ALIBABA CLOUD

阿里云

云原生分布式数据库 PolarDB-X 分布式关系型数据库服务公共云合集

文档版本：20220217

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.动态与公告 15
1.1. 新功能发布记录 15
1.1.1. 版本发布说明 15
1.1.2. 存储节点发布说明 21
1.2. 产品公告 24
1.2.1. 品牌升级 24
1.3. 安全公告 24
1.3.1. 【通知】Apache Log4j2漏洞对PolarDB-X 1.0/2.0无影响 24
2.产品简介 25
2.1. 什么是云原生分布式数据库 25
2.2. 历史 26
2.3. 适用场景 27
2.4. 产品计费 29
2.5. 存储资源包 31
2.6. 术语表 32
3.核心特性 36
3.1. 高可用性与容灾 36
3.2. 强一致分布式事务 36
3.3. 分布式线性扩展 38
3.4. 全局二级索引 40
3.5. 混合负载HTAP 41
3.6. 全局日志变更 43
3.7. 数据备份与恢复 43
3.8. 全链路监控和审计 44
3.9. MySQL生态兼容 45
4.技术白皮书 47

4.1. 行业趋势与背景 47
4.2. 技术架构 48
4.3. 技术原理 52
4.4. 部署与运维 56
4.5. 解决方案与客户案例 59
5.性能白皮书 64
5.1. TPC-C测试 64
5.2. TPC-H测试 68
5.3. Sysbench测试 74
6.快速入门 83
6.1. PolarDB-X首次使用流程 83
6.2. 创建实例 83
6.3. 创建数据库 85
6.4. 连接到数据库 85
6.5. 执行SQL基本操作 86
7.用户指南 89
7.1. 规格说明 89
7.2. 实例管理 90
7.2.1. 实例变配 90
7.2.2. 增加/减少节点 90
7.2.3. 释放实例 91
7.2.4. 查看和升级实例版本 91
7.2.5. 重启实例 93
7.2.6. 实例续费 93
7.2.7. 只读实例 93
7.3. 网络与连接 95
7.3.1. 连接数据库 95
7.3.2. 集群地址和只读地址 97

7.3.3. 申请和释放外网地址 99
7.4. 账号和安全 100
7.4.1. 注册和登录阿里云账号 100
7.4.2. 管理数据库账号 101
7.4.3. 设置白名单 104
7.4.4. 设置SSL加密 106
7.4.5. 设置透明数据加密 109
7.4.6. 三权分立 111
7.4.6.1. 功能介绍 111
7.4.6.2. 开启或关闭三权分立 112
7.4.6.3. 管理三权分立系统账号 114
7.4.6.4. 权限验证 115
7.5. 数据导入和导出 116
7.5.1. 数据同步生态工具 116
7.5.2. 使用DTS导入和导出数据 117
7.5.3. 使用mysqldump导入导出数据 117
7.5.4. 使用程序进行数据导入 120
7.5.5. 使用Batch Tool工具导入导出数据 122
7.6. 数据迁移和同步 125
7.6.1. PolarDB-X 1.0升级至PolarDB-X 2.0 125
7.7. 参数设置 131
7.8. 监控与告警 132
7.8.1. 计算资源监控 132
7.8.2. 存储资源监控 134
7.8.3. 配置告警 136
7.9. 诊断与优化 137
7.9.1. SQL洞察和审计 137
7.9.1.1. 搜索 137

7.9.1.2. SQL洞察 138
7.9.2. SQL审计与分析 140
7.9.2.1. 简介 140
7.9.2.2. 开启SQL审计与分析 141
7.9.2.3. 日志字段 142
7.9.2.4. 日志分析 144
7.9.2.5. 日志报表 148
7.9.3. 性能趋势 153
7.9.4. 实例会话 158
7.9.5. 慢日志 160
7.9.6. 空间分析 163
7.9.7. 实时性能 164
7.9.8. 容量评估 165
7.9.9. 锁分析 166
7.10. 备份与恢复 166
7.10.1. 备份数据 166
7.10.2. 恢复数据 170
7.10.3. 日志备份 173
7.11. 数据订阅 174
8.开发指南 178
8.1. 开发限制 178
8.2. SQL概述 180
8.3. 元数据库和数据字典 180
8.4. 数据类型 185
8.4.1. 数值类型 186
8.4.2. 字符串类型 186
8.4.3. Collation类型 186
8.4.4. 日期和时间类型 187

8.4.5. Json类型 187
8.5. 运算符 187
8.5.1. 逻辑运算符 187
8.5.2. 算术运算符 187
8.5.3. 比较运算符 188
8.5.4. 位运算符 188
8.5.5. 赋值运算符 189
8.5.6. 运算符优先级 189
8.6. 函数 190
8.6.1. 拆分函数 190
8.6.1.1. HASH 190
8.6.1.2. STR_HASH 191
8.6.1.3. UNI_HASH 193
8.6.1.4. RANGE_HASH 195
8.6.1.5. RIGHT_SHIFT 196
8.6.1.6. MM 196
8.6.1.7. DD 197
8.6.1.8. YYYYDD 197
8.6.1.9. WEEK 198
8.6.1.10. MMDD 199
8.6.1.11. YYYYMM 199
8.6.1.12. YYYYWEEK 200
8.6.2. 日期时间函数 201
8.6.3. 字符串函数 207
8.6.4. 转换函数 211
8.6.5. 聚合函数 212
8.6.6. 数学函数 213
8.6.7. 比较函数 217

8.6.8. 位函数 217
8.6.9. 流程控制函数 218
8.6.10. 信息函数 219
8.6.11. 加密和压缩函数 219
8.6.12. 窗口函数 221
8.6.13. JSON函数 224
8.6.14. 其他函数 235
8.7. DDL语句 235
8.7.1. 分库分表语法 236
8.7.1.1. CREATE DATABASE 236
8.7.1.2. DROP DATABASE 237
8.7.1.3. CREATE TABLE 238
8.7.1.4. DROP TABLE 253
8.7.1.5. ALTER TABLE 253
8.7.1.6. TRUNCATE TABLE 257
8.7.1.7. RENAME TABLE 258
8.7.1.8. CREATE INDEX 258
8.7.1.9. DROP INDEX 261
8.7.1.10. CREATE VIEW 261
8.7.1.11. DROP VIEW 262
8.7.2. 分区表语法 262
8.7.2.1. CREATE DATABASE 262
8.7.2.2. DROP DATABASE 264
8.7.2.3. CREATE TABLEGROUP 264
8.7.2.4. DROP TABLEGROUP 265
8.7.2.5. ALTER TABLEGROUP 265
8.7.2.6. CREATE TABLE 267
8.7.2.7. DROP TABLE 285

8.7.2.8. ALTER TABLE 285
8.7.2.9. TRUNCATE TABLE 289
8.7.2.10. RENAME TABLE 290
8.7.2.11. CREATE INDEX 290
8.7.2.12. DROP INDEX 293
8.7.2.13. CREATE VIEW 293
8.7.2.14. DROP VIEW 294
8.7.3. 秒级加字段（Instant Add Column） 294
8.8. DML语句 295
8.8.1. INSERT 295
8.8.2. REPLACE 296
8.8.3. UPDATE 297
8.8.4. DELETE 298
8.8.5. SOURCE 298
8.8.6. EXPORT 299
8.8.7. LOAD DATA 300
8.9. DQL语句 302
8.9.1. SELECT 302
8.9.2. 子查询 304
8.9.3. Grouping Sets、Rollup和Cube扩展 307
8.10. TCL语句 315
8.11. DAL语句 316
8.11.1. DDL管理语句 316
8.11.1.1. SHOW DDL 316
8.11.1.2. SHOW DDL RESULT 318
8.11.1.3. CANCEL DDL 318
8.11.1.4. PAUSE DDL 319
8.11.1.5. CONTINUE DDL 319

8.11.2. SHOW 319
8.11.2.1. SHOW HELP 319
8.11.2.2. 规则和拓扑查询语句 321
8.11.2.3. 慢SQL相关 325
8.11.2.4. 统计信息查询 327
8.11.2.5. SHOW PROCESSLIST 331
8.11.2.6. SHOW GLOBAL INDEX 333
8.11.2.7. SHOW INDEX 335
8.11.2.8. SHOW METADATA LOCK 336
8.11.2.9. SHOW TABLEGROUP 337
8.11.3. SET 337
8.11.3.1. SET语句变量设置 338
8.11.3.2. SET CHARACTER SET 338
8.11.3.3. SET NAMES 339
8.11.4. CHECK TABLE 340
8.11.5. CHECK GLOBAL INDEX 341
8.11.6. KILL 343
8.11.7. USE 344
8.11.8. EXPLAIN 344
8.11.9. SQL限流 349
8.12. Prepared语句 354
8.13. 通过LOCALITY指定库或表的存储位置 355
8.14. 权限管理 357
8.14.1. 账号权限管理 357
8.14.2. 角色权限管理 359
8.14.3. 三权分立下的权限管理 363
8.15. Replication 365
8.16. Sequence 367

8.16.1. 概述 368
8.16.2. 使用限制 370
8.16.3. 显式用法 371
8.16.3.1. 创建Sequence 371
8.16.3.2. 修改Sequence 373
8.16.3.3. 查询与获取Sequence 375
8.16.3.4. 删除Sequence 377
8.16.4. 隐式用法 377
8.16.4.1. 创建 Sequence 378
8.16.4.2. 修改 Sequence 379
8.16.4.3. 查看表信息及相关Sequence类型 379
8.17. TTL表功能 382
8.17.1. 什么是TTL功能 382
8.17.2. 创建TTL表 383
8.17.3. 变更TTL表 385
8.17.4. TTL表的定时任务 387
8.18. 变量 388
8.18.1. 计算层变量 388
8.18.2. 存储层变量 396
8.19. 透明分布式 413
8.19.1. 主键拆分 413
8.19.2. 自动分区模式下的主键拆分 416
8.19.3. 聚簇索引 421
8.19.4. 变更表类型及拆分规则 423
8.19.5. 变更表类型及分区策略 426
8.20. 常见问题 430
8.20.1. DDL常见问题 430
8.20.2. 如何使用全局二级索引 430

8.20.3. 如何使用HINT 435
8.20.4. INDEX HINT 436
8.20.5. 如何自定义SQL超时时间 437
8.20.6. 指定分库执行SQL 437
8.20.7. 扫描全部/部分分库分表 440
9.最佳实践 443
9.1. 如何选择实例规格 443
9.2. 如何选择升配与扩容 444
9.3. 如何选择应用端链接池 445
9.4. 如何快速定位及解决数据库问题 448
9.5. 如何正确处理DDL异常 450
9.6. 如何分析及优化慢SQL 451
9.7. 如何限流慢SQL 458
9.8. 如何异步删除大表 464
9.9. 如何支持热点更新场景 465
9.10. 如何分析数据分布不均衡 468
9.11. 如何高效使用IN查询 472
9.12. 如何优化Batch Insert 476
9.13. 如何优化数据导入导出 481
9.14. 如何优化数据全量抽取 484
9.15. 如何使用Blob对象 486
9.16. 偏高并发场景的实践和优化 487
9.17. 偏分析场景的实践和优化 490
9.18. 混合负载HTAP的实践和优化 494
9.19. 如何支持超大事务 501
10.错误码 508
11.PolarDB-X 1.0 521
11.1. 版本变更记录 521

11.1.1. 实例版本概览 521
11.1.2. V5.4.x系列实例版本说明 521
11.1.3. V5.3.x系列实例版本说明 534
11.1.4. V5.2.x系列实例版本说明 542
11.1.5. V5.1.x系列实例版本说明 544
11.2. 技术白皮书 553
11.2.1. 技术架构 553
11.2.2. 技术原理 555
11.2.3. 解决方案与客户案例 559
11.3. API参考 562

1.动态与公告

1.1. 新功能发布记录

1.1.1. 版本发布说明

本文介绍PolarDB-X实例版本的发布说明。

发布日期：2022-01-07

大版本：V5.4.13

小版本：16415631

类别	说明
新增	支持创建数据库时指定建表的模式（新的分区表模式与老的分库分表模式），默认为分库分表模式，具体语法请参见CREATE DATABASE。
	支持使用MySQL分区表语法创建一级分区的分区表，分区策略包括Hash/Range/List等，详细内容请参见分区类型介绍。
	支持分区表的动态裁剪能力，包括支持分区列条件的常量折叠、区间合并以及前缀查询裁剪等功能。
	支持分区表的JOIN计算下推。
	提供分区表的分区管理能力，包括分区的添加、删除、分裂、合并与迁移等功能，详细内容请参见ALTER TABLEGROUP。
	提供表组及其管理能力（包括表组的创建、删除、变更等），支持分区变更期间JOIN计算下推不受影响，详细内容请参见ALTER TABLEGROUP。
	支持全局索引表使用MySQL分区表语法并按Hash/Range/List等分区策略进行分区，详细内容请参见CREATE INDEX。
	自动拆分支持使用分区表语法，详细内容请参见主键拆分。
	拆分变更增加支持分区表，详细内容请参见自动分区模式下的主键拆分。
	新分区表GSI自动拆分会携带主键，可以处理GSI热点问题。
	支持实例的扩容与缩容。
	支持分区表的TTL及其管理能力（包括调整TTL的初始时间与时间间隔等），详细内容请参见CREATE TABLE。
	SQL Advisor支持推荐广播表。
	支持Instant Add Column功能。
	支持Explain Statistics拉取优化器优化需要的所有信息。
	支持使用SHOW [GLOBAL|LOCAL] DEADLOCKS查看最近一次死锁信息。
	限制cbo的搜索空间，减少复杂查询的优化耗时。
优化	优化Check Table指令，支持校验主表分区、索引表分区与列定义等元数据的一致性。
	优化部分DDL后台操作的数据校验任务的性能，使GSI/扩缩容DDL变更操作加速。
	修复heuristic join reorder的空指针和排序报错的问题。
	修复AccessPathRule导致的RelNode rowtype错误的问题。

类别	说明
缺陷修复	修复当collate为utf8mb4_0900_ai_ci时，创建GSI导致数据不一致的问题。
	修复最大登录次数无法生效的问题。
	修复某些block中estimateSize不准确的问题。
	修复私有协议下Deciaml精度缺失的问题。
	修复私有协议下执行计划传输byte和char类型对比报错的问题。
	修复Alter Table Add/Drop Primary Key后索引元数据缺失的问题。
	修复Alter Table期间外部中断或物理分片超时导致的表结构不一致问题。
	修复Drop index所有物理分片都失败时不自动回滚的问题。

发布日期：2021-12-22

大版本：V5.4.12

小版本：16399913

类别	说明
新增	增加对debezium/maxwel同步工具的兼容。
	增加探测到事务死锁后，事务自动回滚的能力。
优化	优化SPM能力，确保不受DDL影响而失效。
缺陷修复	修复包含隐式主键的insert select语句，在不下推场景下主键值未被sequence合理填充的问题。

发布日期：2021-11-25

大版本：V5.4.12

小版本：16378205

类别	说明
新增	增加ALTER TABLE校验，拒绝执行不支持ALTER TABLE校验的语句。
	在主实例上引入读写分离能力。
优化	优化连接池采用三层数据源的概念影响show ds status和kill all指令的问题。
缺陷修复	修复长事务下容易出现Snapshot too old报错异常的问题。
	修复带有中划线的库名在DTS导数据中报错的问题。
	修复mysqldump报错问题。
	修复CN发生切换主实例时，可能2个CN同时执行同一个DDL任务的问题。
	修复PolarDB-X禁用Alter Table Add Foreign Key的问题。
	修复后台线程无法清理审计日志的问题。
	修复由于未及时维护CN和DN的映射关系，使得未合理建立连接池，导致scale out任务中断的问题。
	修复统计信息涉及索引部分场景下NPE问题。

发布日期：2021-10-28

大版本：V5.4.12 hotfix

小版本：16349923

类别	说明
新增	在主实例上引入基于规则的读写分离能力。
	明确set指令的支持边界，set指令暂不支持包含表达式的计算。
优化	优化死锁检测逻辑，确保陷入到死锁的事务可以被快速中止。
缺陷修复	修复sql_mode设置语句携带表达式时，导致物理建连失败的问题。
	修复跨分片场景中包含limit的物理SQL由于受SQL cache的影响，存在查询结果不符合预期的问题。
	修复2.0模式下新RPC协议，存储节点可能低概率出现拒绝服务的问题。

发布日期：2021-09-14

大版本：V5.4.12

小版本：16315258

类别	说明
新增	新增全新DDL执行引擎。
	支持节点缩容。
	优化器支持Cascades Style物理属性传递。
	支持AES加解密函数。
	新增自动KILL阻塞DDL的长事务的机制。
	支持UPDATE拆分键为表的首列且字段类型为时间的场景。
优化	优化go-sql-driver在prepare协议下，兼容0000-00-00 00:00:00时间的问题。
	优化创建聚簇索引时索引名大小写敏感的问题。
	优化部分场景下INSERT SELECT的事务可见性问题。
	优化DDL分表下的全并行执行策略。
	优化Create/Drop Table过程中，逻辑表的可见性。
	优化复杂DML过程中，由于数据过多易出现执行堆栈溢出问题。
	优化START TRANSACTION READ ONLY事务策略的稳定性，确保该事务开启后禁止数据变更操作。
缺陷修复	修复date_sub函数计算异常问题。
	修复高精度Decimal在特殊情况下解码出错的问题。
	修复点查命中带subpart的索引时，可能触发返回过多数据的问题。
	修复广播表上包含子查询的可下推单表DELETE，子查询中指定库名导致报错的问题。

发布日期：2021-07-05

大版本：V5.4.11

小版本：16251897

类别	说明
新增	支持create table like。
	支持select into outfile语句，可以按指定分隔符和格式导出数据。
	支持show table info from <tablename>语句，可以查看各分片的数据量。
	支持SPM根据不同的参数空间选择不同plan的能力。
	单表和广播表支持随机物理表名。
	优化器支持Cascades搜索空间剪枝。
	更加完善的HTAP能力： • 支持读写分离设置； • 支持利用SPM修正workload； • 支持通过Hint指定路由。
	支持AliSQL秒杀热点的语法。
	支持information_schema buffer相关的系统表：INNODB_BUFFER_POOL_STATS、INNODB_BUFFER_PAGE_LRU 和 INNODB_BUFFER_PAGE。
	新增GENERAL_DYNAMIC_SPEED_LIMITATION参数用于动态调整GSI回填校验和scaleout的限速。
优化	优化统计信息的处理过程，增加TopN用于处理数据倾斜的情况。
	优化统计信息收集过程，减少IO消耗。
	优化私有协议使用原生NIO作为网络层。
缺陷修复	修复的问题如下： • 兼容已下线的事务超时系统变量drds_transaction_timeout。 • 执行语句alter table drop key之后元数据不一致。 • load data导数据过程中对齐mysql对空字符的处理语义。 • 兼容create database对charset和collation的定义。

发布日期：2021-05-26

大版本：V5.4.10

小版本：16219368

类别	说明
缺陷修复	修复的问题如下： • 显示设置 `MERGE_UNION_SIZE` 参数后会导致分布式事务内出现单分库多连接的风险。 • 部分情况下事务内有读写操作时易抛出"multiple read connections are not allowed" 。 • DELETE语句获取表名错误时未加MDL。 • 部分场景下未能正确填充Sequence。 • 特定字符串Collation条件下，GSI表数据不一致。 • NOW函数在涉及事务的部分场景下的更新不及时。

发布日期：2021-05-07

大版本：V5.4.10

小版本：16202941

类别	说明
缺陷修复	• 实例重启后，三权分立模式会失效。 • 通过ALTER TABLE执行DROP PRIMARY KEY或ADD PRIMARY KEY操作后，元数据没有变更。 • 部分场景下物理SQL丢失 `traceId` 。

