLARE SECTION;  
  
    /* 提前创建testdb库 */  
    EXEC SQL CONNECT TO testdb;  
  
    /* 打开自动提交，以下执行EXEC SQL时不用手动COMMIT */  
    EXEC SQL SET AUTOCOMMIT = ON;  
    EXEC SQL DROP TABLE IF EXISTS test_t;  
  
    /* 建表，插入数据 */  
    EXEC SQL CREATE TABLE test_t(f FLOAT, i INT, a INT[10], mstr CHAR(10));  
    EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(1.01, 1, '{0,1,2,3,4,5,6,7,8,9}', 'China');  
  
    /* 关闭自动提交，以下插入数据的SQL语句需要手动COMMIT才能提交 */  
    EXEC SQL SET AUTOCOMMIT = OFF;  
    EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(2.01, 2, '{0,1,2,3,4,5,6,7,8,9}', 'USA');  
    EXEC SQL COMMIT;  
  
    EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(3.01, 3, '{0,1,2,3,4,5,6,7,8,9}', 'AUS');
```

```
EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(4.01, 4, '{0,1,2,3,4,5,6,7,8,9}', 'JAP');
EXEC SQL COMMIT;

EXEC SQL BEGIN DECLARE SECTION;
int a[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int id = 6;
EXEC SQL END DECLARE SECTION;

/* 从宿主变量取数据插入到表中，宿主变量的类型与表定义的类型一致 */
strcpy(temp_str, "RUS");
EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(5.01, 5, :a, :temp_str);
EXEC SQL COMMIT;

EXEC SQL SET AUTOCOMMIT = ON;
EXEC SQL BEGIN;
EXEC SQL INSERT INTO test_t(f, i, a, mstr) VALUES(6.01, :id, :a, 'SIG');
EXEC SQL COMMIT;
EXEC SQL SET AUTOCOMMIT = OFF;

EXEC SQL BEGIN DECLARE SECTION;
float ff = 0.0f;
char tmp_text[25] = "klmnopqrst";
EXEC SQL END DECLARE SECTION;

EXEC SQL SET AUTOCOMMIT = ON;
EXEC SQL BEGIN WORK;

printf("Found ff=%f tmp_text=%10.10s\n", ff, tmp_text);

/* 条件查询语句示例 */
EXEC SQL SELECT f, mstr INTO :ff, :tmp_text FROM test_t WHERE f > (SELECT f FROM test_t WHERE i = 4
OR i < 0) ORDER BY a LIMIT 1;
printf("Found ff=%f tmp_text=%10.10s\n", ff, tmp_text);

EXEC SQL SELECT f, mstr INTO :ff, :tmp_text FROM test_t WHERE mstr = 'JAP' ORDER BY i;
printf("Found ff=%f tmp_text=%10.10s\n", ff, tmp_text);

EXEC SQL SELECT f, mstr INTO :ff, :tmp_text FROM test_t ORDER BY i DESC LIMIT 1;
printf("Found ff=%f tmp_text=%10.10s\n", ff, tmp_text);

EXEC SQL SELECT f, mstr INTO :ff, :tmp_text FROM test_t ORDER BY mstr LIMIT 1;
printf("Found ff=%f tmp_text=%10.10s\n", ff, tmp_text);

EXEC SQL SELECT COUNT(f), a INTO :ff, :tmp_text FROM test_t WHERE i > 2 GROUP BY a LIMIT 1;
printf("Found ff=%f tmp_text=%20.30s\n", ff, tmp_text);

EXEC SQL SELECT COUNT(f), a INTO :ff, :tmp_text FROM test_t WHERE i > 3 GROUP BY a ORDER BY a
LIMIT 1;
printf("Found ff=%f tmp_text=%20.30s\n", ff, tmp_text);

EXEC SQL SELECT SUM(f), a INTO :ff, :tmp_text FROM test_t WHERE i > 2 GROUP BY a ORDER BY a
LIMIT 1;
printf("Found ff=%f tmp_text=%20.30s\n", ff, tmp_text);

EXEC SQL SELECT DISTINCT a INTO :tmp_text FROM test_t ORDER BY a LIMIT 1;

EXEC SQL DROP TABLE test_t;

EXEC SQL COMMIT;
/* 释放连接，释放为宿主变量分配的内存 */
EXEC SQL DISCONNECT;
free(temp_str);

return 0;
}
```

运行结果如下：

```
Found ff=0.000000 tmp_text=klmnopqrst
Found ff=5.010000 tmp_text=RUS
```



```
Found ff=4.010000 tmp_text=JAP
Found ff=6.010000 tmp_text=SIG
Found ff=3.010000 tmp_text=AUS
Found ff=2.000000 tmp_text={9,8,7,6,5,4,3,2,1,0}
Found ff=1.000000 tmp_text={0,1,2,3,4,5,6,7,8,9}
Found ff=7.020000 tmp_text={0,1,2,3,4,5,6,7,8,9}
```

5.8.5.2 日期、时间戳与区间解析

此示例使用库函数对时间和日期类型进行不同操作。具体使用方式请参见[使用库函数](#)章节。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <pgtypes_date.h>
#include <pgtypes_timestamp.h>

char *dates[] = { "19990108foobar",
                  "19990108 foobar",
                  "1999-01-08 foobar",
                  "January 8, 1999",
                  "1999-01-08",
                  "1/8/1999",
                  "1/18/1999",
                  "01/02/03",
                  "1999-Jan-08",
                  "Jan-08-1999",
                  "08-Jan-1999",
                  "99-Jan-08",
                  "08-Jan-99",
                  "08-Jan-06",
                  "Jan-08-99",
                  "19990108",
                  "990108",
                  "1999.008",
                  "J2451187",
                  "January 8, 99 BC",
                  NULL
                };

/* 不可与libc的“times”冲突 */
static char *times[] = { "0:04",
                         "1:59 PDT",
                         "13:24:40 -8:00",
                         "13:24:40.495+3",
                         NULL
                       };

char *intervals[] = { "1 minute",
                      "1 12:59:10",
                      "2 day 12 hour 59 minute 10 second",
                      "1 days 12 hrs 59 mins 10 secs",
                      "1 days 1 hours 1 minutes 1 seconds",
                      "1 year 59 mins",
                      "1 year 59 mins foobar",
                      NULL
                    };

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    date date1;
    timestamp ts1, ts2;
    char *text;
    interval *i1;
    date *dc;
    EXEC SQL END DECLARE SECTION;
```

```
int i, j;
char *endptr;

/* 从文本中解析一个时间戳并将日期转换成字符串 */
ts1 = PGTYPEStimestamp_from_asc("2003-12-04 17:34:29", NULL);
text = PGTYPEStimestamp_to_asc(ts1);

printf("timestamp: %s\n", text);
free(text);

/* 从时间戳中抽取日期部分 */
date1 = PGTYPEStimestamp_from_timestamp(ts1);
dc = PGTYPEStimestamp_new();
*dc = date1;
/* 返回一个日期变量的文本表达 */
text = PGTYPEStimestamp_to_asc(*dc);
printf("Date of timestamp: %s\n", text);
free(text);
PGTYPEStimestamp_free(dc);

for (i = 0; dates[i]; i++)
{
    bool err = false;
    /* 从日期的文本表达解析一个日期 */
    date1 = PGTYPEStimestamp_from_asc(dates[i], &endptr);
    if (date1 == INT_MIN) {
        err = true;
    }
    /* 返回一个日期变量的文本表达 */
    text = PGTYPEStimestamp_to_asc(date1);
    printf("Date[%d]: %s (%c - %c)\n",
        i, err ? "-" : text,
        endptr ? 'N' : 'Y',
        err ? 'T' : 'F');
    free(text);
    if (!err)
    {
        for (j = 0; times[j]; j++)
        {
            int length = strlen(dates[i]) + 1 + strlen(times[j]) + 1;
            char* t = (char *)malloc(length);
            sprintf(t, "%s %s", dates[i], times[j]);
            /* 从文本中解析一个时间戳并将日期转换成字符串 */
            ts1 = PGTYPEStimestamp_from_asc(t, NULL);
            text = PGTYPEStimestamp_to_asc(ts1);
            if (i != 19 || j != 3)
                printf("TS[%d,%d]: %s\n", i, j, errno ? "-" : text);
            free(text);
            free(t);
        }
    }
}

/* 从文本中解析一个时间戳 */
ts1 = PGTYPEStimestamp_from_asc("2004-04-04 23:23:23", NULL);

for (i = 0; intervals[i]; i++)
{
    interval *ic;
    /* 从文本中解析一个区间 */
    i1 = PGTYPEStimestamp_from_asc(intervals[i], &endptr);
    if (*endptr)
        printf("endptr set to %s\n", endptr);
    if (!i1)
    {
        printf("Error parsing interval %d\n", i);
        continue;
    }
}
```

```
    }  
    /* 把一个区间变量加到时间戳变量上 */  
    j = PGTYPEStimestamp_add_interval(&ts1, i1, &ts2);  
    if (j < 0)  
        continue;  
    /* 将一个区间类型变量转换成文本格式 */  
    text = PGTYPEStimestamp_to_asc(i1);  
    printf("interval[%d]: %s\n", i, text ? text : "-");  
    free(text);  
  
    /* 返回一个已分配区间变量的指针 */  
    ic = PGTYPEStimestamp_new();  
    /* 复制一个区间类型的变量 */  
    PGTYPEStimestamp_copy(i1, ic);  
    /* 将一个区间类型变量转换成文本格式 */  
    text = PGTYPEStimestamp_to_asc(i1);  
    printf("interval_copy[%d]: %s\n", i, text ? text : "-");  
    free(text);  
    /* 释放已经分配区间变量的内存 */  
    PGTYPEStimestamp_free(ic);  
    PGTYPEStimestamp_free(i1);  
}  
  
return (0);  
}
```

运行结果如下：

```
Date of timestamp: 2003-12-04  
Date[0]: - (N - T)  
Date[1]: - (N - T)  
Date[2]: - (N - T)  
Date[3]: 1999-01-08 (N - F)  
TS[3,0]: 1999-01-08 00:04:00  
TS[3,1]: 1999-01-08 01:59:00  
TS[3,2]: 1999-01-08 13:24:40  
TS[3,3]: 1999-01-08 13:24:40.495  
Date[4]: 1999-01-08 (N - F)  
TS[4,0]: 1999-01-08 00:04:00  
TS[4,1]: 1999-01-08 01:59:00  
TS[4,2]: 1999-01-08 13:24:40  
TS[4,3]: 1999-01-08 13:24:40.495  
Date[5]: 1999-01-08 (N - F)  
TS[5,0]: 1999-01-08 00:04:00  
TS[5,1]: 1999-01-08 01:59:00  
TS[5,2]: 1999-01-08 13:24:40  
TS[5,3]: 1999-01-08 13:24:40.495  
Date[6]: 1999-01-18 (N - F)  
TS[6,0]: 1999-01-18 00:04:00  
TS[6,1]: 1999-01-18 01:59:00  
TS[6,2]: 1999-01-18 13:24:40  
TS[6,3]: 1999-01-18 13:24:40.495  
Date[7]: 2003-01-02 (N - F)  
TS[7,0]: 2003-01-02 00:04:00  
TS[7,1]: 2003-01-02 01:59:00  
TS[7,2]: 2003-01-02 13:24:40  
TS[7,3]: 2003-01-02 13:24:40.495  
Date[8]: 1999-01-08 (N - F)  
TS[8,0]: 1999-01-08 00:04:00  
TS[8,1]: 1999-01-08 01:59:00  
TS[8,2]: 1999-01-08 13:24:40  
TS[8,3]: 1999-01-08 13:24:40.495  
Date[9]: 1999-01-08 (N - F)  
TS[9,0]: 1999-01-08 00:04:00  
TS[9,1]: 1999-01-08 01:59:00  
TS[9,2]: 1999-01-08 13:24:40  
TS[9,3]: 1999-01-08 13:24:40.495  
Date[10]: 1999-01-08 (N - F)  
TS[10,0]: 1999-01-08 00:04:00  
TS[10,1]: 1999-01-08 01:59:00
```

```
TS[10,2]: 1999-01-08 13:24:40
TS[10,3]: 1999-01-08 13:24:40.495
Date[11]: 1999-01-08 (N - F)
TS[11,0]: 1999-01-08 00:04:00
TS[11,1]: 1999-01-08 01:59:00
TS[11,2]: 1999-01-08 13:24:40
TS[11,3]: 1999-01-08 13:24:40.495
Date[12]: 1999-01-08 (N - F)
TS[12,0]: 1999-01-08 00:04:00
TS[12,1]: 1999-01-08 01:59:00
TS[12,2]: 1999-01-08 13:24:40
TS[12,3]: 1999-01-08 13:24:40.495
Date[13]: 2006-01-08 (N - F)
TS[13,0]: 2006-01-08 00:04:00
TS[13,1]: 2006-01-08 01:59:00
TS[13,2]: 2006-01-08 13:24:40
TS[13,3]: 2006-01-08 13:24:40.495
Date[14]: 1999-01-08 (N - F)
TS[14,0]: 1999-01-08 00:04:00
TS[14,1]: 1999-01-08 01:59:00
TS[14,2]: 1999-01-08 13:24:40
TS[14,3]: 1999-01-08 13:24:40.495
Date[15]: 1999-01-08 (N - F)
TS[15,0]: 1999-01-08 00:04:00
TS[15,1]: 1999-01-08 01:59:00
TS[15,2]: 1999-01-08 13:24:40
TS[15,3]: 1999-01-08 13:24:40.495
Date[16]: 1999-01-08 (N - F)
TS[16,0]: 1999-01-08 00:04:00
TS[16,1]: 1999-01-08 01:59:00
TS[16,2]: 1999-01-08 13:24:40
TS[16,3]: 1999-01-08 13:24:40.495
Date[17]: 1999-01-08 (N - F)
TS[17,0]: 1999-01-08 00:04:00
TS[17,1]: 1999-01-08 01:59:00
TS[17,2]: 1999-01-08 13:24:40
TS[17,3]: 1999-01-08 13:24:40.495
Date[18]: 1999-01-08 (N - F)
TS[18,0]: 1999-01-08 00:04:00
TS[18,1]: 1999-01-08 01:59:00
TS[18,2]: 1999-01-08 13:24:40
TS[18,3]: 1999-01-08 13:24:40.495
Date[19]: 0099-01-08 BC (N - F)
TS[19,0]: 0099-01-08 00:04:00 BC
TS[19,1]: 0099-01-08 01:59:00 BC
TS[19,2]: 0099-01-08 13:24:40 BC
interval[0]: @ 1 min
interval_copy[0]: @ 1 min
interval[1]: @ 1 day 12 hours 59 mins 10 secs
interval_copy[1]: @ 1 day 12 hours 59 mins 10 secs
interval[2]: @ 2 days 12 hours 59 mins 10 secs
interval_copy[2]: @ 2 days 12 hours 59 mins 10 secs
interval[3]: @ 1 day 12 hours 59 mins 10 secs
interval_copy[3]: @ 1 day 12 hours 59 mins 10 secs
interval[4]: @ 1 day 1 hour 1 min 1 sec
interval_copy[4]: @ 1 day 1 hour 1 min 1 sec
interval[5]: @ 1 year 59 mins
interval_copy[5]: @ 1 year 59 mins
Error parsing interval 6
```

5.8.5.3 解析、运算数值类型

此示例使用pgtypes库函数对numeric类型进行不同操作。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>
#include <pgtypes_error.h>
#include <decimal.h>
```

```
char* nums[] = { "2E394", "-2", ".794", "3.44", "592.49E21", "-32.84e4",
    "2E-394", ".1E-2", "+.0", "-592.49E-07", "+32.84e-4",
    ".500001", "-.5000001",
    "1234567890123456789012345678.91", /* 30个数位应转换为十进制 */
    "1234567890123456789012345678.921", /* 31个数位的数字不应转为十进制 */
    "not a number",
    NULL
};

static void check_errno(void);

int main(void)
{
    char *text="error\n";
    char *endptr;
    numeric *num, *nin;
    decimal *dec;
    long l;
    int i, j, k, q, r, count = 0;
    double d;
    numeric **numarr = (numeric **) calloc(1, sizeof(numeric));

    for (i = 0; nums[i]; i++)
    {
        /* 返回由malloc分配的字符串的指针，它包含numeric类型nums[i]的字符串表达 */
        num = PGTYPEStonumeric_from_asc(nums[i], &endptr);
        if (!num) check_errno();
        if (endptr != NULL)
        {
            printf("endptr of %d is not NULL\n", i);
            if (*endptr != '\0')
                printf("endptr of %d is not \\0\n", i);
        }
        if (!num) continue;

        numarr = (numeric **)realloc(numarr, sizeof(numeric *) * (count + 1));
        numarr[count++] = num;

        /* 返回由malloc分配的字符串的指针，它包含numeric类型num的字符串表达 */
        text = PGTYPEStonumeric_to_asc(num, -1);
        if (!text) check_errno();
        printf("num[%d,1]: %s\n", i, text); free(text);
        text = PGTYPEStonumeric_to_asc(num, 0);
        if (!text) check_errno();
        printf("num[%d,2]: %s\n", i, text); free(text);
        text = PGTYPEStonumeric_to_asc(num, 1);
        if (!text) check_errno();
        printf("num[%d,3]: %s\n", i, text); free(text);
        text = PGTYPEStonumeric_to_asc(num, 2);
        if (!text) check_errno();
        printf("num[%d,4]: %s\n", i, text); free(text);

        /* 请求一个指向新分配的numeric变量的指针 */
        nin = PGTYPEStonumeric_new();
        text = PGTYPEStonumeric_to_asc(nin, 2);
        if (!text) check_errno();
        printf("num[%d,5]: %s\n", i, text); free(text);

        /* 将一个numeric类型的变量转换为长整型 */
        r = PGTYPEStonumeric_to_long(num, &l);
        if (r) check_errno();
        printf("num[%d,6]: %ld (r: %d)\n", i, r?0L:l, r);
        if (r == 0)
        {
            /* 把一个长整型变量转换为一个numeric变量 */
            r = PGTYPEStonumeric_from_long(l, nin);
            if (r) check_errno();
            /* 返回由malloc分配的字符串的指针，它包含numeric类型nin的字符串表达 */
        }
    }
}
```

```
text = PGTYPEStnumeric_to_asc(nin, 2);
/* 比较两个numeric变量 */
q = PGTYPEStnumeric_cmp(num, nin);
printf("num[%d,7]: %s (r: %d - cmp: %d)\n", i, text, r, q);
free(text);
}

/* 将一个numeric类型的变量转换成整数 */
r = PGTYPEStnumeric_to_int(num, &k);
if (r) check_errno();
printf("num[%d,8]: %d (r: %d)\n", i, r?0:k, r);
if (r == 0)
{
    /* 把一个整数变量转换成一个numeric变量 */
    r = PGTYPEStnumeric_from_int(k, nin);
    if (r) check_errno();
    /* 返回由malloc分配的字符串的指针，它包含numeric类型nin的字符串表达 */
    text = PGTYPEStnumeric_to_asc(nin, 2);
    q = PGTYPEStnumeric_cmp(num, nin);
    printf("num[%d,9]: %s (r: %d - cmp: %d)\n", i, text, r, q);
    free(text);
}

if (i != 6)
{
    /* 将一个numeric类型的变量转换成双精度类型 */
    r = PGTYPEStnumeric_to_double(num, &d);
    if (r) check_errno();
    printf("num[%d,10]: %g (r: %d)\n", i, r?0.0:d, r);
}

/* 请求一个指向新分配的numeric变量的指针 */
dec = PGTYPEStdecimal_new();
/* 将一个decimal类型的变量转换成numeric */
r = PGTYPEStnumeric_to_decimal(num, dec);
if (r) check_errno();
printf("num[%d,11]: - (r: %d)\n", i, r);
if (r == 0)
{
    /* 将一个decimal类型的变量转换成numeric */
    r = PGTYPEStnumeric_from_decimal(dec, nin);
    if (r) check_errno();
    /* 返回由malloc分配的字符串的指针，它包含numeric类型nin的字符串表达 */
    text = PGTYPEStnumeric_to_asc(nin, 2);
    /* 比较两个numeric变量 */
    q = PGTYPEStnumeric_cmp(num, nin);
    printf("num[%d,12]: %s (r: %d - cmp: %d)\n", i, text, r, q);
    free(text);
}

/* 释放numeric变量的内存 */
PGTYPEStdecimal_free(dec);
PGTYPEStnumeric_free(nin);
printf("\n");
}

for (i = 0; i < count; i++)
{
    for (j = 0; j < count; j++)
    {
        /* 请求一个指向新分配的numeric变量的指针 */
        numeric* a = PGTYPEStnumeric_new();
        numeric* s = PGTYPEStnumeric_new();
        numeric* m = PGTYPEStnumeric_new();
        numeric* d = PGTYPEStnumeric_new();
        /* 把两个numeric变量相加放到第三个numeric变量中 */
        r = PGTYPEStnumeric_add(numarr[i], numarr[j], a);
        if (r)
        {

```

```
        check_errno();
        printf("r: %d\n", r);
    }
    else
    {
        /* 返回由malloc分配的字符串的指针，它包含numeric类型a的字符串表达 */
        text = PGTYPENumeric_to_asc(a, 10);
        printf("num[a,%d,%d]: %s\n", i, j, text);
        free(text);
    }
    /* 把两个numeric变量相减并且把结果返回到第三个numeric变量 */
    r = PGTYPENumeric_sub(numarr[i], numarr[j], s);
    if (r)
    {
        check_errno();
        printf("r: %d\n", r);
    }
    else
    {
        /* 返回由malloc分配的字符串的指针，它包含numeric类型s的字符串表达 */
        text = PGTYPENumeric_to_asc(s, 10);
        printf("num[s,%d,%d]: %s\n", i, j, text);
        free(text);
    }
    /* 把两个numeric变量相乘并且把结果返回到第三个numeric变量 */
    r = PGTYPENumeric_mul(numarr[i], numarr[j], m);
    if (r)
    {
        check_errno();
        printf("r: %d\n", r);
    }
    else
    {
        /* 返回由malloc分配的字符串的指针，它包含numeric类型m的字符串表达 */
        text = PGTYPENumeric_to_asc(m, 10);
        printf("num[m,%d,%d]: %s\n", i, j, text);
        free(text);
    }
    /* 把两个numeric变量相除并且把结果返回到第三个numeric变量 */
    r = PGTYPENumeric_div(numarr[i], numarr[j], d);
    if (r)
    {
        check_errno();
        printf("r: %d\n", r);
    }
    else
    {
        /* 返回由malloc分配的字符串的指针，它包含numeric类型d的字符串表达 */
        text = PGTYPENumeric_to_asc(d, 10);
        printf("num[d,%d,%d]: %s\n", i, j, text);
        free(text);
    }
}

/* 释放一个numeric变量的内存 */
PGTYPENumeric_free(a);
PGTYPENumeric_free(s);
PGTYPENumeric_free(m);
PGTYPENumeric_free(d);
}

for (i = 0; i < count; i++)
{
    /* 返回由malloc分配的字符串的指针，它包含numeric类型numarr[i]的字符串表达 */
    text = PGTYPENumeric_to_asc(numarr[i], -1);
    printf("%d: %s\n", i, text);
    free(text);
    /* 释放内存 */
    PGTYPENumeric_free(numarr[i]);
}
```


[illegible]

5.8.6 ecpg 接口参考

5.8.6.1 概述

ecpg接口参考主要介绍pgtypes库提供的用户在嵌入式SQL-C源码程序中可使用的数据类型相关接口。

pgtypes库将SQL数据类型映射到C语言数据类型，并提供一些接口实现其基本功能和运算。

5.8.6.2 区间类型

表5-132列出了ecpg提供的区间类型（interval）数据的常用接口：

表 5-132 区间类型常用接口

API接口	接口描述	说明
interval* PGTYPEInterval_new(void)	返回一个指向新分配的区间变量的指针。	该函数在堆上创建interval变量，返回值为interval*类型。
void PGTYPEInterval_free(interval *inval)	释放先前分配的区间变量的内存。	释放PGTYPEInterval_new函数创建的interval*类型变量。
interval* PGTYPEInterval_from_asc(char *str, char **endptr)	解析文本表示的区间。	该函数解析输入字符串str并且返回一个已分配的区间变量的指针。ecpg支持解析完整的字符串，目前不支持把第一个非法字符的地址存储在*endptr中，可以把endptr设置为NULL。
char* PGTYPEInterval_to_asc(interval *span)	将一个区间类型的变量转换成它的文本表达。	该函数将span指向的区间变量转换成一个char*。
int PGTYPEInterval_copy(interval *intvlsrc, interval *intvldest)	复制一个区间类型的变量。	该函数将intvlsrc指向的区间变量复制到intvldest指向的区间变量。

示例

请参见[典型应用开发示例](#)章节。

5.8.6.3 数值类型

表5-133列出了ecpg提供的数值类型（numeric/decimal）数据的常用接口：

表 5-133 数值类型常用接口

API接口	接口描述	说明
numeric* PGTPENumeric_new(void)	请求一个指向新分配的numeric变量的指针。	该函数在堆上创建numeric*类型变量，返回值为numeric*类型。

API接口	接口描述	说明
decimal* PGTYPE\$decimal_new(vo id)	请求一个指向新分配的 decimal变量的指针。	该函数在堆上创建 decimal*类型变量，返回 值为decimal*类型。
void PGTYPE\$numeric_free(nu meric *var)	释放先前分配的numeric* 类型变量的内存。	该函数释放通过 PGTYPE\$numeric_new函 数创建的numeric*类型变 量。
void PGTYPE\$decimal_free(de cimal *dst)	释放先前分配的decimal* 类型变量的内存。	该函数释放通过 PGTYPE\$decimal_new函 数创建的decimal*类型变 量。
numeric* PGTYPE\$numeric_from_a sc(char *str, char **endptr)	用于将字符串形式的数值 转换为numeric类型。	有效格式如：-2、.794、 +3.44、 592.49E07、-32.84e-4。 如果解析转换成功，则返 回一个有效指针，否则返 回空指针。ecpg支持解析 完整的字符串，目前不支 持把第一个非法字符的地 址存储在*endptr中，可以 把endptr设置为NULL。
char* PGTYPE\$numeric_to_asc(numeric *num, int dscale)	将一个numeric类型变量 转换为字符串类型。	numeric值将被使用dscale 小数位打印，必要时会取 整。
int PGTYPE\$numeric_add(nu meric *var1, numeric *var2, numeric *result)	将两个numeric变量相加 放到第三个numeric变量 中。	该函数把变量var1和var2 相加放到结果变量result 中。成功时该函数返回0， 报错时返回-1。
int PGTYPE\$numeric_sub(nu meric *var1, numeric *var2, numeric *result)	将两个numeric变量相减 并且把结果返回到第三个 numeric变量。	该函数把变量var2从变量 var1中减除。该操作的结 果被存储在变量result中。 成功时该函数返回0，报错 时返回-1。
int PGTYPE\$numeric_mul(nu meric *var1, numeric *var2, numeric *result)	将两个numeric变量相乘 并且把结果返回到第三个 numeric变量。	该函数把变量var1和var2 相乘。该操作的结果被存 储在变量result中。成功时 该函数返回0，报错时返 回-1。
int PGTYPE\$numeric_div(nu meric *var1, numeric *var2, numeric *result)	将两个numeric变量相除 并且把结果返回到第三个 numeric变量。	该函数用变量var2除变量 var1。该操作的结果被存 储在变量result中。成功时 该函数返回0，报错时返 回-1。

API接口	接口描述	说明
int PGTYPESnumeric_cmp(n umeric *var1, numeric *var2)	比较两个numeric变量。	该函数比较两个numeric变量。报错时会返回INT_MAX。成功时，该函数返回三种可能结果： - 如果var1大于var2，则返回1。 - 如果var1小于var2，则返回-1。 - 如果var1和var2相等，则返回0。
int PGTYPESnumeric_from_i nt(signed int int_val, numeric *var)	将一个整型变量转换成numeric类型。	该函数接受一个有符号整型变量并且把它存储在numeric变量var中。成功时返回0，报错时返回-1。
int PGTYPESnumeric_from_l ong(signed long int long_val, numeric *var)	将一个长整型变量转换成numeric类型。	该函数接受一个有符号长整型变量并且把它存储在numeric变量var中。成功时返回0，报错时返回-1。
int PGTYPESnumeric_copy(n umeric *src, numeric *dst)	将一个numeric变量复制到另一个numeric变量中。	该函数把src指向的变量的值复制到dst指向的变量。成功时该函数返回0，报错时返回-1。
int PGTYPESnumeric_from_d ouble(double d, numeric *dst)	将一个双精度类型的变量转换成numeric类型。	该函数接受一个双精度类型的变量并且把结果存储在dst指向的变量中。成功时该函数返回0，报错时返回-1。
int PGTYPESnumeric_to_dou ble(numeric *nv, double *dp)	将一个numeric类型的变量转换成双精度类型。	该函数将nv指向的变量中的numeric值转换成dp指向的双精度变量。成功时该函数返回0，报错时返回-1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。
int PGTYPESnumeric_to_int(numeric *nv, int *ip)	将一个numeric类型的变量转换成整型类型。	该函数将nv指向的变量的numeric值转换成ip指向的整型变量。成功时该函数返回0，报错时返回-1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。

API接口	接口描述	说明
int PGTYPESnumeric_to_long(numeric *nv, long *ip)	将一个numeric类型的变量转换成长整型类型。	该函数将nv指向的变量的numeric值转换成ip指向的长整型变量。成功时该函数返回0，报错时返回-1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst)	将一个numeric类型的变量转换成decimal类型。	该函数将src指向的变量的numeric值转换成dst指向的decimal变量。成功时该函数返回0，报错时返回-1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst)	将一个decimal类型的变量转换成numeric类型。	该函数将src指向的变量的decimal值转换成dst指向的numeric变量。成功时该函数返回0，报错时返回-1（包括溢出）。

示例

请参见[典型应用开发示例](#)章节。

5.8.6.4 日期类型

[表5-134](#)列出了ecpg提供的日期类型（date）数据的常用接口：

表 5-134 日期类型常用接口

API接口	接口描述	说明
date* PGTYPESdate_new(void)	请求一个指向新分配的date变量的指针。	该函数在堆上创建date*类型变量，返回值为date*类型。
void PGTYPESdate_free(date *d)	释放先前分配的date*类型变量的内存。	该函数释放通过PGTYPESdate_new函数创建的date*类型变量。

API接口	接口描述	说明
date PGTYPESdate_from_asc(char *str, char **endptr)	从日期的文本表达解析一个日期。	该函数接收一个字符串str以及一个指向字符串的指针endptr，将字符串形式转换为日期类型。当前不支持将第一个非法字符的地址存储在*endptr中，可以把endptr设置为NULL。 须知 该函数假定日期格式按照MDY进行格式化，并且当前在ecpg中没有变体可以改变这种格式。
char* PGTYPESdate_to_asc(date dDate)	返回一个日期变量的文本表达。	该函数接收日期dDate作为它的唯一参数。以YYYY-MM-DD格式输出。
date PGTYPESdate_from_timestamp(timestamp dt)	从一个时间戳中抽取日期部分。	该函数接收一个时间戳作为它的唯一参数并且从这个时间戳返回抽取的日期部分。
void PGTYPESdate_julmdy(date jd, int *mdy)	从一个日期类型变量中抽取日、月和年的值。	该函数接收日期jd以及一个指向有3个整数值数组mdy的指针。变量名在这里表明了顺序：mdy[0]表示月份，mdy[1]表示日期，而mdy[2]表示年份。
void PGTYPESdate_mdyjul(int *mdy, date *jdate)	使用指定的整型数值创建日期。	该函数接收3个整数组成的数组mdy作为其第一个参数，三个整数分别用来表示日、月和年。第二个参数是一个指向日期类型变量的指针，它被用来保存操作的结果。
int PGTYPESdate_dayofweek(date d)	接收一个日期变量，然后返回一个整数说明这个日期是星期几。	该函数接收日期变量d作为它唯一的参数，并且返回一个整数，说明这个日期是星期几。 • 0：星期日 • 1：星期一 • 2：星期二 • 3：星期三 • 4：星期四 • 5：星期五 • 6：星期六

API接口	接口描述	说明
void PGTYPESdate_today(date *d)	获取当前日期。	该函数用于获取当前日期并将其存储在date类型的变量中。
int PGTYPESdate_defmt_asc(date *d, const char *fmt, char *str)	使用一个格式掩码把一个char*字符串转换成一个日期类型的值。	该函数接收一个用来保存操作结果的指向日期值的指针d、用于解析日期的格式掩码fmt以及包含日期文本表达的char*字符串str。该函数期望文本表达匹配格式掩码。不需要字符串和格式掩码的一一映射。该函数只分析相继顺序并且查找表示年份位置的yy或者yyyy、表示月份位置的mm以及表示日位置的dd。注意不能使用中文年月日。
int PGTYPESdate_fmt_asc(date dDate, const char *fmtstring, char *outbuf)	使用一个格式掩码将一个日期类型的变量转换成它的文本表达。	该函数接收要转换的日期dDate、格式掩码fmtstring以及将要保存日期的文本表达的字符串outbuf。 成功时，返回0；如果发生报错，则返回一个负值。 合法输入示例如下： <pre>112359 //mmddyy 59/11/23 //yy/mm/dd Nov.23,1959 //mmm.dd,yyyy Mon,Nov.23,1959 // ddd,mmm.dd,yyyy</pre> 格式说明： • dd：一个月中的第几天。 • mm：一年中的第几个月。 • yy：两位数的年份。 • yyyy：四位数的年份。 • ddd：星期几的名称（简写）。 • mmm：月份的名称（简写）。

示例

请参见[典型应用开发示例](#)章节。

5.8.6.5 时间戳类型

表5-135列出了ecpg提供的时间戳（timestamp）数据的常用接口：

表 5-135 时间戳类型常用接口

API接口	接口描述	说明
timestamp PGTYPETimestamp_from_asc(char *str, char **endptr)	从文本解析一个时间戳并放到一个时间戳变量中。	该函数接收一个字符串str以及一个指向字符串的指针endptr。成功时函数返回解析的时间戳，报错时返回PGTYPESInvalidTimestamp，并且设置errno为PGTYPES_TS_BAD_TIMESTAMP。 PGTYPETimestamp_from_asc的合法输入如下所示： 1999-01-08 04:05:06 January 8 04:05:06 1999 PST 1999-Jan-08 04:05:06.789-8 1999-01-08 04:05:06.789（忽略了时区指示符） J2451187 04:05-08:00 1999-01-08 04:05:00（忽略了时区指示符）
char *PGTYPETimestamp_to_asc(timestamp tstamp)	将时间戳转换成char*字符串。	该函数接收时间戳tstamp作为它的唯一参数并且返回一个分配好的包含该时间戳文本表达式的字符串。结果必须用PGTYPESchar_free()释放。
void PGTYPETimestamp_current(timestamp *ts)	获取当前时间戳。	该函数获取当前时间戳，并且将它保存到ts指向的时间戳变量中。
int PGTYPETimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr)	使用一个格式掩码将一个时间戳变量转换成char*字符串。	该函数接收一个指向时间戳的指针ts、一个指向输出缓冲区的指针output、一个为输出缓冲区分配的最大长度str_len以及用于转换的格式掩码fmtstr作为参数。 成功时该函数返回0，报错时则返回一个负值。

API接口	接口描述	说明
int PGYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv)	从一个时间戳中减去另一个时间戳，并且把结果保存在一个区间类型的变量中。	该函数将从ts1指向的时间戳变量中减去ts2指向的时间戳变量，并且把结果存储在iv指向的区间变量中。 成功时该函数返回0，报错时则返回一个负值。
int PGYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d)	将一个字符串按照指定的格式解析为timestamp类型。	该函数接收一个放在变量str中的时间戳文本表达以及放在变量fmt中要使用的格式掩码。结果将被存放在d指向的变量中。 如果格式掩码fmt是NULL，该函数将使用默认的格式掩码%Y-%m-%d %H:%M:%S。
int PGYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout)	把一个interval变量加到一个时间戳变量上。	该函数接收一个指向时间戳变量的指针tin以及一个指向interval变量的指针span，把interval加到时间戳上，然后将结果时间戳保存在tout指向的变量中。 成功时该函数返回0，报错时则返回一个负值。
int PGYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout)	从一个timestamp时间戳变量中减去interval变量。	该函数从tin指向的时间戳变量中减去span指向的interval变量，然后把结果保存在tout指向的变量中。 成功时该函数返回0，报错时则返回一个负值。

示例

请参见[典型应用开发示例](#)章节。

5.9 兼容性参考

Go 驱动兼容

- 数据库的Go驱动当前不适配业界成熟ORM框架（比如xorm）。
- 在创建数据库连接时传入的驱动名称兼容“opengauss”、“postgres”、“postgresql”。

- 设置环境变量GOENV_REGIST_ALL为false后，支持GaussDB的Go驱动与PG的Go驱动并存；若环境变量GOENV_REGIST_ALL缺省或设置为true，则不支持GaussDB的Go驱动与PG的Go驱动并存，因此依赖PG的Go驱动的代码会报错。

ecpg 兼容

ecpg提供使用URL连接方式的连接语法，不仅支持“gaussdb”，还兼容支持“postgresql”。

连接语法：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

其中target支持“postgresql”方式如下：

- tcp:postgresql://hostname[:port][/dbname][?options]
- unix:postgresql://hostname[:port][/dbname][?options]

5.10 附录

5.10.1 JDBC

5.10.1.1 数据类型映射关系

数据类型、JAVA变量类型以及JDBC类型索引关系如表5-136所示。ORA：Oracle，MYSQL：MySQL。

- 非透明多写特性下，兼容Oracle/MySQL。

表 5-136 JDBC 数据类型映射关系

兼容模式	GaussDB数据类型	JAVA变量类型	JDBC类型索引
ORA/MYSQL	oid	java.lang.Long	java.sql.Types.BIGINT
ORA/MYSQL	numeric	java.math.BigDecimal	java.sql.Types.NUMERIC
ORA/MYSQL	tinyint	java.lang.Integer	java.sql.Types.TINYINT
ORA/MYSQL	smallint	java.lang.Integer	java.sql.Types.SMALLINT
ORA/MYSQL	bigint	java.lang.Long	java.sql.Types.BIGINT
ORA/MYSQL	float4	java.lang.Float	java.sql.Types.REAL
ORA/MYSQL	float8	java.lang.Double	java.sql.Types.DOUBLE
ORA/MYSQL	char	java.lang.String	java.sql.Types.CHAR
ORA/MYSQL	character	java.lang.String	java.sql.Types.CHAR
ORA/MYSQL	bpchar	java.lang.String	java.sql.Types.CHAR

兼容模式	GaussDB数据类型	JAVA变量类型	JDBC类型索引
ORA/MYSQL	character varying	java.lang.String	java.sql.Types.VARCHAR
ORA/MYSQL	varchar	java.lang.String	java.sql.Types.VARCHAR
ORA/MYSQL	text	java.lang.String	java.sql.Types.VARCHAR
ORA/MYSQL	name	java.lang.String	java.sql.Types.VARCHAR
ORA/MYSQL	bytea	byte[]	java.sql.Types.BINARY
ORA/MYSQL	blob	java.sql.Blob	java.sql.Types.BLOB
ORA/MYSQL	clob	java.sql.Clob	java.sql.Types.CLOB
ORA/MYSQL	boolean	java.lang.Boolean	java.sql.Types.BIT
ORA	集合类型	java.sql.Array	java.sql.Types.ARRAY
ORA	数组类型	java.sql.Array	java.sql.Types.ARRAY
ORA	record类型	java.sql.Struct	java.sql.Types.STRUCT
MYSQL	date	java.sql.Date	java.sql.Types.DATE
ORA/MYSQL	time	java.sql.Time	java.sql.Types.TIME
ORA/MYSQL	timetz	java.sql.Time	java.sql.Types.TIME
ORA/MYSQL	timestamp	java.sql.Timestamp	java.sql.Types.TIMESTAMP
ORA/MYSQL	smalldatetime	java.sql.Timestamp	java.sql.Types.TIMESTAMP
ORA/MYSQL	timestampz	java.sql.Timestamp	java.sql.Types.TIMESTAMP
ORA/MYSQL	refcursor	java.sql.ResultSet	java.sql.Types.REF_CURSOR java.sql.Types.OTHER

5.10.1.2 数据类型转换规则

数据类型、JAVA变量类型以及JDBC类型索引关系如表5-136所示。

- 非透明多写特性下，兼容ORA/MYSQL。

说明

- 发送到服务端的类型非最终存储到数据库表中的类型，下表中发送到数据库的类型对应到数据库表中的类型请参见《参考》中“SQL参考 > 类型转换 > 值存储”章节，其中的类型转换规则同样适用于MYSQL兼容模式。
- JDBC连接参数详情请参见[连接参数参考](#)。

表 5-137 PreparedStatement 类 set 接口数据转换规则

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
setNull(int parameterIndex, int sqlType)	OR A/ MY SQL	int sqlType	java.lang.I nteger	-	接口参数 sqlType传 入值对应发 送到数据库 类型请参见 表5-138。	sqlType 为 java.sql. Types中 定义的 sql类型 值。
void setNull(int parameterIndex, int sqlType, String typeName)	OR A/ MY SQL	int sqlType	java.lang.I nteger	参数 String typeName=null	请参见表 5-138。	-
void setBoolean(int parameterIndex, boolean x)	OR A/ MY SQL	boolean x	java.lang. Boolean	连接参数 use_bool ean=true	BOOL	-
	OR A/ MY SQL	boolean x	java.lang. Boolean	连接参数 use_bool ean=false（默 认）	INT2	-
void setByte(int parameterIndex, byte x)	OR A/ MY SQL	byte x	byte	-	INT2	-
void setShort(int parameterIndex, short x)	OR A/ MY SQL	short x	java.lang. Short	-	INT2	-
void setInt(int parameterIndex, int x)	OR A/ MY SQL	int x	java.lang.I nteger	-	INT4	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setLong(int parameterIndex, long x)	ORA/ MySQL	long x	java.lang. Long	-	INT8	-
void setFloat(int parameterIndex, float x)	ORA	float x	java.lang. Float	连接参数 binaryTransfer=true	FLOAT4	-
	ORA	float x	java.lang. Float	连接参数 binaryTransfer=false（默认） 并且连接参数 setFloat=true	FLOAT4	-
	ORA	float x	java.lang. Float	连接参数 binaryTransfer=false（默认） 并且连接参数 setFloat=false（默认）	FLOAT8	-
	MySQL	float x	java.lang. Float	-	FLOAT4	-
void setDouble(int parameterIndex, double x)	ORA/ MySQL	double x	java.lang. Double	-	FLOAT8	-
void setBigDecimal(int parameterIndex, BigDecimal x)	ORA/ MySQL	BigDecimal x	java.math. BigDecimal	-	NUMERIC	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setString(int parameterIndex, String x)	ORA/ MY SQL	String x	java.lang. String	连接参数 stringtype=varchar（默认）	VARCHAR	-
	ORA/ MY SQL	String x	java.lang. String	连接参数 stringtype=unspecified	UNSPECIFIED	-
void setBytes(int parameterIndex, byte x[])	ORA/ MY SQL	byte x[]	[]byte	-	BYTEA	-
void setDate(int parameterIndex, java.sql.Date x, Calendar cal)	ORA/ MY SQL	-	-	-	-	与void setDate(int parameterIndex, java.sql. Date x)一致。
void setDate(int parameterIndex, java.sql.Date x)	ORA	java.sql. Date x	java.lang. Date	连接参数 binaryTransfer=true 并且数据库GUC 参数 mapping_date_to_datea 取值为on	DATEA	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	java.sql .Date x	java.lang. Date	连接参数 binaryTransfer=true 并且数据库GUC 参数 mapping_date_to_datea 取值为 off	UNSPECIFIED	-
	ORA/ MY SQL	java.sql .Date x	java.lang. Date	连接参数 binaryTransfer=false（默认） 并且 ORA兼容场景下 需要数据库连接参数 oracleCompatible不配置 为true并且配置不包含 setDateTime	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA	java.sql.Date x	java.lang.Date	连接参数oracleCompatible配置为true或者配置包含setDateTime 并且连接参数binaryTransfer=false（默认） 并且数据库GUC参数mapping_date_to_datea取值为on	DATEA	-
	ORA	java.sql.Date x	java.lang.Date	连接参数oracleCompatible配置为true或者配置包含setDateTime 并且连接参数binaryTransfer=false（默认） 并且数据库GUC参数mapping_date_to_datea取值为off	TIMESTAMP	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
void setTime(int parameterIndex, java.sql.Time x, Calendar cal)	ORA/MYSQL	-	-	-	-	与void setTime(int parameterIndex, java.sql.Time x)一致。
void setTime(int parameterIndex, java.sql.Time x)	ORA/MYSQL	java.sql.Time x	java.lang.Time	ORA兼容场景下需要连接参数oracleCompatible不配置为true并且配置不包含setDateTime	UNSPECIFIED	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); Time time1 = Time.valueOf("12:30:45"); ps.setTime(1, time1);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MySQL	java.sql. Time x	PGTime	参数 java.sql. Time x 未设置时 区	TIME	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); Time time1 = Time.valueOf("12:30: 45"); PGTime time2 = new PGTime(time1.getTime()); ps.setTime(1, time2);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	java.sql. Time x	PGTime	参数 java.sql. Time x 设置了时 区并且不 为null	TIMETZ	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); Time time1 = Time.valueOf("12:30:45"); Calendar calendar = Calendar.getInstance(); PGTime time2 = new PGTime(time1.getTime(), calendar); ps.setTime(1, time2);
	ORA/MYSQL	java.sql. Time x	java.lang. Time	参数 java.sql. Time x 为null	UNSPECIFIED	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); ps.setTime(1, null);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA	java.sql. Time x	java.lang. Time	连接参数 oracleCo mpatibl e配置为 true或者 配置包含 setDate Time	TIMETZ	-
void setTimestamp(in t parameterIndex, java.sql.Timesta mp x, Calendar cal)	ORA/ MY SQL	-	-	-	-	与void setTim estam p(int parame terInde x, java.sql. .Timest amp x) 一致。
void setTimestamp(in t parameterIndex, java.sql.Timesta mp x)	ORA/ MY SQL	java.sql .Timest amp x	java.sql.Ti mestamp	ORA兼 容场景下 需要连接 参数 oracleCo mpatibl e不配置 为true并 且配置不 包含 setDate Time (默认)	UNSPECIFI ED	代码示 例： PreparedSt atement ps = conn.prep areStatem ent("insert into ptest_tabl e values (?)"); Timestam p timestamp 1 = Timestam p.valueOf("2024-01- 01 12:30:45.8 88"); ps.setTime stamp(1, timestamp 1);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MySQL	java.sql. Timestamp x	PGTimestamp	参数 java.sql. Timestamp x未 设置时区	TIMESTAMP	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); Timestamp timestamp1 = Timestamp.valueOf("2024-01-01 12:30:45.888"); PGTimestamp timestamp2 = new PGTimestamp(timestamp1.getTime()); ps.setTimestamp(1, timestamp2);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	java.sql. Timestamp x	PGTimestamp	参数 java.sql. Timestamp x设置了时区并且不为null	TIMESTAMP PTZ	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); Timestamp timestamp1 = Timestamp.valueOf("2024-01-01 12:30:45.888"); Calendar calendar = Calendar.getInstance(); PGTimestamp timestamp2 = new PGTimestamp(timestamp1.getTime(), calendar); ps.setTimestamp(1, timestamp2);
	ORA/ MY SQL	java.sql. Timestamp x	java.sql.Timestamp	参数 java.sql. Timestamp x为null	UNSPECIFIED	代码示例： PreparedStatement ps = conn.prepareStatement("insert into ptest_table values (?)"); ps.setTimestamp(1, null);

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA	java.sql.Timestamp x	java.sql.Timestamp	连接参数oracleCompatible配置为true或者配置包含setDateTime	TIMESTAMPZ	-
void setAsciiStream(int parameterIndex, java.io.InputStream x)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype不为 unspecified	VARCHAR	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x不为 null	VARCHAR	-
void setAsciiStream(int parameterIndex, java.io.InputStream x, int length)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype不为 unspecified（默认）	VARCHAR	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x不为 null	VARCHAR	-
void setAsciiStream(int parameterIndex, java.io.InputStream x, long length)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype不为 unspecified	VARCHAR	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x不为 null	VARCHAR	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setBinaryStream(int parameterIndex, java.io.InputStream x)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=on	BLOB	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=off	BYTEA	-
void setBinaryStream(int parameterIndex, java.io.InputStream x, int length)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=on	BLOB	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=off	BYTEA	-
void setBinaryStream(int parameterIndex, java.io.InputStream x,long length)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=true	BLOB	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	连接参数 blobMode=false (默认)	BYTEA	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setCharacterStream(int parameterIndex, java.io.Reader reader,long length)	ORA/ MY SQL	java.io. Reader reader	java.io.Re ader	连接参数 stringtype=unspecified	UNSPECIFIED	与void setCharacterStream(int parameterIndex, java.io. Reader reader,int length) 接口一致。
void setCharacterStream(int parameterIndex, java.io.Reader reader,int length)	ORA/ MY SQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不为null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-
	ORA/ MY SQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不为null 并且连接 参数 stringtype不为unspecified（默认）	VARCHAR	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不 为null 并且连接 参数 stringtyp e=unspe cified	UNSPECIFI ED	-
	ORA	java.io. Reader reader	java.io.Re ader	参数 javaio.Re ader reader不 为null 并且连接 参数 stringtyp e不为 unspecifi ed（默 认）	VARCHAR	-
	MY SQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader为 null	UNSPECIFI ED	-
void setCharacterStre am(int parameterIndex, java.io.Reader reader)	ORA/ MY SQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader为 null 并且连接 参数 stringtyp e=unspe cified	UNSPECIFI ED	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MYSQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader为 null 并且连接 参数 stringtyp e不为 unspecifi ed（默认）	VARCHAR	-
	ORA/ MYSQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不为null 并且连接 参数 preferQ ueryMo de=simp le 并且连接 参数 stringtyp e=unspe cified	UNSPECIFI ED	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MYSQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不为null 并且连接 参数 preferQ ueryMo de=simp le 并且连接 参数 stringtyp e不为 unspecifi ed（默 认）	VARCHAR	-
	ORA/ MYSQL	java.io. Reader reader	java.io.Re ader	参数 java.io.R eader reader不为null 并且连接 参数 preferQ ueryMo de不为 simple 并且连接 参数 stringtyp e不为 unspecifi ed（默 认）	TEXT	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setUnicodeStream(int parameterIndex, java.io.InputStream x, int length)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x为 null 并且连接 参数 stringtype不为 unspecified（默认）	VARCHAR	-
	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	参数 java.io.InputStream x不为 null	VARCHAR	-
void setBlob(int parameterIndex, Blob x)	ORA/ MY SQL	Blob x	java.sql.BLOB	-	BLOB	-
void setBlob(int parameterIndex, InputStream inputStream)	ORA/ MY SQL	java.io.InputStream x	java.io.InputStream	-	BLOB	-
void setBlob(int parameterIndex, InputStream inputStream, long length)	ORA/ MY SQL	InputStream inputStream	java.io.InputStream	-	BLOB	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setClob(int parameterIndex, Clob x)	OR A/ MY SQL	Clob x	java.sql.CLOB	参数 Clob x为 null	CLOB	-
	OR A/ MY SQL	Clob x	java.sql.CLOB	参数 Clob x不为null 并且连接 参数 stringtype=unspecified	UNSPECIFIED	-
	OR A/ MY SQL	Clob x	java.sql.CLOB	参数 Clob x不为null 并且连接 参数 stringtype不为unspecified（默认）	VARCHAR	-
void setClob(int parameterIndex, Reader reader)	OR A/ MY SQL	Reader reader	java.io.Reader	连接参数 stringtype=unspecified	UNSPECIFIED	-
	OR A/ MY SQL	Reader reader	java.io.Reader	连接参数 stringtype不为unspecified	VARCHAR	-
void setClob(int parameterIndex, Reader reader, long length)	OR A/ MY SQL	Reader reader	java.io.Reader	连接参数 stringtype=unspecified	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	Reader reader	java.io.Re ader	连接参数 stringtype不为 unspecified（默认）	VARCHAR	-
void setArray(int parameterIndex, Array x)	ORA/MYSQL	Array x	java.sql.Ar ray	参数 Array x 为null	UNSPECIFI ED	-
	ORA/MYSQL	Array x	java.sql.Ar ray	参数 Array x 不为null	根据创建 PGArray或者 GaussArray 的参数决定。	-
	ORA	Array x	GaussArra y	-	-	-
	ORA	Array x	GaussArra y	参数 Array x 为null	UNSPECIFI ED	-
void setRowId(int parameterIndex, RowId x)	ORA/MYSQL	RowId x	java.sql.RowId	-	ROWID	-
void setNString(int parameterIndex, String value)	ORA/MYSQL	String x	java.lang.String	连接参数 stringtype=varchar（默认）	VARCHAR	-
	ORA/MYSQL	String x	java.lang.String	连接参数 stringtype=unspecified	UNSPECIFI ED	-
void setSQLXML(int parameterIndex, SQLXML xmlObject)	ORA/MYSQL	SQLXML xmlObj ect	java.sql.S QLXML	-	XML	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
void setObject(int parameterIndex, Object x, int targetSqlType)	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 为null	请参见表5-138。	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.SQLXML	请参见void setSQLXML(int parameterIndex, SQLXML xmlObject)接口。	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.INTEGER	请参见void setInt(int parameterIndex, int x)接口。	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.TINYINT	INT1	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.SMALLINT	INT2	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.BIGINT	INT8	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.REAL	请参见 void setFloat(int parameter Index, float x) 接口。	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.FLOAT 并且连接参数 setFloat=true	FLOAT4	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.FLOAT 并且连接 参数 setFloat=false	FLOAT8	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.DOUBLE	FLOAT8	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.DECIMAL	NUMERIC	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.NUMERIC	NUMERIC	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.CHAR	BPCHAR	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.VARCHAR	VARCHAR	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.LONGVARCHAR 并且参数Object x类型为InputStream	TEXT	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.LONGVARCHAR 并且 Object x 类型不为 InputStream	VARCHAR	-
	ORA	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.DATE 并且参数 Object x 类型不是 LocalDate 并且连接 参数 binaryTransfer=true 并且数据库GUC 参数 mapping_date_to_datea 取值为 on	DATEA	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.DATE 并且参数 Object x 类型不是 LocalDate 并且连接参数 binaryTransfer=true 并且数据库GUC 参数 mapping_date_to_date取值不为 off	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	MySQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DATE 并且参数Object x类型不是LocalDate 并且连接参数binaryTransfer=false（默认）	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DATE 并且参数Object x类型不是LocalDate 并且连接参数binaryTransfer=false（默认） 并且连接参数oracleCompatible不配置为true并且配置不包含setDateTime	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DATE 并且参数Object x类型不是LocalDate 并且连接参数binaryTransfer=false（默认） 并且连接参数oracleCompatible配置为true或者配置包含setDateTime 并且数据库GUC参数mapping_date_to_datea取值为on	DATEA	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DATE 并且参数Object x类型不是LocalDate 并且连接参数binaryTransfer=false（默认） 并且连接参数oracleCompatible配置为true或者配置包含setDateTime 并且数据库GUC参数mapping_date_to_datea取值为off	TIMESTAMP	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DATE 并且参数Object x类型是LocalDate 并且连接参数binaryTransfer=true 并且数据库GUC参数mapping_date_to_datea取值为on	DATEA	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.DATE 并且参数 Object x 类型是 LocalDate 并且连接参数 binaryTransfer=true 并且数据库GUC 参数 mapping_date_to_datea 取值为 on	DATE	-
	ORA/MySQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.TIME 并且参数 Object x 类型不是 LocalTime	请参见 void setTime(int parameter Index, java.sql.Time x) 接口。	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.TIME 并且参数Object x类型是LocalTime	TIME	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.TIMESTAMP 并且参数Object x类型不是LocalDateTime	请参见 void setTimestamp(int parameterIndex, java.sql.Timestamp x, Calendar cal) 接口。	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.TIMESTAMP 并且参数Object x类型是LocalDateTime	TIMESTAMP	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.TIMESTAMP_WITH_TIMEZONE 并且参数 Object x 类型是 OffsetDateTime	TIMESTAMP PTZ	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.TIMESTAMP_WITH_TIMEZONE 并且参数 Object x 类型是 PGTimestamp	请参见 void setTimestamp(int parameterIndex, java.sql.Timestamp, Calendar) 接口。	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BIT 并且连接 参数 use_boolean=true	BOOL	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BIT 并且连接 参数 use_boolean=false（默认）	INT2	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BOOLEAN 并且连接 参数 use_boolean=true	BOOL	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BOOLEAN 并且连接 参数 use_boolean=false（默认）	INT2	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BINARY、 Types.VARBINARY或 Types.LONGVARIABLE	根据参数 Object x数据类型决定，请参见表5-139。	-
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.BLOB 并且参数 Object x 类型是 Blob	请参见void setBlob(int parameterIndex, Blob x)接口。	-

JDBC接口	兼容模式	目标参数名	目标参数JAVA变量类型	条件	发送到数据库的类型	备注
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.BLOB 并且参数Object x类型是InputStream	请参见 void setLong(int parameter Index, long x) 接口。	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.CLOB 并且参数Object x类型是Clob	请参见 void setClob(int parameter Index, Clob x) 接口。	-
	ORA/MYSQL	int targetSqlType	java.lang.Integer	参数Object x不为null 并且参数int targetSqlType=Types.DISTINCT	UNSPECIFIED	-

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
	ORA/ MY SQL	int targetSqlType	java.lang.Integer	参数 Object x 不为null 并且参数 int targetSqlType=Types.ROWID	ROWID	-
void setObject(int parameterIndex, Object x)	ORA/ MY SQL	-	-	-	-	根据参数 Object x 类型， 请参见 表5-139 寻找对应类型。
default void setObject(int parameterIndex, Object x, SQLType targetSqlType,int scaleOrLength)	ORA/ MY SQL	-	-	-	-	根据参数 SQLType targetSqlType 类型， 请参见 表5-140 寻找对应类型。
default void setObject(int parameterIndex, Object x, SQLType targetSqlType)	ORA/ MY SQL	-	-	-	-	根据参数 SQLType targetSqlType 类型， 请参见 表5-140 寻找对应类型。

JDBC接口	兼容模式	目标参数名	目标参数 JAVA变量 类型	条件	发送到数据库的类型	备注
void setObject(int parameterIndex, Object x, int targetSqlType, int scaleOrLength)	ORA/ MYSQL	-	-	-	-	根据参数int targetSqlType 类型， 请参见 表5-140 寻找对应类型。

表 5-138 setNull 接口 java 类型对应数据库类型

JAVA type id	兼容模式	条件	发送到数据库的类型
Types.VARCHAR	ORA/ MYSQL	连接参数 stringtype=unspecified	UNSPECIFIED
Types.VARCHAR	ORA/ MYSQL	连接参数stringtype不为 unspecified（默认）	VARCHAR
Types.LONGVARCHAR	ORA/ MYSQL	连接参数 stringtype=unspecified	UNSPECIFIED
Types.LONGVARCHAR	ORA/ MYSQL	连接参数stringtype不为 unspecified（默认）	VARCHAR
Types.DATE	ORA	连接参数oracleCompatible配置为true或者配置包含 setDateTime 并且数据库GUC参数 mapping_date_to_datea取值为on	DATEA
Types.DATE	ORA	连接参数oracleCompatible配置为true或者配置包含 setDateTime 并且数据库GUC参数 mapping_date_to_datea取值为off	TIMESTAMP
Types.DATE	ORA/ MYSQL	-	DATE
Types.SQLXML	ORA/ MYSQL	-	XML

JAVA type id	兼容模式	条件	发送到数据库的类型
Types.INTEGER	ORA/ MYSQL	-	INT4
Types.TINYINT	ORA/ MYSQL	-	INT1
Types.SMALLINT	ORA/ MYSQL	-	INT2
Types.BIGINT	ORA/ MYSQL	-	INT8
Types.REAL	ORA/ MYSQL	-	FLOAT4
Types.DOUBLE	ORA/ MYSQL	-	FLOAT8
Types.FLOAT	ORA/ MYSQL	-	FLOAT8
Types.DECIMAL	ORA/ MYSQL	-	NUMERIC
Types.NUMERIC	ORA/ MYSQL	-	NUMERIC
Types.CHAR	ORA/ MYSQL	-	BPCHAR
Types.TIME	ORA/ MYSQL	-	UNSPECIFIED
Types.TIMESTAMP	ORA/ MYSQL	-	UNSPECIFIED
Types.TIME_WITH _TIMEZONE	ORA/ MYSQL	-	UNSPECIFIED
Types.TIMESTAMP _WITH_TIMEZONE	ORA/ MYSQL	-	UNSPECIFIED
Types.BOOLEAN	ORA/ MYSQL	-	BOOL
Types.BIT	ORA/ MYSQL	-	BOOL
Types.BINARY	ORA/ MYSQL	-	BYTEA
Types.VARBINARY	ORA/ MYSQL	-	BYTEA

JAVA type id	兼容模式	条件	发送到数据库的类型
Types.LONGVARBINARY	ORA/MYSQL	-	BYTEA
Types.BLOB	ORA/MYSQL	-	BLOB
Types.CLOB	ORA/MYSQL	-	CLOB
Types.ARRAY	ORA/MYSQL	-	UNSPECIFIED
Types.DISTINCT	ORA/MYSQL	-	UNSPECIFIED
Types.STRUCT	ORA/MYSQL	-	UNSPECIFIED
Types.NULL	ORA/MYSQL	-	UNSPECIFIED
Types.OTHER	ORA/MYSQL	-	UNSPECIFIED

表 5-139 setObject 接口参数 object x 类型对应执行接口

类型	兼容模式	条件	发送到数据库的类型
null	ORA/MYSQL	-	请参见表5-137中 setNull(int parameterIndex, int sqlType) 接口。
UUID	ORA/MYSQL	-	UUID
SQLXML	ORA/MYSQL	-	请参见表5-137中 void setSQLXML(int parameterIndex, SQLXML xmlObject) 接口。
String	ORA/MYSQL	-	请参见表5-137中 void setString(int parameterIndex, String x) 接口。
Character	ORA/MYSQL	-	请参见表5-137中 void setString(int parameterIndex, String x) 接口。

类型	兼容模式	条件	发送到数据库的类型
RowId	ORA/MYSQL	-	请参见表5-137中void setRowId(int parameterIndex, RowId x)接口。
BigDecimal	ORA/MYSQL	-	请参见表5-137中void setBigDecimal(int parameterIndex, BigDecimal x)接口。
Short	ORA/MYSQL	-	请参见表5-137中void setShort(int parameterIndex, short x)接口。
Integer	ORA/MYSQL	-	请参见表5-137中void setInt(int parameterIndex, int x)接口。
Long	ORA/MYSQL	-	请参见表5-137中void setLong(int parameterIndex, long x)接口。
Float	ORA/MYSQL	-	请参见表5-137中void setFloat(int parameterIndex, float x)接口。
Double	ORA/MYSQL	-	请参见表5-137中void setDouble(int parameterIndex, double x)接口。
byte[]	ORA/MYSQL	-	请参见表5-137中void setBytes(int parameterIndex, byte x[])接口。
Byte	ORA/MYSQL	-	请参见表5-137中void setByte(int parameterIndex, byte x)接口。
java.sql.Date	ORA/MYSQL	-	请参见表5-137中void setDate(int parameterIndex, java.sql.Date x)接口。
Time	ORA/MYSQL	-	请参见表5-137中void setTime(int parameterIndex, java.sql.Time x)接口。

类型	兼容模式	条件	发送到数据库的类型
Timestamp	ORA/MYSQL	-	请参见表5-137中 void setTimestamp(int parameterIndex, java.sql.Timestamp x) 接口。
LocalDate	MYSQL	-	DATE
	ORA	连接参数 mapping_date_to_datea取值为on	DATEA
	ORA	连接参数 mapping_date_to_datea取值为off	TIMESTAMP
LocalTime	ORA/MYSQL	-	TIME
LocalDateTime	ORA/MYSQL	-	TIMESTAMP
OffsetDateTime	ORA/MYSQL	-	TIMESTAMP TZ
java.util.Date	ORA/MYSQL	-	请参见表5-137中 void setTimestamp(int parameterIndex, java.sql.Timestamp x) 接口。
Boolean	ORA/MYSQL	-	请参见表5-137中 void setBoolean(int parameterIndex, boolean x) 接口。
Blob	ORA/MYSQL	-	请参见表5-137中 void setBlob(int parameterIndex, Blob x) 接口。
Clob	ORA/MYSQL	-	请参见表5-137中 void setClob(int parameterIndex, Clob x) 接口。
Number	ORA/MYSQL	-	NUMERIC

表 5-140 setObject 接口参数 targetSqlType 对应执行接口

targetSqlType 参数值	兼容模式	条件	发送到数据库的类型
Types.SQLXML	ORA/MYSQL	-	请参见表5-137中void setSQLXML(int parameterIndex, SQLXML xmlObject)接口。
Types.INTEGER	ORA/MYSQL	-	请参见表5-137中void setInt(int parameterIndex, int x)接口。
Types.TINYINT	ORA/MYSQL	-	INT1
Types.SMALLINT	ORA/MYSQL	-	请参见表5-137中void setShort(int parameterIndex, short x) 接口。
Types.BIGINT	ORA/MYSQL	-	请参见表5-137中void setLong(int parameterIndex, long x)接 口。
Types.REAL	ORA/MYSQL	-	请参见表5-137中void setFloat(int parameterIndex, float x) 接口。
Types.FLOAT	ORA/MYSQL	连接参数 setFloat=true	请参见表5-137中void setFloat(int parameterIndex, float x) 接口。
	MYSQL	连接参数 setFloat=false(默 认) 并且连接参数 mCompatible不包 含pstatement	请参见表5-137中void setFloat(int parameterIndex, float x) 接口。
	ORA	连接参数 setFloat=false(默 认)	请参见表5-137中void setDouble(int parameterIndex, double x)接口

targetSqlType 参数值	兼容模式	条件	发送到数据库的类型
	MYSQL	连接参数 setFloat=false(默认) 并且连接参数 mCompatible为 true或者包含 pstatement	请参见表5-137中void setDouble(int parameterIndex, double x)接口
Types.DOUBLE	ORA/MYSQL	-	请参见表5-137中void setDouble(int parameterIndex, double x)接口
Types.DECIMAL	ORA/MYSQL	-	请参见表5-137中void setBigDecimal(int parameterIndex, BigDecimal x)接口。
Types.NUMERIC	ORA/MYSQL	-	请参见表5-137中void setBigDecimal(int parameterIndex, BigDecimal x)接口。
Types.CHAR	ORA/MYSQL	-	BPCHAR
Types.VARCHAR	ORA/MYSQL	-	VARCHAR
Types.LONG VARCHAR	ORA/MYSQL	参数Object x类型为 InputStream	TEXT
	ORA/MYSQL	参数Object x类型不 为InputStream	VARCHAR
Types.DATE	ORA/MYSQL	-	请参见表5-137中void setDate(int parameterIndex, java.sql.Date x)接口。
Types.TIME	ORA/MYSQL	-	请参见表5-137中void setTime(int parameterIndex, java.sql.Time x)接口。
Types.TIMESTAMP	ORA/MYSQL	-	请参见表5-137中void setTimestamp(int parameterIndex, java.sql.Timestamp x)接口。

targetSqlType参数值	兼容模式	条件	发送到数据库的类型
Types.TIMESTAMP_WITH_TIMEZONE	ORA/MYSQL	参数Object x类型为OffsetDateTime或者PGTimestamp	请参见表5-137中void setTimestamp(int parameterIndex, java.sql.Timestamp x)接口。
Types.BIT	ORA/MYSQL	连接参数mCompatible不配置为true并且配置不包含pstatement	请参见表5-137中void setBoolean(int parameterIndex, boolean x)接口。
	MYSQL	连接参数mCompatible配置为true或者配置包含pstatement	请参见表5-137中void setInt(int parameterIndex, int x)接口。
Types.BOOLEAN	ORA/MYSQL	-	请参见表5-137中void setBoolean(int parameterIndex, boolean x)接口。
Types.BINARY	ORA/MYSQL	-	请根据Object x数据类型，对照表5-139寻找对应发送到数据库的类型。
Types.VARBINARY	ORA/MYSQL	-	请根据Object x数据类型，对照表5-139寻找对应发送到数据库的类型。
Types.LONGVARBINARY	ORA/MYSQL	-	请根据Object x数据类型，对照表5-139寻找对应发送到数据库的类型。
Types.BLOB	ORA/MYSQL	参数Object x类型为Blob	请参见表5-137中void setBlob(int parameterIndex, Blob x)接口。
	ORA/MYSQL	参数Object x类型为InputStream	请参见表5-137中void setLong(int parameterIndex, long x)接口。
Types.CLOB	ORA/MYSQL	参数Object x类型为Clob	请参见表5-137中void setClob(int parameterIndex, Clob x)接口。
Types.ARRAY	ORA/MYSQL	参数Object x类型为Array	请参见表5-137中void setArray(int parameterIndex, Array x)接口。

targetSqlType参数值	兼容模式	条件	发送到数据库的类型
Types.DISTINCT	ORA/MYSQL	-	UNSPECIFIED
Types.ROWID	ORA/MYSQL	参数Object x类型为RowId	请参见表5-137中void setRowId(int parameterIndex, RowId x) 接口。
	ORA/MYSQL	参数Object x类型为String	ROWID
Types.OTHER	ORA/MYSQL	参数Object x类型为UUID 并且服务端参数SERVER_VERSION值大于8.3.0	UUID
	ORA/MYSQL	参数Object x类型不为PGObject或MAP	UNSPECIFIED

表 5-141 ResultSet 类 get 接口

方法（Method）	处理逻辑	备注
getString(int columnIndex)	结合列OID、数据库类型及连接参数，获取对应的JDBC类型并进行转换。若需转为String类型，会对特定数据库的特殊数据（如十六进制数据）进行针对性解析和处理。	-
getString(String columnLabel)		-
getBoolean(int columnIndex)	若列OID为BOOL，数据长度为1且值为ASCII码值116（小写字母t），则返回true，否则返回false。其他场景下，基于列OID及设置的连接参数将byte数据转换为对应的字符串后，再解析该字符串是否符合boolean语义（如 "true"、"t"、"0"、"1" 等）并返回对应结果。	-
getBoolean(String columnLabel)		-
getBytes(int columnIndex)	基于列OID及设置的连接参数获取对应的JDBC类型，依据该类型将数据先转换为String或long类型，最终强制转换为byte类型返回。若为高精度数值对象，会截断其小数部分。	-
getBytes(String columnLabel)		-

方法（Method）	处理逻辑	备注
getShort(int columnIndex)	基于列OID及设置的连接参数，将数据先转换为String（转换时去除\$、（、）等特殊字符）或long类型，最终强制转换为short类型返回。若为高精度数值对象，会截断其小数部分。	-
getShort(String columnLabel)		-
getInt(int columnIndex)	若列OID为INT4，则直接转换为int类型返回。其他场景下，将数据先转换为String（去除特殊字符）或long类型，最终强制转换为int类型返回。若为高精度数值，截断小数部分。	-
getInt(String columnLabel)		-
getLong(int columnIndex)	若列OID为INT8，则直接转换为long类型返回。其他场景下，将数据先转换为String（去除特殊字符），再解析并转换为long类型返回。若为高精度数值，截断小数部分。	-
getLong(String columnLabel)		-
getFloat(int columnIndex)	若列OID为FLOAT4，则直接转换为float类型返回。其他场景下，将数据先转换为String（去除特殊字符）或double类型，最终强制转换为float类型返回。	-
getFloat(String columnLabel)		-
getDouble(int columnIndex)	若列OID为FLOAT8，则直接转换为double类型返回。其他场景下，将数据先转换为String（去除特殊字符），最终解析并转换为double类型返回。	-
getDouble(String columnLabel)		-
getBigDecimal(int columnIndex, int scale)	基于列OID及设置的连接参数获取对应的JDBC类型，依据该类型将数据先转换为String、Number或long类型，最终构造并返回BigDecimal对象。	四个接口底层调用同一个函数getBigDecimal(int columnIndex, int scale)，其中getBigDecimal(int columnIndex)和getBigDecimal(String columnLabel)调用时scale为-1。
getBigDecimal(String columnLabel, int scale)		
getBigDecimal(int columnIndex)		
getBigDecimal(String columnLabel)		

方法（ Method ）	处理逻辑	备注
getBytes(int columnIndex)	基于列OID及设置的连接参数，部分场景直接保留并返回byte数组；其他场景先将数据转为JDBC类型对应的String，再提取为byte数组。最终会根据Statement.setMaxFieldSize的限制，截取数组长度后返回。	-
getBytes(String columnLabel)		-
getDate(int columnIndex)	基于列OID及设置的连接参数，部分场景直接转换为Timestamp或天数，再结合时区换算为Date返回；其他场景先转换为String，再结合时区解析构造为Date对象返回。	四个接口底层调用同一个函数 getDate(int columnIndex, java.util.Calendar cal)，其中 getDate(int columnIndex) 和 getDate(String columnLabel) 调用时cal为 null。
getDate(String columnLabel)		
getDate(int columnIndex, Calendar cal)		
getDate(String columnLabel, Calendar cal)		
getTime(int columnIndex)	基于列OID及设置的连接参数，部分场景直接转换对应底层数据并结合时区计算Time值；其他场景先转换为String，再结合时区解析构造为Time对象返回。	四个接口底层调用同一个函数 getTime(int columnIndex, java.util.Calendar cal)，其中 getTime(int columnIndex) 和 getTime(String columnLabel) 调用时cal为 null。
getTime(String columnLabel)		
getTime(int columnIndex, Calendar cal)		
getTime(String columnLabel, Calendar cal)		
getTimestamp(int columnIndex)	基于列OID及设置的连接参数，部分场景直接转换对应底层数据并结合时区计算Timestamp值；其他场景先转换为String，再结合时区解析构造为Timestamp对象返回。	四个接口底层调用同一个函数 getTimestamp(int columnIndex, java.util.Calendar cal)，其中 getTimestamp(int columnIndex) 和 getTimestamp(String columnLabel) 调用时cal为 null。
getTimestamp(String columnLabel)		
getTimestamp(int columnIndex, Calendar cal)		
getTimestamp(String columnLabel, Calendar cal)		

方法（Method）	处理逻辑	备注
getAsciiStream(int columnIndex)	基于列OID及设置的连接参数，将数据转为String后，按ASCII编码将其转换为字节数组，并封装为ByteArrayInputStream对象作为InputStream返回。	-
getAsciiStream(String columnLabel)		-
getBinaryStream(int columnIndex)	逻辑参考getBytes。获取符合Statement.setMaxFieldSize长度限制的byte数组后，将其封装为ByteArrayInputStream对象作为InputStream返回。	-
getBinaryStream(String columnLabel)		-
getObject(int columnIndex)	基于列OID及设置的连接参数，获取对应的JDBC类型并进行转换返回。在涉及String类型转换的过程中，会对部分特殊数据进行针对性处理。	-
getObject(String columnLabel)		-
getCharacterStream(int columnIndex)	内部调用getString获取字符串，通过toCharArray()将其转换为字符数组，并以此数组构造CharArrayReader作为Reader返回。	-
getCharacterStream(String columnLabel)		-
getRow()	获取当前结果集游标所在的行号。行号从1开始递增（第一行为1，第二行为2，以此类推）。	-
getObject(int columnIndex, java.util.Map<String, Class<?>> map)	JDBC驱动暂不支持自定义类型映射。当传入的map为空时，默认降级调用getObject(int columnIndex)接口返回数据。	-
getObject(String columnLabel, java.util.Map<String, Class<?>> map)		-
getRef(int columnIndex)	暂不支持。	-
getRef(String columnLabel)		-
getBlob(int columnIndex)	基于列OID及设置的连接参数获取处理后的byte数组，以此底层数据构造并返回基于java.sql.Blob接口实现的实例对象。	-
getBlob(String columnLabel)		-
getClob(int columnIndex)	基于列OID及设置的连接参数获取处理后的byte数组，以此底层数据构造并返回基于java.sql.Clob接口实现的实例对象。	-
getClob(String columnLabel)		-

方法（Method）	处理逻辑	备注
getArray(int columnIndex)	基于列OID及设置的连接参数获取处理后的byte数组，以此底层数据构造并返回基于java.sql.Array接口实现的实例对象。	-
getArray(String columnLabel)		-
getURL(int columnIndex)	内部调用getString获取字符串，并利用该字符串构造java.net.URL对象返回。	-
getURL(String columnLabel)		-
getRowId(int columnIndex)	内部调用getBytes获取byte数组，以此底层数据构造并返回基于java.sql.RowId接口实现的实例对象。	-
getRowId(String columnLabel)		-
getHoldability()	暂不支持。	-
getNClob(int columnIndex)	暂不支持。	-
getNClob(String columnLabel)		-
getSQLXML(int columnIndex)	内部调用getString获取字符串，并利用该字符串构造并返回基于java.sql.SQLXML接口实现的实例对象。	-
getSQLXML(String columnLabel)		-
getNString(int columnIndex)	处理逻辑同getString。	-
getNString(String columnLabel)		-
getNCharacterStream(int columnIndex)	暂不支持。	-
getNCharacterStream(String columnLabel)		-
getObject(int columnIndex, Class<T> type)	根据传入的type参数（Class类型），决定数据库返回byte数据的转换方法，并按指定的类型返回（部分特定类型包含特殊处理逻辑）。	-
getObject(String columnLabel, Class<T> type)		-
getCursorName()	返回null。	-
getMetaData()	返回包含当前结果集所有列的元数据描述。	-

方法（Method）	处理逻辑	备注
getFetchDirection()	返回当前ResultSet的数据抓取方向（fetch direction）。	-
getFetchSize()	返回当前ResultSet的抓取大小（fetch size），即JDBC驱动每次从数据库批量获取的行数提示值。	-
getType()	返回当前ResultSet的类型（游标类型）。	-
getConcurrency()	返回当前ResultSet的并发模式。	-
getStatement()	返回当前ResultSet的Statement对象。	-

说明

GaussDB JDBC调用ResultSet类中的get接口时，具体的数据解码与类型映射逻辑在部分场景与Oracle存在差异。

这种底层驱动的切换，通常会在以下几个维度引发Java应用层面上数据表现的差异：

- 类型映射策略的不同：不同驱动对于数据库原生类型（如通用的NUMBER）的理解不同。部分驱动倾向于强类型的严格映射（如强制映射为BigDecimal），而定制驱动可能为了兼容特定的历史系统，将其泛化映射为Double或String。
- 精度与舍入边界的把控：在处理带有高精度小数或极端大数时，不同驱动的“容忍度”不同。有些驱动在遇到精度溢出时会静默截断或补零，而严谨的驱动则可能直接抛出边界异常。
- 环境与上下文的依赖：对于时间戳等敏感数据，不同驱动在解析纯数字时，引入本地服务器时区偏移补偿的算法和基准可能存在细微出入。
- 空值与脏数据的兜底策略：当数据库底层为NULL或非标数据格式时，不同驱动的正常阻断机制和默认值填充策略（Fallback逻辑）一般会相差很大。
- 隐式类型转换的宽容度与规则：当调用的ResultSet类get接口与数据库底层列类型不严格匹配时（例如对数值/日期列调用getString()，或对字符串列调用getInt()），不同驱动的隐式转换机制存在差异。部分驱动极其宽容，会在底层自动完成跨类型的强制转换；而更严格的驱动可能会直接抛出类型不匹配异常，或在将其隐式转换为字符串时，采用截然不同的默认序列化格式（如时间的默认输出样式不同）。

5.10.1.3 日志管理

GaussDB JDBC驱动程序支持使用日志记录来帮助解决在应用程序中使用GaussDB JDBC驱动程序时遇到的问题。GaussDB JDBC支持如下两种日志管理方式：

- 对接应用程序使用的SLF4J日志框架。
- 对接应用程序使用的JdkLogger日志框架。

SLF4J和JdkLogger是业界Java应用程序日志管理的主流框架，描述应用程序如何使用这些框架超出了本文范围，用户请参见对应的官方文档（SLF4J：<https://www.slf4j.org/manual.html>，JdkLogger：<https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html>）。

方式一：对接应用程序的 SLF4J 日志框架

在建立连接时，url需要配置logger=Slf4JLogger。此方式支持日志管控，SLF4J可通过文件中的相关配置实现强大的日志管控功能，建议使用此方式进行日志管理。

可采用Log4j或Log4j2来实现SLF4J。当采用Log4j实现SLF4J，需要添加如下jar包（*区分版本）：log4j-*.jar、slf4j-api-*.jar、slf4j-log4j-*.jar。同时指定log4j.properties配置文件的路径（相对路径或者绝对路径均可）。

若采用Log4j2实现SLF4J，需要添加如下jar包（*区分版本）：log4j-api-*.jar、log4j-core-*.jar、log4j-slf4j18-impl-*.jar、slf4j-api-*-alpha1.jar。同时指定log4j2.xml配置文件的路径（相对路径或者绝对路径均可）。

须知

此方式依赖SLF4J的通用API接口，如org.slf4j.LoggerFactory.getLogger(String name)、org.slf4j.Logger.debug(String var1)、org.slf4j.Logger.info(String var1)、org.slf4j.Logger.warn(String warn)、org.slf4j.Logger.warn(String warn)等，若以上接口发生变更，日志将无法打印。

非透明多写特性下的示例：

```
//调用时需要指定log4j.properties或者log4j2.xml配置文件路径。

//使用log4j.properties文件来配置日志记录器的属性，例如日志级别、输出目标等。通过调用
PropertyConfigurator.configure方法并传递log4j.properties文件的路径，将这些属性加载到应用程序中。
//PropertyConfigurator.configure("log4j.properties");

//创建Log4j2配置源，指定配置文件log4j2.xml的路径。并初始化日志系统。
//ConfigurationSource source = new ConfigurationSource(new FileInputStream("log4j2.xml"));
//Configurator.initialize(null, source);

public static Connection GetConnection(String username, String passwd){

    String sourceURL = "jdbc:gaussdb://$ip.$port/database?logger=Slf4JLogger";
    Connection conn = null;

    try{
        //创建连接。
        conn = DriverManager.getConnection(sourceURL,username,passwd);
        System.out.println("Connection succeed!");
    }catch (Exception e){
        e.printStackTrace();
        return null;
    }
    return conn;
}
```

log4j.properties配置文件：

```
log4j.logger.com.huawei.gaussdb.jdbc=ALL, log_gsjdbc

# 默认文件输出配置。
log4j.appender.log_gsjdbc=org.apache.log4j.RollingFileAppender
log4j.appender.log_gsjdbc.Append=true
log4j.appender.log_gsjdbc.File=gsjdbc.log
log4j.appender.log_gsjdbc.Threshold=TRACE
log4j.appender.log_gsjdbc.MaxFileSize=10MB
log4j.appender.log_gsjdbc.MaxBackupIndex=5
log4j.appender.log_gsjdbc.layout=org.apache.log4j.PatternLayout
log4j.appender.log_gsjdbc.layout.ConversionPattern=%d %p %t %c - %m%n
log4j.appender.log_gsjdbc.File.Encoding = UTF-8
```

log4j2.xml配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="OFF">
  <appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %p %t %c - %m%n"/>
    </Console>
    <File name="FileTest" fileName="test.log">
      <PatternLayout pattern="%d %p %t %c - %m%n"/>
    </File>
    <!--JDBC Driver日志文件输出配置，支持日志回卷，设定日志大小超过10MB时，创建新的文件，新文件的命名格式为：年-月-日-文件编号-->
    <RollingFile name="RollingFileJdbc" fileName="gsjdbc.log" filePattern="%d{yyyy-MM-dd}-%i.log">
      <PatternLayout pattern="%d %p %t %c - %m%n"/>
      <Policies>
        <SizeBasedTriggeringPolicy size="10 MB"/>
      </Policies>
    </RollingFile>
  </appenders>
  <loggers>
    <root level="all">
      <appender-ref ref="Console"/>
      <appender-ref ref="FileTest"/>
    </root>
    <!--指定JDBC Driver日志，级别为all，可查看所有日志，输出到gsjdbc.log文件中-->
    <logger name="com.huawei.gaussdb.jdbc" level="all" additivity="false">
      <appender-ref ref="RollingFileJdbc"/>
    </logger>
  </loggers>
</configuration>
```

须知

- 在配置文件中，可以指定JDBC与上层业务的配置路径，将JDBC日志与业务日志分别存储到不同文件。
- 同一个项目中配置多个GaussDB数据源时，如果存在部分业务配置了 logger=Slf4JLogger，部分业务未配置，会互相影响日志配置。建议这种情况下连接串统一配置。

方式二：对接应用程序使用的 JdkLogger 日志框架

两种方式：采用logging.properties配置文件、直接在示例代码中进行配置。

- 默认的Java日志记录框架将其配置存储在名为logging.properties的文件中。Java会在Java安装目录的文件夹中安装全局配置文件。logging.properties文件也可以创建并与单个项目一起存储。

logging.properties配置文件：

```
# 指定处理程序为文件。
handlers= java.util.logging.FileHandler

# 指定默认全局日志级别。
.level= ALL

# 指定日志输出管控标准。
java.util.logging.FileHandler.level=ALL
java.util.logging.FileHandler.pattern = gsjdbc.log
java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 30
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.append=false
```

- 直接在示例代码中进行配置：

```
System.setProperty("java.util.logging.FileHandler.pattern", "jdbc.log");
FileHandler fileHandler = new FileHandler(System.getProperty("java.util.logging.FileHandler.pattern"));
Formatter formatter = new SimpleFormatter();
fileHandler.setFormatter(formatter);
Logger logger = Logger.getLogger("com.huawei.gaussdb.jdbc");
logger.addHandler(fileHandler);
logger.setLevel(Level.ALL);
logger.setUseParentHandlers(false);
```

链路跟踪功能

GaussDB JDBC驱动程序提供了应用到数据库的链路跟踪功能，用于将数据库端离散的SQL和应用程序的请求关联起来。该功能需要应用开发者实现com.huawei.gaussdb.jdbc.log.Tracer接口类，并在url中指定接口实现类的全类名。

url示例：

```
String url = "jdbc:gaussdb://$ip:$port/database?traceInterfaceClass=xxx.xxx.xxx.gaussdbTracerImpl";
```

com.huawei.gaussdb.jdbc.log.Tracer接口类定义如下：

```
public interface Tracer {
    // Retrieves the value of traceId.
    String getTraceId();
}
```

com.huawei.gaussdb.jdbc.log.Tracer接口实现类示例：

```
import com.huawei.gaussdb.jdbc.log.Tracer;

public class gaussdbTracerImpl implements Tracer {
    private static MDC mdc = new MDC();

    private final String TRACE_ID_KEY = "traceId";

    public void set(String traceId) {
        mdc.put(TRACE_ID_KEY, traceId);
    }

    public void reset() {
        mdc.clear();
    }

    @Override
    public String getTraceId() {
        return mdc.get(TRACE_ID_KEY);
    }
}
```

上下文映射示例，用于存放不同请求生成的traceId。

```
import java.util.HashMap;

public class MDC {
    static final private ThreadLocal<HashMap<String, String>> threadLocal = new ThreadLocal<>();

    public void put(String key, String val) {
        if (key == null || val == null) {
            throw new IllegalArgumentException("key or val cannot be null");
        } else {
            if (threadLocal.get() == null) {
                threadLocal.set(new HashMap<>());
            }
            threadLocal.get().put(key, val);
        }
    }

    public String get(String key) {
```

```
        if (key == null) {
            throw new IllegalArgumentException("key cannot be null");
        } else if (threadLocal.get() == null) {
            return null;
        } else {
            return threadLocal.get().get(key);
        }
    }

    public void clear() {
        if (threadLocal.get() == null) {
            return;
        } else {
            threadLocal.get().clear();
        }
    }
}
```

业务使用traceId示例，前置条件需要先建表test_trace_id (id int,name varchar(20))。

```
String traceId = UUID.randomUUID().toString().replaceAll("-", "");
gaussdbTraceImpl gaussdbTrace = new gaussdbTraceImpl();
gaussdbTrace.set(traceId);
Connection con = DriverManager.getConnection(url, user, password);
pstmt = con.prepareStatement("SELECT * FROM test_trace_id WHERE id = ?");
pstmt.setInt(1, 1);
pstmt.execute();
pstmt = con.prepareStatement("INSERT INTO test_trace_id VALUES(?,?)");
pstmt.setInt(1, 2);
pstmt.setString(2, "test");
pstmt.execute();
gaussdbTrace.reset();
```

说明

- 使用链路跟踪功能时，应用层链路功能由业务保证。
- 应用必须向JDBC暴露获取traceId的接口，并将该接口实现类配置到JDBC连接字符串中。
- 同一请求的不同SQL使用的traceId须相同。
- 应用传入的traceId不能超过32字节，否则多余字节将被截断。

5.10.1.4 常见问题处理

5.10.1.4.1 batchMode 设置错误

问题现象

设置参数batchMode=on且rewriteBatchedInserts=true，使用JDBC批量插入数据后，提示绑定参数数量与语句需要的参数数量不一致：

```
bind message supplies * parameters, but prepared statement "" requires *
```

示例：

```
// conn是已经创建的Connection对象，创建该connection的url连接串包含
batchMode=on&rewriteBatchedInserts=true
// 批量绑定参数后执行，绑定参数数量会与改写后的INSERT语句的栏位数不匹配，提示异常。
// java.sql.BatchUpdateException: bind message supplies 3 parameters, but prepared statement "" requires 6

PreparedStatement stmt = conn.prepareStatement("INSERT INTO test_tbl VALUES (?, ?, ?)");

stmt.setInt(1, 1);
stmt.setString(2, "aaa");
stmt.setString(3, "bbbb");
stmt.addBatch();
```

```
stmt.setInt(1, 1);  
stmt.setString(2, "aaa");  
stmt.setString(3, "bbbb");  
stmt.addBatch();  
  
stmt.setInt(1, 1);  
stmt.setString(2, "aaa");  
stmt.setString(3, "bbbb");  
stmt.addBatch();  
  
stmt.executeBatch();
```

原因分析

将参数rewriteBatchedInserts设置为true时，批量语句会将多条SQL语句合并为一条，导致语句中预留参数栏位数发生变化，如果batchMode=on，会按照合并前的SQL绑定参数，导致绑定参数数量与语句需要的参数数量不一致。

处理方法

rewriteBatchedInserts设置为true时，将batchMode设置为off。

5.10.1.4.2 Hibernate 框架插入数据开启校验时报错

问题现象

客户从ORA模式数据库迁移到GaussDB数据库，表结构使用DRS工具进行迁移，迁移后客户原业务代码不可用，Hibernate框架校验表结构报错。

```
Schema-validation: wrong column type encountered in column [execute_time] in table [abnormal_event];  
found [int8 (Types#BIGINT)], but expecting [int4 (Types#INTEGER)]
```

示例：

- 原ORA模式数据库中Student表结构为如下格式：
id number(15,0)
sid number(15,0)
age number(10,0)
- 使用DRS工具迁移到GaussDB数据库后，Student表结构为如下格式：
id bigint
sid bigint
age Integer
- 当实际数据库中Student实体类信息如下时：
Long id
Long sid
Long age --Hibernate框架校验表结构报错。

hibernate.cfg.xml配置文件：

```
<hibernate-configuration>  
  <session-factory>  
    <!--GaussDB连接信息-->  
    <property name="connection.driver_class">com.huawei.gaussdb.jdbc.Driver</property>  
    <property name="connection.url">jdbc:gaussdb://x.x.x.x:xx/postgres?currentSchema=public</property>  
    <property name="connection.username">xxxxxx</property>  
    <property name="connection.password">xxxxxx</property>  
  
    <!--以下为可选配置-->  
  
    <!--是否支持方言-->  
    <!--在PG数据库兼容模式下，必须有如下配置-->  
    <property name="dialect">org.hibernate.dialect.PostgreSQL92Dialect</property>  
    <!--在ORA数据库兼容模式下，推荐如下配置-->
```

```
<!--      <property name="dialect">org.hibernate.dialect.OracleDialect</property-->

      <property name="met"></property>

      <!--执行CURD时是否打印SQL语句 -->
      <property name="show_sql">true</property>
      <!--自动建表-->
<!--      <property name="hbm2ddl.auto">create</property-->
      <!--插入数据时开启校验-->
      <property name="hbm2ddl.auto">validate</property>
      <!--关闭校验-->
<!--      <property name="hbm2ddl.auto">none</property-->

      <!-- 资源注册（实体类映射文件）-->
      <mapping resource="student.xml"/>
    </session-factory>
  </hibernate-configuration>
```

插入数据的主函数：

```
// 创建要测试的对象。
Student student = new Student();
student.setId(20L);
student.setAge(222L);
student.setSid(222L);

// 开启事务，基于session得到对象。
Configuration conf = new Configuration().configure();
SessionFactory sessionFactory = conf.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

// 通过session保存数据。
session.save(student);

// 提交事务。
transaction.commit();

// 操作完毕，关闭session连接对象。
session.close();
```

原因分析

- ORA模式数据库能够插入成功并通过校验：Hibernate框架在校验数据时，通过判断插入数据类型java.lang.Long的TypeCode，与表结构的number类型的TypeCode进行对比，结果不一致，随后会把插入数据的long类型的sqlType与表结构的Type作比较，将long类型转成number(19,0)（对应关系由org.hibernate.dialect.OracleDialect进行维护），因为表结构的type是number类型，number(19,0)以number开头，所以校验可以通过。使用ORA模式数据库时，业务代码涉及整数类型的数据，可以使用java.lang.Long类型插入。
- GaussDB数据库校验不同，关闭校验后数据插入成功：Hibernate框架在插入数据做校验时，业务代码是java.lang.Long类型，和表数据类型bigint的TypeCode是对应的，因此不报错。但是表数据类型是Integer时无法对应，首先判断业务代码java.lang.Long类型的TypeCode，与表数据Integer的TypeCode比较，结果不一致，再根据org.hibernate.dialect.PostgreSQLDialect对应关系，会将long类型转换为int8，将表数据类型Integer转换为int4，无法对应，所以校验失败。使用GaussDB数据库时，业务代码涉及整数类型的数据，必须使用表类型对应的Java整型。

处理方法

可使用以下方式进行问题处理：

- 关闭校验功能。
- 客户修改业务代码，针对不同的表数据类型，采用对应的Java整型。

5.10.1.4.3 使用 SSL 方式建连报错或阻塞

问题现象

JDBC使用SSL方式建立连接时，会在客户端获取强随机数，建立连接过程中可能出现以下现象：

场景1：如果连接串中配置了loginTimeout，可能会报Connection attempt timed out。

场景2：如果连接串中不配置loginTimeout，可能会建连阻塞，通过JDK命令jstack <pid>查询到的阻塞栈如下图所示。

```
"Thread-0" #18 prio=5 os_prio=0 tid=0x00007f2ad0385000 nid=0x5429 runnable [0x00007f2aa069b000]
  java.lang.Thread.State: RUNNABLE
    at java.io.FileInputStream.readBytes(Native Method)
    at java.io.FileInputStream.read(FileInputStream.java:255)
    at sun.security.provider.NativePRNG$RandomIO.readFully(NativePRNG.java:424)
    at sun.security.provider.NativePRNG$RandomIO.ensureBufferValid(NativePRNG.java:526)
    at sun.security.provider.NativePRNG$RandomIO.implNextBytes(NativePRNG.java:545)
    - locked <0x000000067273a950> (a java.lang.Object)
    at sun.security.provider.NativePRNG$RandomIO.access$400(NativePRNG.java:331)
    at sun.security.provider.NativePRNG$Blocking.engineNextBytes(NativePRNG.java:268)
    at java.security.SecureRandom.nextBytes(SecureRandom.java:468)
    at java.security.SecureRandom.next(SecureRandom.java:491)
    at java.util.Random.nextInt(Random.java:329)
    at sun.security.ssl.SSLContextImpl.engineInit(SSLContextImpl.java:106)
    at javax.net.ssl.SSLContext.init(SSLContext.java:202)
    at org.postgresql.ssl.LibPQFactory.<init>(LibPQFactory.java:175)
    at org.postgresql.core.SocketFactoryFactory.getSslSocketFactory(SocketFactoryFactory.java:62)
    at org.postgresql.ssl.MakeSSL.convert(MakeSSL.java:33)
    at org.postgresql.core.v3.ConnectionFactoryImpl.enableSSL(ConnectionFactoryImpl.java:723)
    at org.postgresql.core.v3.ConnectionFactoryImpl.tryConnect(ConnectionFactoryImpl.java:203)
    at org.postgresql.core.v3.ConnectionFactoryImpl.openConnectionImpl(ConnectionFactoryImpl.java:330)
    at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.java:58)
    at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:357)
```

原因分析

客户端环境随机数产生的速度太慢，无法满足产品要求，熵源不足，导致服务启动失败。当前已知在一些Linux环境中存在此问题。

处理方法

方法1：启动客户端环境中的haveged服务，增加系统熵池熵值以提高读取随机数的速度。启动命令为：

```
systemctl start haveged
```

方法2：调整客户端jdk配置。

打开\$JAVA_PATH/jre/lib/security/java.security文件，修改以下两个配置项：

```
securerandom.source=file:/dev/.urandom
securerandom.strongAlgorithms=NativePRNGNonBlocking:SUN
```

须知

方法2的本质是在获取强随机数时，使用伪随机数代替，减少需要消耗的熵值。会影响客户端所有使用该jdk的应用，在获取强随机数时会使用伪随机数代替。

5.10.2 ODBC

5.10.2.1 配置参数说明

本章节对odbc.ini文件配置参数进行详细说明，参见[表5-142](#)：

表 5-142 odbc.ini 文件配置参数

参数	描述	示例
[DSN]	数据源的名称。	[gaussdb]
Driver	驱动名，对应odbcinst.ini中的DriverName。	Driver=GaussMPP
Servename	服务器的IP地址。可配置多个IP地址。支持IPv4和IPv6。	Servename=127.0.0.1
Database	要连接的数据库的名称。	Database=db1
Username	数据库用户名称。	Username=omm
Password	数据库用户密码。 说明 ODBC驱动本身已经对内存密码进行过清理，以保证用户密码在连接后不会在内存中保留。 但是如果配置了此参数，由于UnixODBC对数据源文件等进行缓存，可能导致密码长期保留在内存中。 推荐在应用程序连接时，将密码传递给相应API，而非写在数据源配置文件中。同时连接成功后，应当及时清理保存密码的内存段。 须知 配置文件中填写密码时，需要遵循http规则： 1. 字符应当采用URL编码规范，如"!"应写作"%21"，"% "应写作"%25"，因此应当特别注意字符。 2. "+"会被替换为空格" "。	Password=***** *
Port	服务器的端口号。当开启负载均衡时，可配置多个端口号，且需与配置的多IP一一对应。如果开启负载均衡配置多个IP时，仍只配置一个端口号，则默认所有IP共用同一个端口号，即为配置的端口号。	Port=8000
Sslmode	是否启用SSL连接。 说明 Sslmode的可用选项及其说明，请参见表5-143。	Sslmode=allow

参数	描述	示例
Debug	控制调试模式的开启及日志输出级别。 取值范围：0 ~ INT_MAX • 设置为0时表示不启用调试模式。 • 设置为大于0的值时将启用调试日志，会打印gsqLODBC驱动的mylog。日志生成目录为/tmp/，文件名为mylog_xxx.log。 • 设置为1时表示仅输出FATAL级别日志。 • 设置为2时表示输出FATAL和ERROR级别日志。 • 设置为3时表示输出FATAL、ERROR和WARN级别日志。 • 设置为4时表示输出FATAL、ERROR、WARN和INFO级别日志。 • 设置为≥5时表示输出所有级别日志，增加输出TRACE级别日志。 默认值为0。 说明 1. FATAL：报告致命错误。 2. ERROR：报告异常错误。 3. WARN：报告警告信息。 4. INFO：记录程序运行中的正常关键信息。 5. TRACE：提供开发人员使用的调试信息。	Debug=5
CommLog	控制后端通信日志的开启及日志输出级别。 取值范围：0 ~ INT_MAX • 设置为0时表示不启用。 • 设置为大于0的值时将启用后端日志，会打印gsqLODBC驱动的qllog。日志生成目录为/tmp/，文件名为gsqldb_xxx.log。 • 设置为1时表示仅输出ERROR级别日志。 • 设置为2时表示输出ERROR和INFO级别日志。 • 设置为≥3时表示输出ERROR、INFO和TRACE级别日志。 默认值为0。 说明 1. ERROR：报告异常错误。 2. INFO：记录程序运行中的正常关键信息。 3. TRACE：提供开发人员使用的调试信息。	CommLog=1

参数	描述	示例
UseServerSidePrepare	是否开启数据库端扩展查询协议。 取值范围：0，1 - 取值为0表示不开启。 - 取值为1表示开启。 默认值为1。	UseServerSidePrepare=1
UseBatchProtocol	是否开启批量查询协议（开启后可提升DML操作性能）。 取值范围：0，1 - 取值为0时，不使用批量查询协议（主要用于与早期数据库版本通信兼容）。 - 取值为1，且GUC参数support_batch_bind设置为on时，将打开批量查询协议。 默认值为1。	UseBatchProtocol=1
ForExtensionConnector	此开关控制着savepoint是否发送，savepoint相关问题可以注意此开关。 取值范围：0，1 - 取值为0时表示发送savepoint。 - 取值为1时表示不发送savepoint。 默认值为1。	ForExtensionConnector=1
ConnectionExtraInfo	GUC参数connection_info中显示驱动部署路径和进程属主用户的开关。 取值范围：0，1 - 取值为0时表示不打开此开关。 - 取值为1时表示打开此开关。 默认值为0。 说明 当设置为1时，ODBC驱动会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在GUC参数connection_info里，同时可以在《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG_STAT_ACTIVITY”和“PGXC_STAT_ACTIVITY”中查询到。	ConnectionExtraInfo=1
BoolsAsChar	是否将布尔值作为字符处理。 取值范围：0，1 - 取值为0时表示Bools值将会映射为SQL_BIT。 - 取值为1时表示Bools值将会映射为SQL_CHAR。 默认值为1。	BoolsAsChar = 1

参数	描述	示例
RowVersioning	是否在更新一行数据时，允许应用检测数据有没有被其他用户进行修改。 取值范围：0，1 - 取值为0时表示不允许应用检测。 - 取值为1时表示允许应用检测。 默认值为0。	RowVersioning=1
ShowSystemTables	指定是否将默认系统表视为普通SQL表。 取值范围：0，1 - 取值为0时驱动不会将默认系统表视为普通SQL表。 - 取值为1时驱动将默认系统表视为普通SQL表。 默认值为0。	ShowSystemTables=1
AutoBalance	用于开启ODBC连接的负载均衡。 取值范围：0，1 - 取值为0时表示不开启负载均衡。 - 取值为1时开启负载均衡。 默认值为0。 说明 - 当连接的数据库版本低于505.2.0.SPC0200时，容灾集群不支持负载均衡。 - 仅支持单个[DSN]配置开启负载均衡。	AutoBalance=1
RefreshCNListTime	开启负载均衡时可配置该参数，该参数用于设置刷新CN列表的时间。整数型，单位为秒。 取值范围：0 ~ INT_MAX - 取值为0时表示不开启该参数。 - 取值大于0时表示该设定值为刷新CN列表的时间。 默认值为10。	RefreshCNListTime=5
Priority	开启负载均衡时可配置该参数。 取值范围：0，1 - 取值为0时表示不开启该参数。 - 取值为1时开启该参数。 默认值为0。 说明 当Priority开启时，应用程序发起的所有连接优先发送到配置文件中配置的CN上，当配置的CN全部不可用时，连接才会发送到剩余的CN上。	Priority=1

参数	描述	示例
UsingEip	开启负载均衡时可配置该参数。 取值范围：0，1 - 取值为0时表示不开启该参数。 - 取值为1时开启该参数。 默认值为0。 说明 此值用于控制是否使用业务IP（对应pgxc_node表中node_host1和node_port1列）实现负载均衡。当UsingEip开启时，表示使用业务IP实现负载均衡；关闭表示使用数据IP（对应pgxc_node表中node_host和node_port列）实现负载均衡。	UsingEip=1
MaxCacheQueries	控制每个连接缓存的预编译语句个数。 取值范围：0 ~ 4096 默认值为0。 说明 如果设置为0，则不开启客户端预编译语句缓存池。设置为大于4096的值会限制为4096。如果执行过的语句个数超过MaxCacheQueries设置的上限，则淘汰最近最少使用的语句。	MaxCacheQueries=128
MaxCacheSizeMiB	控制每个连接缓存的预编译语句总大小，在MaxCacheQueries大于0时生效。 取值范围：0 ~ 4096 默认值为1。 说明 如果缓存的语句总长度大于MaxCacheSizeMiB则淘汰最近最少使用的语句。单位为MB，设置为大于4096的值会限制为4096。	MaxCacheSizeMiB=10
TcpUserTimeout	在支持TCP_USER_TIMEOUT套接字选项的操作系统上，指定传输的数据在TCP连接被强制关闭之前可以保持未确认状态的最大时长。 取值范围：0 ~ INT_MAX 默认值为0。 说明 0表示使用系统缺省。通过Unix域套接字做的连接忽略这个参数。单位为毫秒。	TcpUserTimeout=5000
StandbyRead	用于指定分布式是否开启备机读模式。 取值范围：0，1 - 取值为0时表示禁用备机读模式。 - 取值为1时表示开启备机读模式。 默认值为0。	StandbyRead=1
Pqopt	用于设置libpq连接参数，参数之间用空格间隔。libpq参数请参见连接参数说明。	Pqopt=keepalives=0

参数	描述	示例
KeepaliveTime	在TCP应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。如果是通过Unix域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。 取值范围：0 ~ INT_MAX 设置为0值或者不配置表示使用系统缺省。 说明 禁用“保持激活”功能需要通过Pqopt参数设置keepalives=0。	KeepaliveTime=2
KeepaliveInterval	在TCP保持激活信息没有被应该传播的服务器承认之后，控制探活报文发送的间隔秒数。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。 取值范围：0 ~ INT_MAX 设置为0值或者不配置表示使用系统缺省。	KeepaliveInterval=2
KeepaliveCount	控制TCP发送保持激活信息的次数。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。 取值范围：0 ~ INT_MAX 设置为0值或者不配置表示使用系统缺省。	KeepaliveCount=2
SocketTimeout	用于控制客户端与服务端建立连接完全成功后的socket读写超时时间。单位为秒，默认为0。 说明 该参数包含数据库语句执行的超时时间。设置过小可能导致正常执行超时，因此默认为0，建议根据实际场景配置非0值。	SocketTimeout=5
SocketTimeoutInConnect	用于控制TCP三次握手成功后，客户端与服务端建立连接阶段的socket读写超时时间。单位为秒，默认为5。 说明 该参数与SocketTimeout的区别在于：由于三次握手成功后的客户端与服务端建立连接阶段可能存在阻塞，因此需要设置不影响语句执行超时的socket读写超时时间。	SocketTimeoutInConnect=5
CancelTimeout	用于控制应用端发送cancel消息的超时时间。 单位为秒，默认值为0。	CancelTimeout=5
TcpSYNRetries	用于控制在TCP三次握手阶段时SYN的最大重传次数。超过该次数仍未建立连接，则会返回报错。 默认值为0。 说明 该参数在支持TCP_SYNCNT套接字选项的操作系统上，指定客户端建立连接三次握手阶段SYN包发送失败而重传的次数。0值表示使用系统缺省。通过Unix域套接字的连接忽略这个参数。	TcpSYNRetries=3

参数	描述	示例
TextAsLongVarchar	将数据库内核中的text类型映射为驱动侧的SQL_LONGVARCHAR类型或者SQL_VARCHAR类型。 取值范围：0，1。 - 取值为0时表示将内核侧text类型映射为驱动侧的SQL_VARCHAR类型。 - 取值为1时表示将内核侧text类型映射为驱动侧的SQL_LONGVARCHAR类型。 默认值为1。	TextAsLongVarchar=1
MaxLongVarcharSize	定义驱动侧的SQL_LONGVARCHAR类型的可支持最大长度。 取值范围：0 ~ INT_MAX。 默认值为8190。	MaxLongVarcharSize=8190
MaxVarcharSize	定义驱动侧的SQL_VARCHAR类型的可支持最大长度。 取值范围：0 ~ INT_MAX。 默认值为255。	MaxVarcharSize=255
TargetServerType	设定连接的主机类型。主机的类型和设定的值一致时才能成功建立连接。设置规则如下： - primary（默认值）：仅对集群中的CN进行连接。 - cluster-primary：仅支持对主集群CN节点进行连接。 - cluster-mainnode：仅支持对容灾集群CN节点进行连接。 说明 cluster-primary和cluster-mainnode选项只支持流式容灾。	TargetServerType=cluster-primary
EnableHostCache	是否开启节点缓存功能。 0表示不开启。 1表示开启。 默认值为1。 说明 节点缓存在连接串配置单IP时不生效。	EnableHostCache=1

表 5-143 Sslmode 的可选项及其描述

Sslmode	是否会启用SSL加密	描述
disable	否	不使用SSL安全连接，为默认值。

Sslmode	是否会启用SSL加密	描述
allow	可能	如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。
prefer	可能	如果数据库支持SSL，则优先使用SSL安全加密连接，但不验证数据库服务器的真实性。
require	是	必须使用SSL安全连接，仅进行数据加密，不验证数据库服务器的真实性。
verify-ca	是	必须使用SSL安全连接，并且验证数据库是否具有可信证书机构签发的证书。 当前Windows ODBC不支持cert方式认证。
verify-full	是	必须使用SSL安全连接，在verify-ca的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。 GaussDB不支持此模式。 当前Windows ODBC不支持cert方式认证。

 说明

用户通过ODBC连接GaussDB服务器时，可以通过开启SSL加密客户端和服务端之间的通讯。在使用SSL时，默认用户已经获取了服务端和客户端所需要的证书和私钥文件，关于证书等文件的获取请参考Openssl相关文档和命令。

5.10.3 libpq

5.10.3.1 连接参数说明

libpq的所有连接参数统一以字符串形式构造和传递，无论参数类型是主机名、端口、密码还是布尔选项。

具体参数及其描述如下表所示：

表 5-144 连接参数

参数	描述	参数默认值
host	要连接的主机名。如果主机名以斜杠（/）开头，则它声明使用Unix域套接字通讯而不是TCP/IP通讯，该值就是套接字文件所存储的目录。如果没有声明host，那么默认是与位于/tmp目录（或者安装GaussDB的时候声明的套接字目录）里面的Unix-域套接字连接。在没有Unix域套接字的机器上，默认与localhost连接。 接受以“,”分割的字符串来指定多个主机名，支持指定多个主机名。	NULL

参数	描述	参数默认值
hostaddr	与之连接的主机的IP地址，是标准的IPv4地址格式，比如，172.28.40.9。如果机器支持IPv6，那么也可以使用IPv6的地址。如果声明了一个非空的字符串，那么使用TCP/IP通讯机制。 接受以“,”分割的字符串来指定多个IP地址，支持指定多个IP地址。 使用hostaddr取代host可以让应用避免一次主机名查找，这一点对于那些有时间约束的应用来说可能是非常重要的。不过，GSSAPI或SSPI认证方法要求主机名（host）。因此，应用下面的规则： 1. 如果声明了不带hostaddr的host那么就强制进行主机名查找。 2. 如果声明中没有host，hostaddr的值给出服务器网络地址。如果认证方法要求主机名，那么连接尝试将失败。 3. 如果同时声明了host和hostaddr，那么hostaddr的值作为服务器网络地址。host的值将被忽略，除非认证方法需要它，在这种情况下它将被用作主机名。 须知 • 如果host不是网络地址hostaddr处的服务器名，那么认证很有可能失败。 • 如果主机名（host）和主机地址都没有，那么libpq将使用一个本地的Unix域套接字进行连接，或者是在没有Unix域套接字的机器上，它将尝试与localhost连接。	NULL
port	主机服务器的端口号，或者在Unix域套接字连接时的套接字扩展文件名。 接受以“,”分割的字符串来指定多个端口号，支持指定多个端口号。	NULL
user	要连接的用户名，默认与运行该应用的用户操作系统名同名的用户。	当前操作系统用户名
dbname	数据库名，默认与用户名相同。	与user参数相同
password	如果服务器要求密码认证，所用的密码。	NULL
connect_timeout	连接的最大等待时间，以秒计，0或者不声明表示无穷大。不建议把连接超时的值设置小于2秒。	0（无超时）
client_encoding	为该连接设置client_encoding配置参数。除了对应的服务器选项接受的值，还可以使用auto从客户端中的当前环境中确定正确的编码（Unix系统上是LC_CTYPE环境变量）。	与服务器端编码一致
tty	该参数当前不生效（早期版本用于指定服务器调试输出的位置）。	NULL

参数	描述	参数默认值
options	指定在连接开始时发送到服务器的命令行选项。例如，将该参数设置为“-c geqo=off”表示设置会话的geqo参数值为off。	NULL
application_name	为application_name配置参数指定一个值，表明当前用户身份。	NULL
fallback_application_name	为application_name配置参数指定一个后补值。如果通过连接参数或PGAPPNAME环境变量没有为application_name给定一个值，将使用这个值。在一般工具程序中，若设置一个默认名，但不希望这个默认名被用户覆盖，可以通过指定一个后补值来实现。	NULL
keepalives	控制客户端侧的TCP保持激活是否使用。缺省值是1，表示保持激活；如果不需要保持激活，可以更改为0，表示关闭。通过Unix域套接字进行连接将忽略这个参数。	1
keepalives_idle	在TCP应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。0值或者不配置表示使用系统缺省。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。	默认值：0，表示使用系统TCP配置值。
keepalives_interval	在TCP保持激活信息没有被应该传播的服务器承认之后，控制探活报文发送的间隔秒数。0值或者不配置表示使用系统缺省。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。	默认值：0，表示使用系统TCP配置值。
keepalives_count	控制TCP发送保持激活信息的次数。0值或者不配置表示使用系统缺省。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。	默认值：0，表示使用系统TCP配置值。
tcp_user_timeout	在支持TCP_USER_TIMEOUT套接字选项的操作系统上，指定传输的数据在TCP连接被强制关闭之前可以保持未确认状态的最大时长。单位为毫秒，0值或者不配置表示使用系统缺省。通过Unix域套接字进行连接将忽略这个参数。	默认值：0，表示使用系统TCP配置值。
tcp_syn_retries	在支持TCP_SYNCNT套接字选项的操作系统上，指定客户端建立连接三次握手阶段SYN包发送失败而重新传输的次数。0值或者不配置表示使用系统缺省。通过Unix域套接字进行连接将忽略这个参数。	默认值：0，表示使用系统TCP配置值。

参数	描述	参数默认值
rw_timeout	设置客户端连接读写超时时间，单位为秒。默认值为0，表示不设置超时时间。 当libpq侧触发超时且连接关闭时，其下发给数据库侧正在运行的业务会被强制终止。该能力受GUC参数check_disconnect_query控制，设置为on表示支持该能力，设置为off表示不支持该能力。	0
sslmode	启用SSL加密的方式： • disable：不使用SSL安全连接。 • allow：如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。 • prefer：如果数据库支持，那么首选使用SSL安全加密连接，但不验证数据库服务器的真实性，为默认值。 • require：必须使用SSL安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。 • verify-ca：必须使用SSL安全连接，当前windows ODBC不支持cert方式认证。 • verify-full：必须使用SSL安全连接，当前windows ODBC不支持cert方式认证。	prefer
sslcompression	如果设置为1（默认），SSL连接之上传送的数据将被压缩（这要求OpenSSL版本为0.9.8或更高）。如果设置为0，压缩将被禁用（这要求OpenSSL版本为1.0.0或更高）。如果建立的是一个没有SSL的连接，这个参数会被忽略。如果使用的OpenSSL版本不支持该参数，它也会被忽略。压缩会占用CPU时间，但是当瓶颈为网络时可以提高吞吐量。如果CPU性能是限制因素，禁用压缩能够改进响应时间和吞吐量。	1（开启SSL时）
sslcert	该参数指定客户端SSL证书的文件名。如果没有建立SSL连接，这个参数会被忽略。	NULL
sslkey	该参数指定用于客户端证书的密钥位置。它能够指定一个从外部“引擎”（引擎是OpenSSL的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立SSL连接，这个参数会被忽略。	NULL
sslrootcert	该参数指定一个包含SSL证书机构（CA）证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。	NULL
sslcrll	该参数指定SSL证书撤销列表（CRL）的文件名。列在这个文件中的证书如果存在，在尝试认证该服务器证书时会被拒绝。	NULL

参数	描述	参数默认值
requirepeer	该参数指定服务器的操作系统用户，例如 requirepeer=postgres。当建立一个Unix域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是否运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与TCP/IP连接上SSL证书相似的服务器认证（注意，如果Unix域套接字在/tmp或另一个公共可写的位置，任何用户能启动一个在那里侦听的服务器。使用这个参数来保证所连接的是一个由可信用户运行的服务器）。这个选项只在实现了peer认证方法的平台上支持。	NULL
krbsrvname	当用GSSAPI认证时，要使用的Kerberos服务名。为了让Kerberos认证成功，这必须匹配在服务器配置中指定的服务名。	NULL
gsslib	用于GSSAPI认证的GSS库。只用在Windows上。设置为gssapi可强制libpq用GSSAPI库来代替默认的SSPI进行认证。	NULL
service	用于附加参数的服务名。它指定保持附加连接参数的pg_service.conf中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。	NULL
authtype	不再使用“authtype”，因此将其标记为“不显示”。将其保留在数组中，以免拒绝旧应用程序中的conninfo字符串，这些应用程序可能仍在尝试设置它。	NULL
remote_node_name	指定连接本地节点的远端节点名称。	NULL
localhost	指定在一个连接通道中的本地地址。	NULL
localport	指定在一个连接通道中的本地端口。	NULL
fencedUdfRPCMode	控制fenced UDF RPC协议是使用unix域套接字或特殊套接字文件名。缺省值是0，意思为关闭。使用unix domain socket模式，文件类型为“.s.PGSQL.%d”，但是要使用fenced udf，文件类型为.s.fencedMaster_unixdomain，可以更改为1，意思为开启。	0

参数	描述	参数默认值
replication	这个选项决定是否该连接应该使用复制协议而不是普通协议。这是PostgreSQL的复制连接以及pg_basebackup之类的工具在内部使用的协议，但也可以被第三方应用使用。支持下列值，大小写无关： • true、on、yes、1、reply 用于集群内的连接，连接进入到物理复制模式。 • standby_cluster、hadr_main_standby、hadr_standby_cn 用于集群间的连接，连接进入到物理复制模式。 • database 连接进入到逻辑复制模式，连接到dbname参数中指定的数据库。 • false、off、no、0、NULL（缺省值） 该连接是一个常规连接，这是默认行为。 在物理或者逻辑复制模式中，仅能使用简单查询协议。	NULL
backend_version	传递到远端的后端版本号。	NULL
prototype	设置当前协议级别，默认：1。该参数当前不生效。	1
enable_ce	控制是否允许客户端连接全密态数据库。默认值为0，不开启密态功能。如果需要开启密态等值查询基本能力，则修改为1或者3。	0
key_info	与enable_ce一起使用，在密态数据库中，用于设置访问外部密钥管理者的参数。	NULL
connection_info	connection_info是一个包含driver_name、driver_version、driver_path和os_user的json字符串。 如果不为NULL，使用connection_info 忽略connectionExtraInfo。 如果为NULL，生成与libpq相关的连接信息字符串，当connectionExtraInfo为0时connection_info只有driver_name和driver_version。	0
connectionExtraInfo	是否将当前驱动的部署路径、进程属主用户信息上报到数据库。默认值为0，表示不打开；设置1，表示打开。 说明 当设置为1时，libpq会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在GUC参数connection_info里，同时可以在PG_STAT_ACTIVITY中查询到。	0

参数	描述	参数默认值
target_session_attrs	设定连接的主机的类型。主机的类型和设定的值一致时才能连接成功。指定多IP或者域名对应的是多IP时才会校验此参数。target_session_attrs的设置规则如下： • any：可以对所有类型的主机进行连接。 • read-write：仅对可读写的主机进行连接。 • read-only：仅对只读的主机进行连接。 • primary（默认值）：仅对主备系统中的主节点进行连接。 • standby：仅对主备系统中的备节点进行连接。 • prefer-standby：首先尝试与主备系统中的备节点进行连接。如果对hosts列表的所有备节点都连接失败，那么尝试“any”模式进行连接。	primary
enable_host_cache	是否开启节点缓存功能，0表示不开启，1表示开启。默认不开启。 说明 节点缓存在连接串配置单IP或者target_session_attrs=any时不生效。	0
hostcache_expiration_time	libpq尝试连接主机后将记录主机状态。在设定的hostcache_expiration_time时间内保持记录的主机状态可信，超时后记录的主机状态失效。参数单位为秒（s），默认值为10，取值范围：0 ~ INT_MAX。 说明 hostcache_expiration_time在enable_host_cache=1时生效。	10
driver_type	区分libpq的上层驱动类型，默认为"GaussDB LIBPQ Driver"。该参数不建议设置，保持默认即可。	GaussDB LIBPQ Driver

参数	描述	参数默认值
force_no_truncation/supportZeroCharacter	使部分数据库执行语句函数和异步命令处理函数在以文本格式传递参数（paramFormats中对应的值为0）时，强制使用函数调用者指定的参数长度（paramLengths中对应的值）。force_no_truncation/supportZeroCharacter的设置规则如下： 1：打开。需要保证客户端字符集和数据库字符集一致。 0（默认值）：关闭。 force_no_truncation/supportZeroCharacter打开后影响以下数据库执行语句函数和异步命令处理函数： - PQexecParams - PQexecParamsBatch - PQexecPrepared - PQexecPreparedBatch - PQsendQueryParams - PQsendQueryPrepared 须知 使用以上涉及的数据库执行语句函数和异步命令处理函数时，传递包含'\0'字符的文本参数值但不希望导致文本参数值被'\0'字符截断，则必须打开此参数。	0

须知

配置以上连接参数时，建议配置合法值，如果配置非法值，可能导致建立连接失败或者实际生效值为默认值。

建立连接的首包介绍

驱动与数据库建立连接过程中，Socket（套接字）建立之后，由驱动以key value形式向数据库发送的第一个报文为首包。首包内容包括用户名（user）、数据库名（database）、部分GUC参数信息（replication、redistribution、backend_version、application_name、client_encoding、connect_timeout、enable_full_encryption、DateStyle、TimeZone、geqo、connection_info、gaussdb.capacity_version）以及options配置参数信息。GUC参数的介绍参见《参考》中“数据库运行参数说明 > GUC参数说明”。

须知

- gaussdb.capacity_version参数表示驱动支持的兼容性能力集的版本号，用于驱动与数据库协商此次建立连接时使用的兼容性特性集合。该参数为内部参数。
- 数据库执行以下命令后，上述首包GUC参数的值将被更改为首包设置的值。

```
reset all;
```

5.10.4 ecpg

5.10.4.1 预备语句

当SQL语句的条件参数在编译期未知，或该语句在业务逻辑中需要被高频重复执行时，建议使用预备语句机制。此机制可显著提升执行效率并防止SQL注入。

- 使用命令PREPARE准备语句。对于对于运行期才确定的动态值使用占位符“?”：
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
- 如果一个语句返回单行，应用程序可以在PREPARE执行语句之后调用EXECUTE，同时使用USING子句为占位符提供实际值：
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
- 如果一个语句返回多行，应用程序可以使用基于预备语句声明的游标。为了绑定输入参数，必须使用USING子句打开游标：
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* 当结果集达到最后时，跳出while循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
 EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
 ...
}
EXEC SQL CLOSE foo_bar;
- 当不再需要预备语句的时候，应释放语句：
EXEC SQL DEALLOCATE PREPARE name;

5.10.4.2 数据类型和宿主变量

5.10.4.2.1 概述

本节旨在详细介绍如何在C语言程序与嵌入式SQL语句之间，利用宿主变量实现高效、安全的数据传递。

在嵌入式SQL-C程序中，C语言被视为宿主语言。凡是定义在C代码中并在EXEC SQL语句里被引用的变量，统称为宿主变量。它们是打通C语言内存空间与数据库内核数据之间的核心媒介。

数据交互的本质是C语言原生数据类型与GaussDB数据类型之间的匹配。为了确保数据在传递过程中不发生精度丢失、类型冲突或内存溢出，须深刻理解类型映射机制。在声明宿主变量时，需要根据GaussDB底层字段的属性，选择与之对应的C语言数据类型（包括初级类型、结构体、联合体及数组等），从而在预编译阶段建立精准的数据对齐。

在SQL语句中引用宿主变量时，无需手动将数据拼接至字符串中。您只需在变量名前添加冒号（:）作为前缀，ecpg预处理器即可自动识别并处理。这种参数化的引用方式不仅简化了开发，更从根本上增强了程序防御SQL注入攻击的能力。

示例如下：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

该语句同时引用了两个C语言变量v1和v2。这表明在同一条SQL语句内，宿主变量可以与普通的SQL常量（如'foo'）混合使用，极大地提升了语句构建的灵活性。

5.10.4.2.2 声明段

要在嵌入式SQL-C程序与数据库之间实现数据的双向传递（例如将C程序的参数传递给SQL语句，或将查询结果回传给C程序），必须在特定的声明段中定义相关的C语言变量。这一段代码的核心作用是告知ecpg预处理器哪些变量将被用于SQL交互，以便预处理器能够准确识别这些变量的内存地址与数据类型。

声明段必须由EXEC SQL BEGIN DECLARE SECTION;指令开始，并以EXEC SQL END DECLARE SECTION;指令结束。在此区间内，您可以按照标准C语言语法声明变量：

```
EXEC SQL BEGIN DECLARE SECTION;
int x = 4;
char foo[16], bar[16];
EXEC SQL END DECLARE SECTION;
```

说明

- 标记段代码开始和结束之间声明的宿主变量类型必须为当前支持的数据类型，请参见[表 5-145](#)。
- 对于简单的单一变量，您可以省略完整的声明段包裹，通过指令前缀进行隐式声明：EXEC SQL int i = 4。
- 不在SQL命令里使用的变量可以在特殊的声明段外面声明。只有在声明区域内声明的变量才能在SQL语句中使用。
- 若宿主变量涉及自定义的结构体或联合体，其完整的类型定义也必须包裹在声明段中，否则ecpg预编译器将无法解析其底层内存结构。

5.10.4.2.3 检索查询

对于常用的检索查询，嵌入式SQL提供了常规命令SELECT和FETCH的特殊变体。这些命令使用特殊的INTO子句，用以指定检索出来的数值存储在哪些宿主变量里。SELECT用于返回单行的查询，FETCH用于使用游标返回多行的查询。

- 使用SELECT

```
/*
 * 假设已存在表test:
 * CREATE TABLE test (a int, b varchar(50));
 */
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

INTO子句出现在选择列表和FROM子句之间。选择列表和INTO后面列表的元素（也叫目标列表）个数必须相同。

- 使用FETCH

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;
...
do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
```



```
...  
} while (...);
```

这里的INTO子句出现在所有SQL子句后面。

5.10.4.2.4 数据类型映射

当ecpg应用程序在GaussDB服务器和C语言程序之间交换值时（例如：从服务器检索查询结果或者执行带有输入参数的SQL语句），在GaussDB数据类型和宿主语言变量类型（具体的C语言数据类型）之间需要进行值的转换。有两种数据类型可以使用：简单的GaussDB数据类型，如integer和text，可以直接被应用程序读取和写入。其他GaussDB数据类型，如timestamp和numeric，只能通过特殊库函数进行访问，请参见[ecpg接口参考](#)章节。

说明

- 当前仅支持对于C语言的基本数据类型的使用或者组合，不支持C++语言中string数据类型用作宿主变量数据类型。
- 当前ecpg仅对SQL的常用数据类型做映射，具体支持项请参见[表5-145](#)。

表 5-145 GaussDB 数据类型和 C 变量类型之间的映射

GaussDB数据类型	宿主变量数据类型
smallint	short
integer	int
bigint	long long int
boolean	boolean
character(n), varchar(n), text	char[n+1], VARCHAR[n+1]
double precision	double
real	float
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
name	char[NAMEDATALEN]
date	date [a] 说明 [a]表示特殊数据类型，此类型可以通过 访问特殊数据类型 访问。
timestamp	timestamp [a]
interval	interval [a]
decimal	decimal [a]
numeric	numeric [a]

5.10.4.2.5 处理字符串

处理SQL字符串数据类型（例如：varchar、text），有两种方式来声明宿主变量：

- 使用char[]（一个char字符串），该方式是C语言程序中处理字符数据最常见的方式。

```
EXEC SQL BEGIN DECLARE SECTION;  
char str[50];  
EXEC SQL END DECLARE SECTION;
```

注意字符串必须控制长度，如果上述示例的宿主变量用作存放查询结果且查询命令返回的字符串长度超过49字节，那么将会发生缓冲区溢出。

- 使用VARCHAR类型（ecpg提供的一种特殊类型）。在一个VARCHAR类型数组上的定义会被转变成一个struct类型。如下声明：

```
VARCHAR var[180];
```

会被转变成：

```
struct varchar_var  
{  
    int len;  
    char arr[180];  
} var;
```

为了在VARCHAR宿主变量中存储字符串，该变量必须声明为包含以零字节（'\0'）作为终止符的字符串。字段arr存放以零字节为终止符的字符串，字段len存放存储在arr中的字符串的长度，计算长度时不包括终止符。当宿主变量被用于一个查询的输入时，如果strlen(arr)和len结果不同，将使用较短的那一个。

说明

- VARCHAR可以写为大写或小写形式，但是不能大小写混合。
- char和VARCHAR类型宿主变量也可以保存其他SQL类型的值，它们将被存储为字符串形式。

5.10.4.2.6 使用非初级类型的宿主变量

非初级类型的宿主变量包括数组、typedef、结构体和指针类型的宿主变量。

数组

有两种将数组作为宿主变量的情况。

- 在char[]或者VARCHAR[]中存储文本字符串。
- 在检索多行查询结果时不使用游标。

如果不使用数组，则处理多行查询结果时**必须使用游标以及FETCH命令**。但是如果使用数组类型作为宿主变量，则一次可以检索多行。数组长度需能容纳查询结果的所有行，否则可能会发生缓冲区溢出。

下面示例为扫描pg_database系统表并且显示所有可用数据库的OID和名称：

前提条件：需要创建testdb库。

```
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
    EXEC SQL BEGIN DECLARE SECTION;
```

```
int dbid[8];
char dbname[8][16];
int i;
EXEC SQL END DECLARE SECTION;

memset(dbname, 0, sizeof(char)* 16 * 8);
memset(dbid, 0, sizeof(int) * 8);

/* 连接到testdb */
EXEC SQL CONNECT TO testdb;

/* 一次检索多行到数组中。 */
EXEC SQL SELECT oid,dbname INTO :dbid, :dbname FROM pg_database;
for (i = 0; i < 8; i++)
    printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);
EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}
```

运行结果如下（具体值取决于本地环境）：

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

结构体

结构体成员变量可用来匹配查询结果列的名称，该结构能在一个宿主变量中处理多列值。

以下示例从pg_database系统表以及使用pg_database_size()函数检索可用数据库的OID、名称和尺寸。在此示例中，按照结构体的成员名匹配SELECT结果的每一列，而不必将多个宿主变量放在FETCH语句中。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int oid;
    char datname[65];
    long long int size;
} dbinfo_t;

dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;
memset(&dbval, 0, sizeof(dbinfo_t));

EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
pg_database;
EXEC SQL OPEN cur1;

/* 在达到结果集末尾时，跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将多列取到一个结构体中。 */
    EXEC SQL FETCH FROM cur1 INTO :dbval;
    /* 打印该结构体的成员。 */
    printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, dbval.size);
}
EXEC SQL CLOSE cur1;
```

运行结果如下（具体值取决于本地环境）：

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

结构体宿主变量可将查询结果的部分列转化成结构体字段，其他的查询结果列可以被分配给其它宿主变量。上述的示例也可以使用结构体外部的size宿主变量重新构造：

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int oid;
    char datname[65];
} dbinfo_t;

dbinfo_t dbval;
long long int size;
EXEC SQL END DECLARE SECTION;

memset(&dbval, 0, sizeof(dbinfo_t));

EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
pg_database;
EXEC SQL OPEN cur1;

/* 在达到结果集末尾时，跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;
while (1)
{
    /* 将多列取到一个结构中。 */
    EXEC SQL FETCH FROM cur1 INTO :dbval, :size;
    /* 打印该结构的成员。 */
    printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
}

EXEC SQL CLOSE cur1;
```

typedef

使用typedef关键字将新类型映射到已有类型：

```
EXEC SQL BEGIN DECLARE SECTION;
typedef char mychartype[40];
typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

也可以使用如下命令：

```
EXEC SQL TYPE serial_t IS long;
```

这种声明不需要写在一个声明段中。

指针

可以声明常见类型的指针：

```
EXEC SQL BEGIN DECLARE SECTION;
int *intp;
char **charp;
EXEC SQL END DECLARE SECTION;
```

5.10.4.2.7 访问特殊数据类型

ecpg支持numeric、decimal、date、timestamp和interval数据类型。由于这些数据类型的内部结构较为复杂，无法被映射到初级数据类型的宿主变量，因此应用程序通过声明特殊类型的宿主变量，以及使用pgtypes库中的函数来处理这些特殊类型。pgtypes库中的接口函数请参见[ecpg接口参考](#)章节。

- timestamp、date

程序必须包含timestamp类型的头文件：

```
#include <pgtypes_timestamp.h>
```

在声明部分中声明类型为timestamp的宿主变量：

```
EXEC SQL BEGIN DECLARE SECTION;  
    timestamp ts;  
EXEC SQL END DECLARE SECTION;
```

读取值到宿主变量之后，使用pgtypes库函数进行处理。如下示例中，使用PGTYPEStimestamp_to_asc()函数将timestamp值转换成文本形式：

```
EXEC SQL SELECT now()::timestamp INTO :ts;  
printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

示例输出如下：

```
ts = 2022-06-27 18:03:56.949343
```

另外，date类型可以用相同的方式处理。程序必须包括pgtypes_date.h头文件，声明一个date类型的宿主变量并用PGTYPEdata_to_asc()函数将其转换成文本形式。

- interval

interval类型的处理与timestamp和date类型类似，不同的是interval类型变量的存储空间需要分配在堆内存中。

示例如下：

```
#include <stdio.h>  
#include <stdlib.h>  
#include <pgtypes_interval.h>  
int main(void)  
{  
    EXEC SQL BEGIN DECLARE SECTION;  
    interval *in;  
    EXEC SQL END DECLARE SECTION;  
  
    /* 需提前创建testdb库 */  
    EXEC SQL CONNECT TO testdb;  
    in = PGTYPEStimestamp_new();  
    EXEC SQL SELECT '1 min'::interval INTO :in;  
    printf("interval = %s\n", PGTYPEStimestamp_to_asc(in));  
    PGTYPEStimestamp_free(in);  
    EXEC SQL COMMIT;  
    EXEC SQL DISCONNECT ALL;  
    return 0;  
}
```

运行结果如下：

```
interval = @ 1 min
```

- numeric、decimal

numeric和decimal类型的处理类似于interval类型，需要定义一个指针，在堆上分配一些内存空间并且使用pgtypes库函数访问该变量。

示例如下：

```
#include <stdio.h>  
#include <stdlib.h>  
#include <pgtypes_numeric.h>  
EXEC SQL WHENEVER SQLERROR STOP;  
int main(void)  
{  
    EXEC SQL BEGIN DECLARE SECTION;  
    numeric *num;  
    numeric *num2;  
    decimal *dec;  
    EXEC SQL END DECLARE SECTION;  
  
    /* 需提前创建testdb库 */  
    EXEC SQL CONNECT TO testdb;
```

```
num = PGTYPENumeric_new();
dec = PGTYPEDecimal_new();

EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;
printf("numeric = %s\n", PGTYPENumeric_to_asc(num, 0));
printf("numeric = %s\n", PGTYPENumeric_to_asc(num, 1));
printf("numeric = %s\n", PGTYPENumeric_to_asc(num, 2));

/* 转换十进制到数值型以显示十进制值 */
num2 = PGTYPENumeric_new();
PGTYPENumeric_from_decimal(dec, num2);
printf("decimal = %s\n", PGTYPENumeric_to_asc(num2, 0));
printf("decimal = %s\n", PGTYPENumeric_to_asc(num2, 1));
printf("decimal = %s\n", PGTYPENumeric_to_asc(num2, 2));
PGTYPENumeric_free(num2);
PGTYPEDecimal_free(dec);
PGTYPENumeric_free(num);

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}
```

运行结果如下：

```
numeric = 12
numeric = 12.4
numeric = 12.35
decimal = 23
decimal = 23.5
decimal = 23.46
```

5.10.4.2.8 处理非初级 SQL 数据类型

本节介绍如何处理ecpg应用中的非标量以及用户定义的SQL级别的数据类型。注意此处和[使用非初级类型的宿主变量](#)章节中介绍的对于非初级类型的宿主变量的处理不同。

- 数组

ecpg不直接支持多维SQL级别数组。一维SQL数组可以被映射到C语言数组类型的宿主变量。但是在创建语句时，如果ecpg无法识别列的类型，则将无法检查C语言数组是否是SQL数组的输入。因此在处理SQL语句的输出时，ecpg需要检查两者是否都是数组。

如果查询语句仅仅访问一个数组的元素，那么用一个能被映射到该元素类型的宿主变量即可。例如：如果列的类型是integer数组，可以使用int类型的宿主变量。如果元素类型是varchar或text，可以使用char[]或VARCHAR[]类型的宿主变量。

示例如下：

```
CREATE TABLE t3 (
  ii integer[]
);
testdb=> SELECT * FROM t3;
      ii
-----
{1,2,3,4,5}
(1 row)
```

检索数组的第四个元素并且把它存储到一个int类型的宿主变量中的示例如下：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}
EXEC SQL CLOSE cur1;
```

示例输出：

ii=4

若要把多个数组元素映射到一个数组类型宿主变量中，数组列以及宿主变量数组的每一个元素都需单独处理。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

注意：无法将一个数组类型的查询结果直接映射到数组类型的宿主变量。

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 错误 */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

- 组合类型

ecpg不直接支持组合类型，例如：在struct中声明成员变量为date数据类型。但是可以单独访问每一个组合类型中的属性或者使用外部字符串表达。

对于如下示例，可以单独访问每一个属性：

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'GaussDB') );
```

如下示例单独检索类型comp_t的每一个属性：

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* 将组合类型列的每一个元素放在 SELECT 列表中。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
while (1)
{
    /* 将组合类型列的每一个元素取到主变量中。 */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;
    printf("intval=%d, textval=%s\n", intval, textval.arr);
}
```

```
}  
EXEC SQL CLOSE cur1;
```

FETCH命令中从SQL语句接收并存储的结果宿主变量可以被集中在一个结构体中。结构体类型的宿主变量的详情请参见[处理字符串](#)章节。如下示例中两个宿主变量intval和textval变成comp_t结构体的成员，并且该结构体在FETCH命令中被指定：

```
EXEC SQL BEGIN DECLARE SECTION;  
typedef struct  
{  
    int intval;  
    varchar textval[33];  
} comp_t;  
comp_t compval;  
EXEC SQL END DECLARE SECTION;  
  
/* 将组合类型列的每一个元素放在 SELECT 列表中。 */  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;  
EXEC SQL OPEN cur1;  
  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
while (1)  
{  
    /* 将 SELECT 列表中的所有值放入一个结构。 */  
    EXEC SQL FETCH FROM cur1 INTO :compval;  
    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);  
}  
EXEC SQL CLOSE cur1;
```

即使在FETCH命令中使用了结构体，SELECT子句中的属性名依然要依次指定，为了更加方便，可以使用一个“*”来处理该组合类型值的所有属性，示例如下：

```
...  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;  
EXEC SQL OPEN cur1;  
  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
  
while (1)  
{  
    /* 将 SELECT 列表中的所有值放入一个结构。 */  
    EXEC SQL FETCH FROM cur1 INTO :compval;  
    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);  
}  
...
```

虽然ecpg无法解析组合类型本身，但是可以通过上述方法将组合类型映射到结构体中。

- 用户定义的基础类型

ecpg使用宿主变量来存放查询结果时，只支持ecpg提供的数据类型，不支持用户自定义的数据类型。对于通过CREATE TYPE创建的数据类型，无法通过宿主变量找到映射关系。

用户自定义类型可以使用外部字符串表达以及char[]或VARCHAR[]类型的宿主变量来处理。

数据类型complex的外部字符串表达是(%lf,%lf)，被定义在函数complex_in()内。如下示例把复杂类型值(1,1)和(3,3)插入到列a和b，并且查询它们的值。

```
EXEC SQL BEGIN DECLARE SECTION;  
varchar a[64];  
varchar b[64];  
EXEC SQL END DECLARE SECTION;  
EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;  
EXEC SQL OPEN cur1;  
  
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```



```
while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :a, :b;
    printf("a=%s, b=%s\n", a.arr, b.arr);
}
EXEC SQL CLOSE cur1;
```

示例运行结果：

a=(1,1), b=(3,3)

5.10.4.3 执行动态 SQL 语句

5.10.4.3.1 概述

在大多数情况下，应用程序执行的SQL语句在编写应用程序时必须是已知的。

但是在某些情况下，SQL语句是在运行时进行构造或由外部提供的。这种情况下不能将SQL语句直接嵌入到C语言源代码中，但是动态SQL语句支持通过一个字符串变量调用所提供的SQL语句。

5.10.4.3.2 执行没有结果集的语句

执行EXECUTE IMMEDIATE命令示例如下：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE IMMEDIATE :stmt;
```

EXECUTE IMMEDIATE可以用于不返回结果集的SQL语句，比如：DDL、INSERT、UPDATE和DELETE语句。但不支持用此种方式执行检索数据的语句，比如：SELECT语句。

5.10.4.3.3 执行具有输入参数的语句

准备一个包含问号“?”作为参数占位符的SQL语句。

使用EXECUTE语句和USING子句来执行该语句并传递参数。示例如下：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;
/* PREPARE 准备一个语句用于执行 */
EXEC SQL PREPARE mystmt FROM :stmt;
...
/* 单引号为有效字符，若用字符串需用双引号 */
EXEC SQL EXECUTE mystmt USING 42, 'foobar';

/* 当不再需要预备语句时，应该释放它 */
EXEC SQL DEALLOCATE PREPARE name;
```

5.10.4.3.4 执行带有结果集的语句

执行具有单独结果集的SQL语句，可以使用EXECUTE。

若保存结果，则需要额外使用INTO子句。示例如下：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

📖 说明

EXECUTE命令支持INTO子句和USING子句。

如果一个查询可能返回多个结果行，则应使用游标，游标详情请参见[使用游标](#)，示例如下：

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.username as dbaname, d.datname "
            " FROM pg_database d, gs_user u "
            " WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}
EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

5.10.4.4 错误处理

5.10.4.4.1 概述

有两种非互斥的方法可以处理嵌入式SQL程序的异常情况和告警：

- 使用WHENEVER命令设置回调，处理告警和错误。参见[设置回调](#)。
- 通过sqlca获取错误或者告警的详细信息，进行相应处理。参见[sqlca](#)。

5.10.4.4.2 设置回调

当告警或者错误发生时，直接执行具体操作进行处理，设置回调命令如下：

```
EXEC SQL WHENEVER condition action;
```

condition取值范围：

- SQLERROR：当在SQL语句执行期间发生错误时，调用指定操作。
- SQLWARNING：当在SQL语句执行期间发生告警时，调用指定操作。
- NOT FOUND：当SQL语句检索或者影响为零行时，调用指定操作。

action取值范围：

- CONTINUE：忽略回调错误条件，继续执行，通常可以用来停止break包含条件，为缺省值。
- GOTO label/GO TO label：跳转到指定label（使用C语言goto语句）。
- SQLPRINT：输出消息到标准错误。
- STOP：调用exit(1)，终止程序。

- DO BREAK：执行C语句break，只能在循环或者switch语句中使用。

示例如下：

```
/* 当出现一个告警时打印一个消息 */
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
/* 发生一个错误时中止程序。 */
EXEC SQL WHENEVER SQLERROR STOP;
```

须知

- EXEC SQL WHENEVER 是 SQL 预处理器指令，而非C语言语句。它用于设置错误或警告的处理动作，该设置对后续所有嵌入式SQL语句生效，除非在第一个EXEC SQL WHENEVER语句与触发错误或警告的SQL语句之间，针对同一情况设置了不同的处理动作。因此下面的两个C语言程序都不会得到预期的效果：

```
/* 错误 */
void func()
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/* 错误 */
void func()
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

- DO BREAK只能用于while/for/switch场景，且执行结束时需要使用CONTINUE语句忽略回调错误条件。

5.10.4.4.3 sqlca

嵌入式SQL接口提供了sqlca（SQL通信区）的全局变量。sqlca包含告警和错误信息。如果在语句执行期间发生多个告警和错误，那么sqlca只保存最后一个信息。在一个多线程的程序中，每一个线程会自动得到它的sqlca副本。

数据结构如下：

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
```

```
char sqlstate[5];  
} sqlca;
```

如果SQL语句没有发生错误，则sqlca.sqlcode为0，sqlca.sqlstate为"00000"。如果发生了告警或者错误，那么sqlca.sqlcode为负数并且sqlca.sqlstate不同于"00000"。SQLSTATE与SQLCODE的具体值请参见[SQLSTATE与SQLCODE](#)章节。

当SQL语句成功执行时，sqlca.sqlerrd[1]将存储被处理行的OID，而sqlca.sqlerrd[2]则记录被处理或返回的行数。若执行过程中出现错误或警告，sqlca.sqlerrm.sqlerrmc将包含描述该错误的字符串，同时sqlca.sqlerrm.sqlerrml表示该错误消息的长度（即strlen()的结果）。需要注意的是，由于sqlerrmc数组长度固定，部分较长的错误消息可能会被截断。

在发生告警时，sqlca.sqlwarn[2]被设置为W。

sqlcaid、sqlabc、sqlerrp、sqlwarn以及sqlerrd的剩余元素目前未包含有用的信息。

示例如下：

```
/* 整合WHENEVER和sqlca实现错误处理 */  
EXEC SQL WHENEVER SQLERROR SQLCALL print_sqlca();  
  
void print_sqlca()  
{  
    fprintf(stderr, "==== sqlca ====\\n");  
    fprintf(stderr, "sqlcode: %ld\\n", sqlca.sqlcode);  
    fprintf(stderr, "sqlerrm.sqlerrml: %d\\n", sqlca.sqlerrm.sqlerrml);  
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\\n", sqlca.sqlerrm.sqlerrmc);  
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\\n", sqlca.sqlerrd[0], sqlca.sqlerrd[1], sqlca.sqlerrd[2],  
        sqlca.sqlerrd[3], sqlca.sqlerrd[4], sqlca.sqlerrd[5]);  
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\\n", sqlca.sqlwarn[0], sqlca.sqlwarn[1],  
        sqlca.sqlwarn[2],  
        sqlca.sqlwarn[3], sqlca.sqlwarn[4], sqlca.sqlwarn[5],  
        sqlca.sqlwarn[6], sqlca.sqlwarn[7]);  
    fprintf(stderr, "sqlstate: %5s\\n", sqlca.sqlstate);  
    fprintf(stderr, "=====\\n");  
}
```

运行结果如下（此处是一个拼写表名错误）：

```
==== sqlca ====  
sqlcode: -400  
sqlerrm.sqlerrml: 49  
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38  
sqlerrd: 0 0 0 0 0 0  
sqlwarn: 0 0 0 0 0 0 0 0  
sqlstate: 42P01  
=====
```

5.10.4.4 SQLSTATE 与 SQLCODE

SQLSTATE是一个由五个字符组成的数组。这五个字符包含数字或大写字母，它表示多种错误或告警情况的代码。SQLSTATE具有一种层次模式：前两个字符表示情况的总体分类，后三个字符表示总体情况的子类。例如：代码00000表示成功状态

SQLCODE以整数的形式反映SQL语句的执行状态。具体而言：

- 零值标志着操作的成功执行；
- 正值通常指示成功但伴随附加信息，其中SQL标准仅明确定义了+100，用以表明上一条命令未对任何行产生影响或返回结果；
- 负值则揭示了执行过程中遭遇的错误。

值得注意的是，SQL标准并未对负值进行具体定义，它们通常由数据库系统自行规定以表示不同的错误情况。

说明

- ecpg为嵌入式SQL新增加的SQLSTATE码有：22002、07001、07002、07006、07009、33000、42601、42804、42P03、YE000和YE001。其余SQLSTATE码沿用内核SQLSTATE码。
- SQLCODE为-400表示ecpg检测到内核服务器返回错误，其SQLSTATE为内核相应错误的SQLSTATE。

表 5-146 SQLSTATE 与 SQLCODE 对应关系表

SQLCODE值	SQLSTATE值	含义
0 (ECPG_NO_ERROR)	SQLSTATE 00000	表示没有错误。
100 (ECPG_NOT_FOUND)	SQLSTATE 02000	一种无害情况，它表示上一个命令检索或者处理了零行，或者已到达游标的末尾。 在循环中处理游标时，可以使用以下代码来检测何时中止该循环，示例如下： <pre>while (1) { EXEC SQL FETCH ... ; if (sqlca.sqlcode == ECPG_NOT_FOUND) break; }</pre> 实际上WHENEVER NOT FOUND DO BREAK也会在内部这样做，所以一般不会直接使用这种方法。
-12 (ECPG_OUT_OF_MEMORY)	SQLSTATE YE001	虚拟内存已被耗尽，数字值被定义为-ENOMEM。
-200 (ECPG_UNSUPPORTED)	SQLSTATE YE000	预处理器产生了一些该库无法识别的内容。
-201 (ECPG_TOO_MANY_ARGUMENTS)	SQLSTATE 07001 或 07002	表示命令指定的宿主变量数量超过该命令预期。
-202 (ECPG_TOO_FEW_ARGUMENTS)	SQLSTATE 07001 或 07002	表示命令指定的宿主变量数量低于该命令预期。
-203 (ECPG_TOO_MANY_MATCHES)	SQLSTATE 21000	表示一个查询已经返回了多行，但是该语句仅支持存储一个结果行。
-204 (ECPG_INT_FORMAT)	SQLSTATE 42804	宿主变量是int类型，但数据库中的数据类型不同，且包含无法转换为int的值。该库使用strtol()进行转换。

SQLCODE值	SQLSTATE值	含义
-205 (ECPG_UINT_FORMAT)	SQLSTATE 42804	宿主变量是unsigned int类型，但数据库中的数据类型不同，且包含无法转换为unsigned int的值。该库使用strtoul()进行转换。
-206 (ECPG_FLOAT_FORMAT)	SQLSTATE 42804	宿主变量是float类型，但数据库中的数据类型不同，且包含无法转换为float的值。该库使用strtod()进行转换。
-207 (ECPG_NUMERIC_FORMAT)	SQLSTATE 42804	宿主变量是numeric类型，但数据库中的数据类型不同，且包含无法转换为numeric的值。
-208 (ECPG_INTERVAL_FORMAT)	SQLSTATE 42804	宿主变量是interval类型，但数据库中的数据类型不同，且包含无法转换为interval的值。
-209 (ECPG_DATE_FORMAT)	SQLSTATE 42804	宿主变量是date类型，但数据库中的数据类型不同，且包含无法转换为date的值。
-210 (ECPG_TIMESTAMP_FORMAT)	SQLSTATE 42804	宿主变量是timestamp类型，但数据库中的数据类型不同，且包含无法转换为timestamp的值。
-211 (ECPG_CONVERT_BOOLEAN)	SQLSTATE 42804	宿主变量是类型boolean而数据库中的数据既不是't'也不是'f'。
-212 (ECPG_EMPTY)	SQLSTATE YE000	发送给SQL服务器的语句是空值（通常在一个嵌入式SQL程序中不会发生，因此它可能指向一个内部错误）。
-213 (ECPG_MISSING_INDICATOR)	SQLSTATE 22002	返回了一个空值并且没有提供空值指示符。
-214 (ECPG_NO_ARRAY)	SQLSTATE 42804	在要求一个数组的地方使用了一个普通变量。
-215 (ECPG_DATA_NOT_ARRAY)	SQLSTATE 42804	在一个要求数组值的地方数据库返回了一个普通变量。
-216 (ECPG_ARRAY_INSERT)	SQLSTATE 42804	该值不能被插入到数组中。
-220 (ECPG_NO_CONN)	SQLSTATE 08003	程序尝试访问一个不存在的连接。
-221 (ECPG_NOT_CONN)	SQLSTATE YE000	程序尝试访问一个存在的连接但是未被开启（这是一个内部错误）。

SQLCODE值	SQLSTATE值	含义
-230 (ECPG_INVALID_STMT)	SQLSTATE 26000	尝试使用的语句还没有被准备好。
-239 (ECPG_INFORMIX_DUPLICATE_KEY)	SQLSTATE 23505	重复键错误，违背唯一约束。
-240 (ECPG_UNKNOWN_DESCRIPTOR)	SQLSTATE 33000	没有找到指定的描述符，尝试使用的语句未准备好。
-241 (ECPG_INVALID_DESCRIPTOR_INDEX)	SQLSTATE 07009	指定的描述符超出范围。
-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)	SQLSTATE YE000	请求了一个非法的描述符（这是一个内部错误）。
-243 (ECPG_VAR_NOT_NUMERIC)	SQLSTATE 07006	在执行一个动态语句期间，数据库返回了一个numeric值而宿主变量不是numeric类型的。
-244 (ECPG_VAR_NOT_CHAR)	SQLSTATE 07006	在执行一个动态语句期间，数据库返回了一个非numeric值而宿主变量是numeric类型的。
-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)	SQLSTATE 21000	子查询的结果不是单一行。
-400 (ECPG_PGSQL)	-	SQL服务器导致了某个错误。该消息包含来自SQL服务器的错误消息。
-401 (ECPG_TRANS)	SQLSTATE 08007	SQL服务器通知不能启动、提交或回滚事务。
-402 (ECPG_CONNECT)	SQLSTATE 08001	无法建立数据库连接。
-403 (ECPG_DUPLICATE_KEY)	SQLSTATE 23505	重复键错误，违背唯一约束。
-404 (ECPG_SUBSELECT_NOT_ONE)	SQLSTATE 21000	子查询的结果不是单一行。
-602 (ECPG_WARNING_UNKNOWN_PORTAL)	SQLSTATE 34000	指定了一个非法的游标名。

SQLCODE值	SQLSTATE值	含义
-603 (ECPG_WARNING_IN_TRANSACTION)	SQLSTATE 25001	事务正在进行。
-604 (ECPG_WARNING_NO_TRANSACTION)	SQLSTATE 25P01	没有活动（正在进行）的事务。
-605 (ECPG_WARNING_PORTAL_EXISTS)	SQLSTATE 42P03	指定了一个现有的游标名。

5.10.4.5 使用库函数

ECPGdebug(int on, FILE *stream)

若函数第一个参数为非0，则开启调试日志，第二个参数表示要打印日志的标准输出流。调试日志在标准输出流上执行，日志包含所有输入的SQL语句以及来自GaussDB服务器的结果。

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqlca.h"

int main()
{
    ECPGdebug(1, stderr);

    /* 需提前创建testdb */
    EXEC SQL CONNECT TO testdb;
    EXEC SQL SET AUTOCOMMIT TO ON;
    EXEC SQL CREATE TABLE T1(a int);
    return (0);
}
```

运行结果如下（[23893]为线程ID，随机生成）：

```
[23893]: ECPGdebug: set to 1
[23893]: ECPGconnect: opening database testdb on <DEFAULT> port <DEFAULT>
[23893]: ECPGsetcommit on line 12: action "on"; connection "testdb"
[23893]: ecpg_execute on line 13: query: create table T1 ( a int ); with 0 parameter(s) on connection testdb
[23893]: ecpg_execute on line 13: using PQexec
[23893]: ecpg_execute on line 13: OK: CREATE TABLE
```

ECPGget_PGconn(const char *connection_name)

返回由给定名称标识的数据库连接句柄。如果connection_name设置为NULL，则返回当前连接句柄。如果没有连接句柄可以被识别，则该函数返回NULL。

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



```
#include "sqlca.h"

int main()
{
    ECPGdebug(1, stderr);

    /* 需提前创建testdb数据库 */
    EXEC SQL CONNECT TO testdb as con1;
    EXEC SQL SET AUTOCOMMIT TO ON;
    EXEC SQL DROP TABLE IF EXISTS T1;
    PGconn *conn;
    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);
    conn = ECPGget_PGconn(NULL);
    printf("conn = %p\n", conn);
    EXEC SQL CREATE TABLE T1(a int);
    return (0);
}
```

运行结果如下（[29709]为线程ID，随机生成。连接生成地址随机，运行结果不固定。）：

```
[29709]: ECPGdebug: set to 1
[29709]: ECPGconnect: opening database testdb on <DEFAULT> port <DEFAULT>
[29709]: ECPGsetcommit on line 12: action "on"; connection "con1"
[29709]: ecpg_execute on line 13: query: drop table if exists T1; with 0 parameter(s) on connection con1
[29709]: ecpg_execute on line 13: using PQexec
[29709]: ecpg_execute on line 13: OK: DROP TABLE
conn = 0x14daf70
conn = 0x14daf70
[29709]: ecpg_execute on line 19: query: create table T1 ( a int ); with 0 parameter(s) on connection con1
[29709]: ecpg_execute on line 19: using PQexec
[29709]: ecpg_execute on line 19: OK: CREATE TABLE
```

ECPGtransactionStatus(const char *connection_name)

返回connection_name连接的当前事务状态。可能的返回值包括：

```
PQTRANS_IDLE, /* connection idle, 连接空闲 */
PQTRANS_ACTIVE, /* command in progress, 命令正在执行 */
PQTRANS_INTRANS, /* idle, within transaction block, 空闲, 在事务块里 */
PQTRANS_INERROR, /* idle, within failed transaction, 空闲, 在失败的事务里 */
PQTRANS_UNKNOWN /* cannot determine status, 未知状态 */
```

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqlca.h"

int main()
{
    ECPGdebug(1, stderr);

    /* 需提前创建testdb数据库 */
    EXEC SQL CONNECT TO testdb as con1;
    EXEC SQL DROP TABLE IF EXISTS T1;
    int a = ECPGtransactionStatus("con1");
    printf("%d\n", a);
    EXEC SQL CREATE TABLE T1(a int);
    EXEC SQL COMMIT;
    return (0);
}
```

运行结果如下（[44153]为线程ID，随机生成）：

```
[44153]: ECPGdebug: set to 1
[44153]: ECPGconnect: opening database testdb on <DEFAULT> port <DEFAULT>
[44153]: ecpg_execute on line 12: query: drop table if exists T1; with 0 parameter(s) on connection con1
```

```
[44153]: ecpg_execute on line 12: using PQexec
[44153]: ecpg_execute on line 12: OK: DROP TABLE
2
[44153]: ecpg_execute on line 15: query: create table T1 ( a int ); with 0 parameter(s) on connection con1
[44153]: ecpg_execute on line 15: using PQexec
[44153]: ecpg_execute on line 15: OK: CREATE TABLE
[44153]: ECPGtrans on line 16: action "commit"; connection "con1"
```

ECPGfree_auto_mem()

释放为输出型宿主变量申请的所有内存，在程序结束时调用（return\exit）。

示例如下：

前提条件：需要提前创建testdb数据库。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqlca.h"

int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    int *ip1=0;
    char **cp2=0;
    int *ipointer1=0;
    int *ipointer2=0;
    int column;
    EXEC SQL END DECLARE SECTION;
    int i;

    EXEC SQL WHENEVER SQLERROR DO sqlprint();
    EXEC SQL CONNECT TO testdb;

    /* 需提前创建testdb数据库 */
    EXEC SQL CREATE TABLE test (a int, b text);
    EXEC SQL INSERT INTO test VALUES (1, 'one');
    EXEC SQL INSERT INTO test VALUES (2, 'two');
    EXEC SQL INSERT INTO test VALUES (NULL, 'three');
    EXEC SQL INSERT INTO test VALUES (NULL, NULL);

    EXEC SQL ALLOCATE DESCRIPTOR mydesc;
    EXEC SQL SELECT * INTO SQL DESCRIPTOR mydesc FROM test;
    EXEC SQL GET DESCRIPTOR mydesc :column=COUNT;
    EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :ip1=DATA, :ipointer1=INDICATOR;
    EXEC SQL GET DESCRIPTOR mydesc VALUE 2 :cp2=DATA, :ipointer2=INDICATOR;

    printf("Result (%d columns):\n", column);
    for (i=0; i < sqlca.sqlerrd[2]; ++i)
    {
        if (ipointer1[i]) printf("NULL, ");
        else printf("%d, ", ip1[i]);

        if (ipointer2[i]) printf("NULL, ");
        else printf("'%s', ", cp2[i]);
        printf("\n");
    }
    ECPGfree_auto_mem();
    printf("\n");

    EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
    EXEC SQL ROLLBACK;
    EXEC SQL DISCONNECT;
    return 0;
}
```

运行结果如下：

```
Result (2 columns):
1, 'one',
2, 'two',
NULL, 'three',
NULL, NULL,
```

5.10.4.6 SQL 描述符区域

5.10.4.6.1 概述

SQL描述符区域是一种处理SELECT、FETCH或者DESCRIBE语句结果的高级方法。

SQL描述符区域把一行数据里的数据和元数据项组合到一个数据结构中。ecpg提供了两种使用描述符区域的方法：命名SQL描述符区域和SQLDA。

5.10.4.6.2 命名 SQL 描述符区域

一个命名SQL描述符区域由一个头部以及一个或多个条目描述符区域构成。头部包含与整个描述区域相关的信息，而条目描述符区域则描述结果行中的某一列。

- 在使用SQL描述符区域之前，需要分配一个SQL描述符区域：

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

- 当不再需要这个描述符区域时，应及时释放：

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

- 要使用一个描述符区域，需要使用INTO子句声明：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集为空，该描述符区域仍会包含查询的元数据。

- 对于还没有执行的预备查询，可以使用DESCRIBE得到其结果集的元数据：

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

在DESCRIBE和FETCH语句中，INTO和USING关键词的使用相似：它们产生结果集以及一个描述符区域的元数据。

- 从头部检索一个描述符区域的值并且将其存储到一个宿主变量中：

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

- 当前只定义了一个头部描述符区域COUNT，它存放描述符区域的条目（即结果集中包含多少列），宿主变量为一个整数类型，需从条目描述符区域中得到一个具体值：

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

num可以是一个字符整数或者一个包含整数的宿主变量。可能的类型如下：

- CARDINALITY（整数）：结果集中的行数
- DATA：实际的数据项（此范围的实际数据类型取决于查询）
- DATETIME_INTERVAL_CODE（整数）：当TYPE是9时，DATETIME_INTERVAL_CODE将具有以下值之一：1表示DATE，2表示TIME，3表示TIMESTAMP，4表示TIME WITH TIME ZONE，5表示TIMESTAMP WITH TIME ZONE
- INDICATOR（整数）：指示符（表示一个空值或者一个值截断）
- LENGTH（整数）：以字符计的数据长度
- NAME(字符串)：列名

- OCTET_LENGTH（整数）：以字节计的数据字符表达的长度。
- PRECISION（整数）：精度（用于类型numeric）
- RETURNED_LENGTH（整数）：以字符计的数据长度
- RETURNED_OCTET_LENGTH（整数）：以字节计的数据字符表达的长度
- SCALE（整数）：比例（用于类型numeric）
- TYPE（整数）：列的数据类型的数字编码
- 要检索字段数值并且把它存储到一个宿主变量里，使用如下命令：
EXEC SQL GET DESCRIPTOR mydesc VALUE num :hostvar = field
num可以是一个字符整数或者一个包含整数的宿主变量。可能的字段有：
 - DATA
 - 实际数据项（此字段的数据类型依赖于这个查询）
 - NAME(字符串)
 - 字段名称
- 手动建立一个描述符区域为一个查询或游标提供输入参数，使用如下命令：
EXEC SQL SET DESCRIPTOR name VALUE numfield = :hostvar;
- 在一个FETCH语句中检索多行记录且用数组类型的宿主变量来存储数据，示例如下：
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;
EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;

5.10.4.6.3 SQLDA

SQLDA（SQL Descriptor Area）是一个C语言结构体，用于动态处理SQL查询的结果集或输入参数。它允许程序在运行时处理未知数量和类型的列或参数。

```
EXEC SQL include sqllda.h;  
sqllda_t *mysqlda;  
EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

注意此处SQL关键词被省略了，[命名SQL描述符区域](#)章节中关于INTO和USING关键词的用例在一定条件下也适用。在一个DESCRIBE语句中，如果使用了INTO关键词，则DESCRIPTOR关键词可以省略。

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

SQLDA 相关结构体

SQLDA 的核心数据结构包括 sqllda_t、sqlvar_t 和 struct sqlname。

- sqllda_t 结构
sqllda_t 结构体用于存储查询结果集的整体信息。其定义如下：

```
struct sqllda_struct {  
    char sqldaid[8];           // SQLDA标识符，通常为"SQLDA"  
    long sqldabc;              // 已分配空间的大小（字节数）  
    short sqln;                // 输入参数的数量或输出结果集的列数  
    short sqld;                // 结果集中的列数  
    struct sqllda_struct *desc_next; // 指向下一个SQLDA（链表结构）  
    struct sqlvar_struct sqlvar[1]; // 结果集的列数组  
};  
typedef struct sqllda_struct sqllda_t;
```

字段解释：

- sqldaid：SQLDA的标识符，通常是固定字符串“SQLDA”。

- sqldbc: 已分配的空间大小，单位为字节。
- sqln: 输入参数的数量，或者在处理查询结果时，表示结果集中的列数。
- sqld: 结果集中的列数。
- desc_next: 用于链式存储多个结果集。
- sqlvar: 存储每个列的数据和元信息。

- sqlvar_t 结构

sqlvar_t 结构体保存单个列的值及其元数据（如类型和长度）。其定义如下：

```
struct sqlvar_struct {  
    short sqltype;           // 列的类型标识符  
    short sqllen;            // 列的二进制长度  
    char *sqldata;           // 指向列数据的指针  
    short *sqlind;           // 指向空指示符的指针  
    struct sqlname sqlname;   // 列名  
};  
typedef struct sqlvar_struct sqlvar_t;
```

字段解释：

- sqltype: 列的数据类型（如整数、字符等）。
- sqllen: 列的数据长度。
- sqldata: 指向存储数据的内存区域。
- sqlind: 指示该列是否为空，0表示非空，-1表示为空。
- sqlname: 列的名称。

- struct sqlname 结构

sqlname 结构体保存列的名称。其定义如下：

```
#define NAMEDATALEN 64  
struct sqlname {  
    short length;            // 列名的长度  
    char data[NAMEDATALEN];  // 列名数据  
};
```

字段解释：

- length: 列名的长度。
- data: 列名字符串。

SQLDA 检索结果集

通过SQLDA检索查询结果集的一般步骤：

- 步骤1** 声明一个sqllda_t结构来接收结果集。
- 步骤2** 执行FETCH/EXECUTE/DESCRIBE命令来处理一个已声明SQLDA的查询。
- 步骤3** 通过查看sqllda_t结构的成员sqln来检查结果集中记录的数量。
- 步骤4** 从sqllda_t结构体的成员sqlvar[0]、sqlvar[1]等中得到每一列的值。
- 步骤5** 沿着sqllda_t结构的成员desc_next指针到达下一行（sqllda_t）。
- 步骤6** 根据需要重复上述步骤。

----结束

示例如下：

```
/* 声明一个sqllda_t结构来接收结果集。*/  
sqllda_t *sqllda1;
```

```
/* 指定一个命令中的SQLDA。如下是一个FETCH命令的例子。*/
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

/* 运行一个循环顺着链表来检索行。*/
sqlda_t *cur_sqlda;
for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}

/* 在循环内部，运行另一个循环来检索行中每一列的数据（sqlvar_t结构）。*/
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen = v.sqllen;
    ...
}

/* 要得到一列的值，应检查sqlvar_t结构的成员sqltype的值。然后，根据列类型切换到一种合适的方法从sqlvar域
中复制数据到一个主变量。*/
char var_buf[1024];
switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqllen));
        break;

    case ECPGt_int:
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;
    ...
}
```

使用 SQLDA 传递查询参数

使用SQLDA传递输入参数给一个预备查询的一般步骤：

- 步骤1** 创建一个预备查询（预备语句）。
- 步骤2** 声明一个sqlda_t结构体作为SQLDA。
- 步骤3** 为SQLDA分配内存区域。
- 步骤4** 在分配好的内存中设置（复制）输入值。
- 步骤5** 打开一个在SQLDA上声明的游标。
- 步骤6** 处理查询结果。
- 步骤7** 释放内存。

----结束

示例如下：

```
/* 首先，创建一个预备语句。*/
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid = s.datid
AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE stmt1 FROM :query;
```

```
/* 为一个SQLDA分配内存，并且在sqlda_t结构的sqln成员变量中设置输入参数的数量。  
* 当预备查询要求两个或多个输入参数时，应用必须分配额外的内存空间，空间的大小为（参数数目 - 1） *  
sizeof(sqlvar_t)。  
* 该示例展示了为两个输入参数分配内存空间。  
*/  
sqlda_t *sqlda2;  
sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));  
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));  
sqlda2->sqln = 2; /* 输入变量的数目 */  
  
/* 内存分配之后，把参数值存储到sqlvar[]数组（当 SQLDA 在接收结果集时，这也是用来检索列值的数  
组）。  
在这个例子中，输入参数是"testdb"（字符串类型）和1（整数类型）。*/  
sqlda2->sqlvar[0].sqltype = ECPGt_char;  
sqlda2->sqlvar[0].sqldata = "testdb";  
sqlda2->sqlvar[0].sqln = 8;  
int intval = 1;  
sqlda2->sqlvar[1].sqltype = ECPGt_int;  
sqlda2->sqlvar[1].sqldata = (char *) &intval;  
sqlda2->sqlvar[1].sqln = sizeof(intval);  
  
/* 通过打开一个游标并且说明之前已经建立好的 SQLDA，输入参数被传递给预备语句。*/  
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;  
  
/* 最后，用完输入 SQLDA 后必须显式地释放已分配的内存空间，这与用于接收查询结果的 SQLDA 不同。*/  
free(sqlda2);
```

5.10.4.7 ecpg 与 Pro*C 兼容性对比

ecpg是GaussDB提供的一种用于C语言程序的嵌入式SQL预处理器，与ORA数据库Pro*C预处理器在编译执行命令、语法、嵌入式语句等行为和语义上存在差异。

ecpg与Pro*C的相关使用差异对比：

- 目前ecpg不支持EXEC SQL CONTEXT ALLOCATE、EXEC SQL CONTEXT USE、EXEC SQL CONTEXT FREE。

说明

ecpg当前不支持CONTEXT申请、使用、释放操作，ecpg有独立的内存管理机制。多线程模式下，ecpg在每个线程中独立地建立连接、执行SQL语句以及相关资源的释放。这一使用方式与Pro*C多线程模式下每个线程各自进行CONTEXT相关申请与释放的处理逻辑一致。

- 目前ecpg不支持EXEC SQL COMMIT WORK RELEASE。

说明

在ecpg中，当业务语句执行COMMIT之后，并没有RELEASE选项，需要通过调用EXEC SQL DISCONNECT、EXEC SQL CLOSE等命令来实现相关资源的释放。Pro*C中EXEC SQL COMMIT带有RELEASE选项。用于释放程序持有的所有连接、游标等资源信息。

- 目前ecpg不支持EXEC SQL ENABLE THREAD。

说明

ecpg编译选项中开启宏定义，在main函数的.pgc文件中定义（define）ENABLE_THREAD_SAFETY。

- 目前ecpg不支持存储过程、Package、匿名块、闪回等特性语法。

5.10.5 日志输出相关参数介绍

用户可以根据需要，通过修改实例数据目录下的gaussdb.conf文件中特定的配置参数来控制日志的输出，从而更好地了解数据库的运行状态。

可调整的配置参数请参见[表5-147](#)。

表 5-147 配置参数

参数名称	描述	取值范围	备注
client_min_messages	配置发送到客户端信息的级别。	• debug5 • debug4 • debug3 • debug2 • debug1 • log • info • notice • warning • error • fatal • panic 默认值：notice	级别不低于设置值的所有消息会被发送。 通常，设置的级别越低，发送给客户端的消息就越多。 对于客户端日志消息：如果 client_min_messages 参数值设置为fatal或panic，该参数的实际值会被强制指定为error。 client_min_messages无论设置为哪个级别，info级别消息都会发送到客户端。
log_min_messages	配置写到服务器日志里信息的级别。	• debug5 • debug4 • debug3 • debug2 • debug1 • log • info • notice • warning • error • fatal • panic 默认值：warning	级别不低于设置值的所有消息会被写入。 通常，设置的级别越低，写入服务器日志的消息就越多。 对于服务器日志消息：当 log_min_messages 设置级别不高于error时，log级别消息都会写入服务器日志。

参数名称	描述	取值范围	备注
log_min_error_statement	配置写到服务器日志中错误SQL语句的级别。	• debug5 • debug4 • debug3 • debug2 • debug1 • log • info • notice • warning • error • fatal • panic 默认值：error	级别不低于设置值的消息对应的所有SQL语句会被写入。 通常，设置的级别越低，写入服务器日志的SQL语句就越多。 对于服务器日志SQL消息：当设置级别不高于error时，log级别消息对应的SQL语句都会写入服务器日志。 只有系统管理员可以修改该参数。
log_min_duration_statement	配置语句执行持续的最短时间。如果某个语句的持续时间大于或者等于设置的毫秒数，则会在日志中记录该语句及其持续时间。打开这个选项可以方便地跟踪需要优化的查询。	INT类型。 默认值：-1。 单位：毫秒。	设置为-1表示关闭这个功能。 只有系统管理员可以修改该参数。
log_connections/ log_disconnections	配置是否在每次会话连接或结束时向服务器日志里打印一条信息。	• on：每次会话连接或结束时向日志里打印一条信息。 • off：每次会话连接或结束时不向日志里打印信息。 默认值：off。	-
log_duration	配置是否记录每个已完成语句的持续时间。	• on：记录每个已完成语句的持续时间。 • off：不记录已完成语句的持续时间。 默认值：off	只有系统管理员可以修改该参数。

参数名称	描述	取值范围	备注
log_statement	配置日志中记录哪些SQL语句。	• none：不记录任何SQL语句。 • ddl：记录数据定义语句。 • mod：记录数据定义语句和数据操作语句。 • all：记录所有语句。 默认值： none	只有系统管理员可以修改该参数。
log_hostname	配置是否记录主机名。	• on：记录主机名。 • off：不记录主机名。 默认值： off	缺省时，连接日志只记录所连接主机的IP地址。打开这个选项会同时记录主机名。 该参数同时影响查看审计结果、系统视图PG_STAT_ACTIVITY和GUC参数log_line_prefix。

上表有关参数级别的说明请参见[表5-148](#)。

表 5-148 日志级别参数说明

级别	说明
DEBUG[1-5]	提供开发人员使用的信息。5级为最高级别，以此类推，1级为最低级别。
INFO	提供用户隐含要求的信息。如在VACUUM VERBOSE过程中的信息。
NOTICE	提供可能对用户有用的信息。如长标识符的截断，作为主键一部分创建的索引。
WARNING	提供给用户的警告。如在事务块范围之外的COMMIT。
ERROR	报告导致当前命令退出的错误。
LOG	报告一些管理员感兴趣的信息。如检查点活跃性。
FATAL	报告导致当前会话终止的原因。
PANIC	报告导致所有会话退出的原因。

6 用户自定义函数

集群启动时，除了启动CN、DN之外，还会启动UDF master进程。当需要执行fenced模式的UDF时，UDF master进程会fork出UDF worker进程，UDF worker进程执行fenced模式的UDF。

6.1 PL/SQL 语言函数

PL/SQL是一种可载入的过程语言。

用PL/SQL创建的函数可以被用在任何可以使用内建函数的地方。例如，可用于创建复杂条件的计算函数、定义操作符以及索引表达式。

SQL被大多数数据库用作查询语言。它是可移植的并且容易学习。但是每一个SQL语句必须由数据库服务器单独执行。

这意味着客户端应用对于每一个查询都要执行以下过程：发送查询到数据库服务器、等待查询被接收、接收并处理结果、进行相关计算、然后发送更多查询给服务器。如果客户端和数据库服务器不在同一台机器上，那么这个过程还会引起进程间通信并且将带来网络负担。

通过PL/SQL，可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力并且使SQL更易用，同时能节省客户端/服务器通信开销。

- 客户端和服务端之间的额外往返通信被消除。
- 客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送。
- 多轮的查询解析可以被避免。

PL/SQL可以使用SQL中所有的数据类型、操作符和函数，应用PL/SQL创建函数的语法为CREATE FUNCTION。

PL/SQL是一种可载入的过程语言，其应用方法与存储过程相似，但存储过程无返回值，PL/SQL语言函数有返回值。

非透明多写特性下，语法和存储过程详细内容请参见《参考》中“SQL参考 > SQL语法”和“存储过程”章节。

非透明多写特性下，XML类型数据支持作为自定义函数的入参、出参、自定义变量和返回值。

7 特性使用指导

7.1 自治事务

自治事务（Autonomous Transaction），在主事务执行过程中新启的独立的事务。自治事务的提交和回滚不会影响主事务已提交的数据，同时自治事务也不受主事务影响。

自治事务在存储过程、函数和匿名块中定义，用PRAGMA AUTONOMOUS_TRANSACTION关键字来声明。

7.1.1 规格约束

- 非透明多写场景下，自治事务执行时，将会在后台启动自治事务session，可以通过max_concurrent_autonomous_transactions设置自治事务执行的最大并行数量，取值范围：0~10000，当max_concurrent_autonomous_transactions参数设置为0时，自治事务将无法执行。
- 在非透明多写场景下，自治事务新启session后，将使用默认session参数，不共享主事务session下对象（包括session级别变量、本地临时变量、全局临时表的数据等）。
- 非透明多写场景下，自治事务新启session后，将主事务session设置的如下GUC参数：session_max_dynamic_memory、behavior_compat_options和nls_date_format同步至自治事务session，GUC参数说明详情请参见《参考》中“数据库运行参数说明 > GUC参数说明”章节。
- 非透明多写场景下，自治事务理论上限为10000，实际上限为动态值，参考GUC参数max_concurrent_autonomous_transactions描述。
- 非透明多写场景下，自治事务受通信缓冲区影响，返回给客户端的信息大小受限于通信缓冲区长度，超过通信缓冲区长度时报错。
- 非透明多写场景下，自治事务执行期间，主事务会给自治事务加锁，异常情况导致此锁残留的场景下，需要依赖gs_clean清理残留锁，清理周期由GUC参数gs_clean_timeout控制，默认值为60秒。自治事务的锁不受lock timeout影响，锁超时时间为2147483s，自治事务执行超过此时间会报错锁超时。
- 非透明多写场景下，自治事务设置建立连接超时时间5s，建立连接尝试5次。建立连接期间不立即响应信号，每次建立连接前检查信号。高并发、高CPU、高内存，以及线程池扩容场景下可能存在超时报错现象。

- 在PACKAGE SPECIFICATION或PACKAGE BODY SPECIFICATION中声明自治事务PRAGMA AUTONOMOUS_TRANSACTION语法，可成功创建PACKAGE，但自治事务不生效。

- 触发器函数不支持自治事务。

```
gaussdb=# CREATE TABLE test_trigger_des_tbl(id1 int, id2 int, id3 int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

```
gaussdb=# CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
$$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO test_trigger_des_tbl VALUES(new.id1, new.id2, new.id3);
    RETURN new;END$$ LANGUAGE plpgsql;
ERROR: Triggers do not support autonomous transactions
DETAIL: N/A
```

```
gaussdb=# DROP TABLE test_trigger_des_tbl;
DROP TABLE
```

- 自治事务不支持非顶层匿名块调用（仅支持顶层自治事务，包括存储过程、函数、匿名块）。

```
gaussdb=# CREATE TABLE t1(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.
CREATE TABLE
```

```
gaussdb=# DECLARE
--PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    DBE_OUTPUT.PRINT_LINE('JUST USE CALL.');
```

```
INSERT INTO t1 VALUES(1,'CAN YOU ROLLBACK!');
```

```
END;
```

```
INSERT INTO t1 VALUES(2,'I WILL ROLLBACK!');
```

```
ROLLBACK;
```

```
END;
```

```
/
```

```
JUST USE CALL.
```

```
ANONYMOUS BLOCK EXECUTE
```

```
gaussdb=# SELECT * FROM t1;
 a | b
---+---
(0 rows)
```

```
gaussdb=# DROP TABLE t1;
DROP TABLE
```

- 自治事务仅支持PROCEDURE OUT参数传递ref cursor参数，不支持IN、INOUT以及FUNCTION传递ref cursor参数；非透明多写场景下，仅支持在存储过程中使用自治事务OUT参数传递ref cursor参数，不支持在客户端（如gsqsl、JDBC）中直接调用。

--创建表

```
gaussdb=# CREATE TABLE sections(section_id INT);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'section_id' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
```

CREATE TABLE

```
gaussdb=# INSERT INTO sections VALUES(1);
```

```
INSERT 0 1
```

```
gaussdb=# INSERT INTO sections VALUES(1);
```

```
INSERT 0 1
```

```
gaussdb=# INSERT INTO sections VALUES(1);
INSERT 0 1
gaussdb=# INSERT INTO sections VALUES(1);
INSERT 0 1
```

a. **PROCEDURE OUT出参传递ref cursor（支持）。**

```
gaussdb=# CREATE OR REPLACE PROCEDURE proc_sys_ref(OUT c1 REFCURSOR)
IS
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    OPEN c1 FOR SELECT section_id FROM sections ORDER BY section_id;

END;
/
CREATE PROCEDURE

gaussdb=# CREATE OR REPLACE PROCEDURE proc_sys_call() AS
DECLARE
    c1 SYS_REFCURSOR;
    temp NUMBER(4);
BEGIN
    proc_sys_ref(c1);
    IF c1%ISOPEN THEN
        RAISE NOTICE '%','OK';
    END IF;

    LOOP
        FETCH c1 INTO temp;
        RAISE NOTICE '%',c1%ROWCOUNT;
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
END;
/
CREATE PROCEDURE

gaussdb=# CALL proc_sys_call();
NOTICE: OK
NOTICE: 1
NOTICE: 2
NOTICE: 3
NOTICE: 4
NOTICE: 4
proc_sys_call
-----

(1 row)
```

b. **PROCEDURE IN或INOUT出参传递REF CURSOR（不支持）。**

```
gaussdb=# CREATE OR REPLACE PROCEDURE proc_sys_ref(IN c1 REFCURSOR)
IS
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    temp NUMBER(4);
BEGIN
    IF c1%ISOPEN THEN
        RAISE NOTICE '%','OK';
    END IF;

    LOOP
        FETCH c1 INTO temp;
        RAISE NOTICE '%',c1%ROWCOUNT;
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
END;
/
CREATE PROCEDURE

gaussdb=# CREATE OR REPLACE PROCEDURE proc_sys_call() AS
DECLARE
```

```
c1 SYS_REFCURSOR;
temp NUMBER(4);
BEGIN
  OPEN c1 FOR SELECT section_id FROM sections ORDER BY section_id;
  proc_sys_ref(c1);
END;
/
CREATE PROCEDURE

gaussdb=# CALL proc_sys_call();
ERROR: Unsupported: ref_cursor parameter is not supported for autonomous transactions.
CONTEXT: SQL statement "CALL proc_sys_ref(c1)"
PL/pgSQL function proc_sys_call() line 7 at PERFORM
```

c. **FUNCTION RETURN**传递ref cursor（不支持）。

```
gaussdb=# DROP PROCEDURE IF EXISTS proc_sys_ref;
gaussdb=# CREATE OR REPLACE function proc_sys_ref() RETURN SYS_REFCURSOR IS
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  c1 SYS_REFCURSOR;
BEGIN
  OPEN c1 FOR SELECT section_id FROM sections ORDER BY section_id;
  return c1;
END;
/
ERROR: Autonomous function do not support ref cursor as return types or out, inout arguments.
DETAIL: N/A
```

d. **FUNCTION OUT**出参传递ref cursor（不支持）。

```
gaussdb=# CREATE OR REPLACE FUNCTION proc_sys_ref(c1 OUT SYS_REFCURSOR)
RETURN SYS_REFCURSOR
IS
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  OPEN c1 FOR SELECT section_id FROM sections ORDER BY section_id;
  RETURN 1;
END;
/
ERROR: Autonomous function do not support ref cursor as return types or out, inout arguments.
DETAIL: N/A
```

e. **非透明多写场景下，客户端中（例如gsq、JDBC）直接调用带ref cursor出参的自治事务PROCEDURE（不支持，此时无法读取cursor数据）。**

```
gaussdb=# CREATE OR REPLACE PROCEDURE proc_sys_ref(OUT c1 REFCURSOR)
IS
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  OPEN c1 FOR SELECT section_id FROM sections ORDER BY section_id;
END;
/
CREATE PROCEDURE
gaussdb=# begin;
BEGIN
gaussdb=# call proc_sys_ref(null);
      c1
-----
<unnamed portal 1>
(1 row)

gaussdb=# fetch "<unnamed portal 1>";
ERROR: cursor "<unnamed portal 1>" does not exist
gaussdb=# end;
ROLLBACK
gaussdb=# DROP PROCEDURE proc_sys_ref;
DROP PROCEDURE
--删除表
gaussdb=# DROP TABLE sections;
DROP TABLE
```

- 非透明多写场景下，自治事务不支持下推（IMMUTABLE、STABLE类型）。

```
gaussdb=# CREATE OR REPLACE PROCEDURE autonomous_test_in_p_116(num1 INT )
IMMUTABLE
AS
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
PERFORM pg_sleep(1);
END;
/
ERROR: Autonomous transactions do not support STABLE/IMMUTABLE.
DETAIL: Please remove stable/immutable.

gaussdb=# CREATE OR REPLACE PROCEDURE autonomous_test_in_p_117(num1 INT ) STABLE AS
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
PERFORM pg_sleep(1);
END;
/
ERROR: Autonomous transactions do not support STABLE/IMMUTABLE.
DETAIL: Please remove stable/immutable.
```
- 非透明多写场景下，不支持检测。

```
gaussdb=# CREATE TABLE test_lock (id INT,a DATE);
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
gaussdb=# INSERT INTO test_lock VALUES (10,SYSDATE),(11,SYSDATE),(12,SYSDATE);
INSERT 0 3
gaussdb=# CREATE OR REPLACE FUNCTION autonomous_test_lock(num1 INT,num2 INT) RETURNS
INTEGER LANGUAGE plpgsql AS $$
DECLARE num3 INT := 4;
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
UPDATE test_lock SET a=SYSDATE WHERE id =11;
RETURN num1+num2+num3;
END;
$$;
CREATE FUNCTION

gaussdb=# START TRANSACTION;
START TRANSACTION
gaussdb=# UPDATE test_lock SET a=SYSDATE WHERE id =11;
UPDATE 1
gaussdb=# CALL autonomous_test_lock(1,1);
ERROR: ERROR: Lock wait timeout: thread 139874535470848 on node datanode1 waiting for
ShareLock on transaction 16214 after 120000.124 ms
DETAIL: blocked by hold lock thread 139874577413888, statement <UPDATE test_lock SET
a=SYSDATE WHERE id =11;>, hold lockmode ExclusiveLock.
CONTEXT: SQL statement "UPDATE test_lock SET a=SYSDATE WHERE id =11"
PL/SQL function autonomous_test_lock(integer,integer) line 5 at SQL statement
referenced column: autonomous_test_lock

gaussdb=# END;
ROLLBACK
gaussdb=# DROP TABLE test_lock;
DROP TABLE
```
- 自治事务函数不支持直接返回record类型或者out出参和record类型同时返回。

```
gaussdb=# CREATE OR REPLACE FUNCTION auto_func() RETURN RECORD
AS
DECLARE
TYPE rec_type IS RECORD(c1 INT, c2 INT);
r rec_type;
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
r.c1:=101;
r.c2:=201;
RETURN r;
```



```
END;  
/  
CREATE FUNCTION  
  
gaussdb=# SELECT auto_func();  
ERROR: unrecognized return type for PLSQL function.  
CONTEXT: referenced column: auto_func
```

- 不支持修改自治事务的隔离级别。

```
gaussdb=# CREATE OR REPLACE PROCEDURE auto_func(r INT)  
AS  
DECLARE  
  a INT;  
  PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
  a:=r;  
END;  
/  
CREATE FUNCTION  
gaussdb=# call auto_func(1);  
ERROR: ERROR: SET TRANSACTION ISOLATION LEVEL must be called before any query  
CONTEXT: SQL statement "SET TRANSACTION ISOLATION LEVEL SERIALIZABLE"  
PL/SQL function auto_func(integer) line 6 at SQL statement  
referenced column: auto_func
```

- 不支持自治事务返回集合类型（setof）。

```
gaussdb=# CREATE OR REPLACE FUNCTION test_set() RETURN SETOF INT  
AS  
DECLARE  
  PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
  RETURN QUERY (SELECT unnest(ARRAY[ARRAY[1, 2], ARRAY[3, 4]]));  
END;  
/  
ERROR: Autonomous transactions do not support RETURN SETOF.  
DETAIL: N/A
```

7.1.2 存储过程支持自治事务

自治事务可以在存储过程中定义，标识符为PRAGMA

AUTONOMOUS_TRANSACTION，其余语法与创建存储过程语法相同，非透明多写场景请参见《参考》中“SQL参考 > SQL语法 > C > CREATE PROCEDURE”章节，示例如下。

```
--建表  
gaussdb=# CREATE TABLE t2(a INT, b INT);  
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.  
CREATE TABLE  
gaussdb=# INSERT INTO t2 VALUES(1,2);  
INSERT 0 1  
gaussdb=# SELECT * FROM t2;  
 a | b  
---+---  
 1 | 2  
(1 row)  
  
--创建包含自治事务的存储过程  
gaussdb=# CREATE OR REPLACE PROCEDURE autonomous_4(a INT, b INT) AS  
DECLARE  
  num3 INT := a;  
  num4 INT := b;  
  PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
  INSERT INTO t2 VALUES(num3, num4);  
  DBE_OUTPUT.PRINT_LINE('JUST USE CALL.');
```

```
/
CREATE PROCEDURE

--创建调用自治事务存储过程的普通存储过程
gaussdb=# CREATE OR REPLACE PROCEDURE autonomous_5(a INT, b INT) AS
DECLARE
BEGIN
    DBE_OUTPUT.PRINT_LINE('JUST NO USE CALL.');
```

a	b
1	2
11	22

```
    INSERT INTO t2 VALUES(666, 666);
    autonomous_4(a,b);
    ROLLBACK;
END;
/
CREATE PROCEDURE

--调用普通存储过程
gaussdb=# SELECT autonomous_5(11,22);
JUST NO USE CALL.
JUST USE CALL.
autonomous_5
-----

(1 row)

--查看表结果
gaussdb=# SELECT * FROM t2 ORDER BY a;
a | b
---+---
1 | 2
11 | 22
(2 rows)

gaussdb=# DROP TABLE t2;
DROP TABLE
```

上述例子，最后在回滚的事务块中执行包含自治事务的存储过程，直接说明了自治事务的特性，即主事务的回滚不会影响自治事务已经提交的内容。

7.1.3 匿名块支持自治事务

自治事务可以在匿名块中定义，标识符为PRAGMA AUTONOMOUS_TRANSACTION，其余语法与创建匿名块语法相同，非透明多写场景请参见《参考》中“存储过程 > 声明语法 > 匿名块”章节，示例如下。

```
gaussdb=# CREATE TABLE t1(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.
CREATE TABLE

gaussdb=# START TRANSACTION;
START TRANSACTION
gaussdb=# DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    DBE_OUTPUT.PRINT_LINE('JUST USE CALL.');
```

a	b
1	YOU WILL ROLLBACK!

```
    INSERT INTO t1 VALUES(1,'YOU ARE SO CUTE,WILL COMMIT!');
END;
/
JUST USE CALL.
ANONYMOUS BLOCK EXECUTE

gaussdb=# INSERT INTO t1 VALUES(1,'YOU WILL ROLLBACK!');
INSERT 0 1
gaussdb=# ROLLBACK;
ROLLBACK
gaussdb=# SELECT * FROM t1;
a | b
---+---
1 | YOU WILL ROLLBACK!
```

```
1 | YOU ARE SO CUTE,WILL COMMIT!  
(1 row)
```

```
gaussdb=# DROP TABLE t1;  
DROP TABLE
```

上述例子，最后在回滚的事务块前执行包含自治事务的匿名块，直接说明了自治事务的特性，即主事务的回滚不会影响自治事务已经提交的内容。

7.1.4 函数支持自治事务

自治事务可以在函数中定义，标识符为PRAGMA AUTONOMOUS_TRANSACTION，其余语法与函数语法相同，非透明多写场景请参见《参考》中“SQL参考 > SQL语法 > C > CREATE FUNCTION”章节，示例如下。

```
gaussdb=# CREATE TABLE t4(a INT, b INT, c 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.  
CREATE TABLE
```

```
gaussdb=# CREATE OR REPLACE FUNCTION autonomous_32(a INT ,b INT ,c TEXT) RETURN INT AS  
DECLARE  
    PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
    INSERT INTO t4 VALUES(a, b, c);  
    RETURN 1;  
END;
```

```
/  
CREATE FUNCTION  
gaussdb=# CREATE OR REPLACE FUNCTION autonomous_33(num1 INT) RETURN INT AS  
DECLARE  
    num3 INT := 220;  
    tmp INT;  
    PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
    num3 := num3/num1;  
    RETURN num3;  
EXCEPTION  
    WHEN DIVISION_BY_ZERO THEN  
        SELECT autonomous_32(num3, num1, SQLERRM) INTO tmp;  
        ROLLBACK;  
        RETURN 0;  
END;
```

```
/  
CREATE FUNCTION  
gaussdb=# SELECT autonomous_33(0);  
autonomous_33
```

```
-----  
0  
(1 row)
```

```
gaussdb=# SELECT * FROM t4;  
a | b | c  
-----  
220 | 0 | division by zero  
(1 row)
```

```
gaussdb=# DROP TABLE t4;  
DROP TABLE
```

上述例子，最后在回滚的事务块中执行包含自治事务的函数，也能直接说明了自治事务的特性，即主事务的回滚不会影响自治事务已经提交的内容。

7.1.5 PACKAGE 支持自治事务

自治事务可以在package中的存储过程或者函数中定义，标识符为PRAGMA AUTONOMOUS_TRANSACTION，其余语法与创建package中存储过程或函数语法相

同，非透明多写场景请参见《参考》中“SQL参考 > SQL语法 > C > CREATE PACKAGE”章节，示例如下。

```
gaussdb=# create database test DBCOMPATIBILITY = 'ORA';
CREATE DATABASE
test=# \c test
--建表
test=# CREATE TABLE t2(a INT, b INT);
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.
CREATE TABLE
test=# INSERT INTO t2 VALUES(1,2);
INSERT 0 1
test=# SELECT * FROM t2;
 a | b
----+----
 1 | 2
(1 row)

--创建包含自治事务的PACKAGE中的存储过程和函数
test=# CREATE OR REPLACE PACKAGE autonomous_pkg AS
  PROCEDURE autonomous_4(a INT, b INT);
  FUNCTION autonomous_32(a INT ,b INT) RETURN INT;
END autonomous_pkg;
/
CREATE PACKAGE
test=# CREATE OR REPLACE PACKAGE BODY autonomous_pkg AS
PROCEDURE autonomous_4(a INT, b INT) AS
DECLARE
  num3 INT := a;
  num4 INT := b;
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO t2 VALUES(num3, num4);
END;
FUNCTION autonomous_32(a INT ,b INT) RETURN INT AS
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO t2 VALUES(a, b);
  RETURN 1;
END;
END autonomous_pkg;
/
CREATE PACKAGE BODY
--创建调用PACKAGE自治事务存储过程和函数的普通存储过程
test=# CREATE OR REPLACE PROCEDURE autonomous_5(a INT, b INT) AS
DECLARE
va INT;
BEGIN
  INSERT INTO t2 VALUES(666, 666);
  autonomous_pkg.autonomous_4(a,b);
  va := autonomous_pkg.autonomous_32(a + 1, b + 1);
  ROLLBACK;
END;
/
CREATE PROCEDURE
--调用普通存储过程
test=# SELECT autonomous_5(11,22);
autonomous_5
-----
(1 row)

--查看表结果
test=# SELECT * FROM t2 ORDER BY a;
 a | b
----+----
 1 | 2
11 | 22
```

```
12 | 23  
(3 rows)  
  
test=# DROP TABLE t2;  
DROP TABLE
```

上述例子，最后在回滚的事务块中执行包含PACKAGE自治事务的存储过程和函数，直接说明了自治事务的特性，即主事务的回滚不会影响自治事务已经提交的内容。

7.2 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。

7.2.1 全量物化视图

7.2.1.1 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《参考》中“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。

7.2.1.2 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7级锁（阻塞DML，DDL等），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。

7.2.1.3 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
```

7.2.2 增量物化视图

7.2.2.1 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。

7.2.2.2 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7级锁（阻塞DML，DDL等），若物化视图的定义为UNION ALL，需要注意业务逻辑，避免锁冲突产生。

7.2.2.3 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
```

7.3 设置密态等值查询

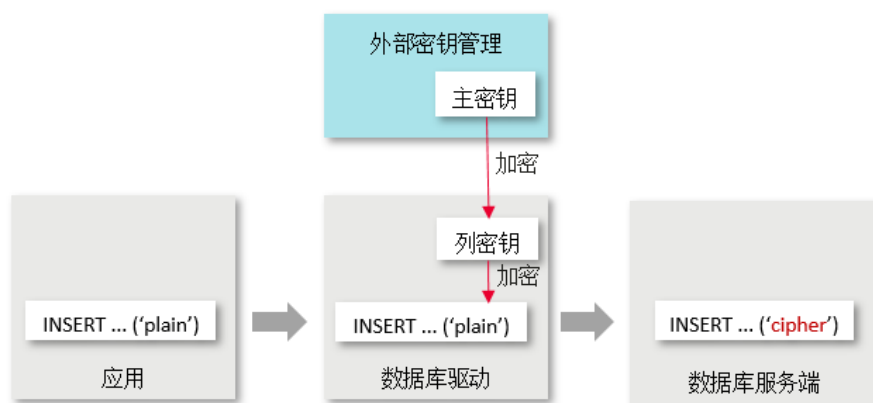
7.3.1 密态等值查询概述

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，其关系如图7-1所示：

图 7-1 加密模型图



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。

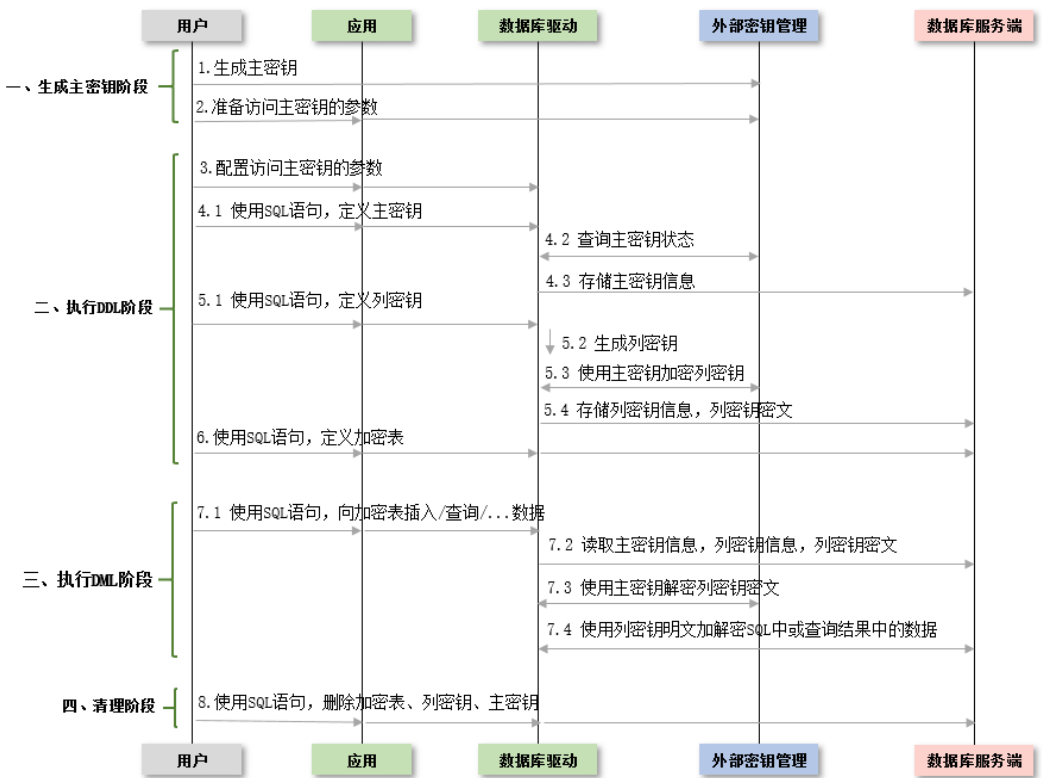
整体流程

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，本章节介绍整体流程。[使用gsql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，用户可直接执行包括但不限于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。

整体流程图如[图7-2](#)所示：

图 7-2 整体流程图



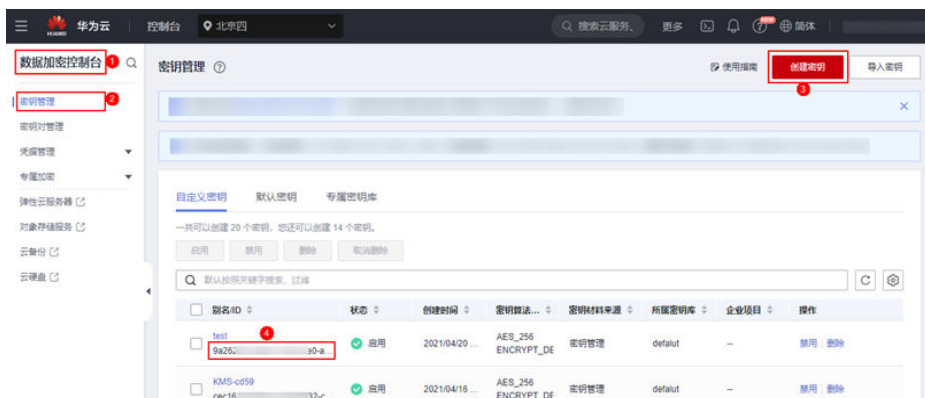
生成主密钥阶段

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，使用上一步新创建的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”，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。

7.3.2 使用 gsql 操作密态数据库

使用 gsql 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。

基于三未信安SJJ1994加密机密钥管理：

```
gaussdb=# \key_info keyType=sdf_kms,library_path={动态库的路径},library_version=SJJ1994,kek_algo=SGD_SM4_CBC,mac_algo=SGD_SM4_MAC,dkey_algo=SGD_SM4_CBC
```

说明

- 使用三未信安SJJ1994加密机前，需要正确配置ini文件，并且保证加密机能正常连接使用。
- library_path可以使用绝对路径或相对路径（相对于\$GAUSSHOME/lib）。
- kek_algo是密钥加密密钥（KEK）用于加密会话密钥的算法，允许配置算法：SGD_SM4_ECB（默认算法）、SGD_SM4_CBC、SGD_SM4_CTR、SGD_AES_ECB、SGD_AES_CBC、SGD_AES_CTR。
- dkey_algo是会话密钥用于加密数据密钥的算法，允许配置算法：SGD_SM4_CBC、SGD_SM4_CTR、SGD_AES_CBC、SGD_AES_CTR。
- mac_algo是会话密钥用于对数据密钥进行完整性校验的算法，允许配置算法：SGD_SM4_MAC、SGD_AES_MAC。
- library_version为可选配置，主要用于定制化对接，当前仅支持配置为SJJ1994。
- kek_algo要与创建CMK密钥时指定的KEY_PATH匹配，即均为AES系列算法或SM4系列算法。
- 目前支持gsq和JDBC驱动。

基于鲲鹏TEE内生密钥管理：

```
gaussdb=# \key_info keyType=sdf_kms,library_path={动态库的路径},mac_algo=SGD_SM4_MAC,dkey_algo=SGD_SM4_CBC
```

示例：\key_info KeyType=sdf_kms,library_path=./libsdf.so,mac_algo=SGD_SM4_MAC,dkey_algo=SGD_SM4_CBC

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：《参考》中“SQL参考 > SQL语法 > C > CREATE CLIENT MASTER KEY”章节。

```
gaussdb=# CREATE CLIENT MASTER KEY cmk2 WITH ( KEY_STORE = sdf_kms, KEY_PATH = '{密钥加密密钥ID}');
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：《参考》中“SQL参考 > SQL语法 > C > CREATE COLUMN ENCRYPTION KEY”章节。

```
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek1 WITH VALUES (CLIENT_MASTER_KEY = cmk1,
ALGORITHM = AES_256_GCM);
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek2 WITH VALUES (CLIENT_MASTER_KEY = cmk2,
ALGORITHM = SM4_HMAC_SM3);
```

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
gaussdb=# DROP COLUMN ENCRYPTION KEY cek2;
DROP COLUMN ENCRYPTION KEY

-- 删除主密钥
gaussdb=# DROP CLIENT MASTER KEY cmk1;
DROP CLIENT MASTER KEY
gaussdb=# DROP CLIENT MASTER KEY cmk2;
DROP CLIENT MASTER KEY
```

----结束

7.3.3 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[获取驱动jar包并配置JDK环境](#)章节。

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

📖 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opengaussjdbc-ce.jar、gsjdbc4.jar、opengaussjdbc.jar。

- opengaussjdbc-ce.jar：主类名为“com.huawei.opengauss.jdbc.Driver”，数据库连接的url前缀为“jdbc:opengauss”。目前仅支持EulerOS操作系统。此驱动包包含密态数据库需要加载的加解密相关依赖库，url前缀为“jdbc:opengauss”的密态数据库场景下推荐使用此驱动包。
- 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。例如：
export LD_LIBRARY_PATH=/tmp:\$LD_LIBRARY_PATH

- 使用gscejdbc.jar或opengaussjdbc-ce.jar驱动包时，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。

说明

使用gscejdbc.jar或opengaussjdbc-ce.jar时，jvm加载class文件需要依赖系统的libstdc++库，若开启密态则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。

- 使用gaussdbjdbc.jar、opengaussjdbc.jar或gsjdbc4.jar时，需要同时解压名为GaussDB-Kernel_数据库版本号_操作系统版本号_64bit_libpq.tar.gz的压缩包到指定目录，并将lib文件夹所在目录路径，添加至LD_LIBRARY_PATH环境变量中。

须知

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

执行 SQL 语句

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开发中与非密态场景操作一致的部分请参考[基于JDBC开发](#)。

● 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-ce.jar: 主类名为“com.huawei.opengauss.jdbc.Driver”，数据库连接的url前缀为“jdbc:opengauss”。  
// 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”。
```

```
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格式请参见《参考》中“SQL参考 > SQL语法 > C > CREATE CLIENT 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。详细语法请参见《参考》中“SQL参考 >
SQL语法 > C > CREATE COLUMN ENCRYPTION KEY”章节
int rc2 = stmt.executeUpdate("CREATE COLUMN ENCRYPTION KEY lmgCEK1 WITH VALUES
(CLIENT_MASTER_KEY = lmgCMK1, 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 = lmgCEK1, encryption_type = DETERMINISTIC),credit_card
varchar(19) encrypted with (column_encryption_key = lmgCEK1, 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 lmgCEK1;");
// 删除客户端主密钥
int rc7 = stmt.executeUpdate("DROP CLIENT MASTER KEY IF EXISTS lmgCMK1;");
// 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 7-1 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 7-2 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);
// 检查返回结果类解密成功与否，失败可获取报错信息，成功可获得明文及长度和原始数据类型
if (decrypt_res.isFailed()) {
    System.out.println(String.format("%s\n", decrypt_res.getErrMsg()));
} else {
    System.out.println(String.format("decrypted plaintext: %s size: %d type: %s\n", decrypt_res.getPlaintext(),
        decrypt_res.getPlaintextSize(), decrypt_res.getOriginalType()));
}
```

执行加密表的预编译 SQL 语句

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

执行加密表的批处理操作

```
// 调用Connection的prepareStatement方法创建预编译语句对象。
Connection conn = DriverManager.getConnection("url","user","password");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO creditcard_info (id_number, name,
credit_card) VALUES (?,?,:)");
// 针对每条数据都要调用setShort设置参数，以及调用addBatch确认该条设置完毕。
int loopCount = 20;
for (int i = 1; i < loopCount + 1; ++i) {
    pstmt.setInt(1, i);
    pstmt.setString(2, "Name " + i);
    pstmt.setString(3, "CreditCard " + i);
    // Add row to the batch.
    pstmt.addBatch();
}
// 调用PreparedStatement的executeBatch方法执行批处理。
int[] rowcount = pstmt.executeBatch();
// 调用PreparedStatement的close方法关闭预编译语句对象。
pstmt.close();
```

7.3.4 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gsqll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gsqsl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

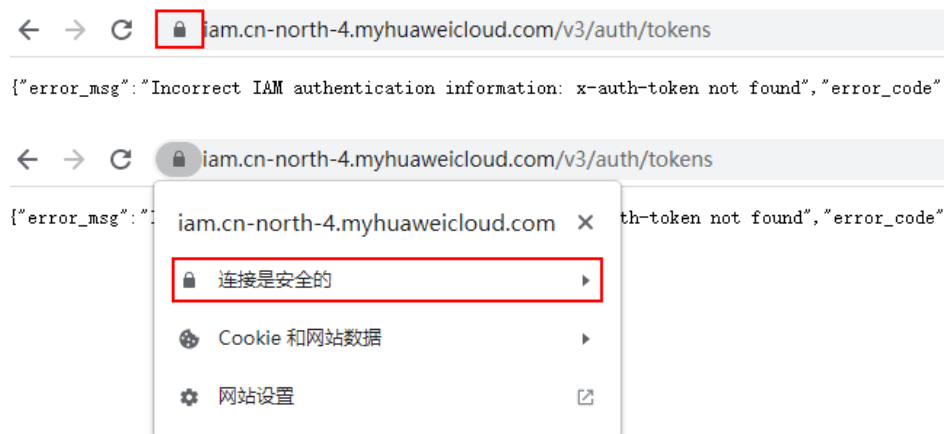
大部分浏览器均会自动下载网站对应的CA证书，并提供证书导出功能。虽然，很多网站也提供自动下载CA证书的功能，但可能因本地环境中存在代理或网关，导致CA证书无法正常使用。所以，建议借助浏览器下载CA证书。下载方式如下：

⚠ 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。

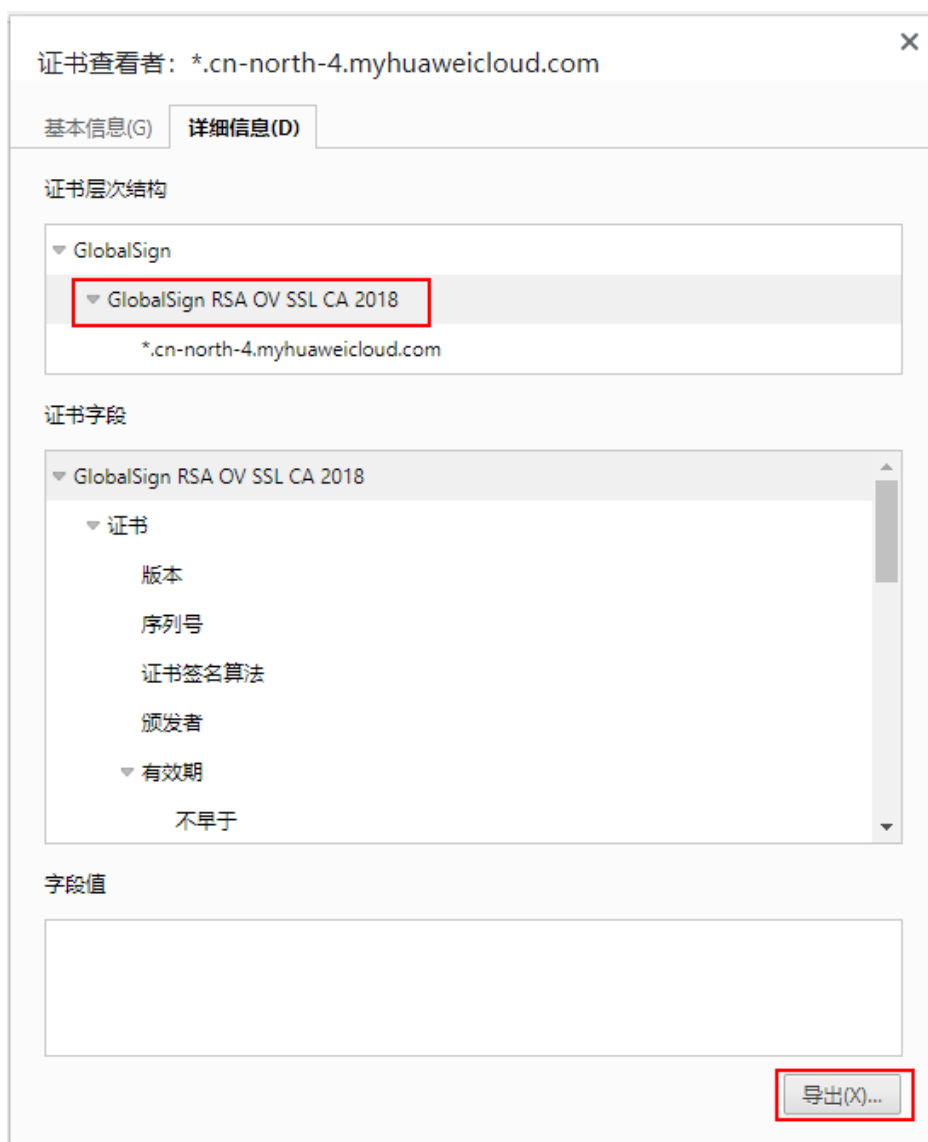
步骤1 输入域名：打开浏览器，在华为云场景中，分别输入IAM服务器地址和KMS服务器地址，地址获取方式参见：[生成主密钥阶段](#)。

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

7.3.5 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[用户自定义函数](#)章节和《参考》中“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《参考》中“系统表和系统视图 > 系统表 > 密态等值查询 > 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. 函数不支持RETURN QUERY EXECUTE 'query'动态查询语句实际返回数据为密态类型，而函数声明返回值为明文类型的场景。
4. 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。
5. 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，是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。
6. 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。
7. 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。
8. 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，且不支持介于 IS/AS 与 BEGIN 之间的声明部分进行解密操作，用户可以在执行函数时用输入参数、输出参数来代替使用。
9. 不支持gs_dump对涉及加密列的function进行备份后再恢复操作。
10. 不支持在函数/存储过程中创建密钥。
11. 密态函数/存储过程不支持触发器。
12. 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。
13. 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后立即使用该加密列进行加密处理，例如，存储过程中创建的加密表，添加加密列，向表中插入数据等组合操作无法支持，由于执行函数/存储过程是在服务端，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。
14. 不支持使用密态数据类型（byteawithoutorderwithequalcol、byteawithoutordercol、_byteawithoutorderwithequalcol、_byteawithoutordercol）创建函数和存储过程。
15. 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[,...])。
16. 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。

7.4 透明数据加密

透明加密提供表级数据加密存储功能。当用户使用本特性提供的语法创建加密表后，数据库向磁盘写入加密表数据前，会自动将其加密；同时，数据库从磁盘读取加密表数据后，会自动将其解密。向加密表中进行数据插入、更新、查询和删除等操作的语法与非加密表一致。

说明

- 使用透明加密前需先开启透明加密功能。开启操作请参见《用户指南》中“账号和网络安全 > GaussDB开启透明加密”章节。

查看透明加密基本配置

步骤1 查看透明加密功能是否已开启

enable_tde取值为on时表示开启，取值为off时表示关闭。

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

步骤2 查看是否已设置访问密钥管理服务的参数

tde_key_info参数为空时表示未设置，tde_key_info不为空时表示已设置。

```
gaussdb=# show tde_key_info;
tde_key_info
-----
keyType=...
```

----结束

操作加密表

步骤1 创建加密表。

创建表时，通过在WITH子句中设置enable_tde=on参数，即可设置该表为加密表。

数据库默认使用'AES_128_CTR'算法对加密表进行加密，如需使用其他算法，可通过encrypt_algo参数设置。

```
gaussdb=# CREATE TABLE t1 (c1 INT, c2 TEXT) WITH (enable_tde = on);
CREATE TABLE
gaussdb=# CREATE TABLE t2 (c1 INT, c2 TEXT) WITH (enable_tde = on, encrypt_algo = 'SM4_CTR');
CREATE TABLE
```

步骤2 查看加密表基本信息。

加密表基本信息存储在pg_class系统表中的reloptions字段中。其中，dek_cipher为数据密钥密文，由数据库自动生成，并由密钥管理服务加密。每个加密表都有1个独立的数据密钥。

```
gaussdb=# SELECT relname,reloptions FROM pg_class WHERE relname = 't1';
relname | reloptions
-----
t1      | {orientation=row,enable_tde=on,encrypt_algo=AES_128_CTR,compression=no,storage_type=USTORE,key_type=...,dek_cipher=...
```

执行如下语句查询数据库下全部加密表。

```
gaussdb=# SELECT relname,reloptions FROM pg_class WHERE reloptions::text LIKE '%enable_tde=on%';
relname | reloptions
-----
t1      | {orientation=row,enable_tde=on,encrypt_algo=AES_128_CTR,compression=no,storage_type=USTORE,key_type=...,dek_cipher=...
```

步骤3 向加密表写入数据。

操作加密表与非加密表的语法一致。数据库将表中数据写入磁盘前，才会自动对加密表的数据进行加密。

```
gaussdb=# INSERT INTO t1 VALUES (1, 'tde plain 123');
INSERT 0 1
```

步骤4 从加密表查询数据。

对于合法用户而言，查询加密表与非加密表的语法一致，加解密操作由数据库自动实现。如果攻击者绕过数据库，直接读取磁盘上加密表对应的数据文件，会发现文件中的数据均已被加密。

```
gaussdb=# SELECT * FROM t1;
c1 | c2
-----+-----
1 | tde plain 123
(1 row)
```

步骤5 轮转加密表的密钥。

为提高安全性，建议定期使用以下语法轮转加密表的数据密钥，即使用新的密钥对数据进行加密。

```
gaussdb=# ALTER TABLE t1 ENCRYPTION KEY ROTATION;
ALTER TABLE
```

轮转密钥后，数据库仍可以正常解密由旧密钥加密的数据。

步骤6 加密表与非加密表转换。

透明加密支持将加密表转换为非加密表，以及将非加密表转换为加密表。建议在每次转换后，手动执行VACUUM FULL tablename命令，以同步转换表中所有数据。

```
gaussdb=# CREATE TABLE t3 (c1 INT, c2 TEXT);
CREATE TABLE
gaussdb=# ALTER TABLE t3 SET (enable_tde = on);
ALTER TABLE
gaussdb=# VACUUM FULL t3;
VACUUM
gaussdb=# ALTER TABLE t3 SET (enable_tde = off);
ALTER TABLE
gaussdb=# VACUUM FULL t3;
VACUUM
```

步骤7 删除加密表。

```
gaussdb=# DROP TABLE IF EXISTS t1, t2, t3;
DROP TABLE
```

----结束

操作加密索引

步骤1 创建加密表。

创建索引的基表，需确保基表也是加密表。

```
gaussdb=# CREATE TABLE t1 (c1 INT, c2 TEXT) WITH (enable_tde = on);
CREATE TABLE
```

步骤2 创建加密索引。

与创建加密表的方式相同，通过在WITH子句中设置enable_tde=on参数，即将索引设置为加密索引。

索引与基表使用相同的加密算法和密钥，对基表进行密钥轮转时，索引也会使用新密钥。

```
gaussdb=# CREATE INDEX i1 ON t1(c2) WITH (enable_tde = on);
CREATE INDEX
```

步骤3 查看加密索引基本信息。

与加密表一样，索引基本信息也存储在pg_class系统表中的reloptions字段中，索引的dek_cipher、encrypt_algo等参数与基表保持一致。

```
gaussdb=# SELECT relname,reloptions FROM pg_class WHERE relname = 'i1';
relname | reloptions
-----
```

```
+-----+
i1      |
{orientation=row,enable_tde=on,encrypt_algo=AES_128_CTR,compression=no,storage_type=USTORE,key_type
=...,dek_cipher=...
```

步骤4 加密索引与非加密索引转换。

透明加密支持将非加密索引转换为加密索引，将加密索引转换为非加密索引。

```
gaussdb=# CREATE TABLE t2 (c1 INT, c2 TEXT) WITH (enable_tde = on);
ALTER TABLE
gaussdb=# CREATE INDEX i2 ON t2(c2);
CREATE INDEX
gaussdb=# ALTER INDEX i2 SET (enable_tde = on);
ALTER INDEX
gaussdb=# ALTER INDEX i2 SET (enable_tde = off);
ALTER INDEX
```

步骤5 自动对索引进行加密。

默认情况下，主动设置enable_tde参数才可创建加密索引。当设置GUC参数tde_index_default_encrypt=on，且以加密表为基表创建索引时，数据库会自动将索引转换为加密索引。示例如下：

```
gaussdb=# CREATE TABLE t3 (c1 INT, c2 TEXT) WITH (enable_tde = on);
ALTER TABLE
gaussdb=# CREATE INDEX i3 ON t3(c2);
CREATE INDEX
gaussdb=# SELECT relname, reloptions FROM pg_class WHERE relname = 'i3';
 relname | reloptions
-----
i3      |
{orientation=row,enable_tde=on,encrypt_algo=AES_128_CTR,compression=no,storage_type=USTORE,key_type
=...,dek_cipher=...

-- 解释：虽然未指定i3为加密索引，但是开启了tde_index_default_encrypt=on，且基表t3是加密表，数据库自动
将i3转换为加密索引
```

步骤6 删除加密表和索引。

```
gaussdb=# DROP TABLE IF EXISTS t1, t2, t3;
DROP TABLE
```

----结束

支持加密 Xlog 和 undo-log

步骤1 创建TDE属性的表。

```
gaussdb=# CREATE TABLE t1 (c1 INT, c2 TEXT) WITH (enable_tde = on);
CREATE TABLE
```

步骤2 记录Xlog的写入位置。

```
gaussdb=# SELECT pg_xlogfile_name_offset((SELECT pg_current_xlog_location()));
 pg_xlogfile_name_offset
-----
(000000010000000000000000D,3121400)
(1 row)
```

步骤3 操作含TDE属性的表，生成携带数据的Xlog。

```
gaussdb=# INSERT INTO t1 VALUES (1, 'ssssssssssssssss');
INSERT 0 1
```

步骤4 基于步骤2获取的Xlog写入位置，执行hexdump命令验证数据在Xlog中已被加密；\$DATADIR为实际的数据目录。

```
gaussdb=# \! hexdump -C $DATADIR/pg_xlog/000000010000000000000000D -s 3121400 -n 2048
002fa0f8 5e 00 00 80 01 00 00 00 00 00 00 00 00 00 00 00 |^.....|
```

```
002fa108 d8 9c 2f 0d 00 00 00 00 e0 16 00 00 00 00 00 00 |./.....|
002fa118 26 74 d4 9e ff 38 b8 17 00 00 00 00 00 00 d8 0e |&t...8.....|
002fa128 00 00 00 00 00 00 52 31 01 00 00 00 00 00 6b 26 |.....R1.....k&|
002fa138 06 00 00 00 00 00 7a 33 01 00 00 00 00 00 7b 33 |.....z3.....{3|
002fa148 01 00 00 00 00 00 f0 3d 2f 0d 00 00 00 00 00 00 |.....=/.....|
002fa158 3e 00 00 80 01 00 00 00 ab 33 01 00 00 00 00 00 |>.....3.....|
002fa168 f8 a0 2f 0d 00 00 00 00 80 16 00 00 00 00 00 00 |./.....|
002fa178 14 79 79 89 ff 18 00 00 00 00 00 10 00 00 00 00 |.yy.....|
002fa188 10 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
002fa198 3e 00 00 80 01 00 00 00 ab 33 01 00 00 00 00 00 |>.....3.....|
002fa1a8 58 a1 2f 0d 00 00 00 00 b0 16 00 00 00 00 00 00 |X./.....|
002fa1b8 6d 09 43 cc ff 18 00 00 00 00 00 10 00 00 00 80 |m.C.....|
002fa1c8 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
002fa1d8 13 02 00 80 01 00 00 00 ab 33 01 00 00 00 00 00 |.....3.....|
002fa1e8 98 a1 2f 0d 00 00 00 00 80 14 00 00 00 00 00 80 |./.....|
002fa1f8 12 61 3e 44 40 60 1d 00 7f 06 00 00 13 32 00 00 |.a>D@`.....2..|
002fa208 57 41 00 00 25 94 24 f9 f6 8d 46 50 8a f7 3f 92 |WA..%$....FP.?.|
002fa218 98 08 2c 57 a2 fc 16 3d 18 dc 6f 67 9e 8a c8 ba |.,W...=..og...|
002fa228 a3 7b 58 3f ba 50 d9 6c 52 d2 91 01 75 a7 a4 d9 |{X?.PIR...u...|
002fa238 05 86 2d b1 10 aa 1c 46 14 7f 00 00 d3 3a 4a 22 |..-....F.....J"|
002fa248 e0 55 00 00 7f 06 00 00 09 00 00 01 00 35 a8 44 |.U.....5.D|
002fa258 00 7f 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
```

步骤5 删除t1表数据，生成undo-log，判断undo文件已被加密；\$DATADIR为实际的数据目录。

预期：无法搜索到数据明文

```
gaussdb=# DELETE FROM t1;
DELETE 1
gaussdb=# \q
grep -c 'ssssssssssssss' $DATADIR/undo/permanent/*
```

步骤6 删除加密表。

```
gaussdb=# DROP TABLE IF EXISTS t1;
DROP TABLE
```

----结束

开启透明数据异步加密

步骤1 查看透明数据异步加密功能是否已开启。

enable_tde_async_encryption取值为on时表示开启，取值为off时表示关闭。

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

步骤2 创建多张透明加密表。

```
gaussdb=# CREATE TABLE t1 (C1 INT, C2 TEXT) WITH (ENABLE_TDE = ON);
CREATE TABLE
gaussdb=# CREATE TABLE t2 (C1 INT, C2 TEXT) WITH (ENABLE_TDE = ON, ENCRYPT_ALGO = 'SM4_CTR');
CREATE TABLE
```

步骤3 查看加密表基本信息。

上一步骤创建的多张加密表使用同一数据密钥，也即异步加密密钥，此处将其标识为<async_dek1>。

```
gaussdb=# SELECT relname, reloptions FROM pg_class WHERE relname = 't1';
relname | reloptions
-----
t1      | {enable_tde=on,encrypt_algo=AES_128_CTR,...,dek_cipher=<async_dek1>}
(1 row)
```

```
gaussdb=# SELECT relname, reloptions FROM pg_class WHERE relname = 't2';
 relname | reloptions
-----
t2       | {enable_tde=on,encrypt_algo=SM4_CTR,...,dek_cipher=<async_dek1>}
(1 row)
```

步骤4 轮转表级加密密钥。

明确提示enable_tde_async_encryption参数打开时，无法轮转表级数据密钥。

```
gaussdb=# ALTER TABLE t1 ENCRYPTION KEY ROTATION;
NOTICE: Unable to rotate key for encrypted table when parameter enable_tde_async_encryption is on.
ALTER TABLE
```

步骤5 轮转异步加密密钥。

```
gaussdb=# ALTER ASYNC ENCRYPTION KEY ROTATION;
ALTER TDE ROTATE ENCRYPTION KEY
```

步骤6 创建多张新的加密表，并查看加密表基本信息。

新创建的多张加密表使用同一数据密钥，也即轮转后的新异步加密密钥，此处将其标识为<async_dek2>；其与步骤2创建加密表密钥不同。

```
gaussdb=# CREATE TABLE t3 (C1 INT, C2 TEXT) WITH (ENABLE_TDE = ON);
CREATE TABLE
gaussdb=# CREATE TABLE t4 (C1 INT, C2 TEXT) WITH (ENABLE_TDE = ON, ENCRYPT_ALGO = 'SM4_CTR');
CREATE TABLE

gaussdb=# SELECT relname, reloptions FROM pg_class WHERE relname = 't3';
 relname | reloptions
-----
t3       | {enable_tde=on,encrypt_algo=AES_128_CTR,...,dek_cipher=<async_dek2>}
(1 row)
gaussdb=# SELECT relname, reloptions FROM pg_class WHERE relname = 't4';
 relname | reloptions
-----
t4       | {enable_tde=on,encrypt_algo=SM4_CTR,...,dek_cipher=<async_dek2>}
(1 row)
```

步骤7 删除全部加密表。

```
gaussdb=# DROP TABLE t1,t2,t3;
DROP TABLE
```

----结束

透明加密加载 KAE 加速引擎

步骤1 确保数据库集群各节点已正确安装KAE2版本加速引擎，确保数据库OS用户具备hisec2设备与KAE日志权限。

说明

KAE加速引擎的安装与权限配置，可参考[鲲鹏社区](#)。

步骤2 集群安装前在各节点配置环境变量用于指向OpenSSL KAE引擎动态库安装路径；或在集群完成安装后配置，但必须重启om_monitor进程才可生效。

```
export OPENSSL_ENGINES=/usr/local/lib/engines-3.0
```

步骤3 启动数据库

查看gs_log日志，有如下日志即表明高斯数据库加载KAE加速引擎成功。
Tde has successfully initialized the KAE engine and fetched the SM4 cipher.

----结束

关闭透明加密

将enable_tde参数配置为off，具体设置方式请参考《参考》中“数据库运行参数说明 > GUC参数说明 > 安全配置”章节。

须知

开启透明加密功能后，不建议关闭，关闭可能导致以下问题：

- 无法创建新加密表，旧加密表数据无法解密读取。
- 若已开启日志加密功能，关闭透明加密会造成已加密的redo/undo log回放失败，可能导致集群启动失败或节点状态异常。
- 若备份加密表或加密日志数据，将导致恢复失败。

常见问题

- 问题一：客户端回显或日志中提示curl失败，返回错误信息包含"Couldn't resolve host"。
原因：对于tde_key_info中配置的域名，操作系统无法映射到其IP。
解决：获取所配置IAM或KMS服务的IP地址，随后使用root用户，向/etc/hosts（沙箱环境为/var/chroot/etc/hosts）文件添加域名解析规则即可。

```
echo 'xxx.xxx.xxx.xxx hostname' >> /etc/hosts
```
- 问题二：客户端回显或日志中错误信息为"failed to find file '\$GAUSSHOME/gs_security/tde.key.cipher'"。
原因：KMS场景下，主机或备机节点上未完成对认证敏感信息加密的动作。
解决：确保全部数据库节点已执行认证敏感信息加密，随后重启数据库。
- 问题三：第三方GM0018标准加密库场景，多次尝试执行透明加密相关DDL/DML语句失败，报错为加密机接口调用错误。
原因：加密机客户端so库配置错误；加密机服务端功能异常；节点环境信号量异常。
解决：联系运维人员进行加密机客户端so库配置与服务端故障排查，节点环境执行ipcs命令确认数据库用户加密机信号量是否被异常清理；排除完成后可尝试重启恢复。

7.5 设置账本数据库

7.5.1 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 7-3 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 7-4 hash_ins 与 hash_del 场景对应关系

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

gaussdb=# \d+ blockchain.ledgermsp_usertable_hist;
      Table "blockchain.ledgermsp_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 ledgermsp.usertable VALUES(1, 'alex'), (2, 'bob'), (3, 'peter');
INSERT 0 3
gaussdb=# SELECT *, hash_69dd43 FROM ledgermsp.usertable ORDER BY id;
 id | name |      hash_69dd43
-----+-----+-----
  1 | alex | 1f2e543c580cb8c5
  2 | bob  | 8fcd74a8a6a4b484
  3 | peter| f51b4b1b12d0354b
(3 rows)

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

gaussdb=# DELETE FROM ledgermsp.usertable WHERE id = 3;
DELETE 1
gaussdb=# SELECT *, hash_69dd43 FROM ledgermsp.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 ledgermsp.usertable;
DROP TABLE
```



```
gaussdb=# DROP SCHEMA ledgernsp;
DROP SCHEMA
```

----结束

7.5.2 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[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《参考》中“系统表和系统视图 > 系统表 > 账本数据库 > 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7-3](#)。

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	relid	relnsp	relname	relhash
	globalhash						
	txcommand						
0	testdb	omm	2021-04-14 07:00:46.32757+08	16393	ledgernsp	usertable	a41714001181a294 6b5624e039e8aee36bff3e8295c75b40 insert into ledge
							rnsnp.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
							rnsnp.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[步骤2](#)中的记录一致。

----结束

7.5.3 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，接口返回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，接口返回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

7.5.4 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归档接口，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，接口返回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的全局区块表数据。执行操作为：

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


如果归档成功，接口返回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节点用户历史表归档成功。

步骤2 连接CN执行全局区块表归档操作。

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

查询结果如下：

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

全局区块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  
-----+-----+-----+-----+-----+-----+-----+-----  
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

7.5.5 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，接口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，可使用该接口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，接口返回修复过程中全局区块表中指定表的hash总和。

恢复用户历史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节点进行用户历史表修复操作。

```
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 | ledgermsp_tab2_hist
32841 | BIN$39B523388049$55C9EE0==$0 | pg_toast_32838
(4 rows)
-- 对用户表执行闪回drop。
gaussdb=# TIMECAPSULE TABLE ledgermsp.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('ledgermsp', 'tab2');
ledger_hist_repair
-----
000000000000000000
(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
-----
000000000000000000
(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: ledgernsp_tab3_hist

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

7.6 基于标签的强制访问控制

7.6.1 定义安全标签

前提条件

数据库正常运行，执行操作的用户需要具有SYSADMIN权限或者继承了内置角色gs_role_seclabel的权限。

背景信息

提供系统表gs_security_label来查看系统中已创建好的安全标签：
SELECT * FROM gs_security_label;

操作步骤

以某个实际业务应用举例说明，比如某公司的信息等级为公开、秘密和绝密，那么就可分别对应成等级L1、L2和L3。信息存储范围为一层到五层各层员工的信息，那么可分别对应成范围G1、G2、G3、G4、G5，以此为基础定义安全标签。

步骤1 创建一个安全标签，标志信息等级为公开，范围为二层和四层。

```
gaussdb=# CREATE SECURITY LABEL label1 'L1:G2,G4';
```

步骤2 创建一个安全标签，标志信息等级为秘密，范围为二层到四层。

```
gaussdb=# CREATE SECURITY LABEL label2 'L2:G2-G4';
```

步骤3 创建一个安全标签，标志信息等级为绝密，范围为一层到五层。

```
gaussdb=# CREATE SECURITY LABEL label3 'L3:G1-G5';
```

步骤4 查询系统表gs_security_label。

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

查询结果如下：

```
gaussdb=# SELECT * FROM gs_security_label;
label_name | label_content
-----+-----
label1    | L1:G2,G4
label2    | L2:G2-G4
label3    | L3:G1-G5
(3 rows)
```

步骤5 删除安全标签。

若要执行其他安全标签章节的示例，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 SECURITY LABEL label1;
gaussdb=# DROP SECURITY LABEL label2;
gaussdb=# DROP SECURITY LABEL label3;
```

----结束

7.6.2 应用安全标签

前提条件

数据库正常运行，执行操作的用户需要具有SYSADMIN权限或者继承了内置角色gs_role_seclabel的权限。

背景信息

- 提供系统表pg_seclabel来查看表或表列上的安全标签：
SELECT * FROM pg_seclabel;
- 提供系统表pg_shseclabel来查看用户/角色上的安全标签：
SELECT * FROM pg_shseclabel;
- 提供系统视图pg_seclabels来查看系统中所有的安全标签记录：
SELECT * FROM pg_seclabels;

操作步骤

数据库中已有用户user1、user2和数据表tbl。

步骤1 对用户user1应用安全标签label1，表示用户user1对应的信息等级为公开，范围为二层和四层。

```
gaussdb=# SECURITY LABEL ON USER user1 is 'label1';
```

步骤2 对用户user2应用安全标签label3，表示用户user2对应的信息等级为绝密，范围为一层到五层。

```
gaussdb=# SECURITY LABEL ON USER user2 is 'label3';
```

步骤3 对表tbl应用安全标签label2，表示表tbl对应的信息等级为秘密，范围为二层到四层。

```
gaussdb=# SECURITY LABEL ON TABLE tbl is 'label2';
```

步骤4 查询系统视图pg_seclabels。

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

查询结果如下：

```
gaussdb=# SELECT * FROM pg_seclabels;
objoid | classoid | objsubid | objtype | objnamespace | objname | provider | label
-----+-----+-----+-----+-----+-----+-----+-----
16399 | 1259 | 0 | table | 2200 | tbl | maclabel | label2
16391 | 1260 | 0 | role |  | user1 | maclabel | label1
16395 | 1260 | 0 | role |  | user2 | maclabel | label3
(3 rows)
```

----结束

7.6.3 基于标签的强制访问控制检查

前提条件

- 数据库正常运行，强制访问控制检查开关（enable_mac_check=on）打开。

- 执行操作的用户已具有所需要访问表的ACL权限。

背景信息

基于安全标签的强制访问控制策略规则由系统内置设定，用户不能更改：

- 插入（INSERT）策略：只有主体安全标记等级小于等于客体安全标记等级且主体安全标记范围是客体安全标记范围的子集时才允许插入数据。
- 查询（SELECT）策略：只有主体安全标记等级大于等于客体安全标记等级且主体安全标记范围是客体安全标记范围的超集时才允许查询数据。
- 修改（UPDATE）和删除（DELETE）策略：只有主体安全标记等级等于客体安全标记等级且主体安全标记范围等于客体安全标记范围时才允许修改和删除数据。
- 若客体未标记，则强制访问控制策略对该客体不生效。
- 若主体未标记，则不能访问任意带有标记的客体。

操作步骤

步骤1 授予用户user1和user2数据表tbl的SELECT和INSERT操作权限。

```
gaussdb=# GRANT SELECT,INSERT ON tbl TO user1,user2;
```

步骤2 用户user1登录数据库执行SELECT和INSERT操作，SELECT失败，INSERT成功，符合强制访问控制策略规则。

- 非透明多写特性

```
gaussdb=> SELECT current_user;
current_user
-----
user1
(1 row)
gaussdb=> SELECT * FROM tbl;
ERROR: permission denied for relation tbl
DETAIL: N/A
gaussdb=> INSERT INTO tbl VALUES (1);
INSERT 0 1
```

步骤3 用户user2登录数据库执行SELECT和INSERT操作，SELECT成功，INSERT失败，符合强制访问控制策略规则。

- 非透明多写特性

```
gaussdb=> SELECT current_user;
current_user
-----
user2
(1 row)
gaussdb=> SELECT * FROM tbl;
fir
----
1
(1 row)
gaussdb=> INSERT INTO tbl VALUES (1);
ERROR: permission denied for relation tbl --非透明多写特性
DETAIL: N/A
```

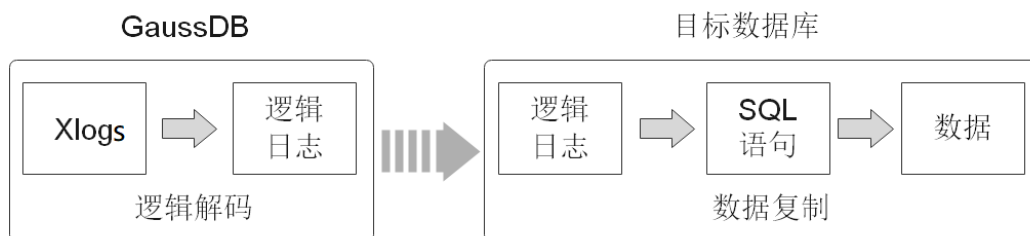
----结束

7.7 逻辑复制

逻辑复制分为逻辑解码与数据复制两部分。逻辑解码提取事务级逻辑日志，由业务或数据库中间件解析后完成数据复制。GaussDB数据库支持通过数据迁移工具定期向异

构数据库同步数据，不具备实时数据复制能力，不足以支撑与异构数据库间并网运行实时数据同步的诉求。因此，GaussDB数据库提供逻辑解码功能，通过解析Xlog生成逻辑日志，供目标数据库实时解析实现数据复制。降低对目标数据库的形态限制，支持异构数据库、同构异形数据库对数据的同步；支持目标数据库进行数据同步期间的数据可读写，实现数据同步低时延。实现逻辑如图7-3所示，本章节仅介绍逻辑解码。

图 7-3 逻辑复制



7.7.1 逻辑解码（非透明多写）

7.7.1.1 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。

须知

当前版本请使用直连DN解码，分布式强一致解码（CN解码）功能默认关闭，如有需要请联系华为工程师评估后打开enable-cn-decode参数。

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

- 推荐从DN抽取逻辑日志。从CN抽取存在较多限制和短板，如需使用请联系华为工程师。
- 为避免安全风险，请保证启用SSL连接。
- 设置GUC参数wal_level为logical。

📖 说明

如果GUC参数wal_level未设置为logical，则并行解码会跳过非logical的日志段，不输出解码日志；串行解码会报错。

- 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

- 每个逻辑复制槽仅解码单个数据库的解码；如需解码多个数据库，需创建多个逻辑复制槽。
- 多路逻辑复制同步时，源端数据库需为每条逻辑复制链路创建独立逻辑复制槽。
- 单个CN节点最多同时运行10个分布式强一致解码任务；单个DN节点最多同时运行20个解码任务。
- 用户需要通过DN端口连接数据库，才可以直接使用SQL函数接口进行逻辑解码操作，相关操作请参见[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。
- 若因DDL反解析失败导致DDL执行失败，可通过重试重新执行。极端场景通过重试仍无法成功，可通过设置GUC参数（enable_logical_replication_ddl = OFF）关闭DDL反解析功能，优先保证业务。
- 并行解码中，decode-style值为'b'或者't'时，单条元组（插入一条元组或者更新元组中原始数据和新数据之和）大于等于1023MB的解码数据会丢弃、并打印日志信息；decode-style值为'j'时，单条元组超过500MB的解码数据会丢弃、并打印日志信息。因此，为避免数据丢失，建议单条元组大小不超过500MB。
- 在wal_level值为logical且表REPLICA IDENTITY值为FULL时，更新超过1GB的表元组会失败。
- 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、DATEA、TIME[WITHOUT TIME ZONE]、TIMESTAMP[WITHOUT TIME ZONE]、CHAR(n)、VARCHAR(n)、TEXT、CLOB（解码成TEXT格式）。
- 在ORA兼容模式下，允许通过空字符串向BLOB或CLOB类型插入长度为0的字符串或NULL值。解码输出二进制格式时，可根据当前列值长度进行区分：如果为0xFFFFFFFF则表示NULL，如果为0则表示长度为0的字符串。例如，插入列值

empty_blob()生成的是非NULL的空BLOB对象，解码后输出长度为0的字符串；而插入列值为"则解码输出NULL。

- 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）。
- 不支持对PUBLIC SCHEMA相关的DDL操作解码。
- 逻辑解码输出结果不包含会话参数设置等状态信息（如：check_function_bodies、b_format_version），执行语句结果若依赖状态信息，在目的端回放时，需要目的端和源端的会话参数一致，否则可能报错或者数据不一致。
- 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“_”、“?”、“-”、“.”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）。
- 对多个数据库的解码需要分别在数据库内创建流复制槽并开始解码，每个数据库的解码都需要单独扫描一遍日志。
- 逻辑解码时不支持强切，强切后需要重新全量导出数据。
- 备机解码时，switchover和failover可能导致解码数据增多，需手动过滤。Quorum协议下，switchover和failover选择升主的备机需要与当前主机日志同步。
- 备机解码时若需删除数据库，需确保数据迁移完成后由主机删除对应逻辑复制槽的数据库。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。
- 备机解码时，因为优先保证备机回放，存在解码线程和备机回放冲突导致函数解码和流式串行解码线程被终止的情况。如果有超大事务需要解码，则可以根据事务的解码时长适当调整相关GUC参数max_standby_streaming_delay、standby_max_query_time和standby_force_recycle_ratio。
- 只支持CN和主DN创建和删除复制槽。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!!!”。
- 当主机删除复制槽时，若备机Xlog回放延迟严重导致备机未能及时删除对应复制槽，后续若使用备机的复制槽进行解码，回放删除操作时将导致解码失败。使用备机复制槽解码时，建议先判断其restart_lsn是否与主机一致，若二者不一致，则该复制槽为残留复制槽。
- 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。
- 主机和备机同时解码场景下，当主机发生故障、主备切换时，可能会存在某个节点超过20个复制链路上限的风险（原来主机的复制链路会转移到新主机）。
- 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，不解码数据插入的DML操作。
- 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，具体配置方式请参见《参考》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。
- 基于目标数据库可能需要源数据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。若目标数据库不需要复制其他相关系统表的内容，则逻辑日志回放过程中需要对相关系统表进行过滤。
- 在开启逻辑复制的场景下，如需创建包含系统列的主键索引，必须将该表的REPLICA IDENTITY属性设置为FULL或是使用USING INDEX指定不包含系统列的、唯一的、非局部的、不可延迟的、仅包括标记为NOT NULL列的索引。
- 在开启逻辑复制的场景下，建议用户表定义主键或唯一约束，以保证回放的确定性，否则回放SQL语句时可能会导致多行修改影响数据正确性。
- 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，配置方式请参见《参考》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，会打印一条WARNING日志。
- 在逻辑解码大事务过程中，系统会将中间解码结果写入数据盘进行暂存。事务写入的Xlog与落盘文件大小比例小于1:1.5，即每写入1GB大事务的Xlog日志，可能产生不超过1.5GB落盘文件，导致磁盘使用率上升，建议节点预留足够磁盘空间，确保在高峰解码期间仍满足安全水位。
- 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/FFFFFFFF。该状态的逻辑复制槽不参与阻塞日志回收或系统表历史版本的回收，但仍占用复制槽的限制数量，需要手动删除。
- 备机解码启动后，向主机发送复制槽推进指令后会占用主机上对应的逻辑复制槽（即标识为活跃状态）。在此之前主机上对应逻辑复制槽为非活跃状态，此状态下如果满足逻辑复制槽强制失效条件则会被标记为失效（即restart_lsn将被设置为7FFFFFFF/FFFFFFFF），备机将无法推进主机复制槽，且备机回放完成复制槽失效日志后当前复制槽的备机解码断开后将无法重连。
- 使用备机解码时建议只在固定备机上启动解码任务：当某个备机节点上的逻辑复制槽被强制失效时，解码任务切换到该备机节点启动时会报错。
- 在备机执行串行逻辑解码过程中，若备机回放进度滞后，可能出现解码快照获取异常并触发报错，建议备机回放完成后重试解码操作。
- 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进程，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，待删除完成后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。执行备份恢复操作时，需要停止所有解码任务，否则有可能导致逻辑复制槽清理失败。
- 集群库级恢复操作后，该数据库数据字典复制槽无法正常使用，需要重新执行实例级逻辑解码数据字典基线化操作（会导致实例上其他数据库的数据字典复制槽失效，需要重建数据字典类型复制槽），执行基线化前请确认评估其他数据库数据字典槽失效风险。
- 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，该DDL语句将不解码。

- 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。
 - 逻辑解码支持的DML类型在Xlog中有如下几种操作：
 - 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进行解码时，无论创建表名或用户名时使用的是大写还是小写，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。
 - 函数解码不支持0x字符。
 - 在直连DN解码场景中，在解码到还未提交的事务时需等待当前事务提交确认：若等待时长超过transaction_sync_timeout参数值，会因超时而导致解码中断。
 - 关于指定策略号的ILM压缩语法，由于高级压缩的特性机制，源端和目标端实例上表的策略号可能会不一致。

```
gaussdb=# ALTER TABLE tbl_ilm ILM ENABLE POLICY p1; --p1为源库的策略号
gaussdb=# ALTER TABLE tbl_ilm ILM DISABLE POLICY p1; --p1为源库的策略号
gaussdb=# ALTER TABLE tbl_ilm ILM DELETE POLICY p1; --p1为源库的策略号
```
 - 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。当业务中包含大量的DDL操作，且逻辑复制槽处于不活跃状态或者客户端消费过慢时，会导致系统表历史版本过多。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：
- 及时清理不再使用的逻辑复制槽。
 - 如果逻辑复制槽推进过慢，请联系技术支持进行处理。
- 并行逻辑解码任务会占用部分CPU资源，如果解码并行数过多，可能会引发资源争抢，影响在线业务。在x86架构部署中，硬件配置规格为128核CPU/1024GB内存，TPCC场景下，每个并行解码任务（并行度为4）对在线业务影响约15%。
 - 如果系统资源不足或运行环境波动，可能出现解码线程创建失败或启动超时，进而导致并行逻辑解码任务启动失败，对应报错：Wait for worker ready failed。偶发问题可通过客户端重试逻辑解码临时规避，如果故障频繁复现，需降低集群整体负载，保障解码任务具备充足资源正常运行。
 - 若表名、列名、索引名等数据库对象名属于系统关键字（如FLOAT、ENTITYESCAPING等），解码结果无法保证这类关键字能被反引号`或双引号"正确包裹转义。因此在目标库应用解码结果时，需遵循目标库的语法规则，按需为关键字补充或修改`或"包裹，避免出现语法解析报错。
 - 当表的分布方式为非Replication类型时，DN解码只解析当前节点的WAL日志。

SQL 函数解码性能

1. 在Benchmarksq1-5.0的100warehouse场景下，采用pg_logical_slot_get_changes时：
 - 单次解码数据量4K行（对应约5MB~10MB日志），解码性能0.3MB/s~0.5MB/s。
 - 单次解码数据量32K行（对应约40MB~80MB日志），解码性能3MB/s~5MB/s。

- 单次解码数据量256K行（对应约320MB~640MB日志），解码性能3MB/s~5MB/s。
 - 单次解码数据量再增大，解码性能无明显提升。
- 如果采用pg_logical_slot_peek_changes + pg_replication_slot_advance方式，解码性能相比采用pg_logical_slot_get_changes时要下降30%~50%。
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。

7.7.1.2 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函数解码请参见《参考》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流式解码请参见[逻辑复制（非透明多写）](#)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。
取值类型：boolean型
取值范围：
 - false：解码出的data列不包含xid信息。
 - true：解码出的data列包含xid信息。默认值：true
- **skip-empty-xacts:**
解码时是否忽略空事务信息。
取值类型：boolean型
取值范围：
 - false：解码时不忽略空事务信息。
 - true：解码时会忽略空事务信息。默认值：false
- **include-timestamp:**
解码信息是否包含commit时间戳。
取值类型：boolean型
取值范围：
 - false：解码信息不包含commit时间戳。
 - true：解码信息包含commit时间戳。默认值：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。
- **only-local:**
是否仅解码本地日志。
取值类型：boolean型
取值范围：
 - false：解码非本地日志和本地日志。
 - true：仅解码本地日志（即RepOriginId为0时，该参数具体信息详见《参考》中“数据库运行参数说明 > GUC参数说明 > 双集群复制参数”章节）。默认值：true
- **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.*');
```

📖 说明

- 在对失效视图执行ALTER VIEW语句时，逻辑解码会记录该语句，此时white-table-list参数无效，即无论视图定义语句中查询的表是否在白名单列表中，失效视图的ALTER VIEW语句均会被解码。
- 视图失效重编译请参见[失效重编译](#)章节。
- 若白名单内表长期无业务则会导致复制槽的restart_lsn不推进引发WAL日志堆积，可以通过周期性调用gs_add_logical_decoding_position_xlog()系统函数进行解决。
- max-txn-in-memory:
内存管控参数，单个事务占用内存大于该值即进行落盘。
取值单位：MB
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，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。并行解码中该参数已废弃使用，不生效。
取值单位：GB
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，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。
取值类型：整型
取值单位：MB
取值范围：10~1048576
默认值：200

📖 说明

- 从内核505.2.1.SPC0900以下版本升级到当前版本后部分场景可能会出现性能下降情况，按照最佳实践调参，最佳实践如下：
- desc-memory-limit设置为100MB时可以支撑此场景：1000张热点表，平均100字段，平均50个分区，平均每10分钟产生1个新的历史版本（例如对表执行DDL修改表结构、autovacuum修改pg_class中元组的reloptions字段）。
 - 其中任一指标若有增加，需要根据增加倍数扩大参数值，例如在2000张热点表，平均200个字段，平均每5分钟产生1个新的历史版本的场景，则需要将desc-memory-limit设置为100*2*2=800MB。
- 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。
取值类型：boolean型
取值范围：
 - false：表示事务的BEGIN逻辑日志不输出事务的用户名。

- true：表示事务的BEGIN逻辑日志输出事务的用户名。
默认值：false
- 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。
取值类型：字符串类型
取值范围：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。
取值类型：字符串类型
取值范围：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。
取值类型：boolean型
取值范围：
 - false：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：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。默认值：true
- 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。
取值类型：boolean型
取值范围：
 - true：仅允许连接备机解码，连接主机解码时会报错退出。
 - false：不做限制，允许连接主机或备机解码。默认值：false

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，GaussDB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。
取值类型：整型
取值单位：毫秒（ms）
取值范围：0~2147483647
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。

说明

sender-timeout为心跳检测的阈值。一旦检测超时，系统将自动重试；若重试失败，则主动断开连接并终止解码任务。

- change-log-max-len:**

逻辑日志缓存长度上限参数。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。

取值单位：字节

取值范围：1~65535

默认值：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。本地缓存的日志条数，本地缓存日志个数不足时，从全局缓存获取。

取值范围：1~65535

默认值：8192

启动解码任务时，并行解码日志的缓存条数受max-queue-cache-num参数上限值控制。前向兼容性：若该参数值设置小于8192，启动解码任务时会自动调整max-decode-to-sender-cache-num参数值，并输出提示信息到系统日志中。
- enable-heartbeat:**

仅流式解码时设置，代表是否输出心跳日志。

取值类型：boolean型

取值范围：

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

默认值：false

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。

取值类型：整型

取值范围：1~20，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：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。

取值类型：整型

取值范围：

- 0：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（即逻辑复制槽为LSN序复制槽）时可使用该方式，否则报错。
- 1：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。

默认值：0

须知

- 当output-order不配置（即默认值为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。

取值类型：boolean型

取值范围：

- true：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：完全交由复制业务调用日志确认接口推进逻辑复制槽。

默认值：false

- 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，CN解码不支持配置此选项。

取值类型：boolean型

取值范围：

- true/on：不输出存储生成列的解码结果。

- false/off: 输出存储生成列的解码结果。

默认值: false/off

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。

取值类型: boolean型

取值范围:

- true: 开启DDL语句的逻辑解码。
- false: 不开启DDL语句的逻辑解码。

默认值: false

- 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。

取值类型: boolean型

取值范围:

- true: 传送JSON格式的DDL反解析结果。
- false: 传送decode-style指定格式的DDL反解析结果。decode-style='j'时，则输出TEXT格式。

默认值: false

- timezone-is-utc:

逻辑解码控制参数，用于控制携带时区的时间类型数据的输出（例如：ORA、MYSQL兼容下的timestampz类型，M兼容的timestamp类型）。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。

取值类型: boolean型

取值范围:

- true: 解码时间类型数据输出0时区的时间。
- false: 解码时间类型数据输出当前数据库时区的时间。

默认值: false

- 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。

取值类型: boolean型

取值范围:

- true: 暂不支持设置。
- false: 不输出sequence值的变更日志的解码结果。

默认值: false

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。

- enable-decode-position:

逻辑解码控制参数，该参数主要控制逻辑解码读取到解码位置日志时，是否向客户端发送该位置日志信息。

取值类型：boolean型

取值范围：

- true：发送位置日志。
- false：不发送位置日志。

默认值：false

说明

此处说明并行解码场景下解码位置日志如何解析：二进制格式首先是字符'L'，表示消息是解码位置日志，之后是位置日志内容，依次为：8字节uint64代表CSN，表示发送解码位置日志时对应的CSN值；8字节uint64代表LSN或者CSN，当解码为LSN序时，表示发送解码位置日志时刻的WAL日志的位置，当解码为CSN序时，表示发送解码位置日志时对应的CSN值；4字节uint32代表时间戳（从1970年1月1日开始），表示调用gs_add_logical_decoding_position_xlog系统函数时产生的时间戳。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。具体解析详见图7-4：

图 7-4 心跳协议格式



- white-table-list1, white-table-list2, white-table-list3, white-table-list4, white-table-list5, white-table-list6, white-table-list7, white-table-list8, white-table-list9:
白名单扩展参数，功能与white-table-list一致，包含需要进行解码的Schema名和表名。该参数仅对流式解码有效，SQL函数解码不支持。
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。
- data-limit
逻辑解码输出数据量控制参数。
在GUC参数logical_decode_options_default中设置时，取值范围：【0，100】的整数。单位：GB。默认值：10。取值为0时，表示不限制解码结果大小。
GUC参数设置需与pg_logical_get_area_changes函数中data-limit入参配合使用，具体请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > 逻辑复制函数”章节“pg_logical_get_area_changes”函数详细说明。
- enable-rowno
逻辑解码控制参数，该参数用以控制解码包含rowid系统列的无主键表且REPLICA IDENTITY选项明确设置为FULL或者实际效果等同于FULL时，是否输出INSERT/

UPDATE/DELETE操作中新元组和旧元组的rowno系统列。使用该选项输出rowno系统列，请确保目标库中存在名称和类型相同的列，否则回放逻辑解码日志可能失败。如有疑问，请联系华为技术工程师提供技术支持。

取值类型：boolean型

取值范围：

- true：输出rowno系统列。
- false：不输出rowno系统列。

默认值：false

须知

- 该选项对sql_decoding插件无效。
- REPLICA IDENTITY选项的实际效果等同于FULL目前指的是Ustore表的NOTHING和DEFAULT选项，详情参见《参考》中“SQL参考 > SQL语法 > A > ALTER TABLE”章节中REPLICA IDENTITY选项的说明。

- enable-cn-decode

逻辑解码控制参数，该参数用以控制解码是否开启CN解码，CN解码默认关闭，需要直连DN解码。

取值类型：boolean型

取值范围：

- true/on：开启CN解码。
- false/off：关闭CN解码。

默认值：false/off，新安装的数据库的默认值。该参数为507.0.0版本新增参数，从507.0.0之前版本升级到当前版本后此参数默认值为false/off，如果升级前存在CN解码任务，升级后需要打开该参数，否则启动CN解码任务会报错。

须知

该选项仅支持在GUC参数logical_decode_options_default中设置，默认关闭，如要开启该参数，请联系华为技术工程师。

分布式强一致解码

须知

当前版本请使用直连DN解码，分布式强一致解码（CN解码）功能默认关闭，如有需要请联系华为工程师评估后打开enable-cn-decode参数。

- 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。

取值类型：整型

取值范围：1~20

默认值：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。

取值类型：整型

取值范围：0~8192

默认值：-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。

取值类型：字符串类型

取值范围：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。decode-style='j'时，DDL输出TEXT格式，DML采用decode-style指定的JSON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c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c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c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则表示该列为rowno系统列，实际rowno系统列长度为8字节；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，其中rowno系统列请按照8字节读取。
 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。

取值类型：整型

取值范围：

- 0：逐条发送解码结果。

- 1: 解码结果累积到达1MB则批量发送解码结果。

默认值: 0

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。
取值类型: 整型
取值范围: 2~1024，且必须为2的整数幂。
默认值: 128
队列长度和解码过程的内存使用量正相关。
- max-logical-decode-memory
并行解码内存管控参数，并行解码任务的总内存（包含缓存）大于该值时则对当前解码事务进行落盘。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。
取值类型: 整型
取值单位: GB
取值范围: 6~1024
默认值: 6
启动解码任务时，受max-control-memory-size参数控制上限值。

须知

- 在JSON解码格式下，由于JSON的树状结构特性，内存可能短暂升高，建议设置为10GB。若设置为6GB，可能会出现短时超6GB的现象，此为预期情况。解码完成后，临时内存将被释放，整体内存不会超过max-logical-decode-memory。解码格式相关设置详见[decode-style](#)参数说明。
 - 业务中处理大TOAST场景时，逻辑解码任务内存参数建议配置：max-logical-decode-memory = 线程并行度 * 1GB + 10GB，如果计算的max-logical-decode-memory参数大于max-control-memory-size，需同时在GUC参数logical_decode_options_default中调整max-control-memory-size大小。
 - 设置时建议不超过max_dynamic_memory总量的50%，否则解码任务占用内存过高，导致在线业务申请内存失败。
- max-control-memory-size
并行解码内存管控参数max-logical-decode-memory的上限值控制参数。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。
取值类型: 整型

取值单位：GB

取值范围：6~1024

默认值：10

如果启动参数max-logical-decode-memory设置值超过该值，启动解码任务时会自动调整max-logical-decode-memory参数值，并输出提示信息到系统日志中。

须知

- 解码选项max-control-memory-size仅支持在GUC参数logical_decode_options_default中设置，如果在启动解码任务时设置该参数，将报错退出。
- 设置时建议不超过max_process_memory总量的50%，否则解码任务过程中存在申请内存溢出风险。

- max-toast-memory-size
并行解码toast内存管控参数。并行解码任务中，读取Xlog中toast字段的内存使用上限值，超过该值则对当前解码事务的toast数据进行落盘。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。
取值类型：整型
取值单位：MB
取值范围：100~1048576
默认值：100

须知

解码选项max-toast-memory-size仅支持在GUC参数logical_decode_options_default中设置，如果在启动解码任务时设置该参数，将报错退出。

- max-queue-cache-num
并行解码参数max-decode-to-sender-cache-num的上限值控制参数。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。
取值类型：整型
取值单位：个数
取值范围：8192~65535
默认值：16384
如果启动参数max-decode-to-sender-cache-num设置值超过该值，启动解码任务时会自动调整max-decode-to-sender-cache-num参数值，并输出提示信息到系统日志中。

须知

解码选项max-queue-cache-num仅支持在GUC参数logical_decode_options_default中设置，如果在启动解码任务时设置该参数，将报错退出。

7.7.1.3 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_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

7.7.1.4 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[逻辑复制（非透明多写）](#)章节。

7.7.1.5 逻辑解码支持 DDL

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

表 7-5 具体支持的 DDL 类型

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

表 7-6 M-Compatibility 模式数据库支持的 DDL 类型

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

功能描述

数据库在执行DML操作的时候，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。

从解析树反解析出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操作执行性能的影响降低幅度不超过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整条语句不解码（可以通过查询系统表pg_proc的“provolatile”字段判断default值和check表达式是否是stable类型或者volatile类型）。

-- 案例场景1：

```
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b1 TIMESTAMP DEFAULT NOW(); -- 's' 不解码  
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b2 INT DEFAULT RANDOM(); -- 'v' 不解码
```

```
gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b3 INT DEFAULT ABS(1); -- 'I' 解码
```

```
-- 案例场景2:
```

```
gaussdb=# CREATE DATABASE m_db with DBCOMPATIBILITY = 'M' template=templatem
encoding 'UTF8';
```

```
m_db=# CREATE TABLE t1(id int) CHAR SET = 'GBK';
```

```
m_db=# ALTER TABLE t1 ADD COLUMN col1 varchar(100) DEFAULT '1'; -- M兼容模式中内部会调用pg_convert函数对字符集进行转换，pg_convert是stable类型的函数，ALTER TABLE语句不解码。
```

- 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。
- 仅支持基本DDL语法。
- 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。严格判断混合事务场景，只要父事务或子事务任一包含DDL操作，则判定为混合事务，后续DML操作将不进行解码。

```
-- 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=# GRANT 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;

-- 父子事务场景，子事务有DDL且回滚，全部不解析
gaussdb=# BEGIN;
gaussdb=# savepoint s1;
gaussdb=# CREATE TABLE t2(col int);
gaussdb=# INSERT INTO t2 VALUES(1);
gaussdb=# ROLLBACK TO s1;
gaussdb=# insert INTO t1 VALUES(2);
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。
- 针对存储过程/函数/包头（PACKAGE SPECIFICATION）/包体（PACKAGE BODY）使用“/”作为结束标识符的SQL，解码结果不包含“/”。第三方工具使用解码结果回放数据时，根据实际场景自行处理。

■ 原始SQL语句：

```
CREATE OR REPLACE PROCEDURE p62_1(var62_1 int)
IS
BEGIN
    var62_1 := 5;
END;
/
```

反解析结果：

```
CREATE OR REPLACE PROCEDURE public.p62_1 ( IN var62_1 pg_catalog.int4 ) VOLATILE
CALLED ON NULL INPUT SECURITY INVOKER COST 100 AS DECLARE BEGIN
    var62_1 := 5;
END ;
```

■ 原始SQL语句：

```
CREATE OR REPLACE FUNCTION proc_add2 (num1 in int, num2 in int, num3 out int)
return int
```

```
AS
BEGIN
num3 := num1 + num2;
END;
/
```

反解析结果：

```
CREATE OR REPLACE FUNCTION public.proc_add2 ( IN num1 pg_catalog.int4, IN num2
pg_catalog.int4, OUT num3 pg_catalog.int4 ) RETURN pg_catalog.int4 VOLATILE CALLED
ON NULL INPUT SECURITY INVOKER COST 100 AS DECLARE
BEGIN
num3 := num1 + num2;
END ;
```

■ 原始SQL语句：

```
CREATE TABLE tbl_test(uid varchar(6) PRIMARY KEY, area_id varchar(5), level int);
--创建包头
CREATE OR REPLACE PACKAGE pkg_test AS
--公有存储过程可以被外部调用。
PROCEDURE proc_ist_tbl_test(v_uid in varchar, v_area_id varchar, v_level int);
PROCEDURE proc_del_tbl_test(v_uid in varchar);
PROCEDURE proc_upd_tbl_test(v_uid in varchar, v_area_id varchar, v_level int);
END pkg_test;
/
--创建包体
CREATE OR REPLACE PACKAGE BODY pkg_test AS
PROCEDURE proc_ist_tbl_test(v_uid in varchar, v_area_id varchar, v_level int) AS
BEGIN
INSERT INTO tbl_test VALUES (v_uid, v_area_id, v_level);
END;
PROCEDURE proc_del_tbl_test(v_uid in varchar) AS
BEGIN
DELETE FROM tbl_test WHERE uid = v_uid;
END;
PROCEDURE proc_upd_tbl_test(v_uid in varchar, v_area_id varchar, v_level int) AS
BEGIN
UPDATE tbl_test SET area_id = v_area_id, level = v_level WHERE uid = v_uid;
END;
var4 int := 10;
--实例化开始。
BEGIN
var4 := 10;
dbe_output.print_line(var4);
END pkg_test;
/
```

反解析结果：

```
CREATE TABLE public.tbl_test (uid pg_catalog."varchar"(6) NOT NULL, area_id
pg_catalog."varchar"(5), level pg_catalog.int4, CONSTRAINT tbl_test_pkey PRIMARY KEY
(uid)) NOCOMPRESS
CREATE OR REPLACE PACKAGE public.pkg_test AUTHID CURRENT_USER IS --公有存储过程
可以被外部调用。
PROCEDURE proc_ist_tbl_test(v_uid in varchar, v_area_id varchar, v_level int);
PROCEDURE proc_del_tbl_test(v_uid in varchar);
PROCEDURE proc_upd_tbl_test(v_uid in varchar, v_area_id varchar, v_level int);
END pkg_test;
CREATE OR REPLACE PACKAGE BODY public.pkg_test IS PROCEDURE
proc_ist_tbl_test(v_uid in varchar, v_area_id varchar, v_level int) AS
BEGIN
INSERT INTO tbl_test VALUES (v_uid, v_area_id, v_level);
END;
PROCEDURE proc_del_tbl_test(v_uid in varchar) AS
BEGIN
DELETE FROM tbl_test WHERE uid = v_uid;
END;
PROCEDURE proc_upd_tbl_test(v_uid in varchar, v_area_id varchar, v_level int) AS
BEGIN
UPDATE tbl_test SET area_id = v_area_id, level = v_level WHERE uid = v_uid;
END;
var4 int := 10;
```

```
--实例化开始。
BEGIN
    var4 := 10;
    db_output.print_line(var4);
END
pkg test;
```

- 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。

SOL命令:

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

- MySQL兼容模式下，不支持ALTER SCHEMA schema_name WITHOUT/WITH BLOCKCHAIN语法反解析。
- 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。
- 逻辑解码不支持解析以CREATE USER方式创建出的Schema。

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 % persistence}s TABLE % if_not_exists}s % identity%D % table_elements}s %
    {with_clause}s % compression}s","identity":
    {"object_name":"test_create_table_a","schema_name":"public","compression":"NOCOMPRESS","persist
      ence":"","with_clause":{"fmt":{"WITH % (with, }s)","with":{"fmt":{"label}s = % value}L","label":
        {"fmt":{"label}I","label":"orientation","value":"row"},"fmt":{"label}s = % value}L","label":
          {"fmt":{"label}I","label":"compression"},"value":"no"}}},"if_not_exists":"","table_elements":{"fmt":{"%
            elements, }s"},"elements":{"fmt":{"name}I % column_type}T","name":"","a","column_type":
              {"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
cid	CommandId	Command ID
tag_type	NodeTag	DDL类型
message_size	Size	日志内容长度

名称	类型	意义
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，该参数需要重启集群才能生效。wal_level参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 预写式日志 > 设置”章节。

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;\n
PROCEDURE testpro1(var3 int); --公有存储过程，可以被外部调用\nEND ldap_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

7.7.1.6 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. 每条WAL日志不能超过500MB。
5. 不支持扩容前的Xlog日志数据找回。
6. 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。
7. 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。
8. 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%"。
9. 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。
10. 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。
11. 如果进行了节点替换，不支持节点替换前的数据找回。
12. 不支持同一节点并发调用数据找回接口。
13. 只支持直连DN方式的数据找回。
14. 旧版本升级提交到新版本时，如果未基线化，则需要执行基线化（执行SELECT gs_logical_dictionary_disabled()和SELECT gs_logical_dictionary_baseline()函数）后才可以使数据找回功能，且只支持对基线化后新产生的Xlog日志进行数据找回，升级前产生的Xlog日志无法解析。

15. 默认支持1年内的数据找回，超过1年的数据将被自动清除。如需找回超过1年的数据，需在清理前调整GUC参数logical_replication_dictionary_retention_time的值。
16. 数据找回功能依赖表的元数据信息，如果元数据异常，数据找回可能无法正常执行并报错退出。
17. 数据找回功能不支持事务粒度找回，仅支持按行找回。回滚事务会打印事务XID，需人工处理；混合事务中的DML语句找回数据，需人工核对正确性。

7.8 分区表

本章节将探讨在大数据量场景下，分区表如何实现查询优化与运维管理，系统地讲解分区表的使用，涵盖其语义、原理、约束及限制等方面。

7.8.1 表分区介绍

7.8.1.1 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20多年时间数据库业界的发展历程中，数据量从最初的MB、GB逐渐发展到现在的TB级数据量，在如此大规模、多样化的数据背景下，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运维管理方式。

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。

7.8.1.2 表分区技术

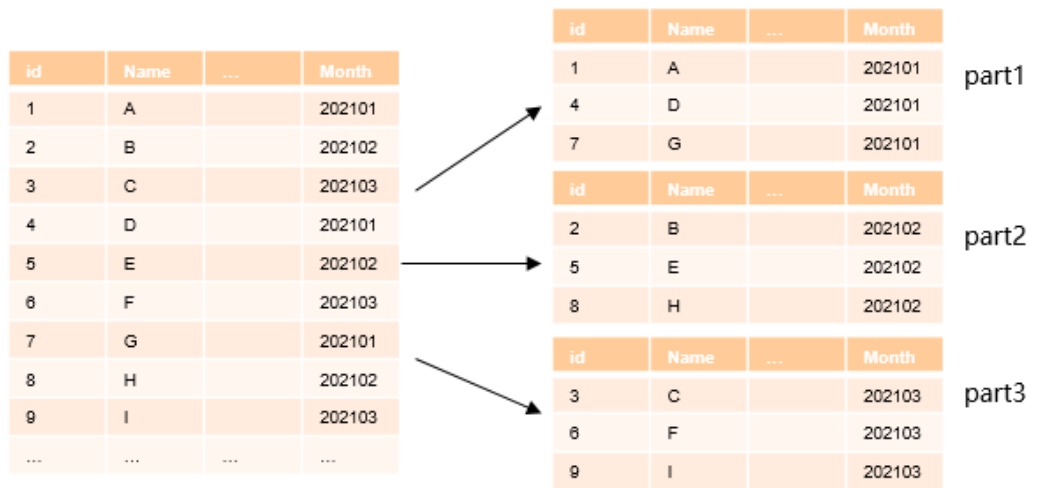
表分区技术（Table-Partitioning）通过将非常大的表或者索引从逻辑上切分为更小、更易管理的逻辑单元（分区），能够让用户对表查询、变更等语句操作具备更小的影响范围，能够让用户通过分区键（Partition Key）快速定位到数据所在的分区，从而避免在数据库中对大表的全量扫描，能够在不同的分区上并发进行DDL、DML操作。从用户使用的角度来看，表分区技术主要有以下能力：

- 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，查询时只需访问相关分区的子集，而非整个表。这种分区剪枝技术能够显著提升查询性能，提供数量级的性能增益。
- 降低运维与查询的并发操作影响：分区表可以显著减少DML语句和DDL语句在并发场景下的相互影响。在大数据量且按时间维度进行分区的场景下，这种优势尤为明显。例如，新数据分区的入库和实时点查操作，以及老数据分区的数据清洗和分区合并等运维操作，可以独立进行，互不干扰。
- 提供大容量场景下灵活的数据运维管理方式：分区表通过物理上对不同分区的数据进行隔离，每个分区可以独立设置物理属性，如启用或禁用压缩、物理存储设置和表空间。此外，分区表支持分区级别的数据管理操作，如数据加载、索引创建和重建，以及备份和恢复，无需对整个表进行操作，从而大大减少了操作时间。

7.8.1.3 数据分区查找优化

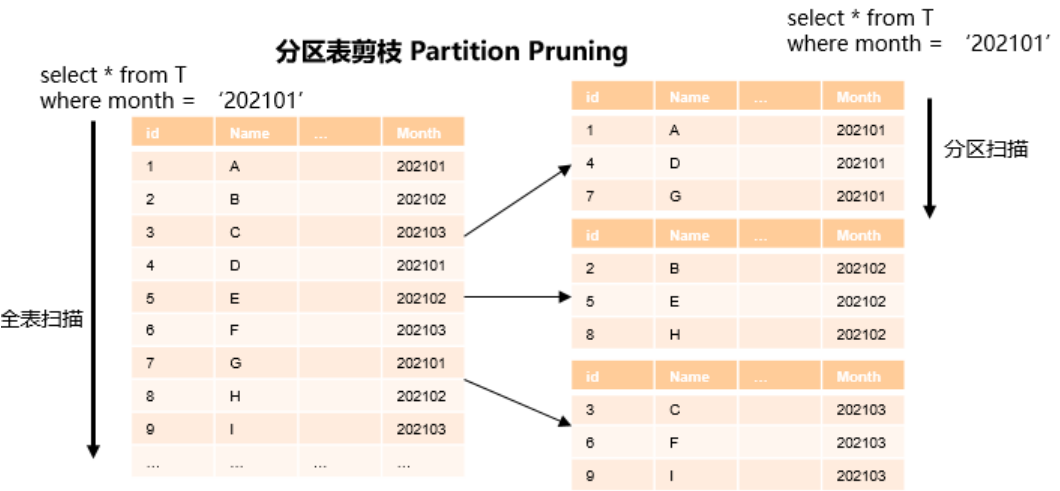
分区表在数据查找方面的优势主要体现在对分区键进行谓词查询的场景。例如，考虑一张以月份（Month）作为分区键的表，如图7-5所示。

图 7-5 分区表示例图



如果使用普通表设计，查询时需要进行全表扫描（Full Table Scan）。而如果以月份为分区键重新设计该表，全表扫描会被优化为分区扫描。当表内数据量很大且历史周期较长时，通过减少扫描的数据量，性能提升将非常显著，如图7-6所示。

图 7-6 分区表剪枝示例图



7.8.1.4 数据分区运维管理

数据生命周期管理（Data Life Cycle Management，DLM）是一套贯穿数据全生命周期的管理流程与策略体系，其核心任务之一在于为不同阶段的数据匹配最适宜且经济高效的存储介质：高频访问的新数据存放于读写速度快、可用性高的存储层，而低频访问的旧数据则迁移至成本较低、性能稍弱的存储层。鉴于旧数据更新频率低，将其压缩并转为只读存储模式更具合理性。

分区表为DLM方案落地创造了理想条件，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。此类底层配置由数据库

运维人员在服务端完成，终端用户无感知，仍可按常规逻辑对表执行查询操作。此外，各分区支持独立开展备份、恢复、索引重建等运维工作，可针对数据集的不同子集实施分治管理，满足用户业务场景的差异化需求。

7.8.2 分区表介绍

分区表（Partitioned Table）是指在单节点内，依据分区键及相应分区策略，对表数据进行逻辑层面的切分，本质上属于水平分区（Horizontal partition）策略。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。通过分区，表、索引及索引组织表可被拆分为更小单元，实现数据库对象的精细化管理与访问。

GaussDB提供了丰富的分区策略和扩展能力，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完成，用户无感知，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：

- 分区表基本概念：从分区表的基本概念出发，介绍分区表的Catalog存储方式以及内部对应原理。
- 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。

7.8.2.1 基本概念

7.8.2.1.1 分区表（分区母表）

分区表作为用户实际操作的表对象，支持常规DML（数据操作语言）的增、删、查、改操作。通常通过在建表DDL（数据定义语言）语句中显式使用PARTITION BY子句进行定义。创建成功以后在pg_class表中新增一条记录，其parttype字段值为'p'（标识一级分区）或's'（标识二级分区），以此标记该记录对应的表为分区母表。

需要注意的是，分区母表本质上是逻辑概念，其对应的物理表文件并不实际存储数据。

示例1：t1_hash为一个分区表，分区类型为hash：

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

gaussdb=# \d+ t1_hash
Table "public.t1_hash"
Column | Type   | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1      | integer |           | plain   |               |
c2      | integer |           | plain   |               |
c3      | integer |           | plain   |               |
Partition By: HASH(c1)
Number of partitions: 10 (View pg_partition to check each partition range.)
Distribute By: HASH(c1)
```

```
Location Nodes: ALL DATANODES
Has OIDs: no
Options: orientation=row, compression=no

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

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

示例2：t1_sub_rr为一个二级分区表，分区类型为range-list：

```
gaussdb=# CREATE TABLE t1_sub_rr (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p_2021 VALUES LESS THAN (2022) (
        SUBPARTITION p_2021_1 VALUES (1),
        SUBPARTITION p_2021_2 VALUES (2),
        SUBPARTITION p_2021_3 VALUES (3)
    ),
    PARTITION p_2022 VALUES LESS THAN (2023) (
        SUBPARTITION p_2022_1 VALUES (1),
        SUBPARTITION p_2022_2 VALUES (2),
        SUBPARTITION p_2022_3 VALUES (3)
    ),
    PARTITION p_2023 VALUES LESS THAN (2024) (
        SUBPARTITION p_2023_1 VALUES (1),
        SUBPARTITION p_2023_2 VALUES (2),
        SUBPARTITION p_2023_3 VALUES (3)
    ),
    PARTITION p_2024 VALUES LESS THAN (2025) (
        SUBPARTITION p_2024_1 VALUES (1),
        SUBPARTITION p_2024_2 VALUES (2),
        SUBPARTITION p_2024_3 VALUES (3)
    ),
    PARTITION p_2025 VALUES LESS THAN (2026) (
        SUBPARTITION p_2025_1 VALUES (1),
        SUBPARTITION p_2025_2 VALUES (2),
        SUBPARTITION p_2025_3 VALUES (3)
    ),
    PARTITION p_2026 VALUES LESS THAN (2027) (
        SUBPARTITION p_2026_1 VALUES (1),
        SUBPARTITION p_2026_2 VALUES (2),
        SUBPARTITION p_2026_3 VALUES (3)
    )
);

gaussdb=# \d+ t1_sub_rr
          Table "public.t1_sub_rr"
Column | Type  | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1      | integer |          | plain   |              |
c2      | integer |          | plain   |              |
c3      | integer |          | plain   |              |
Partition By RANGE(c1) Subpartition By LIST(c2)
Number of partitions: 6 (View pg_partition to check each partition range.)
Number of subpartitions: 18 (View pg_partition to check each subpartition range.)
Has OIDs: no
Distribute By: HASH(c1)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
```

```
--查询t1_sub_rr分区类型。
gaussdb=# SELECT relname, parttype FROM pg_class WHERE relname = 't1_sub_rr';
 relname | parttype 
-----+-----
 t1_sub_rr | s
(1 row)

--删除t1_sub_rr。
gaussdb=# DROP TABLE t1_sub_rr;
```

7.8.2.1.2 分区

分区是分区表中实际保存数据的表，对应的记录通常保存在pg_partition中，各分区通过parentid字段作为外键，与pg_class表中分区母表对应的OID（对象标识符）列建立关联关系。

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;
```

7.8.2.1.3 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策略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。

7.8.2.2 分区策略

分区策略描述了在分区表中数据和分区路由映射规则，在建表时通过DDL语句中的PARTITION BY语法指定。

常见的分区类型有基于条件的范围分区、基于哈希散列函数的哈希分区、基于数据枚举的列表分区：

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

7.8.2.2.1 范围分区

范围分区（Range Partition）依据所设定的每个分区的分区键值范围，来实现数据到各个分区的映射。在实际的生产系统里，范围分区是很常用的一种分区类型，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：

- 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，名为date_202002的分区表示2020年2月的分区，将包含分区键值在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，用于明确该分区的非包含上限值。当数据的分区键值大于或等于当前分区的该上限值时，这些数据将会被添加至下一个分区当中。除第一个分区外，其余所有分区都存在一个由前序分区的VALUES LESS子句所隐式指定的下限值。

针对最高分区，可以定义使用MAXVALUE关键字，MAXVALUE代表着一个虚拟的无限值，在排序规则上，它的优先级高于分区键的其他任何可能取值，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。

- **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)

--删除表和Schema。
gaussdb=# DROP TABLE tpcds.startend_pt;
DROP TABLE
gaussdb=# DROP SCHEMA tpcds;
DROP SCHEMA
```

7.8.2.2.2 哈希分区

哈希分区（Hash Partition）是依据GaussDB内置的哈希算法，对分区键进行运算，从而实现数据到各分区的映射。在分区键取值范围不倾斜（no data skew）的场景下，哈希算法能够让数据行在各个分区之间均匀分布，进而使得各分区的大小大致保持一致，是实现分区间数据均匀分布的理想方法。哈希分区也是范围分区的一种易于使用的替代方法，尤其是当待分区的数据并非历史数据，或者没有明显可用于分区的分区键时。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
);
```

```
--删除表。  
gaussdb=# DROP TABLE bmsql_order_line;
```

上述例子中，使用bmsql_order_line表的ol_d_id列作为依据进行了分区，ol_d_id列是identifier性质的属性列，其本身既不具备时间维度的特征，也无法在某一特定维度上对数据做出区分。在这种情况下，采用哈希分区策略来对该表进行分表处理无疑是一个相当理想的选择。

相比其他分区类型，哈希分区除了需要提前确认分区键不存在严重的数据倾斜问题（即某一个或某几个值出现极高的重复频率）之外，用户仅需指定分区键以及分区数量，便能够完成分区的创建工作。而且，哈希分区能够切实保证每个分区内的数据实现均匀分布，在很大程度上提升了分区表的易用性。

7.8.2.2.3 列表分区

列表分区（List Partition）允许用户在每个分区的描述里，为分区键指定离散值列表，从而控制数据行如何被映射到各个分区。这种分区方式的优势在于，它能以枚举分区值的形式对数据进行分区操作，尤其适合对无序且不相关的数据集进行分组与组织。

在处理分区键值时，若某些值未被定义在列表之中，用户可以借助默认分区（DEFAULT）来存储这些数据。通过这种方式，所有未能映射到其他任何分区的行都不会导致错误产生。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)  
);  
  
--删除表。  
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，同样依据ol_d_id列来进行分区。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，列表分区对分区键的可控性更强，通常能够精确地把目标数据存放在预期的分区中。然而，要是列表中的取值较多，分区定义工作就会变得繁琐。在这种情形下，建议采用哈希分区。

总体而言，列表分区和哈希分区常用于对无序、不相关的数据集进行分组与组织。

须知

列表分区在分区键设置上有明确的数量限制，其分区键最多可支持16列。并且，分区键列数不同时，子分区定义时列表中枚举值对NULL值的处理规则也不同。当分区键仅定义为1列时，子分区列表里的枚举值不允许为NULL值；而当分区键定义为多列时，子分区列表中的枚举值则允许存在NULL值。

7.8.2.2.4 二级分区

二级分区（Subpartition，也叫组合分区）是基本数据分区类型的组合，将表通过一种分区策略进行分区，然后使用第二种分区策略将每个分区进一步细分为二级分区。给定分区的所有二级分区表示数据的逻辑子集。常见的二级分区组合如下所示：

- Range-Range
- Range-List
- Range-Hash
- List-Range
- List-List
- List-Hash
- Hash-Range
- Hash-List
- Hash-Hash

示例如下：

```
--Range-Range
gaussdb=# CREATE TABLE t_range_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 VALUES LESS THAN (10) (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 VALUES LESS THAN (20) (
        SUBPARTITION p2sp1 VALUES LESS THAN (15),
        SUBPARTITION p2sp2 VALUES LESS THAN (20)
    )
);
gaussdb=# DROP TABLE t_range_range;

--Range-List
gaussdb=# CREATE TABLE t_range_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 VALUES LESS THAN (10) (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 VALUES LESS THAN (20) (
```

```

        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_range_list;

--Range-Hash
gaussdb=# CREATE TABLE t_range_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY HASH (c2)
SUBPARTITIONS 2
(
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (20)
);
gaussdb=# DROP TABLE t_range_hash;

--List-Range
gaussdb=# CREATE TABLE t_list_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 VALUES (1, 2) (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 VALUES (3, 4) (
        SUBPARTITION p2sp1 VALUES LESS THAN (5),
        SUBPARTITION p2sp2 VALUES LESS THAN (10)
    )
);
gaussdb=# DROP TABLE t_list_range;

--List-List
gaussdb=# CREATE TABLE t_list_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 VALUES (1, 2) (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 VALUES (3, 4) (
        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_list_list;

--List-Hash
gaussdb=# CREATE TABLE t_list_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY HASH (c2)

```

```

SUBPARTITIONS 2
(
    PARTITION p1 VALUES (1, 2),
    PARTITION p2 VALUES (3, 4)
);
gaussdb=# DROP TABLE t_list_hash;

--Hash-Range
gaussdb=# CREATE TABLE t_hash_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 (
        SUBPARTITION p2sp1 VALUES LESS THAN (5),
        SUBPARTITION p2sp2 VALUES LESS THAN (10)
    )
);
gaussdb=# DROP TABLE t_hash_range;

--Hash-List
gaussdb=# CREATE TABLE t_hash_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 (
        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_hash_list;

--Hash-Hash
gaussdb=# CREATE TABLE t_hash_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY HASH (c2)
SUBPARTITIONS 2
(
    PARTITION p1,
    PARTITION p2
);
gaussdb=# DROP TABLE t_hash_hash;

-- list-list
gaussdb=# CREATE TABLE list_list
(
    month_code VARCHAR2 ( 30 ) NOT NULL ,

```

```
dept_code VARCHAR2 ( 30 ) NOT NULL ,
user_no   VARCHAR2 ( 30 ) NOT NULL ,
sales_amt int
)
PARTITION BY LIST (month_code) SUBPARTITION BY LIST (dept_code, user_no)
(
  PARTITION p_201901 VALUES ( '201902' ),
  PARTITION p_201902 VALUES ( '201903' )
  (
    SUBPARTITION p_201902_a VALUES (('1','120')),
    SUBPARTITION p_201902_b VALUES (('2','240'))
  )
);
gaussdb=# SELECT * FROM pg_get_tabledef('list_list');
gaussdb=# INSERT INTO list_list VALUES('201902', '1', '120', 1);
gaussdb=# INSERT INTO list_list VALUES('201902', '2', '240', 1);
gaussdb=# INSERT INTO list_list VALUES('201902', '1', '120', 1);
gaussdb=# INSERT INTO list_list VALUES('201903', '2', '240', 1);
gaussdb=# INSERT INTO list_list VALUES('201903', '1', '120', 1);
gaussdb=# INSERT INTO list_list VALUES('201903', '2', '240', 1);
gaussdb=# SELECT * FROM list_list ORDER BY month_code;
gaussdb=# DROP TABLE list_list;

-- hash-range
gaussdb=# CREATE TABLE hash_range
(
  month_code VARCHAR2 ( 30 ) NOT NULL ,
  dept_code  VARCHAR2 ( 30 ) NOT NULL ,
  user_no    VARCHAR2 ( 30 ) NOT NULL ,
  sales_amt  int
)
PARTITION BY hash (month_code) SUBPARTITION BY range (dept_code,user_no)
(
  PARTITION p_201901,
  PARTITION p_201902
  (
    SUBPARTITION p_201902_a VALUES LESS THAN ( '2' ,2),
    SUBPARTITION p_201902_b VALUES LESS THAN ( '3' ,4)
  )
);
gaussdb=# INSERT INTO hash_range VALUES('201901', '1', '1', 1);
gaussdb=# INSERT INTO hash_range VALUES('201901', '2', '1', 1);
gaussdb=# INSERT INTO hash_range VALUES('201901', '1', '1', 1);
gaussdb=# INSERT INTO hash_range VALUES('201903', '2', '1', 1);
gaussdb=# INSERT INTO hash_range VALUES('201903', '1', '1', 1);
gaussdb=# INSERT INTO hash_range VALUES('201903', '2', '1', 1);
gaussdb=# SELECT * FROM hash_range;
gaussdb=# DROP TABLE hash_range;
```

说明

Interval分区可以看成是范围分区的一种特殊形式。

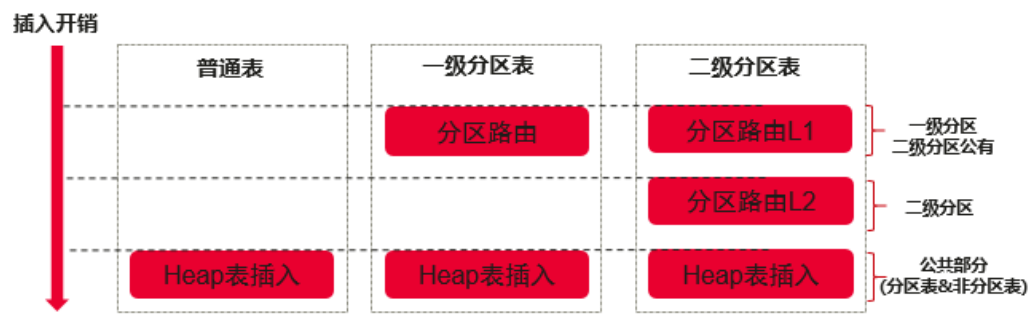
7.8.2.2.5 分区表对导入操作的性能影响

在GaussDB中，相较于非分区表，分区表在数据插入处理流程中额外增加了分区路由环节的开销。

所以从整体来看，分区表场景下的数据插入开销主要由两部分构成，如图7-7所示：

（1）heap-insert基表插入：此部分负责解决元组（tuple）存入对应堆表（heap表）的问题，并且这一操作在普通表和分区表中是通用的；（2）partition-routing分区路由：该部分主要解决分区路由问题，也就是要将元组插入到对应的分区表（partRel）中。

图 7-7 普通表&分区表数据插入



因此对数据插入优化的侧重点如下：

- heap-insert基表插入：
 - 算子底噪优化。
 - heap数据插入。
 - 索引插入build优化（带索引）。
- partition-routing分区路由：
 - 路由查找算法逻辑优化。
 - 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

分区路由的性能在大数据量的单条INSERT语句执行时能得到显著体现。而在UPDATE场景中，其内部操作逻辑更为复杂，它需要先查找出对应要更新的元组，执行DELETE操作将其移除，之后再执行INSERT操作插入新的元组。相较于单条INSERT语句直接插入数据的场景，UPDATE场景的操作步骤更多，流程更繁琐，所以分区路由性能在UPDATE场景下的体现不如单条INSERT语句场景直接。

不同分区类型的路由算法逻辑如表7-7所示：

表 7-7 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(logN)	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。

须知

分区路由的核心处理逻辑是依据导入数据元组的分区键来计算其所属分区，这是分区表相较于非分区表额外增加的开销。这部分开销对最终数据导入性能的具体影响，与服务器的CPU 处理能力、表的宽度以及磁盘和内存的实际容量相关。

通常情况下，可以进行如下粗略估算：

- 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服务器场景下数据导入性能约降低20%。

x86和ARM服务器在分区表导入性能上出现这种细微差异，主要原因是分区路由属于内存内计算强化场景，主流x86体系的CPU在单核指令处理能力方面稍强于ARM。

7.8.2.3 分区基本使用

7.8.2.3.1 创建分区表

规格约束

单个表的最大分区数为1048575（包括一级分区和二级分区）。较多的分区会导致系统中文件数增加、占用更多的系统缓存和锁资源，进而影响系统的性能。在默认配置下，建议单个表的分区数不超过400。如果实际使用的热点分区数超过400，考虑到分区表的使用性能，建议调整FASTPATH_PART的值（调整方法参见GUC参数num_internal_lock_partitions），调整配置后，建议单个表的分区数不超过2000。

创建普通分区表（创建一级分区表）

SQL语言具备强大且灵活多样的功能，其语法树往往较为复杂，分区表亦是如此。分区表的创建可视为在原有非分区表的基础上增添表分区属性。所以，分区表的语法接口可看作是对原有非分区表CREATE TABLE语句的扩展，具体是增加了PARTITION BY 语句部分，同时需指定与分区相关的三个核心元素：

- 分区类型（partType）：用于描述分区表所采用的分区策略，主要包括范围分区、列表分区和哈希分区。
- 分区键（partKey）：用于确定分区表的分区列。目前，范围分区和列表分区支持多列（不超过16列）作为分区键，而哈希分区仅支持单列分区。
- 分区表达式（partExpr）：用于描述分区表具体的分区方式，也就是键值与分区之间的对应映射关系。

这三个重要元素会在建表语句的Partition By Clause子句中体现，具体形式为PARTITION BY partType (partKey) (partExpr[,partExpr]...)。其基本语法框架如下：

```
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 ];
```

非透明多写特性下，在分区表的使用中，存在以下规格约束：

- 范围分区和列表分区最大支持16个分区键，而哈希分区只支持1个分区键。
- 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。但是当范围分区表定义了MAXVALUE分区，或者列表分区表定义了DEFAULT分区时，允许插入空值。

创建二级分区表

二级分区表可视为一级分区表的进一步拓展。在二级分区表结构里，第一层分区是一张逻辑表，并不实际承担数据存储功能，数据真正的存储位置是二级分区节点。

从实现上而言，二级分区表的分区方案是由两个一级分区嵌套而来，一级分区的分区策略详见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE PARTITION”章节。

常见的二级分区表组合策略有：Range-Range分区、Range-List分区、Range-Hash分区、List-Range分区、List-List分区、List-Hash分区、Hash-Range分区、Hash-List分区、Hash-Hash分区等。目前二级分区仅支持行存表，二级分区创建的基本语法框架如下：

```
CREATE TABLE [ IF NOT EXISTS ] subpartition_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 ]
PARTITION BY {RANGE | LIST | HASH} (partKey) SUBPARTITION BY {RANGE | LIST | HASH} (partKey)
(
    PARTITION partition_name partExpr... /* 第一层分区 */
    (
        SUBPARTITION partition_name partExpr ... /* 第二层分区 */
        SUBPARTITION partition_name partExpr ... /* 第二层分区 */
    ),
    PARTITION partition_name partExpr... /* 第一层分区 */
    (
        SUBPARTITION partition_name partExpr ... /* 第二层分区 */
        SUBPARTITION partition_name partExpr ... /* 第二层分区 */
    ),
    ...
)
[ { ENABLE | DISABLE } ROW MOVEMENT ];
```

规格约束：

二级分区表支持列表/哈希/范围分区的任意两两组合。

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：

- 增加分区
- 删除分区
- 删除/清空分区数据
- 切割分区
- 合并分区
- 移动分区
- 交换分区
- 重命名分区

以上常用的分区属性变更语句是对普通表ALTER TABLE语句的扩展，在分区表里的使用方法与普通表基本类似，分区表属性变更的基本语法框架如下：

```
/* 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](#)、《参考》中“SQL参考 > SQL语法 > A > ALTER TABLE PARTITION”和“SQL参考 > SQL语法 > A > ALTER TABLE SUBPARTITION”章节。

7.8.2.3.2 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，具体请参见《参考》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。

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)
);

-- 删除表。
gaussdb=# DROP TABLE list_01;
```

指定分区业务中，PARTITION p_list_1与PARTITION FOR (4)等价，它们实际上指向同一个分区；同理，PARTITION p_list_3与PARTITION FOR (12)等价，同样代表同一个分区。

7.8.2.3.3 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。

当执行指定分区的DQL/DML操作时，若插入的数据不属于目标分区，业务将会报错；若查询的数据不属于目标分区，则跳过对这些数据的处理。

指定分区DQL/DML支持以下几类语法：

- 查询（SELECT）
- 插入（INSERT）
- 更新（UPDATE）
- 删除（DELETE）
- 插入或更新（UPSERT）
- 合并（MERGE INTO）

指定分区进行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;
--插入数据。
gaussdb=# INSERT INTO list_02 VALUES(null, 'alice', 'alice data');
gaussdb=# INSERT INTO list_02 VALUES(2, null, 'bob data');
gaussdb=# 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
----+-----+-----
  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;
gaussdb=# DROP TABLE list_tmp;
```

7.8.3 分区表查询优化

说明

本节示例对应explain_perf_mode的参数值为normal。

7.8.3.1 分区剪枝

分区剪枝是GaussDB提供的一种分区表查询的优化技术，数据库SQL引擎会根据查询条件，只扫描特定的部分分区。该优化动作自动触发，当分区表的查询条件契合剪枝场景时，分区剪枝便会自行启动。

依据剪枝所处阶段的差异，分区剪枝可划分为静态剪枝与动态剪枝两类：

- 静态剪枝发生于优化器阶段，在生成执行计划前，数据库已掌握需访问的分区信息；
- 动态剪枝则在执行器阶段（执行开始或执行过程中）实施，于生成计划时，数据库尚未明确需访问的分区信息，仅判定“具备分区剪枝条件”，具体的剪枝细节由执行器确定。

分区剪枝支持范围如下：

- 支持分区级别：一级分区、二级分区。
- 支持分区类型：全部分区类型，包括范围分区、哈希分区、列表分区。
- 支持表达式类型：比较表达式（<，<=，=，>=，>）、逻辑表达式、数组表达式。对于哈希分区、列表分区，比较表达式只有是等值（=）情况下，才能进行分区剪枝。
- 支持条件类型：常量、参数、函数，函数需要为immutable函数和stable函数（参数化路径剪枝不支持stable函数），不支持子查询等复杂条件。
- 分区键类型和条件类型不同触发类型转换场景，需要保证类型可以相互转换。

需注意的是，只有分区表页面扫描和Local索引扫描才会触发分区剪枝，Global索引没有分区概念，不需要进行剪枝。对于Global索引，会根据索引条件决定去访问哪些分区。所以总体上建议：查询带有分区键的条件时推荐使用Local索引，查询不带分区键的条件时推荐使用Global索引，但Global索引在分区维护操作上会有额外的开销。可以通过创建索引时显式申明GLOBAL/LOCAL关键字来控制索引创建类型。

7.8.3.1.1 分区表静态剪枝

对于检索条件中分区键上带有常量的分区表查询语句，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。

说明

- 静态剪枝只支持常量。部分非常量条件会在计划生成前进行预处理，比如参数条件生成custom plan，immutable函数和stable函数带常量入参，strict函数入参为NULL等，此时均会转为常量，可以走静态剪枝。
- 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：

- 非透明多写特性下，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
Partitioned Seq Scan on public.t1
  Output: c1, c2
  Filter: (t1.c1 = 1)
  Selected Partitions: 1

(12 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
Partitioned Seq Scan on public.t1
  Output: c1, c2
  Filter: (t1.c1 < 1)
  Selected Partitions: 1

(12 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
Partitioned Seq Scan on public.t1
  Output: c1, c2
  Filter: (t1.c1 IS NULL)
```

```
Selected Partitions: 3
(12 rows)
- 非透明多写特性下，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
Partitioned Seq Scan on public.t1
  Output: c1, c2
  Filter: ((t1.c1 = 1) AND (t1.c2 = 2))
  Selected Partitions: 1

(12 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
Partitioned Seq Scan on public.t1
  Output: c1, c2
  Filter: ((t1.c1 = 1) OR (t1.c1 = 2))
  Selected Partitions: 1

(12 rows)
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE NOT c1 = 1;
QUERY PLAN
-----
Data Node Scan
  Output: t1.c1, t1.c2
  Node/s: All datanodes
  Remote query: SELECT c1, c2 FROM public.t1 WHERE NOT c1 = 1

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

(15 rows)
- 非透明多写特性下，数组表达式
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 IN (1, 2, 3);
QUERY PLAN
-----
Data Node Scan
  Output: t1.c1, t1.c2
  Node/s: All datanodes
  Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

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

```

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

(12 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
Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
Selected Partitions: 1

(12 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
Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
Selected Partitions: 1

(12 rows)

```

- 静态剪枝不支持的典型场景具体示例如下：

非透明多写特性下，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)

--删除表。
gaussdb=# DROP TABLE t1;
```

7.8.3.1.2 分区表动态剪枝

当分区表查询语句的检索条件中存在带有变量的情况，由于在优化器阶段无法获取用户输入的绑定参数，所以优化器阶段仅能对indexscan、bitmapindexscan、indexonlyscan等算子的检索条件进行解析。后续在执行器阶段，待获取绑定参数后，才会完成分区筛选。

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。

PBE 动态剪枝

对于检索条件中分区键上带有参数的分区表查询语句，在优化器阶段只能判定“具备分区剪枝条件”，在执行启动完成绑参后，再进行动态剪枝。动态剪枝的触发行为与静态剪枝一致，只是触发阶段不同。

须知

- 对于部分stable函数，如to_timestamp等类型转换函数，其剪枝结果可能会因GUC参数的变动而受到影响。为了保持性能优化效果，可以通过对表执行analyze操作重新生成gplan来解决。
- 由于PBE动态剪枝是基于gplan来进行的，在判断语句是否能进行PBE动态剪枝时，建议将参数plan_cache_mode设置为'force_generic_plan'，排除cplan的干扰。
- 使用动态剪枝时，由于在生成计划阶段计划实际执行的分区尚未确定，不支持生成分类索引计划。

- PBE动态剪枝支持的典型场景具体示例如下：

- 非透明多写特性下，比较表达式。

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

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

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

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

(13 rows)
```

- 非透明多写特性下，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
Partitioned Seq Scan on public.t1
Output: c1, c2
Filter: ((t1.c1 = $1) AND (t1.c2 = $2))
Selected Partitions: 1 (pbe-pruning)

(13 rows)
```

- 非透明多写特性下，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：

- 非透明多写特性下，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: PART
-> 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)
```

- 非透明多写特性下，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)
```

- 非透明多写特性下，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;
```

参数化路径动态剪枝

当分区表的分区键所在列生成参数化路径（分区键所在列和其他表的列进行关联，且分区表作为内表生成NestLoop索引扫描）时，会触发参数化路径动态剪枝。参数化路径动态剪枝会在每次外表扫描完成后，驱动内表扫描时进行一次。

须知

- 参数化路径动态剪枝在多数情况下不支持子查询表达式（剪枝算子选择参数化路径且分区键上包含了一个InitPlan子查询条件除外），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。
- 使用EXPLAIN显示计划时，参数化路径剪枝计划中显示的分区数只代表“可能访问的最大总分区数”，实际访问分区数可能会小很多。
- 使用EXPLAIN ANALYZE显示计划时，分区表参数化路径剪枝计划中会出现“ppi-pruning, actual scanned:XXX”信息，ppi-pruning代表触发了参数化路径剪枝；actual scanned并不代表参数化路径剪枝的结果，在非透明多写场景下，由于每个节点剪枝结果可能不相同，所以actual scanned无法代表参数化路径剪枝结果，该字段只表示静态剪枝和动态剪枝之后的结果。

- 参数化路径动态剪枝支持的典型场景具体示例如下：

- 非透明多写特性下，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)

- 非透明多写特性下，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：

- 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)

- 非透明多写特性下，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)

- 非透明多写特性下，函数

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 JOIN t3 ON t1.c1 =
LENGTHB(t3.c1);
QUERY PLAN
-----
Streaming (type: GATHER)
  Output: t1.c1, t1.c2, t3.c1, t3.c2
  Node/s: All datanodes
  -> Hash Join
    Output: t1.c1, t1.c2, t3.c1, t3.c2
    Hash Cond: ((lengthb(t3.c1)) = t1.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
    -> Hash
      Output: t1.c1, t1.c2
      -> Partition Iterator
        Output: t1.c1, t1.c2
        Iterations: 3
        -> Partitioned Seq Scan on public.t1
          Output: t1.c1, t1.c2
          Distribute Key: t1.c1
          Selected Partitions: 1..3
(23 rows)

--删除表。
gaussdb=# DROP TABLE t1;
gaussdb=# DROP TABLE t2;
gaussdb=# DROP TABLE t3;
```

7.8.3.2 分区算子执行优化

7.8.3.2.1 Partition Iterator 算子消除

场景描述

在当前分区表架构中，执行器通过Partition Iterator算子去迭代访问每一个分区。当分区剪枝结果只有一个分区时，Partition Iterator算子就失去了迭代器的作用，在此情况下消除Partition Iterator算子，可以避免执行时产生一些不必要的开销。由于执行器的PIPELINE架构，Partition Iterator算子会重复执行，在数据量较大的场景下消除Partition Iterator算子的收益十分可观。

示例

非透明多写特性下，消除Partition Iterator算子在GUC参数partition_iterator_elimination开启后才能生效，示例如下：

```
gaussdb=# CREATE TABLE t1 (c1 INT, c2 INT) PARTITION BY HASH(c1) PARTITIONS 2 SUBPARTITION BY
HASH (c2) SUBPARTITIONS 2;
gaussdb=# INSERT INTO t1 SELECT generate_series(1,10),generate_series(1,10);
gaussdb=# SET max_datanode_for_plan = 1;

gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 = 7 AND c2 = 7;
QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
  Node/s: datanode3

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 7 AND c2 = 7
Datanode Name: datanode3
```



```
Partitioned Seq Scan on t1 (cost=0.00..42.23 rows=1 width=8)
  Filter: ((c1 = 7) AND (c2 = 7))
  Selected Partitions: 1
  Selected Subpartitions: 1:1

(10 rows)

gaussdb=# SET partition_iterator_elimination = off;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 = 7 AND c2 = 7;
QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
  Node/s: datanode3

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 7 AND c2 = 7
Datanode Name: datanode3
Partition Iterator (cost=0.00..42.23 rows=1 width=8)
  Iterations: 1, Sub Iterations: 1
  -> Partitioned Seq Scan on t1 (cost=0.00..42.23 rows=1 width=8)
    Filter: ((c1 = 7) AND (c2 = 7))
    Selected Partitions: 1
    Selected Subpartitions: 1:1

(12 rows)

gaussdb=# DROP TABLE t1;
```

注意事项及约束条件

- 只有当GUC参数partition_iterator_elimination开启，且优化器剪枝结果只有一个分区时，目标场景优化才能生效。
- 支持cplan，支持部分gplan场景，如分区键a = \$1的场景（即优化器阶段可以剪枝到一个分区的场景）。
- 支持SeqScan、Indexscan、Indexonlyscan、Bitmapscan、RowToVec、Tidscan算子。
- 非透明多写特性下，支持行存，astore/ustore存储引擎，支持SQLBypass。

7.8.3.2.2 Merge Append

场景描述

当对分区表进行全局排序时，通常SQL引擎的实现方式是先通过Partition Iterator + PartitionScan对分区表做全量扫描，然后进行Sort排序操作，这样难以利用分区局部有序性加速排序过程。针对这类问题，目前分区表支持分区归并排序执行策略，利用Merge Append的执行机制改进分区表的排序机制。

示例

非透明多写特性下，分区表Merge Append的执行机制示例如下：

```
gaussdb=# 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;
gaussdb=# INSERT INTO test_range_pt VALUES
```

```
(generate_series(1,10000),generate_series(1,10000),generate_series(1,10000));
gaussdb=# CREATE INDEX idx_range_b ON test_range_pt(b) LOCAL;
gaussdb=# ANALYZE test_range_pt;
gaussdb=# SET enable_stream_operator=off;
gaussdb=# SET max_datanode_for_plan=1;

gaussdb=# EXPLAIN SELECT * FROM test_range_pt WHERE b >10 AND b < 5000 ORDER BY b LIMIT 10;
QUERY PLAN
-----
Limit (cost=107.85..107.88 rows=10 width=12)
-> Sort (cost=107.85..120.33 rows=10 width=12)
    Sort Key: test_range_pt.b
    -> Data Node Scan on "__REMOTE_LIMIT_QUERY__" (cost=0.00..0.00 rows=10 width=12)
        Node/s: All datanodes

Remote SQL: SELECT a, b, c FROM ONLY public.test_range_pt WHERE b > 10 AND b < 5000 ORDER BY 2
LIMIT 10::bigint
Datanode Name: datanode1
Limit (cost=0.06..1.15 rows=10 width=12)
-> Result (cost=0.06..183.67 rows=10 width=12)
    -> Merge Append (cost=0.06..183.67 rows=10 width=12)
        Sort Key: b
        -> Partitioned Index Scan using idx_range_b on test_range_pt (cost=0.00..19.36 rows=336
width=12)
            Index Cond: ((b > 10) AND (b < 5000))
            Selected Partitions: 1
        -> Partitioned Index Scan using idx_range_b on test_range_pt (cost=0.00..19.36 rows=336
width=12)
            Index Cond: ((b > 10) AND (b < 5000))
            Selected Partitions: 2
        -> Partitioned Index Scan using idx_range_b on test_range_pt (cost=0.00..19.36 rows=336
width=12)
            Index Cond: ((b > 10) AND (b < 5000))
            Selected Partitions: 3
        -> Partitioned Index Scan using idx_range_b on test_range_pt (cost=0.00..19.36 rows=336
width=12)
            Index Cond: ((b > 10) AND (b < 5000))
            Selected Partitions: 4
        -> Partitioned Index Scan using idx_range_b on test_range_pt (cost=0.00..19.36 rows=336
width=12)
            Index Cond: ((b > 10) AND (b < 5000))
            Selected Partitions: 5

(28 rows)

--关闭分区表Merge Append算子。
gaussdb=# SET sql_beta_feature = 'disable_merge_append_partition';
SET

gaussdb=# EXPLAIN SELECT * FROM test_range_pt WHERE b >10 AND b < 5000 ORDER BY b LIMIT 10;
QUERY PLAN
-----
Limit (cost=107.85..107.88 rows=10 width=12)
-> Sort (cost=107.85..120.33 rows=10 width=12)
    Sort Key: test_range_pt.b
    -> Data Node Scan on "__REMOTE_LIMIT_QUERY__" (cost=0.00..0.00 rows=10 width=12)
        Node/s: All datanodes

Remote SQL: SELECT a, b, c FROM ONLY public.test_range_pt WHERE b > 10 AND b < 5000 ORDER BY 2
LIMIT 10::bigint
Datanode Name: datanode1
Limit (cost=102.89..102.91 rows=10 width=12)
-> Sort (cost=102.89..107.08 rows=10 width=12)
    Sort Key: b
    -> Partition Iterator (cost=0.00..66.62 rows=1678 width=12)
        Iterations: 5
    -> Partitioned Seq Scan on test_range_pt (cost=0.00..66.62 rows=1678 width=12)
        Filter: ((b > 10) AND (b < 5000))
        Selected Partitions: 1..5
```

```
(17 rows)
```

```
gaussdb=# DROP TABLE test_range_pt;
```

Merge Append的执行代价远小于普通执行方式。

注意事项及约束条件

- 当分区扫描路径为Index/Index Only时，才支持Merge Append执行机制。
- 当分区剪枝结果大于1时，才支持Merge Append执行机制。
- 当分区索引全部有效且为Btree索引时，才支持Merge Append执行机制。
- 当SQL含有Limit子句时，才支持Merge Append执行机制。
- 当分区扫描使用了Filter时，不支持Merge Append执行机制。
- 当GUC参数sql_beta_feature = 'disable_merge_append_partition'时，不再生成Merge Append路径。

7.8.3.2.3 Max/Min

场景描述

当对分区表使用Max/Min函数时，通常SQL引擎的实现方式是先通过Partition Iterator + PartitionScan对分区表做全量扫描，然后进行Sort + Limit操作。如果扫描分区时使用索引扫描，可以先对每个分区进行Limit操作，计算Max/Min值，最后在分区表上做Sort + Limit操作。此时在分区表上做Sort操作时，由于每个分区已经获取Max/Min值，所以Sort的数据量跟分区数相同，这样就极大的减少了Sort开销。

示例

分区表Max/Min优化示例如下：

```
gaussdb=# 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;
```

```
gaussdb=# INSERT INTO test_range_pt VALUES(generate_series(1,10000), generate_series(1,10000),
generate_series(1,10000));
```

非透明多写特性下，优化前：

```
gaussdb=# EXPLAIN ANALYZE SELECT min(b) FROM test_range_pt;
QUERY PLAN
```

```
-----
Aggregate  (cost=4.56..4.63 rows=1 width=8) (actual time=8.654..8.654 rows=1 loops=1)
->  Streaming (type: GATHER) (cost=4.56..4.63 rows=3 width=8) (actual time=7.825..8.589 rows=3
loops=1)
   Node/s: All datanodes
   -> Aggregate  (cost=4.50..4.50 rows=3 width=8) (actual time=[4.719,4.719]..[5.108,5.108], rows=3)
       -> Partition Iterator  (cost=0.00..4.46 rows=30 width=4) (actual time=[0.110,3.964]..[0.114,4.356],
rows=10000)
           Iterations: 5
           -> Partitioned Index Only Scan using idx_range_b on test_range_pt  (cost=0.00..4.46 rows=30
width=4) (actual time=[0.297,3.370]..[0.569,3.752], rows=10000)
               Heap Fetches: 0
               Selected Partitions: 1..5
```

```
Total runtime: 10.287 ms  
(10 rows)
```

非透明多写特性下，优化后：

```
gaussdb=# CREATE INDEX idx_range_b ON test_range_pt(b) LOCAL;  
gaussdb=# EXPLAIN ANALYZE SELECT min(b) FROM test_range_pt;
```

QUERY PLAN

```
-----  
Result (cost=0.64..0.65 rows=1 width=0) (actual time=2.806..2.806 rows=1 loops=1)  
  InitPlan 1 (returns $2)  
    -> Aggregate (cost=0.63..0.64 rows=1 width=4) (actual time=2.800..2.801 rows=1 loops=1)  
      -> Streaming (type: GATHER) (cost=0.56..0.63 rows=3 width=4) (actual time=2.585..2.786 rows=3  
loops=1)  
        Node/s: All datanodes  
        -> Limit (cost=0.50..0.50 rows=3 width=4) (actual time=[0.323,0.323]..[0.335,0.335], rows=3)  
          -> Sort (cost=0.50..0.52 rows=3 width=4) (actual time=[0.322,0.322]..[0.334,0.334], rows=3)  
            Sort Key: public.test_range_pt.b  
            Sort Method: top-N heapsort  Memory: 25kB ~ 25kB  
            -> Partition Iterator (cost=0.00..0.46 rows=15 width=4) (actual time=[0.081,0.294]..  
[0.084,0.304], rows=15)  
              Iterations: 5  
              -> Limit (cost=0.00..0.46 rows=3 width=4) (actual time=[0.252,0.253]..  
[0.257,0.257], rows=15)  
                -> Partitioned Index Only Scan using idx_range_b on test_range_pt  
(cost=0.00..4.50 rows=3 width=4) (actual time=[0.252,0.252]..[0.255,0.255], rows=15)  
                  Index Cond: (b IS NOT NULL)  
                  Heap Fetches: 0  
                  Selected Partitions: 1..5  
Total runtime: 4.478 ms  
(17 rows)
```

优化后时间消耗远小于优化前。

```
--删除表。  
gaussdb=# DROP TABLE test_range_pt;
```

优化后时间消耗远小于优化前。

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

注意事项及约束条件

- 当分区扫描路径为Index、Index Only时，才支持Max/Min优化。
- 当分区索引全部有效且为Btree索引时，才支持Max/Min优化。

7.8.3.2.4 动态剪枝性能优化

场景描述

当分区键带有等值查询条件时，扫描的分区数量可以裁剪到单分区中，数据库引擎可直接通过分区路由映射到目标分区获取分区信息，实现精确分区定位，提升动态剪枝的性能，从而降低扫描算子初始化的底噪。

说明

动态剪枝性能优化不会在计划上体现。

示例-非透明多写

```
gaussdb=# CREATE TABLE t_list  
(
```

```
c1 INT,
c2 INT
)
PARTITION BY LIST (c1)
(
    PARTITION p0 VALUES (1,2,3),
    PARTITION p1 VALUES (4,5,6),
    PARTITION p2 VALUES (7,8,9),
    PARTITION p3 VALUES (DEFAULT)
);
-- gplan裁剪到单分区走动态剪枝性能优化。
gaussdb=# SET plan_cache_mode = 'force_generic_plan';
gaussdb=# PREPARE p1 AS SELECT /*+tablescan(t_list)*/ c2 FROM t_list WHERE c1=$1;
gaussdb=# EXPLAIN (verbose off, costs off) EXECUTE p1(1);
          QUERY PLAN
-----
Data Node Scan
Node/s: datanode1
Node expr: $1

Remote SQL: SELECT/*+ TableScan(t_list)*/ c2 FROM public.t_list WHERE c1 = $1
Datanode Name: datanode1
Partitioned Seq Scan on t_list
Filter: (c1 = $1)
Selected Partitions: 1

(10 rows)
gaussdb=# DROP TABLE t_list;
```

注意事项及约束条件

- 仅支持分区键带等值条件，且裁剪到单分区的场景。
- 仅支持gplan使用动态剪枝，cplan不支持动态剪枝。
- 仅支持单列分区键，不支持多列分区键。
- 支持SeqScan、IndexScan、IndexonlyScan以及BitmapScan算子。
- 不支持SQLBypass场景。
- 不支持表达式分区键。

7.8.3.3 分区索引

分区表上的索引共有三种类型：

- Global Non-Partitioned Index，如图7-8所示。
- Global Partitioned Index，如图7-9所示。
- Local Partitioned Index，如图7-10所示。

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

图 7-8 Global Non-Partitioned Index

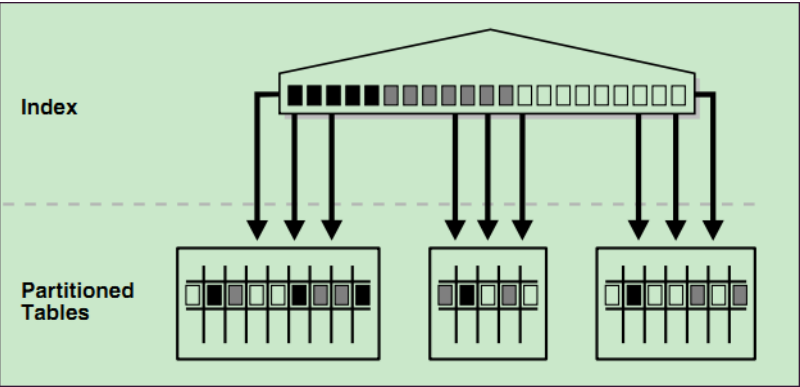


图 7-9 Global Partitioned Index

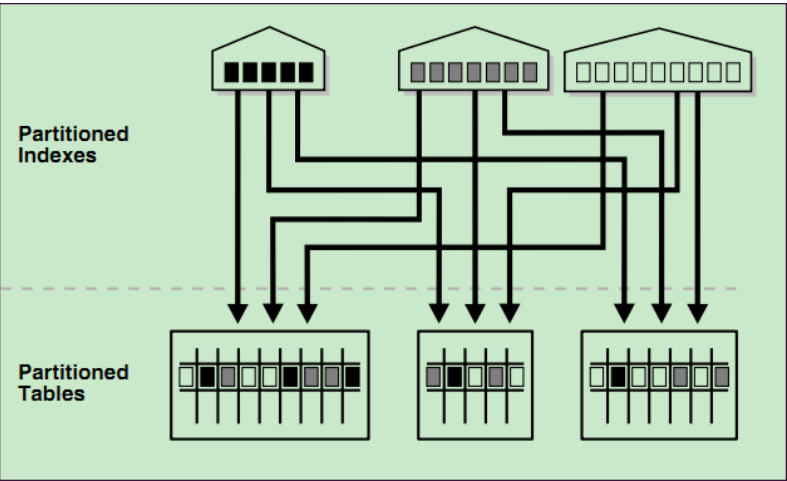
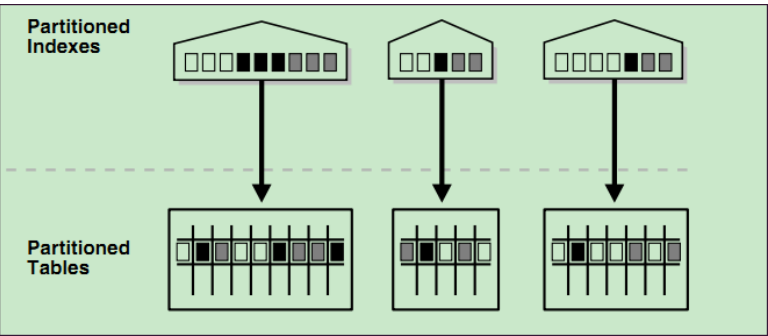


图 7-10 Local Partitioned Index



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。
- 在创建索引时，索引类型的选择取决于唯一约束和主键约束的设置。若约束键涵盖了所有分区键，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，将索引的创建范围指定为单个分区。如此创建的索引仅在指定分区

内有效，在其他分区不产生作用，并且后续新增分区时，也不会自动为这些新分区创建该索引。

- 目前只有在静态剪枝到单个分区的计划中，才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
```

- 删除表

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

7.8.3.4 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完成静态剪枝后，扫描范围被缩减至单个分区的应用场景。其支持收集的统计信息范围如下：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。

7.8.3.4.1 级联收集统计信息

在对分区表执行ANALYZE | ANALYSE语法时，系统会根据用户指定或默认的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《参考》中“SQL参考 > SQL语法 > A > ANALYZE | ANALYSE”中的PARTITION_MODE参数。

须知

- 非透明多写特性下，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	3	15
range_p01	p	3	12
range_p02	p	2	10
range_p03	p	0	0
range_p04	p	3	15
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_nddistinct | 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_distinct,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_distinct,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;
```

7.8.3.4.2 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

gaussdb=# \x

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

[RECORD 1]

+-----

```

schemaname          | public
tablename            | only_firsr_part
attname              | name
inherited             | f
null_frac            | 0
avg_width             | 5
n_distinct            | 1
n_dndistinct         | 1
most_common_vals      | {test}
most_common_freqs     | {1}
histogram_bounds      |
correlation           | 1
most_common_elems     |
most_common_elem_freqs |
elem_count_histogram  |
partitionname         | id11
subpartitionname      |
-[ RECORD 2 ]-----

```

-[RECORD 2]-----

schemaname	public
tablename	only_fisrt_part
attname	id
inherited	f
null_frac	0
avg_width	4
n_distinct	-1
n_dndistinct	1
most_common_vals	
most_common_freqs	
histogram_bounds	
{1,50,100,150,200,250,300,350,400,450,500,550,600,650,700,750,800,850,900,950,1000,1050,1100,1150,1200,1250,1300,1350,1400,1450,1500,1550,1600,1650,1700,1750,1800,1850,1900,1950,2000,2050,2100,2150,2200,2250,2300,2350,2400,2450,2500,2550,2600,2650,2700,2750,2800,2850,2900,2950,3000,3050,3100,3150,3200,3250,3300,3350,3400,3450,3500,3550,3600,3650,3700,3750,3800,3850,3900,3950,4000,4050,4100,4150,4200,4250,4300,4350,4400,4450,4500,4550,4600,4650,4700,4750,4800,4850,4900,4950,5000}	
correlation	1
most_common_elems	
most_common_elem_freqs	
elem_count_histogram	
partitionname	id11
subpartitionname	

```
gaussdb=# q \x
-- 删除分区表
gaussdb=# DROP TABLE only_fisrt_part;
```

优化器使用指定分区统计信息

优化器在处理分区表时，会优先采用指定分区的统计信息来进行查询优化。但是如果指定分区尚未收集统计信息，此时优化器会通过改写分区子句的方式开展剪枝优化工作，详情参见[通过改写分区子句剪枝优化](#)。

```
gaussdb=# SET enable_fast_query_shipping = off;
gaussdb=# CREATE TABLE ONLY_FIRST_PART_TWO
(
    C1 INT,
    C2 BIGINT
)
PARTITION BY RANGE(C1)
(
    PARTITION P_1 VALUES LESS THAN (1000),
    PARTITION P_2 VALUES LESS THAN (3000),
    PARTITION P_3 VALUES LESS THAN (MAXVALUE)
);

gaussdb=# INSERT INTO only_first_part_two SELECT generate_series(1,5000), 0;
gaussdb=# EXPLAIN SELECT * FROM only_first_part_two PARTITION (p_2);
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.81..5.35 rows=20 width=12)
Node/s: All datanodes
-> Partitioned Seq Scan on only_first_part_two (cost=0.00..4.13 rows=20 width=12)
    Selected Partitions: 2
(4 rows)

gaussdb=# EXPLAIN SELECT * FROM only_first_part_two PARTITION (p_1) where c2 = 2;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..4.26 rows=1 width=12)
Node/s: All datanodes
-> Partitioned Seq Scan on only_first_part_two (cost=0.00..4.16 rows=1 width=12)
    Filter: (c2 = 2)
    Selected Partitions: 1
(5 rows)

gaussdb=# DROP TABLE only_first_part_two;
```

通过改写分区子句剪枝优化

当缺少分区级统计信息时，优化器可以通过在逻辑上对分区子句进行伪谓词改写，利用改写后的伪谓词来影响选择率的计算，并结合整表的统计信息，以获得更为准确的行数估算值。

说明

- 只作用于选择率的计算。
- 非透明多写特性下，不支持二级分区。
- 非透明多写特性下，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：非透明多写特性下，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;

gaussdb=# set max_datanode_for_plan=2;
gaussdb=# set enable_stream_operator=off;

-- 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 ONLY public.test_int4_maxvalue PARTITION(id1) WHERE true
Datanode Name: datanode1
    Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..28.61 rows=2561 width=9)
        Selected Partitions:  1

Datanode Name: datanode2
    Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..27.39 rows=2439 width=9)
        Selected Partitions:  1

(12 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 ONLY public.test_int4_maxvalue PARTITION(max_id) WHERE true
Datanode Name: datanode1
    Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..28.61 rows=2561 width=9)
        Selected Partitions:  3

Datanode Name: datanode2
    Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..27.39 rows=2439 width=9)
        Selected Partitions:  3

(12 rows)

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

示例：非透明多写特性下，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 ONLY public.test_default PARTITION(p_1) WHERE true
Datanode Name: datanode1
Partitioned Seq Scan on test_default (cost=0.00..11.16 rows=1016 width=12)
Selected Partitions: 1

Datanode Name: datanode2
Partitioned Seq Scan on test_default (cost=0.00..10.84 rows=984 width=12)
Selected Partitions: 1

(12 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 ONLY public.test_default PARTITION(p_3) WHERE true
Datanode Name: datanode1
Partitioned Seq Scan on test_default (cost=0.00..11.16 rows=1016 width=12)
Selected Partitions: 3

Datanode Name: datanode2
Partitioned Seq Scan on test_default (cost=0.00..10.84 rows=984 width=12)
Selected Partitions: 3

(12 rows)

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

7.8.3.5 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查询性能。

Partition-wise Join分为SMP（Symmetric Multi-Processing，对称多处理）场景和非SMP场景。

7.8.3.5.1 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。
- 需要将query_dop的值设置为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。

- 非透明多写特性下，仅支持FQS计划。

示例

非透明多写特性下：

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

--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 on "__REMOTE_FQS_QUERY__"
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 on "__REMOTE_FQS_QUERY__"
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;
```

7.8.3.5.2 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。

SMP场景下的Partition-wise Join分为Full Partition-wise Join和Partial Partition-wise Join两种方式：

- 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。
- 非透明多写特性下，仅支持Stream计划。
- 非透明多写特性下，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Nest Loop 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。
- 其他规格继承Stream计划和SMP场景的规格。

约束

- 不支持对range分布的中间结果表的Full Partition-wise Join。
- 剪枝到单分区场景下不支持Partition-wise Join。
- 仅支持在同一个Node Group下的Partition-wise Join。
- 不支持bucket表、临时表下的Partition-wise Join。
- 对分区指定DN的场景下，不支持Partition-wise Join。
- 不支持二级分区的Partition-wise Join。
- 仅支持Stream 计划，不支持light-proxy、FQS和PGXC计划。
- 在数据倾斜、分区剪枝等场景下，可能导致执行性能劣化。
- 不支持计算倾斜下的Partition-wise Join。

示例

非透明多写特性下：

```
--创建Hash分区表，相同分布分区键。
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

--创建Hash分区表，不同分布分区键。
gaussdb=# CREATE TABLE hash_part_diff_key
(
    a INTEGER,
    b INTEGER,
    c INTEGER
)
DISTRIBUTE BY HASH(a)
PARTITION BY HASH(b)
(
    PARTITION p1,
    PARTITION p2,
    PARTITION p3,
    PARTITION p4,
    PARTITION p5
);
CREATE TABLE

--创建Range分区表，不同分布分区键。
gaussdb=# CREATE TABLE range_part_diff_key
(
    a INTEGER,
    b INTEGER,
    c INTEGER,
    d INTEGER,
    e INTEGER
) DISTRIBUTE BY RANGE(a) (
    SLICE p1 VALUES LESS THAN (100),
```

```
SLICE p2 VALUES LESS THAN (200),
SLICE p3 VALUES LESS THAN (300),
SLICE p4 VALUES LESS THAN (400),
SLICE p5 VALUES LESS THAN (500)
)
PARTITION BY RANGE(b) (
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (200),
    PARTITION p3 VALUES LESS THAN (300),
    PARTITION p4 VALUES LESS THAN (400),
    PARTITION p5 VALUES LESS THAN (500)
);
CREATE TABLE

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

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

--不走nest loop计划。
gaussdb=# SET enable_material=off;
SET
gaussdb=# SET enable_broadcast=off;
SET
gaussdb=# SET enable_nestloop=off;
SET

-- 插入10万条基础测试数据（a/b/c为随机整数，保证哈希分布的随机性）。
gaussdb=# INSERT INTO hash_part (a, b, c)
SELECT
    FLOOR(RANDOM() * 1000000)::INTEGER AS a,
    FLOOR(RANDOM() * 500000)::INTEGER AS b,
    FLOOR(RANDOM() * 1000)::INTEGER AS c
FROM GENERATE_SERIES(1, 100000);
INSERT 0 100000

-- 循环5次，每次将现有数据翻倍插入（最终总数据量 = 10万 * 2^5 = 320万）。
gaussdb=# DO $$
BEGIN
    FOR i IN 1..5 LOOP
        INSERT INTO hash_part (a, b, c)
        SELECT a, b, c FROM hash_part;
    END LOOP;
END
$$;
ANONYMOUS BLOCK EXECUTE

gaussdb=# INSERT INTO hash_part_diff_key(a, b, c)
SELECT
    FLOOR(RANDOM() * 1000000)::INTEGER AS a,
    FLOOR(RANDOM() * 500000)::INTEGER AS b,
    FLOOR(RANDOM() * 1000)::INTEGER AS c
FROM GENERATE_SERIES(1, 100000);
INSERT 0 100000

gaussdb=# DO $$
BEGIN
    FOR i IN 1..5 LOOP
        INSERT INTO hash_part_diff_key (a, b, c)
        SELECT a, b, c FROM hash_part_diff_key;
    END LOOP;
END
$$;
ANONYMOUS BLOCK EXECUTE
```

```
gaussdb=# INSERT INTO range_part_diff_key(a, b, c)
SELECT
  FLOOR(RANDOM() * 500)::INTEGER AS a,
  FLOOR(RANDOM() * 500)::INTEGER AS b,
  FLOOR(RANDOM() * 500)::INTEGER AS c
FROM GENERATE_SERIES(1, 100000);
INSERT 0 100000

gaussdb=# DO $$
BEGIN
  FOR i IN 1..5 LOOP
    INSERT INTO range_part_diff_key(a, b, c)
    SELECT a, b, c FROM range_part_diff_key;
  END LOOP;
END
$$;
ANONYMOUS BLOCK EXECUTE

--analyze数据。
gaussdb=# ANALYZE;
ANALYZE

--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
  -> Hash Join
    Hash Cond: (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
    -> Hash
      -> 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计划消除掉了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)
```

--查看Partition-wise Join的执行计划。从计划中可以看出，Partition-wise Join计划消除掉了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。hash_part_diff_key表的分布键和分区键都是Join Clause的子集，且一一对应，可以做Full Partition-wise Join。

```
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM hash_part_diff_key t1, hash_part_diff_key t2 WHERE t1.a = t2.a AND t1.b = t2.b;
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) AND (t1.b = t2.b))
        -> Partition Iterator
            Iterations: 5
            -> Partitioned Seq Scan on hash_part_diff_key t1
                Selected Partitions: 1..5
        -> Hash
            -> Partition Iterator
                Iterations: 5
                -> Partitioned Seq Scan on hash_part_diff_key t2
                    Selected Partitions: 1..5
(15 rows)
```

--查看Partition-wise Join的执行计划。从计划中可以看出，Partition-wise Join计划消除掉了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。range_part_diff_key表的分布键和分区键都是Join Clause的子集，且一一对应，且range中每个slice和partition的范围都一致，可以做Full Partition-wise Join。

```
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part_diff_key t1, range_part_diff_key t2 WHERE t1.a = t2.a AND t1.b = t2.b;
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) AND (t1.b = t2.b))
        -> Partition Iterator
            Iterations: 5
            -> Partitioned Seq Scan on range_part_diff_key t1
                Selected Partitions: 1..5
        -> Hash
            -> Partition Iterator
                Iterations: 5
                -> Partitioned Seq Scan on range_part_diff_key t2
                    Selected Partitions: 1..5
(15 rows)
```

-- range分布的中间结果表不再支持Partition-wise Join。

```
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part_diff_key t1, range_part_diff_key t2 ,
range_part_diff_key t3 WHERE t1.a = t2.a AND t1.b = t2.b AND t1.a = t3.a AND t1.b = t3.b;
QUERY PLAN
-----
Streaming (type: GATHER)
Node/s: All datanodes
-> Streaming(type: LOCAL GATHER dop: 1/5)
```

```

Spawn on: All datanodes
-> Hash Join
  Hash Cond: ((t1.a = t3.a) AND (t1.b = t3.b))
  -> Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
    Spawn on: All datanodes
    -> Hash Join (Partition-wise Join)
      Hash Cond: ((t1.a = t2.a) AND (t1.b = t2.b))
      -> Partition Iterator
        Iterations: 5
        -> Partitioned Seq Scan on range_part_diff_key t1
          Selected Partitions: 1..5
      -> Hash
        -> Partition Iterator
          Iterations: 5
          -> Partitioned Seq Scan on range_part_diff_key t2
            Selected Partitions: 1..5
    -> Hash
      -> Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
        Spawn on: All datanodes
        -> Partition Iterator
          Iterations: 5
          -> Partitioned Seq Scan on range_part_diff_key t3
            Selected Partitions: 1..5
(26 rows)

-- 删除分区表。
gaussdb=# DROP TABLE hash_part;
gaussdb=# DROP TABLE hash_part_diff_key;
gaussdb=# DROP TABLE range_part_diff_key;

```

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。
- Partition-wise Join计划的执行性能取决于数据量最大的分区的执行性能。所以在分区间存在严重的数据倾斜或存在分区剪枝的场景下，Partition-wise Join可能导致性能劣化。

7.8.3.5.3 SMP 场景下的 Partial Partition-wise Join

Partial Partition-wise Join是指相互Join的两张表中有一张表是分区表，另一张表可以为任意类型，在任意类型的这张表的上层需要增加一个Stream Redistribute算子，将数据分发后与分区表一侧进行匹配。Partial Partition-wise Join路径生成的条件是分区表的分区键是一个Join key。

使用规格

继承SMP场景下的Full Partition-wise Join的使用规格。

约束

- 不支持对range分布的中间结果表的Partial Partition-wise Join。
- 其他约束继承SMP场景下的Full Partition-wise Join的约束。

示例

非透明多写特性下：

```

--创建Hash分区表，相同分布分区键。
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

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

--不走nest loop计划。
gaussdb=# SET enable_material=off;
SET
gaussdb=# SET enable_broadcast=off;
SET
gaussdb=# SET enable_nestloop=off;
SET

-- 插入10万条基础测试数据（a/b/c为随机整数，保证哈希分布的随机性）。
gaussdb=# INSERT INTO hash_part (a, b, c)
SELECT
    FLOOR(RANDOM() * 1000000)::INTEGER AS a,
    FLOOR(RANDOM() * 500000)::INTEGER AS b,
    FLOOR(RANDOM() * 1000)::INTEGER AS c
FROM GENERATE_SERIES(1, 100000);
INSERT 0 100000

-- 循环5次，每次将现有数据翻倍插入（最终总数据量 = 10万 * 2^5 = 320万）。
gaussdb=# DO $$
BEGIN
    FOR i IN 1..5 LOOP
        INSERT INTO hash_part (a, b, c)
        SELECT a, b, c FROM hash_part;
    END LOOP;
END
$$;
ANONYMOUS BLOCK EXECUTE

--analyze数据。
gaussdb=# ANALYZE;
ANALYZE

--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)和Streaming(type: SPLIT REDISTRIBUTE)
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 INNER JOIN hash_part t2 ON (t1.a = t2.b);
QUERY PLAN

-----
Streaming (type: GATHER)
Node/s: All datanodes
-> Streaming(type: LOCAL GATHER dop: 1/5)
    Spawn on: All datanodes
    -> Hash Join
        Hash Cond: (t2.b = t1.a)
    -> Streaming(type: SPLIT REDISTRIBUTE dop: 5/5)
        Spawn on: All datanodes
        -> Partition Iterator
            Iterations: 5
```

```
-> Partitioned Seq Scan on hash_part t2
    Selected Partitions: 1..5
-> Hash
    -> 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
(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，Partial Partition-wise Join计划消除掉了分布表
hash_part一侧的Streaming算子，即分区表的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.b;
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: (t2.b = t1.a)
        -> Streaming(type: SPLIT REDISTRIBUTE dop: 5/5)
            Spawn on: All datanodes
            -> Partition Iterator
                Iterations: 5
            -> Partitioned Seq Scan on hash_part t2
                Selected Partitions: 1..5
    -> Hash
        -> Partition Iterator
            Iterations: 5
        -> Partitioned Seq Scan on hash_part t1
            Selected Partitions: 1..5
(17 rows)

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

说明

Partition-wise Join计划的执行性能取决于数据量最大的分区的执行性能。所以在分区间存在严重的数据倾斜或存在分区剪枝的场景下，Partition-wise Join可能导致性能劣化。

7.8.4 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。

- 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。

须知

- 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。
 - 大部分分区DDL支持partition/subpartition和partition/subpartition 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。
 - 指定分区语句目前不能走全局索引扫描。
-
- 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：非透明多写特性下，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。

7.8.4.1 新增分区

用户可以在已建立的分表中新增分区，来维护新业务的进行。单个表的最大分区数为1048575（包括一级分区和二级分区）。在默认配置下，建议单个表的分区数不超过400个，具体请参见[创建分区表](#)章节。

须知

- 新增分区不能作用于HASH分区上。
- 非透明多写特性下，新增分区不继承表上的分类索引属性。如果分类索引建立在二级分区表的一级分区上，则在该一级分区下新增二级分区时，新增的二级分区继承分类索引。

7.8.4.1.1 范围分区表新增分区

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

例如，对范围分区表range_sales新增一个分区。

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

须知

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

7.8.4.1.2 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。

7.8.4.1.3 二级分区表新增一级分区

使用ALTER TABLE ADD PARTITION可以在二级分区表中新增一个一级分区，这个操作可以作用在一级分区策略为RANGE或者LIST的情况。如果这个新增一级分区下声明了二级分区定义，则数据库会根据定义创建对应的二级分区；如果这个新增一级分区下没有声明二级分区定义，则数据库会自动创建一个默认的二级分区。

例如，对二级分区表range_list_sales新增一个一级分区，并在下面创建四个二级分区。

```
ALTER TABLE range_list_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01')
TABLESPACE tb1
(
  SUBPARTITION date_202005_channel1 VALUES ('0', '1', '2'),
  SUBPARTITION date_202005_channel2 VALUES ('3', '4', '5') TABLESPACE tb2,
  SUBPARTITION date_202005_channel3 VALUES ('6', '7'),
  SUBPARTITION date_202005_channel4 VALUES ('8', '9')
);
```

或者只对二级分区表range_list_sales进行新增一级分区操作。

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

上面这种行为与如下SQL语句等价。

```
ALTER TABLE range_list_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01')
TABLESPACE tb1
(
  SUBPARTITION date_202005_channel1 VALUES (DEFAULT)
);
```

须知

当二级分区表的一级分区策略为HASH时，不支持通过ALTER TABLE ADD PARTITION命令新增一级分区。

7.8.4.1.4 二级分区表新增二级分区

使用ALTER TABLE MODIFY PARTITION ADD SUBPARTITION可以在二级分区表中新增一个二级分区，这个行为可以作用在二级分区策略为RANGE或者LIST的情况。

例如，对二级分区表range_list_sales的date_202004新增一个二级分区。

```
ALTER TABLE range_list_sales MODIFY PARTITION date_202004 ADD SUBPARTITION date_202004_channel5  
VALUES ('X') TABLESPACE tb2;
```

须知

当二级分区表的二级分区策略为HASH时，不支持通过ALTER TABLE MODIFY PARTITION ADD SUBPARTITION命令新增二级分区。

7.8.4.2 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。

须知

- 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。
- 非透明多写特性下，删除分区时，如果该分区上带有任何属于当前分区的分类索引时，则会级联删除分类索引。

7.8.4.2.1 对一级分区表删除分区

使用ALTER TABLE DROP PARTITION可以删除指定一级分区表的任何一个分区，这个操作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。

7.8.4.2.2 对二级分区表删除一级分区

使用ALTER TABLE DROP PARTITION可以删除二级分区表的一个一级分区，这个操作可以作用在一级分区策略为RANGE或者LIST的情况。数据库会将这个一级分区，以及一级分区下的所有二级分区都删除。

例如，通过指定分区名删除二级分区表range_list_sales的一级分区date_202005，并更新Global索引。

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

或者，通过指定分区值来删除二级分区表range_list_sales中('2020-05-08')所对应的一级分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

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

须知

- 当二级分区表只有一个一级分区时，不支持通过ALTER TABLE DROP PARTITION命令删除一级分区。
- 当二级分区表的一级分区策略为HASH时，不支持通过ALTER TABLE DROP PARTITION命令删除一级分区。

7.8.4.2.3 对二级分区表删除二级分区

使用ALTER TABLE DROP SUBPARTITION可以删除二级分区表的一个二级分区，这个操作可以作用在二级分区策略为RANGE或者LIST的情况。

例如，通过指定分区名删除二级分区表range_list_sales的二级分区date_202005_channel1，并更新Global索引。

```
ALTER TABLE range_list_sales DROP SUBPARTITION date_202005_channel1 UPDATE GLOBAL INDEX;
```

或者，通过指定分区值来删除二级分区表range_list_sales中('2020-05-08', '0')所对应的二级分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

```
ALTER TABLE range_list_sales DROP SUBPARTITION FOR ('2020-05-08', '0');
```

须知

- 当二级分区表所删除的目标分区只有一个二级分区时，不支持通过ALTER TABLE DROP SUBPARTITION命令删除二级分区。
- 当二级分区表的二级分区策略为HASH时，不支持通过ALTER TABLE DROP SUBPARTITION命令删除二级分区。

7.8.4.3 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。在业务迁移的场景下，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，可以声明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，可以声明WITHOUT VALIDATION来提高交换性能。
- 可以声明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，则根据声明子句的不同，存在以下三种行为：

- 声明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。
- 声明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。
- 声明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')
```

须知

如果交换的数据不完全属于目标分区，请不要声明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。

7.8.4.3.1 对一级分区表交换分区

使用ALTER TABLE EXCHANGE PARTITION可以对一级分区表交换分区。

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;
```

7.8.4.3.2 对二级分区表交换二级分区

使用ALTER TABLE EXCHANGE SUBPARTITION可以对二级分区表交换二级分区。

例如，通过指定分区名将二级分区表range_list_sales的二级分区date_202001_channel1和普通表exchange_sales进行交换，不进行分区校验，并更新Global索引。

```
ALTER TABLE range_list_sales EXCHANGE SUBPARTITION (date_202001_channel1) WITH TABLE  
exchange_sales WITHOUT VALIDATION UPDATE GLOBAL INDEX;
```

或者，通过指定分区值将二级分区表range_list_sales中('2020-01-08', '0')所对应的二级分区和普通表exchange_sales进行交换、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_list_sales EXCHANGE SUBPARTITION FOR ('2020-01-08', '0') WITH TABLE  
exchange_sales WITH VALIDATION VERBOSE;
```

须知

不支持对二级分区表的一级分区交换分区。

7.8.4.4 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。

须知

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。

7.8.4.4.1 对一级分区表清空分区

使用ALTER TABLE TRUNCATE PARTITION可以清空指定一级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');
```

7.8.4.4.2 对二级分区表清空一级分区

使用ALTER TABLE TRUNCATE PARTITION可以清空二级分区表的一个一级分区，数据库会将这个一级分区下的所有二级分区都进行清空。

例如，通过指定分区名清空二级分区表range_list_sales的一级分区date_202005，并更新Global索引。

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

或者，通过指定分区值来清空二级分区表range_list_sales中('2020-05-08')所对应的一级分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

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

7.8.4.4.3 对二级分区表清空二级分区

使用ALTER TABLE TRUNCATE SUBPARTITION可以清空二级分区表的一个二级分区。

例如，通过指定分区名清空二级分区表range_list_sales的二级分区date_202005_channel1，并更新Global索引。

```
ALTER TABLE range_list_sales TRUNCATE SUBPARTITION date_202005_channel1 UPDATE GLOBAL INDEX;
```

或者，通过指定分区值来清空二级分区表range_list_sales中('2020-05-08', '0')所对应的二级分区。由于不带UPDATE GLOBAL INDEX子句，执行该命令后Global索引会失效。

```
ALTER TABLE range_list_sales TRUNCATE SUBPARTITION FOR ('2020-05-08', '0');
```

7.8.4.5 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。

须知

- 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。如果分类索引建立在二级分区表的一级分区上，则支持分割其二级分区。
- 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，这会影响分割期间对源分区p1查询，具体请参见[DQL/DML-DDL并发](#)。

7.8.4.5.1 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'), --第一个分区上界是'2020-01-11'
    PARTITION date_202001_p2 VALUES LESS THAN ('2020-01-21'), --第二个分区上界是'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，前面几个分区不能声明MAXVALUE范围，最后一个分区会继承MAXVALUE分区范围。

7.8.4.5.2 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'), --第一个分区范围是('6')
    PARTITION channel2_2 VALUES ('8'), --第二个分区范围是('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;
```

须知

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

7.8.4.5.3 对*-RANGE 二级分区表分割二级分区

使用ALTER TABLE SPLIT SUBPARTITION可以对*-RANGE二级分区表分割二级分区。

例如，假设*-RANGE二级分区表list_range_sales的二级分区channel1_customer4的定义范围为[1000, MAXVALUE)。可以指定分割点1200将二级分区channel1_customer4分割为两个分区，并更新Global索引。

```
ALTER TABLE list_range_sales SPLIT SUBPARTITION channel1_customer4 AT (1200) INTO
(
  SUBPARTITION channel1_customer4_p1, --第一个分区上界是1200
  SUBPARTITION channel1_customer4_p2 --第二个分区上界是MAXVALUE
) UPDATE GLOBAL INDEX;
```

或者，不指定分割点，将二级分区channel1_customer4分割为多个分区，并更新Global索引。

```
ALTER TABLE list_range_sales SPLIT SUBPARTITION channel1_customer4 INTO
(
  SUBPARTITION channel1_customer4_p1 VALUES LESS THAN (1200), --第一个分区上界是1200
  SUBPARTITION channel1_customer4_p2 VALUES LESS THAN (1400), --第二个分区上界是1400
  SUBPARTITION channel1_customer4_p3 --第三个分区上界是MAXVALUE
) 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_range_sales SPLIT SUBPARTITION FOR ('1', 1200) AT (1200) INTO
(
  PARTITION channel1_customer4_p1, -- 第一个分区上界是1200
  PARTITION channel1_customer4_p2 -- 第二个分区上界是MAXVALUE
) UPDATE GLOBAL INDEX;
```

须知

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

7.8.4.5.4 对*-LIST 二级分区表分割二级分区

使用ALTER TABLE SPLIT SUBPARTITION可以对*-LIST二级分区表分割二级分区。

例如，假设*-LIST二级分区表hash_list_sales的二级分区product2_channel2的定义范围为DEFAULT。可以指定分割点将其分割为两个分区，并更新Global索引。

```
ALTER TABLE hash_list_sales SPLIT SUBPARTITION product2_channel2 VALUES ('6', '7', '8', '9') INTO
(
  SUBPARTITION product2_channel2_p1, --第一个分区范围是('6', '7', '8', '9')
  SUBPARTITION product2_channel2_p2 --第二个分区范围是DEFAULT
) UPDATE GLOBAL INDEX;
```

或者，不指定分割点，将二级分区product2_channel2分割为多个分区，并更新Global索引。

```
ALTER TABLE hash_list_sales SPLIT SUBPARTITION product2_channel2 INTO
(
  SUBPARTITION product2_channel2_p1 VALUES ('6', '7', '8'), --第一个分区范围是('6', '7', '8')
  SUBPARTITION product2_channel2_p2 VALUES ('9', '10'), --第二个分区范围是('9', '10')
  SUBPARTITION product2_channel2_p3 --第三个分区范围是DEFAULT
) 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 hash_list_sales SPLIT SUBPARTITION FOR (1200, '6') VALUES ('6', '7', '8', '9') INTO
(
  SUBPARTITION product2_channel2_p1, --第一个分区范围是('6', '7', '8', '9')
  SUBPARTITION product2_channel2_p2 --第二个分区范围是DEFAULT
) UPDATE GLOBAL INDEX;
```

须知

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

7.8.4.6 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。

须知

- 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。如果分类索引建立在二级分区表的一级分区上，则支持合并其二级分区。
- 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。

7.8.4.6.1 对一级分区表合并分区

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;
```

7.8.4.6.2 对二级分区表合并二级分区

使用ALTER TABLE MERGE SUBPARTITIONS可以将多个二级分区合并为一个分区。

例如，将二级分区表hash_list_sales的二级分区product1_channel1、product1_channel2、product1_channel3合并为一个新的分区，并更新Global索引。

```
ALTER TABLE hash_list_sales MERGE SUBPARTITIONS product1_channel1, product1_channel2,  
product1_channel3 INTO  
SUBPARTITION product1_channel1 UPDATE GLOBAL INDEX;
```

7.8.4.7 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。

7.8.4.7.1 对一级分区表移动分区

使用ALTER TABLE MOVE PARTITION可以对一级分区表设置移动分区。

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;`

7.8.4.7.2 对二级分区表移动二级分区

使用ALTER TABLE MOVE SUBPARTITION可以对二级分区表设置移动二级分区。

例如，通过指定分区名将二级分区表range_list_sales的二级分区date_202001_channel1移动到表空间tb1中。

```
ALTER TABLE range_list_sales MOVE SUBPARTITION date_202001_channel1 TABLESPACE tb1;
```

或者，通过指定分区值将二级分区表range_list_sales中('2020-01-08', '0')所对应的二级分区移动到表空间tb1中。

```
ALTER TABLE range_list_sales MOVE SUBPARTITION FOR ('2020-01-08', '0') TABLESPACE tb1;
```

7.8.4.8 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。

7.8.4.8.1 对一级分区表重命名分区

使用ALTER TABLE RENAME PARTITION可以对一级分区表重命名分区。

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;
```

7.8.4.8.2 对二级分区表重命名一级分区

使用ALTER TABLE RENAME PARTITION可以对二级分区表重命名一级分区。

具体方法与一级分区表相同。

7.8.4.8.3 对二级分区表重命名二级分区

使用ALTER TABLE RENAME SUBPARTITION可以对二级分区表重命名二级分区。

例如，通过指定分区名将二级分区表range_list_sales的二级分区date_202001_channel1重命名。

```
ALTER TABLE range_list_sales RENAME SUBPARTITION date_202001_channel1 TO date_202001_channelnew;
```

或者，通过指定分区值将二级分区表range_list_sales中('2020-01-08', '0')所对应的二级分区重命名。

```
ALTER TABLE range_list_sales RENAME SUBPARTITION FOR ('2020-01-08', '0') TO date_202001_channelnew;
```

7.8.4.8.4 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。

7.8.4.9 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';
```

7.8.4.10 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。

7.8.4.10.1 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;
```

7.8.4.10.2 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。这个语法如果作用于二级分区表的一级分区，数据库会同时对这个一级分区下的所有二级分区进行设置。
- 使用ALTER TABLE MODIFY SUBPARTITION可以设置二级分区表上指定二级分区的所有索引分区是否可用。

例如，假设分区表range_sales上存在两个Local索引：range_sales_idx1和range_sales_idx2，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示例：

- 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;
```

假设二级分区表list_range_sales上存在两个Local索引list_range_sales_idx1和list_range_sales_idx2，表下有一级分区channel1，其下属二级分区有channel1_product1、channel1_product2、channel1_product3，二级分区channel1_product1上对应的索引分区名分别为channel1_product1_idx1和channel1_product1_idx2。

下面给出了维护二级分区表一级分区索引的语法：

- 可以通过如下命令设置分区channel1下属二级分区的所有索引分区均不可用，包括二级分区channel1_product1、channel1_product2、channel1_product3。

```
ALTER TABLE list_range_sales MODIFY PARTITION channel1 UNUSABLE LOCAL INDEXES;
```
- 可以通过如下命令重建分区channel1下属二级分区的所有索引分区。

```
ALTER TABLE list_range_sales MODIFY PARTITION channel1 REBUILD UNUSABLE LOCAL INDEXES;
```

下面给出了维护二级分区表二级分区索引的语法：

- 可以通过如下命令单独设置二级分区channel1_product1上的所有索引分区均不可用。

```
ALTER TABLE list_range_sales MODIFY SUBPARTITION channel1_product1 UNUSABLE LOCAL INDEXES;
```
- 可以通过如下命令重建二级分区channel1_product1上的所有索引分区。

```
ALTER TABLE list_range_sales MODIFY SUBPARTITION channel1_product1 REBUILD UNUSABLE LOCAL INDEXES;
```
- 或者通过如下命令单独设置二级分区channel1_product1上的索引分区channel1_product1_idx1不可用。

```
ALTER INDEX list_range_sales_idx1 MODIFY PARTITION channel1_product1_idx1 UNUSABLE;
```
- 通过如下命令单独重建二级分区channel1_product1上的索引分区channel1_product1_idx1。

```
ALTER INDEX list_range_sales_idx1 REBUILD PARTITION channel1_product1_idx1;
```

7.8.5 分区并发控制

分区并发控制给出了分区表DQL、DML、DDL并发过程中的行为规格限制。用户在设计分区表并发业务时，尤其是在进行分区维护操作时，可以参考本章节指导。

7.8.5.1 常规锁设计

分区表通过表锁+分区锁两重设计，在表和分区上分别施加不同级别的常规锁，保证DQL、DML以及DDL并发过程中的合理行为控制。不同级别锁的相容行为如表7-8所示，表中标记为“√”的两种常规锁互不阻塞，可以并行。

表 7-8 常规锁相容行为

常规锁相容行为	ACCESS_SHARE	ROW_SHARE	ROW_EXCLUSIVE	SHARE_UPDATE_EXCLUSIVE	SHARE	SHARE_ROW_EXCLUSIVE	EXCLUSIVE	ACCESS_EXCLUSIVE
ACCESS_SHARE	√	√	√	√	√	√	√	×
ROW_SHARE	√	√	√	√	√	√	×	×
ROW_EXCLUSIVE	√	√	√	√	×	×	×	×
SHARE_UPDATE_EXCLUSIVE	√	√	√	×	×	×	×	×
SHARE	√	√	×	×	√	×	×	×
SHARE_ROW_EXCLUSIVE	√	√	×	×	×	×	×	×
EXCLUSIVE	√	×	×	×	×	×	×	×
ACCESS_EXCLUSIVE	×	×	×	×	×	×	×	×

分区表的不同业务最终都是作用于目标分区上，数据库会给分区表和目标分区施加不同级别的表锁+分区锁，来控制并发行为。表7-9给出了不同业务的锁粒度控制。其中数字1~8分别代表表7-8给出的ACCESS_SHARE、ROW_SHARE、ROW_EXCLUSIVE、SHARE_UPDATE_EXCLUSIVE、SHARE、SHARE_ROW_EXCLUSIVE、EXCLUSIVE、ACCESS_EXCLUSIVE这8种级别的常规锁。

表 7-9 分区表业务锁粒度

业务模型	一级分区表锁级别(表锁+分区锁)	二级分区表锁级别(表锁+一级分区锁+二级分区锁)
SELECT	1-1	1-1-1

业务模型	一级分区表锁级别(表锁+分区锁)	二级分区表锁级别(表锁+一级分区锁+二级分区锁)
SELECT FOR UPDATE	2-2	2-2-2
DML业务，包括INSERT、UPDATE、DELETE、UPSERT、MERGE INTO、COPY	3-3	3-3-3
大部分分区DDL，包括ADD、DROP、EXCHANGE、TRUNCATE、MOVE、RENAME	4-8	4-8-8（作用二级分区表的一级分区） 4-4-8（作用二级分区表的二级分区）
CREATE INDEX（非分类索引）、REBUILD INDEX	5-5	5-5-5
CREATE INDEX（分类索引）	3-5	3-3-5
REBUILD INDEX PARTITION	1-5	1-5-5
ANALYZE、VACUUM	4-4	4-4-4
其他分区表DDL，包括SPLIT/MERGE这两种分区DDL	8-8	8-8-8

如果业务执行施加的表锁和分区锁均满足表7-8，则可以支持业务并行操作，如果表锁和分区锁有任一不相容，则二者不支持业务并行。

说明

DDL (ADD/DROP/TRUNCATE/EXCHANGE/MOVE/RENAME) 操作对分区表施加的表锁级别受GUC参数enable_partition_ddl_lowlevel_lock控制，当enable_partition_ddl_lowlevel_lock为on时，对表施加4级锁；当enable_partition_ddl_lowlevel_lock为off时，对表施加8级锁。

7.8.5.2 DQL/DML-DQL/DML 并发

DQL/DML操作会给表和分区施加1~3级别的常规锁，DQL/DML操作自身互不阻塞。

分区表支持DQL/DML-DQL/DML并发。

7.8.5.3 DQL/DML-DDL 并发

表级DDL、以及SPLIT/MERGE这两种分区DDL会给分区表施加8级锁，阻塞该分区表全部的DQL/DML操作。

ADD、DROP、EXCHANGE、TRUNCATE、MOVE、RENAME这六种分区级DDL会给分区表施加4级锁，并给目标分区施加8级锁。当DQL/DML与DDL作用不同分区时，支持二者执行层面的并发；当DQL/DML与DDL作用相同分区时，后触发业务会被阻塞。

须知

- 业务在进行分区DDL维护操作时，应尽可能避免期间同时对目标分区进行DQL/DML操作。
- 如果并发的DDL与DQL/DML作用目标分区有重叠，由于串行阻塞，DQL/DML既可能先于DDL发生，也可能后于DDL发生，用户应该明确知晓其可能的预期结果。比如当Truncate与Insert作用于同一分区时，如果Truncate先于Insert触发，则业务完成后目标分区存在数据，如果Truncate后于Insert触发，则业务完成后目标分区不存在数据。
- 如果分区表为bucket表，且表上存在全局索引，当执行MERGE、EXCHANGE以及SET TABLESPACE三种分区级DDL操作时会重建全局索引，期间阻塞表上DML业务。

DQL/DML-DDL 跨分区并发

GaussDB支持跨分区的DQL/DML-DDL并发。

例如，定义如下分区表range_sales，下面给出了一些支持并发的例子。

```
gaussdb=# CREATE TABLE range_sales
(
    product_id    INT4 NOT NULL,
    customer_id   INT4 NOT NULL,
    time_id       DATE,
    channel_id    CHAR(1),
    type_id       INT4,
    quantity_sold NUMERIC(3),
    amount_sold   NUMERIC(10,2)
)
PARTITION BY RANGE (time_id)
(
    PARTITION time_2008 VALUES LESS THAN ('2009-01-01'),
    PARTITION time_2009 VALUES LESS THAN ('2010-01-01'),
    PARTITION time_2010 VALUES LESS THAN ('2011-01-01'),
    PARTITION time_2011 VALUES LESS THAN ('2012-01-01')
);

gaussdb=# CREATE TABLE temp
(
    product_id    INT4 NOT NULL,
    customer_id   INT4 NOT NULL,
    time_id       DATE,
    channel_id    CHAR(1),
    type_id       INT4,
    quantity_sold NUMERIC(3),
    amount_sold   NUMERIC(10,2)
);
```

分区表支持的并发业务可以为如下典型场景。

```
--并发case1，插入分区time_2011与清空分区time_2008互不阻塞
gaussdb=# \parallel on
gaussdb=# INSERT INTO range_sales VALUES (455124, 92121433, '2011-09-17', 'X', 4513, 7, 17);
gaussdb=# ALTER TABLE range_sales TRUNCATE PARTITION time_2008 UPDATE GLOBAL INDEX;
gaussdb=# \parallel off

--并发case2，指定分区time_2010查询与交换分区time_2009互不阻塞
gaussdb=# \parallel on
gaussdb=# SELECT COUNT(*) FROM range_sales PARTITION (time_2010);
gaussdb=# ALTER TABLE range_sales EXCHANGE PARTITION (time_2009) WITH TABLE temp UPDATE
GLOBAL INDEX;
gaussdb=# \parallel off

--并发case3，对分区表range_sales做更新与删除分区time_2008互不阻塞，这是因为更新SQL带条件剪枝到分区
```



```
time_2010和time_2011上
gaussdb=# \parallel on
gaussdb=# UPDATE range_sales SET channel_id = 'T' WHERE channel_id = 'X' AND time_id > '2010-06-01';
gaussdb=# ALTER TABLE range_sales DROP PARTITION time_2008 UPDATE GLOBAL INDEX;
gaussdb=# \parallel off

--并发case4，对分区表range_sales的任何DQL/DML操作与新增分区time_2012互不阻塞，这是因为新增分区对其他进行的业务不可见
gaussdb=# \parallel on
gaussdb=# DELETE FROM range_sales WHERE channel_id = 'T';
gaussdb=# ALTER TABLE range_sales ADD PARTITION time_2012 VALUES LESS THAN ('2013-01-01');
gaussdb=# \parallel off
```

DQL/DML-DDL 同分区并发

GaussDB不支持同分区的DQL/DML-DDL并发，后触发业务会被先触发业务阻塞。

原则上，不建议用户在进行分区DDL时，同时对该分区进行DQL/DML操作，因为目标分区存在一个状态的突变过程，可能会导致业务的查询结果不符合预期。

当由于业务模型不合理或无法剪枝等场景导致的DQL/DML和DDL作用分区有重叠时，考虑以下两种场景：

场景一：先触发DQL/DML，再触发DDL。DDL会被阻塞，等待DQL/DML提交后再进行。

场景二：先触发DDL，再触发DQL/DML。DQL/DML会被阻塞，等待DDL提交后再进行，由于分区元信息在此期间发生变化，可能导致预期结果异常。为了保证数据一致性，预期结果按照如下规则制定。

- **ADD分区**

ADD分区会产生一个新的分区，这个新分区对期间触发的DQL/DML操作均是不可见的，无阻塞期。

- **DROP分区**

DROP分区会将已有分区进行删除，期间触发的目标分区DQL/DML操作会被阻塞，阻塞完成后跳过对该分区的处理。

- **TRUNCATE分区**

TRUNCATE分区会清空已有分区的数据，期间触发的目标分区DQL/DML操作会被阻塞，阻塞完成后继续对该分区进行处理。

注意期间触发的目标分区查询是查不到数据的，因为TRUNCATE操作提交后目标分区中不存有任何数据。

- **EXCHANGE分区**

EXCHANGE分区会将一个已有分区与普通表进行交换，期间触发的目标分区DQL/DML操作会被阻塞，阻塞完成后继续对该分区进行处理，该分区的实际数据对应原普通表。

例外：如果分区表上存在GLOBAL索引，EXCHANGE命令带有UPDATE GLOBAL INDEX子句，且期间触发的分区表查询使用了GLOBAL索引，由于无法查询到交换后分区上的数据，在阻塞完成后查询业务会报错。

```
ERROR: partition xxxxxx does not exist on relation "xxxxxx"
```

```
DETAIL: this partition may have already been dropped by cocurrent DDL operations EXCHANGE PARTITION
```

- **SPLIT分区**

SPLIT分区当前会阻塞全表的DQL/DML操作，也包括目标分区本身。SPLIT提交后，DQL/DML操作会基于SPLIT完成后的分区表结构进行。

- **MERGE分区**

MERGE分区当前会阻塞全表的DQL/DML操作，也包括目标分区本身。MERGE提交后，DQL/DML操作会基于MERGE完成后的分区表结构进行。

- **RENAME分区**

RENAME分区不改变分区结构信息，期间触发的DQL/DML操作不会出现任何异常，但会被阻塞，直到RENAME操作提交。

- **MOVE分区**

MOVE分区不改变分区结构信息，期间触发的DQL/DML操作不会出现任何异常，但会被阻塞，直到MOVE操作提交。

须知

- 在DQL/DML业务期间，如果对执行DQL/DML操作的分区，连续同时做多次分区DDL操作，有低概率出现报错，报错原因：分区找不到，分区已经被DDL删除。
- DQL/DML-DDL同分区并发有低概率出现业务死锁或者锁超时。原因：部分DQL/DML操作只在DN施加分区锁，若和DDL在不同DN的加锁顺序不一致则有概率导致死锁/锁超时。为避免死锁，可以设置GUC参数enable_partition_ddl_lowlevel_lock为off，但是会有性能影响，具体可参考GUC参数enable_partition_ddl_lowlevel_lock的说明。

7.8.6 分区表系统视图&DFX

7.8.6.1 分区表相关系统视图

非透明多写特性下，分区表系统视图根据权限分为3类，具体字段信息请参考《参考》中“系统表和系统视图 > 系统视图”章节。

- 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。
- 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。
- 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。

7.8.6.2 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

非透明多写特性下：

- 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_filenode(partition_oid)获取到指定分区表的OID所对应的filenode。

查看分区p1的filenode。

```
SELECT pg_partition_filenode(49294);
pg_partition_filenode
-----
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)
```

- 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)
```

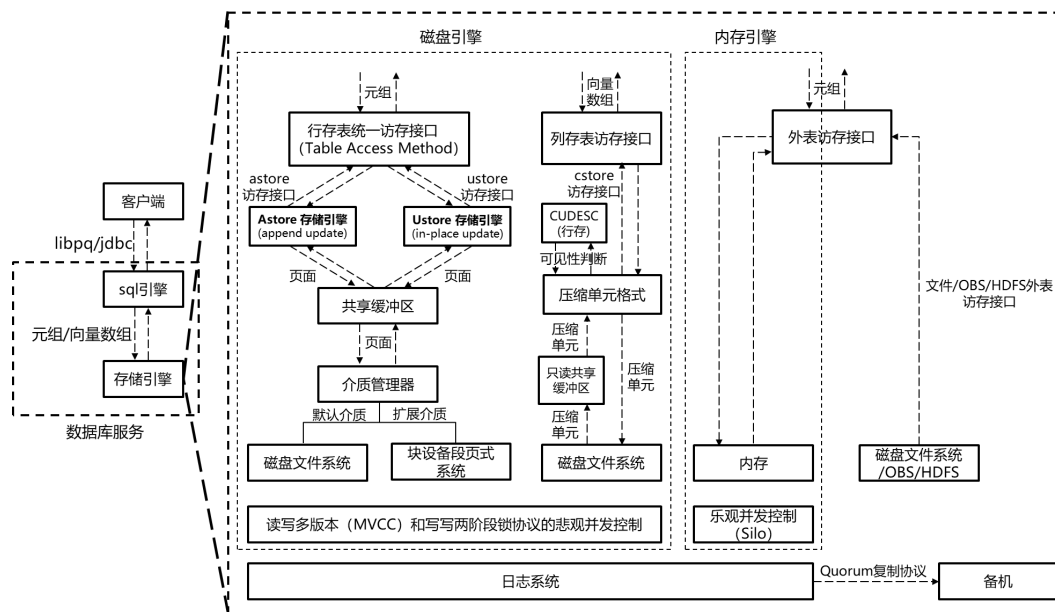
7.9 存储引擎

7.9.1 存储引擎体系架构

7.9.1.1 存储引擎体系架构概述

7.9.1.1.1 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。

7.9.1.1.2 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。

7.9.1.2 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。参数的详细描述如表7-10所示：

表 7-10 存储参数说明

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，参数详情请参见《参考》中“数据库运行参数说明 > GUC参数说明”章节）：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 | 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;
```

7.9.1.3 存储引擎更新说明

7.9.1.3.1 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。

7.9.1.3.2 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。

7.9.1.3.3 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。

7.9.2 Astore 存储引擎

7.9.2.1 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。

7.9.3 Ustore 存储引擎

7.9.3.1 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，I/O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。

7.9.3.1.1 Ustore 特性与规格

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	×
SQL	Table sampling/物化视图/键值锁	×

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 - MAXALIGN(56 + 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 - MAXALIGN(56+2*26+4) = 8080B。
 - 表init_td数量为默认值4时，Tuple长度不能超过8192 - MAXALIGN(56+4*26+4) = 8024B。
 - 表init_td数量为最大值128时，Tuple长度不能超过8192 - MAXALIGN(56+128*26+4) = 4800B。
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 - MAXALIGN(28 + 3 * 4 + 3 * 10) - 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。

7.9.3.1.2 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
```

7.9.3.1.3 Ustore 的最佳实践

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且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《参考》中“SQL参考 > SQL语法 > C > 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 | 0 | PageWriter | 0 | none | none | 0 | 139769678919424 | 139769678919424 | 16915
sgnode | 0 | PageWriter | 0 | none | none | 0 | 139769736066816 | 139769736066816 | 16913
sgnode | 0 | PageWriter | 0 | none | none | 0 | 139769707755264 | 139769707755264 | 16914
sgnode | 0 | PageWriter | 0 | none | none | 0 | 139769761756928 | 139769761756928 | 16912
sgnode | 0 | PageWriter | 0 | none | none | 0 | 139769783772928 | 139769783772928 | 16911
gaussdb=# DROP TABLE test1;
DROP TABLE
```

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《参考》中“SQL参考 > SQL语法 > C > 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
```

须知

通过ALTER TABLE重新设置表的init_td值时，该值仅对新插入的页面生效。对于已存在的页面，其td数量保持不变，需执行VACUUM FULL才能使新设置生效。

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，ustore_attr参数默认开启UPAGE:UBTREE:UNDO三个模块校验。业务现网请保持开启，性能场景除外。

ustore_attr参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 开发人员选项”章节。

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《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > Undo系统函数”章节。

2. 保留给定空间大小的历史版本数据。

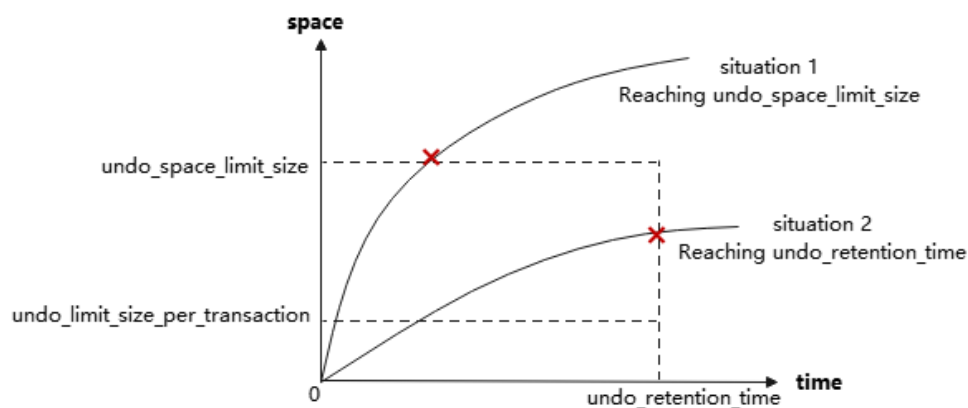
如果业务中存在长事务或大事务可能导致Undo空间膨胀时，需要将undo_space_limit_size调大，undo_space_limit_size默认值为256GB，取值范围为800MB~16TB。

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。



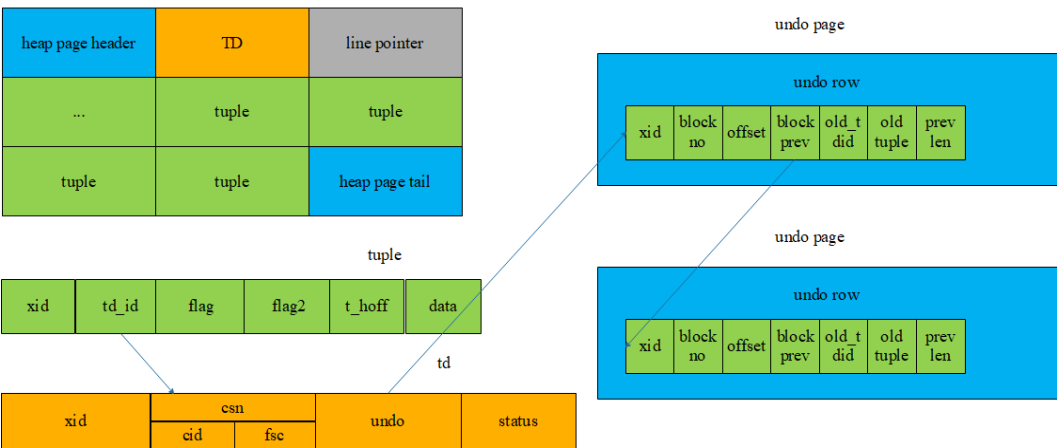
例如：Undo强制回收阈值参数undo_space_limit_size设置为1GB，Undo旧版本保留时间undo_retention_time为900s。如果900s内产生的历史版本数据不足1GB*0.8，则按照900s进行回收限制；否则按照1GB*0.8进行回收限制。遇到该情况时，如果磁盘空闲空间充足，则上调undo_space_limit_size，如果磁盘空闲空间紧缺，则下调undo_retention_time。

7.9.3.2 存储格式

7.9.3.2.1 RCR Uheap

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。



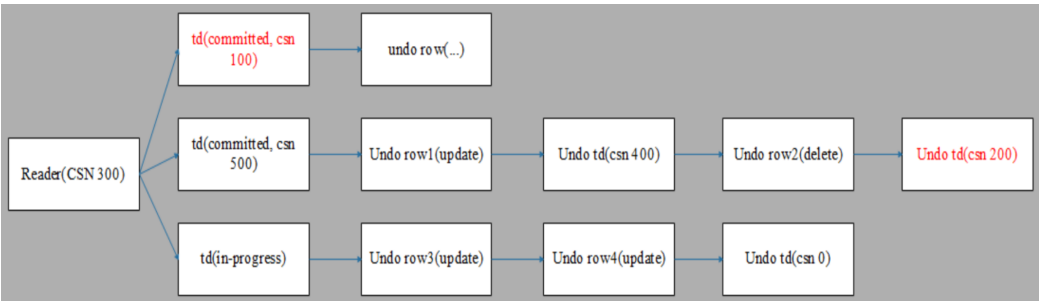
Ustore RCR(Row Consistency Read)的多版本管理是基于数据行的行级多版本管理。不过Ustore将XID记录在了页面的TD(Transaction Directory)区域区别于常见的将XID存储在数据行上，节省了页面空间。事务修改记录时，会将历史数据记录到Undo Row中，在Tuple中的td_id指向的TD槽上记录产生的Undo Row地址(zone_id, block no, page offset)，并将新的数据覆盖写入页面。访问元组时，沿着版本链还原该元组，直到找到自己对应的版本。

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。举例如下：

图中描述了三种使用MVCC快照查询的场景，其中Reader为查询线程，持有csn = 300的快照，需要查询某行数据的可见版本：

- 1. 该行数据指向的td槽位记录事务xid已经提交，并且td csn < 快照csn，说明td xid对快照可见。又因为生成该行数据页面版本的事务xid一定小于td上记录的xid，并且已经提交，因此该数据的页面最新版本也对快照可见，页面最新版本即为快照可见版本。
- 2. 该行数据指向的td槽位记录事务xid已经提交，但是td csn > 快照csn，对快照不可见，因此需要从新到旧遍历Undo链，寻找对快照可见版本（事务xid已提交，并且事务csn < 快照csn），最终找到csn为200的版本。
- 3. 复用td槽位的事务为in-progress状态，td xid对快照不可见，因此同样需要从新到旧遍历Undo链，寻找对快照可见版本。遍历结束后，发现链上找不到对应Undo，说明通过生成该元组页面版本的事务生成的Undo也已经被回收掉，进而说明该元组的所有版本都已经对所有快照可见，页面最新版本即为可见版本。



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間要求；如果满足则返回对应的blocknum用于执行插入操作，否则执行拓展页面逻辑。

每一个表或者分区对应的FSM结构存放在一个独立的FSM文件中，该FSM文件与表数据放在相同的目录下。例如，假设表t1对应的数据文件为32181，则其对应的FSM文件为32181_fsm。FSM内部同样是以数据块的格式存储，这里称为FSM block，FSM block之间的逻辑结构组成了一棵有三层节点的树，树的节点在逻辑上是大顶堆关系。每次在FSM上查找时从根节点进行，一直查找找到叶子节点，然后在叶子节点内搜索到一个可用的页面并返回给业务用于执行后续操作。

该结构不保证和数据页实际可用空间保持实时一致，会在DML的执行过程中进行维护。Ustore会在Auto Vacuum的过程中概率性对该FSM进行修复重建。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。

7.9.3.2.2 UBTREE

其使用的btree做了如下重要的增强，简称Ubtree。

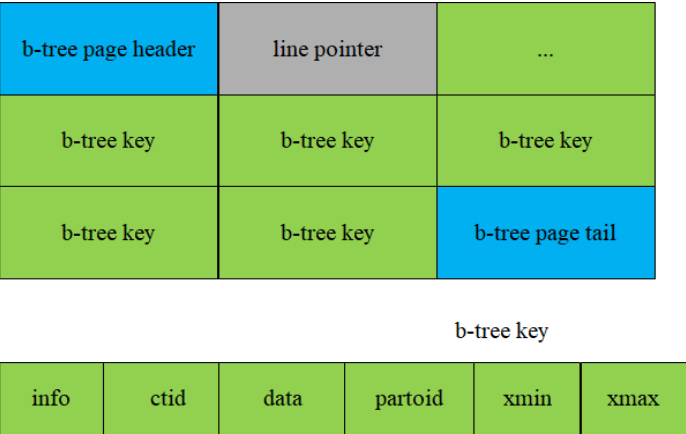
- Ubtree索引增加了事务信息，能够独立进行MVCC；增加了IndexOnlyScan的比例，大大减少回表次数。
- 不依赖Vacuum进行旧版本清理。独立的空间回收能力，索引与堆表解耦，可独立清理，IO平稳度更优。

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7-11所示。

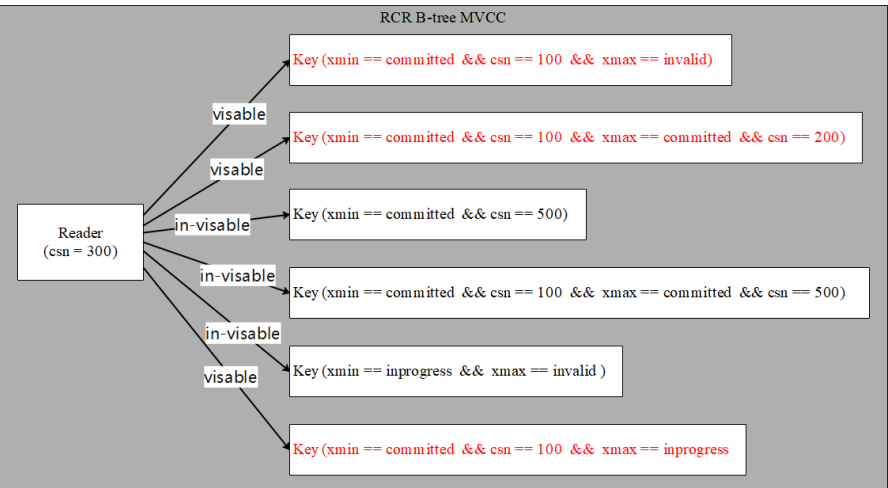
图 7-11 RCR UBTree 多版本管理



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，根据 xmin/xmax 提交状态与 csn 的不同组合，会返回元组不同的可见性判断结果，如图7-12所示。

图 7-12 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7-13所示。

图 7-13 UBTREE 在数据发生更新时的处理

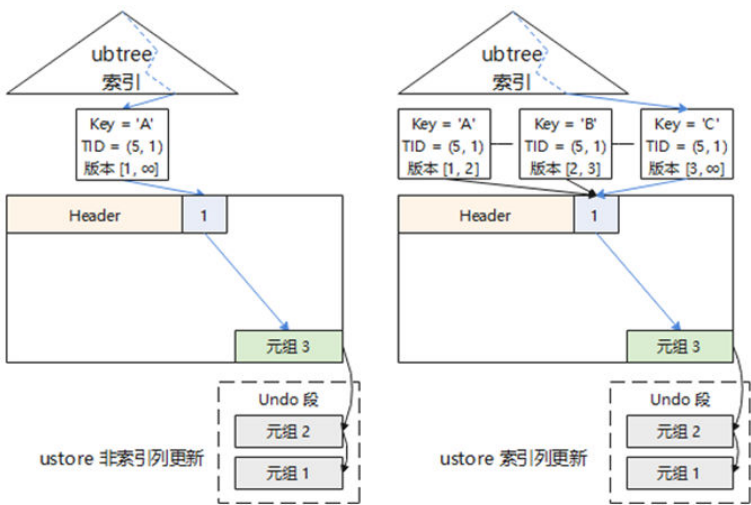


图7-13展示Ubtree在索引列和非索引列更新的差异：

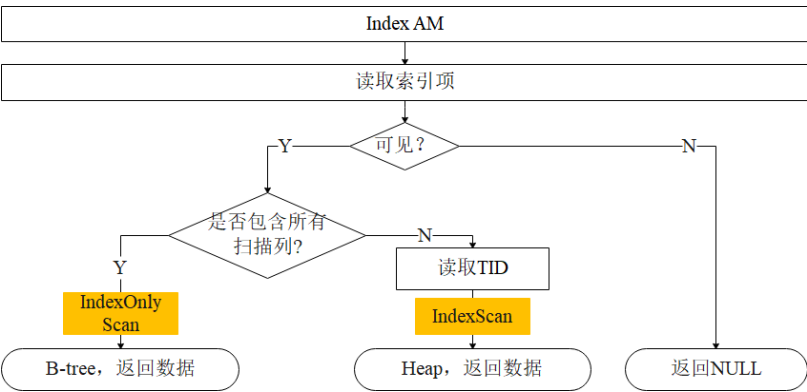
- 在非索引列更新的情况下，索引不发生任何变化。index tuple仍指向第一次插入的data tuple，Uheap不会插入新的data tuple，而是修改当下data tuple并将历史数据存入Undo中。
- 在索引列更新的情况下，Ubtree也会插入新的index tuple，但是会指向同一个data linepointer和同一个data tuple。扫描旧版本的数据则需要从Undo中读取。

- **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7-14所示。

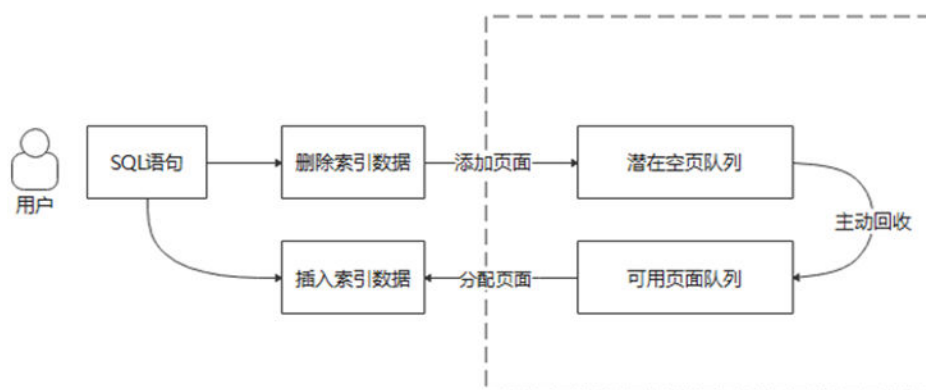
图 7-14 对应的数据元组



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 7-15 索引页面在双循环队列间流动



如图7-15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。

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 7-11 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。

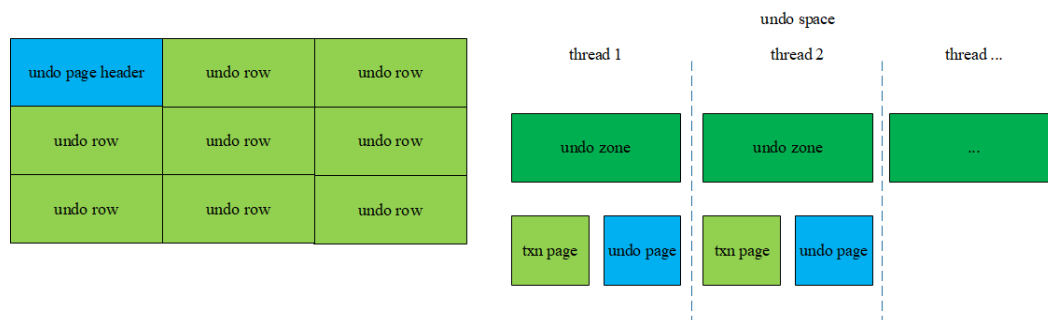
7.9.3.2.3 Undo

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。

回滚段管理

回滚段管理如图7-16所示：

图 7-16 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。

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 '%undostoragetype%';
 name | value
-----+-----
undostoragetype | 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

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。

7.9.3.2.4 Enhanced Toast

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。

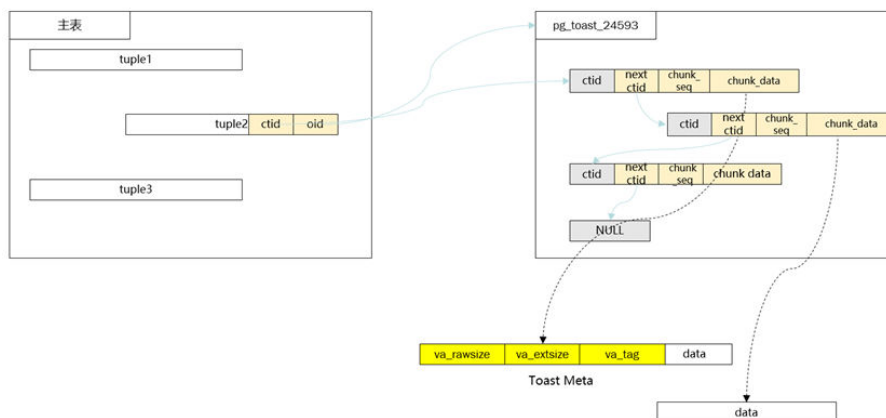
说明

- 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。

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，如图7-17所示。这样极大简化了主表与线外存储表间的映射关系，有效地提升了数据写入与查询的性能。

图 7-17 Enhanced Toast 存储结构



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参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > Enhanced Toast”章节。

- “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。

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。

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间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
```

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)
```

7.9.3.3 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/DDI语句运行时才会真正为该事务分配XID。
2. 事务结束时，会产生代表事务提交状态的CLOG（Commit Log），CLOG共有四种状态：事务运行中、事务提交、事务同步回滚、子事务提交。每个事务的CLOG状态位为2 bits，CLOG页面上每个字节可以表示四个事务的提交状态。
3. 事务结束时，还会产生代表事务提交顺序的CSN（Commit sequence number）。CSN为实例级变量，每个XID都有自己对应的唯一CSN。CSN可以标记事务的以下状态：事务提交中、事务提交、事务回滚、事务已冻结等。

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

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

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

7.9.3.3.2 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会进行父事务的回滚。

7.9.3.4 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。

7.9.3.4.1 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 [, ...] ) ] ]  
[function_name ( [ argument [, ...] ) ] AS ( column_definition [, ...] )  
[from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ] ] }
```

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

参数说明

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

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

使用示例

- 示例（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
```

7.9.3.4.2 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)
--闪回表至此刻之前的特定时间戳
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)
--闪回表至此刻之后的特定时间戳
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP '2023-09-13 20:54:00.641506';
ERROR: The specified timestamp is invalid.
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
-----+-----
    3 | INSERT3
    6 | INSERT6
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
(6 rows)
gaussdb=# DROP TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

7.9.3.4.3 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

[illegible]

```
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;
 rcybaseid | rcydbid | rcyrelid | rcynome | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychange | rcynames | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenid | rcyfrozenid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
(0 rows)

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

--DROP表
gaussdb=# DROP TABLE IF EXISTS flashtest;
DROP TABLE
--查看回收站
gaussdb=# SELECT * FROM gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcynome | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
 rcyrecycletime | rcycreatecsn | rcychange | rcynames | rcyowner | rcytablespace
 | rcyrelfilenode | rcyanrestore | rcyanpurge | rcyfrozenid | rcyfrozenid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
18648 | 12737 | 18641 | BIN$31C14EB48D1$9A85$0==$0 | flashtest | d | 0 |
79354509 | 2023-09-13 20:40:11.360638+08 | 79354506 | 7935450
8 | 2200 | 10 | 0 | 18641 | t | t | 226642 | 226642 |
18648 | 12737 | 18644 | BIN$31C14EB48D4$12E236A0==$0 | pg_toast_18641 | d | 2
| 79354509 | 2023-09-13 20:40:11.36112+08 | 0 |
0 | 99 | 10 | 0 | 18644 | f | f | 226642 | 226642 |
```

```

18648 | 12737 | 18647 | BIN$31C14EB48D7$9A85$0==$0 | flashtest_index | d | 1 |
79354509 | 2023-09-13 20:40:11.361246+08 | 79354508 | 7935450
8 | 2200 | 10 | 0 | 18647 | f | t | 0 | 0 |
(3 rows)

--PURGE索引flashtest_index
gaussdb=# PURGE INDEX flashtest_index;
PURGE INDEX
--查看回收站，回收站中的索引flashtest_index被删除
gaussdb=# SELECT * FROM gs_recyclebin;
rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype |
rcyrecyclecsn | rcyrecycletime | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
--+-+-----+-----+-----+-----+-----+-----+-----+
+-----+
18648 | 12737 | 18641 | BIN$31C14EB48D1$9A85$0==$0 | flashtest | d | 0 |
79354509 | 2023-09-13 20:40:11.360638+08 | 79354506 | 7935450
8 | 2200 | 10 | 0 | 18641 | t | t | 226642 | 226642 |
18648 | 12737 | 18644 | BIN$31C14EB48D4$12E236A0==$0 | pg_toast_18641 | d | 2
| 79354509 | 2023-09-13 20:40:11.36112+08 | 0 |
0 | 99 | 10 | 0 | 18644 | f | f | 226642 | 226642 |
(2 rows)

-- PURGE RECYCLEBIN --
--PURGE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 | rcychangecs | rcynamespace | rcyowner | rcytablespace
| rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+
(0 rows)

-- TIMECAPSULE TABLE { table_name } TO BEFORE DROP [RENAME TO new_tablename] --
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)

--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 | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+

```

```
--+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
18658 | 12737 | 18652 | BIN$31C14EB48DC$9B2B$0==$0 | flashtest | d | 0 |
79354760 | 2023-09-13 20:47:57.075907+08 | 79354753 | 7935475 |
3 | 2200 | 10 | 0 | 18652 | t | t | 226824 | 226824 |
18658 | 12737 | 18655 | BIN$31C14EB48DF$12E46400==$0 | pg_toast_18652 | d | 2 |
| 79354760 | 2023-09-13 20:47:57.07621+08 | 0 |
0 | 99 | 10 | 0 | 18655 | f | f | 226824 | 226824 |
(2 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
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 | rcycreatesn | rcychangesn | rcynamespace | rcyowner | rcytablesapce
| 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 | rcycreatesn | rcy
changesn | rcynamespace | rcyowner | rcytablesapce | 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 | rcychangeccsn | 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 | rcycanrestore | rcycanpurge |
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 | 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 | rcy
changeccsn | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge |
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 | rcyrecreatecsn | rcychangeecs | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid64 | rcybucke
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(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 | rcyrecreatecsn | rcychangeecs | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid64 | rcybucke
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(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 | rcychangecons
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozensid |
rcyfrozensid64 | 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 | 0 | 227928 |
18703 | 12737 | 18697 | BIN$31C14EB4909$9E4D$0==$0 | flashtest | t | 0 |
79356610 | 2023-09-13 21:24:42.872792+08 | 79356606 | 7935660
6 | 2200 | 10 | 0 | 18704 | t | t | 0 | 227928 |
(2 rows)

--drop表
gaussdb=# DROP TABLE IF EXISTS flashtest;
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 | rcychangecons
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozensid |
rcyfrozensid64 | rcybucket
-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

7.9.3.5 常用视图工具

视图类型	类型	功能描述	使用场景	函数名称
解析	全类型	用于解析指定表页面，并返回存放解析内容的路径。	- 查看页面信息。 - 查看元组（非用户数据）信息。 - 页面或者元组损坏。 - 元组可见性问题。 - 校验报错问题。	gs_parse_page_by_path

视图类型	类型	功能描述	使用场景	函数名称
	索引回收队列（URQ）	用于解析UB-tree索引回收队列关键信息。	- UB-tree索引空间膨胀。 - UB-tree索引空间回收异常。 - 校验报错问题。	gs_urq_dump_stat
	回滚段（Undo）	用于解析指定Undo Record的内容，不包含旧版本元组的数据。	- undo空间膨胀。 - undo回收异常。 - 回滚异常。 - 日常巡检。 - 校验报错。 - 可见性判断异常。 - 修改参数。	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。	- WAL日志出错。 - 日志回放出错。 - 页面损坏。	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

视图类型	类型	功能描述	使用场景	函数名称
		用于解析指定表页面和表页面对应的日志，并返回存放解析内容的路径。可以看作一次执行gs_parse_page_bypath和gs_xlogdump_tablepath。该函数执行的前置条件是表文件存在。如果想查看已删除的表的相关日志，请直接调用gs_xlogdump_tablepath。		gs_xlogdump_pars epage_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。	- Undo空间膨胀。 - Undo资源监控。	gs_stat_undo
	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。	WAL写/刷盘监控。	gs_stat_wal_entryt able
		用于统计预写日志（WAL）刷盘状态、位置统计信息。	WAL写/刷盘hang住。	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。	- 页面损坏或者元组损坏。 - 可见性问题。 - 日志回放出错问题。	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。	- UB-tree索引空间膨胀。 - UB-tree索引空间回收异常。	gs_verify_urq
	回滚段（Undo）	用于离线校验Undo Record数据是否存在异常。	- Undo Record异常或者损坏。 - 可见性问题。 - 回滚出错或者异常。	gs_verify_undo_rec ord

视图类型	类型	功能描述	使用场景	函数名称
		用于离线校验Transaction Slot数据是否存在异常。	- Undo Record异常或者损坏。 - 可见性问题。 - 回滚出错或者异常。	gs_verify_undo_slot
		用于离线校验Undo元信息数据是否存在异常。	- 因Undo meta引起的节点无法启动问题。 - Undo空间回收异常。 - Snapshot too old问题。	gs_verify_undo_meta
修复	堆表/索引/Undo文件	用于基于备机修复主机丢失的物理文件。	堆表/索引/Undo文件丢失。	gs_repair_file
	堆表/索引/Undo页面	用于校验并基于备机修复主机受损页面。	堆表/索引/Undo页面损坏。	gs_verify_and_tryrepair_page
		用于基于备机页面直接修复主机页面。		gs_repair_page
		用于基于偏移量对页面的备份进行字节修改。		gs_edit_page_bypath
		用于将修改后的页面覆盖写入到目标页面。		gs_repair_page_bypath
	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_byzone
	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

7.9.3.6 常见问题及定位手段

7.9.3.6.1 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。

长事务阻塞 Undo 空间回收

问题现象

1. 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 |    280   |    248   |    17028   |             17028   |    17025   |             17028
(1 row)
```

3. pg_running_xacts与pg_stat_activity视图查询存在长事务，阻塞oldestxmin和global_recycle_xid推进。如果pg_running_xacts中查询活跃事务的xmin和gs_txid_oldestxmin相等，且通过pid查询pg_stat_activity查询线程执行语句时间过长，则表明有长事务卡住被回收。

SELECT * FROM pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by

xmin::text::bigint asc limit 5;

SELECT * FROM gs_txid_oldestxmin();

SELECT * FROM pg_stat_activity WHERE pid = 长事务所在线程PID;

```
1000f
1000f select * from pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by xmin::text::bigint asc limit 5;
 handle | gxid | state | node | xmin | vacuum | timeline | prepare_xid | pid | next_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
-1 | 0 | 0 | dn_6001_6002_6003 | 55757784113 | f | 52 | 0 | 281303148456336 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277821627392 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281275962257552 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281278217428112 | 0
-1 | 0 | 0 | dn_6001_6002_6003 | 55767847391 | f | 52 | 0 | 281277812763024 | 0
(5 rows)

Time: 1089.559 ms
1000f select * from gs_txid_oldestxmin();
 gs_txid_oldestxmin
-----
55757784113
(1 row)

Time: 2.925 ms
1000f select * from txid_current();
 txid_current
-----
55789826467
(1 row)

Time: 7.058 ms
1000f select * from pg_stat_activity where pid = 281303148456336;
 datid | datname | pid | sessionid | usesysid | username | application_name | client_addr | client_hostname | client_port | backend_start |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
0 | 2023-03-13 18:00:40.254716408 | 2023-03-13 18:00:40.254716408 | 2023-03-13 18:00:40.2547277+08 | f | | | | | | | | |
| select /*no tablescan(part)*/ sum(c).count(*) / sum(c).count(*) from part union select sum(c).count(*) from part; | | driver_name: 'libpq', driver_version: 'GaussDB Kernel 503.1.0 build 5030500' compiled at 2023-03-13 08:56:43 commit 5008 last wr 10553 release' | 37454028 |
(1 row)

Time: 13592.263 ms
1000f
```

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）。

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，即调大max_undo_workers参数值。max_undo_workers参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 回滚相关参数”章节。

7.9.3.6.2 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。

7.9.3.6.3 备机读业务报错:"Index scan conflicts with replay. Try again later. The original maxoff is <number>, but now it is <number>."

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“Index scan conflicts with replay. Try again later.”关键字。

问题分析

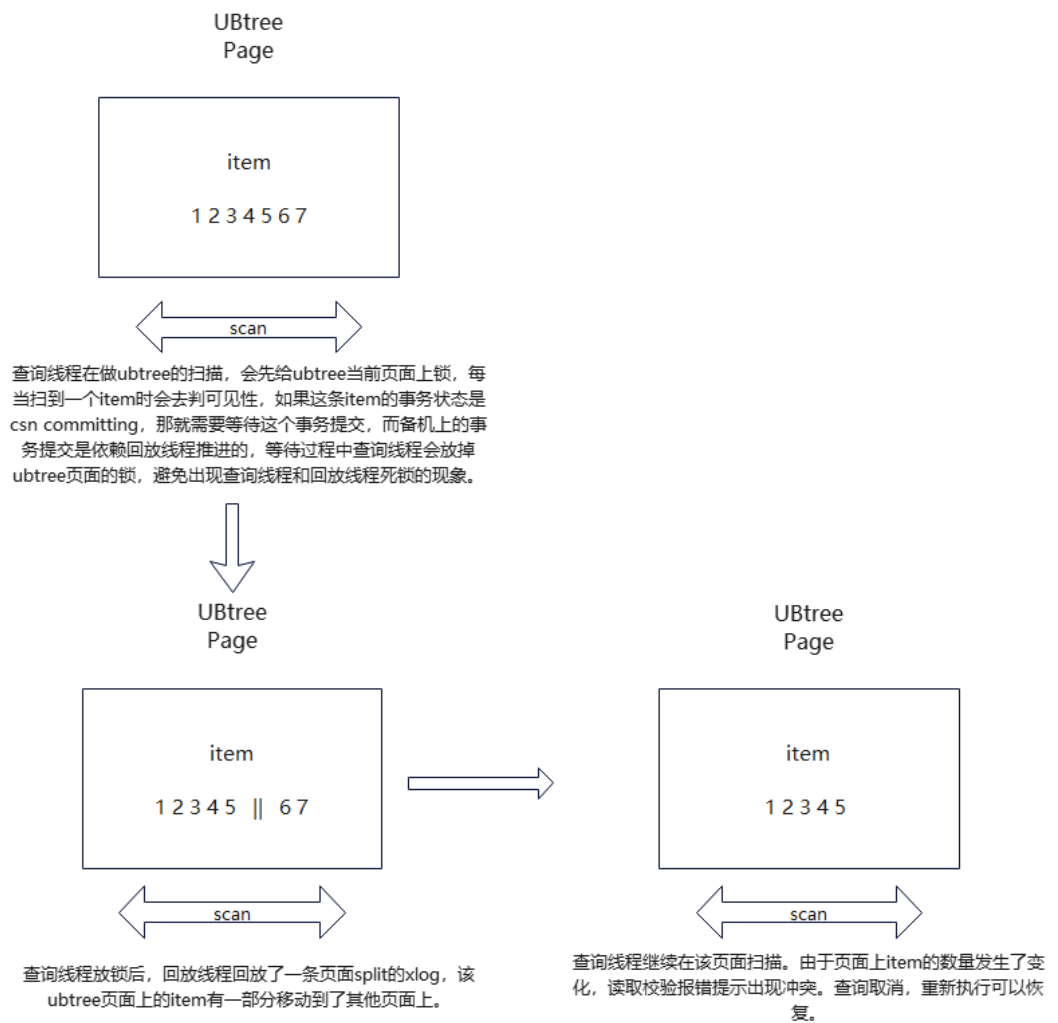
在开启并行回放或串行回放的情况下（查询GUC参数recovery_parse_workers和recovery_max_workers均是1为串行回放；recovery_parse_workers是1，recovery_max_workers大于1为并行回放），备机的查询线程在做索引扫描时，会先对索引页面加读锁，每当扫到一个元组时会去判可见性。如果该元组对应的事务处于committing状态，需要等待该事务提交后再判断。而备机上的事务提交是依赖日志回放线程推进的，这个过程中会对索引页面进行修改，因此需要加锁。查询线程在等待过程中会释放索引页面的锁，否则会出现查询线程等待回放线程进行事务提交，而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 7-18 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。

7.9.3.6.4 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。

7.9.4 数据生命周期管理-OLTP 表压缩

7.9.4.1 特性简介

OLTP表压缩是GaussDB高级压缩中的一个特性。

该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，当前版本支持堆表压缩、跨页透明压缩、索引压缩和copy批量导入压缩。此外，对于压缩策略的配置，当前支持可选压缩级别与库级压缩策略，使得不同业务场景下的压缩需求得以灵活调整。

当前版本不推荐使用高级压缩（ADVANCED），若要使用请联系华为技术工程师。

7.9.4.2 特性约束

对于高级压缩（ADVANCED）：

- 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。
- 支持用户为普通表、分区、二级分区设置ILM策略。
- 支持普通表、分区、二级分区的Astore/Ustore用户表和透明加密表。
- 特性仅在ORA兼容模式、PG兼容模式以及MYSQL兼容模式下有效。
- Ustore不支持编解码，压缩率低于Astore。
- 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。
- 对于ORA兼容模式下的datea类型，若开启mapping_date_to_datea参数，由于数据类型的存储结构发生了变化，压缩率也会受到影响。与date类型相比，datea类型的压缩率可能会更低。
- 对于GET_COMPRESSION_RATIO接口，不支持行级表达式。
- 不支持对压缩表的rowid执行原位更新操作。
- 对于处于压缩状态的用户表，转换为rowid表后，该表数据将被自动解压。

- 对于rowid表，不支持使用rowid/rowno系统列作为行级表达式条件。

对于索引压缩（ADVANCED）：

- 只支持BTree类型索引压缩，场景只支持涉及build/rebuild index相关场景时的压缩和后台调度（定时和手动）时的压缩。
- build index时压缩只支持全部压缩，后台调度则根据页面LMT（Last Modified Time）执行压缩。
- 已压缩的索引需要rebuild index时，只支持全量重新压缩或不压缩，或重建完在后台调度时根据页面LMT压缩。
- 当前版本仅适配Astore，Ustore不支持。
- 索引压缩依赖于索引所在的表或者分区上的策略信息。
- 支持对Astore的普通表、分区表以及hashbucket表上的索引、二级（局部）索引以及全局索引进行索引压缩，不支持全局二级索引压缩。
- 特性仅在ORA兼容模式、PG兼容模式以及MYSQL兼容模式下有效。
- 在某些场景下，可能会导致被压缩索引占用空间膨胀，比如为分区表创建global索引并压缩，合并分区的时候同时更新global索引，由于改变了分区，global索引会有新的key插入，可能会导致global索引发生分裂，最后表现为global索引在分区合并后的占用空间大于合并前的占用空间。
- 对于在线压缩的索引，释放的页面并不能马上复用（需要满足一定条件后在下一轮调度压缩时释放），如果索引key是随机插入的，key可能会插入到已压缩的页面从而导致页面分裂（相比于未压缩页面更有可能分裂，已压缩页面包含的index tuple数往往更多），需要新申请页面，可能会导致已压缩索引的空间膨胀大于未压缩索引的空间膨胀。鉴于此，如果索引key是随机插入的，或者说索引的写入比较难区分冷热，不建议开启索引在线压缩。如果该索引已经开启索引压缩，用户可以选择删除该表的策略以关闭索引压缩（会同时把堆表压缩关闭），或者选择将gs_ilm_param的ENABLE_INDEX_COMPRESS改为0以关闭索引压缩总开关（会导致其他索引也不能压缩）。
- 当表存在多个索引时，索引要么都开启索引压缩，要么都不开启索引压缩，不支持单个索引的压缩控制。
- 不支持与索引跳跃扫描同时使用。如果开启索引跳跃扫描，当扫描到索引压缩行时会直接报错。
- 当数据库从无索引压缩版本升级至有索引压缩版本后，若想要在原表使用索引压缩功能，需要手动重新设置表策略，索引压缩才会生效。
- 如果对索引配置了posting去重压缩，build index时不支持ILM索引压缩。
- 当数据库从无索引压缩版本升级后，若想要在原表使用索引压缩功能，需要手动重新设置表策略，索引压缩才会生效。

对于copy批量导入压缩（ADVANCED）：

- 支持copy批量导入时压缩（包括copy和\copy），不支持insert into select场景下压缩，不支持单行导入时压缩（比如：导入分区表并且开启了违法约束不回滚功能；表上带触发器等）。
- 与copy容错机制（LOG ERRORS或LOG ERRORS DATA）共用时，如果错误数据频繁出现，可能导致压缩无收益。
- TOAST数据不支持导入压缩。
- 由于索引压缩的时机是build/rebuild或后台调度压缩，在copy批量导入压缩时，索引不压缩。

- 升级观察期不支持使用该功能。
- 只支持Astore表，不支持Ustore。
- 支持ORA兼容模式、PG兼容模式、MYSQL兼容模式，不支持M兼容模式。
- 不支持与逻辑解码同时使用，如果开启逻辑解码，则copy导入压缩会直接报错。

对于库级策略（ADVANCED）：

- 库级压缩策略仅支持高级压缩。
- 升级观察期不支持使用库级策略。
- 库级策略不支持表达式功能。
- 在使用gs_dump导出库级策略定义时，需满足以下条件方可生效：必须执行全库导出操作且用户具备系统管理员权限。若上述条件未同时满足，则相关策略将以表、分区或子分区级别的独立策略形式导出，而非作为库级策略处理。

对于跨页透明压缩（TURBO）：

- 仅支持页式下用户创建的Astore/Ustore普通行存表，其他类型表创建时报错。
- 跨页透明压缩不在toast表上生效，对于toast表无压缩效果提升。
- 特性仅在ORA兼容模式、PG兼容模式以及MYSQL兼容模式下有效。
- 跨页透明压缩不支持与其他表压缩算法叠加使用，叠加使用时报错；
- 不支持与透明加密叠加使用，叠加使用时报错。
- 模式编码仅支持ahead、uheap页面类型，其他页面类型不生效，无法提升压缩比。
- 不支持ILM策略ADD、DELETE、ADD+DELETE，否则会报错。
- 分区分裂合并不支持ILM策略类型变更，否则会报错。
- 不支持在reindex时指定ILM压缩策略，否则会报错。
- 分区交换时表和分区的跨页透明压缩配置（compress unit、compressor、encoder）需严格一致，否则报错处理。
- 不支持Online DDL操作，如果指定会自动转化为离线方式执行。
- 跨页透明压缩/解压代价极大，仅支持在冷数据场景中使用。
- 跨页透明压缩的语法和调度方式沿用ILM现有机制，不支持新增兼容性特性。

对于索引压缩（TURBO）：

- 仅支持页式下用户创建的Btree/Ubtrees索引下的数据文件，其他类型索引创建时报错。
- 索引的压缩类型与父表需保持一致，否则会报错。
- 索引不支持分区单独指定跨页透明压缩策略，否则会报错。
- 显式指定索引压缩策略时，以指定策略为准（未指定压缩级别时默认取MEDIUM），未显式指定索引压缩策略时，索引是否压缩和压缩级别与表保持一致。

对于库级策略（TURBO）：

- 升级观察期禁止创建跨页透明压缩表，创建时报错。
- ADIO对跨页透明压缩对象不生效，当开启ADIO时，跨页压缩仍会使用buffered io进行磁盘读写。
- 备份时指定的block buffer选项大小对跨页透明压缩对象的MAINFORK不生效。

对于压缩级别：

- 高级压缩支持可选级别包括medium和high。不指定压缩级别时默认为medium级别，使用lz4压缩算法；指定high级别时，使用zstd压缩算法。
- 跨页透明压缩支持可选级别包括low、medium和high。不指定压缩级别时默认为medium级别。
- 升级观察期不支持指定压缩级别。

7.9.4.3 特性规格

对于高级压缩（ADVANCED）：

- 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TPCC的Orderline表，以及TPCH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。
- 如果指定high级别压缩策略：
 - 压缩率：以tpch.lineitem表为例，high级别比medium级别压缩率Astore和Ustore提升10%。
 - 性能：ZSTD算法适用于对性能不敏感的历史业务场景，访问压缩行的性能会有较大劣化。以TPCC场景为例：默认压缩参数设置（关闭元数据压缩等），4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line都压缩，再执行TPCC业务，1000仓500并发下，Astore性能劣化可能超过30%，Ustore性能劣化可能超过40%。
- 对于TPCH.lineitem表（scale = 1）及TPCC.bmsql_order_line、bmsql_history、bmsql_oorder、bmsql_customer表（32仓），默认参数场景且表未压缩情况下（采样率20%，采样数据量>12MB），GET_COMPRESSION_RATIO接口（数据全压）预测压缩率平均误差不超过10%。
- sysbench场景，默认压缩参数下，开启元数据压缩，与不压缩相比，执行查询（read only）- 过滤条件带rowid性能下降不超过70%。

对于索引压缩（ADVANCED）：

- high级别压缩策略下，单线程对TPCC标准模型的4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line索引build压缩，build耗时平均劣化不超过50%。
- medium级别压缩策略下，单线程对TPCC标准模型的4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line索引build压缩，build耗时平均劣化不超过30%。
- high级别压缩策略下，对TPCC标准模型的4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line索引build压缩，平均压缩率为5比1。
- medium级别压缩策略下，对TPCC标准模型的4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line索引build压缩，平均压缩率2.5比1。
- medium级别压缩策略下，CPU：88U，内存：384GB，磁盘：64TB，TPCC场景，4张流水表bmsql_history、bmsql_new_order、bmsql_oorder、bmsql_order_line堆表和索引都压缩后（关闭元数据压缩），执行TPCC业务，1000仓500并发下，与只压缩堆表相比，性能下降不超过10%。
- medium级别策略下，CPU：88U，内存：384GB，磁盘：64TB，索引和堆表都压缩后（关闭元数据压缩），与只压缩堆表相比，sysbench执行查询（read only）和写入（insert）性能下降不超过10%。

对于copy批量导入时压缩：

- 典型场景压缩率：具体压缩率和数据集以及数据分布相关。以tpch.lineitem表为例，copy批量导入的压缩率和调度时压缩保持一致2:1。
- 性能规格（以16U，64GB为例，默认压缩参数下，即不开启元数据压缩、索引压缩等参数）：
 - a. 导入压缩时性能：
 - 以TPCC和TPCH场景为例，默认压缩相关参数下（不开启元数据压缩等），copy单线程导入，与直接copy导入数据相比，copy导入时压缩的性能（耗时）劣化低于20%（导入后再创建索引，tpch.lineitem表，以及TPCC的bmsql_history，bmsql_oorder和bmsql_order_line表）。
 - 对于压缩率较低的表，由于压缩无I/O收益，并且增加了CPU消耗，这种表copy导入的性能劣化可能较为严重，例如tpcc.bmsql_customer表的性能劣化可能在30%以上，因此对于压缩率较低的表不建议执行copy导入压缩。
 - b. 导入压缩后性能：和调度压缩保持一致，TPCC的4张流水表bmsql_history，bmsql_new_order，bmsql_oorder和bmsql_order_line导入时压缩，不开启元数据压缩和索引压缩，全压缩后再执行TPCC业务，1000仓500并发下，性能下降不超过10%。
- 压缩策略以copy指定策略为准，覆盖表/分区策略：
 - 对于表：以copy指定策略为准，忽略表上ILM压缩策略，即便表上无ILM压缩策略，也执行压缩并导入。
 - 对于分区：以copy指定策略为准，即便某个partition上无ILM压缩策略，对于导入到这个分区上的数据也会执行压缩。
- copy批量导入压缩后的大小，和页面填充因子fillfactor保持一致。
- copy导入压缩会调用编码、压缩算法，会导致CPU算力消耗增加；并且由于数据流分为压缩和非压缩两部分，每个表/分区可能会增加10MB左右的内存使用。

对于跨页透明压缩（TURBO）：

- 功能及资源规格（medium压缩级别：默认compress unit = 16，encoder = none）：
 - 金融典型数据集整库压缩比约4:1，等权压缩比约5:1；物流典型数据集整库压缩比约6:1，等权压缩比约7:1。
 - 压缩解压需要使用内存资源，在开启线程池下，额外需要：最大并发会话数×1MB的内存；
 - 压缩元数据管理需要增加内存资源，额外需要：share buffer大小的1/128的内存。
 - 双写区会增加实例级独立文件，文件大小上限2GB。
 - 备机需要放开JOB任务约束。
- 以下场景可能导致压缩表磁盘空间占用的变化：
 - 主从节点由于使用了不同的GUC参数turbo_compress_pattern_encoder（不同的模式编码器），可能造成主从节点的压缩表有不同的大小。
 - 出于性能考虑，pattern encoder仅在ILM任务压缩的场景下生效，其他压缩场景不生效，因此在开启了pattern encoder后，可能会随着同步压缩场景的触发产生一定存储空间的膨胀，可通过ILM压缩任务消除膨胀。
 - 压缩表执行逻辑扩容，由于pattern encoder仅在ILM任务压缩场景下应用，因此逻辑扩容后表数据会有小幅膨胀（膨胀率不超过pattern encoder的压缩率增益，通常为30%左右）。当前只能在扩容结束后，通过定时ILM压缩任务消除膨胀，且gsilmpolicy_seq值会出现跳变（压缩表扩容产生的中间对象会消耗sequence值）。
- 跨页压缩性能评估方法：

大容量典型用户场景：
1、CPU：2P，内存：1TB，磁盘类型 EVS（超高IO型4块组条带化）或 SAS/SATA SSD（4块组RAID0），搭配NVMe缓存进行加速
2、SATA SSD读带宽500MB/s，SAS SSD读带宽1GB/s
3、用户数据压缩比5:1
4、medium压缩级别：默认compress unit = 16，encoder = none

 - a. 跨页压缩/解压会增加IO资源消耗产生读写放大问题，单次读写IO资源消耗增大3~4倍。
 - b. 跨页压缩/解压会增加CPU算力消耗，以zstd-level1官方benchmark规格为例：压缩速率500MB/s，解压速率1660MB/s，那么在跨页压缩默认配置下，那么单页面解压读耗时增大的计算方法为：
方法1：(compress unit数据大小/解压速率 + compress unit压缩后数据大小/磁盘读速率) / (单页面大小/磁盘读速率)
下面进行举例说明：
举例1：(1)SAS SSD读带宽1GB/s，用户数据压缩比5:1，可得单页面加载的总耗时会增加约13倍。磁盘性能较好，IO耗时接近CPU耗时，会导致CPU解压代价的整体占比变高，此时CPU算力会成为瓶颈
(2)SATA SSD读带宽为500MB/s，用户数据压缩比为5:1，单页面加载耗时增加则约8倍。综上，磁盘性能越好，单页面加载的劣化越明显。
备注：实际用户场景中，业务访问大多具有一定的局部性，不太会出现所有场景都需要冷加载这么极端的情况出现，因此，整体性能结果可参考TPCC冷热模型的实际测试结果。
 - c. 跨页压缩会导致涉及到MAINFORK压缩文件的访问均会由于解压及IO放大问题引入性能劣化，性能劣化评估方法为：
方法2：压缩性能影响评估方法：读压缩页面产生的IO读放大及解压造成的额外耗时倍数N × 压缩页面的业务量占比X%
 - d. 跨页压缩会导致主机recovery时从双写区恢复的时间增加，在开启backend刷脏的情况下，增加时间的度量方法为：
方法3：(需要从双写区读取数据的总IO量 + 需要写回到磁盘的总IO量) / IO吞吐量，总IO量的计算方法为512K × 同一时刻写双写区的会话数量

备注：在冷温热模型下，实际上并发会话刚好同时走到写双写区的概率是非常低的，预计对主机故障恢复的时间影响为毫秒级。

- e. 跨页压缩会导致主/备及日志回放阶段性能劣化、导致备机读性能劣化、导致增量build性能劣化、导致增量备份恢复性能劣化，增加时间的度量参考c。

7.9.4.4 使用说明

使用高级压缩的功能，用户必须申请License才能使用。具体情况请联系华为工程师。

步骤1 执行如下命令开启压缩功能：

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

说明

执行“ALTER DATABASE SET ilm = on”语句需要拥有public权限。

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					Storage
Schema	Name	Type	Owner		
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_ilmpolicies;

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_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 |      |      | 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_ilmobjects;
 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)
```

- 新建带high级别压缩策略的表：

```
gaussdb=# CREATE TABLE ilm_table_h1 (col1 int, col2 text)
 ilm add policy row store compress advanced high row
 after 3 days of no modification on (col1 < 1000);
```

- 为存量表添加high级别压缩策略：

```
gaussdb=# CREATE TABLE ilm_table_h2 (col1 int, col2 text);
gaussdb=# ALTER TABLE ilm_table_h2 ilm add policy row store
compress advanced high row after 3 days of no modification;
```

步骤3 执行压缩评估。

- 手动执行压缩评估。

可通过修改POLICY_TIME参数值，决定评估时间单位为天或者秒，仅系统管理员可以修改，具体修改方法请参见《参考》中“存储过程 > 高级包 > 二次封装接口（推荐）> DBE_ILM_ADMIN”章节中“DBE_ILM_ADMIN.CUSTOMIZE_ILM”接口。

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
$ _postgres1 |
(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 template1';
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_MAINTENANCE_WINDOW_START,  
  REPEAT_INTERVAL => V_MAINTENANCE_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
INDEXJOB_SIZELIMIT	1024
ENABLE_INDEX_COMPRESS	0

(20 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。
- INDEXJOB_SIZELIMIT：控制索引压缩时单个ADO Job可以处理的最大字节数，单位兆，默认为1024。取值范围为大于等于1小于等于2147483647的整数或浮点数，作用时向下取整。
- ENABLE_INDEX_COMPRESS：索引压缩使能开关，0表示关闭，1表示打开，默认为0。

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

7.9.4.5 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_ilmmobjects 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。
以下参数仅支持高级压缩（ADVANCED），不支持跨页透明压缩（TURBO）。
- 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]，表示一行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。

- INDEXJOB_SIZELIMIT：控制索引压缩时单个ADO Job可以处理的最大字节数，单位兆，默认为1024。取值范围为大于等于1小于等于2147483647的整数或浮点数，作用时向下取整。
- ENABLE_INDEX_COMPRESS：索引压缩使能开关，0表示关闭，1表示打开，默认为0。

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 $240/15 \times 10240\text{MB} = 160\text{GB}$ 数据的评估压缩。压缩带宽为100MB/秒，实际压缩仅耗时 $160\text{GB} / (100\text{MB/秒}) = 27$ 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。

7.9.4.6 运维 TIPS

须知

高级压缩特性使用前必须先联系工作人员申请License，否则执行相关命令会报错。

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 limit 1;  
task_id | job_name | job_state | start_time  
| completion_time | comments | statistics |  
node_name  
-----+-----+-----+-----+-----+-----+-----  
+-----+-----+-----+-----+-----+-----+-----  
+-----+-----+-----+-----+-----+-----+-----  
-  
20004 | ilmjob$postgres_test_202503062047417126137 | COMPLETED SUCCESSFULLY |  
2025-03-06 20:47:41.980633+08 | 2025-03-06 20:47:59.136747+08 | |  
SpaceSaving=98503180,BoundTime=1741265260,StartBlkNum=0,LastBlkNum=40709 |  
gs_datanode3  
(1 row)
```


- e. 触发索引build时压缩，若ENABLE_INDEX_COMPRESS已经设置为1，且表添加了ILM策略，此时若在该表CREATE INDEX，则会触发索引build compress，减少索引build时所生成的页面数。
2. 手动停止压缩。
- ```
gaussdb=# CALL DBE_ILM.STOP_ILM (1, FALSE, 'test_job'); -- task_id为1, p_drop_running_Jobs为FALSE, p_Jobname = 'test_job'。
```

表 7-12 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。
- e. 若需开启索引压缩自动调度任务，需要将ENABLE_INDEX_COMPRESS设置为1，即DBE_ILM_ADMIN.CUSTOMIZE_ILM(27, 1)。
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（索引压缩不支持）。

```
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 7-13 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	1：未压缩 2：高级压缩
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 7-14 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（索引压缩不支持）。

```
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 7-15 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 7-16 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。
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 |
-----+-----+-----+-----+-----+-----+-----+
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
```



```
INDEXJOB_SIZELIMIT      | 1024
ENABLE_INDEX_COMPRESS    | 0
(20 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,StartBlkNum=0,LastBlkNum=40
(1 row)

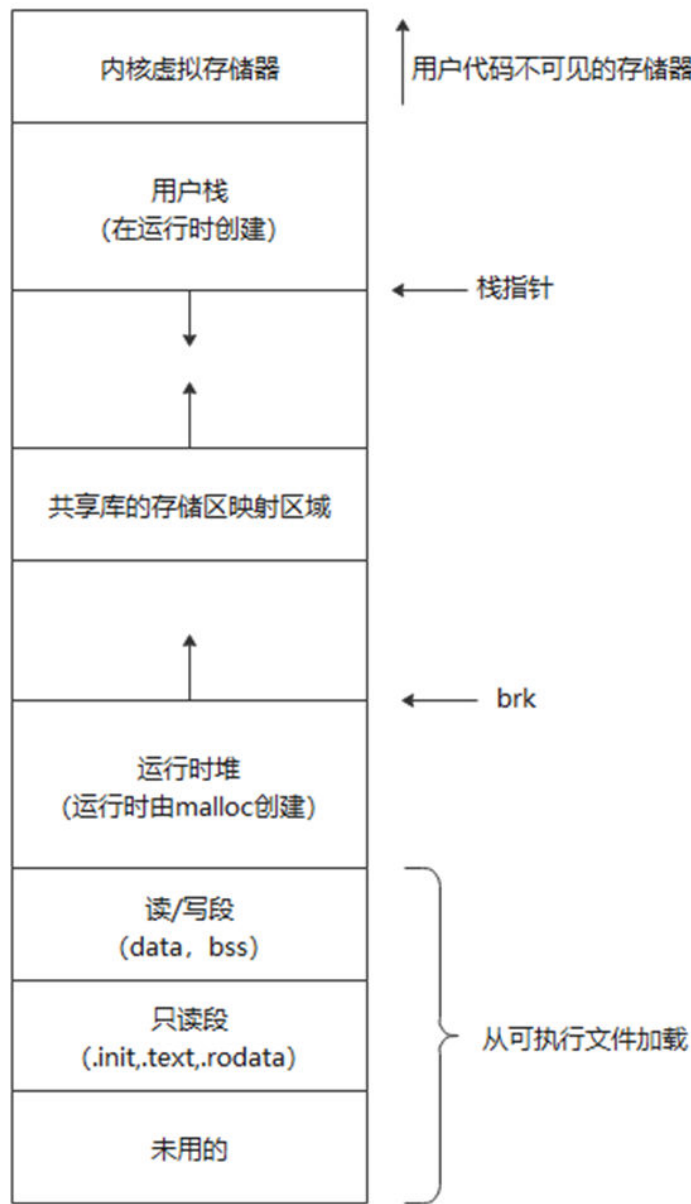
gaussdb=# SELECT * FROM GS_MY_ILMRESULTS;
 task_id | job_name | job_state | start_time | completion_time | comments | statistics
-----+-----+-----+-----+-----+-----+-----
(0 rows)
 task_id | job_name | job_state | start_time | completion_time |
comments | statistics
-----+-----+-----+-----+-----+-----
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,StartBlkNum=0,LastBlkNum=40
(1 row)
```

7.10 内存管理

Linux 内存机制

- 物理内存
物理可见的硬件，即内存条。
- 虚拟内存
进程运行时所有内存空间的总和，以32位系统为例，每个进程有4GB的内存空间，各进程的内存空间具有类似的结构，如图7-19所示：

图 7-19 Linux 内存机制



- 一个新进程建立的时候，将会建立起自己的内存空间，并将此进程的数据、代码等在需要被读写时从磁盘复制到自己的内存空间。进程中每一条数据的地址都由进程控制表中的task_struct记录。task_struct包含一张链表，用来记录内存空间的分配情况，哪些地址有数据、哪些地址无数据、哪些可读、哪些可写都可以通过这个链表记录。
- 每个进程已经分配的内存空间，都与对应的磁盘空间存在映射关系。

说明

建议不要对进程虚拟内存大小进行限制。

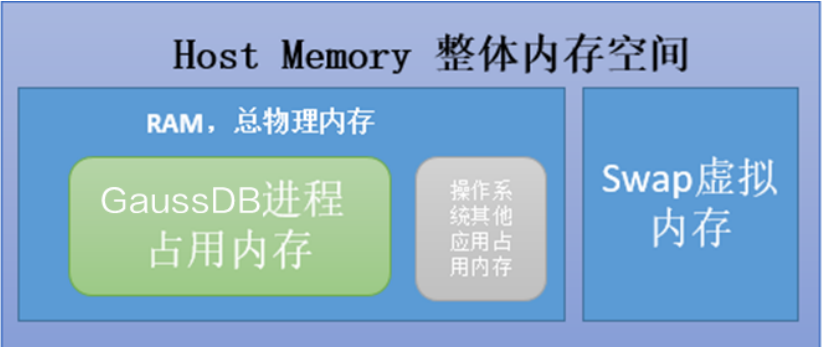
- 物理内存地址和虚拟内存地址
 - 内存分页机制

- 虚拟内存中的页（page）和物理内存页帧（page frame）相同大小。
- 页表（page table）维护虚拟内存页到物理内存页的映射。
- 页面失效（page fault）功能：操作系统根据特定的替换策略（如最少使用，LRU）选择一个虚拟页帧，使其失效，并将其内容写入磁盘。然后将需要访问的页加载到该页帧中，并更新页表中的映射关系。
- 虚拟内存地址：页号（与页表中的页号关联）和偏移量（页的大小）组成。
- 虚存寻址：页表中找到页帧号，若失效则调入页，页帧号及偏移量传给内存管理单元，组成一个物理存在的地址，即可访问物理内存数据。
- Buffer
 - 根据磁盘的读写设计的，把分散的写操作集中进行，减少磁盘碎片和硬盘的反复寻道，从而提高系统性能。
 - Linux有一个守护进程定期清空缓冲内容（即写入磁盘），也可以通过sync命令手动清空缓冲。
- Cache
 - 将读取过的数据保存起来，重新读取时若命中（找到需要的数据）则不读取硬盘，若未命中，则读取硬盘。其中的数据会根据读取频率进行组织，把读取最频繁的内容放在最容易找到的位置，把不再读的内容不断往后排，直至从中删除。
- OOM
 - 内存溢出（Out Of Memory，简称 OOM）是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于能提供的最大内存。此时程序无法运行，系统会提示内存溢出，有时会自动关闭软件，通过重启机器或软件释放掉一部分内存后又可以正常运行该软件，而由系统配置、数据流、用户代码等原因导致的内存溢出错误，即使用户重新执行任务依然无法避免。

GaussDB 内存资源管理

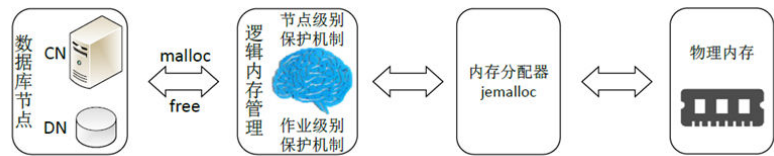
- 内存管理简介
 - GaussDB的动态内存使用方法均基于内存上下文管理，在内存上下文的机制上，引入了逻辑内存管理机制，同时提供多项视图来追踪内存使用情况。
当前GaussDB的内存管理分为两级，分别是：
 - 内存节点级别控制（非透明多写）：通过max_process_memory参数限制CN、DN上可以使用的内存上限。
 - 内存作业级别控制：限制单条query语句最大可使用的内存上限。
- 节点级别内存控制
 - 在Linux操作系统中，内存空间通常由物理内存RAM和虚拟内存Swap组成。其中，物理内存由数据库程序和非数据库程序共同使用；而数据库程序的内存空间由GaussDB进程共同使用。如图7-20所示：

图 7-20 Host Memory 整体内存空间



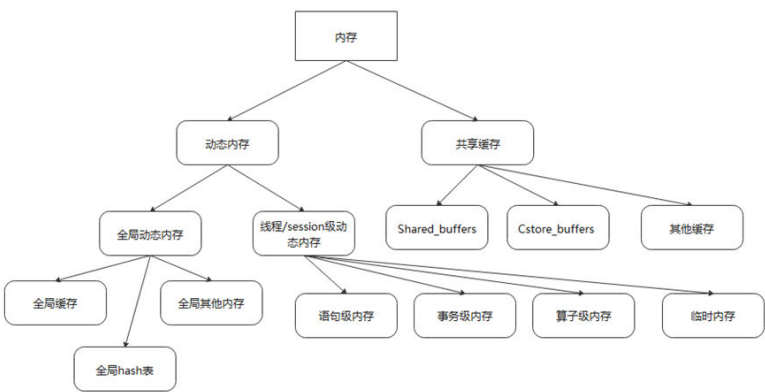
- 由于非数据库程序也需要使用内存，一般建议GaussDB占系统可用内存的80%。
- 作业级别内存控制
 - 提供query_mem参数，根据能使用的内存量和能同时运行的作业数，指定每个作业执行时能够使用的内存量；提供query_max_mem参数，指定每个作业执行时能够使用的内存上限。
 - （非透明多写）提供work_mem参数，设置内部排序操作和Hash表在开始写入临时磁盘文件之前使用的内存大小；提供maintenance_work_mem参数，设置在维护性操作（比如VACUUM、CREATE INDEX、ALTER TABLE ADD FOREIGN KEY等）中可使用的最大内存。
- 逻辑内存管理机制
 - GaussDB内存管理是通过逻辑内存管理来实现的。其实现原理是在原始的内存分配之上增加一层逻辑内存管理，通过检查已分配内存是否超过规定的内存来决定是否为作业分配内存。逻辑内存管理没有更改原有的内存资源分配机制，仅在分配内存之前增加一个逻辑判断层，查看是否达到允许使用的内存上限，来决定是否分配内存。详情如图7-21所示：

图 7-21 逻辑内存管理机制（非透明多写）



- 内存分类

图 7-22 内存分类（非透明多写）



GaussDB的内存使用分类如上所述，其每种类型的说明如下：

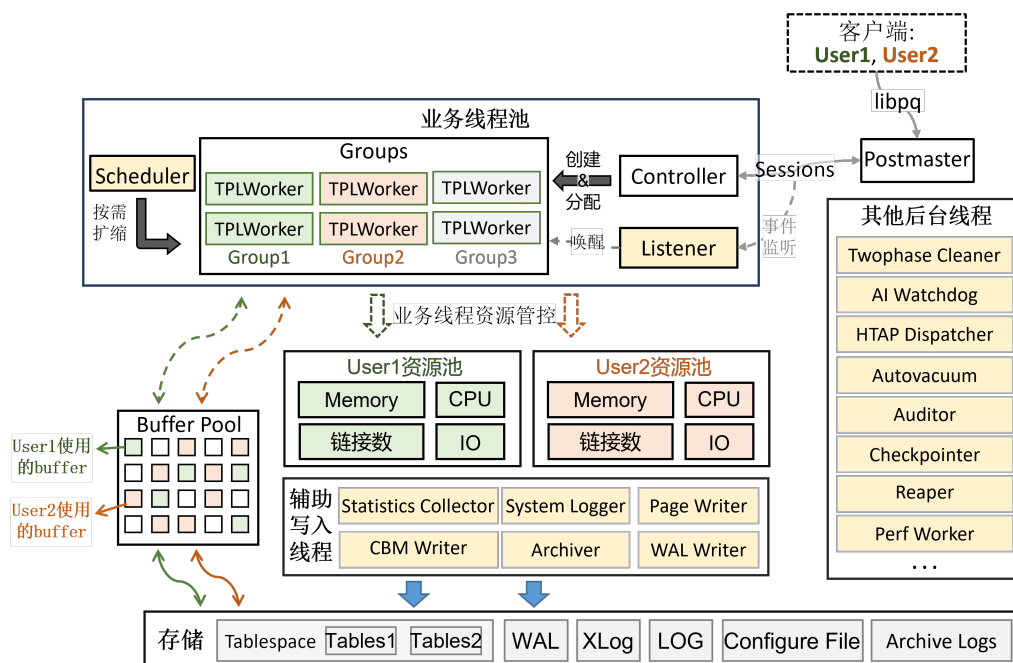
- 动态内存
 - 全局动态内存
 - 全局缓存：全局缓存（Cache）用来提高访问效率，Cache中包括一个系统表元组Cache和一个表模式信息Cache（RelCache），SysCache中存放的是最近使用过的系统表的元组，而RelCache中包含所有最近访问过的表的模式信息（包含系统表的信息）。RelCache中存放的不是元组，而是RelationData数据结构，每一个RelationData结构表示一个表的模式信息，这些信息都由系统表元组中的信息构造而来。
 - 全局hash表：保存全局的统计信息或其他数据的内存hash结构。
 - 全局其他内存：保存全局变量等内容。
 - 线程/session级动态内存
 - 语句级内存：生命周期为整个SQL语句执行阶段，语句执行前内存被申请，执行结束后内存被回收释放。
 - 事务级内存：生命周期为整个事务内，事务开始前内存被申请，事务结束后内存被回收释放。
 - 算子级内存：生命周期为单个算子有效期内，算子执行结束后内存被释放回收。
 - 临时内存：一些临时变量申请的内存，使用后及时释放。
- 共享缓存
 - Shared_buffers
缓存从磁盘读取的数据所使用的内存。
 - 其他缓存
线程所使用的槽位、锁等资源占用的内存。
- 内存上下文机制
 - 内存上下文介绍
 - 概念：内存上下文用来统一控制内存的申请和释放，内存上下文中记录了其所申请的总内存大小，每一处内存申请的大小，申请内存所在的文件、行号等信息以及与其他内存上下文之间的关联信息。
 - 目的：基于数据库中作业执行逻辑，通过内存上下文管理机制，提升查询内存分配、释放效率。
 - 逻辑内存管理（非透明多写）
 - 通过数据库进程内部记账，控制单进程内存使用的上限。
 - 可统计进程总体使用、共享和非共享内存上下文使用及重点模块内存使用（通信库、pooler等）。
 - 逻辑内存的生效机制：enable_memory_limit设置为on，并且max_process_memory减掉缓存等其他预留内存 > 2GB，具体详情请联系管理员。
 - PV_TOTAL_MEMORY_DETAIL视图用于查询数据库进程的内存使用情况，便于分析内存失败后问题，具体字段可参考《参考》中“系统表和

系统视图 > 系统视图 > 其他系统视图 > PV_TOTAL_MEMORY_DETAIL”
章节。

7.11 多线程架构

GaussDB采用单进程多线程架构，这是其高性能和高并发处理能力的关键因素。通过有效的线程调度和管理，GaussDB能够在单个进程中实现高效且稳定的数据处理。线程的种类和功能丰富多样，包括用户线程、系统线程、I/O 线程、日志线程等，线程之间协同工作，共同完成数据库的各项任务。在多线程架构下，更容易实现多个线程资源的共享，例如并行查询和线程池。GaussDB进程架构如图7-23所示。