发布日期：2021-04-25

大版本：V5.4.10

小版本：16193390

类别	说明
新增	支持LOCALITY语法，用于在创建库表时指定该库表的存储节点。
	支持单表、拆分表和广播表之间的相互转换。
	支持GROUPING SETS、ROLLUP和CUBE语法，用于按照多组维度对查询结果进行分组。
	支持基于角色（Role）的权限控制。
	支持三权分立模式。
	支持0~6位时间精度设置。
	• 支持包含全局二级索引的索引推荐。 • 支持针对分布式元数据锁的死锁检测。 • 支持在READ COMMITTED隔离级别下使用TSO事务。 • 在计算节点和存储节点间采用了新的连接池系统。 说明 ◦ 5.4.10版本发布后创建的实例支持该功能，之前创建的实例不支持。 ◦ 该功能默认开启，如需关闭，请提交工单联系技术支持。 • 限制只读实例下的Sequence访问。 • 支持通过 `show global variables like '%ssl%'` 查看SSL连接状态。
优化	• 优化广播表查询性能，避免仅将查询路由到0库导致0库压力过大的问题。 • 优化占用大内存SQL的执行，提升SQL执行速度。 • 优化HashJoin、NLJoin和SemiHashJon的执行方式，使其能按流式方式执行，提升执行速度。 • 优化查询优化器的算法，用于消除JOIN、FILTER、基于BKA（Batched Key Access）算法的IN条件中的重复条件。 • 将默认数据库迁移任务回填限速从10 KB调整到100 KB，提升回填速度。 • 优化SCALEOUT任务执行速度，支持SCALEOUT任务逻辑库级并发。 • 优化部分子查询的执行速度。

类别	说明
缺陷修复	• 修复当通过同一连接重建库时，导致元数据异常的问题。 • 修复 `sql_mode` 值不合法导致SQL语句执行时会报错的问题。 • 修复当分布式死锁出现异常时处理不当的问题。 • 修复在MySQL高可用切换过程中，低版本的Java驱动会断连的问题。 • 修复相关联子查询条件推导结果中缺少OR条件，导致分区裁剪结果不正确的问题。 • 修复当多语句（即用英文分号（;）分割的SQL语句）中包含SET AUTOCOMMIT命令时，执行结果不正确的问题。 • 修复PREPARE协议下异常传递丢失的问题及元数据获取报错的问题。 • 修复某些情况下，当执行包含Interval函数的语句时，物理SQL会报错的问题。 • 修复全局二级索引更新过程中报错，导致事务未释放的问题。 • 修复为单表创建独立的关联Sequence后，当执行INSERT命令时，Sequence会不生效的问题。 • 修复当多个逻辑库同时迁移时，概率性出现迁移任务失败的问题。 • 修复时间类型与函数的兼容性问题，提升对时间类型数据的处理效率。 • 修复监控中的连接数显示不正确的问题。 • 修复ALTER TABLE语句中CONVERT TO CHARACTER SET不能同时修改GSI索引表的问题。

发布日期：2021-03-15

大版本：V5.4.9

小版本：16155600

类别	说明
新增	全局Binlog功能。
优化	提升了存储节点在分布式事务下的稳定性。
缺陷修复	• 高并发场景下存储节点可能崩溃的问题。 • 事务超时机制不正确导致的小概率出现前端连接断开的问题。 • Prepare模式报错不明确的问题。 • Prepare模式部分情况下变量没有对齐的问题。 • 当分片数据不均时，小概率出现查询数据缺失的问题。

发布日期：2021-03-01

大版本：V5.4.9

小版本：16142062

类别	说明
新增	兼容MySQL Collation特性，新增支持若干Collation来确保SQL语句字符串排序的正确性。
	支持主键拆分功能。
	支持聚簇索引功能。
	支持全部MySQL窗口函数。
	支持局部索引智能推荐功能。

类别	说明
	• 支持DELETE语句为单表指定别名。 • 支持ALTER VIEW语法。 • 支持在SQL语句中使用_binary 'data...'，来表示二进制数据。 • 支持JOIN和DML语句中的自动全局二级索引选择。 • 支持事务中的复杂查询。 • PREPARE协议新增支持UNSIGNED数据类型。 • 支持单表下推场景下的变量赋值语法。 • 默认禁止执行未包含条件的UPDATE或DELETE语句。

1.1.2. 存储节点发布说明

本文为您介绍了PolarDB-X存储节点的发布说明。

存储节点的版本随着实例版本升级同步更新，一个完整的PolarDB-X版本号由前半部分的计算节点版本号和后半部分的存储节点版本号组成，例如版本号“polarx-kernel_5.4.12-16349923_xcluster-20210926”中，存储节点版本号为xcluster-20210926。

发布日期：2021-09-26

版本号：xcluster-20210926

类别	说明
新增	新增instant add column。
	新增UDF fast hash checker。
修复	修复私有协议token并发问题。
	修复私有协议网络层低概率拒绝服务的问题。

发布日期：2021-08-05

版本号：xcluster-20210805

类别	说明
新增	支持returning语法。
	支持TSO分配使用无锁算法。

发布日期：2021-07-28

版本号：xcluster-20210728

类别	说明
修复	修复sampling crash问题。

发布日期：2021-07-19

版本号：xcluster-20210719

类别	说明
优化	优化日志内容。
	修复私有协议session vio crash问题。

篓翥	说明
	修复GTS lease不存在时的table泄漏问题。

发布日期：2021-07-07

版本号：xcluster-20210707

类别	说明
修复	修复私有协议session释放不掉的问题。

发布日期：2021-07-01

版本号：xcluster-20210701

类别	说明
优化	优化单分片事务访问流程。
修复	修复TSO batch不合理问题，重构分配算法。

发布日期：2021-06-28

版本号：xcluster-20210628

类别	说明
修复	增加私有协议执行计划构建表达式时的兜底报错。

发布日期：2021-06-24

版本号：xcluster-20210624

类别	说明
新增	新增auto savepoint功能。
	新增InnoDB层随机采样。
	新增默认强制启动polarx udf。
	新增行数和索引feed back。
修复	修复私有协议上事务sequence泄漏的问题。
	修复私有协议double极限精度错误问题。
	修复物理统计信息缺失问题。
	修复hyper log的崩溃问题。
	修复batch TSO失效和TSO分配可能重复的问题。
	修复私有协议utf8mb4作为filter算子的比较参数时的crash问题。

发布日期：2021-03-30

版本号：xcluster-20210330

类别	说明
新增	新增私有协议sql/plan cache。
	新增私有协议chunk传输。
修复	修复TSO事务和DDL混用的crash问题。

发布日期：2021-03-11

版本号：xcluster-20210311

类别	说明
新增	新增SQL闪回功能。
修复	修复非预期prepare wait timeout问题。
	修复高并发情况下xdb crash的问题。

发布日期：2021-02-04

版本号：xcluster-20210204

类别	说明
新增	mysqldump支持基于指定snapshot sequence的dump。
修复	修复"enable_gts"开启的情况下，follower重启后会从老的位点重放binlog events。
	修复leader重新拉起时恢复失败的问题。
	默认GTS的lease设置为2s。
	私有协议支持message中传递CTS，支持sql、plan cache。

发布日期：2020-12-12

版本号：xcluster-20201212

类别	说明
新增	新增消息中传递TSO时间戳。
	支持普通事务1PC提交优化。
修复	修复bloom filter udf中注册的return type不正确导致崩溃。
	修复bloom filter udf中对null值和decimal值的处理。
	修复CTS事务走索引的可见性问题。
	修复1pc的一个崩溃点。
	修复XA相关的若干问题。

发布日期：2020-09-23

版本号：xcluster-20200923

类别	说明
	添加CTS的持久化开关。

类别	说明
新增	
	新增bloom filter UDF
	新增hyper log的log UDF。
	默认加载UDF模块polarx_udf.so。
优化	优化私有协议鉴权流程。
修复	修复assert导致xdb崩溃的问题。
	去掉不需要的私有协议log。

发布日期：2020-09-10

版本号：xcluster-20200910

类别	说明
新增	新增PolarDB-X私有协议。
	新增TSO服务。
	新增基于时间戳的分布式事务。
	新增主备一致性读支持。
修复	修复XA事物在binlog rotate时发生server crash的情况下，重启后无法正确恢复。
	修复内部锁相关的问题导致XA事物在在slave端回放失败的问题。
	修复XA prepare event复制相关的位置信息写入磁盘不正确问题。

1.2. 产品公告

1.2.1. 品牌升级

分布式关系型数据库服务DRDS目前已品牌升级至云原生分布式数据库PolarDB-X。

PolarDB-X融合分布式SQL引擎与分布式自研存储X-DB，专注解决海量数据存储、超高并发吞吐、大表瓶颈以及复杂计算效率等数据库瓶颈问题，历经各届天猫双十一及阿里云各行业客户业务的考验。

此次品牌升级，存量DRDS实例不受影响，可正常续费使用。

1.3. 安全公告

1.3.1. 【通知】Apache Log4j2漏洞对PolarDB-X 1.0/2.0无影响

近期，阿里云安全团队向Apache官方报告了Apache Log4j2远程代码执行漏洞（CVE-2021-44228），云原生分布式数据库PolarDB-X技术团队确认该漏洞对云原生分布式数据库PolarDB-X 1.0/2.0无影响。

更多信息，请参见【漏洞预警】Apache Log4j2 远程代码执行漏洞（CVE-2021-44228）。

2.产品简介

2.1. 什么是云原生分布式数据库

本文介绍什么是云原生分布式数据库PolarDB-X（简称PolarDB-X）。

产品简介

PolarDB-X是阿里巴巴自主设计研发的高性能云原生分布式数据库产品，为用户提供高吞吐、大存储、低延时、易扩展和超高可用的云时代数据库服务。

PolarDB-X始终保持对阿里巴巴集团“双十一购物狂欢节”所有相关业务的全面支撑。历经十余年淬炼，PolarDB-X具备了强数据一致性、高系统稳定性、快速集群弹性等核心关键特性，并在司法财税、交通物流、电力能源等公共事业领域有广泛深入应用。

PolarDB-X坚定遵循自主可控、开放生态的发展思路，持续围绕MySQL开源生态构建分布式能力，以求最大程度降低用户的学习使用成本。

产品特点

云原生+MySQL生态

PolarDB-X已作为标准云产品在世界范围内的13个地区提供服务。依托云资源和容器化部署能力，PolarDB-X可以在数分钟内完成集群创建和变配，整个过程中用户无需进行手工干预。同时PolarDB-X支持按量付费模式，从而帮助用户精准降本。

阿里云及开源社区的多种生态工具对PolarDB-X持续提供不断完善的支持，包括但不限于以下产品：数据传输服务DTS、数据库备份DBS、数据管理服务DMS、数据库自治服务DAS、数据集成Data Integration、云监控、性能测试PTS。

PolarDB-X积极拥抱并努力回馈MySQL生态，目前已经形成对MySQL生态从协议、语法、事务行为、账号体系、安全到命令行工具的全方位兼容。

存储计算分离架构

旨在最大限度地发挥其云数据库的弹性扩展能力，PolarDB-X采用了基于存储计算分离的Shared-Nothing系统架构，该架构使用户可以根据业务需要进行分层容量规划。

PolarDB-X的存储节点（DN）基于阿里巴巴自研的跨可用区部署的三节点强一致数据库X-DB构建。X-DB使用InnoDB引擎，提供MySQL语法全兼容能力，以及对复杂查询的处理能力。X-DB结合PolarDB-X面向HTAP的CBO查询优化器，可精确控制计算下推行为，从而获得更佳的整体性能。

透明分布式体验

让用户以使用单机MySQL数据库的体验，操作分布式数据库是PolarDB-X一贯追求的目标。为此PolarDB-X提供了简单易用的透明分布式能力：

默认主键拆分，让移植到PolarDB-X的数据和业务摆脱对设计“分区键”的依赖。

高性能强一致分布式事务，PolarDB-X采用自研X-Paxos协议，保证数据存储在故障切换过程中RPO=0的基础上，使用TSO策略和分布式的MVCC能力保证了分布式事务的隔离性和一致性。

分布式线性扩展，PolarDB-X基于一致性Hash的分区策略，有效的进行负载均衡和热点抑制，且在扩展过程中保持计算下推和数据一致性的同时实现业务零感知，并行和流控能力为扩展期间业务连续性提供了有力保障。

全局Binlog和全局一致性备份，分别解决分布式数据库各节点数据库向下游流转的难题及各节点备份时间差造成的恢复一致性问题。

多种部署形态

为满足不同行业客户对部署形态的需求，PolarDB-X提供公共云、专有云、专有云DBStack、软件版四种部署形态：

- 公共云：高速迭代，稳定服务，完全托管。目前面向世界范围内13个地区提供高性能云原生分布式数据库服务。
- 专有云（ApsaraStack）：集成阿里云核心产品，满足对安全性、隔离性有合规要求的行业客户。

> **说明** 由于部署资源差异，专有云和公共云的软件版本或有差异。

- 专有云DBStack：轻量级数据库管理服务平台，集成阿里云核心数据库产品，满足构建高性能、高可用、低成本的全场景数据库解决方案用户需求。
- 软件版（PolarDB-X Lite）：在Lite版本中，用户可体验最新的产品特性并以最小资源构建一个分布式数据库集群。

2.2. 历史

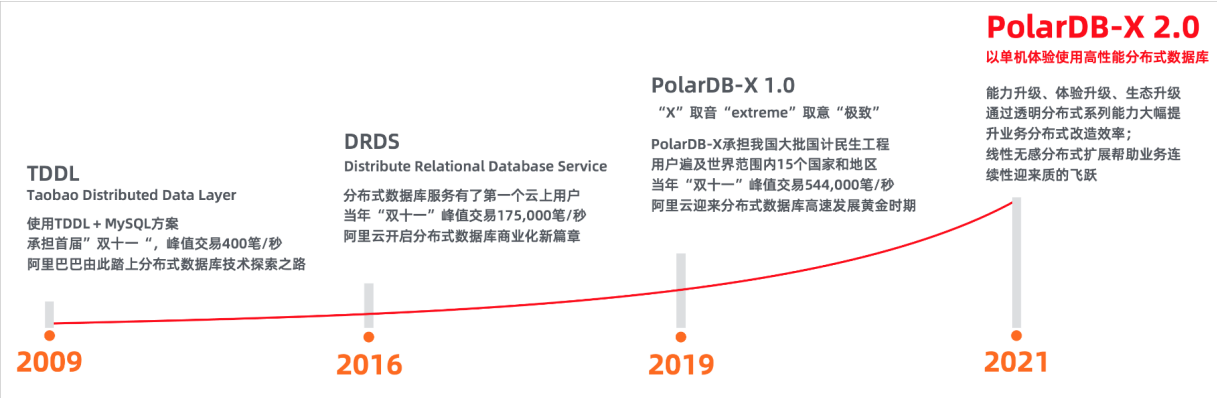
本文介绍了PolarDB-X产品的发展历史。

前言

2009年，阿里巴巴在进行年度收支核算时，发现业务对数据存取资源的需求呈指数级扩张，并与业务指标增长之间产生鸿沟。此时的阿里巴巴清晰的意识到，如果不通过技术创新平衡成本和增长需求，必将对未来发展产生影响。

于是"去IOE"的说法被首次提出。"去IOE"的目的绝不仅仅是摆脱几个传统IT厂商的产品，而是在互联网+、云计算快速兴起后，企业用户迫切需要能够保证其业务发展和创新的更加开放、灵活、高效的IT基础架构。概括地说，"去IOE"推动了用横向扩展取代纵向扩展，用开源软件代替商业软件的进程，可以看作是云计算的奠基之举。

其中对传统商业数据库的替换又是"去IOE"的核心。



TDDL阶段

关键字：阿里巴巴大规模应用；分库分表技术开创者

去掉传统商业数据库后，是否有更适合的产品和解决方案来替代呢？对于数据库来说，答案是明确的：开源+分布式，开源解决成本问题，分布式解决性能和容量问题。

同年11月11日，TDDL（Taobao Distributed Data Layer）首次发布，开创了分布式数据库中间件+开源数据库应用在高并发交易系统的先河。

当时的TDDL虽然是一个客户端jar，但创造性地提出了三层（Matrix、Group、Atom）拆分拓扑结构，满足应用按需制定拆分策略的同时，解决了弹性扩容、本地高可用等企业应用难题。

2011~2015，TDDL成为阿里巴巴数据库系统的统一接入标准，开始面向阿里巴巴所有业务提供分布式数据库服务。目前集团内运行实例约30万套，业务覆盖支付、资金、即时通信、媒体等十余大类。

丰富的业务模型造就了TDDL优秀的MySQL语法兼容性，庞大的业务规模使TDDL打磨出优异的内核稳定性，历年双十一的加持孵化了TDDL业界顶尖的高性能高吞吐。

与此同时，阿里巴巴分布式数据库的商业化进程悄然启动。

DRDS阶段

关键字：云端商业化；高性能SQL引擎

DRDS（Distributed Relational Database Service）于2016年初迎来了第一个公有云付费客户。自此，DRDS一直在不断努力提升单位资源的处理能力，以求最大限度帮助客户降本增效。

DRDS研发团队于2017年发布的新一代高性能分布式SQL引擎，通过PlanCache、FastSQL、定制化的底层驱动使Batch写、含拆分键Select、读写分离等操作具有300%的性能提升；跨库聚合、分布式Join、分布式事务等操作具有200%性能提升。

DRDS提供更低使用成本，包括对不同维度的表的Join操作的支持、内存中二次排序的支持和对内存结果做函数计算的支持等。

DRDS还针对分布式数据库使用场景提供一系列的企业级特性，包括全局Sequence服务、读写分离、数据库账号体系和DRDS后台运维指令集。

凭借优异性能和相对优良的体验，DRDS迅速在公有云积累了一批忠实用户。

DRDS的商业化成功，标志着阿里巴巴分布式数据库技术完成了从内部孵化到市场化运营的阶段性转变，以及从分布式数据库中间件到分布式数据库系统实质性跨越。

PolarDB-X 1.0阶段

关键字：架构与品牌升级；国计民生项目

2018~2019年，DRDS凭借优异稳定性、超高性能以及丰富的企业特性，承接众多政企行业的国计民生项目，积极投入我国信息系统基础设施数字化转型建设，品牌声誉得到大幅提升，逐步成长为代表阿里巴巴的名片级产品。

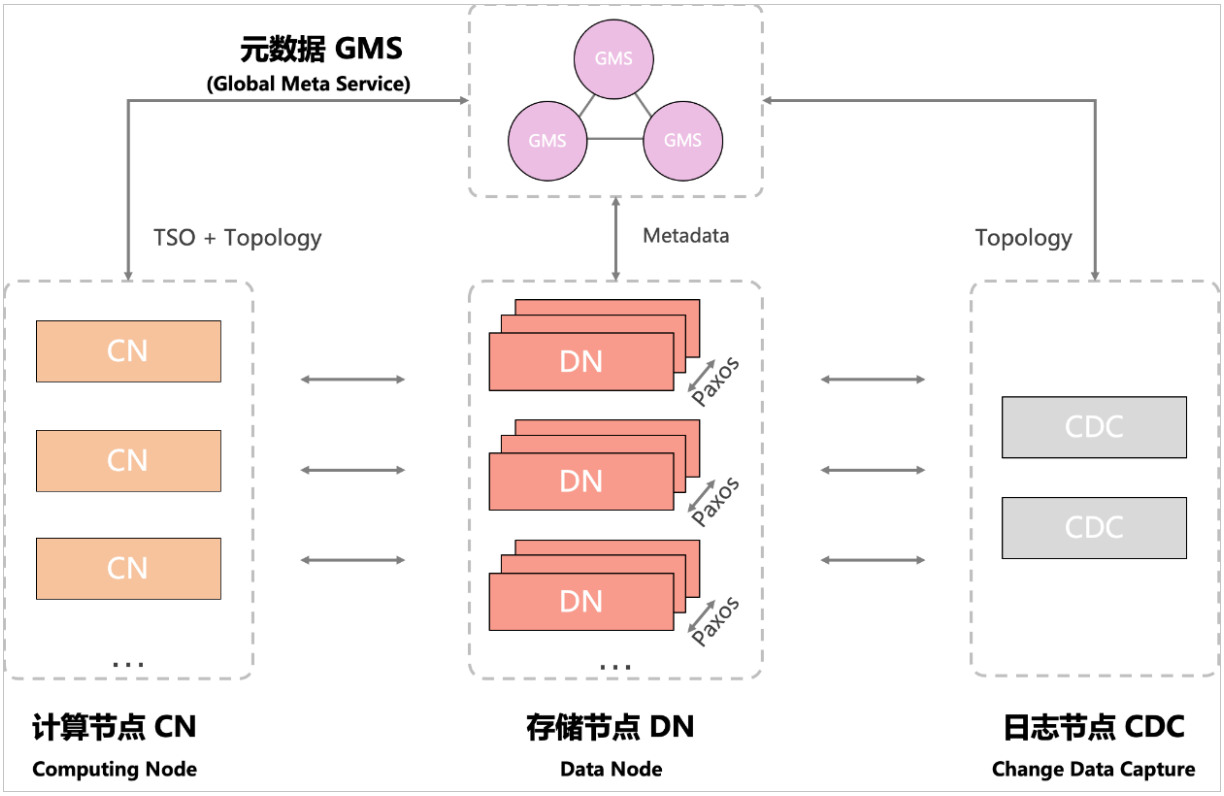
与此同时，DRDS进行品牌升级，命名为PolarDB-X，"PolarDB"是阿里云自研关系型数据库产品家族名称，"X"取音"Extreme"，取意"极致"。