图 7-23 GaussDB 进程架构



多线程具有以下四大主要优势：

优势一：低启动开销

线程的启动开销远小于进程的启动开销。与进程相比，线程是一种更为“节俭”的多任务操作方式。在Linux系统下，启动一个新进程需要分配独立的地址空间，并建立多种数据表来维护其代码段、堆栈段和数据段，这是一种“昂贵”的多任务工作方式。相反，运行于同一进程中的多个线程共享相同的地址空间和大部分数据，因此启动一个线程所需的空间远小于启动一个进程。

优势二：便捷的通信机制

不同进程间具有独立的数据空间，数据传递只能通过通信方式进行，这不仅费时且不方便。而同一进程下的线程共享数据空间，一个线程的数据可以直接被其他线程使用，这不仅快捷而且方便。

优势三：低切换开销

在Linux系统中，进程切换分为两步：切换页目录以使用新的地址空间以及切换内核栈和硬件上下文。对于线程切换，切换页目录以使用新的地址空间是不必要的，只需执行切换内核栈和硬件上下文。因此，线程切换的开销明显小于进程切换。

优势四：较小的系统调用开销

在多线程架构中，系统调用的开销相对较小，因为线程切换仅涉及用户态和内核态之间的切换，而进程切换还需涉及用户态和内核态之间的切换以及地址空间的切换。

主要线程类型

在启动GaussDB时，main函数首先启动服务守护线程Postmaster。随后，Postmaster线程负责启动其他辅助线程，因此Postmaster是所有线程的父线程。当用户的SQL请求到达时，服务守护线程会启动一个worker线程，由该worker线程处理用户请求，执行SQL并返回数据。除了主业务线程外，GaussDB还提供了一些辅助线程，用于实现维护和调度功能。主要线程及其功能介绍如表7-17所示。

表 7-17 GaussDB 中的主要线程类型

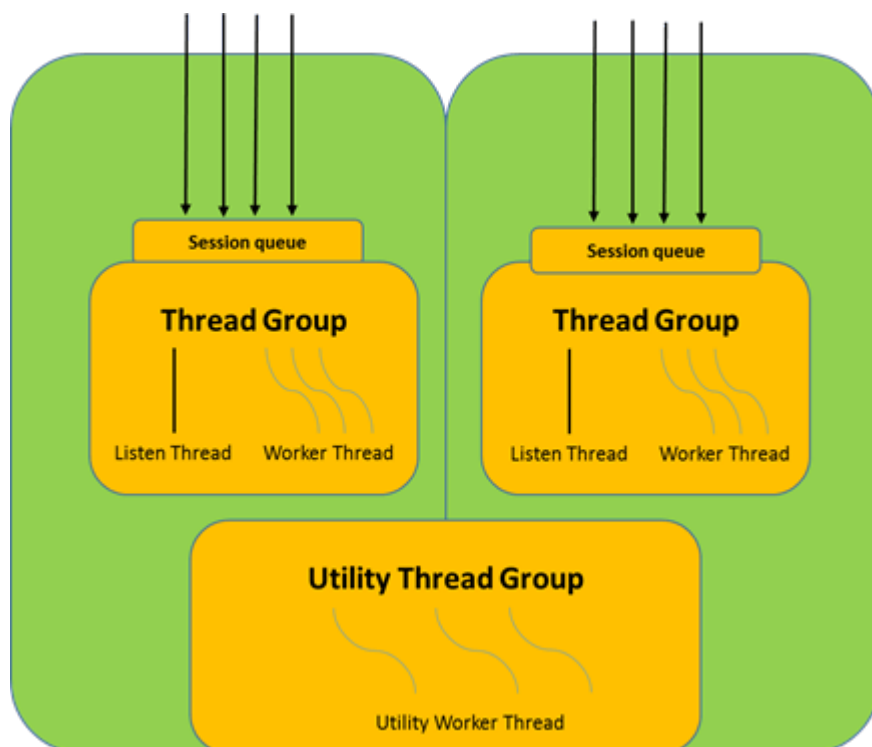
线程	功能介绍
main thread	数据库主线程。 主要有两个功能： 1. 连接监听和接收： - 主线程负责监听来自客户端的连接请求，并接收新的连接。 - 当有新的连接请求到达时，主线程处理连接并分配相应的资源，以确保客户端能够与数据库进行通信。 2. 工作线程监控和处理： - 主线程持续监控所有工作线程的状态，确保工作线程正常运行。 - 根据工作线程的退出状态进行相应处理： - 如果工作线程因FATAL错误退出，主线程会重新启动该工作线程。 - 如果工作线程因PANIC错误退出，主线程会触发整个数据库的重新初始化，以确保数据库系统的稳定性和一致性。
worker thread	工作线程。 GaussDB会根据需要创建多个工作线程来处理SQL语句的执行。每个工作线程都会独立地执行SQL语句，并将结果返回给主线程。
connect thread	建立连接线程。 GaussDB会根据需要创建多个建立连接线程来处理客户端建立连接请求。建立连接完成后将session移交给工作线程执行SQL语句。只有当enable_thread_pool和enable_connect_thread_pool打开时启用此线程类型。
startup	数据库启动线程。 数据库启动时，Postmaster主线程会启动第一个子线程，即Startup线程。该线程主要负责完成数据库的日志REDO（重做）操作，以进行数据库的恢复。日志REDO操作结束后，如果数据库不是备机，Startup线程就会退出。如果数据库是备机，Startup线程将持续运行，以处理接收到的新的日志。

线程	功能介绍
background writer	后台数据写线程。 将数据库数据缓冲区的内容周期性地同步到磁盘上，以确保数据持久性。
checkpointe r	检查点线程。 负责进行检查点操作，完成数据库的周期性检查点和执行检查点命令，以确保数据库的一致性和恢复能力。
WAL writer	后台WAL写线程。 将日志缓冲区的内容周期性地同步到磁盘上，以确保事务日志的持久性。
system logger	运行日志写线程。 将各个线程的运行日志信息写入运行日志文件中，以便于系统监控和故障排查。
autovacuum launcher	垃圾清理启动线程。 主要功能为：通知Postmaster启动一个垃圾清理线程，以及平衡各个垃圾清理线程的负载，以优化系统性能。
autovacuum worker	垃圾清理线程。 对GaussDB数据库中的垃圾数据进行清理，以释放存储空间和提高系统性能。
archiver	日志归档线程。 完成归档操作，将在线日志复制到归档目录，以确保日志的长期保存和备份。

线程池机制

在系统高并发环境下，由于系统资源争抢（包括外部资源（如CPU、网络、I/O等）以及内部资源（如关键锁结构）），往往会导致系统性能明显劣化，同时系统稳定性下降。因此，需要采用线程池技术实现业务连接（session）和后台服务线程（worker thread）的解耦，以确保资源始终处于可控状态，并保证所有业务有序执行，避免相互之间的资源争抢。GaussDB线程池模型如图7-24所示。

图 7-24 GaussDB 线程池模型



线程池技术的整体设计思想是将线程资源池化，并在不同连接中直接复用。系统在启动后，会根据当前用户配置启动一定数量的工作线程。一个工作线程可以服务一个或多个连接会话。当有新的请求到达时，线程池会分配一个空闲线程来处理请求。处理完毕后，线程会归还线程池，等待下一个请求的到来。这样可以避免频繁地创建和销毁线程，从而提高系统的响应速度和资源利用率。

GaussDB的线程模型采用了以下设计以实现有效地处理并发请求，提高数据库的性能和响应能力。

- 主线程：负责连接管理和请求调度。
- 工作线程池：处理客户端请求。
- 并发控制机制：保证数据的一致性和并发性。

7.12 ABO 优化器

智能优化器（AI-based Optimizer）使用AI算法从基数估计、代价估计、计划选择等方面提升优化器的准确度，从而避免出现慢SQL，提高数据库的端到端性能。目前，支持智能（多列）基数估计、自适应计划选择和自适应代价估计。

7.12.1 智能基数估计

7.12.1.1 特性概述

功能定义

智能基数估计通过利用库内贝叶斯网络模型，对多列数据样本的联合分布进行建模。这一过程使得它能够为多列等值查询提供更为精准的基数估算结果。在此基础上，又

能显著提高优化器对于计划和算子选择的准确性。最终，优化器准确性的提升会带来数据库整体吞吐量的提高。

用户场景

在大型企业核心系统和云数据库等高负载业务环境中，复杂查询（尤其是涉及多列组合条件）常因优化器代价估计不准而未能选择预期索引，产生大量慢SQL。此问题的根源在于：当数据存在显著分布倾斜且多列间存在强相关性时，依赖传统统计信息（如直方图）或独立性假设的基数估计器难以准确预测查询结果集大小，导致执行计划次优。这些慢SQL不仅大幅增加关键业务接口响应时间，更会耗尽数据库资源，严重拖累业务成功率和系统整体可用性。本功能目标是利用轻量级数据统计模型，针对多维数据分布进行刻画并且给优化器提供准确选择率估计的能力，进而提高多列场景基数估计的准确率，提升端到端执行性能。

规格约束

该特性自503.1版本引入。使用该特性后的查询性能可能会受模型缓存数量影响，若访问模型数量超过当前缓存个数，会产生内存替换和磁盘访问，有性能下降风险，因此建议合理设置模型缓存个数，模型个数等于创建的BayesNet类型统计信息个数，建议设置为一段时间内能够访问到的统计列个数，当前支持的缓存模型个数默认100个，缓存个数可以通过ai_stats_cache_limit进行调整。

技术架构

智能基数估计方案将轻量概率图模型融合进GaussDB的传统统计信息模块，在统计收集阶段进行模型训练，并且将模型保存在系统表中供优化器使用。

数据库接受特定分析查询语句，判断语句是否包含多列并且调用数据统计模块，数据统计模块在接收ANALYZE指令之后首先针对包含的列进行数据采样，然后针对数据样本进行数据统计，如果数据统计模块发现智能统计GUC参数开启，则进行贝叶斯网络模型的创建，具体包括：

- 利用聚合操作统计并计算出列两两之间的相关性。
- 使用chow-liu算法生成一个树型贝叶斯网络结构。
- 调用贝叶斯网络算子通过遍历数据样本进行模型参数训练，并且将模型参数以二进制的形式存入系统模型表gs_model_warehouse中，并返回模型名称等模型元信息。
- 将模型元信息写入系统统计表中。
- 如果multi_stats_type参数为"ALL"，那么传统统计信息也将存入模型。

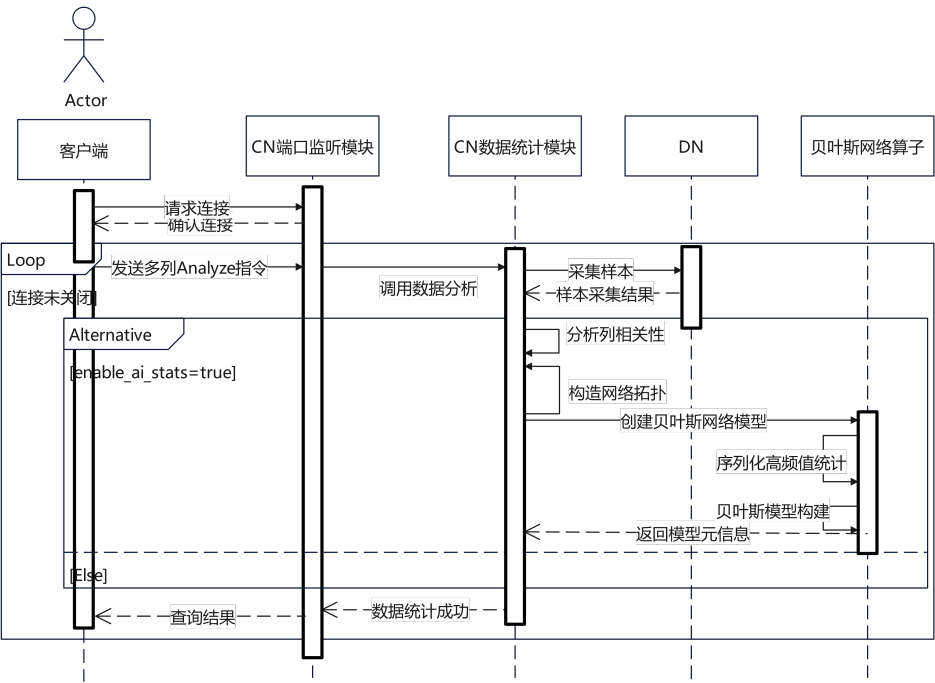
7.12.1.2 技术原理

核心机制

- 模型创建
本功能由用户提交ANALYZE命令或者auto-analyze触发，通过CN样本采集器获取内存中数据样本结构并且包装成数据源结构体。如下图7-25所示，数据源结构体包含两部分信息，一部分是可能存在的高频值统计信息，另一部分是全量数据样本信息。高频值信息包含频度大于1的值（如果等于1，则不认为是高频值），而全量样本信息则是压平为序列一条一条发送给模型。因此模型可以通过获取到的频度判断当前读到的数据是高频值或者是普通样本数据。对于高频值统计数据，

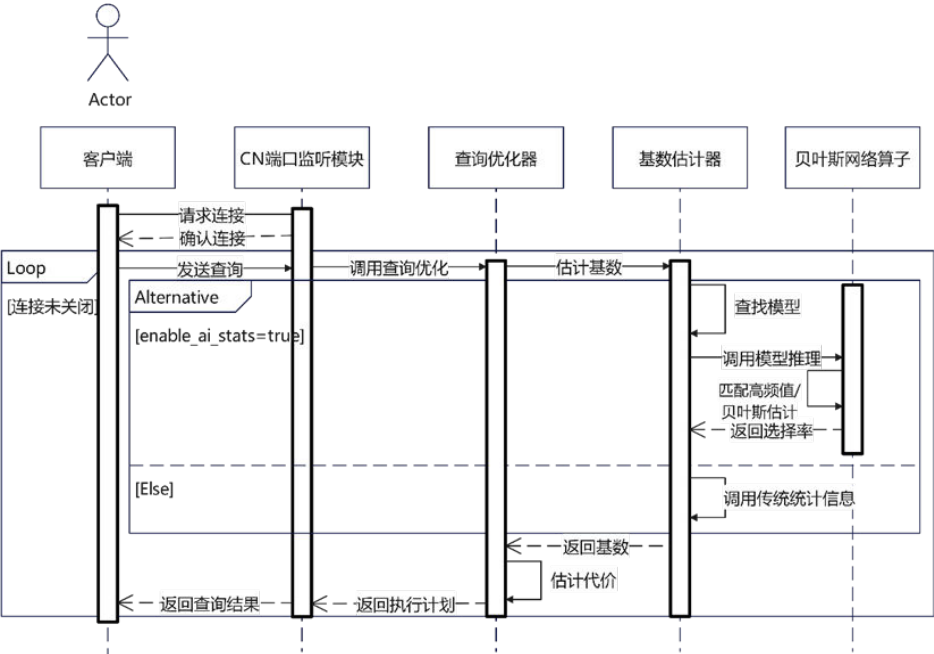
模型算子直接将其序列化保存到模型中，而对于普通样本数据，则对其不属于高频值的部分做贝叶斯网络模型统计。

图 7-25 模型创建序列图



- 模型推理
在基数估计阶段，系统首先将模型从磁盘读到内存中，然后将模型进行反序列化成为内存结构并且使用独立内存上下文进行缓存。如下图7-26所示，对于每条多列等值查询，首先查找高频值哈希表，如果找到则直接返回选择率 $P1(t)$ ，否则计算查询条件在贝叶斯网络模型中的选择率 $P2(t)$ ，最后返回 $P2(t) / (1.0 - SUM(P1))$ ，其中 $SUM(P1)$ 是所有高频值选择率的和。使用本功能后，GaussDB中的多列等值选择率估计优先级为，有高频值优先从高频值哈希表中获取，非高频值通过贝叶斯网络获得选择率，如果上述模型都未创建或者覆盖，则尝试使用多列NDV进行估计，如果统计信息和查询列未匹配，那么采用多列函数依赖。如果上述统计信息均未创建，则将clause条件加回不匹配条件中，然后使用单列+独立性假设的方式估计。

图 7-26 模型推理序列图



- 模型信息显示
调用系统函数gs_ai_stats_explain(relid, stakeys)，其中参数relid为整型，stakeys为数组。该系统函数获得输入参数后会和系统表pg_statistic_ext中的统计信息进行匹配，查找类型为BAYESNET类型的统计信息行，如果找不到，直接反馈失败；否则调用相应的模型解释框架函数，即贝叶斯网络的explain函数，从磁盘中读取模型，反序列化之后将信息重新组合成可读类型，构建成数组返回给系统函数。其中主要包含的内容有贝叶斯网络拓扑、高频值及其选择率、单列上的分桶类型、直方图或者MCV边界值、每个分桶中每个可能值的概率值。其余还包括一些模型通用信息，比如模型训练时间、模型超参数、模型创建时间等。

性能指标

- 高倾斜相关性数据上多列等值查询准确率和涉及索引选择的典型场景端到端性能有效提升。
- TPC-C等场景无劣化。

接口介绍

内置系统函数：

函数名	函数说明
gs_ai_stats_explain	打印对应表和列上的多列智能统计信息。

7.12.1.3 操作指导

前提条件

- 软件版本：内核版本≥GaussDB Kernel 503.1.0.B038。

- 数据库运行正常。
- GUC参数enable_ai_stats设置为on。
- multi_stats_type设置为'BAYESNET'或者'ALL'。
- GUC参数enable_fast_query_shipping设置为off

操作步骤

- 步骤1** 设置采样方式为按照采样率采样，即设置GUC参数default_statistics_target为[-100, -1]之间的整数，表示采样百分比。
- 步骤2** 使用ANALYZE table_name((column1_name,column2_name[, ...])) 进行数据统计和模型创建。
- 步骤3** 输入查询，如果查询涉及到的等值查询列上有统计模型创建，会自动使用统计模型进行选择率估计。
- 步骤4** 不再需要智能统计模型时，使用ALTER TABLE table_name DELETE STATISTICS ((column1_name,column2_name[, ...]))进行统计信息以及模型删除。

----结束

其他使用的方法详见《参考》中“SQL参考 > SQL语法 > A > ALTER TABLE”和“SQL参考 > SQL语法 > A > ANALYZE | ANALYSE”章节。

示例

受环境、版本及部署形态差异影响，示例中的执行计划可能与实际不同。本示例仅作参考，具体优化效果请以实际执行计划为准。

```
-- 建表并插入数据
gaussdb=# DROP TABLE IF EXISTS t1;
DROP TABLE
gaussdb=# CREATE TABLE t1(a INT , b INT, c INT);
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.
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(GENERATE_SERIES(1, 1000),1,1);
INSERT 0 1000
-- 检查GUC参数
gaussdb=# SHOW enable_ai_stats;
enable_ai_stats
-----
on
(1 row)
gaussdb=# SHOW multi_stats_type;
multi_stats_type
-----
bayesnet
(1 row)
gaussdb=# SHOW enable_fast_query_shipping;
enable_fast_query_shipping
-----
off
(1 row)
gaussdb=# SHOW explain_perf_mode;
explain_perf_mode
-----
normal
(1 row)
-- 创建统计模型
gaussdb=# ANALYZE t1((a,b));
ANALYZE
```

[illegible]

[illegible]

7.12.1.4 最佳实践

以下表及数据来源TPC-H 50X。

生成如下数据表:

```
gaussdb=# \d part;
id                | integer                |
p_brand           | character varying(256) |
p_type            | character varying(256) |
p_container       | character varying(256) |
p_mfgr            | character varying(256) |
```

插入10,000,000行数据:

```
gaussdb=# SELECT COUNT(1) FROM part;
10000000
```

在数据表上创建四种不同的多列索引：

```
gaussdb=# SELECT * FROM pg_indexes WHERE tablename='part';
public | part | brand_type_container | | CREATE INDEX brand_type_container ON part USING
btree (p_brand, p_type, p_container) TABLESPACE pg_default
public | part | brand_type_mfgr | | CREATE INDEX brand_type_mfgr ON part USING btree
(p_brand, p_type, p_mfgr) TABLESPACE pg_default
public | part | brand_container_mfgr | | CREATE INDEX brand_container_mfgr ON part USING
btree (p_brand, p_container, p_mfgr) TABLESPACE pg default
```



```
public | part | type_container_mfgr | | CREATE INDEX type_container_mfgr ON part USING  
btree (p_type, p_container, p_mfgr) TABLESPACE pg_default
```

针对数据表生成一批包含多列等值条件的查询，如下：

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM part WHERE p_container='LG CASE' AND  
p_brand='Brand#34' AND p_mfgr='Manufacturer#2' AND p_type='SMALL BRUSHED COPPER';
```

分别测试不创建多列统计信息和创建ABO统计信息场景下的执行计划：

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM part WHERE p_container='LG CASE' AND  
p_brand='Brand#34' AND p_mfgr='Manufacturer#2' AND p_type='SMALL BRUSHED COPPER';  
Bitmap Heap Scan on part (cost=5.30..336.06 rows=17 width=56) (actual time=0.953..7.061 rows=103  
loops=1)  
  Recheck Cond: (((p_brand)::text = 'Brand#34'::text) AND ((p_type)::text = 'SMALL BRUSHED COPPER'::text)  
AND ((p_container)::text = 'LG CASE'::text))  
  Filter: ((p_mfgr)::text = 'Manufacturer#2'::text)  
  Rows Removed by Filter: 773  
  Heap Blocks: exact=871  
-> Bitmap Index Scan on brand_type_container (cost=0.00..5.30 rows=84 width=0) (actual  
time=0.704..0.704 rows=876 loops=1)  
  Index Cond: (((p_brand)::text = 'Brand#34'::text) AND ((p_type)::text = 'SMALL BRUSHED COPPER'::text)  
AND ((p_container)::text = 'LG CASE'::text))  
Total runtime: 7.213 ms  
gaussdb=# EXPLAIN ANALYZE SELECT * FROM part WHERE p_container='LG CASE' AND  
p_brand='Brand#34' AND p_mfgr='Manufacturer#2' AND p_type='SMALL BRUSHED COPPER';  
Bitmap Heap Scan on part (cost=10.59..723.97 rows=210 width=56) (actual time=0.112..0.434 rows=103  
loops=1)  
  Recheck Cond: (((p_type)::text = 'SMALL BRUSHED COPPER'::text) AND ((p_container)::text = 'LG  
CASE'::text) AND ((p_mfgr)::text = 'Manufacturer#2'::text))  
  Filter: ((p_brand)::text = 'Brand#34'::text)  
  Rows Removed by Filter: 64  
  Heap Blocks: exact=167  
-> Bitmap Index Scan on type_container_mfgr (cost=0.00..10.54 rows=183 width=0) (actual  
time=0.081..0.081 rows=167 loops=1)  
  Index Cond: (((p_type)::text = 'SMALL BRUSHED COPPER'::text) AND ((p_container)::text = 'LG  
CASE'::text) AND ((p_mfgr)::text = 'Manufacturer#2'::text))  
Total runtime: 0.533 ms
```

通过上述操作可以看出，在该场景下，ABO基数估计使查询速度得到了有效提升。

受环境、版本及部署形态差异影响，示例中的执行计划可能与实际不同。本示例仅作参考，具体优化效果请以实际执行计划为准。

7.12.1.5 常见问题

- 问题：训练报错。
Model training failed, error XXX.
处理方法：
根据具体的error信息检查训练对象，并重试训练。
- 问题：gs_ai_stats_explain报错。
Cannot find table XXX.
处理方法：
检查传参是否为正确的表名。
- 问题：gs_ai_stats_explain报错。
Column XXX of relation XXX does not exist.
处理方法：
检查传参是否为正确的列名。
- 问题：gs_ai_stats_explain报错。
The AI statistic does not exist.
处理方法：

检查传参是否为正确的表或列，或对应的bayes模型是否已创建。

7.12.2 自适应代价估计

7.12.2.1 特性概述

功能定义

自适应代价估计是505.1版本引入的智能优化器增强特性，包含反馈基数估计与自适应代价参数调整两个模块。该特性持续收集历史查询的实际执行结果（查询计划中算子级别的实际行数基数及执行时间），并基于这些数据自动训练和更新反馈基数估计模型，同时根据实际运行结果动态校准代价模型参数。反馈基数估计模型可以建立查询条件与真实基数之间的映射关系，训练完成的模型可实时应用于新查询，预测其算子基数，并结合调整后的代价模型参数，显著提升复杂场景下优化器代价估计的准确性，进而生成更优的执行计划。

用户场景

在大型企业核心系统和云数据库等高负载业务环境中，复杂查询常因优化器代价估计不准而生成次优查询计划，导致大量慢SQL产生。此问题的根源在于：一方面，基数估计器难以准确预测查询结果集大小；另一方面，优化器的代价模型参数配置存在偏差。两者共同作用导致生成的执行计划次优。这些慢SQL不仅大幅增加关键业务接口响应时间，更会耗尽数据库资源，严重拖累业务成功率和系统整体可用性。

规格约束

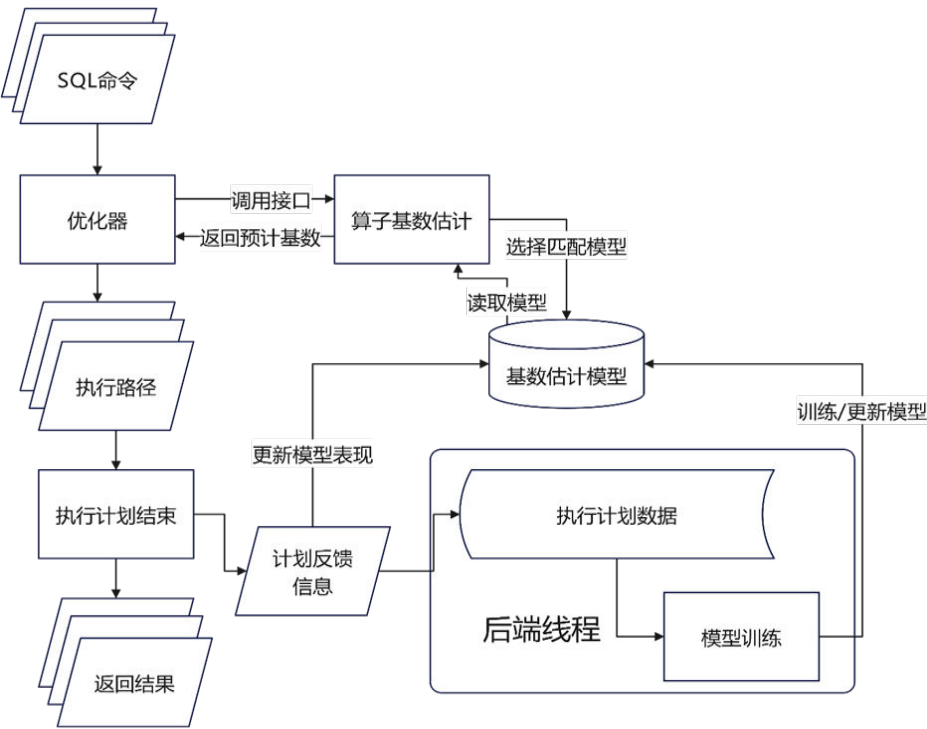
- 只支持对基表扫描算子（SeqScan、IndexScan、IndexOnlyScan、BitmapHeapScan、BitmapIndexScan）和连接算子（NestLoop、MergeJoin、HashJoin）的估计。
- 只支持以下查询条件：简单的连接条件（col1::type =/ <> /> /< col2::type，支持类型转换），数值范围条件（col1::type =/ <> /> /< const，BETWEEN处理为大于和小于两个条件，const类型支持oid、numeric、int、float、timestamp），简单的字符串比较条件（col1::type =/ <> /> /< LIKE/NOT LIKE const，const类型支持BPCHAR、VARCHAR、TEXT）以及IN/OR条件。
- 不支持对分区表、物化视图进行估计。
- 收集到新SQL语句信息后可能会对后续执行计划产生劣化的跳变，只有收集到足够跳变计划信息的优化器才可以逐渐收敛生成最优/较优计划。
- 反馈代价矫正后，经过矫正的智能代价参数只用于SeqScan、IndexScan、IndexOnlyScan、BitmapHeapScan、HashJoin、NestLoop和MergeJoin算子的代价估计。在优化器进行代价估计时，若一个算子及其子算子的类型和条件在反馈基数支持范围内，则会使用反馈模型的基数和矫正代价计算代价；否则会使用默认方法。

技术架构

- 反馈基数估计模块与优化器和执行器协同工作，分为训练与估计两个阶段，如下图7-27所示。在训练阶段，该模块实时从执行器采集历史查询的计划反馈信息（主要包括执行计划树中各算子的实际基数），利用这些数据训练并迭代更新预测反馈基数估计模型，最终将优化后的模型持久化存储至系统表gs_model_warehouse中。在估计阶段，当优化器进行查询优化时，会调用该模块的预测接口，接口自动从gs_model_warehouse加载对应模型，对当前查询计划中

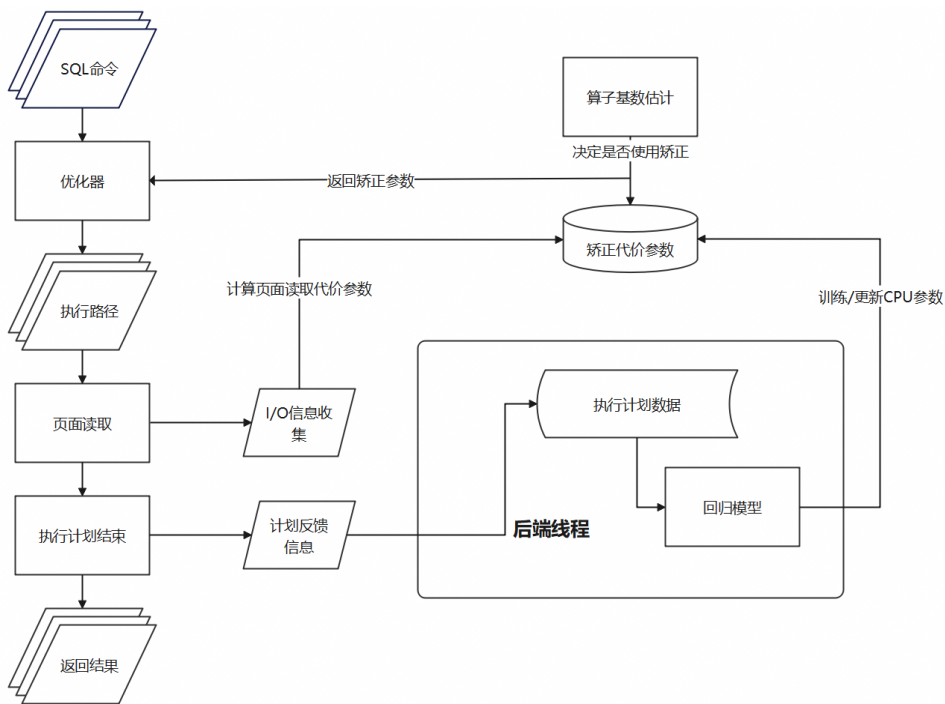
的算子基数进行实时预测，并将高精度的预测结果回传给优化器。模型训练由后台线程异步执行，当收集的算子样本数量累积达到预设阈值，或检测到默认基数估计模型的预测误差超过特定阈值时将唤醒该线程，用户也可主动调用系统函数手动触发更新任务。

图 7-27 反馈基数估计架构



- 自适应代价参数调整的训练信息同样通过与执行器的协同进行收集，主要包含两类数据：I/O耗时信息和执行计划信息。对于I/O耗时，系统在满足特定预定义条件（如模块初始化完成且收集量未达上限）时，会在实际读取数据页的过程中精确测量并累加单次I/O操作的时间消耗以及总读取页面数，如下图7-28所示。该收集过程设有总量限制，确保对系统性能的影响微乎其微。对于执行计划信息，收集发生在查询执行结束时，前提是代价参数尚未完成训练。此时，系统会检查查询是否发生了有效的数据读取。若未读取数据，则转而收集执行计划树中关键算子（如Scan算子）的运行统计信息，包括循环次数、处理元组数量、相关计算操作的CPU开销以及执行耗时。两种信息的收集均为低开销且受控，为核心代价模型的持续优化提供必要输入。

图 7-28 自适应代价参数调整框架



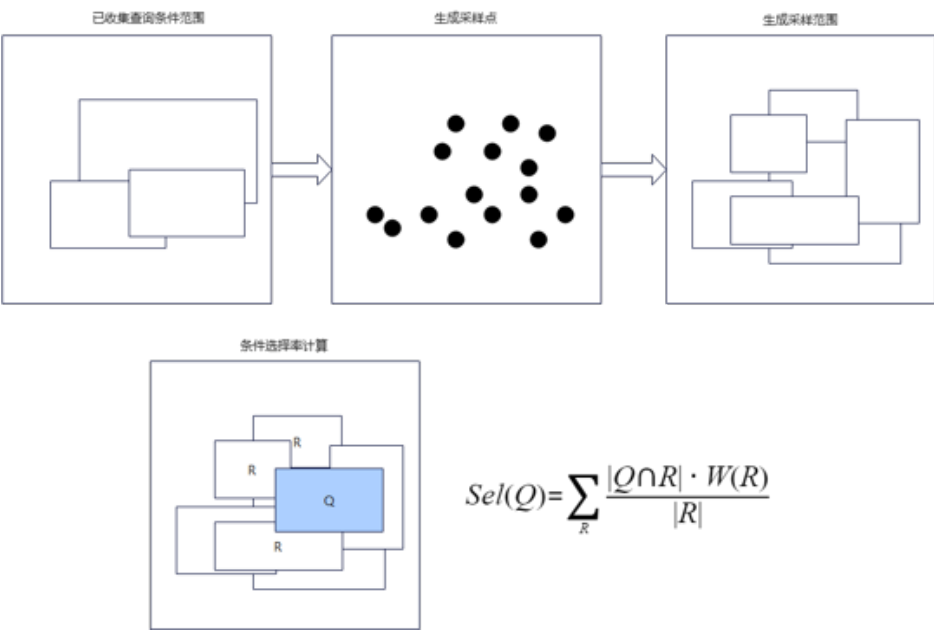
7.12.2.2 技术原理

核心机制

- 反馈基数估计
- 反馈基数估计首先需要为每个待预测算子动态匹配对应的预测模型，其匹配逻辑基于算子特征的唯一哈希编码，该编码综合了算子所涉及的表、属性及查询条件等信息。同时，另一个机制为利用算子级别的历史数据，训练并应用UMM (Unified Mixture Model) 和 KNN (K-Nearest Neighbors) 模型，建模查询条件与实际基数间的关系。
- a. 算子哈希值计算：
- 为准确匹配模型，系统会为每个查询算子生成一个独特的特征标识（哈希值）。该标识的核心在于捕捉算子查询条件的关键模式特征。一个查询算子由四个基本要素描述：
- 属性组（等价类）：表示具有相等关系的属性集合（例如，从a.id = b.id AND b.id = c.id的连接条件中可提取 {a.id, b.id, c.id} 为一组）。
 - 比较操作符：如查询条件中的=、>、<等。
 - 条件值：可以是具体数值（如 a.id = 5），也可以是另一个属性组（例如在a.id < b.id中）。
 - 条件类型：查询条件的类型，包括过滤型查询条件（如 a.id = 5）、连接型查询条件（如 a.id = b.id）等。
- 在计算给定算子的哈希值时，将首先提取以上四个要素，并分别计算四个要素的哈希值，再计算总哈希值，特别地，等价类的哈希值由其包含的所有属性的哈希值组合生成。

- b. UMM模型构建：
- 均匀混合模型 (UMM) 的参数包括：
- n维查询条件空间的边界值，定义了模型覆盖的范围。
 - M个采样得到的超方体，这些超方体位于上述n维查询条件空间内。
 - 每个采样超方体对应的权重。
- 对于一个待预测的查询算子，UMM的基数估计过程如下：
- 根据该算子的查询条件，在n维查询条件空间中确定其对应的查询超方体。
 - 计算该查询超方体与每一个采样超方体的重叠部分的n维测度（在二维情况下即面积，三维即体积，以此类推）。
 - 将这些重叠部分的测度，分别乘以其对应采样超方体的权重。
 - 将所有加权后的测度求和，该总和即为该查询算子的预测基数。
- 以图7-29中的二维情况 (n=2) 为例说明：
- 查询条件空间是一个二维平面，采样超方体即为平面上的矩形（即二维超方体），UMM模型假设在每个采样矩形覆盖的区域内，数据满足均匀分布。
 - 对于待预测算子，其查询条件对应一个矩形区域（图中蓝色矩形所示）。
 - 该查询矩形的预测基数，等于它与所有采样矩形重叠区域面积的加权和（权重为对应采样矩形的权重）。

图 7-29 UMM 模型示例



而UMM模型的训练过程即为构建采样超方体和确定其权重的过程。具体而言，根据收集到的历史查询算子的查询条件信息，确定覆盖这些条件的初始

超立方体集合（代表整个或部分查询空间），从这些初始超立方体中采样，得到M个采样超立方体（子区间），再利用梯度下降算法，训练优化这M个采样超立方体的权重参数。优化的目标是使模型对训练数据的基数预测误差最小。

c. KNN模型构建：

K近邻模型（KNN）核心思路，是利用历史查询数据来建立查询条件的选择率特征与最终查询基数之间的映射关系。

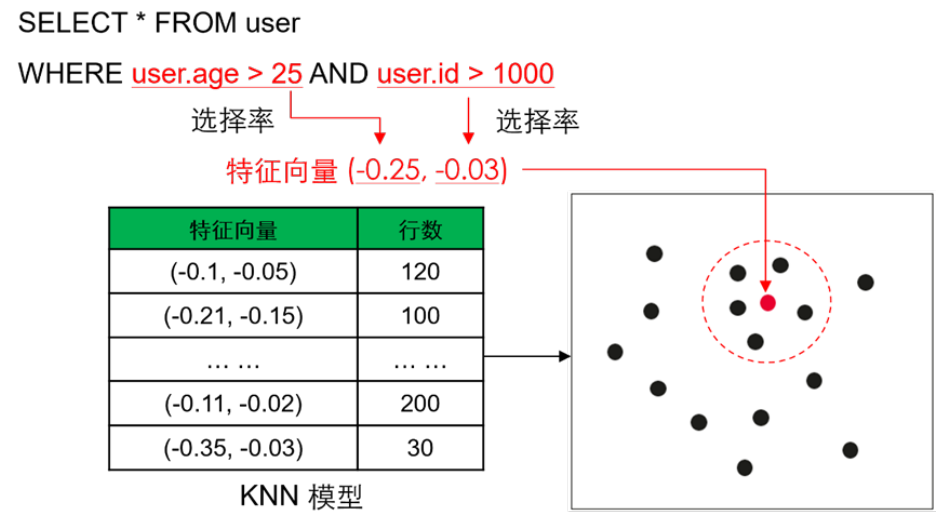
假设关注的算子类型包含若干个（N个）查询条件。每个查询条件都有一个默认的估计选择率，这些选择率值可以组合成一个特征向量（通常记为x）。这个算子对应的实际查询基数就是目标值（通常记为y）。

KNN模型本身并不存储复杂的数学参数，它的核心“知识”来源于其内部保存的历史查询样本。具体来说，一个针对特定查询模式的KNN模型，最多会存储K个历史样本。每个样本都记录了过去一次同类查询的特征向量x（即当时各个查询条件的默认选择率估计值）和其对应的真实查询基数y。

在进行预测时，查询优化器会首先根据当前待估计查询算子的特征（通常通过哈希匹配模式）找到对应的KNN模型。然后，优化器会提取当前查询中各个条件的默认选择率估计值，形成当前的特征向量x。模型接着会在其存储的K个历史样本中，找出与当前特征向量x最相似的K个邻居样本。最终的查询基数预测值（ $\hat{y}$ ）是通过对这K个邻居样本的真实基数（y）进行加权平均得到的。这个加权平均的权重取决于每个邻居样本的特征向量（x）与当前查询特征向量（x）的相似度：两个向量越相似（距离越近），其对应的真实基数在预测中所占的权重就越大。模型使用一种特定的相似度量函数（基于向量间的欧几里得距离）来计算这个权重。

如图7-30的例子所示，在查询条件为2的场景中，每个特征向量可以表示为二维平面上的一个点，右侧图中黑色的点即为KNN模型中的训练数据，在预测过程中，根据待预测的查询条件的选择率，可以得到其特征向量和图中对应的位置（图中的红点），再根据KNN算法，图中红色的圈表示以红点为中心、包含K个最近邻历史样本的搜索范围（即邻域）。这个圈的大小取决于距离红点第K近的历史样本的距离，通过计算当前特征向量（红点）与圈内每个历史样本特征向量（黑点）的相似度（基于欧几里得距离），并根据相似度分配权重，最终通过加权平均预测查询基数（ $\hat{y}$ ）。

图 7-30 KNN 模型示例



- 代价估计参数调整

- a. I/O 成本参数（seq_page_cost / random_page_cost）：
基于实际测量的平均I/O时间（总I/O时间/读取页面数），并依据默认比例关系（seq_page_cost : random_page_cost = 1:4）直接计算：seq_page_cost = 平均I/O时间 / 2，random_page_cost = 平均I/O时间 * 2。
- b. CPU成本参数（cpu_tuple_cost、cpu_index_tuple_cost、cpu_operator_cost）：
 - cpu_tuple_cost（主回归目标）：使用线性回归模型计算。该模型针对顺序扫描（SeqScan）算子建立：执行时间 = 处理元组数 *（cpu_tuple_cost + 表达式单位CPU操作数 * cpu_operator_cost）。
 - cpu_index_tuple_cost：鉴于索引扫描（IndexScan）的性能受顺序性影响显著且分析复杂，该参数不直接回归，而是基于回归得到的cpu_tuple_cost和默认比例设定：cpu_index_tuple_cost = cpu_tuple_cost / 2。
 - cpu_operator_cost：同样不直接回归。考虑到提供的单位CPU操作数可能与实际开销存在较大偏差，且多元回归易受数据质量（如样本同质化）影响，采用默认比例设定：cpu_operator_cost = cpu_tuple_cost / 4。此设定将SeqScan模型简化为无常数项的一元回归：执行时间 = 元组数 * cpu_tuple_cost *（1 + 表达式单位 CPU 操作数 / 4）。
- c. 参数可靠性检验：
 - 训练数据通过滑动窗口管理（默认长度 cost_update_window_size = 5）。
 - 每次窗口数据完全更新后，使用前次训练的参数计算新窗口数据的拟合优度：计算决定系数 $R^2 = 1 - (\sum(\text{估计误差}^2) / \text{窗口长度})$ 。
 - 仅当 $R^2 > 0.95$ 时，才认为本次回归结果可靠并采纳新参数，否则继续收集数据等待后续训练。

性能指标

在开启此功能后，反馈基数估计模型训练收敛的情况下，在JOB（Join-Order-Benchmark）测试集能够产生端到端的平均1.3倍的查询速度提升。

接口介绍

- 核心系统函数：

函数名	函数说明
gs_acm_analyze_workload_manual()	根据当前已收集的算子信息，手动启动模型训练与更新。
gs_stat_get_acm_feedback_operator_info()	查看目前全局内存中已收集到的所有算子信息。
gs_costmodel_calibration_manual()	手动开始收集代价信息并且矫正代价模型参数一次。

具体描述及其他相关系统函数请参见《参考》中“SQL参考 > 函数和操作符 > AI 特性函数”章节。

- 核心GUC参数：

参数名	级别	默认值	描述
enable_adaptive_cost	SIGHUP	on	总功能的开关，控制算子信息收集与反馈基数估计流程的启停，并同步控制后端模型维护线程的启停。
enable_feedback_cardest	USRSET	on	单独控制反馈估计功能的开关，当 enable_adaptive_cost关闭但 enable_feedback_cardest为开启时算子信息会被收集，且反馈基数估计接口仍会被调用，但后端自动模型维护的线程并不会被拉起，可以通过 gs_acm_analyze_workload_manual()函数手动训练模型。
adaptive_cardest_strategy	USRSET	auto	设置选择基数估计模型偏好，可选择类型： ● 'auto'：根据过去预测误差选择模型。 ● 'use_statistics'：优先使用统计信息做基数估计。 ● 'use_feedback'：优先使用反馈模型做基数估计。

adaptive_cost_min_time	USRSET	1000ms	参数用于设置算子收集的SQL时间阈值，只有执行时间大于该值的语句的执行算子才会被收集。
cost_update_window_size	USRSET	5	调整收集用于自适应代价模型参数回归的数据的滑动窗口大小，范围为[1,20]。
adaptive_costtest_strategy	USRSET	L0	设置代价评估使用新/旧代价的策略，当前分为两个等级： • 'L0': 如果计划树代价估计使用的所有基数都是可信任的（基表 SeqScan/基数来自反馈基数/支持反馈基数但尚未训练/估计误差<1.1的默认基数），才会计算和使用新代价做计划选择。 • 'L1': 任何时候都使用新的代价模型计算。

具体描述与其他相关GUC参数请参见《参考》中“数据库运行参数说明 > GUC参数说明 > AI特性”章节。

7.12.2.3 操作指导

前置条件

- 数据库运行正常。
- 反馈基数功能开启。
- 数据库查询解析正常进入计划优化阶段。

使用指导

打开参数enable_adaptive_cost，其他参数使用默认设置即可开启本功能。

反馈基数功能需要先收集基数才能修正统计偏差，只有通过explain (analyze)执行或执行时间超过adaptive_cost_min_time的SQL才会收集基数信息。

当设置adaptive_cardest_strategy=auto（默认配置）时，同类型的SQL（按照sql id区分）第一次执行不会使用反馈基数。对重复执行的同类SQL，会按照过去的执行时间来判断是否可以使用反馈基数。如果出现了严重的计划劣化，即本次执行时间超过了不使用反馈基数平均执行时间的10倍再多5秒，则会将对应的SQL列入黑名单，不再使用反馈基数。例如，某sql不使用反馈基数的平均执行时间为1s，如果使用反馈基数后执行时间超过了 $1 \times 10 + 5 = 15$ 秒，则会列入黑名单。

示例

受环境、版本及部署形态差异影响，示例中的执行计划可能与实际不同。本示例仅作参考，具体优化效果请以实际执行计划为准。

- 通过后台线程自动进行模型训练，调整执行计划。

步骤1 启动数据库，查看enable_adaptive_cost参数是否开启，确保反馈基数估计功能的后台训练已正常开启。

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

步骤2 设置计划展示格式。

```
gaussdb=# SET explain_perf_mode = pretty;
```

步骤3 执行由于错误的基数估计导致次优计划的语句，如含有多种连接路径和连接类型的查询SQL，观察执行计划在多次执行中是否迭代出更优计划。以JOB数据集为例，首先导入JOB数据集，并运行查询语句01b.sql。

```
gaussdb=# EXPLAIN (ANALYZE)
SELECT MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_title,
       MIN(t.production_year) AS movie_year
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info_idx AS mi_idx,
     title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'bottom 10 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND t.production_year BETWEEN 2003 AND 2010
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id;
```

步骤4 观察执行计划可知，总执行时间为1442.612毫秒。重点关注A-rows（实际行数）与E-rows（估计行数）两列数据，可以发现当前计划生成过程中的基数估计存在显著偏差。例如，对于Hash Join(5, 11)算子，其实际行数仅为4行，而估计行数却高达528329行。正是由于此类基数估计偏差，导致该执行计划并非当前最优选择。此时，因后台训练功能已开启，反馈基数估计模块的后台训练线程将被触发，自动执行模型训练任务。

id	operation	A-time	A-rows	E-rows
Peak Memory E-memory A-width E-width E-costs				
1 -> Aggregate		1442.612	1	1

```
29KB | | | | 113 | 107165.90
2 | -> Streaming (type: GATHER) | 1442.467 | 3 | 3 |
150KB | | | | 113 | 107165.90
3 | -> Aggregate | [1382.897,1422.194] | 3 | 3 |
[27KB,28KB] | 1MB | | | 113 | 107165.78
4 | -> Hash Join (5,11) | [1382.845,1422.167] | 4 |
528329 | [10KB,10KB] | 1MB | | | 45 | 105844.94
5 | -> Streaming(type: REDISTRIBUTE) | [435.768,481.418] | 140904 |
1303008 | [69KB,69KB] | 2MB | | | 28 | 55550.75
6 | -> Nested Loop (7,9) | [495.739,559.512] | 140904 |
1303008 | [4KB,4KB] | 1MB | | | 28 | 19718.03
7 | -> Streaming(type: BROADCAST) | [0.063,33.318] | 3 |
3 | [45KB,45KB] | 2MB | | | 4 | 2.30
8 | -> Seq Scan on company_type ct | [0.003,0.074] | 1 |
1 | [32KB,32KB] | 1MB | | | 4 | 2.02
9 | -> Bitmap Heap Scan on movie_companies mc | [475.771,539.464] |
140904 | 1303008 | [74KB,76KB] | 1MB | | | 32 | 15372.37
10 | -> Bitmap Index Scan using company_type_id_movie_companies |
[120.751,130.875] | 1334883 | 1304564 | [356KB,356KB] | 1MB | | | 0 | 3495.96
11 | -> Hash | [833.688,891.113] | 6 | 407937 |
[2MB,2MB] | 16MB | [48,52] | 29 | 44769.35
12 | -> Hash Join (13,14) | [833.646,891.058] | 6 |
407937 | [11KB,11KB] | 1MB | | | 29 | 44769.35
13 | -> Seq Scan on title t | [704.619,725.190] | 868629 |
872254 | [52KB,52KB] | 1MB | | | 25 | 24237.56
14 | -> Hash | [13.300,84.206] | 10 | 345009 |
[1MB,1MB] | 16MB | [20,20] | 4 | 15190.38
15 | -> Streaming(type: REDISTRIBUTE) | [13.221,84.122] | 10 |
345009 | [45KB,45KB] | 2MB | | | 4 | 15190.38
16 | -> Nested Loop (17,19) | [10.662,46.745] | 10 |
345009 | [3KB,3KB] | 1MB | | | 4 | 5702.64
17 | -> Streaming(type: BROADCAST) | [10.177,46.329] | 3 |
3 | [45KB,45KB] | 2MB | | | 4 | 1.75
18 | -> Seq Scan on info_type it | [0.271,0.349] | 1 | 1 |
[32KB,32KB] | 1MB | | | 4 | 1.48
19 | -> Bitmap Heap Scan on movie_info_idx mi_idx | [0.210,0.376] | 10 |
345009 | [23KB,24KB] | 1MB | | | 8 | 4550.86
20 | -> Bitmap Index Scan using info_type_id_movie_info_idx | [0.160,0.271] |
10 | 345009 | [3KB,3KB] | 1MB | | | 0 | 926.57
(20 rows)
```

步骤5 再次执行查询并观察执行计划，可见执行时间已优化（从6763.152毫秒降至5337.537毫秒）。分析基数估计情况，在反馈基数估计模型的预测下，多数算子的基数估计值得到了有效矫正，从而优化了执行计划的生成。

```
gaussdb=# EXPLAIN (ANALYZE)
SELECT MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_title,
       MIN(t.production_year) AS movie_year
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info_idx AS mi_idx,
     title AS t
WHERE ct.kind = 'production companies'
     AND it.info = 'bottom 10 rank'
     AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
     AND t.production_year BETWEEN 2003 AND 2010
     AND ct.id = mc.company_type_id
     AND t.id = mc.movie_id
     AND t.id = mi_idx.movie_id
     AND mc.movie_id = mi_idx.movie_id
     AND it.id = mi_idx.info_type_id;
id | operation | A-time | A-rows | E-rows | Peak
Memory | E-memory | A-width | E-width | E-costs
-----+-----+-----+-----+-----
1 | -> Aggregate | 72.986 | 1 | 1 |
29KB | | | | 113 | 5858.87
2 | -> Streaming (type: GATHER) | 72.860 | 3 | 3 |
```

```
150KB | | | 113 | 5858.87
3 | -> Aggregate | [55.949,56.995] | 3 | 3 |
[26KB,27KB] | 1MB | | 113 | 5858.74
4 | -> Nested Loop (5,7) | [55.815,56.920] | 4 | 57132 |
[8KB,8KB] | 1MB | | 45 | 5715.90
5 | -> Streaming(type: BROADCAST) | [0.099,0.197] | 3 | 3 |
[45KB,45KB] | 2MB | | 4 | 2.30
6 | -> Seq Scan on company_type ct | [0.001,0.038] | 1 | 1 |
[32KB,32KB] | 1MB | | 4 | 2.02
7 | -> Materialize | [55.498,56.637] | 81 | 19 |
[14KB,18KB] | 16MB | | 49 | 5713.55
8 | -> Nested Loop (9,18) | [55.427,56.565] | 78 | 19 |
[9KB,9KB] | 1MB | | 49 | 5713.53
9 | -> Streaming(type: BROADCAST) | [53.757,54.850] | 18 |
36 | [46KB,46KB] | 2MB | | 29 | 5705.90
10 | -> Nested Loop (11,17) | [38.080,44.807] | 6 | 12 |
[6KB,6KB] | 1MB | | 29 | 5704.81
11 | -> Streaming(type: REDISTRIBUTE) | [37.643,43.903] | 10 |
10 | [45KB,45KB] | 2MB | | 4 | 5702.88
12 | -> Nested Loop (13,15) | [0.237,37.279] | 10 | 10 |
[3KB,3KB] | 1MB | | 4 | 5702.64
13 | -> Streaming(type: BROADCAST) | [0.074,37.080] | 3 |
3 | [45KB,45KB] | 2MB | | 4 | 1.75
14 | -> Seq Scan on info_type it | [0.080,0.153] | 1 | 1 |
[32KB,32KB] | 1MB | | 4 | 1.48
15 | -> Bitmap Heap Scan on movie_info_idx mi_idx | [0.113,0.138] | 10 |
345009 | [23KB,24KB] | 1MB | | 8 | 4550.86
16 | -> Bitmap Index Scan using info_type_id_movie_info_idx | [0.039,0.071] | 10 |
[345009 | [3KB,3KB] | 1MB | | 0 | 926.57
17 | -> Index Scan using title_pkey on title t | [0.310,0.761] | 6 | 1 |
[26KB,26KB] | 1MB | | 25 | 0.63
18 | -> Index Scan using movie_id_movie_companies on movie_companies mc |
[0.791,1.698] | 78 | 2 | [16KB,16KB] | 1MB | | 32 | 0.62
(18 rows)
```

----结束

- 通过EXPLAIN ADAPTCOST选项展示模型使用情况，沿用上一节的执行计划。

步骤1 设置计划展示格式，展示模型使用情况仅支持normal模式。

```
gaussdb=# SET explain_perf_mode = normal;
```

步骤2 在EXPLAIN语句中加入ADAPTCOST选项，执行查询。

```
gaussdb=# EXPLAIN (ANALYZE, ADAPTCOST)
SELECT MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_title,
       MIN(t.production_year) AS movie_year
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info_idx AS mi_idx,
     title AS t
WHERE ct.kind = 'production companies'
     AND it.info = 'bottom 10 rank'
     AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
     AND t.production_year BETWEEN 2003 AND 2010
     AND ct.id = mc.company_type_id
     AND t.id = mc.movie_id
     AND t.id = mi_idx.movie_id
     AND mc.movie_id = mi_idx.movie_id
     AND it.id = mi_idx.info_type_id;
```

- 步骤3** 观察执行计划输出，可见在算子详情中新增了cardest字段。当算子应用了反馈基数估计模型进行基数预测时，该字段会显示所加载的模型名称。如第三个算子（Nested Loop）所示，其cardest字段值为UMM_S_3988043888,17038.5。这表明该算子使用了名为此标识符的UMM模型进行预测。模型名称即为该算子特征值（哈希值）。

QUERY PLAN

```
-----
Aggregate (cost=5711.89..5711.96 rows=1 cardest=default width=113) (actual time=98.249..98.250 rows=1
loops=1)
-> Streaming (type: GATHER) (cost=5711.89..5711.96 rows=3 cardest=default width=113) (actual
time=96.016..98.148 rows=3 loops=1)
    Node/s: All datanodes
    -> Aggregate (cost=5711.83..5711.84 rows=3 cardest=default width=113) (actual
time=[49.865,49.866]..[88.874,88.874], rows=3)
        -> Nested Loop (cost=955.32..5711.82 rows=4 cardest=trust UMM_S_3988043888,17038.5
width=45) (actual time=[49.777,49.812]..[88.842,88.842], rows=4)
            Join Filter: (mc.company_type_id = ct.id), (Expression Flatten Optimized)
            Rows Removed by Join Filter: 74
            -> Streaming(type: BROADCAST) (cost=0.00..2.30 rows=3 cardest=default width=4) (actual
time=[0.070,0.075]..[0.099,0.105], rows=3)
                Spawn on: All datanodes
                -> Seq Scan on company_type ct (cost=0.00..2.02 rows=1 cardest=try default width=4)
(actual time=[0.002,0.002]..[0.050,0.052], rows=1)
                    Filter: ((kind)::text = 'production companies'::text), (Expression Flatten Optimized)
                    Rows Removed by Filter: 3
                    -> Materialize (cost=955.32..5709.24 rows=84 cardest=default width=49) (actual
time=[49.590,49.594]..[88.547,88.555], rows=81)
                        -> Nested Loop (cost=955.32..5709.17 rows=84 cardest=trust
UMM_S_229582469,17038.5(mixed) width=49) (actual time=[49.097,49.510]..[87.876,88.476], rows=78)
                            Join Filter: (t.id = mc.movie_id), (Expression Flatten Optimized)
                            -> Streaming(type: BROADCAST) (cost=955.32..5705.36 rows=18 cardest=default
width=29) (actual time=[48.944,48.971]..[87.581,87.619], rows=18)
                                Spawn on: All datanodes
                                -> Nested Loop (cost=955.32..5704.81 rows=6 cardest=trust
UMM_S_550930755,17038.5(mixed) width=29) (actual time=[0.227,30.510]..[77.161,77.197], rows=6)
                                    -> Streaming(type: REDISTRIBUTE) (cost=955.32..5702.88 rows=10
cardest=default width=4) (actual time=[0.073,30.357]..[76.971,76.982], rows=10)
                                        Spawn on: All datanodes
                                        -> Nested Loop (cost=955.32..5702.64 rows=10 cardest=trust
UMM_F_1223820966(mixed) width=4) (actual time=[0.188,0.985]..[34.022,73.538], row
s=10)
                                            -> Streaming(type: BROADCAST) (cost=0.00..1.75 rows=3
cardest=default width=4) (actual time=[0.074,0.867]..[33.869,73.378], rows=3)
                                                Spawn on: All datanodes
                                                -> Seq Scan on info_type it (cost=0.00..1.48 rows=1 cardest=try
default width=4) (actual time=[0.062,0.064]..[0.077,0.077], rows
=1)
                                                    Filter: ((info)::text = 'bottom 10 rank'::text), (Expression
Flatten Optimized)
                                                        Rows Removed by Filter: 112
                                                        -> Bitmap Heap Scan on movie_info_idx mi_idx
(cost=955.32..4550.86 rows=345009 cardest=default width=8) (actual time=[0.067,0.068]..[0
.093,0.097], rows=10)
                                                            Recheck Cond: (info_type_id = it.id), (Expression Flatten
Optimized)
                                                                -> Bitmap Index Scan on info_type_id_movie_info_idx
(cost=0.00..926.57 rows=345009 cardest=default width=0) (actual time=[0.034,
0.034]..[0.047,0.047], rows=10)
                                                                    Index Cond: (info_type_id = it.id)
                                                                    -> Index Scan using title_pkey on title t (cost=0.00..0.63 rows=1 cardest=try
default width=25) (actual time=[0.107,0.107]..[0.215,0.215], rows=6)
                                                                        Index Cond: (id = mi_idx.movie_id)
                                                                        Filter: ((production_year >= 2003) AND (production_year <= 2010))
                                                                        Rows Removed by Filter: 4
                                                                        -> Index Scan using movie_id_movie_companies on movie_companies mc
(cost=0.00..0.62 rows=2 cardest=try default width=32) (actual time=[0.248,0.402]..[0.336,0.
632], rows=78)
                                                                            Index Cond: (movie_id = mi_idx.movie_id)
                                                                            Filter: (note !~ '%(as Metro-Goldwyn-Mayer Pictures)%'::text)
                                                                            Rows Removed by Filter: 15
```

```
Total runtime: 100.224 ms  
(39 rows)
```

----结束

7.12.2.4 常见问题

- 使用本特性进行计划生成，对于同一种查询前几次的计划不同，属于正常现象，收敛之后会获得更优计划。
- 使用缓存计划不会触发本特性属于正常现象，无参数优化不会查找任何反馈模型。
- 对于未通过explain (analyze)执行的sql语句，若生成计划的时间大于执行时间，则不会使用反馈基数。

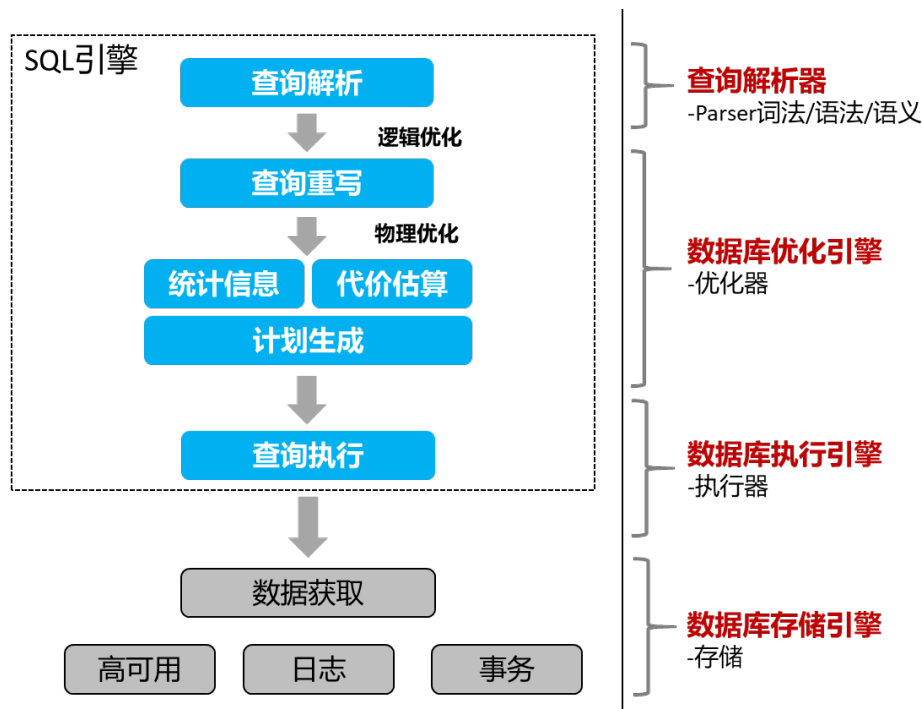
7.13 CBO 优化器

7.13.1 特性概述

优化器是SQL引擎的重要组成部分。在SQL引擎中，原始SQL语句通常会先经过解析器解析生成一棵查询树（QueryTree），然后由优化器对查询树进行逻辑优化和物理优化，最终生成执行器可以实际执行的查询计划（Plan）。其中，逻辑优化主要发生在优化器的规则模型中，因此也称为Rule-Based Optimizer（简称RBO），其主要功能是基于启发式规则对查询树进行改写，使其变成一条等价但可能执行效率更好的SQL。而物理优化则发生在优化器的代价模型中，也称为Cost-Based Optimizer（简称CBO），其主要功能是基于查询树生成不同的执行路径（Path）并根据代价估算选择一种执行方式，如：对于同一张表的访问方式是全表扫描还是索引扫描，不同表的连接方式是Nestloop Join、Hash Join还是Merge Join等。

一条SQL通常能够生成很多可以选择的路径，CBO需要依赖统计信息和代价模型计算每条可选路径的代价，并从中选择代价最小的路径作为最终的执行路径。CBO在SQL引擎中的位置如[图7-31](#)所示：

图 7-31 SQL 引擎架构



本特性重点关注CBO（Cost-Based Optimizer）在SQL引擎中的逻辑实现原理，其主要组成包括：统计信息、扫描路径与连接路径生成、代价估算与计划选择。扫描路径和连接路径决定了SQL在执行器阶段的具体执行方式，而统计信息和代价估算则决定了如何在生成的多条路径中选出代价更好的执行方式。

7.13.2 技术原理

7.13.2.1 统计信息

优化器会为SQL语句产生所有可能的访问路径（执行计划），然后从中选择一条COST值最低的执行路径。为了让优化器能够精确计算每一条执行计划的COST值，需要获取SQL语句执行时访问数据的描述信息和特征，即optimizer statistics（优化器统计信息）。

优化器统计信息包括：基表行数和页面数、NDV（number of distinct values）、MCV（most common value）、直方图（histogram）、空值比例（nullfrac）、多列统计信息等。

7.13.2.1.1 单列统计信息

行数和页面数

行数和页面数描述整表的统计信息，reltuples表示该表包含多少行元组，relpages表示该表包含多少个页面。普通表从pg_class中查询，分区表的分区从pg_partition中查询。行数和页面数用于估算查询结果的大小，通常会结合其他统计信息（如NDV、MCV、直方图等）一起使用，以便更全面地了解数据的分布和存储情况。

行数和页面数查询示例：

```
-- 统计信息章节计划显示均使用normal模式。
gaussdb=# SET explain_perf_mode = normal;
SET
gaussdb=# CREATE TABLE t1(a int, b int);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 1000), generate_series(1, 1000));
INSERT 0 1000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SELECT reltuples,relpages FROM pg_class where relname = 't1';
 reltuples | relpages
-----+-----
      1000 |         5
(1 row)

gaussdb=# CREATE TABLE test_range_pt (a int, b int, c int)
PARTITION BY range(a)
(
PARTITION p1 VALUES LESS THAN(100),
PARTITION p2 VALUES LESS THAN (200),
PARTITION p3 VALUES LESS THAN (300),
PARTITION p4 VALUES LESS THAN (maxvalue)
)ENABLE ROW MOVEMENT;
CREATE TABLE
gaussdb=# INSERT INTO test_range_pt VALUES(generate_series(1, 1000), generate_series(1, 1000));
INSERT 0 1000
gaussdb=# ANALYZE test_range_pt;
ANALYZE
gaussdb=# SELECT reltuples,relpages FROM pg_partition WHERE relname = 'p1';
 reltuples | relpages
-----+-----
        86 |         1
(1 row)
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE test_range_pt;
DROP TABLE
```

📖 说明

对于系统函数，优化器仅基于pg_proc.prorows/procost进行静态成本估算，不维护普通表统计信息，不保证估算行数与运行时实际返回行数一致。用户在查询系统函数时，应避免将高代价系统函数子查询直接参与复杂Join，尤其应避免与LIKE、正则、函数表达式等非等值条件组合使用。建议先将系统函数结果物化为临时表或CTE，再进行过滤和关联，以避免Nested Loop重复执行导致查询耗时过长。

NDV (number of distinct values)

NDV是指在数据库表的某一列中不同值的个数。例如，在员工表中，如果“性别”列只有“男”和“女”两个不同的值，那么该列的NDV为2。

对于数据库优化器来说，准确的NDV信息对执行计划选择至关重要。在表连接操作中，如果连接条件列的NDV很低（即重复值较多），优化器可能倾向于选择哈希连接等算法，这是因为对于大量重复值的列，哈希连接在构建哈希表时往往更高效。

NDV查询示例：

```
gaussdb=# CREATE TABLE Employee(name text, gender text);
CREATE TABLE
gaussdb=# INSERT INTO Employee VALUES('小明','男'),('小张','男'),('小华','女'),('小李','女');
INSERT 0 4
gaussdb=# ANALYZE Employee;
ANALYZE
gaussdb=# SELECT tablename,attname,n_distinct FROM pg_stats WHERE tablename='employee';
tablename | attname | n_distinct
-----+-----+-----
```



```
employee | name |      -1
employee | gender |     -5
(2 rows)
gaussdb=# INSERT INTO Employee SELECT * FROM Employee;
INSERT 0 4
gaussdb=# INSERT INTO Employee SELECT * FROM Employee;
INSERT 0 8
gaussdb=# INSERT INTO Employee SELECT * FROM Employee;
INSERT 0 16
gaussdb=# ANALYZE Employee;
ANALYZE
gaussdb=# SELECT tablename,attname,n_distinct FROM pg_stats WHERE tablename='employee';
tablename | attname | n_distinct
-----+-----+-----
employee | name |      -125
employee | gender |         2
(2 rows)
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# EXPLAIN SELECT * FROM Employee t1, Employee t2 WHERE t1.gender = t2.gender;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=8.41..32.22 rows=512 width=22)
  Node/s: All datanodes
    -> Hash Join (cost=4.41..8.22 rows=512 width=22)
        Hash Cond: (t1.gender = t2.gender)
        -> Seq Scan on employee t1 (cost=0.00..1.18 rows=32 width=11)
        -> Hash (cost=4.01..4.01 rows=64 width=11)
            -> Streaming(type: BROADCAST) (cost=0.00..4.01 rows=64 width=11)
                Spawn on: All datanodes
                -> Seq Scan on employee t2 (cost=0.00..1.18 rows=32 width=11)
(9 rows)

gaussdb=# DROP TABLE Employee;
DROP TABLE
```

n_distinct取值有以下三种情况：

- = 0：表示未知或者未计算。
- > 0：表示去重后唯一值的个数。
- < 0：表示其绝对值是去重后的唯一值个数占总个数的比例。

上述例子中，性别gender列的NDV值为2，代表该列只有两个值，在查询时，join行数估算结果 = 32（外表总行数）* 32（内表总行数）/ 2（内外表NDV最大值）= 512。

MCV（most common value）

MCV是指在数据库表的某一列中出现频率最高的一组值。例如，在一个销售记录表中，“商品类别”列可能会有“电子产品”、“服装”、“食品”等类别，其中“电子产品”、“服装”出现的次数最多，那么“电子产品”和“服装”就是这个列的MCV。

在查询优化过程中，MCV信息可以帮助优化器更好地估算查询结果的行数。当查询条件涉及的列有MCV信息，优化器可以根据筛选值在MCV中的频率来近似计算满足条件的行数，从而更好地判断是否应该使用索引或者全表扫描。当存在数据倾斜时，优化器会结合MCV判断是否采用倾斜优化。

MCV需满足数据平均出现频次的1.25倍。

MCV查询示例：

```
gaussdb=# CREATE TABLE Records (id int, Category text);
CREATE TABLE
gaussdb=# INSERT INTO Records VALUES(generate_series(1, 10), '电子产品');
```

```
INSERT 0 10
gaussdb=# INSERT INTO Records VALUES(generate_series(11, 20), '服装');
INSERT 0 10
gaussdb=# INSERT INTO Records VALUES(21, '食品'), (22, '其它');
INSERT 0 2
gaussdb=# ANALYZE Records;
ANALYZE
gaussdb=# SELECT tablename,attname,most_common_vals,most_common_freqs FROM pg_stats WHERE
tablename='records';
tablename | attname | most_common_vals | most_common_freqs
-----+-----+-----+-----
records | id | |
records | category | {服装,电子产品} | {0.454545,0.454545}
(2 rows)

gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# EXPLAIN SELECT * FROM Records WHERE Category = '服装';
QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.31..1.61 rows=10 width=13)
Node/s: All datanodes
-> Seq Scan on records (cost=0.00..1.14 rows=10 width=13)
Filter: (category = '服装'::text)
(4 rows)

gaussdb=# DROP TABLE Records;
DROP TABLE
```

其中most_common_vals表示MCV的值，most_common_freqs表示MCV值占总数据量的比例。

上述例子中，MCV值中“服装”的选择率为0.454545，在查询时，行数估算结果 = 22（总行数）* 0.454545（MCV值选择率）= 10。

直方图（histogram）

直方图是一种用于描述数据分布情况的工具。在数据库中，它记录了表中某一列值的分布情况，包括每个值或者值区间内数据的出现频率。例如，对于一个学生的“成绩”列，直方图可能会将成绩划分为0-59、60-69、70-79、80-89、90-100这几个区间，每个区间记录有多少学生的成绩落在这个区间内。

直方图是数据库优化器估计查询结果行数的重要依据之一。当查询条件涉及范围查询、模糊查询等情况时，优化器会参考直方图来更准确地估算行数。例如，对于查询“SELECT * FROM students WHERE score BETWEEN 90 AND 100;”，优化器会查看直方图中90-100区间的行数来估计查询结果的大小，从而选择合适的执行计划。

直方图还可以帮助优化器在面对数据分布不均匀的列时，避免选择不合适的索引或者表连接方式。例如，如果一个列的值大部分集中在某个区间，而其他区间很少有值，优化器可以根据直方图信息选择更适合该数据分布的访问方法。

GaussDB采用等频直方图，直方图查询示例：

```
gaussdb=# CREATE TABLE students (nameid int, score int);
CREATE TABLE
gaussdb=# INSERT INTO students VALUES(generate_series(1, 10), generate_series(50, 54));
INSERT 0 10
gaussdb=# INSERT INTO students VALUES(generate_series(11, 50),generate_series(60, 79));
INSERT 0 40
gaussdb=# INSERT INTO students VALUES(generate_series(51, 90),generate_series(80, 89));
INSERT 0 40
gaussdb=# INSERT INTO students VALUES(generate_series(91, 100),generate_series(91, 100));
INSERT 0 10
gaussdb=# ANALYZE students;
ANALYZE
```

[illegible]

```
Expanded display is off.
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# EXPLAIN SELECT * FROM students WHERE score > 90 AND score < 100;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=1.67..3.22 rows=10 width=8)
Node/s: All datanodes
-> Seq Scan on students (cost=1.36..2.75 rows=10 width=8)
Filter: ((score > 90) AND (score < 100))
(4 rows)

gaussdb=# EXPLAIN SELECT * FROM students WHERE score > 97 AND score < 99;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=1.52..2.84 rows=2 width=8)
Node/s: All datanodes
-> Seq Scan on students (cost=1.46..2.75 rows=2 width=8)
Filter: ((score > 97) AND (score < 99))
(4 rows)

gaussdb=# DROP TABLE students;
DROP TABLE
```

histogram_bounds保存直方图各桶的边界值，并按从小到大的顺序排列生成values数组。其中，values[0]到 values[1]确定第一个桶的范围，以此类推。各区间的频率大致相等，这种划分方式能够更准确地反映数据分布，并有效处理极端值和异常值。

直方图总的选择率并不是1，上述例子中由于存在MCV值，MCV的选择率之和为0.9，所以直方图总的选择率为 $1 - 0.9 = 0.1$ 。在查询90-100的区间值时，行数估算结果 = $100（总行数） * 0.1（直方图总的选择率） = 10$ 。在查询97-99的区间值时，由于97-99占了2个区间，一共9个区间，行数估算结果 = $100（总行数） * 0.1 *（2/9）（查询占直方图的比例） = 2$ 。

空值比例（nullfrac）

nullfrac描述整表的null值比例，查看方式：

```
gaussdb=# CREATE TABLE t1(a int, b int);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10));
INSERT 0 10
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 10));
INSERT 0 10
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SELECT tablename, attname, null_frac FROM pg_stats WHERE tablename='t1';
tablename | attname | null_frac
-----+-----+-----
t1        | a       | 0
t1        | b       | .5
(2 rows)

gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE b is null;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.31..1.57 rows=10 width=8)
Node/s: All datanodes
-> Seq Scan on t1 (cost=0.00..1.10 rows=10 width=8)
Filter: (b IS NULL)
(4 rows)

gaussdb=# DROP TABLE t1;
DROP TABLE
```

7.13.2.1.2 多列统计信息

概念

多列统计信息是把一张表中的多个列看作一个整体，计算并获取其在整张表中的数据分布信息，统计信息种类主要包括NDV（distinct值数目）、MCV（most common value）、空值比例（nullfrac）、空值高频值（NULL_MCV）等。

作用

如果一条sql引用了多个列的约束条件，这些列之间相互独立，用单列统计信息计算时，会把这些约束条件拆分成单个子约束条件。但在实际应用中无法保证这些约束条件是相互独立的，因此可能导致估算的误差较大，于是衍生出多列统计信息来计算各个列之间的依赖度以及它们的数据分布。例如，在一个体育课的学生信息表中，课程为篮球的学生性别大多是男生，课程为舞蹈的学生性别大多是女生，当查询课程是篮球，性别是男生时，使用单列统计信息容易导致误差较大，这时就需要使用多列统计信息。

示例：

```
gaussdb=# CREATE TABLE class (studentid int, course_name text, gender text);
CREATE TABLE
gaussdb=# INSERT INTO class VALUES(generate_series(1, 500),'篮球','男');
INSERT 0 500
gaussdb=# INSERT INTO class VALUES(generate_series(501, 1000),'舞蹈','女');
INSERT 0 500
gaussdb=# ANALYZE class;
ANALYZE
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# EXPLAIN SELECT * FROM class WHERE course_name = '篮球';
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..33.69 rows=500 width=15)
  Node/s: All datanodes
  -> Seq Scan on class (cost=0.00..10.25 rows=500 width=15)
       Filter: (course_name = '篮球'::text)
(4 rows)

gaussdb=# EXPLAIN SELECT * FROM class WHERE course_name = '篮球' AND gender = '男';
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..23.22 rows=250 width=15)
  Node/s: All datanodes
  -> Seq Scan on class (cost=0.00..11.50 rows=250 width=15)
       Filter: ((course_name = '篮球'::text) AND (gender = '男'::text))
(4 rows)

--查看单列统计信息
gaussdb=# \x
Expanded display is on.
gaussdb=# SELECT * FROM pg_stats WHERE tablename = 'class';
-[ RECORD 1 ]-----
+-----+-----+
| schemaname | public |
| tablename  | class  |
| attname    | studentid |
| inherited  | f      |
| null_frac  | 0      |
| avg_width  | 4      |
| n_distinct | -1     |
```

[illegible]

```
(0 rows)
--收集多列统计信息
gaussdb=# ANALYZE class ((course_name,gender));
ANALYZE
gaussdb=# EXPLAIN SELECT * FROM class WHERE course_name = '篮球' AND gender = '男';
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..34.94 rows=500 width=15)
  Node/s: All datanodes
    -> Seq Scan on class (cost=0.00..11.50 rows=500 width=15)
        Filter: ((course_name = '篮球'::text) AND (gender = '男'::text))
(4 rows)

--查看多列统计信息
gaussdb=# \x
Expanded display is on.
gaussdb=# SELECT * FROM pg_ext_stats WHERE tablename = 'class';
-[ RECORD 1 ]-----+-----
schemaname          | public
tablename           | class
attname             | 2 3
inherited           | f
null_frac           | 0
avg_width           | 11
n_distinct          | 2
n_dndistinct        | 2
most_common_vals    | {"{篮球,舞蹈}", "{男,女}"}
most_common_freqs   | {.5, .5}
most_common_vals_null |
most_common_freqs_null |
histogram_bounds    |
partitionname       |
subpartitionname    |

gaussdb=# DROP TABLE class;
DROP TABLE
```

上述例子中，从插入的数据可以看出，知道课程名字course_name就可以确定性别gender，如果没有多列统计信息，单独查询课程名字等于篮球时（course_name = '篮球'），预估行数为1000（总行数）* 0.5（课程为篮球的MCV比例）= 500，符合真实数据。但是同时查询课程名字course_name和性别gender时（course_name = '篮球' and gender = '男'），预估行数为1000（总行数）* 0.5（课程为篮球的MCV比例）* 0.5（性别为男的MCV比例）= 250，实际行数为500，不符合真实数据。

当收集完多列统计信息后，可以看到同时查询课程名字course_name和性别gender时（course_name = '篮球' and gender = '男'），预估行数为1000（总行数）* 0.5（课程为篮球和性别为男的MCV比例）= 500，实际行数为500，符合真实数据。

计算方式

通过下述示例数据，说明统计信息的具体计算过程。

示例：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES (1, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 2, 3, 4);
INSERT 0 1
```

```

gaussdb=# INSERT INTO t1 VALUES (1, 2, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 2, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 3, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 3, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 3, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (1, 3, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, 1, 1, 1);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, NULL, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, NULL, 3, 4);
INSERT 0 1
gaussdb=# INSERT INTO t1 VALUES (NULL, NULL, 3, 4);
INSERT 0 1
gaussdb=# SELECT * FROM t1;
 a | b | c | d
---+---+---+---
 1 | 1 | 1 | 1
 1 | 1 | 1 | 1
 1 | 1 | 1 | 1
 1 | 1 | 1 | 1
 1 | 2 | 3 | 4
 1 | 2 | 3 | 4
 1 | 2 | 3 | 4
 1 | 3 | 3 | 4
 1 | 3 | 3 | 4
 1 | 3 | 3 | 4
   | 1 | 1 | 1
   | 1 | 1 | 1
   | 1 | 1 | 1
   | 1 | 1 | 1
   | 3 | 4
   | 3 | 4
   | 3 | 4
(17 rows)
gaussdb=# ANALYZE t1((a,b));
ANALYZE
gaussdb=# \x
Expanded display is on.
gaussdb=# SELECT * FROM pg_ext_stats WHERE tablename = 't1';
-[ RECORD 1 ]-----+-----
schemaname          | public
tablename           | t1
attname             | 1 2
inherited            | f
null_frac            | .176471
avg_width            | 8
n_distinct           | -.235294
n_dndistinct        | -.333333
most_common_vals     | {"{1,1,1}", "{1,2,3}"}
most_common_freqs    | {.235294, .176471, .176471}
most_common_vals_null | {"{NULL}", "{1}"}
most_common_freqs_null | {.235294}
histogram_bounds     |
partitionname        |
subpartitionname     |

gaussdb=# DROP TABLE t1;
DROP TABLE

```


- `null_frac`：NULL值比例。由于存在三行[null,null]，所以 $3/17$ （总行数）= 0.176471。
- `n_distinct`：distinct值数目。去掉空值后，存在四种取值[1,1],[1,2],[1,3],[null,1]， $4/17$ (总行数) = 0.235294。
- `n_dndistinct`：DN平均的distinct值数目。由于需要除以DN数，舍弃了一些精度，导致该值无法精确计算。
- `most_common_vals`：MCV（高频值），本列保存的多列常用数值均不为NULL。每个{}里同位置为一组，比如在该示例中，MCV依次为[1,1],[1,2],[1,3]。
- `most_common_freqs`：MCV值占总行数的比例。[1,1]有4行， $4/17 = 0.23529$ ，[1,2]有3行， $3/17 = 0.176471$ ，[1,3]有3行， $3/17 = 0.176471$ 。
- `most_common_vals_null`：包含NULL值的MCV(高频值)。本列保存的多列常用数值中至少有一个值为NULL但不全为NULL。包含NULL值且不全为NULL的值只有[null,1]。
- `most_common_freqs_null`：包含NULL值的MCV值占总行数的比例。包含NULL值且不全为NULL的值只有[null,1]，并且有4行， $4/17 = 0.23529$ 。
- `histogram_bounds`：直方图信息，多列统计信息无直方图信息。

收集方法

在GaussDB中，默认情况下，ANALYZE只收集表中单列的统计信息，不会收集多个相关列的统计信息。可以通过pg_stats视图查看单列统计信息，通过pg_ext_stats视图查看多列统计信息。收集多列统计信息的方法有以下三种：

- 手动指定列名收集。

例：ANALYZE t1((a,b));

```
gaussdb=# CREATE TABLE t1 (a int, b int);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 2));
INSERT 0 10
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 2));
INSERT 0 10
gaussdb=# ANALYZE t1;
ANALYZE
--查看单列统计信息
gaussdb=# SELECT * FROM pg_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
most_common_vals | most_common_freqs | histogram_bounds | correlation | mos
t_common_elems | most_common_elem_freqs | elem_count_histogram | partitionname |
subpartitionname
-----+-----+-----+-----+-----+-----+-----+-----+
public | t1 | a | f | 0 | 4 | -5 | -5 | {1,2,3,4,5,6,7,8,9,10} |
{.1,.1,.1,.1,.1,.1,.1,.1,.1,.1} |
.409023 |
public | t1 | b | f | 0 | 4 | 2 | -.166667 | {1,2} |
{.5,.5} |
.541353 |
(2 rows)
--查看多列统计信息
gaussdb=# SELECT * FROM pg_ext_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
most_common_vals | most_common_freqs | most_common_vals_null | most_common_freqs_null |
histogr
am_bounds | partitionname | subpartitionname
-----+-----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

```
--收集多列统计信息
gaussdb=# ANALYZE t1((a,b));
ANALYZE
--查看多列统计信息
gaussdb=# SELECT * FROM pg_ext_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+-----+
public      | t1        | 1 2     | f         | 0         | 8         | -5        | -5         |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
{1,2,3,4,5,6,7,8,9,10} | {1,2,1,2,1,2,1,2,1,2} | {1,1,1,1,1,1,1,1,1,1} |
|          |          |         |          |          |          |          |          |
(1 row)

gaussdb=# DROP TABLE t1;
DROP TABLE
```

- 通过ALTER TABLE语句新增多列。

例：ALTER TABLE t1 ADD statistics ((a,b)); 随后ANALYZE t1; 即可收集多列的统计信息。ALTER TABLE只是为这个多列占了一个槽位，并不真正收集统计信息。
删除多列统计信息：ALTER TABLE t1 DELETE statistics((a,b));

```
gaussdb=# CREATE TABLE t1 (a int, b int);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 2));
INSERT 0 10
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 2));
INSERT 0 10
gaussdb=# ANALYZE t1;
ANALYZE
--查看单列统计信息
gaussdb=# SELECT * FROM pg_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+-----+
public      | t1        | a       | f         | 0         | 4         | -5        | -5         |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
{1,1,1,1,1,1,1,1,1,1} |
|          |          |         |          |          |          |          |          |
public      | t1        | b       | f         | 0         | 4         | 2         | -1.66667   |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
{5,5} |
|          |          |         |          |          |          |          |          |
(2 rows)
--查看多列统计信息
gaussdb=# SELECT * FROM pg_ext_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
(0 rows)
--设置多列统计信息
gaussdb=# ALTER TABLE t1 ADD statistics ((a,b));
ALTER TABLE
gaussdb=# ANALYZE t1;
ANALYZE
--查看单列统计信息
gaussdb=# SELECT * FROM pg_stats WHERE tablename = 't1';
 schemaname | tablename | attname | inherited | null_frac | avg_width | n_distinct | n_dndistinct |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
|          |          |         |          |          |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

- 根据索引前缀自动创建多列统计信息。
 - a. 设置参数auto_statistic_ext_columns，取值范围[1,4]。
 - b. 为表创建索引，例如为t1表创建一个索引“CREATE INDEX idx1 ON t1(a,b,c,d);”。
 - c. 收集统计信息。

806

7.13.2.2 路径生成

807

- 扫描路径（ScanPath）

扫描路径是指基表可以被遍历的方式，主要包括：顺序扫描路径（SeqScanPath）、索引扫描路径（IndexScanPath）、位图扫描路径（BitmapScanPath）等。以一个最基本的SQL语句为例：

```
SELECT * FROM t1;
```

其生成的执行计划如下：

```
QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.12..2.27 rows=5 width=16)
  Node/s: All datanodes
    -> Seq Scan on t1 (cost=0.00..2.02 rows=5 width=16)
(3 rows)
```

在该示例中“Seq Scan on t1”是由顺序扫描路径生成的。实际上，对于不同的SQL语句，根据约束条件、数据分布情况的不同优化器可以生成不同的扫描路径，详细说明可以参考[扫描路径生成](#)章节。

- 连接路径（JoinPath）

连接路径主要记录各类基表之间物理连接的方式，主要包括：NestloopJoinPath、HashJoinPath、MergeJoinPath等物理连接路径，详细说明可以参考[连接路径生成](#)章节。对于同一个SQL而言，不同连接方式都可以实现查询逻辑，但是选择不同连接路径的执行效率是不同的。以一个基本的InnerJoin场景为例，其SQL语句如下：

```
SELECT * FROM t1 INNER JOIN t2 ON t1.b = t2.b;
```

该示例可以生成HashJoin路径，其计划如下所示：

```
QUERY PLAN
-----
Streaming (type: GATHER) (cost=187.03..576.53 rows=5000 width=32)
  Node/s: All datanodes
    -> Hash Join (cost=183.03..368.15 rows=5000 width=32)
      Hash Cond: (t1.b = t2.b)
        -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
          Spawn on: All datanodes
            -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
        -> Hash (cost=162.20..162.20 rows=5001 width=16)
          -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
            Spawn on: All datanodes
              -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(11 rows)
```

也可以生成NestloopJoin路径，其计划如下所示：

```
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..42220.27 rows=5000 width=32)
  Node/s: All datanodes
    -> Nested Loop (cost=0.00..42011.90 rows=5000 width=32)
      Join Filter: (t1.b = t2.b)
        -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
          Spawn on: All datanodes
            -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
        -> Materialize (cost=0.00..170.53 rows=5000 width=16)
          -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
            Spawn on: All datanodes
              -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(11 rows)
```

从两个计划的代价可知，对于该场景HashJoin路径生成的计划执行性能要明显好于NestloopJoin路径计划。物理优化阶段的一个重要工作就是尝试生成不同的物理连接路径并从中选出代价最低的连接路径，从而提升查询计划的执行性能。

此外，在优化器中实际进行选择路径时要考虑很多因素，比如根据磁盘类型、CPU参数等调整代价基准值，根据数据分布的统计信息、条件约束、选择率等评

估基表扫描是选择索引扫描路径还是顺序扫描路径，以及对于路径的启动代价、整体代价如何选择，因而在路径生成阶段通常需要保留多条物理路径，在计划生成时通过代价估算模型进行决策。

7.13.2.2.1 扫描路径生成

扫描路径主要包括：顺序扫描路径（SeqScanPath）、索引扫描路径（IndexScanPath）、位图扫描路径（BitmapScanPath）等几种类型，本节主要针对这几类扫描路径进行说明。

- 顺序扫描（SeqScan）

顺序扫描即全表扫描，是GaussDB中最基础的一种扫描方式。这种扫描方式直接从磁盘中按顺序读取基表相关页面，将表中的所有元组都取出来，然后再按照约束条件进行过滤，最终根据查询语句的指示返回符合过滤条件的元组，因此该扫描方式的算法复杂度是 $O(N)$ 。

以下结合具体示例来展示顺序扫描路径的原理，基表建表语句如下：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(1,2,3,4),(2,3,4,5),(3,4,5,6),(4,5,6,7),(5,6,7,8);
INSERT 0 5
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET explain_perf_mode=normal;
SET
```

执行如下SQL语句：

```
gaussdb=# SELECT * FROM t1 WHERE t1.b = 3;
 a | b | c | d
---+---+---+---
 2 | 3 | 4 | 5
(1 row)
```

对于上述SQL，遍历t1表，并将t1表中b列值等于3的行返回，因此生成的查询计划如下所示：

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE t1.b = 3;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..2.15 rows=1 width=16) (actual time=2.450..3.849 rows=1 loops=1)
  Node/s: All datanodes
    -> Seq Scan on t1 (cost=0.00..2.02 rows=1 width=16) (actual time=[0.006,0.006]..[0.179,0.179], rows=1)
      Filter: (b = 3)
      Rows Removed by Filter: 4
    Total runtime: 4.367 ms
(6 rows)
```

该计划中，“Seq Scan on t1”表明由顺序扫描路径生成，“Filter: (b = 3)”是顺序扫描的过滤条件。根据上面计划的实际执行情况可知，执行器遍历了整个t1表，过滤掉4行数据（Rows Removed by Filter: 4），返回了1行数据“（actual time=2.450..3.849 rows=1 loops=1）”，与预期一致。

上述示例删除建表语句如下：

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

- 索引扫描（IndexScan）

GaussDB中表的物理存储方式是按堆表存储的，当插入一条数据时，其在物理磁盘上可能被存放在一个新开辟的页面上，也可能被存放在之前删除元组后空出来的页面上。这样组织的好处是存储结构比较简单，但是也带来一些问题，对于一些带有排序条件或等值过滤条件的场景无法利用有序查找算法来提升执行效率，由于数据是无序存储的，需要顺序扫描遍历整表才能找到所有满足条件的元组。

为了提升数据库的查询性能，GaussDB提供了多种索引扫描方式来满足大数据量有序查询场景下的高性能诉求。一种常见的索引组织方式是B/B+树，其使用有序的树状结构对磁盘上的数据重新进行组织，添加额外的查找目录，使得对于SQL中包含“b = ?”这样过滤条件的数据查找复杂度降低到 $O(\log(N))$ 。

在GaussDB中想要生成索引路径，通常需要在过滤条件的基表列上有索引约束，否则无法生成索引路径。索引类型通常包含：主键索引、唯一索引（有NULL值）、唯一非空索引等。以下面场景为例，建表语句如下：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET explain_perf_mode=normal;
SET
```

执行如下SQL语句：

```
gaussdb=# SELECT * FROM t1 WHERE t1.b = 200;
 a | b | c | d
----+---+---+---
200 | 200 | 2 |
(1 row)
```

对于上述SQL，遍历t1表，并将t1表中b列值等于200的行返回，生成的查询计划如下所示：

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE t1.b = 200;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..2.59 rows=1 width=16) (actual time=2.307..4.253 rows=1
loops=1)
  Node/s: All datanodes
    -> Index Scan using idx_t1_b on t1 (cost=0.00..2.47 rows=1 width=16) (actual time=[0.033,0.033]..
[0.055,0.057], rows=1)
      Index Cond: (b = 200)
    Total runtime: 4.645 ms
(5 rows)
```

该计划中，“Index Scan using idx_t1_b on t1”表明由索引扫描路径生成，使用的是t1表在b列上建立的idx_t1_b索引，而“Index Cond: (b = 200)”是索引扫描条件。根据上面计划的实际执行情况可知，执行器并没有遍历整个t1表，而是直接通过索引找到了b = 200的行。对比前面的顺序扫描方式，其执行计划如下：

```
gaussdb=# EXPLAIN ANALYZE SELECT /*+tablescan(t1)*/ FROM t1 WHERE t1.b = 200;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..28.96 rows=1 width=16) (actual time=3.046..3.272 rows=1
loops=1)
  Node/s: All datanodes
    -> Seq Scan on t1 (cost=0.00..28.84 rows=1 width=16) (actual time=[0.120,0.760]..[0.964,0.964],
rows=1)
      Filter: (b = 200)
```

```
Rows Removed by Filter: 4999
Total runtime: 3.593 ms
(6 rows)
```

上面的查询计划通过`tablescan()`的HINT强制指定了顺序扫描方式，从这个计划可以看到，顺序扫描方式遍历了整个`t1`表，并将 `b = 200` 的行返回，这种扫描方式需要过滤掉4999行无效数据，因此其执行时间对比索引扫描要高很多。

索引扫描本身利用的是以空间换时间的思想，通过对堆存储的无序数据进行额外的索引排序，使其在索引树上变成有组织的数据结构，并与堆上的数据建立映射关系，从而达到可以快速查找目标数据的效果。除了普通索引路径，在一些特定场景下GaussDB还能生成一些特殊类型索引路径，例如，当SQL的过滤条件和投影中只包含索引列时，整个查询所需数据都可以在索引中获取到，就避免了再去查找一次整个元组的回表操作，从而可以生成一条`IndexOnlyScan`路径，典型的场景如下所示：

```
gaussdb=# EXPLAIN ANALYZE SELECT b FROM t1 WHERE t1.b = 200;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..1.49 rows=1 width=4) (actual time=2.171..2.370 rows=1
loops=1)
  Node/s: All datanodes
  -> Index Only Scan using idx_t1_b on t1 (cost=0.00..1.37 rows=1 width=4) (actual
time=[0.030,0.032]..[0.037,0.037], rows=1)
    Index Cond: (b = 200)
    Heap Fetches: 0
  Total runtime: 2.711 ms
(6 rows)
```

该计划中，“`Index Only Scan using idx_t1_b on t1`”表明其查询路径是由`IndexOnlyScan`生成的，实际查找数据时只需要访问索引节点即可，无需再从映射的元组中取数据。

上述示例删除建表语句如下：

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

- 位图扫描（`BitmapScan`）

位图扫描是一种特殊的索引扫描方式，它的原理是利用多个查询条件中的索引路径构建一个`Bitmap`集合，再结合`Bitmap`的交并集操作实现对数据归并处理，从而将原来只能顺序扫描的随机堆表访问转变为利用索引条件有序的堆表访问。前面介绍了`IndexScan`路径生成的条件，在此基础上，如果SQL中存在多个查询约束条件，并且这些约束条件上都能生成索引路径，那么这些索引路径就可以作为位图扫描的待选路径；进而，如果这些约束条件是通过`AND`、`OR`等谓词连接的简单表达式，那么就可以生成`BitmapScan`路径。以下面场景为例，建表语句如下：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET explain_perf_mode=normal;
SET
```



```
gaussdb=# SET enable_bitmapscan=on;
SET
```

执行如下SQL语句：

```
gaussdb=# SELECT * FROM t1 WHERE t1.a = 200 OR t1.b = 300;
 a | b | c | d
-----+-----+----
200 | 200 | 5 |
300 | 300 | 0 |
(2 rows)
```

对于上述SQL，遍历t1表，并将t1表中a列值等于200或者b列值等于300的行返回，其生成的查询计划如下所示：

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE t1.a = 200 OR t1.b = 300;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=2.78..3.96 rows=2 width=16) (actual time=2.289..2.498 rows=2
loops=1)
  Node/s: All datanodes
    -> Bitmap Heap Scan on t1 (cost=2.72..3.83 rows=2 width=16) (actual time=[0.046,0.046]..
[0.097,0.098], rows=2)
      Recheck Cond: ((a = 200) OR (b = 300))
      -> BitmapOr (cost=2.72..2.72 rows=2 width=0) (actual time=[0.034,0.034]..[0.040,0.040],
rows=0)
        -> Bitmap Index Scan on idx_t1_a (cost=0.00..1.36 rows=1 width=0) (actual
time=[0.020,0.020]..[0.027,0.027], rows=1)
          Index Cond: (a = 200)
        -> Bitmap Index Scan on idx_t1_b (cost=0.00..1.36 rows=1 width=0) (actual
time=[0.012,0.012]..[0.013,0.013], rows=1)
          Index Cond: (b = 300)
      Total runtime: 2.890 ms
(10 rows)
```

如上面的示例，位图扫描中实际上包含了4种不同的路径，其包括：BitmapHeap路径、BitmapOr路径或BitmapAnd路径以及IndexScan路径。在该计划中，“Bitmap Heap Scan on t1”表明该计划由Bitmap索引扫描路径生成，使用的是t1表在a列上建立的idx_t1_a索引和b列上建立的idx_t1_b索引。根据上面计划的实际执行情况可知，执行器先分别在两个索引路径上查找满足Index Cond约束的索引值，然后通过BitmapOR算子对索引扫描结果进行合并，最后通过Recheck Cond: ((a = 200) OR (b = 300)) 获取满足条件的结果集。对比顺序扫描方式，其执行计划如下：

```
gaussdb=# EXPLAIN ANALYZE SELECT /*+tablescan(t1)*/ FROM t1 WHERE t1.a = 200 OR t1.b = 300;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=0.06..33.13 rows=2 width=16) (actual time=3.943..4.879 rows=2
loops=1)
  Node/s: All datanodes
    -> Seq Scan on t1 (cost=0.00..33.00 rows=2 width=16) (actual time=[0.164,1.066]..[1.265,1.265],
rows=2)
      Filter: ((a = 200) OR (b = 300))
      Rows Removed by Filter: 4998
      Total runtime: 5.165 ms
(6 rows)
```

上面的查询计划通过tablescan()的HINT强制指定了顺序扫描方式，从这个计划可以看到，顺序扫描方式遍历了整个t1表，并将 a = 200或b=300的行返回，这种扫描方式需要过滤掉4998行无效数据，因此其执行时间对比BitmapOr扫描要高很多。

上述示例删除建表语句如下：

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

7.13.2.2.2 连接路径生成

前面章节介绍了几类常用的基表扫描路径，包括：顺序扫描路径、索引扫描路径和位图扫描路径，但在实际的SQL查询场景中并非只涉及基表查询，而是经常需要对多个表做各种连接操作来生成最终查询结果，SQL查询中常用的连接操作包括：INNER JOIN、LEFT JOIN、SEMI JOIN等，因此针对这些连接运算GaussDB会在CBO优化器阶段生成不同的连接路径。这些路径指的是物理连接路径（区别于INNER JOIN、LEFT JOIN这些逻辑连接运算），即可以被执行器理解的具体算法，常见连接路径实现主要有三类：NestloopJoin路径、HashJoin路径和MergeJoin路径，连接路径的生成过程也就是不断尝试这三种路径的过程。

之所以会有不同的物理连接路径，是因为在不同的数据分布、索引条件和约束场景下，两个基表要建立连接关系，使用不同的物理路径在实际执行过程中性能会有差异，没有一种物理路径可以在所有场景下保持高性能，因此通常要针对不同的情况基于代价选择适合的连接路径，从而提升执行效率。

为了便于描述三种连接路径的生成方式，建立如下数据库表：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 10000),
floor(random()*10));
INSERT 0 10000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# CREATE TABLE t2(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t2_a ON t2(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t2_b ON t2(b);
CREATE INDEX
gaussdb=# INSERT INTO t2 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 10000),
floor(random()*10));
INSERT 0 10000
gaussdb=# ANALYZE t2;
ANALYZE
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET explain_perf_mode=normal;
SET
```

- NestloopJoin路径

NestloopJoin路径又被称为嵌套循环连接路径，是三种连接方式中最通用、最直观的一种连接方式，其在生成连接路径时会选择一张表做驱动表（或称作外表），另一张表做被驱动表（或称作内表），通过驱动表和被驱动表做双层循环遍历的方式进行数据匹配，时间复杂度为 $O(N*M)$ 。以两表的Nestloop场景为例，执行如下SQL语句：

```
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.a != t2.a AND t1.b != t2.b;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..4750131.35 rows=99980001 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..584297.98 rows=99980001 width=32)
    Join Filter: ((t1.a <> t2.a) AND (t1.b <> t2.b))
-> Seq Scan on t1 (cost=0.00..48.33 rows=10000 width=16)
-> Materialize (cost=0.00..999.65 rows=30000 width=16)
```

```
-> Streaming(type: BROADCAST) (cost=0.00..949.65 rows=30000 width=16)
    Spawn on: All datanodes
    -> Seq Scan on t2 (cost=0.00..48.33 rows=10000 width=16)
(9 rows)
```

上述示例是一个典型的Nestloop路径生成计划的场景，该路径以t1表做驱动表，t2表做被驱动表进行双层循环遍历，外表t1表每扫描一行数据，即驱动内表t2表执行一次遍历，并将满足t1.a != t2.a AND t1.b != t2.b匹配条件的内外表行做连接操作。根据NestloopJoin路径的生成原理，这种连接方式除了FULL JOIN场景下无法使用，在类型Join场景下均可生成候选路径，因此可以作为其他连接路径无法生成时的最后选择，但是根据算法复杂度可以看出，在数据规模较大的场景下特别是约束条件过滤性较好时其性能不一定是最好的。

NestloopJoin路径中还有一种特殊的路径形态，叫做参数化路径。假设t1表和t2表做Join的场景，连接条件为t1.a = t2.a，根据谓词下推的原理，这个JOIN条件无法下推到内外表的任何一侧，因为t1.a = t2.a这个连接条件既关联了外表，又关联了内表，必须同时获取两个表的元组后才能应用这个约束条件，这种情况下即使在t2表的a列上存在索引也无法生成索引路径加速数据筛选。为了利用索引条件加速，可以考虑将外表条件列参数化，生成一个特殊的Param类型，在外表驱动内表过程中，当t1.a的值确定后就可以作为一个常数传递到t1.a = t2.a这个约束上，约束条件即可转换为t2.a = Const，对于常量谓词可以正常下推到内表上，从而在t2表上生成索引路径，利用索引条件的特性进行加速，典型的场景如下面SQL所示：

```
gaussdb=# SET enable_hashjoin=off;
SET
gaussdb=# SET enable_mergejoin=off;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..1417.36 rows=10000 width=32)
  Node/s: All datanodes
  -> Nested Loop Left Join (cost=0.00..1000.74 rows=10000 width=32)
    -> Seq Scan on t1 (cost=0.00..48.33 rows=10000 width=16)
    -> Index Scan using idx_t2_a on t2 (cost=0.00..0.28 rows=1 width=16)
        Index Cond: (t1.a = a)
(6 rows)
```

这里为了展示参数化路径，关闭了HashJoin和MergeJoin，从上面的计划可以看出，t1.a = t2.a这个条件被下推到了t2表的索引条件上，当外表每执行一行，即可将该行的a列值作为常数传递到内表，内表即可将常量查询条件作用在t2表的索引上，达到快速检索数据的目的。

- HashJoin路径

HashJoin路径是一种利用hash表加速连接操作的Join方式，要求连接条件指定列的数据类型可以做hash运算（如数值类型），且连接条件必须为等值条件。以两表HashJoin场景为例，执行如下SQL语句：

```
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=93.99..600.78 rows=10000 width=32)
  Node/s: All datanodes
  -> Hash Left Join (cost=89.99..184.15 rows=10000 width=32)
      Hash Cond: (t1.a = t2.a)
      -> Seq Scan on t1 (cost=0.00..48.33 rows=10000 width=16)
      -> Hash (cost=48.33..48.33 rows=9999 width=16)
          -> Seq Scan on t2 (cost=0.00..48.33 rows=10000 width=16)
(7 rows)
```

上面示例是一个典型的HashJoin路径生成计划场景，在该场景中t1表的a列和t2表的a列都是整型值，且满足等值连接约束，因此可以生成HashJoin路径。其中，t1

表作为驱动表，t2表作为被驱动表，在实际执行时，首先会对内表中的数据按指定列（示例场景中为t2表的a列）建hash表，然后使用外表驱动内表，对于外表的每一条数据，利用hash表查找匹配条目。

根据Hash表的原理可知通常情况下，HashJoin路径的效率要高于NestloopJoin路径，对比NestloopJoin计划如下所示：

```
gaussdb=# SET enable_hashjoin=off;
SET
gaussdb=# SET enable_mergejoin=off;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
QUERY PLAN

-----
Streaming (type: GATHER) (cost=4.00..1417.36 rows=10000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..1000.74 rows=10000 width=32)
-> Seq Scan on t1 (cost=0.00..48.33 rows=10000 width=16)
-> Index Scan using idx_t2_a on t2 (cost=0.00..0.28 rows=1 width=16)
    Index Cond: (t1.a = a)
(6 rows)
```

为了保证生成NestloopJoin路径计划，本例中关闭了生成HashJoin和MergeJoin路径的逻辑，对比两个计划可知NestloopJoin路径生成的计划代价明显高于HashJoin路径生成的计划。

- MergeJoin路径

MergeJoin路径是一种利用预排序加速连接操作的join方式，根据连接条件，将基表按连接条件指定列先进行排序，之后像归并排序那样，找出满足连接条件的条目（一般为等值条件）并输出。以两表MergeJoin场景为例，执行如下SQL语句：

```
gaussdb=# SET enable_mergejoin=on;
SET
gaussdb=# SET enable_hashjoin=off;
SET
gaussdb=# EXPLAIN SELECT t1.a,t2.a FROM t1 LEFT JOIN t2 ON t1.a = t2.a;
QUERY PLAN

-----
Streaming (type: GATHER) (cost=4.00..589.11 rows=10000 width=8)
Node/s: All datanodes
-> Merge Left Join (cost=0.00..172.49 rows=10000 width=8)
    Merge Cond: (t1.a = t2.a)
-> Index Only Scan using idx_t1_a on t1 (cost=0.00..61.24 rows=10000 width=4)
-> Index Only Scan using idx_t2_a on t2 (cost=0.00..61.24 rows=10000 width=4)
(6 rows)
```

MergeJoin路径的性能通常介于HashJoin路径和NestloopJoin路径之间，但有些场景下也可以带来明显优势。根据上面的计划可知，对于MergeJoin路径主要利用了归并排序的原理，因为这个SQL中只输出t1表和t2表的a列，a列是分布列且两张表在a列上均有索引，利用索引的有序性可以提升归并排序算法的性能，因此可以带来比HashJoin更好的执行效率。

上述示例删除建表语句如下：

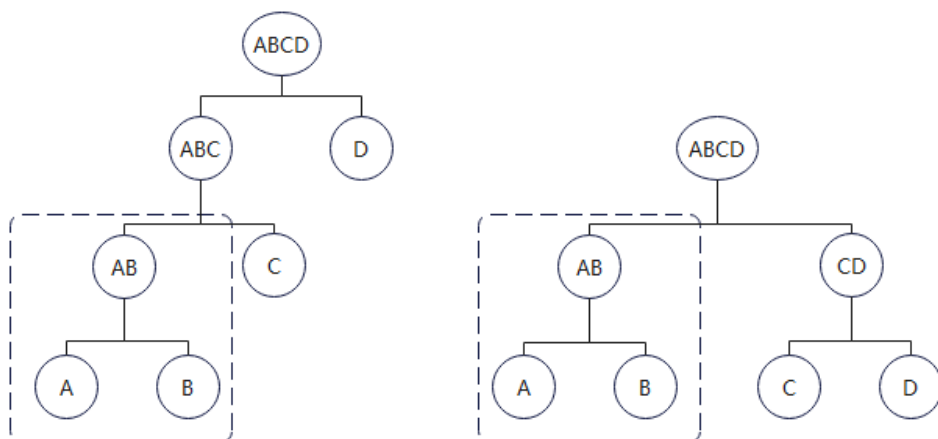
```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE t2;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

上面介绍了GaussDB中常用三类连接路径，但在实际场景中除了要考虑连接算法外还需要考虑表之间的连接顺序。以INNER JOIN为例，对于t1 INNER JOIN t2这样一组JOIN关系，在生成Nestloop Join路径时就有两种选择，可以让t1表做驱动表，也可以让t2表做驱动表，因为NestloopJoin适合小表驱动大表的场景，这时就需要根据t1表和t2表的数据分布情况进行选择，所以在生成连接路径时通常需要考虑表之间不同的连

接顺序组合。由于SQL的复杂性，路径生成过程一般无法通过穷举实现，以最简单的两表JOIN场景为例，就需要至少尝试18种不同的路径（3种连接算法x2种连接顺序x3种扫描路径），实际情况更加复杂，还需要考虑是否有参数化路径、约束条件是否可以下推等，因此对于多表Join的情况，路径规模呈指数级增长。因此数据库中通常会采用一些智能算法来进行剪枝，避免搜索空间爆炸，常用的路径生成算法主要有：动态规划算法和遗传算法。以最广泛使用的动态规划算法为例，其核心思想是：将大问题分解为小问题，解决小问题，存储小问题的最优解，并用这些最优解来构造更大问题的最优解，避免重复计算。算法核心思想说明如下：

- 提取重复子问题

图 7-32 提取子问题示例图



如上图两颗连接树中的AXB的连接操作就属于重复子问题。对于Join Tree的每个子问题，可以通过不断获得子问题的最低代价路径，剪枝掉较差的代价路径，而逐层迭代式地“堆积”出完整连接树的最低代价路径。

- 获取最优子结构

对于路径生成来说，每个叶子节点都是表的扫描路径，但是当前扫描路径最优并不一定代表后续的连接路径最优。例如，顺序扫描代价 < 索引扫描代价，顺序扫描代价 + 排序代价 > 索引扫描代价，所以在优化器计算子问题的代价时，不仅要保存最优解，还同时保存较优解，它需要考虑最优的启动代价路径、参数化路径等。动态规划从求解的方式而言通常有递归和迭代两种方法，目前数据库采用的是迭代的方法进行求解，它先尝试求两个表的最优子路径，然后依次迭代成3个表、4个表。

7.13.2.3 计划生成

前章节介绍了各类扫描路径与连接路径的生成，CBO优化器中会根据代价对这些路径进行筛选并保留最优路径。筛选主要分为两步：第一步是根据初级代价决定是否要创建连接路径，连接路径根据算法的不同、连接顺序的不同以及底层扫描方式的不同可以生成很多条，所以初步筛选的目的是过滤掉一些明显较差的路径，从而节省生成执行计划的时间，但是因为扫描路径种类有限（基本类型只有顺序扫描、索引扫描、位图扫描三种），通常不做筛选，尽量保留可以给连接路径提供更多的选择机会。第二步，在完成初步筛选后对路径节点补充新的代价，如表达式处理的代价、LIMIT约束代价等，这样生成最终可以比较的计划，再根据启动代价、整体代价等影响因素比较决策选取最优路径。选出最优路径后，为了方便执行器的执行，会进一步将其转换为适合执行器的PlanTree，从而执行器就可以执行一个完整的计划。

7.13.2.3.1 代价估算

对于优化器来说，为了能够选出最优的执行计划，需要有可以衡量执行计划的标准。一个直观的比较不同查询计划优劣的标准即执行时间，一般而言，执行时间越短的查询计划越优，反之越差。然而同一个查询计划的执行时间会随着环境变量的不同而变化，如：机器配置（CPU 的频率、内存的大小和磁盘的介质等）会对实际执行性能产生影响，那么就需要一个无量纲的值来代替时间，在GaussDB中使用代价（Cost）这一概念来判断不同执行计划的优劣。代价是一个用于评判执行计划优劣的绝对指标，是一个估算值，而每个算子的代价都有其对应的计算公式，该公式被称为代价模型。通过将统计信息作为输入，代入到代价模型中，即可计算出相应算子的代价，可抽象为如下公式：

$$Cost = F_{operator}(X_{statistics})$$

如上述公式所示，每个算子的代价模型是对其执行逻辑的抽象，对于遇到的每一个可被作为基本计算单位量的操作（如访问页面、表达式运算等），都需要将其以代价为单位进行去量纲化的表示。以顺序扫描为例，其执行逻辑为：从文件中顺序读取页面，在没有谓词的情况下，顺序扫描的代价即扫描所有页面的代价，这里需要对顺序访问单个页面操作的代价进行定义，并作为计算顺序扫描多个页面的单位量。

为定义不同影响因素的评估基准，把一些影响执行效率的关键因素提取出来，抽象成标准值，所有路径的代价都基于标准值去计算。例如，将CPU频率参数、磁盘页面I/O方式、内存I/O方式进行标准化作为代价估算的基准，下面对磁盘I/O基准代价与CPU基准代价进行说明。

- 磁盘页面I/O的基准代价

磁盘页面的I/O方式通常包含：顺序读写和随机读写。以传统的机械硬盘为例，因为机械硬盘在读取数据的时候需要进行寻道，寻道要花费一定时间，因而随机读写就会有大量时间浪费到寻道上，顺序读写则能够节省这些寻道时间，带来I/O性能提升。另外磁盘中通常有缓存，顺序读写时磁盘可以预取数据，提升缓存命中率，相比随机读写也能带来性能的提升。

基于上述读写方式的不同，需要对顺序读写和随机读写定义不同的代价基准，在GaussDB的早期版本中，定义顺序读写的默认值是1.0，随机读写的默认值是4.0。但是随着固态硬盘的普及，固态硬盘随机读写的性能已经大大提高，该基准值已经不再准确，因此在最新的版本中，已经将随机读写的默认值调整为1.1。在不同的生产环境中这两个基准值也并非一成不变，因此GaussDB支持GUC参数根据具体环境进行修改。调整顺序读写页面基准值的GUC参数是seq_page_cost，调整随机读写页面基准值的GUC参数是random_page_cost。

此外，由于数据库本身有缓存系统，如果数据已经在缓存中就可以避免磁盘I/O的开销，这样在进行代价评估时不能简单对每个页面都计算磁盘I/O代价。所以，在进行代价评估时还需要对缓存页面数做一个预估，该值由GUC参数effective_cache_size控制。

- 元组与约束条件的CPU基准代价

数据从磁盘页面中被读取出来后，还需要以元组的形式呈现出来，因此需要依赖CPU的计算资源，这时会产生CPU的计算代价。由于元组的组织结构差异、表达式计算复杂度差异等原因，其消耗的计算资源也是不同的，因此，在GaussDB中主要区分了普通元组、索引元组以及对投影、表达式等不同数据组织形式的CPU计算代价，并且支持通过GUC参数调整这些基准代价：

控制普通元组CPU基准的GUC参数：cpu_tuple_cost，默认值为0.01。
控制索引元组CPU基准的GUC参数：cpu_index_tuple_cost，默认值为0.005。
控制表达式CPU基准的GUC参数：cpu_operator_cost，默认值为0.0025。

除了上述基准代价，GaussDB在进行代价比较时通常还要关注启动代价和总体代价。启动代价指的是SQL语句从开始执行到执行器查询获取到第一条元组的代价，而总代价则是指SQL语句从开始执行到结束的所有代价。因此，有如下公式：

总代价 (total_cost) = 启动代价(startup_cost) + 运行代价(run_cost)

对于不同的扫描路径和连接路径，其启动代价通常不同，因此需要单独进行评估，以下面的场景为例，建表语句如下：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SET explain_perf_mode=normal;
SET
```

执行相同SQL语句，生成不同的路径计划：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE t1.a >= 100 ORDER BY a;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..252.12 rows=4900 width=16)
  Merge Sort Key: a
  Node/s: All datanodes
    -> Index Scan using idx_t1_a on t1 (cost=0.00..47.99 rows=4900 width=16)
        Index Cond: (a >= 100)
(5 rows)
gaussdb=# EXPLAIN SELECT /*+tablescan(t1)*/ FROM t1 WHERE t1.a >= 100 ORDER BY a;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=119.99..324.19 rows=4899 width=16)
  Merge Sort Key: a
  Node/s: All datanodes
    -> Sort (cost=115.99..120.07 rows=4899 width=16)
        Sort Key: a
        -> Seq Scan on t1 (cost=0.57..28.84 rows=4900 width=16)
            Filter: (a >= 100)
(7 rows)
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

上述示例中，可以看到同一条SQL语句在生成不同路径计划时启动代价的差异。第一次执行时选择了索引路径，因为这条语句需要排序，索引路径底层使用B+树能够快速定位到目标数据并且可以保证有序性，其启动代价为0。第二次执行选择的是顺序扫描路径，需要遍历磁盘找到第一条有效数据，启动代价为0.57，再叠加排序操作导致其整体代价高于索引扫描路径。对于本场景中的示例在默认情况下，数据库会根据计划选择策略淘汰掉顺序扫描+排序操作的计划，这里为了便于对比使用tablescan的HINT强制选择了顺序扫描计划。

结合上述示例，顺序扫描路径的代价计算公式为：

startup_cost (启动代价) = (cpu_tuple_cost+cpu_expr_cost) *M<tuple>+seq_page_cost*M<page>
run_cost (运行代价) =cpu_run_cost + disk_run_cost=(cpu_tuple_cost+cpu_expr_cost) *N<tuple>
+seq_page_cost*N<page>

其中 seq_page_cost、cpu_tuple_cost 的值来自前面基准代价的参数配置，本例中分别为：1.0和0.01，N<tuple> 和 N<page> 分别表示表中的元组总数与页面总数，可以使用以下SQL查询获取：

```
gaussdb=# SELECT reltuples, relpages FROM pg_class WHERE relname='t1';
reltuples | relpages
-----+-----
      5000 |      24
(1 row)
```

cpu_expr_cost的值来自WHERE条件中的所有表达式计算代价和，本例中的表达式只包含t1.a >= 100 条件，t1表的a列是INT类型，表达式计算函数为GE（大于等于），因此可以通过如下SQL获取到其计算基准代价系数：

```
gaussdb=# SELECT proname, procost FROM pg_proc WHERE proname = 'int4ge';
proname | procost
-----+-----
int4ge  |      1
(1 row)
```

表达式类型的基准代价由GUC参数cpu_operator_cost控制，本例中的默认值是0.0025。将这些信息带入公式计算得到如下结果：

```
run_cost (运行代价) = (cpu_tuple_cost+cpu_expr_cost) *N<tuple>+seq_page_cost*N<page>
                    = (0.01+1x0.0025)*5000+1.0*24
                    = 86.5
```

但是上面的示例中，实际代价显示的却是28.84, 这是因为用例场景的数据库有三个DN节点，可以使用如下SQL查询CN和DN的节点信息：

```
gaussdb=# SELECT node_name, node_type, hostis_primary FROM PGXC_NODE;
node_name | node_type | hostis_primary
-----+-----+-----
coordinator1 | C      | t
datanode1   | D      | t
datanode2   | D      | t
datanode3   | D      | t
(4 rows)
```

如上查询结果所示，t1表的数据均匀分布在三个数据节点上，因此对于每个DN节点的代价需要取平均值，即：86.5/3=28.84。

启动代价的计算方式如下，因为这里选择的是顺序扫描的路径，即遍历整个数据表，根据启动代价的定义并且结合数据均匀分布假设，可以认为这5000条数据均匀的分布在24个页面上，获取到第一条有用数据的代价为：

```
startup_cost (启动代价) = (cpu_tuple_cost+cpu_expr_cost) *M<tuple>+seq_page_cost*M<page>
                        = (0.01+1*0.0025) *100+1.0*(24/5000*100)
                        = 1.25+0.48
                        = 1.73
```

同理，因为数据均匀分布在3个DN节点上，因此对于一个节点而言，其启动代价为1.73/3=0.57。

因为顺序扫描本身就要扫描整个表，run_cost的部分计算实际已经包含了启动代价的值，所以可以看到总代价和执行代价是一致的。

7.13.2.3.2 计划选择

前面的章节介绍了数据库中几类常用的路径生成方式以及优化器的代价估算模型原理，在此基础上CBO优化器可以选择生成最终的执行计划。在路径生成阶段，优化器利用动态规划、遗传算法等尝试不同的连接路径，并且为这些路径标记了代价信息，优化器只需要根据代价进行比较把更小代价的路径选择出来，这样就得到了一棵最优路径树。

路径选择过程中，主要关注如下几个因素：

- 启动代价
当包含LIMIT约束条件时，路径启动代价是一个非常重要的考虑因素，因为LIMIT条件可以提前终结数据遍历。
- 总代价

路径的总代价是最重要的衡量指标。

- 路径的过滤性

如果一条路径能够提前过滤掉大部分数据，则认为这条路径具有更好的过滤性，其返回给上层更少的行数，可以减少计算量。

- 是否有HINT

HINT是数据库中的一种重要调优手段，可以在代价估算不准确时作为一种补偿措施，因此对于有特定HINT应用的路径其被选择的比重也会提升。本章节暂不讨论此种情况。

基于上述因素，路径选择的总体原则如下（以A路径为已选路径，B路径为新路径做比较说明）：

1. 如果A的总体代价优于B，那么进一步看B的启动代价是否优于A，如果B的启动代价优于A，则保留A路径同时接受新路径B作为候选路径。这里保留B路径是因为最终生成上层计划时如果有LIMIT等一些约束条件，B路径可能带来更好的性能，在整个路径选择过程中，优化器中并非只会保留一条可选路径。如果B路径的启动代价也没有优势，那么该路径就会被舍弃。
2. 如果B路径的总体代价优于A，但A的启动代价优于B，那么保留A路径同时接受新路径B作为候选路径。如果A的启动代价也没有优势，那么接受B作为新的候选路径并舍弃A路径。
3. 如果A路径和B路径的总体代价相差不大，那么进一步比较其启动代价，保留启动代价更优的路径。如果启动代价相差也不大，那么需要看哪条路径的过滤性更好，预估返回行数更少的路径被保留。

上述是路径选择的总体原则，在实际应用中，比较逻辑更为复杂。例如：可以生成并行计划的场景需要考虑数据分布等因素，有路径参数的场景要进一步比较路径参数个数等，但总体策略都是基于代价信息来比较的，CBO优化器通过多层迭代，不断淘汰差的路径，最终选择出一棵完整的最优路径树。但这棵路径树上包含了大量执行器不需要的信息，并且缺少对于AGG函数、LIMIT等约束的处理，因此优化器会进一步补充一些算子逻辑，并将其转化为执行器可以理解的PlanTree，即最终呈现的计划树。这个计划在GaussDB中可以通过EXPLAIN命令打印出来，如下是一个简单的SeqScan计划示例：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE t1.a > 10;
          QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.05..236.84 rows=4991 width=16)
Node/s: All datanodes
-> Seq Scan on t1 (cost=0.05..28.84 rows=4991 width=16)
    Filter: (a > 10)
(4 rows)
```

该示例是一个完整的计划展示，包含了最终计划扫描路径（Seq Scan on t1）、过滤条件约束（Filter: (a > 10)）以及代价信息（cost=0.05..28.84 rows=4991 width=16），如启动代价、总体代价和返回行数估算。

下面结合示例，对GaussDB中常见的计划类型和算子进行介绍。建表语句如下：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 5000),
floor(random()*10));
```

```

INSERT 0 10000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# CREATE TABLE t2(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t2_a ON t2(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t2_b ON t2(b);
CREATE INDEX
gaussdb=# INSERT INTO t2 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 10000
gaussdb=# ANALYZE t2;
ANALYZE
gaussdb=# CREATE TABLE t3(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t3_a ON t3(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t3_b ON t3(b);
CREATE INDEX
gaussdb=# INSERT INTO t3 (a, b, c) VALUES (generate_series(1, 100), generate_series(1, 50),
floor(random()*10));
INSERT 0 100
gaussdb=# ANALYZE t3;
ANALYZE
gaussdb=# SET explain_perf_mode=normal;
SET
gaussdb=# SET enable_nestloop=off;
SET
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=off;
SET

```

- 分布式Stream计划:

```

gaussdb=# EXPLAIN SELECT count(t1.a),t1.c FROM t1 LEFT JOIN t2 ON t1.b = t2.b WHERE t1.a >
ALL(SELECT a FROM t3) GROUP BY t1.c ORDER BY t1.c LIMIT 10;
QUERY PLAN
-----
Limit (cost=5260.77..5260.78 rows=5 width=16)
-> Sort (cost=5260.77..5260.78 rows=5 width=16)
    Sort Key: t1.c
    -> HashAggregate (cost=5260.35..5260.71 rows=5 width=16)
        Group By Key: t1.c
        -> Streaming (type: GATHER) (cost=5260.35..5260.71 rows=15 width=16)
            Node/s: All datanodes
            -> HashAggregate (cost=5260.04..5260.09 rows=15 width=16)
                Group By Key: t1.c
                -> Hash Left Join (cost=364.96..5243.37 rows=10000 width=8)
                    Hash Cond: (t1.b = t2.b)
                    -> Streaming(type: REDISTRIBUTE) (cost=0.00..4836.74 rows=5000 width=12)
                        Spawn on: All datanodes
                        -> Index Scan using idx_t1_b on t1 (cost=0.00..4699.21 rows=5000
width=12)
                            Filter: (SubPlan 1)
                            SubPlan 1
                                -> Materialize (cost=0.00..2.02 rows=900 width=4)
                                    -> Streaming(type: BROADCAST) (cost=0.00..1.52 rows=300
width=4)
                                        Spawn on: All datanodes
                                        -> Seq Scan on t3 (cost=0.00..1.33 rows=100 width=4)
                                            -> Hash (cost=323.30..323.30 rows=9999 width=4)
                                                -> Streaming(type: REDISTRIBUTE) (cost=0.00..323.30 rows=10000
width=4)
                                                    Spawn on: All datanodes

```

```
-> Seq Scan on t2 (cost=0.00..48.33 rows=10000 width=4)
(24 rows)
```

这是一个典型的分布式SQL计划，也是分布式数据库中最常见的一类计划。该计划中包含了前文提到的主要扫描路径（索引扫描、顺序扫描路径）和连接路径（Hash Left Join）。此外，`t1.a > ALL(SELECT a FROM t3)`部分生成了一个子计划（SubPlan）在执行阶段可以由t1表的约束条件驱动独立执行，t1表的扫描算子上层是一个Stream(Redistribute)算子，在分布式数据库中，每张表的数据会根据分布键分布存储在不同的DN节点上，因此t1表与t2表Join时，需要根据b列等值条件对数据进行重分布，从而保证在每个DN节点上数据能够正确连接。Left Join上层是一个Group By的算子（HashAggregate），然后是Stream（Gather算子），该算子主要在CN节点上用于汇总各DN节点返回的计算结果，数据汇总完成后执行排序算子（Sort），最后是LIMIT算子（Limit）。从该计划可以看到，除用于分布式计算的Stream算子外，SQL执行计划的处理逻辑与原始SQL语句——映射。

- 分布式PGXC计划：

```
gaussdb=# SET enable_stream_operator = off;
SET
gaussdb=# EXPLAIN SELECT count(t1.a),t1.c FROM t1 LEFT JOIN t2 ON t1.b = t2.b WHERE t1.a >
ALL(SELECT a FROM t3) GROUP BY t1.c ORDER BY t1.c LIMIT 10;
QUERY PLAN
-----
Limit (cost=250.11..250.12 rows=5 width=8)
-> Sort (cost=250.11..250.12 rows=5 width=8)
    Sort Key: t1.c
    -> HashAggregate (cost=250.00..250.05 rows=5 width=8)
        Group By Key: t1.c
        -> Hash Right Join (cost=62.50..200.00 rows=10000 width=8)
            Hash Cond: (t2.b = t1.b)
            -> Data Node Scan on t2 "_REMOTE_TABLE_QUERY_" (cost=0.00..0.00 rows=10000
width=4)
                Node/s: All datanodes
            -> Hash (cost=0.00..0.00 rows=5000 width=12)
                -> Data Node Scan on t1 "_REMOTE_TABLE_QUERY_" (cost=0.00..0.00 rows=5000
width=12)
                    Node/s: All datanodes
                    Coordinator quals: (SubPlan 1)
                    SubPlan 1
                    -> Materialize (cost=0.00..0.50 rows=100 width=4)
                        -> Data Node Scan on t3 "_REMOTE_TABLE_QUERY_" (cost=0.00..0.00
rows=100 width=4)
                            Node/s: All datanodes
(17 rows)
```

PGXC计划的原理是将部分简单SQL查询直接下推到DN节点上计算并将返回的结果在CN节点上进行汇总，其并行度比Stream计划差，因此通常在Stream计划无法生成时才会选择PGXC计划。如上面示例，通过设置GUC参数`enable_stream_operator`关闭Stream计划生成逻辑强制选择PGXC计划，对比Stream计划该计划中没有Stream算子，并且直接将t1表、t2表和t3表的SeqScan扫描下推到了DN上执行，但其他的Join、分组、排序等操作均在CN节点上执行。

- 分布式下推语句计划：

分布式场景下还有一类查询计划，即对于简单的SQL语句直接将语句下推到每个DN节点上，由DN节点的优化器各自生成执行计划并执行，CN节点只负责汇总结果。典型示例如下所示：

```
gaussdb=# EXPLAIN SELECT * from t1 WHERE b > 10;
QUERY PLAN
-----
Data Node Scan on t1 "_REMOTE_TABLE_QUERY_" (cost=0.00..0.00 rows=9982 width=16)
Node/s: All datanodes
(2 rows)
```

从上面的SQL可以看出，这类计划的查询语句逻辑一般比较简单，不需要多表关联数据分布计算，也没有复杂的聚集函数、窗口函数、分组、排序等约束条件，其不需要依赖CN节点的额外计算能力，可以最大化利用节点的分布式并行计算能力。

上述示例删除建表语句如下：

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE t2;
DROP TABLE
gaussdb=# DROP TABLE t3;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

7.13.3 操作指导

前提条件

本节将结合示例展示优化器基本的计划选择过程，使用如下建表语句：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 10000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# CREATE TABLE t2(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t2_a ON t2(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t2_b ON t2(b);
CREATE INDEX
gaussdb=# INSERT INTO t2 (a, b, c) VALUES (generate_series(1, 10000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 10000
gaussdb=# ANALYZE t2;
ANALYZE
gaussdb=# SET explain_perf_mode=normal;
SET
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=on;
SET
```

操作示例

下面示例通过对同一个SQL的过滤条件取不同的常量值来展示代价模型对CBO优化器进行计划选择时的影响。首先，执行如下SQL语句并查看其选择的计划（过滤条件取值：t2.b <= 100）：

```
gaussdb=# EXPLAIN SELECT * FROM t1 JOIN t2 ON t1.a = t2.c AND t2.b <= 100;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=13.95..16.88 rows=100 width=32)
```

```
Node/s: All datanodes
-> Merge Join (cost=11.89..12.76 rows=100 width=32)
    Merge Cond: (t1.a = t2.c)
    -> Index Scan using idx_t1_a on t1 (cost=0.00..44.47 rows=5000 width=16)
    -> Sort (cost=11.84..12.09 rows=100 width=16)
        Sort Key: t2.c
        -> Streaming(type: REDISTRIBUTE) (cost=0.00..8.52 rows=100 width=16)
            Spawn on: All datanodes
            -> Index Scan using idx_t2_b on t2 (cost=0.00..3.73 rows=100 width=16)
                Index Cond: (b <= 100)
(11 rows)
```

当过滤条件取值变为 `t2.b <= 3000`时，优化器选择生成计划如下所示：

```
gaussdb=# EXPLAIN SELECT * FROM t1 JOIN t2 ON t1.a = t2.c AND t2.b <= 3000;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=49.51..299.97 rows=3000 width=32)
Node/s: All datanodes
-> Hash Join (cost=45.51..174.97 rows=3000 width=32)
    Hash Cond: (t2.c = t1.a)
    -> Streaming(type: REDISTRIBUTE) (cost=0.00..115.24 rows=3000 width=16)
        Spawn on: All datanodes
        -> Seq Scan on t2 (cost=0.00..28.84 rows=3000 width=16)
            Filter: (b <= 3000)
    -> Hash (cost=24.67..24.67 rows=5001 width=16)
        -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
(10 rows)
```

上述SQL中，t1表与t2表做INNER JOIN，连接条件为t1.a = t2.c AND t2.b <=100，其中连接条件t1.a = t2.c不变，当t2.b <= ? 常量值取值100时执行计划选择Merge Join路径，利用t1表和t2表上的索引条件提升执行性能；但当常量值变为3000时，执行计划变成了Hash Join路径，其表明随着优化器会根据连接表的不同、约束条件的变化选择合适的查询计划，从而在实际生产环境中能够有效减少人工调优的场景，提升效率。

后置操作

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE t2;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

7.13.4 常见问题

- 场景一：为什么不能生成索引扫描路径？

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# SET enable_fast_query_shipping=off;
SET
gaussdb=# SET enable_stream_operator = on;
SET
gaussdb=# SET explain_perf_mode=normal;
SET
```

建表语句如上面所示，执行以下SQL为什么没有办法走索引路径：

```
gaussdb=# EXPLAIN SELECT /*+indexscan(t1)*/ FROM t1;
WARNING: unused hint: IndexScan(t1)
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..233.05 rows=5000 width=16)
  Node/s: All datanodes
  -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
(3 rows)

gaussdb=# EXPLAIN SELECT /*+indexscan(t1)*/ FROM t1 WHERE t1.b >= 100;
WARNING: unused hint: IndexScan(t1)
QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.56..232.96 rows=4900 width=16)
  Node/s: All datanodes
  -> Seq Scan on t1 (cost=0.56..28.84 rows=4900 width=16)
      Filter: (b >= 100)
(4 rows)

gaussdb=# EXPLAIN SELECT /*+indexscan(t1)*/ FROM t1 ORDER BY c;
WARNING: unused hint: IndexScan(t1)
QUERY PLAN
-----
Streaming (type: GATHER) (cost=117.88..326.42 rows=5001 width=16)
  Merge Sort Key: c
  Node/s: All datanodes
  -> Sort (cost=113.88..118.05 rows=5001 width=16)
      Sort Key: c
      -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
(6 rows)

gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

索引的优势主要在于数据组织的有序性以及对于过滤性好的条件能够快速查找，否则随机读写对于顺序扫描没有优势。上面三种SQL语句要么没有支持走索引的过滤条件，要么排序条件不在索引列上，因此通常在生成扫描路径时会将索引路径裁剪掉，减小搜索空间。

- 场景二：如何控制选择不同的连接路径和连接顺序？

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# CREATE TABLE t2(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t2_a ON t2(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t2_b ON t2(b);
CREATE INDEX
gaussdb=# INSERT INTO t2 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t2;
ANALYZE
gaussdb=# SET explain_perf_mode=normal;
```

```
SET
gaussdb=# SET enable_nestloop=on;
SET
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=on;
SET
```

假设有如下SQL语句：

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.b = t2.b;
```

优化器默认选择的执行计划如下所示：

```
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.b = t2.b;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=187.03..576.53 rows=5000 width=32)
Node/s: All datanodes
-> Hash Left Join (cost=183.03..368.15 rows=5000 width=32)
    Hash Cond: (t1.b = t2.b)
    -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
        Spawn on: All datanodes
        -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
    -> Hash (cost=162.20..162.20 rows=5001 width=16)
        -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
            Spawn on: All datanodes
            -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)

(11 rows)
```

通常有两种办法来控制选择NestloopJoin计划：使用GUC控制或者使用HINT控制。使用GUC控制只能通过关闭其他路径生成的方式来提高Nestloop Join被选中的概率，而使用HINT则能更加精准地指定连接的类型，以下是两种控制方式的示例：

```
---使用GUC参数控制生成NestloopJoin计划
gaussdb=# SET enable_nestloop=on;
SET
gaussdb=# SET enable_hashjoin=off;
SET
gaussdb=# SET enable_mergejoin=off;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.b = t2.b;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..42220.27 rows=5000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..42011.90 rows=5000 width=32)
    Join Filter: (t1.b = t2.b)
    -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
        Spawn on: All datanodes
        -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
    -> Materialize (cost=0.00..170.53 rows=5000 width=16)
        -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
            Spawn on: All datanodes
            -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)

(11 rows)

---使用HINT控制生成NestloopJoin计划
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=on;
SET
gaussdb=# EXPLAIN SELECT /*+nestloop(t1 t2)*/ FROM t1 LEFT JOIN t2 ON t1.b = t2.b;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..42220.27 rows=5000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..42011.90 rows=5000 width=32)
    Join Filter: (t1.b = t2.b)
    -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
        Spawn on: All datanodes
```

```
-> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
-> Materialize (cost=0.00..170.53 rows=5000 width=16)
-> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
    Spawn on: All datanodes
-> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(11 rows)
```

通常连接顺序是根据生成的路径代价来选择，有些连接顺序的路径甚至无法生成，但对于可以生成的连接顺序，可以通过Leading Hint控制，仍以上面的SQL为例，让t2表做驱动表，其控制方式如下：

```
gaussdb=# SET enable_nestloop=off;
SET
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=off;
SET
gaussdb=# EXPLAIN SELECT /*+leading((t2 t1))*/ FROM t1 LEFT JOIN t2 ON t1.b = t2.b;
QUERY PLAN
-----
Streaming (type: GATHER) (cost=187.03..576.53 rows=5000 width=32)
Node/s: All datanodes
-> Hash Right Join (cost=183.03..368.15 rows=5000 width=32)
    Hash Cond: (t2.b = t1.b)
-> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
    Spawn on: All datanodes
-> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
-> Hash (cost=162.20..162.20 rows=5001 width=16)
    -> Streaming(type: REDISTRIBUTE) (cost=0.00..162.20 rows=5000 width=16)
        Spawn on: All datanodes
    -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
(11 rows)
```

该计划通过leading((t2 t1)) 这个HINT，强制指定t2表做驱动表，t1表做被驱动表。最后执行如下语句删除t1表和t2表：

```
gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE t2;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

更多关于Join顺序、Join方式的HINT使用可以参考：《SQL调优指南》中“使用Plan Hint进行调优 > Join顺序的Hint”和“使用Plan Hint进行调优 > Join方式的Hint”章节。

- 场景三：为什么无法生成Hash Join连接路径？

以下面的SQL场景为例：

```
gaussdb=# CREATE TABLE t1(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t1_a ON t1(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
gaussdb=# INSERT INTO t1 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t1;
ANALYZE
gaussdb=# CREATE TABLE t2(a INT, b INT, c INT, d INT);
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.
CREATE TABLE
gaussdb=# CREATE INDEX idx_t2_a ON t2(a);
CREATE INDEX
gaussdb=# CREATE INDEX idx_t2_b ON t2(b);
CREATE INDEX
```



```
gaussdb=# INSERT INTO t2 (a, b, c) VALUES (generate_series(1, 5000), generate_series(1, 5000),
floor(random()*10));
INSERT 0 5000
gaussdb=# ANALYZE t2;
ANALYZE
gaussdb=# SET explain_perf_mode=normal;
SET
gaussdb=# SET enable_nestloop=off;
SET
gaussdb=# SET enable_hashjoin=on;
SET
gaussdb=# SET enable_mergejoin=on;
SET
gaussdb=# EXPLAIN SELECT * FROM t1 LEFT JOIN t2 ON t1.b != t2.b;
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..1166996.01 rows=24995000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..125537.63 rows=24995000 width=32)
    Join Filter: (t1.b <> t2.b)
    -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
    -> Materialize (cost=0.00..500.46 rows=15000 width=16)
        -> Streaming(type: BROADCAST) (cost=0.00..475.46 rows=15000 width=16)
            Spawn on: All datanodes
            -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(9 rows)

gaussdb=# EXPLAIN SELECT /*+hashjoin(t1 t2)*/ FROM t1 LEFT JOIN t2 ON t1.b != t2.b;
WARNING: unused hint: HashJoin(t1 t2)
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..1166996.01 rows=24995000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..125537.63 rows=24995000 width=32)
    Join Filter: (t1.b <> t2.b)
    -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
    -> Materialize (cost=0.00..500.46 rows=15000 width=16)
        -> Streaming(type: BROADCAST) (cost=0.00..475.46 rows=15000 width=16)
            Spawn on: All datanodes
            -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(9 rows)

gaussdb=# EXPLAIN SELECT /*+mergejoin(t1 t2)*/ FROM t1 LEFT JOIN t2 ON t1.b != t2.b;
WARNING: unused hint: MergeJoin(t1 t2)
               QUERY PLAN
-----
Streaming (type: GATHER) (cost=4.00..1166996.01 rows=24995000 width=32)
Node/s: All datanodes
-> Nested Loop Left Join (cost=0.00..125537.63 rows=24995000 width=32)
    Join Filter: (t1.b <> t2.b)
    -> Seq Scan on t1 (cost=0.00..24.67 rows=5000 width=16)
    -> Materialize (cost=0.00..500.46 rows=15000 width=16)
        -> Streaming(type: BROADCAST) (cost=0.00..475.46 rows=15000 width=16)
            Spawn on: All datanodes
            -> Seq Scan on t2 (cost=0.00..24.67 rows=5000 width=16)
(9 rows)

gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP TABLE t2;
DROP TABLE
gaussdb=# RESET explain_perf_mode;
RESET
```

上述SQL，即使通过HINT或GUC参数也无法控制生成HashJoin计划或MergeJoin计划。实际上除了NestloopJoin是通用路径生成算法外，HashJoin和MergeJoin路径的生成都有一些约束条件。对于HashJoin来说，其算法原理是基于Hash表的，对于非等值连接的情况无法通过Hash表快速查找目标数据，也就不能生成HashJoin路径。

7.14 AutoHint：智能 HINT 推荐

概述

AutoHint（智能HINT推荐）将自动识别慢SQL执行计划树中耗时较长的算子，生成候选HINT进行替换，并通过贪心策略进行探索，最终生成能够提升当前慢SQL执行性能的HINT组合推荐给用户。在提升慢SQL性能的同时可以为用户节约人工调优所带来的精力与时间成本。

须知

容灾集群下不支持智能HINT推荐。

前置条件

- 数据库运行正常。
- 确保当前用户拥有super权限。
- 若反馈基数估计与自适应代价估计功能开启，需要提前运行多次查询保证反馈基数估计模型与代价估计模型收敛。

使用指导

参考AutoHint语法运行相关查询即可得到HINT推荐结果。

最佳实践

反馈基数估计功能与自适应代价关闭场景（设置enable_adaptive_cost=off, adaptive_cardest_strategy=use_statistics）：

步骤1 启动数据库，导入数据。

步骤2 执行慢SQL语句，如含有多种连接路径和连接类型的查询SQL，观察默认场景下计划的执行时间与执行树。以JOB数据集的query11d语句为例。

```
gaussdb=# EXPLAIN ANALYZE SELECT MIN(cn.name) AS from_company,
    MIN(mc.note) AS production_note,
    MIN(t.title) AS movie_based_on_book
FROM company_name AS cn,
    company_type AS ct,
    keyword AS k,
    link_type AS lt,
    movie_companies AS mc,
    movie_keyword AS mk,
    movie_link AS ml,
    title AS t
WHERE cn.country_code != '[pl]'
    AND ct.kind != 'production companies'
    AND ct.kind IS NOT NULL
    AND k.keyword IN ('sequel',
        'revenge',
        'based-on-novel')
    AND mc.note IS NOT NULL
    AND t.production_year > 1965
    AND lt.id = ml.link_type_id
    AND ml.movie_id = t.id
```

```
AND t.id = mk.movie_id
AND mk.keyword_id = k.id
AND t.id = mc.movie_id
AND mc.company_type_id = ct.id
AND mc.company_id = cn.id
AND ml.movie_id = mk.movie_id
AND ml.movie_id = mc.movie_id
AND mk.movie_id = mc.movie_id;
```

步骤3 使用AutoHint相关命令进行Hint推荐。

```
gaussdb=# AUTOHINT ANALYZE
SELECT MIN(cn.name) AS from_company,
       MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_based_on_book
FROM company_name AS cn,
     company_type AS ct,
     keyword AS k,
     link_type AS lt,
     movie_companies AS mc,
     movie_keyword AS mk,
     movie_link AS ml,
     title AS t
WHERE cn.country_code != '[pl]'
  AND ct.kind != 'production companies'
  AND ct.kind IS NOT NULL
  AND k.keyword IN ('sequel',
                   'revenge',
                   'based-on-novel')
  AND mc.note IS NOT NULL
  AND t.production_year > 1965
  AND lt.id = ml.link_type_id
  AND ml.movie_id = t.id
  AND t.id = mk.movie_id
  AND mk.keyword_id = k.id
  AND t.id = mc.movie_id
  AND mc.company_type_id = ct.id
  AND mc.company_id = cn.id
  AND ml.movie_id = mk.movie_id
  AND ml.movie_id = mc.movie_id
  AND mk.movie_id = mc.movie_id;
```

步骤4 得到推荐HINT与预期提升效果。

```
          AUTOHINT
-----
Recommended hintset:
  MergeJoin(@sel$1 k@sel$1 mk@sel$1 mc@sel$1)

Expected boost:
planning:   -7%
execution:  62%
overall:    60%
```

步骤5 使用推荐的HINT，观察执行计划是否改变，以及执行性能是否得到提升。

```
gaussdb=# EXPLAIN ANALYZE SELECT /*+MergeJoin(@sel$1 k@sel$1 mk@sel$1 mc@sel$1)*/ MIN(cn.name)
AS from_company,
       MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_based_on_book
FROM company_name AS cn,
     company_type AS ct,
     keyword AS k,
     link_type AS lt,
     movie_companies AS mc,
     movie_keyword AS mk,
     movie_link AS ml,
     title AS t
WHERE cn.country_code != '[pl]'
  AND ct.kind != 'production companies'
  AND ct.kind IS NOT NULL
  AND k.keyword IN ('sequel',
```

```
'revenge',  
      'based-on-novel')  
AND mc.note IS NOT NULL  
AND t.production_year > 1965  
AND lt.id = ml.link_type_id  
AND ml.movie_id = t.id  
AND t.id = mk.movie_id  
AND mk.keyword_id = k.id  
AND t.id = mc.movie_id  
AND mc.company_type_id = ct.id  
AND mc.company_id = cn.id  
AND ml.movie_id = mk.movie_id  
AND ml.movie_id = mc.movie_id  
AND mk.movie_id = mc.movie_id;
```

----结束

反馈基数估计功能与自适应代价估计功能开启场景：

导入数据后，执行慢SQL多次，直到观察到SQL执行计划稳定后，说明反馈基数估计模型与代价估计模型已收敛，后续操作与上述步骤一致。

常见问题处理

可能会遇到网络问题与最大连接数问题导致探索过程失败，此时请尝试更改最大连接数上限，或等待资源空闲后再重新使用该功能。

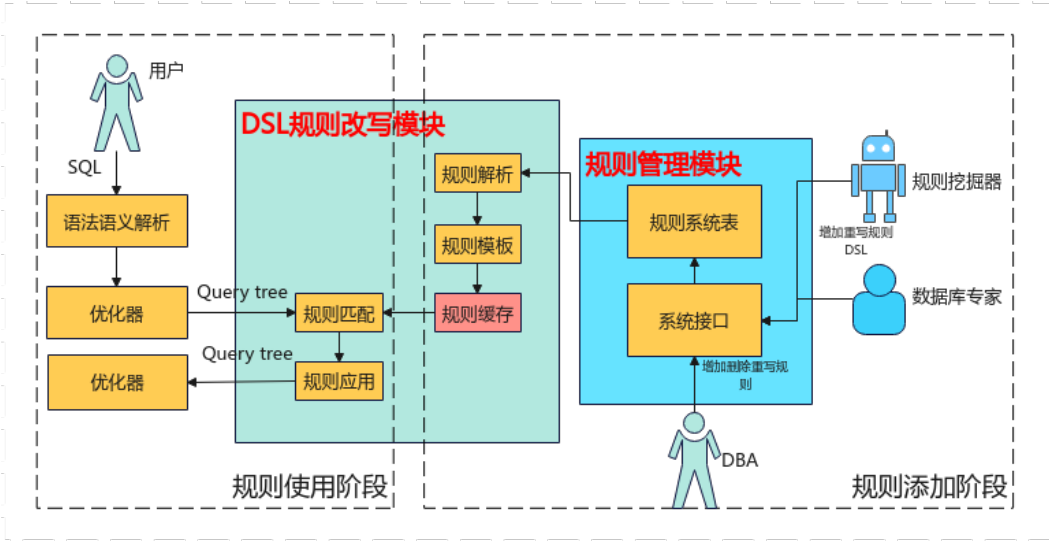
7.15 基于 DSL 的规则引擎

7.15.1 DSL 规则引擎整体流程

DSL规则引擎使用DSL语言对SQL的查询改写规则进行描述，通过统一的原子化接口，形成特定语法，利用这些语法的灵活编排组合，符号化的描述能力，使得查询改写规则可以插件化集成，动态加载，从而增强查询改写的易开发、易部署、灵活可扩展性。

DSL规则引擎处于SQL引擎中，且主要在运行态利用加载的DSL规则对SQL进行等价改写，从而帮助数据库在优化器阶段生成更好的执行路径，提升执行性能，DSL规则引擎整体流程如[图7-33](#)所示。

图 7-33 DSL 规则引擎整体流程



如上图所示，DSL 规则引擎实现主要包含两个阶段：

规则添加阶段维护人员通过 DSL 规则管理模块添加新的 DSL 改写规则，并通过 RELOAD 机制将系统表中的内容同步至 DSL 规则引擎的全局缓存池，然后 DSL 规则引擎在规则匹配与应用框架运行阶段即可从缓存池中加载这些规则，并按照优先级策略依次遍历规则集完成匹配改写。

- 规则添加阶段：
 - DSL 规则来源分为两类：内置规则和外部发现的规则，内置规则如表 7-18 所示，对于内置规则不允许删除，只能禁用。外部规则由 DBA 使用 DSL 语言编写而来，其等价性系统无法保证，因此 DSL 规则引擎约束对于 DSL 规则本身的等价性，异常场景的匹配、替换应由规则设计者保证，对于这些外部来源的 DSL 规则集，引擎只能保证框架本身调度逻辑功能的正确性。
 - 新增 DSL 规则添加成功并执行 RELOAD 操作后可以被应用。DSL 规则引擎在运行过程中应关注 DSL 规则集的生命周期变化，对于新的 SQL 会话请求，应使用变更后的 DSL 规则集进行改写流程调度；而对于已经处于改写流程中的会话则应继续使用旧的副本完成整个查询重写流程。
- 规则使用阶段，完整流程主要包含：
 - a. 用户输入原始 SQL 请求，数据库经过语法、语义解析后得到 QueryTree。
 - b. 使用步骤 1 获得的 QueryTree 作为入参进入 DSL 规则引擎的改写模块，该模块位于优化器的查询重写规则模型中，其应用包含以下子流程：
 - i. 对于每条 DSL 规则，分为匹配流程和应用流程；匹配流程中将 QueryTree 的节点与模板进行匹配，找到满足条件的节点；然后应用规则对节点进行等价改写，生成目标 QueryTree。
 - ii. 对于整个规则集，DSL 规则引擎会按照规则的开关状态进行过滤，并按照规则的优先级进行排序，然后依次调度每条 DSL 规则对 QueryTree 执行上一步中的流程。
 - iii. DSL 规则引擎改写的终止主要考虑两种情况：一是当规则集被完整调度两次，且均无任何改写发生时则终止调度流程；二是当规则集调度次数达到系统设置的阈值时，则强行终止改写流程。

表 7-18 内置规则介绍

场景名称	子场景名称	简要描述	SQL示例
场景一：JOIN连接消除	- 子场景1.1：具有DISTINCT去重语义的外连接消除 rulename: distinct_enabled_outer_join_elimination - 子场景1.2：具有GROUP BY条件去重语义的外连接消除 rulename: group_by_outer_join_elimination	- 子场景1.1：对于左外连接或右外连接，当投影列中只包含外表的字段值并且有DISTINCT关键字约束时，能消除该外连接。 规则默认开启状态：开启。 - 子场景1.2：当GROUP BY条件满足DISTINCT的去重语义时也可以消除外连接。 规则默认开启状态：开启。	- 子场景1.1： SELECT DISTINCT t1.* FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1; 其等价改写结果如下所示： SELECT DISTINCT t1.* FROM t1; - 子场景1.2： SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.c = t2.c GROUP BY t1.a, t1.b, t1.c; 其等价改写结果如下所示： SELECT t1.* FROM t1;
	子场景2：JOIN条件恒为FALSE的外连接消除 rulename: always_false_outer_join_elimination	- 对于左外连接或右外连接，若JOIN条件恒为FALSE，外表的每一行都无法和内表找到匹配行，连接后的投影列为外表每一行均返回一次并对内表投影列补NULL值，因此能消除该外连接。 - 规则默认开启状态：开启。	（假设T1和T2表均只有一个属性列） SELECT * FROM t1 LEFT JOIN t2 ON 1 = 0; 其等价改写结果如下所示： SELECT t1.c1, NULL FROM t1;

场景名称	子场景名称	简要描述	SQL示例
	子场景3：同表同列的SEMI JOIN消除 rulename: same_table_semi_join_elimination	- 结合本场景SQL示例进行说明，当主查询和子查询引用的均为t1表且连接条件为同一列，其中t1表需为简单表，根据EXISTS子查询的语义，若t1.c1不全为NULL，则经过EXISTS子查询过滤后一定可返回t1.c1不为NULL的行；若t1.c1全为NULL，那么EXISTS子查询一定返回FALSE，此时结果集为空。该场景可以进行一些等价优化，消除SEMI JOIN部分。 - 规则默认开启状态：开启。	<pre>SELECT * FROM t1 WHERE EXISTS(SELECT 1 FROM t1 tt WHERE t1.c1 = tt.c1);</pre> 其等价改写结果如下所示： <pre>SELECT * FROM t1 WHERE t1.c1 IS NOT NULL;</pre>
场景二：LIMIT条件下压	- 子场景1.1：外连接Limit下推子查询 rulename: left_join_branch_limit_pushdown - 子场景1.2：交叉连接Limit下推子查询 rulename: cross_join_branch_limit_pushdown	- 如果外连接或交叉连接中没有WINDOW FUNCTION、DISTINCT、GROUP BY或HAVING，且WHERE条件或者ORDER BY仅和连接的一侧表有关时，可将LIMIT语句下压到连接的表的一侧（外连接）或多侧（多表交叉连接）。通过LIMIT下推可有效减少连接的行数，从而降低计划执行的性能开销。 - 规则默认开启状态：开启。	- 子场景1.1： <pre>SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1 LIMIT 1;</pre> 其等价改写结果如下所示： <pre>SELECT * FROM (SELECT * FROM t1 LIMIT 1) t LEFT JOIN t2 ON t.c1 = t2.c1 LIMIT 1;</pre> - 子场景1.2： <pre>SELECT 1 FROM t1, t2 WHERE t1.c1 > 0 ORDER BY t1.c1 LIMIT 1;</pre> 其等价改写结果如下所示： <pre>SELECT 1 FROM (SELECT 1 FROM t1 WHERE t1.c1 > 0 ORDER BY t1.c1 LIMIT 1) tm1, (SELECT 1 FROM t2 LIMIT 1) tm2 LIMIT 1;</pre>

场景名称	子场景名称	简要描述	SQL示例
场景三：冗余的DISTINCT消除	- 子场景1.1：投影列全是常量时可以消除DISTINCT约束 rulename: const_only_distinct_elimination_base - 子场景1.2：投影列全是常量，且有OFFSET且OFFSET值>0时可以消除DISTINCT约束，整个SQL语句可以规约为约束条件为FALSE rulename: const_only_distinct_elimination_limit - 子场景1.3：投影列全是常量，且GROUP BY约束包含所有的投影列，可以消除GROUP BY rulename: const_only_distinct_elimination_groupby	- 子场景1.1：如果SELECT的投影列中只包含常量，则可以通过添加LIMIT 1约束消除DISTINCT，从而提前结束对数据的遍历。 规则默认开启状态：开启。 - 子场景1.2：如果SELECT的投影列中只包含常量，且有OFFSET且OFFSET值>0时，则可以消除约束DISTINCT，整个SQL语句可以规约为约束条件为FALSE，提前结束SQL执行。 规则默认开启状态：关闭。 - 子场景1.3：如果SELECT的投影列中只包含常量，且GROUP BY约束包含所有的投影列，没有HAVING条件、GROUPING SET时，可以消除GROUP BY增加LIMIT 1约束，从而提前结束对数据的遍历。 规则默认开启状态：关闭。	- 子场景1.1： SELECT DISTINCT 1,2 FROM t1; 其等价改写结果如下所示： SELECT 1,2 FROM t1 LIMIT 1; - 子场景1.2： SELECT DISTINCT 1,2 FROM t1 OFFSET 1; 其等价改写结果如下所示： SELECT 1,2 FROM t1 WHERE FALSE; - 子场景1.3： SELECT 1,2 FROM t1 GROUP BY 1,2; 其等价改写结果如下所示： SELECT 1,2 FROM t1 LIMIT 1;

场景名称	子场景名称	简要描述	SQL示例
	- 子场景2.1： 投影列包含唯一键约束时可以消除DISTINCT约束 rulename: unique_key_distinct_elimination_base - 子场景2.2： GROUP BY包含唯一约束投影列时可以消除GROUP BY rulename: unique_key_distinct_elimination_groupby	- 子场景2.1： 如果SELECT语句的投影列中包含唯一性约束的列时，且FROM中不包含复杂表，则可以消除DISTINCT，假设有t1表且其列字段c1上存在唯一非空约束。 规则默认开启状态：开启。 - 子场景2.2： 如果GROUP BY包含唯一约束投影列其语义上等价于DISTINCT约束，因此该场景也可以消除GROUP BY。 规则默认开启状态：关闭。	- 子场景2.1： SELECT DISTINCT c1, c2, c3 FROM t1; 其等价改写结果如下所示： SELECT c1, c2, c3 FROM t1; - 子场景2.2： SELECT t1.a, t1.b, t1.c FROM t1 GROUP BY t1.a; 其等价改写结果如下所示： SELECT t1.a, t1.b, t1.c FROM t1;
场景四：复杂场景扩展	子场景1：SEMI JOIN加速 STREAM计划中的外连接 rulename: semi_join_accelerated_outer_join	- 当小表LEFT JOIN大表时，如果连接条件不是分布列，则大表必须重分布而不能BROADCAST小表，导致性能较差；此时可以将大表先和小表做一次半连接，过滤掉大部分数据后，再与小表做外连接，从而可以将小表BROADCAST。 - 规则默认开启状态：关闭。	<pre>SELECT * FROM t2 small LEFT JOIN t1 big ON small.c1 = big.c1; 其等价改写结果如下所示： SELECT * FROM t2 small LEFT JOIN (SELECT * FROM t1 big WHERE big.c1 IN (SELECT c1 FROM t2)) fakebig ON small.c1 = fakebig.c1;</pre>

场景名称	子场景名称	简要描述	SQL示例
	子场景2：两个表达式子链接比较且存在子结构关系时消减子链接 rulename: subLink_compare_elimination	- 两个包含Count聚集函数的表达式子链接进行等值比较时，如果其中一个子链接可以表示为另一个子链接的子结构，且过滤条件仅相差一个EXISTS子链接条件时，可以对其进行优化。因为在此场景下，实际是想找到EXISTS条件返回FALSE的情况。在这种情况下两个count值是否相等取决于其他条件是否为TRUE。当其他条件为TRUE时，根据AND条件的语义，两个子链接的过滤条件相反，count()取值一定不同；否则取值相同。因此，对于这种场景可以使用NOT EXISTS等价改写消减子链接的数量。 - 规则默认开启状态：开启。	<pre>SELECT t1.a, t1.b FROM t1,t2 WHERE t1.a = t2.a AND t1.b = t1.b AND (SELECT COUNT(*) FROM t1 as t11 WHERE t11.a = t1.a AND t11.b = t1.b) = (SELECT COUNT(*) FROM t1 AS t21 WHERE t21.a = t1.a AND t21.b = t1.b AND EXISTS(SELECT 1 FROM t3 WHERE t1.a = t3.a));</pre> 其等价改写结果如下所示： <pre>SELECT t1.a, t1.b FROM t1,t2 WHERE t1.a = t2.a AND t1.b = t1.b AND NOT EXISTS (SELECT 1 FROM t1 as t11 WHERE t11.a = t1.a AND t11.b = t1.b AND NOT EXISTS(SELECT 1 FROM t3 WHERE t1.a = t3.a));</pre>

场景名称	子场景名称	简要描述	SQL示例
	- 子场景3.1: 常值谓词条件下推到子查询投影列中 rulename: const_predicate_pushdown_to_project - 子场景3.2: 常值谓词条件下推到子查询过滤条件中 rulename: const_predicate_pushdown_to_filter	- 当主查询中包含常值谓词条件时，即表达式一侧是表的字段另一侧是常量，且子链接的查询条件中包含相同表字段时，则可以考虑将该常量条件下推到子链接中。 - 规则默认开启状态：关闭。	- 子场景3.1: SELECT t1.a, (SELECT sum(t2.a) FROM t2 WHERE t2.b = t1.b) FROM t1 WHERE t1.b = 1; 其等价改写结果如下所示: SELECT t1.a, (SELECT sum(t2.a) FROM t2 WHERE t2.b = t1.b AND t1.b = 1) FROM t1 WHERE t1.b = 1; - 子场景3.2: SELECT t1.a FROM t1 WHERE t1.b = 1 AND (SELECT SUM(t2.a) FROM t2 WHERE t2.b = t1.b) = 0; 其等价改写结果如下所示: SELECT a FROM t1 WHERE b = 1 AND (SELECT sum(t2.a) FROM t2 WHERE t2.b = t1.b AND t1.b = 1) = 0;
	子场景4: 相关表达式子链接转交叉连接 rulename: subLink_to_crossJoin	- 对于有聚集函数的等值表达式子链接，且子链接与常量比较时，可以将子链接提取为CTE表与主查询做交叉连接，在分布式场景特定的数据分布情况下，可以利用STREAM的BROADCAST提升性能。 - 规则默认开启状态：关闭。	SELECT * FROM t1 WHERE t1.a = 1 AND (SELECT count(*) FROM t2 WHERE t2.b = t1.b AND t2.b = 1) = 3; 其等价改写结果如下所示: WITH cte AS (SELECT count(*) AS c, b FROM t2 WHERE t2.b = 1 GROUP BY b) SELECT * FROM t1, cte WHERE t1.a = 1 AND cte.b = t1.b AND cte.c = 3;

场景名称	子场景名称	简要描述	SQL示例
	子场景5：表达式子链接聚集函数为Count与0值比较时，可以消除聚集操作 rulename: count_zero_elimination	- 对于表达式相关子链接，如果子链接中的投影列是count()函数，且子链接与0做等值比较，则从语义上表示只有在过滤条件不满足时才为TRUE，因此将其等价改写为LEFT JOIN，并通过对内表的列进行IS NULL过滤，从而消除聚集函数和子计划。 - 规则默认开启状态：开启。	<pre>SELECT t1.* FROM t1 WHERE t1.a = 1 AND (SELECT count(*) FROM t2 WHERE t2.b = t1.b AND t2.b = 1) = 0; 其等价改写结果如下所示： SELECT t1.* FROM t1 LEFT JOIN t2 ON t2.b = 1 AND t2.b = t1.b WHERE t1.a = 1 AND t2.b IS NULL;</pre>
场景五：内连接唯一约束的自连接消除	基于DSL的规则引擎实现唯一约束的自连接消除需要如下两条规则： - 规则一：内连接查询转换为交叉连接 rulename: inner_join_to_cross_join - 规则二：交叉连接唯一约束的自连接消除 rulename: self_crossjoin_elimination	- 当查询块中对多表进行主键或非空唯一约束上的内连接时，可通过查询重写规则消除冗余连接操作，从而减少查询中不必要的连接运算并提升查询整体性能。其原理是相同表在主键或非空唯一约束上的内连接具有严格的一对一匹配关系，因此，其结果集存在数据冗余，所有数据均可通过单次扫描原始表获取。 - 规则默认开启状态：开启。	<pre>SELECT * FROM t1 tt1 INNER JOIN t1 tt2 ON tt1.a = tt2.a INNER JOIN t1 tt3 ON tt1.a = tt3.a WHERE tt1.b > 100; 其等价改写结果如下所示： SELECT tt1.*, tt1.*,tt1.* FROM t1 tt1 WHERE tt1.b > 100 AND tt1.a IS NOT NULL;</pre>

场景名称	子场景名称	简要描述	SQL示例
场景六：ALL非相关子链接改写	- 子场景1.1：过滤条件中的ALL非相关子链接支持改写为MAX聚集函数的表达式子链接 rulename: allsublink_add_max_filter - 子场景1.2：过滤条件中的ALL非相关子链接支持改写为MIN聚集函数的表达式子链接 rulename: allsublink_add_min_filter - 子场景1.3：JOIN条件中的ALL非相关子链接支持改写为MAX聚集函数的表达式子链接 rulename: allsublink_add_max_join - 子场景1.4：JOIN条件中的ALL非相关子链接支持改写为MIN聚集函数的表达式子链接 rulename: allsublink_add_min_join	- 典型SQL如下： SELECT a FROM t1 WHERE a > ALL(SELECT a FROM t2); 对于上述SQL查询，表示遍历t1表并且返回满足过滤条件的行：即t1表的a列大于t2表a列的所有值。从等价逻辑可推导出，该条件等价于确保t1表的a列大于t2表a列的最大值。对于过滤条件或JOIN条件中的 >ALL、>= ALL、< ALL、<= ALL类型的非相关子链接比较场景，均可通过等价改写来优化性能。 - 规则默认开启状态：关闭。	- 子场景1.1： SELECT a FROM t1 WHERE a > ALL(SELECT a FROM t2); 其等价改写结果如下所示： SELECT a FROM t1 WHERE CASE WHEN ((SELECT max(t2.a) AS a FROM t2)) IS NULL THEN true ELSE a > ((SELECT max(t2.a) AS a FROM t2)) END; - 子场景1.2： SELECT a FROM t1 WHERE a < ALL(SELECT a FROM t2); 其等价改写结果如下所示： SELECT a FROM t1 WHERE CASE WHEN ((SELECT min(t2.a) AS a FROM t2)) IS NULL THEN true ELSE a < ((SELECT min(t2.a) AS a FROM t2)) END; - 子场景1.3： SELECT * FROM t1 LEFT JOIN t4 ON t1.b = t4.b AND t1.a > ALL(SELECT a FROM t2); 其等价改写结果如下所示： SELECT * FROM t1 LEFT JOIN t4 ON t1.b = t4.b AND CASE WHEN ((SELECT max(t2.a) AS a FROM t2)) IS NULL THEN true ELSE t1.a > ((SELECT max(t2.a) AS a FROM t2)) END; - 子场景1.4： SELECT * FROM t1 LEFT JOIN t4 ON t1.b = t4.b AND t1.a < ALL(SELECT a FROM t2); 其等价改写结果如下所示： SELECT * FROM t1 LEFT JOIN t4 ON t1.b = t4.b AND CASE WHEN ((SELECT min(t2.a) AS a FROM t2)) IS NULL THEN true ELSE t1.a < ((SELECT min(t2.a) AS a FROM t2)) END; 说明 示例1.1及1.3中>ALL子链接的优化规则同样适用于>=连接的场景；同理，示例1.2及1.4使用<ALL的优化规则同样适用于<=连接的场景。

场景名称	子场景名称	简要描述	SQL示例
场景七：HAVING中MIN/MAX谓词下推过滤条件	子场景：支持HAVING中MIN/MAX谓词下推过滤条件 rulename: minmax_predicate_pushdown	- 在某些场景下，可将HAVING条件中的谓词下推至WHERE条件中，在保证结果一致性的前提下提前过滤数据，从而优化执行性能，其中一类场景为HAVING条件中包含MIN、MAX比较谓词，典型SQL示例如下： SELECT a, min(b) FROM t1 GROUP BY a HAVING min(b) < 10; 分析上述SQL查询可知，其目标是筛选出min(b)<10分组后的聚合结果。若能在分组前过滤掉b<10的行，则可减少参与计算的数据量。基于此逻辑，可推导出新的谓词b<10，并将其下推至过滤条件中。 - 规则默认开启状态：开启。	- 示例1： SELECT a, min(b) FROM t1 GROUP BY a HAVING min(b) < 10; 其等价改写结果如下所示： SELECT a, min(b) FROM t1 WHERE b < 10 GROUP BY a HAVING min(b) < 10; - 示例2： SELECT a, max(b) FROM t1 GROUP BY a HAVING max(b) > 10; 其等价改写结果如下所示： SELECT a, max(b) FROM t1 WHERE b > 10 GROUP BY a HAVING max(b) > 10; 说明 示例1中HAVING谓词中使用<连接的优化规则同样适用于<=连接的场景；同理，示例2中HAVING谓词使用>连接的优化规则同样适用于>=连接的场景。

场景名称	子场景名称	简要描述	SQL示例
场景八：LIMIT下推 UNION ALL子查询	子场景：支持 LIMIT下推 UNION ALL子查询 rulename: unionall_limit_pushdown	- 在SQL标准语法中，UNION ALL用于合并两个或多个SELECT子查询的结果集，且不对结果进行排序和去重，而LIMIT则用于限制返回最终数据集的条数。如果将ORDER BY条件和LIMIT条件均下推至子查询中，可以利用索引的有序性消除排序条件并且利用LIMIT条件提前结束每个子查询的计算，这样，主查询仅需对少量数据进行二次排序再取前n条结果即可（n为LIMIT所指定的数字）。典型SQL示例如下： <pre>SELECT * FROM (SELECT * FROM t1 UNION ALL SELECT * FROM t2) ORDER BY a1 LIMIT 2;</pre> - 升级场景规则默认状态：关闭； 新安装场景规则默认状态：开启。	<pre>SELECT * FROM (SELECT * FROM t1 UNION ALL SELECT * FROM t2) LIMIT 2; 其等价改写结果如下所示： SELECT * FROM ((SELECT * FROM t1 LIMIT 2) UNION ALL (SELECT * FROM t2 LIMIT 2)) LIMIT 2;</pre>

场景名称	子场景名称	简要描述	SQL示例
场景九：LIMIT下推子查询	子场景：支持LIMIT下推子查询 rulename: subquery_limit_pushdown	- 通常SQL语句的执行顺序如下： FROM -> JOIN ON -> WHERE -> GROUP BY -> HAVING -> SELECT -> DISTINCT -> ORDER BY -> LIMIT 当执行至SELECT时，需对前序结果集中的每条数据进行计算，若涉及子链接等复杂计算，整体执行时间将成倍增加。从以上执行顺序可知，排序操作和LIMIT操作通常最后执行。从结果等价性角度分析，优先过滤冗余结果集后再对有效数据执行子链接计算，既可保证执行结果的正确性，又能有效提升执行性能，典型SQL示例如下： <pre>SELECT (SELECT min(a) FROM t2 WHERE t2.c >= t1.b) AS a1 FROM t1 INNER JOIN t3 ON t1.a = t3.a ORDER BY t1.a LIMIT 10;</pre> - 升级场景规则默认状态：关闭；新安装场景规则默认状态：开启。	<pre>SELECT (SELECT min(a) FROM t2 WHERE t2.c >= t1.b) AS a1 FROM t1 INNER JOIN t3 ON t1.a = t3.a ORDER BY t1.a LIMIT 10;</pre> 其等价改写结果如下所示： <pre>SELECT (SELECT min(a) FROM t2 WHERE t2.c >= t.b) AS a1 FROM (SELECT t1.* FROM t1 INNER JOIN t3 ON t1.a = t3.a ORDER BY t1.a LIMIT 10) t ORDER BY t.a LIMIT 10;</pre>

7.15.2 DSL 规则编写规范

7.15.2.1 DSL 规则语法组件介绍

DSL规则引擎定义了一套语言语法规则，DSL语法可以描述为：“原模板 => ”目标模板”的形式，用于将QueryTree与原模板匹配的节点，改写成根据目标模板构建的节点。其中模板由：语法关键词、符号、变量、函数和约束组成，其组成方式如图7-34所示。

图 7-34 匹配模板语法

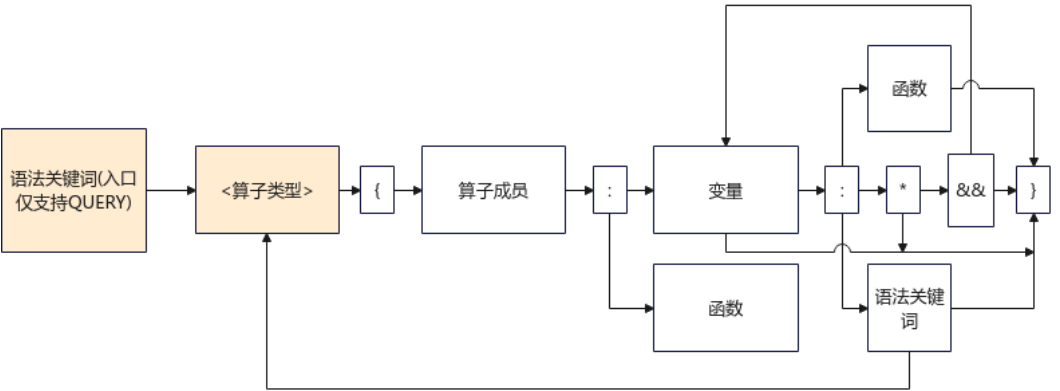


表 7-19 DSL 规则语法组件介绍

语法组件名称	语法组件介绍	约束
语法关键词	- 关键词名称：对应QueryTree的一部分。如FILTER关键词，表示QueryTree中的约束条件。 <>：表示关键词的类型。如JOIN<INNER/LEFT>表示JOIN中的各种类型。为空表示使用默认类型，默认类型定义请参考表7-20和表7-21中支持类型的相关说明。 - 每个算子用括号进行分装‘{’ ‘}’内表示整个语法关键词的内容。每个语法关键词有自己的变量对应QueryTree表达式中对应的成员。如JOIN<INNER>{LEFT: RIGHT: ON: }。表示对应逻辑计划中的JOIN表达式，且是INNER JOIN，其中LEFT表示JOIN的左子树，RIGHT表示JOIN的右子树，ON表示JOIN的连接条件。	- 入口关键词仅支持QUERY关键词。 - 无默认类型的关键词在匹配时，不允许类型为空，否则会在解析阶段报错。
符号	- 冒号 ':'：表示解释符。是对算子成员或变量的赋值操作。 - 且 '&&': 表示递进，也表示需要满足符号连接的所有条件。 - 通配符 '*': 表示全部匹配。	-

语法组件名称	语法组件介绍	约束
变量	DSL提供'\$' + var_string的形式表达变量。变量用于在和QueryTree中的表达式进行匹配时，在表达式匹配成功后将变量和表达式形成映射关系，便于后续处理。以JOIN关键词为例： <pre>JOIN<INNER>{ LEFT: \$_var: * RIGHT: \$r_val: * ON: \$cond: * }</pre> 其中LEFT: \$_var: *表示匹配的是JOIN的左子树，变量为\$_var，匹配的是左子树所有表达式，则左子树表达式和\$_var形成映射关系，即\$_var表示左子树的表达式，同理\$r_val表示匹配右子树的所有表达式，\$cond表示匹配JOIN所有的连接条件。	-
函数	函数是规则实现的基础，约束等复杂语义需要使用函数来表示，其包含的内容可参考表7-23和表7-24。 - 约束检查函数主要作用是检查匹配的表达式是否满足约束条件。 - 表达式计算函数则将变量作为输入，计算后输出的结果，会赋值到新声明的变量中记录下来。	- 函数及其入参需要使用‘（’‘）’进行分隔，否则会在解析时报错。 - 函数入参的变量，需要经过映射赋值，否则会在使用时报错。
约束	表示约束检查，当有多个约束条件时，约束函数通过'&&'进行拼接，如图7-35所示。满足所有约束时表示条件满足，匹配成功。每一个算子都可以有自己的约束。约束一般由约束函数构成，约束函数请参见表7-23。	-

图 7-35 约束表达

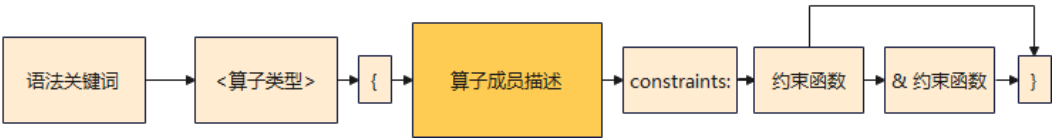


表 7-20 SQL 结构关键字

关键字名称	模板语法	支持类型	解释
QUERY	<pre>QUERY<QUERY_TYPE> { PREPROCESS: (\$PREPROCESS) CTE: (\$cte_expr: *) PROJECT: (\$proj_expr: *) FROM: (\$from_expr: *) FILTER: (\$filter_expr: *) GROUP: GROUP<> {\$group_expr: *} SORT: (\$sort_expr: *) LIMIT: LIMIT<> {\$limit_expr: *} POSTPROCESS: (\$POSTPROCESS) MAINBLOCK: false }</pre>	• QUERY_TYPE: 支持的SQL命令类型, 当前可选值包括: SELECT、DELETE、UPDATE和INSERT。 • 当未显式指定关键字类型时, 表示使用默认类型, 匹配所有当前支持的DML类型。作为匹配关键字时, 支持多个类型的组合, 不同类型间使用' '分隔, 例如: SELECT DELETE。	该关键字用于匹配一个完整的QUERY或SUBQUERY结构, 其中固有参数如下 (以下参数的顺序固定, 需要注意的是, 所有参数均支持省略, 表示不匹配对应参数或执行对应操作): • PREPROCESS: 用于声明构建时执行的前处理操作, 例如: /* 从\$pro中提取\$j表的属性列, 将结果与\$attr映射, 供后续使用 */ PREPROCESS: \$attr:(find_attributes \$j \$pro) • CTE: 匹配CTE。 • PROJECT: 匹配投影列。 • FROM: 匹配FROM表达式, 具体可以是普通表、视图、子查询、JOIN等类型。 • FILTER: 匹配过滤条件。 • GROUP: 匹配GROUP BY条件和HAVING条件。 • SORT: 匹配ORDER BY条件。 • LIMIT: 匹配LIMIT和OFFSET条件。 • POSTPROCESS: 用于声明构建时执行的后处理操作, 例如: /* 删除THIS指代的当前QUERY BLOCK无用的表信息 */ POSTPROCESS: (remove_unused_table THIS) • MAINBLOCK: 用于标识是否为最外层queryblock。

关键字名称	模板语法	支持类型	解释
JOIN	<pre>JOIN<JOIN_TYPE> { ALIAS: (\$alia: *) LEFT: (\$left_expr: *) RIGHT: (\$right_expr: *) ON: (\$on_expr: *) ONLY_TOP: false RECURSE_MODE(\$MODE) }</pre>	- JOIN_TYPE: 支持LEFT JOIN、RIGHT JOIN、INNER JOIN、CROSS JOIN、SEMI JOIN、ANTI JOIN、FULL JOIN几种连接类型。其中对应的可选值如下： LEFT（LEFT JOIN和RIGHT JOIN统一使用该类型描述）、INNER、CROSS、SEMI、ANTI、FULL。 - 该关键字无默认类型，需显式指定类型，否则解析报错。	该关键字用于匹配JOIN连接的内容，其中固有参数如下： - ALIAS: JOIN连接的别名。 - LEFT: JOIN连接中的左表达式。 - RIGHT: JOIN连接中的右表达式，当JOIN_TYPE为CROSS时，需要省略该参数。 - ON: JOIN连接中的连接条件，可省略，表示不进行该参数匹配。 - ONLY_TOP: 对于多个JOIN连接的场景，仅匹配第一个JOIN节点，不进行递归匹配。其取值为true或false，可省略，省略时使用默认值false。 - RECURSE_MODE: 递归模式，支持CURRENTFIRST（前序遍历）、CHILDFIRST（后序遍历）、CURRENTONLY（仅匹配顶层节点）三种方式，可省略，省略时使用默认值CURRENTFIRST。
GROUP	<pre>GROUP<> { GROUP: (\$group_expr: *) FILTER: (\$filter_expr: *) GROUPSET: (\$groupset_expr: *) }</pre>	无类型信息。	该关键字用于匹配一个GROUP BY结构，其中固有参数如下： - GROUP: 匹配GROUP BY条件。 - FILTER: 匹配HAVING条件，可省略，表示不进行该参数匹配。 - GROUPSET: 匹配GROUPSET条件，可省略，表示不进行该参数匹配。
LIMIT	<pre>LIMIT<> { LIMIT: (\$limit_expr: *) OFFSET: (\$offset_expr: *) }</pre>	无类型信息。	该关键字用于匹配一个LIMIT结构，其中固有参数如下： - LIMIT: 匹配LIMIT条件，可省略，表示不进行该参数匹配。 - OFFSET: 匹配OFFSET条件，可省略，表示不进行该参数匹配。

关键字名称	模板语法	支持类型	解释
AGG	<pre>AGG<AGG_TYPE> { ARGS: (\$exprs: *) DISTINCT: * }</pre>	- AGG_TYPE: 支持匹配各类系统内置聚集函数，支持同时指定多个值，使用' '分隔。 - 该关键字类型不显式指定时，表示使用默认类型，匹配所有AGG类型。	该关键字支持匹配所有AGG函数，并且可以指定type类型匹配具体的AGG函数，如：AGG<>表示匹配所有AGG函数；AGG<MIN MAX>表示匹配min和max类型的AGG函数，其中固有参数如下： - AGG_TYPE: AGG函数类型，如包含多个则使用' '进行分隔。AGG函数范围为PG_PROC系统表中存在的聚集函数。 - ARGS: AGG函数的入参。 - DISTINCT: 创建AGG函数时用于指定是否添加DISTINCT约束。取值为TRUE或FALSE，默认为FALSE。 说明 AGG关键字匹配支持省略ARGS参数，当该参数省略时，其他参数也必须省略，如下两个场景所示： - 指定AGG_TYPE，匹配AGG_TYPE对应的聚集函数 /* 匹配MIN、MAX及SUM这三种类型的聚集函数 */ AGG<MIN MAX SUM> {} - AGG_TYPE省略，匹配所有的聚集函数 AGG<> {}

关键字名称	模板语法	支持类型	解释
SUBLINK	<pre>SUBLINK<SUBLINK_TYPE> { TEST_EXPR: (\$test: *) QUERY: (\$exprs: *) }</pre>	• SUBLINK_TYPE： 支持 ANYSUBLINK、 NOTANYSUBLINK、 ALLSUBLINK、 NOTALLSUBLINK、 EXISTSSUBLINK、 NOTEXISTSSUBLINK、 EXPRSUBLINK、 ROWCOMPARESUBLINK类型。 • 该关键字无默认类型，需显式指定类型，否则解析报错。	该关键字用于匹配一个SUBLINK结构，其中固有参数如下： • QUERY：匹配QUERY结构。 • TEST_EXPR：匹配TEST表达式，仅ANYSUBLINK、NOTANYSUBLINK、ALLSUBLINK、NOTALLSUBLINK、ROWCOMPARESUBLINK类型有效。
CROSS	<pre>CROSS<> { ARGS: (\$exprs: *) }</pre>	无类型信息。	该关键字用于匹配一个笛卡尔积，其中固有参数如下： ARGS：笛卡尔积的表信息。

关键字名称	模板语法	支持类型	解释
SET	<pre>SET<SET_TYPE> { LEFT: (\$left_expr: *) RIGHT: (\$right_expr: *) MATCHTYPE_ALLSAME : * IGNOREUNMATCHTYPE E: * }</pre>	- SET_TYPE: 支持UNION、UNIONALL、INTERSECT、INTERSECTALL、EXCEPT、EXCEPTALL类型。 - 作为匹配关键字时，支持多个类型的组合，不同类型间使用' '分隔，例如：UNION UNIONALL。	该关键字用于匹配一个集合，其中固有参数如下： - LEFT: SET关键字的左表达式。 - RIGHT: SET关键字的右表达式。 - MATCHTYPE_ALLSAME: 用于标记是否检查集合中所有操作类型的一致性。当该参数设置为TRUE时，要求集合内所有操作类型必须一致。此参数为可选参数，默认值为TRUE。 - IGNOREUNMATCHTYPE: 用于标记是否忽略集合中不支持的操作类型。当该参数设置为TRUE时，允许集合中包含除匹配类型之外的其他操作类型。此参数为可选参数，默认值为FALSE。 说明 SET关键字匹配支持使用通用模板匹配模式，该模式下可以省略RIGHT关键字及其表达式信息。当存在多个子查询时，该模式可一次性匹配所有满足LEFT关键字中表达式的节点，现固定写法如下 <pre>(MATCHTYPE_ALLSAME采用默认值TRUE, IGNOREUNMATCHTYPE采用默认值FALSE) : SET<xxx> { LEFT: \$set_left : QUERY<xxx> {xxx} }</pre> 后续可以与FOREACH_NODES_WITH_TEMPLATE函数结合使用，使用方法如下： <pre>FOREACH_NODES_WITH_TEMPLATE \$set_left (xxx_func)</pre>

关键字名称	模板语法	支持类型	解释
PROJECT	<pre>PROJECT<> { ARGS: (\$exprs: *) MATCHALL: * }</pre>	无类型信息。	该关键字用于匹配投影列信息，其中固有参数如下： • ARGS：投影列表达式。 • MATCHALL：用于标识是否返回全部匹配的投影列。当该参数设置为TRUE时，返回所有与投影类表达式匹配的投影列；设置为FALSE时，仅返回首个匹配的投影列。此参数为可选参数，默认值为TRUE。 说明 PROJECT关键字的固定用法如下，其返回值为批量匹配的投影列信息，MATCHALL省略使用默认值TRUE： \$proj:PROJECT<> { ARGS: \$exprs: * } 后续可以与FOREACH_NODES_WITH_TEMPLATE函数结合使用，使用方法如下： FOREACH_NODES_WITH_TEMPLATE \$proj (xxx_func)
CTE	<pre>CTE<\$name:*> { QUERY: \$CTE_QUERY }</pre>	无类型信息。	该关键字用于匹配CTE结构，固有参数如下： • name: CTE名。 • QUERY：匹配QUERY结构。

表 7-21 表达式扩展关键字

关键字名称	模板语法	支持类型	解释
OR	<pre>OR<> { ARGS: (\$exprs: *) }</pre>	无类型信息。	该关键字用于匹配一个OR表达式结构，其中固有参数如下： ARGS：OR表达式的内容。
AND	<pre>AND<> { ARGS: (\$exprs: *) }</pre>	无类型信息。	该关键字用于匹配一个AND表达式结构，其中固有参数如下： ARGS：AND表达式的内容。

关键字名称	模板语法	支持类型	解释
NOT	NOT<> { ARGS: (\$exprs: *) }	无类型信息。	该关键字用于匹配一个NOT表达式结构，其中固有参数如下： ARGS: NOT表达式的内容。
BOOLEAN	BOOLEAN<BOOLEAN_TYPE> { ARGS: (\$exprs: *) IGNORENULL: * AUTOMATCH: * }	- BOOLEAN_TYPE: 支持AND、OR、NOT三个可选值，支持使用' '进行组合。 - 当未显式指定关键字类型时，表示使用默认类型，匹配所有的BOOL类型。	该关键字用于匹配一个BOOL类型表达式结构，其中固有参数如下： - ARGS: BOOL类型表达式的内容。 - IGNORENULL: 当被匹配的表达式为空时，是否视为合法。TRUE表示合法，FALSE表示不合法，默认值：FALSE，可省略，省略时使用默认值FALSE。 - AUTOMATCH: 是否从给定的表达式中自动匹配BOOL表达式。TRUE表示自动匹配给定条件中包含的BOOL表达式；FALSE表示给定的表达式必须为一个完整的BOOL表达式，默认值：FALSE，可省略，省略时使用默认值FALSE。
CONST	CONST<CONST_TYPE> { ARGS: (\$exprs: *) }	- CONST_TYPE: 支持INTEGER，BOOL类型。 - 该关键字类型不显式指定时，表示使用默认类型，匹配INTEGER类型。	该关键字用于匹配CONST表达式内容，其中固有参数如下： ARGS: CONST表达式的内容，可为空。
NULLTEST	NULLTEST<NULLTEST_TYPE> { ARGS: (\$exprs: *) }	- NULLTEST_TYPE: 支持ISNULL，NOTNULL类型。 - 该关键字无默认类型，需显式指定类型，否则解析报错。	该关键字用于匹配一个ISNULL或NOTNULL表达式，其中固有参数如下： ARGS: NULLTEST表达式的内容。

关键字名称	模板语法	支持类型	解释
OPERATOR	<pre>OPERATOR<OPERATOR_TYPE>{ LEFT: (\$left_expr: *) RIGHT: (\$right_expr: *) IGNORESIDE: true }</pre>	- OPERATOR_TYPE: 支持EQ, NE, GT, GE, LT, LE, PLUS类型。支持多个类型的组合, 不同类型间使用' '分隔。 - 该关键字无默认类型, 需显式指定类型, 否则解析报错。	该关键字用于匹配一个OPERATOR表达式, 其中固有参数如下: - LEFT: OPERATOR表达式的左表达式。 - RIGHT: OPERATOR表达式的右表达式。 - IGNORESIDE: 是否忽略左右节点的顺序, 可省略, 省略时使用默认值TRUE。
CASE	<pre>CASE<>{ ARGS: (\$expr: *) DEFRESULT: (\$def: *) }</pre>	无类型信息。	该关键字用于匹配一个CASE表达式, 其中固有参数如下: - ARGS: 匹配的表达式。 - DEFRESULT: 默认结果表达式。
WHEN	<pre>WHEN<>{ ARGS: (\$expr: *) DEFRESULT: (\$def: *) }</pre>	无类型信息。	该关键字用于匹配一个WHEN表达式, 其中固有参数如下: - ARGS: 匹配的WHEN条件。 - DEFRESULT: 返回结果表达式。
COL	<pre>COL<> { ARGS: (\$exprs: *) }</pre>	无类型信息。	该关键字用于匹配一个表的属性列表表达式, 其中固有参数如下: ARGS: COL表达式的内容。

表 7-22 补充关键字

关键字名称	模板语法	解释
CONSTRAINTS	<pre>CONSTRAINTS: /* CONTENT */</pre>	该关键字用于约束结构关键字或表达式扩展关键字, 可以作为表7-20及表7-21包含关键字的最后一个入参使用。示例如下: <pre>/* 约束QUERY关键字包含LIMIT子句 */ QUERY<> { PROJECT: \$proj:* FILTER: \$sub_filter:* CONSTRAINTS: (check_has_constraint THIS LIMIT) }</pre>

关键字名称	模板语法	解释
TEMPLATE	TEMPLATE { /* CONTENT */ }	该关键字用于约束函数或表达式计算函数的入参，其内容不可以为空，内容可以为结构关键字或表达式扩展关键字，当函数实现需要使用模板而非其构造成的表达式时，可以使用该关键字，各函数支持的关键字参见各函数说明。示例如下： <pre>/* find_nodes_with_template使用模板template {DSLSTRUCT}匹配\$redundant_info中的相关内容，并将结果映射到\$r_info_template，供后续使用，如果DSLSTRUCT不使用TEMPLATE关键字，则会被解析为一个表达式而不是模板 */ \$r_info_template: (find_nodes_with_template \$redundant_info template {DSLSTRUCT})</pre>
THIS	THIS	该关键字用于约束函数或表达式计算函数的入参，指代当前所属范围的QUERY BLOCK。示例如下： <pre>/* 删除THIS指代的当前QUERY BLOCK无用的表信息 */ POSTPROCESS: (remove_unused_table THIS)</pre>
ROOT	ROOT	该关键字用于约束函数或表达式计算函数的入参，指代第一层的QUERY BLOCK。示例如下： <pre>/* 约束ROOT指代第一层QUERY BLOCK包含LIMIT子句 */ CONSTRAINTS: (check_has_constraint ROOT LIMIT)</pre>

表 7-23 约束函数

函数名	函数原型	参数说明	功能描述
CHECK_HAS_CONSTRANT	bool check_has_constraint(DSLNode \$args, DSLType \$flags)	• \$args: 类型：DSLNode 描述：待检查的 DSLNode 节点（对应内部实现为 QueryTree 或者 Node 节点）。 • \$flags: 描述：支持检查的类型标志（仅检查是否存在）如下： VOLATILE（v 函数）、STABLE（s 函数）、IMMUTE（i 函数）、AGG（AGG 函数）、WINAGG（窗口函数）、DISTINCT（DISTINCT 关键字）、DISTINCT_ON（DISTINCT ON 关键字）、ROWNUM（有 ROWNUM 关键字）、MODIFYCTE（UPDATE 等操作类型的 CTE 表）、LIMIT（LIMIT 约束）、OFFSET（LIMIT OFFSET 约束）、SET（集合类型）、GROUP（GROUP BY 分组）、GROUPSET（GROUPING SETS 类型）、HAVING（Having 条件）、ROWMARK（FOR Update 等有行锁的场景）、CTE（存在 CTE 列表）、RETURNSET（返回多列的函数类	检查节点中是否包含待检查的 Flags 标志，根据返回结果判断是否满足约束条件。

函数名	函数原型	参数说明	功能描述
		型)、SUBLINK（存在子链接）、PLACEHOLDERVAR（存在PLACEHOLDERVAR）、UNPIVOT（存在行列转换）、UPPERVAR（存在上层参数，一般用于判断是否相关子链接）、SORT（存在ORDER BY排序）、RECURSIVE_CTE（存在递归CTE）、FULLGROUP（表示投影列中表属性列必须在Group中）。	
CHECK_IS_SINGLE_TABLE	bool check_is_single_table(DSLNode \$args)	\$args: 类型: DSLNode 描述: 待检查的DSLNode节点。	检查节点中是否为单表。

函数名	函数原型	参数说明	功能描述
CHECK_IS_EQUAL	bool check_is_equal(DSLNode \$args1, DSLNode \$args2, DSLType \$flags)	• \$args1: 类型: DSLNode 描述: 待等值检查的DSLNode节点1。 • \$args2: 类型: DSLNode 描述: 待等值检查的DSLNode节点2。 • \$flags: 类型: DSLType 描述: 可以选择 STRICT或者 PHYSIC, 该参数可以省略, 省略时使用默认值 STRICT。 - 当检查类型为 STRICT时, 待检查的节点相等的要求是所有成员变量均相等。 - 当检查类型为 PHYSIC时, 当前支持VAR进行该类型的等值比较, 需要满足参数来自的表类型相同, 表位置相同, 参数类型相同即可判断为结构相同。	检查节点1和节点2是否满足等值条件。
CHECK_IS_SUBSET	bool check_is_subset(List \$sub, List \$parent)	• \$sub: 类型: List 描述: 待检查的类型集合。 • \$parent: 类型: List 描述: 用于检查的父集合。	判断一个集合是否为另一个集合的子集。

函数名	函数原型	参数说明	功能描述
CONTAIN_ATTR_OF	bool contain_attr_of(List \$args, List \$tables)	\$args: 类型：List 描述：待检查的属性列集合。 \$tables: 类型：List 描述：指定的表节点，可以通过 FIND_TABLES 获取。	判断是否包含指定表的属性列。

函数名	函数原型	参数说明	功能描述
CHECK_TYPE_FOREACH	bool check_type_foreach(List \$arg, DSLType \$check_type, DSLType \$any_all_type)	• \$arg: 类型：List/ DSLNode 描述：待检查的表达式集合。 • \$check_type: 类型：DSLType 描述：支持的表达式类型：AND（AND表达式）、OR（OR表达式）、NOT（NOT表达式）、CONST（常数类型）、COUNT_STAR（COUNT(*)表达式）、SUBQUERY（子查询）、NOTNULL（属性列NOTNULL属性）、COL（为属性列）。 • \$any_all_type: 类型：DSLType 描述：可以选择ALL或者ANY，该参数可以省略，省略时使用默认值ALL。 - ALL，表示所有待检查项均为表达式类型。 - ANY，表示有一个检查项为表达式类型即可。	检验待检查表达式类型。
CHECK_IS_EMPTY	bool check_is_empty(DSLNode \$arg)	\$arg: 类型：DSLNode 描述：待检查的表达式。	判断表达式是否为空。

函数名	函数原型	参数说明	功能描述
CHECK_CONSTRAINT_FOREACH	bool check_constraint_foreach(List \$args, DSLType \$flags)	• \$args: 类型：List 描述：待检查的表集合，可以通过 FIND_TABLES 获取。 • \$flags: 描述：支持检查的类型标志（仅检查是否存在）如下： WINAGG（窗口函数）、DISTINCT（DISTINCT关键字）、DISTINCT_ON（DISTINCT ON关键字）、MODIFYCTE（UPDATE等操作类型的CTE表）、LIMIT（LIMIT约束）、OFFSET（LIMIT OFFSET约束）、SET（集合类型）、GROUP（GROUP BY分组）、GROUPSET（GROUPING SETS类型）、HAVING（Having条件）、ROWMARK（FOR Update等有行锁的场景）、UNPIVOT（存在行列转换）、SORT（存在ORDER BY排序）。	检查所有表包含待检查的Flags标志。

函数名	函数原型	参数说明	功能描述
CHECK_IS_SAME_TABLE	bool check_is_same_table(List \$args1, List \$args2)	• \$args1: 类型：List 描述：待检查的表集合1，可以通过 FIND_TABLES 获取。 • \$args2: 类型：List 描述：待检查的表集合2，可以通过 FIND_TABLES 获取。	检查表集合是否相等。
CHECK_IS_SIMPLE_COLUMN	bool check_is_simple_column(List \$args1, List \$dsl_simple_col_type_list, DSLType \$any_all_type)	• \$args1: 类型：List 描述：待检查的节点集合。 • \$dsl_simple_col_type_list: 类型：List 描述：代表允许通过检查的节点类型。 可选值：COL、CONST、DSLEXPR、IGNOREAGG，支持使用' ' 进行组合。 • \$any_all_type: 类型：DSLType 描述：用于指定在检查满足条件的结构体时的行为。 可选值：ALL、ANY。 - ALL，表示所有待检查项均符合条件时返回true。 - ANY，表示存在一个检查项符合条件时即可返回true。	检查列表中的节点是否为指定节点类型： • COL（表的单个属性列）。 • CONST（单个常量值）。 • EXPR（单个简单表达式，如t.a = 1）。 • IGNOREAGG（忽略AGG表达式）。

表 7-24 表达式计算函数

函数名	函数原型	参数说明	功能描述
EXPR_COMPUTE	Node expr_compute(Node \$args)	- 入参： \$args： 类型：Node 描述：QueryTree 中的表达式。 - 出参： \$result： 类型：Node 描述：经过常量计 算后的表达式。	常量表达式计算，例 如将提取的表达式中 可以计算为常量的表 达式进行预处理，方 便后续逻辑分析。 例如，将表达式中的 1+1进行提前计算， 得到常量值2；或者 将1= 0进行提前计 算，得到常量值 false。
BUILD_PRO JECT	Node build_project(Node \$args)	- 入参： \$args： 类型：Node 描述：表的属性 列，可以通过 FIND_ATTRIBUTES 等函数获取。 - 出参： \$result： 类型：Node 描述：构造好的投 影列集合。	通过传入的表属性 列，构造对应的投影 列集合。
BUILD_TAB LE	Node build_table(Node \$args)	- 入参： \$args： 类型：QUERY/ Node 描述：QUERY结构 体或FIND_TABLES 的返回值。 - 出参： \$result： 类型：Node 描述：新构造的 表。	根据入参构造新的 表。

函数名	函数原型	参数说明	功能描述
BUILD_BOOL_EXPR	Node build_bool_expr(DSLType \$bool_type, Node \$args1, DSLNode \$args2)	- 入参： \$bool_type： 类型：DSLType 描述：可以选择AND或者OR，分别表示构造AND表达式或OR表达式。 \$args1： 类型：Node 描述：表达式的参数1，一般为一个链表，该函数使用该链表中的元素构造布尔表达式。 \$args2： 类型：DSLNode 描述：表达式的参数2，构造表达式使用的TEMPLATE，该函数支持的TEMPLATE节点为：NULLTEST。 当该参数为空时，表示直接使用\$args1中的元素进行布尔表达式的构造，如： /* 使用\$v_list中的元素构造AND表达式 */ (build_bool_expr AND \$v_list) 当该参数不为空时，\$args1中的元素先使用\$args2的TEMPLATE构造表达式，再使用该表达式构造最终的布尔表达式，如： /* 使用\$v_list中的元素构造ISNULL表达式,再使用构造的表达式构造AND表达式 */ (build_bool_expr	构造指定类型的布尔表达式。

函数名	函数原型	参数说明	功能描述
		<pre>AND \$v_list TEMPLATE {NULLTEST<ISNULL >{args: ELEMENT}})</pre> 出参： \$result： 类型：Node 描述：构造的表达式。	
BUILD_GROUP	Node build_group(Node \$args)	- 入参： \$args： 类型：Node 描述：表的属性列，可以通过 FIND_ATTRIBUTES 等函数获取。 - 出参： \$result： 类型：Node 描述：构造好的分组集合。	根据输入表属性列构造分组信息。
BUILD_FROM_EXPR_BY_QUERY	Node build_fromexpr_by_query (Query \$query, Node \$table)	- 入参： \$query： 类型：Query 描述：构建的表结构所属的 Query 结构，该结构包含表构造的信息。 \$table： 类型：Node 描述：需要创建的表，通常为原始表结构的一部分，通过匹配获取。 - 出参： \$result： 类型：Node 描述：构造的目标表结构表达式。	根据传入的表创建 Query 结构体的 From 表达式。

函数名	函数原型	参数说明	功能描述
REMOVE_DISTINCT	bool remove_distinct (DSLEExpr \$query)	- 入参： \$args： 类型：DSLEExpr/ Query 描述：可以选择 THIS、ROOT或 Query结构。 - THIS：表示当前 层的QUERY。 - ROOT：表示根 QUERY。 - QUERY：表示目 标QUERY。 - 出参： \$result： 类型：bool 描述：TRUE表示删 除成功，反之 FALSE表示失败。	删除目标QUERY的 DISTINCT部分。
REMOVE_UNUSED_TABLE	bool remove_unused_table (DSLEExpr \$query)	- 入参： \$args： 类型：DSLEExpr/ Query 描述：可以选择 THIS、ROOT或 Query结构。 - THIS：表示当前 层的QUERY。 - ROOT：表示根 QUERY。 - QUERY：表示目 标QUERY。 - 出参： \$result： 类型：bool 描述：TRUE表示删 除成功，反之 FALSE表示失败。	删除目标QUERY的无 用表，当对QUERY的 FROM部分进行修改 或重新创建时，需要 执行该函数。

函数名	函数原型	参数说明	功能描述
FIND_UNIQUE_ATTR	List find_unique_attr(List \$args, UniqueType \$type)	- 入参： \$args： 类型：List 描述：表属性列集合。 \$type： 类型：UniqueType 描述：唯一键类型，支持 NotNullUnique（非空唯一索引）和 Unique（唯一索引）。 - 出参： \$result： 类型：List 描述：返回满足条件的属性值，如果有多个（如：唯一索引）则仅返回匹配到的第一个约束。	检查表基表是否包含唯一键。

函数名	函数原型	参数说明	功能描述
FIND_TABLES	List find_tables(DSLNode \$args, TableType \$type, DSLType \$include_join)	- 入参： \$args： 类型：DSLNode 描述：待提取的原始节点信息。 \$type： 类型：TableType 描述：要提取的表类型。 - ALL：所有表类型。 - BASEREL：基表。 - QUERY：子查询。 - JOIN：JOIN节点的表。 - FUNCTION：函数表。 - VALUES：VALUES表。 - CTE：CTE表。 - TBLRESULT：RESULT类型的表。 - XMLTABLE：XML表。 \$include_join： 类型：DSLType 描述：是否包含节点上层JOIN节点。 INCLUDEJOIN：包含节点上层JOIN节点。 NOTINCLUDEJOIN：不包含节点上层JOIN节点。 可以忽略，忽略时使用默认值NOTINCLUDEJOIN。	从节点中提取所有的Table表。

函数名	函数原型	参数说明	功能描述
		出参： \$result： 类型：List 描述：返回表信息。	

函数名	函数原型	参数说明	功能描述
FIND_ATTRIBUTES	List find_attributes(DSLNode \$tablelist, DSLNode \$args) List find_attributes(DSLNode \$tablelist, DSLNode \$args, DSLNode \$table_query, DSLNode \$find_attri_marks, DSLNode \$do_dedup)	- 入参： \$tablelist: 类型：DSLNode 描述：目标提取表的集合，可以通过find_tables函数进行获取。 \$args: 类型：DSLNode 描述：待提取的原始节点信息。 \$table_query 类型：DSLNode 描述：表集合所属的QUERY。 可选值：THIS（当前层QUERY）、ROOT（主查询QUERY）、变量（以\$开头，在匹配阶段获取的变量）。 \$find_attri_marks 类型：DSLNode 描述：提取的列需要满足的属性限制。 默认值：ALL。 可选值：ALL（无属性要求）、NOTNULL（非空属性）、UNIQUE（UNIQUE属性）及其组合。 \$do_dedup 类型：DSLNode 描述：是否对提取的结果进行去重。 默认值：DEDUPLICATE。	从节点中提取相关表属性。

函数名	函数原型	参数说明	功能描述
		可选值： DEDUPLICATE（去重）、 NOTDEDUPLICATE（不去重）。 出参： \$result： 类型：List 描述：返回表信息。	
APPEND	List append(List \$arg1, List \$arg2)	- 入参： \$arg1： 类型：List/ Node 描述：列表类型节点。 \$arg2： 类型：List/ Node 描述：列表或单个节点类型。 - 出参： 类型：List 描述：列表或单个节点类型。	在一个List中追加另一个节点或List，或者将两个节点构造造成一个List。

函数名	函数原型	参数说明	功能描述
REPLACE_WITH	<pre>Node replace_with(DSLNode \$arg1, DSLNode \$arg2, DSLNode \$arg3) bool replace_with(DSLNode \$arg1, DSLNode \$arg2, DSLNode \$arg3, DSLNode \$replace_node_option) bool replace_with(DSLNode \$arg1, DSLNode \$arg2, DSLNode \$arg3, DSLNode \$replace_node_option, DSLNode \$replace_node_mode)</pre>	- 入参： \$arg1: 类型：DSLNode 描述：被替换节点的父节点。 \$arg2: 类型：DSLNode 描述：被替换的节点。 \$arg3: 类型：DSLNode 描述：替换的内容节点。 \$replace_node_option 类型：DSLNode 描述：新节点替换旧节点时的替换方式。 默认值：ASNEW。 可选值：DIRECT（浅拷贝）、ASNEW（深拷贝），该参数可以省略，省略时使用默认值ASNEW。 \$replace_node_mode 类型：DSLNode 描述：遍历被替换结构体时的行为，具体参见可选值中的描述。 默认值：ANY。 可选值：ANY（仅替换第一个满足条件的节点）、ALL（替换所有满足条件的节点），该参数可以省略，省略时使用默认值ANY。 - 出参：	用\$arg3节点替换\$arg1父节点中的\$arg2节点。

函数名	函数原型	参数说明	功能描述
		类型：Node 描述：替换后的节点。	
REPLACE_WITH_REC	Node replace_with_rec (DSLNode \$arg1, DSLNode \$arg2, DSLNode \$arg3) bool replace_with_rec (DSLNode \$arg1, DSLNode \$arg2, DSLNode \$arg3, DSLNode \$replace_node_option)	- 入参： \$arg1： 类型：DSLNode 描述：被替换节点的父节点。 \$arg2： 类型：DSLNode 描述：被替换的节点。 \$arg3： 类型：DSLNode 描述：替换的内容节点。 \$replace_node_option 类型：DSLNode 描述：新节点替换旧节点时的替换方式。 默认值：ASNEW。 可选值：DIRECT（浅拷贝）、ASNEW（深拷贝）、UPDATE_COL_REF（更新Var结构体中的varno属性），该参数可以省略，省略时使用默认值ASNEW。 - 出参： 类型：Node 描述：替换后的节点。	用\$arg3节点替换父节点中递归查找到的\$arg1节点。

函数名	函数原型	参数说明	功能描述
REPLACE_TABLE	Node replace_table (QUERY \$arg1, DSLNode \$arg2, DSLNode \$arg3, DSLNode \$arg4 = DIRECT)	- 入参： \$arg1： 类型：QUERY 描述：被替换节点的QUERY节点。 \$arg2： 类型：DSLNode 描述：被替换的表节点。 \$arg3： 类型：DSLNode 描述：替换的表节点。 \$arg4： 类型：DSLNode 描述：该参数为可选参数，表示替换表的方式。若未显式指定，默认值为DIRECT；若显式指定，取值仅可为ADD。 默认值：DIRECT，表示使用直接替换的方式。 可选值：ADD，表示新增节点替换。 - 出参： - 类型：Node 描述：替换后的节点。	使用\$arg4标识的方式，用\$arg3表节点替换\$arg1QUERY节点中的\$arg2表节点。

函数名	函数原型	参数说明	功能描述
REPLACE_TABLE_FOREACH	Node replace_table (QUERY \$arg1, DSLNode \$arg2, DSLNode \$arg3)	- 入参： \$arg1： 类型：QUERY 描述：被替换节点的QUERY节点。 \$arg2： 类型：List 描述：被替换的表节点列表。 \$arg3： 类型：DSLNode 描述：替换的表节点。 - 出参： 类型：Node 描述：替换后的节点。	用\$arg3表节点替换QUERY节点中的\$arg2集合中所有表节点。
FIND_EXPR_WITHOUT	List find_expr_without(Expr \$args1, Expr \$args2)	- 入参： \$args1： 类型：Expr 描述：待处理的父表达式节点。 \$args2： 类型：Expr 描述：要排除的表达式节点。 - 出参： 类型：List 描述：排除目标表达式后剩余的表达式节点列表。	从\$args1父表达式节点中查找不包含指定\$args2表达式节点的部分。

函数名	函数原型	参数说明	功能描述
FIND_EXPR_WITH	List find_expr_with(Expr \$args, DSLType \$flags, DSLType \$any_all_type)	- 入参： \$args： 类型：Expr 描述：待处理的表达式节点。 \$flags： 类型：DSLType 描述：支持包含的类型，如：UPPERVAR表示包含上层查询块的Var。 \$any_all_type： 类型：DSLType 描述：提取满足条件的表达式，可以选择ALL或者ANY。 ALL：提取所有满足条件的表达式。 ANY：提取第一个满足条件的表达式。 - 出参： 类型：List 描述：满足条件的节点。	从\$args表达式中查找包含指定条件的部分。
LIST_FIRST	Node list_first(List \$args)	- 入参： \$args： 类型：List 描述：待处理的列表。 - 出参： 类型：Node 描述：提取出的节点。	提取\$args列表的第一个节点。

函数名	函数原型	参数说明	功能描述
LIST_LENGTH	Integer list_length(List \$args)	- 入参： \$args： 类型：List 描述：待处理的列表。 - 出参： 类型：INTEGER 描述：列表的长度。	获取列表的长度。
PULL_RELA_ATTR	List pull_rela_attr(Expr \$args, DSLType \$flags)	- 入参： \$args： 类型：Expr 描述：待处理的表达式节点。 \$flags： 类型：DSLType 描述：提取指定层的表达式，可以选择THIS、PARENT或PARENTS。 THIS：提取当前层的表达式。 PARENT：提取上一层的表达式。 PARENTS：提取上两层的表达式。 - 出参： 类型：List 描述：满足条件的表达式列表。	提取表达式中满足层数条件的表达式节点。

函数名	函数原型	参数说明	功能描述
PULL_EXPR_TO_QUERY	Node pull_expr_to_query(List \$args, Query \$query)	- 入参： \$arg： 类型：Node 描述：待处理的表属性节点。 \$query： 类型：Query 描述：提取时的目标QUERY。 - 出参： 类型：Node 描述：提取后的节点。	将\$args表属性节点提取到\$query的节点中。
BUILD_EXPR_THROUGH_PULLUP	Node build_expr_through_pullup(Node \$args, Query \$query)	- 入参： \$args 类型：Node 描述：待处理的表达式节点。 \$query 类型：Query 描述：提取时的目标QUERY，可以为build_table的结果，表示新建Query。 - 出参： 类型：Node 描述：提取后的节点。	通过相关的表达式和新创建的表，构建新的表达式。其中表达式中的表属性与新的表属性映射。

函数名	函数原型	参数说明	功能描述
REMOVE_QUERY_OPTION	bool remove_query_option(Query \$query, DSLType \$remove_query_options)	- 入参： \$query： 类型：Query 描述：代表需要移除指定部分的Query结构体。 可选值：THIS、ROOT、Query结构。 - THIS：表示当前层的QUERY。 - ROOT：表示根QUERY。 - QUERY：表示目标QUERY。 \$remove_query_options 类型：DSLType 描述：代表Query结构体中需要被移除的部分。 可选值：DISTINCT、DISTINCT_ON、GROUP、SORT、LIMIT、OFFSET及其组合值。 - 出参： 类型：bool 描述：代表移除是否成功。	移除Query结构体中的指定部分。
BUILD_WHEN_EXPR	CaseWhen build_when_expr(CaseWhen \$casewhen)	- 入参： \$casewhen： 类型：CaseWhen 描述：待生成的CaseWhen结构模板。 - 出参： 类型：CaseWhen 描述：生成的CaseWhen结构。	生成CaseWhen结构体。

函数名	函数原型	参数说明	功能描述
BUILD_CASE_EXPR	CaseExpr build_case_expr(CaseExpr \$caseexpr)	- 入参： \$caseexpr： 类型：CaseExpr 描述：待生成的CaseExpr结构模板。 - 出参： 类型：CaseExpr 描述：生成的CaseExpr结构。	生成CaseExpr结构体。
FIND_SAME_TABLES	List find_same_tables(List \$args, DSLType \$type)	- 入参： \$args： 类型：List 描述：待处理的表的List。 \$type： 类型：DSLType 描述：待处理的表类型。 可选值： BASEREL（基表）。 \$table_check_options 类型：DSLType 描述：在函数处理过程中，对于特殊场景的处理方式。 可选值：空值（默认场景，按照默认方式处理）、SUPPORT_SPECIFIED_PARTITION（支持指定分区场景）。 - 出参： 类型：List 描述：相同表的Oid组成的链表。	查找相同的表，默认不支持指定分区场景。 说明 指定分区场景示例： SELECT * FROM t1 partition(p1), t2 partition(p2) WHERE t1.c1=t2.c1; 该示例中，使用partition关键字指定了特定的分区。

函数名	函数原型	参数说明	功能描述
FIND_NODES_WITH_TEMPLATE	List find_nodes_with_template(Node \$node, DSLNode \$template, DSLType \$rec_mode)	- 入参： \$node： 类型：Node 描述：待被遍历处理的结构体。 \$template： 类型：DSLNode 描述：待匹配的目标模板。使用TEMPLATE关键字，对目标模板进行描述，格式为TEMPLATE{FUNC_TEMPLATE_NODE}。 \$rec_mode： 类型：DSLType 描述：遍历处理的模式。 可选值： CURRENTFIRST（前序遍历）、 CHILDFIRST（后序遍历）、 CURRENTONLY（仅匹配顶层节点），该参数可以省略，表示非递归模式，当该参数不为空时，入参\$template仅支持为JOIN的模板。如： /* 以后序遍历的方式，使用JOIN模板在\$fr中进行匹配 */ find_nodes_with_template \$fr template {JOIN<INNER>{ LEFT: \$l : * RIGHT: \$r : * on: \$o : *}} CHILDFIRST - 出参： 类型：List 描述：与目标模板相匹配的结构体所组成的链表。	遍历结构体，查找与目标模板相匹配的结构。

函数名	函数原型	参数说明	功能描述
FOREACH_NODES_WITH_TEMPLATE	bool foreach_nodes_with_template(DSLNode \$arg1, DSLNode \$arg2)	- 入参： \$arg1： 类型： DSLNode，具体为DSLExpr，其内部以链表形式存储 find_nodes_with_template函数返回的结果。 描述：待处理的结构体的集合。 \$arg2： 类型：DSLNode 描述：对结构体的处理方式。 - 出参： 类型：bool 描述：函数执行是否成功。	遍历链表中的每一个元素，并以指定的方式对每一个元素进行处理。

函数名	函数原型	参数说明	功能描述
BUILD_SUB LINK	Sublink build_sublink(Node \$sublink) Sublink build_sublink(Node \$sublink, List \$columns, List \$operator)	- 入参： \$sublink： 类型：可以为 Sublink或使用 NOT类型的 BoolExpr嵌套的 Sublink。 描述：待构建的 SubLink结构体 模板。 \$columns： 类型：List 描述：在ANY、 ALL、 ROWCOMPARE 类型的Sublink 中，需要和 Sublink对比的 列。 \$operator： 类型：List 描述：需要在 ANY、ALL、 ROWCOMPARE 类型的Sublink 中执行的操作。 可选值：EQ、 NE、GT、GE、 LT、LE、 PLUS、ANY。 - 出参： - 类型：Sublink - 描述：构建的 Sublink结构 体。	构建Sublink结构 体。 - 对于ANY、ALL、 ROWCOMPARE 类型的 SUBLINK，使用 包含三个入参的 函数。 - 对于其他类型的 SUBLINK，使用 包含一个入参的 函数。

函数名	函数原型	参数说明	功能描述
DSLNODE_COPY	Node dslnode_copy(Node \$node, DSLType \$copy_node_option)	- 入参： \$node： 类型：Node 描述：待复制的 DSLNode 结构体。 \$copy_node_option： 类型：DSLType 描述：复制的方式。 可选值： DIRECT（浅拷贝）、ASNEW（深拷贝），该参数可以省略，省略时使用默认值 DIRECT。 - 出参： 类型：Node 描述：复制生成的 DSLNode 结构体。	复制 DSLNode 结构体。

函数名	函数原型	参数说明	功能描述
FIND_EQUALCLASS_VARS	List find_equalclass_vars(Bool Expr \$exprs, DSLNode \$bool_expr_type, DSLNode \$operator_type)	- 入参： \$exprs： 类型：由 OpExpr所组成 的集合，当前支持AND类型的 BoolExpr。 描述：生成等价类时所依据的表达式。 \$bool_expr_type： 类型：DSLNode 描述：在生成等价类时，支持的多个表达式之间的逻辑关系类型。 可选值：AND。 \$operator_type： 类型：DSLNode 描述：在生成等价类时，支持的OpExpr的类型。 可选值：EQ。 - 出参： 类型：二维List。其中每个子List代表一个等价类；子List中的元素为Var，这些Var之间存在等价类关系。 描述：等价类组成的链表。	根据表达式生成等价类。

函数名	函数原型	参数说明	功能描述
FIND_REDUNDANT_INFO	List find_redundant_info(List \$equalclass_list)	- 入参： \$equalclass_list： 类型：二维List。其中每个子List代表一个等价类；子List中的元素为Var，这些Var之间存在等价关系。 描述：生成冗余信息时所依据的等价类信息。该信息由find_equalclass_vars函数生成。 - 出参： 类型：List。其内部元素为RedundantInfo结构体，结构体包含必须表和冗余表等信息。 描述：根据等价类信息生成的冗余信息，用于在某些优化规则中消除冗余表。	根据等价类信息生成冗余信息，用于在某些优化规则中消除冗余表。

函数名	函数原型	参数说明	功能描述
REMOVE_USELESS_CONDITION	BoolExpr remove_useless_condition(Query \$query, BoolExpr \$expr, DSLType \$scenario_option, List \$vars)	- 入参： \$query： 类型：Query 描述：代表需要移除指定部分的Query结构体。 可选值：THIS、ROOT、QUERY结构。 - THIS：表示当前层的QUERY。 - ROOT：表示根QUERY。 - QUERY：表示目标QUERY。 \$expr 类型：BoolExpr 描述：代表需要被处理的谓词条件的集合。 \$scenario_option 类型：DSL_TYPE 描述：代表需要处理的场景。 可选值：SAME_VAR_EQUAL_OPEXPR（同一个Var相等的场景，例如：var1=var1）。 \$vars 类型：List 描述：代表被处理的var的List。 - 出参： 类型：BoolExpr 描述：代表移除冗余的谓词条件后的BoolExpr。	删除冗余的谓词条件。

7.15.2.2 DSL 规则示例与说明

说明

本章节内容为DSL规则匹配的示例与说明，执行结果不为实际执行结果。

规则示例场景描述

- 简要描述：两个包含Count聚集函数的表达式子链接进行等值比较时，如果其中一个子链接可以表示为另一个子链接的子结构，且过滤条件仅相差一个EXISTS子链接条件时，可以对其进行优化。因为在此场景下，实际是想找到EXISTS条件返回FALSE的情况，在这种情况下两个count值是否相等取决于其他条件是否为TRUE。当其他条件为TRUE时，根据AND条件的语义，两个子链接的过滤条件相反，count()取值一定不同；否则取值相同。因此，对于这种场景可以使用NOT EXISTS等价改写消减子链接的数量。

- SQL示例：

改写前：

```
SELECT t1.a,  
       t1.b  
FROM table1 as t1,  
     table2 as t2  
WHERE t1.a = t2.a  
      AND t1.b = t1.b  
      AND (  
          SELECT COUNT(*)  
            FROM table1 as t11  
            WHERE t11.a = t1.a  
              AND t11.b = t1.b  
        ) = (  
          SELECT COUNT(*)  
            FROM table1 as t21  
            WHERE t21.a = t1.a  
              AND t21.b = t1.b  
          AND EXISTS(  
              SELECT 1  
                FROM t3  
                WHERE t1.a = t3.a  
            )  
        )  
);
```

改写后：

```
SELECT t1.a,  
       t1.b  
FROM table1 as t1,  
     table2 as t2  
WHERE t1.a = t2.a  
      AND t1.b = t1.b  
      AND NOT EXISTS (  
          SELECT 1  
            FROM table1 as t11  
            WHERE t11.a = t1.a  
              AND t11.b = t1.b  
          AND NOT EXISTS(  
              SELECT 1  
                FROM t3  
                WHERE t1.a = t3.a  
            )  
        )  
);
```

规则示例场景 DSL 规则

```
QUERY <> {  
    FILTER: $exprs: * &&  
        (AND<>{
```

```

ARGS: $and_exp: * && ($seq_expr: OPERATOR<EQ>{
  LEFT: SUBLINK<EXPRSUBLINK>{
    QUERY: $ori_query: QUERY<>{
      PROJECT: $proj1: AGG<COUNT>{ARGS: $agg_args1: *}
      FROM: $input1: * && ($sub_l_t: (find_tables $input1 ALL))
      FILTER: $FILTER1: * && (AND<> {ARGS: $f_1: * && (OPERATOR<EQ>{
        LEFT: $f_1_l: *
        RIGHT: $f_1_r: *
        CONSTRAINTS: ((check_has_constraint $f_1_l UPPERVAR)
          && !(check_has_constraint $f_1_r UPPERVAR)) ||
          (!(check_has_constraint $f_1_l UPPERVAR)
            && (check_has_constraint $f_1_r UPPERVAR)) &&
          !(check_has_constraint THIS MODIFYCTE | VOLATILE)
        )))
      GROUP: GROUP<> {
        GROUP: $gr1: * && ($att_list: (find_attributes $gr1 THIS))
        FILTER: $h_c1: *
        GROUPSET: $gr_s1: *
      }
      SORT: $s1: *
    }
  }
  RIGHT: SUBLINK<EXPRSUBLINK>{
    QUERY: QUERY<>{
      PROJECT: $proj2: AGG<COUNT>{ARGS: $agg_args2: *}
      FROM: $input2: * && ($sub_r_t: (find_tables $input2 ALL))
      FILTER: $FILTER2: * && (AND<> {ARGS: $f_2: *
        && ($orther_f: SUBLINK<existssublink>{QUERY: $ssub_query: *}})
        && ($same_set: (find_expr_without $f_2 $orther_f))
      }
      GROUP: GROUP<> {
        GROUP: $gr2: *
        FILTER: $h_c2: *
        GROUPSET: $gr_s2: *
      }
      SORT: $s2: *
      CONSTRAINTS: (check_is_equal $same_set $f_1)
        && (check_is_equal $gr1 $gr2)
        && (check_is_equal $h_c1 $h_c2) && (check_is_equal $gr_s1 $gr_s2)
        && (check_is_equal $s1 $s2)
        && (((check_type_foreach $proj1 COUNT_STAR)
          && (check_type_foreach $proj2 COUNT_STAR)) ||
          ((check_type_foreach (expr_compute $agg_args1) CONST)
            && (check_type_foreach (expr_compute $agg_args2) CONST)) ||
          ((check_type_foreach $agg_args1 COL)
            && (check_type_foreach $agg_args2 COL)
            && (check_is_equal $agg_args1 $agg_args2)))
        && !(check_has_constraint THIS MODIFYCTE | VOLATILE)
    }
  }
  IGNORESIDE:true
})
}} && ($orther_expr: (find_expr_without $and_exp $seq_expr))
CONSTRAINTS: (check_has_constraint ROOT SUBLINK) &&
  (check_type_foreach $exprs AND)
  && (check_type_foreach $FILTER1 AND) && (check_type_foreach $FILTER2 AND) &&
  (check_is_same_table $sub_l_t $sub_r_t)
  && !(check_has_constraint ROOT MODIFYCTE | VOLATILE)
}
=>
QUERY <> {
  FILTER: (append $orther_expr NOT<>{
    ARGS: SUBLINK<existsSUBLINK>{
      QUERY: QUERY<SELECT>{
        PROJECT: CONST<INTEGER>{ARGS:1}
        FROM: (build_fromexpr_by_query $ori_query $input1)
        FILTER: AND <> { ARGS: (append $same_set NOT<>{
          ARGS: $orther_f
        })
      }
    }
  }
}

```

```
GROUP: GROUP<> {
  GROUP: (build_group $att_list)
  FILTER: $h_c1
  GROUPSET: $gr_s1
}
SORT: $s1
}
}
}
)
```

规则示例场景 DSL 规则详细解析

- 匹配部分：
 - QUERY<>：表示匹配的是默认类型的QUERY，即SELECT类型的QUERY。
 - FILTER<>：表示匹配的是约束条件。
 - ARGS: \$exprs: *：表示约束条件的所有内容映射到变量exprs上；&&表示进一步匹配。
 - AND<>：表示仅匹配AND条件。
 - ARGS: \$and_exp: *：表示将匹配的AND表达式的所有内容与变量and_exp做映射。ARGS: OPERATOR<EQ>：表示匹配等值表达式。
 - LEFT: SUBLINK<EXPRSUBLINK>、RIGHT: SUBLINK<EXPRSUBLINK>表示等值表达式的左右取值必须是子链接，且子链接中PROJECT: \$proj1: AGG<COUNT>{}：表示投影列必须是COUNT()。
 - ((check_has_constraint \$f_1_l UPPERVAR) && !(check_has_constraint \$f_1_r UPPERVAR)) ||
 (!(check_has_constraint \$f_1_l UPPERVAR) && (check_has_constraint \$f_1_r UPPERVAR))：约束检查等值表达式一侧包含上层的VAR，即包含相关性条件。
 - GROUP, SORT：匹配子链接中的GROUP及SORT部分。
 - AND<> {ARGS: \$f_2: * && (\$orther_f: SUBLINK<EXISTSSUBLINK>{QUERY: \$ssub_query: *})}：对于右SUBLINK在约束里的AND条件中存在一个EXISTSSUBLINK。
 - \$same_set: (find_expr_without \$f_2 \$orther_f)：表示提取\$f_2中排除\$orther_f的部分，映射到变量\$same_set上。
 - IGNORESIDE:true：表示左右两侧值可以互换进行模板匹配。
 - CONSTRAINTS: (check_is_equal \$same_set \$f_1)：表示两个子链接除右侧子链接的EXISTSSUBLINK外，其余部分都相同。
 - (\$orther_expr: (find_expr_without \$and_exp \$eq_expr))：同样表示从约束中提取非匹配的部分。
 - (((check_type_foreach \$proj1 COUNT_STAR) && (check_type_foreach \$proj2 COUNT_STAR)) || ((check_type_foreach (expr_compute \$agg_args1) CONST) && (check_type_foreach (expr_compute \$agg_args2) CONST)) || ((check_type_foreach \$agg_args1 COL) && (check_type_foreach \$agg_args2 COL) && (check_is_equal \$agg_args1 \$agg_args2)))：表示左右子链接的count()函数的参数只能为count(*)、count(常量)或count相同的列。
- 改写部分：
 - QUERY<>和原模板一致。

- Filter<>和原模板一致。
- ARGS: 表示成员变量；append是内部函数表达将两个表达式组成链表；append \$orther_expr NOT<>表示\$orther_expr所映射的表达式和NOT后续表示的表达式做组合。
- ARGS: SUBLINK<EXISTSSUBLINK>: Not下是一个EXISTS子链接。
- GROUP, SORT: 构建子链接的GROUP或SORT部分。
- QUERY: QUERY<SELECT> 表示子链接的QUERY成员为一个QUERY表达式。
- PROJECT: CONST<INTEGER>{ARGS:1}: 表示投影列为整数1。
- FROM: (build_from_expr_from_query \$ori_query \$input1): 表示使用函数build_from_expr_from_query构造FROM部分。
- FILTER: AND <> { ARGS : (append \$same_set NOT<>: {ARGS: \$orther_f} 表示FILTER是AND表达式，表达式内容由append函数将\$same_set所映射的表达式与\$orther_f映射表达式构造NOT后进行组合。

规则示例执行结果演示

```
EXPLAIN (costs off)SELECT t1.C, t1.b, t2.b FROM t1, t2 WHERE t1.a = t2.a AND (SELECT COUNT(*) FROM t1
as t11 WHERE t11.a = t1.a AND t11.b = t1.b) = (SELECT COUNT(*) FROM t1 AS t21 WHERE t21.a = t1.a
AND t21.b = t1.b AND EXISTS(SELECT 1 FROM t3 WHERE t1.a = t3.a));
NOTICE: Query with blockname(sel$1) is rewritten by Gs_Rewriter: SELECT t1.c, t1.b, t2.b FROM
dsl_exprsublink_elimination.t1, dsl_exprsublink_elimination.t2 WHERE t1.a = t2.a AND NOT (EXISTS (SELECT
1::bigint FROM dsl_exprsublink_elimination.t1 t11 WHERE t11.a = t1.a AND t11.b = t1.b AND NOT (EXISTS
(SELECT 1 AS "?column?" FROM dsl_exprsublink_elimination.t3 WHERE t1.a = t3.a))))); with rule_name
sublink_compare_elimination
```

QUERY PLAN

```
Hash Join
Hash Cond: (t2.a = t1.a), (Expression Flatten Optimized)
-> Seq Scan on t2
-> Hash
-> Merge Anti Join
Merge Cond: ((t1.a = t11.a) AND (t1.b = t11.b)), (Expression Flatten Optimized)
Join Filter: (NOT (alternatives: SubPlan 1 or hashed SubPlan 2)), (Expression Flatten Optimized)
-> Sort
Sort Key: t1.a, t1.b
-> Seq Scan on t1
-> Sort
Sort Key: t11.a, t11.b
-> Seq Scan on t1 t11
SubPlan 1
-> Seq Scan on t3
Filter: (t1.a = a), (Expression Flatten Optimized)
SubPlan 2
-> Seq Scan on t3
(18 rows)
```

7.15.3 DSL 规则引擎最佳实践

基于DSL的规则引擎的使用受GUC参数控制，所以需要打开相关的参数才可以生效（enable_dsl_ruleengine = ON）。

下面给出最佳实践样例。

📖 说明

最佳实践中系统函数的调用权限请查看以下章节的相关说明：

- 非透明多写特性：请参见《参考》中“SQL参考 > 函数和操作符 > DSL规则管理函数”章节。

- 场景一：拥有权限的合法用户可以通过调用dsl_add_rule的函数接口添加一条DSL规则。

```
gaussdb=# SELECT dsl_add_rule(
'compute_const',
'QUERY<> {FILTER: $expr: *}
=>
QUERY<> {FILTER: (expr_compute $expr)}
',
777,
true);
 dsl_add_rule
-----
compute_const
(1 row)

-- 加载新规则，使规则可以被应用
gaussdb=# SELECT dsl_reload_rule();
 dsl_reload_rule
-----
success
(1 row)
```

- 场景二：拥有权限的合法用户可以通过执行SQL语句查询系统表中支持的DSL规则。

```
-- 切换扩展显示模式，便于观察计划
gaussdb=# \x
Expanded display is on.

gaussdb=# SELECT * FROM GS_DSL_REWRITERULE WHERE rulename = 'compute_const';
-[ RECORD 1 ]-----
rulename      | compute_const
ruledesc      |
enabled       | t
externalrule  | t
costbased     | f
priority      | 777
dbcompatibility | 31
planmode      | 3
createtime    | xxxx-xx-xx xx:xx:xx.596976+08
lastupdatetime |
rulehash      | xxxxxxxxxxxxxxxxxxxx
ruletext      | QUERY<> {FILTER: $expr: *}
              | =>
              | QUERY<> {FILTER: (expr_compute $expr)}
              |
```

- 场景三：拥有权限的合法用户可以通过调用dsl_set_rule的函数接口修改系统表中的DSL规则。

```
-- 设置rule_name为compute_const的规则，priority属性为8
gaussdb=# SELECT dsl_set_rule('compute_const', 'priority', '778');
-[ RECORD 1 ]+--
dsl_set_rule | 1

gaussdb=# SELECT * FROM GS_DSL_REWRITERULE WHERE rulename = 'compute_const';
-[ RECORD 1 ]-----
rulename      | compute_const
ruledesc      |
enabled       | t
externalrule  | t
costbased     | f
priority      | 778 --规则优先级被设置为8
dbcompatibility | 31
planmode      | 3
createtime    | xxxx-xx-xx xx:xx:xx.596976+08
lastupdatetime | xxxx-xx-xx xx:xx:xx.822819+08
rulehash      | xxxxxxxxxxxxxxxxxxxx
ruletext      | QUERY<> {FILTER: $expr: *}
              | =>
```


- ```
| QUERY<> {FILTER: (expr_compute $expr)}
|
```
- 场景四：应用规则对满足条件的SQL进行查询改写。  
-- 设置开启使用DSL规则引擎  
gaussdb=# SET enable\_dsl\_ruleengine = ON;  
SET  
gaussdb=# SHOW enable\_dsl\_ruleengine;  
enable\_dsl\_ruleengine  
-----  
on  
(1 row)  
-- 设置开启使用DSL规则日志打印  
gaussdb=# set logging\_module = 'on(OPT\_DSL\_REWRITE)';  
SET  
  
-- 关闭扩展显示模式  
gaussdb=# \x  
Expanded display is off.  
  
gaussdb=# EXPLAIN(VERBOSE ON) SELECT \* FROM t1 WHERE t1.a = 1 OR TRUE;  
NOTICE: Query with blockname(sel\$1) is rewritten by Gs\_Rewriter: SELECT a FROM public.t1 WHERE  
true; with rule\_name compute\_const  
QUERY PLAN  
-----  
[Parameterized]  
Seq Scan on public.t1 (cost=0.00..34.02 rows=2402 width=4)  
Output: a  
(3 rows)
  - 场景五：拥有权限的合法用户可以通过调用dsl\_del\_rule接口删除一条已经存在DSL规则，参数为规则名。  
gaussdb=# SELECT dsl\_del\_rule('compute\_const');  
dsl\_del\_rule  
-----  
1  
(1 row)  
-- 删除缓存中的规则  
gaussdb=# SELECT dsl\_reload\_rule();  
dsl\_reload\_rule  
-----  
success  
(1 row)

## 7.15.4 常见问题处理

### ERROR: syntax error at 'XXXX'.

- 场景一：关键字使用错误示例。  
gaussdb=# SELECT dsl\_add\_rule(  
'compute\_const',  
'JOIN<> {ARGS: \$expr: \*}'  
=>  
FILTER<> {ARGS: (expr\_compute \$expr)  
}',  
3,  
true);  
ERROR: syntax error at 'JOIN' --  
LINE 3: 'JOIN<> {ARGS: \$expr: \*}'  
^  
CONTEXT: referenced column: dsl\_add\_rule  
  
根因分析：JOIN关键字使用错误，不能用于模板入口语法关键词，关键字使用请参考[语法关键词](#)语法规范。
- 场景二：函数未使用 ‘（’ ‘）’ 进行分隔错误场景示例。  
gaussdb=# SELECT dsl\_add\_rule(  
'compute\_const',

```
'FILTER<> {ARGS: $expr: *}
=>
FILTER<> {ARGS: expr_compute $expr
}',
3,
true);
ERROR: syntax error at 'expr_compute'
LINE 5: FILTER<> {ARGS: expr_compute $expr
 ^
CONTEXT: referenced column: dsl_add_rule
```

根因分析：函数关键字及其入参变量未使用 ‘（’ ‘）’ 进行分隔，函数使用请参考[函数](#)语法规范。

## ERROR: Duplicate key value violates unique constraint "gs\_dsl\_rewriterule\_rulename\_index".

场景：添加规则rulename重复错误示例。

```
-- 添加rulename为 'compute_const' 的规则
gaussdb=# SELECT dsl_add_rule(
'compute_const',
'FILTER<> {ARGS: $expr: *}
=>
FILTER<> {ARGS: (expr_compute $expr)
}',
3,
true);
 dsl_add_rule

compute_const
(1 row)

-- 再次添加rulename为 'compute_const' 的规则报错
gaussdb=# SELECT dsl_add_rule(
gaussdb=# 'compute_const',
gaussdb=# 'FILTER<> {ARGS: $expr: *}
gaussdb=# =>
gaussdb=# FILTER<> {ARGS: (expr_compute $expr)
gaussdb=# }',
gaussdb=# 3,
gaussdb=# true);
ERROR: Duplicate key value violates unique constraint "gs_dsl_rewriterule_rulename_index".
DETAIL: Key (rulename)=(compute_const) already exists.
CONTEXT: referenced column: dsl_add_rule
```

根因分析：rulename需全局唯一，相关约束请查看以下章节的相关说明：

- 非透明多写特性：请参见《参考》中“系统表和系统视图 > 系统表 > 基于DSL的规则引擎”章节。

## 7.16 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。

### 7.16.1 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《参考》中“SQL参考 > SQL语法 > C > 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。

#### 说明

- 为防止对服务端文件任意读，系统管理员及运维模式下运维管理员使用时会受enable\_copy\_server\_files及safe\_data\_path两个参数的控制。
- 当参数enable\_copy\_server\_files关闭时，只允许初始用户创建file\_fdw类型外表，打开时系统管理员及运维模式下运维管理员可以创建file\_fdw类型外表，以防止用户越权查看或修改敏感文件。
- 当参数enable\_copy\_server\_files打开时，管理员可以通过GUC参数safe\_data\_path设置用户导入导出的路径校验，文件必须为safe\_data\_path路径的子路径，未设置此GUC参数的时候（默认情况），不对用户使用的路径进行拦截。

使用file\_fdw创建的外部表可以有下列选项：

- 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，非必需参数，和COPY语句的FORMAT选项相同。
- header  
指定的文件是否有标题行，非必需参数，与COPY语句的HEADER选项相同。
- delimiter  
指定文件的分隔符，非必需参数，与COPY的DELIMITER选项相同。
- quote  
指定文件的引用字符，非必需参数，与COPY的QUOTE选项相同。
- escape  
指定文件的转义字符，非必需参数，与COPY的ESCAPE选项相同。
- null  
指定文件的null字符串，非必需参数，与COPY的NULL选项相同。
- encoding  
指定文件的编码，非必需参数，与COPY的ENCODING选项相同。
- force\_not\_null  
这是一个布尔选项，非必需参数。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。
- 对外表做查询操作时不允许写操作。
- 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。

## 7.17 GTM

### 7.17.1 特性概述

#### GTM 概述

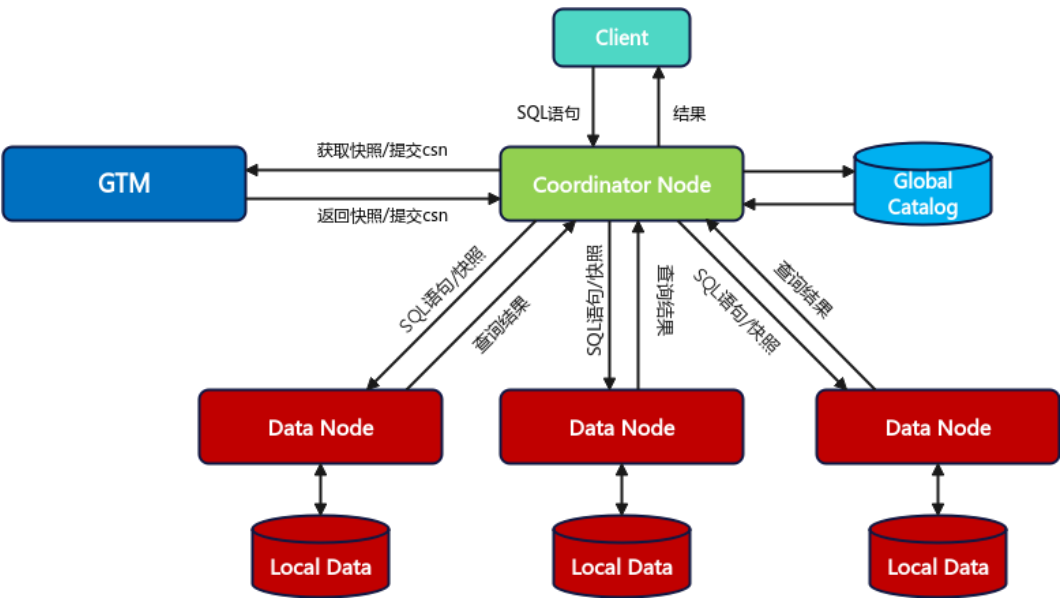
GTM全称为全局事务管理器（Global Transaction Manager），主要分发事务XID和快照Snapshot以及序列号Sequence，保证全局事务的一致性。

#### 技术架构

GTM采用主备方式，是系统中常驻的服务进程，使用多线程架构实现。其主要功能有三个：生成事务XID，生成快照Snapshot，管理序列号Sequence。

GTM为被动接受连接，通常只有CN会主动连接GTM，DN在AutoVacuum（后台元组清理线程）工作时连接GTM。GTM在整个高可用架构的关系如[图7-36](#)所示。

图 7-36 GTM 架构关系



说明

GTM-Free模式下，CN不会通过GTM获取快照及提交CSN，具体细节参见[GTM-Free](#)章节。

GTM的三种状态：

Pending：表示初始启动，即GTM待仲裁。

Primary：表示为主GTM。

Standby：表示为备GTM。

GTM的两种连接状态：

Connection ok：表示主备连接正常。

Connection bad：表示主备连接异常。

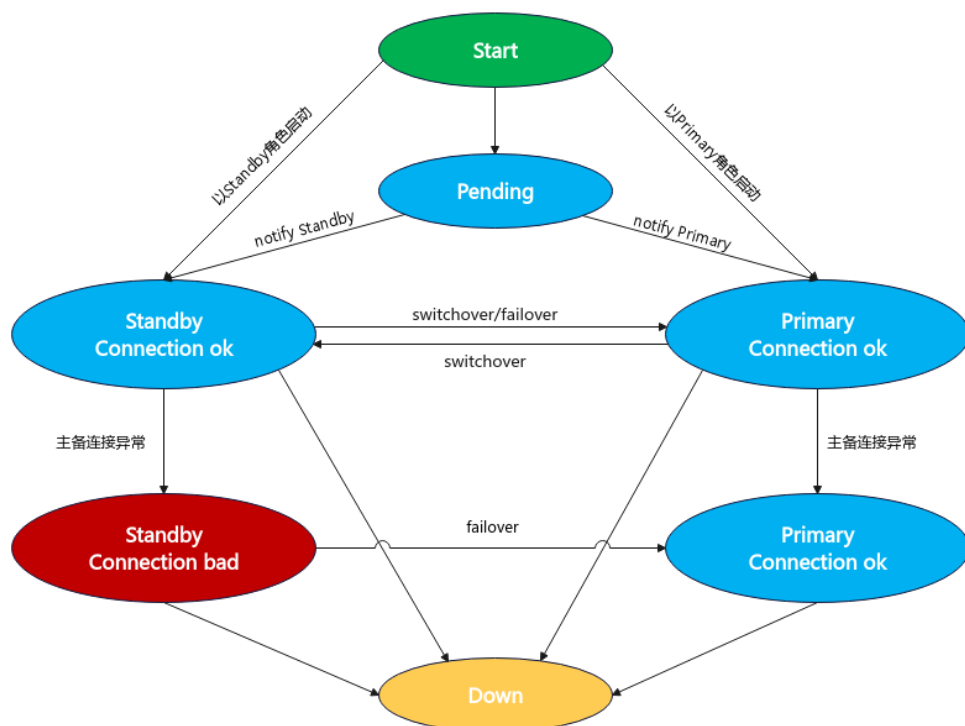
GTM的同步状态：

Sync：同步模式（主机写入数据时需要等待备机同步）。

Most available：最大可用模式（主机写入数据时不需要等待备机同步）。

GTM的三种状态和两种连接状态，如[图7-37](#)所示：

图 7-37 GTM 状态变更



#### 说明

集群部署中，CM都是以Pending模式启动GTM组件，再通过主备仲裁后通知GTM升主或者降备。用户可通过gtm\_ctl工具选择GTM以指定启动模式启动，但该操作需谨慎执行，不建议用户直接对GTM组件状态进行操作，GTM状态应由CM进行管理。

## 7.17.2 GTM 模式

为适应不同的并发和一致性要求，GaussDB提供了三种不同的GTM模式：GTM、GTM-Lite、GTM-Free。三种模式之间的主要区别是中心事务管理节点GTM的压力和事务处理流程不同。

当前版本不支持三种模式之间的相互切换，因此建议安装时默认使用GTM-Lite模式，以确保在升级前后GTM模式保持不变。

### 7.17.2.1 GTM

#### 7.17.2.1.1 GTM 模式简介

##### 功能定义

GTM模式采用全局事务id的策略，全局所有节点都需要通过GTM获取统一递增的事务号CSN，同时事务提交需要通过GTM获取递增的事务号CSN以保证分布式事务的一致性。

##### 用户场景

当前版本已不再支持GTM模式，在用户需要保证全局事务一致性的情况下，GTM-Lite模式为更优选择。

## 规格约束

从503.0版本开始不支持GTM模式，配置GTM模式仍生效为GTM-Lite模式。

### 7.17.2.1.2 技术原理

在GTM模式下，GTM负责维护全局事务管理、全局CSN和Sequence的递增。

执行CN在开启事务时需向GTM申请全局递增的事务号以保证分布式事务的一致性，并在提交时通过GTM获取全局CSN作为事务提交CSN。

### 7.17.2.2 GTM-Lite

#### 7.17.2.2.1 GTM-Lite 模式简介

## 功能定义

分布式事务强一致性要求下，GTM-Lite模式是性能最优的部署模式，其通过将全局事务id转换为本地事务id，同时维护全局csn，既保证了全局事务强一致性又消减了单点性能瓶颈。

在GTM模式下，由于采用全局事务id的策略，所以每个活跃事务要在GTM注册槽位，GTM负责管理和分发全局事务id，这导致业务并发量大的时候，GTM容易成为单点瓶颈。

与GTM模式相反，GTM-Free模式下CN/DN不与GTM交互，通过CN/DN分别维护本地事务id来保证事务系统的正常运行。这种模式下由于缺少全局事务id以及GTM的其他协调（csn），因此事务一致性会受到影响，从而限制了这种模式的适用范围。

综上，GTM和GTM-Free这两种模式在效率和一致性要求上并没有达到一个很好的平衡，而影响这种平衡的主要因素在于协调，协调带来更多一致性的保障，但是却降低了性能。因此，GTM-Lite模式则在尽可能少的协调情况下，达到尽可能高的性能。

## 用户场景

当前GTM-Lite模式为主流使用场景，在用户需要保证全局事务一致性的情况下，性能最优和并发瓶颈最低场景下的最佳选择。

### 7.17.2.2.2 技术原理

GTM-Lite的主要目标就是在消除GTM瓶颈影响的同时，尽可能通过更少的信息交互，并且协调事务的并发，从而保证事务一致性的同时提升性能。因此，GTM-Lite的主要技术原理包含以下4点：

- 本地事务id：不再从GTM获取事务id，每个CN/DN各自维护自己的事务id；
- 全局csn：GTM只维护全局csn，不再维护事务槽位信息；
- 单节点事务优化：引入多版本prepared array（二阶段事务数组）优化单节点事务可见性判断；
- 本地多版本快照回收：引入Snapshot is invalid（无效快照，当前使用快照小于当前全局最小csn）机制，解决没有全局事务id后，每个节点RecentGlobalXmin（最近使用的最小全局事务号）更新问题，以支持本地多版本快照回收。

### 7.17.2.3 GTM-Free

### 7.17.2.3.1 GTM-Free 模式简介

#### 功能定义

GTM-Free模式下，中心事务管理节点不再参与事务处理，从而消除了GTM的单点瓶颈，提升了事务处理性能。然而，在一致性方面，该模式仅支持所有事务完成后保证读取的外部一致性，不支持分布式事务的强一致性读取，也不支持以下这种依赖查询结果的事务一致性。

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

#### 用户场景

GTM-Free模式适用于业务对分布式事务没有强一致性要求，且用户表使用合理分布键的场景。

#### 规格约束

GTM-Free模式下，由于采用了分布式事务最终一致的执行和并发控制机制，因此约束了部分语法的使用场景和使用方式。如果业务确有使用被限制语法的需求，那么在明确了解最终一致性行为的基础上，需要对业务进行一定程度的改造。主要涉及的约束语法和改造建议如下：

1. 总体原则：所有用户表必须指定分布键（DISTRIBUTE BY），并且选择合理的分布键：
  - 考虑数据分布均匀。
  - 尽量选择查询中的关联条件作为分布键，保证关联查询不会引起DN节点之间的数据流动。
  - 考虑将表的主键作为分布键。
2. SELECT：
  - 查询时，WHERE条件应包含所有分布键字段等值查询条件。
  - 避免在SELECT目标列中使用子查询，可能导致计划无法下推到DN执行，影响执行性能。

#### 3. DML：

默认不支持跨节点事务，如果所执行的DML语句包含跨节点事务，会报错处理，具体分为两种场景：

- a. 如果用户语句在数据库内部被拆分成多条独立语句执行，会报错如下：  
INSERT/UPDATE/DELETE/MERGE contains multiple remote queries under GTM-free mode  
Unsupport DML two phase commit under gtm free mode. modify your SQL to generate light-proxy or fast-query-shipping plan.

此时需要修改语句，在单节点执行。例如：

```
INSERT INTO t SELECT * FROM b WHERE b.c = xx;
```

假设t表和b表的分布键不同，且上述WHERE条件只会过滤出一条数据。在不打开enable\_stream\_operator的情况下，上面的查询在数据内部会被拆分成两条独立语句串行执行：首先执行SELECT \* FROM b WHERE b.c = xx从某个DN节点抽取到目标记录；然后再执行INSERT INTO t语句，将抽取的目标记录下发到另一个DN节点完成插入。在GTM-Free模式下，这样的语句执行方式会返回上述报错。类似的，CREATE TABLE AS SELECT \* FROM、带子查询的DELETE/JOIN/INSERT等语句，也可能出现类似报错。

业务改造方案：在业务执行之前，需要增加以下命令来打开流算子，使得业务语句可以被整体下推执行。



```
SET enable_stream_operator=on;
```

- b. 如果同一个用户语句在数据库内部涉及多节点执行，会报错如下：

```
Your SQL needs more than one datanode to be involved in.
```

此时建议对语句进行修改，使得能够在单节点上执行。例如：

```
INSERT INTO t VALUES(3,3),(1,1);
```

假设(3,3)和(1,1)分布在不同的DN节点上，那么上述语句在数据库内部的执行过程会涉及两个DN节点。在GTM-Free模式下，这样的语句执行方式会返回上述报错。

业务改造方案：对于上述语句，如果业务确需要在多个节点上执行，需要在语句中添加一个hint来避免报错，如下：

```
INSERT /*+ multinode */ INTO t VALUES(3,3),(1,1);
```

类似的，对DELETE和UPDATE语句也有类似约束，一般建议用户在DELETE和UPDATE语句的WHERE条件中加上分布键等值过滤条件。

4. 建议开发阶段在jdbc连接串内设置application\_type=perfect\_sharding\_type，这样所有跨节点读写操作的SQL都会报错，用来提示开发人员尽早优化语句。

### 7.17.2.3.2 技术原理

GTM-Free模式采用各节点使用本地事务id和本地快照csn的方式，消除了各节点事务对于GTM的依赖。

因此GTM-Free模式对于分布式事务的一致性没有强保证。

## 7.17.3 操作指导

### 设置 GTM 模式

相关的GUC参数包括enable\_gtm\_free和gtm\_option（具体参见《参考》中“数据库运行参数说明 > GUC参数说明 > GTM相关参数”章节），可通过gsql执行show语句查询当前GTM模式：

```
SHOW enable_gtm_free;
SHOW gtm_option;
```

具体模式判断方法如下：

- GTM模式：enable\_gtm\_free=off 且 gtm\_option=0。（当前版本已不支持）
- GTM-Lite模式：enable\_gtm\_free=off 且 gtm\_option=1。
- GTM-Free模式：enable\_gtm\_free=on 或 gtm\_option=2。

通过以上方式可设置GTM使用模式。

#### 说明

由于GTM-Lite模式在分布式事务强一致性条件下，减轻了单点性能压力，事务的并发性能优于GTM模式，当前版本已不再支持GTM模式。设置enable\_gtm\_free=off 且 gtm\_option=0，仍为GTM-Lite模式。

## 7.17.4 分布式最佳实践

GTM部署与业务需求、集群规模强相关。

## 7.17.5 常见问题

### 问题 1: SQL 语句执行卡顿，查看日志存在 GTM 相关 WARNING

现象分析：SQL语句执行需要通过GTM获取快照来进行事务可见性判断，日志中存在以下告警，该现象为连接GTM失败。

Can not connect to all GTMs and then retry(x times).

处理方法：可通过以下命令查询GTM状态：

```
cm_ctl query -Cvip
```

- 若GTM不存在Primary角色，为GTM正处于倒换状态，则等待倒换完成即可。
- 若GTM存在Primary角色，且状态正常，则可能为节点网络通信故障，通过执行ping命令，诊断CN节点与GTM Primary的网络通信是否正常。

### 问题 2: 慢 SQL 诊断等待事件耗时最大为 gtm get snapshot

现象分析：出现慢SQL后，使用statement\_history视图（使用方式参考《参考》中“系统表和系统视图 > 系统表 > 其他系统表 > STATEMENT\_HISTORY”章节）查询结果，wait\_event耗时最大为gtm get snapshot。

处理方法：通过执行ping命令，诊断CN节点与GTM Primary的网络时延是否正常。

## 7.18 敏感数据发现

敏感数据发现功能提供函数gs\_sensitive\_data\_discovery()和gs\_sensitive\_data\_discovery\_detail()，通过调用的不同函数，指定扫描对象和敏感数据分类器，得到对应扫描对象不同明细级别的敏感数据信息。

### 使用敏感数据发现函数扫描数据

#### 步骤1 创建扫描目标。

```
gaussdb=# CREATE SCHEMA test_schema1;
gaussdb=# CREATE TABLE test_schema1.normal_table1 (email VARCHAR(50), phonenum text, creditcard
bytea, name BPCHAR(50), encrypted_data NVARCHAR2(500));
gaussdb=# INSERT INTO test_schema1.normal_table1 VALUES ('Gauss_1@huawei.com', '13345678901',
'4951-7843-2960-0886', '高斯', 'r1Pj4DKQopSmZfSr/mXwieGcrE+O2iX3G04w1Tda0zqX/pD/
Az90tLuvy0ZnCmUg0/Kl/g89ZqxxAZzta8jQ6bvicRpdXrDCQTHAfsXYSEScB9Idqtw7kdm1LHq89qs');
```

#### 步骤2 调用敏感数据发现函数扫描目标。

- 调用gs\_sensitive\_data\_discovery()函数，扫描目标数据，返回统计的扫描结果。  
gaussdb=# SELECT \* FROM gs\_sensitive\_data\_discovery('test\_schema1.normal\_table1', 'all');  
target | percentage\_of\_sensitive\_data |  
classifier\_type\_distribution  
-----+-----  
test\_schema1.normal\_table1 | 5/5 | Email: 1/5 CreditCard: 1/5 PhoneNumber: 1/5  
ChineseName: 1/5 EncryptedContent: 1/5  
(1 row)
- 调用gs\_sensitive\_data\_discovery\_detail()函数，扫描目标数据，返回详细的扫描结果。  
gaussdb=# SELECT \* FROM gs\_sensitive\_data\_discovery\_detail('test\_schema1', 'all');  
schema | table | column | classifier | laws\_and\_regulations  
-----+-----+-----+-----+-----  
test\_schema1 | normal\_table1 | email | Email | GDPR  
test\_schema1 | normal\_table1 | phonenum | PhoneNumber | GDPR

```
test_schema1 | normal_table1 | creditcard | CreditCard | GDPR
test_schema1 | normal_table1 | name | ChineseName | GDPR
test_schema1 | normal_table1 | encrypted_data | EncryptedContent | GDPR
(5 rows)
```

### 步骤3 清理数据。

```
gaussdb=# DROP TABLE test_schema1.normal_table1;
gaussdb=# DROP SCHEMA test_schema1;
(3 row)
```

----结束

## 7.19 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

请输入如下命令查看动态数据脱敏功能是否已开启。

```
gaussdb=# SHOW enable_security_policy;
enable_security_policy

on
(1 row)
```

enable\_security\_policy取值为on时表示开启，取值为off时表示关闭。如果显示为off时请参考《参考》中“数据库运行参数说明 > GUC参数说明 > 安全配置”章节进行设置。

### 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 j3k2l3 | 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：《参考》中“SQL参考 > SQL语法 > C > 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，支持适配用户创建的自定义函数。

```
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 查询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

## 7.20 HTAP 混合负载

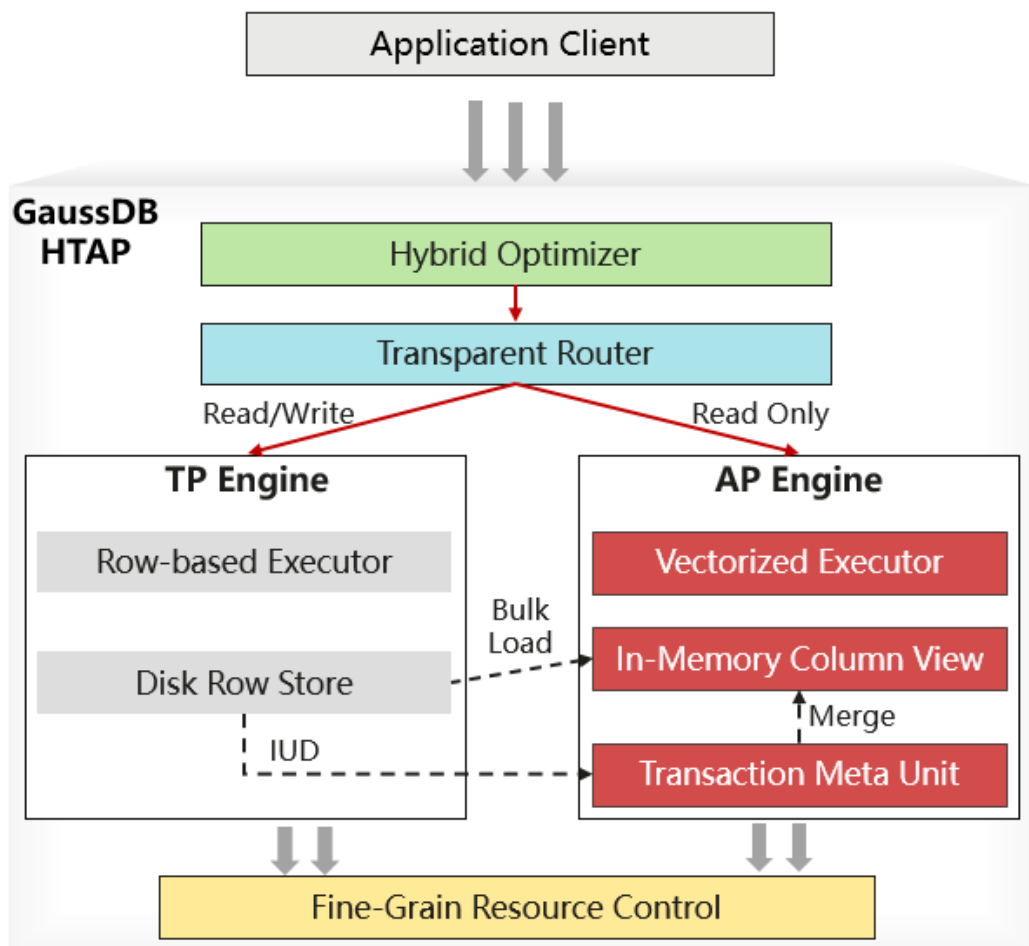
### 7.20.1 概述

HTAP（Hybrid Transactional/Analytical Processing，混合事务/分析处理）在传统交易型数据库以磁盘行存与火山模型执行器高效支持交易型查询负载基础上，通过引入列式存储、向量化执行引擎、混合负载透明路由，在单个交易数据库内同时实现高效的交易型和分析型查询负载处理能力。

## HTAP 架构

HTAP在原有行存数据引擎与执行引擎的基础上，引入了IMCV（In-memory Column View，列式内存引擎）、混合负载透明路由和向量化执行引擎，以提升实时AP查询的处理能力，架构图如图7-38所示。

图 7-38 HTAP 架构图



- IMCV：实现内存列格式维护行存数据快照，支持更快的列数据扫描。
- 向量化执行引擎：实现列扫描数据的批量执行能力，支持更快的连接、聚合和表达式评估运算。
- TMU（Transaction Metadata Unit，实时事务单元）：在IMCV快照基础上记录实时活跃事务更新行，支持IMCV数据扫描获取实时活数据，即实现IMCV扫描与SeqScan结果强一致。
- 混合负载透明路由：识别查询是TP（Transactional Processing，事务处理）点查或AP（Analytical Processing，分析处理）分析型查询，优化并路由至行或列引擎高效执行。
- 资源管控与调度：实现HTAP混合负载下TP、AP负载的内存资源管控能力。

## 7.20.2 使用指导

GaussDB分布式部署数据库的兼容性默认为MySQL，HTAP功能仅支持ORA兼容性数据库。因此需要创建一个ORA兼容性的数据库，该章节所有示例均在该ORA兼容性数据库下操作。

示例

```
gaussdb=# CREATE DATABASE htapdb WITH DBCOMPATIBILITY='ORA';
CREATE DATABASE
gaussdb=# \c htapdb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "htapdb" as user "xxx".
htapdb=#
```

### 7.20.2.1 参数设置

- HTAP混合负载功能。  
HTAP混合负载功能由参数enable\_htap控制，开启为on，关闭为off。  
enable\_htap为实例级别参数，详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > HTAP混合负载”章节。
- IMCV列存扫描功能。  
IMCV列存扫描功能由参数enable\_imcvscan控制，开启为on，关闭为off。  
enable\_imcvscan为会话级别参数，可以在配置文件中设置或者在会话过程中执行set enable\_imcvscan=on。
- HTAP路由模式设置。  
HTAP路由模式由参数htap\_router\_mode控制，设置为auto表示根据实际执行过程中的代价计算选择行存或者列存引擎。htap\_router\_mode为会话级别参数，可以在配置文件中设置或者在会话过程中执行set htap\_router\_mode='auto'。

#### 说明

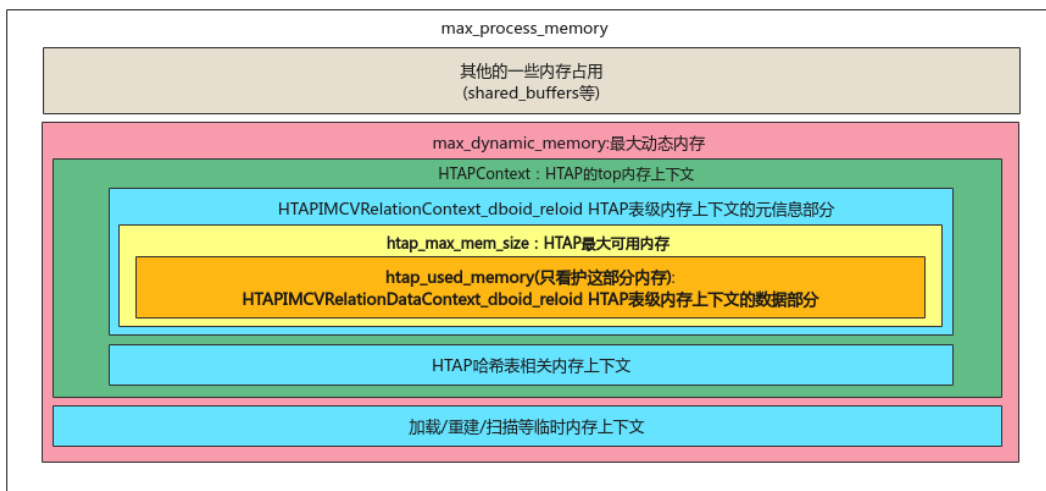
HTAP新增参数可以根据实际使用情况进行组合设置，新增参数说明请参见《参考》中“数据库运行参数说明 > GUC参数说明 > HTAP混合负载”章节。

### 7.20.2.2 内存管控

HTAP的数据全部存放于GaussDB的内存上，配备对应的内存管控功能，避免内存占用过多影响行存查询或导致数据库内存溢出。

HTAP的内存同时包含动态共享内存和非动态共享内存两部分。非动态共享内存部分为加载/重建/扫描等临时内存上下文等。动态共享内存为三层结构：第一层挂载在顶级动态内存上下文上，用于存放IMCV元信息以及HTAP必要的辅助数据如哈希表等；第二层挂载于第一层内存上下文，存放各IMCV表中相关元信息，主要存放IMCU元信息和TMU数据等；第三层为IMCU数据信息，挂载于第二层，包含了IMCU实际数据等。内存上下文结构如[图7-39](#)所示。全部HTAP内存中使用htap\_max\_mem\_size看护HTAP表级内存上下文的数据部分内存开销，其他视为底噪不看护。

图 7-39 HTAP 内存上下文结构



相关GUC参数为htap\_max\_mem\_size和htap\_memctl\_policy，详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > HTAP混合负载”章节。

- htap\_max\_mem\_size：设置HTAP可用内存大小。取值范围为100MB~1TB。默认为1GB。
- htap\_memctl\_policy：设置HTAP特性内存管理的策略模式。取值范围为'swap'、'escape'，默认为'escape'。

HTAP在pg\_total\_memory\_detail系统视图中新增了memorytype为max\_htap\_memory和htap\_used\_memory的两行记录，分别用于表示HTAP最大可用内存和HTAP当前已用内存，该视图详情请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG\_TOTAL\_MEMORY\_DETAIL”章节。

用户可以通过使用gs\_global\_imcv\_info()系统函数确认表的IMCV基本信息，该函数详情请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > HTAP系统函数”章节。

## 示例

```
--使用分布式3C2D，1主1备集群，gsq连接CN5001节点。
--查看GUC参数。
htapdb=# SHOW htap_max_mem_size;
 htap_max_mem_size

1GB
(1 row)

htapdb=# SHOW htap_memctl_policy;
 htap_memctl_policy

swap
(1 row)

--连接DN6001节点主机，查看DN节点内存状态以及表信息。
htapdb=# SELECT * FROM pg_total_memory_detail WHERE memorytype='max_htap_memory' OR
memorytype='htap_used_memory';
 nodeid | memorytype | memorybytes
-----+-----+-----
 6001 | max_htap_memory | 1024
 6001 | htap_used_memory | 0
(2 rows)
```

```
--连接CN5001节点。
htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info();
 node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size |
cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

--创建普通表htap_imcv_mem_ctl_1并插入少量数据。
htapdb=# CREATE TABLE htap_imcv_mem_ctl_1(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) distribute by hash(id);
CREATE TABLE

htapdb=# INSERT INTO htap_imcv_mem_ctl_1 VALUES (generate_series(1,1000000),
generate_series(1,1000000), generate_series(1,1000000), 'Jack', 'AAAA');
INSERT 0 1000000

--对小数据量的表执行ALTER COLVIEW创建IMCV。
htapdb=# ALTER TABLE htap_imcv_mem_ctl_1 COLVIEW(id, salary, name, comments);
ALTER TABLE

--执行手动加载，确保所有行存数据都已加载到IMCV。
htapdb=# SELECT total from gs_imcv_flush('htap_imcv_mem_ctl_1');
 total

 6
(1 row)

--查看DN节点内存状态以及表信息。
htapdb=# EXECUTE DIRECT ON (dn_6001_6002) $$SELECT * FROM pg_total_memory_detail WHERE
memorytype='max_htap_memory' OR memorytype='htap_used_memory'$$;
 nodename | memorytype | memorybytes
-----+-----+-----
dn_6001_6002 | max_htap_memory | 1024
dn_6001_6002 | htap_used_memory | 3
(2 rows)

htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info();
 node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size |
cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+
dn_6001_6002 | htap_imcv_mem_ctl_1 | 4 | 1 3 4 5 | 3 | 15 | 3553162 |
0 |
dn_6003_6004 | htap_imcv_mem_ctl_1 | 4 | 1 3 4 5 | 3 | 15 | 3545231 |
0 |
(2 rows)

--向已有IMCV的表中插入少量数据。
htapdb=# INSERT INTO htap_imcv_mem_ctl_1 SELECT * FROM htap_imcv_mem_ctl_1;
INSERT 0 1000000

--查看DN节点内存状态以及表信息，其中部分新数据可能存在于TMU中，htap_used_memory会在CU重建后增加。
--一段时间后
htapdb=# EXECUTE DIRECT ON (dn_6001_6002) $$SELECT * FROM pg_total_memory_detail WHERE
memorytype='max_htap_memory' OR memorytype='htap_used_memory'$$;
 nodename | memorytype | memorybytes
-----+-----+-----
dn_6001_6002 | max_htap_memory | 1024
dn_6001_6002 | htap_used_memory | 6
(2 rows)

htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
```



```
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info();
 node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size |
cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----
+-----+
dn_6001_6002 | htap_imcv_mem_ctl_1 | 4 | 1 3 4 5 | 6 | 25 | 6303358 | 0
| 0
dn_6003_6004 | htap_imcv_mem_ctl_1 | 4 | 1 3 4 5 | 6 | 25 | 6295427 | 0
| 0
(2 rows)

htapdb=# DROP TABLE htap_imcv_mem_ctl_1;
DROP TABLE
```

7.20.2.3 DDL

7.20.2.3.1 创建和卸载 IMCV

普通表创建 IMCV

- 语法
  - HTAP可以通过ALTER方式对普通表创建IMCV，如表7-25所示。  
详情请参见《参考》中“SQL参考 > SQL语法 > A > ALTER TABLE”章节。

表 7-25 ALTER 方式创建 IMCV 语法和相关影响

| 语法                                          | 加载列              |
|---------------------------------------------|------------------|
| ALTER TABLE ... COLVIEW                     | 全部列。             |
| ALTER TABLE ... NOCOLVIEW                   | 无，IMCV信息被删除。     |
| ALTER TABLE ... NOCOLVIEW COLVIEW(col_list) | col_list中全部列。    |
| ALTER TABLE ... COLVIEW NOCOLVIEW(col_list) | 不在col_list中的其他列。 |

- HTAP可以通过CREATE方式对普通表创建IMCV，创建普通表时可以指定表级IMCV或者列级IMCV。  
详情请参见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE”章节。  
表级IMCV和列级IMCV互相影响决定最终加载哪些IMCV列。表级和列级IMCV关键字的影响关系如表7-26所示。

表 7-26 表级和列级 IMCV 关键字影响关系

| 表级IMCV关键字 | 列级IMCV关键字            | 最终加载列              |
|-----------|----------------------|--------------------|
| COLVIEW   | 无/COLVIEW            | 全部列。               |
| COLVIEW   | 无/COLVIEW/ NOCOLVIEW | 除定义NOCOLVIEW外的其他列。 |

| 表级IMCV关键字       | 列级IMCV关键字               | 最终加载列         |
|-----------------|-------------------------|---------------|
| NOCOLVIEW<br>/无 | 无/COLVIEW/<br>NOCOLVIEW | 定义了COLVIEW的列。 |

• 示例

- 通过ALTER方式创建IMCV。

--创建普通表htap\_test并插入数据。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) with (storage_type = ustore) distribute by hash(id);
CREATE TABLE
```

```
htapdb=# INSERT INTO htap_test VALUES (generate_series(1,100), 1, 8000, 'Allen', 'For test');
INSERT 0 100
```

--建表插入数据后执行ALTER COLVIEW创建IMCV。

```
htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE
htapdb=# SELECT total FROM gs_imcv_flush('htap_test');
total

2
(1 row)
```

```
htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
```

```
node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
```

```
-----+-----+-----+-----+-----+-----+-----+-----+-----
```

```
+-----+-----+
dn_6001_6002 | htap_test | 5 | 1 2 3 4 5 | 1 | 6 | 8153 | 0
| 0
```

```
dn_6003_6004 | htap_test | 5 | 1 2 3 4 5 | 1 | 6 | 8185 | 0
| 0
```

(2 rows)

--卸载IMCV。

```
htapdb=# ALTER TABLE htap_test NOCOLVIEW;
ALTER TABLE
```

```
htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
```

```
node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
```

```
-----+-----+-----+-----+-----+-----+-----+-----+-----
```

+-----+-----+
(0 rows)

--加载除dept\_id/comments外的其他列。

```
htapdb=# ALTER TABLE htap_test COLVIEW NOCOLVIEW(dept_id, comments);
ALTER TABLE
```

```
htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
```

```
node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
```

```
-----+-----+-----+-----+-----+-----+-----+-----+-----
```

+-----+-----+

```
dn_6001_6002 | htap_test | 3 | 1 3 4 | 1 | 4 | 8153 | 0
| 0
dn_6003_6004 | htap_test | 3 | 1 3 4 | 1 | 4 | 8185 | 0
| 0
(2 rows)

--卸载IMCV。
htapdb=# ALTER TABLE htap_test NOCOLVIEW;
ALTER TABLE

--加载id/name列。
htapdb=# ALTER TABLE htap_test NOCOLVIEW COLVIEW(id, name);
ALTER TABLE

htapdb=# SELECT node_name, rel_name, imcvnattr, imcvkey, rowgroup_num, cu_num_in_mem,
cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | 2 | 1 4 | 1 | 3 | 8153 | 0
| 0
dn_6003_6004 | htap_test | 2 | 1 4 | 1 | 3 | 8185 | 0
| 0
(2 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

#### 说明

gs\_imcv是HTAP相关的共享系统表，详情请参见《参考》中“系统表和系统视图 > 系统表 > HTAP > GS\_IMCV”章节。

- 建表时指定表级IMCV关键字为COLVIEW，不指定列级IMCV关键字。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) COLVIEW with (storage_type = ustore) distribute by hash(id);
CREATE TABLE

--查询系统表gs_imcv，表的全部列均加载IMCV。
htapdb=# SELECT relname,imcvkey FROM gs_imcv;
 relname | imcvkey
-----+-----
 htap_test | 1 2 3 4 5
(1 row)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

- 建表时指定表级IMCV关键字为COLVIEW，指定部分列级IMCV关键字为NOCOLVIEW，其余列不指定IMCV关键字。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int NOCOLVIEW,
 salary int,
 name varchar(20) NOCOLVIEW,
 comments varchar(30)) COLVIEW WITH (storage_type = ustore) DISTRIBUTE BY
HASH(id);
CREATE TABLE

--查询系统表gs_imcv，列id、salary和comments加载IMCV。
htapdb=# SELECT relname,imcvkey FROM gs_imcv;
 relname | imcvkey
-----+-----
 htap_test | 1 3 5
```

```
(1 row)

htapdb=# DROP TABLE htap_test;
DROP TABLE

- 建表时不指定表级IMCV关键字，部分列指定IMCV关键字为COLVIEW，其余列不指定IMCV关键字。
htapdb=# CREATE TABLE htap_test(
 id int COLVIEW,
 dept_id int,
 salary int,
 name varchar(20) COLVIEW,
 comments varchar(30)) WITH (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE

--查询系统表gs_imcv，列id、name加载IMCV。
htapdb=# SELECT relname,imcvkey FROM gs_imcv;
 relname | imcvkey
-----+-----
 htap_test | 1 4
(1 row)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

分区表创建 IMCV

- 语法
  - HTAP可以通过ALTER方式对分区表创建IMCV，分区表ALTER方式创建IMCV继承普通表的用法。对分区表额外增加对分区的ALTER语法，如下所示：  
ALTER TABLE table\_name MODIFY PARTITION part\_name COLVIEW;  
详情请参见《参考》中“SQL参考 > SQL语法 > A > ALTER TABLE”章节。
  - HTAP可以通过CREATE方式对分区表创建IMCV，创建分区表时可以指定表级IMCV、列级IMCV或分区级IMCV。  
具体语法请参见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE”章节  
分区表的IMCV加载列同样由表级IMCV和列级IMCV决定，规则与普通表一致。分区表有分区级IMCV，分区级IMCV和表级IMCV互相影响，决定最终有哪些分区加载IMCV。分区的IMCV列信息和基表保持一致。表级IMCV和分区级IMCV关键字的影响关系如表7-27所示。

表 7-27 表级和分区级 IMCV 关键字影响关系

| 表级IMCV关键字               | 分区IMCV关键字 | 分区是否创建IMCV |
|-------------------------|-----------|------------|
| COLVIEW                 | 无/COLVIEW | 是          |
| 无/COLVIEW/<br>NOCOLVIEW | NOCOLVIEW | 否          |
| 无/NOCOLVIEW             | COLVIEW   | 是          |

- 示例
  - 指定分区加载IMCV。  
--创建分区表htap\_test并插入数据。  
htapdb=# CREATE TABLE htap\_test(  
 id int,

```
dept_id int,
salary int,
name varchar(20),
comments varchar(30)
) WITH(storage_type = ustore) DISTRIBUTE BY HASH(id)
PARTITION BY HASH (id)(
PARTITION id_p1,
PARTITION id_p2,
PARTITION id_p3);
CREATE TABLE

htapdb=# INSERT INTO htap_test VALUES (generate_series(1,100), 1, 8000, 'Allen', 'For test');
INSERT 0 100

--表htap_test没有加载IMCV，系统表gs_imcv查询记录为空。
htapdb=# SELECT relname,imcvkey FROM gs_imcv;
 relname | imcvkey
-----+-----
(0 rows)

--通过ALTER为指定分区id_p1加载IMCV。
htapdb=# ALTER TABLE htap_test MODIFY PARTITION id_p1 COLVIEW;
ALTER TABLE

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info()
ORDER BY node_name, rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num |
cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 1 | 6 | 8101
| 0 | 0
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 1 | 6 | 8101
| 0 | 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
(4 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

- 创建分区表时指定表级IMCV关键字为COLVIEW，不指定分区级IMCV关键字。

```
htapdb=# CREATE TABLE htap_test(
id int,
dept_id int,
salary int,
name varchar(20),
comments varchar(30)
) COLVIEW WITH(storage_type = ustore) DISTRIBUTE BY HASH(id)
PARTITION BY HASH (id)(
PARTITION id_p1,
PARTITION id_p2,
PARTITION id_p3);
CREATE TABLE

--所有分区均加载IMCV。
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info()
ORDER BY node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num |
cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6001_6002 | htap_test | id_p2 | 5 | 1 2 3 4 5 | 0 | 0 | 0
```

```
| 0 | 0
dn_6001_6002 | htap_test | id_p3 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p2 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p3 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
(8 rows)
```

```
htapdb=# DROP TABLE htap_test;
DROP TABLE
```

- 创建分区表时指定表级IMCV关键字为COLVIEW，指定部分分区IMCV关键字为NOCOLVIEW。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)
) COLVIEW WITH(storage_type = ustore) DISTRIBUTE BY HASH(id)
PARTITION BY HASH (id)(
 PARTITION id_p1,
 PARTITION id_p2 NOCOLVIEW,
 PARTITION id_p3);
CREATE TABLE
```

--除id\_p2外的其他分区均加载IMCV。

```
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info()
ORDER BY node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num |
cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6001_6002 | htap_test | id_p3 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p1 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p3 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
(6 rows)
```

```
htapdb=# DROP TABLE htap_test;
DROP TABLE
```

- 创建分区表时指定表级IMCV关键字为NOCOLVIEW，指定部分分区IMCV关键字为COLVIEW。

--一级分区。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)
) NOCOLVIEW WITH(storage_type = ustore) DISTRIBUTE BY HASH(id)
PARTITION BY HASH (id)(
 PARTITION id_p1,
```

```

PARTITION id_p2 COLVIEW,
PARTITION id_p3);
CREATE TABLE

--分区id_p2加载IMCV，其余分区不加载IMCV。
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info()
ORDER BY node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | id_p2 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6001_6002 | htap_test | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | id_p2 | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
dn_6003_6004 | htap_test | 5 | 1 2 3 4 5 | 0 | 0 | 0
| 0 | 0
(4 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

普通表卸载 IMCV

- 语法  
通过ALTER语句删除IMCV列存数据并删除IMCV元信息。  
ALTER TABLE table\_name NOCOLVIEW;

- 示例  
ALTER TABLE ... NOCOLVIEW方式删除IMCV。  
htapdb=# CREATE TABLE htap\_test(  
id int,  
dept\_id int,  
salary int,  
name varchar(20),  
comments varchar(30)) COLVIEW WITH (storage\_type = ustore) DISTRIBUTE BY HASH(id);  
CREATE TABLE  
  
htapdb=# SELECT node\_name, rel\_name, partition\_name, imcvnattr, imcvkey, rowgroup\_num,  
cu\_num\_in\_mem, cu\_mem\_size, cu\_num\_in\_disk, cu\_disk\_size FROM gs\_global\_imcv\_info() ORDER BY  
node\_name,rel\_name;  
 node\_name | rel\_name | partition\_name | imcvnattr | imcvkey | rowgroup\_num | cu\_num\_in\_mem | cu\_mem\_size | cu\_num\_in\_disk | cu\_disk\_size  
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----  
dn\_6001\_6002 | htap\_test | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0  
| 0  
dn\_6003\_6004 | htap\_test | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0  
| 0  
(2 rows)  
  
htapdb=# ALTER TABLE htap\_test NOCOLVIEW;  
ALTER TABLE  
  
--删除IMCV后，系统函数gs\_global\_imcv\_info()查询无记录。  
htapdb=# SELECT node\_name, rel\_name, partition\_name, imcvnattr, imcvkey, rowgroup\_num,  
cu\_num\_in\_mem, cu\_mem\_size, cu\_num\_in\_disk, cu\_disk\_size FROM gs\_global\_imcv\_info() ORDER BY  
node\_name,rel\_name;  
 node\_name | rel\_name | partition\_name | imcvnattr | imcvkey | rowgroup\_num | cu\_num\_in\_mem | cu\_mem\_size | cu\_num\_in\_disk | cu\_disk\_size  
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----  
(0 rows)

```
htapdb=# DROP TABLE htap_test;
DROP TABLE
```

## 分区表卸载 IMCV

- 语法

分区表继承普通表删除IMCV的方式，并且可以针对分区删除IMCV列存数据和IMCV元信息。

```
ALTER TABLE table_name MODIFY PARTITION part_name NOCOLVIEW;
```

- 示例

对分区表删除IMCV。

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)
) WITH (storage_type = ustore) DISTRIBUTE BY HASH(id)
PARTITION BY RANGE (id)
(
 PARTITION id_p1 VALUES LESS THAN (100),
 PARTITION id_p2 VALUES LESS THAN (200) COLVIEW,
 PARTITION id_p3 VALUES LESS THAN (300) COLVIEW,
 PARTITION id_p4 VALUES LESS THAN (400) ,
 PARTITION id_p5 VALUES LESS THAN (MAXVALUE));
CREATE TABLE
```

```
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
```

| node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size |
|-----------|----------|----------------|-----------|---------|--------------|---------------|-------------|----------------|--------------|
|-----------|----------|----------------|-----------|---------|--------------|---------------|-------------|----------------|--------------|

|              |           |       |   |           |   |   |   |  |   |
|--------------|-----------|-------|---|-----------|---|---|---|--|---|
| dn_6001_6002 | htap_test | id_p3 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |
| dn_6001_6002 | htap_test |       |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  | 0 |
| dn_6001_6002 | htap_test | id_p2 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |
| dn_6003_6004 | htap_test | id_p3 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |
| dn_6003_6004 | htap_test |       |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  | 0 |
| dn_6003_6004 | htap_test | id_p2 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |

(6 rows)

```
htapdb=# ALTER TABLE htap_test MODIFY PARTITION id_p2 NOCOLVIEW;
ALTER TABLE
```

```
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
```

| node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem | cu_mem_size | cu_num_in_disk | cu_disk_size |
|-----------|----------|----------------|-----------|---------|--------------|---------------|-------------|----------------|--------------|
|-----------|----------|----------------|-----------|---------|--------------|---------------|-------------|----------------|--------------|

|              |           |       |   |           |   |   |   |  |   |
|--------------|-----------|-------|---|-----------|---|---|---|--|---|
| dn_6001_6002 | htap_test | id_p3 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |
| dn_6001_6002 | htap_test |       |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  | 0 |
| dn_6003_6004 | htap_test | id_p3 |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  |   |
| dn_6003_6004 | htap_test |       |   |           |   |   |   |  |   |
| 0            | 0         |       | 5 | 1 2 3 4 5 | 0 | 0 | 0 |  | 0 |



```
(4 rows)

htapdb=# ALTER TABLE htap_test DROP PARTITION id_p3;
ALTER TABLE

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

### 7.20.2.3.2 其他 DDL

#### TRUNCATE

TRUNCATE表时会清除行存数据，同时级联处理加载的IMCV列存数据，不删除IMCV元信息。

TRUNCATE分区时会清除行存表分区数据，同时级联处理该分区的IMCV列存数据，不删除IMCV元信息。

示例：

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) WITH (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE

htapdb=# INSERT INTO htap_test VALUES (generate_series(1,30), 1, 8000, 'Alice', 'for test');
INSERT 0 30

htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE

htapdb=# SELECT count(*) FROM htap_test;
 count

 30
(1 row)

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8099 | 0
| 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8099 | 0
| 0
(2 rows)

htapdb=# TRUNCATE TABLE htap_test;
TRUNCATE TABLE
htapdb=# SELECT count(*) FROM htap_test;
 count

 0
```

```
(1 row)

--TRUNCATE不删除IMCV元信息，从系统函数gs_global_imcv_info()中仍能查询到记录。
htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0
| 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0
| 0
(2 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

## DROP

DROP表时会删除行存表数据，同时级联删除该表的所有IMCV列存数据，以及对应的IMCV元信息。

DROP分区时会删除行存表的对应分区数据，同时级联删除该分区的IMCV列存数据，以及对应的IMCV元信息。

示例：

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) with (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE

htapdb=# INSERT INTO htap_test VALUES (generate_series(1,30), 1, 8000, 'Alice', 'for test');
INSERT 0 30

htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE

htapdb=# SELECT count(*) FROM htap_test;
 count

 30
(1 row)

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0
| 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 0 | 0 | 0 | 0
| 0
(2 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
```

```
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
(0 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

## VACUUM FULL

VACUUM FULL会在对行存表进行处理的同时级联处理列存数据。

对于HTAP列存数据的处理包含两个阶段：

1. 删除原有的IMCV信息。
2. 重建IMCV，通过重新加载实现IMCV列存数据内存紧凑。

示例：

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) WITH (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE

htapdb=# INSERT INTO htap_test VALUES (generate_series(1,30), 1, 8000, 'Alice', 'for test');
INSERT 0 30

htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8099 | 0
| 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8099 | 0
| 0
(2 rows)

htapdb=# DELETE FROM htap_test WHERE id<10;
DELETE 9

htapdb=# VACUUM FULL htap_test;
VACUUM

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
dn_6001_6002 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8087 | 0
| 0
dn_6003_6004 | htap_test | | 5 | 1 2 3 4 5 | 1 | 6 | 8093 | 0
| 0
(2 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

## 7.20.2.4 DML/DQL

### 7.20.2.4.1 DML

HTAP通过实时事务单元TMU记录DML操作带来的增量数据变更，并维护IMCV列查询的数据新鲜度，保障列存与行存实时数据的一致性。

示例：

```
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) with (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE

htapdb=# INSERT INTO htap_test VALUES (1, 1, 2000, 'Allen', 'for test');
INSERT 0 1

htapdb=# INSERT INTO htap_test VALUES (2, 2, 3000, 'Lily', 'for test');
INSERT 0 1

htapdb=# INSERT INTO htap_test VALUES (3, 2, 4000, 'Ann', 'for test');
INSERT 0 1

--加载IMCV。
htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE

htapdb=# SELECT reloid, relname, dbname, username, parentoid, imcvispart, imcvnattr, imcvkey, priority
FROM gs_imcv;
 reloid | relname | dbname | username | parentoid | imcvispart | imcvnattr | imcvkey | priority
-----+-----+-----+-----+-----+-----+-----+-----+-----
 17949 | htap_test | test | dbdev | 0 | f | 5 | 1 2 3 4 5 | 1
(1 row)
```

--向表中插入新数据。

```
htapdb=# INSERT INTO htap_test VALUES (4, 2, 3000, 'Bob', 'for test');
INSERT 0 1

htapdb=# SET enable_stream_operator=on;
SET

htapdb=# SET enable_fast_query_shipping=off;
SET

--生成并选择执行IMCV执行计划。
htapdb=# EXPLAIN SELECT /*+imcvscan(htap_test)*/ * FROM htap_test;
QUERY PLAN

Row Adapter (cost=36.74..36.74 rows=100 width=27)
-> Vector Streaming (type: GATHER) (cost=3.12..36.74 rows=100 width=27)
 Node/s: All datanodes
 -> Imcv Scan on htap_test (cost=0.00..32.05 rows=100 width=27)
(4 rows)
```

--获取IMCV查询结果，与行存数据一致。

```
htapdb=# SELECT /*+imcvscan(htap_test)*/ * FROM htap_test;
 id | dept_id | salary | name | comments
-----+-----+-----+-----+-----
 1 | 1 | 2000 | Allen | for test
 2 | 2 | 3000 | Lily | for test
 3 | 2 | 4000 | Ann | for test
 4 | 2 | 3000 | Bob | for test
(4 rows)
```

```
htapdb=# DROP TABLE htap_test;
DROP TABLE
```

#### 7.20.2.4.2 DQL

在正确设置参数以及对目标表创建IMCV后便可进行DQL查询。在htap\_router\_mode为auto时，HTAP透明路由会根据数据和SQL特征选择最优执行计划，因此即使对目标表加载了IMCV，透明路由仍有可能会根据代价计算选择行存计划，具体执行计划可通过EXPLAIN查看。

如果希望SQL执行时一定走IMCV列存计划，可以设置htap\_router\_mode为column或者在SQL语句中设置IMCV hint。

透明路由的使用请参见[混合负载透明路由](#)。

示例：

```
htapdb=# SET htap_router_mode='auto';
SET
htapdb=# SHOW htap_router_mode;
htap_router_mode

auto
(1 row)

--建表、插入数据后加载IMCV。
htapdb=# CREATE TABLE htap_test(
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) WITH (storage_type = ustore) DISTRIBUTE BY HASH(id);
CREATE TABLE
htapdb=# INSERT INTO htap_test VALUES (generate_series(1,100), 1, 8000, 'Allen', 'For test');
INSERT 0 100
htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE
htapdb=# SELECT * FROM gs_imcv;
reloid | relname | dbname | username | parentoid | imcvpart | imcvnattr | imcvkey | priority
-----+-----+-----+-----+-----+-----+-----+-----+-----
16982 | htap_test | test | dbdev | 0 | f | 5 | 1 2 3 4 5 | 1
(1 row)

--由于HTAP路由模式为auto，透明路由基于代价计算选择行存的执行计划。
htapdb=# EXPLAIN SELECT count(*) FROM htap_test;
QUERY PLAN

Aggregate (cost=13.23..13.27 rows=1 width=8)
-> Streaming (type: GATHER) (cost=13.23..13.27 rows=2 width=8)
Node/s: All datanodes
-> Aggregate (cost=13.16..13.17 rows=2 width=8)
-> Seq Scan on htap_test (cost=0.00..13.13 rows=20 width=0)
(5 rows)

--IMCV hint指定IMCV列存的执行计划。
htapdb=# EXPLAIN SELECT /*+imcvscan(htap_test)*/ count(*) FROM htap_test;
QUERY PLAN

Row Adapter (cost=32.28..32.28 rows=1 width=8)
-> Vector Aggregate (cost=32.24..32.28 rows=1 width=8)
-> Vector Streaming (type: GATHER) (cost=32.24..32.28 rows=2 width=8)
Node/s: All datanodes
-> Vector Aggregate (cost=32.17..32.18 rows=2 width=8)
-> Imcv Scan on htap_test (cost=0.00..32.05 rows=100 width=0)
(6 rows)

htapdb=# SET htap_router_mode='column';
SET
```

```
htapdb=# SHOW htap_router_mode;
htap_router_mode

column
(1 row)

--HTAP路由模式为column时，强制选择IMCV列存的执行计划。
htapdb=# EXPLAIN SELECT count(*) FROM htap_test;
 QUERY PLAN

Aggregate (cost=2.69..2.73 rows=1 width=8)
-> Streaming (type: GATHER) (cost=2.69..2.73 rows=2 width=8)
 Node/s: All datanodes
-> Aggregate (cost=2.62..2.63 rows=2 width=8)
 -> Seq Scan on htap_test (cost=0.00..2.50 rows=100 width=0)
(5 rows)

htapdb=# DROP TABLE htap_test;
DROP TABLE
```

### 7.20.2.4.3 列回行混合扫描

列回行混合扫描，是一种特殊的列存扫描方式。当查询所访问的列并非全部创建了列存，而且查询谓词涉及的所有列均创建IMCV，基于代价可生成列回行混合扫描计划。

该特性可在内存有限、仅部分列创建IMCV的情况下，利用创建IMCV的列加速查询。

示例：

```
-- 只有部分列存在列存数据的情况下，查询可以走列回行混合扫描。
htapdb=# DROP TABLE IF EXISTS test_table;
CREATNOTICE: table "test_table" does not exist, skipping
DROP TABLE

htapdb=# CREATE TABLE test_table (
 supplier_txn_base_id bigint NOT NULL,
 supplier_id bigint NOT NULL,
 supplier_address_id bigint,
 supplier_txn_code nvarchar2(30) NOT NULL,
 shipment_code nvarchar2(30),
 invoice_currency_code nvarchar2(15)) DISTRIBUTE BY HASH(supplier_txn_base_id);
CREATE TABLE

htapdb=# INSERT INTO test_table VALUES (200, 100, -1, 'CNY', 'ee', 'CNY');
INSERT 0 1

htapdb=# INSERT INTO test_table VALUES (300, 200, -1, 'EUR', 'de', 'EUR');
INSERT 0 1

htapdb=# INSERT INTO test_table VALUES (200, 300, -1, 'US', 'de', 'US');
INSERT 0 1

htapdb=# SET enable_imcvscan=on;
SET

htapdb=# SET htap_router_mode='column';
SET

-- 部分列创建IMCV。
htapdb=# ALTER TABLE test_table COLVIEW(supplier_id, supplier_txn_code);
ALTER TABLE

htapdb=# SELECT node_name, rel_name, partition_name, imcvnattr, imcvkey, rowgroup_num,
cu_num_in_mem, cu_mem_size, cu_num_in_disk, cu_disk_size FROM gs_global_imcv_info() ORDER BY
node_name,rel_name;
 node_name | rel_name | partition_name | imcvnattr | imcvkey | rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```

```
dn_6001_6002 | test_table | | | 2 | 2 4 | | 1 | | 3 | | 12112 | | 0
| 0
dn_6003_6004 | test_table | | | 2 | 2 4 | | 0 | | 0 | | 0 | | 0
| 0
(4 rows)

-- 关闭列回行混合扫描。
htapdb=# SET enable_hybrid_imcvscan=off;
SET

htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_table t WHERE t.supplier_id >= 200;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_table t
Filter: (supplier_id >= 200)
(4 rows)

-- 开启列回行混合扫描。
htapdb=# SET enable_hybrid_imcvscan=on;
SET

-- 查询访问到未创建IMCV的列、谓词列创建了IMCV，仍支持列存扫描。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_table t WHERE t.supplier_id >= 200;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on test_table t
Filter: (supplier_id >= 200)
(5 rows)

htapdb=# DROP TABLE test_table;
DROP TABLE
```

### 7.20.2.5 混合负载透明路由

混合查询负载支持多种路由模式：自动路由、强制列存、强制行存，通过设置 `htap_router_mode='auto'`、`'column'`、`'row'` 来选择。

## 数据准备

创建一个包含几行数据的小表（`test_small`）以及包含3千行数据的大表（`test_large`）并插入数据。小表执行行存扫描更快，大表执行列存扫描更快。

示例：

```
htapdb=# CREATE TABLE test_small (
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30)) DISTRIBUTE BY HASH(id);

CREATE TABLE

htapdb=# INSERT INTO test_small VALUES (1, 1, 1000, 'Jack', 'AAAA');
INSERT 0 1

htapdb=# INSERT INTO test_small VALUES (2, 4, 2000, 'Sam', 'BBBB');
INSERT 0 1

htapdb=# INSERT INTO test_small VALUES (3, 2, 3000, 'Sara', 'CCCC');
INSERT 0 1

htapdb=# INSERT INTO test_small VALUES (4, 6, 4000, 'Oulu', 'DDDD');
INSERT 0 1
```

```
htapdb=# INSERT INTO test_small VALUES (5, 3, 5000, 'John', 'EEEE');
INSERT 0 1

htapdb=# CREATE TABLE test_large (
 id int,
 dept_id int,
 salary int,
 name varchar(20),
 comments varchar(30))DISTRIBUTE BY HASH(id);
CREATE TABLE

htapdb=# INSERT INTO test_large VALUES (generate_series(1,15000), 1, 1000, 'large', 'large table');
INSERT 0 15000

--对表test_small整表创建IMCV。
htapdb=# ALTER TABLE test_small COLVIEW;
ALTER TABLE

--对表test_large部分列创建IMCV。
htapdb=# ALTER TABLE test_large COLVIEW(id,dept_id);
ALTER TABLE
```

## 自动路由模式

通过设置htap\_router\_mode的值为'auto'，可开启自动路由模式。自动路由模式可根据相关表、列是否创建IMCV，以及行列计划代价高低，自动选择行、列、行列混合执行计划。

示例：

```
htapdb=# SET htap_router_mode = 'auto';
SET

htapdb=# SHOW htap_router_mode;
 htap_router_mode

auto
(1 row)

--行存计划代价更低，选择行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_small;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_small
(3 rows)

--列存计划代价更低，选择列存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM test_large;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on test_large
(4 rows)

--查询目标列未创建IMCV，选择行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_large;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_small
(3 rows)
```



## 强制列存模式

以AP型负载为主的情况下，可通过设置htap\_router\_mode的值为'column'，打开强制列存模式。该模式下，当查询相关表和列存在IMCV时，在满足列存查询的条件下，将无视代价高低，优先选择列存执行计划。

示例：

```
htapdb=# SET htap_router_mode = 'column';
SET
htapdb=# SHOW htap_router_mode;
 htap_router_mode

column
(1 row)

--表中所有列均创建IMCV，执行列存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_small;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
 Node/s: All datanodes
 -> Imcv Scan on test_small
(4 rows)

--查询目标列创建IMCV，执行列存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM test_large;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
 Node/s: All datanodes
 -> Imcv Scan on test_large
(4 rows)

--查询目标列未创建IMCV，执行行存扫描。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_large;
QUERY PLAN

Streaming (type: GATHER)
 Node/s: All datanodes
 -> Seq Scan on test_small
(3 rows)
```

## 强制行存模式

以TP型负载为主的情况下，可通过设置htap\_router\_mode的值为'row'，打开强制行存模式。该模式下，无论是否满足列存执行计划的条件，均选择行存计划。

示例：

```
htapdb=# SET htap_router_mode = 'row';
SET
htapdb=# SHOW htap_router_mode;
 htap_router_mode

row
(1 row)

--强制执行行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_small;
QUERY PLAN

Streaming (type: GATHER)
 Node/s: All datanodes
```

```
-> Seq Scan on test_small
(3 rows)

--强制执行行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM test_large;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_large
(3 rows)

--强制执行行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM test_large;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_large
(3 rows)
```

## Hint 指定路由模式

单条查询可使用Hint指定目标表的列存扫描执行计划，即当指定表存在列存并且加载列满足查询要求时，强制该表执行列存扫描。例如，基于Hint语法 “/\* + imcvscan (tbl) \*/” 实现。

示例：

```
htapdb=# SET htap_router_mode = 'row';
SET
htapdb=# SHOW htap_router_mode;
 htap_router_mode

row
(1 row)

--虽然行存计划代价更低，但基于Hint选择列存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT /*+imcvscan (test_small)*/ * FROM test_small;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on test_small
(4 rows)

--基于Hint强制选择列存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT /*+imcvscan (test_large)*/ id,dept_id FROM test_large;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on test_large
(4 rows)

--Hint指定表加速列不满足查询需求，选择行存计划。
htapdb=# EXPLAIN (COSTS OFF) SELECT /*+imcvscan (test_large)*/ * FROM test_large;
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on test_large
(3 rows)

--Hint指定test_large执行列存扫描。
htapdb=# EXPLAIN (COSTS OFF) SELECT /*+imcvscan (test_large)*/ test_small.id, test_large.dept_id FROM
```

```
test_small, test_large WHERE test_small.id = test_large.id;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
 Node/s: All datanodes
 -> Vector Ca Hash Join
 Hash Cond: (test_large.id = test_small.id)
 -> Imcv Scan on test_large
 -> Vector Adapter(type: BATCH MODE)
 -> Seq Scan on test_small
(8 rows)

--Hint指定连接两表均执行列存扫描。
htapdb=# EXPLAIN (COSTS OFF) SELECT /*+imcvscan (test_large) imcvscan (test_small)*/ test_small.id,
test_large.dept_id FROM test_small, test_large WHERE test_small.id = test_large.id;
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
 Node/s: All datanodes
 -> Vector Ca Hash Join
 Hash Cond: (test_small.id = test_large.id)
 -> Imcv Scan on test_small
 -> Imcv Scan on test_large
(7 rows)

htapdb=# DROP TABLE test_large;
DROP TABLE

htapdb=# DROP TABLE test_small;
DROP TABLE
```

### 7.20.2.6 列存数据维护

#### 列存数据加载

列存数据加载是指首次根据行存数据生成该表对应的列存数据。在启用异步加载模式后，通过DDL命令为表创建IMCV时，系统仅会建立列存的元数据结构，并不会立刻生成实际的列存数据。

系统提供两种列存数据的加载方式：

- 后台HTAP Worker自动加载  
系统会在后台调度HTAP Worker，对目标表自动完成列存数据的构建。该方式无需人工干预，但其执行时间可能受到后台HTAP Worker的线程数量（由GUC参数htap\_flush\_workers决定）、系统负载以及调度策略的影响。
- 手动触发加载  
用户可直接调用gs\_imcv\_flush()接口，立即开始对指定表进行列存数据加载。当后台加载未及时完成时，用户可主动执行该接口加速加载进程。加载效率与参数设置的并行度有关。

#### 说明

- HTAP Worker数量由GUC参数htap\_flush\_workers决定，设置方法请参见《参考》中“数据库运行参数说明 > GUC参数说明 > HTAP混合负载”章节。
- gs\_imcv\_flush是用于列存数据加载/重建的系统函数，使用方法请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > HTAP系统函数”章节。

自动加载示例：

```
-- 创建普通表并插入数据。
htapdb=# DROP TABLE IF EXISTS htap_gs_imcv_flush_nopart;
```

```
htapdb=# CREATE TABLE htap_gs_imcv_flush_nopart(
 c1 bigint,
 c2 varchar(256)
) WITH (STORAGE_TYPE=USTORE) DISTRIBUTE BY HASH(c1);
CREATE TABLE

htapdb=# INSERT INTO htap_gs_imcv_flush_nopart (c1, c2)
SELECT
 g AS c1,
 'data_' || g AS c2
FROM generate_series(1, 256000) AS g;
INSERT 0 256000

-- 创建一级分区表并插入数据。
htapdb=# DROP TABLE IF EXISTS htap_gs_imcv_flush_part;
htapdb=# CREATE TABLE htap_gs_imcv_flush_part (
 c1 bigint,
 c2 bigint,
 c3 varchar(256)
) DISTRIBUTE BY HASH(c1)
PARTITION BY RANGE (c1)(
 PARTITION htap_gs_imcv_flush_part_p1 VALUES LESS THAN (2),
 PARTITION htap_gs_imcv_flush_part_p2 VALUES LESS THAN (3)
);
CREATE TABLE

htapdb=# INSERT INTO htap_gs_imcv_flush_part (c1, c2, c3)
SELECT
 CASE
 WHEN g <= 256000 THEN 1
 ELSE 2
 END AS c1,
 g AS c2,
 'data_' || g AS c3
FROM generate_series(1, 512000) AS g;

-- 为非分区表和分区表创建IMCV。
htapdb=# ALTER TABLE htap_gs_imcv_flush_nopart COLVIEW;
ALTER TABLE
htapdb=# ALTER TABLE htap_gs_imcv_flush_part COLVIEW;
ALTER TABLE

-- DN节点查看IMCV信息（其中，rowgroup_num为表的实际行组数量，populated_rowgroup_num为表已完成加载的行组数量）。
htapdb=# SELECT rel_name, partition_name, rowgroup_num, populated_rowgroup_num FROM
gs_htap_imcv_info();
 rel_name | partition_name | rowgroup_num | populated_rowgroup_num
-----+-----+-----+-----
 htap_gs_imcv_flush_part | htap_gs_imcv_flush_part_p1 | 2 | 0
 htap_gs_imcv_flush_part | | 0 | 0
 htap_gs_imcv_flush_part | htap_gs_imcv_flush_part_p2 | 2 | 0
 htap_gs_imcv_flush_nopart | | 2 | 2

-- 等待一段时间后查看IMCV信息（等待后台加载完成）。
htapdb=# SELECT rel_name, partition_name, rowgroup_num, populated_rowgroup_num FROM
gs_htap_imcv_info();
 rel_name | partition_name | rowgroup_num | populated_rowgroup_num
-----+-----+-----+-----
 htap_gs_imcv_flush_part | htap_gs_imcv_flush_part_p1 | 2 | 2
 htap_gs_imcv_flush_part | | 0 | 0
 htap_gs_imcv_flush_part | htap_gs_imcv_flush_part_p2 | 2 | 2
 htap_gs_imcv_flush_nopart | | 2 | 2
```

手动加载示例：

```
-- 重新创建非分区表htap_gs_imcv_flush_nopart与一级分区表htap_gs_imcv_flush_part并插入数据，创建方式与上文示例相同。
-- 1.指定关系加载列存数据。
-- (1)指定非分区表加载列存数据。
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_nopart');
```

```
total | success | error | skip
-----+-----+-----+-----
2 | 2 | 0 | 0
(1 row)
-- (2)指定分区表加载列存数据。
-- (2.1)指定一级分区表加载列存数据。
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', 'htap_gs_imcv_flush_part_p1');
total | success | error | skip
-----+-----+-----+-----
2 | 2 | 0 | 0
(1 row)
-- (2.2)指定基表加载列存数据（p2加载成功，p1已加载过，本次加载跳过）。
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', '');
total | success | error | skip
-----+-----+-----+-----
4 | 2 | 0 | 2
(1 row)
-- 2.指定加载列存数据的并行度。
htapdb=# ALTER TABLE htap_gs_imcv_flush_part NOCOLVIEW;
ALTER TABLE
htapdb=# ALTER TABLE htap_gs_imcv_flush_part COLVIEW;
ALTER TABLE
htapdb=# \timing on
Timing is on.
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', '', 2);
total | success | error | skip
-----+-----+-----+-----
4 | 4 | 0 | 0
(1 row)

Time: 406.391 ms
htapdb=# \time off
Timing is off.
htapdb=# ALTER TABLE htap_gs_imcv_flush_part NOCOLVIEW;
ALTER TABLE
htapdb=# ALTER TABLE htap_gs_imcv_flush_part COLVIEW;
ALTER TABLE
htapdb=# \timing on
Timing is on.
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', '', 2);
total | success | error | skip
-----+-----+-----+-----
4 | 4 | 0 | 0
(1 row)

Time: 332.194 ms
htapdb=# \time off
Timing is off.
```

## 列存数据重建

在完成列存加载之后，后续所有的DML（INSERT/UPDATE/DELETE）都会产生新的增量数据。这些增量由实时事务单元TMU维护，但不会即时同步到已经加载的列存数据中。

当这些增量积累过多时，列存数据会逐渐落后于实际行存内容，从而影响列存扫描性能。此时需要进行列存数据重建，把累积的增量同步到列存中，保持列存数据的新鲜度和查询性能。

系统同样提供两种重建方式：

- 后台HTAP Worker自动重建

当增量数据达到一定规模，或者距离上一次重建一定时间后，系统在后台自动触发列存重建任务。重建耗时同样会受到后台线程数量和调度的影响。

 说明

后台自动重建的触发条件：

- 增量数据所涉及的页面数达到一定比例，由GUC参数htap\_flush\_pagenum控制。
- 距离上次重建达到一定的时间间隔，由 GUC 参数htap\_flush\_interval控制。

设置方法请参见《参考》中“数据库运行参数说明 > GUC参数说明 > HTAP混合负载”章节。

- 手动触发重建

如果后台重建未及时执行，或者用户希望立即更新列存内容，可直接调用gs\_imcv\_flush()接口手动触发重建。重建效率与参数设置的并行度有关。

自动重建示例：

```
-- 查看同步时间间隔和脏页阈值。
htapdb=# SHOW htap_flush_interval;
 htap_flush_interval

5min
(1 row)
htapdb=# SHOW htap_flush_pagenum;
 htap_flush_pagenum

512
(1 row)
-- 向一级分区表htap_gs_imcv_flush_part中更新数据。
htapdb=# UPDATE htap_gs_imcv_flush_part SET c3 = 'abcd' WHERE c2 IN (2, 20002);
UPDATE 2
-- 查看脏页数（未达重建条件的脏页阈值）。
htapdb=# SELECT rel_name, sum(dirty_page_count) as dirty_page_count FROM
gs_global_htap_tmu_chunk_meta('htap_gs_imcv_flush_part',0) GROUP BY rel_name;
 rel_name | dirty_page_count
-----+-----
 htap_gs_imcv_flush_part | 2
(1 row)
-- 等待5min后查看脏页数，此时后台重建完成（达到设置的重建间隔后触发后台重建完成）。
htapdb=# SELECT rel_name, sum(dirty_page_count) as dirty_page_count FROM
gs_global_htap_tmu_chunk_meta('htap_gs_imcv_flush_part',0) GROUP BY rel_name;
 rel_name | dirty_page_count
-----+-----
 htap_gs_imcv_flush_part | 0
(1 row)
-- 向一级分区表htap_gs_imcv_flush_part中插入足够数据（大于脏页阈值）。
htapdb=# INSERT INTO htap_gs_imcv_flush_part (c1, c2, c3)
SELECT 1, 1, 'c3' FROM generate_series(1, 120000);
INSERT 120000
-- 查看脏页数。
htapdb=# SELECT rel_name, sum(dirty_page_count) as dirty_page_count FROM
gs_htap_tmu_chunk_meta('htap_gs_imcv_flush_part',0) GROUP BY rel_name;
 rel_name | dirty_page_count
-----+-----
 htap_gs_imcv_flush_part | 557
(1 row)
-- 等待后台自动重建完成后再次查看脏页数。
htapdb=# SELECT rel_name, sum(dirty_page_count) as dirty_page_count FROM
gs_htap_tmu_chunk_meta('htap_gs_imcv_flush_part',0) GROUP BY rel_name;
 rel_name | dirty_page_count
-----+-----
 htap_gs_imcv_flush_part | 0
(1 row)
```

手动重建示例：

```
-- 使用gs_imcv_flush()接口手动触发重建的方式与手动触发加载的方式相同。
-- 向一级分区表htap_gs_imcv_flush_part中插入数据。
htapdb=# UPDATE htap_gs_imcv_flush_part SET c3 = 'efgh' WHERE c1=1;
```

```
UPDATE 24
-- 手动触发非强制全表重建（第四个参数缺省或为0，只重建有脏页标记的rowgroup）。
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', '', 2);
total | success | error | skip
-----+-----+-----+-----
4 | 1 | 0 | 3
-- 手动触发强制全表重建（第四个参数为1，基于行存重新构建全部rowgroup）。
htapdb=# SELECT * FROM gs_imcv_flush ('htap_gs_imcv_flush_part', '', 2,1);
total | success | error | skip
-----+-----+-----+-----
4 | 4 | 0 | 0
(1 row)

htapdb=# DROP TABLE htap_gs_imcv_flush_nopart;
DROP TABLE

htapdb=# DROP TABLE htap_gs_imcv_flush_part;
DROP TABLE
```

### 7.20.2.7 备机能力

分布式下HTAP不支持备机。

设置htap\_deploy\_mode参数后，实际生效值为'primary'，即仅支持主机开启HTAP功能。

### 7.20.2.8 列存数据磁盘扩展

HTAP提供列存数据磁盘扩展功能，即当实际列存数据空间占用大于HTAP内存上限时，超出部分通过磁盘空间管理。该功能可通过设置参数htap\_memctl\_policy='swap'开启。

该功能同时配套冷热数据管理来尽可能减少访问数据的磁盘I/O开销。列存元数据始终存储在内存中，当HTAP可用内存不足时，后台将访问频次较低的列存数据持久化到磁盘上，当内存充足时，则尽可能将磁盘中的热点数据重新加载到内存中维护。列存扫描查询过程中不会进行冷热数据的换入换出，而是由后台线程根据数据冷热进行冷热数据的换入换出。

示例：

```
--查看GUC参数。
htapdb=# SHOW htap_memctl_policy;
htap_memctl_policy

swap
(1 row)

htapdb=# SHOW htap_max_mem_size;
htap_max_mem_size

100MB
(1 row)

htapdb=# CREATE TABLE t1(c1 int, c2 int, c3 varchar(512)) WITH (storage_type=ustore);
CREATE TABLE
htapdb=# INSERT INTO t1 VALUES(
 generate_series(0,2000000),
 generate_series(0,2000000),
 repeat('abcabcabc', 50)
);
INSERT 0 2000001

htapdb=# ALTER TABLE t1 COLVIEW;
ALTER TABLE
```

```
htapdb=# SELECT 1 FROM gs_imcv_flush('t1');
?column?

 1
(1 row)

htapdb=# \x on
Expanded display is on.
--查看溢出的磁盘文件状态。
htapdb=# SELECT * FROM gs_global_htap_file_status('htapdb');
-[RECORD 1]-----+-----
node_name | dn_6003_6004
db_oid | 42055
file_name | global/imcu/42055/2_imcu
file_id | 2
extent_size | 1
allocated_blocks | 422
total_blocks | 16384
high_water_mark | 1043
utilization | .0257568
ext |
-[RECORD 2]-----+-----
node_name | dn_6001_6002
db_oid | 42085
file_name | global/imcu/42085/2_imcu
file_id | 2
extent_size | 1
allocated_blocks | 701
total_blocks | 16384
high_water_mark | 922
utilization | .0427856
ext |

htapdb=# DROP TABLE t1;
DROP TABLE
```

### 7.20.2.9 列存数据压缩

HTAP提供数据压缩能力，提升内存资源使用率。该功能可通过htap\_compression\_methods参数开启。

由于列存扫描需要对压缩数据进行解压，因此开启压缩功能后，相较于未开启压缩时，对列存扫描性能有一定影响。

针对不同的数据类型，支持自适应选择更高效的压缩算法，当前支持的算法类型有Dict、Delta+RLE、LZ4。

示例：

```
--查看GUC参数。
htapdb=# SHOW htap_compression_methods;
 htap_compression_methods

all
(1 row)

--创建行存表并插入数据。
htapdb=# CREATE TABLE test_compression(id int, a numeric(15,2), b varchar(100));
CREATE TABLE
htapdb=# INSERT INTO test_compression SELECT id,FLOOR(id/100)+0.5, SUBSTRING(md5(id),0.5) FROM
GENERATE_SERIES(1,1000) AS t(id);
INSERT 0 1000

--创建IMCV。
htapdb=# ALTER TABLE test_compression COLVIEW;
ALTER TABLE
```



```
--通过gs_imcu_meta系统函数查询test_compression表每列的压缩算法和压缩率。
htapdb=# SELECT node_name, rel_name,imcu_size, imcu_data_size, compressed_data_size,
compressed_ratio FROM gs_global_imcu_meta('test_compression', NULL);
 node_name | rel_name | imcu_size | imcu_data_size | compressed_data_size | compressed_ratio
-----+-----+-----+-----+-----+-----
 datanode1 | test_compression | 5075 | 8008 | 1075 | 7.4493
 datanode1 | test_compression | 6034 | 4008 | 2034 | 1.9705
 datanode1 | test_compression | 4083 | 14008 | 83 | 168.771
 datanode1 | test_compression | 36048 | 36008 | 32048 | 1.12356
(4 rows)

htapdb=# DROP TABLE test_compression;
DROP TABLE
```

## 7.21 在线 DDL

在线DDL支持对正常业务影响较小的情况下进行DDL，减少对业务的影响，提高业务运行的连续性和数据库的可用性。部分在线DDL只需要修改元数据，不涉及重建数据，操作时长较快，对业务影响较小。传统DDL操作若涉及重建数据，例如修改列类型、移动表空间、VACUUM FULL等操作，会对操作的表加长时间的排他锁，阻塞对该表的并发读写操作，影响用户业务运行。阻塞时长与数据量成正比，当数据量较大时，阻塞时长可达数小时、甚至数天。相比之下，在线DDL不会长时间加排他锁，执行中允许对该表的并发读写操作，减少对用户业务运行影响。

### 说明

- 使用DDL功能时，数据库需正常运行。

## 使用在线 DDL 操作过程中进行并发读写操作

### 步骤1 创建表。

```
gaussdb=# CREATE TABLE t1(c1 INT,c2 INT);
gaussdb=# INSERT INTO t1 VALUES(generate_series(1,10), generate_series(1,10));

-- 查看表格
gaussdb=# \d+ t1
 Table "public.t1"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 c1 | integer | | plain | |
 c2 | integer | | plain | |
Has OIDs: no
Distribute By: HASH(c1)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
```

### 步骤2 历史版本已支持的在线DDL操作（仅部分示例，全部历史版本支持的在线DDL操作请参见《参考》中“SQL参考 > SQL语法 > Online DDL及其操作”）。

- 在线添加列。

```
gaussdb=# ALTER TABLE t1 ADD COLUMN c3 integer;
ALTER TABLE
```
- 在线删除列。

```
gaussdb=# ALTER TABLE t1 drop COLUMN c3;
ALTER TABLE
```
- 在线重命名表。

```
gaussdb=# ALTER TABLE t1 RENAME TO t2;
ALTER TABLE
gaussdb=# ALTER TABLE t2 RENAME TO t1;
ALTER TABLE
```

**步骤3 设置ONLINE关键字并进行相关操作。**

- 在线修改列类型。  
gaussdb=# ALTER TABLE ONLINE t1 ALTER c2 TYPE BIGINT;  
ALTER TABLE
- 在线添加非空约束。  
gaussdb=# ALTER TABLE ONLINE t1 ALTER c1 SET NOT NULL;  
ALTER TABLE
- 在线添加检查约束。  
gaussdb=# ALTER TABLE ONLINE t1 ADD CONSTRAINT c2\_k CHECK(c2 > 0);  
ALTER TABLE
- 在线修改添加唯一约束。  
gaussdb=# ALTER TABLE ONLINE t1 ADD UNIQUE(c1);  
NOTICE: ALTER TABLE / ADD UNIQUE will create implicit index "t1\_c1\_key" for table "t1"  
ALTER TABLE
- 在线表级移动表空间。  
gaussdb=# CREATE TABLESPACE tbs RELATIVE LOCATION 'tablespace1/tbs';  
CREATE TABLESPACE  
gaussdb=# ALTER TABLE ONLINE t1 SET TABLESPACE tbs;  
ALTER TABLE
- 以上在线DDL操作中，可进行并发读写操作。  
gaussdb=# INSERT INTO t1 VALUES(11, 11);  
INSERT 0 1

**步骤4 查看表结构。**

```
gaussdb=# \d+ t1
 Table "public.t1"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1 | integer | not null | plain | |
c2 | bigint | | plain | |
Indexes:
 "t1_c1_key" UNIQUE CONSTRAINT, ubtree (c1) WITH (storage_type=USTORE) TABLESPACE pg_default
Check constraints:
 "c2_k" CHECK (c2 > 0)
Has OIDs: no
Tablespace: "tbs"
Distribute By: HASH(c1)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
```

**步骤5 使用VACUUM FULL在线回收表空间。**

```
gaussdb=# VACUUM FULL t1 ONLINE;
VACUUM
```

----结束

## 使用分区在线 DDL 操作过程中进行并发读写操作

**步骤1 创建分区表和普通表。**

```
gaussdb=# CREATE TABLE t1(c1 INT,c2 INT) PARTITION BY RANGE (c2)
(
 PARTITION P1 values less than (5),
 PARTITION P2 values less than (10),
 PARTITION P3 values less than (101)
);
gaussdb=# INSERT INTO t1 VALUES(generate_series(1,100), generate_series(1,100));
gaussdb=# CREATE TABLE t1_exchange(c1 INT,c2 INT);
gaussdb=# INSERT INTO t1_exchange VALUES(generate_series(10,50), generate_series(10,50));
-- 查看表格
gaussdb=# \d+ t1
 Table "public.t1"
Column | Type | Modifiers | Storage | Stats target | Description
```

```
-----+-----+-----+-----+-----+-----+
c1 | integer | | plain | |
c2 | integer | | plain | |
Partition By RANGE(c2)
Number of partitions: 3 (View pg_partition to check each partition range.)
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
gaussdb=# \d+ t1_exchange
 Table "public.t1_exchange"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+
c1 | integer | | plain | |
c2 | integer | | plain | |
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
```

### 步骤2 设置ONLINE关键字并进行相关操作。

- 在线合并分区。

```
gaussdb=# ALTER TABLE ONLINE t1 MERGE PARTITIONS p1, p2 into PARTITION p2_new;
ALTER TABLE
gaussdb=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='t1'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries | oid
-----+-----+-----+
p2_new | {10} | 4397673
p3 | {101} | 4397634
(2 rows)
```

- 在线切割分区。

```
gaussdb=# ALTER TABLE ONLINE t1 SPLIT PARTITION p2_new AT (5) INTO (PARTITION P1,
PARTITION P2);
ALTER TABLE
gaussdb=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='t1'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries | oid
-----+-----+-----+
p1 | {5} | 4397706
p2 | {10} | 4397707
p3 | {101} | 4397634
(3 rows)
```

- 在线交换分区。

```
gaussdb=# ALTER TABLE ONLINE t1 EXCHANGE PARTITION (p3) WITH TABLE t1_exchange;
ALTER TABLE
gaussdb=# SELECT COUNT(*) FROM t1 PARTITION (p3);
count

41
(1 row)
```

- 在线移动分区表空间。

```
gaussdb=# CREATE TABLESPACE tbs RELATIVE LOCATION 'tablespace1/tbs';
CREATE TABLESPACE
gaussdb=# ALTER TABLE ONLINE t1 MOVE PARTITION p1 TABLESPACE tbs;
ALTER TABLE
gaussdb=# SELECT relname, spcname FROM pg_partition t1, pg_tablespace t2 WHERE
T1.reltablespace=t2.oid and t1.parentid='t1'::regclass;
 relname | spcname
-----+-----+
p1 | tbs
(1 row)
```

- 以上在线DDL操作中，可进行并发读写操作。

```
gaussdb=# INSERT INTO t1 VALUES(11, 11);
INSERT 0 1
gaussdb=# INSERT INTO t1_exchange VALUES(51, 51);
INSERT 0 1
```

### 步骤3 查看表定义。

```
gaussdb=# \d+ t1
 Table "public.t1"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 c1 | integer | | plain | |
 c2 | integer | | plain | |
Partition By RANGE(c2)
Number of partitions: 3 (View pg_partition to check each partition range.)
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off
gaussdb=# \d+ t1_exchange
 Table "public.t1_exchange"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 c1 | integer | | plain | |
 c2 | integer | | plain | |
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off

gaussdb=# SELECT relname, reltablespace, boundaries FROM pg_partition;
 relname | reltablespace | boundaries
-----+-----+-----
 t1 | | 0 |
 p1 | | 4397520 | {5}
 p2 | | 0 | {10}
 p3 | | 0 | {101}
(4 rows)
```

#### 步骤4 使用VACUUM FULL在线回收分区空间。

```
gaussdb=# VACUUM FULL t1 PARTITION (p1) ONLINE;
VACUUM
```

----结束

## 7.22 bucket 分布表

### bucket 分布表简介

bucket分布表底层采用段页式存储，按照用户指定的不同分布方式，对用户数据按bucket进行物理切分。

目前bucket分布表支持Astore hash分布表和Ustore hash分布表；不支持复制表、临时表（本地临时表和全局临时表）、unlogged表。

### bucket 分布表的优势

当用户的数据量增加以至于需要进行集群扩容时，可以精确控制需要搬迁的bucket文件，针对这些bucket动态的产生追增日志流，追增日志流独立于系统WAL日志，不会对正常的业务流产生影响，并且只存在于扩容阶段。

### 7.22.1 hashbucket

#### hashbucket 表简介

hashbucket表将数据根据分布列的hash值进行分片，具有相同hash值的数据存储在同一组bucket段页式文件中，每一组bucket文件之间完全解耦。

### 7.22.1.1 创建 hashbucket 表

在一个数据库中创建第一张hashbucket表前，需要先将当前database绑定到集群默认的node group上。

**步骤1** （可选）通过给DATABASE绑定node group生成CN上hash分布和DN上bucket的映射关系，语法如下：

```
ALTER DATABASE databasename TO GROUP groupname;
```

语法详情请参见《参考》中“SQL参考 > SQL语法 > A > ALTER DATABASE”章节中相关描述。

- 查询NODE GROUP。

```
gaussdb=# select group_name from pgxc_group where is_installation = true;
group_name

group_version1
(1 row)
```

- 绑定到集群默认的NODE GROUP。具体语法请参见《参考》中“SQL参考 > SQL语法 > A > ALTER DATABASE”章节和“SQL参考 > SQL语法 > C > CREATE NODE GROUP”章节。

```
gaussdb=# ALTER DATABASE postgres TO GROUP group_version1;
ALTER DATABASE
```

#### 说明

如果数据库第一次创建hashbucket表前不绑定NODE GROUP，则该建表命令会自动绑定集群当前默认的NODE GROUP上。

```
gaussdb=# CREATE TABLE hashbucket_table1(c1 VARCHAR2 (30)) WITH(storage_type=ustore,
hashbucket=on) distribute by hash(c1);
NOTICE: This is the first time ever to create a hashbucket table in this database. The database
will be automatically altered to the installation node group "group_version1".
NOTICE: hashbucket table need segment storage, set segment to on by default
CREATE TABLE
```

**步骤2** 创建hashbucket表语法与创建普通表CREATE TABLE语法一致，在建表时需要额外设置参数hashbucket=on，且显式或隐式指定hash分布列。

- 存储参数hashbucket默认值为off，取值范围：
  - on：表示创建hashbucket表，
  - off：表示非hashbucket表。
- 指定hash分布列：hashbucket表要求至少有一列可以作为hash分布列，该列决定数据分布的bucket。hash分布列可以在建表时通过子句DISTRIBUTE BY hash(column\_name)显式指定，不显式指定时会隐式地选择第一个满足hash分布条件的列。

#### ----结束

- 创建hashbucket普通表。

```
gaussdb=# CREATE TABLE hashbucket_table1(c1 VARCHAR2 (30)) WITH(storage_type=ustore,
hashbucket=on) DISTRIBUTE BY hash(c1);
NOTICE: hashbucket table need segment storage, set segment to on by default
CREATE TABLE
```

- 创建hashbucket分区表。

```
gaussdb=# CREATE TABLE hashbucket_partiton_table_range1 (
c1 int,
c2 bigint NOT NULL,
c3 VARCHAR2 (30)
) WITH(storage_type=ustore, hashbucket=on) DISTRIBUTE BY hash(c3)
PARTITION BY RANGE(c1)(
```

```
PARTITION p1 VALUES LESS THAN (100000),
PARTITION p2 VALUES LESS THAN (200000),
PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
CREATE TABLE
```

具体语法请参见《参考》中“SQL参考 > SQL语法 > C > CREATE TABLE”章节和“SQL参考 > SQL语法 > C > CREATE TABLE PARTITION”章节。

#### 说明

- 在每个数据库内创建第一张hashbucket表比普通表会更慢，这是由于创建第一张hashbucket表时会创建1024组段页式文件。
- 多张表并发创建时，需要等第一张表的段页式文件创建完成后，其他表才会创建成功。
- 设置GUC参数enable\_hashbucket为on后，可以不指定WITH(hashbucket=on)，默认创建hashbucket表。详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 其它选项”章节中enable\_hashbucket参数的介绍。

### 7.22.1.2 bucket 索引

hashbucket表支持创建bucket全局索引（跨bucket索引，cross-bucket index，CBI）和bucket本地索引（local-bucket index，LBI）。hashbucket表通过指定索引存储参数crossbucket为on/off创建CBI/LBI索引，不指定crossbucket时默认创建LBI索引。

#### 7.22.1.2.1 bucket 索引推荐使用场景

CBI索引以普通段页式方式存储，可以理解为所有bucket索引元组全部创建到统一的一个索引中；LBI索引以hashbucket段页式方式存储，可以理解为每个bucket各自创建一个子索引，由LBI索引统一管理。

LBI为hashbucket存储提供bucket内的索引能力；CBI为hashbucket存储提供跨bucket的索引能力，在创建索引时，一个索引文件可以对应表级或分区级的所有bucket文件。CBI索引可以为指定粒度的表提供跨bucket的索引扫描，避免顺序扫描当前表下的大量bucket文件才能查到目标元组，提升查询性能。CBI、LBI推荐使用场景：

- 业务语句的hash分布列都在同一个bucket上，或者语句条件包含hash分布列时，剪枝后扫描结果只涉及少数bucket时，需要依靠bucket内的索引能力，推荐使用LBI；
- 业务语句的条件不包含hash分布列，或按条件过滤后仍需要扫描较多bucket时，需要依靠跨bucket的索引能力，推荐使用CBI。

### bucket 索引支持情况

hashbucket分区表索引根据是否为GPI索引/是否为CBI索引组成四种组合，内核支持情况见下表：

| hashbucket分区表索引类型           | CBI(cross-bucket index) | LBI(local-bucket index) |
|-----------------------------|-------------------------|-------------------------|
| GPI(global partition index) | yes                     | yes                     |
| LPI(local partition index)  | yes                     | yes                     |

- 创建hashbucket表的LBI索引。  
gaussdb=# CREATE INDEX idx1 ON hashbucket\_table1(c1) WITH(crossbucket=off);  
CREATE INDEX
- 创建hashbucket表的CBI索引。  
gaussdb=# CREATE INDEX idx2 ON hashbucket\_table2(c1) WITH(crossbucket=on);  
CREATE INDEX
- 创建hashbucket分区表的LPI+LBI索引。  
gaussdb=# CREATE INDEX idx1\_tb1 ON hashbucket\_partiton\_table\_range1(c1) LOCAL  
WITH(crossbucket=off);  
CREATE INDEX
- 创建hashbucket分区表的LPI+CBI索引。  
gaussdb=# CREATE INDEX idx2\_tb1 ON hashbucket\_partiton\_table\_range1(c2) LOCAL  
WITH(crossbucket=on);  
CREATE INDEX
- 创建hashbucket分区表的GPI+LBI索引。  
gaussdb=# CREATE INDEX idx1\_tb1 ON hashbucket\_partiton\_table\_range1(c1) GLOBAL  
WITH(crossbucket=off);  
CREATE INDEX
- 创建hashbucket分区表的GPI+CBI索引。  
gaussdb=# CREATE INDEX idx2\_tb1 ON hashbucket\_partiton\_table\_range1(c2) GLOBAL  
WITH(crossbucket=on);  
CREATE INDEX

具体语法请参见《参考》中“SQL参考 > SQL语法 > C > CREATE INDEX”章节。

### 7.22.1.3 hashbucket 表特性规格约束

#### hashbucket 表约束

- hashbucket表只支持分布式Astore/Ustore行存储、hash分布键，不支持RANGE、LIST、MURMURhash分布，支持分区表，不支持二级分区表，不支持临时表、unlogged表，不支持外表、不支持复制表。
- hashbucket表不支持GIN/GIST索引。
- hashbucket表绑定段页式存储，即hashbucket=on隐含segment=on。
- 由于hashbucket表新增了bucket维度的文件存储信息，出于性能考虑，需要适当调大segment\_buffers。
- hashbucket表不支持细粒度备份恢复的表级备份，支持细粒度表级恢复，支持细粒度库级备份和恢复，支持集群级备份和恢复，支持pitr。
- hashbucket表不支持在线DDL操作，不支持M兼容性DDL、DML操作。
- 小表（即普通表小于8MB，分区表小于8MB \* 分区数量）不建议创建成hashbucket表，否则顺序扫描性能会较普通表劣化2倍。hashbucket表由于使用段页式存储，且物理文件存储层多了bucket维度，空表或小数据量的表存在一定的存储空间预占用。单DN 10GB以上数据存储场景相较于普通表基本无空间膨胀。
- 不支持创建hashbucket类型的全量物化视图；不支持针对hashbucket表创建增量物化视图。
- 不支持Ustore hashbucket表和Astore hashbucket表库级混用。
- 当带有GPI索引的hashbucket分区表执行ALTER TABLE ... ADD/DROP/TRUNCATE/SPLIT PARTITION ... UPDATE GLOBAL INDEX命令时，被删除的分区会残留少量元数据页面，这些残留页面依赖于段页式自动残留清理特性开启状态下进行删除，即GUC参数enable\_auto\_segment\_remain\_cleanup=on（默认值为on）。建议在hashbucket分区表涉及以上DDL操作时维持此参数为on。

- 不支持使用枚举类型、复合类型、复合数组类型、表类型、特殊类型（"regproc"、"pg\_node\_tree"、"regprocedure"、"regoper"、"regoperator"、"regclass"、"regtype"、"\_regprocedure"、"\_regoper"、"\_regoperator"、"\_regclass"、"\_regtype"、"regconfig"、"regdictionary"、"\_regconfig"和"\_regdictionary"）作为列来创建或修改hashbucket表。对于已经创建成功且包含上述类型列的hashbucket表，不支持扩容操作。

## Ustore hashbucket 表约束

- Ustore hashbucket表继承hashbucket表的所有约束。
- Ustore hashbucket表继承Ustore表的约束。不支持在事务块/存储过程中执行ALTER TABLE MERGE PARTITIONS的操作。在线并行创建/重建索引支持CBI、LBI索引，支持Ustore普通索引、GLOBAL索引、LOCAL索引，不支持PCR索引。
- Ustore hashbucket表不支持ubtree批量插入特性。
- Ustore hashbucket表不支持GSI在线创建索引。
- Ustore hashbucket表创建CBI索引不支持include。
- Ustore hashbucket表不支持表类型转换。

## hashbucket 表特性规格

hashbucket表（普通表/分区表）DML性能相比于普通表（普通表/分区表）性能劣化5%以内。

### 7.22.1.4 hashbucket 剪枝

在使用hashbucket表的时候，当select语句中谓词列和hashbucket表分布列相同时，会对hashbucket表进行剪枝操作，大幅缩减访问的bucket的数量。示例如下：

```
--准备环境。
gaussdb=# SET enable_fast_query_shipping=off;
gaussdb=# CREATE TABLE t1 (a int, b int, c int) WITH (hashbucket=on, storage_type=astore);
gaussdb=# INSERT INTO t1 VALUES(generate_series(1, 10), generate_series(1, 10));
gaussdb=# ANALYZE t1;
--展示执行计划。
gaussdb=# EXPLAIN (costs false,verbose on) SELECT count(*) FROM t1 WHERE a = 1;
 QUERY PLAN

Aggregate
 Output: pg_catalog.count(*)
 -> Streaming (type: GATHER)
 Output: (count(*))
 Node/s: datanode1
 -> Aggregate
 Output: count(*)
 -> Seq Scan on public.t1
 Output: a, b, c
 Distribute Key: a
 Filter: (t1.a = 1)
 Selected Buckets 1 of 1024 : 325
(12 rows)
```

可以看到计划中显示选择了325这个编号的bucket，则在实际执行的时候会直接读取这个bucket的数据，减少无意义的读取。

#### 说明

关于hashbucket的计划需要打开explain verbose才能显示。



## hashbucket 参数化路径剪枝

当选择参数化路径计划且hashbucket表作为内表时，若参数化的列是hashbucket表的分布列，会进行参数化路径的剪枝，动态削减访问bucket的数量。

```
--准备环境。
gaussdb=# SET enable_fast_query_shipping=off;
gaussdb=# CREATE TABLE t2 (a int, b int, c int) WITH (hashbucket=on, storage_type=astore);
gaussdb=# INSERT INTO t2 VALUES(generate_series(1, 10), generate_series(1, 10));
gaussdb=# CREATE INDEX index_t2 on t2(a, b);
gaussdb=# ANALYZE t2;
--展示执行计划。
gaussdb=# EXPLAIN (costs false,verbose on) SELECT /*+ nestloop_index(t2, index_t2)*/ FROM t1 JOIN t2
ON t1.b = t2.a;

QUERY PLAN

Streaming (type: GATHER)
 Output: t1.a, t1.b, t1.c, t2.a, t2.b, t2.c
 Node/s: All datanodes
 -> Nested Loop
 Output: t1.a, t1.b, t1.c, t2.a, t2.b, t2.c
 -> Streaming(type: REDISTRIBUTE)
 Output: t1.a, t1.b, t1.c
 Distribute Key: t1.b
 Spawn on: All datanodes
 Consumer Nodes: All datanodes
 -> Seq Scan on public.t1
 Output: t1.a, t1.b, t1.c
 Distribute Key: t1.a
 Selected Buckets 1024 of 1024 : all
 -> Index Scan using index_t2 on public.t2
 Output: t2.a, t2.b, t2.c
 Distribute Key: t2.a
 Index Cond: (t2.a = t1.b)
 Selected Buckets 1024 of 1024 : all (ppi-pruning)
(19 rows)
```

可以看到在Selected Buckets 这段有ppi-pruning标识，标识这个计划会执行参数化路径剪枝。

若需要知道具体的剪枝操作，可以在DN上执行explain analyze语句。

```
--直连DN。
gaussdb=# EXPLAIN (costs false,verbose on,analyze on) SELECT /*+ nestloop_index(t2, index_t2)*/ FROM t1
JOIN t2 on t1.b = t2.a;

QUERY PLAN

Nested Loop (actual time=4.743..10.928 rows=6 loops=1)
 Output: t1.a, t1.b, t1.c, t2.a, t2.b, t2.c
 -> Seq Scan on public.t1 (actual time=4.607..10.471 rows=6 loops=1)
 Output: t1.a, t1.b, t1.c
 Selected Buckets 1024 of 1024 : all
 -> Index Scan using index_t2 on public.t2 (actual time=0.124..0.132 rows=6 loops=6)
 Output: t2.a, t2.b, t2.c
 Index Cond: (t2.a = t1.b)
 Selected Buckets 1024 of 1024 : all (ppi-pruning) actual scan bucket num:6
Total runtime: 23.424 ms
(10 rows)
```

可以看到计划中有actual scan bucket num:6 这样一个字段，表示总共访问了6个bucket。

## 特性约束

- 支持bucket类型：hashbucket。
- 支持算子类型：indexscan、indexonlyscan、bitmapscan。

- 支持表达式类型：比较表达式（<, <=, =, >=, >）、逻辑表达式(and)。
- 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。
- 只支持LBI(local bucket index) 不支持 cross bucket index。
- 只支持单分布列。
- 外连接驱动hashbucket表时，只支持=条件。

## 7.22.2 bucket 分布表最佳实践

### 7.22.2.1 设置 segment\_buffers

segment\_buffers是用来设置段页式元数据页的内存大小的参数，为了提高性能需要设置足够大的buffer，保证活跃表的元数据页面都能缓存在buffer中，不被频繁置换。

hashbucket表基于段页式存储格式，由1024个bucket桶组成，每一个bucket桶由一组bucket段页式文件管理。其中DN上元数据页面主要包括bucket1号文件和段页式1号文件两部分，CN上主要为段页式1号文件。

#### bucket1 号文件的元数据页面

以下表或索引都会在每个bucket的1号文件中申请3个页面（main\_fork、fsm\_fork、vm\_fork）作为hashbucket元数据页面：

- 表：普通表、分区表的每个分区、toast表、toast索引。
- LBI索引：普通表的LBI索引、GLOBAL的LBI索引、LOCAL每个分区的LBI索引。

可以按照以下公式估算页面数：

单个DN上bucket桶个数 \* 3 \* ((普通表个数 + 分区表的分区个数) \* 3 + LBI索引个数 + LOCAL的LBI索引分区个数)

公式中第一个3表示main\_fork、fsm\_fork、vm\_fork，第二个3表示表、toast表、toast索引。为方便计算假设用户表都有toast表和toast索引。

#### 段页式 1 号文件的元数据页面

每个表或索引都会在段页式1号文件中申请1个页面作为元数据页面，可以按照以下公式估算页面数：

(普通表个数 + 分区表的分区个数) \* 3 + 索引个数 + LOCAL索引分区个数

公式中的3表示表、toast表、toast索引，为方便计算假设用户表都有toast表和toast索引。

### segment\_buffers 设置示例

假设活跃表共10张，其中4张普通表，6张分区表。普通表共8个索引，其中5个是LBI索引。每个分区表都有20个分区，分区表共12个索引，其中7个是LOCAL的LBI索引。假设集群有3个DN分片，每个DN上分配342个bucket桶。

估算元数据页面：

- bucket1号文件：

单个DN上bucket桶个数 \* 3 \* ((普通表个数 + 分区表的分区个数) \* 3 + 普通表LBI索引个数 + LOCAL的LBI索引分区个数)

```
= 342 * 3 * ((4 + 6 * 20) * 3 + 5 + 7 * 20)
= 530442个页面
≈ 4144MB
```

- 段页式1号文件：

```
(普通表个数 + 分区表的分区个数) * 3 + 索引个数 + LOCAL索引分区个数
= (4 + 6 * 20) * 3 + (8 + 12) + 7 * 20
= 532个页面
≈ 4MB
```

此示例中在内存足够的情况下，建议每个DN的 `segment_buffers` 设置为不小于4148MB，每个CN的`segment_buffers`不小于4MB。

#### 📖 说明

- 建议设置`segment_buffers`大于等于活跃表的元数据页面总空间，避免元数据页面buffer置换导致性能劣化。
- `hashbucket`表尤其是分区表占用的元数据页面比较多，建议根据以上公式估算并设置`segment_buffers`。
- 注意内存设置应满足条件：`data_replicate_buffer_size + segment_buffers + shared_buffers + wal_buffers + temp_buffers + maintenance_work_mem + work_mem + query_mem + (备机) wal_receiver_buffer_size < max_process_memory < 物理机内存大小`。在物理机总内存大小不足的情况下，建议`segment_buffers`设置为(`shared_buffers`的参考值 \* 估算的元数据量 / 总数据量)。
- 一些DDL操作需要加载大量元数据页面，如 `\d+` 操作会加载所有表的元数据页面，元数据页面第一次加载到缓冲区中时或`segment_buffers`参数大小不足时会导致执行时间较长。
- 设置`segment_buffers`参数后需要重启集群才生效。

### 7.22.2.2 bucket 分布表的空间预占用

GUC参数`enable_segment_datafile_preallocate`控制是否打开段页式文件扩展立即分配磁盘空间模式。该参数为`off`时，以空洞的方式扩展文件，实际使用的时候才占用磁盘空间；该参数为`on`时，以预占用的方式扩展文件，文件扩展出来就占用磁盘空间。默认值为`on`。

#### 📖 说明

- 磁盘空间较小时建议关闭`enable_segment_datafile_preallocate`。
- 关闭`enable_segment_datafile_preallocate`时，对于全量build或备份恢复场景，数据复制到目标节点后，空洞会被填0，填0后占用磁盘空间，导致目标节点磁盘空间占用大于源节点。

`bucket`分布表的空间占用包括元数据空间和数据空间，由5个文件组成（1-5）。1号文件为`bucket`元数据文件，2-5号文件为数据文件。

元数据页面的计算方式请参见[设置segment\\_buffers](#)，数据页面与数据量有关，扩展方式为：

- 段页式2号文件：  
每次创建CBI索引时，会在段页式 2 号文件为其申请一个 `extent`（8个页面），不论当前表中是否有数据。如果没有空闲的 `extent`，则会先拓展段页式文件（扩展步长128MB）然后再分配。
- `bucket` 2号文件：  
每次创建 `hashbucket` 表的LBI索引或`toast`索引时，会在 2 号文件为其申请一个新的 `extent`（8个页面），不论当前表中是否有数据。如果没有空闲的 `extent`，则会先拓展段页式文件（扩展步长8MB）然后再分配。

`bucket`分布表预占用空间，计算方法如下：

- enable\_segment\_datafile\_preallocate为off时：

| 预占用空间    | hashbucket 表                                                                                                         | hashbucket 分区表                                                                                                                                                             |
|----------|----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| heap页面数  | (hashbucket 表个数 + toast表个数)*1024                                                                                     | (hashbucket 分区表的分区总数 + toast表分区总数)*1024                                                                                                                                    |
| index页面数 | (hashbucket 表的CBI个数)*2+ (toast索引个数 + hashbucket表的LBI个数) * 1024 * 2<br>• 2：表示1号文件1个页面+2号文件1个页面。<br>• 1024：表示bucket总数。 | [(hashbucket 分区表的toast索引个数 + LOCAL的LBI个数) * 1024 + LOCAL的CBI索引个数] * 分区数 * 2 + GLOBAL的LBI个数 * 1024 * 2 + GLOBAL的CBI个数 * 2<br>• 2：表示1号文件1个页面+2号文件1个页面。<br>• 1024：表示bucket总数。 |

- enable\_segment\_datafile\_preallocate为on时：

| 预占用空间    | hashbucket 表                                                                                                           | hashbucket 分区表                                                                                                                                                            |
|----------|------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| heap页面数  | (hashbucket 表个数 + toast表个数)*1024                                                                                       | (hashbucket 分区表的分区总数 + toast表分区总数)*1024                                                                                                                                   |
| index页面数 | (hashbucket 表的CBI个数)*9 + (toast索引个数 + hashbucket 表的LBI个数) * 1024 * 9<br>• 9：表示1号文件1个页面+2号文件8个页面。<br>• 1024：表示bucket总数。 | [(hashbucket 分区表的toast索引个数 + LOCAL的LBI个数) * 1024 + LOCAL的CBI索引个数] * 分区数 * 9 + GLOBAL的LBI个数 * 1024 * 9+ GLOBAL的CBI个数 * 9<br>• 9：表示1号文件1个页面+2号文件8个页面。<br>• 1024：表示bucket总数。 |

## 空间预占用示例

- 普通表  
假设表包含1个toast索引、1个LBI索引、1个CBI索引。
  - 关闭空间预占用：数据占用空间：2 \* 8KB \* 1024 = 16MB  
索引占用空间：2 \* (8KB + 8KB) \* 1024 + (8KB + 8KB) ≈ 32MB  
总共：48MB
  - 打开空间预占用：数据占用空间：2 \* 8KB \* 1024 = 16MB  
索引占用空间：2 \* (8KB + 64KB) \* 1024 + (8KB + 64KB) ≈ 144MB  
总共：232MB
- 分区表（每个分区相当于1张普通表）  
假设表包含730个分区，1个主键LOCAL索引、toast表、toast索引、1个LPI+LBI索引、1个LPI+CBI索引、1个GPI+CBI索引、1个GPI+LBI索引。

- 关闭空间预占用：数据占用空间：  $2 * 730 * 8KB * 1024 = 11GB$   
索引占用空间：  $730 * [ 3 * (8KB + 8KB) * 1024 + (8KB + 8KB) ] + (8KB + 8KB) + (8KB + 8KB) * 1024 \approx 34GB$   
总共：45GB
- 打开空间预占用：数据占用空间：  $2 * 730 * 8KB * 1024 = 11GB$   
索引占用空间：  $730 * [ 3 * (8KB + 64KB) * 1024 + (8KB + 64KB) ] + (8KB + 64KB) + (8KB + 64KB) * 1024 \approx 154GB$   
总共：165GB

## 7.23 失效重编译

### 7.23.1 特性概述

#### 功能定义

失效重编译是指在数据库中，当视图对象失效后，进行的重新编译操作。它的目的是恢复失效对象的功能，确保这些对象能够正确执行。失效可能由多种原因引起，尤其在数据库架构或依赖的对象发生变化后，比如表结构的修改、依赖对象的删除或更新等。

失效重编译主要涉及以下几个功能模块：

- 一次性入库：创建视图，不满足编译检查要求，例如依赖的外部对象尚未创建，也能将视图插入到数据库的系统表中。
- 级联失效：当被依赖的对象变更时，依赖对象会级联失效。例如变更表字段时，依赖当前表的对象obj1会被失效，同时，依赖对象obj1的对象obj2会被级联失效。
- 重编译：包括手动重编译、自动重编译和批量重编译。
  - 手动重编译：

```
ALTER VIEW objname COMPILE;
```
  - 自动重编译：  
GaussDB会自动尝试重编译失效的对象，例如当失效的视图被调用时，数据库会在执行时自动重编译。
  - 批量重编译：  
通过以下语句可查询数据库中的所有失效对象：

```
SELECT * FROM pg_object WHERE valid='f';
```

  
通过以下语句可调用接口对所有失效对象进行批量重编译：

```
CALL PG_CATALOG.GS_COMPILE_SCHEMA(schema_name, false, 10);
```

使用失效重编译功能，需要开启如下GUC参数：

- enable\_force\_create\_obj=on，提供视图一次性入库功能。
- enable\_view\_invalidation=on，开启视图的级联失效功能，参数值设置为off时表示关闭视图的级联失效功能。

#### 用户场景

场景一：用户第一次部署数据库。

场景二：用户业务变更。

## 规格约束

失效重编译存在以下规格约束：

- 兼容性：支持所有兼容模式数据库。
- 功能规格：
  - 创建视图指定分区，新增视图与分区之间依赖关系；创建视图使用自定义类型或CAST类型转换，新增视图与自定义类型或CAST类型转换之间依赖关系。

### 说明

依赖关系类型请参见《参考》中“系统表和系统视图 > 系统表 > 其他系统表 > PG\_DEPEND”章节。

- 在开启事务后恢复视图依赖对象情况下，不支持通过调用视图触发其自动重编译。
- 功能约束：
  - 备机不支持失效重编译。
  - 不支持同义词发生DDL变更触发视图失效。
  - 通过DBLINK访问远端同义词的视图，不受远端同义词变更影响。
  - 临时视图及依赖本地临时表的视图不支持自动重编译，需手动重编译。
  - 回收站中视图对象不支持失效重编译。

## 技术架构

失效重编译涉及模块：系统表、语法。

- 系统表：
  - PG\_OBJECT：记录对象有效性信息。
  - PG\_DEPEND：记录数据库对象间的依赖关系。
  - GS\_DEP\_SOURCE：记录对象原始定义信息和部分编译产物。
- 语法：ALTER VIEW objname COMPILE语法、视图创建语法、视图访问语法。

## 7.23.2 技术原理

### 核心机制

- 依赖关系记录  
记录数据库对象间的依赖关系，用于数据库对象间的级联失效。
- 级联失效  
当依赖对象发生变更时，基于对象的依赖关系，沿依赖链路级联失效。
- 重编译  
通过自动重编译、手动重编译将数据库对象刷为有效。

## 接口介绍

表 7-28 失效重编译接口介绍

| 失效重编译特性支持的接口 |                                                                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| 语法           | ALTER VIEW view_name COMPILE;                                                                                                                 |
| 系统表          | <ul style="list-style-type: none"><li>PG_DEPEND：记录数据库对象间的依赖关系。</li><li>GS_DEP_SOURCE：记录对象原始定义信息和部分编译产物。</li><li>PG_OBJECT：记录对象状态信息。</li></ul> |
| GUC参数        | <ul style="list-style-type: none"><li>enable_force_create_obj=on，提供视图一次性入库功能。</li><li>enable_view_invalidation=on，提供视图的级联失效功能。</li></ul>      |
| 函数           | PG_CATALOG.GS_COMPILE_SCHEMA重编译指定SCHEMA下视图。当编译视图遇到报错时，异常将会被捕获，继续编译其他对象，直到全部对象编译成功或者到达失败尝试次数后停止。                                               |

### 7.23.3 操作指导

#### 前提条件

- 控制视图的级联失效：设置GUC参数enable\_view\_invalidation为on。
- 控制视图一次性入库：设置GUC参数enable\_force\_create\_obj为on。

#### 语法

支持通过如下语法对视图重新编译：

```
ALTER VIEW view_name COMPILE;
```

#### 注意事项

- 功能使用前，请先检查是否存在失效对象，失效对象被调用，可能会出现一些异常情况。
- 建议避免构造对象依赖成环的场景，访问和编译会出现异常情况。
- 执行失效重编译过程中，不建议执行其它DML操作。
- 视图重编译失败时，可通过CREATE OR REPLACE VIEW语法重建视图。

#### 操作步骤

以下步骤为整体流程步骤，具体执行代码请参见[示例](#)。

- 步骤1** 在数据库中手动创建视图对象。
- 步骤2** 对视图依赖对象进行DDL变更。
- 步骤3** 通过PG\_OBJECT系统表检查是否存在失效对象，失效对象的valid字段置为false。
- 步骤4** 执行ALTER COMPILE语法或pg\_catalog.gs\_compile\_schema批量重编译接口进行重编译。

**步骤5** 检查PG\_OBJECT系统表中是否存在valid字段是false的对象，确认全部对象变为有效。

**步骤6** 正常开启业务。

----结束

## 示例

- 示例一：视图手动重编译

```
-- 开启失效重编译参数。
gaussdb=# SET enable_view_invalidation=on;

-- 创建表，视图依赖表。
gaussdb=# CREATE TABLE tetab_v1(a int,b int);
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.
CREATE TABLE
gaussdb=# CREATE VIEW teview1 AS SELECT a, b FROM tetab_v1;
-- 删除表。
gaussdb=# DROP TABLE tetab_v1;
-- 查询视图有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'teview1';
 relname | valid
-----+-----
teview1 | f
(1 row)
-- 重建视图依赖表。
gaussdb=# CREATE TABLE tetab_v1(a int,b int);
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.
CREATE TABLE
-- 手动编译视图，再次查看对象有效性。
gaussdb=# ALTER VIEW teview1 COMPILE;
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'teview1';
 relname | valid
-----+-----
teview1 | t
(1 row)
-- 删除视图。
gaussdb=# DROP VIEW teview1;
-- 删除表。
gaussdb=# DROP TABLE tetab_v1;
```

- 示例二：视图自动重编译

```
-- 开启失效重编译参数。
gaussdb=# SET enable_view_invalidation=on;

-- 创建表，视图依赖表。
gaussdb=# CREATE TABLE tetab_v2(a int,b int);
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.
CREATE TABLE
gaussdb=# CREATE VIEW teview2 AS SELECT a, b FROM tetab_v2;
-- 删除表。
gaussdb=# DROP TABLE tetab_v2;
-- 查询视图有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'teview2';
 relname | valid
-----+-----
teview2 | f
(1 row)
-- 重建视图依赖表。
gaussdb=# CREATE TABLE tetab_v2(a int,b int);
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.
CREATE TABLE
```



```
-- 使用视图自动重编译，再次查看对象有效性。
gaussdb=# SELECT * FROM teview2;
 a | b
----+----
(0 rows)
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'teview2';
 relname | valid
-----+-----
teview2 | t
(1 row)
-- 删除视图。
gaussdb=# DROP VIEW teview2;
-- 删除表。
gaussdb=# DROP TABLE tetab_v2;
```

- 示例三：视图批量重编译

```
-- 开启失效重编译参数。
gaussdb=# SET enable_view_invalidation=on;

-- 创建schema、表，视图依赖表，视图依赖视图
gaussdb=# DROP SCHEMA IF EXISTS tesch2 CASCADE;
gaussdb=# CREATE SCHEMA tesch2;
gaussdb=# SET current_schema = tesch2;
gaussdb=# CREATE TABLE tab1 (c1 INT, c2 INT);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# CREATE VIEW v_tab1 AS SELECT * FROM tab1;
gaussdb=# CREATE VIEW v_tab2 AS SELECT * FROM v_tab1;
gaussdb=# CREATE VIEW v_tab3 AS SELECT * FROM tab1;
-- 删除表。
gaussdb=# DROP TABLE tab1;
-- 查询对象有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'v_tab1' or c.relname = 'v_tab2' or c.relname = 'v_tab3';
 relname | valid
-----+-----
v_tab1 | f
v_tab2 | f
v_tab3 | f
(3 rows)
-- 恢复删除的表。
gaussdb=# CREATE TABLE tab1 (c1 INT, c2 INT);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
-- 批量重编译tesch2命名空间下视图对象。
gaussdb=# CALL pg_catalog.gs_compile_schema('tesch2', true, 1);
INFO: successful
CONTEXT: referenced column: gs_compile_schema
SQL statement "select pkg_util.gs_compile_schema(:schema_name, :compile_all, :retry_times);"
PL/pgSQL function gs_compile_schema(character varying,boolean,integer) line 24 at EXECUTE
statement
INFO: The number of executions is 1
INFO: successful
gs_compile_schema
-----+-----
t
(1 row)
-- 再次查询视图有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'v_tab1' or c.relname = 'v_tab2' or c.relname = 'v_tab3';
 relname | valid
-----+-----
v_tab1 | t
v_tab2 | t
v_tab3 | t
(3 rows)
-- 删除视图。
```

```
gaussdb=# DROP VIEW v_tab3;
gaussdb=# DROP VIEW v_tab2;
gaussdb=# DROP VIEW v_tab1;
-- 删除表。
gaussdb=# DROP TABLE tab1;
gaussdb=# RESET current_schema;
-- 删除schema。
gaussdb=# DROP SCHEMA tesch2 CASCADE;
```

## 7.23.4 常见问题

Q：怎么检查失效重编译特性是否可用？

A：参考执行如下命令，enable\_view\_invalidation值为on表示支持视图的级联失效，enable\_force\_create\_obj值为on表示支持视图一次性入库。

```
gaussdb=# SELECT name,setting FROM pg_settings WHERE name in ('ddl_invalidation',
'enable_view_invalidation','enable_force_create_obj');
 name | setting
-----+-----
enable_force_create_obj | on
enable_view_invalidation | on
(2 rows)
```

Q：怎么查询对象有效性？

A：查看指定对象有效性可以参考如下示例，从相关系统表查找对象oid，根据oid查询pg\_object的valid字段，valid值为true表示有效，值为false表示无效。如下示例为查看视图信息。

```
gaussdb=# SELECT oid FROM pg_class WHERE relname = 'b_tview1';
 oid

221209
(1 row)
-- 视图b_tview1有效。
gaussdb=# SELECT object_type, valid FROM pg_object WHERE object_oid = 221209;
 object_type | valid
-----+-----
v | t
(1 row)
```

Q：怎么进行重编译？

A：可以通过自动重编译、手动重编译、批量重编译的方式进行重编译操作：

- 自动重编译：例如视图view1依赖表tab1，表tab1变更会导致视图view1失效，恢复表tab1变更后，通过调用视图view1，视图会自动编译为有效。
- 手动重编译：例如视图view1依赖表tab1，表tab1变更会导致视图view1失效，恢复表tab1变更后，通过执行ALTER VIEW view1 COMPILE，视图会被编译为有效。
- 批量重编译：适用于对象较多，依赖关系复杂的场景，通过执行以下语句批量编译指定schema下的对象。

```
CALL pg_catalog.gs_compile_schema(schema_name, false, 10);
```

## 常见问题示例

- 问题一：在开启事务后恢复视图依赖对象，不支持通过调用视图触发其自动重编译。  
-- 设置如下参数。  
gaussdb=# SET enable\_view\_invalidation=on;  
  
-- 创建表。  
gaussdb=# CREATE TABLE base\_tab1(a int, b int);

```
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.
CREATE TABLE
-- 创建视图，视图依赖表。
gaussdb=# CREATE VIEW base_view1 AS SELECT a, b FROM base_tab1;
-- 删除表，视图失效。
gaussdb=# DROP TABLE base_tab1;
-- 查询视图有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'base_view1';
 relname | valid
-----+-----
base_view1 | f
(1 row)
-- 开启视图，在事务内恢复视图依赖的表。
gaussdb=# BEGIN;
gaussdb=# CREATE TABLE base_tab1(a int, b int);
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.
CREATE TABLE
-- 使用视图自动重编译，重编译失败。
gaussdb=# SELECT * FROM base_view1;
ERROR: View "base_view1" is invalid and please check view definition.
LINE 1: select * from base_view1;
 ^
-- 回滚事务。
gaussdb=# ROLLBACK;
-- 重新开启视图，在视图内恢复视图依赖的表并手动重编译视图。
gaussdb=# BEGIN;
gaussdb=# CREATE TABLE base_tab1(a int, b int);
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.
CREATE TABLE
gaussdb=# ALTER VIEW base_view1 COMPILE;
-- 提交视图。
gaussdb=# COMMIT;
-- 再次查询视图有效性。
gaussdb=# SELECT c.relname, o.valid FROM pg_catalog.pg_object o JOIN pg_catalog.pg_class c ON
c.oid = o.object_oid WHERE c.relname = 'base_view1';
 relname | valid
-----+-----
base_view1 | t
(1 row)
-- 删除视图。
gaussdb=# DROP VIEW base_view1;
-- 删除表。
gaussdb=# DROP TABLE base_tab1;
```

- 问题二：不支持同义词发生DDL变更触发视图失效。

```
-- 设置如下参数。
gaussdb=# SET enable_view_invalidation=on;

-- 创建表。
gaussdb=# CREATE TABLE base_tab2(a int, b int);
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.
CREATE TABLE
-- 为表创建同义词。
gaussdb=# CREATE SYNONYM syn_tab FOR base_tab2;
-- 创建视图，视图依赖同义词。
gaussdb=# CREATE VIEW base_view2 AS SELECT a, b FROM syn_tab;
-- 删除同义词，与视图存在依赖不允许删除。
gaussdb=# DROP SYNONYM syn_tab;
ERROR: synonym syn_tab cannot be dropped because other objects depend on it.
DETAIL: view base_view2 depends on synonym public.syn_tab
HINT: Use DROP ... CASCADE to drop the dependent objects.
-- 删除视图。
gaussdb=# DROP VIEW base_view2;
-- 删除同义词。
gaussdb=# DROP SYNONYM syn_tab;
```

```
-- 删除表。
gaussdb=# DROP TABLE base_tab2;
```

- 问题三：临时视图不支持自动重编译，需手动重编译。
  - 设置如下参数。  
gaussdb=# SET enable\_view\_invalidation=on;
  - 创建表。  
gaussdb=# CREATE TABLE base\_tab3(a int, b int);  
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.  
CREATE TABLE
  - 创建临时视图。  
gaussdb=# CREATE TEMP VIEW temp\_view AS SELECT a, b FROM base\_tab3;
  - 删除表。  
gaussdb=# DROP TABLE base\_tab3;
  - 查询视图有效性。  
gaussdb=# SELECT c.relname, o.valid FROM pg\_catalog.pg\_object o JOIN pg\_catalog.pg\_class c ON  
c.oid = o.object\_oid WHERE c.relname = 'temp\_view';  
relname | valid  
-----+-----  
temp\_view | f  
(1 row)
  - 重建视图依赖的表。  
gaussdb=# CREATE TABLE base\_tab3(a int, b int);  
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.  
CREATE TABLE
  - 使用视图触发自动重编译，重编译报错。  
gaussdb=# SELECT \* FROM temp\_view;  
ERROR: View "temp\_view" is invalid and please check view definition.  
LINE 1: SELECT \* FROM temp\_view;  
                  ^
  - 手动重编译视图。  
gaussdb=# ALTER VIEW temp\_view COMPILE;
  - 再次查询视图有效性。  
gaussdb=# SELECT c.relname, o.valid FROM pg\_catalog.pg\_object o JOIN pg\_catalog.pg\_class c ON  
c.oid = o.object\_oid WHERE c.relname = 'temp\_view';  
relname | valid  
-----+-----  
temp\_view | t  
(1 row)
  - 删除视图。  
gaussdb=# DROP VIEW temp\_view;
  - 删除表。  
gaussdb=# DROP TABLE base\_tab3;

## 7.24 存储过程性能定位方法

存储过程性能定位方法根据使用场景可以分为两类：一类是存储过程执行期间，观察存储过程正在执行的语句，定位阻塞场景；一类是存储过程执行完成后，查询相关系统表、系统视图等对象，获取执行记录，确认慢场景。

非透明多写特性下，两类场景均支持。

### 7.24.1 存储过程执行期间性能定位方法

当出现存储过程长时间执行阻塞未结束的情况时，可以通过下面的方法分析。

#### 7.24.1.1 pg\_stat\_activity 查询存储过程正在执行的语句

pg\_stat\_activity支持查看会话正在执行的SQL语句，当该会话正在执行存储过程时，可以看到执行到存储过程内的哪条语句。

7.24.1.1.1 特性概述

功能定义

pg\_stat\_activity系统视图的query字段显示该会话当前正在执行的SQL语句，在R2C10版本后，扩展支持查看存储过程内正在执行的SQL语句。

用户场景

会话执行存储过程阻塞时间较长，存储过程仍在执行未结束，需要确认当前会话执行到了存储过程的所处阶段，执行的语句，可以通过pg\_stat\_activity直观地看到相关的信息。

规格约束

- 必须开启GUC参数track\_activities=on、track\_procedure\_sql=on；
- 由于需要考虑数据库的性能，支持查看以下类型的语句：变量赋值语句、IF条件语句、ELSEIF语句、CASE语句、OPEN CURSOR语句、FETCH CURSOR语句，FORCURSOR语句、动态语句、LOOP语句、WHILE语句、FOR循环语句、PERFORM语句、DML语句、嵌套子程序、COMMIT/ROLLBACK语句、SAVE POINT语句；
- 执行到不支持记录的语句时，pg\_stat\_activity系统视图query字段显示的是上一条支持记录的语句。

技术架构

pg\_stat\_activity中的记录以会话为单位，统计每个会话正在执行的语句，对于存储过程场景，默认记录的query值是最顶层的存储过程调用语句。执行到存储过程内语句时，对支持记录显示的场景，在相关语句处理逻辑中，覆盖当前会话正在执行的query信息。

7.24.1.1.2 技术原理

核心机制

pg\_stat\_activity系统视图支持显示当前正在执行的语句，在语句解析期间会调用接口，将语句信息写到query字段。为了记录存储过程内正在执行的语句，当前的实现机制是在存储过程内语句解析期间，按照"query string: 最顶层的执行语句, current level: 当前语句的直接调用语句, lineno: 当前语句在执行的存储过程中的行号, execute stmt: 当前正在执行的语句" 的格式拼接后，覆盖pg\_stat\_activity指定行的query值。

性能指标

开启track\_procedure\_sql，记录存储过程正在执行的SQL语句，性能劣化在5%左右。

接口介绍

| pg_stat_activity记录存储过程内正在执行的语句的相关接口 |                  |
|-------------------------------------|------------------|
| 系统表                                 | pg_stat_activity |

| pg_stat_activity记录存储过程内正在执行的语句的相关接口 |                                                                                                                                                     |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| GUC参数                               | <ul style="list-style-type: none"><li>track_activities控制pg_stat_activity是否记录信息。</li><li>track_procedure_sql控制支持记录存储过程内正在执行的语句功能是否启用，默认开启。</li></ul> |

7.24.1.1.3 操作指导

前提条件

需要开启track\_activities和track\_procedure\_sql参数，参数值都设置为on。

语法

不涉及。

注意事项

pg\_stat\_activity系统视图中query字段显示长度受GUC参数track\_activity\_query\_size控制，默认1024字节，语句过长会被截断显示不完整。

示例

```
-- 创建存储过程，可参考如下示例
gaussdb=# CREATE OR REPLACE PROCEDURE p1()
AS
BEGIN
pg_sleep(2);
END;
/
CREATE PROCEDURE
gaussdb=# CREATE OR REPLACE PROCEDURE p2()
AS
BEGIN
pg_sleep(2);
END;
/
CREATE PROCEDURE
gaussdb=# CREATE OR REPLACE PROCEDURE p3()
AS
BEGIN
p1();
p2();
END;
/
CREATE PROCEDURE
gaussdb=# CREATE OR REPLACE PROCEDURE p4()
AS
BEGIN
p1();
p2();
END;
/
CREATE PROCEDURE
gaussdb=# CREATE OR REPLACE PROCEDURE p5()
AS
BEGIN
```

```
p3();
p4();
END;
/
CREATE PROCEDURE
-- 会话A，调用存储过程
gaussdb=# CALL p5();
-- 会话B，在会话A执行期间查看是会话A正在执行的SQL，等会话A执行完成后查看语句信息，是会话A执行的上一
-- 条会被记录下来的语句。
gaussdb=# SELECT query FROM pg_stat_activity WHERE query LIKE '%call%' AND sessionid !=
pg_current_sessionid();

query

query string: call p5();, current level: p2(), lineno: 3, execute stmt: CALL pg_sleep(2)
(1 row)
```

## 7.24.1.2 trace 存储过程执行的语句

### 7.24.1.2.1 特性概述

#### 功能定义

在505.2以后的版本中支持了trace存储过程正在执行的SQL语句能力，开启功能后，数据库会在存储过程执行期间将执行的每条语句记录下来，方便实时查看存储过程内正在执行的语句。

#### 用户场景

存储过程执行期间，通过pg\_stat\_activity可以查看该存储过程的执行状态，通过开启trace功能，可以对存储过程执行实时监测，结合语句的执行计划，进一步分析执行慢的原因。

#### 规格约束

- pl\_trace\_func\_start参数level范围：session、portal、time、threshold（不区分大小写）。
- pl\_trace\_func\_start参数max\_num范围：100~10000。
- pl\_trace\_func\_start参数threshold仅支持大于等于零的入参，单位为毫秒，threshold参数仅在level为threshold时生效。
- pl\_trace\_func\_start参数session\_id若不存在则会报错。
- pl\_trace\_func\_start参数session\_id的session已被trace并且与调用函数的session相同时则会报warning并将trace参数修改为本次的入参。
- pl\_trace\_func\_start参数session\_id的session已被trace并且与调用函数的session不相同则会报错。
- pl\_trace\_func\_end参数session\_id若不存在则会报错，若该session未被trace则会报错。
- pl\_trace\_func\_clean参数session\_id若为null则清除所有的trace信息，若trace信息中有该session\_id的数据则清除数据，若不存在则报错。
- pl\_trace\_func\_clean不可清理正在执行trace的session上的数据，若该session正在trace会跳过清理并报warning提示。
- trace功能即使执行的函数报错，也会记录已经执行过的语句。
- trace功能支持同时记录最多100个会话的trace信息。

- trace功能仅支持在线程池模式(enable\_thread\_pool = on)下使用。
- trace功能仅在被trace的会话执行过存储过程后才能使用。
- trace功能不支持匿名块语句、package的实例化语句。
- trace功能不支持显示嵌套子程序内部语句，但会显示子程序行号和子程序函数名。
- trace功能不支持dbe\_perf模式下的存储过程。

技术架构

GaussDB执行PL/SQL语句的过程可以简单拆解为以下几个过程：函数执行、语句块执行、语句执行。在具体语句的执行逻辑前后添加回调函数，可实现对正在执行语句的监测，并进行语句记录和统计信息收集。

7.24.1.2.2 技术原理

核心机制

trace存储过程中正在执行的语句的实现依赖于存储过程期间设置的回调函数，完成语句和统计信息的记录与收集。在存储过程执行前，执行dbe\_perf.pl\_trace\_func\_start接口，数据库会设置trace参数以及回调函数，启用trace功能；存储过程执行期间，语句前后的回调函数会把相应语句的记录和信息存放到链表中；存储过程执行完成后，调用dbe\_perf.pl\_trace\_func\_end接口，清理trace参数，关闭trace功能；调用dbe\_perf.pl\_get\_trace\_info接口，从链表中读取数据并展示信息。

性能指标

DBE\_PERF.pl\_trace\_func\_start函数参数max\_num控制本次trace记录的最大信息数量，取值范围100~10000，该参数值为10000情况下，性能劣化在5%左右。

接口介绍

| trace存储过程执行语句支持的接口 |                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数                 | <ul style="list-style-type: none"><li>● DBE_PERF.PL_TRACE_FUNC_START(session_id, max_num, level, threshold)函数 用于开启trace功能。</li><li>● DBE_PERF.PL_TRACE_FUNC_END(session_id) 函数用于关闭trace功能。</li><li>● DBE_PERF.PL_TRACE_FUNC_CLEAN(session_id) 函数用于清理记录的trace信息。</li><li>● DBE_PERF.PL_GET_TRACE_INFO(session_id, level, database, func, source, plan, time, threshold) 函数用于查看记录的trace信息。</li></ul> |



### 7.24.1.2.3 操作指导

#### 前提条件

需要在线程池模式(enable\_thread\_pool = on)下执行。

#### 语法

不涉及。

#### 注意事项

- PL\_TRACE\_FUNC\_CLEAN不可清理正在执行TRACE的session上的数据。
- TRACE功能不支持DBE\_PERF模式下的存储过程。

#### 示例

```
-- 需要设置参数 enable_thread_pool = on开启线程池模式
--正常创建并执行存储过程
gaussdb=# DROP TABLE IF EXISTS t1;
NOTICE: table "t1" does not exist, skipping
DROP TABLE
gaussdb=# CREATE TABLE t1(a INT);
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.
CREATE TABLE
gaussdb=# CREATE OR REPLACE PROCEDURE p1()
AS
BEGIN
FOR i IN 1..10 LOOP
INSERT INTO t1 VALUES (1);
END LOOP;
END;
/
CREATE PROCEDURE
--在当前session开启trace功能
gaussdb=# CALL DBE_PERF.PL_TRACE_FUNC_START(pg_current_sessid());
pl_trace_func_start

(1 row)
--执行存储过程
gaussdb=# CALL p1();
p1

(1 row)
--查看获取的trace信息
gaussdb=# SELECT * FROM dbe_perf.pl_get_trace_info();
 session_id | level | database | func | source | plan | time | threshold
-----+-----+-----+-----+-----+-----+-----+-----
 2 | SESSION | postgres | 17542 | line 3, for i in 1..10 loop | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
 2 | SESSION | postgres | 17542 | line 4, insert into t1 values (1); | | |
(11 rows)
--关闭trace功能
```

```
gaussdb=# CALL dbperf.pl_trace_func_end(pg_current_session());
pl_trace_func_end

(1 row)
--清理记录的数据
gaussdb=# CALL dbperf.pl_trace_info_clean(pg_current_session());
pl_trace_info_clean

(1 row)
```

## 7.24.2 存储过程执行完成后性能定位方法

在非透明多写特性下，当存储过程已经执行完成，可以通过下面的方法定位分析。

### 7.24.2.1 存储过程慢 SQL

存储过程慢SQL特性在DBE\_PERF.STATEMENT\_HISTORY、DBE\_PERF.STATEMENT相关表和视图做扩展，支持查看存储过程中执行慢的语句。

#### 7.24.2.1.1 特性概述

#### 功能定义

503.1.SPC1800版本以后，支持存储过程慢SQL相关特性，可以通过DBE\_PERF.STATEMENT\_HISTORY、DBE\_PERF.STATEMENT分析存储过程内执行慢的语句。

#### 用户场景

当出现执行慢的存储过程语句时，如果不确定该语句是被哪个存储过程调用，可以借助慢SQL相关特性查找最外层调用语句、单次执行产生的指标、单条语句累计执行产生的消耗等信息。

#### 规格约束

- 需要设置参数instr\_unique\_sql\_track\_type = 'all'，DBE\_PERF.STATEMENT\_HISTORY、DBE\_PERF.STATEMENT记录存储过程内语句。
- 查询DBE\_PERF.STATEMENT\_HISTORY必须具有SYSADMIN权限或者MONITOR ADMIN权限。只可在系统库中查询到结果，用户库中无法查询。
- DBE\_PERF.STATEMENT\_HISTORY系统表受track\_stmt\_stat\_level控制，默认为"OFF,L0"，第一部分控制Full SQL，第二部分控制Slow SQL。该表不同列限制不同级别才能记录，部分列不同级别记录信息不同。
- 查询视图DBE\_PERF.STATEMENT必须具有SYSADMIN权限或者MONITOR ADMIN权限。CN上可以看到此CN接收到的归一化SQL的全量统计信息（包含DN相关信息）；DN上仅可看到此节点执行的归一化SQL的统计信息。
- 访问WDR快照数据需要SYSADMIN或MONADMIN权限。GUC参数enable\_wdr\_snapshot值为on，且快照数量大于等于2时才可以生成WDR报告。
- 支持记录的存储过程语句包括：DML语句、动态语句、PERFORM语句、OPEN CURSOR语句和FETCH CURSOR语句，其中OPEN和FETCH语句的QUERY会被转换为游标实际执行的查询语句。

技术架构

GAUSSDB 中慢SQL指的是执行时间超过阈值log\_min\_duration\_statement的语句。

存储过程慢SQL的支持分为DBE\_PERF.STATEMENT\_HISTORY和DBE\_PERF.STATEMENT两部分。

- 对DBE\_PERF.STATEMENT\_HISTORY系统表，语句每次执行会产生记录，根据track\_stmt\_stat\_level参数值以及当前语句是否是慢SQL，决定该记录是否会刷新到DBE\_PERF.STATEMENT\_HISTORY。
- DBE\_PERF.STATEMENT记录归一化信息，底层实现是哈希表，unique\_query\_id、parent\_unique\_sql\_id、userid、dbid相同的语句会归一到同一条记录。

7.24.2.1.2 技术原理

核心机制

当前支持语句执行前生成unique\_query\_id，并在DBE\_PERF.STATEMENT插入一条记录，供后续归一、上报刷新使用；语句执行期间，在DBE\_PERF.STATEMENT中刷新相应语句的产生的指标；语句执行完，执行记录会判断是否是慢SQL，结合track\_stat\_stmt\_level的值，决定是否需要记录到DBE\_PERF.STATEMENT\_HISTORY。

存储过程场景是在存储内部语句执行前，将存储过程调用语句的unique\_sql\_id保存，作为存储过程内语句的parent\_unique\_sql\_id，通过parent\_unique\_sql\_id和unique\_sql\_id的值，维护语句间的调用关系。存储过程内语句执行时，在DBE\_PERF.STATEMENT插入记录，每条语句执行完，结合该语句是否执行慢，以及参数track\_stmt\_stat\_level的值确认是否需要记录到DBE\_PERF.STATEMENT\_HISTORY中。和非存储过程语句的区别在于，DBE\_PERF.STATEMENT表中记录的指标是在整个存储过程执行完成后才会统一刷新。

性能指标

instr\_unique\_sql\_track\_type设置成all，存储过程执行耗时相比于设置成top时，劣化5%左右。

接口介绍

| 存储过程慢SQL支持的接口 |                                                                                                                                |
|---------------|--------------------------------------------------------------------------------------------------------------------------------|
| 函数            | DBE_PERF.GET_FULL_SQL_BY_PARENT_ID_AND_TIMESTAMP 根据指定parent_unique_sql_id递归查询执行时间段内DBE_PERF.STATEMENT_HISTORY中，该语句及所有子语句的执行记录。 |
| 系统表           | DBE_PERF.STATEMENT_HISTORY字段parent_unique_sql_id，表示调用该语句的存储过程的unique_query_id，最顶层语句该值为0。                                       |
| 系统视图          | DBE_PERF.STATEMENT字段parent_unique_sql_id，表示调用该语句的存储过程的unique_query_id，最顶层语句该值为0。                                               |

| 存储过程慢SQL支持的接口 |                                                     |
|---------------|-----------------------------------------------------|
| GUC参数         | instr_unique_sql_track_type参数all选项控制开启存储过程慢SQL特性功能。 |

7.24.2.1.3 操作指导

前提条件

需要设置instr\_unique\_sql\_track\_type值为all。

语法

不涉及。

注意事项

DBE\_PERF.STATEMENT表记录上限受instr\_unique\_sql\_count控制，instr\_unique\_sql\_track\_type默认值为all，默认记录存储过程内语句，会使该表记录的语句条数更容易达到阈值，达到阈值后不会产生新的记录，当数据量达到阈值可以使用reset\_unique\_sql(scope, clean\_type, clean\_value) 接口清理该表数据。

查询DBE\_PERF.STATEMENT\_HISTORY需要在系统库执行，其他库中不保存数据。

示例

```
-- 检查instr_unique_sql_track_type的值，确认该参数值为all
gaussdb=# SHOW instr_unique_sql_track_type;
gaussdb=# SET instr_unique_sql_track_type='all';
-- 为了方便观察，临时设置记录全量SQL 的L0级别指标
gaussdb=# SET track_stmt_stat_level='L0,L0';
-- 创建测试库
gaussdb=# CREATE DATABASE tedb;
-- 创建存储过程并执行
gaussdb=# CREATE TABLE test(a INT,b INT);
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.
gaussdb=# INSERT INTO test VALUES(generate_series(1,30),2);
gaussdb=# CREATE OR REPLACE PROCEDURE test_proc1()
AS
num INT;
BEGIN
SELECT COUNT(*) INTO num FROM test;
END;
/
gaussdb=# CALL test_proc1();
test_proc1

(1 row)
-- 递归查询DBE_PERF.statement中的记录，查询结果时间类指标受实际环境影响，结果不固定。
gaussdb=# WITH p AS (SELECT * from DBE_PERF.STATEMENT WHERE query LIKE 'call test_proc1%' LIMIT 1)
SELECT
query,
n_soft_parse,
n_hard_parse,
n_returned_rows,
db_time,
cpu_time,
```

```
execution_time,
parse_time,
plan_time,
rewrite_time,
pl_execution_time,
pl_compilation_time,
data_io_time
FROM (
SELECT * FROM p UNION ALL SELECT * from DBE_PERF.STATEMENT
WHERE parent_unique_sql_id = (SELECT unique_sql_id FROM p)
);
 query | n_soft_parse | n_hard_parse | n_returned_rows | db_time | cpu_time |
execution_time | parse_time | plan_time | rewrite_time | pl_execution_time |
pl_compilation_time | data_io_time
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
call test_proc1(); | 0 | 2 | 1 | 21590 | 13130 | 13769 | 327
| 728 | 11 | 12539 |
 659 | 2348
SELECT COUNT(*) FROM test | 0 | 1 | 1 | 19194 | 17705 | 289
| 327 | 728 | 11 | 9324 |
 659 | 2348
(2 rows)

-- 同理查询DBE_PERF.statement_history，需要在系统库查询
gaussdb=# WITH p AS (SELECT * FROM DBE_PERF.STATEMENT_HISTORY WHERE query LIKE 'call
test_proc1%' LIMIT 1)
SELECT
parent_unique_sql_id,
unique_query_id,
query,
n_soft_parse,
n_hard_parse,
n_returned_rows,
db_time,
cpu_time,
execution_time,
parse_time,
plan_time,
rewrite_time,
pl_execution_time,
pl_compilation_time,
data_io_time
FROM (
SELECT * FROM DBE_PERF.STATEMENT_HISTORY
WHERE unique_query_id = (SELECT unique_query_id FROM p)
UNION ALL
SELECT * FROM DBE_PERF.STATEMENT_HISTORY
WHERE parent_unique_sql_id = (SELECT unique_query_id FROM p)
);
parent_unique_sql_id | unique_query_id | query | n_soft_parse | n_hard_parse |
n_returned_rows | db_time | cpu_time | execution_time | parse_time | plan_t
ime | rewrite_time | pl_execution_time | pl_compilation_time | data_io_time
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
 0 | 4276558564 | call test_proc1(); | 0 | 2 | 1 | 14842
| 7871 | 13480 | 0 |
 0 | 0 | 12539 | 0 | 0
 4276558564 | 1011671199 | SELECT COUNT(*) FROM test | 0 | 1 |
1 | 12446 | 12446 | 0 | 327 |
728 | 11 | 9324 | 659 | 615
(2 rows)

-- 非系统库查询dbe_perf.statement_history，不存放数据
gaussdb=# \c tedb
tedb=# WITH p AS (SELECT * FROM DBE_PERF.STATEMENT_HISTORY WHERE query LIKE 'call test_proc1%'
LIMIT 1)
```

```
SELECT
parent_unique_sql_id,
unique_query_id,
query,
n_soft_parse,
n_hard_parse,
n_returned_rows,
db_time,
cpu_time,
execution_time,
parse_time,
plan_time,
rewrite_time,
pl_execution_time,
pl_compilation_time,
data_io_time
from (
SELECT * FROM DBE_PERF.STATEMENT_HISTORY
WHERE unique_query_id = (SELECT unique_query_id FROM p)
UNION ALL
SELECT * FROM DBE_PERF.STATEMENT_HISTORY
WHERE parent_unique_sql_id = (SELECT unique_query_id FROM p)
);
parent_unique_sql_id | unique_query_id | query | n_soft_parse | n_hard_parse | n_returned_rows | db_time |
cpu_time | execution_time | parse_time | plan_time | rewrite_time | pl_execu
tion_time | pl_compilation_time | data_io_time
-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
(0 rows)

-- 切换回初始数据库，并删除测试数据库
tedb=# \c postgres
gaussdb=# DROP DATABASE tedb;
```

### 7.24.3 常见问题

Q: 当遇到执行慢的存储过程，需要怎么分析？

A: 遇到慢存储过程，可以参考下面的流程，分析执行慢的原因：

- 如果该存储过程正在执行，可以通过pg\_stat\_activity查询存储过程的执行状态；通过pg\_stat\_activity获取session\_id后，还调用trace接口，实时监测执行信息，语句执行计划，进一步分析执行慢的原因；
- 如果该存储过程已经执行完，在不确定调用路径时，可以使用dbe\_perf.statement\_history，查找执行慢的语句，分析语句执行计划、等待事件，通过debug\_query\_id相同，parent\_unique\_sql\_id为0的条件，可以找到最外层的调用语句。

Q: 存储过程执行期间，如何结合pg\_stat\_activity和trace共同分析存储过程的执行情况？

A: 假设某个会话正在执行存储过程跑批业务，出现了执行时间过长且一直没有执行结束的情况，可以参考如下过程进行分析。

1. 参考如下语句，查看正在执行存储过程的会话ID；

```
select sessionid, query from pg_stat_activity where query like '%call%' and sessionid != pg_current_sessid();
```

2. 调用dbe\_perf.pl\_trace\_func\_start(sessionid)函数对相应会话进行trace；
3. 开始trace就可以通过dbe\_perf.pl\_get\_trace\_info函数进行实时的观测，结合语句执行计划分析执行慢的原因。

## 7.25 数据库连接驱动指南

### 7.25.1 JDBC

#### 7.25.1.1 概述

JDBC（Java Database Connectivity，Java数据库连接）为多种关系型数据库提供统一访问接口，使Java应用程序能够与各种不同的数据库进行交互，从而实现数据的持久化和管理。JDBC允许Java应用程序通过SQL语句来执行数据库操作，包括查询、插入、更新和删除数据等。

GaussDB库提供了对JDBC 4.2特性的支持，可以使用JDK1.8、JDK17、JDK21版本（需要使用对应版本的驱动包）编译程序代码，不支持JDBC桥接ODBC方式。

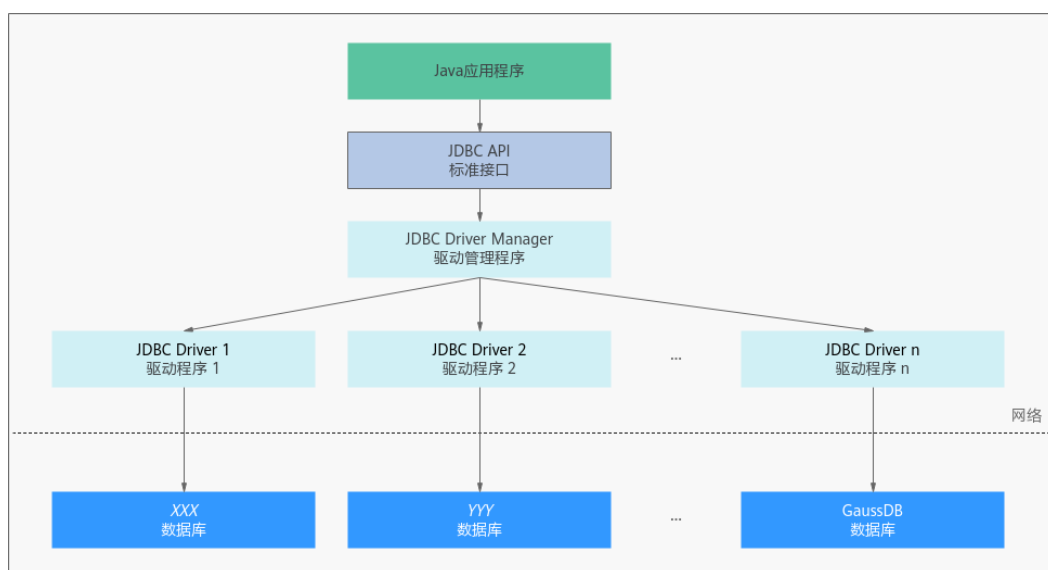
#### 7.25.1.2 技术原理

JDBC架构主要包括三个组件：

- JDBC API（Interface Layer）
- JDBC驱动管理程序（Driver Manager）
- JDBC驱动程序（Driver Layer）

Java应用程序发起数据库操作请求（如查询、更新），通过JDBC API接口与JDBC驱动管理程序通信，而JDBC驱动管理程序负责加载和管理JDBC驱动程序。JDBC驱动程序则负责与特定的数据库通信，发送SQL到数据库。数据库执行SQL操作并返回结果集。JDBC的系统架构请参见图7-40。

图 7-40 JDBC 系统架构



接口介绍

JDBC接口是一套提供给用户的API方法，使Java应用程序能够与各种不同的数据库进行交互。java.sql包是JDBC核心接口。[表7-29](#)介绍了JDBC中常用的一些接口和方法，涵盖了连接数据库、执行SQL语句、获取结果集、关闭连接等常见操作。

说明

JDBC接口和方法的详细介绍，请参见[JDBC接口参考](#)章节。

表 7-29 JDBC 常用接口方法参考

| 接口                         | 接口功能描述                      | 关键方法                                                    | 方法功能描述                          |
|----------------------------|-----------------------------|---------------------------------------------------------|---------------------------------|
| java.sql.DriverManager     | 管理JDBC驱动，提供获取数据库连接的入口。      | getConnection(String url, String user, String password) | 通过URL+用户名+密码建立数据库连接。            |
|                            |                             | getConnection(String url, Properties info)              | 通过URL+Properties（含更多参数）建立数据库连接。 |
| java.sql.Connection        | 代表与数据库的会话，管理事务和创建Statement。 | createStatement()                                       | 创建Statement。                    |
|                            |                             | prepareStatement(String sql)                            | 创建PreparedStatement。            |
|                            |                             | prepareCall(String sql)                                 | 创建CallableStatement。            |
|                            |                             | close()                                                 | 关闭数据库连接。                        |
| java.sql.Statement         | 执行静态SQL语句（无参数）。             | executeQuery(String sql)                                | 执行查询类SQL语句，返回ResultSet。         |
|                            |                             | executeUpdate(String sql)                               | 执行数据修改类SQL语句（如增删改），返回影响行数。      |
|                            |                             | execute(String sql)                                     | 执行任意SQL，返回布尔值表示是否有结果集。          |
|                            |                             | addBatch(String sql)                                    | 添加批处理语句。                        |
|                            |                             | executeBatch()                                          | 执行批处理。                          |
|                            |                             | close()                                                 | 关闭Statement对象。                  |
| java.sql.PreparedStatement | 预编译SQL语句（支持参数化查询，防止SQL注入）。  | setInt(int parameterIndex, int x)                       | 设置参数。                           |
|                            |                             | setString(int parameterIndex, String x)                 |                                 |



| 接口                         | 接口功能描述     | 关键方法                                               | 方法功能描述                       |
|----------------------------|------------|----------------------------------------------------|------------------------------|
|                            |            | setFloat(int parameterIndex, float x)              |                              |
|                            |            | executeQuery()                                     | 执行已编译的查询类SQL语句，返回ResultSet。  |
|                            |            | executeUpdate()                                    | 执行已编译的数据修改类SQL语句，返回影响行数。     |
|                            |            | execute()                                          | 执行已编译的任意SQL语句，返回布尔值表示是否有结果集。 |
|                            |            | addBatch()                                         | 添加预编译语句到批处理。                 |
|                            |            | executeBatch()                                     | 执行批处理。                       |
|                            |            | close()                                            | 关闭PreparedStatement对象。       |
| java.sql.CallableStatement | 调用数据库存储过程。 | registerOutParameter(int parameterIndex, int type) | 注册输出参数。                      |
|                            |            | getInt(int columnIndex)                            | 获取存储过程的输出参数值。                |
|                            |            | getString(int columnIndex)                         |                              |
|                            |            | getFloat(int columnIndex)                          |                              |
| java.sql.ResultSet         | 表示查询结果集。   | next()                                             | 移动游标到下一行。                    |
|                            |            | getInt(int columnIndex)                            | 按列标获取int型数据。                 |
|                            |            | getInt(String columnLabel)                         | 按列名获取int型数据。                 |
|                            |            | getString(int columnIndex)                         | 按列标获取String型数据。              |
|                            |            | getString(String columnLabel)                      | 按列名获取String型数据。              |
|                            |            | getMetaData()                                      | 获取结果集元数据（如列名、类型）。            |
|                            |            | close()                                            | 关闭ResultSet对象。               |

| 接口                             | 接口功能描述              | 关键方法                          | 方法功能描述 |
|--------------------------------|---------------------|-------------------------------|--------|
| java.sql.ResultSet<br>MetaData | 描述结果集的元数据（如列名、类型等）。 | getColumnCount()              | 获取列数。  |
|                                |                     | getColumnName(int column)     | 获取列名。  |
|                                |                     | getColumnTypeName(int column) | 获取列类型。 |

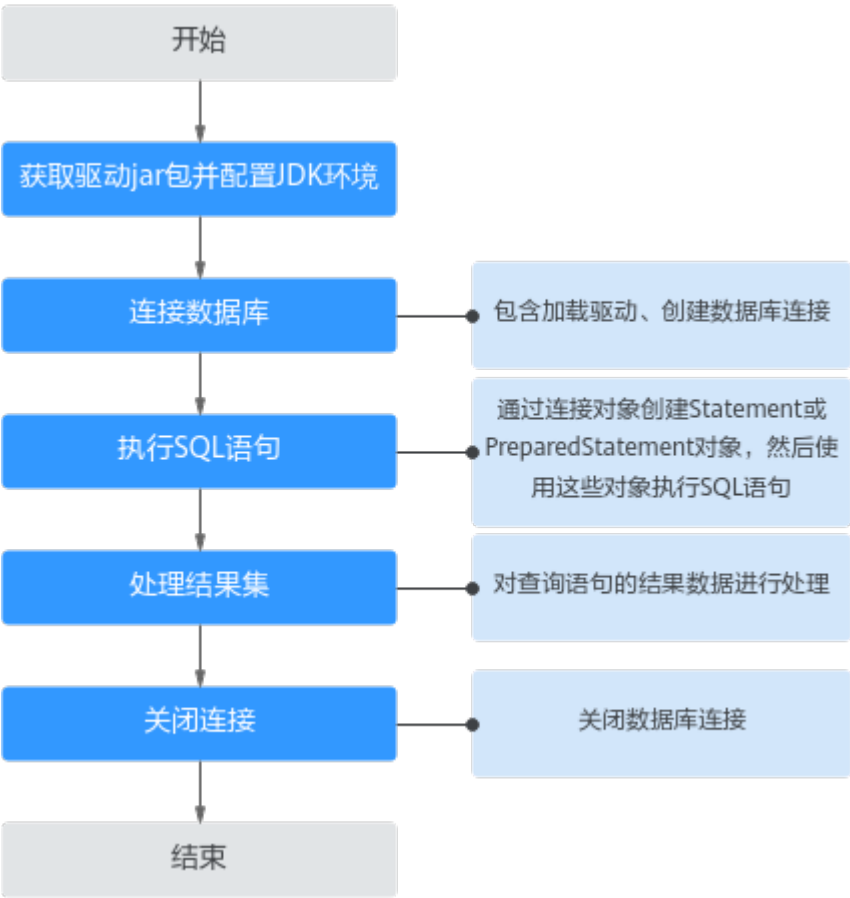
7.25.1.3 操作指导

7.25.1.3.1 开发流程

采用JDBC开发应用程序的流程，请参见图7-41。

具体包含获取驱动jar包并配置JDK环境、连接数据库、执行SQL语句、处理结果集、关闭连接等几个部分。

图 7-41 采用 JDBC 开发应用程序的流程



7.25.1.3.2 获取驱动 jar 包并配置 JDK 环境

环境准备包括获取驱动jar包、配置JDK环境。

## 获取驱动 jar 包

解压发布包，并从中获取对应的驱动jar包：Jdk8发布包提供支持JDK1.8环境的驱动包；Jdk17发布包提供支持JDK17环境的驱动包；Jdk21发布包提供支持JDK21环境的驱动包。

Jdk8发布包目录结构（以X86为例）如下：

- GaussDB-Kernel\_数据库版本号\_Jdbc\_X86\_Jdk8\_Distributed.tar.gz
  - GaussDB-Kernel\_数据库版本号\_操作系统\_64bit\_Jdbc.tar.gz
    - gaussdbjdbc.jar
    - gscejdbc.jar
    - gsjdbc4.jar
    - gsjdbc200.jar
    - opengaussjdbc.jar
    - opengaussjdbc-ce.jar

Jdk17发布包目录结构（以X86为例）如下：

- GaussDB-Kernel\_数据库版本号\_Jdbc\_X86\_Jdk17\_Distributed.tar.gz
  - GaussDB-Kernel\_数据库版本号\_操作系统\_64bit\_Jdbc.tar.gz
    - gaussdbjdbc.jar
    - gscejdbc.jar
    - gsjdbc4.jar
    - gsjdbc200.jar
    - opengaussjdbc.jar
    - opengaussjdbc-ce.jar

Jdk21发布包目录结构（以X86为例）如下：

- GaussDB-Kernel\_数据库版本号\_Jdbc\_X86\_Jdk21\_Distributed.tar.gz
  - GaussDB-Kernel\_数据库版本号\_操作系统\_64bit\_Jdbc.tar.gz
    - gaussdbjdbc.jar
    - gscejdbc.jar
    - gsjdbc4.jar
    - gsjdbc200.jar
    - opengaussjdbc.jar
    - opengaussjdbc-ce.jar

根据需要从发布包中获取对应的jar驱动包：

- 对于JDK1.8环境，使用Jdk8的发布包：
  - gaussdbjdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包适用于JDK1.8环境。
  - gscejdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包打包了密态数据库需要加载的加解密相关的依赖库，密态场景使用此驱动包。目前仅支持EulerOS操作系统。
- 对于JDK17环境，使用Jdk17的发布包：
  - gaussdbjdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包适用于JDK17环境。
  - gscejdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包包含密态数据库所需的加解密依赖库，密态场景使用此驱动包。目前仅支持EulerOS操作系统。
- 对于JDK21环境，使用Jdk21的发布包：
  - gaussdbjdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包适用于JDK21环境。
  - gscejdbc.jar：主类名为“com.huawei.gaussdb.jdbc.Driver”，数据库连接的url前缀为“jdbc:gaussdb”，此驱动包包含密态数据库所需的加解密依赖库，密态场景使用此驱动包。目前仅支持EulerOS操作系统。

#### 须知

- 使用gscejdbc.jar驱动包时，需要先设置环境变量LD\_LIBRARY\_PATH。具体使用方式参见[使用JDBC操作密态数据库](#)。
- jdbc发布件jar包按照架构分类，gscejdbc.jar包必须与对应的部署环境一致才能使用，其他jar包无需与部署环境一致。
- Jdk17和Jdk21发布包的驱动暂不支持JDBC4.3标准新特性。

## 配置 JDK 环境

客户端需配置JDK环境，本章节以配置JDK1.8为例。JDK支持Windows、Linux等多种平台。以Windows为例，配置方法如下。

**步骤1** 在DOS窗口（Windows下的命令提示符）输入以下命令查看JDK版本。

```
java -version
```

确认是否已安装JDK1.8。若未安装，请从[官方网站](#)下载安装包并安装。

**步骤2** 根据如下步骤配置系统环境变量。

1. 右键单击“我的电脑”，选择“属性”。
2. 在“系统”页面左侧导航栏单击“高级系统设置”。

#### 说明

因操作系统版本之间存在差异，操作时请以实际路径为准。

3. 在“系统属性”页面，“高级”页签上单击“环境变量”。
4. 在“环境变量”页面上，“系统变量”区域单击“新建”或“编辑”配置系统变量。变量说明如[表7-30](#)所示。

表 7-30 变量说明

| 变量名       | 操作                                                                                     | 变量值                                                                                                                                                                                          |
|-----------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JAVA_HOME | <ul style="list-style-type: none"><li>– 若存在，则单击“编辑”。</li><li>– 若不存在，则单击“新建”。</li></ul> | JAVA的安装目录。<br>例如：C:\Program Files\Java\jdk1.8.0_131。                                                                                                                                         |
| Path      | 单击“编辑”。                                                                                | <ul style="list-style-type: none"><li>– 若配置了JAVA_HOME，则在变量值的最前面加上：<br/>%JAVA_HOME%\bin。</li><li>– 若未配置JAVA_HOME，则在变量值的最前面加上JAVA安装的全路径：<br/>C:\Program Files\Java\jdk1.8.0_131\bin。</li></ul> |
| CLASSPATH | 单击“新建”。                                                                                | %JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar。                                                                                                                                                   |

----结束

7.25.1.3.3 连接数据库

加载驱动后，才能调用DriverManager.getConnection方法创建数据库连接。本章节介绍了DriverManager.getConnection(String url, String user, String password)方法和DriverManager.getConnection(String url, Properties info)方法创建数据库连接的操作步骤。

getConnection(String url, String user, String password)方法

使用DriverManager.getConnection(String url, String user, String password)方法创建数据库连接，步骤如下：

步骤1 导入java.sql.Connection和java.sql.DriverManager。

```
import java.sql.Connection;
import java.sql.DriverManager;
```

java.sql.Connection是数据库连接接口，通过java.sql.DriverManager的getConnection()方法让应用程序连接到数据库。此外，用户需要根据实际的应用场景，再导入其他的接口和类，具体请参见JDBC接口参考章节。

步骤2 指定数据库sourceURL（\$ip、\$port、database需要用户自行修改）、用户名和密码。

```
将用户名和密码直接写入代码可能存在安全风险，建议在环境变量中存放。
String sourceURL = "jdbc:gaussdb://$ip.$port/database";
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
```

步骤3 加载驱动。

```
1. 在代码运行工具（如IDE）中添加gaussdbjdbc.jar包。
2. 执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。
String driver = "com.huawei.gaussdb.jdbc.Driver";
Class.forName(driver);
```

#### 步骤4 创建数据库连接。

调用DriverManager.getConnection(String url, String user, String password)，进行数据库连接。

```
Connection conn = DriverManager.getConnection(sourceURL, userName, password);
```

----结束

### getConnection(String url, Properties info)方法

使用DriverManager.getConnection(String url, Properties info)接口创建数据库连接，步骤如下：

#### 步骤1 导入java.sql.Connection、java.sql.DriverManager、java.util.Properties。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
```

java.util.Properties的setProperty()方法，用于设置Properties对象的属性值。此外，用户需要根据实际的应用场景，再导入其他的接口和类，具体请参见[JDBC接口参考章节](#)。

#### 步骤2 指定数据库sourceURL（\$ip、\$port、database需要用户自行修改）、用户名和密码。

将用户名和密码直接写入代码可能存在安全风险，建议在环境变量中存放。

```
String sourceURL = "jdbc:gaussdb://$ip.$port/database";
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
```

#### 步骤3 创建Properties对象，并将userName和password设置为该对象的属性值。

```
Properties info = new Properties();
info.setProperty("user", userName);
info.setProperty("password", password);
```

#### 步骤4 加载驱动。

1. 在代码运行工具（如IDE）中添加gaussdbjdbc.jar包。
2. 执行以下命令加载数据库驱动程序“com.huawei.gaussdb.jdbc.Driver”。

```
String driver = "com.huawei.gaussdb.jdbc.Driver";
Class.forName(driver);
```

#### 步骤5 创建数据库连接。

调用DriverManager.getConnection(String url, Properties info)，进行数据库连接。

```
Connection conn = DriverManager.getConnection(sourceURL, info);
```

----结束

### 7.25.1.3.4 执行 SQL 语句

应用程序通过执行SQL语句来操作数据库。本章节通过示例介绍CREATE TABLE、INSERT、UPDATE、SELECT等操作。

- [执行普通SQL语句](#)：创建表。
- [执行批量插入处理语句](#)：批量插入数据。
- [执行预处理更新语句](#)：更新数据。
- [执行查询语句](#)：返回ResultSet。

## 执行普通 SQL 语句

前置条件：已连接数据库，连接对象为conn。

执行普通SQL语句，创建customer\_t1表的步骤如下：

**步骤1** 调用Connection接口的createStatement()方法创建语句对象stmt。

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

**步骤2** 调用Statement接口的executeUpdate()方法执行SQL语句。

```
int rc = stmt.executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name
VARCHAR(32))");
```

**步骤3** 调用Statement接口的close()方法关闭语句对象stmt。

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

----结束

## 执行批量插入处理语句

创建预处理语句后执行批量插入处理，数据库只创建一次执行计划，节省了语句的编译和优化时间。

前置条件：已通过[执行普通SQL语句](#)创建customer\_t1表。

执行预处理语句，批量插入数据的步骤如下：

**步骤1** 调用Connection接口的prepareStatement()方法创建预处理语句对象pst。

```
PreparedStatement pst = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?,?)");
```

**步骤2** 针对每条数据调用对应接口设置参数，以及调用addBatch()将SQL语句添加到批处理中。

```
for (int i = 0; i < 3; i++) {
 pst.setInt(1, i);
 pst.setString(2, "data " + i);
 pst.addBatch();
}
```

**步骤3** 调用PreparedStatement接口的executeBatch()方法执行批量插入处理。

```
pst.executeBatch();
```

**步骤4** 调用PreparedStatement接口的close()方法关闭预处理语句对象pst。

```
pst.close();
```

----结束

### 说明

在数据库批处理操作中，应遵循以下关键原则以确保性能：

#### 1. 持续执行原则

- 批处理程序应当保持持续执行，避免中途终止。
- 异常终止会显著影响数据库整体性能。

#### 2. 事务管理优化

- 必须禁用自动提交功能。  
`conn.setAutoCommit(false);`
- 采用分批提交策略而非单行提交。
- 需要根据数据量和系统资源确定合适的批量大小。

## 执行预处理更新语句

预处理语句是只编译和优化一次，可以通过设置不同的参数值多次使用。由于已经预先编译好，后续使用会减少执行时间。因此，如果多次执行一条语句，请选择使用预处理语句。

前置条件：已通过[执行批量插入处理语句](#)向customer\_t1表中插入数据。

执行预处理SQL语句对数据进行更新的步骤如下：

**步骤1** 调用Connection接口的prepareStatement()方法创建预编译语句对象pstmt。

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE customer_t1 SET c_customer_name = ?
WHERE c_customer_sk = 1");
```

**步骤2** 调用PreparedStatement接口的setString()方法设置参数。

```
pstmt.setString(1, "new Data");
```

**步骤3** 调用PreparedStatement接口的executeUpdate()方法执行预处理SQL语句。

```
int rowcount = pstmt.executeUpdate();
```

**步骤4** 调用PreparedStatement接口的close()方法关闭预处理语句对象pstmt。

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

----结束

### 须知

prepareStatement设置绑定参数后，最终会构建成一个B报文或U报文，在下一步执行SQL语句时发给服务端。B报文或U报文有最大长度限制（不能超过1023MB），如果一次绑定数据过大，可能因报文过长导致异常。因此prepareStatement设置绑定参数时需要注意评估和控制绑定数据的大小，避免出现超出报文上限要求的现象。

## 执行查询语句

前置条件：已[执行预处理更新语句](#)。

执行查询语句的步骤如下：

**步骤1** 调用Connection接口的createStatement()方法创建语句对象stmts。

```
Statement stmts = conn.createStatement();
```

**步骤2** 调用Statement接口的executeQuery()方法执行SQL查询语句，并返回ResultSet。

```
ResultSet rs = stmts.executeQuery("SELECT * FROM customer_t1");
```

----结束

### 7.25.1.3.5 处理结果集

[执行查询语句](#)后，需要对返回的结果集进行处理。

**步骤1** 使用while循环和next()方法逐行遍历结果集。

```
while (rs.next()) {
 // 使用列索引获取值
 int c_customer_sk = rs.getInt(1); // 获取第一列的int型数据
 String c_customer_name = rs.getString(2); // 获取第二列的String型数据
 System.out.println("c_customer_sk: " + c_customer_sk + ", c_customer_name: " + c_customer_name);
}
```



#### 步骤2 调用close()方法关闭资源。

```
rs.close(); // 关闭ResultSet对象
stmts.close(); // 关闭Statement对象
```

----结束

### 7.25.1.3.6 关闭数据库连接

使用数据库连接完成相应的数据操作后，需要调用close()方法关闭数据库连接。

```
conn.close();
```

#### 说明

Connection、Statement和ResultSet接口都有close()方法，在使用完对象后应把它们关闭。

### 7.25.1.3.7 最佳实践

GaussDB为用户提供JDBC的最佳实践。

具体请参见《最佳实践》中“JDBC最佳实践”章节。

### 7.25.1.4 常见问题

#### 7.25.1.4.1 batchMode 设置错误

#### 问题现象

设置url参数batchMode=on且rewriteBatchedInserts=true，使用JDBC批量插入数据后，提示绑定参数数量与语句需要的参数数量不一致：

```
bind message supplies * parameters, but prepared statement "" requires *
```

示例：

```
// conn是已经创建的Connection对象，创建该Connection的url参数包含
batchMode=on&rewriteBatchedInserts=true
```

```
// 批量绑定参数后执行，绑定参数数量会与改写后的INSERT语句的栏位数不匹配，提示异常。
```

```
// java.sql.BatchUpdateException: bind message supplies 3 parameters, but prepared statement "" requires 6
```

```
PreparedStatement stmt = conn.prepareStatement("INSERT INTO test_tbl VALUES (?, ?, ?)");
```

```
stmt.setInt(1, 1);
stmt.setString(2, "aaa");
stmt.setString(3, "bbbb");
stmt.addBatch();
```

```
stmt.setInt(1, 1);
stmt.setString(2, "aaa");
stmt.setString(3, "bbbb");
stmt.addBatch();
```

```
stmt.setInt(1, 1);
stmt.setString(2, "aaa");
stmt.setString(3, "bbbb");
stmt.addBatch();
```

```
stmt.executeBatch();
```

## 原因分析

将参数rewriteBatchedInserts设置为true时，批量语句会将多条SQL语句合并为一条，导致语句中预留参数栏位数发生变化，如果batchMode=on，会按照合并前的SQL绑定参数，导致绑定参数数量与语句需要的参数数量不一致。

## 处理方法

rewriteBatchedInserts设置为true时，将batchMode设置为off。

### 7.25.1.4.2 Hibernate 框架插入数据开启校验时报错

## 问题现象

客户从ORA兼容模式数据库迁移到GaussDB数据库，表结构使用DRS工具进行迁移，迁移后客户原业务代码不可用，Hibernate框架校验表结构报错。

Schema-validation: wrong column type encountered in column [execute\_time] in table [abnormal\_event]; found [int8 (Types#BIGINT)], but expecting [int4 (Types#INTEGER)]

示例：

- 原ORA兼容模式数据库中Student表结构为如下格式：  
id number(15,0)  
sid number(15,0)  
age number(10,0)
- 使用DRS工具迁移到GaussDB数据库后，Student表结构为如下格式：  
id bigint  
sid bigint  
age Integer
- 当实际数据库中Student实体类信息如下时：  
Long id  
Long sid  
Long age --Hibernate框架校验表结构报错。

hibernate.cfg.xml配置文件：

```
<hibernate-configuration>
 <session-factory>
 <!--GaussDB连接信息-->
 <property name="connection.driver_class">com.huawei.gaussdb.jdbc.Driver</property>
 <property name="connection.url">jdbc:gaussdb://x.x.x.x:xx/postgres?currentSchema=public</property>
 <property name="connection.username">xxxxxx</property>
 <property name="connection.password">xxxxxx</property>

 <!--以下为可选配置-->

 <!--是否支持方言-->
 <!--在PG兼容模式下，必须有如下配置-->
 <property name="dialect">org.hibernate.dialect.PostgreSQL92Dialect</property>
 <!--在ORA兼容模式下，推荐如下配置-->
 <!-- <property name="dialect">org.hibernate.dialect.OracleDialect</property>-->

 <property name="met"></property>

 <!--执行CURD时是否打印SQL语句 -->
 <property name="show_sql">true</property>
 <!--自动建表-->
 <!-- <property name="hbm2ddl.auto">create</property>-->
 <!--插入数据时开启校验-->
 <property name="hbm2ddl.auto">validate</property>
 <!--关闭校验-->
 <!-- <property name="hbm2ddl.auto">none</property>-->
 </session-factory>
</hibernate-configuration>
```

```
<!-- 资源注册（实体类映射文件）-->
<mapping resource="student.xml"/>
</session-factory>
</hibernate-configuration>
```

插入数据的主函数：

```
// 创建要测试的对象。
Student student = new Student();
student.setId(20L);
student.setAge(222L);
student.setSid(222L);

// 开启事务，基于session得到对象。
Configuration conf = new Configuration().configure();
SessionFactory sessionFactory = conf.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

// 通过session保存数据。
session.save(student);

// 提交事务。
transaction.commit();

// 操作完毕，关闭session连接对象。
session.close();
```

## 原因分析

1. ORA模式数据库能够插入成功并通过校验：Hibernate框架在校验数据时，通过判断插入数据类型java.lang.Long的TypeCode，与表结构的number类型的TypeCode进行对比，结果不一致，随后会把插入数据的long类型的sqlType与表结构的Type作比较，将long类型转成number(19,0)（对应关系由org.hibernate.dialect.OracleDialect进行维护），因为表结构的type是number类型，number(19,0)以number开头，所以校验可以通过。使用ORA模式数据库时，业务代码涉及整数类型的数据，可以使用java.lang.Long类型插入。
2. GaussDB数据库校验不同，关闭校验后数据插入成功：Hibernate框架在插入数据做校验时，业务代码是java.lang.Long类型，和表数据类型bigint的TypeCode是对应的，因此不报错。但是表数据类型是Integer时无法对应，首先判断业务代码java.lang.Long类型的TypeCode，与表数据Integer的TypeCode比较，结果不一致，再根据org.hibernate.dialect.PostgreSQLDialect对应关系，会将long类型转换为int8，将表数据类型Integer转换为int4，无法对应，所以校验失败。使用GaussDB数据库时，业务代码涉及整数类型的数据，必须使用表类型对应的Java整型。

## 处理方法

可使用以下方式进行问题处理：

- 关闭校验功能。
- 客户修改业务代码，针对不同的表数据类型，采用对应的Java整型。

### 7.25.1.4.3 使用 SSL 方式建连报错或阻塞

## 问题现象

JDBC使用SSL方式建立连接时，会在客户端获取强随机数，建立连接过程中可能出现以下场景的报错。

场景1：出现如下图所示报错。

```
"Thread-0" #18 prio=5 os_prio=0 tid=0x00007f2ad0385000 nid=0x5429 runnable [0x00007f2aa069b000]
 java.lang.Thread.State: RUNNABLE
 at java.io.FileInputStream.readBytes(Native Method)
 at java.io.FileInputStream.read(FileInputStream.java:255)
 at sun.security.provider.NativePRNG$RandomIO.readFully(NativePRNG.java:424)
 at sun.security.provider.NativePRNG$RandomIO.ensureBufferValid(NativePRNG.java:526)
 at sun.security.provider.NativePRNG$RandomIO.implNextBytes(NativePRNG.java:545)
 - locked <0x000000067273a950> (a java.lang.Object)
 at sun.security.provider.NativePRNG$RandomIO.access$400(NativePRNG.java:331)
 at sun.security.provider.NativePRNG$Blocking.engineNextBytes(NativePRNG.java:268)
 at java.security.SecureRandom.nextBytes(SecureRandom.java:468)
 at java.security.SecureRandom.next(SecureRandom.java:491)
 at java.util.Random.nextInt(Random.java:329)
 at sun.security.ssl.SSLContextImpl.engineInit(SSLContextImpl.java:106)
 at javax.net.ssl.SSLContext.init(SSLContext.java:282)
 at org.postgresql.ssl.LibPQFactory.<init>(LibPQFactory.java:175)
 at org.postgresql.core.SocketFactoryFactory.getSslSocketFactory(SocketFactoryFactory.java:62)
 at org.postgresql.ssl.MakeSSL.convert(MakeSSL.java:33)
 at org.postgresql.core.v3.ConnectionFactoryImpl.enableSSL(ConnectionFactoryImpl.java:723)
 at org.postgresql.core.v3.ConnectionFactoryImpl.tryConnect(ConnectionFactoryImpl.java:203)
 at org.postgresql.core.v3.ConnectionFactoryImpl.openConnectionImpl(ConnectionFactoryImpl.java:330)
 at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.java:58)
 at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:357)
```

场景2：建连阻塞。如果url连接串中配置了loginTimeout后，会报Connection attempt timed out，如果不配置该参数，会一直阻塞。

## 原因分析

客户端环境随机数产生的速度太慢，无法满足产品要求，熵源不足，导致服务启动失败。当前已知在一些Linux环境中存在此问题。

## 处理方法

方法1：启动客户端环境中的haveged服务，增加系统熵池熵值以提高读取随机数的速度。启动命令为：

```
systemctl start haveged
```

方法2：调整客户端jdk配置。

打开\$JAVA\_PATH/jre/lib/security/java.security文件，按如下方法修改以下两个配置项：

```
securerandom.source=file:/dev/.urandom
securerandom.strongAlgorithms=NativePRNGNonBlocking:SUN
```

## 7.25.2 ODBC

### 7.25.2.1 概述

ODBC（Open Database Connectivity，开放数据库互连）是由Microsoft公司基于X/OPEN CLI提出的数据库访问应用程序编程接口，采用C/C++语言实现。该技术通过提供标准化的数据访问接口，使应用程序能够以统一的方式访问异构数据库管理系统（DBMS），有效实现了数据库访问与具体DBMS类型及操作系统平台之间的解耦。

ODBC允许应用程序使用SQL来查询、插入、更新和删除数据库中的数据。应用程序通过ODBC提供的API与数据库进行交互，增强了应用程序的可移植性、扩展性和可维护性。

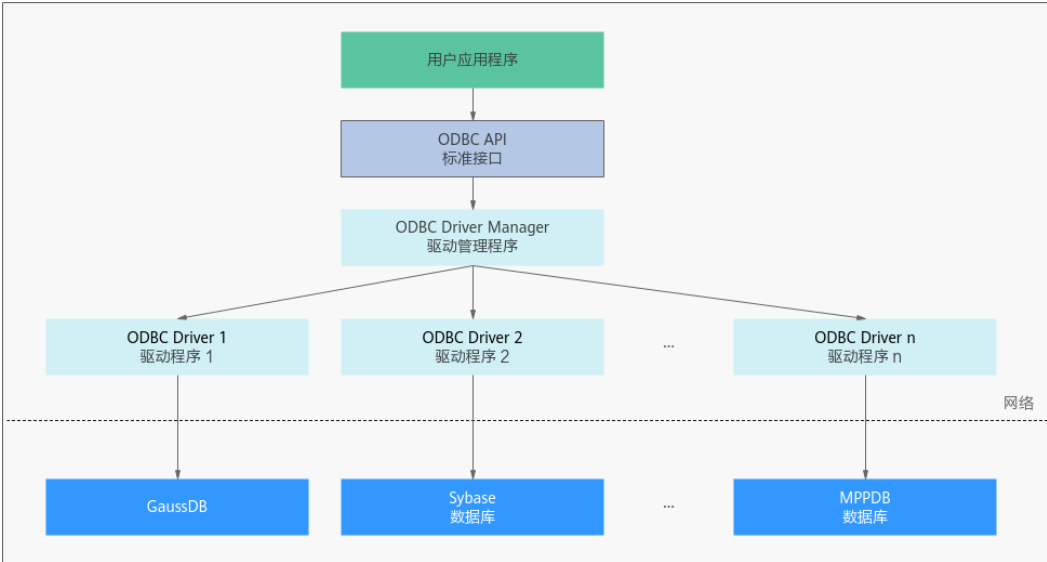
### 7.25.2.2 技术原理

ODBC的架构包括三个主要组件：应用程序、ODBC驱动管理程序和ODBC驱动程序。应用程序使用ODBC API与ODBC驱动管理程序通信，而ODBC驱动管理程序负责加载

和管理ODBC驱动程序。ODBC驱动程序则负责与特定的数据库通信，执行SQL查询并返回结果。操作系统分为UNIX/Linux和Windows两种，所以ODBC的架构也主要分为两种形式。整体的架构二者是一致的，均是应用程序通过驱动程序管理器调用数据库的驱动来连接数据库，区别在于驱动管理器与对应数据库的驱动不同。

ODBC的系统架构请参见图7-42。

图 7-42 ODBC 系统架构



ODBC 支持的平台

GaussDB目前在以下环境中提供对ODBC3.5的支持。

表 7-31 ODBC 支持的平台

| 操作系统                | 平台      |
|---------------------|---------|
| EulerOS V2.0SP5     | x86_64位 |
| EulerOS V2.0SP9     | ARM64位  |
| EulerOS V2.0SP10    | x86_64位 |
| EulerOS V2.0SP10    | ARM64位  |
| Windows 7           | x86_32位 |
| Windows 7           | x86_64位 |
| Windows Server 2008 | x86_32位 |
| Windows Server 2008 | x86_64位 |
| Kylin V10           | x86_64位 |
| Kylin V10           | ARM64位  |
| UnionTech V20       | x86_64位 |

| 操作系统                     | 平台      |
|--------------------------|---------|
| UnionTech V20            | ARM64位  |
| Huawei Cloud EulerOS 2.0 | x86_64位 |
| Huawei Cloud EulerOS 2.0 | ARM64位  |
| Suse 12.5                | x86_64位 |
| openEuler 22.03 LTS SP4  | ARM64位  |
| BCLinux-R8-U2            | x86_64位 |
| BCLINUX21.10             | x86_64位 |
| REDHAT 8.10              | x86_64位 |
| REDHAT 8.10              | ARM64位  |
| NFSChina Server 4.0      | x86_64位 |
| NFSChina Server 4.0      | ARM64位  |
| Ubuntu 22.04 LTS         | x86_64位 |
| Ubuntu 22.04 LTS         | ARM64位  |
| Suse15.7                 | x86_64位 |
| Suse15.7                 | ARM64位  |
| Huawei Cloud EulerOS 3.0 | x86_64位 |
| Huawei Cloud EulerOS 3.0 | ARM64位  |
| Kylin V11                | ARM64位  |

UNIX/Linux系统下的驱动程序管理器主要有unixODBC和iODBC，在这选择驱动管理器 unixODBC-2.3.7作为连接数据库的组件。

Windows系统自带ODBC驱动程序管理器，在控制面板->管理工具中可以找到数据源（ODBC）选项。

 说明

当前数据库ODBC驱动基于开源版本，对于tinyint、smalldatetime、nvarchar2类型，在获取数据类型的时候，可能会出现不兼容的情况。

在Suse12.5上，默认没有libltdl库，需要安装libltdl这个库后，才可以使用ODBC。

接口介绍

ODBC提供了一组标准的C语言API接口，使得应用程序可以与不同类型的数据库进行通信。ODBC的接口被广泛用于各种编程语言和数据库管理系统之间的交互。[表7-32](#)对ODBC中常用的一些接口（函数）进行了介绍，涵盖了连接数据库、执行查询、获取结果等常见操作。

 说明

详细接口请参见[ODBC接口参考](#)章节。

表 7-32 ODBC 常用接口参考

| 接口             | 功能描述                                                                                                                             | 原型                                                                                                                                                                                          |
|----------------|----------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLAllocHandle | 分配环境、连接、语句或描述符的句柄。                                                                                                               | SQLRETURN SQLAllocHandle(SQLSMALLINT HandleType, SQLHANDLE InputHandle, SQLHANDLE *OutputHandlePtr);                                                                                        |
| SQLConnect     | 在驱动程序和数据源之间建立连接。连接上数据源之后，可以通过连接句柄访问到所有有关连接数据源的信息，包括程序运行状态、事务处理状态和错误信息。                                                           | SQLRETURN SQLConnect(SQLHDBC ConnectionHandle, SQLCHAR *ServerName, SQLSMALLINT NameLength1, SQLCHAR *UserName, SQLSMALLINT NameLength2, SQLCHAR *Authentication, SQLSMALLINT NameLength3); |
| SQLPrepare     | 准备一个将要使用的SQL语句。<br>需要注意的是，ODBC发送准备好的语句默认不支持内核复用计划，会导致每次执行都需要重新生成计划，CPU占用率高。如果业务对计划复用有需求，建议开启 <a href="#">MaxCacheQueries</a> 参数。 | SQLRETURN SQLPrepare(SQLHSTMT StatementHandle, SQLCHAR *StatementText, SQLINTEGER TextLength);                                                                                              |
| SQLExecDirect  | 使用当前句柄，执行一条准备好的语句。对于一次只执行一条SQL语句，SQLExecDirect是最快的执行方式。                                                                          | SQLRETURN SQLExecDirect(SQLHSTMT StatementHandle, SQLCHAR *StatementText, SQLINTEGER TextLength);                                                                                           |
| SQLExecute     | SQLExecute 用于执行一个已经通过SQLPrepare 函数预编译的 SQL 语句。                                                                                   | SQLRETURN SQLExecute(SQLHSTMT StatementHandle);                                                                                                                                             |
| SQLGetData     | SQLGetData返回结果集中某一列的数据。可以多次调用它来部分检索不定长度的数据。                                                                                      | SQLRETURN SQLGetData(SQLHSTMT StatementHandle, SQLUSMALLINT Col_or_Param_Num, SQLSMALLINT TargetType, SQLPOINTER TargetValuePtr, SQLLEN BufferLength, SQLLEN *StrLen_or_IndPtr);            |

| 接口            | 功能描述                         | 原型                                                 |
|---------------|------------------------------|----------------------------------------------------|
| SQLFetch      | 从结果集中取下一个行集的数据，并返回所有被绑定列的数据。 | SQLRETURN SQLFetch(SQLHSTMT StatementHandle);      |
| SQLDisconnect | 关闭一个与特定连接句柄相关的连接。            | SQLRETURN SQLDisconnect(SQLHDBC ConnectionHandle); |

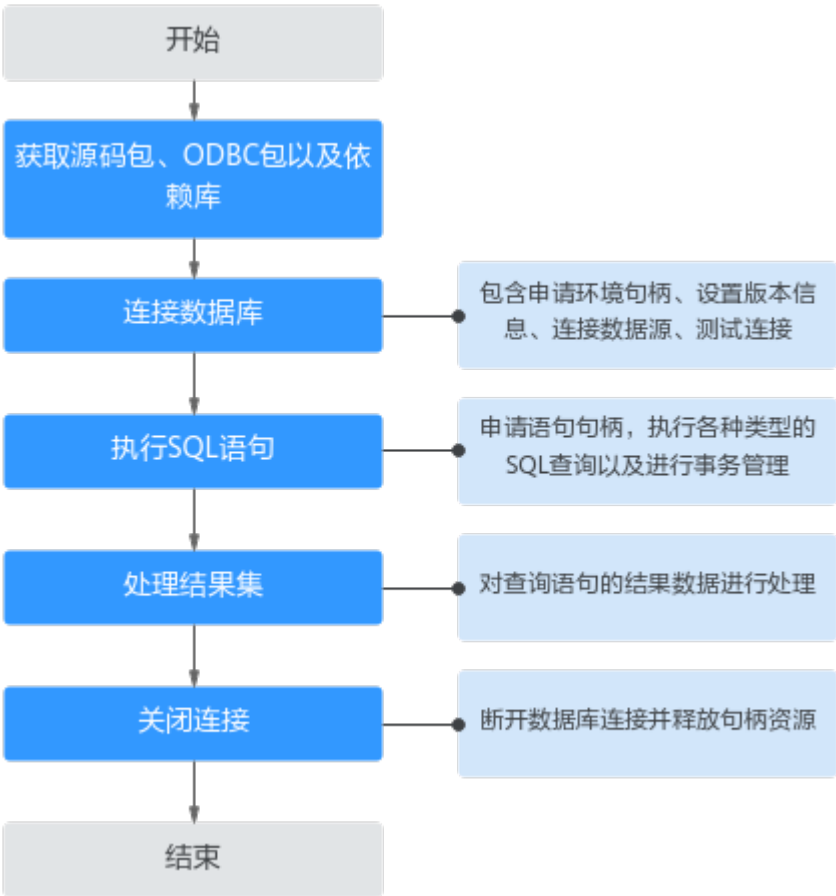
7.25.2.3 操作指导

7.25.2.3.1 开发流程

ODBC开发流程如图7-43所示。

具体包含获取源码包、ODBC包以及依赖库、连接数据库、执行SQL语句、处理结果集、关闭连接五个部分。

图 7-43 ODBC 开发流程





### 7.25.2.3.2 获取源码包、ODBC 包以及依赖库

基于ODBC开发用户需要获取unixODBC源码包、ODBC包以及依赖库。

#### 获取源码包

- unixODBC源码包获取参考地址：<https://www.unixodbc.org/unixODBC-2.3.7.tar.gz>。
- MD5文件下载地址：<https://www.unixodbc.org/unixODBC-2.3.7.tar.gz.md5>。

下载后需使用MD5文件对unixODBC源码包进行完整性校验（查看MD5值，对比MD5值是否与源码包一致）。

#### 获取驱动包

下载表7-33中的驱动包和驱动包校验包。

表 7-33 驱动包下载列表

| 版本        | 下载地址                                          |
|-----------|-----------------------------------------------|
| V2.0-10.x | <a href="#">驱动包</a><br><a href="#">驱动包校验包</a> |

为了防止软件包在传递过程或存储期间被恶意篡改，下载软件包时需下载对应的校验包对软件包进行校验，校验方法如下：

- Linux操作系统软件包完整性校验：
  - a. 上传软件包和软件包校验包到虚拟机的同一目录下。
  - b. 执行如下命令，校验软件包完整性。

```
cat GaussDB_driver.zip.sha256 | sha256sum --check
```

如果回显OK，则校验通过。

```
GaussDB_driver.zip: OK
```
- Windows操作系统软件包完整性校验：
  - a. 使用快捷键“Win+R”打开“运行”窗口。在“打开”栏，输入“cmd”，按“Enter”回车，打开命令行页面。
  - b. 执行以下命令，获取驱动包的Hash值。

```
certutil -hashfile {驱动包本地目录}\{驱动包名} sha256
```

    - {驱动包本地目录}：请根据实际下载目录进行替换。例如：C:\Users
    - {驱动包名}：请根据实际下载的驱动包名进行替换。例如：GaussDB\_driver.zip示例：

```
certutil -hashfile C:\Users\GaussDB_driver.zip sha256
```
  - c. 将b获取到的Hash值和表7-33中获取到的驱动包校验包的Hash值进行比较。
    - 若一致则通过校验。

- 若不一致，请重新下载驱动包，重复a~c进行校验。

## 获取 ODBC 包以及依赖库

ODBC包以及依赖库支持两种操作系统（Linux和Windows），请用户根据实际情况进行选择。

- Linux系统：

从驱动包中获取，包名为GaussDB-Kernel\_数据库版本号\_操作系统版本号\_64bit\_Odbc.tar.gz。Linux环境下，开发应用程序要用到unixODBC提供的头文件（sql.h、sqlext.h等）和库libodbc.so。

这些头文件和库可从unixODBC-2.3.7的源码包中获得。

- Windows系统：

从驱动包中获取，包名为GaussDB-Kernel\_数据库版本号\_Windows\_X86\_Odbc.tar.gz（32位）和GaussDB-Kernel\_数据库版本号\_Windows\_X64\_Odbc.tar.gz（64位）。

Windows环境下，开发应用程序用到的相关头文件和库文件由系统自带。

### 7.25.2.3.3 连接数据库

ODBC连接数据库之前需要准备好所需资源。连接数据库是通过配置ODBC数据源，使用ODBC API或者相应的驱动程序，实现应用程序与数据库之间的通信和交互。本章节介绍如何在Linux/Windows系统下配置数据源并连接到数据库。

## Linux 下配置数据源

**步骤1** 安装unixODBC（默认unixODBC源码包已在环境准备中获取）。如果机器上已经安装了其他版本的unixODBC，可以直接覆盖安装。

以unixODBC-2.3.7版本为例，在客户端执行如下命令安装unixODBC。

```
tar zxvf unixODBC-2.3.7.tar.gz
cd unixODBC-2.3.7
```

```
./configure --enable-gui=no #如果要在ARM服务器上编译，请追加一个configure参数：--build=aarch64-unknown-linux-gnu
make
#安装可能需要root权限
make install
```

### 说明

- 目前不支持unixODBC-2.2.1版本。
- unixODBC默认安装到“/usr/local”目录下，生成数据源文件到“/usr/local/etc”目录下，库文件生成在“/usr/local/lib”目录。
- 通过编译带有--enable-fastvalidate=yes选项的unixODBC来获得更高性能。但此选项可能会导致向ODBC API传递无效句柄的应用程序发生故障，而不是返回SQL\_INVALID\_HANDLE错误。

**步骤2** 替换客户端GaussDB驱动程序。

将GaussDB-Kernel\_数据库版本号\_操作系统版本号\_64bit\_Odbc.tar.gz解压。解压后会得到两个文件夹：lib与odbc，在odbc文件夹中还会有一个lib文件夹。将解压后得到的/lib文件夹与/odbc/lib文件夹中的所有动态库都复制到“/usr/local/lib”目录下。

**步骤3** 配置数据源。

1. 配置ODBC驱动文件。

在“/usr/local/etc/odbcinst.ini”文件中追加以下内容。

```
[GaussMPP]
Driver64=/usr/local/lib/gsqlodbcw.so
setup=/usr/local/lib/gsqlodbcw.so
```

odbcinst.ini文件中的配置参数说明如表7-34所示：

表 7-34 odbcinst.ini 文件配置参数

| 参数           | 描述                         | 示例                                   |
|--------------|----------------------------|--------------------------------------|
| [DriverName] | 驱动器名称，对应数据源DSN中的驱动名。       | [GaussMPP]                           |
| Driver64     | 驱动动态库的路径。                  | Driver64=/usr/local/lib/gsqlodbcw.so |
| setup        | 驱动安装路径，与Driver64中动态库的路径一致。 | setup=/usr/local/lib/gsqlodbcw.so    |