PolarDB-X 1.0支持以PolarDB MySQL作为存储节点，大幅提高集群IO能力以及柔性分布式事务，且面向政企客户需求增强了安全特性，例如，一致性备份恢复、SQL闪回、SQL审计等。

PolarDB-X 2.0阶段

关键字：透明分布式、开源

PolarDB-X 2.0是阿里巴巴分布式数据库有史以来最大幅度的版本更新。产品基于透明分布式理念提供了默认主键拆分策略、基于TSO和MVCC的高性能强一致分布式事务、基于一致性Hash分区策略的分布式线性扩展能力、全局一致性Binlog和全局一致性备份能力。数据节点(DN)采用阿里巴巴自研的基于X-Paxos的三副本强一致MySQL分支，确保在容灾过程中RPO=0。



目前PolarDB-X 2.0已经采用全新架构在阿里云世界范围内的13个国家和地区提供服务。并提供公有云、专有云、软件版和DBStack多种部署形态满足不同行业客户的需求。

PolarDB-X 2.0已全面拥抱开源，毫无保留地和全世界的MySQL社区共同成长，将高性能分布式数据库质量进一步提升，使用门槛尽可能降低。

未来，国产化、HTAP混合负载、快速容量变更、多区域容灾多活等均是PolarDB-X 2.0的探索方向。

2.3. 适用场景

本文介绍了PolarDB-X的适用场景。

高负载低延时交易

场景描述

交易场景广泛存在于互联网业务中，交易系统是信息系统中最为核心的组件之一。业务连续性、事务一致性和系统安全性是交易系统正常运行的基础，长时间高负载低延时的运行是互联网时代交易系统的发展方向。

产品能力

PolarDB-X采用搭载自研多数派共识协议X-Paxos，并应用于存储节点提供三副本强一致能力，确保高可用切换和容灾场景下RPO=0。基于全局时钟TSO策略和分布式的MVCC多版本，分布式事务可确保多节点数据访问的事务一致性。PolarDB-X已通过中国信通院《金融级分布式事务数据库稳定性专项评测》，且持续12年支撑“双十一”全球购物狂欢节，集稳定性与高性能于一身。

大峰谷差流量

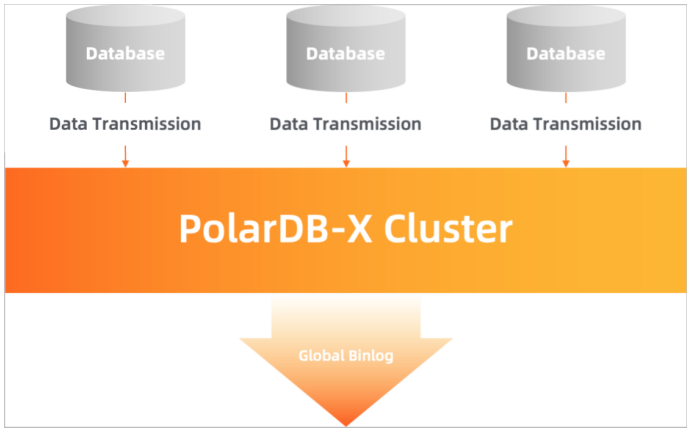
场景描述

大峰谷差是指特定周期内系统峰值负载是谷值负载的20倍以上的系统访问场景，该场景多见于“秒杀”、“拼团”和“限时优惠”相关业务中。系统管理员常陷于容量安全与成本控制之间的两难，所以成本优化方案和平滑扩缩容能力是该场景的核心诉求。

产品能力

PolarDB-X采用分布式线性扩展机制，在扩展过程中保持计算下推和数据一致性的同时实现业务零感知，结合流量控制能力进一步提升扩展过程的业务稳定性。同时PolarDB-X提供历史数据清理和归档能力，使庞大的数据存储成本得到有效控制。

数据集中存储



场景描述

该场景也称“数据大集中”或“数据归集”，属于企业数据架构中的ODS层，具有承担各垂直业务数据源的数据汇总功能。高并发写入、大容量存储、多维度查询、低成本流出是该场景的主要诉求。

产品能力

- PolarDB-X可根据访问量和存储量方便的进行Scale-Up/Down和Scale-Out/In，按需满足高并发写入、大容量存储需求。
- 基于并行计算的DML以及大事务的支持能力，可以有效满足跑批处理和执行效率。
- 高性能全局二级索引（GSI）让用户不局限于拆分规则，可根据任意维度对PolarDB-X进行查询。
- 全局Binlog是保证事务有序性的分布式数据库统一变更日志服务，且兼容MySQL Binlog文件格式和Dump协议，使下游数据消费轻松便捷。

分布式快速改造

场景描述

当业务体量即将突破单机数据库承载极限和单表过大导致性能、维护问题时，分布式改造是解决上述问题的高性价比方案。数据库作为分布式改造的重点难点，"和使用单机数据库一样使用分布式数据库"一直是广大用户的核心诉求。

产品能力

PolarDB-X推出"透明分布式"系列能力，从连接、开发到管理行为均最大限度保留单机MySQL的使用体验，让用户的分布式改造周期大幅缩短，研发运维团队的原有技术栈最大限度保留。目前，PolarDB-X具备从单机到分布式的平滑演进能力，支持动态通过DDL将一张大表动态调整为分布式的分区表，结合分布式事务、以及兼容MySQL binlog的数据回流，可完成单机到分布式的快速改造。

数据库国产化替换

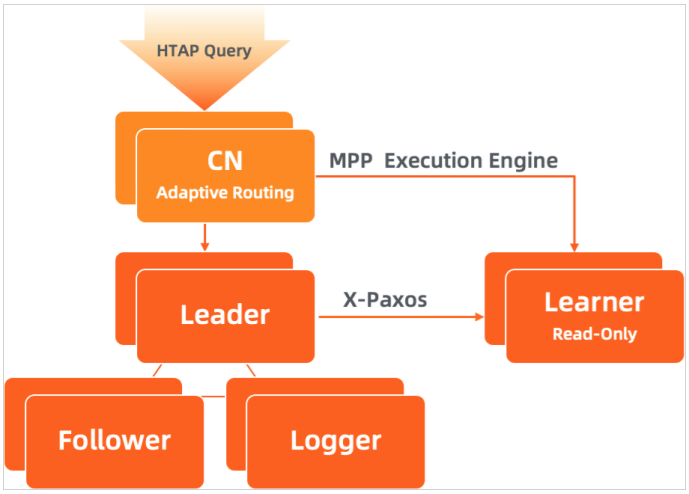
场景描述

信息系统国产化是大型企业实现数字化转型的重要一环，数据库作为核心基础软件首当其冲，目前数据库国产化改造已经在电信、泛金融、能源和企事业单位广泛开展。

产品能力

PolarDB-X是阿里云自研的云原生分布式数据库，具备国产化、自主可控的能力。另外，具有公有云、专有云、DBStack和软件版多种部署形态、完善的交付和服务团队，已经帮助百余家企业完成商业数据库替换、核心数据库系统分布式改造、分布式数据库技术培训与架构咨询等多项任务。

混合负载访问



场景描述

互联网业务的实时化、智能化趋势催生了事务数据与分析数据在相同数据源内进行混合访问（HTAP）的需求。数据一致性、访问便捷度和访问安全性是混合负载访问场景的主要诉求。

产品能力

PolarDB-X基于并行计算和弹性扩展能力，可实现对在线数据做实时报表分析；同时引入智能读写分离、只读副本做分析，保障OLTP和OLAP访问的隔离性的同时，提供统一访问入口，自动选择查询执行引擎。使用一份数据完成两种类型访问，免去ETL的成本和延时。

2.4. 产品计费

PolarDB-X一体化实例由多个逻辑节点构成，部署多个节点进行水平扩展，每个节点闭环整合计算资源与存储资源，运维管理更加便利。本文主要介绍PolarDB-X产品如何计费。

计费方式

PolarDB-X支持如下两种计费方式。

> **说明** PolarDB-X暂不支持按量付费实例转换成包年包月，包年包月实例可以转换成按量付费。

计费方式	说明
包年包月（也称为预付费）	• 在新建实例时支付费用。 • 包年包月实例无法释放。
按量付费（也称为按小时付费、后付费）	• 按小时计费，实时扣费的计费方式。在一个计费周期内，如果您使用PolarDB-X实例时间不足一小时，最少按一小时收费。 • 适合短期需求，用完可立即释放实例节省费用，关于如何释放实例，详情请参见释放实例。

计费项

> **说明** 使用PolarDB-X产生的内网流量免费（如在云服务器ECS和PolarDB-X之间进行的数据传输是免费的）。

计费项	说明	价格
逻辑节点	• PolarDB-X的逻辑节点包括逻辑主节点和只读节点，两者计费标准不同。 • PolarDB-X的逻辑节点支持**包年包月**和**按量付费**两种计费方式。 **说明** • PolarDB-X主实例的所有节点均为主节点，只读实例的所有节点均为只读节点。 • PolarDB-X节点支持升配和降配，操作完成后按照新规格价格进行计费。关于如何进行实例变配，请参见实例变配。	详情请参见逻辑节点价格。
存储空间	存储空间是指数据文件、索引文件、日志文件（在线日志和归档日志）和临时文件等占用的空间。 **说明** 购买PolarDB-X实例后，系统会自动创建数据库正常运行所需的相关文件并占用一定的存储空间，包括上述几种文件。	节点的付费方式不同，存储空间的价格也不同，详细价格请参见下存储空间价格。

逻辑节点价格

- 逻辑节点价格表（主节点）

节点规格	包年包月（元/每月）	按量付费（元/每小时）
4核16GB	2,680.90	5.59
8核32GB	5,742.18	11.97
16核64GB	12,657.78	26.38
32核128GB	25,133.65	52.37

- 逻辑节点价格表（只读节点）

节点规格	包年包月（元/每月）	按量付费（元/每小时）
4核16GB	2,144.72	4.48
8核32GB	4,593.74	9.58
16核64G	10,126.22	21.11
32核128GB	20,106.92	41.90

存储空间价格

存储空间采用Serverless方式，购买时无需选择容量，您只需对实际使用的数据量按量付费。

主节点的存储空间价格

主节点付费方式	存储空间价格
按量付费的主节点	0.0017元/GB/小时

主节点付费方式	存储空间价格
包年包月的主节点	0.001元/GB/小时

只读节点的存储空间价格

只读节点的付费方式	存储空间价格
按量付费的只读节点	0.00136元/GB/小时
包年包月的只读节点	0.001元/GB/小时

2.5. 存储资源包

本文主要介绍了存储资源包的计费方式。

简介

PolarDB-X提供了预付费形式的存储容量资源包，可以抵扣同一区域下按量付费的磁盘计费账单，适用于多种磁盘类型。每个按量付费实例的存储空间都是根据数据量和存储时长单独收费的，若您需要的数据量较大，推荐配合存储资源包一起使用，相比按量付费云盘，存储资源包更加灵活，性价比更高。

在云原生分布式数据库控制台的资源包管理菜单中，您可以购买存储资源包，最低购买规格为500 GB，每次最低增加100 GB，您可以按照实际情况进行选择。

存储资源包在购买后立即生效。生效日期按购买时间的整点计算，从生效日期开始抵扣按量付费计费账单，直至过期，过期时间为到期日的次日零点。假设您在2019-08-20 09:10:00购买了一个10 T的存储资源包，有效期为一年。则该资源包的生效日期为2019-08-20 09:00:00，过期时间为2020-08-21 00:00:00。

注意事项

- 存储资源包仅支持抵扣同一区域中的磁盘账单。
- 仅支持抵扣按量付费账单。
- 暂不支持通过API创建和管理存储资源包。
- 存储资源包暂不支持购买后降低存储资源包的规格。

计费规则

存储资源包支持预付费的计费方式，购买时一次性支付所有费用，存储资源包生效期内不再缴纳费用。一旦您购买了一个区域的存储资源包，在过期之前，均可以自动匹配区域内符合条件的按量付费磁盘并抵扣按量付费账单。

如果您实际使用的磁盘容量超过了存储资源包容量，超出部分仍采用按量付费的计费方式。例如购买了1 TB的存储包，有三个数据库实例，存储容量分别是300 GB、400 GB和500 GB，那么300+400+500-1000=200 GB，存储包抵扣后，超出的200 GB需要以按量付费的方式收取费用。

您可以在费用中心管理控制台查看存储资源包的每小时抵扣记录。

定价

存储资源包费用计算方式：

预付费用=有效期 × 存储资源包容量 × 存储资源包单价

存储容量包单价如下：

单位	中国内地	中国香港及海外
1 GB	0.8/元/月	1.2/元/月

过期

存储资源包超出有效期后，无法继续抵扣磁盘的按量付费账单。如果未续费或者重新购买存储资源包，所有磁盘均采用按量付费计费方式。

退款

退款规则如下：

- 购买存储资源包后，五天内可申请无理由退款。

  > **说明** 每个阿里云账号每年只有一次五天无理由退款的机会，即每个账号每年最多退款一次，可退还的上限是一个存储资源包。

- 其他情况，请提交工单联系阿里云客服处理。

退款时会扣除掉已经消费的金额，已使用的代金券等优惠折扣不退还。详情请参见退款规则及退款流程。

2.6. 术语表

本文介绍云原生分布式关系型数据库PolarDB-X使用过程中遇到的术语。

名词	描述
地域	数据中心所在的地理位置。
可用区	可用区是指在某一地域内，具有独立电力和网络的物理区域。同一可用区内实例之间的网络延时更小。
集群	PolarDB-X采用多节点集群的架构，集群中有多个CN和DN。
规格	集群规格。PolarDB-X每个节点的资源配置，例如8核64GB。
访问点	访问点定义了数据库的访问入口，访问点也称为接入点。每个集群都提供了多个访问点，每个访问点可以连接1个或多个节点。例如，主访问点永远指向主节点，集群访问点提供了读写分离能力，连接了主节点和多个只读节点。访问点中包含的主要是数据库链路属性，例如读写状态、节点列表、负载均衡、一致性级别等。
访问地址	访问地址是访问点在不同网络平面中的载体，一个访问点可能包含私网和公网两种访问地址。访问地址中包含了一些网络属性，例如域（Domain）、IP地址、专有网络 （VPC）、交换机（VSwitch）等。
集群地址	整合集群下的多个节点，对外提供一个统一的读写地址。集群地址具有自动弹性、智能路由、读写分离、负载均衡、一致性协调等能力。
只读地址	整合集群下的多个只读节点，对外提供只读服务。只读服务提供了强一致性和弱一致性两种只读选项。
主实例	对外提供数据库服务的主集群叫主实例，提供的是集群地址出去。
只读实例	只读实例是对主实例资源的一种扩展补充，兼容主实例的SQL查询语法，只读实例与主实例可以共享同一份数据，通过物理资源隔离的方式，缓解主实例的负载压力，降低业务架构的链路复杂度，无需进行额外的数据同步操作，节省运维及预算成本。
工作负载	工作负载分为TP和AP，将事务内操作、写操作、简单查询操作归纳为TP负载，复杂的查询操作归纳为AP负载。

智能路由	当用户在集群地址上开启了智能路由，会基于统计信息识别SQL的工作负载类型来做路由，比如将识别为AP负载的流量路由给只读实例。
读写分离	当用户在集群地址上开启了读写分离，会根据SQL属性做直接路由，比如将事务内和写操作都路由给主实例，查询操作统一路由给只读实例。
数据查询一致性	只读地址对外提供的查询一致性类型，分为强一致性和弱一致性。强一致性：可以查看到主实例上提交的最新数据，满足全局数据查询一致性。如果主备延迟过大，提交的查询会等待；弱一致性：可以查看到当前只读实例上最新数据，如果在主实例和只读实例间数据同步有延迟的话，不会等待，查询到数据立即返回。
三权分立	可以将高权限账号拥有的权限分给系统管理员、安全管理员和审计管理员这3个角色，避免因权限高度集中带来的风险，增强数据库的安全性。三权分立主要运用于金融业务场景中。
白名单	为PolarDB-X实例提供访问安全保护，设置白名单并不会影响PolarDB-X实例的正常运行。
审计日志	记录用户操作记录，SQL审计日志默认保存45天
HTAP（Hybrid Transactional and Analytical Processing）	即同时具备在线事务处理（OLTP）与在线分析处理（OLAP）的能力。
计算节点（Compute Node，简称CN）	PolarDB-X集群是存储计算分离架构，计算层由多个计算节点构成，每一个节点关系对等，规格相同。 计算节点包括 SQL解析器、优化器、执行器等模块。
存储节点（Data Node，简称DN）	PolarDB-X集群是存储计算分离架构，存储层由多个存储节点构成，每一个节点关系对等，规格相同。 存储节点负责数据的持久化，基于多数派 Paxos 协议提供数据高可靠、强一致保障。
日志节点（Change Data Capture，简称CDC）	日志节点提供完全兼容MySQL Binlog格式和协议的增量订阅能力，提供兼容MySQL Replication协议的主从复制能力。又称为增量数据捕捉。
元数据服务（Global Meta Service，简称GMS）	元数据服务负责维护全局强一致的Table/Schema、Statistic等系统Meta信息、维护账号、权限等安全信息，同时提供全局授时服务（Timestamp Oracle，简称TSO）。
CTS（Commit Timestamp）扩展	全局提交时间戳功能的扩展，可解决分布式事务模式下如果使用快照读可见性问题。
ACID	ACID指数据库管理系统在写入或更新资料的过程中，为保证事务是正确可靠的，所必须具备的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）以及持久性（Durability）。
分库分表	按照一定的拆分规则根据拆分键将一张表中的数据拆分到多个数据库与数据表中。
广播表	广播表不做拆分，且所在数据库的所有数据节点均具有该表的副本。
单表	没有做拆分的表为单表。

分区表	带有分区的表，每个分区的数据单独存储。
表组（Table Group）	为了尽可能避免跨机器查询，提高性能，可将某些表组成表组，表组中的所有表必须具有同样的Partition数目、相同的分区算法和同样的分区键。
分区组（Partition Group）	当表组中的表为分区表时，该表组的所有表的某个分区构成一个分区组，分区组是分区调度的基本单元，属于某个分区组的所有表分区始终会位于同一个存储节点。
Join Group	Join Group由多个表构成，在同一个Join Group中的表，Join操作会尝试进行下推优化，如果成功下推，则执行效率会更高；不在同一个Join Group中的表，Join操作不会尝试进行下推优化。
逻辑查询	客户端发送至PolarDB-X的SQL查询。
物理查询	在数据节点执行的SQL查询。
逻辑连接	客户端到PolarDB-X计算节点的连接。
物理连接	PolarDB-X计算节点到PolarDB-X数据节点的连接。
分布式执行计划	在分布式数据库中，对SQL查询语句进行优化后的执行方案。该执行计划可反映逻辑SQL在各节点的执行情况。
代价模型	用于估算物理查询的执行计划的代价，PolarDB-X的执行代价用（CPU、Memory、IO、Net）四元组来描述。
执行模型	与传统数据库采用Volcano执行模型不同，PolarDB-X采用的是Pull与Push结合的混合执行模型。
CBO（Cost Based Optimizer）	基于代价的优化器
RBO（Rule Based Optimizer）	基于规则的优化器
算子	执行计划由一系列算子构成，算子是基本的执行计划单元。
调度	将某个作业或者部分作业放到另外一台机器上执行。
Online DDL	执行DDL操作（例如创建索引）的同时不阻塞并发的DML操作。
分布式事务	同一事务内的操作涉及多个数据节点。
全局时间戳	集群内全局唯一且递增的时间戳。
局部索引	存储节点内，由MySQL维护的索引。又称二级索引。