2. 配置数据源文件。

在“/usr/local/etc/odbc.ini”文件中追加以下内容。

```
[gaussdb]
Driver=GaussMPP
#数据库Server IP
Servername=127.0.0.1
#数据库名
Database=db1
#数据库用户名
Username=omm
#数据库用户密码
Password=
#数据库侦听端口
Port=8000
Sslmode=allow
```

odbc.ini文件配置参数说明如表7-35所示，更加详细的参数说明请参见Linux系统下配置数据源章节。

表 7-35 odbc.ini 文件配置参数

| 参数         | 描述                              | 示例                   |
|------------|---------------------------------|----------------------|
| [DSN]      | 数据源的名称。                         | [gaussdb]            |
| Driver     | 驱动名，对应odbcinst.ini中的DriverName。 | Driver=GaussMPP      |
| Servername | 服务器的IP地址。可配置多个IP地址。支持IPv4和IPv6。 | Servername=127.0.0.1 |
| Database   | 要连接的数据库的名称。                     | Database=db1         |
| Username   | 数据库用户名称。                        | Username=omm         |

| 参数       | 描述                                                                                                                                                                                                                                                                                                                                                                                                                         | 示例                               |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| Password | <p>数据库用户密码。</p> <p><b>说明</b></p> <ul style="list-style-type: none"> <li>ODBC驱动本身已经对内存密码进行过清理，以保证用户密码在连接后不会再在内存中保留。</li> <li>但是如果配置了此参数，由于UnixODBC对数据源文件等进行缓存，可能导致密码长期保留在内存中。</li> <li>推荐在应用程序连接时，将密码传递给相应API，而非写在数据源配置文件中。同时连接成功后，应当及时清理保存密码的内存段。</li> <li>配置文件中填写密码时，需要遵循http规则： <ul style="list-style-type: none"> <li>字符应当采用URL编码规范，如"!"应写作"%21"，"%"应写作"%25"，因此应当特别注意字符。</li> <li>"+"会被替换为空格" "。</li> </ul> </li> </ul> | <p>Password=*****</p> <p>***</p> |
| Port     | <p>服务器的端口号。当开启负载均衡时，可配置多个端口号，且需与配置的多IP一一对应。如果开启负载均衡配置多个IP时，仍只配置一个端口号，则默认所有IP共用同一个端口号，即为配置的端口号。</p>                                                                                                                                                                                                                                                                                                                         | <p>Port=8000</p>                 |
| Sslmode  | <p>是否启用 SSL 连接。</p> <p><b>说明</b></p> <p>关于Sslmode的选项的允许值，具体信息如<a href="#">表3 Sslmode的可选项及其描述</a>所示。</p>                                                                                                                                                                                                                                                                                                                    | <p>Sslmode=allow</p>             |

| 参数      | 描述                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 示例        |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| Debug   | <p>控制调试模式的开启及日志输出级别。</p> <p>取值范围：0 ~ INT_MAX</p> <ul style="list-style-type: none"> <li>– 设置为0时表示不启用调试模式。</li> <li>– 设置为大于0的值时将启用调试日志，会打印gsqLODBC驱动的mylog。日志生成目录为/tmp/，文件名为mylog_xxx.log。</li> <li>– 设置为1时表示仅输出FATAL级别日志。</li> <li>– 设置为2时表示输出FATAL和ERROR级别日志。</li> <li>– 设置为3时表示输出FATAL、ERROR和WARN级别日志。</li> <li>– 设置为4时表示输出FATAL、ERROR、WARN和INFO级别日志。</li> <li>– 设置为≥5时表示输出所有级别日志，增加输出TRACE级别日志。</li> </ul> <p>默认值为0。</p> <p><b>说明</b></p> <ol style="list-style-type: none"> <li>1. FATAL：报告致命错误。</li> <li>2. ERROR：报告异常错误。</li> <li>3. WARN：报告警告信息。</li> <li>4. INFO：记录程序运行中的正常关键信息。</li> <li>5. TRACE：提供开发人员使用的调试信息。</li> </ol> | Debug=5   |
| CommLog | <p>控制后端通信日志的开启及日志输出级别。</p> <p>取值范围：0 ~ INT_MAX</p> <ul style="list-style-type: none"> <li>– 设置为0时表示不启用。</li> <li>– 设置为大于0的值时将启用后端日志，会打印gsqLODBC驱动的qlog。日志生成目录为/tmp/，文件名为gsqldb_xxx.log。</li> <li>– 设置为1时表示仅输出ERROR级别日志。</li> <li>– 设置为2时表示输出ERROR和INFO级别日志。</li> <li>– 设置为≥3时表示输出ERROR、INFO和TRACE级别日志。</li> </ul> <p>默认值为0。</p> <p><b>说明</b></p> <ol style="list-style-type: none"> <li>1. ERROR：报告异常错误。</li> <li>2. INFO：记录程序运行中的正常关键信息。</li> <li>3. TRACE：提供开发人员使用的调试信息。</li> </ol>                                                                                                                                                | CommLog=1 |

| 参数                    | 描述                                                                                                                                                            | 示例                      |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| UseServerSidePrepare  | 是否开启数据库端扩展查询协议。<br>取值范围：0，1<br>- 取值为0表示不开启。<br>- 取值为1表示开启。<br>默认值为1。                                                                                          | UseServerSidePrepare=1  |
| UseBatchProtocol      | 是否开启批量查询协议（打开可提高DML性能）。<br>取值范围：0，1<br>- 取值为0时，不使用批量查询协议（主要用于与早期数据库版本通信兼容）。<br>- 取值为1，并且数据库support_batch_bind参数存在且为on时，将打开批量查询协议。<br>默认值为1。                   | UseBatchProtocol=1      |
| ForExtensionConnector | 此开关控制着savepoint是否发送，savepoint相关问题可以注意此开关。<br>取值范围：0，1<br>- 取值为0时表示发送savepoint。<br>- 取值为1时表示不发送savepoint。<br>默认值为1。                                            | ForExtensionConnector=1 |
| ConnectionExtraInfo   | GUC参数connection_info中显示驱动部署路径和进程属主用户的开关。<br>取值范围：0，1<br>- 取值为0时表示不打开此开关。<br>- 取值为1时表示打开此开关。<br>默认值为0。<br><b>说明</b><br>当设置为1时，ODBC驱动会将当前驱动的部署路径、进程属主用户上报到数据库中。 | ConnectionExtraInfo=1   |
| BoolsAsChar           | 是否将布尔值作为字符处理。<br>取值范围：0，1<br>- 取值为0时表示Bools值将会映射为SQL_BIT。<br>- 取值为1时表示Bools值将会映射为SQL_CHAR。<br>默认值为1。                                                          | BoolsAsChar = 1         |

| 参数                | 描述                                                                                                                                                                                                                                  | 示例                  |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| RowVersioning     | <p>是否在更新一行数据时，允许应用检测数据有没有被其他用户进行修改。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时表示不允许应用检测。</li> <li>取值为1时表示允许应用检测。</li> </ul> <p>默认值为0。</p>                                                                     | RowVersioning=1     |
| ShowSystemTables  | <p>是否将默认系统表视为普通SQL表。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时驱动不会将默认系统表视为普通SQL表。</li> <li>取值为1时驱动将默认系统表视为普通SQL表。</li> </ul> <p>默认值为0。</p>                                                                   | ShowSystemTables=1  |
| AutoBalance       | <p>ODBC控制负载均衡的开关。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时表示不开启负载均衡。</li> <li>取值为1时开启负载均衡。</li> </ul> <p>默认值为0。</p> <p><b>说明</b><br/>连接的数据库版本在505.2.0.SPC0200以下时，不支持容灾集群负载均衡。</p>                           | AutoBalance=1       |
| RefreshCNListTime | <p>开启负载均衡时可配置该参数，该参数为刷新CN列表的时间。整数型，单位为秒。</p> <p>取值范围：0 ~ INT_MAX</p> <ul style="list-style-type: none"> <li>取值为0时表示不开启该参数。</li> <li>取值大于0时表示该设定值为刷新CN列表的时间。</li> </ul> <p>默认值为10</p>                                                | RefreshCNListTime=5 |
| Priority          | <p>开启负载均衡时可配置该参数。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时表示不开启该参数。</li> <li>取值为1时开启该参数。</li> </ul> <p>默认值为0。</p> <p><b>说明</b><br/>当Priority开启时，应用程序发起的所有连接优先发送到配置文件中配置的CN上，当配置的CN全部不可用时，连接才会发送到剩余的CN上。</p> | Priority=1          |

| 参数              | 描述                                                                                                                                                                                                                                                                                                     | 示例                  |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| UsingEip        | <p>开启负载均衡时可配置该参数。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时表示不开启该参数。</li> <li>取值为1时开启该参数。</li> </ul> <p>默认值为0。</p> <p><b>说明</b><br/>此值用于控制是否使用业务IP（对应pgxc_node表中node_host1和node_port1列）做负载均衡。当UsingEip开启时，表示使用业务IP做负载均衡；关闭表示使用数据IP（对应pgxc_node表中node_host和node_port列）做负载均衡。</p> | UsingEip=1          |
| MaxCacheQueries | <p>控制每个连接缓存的预编译语句个数。</p> <p>取值范围：0 ~ 4096</p> <p>默认值为0。</p> <p><b>说明</b><br/>如果设置为0，则不开启客户端预编译语句缓存池。设置为大于4096的值会限制为4096。如果执行过的语句个数超过MaxCacheQueries设置的上限，则淘汰最近最少使用的语句。</p>                                                                                                                             | MaxCacheQueries=128 |
| MaxCacheSizeMiB | <p>控制每个连接缓存的预编译语句总大小，在MaxCacheQueries大于0时生效。</p> <p>取值范围：0 ~ 4096</p> <p>默认值为1。</p> <p><b>说明</b><br/>如果缓存的语句总长度大于MaxCacheSizeMiB则淘汰最近最少使用的语句。单位为MB，设置为大于4096的值会限制为4096。</p>                                                                                                                            | MaxCacheSizeMiB=10  |
| TcpUserTimeout  | <p>在支持TCP_USER_TIMEOUT套接字选项的操作系统上，指定传输的数据在TCP连接被强制关闭之前可以保持未确认状态的最大时长。</p> <p>取值范围：0 ~ INT_MAX</p> <p>默认值为0。</p> <p><b>说明</b><br/>0表示使用系统缺省。通过Unix域套接字做的连接忽略这个参数。单位为毫秒。</p>                                                                                                                             | TcpUserTimeout=5000 |
| StandbyRead     | <p>用于指定分布式是否开启备机读模式。</p> <p>取值范围：0，1</p> <ul style="list-style-type: none"> <li>取值为0时表示禁用备机读模式。</li> <li>取值为1时表示开启备机读模式。</li> </ul> <p>默认值为0。</p>                                                                                                                                                      | StandbyRead=1       |
| Pqopt           | <p>用于设置libpq连接参数，参数之间用空格间隔。libpq参数请参见<a href="#">连接参数说明</a>章节。</p>                                                                                                                                                                                                                                     | Pqopt=keepalives=0  |



| 参数                     | 描述                                                                                                                                                                                                | 示例                       |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| KeepaliveTime          | <p>在TCP应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。</p> <p>取值范围：0 ~ INT_MAX</p> <p>设置为0值或者不配置表示使用系统缺省。</p> <p><b>说明</b><br/>禁用“保持激活”功能需要通过Pqopt参数设置 keepalives=0。</p> | KeepaliveTime=2          |
| KeepaliveInterval      | <p>在TCP保持激活信息没有被应该传播的服务器承认之后，控制探活报文发送的间隔秒数。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。</p> <p>取值范围：0 ~ INT_MAX</p> <p>设置为0值或者不配置表示使用系统缺省。</p>                                                     | KeepaliveInterval=2      |
| KeepaliveCount         | <p>控制TCP发送保持激活信息的次数。如果是通过 Unix 域套接字进行连接，或者禁用了“保持激活”功能，则该参数将被忽略。</p> <p>取值范围：0 ~ INT_MAX</p> <p>设置为0值或者不配置表示使用系统缺省。</p>                                                                            | KeepaliveCount=2         |
| SocketTimeout          | <p>用于控制客户端与服务端建立连接完全成功后的socket读写超时时间。单位为秒，默认为0。</p> <p><b>说明</b><br/>该参数包括了数据库语句执行的超时时间，设置的过小可能导致语句执行时间本身发生超时，因此默认为0，需根据场景配置，建议设置非0值。</p>                                                         | SocketTimeout=5          |
| SocketTimeoutInConnect | <p>用于控制TCP三次握手成功后，客户端与服务端建立连接阶段的socket读写超时时间。单位为秒，默认为5。</p> <p><b>说明</b><br/>该参数区别于SocketTimeout：由于三次握手成功后的客户端与服务端建立连接阶段可能存在阻塞，因此需要设置不影响语句执行超时的socket读写超时时间。</p>                                  | SocketTimeoutInConnect=5 |
| CancelTimeout          | <p>用于控制应用端发送cancel消息的超时时间。单位为秒，默认为0。</p>                                                                                                                                                          | CancelTimeout=5          |
| TcpSYNRetries          | <p>用于控制TCP三次握手阶段时SYN最多重传的次數，超过该次数仍未建连成功即报错。默认为0次。</p> <p><b>说明</b><br/>该参数在支持TCP_SYNCNT套接字选项的操作系统上，指定客户端建立连接三次握手阶段SYN包发送失败而重传的次數。0值表示使用系统缺省。通过 Unix域套接字的连接忽略这个参数。</p>                             | TcpSYNRetries=3          |

| 参数                 | 描述                                                                                                                                                                                                                                                                 | 示例                               |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| TextAsLongVarchar  | 将内核侧text类型映射为驱动侧的SQL_LONGVARCHAR类型或者SQL_VARCHAR类型。<br>取值范围：0，1。<br>取值为0时表示将内核侧text类型映射为驱动侧的SQL_VARCHAR类型。<br>取值为1时表示将内核侧text类型映射为驱动侧的SQL_LONGVARCHAR类型。<br>默认值为1。                                                                                                  | TextAsLongVarchar=1              |
| MaxLongVarcharSize | 驱动侧的SQL_LONGVARCHAR类型的最大长度。<br>取值范围：0 ~ INT_MAX。<br>默认值为8190。                                                                                                                                                                                                      | MaxLongVarcharSize=8190          |
| MaxVarcharSize     | 驱动侧的SQL_VARCHAR类型的最大长度。<br>取值范围：0 ~ INT_MAX。<br>默认值为255。                                                                                                                                                                                                           | MaxVarcharSize=255               |
| TargetServerType   | 设定连接的主机的类型。主机的类型和设定的值一致时才能连接成功。设置规则如下： <ul style="list-style-type: none"> <li>primary（默认值）：仅对主备系统中的主节点进行连接。</li> <li>cluster-primary：仅对主集群CN节点能进行连接。</li> <li>cluster-mainnode：仅对容灾集群CN节点能进行连接。</li> </ul> <b>说明</b><br>cluster-primary和cluster-mainnode选项只支持流式容灾。 | TargetServerType=cluster-primary |
| EnableHostCache    | 是否开启节点缓存功能，0表示不开启，1表示开启。默认开启。                                                                                                                                                                                                                                      | EnableHostCache=1                |

表 7-36 Sslmode 的可选项及其描述

| Sslmode | 是否会启用SSL加密 | 描述                                          |
|---------|------------|---------------------------------------------|
| disable | 否          | 不使用SSL安全连接，为默认值。                            |
| allow   | 可能         | 如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。 |
| prefer  | 可能         | 如果数据库支持，那么首选使用SSL安全加密连接，但不验证数据库服务器的真实性。     |

| Sslmode     | 是否会启用SSL加密 | 描述                                                                    |
|-------------|------------|-----------------------------------------------------------------------|
| require     | 是          | 必须使用SSL安全连接，但是只做了数据加密，并不验证数据库服务器的真实性。                                 |
| verify-ca   | 是          | 必须使用SSL安全连接，并且验证数据库是否具有可信证书机构签发的证书。                                   |
| verify-full | 是          | 必须使用SSL安全连接，在verify-ca的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。GaussDB不支持此模式。 |

### 📖 说明

用户通过ODBC连接GaussDB服务器时，可以通过开启SSL加密客户端和服务端之间的通讯。在使用SSL时，默认用户已经获取了服务端和客户端所需要的证书和私钥文件，关于证书等文件的获取请参考Openssl相关文档和命令。

#### 步骤4 在客户端配置环境变量。

```
vim ~/.bashrc
```

在配置文件中追加以下内容。

```
export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH
export ODBCYSINI=/usr/local/etc
export ODBCINI=/usr/local/etc/odbc.ini
```

#### 步骤5 执行如下命令使设置生效。

```
source ~/.bashrc
```

#### 步骤6 测试连接。

安装后/usr/bin下面会存放生成的二进制，可执行isql -v gaussdb（数据源名称）命令。

- 如果显示如下信息，表明配置正确，连接成功。

```
+-----+
| Connected! |
| |
| sql-statement |
| help [tablename] |
| quit |
| |
+-----+
```

- 若显示ERROR信息，则表明配置错误。请检查上述配置步骤是否正确执行。

### 📖 说明

目前通过ODBC连接数据库时，会如下设置内核参数：

```
SET extra_float_digits = 2;
SET DateStyle = 'ISO';
```

这些参数可能会导致ODBC客户端的行为与gsqsl客户端的行为不一致，例如，Date数据显示方式、浮点数精度表示等。如果实际期望和这些配置不符，建议在ODBC应用代码中显式设定这些参数。

M-Compatibility模式数据库下，extra\_float\_digits的默认值为0。

----结束

## Windows 下配置数据源

Windows操作系统自带ODBC数据源管理器，无需用户手动安装管理器便可直接进行配置。

### 步骤1 替换客户端GaussDB驱动程序。

根据需要，将包名为GaussDB-Kernel\_数据库版本号\_Windows\_X64\_Odbc.tar.gz的64位驱动或包名为GaussDB-Kernel\_数据库版本号\_Windows\_X86\_Odbc.tar.gz的32位驱动解压后，单击gsqlodbc.exe进行驱动安装。

### 步骤2 打开驱动管理器。

请使用ODBC版本对应的ODBC驱动管理器（如果使用64位ODBC驱动，必须要使用64位的ODBC驱动管理器，假设操作系统安装盘符为C盘，如果是其他盘符，请对路径做相应修改）。

- 如果需要在64位操作系统使用32位ODBC驱动请使用：C:\Windows\SysWOW64\odbcad32.exe，请勿直接使用“控制面板 > 管理工具 > 数据源(ODBC)”。

#### 说明

WOW64的全称是"Windows 32-bit on Windows 64-bit"，C:\Windows\SysWOW64\存放的是64位系统上的32位运行环境。而C:\Windows\System32\存放的是与操作系统一致的运行环境，具体的技术信息请查阅Windows的相关技术文档。

- 32位操作系统请使用：C:\Windows\System32\odbcad32.exe，或者单击“计算机 > 控制面板 > 管理工具 > 数据源(ODBC)”打开驱动管理器。
- 64位操作系统请使用：控制面板 > 管理工具 > 数据源(ODBC) 打开驱动管理。

### 步骤3 配置数据源。

在打开的驱动管理器上，选择“用户DSN > 添加 > GaussDB Unicode”，然后进行配置。如下图7-44所示：

图 7-44 ODBC Driver Setup

GaussDB ANSI ODBC Driver (gsqloDBC) Setup

|                        |                          |                 |         |
|------------------------|--------------------------|-----------------|---------|
| Data Source            | gaussdb                  | Description     |         |
| Database               |                          | SSL Mode        | disable |
| Server                 |                          | Port            |         |
| User Name              |                          | Password        |         |
| AutoBalance            | <input type="checkbox"/> | Refresh Time    |         |
| UsingEip               | <input type="checkbox"/> | MaxCacheQueries |         |
| TargetServerType       | primary                  | MaxCacheSizeMiB | 1       |
| StandbyRead            | <input type="checkbox"/> | SocketTimeout   |         |
| SocketTimeoutInConnect | 5                        | CancelTimeout   | 10      |
| TcpuserTimeout         |                          | TcpSYNRetries   |         |

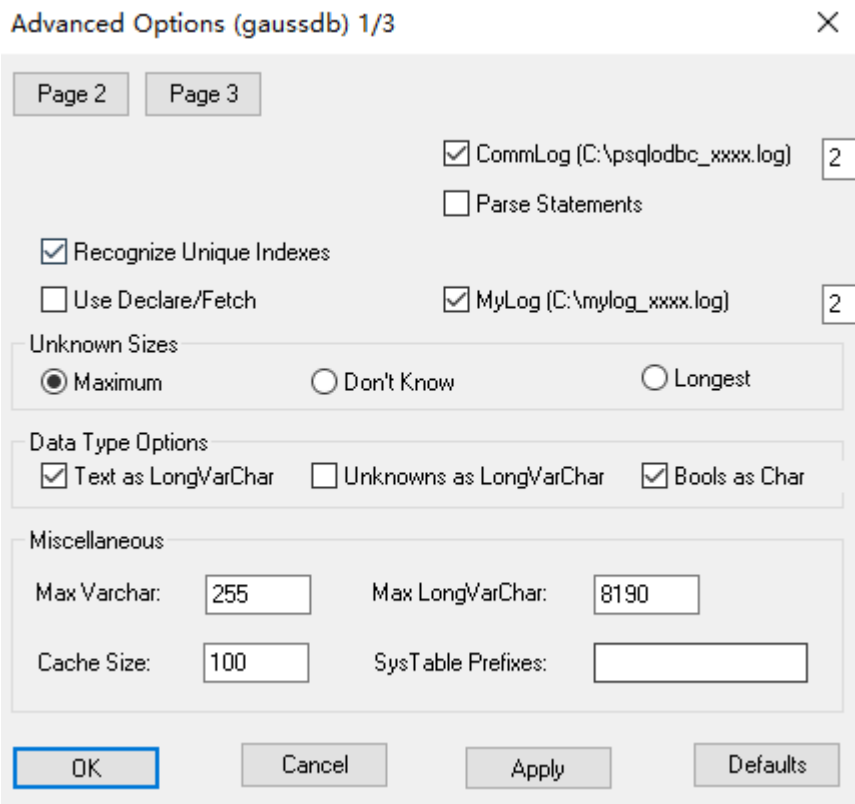
Options

Datasource Global Manage DSN Save Test Cancel

参数说明请参见[Linux下配置数据源](#)文件配置参数。

其中单击Datasource可以选择配置是否打印日志，单击后弹出对话框如下图7-45所示：

图 7-45 Advanced Options Page1



须知

此界面上配置的用户名及密码信息，将会被记录在Windows注册表中，再次连接数据库时不再需要输入认证信息。但是出于安全考虑，建议在单击“Save”按钮保存配置信息前，清空相关敏感信息，在使用ODBC的连接API时，再传入所需的用户名、密码信息。

步骤4 SSL模式。

将步骤3中设置窗口的“SSL Mode”选项调整至“require”。

表 7-37 Sslmode 的可选项及其描述

| Sslmode | 是否会启用SSL加密 | 描述                                          |
|---------|------------|---------------------------------------------|
| disable | 否          | 不使用SSL安全连接。                                 |
| allow   | 可能         | 如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。 |

| Sslmode     | 是否会启用SSL加密 | 描述                                                                                |
|-------------|------------|-----------------------------------------------------------------------------------|
| prefer      | 可能         | 如果数据库支持，那么首选使用SSL安全加密连接，但不验证数据库服务器的真实性。                                           |
| require     | 是          | 必须使用SSL安全连接，但是只做了数据加密，并不验证数据库服务器的真实性。                                             |
| verify-ca   | 是          | 必须使用SSL安全连接，并且验证数据库是否具有可信证书机构签发的证书。当前windows ODBC不支持cert方式认证。                     |
| verify-full | 是          | 必须使用SSL安全连接，在verify-ca的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。当前windows ODBC不支持cert方式认证。 |

### 说明

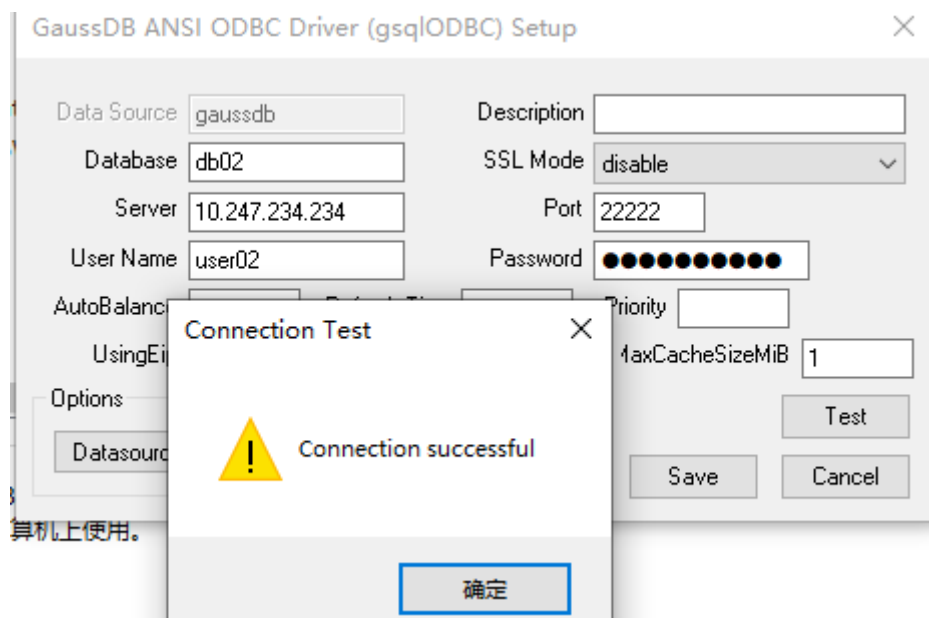
用户通过ODBC连接GaussDB服务器时，可以通过开启SSL加密客户端和服务端之间的通讯。在使用SSL时，默认用户已经获取了服务端和客户端所需要的证书和私钥文件，关于证书等文件的获取请参考Openssl相关文档和命令。

### 步骤5 测试连接。

单击Test进行测试。

- 如果显示如下，则表明配置正确，连接成功。

图 7-46 配置成功



- 若显示ERROR信息，则表明配置错误。请重新检查上述配置是否正确。

## 说明

目前通过ODBC连接数据库时，会设置以下内核参数：

```
SET extra_float_digits = 2;
SET DateStyle = 'ISO';
```

这些参数可能会导致ODBC客户端的行为与gsqldb客户端的行为不一致，例如，Date数据显示方式、浮点数精度表示等。如果实际期望和这些配置不符，建议在ODBC应用代码中显式设定这些参数。

M-Compatibility模式数据库下，extra\_float\_digits的默认值为0。

## ----结束

## 操作示例

以开发源程序DBtest.c为例，此示例演示如何通过ODBC获取和处理GaussDB中的数据。

前提条件：数据源已配置成功（完整示例请参见[获取和处理数据库中的数据](#)章节）。

```
// DBtest.c (compile with: libodbc.so)
// 程序头文件和全局变量请参考完整示例

// 申请环境句柄。
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
 printf("Error AllocHandle\n");
 exit(0);
}

// 设置版本信息（环境属性）。
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

// 申请连接句柄。
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
 SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
 exit(0);
}

// 获取用户名和用户密码。
char *userName;
userName = getenv("EXAMPLE_USERNAME_ENV");
char *password;
password = getenv("EXAMPLE_PASSWORD_ENV");

// 设置连接属性。
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER *)SQL_AUTOCOMMIT_ON, 0);

// 连接数据库，这里的userName与password分别表示连接数据库的用户名和用户密码。
// 如果odbc.ini文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦odbc.ini
// 权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
 (SQLCHAR*) userName, SQL_NTS, (SQLCHAR*) password, SQL_NTS);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
 printf("Error SQLConnect %d\n", V_OD_erg);
 SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
 exit(0);
}
printf("Connected !\n");
```

### 7.25.2.3.4 执行 SQL 语句

#### 说明

- ODBC为应用程序与数据库的中心层，负责把应用程序发出的SQL指令传到数据库当中，自身并不解析SQL语法。故在应用程序中写入带有保密信息的SQL语句时（如明文密码），保密信息会被暴露在驱动日志中。
- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，如果其中有一个语句失败，那么整个请求都将会被回滚。

执行SQL语句相关操作示例如下（完整示例请参见[获取和处理数据库中的数据](#)章节）。

```
// 设置语句属性。
SQLSetStmtAttr(V_OD_hstmt, SQL_ATTR_QUERY_TIMEOUT, (SQLPOINTER *)3, 0);

// 申请语句句柄。
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);

// 直接执行SQL语句。
SQLExecDirect(V_OD_hstmt, "drop table IF EXISTS customer_t1", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name
VARCHAR(32));", SQL_NTS);
SQLExecDirect(V_OD_hstmt, "insert into customer_t1 values(25, 'li')", SQL_NTS);

// 准备执行。
SQLPrepare(V_OD_hstmt, "insert into customer_t1 values(?)", SQL_NTS);

// 绑定参数。
SQLBindParameter(V_OD_hstmt,1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
&value, 0, NULL);

// 执行准备好的语句。
SQLExecute(V_OD_hstmt); SQLExecDirect(V_OD_hstmt, "select c_customer_sk from customer_t1",
SQL_NTS);
```

### 7.25.2.3.5 处理结果集

ODBC处理结果集是从数据库中获取数据并将其提供给应用程序进行处理，作用包括但不限于：检索数据、数据展示、数据处理、数据传输和业务逻辑实现等。

操作示例如下（完整示例请参见[获取和处理数据库中的数据](#)章节）。

```
// 在执行完成SQL语句后，获取结果集某一列的属性。
SQLColAttribute(V_OD_hstmt,1,SQL_DESC_TYPE,typename,100,NULL,NULL);
printf("SQLColAttribute %s\n",typename);

// 绑定结果集。
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER)&V_OD_buffer,150,
(SQLLEN *)&V_OD_err);

// 通过SQLFetch取结果集中数据。
V_OD_erg=SQLFetch(V_OD_hstmt);

// 通过SQLGetData获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
 SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER)&V_OD_id,0,NULL);
 printf("SQLGetData ----ID = %d\n",V_OD_id);
 V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");
```

### 7.25.2.3.6 关闭连接

关闭数据库连接操作示例如下（完整示例请参见[获取和处理数据库中的数据](#)章节）。



```
// 断开数据源连接并释放句柄资源。
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt);
SQLDisconnect(V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env);
```

### 7.25.2.3.7 最佳实践

GaussDB为用户提供ODBC的最佳实践。

具体请参见《最佳实践》中“ODBC最佳实践”章节。

### 7.25.2.4 常见问题

#### 连接数据源时报错：[UnixODBC][Driver Manager]Can't open lib 'xxx/xxx/gsqlodbcw.so':file not found

可能的原因及确认方法：

- odbcinst.ini文件中配置的路径不正确。  
确认的方法：执行ls命令查询错误信息中的路径，以确保该gsqlodbcw.so文件存在，同时具有执行权限。
- gsqlodbcw.so的依赖库不存在，或者不在系统环境变量中。  
确认的方法：执行ldd命令查询错误信息中的路径，如果是缺少libodbc.so.1等UnixODBC的库，那么按照[连接数据库](#)中的步骤重新配置UnixODBC，并确保它的安装路径下的lib目录添加到了LD\_LIBRARY\_PATH中。如果重装仍无法解决，可以手动将数据库安装包下的unixodbc/lib下的内容复制到UnixODBC的安装路径下的lib目录。如果是缺少其他库，请将ODBC驱动包中的lib目录添加到LD\_LIBRARY\_PATH中。如果缺少其他标准库，请自行安装。

#### 连接数据源时报错：[UnixODBC]connect to server failed: no such file or directory.

可能的原因及确认方法：

- 配置了错误的/不可达的数据库地址，或者端口。  
请检查数据源配置中的Servername及Port配置项。
- 服务器侦听不正确。  
如果确认Servername及Port配置正确，请根据[Linux下配置数据源](#)或者[Windows下配置数据源](#)中数据库服务器的相关配置，确保数据库侦听了合适的网卡及端口。
- 防火墙及网闸设备。  
请确认防火墙设置，将数据库的通信端口添加到可信端口中。  
如果有网闸设备，请确认相关的设置。

#### 错误提示：authentication method 10 not supported.

可能的原因：

使用开源客户端时可能会遇到此问题。数据库中存储的密码校验只存储了SHA256格式哈希，而开源客户端只识别MD5校验，双方校验方法不匹配报错。

## 📖 说明

- 数据库并不存储用户密码，只存储用户密码的哈希码。
- 当用户更新用户密码或者新建用户时，数据库会同时存储两种格式的哈希码，这时将兼容开源的认证协议。
- 当旧版本升级到新版本时，由于哈希的不可逆性，所以数据库无法还原用户密码，进而生成新格式的哈希，所以仍然只保留了SHA256格式的哈希，导致仍然无法使用MD5做密码认证。
- MD5加密算法安全性低，存在安全风险，建议使用更安全的加密算法。

要解决该问题，可以更新用户密码，或者新建一个用户，赋予同等权限，使用新用户连接数据库。

## 错误提示：Driver's SQLAllocHandle on SQL\_HANDLE\_DBC failed.

可能的原因：

可执行文件（比如UnixODBC的isql，以下都以isql为例）与数据库驱动（gsqldbcs.so）依赖于不同的ODBC的库版本：libodbc.so.1或者libodbc.so.2。此问题可以通过如下方式确认：

```
ldd `which isql` | grep odbc
ldd gsqlodbcw.so | grep odbc
```

若输出的libodbc.so后的后缀数字不同或者指向不同的磁盘物理文件，则可确认存在版本冲突问题。isql与gsqldbcs.so都会要求加载libodbc.so，若它们加载了不同的物理文件，便会导致两套完全同名的函数列表同时出现在同一个可见域里（UnixODBC的libodbc.so.\*的函数导出列表完全一致），产生冲突，无法加载数据库驱动。

解决方法：

确保需保留的UnixODBC版本后，卸载冲突版本（如libodbc.so.2），对保留的libodbc.so.1文件创建同名但后缀为.so.2的软链接（即新建的libodbc.so.2指向libodbc.so.1）。

## 字符编码出现乱码

可能的原因：

客户端/服务端字符集不一致（如客户端使用GBK，服务端使用UTF8）。

解决方法：

统一应用、ODBC驱动、数据库三端的编码。

## 错误提示：unsupported frontend protocol 3.51: server supports 1.0 to 3.0.

可能的原因：

目标数据库版本过低，或者目标数据库为开源数据库。

解决方法：

使用对应版本的数据库驱动连接目标数据库。

## 7.26 基于 hashbucket 表的在线扩容

### 7.26.1 特性概述

#### 功能描述

随着集群部署时间及业务的增长，数据库在运行性能及存储上逐渐会达到瓶颈，需要通过增加主机来提升集群的性能及存储能力，实现业务的平稳运行。GaussDB提供了gs\_expand工具进行主机扩容和数据重分布的操作。

#### 用户场景

当集群性能及存储能力随业务增长达到瓶颈时，需新增主机以提升集群性能与存储容量，确保业务持续稳定运行。离线扩容会阻塞业务流程，因此需通过在线扩容保障扩容重分布期间业务正常运行。

#### 规格约束

- 集群扩容时，请勿修改\$PGHOST目录下的expandimplolap\_step/expandimplolap\_step.dat文件，否则会影响扩容阶段和重分布阶段的流程重新执行。
- 扩容的DN参数配置和分配的物理资源需与现有节点的DN保持一致；当新扩容的服务器资源更多时，可以配置更多DN数量。
- 新增主机机器至少满足集群的安装要求，新集群整体性能取决于集群中配置最低的机器。约束条件：
  - 两副本集群新增主机数大于等于2。
  - 其它类型集群新增主机数大于等于3。
  - 新增主机的locale、encoding等与原有主机保持一致。
  - 新主机完成集群安装路径和用户组规划。
- 新增主机的操作系统与已有集群主机操作系统处于同一个大版本，支持与小版本混搭。
- 若新增主机硬件规格（内存/硬盘）低于现有集群，需要逐步将性能较低的服务器替换为性能更高的服务器以满足集群的性能要求。
- 在前一次扩容结束后才能再次扩容。例如，扩容过程中由于断电等原因导致扩容未完成时，不允许再次进行扩容，需要先修复结束前一次扩容。
- 若存在第三方插件，扩容后需手动COPY相关依赖到新扩节点。
- 异构扩容（扩容服务器与现有集群服务器的硬件规格不一致）时，同批次扩容服务器的软硬件规格需保持一致。
- 数据库版本与OM版本不一致（数据库代码与OM代码不来自同一软件包）时，不支持扩容操作。
- 扩容资源使用情况：
  - 极限性能场景（每批bucket数量为1024，每次bucket上线最多1024）不控制I/O资源使用。
  - 在线场景（默认参数配置，每批bucket数量64，每次bucket上线最多64），磁盘I/O占用最大值的20%。

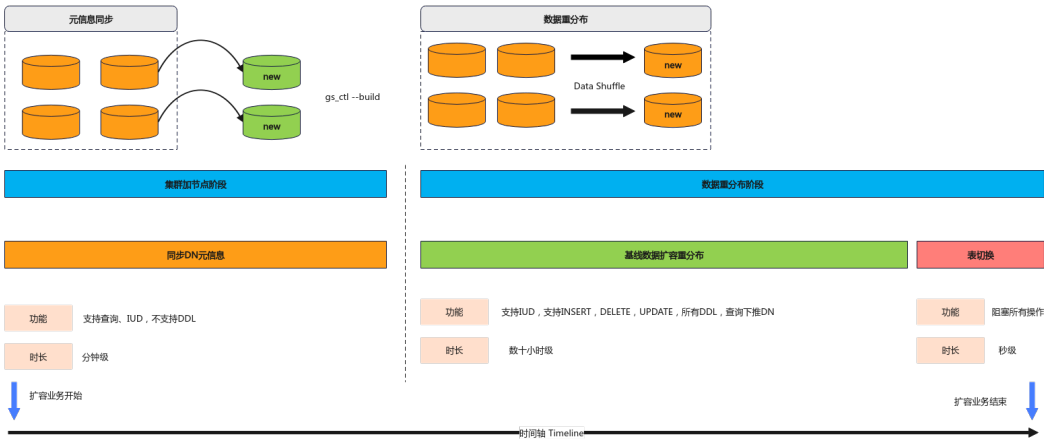
- I/O及追增Xlog日志占用少量磁盘空间，受保留Xlog数量参数影响。
- 存在CBI索引时，在线索引类型转换可能出现影响业务性能及抢占资源等现象，将继承在线建索引规格约束。

技术架构

在线扩容过程包含两个核心阶段：集群加节点阶段和数据重分布阶段。

- 集群加节点阶段：对新加入的DN或CN节点进行基础元信息的同步。
- 数据重分布阶段：将数据从旧集群迁移至新集群。

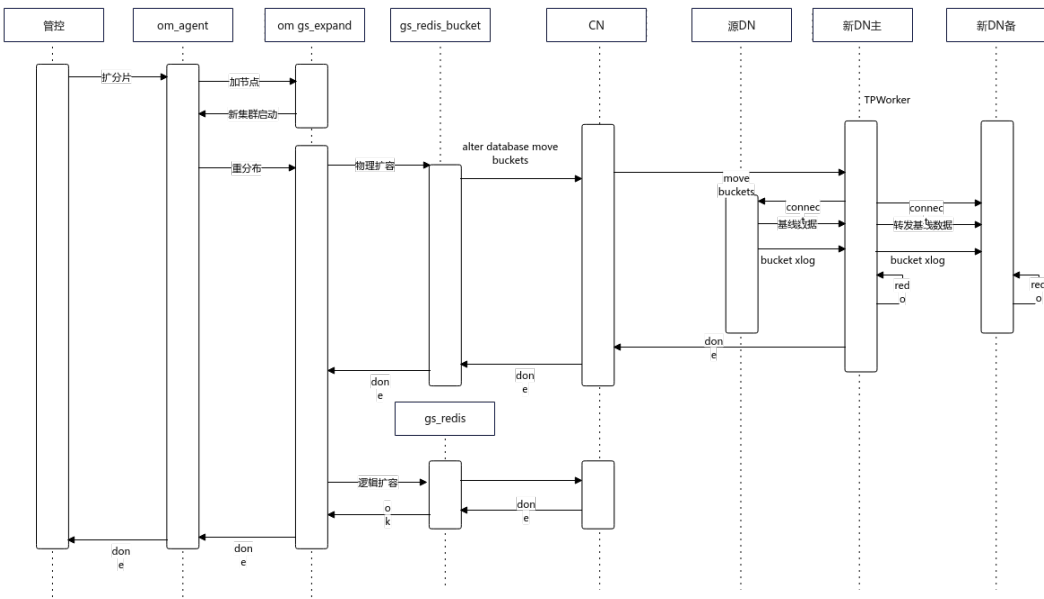
图 7-47 技术架构



7.26.2 技术原理

核心机制

图 7-48 核心机制



 说明

由于gs\_expand工具集成了物理扩容和逻辑扩容功能，因此图示中包含逻辑扩容相关内容。建议用户在物理扩容操作时避免数据库内出现bucket表与非bucket表混用情况。

性能指标

集群数据节点读带宽500MB/s、写带宽800MB/s，配置万兆网络。当每个DN独占磁盘、业务负载低峰且不做资源管控（资源管控级别为f，表内并发度8，扫描并发度4）时，单DN分片1TB数据翻倍扩容耗时如下：

- 普通表：3小时完成。
- hashbucket表：2小时完成（每批bucket数量1024，单次bucket上线最大1024）。

接口介绍

- gs\_expand  
gs\_expand用于在业务规模扩展及数据量增长过程中，当集群性能与存储资源达到瓶颈时，通过新增主机节点提升集群整体处理能力和存储容量。

 说明

- 建议用户使用此接口完成扩容加节点和数据重分布操作。
- gs\_redis\_bucket  
gs\_redis\_bucket用于对扩容加节点后的集群中hashbucket表执行数据重分布，主要由OM脚本调用执行。

 说明

- 不建议用户直接调用此接口，建议用户使用gs\_expand工具完成数据重分布操作。
- gs\_redis  
gs\_redis用于对扩容后的集群中非hashbucket表执行数据重分布，同样由OM脚本调用执行。

 说明

- 不建议用户直接调用此接口，建议用户使用gs\_expand工具完成数据重分布操作。
- 系统函数

| 函数名称                           | 说明                                                        |
|--------------------------------|-----------------------------------------------------------|
| gs_redis_get_plan              | 获取完整的迁移计划。只支持在扩容期间调用；只支持管理员权限用户查询；只支持CN上调用。               |
| gs_redis_get_bucket_statistics | 获取日志流传输状态。只支持在扩容期间调用；只支持管理员权限用户查询。                        |
| gs_redis_set_distributed_db    | 设置当前正在hashbucket重分布的数据库。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。 |

| 函数名称                                       | 说明                                                                        |
|--------------------------------------------|---------------------------------------------------------------------------|
| gs_redis_hashbucket_update_segment_header  | 更新当前数据库所有hashbucket表的header。只支持在扩容期间调用；只支持管理员权限用户查询；只支持CN上调用。             |
| gs_redis_local_get_segment_header          | 获取段页式表header。只支持DN上调用。                                                    |
| gs_redis_local_update_segment_header       | 更新段页式表header。只支持DN上调用。                                                    |
| gs_redis_hashbucket_update_inverse_pointer | 更新当前数据库所有hashbucket表本批次bucket的反向指针。只支持在扩容期间调用；只支持管理员权限用户查询；只支持CN上调用。      |
| gs_redis_local_get_inverse_pointer         | 获取反向指针。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。                                  |
| gs_redis_local_update_inverse_pointer      | 记录并更新反向指针的Xlog。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。                          |
| gs_redis_local_set_hashbucket_frozenxid    | 修改hashbucket表在系统表中的relfrozenxid64值。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。      |
| gs_redis_set_hashbucket_frozenxid          | 修改新DN节点hashbucket表在系统表中的relfrozenxid64值。只支持在扩容期间调用；只支持管理员权限用户查询；只支持CN上调用。 |
| gs_redis_set_nextxid                       | 修改DN节点next_xid值。只支持在扩容期间调用。只支持管理员权限用户查询；只支持DN上调用。                         |
| gs_redis_set_csn                           | 修改DN节点next_csn值。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用；只支持在GTM_FREE模式下调用。       |
| gs_redis_check_bucket_flush                | 查询RTO回放是否完成私有buffer的全量刷盘。只支持在扩容期间调用；只支持管理员权限用户查询。                         |
| gs_redis_get_flush_page_lsn                | 查询bucket扩容回放的刷页信息。暂不支持发布版本调用；只支持在扩容期间调用；只支持管理员权限用户查询。                     |
| gs_redis_show_bucketxid                    | 查询指定bucket对应的bucketxid。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。                  |
| gs_redis_drop_bucket_files                 | 删除源节点已经物理搬迁的文件。只支持在扩容期间调用；只支持管理员权限用户查询；只支持CN上调用。                          |
| gs_redis_local_drop_bucket_files           | 删除对应的bucket list。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。                        |

| 函数名称                             | 说明                                                                  |
|----------------------------------|---------------------------------------------------------------------|
| gs_redis_local_drop_bucket_files | 用于rangebucket表删除对应数据库的bucket list。只支持在扩容期间调用；只支持管理员权限用户查询；只支持DN上调用。 |
| gs_redis_get_undo_state          | 获取扩容重分布期间的Undo回滚状态。仅支持在扩容期间调用；仅支持管理员权限用户查询。                         |

#### 📖 说明

- 详细描述请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > bucket系统函数”章节。
- 不建议用户直接调用以上系统函数接口，建议用户使用gs\_expand工具完成扩容加节点和数据重分布操作。
- 表  
以下表的创建发生在进行逻辑扩容操作期间。

| 表名                    | 说明                            |
|-----------------------|-------------------------------|
| redis_progress_detail | 记录非hashbucket表扩容重分布执行时间。      |
| redis_status          | 记录集群重分布总体进度。                  |
| pgxc_redistb          | 记录需要重分布的用户表的信息，扩容重分布结束后会自动删除。 |

## 7.26.3 操作指导

### 前提条件

- 集群状态必须为Normal或Degraded状态，集群Degraded状态仅支持实例异常引发的集群状态异常，不支持VM异常（指非软件类异常，例如：磁盘故障、通信故障）导致的集群状态异常。
- 请确保实例状态异常节点的网络连接正常。
- 实例状态：CN全部正常，GTM、CMS至少有主节点存活，ETCD实例状态满足多数正常。
- DN实例状态有以下三种情况：
  - 两副本集群单个分片内至少主DN实例正常（hashbucket表扩容要求主备DN都正常）。
  - 两AZ集群DN实例单个分片至少存活一主一备。
  - 除了以上两种之外的其他情况DN实例单个分片异常数量小于一半。
- 集群扩容要求整个集群未被锁定，集群配置文件的配置信息正确且与当前集群配置一致。

- 已按照扩容的集群配置文件执行过前置脚本gs\_preinstall，即预安装操作执行完毕。
- 新增主机与现有集群之间通信已建立且网络正常。
- 扩容前需确保集群所有CN可连接，在集群用户下执行gs\_check -i CheckDBConnection命令检查CN是否可连接。

```
Parsing the check items config file successfully
Distribute the context file to remote hosts successfully
Start to health check for the cluster. Total Items:1 Nodes:3

Checking... [=====] 1/1
Start to analysis the check result
Start to analysis the item CheckDBConnection
analysis item: CheckDBConnection, use time: 0.021239858004264534
Analysis the item CheckDBConnection finished
AnalysisResult total time: 0.021775729983346537
CheckDBConnection.....OK
The item run on 3 nodes. success: 3

Analysis the check result successfully
Success. All check items run completed. Total:1 Success:1
For more information please refer to /script/gspylib/inspection/output/
CheckReport_202505083489941202.tar.gz
```

- 扩容前需确保每个库存在public schema，否则扩容会失败。
- 扩容前需删除各个CN和DN节点上的逻辑复制槽。  
依次在各个CN和DN节点上执行，获取所有的逻辑复制槽。  
gaussdb=# SELECT \* FROM pg\_catalog.pg\_get\_replication\_slots() WHERE slot\_type='logical';  
  
依次在各个CN和DN节点上执行，删除查询到的逻辑复制槽。  
gaussdb=# SELECT \* FROM pg\_drop\_replication\_slot('slot\_name');
- 扩容的新节点不能包含ETCD组件。
- 扩容前需退出创建了临时表的客户端连接，因为在扩容过程中及扩容成功后临时表会失效，操作临时表也会失败。
- 如果数据库中包含的对象数量（数据表、视图、索引）超过50万，为了提高性能且避免出现内存问题，建议扩容前将CN和DN的max\_prepared\_transactions和max\_locks\_per\_transaction参数分别设置为1000和512。如果max\_prepared\_transactions和max\_locks\_per\_transaction参数值大于前述的建议值，则无需设置。max\_prepared\_transactions参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。max\_locks\_per\_transaction参数具体说明，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 锁管理”章节。
- 扩容前，需要检查已有集群的comm\_max\_datanode参数值。
  - 使用gsqll连接CN执行show comm\_max\_datanode。
  - 如果comm\_max\_datanode的值小于扩容后集群的DN数，则需要设置comm\_max\_datanode参数，具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 连接和认证 > 通信库参数”章节。假如原有集群DN数为128，新扩128个DN时，需在扩容前把comm\_max\_datanode参数设置为256。
- 扩容执行到新集群启动阶段后，系统无法回滚到扩容前状态。
- 重分布资源管控对象redisuser、redisrespool、redisclass和redisgrp不能被占用。如果集群中存在redisuser用户，扩容将直接报错退出，且扩容期间禁止创建redisuser用户。
- 重分布过程中需避免以下操作：
  - 将其他用户绑定到重分布专用的资源池redisrespool。



- 使用gs\_cgroup -M等操作破坏当前Cgroups配置。若用户在Cgroups出现问题时使用gs\_cgroup --revert恢复默认配置，则需重启集群以确保新建立的redisclass和redisgrp生效。
- 若当前集群有对hashbucket的表开启OLTP表压缩，需要等待全部压缩任务结束，并关闭自动调度任务（扩容期间不支持压缩调度）。
- 重分布前校验无效索引残留，需遍历数据库执行如下命令，如果返回值大于等于1，用户需手动失效索引，即执行ALTER INDEX操作。

```
gaussdb=# select count(*) from pg_index where indisvalid = 'f' and indisusable = 't';
count

 0
(1 row)
gaussdb=# ALTER INDEX i1 UNUSABLE;
ALTER INDEX
```

## 命令格式

- 集群扩容  

```
gs_expand -t dilatation -X XMLFILE [-l LOGFILE] [--parallel-jobs=NUMBER] [--time-out=SECS] [--dilatation-mode=MODE] [--nodegroup-name=LCGROUPNAME | --addto-elastic-group]
```
- 重分布参数设置  

```
gs_expand -t parametercfg [--lockwait-timeout=MSEC] [--lockwait-interval=SECS] [--trylock-threshold=NUMBER] [--enable-cancel=Boolean] [--last-catchup-threshold=MSEC] [--catchup-times=NUMBER] [--write-error-mode=Boolean] [--catchup-query-dop=NUMBER] [--parallel-catchup-threshold=NUMBER] [--parallel-reindex-jobs=NUMBER] [--redis-retry-times=NUMBER] [--redis-retry-interval=NUMBER] [--redis-bucket-parse-redo-num=NUMBER] [--redis-bucket-workers-per-paser=NUMBER] [--redis-bucket-keep-bucketxlogs=NUMBER]
```
- 数据重分布  

```
gs_expand -t redistribute [--parallel-jobs=NUMBER] [--resource-level=f/h/m/l] [--failure-limit=NUMBER] [--build-redisb] [--redis-mode=MODE] [-l LOGFILE] [--priority-tables=FILE] [--exclude-tables=FILE] [--time-out=SECS] [--max-worker] [--max-dop] [--join-tables=FILE]
```
- 回滚  

```
gs_expand -t crashclean -X XMLFILE [-l LOGFILE]
```

## 物理扩容注意事项

- 集群重分布状态下（即加节点并创建了新node group后）不支持绑定node group。
- hashbucket表删的部分索引、表达式索引形式的GSI会在重分布过程中进行离线重建，重建期间会阻塞业务，非部分索引、非表达式索引形式的GSI，在重分布过程中进行在线重建，不会阻塞业务。加节点、重分布的全程GSI保持有效。
- 在hashbucket表扩容加节点阶段，需要在创建node group前遍历数据库更新元数据，若此时存在频繁DROP DATABASE操作，加节点过程重试次数超出上限后报错失败，需待集群上此类DDL变更结束后重新执行扩容。
- 若hashbucket表扩容加节点阶段检测到wait\_clean\_gpi=y的分区残留，将报错“Please manually perform the 'vacuum' operation on these databases.”，需按提示连接库执行VACUUM后重新执行扩容。
- 若重分布期间主DN实例（含旧DN分片和新扩容DN分片）或新增分片的备DN实例故障，则会导致重分布失败退出，需重新执行重分布流程。
- 当前版本hashbucket表扩容支持重分布前新增分片的实例故障且满足多数派（主DN与备DN存活的总个数超过该分片副本数的1/2）可用场景下的扩容：
  - 对于一主多备场景，满足多数派即可支持扩容，如一主两备支持单个备DN故障状态下的扩容。

- 对于一主一备一logger场景，只支持logger实例故障状态下的扩容，不支持备DN故障状态下扩容。
- 对于一主一备场景，仅支持主备均存活时扩容。
- 当前版本支持hashbucket表在扩容期间执行CREATE TABLE/DROP TABLE/TRUNCATE TABLE三种hashbucket表级DDL操作，其他DDL当前均不支持。
  - 除toast索引外，若在CREATE/DROP/TRUNCATE目标hashbucket表时进行级联操作索引，将导致表级DDL失败。
  - hashbucket分区表仅支持整表的CREATE/DROP/TRUNCATE操作，不支持对单个分区进行此类操作。
  - 在扩容加节点阶段更新元数据或bucket重分布更新反向指针时，若执行DROP TABLE则存在窗口期导致以上操作失败，需要重新执行扩容。
- 扩容期间不建议对CBI索引或带CBI索引的表进行删除操作，可能导致扩容重分布失败，需要重新执行扩容。
- hashbucket表扩容期间不支持连接CN调用global\_space\_shrink对段页式文件进行缩容，由于资源争抢问题，同样不建议连接DN进行段页式文件缩容。
- hashbucket表扩容会忽略read-only模式的参数，仍采用在线模式执行。扩容过程中使用redisuser用户进行重分布操作，以避免线程残留导致的扩容失败，且不会自动重试。
- hashbucket表扩容期间不支持的特性：逻辑复制、细粒度备份恢复、DN在线缩容、二级分区、node group迁移。
- 在hashbucket表扩容期间，涉及Ustore hashbucket表CBI索引的事务，在提交前会缓存至内存，提交后释放。若此部分内存使用量累计超过1GB，扩容流程会报错终止。报错后可通过重新执行扩容流程进行重试。
- 闪回查询尝试访问hashbucket表扩容期间或扩容前的snapshot时，可能由于CSN时间过长报错“snapshot csn xxx less or equal to bucketcsn xxx from old DN”。
- 扩容前的长事务快照读取新节点迁移后上线的数据可能报错“snapshot csn less or equal to bucketcsn from old DN”。

## 操作步骤

### 说明

- 请确保有可用分布式集群。
- 集群部署配置文件以clusterconfig.xml为例（可自行修改），以下操作步骤使用omm用户执行。
- 在线扩容主要分为集群加节点阶段和重分布阶段：
  - [步骤1](#)至[步骤7](#)为集群加节点阶段。
  - [步骤8](#)为重分布准备阶段。
  - [步骤9](#)至[步骤11](#)为重分布阶段。

**步骤1** 修改集群部署配置文件clusterconfig.xml，添加新增主机的配置信息。

1. 在集群信息的nodeNames字段末尾追加所有新增主机的名称（原主机名称顺序保持不变），同时在sqlExpandNames字段中补充所有新增的主机名称。例如，gaussdb04、gaussdb05和gaussdb06为新增主机名称，192.168.255.253、192.168.255.254和192.168.255.255为对应IP地址。

```
<CLUSTER>
<PARAM name="clusterName" value="mppdbCluster" />
<PARAM name="nodeNames"
```

```
value="gaussdb01,gaussdb02,gaussdb03,gaussdb04,gaussdb05,gaussdb06" />
<PARAM name="backIp1s"
value="192.168.255.250,192.168.255.251,192.168.255.252,192.168.255.253,192.168.255.254,192.168.25
5.255" />
<PARAM name="gaussdbAppPath" value="/data/lixy1/install/app" />
<PARAM name="gaussdbLogPath" value="/data/lixy1/install/log" />
<PARAM name="gaussdbToolPath" value="/data/lixy1/tool/wisquery" />
<PARAM name="clusterType" value="single-primary-multi-standby"/>
<PARAM name="sqlExpandNames" value="gaussdb04, gaussdb05, gaussdb06"/>
</CLUSTER>
```

## 2. 在配置文件clusterconfig.xml中添加新增主机的实例信息。

```
<!-- plat4上的节点部署信息 -->
<DEVICE sn="1000004">
 <PARAM name="name" value="gaussdb04"/>
 <PARAM name="azName" value="AZ1"/>
 <PARAM name="azPriority" value="1"/>
 <!-- 如果服务器只有一个网卡可用，将backIp1和sshIp1配置成同一个IP -->
 <PARAM name="backIp1" value="192.168.255.253"/>
 <PARAM name="sshIp1" value="192.168.255.253"/>
 <!-- cm -->
 <PARAM name="cmServerPortStandby" value="25500"/>
 <PARAM name="cmDir" value="/data/lixy1/install/cmserver"/>

 <!-- cn -->
 <PARAM name="cooNum" value="0"/>
 <PARAM name="cooPortBase" value="25108"/>
 <PARAM name="cooListenIp1" value="192.168.255.253"/>
 <PARAM name="cooDir1" value="/data/lixy1/install/coordinator"/>
 <!-- etcd -->
 <!-- DN -->
 <PARAM name="dataNum" value="1"/>
 <PARAM name="dataPortBase" value="26600"/>
 <PARAM name="dataNode1" value="/data/lixy1/install/master4,gaussdb05,/data/lbs2/install/
slave4_1,gaussdb06,/data/lbs3/install/slave4_2"/>
</DEVICE>
<!-- plat5上的节点部署信息 -->
<DEVICE sn="1000005">
 <PARAM name="name" value="gaussdb05"/>
 <PARAM name="azName" value="AZ1"/>
 <PARAM name="azPriority" value="1"/>
 <!-- 如果服务器只有一个网卡可用，将backIp1和sshIp1配置成同一个IP -->
 <PARAM name="backIp1" value="192.168.255.254"/>
 <PARAM name="sshIp1" value="192.168.255.254"/>
 <!-- cm -->
 <PARAM name="cmServerPortStandby" value="25500"/>
 <PARAM name="cmDir" value="/data/lixy1/install/cmserver"/>

 <!-- cn -->
 <PARAM name="cooNum" value="0"/>
 <PARAM name="cooPortBase" value="25108"/>
 <PARAM name="cooListenIp1" value="192.168.255.254"/>
 <PARAM name="cooDir1" value="/data/lixy1/install/coordinator"/>
 <!-- etcd -->
 <!-- DN -->
 <PARAM name="dataNum" value="1"/>
 <PARAM name="dataPortBase" value="26800"/>
 <PARAM name="dataNode1" value="/data/lixy1/install/master5,gaussdb04,/data/lbs2/install/
slave5_1,gaussdb06,/data/lbs3/install/slave5_2"/>
</DEVICE>
<!-- plat6上的节点部署信息 -->
<DEVICE sn="1000006">
 <PARAM name="name" value="gaussdb06"/>
 <PARAM name="azName" value="AZ1"/>
 <PARAM name="azPriority" value="1"/>
 <!-- 如果服务器只有一个网卡可用，将backIp1和sshIp1配置成同一个IP -->
 <PARAM name="backIp1" value="192.168.255.255"/>
 <PARAM name="sshIp1" value="192.168.255.255"/>
 <!-- cm -->
 <PARAM name="cmServerPortStandby" value="25500"/>
```

```
<PARAM name="cmDir" value="/data/lixy1/install/cmserver"/>

<!-- cn -->
<PARAM name="cooNum" value="0"/>
<PARAM name="cooPortBase" value="25108"/>
<PARAM name="cooListenIp1" value="192.168.255.255"/>
<PARAM name="cooDir1" value="/data/lixy1/install/coordinator"/>
<!--etcd-->
<!--DN-->
<PARAM name="dataNum" value="1"/>
<PARAM name="dataPortBase" value="27000"/>
<PARAM name="dataNode1" value="/data/lixy1/install/master6,gaussdb04,/data/lbs2/install/
slave6_1,gaussdb05,/data/lbs3/install/slave6_2"/>
</DEVICE>
```

#### 说明

- 新增节点不能包含主备GTM和主备CM Server。
- 新增节点上的DN必须自行成环，以避免破坏原有主备关系。

**步骤2** 以root用户登录服务器。

**步骤3** 进入安装包所在目录。例如，安装包路径为/opt/software/gaussdb，实际路径请按环境替换。

```
cd /opt/software/gaussdb
```

**步骤4** 执行环境准备操作，解压安装包，进入安装包所在目录的script目录，使用root用户执行预安装操作。

```
./gs_preinstall -U omm -G dbgrp -X /opt/software/gaussdb/clusterconfig_dilatation.xml --alarm-type=5
Parsing the configuration file.
Successfully parsed the configuration file.
Check the preinstallation result file.
Successfully delete task result file
Installing the tools on the local node.
Successfully installed the tools on the local node.
Are you sure you want to create trust for root (yes/no)?yes
Please enter password for root
Password:
Successfully created SSH trust for the root permission user.
Setting HOST_IP env
Successfully set HOST_IP env.
set hotpatch env.
clear hotpatch env.
Distributing package.
Successfully distribute local lib to remote pylib.
Begin to distribute package to tool path.
Successfully distribute package to tool path.
Begin to distribute package to package path.
Successfully distribute package to package path.
Successfully distributed package.
Are you sure you want to create the user[omm] and create trust for it (yes/no)? yes
Preparing SSH service.
Successfully prepared SSH service.
Installing the tools in the cluster.
Successfully installed the tools in the cluster.
Checking hostname mapping.
Successfully checked hostname mapping.
Creating SSH trust for [omm] user.
Please enter password for current user[omm]
Password:
Successfully created SSH trust for [omm] user.
Checking OS version.
Successfully checked OS version.
Creating cluster's path.
Successfully created cluster's path.
Set and check OS parameter.
Setting OS parameters.
```

```
Successfully set OS parameters.
Warning: Installation environment contains some warning messages.
Please get more details by "/opt/software/gaussdb/script/gs_checks -i A -h
gaussdb01,gaussdb02,gaussdb03,gaussdb04,gaussdb05,gaussdb06".
Set and check OS parameter completed.
Preparing CRON service.
Successfully prepared CRON service.
Setting user environmental variables.
Successfully set user environmental variables.
Configuring alarms on the cluster nodes.
Successfully configured alarms on the cluster nodes.
Setting the dynamic link library.
Successfully set the dynamic link library.
Setting pssh path
Successfully set pssh path.
Setting Cgroup.
Successfully set Cgroup.
Set ARM Optimization.
Successfully set ARM Optimization.
Setting finish flag.
Scene: non-huaweiyun
Successfully set finish flag.
Preinstallation succeeded.
Check the preinstallation result file.
Modify the permission on the preinstallation result file.
```

**步骤5** 切换至集群（omm）用户。

```
su - omm
```

**步骤6** 执行如下命令，设置扩容加节点和重分布参数。

若扩容重分布操作处于业务低谷期或者无业务期间，可设置参数redis-bucket-reindex-cbi-offline为true，并执行以下命令，否则不建议开启此参数，并跳过本步骤。

```
设置CBI索引的重建方式，若用户业务有低峰窗口，可以开启此参数提升离线扩容场景下CBI索引的重建性能。
gs_expand -t parametercfg --redis-bucket-reindex-cbi-offline=true
Redis parameters are set successfully.
```

**步骤7** 执行集群加节点操作。

```
参数建议使用默认值
gs_expand -t dilatation -X /opt/software/gaussdb/clusterconfig_dilatation.xml
Check cluster version consistency.
Successfully checked cluster version.
Distributing cluster dilatation XML file.
Checking static configuration.
Static configuration is matched with the old configuration file.
Checking the cluster status.
Performing health check.
Health check succeeded.
Backing up parameter files.
Successfully backed up parameter files.
Installing.
Deleting instances from ['gaussdb04', 'gaussdb05', 'gaussdb06'].
Uninstalling applications on ['gaussdb04', 'gaussdb05', 'gaussdb06'].
Successfully uninstalled applications on ['gaussdb04', 'gaussdb05', 'gaussdb06'].
Checking installation environment on all nodes.
Installing applications on all nodes.
Resume om hotpatch.
Successfully resume om hotpatch.
Synchronizing alarmItem configuration to new nodes.
Successfully synchronized alarmItem configuration to new nodes.
Installation is completed.
Configuring.
Configuring new nodes.
Checking node configuration on all nodes.
Initializing instances on all nodes.
Configuring pg_hba on all nodes.
Successfully configured new nodes.
Begin scpCertFile.
```

```
ScpCertFile is completed.
Rebuilding new nodes.
Locking cluster.
Successfully locked cluster.
Restoring new nodes.
Successfully restored new nodes.
Successfully rebuild new nodes.
Configuration is completed.
Starting new nodes.
Successfully started new nodes.
Synchronizing.
Updating cluster configuration.
Successfully updated cluster configuration.
Synchronization is completed.
Successfully enable node whiteList.
Update resource control config file result: Not found resource config file on this node..
Starting new cluster.
Waiting for the cluster status to become available.
.....
The cluster status is Degraded.
Unlocking cluster.
Successfully unlocked cluster.
Creating new node group.
Successfully created new node group.
Waiting for the redistributing status to become yes.
The cluster redistributing status is yes.
Successfully started new cluster.
Successfully disabled delay xlog recycle.
Starting timeseries table redistribute.
Dilatation succeed.
```

可以使用集群用户（omm）执行以下命令查看扩容加节点进度。

```
可以使用以下命令查询扩容加节点处于什么阶段
gs_expand -t progress -X /opt/software/gaussdb/clusterconfig_dilatation.xml
为加节点前
Querying current dilatation progress.
Current expansion progress is:
Status: Expansion completed or not started, Detail: No exist expansion progress and step file.

加节点开始，初始化阶段
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Init expand

加节点期间，安装新节点
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Install new nodes

加节点完成
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Begin redistribute
```

## 说明

- 扩容加节点出现错误后，系统会自动执行回滚，若回滚失败，则用户需要调用gs\_expand工具的crashclean命令进行清理，将集群恢复，清理完成后再次执行集群扩容命令，重新执行步骤7。  
gs\_expand -t crashclean -X /opt/software/gaussdb/clusterconfig\_dilatation.xml  
Distributing cluster dilatation XML file.  
Checking static configuration.  
Static configuration is matched with the old configuration file.  
Crash clean for extension.  
Checking the cluster status.  
Last time end with Config new nodes.  
Rolling back.  
Unlocking cluster.  
Successfully unlocked cluster.  
Deleting instances from ['gaussdb04', 'gaussdb05', 'gaussdb06'].  
Successfully deleted instances from ['gaussdb04', 'gaussdb05', 'gaussdb06'].  
Uninstalling applications on ['gaussdb04', 'gaussdb05', 'gaussdb06'].  
Successfully uninstalled applications on ['gaussdb04', 'gaussdb05', 'gaussdb06'].  
Rollback is completed.  
Cleaning backedup cluster parameter file.  
Successfully cleaned backedup cluster parameter file.  
Crashclean succeed.
- 扩容加节点成功后，重分布状态会变成Yes，为随后的数据重分布做准备。
- 扩容加节点过程中，如果新增主机网络连接中断，会导致扩容加节点操作失败，并在回滚时发生故障的主机程序和数据目录等不会回滚。此时集群已回滚至扩容加节点前状态，用户需手动清理故障主机，再重新执行扩容操作。
- 如果扩容加节点前修改过GUC参数listen\_addresses、local\_bind\_address、port、pgxc\_node\_name、pooler\_port、log\_directory和audit\_directory，扩容加节点成功后，新增主机无法同步这些参数在原主机上的值，会设置为默认值，此时需要修改新增主机上的GUC参数，各参数具体说明请参见《参考》中“数据库运行参数说明 > GUC参数说明”章节。

### 步骤8 执行如下命令，将集群中的实例恢复为初始配置的主备状态。

```
gs_om -t switch --reset
Operating: Switch reset.
cm_ctl: switchover successfully.
Operation succeeded: Switch reset.
```

### 步骤9 执行如下命令，设置重分布参数。

```
在线扩容重分布场景下，建议将redis-bucket-batchsize参数设置为10，redis-bucket-tx-bucket-max设置为20。
gs_expand -t parametercfg --redis-bucket-batchsize=10 --redis-bucket-tx-bucket-max=20
Redis parameters are set successfully.
```

### 步骤10 执行数据重分布操作。

```
parallel-jobs参数为指定数据重分布过程的并发执行任务个数，默认值为1，建议设置为4，最大值可为8，参考环境信息配置。
gs_expand -t redistribute --redis-mode=insert --parallel-jobs=4
Last time end with Begin redistribute.
Checking the cluster status.
Redistribution
Successfully setup resource control.
If you want to check the progress of hashbucket table redistribution, please check the log file on gaussdb01: /data/lixy1/install/log/lixy/bin/gs_redis/gs_redis_bucket-2025-05-08_144810.log.
Successfully do hashbucket redistribution.
If you want to check the progress of logic redistribution, please check the log file on gaussdb01: /data/lixy1/install/log/lixy/bin/gs_redis/gs_redis-2025-05-08_144816.log.
Successfully clear resource control.
Successfully do common redistribution.
No need to do timeseries redistribute.
Redistribution succeeded.
```

## 说明

- 重分布过程中如出现以下日志，说明正在对表进行重分布操作，可进入\$GAUSSLOG/bin/gs\_redis/目录下，打开对应日志，查询表的重分布情况。  
# 正在对hashbucket表进行重分布操作  
If you want to check the progress of hashbucket table redistribution, please check the log file on gaussdb01: \$GAUSSLOG/bin/gs\_redis/gs\_redis\_bucket-2025-05-08\_144810.log.  
# 正在对非hashbucket表进行重分布操作  
If you want to check the progress of logic redistribution, please check the log file on gaussdb01: \$GAUSSLOG/bin/gs\_redis/gs\_redis-2025-05-08\_144816.log.
- 在重分布执行过程中，用户应当避免执行超过20分钟的查询（在重分布执行时申请写锁的默认时间为20分钟）。否则可能导致重分布等待加锁超时失败。
- 重分布完成后，需检查CN和DN实例的default\_transaction\_isolation参数，确保所有实例参数都设置为“read committed”。default\_transaction\_isolation参数的具体说明请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 客户端连接缺省设置 > 语句行为”章节。

### 步骤11 连接默认的数据库，查看重分布状态。

```
gaussdb=# select * from redis_status order by updated;
 status | updated
-----+-----
 REDISTRIBUTION SETUP | 2025-05-08 17:30:37.161
 REDISTRIBUTION STARTED | 2025-05-08 17:30:38.899
 REDISTRIBUTION COMPLETE | 2025-05-08 17:37:31.653
 REDISTRIBUTION SHUTDOWN | 2025-05-08 17:37:36.485
(4 rows)
```

也可以使用集群用户(omm)执行以下命令查看重分布进度。

```
可以使用以下命令查询扩容重分布处于什么阶段
gs_expand -t progress -X /opt/software/gaussdb/clusterconfig_dilatation.xml
重分布前置检查阶段
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Begin redistribute

重分布准备阶段
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Prepare redistribute

重分布期间
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Do redistribute

重分布后处理阶段
Querying current dilatation progress.
Current expansion progress is:
Status: Running, Detail: Post redistribute
```

----结束

## 7.26.4 常见问题

Q: 当在线扩容中遇到报错：“visibility\_heap\_tuple\_satisfies\_mvcc snapshot csn <number> less or equal to bucketcsn <number> from old DN.”。

A: 扩容前长事务快照在读取新节点迁移后上线数据时，可能触发该报错。此现象为预期内的正常机制，不会对业务运行造成影响。

Q: 加节点过程中遇到报错：“Failed to lock cluster.”。

A: 需检查集群中是否存在未终止的长事务，此类情况会导致加节点流程无法获取集群锁。



## 7.27 备机读

### 7.27.1 特性概述

#### 功能定义

开启备机读功能后，备机将支持只读事务并满足数据一致性要求，有效缓解主机负载压力。

#### 用户场景

系统需处理高并发访问请求，同时要求高可靠性与可用性，通常部署多个副本。通过设置数据库备机为可读模式，可以将部分查询请求分散到备机上执行，降低主数据库负载，充分利用备机硬件资源，减少硬件投入成本。

#### 快速启动

- 配置备机读功能需启用参数：hot\_standby=on；GTM\_LITE模式需激活：auto\_csn\_barrier=on。

- 通过gsql方式连接数据库：

```
gsql -d {DB_NAME} -p {DB_PORT} -h {DB_HOST} ...
```

- DB\_NAME：数据库名。
- DB\_PORT：数据库CN节点port。
- DB\_HOST：数据库CN节点host。

在单集群模式及容灾部署主集群CN上，建立连接后执行以下会话配置开启会话级备机读。

```
SET enable_standby_read=on;
```

容灾部署、灾备集群环境中禁止执行该配置，会话建立后即默认开启容灾读功能。

- 通过JDBC方式连接数据库：

单集群配置：

```
jdbc:gaussdb://node1:port,node2:port,node3:port/database?enableStandbyRead=true
```

enableStandbyRead：连接CN后启用备机读操作，需配合[参数配置](#)生效。

容灾读场景（灾备集群）配置：

```
jdbc:gaussdb://node1:port,node2:port,node3:port/database?
isasterToleranceCluster=true&autoBalance=true
```

- disasterToleranceCluster：启用后自动判断目标集群是主集群还是灾备集群；配置IP需指向目标集群CN的IP。
- autoBalance：表示开启负载均衡。
  - true/balance/roundrobin：表示开启JDBC负载均衡功能，将应用程序的多个连接均衡分配至数据库集群中的各个可用CN。
  - priorityn：表示开启JDBC优先级负载均衡功能，将应用程序的多个连接均衡到URL中前n个可用的CN数据库节点。
  - shuffle：表示开启JDBC随机负载均衡功能，将应用程序的多个连接随机均衡到数据库集群中的各个可用CN。

- specified: 表示开启JDBC负载均衡功能，并且只在用户URL中指定的节点上建立连接。
- shufflen: 表示开启JDBC在指定节点上优先随机负载均衡功能，将应用程序的多个连接优先随机分配至URL中前n个可用的CN数据库节点。当URL上配置前n个节点全部不可用时，连接会随机分配至其他可用CN节点。
- false: 表示不开启JDBC负载均衡功能和优先级负载均衡功能。

#### 说明

灾备集群启用分布式容灾读模式时，不可配置enableStandbyRead=on，否则请求将被拦截。

## 规格约束

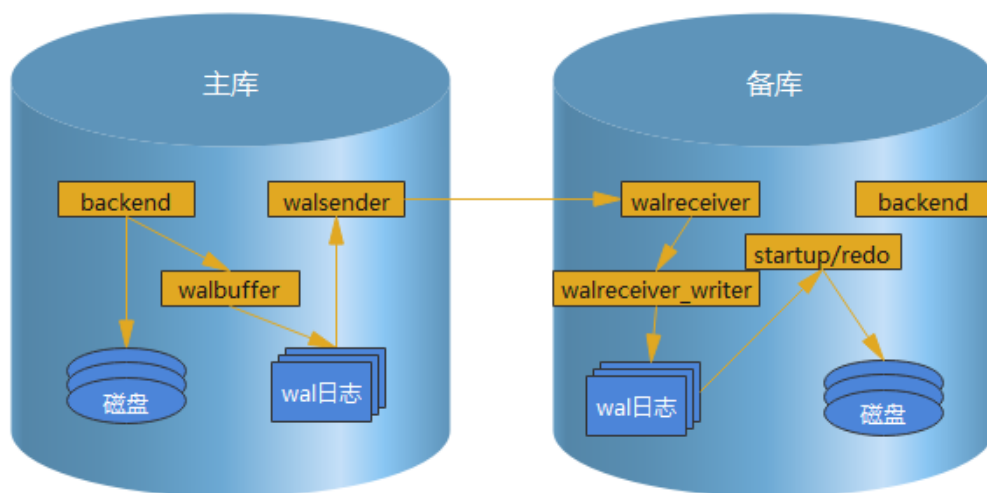
- 备机读性能与硬件规格、I/O负载及实际业务高度相关，不同的业务场景下备机读性能差异较大。
- 典型商用场景（大表300GB + 主机带100并发UPDATE + VACUUM，主备200并发读且I/O、CPU资源未出现瓶颈）：
  - 串行回放模式下备机读性能达主机90%。
  - 并行回放模式下备机读性能在触发VACUUM时维持45% ~ 90%区间。
  - 极致RTO模式下备机读性能较串行模式最多劣化10%（约主机80%）。
- 只读操作不阻塞DML日志回放，主备时延可控制在1秒内。
- 只读会阻塞DDL等修改系统表操作的日志回放，冲突发生时达到max\_standby\_streaming\_delay阈值将终止查询。
- 打开极致RTO只读场景时，只读操作阻塞部分DDL回放但不影响DML；需额外分配内存/磁盘资源缓存历史数据，查询时长受存储空间限制。
- 主备切换、备机异常重启或备机加回/加副本后，需回放至上次的读位置，方可支持备机读。
- enable\_standby\_read参数不支持在事务块和存储过程中启用。
- 分布式备机读不支持自治事务功能。
- 当开启极致RTO后执行删除文件的DDL操作时，备机会延迟删除相关文件，从而导致备机数据目录下的文件数目多于主机。此时若查询主备机数据库大小，将会显示备机数据库大于主机数据库。
- 开启极致RTO备机读后，仅支持可重复读隔离级别，设置其他隔离级别不会报错但不生效。
- 备机读不支持临时表；由于临时表不记录Xlog日志导致数据无法同步至备机，备机读不支持临时表的读操作。

## 7.27.2 技术原理

### 7.27.2.1 三种回放模式及备机读

Redo Log（本章节统一称Xlog）是用于恢复已确认但未写入数据库的数据，记录数据修改后的值。GaussDB备机通过同步Xlog并进行回放实现主备机数据同步。流式同步原理如图7-49所示。

图 7-49 流式同步原理



1. 主机会话线程在数据更新时优先确保Xlog写入磁盘。
2. walsender线程持续将已落盘的本地Xlog传输至备机的walreceiver进程。
3. 备机walwriter进程将接收的Xlog写入磁盘，并向主机反馈接收位置、刷盘位置及当前回放位置，主机根据此信息完成多数派同步决策。
4. 备机回放线程读取本地Xlog并执行回放操作，完成实际数据同步。

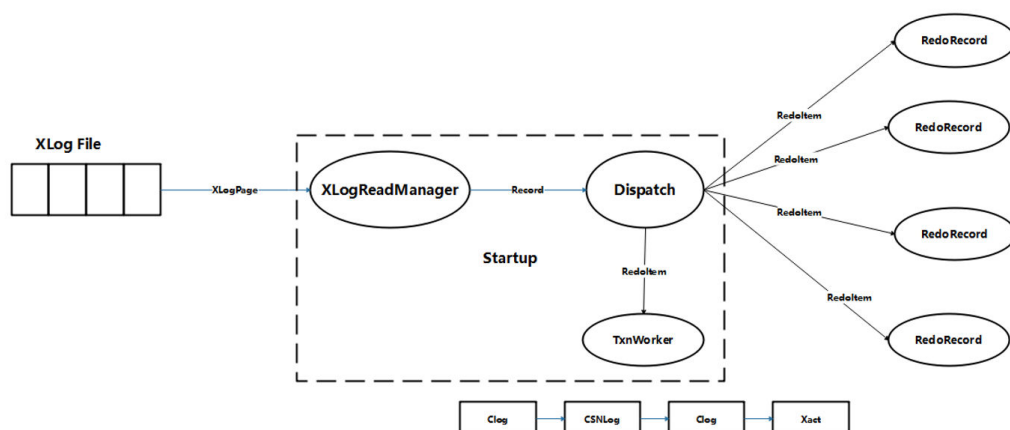
## 串行回放

- 按主机记录的顺序依次回放Xlog。
- 单个CN/DN实例串行回放备机读：  
备机读线程在首次打开relation时，会获取当前备机回放的最新Global Xmin作为自身snapshot xmin，并以此快照执行读操作。  
事务状态由csn committing、csn committed两条Xlog组成；当备机回放完csn committed日志后，该事务可见。
- Page clean up ( VACUUM ) 日志冲突：  
当备机回放VACUUM日志时，若与备机snapshot中xmin较小的读事务冲突，此类读会话将在超过max\_standby\_streaming\_delay阈值后被强制终止。

## 并行回放

利用现代CPU多核处理能力，针对回放日志流程中耗时最长的“回放日志流程”分配多个线程，将回放日志任务按照relation粒度分发到多个线程同时处理，显著提升并行度。并行回放日志分发模型如图7-50所示。

图 7-50 并行回放日志分发模型



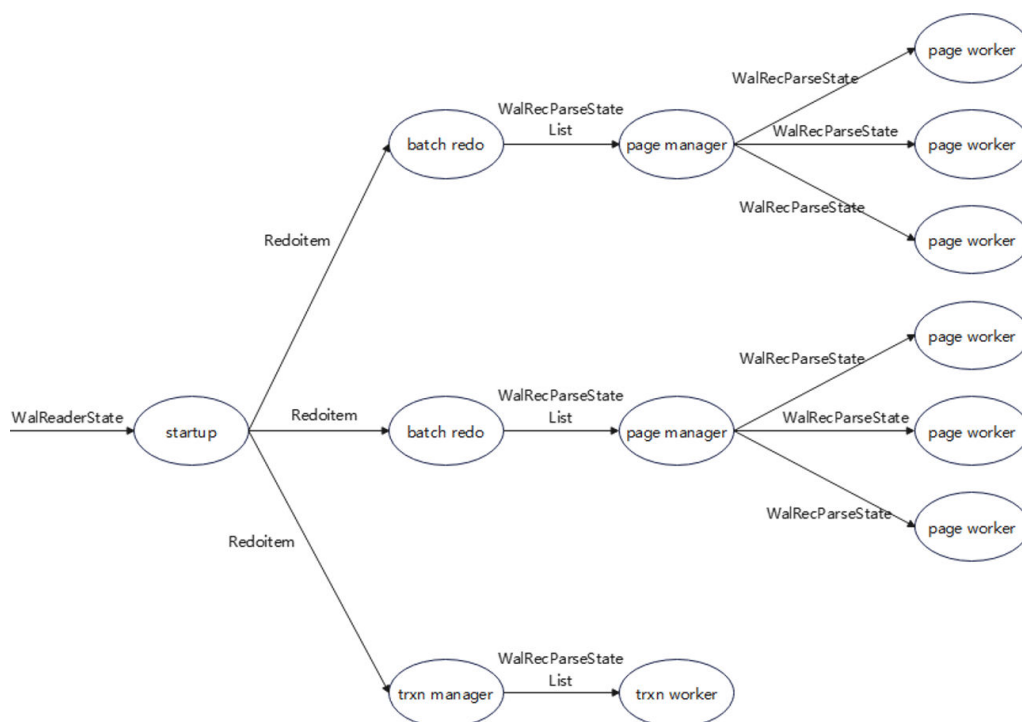
- 备机通过walreader顺序读取Xlog，根据relation id分发至不同redo worker线程并行处理。
- 事务提交的Xlog由独立的事务回放线程进行回放。
- 单个CN/DN实例上的并行回放备机读：  
查询与回放冲突的解决机制与串行回放模式一致。
- 由于并行回放采用文件级并行机制，某一时刻可能出现索引与表回放速度差异导致的两种异常场景：
  - a. 索引回放比表快，索引Tuple指向的Heap Tuple尚未写入Heap。
  - b. 索引回放比表慢，指向Heap的索引Tuple尚未写入索引。

在情况1下，索引回放比表快，索引扫描相关操作符先访问索引，再访问表，会出现tid不存在或者页面不存在报错。

## 极致 RTO 回放

回放优化基于现代CPU多核处理能力，将回放流程拆解为多级流水线架构，包含独立的事务回放pipeline及多个数据回放pipeline，实现按页面粒度分发Xlog执行回放。极致RTO回放原理如图7-51所示。

图 7-51 极致 RTO 回放原理

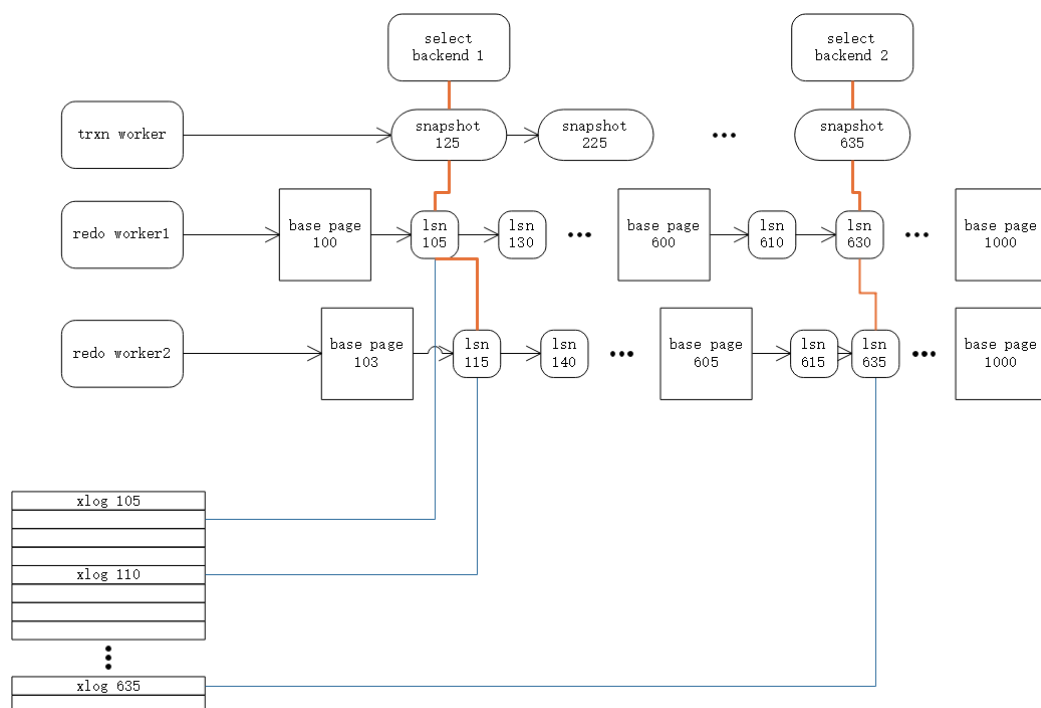


相较于并行回放模式，极致RTO回放通过进一步提升并行度并构建多级流水线架构，每条流水线由page manager将Xlog分发至多个page worker并行处理。

单节点CN/DN极致RTO备机读机制：

引入多版本读机制：当页面回放导致历史版本数据被覆盖时，日志系统将记录历史版本页面的base page；其余Xlog以LSN链形式持久化存储。读操作根据快照LSN判断应读取最新页面或历史版本页面。极致RTO备机读原理如图7-52所示。

图 7-52 极致 RTO 备机读原理



读线程select backend1执行涉及多个页面的读操作时，在打开relation阶段会获取当前极致RTO快照（如快照125），提取该relation1的base page 100，回放LSN链上的Xlog直至达到读快照LSN阈值，随后依次获取relation2等其他数据页的base page并回放对应LSN链上的Xlog，最终整合所有页面数据完成历史版本读操作。

针对引发物理数据页删除的部分DDL操作，如VACUUM、DROP TABLE、TRUNCATE TABLE、DROP DATABASE，增加延迟删除机制，同时通过系统表MVCC机制与syscache的invalid msg机制协同实现系统表的多版本处理，有效解决查询与回放此类DDL操作的冲突。

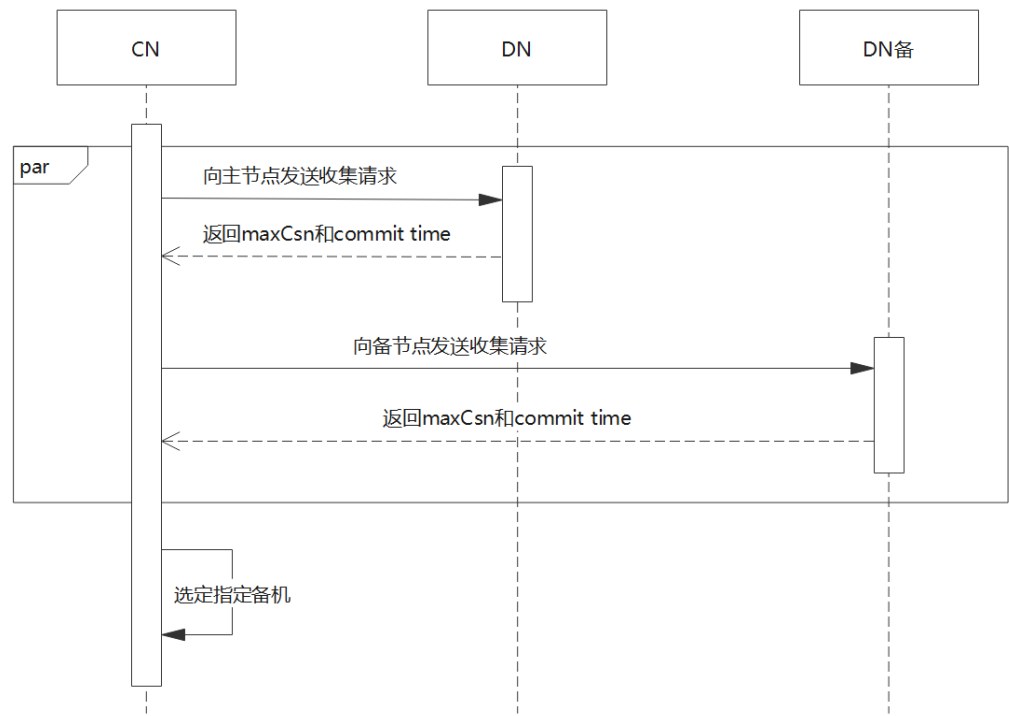
### 7.27.2.2 分布式 GTM-FREE 备机读

分布式GTM-FREE模式通过备机提供弱一致性读能力，有效降低主机负载。

在分布式场景下，CN内置的收集线程持续向主节点（DN）及备节点（DN）采集CSN与提交时间（commit time），用于校验备机节点的CSN值与事务延迟是否满足预设阈值，从而筛选出符合条件的备DN将部分读请求下发执行。

CN的收集线程定期汇总各备机的maxcsn值与commit time，并记录至CN缓存中，流程如图7-53所示：

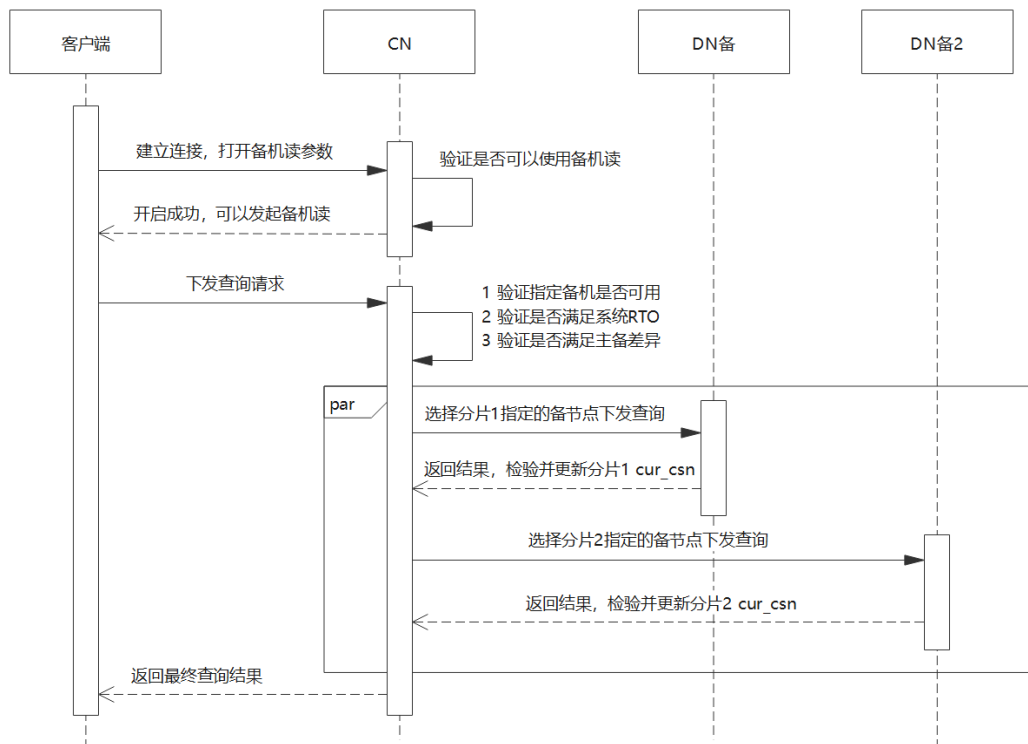
图 7-53 CN 收集线程



当执行查询时，基于采集的DN最新commit time数据，需同时满足standby\_read\_delay与standby\_read\_rto参数约束，针对每个分片独立筛选可用备机。查询请求将下发至选定备节点，由备机基于本地快照执行查询操作。若所有分片均无可用备机，则直接在CN层抛出异常并终止请求。

GTM-FREE备机读流程如图7-54所示。

图 7-54 GTM-FREE 备机读流程



说明

GTM-FREE备机读机制仅校验各分片备机相对于分片主机的延迟指标，不保证各个分片均使用同一全局事务号进行读操作，因此无法实现瞬时一致性。该方案提供最终一致性保障，即当查询涉及的所有表在所有分片完成日志回放后，备机查询结果将与主机读操作保持完全一致。

特性约束

- 1. 读到的数据与主机存在一定延时，主备差异超过指定数值（由 standby\_read\_delay指定）时，会报错。
- 2. 提供弱一致读，仅保证session内部的快照递增序，session之间不保证。
- 3. 不支持stream计划，开启enable\_stream\_operator参数后，备机读会采用pgxc计划替代stream计划。
- 4. 不支持备机间的负载均衡。
- 5. 主机性能和RTO优先，备机读业务其次。在没有可用备机时均会报错，备机读的查询不会发送到主机上。
- 6. 备机读业务会影响备机RTO，若所有备机的RTO都超过指定阈值（通过 standby\_read\_rto指定），备机读业务报错，业务侧通过查看错误码进行熔断。
- 7. 允许GTM-Free模式下gsql直接连接备机。

7.27.2.3 分布式 GTM-LITE 备机读

提供强一致性读的能力。

从[三种回放模式及备机读](#)可以了解到串并行模式下，GaussDB通过对比tuple和Snapshot中的xmin信息判断tuple的可见性。如果tuple来自CSN小于Snapshot中CSN的事务，则该tuple对当前查询可见。当查询语句在集群的CN节点获取快照后，会将一

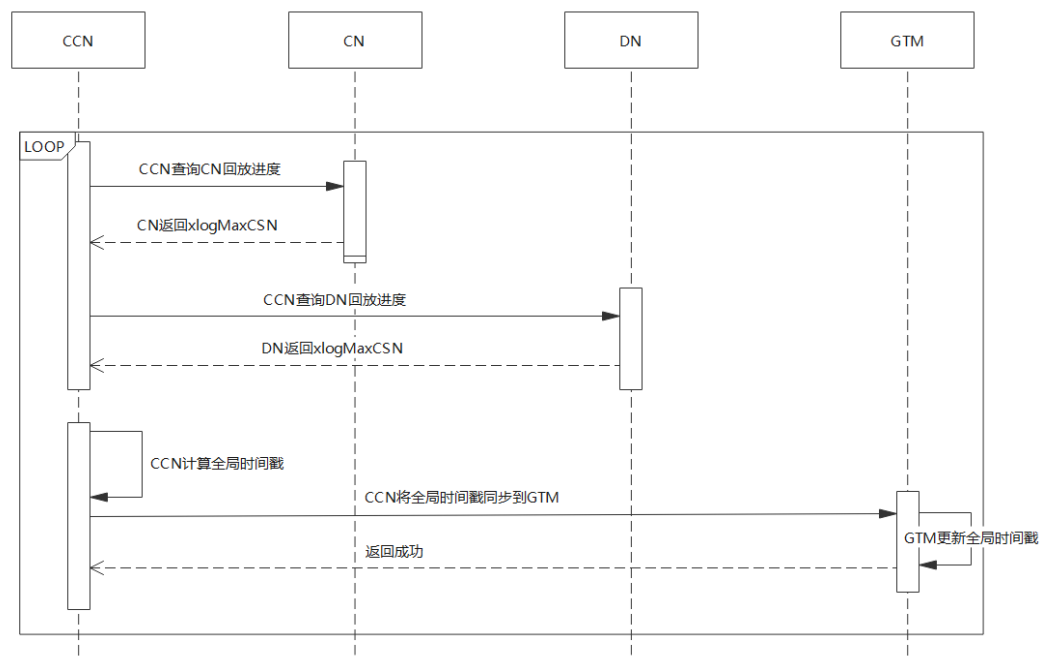


致性点下发给DN节点。相关的DN使用CN下发的快照中ConsistencyPointCSN判断tuple可见性。

启用GUC参数auto\_csn\_barrier进行barrier打点，通过CN内置的收集线程持续向备DN收集barrier点的CSN，计算出全局一致性CSN后对备机节点信息进行持久化存储与缓存处理。查询线程基于缓存数据筛选满足一致性点的备DN，将部分读请求下发至目标备节点执行。备DN通过读取备机一致性点的CSN执行查询操作，最终将查询结果回传至CN。

CN开启CSN打点并记录到GTM的流程如图7-55所示。

图 7-55 CN 打点记录 CSN



1. 启用GUC参数“auto\_csn\_barrier=on”开启barrier打点，生成barrier日志。各备节点回放时更新XlogMaxCSN。
2. CN节点后台线程standby\_read\_main（复用GTM\_FREE后台线程）定时维护全局一致性点。该线程周期性向所有备DN发送消息查询日志回放进度。
3. 备DN收到来自CN的查询请求后返回各自维护的barrier日志回放的最大CSN，即XlogMaxCSN。
4. CN节点计算各分片XlogMaxCSN中位数，取所有分片中位数最小值作为一致性点CSN，并持久化到GTM。

该机制确保所有CSN小于一致性点的事务日志已在多数节点完成回放。完成收集后CN将一致性点CSN同步至GTM供查询使用。

极致 RTO 场景

在极致RTO备机读机制中，快照新增read\_lsn字段用于获取指定LSN版本的历史页数据。DN节点需维护CSN与LSN的映射关系，并记录xmin/xmax信息。当DN收到CN下发的全局一致性快照时，根据CSN查找对应LSN和xmin、xmax信息，构建完整快照。csn-lsn info结构如下图所示。

图 7-56 csn-lsn info 结构

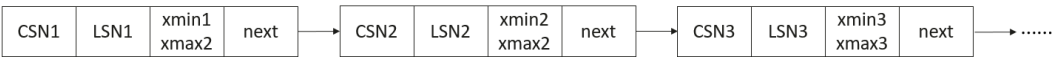
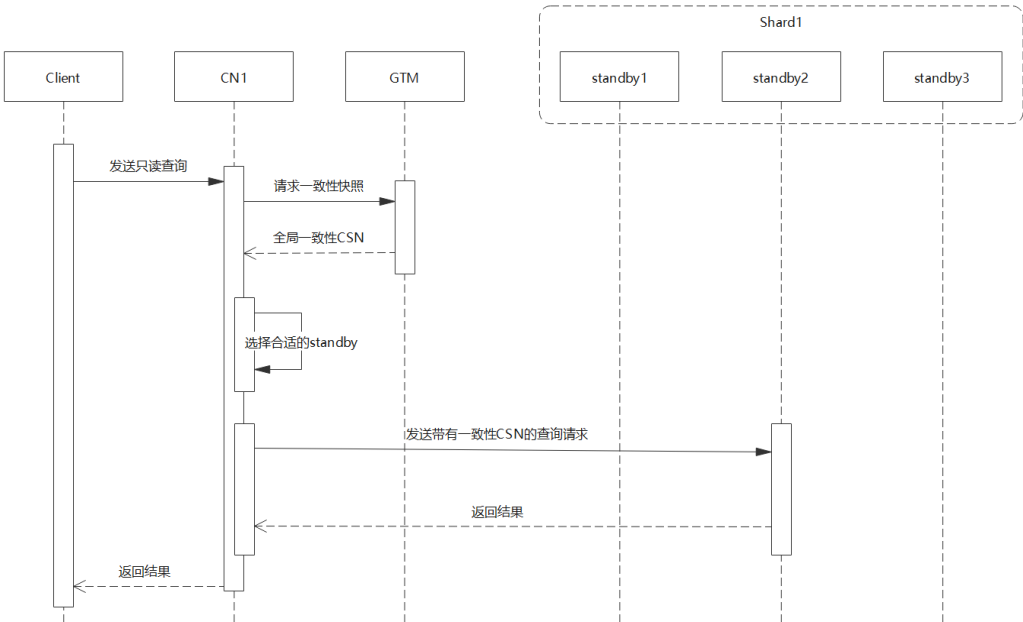


图 7-57 查询整体流程



查询流程：

1. 客户端启用enable\_standby\_read参数后发起查询请求给CN。
2. CN从GTM获取后台线程standby\_read\_main设置的一致性点CSN，注入本地快照。
3. CN根据快照CSN（极致RTO模式额外校验回收位置，即DN上的csn-lsn info链上是否保存对应快照）选择符合条件的备DN。
4. CN下发查询请求至备DN。
5. 备DN收到CN请求后，将接收到的CSN注入本地快照，确保CN与DN使用统一全局CSN进行查询。
6. 备DN执行查询并将结果返回CN
7. CN最终向客户端返回查询结果。

特性约束

- 分布式GTM-Lite支持备机读的规格与约束：
  - 前置条件：需要开启barrier打点功能，通过参数auto\_csn\_barrier控制。
  - 回放冲突类约束
    - 当查询和回放有锁相关的冲突时（主要为表的DDL操作），与串并行回放备机读相同，取消查询由参数max\_standby\_streaming\_delay（参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 备服务器”章节）控制，报错信息是“canceling statement due to conflict with recovery”。

- 查询时间超出了参数standby\_max\_query\_time（参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 备服务器”章节），报错信息是“User query has exceeded the duration threshold.”。
- 触发了备机读文件的强制回收，由参数max\_standby\_base\_page\_size、max\_standby\_lsn\_info\_size和standby\_force\_recycle\_ratio（参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 备服务器”章节）控制。
- 备机回放段页式物理空间收缩操作相关日志。
- 主机执行REINDEX DATABASE时，备机上的查询和relmap类型日志回放有冲突，报错信息是“User query might have needed to see row versions that must be removed”。
- 主机执行DROP DATABASE、DROP TABLESPACE操作时，备机在回放相应的日志时可能会跟查询冲突，报错信息是“canceling statement due to conflict with recovery”。
- 快照类约束
  - 在极致RTO备机读场景下，查询快照信息是由CN发来，可能有滞后，一些数据信息已经更新，在以下几种场景中可能会报错：
    - REINDEX DATABASE等操作，导致relmap变更，报错“snapshot lsn is smaller than lastReplayedConflictLsn due to relmap redo”。
    - 节点启动，报错“snapshot lsn is smaller than lastReplayedConflictLsn due to starting action”。
    - 备机读文件强制回收，报错“snapshot lsn is smaller than lastReplayedConflictLsn due to force recycle”。
  - 在极致RTO备机读场景下，集群启停或故障、极致RTO模式下由于某种原因（磁盘空间不足、修改GUC参数、某些节点回放没有跟上等）触发了强制回收，均可能导致备机读报错结束查询（ERROR: snapshot csn is not in csn-lsn list, DETAIL: snapshot csn is 123456789）。
  - 在串行/并行回放备机读场景下，主机频繁VACUUM会导致备机读查询的数据被提前清理掉等问题，可能出现查询和回放冲突，报错并结束查询（ERROR: gtm csn small: gtm csn 123456789, lastReplayedConflictCSN 123456999）。
  - 在对表进行DDL操作时，CN上的系统表已经被修改，但是备DN可能还未回放这些DDL日志，当使用老快照查询该表时，CN和备DN的元数据可能会不一致。CN在检测到该场景后，会结束访问该表的查询并报错，需要业务进行重试（ERROR: current snapshot is invalid for this relation/partition）。
  - 在CN与DN连接时，需根据当前全局一致性点计算对应分片中可用的DN副本，以下场景可能出现报错：

```
LOG: Current slice cannot select a normal standby for standby read, slice: <int> ...
ERROR: pooler_get_dn_handles invalid Datanode number: -1, dnNum: <int>, u_sess->pgxc_ext.NumDataNodes[<int>]
```

    - 场景一：同一数据副本在两个DN上的回放速度不一致，且回放较快的DN发生重启。

处理方法：全局一致性点将停止推进，需等待慢速分片追赶至全局一致性点，报错将自动消除。

- 场景二：容灾映射刚完成构建，但尚未完成CN/DN回放与回收信息采集。

处理方法：该报错仅在启动初期短暂出现，等待数秒后不再触发。

- 场景三：在极致RTO模式下，不同分片的回放进度差异显著，导致回放较快的分片其回收点超过了全局一致性点。

处理方法：等待业务高峰过后，各分片回放差异减小，报错消失；如果仅个别CN回放较慢，可通过执行cm\_ctl stop命令手动停止该CN以规避问题。

#### - 资源管控

- 磁盘空间：当前极致RTO备机读文件的空间占用已有阈值保护，参考max\_standby\_base\_page\_size、max\_standby\_lsn\_info\_size、standby\_force\_recycle\_ratio（参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 备服务器”章节）。
- 内存和IO：需求支持备机读独立BufferPool和独立刷脏，参考enable\_standby\_bufferpool（参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 备服务器”章节）。
- CPU：当前无资源管控能力，在当节点CPU资源充足的场景下（不超过70%），备机读和回放没有影响。
- 对于小规格（8U以下）场景，极致RTO回放会占用较多资源，不建议打开极致RTO备机读。

#### - 性能规格

- 打开选AZ（standby\_read\_use\_az\_info）或者负载均衡功能（standby\_read\_use\_load\_balance），由于需要遍历所有可用备DN节点，再去从中做选择，所以会导致查询性能下降。
  - 当主机频繁执行DDL操作时，除了会导致备机上的查询与回放冲突被取消外，还会导致备机上的查询变慢。
  - 在96U768G物理机环境下，3C3D混合部署，独立数据盘，配置出口参数，初始数据100G，主备100并发读，主机100并发update、100并发insert，备机读性能达到主机读性能的80%。
- 主机在进行在线重建索引、删除索引、重建索引、删除分区时，CN下发到备机的查询可能会访问到已经被删除的索引或还未被创建的索引，可能会报以下错误，此时业务需要重试。主要的报错类型包括但不限于以下内容：
- 普通表："Could not open relation with OID xxx."
  - "contains unexpected zero page at block xxx."
  - "index xxx is not a btree."
  - "Tuple has wrong number of attributes in index xxx."
  - "xxx doesn't contain target block."
  - "The target page is not btree."

- "index xxx is not valid on standby."。
- 分区表: "Partition oid xxx is invalid when opening partition current tid."。
- "could not open block during recovery delete object, please try again later."。
- "search sys cache failed when get index list, indexrelid xxx, indrelid xxx, relation xxx"。
- "partition xxx does not exist on relation xxx"。
- 在主机并发执行指定位置增删列语句（如ALTER TABLE ADD FIRST/AFTER等）时，若系统表并发修改，备机查询可能出现列类型不匹配的错误，需业务重试。
- 备机回放系统表时不通过延迟DDL。当备机故障恢复并重新对外提供服务后，CN可能下发旧的快照，导致查询时出现找不到历史版本的报错："standby\_read\_buf can't find history version page"。
- 在分布式场景下，主机执行DDL且备机执行Stream计划的查询时，可能出现CN和备DN看到的表定义不一致，从而导致查询报错。其原因为：分布式备机读执行Stream计划查询时，计划在CN生成，然后下发给备DN执行；而CN作为主机角色，生成Stream计划时使用SnapshotNow访问系统表，备DN执行时则使用SnapshotMvcc读取数据，因此会出现数据与计划中的表定义不一致的问题，引发查询报错。分布式容灾读场景不涉及该问题，因为灾备集群的CN也是备机，使用SnapshotMvcc访问系统表。这类DDL主要包括：增删列、重命名列、扩展列类型varchar大小、修改列类型numeric精度、修改ENUM/SET列定义、重命名表、增删分区、重命名表空间、创建索引CONCURRENTLY、重建索引CONCURRENTLY等。当遇到这类报错时，备机读业务需要重试。主要的报错类型包括但不限于以下内容：
  - attribute xxx has wrong type.
  - invalid attribute number xxx;
  - attribute number xxx exceeds number of columns xxx.
  - pg\_class entry for relid xxx vanished during xxx.
  - cache lookup failed for index xxx.

#### 7.27.2.4 分布式 GTM-LITE 容灾读

分布式GTM-LITE容灾读的实现原理与分布式备机读机制基本一致。

启用auto\_csn\_barrier参数后，系统进行barrier打点。在容灾读场景下，收集线程除持续向备DN节点收集barrier点的CSN外，还会对包括自身的每个CN节点发起请求，将每个CN视为独立分片参与校验。该机制要求至少存在一个CN节点能获取有效CSN值，最终计算出全局一致性的CSN保存至GTM中。

查询执行阶段，当所有备机节点满足一致性点时，系统优先选择级联备DN节点下发请求。级联备DN节点通过获取备机读的一致性点CSN值，基于csn-lsn info中查询对应LSN记录，最终将查询结果返回至CN节点。

分布式读机制依赖pgxc\_node表维护的节点信息进行CN本地通信列表初始化。在容灾场景中，由于灾备集群的CN是主集群CN的备机，CN节点通过回放主集群pgxc\_node

表数据构建的，无法自主更新表中内容。因此主集群的pgxc\_node表需额外记录灾备集群所有节点的静态信息。灾备集群的内部业务在访问系统表pgxc\_node时，将通过GUC参数cluster\_name（记录本集群的集群名）对表中的cluster\_name字段进行过滤，筛选出本集群的节点信息。

## 特性约束

- 容灾机制类约束：
  - 容灾集群不支持COPY TO操作，由于容灾集群的CN是备机，不能分配XID，COPY TO在CN上需要申请XID，故无法执行。执行时报错（ERROR: cannot assign TransactionIds during recovery）。
  - 分布式容灾单副本模式，不支持分布式读操作，可以使用维护模式进行连接执行本地查询。执行分布式查询时会出现报错（ERROR: Currently disaster cluster read is not supported）。
  - 如果主集群与容灾集群之间出现断连，那么从容灾集群上查询到的数据可能是旧数据。
  - GTM-Lite方案，在CN与DN连接时，需要根据当前全局一致性点，计算出对应分片中，可用的DN副本，以下场景可能出现报错：Current slice cannot select a normal standby for standby read, slice: 1
    - 同一副本中，如果出现两个DN回放快慢不一致，且更快的DN出现重启时。处理方法：此时全局一致性点不会再推进，等待回放慢的分片跟上全局一致性点，报错消失
    - 容灾映射刚完成构建，但还未收集到各个CN、DN的回放、回收信息时。处理方法：仅启动时间窗会短暂出现，等待几秒钟后不再出现。
    - 极致RTO模式下，如果出现不同分片回放差异很大，部分回放快的分片回收点超过全局一致性点时。报错后处理方法：等待业务高峰通过，各分片回放差异减小后，报错消失，如果仅有个别CN回放很慢，可以通过cm\_ctl stop命令，手工停止CN规避此问题。
  - 当灾备集群出现CN剔除、CN加回、版本升级至506版本升级提交时，会对依赖pgxc\_node表通信缓存产生影响，此时会触发内部的reload机制，reload期间（时间窗3s左右）内会出现以下已知报错：
    - node id %d is in pgxc\_group but not in the handles, maybe rebuilding DisasterCache. 根因：缓存中pgxc\_group与缓存中通信dn矩阵中记录的OID不匹配。
    - cache lookup failed for node %u. 根因：pgxc\_node表中的oid，与缓存中记录的节点信息oid不匹配。
    - Cannot reload pool when in a transaction block. 根因：会话在事务中，检测发现pgxc\_node更新，但此处事务不支持reload pool，此处会报错，但不会持续报错。
    - pooler: shmemNums does not match with usessNums, pgxc\_node reload failed. 根因：通信缓存agent中记录的备机节点个数是旧的，但会话内存变量记录的备机节点个数，已经随pgxc\_node表的更新而刷新了。
  - 因临时表、临时视图、临时序列和unlogged表，不产生Xlog，也不会同步到备机，所以备机读不支持查询此类对象，查询会报错：

- ERROR: Temporary or unlogged table cannot be accessed on the standby.
- 回放查询冲突类约束：
  - 主集群频繁vacuum、频繁DDL会影响日志回放速度，回放机制本身可能出现查询与回放冲突，回放日志时，如果持有冲突的会话，已经执行超过max\_standby\_streaming\_delay的时间，会主动结束查询，返回错误：ERROR: canceling statement due to conflict with recovery / ERROR: terminating connection due to conflict with recovery。
- 快照类约束：
  - 串并行回放容灾读：
    - 主集群做DDL操作，备集群回放DDL日志的时候，DDL在备集群上的某些节点中可能还没回放成功，校验快照失败，报错并结束查询（ERROR:current snapshot is invalid for this partition/relation）。
    - 主集群频繁vacuum会导致灾备集群查询的数据被提前清理掉等问题，读会话会检查快照是否大于DN最新的回放查询冲突点，不满足则说明此快照已经太旧了，不能读最新数据，主动报错并结束查询（ERROR: gtm csn small: gtm csn 123456789, lastReplayedConflictCSN 123456999）。

极致RTO容灾读：

- 由于磁盘空间不足、修改GUC参数触发了强制回收，或发生过启停，或某一分片回放慢导致全局一致性点长时间推进缓慢，均会导致正常回放的CN/DN，由于本地历史版本不支持当前一致性点而不可读，报错结束查询（ERROR: snapshot csn is not in csn-lsn list）。
- 系统表（relfilenode<16384）不会处理延迟DDL。在涉及系统表的TRUNCATE或DROP操作时，相关文件将被直接删除，包括极致RTO历史版本。在此类场景下，备机读操作可能因无法定位历史版本而报错：“ERROR: standby\_read\_buf can't find history version page/ERROR: block old version can not found”。
- 段页式表，均不会进行延迟DDL处理，所以在段页式表触发TRUNCATE、DROP、REINDEX类的DDL操作时，备机回放会直接删除原数据文件，导致读段页式表0号页面报错。
- 性能约束
  - 当主备集群查询所能占用到的资源相同时，对于并行回放下的Astore表：
    - 在主集群无写业务、备集群的日志都回放完成的场景下，对于select count(\*) from xxx语句的负载，从时延的维度上观察，备集群的查询性能达到主集群的80%。
    - 在带有写业务的场景下：主集群和容灾集群的查询性能可能都会受业务影响而有波动，单次查询不保证容灾集群的性能达到主集群的80%；在无DDL的场景下，对于sysbench和TPCC的负载，从TPS、第95百分位数和平均时延的维度上观察，容灾集群的性能达到主集群的80%。
    - 如果有多个查询的主节点或备节点部署在同一个机器上，因这个机器的资源不足导致查询性能下降的情况下，则上面两个场景不成立。
    - 当发现前两个场景不成立时，建议检查容灾集群中各节点是否存在资源（CPU、内存、磁盘）受限的情况。

- 由于容灾集群在并行回放下的Astore表不支持index only scan查询，因而对于select count(\*) from xxx语句的负载可能需要读取更多的数据页面，建议通过调整shared\_buffers参数进行缓存优化。
- 兼容性场景约束

当升级版本升级至506版本时，升级观察期期间，会使用旧版本映射方案处理容灾读通信列表，此时有相关约束：

  - 容灾集群上pgxc\_node表的内容是同步自主集群的，所以容灾集群的CN会根据pgxc\_node表中的node\_oid与真实的节点信息做一层映射，但主备集群的node\_name不是完全对应的，这会导致通过EXECUTEDIRECT ON语法实现的视图在容灾集群上的查询结果出现异常（结果重复或缺失），建议直接在对应的节点上查询。
  - 主备集群异构、CN数量不一致时，进行灾备集群读之前，应先查询出与主集群存在映射关系的灾备集群CN，再根据查询出的CN进行相关业务操作（查询方式：SELECT \* FROM pgxc\_disaster\_read\_status()）。如果连接到和主集群不存在映射关系的CN，由于此CN自身的一致性点不参与全局一致性点的计算，可能会出现连接报错（ERROR: cn data invisible: local csn 123456789, gtm snapshotcsn 123456999）。
  - 如果使用JDBC连接容灾集群，且开启了负载均衡功能，那么查询请求会发送到主集群上，导致查询性能受到影响。
  - 重启cn容灾搭建完成后，因容灾读本集群拓扑节点信息，依赖pgxc\_node表回放后内部映射机制的构建。在容灾映射构建完成前，读请求会出现报错：Not support disaster read before init. 容灾集群关系搭建完成，灾备集群正常启动后，10s左右（主集群、灾备集群之间无RTO差距）此读报错消失。
  - 容灾集群由于pgxc\_node表，是通过回放记录主集群的拓扑信息，如果出现，1）双集群集群间倒换，2）1：N主备集群部署场景下，主集群倒换，3）主集群节点替换、CN剔除等，主集群拓扑结构变更导致灾备pgxc\_node表被动更新时，由于分布式查询CN中使用的部分系统缓存，通过pgxc\_node表中的记录oid作为key，当线程读到新的pgxc\_node表信息，但系统缓存还未来得及更新时，此时间窗（3s左右）内会出现以下已知报错：
    - node id %d is in pgxc\_group but not in the handles, maybe rebuilding DisasterCache. 根因：缓存中pgxc\_group与缓存中通信dn矩阵中记录的OID不匹配。
    - cache lookup failed for node %u. 根因：pgxc\_node表中的oid，与缓存中记录的节点信息oid不匹配。
    - Cannot reload pool when in a transaction block. 根因：会话在事务中，检测发现pgxc\_node更新，但此处事务不支持reload pool，此处会报错，但不会持续报错。
    - pooler: shmemNums does not match with usessNums, pgxc\_node reload failed. 根因：通信缓存agent中记录的备机节点个数是旧的，但会话内存变量记录的备机节点个数，已经随pgxc\_node表的更新而刷新了。
- GTM-Lite备机读类约束

GTM-Lite容灾读与备机读使用相同的机制处理读逻辑，也会继承 [分布式GTM-Lite支持备机读的规格与约束](#)。

## 7.27.3 操作指导



7.27.3.1 环境规格与回放类型选择

回放模式选择

当 recovery\_parse\_workers = 1、recovery\_redo\_workers = 1、recovery\_max\_workers = 1 时，回放模式为串行回放。

当 recovery\_parse\_workers = 1、recovery\_redo\_workers = 1、recovery\_max\_workers > 1 时，回放模式为并行回放。

当 recovery\_parse\_workers > 1 或 recovery\_redo\_workers > 1 时，回放模式为极致 RTO，此时recovery\_max\_workers参数无效。

当前备机读功能默认开启（hot\_standby=on），由于极致RTO备机可读功能需通过磁盘空间保存历史版本数据，会额外占用磁盘资源。实际占用空间与业务类型强相关，在磁盘空间受限的小规格场景下，需结合实际业务模型及批处理波峰特征提前测试验证，确认是否适用极致RTO回放模式。

在RTO高敏感场景且无备机读需求时，建议关闭备机读功能，以释放更多CPU及I/O资源用于备机回放。

根据硬件规格与回放模式推荐：

表 7-38 不同 CPU、内存和部署模式下的参数设置参考

| 编号 | CP<br>U个<br>数 | 内<br>存<br>( G<br>B) | 是否<br>混合<br>部署 | recovery<br>_parse_w<br>orkers | recovery_redo_<br>worker<br>s | 回放线程总<br>数                      | 备注                           |
|----|---------------|---------------------|----------------|--------------------------------|-------------------------------|---------------------------------|------------------------------|
| 1  | 4             | -                   | -              | 1                              | 1                             | recovery_m<br>ax_workers<br>参数值 | 不推荐开极致<br>RTO。               |
| 2  | 8             | -                   | 是              | 1                              | 1                             | recovery_m<br>ax_workers<br>参数值 | 不推荐开极致<br>RTO。               |
| 3  | 8             | 64                  | 否              | 1                              | 3                             | 10                              | -                            |
| 4  | 16            | 128                 | 是              | 1                              | 2                             | 9                               | -                            |
| 5  | 16            | 128                 | 否              | 2                              | 3                             | 15                              | -                            |
| 6  | 32            | 256                 | 是              | 2                              | 2                             | 13                              | -                            |
| 7  | 32            | 256                 | 否              | 2                              | 8                             | 25                              | -                            |
| 8  | 64            | 512                 | 是              | 2                              | 4                             | 17                              | -                            |
| 9  | 64            | 512                 | 否              | 2                              | 8                             | 25                              | 大于此硬件规格<br>的系统均参考此参<br>数设置值。 |
| 10 | 96            | 768                 | -              | 2                              | 8                             | 25                              | 大于此硬件规格<br>的系统均参考此参<br>数设置值。 |

16U 典型场景

- 场景描述：
- 1. 服务器硬件规格：  
CPU：2.60GHz 16U；内存：4GB\*16；磁盘：SSD。
  - 2. sysbench insert+update（1:1），并行度200，普通表5个。
  - 3. 并行回放GUC参数：recovery\_parse\_workers=1，recovery\_redo\_workers=1，recovery\_max\_workers=4。
  - 4. 极致RTO GUC参数：recovery\_parse\_workers=2，recovery\_redo\_workers=2，recovery\_max\_workers=4。

测试场景：

sysbench insert：update=1:1，单表大小固定20GB；对比不同表数量（1张表、5张表、10张表、20张表）下，极致RTO回放与并行回放的回放速度差异。

| 表数量  | 并行回放（MB） | 极致RTO（MB） | 差异     |
|------|----------|-----------|--------|
| 1张表  | 57       | 57        | 0.00%  |
| 5张表  | 60       | 64        | 6.67%  |
| 10张表 | 47.5     | 48        | 1.05%  |
| 20张表 | 25       | 30        | 20.00% |

64U 典型场景

- 场景描述：
- 1. 服务器硬件规格：  
CPU：Intel(R) Xeon(R) Gold 6278C 2.60GHz 64U；内存：16GB\*16；磁盘：SSD。
  - 2. sysbench insert+update（1:1），并行度1000，普通表5个。
  - 3. 并行回放GUC参数：recovery\_parse\_workers=1，recovery\_redo\_workers=1，recovery\_max\_workers=8。
  - 4. 极致RTO GUC参数：recovery\_parse\_workers=2，recovery\_redo\_workers=8，recovery\_max\_workers=8。

测试场景：

sysbench insert：update=1:1，引入不同时长的长事务（无长事务、6分钟、10分钟），对比极致RTO和并行回放的回放速度。存在长事务时，极致RTO回放速度较并行回放提升26%以上。

| 长事务  | 并行回放（MB） | 极致RTO（MB） | 差异    |
|------|----------|-----------|-------|
| 无长事务 | 332.514  | 343.773   | 3.39% |

| 长事务     | 并行回放（MB） | 极致RTO（MB） | 差异     |
|---------|----------|-----------|--------|
| 6分钟长事务  | 288.424  | 363.432   | 26.01% |
| 10分钟长事务 | 310.269  | 399.501   | 28.76% |

典型场景回放速率

机器规格：  
CPU：Intel(R) Xeon(R) Platinum 8378A CPU 3.00GHz 64U；内存：16G\*32；磁盘：SSD。  
场景描述：使用业务场景批量INSERT，并行度50，普通表1个。

表 7-39 典型场景回放速率对比

| 场景      | 机器规格        | 回放类型  | 参数配置                                                                                                       | 回放线程数 | Xlog 积压大小 | 主机 Xlog生成速率 | 备机回放速率        | buffer_pool |
|---------|-------------|-------|------------------------------------------------------------------------------------------------------------|-------|-----------|-------------|---------------|-------------|
| 现网环境    | 主备 64U512GB | 并行回放  | recovery_parse_workers=1<br>recovery_redo_workers=1<br>recovery_max_workers=4                              | 4     | 40GB      | 240M/s左右    | 120 M/s左右     | 180GB       |
| 实验室测试环境 | 主备 96U512GB | 并行回放  | recovery_parse_workers=1<br>recovery_redo_workers=1<br>recovery_max_workers=4<br>recovery_time_target=3600 | 4     | 30GB      | 250M/s      | 100 M/s左右     | 180GB       |
| 实验室测试环境 | 主备 96U512GB | 极致RTO | recovery_parse_workers=2<br>recovery_redo_workers=4<br>recovery_max_workers=4<br>recovery_time_target=3600 | 17    | 0         | 250M/s      | 200-250 M/s左右 | 180GB       |

### 7.27.3.2 操作步骤

#### 步骤1 内核相关参数配置

recovery\_parse\_workers、recovery\_redo\_workers以及recovery\_max\_workers，请参考[环境规格与回放类型选择](#)进行选择。

```
wal_level=hot_standby
```

日志回放级别需大于或等于hot\_standby。

```
hot_standby=on
```

备机读模块开关需设置为on。

```
max_standby_streaming_delay=10s
```

回放查询冲突时，备机在取消查询前会等待的时间；此参数会影响RTO。

**如果环境规格与回放类型选择了极致RTO配置，则需要对以下GUC参数进行配置：**

```
standby_max_query_time=10s
```

回收逻辑计算回收点时，会取消超过standby\_max\_query\_time的查询。建议与max\_standby\_streaming\_delay配置保持一致。

```
max_standby_base_page_size=256GB
```

极致RTO备机读多版本历史空间的阈值，历史页面的base\_page达到阈值的80%时，会触发强制回收并取消当前所有持有快照小于新回收点的查询。

#### 说明

由于极致RTO为解决删除文件类型的回放查询冲突，引入多版本读机制，需持续保存历史版本页面。该空间与Xlog生成速率及实际业务类型强相关，且无法量化，需要以实际业务类型进行验证，才能获得较为可靠的建议占用空间大小。具体场景建议如下：

1. 需保证读操作不报错，但无法避免备机长查询，且业务可接受RTO在部分查询时临时增加，同时磁盘空间受限：建议指定备机使用串行/并行回放备机读。
2. 需保证读操作不报错，无法避免备机长查询，且业务不可接受RTO增加，但磁盘空间充足：参考极致RTO回放查询冲突白名单，约束主集群DDL操作，并根据Xlog生成速率及standby\_read目录空间膨胀比例，配置更大的max\_standby\_base\_page\_size与max\_standby\_lsn\_info\_size（经验值为base\_page大小是lsn\_info的5 ~ 7倍）。
3. 可接受读报错后重试或避免长查询，但对RTO和磁盘空间有严格要求：需基于实际业务Xlog生成速率及standby\_read目录空间膨胀比例，进行针对性配置。

```
max_standby_lsn_info_size=256GB
```

极致RTO备机读多版本历史空间的阈值，历史页面的base\_page达到阈值的80%时，会触发强制回收并取消当前所有持有快照小于新回收点的查询。配置建议参考max\_standby\_base\_page\_size。

#### 步骤2 选择GTM\_FREE/GTM\_LITE模式

- GTM\_FREE模式：

```
enable_standby_read=on
```

SIGHUP级别GUC参数，推荐通过连接串配置或在每个session初始化后执行SQL命令设置：set enable\_standby\_read=on。

```
standby_read_delay=10000000ms
```

SIGHUP级别GUC参数，定义主备最大时延，超过该值时禁止向备机下发请求。

```
standby_read_rto=60000ms
```

设置系统RTO阈值，超过该值时禁止备机读操作。

```
enable_gtm_free=on
```

开启GTM\_FREE模式，POSTMASTER级别参数，需重启生效。

```
gtm_option=2
```

开启GTM\_FREE模式，POSTMASTER级别参数，需重启生效。

- GTM\_LITE模式：

```
enable_standby_read=on
```

SIGHUP级别GUC参数，推荐通过连接串配置或在每个session初始化后执行SQL命令设置：set enable\_standby\_read=on;

```
enable_gtm_free=off
```

关闭GTM\_FREE模式，POSTMASTER级别参数，需重启生效。

```
gtm_option=1
```

开启GTM\_LITE模式，POSTMASTER级别参数，需重启生效。

```
auto_csn_barrier=on
```

开启自动打点功能。

- GTM\_LITE容灾读模式：

```
enable_gtm_free=off
```

关闭GTM\_FREE模式，POSTMASTER级别参数，需重启生效。

```
gtm_option=1
```

开启GTM\_LITE模式，POSTMASTER级别参数，需重启生效。

主集群配置：

```
auto_csn_barrier=on
```

开启自动打点功能。

### 步骤3 JDBC相关参数配置

单集群配置：

```
jdbc:gaussdb://node1:port,node2:port,node3:port/database?enableStandbyRead=true
```

enableStandbyRead：连接CN后启用备机读操作，需配合内核参数配置生效。

容灾读场景（灾备集群）配置：

```
jdbc:gaussdb://node1:port,node2:port,node3:port/database?isasterToleranceCluster=true&autoBalance=true
```

- disasterToleranceCluster：启用后，会自动判断目标集群是主集群还是灾备集群；配置的IP需指向目标集群CN的IP。
- autoBalance：表示开启负载均衡。
  - true、balance或roundrobin：表示开启JDBC负载均衡功能，将应用程序的多个连接均衡分配至数据库集群中的各个可用CN。
  - priorityn：表示开启JDBC优先级负载均衡功能，将应用程序的多个连接均衡到URL中前n个可用的CN数据库节点。
  - shuffle：表示开启JDBC随机负载均衡功能，将应用程序的多个连接随机均衡到数据库集群中的各个可用CN。
  - specified：表示开启JDBC负载均衡功能，并且只在用户URL中指定的节点上建立连接。
  - shufflen：表示开启JDBC在指定节点上优先随机负载均衡功能，将应用程序的多个连接优先随机分配至URL中前n个可用的CN数据库节点。当URL上配置前n个节点全部不可用时，连接会随机分配至其他可用CN节点。
  - false：表示不开启JDBC负载均衡功能和优先级负载均衡功能。

### 📖 说明

灾备集群启用分布式容灾读模式时，不可配置enableStandbyRead=on，否则请求将被拦截。

----结束

## 7.27.3.3 注意事项

- 备机读设计思路是在保证回放RTO正常的前提下，通过提供只读功能分担主机压力并降低成本。其设计与实现过程中，始终优先保障备机回放功能不受影响。可通过配置优化或使用方法提升备机读成功率，但在实际业务场景中，难以保证备机上读操作100%成功，故对读要求极高的业务场景，仍建议部署在主机执行。
- 备机读功能存在长查询保持时间与备机RTO延迟的矛盾。开启极致RTO模式通过增加存储空间（在不影响回放速率的前提下延长查询保持时间），本质上是一种空间换时间的优化策略。建议在使用备机读功能时尽量避免长查询操作。
- 读写分离功能仅支持会话级控制。若在备机读会话（即enable\_standby\_read=on状态）中执行写操作，系统将抛出错误。需注意该功能不支持备机异常时的业务切换至主机。

## 7.27.4 常见问题

### 查询与回放冲突取消查询

- 问题现象：当检测到查询与回放冲突时，回放线程将等待参数max\_standby\_streaming\_delay设定的延迟时间，超时后向冲突线程发送取消指令，导致查询被终止并返回报错信息。

报错详情如下：

User was holding shared buffer pin for too long.

触发场景：备库在回放VACUUM相关类型日志时，若回放线程需获取的buffer因查询线程持有pin锁而等待，且此时存在活跃查询访问该数据页，则触发buffer pin冲突。系统将等待max\_standby\_streaming\_delay参数指定时长后取消冲突查询（通常因冲突查询提前完成而极少触发）。

User was holding a relation lock for too long.

触发场景：DROP TABLE、TRUNCATE或多数ALTER TABLE等操作会获取与SELECT相冲突的锁，若备库在查询活跃的表上回放此类锁操作，将触发锁冲突，等待时长由max\_standby\_streaming\_delay参数控制。

User was or might have been using tablespace that must be dropped.

触发场景：主库删除表空间时，备库的查询恰好在向该表空间写临时文件，则发生冲突。

User query might have needed to see row versions that must be removed.

触发场景：参数max\_standby\_streaming\_delay决定查询与回放冲突的处理策略。备机暂停日志回放直至达到参数指定时长上限，若冲突查询仍持续运行，则终止查询并抛出错误。

User transaction caused buffer deadlock with recovery.

触发场景：回放线程与备机读线程形成死锁时，系统在出现冲突未达到max\_standby\_streaming\_delay阈值前，每5秒执行一次死锁检测。

User was connected to a database that must be dropped.

触发场景：主库执行DROP DATABASE删除数据库时，若备库存在活动会话连接至该数据库，则产生冲突。

- 原因分析：在查询与回放冲突场景下，系统优先保障回放功能正常执行，会取消查询。可能引发冲突的回放操作类型包括：VACUUM、执行8级锁的DDL/DCL语句、DROP DATABASE、DROP TABLESPACE。

### 📖 说明

极致RTO方案已针对VACUUM、DROP TABLE、TRUNCATE TABLE（仅TABLE）、DROP DATABASE解除冲突限制。

- 在极致RTO回放方案下，主机执行常用DDL时，备机回放对应DDL过程中可能出现的阻塞情况：

表 7-40

| 主节点操作                                            |                    | 验证结果<br>(当前用户、表、库) | 操作语句                                                                                                                      | 验证结果<br>(非当前用户、表、库) | 操作语句                                                                                                                      |
|--------------------------------------------------|--------------------|--------------------|---------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------------------------------------------------------------------------------------------------------------|
| add/<br>drop/<br>split/<br>exchange<br>partition | add partition      | 不阻塞                | ALTER TABLE test_p1 ADD PARTITION p4 VALUES LESS THAN (50);                                                               | 不阻塞                 | ALTER TABLE test_p2 ADD PARTITION p4 VALUES LESS THAN (50);                                                               |
|                                                  | drop partition     | 阻塞                 | ALTER TABLE test_p1 DROP PARTITION p3;                                                                                    | 不阻塞                 | ALTER TABLE test_p2 DROP PARTITION p3;                                                                                    |
|                                                  | split partition    | 阻塞                 | ALTER TABLE test_p1 SPLIT PARTITION p4 INTO (PARTITION p4_1 VALUES LESS THAN (40), PARTITION p4_2 VALUES LESS THAN (50)); | 不阻塞                 | ALTER TABLE test_p2 SPLIT PARTITION p4 INTO (PARTITION p4_1 VALUES LESS THAN (40), PARTITION p4_2 VALUES LESS THAN (50)); |
|                                                  | exchange partition | 阻塞                 | CREATE TABLE test_ep1(col1 INT,col2 INT);                                                                                 | 不阻塞                 | CREATE TABLE test_ep2(col1 INT,col2 INT);                                                                                 |
|                                                  |                    |                    | INSERT INTO test_ep1 VALUES (GENERATE_SERIES(1,5), 1000);                                                                 |                     | INSERT INTO test_ep2 VALUES (GENERATE_SERIES(1,5), 1000);                                                                 |
|                                                  |                    |                    | ALTER TABLE test_p1 EXCHANGE PARTITION (p1) WITH TABLE test_ep1 VERBOSE;                                                  |                     | ALTER TABLE test_p2 EXCHANGE PARTITION (p1) WITH TABLE test_ep2 VERBOSE;                                                  |
| create/<br>alter/<br>drop<br>role/<br>user       | create role        | 不阻塞                | CREATE ROLE tpcc1 WITH LOGIN PASSWORD '*****';                                                                            | 不阻塞                 | CREATE ROLE tpcc2 WITH LOGIN PASSWORD '*****';                                                                            |
|                                                  | alter role         | 不阻塞                | ALTER ROLE tpcc1 PASSWORD '*****';                                                                                        | 不阻塞                 | ALTER ROLE tpcc2 PASSWORD '*****';                                                                                        |

| 主节点操作                                                                   |                    | 验证结果（当前用户、表、库）          | 操作语句                                                          | 验证结果（非当前用户、表、库）          | 操作语句                                                                   |
|-------------------------------------------------------------------------|--------------------|-------------------------|---------------------------------------------------------------|--------------------------|------------------------------------------------------------------------|
|                                                                         | drop role          | 阻塞                      | DROP ROLE tpcc1;                                              | 不阻塞                      | DROP ROLE tpcc2;                                                       |
|                                                                         |                    |                         | CREATE ROLE admin WITH LOGIN PASSWORD '*****',                |                          |                                                                        |
|                                                                         |                    |                         | ALTER SCHEMA tpcc1 OWNER TO admin;                            |                          |                                                                        |
|                                                                         |                    |                         | ALTER TABLE t1 OWNER TO admin;                                |                          |                                                                        |
|                                                                         |                    |                         | REVOKE ALL PRIVILEGES ON SCHEMA public FROM tpcc1;            |                          |                                                                        |
|                                                                         |                    |                         | DROP ROLE tpcc1;                                              |                          |                                                                        |
|                                                                         | creat user         | 不阻塞                     | create user tpcc identified by '*****',                       | 不阻塞                      | create user tpcc3 identified by '*****',                               |
|                                                                         | alter user         | 不阻塞                     | ALTER USER tpcc IDENTIFIED BY '*****',                        | 不阻塞                      | ALTER USER tpcc3 IDENTIFIED BY '*****',                                |
| drop user                                                               | 阻塞                 | drop user tpcc CASCADE; | 不阻塞                                                           | drop user tpcc3 CASCADE; |                                                                        |
| creat e/<br>alter/<br>drop sche ma/<br>table space /<br>table/<br>index | create schema      | 不阻塞                     | CREATE SCHEMA school;                                         | 不阻塞                      | CREATE SCHEMA school1;                                                 |
|                                                                         | alter schema       | 不阻塞                     | ALTER SCHEMA school RENAME TO middle_school;                  | 不阻塞                      | ALTER SCHEMA school1 RENAME TO middle_school1;                         |
|                                                                         | drop schema        | 阻塞                      | DROP SCHEMA middle_school CASCADE;                            | 不阻塞                      | DROP SCHEMA middle_school1;                                            |
|                                                                         | create tablesp ace | 不阻塞                     | create tablespace tablespace_school LOCATION '/home/perfadm'; | 不阻塞                      | create tablespace tablespace_school3 LOCATION '/var/standard/var/lib'; |



| 主节点操作 |                  | 验证结果<br>(当前用户、表、库) | 操作语句                                                                   | 验证结果<br>(非当前用户、表、库) | 操作语句                                                                                         |
|-------|------------------|--------------------|------------------------------------------------------------------------|---------------------|----------------------------------------------------------------------------------------------|
|       | alter tablespace | 阻塞                 | create tablespace tablespace_school2 LOCATION '/var/standard/var/lib'; | 不阻塞                 | CREATE TABLE students3 ( id INT, name VARCHAR2(100), age INT) TABLESPACE tablespace_school3; |
|       |                  |                    | ALTER TABLE students SET TABLESPACE tablespace_school2 ;               |                     | create tablespace tablespace_school2 LOCATION '/var/standard/usr/local/core';                |
|       |                  |                    | -                                                                      |                     | ALTER TABLE students3 SET TABLESPACE tablespace_school2;                                     |
|       | drop tablespace  | 不阻塞                | DROP TABLESPACE tablespace_school2 ;                                   | 阻塞                  | DROP TABLESPACE tablespace_school2;                                                          |
|       | create table     | 不阻塞                | create table t1 (id int);                                              | 不阻塞                 | create table t2 (id int);                                                                    |
|       | alter table      | 阻塞                 | ALTER TABLE t1 ADD COLUMN age INT;                                     | 不阻塞                 | ALTER TABLE t2 ADD COLUMN age INT;                                                           |
|       | drop table       | 阻塞                 | drop table t1;                                                         | 不阻塞                 | drop table t2;                                                                               |
|       | create index     | 不阻塞                | CREATE INDEX t1_index1 ON t1(id);                                      | 不阻塞                 | CREATE INDEX t2_index1 ON t2(id);                                                            |
|       | alter index      | 阻塞                 | ALTER INDEX t1_index1 SET (fillfactor = 80);                           | 不阻塞                 | ALTER INDEX t2_index1 SET (fillfactor = 80);                                                 |
|       | drop index       | 阻塞                 | drop INDEX t1_index1;                                                  | 不阻塞                 | drop INDEX t2_index1;                                                                        |

| 主节点操作                                                                         |                                | 验证结果<br>(当前用户、表、库) | 操作语句                                                                                  | 验证结果<br>(非当前用户、表、库) | 操作语句                                                                                  |
|-------------------------------------------------------------------------------|--------------------------------|--------------------|---------------------------------------------------------------------------------------|---------------------|---------------------------------------------------------------------------------------|
| create/<br>alter/<br>drop<br>sequence                                         | create<br>sequence             | 不阻塞                | CREATE<br>SEQUENCE seq1<br>START 101<br>INCREMENT 10;                                 | 不阻塞                 | CREATE SEQUENCE<br>seq3 START 101<br>INCREMENT 10;                                    |
|                                                                               | alter<br>sequence              | 不阻塞                | ALTER SEQUENCE<br>seq1 RESTART<br>WITH 200;                                           | 不阻塞                 | ALTER SEQUENCE<br>seq3 RESTART WITH<br>200;                                           |
|                                                                               |                                |                    | ALTER SEQUENCE<br>seq1 INCREMENT<br>BY 2;                                             |                     | ALTER SEQUENCE<br>seq3 INCREMENT<br>BY 2;                                             |
|                                                                               | drop<br>sequence               | 阻塞                 | DROP SEQUENCE<br>seq1;                                                                | 不阻塞                 | DROP SEQUENCE<br>seq3;                                                                |
| grant<br>/<br>revoke/<br>alter<br>default<br>privileges/<br>reassign<br>owned | grant                          | 不阻塞                | GRANT SELECT ON<br>TABLE t1 TO tpcc;                                                  | 不阻塞                 | GRANT SELECT ON<br>TABLE t1 TO tpcc;                                                  |
|                                                                               |                                |                    | GRANT SELECT,<br>INSERT ON t1 TO<br>tpcc;                                             |                     | GRANT SELECT,<br>INSERT ON t1 TO<br>tpcc;                                             |
|                                                                               | revoke                         | 不阻塞                | REVOKE SELECT<br>ON TABLE t1<br>FROM tpcc;                                            | 不阻塞                 | REVOKE SELECT ON<br>TABLE t1 FROM<br>tpcc;                                            |
|                                                                               | alter<br>default<br>privileges | 不阻塞                | ALTER DEFAULT<br>PRIVILEGES IN<br>SCHEMA public<br>GRANT SELECT ON<br>TABLES TO tpcc; | 不阻塞                 | ALTER DEFAULT<br>PRIVILEGES IN<br>SCHEMA public<br>GRANT SELECT ON<br>TABLES TO tpcc; |
|                                                                               | reassign<br>owned              | 阻塞                 | CREATE ROLE<br>admin WITH<br>LOGIN PASSWORD<br>'*****';                               | 不阻塞                 | CREATE ROLE<br>admin WITH LOGIN<br>PASSWORD '*****';                                  |
|                                                                               |                                |                    | REASSIGN OWNED<br>BY tpcc TO admin;                                                   |                     | REASSIGN OWNED<br>BY tpcc TO admin;                                                   |

| 主节点操作              |                    | 验证结果<br>(当前用户、表、库) | 操作语句                                       | 验证结果<br>(非当前用户、表、库) | 操作语句                                                                                                                                                                                            |
|--------------------|--------------------|--------------------|--------------------------------------------|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| truncate partition | truncate partition | 阻塞                 | ALTER TABLE test_p1 TRUNCATE PARTITION p1; | 不阻塞                 | CREATE TABLE test_p2 (col1 INT, col2 INT) PARTITION BY RANGE (col1) (<br>PARTITION p1 VALUES LESS THAN (10),<br>PARTITION p2 VALUES LESS THAN (20),<br>PARTITION p3 VALUES LESS THAN (30)<br>); |
|                    |                    |                    |                                            |                     | INSERT INTO test_p2 (col1, col2) VALUES (5, 10);                                                                                                                                                |
|                    |                    |                    |                                            |                     | INSERT INTO test_p2 (col1, col2) VALUES (12, 20);                                                                                                                                               |
|                    |                    |                    |                                            |                     | INSERT INTO test_p2 (col1, col2) VALUES (20, 30);                                                                                                                                               |
|                    |                    |                    |                                            |                     | ALTER TABLE test_p2 TRUNCATE PARTITION p1;                                                                                                                                                      |
| reindex            | reindex            | 阻塞                 | REINDEX TABLE t1;                          | 不阻塞                 | REINDEX TABLE t2;                                                                                                                                                                               |

- 处理方法：需通过业务层面对受影响查询进行重试。

备机读场景下 8 级锁冲突引发死锁问题

- 问题现象：当主机端短时间内生成多条8级锁日志时，备机读操作可能触发会话线程死锁并抛出错误码“GAUSS-23827: deadlock detected.”。

- 问题原因：备机回放线程在连续处理8级锁日志时，先持有relation1的8级锁并尝试获取relation2的8级锁，而查询线程持有relation2的1级锁并试图获取relation1的1级锁，形成相互等待的锁循环。会产生8级锁的语句类型：ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL，或未声明其他锁模式的LOCK TABLE操作。
- 处理方法：建议业务侧实施重试应对该异常场景。

## 备机索引扫描与日志回放锁冲突

- 问题现象：在串行回放与并行回放场景下，备机执行读业务时报错：“Index scan conflicts with replay. Try again later. The original maxoff is , but now it is.”。
- 问题原因：备机查询线程在执行索引扫描时，首先对索引页面加读锁，并在扫描过程中逐条检查元组的可见性。若该元组对应事务处于提交中（committing）状态，则需等待其提交后再进行判断。由于备机事务提交依赖日志回放线程推进，而回放线程在此期间会对索引页面执行修改，因此需要加锁。查询线程在等待过程中必须主动释放已获取的索引页锁，否则将形成查询线程等待回放线程提交事务、同时回放线程等待查询线程释放锁的死锁局面。
- 规避方法：通过业务端进行重试。

## 主备元数据异步引发备机查询异常

- 问题现象：在串行回放与并行回放场景下，备机执行读业务时报错：“relation \<string>\" does not exist”、“column \<string>\" does not exist”。
- 问题原因：DDL操作会修改元数据，导致主备元数据不一致。此时备机读操作将因元数据不一致触发上述错误或其他错误信息。此场景下需等待备机完成对应元数据日志回放（与主机元数据状态一致后），备机读业务方能正常执行。
- 规避方案：确保主备元数据同步完成后执行备机读操作。

# 7.28 抗过载

## 7.28.1 SQL 限流

### 7.28.1.1 特性概述

#### 用户场景

在数据库系统运行过程中，时常会遇到突发的业务请求流量过高、或者某些SQL占用资源过高的情况。SQL限流功能可以限制此类SQL的执行，阻止其耗尽数据库资源从而影响正常业务SQL的执行。

#### 功能描述

本特性可以实现多种规则的SQL限流能力，限制CN进程级别的某类SQL语句或全部SQL语句的最大并发数，包括：

- 根据Unique SQL ID进行限流：在明确某条SQL为慢SQL或者占用资源较高时，可以通过Unique SQL ID对该SQL进行限流，避免业务大量执行此SQL而影响其他业务；

- 根据SQL类型及关键字进行限流：在明确某类SQL请求可能会随业务量增长而增长的时候，使用SQL类型和关键字对此类SQL进行限流；
- 根据客户端信息进行限流：在明确某客户端业务会占用资源较高时，可通过客户端信息，包括IP、ApplicationName，进行限流，避免该客户端业务大量执行SQL而影响其他业务。
- 由于某些业务高峰是可以预知的，在仅希望在业务高峰时段对SQL请求进行限制的情况下，可以设置SQL限流的生效时间，避免限流规则常驻系统；
- 当只希望限流规则作用于业务库，而对系统库的SQL不做控制，可以按不同库的维度进行限流；
- 除了对某类SQL进行限流，本特性还提供CN进程级别的限流能力；
- 对于限流规则，提供查询统计的能力，可以查询所有的限流规则，并根据规则列表对限流规则进行管理。此外还可以查询限流规则的生效次数。

## 规格约束

- 只支持在CN上限流。
- 对于Unique SQL ID限流，需要设置GUC参数enable\_resource\_track = on, instr\_unique\_sql\_count > 0。
- 对于关键字限流，按照并发度排序，并发度越低优先级越高。关键字不区分大小写，支持模糊匹配。
- 对于客户端信息限流，优先采用精确匹配规则（需同时匹配IP与ApplicationName）。若无生效的精确匹配规则，则针对仅匹配IP或仅匹配ApplicationName的规则，按并发度升序排序，并发度越低者优先级越高。所有关键字均区分大小写，且不支持模糊匹配。
- 数据库名称区分大小写。删除某个数据库，再创建同名的数据库，会导致所有已录入的针对这个数据库的限流规则失效。
- 基于资源的CN进程级别最大活跃并发数限流，按照并发度排序，并发度越低优先级越高，资源利用率的采集存在10s左右的时间差。
- 对于限流周期结束的规则，系统会在下次限流规则触发时将其缓存中的is\_valid标记为false。系统表内的is\_valid字段由后台线程负责检测与刷新，该过程存在延迟，最长延迟时间不超过10分钟。
- 在事务中对限流规则执行了增、删、改操作且该事务最终被回滚时，会导致限流规则缓存中的数据与限流规则系统表不一致，此时需及时调用gs\_refresh\_workload\_rule\_cache函数刷新限流规则缓存。
- 对于SQL限流次数的统计是数据库从启动到当前的累计次数，重启后从0开始统计。
- 游标、存储过程中的SQL语句不会被限流。
- 管理员用户执行的SQL语句不会被限流。
- 限流规则数据库间不共享，创建限流规则时需要连接目标库。如果创建限流规则的CN被剔除并触发全量Build，则会继承全量Build目标CN中的限流规则，因此建议尽量在各个CN上都创建对应的限流规则。
- CN之间由于Unique SQL ID不同，不共享限流规则，需要用户手动在不同的CN上创建对应的限流规则。

## 7.28.1.2 技术原理

### 核心机制

GaussDB启动时，会将SQL限流规则从系统表加载到内存中。SQL限流的流程如下：

1. 用户执行任意SQL语句时，GaussDB会根据该SQL语句去内存中匹配满足条件的限流规则。
2. 如果匹配到限流规则，将内存中该规则的dop字段值加1。dop字段用于记录限流规则的实时并发度。
3. 如果匹配到的限流规则的dop字段值大于或等于该规则允许的最大并发度，则进行限流：立即终止该SQL语句的执行，并报错提示用户。
4. 当限流生效，或限流未生效SQL语句正常执行完毕时，将匹配到的限流规则的dop字段值减1。

### 接口介绍

SQL限流主要使用以下系统函数，具体描述请参见《参考》中“SQL参考 > 函数和操作符 > SQL限流函数”章节。

| 函数名                            | 函数说明                                                      |
|--------------------------------|-----------------------------------------------------------|
| gs_add_workload_rule           | 创建一条SQL限流规则。需要具有sysadmin权限的用户才可执行。                        |
| gs_update_workload_rule        | 更新一条SQL限流规则，需要重新设置全部参数，不支持只指定部分参数。需要具有sysadmin权限的用户才可执行。  |
| gs_delete_workload_rule        | 删除一条SQL限流规则。需要具有sysadmin权限的用户才可执行。                        |
| gs_get_workload_rule_stat      | 查询SQL限流规则拦截SQL语句的次数。需要具有sysadmin权限的用户才可执行。                |
| gs_refresh_workload_rule_cache | 刷新老限流规则缓存，使其与限流规则系统表保持一致。不支持在事务中执行。需要具有sysadmin权限的用户才可执行。 |

## 7.28.1.3 操作指导

### 前提条件

- GaussDB版本 ≥ 505.0.0。
- 需要具有sysadmin权限的用户执行以下操作。

### 示例

```
-- 创建根据Unique SQL ID进行限流，且规则名称为rule_1、Unique SQL ID为844424930505717、最大并发度为10、立即生效的SQL限流规则。
gaussdb=# SELECT gs_add_workload_rule('sqlid', 'rule_1', '{}', NULL, NULL, 10, '{id=844424930505717}');
gs_add_workload_rule

```

```
1
(1 row)

-- 创建根据SQL类型及关键字进行限流，且SQL类型为select、数据库为postgres和tpcc、关键字为group by和
join、最大并发度为1、指定生效时间段的SQL限流规则。
gaussdb=# SELECT gs_add_workload_rule('select', 'rule_2', '{postgres, tpcc}', '2026-01-01 00:00:00',
'2026-01-02 00:00:00', '1', '{group by, join}');
gs_add_workload_rule

2
(1 row)

-- 创建根据系统资源进行限流，且当CPU使用率大于60%，同时内存使用率大于80%时生效、最大并发度为10的
SQL限流规则。
gaussdb=# SELECT gs_add_workload_rule('resource', 'rule_3', '{}', NULL, NULL, '10', '{cpu-60, memory-80}');
gs_add_workload_rule

3
(1 row)

-- 创建根据客户端ApplicationName进行限流，且规则名称为rule_4、ApplicationName为app1、最大并发度为
20、立即生效的SQL限流规则。
gaussdb=# SELECT gs_add_workload_rule('client', 'rule_4', '{}', NULL, NULL, 20, '{appname=app1}');
gs_add_workload_rule

4
(1 row)

-- 修改rule_id为1的SQL限流规则（该规则类型为根据Unique SQL ID进行限流），修改Unique SQL ID为
676844332506578、最大并发度为20、生效时间为指定时间段。
gaussdb=# SELECT gs_update_workload_rule(1, 'rule_1', '{}', '2026-01-01 00:00:00', '2026-01-02 00:00:00',
20, '{id=676844332506578}');
gs_update_workload_rule

t
(1 row)

-- 刷新限流规则缓存。
gaussdb=# SELECT gs_refresh_workload_rule_cache();
gs_refresh_workload_rule_cache

t
(1 row)

-- 查询rule_id为1的SQL限流规则的统计信息。
gaussdb=# SELECT * from gs_get_workload_rule_stat(1);
rule_id | validate_count
-----+-----
1 | 0
(1 row)

-- 查询全部SQL限流规则的统计信息。
gaussdb=# SELECT * from gs_get_workload_rule_stat(-1);
rule_id | validate_count
-----+-----
1 | 0
2 | 0
3 | 0
4 | 0
(4 rows)

-- 删除rule_id为1的SQL限流规则。
gaussdb=# SELECT gs_delete_workload_rule(1);
gs_delete_workload_rule

t
(1 row)

-- 删除全部SQL限流规则。
```

```
gaussdb=# SELECT gs_delete_workload_rule(rule_id) FROM gs_workload_rule;
gs_delete_workload_rule

t
t
t
(3 rows)
```

#### 7.28.1.4 常见问题

问题：找不到gs\_workload\_rule系统表。

解决：

SQL限流特性是自505.0.0版本开始引入的，请确定您的数据库版本大于等于此版本。

问题：设置了SQL限流规则，然后执行SQL语句，但没有触发SQL限流。

解决：

可以按下面步骤排查：

- 查看GUC参数enable\_workload\_rule的取值是否为on？当该参数取值为off时，SQL限流不生效。
- 查看执行SQL语句的用户是不是管理员？SQL限流对管理员用户无效。
- 查看执行SQL语句的用户是否在SQL限流白名单中？白名单中的用户不会触发SQL限流。查看方法如下：  
gaussdb=# SHOW workload\_uncontrolled\_users;
- 查看SQL限流规则设置是否正确？例如：sqlid是否正确；数据库是否正确；关键字是否正确；SQL限流规则是否过期；SQL限流生效的开始时间是否小于结束时间；max\_workload是否设置正确等。

### 7.28.2 内存抗过载

#### 7.28.2.1 特性概述

##### 功能定义

数据库内存出现过载问题时，快速关闭会话并禁止新连接接入，待内存恢复正常状态后恢复对外服务能力。

##### 用户场景

当数据库由于I层异常使得SQL执行时延显著上升，导致大量会话积压，进一步引发内存资源耗尽，最终使数据库无法对外提供服务时，能够快速实现逃生，短时间内恢复对外提供服务的能力。

##### 说明

I层指的是IaaS(Infrastructure as a Service)，主要提供计算、存储、网络类基础服务，典型的I层云服务，例如：弹性云服务器。



## 规格约束

- 内存过载触发逃生时，默认不对sysadmin或monitoradmin权限的用户的session做清理操作，若想对sysadmin或monitoradmin权限的用户的session做清理操作，请设置GUC参数resilience\_escape\_user\_permissions，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节的详细描述和使用方法。
- 升级模式下，不触发该特性功能。

### 7.28.2.2 技术原理

#### 核心机制

GUC参数resilience\_memory\_reject\_percent会设置两个阈值：  
"recover\_memory\_percent, overload\_memory\_percent"，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

wlm\_monitor线程定时检查动态内存使用率是否大于overload\_memory\_percent %，条件满足时触发过载逃生，阻止后续的会话接入，优先关闭处于idle状态的会话，会话状态相同时按照内存使用量降序关闭会话。当动态内存使用率降至小于recover\_memory\_percent %后，停止关闭会话并允许连接接入。

动态内存使用率 =  $\text{dynamic\_used\_memory} / \text{max\_dynamic\_memory} * 100\%$ ，  
dynamic\_used\_memory和max\_dynamic\_memory可以通过查询  
PV\_TOTAL\_MEMORY\_DETAIL视图获取，具体描述请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PV\_TOTAL\_MEMORY\_DETAIL”章节。

会话状态可以通过查询PG\_STAT\_ACTIVITY视图获取，具体描述请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG\_STAT\_ACTIVITY”章节。

#### 接口介绍

内存抗过载主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

表 7-41 GUC 参数

| 参数名                  | 参数说明                           |
|----------------------|--------------------------------|
| use_workload_manager | 设置是否开启资源管理功能，该参数打开是内存抗过载生效的前提。 |

内存抗过载主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

表 7-42 GUC 参数

| 参数名                 | 参数说明                           |
|---------------------|--------------------------------|
| enable_memory_limit | 设置是否打开内存管理功能，该参数打开是内存抗过载生效的前提。 |

| 参数名                                | 参数说明                                 |
|------------------------------------|--------------------------------------|
| resilience_memory_reject_percent   | 用于控制内存过载逃生功能的动态内存占用百分比。              |
| resilience_escape_user_permissions | 设置是否开启sysadmin和monadmin权限用户的内存抗过载功能。 |

### 7.28.2.3 操作指导

#### 前提条件

GUC参数use\_workload\_manager及enable\_memory\_limit参数打开，这两个参数默认是打开的，resilience\_memory\_reject\_percent需要手动设置。

#### 操作步骤

**步骤1** 查看enable\_memory\_limit参数是否打开。

```
gaussdb=# SHOW enable_memory_limit;
enable_memory_limit

on
(1 row)
```

**步骤2** 查看use\_workload\_manager参数是否打开。

```
gaussdb=# SHOW use_workload_manager;
use_workload_manager

on
(1 row)
```

**步骤3** 设置resilience\_memory\_reject\_percent参数，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

**步骤4** 如果需要设置sysadmin和monadmin权限用户过载逃生，请设置resilience\_escape\_user\_permissions参数，参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

----结束

### 7.28.2.4 常见问题

**问题：**过载逃生不生效，导致数据库动态内存使用率高于逃生阈值。

**解决：**查看use\_workload\_manager、enable\_memory\_limit是否打开，若关闭则无法生效。

```
gaussdb=# SHOW enable_memory_limit;
enable_memory_limit

on
(1 row)

gaussdb=# SHOW use_workload_manager;
use_workload_manager

on
(1 row)
```

查看resilience\_memory\_reject\_percent参数是否成功设置，如果设置时恢复阈值大于触发逃生的阈值，会导致参数设置失败，仍为默认值，通过系统视图pg\_roles查看业务的执行用户是否具有sysadmin或monadmin权限，有这两种权限的用户默认不触发过载逃生。

```
gaussdb=# SHOW resilience_memory_reject_percent;
resilience_memory_reject_percent

50,90
(1 row)

gaussdb=# SELECT rolname, rolsystemadmin, rolmonitoradmin FROM pg_roles WHERE rolname = 'Mike';
rolname | rolsystemadmin | rolmonitoradmin
-----+-----+-----
Mike | t | t
(1 rows)

gaussdb=# SHOW resilience_escape_user_permissions;
resilience_escape_user_permissions

(1 row)
```

问题：频繁触发过载逃生。

解决：查看resilience\_memory\_reject\_percent设置是否过小，将过载的阈值增大。

## 7.28.3 线程池抗过载

### 7.28.3.1 特性概述

#### 功能定义

数据库线程池出现过载问题时，快速关闭会话并禁止新连接接入，待线程池恢复正常状态后恢复对外服务能力。

#### 用户场景

当数据库由于I层异常使得SQL执行时延显著上升，导致大量会话积压，进一步引发线程池资源耗尽，最终使数据库无法对外提供服务时，能够快速实现逃生，短时间内恢复对外提供服务的能力。

#### 说明

I层指的是IaaS(Infrastructure as a Service)，主要提供计算、存储、网络类基础服务，典型的I层云服务，例如：弹性云服务器。

#### 规格约束

- GUC参数use\_workload\_manager、enable\_thread\_pool打开时生效。
- 线程池过载触发逃生时，默认不对sysadmin或monitoradmin权限的用户的session做清理操作，若想对sysadmin或monitoradmin权限的用户的session做清理操作，请设置GUC参数resilience\_escape\_user\_permissions，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节的详细描述和使用方法。
- 升级观察期下，不触发该特性功能。

7.28.3.2 技术原理

核心机制

GUC参数resilience\_threadpool\_reject\_cond会设置两个阈值：  
“recover\_threadpool\_percent, overload\_threadpool\_percent” ,

wlm\_monitor线程定时检查线程池使用率是否大于overload\_threadpool\_percent %，  
条件满足时触发过载逃生，阻止后续的会话接入，优先关闭处于idle状态的会话，会话  
状态相同时按照查询的开始时间升序关闭会话。当线程池使用率降至小于  
recover\_threadpool\_percent %后，停止关闭会话并允许连接接入。

线程池的使用状态可以通过查询DBE\_PERF.LOCAL\_THREADPOOL\_STATUS视图获取，  
参见《参考》中“Schema > DBE\_PERF Schema > Session/Thread >  
DBE\_PERF.LOCAL\_THREADPOOL\_STATUS” 章节。

会话状态及查询的开始时间可以通过查询PG\_STAT\_ACTIVITY视图获取，参见《参考》  
中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG\_STAT\_ACTIVITY” 章节。

接口介绍

线程池抗过载主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数  
说明 > GUC参数说明 > 负载管理” 章节。

表 7-43 GUC 参数

| 参数名                  | 参数说明                                |
|----------------------|-------------------------------------|
| use_workload_manager | 设置是否开启资源管理功能，该参数打开是线程池抗过<br>载生效的前提。 |

线程池抗过载主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数  
说明 > GUC参数说明 > 资源消耗 > 内存” 章节。

表 7-44 GUC 参数

| 参数名                                    | 参数说明                                      |
|----------------------------------------|-------------------------------------------|
| resilience_escape_user_p<br>ermissions | 设置是否开启sysadmin和monadmin权限用户的线程池<br>抗过载功能。 |

线程池抗过载主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数  
说明 > GUC参数说明 > 线程池” 章节。

表 7-45 GUC 参数

| 参数名                | 参数说明         |
|--------------------|--------------|
| enable_thread_pool | 控制是否使用线程池功能。 |

| 参数名                               | 参数说明                  |
|-----------------------------------|-----------------------|
| resilience_threadpool_reject_cond | 用于控制线程池过载逃生的线程池使用率比例。 |

### 7.28.3.3 操作指导

#### 前提条件

GUC参数use\_workload\_manager及enable\_thread\_pool参数打开，这两个参数默认打开，resilience\_threadpool\_reject\_cond需要手动设置。

#### 操作步骤

**步骤1** 查看enable\_thread\_pool参数是否打开。

```
gaussdb=# SHOW enable_thread_pool;
enable_thread_pool

on
(1 row)
```

**步骤2** 查看use\_workload\_manager参数是否打开。

```
gaussdb=# SHOW use_workload_manager;
use_workload_manager

on
(1 row)
```

**步骤3** 设置resilience\_threadpool\_reject\_cond参数，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 线程池”章节。

**步骤4** 如果需要设置sysadmin和monadmin权限用户过载逃生，请设置resilience\_escape\_user\_permissions参数，参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

----结束

### 7.28.3.4 常见问题

问题：频繁触发过载逃生。

解决：查看resilience\_threadpool\_reject\_cond设置的过载阈值是否过小，将过载的阈值增大。

问题：过载逃生不生效，线程池使用率高于逃生阈值。

解决：查看use\_workload\_manager和enable\_thread\_pool参数是否打开，若关闭则过载逃生功能无法生效。如果均已打开，查询resilience\_threadpool\_reject\_cond参数是否成功设置。

```
gaussdb=# SHOW enable_thread_pool;
enable_thread_pool

on
(1 row)
```

```
gaussdb=# SHOW use_workload_manager;
use_workload_manager

on
(1 row)

gaussdb=# SHOW resilience_threadpool_reject_cond;
resilience_threadpool_reject_cond

50,90
(1 row)
```

问题：触发过载逃生后，连接一直无法恢复。

解决：检查sysadmin和monadmin权限用户是否占用线程过多，本功能默认不会关闭sysadmin和monadmin权限用户的会话，触发过载逃生后可能导致一直无法恢复，通过设置resilience\_escape\_user\_permission，控制sysadmin和monadmin权限用户的逃生。

## 7.28.4 建连线程池

### 7.28.4.1 特性概述

#### 用户场景

在数据库系统运行过程中，经常出现由于并发建连数量大，建连任务抢占线程池，导致业务缺少线程池资源，出现业务吞吐量大幅下降的情况。增加建连线程池可以减少连接的接入对已建连业务执行SQL性能的影响，防止业务执行缓慢或吞吐量大幅跌落。

#### 功能描述

通过在已有线程池中拆分出一个独立线程池，专门用于数据库处理建连流程，将业务与建连进行解耦，不在同一个线程池中争抢资源，互不干扰。

#### 规格约束

- 运维端口接入的连接不使用建连线程池。
- 当且仅当GUC参数enable\_connect\_thread\_pool和enable\_thread\_pool取值都为on时才能开启建连线程池功能。
- 升级模式下，不触发该特性功能。

### 7.28.4.2 技术原理

#### 核心机制

打开建连线程池功能后，建连流程会发生变化，GaussDB收到建连事件时，会将任务派发到建连线程池中，由该线程池中的worker线程绑定到此连接对应的session执行初始化，初始化完成后，继续执行建连阶段的sql语句，建连结束后，将session派发到业务线程池中执行后续业务，详细流程如下：

1. 客户端（gsq或jdbc等）触发建连操作。
2. GaussDB收到建连请求，创建session，派发session到建连线程池中。

3. 建连线程池获取一条空闲ConnTPLWorker线程绑定session执行建连初始化，包括鉴权、首包处理等。
4. 连接初始化完成后，由ConnTPLWorker线程继续执行建连阶段的sql语句。
5. 建连结束后，由ConnTPLWorker线程将session派发至业务线程池中等待下一次任务。
6. 业务线程池获取一条空闲TPLWorker线程绑定session执行后续sql业务并返回执行结果。

## 关键接口

建连线程池主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 线程池”章节。

表 7-46 GUC 参数

| 参数名                        | 参数说明                      |
|----------------------------|---------------------------|
| enable_thread_pool         | 控制线程池是否打开的开关，默认为on。       |
| enable_connect_thread_pool | 控制建连线程池是否打开的开关，默认为on。     |
| connect_thread_pool_attr   | 控制建连线程池的初始配置参数，默认为'auto'。 |

### 7.28.4.3 操作指导

#### 前提条件

GUC参数enable\_connect\_thread\_pool和enable\_thread\_pool取值都为on。

#### 操作步骤

**步骤1** 查看enable\_connect\_thread\_pool参数是否打开。

```
gaussdb=# show enable_connect_thread_pool;
enable_connect_thread_pool

on
(1 row)
```

**步骤2** 查看enable\_thread\_pool参数是否打开。

```
gaussdb=# show enable_thread_pool;
enable_thread_pool

on
(1 row)
```

**步骤3** 查看建连线程池参数配置，group\_id为-1表示是建连线程池。

```
gaussdb=# SELECT * from DBE_PERFLOCAL_THREADPOOL_STATUS where group_id='-1';
node_name | group_id | bind_numa_id | bind_cpu_number | listener |
worker_info | | session_info |
| stream_info |
-----+-----+-----+-----+-----
```

```
+-----+-----+
+-----+
singlenode | -1 | | 0 | 1 | default: 8 new: 0 expect: 8 actual: 8 idle: 8 pending: 0 |
total: 0 waiting: 0 running: 0 idle: 0
| total: 0 running: 0 idle: 0
(1 row)
```

----结束

## 7.28.5 并发场景支持抗过载逃生能力

### 7.28.5.1 特性概述

#### 功能定义

将执行时间超过一定时间的SQL识别为慢SQL，限制慢SQL占用线程池总线程数的百分比及限制慢SQL的IOPS。

#### 用户场景

慢SQL过多导致无空闲线程池线程供给正常SQL使用。

#### 规格约束

功能约束：

- 需要GUC参数enable\_thread\_pool取值为on。
- 仅对非sysadmin、monitoradmin用户执行的SELECT类型的语句生效。
- 限制IOPS功能需要GUC参数use\_workload\_manager取值为on。

### 7.28.5.2 技术原理

#### 核心机制

设置GUC参数resilience\_ctrlstmt\_detect\_timelimit或wlmrule Plan Hint，业务线程会在执行过程中设置对应时间的定时器，定时器到期后将当前业务线程标识为执行慢SQL的线程，对于执行慢SQL的线程将执行以下管控：

- 当执行慢SQL的线程数量超过resilience\_ctrlslot\_available\_maxpercent取值乘以线程池线程数量最大值后，后续执行慢SQL的线程强制cancel。
- 当执行慢SQL的线程IOPS超过resilience\_ctrlstmt\_control\_iopslimit，使对应线程睡眠0~1秒内的随机值。

#### 接口介绍

快慢车道主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。



表 7-47 GUC 参数

| 参数名                                      | 参数说明                                                            |
|------------------------------------------|-----------------------------------------------------------------|
| resilience_ctrlslot_available_maxpercent | 最多允许慢SQL占用线程池线程数的百分比。                                           |
| resilience_ctrlstmt_detect_timelimit     | 正常SQL语句被标记为慢SQL所需的执行时间，0表示不做慢SQL识别，大于0表示当SQL执行时间超过该时间时被标记为慢SQL。 |
| resilience_ctrlstmt_control_iopslimit    | 正常SQL语句被标记为慢SQL后，慢SQL可使用的最大IOPS上限。                              |

快慢车道使用wlmrule Plan Hint，具体描述请参见《SQL调优指南》中“使用Plan Hint进行调优 > 设置慢SQL管控规则的Hint”章节。

表 7-48 Plan Hint

| Plan Hint                                         | Plan Hint说明                                             |
|---------------------------------------------------|---------------------------------------------------------|
| wlmrule("time_limit, max_execute_time, max_iops") | 针对想要进行执行时间/资源管控的SQL语句，设置其被标记为慢SQL的执行时间，最大执行时间，最大IOPS上限。 |

### 7.28.5.3 操作指导

#### 前提条件

GUC参数enable\_thread\_pool取值为on，若需要对慢SQL限制IOPS则需设置GUC参数use\_workload\_manager取值为on。

#### 注意事项

- GUC参数全局生效，wlmrule Plan Hint仅对设置的SQL语句生效。
- GUC参数resilience\_ctrlstmt\_detect\_timelimit、resilience\_ctrlstmt\_control\_iopslimit与wlmrule Plan Hint中的设置项存在重叠，在设置了对应GUC参数的情况下，若SQL语句设置了wlmrule Plan Hint，仅wlmrule Plan Hint中的设置项生效，否则对应GUC参数生效。
- wlmrule Plan Hint还可额外通过max\_execute\_time设置SQL语句的最大执行时间，执行时间超过该时长后被强制cancel。
- GUC参数resilience\_ctrlstmt\_detect\_timelimit取值为0时，即使resilience\_ctrlslot\_available\_maxpercent和resilience\_ctrlstmt\_control\_iopslimit有值，对应功能也无法生效。

#### 操作步骤

GUC参数的操作指导请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

wlmrule Plan Hint的操作指导请参见《SQL调优指南》中“使用Plan Hint进行调优 > 设置慢SQL管控规则的Hint”章节。

#### 7.28.5.4 常见问题

问题：设置resilience\_ctrlstmt\_control\_iopslimit后IO管控未明显生效？

解答：检查GUC参数use\_workload\_manager取值是否为on，由于执行慢SQL的线程睡眠0~1秒内的随机值，无法将IOPS完全控制在resilience\_ctrlstmt\_control\_iopslimit以下。

## 7.29 Unix Domain Socket 运维通道

### 7.29.1 特性概述

#### 功能定义

GaussDB提供的应急链路，客户端可以通过接入该链路进行应急命令下发，本功能后简称“应急通道”。

#### 用户场景

- 极端负载场景下，如CPU过载、内存过载、线程池满，连接池满，CM（集群管理组件）通过网络建连（libpq）连接数据库探活可能失败，进而触发主备倒换，通过应急通道可以避免此场景下的主备倒换。
- 运维人员通过运维端口连接数据库也可能失败，导致一些关键运维命令无法及时下发，此时可以通过应急通道进行关键命令下发。

#### 规格约束

- 仅支持CN/DN节点，不支持1主1备1日志组网中的日志节点。
- 本需求支持场景：
  - GaussDB进程CPU负载峰值达到95%。
  - 最大连接数达到max\_connections上限，无法新建连接，GUC参数max\_connections的描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 连接和认证 > 连接设置”章节。
  - 内存过载，日志中已不断报memory is temporarily unavailable类错误，无法正常分配内存执行SQL。
  - 线程池占满，出现会话堆积，新会话处理缓慢。
- 通过gsql进行应急通道视图查询或运维，5s内返回结果。mem\_log生成为异步执行，保证5s内触发生成，生成完毕的时间受磁盘I/O、内存上下文数量及业务并发影响；会话查杀为异步执行，保证5s内下发查杀信号，查杀的成功性及时间受会话执行的业务、锁资源等因素影响。
- 应急通道运维仅支持本地连接节点。
- 通过gsql进行mem\_log记录需开启use\_workload\_manager、enable\_memory\_limit，请参见[操作指导](#)。
- 高CPU场景，需保证enable\_control\_group参数的开启（默认开启，可进行CPU隔离），才可以保证应急通道100%运维成功，GUC参数enable\_control\_group具体

描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

## 7.29.2 技术原理

### 核心机制

- 进程间通信

内核与CM交互链路不应受制于网络问题、高负载问题，采用进程间通信方式交互，具体使用了Unix domain socket（后简称UDS）实现。

分析CM探活的场景，需考虑以下几点：

- 指标时效：CM探活指标获取频率较高，部分指标采集周期为1次/s，采用共享内存具有滞后性，且需要额外考虑对指标时效性的判断；使用UDS的实时性较高，较符合该场景。
- 参数可变：原CM探活是通过网络建连后执行SQL的方式，CM会动态的通过参数拼接SQL，内核侧无法提前预判入参并采集。在这种场景下，CM需将查询指令置于共享内存，内核读取后再执行指标查询操作，最终转化为更大的时效问题。

综合考虑采用UDS方式实现CM和内核的交互。

- 应急通道资源预留

内核应急通道预留资源，以在极端负载场景下仍能提供探活能力，且在执行探活指标获取/运维的过程中，尽可能少地依赖I/O。

- 预留CPU资源：绑定HA\_BACKEND cgroup组；
- 预留内存资源：分析应急通道线程所使用的内存较少，但优先级极高，因此内存上下文用量不设上限，通过应急通道线程查询视图或执行运维操作前执行内存上下文切换。

- 内核、应急通道线程与CM(gsql)交互主流程

- CM、内核完成UDS的初始化。
- CM与内核建立长连接。
- CM内线程按需、定期向内核发送报文以查询指标或进行仲裁后的运维指令下发。
- 内核解析报文，反序列化参数，根据指令执行查询/运维操作。对于运维操作，根据具体命令逻辑确定是同步/异步接口，对于指标查询均为同步接口。内核获取结果集后进行序列化、报文封装及报文发送。特别地，内核在获取CM所需信息时，确保只加SpinLock和极少数必要的哈希表相关锁。

考虑该长连接可能由于异常原因断开或阻塞，为保证探活链路通畅需有如下机制：

- 内核探活线程需有保活机制，当异常退出后应及时重拉。
- CM与内核的链路若异常断开，CM需进行一定次数的建连尝试，若无法成功建连，则结合原网络建连探活的结果仲裁节点状态。
- CM与内核间的通信应设置超时与重试机制，当请求达到一定重试次数仍未返回结果时，则结合原网络建连探活的结果仲裁节点状态。

- 探活机制

CM通过网络链路探活的原核心流程为建连后发送“SELECT 1”，高负载下通过网络链路探活失败时，会通过应急通道探活作为代替。因此内核新增一个探活指令，其具体实现为：通过wlm\_monitor线程异步执行SELECT 1操作并记录结果，

覆盖除建连外的SQL执行过程链路的检测，应急通道线程对结果进行获取并反馈给CM。

- 简易运维命令  
gsql提供简易运维元命令，其原理是gsql客户端执行简易命令时，将简易运维命令翻译为指定的sql后发送给服务端。

接口介绍

- GUC参数  
Unix Domain Socket运维通道主要使用以下GUC参数，具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

表 7-49 GUC 参数

| 参数名                     | 参数说明                           |
|-------------------------|--------------------------------|
| enable_emergency_access | 设置是否开启Unix Domain Socket 运维通道。 |

- 应急通道的运维命令  
gsql工具提供-O选项执行应急通道的运维命令，具体示例参见[操作指导](#)。gsql的使用请参见《参考》中“工具参考 > 数据库连接工具 > gsql连接数据库 > gsql使用指导”章节。

表 7-50 gsql 支持的 Unix Domain Socket 运维通道的运维命令

| 参数                                           | 参数说明                                                                                                                                                                                                                                                                                                                                                                                                                           | 取值范围                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -O, --emergency-command=E<br>MERGENCYCOMMAND | <p>Unix Domain Socket运维通道的运维命令。</p> <p>在极端情况下，gsql工具无法与GaussDB建立连接时，可以使用该命令进行应急运维。此命令通过Unix Domain Socket运维通道与GaussDB通信，实现一键运维功能。</p> <p>-O选项的参数格式具体如下：<br/>"{port} {command} [{param} [...]]"</p> <p>{port}：数据库监听端口。</p> <p>{command}：具体运维命令。</p> <p>{param}：该运维命令的参数，可能有0个或多个。</p> <p><b>说明</b></p> <ul style="list-style-type: none"> <li>使用此选项时，gsql工具必须与GaussDB进程位于同一台服务器上。</li> <li>如果运维命令的参数包含空格，请在参数两边加上引号。</li> </ul> | <ul style="list-style-type: none"> <li>"{port} mem_log"：记载全局的内存概况与全局级内存上下文、线程级内存上下文、session级内存上下文的top20内存上下文占用详情。该功能仅在GUC参数use_workload_manager开启时有效，范围同内存过载记录mem_log特性。无参数。示例：<br/>-O "19995 mem_log"</li> <li>"{port} kill_wait_event {wait_event}"：查杀指定等待事件会话。参数{wait_event}表示具体的锁、轻量级锁以及I/O的信息等。示例：<br/>-O "19995 kill_wait_event relation"</li> <li>"{port} kill_business_sql"：查杀非rdsAdmin用户的全部业务SQL语句。无参数。示例：<br/>-O "19995 kill_business_sql"</li> <li>"{port} kill_slow_sql {username}"：批量查杀非默认数据库下，指定用户的状态为非idle的执行超过1分钟的SQL语句。参数{username}表示用户名。示例：<br/>-O "19995 kill_slow_sql omm"</li> <li>"{port} kill_slow_sql {username} {database}"：批量查杀指定用户在指定数据库中的执行超过1分钟的SQL语句。参数{username}表示用户名，{database}表示数据库名。示例：<br/>-O "19995 kill_slow_sql omm db1"</li> <li>"{port} kill_executing_sql {username}"：查杀非默认数据库下，指定用户的状态为active的正在执行</li> </ul> |

| 参数 | 参数说明 | 取值范围                                                                                                                                                                                                                                                                                                                                                                           |
|----|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    |      | <p>的SQL语句。参数 {username}表示用户名。</p> <p>示例：</p> <p>-O "19995 kill_executing_sql omm"</p> <ul style="list-style-type: none"><li>• "{port} kill_mem_topn {top n}": 查杀内存占用 top n的SQL语句。参数 {top n}表示指定的top n。示例：<br/>-O "19995 kill_mem_topn 10"</li><li>• "{port} kill_cpu_topn {top n}": 查杀CPU TIME 占用top n的SQL语句。参数{top n}表示指定的top n。示例：<br/>-O "19995 kill_cpu_topn 10"</li></ul> |

- gsql的简易运维元命令  
gsql使用简易运维元命令使用示例参见[操作指导](#)，gsql的元命令使用请参见《参考》中“工具参考 > 数据库连接工具 > gsql连接数据库 > 元命令参考”章节。

表 7-51 常用简易运维元命令

| 简易运维命令                                           | 参数解释                                                                               | 实际执行的SQL                                                                                                       |
|--------------------------------------------------|------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>\view stmt_track {unique_sql_id }</code>   | stmt_track：记录指定SQL的L2级别的FULLSQL信息。<br>unique_sql_id：需要记录的SQL的unique_sql_id。        | <code>SELECT * FROM pg_catalog.dynamic_func_control('LOCAL', 'STMT', 'TRACK', '{unique_sql_id}', 'L2');</code> |
| <code>\view stmt_untrack {unique_sql_id }</code> | stmt_untrack：取消记录指定SQL的L2级别的FULLSQL信息。<br>unique_sql_id：需要取消记录的SQL的 unique_sql_id。 | <code>SELECT * FROM pg_catalog.dynamic_func_control('LOCAL', 'STMT', 'UNTRACK', '{unique_sql_id}');</code>     |
| <code>\view stmt_list</code>                     | stmt_list：查询记录列表。                                                                  | <code>SELECT * FROM pg_catalog.dynamic_func_control('LOCAL', 'STMT', 'LIST', {});</code>                       |
| <code>\view stmt_clean</code>                    | stmt_clean：清理记录列表。                                                                 | <code>SELECT * FROM pg_catalog.dynamic_func_control('LOCAL', 'STMT', 'CLEAN', {});</code>                      |

| 简易运维命令                        | 参数解释                                                 | 实际执行的SQL                                                                                                                                                                                                                                                                                                                   |
|-------------------------------|------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view explain_and_stack {pid} | explain_and_stack: 查询指定会话的运行计划及堆栈。<br>pid: 指定会话的pid。 | SELECT<br>pg_catalog.gs_get_explain(pid), pg_catalog.gs_stack(pid)<br>FROM<br>pg_catalog.pg_stat_activity<br>WHERE pid = {pid};                                                                                                                                                                                            |
| \view slow_sql {top_n}        | slow_sql: 查询实时慢SQL情况。<br>top_n: 查询的行数。               | SELECT pid, sessionid, pg_catalog.SUBSTRING(query, 0, 100), state, username, pg_catalog.NOW() - query_start AS elp FROM pg_catalog.pg_stat_activity WHERE state != 'idle' AND username NOT IN ('rdsAdmin', 'rdsMetric', 'rdsBackup', 'rdsRepl', 'dbmind_monitor_agent', 'dbmind_monitor') ORDER BY elp DESC LIMIT {top_n}; |
| \view slow_trans {top_n}      | slow_trans: 查询实时长事务情况。<br>top_n: 查询的行数。              | SELECT pid, sessionid, pg_catalog.SUBSTRING(query, 0, 100), state, username, pg_catalog.NOW() - xact_start AS elp FROM pg_catalog.pg_stat_activity WHERE state != 'idle' AND username NOT IN ('rdsAdmin', 'rdsMetric', 'rdsBackup', 'rdsRepl', 'dbmind_monitor_agent', 'dbmind_monitor') ORDER BY elp DESC LIMIT {top_n};  |

| 简易运维命令                                                               | 参数解释                                                                                                                                                                    | 实际执行的SQL                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view<br>wait_event                                                  | wait_event（不带参数）：<br>查询当前语句和等待事件。                                                                                                                                       | SELECT a.pid, a.sessionid,<br>a.datname, a.username,<br>a.client_addr, a.xact_start,<br>a.query_start,<br>(pg_catalog.NOW() -<br>a.query_start)::text<br>"query_runtime",<br>w.wait_status, w.wait_event,<br>w.locktag, w.lockmode,<br>w.block_sessionid, a.query<br>FROM<br>pg_catalog.pg_stat_activity a<br>JOIN<br>pg_catalog.pg_thread_wait_status w ON a.sessionid =<br>w.sessionid WHERE a.pid <><br>pg_catalog.pg_backend_pid()<br>AND a.state = 'active' AND<br>a.client_addr IS NOT NULL<br>ORDER BY query_runtime<br>DESC; |
| \view<br>wait_event<br>{unique_sql_id}<br>{start_time}<br>{end_time} | wait_event（带有三个参数）：查询指定慢SQL的历史等待事件。<br><br>unique_sql_id：需要查询的慢SQL的unique_sql_id。<br>start_time：开始时间。如<br>'2025-01-01 00:00'。<br>end_time：结束时间。如<br>'2025-01-01 00:00'。 | SELECT *,<br>pg_catalog.statement_detail_decode(details, 'plaintext', true) FROM<br>dbe_perf.statement_history<br>WHERE unique_query_id =<br>{unique_sql_id} AND<br>start_time > {start_time} AND<br>start_time < {end_time}<br>ORDER BY db_time DESC;                                                                                                                                                                                                                                                                               |
| \view<br>thr_wait_event                                              | thr_wait_event（不带参数）：查询实时整体等待事件。                                                                                                                                        | SELECT wait_status,<br>wait_event,<br>pg_catalog.COUNT(*) FROM<br>pg_catalog.pg_thread_wait_status GROUP BY 1, 2 ORDER BY 3 DESC;                                                                                                                                                                                                                                                                                                                                                                                                    |
| \view<br>thr_wait_event<br>{sessionid}                               | thr_wait_event（带有一个参数）：查询指定语句的等待事件。<br><br>sessionid：指定的sessionid。                                                                                                      | SELECT * FROM<br>pg_catalog.pg_thread_wait_status WHERE sessionid =<br>{sessionid};                                                                                                                                                                                                                                                                                                                                                                                                                                                  |



| 简易运维命令                                       | 参数解释                                                                                                    | 实际执行的SQL                                                                                                                                                   |
|----------------------------------------------|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view gs_asp<br>{start_time}<br>{end_time}   | gs_asp：查询历史指定时间段整体等待事件。<br>start_time：开始时间。如 '2025-01-01 00:00'。<br>end_time：结束时间。如 '2025-12-31 23:59'。 | SELECT wait_status, event, pg_catalog.COUNT(*) FROM pg_catalog.gs_asp WHERE sample_time BETWEEN {start_time} AND {end_time} GROUP BY 1, 2 ORDER BY 3 DESC; |
| \view pg_stats<br>{table_name}<br>{att_name} | pg_stats：查询指定表的统计信息分布。<br>table_name：表名。<br>att_name：列名。                                                | SELECT * FROM pg_catalog.pg_stats WHERE tablename = '{table_name}' AND attname = '{att_name}';                                                             |
| \view autovac<br>{table_name}                | autovac：查询统计信息更新阈值。<br>table_name：表名。                                                                   | SELECT * FROM pg_catalog.pg_autovac_statuses('{table_name}'::regclass);                                                                                    |
| \view thr_pool                               | thr_pool：查询线程池使用情况。                                                                                     | SELECT node_name, worker_info FROM dbperf.global_threadpool_status;                                                                                        |
| \view failed_proc<br>{proc_name}             | failed_proc：查询失效的视图或存储过程。<br>proc_name：失效的函数。                                                           | SELECT re.ev_class, cl.relname FROM pg_catalog.pg_rewrite re, pg_catalog.pg_class cl WHERE re.ev_action LIKE '%{proc_name}%' AND cl.oid = re.ev_class;     |

| 简易运维命令                   | 参数解释                                     | 实际执行的SQL                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------------|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view<br>failed_proc_pkg | failed_proc_pkg: 查询失效<br>存储过程对应的package。 | SELECT DISTINCT object_type,<br>pg_catalog.pg_get_userbyid(c<br>reator) AS owner, CASE<br>object_type WHEN 'P' THEN<br>( SELECT DISTINCT<br>pg_catalog.UPPER(pkgname)<br>FROM pg_catalog.gs_package<br>WHERE oid = ( SELECT<br>propackageid FROM<br>pg_catalog.pg_proc WHERE<br>oid = object_oid ) ) WHEN 'B'<br>THEN ( SELECT DISTINCT<br>pg_catalog.UPPER(pkgname)<br>FROM pg_catalog.gs_package<br>WHERE oid = object_oid )<br>WHEN 'S' THEN ( SELECT<br>DISTINCT<br>pg_catalog.UPPER(pkgname)<br>FROM pg_catalog.gs_package<br>WHERE oid = object_oid )<br>END package_name FROM<br>pg_catalog.pg_object WHERE<br>valid = 'f' AND object_type IN<br>( 'P', 'B', 'S' ); |
| \view<br>failed_index    | failed_index: 查询不可用<br>的索引。              | SELECT a.schemaname,<br>a.relname, a.indexrelname,<br>b.indisusable FROM<br>pg_catalog.pg_stat_user_inde<br>xes a, pg_catalog.pg_index b<br>WHERE a.indexrelid =<br>b.indexrelid AND<br>b.indisusable = 'f';                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| \view<br>table_size      | table_size: 查询当前库下<br>所有表的大小，按降序输<br>出。  | SELECT schemaname,<br>relname,<br>pg_catalog.pg_total_relation_<br>size(pg_catalog.CONCAT(sche<br>maname, '.', relname)) /<br>1024 / 1024 / 1024 "GB"<br>FROM<br>pg_catalog.pg_stat_user_table<br>s ORDER BY 3 DESC;                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

| 简易运维命令                                                   | 参数解释                                                                                                                                        | 实际执行的SQL                                                                                                                                                                                                                                             |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view<br>dead_tup<br>{top_n}                             | dead_tup: 查询死元组的数量。<br>top_n: 查询的行数。                                                                                                        | SELECT relname, n_live_tup, n_dead_tup, (n_dead_tup / (n_live_tup + 1)) AS rating, last_vacuum, last_autovacuum, last_analyze, last_autoanalyze FROM pg_catalog.pg_stat_user_tables ORDER BY rating DESC, n_dead_tup DESC LIMIT {top_n};             |
| \view<br>table_size<br>{table_name}                      | table_size: 查询指定表的大小。<br>table_name: 表名。                                                                                                    | SELECT * FROM pg_catalog.pg_size_pretty(pg_catalog.pg_total_relation_size('{table_name}'::regclass));                                                                                                                                                |
| \view login<br>{start_time}<br>{end_time}<br>{ip} {port} | login: 查询指定IP和端口的登录即退出信息。<br>start_time: 开始时间。如 '2024-01-01 00:00:00'。<br>end_time: 结束时间。如 '2025-12-31 23:59:59'。<br>ip: IP地址。<br>port: 端口。 | SELECT *, pg_catalog.SUBSTRING( client_conninfo::varchar, pg_catalog.POSITION(client_conninfo::varchar, '@'::varchar) + 1, 15 ) AS ip FROM pg_catalog.pg_query_audit({start_time}, {end_time}) WHERE ip = '{ip}' AND remote_port = '{port}';         |
| \view mem                                                | mem: 查询当前内存使用信息                                                                                                                             | SELECT * FROM pg_catalog.pg_total_memory_detail;                                                                                                                                                                                                     |
| \view thr_mem<br>{top_n}                                 | thr_mem ( 带有一个参数 ): 查询数据库所有线程的内存上下文使用情况。<br>top_n: 查询的行数。                                                                                   | SELECT contextname, pg_catalog.SUM(totalsize) / 1024 / 1024 "totalsize", pg_catalog.SUM(freesize) / 1024 / 1024 "freesize", pg_catalog.COUNT(*) "sum" FROM pg_catalog.gs_thread_memory_context GROUP BY contextname ORDER BY sum DESC LIMIT {top_n}; |
| \view thr_mem<br>{context_name}<br>{top_n}               | thr_mem ( 带有两个参数 ): 查询使用内存最多的线程的线程id。<br>context_name: 内存上下文的名称。<br>top_n: 查询的行数。                                                           | SELECT * FROM pg_catalog.pv_thread_memory_detail() WHERE contextname = '{context_name}' ORDER BY totalsize DESC LIMIT {top_n};                                                                                                                       |

| 简易运维命令                                          | 参数解释                                                                 | 实际执行的SQL                                                                                                                                                                                                                                                                             |
|-------------------------------------------------|----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view<br>sess_mem<br>{top_n}                    | sess_mem：查询数据库所有session的内存上下文使用情况。<br>top_n：查询的行数。                   | SELECT contextname,<br>pg_catalog.SUM(totalsize) /<br>1024 / 1024 "totalsize",<br>pg_catalog.SUM(freesize) /<br>1024 / 1024 "freesize",<br>pg_catalog.COUNT(*) "sum"<br>FROM<br>pg_catalog.gs_session_memory_context GROUP BY<br>contextname ORDER BY sum<br>DESC LIMIT {top_n};     |
| \view<br>thr_mem_detail {tid}<br>{context_name} | thr_mem_detail：查询指定线程的内存使用情况。<br>tid：线程id。<br>context_name：内存上下文的名称。 | SELECT * FROM<br>pg_catalog.gs_get_thread_memory_detail({tid},<br>'{context_name}');                                                                                                                                                                                                 |
| \view<br>sess_mem_tp<br>{context_name}          | sess_mem_tp：查询指定内存上下文的内存使用情况。<br>context_name：内存上下文的名称。              | SELECT * FROM<br>pg_catalog.gs_get_session_memory_detail('{context_name}');                                                                                                                                                                                                          |
| \view<br>shared_mem<br>{top_n}                  | shared_mem：查询数据库全局共享内存上下文的使用情况。<br>top_n：查询的行数。                      | SELECT contextname,<br>pg_catalog.SUM(totalsize) /<br>1024 / 1024 "totalsize MB",<br>pg_catalog.SUM(freesize) /<br>1024 / 1024 "freesize MB",<br>pg_catalog.COUNT(*) "sum"<br>FROM<br>pg_catalog.gs_shared_memory_detail GROUP BY<br>contextname ORDER BY sum<br>DESC LIMIT {top_n}; |
| \view<br>sess_mem_detail {top_n}                | sess_mem_detail：查询session级内存上下文的使用情况。<br>top_n：查询的行数。                | SELECT contextname,<br>pg_catalog.SUM(totalsize) /<br>1024 / 1024 "totalsize MB",<br>pg_catalog.COUNT(1)<br>"count" FROM<br>pg_catalog.pv_session_memory_detail() GROUP BY<br>contextname ORDER BY 2<br>DESC LIMIT {top_n};                                                          |

| 简易运维命令                                                           | 参数解释                                                                                                     | 实际执行的SQL                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \view<br>sess_mem_detail_tp {top_n}                              | sess_mem_detail_tp: 查询session级内存上下文的使用情况（仅在线程池模式开启时生效）。<br>top_n: 查询的行数。                                 | SELECT sessid,<br>pg_catalog.SUM(totalsize) /<br>1024 / 1024 "totalsize MB",<br>pg_catalog.COUNT(1)<br>"count" FROM<br>pg_catalog.gs_session_memory_context GROUP BY sessid<br>ORDER BY 2 DESC LIMIT<br>{top_n};                                                       |
| \view<br>thr_mem_pv {top_n}                                      | thr_mem_pv: 查询线程级内存上下文的使用情况。<br>top_n: 查询的行数。                                                            | SELECT contextname,<br>pg_catalog.SUM(totalsize) /<br>1024 / 1024 "totalsize",<br>pg_catalog.SUM(freesize) /<br>1024 / 1024 "freesize",<br>pg_catalog.COUNT(1)<br>"count" FROM<br>pg_catalog.pv_thread_memory_detail() GROUP BY 1<br>ORDER BY 2 DESC LIMIT<br>{top_n}; |
| \view<br>total_mem {top_n}                                       | total_mem: 查询当前数据库节点的内存使用情况。<br>top_n: 查询的行数。                                                            | SELECT * FROM<br>pg_catalog.gs_total_memory_detail LIMIT {top_n};                                                                                                                                                                                                      |
| \view avg_cpu {top_n}                                            | avg_cpu: 查询SQL语句的平均CPU时间，按降序输出。<br>top_n: 查询的行数。                                                         | SELECT unique_sql_id,<br>cpu_time / (n_calls)<br>"average_cpu_time" FROM<br>dbe_perf.statement WHERE<br>n_calls != 0 ORDER BY 2<br>DESC LIMIT {top_n};                                                                                                                 |
| \om<br>create_abort_sql_patch<br>{patch_name}<br>{unique_sql_id} | create_abort_sql_patch: 创建避险SQL PATCH。<br>patch_name: SQL PATCH名。<br>unique_sql_id: 指定SQL的unique_sql_id。 | SELECT * FROM<br>dbe_sql_util.create_abort_sql_patch('{patch_name}',<br>{unique_sql_id});                                                                                                                                                                              |

| 简易运维命令                                                               | 参数解释                                                                                                                           | 实际执行的SQL                                                                                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>\om kill_sess_thr {username} {database} {context_name} }</pre>  | <p>kill_sess_thr: 根据线程内存占用，查杀内存占用高的SQL语句。</p> <p>username: 用户名。</p> <p>database: 数据库名。</p> <p>context_name: 内存上下文名称。</p>       | <pre>SELECT pg_catalog.pg_terminate_session(pid, sessionid), pid, sessionid FROM pg_catalog.pg_stat_activity WHERE username = '{username}' AND datname != 'postgres' AND datname = '{database}' AND pid IN ( SELECT tid FROM pg_catalog.gs_thread_memory_context WHERE contextname = '{context_name}' ) AND state = 'idle';</pre> |
| <pre>\om kill_sess_sess {username} {database} {context_name} }</pre> | <p>kill_sess_sess: 根据session内存占用，查杀内存占用高的SQL语句。</p> <p>username: 用户名。</p> <p>database: 数据库名。</p> <p>context_name: 内存上下文名称。</p> | <pre>SELECT pg_catalog.pg_terminate_session(pid, sessionid), pid, sessionid FROM pg_catalog.pg_stat_activity WHERE username = '{username}' AND datname != 'postgres' AND datname = '{database}' AND pid IN ( SELECT threadid FROM pg_catalog.gs_session_memory_context WHERE contextname = '{context_name}' );</pre>              |

- 应急通道视图 DBE\_PERF.EMERGENCY\_OPERATION\_RECORDS  
该视图用于查看各CN/DN节点对Unix Domain Socket运维通道的使用情况

表 7-52 DBE\_PERF.EMERGENCY\_OPERATION\_RECORDS 字段

| 参数          | 类型         | 描述                   |
|-------------|------------|----------------------|
| record_time | timestampz | 使用运维通道的时间。           |
| action      | text       | 使用运维通道执行的操作。         |
| source      | text       | 接入运维通道的来源，取值为gsq和cm。 |
| detail      | text       | 执行操作的具体结果。           |

## 7.29.3 操作指导

### 前提条件

应急通道需要打开GUC参数enable\_emergency\_access，另外生成mem\_log的运维能力依赖于wlm\_monitor线程及内存管理功能，需要打开GUC参数use\_workload\_manager和enable\_memory\_limit，参数均默认打开。

GUC参数enable\_emergency\_access具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

GUC参数use\_workload\_manager具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

GUC参数enable\_memory\_limit具体描述请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。

### 示例

#### 示例一

执行Unix Domain Socket运维通道的运维命令。

以下为如何通过Unix Domain Socket运维通道查生成mem\_log的简单示例：

- 步骤1** 登录GaussDB节点所在服务器，执行以下gsq命令，确认是否生成内存概况以及内存上下文等使用记录。

```
gsq -O "{port} mem_log"
```

其中，{port}为数据库端口。当显示true时，则表示已生成具体的详情信息；如果执行失败，则显示失败原因。

-- 示例中19995为数据库监听端口。

```
gsq -O "19995 mem_log"
true
```

- 步骤2** 查看“\$GAUSSLOG/mem\_log/{nodename}”目录下生成的mem\_log文件内容，其中{nodename}为数据库节点名称。

-- 示例中datanode为数据库节点名称。

```
ls -l $GAUSSLOG/mem_log/datanode
total 8
-rw-----. 1 omm omm 7543 May 28 15:15 mem_log-2025-05-28_151518.log
```

- 步骤3** 执行以下命令查看文件内容。

```
cat $GAUSSLOG/mem_log/datanode/mem_log-time.log
```

其中，mem\_log-time.log为步骤2中“\$GAUSSLOG/mem\_log/datanode”路径下生成的文件（文件名后缀为.log）。

----结束

#### 示例二

gsq执行简易运维元命令\view mem。

--连接数据库。（根据实际需要替换数据库名字和端口号）

```
gsq -d postgres -p 8000
gsq ((GaussDB Kernel XXX.X.XXX build f521c606) compiled at 2025-08-30 11:51:09 last mr 24890 release)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
gaussdb=# \view mem
nodename | memorytype | memorybytes
```

|           |                                  |       |
|-----------|----------------------------------|-------|
| cn_5001   | max_process_memory               | 35840 |
| cn_5001   | process_used_memory              | 12417 |
| cn_5001   | max_dynamic_memory               | 28532 |
| cn_5001   | dynamic_used_memory              | 3448  |
| cn_5001   | dynamic_peak_memory              | 8966  |
| cn_5001   | dynamic_used_shrctx              | 2081  |
| cn_5001   | dynamic_peak_shrctx              | 2815  |
| cn_5001   | max_backend_memory               | 1544  |
| cn_5001   | backend_used_memory              | 1     |
| cn_5001   | high_backend_thread_memory       | 56    |
| cn_5001   | high_backend_thread_max_memory   | 921   |
| cn_5001   | medium_backend_thread_memory     | 358   |
| cn_5001   | medium_backend_thread_max_memory | 3072  |
| cn_5001   | low_backend_thread_memory        | 0     |
| cn_5001   | low_backend_thread_max_memory    | 1536  |
| cn_5001   | max_shared_memory                | 5747  |
| cn_5001   | shared_used_memory               | 4866  |
| cn_5001   | max_cstore_memory                | 16    |
| cn_5001   | cstore_used_memory               | 0     |
| cn_5001   | max_sctpcomm_memory              | 0     |
| cn_5001   | sctpcomm_used_memory             | 0     |
| cn_5001   | sctpcomm_peak_memory             | 0     |
| cn_5001   | other_used_memory                | 3998  |
| cn_5001   | gpu_max_dynamic_memory           | 0     |
| cn_5001   | gpu_dynamic_used_memory          | 0     |
| cn_5001   | gpu_dynamic_peak_memory          | 0     |
| cn_5001   | pooler_conn_memory               | 1     |
| cn_5001   | pooler_freeconn_memory           | 0     |
| cn_5001   | storage_compress_memory          | 0     |
| cn_5001   | udf_reserved_memory              | 0     |
| cn_5001   | llvm_used_memory                 | 9     |
| cn_5001   | max_htap_memory                  | 0     |
| cn_5001   | htap_used_memory                 | 0     |
| (33 rows) |                                  |       |

## 7.29.4 常见问题

1. **问题现象：**执行gsq -O "{port} mem\_log"后没有生成mem\_log文件。

**解决办法：**生成mem\_log的运维能力依赖于wlm\_monitor线程，及内存管理功能，需要打开use\_workload\_manager及enable\_memory\_limit参数，查看use\_workload\_manager及enable\_memory\_limit参数是否已经打开。

```
gaussdb=# SHOW use_workload_manager;
use_workload_manager

on
(1 row)

gaussdb=# SHOW enable_memory_limit;
enable_memory_limit

on
(1 row)
```

use\_workload\_manager参数设置方法请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节，enable\_memory\_limit参数设置方法请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

2. **问题现象：**非数据库节点无法执行应急通道的运维命令。

**解决办法：**本功能只支持本地连接，需要在数据库节点进行使用。

## 7.30 CM 仲裁及故障检测原理



## 7.30.1 特性概述

CM模块分为CMServer和CMAgent两个模块。

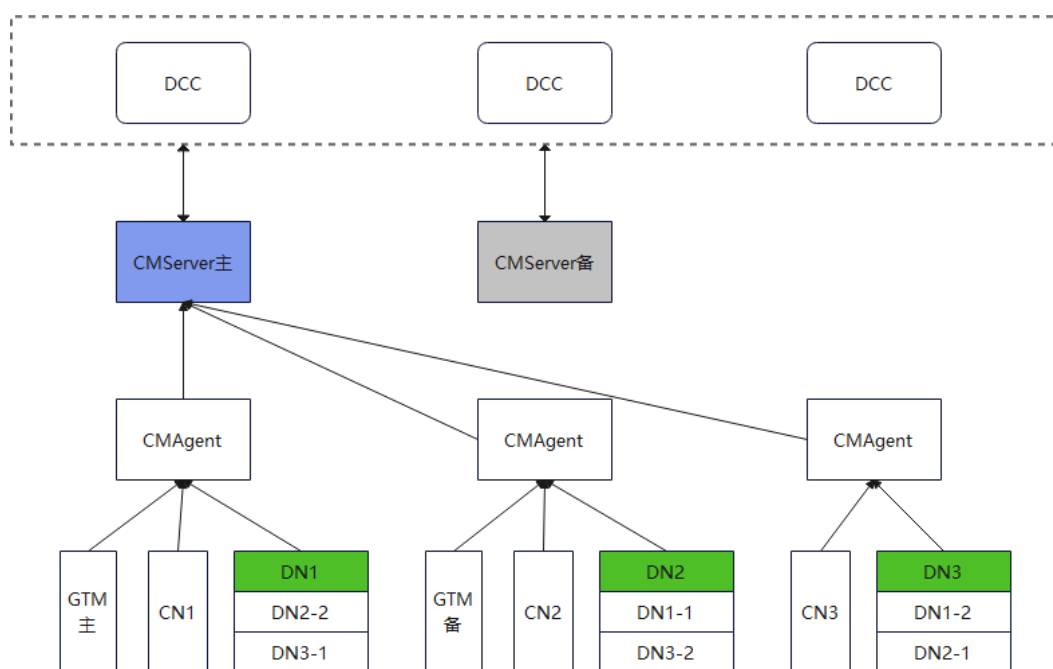
CMServer是整个集群的大脑，负责集群中各GTM、CN、DN实例状态的监控，以及每一对GTM，DN主备HA角色的仲裁，根据CMAgent上报的各实例节点状态信息来仲裁实例是否需要状态变更，以及需要哪种变更。正常场景下，集群需要有且只有一个CMServer主进行相关仲裁动作，CMServer会定期向ETCD写心跳数据，如果集群中CMServer主发生故障，需要通过ETCD仲裁选主。

CMAgent的主要功能包括启动本节点所有实例、停止本节点所有实例、监控上报本节点所有实例状态给CMServer以及执行CMServer下发的仲裁命令。

CMAgent是多线程监控组件，线程结构如图7-58所示。线程包括：

1. 主线程负责上报节点实例状态，以及执行CMServer下发的命令。
2. 启停线程负责启动和停止节点实例。
3. 与GTM/DN/CN连接线程负责查询实例状态。

图 7-58 CMAgent 线程结构



## 7.30.2 技术原理

### 7.30.2.1 仲裁原理

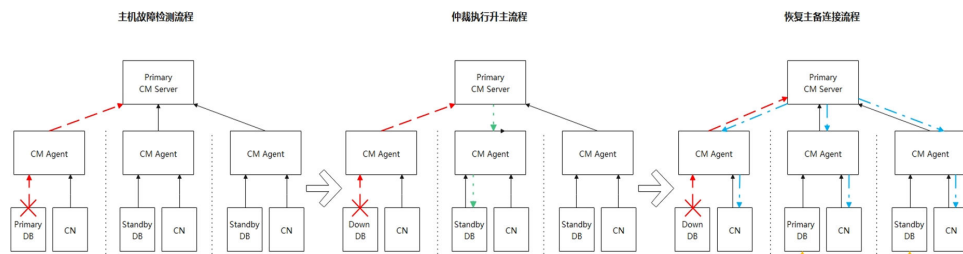
#### 7.30.2.1.1 DN 仲裁

CM支持的DN仲裁模式主要分为Quorum模式和DCF模式，默认模式为Quorum模式。

Quorum模式DN仲裁选主流程，如下图所示：

1. DN主实例故障（心跳超时6s/进程Down/磁盘故障/端口故障/僵死）。
2. CM检测到无主，即开始给正常节点发送lock1命令尝试锁住备DN。如果Redo回放完并且主备通信断开，则lock1成功，DN会处于prohibit\_connection状态，此时DN不再接受新日志且本地日志已回放完，CM可以获取到DN的本地日志最大值。
3. 从锁住的多数派备DN中，选择本地日志最大的备DN，并通知该备DN升主。
4. CM通知CN更新pgxc\_node，pgxc\_node中包含着主DN的路由信息，CN通过pgxc\_node来连接DN主。
5. CM发起lock2命令给备DN，lock2命令包含新主DN的路由信息，备DN可以连接新主。
6. CM发起unlock解锁，让主备DN恢复默认的自由连接状态。
7. 更新ETCD中的主备关系，1是Primary，2是Standby。
8. 服务恢复。

图 7-59 Quorum 模式 DN 仲裁选主流程



多AZ场景下，DN主故障处理原则：

DN主故障，依次优先选择用户初始指定、数据最新、原主同AZ备、AZ优先级最大（安装时配置）的DN备升主。主恢复，其角色降备，不自动回切为主，可以手动回切。如果原主的日志能够追上新主，则进行日志追赶；如果追不上，则在原主上进行数据全量重建。

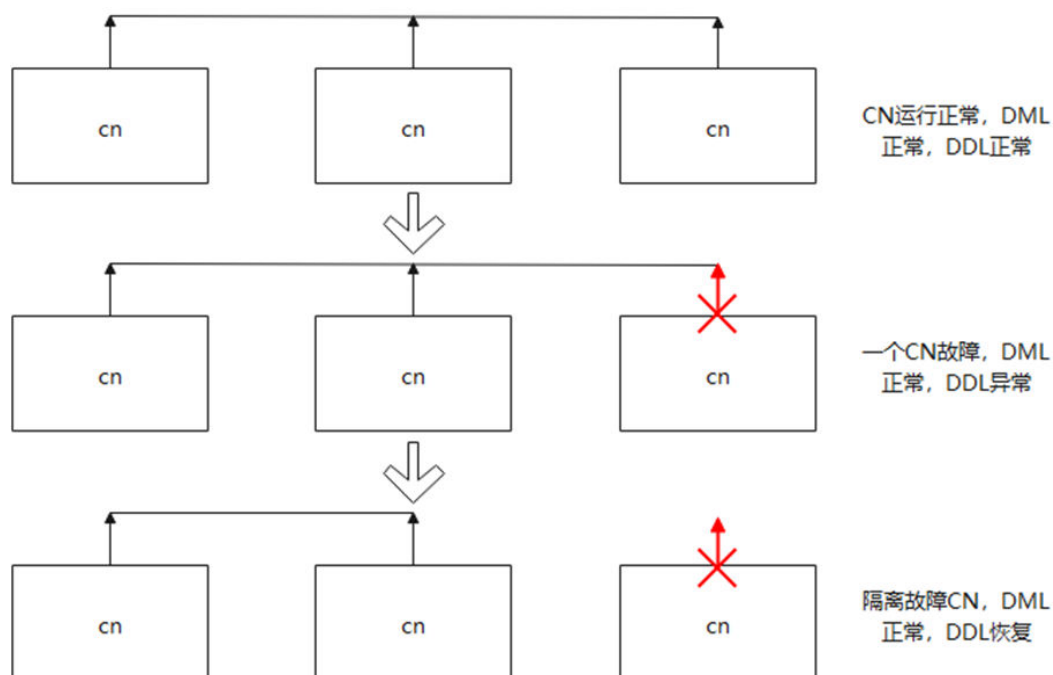
### 7.30.2.1.2 CN 仲裁

CN没有主备之分，同时CN的DDL操作为所有CN强一致性同步操作，CN故障后，DDL操作会受到影响。当CN故障超过25s一直没有恢复的话，CM认为此CN无法提供服务，进入CN的剔除流程。CN剔除流程如下图所示。

CN剔除：

1. CM检测到CN连续故障（进程Down/僵死/多次重启/磁盘故障/端口冲突/网卡故障/CN连不上所有主DN）时长超过25s，CMS会发起CN剔除的操作，CN处于Deleting状态，更新ETCD中值为8。
2. CM更新其他节点的pgxc\_node系统表，CN处于Deleted状态，更新ETCD中值为7。
3. 数据库状态恢复，CN可以执行DDL。

图 7-60 CN 剔除流程



CN 加回流程是反向的，满足条件即可修复：

1. CN 故障恢复，满足加回条件。
2. build 同步 CN 日志，更新 pgxc\_node。
3. 将 CN 状态置为 Normal，更新 ETCD 中值为 4。

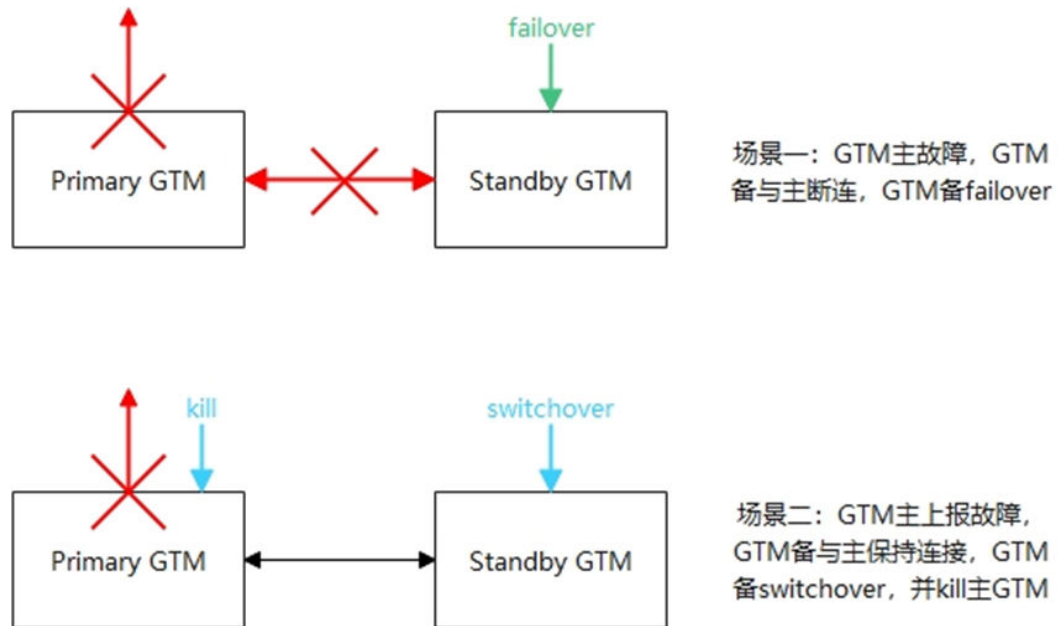
### 7.30.2.1.3 GTM 仲裁

GTM 名为全局事务管理器，系统中只有一个，采用主备方式，主机故障后，备机可以接管服务。主 GTM 不与备 GTM 进行数据同步，备 GTM 仅尝试与主 GTM 进行连接状态校验，GTM 主备中需要同步的数据，都是备份在 ETCD 上。GTM 仲裁选主流程如下图所示。

GTM 仲裁选主：

1. GTM 主故障（进程 Down/磁盘故障/端口冲突/网卡故障）。
2. 判断 GTM 备与主的连接状态。
3. 若 GTM 备与主连接断开，直接将上报正常状态的备节点 failover 升主。
4. 若 GTM 备与主保持连接未断连，触发 switchover 主备切换，并终止原主 GTM 进程。
5. 切换完成后，通知 CN 更新 GTM 连接。

图 7-61 GTM 仲裁选主流程

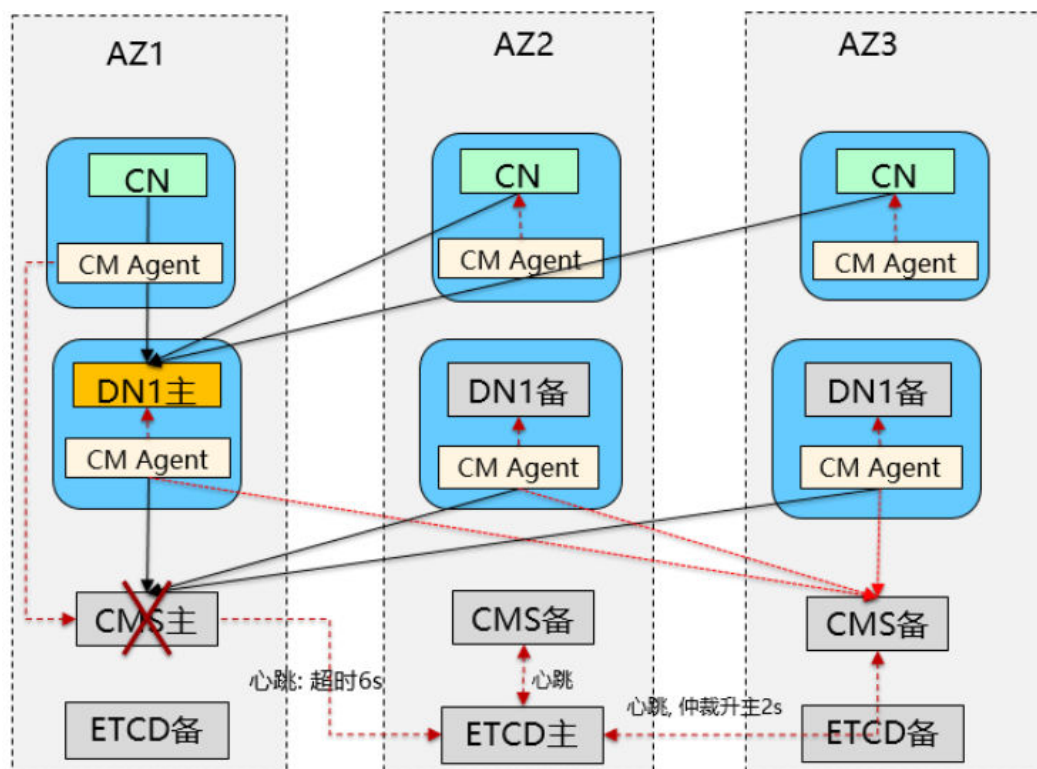


#### 7.30.2.1.4 CMServer 仲裁

CMServer负责整个集群中实例的仲裁及命令的下发，相当于集群的大脑。当实例出现故障时，CMServer会仲裁实例以试图使得集群恢复可用状态，但当CMServer自身面临故障时，并没有一个更上层的组件来进行仲裁处理，CMServer依靠的是自身的自仲裁机制保证集群的可用性。

CMServer自仲裁选主：

1. CMServer会持续往ETCD中写入心跳。
2. CMServer主异常后，不再向ETCD中更新心跳，CMServer备发现CMServer主心跳超时（6s）后，进入自选主流程。
3. CMServer备满足心跳连续，且存在CMAgent预连接，就会自行升主并将本CMServer的升主信息写入ETCD。其余CMServer看到ETCD中的升主信息，就退出选主流程。
4. CMServer选主成功。



### 7.30.2.1.5 ETCD 仲裁

ETCD是一种开源分布式键值库，用于共享配置和服务发现，负责保存集群的拓扑信息。它提供存储以及获取数据的接口，并通过协议保证ETCD集群中的多个节点数据的强一致性。ETCD可用于存储元信息以及共享配置信息，同时提供监听机制，允许客户端监听特定key或某些key的变更，实现变更的实时监听与推送。此外，ETCD提供key的过期以及续约机制，客户端通过定时刷新来实现续约，适用于监控与服务注册发现。ETCD 还提供原子的CAS（Compare-and-Swap）和 CAD（Compare-and-Delete）支持，支持分布式锁以及Leader选举。

ETCD作用:

1. 全局事务ID、Sequence依赖ETCD存储。
2. CMServer自身仲裁和CMServer对其他数据库组件的仲裁依赖ETCD。
3. 存储各组件的主备信息依赖ETCD。