全局二级索引（Global Secondary Index，简称GSI）	全局的二级索引，索引中的数据按照指定的拆分方式分布在各个存储节点上。
聚簇索引	一种特殊的全局二级索引，默认覆盖列包含主表的所有列，所有查询均不用回表，从而避免回表带来的额外开销。
主键拆分	以主键为拆分维度的横向拆分方式。
水平扩缩容	增加或者减少节点数量，例如从四个节点增加至八个节点。
垂直升降配	升高或者降低单个节点的配置，例如节点规格从4C8G升高至16C32G。
XPaxos	阿里巴巴集团设计的分布式一致性协议。
Leader/Follower/Learner	XPaxos的节点类型，Leader负责发起提案；Follower具有投票权；Learner仅能获取变更，无权参与投票。

3.核心特性

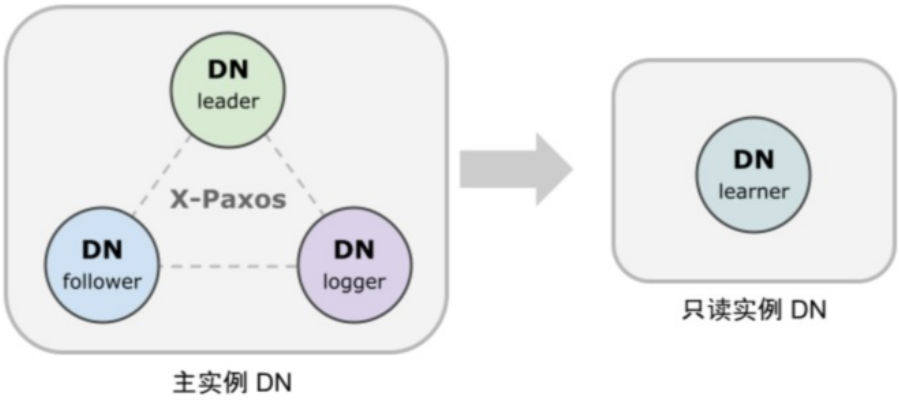
3.1. 高可用性与容灾

本文详细介绍了高可用与容灾特性。

在生产环境部署数据库时，往往会搭建多个副本（Replica），保证数据库集群的高可用性以及数据的持久性。传统的部署方式是一主一备，即主备间通过日志同步数据变更。但是主备复制存在先天性缺陷，以常见的MySQL半同步复制为例，一旦网络延迟超出阈值，同步就会退化到异步复制。此时如果主节点宕机，副本可能丢失已提交的数据，也就是常说的副本不一致。

为了保证副本间的强一致性，现代数据库往往采用以Paxos为代表的多数派复制协议。Paxos通常要求集群中至少存在3个节点，每次写入都要获得超过半数节点的确认，即便其中1个节点宕机，集群也仍然能正常提供服务。Paxos算法能够保证副本间的强一致性，彻底解决副本不一致问题。

PolarDB-X在副本复制方面采用了X-Paxos。X-Paxos是阿里巴巴自研的Paxos协议实现，起源于AliSQL（阿里内部的MySQL分支）。基于朴素的Paxos实现，它在功能、性能上都做了大量优化，且经历了数十载的双十一考验，稳定可靠。



X-Paxos实现了Multi-Paxos算法，通常有一个相对稳定的Leader节点用于处理读写请求。如果Leader节点因为某些意外情况发生宕机或超时，Follower节点就会重新发起选主投票，如果得到超过半数的选票则成为新的Leader节点。Logger节点只负责保存日志以及参与Paxos投票。Learner节点通常在只读实例中，它仅仅接受主机群的变更日志，不参与Paxos投票。

此外，X-Paxos还支持动态添加删除节点、权重化选主、Leader主动回切等企业级特性，允许用户根据业务需求灵活定义部署方式。

多机房部署

基于Paxos复制协议，PolarDB-X可以部署到多个机房中，以实现机房级容灾。常见的部署方式有同城三机房、两地三中心等，其中后者主要应用在混合云部署中。由于Paxos协议的特性，通常三个机房中有一个主机房负责对外提供服务。

跨地域多活

如果业务有多活需求，出于性能考虑，通常建议在不同地域分别部署PolarDB-X实例，彼此之间通过DTS等数据同步工具相连接。数据要根据实际业务需求做合理的分区，以保证各个地域的数据不会相互冲突。PolarDB-X全局变更日志特性支持实例到实例的数据同步。

3.2. 强一致分布式事务

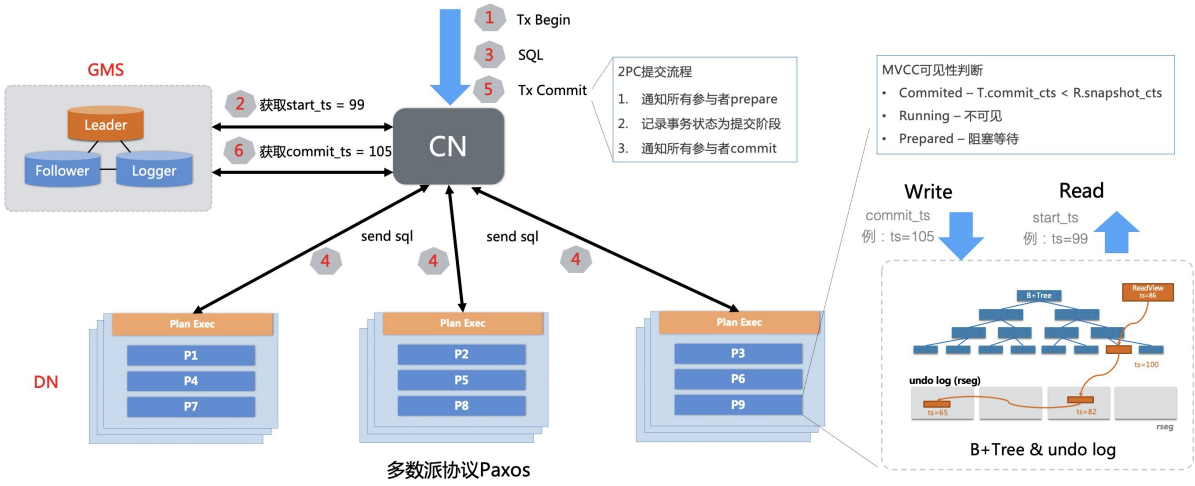
本文详细介绍了什么是强一致分布式事务及如何实现。

ACID分布式事务

PolarDB-X原生支持分布式事务，并保证事务的ACID性质。

- 原子性（Atomicity）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

PolarDB-X通过引入中心授时服务（TSO），结合多版本并发控制（MVCC），确保读取到一致的快照，而不会读到事务的中间状态。如下图所示，提交事务时，计算节点（CN）执行事务时从TSO获取到时间戳，随着数据一同提交到存储节点（DN）多版本存储引擎上，CN通过读取快照时间戳去DN上读取相应版本的数据。



TSO中心授时

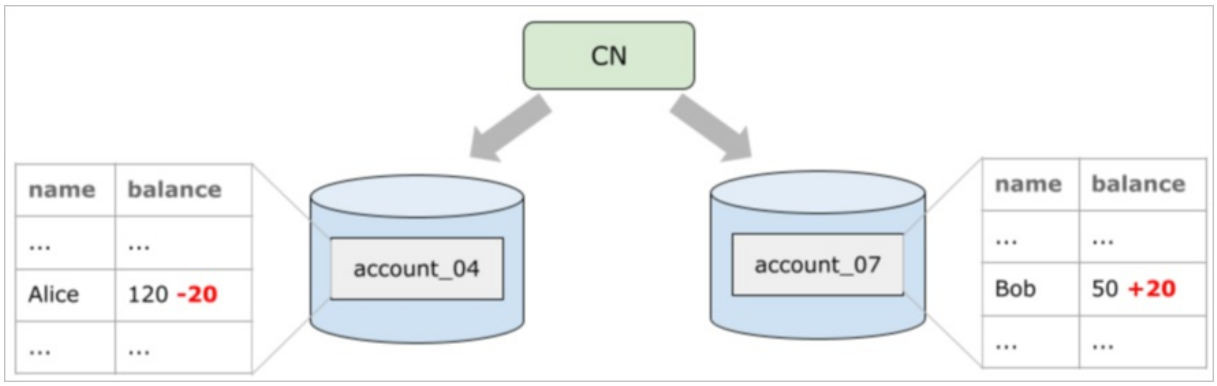
PolarDB-X分布式事务支持MVCC多版本，分布式的版本号依赖于中心授时服务来生成全局单调递增的时间戳，基于中心授时为版本提供事务的一致性读取。

PolarDB-X元数据服务（GMS）基于Paxos共识协议提供一个具备三节点高可用性的服务，PolarDB-X的计算节点（CN）会通过RPC接口和元数据服务（GMS）进行通讯并获取中心授时。

事务2PC提交

以转账场景为例，假设用户账户是一张分区表，那么同一笔转账交易的转入方和转出方很可能位于不同的数据节点上，因此需要分布式事务来保证转账结果正确。

```
BEGIN;
UPDATE account SET balance = balance - 20 WHERE name = 'Alice';
UPDATE account SET balance = balance + 20 WHERE name = 'Bob';
COMMIT;
```



如果事务内写入的数据涉及多个分区，PolarDB-X的计算节点将会使用两阶段提交（Two-phase Commit Protocol，简称2PC）方式提交事务，即便在事务提交过程中发生节点宕机等问题，基于2PC的事务恢复机制也能确保事务原子性。

MVCC多版本

以上面的转账场景为例，如何在转账的同时查询所有账户的余额总额（“对账”）：

```
SELECT SUM(balance) FROM account;
```

因查询操作的数据涉及多个分区，PolarDB-X首先会获取中心授时确定读取版本，读取过程中会对每行数据的MVCC多版本进行可见性判断，确保只会读取在全局时间戳之前已完成提交的事务。

例如转账事务在多个数据节点的提交有先后时间差，已提交的分支事务因为数据版本号不满足可见性，正在提交的事务数据全部不可见，从而确保总额数据读取的一致性。

分布式事务的下游生态

读写分离的一致性

事务型的分布式数据库一般会采用读写分离的模式提升读的性能，因此分布式事务除了保障主库的一致性以外，还需要保证用户在使用读写分离模式下对于备库的一致性。

PolarDB-X产品支持主实例和只读实例，主实例是由基于Paxos多数派协议的Leader/Follower角色组成，只读实例是由基于Paxos协议的Learner角色组成。PolarDB-X针对分布式事务一致性的设计，除了在存储节点（DN）的leader主副本中保存事务信息之外，也会将数据的事务多版本信息同步到learner副本中，可以保证只读实例上的多个分区数据的读的一致性，具体特性请参见混合负载HTAP。

例如，主库中写入了一个版本为100的数据，而下一次的查询中带着中心授时返回的版本为101。当查询被路由到只读的learner节点时，即使learner节点有数据复制延迟，PolarDB-X也可以阻塞读来满足读的一致性。

全局变更日志CDC

事务型的分布式数据库除了面向在线业务的高并发写入以外，一般还需要将在线数据同步给下游的灾备数据、汇聚业务或数据仓库系统等，下游业务对于事务日志的一致性也有比较强的诉求，例如事务不能乱序、事务原子性、DDL支持同步等。

MySQL binlog是MySQL数据库中记录变更数据的**二进制日志**，它可以看做是一个消息队列，队列中按顺序保存了MySQL中详细的增量变更信息，通过消费队列中的变更条目，下游系统或工具实现了与MySQL的实时数据同步，这样的机制也称为CDC（Change Data Capture，增量数据捕捉）。

PolarDB-X存储节点（DN）在变更日志设计上也会保存分布式事务信息，通过PolarDB-X日志组件（CDC）收集分布式下多个存储节点（DN）的日志流，进行汇总、重组、排序、落盘等过程，提供满足分布式事务一致性语义的**二进制日志**，并全面兼容MySQL binlog协议和生态。具体特性请参见全局日志变更。

备份恢复的一致性

传统关系型数据库一般会采用数据库的全量备份来保证数据安全。在遇到数据异常、数据库被删的风险时，需要通过数据库的备份集进行快速恢复，其中数据恢复也要保证事务的一致性。另外，分布式数据库通常数据存储规模更大，对于备份恢复的一致性有更大的挑战。

PolarDB-X在存储节点（DN）的数据和变更日志中都保存了分布式事务的中心授时（包含了时间戳信息），任意时间点的数据恢复（PITR，point-in-time recovery）都可以快速将时间戳转化为分布式的中心授时，在备份恢复中按数据的版本可见性进行处理，同时PolarDB-X结合分布式的多节点并行来全面提升备份恢复的效率。具体特性请参见数据备份与恢复。

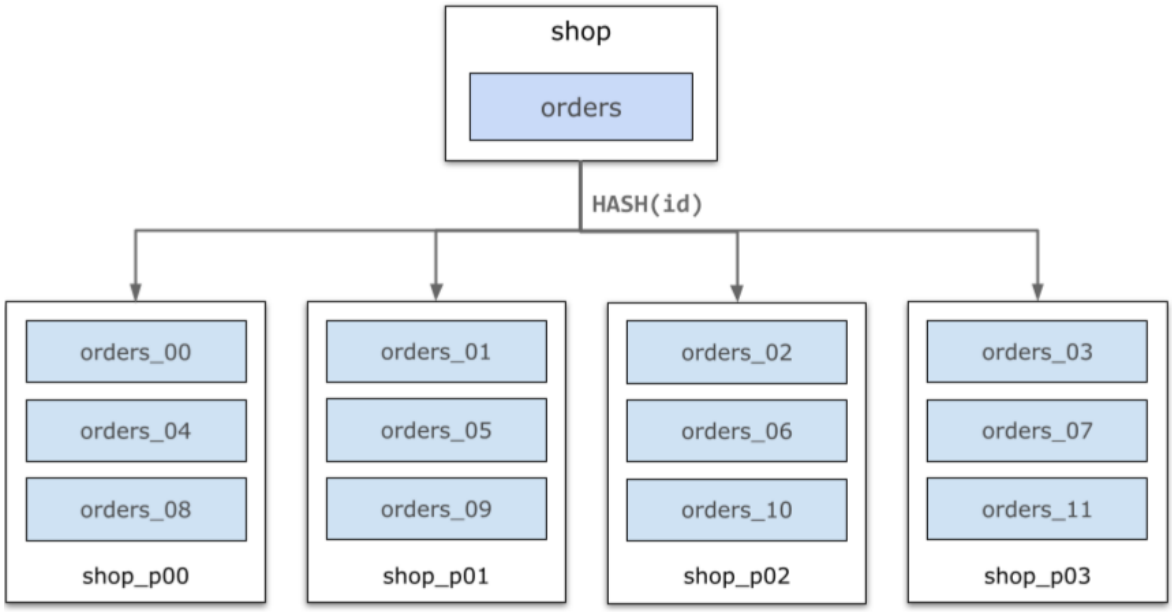
3.3. 分布式线性扩展

本文详细介绍了分布式线性扩展的特性。

数据物理分布

PolarDB-X将数据表以水平分区的方式，分布在多个存储节点（DN）上。数据分区方式由分区函数决定，PolarDB-X支持哈希（Hash）、范围（Range）等常用的分区函数。

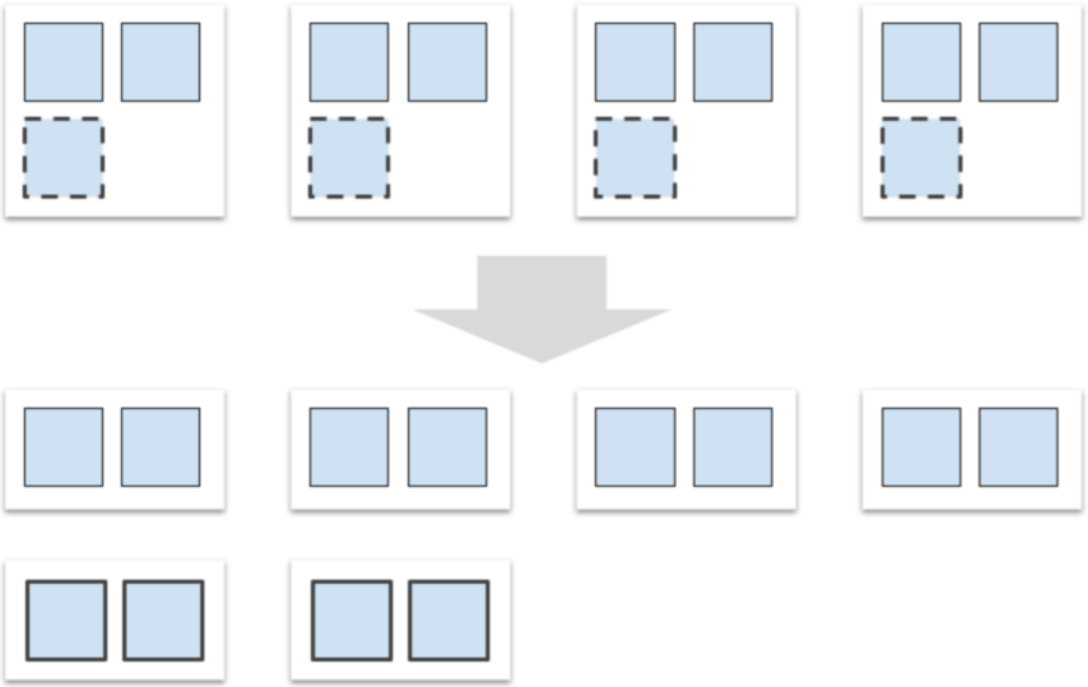
以下图为例，shop库中的orders表根据每行数据的ID属性进行哈希，被分区水平切分成orders_00～orders_11共计12个分区，均匀分布在4个数据节点上。PolarDB-X的分布式SQL层将会自动完成查询路由、结果合并等。



扩容迁移

随着业务的增长，数据量越来越大，往往需要添加更多的存储节点以承载更多的数据。当新的存储节点加入集群时，PolarDB-X将自动触发扩容任务，将数据进行再平衡（Rebalance）。

以下图为例，orders表原本分布在4个数据节点上。进行扩容后，集群的数据节点数量从4个增加到6个，触发再平衡任务，将部分数据分区从旧节点移动到新节点上。这一过程在后台利用空闲资源完成，对业务线上流量无影响。