ETCD仲裁选主:

1. ETCD自选主是基于Raft分布式一致性算法，此算法可确保集群中所有节点之间的数据存储一致性。
2. ETCD如果主节点（Leader）故障，备节点（Follower）发起投票，获得多数投票的（Candidate）升主。然后通知CMServer主已经进行升主操作，CMServer主和GTM都会去找新的ETCD主进行数据同步。

### 7.30.2.2 故障检测原理

#### 7.30.2.2.1 磁盘检测

- 规格:

- 只检测DN、CN、GTM、CM（CMA、CMS）的数据盘、日志盘以及DN的表空间所在盘。
- 检测规则：
  - 检测通过读写临时文件实现，针对文件会有四个操作：open（打开文件），write（写文件），read（读文件），unlink（删除文件）。其中open、write、read任意一个操作失败或者执行超时，就会判定磁盘发生故障或处于hang状态，并触发仲裁。相反，unlink操作失败或执行超时，则不会作为触发仲裁的条件。
  - 非DIRECT读写：每隔1秒采用非DIRECT的方式向磁盘中写入1B的数据，如果读写失败，通过错误码判断失败的原因，其中如果是EIO（IO错误）、EROFS（文件系统只读）、ENOSPC（磁盘空间不足）、EACCES（权限不足）这些原因，会判定故障，其他类型错误暂不处理。
  - DIRECT读写：默认每隔5秒（可配）采用DIRECT的方式向磁盘中写入512B（如果被检测磁盘为条带化，则写入256KB）的数据，如果写入操作阻塞超过20s（如果无法判断被检测磁盘是否为条带化，则超时时间为60s，写入数据的大小为256KB）没有返回，则判定磁盘处于I/O hang。
- 高负载下磁盘hang
  - 简介：CM定时从数据库获取物理I/O指标（I/O文件读写访问次数、I/O文件读写流量）。CM检测到磁盘hang后，先判断I/O指标是否达到高负载标准（I/O文件读写访问次数10次/秒或I/O文件读写流量256kB/秒），如果达到高负载标准，则不继续进行磁盘hang的处理动作。
  - 约束：
    - 高负载场景定义：达到I/O文件读写访问次数10次/秒 或 I/O文件读写流量256kB/秒，认为处于高负载。
    - 高负载场景，不进行磁盘hang处理。非高负载场景，进行磁盘hang处理，RTO满足60s。
    - 获取不到I/O指标场景，则进行磁盘hang的处理，RTO满足60s。
- 说明：
  - 写入的数据大小可配置，配置的写入大小需要根据磁盘的对齐规则进行对齐，如果配置时没对齐，cm会自动向上对齐。在磁盘LVM条带化场景，为保障写入检测能覆盖到所有磁盘，建议写入大小根据磁盘数量和数据分块大小动态调整。
  - I/O hang超时时间可配置，需要根据磁盘性能进行配置，性能差的磁盘，超时时间可以适当配置长一些，进而减少业务I/O对磁盘检测的影响，相应的，hang场景RTO将会变长。配置成0则关闭hang检测。
  - 写入大小过大、检测周期过短时，可能对磁盘IO有一定的争抢，进而影响数据库性能。

#### 7.30.2.2.2 DN 僵死检测

当DN进程处于D、T、Z或者连接超时等僵死状态时，DN无法及时处理业务请求，导致业务中断，影响业务正常功能。CM支持根据配置值定时检测DN是否处于僵死状态，若DN连续处于僵死状态的次数达到设定值，将自动执行重启恢复操作。

#### 7.30.2.2.3 急救室能力

急救室是一个高度抽象的框架模型，由于故障存在假阳性和假阴性情况，因此需要框架更加的灵活，以应对不同场景和需求下的高可用。当前具备线程心跳自检、检测磁

盘I/O hang（由于磁盘故障，导致无法进行数据写入和读取等）问题并自动隔离主DN的能力、系统资源监控能力、网络探测能力和支持异步日志功能。

#### 7.30.2.2.4 硬件故障处理

硬件故障处理属于端到端全栈高可用能力。

### 7.30.3 操作指导

仲裁和故障检测均为自动进行。

### 7.30.4 常见问题

Q：CN剔除故障恢复后未加回。

A：对应节点CMAgent日志搜索autorepair，查看不能加回原因。故障恢复后，CM如果30分钟内都没有加回CN，便不会再尝试加回。如需加回，可通过节点替换修复。

Q：多数派故障后集群无主。

A：请参见《最佳实践》中“多数派故障最佳实践”章节。

Q：DN发生主备倒换。

A：在原主节点CMAgent日志中搜索killing xxx by force、inst(yyy) data(yyy) killed，或在CMServer主日志中搜索KEY\_EVENT\_RESTART；若存在相关记录，则表明是CM仲裁触发的主备倒换。随后，需在进程终止时间点附近检索是否存在io hang或phony dead等关键字，以确认主备倒换的具体原因。若未找到上述CM仲裁相关记录，则说明非CM仲裁导致，应进一步排查DN进程退出的原因。

Q：故障后主备倒换RTO超时。

A：在CMServer主节点的key\_event\_xxx日志中找KEY\_EVENT\_FAILOVER，KEY\_EVENT\_FAILOVER日志之前的时间为CM仲裁选主时间，之后的时间为DN升主时间。

1. 如果是仲裁时间过长，查看对应节点CMAgent和CMServer主日志，确定CM感知到故障的时间及决策的时间，排查是否出现叠加故障，如ETCD主、CMS主和DN主一起故障。
2. 如果DN升主时间过长，查看failover升主节点对应的DN日志，排查升主慢的原因。

## 7.31 集群内与集群间倒换原理

### 7.31.1 集群内倒换

### 7.31.1.1 特性概述

#### 功能定义

实现集群内主DN与备DN角色的计划内切换（switchover），保障业务连续性。

#### 用户场景

在安装集群、升级、扩容等相关流程中，均包含集群内switchover流程。

#### 规格约束

- 主DN无业务，备DN无日志积压的场景下（备DN RTO<10秒），switchover超时时间为120秒。
- 在使用switchover前，应提前检查升主备DN的RTO值，如果RTO较大，但客观需要执行switchover，需手动指定超时时间（默认120秒）。
- 级联备DN不能直接转换为主DN，需要先通过switchover或failover成为备DN，再成为主DN。

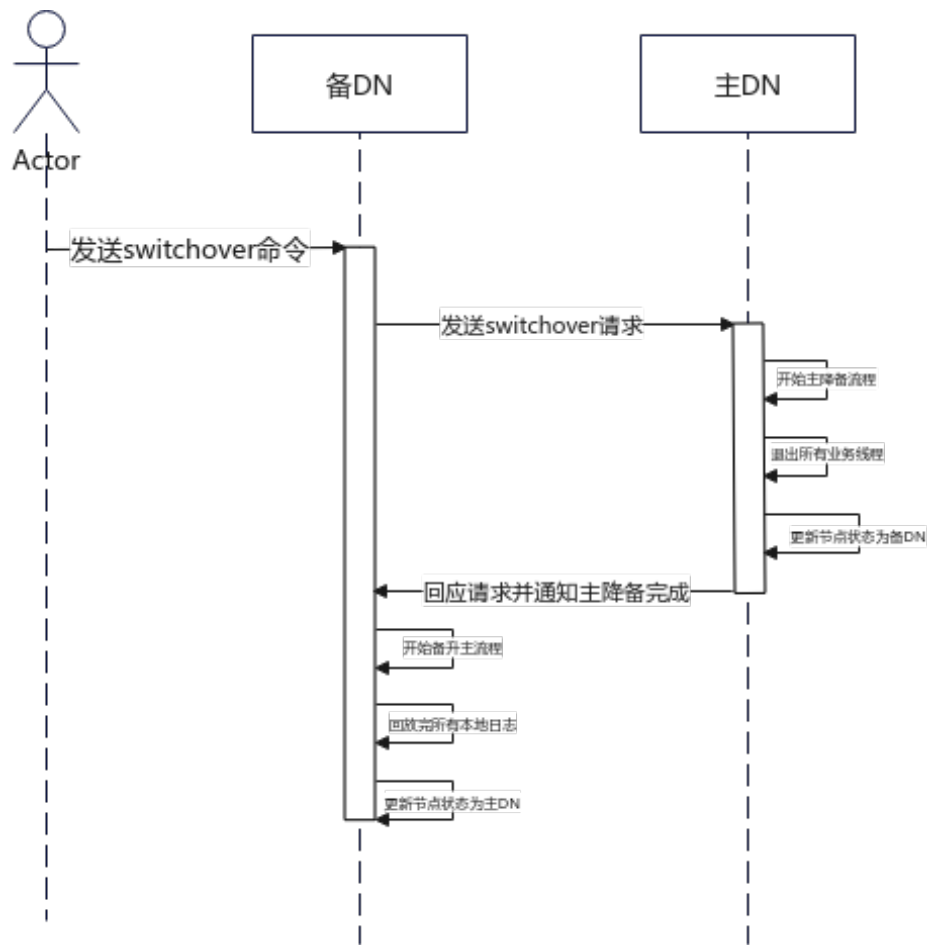
### 7.31.1.2 技术原理

switchover执行流程分为以下核心阶段：

1. 使用cm\_ctl下发switchover命令后，备DN会向主DN发送降备请求。
2. 主DN收到降备请求后，退出所有业务线程并进行销毁共享内存等相关操作。
3. 主DN更新节点状态为备DN，通知备DN主降备已完成。
4. 备DN收到主降备完成消息后，完成本地日志的回放并更新节点状态为主DN。
5. 完成switchover流程。



图 7-62 集群内倒换原理



7.31.1.3 操作指导

前提条件

在进行switchover操作之前，需要先确保数据库状态正常、所有业务结束，并使用pg\_get\_senders\_catchup\_time视图查询无主备追赶后，再进行switchover操作。

注意事项

switchover前主动执行一次CHECKPOINT，保证脏页刷盘完成，避免switchover执行超时。

示例

- 指定AZ的数据库主备倒换：  
`cm_ctl switchover -z AZ1`
- 指定实例的数据库主备倒换：  
`cm_ctl switchover -n 1 -D /data1/omm/cluster/data/datanode1p`
- 将所有GTM、DN实例统一从主切换到备，并选择相应分片的其他实例升主：  
`cm_ctl switchover -A`
- 重置集群主备关系为初始配置状态：  
`cm_ctl switchover -a`

- DCC模式下指定cm\_server主备切换：  
`cm_ctl switchover -n 1 -D /data1/omm/cluster/cm/cm_server`

#### 7.31.1.4 常见问题

- 主降备超时导致switchover超时：  
主降备超时主要是因为后台线程或者关键线程的退出超时。需要查看日志定位（\$GAUSSLOG/gs\_log/dn\_xxxx/gaussdb-\*.log）。
- 备升主超时导致switchover超时：
  - 观测是否有Xlog积压。
  - 查询日志在升主阶段是否有错误信息，判断是否是异常导致升主流程中断。

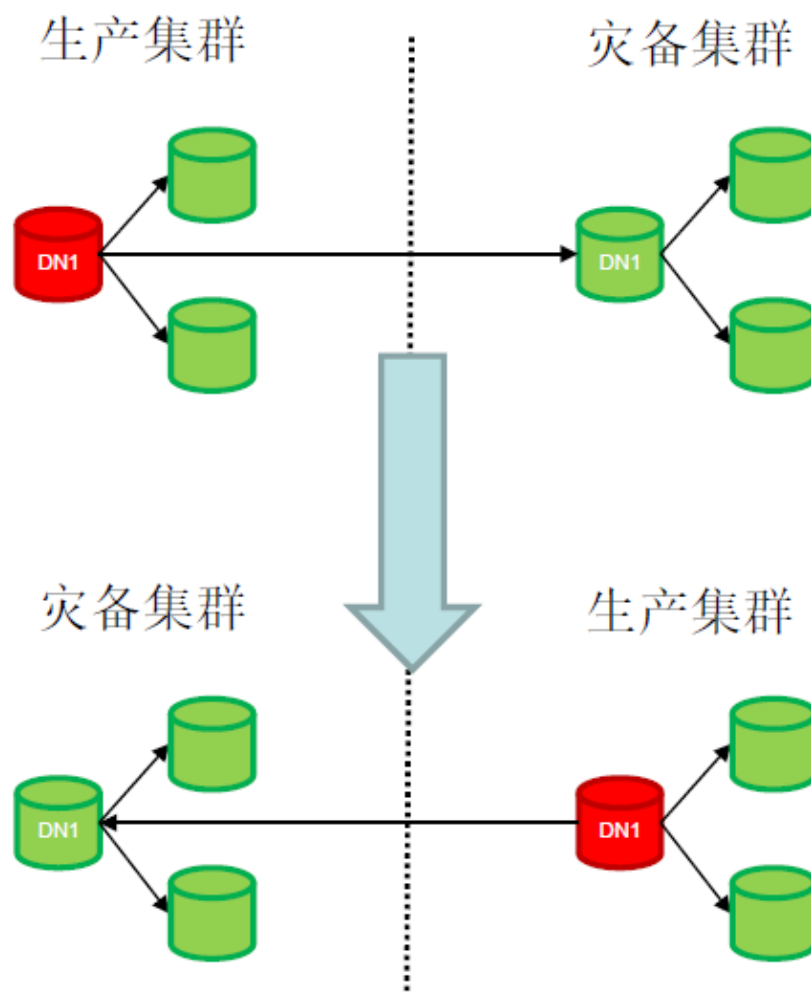
### 7.31.2 集群间倒换

#### 7.31.2.1 特性概述

##### 功能定义

在确保数据零丢失的前提下，通过计划内切换主数据库实例和灾备数据库实例的角色（Switchover），实现在数据库维护升级与容灾演练等场景中对系统可用性的保证。倒换原理如下图所示。

图 7-63 集群间倒换原理



## 用户场景

银行核心交易系统需定期验证容灾能力，通过主动switchover将主集群切换至同城或者异地的灾备集群，验证RPO=0（零数据丢失）且RTO（业务恢复时间）满足规格。

## 规格约束

规格约束根据容灾形态的选择有部分差异：

- 基于流式复制的异地容灾解决方案。
  - 基于paxos协议：
    - 在没有日志积压（即容灾监控指标RPO≤10秒，RTO≤1秒），sequence个数少于10，主备集群实例均处于Normal状态且所有组件（CN、DN、ETCD、GTM、CM\_Agent、CM\_Server）状态正常，且打开快速倒换模式的情况下，启动容灾倒换可在内核时间30秒内完成倒换，特别地，当主备集群异构且CN数量不一致，倒换完成之后，CN数量多的集群会以Degraded态退出。

- 在没有日志积压（即容灾监控指标RPO≤10秒，RTO≤1秒），sequence个数少于10，主备集群实例部分组件（CN、DN、ETCD、GTM、CM\_Agent、CM\_Server）状态异常且仍满足少数派故障的情况下，且打开快速倒换模式的情况下，启动容灾倒换可在内核时间90秒内完成倒换。
- 在有日志积压（即容灾监控指标RPO>10秒，RTO>1秒），日志传输、日志回放会增加容灾倒换的耗时，无法实现规格内的倒换。
- 在ARM平台下，如果PAGESIZE（查询方式：getconf PAGESIZE）不满足64KB规格，无法实现规格内的倒换。
- 基于quorum协议：
  - 在没有日志积压（即容灾监控指标RPO≤10秒，RTO≤1秒），sequence个数少于10，主备集群实例均处于Normal状态且所有组件（CN、DN、ETCD、GTM、CM\_Agent、CM\_Server）状态正常，且打开快速倒换模式的情况下，启动容灾倒换可在内核时间20秒内完成倒换，特别地，当主备集群异构且CN数量不一致，倒换完成之后，CN数量多的集群会以Degraded态退出。
  - 在没有日志积压（即容灾监控指标RPO≤10秒，RTO≤1秒），sequence个数少于10，主备集群实例部分组件（CN、DN、ETCD、GTM、CM\_Agent、CM\_Server）状态异常且仍满足少数派故障的情况下，且打开快速倒换模式的情况下，启动容灾倒换可在内核时间90秒内完成倒换。
  - 在有日志积压（即容灾监控指标RPO>10秒，RTO>1秒），日志传输、日志回放会增加容灾倒换的耗时，无法实现规格内的倒换。
  - 在ARM平台下，如果PAGESIZE（查询方式：getconf PAGESIZE）不满足64KB规格，无法实现规格内的倒换。
- 基于同城双中心高可用方案。
  - 在备集群回放能追上主集群且备集群网络正常的场景下，若集群整体运行正常，内核执行failover的平均RTO小于30秒；若集群出现少数派故障，内核执行failover的平均RTO小于90秒（具体时间受集群待回放的日志量影响，当前数据为3CN3DN3副本配置的情况）。
  - 在备集群回放能追上主集群且备集群网络正常的场景下，若不存在日志积压（即容灾监控指标RTO ≤ 1秒）、Sequence数量少于10，且集群整体运行正常，内核执行switchover的平均RTO小于1分钟；若集群出现少数派故障，内核执行switchover的平均RTO小于2分钟（具体时间受集群待回放的日志量影响，当前数据为3CN3DN3副本配置的情况）。
  - 如果出现超规格场景，比如短暂大业务压力场景，RTO将不可避免地升高。实测表明，当前版本若要满足上述failover和switchover的RTO约束，需保证主集群单分片日志产生速率控制在特定范围内，典型配置下的日志产生速率要求如表7-53所示。

表 7-53 典型配置下日志产生速率

| 典型配置              | 支持主集群单分片日志产生速率 |
|-------------------|----------------|
| 96U/768G/SATA SSD | ≤7.5MB/s       |

集群间倒换RTO规格演进请参见表7-54。

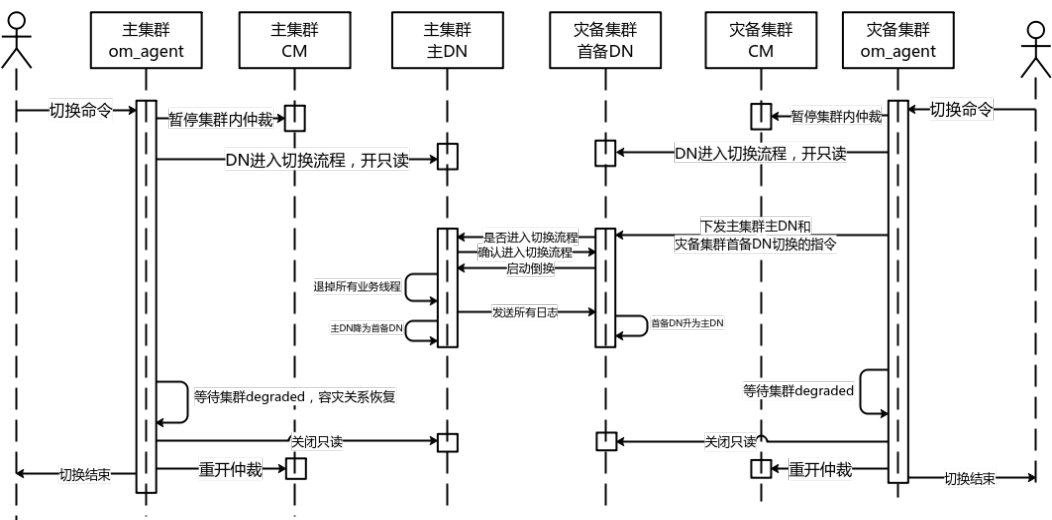
表 7-54 集群间倒换 RTO 规格演进

| 容灾形态            | 部署方式 | R2C 10                   | 503. 1                    | 503. 1 SPC 200 0c | 503. 2 | 505 | 505. 1 | 505. 2 | 505. 2.1 SPC 010 0 | 50 6.0 | 506 .1 及以后 |
|-----------------|------|--------------------------|---------------------------|-------------------|--------|-----|--------|--------|--------------------|--------|------------|
| 基于流式复制的异地容灾解决方案 | 集中式  | 5分钟 (SPC 660 之后版本优化到50秒) | 5分钟 (SP C17 00之后版本优化到50秒) | 50秒               | 50秒    | 50秒 | 50秒    | 50秒    | 50秒                | 10秒    | 10秒        |
|                 | 分布式  | 10分钟                     | 10分钟                      | 10分钟              | 10分钟   | 60秒 | 60秒    | 30秒    | 30秒                | 20秒    | 20秒        |
| 基于同城双中心高可用方案    | 集中式  | 2分钟                      | 2分钟                       | 60秒               | 2分钟    | 2分钟 | 2分钟    | 2分钟    | 60秒                | 60秒    | 60秒        |
|                 | 分布式  | 不支持                      | 不支持                       | 不支持               | 不支持    | 3分钟 | 2分钟    | 2分钟    | 2分钟                | 60秒    | 60秒        |

7.31.2.2 技术原理

1. 用户在管控平台下发集群倒换命令后，管控会对主备集群同时发送倒换请求。
2. 灾备集群收到切换命令后，暂停集群内仲裁，开启首备DN只读。
3. 灾备集群的首备DN通过主备集群数据传输链路向主集群的主DN发起倒换请求，并接收增量日志进行回放。
4. 完成回放后，灾备集群升为主集群。
5. 主集群在收到灾备集群的倒换请求后，会退掉所有业务线程并清理相关资源，再降为备集群。
6. 主备集群的角色更新后，om\_agent关闭DN只读并重开CM仲裁，完成倒换。

图 7-64 集群间倒换流程



接口介绍

- 1. 基于流式复制的异地容灾解决方案，通过OM Agent URI /v1/om/agent/mgr/cluster/streaming\_disaster\_recovery\_switchover向主备数据库实例发送计划内 switchover的请求。
- 2. 基于同城双中心高可用方案，通过OM Agent URI /v1/om/agent/mgr/cluster/dorado\_switch向数据库实例发送降为灾备数据库实例的请求。

7.31.2.3 操作指导

- 基于流式复制的异地容灾解决方案，请参见[基于流式复制的异地容灾演练实践参考](#)。
- 基于同城双中心高可用方案，请参见[同城双中心高可用方案容灾演练实践参考](#)。

须知

集群间倒换是会影响业务的高危操作，建议根据数据库管控平台的指导材料，在管控平台上进行统一操作。

7.31.2.3.1 基于流式复制的异地容灾演练实践参考

容灾演练介绍

使用计划内倒换switchover功能可实现容灾演练。该操作可实现主备集群角色互换，原灾备集群升主对外提供服务，原主集群降备。该功能执行过程保证无数据丢失，即RPO=0。

表 7-55 容灾演练流程

| 步骤 | 操作           | 说明                                                                        |
|----|--------------|---------------------------------------------------------------------------|
| 1  | 前置检查项        | 对容灾切换的前置条件进行检查。                                                           |
| 2  | 执行切换         | 主备集群调用计划内切换 switchover接口进行切换操作。                                           |
| 3  | 容灾演练后置检查     | 容灾切换完成后，核查数据库集群状态，数据库集群类型，容灾状态等信息，并上报未参与容灾切换的实例ID等。相关信息用于核查流程执行结果，决策修复策略。 |
| 4  | 切换出现异常情况下的修复 | 根据后置检查结果，对容灾演练中的异常场景进行修复。                                                 |

前置检查项

| check项描述                                                | 检查方式                                                                                                                                                        | 检查原因                                                                            | 检查不通过时的处理       |
|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|-----------------|
| 数据面IP需要使用ping测试互通，port使用curl命令测试互通，主备集群监听的端口都需要从对端进行核查。 | 主备集群所有节点进行双向测试，IP为数据面IP。<br>ping <ip><br>CN<br>curl <ip>:8001<br>DN<br>curl <ip>:30501<br>curl <ip>:30505                                                   | 切换前原主集群的baseport +1，+5端口处于使用状态，但备集群的baseport +1，+5端口并不在使用状态。检查可确保切换完成后主备集群连接正常。 | 端口确认放通后，重新进行检测。 |
| 集群状态（Normal/Degraded）。                                  | 可通过cm_ctl query -Cvp进行查询。                                                                                                                                   | 主备集群有其中一个集群处于 unavailable 状态会导致容灾切换失败。                                          | 集群异常排除后重新查询。    |
| 观察监控指标RPO是否小于10秒，RTO<=1秒。                               | 可通过om_agent接口POST /v1/om/agent/mgr/cluster/streaming_disaster_recovery_query 进行查询。<br><br>对接adaptor的情况下可以调用 "streaming_disaster_recovery_query"接口，直接观察执行结果。 | RPO指标过高说明有较多日志需要传到灾备集群，RTO指标过高会导致切换过程有较多日志需要回放。                                 | 等待指标下降到要求规格内。   |

| check项描述                                           | 检查方式                                                                                                                                                                                                                                                     | 检查原因                   | 检查不通过时的处理    |
|----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|--------------|
| 容灾状态是否主集群处于“增量同步中/archive”，灾备集群处于“增量恢复中/recovery”。 | 可通过om_agent接口POST /v1/om/agent/mgr/cluster/streaming_disaster_recovery_query 进行查询。<br><br>对接adaptor的情况下可以调用 "streaming_disaster_recovery_query"接口，直接观察执行结果。                                                                                              | 不正常的容灾状态会导致容灾切换执行失败。   | 容灾异常排除后重新查询。 |
| 一致性点推进检查。                                          | cm_ctl query -vg   grep Global_target_barrierId<br>Global_target_barrierId:<br>csn_000000000000017259643_170727842794<br>1<br>cm_ctl query -vg   grep Global_target_barrierId<br>Global_target_barrierId:<br>csn_000000000000017259645_170727842954<br>3 | 一致性点推进异常会导致灾备集群日志回放异常。 | -            |

## 执行切换

满足前置检查条件，通过以下接口执行切换：

OM Agent URI /v1/om/agent/mgr/cluster/streaming\_disaster\_recovery\_switchover  
执行主备切换。

## 容灾演练后置检查

容灾集群执行完容灾switchover后，通过 OM Agent URI /v1/om/agent/mgr/cluster/dr\_cluster\_postcheck检查集群状态，集群类型，容灾状态等信息，并上报未参与容灾切换或者容灾升主的实例ID等。相关信息用于接口调用方核查流程执行结果，决策修复策略。

接口返回结果示例：

- cluster\_state：集群状态。
- hadr\_cluster\_stat：容灾状态。
- not\_in\_switchover\_instance\_ids：未参与主备倒换的组件实例ID。
- incorrect\_params\_instance\_ids：参数配置错误的组件实例ID。
- hadr\_primary\_cluster\_name：当前容灾中主集群的集群名称。

## 切换出现异常情况下的修复

处理依据：切换流程执行结束后，查看两个集群后置检查接口的返回结果，详细操作请参见[容灾演练后置检查](#)。

- 情况一：  
后置检查接口的返回hadr\_cluster\_stat：两个集群的容灾状态无archive，该情况说明切换后无主集群。



#### 修复方案：

对原主集群调用“容灾切换异常场景集群修复”接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_manual\_recovery，快速恢复原主集群对外提供服务。

基于集群管理模式的容灾关系时，修复完成后可通过集群刷新容灾序列接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_serial\_change指定“opt\_type”为“set\_dr\_repl\_dest\_conf”修改集群日志接收端信息。

- 情况二：

后置检查接口的返回hadr\_cluster\_stat：两个集群的容灾状态无recovery，该情况说明切换后无灾备集群。

#### 修复方案：

对原主集群调用“容灾切换异常场景集群修复”接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_manual\_recovery，修复为灾备集群。

基于集群管理模式的容灾关系时，修复完成后可通过集群刷新容灾序列接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_serial\_change指定“opt\_type”为“set\_dr\_repl\_dest\_conf”修改集群日志接收端信息。

- 情况三：

后置检查接口的返回hadr\_cluster\_stat：原主集群状态为archive，原灾备集群状态为recovery，该情况说明切换流程异常，并且主备集群都发生回滚，容灾关系未改变。

#### 修改方案：

依据后置检查接口的返回的以下值是否为空，针对异常组件进行修复。

- not\_in\_switchover\_instance\_ids：未参与主备倒换的组件实例ID。
- incorrect\_params\_instance\_ids：参数配置错误的组件实例ID（若其中存在['dr\_dest\_name', 'Incorrect']，请使用情况五对dr\_dest\_name进行修复）。

- 情况四：

后置检查接口的返回hadr\_cluster\_stat：原主集群已经降备为recovery，原灾备集群升主为archive。该情况说明切换流程异常，并且主备集群未发生回滚，容灾关系已改变，切换实际已经完成。

依据后置检查接口的返回的以下值是否为空，针对异常组件进行修复。

- not\_in\_switchover\_instance\_ids：未参与主备倒换的组件实例ID。
- incorrect\_params\_instance\_ids：参数配置错误的组件实例ID（若其中存在['dr\_dest\_name', 'Incorrect']，请使用情况五对dr\_dest\_name进行修复）。

- 情况五：

后置检查接口返回的incorrect\_params\_instance\_ids中存在['dr\_dest\_name', 'Incorrect']时，通过集群刷新容灾序列接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_serial\_change指定“opt\_type”为“set\_dr\_repl\_dest\_conf”修改集群日志接收端信息。

## 2.1. 组件异常

未参与容灾切换的实例ID使用后置检查接口上报后，可以通过 OM Agent URI /v1/om/agent/mgr/cluster/repair进行修复。

#### 修复场景：

| 组件类型     | 问题场景                     |
|----------|--------------------------|
| CMS(DCC) | 组件异常未参与容灾切换导致参数未设置。      |
| CMA      | 组件异常未参与容灾切换导致参数未设置。      |
| ETCD     | 组件异常未参与容灾切换导致参数未设置。      |
| CN       | 组件异常未参与容灾切换导致状态异常或者参数异常。 |
| DN       | 组件异常未参与容灾切换导致状态异常或者参数异常。 |

7.2. 集群异常

容灾主备集群switchover切换时，由于异地网络异常、集群组件异常等原因导致切换异常失败，可通过 OM Agent URI /v1/om/agent/mgr/cluster/dr\_cluster\_manual\_recovery恢复集群为容灾主集群或者灾备集群。

- 针对无主场景（两个灾备集群），调用内核修复接口进行修复，优先快速恢复原主集群对外提供服务。
- 针对双主场景（两个主集群），有新主集群对外提供服务，原主处于只读状态，调用内核修复接口修复为新灾备集群。

基于集群管理模式的容灾关系时，修复完成后可通过刷新容灾序列接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_serial\_change 指定"opt\_type"为 "set\_dr\_repl\_dest\_conf"修改集群日志接收端信息。

7.31.2.3.2 同城双中心高可用方案容灾演练实践参考

容灾演练介绍

使用计划内主备集群switchover切换功能可实现容灾演练。该操作可实现主备集群角色互换，原灾备集群升主对外提供服务，原主集群降备。该功能执行过程保证无数据丢失，即RPO=0。

表 7-56 容灾演练流程

| 步骤 | 操作           | 说明                                                                  |
|----|--------------|---------------------------------------------------------------------|
| 1  | 前置检查项        | 对容灾切换的前置条件进行检查。                                                     |
| 2  | 执行切换         | 主备集群调用计划内切换switchover接口进行切换操作。                                      |
| 3  | 容灾演练后置检查     | 容灾切换完成后，核查集群状态，集群类型，容灾状态等信息，并上报未参与容灾切换的实例ID等。相关信息用于核查流程执行结果，决策修复策略。 |
| 4  | 切换出现异常情况下的修复 | 根据后置检查结果，对容灾演练中的异常场景进行修复。                                           |

## 前置检查项

| check项描述                                                | 检查方式                                                                                                                                                                                                                                             | 检查原因                                                                                   | 检查不通过时的处理       |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|-----------------|
| 数据面IP需要使用ping测试互通，port使用curl命令测试互通，主备集群监听的端口都需要从对端进行核查。 | 主备集群所有节点进行双向测试，IP为数据面IP。<br>ping <ip><br>CN<br>curl <ip>:39001<br>DN<br>curl <ip>:45001<br>curl <ip>:45021。每个分片+20                                                                                                                               | 切换前原主集群CN的39001，DN的45001等端口处于使用状态，但备集群的CN的39001，DN的45001等端口并不在使用状态。检查可确保切换完成后主备集群连接正常。 | 端口确认放通后，重新进行检测。 |
| 集群状态Normal。                                             | 可通过cm_ctl query -Cvp进行查询。                                                                                                                                                                                                                        | 主备集群有其中一个集群处于非Normal或者Degraded情况会导致容灾。                                                 | 集群异常排除后重新查询。    |
| 观察监控指标RPO是否等于0秒，RTO<=1秒。                                | 可通过om_agent接口POST /v1/om/agent/mgr/cluster/dorado_query进行查询。                                                                                                                                                                                     | RPO指标过高说明有较多日志需要传到灾备集群，RTO指标过高会导致切换过程有较多日志需要回放。                                        | 等待指标下降到要求规格内。   |
| 容灾状态是否主集群处于“增量同步中/archive”，灾备集群处于“增量恢复中/recovery”。      | 可通过om_agent接口POST /v1/om/agent/mgr/cluster/dorado_query进行查询。                                                                                                                                                                                     | 不正常的容灾状态会导致容灾切换执行失败。                                                                   | 容灾异常排除后重新查询。    |
| 一致性点推进检查。                                               | cm_ctl query -vg   grep Global_target_barrierId<br>Global_target_barrierId:<br>csn_000000000000017259643_1707278427941<br>cm_ctl query -vg   grep Global_target_barrierId<br>Global_target_barrierId:<br>csn_000000000000017259645_1707278429543 | 一致性点推进异常会导致灾备集群日志回放异常。                                                                 | -               |

## 执行切换

满足前置检查条件，通过以下接口执行切换：

OM Agent URI /v1/om/agent/mgr/cluster/dorado\_switch执行主备集群计划内切换。

## 容灾演练后置检查

通过OM Agent URI /v1/om/agent/mgr/cluster/dr\_cluster\_postcheck执行容灾演练后置检查。

后台执行后置检查参考：

- 按主集群：

```
curl -H "Content-Type:application/json" -d '{"drMode": " dorado_cluster ", "drOptionType": "switchover", "drClusterMode": "primary"}' -X POST $HOST_IP:port/v1/om/agent/mgr/cluster/dr_cluster_postcheck
```
- 按备集群：

```
curl -H "Content-Type:application/json" -d '{"drMode": " dorado_cluster ", "drOptionType": "switchover", "drClusterMode": "disaster_standby"}' -X POST $HOST_IP:port/v1/om/agent/mgr/cluster/dr_cluster_postcheck
```

容灾集群执行完容灾switchover后，通过此接口核查集群状态，集群类型，容灾状态等信息，并上报未参与容灾切换或者容灾升主的实例ID等。相关信息用于接口调用方核查流程执行结果，决策修复策略。

接口返回结果示例：

- cluster\_state：集群状态。
- hadr\_cluster\_stat：容灾状态。
- not\_in\_switchover\_instance\_ids：未参与主备倒换的组件实例ID。
- incorrect\_params\_instance\_ids：参数配置错误的组件实例ID。
- hadr\_primary\_cluster\_name：当前容灾中主集群的集群名称。

## 切换出现异常情况下的修复

处理依据：切换流程执行结束后，查看两个集群后置检查接口的返回结果，详细操作请参见[容灾演练后置检查](#)。

- 情况一：后置检查接口的返回hadr\_cluster\_stat：两个集群的容灾状态显示为recovery，该情况说明切换后无主集群。  
修复方案：参见[集群异常](#)进行修复。
- 情况二：后置检查接口的返回hadr\_cluster\_stat：原主集群状态为archive，原灾备集群状态为recovery，该情况说明切换流程异常，并且原主集群发生回滚，容灾关系未改变。  
修改方案：参见[组件异常](#)进行修复。
- 情况三：后置检查接口的返回hadr\_cluster\_stat：原主集群已经降备为recovery，原灾备集群升主为archive。该情况说明切换流程异常，但容灾关系已改变，切换实际已经完成。  
修改方案：参见[组件异常](#)进行修复。

### ?1. 组件异常

依据后置检查接口（参见容灾演练后置检查）的返回的以下值是否为空，针对异常组件进行修复：

- not\_in\_switchover\_instance\_ids：未参与主备倒换的组件实例ID。
- incorrect\_params\_instance\_ids：参数配置错误的组件实例ID。

节点修复接口：  
  
POST /v1/om/agent/mgr/cluster/repair。

修复场景：

| 组件类型     | 问题场景                     |
|----------|--------------------------|
| CMS(DCC) | 组件异常未参与容灾切换导致参数未设置。      |
| CMA      | 组件异常未参与容灾切换导致参数未设置。      |
| CN       | 组件异常未参与容灾切换导致状态异常或者参数异常。 |
| DN       | 组件异常未参与容灾切换导致状态异常或者参数异常。 |

7.2. 集群异常

容灾主备集群switchover切换，由于异地网络异常、集群组件异常等原因导致切换异常失败，可通过该接口恢复集群为容灾主集群或者灾备集群。可使用“容灾切换异常场景集群修复”接口进行修复。

- 集群修复接口：  
POST /v1/om/agent/mgr/cluster/dr\_cluster\_manual\_recovery。
- 查询阶段接口：  
对接adaptor的情况下可以使用"dorado\_switchover\_precheck"，确认主集群降备之后，备集群检查读取主降备成功标识（Reduce standby success），为备升主提供依据。该标记没写入情况下，无法保证日志都已经完成同步，因此需要恢复原主集群；该标记已经写入的情况下，日志已经完成主备集群同步，可以恢复原备（新主）集群。

场景一：该标记位未写入，主降备流程异常会自动回滚，主集群自动恢复；回滚异常会呈现无主场景（两个灾备），使用内核修复接口进行修复，优先快速恢复原主集群对外提供服务。

场景二：该标记位已写入，说明备升主流程已经启动，备升主异常场景，针对无主场景（两个灾备集群），使用内核修复接口进行修复，快速恢复原备（新主）集群对外提供服务。

后台修复命令参考：curl -H "Content-Type:application/json" -d '{"drMode": "dorado\_cluster", "drClusterMode": "primary"}' -X POST \$HOST\_IP:\$PORT/v1/om/agent/mgr/cluster/dr\_cluster\_manual\_recovery。

- \$HOST\_IP：执行命令节点管理IP。
- \$PORT：OM\_AGENT监听port。

7.31.2.4 常见问题

1. 基于流式复制的异地容灾解决方案，请参见[两地三中心跨Region容灾计划内倒换switchover流程异常](#)。
2. 基于同城双中心高可用方案，请参见[同城双中心计划内倒换switchover流程异常](#)。

### 7.31.2.4.1 两地三中心跨 Region 容灾计划内倒换 switchover 流程异常

#### 问题现象

- **现象一：**  
计划内switchover中主集群执行命令返回如下错误，提示主集群产生一致性点失败。  
Result exception error : Failed to generate switchover barrier before switchover
- **现象二：**  
计划内switchover中主集群执行命令返回如下错误，提示主集群业务截断出现失败。  
Result exception error : Failed to truncate service before switchover
- **现象三：**  
计划内switchover中灾备集群执行命令返回如下错误，提示灾备集群核查数据一致性点出现超时。  
Result exception error : Failed to do check switchover\_barrier on all main standby dn and cn. Because check timeout: XXs
- **现象四：**  
计划内switchover启用快速倒换模式后，容灾倒换出现失败。
- **现象五：**  
主备倒换过程中，有故障节点未参与容灾倒换。
- **现象六：**  
计划内switchover成功，但倒换过程中，原主集群的原主DN或者CN发生进程退出。

#### 原因分析

- **现象一原因：**  
在主集群接收到计划内switchover命令，主集群降为灾备集群前会产生一致性点switchover barrier。这是执行switchover的前提，用于保证主备集群所有CN节点和所有DN分片的日志停止在一致性点。由于主集群内网络抖动等原因导致主集群内产生switchover barrier失败将放弃本次计划内switchover。
- **现象二原因：**  
在主集群接收到计划内switchover命令，主集群降为灾备集群前会产生一致性点switchover barrier，并进行业务截断终止日志写入操作。这是执行switchover的前提，用于保证主备集群所有CN节点和所有DN分片的日志停止在一致性点。由于主集群内CN/DN实例异常，异地网络异常，灾备集群不会响应等原因导致主集群业务截断失败将放弃本次计划内switchover。
- **现象三原因：**  
在灾备集群接收到计划内switchover命令，灾备集群升为主集群前会在所有CN和首备DN上查询是否收到一致性点switchover barrier。这是执行switchover的前提，用于保证主备集群所有CN节点和所有DN分片的日志停止在一致性点。由于异地网络异常等原因，灾备集群在超时时间内无法获得switchover barrier将放弃执行本次计划内switchover。
- **现象四原因：**  
快速倒换失败原因较多，这里补充下简单定位手段快速定位定界
  - a. 从最新的OM日志(最新的\$GAUSSLOG/om/gs\_om\*日志)中关键字查询，判断是否切换失败，以及存在回滚。

- b. 根据查询的Error信息中的详细报错信息，可定位到具体快速倒换中的哪一步出现问题。
- c. 根据上一步定位出的失败原因，去对应的集群节点查看om-agent日志或者gs\_log日志，获取到具体的失败原因。

提供个定位案例说明下上述过程：

在查询到错误日志“Failed to do streaming disaster cluster switchover”，并且获取到错误信息为“switchover timeout”时，通过打印出来的实例，去对应的主、主备节点上搜索gs\_log日志里的switchover，判断内核的switchover是在哪里失败或者超时的，通过收发消息的打印，判断是网络交互出问题，还是在指定时间内日志没有发起，或者是线程退出超时等等。

- **现象五原因：**

因服务器宕机，网络中断等原因导致节点未参与主备倒换。

- **现象六原因：**

在对应进程日志中搜索关键字“abort switchover”，如果有该关键字，说明是因为降备超时，进程主动退出，属于正常现象，待进程被CM重新拉起即可。如果没有，则需要进一步排查日志确定原因。

## 处理方法

**步骤1** 主集群执行switchover主降备命令，灾备集群执行switchover灾备升主命令超时或者失败会回滚为倒换前状态，计划内倒换switchover可在主集群和灾备集群重新执行。

**步骤2** 若多次执行switchover均出现失败，需进一步分析流式容灾相关日志文件。

----结束

快速倒换流程故障处理步骤：

**步骤1** 通过cm\_ctl工具确认集群是否完成倒换或者回滚，实例状态是否正常，异常节点可使用节点修复功能。

```
cm_ctl query -Cvp
```

主集群normal态示例：

```
[CMServer State]
```

| node     | instance state |
|----------|----------------|
| 1 <IP> 1 | Primary        |
| 2 <IP> 2 | Standby        |
| 3 <IP> 3 | Standby        |

```
[ETCD State]
```

| node        | instance state |
|-------------|----------------|
| 1 <IP> 7001 | StateFollower  |
| 2 <IP> 7002 | StateLeader    |
| 3 <IP> 7003 | StateFollower  |

```
[Cluster State]
```

```
cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL
```

```
[Coordinator State]
```

```

node instance state

1 <IP> 5001 <PORT> Normal
2 <IP> 5002 <PORT> Normal
3 <IP> 5003 <PORT> Normal

[Central Coordinator State]

node instance state

1 <IP> 5001 Normal

[GTM State]

node instance state sync_state

1 <IP> 1001 P Primary Connection ok Sync
2 <IP> 1002 S Standby Connection ok Sync
3 <IP> 1003 S Standby Connection ok Sync

[Datanode State]

node instance state | node instance state | node instance
state

1 <IP> 6001 <PORT> P Primary Normal | 2 <IP> 6002 <PORT> S Standby Normal | 3 <IP> 6003
<PORT> S Standby Normal
2 <IP> 6004 <PORT> P Primary Normal | 3 <IP> 6005 <PORT> S Standby Normal | 1 <IP> 6006
<PORT> S Standby Normal
3 <IP> 6007 <PORT> P Primary Normal | 1 <IP> 6008 <PORT> S Standby Normal | 2 <IP> 6009
<PORT> S Standby Normal

```

#### 灾备集群normal态示例：

```

[CMServer State]

node instance state

1 <IP> 1 Primary
2 <IP> 2 Standby
3 <IP> 3 Standby

[ETCD State]

node instance state

1 <IP> 7001 StateLeader
2 <IP> 7002 StateFollower
3 <IP> 7003 StateFollower

[Cluster State]

cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL

[Coordinator State]

node instance state

1 <IP> 5001 <PORT> Normal
2 <IP> 5002 <PORT> Normal
3 <IP> 5003 <PORT> Normal

[GTM State]

node instance state sync_state

```



```
1 <IP> 1001 P Primary Connection ok Sync
2 <IP> 1002 S Standby Connection ok Sync
3 <IP> 1003 S Standby Connection ok Sync

[Datanode State]

node instance state | node instance state | node instance
state

1 <IP> 6001 <PORT> P Main Standby Normal | 2 <IP> 6002 <PORT> S Cascade Standby Normal |
3 <IP> 6003 <PORT> S Cascade Standby Normal
2 <IP> 6004 <PORT> P Main Standby Normal | 3 <IP> 6005 <PORT> S Cascade Standby Normal |
1 <IP> 6006 <PORT> S Cascade Standby Normal
3 <IP> 6007 <PORT> P Main Standby Normal | 1 <IP> 6008 <PORT> S Cascade Standby Normal |
2 <IP> 6009 <PORT> S Cascade Standby Normal
```

**步骤2** 倒换失败集群实例参数，使用gs\_om工具进行查询。

```
echo "password" | gs_om -t dr_post_check -c [switchover|failover|query] -m [primary|disaster_standby] --dr_type streaming_cluster
```

- 主数据库实例相关参数：
  - cmserver参数backup\_open=0, dn\_arbitrate\_mode=quorum, enable\_transaction\_read\_only=on, cm\_hadr\_process\_type=none。
  - cmagent参数agent\_backup\_open=0, enable\_auto\_start=1, cm\_hadr\_process\_type=none。
  - etcd参数/xxxx/CMServer/backup\_open=0。
  - CN参数stream\_cluster\_run\_mode=cluster\_primary, hadr\_process\_type=none, auto\_csn\_barrier=1, default\_transaction\_read\_only=off。
  - DN参数stream\_cluster\_run\_mode=cluster\_primary, hadr\_process\_type=none, default\_transaction\_read\_only=off。
- 灾备数据库实例相关参数：
  - cmserver参数backup\_open=2, dn\_arbitrate\_mode=quorum, enable\_transaction\_read\_only=on, cm\_hadr\_process\_type=none。
  - cmagent参数agent\_backup\_open=2, enable\_auto\_start=1, cm\_hadr\_process\_type=none。
  - etcd参数/xxxx/CMServer/backup\_open=2。
  - CN参数stream\_cluster\_run\_mode=cluster\_standby, hadr\_process\_type=none, auto\_csn\_barrier=0, default\_transaction\_read\_only=off。
  - DN参数stream\_cluster\_run\_mode=cluster\_standby, hadr\_process\_type=none, default\_transaction\_read\_only=off。

**步骤3** 出现双灾备集群实例状态的极端场景，需人工介入修复。

- 出现双灾备数据库实例场景，一般指主集群降为灾备集群并且未回滚，灾备集群升主流程出现回滚。此时两个集群均不可写入。
- 使用gs\_om工具对原主数据库实例进行修复，目标恢复为主数据库实例。

```
echo "password" | gs_om -t dr_cluster_manual_recovery -m primary --dr_type streaming_cluster [--time-out=SECS]
```
- 修复成功，关闭只读，相关参数default\_transaction\_read\_only置为off。

**步骤4** 出现双主集群实例状态极端场景，需人工介入修复。

- 出现双主集群场景，一般指灾备集群升主成功并且关闭了只读；主集群降为灾备流程出现失败并且出现回滚，并且保持只读打开状态。

- 集群使用gs\_om工具对原主集群进行修复，目标变为灾备集群。  

```
echo "password" | gs_om -t dr_cluster_manual_recovery -m disaster_standby --dr_type streaming_cluster [--time-out=SECS]
```
- 原主集群因为开了只读，不会有客户数据进入，但是日志依然会分叉，修复过程较长。修复成功，关闭只读，参数default\_transaction\_read\_only置于off。

#### 📖 说明

- 当使用**步骤3**、**步骤4**中描述的修复工具（dr\_cluster\_manual\_recovery）对集群进行修复时，若集群存在日志堆积，修复的具体时长受集群重启后待回放的日志量影响。
- 可通过排查对应集群节点的gs\_log日志确认，若日志中持续打印“xlog redo at lsn X/XXX”，且节点为STARTING状态，可以判断节点正在回放堆积的日志，等待修复流程结束即可。

**步骤5** 基于集群管理模式的容灾关系时，后置检查接口返回的incorrect\_params\_instance\_ids中存在['dr\_dest\_name', 'Incorrect']时，修复完成后可通过刷新容灾序列接口POST /v1/om/agent/mgr/cluster/dr\_cluster\_serial\_change 指定"opt\_type"为"set\_dr\_repl\_dest\_conf"修改集群日志接收端信息

**步骤6** 故障节点经过硬件维修、上电等操作，重新加入集群。

#### 须知

该节点加入集群后，集群状态有可能显示为Normal，该节点上实例状态也有可能显示Normal，但该节点上实例的部分配置参数不正确，仍需要完成如下修复流程。

- 通过后置检查接口查询未参与容灾倒换的实例。  

```
echo "password" | gs_om -t dr_post_check -c switchover -m primary|disaster_standby --dr_type streaming_cluster
```
- 使用cm\_ctl stop停掉未参与容灾倒换的coordinator、datanode以及cm\_server实例。  

```
cm_ctl stop -n NODEID -D DATADIR
```

**步骤7** 删除残留的归档槽。

- 排查CN和DN是否存在归档槽残留。  

```
SELECT slot_name FROM pg_catalog.pg_get_replication_slots() where slot_type='physical' and slot_name not in ('gs_roach_full', 'gs_roach_inc', 'gs_roach_table_full', 'gs_roach_table_inc') and slot_name not like 'cn_%' and slot_name not like 'dn_%';
```
- 若存在归档槽，则执行如下SQL删除（slot\_name为查询到的归档槽名）。  

```
SELECT * FROM pg_catalog.pg_drop_replication_slot('slot_name');
```

----结束

### 7.31.2.4.2 同城双中心计划内倒换 switchover 流程异常

#### 问题现象

- 现象一：  
主降备出现流程失败，查看日志primary\_demote超时失败。
- 现象二：  
备升主出现等待集群状态恢复为normal状态超时，集群状态Unavailable，存在分片无主DN。
- 现象三：

容灾倒换出现失败。

- 现象四：  
主备倒换过程中，有故障节点未参与容灾倒换。
- 现象五：  
计划内switchover主降备成功，但在降备过程中，主集群的主DN或CN进程发生退出。

## 原因分析

- 现象一原因：  
查看失败节点日志，主要是checkpoint时，需要刷新的脏页较多，超过primary\_demote的等待时间。
- 现象二原因：  
日志回放所需时间超过了switchover升主流程等待时间（300秒），导致集群无法在300秒内恢复为normal或degraded状态，switchover升主流程失败。
- 现象三原因：  
倒换失败原因较多，这里补充下简单定位手段快速定位定界
  - a. 从最新的OM日志(最新的\$GAUSSLOG/om/gs\_om\*日志)中关键字查询，判断是否切换失败，以及存在回滚。
  - b. 根据查询的Error信息中的详细报错信息，或者\$PGHOST目录的步骤文件do\_dorado\_switch.dat，可定位到具体快速倒换中的哪一步出现问题。
  - c. 根据上一步定位出的失败原因，去对应的集群节点查看OM\_Agent日志或者gs\_log日志，获取到具体的失败原因。
- 现象四原因：  
因服务器宕机，网络中断等原因导致节点未参与主备倒换。
- 现象五原因：  
在对应进程日志中搜索关键字“abort switchover”。若存在该关键字，表示因降备超时导致进程主动退出，属于正常现象，待进程被CM重新拉起即可。若未找到，则需要进一步排查日志确定原因。

## 处理方法

- 现象一处理方法：

**步骤1** 等待主DN降为备DN成功。

**步骤2** 重入switchover主降备流程。

----结束

- 现象二处理方法：

**步骤1** 等待集群恢复normal或者degraded状态后。

**步骤2** 重新执行switchover命令，进行重试。

----结束

倒换流程故障处理步骤：

**步骤1** 通过cm\_ctl工具确认集群是否完成倒换或者回滚，集群状态是否正常，异常节点可使用节点修复功能。

```
cm_ctl query -Cvp
```

主集群normal态示例：

```
[CMServer State]

node instance state

1 <IP> 1 Primary
2 <IP> 2 Standby
3 <IP> 3 Standby

[ETCD State]

node instance state

1 <IP> 7001 StateFollower
2 <IP> 7002 StateLeader
3 <IP> 7003 StateFollower

[Cluster State]

cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL

[Coordinator State]

node instance state

1 <IP> 5001 <PORT> Normal
2 <IP> 5002 <PORT> Normal
3 <IP> 5003 <PORT> Normal

[Central Coordinator State]

node instance state

1 <IP> 5001 Normal

[GTM State]

node instance state sync_state

1 <IP> 1001 P Primary Connection ok Sync
2 <IP> 1002 S Standby Connection ok Sync
3 <IP> 1003 S Standby Connection ok Sync

[Datanode State]

node instance state | node instance state | node instance
state

1 <IP> 6001 <PORT> P Primary Normal | 2 <IP> 6002 <PORT> S Standby Normal | 3 <IP> 6003
<PORT> S Standby Normal
2 <IP> 6004 <PORT> P Primary Normal | 3 <IP> 6005 <PORT> S Standby Normal | 1 <IP> 6006
<PORT> S Standby Normal
3 <IP> 6007 <PORT> P Primary Normal | 1 <IP> 6008 <PORT> S Standby Normal | 2 <IP> 6009
<PORT> S Standby Normal
```

灾备集群normal态示例：

```
[CMServer State]

node instance state

```

```
1 <IP> 1 Primary
2 <IP> 2 Standby
3 <IP> 3 Standby

[ETCD State]

node instance state

1 <IP> 7001 StateFollower
2 <IP> 7002 StateLeader
3 <IP> 7003 StateFollower

[Cluster State]

cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL

[Coordinator State]

node instance state

1 <IP> 5001 <PORT> Normal
2 <IP> 5002 <PORT> Normal
3 <IP> 5003 <PORT> Normal

[Central Coordinator State]

node instance state

1 <IP> 5001 Normal

[GTM State]

node instance state sync_state

1 <IP> 1001 P Primary Connection ok Sync
2 <IP> 1002 S Standby Connection ok Sync
3 <IP> 1003 S Standby Connection ok Sync

[Datanode State]

node instance state | node instance state | node instance
state

1 <IP> 6001 <PORT> P Standby Normal | 2 <IP> 6002 <PORT> S Standby Normal | 3 <IP> 6003
<PORT> S Standby Normal
2 <IP> 6004 <PORT> P Standby Normal | 3 <IP> 6005 <PORT> S Standby Normal | 1 <IP> 6006
<PORT> S Standby Normal
3 <IP> 6007 <PORT> P Standby Normal | 1 <IP> 6008 <PORT> S Standby Normal | 2 <IP> 6009
<PORT> S Standby Normal
```

**步骤2** 倒换失败集群参数，使用gs\_om工具进行查询。

```
gs_om -t dr_post_check -c [switchover|failover|query|restore] -m [primary|disaster_standby] --dr_type
dorado_cluster
```

- 主集群相关参数：
  - cmserver参数backup\_open=0，dn\_arbitrate\_mode=quorum，cm\_hadr\_process\_type=none。
  - cmagent参数agent\_backup\_open=0，enable\_auto\_start=1，cm\_hadr\_process\_type=none。
  - etcd参数/xxxx/CMServer/backup\_open=0。
  - CN参数cluster\_run\_mode=cluster\_primary，hadr\_process\_type=none，auto\_csn\_barrier=1，default\_transaction\_read\_only=off。

- DN参数cluster\_run\_mode=cluster\_primary, hadr\_process\_type=none, default\_transaction\_read\_only=off。
- 灾备集群相关参数：
  - cmserver参数backup\_open=1, dn\_arbitrate\_mode=quorum, cm\_hadr\_process\_type=none。
  - cmagent参数agent\_backup\_open=1, enable\_auto\_start=1, cm\_hadr\_process\_type=none。
  - etcd参数/xxxx/CMServer/backup\_open=1。
  - CN参数cluster\_run\_mode=cluster\_standby, hadr\_process\_type=none, auto\_csn\_barrier=0, default\_transaction\_read\_only=off。
  - DN参数cluster\_run\_mode=cluster\_standby, hadr\_process\_type=none, default\_transaction\_read\_only=off。

**步骤3** 后置接口若发现有未参与倒换、参数配置错误的实例，出现双灾备集群状态的极端场景，需人工介入修复。

- 两种情况可导致集群无主集群可用：
  - 1.主降备流程在写下Reduce standby success之前，数据库主集群降备失败，进入回滚流程，回滚失败。
  - 2.灾备集群升主流程失败。使用gs\_om工具对集群进行修复，快速修复主集群对外提供服务。

```
echo "password" | gs_om -t dr_cluster_manual_recovery -m primary --dr_type dorado_cluster [--time-out=SECS]
```

**步骤4** 故障节点经过硬件维修、上电等操作，重新加入集群。

#### 须知

该节点加入集群后，集群状态有可能显示为Normal，该节点上集群状态也有可能显示Normal，但该节点上集群的部分配置参数不正确，仍需要完成如下修复流程。

- 通过后置检查接口查询未参与容灾倒换的集群。

```
gs_om -t dr_post_check -c switchover -m primary|disaster_standby --dr_type dorado_cluster
```
- 使用cm\_ctl stop停掉未参与容灾倒换的coordinator、datanode以及cm\_server实例。

```
cm_ctl stop -n NODEID -D DATADIR
```

----结束

## 7.32 主备重建

### 7.32.1 特性概述

主备重建是GaussDB数据库服务高可用（HA）体系中的核心修复特性，它通过从主节点或完整备节点进行全量或增量同步数据的方式，解决因备节点数据延迟、磁盘损坏、日志断裂等原因导致其与主节点数据不一致的问题，从而实现备节点的快速纳管与集群可用性的恢复。该特性是实现集群高可用性、保证数据最终一致性的关键技术保障。

## 功能定义

通过基于WAL日志的物理复制与回放技术，解决因备节点故障、宕机或数据损坏导致的数据不一致问题，实现备节点数据的自动化或手动化全量/增量重建，确保主备集群的数据最终一致性，保障集群的持续高可用。

## 用户场景

- **行业领域：**金融（支付、清算）、电信（计费、账务）、政务（一网通办）等对数据可靠性与服务连续性要求极高的行业。
- **业务痛点：**
  - **硬件故障：**备节点所在服务器磁盘损坏，导致数据文件丢失或不可用。
  - **网络故障：**备节点长时间与主节点网络中断，积压的日志已被主节点清理，导致日志断点，无法通过常规日志回放进行同步。
  - **人为误操作：**在备节点上执行了误写操作，污染了数据。
- **常见触发条件：**
  - 集群内HA自动恢复
    - i. **WAL segment removed：**备节点所请求的日志LSN在主节点上不存在，导致主备日志复制出现脱节现象。
    - ii. **System id not match：**备节点上记录的system identifier与主节点不一致。通常发生于集群安装阶段或节点替换、升副本等有新节点加入集群中时，需要通过**全量重建**方式进行数据同步。
    - iii. **DCF log loss：**基于dcf的一致性复制协议，当备节点发现本地记录的最大dcf index在主节点上不存在时，需要通过重建方式进行恢复。
  - 手动触发  
在某些特殊场景下，需要手动触发重建修复的场景。  
例如：在业务高峰期下，主节点网络资源和计算资源受限，无法触发自动恢复，需要等到业务低峰期再进行重建修复的场景。
  - 集群管理流程中触发
    - i. 容灾集群的搭建和加回。
    - ii. 节点修复/替换。
    - iii. 备份恢复。

## 规格约束

- **兼容性：**
  - **版本支持：**（具体版本号请以实际发布为准）。

表 7-57 主备重建版本支持情况

| 特性         | 版本                                         |
|------------|--------------------------------------------|
| 主备全量/增量重建  | GaussDB V500R001C00及以上版本                   |
| 备机到备机的增量重建 | GaussDB 505.1及以上版本                         |
| 异步build    | GaussDB 505.2.1 spc0800、GaussDB 507.0及以上版本 |

| 特性          | 版本                 |
|-------------|--------------------|
| 并行重建和并行压缩重建 | GaussDB 506.0及以上版本 |

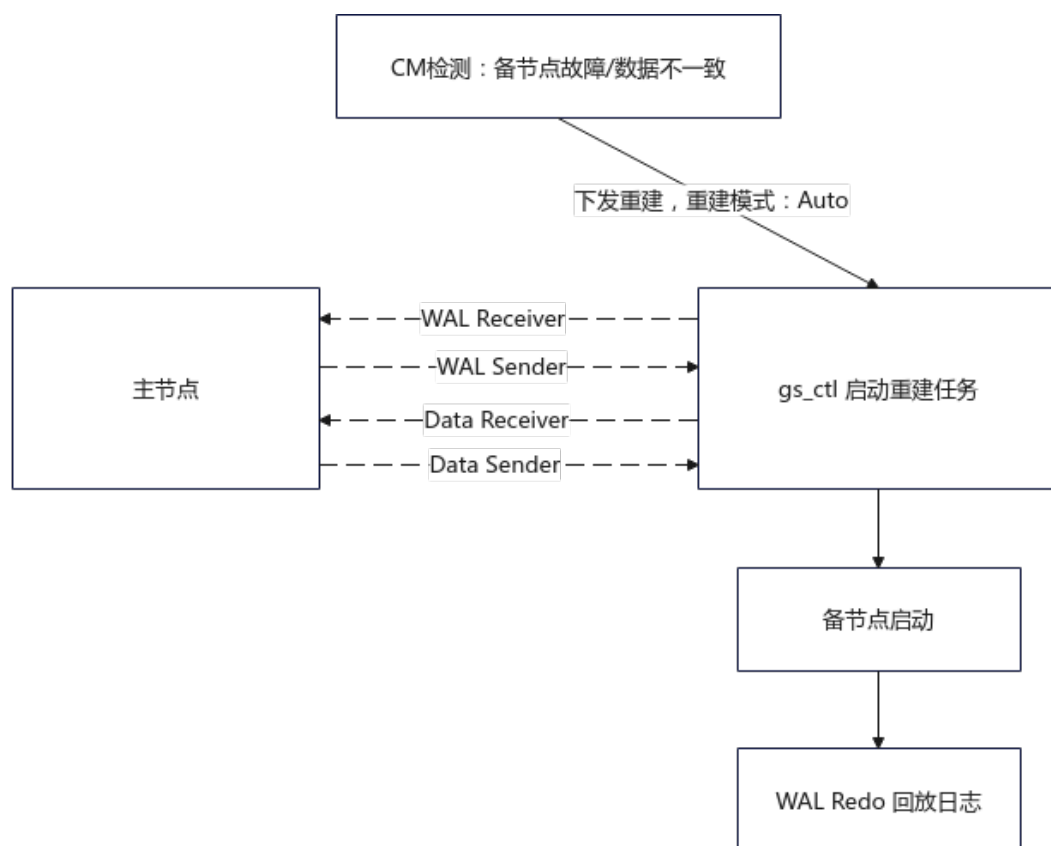
- **生态工具**：可通过OM（运维管理服务）、CM（数据库管理服务）、gs\_ctl命令行工具进行操作。
- **性能限制**：
  - **网络带宽**：重建过程涉及数据传输，对网络带宽存在消耗，建议避开业务高峰时段。
  - **主节点负载**：重建期间，主节点需要承担读取数据页和发送数据的任务，会对主节点性能产生一定影响（CPU、I/O压力增大）。
  - **磁盘I/O**：目标备节点需要接收并写入大量数据，在网络不受限制的场景下，磁盘I/O会是主要瓶颈。
- **功能约束**：
  - **主节点唯一性**：同一时刻，CM只会让主节点为一个备节点提供重建服务（但主节点支持为多个备节点同时进行数据重建）。
  - **重建过程互斥**：一个备节点不能同时从两个不同的源节点进行重建。
  - **级联备节点**：级联备节点（Cascade Standby）的重建源只能是其直接上游节点。

## 技术架构

GaussDB主备重建的核心架构流程如下图所示，主要涉及**CM**、**WAL Sender**、**WAL Receiver**、**Data Sender**、**Data Receiver**和**WAL Redo**等核心组件：



图 7-65 重建流程图



#### 核心组件说明：

- **CM**：集群高可用管理服务。负责检测节点状态，在备用节点异常时决策是否需要触发重建，并协调重建任务的开始与停止。
- **WAL Sender**：主节点上的WAL日志发送线程。负责将重建期间产生的WAL日志数据发送到备用节点。在重建任务的开始和结束，都会向备用节点发送一个一致性快照点（checkpoint），作为回放的起点和终点。
- **Data Sender**：主节点上的数据发送线程。负责从磁盘读取完整的数据文件，并将其流式传输到备用节点。
- **WAL Receiver**：备用节点上的WAL日志接收进程。负责接收从主节点发送来的WAL日志数据。
- **Data Receiver**：备用节点上的数据接收线程。负责接收来自主节点的数据文件，并将其写入到本地磁盘。
- **WAL Redo**：备用节点上的WAL日志回放线程。在接收完全量数据，备用节点拉起后，会回放重建起点之后的所有WAL日志，最终使备用节点数据与主节点完全一致。

#### 数据流向：

1. **数据链路**：全量数据文件从主节点**Data Sender**流向备用节点**Data Receiver**并落盘。
2. **日志链路**：WAL日志从主节点**WAL Sender**流向备用节点**WAL Receiver**并落盘，等待备用节点启动后，由**WAL Redo**线程进行回放。

#### 决策点：

- **重建模式选择**：除特殊场景外，当前CM服务默认下发重建模式为**Auto Build**，优先进行增量重建，若增量重建三次失败，转全量重建。

#### 核心组件说明：

- **CM**：集群高可用管理服务。负责检测节点状态，在备节点异常时决策是否需要触发重建，并协调重建任务的开始与停止。
- **WAL Sender**：主节点上的WAL日志发送线程。负责将重建期间产生的WAL日志数据发送到备节点。在重建任务的开始和结束，都会向备机发送一个一致性快照点（checkpoint），作为回放的起点和终点。
- **Data Sender**：主节点上的数据发送线程。负责从磁盘读取完整的数据文件，并将其流式传输到备节点。
- **WAL Receiver**：备节点上的WAL日志接收进程。负责接收从主节点发送来的WAL日志数据。
- **Data Receiver**：备节点上的数据接收线程。负责接收来自主节点的数据文件，并将其写入到本地磁盘。
- **WAL Redo**：备节点上的WAL日志回放线程。在接收完全量数据，备节点拉起后，会回放重建起点之后的所有WAL日志，最终使备节点数据与主节点完全一致。

#### 数据流向：

1. **数据链路**：全量数据文件从主节点**Data Sender**流向备节点**Data Receiver**并落盘。
2. **日志链路**：WAL日志从主节点**WAL Sender**流向备节点**WAL Receiver**并落盘，等待备节点启动后，由**WAL Redo**线程进行回放。

#### 决策点：

- **重建模式选择**：除特殊场景外，当前CM服务默认下发重建模式为**Auto Build**，优先进行增量重建，若增量重建三次失败，转全量重建。

## 7.32.2 技术原理

### 核心机制

1. **一致性快照同步（Establish Snapshot）**  
重建开始时，主节点上的WAL Sender线程首先会获取一个当前的一致性点（Checkpoint），并将该点的LSN（Log Sequence Number）和时间线（Timeline）信息发送给备节点。此快照点确保了重建数据的起点是一致的。
2. **全量数据文件流式传输（Streaming Base Backup）**  
主节点的Data Sender线程基于之前获取的一致性快照点，读取所有数据文件，并通过TCP连接将其以数据流的形式发送到备节点的Data Receiver线程中，Receiver线程将其写入到本地数据目录的相应位置。
3. **增量日志回放（WAL Redo）**  
在数据传输开始之后，主节点的WAL Sender线程会持续将快照点之后产生的WAL日志发送给备节点。备节点先接收并存储这些日志，待全量数据接收完毕后，WAL Redo线程会从快照点开始按顺序回放这些日志，将数据追赶到最新状态。

## 特性介绍

### 1. 全量重建（Full Build）

#### - 特性描述

通过物理复制整个主节点数据文件的方式，为故障备节点提供一个全新的、完整的数据副本。并在拷贝数据同步期间，主节点产生的所有WAL日志，通过回放达到数据一致性。

该特性的特征为复制的数据量大，耗时长。

#### - 适用场景

- i. 磁盘损坏、数据文件丢失或损坏。
- ii. 备节点WAL日志与主节点脱节（回放所需WAL日志已经被主节点清理）。
- iii. 部分集群管理流程，如：新增节点加入集群。
- iv. 人为误操作，导致数据被污染。

### 2. 增量重建（Incremental Build）

#### - 特性描述

在主备节点出现日志分叉（WAL segment removed）之后，通过对比主备节点本地的WAL日志，寻找到最近分叉点所对应的LSN。将备节点在分叉点之后的数据文件从主节点同步并覆盖，并同步主节点分叉点之后的WAL日志，通过回放达到数据一致性。

仅同步故障期间的数据变更，而非完整的数据集，比全量重建拥有更高的效率。

**当前该模式是集群内HA的默认恢复方式。**

#### - 适用场景

- i. 备节点进程异常宕机，但数据目录完好，仅需少量数据即可修复。
- ii. 集群内failover，原主节点降备后，发生日志分叉。
- iii. 容灾集群的解除和加回。

### 3. 并行Build（Parallel Build）

#### - 特性描述

当前GaussDB全量和增量重建都是通过单线程的方式完成数据传输，在大容量场景下重建速度较慢，不满足客户业务诉求。为提升在大容量场景的重建传输速度，并行build在全量重建和增量重建的基础上，引入多线程并行传输数据的机制，充分利用硬件资源，提升重建速度。

#### ■ 并行重建

- 多线程并行传输数据，支持线程数参数可配，最大程度利用网络带宽和存储设备I/O。通过设置GUC参数parallel\_build\_thread\_num可以控制并行Build的并行度。

#### ■ 压缩重建

- 采用LZ4无损压缩算法，对传输数据进行压缩，以节省网络带宽。通过设置GUC参数enable\_parallel\_build\_compression可以控制是否在并行重建传输数据的过程中进行压缩。

#### - 适用场景

TB级别以上的数据节点，需要通过全量重建的方式恢复。

- 特性约束  
在CPU、存储设备I/O、网络带宽充足的场景下，可以开启本特性提升重建效率；否则，会因为硬件资源受限而无法观察到明显增益。
- 4. 异步Build（Async Build）
  - 特性描述  
异步Build的核心目的是在发生大规模节点故障时，**优先保障主节点生产业务的连续性和可用性**，避免其因同步备节点数量不足而挂起或阻塞，同时让故障备节点能以一种“降级”状态快速回归集群参与仲裁。  
当触发region级别连续故障时，可能会出现集群内的多个备节点出现异常。此时，由于主机的业务不满足足够的同步备数量，事务无法提交，阻塞客户生产业务的运行。  
**异步Build**在当前全量和增量重建的基础上，将**WAL Receiver**模拟成正常备节点的WAL日志请求线程，替代原来重建流程中同步WAL日志的方案，使得备节点在重建期间，依然可以参与事务提交。  
当前异步build模式通过GUC参数async\_build控制，默认开启。
  - 适用场景
    - i. Region/AZ级别故障后的业务保活与快速恢复。
    - ii. 避免大规模节点重启时的业务抖动。
  - 特性约束
    - i. 仅支持**Quorum模式**下集群内主节点到备节点的全量和增量重建。
    - ii. 在节点总数大于3，且同步备个数大于2的场景下，至少残留一主一备的正常节点时，该特性才生效。

## 性能指标

- 测试环境：
  - 硬件配置
    - CPU：64核
    - 内存：256GB
    - 磁盘：NVMe SSD RAID组
    - 网络：10万兆专用网络（测试网络无带宽传输上限）
  - 数据模型
    - TPC-C基准测试数据 4W仓，规模约 4.5TB。
    - 特点：TPC-C模型包含多张宽表、大量索引和关系，其数据文件在磁盘上并非连续紧凑存储。重建过程中会产生大量随机I/O，对磁盘IOPS和延迟有较高要求，**实际有效传输速率会低于理论顺序读写带宽**。
- 基准值：
  - 全量重建耗时：与数据量成正比，在上述环境中，重建速度可达 **300-500 MB/s**（受网络、磁盘I/O、加密压缩等配置影响）。
  - 主节点性能影响：重建期间，主节点TPC-C TPMC性能下降约 **5%~15%**。
- 性能数据对比：

表 7-58 重建性能数据

| 重建方式           | 并行度 | 是否压缩 | CPU消耗<br>(备节点) | 磁盘I/O消耗<br>(备节点) | 重建完成<br>时间(包含回放时间) |
|----------------|-----|------|----------------|------------------|--------------------|
| 全量重建           | 1   | 否    | 48%            | 550MB/s          | 04:20:00           |
| 全量重建 + 并行      | 2   | 否    | 96%            | 1100MB/s         | 02:45:00           |
| 全量重建 + 并行      | 5   | 否    | 250%           | 2600MB/s         | 01:45:00           |
| 全量重建 + 并行      | 10  | 否    | 500%           | 4500MB/s         | 01:25:00           |
| 全量重建 + 并行 + 压缩 | 2   | 是    | 60%            | 700MB/s          | 03:10:00           |
| 全量重建 + 并行 + 压缩 | 5   | 是    | 140%           | 1700MB/s         | 02:05:00           |
| 全量重建 + 并行 + 压缩 | 10  | 是    | 300%           | 3500MB/s         | 01:45:00           |

每增加一个并行度，客户端（备节点）CPU会增加50%的计算资源消耗，服务端（主节点）也会同步增加计算资源和存储资源的消耗。随着并行度的提升，磁盘I/O接近上限后，对于重建的性能提升收益下降，需要根据实际业务资源消耗，选择重建方式。

接口介绍

表 7-59 GUC 参数

| 参数                                | 参数说明                                                         | 取值范围                     |
|-----------------------------------|--------------------------------------------------------------|--------------------------|
| parallel_build_thread_num         | 代表并行重建的并行度。当取值为1时表示运行原有单线程重建代码，当取值为n时会建立n条线程完成重建时的数据传输。      | 取值范围：整型（1~10）<br>默认值：1   |
| enable_parallel_build_compression | 控制是否在并行重建传输数据的过程中进行LZ4压缩，仅在parallel_build_thread_num>1情况下生效。 | 取值范围：on   off<br>默认值：off |

| 参数          | 参数说明                                                                                                                                                                                                 | 取值范围                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| async_build | 控制是否使用异步build流程重建故障DN。开启时采用异步流程重建，在剩余副本数不足多数派时，可优先恢复业务写入能力；关闭时采用同步逻辑重建，剩余副本数不足多数派时需等待数据与日志完全同步方可恢复业务。启用本功能后，若要求重建中的备DN参与多数派提交，需满足被重建的备DN在参数synchronous_standby_names配置的同步列表中，且除被重建DN外至少有一个同步列表中的DN存活 | 取值范围：on   off<br>默认值：on |

- 并行Build  
请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 发送端服务器”章节。
- 异步Build  
请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 双机复制 > 主服务器”章节。

表 7-60 Build 参数

| 参数              | 参数说明         | 取值范围                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -b, --mode=MODE | 指定重建DN备机的模式。 | <p>mode的取值：</p> <ul style="list-style-type: none"><li>• full：通过全量镜像的方式重新同步DN主机的数据目录。</li><li>• incremental：通过解析Xlog日志获取主备DN差异的数据进行增量修复备DN。</li><li>• auto（缺省值）：优先尝试进行增量重建，gs_ctl服务控制工具会根据重建失败后是否可以再增量重建而选择继续增量重建或者全量重建。以下场景，gs_ctl工具会选择全量重建的模式恢复DN实例。<ul style="list-style-type: none"><li>- 主机或备机的差异日志已回收或损坏。</li><li>- 主机或备机记录检查点信息的控制文件（即\$PGDATA/global/pg_control）丢失或损坏。</li><li>- system identifier不一致。</li><li>- 增量重建的状态文件（即\$PGDATA/global/rewind_info）丢失或损坏。</li></ul></li><li>• standby_full：使用备机重建备机或容灾场景下的首备重建灾备数据库实例内故障的备机。需要数据源备机或首备状态为Normal。</li><li>• standby_incremental：使用备机增量重建备机或容灾场景下的首备增量重建灾备数据库实例内故障的备机。需要数据源备机或首备状态为Normal。</li><li>• standby_auto：先进行备机增量重建备机，gs_ctl服务控制工具会根据重建失败后是否可以再增量重建而选择继续增量重建或</li></ul> |

| 参数                   | 参数说明                                       | 取值范围                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                      |                                            | <p>者全量重建。以下场景，gs_ctl工具会选择全量重建的模式恢复DN实例。</p> <ul style="list-style-type: none"> <li>- 主机或备机的差异日志已回收或损坏。</li> <li>- 主机或备机记录检查点信息的控制文件（即\$PGDATA/global/pg_control）丢失或损坏。</li> <li>- system identifier不一致。</li> <li>- 增量重建的状态文件（即\$PGDATA/global_rewind_info）丢失或损坏。</li> </ul> <ul style="list-style-type: none"> <li>• cross_cluster_full: 基于共享存储的同城双中心场景下的跨数据库实例全量build。</li> <li>• cross_cluster_incremental: 基于共享存储的同城双中心场景下的跨数据库实例增量build。</li> </ul> |
| -D, --pgdata=DATADIR | 指定数据目录的位置。                                 | DATADIR的取值必须为有效的数据目录。                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| -Z NODE-TYPE         | 指定当前连接数据库的运行模式。                            | NODE-TYPE的取值为coordinator或者datanode。                                                                                                                                                                                                                                                                                                                                                                                                                     |
| -C CONNECTOR         | 到源端DN的连接串，通过连接源端DN进行本节点上DN的重建。只支持备机到备机的连接。 | <p>连接串包含本节点待重建DN的IP和端口和源端DN的IP和端口，格式如下：</p> <pre>'localhost=Local DN IP localport=Local DN Port+1 remotehost=DN IP remoteport=DN Port+1'</pre>                                                                                                                                                                                                                                                                                                          |



| 参数                         | 参数说明                                   | 取值范围                                                                                                                                                                                                                                |
|----------------------------|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -M                         | -M后面需要跟SERVERMODE参数，表示在启动时指定数据库的启动模式。  | SERVERMODE的取值范围： <ul style="list-style-type: none"> <li>primary：本端以主机模式启动。</li> <li>standby：本端以备机模式启动。</li> <li>pending：本端处于等待状态，等待提升为主机或者备机。</li> <li>hadr_main_standby：本端以首备模式启动。</li> <li>cascade_standby：本端以级联备模式启动。</li> </ul> |
| -q                         | build完成后不自动启动，需要调用者启动。                 | -                                                                                                                                                                                                                                   |
| -r, --recvtimeout=INTERVAL | 指定重建过程中备机日志接收等待主机响应的超时时间。              | <ul style="list-style-type: none"> <li>取值范围：整型</li> <li>默认值：120（秒）</li> </ul>                                                                                                                                                       |
| -t, --timeout=SECS         | 等待数据库启动或者关闭完成的最大秒数。                    | <ul style="list-style-type: none"> <li>取值范围：整型</li> <li>默认值：60（秒）</li> </ul>                                                                                                                                                        |
| -U USERNAME                | 流式容灾备集群首备节点进行跨集群重建时使用的replication用户名。  | -                                                                                                                                                                                                                                   |
| -P PASSWORD                | 流式容灾备集群首备节点进行跨集群重建时使用的replication用户密码。 | -                                                                                                                                                                                                                                   |
| -Q                         | 在重建完成后进行刷写共享盘操作，仅Dorado容灾集群涉及。         | <ul style="list-style-type: none"> <li>copy_from_local：从本地往共享存储上复制。</li> <li>copy_from_share：从共享存储往本地存储上复制。</li> <li>force_copy_from_local：强制从本地往共享存储上复制。</li> </ul>                                                                |

## 7.32.3 操作指导

### 前提条件

- 软硬件要求：

- 主备节点间网络互通，且防火墙已开放数据库端口和高可用通道端口。
- 备节点磁盘空间充足，至少能容纳一份完整的数据副本。
- **依赖项：**
  - 主备节点GaussDB软件版本必须完全一致。
  - 主节点状态正常。
- **权限要求：**执行重建操作的操作系统用户须对主备节点的数据目录有读写权限，并且拥有数据库管理员权限。

## 语法

```
gs_ctl build -D DATADIR -Z NODE-TYPE [-b MODE] [-M SERVERMODE]
```

## 注意事项

- **高危操作：**重建过程中，目标备节点上的旧数据将被**全部清除**。请确保操作对象正确，避免误操作。
- **性能影响：**重建操作会大量消耗主节点的网络和存储资源，**务必在业务低峰期或维护窗口进行**。
- **监控：**重建过程耗时较长，建议在操作后持续监控集群状态、网络流量和主节点性能。

## 操作步骤

以下演示当备节点（dn\_6002）故障后，通过命令行手动触发重建的流程。

### 步骤1 登录目标节点。

以数据库用户（非root用户）登录到目标节点。

若数据库进程部署在沙箱或容器中，需要进入对应的沙箱或容器。

### 步骤2 查询目标节点信息。

```
cm_ctl query -Cvid
```

通过节点状态，确认目标节点是否存在异常（状态非Normal）。

通过节点名称，确认目标节点的数据目录路径。例如：/var/lib/engine/data1/data/dn\_6002（下述步骤以该路径作为演示）。

### 步骤3 执行重建命令。

1. 自动重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b auto -M standby
```
2. 全量重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b full -M standby
```
3. 增量重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b incremental -M standby
```
4. 以备节点做服务端，进行自动重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b standby_auto -M standby
```
5. 以备节点做服务端，进行全量重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b standby_full -M standby
```
6. 以备节点做服务端，进行增量重建。  

```
gs_ctl build -D /var/lib/engine/data1/data/dn_6002 -Z datanode -b standby_incremental -M standby
```

**步骤4** 监控重建进度。

```
gs_ctl querybuild -D /var/lib/engine/data1/data/dn_6002
```

**步骤5** 验证重建结果。

当备节点状态显示为Normal后，连接到主节点执行查询，验证备节点是否在同步流复制状态。

1. Quorum

```
SELECT application_name, state, sync_state, sync_priority FROM pg_stat_replication;
```

如果查询结果中能看到该备节点，且state为Streaming，则表明重建成功，同步正常。

2. DCF

```
SELECT channel, peer_role, peer_dcf_role, peer_state FROM gs_paxos_stat_replication;
```

如果查询结果中能看到对应IP的备节点，且peer\_state为Normal，则表明重建成功，同步正常。

----结束

## 7.32.4 常见问题

- Q1：重建任务启动失败，报错“could not connect to server: Connection timed out”。
  - **问题现象：**执行gs\_ctl build命令后，长时间等待后返回连接超时错误。
  - **原因分析：**
    - i. 主备节点之间的网络不通。
    - ii. 主节点的监听地址（listen\_addresses）未配置为备节点可访问的IP或\*。
    - iii. 防火墙或安全组规则阻止了数据库端口或流复制端口的通信。
  - **处理方法：**
    - i. 使用ping和telnet <primary\_ip> <port>命令检查网络连通性。
    - ii. 检查主节点gaussdb.conf中的listen\_addresses配置。
    - iii. 检查并放通防火墙规则。
- Q2：重建过程中，主节点性能急剧下降，业务受到影响。
  - **问题现象：**重建开始后，主数据库的查询和事务响应时间变长。
  - **原因分析：**全量数据读取和发送占用了大量磁盘I/O和网络I/O资源。
  - **处理方法：**
    - i. **紧急处理：**暂停重建任务（可通过kill进程实现，需谨慎操作）。
    - ii. **规划规避：**在业务低峰期重新执行重建任务。
    - iii. **长期优化：**考虑升级网络带宽，或使用更高性能的存储介质。
- Q3：重建进度缓慢，远低于预期速度。
  - **问题现象：**重建持续了非常长的时间，监控显示网络或磁盘利用率不高。
  - **原因分析：**
    - i. 网络带宽本身不足或存在波动。
    - ii. 主节点或备节点的磁盘I/O存在瓶颈（如慢磁盘、RAID降级）。
    - iii. 主节点业务负载极高，CPU资源争用严重，影响了Data Sender进程的调度。

- **处理方法：**
  - i. 使用iostat -x 1和nethogs等工具定位是网络还是磁盘的瓶颈。
  - ii. 检查存储系统状态（如RAID卡状态、SSD寿命）。
  - iii. 在业务低峰期进行重建。

## 7.33 升级

用户安装首版数据库后，若需新增功能或调整业务逻辑，可在保留原有数据库拓扑及数据的前提下，更新数据库二进制、系统对象及配置文件。用户可根据提供的新特性决定是否升级现有系统。

- 就地升级：需停止业务，一次性升级集群中所有节点。
- 灰度升级：支持全业务操作。优先选择部分节点升级并进入灰度观察期，验证新版本稳定性后，再升级剩余节点完成全部升级操作。
- 滚动升级：支持全业务操作。
  - 独立部署模式：按 DN 分片滚动升级，用户可指定部分分片先行升级，验证后升级剩余分片，或一次性升级所有分片。
  - 混合部署模式：按 AZ 分片滚动升级，用户可指定部分分片先行升级，验证后升级剩余分片，或一次性升级所有分片。

### 7.33.1 就地升级（非透明多写）

#### 7.33.1.1 特性概述

就地升级是一种离线升级方式，需停止业务并中断服务，一次性完成集群所有节点的升级。根据升级过程中是否涉及系统对象修改，现有升级工具的应用场景可分为两类：

- 小版本（二进制）升级：适用于补丁安装、漏洞修复等场景。
- 大版本升级：适用于新特性、新功能和模块发布等场景。

#### 功能定义

通过停止集群运行、更新系统对象及切换二进制文件，实现数据库的离线就地升级。

#### 用户场景

用户安装首版数据库后，若需新增功能或调整业务逻辑，可在保留原有数据库拓扑及数据的前提下，更新数据库的二进制文件、系统对象及配置文件。是一种稳定可靠的升级方式，保障业务连续性。

#### 规格约束

- 就地升级升级过程中会对数据库进行启停，设置数据库只读，在此期间会造成数据库业务中断。
- 集群所有节点一次性完成升级。

## 技术架构

图 7-66 就地升级技术架构

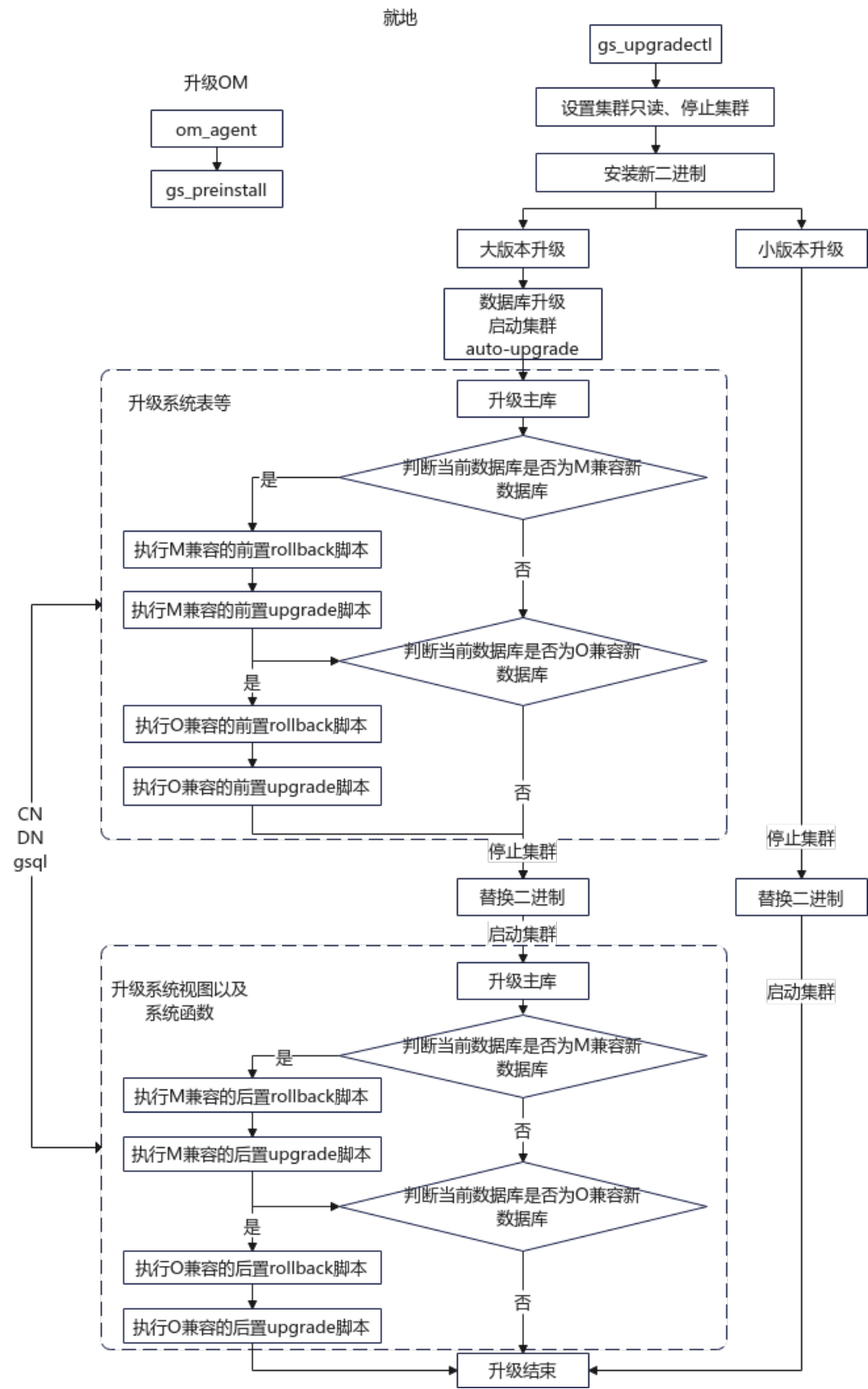
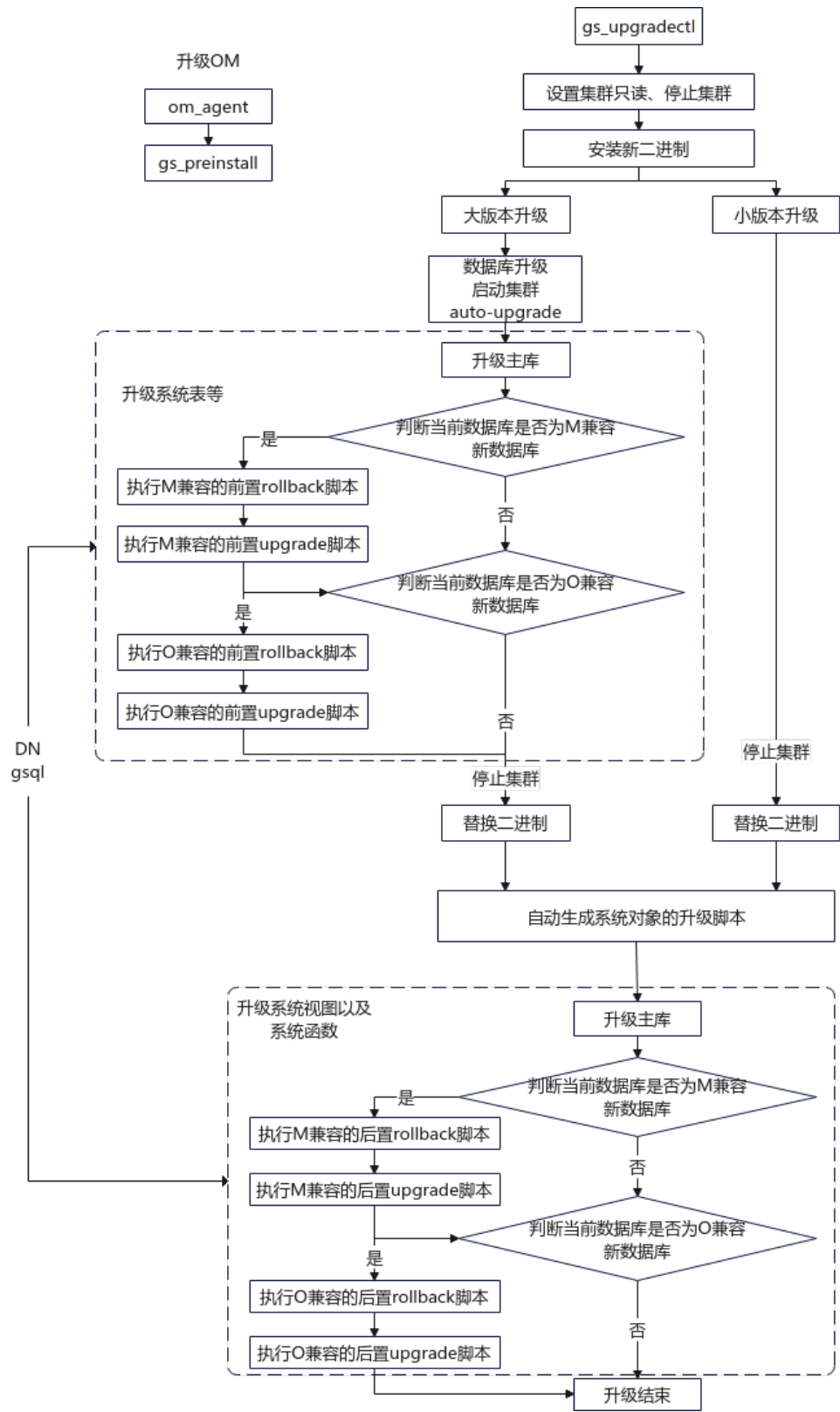


图 7-67 就地升级指定--auto\_script 参数自动生成脚本技术架构



数据库升级体系向上对接数据库运维管理平台，通过标准化升级接口及专用工具 `gs_upgradectl` 对接 `DN` 和 `gsql`，最终依托升级脚本完成数据库升级。

指定auto\_script参数升级脚本在切换二进制之后自动生成，小版本升级也会生成脚本并执行。该参数为内部升级方式过渡使用参数，故未对数据库运维管理平台开放使用。

就地升级回滚为物理回滚，指定auto\_script参数不会生效。

### 7.33.1.2 技术原理

#### 核心机制

支持两类升级模式：

1. 小版本升级：内核版本号保持不变，适用于补丁或漏洞修复。
2. 大版本升级：内核版本号变更，适用于新特性或功能发布。

内核版本号在升级文件压缩包中的version.cfg的第二行查看。

#### 性能指标

1. 部署形态为3C1D。
2. 升级路径版本跨度不超过2个C版本。
3. 业务空载。
4. 集群状态、网络及硬件条件无异常。
5. 一次性升级全部节点时，集群升级默认数据库数量总时长≤20分钟。

### 7.33.1.3 操作指导

#### 前提条件

- 环境要求：
  - 所有服务器操作系统运行状态正常。
  - 网络环境稳定无异常。
- 依赖项：已成功执行前置脚本gs\_preinstall完成环境预配置。
- 权限要求：需要使用执行gs\_preinstall时指定的集群用户进行升级操作。

#### 语法

- 显示帮助信息。  
`gs_upgradectl -? | --help`
- 显示版本号信息。  
`gs_upgradectl -V | --version`
- 选择升级策略。  
`gs_upgradectl -t chose-strategy [-l LOGFILE]`
- 升级集群。  
`gs_upgradectl -t auto-upgrade -X XMLFILE --inplace [-l LOGFILE] [--force] [--ignoreVMFaults=HOSTNAME][--auto_script]`
- 回滚升级。  
`gs_upgradectl -t auto-rollback -X XMLFILE [-l LOGFILE] [--force] [--ignoreVMFaults=HOSTNAME][--auto_script]`
- 升级提交。  
`gs_upgradectl -t commit-upgrade -X XMLFILE [-l LOGFILE] [--force_commit] [--skip_rebalance_set]`



## 说明

一旦提交操作完成，则不能再执行回滚操作。

## 核心约束

1. 业务要求
  - 就地升级需要停止业务运行。
  - 集群需满足：
    - 主备DN数据完全同步。
    - 运行状态正常。
2. 操作限制
  - 禁止与节点替换/扩容/缩容操作并行执行。
  - 不支持虚拟IP配置。
  - 升级期间禁用kerberos开关。
  - 就地升级中不支持Roach备份。
  - 禁止修改安装包内version.cfg文件。
3. 资源要求
  - DN节点磁盘使用率必须低于85%。
  - 升级后原归档日志失效。
  - 升级后原二进制目录将被清除，请不要存放个人数据。
4. 参数管控
  - 升级期间禁止修改以下GUC参数：wal\_level、max\_connections、max\_prepared\_transactions、max\_locks\_per\_transaction。
  - 如果就地升级过程中出现异常导致升级失败，且自动回滚失败时，需要用户自动执行回滚命令进行手动回滚。回滚成功后GUC参数配置自动失效。

## 进程故障升级约束

- 少数派VM故障或少数派进程故障升级约束：
  - 前置条件：
    - i. 下发升级命令之前，集群状态至少为Degraded。
    - ii. 需要下发增加--ignoreVMFaults参数的升级命令。
  - 异常处理：
    - 升级过程中，若主流程VM节点故障，升级报错为正常报错，重新将升级命令下发至正常节点即可。
    - 通过增加--ignoreVMFaults参数，被隔离的节点不会被升级。升级后需通过节点替换完成集群修复。
    - 被隔离的进程会忽略掉无法启动的错误。
- 多数派进程故障升级约束：
  - 适用范围：

- 只支持小版本升级，不支持大版本升级。
- 由于“安装目录\bin”下的gaussdb二进制文件损坏导致的故障时支持小版本强制升级，由于VM级故障导致的故障时不支持升级。
- 强制升级限制：
  - 升级不带有修复功能，升级后不保证集群一定可用。
  - 强制升级未提交或回滚前，即使集群恢复正常，也无法进行正常的升级操作。

## 就地升级操作步骤

**步骤1** 以root身份登录节点。

**步骤2** 进入安装包解压出的script目录下。

```
cd /opt/software/gaussdb_upgrade/script
```

**步骤3** 执行前置脚本gs\_preinstall。

```
./gs_preinstall -U omm -G dbgrp -X /opt/software/GaussDB_Kernel/clusterconfig.xml --alarm-type=1 --non-interactive
```

上述命令中的“clusterconfig.xml”为安装配置文件。

**步骤4** 切换至omm（集群用户）用户。

```
su - omm
```

**步骤5** 集群状态正常时，使用如下命令进行升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --inplace
```

**步骤6** 数据库状态正常时，使用如下命令自动生成升级脚本并进行升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --inplace --auto_script
```

**步骤7** 如果少数派VM或少数派进程故障，则使用如下命令进行升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --inplace --ignoreVMFaults=nodeName1,nodeName2,...
```

### 说明

*nodeName1,nodeName2,...*代表集群中需要忽略的故障的节点名列表和有进程故障的节点名列表。

**步骤8** 如果因gaussdb二进制文件损坏导致多数派进程故障，需要执行小版本强制升级，可以使用如下命令。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --inplace --force
```

**步骤9** 升级版本查询。

```
gs_ssh -c "gsql -V"
```

### 说明

若查询结果不是目标版本，可执行回滚后重新升级。

**步骤10** 检查升级集群状态。

```
cm_ctl query -Cvdi
```

查询结果的Cluster State为normal代表集群正常。

**步骤11** 升级提交。

```
gs_upgradectl -t commit-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

#### 📖 说明

一旦提交操作完成，则不能再执行回滚操作。

----结束

### 就地升级版本回滚步骤

**步骤1** 执行如下命令完成版本回滚（回滚内核代码）。回滚完成，如果需保持内核代码和om代码的一致，可以执行旧包的前置命令，依次执行**步骤1~步骤3**，若要重新升级，则需按照**步骤1~步骤3**，重新执行新版本包的前置。

```
gs_upgradectl -t auto-rollback -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

#### 📖 说明

- 如果准备回滚的节点或节点上的少数派进程故障，则使用如下命令进行回滚。  

```
gs_upgradectl -t auto-rollback -X /opt/software/GaussDB_Kernel/clusterconfig.xml --ignoreVMFaults=nodeName1,nodeName2,...
```

  
*nodeName1,nodeName2,...*代表集群中需要忽略的故障的节点名列表或有进程故障的节点名列表。  
需要在回滚时指定需要忽略的故障的节点名列表或有进程故障的节点名列表。

**步骤2** 查看回滚之后的版本号。

```
gsql -V
```

----结束

#### 7.33.1.4 常见问题

- 问题现象：  
新集群与旧集群的Commit ID完全一致。
- 原因分析：
  - 升级包版本与基线包版本相同，不允许升级。
  - 未执行升级操作直接发起Commit请求，违反操作流程规范。
- 处理方法：
  - 更换与基线包版本存在差异的有效升级包。
  - 按顺序执行操作：
    - 先完成升级流程。
    - 再执行Commit操作。

### 7.33.2 灰度升级（非透明多写）

#### 7.33.2.1 特性概述

##### 功能定义

灰度升级是一种支持优先升级部分节点的在线升级方式。通过先选择部分节点进行灰度升级，进入灰度观察期观察新版本稳定性，再升级剩余节点的方式，实现全部升级操作。

其核心实现包含以下三个维度：

- 系统表与函数兼容性
  - 针对大版本升级导致的系统表结构变更，将新/旧版本系统表结构及系统函数固化于二进制内。
  - 确保新旧版本二进制均能解析并使用新旧系统表元组。
- 二进制替换策略
  - 对于大版本升级和二进制升级涉及的新、旧二进制替换，优先灰度替换指定节点的二进制，观察系统运行稳定性。
  - 待系统运行一定时间后，再替换剩余节点的二进制。
- 操作系统与硬件升级协同
  - 涉及操作系统/硬件升级场景：  
灰度升级部分节点前，需将目标节点的主实例切换至非灰度节点。
  - 仅数据库二进制升级场景：
    - 为了尽可能降低对于业务的影响，采用同一节点两套二进制共存机制，通过软连接切换实现进程版本的切换升级。
    - 业务中断时间控制：升级全部节点，业务中断时间不超过10s；采用-g或-h的方式，每次升级部分节点时，均会产生不超过10s的业务中断。

## 用户场景

支持全业务操作。支持用户自定义灰度升级顺序，进行节点级滚动灰度升级。既可先升级备DN，升级成功后可主动进行主DN的切换，具备主动控制DN主备切换时机的能力。

## 规格约束

1. 部署要求
  - 支持形态：3C1D。
  - 版本跨度：升级路径版本相差不超过2个C版本。
2. 环境条件
  - 业务负载：升级期间需保持业务空载。
  - 系统状态：集群健康状态、网络及硬件环境需满足运行标准。
3. 性能指标

一次性升级全部节点时，默认数据库数量下，升级总时长≤20分钟。

## 技术架构

图 7-68 灰度升级技术架构

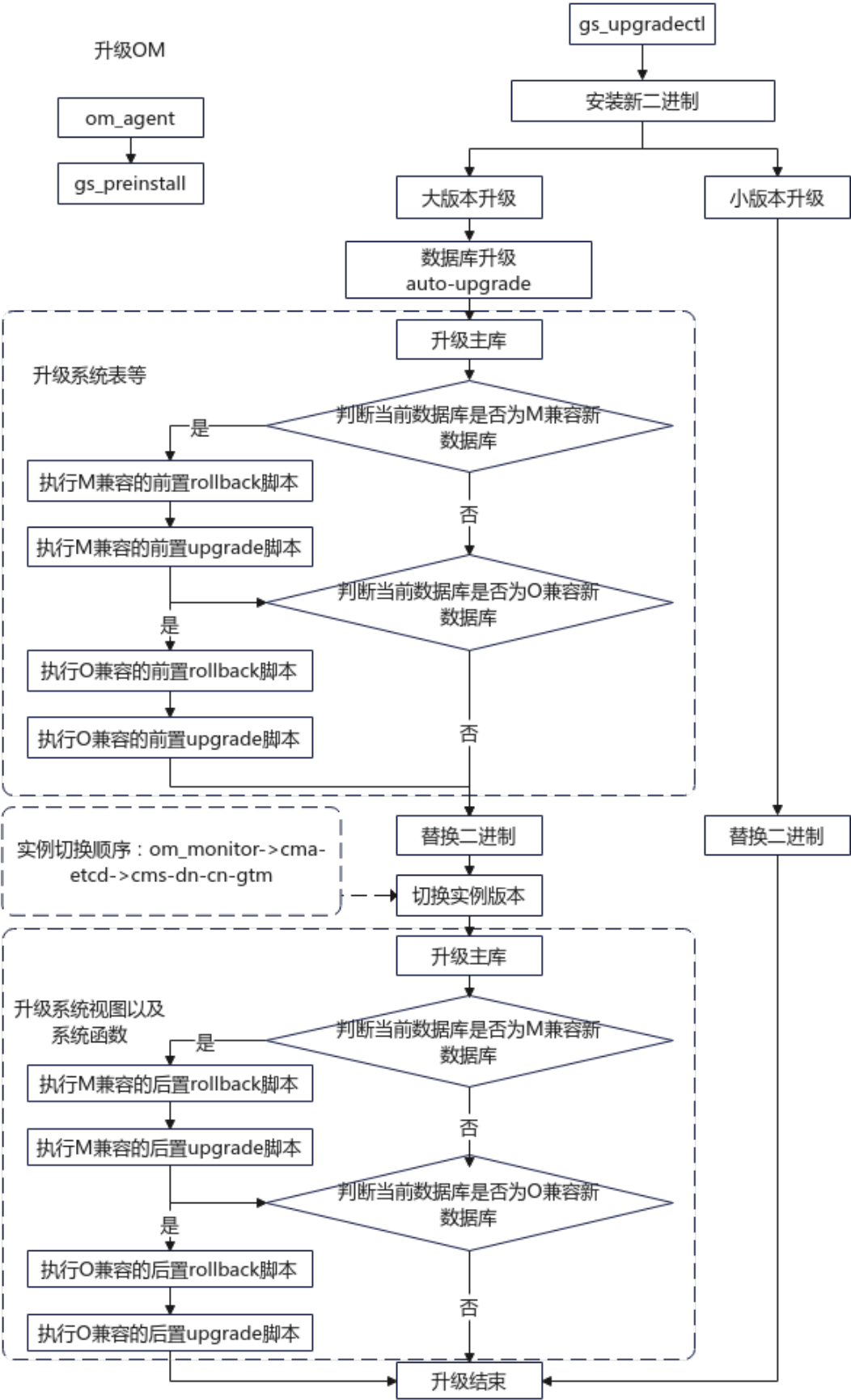
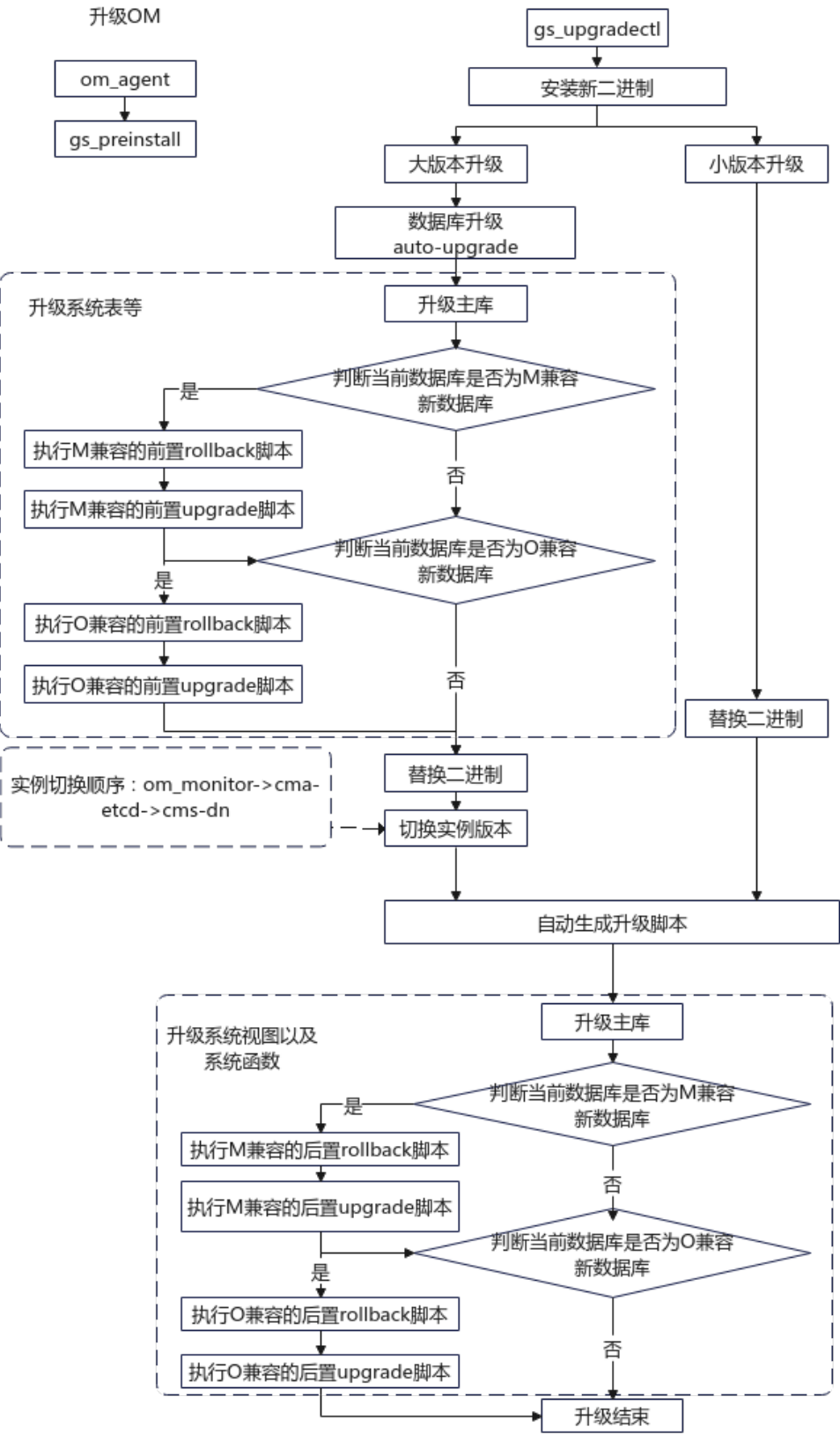


图 7-69 灰度升级指定--auto\_script 参数自动生成脚本技术架构



数据库升级体系向上对接数据库运维管理平台，通过标准化升级接口及专用工具gs\_upgradectl对接DN和gsq，最终依托升级脚本完成数据库升级。

指定auto\_script参数升级脚本在切换二进制之后自动生成，小版本升级也会生成脚本并执行。该参数为内部升级方式过度使用参数，故未对数据库运维管理平台开放使用。

为最小化二进制切换对业务的影响，采用分阶段的二进制切换机制：

1. 切换进程管理组件om\_monitor和cm\_agent；
2. 切换不影响业务的集群管理组件cm\_server和ETCD；
3. 切换业务影响组件DN、CN和GTM。

## 7.33.2.2 技术原理

### 核心机制

灰度升级通过分阶段实施可达成以下目的：

- 备DN优先升级：优先升级部分备DN节点，升级失败不影响业务连续性。
- CN节点部分升级：先升级部分CN节点，用户将业务路由到未升级的其余CN节点，保证升级过程中业务不中断。
- 按业务影响优先级升级组件：先升级业务影响小的组件节点（如ETCD、CMS），即使升级失败也可将对业务影响控制在最小范围。
- 观察期验证机制：每批节点升级后均提供升级观察期窗口，验证升级状态，动态评估升级风险。

### 性能指标

1. 升级全部节点，业务中断时间不超过10s。
2. 采用-g或-h的方式，每次升级部分节点时，业务中断时间不超过10s。
3. 部署形态3C1D，升级路径版本相差不超过2个C版本，业务空载，集群状态、网络及硬件条件无异常时，一次性升级全部节点时集群默认数据库数量升级总时长在20分钟内。

## 7.33.2.3 操作指导

### 前提条件

- 环境要求：
  - 所有服务器操作系统运行状态正常。
  - 网络环境稳定无异常。
- 依赖项：已成功执行前置脚本gs\_preinstall完成环境预配置。
- 权限要求：需要使用执行gs\_preinstall时指定的集群用户进行安装操作。

### 语法

- 显示帮助信息。  
`gs_upgradectl -? | --help`
- 显示版本号信息。  
`gs_upgradectl -V | --version`
- 选择升级策略。  
`gs_upgradectl -t chose-strategy [-l LOGFILE]`



- 升级集群。  
`gs_upgradectl -t auto-upgrade -X XMLFILE [-g NODENUMBER | -h HOSTNAME | --continue] [-l LOGFILE] [--rolling] [--grey] [--begin=ShardStartNumber] [--end=ShardEndNumber] [--force] [--ignoreVMFaults=HOSTNAME][--auto_script]`
- 回滚升级。  
`gs_upgradectl -t auto-rollback -X XMLFILE [-l LOGFILE] [--force] [--hadr_force] [--ignoreVMFaults=HOSTNAME][--auto_script]`
- 升级提交。  
`gs_upgradectl -t commit-upgrade -X XMLFILE [-l LOGFILE] [--force_commit] [--skip_rebalance_set]`

#### 说明

一旦提交操作完成，则不能再执行回滚操作。

## 核心约束

1. 业务要求
  - 灰度升级支持全业务操作。
  - 集群需满足：
    - 主备DN数据完全同步。
    - 运行状态正常。
  - 业务压力大的情况下建议开启流控。
2. 操作限制
  - 禁止与节点替换/扩容/缩容操作并行执行。
  - 不支持虚拟IP配置。
  - 升级期间禁用kerberos开关。
  - 禁止修改安装包内version.cfg文件。
  - 选择灰度升级方式进行升级时，升级过程中不支持节点替换/修改IP/扩容/缩容/hotpatch，部分节点升级成功后不建议执行节点修复；不支持增加CN。
3. 资源要求
  - DN节点磁盘使用率必须低于85%。
  - 升级后原归档日志失效。
  - 升级后原二进制目录将被清除，请不要存放个人数据。
4. 参数管控
  - 升级期间禁止修改以下GUC参数：wal\_level、max\_connections、max\_prepared\_transactions、max\_locks\_per\_transaction。
  - 如果灰度升级过程中出现异常导致升级失败，需用户手动回滚，且回滚成功后才能进行下一次升级。回滚成功后升级过程中设置的GUC参数将失效。
5. Roach备份与升级的交互：
  - 灰度升级观察期间支持Roach备份。
  - 灰度升级部分节点时，参与备份的节点大版本要求一致，否则会导致备份失败。DISK备份会校验所有节点。OBS、NAS、XBSA备份会校验参与备份的节点，关于备份节点的选取规则为：
    - 备份下发节点。

- GTM：主GTM。
- CMS：主CMS。
- DN：主机备份为每个分片的主机，备机备份为每个分片上正常的回放速度最快的备DN。
- CN：不开归档时为第一个正常的CN，开归档时为所有正常的CN。

## 进程故障升级约束

- 少数派VM故障或少数派进程故障升级约束：
  - 前置条件：
    - i. 下发升级命令之前，集群状态至少为Degraded。
    - ii. 需要下发增加--ignoreVMFaults参数的升级命令。
  - 异常处理：
    - 升级过程中，若主流程VM节点故障，升级报错为正常报错，重新将升级命令下发至正常节点即可。
    - 通过增加--ignoreVMFaults参数，被隔离的节点不会被升级。升级后需通过节点替换完成集群修复。
    - 被隔离的进程会忽略掉无法启动的错误。
- 多数派VM故障或多数派进程故障升级约束：
  - 适用范围：
    - 只支持小版本升级，不支持大版本升级。
    - 由于“安装目录\bin”下的gaussdb二进制文件损坏导致的多数派进程故障支持小版本强制升级，由于VM级故障导致的故障不支持升级。（内核版本号不变的升级方式小版本升级，否则就是大版本升级。内核版本号在升级文件压缩包中的version.cfg的第二行查看）。
  - 强制升级限制：
    - 升级不带有修复功能，升级后不保证集群一定可用。
    - 强制升级未提交或回滚前，即使集群恢复正常，也无法进行正常的升级操作。
    - 支持回滚操作。回滚成功后，不保证集群可用。

## 灰度升级操作步骤

**步骤1** 以root身份登录节点。

**步骤2** 进入安装包解压出的script目录下。

```
cd /opt/software/gaussdb_upgrade/script
```

**步骤3** 执行前置脚本gs\_preinstall。

```
./gs_preinstall -U omm -G dbgrp -X /opt/software/GaussDB_Kernel/clusterconfig.xml --alarm-type=1 --non-interactive
```

上述命令中的“clusterconfig.xml”为安装配置文件。

**步骤4** 切换至omm（集群用户）用户。

```
su - omm
```

**步骤5** 灰度升级可以指定多种方式进行（两种方式均可选择，如果需要指定特定节点时，可以选择-h方式）：

1. 先升级指定特定节点，再升级其余节点方式，请参考[步骤6](#)、[步骤7](#)、[步骤9](#)。
2. 先升级指定数量的节点，再升级其余节点方式，请参考[步骤8](#)、[步骤9](#)。

**步骤6** 灰度升级1个指定节点“pekpexper00001”。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -h pekpexper00001
```

#### 说明

- 如果准备升级的节点或节点上的少数派进程故障，则使用如下命令进行升级。  

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -h pekpexper00001 --ignoreVMFaults=nodeName1,nodeName2,...
```

  
=nodeName1,nodeName2,...代表集群中需要忽略的故障的节点名列表或有进程故障的节点名列表。

需要在第一次灰度升级时指定需要忽略的故障的节点名列表或有进程故障的节点名列表。后续灰度升级时不需要再次指定，除非有新增的故障节点和进程故障节点。

**步骤7** 灰度升级3个指定节点“pekpexper00002、pekpexper00003、pekpexper00004”。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -h pekpexper00002,pekpexper00003,pekpexper00004
```

升级成功后，执行[步骤9](#)升级其余节点。

**步骤8** 灰度升级3个节点。（选择先灰度升级集群中部分节点时，-g指定节点个数不能超过5；选择灰度升级集群中全部节点时，-g后面指定集群全部节点个数。）

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -g 3
```

升级成功后，执行[步骤9](#)升级其余节点。

**步骤9** 在[步骤8](#)升级成功的节点上升级数据库实例所有剩余节点。

- 使用--continue参数升级数据库实例所有剩余节点。  

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --continue
```
- 使用--continue及--auto\_script参数升级数据库实例所有剩余节点。  

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --continue --auto_script
```

**步骤10** 独立部署模式下，灰度升级可采用分片升级的方式，对集群进行灰度升级。用户可以一次指定任意多个分片进行升级，升级命令如下。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=ShardStartNumber --end=ShardEndNumber --grey
```

如果指定升级第1个分片，命令如下：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=1 --end=1 --grey
```

混合部署模式下，灰度升级可采用AZ级的升级方式，对集群进行升级。用户可以指定任意AZ的所有节点进行升级，升级命令如下：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -h nodename1,nodename2... --rolling --grey
```

注意：命令中的nodename必须要是一个AZ或者多个AZ的所有节点。若一次性升级集群中所有节点，假设节点总数为n，则可使用如下组合命令。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -g n --rolling --grey
```

**步骤11** 使用--auto\_script参数通过自动生成升级脚本的方式灰度升级全部3个节点。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -g 3 --auto_script
```

#### 步骤12 升级版本查询。

```
gs_ssh -c "gsql -V"
```

##### 说明

若查询结果不是目标版本，可执行回滚后重新升级。

#### 步骤13 检查升级集群状态。

```
cm_ctl query -Cdev
```

查询结果的Cluster State为normal代表集群正常。

#### 步骤14 升级提交。

```
gs_upgradectl -t commit-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

##### 说明

一旦提交操作完成，则不能再执行回滚操作。

----结束

### 灰度升级回滚步骤

**步骤1** 执行如下命令完成版本回滚（回滚内核代码）。回滚完成，如果需保持内核代码和om代码的一致，可以执行旧包的前置命令，依次执行**步骤1~步骤3**。若要重新升级，则需按照**步骤步骤1~步骤3**，重新执行新版本包的前置。

```
gs_upgradectl -t auto-rollback -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

##### 说明

- 如果准备回滚的节点或节点上的少数派进程故障，则使用如下命令进行回滚。  
gs\_upgradectl -t auto-rollback -X /opt/software/GaussDB\_Kernel/clusterconfig.xml --  
ignoreVMFaults=nodeName1,nodeName2,...  
=nodeName1,nodeName2,...代表集群中需要忽略的故障的节点名列表或有进程故障的节点名列表。  
需要在回滚时指定需要忽略的故障的节点名列表或有进程故障的节点名列表。

**步骤2** 查看回滚之后的版本号。

```
gsq -V
```

----结束

### 7.33.2.4 常见问题

- Q：从A版本升级至B版本未执行提交操作，继续升级至C版本时触发Commit ID校验拦截。
- A：请在A版本升级到B版本提交后，再从B升级到C。

## 7.33.3 滚动升级

### 7.33.3.1 特性概述

#### 功能定义

滚动升级允许用户指定部分DN分片或AZ分片先升级，进入滚动观察期观察新版本是否运行正常，再升级剩余分片，实现全部升级操作。也可以一次升级所有分片。

## 用户场景

滚动升级支持全业务操作，具体实现方式根据部署模式区分：

- 独立部署模式：采用滚动升级DN分片的方式对集群进行升级。
- 混合部署模式：采用滚动升级AZ分片的方式对集群进行升级。要求输入的节点必须是一个AZ或者多个AZ的所有节点。

## 规格约束

- 按DN分片滚动升级要求集群是独立部署的，每个实例都部署在单独的节点上。
- 升级每个分片时，都会产生不超过10s的业务中断。升级最后一个分片，会中断2~3次业务。

### 7.33.3.2 技术原理

#### 技术架构

图 7-70 滚动升级技术架构

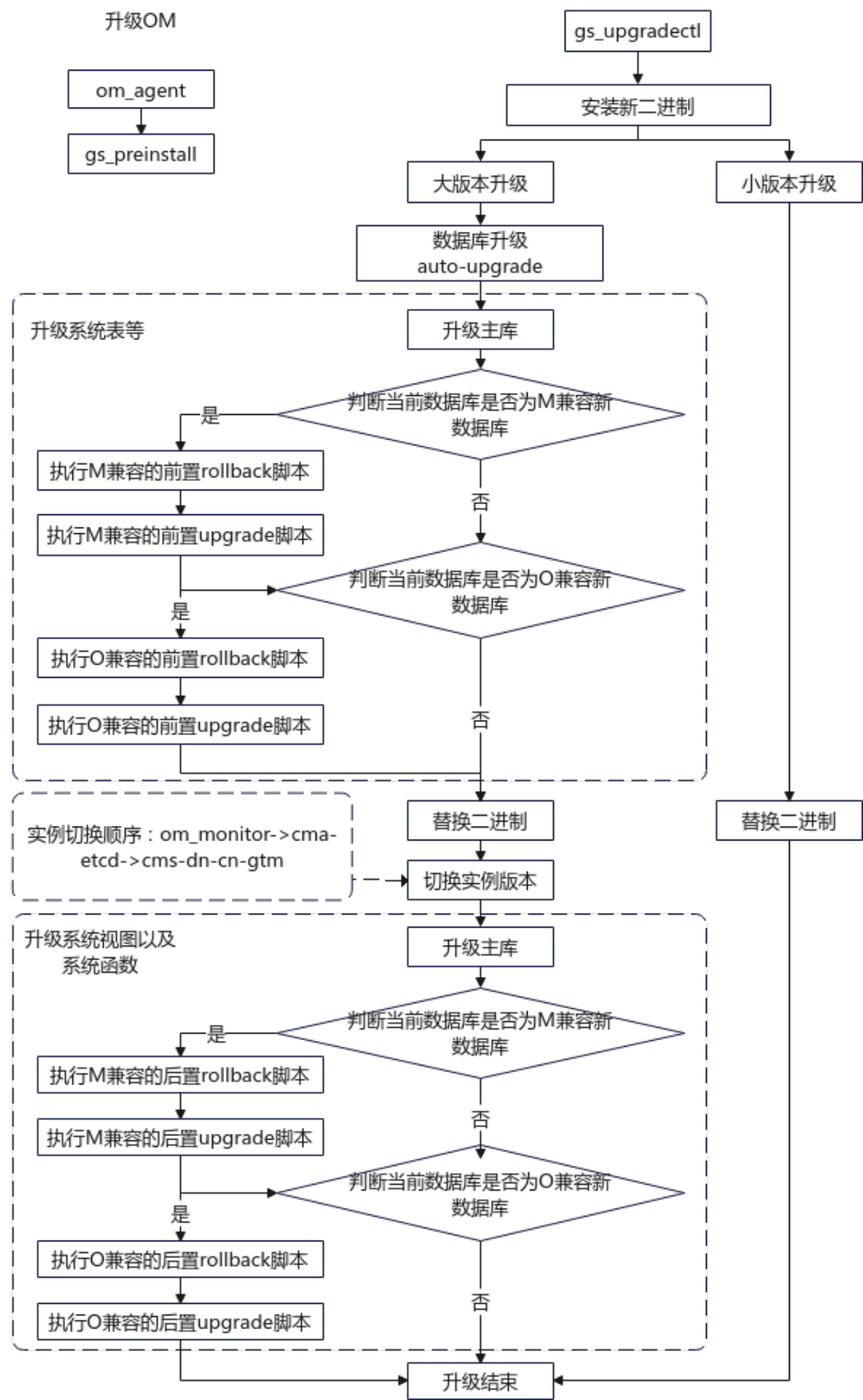
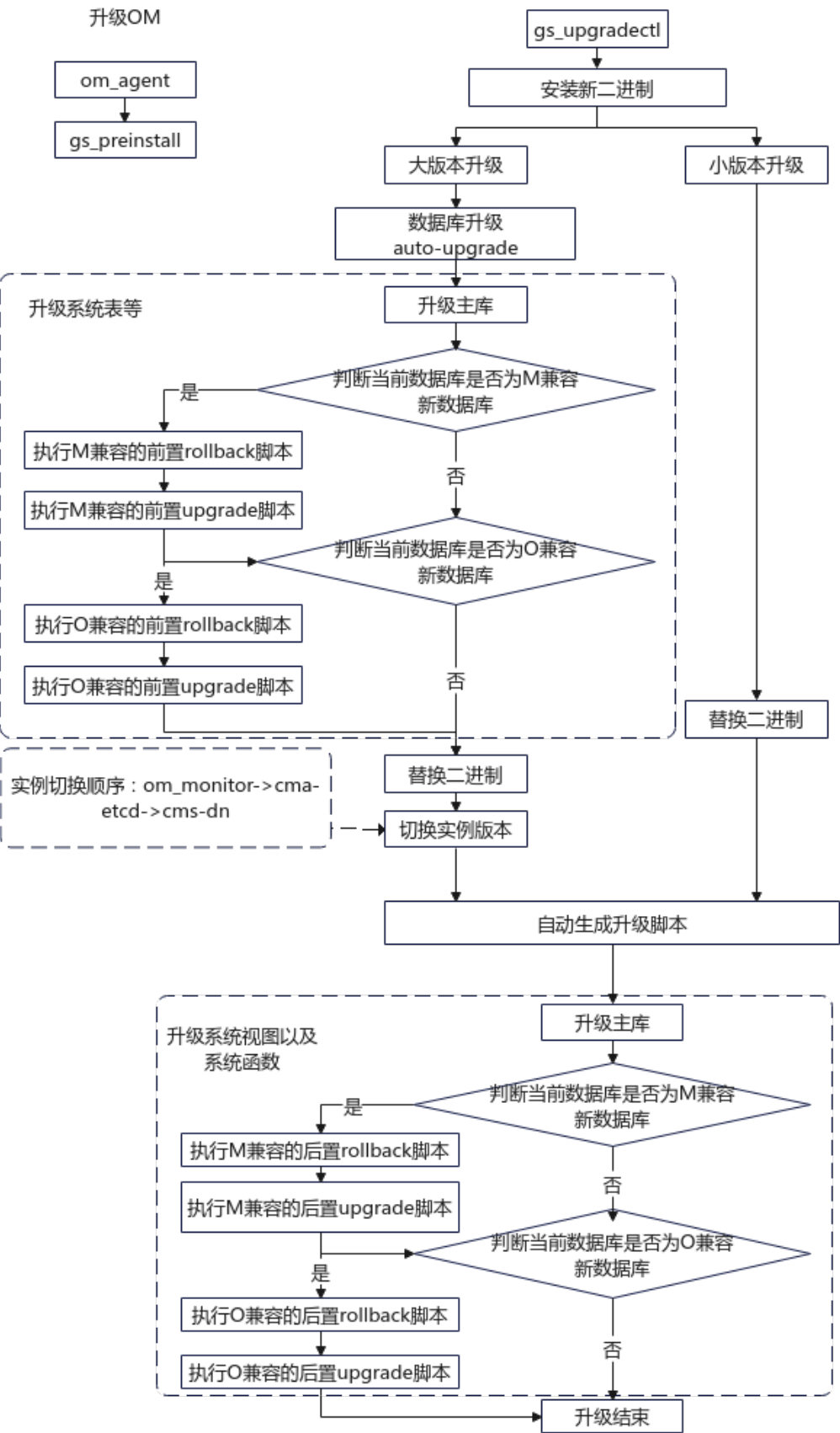


图 7-71 滚动升级指定--auto\_script 参数自动生成脚本技术架构





数据库升级体系向上对接数据库运维管理平台，通过标准化升级接口及专用工具gs\_upgradectl对接DN和gsqsl，最终依托升级脚本完成数据库升级。

指定auto\_script参数升级脚本在切换二进制之后自动生成，小版本升级也会生成脚本并执行。该参数为内部升级方式过度使用参数，故未对数据库运维管理平台开放使用。

为最小化二进制切换对节点业务的影响，采用分阶段的二进制切换策略：

1. 首先切换进程管理组件om\_monitor和cm\_agent。
2. 其次切换不影响业务的集群管理组件cm\_server和ETCD。
3. 最后切换业务影响组件DN、CN和GTM。

对于实例切换，不同实例采用不同的滚动方式：

- 一主多备实例：备机优先切换后执行主机切换。
- 具体滚动升级方式如下：
  - CN：多轮升级，每轮单节点切换。
  - GTM/CMS：两轮升级。首轮批量升级所有备机，次轮执行主机升级。
  - DN/ETCD：多轮升级。先升级少数派节点，再升级剩余少数派节点，最后升级主节点。具体策略：初始轮次升级两个节点（各一个）；当节点总数为奇数时，升级(n/2)个节点；为偶数时，升级(n/2 - 1)个节点。

## 性能指标

1. 单分片升级业务中断时长≤10秒。
2. 升级最后一个分片时会中断业务2 ~ 3次。
3. 部署形态3C1D，升级路径版本相差不超过2个C版本。
4. 全节点升级时效：业务空载，集群状态、网络及硬件条件无异常时，一次性升级全部节点时集群默认数据库数量升级总时长在20min内。

### 7.33.3.3 操作指导

#### 前提条件

- 环境要求：
  - 所有服务器操作系统运行状态正常。
  - 网络环境稳定无异常。
- 依赖项：已成功执行前置脚本gs\_preinstall完成环境预配置。
- 权限要求：需要使用执行gs\_preinstall时指定的集群用户进行安装操作。

#### 语法

- 显示帮助信息。  
gs\_upgradectl -? | --help
- 显示版本号信息。  
gs\_upgradectl -V | --version
- 选择升级策略。  
gs\_upgradectl -t chose-strategy [-l LOGFILE]
- 升级集群。

```
gs_upgradectl -t auto-upgrade -X XMLFILE [-g NODENUMBER | -h HOSTNAME | --continue] [-l LOGFILE] [--rolling]
[--grey] [--begin=ShardStartNumber] [--end=ShardEndNumber] [--force] [--
ignoreVMFaults=HOSTNAME][--auto_script]
```

- 回滚升级。

```
gs_upgradectl -t auto-rollback -X XMLFILE [-l LOGFILE] [--force] [--hadr_force] [--
ignoreVMFaults=HOSTNAME][--auto_script]
```

- 升级提交。

```
gs_upgradectl -t commit-upgrade -X XMLFILE [-l LOGFILE] [--force_commit] [--skip_rebalance_set]
```

### 说明

- 一旦提交操作完成，则不能再执行回滚操作。

## 核心约束

### 1. 业务要求

- 集群需满足：
  - 主备DN数据完全同步。
  - 运行状态正常。
- 业务压力大的情况下建议开启流控。

### 2. 操作限制

- 禁止与节点替换/扩容/缩容操作并行执行。
- 不支持虚拟IP配置。
- 升级期间禁用kerberos开关。
- 禁止修改安装包内version.cfg文件。
- 选择灰度升级方式进行升级时，升级过程中不支持节点替换/修改IP/扩容/缩容/hotpatch，部分节点升级成功后不建议执行节点修复；不支持增加CN。

### 3. 资源要求

- DN节点磁盘使用率必须低于85%。
- 升级后原归档日志失效。
- 升级后原二进制目录将被清除，请不要存放个人数据。

### 4. 参数管控

- 升级期间禁止修改以下GUC参数：wal\_level、max\_connections、max\_prepared\_transactions、max\_locks\_per\_transaction。
- 如果灰度升级过程中出现异常导致升级失败，需用户手动回滚，且回滚成功后才能进行下一次升级。回滚成功后升级过程中设置的GUC参数将失效。

### 5. Roach备份与升级的交互：

- 灰度升级观察期间支持Roach备份。
- 灰度升级部分节点时，参与备份的节点大版本要求一致，否则会导致备份失败。DISK备份会校验所有节点。OBS、NAS、XBSA备份会校验参与备份的节点，关于备份节点的选取规则为：
  - 备份下发节点。
  - GTM：主GTM。
  - CMS：主CMS。

- DN：主机备份为每个分片的主机，备机备份为每个分片上正常的回放速度最快的备DN。
- CN：不开归档时为第一个正常的CN，开归档时为所有正常的CN。

## 进程故障升级约束

- 少数派VM故障或少数派进程故障升级约束：
  - 前置条件：
    - i. 下发升级命令之前，集群状态至少为Degraded。
    - ii. 需要下发增加--ignoreVMFaults参数的升级命令。
  - 异常处理：
    - 升级过程中，若主流程VM节点故障，升级报错为正常报错，重新将升级命令下发至正常节点即可。
    - 通过增加--ignoreVMFaults参数，被隔离的节点不会被升级。升级后需通过节点替换完成集群修复。
    - 被隔离的进程会忽略掉无法启动的错误。
- 多数派VM故障或多数派进程故障升级约束：
  - 适用范围：
    - 只支持小版本升级，不支持大版本升级。
    - 由于“安装目录\bin”下的gaussdb二进制文件损坏导致的多数派进程故障支持小版本强制升级，由于VM级故障导致的故障不支持升级。（内核版本号不变的升级方式为小版本升级，否则就是大版本升级。内核版本号在升级文件压缩包中的version.cfg的第二行查看）。
  - 强制升级限制：
    - 升级不带有修复功能，升级后不保证集群一定可用。
    - 强制升级未提交或回滚前，即使集群恢复正常，也无法进行正常的升级操作。
    - 支持回滚操作。回滚成功后，不保证集群可用。

## 滚动升级操作步骤

**步骤1** 以root身份登录节点。

**步骤2** 进入安装包解压出的script目录下。

```
cd /opt/software/gaussdb_upgrade/script
```

**步骤3** 执行前置脚本gs\_preinstall。

```
./gs_preinstall -U omm -G dbgrp -X /opt/software/GaussDB_Kernel/clusterconfig.xml --alarm-type=1 --non-interactive
```

上述命令中的“clusterconfig.xml”为安装配置文件。

**步骤4** 切换至omm（集群用户）用户。

```
su - omm
```

**步骤5** 独立部署模式下，滚动升级采用分片升级的方式，对集群进行升级。用户可以一次指定任意多个分片进行升级，升级命令如下。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=ShardStartNumber --end=ShardEndNumber
```

如果指定升级第1个分片，命令如下：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=1 --end=1
```

混合部署模式下，滚动升级采用AZ级的升级方式，对集群进行升级。用户可以指定任意AZ的所有节点进行升级，升级命令如下：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -h nodename1,nodename2... --rolling
```

注意：命令中的nodename必须要是一个AZ或者多个AZ的所有节点。若一次性升级集群中所有节点，假设节点总数为n，则可使用如下组合命令：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -g n --rolling
```

#### 说明

- 如果准备升级的节点或节点上的少数派进程故障，则使用如下命令进行升级。  

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=1 --end=1 --ignoreVMFaults=nodeName1,nodeName2,...
```

  
=nodeName1,nodeName2,...代表集群中需要忽略的故障的节点名列表或有进程故障的节点名列表。  
需要在第一次滚动升级时指定需要忽略的故障的节点名列表或有进程故障的节点名列表。后续滚动升级时不需要再次指定，除非有新增的故障节点和进程故障节点。

如果指定升级第3、4、5分片，命令如下：

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml --begin=3 --end=5
```

本次指定的分片升级成功后，用户可以继续使用本命令，指定未升级的分片进行升级操作，直到完成所有分片的升级。

**步骤6** 使用--auto\_script参数通过自动生成升级脚本的方式滚动升级全部3个节点。

```
gs_upgradectl -t auto-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml -g n --rolling --auto_script
```

**步骤7** 升级版本查询。

```
gs_ssh -c "gsql -V"
```

**步骤8** 检查升级集群状态。

```
cm_ctl query -Cdevi
```

查询结果的Cluster State为normal代表集群正常。

**步骤9** 升级提交。

```
gs_upgradectl -t commit-upgrade -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

#### 说明

一旦提交操作完成，则不能再执行回滚操作。

----结束

## 滚动升级回滚步骤

**步骤1** 执行如下命令完成版本回滚（回滚内核代码）。回滚完成，如果需保持内核代码和om代码的一致，可以执行旧包的前置命令，依次执行[步骤1](#)~[步骤3](#)。若要重新升级，则需按照[步骤1](#)~[步骤3](#)，重新执行新版本包的前置。

```
gs_upgradectl -t auto-rollback -X /opt/software/GaussDB_Kernel/clusterconfig.xml
```

#### 说明

- 如果准备回滚的节点或节点上的少数派进程故障，则使用如下命令进行回滚。  

```
gs_upgradectl -t auto-rollback -X /opt/software/GaussDB_Kernel/clusterconfig.xml --
ignoreVMFaults=nodeName1,nodeName2,...
```

  
=nodeName1,nodeName2,...代表集群中需要忽略的故障的节点名列表或有进程故障的节点名列表。  
需要在回滚时指定需要忽略的故障的节点名列表或有进程故障的节点名列表。

#### 步骤2 查看回滚之后的版本号。

```
gsql -V
```

----结束

### 7.33.3.4 常见问题

- Q：从A版本升级至B版本未执行提交操作，继续升级至C版本时触发Commit ID校验拦截。
- A：请在A版本升级到B版本提交后，再从B升级到C。

## 7.34 节点修复、替换

### 7.34.1 特性概述

随着企业数据规模持续扩大，节点数量急剧增长，硬件损坏概率相应攀升，物理节点替换修复已成为日常运维的常规操作。传统离线节点替换方案难以满足业务连续性需求，频繁的业务中断将给客户造成重大损失。当前主流数据库产品在节点替换过程中，普遍存在需中断业务或仅支持有限操作的局限性，无法适应海量数据场景下常态化物理节点替换需求。在线节点替换特性解决了上述痛点，显著提升数据库运行可靠性，为用户提供持续稳定的数据服务。

#### 功能定义

当数据库集群内某节点因硬件故障或组件异常导致数据库实例不可用时，通过本功能可实现数据库实例的恢复。

#### 用户场景

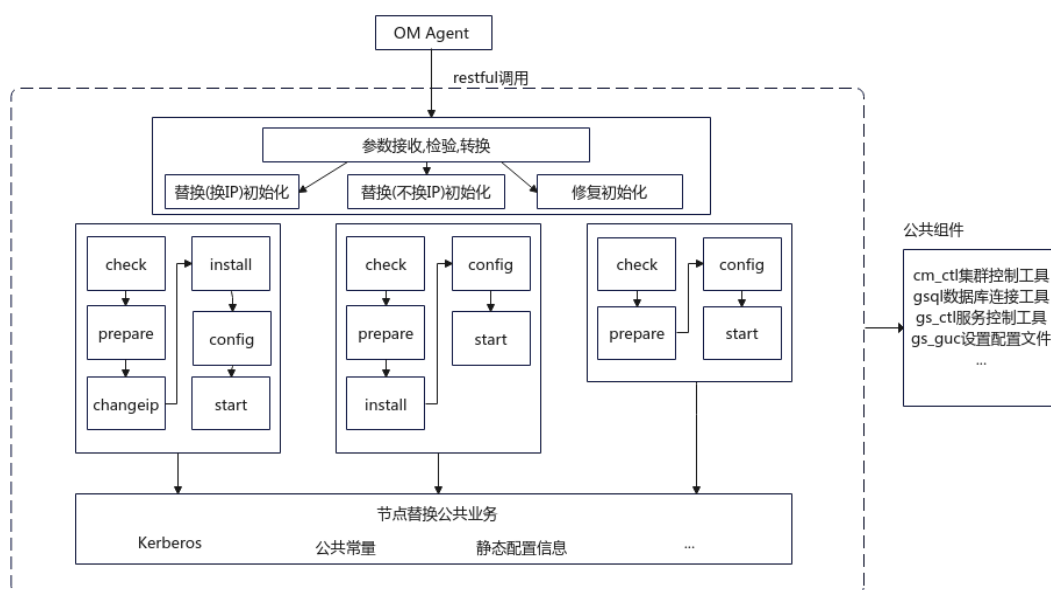
- 节点替换（不换IP）：需调整新节点的主机名与IP至与故障节点一致后执行替换。适用于替换前后节点的主机名与IP一致的场景，具有前置条件少、实施效率高的特点。
- 节点替换（换IP）：无需修改新节点的主机名与IP，直接执行替换操作。该方案等效于集群温备方案，可在不调整新节点IP与主机名的情况下，将处于运行态的节点直接替换为新节点。适用于替换节点的主机名与IP和被替换节点不一致的场景，或高性能节点替换为一个性能较差的节点的场景，实施需满足较多前置条件。
- 节点修复：对异常组件执行重新安装及数据重建操作，可实现故障节点快速恢复。

## 规格约束

- 分布式单节点不支持该工具。
- 节点修复、替换操作需在健康节点发起，系统保证同一时间仅存在一个操作进程，禁止多节点并行操作。

## 技术架构

节点修复、替换的端到端体系架构概述：从端到端的体系架构来看，节点修复、节点替换向上对接管控，为管控提供RESTful接口；通过公共组件提供的各种功能实现集群的组件修复或拓扑结构变更。如下图所示：



## 7.34.2 技术原理

### 核心机制

节点修复与节点替换操作可在线完成，无需中断业务，主要依赖于数据库配置的动态加载生效及数据build后实例正常启动。具体流程步骤如下：

- 步骤1** （可选，节点替换使用）集群管理组件CM将故障节点IP从集群中剔除，并将新节点IP加入集群内。此操作主要包括配置文件刷新、系统表刷新等，所有配置均动态加载生效。
  - 步骤2** （可选，节点替换使用）在新节点上创建安装目录、日志目录、解压组件二进制、证书生成等。此步骤完成后，新节点所有组件均转化为待修复状态，等待下一步修复操作。
  - 步骤3** 执行故障组件修复，主要包括待修复组件初始化、配置文件生成、数据全量/增量build。
  - 步骤4** 启动修复后的组件，并检测所有组件状态均已恢复正常。
- 结束

## 7.34.3 操作指导

### 前提条件

- 节点修复
  - 该工具支持对故障的ETCD/DN/CN/CM/GTM进行节点修复。
  - 若参数enable\_synclist\_single\_inst值为off，则在一主多备场景下，每组DN中至少保证多数派实例正常工作，仅允许少数派实例故障。此时通过gs\_replace修复故障DN不影响业务。修复DN实例时，需确保DN环中存在CM可正常监控的主DN，以保障数据一致性。
  - 集群内分别至少存在一个正常运行的实例含：CM Server、GTM、CN。
  - 若集群中部署有ETCD，则正常的ETCD数量必须大于ETCD总数量的一半。
  - 修复CN实例时，集群内必须至少存在一个CN状态为Normal，否则修复不成功。
  - 忽略故障节点升级后，若故障节点因升级导致与正常节点版本不一致，不能直接进行修复，需改用节点替换功能。
  - 修复多个CN时，最大并发修复数量与正常CN的max\_wal\_senders参数有关，单次操作允许修复的CN数量为max\_wal\_senders参数值的一半。
- 节点替换（不换IP）
  - 若参数enable\_synclist\_single\_inst值为off，则在一主多备场景下，每组DN中至少保证多数派实例正常工作，仅允许少数派实例故障。此时通过gs\_replace节点替换不影响业务。修复DN实例时，需确保DN环中存在CM可正常监控的主DN，以保障数据一致性。
  - 使用gs\_replace替换DN实例时，被替换的DN不能是主DN，否则替换不成功。
  - 集群内分别至少存在一个可正常运行的实例含：CM Server、GTM、CN。
  - 如果集群中已部署ETCD，则正常的ETCD个数必须大于ETCD总个数的一半。
  - 新节点的IP、主机名等信息必须与原故障节点保持一致。
  - 不支持备DN节点替换为日志节点，且不支持日志节点替换为备DN节点。
- 节点替换（换IP）
  - 若参数enable\_synclist\_single\_inst值为off，则在一主多备场景下，每组DN中至少保证多数派实例正常工作，仅允许少数派实例故障，此时通过gs\_replace替换故障DN不影响业务。修复DN实例时，需确保DN环中存在CM可正常监控的主DN，以保障数据一致性。
  - 使用gs\_replace替换DN实例时，被替换的DN不能是主DN，否则替换不成功。
  - 集群内分别至少存在一个可正常运行的实例含：CM Server、GTM、CN。
  - 如果集群中已部署ETCD，则正常的ETCD个数必须大于ETCD总个数的一半。
  - 用户可执行单节点卸载操作，以移除故障节点。具体操作包括：卸载需被替换的节点或手动清理损坏节点上的数据库相关目录及残留进程。
  - 需提前准备基于新节点IP的集群配置文件，该文件仅调整原配置中的IP和主机名，其余内容保持不变。
  - 混合部署不支持备DN节点替换为日志节点，且不支持日志节点替换为备DN节点。



## 注意事项

- 节点修复

- 若集群中存在状态非Deleted的故障CN，那么在修复过程中用户执行DDL会报错，但DML可以正常执行，其他业务场景不受影响。

### 说明

修复CN的过程中会短暂锁集群。锁集群期间，若用户下发包含显式事务启动的DDL语句，该操作将进入等待状态。集群解锁后或等待时间超过20分钟会报错。如执行创建临时表操作，集群解锁后会报错“Don't support temp table when need reconnect pooler”。

- 若故障实例所在节点的安装目录下（\$GAUSSHOME/bin/）的二进制文件损坏或丢失，则不能通过替换实例进行修复。需复制其他正常节点对应的二进制文件到该节点，或将该节点卸载后再通过节点替换进行修复。
- 节点修复操作将修复故障节点下的全部故障实例。
- 在config阶段优先修复指定节点的CM Agent组件，以确保能正常获取节点实例状态。如果节点上的某些实例被人为停止，在CM Agent组件修复成功之后，这些原来正常的实例会被正常拉起，而不会被修复。如果在一定时间内拉起失败，这些实例将会被修复。
- 修复故障实例过程中系统将关闭“自动剔除故障CN”功能，修复完成后系统再次打开该功能。因此建议在开始修复前确认故障的CN已经被自动剔除（即故障的CN状态为Deleted），否则在修复过程中用户执行DDL会报错。
- 若修复过程中因步骤错误导致失败，需使用原命令重试修复流程。部分实例可能已修复并启动，但状态未完全恢复，需继续执行修复直至所有实例状态正常。

- 节点替换（不换IP）

- 若集群中存在状态非Deleted的故障CN，那么在修复过程中用户执行DDL会报错，但DML可以正常执行，其他业务场景不受影响。因此建议在开始替换前确认故障的CN已经被自动剔除（即故障的CN状态为Deleted）。

### 说明

修复CN的过程中会短暂锁集群。锁集群期间，若用户下发包含显式事务启动的DDL语句，该操作将进入等待状态。集群解锁后或等待时间超过20分钟会报错。如执行创建临时表操作，集群解锁后会报错“Don't support temp table when need reconnect pooler”。

- 当故障节点与新节点为同一节点时，需在替换操作前执行单节点卸载流程。

- 节点替换（换IP）

- 若集群中存在状态非Deleted的故障CN，那么在修复过程中用户执行DDL会报错，但DML可以正常执行，其他业务场景不受影响。因此建议在开始替换前确认故障的CN已经被自动剔除（即故障的CN状态为Deleted）。

### 说明

修复CN的过程中会短暂锁集群。锁集群期间，若用户下发包含显式事务启动的DDL语句，该操作将进入等待状态。集群解锁后或等待时间超过20分钟会报错。如执行创建临时表操作，集群解锁后会报错“Don't support temp table when need reconnect pooler”。

- 若某节点完全故障且其他节点存在部分实例故障，在满足节点替换（换IP）条件的前提下，替换成功后仍需对其他部分故障节点执行节点修复操作。
- 若集群中部署ETCD，部署ETCD的节点轮流被替换且未重启过主备GTM，则在最后一个节点被替换完成后，需要重启主备GTM以保障集群可靠性。



示例：假设集群中部署3个ETCD分别在节点node1、node2、node3上，替换流程如下：

- node1故障替换为node4，未重启主备GTM。
- node2故障替换为node5，未重启主备GTM。
- node3故障替换为node6，替换完成后需重启集群主备GTM。
- 若故障节点处于断连状态，执行替换后需由运维人员手动清理以下残留内容：
  - 节点上的数据库数据。
  - 残留的数据库进程、实例目录及环境变量（清理路径需与新节点配置保持一致，可类比查看新节点的以上信息对该节点进行清理）。
- 在备DN节点替换为日志节点的场景中，必须完成所有分片替换操作后，方可执行扩容、升降副本或缩容等运维操作。

## 示例

### 示例一 节点修复

以修复节点plat1、plat2上的故障实例为例。

**步骤1** 以操作系统用户omm登录数据库任意一台正常的节点。

**步骤2** 使用如下命令对需要修复实例的节点进行配置操作。配置操作会清理需修复实例的空间，初始化需修复实例，配置需修复实例。

```
gs_replace -t config -h plat1,plat2
```

如果收到提示：“GAUSS\_50201: The XXX does not exist.”，则请检查对应的实例数据目录是否存在。如果不存在，请重新创建目录后再次执行上述命令。

#### 说明

- 若指定节点的表空间所在磁盘出现故障，从而导致表空间中的数据损坏，更换新磁盘后需指定--force参数强制恢复该节点的表空间数据。如果在config阶段指定--force参数，则在start阶段也必须指定--force参数。
- 替换CN实例过程中，不可在CN状态未恢复至Normal时连接该节点执行业务操作。
- 实例修复前，用户在故障实例上手动配置的GUC参数或gs\_hba.conf配置的白名单会丢失，需要重新设置。
- 实例修复过程中系统将关闭CM隔离特性开关，修复完成后系统恢复该功能至替换前状态。

**步骤3** 使用如下命令对需要修复实例的节点进行启动操作。启动操作会启动需修复实例的节点。

```
gs_replace -t start -h plat1,plat2
```

**步骤4** 使用如下命令重置实例状态。重置过程会恢复集群初始状态，以保证各节点的负载都是均衡的。

### 📖 说明

执行SWITCH操作前需确保：

- 集群状态正常。
- 所有业务已结束。
- 通过PG\_GET\_SENDERS\_CATCHUP\_TIME视图确认无主备追赶。

```
gs_om -t switch --reset
```

**步骤5** 执行`gs_om -t status`重新查看集群状态。确认集群balanced状态是否已恢复为“`Yes`”。

```
[Cluster State]
cluster_state : Normal
redistributing : No
balanced : Yes
```

### ----结束

#### 示例二 节点替换（不换IP）

以需要替换的节点为plat3为例。

**步骤1** 以root用户登录到新节点plat3，检查字符集。

```
echo $LANG
en_US.UTF-8
```

新节点上的字符集、编码方式等信息，需与其他节点保持一致。

如果不一致请使用如下方式设置字符集。

```
export LANG="en_US.UTF-8"
```

**步骤2** 以root用户登录集群的任意一台正常的节点。

**步骤3** 进入安装包解压后的script文件夹。例如，安装包存放路径为/opt/software/gaussdb。

```
cd /opt/software/gaussdb/script
```

**步骤4** 准备集群环境。

```
./gs_preinstall -U omm -G dbgrp -X ../clusterconfig.xml --alarm-type=5
```

omm为运行集群的操作系统用户，dbgrp为操作系统用户的群组名称，clusterconfig.xml为集群配置文件（配置文件中IP和主机名均与老节点保持一致），此示例中假设其存储在安装包存放路径下。

**步骤5** 切换为运行集群的操作系统用户omm。

```
su - omm
```

**步骤6** 执行如下命令完成新增节点的安装操作。

```
gs_replace -t install -h plat3
```

**步骤7** 执行如下命令完成新增节点的配置操作。

```
gs_replace -t config -h plat3
```

### 📖 说明

- 在节点替换CN实例过程中，不可在CN状态未恢复至Normal时连接该节点执行业务。
- 节点替换前，用户在故障节点上手动配置的GUC参数或gs\_hba.conf配置的白名单会丢失，需要重新设置。
- 节点替换过程中系统将关闭CM隔离特性开关，完成后系统恢复该功能至替换前状态。

**步骤8** 执行如下命令完成新增节点的启动操作。

```
gs_replace -t start -h plat3
```

**步骤9** 执行如下命令恢复集群初始状态，以保证负载均衡。

```
gs_om -t switch --reset
```

**步骤10** 执行`gs_om -t status`重新查看集群状态。确认集群balanced状态是否已恢复为“`Yes`”。

```
[Cluster State]
cluster_state : Normal
redistributing : No
balanced : Yes
```

----结束

### 示例三 节点替换（换IP）

**步骤1** 以root用户登录新节点，并检查新节点上的字符集。

```
echo $LANG
en_US.UTF-8
```

新节点上的字符集、编码方式等信息，使与其他节点保持一致。

如果不一致请使用如下方式设置字符集。

```
export LANG="en_US.UTF-8"
```

**步骤2** 以root身份登录集群的任意一台正常节点。

**步骤3** 进入安装包解压后的script文件夹。例如，安装包存放路径为/opt/software/gaussdb。

```
cd /opt/software/gaussdb/script
```

**步骤4** 准备集群环境。

```
./gs_preinstall -U omm -G dbgrp -X ../clusterconfig.xml --alarm-type=5
```

omm为运行集群的操作系统用户，dbgrp为操作系统用户的群组名称，clusterconfig.xml为集群配置文件（配置文件中IP和主机名均已替换为新节点），此示例中假设其存储在安装包存放路径下。

**步骤5** 切换为运行集群的操作系统用户omm。

```
su - omm
```

**步骤6** 执行如下命令进行节点替换操作。

```
gs_replace -t warm-standby -X clusterconfig.xml
```

如果执行过程中出现异常，请根据提示信息排除异常后重新执行该命令。

#### 说明

- 节点替换CN实例过程中，不可在CN状态未恢复至Normal时连接该节点执行业务。
- 节点替换前，用户在故障节点上手动配置的GUC参数或gs\_hba.conf配置的白名单会丢失，需要重新设置。
- 节点替换过程中系统将关闭CM隔离特性开关，完成后系统恢复该功能至替换前状态。

**步骤7** 执行如下命令恢复集群初始状态，以保证负载均衡。

```
gs_om -t switch --reset
```

**步骤8** 执行`gs_om -t status`重新查看集群状态。确认集群balanced状态是否已恢复为“`Yes`”。

```
[Cluster State]
cluster_state : Normal
```

```
redistributing : No
balanced : Yes
```

----结束

## 7.34.4 常见问题

- Q: 混合部署的情况下，被替换的旧节点上存在某分片的主DN，导致替换失败，如何处理？
- A: 被替换的节点上不能存在主DN，需使用"cm\_ctl switchover -n nodeId -D datadir"将主DN切换至同分片其他节点上，再继续节点替换操作。
- Q: 某实例故障，优先选择节点替换还是节点修复？
- A: 实例故障建议优先选择节点修复，因为节点修复只针对故障的实例，正常实例不会受影响。节点替换主要用于服务器宕机或系统故障后，使用一个新节点对老节点进行替换，流程相对复杂。

## 7.35 计划缓存

计划缓存是一种通用的SQL执行计划复用机制。通过在执行相同SQL时复用历史缓存的执行计划，无需重新生成计划，降低重复生成执行计划的开销，从而优化执行相同SQL语句的性能。本质上是一种用内存换取性能的机制，适用于对于SQL语句复用的场景。

用户可通过特定语法指定使用计划缓存能力。除此之外，可通过GUC参数 `cursor_sharing` 对非指定计划复用的SQL语句缓存计划，并进行计划复用。为了使会话间的计划可进行复用，并降低整体计划缓存的内存，支持全局共享计划缓存GPC能力。为保证计划缓存内存不会超过内存上限，同时支持计划缓存内存上限管控能力。

计划缓存功能可分为：[预编译计划缓存](#)、[自动参数化和相同语句计划复用](#)、[全局共享计划缓存](#)以及[计划缓存内存上限管控](#)。

### 7.35.1 预编译计划缓存

#### 7.35.1.1 特性概述

##### 功能定义

用户可指定相同的SQL模板语句，重复执行该语句时可以跳过解析和计划生成操作，复用上次执行时生成的计划缓存，从而提升执行速度。用户可以在SQL语句上设置绑定参数，每次执行时绑定不同参数，复用计划模板。可分为以下步骤：

- 预编译阶段：发送一个SQL语句（可绑定参数），并提供一个语句名作为键Key。
- 执行阶段：指定语句名Key，并提供参数值以执行已编译缓存的SQL语句。

其中绑定参数非必选项，需要在相同SQL模板上设置不同参数时指定即可。当业务涉及的SQL查询重复率高时，适合使用预编译计划缓存的方式执行。

可通过JDBC中PreparedStatement相关方法、gsql中PREPARE EXECUTE语法等方式使用预编译计划缓存功能。JDBC具体用法请参见[执行SQL语句](#)，PREPARE EXECUTE具体用法请参见《参考》中“SQL参考 > SQL语法 > P > PREPARE”章节。

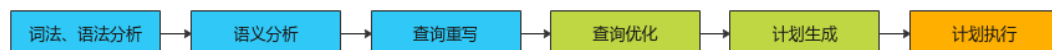
在存储过程定义中的固定SQL语句默认会在第一次执行时将SQL语句解析并生成计划缓存，可复用计划。

## 规格约束

1. 预编译的语句和对应Key为会话级别（并非全局共享），只能在本会话内使用Key映射到预期计划中，当计划无法共享时，计划也为会话内私有。
2. 在生成Generic Plan复用缓存计划时，如果修改仅影响计划的GUC参数，不会直接生效影响正在执行的复用计划。在生成Custom Plan的时，计划每次会新建，修改仅影响计划的GUC参数会及时生效。
3. 如果计划缓存所依赖的对象发生变化（如：表被执行ALTER修改或ANALYZE刷新）时，计划缓存会由于依赖对象元信息发生变化而被标记为失效，触发计划失效后，会引发计划重建以及系统开销，可能引发性能波动。
4. 如果预备语句所依赖的表结构发生变化（例如添加、删除或修改列定义），缓存计划中期望的结果结构可能与实际的表结构不匹配，后续执行该语句时，将导致非预期结果。

### 7.35.1.2 技术原理

#### 核心机制



SQL执行流程中，预编译计划缓存可复用从词法、语法分析到计划生成的完整流程，其中计划缓存复用效果受计划类型影响，复用成功后，每次执行仅需处理计划执行模块。

预编译阶段，将对词法语法解析、语义分析及查询重写结果进行缓存，并启动查询优化与计划生成操作，生成计划时选择Generic Plan或Custom Plan。两种计划类型具有不同特性：

- Generic Plan：允许不同参数值使用同一执行计划。
- Custom Plan：固化参数值至计划中，需为每个参数组合重建计划。

对于不存在绑定参数场景下，Generic Plan与Custom Plan执行计划相同，默认选择Generic Plan。

两种计划各有优势：Generic Plan可降低优化器计算开销；Custom Plan基于具体参数值进行代价估算，确保执行计划最优性。具体选择机制请参见《SQL调优指南》中“使用Plan Hint进行调优 > Custom Plan和Generic Plan选择的Hint”章节与《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节中的“plan\_cache\_mode”。

计划缓存复用范围取决于计划类型：

- Generic Plan：完整复用语法语法分析至计划生成的中间结果。
- Custom Plan：仅复用语义解析层（词法语法分析、语义分析、查询重写）。

当计划缓存依赖的数据库对象发生变更时（如DDL修改系统表元数据），缓存计划将与新元信息不匹配导致失效。此时系统通过元数据变更事件自动触发缓存失效机制，强制在下次执行时重建计划。

## 接口介绍

表 7-61 预编译计划缓存涉及的视图说明

| 视图                         | 说明                                                                                        |
|----------------------------|-------------------------------------------------------------------------------------------|
| PG_PREPARED_STATEMENTS     | 该视图显示当前会话所有可用的预备语句的信息。具体请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG_PREPARED_STATEMENTS”章节。    |
| GS_ALL_PREPARED_STATEMENTS | 该视图显示所有会话中可用的预备语句的信息。具体请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > GS_ALL_PREPARED_STATEMENTS”章节。 |

### 7.35.1.3 操作指导

#### 示例

- JDBC示例

- 调用Connection接口的prepareStatement方法创建预编译语句对象pstmt。  

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE customer_t1 SET
c_customer_name = ? WHERE c_customer_sk = 1");
```
- 调用PreparedStatement接口的setString方法设置参数。  

```
pstmt.setString(1, "new Data");
```
- 调用PreparedStatement接口的executeUpdate方法执行预编译SQL语句。  

```
int rowcount = pstmt.executeUpdate();
```
- 调用PreparedStatement接口的close方法关闭预编译语句对象pstmt。  

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

完整示例：

- 建表。  

```
gaussdb=# CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name
VARCHAR(32));
CREATE TABLE
```
- JDBC脚本。  

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;
public class TestBatch {
 public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
 // 指定数据库sourceURL（$ip、$port、database请根据业务场景进行修改）。
 String sourceURL = "jdbc:gaussdb://$ip:$port/database";
 // 用户名和密码从环境变量中获取。
 String userName = System.getenv("EXAMPLE_USERNAME_ENV");
 String password = System.getenv("EXAMPLE_PASSWORD_ENV");
 Class.forName(driver);
 return DriverManager.getConnection(sourceURL, userName, password);
 }
 public static void main(String[] args) {
 String sql = "UPDATE customer_t1 SET c_customer_name = ? WHERE c_customer_sk = 1";
 try (Connection conn = getConnection(); PreparedStatement pstmt =
 conn.prepareStatement(sql)) {
 conn.setAutoCommit(false);
 pstmt.setString(1, "new Data");
 }
 }
}
```

```
int rowcount = pstmt.executeUpdate();
conn.commit();
pstmt.close();
conn.close();
} catch (ClassNotFoundException | SQLException e) {
 throw new RuntimeException(e);
}
}
```

- JDBC示例

- a. 准备批量操作SQL语句。

使用preparedStatement对语句进行预编译。用户可根据实际业务场景替换成业务具体需要执行的语句，以下为向测试表中插入数据的语句：

```
String sql = "INSERT INTO TEST_BATCH(v1,v2) VALUES(?,?)";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
```

- b. 批量绑定参数。

通过preparedStatement绑定参数，然后执行addBatch将SQL语句添加到批量执行列表。

```
for(int i=0;i<5;i++){
 preparedStatement.setString(1,"value1_"+i);
 preparedStatement.setString(2,"value2_"+i);
 preparedStatement.addBatch();
}
```

- c. 执行批量操作。

preparedStatement执行executeBatch执行批量操作，返回一个int类型的数组results，通过打印results数组可以查看批量操作影响的数据条目数。

```
Int[] results = preparedStatement.executeBatch();
System.out.println(Arrays.toString(results));
```

- d. 释放资源和关闭数据库连接。

```
preparedStatement.close();
conn.close();
```

完整示例：

- a. 建表。

```
gaussdb=# CREATE TABLE TEST_BATCH(
V1 TEXT,
V2 TEXT);
CREATE TABLE
```

- b. java脚本。

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;

public class TestBatch {
 public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
 // 指定数据库sourceURL ($ip、$port、database请根据业务场景进行修改)。
 String sourceURL = "jdbc:gaussdb://$ip:$port/database";
 // 用户名和密码从环境变量中获取。
 String userName = System.getenv("EXAMPLE_USERNAME_ENV");
 String password = System.getenv("EXAMPLE_PASSWORD_ENV");
 Class.forName(driver);
 return DriverManager.getConnection(sourceURL, userName, password);
 }
 public static void main(String[] args) {
 String sql = "insert into TEST_BATCH(v1,v2) values(?,?)";
 try (Connection conn = getConnection(); PreparedStatement preparedStatement =
 conn.prepareStatement(sql)) {
 conn.setAutoCommit(false);
```

```
 for (int i = 0; i < 5; i++) {
 preparedStatement.setString(1,"value1_" +i);
 preparedStatement.setString(2,"value2_" +i);
 preparedStatement.addBatch();
 }
 int[] results = preparedStatement.executeBatch();
 conn.commit();
 System.out.println(Arrays.toString(results));
 preparedStatement.close();
 conn.close();
 } catch (ClassNotFoundException | SQLException e) {
 throw new RuntimeException(e);
 }
}
```

- **gsq|下PREPARE EXECUTE示例**

--创建表reason。

```
gaussdb=# CREATE TABLE reason (
 CD_DEMO_SK int NOT NULL,
 CD_GENDER varchar(10),
 CD_MARITAL_STATUS varchar(10)
);
```

--为一个INSERT语句创建一个预备语句并执行预编译语句。

```
gaussdb=# PREPARE insert_reason(int,varchar(10),varchar(10)) AS INSERT INTO reason
VALUES($1,$2,$3);
gaussdb=# EXECUTE insert_reason(52, 'AAAAAAAADD', 'reason 52');
```

--查询数据。

```
gaussdb=# SELECT * FROM reason;
cd_demo_sk | cd_gender | cd_marital_status
```

```
-----+-----+-----
 52 | AAAAAAADD | reason 52
(1 row)
```

```
gaussdb=# SELECT name,statement,parameter_types,from_sql,is_deleted,size, valid, is_reused FROM
gs_all_prepared_statements WHERE statement LIKE '%reason%';
```

```
name | statement | parameter_types | from_sql | is_deleted | size | valid | is_reused
-----+-----+-----+-----+-----+-----+-----+-----
insert_reason | PREPARE insert_reason(int,varchar(10),varchar(10)) AS INSERT INTO reason
VALUES($1,$2,$3); | {integer,"character varying","
character varying"} | t | f | 23552 | t | f
```

```
-----+-----+-----+-----+-----+-----+-----+-----
insert_reason | PREPARE insert_reason(int,varchar(10),varchar(10)) AS INSERT INTO reason
VALUES($1,$2,$3); | {integer,"character varying","
character varying"} | t | f | 23552 | t | f
(1 row)
```

--删除表reason。

```
gaussdb=# DROP TABLE reason;
```

- **存储过程示例**

```
gaussdb=# CREATE TABLE t1(a int);
CREATE TABLE
gaussdb=# CREATE OR REPLACE PROCEDURE test_proc AS
BEGIN
```

```
 INSERT INTO t1 VALUES (1);
END;
```

```
/
CREATE PROCEDURE
gaussdb=# CALL test_proc();
test_proc
```

-----

(1 row)

--查视图。

```
gaussdb=# SELECT func_signature,query FROM
```



```
pg_catalog.gs_plsql_statements_details(pg_current_sessid());
func_signature | query
-----+-----
test_proc(), fn_collation: 0, search_path: addUser: false, schemas: 2200 | INSERT INTO t1 VALUES (1)
(1 row)

gaussdb=# DROP TABLE t1;
DROP TABLE
gaussdb=# DROP PROCEDURE test_proc();
DROP PROCEDURE
```

### 7.35.1.4 常见问题

1. **问题现象：**为什么计划缓存没有复用？

**原因分析：**preparethreshold等url参数设置不当会引发计划无法缓存复用。

**解决办法：**请参见[连接参数参考](#)章节。预计在第prepareThreshold次发送SQL时才会指定语句Key，在会话上缓存计划并等待后续复用。在第prepareThreshold次前，发送的是不带语句Key的unnamed语句，这种语句的计划无法复用，每次下发新的SQL会被清理覆盖。

2. **问题现象：**计划缓存过多时怎么办？

**原因分析：**JDBC反复创建新预编译语句，并对原始语句并未进行关闭，会引发计划缓存过多的问题。url参数中preparedStatementCacheQueries和preparedStatementCacheSizeMiB可控制一个会话上的缓存query数量。但需要JDBC指定的预编译语句已经调用了close方法。

**解决办法：**可修改JDBC语句，主动关闭不再使用的预编译语句，或者通过GaussDB的计划缓存内存上限管控功能控制。

3. **问题现象：**特定SQL语句在启用Custom Plan时执行正常，但在启用Generic Plan时执行报错。

**原因分析：**Custom Plan和Generic Plan在生成阶段处理参数的方式存在差异。例如，对于包含t1 FULL JOIN t2 ON \$1的查询：

- Custom Plan：执行时会传入参数（\$1）解析为具体常量值。优化器可利用此常量（如：识别为布尔值TRUE）生成高效的执行路径（如：MERGE JOIN）。
- Generic Plan：生成计划时不会对参数（\$1）进行求值，仅将其视为未知值。因此，优化器无法基于常量条件进行特定优化（如：转换为MERGE JOIN），可能导致无法生成有效计划或选择了错误的计划。

最终表现为：该SQL在前N次（使用Custom Plan）执行成功，第N+1次及之后（可能切换为Generic Plan）执行失败。

**解决办法：**针对此类特定SQL语句，可通过以下方式强制启用Custom Plan：

- 在应用连接串中设置参数，调整连接参数。
- 使用执行计划优化（Hint），在SQL语句中嵌入特定数据库的Hint。

## 7.35.2 自动参数化和相同语句计划复用

### 7.35.2.1 特性概述

#### 功能定义

- 相同语句计划复用：是一项通过缓存执行计划优化查询性能的技术。其核心原理是缓存SQL语句及已生成的执行计划，当后续遇到相同SQL文本的SQL语句（包括所有字面值、大小写、空格等都需相同）时，直接复用缓存计划，从而跳过重复

的解析、优化等耗时阶段。该功能支持GSQL发送的简单查询和PBE（包含batch）场景，SQL文本中可以包含用户指定的绑参。

- 自动参数化：参数化过程会将SQL语句中的字面量替换为绑定参数变量。当参数化后的SQL语句相同，可复用同一个计划模板。支持简单查询和PBE（包含batch）中无绑定参数的场景。原始SQL语句不能存在用户指定绑定参数，如果已包含用户指定绑参，仅能支持相同语句计划复用。

相同语句计划复用技术和自动化参数技术可通过GUC参数cursor\_sharing控制，GUC参数cursor\_sharing的行为介绍，请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。

- 当GUC参数cursor\_sharing设置为force时，同时启用自动参数化以及相同语句计划复用功能。对于满足自动参数化条件的语句，则优先通过自动参数化缓存；满足不了则通过相同语句计划复用缓存。例如在MYSQL或M兼容模式下，不支持参数化，会自动按照cursor\_sharing = exact的行为实现。
- 当GUC参数cursor\_sharing设置为exact时，仅启用相同语句计划复用功能。
- 当GUC参数cursor\_sharing设置为off时，关闭自动参数化以及相同语句计划复用功能。

在计划模板无法复用的情况下，无法通过计划复用带来性能提升。且如果存入大量不同的SQL模板，会引起计划缓存内存占用过多，触发计划内存上限淘汰，导致性能下降，极端场景下可能会引起掉零。

## 规格约束

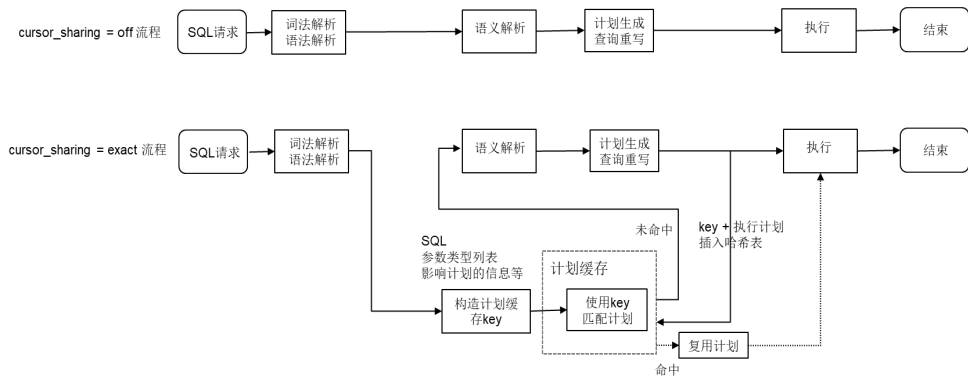
- 相同语句计划复用：
  - 仅支持简单查询语句以及PBE语句。
  - 仅支持DML语句。
  - 不支持存储过程内语句。
  - M-Compatibility兼容模式下，不支持SHOW INDEX语句以及不支持SELECT INTO outfile语句。
- 自动参数化：
  - 仅支持简单查询语句以及无绑定参数的PBE（包含batch场景）语句。
  - 仅支持DML语句。
  - 不支持存储过程内语句。
  - 不支持语句本身含有绑定参数的语句。
  - 不支持lightproxy场景下含GSI表的语句。
  - 不支持含有DBLINK表的语句。
  - 不支持MYSQL兼容模式数据库。
  - M-Compatibility兼容模式下，设置GUC参数m\_format\_behavior\_compat\_options包含select\_column\_name选项后，不支持自动参数化。
  - M-Compatibility兼容模式下，存在用户变量的语句不支持自动参数化。
  - M-Compatibility兼容模式下，不支持SHOW INDEX语句自动参数化，不支持SELECT INTO outfile语句自动参数化。
  - 部分常量不支持参数化，包含类型后面的精度常量，如：int(n)、ORDER BY/GROUP BY后的列常量、Hint内常量、\_charset后的常量以及分区表分区中的常量等。

7.35.2.2 技术原理

核心机制

- 相同语句计划复用

图 7-72 相同语句计划复用流程原理图

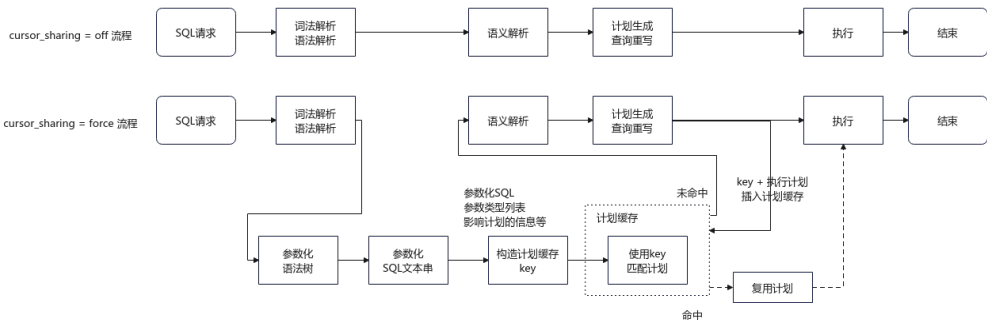


用户SQL语句经词法解析与语法解析后，对于相同的SQL语句计划实现复用。当用户第一次执行某SQL语句时，数据库对SQL语句进行语义解析及计划生成，同时将语义解析结果及计划根据SQL等关键信息作为键（Key）构建计划缓存哈希表，进行缓存。当后续遇到相同SQL文本的SQL语句（包括所有字面值、大小写、空格等都需相同）时，根据键（Key）从计划缓存哈希表中匹配计划缓存并直接复用相同的缓存计划，从而跳过重复的解析、优化等耗时阶段。

相较于非相同语句计划复用流程，相同语句计划复用增加了以下操作逻辑：

- 构造计划缓存键（Key）
  - 使用键（Key）匹配计划
  - 复用计划
  - 存储键（Key）与执行计划相关信息
- 自动参数化

图 7-73 自动参数化流程原理图



- a. 用户SQL语句经词法解析与语法解析生成原始语法树后，自动参数化模块将遍历语法树节点，对常量节点进行参数化。具体操作包括将常量节点转换为参数节点以及记录参数类型等信息。
- b. 根据参数化的节点信息生成SQL模板。

- c. 当某模板SQL首次执行时，数据库引擎基于参数化的语法树进行执行以下操作：
  - 语义分析
  - 计划生成
  - 缓存计划
- d. 当再次执行相同SQL模板语句时，启用缓存复用机制（即后续相同模板SQL复用缓存计划）。

相较于非参数化流程，自动参数化增加了以下操作逻辑：

- 参数化语法树
- 参数化SQL文本串
- 构造计划缓存key
- 使用键（Key）匹配计划
- 复用计划
- 存储键（Key）与执行计划相关信息
- 绑定参数化常量信息执行

## 接口介绍

| GUC参数名         | GUC参数说明                                                                  |
|----------------|--------------------------------------------------------------------------|
| cursor_sharing | 控制自动参数化和相同语句计划复用功能开启。具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。 |

| Hint名                | Hint说明                                                                                                           |
|----------------------|------------------------------------------------------------------------------------------------------------------|
| cursor_sharing_exact | 指定单个SQL语句使用相同语句计划复用，其作用等同于cursor_sharing=exact。具体请参见《SQL调优指南》中“使用Plan Hint进行调优 > 指定cursor_sharing为exact的Hint”章节。 |

| 函数或视图名                     | 函数或视图说明                                                                                                                                                     |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DBE_PERF.GS_PLANCACHE_STAT | <p>显示当前进程内全部session的可复用计划的语句信息，查看该视图需要初始用户或MONITORADMIN权限。</p> <p>具体请参见《参考》中“Schema &gt; DBE_PERF Schema &gt; Query &gt; DBE_PERF.GS_PLANCACHE_STAT”章节。</p> |

| 函数或视图名                                         | 函数或视图说明                                                                                                                                                                    |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GS_ALL_PREPARED_STATEMENTS                     | 视图新增is_reused字段表示该缓存是否被重用。<br>具体请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > GS_ALL_PREPARED_STATEMENTS”章节。                                                                         |
| DBE_PERF.GLOBAL_PLANCACHE_STATUS               | 视图新增parameterized_info字段，表示该语句通过cursor_sharing的exact或force的流程执行。<br>具体请参见《参考》中“Schema > DBE_PERF Schema > Global PlanCache > DBE_PERF.GLOBAL_PLANCACHE_STATUS”章节。          |
| DBE_PERF.GS_DIST_PLANCACHE_STAT                | 显示全部进程（包含备机）的可复用计划的信息，查看该视图需要初始用户或MONITORADMIN权限。具体请参见《参考》中“Schema > DBE_PERF Schema > Query > DBE_PERF.GS_DIST_PLANCACHE_STAT”章节。                                         |
| pg_catalog.gs_dist_plancache_stat()            | 查看包含备机的全部进程的可复用计划的语句信息的系统函数，函数返回信息和DBE_PERF.GS_DIST_PLANCACHE_STAT一致。查看该函数需要初始用户或MONITORADMIN权限。具体请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。                                     |
| pg_catalog.gs_plancache_stat()                 | 查看当前进程内全部session的可复用计划的语句信息，函数返回信息和DBE_PERF.GS_PLANCACHE_STAT一致。查看该函数需要初始用户或MONITORADMIN权限。<br>具体请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。                                      |
| DBE_PERF.global_plancache_status(is_with_plan) | 显示在所有节点上的全局计划缓存的状态信息，即plan_info列。入参is_with_plan为bool类型，默认值为false。当入参为true时，显示plan_info的信息。函数返回信息和DBE_PERF.GLOBAL_PLANCACHE_STATUS一致。具体请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。 |

当设置GUC参数cursor\_sharing = force时，启用自动参数化的场景下，通过EXPLAIN查看计划是否走自动参数化：

| EXPLAIN模式     | 计划展示说明                         |
|---------------|--------------------------------|
| normal/pretty | 被参数化的计划会显示[Parameterized]的tag。 |

### 7.35.2.3 操作指导

- cursor\_sharing = exact，仅启用相同语句计划复用功能。

## a. 建库、建表及设置参数。

```
gaussdb=# CREATE DATABASE ora_db dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c ora_db
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora_db" as user "yumingyue".
ora_db=# CREATE TABLE cursor_sharing_exact_tbl(col1 int, col2 int) DISTRIBUTE BY hash(col1);
CREATE TABLE
ora_db=# INSERT INTO cursor_sharing_exact_tbl VALUES(1, 1);
INSERT 0 1
ora_db=# SET cursor_sharing = exact;
SET
```

## b. 执行查询。

```
ora_db=# SELECT * FROM cursor_sharing_exact_tbl WHERE col1 = 1;
 col1 | col2
-----+-----
 1 | 1
(1 row)
```

## c. 查询视图dbe\_perf.gs\_plancache\_stat展示了该语句的缓存信息。

```
ora_db=# SELECT parameterized_sql, parameter_types, is_parameterized FROM
dbe_perf.gs_plancache_stat WHERE parameterized_sql LIKE '%from cursor_sharing_exact_tbl%';

parameterized_sql | parameter_types | is_parameterized
-----+-----
select * from cursor_sharing_exact_tbl where col1 = 1; | {}
select parameterized_sql, parameter_types, is_parameterized from dbe_perf.gs_plancache_stat where parameterized_sql like '%from cursor_sharing_exact_tbl%'; | {}
(2 rows)
```

## d. 若环境开启了GPC，查询GPC视图

DBE\_PERF.GLOBAL\_PLANCACHE\_STATUS，字段parameterized\_info被标记为exact表示该语句通过相同语句计划复用特性执行。若环境未开启GPC，请开启enable\_global\_plancache后重启数据库，并重新执行a~c。

```
ora_db=# SHOW enable_global_plancache;
enable_global_plancache

on
(1 row)

ora_db=# SELECT nodename, query, parameterized_info FROM
DBE_PERF.GLOBAL_PLANCACHE_STATUS WHERE query LIKE '%from cursor_sharing_exact_tbl%';
nodename | query | parameterized_info
-----+-----
cn1 | select parameterized_sql, parameter_types, is_parameterized from dbe_perf.gs_plancache_stat where parameterized_sql like '%from cursor_sharing_exact_tbl%'; | exact
cn1 | select nodename, query, parameterized_info from DBE_PERF.GLOBAL_PLANCACHE_STATUS where query like '%from cursor_sharing_exact_tbl%'; | exact
cn1 | select * from cursor_sharing_exact_tbl where col1 = 1; | exact
datanode1 | select * from cursor_sharing_exact_tbl where col1 = 1; | exact
(4 rows)
```

## e. 删库恢复环境。

```
ora_db=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
```

```
You are now connected to database "postgres" as user "yumingyue".
gaussdb=# DROP DATABASE ora_db;
DROP DATABASE
```

- `cursor_sharing = force`，同时启用自动参数化以及相同语句计划复用功能。

a. 建库、建表及设置参数。

```
gaussdb=# CREATE DATABASE ora_db dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c ora_db
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora_db" as user "yumingyue".
ora_db=# CREATE TABLE cursor_sharing_force_tbl(col1 int, col2 int) DISTRIBUTE BY hash(col1);
CREATE TABLE
ora_db=# INSERT INTO cursor_sharing_force_tbl VALUES(1, 1);
INSERT 0 1
ora_db=# SET cursor_sharing = force;
SET
```

b. EXPLAIN查看计划，被参数化的计划会显示[Parameterized]的tag。

```
ora_db=# EXPLAIN (costs off) SELECT * FROM cursor_sharing_force_tbl WHERE col1 = 1;
QUERY PLAN

[Parameterized]
Data Node Scan
 Node/s: datanode1
(3 rows)
```

c. 执行语句。

```
ora_db=# SELECT * FROM cursor_sharing_force_tbl WHERE col1 = 1;
 col1 | col2
-----+-----
 1 | 1
(1 row)
```

d. 查询视图`dbe_perf.gs_plancache_stat`展示了该语句的缓存信息，如`parameterized_sql`字段展示了对该SQL语句参数化后的结果。

```
ora_db=# SELECT parameterized_sql, parameter_types, is_parameterized FROM
dbe_perf.gs_plancache_stat WHERE parameterized_sql LIKE '%from cursor_sharing_force_tbl%';
 parameterized_sql | parameter_types | is_parameterized
-----+-----+-----
select * from cursor_sharing_force_tbl where col1 = ?; | {integer} | t
(1 row)
```

e. 查询`dbe_perf.statement`视图，`query`字段展示参数化后的语句信息，`$`表示被参数化的常量。（需要保证当前`unique sql`记录条数少于最大可记录条数`instr_unique_sql_count`）。

```
ora_db=# SELECT query FROM dbe_perf.statement WHERE query LIKE '%from
cursor_sharing_force_tbl%';
 query

select * from cursor_sharing_force_tbl where col1 = $1;
(1 row)
```

f. 删库恢复环境。

```
ora_db=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "yumingyue".
gaussdb=# DROP DATABASE ora_db;
DROP DATABASE
```

## 7.35.2.4 常见问题

1. Q：自动参数化是否存在计划跳变问题？

A：在自动参数化下，部分语句计划可能会发生变化。由于自动参数化将语句中的常量替换为参数，再进行绑参执行。在计划生成的过程中，部分场景下对常量及参数的处理有区别，因此导致设置`cursor_sharing = force`后，出现部分计划跳变的问题。

如下示例中，SELECT \* FROM cursor\_sharing\_tbl WHERE col1 = 1;在 cursor\_sharing = off场景使用索引扫描，在cursor\_sharing = force场景下使用全表扫描。

```
gaussdb=# CREATE DATABASE ora_db dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c ora_db
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora_db" as user "yumingyue".
ora_db=# CREATE TABLE cursor_sharing_tbl(col1 int, col2 int) DISTRIBUTE BY hash(col1);
CREATE TABLE
ora_db=# CREATE INDEX cursor_sharing_tbl_idx ON cursor_sharing_tbl(col1) WHERE col1 != 0;
CREATE INDEX
ora_db=# SET plan_cache_mode = 'force_generic_plan';
SET
ora_db=# SET max_datanode_for_plan = 1;
SET
ora_db=# SET cursor_sharing = off;
SET
ora_db=# EXPLAIN verbose SELECT * FROM cursor_sharing_tbl WHERE col1 = 1;
 QUERY PLAN

Data Node Scan (cost=0.00..0.00 rows=0 width=0)
 Output: cursor_sharing_tbl.col1, cursor_sharing_tbl.col2
 Node/s: datanode1
 Remote query: SELECT col1, col2 FROM public.cursor_sharing_tbl WHERE col1 = 1

Remote SQL: SELECT col1, col2 FROM public.cursor_sharing_tbl WHERE col1 = 1
Datanode Name: datanode1
 Bitmap Heap Scan on public.cursor_sharing_tbl (cost=1.36..9.58 rows=11 width=8)
 Output: col1, col2
 Recheck Cond: (cursor_sharing_tbl.col1 = 1), (Expression Flatten Optimized)
 -> Bitmap Index Scan on cursor_sharing_tbl_idx (cost=0.00..1.36 rows=11 width=0)
 Index Cond: (cursor_sharing_tbl.col1 = 1)

(13 rows)

ora_db=# SET cursor_sharing = force;
SET
ora_db=# EXPLAIN verbose SELECT * FROM cursor_sharing_tbl WHERE col1 = 1;
 QUERY PLAN

[Parameterized]
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
 Output: cursor_sharing_tbl.col1, cursor_sharing_tbl.col2
 Node/s: datanode1
 Remote query: SELECT col1, col2 FROM public.cursor_sharing_tbl WHERE col1 = 1

Remote SQL: SELECT col1, col2 FROM public.cursor_sharing_tbl WHERE col1 = 1
Datanode Name: datanode1
[Parameterized]
Seq Scan on public.cursor_sharing_tbl (cost=0.00..36.86 rows=11 width=8)
 Output: col1, col2
 Filter: (cursor_sharing_tbl.col1 = $1), (Expression Flatten Optimized)

(13 rows)

ora_db=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "yumingyue".
gaussdb=# DROP DATABASE ora_db;
DROP DATABASE
```

该场景由于自动参数化将SELECT \* FROM cursor\_sharing\_tbl WHERE col1 = 1;中常量1作为参数生成计划，在设置plan\_cache\_mode = 'force\_generic\_plan';强制走gplan的场景，会导致cursor\_sharing\_tbl\_idx部分索引的索引条件为参数，索引失效，因此使用全表扫描。



计划跳变场景可通过设置`plan_cache_mode = 'force_custom_plan'`强制走cplan解决，或关闭自动参数化解决。

2. Q：自动参数化及相同语句计划复用是否存在计划缓存过多问题？

A：自动参数化及相同语句计划复用机制通过缓存已执行语句的执行计划，避免在相同语句场景下重复生成执行计划，从而提高查询执行效率。然而，此机制可能导致执行计划缓存占用过多内存。当前系统已启用计划缓存上限管控功能。若观察到session级计划缓存占用异常，可调整`max_plancache_memory_ratio`参数进行优化。

## 7.35.3 全局共享计划缓存

### 7.35.3.1 特性概述

#### 功能定义

全局共享计划缓存（Global Plan Cache, GPC）功能可共享不同会话间的执行计划。GPC在预编译计划缓存、自动参数化和相同语句计划复用的基础上生效，将缓存在会话内的计划存入GPC，使其在全局可见，其他会话可根据实际的SQL语句在全局计划缓存中获取所需的计划，并复用共享计划缓存进行执行操作。

GPC可通过GUC参数`enable_global_plancache`进行控制。不开启GPC（即GUC参数`enable_global_plancache`设置为off）时，每个会话仅对自己执行过的计划进行缓存，即存在相同的业务会话且均执行相同的查询，每个会话上都需要单独生成一次私有的计划。开启GPC（即GUC参数`enable_global_plancache`设置为on）后，由于计划在全局范围可获取，新建的连接可以直接在共享计划缓存中获取计划（并非单独生成计划）。因此GPC可降低会话间重复计划缓存占用的内存，且能使新建会话直接使用缓存计划，降低新建会话上的计划生成开销，从而提升查询效率。

#### 说明

- 每个CN或DN上设置的GUC参数`enable_global_plancache`值须保持一致，否则会出现异常报错。

#### 用户场景

数据库运行时，若执行计划仅缓存在当前会话中，会浪费大量内存，通过会话间的全局共享计划缓存GPC，可以节省大量内存。

#### 规格约束

1. 对指定的预编译计划缓存、自动参数化和相同语句计划复用的计划缓存可支持全局共享。
2. 计划缓存在生成Generic Plan时，才可存入共享。
3. 对临时表、全局临时表的计划即使生成Generic Plan也不会存入共享。
4. STREAM计划不支持存入共享。
5. 客户端支持直连DN时，不支持执行PBE或PREPARE EXECUTE语句。

7.35.3.2 技术原理

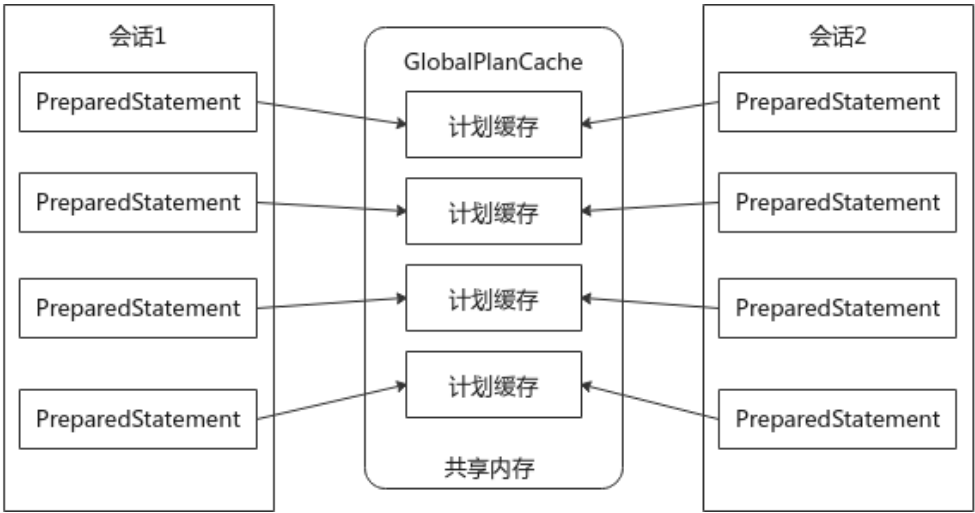
核心机制

以预编译PreparedStatement为例，计划缓存在当前会话上时，内存位置如图7-74所示，开启GPC全局共享计划后，内存位置如图7-75所示。

图 7-74 未开启 GPC 计划缓存内存位置图



图 7-75 开启 GPC 计划缓存后内存位置图



用户指定预编译计划缓存，在会话上基于指定键Key（SQL语句和会话级GUC参数）生成预编译语句，通过Key在共享计划缓存中进行匹配并直接引用已匹配的缓存计划。自动参数化、相同语句计划复用、存储过程中计划缓存共享逻辑与之类似。

- 由于计划缓存在会话上，CN与DN间计划缓存复用和会话存在耦合关系。CN连接一个DN执行PBE后，连接不会释放，必须保留长连接，确保在CN上转发语句Key

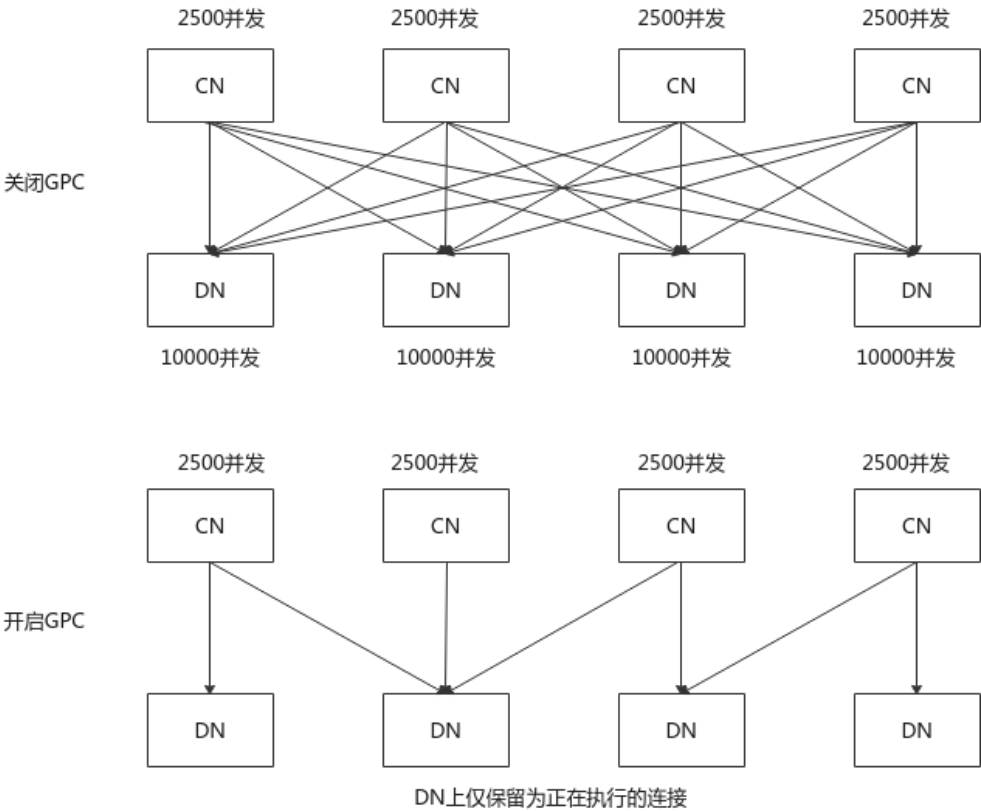
给DN会话时获取到预期的计划缓存。仅当CN上会话退出之后，才能释放和该DN的连接。

- 在高并发场景下，DN将成为集群的连接瓶颈。  
如图7-76所示，CN上长连接场景下，假设每个CN为2500并发，在4CN+4DN的集群上，由于DN上保留了所有CN的PBE连接，每个DN上有10000并发。每个会话都会占用内存，所以DN的主要瓶颈在内存，系统整体并发能力受限于单DN的并发能力。

当开启GPC时，CN每次执行预编译语句均直接转发原始PBE命令给DN，与预编译计划缓存场景不同，不通过语句名作为Key在会话中获取计划缓存，而是通过每次转发给DN的PBE中的SQL文本作为Key，从全局共享计划中获取计划缓存。

通过GPC功能，无需保留会话进行缓存计划，使得CN与DN间会话可以和预编译语句解耦，不必保持长连接。因此DN上的连接数为正在执行的连接数，执行结束的可直接释放，且DN上不存在预编译语句缓存，每次将从原始SQL直接解析。

图 7-76 开启/关闭 GPC 场景下，CN 与 DN 连接数对比



接口介绍

| 名称            |                         | 说明                                                               |
|---------------|-------------------------|------------------------------------------------------------------|
| GU<br>C参<br>数 | enable_global_plancache | 控制全局计划缓存功能开启，具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。 |

| 名称     |                                                | 说明                                                                                                                               |
|--------|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
|        | gpc_clean_timeout                              | 控制不使用的共享计划的保留时间，具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。                                                              |
|        | disable_gpc_store                              | 用于阻止后续计划存入全局计划缓存，具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。                                                             |
|        | enable_gpc_grayrelease_mode                    | 在不重启集群的情况下开启GPC，具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。                                                              |
| Hint名  | no_gpc                                         | 强制单个查询语句不在全局共享计划缓存，具体请参见《SQL调优指南》中“使用Plan Hint进行调优”章节。                                                                           |
| 函数或视图名 | DBE_PERF.GLOBAL_PLANCACHE_STATUS               | 该视图显示GPC全局计划缓存状态信息，具体请参见《参考》中“Schema > DBE_PERF Schema > Global Plancache”章节。                                                    |
|        | DBE_PERF.LOCAL_PLANCACHE_STATUS                | 该视图显示当前节点的GPC全局计划缓存状态信息，具体请参见《参考》中“Schema > DBE_PERF Schema > Global Plancache”章节。                                               |
|        | DBE_PERF.LOCAL_PLANCACHE_CEALN                 | 清理当前节点上无人使用的全局计划缓存，具体请参见《参考》中“Schema > DBE_PERF Schema > Global Plancache”章节。                                                    |
|        | DBE_PERF.GLOBAL_PLANCACHE_CLEAN                | 该视图清理当前节点上无人使用的全局计划缓存，具体请参见《参考》中“Schema > DBE_PERF Schema > Global Plancache”章节。                                                 |
|        | pg_catalog.plancache_status(is_with_plan)      | 显示在当前节点上的全局计划缓存的状态信息，即plan_info列。入参is_with_plan为bool类型，默认值为false。当入参为true时，显示plan_info的信息。具体请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。 |
|        | dbe_perf.global_plancache_status(is_with_plan) | 显示在所有节点上的全局计划缓存的状态信息，即plan_info列。入参is_with_plan为bool类型，默认值为false。当入参为true时，显示plan_info的信息。具体请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。 |
|        | DBE_PERF.global_plancache_clean()              | 清理所有节点上无人使用的全局计划缓存，具体请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > 其它函数”章节。                                                                 |
|        | pg_catalog.plancache_clean()                   | 清理当前节点上无人使用的全局计划缓存，具体请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > 其它函数”章节。                                                                 |

| 名称                                                              | 说明                                                                                                                   |
|-----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| pg_catalog.global_plancache_set_invalid(hashkey, unique_sql_id) | 支持通过bigint类型的入参hashkey和unique_sql_id手动失效本节点的GPC缓存计划，具体请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > 其它函数”章节。                    |
| pg_catalog.global_plancache_show_plan(hashkey, unique_sql_id)   | 通过bigint类型的入参hashkey和unique_sql_id，用于查看对应hashkey和unique_sql_id的GPC缓存计划，具体请参见《参考》中“SQL参考 > 函数和操作符 > 系统管理函数 > 其它函数”章节。 |

### 7.35.3.3 操作指导

#### 前提条件

- 开启PBE查询和存储过程中语句的执行计划进行缓存共享。将enable\_global\_plancache参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。
- 以上操作需要重启集群生效。
- 允许后续执行计划存入全局计划缓存。  
SET disable\_gpc\_store = off;

#### 说明

本示例使用1CN+2DN分布式集群，其他配置结果可能略有不同。

#### 操作步骤

- JDBC用例
  - 准备数据。

```
gaussdb=> CREATE TABLE test_gpc_jdbc(col_a int, col_b int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'col_a' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=> INSERT INTO test_gpc_jdbc VALUES (generate_series(1, 1000), generate_series(1, 1000));
INSERT 0 1000
```
  - 执行JDBC。

// 以下用例以gsjdb4.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。

// gsjdb4.jar: 主类名为“org.postgresql.Driver”，数据库连接的url前缀为“jdbc:postgresql”。

```
public static void main(String[] args) throws IOException {
 // 链接字符串中prepareThreshold = 1，表示从第一个语句开始就是named PBE。
 String urls = "jdbc:postgresql://localhost:48100/postgres?prepareThreshold=1";
 // 在环境变量USER、PASSWORD配置用户名和密码
 String username = System.getenv("USER");
 String passwd = System.getenv("PASSWORD");
 Properties urlProps = new Properties();
 urlProps.setProperty("user", username);
 urlProps.setProperty("password", passwd);

 try {
 Connection con = DriverManager.getConnection(urls, urlProps);
 con.setAutoCommit(true);
 }
}
```

```
Statement stmt = con.createStatement();
// 为方便快速查看计划存入缓存的情况，设置plan_cache_mode=force_generic_plan。
stmt.execute("set plan_cache_mode = 'force_generic_plan'");
// 调用同一个Connection接口创建prepareStatement并执行语句。
String selectsql = "select * from test_gpc_jdbc where col_a = ? and col_b = ?";
PreparedStatement pstmt = con.prepareStatement(selectsql);
pstmt.setInt(1, 100);
pstmt.setInt(2, 200);
pstmt.execute();

stmt.close();
pstmt.close();
con.close();
} catch (SQLException e) {
 e.printStackTrace();
 System.exit(1);
}
}
```

- c. 登录默认数据库postgres查看GPC视图，可查询到刚执行的语句，最后通过系统视图清理。

```
gaussdb=> SELECT query, refcount, valid FROM db_perf.global_plancache_status WHERE query
LIKE '%test_gpc_jdbc%';
```

| query                                                         | refcount | valid |
|---------------------------------------------------------------|----------|-------|
| select * from test_gpc_jdbc where col_a = \$1 and col_b = \$2 | 0        | t     |
| select * from test_gpc_jdbc where col_a = \$1 and col_b = \$2 | 0        | t     |

(2 rows)

```
gaussdb=> SELECT * FROM db_perf.global_plancache_clean();
global_plancache_clean
```

| t |
|---|
|   |

(1 row)

```
gaussdb=> SELECT query, refcount, valid FROM db_perf.global_plancache_status WHERE query
LIKE '%test_gpc_jdbc%';
```

| query | refcount | valid |
|-------|----------|-------|
|-------|----------|-------|

(0 rows)

- d. 删除表。

```
gaussdb=> DROP TABLE test_gpc_jdbc;
DROP TABLE
```

- 函数用例

- a. 准备数据和函数。

```
gaussdb=> CREATE TABLE test_gpc_proc(col_a int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'col_a' as the distribution column by default.
```

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

```
CREATE TABLE
```

```
gaussdb=> INSERT INTO test_gpc_proc VALUES(generate_series(1, 1000));
```

```
INSERT 0 1000
```

```
gaussdb=> CREATE OR REPLACE FUNCTION count_beyond
```

```
gaussdb-> (
```

```
gaussdb(> threshold int
```

```
gaussdb(>)
```

```
gaussdb-> RETURNS int
```

```
gaussdb-> AS $$
```

```
gaussdb$> DECLARE
```

```
gaussdb$> count_number int;
```

```
gaussdb$> BEGIN
```

```
gaussdb$> select count(*) from test_gpc_proc where col_a > threshold into count_number;
```

```
gaussdb$> return count_number;
```

```
gaussdb$> END;
```

```
gaussdb$> $$ LANGUAGE 'plpgsql';
```

```
CREATE FUNCTION
```

```
gaussdb=> SET plan_cache_mode='force_generic_plan';
```

```
SET
gaussdb=> SET enable_stream_operator=off;
SET
```

b. 调用函数并查看GPC视图。

```
gaussdb=> SELECT count_beyond(200);
count_beyond

 800
(1 row)
```

```
gaussdb=> SELECT query, refcount, valid FROM db_perf.global_plancache_status WHERE query
LIKE '%test_gpc_proc%';
 query | refcount | valid
-----+-----+-----
select count(*) from test_gpc_proc where col_a > threshold | 1 | t
(1 row)
```

c. 删除函数及数据。

```
gaussdb=> DROP FUNCTION count_beyond;
DROP FUNCTION
gaussdb=> DROP TABLE test_gpc_proc;
DROP TABLE
```

### 7.35.3.4 常见问题

1. 为什么计划无法存入共享缓存？
  - 如果计划没有选择Generic Plan，或SQL中包含临时表或全局临时表时，会导致计划无法共享。具体约束请参见[规格约束](#)。
  - 如果共享计划缓存达到内存上限，且无法淘汰，也会无法存入共享。具体请参见[计划缓存内存上限管控](#)。
2. GUC参数enable\_global\_plancache一定要重启集群生效吗？

不需要，可通过enable\_gpc\_grayrelease\_mode参数控制滚动重启各个节点以实现开启GPC。具体请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。

## 7.35.4 计划缓存内存上限管控

### 7.35.4.1 特性概述

#### 功能定义

在数据库系统中，对于PBE场景以及启用cursor\_sharing功能时，会话级和全局共享级计划缓存机制可能面临内存溢出风险。具体而言，若客户端频繁提交差异化SQL语句，会导致内核持续生成新的计划缓存条目。在缺乏有效清理机制的情况下，这些缓存条目会在内存中无限制累积，最终可能引发内存资源耗尽、系统不可用的严重故障。

为解决此隐患，GaussDB创新性地实现了双维度内存管控体系：

1. 会话级管控：通过动态阈值限制单个会话的私有计划缓存内存占比。
2. 全局共享级管控：采用LRU-K算法对全局共享缓存实施准入控制，优先保留高频执行的优质计划。

该机制通过双重约束（会话内存上限+全局淘汰策略），确保计划缓存始终运行在预设的安全内存范围内。当缓存总量达到预警阈值时，系统自动触发异步清理线程，按策略淘汰低效缓存条目，在保障查询性能的同时，有效规避内存溢出风险。



用户场景

在数据库运行过程中，若因计划缓存条目持续累积且缺乏有效清理机制，导致动态内存资源耗尽，将引发业务请求无法正常执行、数据库服务中断等严重故障。通过本特性的内存管控机制，系统能够对计划缓存实施精准的内存配额管理，确保关键业务始终获得必要的内存资源支持，从而有效维持核心服务可用性，显著增强系统健壮性。

规格约束

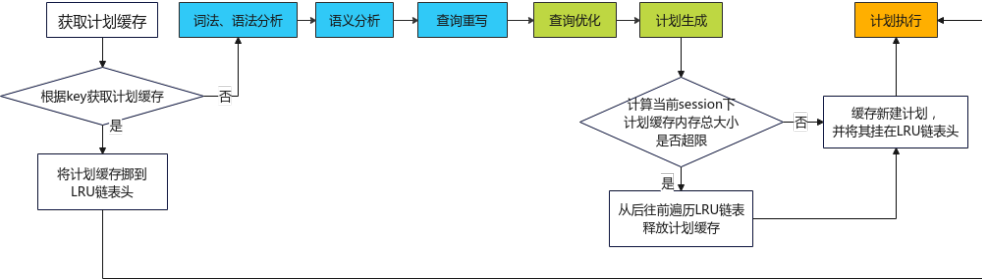
- 1. 在计划缓存淘汰触发计划重建过程中，系统将维持当前GUC参数配置状态，不会回滚或重置历史参数设置。若重建操作执行前发生GUC参数变更，新生成的执行计划将基于最新参数环境构建，此机制可能导致淘汰前后的计划在执行策略或资源分配层面产生差异。该行为属于系统设计的预期特性，而非功能异常。
- 2. 对于会话级计划缓存管控，仅对预编译（PREPARED）语句声明的查询框架生成的执行计划以及GUC参数cursor\_sharing参数化功能触发的缓存计划占用的内存进行管控，其他内存不做管控（临时表空间内存分配、会话级排序区/哈希区、用户连接上下文内存以及其他非计划缓存相关内存对象等）。
- 3. 对于全局共享计划缓存（Global Plan Cache, GPC）实施全维度内存管控，支持管理所有纳入GPC的缓存计划内存分配以及支持多维度计划类型内存的管控，包括PREPARED语句创建的预编译执行计划以及通过GUC参数cursor\_sharing参数化生成的通用查询计划。

7.35.4.2 技术原理

核心机制

- 会话级计划缓存控制  
GaussDB接收到SQL语句后，首次执行该SQL时，对其进行词语法解析、查询重写等操作后生成执行计划，并在会话中对执行计划进行缓存并计算其占用的内存大小，并将生成执行计划必要的上下文信息进行保存，用于该执行计划触发淘汰机制后的重建操作。具体流程如图7-77所示。

图 7-77 会话级计划缓存控制流程



会话级的计划缓存管控通过一个LRU链表，根据执行计划的执行频次进行优先展示的逻辑实现。当对执行计划进行缓存完成后，会将计划指针记录在LRU链表头部，每次使用过的执行计划缓存会移动到链表头部，即越靠近LRU链表头部的执行计划，即为最近使用频次越高的的执行计划。

在新建计划时，会检查当前会话内计划缓存总大小是否超过设定阈值，如果超过，则从LRU链表尾部开始遍历计划并释放，直到会话级计划缓存总大小小于设定阈值。

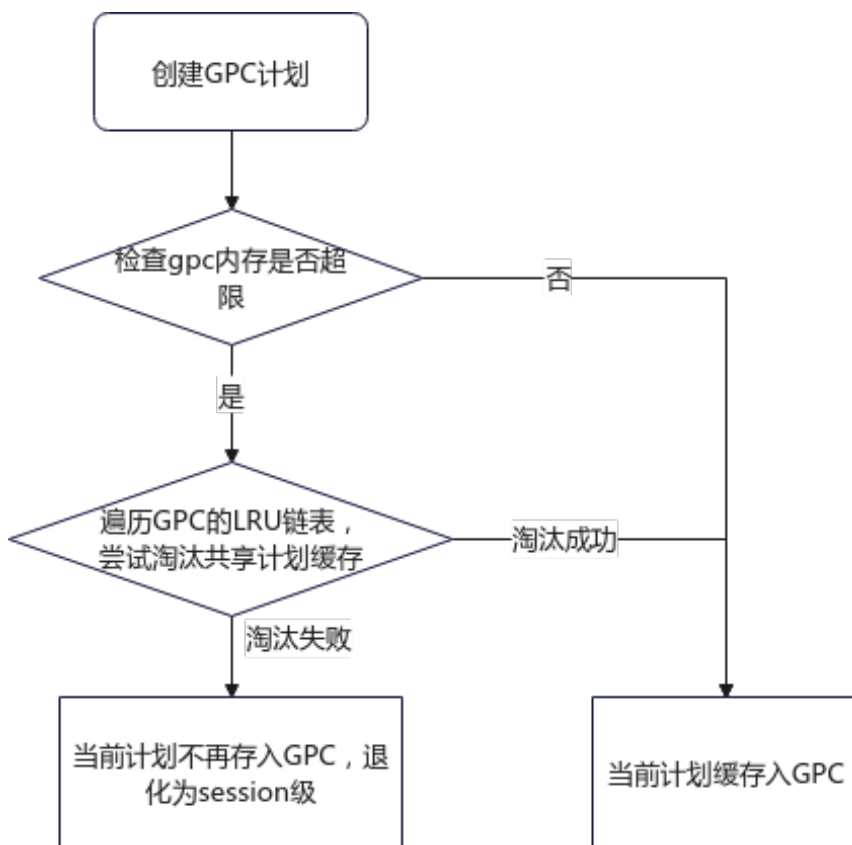


下次重复执行时，会尝试获取计划缓存。如果计划缓存已存在，则将计划指针移动到LRU链表头，继续正常执行；否则需要重新创建计划，新建计划流程和首次生成计划一致。

- 全局共享级计划缓存GPC控制

全局共享级计划缓存GPC管控，是在会话级计划缓存管控基础上实现的。由于全局共享级计划同一时间下可能被多个会话同时使用，无法直接淘汰并清理出可用的内存，即GPC仅在没有会话使用时才能触发淘汰。具体流程如图7-78所示。

图 7-78 全局共享级计划缓存控制流程



为保证全局共享级计划缓存实时生效，如果全局共享级计划缓存内存不足且无法淘汰时，则计划无法存入全局共享，进行会话级计划缓存管控控制。会话级计划缓存被串行使用，在本会话内可保证需要清理时直接删除。

检查当前计划存入GPC后是否会导致GPC总内存大小超限，如果未超限，则正常存入GPC；否则，开始遍历LRU链表并尝试淘汰。如果淘汰成功，该计划存入GPC；如果淘汰失败，计划仅存入当前会话，无法存入GPC。

#### 说明

计划缓存淘汰后重建会导致词语法解析等流程的再执行，会带来额外的性能开销。

## 接口介绍

- GUC参数说明

表 7-62 计划缓存管控功能 GUC 参数说明

| 管控等级  | 参数名                           | 参数说明                                                                                         |
|-------|-------------------------------|----------------------------------------------------------------------------------------------|
| 会话级   | max_plancache_memory_ratio    | 用于管控会话级计划缓存，具体信息请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节中关于max_plancache_memory_ratio的描述。      |
| 全局共享级 | global_plancache_memory_ratio | 用于管控全局共享级计划缓存，具体信息请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节中关于global_plancache_memory_ratio的描述。 |

- 视图中涉及列说明

表 7-63 计划缓存管控功能相关视图说明

| 视图名                                  | 涉及列说明                                                         |
|--------------------------------------|---------------------------------------------------------------|
| GS_ALL_PREPARED_STATEMENTS           | size：表示该PREPARE语句的计划占用内存大小，单位为Byte。<br>is_deleted：表示该计划是否被淘汰。 |
| DBE_PERF.GS_PLAN_CACHE_STAT          | size：表示该PREPARE语句的计划占用内存大小，单位为Byte。<br>is_deleted：表示该计划是否被淘汰。 |
| DBE_PERF.GLOBAL_PLANCACHE_STATISTICS | size：表示该GPC计划占用内存大小，单位为Byte。                                  |
| DBE_PERF.LOCAL_PLANCACHE_STATUS      | size：表示该GPC计划占用内存大小，单位为Byte。                                  |

### 7.35.4.3 操作指导

#### 7.35.4.3.1 会话级计划缓存管控

##### 前提条件

- 启用逻辑内存管理模块，将enable\_memory\_limit参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。设置成功后，重启数据库集群。
- 打开会话级计划缓存管控。  
将GUC参数max\_plancache\_memory\_ratio设置为大于0的取值。

## 语法

通过gsq创建PREPARE语句，或通过JDBC创建PREPARE语句。

- 通过gsq创建：  
PREPARE name [ ( data\_type [, ...] ) ] AS statement;
- 通过JDBC创建：  
PreparedStatement pstmt = connection.prepareStatement(sql);

## 注意事项

- 会话级计划缓存管控仅支持管控PREPARE语句和自动参数化创建的计划缓存，不支持管控通过存储过程创建的计划缓存。
- 缓存淘汰策略只在执行计划创建、重建后内存大小达到或超过上限后触发。

## 操作步骤

- 打开逻辑内存管理模块以及设置会话级计划缓存管控，将enable\_memory\_limit参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。设置成功后，重启数据库集群。  
SET max\_plancache\_memory\_ratio = 60;
- 创建大量PREPARE语句。
- 观察内存视图，在计划缓存大小达到阈值后不再增长，且PREPARE语句视图中，存在计划被淘汰的场景（即GS\_ALL\_PREPARED\_STATEMENTS视图的is\_deleted列值为t）。

## 示例

```
gaussdb=# CREATE TABLE plancache_memctl_tbl (col1 int, col2 varchar(10));
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
gaussdb=# INSERT INTO plancache_memctl_tbl (col1, col2) VALUES (1, 'test');
INSERT 0 1
gaussdb=# SET max_plancache_memory_ratio=60;
SET
gaussdb=# SET plan_cache_mode=force_generic_plan;
SET
gaussdb=# PREPARE p1 AS SELECT * FROM plancache_memctl_tbl WHERE col1=$1;execute p1(1);
PREPARE
col1 | col2
-----+-----
1 | test
(1 row)

gaussdb=# PREPARE p2 AS SELECT * FROM plancache_memctl_tbl WHERE col1=$1;execute p2(1);
PREPARE
col1 | col2
-----+-----
1 | test
(1 row)

-- 创建大量计划缓存（根据业务实际进行创建，示例中不做展示）。

-- 查看内存视图。
gaussdb=# SELECT contextname, sum(totalsize)/1024/1024 sum, count(*) count FROM
pv_session_memory_detail WHERE contextname ILIKE '%cachedplan%' GROUP BY contextname;
contextname | sum | count
-----+-----+-----
CachedPlan | .08398437500000000000 | 6
CachedPlanQuery | .02539062500000000000 | 6
```

```
CachedPlanSource | .035156250000000000 | 6
(3 rows)

-- 查看PREPARE语句。
gaussdb=# SELECT name,statement,size,is_deleted,valid FROM GS_ALL_PREPARED_STATEMENTS WHERE
sessionid = pg_current_sessid() ORDER BY name;
 name | statement | size | is_deleted | valid
-----+-----+-----+-----+-----
p1 | prepare p1 as select * from plancache_memctl_tbl where col1=$1; | 0 | t | f
p2 | prepare p2 as select * from plancache_memctl_tbl where col1=$1; | 0 | t | f
p3 | prepare p3 as select * from plancache_memctl_tbl where col1=$1; | 0 | t | f
p4 | prepare p4 as select * from plancache_memctl_tbl where col1=$1; | 0 | t | f
p5 | prepare p5 as select * from plancache_memctl_tbl where col1=$1; | 15360 | f | t
(5 rows)

gaussdb=# DROP TABLE plancache_memctl_tbl;
DROP TABLE
```

### 7.35.4.3.2 全局共享级计划缓存管控

#### 前提条件

- 开启PBE查询和存储过程中语句的执行计划进行缓存共享，将enable\_global\_plancache参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节。
- 启用逻辑内存管理模块，将enable\_memory\_limit参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。设置成功后，重启数据库集群。
- 开启GPC内存管控。  
将GUC参数global\_plancache\_memory\_ratio设置为大于0的值，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