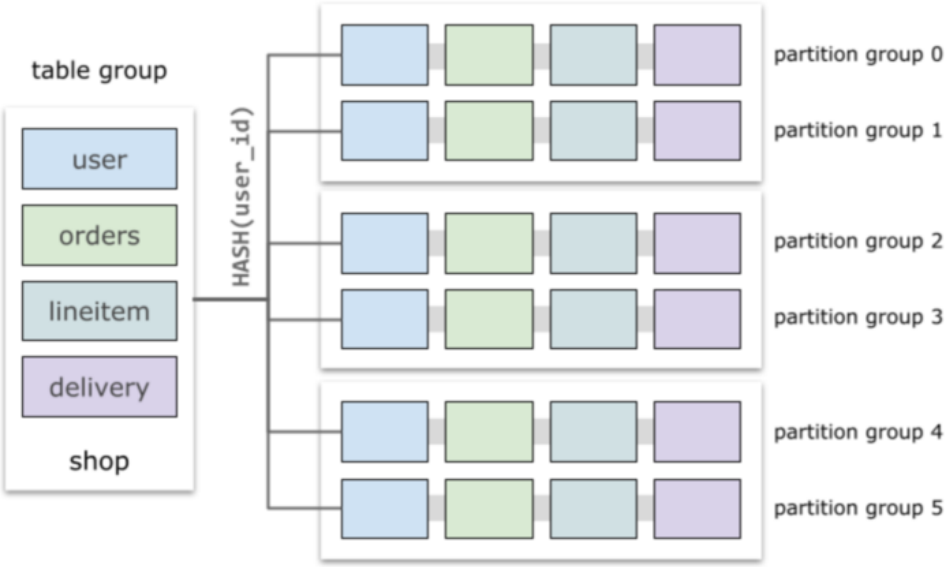
表组

为了提升Join查询性能，PolarDB-X推荐根据业务场景设计各个表的分区键，使得有关联关系的表具有相同的分区方式。

例如下图中，user、orders、lineitem、delivery这四张表都以HASH（user_id）作为分区函数，如下的示意SQL语句可以下推到存储节点上执行，性能更好。

```
SELECT * FROM user, orders WHERE user.id = orders.user_id WHERE user.id = ...
```

PolarDB-X引入了表组（Table Group）概念来描述分区方式相同的表。通常情况下，分区函数以及列类型相同的表会被隐式地自动归入同一表组中。表组内的分区组（Partition Group）包含了这些表对应的分区。

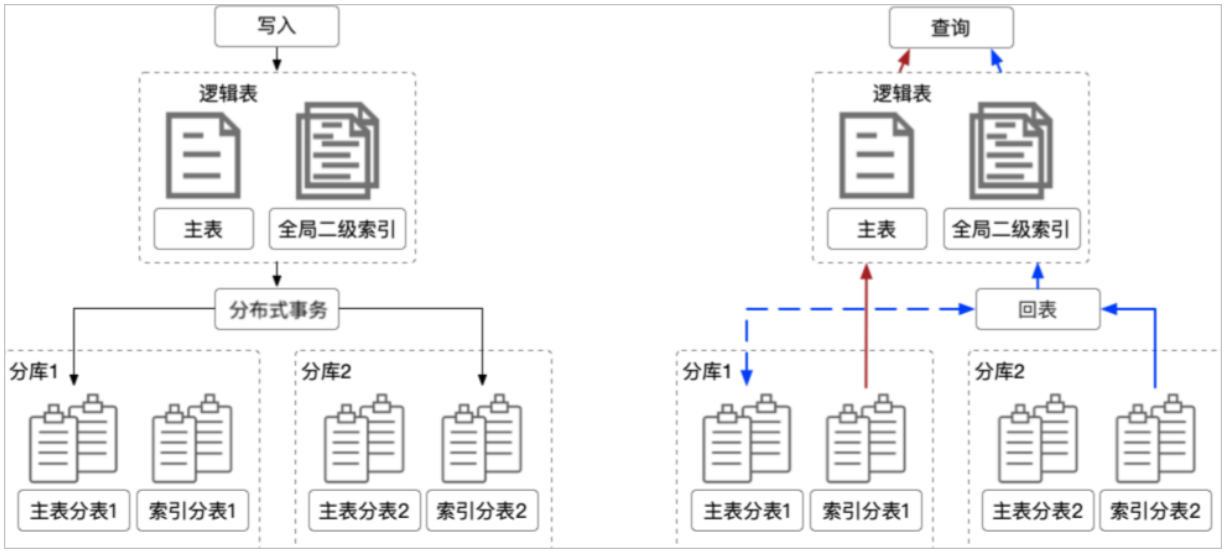


表组的概念对于分区的迁移、分裂与合并至关重要。在上面的例子中，如果对user表的某个分区进行迁移而不迁移orders表的对应分区，那么上文中的查询将无法下推到存储节点，导致性能大幅下降。但有了表组的概念之后，由于user表和orders表位于同一表组中，它们的分区迁移、分裂与合并都会同步进行，解决了上述的问题。

3.4. 全局二级索引

全局二级索引（Global Secondary Index，GSI）是PolarDB-X中的一项重要特性，相比于本地二级索引，全局二级索引中的数据按照指定的拆分方式分布在各个存储节点上。通过全局二级索引，用户能够按需增加拆分维度、提供全局唯一约束等。

每个GSI对应一张分布式索引表，和其他分布式表一样，按照指定的分区规则水平拆分为多张物理表。PolarDB-X使用分布式事务维护主表和索引表之间数据强一致。



全局二级索引还支持以下特性：

- 支持选择覆盖列，减少回表操作开销。
- 在线表结构变更，添加GSI不锁主表。
- 支持通过HINT指定索引，自动判断是否需要回表。

示例1：增加拆分维度。例如，对于在线商城的订单表，假设按照买家用户维度拆分，那么对于卖家查询（例如，查询某个卖家的本月所有订单）就需要扫描所有分区。但是借助全局二级索引，可以仅仅扫描相应卖家所在的索引表分区，便可快速找到所需的订单信息。

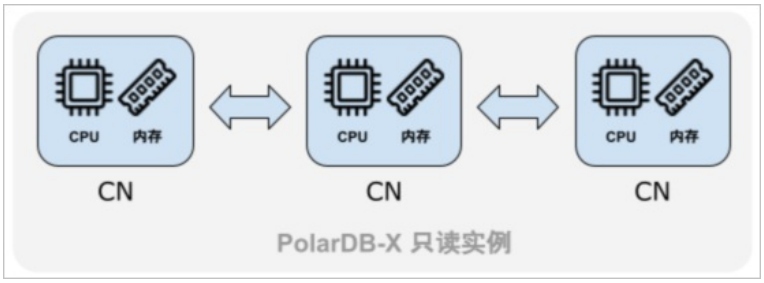
示例2：全局唯一约束。例如，假设用户表是一张分布式表，按照用户ID分区。如果要求用户手机号需要全局唯一，那么本地索引无法满足，必须构建一个以手机号作为索引键（同时也是分区键）的唯一索引。

3.5. 混合负载HTAP

PolarDB-X是一款支持HTAP（Hybrid Transaction/Analytical Processing）的数据库，在支持高并发、事务性请求的同时，也对分析型的复杂查询提供了良好的支持。

> **说明** 分析型查询指的是涉及数据量较大、计算比较复杂的查询，例如对一定时间区间内的数据进行聚合，相比于业务中常见的简单查询，这类查询往往要执行数秒甚至数分钟，需要消耗较多的计算资源。

为了提高复杂分析型查询的速度，PolarDB-X将计算任务切分并调度到多个计算节点上，从而利用多个节点的计算能力，加速查询的执行。这种方式也称为MPP并行计算（Massively Parallel Processing，简称MPP）。目前只有PolarDB-X只读实例默认开启了MPP并行计算能力。



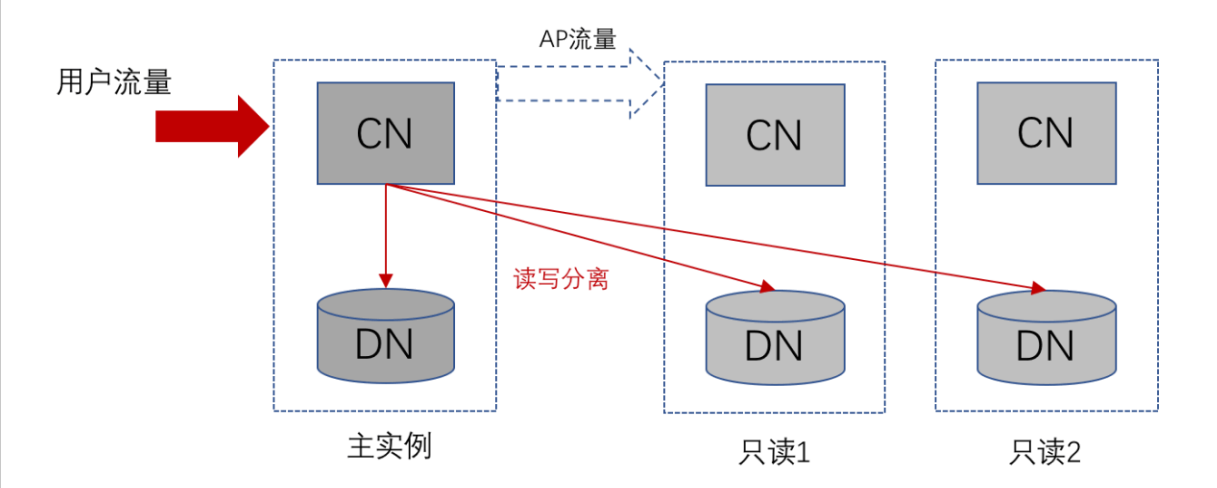
查询优化器

PolarDB-X的优化器面向HTAP负载设计，对复杂查询有着良好的支持。TP（Transaction Processing）类事务型查询包含的表数量通常有限（例如3个以内），并且Join条件往往被索引覆盖，且查询涉及的数据量较小。而对于不符合上述特征的复杂查询，对优化器提出了更高的要求。

PolarDB-X采用了基于代价的优化器技术，能够根据实际数据量、数据分布情况等，搜索到较优的执行计划，例如，对Join顺序进行调整、选择合适的Join或聚合算法，对关联子查询去关联化等。执行计划的好坏很大程度上决定了查询效率，查询优化对于AP（Analytical Processing）类分析型查询至关重要。

读写分离

当在线业务流量比较多时，对PolarDB-X主实例压力比较大。此种场景下，PolarDB-X建议您购买只读实例，按照预先设置好的比例将一部分TP类读查询通过主实例CN转发给只读DN，这个过程称为读写分离。通过读写分离对读流量进行分流，可以减轻主实例存储层DN的读压力。



设置读写分离后，读请求可以配置为强一致性读和弱一致性读。

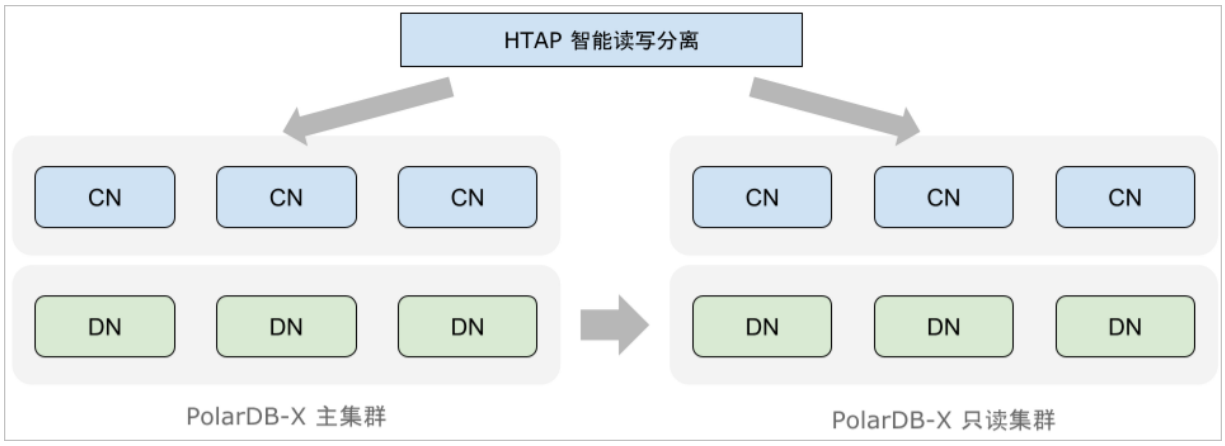
- 强一致性读：路由到只读实例的请求，保证一定能够查询到读请求执行前在主实例上已完成更新的数据，提供外部强一致性；
- 弱一致性读：路由到只读实例的请求，读请求仅访问只读实例上当前的最新数据，会因为主从异步复制的架构产生数据读取延迟。

您可以通过Hint指定哪些读SQL在主实例上执行，也可以预先设置读写比例，详情请参见集群地址和只读地址。

智能读写分离

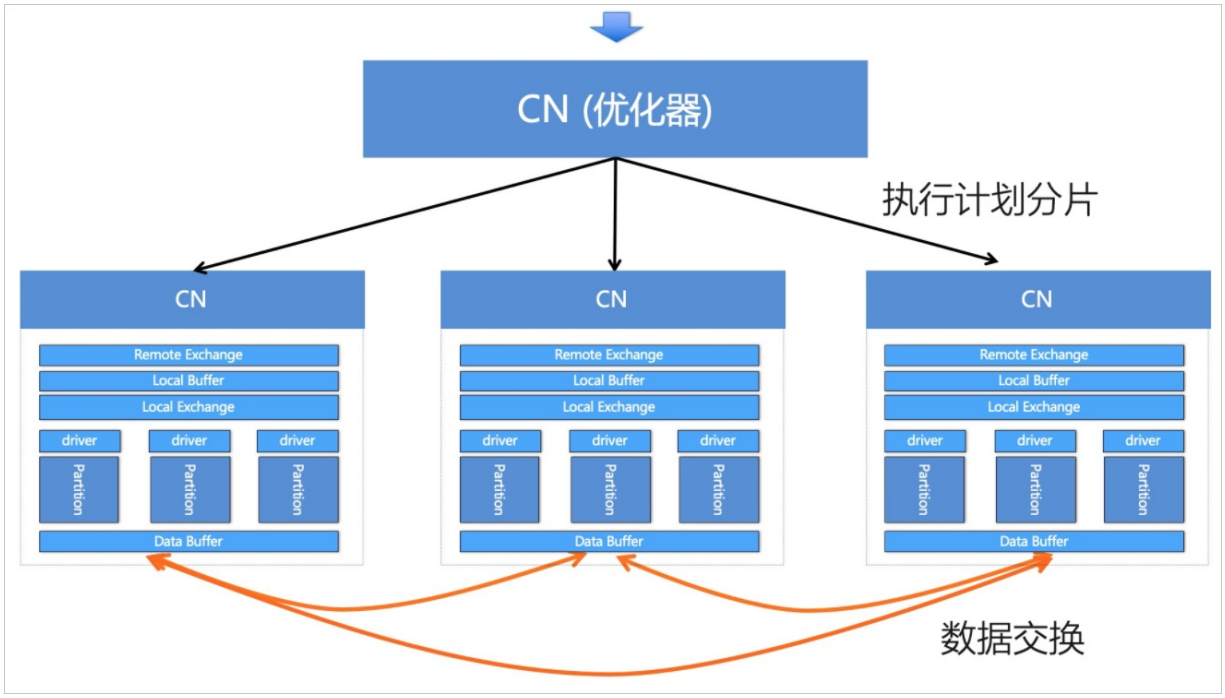
现阶段，HTAP数据库实际应用的一大障碍是AP类查询对TP类查询的影响。为了解决这一问题，PolarDB-X建议您部署独立的只读实例，只读实例与原实例在硬件资源上完全分离，从而将AP类查询对TP类查询的影响降到最低。

PolarDB-X优化器支持基于代价将请求区分为TP与AP负载，其中AP查询会被进一步改写为分布式执行计划，发往只读实例进行计算，避免它对主实例的TP类查询造成影响，这个过程称为智能读写分离。



分布式执行

分布式执行计划下一步会被切分为多个阶段（Stage），每个阶段又会生成多个并行的分片（Split），这些分片被下发到多个计算节点执行。计算节点之间通过高速网络互联。在计算过程中往往需要对中间结果进行多次交换（Exchange），直到最后一个Stage将结果收集汇总、回复给发起查询的客户端。



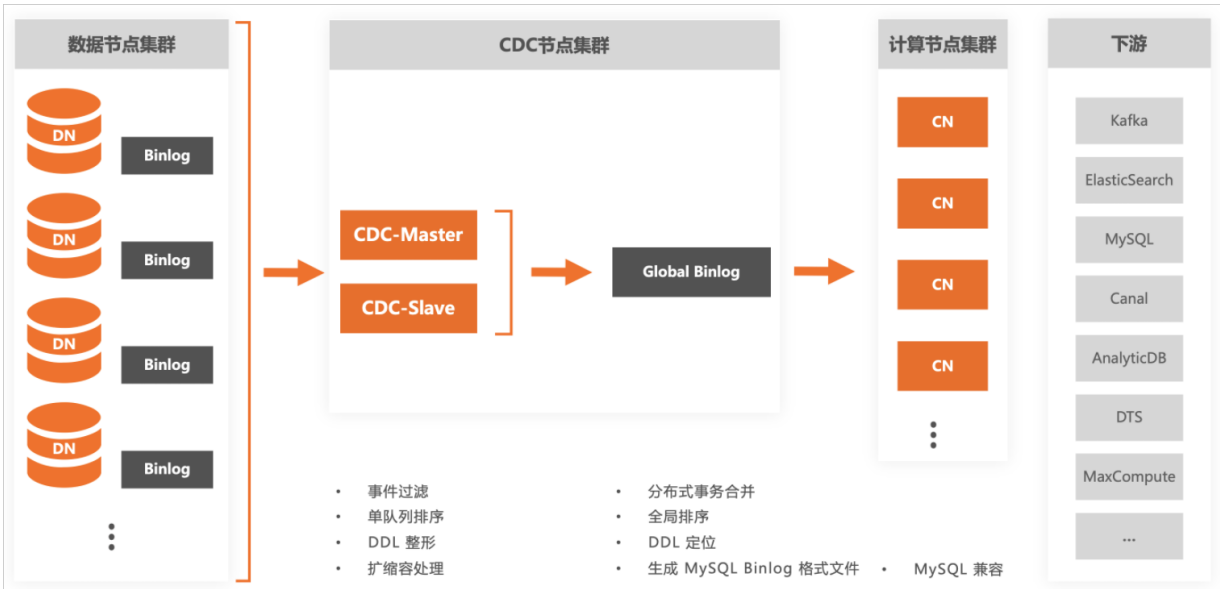
全局一致性读

传统读写分离架构下，数据复制的延迟可能带来的数据写后读（read-after-write）不一致问题。PolarDB-X中，对于路由给只读实例的查询，默认开启全局一致性读能力，确保业务不会读到过期的数据，向主实例写入成功后能在只读库读到写入的数据。

3.6. 全局日志变更

MySQL binlog是MySQL记录变更数据的二进制日志，它可以看做是一个消息队列，队列中按顺序保存了MySQL中详细的增量变更信息，通过消费队列中的变更条目，下游系统或工具实现了与MySQL的实时数据同步，这样的机制也称为CDC（Change Data Capture，增量数据捕捉）。

PolarDB-X是兼容MySQL生态的分布式数据库。通过实例内PolarDB-X的CDC组件，能够提供与MySQL binlog格式兼容的变更日志，并且对外隐藏掉实例扩缩容、分布式事务、全局索引等分布式特性，让您获得与单机MySQL数据库一致的使用体验。



异地多活

除了通过CDC将数据导入其他外部系统中，PolarDB-X的CDC也可以用于实现异地多活的业务部署。例如，将用户ID按照所在地区划分到不同机房，写入操作必须在特定机房进行，而读取操作可以读就近的“副本”，这些副本数据就可以通过CDC从原机房同步得到。

3.7. 数据备份与恢复

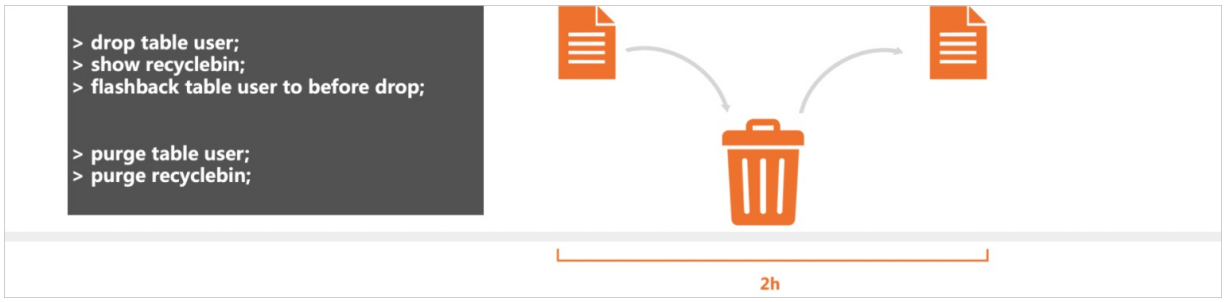
数据备份和恢复是数据库必不可少的能力，PolarDB-X提供不同粒度的数据恢复能力，包括实例级的一致性备份恢复能力、表级的表回收站能力、SQL级的SQL闪回能力等。