#### 语法

通过gsqll创建PREPARE语句，或通过JDBC创建PREPARE语句。

- 通过gsqll创建：  
PREPARE name [ ( data\_type [, ...] ) ] AS statement;
- 通过JDBC创建：  
PreparedStatement pstmt = connection.prepareStatement(sql);

#### 注意事项

- GPC计划缓存存在并发访问，因此无法保证淘汰策略成功。
- 达到管控上限后且GPC计划淘汰失败时，再次创建的计划将退化为会话级。

#### 操作步骤

- 打开逻辑内存管理和会话级计划缓存管控，将enable\_memory\_limit参数设置为on，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 资源消耗 > 内存”章节。设置成功后，重启数据库集群。
- 将global\_plancache\_memory\_ratio参数设置为30，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。
- 创建大量PREPARE语句。
- 观察内存视图，在计划缓存大小达到阈值后不再增长。

## 示例

将global\_plancache\_memory\_ratio参数设置为30，具体设置请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 负载管理”章节。

gsql连接数据库：

```
gaussdb=# CREATE TABLE plancache_memctl_tbl (col1 int, col2 varchar(10));
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
gaussdb=# INSERT INTO plancache_memctl_tbl (col1, col2) VALUES (1, 'test');
INSERT 0 1
gaussdb=# SET plan_cache_mode=force_generic_plan;
SET
gaussdb=# PREPARE p1 AS SELECT * FROM plancache_memctl_tbl WHERE col1=$1;execute p1(1);
PREPARE
col1 | col2
-----+-----
1 | test
(1 row)

gaussdb=# PREPARE p2 AS SELECT * FROM plancache_memctl_tbl WHERE col1=$1;execute p2(1);
PREPARE
col1 | col2
-----+-----
1 | test
(1 row)

-- 创建大量计划缓存（根据业务实际进行创建，示例中不做展示）。

-- GPC计划内存大小。
gaussdb=# SELECT sum(size) FROM DBE_PERF.GLOBAL_PLANCACHE_STATUS;
sum

159744
(1 row)

gaussdb=# DROP TABLE plancache_memctl_tbl ;
DROP TABLE
```

### 7.35.4.4 常见问题

1. **问题现象：**打开计划缓存管控功能后执行业务语句，仍出现内存不可用的问题。

**解决办法：**请通过以下方式进行解决：

- a. 确认内存管控参数max\_plancache\_memory\_ratio和global\_plancache\_memory\_ratio是否为0，0表示关闭功能。
- b. 查看参数enable\_memory\_limit是否为off，该参数受max\_process\_memory、shared\_buffers和元数据信息等影响，可能被动设置为off，当enable\_memory\_limit被设置为off后，请排查max\_process\_memory和shared\_buffers是否设置不合理，enable\_memory\_limit为on时，计划缓存管控功能才能生效。
- c. 调小参数max\_plancache\_memory\_ratio和global\_plancache\_memory\_ratio的值，使缓存阈值大小降低，剩余更多的内存资源。

2. **问题现象：**打开计划缓存管控功能后执行业务语句，复用缓存发现计划跳变。

**解决办法：**请查看业务执行过程中是否有影响计划生成的参数变更。由于计划缓存的淘汰重建机制，该过程中如果有影响计划生成的参数变更，那么淘汰前和重建后的计划可能存在差异。

## 7.36 DATABASE LINK

### 7.36.1 特性概述

#### 功能定义

DBLink功能是一种跨数据库访问的功能，用户可以在本地数据库利用DATABASE LINK（数据库连接）与远程数据库建立连接，通过DATABASE LINK实现访问远程数据库的目标。

#### 用户场景

用户需要访问远程数据库表，提供查询，修改等操作。

#### 规格约束

详情请参见[规格约束](#)。

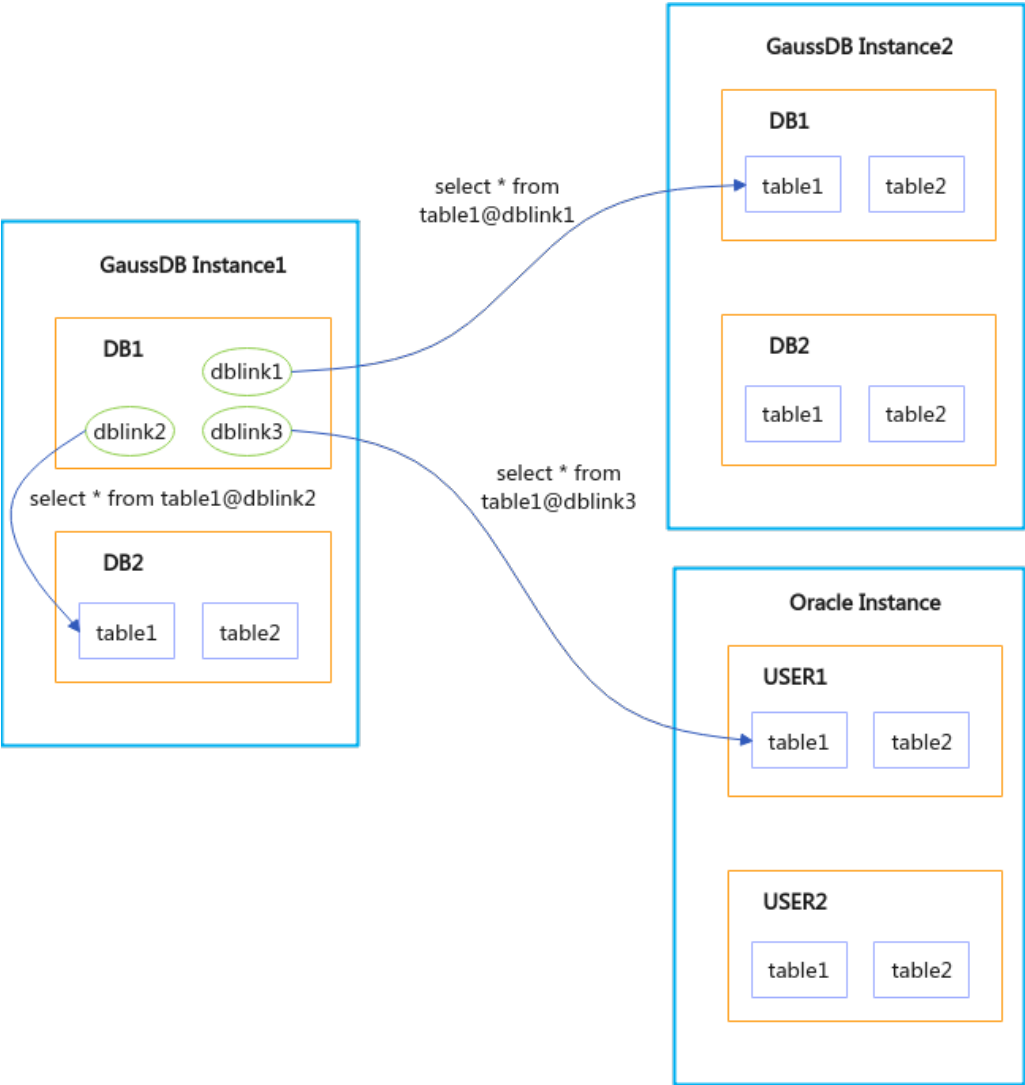
#### 技术架构

数据库连接（DATABASE LINK）通过创建DATABASE LINK对象，允许以对象名@dblink的语法访问远程数据库对象。该机制支持：

- 对远程表执行查询（SELECT）、更新（UPDATE）、插入（INSERT）等操作。
- 将远程对象映射为本地逻辑对象，使用标准SQL语法执行操作。
- 将远程对象与本地对象结合使用，实现跨数据库对象关联JOIN操作。

DATABASE LINK访问远程数据库的示意图如[图7-79](#)所示。可以在本地实例中访问当前实例中的不同DATABASE中的对象，也可以访问不同实例中的对象。

图 7-79 DATABASE LINK 访问远程数据库的示意图

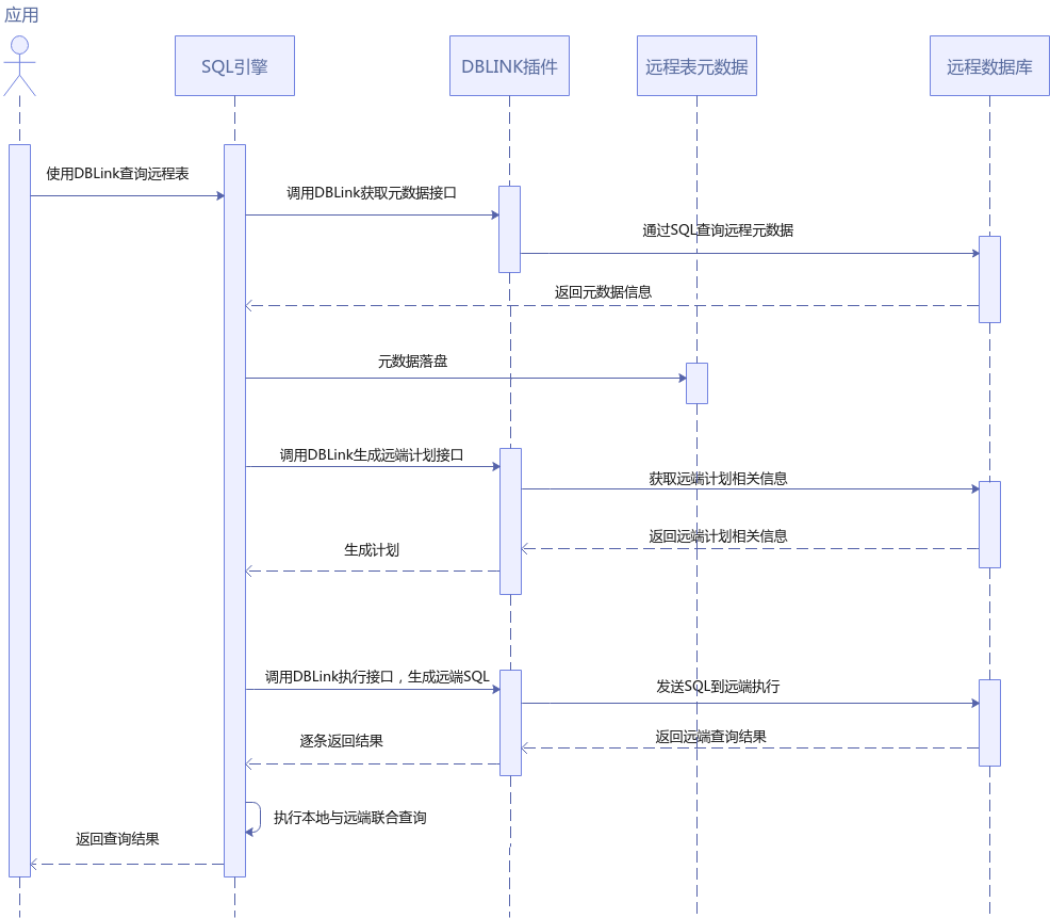


### 7.36.2 技术原理

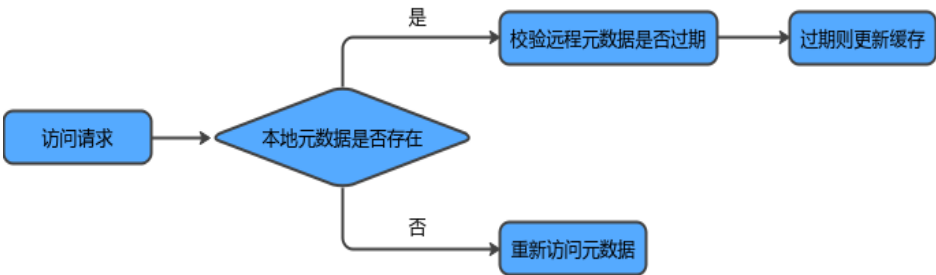
#### 核心机制

DATABASE LINK查询远程表基本流程的时序图如[图7-80](#)所示。

图 7-80 DATABASE LINK 查询远程表基本流程的时序图



- 语法解析阶段  
用户通过使用DATABASE LINK查询远程对象时，解析器会识别@dblink语法标识符，并生成对应的DBLink关系节点，标记远程对象属性。
- 语义分析阶段  
若目标表为远程对象（table@dblink）：
  - 数据库将调用DBLink插件接口，向远程数据库发起访问元数据信息请求，并在此过程中主动开启一个远程事务。
  - 获取到远程元数据并缓存至本地数据库系统表中。
  - 后续访问流程：



- 查询优化阶段  
优化器通过DBLink插件的接口生成向远程库发送的SQL语句，以获取查询代价和执行计划。



- 执行器阶段  
执行算子调用DBLink插件接口，向远程数据库发送SQL请求，获取远程结果集并序列化为本地数据格式，结果逐层向上传递。
- 事务提交阶段  
提交本地事务时自动同步远程事务状态。

## 接口介绍

DATABASE LINK主要使用的系统视图如表7-64所示，具体描述请参见《参考》中“系统表和系统视图 > 系统视图 > DATABASE LINK”章节。

表 7-64 系统视图

| 视图名         | 视图说明                                                                            |
|-------------|---------------------------------------------------------------------------------|
| GS_DB_LINKS | 显示DATABASE LINK对象的相关信息，用户可以查看属于自己和PUBLIC级别的DATABASE LINK信息。                     |
| V\$DBLINK   | 该视图数据取自于GS_DB_LINKS，但只记录在本次session中曾被使用过或正在使用的DATABASE LINK信息。默认只有管理员用户可以查看该视图。 |

DATABASE LINK主要使用的系统函数如表7-65所示，具体描述请参见《参考》中“SQL参考 > 函数和操作符 > DATABASE LINK函数”章节。

表 7-65 系统函数

| 函数名                                              | 函数说明                                                                                                                                                                                          |
|--------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| close_database_link(dblink_name name)            | 通过DBLink名称主动关闭当前session下该DBLink的连接。                                                                                                                                                           |
| gs_ora_diag(dblink_name name DEFAULT NULL::name) | 该函数只能获取连接Oracle库的DBLink对应的信息。通过DBLink名称获取gs_ora_fdw版本、GaussDB版本、Oracle数据库客户端版本以及Oracle数据库服务端版本信息。当参数为NULL时，如果查询用户是管理员用户还会返回ORACLE_HOME、ORACLE_SID、TNS_ADMIN、TWO_TASK、LDAP_ADMIN等信息，非管理员用户不显示。 |
| gs_ora_get_handle_alloc(dblink_oid oid)          | 通过DBLink的OID，获取该DBLink通过OCI使用的内存大小，单位为字节。                                                                                                                                                     |
| gs_ora_get_handle_alloc_all()                    | 获取每个节点的环境句柄，所有DBLink通过OCI使用的内存大小计算总和，单位为字节。                                                                                                                                                   |
| gs_ora_connection_status(dblink_oid oid)         | 通过DBLink的OID，获取该DBLink在当前session下的连接状态。                                                                                                                                                       |
| is_dblink_in_use(dblink_oid oid)                 | 通过DBLink的OID，获取该DBLink是否被使用。                                                                                                                                                                  |

| 函数名                                          | 函数说明                                          |
|----------------------------------------------|-----------------------------------------------|
| dblink_has_updatasent<br>(dblink_oid oid)    | 判断当前事务中是否使用OID对应的DATABASE LINK进行发送了DML语句且未提交。 |
| is_dblink_in_transaction(<br>dblink_oid oid) | 通过DBLink的OID，获取该DBLink是否正在事务中。                |

DATABASE LINK主要使用的系统表如表7-66所示，具体描述请参见《参考》中“系统表和系统视图 > 系统表 > DATABASE LINK”章节。

表 7-66 系统表

| 表名               | 描述                                                                                   |
|------------------|--------------------------------------------------------------------------------------|
| GS_DATABASE_LINK | 该表用于存储DATABASE LINK信息的系统表，主要记录的是DATABASE LINK对象的详细信息。只有具备SYSADMIN权限的用户才可以对该系统表进行读操作。 |

## 7.36.3 操作指导

### 前置操作

- 连接GaussDB数据库时，需要在远程数据库节点机器上设置允许DATABASE LINK进行访问，并配置远程连接。某些情况集群白名单中也需要添加DN的IP。
- 创建DATABASE LINK。
  - 创建DATABASE LINK对象，具体语法请参见《参考》中“SQL参考 > SQL语法 > C > CREATE DATABASE LINK”章节。
  - 创建DATABASE LINK时可指定PUBLIC或PRIVATE，PRIVATE DATABASE LINK仅能被创建者进行访问，PUBLIC DATABASE LINK可被所有用户进行访问。所有已创建的DATABASE LINK信息都存放在本地数据库的系统视图GS\_DB\_LINKS中。
- 修改DATABASE LINK信息。

修改指定DATABASE LINK对象信息，具体语法请参见《参考》中“SQL参考 > SQL语法 > A > ALTER DATABASE LINK”章节。
- 删除指定DATABASE LINK。

删除指定DATABASE LINK对象，具体语法请参见《参考》中“SQL参考 > SQL语法 > D > DROP DATABASE LINK”语句。
- 通过DATABASE LINK进行SELECT操作。

```
[WITH [RECURSIVE] with_query [, ...]]
SELECT [/*+ plan_hint */] [ALL | DISTINCT [ON (expression [, ...])]]
{ * | {expression [[AS] output_name]} [, ...] }
[FROM from_item [, ...]]
[WHERE condition]
[[START WITH condition] CONNECT BY [NOCYCLE] condition [ORDER SIBLINGS BY expression]]
[GROUP BY grouping_element [, ...]]
[HAVING condition [, ...]]
[{ UNION | INTERSECT | EXCEPT | MINUS } [ALL | DISTINCT] select]
[ORDER BY {expression [[ASC | DESC | USING operator] | nlssort_expression_clause] [NULLS
```

```
{ FIRST | LAST } } [, ...]
[LIMIT { [offset,] count | ALL }]
[OFFSET start [ROW | ROWS]]
[{ FOR { UPDATE | SHARE } [OF table_name [, ...]] [...] }];
{ [ONLY] table_name [*] @ dblink [[AS] alias [(column_alias [, ...])]]
| (select) [AS] alias [(column_alias [, ...])]
| with_query_name [[AS] alias [(column_alias [, ...])]]
| [function_name] ([argument [, ...]] [AS] alias [(column_alias [, ...] | column_definition [, ...])]
| [function_name] ([argument [, ...]]) AS (column_definition [, ...])
| from_item [NATURAL] join_type from_item [ON join_condition | USING (join_column [, ...])] }];
```

- 通过DATABASE LINK进行INSERT操作。

```
[WITH [RECURSIVE] with_query [, ...]]
INSERT [/*+ plan_hint */] INTO table_name @ dblink [(column_name [, ...])]
{ DEFAULT VALUES
| VALUES { ({ expression | DEFAULT } [, ...]) } [, ...]
| query }
[RETURNING { { output_expression [[AS] output_name] } [, ...] }];
```

- 通过DATABASE LINK进行UPDATE操作。

```
UPDATE [/*+ plan_hint */] [ONLY] table_name @ dblink [[AS] alias]
SET { column_name = { expression | DEFAULT }
| (column_name [, ...]) = { ({ expression | DEFAULT } [, ...]) | sub_query } } [, ...]
[FROM from_list] [WHERE condition]
[ORDER BY { expression [[ASC | DESC | USING operator] | nlssort_expression_clause] [NULLS
{ FIRST | LAST } }] [, ...]]
[LIMIT { [offset,] count | ALL }]
[RETURNING { { output_expression [[AS] output_name] } [, ...] }];
```

where sub\_query can be:

```
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
{ * | { expression [[AS] output_name] } [, ...] }
[FROM from_item [, ...]]
[WHERE condition]
[GROUP BY grouping_element [, ...]]
[HAVING condition [, ...]];
```

- 通过DATABASE LINK进行DELETE操作。

```
[WITH [RECURSIVE] with_query [, ...]]
DELETE [/*+ plan_hint */] FROM [ONLY] table_name @ dblink [[AS] alias]
[USING using_list]
[WHERE condition]
[ORDER BY { expression [[ASC | DESC | USING operator] | nlssort_expression_clause] [NULLS
{ FIRST | LAST } }] [, ...]]
[LIMIT { [offset,] count | ALL }]
[RETURNING { { output_expr [[AS] output_name] } [, ...] }];
```

- 通过DATABASE LINK进行LOCK TABLE操作。

```
LOCK [TABLE] { [ONLY] name @ dblink [, ...] }
[IN { ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE |
SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE } MODE]
[NOWAIT];
```

- 调用远程数据库的存储过程或函数。

```
CALL | SELECT [schema.] { func_name@dblink | procedure_name@dblink } (param_expr);
```

## 📖 说明

- 使用已创建的DATABASE LINK对远程数据库对象进行访问的语法和访问本地对象的语法基本一致，区别在于，当访问远程数据库对象时，需在被访问的远程数据库对象名称后添加@dblink。SQL语句具体支持情况存在部分约束条件，具体请参见[表7-68](#)。
- 上述语法中SQL语句涉及到的DATABASE LINK无关参数含义与原SQL语句中含义相同。
- 执行涉及到远程表的连接查询，需要指定列名时，可以在列名后添加“@dblink”，表示指定的列为DATABASE LINK指向的远程表的列，远程表的列不支持\*写法，如：  
remote\_t1.\*@dblink。

## 规格约束

- 兼容性约束

- DATABASE LINK特性仅在ORA兼容模式数据库下可以使用。
- DATABASE LINK特性只支持连接GaussDB。
- 当通过DATABASE LINK连接GaussDB时，用户需要保证本地和远程数据库的兼容性参数DBCOMPATIBILITY和GUC参数behavior\_compat\_options、a\_format\_dev\_version、a\_format\_version取值一致。
- 连接GaussDB时，DATABASE LINK开启远程连接事务前，会自动设置如下GUC参数：

```
SET search_path=pg_catalog, '$user', 'public';
SET explain_perf_mode='normal';
SET session_timeout=0;
SET datestyle=ISO;
SET intervalstyle=postgres;
SET extra_float_digits=3;
```

### 说明

其余参数为远程数据库设置的参数，远程参数与本地参数不同时，可能会出现数据显示格式不一致等情况，使用时应尽量保证远程与本地参数相同。

- 连接GaussDB时，远程数据库与本地数据库需要保证相同的datea类型，GUC参数mapping\_date\_to\_datea才能正常使用。

- 字符集约束

当连接GaussDB时，如果本地数据库和远程数据库字符集不同，可能会出现无法转换，并返回远程的报错信息的情况。当本地数据库字符编码为GB18030\_2022时，发送到远程的请求字符编码会被转换为GB18030。因此，若本地数据库的字符集为GB18030\_2022时，远程数据库字符集只能是GB18030或GB18030\_2022。

- 权限约束

- 禁止使用DATABASE LINK连接初始用户。禁止初始用户进行创建、修改和删除DATABASE LINK对象的操作。
- 创建DATABASE LINK权限需要使用GRANT语法赋予，新建用户默认无权限，系统管理员拥有权限。具体请参见《参考》中“SQL参考 > SQL语法 > G > GRANT”章节。
- 当赋予用户创建DATABASE LINK权限时，即默认许可用户使用远程数据库的IP地址对其进行访问。若不希望存在该情况，请不要使用GRANT对用户赋权。
- 本地用户对DATABASE LINK的使用权限：  
如果指定了PUBLIC关键词，即为公有的DATABASE LINK，可以被所有用户/模式使用。  
如果未指定PUBLIC关键词，就是私有的DATABASE LINK，仅能被当前用户/模式使用（包括SYSADMIN用户也无法跨Schema使用DATABASE LINK）。
- 通过DATABASE LINK访问远程数据库对象的权限：  
通过DATABASE LINK绑定的远程连接用户的权限，进行远程数据库对象的访问。

- 连接约束

- 当未指定CURRENT\_USER或CONNECT TO连接串时（即使用当前数据库初始用户名和空的密码连接），会连接失败。

- 非透明多写特性下，使用本机IP及对应端口可连接当前实例数据库。由于CN和DN可能分布在不同服务器中，建议host参数不要使用127.0.0.1和localhost，可能会出现连接失败的情况。
- DATABASE LINK创建时，不会对其是否可连接成功进行验证，如果缺乏相关的关键字，可能会在使用时报错。
- 如果在创建DATABASE LINK对象后重新生成密钥文件，使用DATABASE LINK时将会报错。因为创建DATABASE LINK时使用的密钥文件与后续解密时使用的密钥文件不一致。
- OPTION选项中conn\_opt不支持设置密码且传入的libpq参数不能和原有参数重复。
- 升级约束  
升级未提交情况下，无法创建使用DATABASE LINK。
- 元数据和锁相关约束
  - 使用DATABASE LINK对远程表操作时，会在本地创建与远程对应的Schema，若本地不存在该表的元数据信息，会将元数据信息写入本地系统表中，此时会使用7级锁保证写入的一致性，持续到事务结束后释放锁，删除DATABASE LINK时会将相应的元数据信息删除。
  - 使用DATABASE LINK时，在本地创建的表仅用于存储远程表的元数据信息，无法通过\d或pg\_get\_tabledef函数查询该表的结构。
  - 如果业务中存在长事务首次使用DATABASE LINK操作远程对象时，会持续持锁直到事务结束，其他首次使用DATABASE LINK的事务会被阻塞。可通过对远程对象进行简单查询操作，使其元数据快速缓存到本地进行规避该情况。另外，远程表结构发生变化时，本地要更新存储的元数据信息，也会有类似阻塞情况。
  - 在本地创建与远程对应的Schema时，本地的Schema名称长度上限为63，格式为“[username#]远程Schema@dblink\_name”，其中DATABASE LINK为PRIVATE时，username#不可省略。
  - 由于部分谓词无法下推，通过DATABASE LINK在使用UPDATE、DELETE及FOR UPDATE功能时，加锁的范围可能会扩大，可通过打印计划查询下推语句判断加锁范围。
  - 使用DATABASE LINK对远程表操作时，会创建一个单节点的node group随机绑定一个DN。
- 事务约束  
使用DATABASE LINK时，本地事务和远程事务存有以下关系：
  - 本地事务会同步控制远程事务的提交/回滚状态。
  - 隔离级别的对应关系如表7-67所示。

**表 7-67** 连接非透明多写特性下 GaussDB 数据库隔离级别对应关系表

| 本地隔离级别           | 远程隔离级别          |
|------------------|-----------------|
| Read Uncommitted | Repeatable Read |
| Read Committed   | Repeatable Read |
| Repeatable Read  | Repeatable Read |
| Serializable     | Serializable    |

## 说明

- 本地事务提交过程中会向远程发送事务提交请求，如果远程事务提交成功后出现异常情况导致本地的事务提交失败（如连接异常、本地集群实例异常等情况），远程的事务提交无法被撤回，可能出现本地事务与远程事务不一致的情况。
- DATABASE LINK不支持XA协议接口，可能出现本地事务与远程事务不一致的情况。
- 支持SQL范围约束
  - DATABASE LINK相关语句支持的情况如表7-68所示。
  - DATABASE LINK相关表类型支持情况如表7-69所示。
- DATABASE LINK函数和存储过程调用约束
  - 仅在非透明多写特性下支持通过DATABASE LINK连接调用远程函数或者存储过程。
  - 当通过DATABASE LINK调用远程数据库中的函数时，本地数据库需创建与远程函数名称和参数列表完全一致的函数，若远程函数存在重载，则需要在本本地数据库创建对应的所有重载函数，否则会导致参数匹配失败。
  - DATABASE LINK调用远程数据库的存储过程和函数时，参数或返回值类型不能包含自定义类型，不支持OUT/INOUT参数或有默认值的参数，并且不能为PACKAGE内函数、聚集函数、窗口函数以及返回集合的函数；不支持使用SELECT \* FROM func@dblink();形式调用远程函数。
  - PLSQL\_BODY内通过DATABASE LINK调用远程数据库的存储过程或函数时，参数或返回值类型不能包含自定义类型，不支持OUT/INOUT参数或有默认值的参数，并且不能为PACKAGE内函数、重载函数、聚集函数、窗口函数以及返回集合的函数。
  - DATABASE LINK调用远程数据库的存储过程和函数不指定Schema，默认调用PUBLIC下的函数或存储过程。
  - DATABASE LINK调用远程数据库的存储过程和函数时，参数列表不支持通过“:=”或者“=>”的方式为参数赋值。

## 说明

- PLSQL\_BODY内调用远程数据库的存储过程或函数时，应使用[CALL | SELECT] [ schema. ] { func\_name@dblink | procedure\_name@dblink } ( param\_expr )的语法格式调用。
- PLSQL\_BODY内调用远程数据库的无参存储过程或函数时，应使用[CALL | SELECT] [ schema. ] { func\_name@dblink | procedure\_name@dblink } ( )的语法格式调用。
- 同义词
  - 不支持将DATABASE LINK名创建一个同义词的使用方法。
  - 当使用DATABASE LINK访问的远程对象创建为同义词时，若未指定远程对象的Schema，则默认采用创建同义词的当前Schema作为远程对象的Schema。
  - 不支持通过DATABASE LINK调用远程数据库中指向一个DATABASE LINK对象的同义词。例如：
    - i. 在数据库a（db1）中创建表a（table1）。
    - ii. 在数据库b（db2）中创建连接数据库a（db1）的DATABASE LINK对象（dblink1），并创建同义词（CREATE SYNONYM table11 FOR user1.table1@dblink1;）。
    - iii. 在数据库c（db3）创建连接数据库b（db2）的DATABASE LINK对象（dblink2），通过dblink2调用数据库b（db2）上的同义词table11（执行SELECT \* FROM table11 @dblink2;，会返回 “If the remote



synonym is a dblink object, dblink access is not supported” 的报错信息）。

- 创建同义词的注意事项请参见《参考》中“SQL参考 > SQL语法 > C > CREATE SYNONYM”章节。

- 表类型约束

连接GaussDB时，表类型约束如下：

- HASHBUCKET：不支持通过DATABASE LINK对远程Hash bucket表进行查询或DML操作。
- SLICE：不支持通过DATABASE LINK对远程SLICE表进行查询或DML操作。
- 复制表：不支持通过DATABASE LINK对远程复制表进行查询或DML操作。
- TEMPORARY：不支持通过DATABASE LINK对远程临时表进行查询或DML操作。

- 视图约束

目前支持对DATABASE LINK的远程表创建视图，当远程表本身的结构发生变化时，该视图使用时会触发视图重编译，支持视图失效重编译后均为该行为，不受参数enable\_view\_invalidation影响。例如：

- a. 在数据库db1中创建表table1。
- b. 在数据库db2中创建连接数据库db1的DATABASE LINK对象dblink1，并创建视图view1（CREATE VIEW view1 AS SELECT \* FROM table1@dblink1;）。
- c. 在数据库db1中删除表table1的一列，在数据库db2上查询视图view1会重编译该视图，且重编译失败，并产生重编译失败报错。
- d. 在数据库db1中恢复表table1删除列，在数据库db2上查询视图view1会重编译该视图，且重编译成功，并正常返回查询结果。

### 说明

视图失效重编译及相关参数说明请参见[失效重编译](#)章节。

- 其他场景

- DATABASE LINK表不支持触发器，包括触发器调用函数内使用DATABASE LINK场景、触发器调用函数为DATABASE LINK函数、在DATABASE LINK上定义触发器等情况。
- 暂不支持UPSERT、MERGE语法。
- 不支持CURRENT CURSOR语法。
- 不支持查询表的隐藏字段。
- 不支持DATABASE LINK访问远端类型。
- 直连DN无法使用DATABASE LINK功能。

- DUMP与备份约束

- 不支持DATABASE LINK相关数据库对象的DUMP，备机不支持DATABASE LINK调用，也不支持被DATABASE LINK连接。
- 不支持DATABASE LINK相关数据库对象的集群备份后恢复使用。因为不同集群的密钥文件不同，在使用 DATABASE LINK 时需要使用集群密钥文件进行解密。

- JOIN下推约束

DATABASE LINK不支持JOIN下推。

- 谓词下推约束

- 仅支持WHERE子句使用数据库内置的数据类型、操作符和函数，并且使用的函数为IMMUTABLE类型。
- 不支持WHERE子句中同时使用到多张表的条件下推。
- 聚集函数下推约束  
仅支持单表且没有GROUP子句、ORDER BY子句、HAVING子句、LIMIT子句的SELECT语句，并且不支持窗口函数。
- HINT下推  
支持针对DATABASE LINK表对象的Hint条件下推，仅限Scan方式的Hint下推，语法格式如下：  
[no] tablescan|indexscan|indexonlyscan(table [index])  
并要求在一个queryblock中的表名或表别名不能重复。

表 7-68 支持 SQL 范围

| SQL类型                  | 操作对象              | 支持选项说明        | 执行上下文                           |
|------------------------|-------------------|---------------|---------------------------------|
| 创建<br>DATABASE<br>LINK | DATABAS<br>E LINK | -             | 普通事务<br>块、存储过<br>程、函数以<br>及高级包。 |
| 修改<br>DATABASE<br>LINK | DATABAS<br>E LINK | 仅支持用户名、密码的修改。 | 普通事务<br>块、存储过<br>程、函数以<br>及高级包。 |
| 删除<br>DATABASE<br>LINK | DATABAS<br>E LINK | -             | 普通事务<br>块、存储过<br>程、函数以<br>及高级包。 |



| SQL类型    | 操作对象            | 支持选项说明                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | 执行上下文                    |
|----------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| SELECT语句 | 普通表、普通视图、全量物化视图 | <ul style="list-style-type: none"> <li>WHERE子句</li> <li>DATABASE LINK表和内部表JOIN</li> <li>DATABASE LINK表和DATABASE LINK表JOIN</li> <li>聚集函数</li> <li>ORDER BY子句</li> <li>WINDOW子句</li> <li>LIMIT、OFFSET、FETCH子句</li> <li>GROUP BY子句、HAVING子句</li> <li>UNION子句</li> <li>WITH子句</li> <li>START WITH子句和CONNECT BY子句</li> <li>ROWNUM使用</li> <li>PIVOT子句</li> <li>UNPIVOT子句</li> <li>FOR UPDATE子句</li> </ul> <p><b>说明</b><br/>仅支持FOR UPDATE、FOR UPDATE NOWAIT、FOR UPDATE WAIT用法。</p> | 普通事务块、存储过程、函数、高级包以及逻辑视图。 |
| INSERT语句 | 普通表             | <ul style="list-style-type: none"> <li>WITH子句</li> <li>多VALUE插入</li> <li>RETURNING子句</li> </ul> <p><b>说明</b></p> <ul style="list-style-type: none"> <li>不支持ON DUPLICATE KEY UPDATE子句。</li> <li>不支持INSERT ALL语句。</li> </ul>                                                                                                                                                                                                                                                  | 普通事务块、存储过程、函数以及高级包。      |
| UPDATE语句 | 普通表             | <ul style="list-style-type: none"> <li>WITH子句</li> <li>LIMIT子句</li> <li>ORDER BY子句</li> <li>WHERE子句</li> <li>RETURNING子句</li> </ul> <p><b>说明</b><br/>不支持多表UPDATE操作。</p>                                                                                                                                                                                                                                                                                                     | 普通事务块、存储过程、函数以及高级包。      |

| SQL类型        | 操作对象 | 支持选项说明                                                                                                                                                     | 执行上下文               |
|--------------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| DELETE语句     | 普通表  | <ul style="list-style-type: none"><li>WITH子句</li><li>LIMIT子句</li><li>ORDER BY子句</li><li>WHERE子句</li><li>RETURNING子句</li></ul> <p>说明<br/>不支持多表DELETE操作。</p> | 普通事务块、存储过程、函数以及高级包。 |
| LOCK TABLE语句 | 普通表  | <ul style="list-style-type: none"><li>LOCKMODE子句</li><li>NOWAIT子句</li></ul>                                                                                | 普通事务块。              |

 说明

对于分区表，支持PARTITION子句指定分区。

表 7-69 连接非透明多写特性的 GaussDB 数据库表类型支持范围

| 维度         | GaussDB表类型               |        | DATABASE LINK支持情况 |
|------------|--------------------------|--------|-------------------|
| TEMP选项     | 临时表                      |        | 不支持。              |
|            | 全局临时表                    |        | 不支持。              |
| UNLOGGED选项 | 非日志表                     |        | 支持。               |
| 存储特性       | 行存                       | Astore | 支持。               |
|            |                          | Ustore | 不支持。              |
|            | 分区表                      |        | 不支持。              |
|            | 二级分区表                    |        | 不支持。              |
| 视图         | DATABASE LINK访问远程视图      |        | 支持SELECT，不支持DML。  |
|            | 本地视图通过DATABASE LINK关联远程表 |        | 支持SELECT，不支持DML。  |

使用示例

```
连接GaussDB数据库
-- DATABASE LINK相关DDL语句。
-- 前置操作。
gaussdb=# CREATE USER jack WITH PASSWORD '*****';
CREATE ROLE
-- 赋予用户创建DATABASE LINK对象的权限。
gaussdb=# GRANT CREATE PUBLIC DATABASE LINK TO jack;
GRANT
```

```
-- 赋予用户删除DATABASE LINK对象的权限。
gaussdb=# GRANT DROP PUBLIC DATABASE LINK TO jack;
GRANT ROLE
-- 赋予用户修改DATABASE LINK对象的权限。
gaussdb=# GRANT ALTER PUBLIC DATABASE LINK TO jack;
GRANT ROLE
-- 回收用户创建DATABASE LINK对象的权限。
gaussdb=# REVOKE CREATE PUBLIC DATABASE LINK FROM jack;
REVOKE
-- 回收用户删除DATABASE LINK对象的权限。
gaussdb=# REVOKE DROP PUBLIC DATABASE LINK FROM jack;
REVOKE ROLE
-- 回收用户修改DATABASE LINK对象的权限。
gaussdb=# REVOKE ALTER PUBLIC DATABASE LINK FROM jack;
REVOKE ROLE
-- 清理环境。
gaussdb=# DROP USER jack;
DROP ROLE

-- 前置操作。
-- 创建一个兼容性为ORA的数据库。
gaussdb=# CREATE DATABASE ora_test_db DBCOMPATIBILITY 'ORA';
CREATE DATABASE
-- 切换数据库。
gaussdb=# \c ora_test_db
-- 创建拥有系统管理员权限的用户。
ora_test_db=# CREATE USER local_user WITH SYSADMIN PASSWORD '*****';
CREATE ROLE
ora_test_db=# SET ROLE local_user PASSWORD '*****';
SET
-- 创建DATABASE LINK对象，host也可以是IPv6地址。
ora_test_db=> CREATE PUBLIC DATABASE LINK dblink CONNECT TO 'remote_user' IDENTIFIED BY '*****'
USING (host '***.***.***.***', port '*****', dbname 'remote_db');
CREATE DATABASE LINK
-- 修改DATABASE LINK信息。
ora_test_db=> ALTER PUBLIC DATABASE LINK dblink CONNECT TO 'remote_user' IDENTIFIED BY '*****';
ALTER DATABASE LINK
-- 删除DATABASE LINK对象。
ora_test_db=> DROP PUBLIC DATABASE LINK dblink;
DROP DATABASE LINK

-- 清理环境。
ora_test_db=> RESET ROLE;
RESET
ora_test_db=# DROP USER local_user;
DROP ROLE
ora_test_db=# \c postgres
gaussdb=# DROP DATABASE ora_test_db;
DROP DATABASE

-- DATABASE LINK具体操作语句。
-- 前置操作。
gaussdb=# CREATE USER local_user WITH SYSADMIN PASSWORD '*****';
CREATE ROLE
gaussdb=# CREATE USER remote_user WITH SYSADMIN PASSWORD '*****';
CREATE ROLE
-- 创建远程数据库。
gaussdb=# CREATE DATABASE remote_db DBCOMPATIBILITY = 'ORA';
CREATE DATABASE
-- 创建测试DATABASE LINK数据库。
gaussdb=# CREATE DATABASE local_db DBCOMPATIBILITY = 'ORA';
CREATE DATABASE
gaussdb=# \c remote_db
remote_db=# SET ROLE remote_user PASSWORD '*****';
SET
remote_db=> CREATE SCHEMA remote_user;
CREATE SCHEMA
-- 创建普通表。
remote_db=> CREATE TABLE remote_tb(f1 int, f2 text, f3 text[]) WITH (STORAGE_TYPE = ASTORE);
```

```
CREATE TABLE
remote_db=> INSERT INTO remote_tb VALUES (0,'a',{'a0',"b0","c0"});
INSERT 0 1
remote_db=> INSERT INTO remote_tb VALUES (1,'bb',{'a1',"b1","c1"});
INSERT 0 1
remote_db=> INSERT INTO remote_tb VALUES (2,'cc',{'a2',"b2","c2"});
INSERT 0 1

-- 创建函数。
remote_db=> CREATE OR REPLACE FUNCTION f(a in int, b in int)
RETURN int AS
 tmp int := a + b;
BEGIN
 RETURN tmp;
END;
/
CREATE FUNCTION

-- 创建同义词。
remote_db=> CREATE SYNONYM remote_sy FOR remote_tb;
CREATE SYNONYM

remote_db=> \c local_db
local_db=# SET ROLE local_user PASSWORD '*****';
SET
local_db=> CREATE SCHEMA local_user;
CREATE SCHEMA
local_db=> CREATE TABLE local_tb(f1 int, f2 text, f3 text[]) WITH (STORAGE_TYPE = ASTORE);
CREATE TABLE
local_db=> INSERT INTO local_tb VALUES (2,'c',{'a2',"b2","c2"});
INSERT 0 1
local_db=> CREATE PUBLIC DATABASE LINK dblink CONNECT TO 'remote_user' IDENTIFIED BY '*****'
USING (HOST '***.***.***.***', PORT '*****', DBNAME 'remote_db'); --host和port需要根据实际情况填写。
CREATE DATABASE LINK
-- 查询远程表。
local_db=> SELECT * FROM remote_tb@dblink ORDER BY f1;
 f1 | f2 | f3
-----+-----
 0 | a | {a0,b0,c0}
 1 | bb | {a1,b1,c1}
 2 | cc | {a2,b2,c2}
(3 rows)
-- 向远程表插入数据。
local_db=> INSERT INTO remote_tb@dblink VALUES (4,'d',{'a1',"b2","c3"});
INSERT 0 1
-- 更新远程表。
local_db=> UPDATE remote_tb@dblink SET f2 = 'aa' WHERE f1 = 0;
UPDATE 1
-- 删除远程表数据。
local_db=> DELETE remote_tb@dblink WHERE f1 = 1;
DELETE 1
-- 本地表JOIN远程表。
local_db=> SELECT * FROM remote_tb@dblink JOIN local_tb ON local_tb.f1 = remote_tb.f1@dblink;
 f1 | f2 | f3 | f1 | f2 | f3
-----+-----+-----
 2 | cc | {a2,b2,c2} | 2 | c | {a2,b2,c2}
(1 row)
-- AGG函数。
local_db=> SELECT count(*) FROM remote_tb@dblink;
 count

 3
(1 row)
-- 访问远程函数。
local_db=> SELECT f@dblink(1,2);
 f

 3
(1 row)
```

```
-- PLSQL_BODY内访问远程函数。
local_db=> CREATE OR REPLACE FUNCTION call_f(a in int, b in int)
RETURN INT AS
 tmp int;
BEGIN
 tmp := f@dblink(a, b);
 RETURN tmp;
END;
/
CREATE FUNCTION
local_db=> SELECT call_f(1, 2);
call_f

 3
(1 row)
-- 创建DATABASE LINK对象的同义词。
local_db=> CREATE SYNONYM local_sy FOR remote_user.remote_tb@dblink;
CREATE SYNONYM
local_db=> SELECT * FROM local_sy ORDER BY f1;
f1 | f2 | f3
-----+-----
0 | aa | {a0,b0,c0}
2 | cc | {a2,b2,c2}
4 | d | {a1,b2,c3}
(3 rows)
-- 访问远程数据库的同义词。
local_db=> SELECT * FROM remote_sy@dblink ORDER BY f1;
f1 | f2 | f3
-----+-----
0 | aa | {a0,b0,c0}
2 | cc | {a2,b2,c2}
4 | d | {a1,b2,c3}
(3 rows)
-- DATABASE LINK支持部分Hint下推。
local_db=> EXPLAIN (VERBOSE, COSTS off) SELECT /*+ tablescan(remote_sy) */ * FROM remote_sy@dblink;
QUERY PLAN

Foreign Scan on "remote_user@dblink".remote_tb remote_sy
 Output: f1, f2, f3
 Exec Nodes: (dblink#nodegroup#0) datanode1
 Remote SQL: SELECT /*+ tablescan(remote_tb) */ f1, f2, f3 FROM remote_user.remote_tb
(4 rows)
-- 查看DATABASE LINK系统表gs_database_link。
local_db=> SELECT dlname,downer,options FROM gs_database_link;
dlname | downer | options
-----+-----+-----
dblink | 0 | {host=***.***.***.***,port=*****,dbname=remote_db}
(1 row)
-- 查看DATABASE LINK系统视图gs_db_links。
local_db=> START TRANSACTION;
START TRANSACTION
local_db=> SELECT * FROM remote_sy@dblink ORDER BY f1;
f1 | f2 | f3
-----+-----
0 | aa | {a0,b0,c0}
2 | cc | {a2,b2,c2}
4 | d | {a1,b2,c3}
(3 rows)
local_db=> SELECT intransaction FROM gs_db_links;
intransaction

t
(1 row)
local_db=> END;
COMMIT
-- 清理环境。
local_db=> \c postgres
```

```
gaussdb=# DROP DATABASE local_db;
DROP DATABASE
gaussdb=# DROP DATABASE remote_db;
DROP DATABASE
gaussdb=# DROP USER local_user;
DROP ROLE
gaussdb=# DROP USER remote_user;
DROP ROLE
```

## 7.36.4 常见问题

Q1：DATABASE LINK查询远程数据时，一直在等锁释放，这是为什么？

**问题现象：**DATABASE LINK查询远程数据时，一直在等锁。

**原因分析：**业务中存在长事务首次使用DATABASE LINK操作远程对象，会持续持锁直到事务结束，其他首次使用DATABASE LINK的事务会被阻塞。

**处理方法：**通过一条快速执行的语句，先对要使用的远程对象做查询操作，使其元数据信息缓存至本地进行规避该情况。

## 7.37 单节点表

### 7.37.1 特性概述

单节点表是分布式数据库在企业级应用环境中针对数据量较小、难以进行有效分片的表所采用的优化方案。其核心特征包括：

- 采用复制表分布类型设计。
- 元数据定义全局分布（所有节点均可见），表结构信息全局同步至所有节点，保障跨节点SQL解析能力。
- 数据实体仅存储于指定的单个数据节点（DN）。

### 7.37.2 技术原理

#### 单节点表与节点组（Node Group）

- 基础绑定规则

单节点表必须且只能创建在系统预置的单节点组Node Group上，具体要求如[表 7-70](#)所示。

**表 7-70** 单节点组 Node Group 核心属性

| 属性类别 | 规范说明                                                                  |
|------|-----------------------------------------------------------------------|
| 类型标识 | 固定为's'类型。                                                             |
| 命名规则 | 前缀强制为sys_single_group，后缀为数字序号（如：sys_single_group1、sys_single_group2）。 |
| 成员限制 | 仅包含1个数据节点（分片主DN），形成物理隔离的单点存储单元。                                       |

| 属性类别 | 规范说明                                                     |
|------|----------------------------------------------------------|
| 创建方式 | 系统为每个DN自动创建单节点组Node Group，单节点表必须创建在系统预置的单节点组Node Group上。 |

- 版本升级兼容性规范（从低版本升级至507.0版本）
  - 升级预检阶段：
    - 冲突检测：
 

扫描所有用户自定义的节点组Node Group，校验是否存在使用保留前缀sys\_single\_group的非法命名。
    - 处置策略：
 

若存在冲突（即存在使用保留前缀sys\_single\_group的非法命名）：立即终止升级流程，建议进行修改冲突的节点组。

```
ALTER NODE GROUP sys_single_group_userdefined RENAME TO custom_group1;
```

无冲突：通过预检，进入升级流程。
  - 升级执行阶段：
    - 自动为所有'D'类型数据节点（DN），生成合规的单节点组Node Group。
    - 命名规则：sys\_single\_group\${seq}（如：DN2对应sys\_single\_group2）。
    - 建立元数据映射关系，确保表的级联关联。

## 单节点表支持 DDL 核心机制

1. 在单节点表配置中，存在两个关键reloptions配置选项：
  - is\_single\_node：用于标识该表是否为单节点表。
  - is\_only\_meta：用于标识当前数据节点（DN）是否仅存储元数据，而不存储实际数据。

### 说明

在仅存储元数据的数据节点上对单节点表执行DML操作：

- 如果操作中的表达式不为false或无法直接优化为false（如：对带有NOT NULL约束的列执行IS NULL条件判断）时，系统将报错。
- 如果操作中的表达式本身就是false或可以直接优化为false时，系统不会报错，而是返回空结果。

2. 单节点表的创建策略由以下GUC参数协同控制：
  - single\_node\_table\_strategy：定义创建单节点表的行为。
  - default\_nodegroup：指定创建Schema时默认设置单节点属性的功能。

### 说明

参数的具体使用请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。

3. 仅支持单节点表与分布表在线相互转换。

## 单节点表支持 DML 核心机制

1. 单节点表复用复制表的分布式属性，主要涉及SQL解析器、优化器、执行器适配。
2. SQL引擎在完成语法解析生成语法树后，进入语义解析阶段。语义解析负责对语法树进行语义检查（验证表存在性、列数据类型兼容性等语义规则）并构建单节点表的范围表，最终生成逻辑查询树（构建查询树时识别表类型（单节点/分布式）及其所属Node Group）。
3. 单节点表分布式计划支持语句下推（LightProxy、FQS）、计划下推（Stream计划）、部分语句下推（XC计划）。判断SQL语句是否可直接下推DN：内部遍历查询树，对每个查询树节点获取待下推的节点执行列表，并获取表对应的分布信息。对于单节点表，可通过所属的单节点组Node Group控制执行的DN范围。如果SQL中包含的单节点表或其他非单节点表不在一个DN上（如：通过JOIN交互），由于两个表对应的节点执行列表不相同，无法合并到同一DN上，导致语句无法下推从而执行Stream计划。对于DML场景，会对源表和目标表的分布情况进行判断，如果不一致，会在源表上增加Stream节点。

## 7.37.3 操作指导

### 7.37.3.1 单节点表 DDL 操作

#### 规格约束

1. 单节点表仅支持创建为复制表类型，不支持创建为分布表（指定分布列）。
2. 继承复制表约束：不支持创建全局二级索引（GSI），不支持定义为hash bucket表。
3. 仅支持在系统预置的单节点组Node Group上创建单节点表。
4. M-Compatibility模式数据库下不支持使用单节点表。
5. 从不支持单节点表的版本升级到新版本，升级观察期间，不支持默认创建单节点表。
6. 从不支持单节点表的版本升级到新版本，升级观察期间，新版本系统预置的单节点组Node Group上不允许进行创建表操作。
7. 当系统预置的单节点组Node Group上已创建单节点表，不支持对Node Group进行缩容。
8. 不支持物化视图、不支持外表。
9. 单节点表逻辑解码和备份恢复只支持同构场景，异构场景可能会导致错误。
10. 仅支持复制表基础上的本地临时表、unlogged表和全局临时表、一级分区表模式，其他表类型均不支持。

#### 创建单节点表

可通过以下方式进行创建单节点表，不同的创建方式需满足不同条件：

- 显式创建单节点表
  - 设置GUC参数single\_node\_table\_strategy的值不为FORBID。
  - 在CREATE TABLE语句中，显式指定TO GROUP group\_name子句，且指定的节点组为系统预定义的单节点组Node Group（类型标识为's'），其名称需以“sys\_single\_group”为前缀进行命名。如下所示：

```
CREATE TABLE table1 (id INT) TO GROUP sys_single_group1;
```



- 隐式创建单节点表
  - 在CREATE TABLE语句中，未明确指定TO GROUP group\_name子句时，系统自动选择一个类型为's'的节点组Node Group，其名称前缀为“sys\_single\_group”。
  - 设置GUC参数single\_node\_table\_strategy：
    - 当设置为ALLOW\_DEFAULT\_NON\_PART时，支持隐式创建非分区表类型的单节点表。
    - 当设置为ALLOW\_DEFAULT\_ALL，既支持隐式创建非分区表类型的单节点表，也支持隐式创建一级分区表类型的单节点表。
  - 待创建表归属的Schema具有单节点属性，具体请参见[设置Schema的单节点属性](#)相关描述。

### 📖 说明

GUC参数的具体使用请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。

#### 语法示例：

```
-- 显式创建单节点表。
gaussdb=# CREATE TABLE single_table(a int) DISTRIBUTE BY REPLICATION TO GROUP sys_single_group1;
CREATE TABLE

gaussdb=# \d+ single_table
 Table "public.single_table"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a | integer | | plain | |
Has OIDs: no
Distribute By: REPLICATION
Location Nodes: datanode1
Options: orientation=row, is_single_node=true, logical_repl_node=-1, compression=no,
storage_type=USTORE, segment=off

-- 将GUC参数single_node_table_strategy设置为allow_default_all,详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。
-- 设置完成后，查看single_node_table_strategy的值是否为allow_default_all。
gaussdb=# SHOW single_node_table_strategy;
single_node_table_strategy

allow_default_all
(1 row)

gaussdb=# CREATE SCHEMA schema1 SET NODE GROUP sys_single_group1;
CREATE SCHEMA

-- 隐式创建单节点表。
gaussdb=# CREATE TABLE schema1.single_table1(a int);
CREATE TABLE

gaussdb=# \d+ schema1.single_table1
 Table "schema1.single_table1"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 a | integer | | plain | |
Has OIDs: no
Distribute By: REPLICATION
Location Nodes: datanode1
Options: orientation=row, is_single_node=true, logical_repl_node=-1, compression=no,
storage_type=USTORE, segment=off

gaussdb=# DROP TABLE single_table;
DROP TABLE
gaussdb=# DROP TABLE schema1.single_table1;
```

```
DROP TABLE
gaussdb=# DROP SCHEMA schema1 CASCADE;
DROP SCHEMA
```

## 设置 Schema 的单节点属性

可通过以下方式进行设置Schema的单节点属性，不同的设置方式需满足不同条件：

- 显式设置Schema的单节点属性

在CREATE SCHEMA语句中，显式指定SET NODE GROUP group\_name子句，且指定的节点组为系统预定义的单节点组Node Group（类型标识为's'），其名称需以“sys\_single\_group”为前缀进行命名。如下所示：

```
CREATE SCHEMA schema1 SET NODE GROUP sys_single_group1;
```

- 隐式设置Schema的单节点

- 在CREATE SCHEMA语句中，未明确指定SET NODE GROUP group\_name子句时，系统自动选择一个类型为's'的节点组Node Group，其名称前缀为“sys\_single\_group”。
- 在普通用户下创建具有单节点属性的Schema，不支持在初始用户、具有运维/安全等管理员权限的用户下隐式设置Schema的单节点属性。
- 设置GUC参数default\_nodegroup的值为SCHEMA。

### 说明

- GUC参数的具体使用请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。
- CREATE SCHEMA语法包含CREATE TABLE子句的复合SQL语句不支持单节点表相关功能，既不支持单节点表创建，也不支持模式设置单节点表属性。具体语法请参见《参考》中“SQL参考 > SQL语法 > C > CREATE SCHEMA”章节。

### 语法示例：

-- 创建模式，显式设置单节点属性。

```
gaussdb=# CREATE SCHEMA schema1 SET NODE GROUP sys_single_group1;
CREATE SCHEMA
```

```
gaussdb=# SELECT * FROM pg_namespace WHERE nspname='schema1';
 nspname | nspowner | nsptimeline | nspacl | in_redistribution | nspblockchain | nspcollation | nspgroupoid
-----+-----+-----+-----+-----+-----+-----+-----
 schema1 | 10 | 0 | | f | | | 16389
(1 row)
```

```
gaussdb=# DROP SCHEMA schema1 CASCADE;
DROP SCHEMA
```

```
gaussdb=# CREATE USER test_usr PASSWORD "*****";
CREATE ROLE
```

```
gaussdb=# GRANT ALL ON DATABASE POSTGRES TO test_usr;
GRANT
```

-- 切换到test\_usr用户。

-- 将GUC参数default\_nodegroup设置为schema，详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。

-- 设置完成后，查看default\_nodegroup的值是否为schema。

```
gaussdb=> SHOW default_nodegroup;
default_nodegroup
```

```

 schema
(1 row)
```

-- 创建模式，隐式设置单节点属性。

```
gaussdb=> CREATE SCHEMA schema2;
CREATE SCHEMA
```

```
gaussdb=> SELECT * FROM pg_namespace WHERE nspname='schema2';
 nspname | nspowner | nsptimeline | nspacl | in_redistribution | nspblockchain | nspcollation | nspgroupoid
-----+-----+-----+-----+-----+-----+-----+-----
 schema2 | 16395 | 0 | n | f | | 16390
(1 row)

gaussdb=# DROP SCHEMA schema2 CASCADE;
DROP SCHEMA
```

## 修改 Schema 的单节点属性

- 修改单节点属性需满足以下条件：

在ALTER SCHEMA语句中，指定SET NODE GROUP group\_name子句，且指定的节点组为系统预定义的单节点组Node Group（类型标识为's'），其名称需以“sys\_single\_group”为前缀进行命名。如下所示：

```
ALTER SCHEMA schema1 SET NODE GROUP sys_single_group1;
```

- 删除单节点属性需满足以下条件：

ALTER SCHEMA语法指定DROP NODE GROUP子句。

语法示例：

```
gaussdb=# CREATE SCHEMA schema1;
CREATE SCHEMA
```

```
gaussdb=# SELECT * FROM pg_namespace WHERE nspname='schema1';
 nspname | nspowner | nsptimeline | nspacl | in_redistribution | nspblockchain | nspcollation | nspgroupoid
-----+-----+-----+-----+-----+-----+-----+-----
 schema1 | 10 | 0 | n | f | |
(1 row)
```

-- 修改Schema的单节点属性。

```
gaussdb=# ALTER SCHEMA schema1 SET NODE GROUP sys_single_group1;
ALTER SCHEMA
```

```
gaussdb=# SELECT * FROM pg_namespace WHERE nspname='schema1';
 nspname | nspowner | nsptimeline | nspacl | in_redistribution | nspblockchain | nspcollation | nspgroupoid
-----+-----+-----+-----+-----+-----+-----+-----
 schema1 | 10 | 0 | n | f | | 16389
(1 row)
```

-- 删除Schema的单节点属性。

```
gaussdb=# ALTER SCHEMA schema1 DROP NODE GROUP;
ALTER SCHEMA
```

```
gaussdb=# SELECT * FROM pg_namespace WHERE nspname='schema1';
 nspname | nspowner | nsptimeline | nspacl | in_redistribution | nspblockchain | nspcollation | nspgroupoid
-----+-----+-----+-----+-----+-----+-----+-----
 schema1 | 10 | 0 | n | f | |
(1 row)
```

```
gaussdb=# DROP SCHEMA schema1 CASCADE;
DROP SCHEMA
```

### 7.37.3.2 单节点表在线转换操作

#### 规格约束

- 单节点表仅支持创建为复制表类型，不支持创建为分布表（指定分布列）。
- 单节点表不支持创建全局二级索引（GSI）。含有GSI索引的分布表转化成单节点后，GSI索引会转换为普通唯一索引；单节点表转Hash分布表时，不包含分布键的唯一索引会转换为GSI索引。

3. 仅支持单节点表与分布表相互转换，不支持修改分布列。
4. 单节点表与分布表相互转换时，不支持逻辑复制。
5. 仅支持单节点表与分布表在线相互转换，不支持离线。
6. 单节点表转换为分布式表，仅支持转换为Hash分布、Range分布或List分布的分布表。同理，仅支持Hash分布表、Range分布表以及List分布表转化为单节点表。
7. 仅支持在系统预置的单节点组Node Group上创建单节点表。
8. 单节点表在线转换，仅支持普通表和分区表，其他表类型均不支持（包括二级分区表、段页式表、hash bucket表、临时表、unlogged表等）。
9. 单节点表和分布式表相互转换，DDL功能可并发读写。长事务可能会阻塞在线DDL，建议避免在长事务存在时进行在线DDL。本特性执行后期将阻塞DQL操作，以及阻塞DML操作。
10. 该特性同时受在线DDL特性约束影响。

## 前置条件

设置GUC参数single\_node\_table\_strategy不为FORBID，GUC参数的具体使用请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 单节点表”章节。

## 使用示例

- 分布表转单节点表

```
-- 创建分布表。
gaussdb=# CREATE TABLE test (c1 int, c2 int, c3 numeric(64,8)) WITH (storage_type = astore)
DISTRIBUTE BY HASH(c2) PARTITION BY RANGE (c2)
(PARTITION P1 VALUES LESS THAN (5),
PARTITION P2 VALUES LESS THAN (10),
PARTITION P3 VALUES LESS THAN (20));
CREATE TABLE
gaussdb=# \d+ test
Table "public.test"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1 | integer | | plain | |
c2 | integer | | plain | |
c3 | numeric(64,8) | | main | |
Partition By RANGE(c2)
Number of partitions: 3 (View pg_partition to check each partition range.)
Has OIDs: no
Distribute By: HASH(c2)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=astore, compression=no

-- 将分布表转化为单节点表。
gaussdb=# ALTER TABLE ONLINE test DISTRIBUTE BY REPLICATION TO GROUP sys_single_group1;
ALTER TABLE
-- 查看单节点表。
gaussdb=# \d+ test
Table "public.test"
Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
c1 | integer | | plain | |
c2 | integer | | plain | |
c3 | numeric(64,8) | | main | |
Partition By RANGE(c2)
Number of partitions: 3 (View pg_partition to check each partition range.)
Has OIDs: no
Distribute By: REPLICATION
Location Nodes: datanode1
Options: orientation=row, storage_type=astore, compression=no, is_single_node=true,
logical_repl_node=-1
```

- 单节点表转换为分布表

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

-- 创建单节点表。

```
gaussdb=# DROP TABLE IF EXISTS test;
DROP TABLE
gaussdb=# CREATE TABLE test (c1 int , c2 int, c3 numeric(64,8)) WITH (storage_type = astore)
DISTRIBUTE BY REPLICATION TO GROUP sys_single_group2 PARTITION BY RANGE (c2)
(PARTITION P1 VALUES LESS THAN (5),
PARTITION P2 VALUES LESS THAN (10),
PARTITION P3 VALUES LESS THAN (20));
CREATE TABLE
gaussdb=# \d+ test
```

| Column | Type          | Modifiers | Storage | Stats target | Description |
|--------|---------------|-----------|---------|--------------|-------------|
| c1     | integer       |           | plain   |              |             |
| c2     | integer       |           | plain   |              |             |
| c3     | numeric(64,8) |           | main    |              |             |

Partition By RANGE(c2)  
Number of partitions: 3 (View pg\_partition to check each partition range.)  
Has OIDs: no  
Distribute By: REPLICATION  
Location Nodes: datanode2  
Options: orientation=row, storage\_type=astore, is\_single\_node=true, logical\_repl\_node=-2, compression=no

-- 将单节点表转化为分布表。

```
gaussdb=# ALTER TABLE ONLINE test DISTRIBUTE BY HASH(c2);
ALTER TABLE
-- 查看分布表
gaussdb=# \d+ test
```

| Column | Type          | Modifiers | Storage | Stats target | Description |
|--------|---------------|-----------|---------|--------------|-------------|
| c1     | integer       |           | plain   |              |             |
| c2     | integer       |           | plain   |              |             |
| c3     | numeric(64,8) |           | main    |              |             |

Partition By RANGE(c2)  
Number of partitions: 3 (View pg\_partition to check each partition range.)  
Has OIDs: no  
Distribute By: HASH(c2)  
Location Nodes: ALL DATANODES  
Options: orientation=row, storage\_type=astore, compression=no

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

7.37.3.3 单节点表 DML 操作

前置条件

- 当前表是否为单节点表，可通过如下步骤确认是否正确创建单节点表。

```
-- 创建的单节点表t_ng1。
gaussdb=# CREATE TABLE t_ng1(c1 int, c2 int) DISTRIBUTE BY REPLICATION TO GROUP
sys_single_group1;
gaussdb=# \d+ t_ng1
```

| Column | Type    | Modifiers | Storage | Stats target | Description |
|--------|---------|-----------|---------|--------------|-------------|
| c1     | integer |           | plain   |              |             |
| c2     | integer |           | plain   |              |             |

Has OIDs: no  
Distribute By: REPLICATION  
Location Nodes: datanode1  
Options: orientation=row, is\_single\_node=true, logical\_repl\_node=-1, compression=no, storage\_type=USTORE, segment=off

```
gaussdb=# SELECT reloptions FROM pg_class WHERE relname = 't_ng1';
 reloptions

{orientation=row,is_single_node=true,logical_repl_node=-1,compression=no,storage_type=USTORE,segment=off}
(1 row)
```

- 当前单节点表所在节点组Node Group是否为单节点组sys\_single\_group，可通过如下步骤确认单节点表所属的节点组Node Group信息。  
--查询pgxc\_class系统表，确认单节点表所属节点组信息，其中group name必须是sys\_single\_group[%d]。  
gaussdb=# SELECT pgroup, nodeoids FROM pgxc\_class WHERE pcrelid = (SELECT oid FROM pg\_class WHERE relname = 't\_ng1');  
 pgroup | nodeoids  
-----+-----  
sys\_single\_group1 | 16385  
(1 row)  
gaussdb=# DROP TABLE t\_ng1;

## 规格约束

- 规格：
  - a. 单节点表支持INSERT、UPDATE、DELETE、UPSERT、MERGE、SELECT以及SELECT FOR UPDATE | SHARED（FOR UPDATE | SHARED场景和原Ustore/Astore约束保持一致）常用DML语法语句。
  - b. 单节点表支持和非单节点表（如：分布表、分区表）交互DML操作，分布表包括：复制表、Range分布表、List分布表、Hash分布表、hash bucket表以及单节点表；分区表包括：二级分区表以及一级分区表。
  - c. 单节点表支持SMP、存储过程内使用单节点表、支持TRIGGER。
  - d. 单节点表分布属性保持和复制表分布属性一致，单节点表的功能规格整体保持和复制表支持功能规格一致。
  - e. 通过DATABASE LINK对远端单节点表进行访问后（如：DML操作），会在系统表pg\_foreign\_table的ftoptions字段中标记is\_single\_datanode\_table=true的信息（仅单节点表存在该标识）。单节点表转换为其他表类型后，再次通过DML操作才会更新pg\_foreign\_table系统表中的ftoptions字段。
- 约束：
  - a. MERGE INTO USING子句中，目标表和源表均为单节点表时，需要为相同单节点组Node Group下的单节点表。
  - b. 单节点表支持INSERT ALL，但INSERT ALL的所有目标表必须为相同单节点组Node Group下的单节点表。
  - c. 支持通过DATABASE LINK对远端单节点表进行查询或DML操作。但是包含DATABASE LINK单节点表的SQL语句内，不支持包含DATABASE LINK非单节点表。
  - d. 当使用“/\*+ stream \*/ Hint”指定连接顺序时，若涉及分布式表与单节点表的混合连接，优化器可能忽略该Hint，主要由于优化器自动评估执行成本，若发现比Hint更优的方案，将优先采用替代计划。
  - e. 当级联收集单节点表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。
  - f. 单节点表仅支持基于单节点组Node Group功能使用。
  - g. 单节点表和非单节点表交互，不支持进行SMP操作。
  - h. 单节点表 and 不同DN的单节点表交互，不支持进行SMP操作。

## 使用示例

```
-- sys_single_group1、sys_single_group2为单节点组Node Group。
-- 创建单节点表。
gaussdb=# CREATE TABLE t_sig1(c1 int, c2 int, c3 int) TO GROUP sys_single_group1;
gaussdb=# CREATE TABLE t_sig2(c1 int primary key, c2 int, c3 int) TO GROUP sys_single_group2;
-- 单节点表导入数据。
gaussdb=# INSERT INTO t_sig1 VALUES(generate_series(1, 20), generate_series(21, 40), generate_series(1, 20));
```

### 场景：单节点表典型DML

```
-- 单节点表SELECT。
gaussdb=# SELECT t_sig1.* FROM t_sig1 WHERE t_sig1.c1 > 5 AND (t_sig1.*)::t_sig1 IS NOT NULL ORDER BY 1
LIMIT 5;
c1 | c2 | c3
----+----+----
6 | 26 | 6
7 | 27 | 7
8 | 28 | 8
9 | 29 | 9
10 | 30 | 10
(5 rows)
-- 单节点表SELECT FOR UPDATE | SHAR。
gaussdb=# SELECT t_sig1.* FROM t_sig1 WHERE t_sig1.c1 > 15 ORDER BY 1, 2 FOR UPDATE;
c1 | c2 | c3
----+----+----
16 | 36 | 16
17 | 37 | 17
18 | 38 | 18
19 | 39 | 19
20 | 40 | 20
(5 rows)
gaussdb=# SELECT t_sig1.* FROM t_sig1 WHERE t_sig1.c1 > 15 ORDER BY 1, 2 FOR SHARE;
c1 | c2 | c3
----+----+----
16 | 36 | 16
17 | 37 | 17
18 | 38 | 18
19 | 39 | 19
20 | 40 | 20
(5 rows)
-- 单节点表INSERT SELECT。
gaussdb=# INSERT INTO t_sig2 SELECT * FROM t_sig1;
-- 单节点表UPSERT。
gaussdb=# INSERT INTO t_sig1 VALUES (1, 21) ON duplicate key UPDATE c2 = c2 + 100;
-- 单节点表UPDATE。
gaussdb=# UPDATE t_sig1 SET c2 = (SELECT t_sig1.c1 FROM t_sig1 WHERE t_sig1.c2 = 20) WHERE c1 = 100;
-- 单节点表DELETE。
gaussdb=# DELETE FROM t_sig1 WHERE t_sig1.c1 = 100;
-- 单节点表MERGE。
gaussdb=# CREATE TABLE t_sig11(c1 int, c2 int, c3 int) TO GROUP sys_single_group1;
gaussdb=# INSERT INTO t_sig11 VALUES(100, 100, 100), (101, 101, 101);
gaussdb=# MERGE INTO t_sig1 AS mt1 USING t_sig11 AS mt2 ON mt1.c1 = mt2.c1 WHEN MATCHED THEN
UPDATE SET mt1.c2 = 100 WHEN NOT MATCHED THEN INSERT (c1, c2, c3) VALUES (100, 200, 300);
-- 删除单节点表。
gaussdb=# DROP TABLE t_sig1;
gaussdb=# DROP TABLE t_sig2;
gaussdb=# DROP TABLE t_sig11;
```

### 场景：单节点表和分布表交互

```
-- 创建表。
gaussdb=# CREATE TABLE t_rep(c1 int, c2 int) DISTRIBUTE BY replication; -- 复制表
gaussdb=# CREATE TABLE t_dis(c1 int, c2 int) DISTRIBUTE BY hash(c1); -- hash分布表
gaussdb=# CREATE TABLE t_sig1(c1 int, c2 int) DISTRIBUTE BY replication TO GROUP sys_single_group1; --
单节点表
gaussdb=# CREATE TABLE t_sig2(c1 int, c2 int) DISTRIBUTE BY replication TO GROUP sys_single_group2; --
单节点表
gaussdb=# CREATE TABLE t_sig11(c1 int, c2 int) DISTRIBUTE BY replication TO GROUP sys_single_group1;--
```

单节点表

```
gaussdb=# INSERT INTO t_rep SELECT v,v FROM generate_series(1,10) AS v;
gaussdb=# INSERT INTO t_sig1 VALUES (generate_series(11, 20), generate_series(11, 20));
gaussdb=# INSERT INTO t_dis VALUES (generate_series(11, 20), generate_series(11, 20));
gaussdb=# INSERT INTO t_sig2 VALUES (generate_series(11, 20), generate_series(11, 20));
-- UNION集合场景。
gaussdb=# SELECT * FROM t_sig1 UNION all SELECT * FROM t_sig2 ORDER BY 1;
 c1 | c2
-----+-----
 11 | 11
 11 | 11
 12 | 12
 12 | 12
 13 | 13
 13 | 13
 14 | 14
 14 | 14
 15 | 15
 15 | 15
 16 | 16
 16 | 16
 17 | 17
 17 | 17
 18 | 18
 18 | 18
 19 | 19
 19 | 19
 20 | 20
 20 | 20
(20 rows)
gaussdb=# SELECT * FROM t_sig1 UNION all SELECT * FROM t_rep ORDER BY 1 LIMIT 5;
 c1 | c2
-----+-----
 1 | 1
 2 | 2
 3 | 3
 4 | 4
 5 | 5
(5 rows)
gaussdb=# SELECT * FROM t_sig1 UNION ALL SELECT * FROM t_dis ORDER BY 1 LIMIT 5 OFFSET 2;
 c1 | c2
-----+-----
 12 | 12
 12 | 12
 13 | 13
 13 | 13
 14 | 14
(5 rows)
-- JOIN场景。
gaussdb=# SELECT * FROM t_sig1 LEFT JOIN t_dis ON t_sig1.c1 = t_dis.c1 ORDER BY 1,2 LIMIT 5;
 c1 | c2 | c1 | c2
-----+-----+-----+-----
 11 | 11 | 11 | 11
 12 | 12 | 12 | 12
 13 | 13 | 13 | 13
 14 | 14 | 14 | 14
 15 | 15 | 15 | 15
(5 rows)
gaussdb=# SELECT * FROM t_sig1 LEFT JOIN t_rep ON t_sig1.c1 = t_rep.c1 + 10 ORDER BY 1,2 LIMIT 5;
 c1 | c2 | c1 | c2
-----+-----+-----+-----
 11 | 11 | 1 | 1
 12 | 12 | 2 | 2
 13 | 13 | 3 | 3
 14 | 14 | 4 | 4
 15 | 15 | 5 | 5
(5 rows)
-- 相同单节点组Node Group上的单节点表下推场景。
-- SELECT 查询下推。
```



```
gaussdb=# EXPLAIN (costs off) SELECT * FROM t_sig1 WHERE c1 IN (SELECT c1 FROM t_sig11 WHERE c2 > 10 LIMIT 2);
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: datanode1
(2 rows)
-- INSERT SELECT下推。
gaussdb=# EXPLAIN (costs off) INSERT INTO t_sig1 SELECT * FROM t_sig11 WHERE c1 > 10;
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: (sys_single_group1) datanode1
(2 rows)
-- UPDATE下推。
gaussdb=# EXPLAIN (costs off) UPDATE t_sig1 SET (c1, c2) = (SELECT c1 + 10, c2 - c1 FROM t_sig11);
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: (sys_single_group1) datanode1
(2 rows)
-- DELETE下推。
gaussdb=# EXPLAIN (costs off) DELETE FROM t_sig1 WHERE c1 = (SELECT c1 FROM t_sig11 WHERE c2 > 10);
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: (sys_single_group1) datanode1
(2 rows)
-- JOIN下推。
gaussdb=# EXPLAIN (costs off) SELECT * FROM t_sig1, t_sig11 WHERE t_sig1.c1 = t_sig11.c2;
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: (sys_single_group1) datanode1
(2 rows)
-- UNION ALL下推。
gaussdb=# EXPLAIN (costs off) SELECT * FROM t_sig1 UNION ALL SELECT * FROM t_sig11 LIMIT 4;
QUERY PLAN

Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
Node/s: (GenGroup) datanode1
(2 rows)

-- 单节点表SMP场景。
--注意：需要在DEBUG版本下设置query_dop=1002强制生效SMP。
gaussdb=# SET query_dop = 1002;
gaussdb=# SET enable_fast_query_shipping = off;
-- 范围查询。
gaussdb=# EXPLAIN (costs off) SELECT * FROM t_sig1;
QUERY PLAN

Streaming (type: GATHER)
Node/s: (GenGroup) datanode1
-> Streaming(type: LOCAL GATHER dop: 1/2)
 Spawn ON: (sys_single_group1) datanode1
 -> Seq Scan ON t_sig1
(5 rows)
-- 聚集操作。
gaussdb=# EXPLAIN (costs off) SELECT count(*) FROM t_sig1;
QUERY PLAN

Streaming (type: GATHER)
Node/s: (GenGroup) datanode1
-> Aggregate
 -> Streaming(type: LOCAL GATHER dop: 1/2)
 Spawn ON: (sys_single_group1) datanode1
 -> Aggregate
 -> Seq Scan ON t_sig1
```

```
(7 rows)
-- 点查。
gaussdb=# EXPLAIN (costs off) SELECT avg(c1) as avg, c1 FROM t_sig1 WHERE c1 > 15 GROUP BY c1
HAVING c1 < 19 ORDER BY 1, 2;
 QUERY PLAN

Streaming (type: GATHER)
Node/s: (GenGroup) datanode1
-> Sort
 Sort Key: (pg_catalog.avg((avg(c1))))), c1
 -> Streaming(type: LOCAL GATHER dop: 1/2)
 Spawn ON: (sys_single_group1) datanode1
 -> HashAggregate
 GROUP BY Key: c1
 -> Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)
 Spawn ON: (sys_single_group1) datanode1
 -> HashAggregate
 GROUP BY Key: c1
 -> Seq Scan ON t_sig1
 Filter: ((c1 > 5) AND (c1 < 9))

(14 rows)
-- 删除表。
gaussdb=# DROP TABLE t_rep;
gaussdb=# DROP TABLE t_dis;
gaussdb=# DROP TABLE t_sig1;
gaussdb=# DROP TABLE t_sig2;
gaussdb=# DROP TABLE t_sig11;
```

## 7.37.4 常见问题

**问题现象：**执行关于单节点表DML语句中含有单节点读或写操作，发生报错。

**原因分析：**由于单节点表只在对应的sys\_single\_group的DN上存储数据，当表的scan算子或者modifytable算子下推到非数据存储的DN执行时，会发生报错拦截

**处理方法：**通过EXPLAIN查看执行计划是否合理，或者观察对应DN节点是否是单节点表的数据节点。

## 7.38 ROWID 表

### 7.38.1 特性概述

ROWID表是指包含ROWID/ROWNO系统列的表，ROWID不是普通的数据库列，而是一个伪列，表示表中每一行的唯一标识符。ROWID提供了对表中行的物理位置的直接访问，不需要在表中定义，每行数据自动拥有，不同表的ROWID可能重复，不支持通过INSERT/UPDATE/DELETE操作修改。本特性支持的ROWID表相关功能包括：

- ROWID表的创建、删除，包含CREATE TABLE AS、CREATE TABLE LIKE方式。
- 在ROWID表执行DML操作时，对ROWID/ROWNO系统列的维护。
- ROWID表的维护操作，包括VACUUM FULL、CLUSTER、ALTER TABLE SET TABLESPACE。
- 将非ROWID表转换为ROWID表，支持ALTER TABLE SET WITH ROWID。
- 对跨页透明压缩、闪回等其他使用到ROWID表的特性的支持。

## 说明

- ROWID列包含表的节点以及tableoid信息，当表的分布、分区有变动时，ROWID列可能发生变化：
  - 分布式信息改变：如UPDATE分布列，扩容缩容场景下重分布表，单节点表与分布表互相转换。
  - 分区信息改变：如UPDATE分区键导致跨分区，分区表进行DDL操作（EXCHANGE、MERGE、SPLIT、MOVE）。
- 触发器类型变量不支持访问ROWID/ROWNO系统列。
- 若范围表（子查询）无ROWID属性，则查询报错，比如Dual表。
- 不支持直连DN节点场景下使用ROWID系统列。
- 往ROWID表中插入元组时（包括COPY INTO，VACUUM FULL等间接插入元组的场景），需要读取ROWNO系统列的序列，由于分布式场景下序列需要从GTM中获取，会造成性能劣化，在高并发场景下，性能劣化更加明显。
- 对ROWID表进行细粒度备份恢复时，不保证新表中ROWID列的值与原表中ROWID列的值相同。

## 7.38.2 使用指导

### 7.38.2.1 DDL

ROWID表的创建和删除（ASTORE+非段页式+非分区表为例）：

```
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".
--步骤1: 创建ROWID表
ora=# CREATE TABLE t1 (c1 INT) WITH (storage_type = ASTORE, segment = OFF, hasrowid = ON);
CREATE TABLE
--步骤2: 观察到创建的ROWNO序列、ROWNO索引对象
ora=# \d
 List of relations
Schema | Name | Type | Owner | Storage
-----+-----+-----+-----+-----
public | t1 | table | Mike |
{orientation=row,storage_type=astore,segment=off,hasrowid=on,compression=no}
public | t1_seq | sequence | Mike |
(2 rows)
ora=# \di
 List of relations
Schema | Name | Type | Owner | Table | Storage
-----+-----+-----+-----+-----+-----
public | t1_rowno_idx | index | Mike | t1 |
(1 row)
--步骤3: 执行数据插入和ROWID查询
ora=# INSERT INTO t1 VALUES (1), (2), (3);
INSERT 0 3
ora=# SELECT rowid, rowno, * FROM t1;
 rowid | rowno | c1
-----+-----+-----
AAAAAAAAAP//AAAHvoAAAAAAAAAAB | 1 | 1
AAAAAAAAAP//AAAHvoAAAAAAAAAAC | 2 | 2
AAAAAAAAAP//AAAHvoAAAAAAAAAAD | 3 | 3
(3 rows)
ora=# SELECT rowid, rowno, * FROM t1 WHERE rowid = 'AAAAAAAAAP//AAAHvoAAAAAAAAAAC';
 rowid | rowno | c1
-----+-----+-----
AAAAAAAAAP//AAAHvoAAAAAAAAAAC | 2 | 2
(1 row)
```

```
--步骤4: 删除ROWID表
ora=# DROP TABLE t1;
DROP TABLE
--步骤5: 观察到ROWNO序列、ROWNO索引对象已被级联删除
ora=# \d
No relations found.
ora=# \di
No relations found.
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "$user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

非ROWID表转换为ROWID表（ASTORE+非段页式+非分区表为例）：

```
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
--步骤1: 创建非ROWID表
ora=# CREATE TABLE t1 (c1 INT) WITH (storage_type = ASTORE, segment = OFF);
CREATE TABLE
--步骤2: 观察到只有表对象
ora=# \d
 List of relations
Schema | Name | Type | Owner | Storage
-----+-----+-----+-----+-----
public | t1 | table | Mike | {orientation=row,storage_type=astore,segment=off,compression=no}
(1 row)

ora=# \di
No relations found.
--步骤3: 将非ROWID表转换为ROWID表
ora=# ALTER TABLE t1 SET WITH ROWID;
ALTER TABLE
--步骤4: 观察到创建的ROWNO序列、ROWNO索引对象
ora=# \d
 List of relations
Schema | Name | Type | Owner | Storage
-----+-----+-----+-----+-----
public | t1 | table | Mike | {orientation=row,storage_type=astore,segment=off,compression=no,hasrowid=on}
public | t1_seq | sequence | Mike |
(2 rows)

ora=# \di
 List of relations
Schema | Name | Type | Owner | Table | Storage
-----+-----+-----+-----+-----+-----
public | t1_rowno_idx | index | Mike | t1 |
(1 row)
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "$user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

DDL语法的详细介绍请参见《参考》中“SQL参考 > SQL语法”章节

### 7.38.2.2 ROWID 类型

ROWID包含定位表中某一行数据所需的关键信息，可用于快速定位指定数据行。

ROWID类型由BASE64编码的字符串组成，包含如下信息：

- 节点的分片编号，说明数据存储节点位置。
- hashbucket表的bucket编号。

- 表的oid信息。
- 行的编号，每一行数据的编号是唯一的。

ROWID类型的详细介绍请参见《参考》中“SQL参考 > 数据类型 > ROWID类型”章节。

### 7.38.2.3 DML

ROWID系统列支持在DML语句中使用，包括INSERT/SELECT/UPDATE/DELETE语句，示例如下：

```
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
ora=# CREATE TABLE test(a INT, b INT) WITH (hasrowid);
CREATE TABLE
ora=# -- INSERT
ora=# INSERT INTO test values(1, 1);
INSERT 0 1
ora=# -- ROWID表插入语句行记录会携带ROWID系统列
ora=# SELECT ROWID FROM test;
 rowid

AAAAAAAAAP//AAAEAAAAAAAAAAB
(1 row)
ora=# -- SELECT，使用ROWID系统列作为WHERE条件
ora=# SELECT * FROM test WHERE ROWID = 'AAAAAAAAAP//AAAEAAAAAAAAAAB';
 a | b
---+---
 1 | 1
(1 row)
ora=# -- UPDATE，使用ROWID系统列作为WHERE条件
ora=# UPDATE test SET b = 2 WHERE ROWID = 'AAAAAAAAAP//AAAEAAAAAAAAAAB';
UPDATE 1
ora=# -- DELETE，使用ROWID系统列作为WHERE条件
ora=# DELETE FROM test WHERE ROWID = 'AAAAAAAAAP//AAAEAAAAAAAAAAB';
DELETE 1
ora=# -- CLEAN
ora=# DROP TABLE test;
DROP TABLE
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

在DML语句中使用ROWID系统列时，若打开GUC参数opt\_behavior\_knob的ROWID\_OPT选项（详情请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 其他优化器选项”章节内容），则支持**ROWID系统列谓词条件改写**、**ROWID系统列ORDER BY语句改写**、**ROWID系统列max/min优化**、**ROWID系统列分布剪枝和分区剪枝**查询优化场景。

## ROWID 系统列谓词条件改写

当ROWID系统列用于WHERE子句或JOIN ON子句的谓词条件判断时，可进行条件语句的改写，当前仅支持比较操作符（=、<>、<、>、<=、>=）作为谓词条件的语句改写。

- 当操作符为等号（=）时，支持全场景改写，形如“ROWID = src\_expr”的谓词条件（当src\_expr也为ROWID系统列时，不进行改写，无实际意义），改写规则如下：

```
ROWID = src_expr AND (-1)::smallint = ROWID_HASHBUCKETID(src_expr) AND tableoid =
ROWID_TABLEOID(src_expr) AND rowno = ROWID_SEQUENCE(src_expr)
```

改写后，前三个条件分别触发分布剪枝、bucket剪枝和分区剪枝，最后一个条件触发索引扫描以提升扫描性能。对于不可改写的表达式，比如IN表达式“rowid IN (\$1,\$2,...)”，会进行全表扫描，遍历每行元组进行条件检查，此时不进行剪枝和索引扫描。

- 当操作符为非等号（<>、<、>、<=、>=）时，仅支持部分场景改写，比如范围表达式“ROWID > src\_expr”。由于ROWID的比较需综合考虑内部字段（node\_slice、bucket\_id、table\_oid、row\_no），改写仅适用于单节点普通表，且执行计划需为Light Proxy、FQS或PGXC，同时要求src\_expr为可计算常量。依次判断node\_id、table\_id和bucket\_id，当上述三者与给定rowid常量不同时，直接根据操作符进行比较，并将结果作为常量表达式返回；当它们与给定rowid常量均相同时，则将表达式改写为rowno系统列的谓词条件，即：

```
rowno > /< /< /> = /< = ROWID_SEQUENCE(src_expr)
```

对于无法改写的场景，仍执行全表扫描。

### 说明

ROWID系统列谓词条件改写遵从如下规格约束：

- WHERE/JOIN ON条件中，当操作符两侧都为ROWID系统列时不改写。
- 形如ROWID IN/NOT IN (xxx)场景，当右侧表达式为子查询表达式、自定义类型表达式、数组表达式时不支持改写。
- 形如NULLIF(ROWID,xxx)或NULLIF(ROWID,xxx)的NULLIF表达式不改写（ROWID = NULLIF(xxx,xxx)可以正常改写）。
- DISTINCT FROM表达式不改写。
- HAVING子句不改写。
- 操作符左右两侧均为RECORD类型表达式时不支持改写。
- ALL/ANY/SOME表达式不支持改写，且形如ROWID = ALL/ANY/SOME (xxx)或ALL/ANY/SOME (xxx) = ROWID的判断条件不支持改写。

上述约束同时适用于CN与DN节点，是否进行谓词改写取决于各节点实际处理的SQL形态。当原始SQL在CN上不满足改写条件时，CN不执行改写，但若下推至DN的SQL满足改写条件，仍可在DN节点进行谓词改写。

### 示例：

由于在不同场景下表oid字段值不同，以下示例仅作参考，执行时以实际输出结果为准。

以GUC参数explain\_perf\_mode设置为normal为例。

```
-- 数据准备
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".

ora=# CREATE TABLE t1(a int, c int) WITH (storage_type=astore, hasrowid=ON);
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.
CREATE TABLE
ora=# INSERT INTO t1 VALUES(1, 1);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(2, 2);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(3, 3);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(3, 4);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(1, 5);
INSERT 0 1
```

```

ora=# INSERT INTO t1 VALUES(2, 6);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(3, 7);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(3, 8);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(1, 9);
INSERT 0 1
ora=# INSERT INTO t1 VALUES(2, 10);
INSERT 0 1
ora=# CREATE OR REPLACE FUNCTION func RETURN rowid AS
ora$# temp rowid;
ora$# res int;
ora$# BEGIN
ora$# SELECT rowid INTO temp FROM t1 ORDER BY rowid LIMIT 1;
ora$# RETURN temp;
ora$# END;
ora$# /
CREATE FUNCTION
ora=# EXPLAIN (costs off, verbose) SELECT * FROM t1 WHERE rowid = func();
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.a, t1.c
 Node/s: All datanodes
 Remote query: SELECT a, c, rowid, tableoid, rowno FROM ONLY public.t1 WHERE true
 Coordinator quals: ((t1.rowid = func())) AND ((-1)::smallint = rowid_hashbucketid(func())) AND
(t1.tableoid = rowid_tableoid(func())) AND (t1.rowno = rowid_sequence(func()))
(5 rows)
ora=# SELECT * FROM t1 WHERE rowid = func();
 a | c
---+---
 1 | 1
(1 row)
ora=# --HAVING子句，不支持改写
ora=# EXPLAIN (costs off, verbose) SELECT max(a),max(rowid) FROM t1 GROUP BY rowid HAVING rowid =
'AAAAAAAAAP//BBBBBAAAAAAAAAAB';
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
QUERY PLAN

HashAggregate
 Output: max((max(a))), max((max(rowid))), rowid
 Group By Key: t1.rowid
 -> Streaming (TYPE: GATHER)
 Output: (max(a)), (max(rowid)), rowid
 Node/s: datanode1
 -> HashAggregate
 Output: max(a), max(rowid), rowid
 Group By Key: t1.rowid
 -> Seq Scan ON public.t1
 Output: rowid, a
 Distribute Key: a
 Filter: (t1.rowid = 'AAAAAAAAAP//BBBBBAAAAAAAAAAB'::rowid)
(13 rows)
ora=# SELECT max(a),max(rowid) FROM t1 GROUP BY rowid HAVING rowid = 'AAAAAAAAAP//
BBBBBAAAAAAAAAAB';
 max | max
-----+-----
(0 rows)
ora=# SELECT max(a) AS newcol1,rowid_sequence(max(rowid)) AS newcol2 FROM t1 GROUP BY rowid
ORDER BY newcol1,newcol2;
 newcol1 | newcol2
-----+-----
 1 | 1
 1 | 3

```

```
1 | 5
2 | 2
2 | 4
2 | 6
3 | 11
3 | 12
3 | 13
3 | 14
(10 rows)
ora=# -- NULLIF表达式本身不改写, rowid = NULLIF(xxx)改写
ora=# EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, rowid);
SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, rowid);
EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, 'AAAAAAAAAP//
AAAAAAAAAAAAAAAAAAAA');
SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, 'AAAAAAAAAP//
AAAAAAAAAAAAAAAAAAAA');
QUERY PLAN

Data Node Scan ON "__REMOTE_FQS_QUERY__"
Output: t1.rowid
Node/s: All datanodes
Remote query: SELECT rowid FROM public.t1 WHERE rowid = NULLIF(rowid, rowid) AND (-1)::smallint =
rowid_hashbucketid(NULLIF(rowid, rowid)) AND tableoid = rowid_tableoid(NULLIF(rowid, rowid)) AND
rowno
= rowid_sequence(NULLIF(rowid, rowid))
(4 rows)
ora=# SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, rowid);
rowid

(0 rows)
ora=# EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, 'AAAAAAAAAP//
AAAAAAAAAAAAAAAAAAAA');
QUERY PLAN

Data Node Scan ON "__REMOTE_FQS_QUERY__"
Output: t1.rowid
Node/s: All datanodes
Remote query: SELECT rowid FROM public.t1 WHERE rowid = NULLIF(rowid, 'AAAAAAAAAP//
AAAAAAAAAAAAAAAAAAAAA::rowid) AND (-1)::smallint = rowid_hashbucketid(NULLIF(rowid, 'AAAAAAAAAP//
AAAAAAAAAAAAAAAAAAAAA::row
id)) AND tableoid = rowid_tableoid(NULLIF(rowid, 'AAAAAAAAAP//AAAAAAAAAAAAAAAAAAAAA::rowid)) AND
rowno = rowid_sequence(NULLIF(rowid, 'AAAAAAAAAP//AAAAAAAAAAAAAAAAAAAAA::rowid))
(4 rows)
ora=# SELECT rowid FROM t1 WHERE rowid = NULLIF(rowid, 'AAAAAAAAAP//AAAAAAAAAAAAAAAAAAAA');
rowid

AAAAAAAAAP//AAAECPAAAAAAAAAB
AAAAAAAAAP//AAAECPAAAAAAAAAC
AAAAAAAAAP//AAAECPAAAAAAAAAD
AAAAAAAAAP//AAAECPAAAAAAAAAE
AAAAAAAAAP//AAAECPAAAAAAAAAF
AAAAAAAAAP//AAAECPAAAAAAAAAG
AAAAAAAABP//AAAECPAAAAAAAAAL
AAAAAAAABP//AAAECPAAAAAAAAAM
AAAAAAAABP//AAAECPAAAAAAAAAN
AAAAAAAABP//AAAECPAAAAAAAAAO
```



```
(10 rows)
ora=# -- rowexpr不改写
ora=# EXPLAIN (costs off, verbose) SELECT a FROM t1 WHERE (rowid,c) IN (SELECT rowid,c FROM t1);
WARNING: Statistics IN some tables OR columns(public.t1.a, public.t1.c) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Streaming (TYPE: GATHER)
 Output: public.t1.a
 Node/s: All datanodes
 -> Hash Right Semi Join
 Output: public.t1.a
 Hash Cond: ((public.t1.rowid = public.t1.rowid) AND (public.t1.c = public.t1.c))
 -> Streaming(TYPE: BROADCAST)
 Output: public.t1.rowid, public.t1.c
 Spawn ON: All datanodes
 Consumer Nodes: All datanodes
 -> Seq Scan ON public.t1
 Output: public.t1.rowid, public.t1.c
 Distribute Key: public.t1.a
 -> Hash
 Output: public.t1.a, public.t1.rowid, public.t1.c
 -> Seq Scan ON public.t1
 Output: public.t1.a, public.t1.rowid, public.t1.c
 Distribute Key: public.t1.a

(18 rows)
ora=# SELECT a FROM t1 WHERE (rowid,c) IN (SELECT rowid,c FROM t1) ORDER BY a;
a

1
1
1
2
2
2
3
3
3
3
(10 rows)
ora=# CREATE TYPE t1_type AS (rowid rowid, c int);
CREATE TYPE
ora=# EXPLAIN (costs off, verbose) SELECT a FROM t1 WHERE (rowid, c) IN t1_type(func(), 1);
WARNING: Statistics IN some tables OR columns(public.t1.a, public.t1.c) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.a
 Node/s: All datanodes
 Remote query: SELECT a, rowid FROM ONLY public.t1 WHERE c = 1
 Coordinator quals: (t1.rowid = func())

(5 rows)
ora=# SELECT a FROM t1 WHERE (rowid, c) IN t1_type(func(), 1) ORDER BY a;
a

1
(1 row)
ora=# -- IN/not IN
ora=# EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid IN func();
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.rowid
 Node/s: All datanodes
 Remote query: SELECT rowid, tableoid, rowno FROM ONLY public.t1 WHERE true
```

```
Coordinator quals: ((t1.rowid = func())) AND ((-1)::smallint = rowid_hashbucketid(func())) AND
(t1.tableoid = rowid_tableoid(func())) AND (t1.rowno = rowid_sequence(func()))
(5 rows)
ora=# SELECT rowid FROM t1 WHERE rowid IN func();
 rowid

AAAAAAAAAP//AAAECPAAAAAAAAAAB
(1 row)
ora=# EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid IN (func(), func()); --rowexpr不改
写
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
Output: t1.rowid
Node/s: All datanodes
Remote query: SELECT rowid FROM ONLY public.t1 WHERE true
Coordinator quals: (t1.rowid = ANY (ARRAY[func(), func()])))
(5 rows)
ora=# SELECT rowid FROM t1 WHERE rowid IN (func(), func());
 rowid

AAAAAAAAAP//AAAECPAAAAAAAAAAB
(1 row)
ora=# EXPLAIN (costs off, verbose) SELECT rowid FROM t1 WHERE rowid not IN func();
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
Output: t1.rowid
Node/s: All datanodes
Remote query: SELECT rowid FROM ONLY public.t1 WHERE true
Coordinator quals: (t1.rowid <> func())
(5 rows)
ora=# SELECT rowid FROM t1 WHERE rowid not IN func();
 rowid

AAAAAAAAAP//AAAECPAAAAAAAAAAC
AAAAAAAAAP//AAAECPAAAAAAAAAAD
AAAAAAAAAP//AAAECPAAAAAAAAAAE
AAAAAAAAAP//AAAECPAAAAAAAAAAF
AAAAAAAAAP//AAAECPAAAAAAAAAAG
AAAAAAAAABP//AAAECPAAAAAAAAAAL
AAAAAAAAABP//AAAECPAAAAAAAAAAM
AAAAAAAAABP//AAAECPAAAAAAAAAAN
AAAAAAAAABP//AAAECPAAAAAAAAAAO
(9 rows)
ora=# -- distinct FROM不改写
ora=# EXPLAIN (costs off, verbose) SELECT a FROM t1 WHERE rowid is distinct FROM func();
WARNING: Statistics IN some tables OR columns(public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
 QUERY PLAN

Data Node Scan ON t1 "_REMOTE_TABLE_QUERY_"
Output: t1.a
Node/s: All datanodes
Remote query: SELECT a, rowid FROM ONLY public.t1 WHERE true
Coordinator quals: (t1.rowid IS DISTINCT FROM func())
(5 rows)
ora=# SELECT a FROM t1 WHERE rowid is distinct FROM func() ORDER BY a;
 a
--
1
1
2
2
2
```

```

3
3
3
(9 rows)
-- 在JOIN ON子句的谓词条件改写
ora=# CREATE TABLE t2(a int, c text) WITH (storage_type=astore, hasrowid=on);
CREATE TABLE
ora=#
ora=# INSERT INTO t2 VALUES(1, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(2, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(3, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(3, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(1, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(2, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(3, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(3, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(1, func());
INSERT 0 1
ora=# INSERT INTO t2 VALUES(2, func());
INSERT 0 1
ora=# EXPLAIN (costs off, verbose) SELECT *,t1.rowid, t2.rowid FROM t1 INNER JOIN t2 ON t1.rowid = t2.c;
WARNING: Statistics IN some tables OR columns(public.t1.a, public.t2.a, public.t2.c) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.

QUERY PLAN

Streaming (TYPE: GATHER)
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Node/s: All datanodes
 -> Nested Loop
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Inner Unique: true
 -> Streaming(TYPE: BROADCAST)
 Output: t2.a, t2.c, t2.rowid
 Spawn ON: All datanodes
 Consumer Nodes: All datanodes
 -> Seq Scan ON public.t2
 Output: t2.a, t2.c, t2.rowid
 Distribute Key: t2.a
 Filter: (((-1)::smallint = rowid_hashbucketid((t2.c)::rowid))
 -> Index Scan using t1_rowno_idx ON public.t1
 Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
 Distribute Key: t1.a
 Index Cond: (t1.rowno = rowid_sequence((t2.c)::rowid))
 Filter: (((-1)::smallint = rowid_hashbucketid(t1.rowid)) AND ((t2.c)::rowid = t1.rowid) AND
(rowid_tableoid((t2.c)::rowid) = t1.tableoid))
(19 rows)
ora=# SELECT *,t1.rowid, t2.rowid FROM t1 INNER JOIN t2 ON t1.rowid = t2.c;
a | c | a | c | rowid | rowid
-----+-----+-----+-----+-----+-----
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAABP//
AAAECAAAAAAAAAAAL
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAABP//
AAAECAAAAAAAAAAAM
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAABP//
AAAECAAAAAAAAAAN
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAABP//
AAAECAAAAAAAAAAO
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAP//AAAECAAAAAAAAAAAB
AAAAAAAAAP//AAAECAAAAAAAAAAAB

```

```
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
(10 rows)
ora=#
ora=# EXPLAIN (costs off, verbose) SELECT *,t1.rowid, t2.rowid FROM t1 LEFT JOIN t2 ON t1.rowid = t2.c;
WARNING: Statistics IN some tables OR columns(public.t1.a, public.t2.a, public.t2.c) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
QUERY PLAN

Streaming (TYPE: GATHER)
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Node/s: All datanodes
 -> Nested Loop Left Join
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Join Filter: ((t1.rowid = (t2.c)::rowid) AND (t1.tableoid = rowid_tableoid((t2.c)::rowid)) AND (t1.rowno
= rowid_sequence((t2.c)::rowid)))
 -> Index Scan using t1_rowno_idx ON public.t1
 Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
 Distribute Key: t1.a
 -> Materialize
 Output: t2.a, t2.c, t2.rowid
 -> Streaming(TYPE: BROADCAST)
 Output: t2.a, t2.c, t2.rowid
 Spawn ON: All datanodes
 Consumer Nodes: All datanodes
 -> Seq Scan ON public.t2
 Output: t2.a, t2.c, t2.rowid
 Distribute Key: t2.a
 Filter: ((-1)::smallint = rowid_hashbucketid((t2.c)::rowid))
(19 rows)
ora=# SELECT *,t1.rowid, t2.rowid FROM t1 LEFT JOIN t2 ON t1.rowid = t2.c;
a | c | a | c | rowid | rowid |
-----+-----+-----+-----+-----+-----+
3 | 3 | | | AAAAAAAAABP//AAAECPAAAAAAAAAAL |
3 | 4 | | | AAAAAAAAABP//AAAECPAAAAAAAAAAM |
3 | 7 | | | AAAAAAAAABP//AAAECPAAAAAAAAAAN |
3 | 8 | | | AAAAAAAAABP//AAAECPAAAAAAAAAAO |
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 1 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 3 | AAAAAAAAAAP//AAAECPAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAB |
AAAAAAAAAAP//AAAECPAAAAAAAAAAB
2 | 2 | | | AAAAAAAAAAP//AAAECPAAAAAAAAAAC |
1 | 5 | | | AAAAAAAAAAP//AAAECPAAAAAAAAAAD |
2 | 6 | | | AAAAAAAAAAP//AAAECPAAAAAAAAAAE |
```

```
1 | 9 | | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAF |
2 | 10 | | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAG |
(19 rows)
ora=# EXPLAIN (costs off, verbose) SELECT *,t1.rowid, t2.rowid FROM t1 RIGHT JOIN t2 ON t1.rowid = t2.c;
WARNING: Statistics IN some tables OR columns(public.t2.a, public.t2.c, public.t1.a) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
QUERY PLAN

Streaming (TYPE: GATHER)
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Node/s: All datanodes
 -> Hash Right Join
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Hash Cond: ((t1.rowid = (t2.c)::rowid) AND (t1.tableoid = rowid_tableoid((t2.c)::rowid)) AND
(t1.rowno = rowid_sequence((t2.c)::rowid)))
 Join Filter: ((-1)::smallint = rowid_hashbucketid((t2.c)::rowid))
 -> Streaming(TYPE: BROADCAST)
 Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
 Spawn ON: All datanodes
 Consumer Nodes: All datanodes
 -> Index Scan using t1_rowno_idx ON public.t1
 Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
 Distribute Key: t1.a
 -> Hash
 Output: t2.a, t2.c, t2.rowid
 -> Seq Scan ON public.t2
 Output: t2.a, t2.c, t2.rowid
 Distribute Key: t2.a
(19 rows)
ora=# SELECT *,t1.rowid, t2.rowid FROM t1 RIGHT JOIN t2 ON t1.rowid = t2.c;
a | c | a | | rowid | | rowid
-----+-----+-----+-----+-----+-----
1 | 1 | 2 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAAG
1 | 1 | 1 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAAF
1 | 1 | 2 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAAE
1 | 1 | 1 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAD
1 | 1 | 2 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAC
1 | 1 | 1 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB |
AAAAAAAAAP//AAAECPAAAAAAAAAAB
1 | 1 | 3 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAABP//
AAAECPAAAAAAAAAAAO
1 | 1 | 3 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAABP//
AAAECPAAAAAAAAAAN
1 | 1 | 3 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAABP//
AAAECPAAAAAAAAAAM
1 | 1 | 3 | | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAAAAAP//AAAECPAAAAAAAAAAAB | AAAAAAABP//
AAAECPAAAAAAAAAAL
(10 rows)
ora=#
ora=# EXPLAIN (costs off, verbose) SELECT *,t1.rowid, t2.rowid FROM t1 FULL JOIN t2 ON t1.rowid = t2.c;
WARNING: Statistics IN some tables OR columns(public.t1.a, public.t2.a, public.t2.c) are not collected.
HINT: Do analyze for them IN ORDER to generate optimized plan.
QUERY PLAN

Streaming (TYPE: GATHER)
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Node/s: All datanodes
 -> Hash Full Join
 Output: t1.a, t1.c, t2.a, t2.c, t1.rowid, t2.rowid
 Inner Unique: true
 Hash Cond: (((t2.c)::rowid = t1.rowid) AND (rowid_tableoid((t2.c)::rowid) = t1.tableoid) AND
(rowid_sequence((t2.c)::rowid) = t1.rowno))
```

```
Join Filter: ((-1)::smallint = rowid_hashbucketid((t2.c)::rowid))
-> Streaming(TYPE: REDISTRIBUTE)
Output: t2.a, t2.c, t2.rowid, (rowid_sequence((t2.c)::rowid))
Distribute Key: (rowid_sequence((t2.c)::rowid))
Spawn ON: All datanodes
Consumer Nodes: All datanodes
-> Seq Scan ON public.t2
Output: t2.a, t2.c, t2.rowid, rowid_sequence((t2.c)::rowid)
Distribute Key: t2.a
-> Hash
Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
-> Streaming(TYPE: REDISTRIBUTE)
Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
Distribute Key: t1.rowno
Spawn ON: All datanodes
Consumer Nodes: All datanodes
-> Index Scan using t1_rowno_idx ON public.t1
Output: t1.a, t1.c, t1.rowid, t1.tableoid, t1.rowno
Distribute Key: t1.a

(26 rows)
ora=# SELECT *,t1.rowid, t2.rowid FROM t1 FULL JOIN t2 ON t1.rowid = t2.c;
a | c | a | c | rowid | rowid |
-----+-----+-----+-----+-----+-----+
1 | 1 | 1 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAB
1 | 1 | 2 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAC
1 | 1 | 1 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAD
1 | 1 | 2 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAE
1 | 1 | 1 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAF
1 | 1 | 2 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAAAP//AAECzAAAAAAAAAAG
1 | 1 | 3 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAABP//AAECzAAAAAAAAAAL
1 | 1 | 3 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAABP//AAECzAAAAAAAAAAM
1 | 1 | 3 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAABP//AAECzAAAAAAAAAAN
1 | 1 | 3 | AAAAAAAAAAP//AAECpAAAAAAAAAAB | AAAAAAAAAAP//AAECpAAAAAAAAAAB |
AAAAAAAAABP//AAECzAAAAAAAAAAO
1 | 9 | | AAAAAAAAAAP//AAECpAAAAAAAAAAF |
2 | 2 | | AAAAAAAAAAP//AAECpAAAAAAAAAAC |
3 | 7 | | AAAAAAAAABP//AAECpAAAAAAAAAAN |
3 | 4 | | AAAAAAAAABP//AAECpAAAAAAAAAAM |
2 | 10 | | AAAAAAAAAAP//AAECpAAAAAAAAAAG |
2 | 6 | | AAAAAAAAAAP//AAECpAAAAAAAAAAE |
3 | 3 | | AAAAAAAAABP//AAECpAAAAAAAAAAL |
3 | 8 | | AAAAAAAAABP//AAECpAAAAAAAAAAO |
1 | 5 | | AAAAAAAAAAP//AAECpAAAAAAAAAAD |

(19 rows)

ora=# -- 单节点表
ora=# CREATE TABLE t1_single(a int, c int) WITH (storage_type=astore, hasrowid=on) to group
sys_single_group1;
CREATE TABLE
ora=# INSERT INTO t1_single VALUES(1, 1);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(2, 2);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(3, 3);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(3, 4);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(1, 5);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(2, 6);
```

```
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(3, 7);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(3, 8);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(1, 9);
INSERT 0 1
ora=# INSERT INTO t1_single VALUES(2, 10);
INSERT 0 1
-- WHERE子句改写
ora=# EXPLAIN (costs off, verbose) SELECT * FROM t1_single WHERE rowid = func();
WARNING: Statistics in some tables or columns(public.t1_single) are not collected.
HINT: Do analyze for them in order to generate optimized plan.
 QUERY PLAN

Data Node Scan on t1_single "_REMOTE_TABLE_QUERY_"
Output: t1_single.a, t1_single.c
Node/s: (sys_single_group1) datanode1
Remote query: SELECT a, c, rowid, tableoid, rowno FROM ONLY public.t1_single WHERE true
Coordinator quals: ((t1_single.rowid = func())) AND ((-1)::smallint = rowid_hashbucketid(func())) AND
(t1_single.tableoid = rowid_tableoid(func())) AND (t1_single.rowno = rowid_sequence(func()))
(5 rows)

-- 单节点表非等值场景改写
ora=# SET max_datanode_for_plan = 1;
SET
ora=# EXPLAIN verbose SELECT * FROM t1_single WHERE rowid > 'AAAAAAAAAP//AAAI/TAAAAAAAAAAB';
 QUERY PLAN

Data Node Scan on "__REMOTE_LIGHT_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Output: t1_single.a, t1_single.c
Node/s: datanode1
Remote query: SELECT a, c FROM public.t1_single WHERE rowid > 'AAAAAAAAAP//AAAI/
TAAAAAAAAAAB'::rowid

Remote SQL: SELECT a, c FROM public.t1_single WHERE rowid > 'AAAAAAAAAP//AAAI/
TAAAAAAAAAAB'::rowid
Datanode Name: datanode1
[Bypass]
Index Scan using t1_single_rowno_idx on public.t1_single (cost=0.00..24.88 rows=716 width=8)
Output: a, c
Index Cond: (t1_single.rowno > 1::bigint)

(12 rows)

-- 清理数据
ora=# DROP TABLE t1;
DROP TABLE
ora=# DROP TABLE t2;
DROP TABLE
ora=# DROP TABLE t1_single;
DROP TABLE
ora=# DROP FUNCTION func;
DROP FUNCTION
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "$user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

## ROWID 系统列 ORDER BY 语句改写

对于ORDER BY rowid的场景，分布式下发到DN节点的SQL可改写为ORDER BY tableoid, rowno，如果为普通表、单节点表或者剪枝到单分区的场景，可进一步简化

为ORDER BY rowno。rowid系统列排序规则是按照其包含的字段node\_slice、table\_oid、bucket\_id、row\_no依次比较排序。但rowid系统列本身并未建立索引，而rowno字段建立了唯一索引，因此对于普通表、单节点表或者剪枝到单分区的场景，ORDER BY rowid实际等效于ORDER BY rowno系统列，而rowno系统列上具有索引，使用上更高效，在两者排序等价场景下将ORDER BY rowid改写为ORDER BY rowno。

### 说明

ROWID系统列ORDER BY语句改写，若SQL未下推至DN节点，不支持ORDER BY rowid改写；若SQL下推至DN节点执行时，遵从如下规格约束：

- 普通表和单节点表ORDER BY rowid单列场景下，改写为ORDER BY rowno系统列。
- 分区表和单节点分区表ORDER BY rowid单列场景下，改写为ORDER BY tableoid, rowno，若剪枝到单分区，则进一步改写为ORDER BY rowno。
- 普通表、单节点表或者分区表剪枝到单分区场景下，ORDER BY多列且其中一列为tableoid系统列时，支持去除冗余的ORDER BY tableoid（详情见下文介绍）。
- ORDER BY多列时，不支持ORDER BY rowid改写。
- 语句包含GROUP、DISTINCT时，不支持ORDER BY rowid改写。
- 聚集函数中的ORDER BY rowid不支持改写。
- 仅基表为物理表的支持改写，基表为子查询或CTE等不支持改写。

除ORDER BY rowid改写场景外，还支持对冗余tableoid系统列条件进行消除（包括用户添加的tableoid判断条件），满足以下条件时支持去除tableoid排序：

- 排序键大于或等于2个。
- 若SQL未下推至DN节点，则仅支持普通复制表、单节点表，以及可以裁剪到单分区的分区复制表和单节点分区表；若SQL下推至DN，则支持普通表、单节点表，或者分区表剪枝到单分区。

示例：

由于不同场景下tableoid字段值不同，以下示例仅作参考，执行时以实际输出结果为准。

以GUC参数explain\_perf\_mode设置为normal为例。

```
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".
-- 普通表场景
ora=# SET max_datanode_for_plan = 64;
SET
ora=# SET cursor_sharing = force;
SET
ora=# SET dist_query_adaptive_mode=off;
SET
ora=# CREATE TABLE t2 (
ora(# c1 INTEGER,
ora(# c2 INTEGER,
ora(# c3 INTEGER
ora(#) WITH (storage_type = astore, hasrowid);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' AS the distribution column BY default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
ora=# INSERT INTO t2 VALUES(generate_series(1,100),generate_series(1,100),generate_series(1,300));
INSERT 0 300
ora=# ANALYZE t2;
ANALYZE
ora=# SET enable_stream_operator=off;
```



```

SET
-- light proxy
ora=# SET enable_fast_query_shipping = on;
SET
ora=# SET enable_light_proxy = on;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ * FROM t2 WHERE c1 = 100 ORDER
BY rowid;

 QUERY PLAN

[Parameterized]
Data Node Scan on "__REMOTE_LIGHT_QUERY__"
Output: t2.c1, t2.c2, t2.c3
Node expr: $1
Remote query: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t2 WHERE c1 = $1 ORDER BY rowid

Remote SQL: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t2 WHERE c1 = 100 ORDER BY rowid
Datanode Name: datanode2
[Parameterized]
Index Scan using t2_rowno_idx on public.t2
Output: c1, c2, c3, rowid, tableoid, rowno
Filter: (t2.c1 = $1)
(13 rows)
ora=# SELECT /*+ indexscan(t2) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2 WHERE c1 =
100 ORDER BY rowid LIMIT 1;
rowid_nodeid | rowid_sequence | c3
-----+-----+-----
1 | 108 | 100
(1 row)
-- FQS
ora=# SET enable_fast_query_shipping = on;
SET
ora=# SET enable_light_proxy = off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ * FROM t2 WHERE c1 = 100 ORDER
BY rowid;

 QUERY PLAN

[Parameterized]
Data Node Scan on "__REMOTE_FQS_QUERY__"
Output: t2.c1, t2.c2, t2.c3
Node expr: $1
Remote query: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t2 WHERE c1 = $1 ORDER BY rowid

Remote SQL: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t2 WHERE c1 = 100 ORDER BY rowid
Datanode Name: datanode2
[Parameterized]
Index Scan using t2_rowno_idx on public.t2
Output: c1, c2, c3, rowid, tableoid, rowno
Filter: (t2.c1 = $1)
(13 rows)
ora=# SELECT /*+ indexscan(t2) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2 WHERE c1 =
100 ORDER BY rowid LIMIT 1;
rowid_nodeid | rowid_sequence | c3
-----+-----+-----
1 | 108 | 100
(1 row)
-- stream
ora=# RESET enable_stream_operator;
RESET
ora=# SET enable_light_proxy = off;
SET
ora=# SET enable_fast_query_shipping = off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ * FROM t2 ORDER BY rowid;
WARNING: unused hint: IndexScan(t2)
 QUERY PLAN

```

```

[Parameterized]
Streaming (type: GATHER)
Output: c1, c2, c3, rowid
Merge Sort Key: t2.rowid
Node/s: All datanodes
-> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2.rowid
 -> Seq Scan on public.t2
 Output: c1, c2, c3, rowid
 Distribute Key: c1
(11 rows)
ora=# SELECT /*+ indexscan(t2) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2 ORDER BY
rowid LIMIT 1;
rowid_nodeid | rowid_sequence | c3
-----+-----+-----
0 | 1 | 1
(1 row)
ora=#
ora=# RESET enable_light_proxy;
RESET
ora=# RESET enable_stream_operator;
RESET
ora=# RESET enable_fast_query_shipping;
RESET

-- 分区表场景
ora=# CREATE TABLE t2_p (
ora(# c1 INTEGER,
ora(# c2 INTEGER,
ora(# c3 INTEGER
ora(#) WITH (storage_type = astore, hasrowid)
ora=# PARTITION BY RANGE (c2)
ora-# (
ora(# PARTITION p2023q1 VALUES LESS THAN (100),
ora(# PARTITION p2023q2 VALUES LESS THAN (200),
ora(# PARTITION p2023q3 VALUES LESS THAN (MAXVALUE)
ora(#);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' AS the distribution column BY default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
ora=# INSERT INTO t2_p VALUES(generate_series(1,200),generate_series(1,300),generate_series(1,600));
INSERT 0 600
ora=# ANALYZE t2_p;
ANALYZE
ora=#
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p ORDER BY rowid;
WARNING: unused hint: IndexScan(t2_p)
 QUERY PLAN

Streaming (type: GATHER)
Output: c1, c2, c3, rowid
Merge Sort Key: t2_p.rowid
Node/s: All datanodes
-> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2_p.rowid
 -> Partition Iterator
 Output: c1, c2, c3, rowid
 Iterations: 3
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid
 Distribute Key: c1
 Selected Partitions: 1..3
(14 rows)
ora=# SET enable_stream_operator=off;
SET

```

```
-- light proxy
ora=# SET enable_fast_query_shipping = on;
SET
ora=# SET enable_light_proxy = on;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c1=100
ORDER BY rowid;

QUERY PLAN

[Parameterized]
Data Node Scan on "__REMOTE_LIGHT_QUERY__"
 Output: t2_p.c1, t2_p.c2, t2_p.c3
 Node expr: $1
 Remote query: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3 FROM public.t2_p WHERE c1 = $1 ORDER BY rowid

Remote SQL: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3 FROM public.t2_p WHERE c1 = 100 ORDER BY rowid
Datanode Name: datanode2
[Parameterized]
Sort
 Output: c1, c2, c3, rowid, tableoid, rowno
 Sort Key: t2_p.tableoid, t2_p.rowno
 -> Partition Iterator
 Output: c1, c2, c3, rowid, tableoid, rowno
 Iterations: 3
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid, tableoid, rowno
 Filter: (t2_p.c1 = $1)
 Selected Partitions: 1..3

(20 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p WHERE
c1=100 ORDER BY rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----
 1 | 108 | 100
(1 row)
-- FQS
ora=# SET enable_fast_query_shipping = on;
SET
ora=# SET enable_light_proxy = off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p ORDER BY rowid;--
pgxc

QUERY PLAN

[Parameterized]
Sort
 Output: t2_p.c1, t2_p.c2, t2_p.c3, t2_p.rowid
 Sort Key: t2_p.rowid
 -> Data Node Scan on "__REMOTE_SORT_QUERY__"
 Output: t2_p.c1, t2_p.c2, t2_p.c3, t2_p.rowid
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3, rowid FROM ONLY public.t2_p WHERE true
ORDER BY 4

Remote SQL: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3, rowid FROM ONLY public.t2_p WHERE true ORDER
BY 4
Datanode Name: datanode1
[Parameterized]
Sort
 Output: c1, c2, c3, rowid, tableoid, rowno
 Sort Key: t2_p.tableoid, t2_p.rowno
 -> Result
 Output: c1, c2, c3, rowid, tableoid, rowno
```

```

One-Time Filter: $1
-> Partition Iterator
 Output: c1, c2, c3, rowid, tableoid, rowno
 Iterations: 3
-> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid, tableoid, rowno
 Selected Partitions: 1..3

Datanode Name: datanode2
[Parameterized]
Sort
 Output: c1, c2, c3, rowid, tableoid, rowno
 Sort Key: t2_p.tableoid, t2_p.rowno
-> Result
 Output: c1, c2, c3, rowid, tableoid, rowno
 One-Time Filter: $1
 -> Partition Iterator
 Output: c1, c2, c3, rowid, tableoid, rowno
 Iterations: 3
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid, tableoid, rowno
 Selected Partitions: 1..3

(40 rows)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c1=100
ORDER BY rowid;

 QUERY PLAN

[Parameterized]
Data Node Scan on "__REMOTE_FQS_QUERY__"
 Output: t2_p.c1, t2_p.c2, t2_p.c3
 Node expr: $1
 Remote query: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3 FROM public.t2_p WHERE c1 = $1 ORDER BY rowid

Remote SQL: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3 FROM public.t2_p WHERE c1 = 100 ORDER BY rowid
Datanode Name: datanode2
[Parameterized]
Sort
 Output: c1, c2, c3, rowid, tableoid, rowno
 Sort Key: t2_p.tableoid, t2_p.rowno
-> Partition Iterator
 Output: c1, c2, c3, rowid, tableoid, rowno
 Iterations: 3
-> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid, tableoid, rowno
 Filter: (t2_p.c1 = $1)
 Selected Partitions: 1..3

(20 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p WHERE
c1=100 ORDER BY rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----
 1 | 108 | 100
(1 row)
-- stream
ora=# RESET enable_stream_operator;
RESET
ora=# SET enable_light_proxy = off;
SET
ora=# SET enable_fast_query_shipping = off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p ORDER BY rowid;
WARNING: unused hint: IndexScan(t2_p)

 QUERY PLAN

[Parameterized]
Streaming (type: GATHER)
 Output: c1, c2, c3, rowid

```

```

Merge Sort Key: t2_p.rowid
Node/s: All datanodes
-> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2_p.rowid
-> Partition Iterator
 Output: c1, c2, c3, rowid
 Iterations: 3
-> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid
 Distribute Key: c1
 Selected Partitions: 1..3

(15 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p ORDER BY
rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----
 0 | 1 | 1
(1 row)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c1=100
ORDER BY rowid;
WARNING: unused hint: IndexScan(t2_p)
 QUERY PLAN

[Parameterized]
Streaming (type: GATHER)
 Output: c1, c2, c3, rowid
 Node/s: datanode2
 -> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2_p.rowid
 -> Partition Iterator
 Output: c1, c2, c3, rowid
 Iterations: 3
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid
 Distribute Key: c1
 Filter: (t2_p.c1 = 100)
 Selected Partitions: 1..3

(15 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p WHERE
c1=100 ORDER BY rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----
 1 | 108 | 100
(1 row)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c2=100
ORDER BY rowid;
WARNING: unused hint: IndexScan(t2_p)
 QUERY PLAN

[Parameterized]
Streaming (type: GATHER)
 Output: c1, c2, c3, rowid
 Merge Sort Key: t2_p.rowid
 Node/s: All datanodes
 -> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2_p.rowid
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid
 Distribute Key: c1
 Filter: (t2_p.c2 = 100)
 Selected Partitions: 2

(13 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p WHERE
c2=100 ORDER BY rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----

```

```

0 | 402 | 400
(1 row)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c1=100 and
c2=100 ORDER BY rowid;
WARNING: unused hint: IndexScan(t2_p)
 QUERY PLAN

[Parameterized]
Streaming (type: GATHER)
 Output: c1, c2, c3, rowid
 Node/s: datanode2
 -> Sort
 Output: c1, c2, c3, rowid
 Sort Key: t2_p.rowid
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, rowid
 Distribute Key: c1
 Filter: ((t2_p.c1 = 100) AND (t2_p.c2 = 100))
 Selected Partitions: 2
(12 rows)
ora=# SELECT /*+ indexscan(t2_p) */ rowid_nodeid(rowid),rowid_sequence(rowid),c3 FROM t2_p WHERE
c1=100 and c2=100 ORDER BY rowid LIMIT 1;
 rowid_nodeid | rowid_sequence | c3
-----+-----+-----
1 | 108 | 100
(1 row)
ora=# RESET enable_light_proxy;
RESET
ora=# RESET enable_stream_operator;
RESET
ora=# RESET enable_fast_query_shipping;
RESET
-- 冗余table_oid排序去除
ora=# SET enable_stream_operator=off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ rowid,rowno FROM t2 ORDER BY
tableoid,c1;
 QUERY PLAN

[Parameterized]
Sort
 Output: t2.rowid, t2.rowno, t2.tableoid, t2.c1
 Sort Key: t2.tableoid, t2.c1
 -> Data Node Scan on "__REMOTE_SORT_QUERY__"
 Output: t2.rowid, t2.rowno, t2.tableoid, t2.c1
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t2)*/ rowid, rowno, tableoid, c1 FROM ONLY public.t2 WHERE
true ORDER BY 3, 4

Remote SQL: SELECT/*+ IndexScan(t2)*/ rowid, rowno, tableoid, c1 FROM ONLY public.t2 WHERE true
ORDER BY 3, 4
Datanode Name: datanode1
[Parameterized]
Sort
 Output: rowid, rowno, tableoid, c1
 Sort Key: t2.c1
 -> Result
 Output: rowid, rowno, tableoid, c1
 One-Time Filter: $1
 -> Seq Scan on public.t2
 Output: rowid, rowno, tableoid, c1

Datanode Name: datanode2
[Parameterized]
Sort
 Output: rowid, rowno, tableoid, c1
 Sort Key: t2.c1

```

```
-> Result
 Output: rowid, rowno, tableoid, c1
 One-Time Filter: $1
 -> Seq Scan on public.t2
 Output: rowid, rowno, tableoid, c1
(32 rows)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2_p) */ * FROM t2_p WHERE c2 = 90
ORDER BY tableoid,c1;
 QUERY PLAN

[Parameterized]
Sort
 Output: t2_p.c1, t2_p.c2, t2_p.c3, t2_p.tableoid
 Sort Key: t2_p.tableoid, t2_p.c1
 -> Data Node Scan on "__REMOTE_SORT_QUERY__"
 Output: t2_p.c1, t2_p.c2, t2_p.c3, t2_p.tableoid
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3, tableoid FROM ONLY public.t2_p WHERE c2 =
90 ORDER BY 4, 1

Remote SQL: SELECT/*+ IndexScan(t2_p)*/ c1, c2, c3, tableoid FROM ONLY public.t2_p WHERE c2 = 90
ORDER BY 4, 1
Datanode Name: datanode1
[Parameterized]
Sort
 Output: c1, c2, c3, tableoid
 Sort Key: t2_p.c1
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, tableoid
 Filter: (t2_p.c2 = $1)
 Selected Partitions: 1 (pbe-pruning)

Datanode Name: datanode2
[Parameterized]
Sort
 Output: c1, c2, c3, tableoid
 Sort Key: t2_p.c1
 -> Partitioned Seq Scan on public.t2_p
 Output: c1, c2, c3, tableoid
 Filter: (t2_p.c2 = $1)
 Selected Partitions: 1 (pbe-pruning)

(30 rows)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT * FROM (SELECT /*+ indexscan(t2) */ rowid,* FROM t2
ORDER BY c2,tableoid);
 QUERY PLAN

--
[Parameterized]
Subquery Scan on __unnamed_subquery__
 Output: __unnamed_subquery__.rowid, __unnamed_subquery__.c1, __unnamed_subquery__.c2,
__unnamed_subquery__.c3
 -> Sort
 Output: t2.rowid, t2.c1, t2.c2, t2.c3, t2.tableoid
 Sort Key: t2.c2, t2.tableoid
 -> Data Node Scan on "__REMOTE_SORT_QUERY__"
 Output: t2.rowid, t2.c1, t2.c2, t2.c3, t2.tableoid
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t2)*/ rowid, c1, c2, c3, tableoid FROM ONLY public.t2
WHERE true ORDER BY 3, 5

Remote SQL: SELECT/*+ IndexScan(t2)*/ rowid, c1, c2, c3, tableoid FROM ONLY public.t2 WHERE true
ORDER BY 3, 5
```

```

Datanode Name: datanode1
[Parameterized]
Sort
 Output: rowid, c1, c2, c3, tableoid
 Sort Key: t2.c2
 -> Result
 Output: rowid, c1, c2, c3, tableoid
 One-Time Filter: $1
 -> Seq Scan on public.t2
 Output: rowid, c1, c2, c3, tableoid

Datanode Name: datanode2
[Parameterized]
Sort
 Output: rowid, c1, c2, c3, tableoid
 Sort Key: t2.c2
 -> Result
 Output: rowid, c1, c2, c3, tableoid
 One-Time Filter: $1
 -> Seq Scan on public.t2
 Output: rowid, c1, c2, c3, tableoid
(34 rows)
-- 复制表场景，ORDER BY未下推至DN，去除tableoid排序
ora=# CREATE TABLE t1 (
ora(# c1 INTEGER,
ora(# c2 INTEGER,
ora(# c3 INTEGER
ora(#) WITH (storage_type = astore) DISTRIBUTE BY REPLICATION;
CREATE TABLE
ora=#
ora=# INSERT INTO t1 VALUES(generate_series(1,300),generate_series(1,300),generate_series(1,300));
INSERT 0 300
ora=# CREATE OR REPLACE FUNCTION func RETURN integer VOLATILE AS
ora$# BEGIN
ora$# RETURN 1;
ora$# END;
ora$# /
CREATE FUNCTION
ora=# SET enable_stream_operator=off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT tableoid,func() FROM t1 ORDER BY tableoid, c1;
WARNING: Statistics in some tables OR columns(public.t1) are not collected.
HINT: Do ANALYZE for them in ORDER to generate optimized plan.
 QUERY PLAN

Sort
 Output: t1.tableoid, (func()), t1.c1
 Sort Key: t1.c1
 -> Data Node Scan on t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.tableoid, func(), t1.c1
 Node/s: datanode1
 Remote query: SELECT tableoid, c1 FROM ONLY public.t1 WHERE true

Remote SQL: SELECT tableoid, c1 FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
 Seq Scan on public.t1
 Output: tableoid, c1
(13 rows)
ora=# RESET enable_stream_operator;
RESET

-- 单节点表
ora=# CREATE TABLE t1_single (
ora(# c1 INTEGER,
ora(# c2 INTEGER,
ora(# c3 INTEGER
ora(#) WITH (storage_type = astore, hasrowid) to group sys_single_group1;

```



```
CREATE TABLE

ora=# EXPLAIN (verbose, costs false) SELECT /*+ indexscan(t2) */ * FROM t1_single WHERE c1 = 100
ORDER BY rowid;

 QUERY PLAN

Data Node Scan on "__REMOTE_LIGHT_QUERY__"
 Output: t1_single.c1, t1_single.c2, t1_single.c3
 Node/s: datanode1
 Remote query: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t1_single WHERE c1 = 100 ORDER BY
tableoid, r
owno

Remote SQL: SELECT/*+ IndexScan(t2)*/ c1, c2, c3 FROM public.t1_single WHERE c1 = 100 ORDER BY
tableoid, rowno
Datanode Name: datanode1
Sort
 Output: c1, c2, c3, tableoid, rowno
 Sort Key: t1_single.rowno
 -> Seq Scan on public.t1_single
 Output: c1, c2, c3, tableoid, rowno
 Filter: (t1_single.c1 = 100)

(14 rows)

-- 清理数据
ora=# DROP TABLE t1;
DROP TABLE
ora=# DROP TABLE t2;
DROP TABLE
ora=# DROP TABLE t2_p;
DROP TABLE
ora=# DROP TABLE t1_single;
DROP TABLE
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "$user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

## ROWID 系统列 max/min 优化

对于普通表场景，由于ROWID到rowno的映射严格保序，min/max聚集函数条件为ROWID单列时，可改写为rowno系统列的表达式进行优化。利用索引的有序性，将min/max对应的聚集操作优化为LIMIT和索引扫描，减少数据读取和计算开销。形如“SELECT max(rowid) FROM t1”的SQL语句遵循如下改写规则：

```
SELECT rowid FROM t1 ORDER BY rowno DESC LIMIT 1
```

### 说明

ROWID系统列max/min优化遵从如下规格约束：

- 普通表场景仅支持SQL语句下推至DN时改写，单节点表在此基础上也支持执行stream计划时的改写，不支持分区表等。
- 当关闭GUC参数enable\_indexscan时或者有其他不能进行rowno列索引扫描的场景时，该优化默认不生效。enable\_indexscan参数详情可见《参考》中“数据库运行参数说明 > GUC参数说明 > 查询规划 > 优化器方法配置”章节内容。

示例：

由于不同场景下表oid字段值不同，以下示例仅作参考，执行时以实际输出结果为准。

以GUC参数explain\_perf\_mode设置为normal为例。

```
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".
-- 准备数据
ora=# SET max_datanode_for_plan = 64;
SET
ora=# SET cursor_sharing = force;
SET
ora=# SET dist_query_adaptive_mode=off;
SET
ora=# CREATE TABLE t1 (a int, b int) WITH (STORAGE_TYPE=Astore, hasrowid=ON);
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.
CREATE TABLE
ora=# INSERT INTO t1 VALUES (generate_series(1,1000), generate_series(3001,4000));
INSERT 0 1000
ora=# INSERT INTO t1 VALUES (generate_series(1001,2000), generate_series(3001,4000));
INSERT 0 1000
ora=# CREATE INDEX idx_t1_b ON t1(b);
CREATE INDEX
ora=# ANALYZE t1;
ANALYZE
-- max/min优化场景
ora=# SET enable_stream_operator=off;
SET
ora=# SET enable_fast_query_shipping = ON;
SET
ora=# SET enable_light_proxy = ON;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */ max(rowid) FROM t1;
QUERY PLAN

[Parameterized]
Aggregate
 Output: max((max(t1.rowid)))
 -> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
 Output: (max(t1.rowid))
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t1)*/ max(rowid) FROM ONLY public.t1 WHERE true

Remote SQL: SELECT/*+ IndexScan(t1)*/ max(rowid) FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
[Parameterized]
Result
 Output: $0
 InitPlan 1 (returns $0)
 -> Limit
 Output: public.t1.rowid, public.t1.rowno
 -> Result
 Output: public.t1.rowid, public.t1.rowno
 One-Time Filter: $1
 -> Index Scan Backward using t1_rowno_idx ON public.t1
 Output: public.t1.rowid, public.t1.rowno

Datanode Name: datanode2
[Parameterized]
Result
 Output: $0
 InitPlan 1 (returns $0)
 -> Limit
 Output: public.t1.rowid, public.t1.rowno
 -> Result
```

```
Output: public.t1.rowid, public.t1.rowno
One-Time Filter: $1
-> Index Scan Backward using t1_rowno_idx ON public.t1
Output: public.t1.rowid, public.t1.rowno
(35 rows)
ora=# SELECT /*+ indexscan(t1) */ rowid_sequence(max(rowid)) FROM t1;
rowid_sequence

2004
(1 row)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */ min(rowid) FROM t1;
QUERY PLAN

[Parameterized]
Aggregate
Output: min((min(t1.rowid)))
-> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
Output: (min(t1.rowid))
Node/s: All datanodes
Remote query: SELECT/*+ IndexScan(t1)*/ min(rowid) FROM ONLY public.t1 WHERE true

Remote SQL: SELECT/*+ IndexScan(t1)*/ min(rowid) FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
[Parameterized]
Result
Output: $0
InitPlan 1 (returns $0)
-> Limit
Output: public.t1.rowid, public.t1.rowno
-> Result
Output: public.t1.rowid, public.t1.rowno
One-Time Filter: $1
-> Index Scan using t1_rowno_idx ON public.t1
Output: public.t1.rowid, public.t1.rowno

Datanode Name: datanode2
[Parameterized]
Result
Output: $0
InitPlan 1 (returns $0)
-> Limit
Output: public.t1.rowid, public.t1.rowno
-> Result
Output: public.t1.rowid, public.t1.rowno
One-Time Filter: $1
-> Index Scan using t1_rowno_idx ON public.t1
Output: public.t1.rowid, public.t1.rowno
(35 rows)
ora=# SELECT /*+ indexscan(t1) */ rowid_sequence(min(rowid)) FROM t1; -- 1
rowid_sequence

1
(1 row)
-- 多个max/min使用场景，支持优化
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */ rowid_sequence(max(rowid)),max(b)
FROM t1;
QUERY PLAN

[Parameterized]
Aggregate
Output: rowid_sequence(max((max(t1.rowid)))), max((max(t1.b)))
-> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
Output: (max(t1.rowid)), (max(t1.b))
Node/s: All datanodes
Remote query: SELECT/*+ IndexScan(t1)*/ max(rowid), max(b) FROM ONLY public.t1 WHERE true
```

```
Remote SQL: SELECT /*+ IndexScan(t1)*/ max(rowid), max(b) FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
[Parameterized]
Result
Output: $0, $1
InitPlan 1 (returns $0)
-> Limit
Output: public.t1.rowid, public.t1.rowno
-> Result
Output: public.t1.rowid, public.t1.rowno
One-Time Filter: $1
-> Index Scan Backward using t1_rowno_idx ON public.t1
Output: public.t1.rowid, public.t1.rowno
InitPlan 2 (returns $1)
-> Limit
Output: public.t1.b
-> Result
Output: public.t1.b
One-Time Filter: $1
-> Index Only Scan Backward using idx_t1_b ON public.t1
Output: public.t1.b
Index Cond: (public.t1.b IS NOT NULL)

Datanode Name: datanode2
[Parameterized]
Result
Output: $0, $1
InitPlan 1 (returns $0)
-> Limit
Output: public.t1.rowid, public.t1.rowno
-> Result
Output: public.t1.rowid, public.t1.rowno
One-Time Filter: $1
-> Index Scan Backward using t1_rowno_idx ON public.t1
Output: public.t1.rowid, public.t1.rowno
InitPlan 2 (returns $1)
-> Limit
Output: public.t1.b
-> Result
Output: public.t1.b
One-Time Filter: $1
-> Index Only Scan Backward using idx_t1_b ON public.t1
Output: public.t1.b
Index Cond: (public.t1.b IS NOT NULL)

(53 rows)
ora=# SELECT /*+ indexscan(t1) */ rowid_sequence(max(rowid)),max(b) FROM t1;
rowid_sequence | max
-----+-----
2004 | 4000
(1 row)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */
rowid_sequence(max(rowid)),max(rowno) FROM t1;
QUERY PLAN

[Parameterized]
Aggregate
Output: rowid_sequence(max((max(t1.rowid)))), max((max(t1.rowno)))
-> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
Output: (max(t1.rowid)), (max(t1.rowno))
Node/s: All datanodes
Remote query: SELECT /*+ IndexScan(t1)*/ max(rowid), max(rowno) FROM ONLY public.t1 WHERE
true

Remote SQL: SELECT /*+ IndexScan(t1)*/ max(rowid), max(rowno) FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
[Parameterized]
Result
```

```

Output: $0, $1
InitPlan 1 (returns $0)
-> Limit
 Output: public.t1.rowid, public.t1.rowno
-> Result
 Output: public.t1.rowid, public.t1.rowno
 One-Time Filter: $1
 -> Index Scan Backward using t1_rowno_idx ON public.t1
 Output: public.t1.rowid, public.t1.rowno
InitPlan 2 (returns $1)
-> Limit
 Output: public.t1.rowno
-> Result
 Output: public.t1.rowno
 One-Time Filter: $1
 -> Index Only Scan Backward using t1_rowno_idx ON public.t1
 Output: public.t1.rowno

Datanode Name: datanode2
[Parameterized]
Result
Output: $0, $1
InitPlan 1 (returns $0)
-> Limit
 Output: public.t1.rowid, public.t1.rowno
-> Result
 Output: public.t1.rowid, public.t1.rowno
 One-Time Filter: $1
 -> Index Scan Backward using t1_rowno_idx ON public.t1
 Output: public.t1.rowid, public.t1.rowno
InitPlan 2 (returns $1)
-> Limit
 Output: public.t1.rowno
-> Result
 Output: public.t1.rowno
 One-Time Filter: $1
 -> Index Only Scan Backward using t1_rowno_idx ON public.t1
 Output: public.t1.rowno

(51 rows)
ora=# SELECT /*+ indexscan(t1) */ rowid_sequence(max(rowid)),max(rowno) FROM t1;
rowid_sequence | max
-----+-----
 2004 | 2004
(1 row)
-- 关闭enable_indexscan参数场景，不支持max/min优化
ora=# SET enable_indexscan = off;
SET
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */ max(rowid) FROM t1;
QUERY PLAN

[Parameterized]
Aggregate
Output: max((max(t1.rowid)))
-> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
 Output: (max(t1.rowid))
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t1)*/ max(rowid) FROM ONLY public.t1 WHERE true

Remote SQL: SELECT/*+ IndexScan(t1)*/ max(rowid) FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
[Parameterized]
Aggregate
Output: max(rowid)
-> Result
 Output: rowid
 One-Time Filter: $1
 -> Seq Scan ON public.t1

```

```
Output: rowid

Datanode Name: datanode2
[Parameterized]
Aggregate
 Output: max(rowid)
 -> Result
 Output: rowid
 One-Time Filter: $1
 -> Seq Scan ON public.t1
 Output: rowid
(29 rows)
ora=# SET enable_indexscan = ON;
SET
-- complex SQL, 不支持max/min优化
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t1) */ rowid_sequence(max(rowid)) FROM
(SELECT rowid,a FROM t1 WHERE a = 1);
 QUERY PLAN

[Parameterized]
Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
 Output: (rowid_sequence(max(__unnamed_subquery__.rowid)))
 Node expr: $1
 Remote query: SELECT/*+ IndexScan(t1)*/ rowid_sequence(max(rowid)) AS rowid_sequence FROM
(SELECT t1.rowid, t1.a FROM public.t1 WHERE t1.a = $1) __unnamed_subquery__

Remote SQL: SELECT/*+ IndexScan(t1)*/ rowid_sequence(max(rowid)) AS rowid_sequence FROM (SELECT
t1.rowid, t1.a FROM public.t1 WHERE t1.a = 1) __unnamed_subquery__
Datanode Name: datanode1
[Parameterized]
Aggregate
 Output: rowid_sequence(max(t1.rowid))
 -> Seq Scan ON public.t1
 Output: t1.rowid
 Filter: (t1.a = $1)
(15 rows)
-- 分区表场景, 不支持max/min优化
ora=# CREATE TABLE t2
ora=# (a int,
ora=# b int,
ora=# c int)
ora=# WITH (STORAGE_TYPE=Astore, hasrowid=ON)
ora=# PARTITION BY RANGE (a)
ora=# (
ora=# PARTITION P1 VALUES LESS THAN(100),
ora=# PARTITION P2 VALUES LESS THAN(200),
ora=# PARTITION P3 VALUES LESS THAN(300),
ora=# PARTITION P8 VALUES LESS THAN(MAXVALUE)
ora=#);
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.
CREATE TABLE
ora=# INSERT INTO t2 VALUES (generate_series(1,1000), generate_series(1001,2000));
INSERT 0 1000
ora=# ANALYZE t2;
ANALYZE
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ max(rowid) FROM t2;
 QUERY PLAN

[Parameterized]
Aggregate
 Output: max((max(t2.rowid)))
 -> Data Node Scan ON "__REMOTE_GROUP_QUERY__"
 Output: (max(t2.rowid))
 Node/s: All datanodes
 Remote query: SELECT/*+ IndexScan(t2)*/ max(rowid) FROM ONLY public.t2 WHERE true
```

```

Remote SQL: SELECT/*+ IndexScan(t2)*/ max(rowid) FROM ONLY public.t2 WHERE true
Datanode Name: datanode1
[Parameterized]
Aggregate
 Output: max(rowid)
 -> Result
 Output: rowid
 One-Time Filter: $1
 -> Partition Iterator
 Output: rowid
 Iterations: 4
 -> Partitioned Seq Scan ON public.t2
 Output: rowid
 Selected Partitions: 1..4

Datanode Name: datanode2
[Parameterized]
Aggregate
 Output: max(rowid)
 -> Result
 Output: rowid
 One-Time Filter: $1
 -> Partition Iterator
 Output: rowid
 Iterations: 4
 -> Partitioned Seq Scan ON public.t2
 Output: rowid
 Selected Partitions: 1..4

(37 rows)
ora=# EXPLAIN (VERBOSE, COSTS FALSE) SELECT /*+ indexscan(t2) */ max(rowid) FROM t2 WHERE a = 1;
 QUERY PLAN

[Parameterized]
Data Node Scan ON "__REMOTE_LIGHT_QUERY__"
 Output: (max(t2.rowid))
 Node expr: $1
 Remote query: SELECT/*+ IndexScan(t2)*/ max(rowid) AS max FROM public.t2 WHERE a = $1

Remote SQL: SELECT/*+ IndexScan(t2)*/ max(rowid) AS max FROM public.t2 WHERE a = 1
Datanode Name: datanode1
[Parameterized]
Aggregate
 Output: max(rowid)
 -> Partitioned Seq Scan ON public.t2
 Output: rowid
 Filter: (t2.a = $1)
 Selected Partitions: 1 (pbe-pruning)

(16 rows)

-- 单节点表
ora=# CREATE TABLE t1_single (a int, b int) WITH (STORAGE_TYPE=Astore, hasrowid=on) to group
sys_single_group1;
CREATE TABLE
ora=# INSERT INTO t1_single VALUES (generate_series(1,1000), generate_series(3001,4000));
INSERT 0 1000
ora=# INSERT INTO t1_single VALUES (generate_series(1001,2000), generate_series(3001,4000));
INSERT 0 1000
ora=# EXPLAIN (verbose, costs false) SELECT /*+ indexscan(t1_single) */ max(rowid) FROM t1_single;
 QUERY PLAN

Data Node Scan on "__REMOTE_LIGHT_QUERY__"
 Output: (max(t1_single.rowid))
 Node/s: datanode1
 Remote query: SELECT/*+ IndexScan(t1_single)*/ max(rowid) AS max FROM public.t1_single

```

```
Remote SQL: SELECT/*+ IndexScan(t1_single)*/ max(rowid) AS max FROM public.t1_single
Datanode Name: datanode1
Result
Output: $0
InitPlan 1 (returns $0)
-> Limit
Output: public.t1_single.rowid, public.t1_single.rowno
-> Index Scan Backward using t1_single_rowno_idx on public.t1_single
Output: public.t1_single.rowid, public.t1_single.rowno

(15 rows)

-- 清理数据
ora=# DROP TABLE t2;
DROP TABLE
ora=# DROP TABLE t1;
DROP TABLE
ora=# DROP TABLE t1_single;
DROP TABLE
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

## ROWID 系统列分布剪枝和分区剪枝

分布剪枝在ROWID系统列作为等值谓词条件的场景下触发，可将查询裁剪至唯一的单个DN节点。在分布式静态剪枝阶段，系统会在分布键剪枝之前优先执行ROWID剪枝。在获取ROWID列谓词后，对于能够明确裁剪至单DN节点的场景，直接计算下推节点；对于外部参数场景，则保存相关参数，并在动态剪枝阶段再计算下推节点；对于不支持的场景，则进行常规分布键剪枝。

### 说明

分布剪枝与当前分布列剪枝遵循相同的规格约束，如下：

- 仅支持ROWID作为等值表达式或者NULL表达式。
- 对于布尔类型组合场景，仅对AND表达式中首个ROWID条件剪枝，不对OR表达式剪枝。
- 仅支持常量（静态剪枝）、参数（动态剪枝）两种场景的剪枝。
- 静态剪枝支持4种分布式计划，动态剪枝只支持CN轻量化、FQS和PGXC计划。
- 分布剪枝不支持函数。
- 当分布剪枝涉及的列或条件存在类型转换时，仅支持隐式二进制强制转换，不支持函数转换或显式转换。

在触发ROWID系统列谓词条件改写，且改写后条件包含tableoid等值判断的场景下，可进行分区剪枝裁剪到唯一单分区。同时支持常量和参数条件，对于ROWID常量条件触发静态剪枝，对于ROWID参数条件触发动态剪枝。更多分区剪枝详情请参见[分区剪枝](#)章节内容。

示例：

分布剪枝示例如下：

由于不同场景下表oid字段值不同，以下示例仅作参考，执行时以实际输出结果为准。

以GUC参数explain\_perf\_mode设置为normal为例。

```
-- 数据准备
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
```



```
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".
ora=# SET max_datanode_for_plan = 1;
SET
ora=# SET query_dop=1;
SET
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# SET auto_explain_level=notice;
SET
ora=# CREATE TABLE t1 (c1 int, c2 int) WITH (storage_type=astore, hasrowid=on);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
ora=# INSERT INTO t1 SELECT generate_series(1,3), generate_series(1,4);
INSERT 0 12
ora=# ANALYZE t1;
ANALYZE
-- CN轻量化
-- 常量表达式静态剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE';
 QUERY PLAN

Data Node Scan on "__REMOTE_LIGHT_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid) AND tableoid = rowid_tableoid('AAAAAAAAAP//AAAEEOAAAAAAAAAAE':rowid) AND rowno = rowid_sequence('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid)

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid) AND tableoid = rowid_tableoid('AAAAAAAAAP//AAAEEOAAAAAAAAAAE':rowid) AND rowno = rowid_sequence('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE':rowid)
Datanode Name: datanode1
 Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Index Cond: (t1.rowno = 4::bigint)
 Filter: ((t1.rowid = 'AAAAAAAAAP//AAAEEOAAAAAAAAAAE':rowid) AND (t1.tableoid = 16654::oid))
(12 rows)
-- 参数表达式动态剪枝
ora=# SET plan_cache_mode = force_generic_plan;
SET
ora=# PREPARE p1 AS SELECT * FROM t1 WHERE rowid = $1;
PREPARE
ora=# EXPLAIN (costs off, verbose on) EXECUTE p1('AAAAAAAAAP//AAAEEOAAAAAAAAAAE');
 QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Node expr: $1
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid = $1 AND (-1)::smallint =
rowid_hashbucketid($1) AND tableoid = rowid_tableoid($1) AND rowno = rowid_sequence($1)
```

```

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid = $1 AND (-1)::smallint =
rowid_hashbucketid($1) AND tableoid = rowid_tableoid($1) AND rowno = rowid_sequence($1)
Datanode Name: datanode1
Result
 Output: c1, c2
 One-Time Filter: ((-1)::smallint = rowid_hashbucketid($1))
 -> Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Index Cond: (t1.rowno = rowid_sequence($1))
 Filter: ((t1.rowid = $1) AND (t1.tableoid = rowid_tableoid($1)))
(16 rows)
ora=# EXECUTE p1('AAAAAAAAP//AAAEEOAAAAAAAAAAE');
c1 | c2
----+----
 2 | 1
(1 row)
ora=# DEALLOCATE p1;
DEALLOCATE
-- AND表达式剪枝首个ROWID表达式，静态剪枝
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
 QUERY PLAN

Data Node Scan on "__REMOTE_LIGHT_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL AND c1 = 1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL AND c1 = 1
Datanode Name: datanode1
 Seq Scan on public.t1
 Output: c1, c2
 Filter: (t1.c1 = 1)
(11 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
c1 | c2
----+----
 1 | 1
 1 | 4
 1 | 3
 1 | 2
(4 rows)
-- OR表达式不支持剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
 QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: All datanodes
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL OR c1 = 1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL OR c1 = 1
Datanode Name: datanode1
 Seq Scan on public.t1
 Output: c1, c2
(10 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
c1 | c2
----+----
 1 | 1
 2 | 2
 1 | 4
 2 | 1
 1 | 3
 2 | 4
 1 | 2

```

```
2 | 3
3 | 3
3 | 2
3 | 1
3 | 4
(12 rows)

-- FQS计划
ora=# SET enable_light_proxy = off;
SET
ora=# SET plan_cache_mode = force_custom_plan;
SET
-- 常量表达式静态剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE';
 QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid) AND tableoid = rowid_tableoid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid) AND rowno = rowid_sequence('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid)

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid) AND tableoid = rowid_tableoid('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid) AND rowno = rowid_sequence('AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid)
 Datanode Name: datanode1
 Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Index Cond: (t1.rowno = 4::bigint)
 Filter: ((t1.rowid = 'AAAAAAAAAP//AAAEEOAAAAAAAAAAE'::rowid) AND (t1.tableoid = 16654::oid))
(12 rows)
-- 参数表达式动态剪枝
ora=# SET plan_cache_mode = force_generic_plan;
SET
ora=# PREPARE p1 AS SELECT * FROM t1 WHERE rowid = $1;
PREPARE
ora=# EXPLAIN (costs off, verbose on) EXECUTE p1('AAAAAAAAAP//AAAEEOAAAAAAAAAAE');
 QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Node expr: $1
 Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid = $1 AND (-1)::smallint =
rowid_hashbucketid($1) AND tableoid = rowid_tableoid($1) AND rowno = rowid_sequence($1)

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid = $1 AND (-1)::smallint =
rowid_hashbucketid($1) AND tableoid = rowid_tableoid($1) AND rowno = rowid_sequence($1)
 Datanode Name: datanode1
 Result
 Output: c1, c2
 One-Time Filter: ((-1)::smallint = rowid_hashbucketid($1))
 -> Index Scan using t1_rowno_idx on public.t1
```

```

Output: c1, c2
Index Cond: (t1.rowno = rowid_sequence($1))
Filter: ((t1.rowid = $1) AND (t1.tableoid = rowid_tableoid($1)))
(16 rows)
ora=# EXECUTE p1('AAAAAAAAP//AAAEEOAAAAAAAAAAE');
c1 | c2
----+----
 2 | 1
(1 row)
ora=# DEALLOCATE p1;
DEALLOCATE
-- AND表达式剪枝首个ROWID表达式，静态剪枝
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
Output: t1.c1, t1.c2
Node/s: datanode1
Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL AND c1 = 1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL AND c1 = 1
Datanode Name: datanode1
Seq Scan on public.t1
Output: c1, c2
Filter: (t1.c1 = 1)
(11 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
c1 | c2
----+----
 1 | 1
 1 | 4
 1 | 3
 1 | 2
(4 rows)
-- OR表达式不支持剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1; --all
QUERY PLAN

Data Node Scan on "__REMOTE_FQS_QUERY__"
Output: t1.c1, t1.c2
Node/s: All datanodes
Remote query: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL OR c1 = 1

Remote SQL: SELECT c1, c2 FROM public.t1 WHERE rowid IS NOT NULL OR c1 = 1
Datanode Name: datanode1
Seq Scan on public.t1
Output: c1, c2
(10 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
c1 | c2
----+----
 1 | 1
 2 | 2
 1 | 4
 2 | 1
 1 | 3
 2 | 4
 1 | 2
 2 | 3
 3 | 3
 3 | 2
 3 | 1
 3 | 4
(12 rows)

```

```
-- stream计划
ora=# SET enable_fast_query_shipping = off;
SET
ora=# SET plan_cache_mode = force_custom_plan;
SET
-- 常量表达式静态剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE';
 QUERY PLAN

Streaming (type: GATHER)
Output: c1, c2
Node/s: datanode1
-> Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Distribute Key: c1
 Index Cond: (t1.rowno = 4::bigint)
 Filter: ((t1.rowid = 'AAAAAAAAAP//AAAEEOAAAAAAAAAAE'::rowid) AND (t1.tableoid = 16654::oid))
(8 rows)
-- 参数表达式，动态剪枝不支持stream计划
ora=# SET plan_cache_mode = force_generic_plan;
SET
ora=# PREPARE p1 AS SELECT * FROM t1 WHERE rowid = $1;
PREPARE
ora=# EXPLAIN (costs off, verbose on) EXECUTE p1('AAAAAAAAAP//AAAEEOAAAAAAAAAAE');
 QUERY PLAN

Streaming (type: GATHER)
Output: c1, c2
Node/s: All datanodes
-> Result
 Output: c1, c2
 One-Time Filter: ((-1)::smallint = rowid_hashbucketid($1))
 -> Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Distribute Key: c1
 Index Cond: (t1.rowno = rowid_sequence($1))
 Filter: ((t1.rowid = $1) AND (t1.tableoid = rowid_tableoid($1)))
(11 rows)
ora=# EXECUTE p1('AAAAAAAAAP//AAAEEOAAAAAAAAAAE');
 c1 | c2
-----+-----
 2 | 1
(1 row)
ora=# DEALLOCATE p1;
DEALLOCATE
-- AND表达式剪枝首个ROWID表达式，静态剪枝
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
 QUERY PLAN

Streaming (type: GATHER)
Output: c1, c2
Node/s: datanode1
-> Seq Scan on public.t1
 Output: c1, c2
 Distribute Key: c1
 Filter: (t1.c1 = 1)
(7 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
 c1 | c2
-----+-----
 1 | 1
 1 | 4
 1 | 3
 1 | 2
(4 rows)
```

```
-- OR表达式不支持剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
QUERY PLAN

Streaming (type: GATHER)
 Output: c1, c2
 Node/s: All datanodes
 -> Seq Scan on public.t1
 Output: c1, c2
 Distribute Key: c1
(6 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
c1 | c2
----+----
 1 | 1
 2 | 2
 1 | 4
 2 | 1
 1 | 3
 2 | 4
 1 | 2
 2 | 3
 3 | 3
 3 | 2
 3 | 1
 3 | 4
(12 rows)

-- PGXC计划
ora=# SET enable_fast_query_shipping = off;
SET
ora=# SET enable_stream_operator = off;
SET
ora=# SET plan_cache_mode = force_custom_plan;
SET
-- 常量表达式静态剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE';
QUERY PLAN

Data Node Scan on t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid AND tableoid = 16654::oid AND rowno = 4::bigint

Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE rowid = 'AAAAAAAAAP//
AAAEEOAAAAAAAAAAE'::rowid AND tableoid = 16654::oid AND rowno = 4::bigint
Datanode Name: datanode1
 Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Index Cond: (t1.rowno = 4::bigint)
 Filter: ((t1.rowid = 'AAAAAAAAAP//AAAEEOAAAAAAAAAAE'::rowid) AND (t1.tableoid = 16654::oid))
(12 rows)
-- 参数表达式动态剪枝
ora=# SET plan_cache_mode = force_generic_plan;
SET
ora=# PREPARE p1 AS SELECT * FROM t1 WHERE rowid = $1;
PREPARE
ora=# EXPLAIN (costs off, verbose on) EXECUTE p1('AAAAAAAAAP//AAAEEOAAAAAAAAAAE');
QUERY PLAN

Result
```

```
Output: t1.c1, t1.c2
One-Time Filter: ((-1)::smallint = rowid_hashbucketid($1))
-> Data Node Scan on t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.c1, t1.c2
 Node expr: $1
 Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE rowid = $1 AND tableoid =
rowid_tableoid($1) AND rowno = rowid_sequence($1) AND (-1)::smallint = rowid_hashbucketid($1)

Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE rowid = 'AAAAAAAAP//
AAAEEOAAAAAAAAAAAE':rowid AND tableoid = 16654::oid AND rowno = 4::bigint
Datanode Name: datanode1
 Index Scan using t1_rowno_idx on public.t1
 Output: c1, c2
 Index Cond: (t1.rowno = 4::bigint)
 Filter: ((t1.rowid = 'AAAAAAAAP//AAAEEOAAAAAAAAAAAE':rowid) AND (t1.tableoid = 16654::oid))
(15 rows)
ora=# EXECUTE p1('AAAAAAAAP//AAAEEOAAAAAAAAAAAE');
c1 | c2
----+----
 2 | 1
(1 row)
ora=# DEALLOCATE p1;
DEALLOCATE
-- AND表达式剪枝首个ROWID表达式，静态剪枝
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
 QUERY PLAN

Data Node Scan on t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.c1, t1.c2
 Node/s: datanode1
 Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE c1 = 1

Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE c1 = 1
Datanode Name: datanode1
 Seq Scan on public.t1
 Output: c1, c2
 Filter: (t1.c1 = 1)
(11 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL AND c1 = 1;
c1 | c2
----+----
 1 | 1
 1 | 4
 1 | 3
 1 | 2
(4 rows)
-- OR表达式不支持剪枝
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
 QUERY PLAN

Data Node Scan on t1 "_REMOTE_TABLE_QUERY_"
 Output: t1.c1, t1.c2
 Node/s: All datanodes
 Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE true

Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE true
Datanode Name: datanode1
 Seq Scan on public.t1
 Output: c1, c2
(10 rows)
ora=# SELECT * FROM t1 WHERE rowid IS NOT NULL OR c1 = 1;
c1 | c2
----+----
 1 | 1
```

```
2 | 2
1 | 4
2 | 1
1 | 3
2 | 4
1 | 2
2 | 3
3 | 3
3 | 2
3 | 1
3 | 4
(12 rows)

-- 清理数据
ora=# DROP TABLE t1;
DROP TABLE
ora=# RESET max_datanode_for_plan;
RESET
ora=# RESET query_dop;
RESET
ora=# RESET enable_light_proxy;
RESET
ora=# RESET enable_fast_query_shipping;
RESET
ora=# RESET enable_stream_operator;
RESET
ora=# RESET plan_cache_mode;
RESET
ora=# RESET auto_explain_level;
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "$user".
gaussdb=# DROP DATABASE ora;
DROP DATABASE
```

分区剪枝示例如下：

由于不同场景下tableoid字段值不同，以下示例仅作参考，执行时以实际输出结果为准。

```
-- 准备数据
gaussdb=# CREATE DATABASE ora dbcompatibility 'ORA';
CREATE DATABASE
gaussdb=# \c ora
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora" as user "$user".
ora=# SET max_datanode_for_plan = 1;
SET
ora=# SET query_dop=1;
SET
ora=# SET auto_explain_level=notice;
SET
ora=# CREATE TABLE t_part (c1 int, c2 int) WITH (storage_type=astore, hasrowid=on) PARTITION BY
HASH(c1) PARTITIONS 4;
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
ora=# INSERT INTO t_part SELECT generate_series(1,3), generate_series(1,4);
INSERT 0 12
ora=# ANALYZE t_part;
ANALYZE
-- CN轻量化，cplan场景
ora=# SET plan_cache_mode = force_custom_plan;
SET
-- 剪枝至p1分区
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO';
 QUERY PLAN


```



```

Data Node Scan on "_REMOTE_LIGHT_QUERY_"
Output: t_part.c1, t_part.c2
Node/s: datanode2
Remote query: SELECT c1, c2 FROM public.t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid) AND tableoid = rowid
_tableoid('AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND rowno = rowid_sequence('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid)

Remote SQL: SELECT c1, c2 FROM public.t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid) AND tableoid = rowid_tab
leoid('AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND rowno = rowid_sequence('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid)
Datanode Name: datanode2
Partitioned Index Scan using t_part_rowno_idx on public.t_part
Output: c1, c2
Index Cond: (t_part.rowno = 14::bigint)
Filter: ((t_part.rowid = 'AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND (t_part.tableoid =
16664::oid))
Selected Partitions: 1
(13 rows)

-- FQS计划, gplan场景
ora=# SET enable_light_proxy = off;
SET
ora=# SET plan_cache_mode = force_generic_plan;
SET
-- 剪枝至p1分区
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO';
QUERY PLAN

Data Node Scan on "_REMOTE_FQS_QUERY_"
Output: t_part.c1, t_part.c2
Node/s: datanode2
Remote query: SELECT c1, c2 FROM public.t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid) AND tableoid = rowid
_tableoid('AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND rowno = rowid_sequence('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid)

Remote SQL: SELECT c1, c2 FROM public.t_part WHERE rowid = 'AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid AND (-1)::smallint = rowid_hashbucketid('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid) AND tableoid = rowid_tab
leoid('AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND rowno = rowid_sequence('AAAAAAAABP//
AAAEYAAAAAAAAAAO':rowid)
Datanode Name: datanode2
Partitioned Index Scan using t_part_rowno_idx on public.t_part
Output: c1, c2
Index Cond: (t_part.rowno = 14::bigint)
Filter: ((t_part.rowid = 'AAAAAAAABP//AAAEYAAAAAAAAAAO':rowid) AND (t_part.tableoid =
16664::oid))
Selected Partitions: 1
(13 rows)

-- stream计划, cplan场景
ora=# SET enable_fast_query_shipping = off;
SET
ora=# SET plan_cache_mode = force_custom_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t_part WHERE rowid = 'AAAAAAAABP//
```

```

AAAAEYAAAAAAAAAAO';

QUERY PLAN

Streaming (type: GATHER)
 Output: c1, c2
 Node/s: datanode2
 -> Partition Iterator
 Output: c1, c2
 Iterations: PART
 -> Partitioned Index Scan using t_part_rowno_idx on public.t_part
 Output: c1, c2
 Distribute Key: c1
 Index Cond: (t_part.rowno = 14::bigint)
 Filter: ((t_part.rowid = 'AAAAAAAABP//AAAAEYAAAAAAAAAAO'::rowid) AND (t_part.tableoid =
16664::oid))
 Selected Partitions: 1..4 (pbe-pruning)
(12 rows)
-- stream计划, gplan场景
ora=# SET plan_cache_mode = force_generic_plan;
SET
ora=# EXPLAIN (costs off, verbose on) SELECT * FROM t_part WHERE rowid = 'AAAAAAAABP//
AAAAEYAAAAAAAAAAO';

QUERY PLAN

Streaming (type: GATHER)
 Output: c1, c2
 Node/s: datanode2
 -> Partition Iterator
 Output: c1, c2
 Iterations: PART
 -> Partitioned Index Scan using t_part_rowno_idx on public.t_part
 Output: c1, c2
 Distribute Key: c1
 Index Cond: (t_part.rowno = 14::bigint)
 Filter: ((t_part.rowid = 'AAAAAAAABP//AAAAEYAAAAAAAAAAO'::rowid) AND (t_part.tableoid =
16664::oid))
 Selected Partitions: 1..4 (pbe-pruning)
(12 rows)

-- 清理数据
ora=# DROP TABLE t_part;
DROP TABLE
ora=# RESET max_datanode_for_plan;
RESET
ora=# RESET query_dop;
RESET
ora=# RESET enable_light_proxy;
RESET
ora=# RESET enable_fast_query_shipping;
RESET
ora=# RESET enable_stream_operator;
RESET
ora=# RESET plan_cache_mode;
RESET
ora=# RESET auto_explain_level;
RESET
ora=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "postgres".
gaussdb=# DROP DATABASE ora;
DROP DATABASE

```

# 8 基础概念

## 8.1 数据库对象

数据库对象是数据库逻辑结构的集合，常见的数据库对象有：表、视图、索引、序列、存储过程、包、用户等。

### 数据库

数据库用于管理各类数据对象，与其他数据库隔离。创建数据对象时可以指定对应的表空间，如果不指定相应的表空间，相关的对象会默认保存在PG\_DEFAULT空间中。数据库管理的对象可分布在多个表空间上。

### 表空间

在GaussDB中，表空间是一个目录，实例中可以存在多个表空间，里面存储的是它所包含的数据库的各种物理文件。由于表空间是一个目录，仅起到了物理隔离的作用，其管理功能依赖于文件系统。

### 表

表是一系列二维数组的集合，是数据组织的基础单元，由行（row）和列（column）构成。列又称为字段名。数据库表中的一行数据称为一条记录，多数是由同类信息组成。

- 表分为普通表和临时表。
  - 临时表的生命周期仅存在于会话期间，会话断开后会自动删除临时表。
  - 临时表可以和普通表重名。当同时存在同名的临时表和普通表时，查看表结构、DML等操作都作用于临时表。
- 列  
列用于记录表上某个属性的值，用户给每个属性定义的名称称为列名。除了列名以外，还有数据类型以及数据类型最大长度（精度）等信息。
- 行  
行表示表中单条记录中所有列的集合。表中每行代表一组相关数据，且每行都具有相同的结构。

## 视图

视图是一个虚拟表，其本身不存储数据。本质上视图是一个预定义的查询语句，用户可以通过访问视图来检索数据。在SELECT语句中涉及到的表称为基表，当基表数据发生变化，视图也会随之变化。删除视图不会对基表产生影响。

- 优点
  - 简单性：视图可以简化用户对数据的理解 and 操作。常用的查询可以定义为视图，用户无需每次都指定全部条件。
  - 安全性：通过视图，用户只能访问其可见的数据，表中其他列对用户不可见，从而实现更细粒度的权限控制。
- 缺点
  - 性能问题：尽量避免在视图使用视图（嵌套视图），因为这可能会使查询的性能下降。
  - 维护的复杂性：需要考虑修改了基表表结构后视图是否可用。

## 索引

索引是一种数据结构，包含着对数据表中所有记录的引用指针，类似于书籍的目录，用于加快数据库查询速度和性能。

- 优点
  - 加快查询速度，降低数据库的I/O成本。
  - 降低数据库的排序成本，可以减少分组和排序的时间。
  - 通过唯一索引，可以保证表中数据的唯一性。
- 缺点
  - 索引需要占用额外的存储空间。
  - 在插入和修改数据时，需要花费更多时间。
- 索引的可用性和可见性
  - 索引可用性  
在删除分区场景，如果没有指定需要重建索引字段，会将索引标记为UNUSABLE（即不可用状态）。在DML操作中，索引无需维护，并且优化器会忽略该索引。
  - 索引可见性  
索引的可见性指的是优化器是否忽略该索引。如果索引不可见，优化器会忽略它，但在DML操作中，索引仍需维护。通常，在删除索引之前，可以先将索引设置为不可见状态，以观察对业务的影响。确认无影响后，再删除索引。

## 用户和角色

- 用户（USER）是数据库的访问主体，每个用户都有自己的用户名和密码。用户可以执行数据库操作，而用户的权限通常来源于分配给用户的角色。
  - 初始用户：  
集群安装过程中自动生成的账户称为初始用户。初始用户也是系统管理员、安全管理员、审计管理员、监控管理员、运维管理员和安全策略管理员，拥有系统的最高权限，能够执行所有的操作。如果安装时不设置初始用户名，则该账户与进行集群安装的操作系统用户同名。如果在安装集群时不

置初始用户的密码，安装完成后密码为空，在执行其他操作前需要通过gsq客户端修改初始用户的密码。如果初始用户密码为空，则除修改密码外，无法执行其他SQL操作以及升级、扩容、节点替换等操作。

#### 说明

- 初始用户的OID为10，可以通过gs\_roles视图查询。
- 初始用户会绕过所有权限检查。建议仅将初始用户作为DBA管理用途，而非业务应用。

#### - 系统管理员（SYSADMIN）：

系统管理员是指具有SYSADMIN属性的账户，默认安装情况下具有与对象所有者相同的权限，但不包括dbe\_perf模式的对象权限。

要创建新的安全管理员，三权分立关闭时，请以系统管理员或者安全管理员身份连接数据库。三权分立打开时，请以安全管理员身份连接数据库，并使用带CREATEROLE选项的语句进行设置。详情请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建时指定该用户为系统管理员。
gaussdb=# CREATE USER u1 WITH SYSADMIN PASSWORD '*****';

--创建用户后使用ALTER语句指定该用户为系统管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 SYSADMIN;

--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP USER u2;
```

#### - 安全管理员（CREATEROLE）：

安全管理员是指具有CREATEROLE属性的账户，具有创建、修改、删除用户或角色的权限，和授予或者撤销任何非系统管理员、内置角色、永久用户、运维管理员的权限。

要创建新的安全管理员，三权分立关闭时，请以系统管理员或者安全管理员身份连接数据库。三权分立打开时，请以安全管理员身份连接数据库，并使用带CREATEROLE选项的语句进行设置。详情请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建用户时指定用户为安全管理员。
gaussdb=# CREATE USER u1 WITH CREATEROLE PASSWORD '*****';

--创建用户后使用ALTER语句指定该用户为安全管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 CREATEROLE;

--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP USER u2;
```

#### - 审计管理员（AUDITADMIN）：

审计管理员是指具有AUDITADMIN属性的账户，具有查看和删除审计日志的权限。

要创建新的审计管理员，三权分立关闭时，请以系统管理员或者安全管理员身份连接数据库。三权分立打开时，只能以初始用户身份连接数据库，并使用带AUDITADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建时指定该用户为审计管理员。
gaussdb=# CREATE USER auditadmin WITH AUDITADMIN password '*****';
```

```
--创建用户后使用ALTER语句指定该用户为审计管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 AUDITADMIN;
```

```
--删除。
gaussdb=# DROP USER auditadmin;
gaussdb=# DROP USER u2;
```

#### - 监控管理员（MONADMIN）

监控管理员是指具有MONADMIN属性的账户，具有查看dbe\_perf模式下视图和函数的权限，亦可以对dbe\_perf模式的对象权限进行授予或收回。

要创建新的监控管理员，请以系统管理员身份连接数据库，并使用带MONADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建时指定该用户为监控管理员。
gaussdb=# CREATE USER u1 WITH MONADMIN password '*****';
```

```
--创建用户后使用ALTER语句指定该用户为监控管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 MONADMIN;
```

```
--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP USER u2;
```

#### - 运维管理员（OPRADMIN）

运维管理员是指具有OPRADMIN属性的账户，具有使用Roach工具执行备份恢复的权限。

要创建新的运维管理员，请以初始用户身份连接数据库，并使用带OPRADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建时指定该用户为运维管理员。
gaussdb=# CREATE USER u1 WITH OPRADMIN password '*****';
```

```
--创建用户后使用ALTER语句指定该用户为运维管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 OPRADMIN;
```

```
--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP USER u2;
```

#### - 安全策略管理员（POLADMIN）

安全策略管理员是指具有POLADMIN属性的账户，具有创建资源标签、脱敏策略和统一审计策略的权限。

要创建新的安全策略管理员，请以系统管理员用户身份连接数据库，并使用带POLADMIN选项的语句进行设置。详细请参见《参考》中“SQL参考 > SQL语法 > C > CREATE USER”语句或《参考》中“SQL参考 > SQL语法 > A > ALTER USER”语句。

```
--创建时指定该用户为安全策略管理员。
gaussdb=# CREATE USER u1 WITH POLADMIN password '*****';
```

```
--创建用户后使用ALTER语句指定该用户为安全策略管理员。
gaussdb=# CREATE USER u2 PASSWORD '*****';
gaussdb=# ALTER USER u2 POLADMIN;
```

```
--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP USER u2;
```

- 普通用户：这些用户通常被赋予执行特定业务操作所需的权限。普通用户的创建可以通过CREATE USER语句，并且可以指定密码、有效时间、连接限制等。  
--创建用户u1。  
gaussdb=# CREATE USER u1 PASSWORD '\*\*\*\*\*';  
  
--删除。  
gaussdb=# DROP USER u1;
- 角色（ROLE）是一组权限的集合，用于组织和管理权限。通过将角色授予给用户，使用户继承角色的所有权限。角色之间的权限可以继承。
  - public角色：GaussDB提供了一个隐式定义的角色public，所有创建的用户和角色默认属于public角色。可以通过GRANT和REVOKE语句来修改PUBLIC的权限。
  - 内置角色：GaussDB提供了一组默认角色，以gs\_role\_开头命名，这些角色提供了对特定高权限操作的访问。例如，gs\_role\_signal\_backend角色允许调用函数来取消或终止其他会话，而gs\_role\_tablespace角色允许创建表空间。
- 用户与角色的关系：用户的权限来自于角色。用户可以看作是具有登录权限的角色，而角色可以看作是没有登录权限的用户。通过角色，可以更方便地管理和分配权限，尤其是在有多个用户需要相同权限的情况下。通过这些机制，GaussDB实现了灵活且安全的权限管理，使得数据库管理员可以根据业务需求和安全性要求，合理地分配和管理用户权限。  
--创建角色r1。  
gaussdb=# CREATE ROLE r1 PASSWORD '\*\*\*\*\*';  
  
--创建用户u1。  
gaussdb=# CREATE USER u1 PASSWORD '\*\*\*\*\*';  
  
--将角色r1权限赋予u1。  
gaussdb=# GRANT r1 TO u1;  
  
--查询上述的角色和用户信息。  
gaussdb=# \du r1|u1;  
List of roles  
Role name | Attributes | Member of  
-----+-----+-----  
r1 | Cannot login | {}  
u1 | | {r1}  
  
--删除用户和角色。  
gaussdb=# DROP USER u1;  
gaussdb=# DROP ROLE r1;

## 8.2 模式（Schema）

### 简介

模式（Schema）是数据库的对象之一，用于组织数据库中的表、视图、索引、函数和操作符等对象，其作用如下：

- 逻辑隔离：将数据库对象组织成易于管理的逻辑组，不同的模式可以包含同名的对象而不冲突。
- 权限控制：通过模式可以更细粒度地控制用户对数据库的访问权限。可以在模式级别授予或撤销权限，确保数据的安全性和隔离性。
- 便于管理：模式使得数据库对象管理更加方便灵活，特别是在多应用、多用户的环境中。

## 创建和管理模式

如下仅介绍一些常用的语法，详情见《参考》中“SQL参考 > SQL语法”下的CREATE SCHEMA、ALTER SCHEMA和DROP SCHEMA章节。

- 创建Schema

```
gaussdb=# CREATE SCHEMA sch1 ;
CREATE SCHEMA
```

```
--查看Schema信息。
gaussdb=# \dn sch1
List of schemas
Name | Owner
-----+-----
sch1 | omm
(1 row)
```

- 修改Schema

```
--前置操作，创建用户。
gaussdb=# CREATE USER u1 PASSWORD '*****';
CREATE ROLE
```

```
--修改模式的名称。
gaussdb=# ALTER SCHEMA sch1 RENAME TO sch2;
ALTER SCHEMA
```

```
--修改模式的拥有者。
gaussdb=# ALTER SCHEMA sch2 OWNER TO u1;
ALTER SCHEMA
```

```
--查看Schema信息。
gaussdb=# \dn sch*
List of schemas
Name | Owner
-----+-----
sch2 | u1
(1 row)
```

- 删除Schema

```
gaussdb=# DROP SCHEMA sch2;
DROP SCHEMA
```

```
--删除用户。
gaussdb=# DROP USER u1;
DROP ROLE
```

## 搜索路径

搜索路径用于确定在解析数据库对象名称时，数据库将按照什么顺序去对应的模式下查找。搜索路径的设置可以影响SQL查询的解析和执行。当用户使用一个对象（表、视图、索引、函数等）而没有指定Schema时，数据库会按照搜索路径中定义的顺序依次查找，直到找到该对象时结束。

- 查询搜索路径。

如下示例中查询结果：在没有指定模式名的情况下，数据库在查找对象时首先在与用户名同名的模式下查找，然后再去public模式下查找。

```
--查询当前搜索路径。
gaussdb=# SHOW search_path;
search_path

"$user",public
(1 row)
```

- 设置搜索路径。

- 搜索路径的设置只在当前会话中生效，不会影响到其他会话。



- 创建对象时，如果没有指定Schema，对象将默认创建在搜索路径中第一个Schema中。

```
--创建模式。
gaussdb=# CREATE SCHEMA sch1;
CREATE SCHEMA
gaussdb=# CREATE SCHEMA sch2;
CREATE SCHEMA

--分别在两个模式下创建名称相同的表并插入数据。
gaussdb=# CREATE TABLE sch1.tb_test(c1 int);
CREATE TABLE
gaussdb=# INSERT INTO sch1.tb_test VALUES (1);
INSERT 0 1
gaussdb=# CREATE TABLE sch2.tb_test(c1 int);
CREATE TABLE
gaussdb=# INSERT INTO sch2.tb_test VALUES (2);
INSERT 0 1

--设置搜索路径sch1顺序在前查看表数据。
gaussdb=# SET search_path = sch1,sch2,public;
SET
gaussdb=# SELECT * FROM tb_test;
c1

1
(1 row)

--设置搜索路径sch2顺序在前查看表数据。
gaussdb=# SET search_path = sch2,sch1,public;
SET
gaussdb=# SELECT * FROM tb_test;
c1

2
(1 row)

--重置search_path并删除模式和表。
gaussdb=# RESET search_path;
RESET
gaussdb=# DROP TABLE sch1.tb_test;
DROP TABLE
gaussdb=# DROP TABLE sch2.tb_test;
DROP TABLE
gaussdb=# DROP SCHEMA sch1;
DROP SCHEMA
gaussdb=# DROP SCHEMA sch2;
DROP SCHEMA
```

## 8.3 约束（Constraint）

SQL约束用于规定表中的数据规则。如果一个用户试图在表中插入违反约束的数据，会抛出错误。约束分为以下几类：

- 非空约束
- 唯一约束
- 主键约束
- 检查约束

### 非空约束（NOT NULL）

- 含有非空约束的字段不能包含NULL值。

- 只能创建字段级非空约束。
- 在同一个表中可以定义多个非空约束。

```
--创建表添加非空约束示例。
gaussdb=# CREATE TABLE tbl_person1(
 id int not null,
 name varchar(50)
);
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

--插入不符合约束的数据，触发约束报错。
gaussdb=# INSERT INTO tbl_person1 (id,name) VALUES (NULL,'Bob');
ERROR: dn_6001_6002_6003: The null value in column "id" violates the not-null constraint.
DETAIL: Failing row contains (null, Bob).

--删除表。
gaussdb=# DROP TABLE tbl_person1;
DROP TABLE
```

## 唯一约束（UNIQUE）

- 唯一约束的字段中不能包含重复值。
- 在同一个表中可以定义多个唯一约束。
- 在定义唯一约束后，数据库会自动为约束的字段添加一个唯一索引。
- 对同一个字段可以同时定义非空约束和唯一约束，如果字段只定义了唯一约束而没有定义非空约束，那么该字段可以包含多个NULL值。
- 一个表中可以定义多个唯一约束。

```
--创建表添加字段级唯一约束示例。
gaussdb=# CREATE TABLE tbl_person2(
 id int UNIQUE,
 name varchar(50)
);
NOTICE: CREATE TABLE / UNIQUE will create implicit index "tbl_person2_id_key" for table "tbl_person2"
CREATE TABLE

--创建表添加表级唯一约束示例。
gaussdb=# CREATE TABLE tbl_person3(
 id int,
 name varchar(50),
 CONSTRAINT uq_person3_id UNIQUE(id)
);
NOTICE: CREATE TABLE / UNIQUE will create implicit index "uq_person3_id" for table "tbl_person3"
CREATE TABLE

--插入重复的值触发唯一约束报错。
gaussdb=# INSERT INTO tbl_person2 VALUES (1,'Bob');
INSERT 0 1
gaussdb=# INSERT INTO tbl_person2 VALUES (1,'Bob');
ERROR: dn_6004_6005_6006: Duplicate key value violates unique constraint "tbl_person2_id_key".
DETAIL: Key (id)=(1) already exists.

--重复的NULL值不触发唯一约束。
gaussdb=# INSERT INTO tbl_person2 VALUES (null,'Bob');
INSERT 0 1
gaussdb=# INSERT INTO tbl_person2 VALUES (null,'Bob');
INSERT 0 1

--删除。
gaussdb=# DROP TABLE tbl_person2,tbl_person3;
DROP TABLE
```

## 主键约束（PRIMARY KEY）

- 主键约束的字段不能包含重复值和NULL值。
- 可以为一个或者多个字段定义主键约束。
- 创建主键约束后，数据库会自动为该字段创建一个唯一索引和非空约束。
- 一个表中只能定义一个主键约束。

```
--创建表添加字段级主键约束示例。
gaussdb=# CREATE TABLE tbl_person4(
 id int PRIMARY KEY,
 name varchar(50)
);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tbl_person4_pkey" for table "tbl_person4"
CREATE TABLE

--创建表添加表级主键约束示例。
gaussdb=# CREATE TABLE tbl_person5(
 id int,
 name varchar(50),
 CONSTRAINT pk_person5_id PRIMARY KEY(id)
);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "pk_person5_id" for table "tbl_person5"
CREATE TABLE

--删除表。
gaussdb=# DROP TABLE tbl_person4,tbl_person5;
DROP TABLE
```

## 检查约束（CHECK）

- 检查约束的的表达式必须引用表中的一个或多个字段，并且表达式的计算结果必须是一个布尔值，在表达式中不能包含子查询。
- 检查约束可以定义为字段级或者表级。
- 对同一个字段可以同时定义检查约束和非空约束。
- 对同一个字段可以同时定义多个检查约束。
- 同一个表中可以定义多个检查约束。

```
--创建表添加检查约束示例。
gaussdb=# CREATE TABLE tbl_person6(
 id int PRIMARY KEY,
 name varchar(50) NOT NULL,
 age int CHECK(age > 0 AND age < 200) NOT NULL
);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tbl_person6_pkey" for table "tbl_person6"
CREATE TABLE

--插入不符合约束的数据，触发约束报错。
gaussdb=# INSERT INTO tbl_person6 VALUES (1,'Bob',500);
ERROR: dn_6004_6005_6006: New row in relation "tbl_person6" violates check constraint "tbl_person6_age_check".
DETAIL: N/A

--删除表。
gaussdb=# DROP TABLE tbl_person6;
DROP TABLE
```

## 8.4 序列（Sequence）

### 简介

序列（Sequence）是一种数据库对象，用于生成数字队列。序列的值可以按照设定的增量递增或者递减。Sequence默认最大值为 $2^{63}-1$ ，如果使用了Large标识则最大值可以支持到 $2^{127}-1$ 。

### 示例

- 序列的创建和使用

```
--创建序列。
gaussdb=# CREATE SEQUENCE seq_test
 START WITH 1
 INCREMENT BY 1
 MAXVALUE 99999;

--查看序列下一个值。
gaussdb=# SELECT nextval('seq_test');
 nextval

 1
(1 row)

--删除序列。
gaussdb=# DROP SEQUENCE seq_test;
DROP SEQUENCE
```

- 序列空洞

序列设置缓存后虽然可以提升性能，但是也有可能产生空洞。

```
--创建表。
gaussdb=# CREATE TABLE tb_test(c1 int,c2 varchar);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--创建与表关联的序列。
gaussdb=# CREATE SEQUENCE seq_test
 START 1
 CACHE 10
 OWNED BY tb_test.c1;
CREATE SEQUENCE

--设置字段的默认值。
gaussdb=# ALTER TABLE tb_test ALTER COLUMN c1 SET DEFAULT nextval('seq_test'::regclass);
ALTER TABLE

--插入数据。
gaussdb=# INSERT INTO tb_test (c2) VALUES ('a');
INSERT 0 1
gaussdb=# INSERT INTO tb_test (c2) VALUES ('b');
INSERT 0 1

--退出数据库重新登录后再次插入数据。
gaussdb=# INSERT INTO tb_test (c2) VALUES ('c');
INSERT 0 1

--查询数据，序列出现了空洞。
gaussdb=# SELECT * FROM tb_test ORDER BY c1;
 c1 | c2
----+----
 1 | a
 2 | b
```

```
11 | c
(3 rows)

--删除表。
gaussdb=# DROP TABLE tb_test;
DROP TABLE
```

## 8.5 系统列

系统列是指数据库表中由数据库系统自动创建和维护的列，这些列用于存储表相关的元数据信息。系统列包含以下几种：

- **oid**：对象标识符，是数据库中每个表的唯一标识。
- **ctid**：行标识符，用于标识表中的行，每行都有一个唯一的ctid。
- **xmin**：事务标识符，表示表中插入该行的最小事务ID。
- **xmax**：事务标识符，表示表中删除或者更新该行的最大事务ID。
- **cmin**和**cmax**：命令标识符，分别表示插入和删除该行的命令ID。
- **xc\_node\_id**：节点标识符，用于标识数据所在的物理节点。
- **rowid**：全局唯一的逻辑行标识符，由slice\_id、bucket\_id、table\_oid和rowno动态计算生成。
- **rowno**：表内唯一的物理行序号，通过专属序列自动生成并持久化存储。

## 8.6 分区表

### 简介

数据库表分区是一种数据库优化的技术，可以将大表物理上分割成多个较小的部分以提高查询性能。每个部分称为一个分区，这些分区可以存储在不同的存储设备上。

- **优点：**
  - **提高查询性能**：通过减少扫描的数据量使查询性能有显著地提升。
  - **优化存储**：通过把不同分区存放不同的存储介质上，来平衡性能和成本。
  - **增加可维护性**：分区表的维护操作（数据清理、重建索引）可以从分区的粒度来进行，减少对整个系统的影响。
  - **提高并发性**：分区表可以提高并发性，因为多个分区可以并行处理。例如，多个查询可以同时访问不同的分区，而不会相互干扰。
- **缺点：**
  - **内存资源占用**：分区表使用内存大致为（分区数 \* 3 / 1024）MB，当分区数太多导致内存不足时，会间接导致性能急剧下降。
  - **分区策略复杂性**：制定和实施合适的分区策略需要技术知识和经验。如果分区策略选择不当，可能会导致数据分布不均衡，进一步影响性能。
  - **备份恢复的复杂性**：虽然可以单独备份和恢复分区，但这也意味着需要更细致的备份策略和管理工作。

### 注意事项

- **适合使用分区表的场景**

- 提高查询性能：表的数据量大，且具有某些特性的数据在其中某个场景中经常会用到，可以通过减少查询时扫描数据量来提高性能。如：经常以月、季度、年为单位做分析的表。
  - 平衡性能和成本：表的数据量过大，需要将冷数据（不常访问的数据）移动到低成本存储，而将热数据（频繁访问的数据）保留在高性能存储上。
  - 大数据量表管理：表的数据量过大，需要在多个存储介质上存储的场景。
- **分区键的选择：**  
选择分区表的分区键是一个重要的设计决策，因为它直接影响到数据库的性能、可维护性以及数据管理的效率。
    - 优化查询：选择常用查询的分区键。例如，经常根据日期进行查询的表，可以选择日期字段为分区键。
    - 数据分布：选择分区键时，考虑数据的分布情况。避免某些分区数据量过大而其他分区数据量过小的情况。
    - 分区数量和管理：控制分区数量，避免创建过多分区，分区过多会导致管理复杂度上升和性能下降。

## 分区表类型

GaussDB数据库支持的分区表为范围分区表、列表分区表和哈希分区表。

- 范围分区表：将数据基于范围映射到不同的分区。范围由创建分区表时指定的分区键决定。常用于时间序列数据，例如按日期、月份或年份进行分区。
- 列表分区表：根据分区键的值将数据分配到不同的分区，分区中包含的键值由创建分区表时指定。适用于分类明确的数据，例如订单状态、设备类型或地区代码。
- 哈希分区表：将数据根据哈希算法映射到不同分区中，分区个数在创建分区表时指定。适用于需要均匀分布数据以平衡负载的场景。

## 示例

```
--范围分区表示例1。
gaussdb=# CREATE TABLE test_range1(
 id INT,
 info VARCHAR(20)
) PARTITION BY RANGE (id) (
 PARTITION p1 VALUES LESS THAN (100),
 PARTITION p2 VALUES LESS THAN (200),
 PARTITION p3 VALUES LESS THAN (300),
 PARTITION pmax VALUES LESS THAN (MAXVALUE)
);
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

--范围分区表示例2。
gaussdb=# CREATE TABLE test_range2(
 id INT,
 info VARCHAR(20)
) PARTITION BY RANGE (id) (
 PARTITION p1 START(1) END(600) EVERY(200),
 PARTITION p2 START(600) END(800),
 PARTITION pmax START(800) END(MAXVALUE)
);
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

--列表分区表示例。
```

```
gaussdb=# CREATE TABLE test_list (NAME VARCHAR (50), area VARCHAR (50))
PARTITION BY LIST (area) (
 PARTITION p1 VALUES ('Beijing'),
 PARTITION p2 VALUES ('Shanghai'),
 PARTITION p3 VALUES ('Guangzhou'),
 PARTITION p4 VALUES ('Shenzhen'),
 PARTITION pdefault VALUES (DEFAULT)
);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'name' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

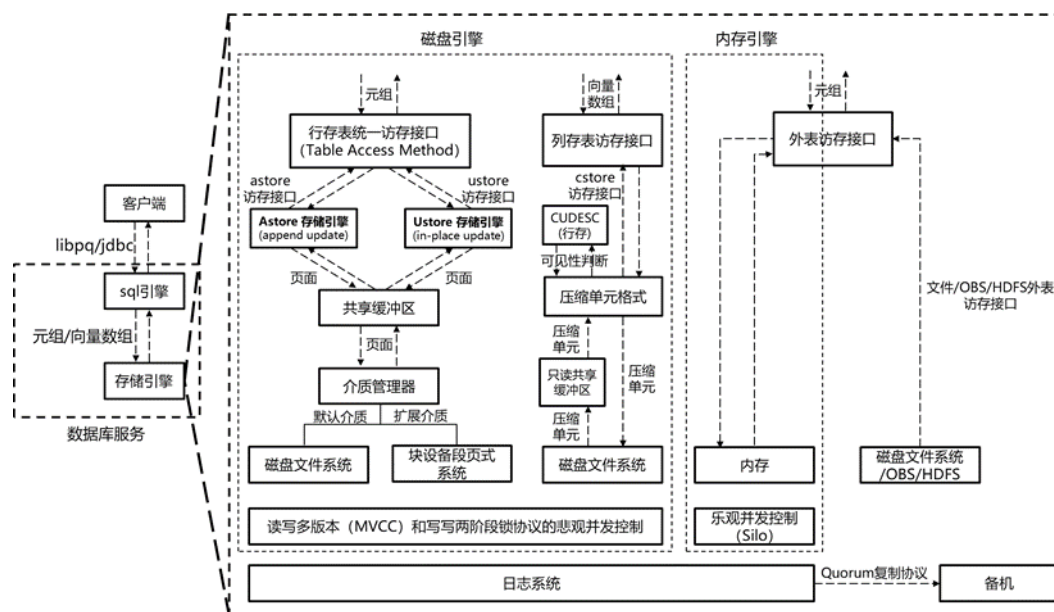
--哈希分区表示例1。
gaussdb=# CREATE TABLE test_hash1(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--哈希分区表示例2。
gaussdb=# CREATE TABLE test_hash2(c1 int) PARTITION BY HASH(C1)(
 PARTITION pa,
 PARTITION pb,
 PARTITION pc
);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--删除表。
gaussdb=# DROP TABLE test_range1;
DROP TABLE
gaussdb=# DROP TABLE test_range2;
DROP TABLE
gaussdb=# DROP TABLE test_list;
DROP TABLE
gaussdb=# DROP TABLE test_hash1;
DROP TABLE
gaussdb=# DROP TABLE test_hash2;
DROP TABLE
```

## 8.7 存储引擎

数据库存储引擎负责数据的存储、检索和管理。从整个数据库服务的组成架构来看，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），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，用于屏蔽底层行存引擎带来的差异，使不同行存引擎可以分别独立演进。如下图所示。



在此基础上，存储引擎通过日志系统提供数据的持久化和可靠性保障。通过并发控制（事务）系统保证并发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。

## 为什么需要存储引擎？

不同业务对数据写入和更新的诉求存在差异：有的以写入为主、更新较少，更关注写入吞吐与异常恢复的简单性；有的则存在大量更新，更关注高频更新下的空间膨胀控制与稳定的事务时延。同一种数据组织方式很难同时兼顾写入效率与更新时的空间膨胀控制，难以满足这些相互冲突的需求。

如果数据库只内置一种固定的存储方式，就只能在某一类场景下表现良好，难以适配多样化的业务。为此，GaussDB采用可插拔存储引擎架构：在统一的事务、日志、并发控制与缓存管理之上抽象出Table Access Method接口，将“如何存储数据”与“如何管理事务”解耦。用户可以按表选择Astore或Ustore存储引擎，从而在同一个数据库实例中，让不同业务特征的表各自使用最适合的存储引擎，兼顾性能、空间与可靠性。

## 存储引擎的分类

在GaussDB中，存储引擎可分为Ustore与Astore。Ustore为原位更新引擎，是GaussDB Kernel的默认存储引擎，新旧版本分离存储、空间膨胀可控并支持完整闪回，更适合高并发更新场景；Astore为追加更新引擎，写入高效、恢复简单。



## Ustore 存储引擎

Ustore采用In-place Update更新模式，即原地更新模式，是存储内核新增的一种存储模式。区别于Append Update（追加更新）模式，Ustore存储引擎在频繁UPDATE（页面内更新）的场景下有较好的表现。

In-place Update存储模式提供“原地更新”能力，主要思路是将最新版本的“有效数据”和历史版本的“垃圾数据”分离存储。将最新版本的“有效数据”存储在数据页面上，而单独开辟一段Undo（回滚）空间，用于统一管理历史版本的“垃圾数据”，因此数据空间不会由于频繁更新而膨胀，垃圾回收效率更高。通过NUMA-aware的Undo子系统设计，使得Undo子系统在多核平台上高效扩展。同时通过对元组和数据页面结构的重新设计，减少存储空间的占用。采用多版本索引技术，解决索引膨胀问题，去除对Autovacuum（异步垃圾清理线程）机制的依赖，提升存储空间的回收复用效率。

### Ustore 存储引擎的优势是什么？

- **空间膨胀可控：**最新版本与历史版本分离存储，历史版本集中由Undo空间管理并可批量回收，频繁更新下数据页与索引的空间膨胀得到有效抑制。
- **原位更新、CTID相对稳定：**非索引列/索引列更新堆表均可原地完成，元组CTID保持相对稳定；大并发更新同一行时先到先得，更新时延相对稳定。
- **扫描范围更小、回表更少：**借助多版本B-tree索引（携带事务信息、可独立进行MVCC）提升IndexOnlyScan比例，大幅减少回表次数。
- **自治式空间回收：**不依赖VACUUM进行旧版本清理，索引与堆表解耦、可独立清理，I/O更平稳。
- **企业级闪回能力：**结合Undo空间可实现更高效、更全面的闪回查询、闪回表等，支持快速回退误操作。

### Ustore 存储引擎的使用场景

- **高频更新的OLTP场景：**如订单、账务等存在大量UPDATE/DELETE的业务，原位更新结合Undo分离存储，可有效抑制数据页与索引的空间膨胀。
- **大并发更新同一行：**原位更新机制保证元组CTID相对稳定、先到先得，更新时延相对稳定。
- **需要完整闪回与回收站能力：**依托Undo空间可实现更高效、更全面的闪回查询、闪回表等企业级恢复功能。
- **对空间可控性要求高：**通过DML过程中的动态页面清理去除对VACUUM的依赖，自治式空间管理避免异步清理带来的I/O抖动。

### Ustore 存储引擎的工作原理

主要功能模块如下表所示：

| 模块          | 说明                                                                                               |
|-------------|--------------------------------------------------------------------------------------------------|
| Ustore表存取管理 | 向上对接SQL引擎，提供对Ustore表的行级查询、插入、删除、修改等操作接口，向下根据Ustore表页间、页内结构，以及Ustore表元组结构，完成对Ustore表文件的遍历和增删改查操作。 |

|              |                                                                             |
|--------------|-----------------------------------------------------------------------------|
| Ustore索引存取管理 | 向上对接SQL引擎，提供对索引表的行级查询、插入、删除等操作接口，向下根据索引表页间、页内结构，以及索引表元组结构，完成对指定索引键的查找和增删操作。 |
| Ustore表页面结构  | 包括Ustore表元组在页面内的具体组织形式，在页面内插入元组操作、页面整理操作、页面初始化操作等。                          |
| Ustore表元组结构  | 包括Ustore表元组的结构、填充、解构、修改、字段查询、变形等操作。                                         |
| Undo管理       | 包括Undo记录的结构、填充、编码、解码等操作。                                                    |
| 多版本索引        | 包括Ustore专用多版本索引UB-tree的页面结构、查询、修改、可见性检查、垃圾回收等模块。                            |

## 示例

- 创建一个Ustore存储引擎表，插入部分数据，查看元组位置信息。

```
-- 创建一个Ustore表
gaussdb=# CREATE TABLE tb_t1(id int, name VARCHAR(20)) WITH (STORAGE_TYPE=Ustore);
CREATE TABLE

-- 插入数据
gaussdb=# INSERT INTO tb_t1 VALUES (1,'Joe'),(2,'Jack');
INSERT 0 2

-- 查看元组的ctid信息
-- ctid (blkno, offset) 为元组在数据页上的位置，blkno为页面编号，offset是元组在页面内的偏移量。
gaussdb=# select *, ctid from tb_t1;
 id | name | ctid
-----+-----
 1 | Joe | (0,1)
 2 | Jack | (0,2)
(2 rows)
```
- 修改一条记录，查看元组位置信息，可以看出元组的ctid并未发生变化，即被原位更新了；更新前的旧数据会被存放至Undo空间。

```
-- 修改id为2的元组
gaussdb=# UPDATE tb_t1 SET name='Scott' WHERE id=2;
UPDATE 1

-- 再次查看ctid信息
gaussdb=# select *, ctid from tb_t1;
 id | name | ctid
-----+-----
 1 | Joe | (0,1)
 2 | Scott | (0,2)
(2 rows)
```

## 回滚段与 MVCC

- 回滚段

旧版本数据会集中在回滚段的Undo目录中，为了减少读写冲突，旧版本数据（回滚记录）采用追加写的方式写入数据目录的Undo目录下。这样旧版本数据的读取和写入不会发生冲突，同一个事务的旧版本数据也会连续存放，便于进行回滚操作。为了减少并发写入时的竞争，Undo目录空间被划分成多个逻辑区域（UndoZone，回滚段逻辑区域）。线程会在自己的逻辑区域上进行分配，与其他线程完全隔离，从而写入旧数据分配空间时就不会有额外的锁开销。UndoZone还可以按照CPU的NUMA核进行划分，每个线程会从当前的NUMA核上的

UndoZone进行分配，进一步提升分配效率。在分配Undo空间时会按照事务粒度进行记录，旧版本数据一旦确认没有事务进行访问，就会进行回收。

- **MVCC**

Ustore的可见性检查和Astore类似，将快照CSN和元组删除和插入事务的CSN进行比较，判断元组是否可见。Ustore和Astore使用同一套事务管理机制和快照管理机制。

Ustore和Astore最大的区别在于Astore会在页面上保留旧版本数据，而Ustore会将旧版本数据放到回滚段统一存放。在需要获取旧版本数据时，Astore可以直接从tuple的头部读取到元组的插入和删除的事务号(XID)，来判断元组的可见性。但是Ustore需要从回滚段里读取旧版本的事务信息，来判断旧版本是否可见。由于从回滚段中读取旧版本数据存在相对昂贵的开销，Ustore通过一系列的优化手段来避免从回滚段中读取旧版本数据。

Ustore在获取元组时，会先检查对应的事务目录。事务目录分为有效和无效两种。当事务目录是有效的，Ustore直接就会得到元组上最新的事务。

如果事务目录被冻结（FROZEN），意味着元组已经在所有的事务中都会可见。如果事务目录中的事务id小于oldestXidInUndo，意味着元组已经足够旧，在所有事务中都可见。同时会把事务目录置成冻结，来加速后续的查询。

如果元组被标记有一个无效事务目录，意味着修改元组的事务已经提交，并且比当前的事务目录中的事务旧。此时Ustore会使用事务目录中的事务进行可见性判断。如果可见，意味着修改元组的事务已经可见，就不需要从Undo目录中再读取事务信息。

## 空间管理和回收

不同于Astore的空间管理和回收机制，Ustore实现了自治式的空间管理机制。Ustore中的堆以及索引的空间分配和回收都在业务运行的过程中平稳地进行，不依赖VACUUM及Autovacuum清理机制。

- **自治式堆页面空间管理**

Ustore中堆页面的自治式空间管理，建立在与Astore类似的轻量级堆页面清理机制的基础上。在执行DML及DQL操作的过程中，Ustore都会进行堆数据页面清理，以取代VACUUM清理机制。Ustore会清理已经提交的被删除元组。

对于Astore而言，复用数据元组的行指针前必须保证对应的索引元组已经被清理。这是为了防止通过索引元组访问已经被复用的行指针，导致取到错误的数数据。在Astore中需要通过VACUUM操作将这样的无效索引元组统一清除掉后才能复用行指针，这使得堆页面和索引页面的清理逻辑耦合在一起，也会导致间断性的大量I/O。在Ustore中能高效地单独进行数据和索引页面的清理，因为带有版本信息的UB-tree能够独立检测并过滤掉无效的索引元组，不会通过无效索引元组访问对应的数据表。

堆页面的空间管理机制复用GaussDB中的FSM来管理UHeap中的可用空间。在成功对页面进行清理后，会将其空闲空间刷新到对应的FSM页面中。为了避免每次页面清理都需要更新整个树状结构的FSM，从而带来额外的开销，引入了一个更新整个FSM的概率计算。考虑当前清理后的可用空间占预留可用空间（Reserved Free Space）阈值的百分比，计算得出清理一个页面后更新FSM的概率。也就是说，页面清理获得的可用空间越大，更新整个FSM的概率也就越大。

当数据元组被删除时，会在页面上记录对应的潜在空闲空间（Potential Free Space），该值用于估计页面上的空闲空间。在运行过程中，有多个场景会对页面尝试进行清理。DML语句执行过程中，INSERT、UPDATE以及DELETE操作都会拿到页面的写锁。如果发现空间不足，或者检测到潜在空闲空间到达某个阈值，会尝试对页面进行清理。DQL查询语句执行的过程中若检测到页面上潜在空闲空间

到达阈值，也同样会尝试申请页面的写锁；如果拿到了页面的写锁，会尝试对页面进行清理。

由于当前的清理机制是基于访问进行清理的，存在部分页面有可清理的元组，但一直不被访问的页面，导致不能通过这一机制正确地清理。为了解决这一问题，引入了基于概率的清理方案。在一些寻找新的可用空间页面的操作中，若通过FSM发现没有足够的可用空间，在对物理文件进行扩展前，会“随机”选取一些页面进行清理，该清理算法能找出这些潜在需要清理的页面。

- **自治式索引页面空间管理**

索引页面的空间管理不依靠FSM数据结构，而是依靠特有的URQ（UB-tree Recycle Queue）结构，简称为回收队列。索引回收队列单独储存在UB-tree索引对应的.urq文件中，没有原有B-tree索引的.fsm文件。

索引中的回收队列分为两部分，一部分是潜在空页队列（Potential Empty Page Queue），一部分是可用页面队列（Available Page Queue）。两个队列都是跨页面的循环队列，其中每个元素都会储存blkno以及XID。其中blkno表示该元素对应索引页面的block number；XID表示该页面在何时能够被回收或复用。这些元素在循环队列单个页内按照XID的顺序进行排序，以便于快速找到XID小（最可能被回收或复用）的页面。

对于潜在空页队列而言，里面存放页内元组已经被全部删除但还没有全部无效的页面，其中的XID就标志页面中最后一个元组无效的可能时机。在系统整体的oldestXmin超过该XID后，该页面就有可能被从索引上删除，但也可能因为新插入元组或删除元组的事务中止而导致页面不能被删除。潜在空页队列中的页面在成功被删除后会被放入可用页面队列，并记录删除时最新事务的XID。

对于可用页面队列而言，里面存放已经被删除，可以或即将可以被复用的页面。其中XID就表示该页面可以被复用的时机。这样的页面复用时延是来自UB-tree索引页面删除时可能的并发访问导致的。

在业务运行的过程中，索引会不断尝试对潜在空页队列中的页面进行回收。在索引申请新的页面时，会查找当前可用的空闲页面。当可用页面队列中没有可用页面时，一般会通过扩展索引物理文件的方式来获得新的页面。但也存在物理文件批量扩展，或扩展后还未来得及使用就出错退出的情况。此时在回收队列的元信息页面中保存了已正确追踪的页面数量，若该数量少于整个索引表的页面数量，会尝试去使用这一部分未追踪的页面，并更新已追踪的页面数量。

## Astore 存储引擎

Astore（Append Store追加存储）是GaussDB存储引擎设计模型中的一种，数据在这种模型下是以追加的方式进行存储的。当数据更新时，会追加到现有数据的末尾，而不是覆盖旧的数据。

### Astore 存储引擎的优势是什么？

- **写入高效：**采用追加存储，数据直接追加到末尾、避免数据的移动与重建，写入性能高，适合频繁插入、少量更新的场景。
- **异常恢复简单：**没有回滚段，旧数据记录在原数据文件中，数据库异常crash后恢复无需像Ustore那样同步Redo和Undo，恢复流程更简单。
- **回滚快：**回滚并不删除数据，因此回滚可以很快完成。
- **WAL日志更简单：**仅需记录数据文件的变化，不需要记录回滚段的变化。

### Astore 存储引擎的使用场景

- **写入操作频率较多，更新较少的业务：**如日志、流水、归档等以INSERT为主、UPDATE/DELETE较少的场景，追加写可获得较高的写入吞吐。

- **历史数据留存：**旧版本数据不会被立即覆盖，适合需要保留历史记录、便于数据恢复与重建的场景。
- **对恢复简单性要求高：**希望异常恢复与回滚流程尽量简单、快速的场景。需要注意的是，Astore 在频繁更新时易出现空间膨胀，需配合VACUUM定期清理。

Astore 存储引擎的工作原理

整体框架

作为行存储子格式之一，Astore有自己的堆表元组结构、堆表页面结构、元组多版本机制，以及空闲空间管理和回收机制。

Astore以堆表形式存储数据，表中数据没有特定顺序，元组更新基于在同页面追加写实现，因此Astore堆表页面上新旧元组可以共存。

主要功能模块如下表所示：

| 模块               | 说明                                                                                                                                        |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Astore访存管理       | 提供Astore行存储格式表的具体访存操作实现，包括：对Astore堆表的行级查询、插入、删除、修改等操作接口；Astore堆表行级多版本机制和元组可见性判断；根据Astore堆表页间、页内结构，以及Astore堆表元组结构，完成对Astore堆表文件的遍历和增删改查操作。 |
| Astore索引访存管理     | 向上对接SQL引擎，提供对索引表的行级查询、插入、删除等操作接口，向下根据索引表页间、页内结构，以及索引表元组结构，完成对指定索引键的查找和增删操作。                                                               |
| Astore堆表/索引表页面结构 | 包括Astore堆表/索引表元组在页面内的具体组织形式，在页面内插入元组操作、页面整理操作、页面初始化、页面加解密、页面CRC（Cyclic Redundancy Check，循环冗余码校验）校验操作等。                                    |
| Astore堆表元组结构     | 包括Astore堆表元组的结构、填充、解构、修改、字段查询、变形、压缩、解压等操作。                                                                                                |
| 空间管控             | 结合FSM（Free Space Map），支持轻量级在线空间管控以及异步清理（Autovacuum）等空间管控机制。                                                                               |

示例

- 创建一个Astore存储引擎表，插入部分数据，查看元组位置信息。

```
-- 创建一个Astore表
gaussdb=# CREATE TABLE t1(id int, name VARCHAR(20)) WITH (STORAGE_TYPE=Astore);
CREATE TABLE
-- 插入数据
gaussdb=# INSERT INTO t1 VALUES (1,'Joe'),(2,'Jack');
INSERT 0 2
-- 查看元组的ctid信息
-- ctid (blkno, offset) 为元组在数据页上的位置，blkno为页面编号，offset是元组在页面内的偏移量。
gaussdb=# select *, ctid from t1;
 id | name | ctid
-----+-----
 1 | Joe | (0,1)
 2 | Jack | (0,2)
(2 rows)
```

- 修改一条记录，查看元组位置信息。

```
-- 修改id为2的元组
gaussdb=# UPDATE t1 SET name='Scott' WHERE id=2;
UPDATE 1
-- 再次查看ctid信息
gaussdb=# select *, ctid from t1;
 id | name | ctid
-----+-----
 1 | Joe | (0,1)
 2 | Scott | (0,3)
(2 rows)
```

可以看出被更新元组的ctid由(0,2)变为(0,3)，新元组追加到现有数据的末尾，而不是覆盖旧的数据，为非原位更新。旧版本数据依然存放在当前页面。

## MVCC

Astore支持单独的多版本元组并发控制机制，即为同一条记录保留多个历史版本的物理元组以解决对同一条记录的读、写并发冲突（读事务和写事务工作在不同版本的物理元组上）。

Astore存储格式为追加写优化设计，其元组历史版本与当前版本都存储在页面中。当一个更新操作将v0版本元组更新为v1版本元组之后，如果v0版本元组所在页面仍然有空闲空间，则直接在该页面内插入更新后的v1版本元组，并将v0版本的元组指针指向v1版本的元组指针。在这个过程中，新版本元组以追加写的方式和被更新的旧版本元组混合存放，这样可以减少更新操作的I/O开销。然而，需要指出的是，由于新、老版本元组是混合存放的，因此在清理旧版本元组时需要的清理开销会比较大。因此，Astore存储格式比较适合频繁插入、少量更新的业务场景。

Astore的元组中会存储事务信息（xmin，xmax等），xmin表示元组被插入时的事务信息，xmax表示元组被删除或更新时的事务信息，在默认隔离级别下（读已提交），只需结合CSN（Commit Sequence Number，事务提交序列号）判断元组上的事务信息是否可见即可判断元组是否可见，例如，将元组从v0版本更新到v1版本，并发事务中有快照查询到该元组，元组v0可见的条件是：xmin字段对应的CSN值小于等于查询快照的CSN，且xmax对应的CSN值大于读查询快照的CSN；元组v1可见的条件是：xmin字段对应的CSN值小于等于读查询快照的CSN（v1元组的xmax字段为0，无需判断）。当v0元组不可见时，可以通过v0元组上记录的ctid找到v1元组，继续判断其可见性。

## 空间管理和回收

Astore中采用最大堆二叉树结构来记录和管理堆表页面的空闲空间，该最大堆二叉树结构按照页面粒度进行与存储介质的读写操作，并单独储存于专门的空闲空间位图文件中（Free Space Map，简称FSM）。

所有页面分为叶子节点页面和内部节点页面两种。两种页面的页面内部结构完全相同，区别在于：对于叶子节点页面，其页面中记录的二叉树的叶子节点对应堆/索引表页面的空闲空间程度；对于内部节点页面，其页面中记录的二叉树的叶子节点对应下层FSM页面的最大空闲空间程度。

使用FSM页面中的1个字节（即256档）来记录一个堆/索引页面的空闲空间程度。在FSM页面中不会记录任何堆/索引页面的页号信息，也不会记录任何根、子FSM节点页面的页号信息，这些信息主要通过以下的规则来计算得到：

1. 在一个FSM页面内部，二叉树节点按照从上到下、从左到右逐层排布，即：第一个字节为根节点的空闲程度，第二个字节为第一层内部节点最左边节点的空闲程度，依次类推。

2. 所有FSM页面在物理存储上采用深度优先顺序，即某个FSM页面之前所有的物理页面包括：该FSM页面所在子树的所有上层节点，加上该FSM页面所有左侧子树。

3.所有FSM叶子节点页面中的二叉树的叶子节点，对应堆/索引表页面的空闲空间程度，且根据从左到右的顺序，分别对应第1个、第2个、…、第n个堆/索引表物理页面。

4.除了3中这些FSM节点之外，其他FSM父节点保存子节点（子树）中空闲空间的最大值。根据上述算法，可以高效地查询出具有足够空闲空间的堆/索引页面的页面号，并将待插入的数据插入其中。

空闲空间的管理难点在于空闲空间的回收。在GaussDB中，对于Astore存储格式，有3种回收空闲空间的方式。

- **轻量级堆页面清理**

当查询扫描到某个Astore堆表页面时，会借机尝试清理该页面上已经被删除的旧元组。由于只是顺带清理该页面内容，因此只能删除元组内容本身，元组指针还需要保留，以免在索引侧造成空引用或空指针。一个比较特殊的情况是HOT场景。HOT场景是指对于该表上所有的索引更新前后的索引键值均没有发生变化，因此对于更新后的元组只需要插入堆表元组而不需要新插入索引元组。对于同一个页面内一条HOT链上的多个元组，如果它们都足够旧了，那么在清理时可以额外删除所有中间的元组指针，只保留第一个版本的元组指针，并将其重定向到第一个不用被清理的元组版本的元组指针。

- **中量级堆页面和索引页面清理**

GaussDB提供VACUUM语句来让用户主动执行对某个Astore表（或某个库中所有的Astore表）及其上的索引进行中量级清理。中量级清理过程中，不阻塞相关表的查询和DML操作。由于在Astore表中，新、旧版本元组是混合存储的，因此，与顺带执行的轻量级清理相比，Astore表的中量级清理需要进行全表顺序（或索引）扫描，才能识别出所有待清理的旧版本元组。对于扫描出来的确认要清理的元组，会首先清理索引中的元组，然后再清理堆表中的元组，从而可以避免出现索引空指针的问题。

- **重量级堆页面和索引页面清理**

无论是轻量级清理，或是中量级清理，都只能局部清理Astore页面中的死亡元组，无法真正实现对这些空闲空间的释放（被清理出的空间，仍然只能被该表使用）。因此，GaussDB还提供了VACUUM FULL语句来让用户主动执行对某个Astore表（或某个库中所有Astore表）及其上的索引进行重量级清理。重量级清理将一个表中所有仍未死亡（但是可能已经被删除）的元组重新紧密插入到新的堆表文件中并在此基础上重新创建所有索引，从而实现对空闲空间的彻底回收。在重量级清理的主体流程中只允许用户执行只读查询操作，在重量级清理的提交流程中只读查询操作也会被阻塞。为了尽可能提高重新创建的索引性能，如果用户堆表上有索引，那么上述全表扫描会采用索引扫描。

## Astore 与 Ustore 的区别

Ustore与Astore都是GaussDB支持的存储引擎，二者的共同点在于：都基于统一的事务、日志与并发控制机制，都用于承载行式组织的业务数据，并都支持回收站（闪回DROP/TRUNCATE）功能。

二者的核心差异在于多版本的实现方式：Ustore采用原位更新，最新版本与历史版本分离存储，并依赖Undo回滚段管理历史版本；Astore采用追加存储，最新版本与历史版本不分离、无回滚段。这一差异主要源于设计目标不同——Ustore追求更新场景下的空间可控与功能丰富，Astore追求简单可靠的恢复与高效写入。

因此，Ustore更适合高频更新的OLTP场景，空间膨胀可控并支持完整闪回，但更新与单条扫描会带来略高的开销；Astore异常恢复流程更简单、回滚更快，但频繁更新时易出现空间膨胀。

| 对比维度  | Ustore（原位更新）                                 | Astore（追加更新）        |
|-------|----------------------------------------------|---------------------|
| 共同点   | 均为行存引擎，基于统一事务/日志/并发控制，均支持回收站闪回DROP/TRUNCATE。 |                     |
| 多版本实现 | 原位更新，新旧版本分离存储。                               | 追加存储，新旧版本不分离。       |
| 回滚段   | 有Undo回滚段。                                    | 无回滚段。               |
| 空间管理  | 旧版本可批量回收，膨胀可控。                               | 频繁更新易空间膨胀。          |
| 异常恢复  | 需Redo+Undo，恢复较复杂。                            | 流程简单，回滚快。           |
| 闪回能力  | 支持闪回查询/表/DROP/TRUNCATE等。                     | 仅支持闪回DROP/TRUNCATE。 |

## 与存储引擎相关的特性和操作

在了解存储引擎的核心概念与分类后，如需了解更多细节以及在实际业务中配置使用，可参考以下对应章节：

- 设置存储引擎：建表时通过`WITH (STORAGE\_TYPE)`指定引擎类型，并可通过GUC参数`enable\_default\_ustore\_table`控制行存表默认引擎，详情请参考[设置存储引擎](#)章节。
- 查询表的存储方式：查看现有表所使用的存储引擎，详情请参考[如何查询表的存储方式？](#) 章节。
- Ustore使用与最佳实践：了解Ustore的特性规格、测试方法与最佳实践，详情请参考[Ustore存储引擎](#)章节。

## 8.8 索引

### 简介

索引是数据库中用于加速数据检索的一种数据结构，通过创建指向表中数据的指针，使得数据库可以快速定位和访问特定的行，而无需扫描整个表。索引类似于书籍的目录，帮助用户快速定位所需要的内容，从而显著提高查询性能。虽然索引可以提高数据访问速度，但同时也增加了插入、更新、和删除操作的处理时间。所以是否要为表创建索引、索引建立在哪些字段上，是创建索引前必须要考虑的问题。

### 索引类型

- Btree：使用一种类似于B+tree的结构来存储数据的键值，叶节点按键值顺序存储数据。适用于等值查询、范围查询、排序和分组的操作。
- UBtree：仅供Ustore表使用的多版本Btree索引，索引页面上包含事务信息，并能自主回收页面。Ubtree索引默认开启insertpt功能。
- GIN：通用倒排索引，可用于处理索引项为组合值的情况，如数组或者JSONB。
- UGIN：通用倒排索引，一种仅供Ustore表使用的GIN索引。
- GIST：适用于几何和地理等多维数据类型和集合数据类型。目前支持的数据类型包括box、point、poly、circle、tsvector、tsquery和range。



## 示例

- 单列索引

基于单个列创建的索引，适用于简单的查询条件。