一致性备份恢复

一致性备份恢复提供实例级任意时间点（精确到秒级）的历史数据恢复能力。PolarDB-X的一致性备份恢复特性基于定期的全量备份以及全局binlog实现，关于全局binlog的更多信息参见全局日志变更。

以上图为例，假设PolarDB-X实例配置为每周一、周二和周五的零点进行备份。某天，需要将数据恢复到上周日的 14:25:26，那么系统会选择离恢复点最近的一个全量备份集，上周五零点的全量备份，并从周五零点开始重放全局binlog，直到周日14:25:26结束，从而获得所需的一致性快照。

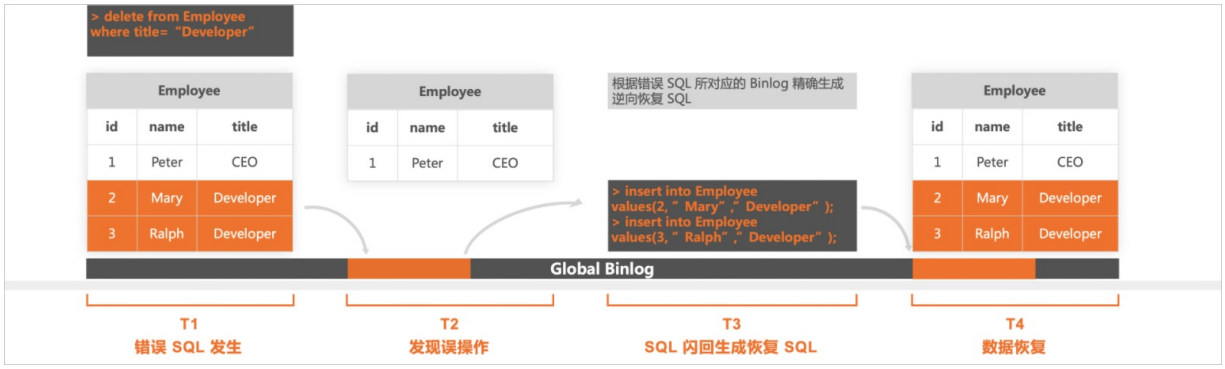
表回收站



当表回收站功能开启时，被删除或清空的表会被暂时放入回收站，若2小时内需要恢复表，可以在控制台的表回收站中找回。

SQL闪回

SQL闪回功能能够在误操作发生后，帮助您回滚或部分回滚之前的操作。例如，当 `DELETE` 语句误删除数据时，可以通过SQL闪回功能为它生成逆向恢复SQL（即相应的 `INSERT` 语句），从而恢复丢失的数据。



3.8. 全链路监控和审计

PolarDB-X具有丰富的监控和审计功能，帮助您实时监测数据库的各项性能指标和运行状态。

监控与报警

您可以在PolarDB-X控制台上查看计算资源监控、存储资源监控和数据库监控等监控信息，或依据这些监控指标添加报警规则。

- 计算资源监控展示了计算节点的监控指标，包括CPU使用率、内存使用量、网络流量（流入流出）、QPS、查询延迟、前端连接数、活跃线程数等。
- 存储资源监控展示了存储节点的监控指标，包括CPU使用率、内存使用量、磁盘空间用量、网络流量（流入流出）、连接数、活跃线程数、IOPS、QPS、TPS、缓冲池用量、临时表创建数量等。

SQL审计与分析

PolarDB-X联合SLS日志服务推出SQL审计与分析功能，将审计日志投递到日志服务中，实现了日志的实时查询、可视化分析、告警、投递、加工等操作。您仅需在控制台上打开功能开关，即可实时进行SQL日志的审计与分析；也可以基于某些指标定制实时的监测与告警，当关键业务出现异常时可通过多种途径进行告警通知。

SQL审计与分析功能默认关闭，您可以在PolarDB-X控制台上随时开启。审计日志默认保存45天。审计与分析功能本身免费，但日志服务会对存储空间、读取流量、请求数量、数据加工、数据投递等进行收费。

实例会话

PolarDB-X控制台上可以查看当前实例会话，也可以选择终止会话。会话统计页面可查看按客户端统计、按namespace统计等的聚合统计信息。

10秒SQL分析功能可以在10秒内对会话进行采样（每秒采样一次），然后对采样结果进行汇总分析，通过分析结果可以很容易看出执行次数最多的查询是否存在慢SQL。

慢日志

PolarDB-X提供慢日志查询功能，并且将CPU利用率和慢日志数量结合在一起，能帮助您快速发现影响数据库性能的关键SQL。慢日志功能提供交互页面，您可以查看慢SQL模板、慢SQL明细，以及每条慢SQL执行次数、耗时、返回行数等关键信息。

慢SQL诊断优化还能根据SQL执行计划智能地给出优化方案，例如，推荐创建本地索引或全局索引。

SQL限流

为防止少数SQL占用过多资源影响业务整体运行，PolarDB-X提供了SQL限流功能，用于限制特定查询的并发度。您可以通过控制台创建限流匹配规则，限制其最大并发度、最长等待队列长度。超过最大并发度的查询将进入等待队列，如果等待队列超过阈值，将会直接报错。

SQL自动限流能够自动识别出慢查询，并为其创建限流规则。该功能需要您手动开启。

3.9. MySQL生态兼容

PolarDB-X将兼容MySQL以及周边生态作为核心设计目标之一。本文从SQL语法、事务行为、导入导出等维度总结了兼容性的相关特性。

MySQL协议

PolarDB-X通讯协议兼容MySQL协议，可以使用常见的MySQL客户端直接连接到PolarDB-X实例，包括JDBC Driver、ODBC Driver、Golang Driver等，并且兼容MySQL SSL、Prepare、Load等传输协议。

SQL兼容性

PolarDB-X兼容MySQL的各种DML、DAL、DDL语法，其中包括：

- 兼容绝大部分MySQL函数（包括JSON函数、加密解密函数等）。
- 兼容MySQL 8.0的视图、CTE、窗口函数、分析函数等。
- 支持MySQL的各种数据类型，包括类型精度的支持（比如时间戳、Decimal类型）。
- 兼容常见的MySQL字符串Charset及Collation。
- 兼容绝大部分information_schema视图。

更多信息请参见开发指南。

ACID事务

PolarDB-X事务采用基于乐观读（ReadView）、悲观锁的设计，支持ANSI标准中的四种隔离级别，且行为和MySQL一致。与MySQL相同，PolarDB-X默认采用可重复读（Repeatable Read）隔离级别，该级别下更新范围条件会引入间隙锁（Gap Lock）。

PolarDB-X基于回滚段的MVCC机制对大事务比较友好，可支持小时级长时间事务、GB级写入量大事务。

账号与权限

PolarDB-X账号权限系统的用法跟MySQL一致，支持GRANT、REVOKE、SHOW GRANTS、CREATE USER、DROP USER、SET PASSWORD等语句。目前支持库级和表级权限的授予，暂不支持列级别权限。更多信息请参见账号与权限。

PolarDB-X还支持基于角色（Role）的权限管理。角色（Role）是分配给账号（User）的权限的集合，定义了允许账号在应用程序中查看和执行的操作。PolarDB-X中的Role操作与MySQL的操作兼容。

数据备份与安全

PolarDB-X兼容MySQL的常规备份策略。备份操作通常都发生在非主节点上，可以确保备份操作对在线业务流量无影响，同时支持定时全量备份、实时binlog增量备份的能力。支持任意时间点的一致性恢复。

PolarDB-X兼容MySQL的透明数据加密TDE，支持将数据表空间的文件做加密处理，确保业务数据的安全性。

数据导入导出

PolarDB-X兼容MySQL binlog复制协议。可以将PolarDB-X实例看作一个普通的MySQL节点，将其他MySQL节点作为PolarDB-X的同步源端或目标端。

PolarDB-X的binlog格式和MySQL原生格式一致，因此也可以用于CDC场景，例如，利用canal等将PolarDB-X的写入数据同步到其他存储中。

生态工具

阿里云提供众多与数据相关的产品和服务，PolarDB-X通过打通与其他云产品的连接，提供了更加原生的生态互通能力，为更多的业务场景提供了便利。PolarDB-X兼容阿里云数据库的生态工具，包括DMS、DTS、DBS、DAS、DataWorks等。

- **与DTS集成**

  DTS是一款集数据迁移、订阅及实时同步功能于一体的数据传输产品。PolarDB-X通过与DTS深度集成，提供了覆盖几乎所有常见数据库类型的数据导入和导出链路，详细内容请参见使用DTS导入和导出数据。

- **与DMS集成**

  DMS是一款集多种服务于一体的的数据管理服务产品。通过DMS可以对PolarDB-X实例进行数据管理、结构管理、用户授权等多种操作，详细内容请参见DMS官方文档。

- **与DAS集成**

  DAS是一款基于机器学习和专家经验实现数据库自感知、自修复、自优化、自运维及自安全的云服务。PolarDB-X目前已接入性能趋势、索引推荐、空间分析、SQL限流、实时性能、锁分析、SQL洞察等能力，详细内容请参见DAS官方文档。

- **与云监控集成**

  云监控是一款云资源监控报警产品。PolarDB-X已接入云监控并提供对计算资源和存储资源的监控告警能力，具体可参考监控与告警。

4.技术白皮书

4.1. 行业趋势与背景

本文介绍了数据库行业的发展及PolarDB-X的技术发展历程。

行业趋势

数据库系统至关重要

数据库与操作系统、中间件并称为系统软件的三架马车，是企业IT系统不可或缺的组件，也是互联网应用级企业信息管理系统存储数据和管理数据的核心平台。数据库系统是一切应用软件的数据处理及交换核心，是所有基础软件进行数据存储、查询、分析处理的中心。数据库系统运行性能是否高效稳定、接口语言是否通用，直接决定了上层应用能否给用户提供最好的性能，同时直接影响开发人员效率。根据全球权威IT研究咨询公司Gartner数据，2017年全球企业基础软件市场规模1958.52亿美金，其中数据库市场规模388亿美金，占比近20%，是最大组成部分。

分布式数据库是发展方向

近年来，随着互联网、大数据的飞速发展，特别是“双十一”指数型的成交总额发展曲线，让世界看到了中国电子商务业务的火箭式发展势头。而同时，对于背后的业务支撑系统来说，同样经历了火箭式的系统压力增长。以“双十一”为例，除了直接面向用户的电商网站系统，还有背后各相关物流公司的物流系统，各相关银行的支付系统， 各卖家的仓储系统等等。

随着普惠金融、数字金融的快速推进，数据能力也已经成为金融机构在新时代提升业务能力的重要抓手。与此同时，移动互联网和电子支付业务的蓬勃发展，给金融行业的典型应用场景，如核心账户与账务交易、在线支付/移动支付交易业务、实时交易监控与指标分析等，带来新数据形态下的金融系统能力需求。

传统的银行支付系统通常采用所谓的“IOE”结构，即IBM的大型机或小型机、Oracle的数据库、EMC的存储设备，这套系统仍存在成本巨大且与服务提供商严重捆绑的隐患。随着移动互联网的快速发展以及电子支付的兴起，对于银行核心支付系统来说是爆炸式的业务增速。传统的“IOE”系统为集中式系统，但是随着“摩尔定律”的逐渐失效，单台服务器上的性能极限已经慢慢显露，在这种情况下，技术人员逐渐开始探索将原有的集中式系统改造为分布式系统，通过多台中低端服务器的水平扩展方式替换一台高性能服务器垂直扩展方式，同时完成降低成本、增长性能的目的。在这种需求下，未来大规模分布式事务型数据库成为解决分布式系统数据存储、管理的主要方向。

PolarDB-X技术发展

产品前言

PolarDB-X是由阿里巴巴自主研发的云原生分布式数据库，融合分布式SQL引擎DRDS与分布式自研存储X-DB，基于云原生一体化架构设计，可支撑千万级并发规模及百PB级海量存储。专注解决海量数据存储、超高并发吞吐、大表瓶颈以及复杂计算效率等数据库瓶颈问题，历经各届天猫双十一及阿里云各行业客户业务的考验，助力企业加速完成业务数字化转型。

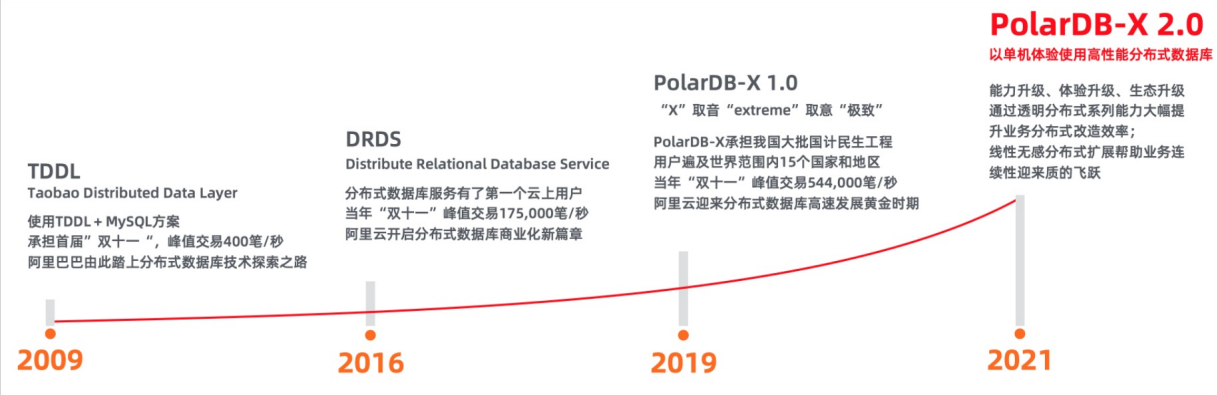
PolarDB-X核心能力通过标准关系型数据库技术实现，配合完善的管控运维及产品化能力，使其具备稳定可靠、高度可扩展、持续可运维、类传统单机MySQL数据库体验的特点。

PolarDB-X在公共云和专有云环境沉淀打磨多年，历经各届天猫双十一核心交易业务及各行业阿里云客户业务的考验。承载大量用户核心在线业务，横跨互联网、金融支付、教育、通信、公共事业等行业，是阿里巴巴集团内部所有在线核心业务及众多阿里云客户业务接入分布式数据库的事实标准。

发展历程

2003年，淘宝网成立之初，采用经典的LAMP架构（Linux-Apache-MySQL-PHP）。随着用户量迅速增长，单机MySQL数据库很快便无法满足数据存储需求。之后，淘宝网进行了架构升级，数据库改用Oracle。随着用户量的继续快速增长，Oracle数据库也开始成批的增加，即使这样，仍然无法满足业务对数据库扩展性的诉求。因此，阿里巴巴内部在2009年时发起了著名的去IOE运动，PolarDB-X也开启了自己的演进之路。

整个技术架构的发展，可分为四个阶段：



TDDL

去IOE的关键一环是实现对Oracle的替换。当时淘宝的业务体量已很难用成熟的技术产品支撑，为了避免以后出现技术瓶颈，技术的自力更生和自主可控成为一个核心诉求。一方面，随着x86技术日趋成熟，稳定性与小型机的差距不断缩小，另一方面，MySQL采用轻量化线程模型并具备高并发的支持能力，其生态逐步完善，因此新方案采用了基于Sharding技术+开源MySQL的分布式架构（TDDL+ AliSQL），这代产品的特征是以解决扩展性为目标、面向系统架构使用，尚不具备产品化能力。

DRDS

随着TDDL时代架构的逐渐成熟，2014年开始，基于阿里云，数据库的发展走上了云数据库之路。作为分库分表技术的开创者，我们推出了DRDS+RDS的分布式云数据库服务。这代产品的特征是采用Share-Nothing架构，以解决存储扩展性为出发点，提供面向用户的产品化交付能力。DRDS是国内第一家落地分布式技术的云服务，成为云市场上分布式数据库技术方向的开创者和引领者。

PolarDB-X 1.0

针对分库分表使用中的痛点，我们不断进行产品能力迭代，陆续支持了分布式事务、全局二级索引、异步DDL等内核特性，持续改进SQL兼容性，实现子查询展开、Join下推等复杂优化，并开发了平滑扩容、一致性备份恢复、SQL闪回、SQL审计等运维能力。这期间，我们不断扩展 Sharding 技术的能力边界，探寻它的能力上限。这个探索的过程，一方面使我们的计算层能力更加稳定、丰富和标准化，另一方面也促使着DRDS从中间件到分布式数据库的蜕变。2019年PolarDB-X全新产品线发布，成为一个新的里程碑。

PolarDB-X 2.0

2018年开始，我们逐渐触碰到了计算层的能力边界，比如无法提供RR隔离级别的事务能力、计算下推受限于SQL表达能力、数据查询的传输效率底下、多副本的线性一致性不可控等，这些问题像一个无法穿透的屏障，但我们能看到屏障的对面是什么，能看到所有障碍都指向了同一个方向：计算层需要与存储层深度融合。

庆幸的是，我们的AliSQL分支从诞生起就没有停止前进的步伐。通过集团业务多年的技术锤炼，基于AliSQL演化而来的X-DB数据库（包括X-Paxos协议库、X-Engine存储引擎等），在三副本、低成本存储等技术有了非常好的沉淀。

与此同时，基于云原生架构理念的PolarDB，通过引入RDMA网络优化存储计算分离架构，实现了一写多读的能力，并提供资源池化，降低用户成本；具备优化并提供秒级备份恢复、秒级弹性等能力，成为公有云增速最快的数据库产品。

基于这些技术探索和沉淀，我们开始思考基于云架构的分布式数据库应该是什么样的形态。

从用户角度来看，需要满足用户使用云的一些期望，比如用户的数据库数据永远不会丢，即使主机异常宕机，这里需要有数据强一致以及高可用容灾等能力；比如随着移动互联网和IoT的普及，数据层面会有爆炸式的增长，以及疫情之后有更多的企业会关注IT成本，因此高性能、低成本和可扩展的计算和存储能力也成为普适性诉求；另外，按查询付费的弹性能力，也是市场的一个诉求。

因此，下一代的分布式数据库需要具备：金融级高可用和容灾、水平扩展、低成本存储、按需弹性、透明分布式、HTAP混合负载、融合新硬件等特性。

2021年，我们完成原DRDS SQL引擎和X-DB数据库存储技术的融合，并结合PolarDB的云原生特性，承上启下推出了新一代的云原生分布式数据库，就此开启了PolarDB-X 2.0时代。该时代的产品专注解决单机解决不好的分布式扩展性问题，满足分布式数据一致性要求，并支持从单机到分布式的平滑演进，利用云原生技术的优势提供低成本和弹性能力，在交付上具备线上公有云、线下专有云、轻量化等全形态输出。

4.2. 技术架构

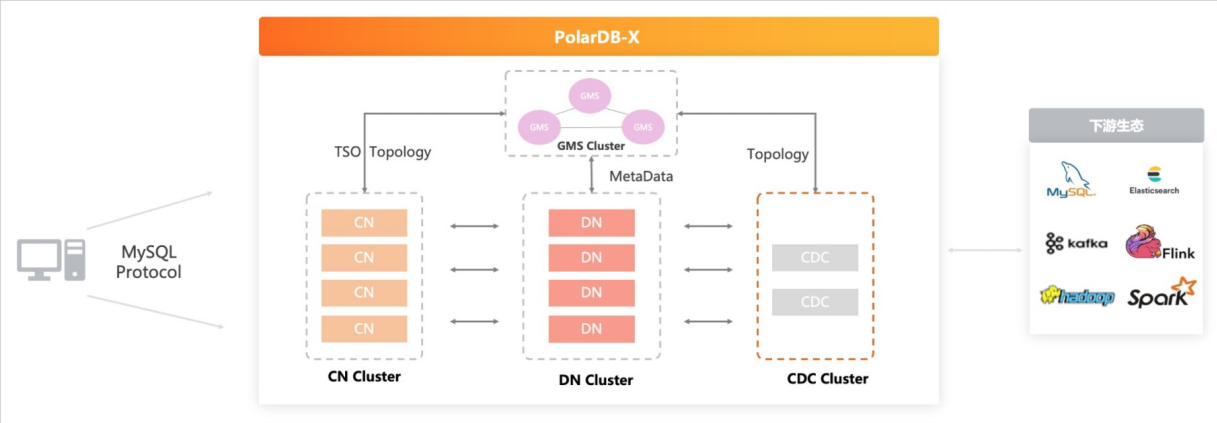
本文介绍了PolarDB-X的技术架构及优势。

架构演进理念

当前，分布式领域有3大技术方向：Sharding技术，NewSQL原生分布式技术，云原生DB技术。每种分布式都有其独特的优势和特点。的架构PolarDB-X继承了DRDS和X-DB技术的稳定性，结合了PolarDB的云原生技术，融入了NewSQL对于分布式数据一致性的能力，为用户提供新的**云原生+分布式**的产品体验。

整体架构

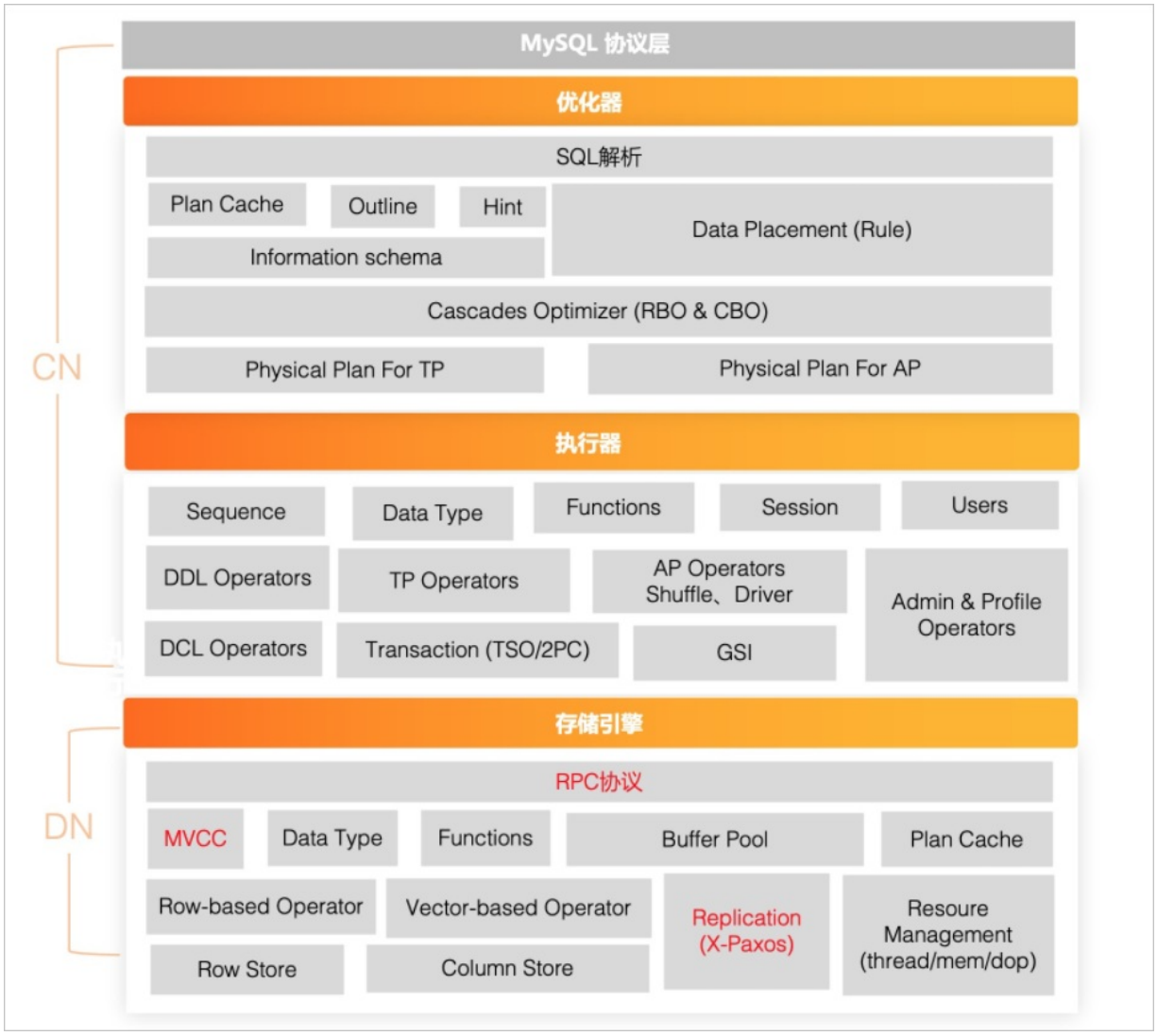
PolarDB-X的整体架构如下：



核心组件

- 元数据服务（Global Meta Service，GMS），主要提供分布式的元数据，提供全局授时服务（TSO）、维护Table/Schema、Statistic等Meta信息、维护账号、权限等安全信息。
- 计算节点（Compute Node，CN），主要提供分布式SQL引擎，包含核心的优化器和执行器。基于无状态的SQL引擎提供分布式路由和计算，解决分布式事务2PC协调、分布式DDL执行、全局索引维护等。
- 存储节点 （Date Node，DN），主要提供数据存储引擎，基于多数派Paxos共识协议提供高可靠存储、分布式事务的MVCC多版本存储，另外提供计算下推能力满足分布式的计算下推要求（比如Project/Filter/Join/Agg等下推计算），可支持本地盘和共享存储。
- 日志节点（Change Data Capture，CDC），主要提供兼容MySQL生态的主备复制协议，兼容Binlog协议和数据格式、支持主备复制Replication的协议和交互。

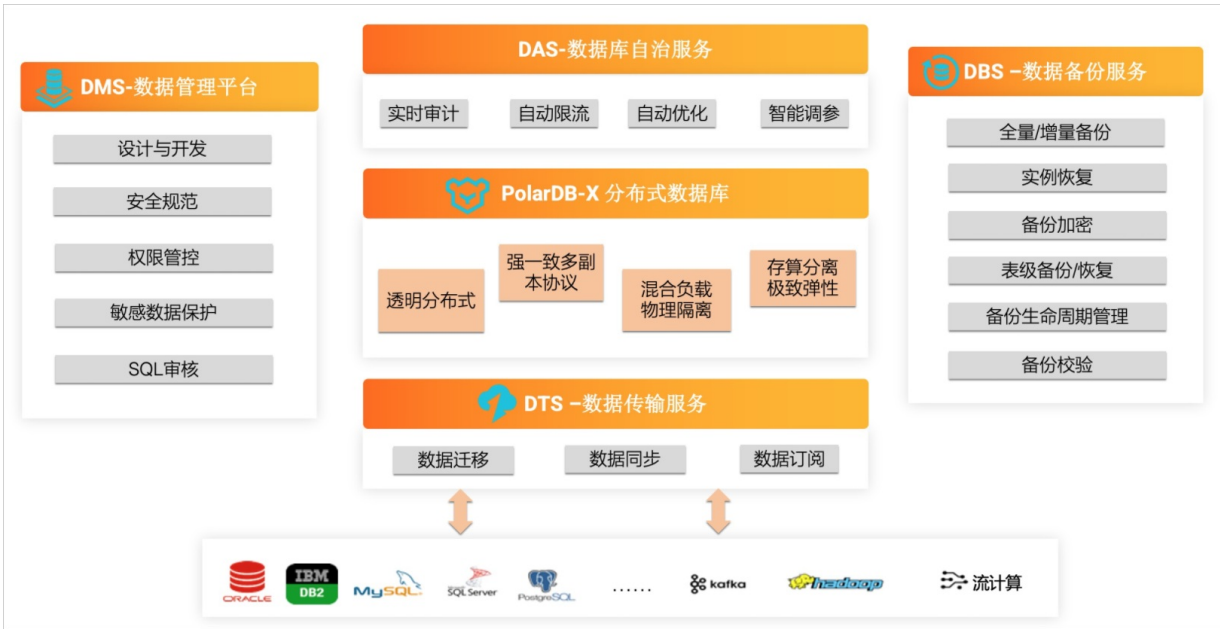
核心模块



如同大多数传统单机关系型数据库，PolarDB-X分为网络层、SQL解析层、优化层、执行层、存储层，其中优化层包含逻辑优化和物理优化，执行层包含单机两阶段执行、单机并行执行和多机并行执行，存储层包含应用了多种传统单机数据库优化和执行技术。

生态工具

PolarDB-X高度兼容MySQL协议和生态，可支持MySQL开源生态中常用的MySQL驱动（JDBC/ODBC）、多语言兼容（Java/GO/C/C++/Python等）、数据导入导出工具、各种客户端GUI等。PolarDB-X提供完整的生态工具方案，如下图所示：



闭环的生态工具：

- 数据管理DMS，基于阿里巴巴集团十余年的数据库服务平台的云版本，提供免安装、免运维、即开即用、多种数据库类型与多种环境统一的web数据库管理终端，可以为企业用户快速复制搭建与阿里集团同等安全、高效、规范的数据库DevOps研发流程解决方案。
- 数据库自治服务（Database Autonomy Service，简称DAS），一种基于机器学习和专家经验实现数据库自感知、自修复、自优化、自运维及自安全的云服务，帮助用户消除数据库管理的复杂性及人工操作引发的服务故障，有效保障数据库服务的稳定、安全及高效。
- 数据传输服务（Data Transmission Service，简称DTS），支持关系型数据库、NoSQL、大数据等数据源，集数据迁移、订阅及实时同步功能于一体，能够解决公共云、混合云场景下，远距离、毫秒级异步数据传输难题。其底层基础设施采用阿里双11异地多活架构，为数千下游应用提供实时数据流。
- 数据库备份（Database Backup，简称DBS），低成本、企业级的云原生备份平台。它可以为多种环境的数据库提供强有力的保护，包括本地数据中心、其他云厂商、公共云及混合云。

产品架构

PolarDB-X作为阿里云的成熟云产品，在产品化的配套能力上支持比较完整，包括白屏化的运维操作、多样化的交付形态、完整的OpenAPI、配套的生态工具等。

PolarDB-X部署架构整体基于Kubernetes，运行在高性能的物理机，产品架构设计如下：

1. PolarDB-X实例由多个节点组成，产品购买以节点为单位，初次购买必须包含2个节点，后续扩容和缩容最小为1个节点。每个节点会有多种规格，比如4c16g、8c32g、16c64g等。
2. PolarDB-X根据资源隔离诉求的不同，可以分为：通用规格、独享规格、独占物理机规格。 比如通用规格，同一硬件资源下的不同PolarDB-X实例，会互相充分利用彼此空闲的计算资源（如CPU），通过复用计算资源享受规模红利，性价比会更高。而独享规格每个PolarDB-X实例会独占所分配到的计算资源（如CPU），而不会与同一服务器上的其他集群共享资源，性能表现更加稳定。
3. PolarDB-X支持公有云、混合云的多样化交付。在公有云，支持多地域、多可用区的交付，同时在网络和安全层面，支持VPC、IP白名单、非对称账号密码、TDE加密等方式，确保数据服务安全。 在混合云，支持基于DBStack轻量化输出，可在用户已有机器硬件上完成数据库部署和运维。

架构优势

高可用

经过阿里多年双十一验证的X-DB（X-Paxos共识协议能力），提供数据强一致，保证节点故障切换时RPO=0。另外支持多样化的部署和容灾能力，比如基于Paxos强同步的同城三机房、三地五中心，另外搭配binlog异步复制的两地三中心、异地灾备、异地多活等。尤其在异地长距离传输上，基于Batching & Pipelining 进行网络优化来提升性能。

高兼容

PolarDB-X主要兼容MySQL，包括SQL、函数类型等，技术上引入全局时间授时服务，提供全局一致性的分布式事务能力，通过TSO+2PC提供数据库完整的ACID能力，满足分布式下的Read-Committed/Repeatable-Read的隔离级别。同时在分布式事务的基础上，提供全局二级索引能力，通过事务多写保证索引和主表数据强一致的同时，引入基于代价的CBO优化器实现索引选择。除此以外，在元数据和生态对接层面，PolarDB-X基于Online DDL的技术提供了分布式下元数据的一致性。同时硬件层面，兼容主流国产操作系统和芯片认证，比如麒麟、鲲鹏、海光等。

另外在业界主流的分布式数据库里，分布式下的redolog/binlog等数据库变更日志其实一直被厂商所忽视，从关系数据库的发展历史来看，生态和标准对于市场规模化非常重要，PolarDB-X支持全局binlog能力，全面兼容和拥抱MySQL数据库生态，用户可以将PolarDB-X当做一个MySQL库，采用标准的binlog dump协议获取binlog日志。

高扩展

PolarDB-X基于Share-Nothing的架构支持水平扩展，同时支持数据库在线扩缩容能力，在OLTP场景下可支持千万级别的并发、以及PB级别的数据存储规模，同样在OLAP场景下，引入MPP并行查询技术，扩展机器后查询能力可线性提升，满足TPC-H等的复杂报表查询诉求。

HTAP

随着移动互联网和Iot设备的普及，数据会产生爆炸式的增长趋势，传统的OLTP和OLAP的解决方案是基于简单的读写分离或者ETL模型，将在线库的数据T+1的方式抽取到数据仓库中进行计算，这种方案存在存储成本高、实时性差、链路和维护成本高等缺陷。PolarDB-X设计中支持OLTP和OLAP的混合负载的能力，可以在一个实例里同时运行TPC-C和TPC-H的benchmark测试，保证AP的查询不影响TP流量的稳定性。核心技术层面，也有所创新，比如我们会在计算层精确识别出TP和AP的流量，结合多副本的特性和多副本的一致性读能力，智能将TP和AP路由到不同的副本上，同时在AP链路上默认开启MPP并行查询技术，从而在满足隔离性的基础上，线性提升AP的查询能力。在存储层上，我们也在完善计算下推能力，未来也会提供高性能列存引擎，实现行列混合的HTAP能力。

极致弹性

PolarDB-X结合PolarDB云原生的技术，可以基于PolarDB的共享存储+RDMA网络优化能力，提供秒级备份、极速弹性、以及存储按需扩展的能力。基于共享存储的基础上，结合分布式的多点写入能力，可以在不迁移数据的前提下提供秒级弹性的能力，给到用户完全不一样的弹性体验。

开放生态

PolarDB-X全面拥抱和坚定MySQL的开源生态，做到代码完全自主可控的同时满足分布式MySQL的兼容性，架构做到简单开放，只要具备一定MySQL背景的人员即可完成持续运维。除此以外，PolarDB-X和阿里云的数据库生态有完整的闭环对接，支持如DTS/DBS/DMS等，可打通阿里云的整个大生态。

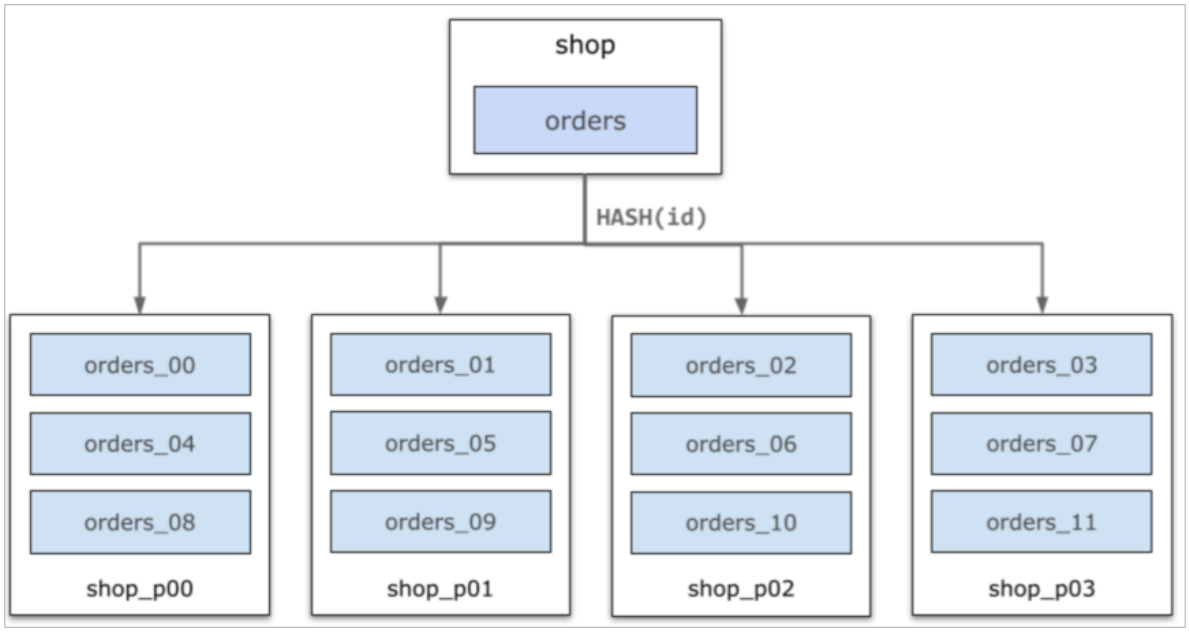
4.3. 技术原理

本文介绍了PolarDB-X的主要技术原理。

分布式线性扩展

PolarDB-X将数据表以水平分区的方式，分布在多个存储节点（DN）中。数据分区方式由分区函数决定，PolarDB-X支持哈希（Hash）、范围（Range）等常用的分区函数。

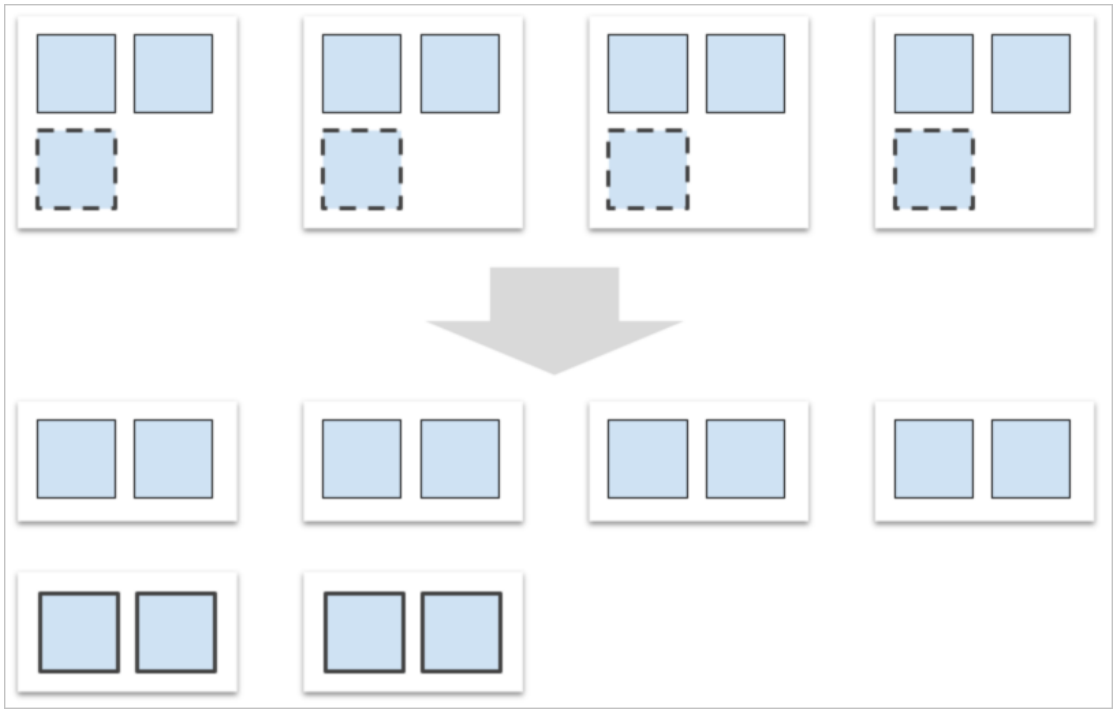
以下图为例，shop库中的orders表根据每行数据的ID属性的哈希，被分区水平切分成orders_00～orders_11共计12个分区，均匀分布在4个数据节点上。对于用户来说，通常无需关心具体的数据分布，PolarDB-X的分布式SQL层将会自动完成查询路由、结果合并等。