```
--创建表。
gaussdb=# CREATE TABLE tb_t1(c1 int, c2 int);

--创建索引。
gaussdb=# CREATE INDEX idx_t1_c1 ON tb_t1(c1);

--删除表。
gaussdb=# DROP TABLE tb_t1;
```

- 多列索引

基于多个列的组合索引，适用于多条件查询。

```
--创建表。
gaussdb=# CREATE TABLE tb_t2(c1 int, c2 int, c3 int);

--创建索引。
gaussdb=# CREATE INDEX idx_t2_c1_c2 ON tb_t2(c1,c2);

--删除表。
gaussdb=# DROP TABLE tb_t2;
```

- 唯一索引

确保索引列唯一，同时允许NULL值。为表添加唯一约束时，数据库自动为表添加唯一索引。

```
--创建表。
gaussdb=# CREATE TABLE tb_userinfo(uid char(5), name varchar(20));

--创建唯一索引。
gaussdb=# CREATE UNIQUE INDEX idx_userinfo_uid ON tb_userinfo(uid);

--删除表。
gaussdb=# DROP TABLE tb_userinfo;
```

- 表达式索引

表达式索引是一种基于计算表达式或函数结果创建的索引，而非直接基于表中的原始列。它允许对一些计算或转换后的值进行快速检索，适用于查询条件中包含函数或表达式的情况。

```
--创建表及插入数据。
gaussdb=# CREATE TABLE tb_test(c1 varchar,c2 varchar);
gaussdb=# INSERT INTO tb_test VALUES ('AAa','AAA12');
gaussdb=# INSERT INTO tb_test VALUES ('ABa','AAA13');
gaussdb=# INSERT INTO tb_test VALUES ('AAc','AAA14');

--创建索引。
gaussdb=# CREATE INDEX idx_test_c1_lower ON tb_test(lower(c1));

--查询。
gaussdb=# SELECT * FROM tb_test WHERE lower(c1) = 'aac';

--删除表。
gaussdb=# DROP TABLE tb_test;
```

- 部分索引

部分索引仅对表中满足特定条件的数据建立索引，从而减少索引体积，提升写入的性能。

```
--创建表及插入数据。
gaussdb=# CREATE TABLE tb_users(uid varchar, name varchar, is_active boolean);
gaussdb=# INSERT INTO tb_users VALUES('bc'||generate_series(1,100),'uname'||
generate_series(1,100),true);
gaussdb=# INSERT INTO tb_users VALUES('ac'||generate_series(1,50),'uname'||
generate_series(1,50),false);
```

```
--创建索引。
gaussdb=# CREATE INDEX idx_users_uid_act ON tb_users(uid) WHERE is_active = true;

--查询时自动利用部分索引。
gaussdb=# SELECT * FROM tb_users WHERE uid = 'bc001' AND is_active = true;

--删除表。
gaussdb=# DROP TABLE tb_users;
```

- 分区索引

```
--创建表。
gaussdb=# CREATE TABLE student(id int, name varchar(20)) PARTITION BY RANGE (id) (
 PARTITION p1 VALUES LESS THAN (200),
 PARTITION pmax VALUES LESS THAN (MAXVALUE)
);

--创建LOCAL分区索引不指定索引分区的名称。
gaussdb=# CREATE INDEX idx_student1 ON student(id) LOCAL;

--查看索引分区信息，发现LOCAL索引分区数和表的分区数一致。
gaussdb=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student1'::regclass;
 relname

p1_id_idx
pmax_id_idx
(2 rows)

--删除LOCAL分区索引。
gaussdb=# DROP INDEX idx_student1;

--创建GLOBAL索引。
gaussdb=# CREATE INDEX idx_student2 ON student(id) GLOBAL;

--查看索引分区信息，发现GLOBAL索引分区数和表的分区数不一致。
gaussdb=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student2'::regclass;
 relname

(0 rows)

--删除GLOBAL索引。
gaussdb=# DROP INDEX idx_student2;

--创建LOCAL表达式索引，不指定索引分区的名称。
gaussdb=# CREATE INDEX idx_student3 ON student(abs(id)) LOCAL;

--查看索引分区信息，发现LOCAL索引分区数和表的分区数一致。
gaussdb=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student3'::regclass;
 relname

p1_abs_idx
pmax_abs_idx
(2 rows)

--删除LOCAL分区表达式索引。
gaussdb=# DROP INDEX idx_student3;

--创建GLOBAL表达式索引。
gaussdb=# CREATE INDEX idx_student4 ON student(abs(id)) GLOBAL;

--查看索引分区信息，GLOBAL表达式索引分区数和表的分区数不一致。
gaussdb=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student4'::regclass;
 relname

(0 rows)

--删除GLOBAL表达式索引。
gaussdb=# DROP INDEX idx_student4;

--删除表。
gaussdb=# DROP TABLE student;
```

- UGIN索引

```
--创建表。
gaussdb=# CREATE TABLE test_trgm(t varchar(100)) WITH (STORAGE_TYPE=USTORE);

--插入数据。
gaussdb=# INSERT INTO test_trgm VALUES('qwertyu0001');
gaussdb=# INSERT INTO test_trgm VALUES('qwertyu0099');
gaussdb=# INSERT INTO test_trgm VALUES('2024新年快乐7days');

--创建UGIN索引。
gaussdb=# create index trgm_idx on test_trgm using ugin (t ugin_trgm_ops);

--LIKE查询语句。
gaussdb=# select * from test_trgm where t like '%24新%';

--删除。
DROP TABLE test_trgm;
```

- GIN索引

```
--创建表。
gaussdb=# CREATE TABLE test_gin (id INT, info INT[]) WITH(STORAGE_TYPE = ASTORE);

--创建索引。
gaussdb=# CREATE INDEX test_gin_idx ON test_gin USING GIN(info);

--插入数据。
gaussdb=# INSERT INTO test_gin SELECT g, ARRAY[1, g % 5, g] FROM generate_series(1, 20000) g;

--查询语句。
gaussdb=# SELECT * FROM test_gin WHERE info @> '{2}' AND info @> '{22}' ORDER BY id, info;

--删除表。
gaussdb=# DROP TABLE test_gin;
```

- GiST索引

```
--创建表。
gaussdb=# CREATE TABLE gist_box_tbl(f1 point, f2 int)WITH(storage_type = astore);

--创建索引。
gaussdb=# CREATE INDEX gist_box_index ON gist_box_tbl USING gist(f1);

--插入数据。
gaussdb=# INSERT INTO gist_box_tbl VALUES(point(1,2),1);
gaussdb=# INSERT INTO gist_box_tbl VALUES(point(3,4),3);

--查询语句。
gaussdb=# SELECT count(*) FROM gist_box_tbl WHERE f1 ~= point(1, 2);

--删除表。
gaussdb=# DROP TABLE gist_box_tbl;
```

## 8.9 物化视图

### 物化视图简介

物化视图是一种特殊的物理表，与视图的区别是物化视图存储了查询结果，而不是查询的定义。相比视图物化视图在查询时有更高的速度，但物化视图无法实时更新，对于需要实时响应的业务不够灵活。物化视图又分为增量物化视图和全量物化视图。

- 全量物化视图：全量物化视图需要刷新时，会对视图进行全量更新，即重新执行查询并将结果替换到当前视图中。适用于数据更新频率较低，查询成本较高的场景，因为全量物化视图刷新成本较高，不适合频繁更新的场景。
- 增量物化视图：增量物化视图允许对视图进行增量更新，即只更新上次刷新后发生变化的数据。适用于刷新频率较高的场景。虽然增量物化视图刷新成本较低，

但是支持的场景较少（例如增量物化视图目前不支持聚合函数）目前仅支持简单过滤查询和基表UNION ALL。

## 示例

- 全量物化视图

```
--建表及插入数据(全量物化视图只支持基表是Astore)。
gaussdb=# CREATE TABLE tb_test(pid varchar(5),snum varchar(5)) WITH (storage_type = astore);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'pid' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# INSERT INTO tb_test VALUES ('00001','aaaaa');
INSERT 0 1
gaussdb=# INSERT INTO tb_test VALUES ('00001','aacba');
INSERT 0 1
gaussdb=# INSERT INTO tb_test VALUES ('00002','badbc');
INSERT 0 1

--创建全量物化视图。
gaussdb=# CREATE MATERIALIZED VIEW mv_test AS SELECT pid,count(snum) cnt FROM tb_test
GROUP BY pid;
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'pid' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
INSERT 0 2

--查询全量物化视图。
gaussdb=# SELECT * FROM mv_test;
 pid | cnt
-----+-----
00002 | 1
00001 | 2
(2 rows)

--插入数据后继续查询全量物化视图。
gaussdb=# INSERT INTO tb_test VALUES ('00002','bdacc');
INSERT 0 1
--查询结果没有变化，物化视图需要手动刷新。
gaussdb=# SELECT * FROM mv_test;
 pid | cnt
-----+-----
00002 | 1
00001 | 2
(2 rows)

--刷新全量物化视图。
gaussdb=# REFRESH MATERIALIZED VIEW mv_test;
INSERT 0 2

--查询全量物化视图。
gaussdb=# SELECT * FROM mv_test;
 pid | cnt
-----+-----
00002 | 2
00001 | 2
(2 rows)

--删除表和物化视图。
gaussdb=# DROP MATERIALIZED VIEW mv_test;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE tb_test;
DROP TABLE
```

- 增量物化视图

```
--创建表。
gaussdb=# CREATE TABLE tb_test(c1 int) WITH (storage_type = astore);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
```

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

--增量物化视图中包含聚合函数报错。
gaussdb=# CREATE INCREMENTAL MATERIALIZED VIEW imv_test AS SELECT count(*) FROM tb_test;
ERROR: Feature not supported
DETAIL: aggregates on incremental materialized view creation

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

--增量刷新增量物化视图。
gaussdb=# REFRESH INCREMENTAL MATERIALIZED VIEW imv_test;
REFRESH MATERIALIZED VIEW

--全量刷新增量物化视图。
gaussdb=# REFRESH MATERIALIZED VIEW imv_test;
REFRESH MATERIALIZED VIEW

--删除表和物化视图。
gaussdb=# DROP MATERIALIZED VIEW imv_test;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE tb_test;
DROP TABLE
```

## 8.10 事务

事务（Transaction）是数据库管理系统执行过程中的最小逻辑单位，由一个有限的数据库操作序列构成（通常由事务开始与事务结束之间执行的全部数据库操作组成）。这些操作要么全部成功执行，要么全部不执行，是一个不可分割的执行单位。

当事务提交时，数据库管理系统必须确保事务中的所有操作都成功完成。如果事务中的任何一部分操作未成功，系统会将数据库回滚到事务开始之前的状态。以下是事务的四个特性（ACID）：

- 原子性（Atomicity）整个事务中的所有操作，要么全部完成，要么全部不完成，不可停滞在某个中间环节。
- 一致性（Consistency）事务必须始终保持系统处于一致的状态。
- 隔离性（Isolation）隔离性确保并发执行的事务彼此隔离，不会互相干扰。
- 持久性（Durability）一旦事务提交，对数据库的改变应该永久保存在数据库中。

### 事务隔离级别

- 脏读、幻读、不可重复读介绍：
  - 脏读：当前事务能读取到另一个事务尚未提交的数据。
  - 幻读：在同一个事务中，多次执行相同的查询时，由于其他事务的插入和删除操作，导致结果集中的行数发生变化。
  - 不可重复读：在同一个事务中，多次读取同一数据时，由于其他事务的修改操作，导致读取的结果不一致。
- GaussDB隔离级别的特点：
  - Read Committed（读已提交）：默认的隔离级别。当前事务只能读取到其他事务已提交的数据，从而避免了脏读。但是其他事务可以在两次读取期间修改或者插入数据导致出现幻读和不可重复读的情况。
  - Repeatable Read（可重复读）：当前事务在执行期间可以看到一致的数据，避免了脏读和不可重复读。

- Serializable（可串行化）：GaussDB目前功能上不支持此隔离级别，设置该隔离级别时，等价于REPEATABLE READ。

## 事务控制语句

- START TRANSACTION：开启事务，如果声明了隔离级别、读写模式，那么新事务就使用这些特性。此语句在使用过程中可选。
- BEGIN：开启事务，如果声明了隔离级别、读写模式，那么新事务就使用这些特性。此语句在使用过程中可选。
  - 会话的系统变量autocommit的值为off时，表示关闭事务自动提交功能，就不需要BEGIN或者START TRANSACTION命令来标识多个SQL组成的一个事务。
  - 会话的系统变量autocommit的值为on时，表示开启事务自动提交功能，该模式下，每条SQL都是一个独立的事务。如果还要多个SQL组成一个事务，就需要BEGIN或者START TRANSACTION命令开启一个事务。
- SAVEPOINT：用于在当前事务里新建一个保存点。保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。
- COMMIT|END：提交并结束当前事务，让所有的修改持久化并生效，清除所有保存点释放事务锁。
- ROLLBACK：回滚整个事务已做的修改，清除所有保存点释放事务锁。
- ROLLBACK TO SAVEPOINT：回滚到一个保存点，隐含地删除所有在该保存点之后建立的保存点。

## 语法格式

- START TRANSACTION

```
START TRANSACTION
[
{
ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE
READ }
| { READ WRITE | READ ONLY }
} [, ...]
];
```
- BEGIN

```
BEGIN [WORK | TRANSACTION]
[
{
ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE
READ }
| { READ WRITE | READ ONLY }
} [, ...]
];
```
- SAVEPOINT

```
SAVEPOINT savepoint_name;
```
- ROLLBACK TO SAVEPOINT

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name;
```

## 示例

```
--建表。
gaussdb=# CREATE TABLE test1(c1 INT);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
```

```
CREATE TABLE

--以隔离级别为READ COMMITTED，读/写方式启动事务。
gaussdb=# START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
START TRANSACTION

--插入数据。
gaussdb=# INSERT INTO test1 (c1) VALUES (1);
INSERT 0 1

--建立一个保存点。
gaussdb=# SAVEPOINT my_savepoint;
SAVEPOINT

--插入数据。
gaussdb=# INSERT INTO test1 (c1) VALUES (2);
INSERT 0 1

--回滚到保存点。
gaussdb=# ROLLBACK TO SAVEPOINT my_savepoint;
ROLLBACK

--查看数据，可以看到数据已经回滚到保存点之前的状态。
gaussdb=# SELECT * FROM test1;
 c1

 1
(1 row)
--由于隔离级别是READ COMMITTED且事务还未结束，所以新会话查看该表将没有任何数据。
gaussdb=# SELECT * FROM test1;
 c1

(0 rows)

--插入数据。
gaussdb=# INSERT INTO test1 (c1) VALUES (3);
INSERT 0 1

--结束事务。
gaussdb=# COMMIT;
COMMIT

--查看数据，事务已经结束，在另一个会话中可以查询到数据。
gaussdb=# SELECT * FROM test1;
 c1

 3
 1
(2 rows)

--删除表。
gaussdb=# DROP TABLE test1;
DROP TABLE
```

## 8.11 锁

### 锁简介

数据库通过锁对公共资源进行并发控制，使用锁的一般流程操作可以简述为3步：加锁、临界区操作、释放锁。

当对表进行DDL/DML操作时，数据库会对表进行加锁操作，在事务结束时释放。GaussDB提供了8个级别的锁分别用于不同语句的并发，各操作对应的锁如下表所示：

表 8-1 锁

| 锁级别 | 名称                       | 用途                                     | 冲突关系            |
|-----|--------------------------|----------------------------------------|-----------------|
| 1   | AccessShareLock          | SELECT                                 | 8               |
| 2   | RowShareLock             | SELECT FOR UPDATE/FOR SHARE            | 7,8             |
| 3   | RowExclusiveLock         | INSERT/UPDATE/DELETE                   | 5,6,7,8         |
| 4   | ShareUpdateExclusiveLock | VACUUM                                 | 4,5,6,7,8       |
| 5   | ShareLock                | CREATE INDEX                           | 3,4,6,7,8       |
| 6   | ShareRowExclusiveLock    | ROW SELECT ... FOR UPDATE              | 3,4,5,6,7,8     |
| 7   | ExclusiveLock            | BLOCK ROW SHARE/ SELECT ... FOR UPDATE | 2,3,4,5,6,7,8   |
| 8   | AccessExclusiveLock      | DROP CLASS/VACUUM FULL                 | 1,2,3,4,5,6,7,8 |

当两个事务的锁产生冲突时，未获取到锁的线程会等待锁。如果等待时间超过了系统设置的参数lockwait\_timeout（默认20分钟），则会发生锁等待超时。其他与锁相关的参数见《参考》中“数据库运行参数说明 > GUC参数说明 > 锁管理”。

**锁示例：**

```
--建表并插入数据。
gaussdb=# CREATE TABLE test1(c1 INT, c2 VARCHAR(5));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# INSERT INTO test1 VALUES (1,'a'),(2,'b'),(3,'c');
INSERT 0 3

--查看会话参数。
gaussdb=# SHOW lockwait_timeout;
lockwait_timeout

20min
(1 row)

--在第一个会话中执行。
gaussdb=# BEGIN;
BEGIN
gaussdb=# INSERT INTO test1 VALUES (4,'d');
INSERT 0 1

--在第二个会话中执行，该SQL会一直等到第一个会话结束才开始执行。
gaussdb=# CREATE INDEX idx_test1_c1 ON test1(c1);

--第一个会话中结束事务。
```



```
gaussdb=# END;
COMMIT

--删除表。
gaussdb=# DROP TABLE test1;
DROP TABLE
```

## 查询锁等待

```
--建表并插入数据。
gaussdb=# CREATE TABLE test2(id INT,info VARCHAR(10));
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
gaussdb=# INSERT INTO test2 VALUES (1,'info1'),(2,'info2');
INSERT 0 2
```

```
--第一个会话中执行。
gaussdb=# BEGIN;
BEGIN
gaussdb=# CREATE INDEX idx_test2_id ON test2(id);
CREATE INDEX
```

```
--第二个会话中执行。
gaussdb=# INSERT INTO test2 VALUES (3,'info3');
```

- 打开一个新的会话，通过如下SQL查询出等待执行的SQL。

```
gaussdb=# SELECT datname,pid,query,query_id,waiting FROM pg_stat_activity WHERE waiting = TRUE;
datname | pid | query | query_id | waiting
-----+-----+-----+-----+-----
postgres | 140389444876032 | INSERT INTO test2 VALUES (3,'info3'); | 3940649678789642 | t
(1 row)
```

- 通过pid查询等待事件。

```
gaussdb=# SELECT node_name,db_name,query_id,tid,wait_status,wait_event FROM
pg_thread_wait_status WHERE tid = 140389444876032 AND db_name IS NOT NULL;
node_name | db_name | query_id | tid | wait_status | wait_event
-----+-----+-----+-----+-----+-----
dn_6001 | postgres | 3940649678789642 | 140389444876032 | acquire lock | relation
(1 row)
```

- 通过如下SQL查询出持锁SQL的pid。

```
--通过如下sql查询出持锁语句关联的pid， locktype和上面查询出的wait_event保持一致。
gaussdb=# SELECT relation,pid,mode,granted FROM pg_locks WHERE pid = 140389444876032 AND
locktype='relation';
```

```
relation | pid | mode | granted
-----+-----+-----+-----
26466 | 140389444876032 | RowExclusiveLock | f
(1 row)
```

```
-- 通过如下语句查询出持锁语句的pid， granted字段为t表示持锁语句。
```

```
gaussdb=# SELECT relation,pid,mode,granted FROM pg_locks WHERE relation = 26466;
relation | pid | mode | granted
-----+-----+-----+-----
26466 | 140389444876032 | RowExclusiveLock | f
26466 | 140389939934976 | AccessShareLock | t
26466 | 140389939934976 | ShareLock | t
(3 rows)
```

- 通过如下SQL可以查询出持锁语句的详细信息。

如下信息显示该语句状态为"idle in transaction" 表示事务待提交。手动提交事务后，待执行的INSERT语句才会开始执行。

```
gaussdb=# SELECT datname,pid,query,query_id,waiting,state FROM pg_stat_activity WHERE pid =
140389939934976;
datname | pid | query | query_id | waiting | state
-----+-----+-----+-----+-----+-----
postgres | 140389939934976 | CREATE INDEX idx_test2_id ON test2(id); | 0 | f | idle in
transaction
(1 row)
```

根据实际情况决定是否通过pg\_terminate\_backend函数去对应的节点上结束持锁SQL的线程。

示例：

```
gaussdb=# SELECT pg_terminate_backend(140389939934976);
pg_terminate_backend

t
(1 row)

--删除表。
gaussdb=# DROP TABLE testl2;
DROP TABLE
```

## 8.12 PL/SQL

### 8.12.1 PL/SQL 简介

PL/SQL 提供了一种服务器端的存储过程语言，它易于使用，具备安全性且可移植。通过该语言，可以创建对象以访问和操作数据库数据。

#### PL/SQL 子程序

PL/SQL子程序是一个命名的PL/SQL块，它允许调用方提供只输入、只输出或者输入和输出的参数。子程序主要包括四种类型：存储过程、函数、包和触发器。

- **子程序创建**

子程序结构如下：

- 声明部分（可选）  
声明部分包括类型、变量、常量、显式游标和嵌套子程序的声明。这些项对于子程序都是本地的，当子程序执行结束后就不存在了。
- 执行部分（可选）  
执行部分包括赋值语句、控制执行语句和数据库操作语句。
- 异常处理部分（可选）  
异常处理部分包括处理异常的代码。

在子程序中添加注释会增加子程序的可读性，单行注释是以双减号开头的“--”，注释范围到行尾结束，多行注释是以“/\*”开头，以“\*/”结尾。

- **子程序执行**

可以通过如下三种方式执行子程序：

- 在应用程序中直接调用子程序。
- 在子程序或触发器中调用。
- 使用客户端工具。

### 8.12.2 游标（Cursor）

#### 简介

游标（Cursor）用于处理SQL影响的结果集，允许逐行地访问和操作这些结果集。它是指向查询结果集的一个指针，可以对每一行数据进行单独的处理。游标主要分为显示游标和隐式游标。

## 游标的属性

%FOUND：用于测试对游标进行FETCH的操作是否检索到行，如果FETCH检索到行，则返回TRUE。

%NOTFOUND：与%FOUND相反，如果FETCH操作没有检索到行，则返回TRUE。

%ISOPEN：用于判断游标是否打开。

%ROWCOUNT：用于返回游标影响的行数。

## 显式游标

显式游标是用户显式定义和控制的游标，用于处理返回多行数据的SELECT查询。用户需要显示地在PL/SQL中有显式的定义游标、打开游标、使用游标和关闭游标等步骤。

- 特点：
  - 灵活性：用户可以对查询结果进行精细的控制。可以逐行提取数据，并根据业务逻辑进行相应的处理。
  - 可扩展性：由于显式游标是显式定义的，因此可以轻松地修改或增加新的处理逻辑。
  - 复杂性：增加了代码的复杂性，用户需要仔细管理游标的生命周期，并确保在适当时间关闭游标以释放资源。
  - 资源消耗：虽然显示游标在某些情况下可以提高性能，但是在其他情况下可能会消耗更多的系统资源，例如，频繁地打开和关闭游标可能会增加数据库的开销，影响系统的整体性能。
- 适用场景：

需要对结果集进行复杂的操作，如逐行处理、数据转换等。

## 隐式游标

隐式游标是数据库系统自动管理的游标。当执行SQL语句时，数据库会自动创建一个隐式游标来处理这些操作。

- 特点：
  - 简洁性：无需显式定义和控制游标，减少了编程工作。
  - 性能：由于是数据库系统自动管理，通常具有较好的性能。
  - 缺乏灵活性：用户无法自定义游标的属性和行为。

## 示例

- 显式游标

```
--前置操作，建表和插入数据。
gaussdb=# CREATE TABLE tb_t1(c1 int,c2 int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# INSERT INTO tb_t1 VALUES (generate_series(1,10),generate_series(1,10));
INSERT 0 10

--显式游标示例。
gaussdb=# DECLARE
v_c1 int;
v_c2 int;
v_res int;
```

```
--定义游标。
CURSOR v_cur IS SELECT * FROM tb_t1;
BEGIN
--打开游标。
OPEN v_cur;
LOOP
--使用游标。
FETCH v_cur INTO v_c1,v_c2;
EXIT WHEN v_cur %notfound;
v_res = mod(v_c1,2);

IF v_res = 1 THEN
UPDATE tb_t1 SET c2 = v_c1+1 WHERE c1 = v_c1;
END IF;

END LOOP;
--关闭游标。
CLOSE v_cur;
END;
/
ANONYMOUS BLOCK EXECUTE

--查看表结果。
gaussdb=# SELECT * FROM tb_t1;
c1 | c2
---+---
1 | 2
2 | 2
3 | 4
4 | 4
5 | 6
6 | 6
7 | 8
8 | 8
9 | 10
10 | 10
(10 rows)

--删除表。
gaussdb=# DROP TABLE tb_t1;
DROP TABLE
```

- 隐式游标

```
--前置操作，创建表。
gaussdb=# CREATE TABLE tb_t1(c1 int,c2 int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# CREATE TABLE tb_t2(c1 int,c2 int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--隐式游标使用。
gaussdb=# BEGIN
INSERT INTO tb_t1 VALUES (generate_series(1,4),generate_series(1,4));
RAISE INFO '向tb_t1表中插入了%条数据',SQL%ROWCOUNT;

INSERT INTO tb_t2 VALUES (generate_series(1,6),generate_series(1,6));
RAISE INFO '向tb_t2表中插入了%条数据',SQL%ROWCOUNT;

UPDATE tb_t1 SET c2 = c2*2 WHERE c1 > 4;
RAISE INFO '表tb_t1中修改了数据%条',SQL%ROWCOUNT;

UPDATE tb_t2 SET c2 = c1*2 WHERE c1 > 4;
RAISE INFO '表tb_t2修改了数据%条',SQL%ROWCOUNT;
END;
/
INFO: 向tb_t1表中插入了4条数据
INFO: 向tb_t2表中插入了6条数据
```

```
INFO: 表tb_t1中修改了数据0条
INFO: 表tb_t2修改了数据2条
ANONYMOUS BLOCK EXECUTE

--查看表tb_t1,tb_t2的数据。
gaussdb=# SELECT * FROM tb_t1 ORDER BY 1;
 c1 | c2
----+----
 1 | 1
 2 | 2
 3 | 3
 4 | 4
(4 rows)

gaussdb=# SELECT * FROM tb_t2 ORDER BY 1;
 c1 | c2
----+----
 1 | 1
 2 | 2
 3 | 3
 4 | 4
 5 | 10
 6 | 12
(6 rows)

--删除表。
gaussdb=# DROP TABLE tb_t1;
DROP TABLE
gaussdb=# DROP TABLE tb_t2;
DROP TABLE
```

## 8.12.3 存储过程（Procedure）

### 简介

存储过程是预编译并存储在数据库中的程序，由一组SQL语句组成，用于执行一系列数据库操作。存储过程可以被多个应用程序调用，从而提高代码的复用性和维护性。

存储过程的特点：

- 复杂数据操作简化：执行需要多步操作的复杂业务逻辑，如批量数据处理、数据转换等。
- 性能优化：通过预编译和减少网络通信，提高数据库操作的执行效率。
- 安全性增强：将业务逻辑封装在存储过程中，可以限制直接访问数据库表，增强数据安全性。
- 简化开发和维护：将业务逻辑集中在数据库中，简化应用程序的开发和维护。

### 示例

如下示例中：存储过程prc\_test的作用是将表tb\_test1中数据插入到表tb\_test2中，如果插入失败，将调用时间、错误信息、和数据插入到tb\_test\_err中。

```
--环境准备：建表和插入数据。
gaussdb=# CREATE TABLE tb_test1(id int,info text);
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
gaussdb=# CREATE TABLE tb_test2(id int PRIMARY KEY, info text);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tb_test2_pkey" for table "tb_test2"
CREATE TABLE
gaussdb=# INSERT INTO tb_test1 VALUES (generate_series(1,5),'AAA');
INSERT 0 5
gaussdb=# INSERT INTO tb_test2 VALUES (generate_series(1,2),'aaa');
```

```
INSERT 0 2
gaussdb=# CREATE TABLE tb_test_err(
 err_time text,
 err_info text,
 id int,
 info text
);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'err_time' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--创建存储过程prc_test。
gaussdb=# CREATE OR REPLACE PROCEDURE prc_test(v_time IN text) AS
--定义游标。
 CURSOR v_cur IS SELECT * FROM tb_test1;
 v_id int;
 v_info text;
BEGIN
--打开游标。
 OPEN v_cur;
 LOOP
--使用游标。
 FETCH v_cur INTO v_id,v_info;
--退出循环条件：游标取不到数据时退出循环。
 EXIT WHEN v_cur %notfound;
 BEGIN
--将tb_test1表中数据插入到tb_test2中。
 INSERT INTO tb_test2 VALUES (v_id,v_info);
--异常处理：执行失败时将错误信息插入到tb_test_err表中。
 EXCEPTION
 WHEN OTHERS THEN
 INSERT INTO tb_test_err VALUES (v_time,sqlerrm,v_id,v_info);
 END;
 END LOOP;
--关闭游标。
 CLOSE v_cur;
END;
/
CREATE PROCEDURE

--调用存储过程传入字符串2019-01-31。
gaussdb=# CALL prc_test('2019-01-30');

--查看结果。
gaussdb=# SELECT * FROM tb_test2 ORDER BY id;
 id | info
-----+-----
 1 | aaa
 2 | aaa
 3 | AAA
 4 | AAA
 5 | AAA
(5 rows)

--查看插入失败的数据信息。
gaussdb=# SELECT * FROM tb_test_err;
 err_time | err_info | id | info
-----+-----+-----+-----
2019-01-30 | Duplicate key value violates unique constraint "tb_test2_pkey". | 1 | AAA
2019-01-30 | Duplicate key value violates unique constraint "tb_test2_pkey". | 2 | AAA
(2 rows)

--删除表和存储过程。
gaussdb=# DROP PROCEDURE prc_test;
DROP PROCEDURE
gaussdb=# DROP TABLE tb_test1,tb_test2,tb_test_err;
DROP TABLE
```

### 8.12.4 函数（Function）

#### 简介

函数是一种数据库对象，用来返回单个值或表对象。函数必须有一个返回值，并且定义了返回值的类型。适用于需要返回单个值的情况，如进行计算、数据转换或返回表数据。

#### 示例

如下示例中创建了一个名为add\_num()的函数，入参是两个整数参数，并返回它们的和。

```
--创建函数。
gaussdb=# CREATE OR REPLACE FUNCTION add_num(int, int) RETURNS int AS $$
BEGIN
 RETURN $1+$2;
END;
$$ LANGUAGE plpgsql;

--使用函数add_num()。
gaussdb=# SELECT add_num(3,4);
 add_num

 7
(1 row)

--删除该函数。
gaussdb=# DROP FUNCTION add_num;
DROP FUNCTION
```

### 8.12.5 触发器（Trigger）

#### 简介

触发器是一种特殊的函数，当某个数据变更事件（INSERT、UPDATE、DELETE 或者 TRUNCATE）或者数据库事件（DDL语句）发生时自动执行，而不是由用户或者应用程序进行调用。

表 8-2 表和视图上支持的触发器种类

| 触发时机       | 触发事件                 | 行级  | 语句级  |
|------------|----------------------|-----|------|
| BEFORE     | INSERT/UPDATE/DELETE | 表   | 表和视图 |
|            | TRUNCATE             | 不支持 | 表    |
| AFTER      | INSERT/UPDATE/DELETE | 表   | 表和视图 |
|            | TRUNCATE             | 不支持 | 表    |
| INSTEAD OF | INSERT/UPDATE/DELETE | 视图  | 不支持  |
|            | TRUNCATE             | 不支持 | 不支持  |

## 管理触发器

- **创建触发器**

```
--创建company表。
gaussdb=# CREATE TABLE company(
 ID INT PRIMARY KEY NOT NULL,
 NAME TEXT NOT NULL,
 AGE INT NOT NULL,
 ADDRESS CHAR(50),
 SALARY REAL);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "company_pkey" for table "company"
CREATE TABLE

--创建audit表。
gaussdb=# CREATE TABLE audit(
 EMP_ID INT NOT NULL,
 ENTRY_DATE TEXT NOT NULL);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'emp_id' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--使用CREATE FUNCTION语句创建一个触发器函数。
gaussdb=# CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$
BEGIN
 INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);
 RETURN NEW;
END;
$example_table$ LANGUAGE plpgsql;
CREATE FUNCTION

--使用CREATE TRIGGER语句将该函数与表进行关联。
gaussdb=# CREATE TRIGGER example_trigger AFTER INSERT ON company FOR EACH ROW EXECUTE
PROCEDURE auditlogfunc();
CREATE TRIGGER

--向company表中插入数据。
gaussdb=# INSERT INTO company (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, '张三', 32, '济南',
20000.00);
INSERT 0 1

--查看audit表数据。
gaussdb=# SELECT * FROM audit;
 emp_id | entry_date
-----+-----
 1 | 2024-12-09 17:19:54.70118+08
(1 row)
```

- **修改触发器**

GaussDB提供了ALTER TRIGGER 语句，用于修改触发器。

```
--修改触发器的名称。
gaussdb=# ALTER TRIGGER example_trigger ON company RENAME TO example_trigger1;
ALTER TRIGGER
```

默认创建的触发器处于启用状态，可以使用以下语句禁用或者启用某个触发器。

```
--触发器的禁用。
gaussdb=# ALTER TABLE company DISABLE TRIGGER example_trigger1;
ALTER TABLE
```

```
--向company表中插入数据，查看audit表。
gaussdb=# INSERT INTO company (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, '李四', 30, '安徽',
30000.00);
INSERT 0 1
```

```
gaussdb=# SELECT * FROM audit;
 emp_id | entry_date
-----+-----
 1 | 2024-12-09 11:25:55.992644+08
```



```
(1 row)
```

```
--触发器的启用。
gaussdb=# ALTER TABLE company ENABLE TRIGGER example_trigger1 ;
ALTER TABLE
```

- **查看触发器**

视图information\_schema.triggers中存储了关于触发器的信息。

--使用如下元命令使查询结果竖行回显。

```
gaussdb=# \x
```

Expanded display is on.

```
gaussdb=# SELECT * FROM information_schema.triggers WHERE trigger_name = 'example_trigger1';
```

```
-[RECORD 1]-----+-----
trigger_catalog | postgres
trigger_schema | public
trigger_name | example_trigger1
event_manipulation | INSERT
event_object_catalog | postgres
event_object_schema | public
event_object_table | company
action_order |
action_condition |
action_statement | EXECUTE PROCEDURE auditlogfunc()
action_orientation | ROW
action_timing | AFTER
action_reference_old_table |
action_reference_new_table |
action_reference_old_row |
action_reference_new_row |
created |
```

```
gaussdb=# \x
```

Expanded display is off.

- **删除触发器**

```
gaussdb=# DROP TRIGGER example_trigger1 ON company;
DROP TRIGGER
```

--删除表。

```
gaussdb=# DROP TABLE company;
DROP TABLE
```

```
gaussdb=# DROP TABLE audit;
DROP TABLE
```

## 8.12.6 包（Package）

包（Package）是一组相关的子程序，包括包所使用的游标和变量，一起存储在数据库中，以便作为一个单元继续使用。打包的子程序可以由应用程序或用户显式调用。具有以下特点：

- **促进模块化编程：**通过将相关的函数和存储过程封装在一个Package中，可以更容易地实现模块化编程。模块化可以提高代码的可读性和可维护性。
- **断开依赖链：**对某个数据库对象的修改（如表结构的更改）不会导致依赖于该对象的所有其他对象（如存储过程、函数、视图等）变得无效，从而避免了代价较高的重编译过程。
- **性能优化：**Package在数据库中是以编译形式存储的，当Package被调用时，数据库不需要再次编译代码，从而提高了执行效率。
- **命名空间管理：**Package提供了一种命名空间，可以避免不同模块之间的命名冲突。在同一个Package内，可以有不同的函数和过程，即使它们在其他上下文中有相同的名称也不会冲突。
- **安全性：**通过Package可以更好地控制对数据库对象的访问。

## 创建包

创建包需要两部分内容，包头和包体。包头需要声明所有公共结构，而包体需要定义包的所有结构，包括公共部分和私有部分。

如下示例中，创建emp表，和pkg\_emp\_mgr包。包中有三个存储过程。

- pkg\_emp\_mgr.emp\_add往emp表中插入员工信息。
- pkg\_emp\_mgr.emp\_del作用是删除emp表中员工信息。
- pkg\_emp\_mgr.emp\_res的作用是修改emp表中信息。

```
--PACKAGE仅支持在ORA兼容性数据库中使用。
gaussdb=# CREATE DATABASE ora_testdb DBCOMPATIBILITY 'ORA';
CREATE DATABASE

gaussdb=# \c ora_testdb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "ora_testdb" as user "omm".
--创建emp表。
ora_testdb=# CREATE TABLE emp(
 id INT, -- 员工编号
 name VARCHAR(50), -- 员工姓名
 depno INT, -- 部门编号
 sal FLOAT -- 工资
);
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

--创建包头。
ora_testdb=# CREATE OR REPLACE PACKAGE pkg_emp_mgr AS
 PROCEDURE emp_add(v_id INT, v_name VARCHAR, v_depno INT, v_sal FLOAT);
 PROCEDURE emp_del(v_id INT);
 PROCEDURE emp_res(v_id INT, v_depno INT, v_sal VARCHAR);
END pkg_emp_mgr;
/

--创建包体。
ora_testdb=# CREATE OR REPLACE PACKAGE BODY pkg_emp_mgr AS
 PROCEDURE emp_add(v_id INT, v_name VARCHAR, v_depno INT, v_sal FLOAT) AS
 BEGIN
 INSERT INTO emp VALUES (v_id, v_name, v_depno, v_sal);
 END;
 PROCEDURE emp_del(v_id INT) AS
 BEGIN
 DELETE FROM emp WHERE id = v_id;
 END;
 PROCEDURE emp_res(v_id INT, v_depno INT, v_sal VARCHAR) AS
 BEGIN
 UPDATE emp SET depno = v_depno, sal = v_sal WHERE id = v_id;
 END;
END pkg_emp_mgr;
/

--调用pkg_emp_mgr.emp_add。
ora_testdb=# CALL pkg_emp_mgr.emp_add(1, 'scott', 10, 4000.00);
ora_testdb=# CALL pkg_emp_mgr.emp_add(2, 'july', 20, 3000.00);
--查询表数据。
ora_testdb=# SELECT * FROM emp;
 id | name | depno | sal
----+-----+-----+-----
 1 | scott | 10 | 4000
 2 | july | 20 | 3000
(2 rows)

--调用pkg_emp_mgr.emp_res。
ora_testdb=# CALL pkg_emp_mgr.emp_res(2, 30, 6000.00);
--查询数据。
```

```
ora_testdb=# SELECT * FROM emp;
id | name | depno | sal
-----+-----+-----+-----
 1 | scott | 10 | 4000
 2 | july | 30 | 6000
(2 rows)

--调用pkg_emp_mgr.emp_del。
ora_testdb=# CALL pkg_emp_mgr.emp_del(2);
--查询数据。
ora_testdb=# SELECT * FROM emp;
id | name | depno | sal
-----+-----+-----+-----
 1 | scott | 10 | 4000
(1 row)
--删除包。
ora_testdb=# DROP PACKAGE pkg_emp_mgr;
NOTICE: drop cascades to 3 other objects
DETAIL: drop cascades to function public.emp_add(integer,character varying,integer,double precision)
drop cascades to function public.emp_del(integer)
drop cascades to function public.emp_res(integer,integer,character varying)
DROP PACKAGE

--删除表。
ora_testdb=# DROP TABLE emp;
DROP TABLE

--删除数据库。
ora_testdb=# \c postgres
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "postgres" as user "omm".

gaussdb=# DROP DATABASE ora_testdb;
DROP DATABASE
```

## 8.13 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的事务，将数据从人为不正确的操作中进行恢复。按照恢复对象与底层机制的不同，GaussDB闪回恢复主要分为：闪回查询、闪回表、闪回DROP/TRUNCATE和闪回库。

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，其中通过闪回DROP/TRUNCATE恢复已提交的事务，只需要秒级，并且恢复时间和数据库大小无关。

### 说明

- Ustore存储引擎支持闪回查询、闪回表、闪回DROP/TRUNCATE和闪回库；Astore存储引擎只支持闪回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。

## 为什么需要闪回恢复

在数据库的日常运维与开发过程中，人为误操作是难以完全避免的风险，例如误删一张表、误执行TRUNCATE清空数据，或在业务逻辑变更中错误地更新了关键记录。这类逻辑损坏通常只影响一张或一组表，而非整个数据库实例。

闪回恢复的核心价值在于：当上述逻辑损坏发生时，无需对整个实例做备份恢复或PITR（时间点恢复）重建，即可针对受损的表、库等对象进行快速、局部的还原，从而显著降低人为误操作带来的业务中断风险。

## 闪回恢复的优势是什么

- **秒级恢复**：通过闪回DROP/TRUNCATE恢复已提交的删除、清空数据只需秒级，恢复时间与数据库大小无关，相比备份恢复和PITR大幅缩短了恢复窗口。
- **局部精准恢复**：当逻辑损坏仅限于一张或一组表时，可单独将受损对象恢复至特定时间点，无需对整个实例进行回退，避免影响其他正常业务数据。
- **多种恢复粒度**：提供闪回查询、闪回表、闪回DROP/TRUNCATE以及闪回库等多种能力，覆盖从“查看历史数据”到“恢复被删除对象”的不同恢复需求。
- **时间点与CSN双重定位**：既支持以时间戳（TIMESTAMP）方式指定闪回点，也支持以CSN（Commit Sequence Number，提交序列号）方式精确定位一致性点，满足不同精度的恢复要求。

## 闪回恢复的使用场景

- **误删表/误清空后的快速找回**：开发或运维人员在变更过程中误执行了DROP或TRUNCATE操作，导致业务表数据丢失。在已开启回收站（recyclebin）的前提下，可使用闪回DROP或闪回TRUNCATE从回收站还原被删除的表及其索引、约束等附属结构，秒级恢复业务。
- **误更新数据的逻辑重建**：业务团队在批量更新或脚本执行中错误地修改了关键记录，但损坏范围仅限于个别表。在undo\_retention\_time已保留相应时间范围Undo旧版本的前提下，可先通过闪回查询查看该表在出错前某一时点的快照数据，确认正确版本后，再使用闪回表将整张表恢复至该时间点，实现表级数据还原而不影响其他对象。
- **历史数据审计与比对**：数据分析或审计人员需要查看某张表在过去某个时间点的状态以核对数据变化。在undo\_retention\_time已保留相应历史版本的前提下，借助闪回查询可直接读取指定TIMESTAMP或CSN下的历史版本，无需额外搭建历史库，即可完成数据的回溯查看与逻辑重建。

## 闪回恢复的工作原理

从整体上看，GaussDB闪回恢复依托两大底层机制实现：一是Ustore存储引擎的Undo多版本机制，二是回收站机制。前者负责保留行数据的历史版本，支撑闪回查询与闪回表；后者负责保存被删除对象的物理文件，支撑闪回DROP、闪回TRUNCATE与闪回库。

基于Undo的闪回（闪回查询、闪回表）依赖MVCC多版本并发控制。Ustore在更新或删除数据时，会通过Undo子系统将旧版本单独保存，并由GUC参数undo\_retention\_time控制旧版本的保留时长。执行闪回查询时，系统根据指定的时间点或CSN检索对应的旧版本数据并返回；执行闪回表时，系统在此基础上删除该时间点之后产生的增量数据，并找回该区间内被删除的数据，从而将整张表还原到指定时间点。

基于回收站的闪回（闪回DROP、闪回TRUNCATE、闪回库）依赖回收站机制。开启回收站后，执行表或索引删除时，系统会在原对象所在SCHEMA中自动生成以“BIN”开头的隐藏对象，记录被删除对象的物理文件位置；闪回时再依据这些记录还原物理文件，实现已DROP或TRUNCATE对象的恢复。回收站对象的保留时长由recyclebin\_retention\_time（表级）等参数控制，超时后自动清理。其中，闪回库依赖独立的库级回收站（gs\_db\_recyclebin），由enable\_db\_recyclebin、dbrecyclebin\_retention\_time等参数控制。

通过Undo与回收站两套机制的协同，闪回恢复既能读取与还原行级历史版本，又能找回被整体删除的对象。

## 闪回恢复的分类

按照恢复对象与底层机制的不同，GaussDB闪回恢复主要分为以下几类：

- **闪回查询**

- **简介**

查询过去某个时间点表的快照数据，用于查看和逻辑重建被意外删除或更改的受损数据。基于MVCC多版本机制，通过检索旧版本获取指定历史版本数据。

- **前提条件**

需要设置GUC参数undo\_retention\_time，该参数的值表示旧版本undo的保留时间。

- **示例**

```
--前置操作建表和插入数据。
gaussdb=# CREATE TABLE tb_t1 (c1 int, c2 int) WITH(storage_type = ustore);
gaussdb=# INSERT INTO tb_t1 values (1,1),(2,2);

--查询当前时间和CSN。
gaussdb=# SELECT now(), int8in(xidout(next_csn)) FROM gs_get_next_xid_csn() limit 1;
 now | int8in
-----+-----
2025-04-22 15:22:47.574995+08 | 2020912
(1 row)

--模拟数据误操作。
gaussdb=# UPDATE tb_t1 SET C1=222,C2=222 WHERE C1 = 2;
gaussdb=# SELECT * FROM tb_t1;
 c1 | c2
----+----
 1 | 1
222 | 222
(2 rows)

--根据时间查询该表修改之前的数据。
gaussdb=# SELECT * FROM tb_t1 TIMECAPSULE TIMESTAMP '2025-04-22 15:22:47.574995';
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)

--根据CSN查询该表修改之前的数据。
gaussdb=# SELECT * FROM tb_t1 TIMECAPSULE CSN '2020912';
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)

--删除。
gaussdb=# DROP TABLE tb_t1;
```

- **闪回表**

- **简介**

闪回表可以将表恢复至特定时间点或CSN点。当逻辑损坏仅限于一个或一组表、而非整个数据库时，可借助该特性（基于MVCC多版本机制）快速实现表级数据还原。

- **前提条件**

需要设置GUC参数undo\_retention\_time，该参数的值表示旧版本undo的保留时间。

### - 示例

--前置操作建表和插入数据。

```
gaussdb=# CREATE TABLE tb_t2 (c1 int, c2 int) WITH(storage_type = ustore);
gaussdb=# INSERT INTO tb_t2 VALUES (1,1),(2,2);
```

--查询当前时间和CSN。

```
gaussdb=# SELECT now(), int8in(xidout(next_csn)) FROM gs_get_next_xid_csn() limit 1;
 now | int8in
-----+-----
2025-04-22 15:32:22.99132+08 | 2021175
(1 row)
```

--模拟数据误操作。

```
gaussdb=# UPDATE tb_t2 SET c1 = 222, c2 = 222 WHERE C1 = 2;
gaussdb=# SELECT * FROM tb_t2;
 c1 | c2
----+----
 1 | 1
 222 | 222
(2 rows)
```

--将表闪回至误操作之前时间点。

```
gaussdb=# TIMECAPSULE table tb_t2 TO TIMESTAMP to_timestamp('2025-04-22
15:32:22.99132','YYYY-MM-DD HH24:MI:SS.FF');
gaussdb=# SELECT * FROM tb_t2;
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)
```

--也可以使用如下SQL,根据CSN闪回该表。

```
gaussdb=# UPDATE tb_t2 SET c1 = 333, c2 = 333 where c1 = 2;
gaussdb=# TIMECAPSULE table tb_t2 TO CSN 2021175;
gaussdb=# SELECT * FROM tb_t2;
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)
```

--删除。

```
gaussdb=# DROP TABLE tb_t2;
```

## ● 闪回DROP/TRUNCATE

### - 简介

- 闪回DROP：可以恢复意外删除的表，从回收站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

- 设置GUC参数enable\_recyclebin的值为on，启用回收站功能。
- 设置GUC参数recyclebin\_retention\_time，该参数的值为回收站保留对象的时间，超时的对象回收站会自动清理。

### - 示例

--前置操作建表和插入数据。

```
gaussdb=# CREATE TABLE tb_t3 (c1 int,c2 int) WITH(storage_type = ustore);
```

```
gaussdb=# INSERT INTO tb_t3 VALUES (1,1), (2,2);

--删除该表。
gaussdb=# DROP TABLE tb_t3;

--闪回至删除之前。
gaussdb=# TIMECAPSULE TABLE tb_t3 TO BEFORE DROP;
gaussdb=# SELECT * FROM tb_t3;
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)

--使用TRUNCATE清空该表。
gaussdb=# TRUNCATE TABLE tb_t3;

--闪回至清空之前。
gaussdb=# TIMECAPSULE TABLE tb_t3 TO BEFORE TRUNCATE;
gaussdb=# SELECT * FROM tb_t3;
 c1 | c2
----+----
 1 | 1
 2 | 2
(2 rows)

--删除。
DROP TABLE tb_t3;
```

● **闪回库**

– **简介：**

恢复意外删除的库，从库级回收站中恢复被删除的库及其关联对象。闪回库基于回收站机制，通过还原回收站中记录的库的物理文件，实现已DROP库的恢复。

– **前提条件**

- 开启enable\_db\_recyclebin参数，启用回收站。
- dbrecyclebin\_retention\_time参数用于设置回收站对象的保留时间，超过该时间的回收站对象将被自动清理。
- dbrecyclebin\_maxsize参数用于设置回收站容量，超过容量将不能放入回收站。

– **示例**

开启库级回收站后，闪回回收站中的数据库。

```
--查看回收站。
gaussdb=# SELECT * FROM gs_db_recyclebin;
 rcydbid | rcyname | rcyoriginname | rcyrecyclecsn | rcyrecycletime | rcytablesaces | rcydbsize
-----+-----+-----+-----+-----+-----+-----
(0 rows)

--创建数据库。
gaussdb=# CREATE DATABASE flashback_db;
CREATE DATABASE
--删除数据库放入回收站。
gaussdb=# DROP DATABASE flashback_db;
DROP DATABASE
gaussdb=# SELECT * FROM gs_db_recyclebin;
 rcydbid | rcyname | rcyoriginname | rcyrecyclecsn | rcyrecycletime | rcytablesaces | rcydbsize
-----+-----+-----+-----+-----+-----+-----
 82523 | BIN$1425B67F$2F7B1738==$0 | flashback_db | 13916 | 2024-12-11 09:02:35.20606-05 | 1663 | 26272
(1 row)
```

```
--可通过回收站名闪回drop后的数据库。
gaussdb=# TIMECAPSULE DATABASE "BIN$1425B67F$2F7B1738==$0" to before drop;
TimeCapsule Database
--可通过原名闪回drop后的数据库。
gaussdb=# TIMECAPSULE DATABASE flashback_db to before drop;
TimeCapsule Database
--可通过回收站名闪回后重命名。
gaussdb=# TIMECAPSULE DATABASE "BIN$1425B67F$2F7B1738==$0" to before drop rename to
flashback_db1;
TimeCapsule Database
--可通过原名闪回后重命名。
gaussdb=# TIMECAPSULE DATABASE flashback_db to before drop rename to flashback_db1;
TimeCapsule Database
gaussdb=# SELECT * FROM gs_db_recyclebin;
rcydbid | rcyname | rcyoriginname | rcyrecyclecsn | rcyrecycletime | rcytablespaces | rcydbsize
-----+-----+-----+-----+-----+-----+-----
(0 rows)
--查看闪回后的库。
gaussdb=# SELECT * FROM pg_database where datname = 'flashback_db';
 datname | datdba | encoding | datcollate | datctype | datistemplate | datallowconn |
datconnlimit | datlastsysoid | datfrozenxid | dattablespace
| datcompatibility | datacl | datfrozenxid64 | datminmxid | dattimezone | dattype
-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
 flashback_db | 10 | 7 | en_US.UTF-8 | en_US.UTF-8 | f | t | -1
| 12995 | 0 | 1663
| A | 40936 | 2 | PRC | D
(1 row)
```

闪回恢复与备份恢复的区别

闪回恢复与备份恢复都是GaussDB的数据恢复手段，二者的共同点在于：都用于在数据发生丢失或损坏后，将数据恢复到出错前的状态，以保障数据的安全可靠。

二者的核心差异在于数据来源与恢复方式：闪回基于运行实例内部保留的Undo历史版本与回收站数据，无需借助外部数据即可在线恢复；而备份恢复基于独立保存的物理备份集（全量备份与增量备份）以及归档WAL日志（用于PITR），需要重新加载备份并回放日志来重建数据。这一差异主要源于数据来源不同——闪回依赖的是在线保留、有保留时间限制的历史版本与回收站对象，备份恢复依赖的是离线保存的完整备份与日志。

由于上述差异，二者在恢复粒度、速度与适用范围上各有侧重：闪回可对表、库等对象进行局部恢复，恢复速度快——其中闪回DROP/TRUNCATE可达秒级且与数据库大小无关，但只能恢复到Undo或回收站保留期之内；备份恢复通常以整个实例或数据库为单位进行重建，恢复时长随数据量增长可达数分钟甚至数小时，但能够恢复到更早的历史时间点，并可应对介质损坏、实例级灾难等闪回无法覆盖的场景。因此，闪回更适合保留期内人为误操作的快速恢复，备份恢复更适合超出保留期、整库灾难或介质故障的恢复场景，二者互为补充。

| 对比维度 | 闪回恢复                      | 备份恢复                    |
|------|---------------------------|-------------------------|
| 共同点  | 均为数据恢复手段，都用于将数据恢复到出错前的状态。 |                         |
| 数据来源 | 实例内Undo历史版本与回收站。          | 独立的物理备份集与归档WAL日志（PITR）。 |
| 恢复粒度 | 表级/库级等局部对象。               | 整个实例或数据库。               |



| 对比维度 | 闪回恢复                           | 备份恢复               |
|------|--------------------------------|--------------------|
| 恢复速度 | 较快；闪回DROP/TRUNCATE可达秒级、与库大小无关。 | 数分钟至数小时，与数据量相关。    |
| 时间范围 | 限于Undo/回收站保留期内。                | 可恢复到更早的历史时间点。      |
| 典型场景 | 保留期内人为误操作恢复。                   | 介质损坏、实例级灾难、超保留期恢复。 |

## 与闪回恢复相关的特性和操作

在了解闪回恢复的核心概念与适用场景后，如需了解更多详情以及在实际业务中配置使用，可参考以下对应章节：

- 前置开关与参数配置：
  - 开启闪回功能：闪回查询/闪回表依赖Undo旧版本保留，需配置undo\_retention\_time；关闭（设为0）时无法使用闪回功能。详情参考《参考》中“数据库运行参数说明 > GUC参数说明 > 闪回相关参数”章节。
  - 开启回收站：闪回DROP/TRUNCATE需将enable\_recyclebin设置为on，闪回库需将enable\_db\_recyclebin设置为on，并可通过recyclebin\_retention\_time、dbrecyclebin\_retention\_time设置回收站对象的保留时长。详情参考《参考》中“数据库运行参数说明 > GUC参数说明 > 闪回相关参数”章节。
- 闪回恢复操作：
  - 闪回查询：使用“TIMECAPSULE { TIMESTAMP | CSN } expression”查询表在指定时间点或CSN点的历史数据，详情请参考[闪回查询](#)章节。
  - 闪回表：使用“TIMECAPSULE TABLE table\_name TO { TIMESTAMP | CSN } expression”将表恢复至指定时间点，详情请参考[闪回表](#)章节。
  - 闪回DROP/TRUNCATE：从回收站恢复被删除或被清空的表及其附属结构，详情请参考[闪回DROP/TRUNCATE](#)章节。

## 8.14 VACUUM

在GaussDB中，当数据被更新或删除时，系统并不会立即释放物理空间，而是将其标记为“可重用”。随着时间推移，表中的死元组会越来越多。死元组（dead tuples）是指已被删除或因更新而被新版本覆盖，不再被任何活跃事务引用的历史数据行。

上述情况不仅会导致存储空间利用率下降，也可能影响查询性能，因为查询需要遍历这些无用的元组。而VACUUM命令的主要任务就是清理这些死元组，并回收相应的存储空间。

### 为什么需要 VACUUM

概括来说，VACUUM的核心目的就是清理持续累积的死元组、回收其占用的可重用空间，从而抑制Astore表膨胀、保障空间利用率与查询性能。

Astore存储引擎采用多版本并发控制（MVCC）来保证数据在高并发环境中的一致性和可访问性，每个事务在它开始的时间点都在其自己的数据库快照上运行，这就意味着

无法立即删除过时的数据。当进行UPDATE和DELETE操作时，Astore存储引擎会为其其他正在运行中的事务保留数据的历史版本：对于删除操作，并不会真正在数据页上删除元组，而是标记其被删除，这些被标记删除的元组即死元组；更新操作同样采用标记机制，相当于先标记删除原数据行，再插入新数据行。由于删除并非真正删除数据，死元组依旧占用磁盘空间，就会出现虽然查询表数据很少，但是表所占磁盘空间很大的情况。此外，由于Astore的更新是基于追加写的，数据的新老版本在数据页面上共存，这进一步导致了数据库表空间的膨胀。

需要说明的是，Ustore存储引擎的日常旧版本回收由Undo子系统自治完成，不依赖VACUUM；但Ustore仍支持VACUUM与VACUUM FULL（如彻底回收物理空间），且系统表清理与空闲空间映射（FSM）维护仍依赖autovacuum。

## VACUUM 的优势是什么

- **控制表膨胀：**清理死元组并维护FSM，将过时数据占用的空间标记为可重用，供后续插入操作使用，避免表空间持续膨胀，提升空间复用能力。
- **轻量非阻塞：**普通VACUUM在清理过程中获取较低级别的锁，不阻塞业务的正常读写，对数据库性能影响较小，可在业务运行期间执行。
- **冻结旧事务：**在清理过程中尝试冻结旧事务，使元组对所有快照可见，从而在可见性判断、修改前检查等环节起到加速作用，便于推进表的冻结事务号（relfrozenxid）并回收CLOG/CSNLOG等事务文件。
- **支持自动清理：**通过autovacuum守护线程，数据库可在死元组达到阈值时自动触发清理与统计信息更新，减少人工运维负担。

## VACUUM 的使用场景

- **高频增删改表的膨胀治理：**对于订单、账务等存在大量UPDATE/DELETE的Astore业务表，运维团队常面临表占用空间远大于有效数据的膨胀问题。此时定期对表执行VACUUM，可清理死元组并将空间标记为可重用，从而控制表膨胀、维持稳定的查询性能，且执行期间不阻塞正常业务读写。
- **彻底回收磁盘空间：**当业务表已严重膨胀、需要将磁盘空间真正归还操作系统时，运维人员可在业务低谷期对表执行VACUUM FULL。该操作会重建原表并物理释放空间，但执行期间会锁表，期间无法对该表进行读写访问，因此适合在停业务窗口或低峰期对大表进行整理。
- **常态化自动维护：**在无需人工干预的常规运维中，DBA可开启autovacuum，由守护线程自动监测各表的死元组数量，在达到阈值时自动执行VACUUM与统计信息收集，使表膨胀和统计信息陈旧问题在日常运行中被持续抑制。

## VACUUM 的工作原理

VACUUM的执行可概括为“扫描清理、空间维护、事务冻结、信息更新”几个环节。系统首先扫描表中的页面，默认Lazy模式下会借助可见性映射（VM）跳过全可见页面，并将其中的死元组标记为可重用状态，但普通VACUUM并不重新组织活跃元组的物理存储位置，也不立即把空间归还操作系统，而是通过维护FSM记录可用空间，供后续插入复用。

在此基础上，VACUUM会尝试冻结旧事务，将元组事务标记为对所有快照可见，以加速可见性判断；同时更新VM、统计信息并回收事务文件等。普通VACUUM清理时获取较低级别的锁，不阻塞业务的正常读写。

自动清理由autovacuum守护线程实现，它由launcher与worker两类线程协同：autovacuum launcher在参数开启时常驻，负责调度；autovacuum worker实际连接数据库并对选定的表执行VACUUM。其触发依据死元组数量阈值，当死元组数超过

$\text{autovacuum\_vacuum\_scale\_factor} \times \text{表行数} + \text{autovacuum\_vacuum\_threshold}$ 时即触发自动清理。通过手动VACUUM与autovacuum的配合，数据库得以在长期运行中持续回收空间、维护统计信息，保障性能稳定。

## VACUUM 的分类

按照清理力度和触发方式，VACUUM相关操作可分为以下几类：

1. 按清理力度，可分为普通VACUUM与VACUUM FULL。普通VACUUM仅标记并复用死元组空间、轻量非阻塞；VACUUM FULL重建表并将空间归还操作系统、需锁表，清理更彻底但代价更高。

### - VACUUM

#### ■ 示例

```
--创建表tbl_test，并插入数据。
gaussdb=# CREATE TABLE tbl_test(c1 int);
gaussdb=# INSERT INTO tbl_test VALUES (1);

--查看数据，和数据的ctid。
gaussdb=# SELECT ctid,* FROM tbl_test;
 ctid | c1
-----+----
(0,1) | 1
(1 row)

--删除该数据。
gaussdb=# DELETE FROM tbl_test;

--重新插入一条数据，使用了一个新的ctid。
gaussdb=# INSERT INTO tbl_test VALUES (2);
gaussdb=# SELECT ctid,* FROM tbl_test;
 ctid | c1
-----+----
(0,2) | 2
(1 row)

--使用VACUUM命令回收之后，在插入数据，复用了旧的空间。
gaussdb=# VACUUM ANALYZE tbl_test;
gaussdb=# INSERT INTO tbl_test VALUES (3);
gaussdb=# SELECT ctid,* FROM tbl_test;
 ctid | c1
-----+----
(0,1) | 3
(0,2) | 2
(2 rows)

--删除表。
gaussdb=# DROP TABLE tbl_test;
```

### - VACUUM FULL

#### ■ 示例

```
--建表。
gaussdb=# CREATE TABLE tbl_test2(c1 int);

--插入10万条数据并查看表的大小。
gaussdb=# INSERT INTO tbl_test2 VALUES (generate_series(1,100000));
gaussdb=# SELECT 'tbl_test2' AS tablename, pg_size_pretty(pg_relation_size('tbl_test2')) AS size;
tablename | size
-----+-----
tbl_test2 | 3048 kB
(1 row)

--删除数据并查看表大小。
gaussdb=# DELETE FROM tbl_test2;
```

```
gaussdb=# SELECT 'tbl_test2' AS tablename, pg_size_pretty(pg_relation_size('tbl_test2')) AS size;
tablename | size
-----+-----
tbl_test2 | 3048 kB
(1 row)

--使用VACUUM FULL回收空间,并查看表的大小。
gaussdb=# VACUUM FULL ANALYZE tbl_test2;
gaussdb=# SELECT 'tbl_test2' AS tablename, pg_size_pretty(pg_relation_size('tbl_test2')) AS size;
tablename | size
-----+-----
tbl_test2 | 0 bytes
(1 row)
```

2. 按触发方式，可分为上文已介绍的手动VACUUM与自动清理（autovacuum）。手动VACUUM由用户主动执行，可控性强；autovacuum由守护线程根据死元组阈值自动触发，适合常态化维护。需要注意，业务DDL可终止autovacuum进程，但无法终止手动执行的VACUUM。

– 自动清理（autovacuum）

表 8-3 autovacuum 参数说明

| 参数名称                            | 默认值 | 参数解释                                                                                                                                                                                                                    |
|---------------------------------|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| autovacuum                      | on  | 控制autovacuum是否打开。                                                                                                                                                                                                       |
| autovacuum_naptime              | 600 | 控制两次autovacuum操作的时间间隔。默认值为600，表示600秒。                                                                                                                                                                                   |
| autovacuum_mode                 | mix | 控制autovacuum线程的具体操作： <ul style="list-style-type: none"> <li>analyze：表示只执行autoanalyze。</li> <li>vacuum：表示只执行autovacuum。</li> <li>mix：表示autoanalyze和autovacuum都执行。</li> <li>none：表示autoanalyze和autovacuum都不执行。</li> </ul> |
| autovacuum_max_workers          | 3   | 同时运行autovacuum worker的最大线程数。                                                                                                                                                                                            |
| autovacuum_vacuum_scale_factor  | 0.2 | 用于判断是否触发VACUUM。<br>取值范围：0.0~100.0。                                                                                                                                                                                      |
| autovacuum_vacuum_threshold     | 50  | 用于判断是否触发VACUUM。<br>取值范围：0~2147483647。                                                                                                                                                                                   |
| autovacuum_analyze_scale_factor | 0.1 | 用于判断是否触发ANALYZE。<br>取值范围：0.0~100                                                                                                                                                                                        |

| 参数名称                         | 默认值                                        | 参数解释                                                  |
|------------------------------|--------------------------------------------|-------------------------------------------------------|
| autovacuum_analyze_threshold | 50                                         | 用于判断是否触发ANALYZE。<br>取值范围：0~2147483647                 |
| autovacuum_vacuum_cost_limit | -1，表示该参数同vacuum_cost_limit（控制手动vacuum的参数）。 | autovacuum操作使用的开销值，即开销达到该值之前不会进入休眠。                   |
| autovacuum_vacuum_cost_delay | -1，表示该参数同vacuum_cost_delay（控制手动vacuum的参数）。 | autovacuum操作开销达到autovacuum_vacuum_cost_limit值之后休眠的时间。 |

当死元组数量达到阈值就会触发autovacuum，具体的阈值计算公式如下：  
$$\text{number\_of\_dead\_tuples} > \text{autovacuum\_vacuum\_scale\_factor} * \text{reltuples} + \text{autovacuum\_vacuum\_threshold}$$

3. 按作用对象，VACUUM相关操作可作用于普通表、系统表、Toast表、分区表、索引、压缩表（ILM）等多种对象。不同对象对VACUUM、AUTOVACUUM、VACUUM FULL三种操作的支持情况如下表所示。

表 8-4 VACUUM 对象类型

| 对象类型     | VACUUM | AUTOVACUUM | VACUUM FULL |
|----------|--------|------------|-------------|
| 普通表      | √      | √          | √           |
| 系统表      | √      | √          | √           |
| Toast表   | √      | √          | √           |
| EnToast表 | √      | √          | ×           |
| 分区表      | √      | √          | √           |
| 二级分区表    | √      | √          | √           |
| 临时表      | √      | ×          | √           |
| 全局临时表    | √      | ×          | √           |
| 索引       | √      | √          | ×           |
| 压缩表（ILM） | √      | √          | √           |

VACUUM 与 VACUUM FULL 的区别

VACUUM与VACUUM FULL都是GaussDB中用于清理表空间、处理死元组的命令，二者的共同点在于都用于回收过时数据占用的空间、缓解表膨胀。

二者的核心差异在于清理方式与对业务的影响：普通VACUUM只将死元组空间标记为可重用，并不进行物理删除，也不把空间归还操作系统，过程中仅获取4级锁（ShareUpdateExclusiveLock），不影响读写；VACUUM FULL则会重建原表、物理删除死元组并将空间释放给操作系统，但执行期间会获取8级锁（AccessExclusiveLock），阻塞其他事务的读写访问。

这一差异决定了二者的适用场景不同：VACUUM是轻量级、不阻塞业务的常规清理手段，适合日常频繁执行；VACUUM FULL是重量级操作，执行时间随数据量增长，适合在业务低谷期对严重膨胀的表做彻底空间回收。

| 对比维度 | VACUUM                                 | VACUUM FULL                      |
|------|----------------------------------------|----------------------------------|
| 共同点  | 均用于清理死元组、缓解表膨胀。                        |                                  |
| 清理方式 | 标记死元组空间为可重用，不物理删除。                     | 重建原表，物理删除死元组。                    |
| 空间归还 | 仅供库内复用，不归还操作系统。                        | 将空间释放给操作系统。                      |
| 是否锁表 | 获取4级锁（ShareUpdateExclusiveLock），不影响读写。 | 获取8级锁（AccessExclusiveLock），影响读写。 |
| 性能影响 | 轻量级，影响小。                               | 重量级，影响大。                         |
| 适用场景 | 日常常规清理。                                | 业务低谷期彻底回收空间。                     |

## Astore VACUUM 与 Ustore VACUUM 的区别

本节对比GaussDB新一代存储引擎Ustore与Astore在VACUUM机制和功能上的差异。Astore的堆表与Btree索引、Ustore的堆表与UB-tree索引（分为RCR与PCR两种格式）在AUTOVACUUM、VACUUM、VACUUM FULL三种操作下的功能支持情况如下三张表所示。

表 8-5 表 3 AUTOVACUUM 功能对比

| AUTOVACUUM  | Astore | Btree | Ustore | Ubtree (RCR) | Ubtree (PCR) |
|-------------|--------|-------|--------|--------------|--------------|
| 扫描并删除行的无效版本 | √      | √     | ×      | √            | √            |
| 维护统计信息      | √      | √     | ×      | ×            | ×            |
| 维护FSM准确性    | √      | √     | √      | √            | √            |
| 维护VM准确性     | √      | ×     | ×      | ×            | ×            |
| 冻结过老的事务号    | √      | ×     | ×      | √            | ×            |

| AUTOVACUUM | Astore | Btree | Ustore | Ubtree<br>( RCR ) | Ubtree<br>( PCR ) |
|------------|--------|-------|--------|-------------------|-------------------|
| 回收事务文件     | √      | √     | √      | √                 | √                 |
| 释放物理空间     | ×      | ×     | ×      | ×                 | ×                 |

表 8-6 表 4 VACUUM 功能对比

| VACUUM      | Astore | Btree | Ustore | Ubtree<br>( RCR ) | Ubtree ( PCR ) |
|-------------|--------|-------|--------|-------------------|----------------|
| 扫描并删除行的无效版本 | √      | √     | √      | √                 | √              |
| 维护统计信息      | √      | √     | √      | √                 | √              |
| 维护FSM准确性    | √      | √     | √      | √                 | √              |
| 维护VM准确性     | √      | ×     | ×      | ×                 | ×              |
| 冻结过老的事务号    | √      | ×     | √      | √                 | √              |
| 回收事务文件      | √      | √     | √      | √                 | √              |
| 释放物理空间      | ×      | ×     | ×      | ×                 | ×              |

表 8-7 表 5 VACUUM FULL 功能对比

| VACUUM FULL | Astore | Btree | Ustore | Ubtree<br>( RCR ) | Ubtree<br>( PCR ) |
|-------------|--------|-------|--------|-------------------|-------------------|
| 释放物理存储空间    | √      | √     | √      | √                 | √                 |

与 VACUUM 相关的特性和操作

- 在了解VACUUM的核心概念与分类后，如需了解更多细节以及在实际业务中配置使用，可参考以下对应章节：
- 对表执行VACUUM：对指定表执行清理操作以回收空间，详情参考《 管理员指南 》中“导入数据 > 对表执行VACUUM”章节。
  - VACUUM语法：了解VACUUM/VACUUM FULL命令的完整语法与参数，详情参考《 参考 》中“SQL参考 > SQL语法 > V > VACUUM”章节。
  - 自动清理与参数配置：通过autovacuum、autovacuum\_vacuum\_scale\_factor、autovacuum\_vacuum\_threshold等参数控制自动清理的开启与触发条件，详情参考《 参考 》中“数据库运行参数说明 > GUC参数说明 > 自动清理”章节及《 性能调优 》中“VACUUM/autovacuum介绍与调优 > autovacuum的相关参数”章节。

- VACUUM/autovacuum调优：针对清理效率与膨胀控制进行优化，详情参考《性能调优》中“VACUUM/autovacuum介绍与调优”章节。

## 8.15 TOAST 表

### 简介

TOAST（The Oversized-Attribute Storage Technique）是一种特殊的存储机制。只有一些具有变长表现形式的数据才会支持TOAST，比如text类型。TOAST机制的主要目的是为了解决数据库中单个字段过大，无法直接存储在标准数据页中的问题。由于在GaussDB存储方式中，一条数据所有的字段组合在一起称为tuple，多个tuple组成一个page。page是数据在文件存储中的基本单位。GaussDB中的page默认大小为8KB，当某行数据大小超过page的大小时，数据库会启动TOAST，确保这些大字段能够被有效存储并管理。

行外存储是TOAST技术的一部分，该技术将大的字段压缩或切片成多个物理行存到另一张表（TOAST表），而主表只保留对这些大字段值的引用（通常是一个指针）。这样做的好处是，可以避免单行数据过大导致性能问题，同时仍然保持对大字段数据的有效访问。

如果某张表有一张关联的TOAST表，在PG\_CLASS表里的reltoastrelid字段记录该TOAST表的OID，如果没有关联的TOAST表，reltoastrelid的值为0。

GaussDB中表的字段支持多种存储策略。有以下四种不同的存储策略：

1. PLAIN：
  - 不允许压缩和行外存储。这种策略通常用于不能使用TOAST机制的数据类型。
  - 适用于不需要TOAST就能存放的数据类型，如整数类型、字符类型等。
  - 对于存储长度超过page大小的类型（如text），不允许采用此策略。
2. EXTENDED：
  - 允许压缩和行外存储。这是大多数可以使用TOAST机制的数据类型的默认存储策略。
  - 数据库会首先尝试对数据进行压缩，如果压缩后的数据仍然太大，则进行行外存储。
  - EXTENDED是大多数可TOAST的数据类型默认策略，如text、bytea等。
3. EXTERNAL：
  - 允许行外存储，但不允许压缩。
  - 这种策略可以使text、bytea数据类型的字符串操作更快，因为不需要读取和解压整行数据。
4. MAIN：
  - 允许压缩。
  - 主要适用于numeric类型的数据。
  - 数据库会首先尝试对数据进行压缩，如果压缩后的数据仍然太大，则进行行外存储。

### 示例

```
--建表。
gaussdb=# CREATE TABLE tb_t1(id int, info text);
```



```
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
gaussdb=# CREATE TABLE tb_t2(id int, age int);
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

--查看表tb_t1信息，info字段为text类型，且Storage为extended。
--部分回显内容如下：
gaussdb=# \d+ tb_t1
 Table "public.tb_t1"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id | integer | | plain | |
 info | text | | extended| |

--查看表tb_t2信息，所有字段为int类型。
--部分回显内容如下：
gaussdb=# \d+ tb_t2
 Table "public.tb_t2"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
 id | integer | | plain | |
 age | integer | | plain | |

--如下查询表示tb_t1有关联的TOAST表，tb_t2表没有关联的TOAST表。
gaussdb=# SELECT oid,relname,reltoastrelid FROM pg_class WHERE relname in ('tb_t1','tb_t2');
 oid | relname | reltoastrelid
-----+-----+-----
34519 | tb_t1 | 34522
34524 | tb_t2 | 0
(2 rows)

--查询tb_t1关联的TOAST表信息。
gaussdb=# SELECT relname,oid FROM pg_class WHERE oid = '34522';
 relname | oid
-----+-----
pg_toast_34519 | 34522
(1 row)

--删除。
gaussdb=# DROP TABLE tb_t1,tb_t2;
DROP TABLE
```

## 8.16 HTAP

### 简介

HTAP（Hybrid Transactional/Analytical Processing，混合事务分析处理）数据库架构是一种突破传统的处理事务型工作负载（OLTP）与分析型工作负载（OLAP）系统分离模式的技术创新方案。该架构通过统一数据存储引擎和计算资源调度机制，实现OLTP与OLAP在同一个技术架构内的协同运作，有效消除传统架构因数据冗余、延迟同步和资源隔离导致的分析效率低下问题，为实时决策支持提供统一数据基础设施。

GaussDB HTAP架构创新性地传统行存引擎与执行引擎基础上，深度融合IMCV（In-memory Column View，列式内存引擎）、混合负载透明路由机制以及向量化执行引擎，构建完整的行列混存技术体系。该架构通过SQL级数据迁移接口实现行存数据向IMCV的实时转换，依托内存持久化技术保障TP/AP操作的数据强一致性，有效解决传统行列混合架构的数据同步延迟问题。其核心突破在于：基于多维度代价模型的智能路由引擎可实时解析混合负载特征，自动将OLTP请求定向至行存引擎保障事务性

能，同时将复杂分析查询精准路由至IMCV引擎实现性能的有效提升，最终通过统一SQL接口对外提供无感知的实时分析能力。

### 说明

当前HTAP仅支持Ustore存储引擎。

## 示例

本节以完整的执行流程为例，介绍如何使用HTAP混合负载查询。

**步骤1** 开启HTAP功能并设置相关参数。具体操作请参见《最佳实践》中“HTAP最佳实践 > 前置准备”章节。

分布式下HTAP仅支持ORA兼容性，因此创建ORA兼容性数据库，后续步骤中所有操作均在该数据库下进行。

```
htapdb=# CREATE DATABASE htapdb WITH DBCOMPATIBILITY='ORA';
CREATE DATABASE
htapdb=# \c htapdb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "htapdb" as user "xxx".
```

**步骤2** 执行HTAP DDL，创建IMCV元数据并将行存数据加载至IMCV。

- 通过ALTER方式，对已经存在的表创建IMCV。

```
--场景1:
--已经存在的普通表全部列创建IMCV。
htapdb=# DROP TABLE IF EXISTS htap_test;
NOTICE: table "htap_test" does not exist, skipping
DROP TABLE
htapdb=# CREATE TABLE htap_test(id int, dept_id int, salary int, name varchar(20), comments
varchar(20)) WITH (storage_type = ustore);
CREATE TABLE
htapdb=# INSERT INTO htap_test VALUES (generate_series(1,3000), 1, 1000, 'large', 'large table');
INSERT 0 3000
htapdb=# ALTER TABLE htap_test COLVIEW;
ALTER TABLE

--场景2:
--已经存在的普通表指定字段创建IMCV。
htapdb=# DROP TABLE IF EXISTS htap_test;
DROP TABLE
htapdb=# CREATE TABLE htap_test(id int, dept_id int, salary int, name varchar(20), comments
varchar(20)) WITH (storage_type = ustore);
CREATE TABLE
htapdb=# INSERT INTO htap_test VALUES (generate_series(1,3000), 1, 1000, 'large', 'large table');
INSERT 0 3000
htapdb=# ALTER TABLE htap_test COLVIEW(id, dept_id, salary);
ALTER TABLE

--场景3:
--已经存在的分区表指定分区创建IMCV。
htapdb=# DROP TABLE IF EXISTS htap_test;
DROP TABLE
htapdb=# CREATE TABLE htap_test(id int, dept_id int, salary int, name varchar(20), comments
varchar(20)) WITH (storage_type = ustore)
PARTITION BY RANGE(id)
(
PARTITION p1 VALUES LESS THAN (1001),
PARTITION p2 VALUES LESS THAN (2001),
PARTITION p3 VALUES LESS THAN (3001)
);
CREATE TABLE
htapdb=# INSERT INTO htap_test VALUES (generate_series(1,3000), 1, 1000, 'large', 'large table');
INSERT 0 3000
--仅指定分区p1创建IMCV。
```

```
htapdb=# ALTER TABLE htap_test MODIFY PARTITION p1 COLVIEW;
ALTER TABLE
```

- 通过CREATE方式，在创建表时创建IMCV。
  - 场景1：  
--建表时全部字段创建IMCV。  
htapdb=# DROP TABLE IF EXISTS htap\_test;  
DROP TABLE  
htapdb=# CREATE TABLE htap\_test(id int, dept\_id int, salary int, name varchar(20), comments  
varchar(20)) COLVIEW WITH (storage\_type = ustore);  
CREATE TABLE
  - 场景2：  
--建表时指定字段创建IMCV。  
htapdb=# DROP TABLE IF EXISTS htap\_test;  
DROP TABLE  
--指定加载(id, dept\_id, name)。  
htapdb=# CREATE TABLE htap\_test(id int, dept\_id int, salary int, name varchar(20) NOCOLVIEW,  
comments varchar(20) NOCOLVIEW) COLVIEW WITH (storage\_type = ustore);  
CREATE TABLE
  - 场景3：  
--建表时指定分区创建IMCV。  
htapdb=# DROP TABLE IF EXISTS htap\_test;  
DROP TABLE  
htapdb=# CREATE TABLE htap\_test(id int, dept\_id int, salary int, name varchar(20), comments  
varchar(20)) COLVIEW WITH (storage\_type = ustore)  
PARTITION BY RANGE(id)  
(  
PARTITION p1 VALUES LESS THAN (1001),  
PARTITION p2 VALUES LESS THAN (2001) NOCOLVIEW,  
PARTITION p3 VALUES LESS THAN (3001) NOCOLVIEW  
);  
CREATE TABLE

### 步骤3 查看创建IMCV的表的基本信息。

```
htapdb=# SELECT relroid,relname,imcvnattr,imcvkey FROM gs_imcv;
relroid | relname | imcvnattr | imcvkey
-----+-----+-----+-----
33248 | htap_test | 3 | 1 2 3
(1 row)
-- 通过gs_htap_imcv_info()函数查询表的基本信息，包含已加载的行组数、列存数据量等。
htapdb=# SELECT * FROM gs_htap_imcv_info();
rel_name | partition_name | subpartition_name | schema_name | user_name | imcvnattr | imcvkey | priority
| rowgroup_num | populated_rowgroup_num | cu_num_in_
mem | cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----
htap_test | | | public | **** | 3 | 1 2 3 | 1 | 0 | 0
| 0 | 0 | 0 | 0
(1 row)
```

### 步骤4 执行查询，根据业务负载特征，准确路由至行存/列存引擎。

```
--自动透明路由模式：根据相关表、列是否加载列存以及判断行存/列存计划代价的高低，自动选择行存、列存以及行列混合执行计划。
htapdb=# SET htap_router_mode='auto';
SET
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM htap_test; --列存计划代价更低，选择列存计划。
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on htap_test
(4 rows)
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM htap_test; --加载列未覆盖查询要求，选择行存计划。
QUERY PLAN
```

```

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on htap_test
(3 rows)

--强制列存模式：当查询相关表和列存在列存加速时，自动执行列存计划，若表不存在列存加速或者加载列未覆盖查询要求，则执行行存计划。
htapdb=# SET htap_router_mode='column';
SET
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM htap_test; --强制执行列存计划。
QUERY PLAN

Row Adapter
-> Vector Streaming (type: GATHER)
Node/s: All datanodes
-> Imcv Scan on htap_test
(4 rows)
htapdb=# EXPLAIN (COSTS OFF) SELECT * FROM htap_test; --加载列未覆盖查询要求，选择执行行存计划。
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on htap_test
(3 rows)

--强制行存模式：无论是否加载HTAP列存，对于查询负载均选择执行行存计划。
htapdb=# SET htap_router_mode='row';
SET
htapdb=# EXPLAIN (COSTS OFF) SELECT id,dept_id FROM htap_test; --强制执行行存计划。
QUERY PLAN

Streaming (type: GATHER)
Node/s: All datanodes
-> Seq Scan on htap_test
(3 rows)
```

**步骤5 【可选】卸载IMCV。**

```
--卸载IMCV表。卸载IMCV后，需要重新执行步骤2，才能继续使用HTAP功能。
htapdb=# ALTER TABLE htap_test NOCOLVIEW;
ALTER TABLE
htapdb=# SELECT * FROM gs_htap_imcv_info();
 rel_name | partition_name | subpartition_name | schema_name | user_name | imcvnattr | imcvkey | priority |
rowgroup_num | populated_rowgroup_num | cu_num_in_mem |
cu_mem_size | cu_num_in_disk | cu_disk_size
-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

----结束

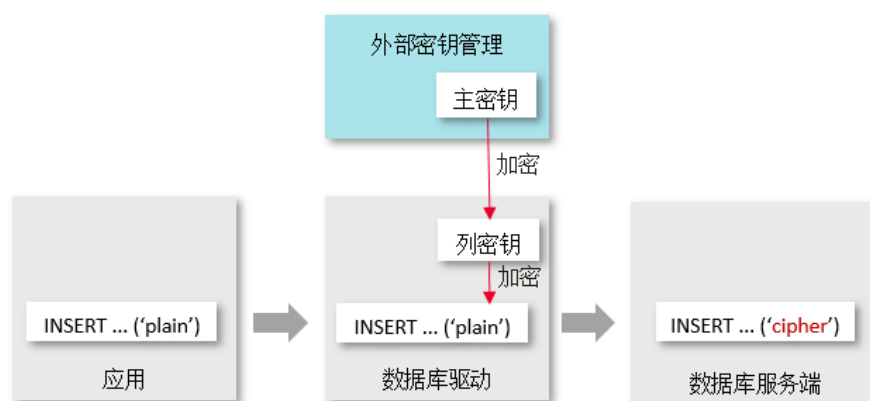
## 8.17 全密态数据库

全密态数据库是指能够解决数据全生命周期的隐私保护问题，使得系统无论在何种业务场景和环境下，数据在传输、运算以及存储的各个环节始终都处于密文状态，同时密钥掌握在授权用户手中的数据库管理系统。

### 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，其关系如[图8-1](#)所示：

图 8-1 加密模型图



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。

## 密钥管理

全密态数据库目前支持以下密钥管理工具进行独立管理：

- Huawei KMS：由华为云提供的在线密钥管理服务，提供创建、删除、查询和备份密钥等功能，并支持在线使用密钥对数据进行加解密。
- user\_token：由用户提供的密码在客户端派生密钥，或者直接对接密钥。
- sdf\_kms：支持基于鲲鹏TEE的内生密钥管理、支持三未信安SJJ1994加密机。

## 数据库驱动

全密态数据库目前支持的数据库驱动有gsql、JDBC和GO驱动。

### 使用 gsql 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gsq工具无法识别完整参数。

基于三末信安SJJ1994加密机密钥管理：

```
gaussdb=# \key_info keyType=sdf_kms,library_path={动态库的路径},library_version=SJJ1994,kek_algo=SGD_SM4_CBC,mac_algo=SGD_SM4_MAC,dkey_algo=SGD_SM4_CBC
```

### 说明

- 使用三末信安SJJ1994加密机前，需要正确配置ini文件，并且保证加密机能正常连接使用。
- library\_path可以使用绝对路径或相对路径（相对于\$GAUSSHOME/lib）。
- kek\_algo是密钥加密密钥（KEK）用于加密会话密钥的算法，允许配置算法：SGD\_SM4\_ECB（默认算法）、SGD\_SM4\_CBC、SGD\_SM4\_CTR、SGD\_AES\_ECB、SGD\_AES\_CBC、SGD\_AES\_CTR。
- dkey\_algo是会话密钥用于加密数据密钥的算法，允许配置算法：SGD\_SM4\_CBC、SGD\_SM4\_CTR、SGD\_AES\_CBC、SGD\_AES\_CTR。
- mac\_algo是会话密钥用于对数据密钥进行完整性校验的算法，允许配置算法：SGD\_SM4\_MAC、SGD\_AES\_MAC。
- library\_version为可选配置，主要用于定制化对接，当前仅支持配置为SJJ1994。
- kek\_algo要与创建CMK密钥时指定的KEY\_PATH匹配，即均为AES系列算法或SM4系列算法。
- 目前支持gsq和JDBC驱动。

基于鲲鹏TEE内生密钥管理：

```
gaussdb=# \key_info keyType=sdf_kms,library_path={动态库的路径},mac_algo=SGD_SM4_MAC,dkey_algo=SGD_SM4_CBC
```

示例：\key\_info KeyType=sdf\_kms,library\_path=./libsdf.so,mac\_algo=SGD\_SM4\_MAC,dkey\_algo=SGD\_SM4\_CBC

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：《参考》中“SQL参考 > SQL语法 > C > CREATE CLIENT MASTER KEY”章节。

```
gaussdb=# CREATE CLIENT MASTER KEY cmk2 WITH (KEY_STORE = sdf_kms, KEY_PATH = '{密钥加密密钥ID}');
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：《参考》中“SQL参考 > SQL语法 > C > CREATE COLUMN ENCRYPTION KEY”章节。

```
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek1 WITH VALUES (CLIENT_MASTER_KEY = cmk1, ALGORITHM = AES_256_GCM);
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek2 WITH VALUES (CLIENT_MASTER_KEY = cmk2, ALGORITHM = SM4_HMAC_SM3);
```

#### 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
gaussdb=# DROP COLUMN ENCRYPTION KEY cek2;
DROP COLUMN ENCRYPTION KEY

-- 删除主密钥
gaussdb=# DROP CLIENT MASTER KEY cmk1;
DROP CLIENT MASTER KEY
gaussdb=# DROP CLIENT MASTER KEY cmk2;
DROP CLIENT MASTER KEY
```

----结束

## 8.18 DDL

### 简介

数据库的 DDL 功能用于数据库对象的变更。GaussDB支持如下DDL功能：

- 索引操作
- 主键操作
- 列操作
- 生成列操作
- 外键操作
- 表操作
- 分区操作
- 模式操作
- 数据库实例操作
- 表空间操作

按照是否涉及 DDL 目标对象数据的处理，DDL 语句可以划分为两种类型：



- 逻辑 DDL 语句：仅修改数据库对象的元数据，不处理对象存储的数据，例如变更表名或变更列名。
- 物理 DDL 语句：同时修改变更对象的元数据和对象所存储的用户数据。例如，为表创建索引，需要修改表定义并进行全表扫描以构建新索引。

## 原理

- 逻辑DDL语句：因仅需要修改数据库对象的元数据，因此执行时间通常较短，只需要几十毫秒或者几秒。执行这类 DDL 语句几乎不消耗系统资源，因此不会影响业务负载。
- 物理DDL语句：执行时间较长，且执行时间与表的数据量、机器配置以及业务负载有关。离线DDL执行期间加排他锁，数据重组与元数据修改完后提交后释放锁，因此执行期间会阻塞业务执行。物理DDL会长时间阻塞业务，因此GaussDB提供在线DDL在对业务影响较小的情况下进行DDL，减少对业务的影响，提高业务运行的连续性和数据库的可用性。在线DDL不会长时间加排他锁，执行中允许对该表的并发读写操作，减少对用户业务运行影响。

## 示例

本小节介绍GaussDBDDL基本功能，因DDL功能比较多，仅介绍部分常用操作，如列操作和表操作等。

### 步骤1 创建表。

```
gaussdb=# CREATE TABLE t1(c1 INT,c2 INT);
CREATE TABLE
gaussdb=# INSERT INTO t1 VALUES(generate_series(1,10), generate_series(1,10));
INSERT 0 10
```

### 步骤2 增加列。

```
gaussdb=# ALTER TABLE t1 ADD COLUMN c3 INTEGER;
ALTER TABLE
```

### 步骤3 删除列。

```
gaussdb=# ALTER TABLE t1 DROP COLUMN c3;
ALTER TABLE
```

### 步骤4 重命名表。

```
gaussdb=# ALTER TABLE t1 RENAME TO t2;
ALTER TABLE
gaussdb=# ALTER TABLE t2 RENAME TO t1;
ALTER TABLE
```

### 步骤5 修改列类型。

- 在线修改列类型。  
gaussdb=# ALTER TABLE ONLINE t1 ALTER c2 TYPE BIGINT;  
ALTER TABLE
- 离线修改列类型。  
gaussdb=# ALTER TABLE t1 ALTER c2 TYPE INT;  
ALTER TABLE

----结束

## 另请阅读

- 离线DDL另请阅读《参考》中“SQL参考 > SQL语法 > DDL语法一览表”章节。
- 在线DDL另请阅读[在线DDL](#)、《参考》中“SQL参考 > SQL语法 > Online DDL及其操作”章节。

## 8.19 预写式日志

预写式日志（Write Ahead Log，缩写为WAL，也称为Xlog）是一种用于保证数据可靠性和系统可恢复性的技术。其核心思想是：在对数据文件进行修改之前，必须先将这些修改的操作记录写入日志文件，并将日志文件刷新到持久化存储器中。在系统崩溃时，可以使用WAL日志对集群进行恢复操作。WAL日志记录的详细程度分为几个级别，由GUC参数wal\_level控制。

### 原理

- 日志运行机制

GaussDB启动时，会分配一段内存用来缓存数据库运行期间生成的日志，这个缓冲区大小由参数wal\_buffers控制。要确保事务的持久化，Wal\_buffers中的日志就需要落盘。

WAL日志会在以下两种情况下写硬盘：

- Commit事务提交时。系统会调用fdatasync函数，将wal\_buffers中的WAL数据写入WAL日志文件（wal\_sync\_method可控制commit时调用的函数，默认为fdatasync）。
- walWriter定期写。walWriter进程会定期将wal\_buffers中的WAL数据写入WAL日志文件。wal\_writer\_delay参数用来控制walwriter进程工作的时间间隔，默认200ms。如果时间过长可能造成WAL缓冲区的内存不足，时间过短会引起WAL不断写入，增加磁盘I/O负担。

- 日志文件存储路径

路径为数据目录/pg\_xlog。

- CN的WAL日志存储路径为 /<cn数据目录>/pg\_xlog。单个WAL日志文件的大小默认为16MB，通过参数wal\_segment\_size设置，一般不做调整。

```
gaussdb=# SHOW wal_segment_size;
wal_segment_size
```

```

16MB
(1 row)
```

- 以一个DN为例，数据目录可以通过如下命令查询。其中/opt/gaussdb/data/dn/db\_6001代表集群节点的数据目录，当前节点的Xlog日志存放在 /opt/gaussdb/data/dn/db\_6001/pg\_xlog目录。

```
$ cm_ctl query -Cvd
[CMServer State]
```

| node | instance         | state                                  |
|------|------------------|----------------------------------------|
| 1    | 192.168.10.182 1 | /opt/gaussdb/data/cm/cm_server Primary |

```
[Cluster State]
```

```
cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL
```

```
[Datanode State]
```

| node | instance            | state                                  |
|------|---------------------|----------------------------------------|
| 1    | 192.168.10.182 6001 | /opt/gaussdb/data/cm/cm_server Primary |

- 日志文件命名格式

日志文件以段文件的形式存储的，每个段为16MB，并分割成若干页，每页8KB。一个段文件的名称由24个十六进制组成，对WAL日志的命名由如下部分组成：时间线+日志文件标号+日志文件段标号，请参考表8-8。

日志文件的段标号上限为FF，若有下一个WAL日志文件，需要将文件标号+1，段标号归零，例如0000000100000000000000FF的后一个文件为000000010000000100000000。时间线由1开始，日志文件标号和日志文件的段标号由0开始。

表 8-8 WAL 日志文件命名规范

| 字段名称            | 字段含义                   |
|-----------------|------------------------|
| timeline        | 时间线。取值范围：8位16进制数字。     |
| wal_file_num    | 日志文件标号。取值范围：8位16进制数字。  |
| wal_segment_num | 日志文件段标号。取值范围：8位16进制数字。 |

典型的WAL日志文件名如下所示，每个WAL日志文件大小均为16MB。

```
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 11 12:33 0000000100000001000000F2
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 11 18:10 0000000100000001000000F3
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 11 22:18 0000000100000001000000F4
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 12 03:58 0000000100000001000000F5
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 12 07:59 0000000100000001000000F6
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 12 13:45 0000000100000001000000F7
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 12 17:30 0000000100000001000000F8
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 12 23:10 0000000100000001000000F9
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 13 03:10 0000000100000001000000FA
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 13 08:39 0000000100000001000000FB
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 13 12:43 0000000100000001000000FC
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 13 17:08 0000000100000001000000FD
-rW----- 1 gaussdb d gaussdb d 16777216 Jun 13 16:32 0000000100000001000000FE
```

#### 说明

WAL日志文件名的数字一般情况下是顺序增长使用的（要把所有可用数字都用光也需要非常长的时间），但也存在循环使用的情况。

- 日志内容说明

WAL日志的内容取决于记录事务的类型，在系统崩溃时可以利用WAL日志进行恢复。默认配置下，GaussDB每次启动时会先读取WAL日志进行恢复。

- 日志使用场景

- 数据库崩溃恢复：数据库意外崩溃时，因为Xlog日志中已经完整记录了数据库的变化，在数据库重启后就可以通过回放checkpoint之后的Xlog日志，把之前修改过的数据页面（记录在Xlog里，但是还没有来得及修改数据文件）落盘，确保数据的完整性。这个过程是在数据库启动后自动完成的。
- 主备DN数据同步：GaussDB的备DN通过接收来自主DN的Xlog，然后应用Xlog来达到主备一致。主数据库和备数据库之间的数据同步也可以通过Xlog来实现。
- 数据库备份恢复：用GaussRoach对GaussDB做备份时，不仅要复制数据文件，还需要备份期间产生的Xlog，才能构成一个完整的备份。使用全量备份或增量备份+Xlog，可以做到指定时间点的恢复，即恢复到全量备份之后的指定时间点（即PITR：Point-In-Time Recovery）。
- 逻辑复制：逻辑复制工具，如DRS，是通过解析Xlog来获取数据流并进行增量数据同步的。

- Xlog的回收

数据库运行期间，会持续生成Xlog日志，GaussDB通过checkpoint机制来进行Xlog日志清理（通常被称为Xlog回收）。Xlog日志回收相关参数请参见表8-9，相关参数说明及参数设置方式请参见《参考》中“数据库运行参数说明”章节。

表 8-9 Xlog 日志回收相关参数

| 参数名称                           | 参数说明                                                                                                                                                                                                            |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| checkpoint_timeout             | 设置自动检查点之间的最长时间。在上一次自动检查点后，多长时间必须做下一次自动检查点，默认值为15min。如果在两次自动检查点期间堆积的Xlog日志数量大于checkpoint_segments设置的值，也会触发一次自动检查点。                                                                                               |
| checkpoint_segments            | 设置checkpoint_timeout周期内所保留WAL日志文件的最大数量。每个日志文件大小为16MB。                                                                                                                                                           |
| enable_incremental_checkpoint  | 增量检查点开关，默认值为on。                                                                                                                                                                                                 |
| incremental_checkpoint_timeout | 开启增量检查点开关（enable_incremental_checkpoint）之后，设置自动检查点之间的最长时间。<br>默认值为1min。即每分钟会启动一次增量检查点，同样会检查是否有Xlog需要清理。                                                                                                         |
| wal_keep_segments              | Xlog日志文件的数量，设置“pg_xlog”目录下保留日志文件的最小数量。备机通过获取主机的Xlog日志进行流复制。默认值为128。一般情况下，数据库在执行checkpoint时，会检查当前的pg_xlog目录下的Xlog数量是否大于wal_keep_segments设置的值，如果数量大于wal_keep_segments值，就从旧的一端开始清理，保留wal_keep_segments设定数量的Xlog日志。 |
| enable_xlog_prune              | 设置在任一备机断连时，主机是否根据Xlog日志的大小超过参数max_size_for_xlog_prune的值而回收日志，on为开启。在备机断联时，为确保主备DN间Xlog日志是连续的，将不对Xlog日志进行清理。但是，如果开启enable_xlog_prune，且Xlog堆积的容量大于max_size_for_xlog_prune指定的值，一般为256GB，同样会触发Xlog日志清理。             |

| 参数名称                    | 参数说明                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| max_size_for_xlog_prune | <p>在备机故障时主机保留的Xlog最大数量。在enable_xlog_prune打开时生效，工作机制如下：</p> <ol style="list-style-type: none"><li>1. replconninfo系列guc参数配置的所有备机都连接主机时，则该参数不生效。</li><li>2. replconninfo系列guc参数配置的备机断连时，则该参数生效。当主机Xlog日志量大于该参数值，会强制回收。例外：在同步提交模式下（即synchronous_commit参数不是local/off时），如果存在连接中的备机，则主机会考虑保留多数派备机中最小日志接受位置以后的日志，这种情况下，保留的日志可能多于max_size_for_xlog_prune参数值。</li><li>3. 若存在build中的备机，则该参数不生效，主机日志会全量保留，防止build操作期间由于日志被回收引发的失败。</li></ol> |

## 示例

- 查询单个WAL日志文件大小：

```
gaussdb=# SHOW wal_segment_size;
wal_segment_size

16MB
(1 row)
```

- 查询WAL日志数据目录：

```
$ cm_ctl query -Cvd
[CMServer State]

node instance state

1 192.168.10.182 1 /opt/gaussdb/data/cm/cm_server Primary

[Cluster State]

cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL

[Datanode State]

node instance state

1 192.168.10.182 6001 /opt/gaussdb/data/dn/dn_6001 P Primary Normal
```

## 8.20 极致 RTO

极致RTO（Recovery Time Objective，恢复时间目标）功能是通过充分利用服务器的多核CPU资源，建立多级日志回放流水线，使备机日志回放的并发度提高到页面级，实现回放加速的高可用功能。

GaussDB数据库的主机通过发送WAL日志给备机回放，实现主备同步。当业务压力过大时，如果备机的回放速度跟不上主机的速度，在系统长时间的运行后，备机上就会出现日志累积。此时当主机故障后，数据库恢复提供服务能力需要很长时间，影响系

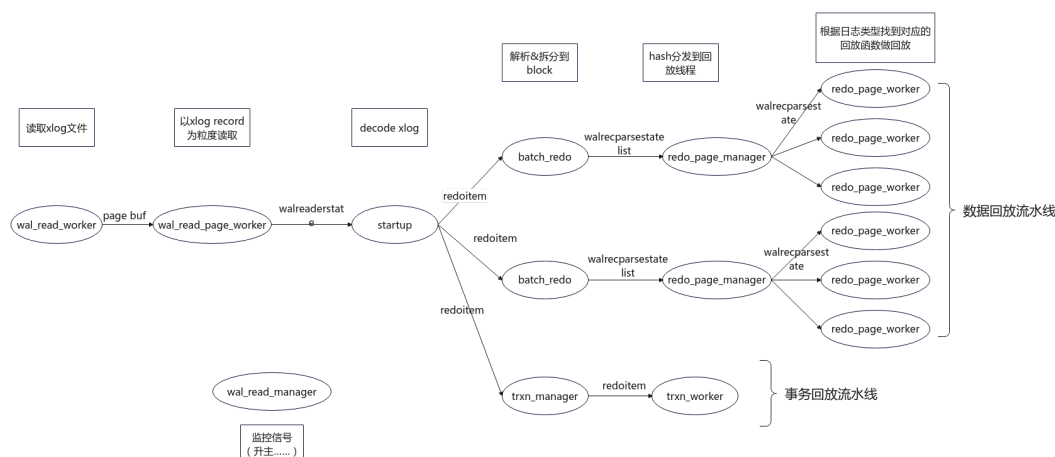
统可用性。在硬件资源充足的情况下，开启极致RTO特性，可以提高备机回放速度，减少主机故障后数据库的恢复时间，提高系统的可用性。

极致RTO功能可通过设置参数`recovery_parse_workers`和`recovery_redo_workers`开启，`recovery_parse_workers`代表开启的数据回放流水线数量，`recovery_redo_workers`代表每条数据回放流水线上启动的数据回放线程数量。如上两个参数中有一个大于1即为开启极致RTO。

## 原理

极致RTO特性是通过充分利用服务器的多核CPU资源，采用多级流水线架构实现回放加速的高可用功能。其主要原理是流水线上游的三个线程分别负责从硬盘以字节流的形式读取日志到缓存、将字节流解析成逐条Xlog、解析Xlog语义并进行分发，实现了对日志的高效并行预处理。流水线下游分为多条数据回放流水线和一条事务回放流水线，对数据页面日志和事务日志的回放实现了并行化。在对数据页面日志进行分发时，先以表级粒度进行拆分和分发到不同的数据回放流水线，然后再按照页面粒度拆分和分发给不同的回放线程进行回放。分发的环节通过利用相关哈希算法，保障同一张表的日志由同一条流水线回放，表上同一个页面的日志由同一个数据回放线程回放。由此，实现了将备机日志回放的并发度提高到页面级，并可通过设置参数扩展流水线和回放线程的数量，提高回放速度和回放并发度。架构图如**图8-2**所示。

图 8-2 极致 RTO 架构示意图



## 示例

**步骤1** 完成GaussDB集群的安装部署。

**步骤2** 完成数据库安装后，通过修改GUC参数`recovery_parse_workers`和`recovery_redo_workers`开启极致RTO回放模式，具体操作请参见《参考》中“数据库运行参数说明 > GUC参数说明 > 预写式日志 > 日志回放”章节。  
`recovery_parse_workers`代表开启的数据回放流水线数量，`recovery_redo_workers`代表每个数据回放流水线的回放线程数量。参数修改后需要重启数据库后才可生效。

**步骤3** 在主机执行DML业务测试。然后通过`cm_ctl query -rv`查询备机回放状态。其中`speed`字段表示此时的回放速度，`delay`字段表示当前备机落后主机的时延。

```
node : 3
instance_id : 6003
node_ip : 8.92.5.241
data_path : /data/omm/data/slave12
type : Datanode
instance_state : Standby
```

```
is_by_query : 1
primary_flush_location : 00000023/5D6A4998
standby_received_location : 00000023/5D6A1560
standby_replay_location : 00000023/5AE12260
static_connections : 2
HA_state : Normal
reason : Normal
redo_start_location : 00000022/5D9EB3D0
min_recovery_location : 00000023/5569D660
read_location : 00000023/5D66F6D0
last_replayed_end_location : 00000023/5AE12260
recovery_done_location : 00000000/00000000
local_max_lsn : 00000023/5D6743A8
read_xlog_io_counter : 32253
read_xlog_io_total_dur : 3840586
read_data_io_counter : 1649089
read_data_io_total_dur : 168654452
write_data_io_counter : 0
write_data_io_total_dur : 0
process_pending_counter : 0
process_pending_total_dur : 0
apply_counter : 5
apply_total_dur : 16095
speed(est.) : 78664 KB/s
delay(est.) : 0 s
senderPercent : 0%
receiverPercent : 99%
worker_info :
id q_use q_max_use rec_cnt
0 42 0 23119337
10 41 0 12575866
```

----结束

## 注意事项

- 极致RTO功能可通过设置参数`recovery_parse_workers`和`recovery_redo_workers`开启。`recovery_parse_workers`代表开启的回放流水线数量，`recovery_redo_workers`代表每个流水线的worker数量。如上两个参数中有一个大于1即为开启极致RTO。
- 打开极致RTO后，备机会额外启动的回放线程总数=`recovery_parse_workers * (recovery_redo_workers + 2) + 5`。回放线程越多，占用CPU、内存和I/O资源越多。请根据实际硬件配置合理设置参数，避免因参数设置过大，导致CPU和内存占用过高，进而导致系统启动异常。

## 另请阅读

极致RTO相关参数描述参见《参考》中“数据库运行参数说明 > GUC参数说明 > 预写式日志 > 日志回放”章节中`recovery_parse_workers`和`recovery_redo_workers`参数描述。

## 8.21 DCF

### 简介

#### 概念

DCF ( Distributed Consensus Framework ) 分布式一致性框架，是一款自研的高性能、高度成熟可靠、易扩展、易使用的独立基础库，其他系统通过API接口可方便地集



成DCF组件。DCF基于Paxos协议实现，解决分布式一致性问题，提升集群高可靠高可用能力。

## 作用和价值

DCF常用于数据库高可用场景，保证数据库多副本日志存储的一致性，以及对外提供高可用能力；同时，DCF具备自选主能力，单点故障时RTO小。

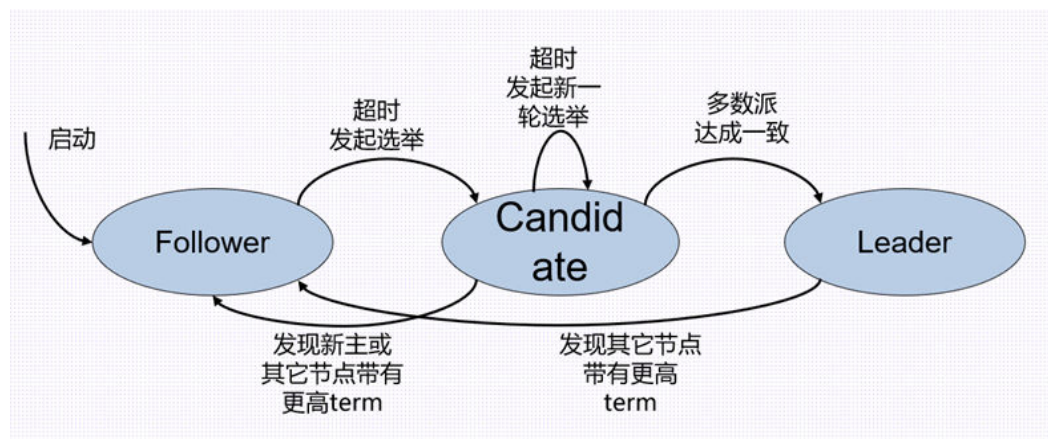
## DCF与Quorum的对比

- Quorum：分布式系统中确保数据一致性和可靠性的机制，主要用于多副本存储系统。无法自选主自仲裁，当出现日志分叉时，需要做build恢复。
- DCF：基于Paxos协议开发的分布式一致性框架，具备自选主自仲裁逻辑，主机故障时备机自动感知，快速选主，RTO小；事务提交、回放等操作前都进行一致性控制，在节点故障、主备切换时不会出现日志分叉问题，避免了不必要的build。

## 原理

## DCF自选主基本原理

图 8-3 DCF 自选主原理示意图



如图8-3所示，DCF自选主有Follower，Candidate和Leader三种基础角色。

**超时驱动：**Leader每隔Heartbeat Timeout发送心跳，Follower超过Election Timeout未收到主机心跳时发起选举，变为Candidate状态。

**随机开始:** 在Election Timeout基础上叠加随机值, 降低选举碰撞概率。

**选举动作：**term增加，广播投票消息，变为Candidate状态，根据日志判断是否投票，获得超过半数票当选Leader。

**正确性：**同一term最多形成一个Leader，通过严格数学证明防止脑裂。

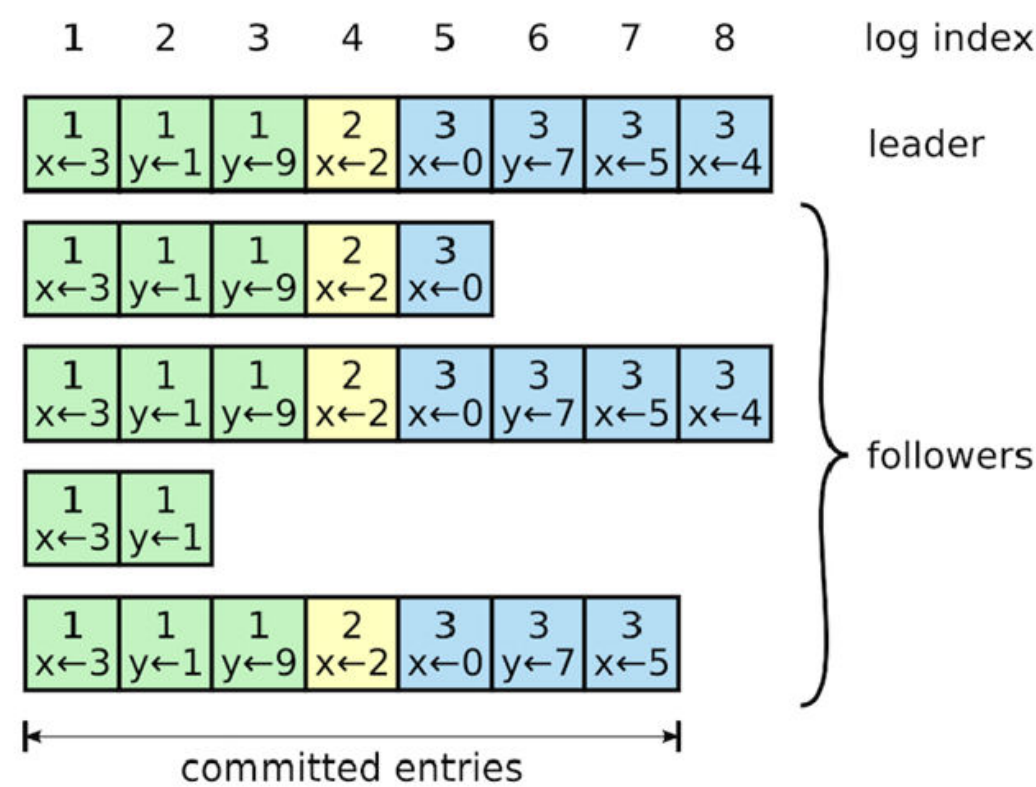
 **说明**

term相当于分布式系统的"逻辑时钟"，用于解决节点间的leadership竞争、日志一致性校验和过期信息识别等问题。

## DCF日志复制基本原理



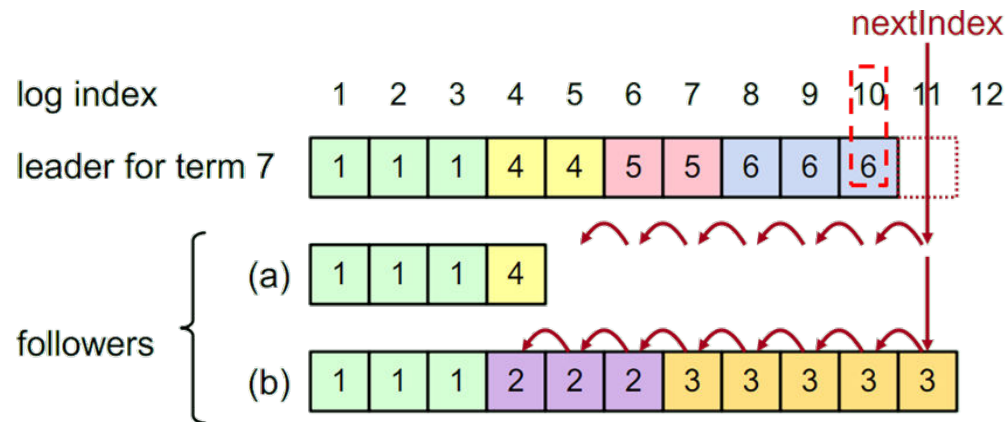
图 8-4 DCF 日志复制原理示意图 1



如图8-4所示，主机产生日志，复制到备机，日志达成多数派一致后提交，最终每个节点日志相同，保证强一致。

- 每条日志包含log term和log index。
- 某个值如果被大多数节点写入日志且在当前任期内则日志可以被提交（任期的概念见上文说明中的term描述）。
- 已经提交的日志不可更改，可以应用到上层调用者。

图 8-5 DCF 日志复制原理示意图 2



当出现主备日志不匹配的情况时DCF会进行自动修复。图8-5中描述了三个节点，term为7的Leader节点和两个不同日志状态的Followers节点。

对于follower(a)节点，日志在index为4之后缺失，index1-4的日志和主机任期完全一致。主机会从初始nextIndex=10给follower(a)同步日志，但是follower(a)没有该位置的日志，会返回日志不匹配信息。主机收到响应后，会将nextIndex减一（变成9），继续上述的同步过程，直到找到nextIndex=5时，向follower(a)同步后续日志。

对于follower(b)节点，日志与主机存在冲突。主机从nextIndex=10开始持续向follower(b)同步日志，直到nextIndex=3时，备机的日志才与主机该index的日志任期一致。此时，主机会从nextIndex=4开始，向follower(b)同步自己的日志，同时覆盖follower(b)index4之后的冲突日志。

该过程可概括为以下三点：

- Leader找到Follower日志不匹配的起始点。
- 对不匹配的节点日志进行重写。
- 最终保证所有节点的日志一致。

## 示例

- DCF的profile日志中选举、复制相关信息如下：

图 8-6 DCF 选举、复制相关信息示例

| [REP] | stream_id | node_id | pause | role | leader | term | commit_index | match_index | next_index |
|-------|-----------|---------|-------|------|--------|------|--------------|-------------|------------|
| [REP] | 1         | 1       | 0     | -    | -      | -    | -            | (0,0)       | 0          |
| [REP] | 1         | 2       | 0     | -    | -      | -    | -            | (2,5065)    | 5066       |
| [REP] | 1         | 3       | -     | 1    | 3      | 2    | (2,5065)     | (2,5065)    | 5066       |

stream\_id：日志流id，默认为1。

node\_id：节点id。

pause：对节点是否开启流控，0表示未开启，非0表示开启。

role：1：leader，2：follower，3：logger，4：passive，5：pre\_candidate，6：candidate，7：cascade\_follower。

term：当前节点的term。

commit\_index：当前节点的一致性index。

match\_index：当前节点收到的DCF日志的index。

next\_index：当前节点期望主节点发送的DCF日志的index。

- DCF的profile日志中存储相关信息如下：

图 8-7 DCF 存储相关信息示例

| [STG]:stm id | stm first | stm last      | stg first | stg last      | applied       | cache begin | [stg_manager.c:173] |
|--------------|-----------|---------------|-----------|---------------|---------------|-------------|---------------------|
| [STG]: 1     | 1         | 5065          | 1         | 5065          | 5065          | 1           | [stg_manager.c:182] |
| [STG]: 1     | 0         | 3001300000001 | 0         | 3001300000001 | 3001300000001 | 0           | [stg_manager.c:191] |

stm id：日志流id，默认为1。

stm first：日志流的第一条日志位置，调用dcf\_truncate时数值会进行更新。

stm last：日志流的最后一条日志位置。

stg first：DCF数据目录下已存储的第一条日志，调用dcf\_truncate时数值会进行更新。

stg last：DCF数据目录下已存储的最后一条日志。

applied：DCF通知DN应用的日志信息位置（DN更新consensus lsn的位置信息）。

cache begin：存储模块日志消息缓存cache数组的第一条日志点位，cache在日志apply后或者达到阈值后会触发回收，一般cache begin大于stm first，stm last - cache begin表示cache数组内存储的日志个数。

## 另请阅读

《参考》中“数据库运行参数说明 > GUC参数说明 > DCF参数设置”章节。

## 8.22 CM

### 简介

CM（Cluster Manager）是一款数据库资源管理与监控组件，提供高可用、自动化的实例管理和故障恢复能力，保障集群的稳定运行。

GaussDB提供了cm\_ctl工具用来控制集群服务。

#### 核心组件

- **CMS（cm\_server）**
  - 是用来进行集群管理和实例仲裁的组件。
  - 功能：
    - 接收各个节点上CMA发送的集群各实例状态。
    - 监控实例的状态变化并进行仲裁命令的下发。
    - 提供集群整体状态查询接口。
  - 高可用特性：支持主备或一主多备配置，当主CMS出现故障时自动切换到备CMS接管业务继续执行。
- **CMA（cm\_agent）**
  - 是部署在集群每个主机上，用来启停和监控各个数据库实例进程的集群管理组件。
  - 功能：
    - 启停本机数据库实例。
    - 监控实例状态并上报给CMS。
    - 执行CMS下发的仲裁命令。
- **OMM（om\_monitor）**
  - 看护cm\_agent的定时任务组件。
  - 功能：
    - 看护cm\_agent进程，定时检查其运行状态。
    - 在cm\_agent进程停止时自动重启，确保监控不中断。
- **DDB（distribute database）**
  - 用于CMS自仲裁选主和持久化存储仲裁数据的管理服务组件。
  - CMS可以选择使用ETCD或DCC。

- ETCD：高度一致的分布式键值存储组件。
- DCC：分布式配置中心组件，基于DCF（Distributed Consensus Framework）实现，DCF采用Paxos协议来保证高可用与一致性。

## 作用

- **统一管理**  
提供集群、节点、实例的统一启停、查询、切换功能，提升管理效率。
- **实时监控**  
多维度监控节点、实例、网络通信、文件系统等资源的运行状态，有问题可及时发现。
- **灵活扩展**  
支持自定义资源监控和故障检测规则，适应不同业务场景。
- **自动化运维**  
提供故障自动主备切换等能力，降低人工运维负担。
- **高可用性保障**  
主备cm\_server架构，自动故障接管，避免单点故障，提升系统可用性。

## 示例

- 查询集群状态。

```
cm_ctl query -Cvd
```

```
[CMServer State]
```

| node | instance          | state                                                  |
|------|-------------------|--------------------------------------------------------|
| 1    | kwephis32851484 1 | /var/standard/usr/local/cm/cm_server/cm_server Primary |

```
[Cluster State]
```

```
cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL
```

```
[Coordinator State]
```

| node | instance             | state                             |
|------|----------------------|-----------------------------------|
| 1    | kwephis32851484 5001 | /var/standard/usr/local/cn Normal |

```
[Central Coordinator State]
```

| node | instance             | state                             |
|------|----------------------|-----------------------------------|
| 1    | kwephis32851484 5001 | /var/standard/usr/local/cn Normal |

```
[GTM State]
```

| node | instance             | state                                 |
|------|----------------------|---------------------------------------|
| 1    | kwephis32851484 1001 | /var/standard/usr/local/gtm P Primary |

```
[Datanode State]
```

| node | instance             | state                                                              |
|------|----------------------|--------------------------------------------------------------------|
| 1    | kwephis32851484 6001 | /var/standard/var/lib/engine/data1/data/datanode1 P Primary Normal |
| 1    | kwephis32851484 6002 | /var/standard/var/lib/engine/data1/data/datanode2 P Primary Normal |

- 检测进程运行状态。  
-- 检测gaussdb进程运行状态，/var/standard/var/lib/engine/data1/data/datanode1为指定实例的数据目录，可以通过cm\_ctl query -Cvd查询获得。  
cm\_ctl check -B gaussdb -T /var/standard/var/lib/engine/data1/data/datanode1  
  
-- 执行如下命令，查看上一个命令的返回值，2表示进程存在。  
echo \$?  
2
- 设置CM相关参数。  
-- 设置cm\_agent参数log\_file\_size值为20。  
cm\_ctl set --param --agent -k "log\_file\_size=20"  
cm\_ctl: set cm\_agent.conf success.  
  
-- 动态加载cm\_agent参数。  
cm\_ctl reload --param --agent  
cm\_ctl: reload cm\_agent.conf success.
- 查询CM相关参数。  
cm\_ctl list --param --agent  
  
[conf of node(1)]  
log\_dir = /var/standard/var/log/Mike/Mike/cm/cm\_agent  
log\_file\_size = 20  
log\_min\_messages = WARNING  
incremental\_build = on  
security\_mode = off  
upgrade\_from = 0  
alarm\_component = /opt/huawei/snas/bin/snas\_cm\_cmd  
.....

## 另请阅读

《参考》中“日志参考 > 系统日志”章节

《参考》中“数据库运行参数说明 > CM参数说明”章节

## 8.23 调用堆栈

### 简介

调用堆栈（Call Stack）是程序运行时由操作系统和编译器协同管理的核心机制，基于栈帧（Stack Frame）结构与硬件级栈指针实现，用于动态追踪函数调用层次与执行状态。在数据库运维场景中，调用堆栈的全链路可视化能力（如GaussDB的gs\_stack()函数）可实时捕获线程级函数调用链，精确定位死锁、事务阻塞等高复杂度问题，相较于传统gdb分析方式，其毫秒级响应、线程级精细诊断及无中断服务特性，大幅提升运维效率，缩短故障修复时间。

#### 说明

数据结构栈和调用堆栈的差异：

- 数据结构栈：纯逻辑模型，基于后进先出（LIFO）原则，仅提供入栈（push）、出栈（pop）等基础操作，无运行时上下文管理功能。
- 调用堆栈：物理内存实现，除基础栈操作外，还依赖编译器生成栈帧布局、操作系统分配连续内存，并利用寄存器（如ESP）动态跟踪调用层级，集成上下文保存/恢复机制，直接支撑程序的函数嵌套、递归等复杂逻辑。

### 原理

堆栈基于栈帧（Stack Frame）机制与硬件级栈指针支持，在程序运行时由操作系统和编译器协同管理。每个函数调用都会在运行时堆栈中创建一个独立的栈帧，用于存储

该函数的局部变量、形参、返回地址以及调用上下文信息。当函数执行完毕后，其对应的栈帧被自动弹出，控制权返回至调用者，实现状态的无缝恢复。堆栈的底层实现通常采用连续内存区域（如数组）或动态链表结构，通过维护一个指向栈顶的指针（如ESP寄存器在x86架构中）实现高效的入栈（push）与出栈（pop）操作。此外，现代处理器提供了专门的指令支持（如PUSH、POP），进一步提升了堆栈操作的性能与可靠性。在递归调用中，每层递归都会新增一个栈帧，直到达到终止条件，从而天然支持函数调用的嵌套与回溯逻辑。

基于运行时堆栈的精准采集能力，GaussDB通过引入gs\_stack()函数和gs\_ctl stack工具，构建了从SQL层到内核线程级的全链路调用栈可视化机制。该机制依托于操作系统级线程调度与信号处理框架，支持在数据库运行态下动态捕获任意线程的函数调用轨迹，实现对死锁、hang、长事务阻塞等疑难问题的快速定位。其核心原理在于：在满足信号注册（如SIGURG）与线程可响应状态的前提下，利用调试符号表与栈帧回溯技术，将底层汇编调用链还原为可读的函数调用路径，最高支持128层栈深度，覆盖典型业务场景中的复杂嵌套调用。

相较于依赖gdb（如pstack工具）的调用链分析方式，该机制具备三大优势：一是实时性高，无需中断服务或改变父进程状态即可获取当前运行状态；二是环境兼容性强，以数据库原生工具形式集成，无需安装额外依赖；三是运维粒度灵活，支持通过SQL指令批量获取多个线程的调用栈，而无需逐线程手动操作，并自动关联事务上下文，降低操作复杂度。

此外，GaussDB在实现上充分考虑生产环境约束：保留原始符号信息（未strip），确保调用栈可读性；通过限制并发获取以避免对系统造成额外负载；并在多线程场景下提供统一输出格式，便于自动化分析工具集成。整体机制已成为数据库内核故障排查的关键基础设施，使运维人员能够在毫秒级响应中实现从现象到根因的闭环定位，大幅缩短MTTR（平均修复时间）。

示例

gs\_stack()函数使用方式请参见《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。

- SELECT \* FROM gs\_stack(pid)获取指定线程的调用栈。  
gaussdb=# SELECT \* FROM gs\_stack(140518384408320); --- 获取指定pid的线程调用栈  
gs\_stack  
-----  
unknown\_symbol +  
poll + 0x2d +  
WaitLatchOrSocket(Latch volatile\*, int, int, long) + 0x307 +  
sys\_logger\_main(int) + 0xfb4 +  
int gauss\_db\_thread\_main<(knl\_thread\_role)17>(knl\_thread\_arg\*) + 0x5ab+  
internal\_thread\_func(void\*) + 0x2d +  
ThreadStarterFunc(void\*) + 0x74 +  
unknown\_symbol +  
\_\_clone + 0x6d +  
(1 row)
- SELECT \* FROM gs\_stack()获取所有线程的调用栈。  
gaussdb=# SELECT \* FROM gs\_stack(); --- 获取所有线程的调用栈  
tid | lwtid |  
stack  
-----+-----  
-----  
140519906720128 | 6750 |  
unknown\_symbol  
+  
| | poll +  
0x2d +

|                |   |                                                                                  |   |
|----------------|---|----------------------------------------------------------------------------------|---|
|                |   | CommWaitPollParam::caller(int (*)(pollfd*, unsigned long, int), unsigned long) + |   |
| 0x34           |   | +                                                                                |   |
|                |   | int comm_socket_call<CommWaitPollParam, int (*)(pollfd*, unsigned long,          |   |
|                |   | int)>(CommWaitPollParam*, int (*)(pollfd*, unsigned long, int)) + 0x28           | + |
|                |   | comm_poll(pollfd*, unsigned long, int) +                                         |   |
| 0xb1           |   |                                                                                  | + |
|                |   | server_loop() +                                                                  |   |
| 0xa33          |   |                                                                                  | + |
|                |   | postmaster_main(int, char**) +                                                   |   |
| 0x4214         |   |                                                                                  | + |
|                |   | main +                                                                           |   |
| 0x53c          |   |                                                                                  | + |
|                |   | __libc_start_main +                                                              |   |
| 0xf5           |   |                                                                                  | + |
|                |   |                                                                                  |   |
| unknown_symbol |   |                                                                                  |   |
|                | + |                                                                                  |   |
|                |   |                                                                                  |   |
|                |   | 140518183073536   6817                                                           |   |
| unknown_symbol |   |                                                                                  |   |
|                | + |                                                                                  |   |
|                |   | poll +                                                                           |   |
| 0x2d           |   |                                                                                  | + |
|                |   | WaitLatchOrSocket(Latch volatile*, int, int, long) +                             |   |
| 0x307          |   |                                                                                  | + |
|                |   | WaitLatch(Latch volatile*, int, long) +                                          |   |
| 0x2e           |   |                                                                                  | + |
|                |   | AlarmCheckerMain() +                                                             |   |
| 0x2e1          |   |                                                                                  | + |
|                |   | int gauss_db_thread_main<(knl_thread_role)20>(knl_thread_arg*) +                 |   |
| 0x612          |   |                                                                                  | + |
|                |   | internal_thread_func(void*) +                                                    |   |
| 0x2d           |   |                                                                                  | + |
|                |   | ThreadStarterFunc(void*) +                                                       |   |
| 0x74           |   |                                                                                  | + |
|                |   |                                                                                  |   |
| unknown_symbol |   |                                                                                  |   |
|                | + |                                                                                  |   |
|                |   | __clone +                                                                        |   |
| 0x6d           |   |                                                                                  | + |
|                |   |                                                                                  |   |
|                |   | 140518384408320   6816                                                           |   |
| unknown_symbol |   |                                                                                  |   |
|                | + |                                                                                  |   |
|                |   | poll +                                                                           |   |
| 0x2d           |   |                                                                                  | + |
|                |   | WaitLatchOrSocket(Latch volatile*, int, int, long) +                             |   |
| 0x307          |   |                                                                                  | + |
|                |   | sys_logger_main(int) +                                                           |   |
| 0xfb4          |   |                                                                                  | + |
| --More--       |   |                                                                                  |   |

注意事项

gs\_ctl stack -D基于SIGUSR1信号实现通信，在GaussDB进程启动过程中，尚未注册SIGUSR1信号的处理函数时，外部调用gs\_ctl stack -D会导致GaussDB进程采用默认的SIGUSR1处理函数，进而引发进程直接退出。

另请阅读

《参考》中“SQL参考 > 函数和操作符 > 统计信息函数”章节。

## 8.24 等待事件

### 简介

等待事件是指数据库中的进程或线程在执行过程中，由于某些资源不可用或需要等待某些条件满足而暂时无法继续执行的状态。

当一个事务或查询需要获取某个锁、读写磁盘、等待远程节点响应或访问共享内存结构时，就会进入“等待”状态。这些等待虽然短暂，但如果频繁发生或单次耗时过长，就会显著拖慢整体吞吐量。传统的性能监控往往只能看到某条 SQL 执行较慢，却无法定位慢的原因，这正是等待事件功能诞生的初衷。

GaussDB通过引入完善的等待事件体系，不仅记录了是否发生等待事件，还精确统计了“等待时长、等待次数及最长/最短/平均等待时间”，并支持按实例、线程、SQL 语句三个级别进行聚合分析，极大提升了系统可观测性和问题定位效率。总的来说，当客户作业不满足时延要求或者不满足客户预期时，等待事件功能可以方便运维人员及开发人员了解系统整体以及每个线程更详细的执行状态，辅助异常定位排查和系统优化。

### 原理

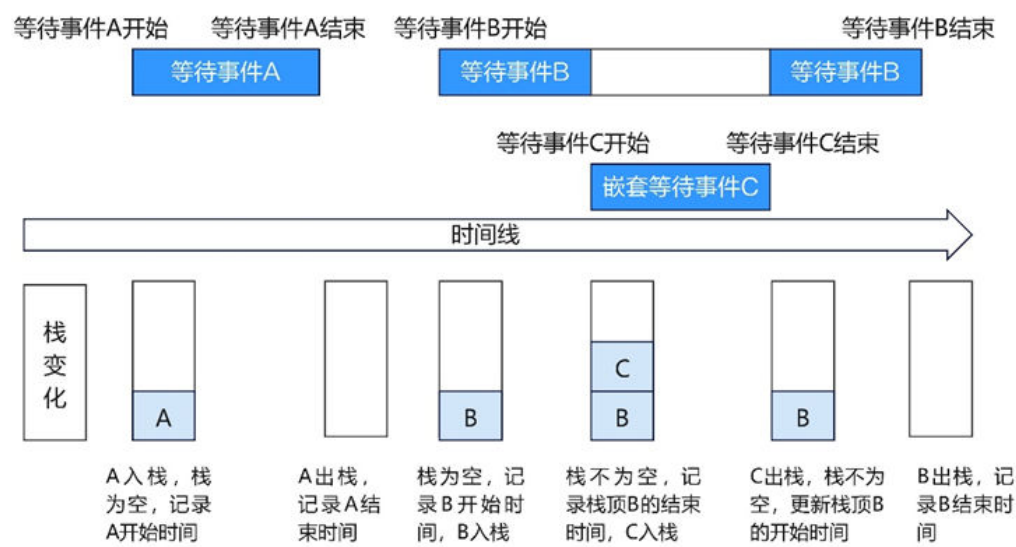
等待事件的设计理念是将等待行为视为一种可追踪、可聚合的时间片段。由于等待事件存在嵌套关系，因此通过栈式结构实现嵌套关系的自动上报与归因分析。整个等待事件的记录与统计依托于一个线程本地的等待事件栈，每一个工作线程（backend thread）在运行期间都维护着这样一个栈结构，用于跟踪当前正在发生的等待链路，等待事件功能原理如图8-8所示。

当线程进入一个需要等待的操作时（例如申请锁、读写文件、等待远程响应），系统会将当前操作所归属的等待事件压入栈顶。若此时栈中已有其他等待事件存在，则将栈顶等待事件的起始时间、等待时长等信息记录下来，并计入全局统计视图。

当等待事件结束，线程退出该等待状态时，栈顶事件被弹出，系统会记录该事件的完整耗时。若栈中仍有剩余事件，则将栈顶事件的开始时间更新为当前时刻，表明它仍在持续等待，从而维持等待链的连续性。



图 8-8 等待事件原理示意图



说明

- 等待事件可分为以下四类：
- STATUS：表示执行关键代码段的等待，常见于耗时较长的同步操作或资源协调，例如：等待应用侧发数据（STATUS/wait cmd）、Pooler CN向DN建连（STATUS/pooler create conn）、GTM获取快照、SQL算子执行等。
  - IO\_EVENT：表示I/O操作的等待，当查询涉及大量数据扫描或大表访问时，该类等待事件值较大，典型场景包括读写HEAP文件、写WAL日志等。
  - LWLOCK：用于保护内存结构的锁的等待，用于保护全局共享的内存结构，如全局共享的HASH表，常见于等待分区锁等场景。
  - LOCK：数据库内部对象锁的等待，涵盖relation锁、tuple锁、事务锁等，当出现大量并发更新时，该类等待事件值较大。

示例

- 查看当前节点的等待事件统计信息。  
WAIT\_EVENTS使用详解，请参见《参考》中“Schema > DBE\_PERF Schema > Wait Events > DBE\_PERF.WAIT\_EVENTS”章节。  
--查看当前节点等待事件统计，包括每一类型等待事件的累计等待时间（total\_wait\_time）和平均等待时间（avg\_wait\_time）等。  
gaussdb=# \x  
Expanded display is on.  
gaussdb=# SELECT nodename, type, event, wait, total\_wait\_time, avg\_wait\_time, last\_updated FROM dbe\_perf.wait\_events ORDER BY total\_wait\_time DESC;  
-[ RECORD 1 ]-----  
nodename | coordinator1  
type | STATUS  
event | wait node  
wait | 393763  
total\_wait\_time | 342584597958  
avg\_wait\_time | 870027  
last\_updated | 2025-09-08 15:49:07.170297+08  
-[ RECORD 2 ]-----  
nodename | coordinator1  
type | STATUS  
event | wait cmd  
wait | 351923  
total\_wait\_time | 1438714213

```
avg_wait_time | 4088
last_updated | 2025-09-08 15:49:08.257995+08
-[RECORD 3]---+-----
--More--
```

- 查看当前节点中各线程的等待阻塞情况。

PG\_THREAD\_WAIT\_STATUS使用详解，请参见《参考》中“系统表和系统视图 > 系统视图 > 其他系统视图 > PG\_THREAD\_WAIT\_STATUS”章节。

--查看当前节点中各线程的等待阻塞情况，wait\_status字段表示当前线程正在等待的事件类型，例如wait\_status字段为acquire lwlock时，表示正在等待获取轻量级锁、事务锁。

```
gaussdb=# \x
Expanded display is on.
gaussdb=# SELECT node_name, db_name, thread_name, tid, query_id, sessionid, lwtid, wait_status
FROM pg_thread_wait_status;
-[RECORD 1]-----
node_name | coordinator1
db_name |
thread_name | jit compile worker
tid | 140601850853120
query_id | 0
sessionid | 140601850853120
lwtid | 56358
wait_status | none
-[RECORD 2]-----
node_name | coordinator1
db_name |
thread_name | PageWriter
tid | 140603134310144
query_id | 0
sessionid | 140603134310144
lwtid | 56261
wait_status | none
-[RECORD 3]-----
--More--
```

- 查看FULL/SLOW SQL在执行过程中的等待事件。

STATEMENT\_HISTORY使用详解，请参见《参考》中“系统表和系统视图 > 系统表 > 其他系统表 > STATEMENT\_HISTORY”章节。

--查看FULL/SLOW SQL在执行过程中的发生的所有等待事件及其耗时。

```
gaussdb=# \x
Expanded display is on.
gaussdb=# SELECT query, unique_query_id, pg_catalog.statement_detail_decode(details, 'plaintext',
true) FROM pg_catalog.statement_history;
-[RECORD 1]-----query | select
gs_perf_do_shaking_collect(?);
unique_query_id | 347666798
detail | -----Wait Events Area-----
 | '1' STATUS wait xact start command 139 (us)
 | '2' STATUS gtm get snapshot 125 (us)
 | '3' STATUS wait xact commit command 121 (us)
 | '4' IO_EVENT BufHashTableSearch 26 (us)
 | '5' STATUS flush data 11 (us)
-[RECORD 2]-----
--More--
```

## 注意事项

所有等待事件统计数据均保存在内存中的累积视图中，属于运行时状态，节点重启后清空。

## 8.25 备份恢复

### 简介

备份是通过复制数据库文件到指定的备份位置的过程，主要为了防止因硬件故障、人为误删、勒索攻击等场景数据丢失，从而可以更好地保护数据安全。恢复是将备份数据还原到数据库系统的过程。

#### 备份方式分类：

- 物理备份：是指复制数据库的物理文件到指定的备份位置，不涉及数据库内部逻辑结构，备份速度快，但备份文件不可读，适合大规模数据库。
- 逻辑备份：是指将数据库中的逻辑对象（表、索引、触发器等）以逻辑关联的格式导出。备份文件可读性好，方便查看，但备份速度较慢。

#### 恢复方式的分类：

- 物理恢复：把原数据库中复制的物理数据文件恢复到新数据库中的过程。
- 逻辑恢复：把数据库中导出的数据文件（SQL语句或可读的文本格式）再导入原来的数据库中的过程。

#### 备份恢复的作用：

- 数据安全

防止因硬件故障、误操作或勒索攻击导致的数据丢失，保障数据完整性。

- 业务连续性

通过快速恢复减少停机时间，满足金融、医疗等行业的高可用要求。

- 合规性

支持数据长期保留与审计，符合行业监管要求。

### 示例

全量物理备份数据库。

```
--准备工作。
cd $GPHOME/script #切换至roach工具路径。

--查询集群状态。
cm_ctl query -Cv
[CMServer State]

node instance state

1 pekphis351063 1 Primary
2 pekphis351064 2 Standby
3 pekphis351065 3 Standby

[Cluster State]

cluster_state : Normal
redistributing : No
balanced : Yes
current_az : AZ_ALL
...
```

只有集群状态正常时，备份才有意义，从上面回显可以看出，集群状态是正常的。

```
--登录数据库，写入数据。
gaussdb=# create table test(id int);
CREATE TABLE
gaussdb=# insert into test values (GENERATE_SERIES(1,100000));
INSERT 0 100000

--开始备份。
python3 GaussRoach.py -t backup --master-port 6000 --media-destination /home/userA/media --media-
type DISK --metadata-destination /home/userA/metadata
Parsing the configuration file.

Roach master is ready to accept connections. Starting agents now...
Performing presetup activities...

N0 pekphis351063: 100% COMPLETED
N1 pekphis351064: 100% COMPLETED
N2 pekphis351065: 100% COMPLETED

[MASTER]Backup operation SUCCESSFUL. Backup key: 20250819_161344
[MASTER] Time taken : 26s

Successfully send signal to kill all gs_roach.
Cleaning temp files after backup or restore.
Successfully cleaned temp files after backup or restore.
Roach operation backup completed.
```

用户必须设置相关必选参数来保证备份操作的成功完成。

全量物理恢复数据库。

```
--恢复数据库。
python3 GaussRoach.py -t restore --clean --master-port 6000 --media-destination /home/userA/media --
media-type DISK --backup-key 20250819_161344 --metadata-destination /home/userA/metadata
Cleaning cluster.
Parsing the configuration file.
Stopping cluster.
Successfully stopped cluster.
Cleaning data before restoration.
Successfully cleaned data before restoration.
Successfully cleaned cluster.

Roach master is ready to accept connections. Starting agents now...
Performing presetup activities...

N0 pekphis351063: 100% COMPLETED
N1 pekphis351064: 100% COMPLETED
N2 pekphis351065: 100% COMPLETED

[MASTER] Restore SUCCESSFUL.
[MASTER] Time taken : 17s

Cleaning temp files after backup or restore.
Successfully cleaned temp files after backup or restore.
Roach operation restore completed.
```

用户必须设置相关必选参数来保证恢复操作的成功完成。

```
--恢复完成后，启动数据库。
python3 GaussRoach.py -t start
Stopping the cluster.
Successfully stopped the cluster.
Pre-cleaning ETCD before 'restore-new-cluster' startup...
Successfully pre-cleaned ETCD.
Starting the cluster.
Starting DN master.
Successfully started DN master.
Starting CN.
Successfully started CN.
Starting ETCD.
```

```

Successfully started ETCD.
Starting GTM.
Successfully started GTM.
Starting build DN standby.
Successfully start DN standby.
Starting DN dummy standby.
Successfully start DN dummy standby.
Updating catalog phase 1.
Successfully updated catalog phase 1.
Stopping DN instances.
Successfully stop DN instances.
Starting DN master.
Successfully started DN master.
Starting DN standby.
Successfully start DN standby.
Starting CN.
Successfully started CN.
Starting CN.
Successfully started CN.
Updating catalog phase 2.
Successfully updated catalog phase 2.
Starting CN.
Successfully started CN.
Stopping DN instances.
Successfully stop DN instances.
Starting the cluster.
Successfully started primary instance. Please wait for standby instances.
.
Successfully started standby instances.
Waiting for the cluster become normal...
Stopping the cluster.
Successfully stopped the cluster.
Starting the cluster.
Successfully started primary instance. Please wait for standby instances.
.
Successfully started standby instances.
=====
cluster_state : Normal
redistributing : No
node_count : 3
Coordinator State
 normal : 3
 abnormal : 0
GTM State
 primary : 1
 standby : 2
 abnormal : 0
 down : 0
Datanode State
 primary : 3
 standby : 6
 secondary : 0
 building : 0
 abnormal : 0
 down : 0
Successfully started the cluster.
Cleaning backup slot after restore.
Roach operation start completed.
--再次登录数据库验证数据是否完整。
gaussdb=# \d
 List of relations
 Schema | Name | Type | Owner | Storage
-----+-----+-----+-----+-----
 public | test | table | zwb2 | {orientation=row,compression=no,storage_type=USTORE,segment=off}
(1 row)

--查询数据。
gaussdb=# select count(*) from test;
 count

 1

```

|         |
|---------|
| -----   |
| 100000  |
| (1 row) |

另请阅读

若要了解更多的备份恢复相关的操作，另请阅读：

《参考》中“工具参考 > 数据导入导出工具”章节。

# 9 FAQ-非透明多写

## 9.1 数据库系统概述

### 9.1.1 GaussDB 支持哪些硬件平台？

答：GaussDB支持运行在以下硬件平台上：

- 支持基于x86\_64/ARM的通用PC服务器。
- 支持本地存储（SATA、SAS、SSD、弹性云EVS服务）。
- 支持千兆、万兆Ethernet网络。数据库运行环境通常需要确保网络时延稳定，且建议集群低负载时（CPU小于5%），时延50us~70us；集群高负载时（CPU大于90%），时延不超过100us。

### 9.1.2 如何查看当前数据库的隔离级别？

答：事务的隔离级别共三种，分别为：

- READ COMMITTED：读已提交隔离级别，事务只能读已提交的数据而不会读未提交的数据，这是缺省值。
- REPEATABLE READ：事务可重复读隔离级别。
- Serializable：目前功能上不支持此隔离级别，等价于REPEATABLE READ。

使用如下语句查看数据库隔离级别。

```
gaussdb=# SHOW transaction_isolation;
transaction_isolation

read committed
(1 row)
```

### 9.1.3 GaussDB 易用性如何？

答：GaussDB提供了丰富的工具和API，方便客户进行二次开发和集成，也使得开发者可以轻松地使用它来进行数据存储和处理，同时它提供了简单易用的操作界面和管理工具，降低了客户的使用门槛和学习成本等。

### 9.1.4 如何在 GaussDB 当前事务中建立保存点及删除保存点？

答：保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。SAVEPOINT用于在当前事务里建立一个新的保存点。RELEASE SAVEPOINT删除一个当前事务先前定义的保存点。

具体操作请参见如下示例：

```
--创建SCHEMA及表table1。
gaussdb=# CREATE SCHEMA tpcds;
gaussdb=# CREATE TABLE tpcds.table1(a int);

--开启事务。
gaussdb=# START TRANSACTION;

--插入数据。
gaussdb=# INSERT INTO tpcds.table1 VALUES (3);

--建立保存点。
gaussdb=# SAVEPOINT my_savepoint;

--插入数据。
gaussdb=# INSERT INTO tpcds.table1 VALUES (4);

--删除保存点。
gaussdb=# RELEASE SAVEPOINT my_savepoint;

--提交事务。
gaussdb=# COMMIT;

--删除SCHEMA。
gaussdb=# DROP SCHEMA tpcds CASCADE;
```

### 9.1.5 事务的作用是什么？

答：在数据库管理系统中，事务用于管理一组数据库操作，使它们作为一个不可分割的工作单元，要么全部执行成功，要么全部回滚。事务确保数据库的一致性和完整性，并且可以恢复到之前的状态，以防止发生意外或错误情况。

事务的作用包括：

- 保证数据的一致性：事务确保在一组操作中，要么所有操作都成功执行，要么全部回滚，从而保证数据库数据的一致性。
- 确保数据的完整性：事务通过ACID属性（原子性、一致性、隔离性和持久性）来确保数据的完整性，防止数据丢失或损坏。
- 提供隔离性：事务的隔离性确保了多个并发事务之间的隔离，避免了数据交叉访问和干扰，保证了每个事务的独立性。
- 支持原子性：事务要么全部执行成功，要么全部回滚，这种原子性确保了事务的完整性。
- 支持持久性：一旦事务被提交，其对数据库的修改就会持久保存。

### 9.1.6 开启事务的方式有哪些？

答：开启事务方式有两种，BEGIN可以用于开始一个事务，START TRANSACTION也用于开始一个事务。

- BEGIN的用法。  
--建表并插入数据。  
gaussdb=# CREATE TABLE tbl\_test1(col1 int, col2 int);  
gaussdb=# INSERT INTO tbl\_test1 VALUES (1,1), (2,2), (3,3);



- ```
--以默认方式启动事务。
gaussdb=# BEGIN;
INSERT INTO tbl_test1 VALUES (4,4);
END;

--删除表。
gaussdb=# DROP TABLE tbl_test1;
```
- **START TRANSACTION的用法。**

```
--创建SCHEMA。
gaussdb=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
gaussdb=# CREATE TABLE tpcds.reason (c1 int, c2 int);

--以默认方式启动事务。
gaussdb=# START TRANSACTION;
gaussdb=# SELECT * FROM tpcds.reason;
gaussdb=# END;

--删除表tpcds.reason。
gaussdb=# DROP TABLE tpcds.reason;

--删除SCHEMA。
gaussdb=# DROP SCHEMA tpcds;
```

9.1.7 数据库事务正确执行的四个特性是什么？

答：数据库管理系统（DBMS）在写入或更新资料的过程中，为保证事务（transaction）正确执行，必须具备的四个特性ACID，Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）。

- **Atomicity（原子性）：**又称为不可分割性。一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- **Consistency（一致性）：**在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
- **Isolation（隔离性）：**又称为独立性。数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- **Durability（持久性）：**事务处理结束后，对数据的修改是永久的，即便系统故障也不会丢失。

9.1.8 什么是脏读、幻读、不可重复读？

答：脏读、幻读、不可重复读都是事务并发存在的问题。

- **脏读：**A事务读取到B事务修改过但是没有提交的数据，B事务回滚。
- **幻读：**在前后两个时间段内执行对同一个数据的读取，可能出现不一致的结果。
- **不可重复读：**在同一事务中两次查询数据的内容不同。

9.1.9 什么是 MVCC？

答：MVCC，即Multi-Version Concurrency Control（多版本并发控制）。MVCC是数据库并发控制协议的一种，写事务不会立即修改元组内容，每次操作都会在新的版本

之上创建新的版本，并且会保留旧的版本。当某个事务要读取数据时，数据库系统会从所有版本中选取符合该事务隔离级别要求的版本。

MVCC的主要优点是：读数据的锁请求和写数据的锁请求不冲突，可以实现读不阻塞写，写不阻塞读。

9.2 数据库连接管理

9.2.1 如何查询数据库活跃连接数？

答：设置参数track_activities为on，查询视图pgxc_stat_activity。

```
--设置参数track_activities为on。
gaussdb=# SET track_activities = on;

--查询。
gaussdb=# SELECT COUNT(*) FROM pgxc_stat_activity WHERE state='active';
count
-----
      19
(1 row)
```

9.2.2 通过客户端无法连接数据库的原因有哪些？

答：在使用数据库时，经常会通过客户端去连接数据库，只有正确地连接数据库，才能在数据库进行相关的操作，那么在使用数据库过程中遇到无法连接数据库的情况，常见的原因有以下几点：

- 系统连接数据库数量超过了最大连接数量。
- 用户不具备访问该数据库的权限。
- 当前客户端IP信息没有配置进数据库服务器的pg_hba.conf文件中。
- 网络连接故障。

9.2.3 GaussDB 支持哪些客户端接入认证方式？

答：GaussDB支持三种认证方式：

- 基于主机的认证：服务器端根据客户端的IP地址、用户名及要访问的数据库查看配置文件，从而判断用户是否通过认证。
- 口令认证：包括远程连接的加密口令认证和本地连接的非加密口令认证。
- SSL加密：使用OpenSSL（开源安全通信库）提供服务器端和客户端安全连接的环境。

以上三种认证方式都需要配置gs_hba.conf文件。gs_hba.conf文件的格式是一行写一条信息，表示一个认证规则，空白和注释（以#开头）被忽略。每个认证规则是由若干空格和/，空格和制表符分隔的字段组成。如果字段用引号包围，则它可以包含空白。一条记录不能跨行存在。

9.2.4 如何查看数据库应用连接数以及最大连接数？

答：当应用程序与数据库的连接数超过最大值，则新的连接无法建立。因此应及时释放空闲的连接或者增加最大连接数，保证数据库应用程序的正常连接。

```
--查看数据库应用连接数。
gaussdb=# SELECT count(*) FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

```
count
-----
65
(1 row)

--查看数据库现有最大连接数。
gaussdb=# SHOW max_connections;
max_connections
-----
1024
(1 row)
```

9.2.5 如何获取当前连接数据库的用户名？

答：执行以下命令查询当前用户名。

```
gaussdb=# SELECT current_user;
```

9.2.6 如何查看指定用户的会话连接数上限？

答：查看指定用户的会话连接数上限，可以使用如下SQL查询语句：

```
--查看omm用户的会话连接上限。
gaussdb=# SELECT ROLNAME,ROLCONNLIMIT FROM PG_ROLES WHERE ROLNAME='omm';
rolname | rolconnlmit
-----+-----
omm      |      -1
(1 row)
说明：-1表示没有对用户omm设置连接数的限制。

--查看omm1用户的会话连接上限。
gaussdb=# SELECT ROLNAME,ROLCONNLIMIT FROM PG_ROLES WHERE ROLNAME='omm1';
rolname | rolconnlmit
-----+-----
omm1     |      10
(1 row)
说明：用户omm1设置连接数的限制为10。
```

9.3 数据库安全

9.3.1 如何手动锁定和解锁用户？

答：在数据库使用过程中，如果管理员发现某账户存在非法访问等异常情况，可手动锁定该账户。当判断该账户恢复正常后，管理员可手动解锁该账户。

以手动锁定和解锁用户jack为例：

- 手动锁定用户。
gaussdb=# ALTER USER jack ACCOUNT LOCK;
ALTER ROLE
- 手动解锁用户。
gaussdb=# ALTER USER jack ACCOUNT UNLOCK;
ALTER ROLE

9.3.2 GaussDB 支持哪些安全管理？

答：支持SSL安全网络连接、用户权限管理、密码管理、安全审计等功能，保证数据库在管理层、应用层、系统层和网络层的安全性。GaussDB的Agent模式中，由于http协议没有数据加密，数据为明文传输存在安全隐患，故默认配置为https协议。

9.3.3 创建模型服务前，如何生成 OBS 传输密钥？

答：创建服务时会对API-KEY进行加密操作，原理是复用OBS传输使用的密钥文件，如果该文件不存在则需要参考如下命令生成：

```
gs_guc generate -S username -D $GAUSSHOMe/bin -o obsserver
```

9.3.4 动态脱敏策略如何生效？

GaussDB的安全策略开关默认是off关闭状态，通常需要设置为on开启状态，使所有动态脱敏策略生效；除了安全总开关，各项脱敏策略也有对应的开关。只有开关开启，对应的脱敏策略才能生效。

- 开启安全总开关。

```
--设置guc参数enable_security_policy为on。  
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "enable_security_policy=on"
```

- 只设置单项脱敏策略。

```
--开启脱敏策略。  
gaussdb=# ALTER MASKING POLICY policy_name enable;
```

9.3.5 如何设置账号的有效期？

答：在数据库使用过程中，会创建一些用户对数据库进行相关操作。创建用户时，需要限制用户的操作期限（即有效开始时间和有效结束时间），具体的设置方法如下：

步骤1 通过gscli连接数据库。

步骤2 创建用户并指定用户的有效开始时间和有效结束时间。

```
gaussdb=# CREATE USER jack WITH PASSWORD '*****' VALID BEGIN '2015-10-10 08:00:00' VALID UNTIL  
'2016-10-10 08:00:00';  
CREATE ROLE
```

当显示CREATE ROLE信息时表示创建用户成功。

步骤3 当用户已不在有效使用期内，重新设定账号的有效期，包括有效开始时间和有效结束时间。

```
gaussdb=# ALTER USER jack WITH VALID BEGIN '2016-11-10 08:00:00' VALID UNTIL '2017-11-10 08:00:00';  
ALTER ROLE
```

当显示ALTER ROLE信息时表示重新设定成功。

----结束

9.3.6 GaussDB 如何开启审计策略，支持哪些内容的审计？

答：审计总开关audit_enabled支持动态加载。在数据库运行期间修改该配置项的值会立即生效，无需重启数据库。默认值为on，表示开启审计功能。除了审计总开关，各个审计项也有对应的开关。只有开关开启，对应的审计功能才能生效。各审计项的开关支持动态加载。在数据库运行期间修改审计开关的值，不需要重启数据库便可生效。

- 开启审计总开关。

```
--设置guc参数audit_enabled=on。  
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "audit_enabled = on"
```

- 只设置单独一项审计项。

```
--开启对数据库对象（DATABASE、SCHEMA、USER、NODE GROUP、SQL Patch）的DDL操作的审计。  
gs_guc reload -Z coordinator -D /gaussdb/data/coordinator -c "audit_system_object=67121159"
```

 说明

各审计项的默认参数都符合安全标准，用户可以根据需要开启其他审计功能，但会对性能有一定影响。详细配置审计项请参见《开发指南》中“数据库安全 > 数据库审计”章节。

9.3.7 如何查看某个用户有哪些表的权限？

答：可使用information_schema.table_privileges系统视图查看。

```
--查询jim用户有哪些表的权限。
gaussdb=# SELECT * FROM information_schema.table_privileges WHERE grantee = 'jim';
 grantor | grantee | table_catalog | table_schema | table_name | privilege_type | is_grantable | with_hierarchy
-----+-----+-----+-----+-----+-----+-----+-----
jim      | jim     | postgres      | jim          | reason     | INSERT        | YES         | NO
jim      | jim     | postgres      | jim          | reason     | SELECT        | YES         | YES
jim      | jim     | postgres      | jim          | reason     | UPDATE        | YES         | NO
jim      | jim     | postgres      | jim          | reason     | DELETE        | YES         | NO
jim      | jim     | postgres      | jim          | reason     | TRUNCATE      | YES         | NO
jim      | jim     | postgres      | jim          | reason     | REFERENCES    | YES         | NO
jim      | jim     | postgres      | jim          | reason     | TRIGGER       | YES         | NO
(7 rows)
```

系统视图information_schema.table_privileges字段如表9-1所示。

表 9-1 table_privileges 字段说明

字段	数据类型	描述
grantor	information_schema.sql_identifier	赋权用户。
grantee	information_schema.sql_identifier	被赋权用户。
table_catalog	information_schema.sql_identifier	包含该表的数据库名。
table_schema	information_schema.sql_identifier	包含该表的模式名。
table_name	information_schema.sql_identifier	表名。
privilege_type	information_schema.character_data	被赋予的权限类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、ANALYZE、VACUUM、ALTER、DROP或TRIGGER。
is_grantable	information_schema.yes_or_no	权限是否可赋予其他用户，YES表示可赋予，NO表示不可赋予。
with_hierarchy	information_schema.yes_or_no	是否允许在表继承层级上的特定操作。当特定操作作为SELECT时显示YES，否则为NO。

9.3.8 GaussDB 如何开启三权分立？

答：在实际业务管理中，为了避免系统管理员拥有过度集中的权利带来高风险，可以设置三权分立，将系统管理员的用户管理和审计管理的权限分给安全管理员和审计管

理员。设置参数“enableSeparationOfDuty”为“on”，则会开启三权分立，限制系统管理员的权限。不允许系统管理员创建或更改用户配置。同时，建议关闭GUC参数“enable_copy_server_files”，取消系统管理员的copy权限，防止系统管理员通过copy命令读取或者修改用户配置文件。配置方法如下：

步骤1 切换实例用户omm。

```
su - omm
```

步骤2 执行以下语句修改参数enableSeparationOfDuty为on。

```
gs_guc set -Z coordinator -Z datanode -N all -I all -c "enableSeparationOfDuty = on"
```

步骤3 重启数据库。

```
gs_om -t stop  
gs_om -t start
```

----结束

9.3.9 GaussDB 如何创建只读用户？

答：在业务场景中，数据库管理员可以通过SCHEMA来划分不同业务。例如需要创建一个只读用户user1，允许这个用户访问sc_test模式下所有的表（包括以后创建的新表），但是不允许插入、修改和删除数据。具体过程如下：

```
--前置操作：创建用户和模式sc_test。  
gaussdb=# CREATE ROLE u1 WITH SYSADMIN PASSWORD '*****';  
gaussdb=# CREATE SCHEMA sc_test;  
gaussdb=# ALTER SCHEMA sc_test OWNER TO u1;  
  
--创建用户user1。  
gaussdb=# CREATE ROLE user1 WITH LOGIN PASSWORD '*****';  
  
--赋予用户user1对模式sc_test的访问权限。  
gaussdb=# GRANT USAGE ON SCHEMA sc_test TO user1;  
  
--赋予用户user1对模式sc_test下所有表的SELECT权限。  
gaussdb=# GRANT SELECT ON ALL TABLES IN SCHEMA sc_test TO user1;  
  
--赋予用户user1对模式sc_test下未创建的表的SELECT的权限。  
gaussdb=# ALTER DEFAULT PRIVILEGES FOR USER u1 GRANT SELECT ON TABLES TO user1;  
  
--删除。  
gaussdb=# ALTER DEFAULT PRIVILEGES FOR USER u1 REVOKE SELECT ON TABLES FROM user1;  
gaussdb=# DROP SCHEMA sc_test;  
gaussdb=# DROP ROLE user1;  
gaussdb=# DROP ROLE u1;
```

9.3.10 GaussDB 会话有几种状态？如何查询用户的会话状态？

答：共有6种状态，具体如下：

- active：后台正在执行一个查询。
- idle：后台正在等待一个新的客户端命令。
- idle in transaction：后台在事务中，但事务中没有语句在执行。
- idle in transaction (aborted)：后台在事务中，但事务中有语句执行失败。
- fastpath function call：后台正在执行一个fast-path函数。
- disabled：如果后台禁用track_activities，则报告这个状态。

查询用户的会话状态，请参照如下示例：

```
gaussdb=# SELECT datname, username, usesysid, state, pid FROM pg_stat_activity;  
datname | username | usesysid | state | pid
```

```
postgres | omm1 | 10 | idle | 139669908420352
postgres | omm1 | 10 | active | 139669573990144
```

9.3.11 如何查询 SSL 证书的失效日期？

答：GaussDB实例的SSL证书相关信息保存路径为\$GAUSSHOME/share/sslcert/。具体操作步骤如下所示：

步骤1 进入SSL证书相关信息保存的目录。

```
[omm1@node41 ~]$ cd $GAUSSHOME/share/sslcert/
```

步骤2 进入client.crt证书所在的目录。

```
[omm1@node41 ~]$ cd ./pgsql && ll
total 56
-rw----- 1 omm1 dbgrp 5690 Apr  7 11:14 cacert.pem
-rw----- 1 omm1 dbgrp 5713 Apr  7 11:14 client.crt
-rw----- 1 omm1 dbgrp 2654 Apr  7 11:14 client.key
-rw----- 1 omm1 dbgrp  56 Apr  7 11:14 client.key.cipher
-rw----- 1 omm1 dbgrp  24 Apr  7 11:14 client.key.rand
-rw----- 1 omm1 dbgrp 4542 Apr  7 11:14 openssl.cnf
-rw----- 1 omm1 dbgrp 5713 Apr  7 11:14 server.crt
-rw----- 1 omm1 dbgrp 2654 Apr  7 11:14 server.key
-rw----- 1 omm1 dbgrp  56 Apr  7 11:14 server.key.cipher
-rw----- 1 omm1 dbgrp  24 Apr  7 11:14 server.key.rand
```

步骤3 使用openssl命令工具输出证书的日期。

```
--notBefore表示证书开始有效日期，notAfter表示证书失效日期。
[omm1@node41 ~]$ openssl x509 -in client.crt -noout -dates
notBefore=Apr  7 03:14:21 2024 GMT
notAfter=Apr  2 03:14:21 2044 GMT
```

----结束

9.3.12 如何配置开启密码复杂度校验？

答：为了保证数据库的使用安全，建议配置开启密码复杂度校验，在创建用户或者修改密码时，校验密码复杂度。

配置及验证方法如下：

```
--配置方法。
gs_guc reload -Z coordinator -N all -I all -c "password_policy=1"

--验证方法。
gaussdb=# SELECT name,setting FROM pg_settings WHERE name='password_policy';
 name      | setting 
-----+-----
 password_policy | 1
(1 row)
```

9.3.13 如何检查数据库初始化用户角色？

答：数据库初始化用户角色具有数据库的最高权限，并且具有所有的系统权限和对象权限。建议仅将此用户作为DBA管理用途。

```
--检查初始化用户角色。
gaussdb=# SELECT rolname FROM pg_roles WHERE rolsuper=true;
 rolname 
-----
 omm
(1 row)
```

9.3.14 哪些系统视图与审计信息有关？

答：以下3个视图与审计信息有关，需要有系统管理员或安全策略管理员权限才可以访问这些视图：

- GS_AUDITING：显示对数据库相关操作的所有审计信息。
- GS_AUDITING_ACCESS：显示对数据库DML相关操作的所有审计信息。
- GS_AUDITING_PRIVILEGE：显示对数据库DDL相关操作的所有审计信息。

9.3.15 如何配置日志文件权限？

答：当开启参数logging_collector时，可以使用参数log_file_mode设置服务器日志文件的权限。因为日志文件中可能含有敏感数据，所以日志文件访问必须受到限制。

配置方法如下：

```
--执行以下语句修改参数log_file_mode为0600。  
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "log_file_mode=0600"
```

检查方法如下：

```
gaussdb=# SELECT name,setting FROM pg_settings WHERE name='log_file_mode' AND setting != '0600';
```

9.3.16 如何配置数据库账号登录失败尝试次数？

答：在数据库使用过程中，推荐配置账户登录失败尝试次数，防止密码被暴力破解。配置参数failed_login_attempts后，当失败次数超过此参数值后，账户将被自动锁定。

--配置GUC参数failed_login_attempts等于10。

```
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "failed_login_attempts=10"
```

9.3.17 透明加密和全密态有什么区别？

答：透明加密和全密态都可以加密数据，但这是两种不同的加密模式，具体区别如下：

- 加密位置不同：透明加密在数据库存储模块后加密，全密态在数据库驱动中加密。
- 加密对象不同：透明加密数据页中的全部数据，全密态加密SQL语句中的某几列数据。
- 加密性能不同：透明加密一次可以处理8k的数据页，全密态一次仅可以处理SQL语句中的一条数据，透明加密性能高于全密态。
- 安全性不同：全密态安全性高于透明加密。

9.3.18 如何保证数据的安全性？

答：GaussDB通过多种安全措施来保证数据的安全性，包括：

- 加密技术：对敏感数据进行加密存储和传输，防止数据泄露。
- 访问控制：通过用户认证和权限管理，控制不同的用户对数据的访问权限。
- 备份和恢复：定期备份数据，并在需要时能够迅速恢复数据，防止数据丢失。
- 审计和监控：可通过记录用户的操作行为，并进行实时监控和告警，以便及时发现和处理安全事件。

9.3.19 GaussDB 如何创建弱口令？

答：在GaussDB数据库中，弱口令字典默认为空，用户可以手动向系统表gs_global_config中新增一条或多条弱口令。

示例如下：

```
--向gs_global_config系统表中插入单个弱口令。
gaussdb=# CREATE WEAK PASSWORD DICTIONARY WITH VALUES ('password1');

--向gs_global_config系统表中插入多个弱口令。
gaussdb=# CREATE WEAK PASSWORD DICTIONARY WITH VALUES ('password2'),('password3');

--查看gs_global_config系统表弱口令。
gaussdb=# SELECT * FROM gs_global_config WHERE NAME LIKE 'weak_password';
   name   | value
-----+-----
weak_password | password1
weak_password | password2
weak_password | password3
(3 rows)
```

9.3.20 GaussDB 中模式、用户和角色如何理解？

答：模式是对数据库做一个逻辑分割，所有的数据库对象都建立在模式下面。模式和用户是弱绑定的，弱绑定是指虽然创建用户的同时会自动创建一个同名模式，但用户也可以单独创建模式，并且为用户指定其他的模式。

用户和角色是用来控制对数据库的访问。一个数据库可能有很多的用户需要访问，为了方便管理这些用户，将用户组成一个数据库角色。根据角色自身的设置不同，一个角色可以看做是一个数据库用户，或者一组数据库用户。角色和用户之间的区别只在于角色默认是没有LOGIN权限的。在业务设计中，仅建议通过ROLE来配置权限，而不是用来访问数据库。

9.3.21 为什么 REVOKE 某个用户的 CONNECT ON DATABASE 权限后还可以登录数据库？

答：使用REVOKE CONNECT ON DATABASE dbname FROM u1命令撤销u1的权限后，因为数据库的CONNECT权限授予了public，需要指定public来实现。

- GaussDB提供了一个隐式定义的拥有所有角色组public，所有创建的用户和角色默认拥有public所拥有的权限。需要撤销或重新授予用户和角色public的权限，可通过GRANT和REVOKE指定关键字public实现。
- GaussDB会将某些类型的对象上的权限授予public。默认情况以下这些对象的权限会授予public：
 - 数据库的CONNECT权限。
 - CREATE TEMP TABLE权限。
 - 函数的EXECUTE权限。
 - 语言和数据类型（包括域）的USAGE权限。
- 对象的拥有者可以撤销默认授予public的权限，并授予权限给其他用户。

具体示例如下：

1. 创建用户u1和数据库db_test。

```
gaussdb=# CREATE USER u1 PASSWORD '*****';
gaussdb=# CREATE DATABASE db_test;
```

2. 测试确定u1用户可以正常连接数据库test。后退出数据库。
gsql -d db_test -h xxx.xxx.xxx.xxx -p xxxx -U u1 -W '*****'
3. 执行REVOKE命令撤销public连接db_test的权限。
gaussdb=# REVOKE CONNECT ON DATABASE db_test FROM public;
4. 验证结果，使用u1连接数据库。结果显示u1用户已经无法连接数据库。
gsql -d db_test -h xxx.xxx.xxx.xxx -p xxxx -U u1 -W '*****'
gsql: FATAL: permission denied for database "db_test"
DETAIL: User does not have CONNECT privilege.

--删除。
gaussdb=# DROP USER u1;
gaussdb=# DROP DATABASE db_test;

9.3.22 如何查看数据库用户的创建时间？

答：通过系统表PG_AUTH_HISTORY中的passwordtime字段查看用户密码的初始设置时间，即该用户的创建时间。

```
gaussdb=# SELECT t1.username, t1.usesysid, min(t2.passwordtime) as create_time FROM pg_user t1,
pg_auth_history t2 WHERE t1.username='jerry' AND t1.usesysid = t2.rolid GROUP BY t1.username,
t1.usesysid;
username | usesysid |      create_time
-----+-----+-----
jerry    | 27311   | 2024-07-23 14:46:08.284336+08
(1 row)
```

9.4 数据库使用入门

9.4.1 GaussDB 的 DROP、TRUNCATE、DELETE 这三种删除数据的方式主要区别是什么？

答：DROP、TRUNCATE、DELETE这三种删除方式的区别主要体现在删除的速度和删除的范围上，具体如下：

- DROP语句的删除速度最快，可以删除整个表，包括表结构、数据、索引和权限。
- TRUNCATE语句的删除速度中等，可以删除表中的所有数据，但不删除表结构。
- DELETE语句的删除速度最慢，可以根据条件删除表中的数据，但不包括表结构。

9.4.2 GaussDB 数据库一个汉字占几个字节？

答：取决于数据库的字符集编码，GBK字符集一个汉字占2个字节，UTF8字符集一个汉字占3个字节。

9.4.3 如何并行创建索引？

答：参考如下方法：

```
--根据实际情况设置maintenance_work_mem参数的大小。
gaussdb=# SET maintenance_work_mem = '8GB';

--建表。
gaussdb=# CREATE TABLE table_name (col1 int, col2 int);

--修改表创建索引的线程数量,需根据实际情况修改线程数。
gaussdb=# ALTER TABLE table_name SET (parallel_workers=4);

--创建索引。
```

```
gaussdb=# CREATE INDEX index_name ON table_name(col1);  
  
--重置该表的parallel_workers参数。  
gaussdb=# ALTER TABLE table_name RESET (parallel_workers);  
  
--删除索引。  
gaussdb=# DROP INDEX index_name;  
  
--删除表。  
gaussdb=# DROP TABLE table_name;
```

9.4.4 如何查看 GaussDB KernelGaussDB 客户端与服务端字符编码类型？

答：执行如下语句查看客户端字符串编码类型。

```
gaussdb=# SHOW client_encoding;  
client_encoding  
-----  
UTF8  
(1 row)
```

执行如下语句查看服务端字符编码类型。

```
gaussdb=# SHOW server_encoding;  
server_encoding  
-----  
SQL_ASCII  
(1 row)
```

9.4.5 如何查看当前会话 ID 以及登录时间？

答：执行如下SQL可查看当前会话ID以及登录时间。

```
--查看当前会话ID。  
gaussdb=# SELECT pg_backend_pid() AS session_id;  
session_id  
-----  
139883131631360  
(1 row)  
  
--查看当前会话的登录时间。  
gaussdb=# SELECT pg_stat_activity.backend_start AS login_time FROM pg_stat_activity WHERE  
pg_stat_activity.pid = pg_backend_pid();  
login_time  
-----  
2024-07-19 16:39:45.429437+08  
(1 row)
```

9.4.6 如何查看某张表的创建时间？

答：在系统视图PG_TABLES里created字段的取值为表的真实创建时间。

```
--创建表。  
gaussdb=# CREATE TABLE test(id int, name varchar(10));  
  
--查询test表创建时间。  
gaussdb=# SELECT tablename,created FROM pg_tables WHERE tablename = 'test';  
tablename |      created  
-----+-----  
test      | 2024-01-12 14:50:59.611988+08  
(1 row)
```

9.4.7 如何查看数据库参数值？

答：数据库参数可以通过SHOW命令查看，或者通过pg_settings视图查看。例如：

```
--方式一：通过SHOW命令查看shared_buffers参数值。
gaussdb=# SHOW shared_buffers;
shared_buffers
-----
1GB
(1 row)

--方式二：通过pg_settings视图查看shared_buffers参数值。
gaussdb=# SELECT name, setting FROM pg_settings WHERE name='shared_buffers';
 name | setting
-----+-----
shared_buffers | 131072
(1 row)
```

9.4.8 如何查看外表信息？

答：通过pg_get_tabledef()函数可以查看外表信息，具体如下示例：

```
--创建SERVER。
gaussdb=# CREATE SERVER my_server FOREIGN DATA WRAPPER log_fdw;
--创建外表。
gaussdb=# CREATE FOREIGN TABLE foreign_tbl (col1 text) SERVER my_server OPTIONS (logtype 'gs_log');
--查看foreign_tbl外表信息。
gaussdb=# SELECT * FROM pg_get_tabledef('foreign_tbl');
pg_get_tabledef
-----
SET search_path = public;      +
CREATE FOREIGN TABLE foreign_tbl (+
  col1 text                      +
)                                +
SERVER my_server                +
OPTIONS (                        +
  logtype 'gs_log'              +
)                                +
DISTRIBUTE BY ROUNDROBIN;
(1 row)
--删除外表。
gaussdb=# DROP FOREIGN TABLE foreign_tbl;
--删除SERVER。
gaussdb=# DROP SERVER my_server;
```

9.4.9 如何查看系统中正在运行的 SELECT 语句？

答：设置参数track_activities为on。当此参数取值为on时，数据库系统才会收集当前活动查询的运行信息。

```
gaussdb=# SET track_activities = on;
```

查看正在运行的查询语句。以查看正在运行的查询语句所连接的数据库名、执行查询的用户、查询状态及查询对应的PID为例：

```
gaussdb=# SELECT datname, username, state,pid,query FROM pgxc_stat_activity WHERE state = 'active';
 datname | username | state | pid | query
-----+-----+-----+-----+-----
postgres | ommittj | active | 281431140520064 |
postgres | ommittj | active | 281431621030016 |
postgres | ommittj | active | 281430469562496 |
postgres | ommittj | active | 281430402191488 | WLM fetch collect info from data nodes
postgres | ommittj | active | 281429778288768 | SELECT datname, username, state,pid,query FROM
pg_stat_activity where state = 'active';
(5 rows)
```

9.4.10 如何查询表存在的索引？

答：通过pg_indexes视图可查询表存在的索引。示例如下：

```
--查询tbl_test1表存在的索引。
gaussdb=# SELECT * FROM pg_indexes WHERE tablename='tbl_test1';
 schemaname | tablename | indexname | tablespace |
indexdef
-----+-----+-----+-----+
public | tbl_test1 | idx_test1 | | CREATE INDEX idx_test1 ON tb1_test1 USING ubtree (a) WITH
(storage_type=USTORE) TABLESPACE pg_default
(1 row)
```

9.4.11 如何查询表的存储方式？

答：可通过表定义函数pg_get_tabledef()查询，row表示行存。具体查询方式如下所示：

```
--查询test_1表的存储方式。
gaussdb=# SELECT * FROM pg_get_tabledef('test_1');
pg_get_tabledef
-----
SET search_path = public;
CREATE TABLE test_1 (
  a integer
)
WITH (orientation=row, compression=no, storage_type=USTORE, segment=off)+
DISTRIBUTE BY HASH(a)
TO GROUP group_version1;
(1 row)
```

9.4.12 如何查询表和数据库容量？

答：使用查询函数pg_table_size()和pg_database_size()。

```
--创建表并插入数据。
gaussdb=# CREATE TABLE test(id int, name varchar(10));
gaussdb=# INSERT INTO test VALUES(1,'Tony'),(2,'Thor'),(3,'Hunk');
--查询表test容量。
gaussdb=# SELECT pg_table_size('test');
pg_table_size
-----
8192
(1 row)

--查询数据库postgres容量。
gaussdb=# SELECT pg_database_size('postgres');
pg_database_size
-----
235134980
(1 row)
```

9.4.13 如何查询会话数？

答：查询会话数，可以使用如下SQL查询语句：

```
gaussdb=# SELECT state,count(*) FROM pgxc_stat_activity group by 1 order by 2 desc;
 state | count
-----+-----
idle | 30
active | 22
 | 12
(3 rows)
```

说明：state值为空表明是其它用户的会话。

9.4.14 如何查询数据库、表空间、表的大小？

- 查询数据库大小：

```
gaussdb=# SELECT datname, pg_database_size(datname), pg_database_size(datname)/
1024/1024/1024 as "dataSize_GB" FROM pg_database;
 datname      | pg_database_size | dataSize_GB
-----+-----+-----
 template1    |       79208460 | .0737686268985271
 template0    |       79208460 | .0737686268985271
 postgres     |    1398554636 | 1.30250550433993
 mysql_compatible_db |       79208460 | .0737686268985271
(4 rows)
```

- 查询表空间大小：

```
gaussdb=# SELECT pg_size_pretty(pg_tablespace_size('pg_default'));
pg_size_pretty
-----
345 MB
(1 row)
```

- 查询表大小：

```
gaussdb=# SELECT pg_size_pretty(pg_relation_size('table_name'));
pg_size_pretty
-----
0 bytes
(1 row)
```

9.4.15 如何查询数据库运行时间？

答：查询数据库运行时间，可以使用如下SQL查询语句：

```
gaussdb=# SELECT pg_postmaster_start_time() startup_time, (sysdate - startup_time) run_time;
```

9.4.16 如何查询数据库最近一次重启时间？

答：查询数据库最近一次重启时间，可以使用如下SQL查询语句：

```
gaussdb=# SELECT pg_postmaster_start_time();
pg_postmaster_start_time
-----
2024-08-01 15:34:51.738604+08
(1 row)
```

9.4.17 如何创建默认为当前时间的列？

答：建表时设置字段默认值为CURRENT_TIMESTAMP。

```
gaussdb=# CREATE TABLE tbl (id int, modtime date DEFAULT CURRENT_TIMESTAMP);
```

9.4.18 如何创建只读视图以及判断该视图是否是只读视图？

答：创建只读视图。

```
--创建test_tb表。
gaussdb=# CREATE TABLE test_tb(c1 int, c2 int);

--创建只读视图test_v。
gaussdb=# CREATE VIEW test_v AS SELECT * FROM test_tb WITH READ ONLY;
```

判断该视图是否是只读视图。

```
--向视图中写入数据来判断视图是否是只读视图。
gaussdb=# INSERT INTO test_v VALUES (1, 1);
ERROR: Cannot perform a DML operation on a read-only view.
```

```
--删除表及视图。  
gaussdb=# DROP VIEW test_v;  
gaussdb=# DROP TABLE test_tb;
```

9.4.19 如何关闭系统表结构修改配置？

答：参数allow_system_table_mods允许修改系统表的结构。开发选项参数和GaussDB内部协同工作，在某些情况下，可以协助恢复遭到严重破坏的数据库。建议将参数allow_system_table_mods设为off，设置完参数后，需要重启数据库。

配置及验证方法如下：

```
--配置方法。  
gs_guc set -Z coordinator -Z datanode -N all -I all -c "allow_system_table_mods=off"  
  
--检查方法。  
gaussdb=# SELECT name,setting FROM pg_settings WHERE name = 'allow_system_table_mods';  
      name      | setting  
-----+-----  
allow_system_table_mods | off  
(1 row)
```

9.4.20 如何判断字段是否为 NULL？

答：使用IS NULL和IS NOT NULL来判断字段是否为NULL，例如：

```
gaussdb=# SELECT * FROM tab WHERE col1 IS NULL;
```

9.4.21 如何删除重复的行？

答：可以用如下两种方式。

- 方式一：

该方式仅适用于分布表，且关联条件必须包含分布列，对于复制表，相同数据的ctid在各个DN上可能不一致，因此无法使用此方式删除重复行。

```
--建表和插入数据。  
gaussdb=# CREATE TABLE tbl_del(c1 varchar,c2 int) distribute by hash(c1);  
gaussdb=# INSERT INTO tbl_del VALUES ('a',1),('a',1);  
gaussdb=# INSERT INTO tbl_del VALUES ('b',2),('b',2);  
gaussdb=# INSERT INTO tbl_del VALUES ('c',3);
```

```
--查询去重后的数据。  
gaussdb=# SELECT distinct * FROM tbl_del;  
c1 | c2  
----+----  
b | 2  
a | 1  
c | 3  
(3 rows)
```

```
--使用如下SQL去除重复的数据。  
gaussdb=# DELETE FROM tbl_del t1 WHERE ctid <>(  
    SELECT max(ctid) FROM tbl_del t2 WHERE t1.c1 = t2.c1  
);
```

```
--查询数据。  
gaussdb=# SELECT * FROM tbl_del;  
c1 | c2  
----+----  
a | 1  
b | 2  
c | 3  
(3 rows)
```

```
--删除。
gaussdb=# DROP TABLE tbl_del;

• 方式二：
--建表和插入数据。
gaussdb=# CREATE TABLE tbl_del(c1 varchar,c2 int);
gaussdb=# INSERT INTO tbl_del VALUES ('a',1),('a',1);
gaussdb=# INSERT INTO tbl_del VALUES ('b',2),('b',2);
gaussdb=# INSERT INTO tbl_del VALUES ('c',3);

--查询去重后的数据。
gaussdb=# SELECT distinct * FROM tbl_del;
c1 | c2
----+----
b | 2
a | 1
c | 3
(3 rows)

--复制一张表tbl_del2。
gaussdb=# CREATE TABLE tbl_del2 AS SELECT * FROM tbl_del;

--清空tbl_del表,并插入去重后的数据。
gaussdb=# TRUNCATE TABLE tbl_del;
gaussdb=# INSERT INTO tbl_del SELECT distinct * FROM tbl_del2;

--查询。
gaussdb=# SELECT * FROM tbl_del;
c1 | c2
----+----
b | 2
a | 1
c | 3
(3 rows)

--删除。
gaussdb=# DROP TABLE tbl_del,tbl_del2;
```

9.4.22 如何使用 SQL 语句查看数据库的模式？

答：可以通过gs_deployment()系统函数查看当前数据库的部署形态，示例如下：

```
gaussdb=# SELECT gs_deployment();
gs_deployment
-----
Distribute
(1 row)
```

9.4.23 如何通过 SQL 语言查询表、索引、视图的定义？

- 查询表定义：
gaussdb=# SELECT * FROM pg_get_tabledef('table_name');
- 查询索引定义：
gaussdb=# SELECT oid FROM pg_class WHERE relname='index_name';
gaussdb=# SELECT * FROM pg_get_indexdef(oid);
- 查询视图定义：
gaussdb=# SELECT viewname, definition FROM pg_views WHERE viewname='view_name';

9.4.24 数据分区都有哪些优点？

答：优点如下：

- 改善可管理性：利用分区，可以将表和索引划分为一些更小、更易管理的单元。有了分区，维护操作可以专门针对表的特定部分执行。

- 可提升删除操作的性能：删除数据时可以删除整个分区，与分别删除每行相比，这种操作非常高效和快速。通过ALTER TABLE ... DROP PARTITION语法进行删除分区表操作。
- 改善查询性能：通过限制要检查或操作的数据数量，分区可以带来许多性能优势。
- 分区剪枝：也称为分区消除，是GaussDB在执行时过滤掉不需要扫描的分区，只对相关的分区进行扫描的技术。分区剪枝通常可以将查询性能提高若干数量级。
- 智能化分区连接：通过使用一种称为智能化分区连接的技术，分区还可以改善多表连接的性能。当将两个表连接在一起，并且至少其中一个表使用连接键进行分区时，可以应用智能化分区连接。智能化分区连接将一个大型连接分为多个较小的连接，这些较小的连接包含与连接的表“相同”的数据集。这里，“相同”定义为恰好包含连接的两端中相同的分区键值集，因此可以确保只有这些“相同”数据集的连接才会有效，而不必考虑其他数据集，目前不支持列表分区和哈希分区。

9.4.25 唯一约束和唯一索引有什么区别？

答：唯一约束和唯一索引的主要区别如下：

- 唯一约束和唯一索引概念不同。
唯一约束确保一列或者一组列中包含的数据对于表中所有的行都是唯一的。唯一索引用于限制索引字段值的唯一性，或者是多个字段组合值的唯一性。CREATE UNIQUE INDEX创建唯一索引。
- 唯一约束和唯一索引功能不同。
约束主要是为了保证数据的完整性，索引主要是为了辅助查询。
- 唯一约束和唯一索引使用方法不同。
 - 创建唯一约束，会自动创建一个同名的唯一索引，该索引不能单独删除，删除约束会自动删除索引。唯一约束是通过唯一索引来实现数据的唯一。
 - 创建一个唯一索引，这个索引是独立的，并且可以单独删除。
 - 如果一个列上有约束和索引，且两者可以单独地删除。可以先建唯一索引，再建同名的唯一约束。
 - 如果表的一个字段，要作为另外一个表的外键，这个字段必须有唯一约束（或是主键），如果只是有唯一索引，就会报错。

9.4.26 为什么查询中无法识别表名或者列名？

答：无法识别名称最常见的原因是创建过程中表名或者列名使用了双引号引入了大小写。使用双引号时，表名和列名（称为标识符）在存储是区分大小写的，所以在查询时必须使用双引号且大小写需要与创建时匹配。因为某些接口或应用程序在创建时会自动引入双引号，所以建议创建时统一使用小写。示例：

```
--创建表。
gaussdb=# CREATE TABLE "TB_test" ("Tid" int, num int);

--查询表时不加双引号，默认都为小写，会报错。
gaussdb=# SELECT * FROM TB_test;
ERROR: relation "tb_test" does not exist
LINE 1: SELECT * FROM TB_test;

gaussdb=# SELECT Tid FROM "TB_test";
ERROR: Column "tid" does not exist
LINE 1: SELECT Tid FROM "TB_test"
```

```
--查询时加双引号。
gaussdb=# SELECT "Tid" FROM "TB_test";
Tid
-----
(0 rows)

--删除表。
gaussdb=# DROP TABLE "TB_test";
```

9.4.27 以 pg_temp 或 pg_toast_temp 开头的 SCHEMA 是什么？

答：在查询SCHEMA的时候，发现查询结果中存在以pg_temp或pg_toast_temp开头的SCHEMA如下所示。

```
gaussdb=# SELECT nspname FROM pg_namespace WHERE nspname LIKE 'pg_t%' AND nspname !=
'pg_toast';
nspname
-----
pg_temp_cn_5001_3_2_3795
pg_toast_temp_cn_5001_3_2_3795
(2 rows)
```

以pg_temp或者pg_toast_temp开头的表是在创建临时表时，通过每个会话以pg_temp开头的SCHEMA来保证只对当前会话可见。因此，不建议用户在日常操作中手动删除以pg_temp、pg_toast_temp开头的SCHEMA。

临时表只在当前会话可见，会话结束后会自动删除，相对应的SCHEMA也会被删除。

9.4.28 在使用了字符串处理函数后查询就不使用索引，该如何处理？

答：在索引列上使用函数处理后就不会走标准的索引，可以创建一个表达式索引，以lpad()函数为例。

```
--建表并插入数据。
gaussdb=# CREATE TABLE tbl_test (c1 varchar);
gaussdb=# INSERT INTO tbl_test VALUES (generate_series(1,10000));

--普通索引使用lpad()函数，查询不会使用该索引。
gaussdb=# CREATE INDEX idx_test_c1 ON tbl_test(c1);
gaussdb=# SET max_datanode_for_plan = 1;
gaussdb=# SET explain_perf_mode = normal;
--执行计划部分内容如下。
gaussdb=# EXPLAIN SELECT * FROM tbl_test WHERE lpad(c1,5,'0') = '00324';
QUERY PLAN
-----
Data Node Scan on "__REMOTE_FQS_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes

Remote SQL: SELECT c1 FROM public.tbl_test WHERE lpad(c1::text, 5, '0')::text = '00324'::text
Datanode Name: datanode1
Seq Scan on tbl_test (cost=0.00..96.22 rows=25 width=32)
Filter: (lpad((c1)::text, 5, '0')::text = '00324'::text)

(8 rows)

--修改为表达式索引后，该查询会使用索引。
gaussdb=# DROP INDEX idx_test_c1;
gaussdb=# CREATE INDEX idx_test_c1 ON tbl_test(lpad(c1,5,'0'));
gaussdb=# EXPLAIN SELECT * FROM tbl_test WHERE lpad(c1,5,'0') = '00324';
QUERY PLAN
-----
Data Node Scan on "__REMOTE_FQS_QUERY__" (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes

Remote SQL: SELECT c1 FROM public.tbl_test WHERE lpad(c1::text, 5, '0')::text = '00324'::text
```

```
Datanode Name: datanode1
Bitmap Heap Scan on tbl_test (cost=1.55..18.09 rows=25 width=32)
  Recheck Cond: (lpad((c1)::text, 5, '0')::text) = '00324'::text
  -> Bitmap Index Scan on idx_test_c1 (cost=0.00..1.54 rows=25 width=0)
    Index Cond: (lpad((c1)::text, 5, '0')::text) = '00324'::text

(10 rows)
--删除。
gaussdb=# DROP INDEX idx_test_c1;
gaussdb=# DROP TABLE tbl_test;
```

9.5 开发设计

9.5.1 GaussDB 单表支持的最大列数？

答：1600列，但具体数值会因字段类型不同而变化。建表时不校验字段类型，数据存入时进行校验。例如bigint类型字段每个占用8字节数据，1600列总占用12800字节，超过单页容量8KB时插入会报错。

说明

字段超长时可能存储到toast表，具体逻辑如下：

- 元组总长度超过2KB时，优先尝试压缩最长列。
- 压缩后仍超限且字段长度超过toast元信息开销（24字节）时，才启用toast表存储。
- toast表以列为维度存储完整数据，主表仅保存查询元信息。
- 在压缩后，无论是否存放toast表（存toast表时，主表需额外存储toast表的元信息；不存toast表时，主表存放压缩后的列数据），单行总长度超过单页大小8KB时均会报错。

9.5.2 SQL 语言分为哪几类？

答：SQL语言分为3类，分别为：

- DDL（Data Definition Language，数据定义语言），用于定义或修改数据库中的对象，如：表、索引、视图等。
- DML（Data Manipulation Language，数据操作语言），用于对数据库表中的数据进行操作。如：插入、更新、查询、删除。
- DCL（Data Control Language，数据控制语言），用于设置或更改数据库用户或角色权限。

9.5.3 触发器的作用是什么？

答：触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。

- 可以强化约束，维护数据的完整性和一致性。
- 可以跟踪数据库内的操作，不允许未经许可的更新和变化。
- 可以联级运算。

9.5.4 函数和存储过程的区别是什么？

答：函数和存储过程是数据库中常见的两种对象，两者都可以封装一系列的SQL语句，以完成某些特定的操作。两者区别见[表9-2](#)：

表 9-2 函数和存储过程的区别

函数	存储过程
函数的标识符为FUNCTION。	存储过程标识符为PROCEDURE。
函数必须有返回值。	存储过程可以有返回值，也可以没有返回值。
函数可以输入输出参数。	存储过程可以输入输出参数。
函数只可以调用SELECT语句，不可以对表中的数据做出修改。	存储过程可以对表中的数据做出修改，如UPDATE、DELETE和INSERT等。
不可以函数中调用存储过程。	可以在存储过程中调用函数和存储过程。可以简化一系列复杂语句。
函数适用于需要返回单个值的情况，比如计算某个数据、字符串处理、返回表等。	存储过程适用于需要执行DML操作的情况，比如批量插入、更新、删除数据等。

9.5.5 如何查询当前数据库中有哪些触发器，以及触发器的定义？

答：可以使用如下方式查询出当前数据库中的触发器信息。

- 前置操作：创建表和触发器。

```
--建表。
gaussdb=# CREATE TABLE test(c1 int);

--创建触发器函数。
gaussdb=# CREATE OR REPLACE FUNCTION func_test() RETURNS TRIGGER AS
$$
DECLARE
BEGIN
    UPDATE test SET c1 = c1+200 WHERE id = NEW.id;
    RETURN NEW;
END$$ LANGUAGE plpgsql;

--创建触发器。
gaussdb=# CREATE TRIGGER trg_ins_test
AFTER INSERT ON test
FOR EACH ROW
EXECUTE PROCEDURE func_test();
```

- 查询当前数据库中的触发器和触发器的定义。

```
gaussdb=# SELECT tgname FROM pg_trigger;
tgname
-----
trg_ins_test
(1 row)

gaussdb=# SELECT pg_get_triggerdef(oid) FROM pg_trigger where tgname = 'trg_ins_test';
pg_get_triggerdef
-----
CREATE TRIGGER trg_ins_test AFTER INSERT ON test FOR EACH ROW EXECUTE PROCEDURE
func_test()
(1 row)

--查询触发器函数的定义。
gaussdb=# \sf func_test()
CREATE OR REPLACE FUNCTION public.func_test()
RETURNS trigger
LANGUAGE plpgsql
```

```
NOT FENCED NOT SHIPPABLE
AS $function$
DECLARE
BEGIN
    UPDATE test SET c1 = c1+200 WHERE id = NEW.id;
    RETURN NEW;END$function$;

--删除表。
--删除表之后触发器会被级联删除，触发器的相关函数不会被级联删除。
gaussdb=# DROP TABLE test;
gaussdb=# DROP FUNCTION func_test();
```

9.5.6 如何查询分区表分区信息和索引信息？

答：可以使用如下两种方式确认：

- 使用pg_get_tabledef()函数查看表的定义，返回信息包含建表SQL、注释、索引、约束等。

例如：

```
gaussdb=# SELECT pg_get_tabledef('table_name');
```

- 分区信息可以查询视图pg_partition，索引信息可以查询视图pg_indexes。

9.5.7 如何查询一个表涉及到的所有触发器？

答：通过如下SQL查询（请根据实际情况修改表名）。

```
--查询一个表涉及到的所有触发器。
gaussdb=# SELECT t1.relname table_name, t2.tgname trigger_name
FROM pg_class t1, pg_trigger t2
WHERE t1.oid = t2.tgrelid AND
      t1.relname = 'tablename';
```

9.5.8 数据库对象名称的长度是否有限制？

答：在创建数据库对象时，命名长度不可超过63个字节，超过数据库对象命名长度限制（63字节）时，数据库会将最后一个字节（而不是字符）截断，可能造成出现半个字符的情况。

9.5.9 选择数据库字符集时，应考虑哪些因素？

答：在选择数据库字符集时，应考虑以下因素：

- 应用支持语言的需求
- 已有数据的兼容性
- 性能要求
- 字符运算需求
- 跨平台的兼容性
- 存储空间和性能
- 国家化支持

9.6 其他

9.6.1 如何检查磁盘类型？

答：查看/sys/block/<磁盘名>/queue/rotational文件的数值。1代表旋转，表示磁盘是机械硬盘HDD，0代表非旋转，表示磁盘是固态硬盘SSD。

```
cat /sys/block/sdb/queue/rotational
0 ---固态硬盘SSD

cat /sys/block/sdb/queue/rotational
1 ---机械硬盘HDD
```

9.6.2 如何查询活跃元组、脏元组、脏元组比例？

答：查询方法如下：

```
--查询排名前10的活跃元组、脏元组以及脏元组比例。
gaussdb=# SELECT schemaname||'.'||relname AS
table_name,pg_size_pretty(pg_relation_size(schemaname||'.'||relname)) AS
table_size,n_dead_tup,n_live_tup,round(n_dead_tup * 100 / (n_live_tup + n_dead_tup),2) AS dead_tup_ratio
FROM pg_stat_all_tables WHERE n_dead_tup >= 1000 ORDER BY dead_tup_ratio DESC LIMIT 10;
```

table_name	table_size	n_dead_tup	n_live_tup	dead_tup_ratio
pg_catalog.pg_statistic	1136 kB	1425	647	68.77
snapshot.snap_global_stat_all_indexes	8072 kB	5559	16596	25.09
snapshot.snap_summary_stat_all_indexes	7672 kB	5466	16596	24.78
snapshot.snap_class_vital_info	3832 kB	3171	11094	22.23
snapshot.snap_global_file_iostat	11 MB	11333	46687	19.53
snapshot.snap_summary_file_iostat	10 MB	11325	46687	19.52
snapshot.snap_summary_statement	13 MB	3832	16131	19.20
snapshot.snap_summary_statio_all_indexes	8128 kB	4360	19952	17.93
snapshot.snap_global_statio_all_indexes	8456 kB	4275	19952	17.65
snapshot.snap_global_stat_all_tables	8216 kB	3038	16409	15.62

(10 rows)

说明

脏元组可作为判断表膨胀程度的指标，高比例的占比可能会导致表的膨胀，需要及时回收空间，避免影响数据库的性能。具体请参见《参考》中“SQL参考 > SQL语法 > V > VACUUM”章节。

9.6.3 VACUUM 与 VACUUM FULL 的主要区别？

答：在GaussDB中，VACUUM与VACUUM FULL都是用来清理表空间的命令，但它们有着明显的区别，区别如下：

- VACUUM：遍历表中的所有行，把表中的dead tuples行（即不再被任何活跃事务所引用的行）进行删除标记，并没有真正进行物理删除，将dead tuple行的空间标记为可重用的状态；VACUUM过程中不会锁表，其它事务可以正常访问数据表。
- VACUUM FULL：物理删除表中的dead tuples行，释放空间给操作系统；VACUUM FULL过程中表会被锁定，不允许其它事务访问。

VACUUM是一个轻量级的、不会阻塞数据库操作的命令，对数据库性能影响较小；而VACUUM FULL是一个重量级的、会阻塞数据库操作的命令，对数据库性能影响较大。在实际使用中，应根据数据库的负载情况和业务需求选择。

9.6.4 GaussDB KernelGaussDB 中 pg_terminate_backend 和 pg_terminate_session 有什么区别？

答：GaussDB中，pg_terminate_backend和pg_terminate_session都是用于终止数据库会话的命令。

- `pg_terminate_backend`命令是终止一个后台线程，它会强制终止该线程正在执行的任何操作，包括正在进行的事务。可以用于终止一个长时间运行的查询或者一个阻塞的事务。
- `pg_terminate_session`命令是终止一个后台session，它会终止该会话正在执行的任何操作，但不会强制终止正在进行的事务。可以用于终止一个用户的会话，不会影响其他用户的会话。

9.6.5 如何显示数据库文件信息？

答：`gs_ctlcontroldata`工具用来显示数据库数据目录下`pg_control`文件的信息。

```
--查看gs_ctlcontroldata工具的版本信息。
[omm1@node41 ~]$ gs_ctlcontroldata -V
gs_ctlcontroldata (GaussDB Kernel 505.2.RC1 build 939a176f) compiled at 2024-04-30 20:05:50 last mr 17511
release

--查找pg_control文件所在的路径。
[root@node41 ~]# find / -name pg_control
/opt/software/db5052/gaussdb/data/slave31/global/pg_control
/opt/software/db5052/gaussdb/data/coordinator/global/pg_control
/opt/software/db5052/gaussdb/data/master1/global/pg_control
/opt/software/db5052/gaussdb/data/slave21/global/pg_control

--这里以master1目录为例，查看pg_control文件的信息，DATADIR路径需写到上面查询出的路径的data层的下一
层目录。
[omm1@node41 ~]$ gs_ctlcontroldata /opt/software/db5052/gaussdb/data/master1
pg_control version number:          923
Catalog version number:            201611171
Database system identifier:         18020659789221943256
Database cluster state:             in production
pg_control last modified:           Sat 11 May 2024 10:43:58 AM CST
Latest checkpoint location:         0/134044D8
Prior checkpoint location:          0/13403BB8
Latest checkpoint's REDO location:  0/13404458
Latest checkpoint's TimeLineID:     1
Latest checkpoint's full_page_writes: off
Latest checkpoint's NextXID:        70025
Latest checkpoint's NextOID:        24576
Latest checkpoint's NextMultiXactId: 2
Latest checkpoint's NextMultiOffset: 0
Latest checkpoint's oldestXID:      3
Latest checkpoint's oldestXID's DB:  2
Latest checkpoint's oldestActiveXID: 70025
Latest checkpoint's remove_lsn:     0/1
Time of latest checkpoint:           Sat 11 May 2024 10:43:57 AM CST
Minimum recovery ending location:    0/0
Backup start location:               0/0
Backup end location:                 0/0
End-of-backup record required:       no
Current wal_level setting:           hot_standby
Current max_connections setting:     1084
Current max_prepared_xacts setting:  800
Current max_locks_per_xact setting:  256
Maximum data alignment:              8
Database block size:                 8192
Blocks per segment of large relation: 131072
WAL block size:                      8192
Bytes per WAL segment:               16777216
Maximum length of identifiers:       64
Maximum columns in an index:         32
Maximum size of a TOAST chunk:       1996
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Database system TimeLine:            1
```

9.6.6 GaussDB KernelGaussDB Xlog 日志自动回收的频率是多久？

答：GaussDB后台自动回收Xlog日志的线程是每一分钟会检测一次，回收满足条件的Xlog日志。

当日志量达到一定数量时，日志才会触发回收，该值的计算方式是：

$wal\_keep\_segments + checkpoint\_segments * 2 + 3$ ，假设checkpoint_segments设置128，wal_keep_segments设置128，日志量就是 $(128 + 128 * 2 + 3) * 16MB$ 。

9.6.7 什么是死元组？

答：GaussDB目前使用Astore存储引擎，数据的更新是通过追加方式实现的，即对数据执行UPDATE或DELETE操作只是被标识为删除状态，并未立即释放空间。这种处理对于多版本并发控制（MVCC）是必要的，如果一个元组的版本仍有可能被其它事务看到，那么就不能删除。

当事务提交后，过期元组版本将对事务不再有效，这些元组就会被标记为死元组。

死元组的产生也是因为GaussDB中没有单独的回滚段，而是将历史版本直接记录在数据页面中。过多的死元组会占用大量的存储空间，同时大量未清理的死元组存在，会导致索引指向大量无效页面，从而影响到SQL通过索引扫描的效率。

9.6.8 如何查询表的死元组？

答：使用pg_stat_all_tables查询死元组。

```
gaussdb=# SELECT relname,n_live_tup+n_dead_tup all_tup,n_dead_tup,last_autovacuum,last_autoanalyze
FROM pg_stat_all_tables WHERE relname='test02';
 relname | all_tup | n_dead_tup | last_autovacuum | last_autoanalyze
-----+-----+-----+-----+-----
 test02 | 500000 | 10000 | 2023-04-26 14:09:33.933982+08 | 2023-04-26 14:17:34.0364+08
```

9.6.9 什么是 OID？

答：对象标识符（OID，Object identifiers），是数据库对象的唯一标识，这个对象可以是数据库、表、索引、视图等。

9.6.10 什么是 UDF？

答：用户自定义函数（User Defined Function），用户可以根据自己的需求自定义一个函数嵌入到GaussDB中来实现特定的功能，以满足不同的业务场景。

9.6.11 存储过程嵌套调用的深度是多少？

答：存储过程调用存储过程的调用深度是没有限制的，但是不能超过最大栈深度，最大栈深度由GUC参数max_stack_depth控制，限制了会话中的存储过程嵌套调用的最大深度。

9.6.12 TOAST 是什么？

答：TOAST（The Oversized-Attribute Storage Technique）是GaussDB中的一个特殊表，主要用于存储超长数据字段。当一个元组的完整结构（包括字段）的长度超过了一个页面的大小限制时，GaussDB会启动TOAST机制，将这些字段压缩或切片成多个物理行存到另一张表（TOAST表）。

9.6.13 沙箱的作用是什么？

答：沙箱是一种安全机制。主要作用是给不受信任的程序提供一个隔离和受限的运行环境，使得高风险程序只能访问它需要访问的资源，其它不需要访问的资源禁止其访问，降低高风险程序自身的漏洞危害系统、危害其它进程、访问系统机密数据、隐私数据等风险。

9.6.14 什么是向量数据库？

答：向量数据库是一种专门设计用于存储、管理和检索高维度向量数据的数据库系统。与传统关系型数据库相比，它们在数据模型、索引机制和查询处理上有显著的不同。向量数据库通过优化的索引技术来提高非结构化数据的检索效率，同时，它们也能够支持复杂的数据分析和机器学习模型，使得数据处理更加高效和智能。

向量数据库的主要应用场景包括：

- 推荐系统：例如，在视频推荐、新闻推荐、购物推荐等场景。
- 图像识别：例如，通过图片搜索商品、人脸识别等场景。
- 非结构化数据处理：例如，音频、视频、图像等数据可以通过转化处理得到相应的向量数据，再进行关键性分析。
- 大模型智能问答系统：例如，向量数据库被用于改进语义搜索引擎，通过存储和检索文本内容的向量表达，提供更准确的搜索结果。

9.6.15 什么是GIN索引？

答：GIN（Generalized Inverted Index）通用倒排索引。当处理索引项为组合值的情况，查询时需要通过索引搜索出出现在组合值中的特定元素值。例如，文档是由多个单词组成，需要查询出文档中包含的特定单词。GIN索引大大加速模糊查询。

GIN索引使用满足以下条件：

- 在同一个表的同一个列上建立了多个GIN索引。
- 这些GIN索引使用了不同的parser（即分隔符不同）。
- 在查询中使用了该列，且执行计划中使用索引进行扫描。
- 为了避免使用不同GIN索引导致查询结果不同的问题，需要保证在物理表的一列上只有一个GIN索引可用。

9.6.16 存储过程可以调用触发器吗？

答：可以，存储过程可以通过执行SQL语句（如UPDATE）触发TRIGGER。

9.6.17 如何查询表中的随机几行？

答：可以使用如下方式查询表中随机几条数据：

```
--建表并插入100条数据。
gaussdb=# CREATE TABLE test(c1 int);
gaussdb=# INSERT INTO test VALUES (generate_series(1,100));

--查询表中随机4行数据。
gaussdb=# SELECT * FROM test ORDER BY random() limit 4;
c1
----
99
50
```

```
78
35
(4 rows)
```

9.6.18 为什么 SEQUENCE 的编号存在间隙？

答：为了提高并发性，每个会话按照序列的CACHE参数大小，在用户的私有内存空间缓存一定数量的序列值。会话结束后会丢弃会话缓存的序列值，因此序列值会出现跳跃。

```
--创建表和序列。
gaussdb=# CREATE SEQUENCE seq_test CACHE 10;

--使用nextval查看下一个序列的值，并退出会话。
gaussdb=# SELECT nextval('seq_test');
 nextval
-----
      1
(1 row)
gaussdb=# SELECT nextval('seq_test');
 nextval
-----
      2
(1 row)

--在另一个会话中使用nextval()查看下一个序列的值。
gaussdb=# SELECT nextval('seq_test');
 nextval
-----
     11
(1 row)

--删除序列。
gaussdb=# DROP SEQUENCE seq_test;
```

9.6.19 如何解决 power()函数计算结果数值太大时的误差？

答：一般power()函数计算结果大数值时有误差，例如2的60次方，准确值为1152921504606846976，使用SELECT power(2,60)的值为1.15292150460685e+18存在误差。具体解决办法如下：

```
gaussdb=# SELECT power(2,60);
 power
-----
1.15292150460685e+18
(1 row)

--如下查询结果表明：两个double precision类型的入参时返回值也是double precision类型，该类型精确到小数点后16位。
gaussdb=# \df power;
List of functions
Schema | Name | Result data type | Argument data types | Type | fencedmode | propackage |
prokind
-----+-----+-----+-----+-----+-----+-----+
pg_catalog | power | double precision | double precision, double precision | normal | f | f | f |
pg_catalog | power | numeric | numeric, numeric | normal | f | f | f |
(2 rows)

--使用如下方式查询。
gaussdb=# SELECT power(2::numeric,60::numeric);
 power
-----
1152921504606846976.0000000000000000
(1 row)

--使用round()函数取消小数点。
gaussdb=# SELECT round(power(2::numeric,60::numeric),0);
 round
```

```
-----  
1152921504606846976  
(1 row)
```

9.6.20 什么是 CTID?

答：CTID通过表中的块和位置来标识特定的物理行。

- 格式(blockid,itemid)：比如(0,1)中的0标识块id，表示在0块的第一条记录。
- 索引使用CTID来指向物理行。
- 行的CTID会在更新时候变化，因此CTID不能用作长期标识符。

9.6.21 如何查询表中某列第 N 大的数据?

答：可以使用窗口函数来实现。请根据实际情况选择row_number()、rank()和dense_rank()三个窗口函数其中之一。

```
--建表插入数据。  
gaussdb=# CREATE TABLE tbl_test(name varchar(20),tamo int);  
gaussdb=# INSERT INTO tbl_test VALUES('Bob',7000),('Jack',6500),('July',6800);  
gaussdb=# INSERT INTO tbl_test VALUES('Luna',7200),('Henry',7000),('John',8000);  
  
--使用如下SQL查询出交易额（tamo）排名第三的数据。  
gaussdb=# SELECT * FROM (SELECT *,dense_rank() over(order by tamo desc) rank FROM tbl_test) WHERE  
rank = 3;  
name | tamo | rank  
-----+-----+-----  
Bob | 7000 | 3  
Henry | 7000 | 3  
(2 rows)  
  
--删除。  
gaussdb=# DROP TABLE tbl_test;
```

9.6.22 如何查询字段中包含非数字的数据?

答：可以通过如下SQL来查询。

```
--建表插入数据。  
gaussdb=# CREATE TABLE tbl_test(c1 varchar);  
gaussdb=# INSERT INTO tbl_test VALUES('1'),('11'),('1a3'),('1b'),('d1');  
  
--查询数据。  
gaussdb=# SELECT * FROM tbl_test WHERE NOT regexp_like(c1,'^[[:digit:]]*$');  
c1  
-----  
1a3  
1b  
d1  
(3 rows)  
  
--删除。  
gaussdb=# DROP TABLE tbl_test;
```

9.6.23 如何修改列的顺序?

答：可以使用如下方式实现修改列的顺序。

```
--建表并插入数据。  
gaussdb=# CREATE TABLE tbl_test(  
id varchar(5),  
age int CHECK(age>0 and age<=200),  
name varchar(20),  
CONSTRAINT pk_test_id PRIMARY KEY (id)  
);
```

```
gaussdb=# INSERT INTO tbl_test VALUES ('00010',18,'Bob');
gaussdb=# INSERT INTO tbl_test VALUES ('00011',26,'Jack');
gaussdb=# INSERT INTO tbl_test VALUES ('00012',23,'Jim');

--查看表结构。
gaussdb=# \d tbl_test

--新建表tbl_test_new调换age和name字段的顺序。
gaussdb=# CREATE TABLE tbl_test_new(id varchar(5),name varchar(20),age int);
gaussdb=# ALTER TABLE tbl_test_new ADD CONSTRAINT pk_test_new_id PRIMARY KEY (id);
gaussdb=# ALTER TABLE tbl_test_new ADD CONSTRAINT chk_test_new_age CHECK(age>10 AND age<=200);

--新表插入数据。
gaussdb=# INSERT INTO tbl_test_new SELECT id,name,age FROM tbl_test;

--分别修改表名。
gaussdb=# ALTER TABLE tbl_test RENAME TO tbl_test_bak;
gaussdb=# ALTER TABLE tbl_test_new RENAME TO tbl_test;

--删除。
gaussdb=# DROP TABLE tbl_test;
gaussdb=# DROP TABLE tbl_test_bak;
```

9.6.24 count(*)和 count(column)的区别是什么？

答：count()函数用于计算行数的聚合函数，区别如下：

- count(*)统计包括了所有列的行数，在统计结果时不会忽略值为NULL的数据。
- count(column)只包括列名那一列，在统计结果时会忽略值为NULL的数据。

```
--建表并插入数据。
gaussdb=# CREATE TABLE test(c1 int);
gaussdb=# INSERT INTO test VALUES(1),(2),(3);
gaussdb=# INSERT INTO test VALUES(NULL);

--count(*)统计行数,不忽略值为NULL的数据。
gaussdb=# SELECT count(*) FROM test;
count
-----
4
(1 row)

--count(column)统计行数,忽略值为NULL的数据。
gaussdb=# SELECT count(c1) from test;
count
-----
3
(1 row)

--删除。
gaussdb=# DROP TABLE test;
```

9.6.25 GaussDB 都有哪些通配符，如何使用？

答：GaussDB数据库支持以下三种通配符：

- %：表示任意数量的字符，包括0个，用于LIKE和NOT LIKE语句中。
- _：表示一个字符，用于LIKE和 NOT LIKE语句中。
- *：表示任意数量字符，包括0个，在部分元命令中使用。

示例：

```
--任意数量字符的通配符使用，查询tbl_test表中col1字段以ab开头任意多个字符结尾的数据。
gaussdb=# SELECT * FROM tbl_test WHERE col1 LIKE 'ab%';

--一个字符的通配符使用，查询tbl_test表中col1字段以a开头b结尾中间任意单个字符串的数据。
```

```
gaussdb=# SELECT * FROM tbl_test WHERE col1 LIKE 'a_b';  
--查询所有以tbl开头的表。  
gaussdb=# \dt tbl*
```

9.6.26 LIMIT 2、LIMIT 2,3、LIMIT 2 OFFSET 3 的区别？

答：LIMIT和OFFSET是用于限制SELECT语句中查询数据返回结果数量的关键字。

- LIMIT 2：表示查询结果只返回前2行数据。
- LIMIT 2,3：表示查询结果从第3行开始返回，一共返回3行数据。
- LIMIT 2 OFFSET 3：表示查询结果从第4行开始返回，一共返回2行数据。

9.6.27 空字符串和 NULL 的关系？

答：GaussDB中对空字符串是否为NULL和数据库的兼容模式有关。

- ORA兼容模式中空字符串判断为NULL。
gaussdb=# CREATE DATABASE db_test1 DBCOMPATIBILITY = 'ORA';
gaussdb=# \c db_test1
db_test1=# SELECT " IS NULL;
?column?

t
(1 row)
- MYSQL兼容模式中空字符串判断不为NULL。
gaussdb=# CREATE DATABASE db_test2 DBCOMPATIBILITY = 'MYSQL';
gaussdb=# \c db_test2
db_test2=# SELECT " IS NULL;
?column?

f
(1 row)

9.6.28 日期/时间类型哪些带时区，哪些不带时区？

答：date、time、timestamp、smalldatetime、datea等类型不带时区，timetz、timestampz等类型带时区。

```
--timestamp类型不带时区。  
gaussdb=# SELECT '2019-01-01'::timestamp;  
timestamp  
-----  
2019-01-01 00:00:00  
(1 row)  
  
--timestampz类型带时区。  
gaussdb=# SELECT '2019-01-01'::timestampz;  
timestampz  
-----  
2019-01-01 00:00:00+08  
(1 row)
```

9.6.29 如果建表时没有指定分布列，数据会怎么存储？

答：建表时没有指定分布列，数据会以如下三种场景存储：

- 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
gaussdb=# CREATE TABLE tb_test1
(
W_WAREHOUSE_SK INTEGER PRIMARY KEY,

```
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tb_test1_pkey" for table "tb_test1"
CREATE TABLE

gaussdb=# SELECT getdistributekey('tb_test1');
getdistributekey
-----
w_warehouse_sk
(1 row)

--删除表。
gaussdb=# DROP TABLE tb_test1;
```

- 若建表时不包含主键/唯一约束，但存在数据类型支持作分布列的列，则选取HASH分布，分布列为第一个数据类型支持作分布列的列。

```
gaussdb=# CREATE TABLE tb_test2
(
  W_WAREHOUSE_SK      INTEGER          ,
  W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)
);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_sk' as the distribution
column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# SELECT getdistributekey('tb_test2');
getdistributekey
-----
w_warehouse_sk
(1 row)

--删除表。
gaussdb=# DROP TABLE tb_test2;
```

- 若建表时不包含主键/唯一约束，也不存在数据类型支持作分布列的列，选取ROUNDROBIN分布。

```
gaussdb=# CREATE TABLE tb_test3
(
  W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)
);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_id' as the distribution
column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# SELECT getdistributekey('tb_test3');
getdistributekey
-----
w_warehouse_id
(1 row)

--删除表。
gaussdb=# DROP TABLE tb_test3;
```

9.6.30 什么是数据倾斜，如何检查？

答：数据倾斜即多个DN内数据分布不均，对于hash分表策略，如果分布列选择不当，可能导致数据倾斜，查询时部分DN出现效率短板，从而影响整体查询性能。因此在采用Hash分表策略之后需对表的数据定期进行数据倾斜性检查或监控，以确保数据在各个DN上是均匀分布的。执行如下语句，查看各DN内元组数目。

```
gaussdb=# SELECT a.count,b.node_name FROM (SELECT count(*) AS count,xc_node_id FROM tablename
GROUP BY xc_node_id) a, pgxc_node b WHERE a.xc_node_id=b.node_id ORDER BY a.count DESC;
```

如果各DN内元组数目相差较大（如相差数倍、数十倍），则表明已发生数据倾斜现象。需要重新选择分布列，当前不支持通过ALTER TABLE语句调整分布列，因此调整分布列时需要重新建表。

9.6.31 使用 LIKE 进行模糊查找时如何忽略大小写？

答：可以将LIKE改为ILIKE，也可以使用upper或者lower函数。具体示例如下：

```
--前置操作创建表和插入数据。
gaussdb=# CREATE TABLE tbl_test1(c1 varchar);
gaussdb=# INSERT INTO tbl_test1 VALUES ('EEE'),('ABC'),('abc'),('aabccd');

--使用like进行模糊查询，结果过滤了所有大写的字符串，不符合预期。
gaussdb=# SELECT * FROM tbl_test1 WHERE c1 LIKE 'ab%';
 c1
----
 abc
(1 row)

--使用ILIKE进行模糊查询。
gaussdb=# SELECT * FROM tbl_test1 WHERE c1 ILIKE 'ab%';
 c1
----
 ABC
 abc
(2 rows)

--使用upper()函数。
gaussdb=# SELECT * FROM tbl_test1 WHERE upper(c1) LIKE 'AB%';
 c1
----
 ABC
 abc
(2 rows)

--删除。
gaussdb=# DROP TABLE tbl_test1;
```