扩容迁移

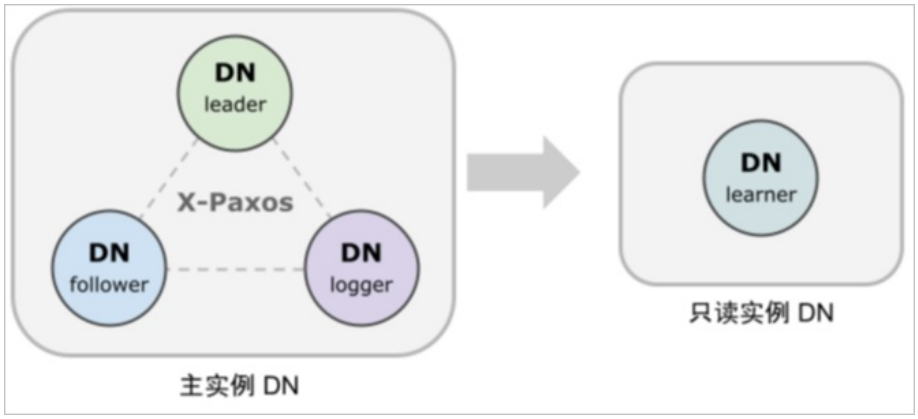
随着业务的增长，数据量越来越大，往往需要添加更多的数据节点以承载更多的数据。当新的数据节点加入实例时，PolarDB-X将自动触发扩容任务，将数据进行再平衡（Rebalance）。

以下图为例，orders表原本分布在4个数据节点上。用户进行扩容后，实例的数据节点数量从4个增加到6 个，触发PolarDB-X的再平衡任务，将部分数据分区从旧节点移动到新节点上。这一过程在后台利用空闲资源完成，对业务线上流量无影响。

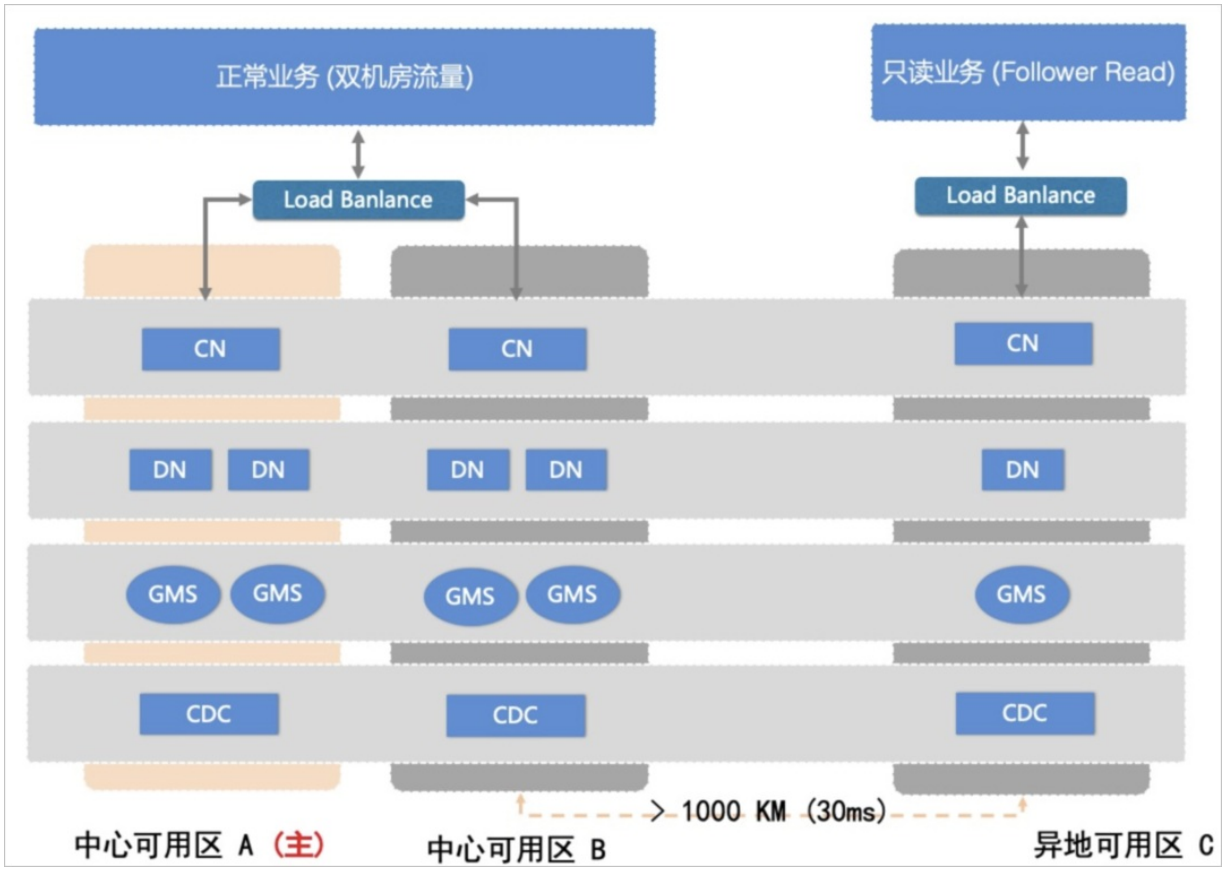


高可用与容灾

在生产环境部署数据库时往往会搭建多个副本（Replica），保证数据库实例的高可用性以及数据的持久性。为了保证副本间的强一致性，现代数据库往往采用以Paxos算法为代表的多数派复制协议，它要求实例中至少存在3个节点，每次写入都要获得超过半数节点的确认，即便其中1个节点宕机实例也仍然能正常提供服务。PolarDB-X采用X-Paxos复制协议，X-Paxos是阿里巴巴自研的Paxos协议实现，在功能、性能上都做了大量优化，且经历了数十载的双十一考验，稳定可靠。



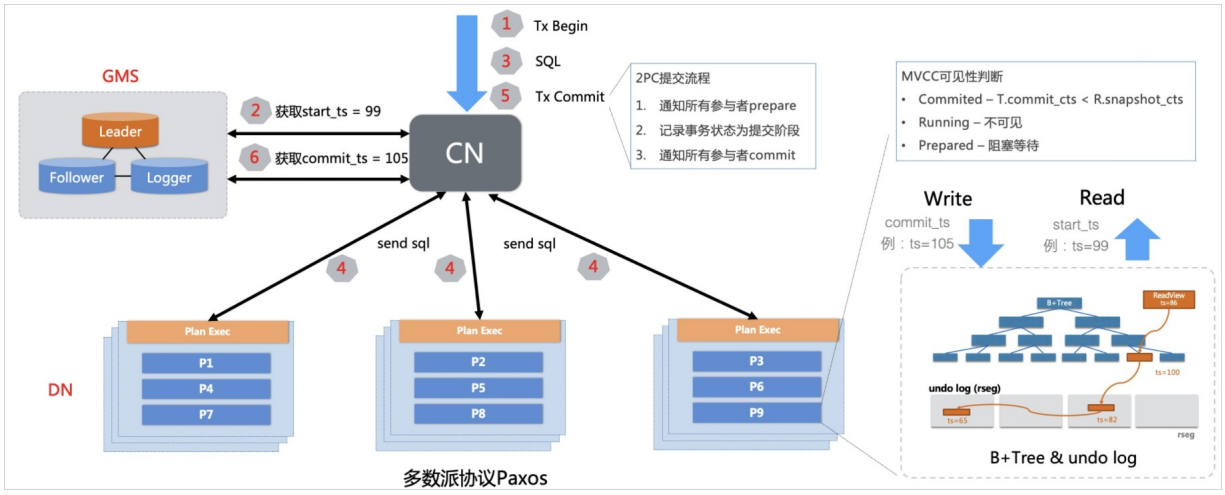
基于Paxos复制协议，PolarDB-X可以部署到多个机房中，以实现机房级容灾。常见的部署方式有同城三机房、两地三中心等，其中后者主要应用在混合云部署中。由于Paxos协议的特性，通常三个机房中有一个主机房负责对外提供服务。



分布式事务

PolarDB-X原生支持分布式事务，并保证事务的ACID性质——原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

PolarDB-X通过引入中心授时节点（TSO），结合多版本并发控制（MVCC），保证读取到的一定是一致的快照，而不会读到转账事务的中间状态。如下图所示，提交事务时，计算节点（CN）执行事务时从TSO 获取到时间戳，随着数据一同提交到存储节点（DN）多版本存储引擎上。读取时，如果查询操作的数据涉及多个分区，PolarDB-X首先会获取全局时钟作为读取版本，对每行数据的MVCC多版本进行可见性判断，确保读到全局一致的数据版本。

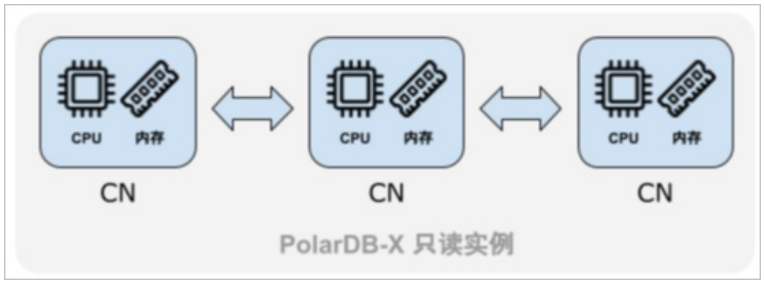


分布式事务也是许多分布式特性的基础，例如：对于读写分离，数据的事务多版本信息也会被同步到 Learner副本，保证读只读实例不会因为同步延迟读到过期数据；全局变更日志中，通过时间戳保证分布式事务顺序；任意时间点的数据恢复（PITR，point-in-time recovery）中，利用分布式事务时间戳，能够精准找到相应时间的、全局一致的数据版本。

混合负载 HTAP

PolarDB-X是一款支持 HTAP (Hybrid Transaction/Analytical Processing) 的数据库：在支持高并发、事务性请求的同时，也对分析型的复杂查询提供了良好的支持。分析型查询指的是涉及数据量较大、计算比较复杂的查询，例如对一定时间区间内的数据进行聚合。相比于业务中常见的简单查询，这类查询往往要执行数秒甚至数分钟，需要消耗较多的计算资源。

为了加速复杂分析型查询，PolarDB-X将计算任务切分并调度到多个计算节点上，从而利用多个节点的计算能力，加速查询的执行。这种方式也称为MPP并行计算。



PolarDB-X的优化器面向HTAP负载设计，对复杂查询有着良好的支持。PolarDB-X采用了基于代价的优化器技术，能够根据实际数据量、数据分布情况等，搜索到较优的执行计划，例如，对Join顺序进行调整、选择合适的Join或聚合算法、对关联子查询去关联化等。

PolarDB-X优化器会基于代价估计将请求区分为TP与AP负载，其中AP查询会被进一步改写为分布式执行计划，发往只读实例进行计算，避免它对主实例的TP查询造成影响。

MySQL生态兼容

PolarDB-X将兼容MySQL以及周边生态作为核心设计目标之一。本文从SQL语法、事务行为、导入导出等角度总结了兼容性相关特性，详细内容请参见MySQL生态兼容。

PolarDB-X通讯协议兼容MySQL协议，可以使用常见的MySQL客户端直接连接到PolarDB-X实例，包括 JDBC Driver、ODBC Driver、Golang Driver等。兼容MySQL SSL、Prepare、Load等传输协议。

PolarDB-X兼容MySQL的各种DML、DAL、DDL语法，包括：

- 兼容绝大部分MySQL函数（包括JSON函数、加密解密函数等）。
- 兼容MySQL 8.0的视图、CTE、窗口函数、分析函数等。
- 支持MySQL的各种数据类型，包括支持类型精度（比如时间戳、Decimal 类型）。
- 兼容常见的MySQL字符串Charset及Collation。
- 兼容绝大部分information_schema视图。

更多信息参见开发指南。

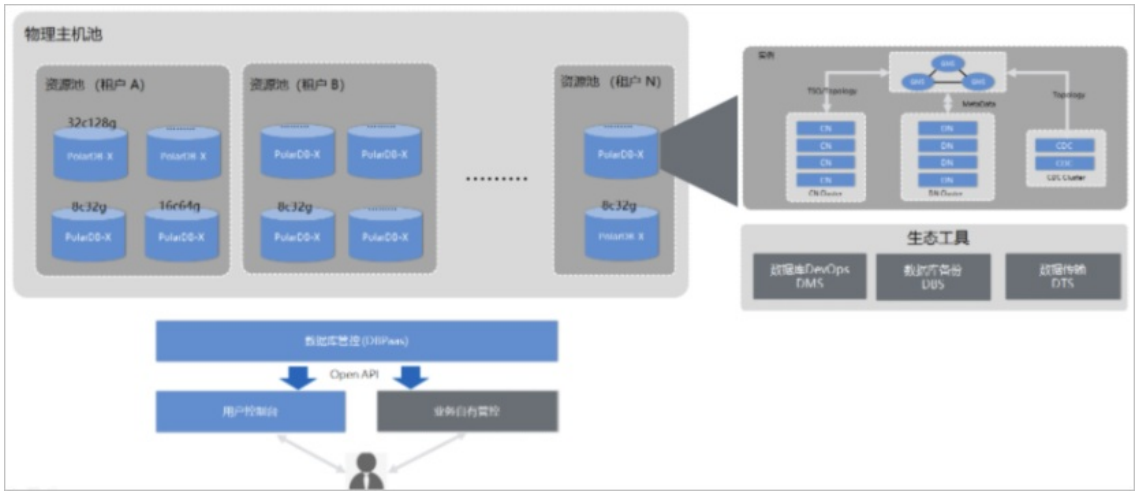
4.4. 部署与运维

本文介绍了PolarDB-X的部署及运维能力。

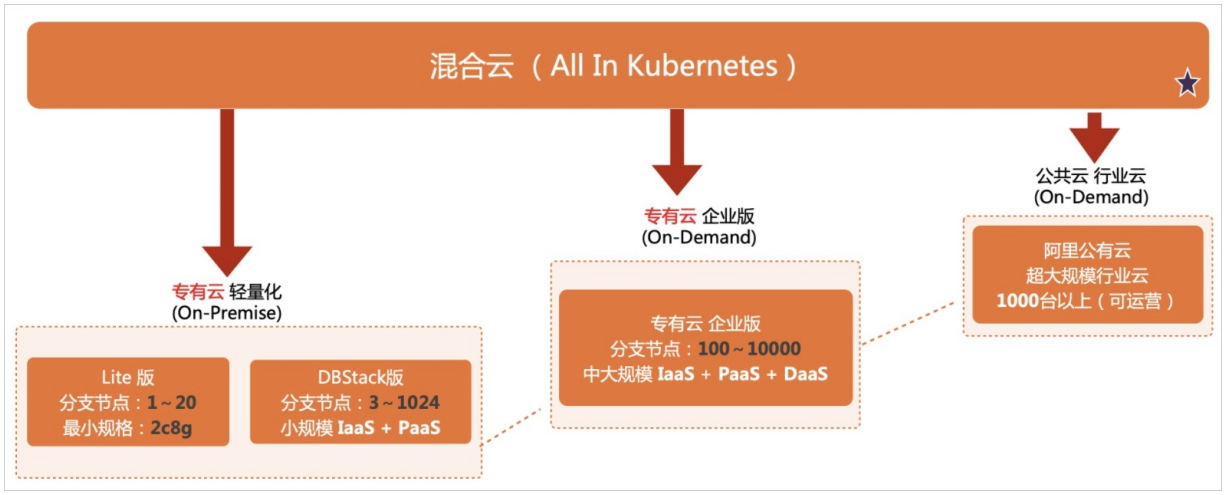
部署形态

PolarDB-X是一款在阿里云上商业售卖的云产品，在产品化的配套支持上比较完整，包括多样化的交付形态、完整的OpenAPI、配套的生态工具等。

PolarDB-X的DBPaas架构提供了对数据库实例的统一运维，除了DBPaas内置的用户控制台以外，允许用户通过OpenAPI接入业务自有管控。



PolarDB-X的DBPaas架构基于Kubernetes，能够很好的支持公有云以及混合云多形态的交付能力。公有云（或大规模行业云）部署最灵活。混合云部署可以选择企业版或更加轻量级的DBStack版，后者在生产级别可支持最少3台主机。



自动化运维能力

PolarDB-X DBPass控制台集成了创建释放实例、备份恢复、配置管理、监控报警、诊断优化等功能。用户可以通过控制台页面管理数据库、用户、权限、参数等，或执行数据备份、设置监控报警等日常运维工作。

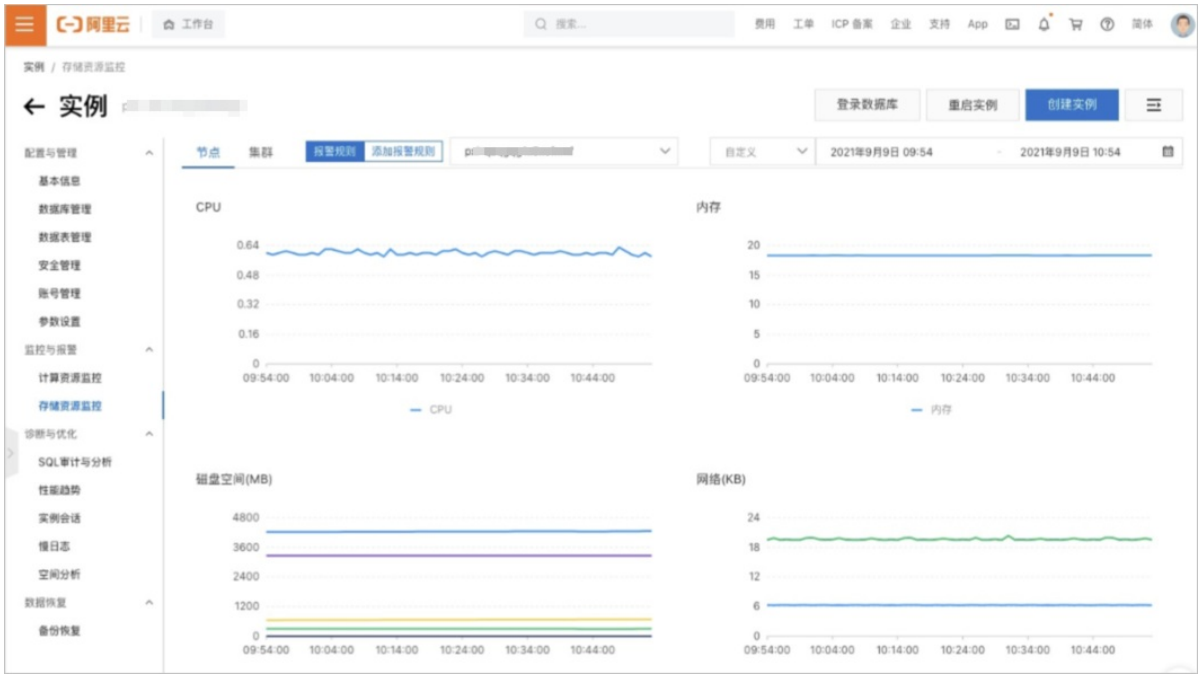
为了便于用户将PolarDB-X与其他系统集成，PolarDB-X DBPass平台也提供了丰富的OpenAPI接口，包括实例管理接口、数据库管理接口，监控数据查询接口、参数管理等。详细接口列表请参见API概览。

监控与诊断优化

资源监控

您可以在PolarDB-X控制台上查看计算资源监控、存储资源监控和数据库监控等监控信息，或依据这些监控指标添加报警规则。

- 计算资源监控是计算节点的监控指标，包括CPU使用率、内存使用量、网络流量（流入流出）、QPS、查询延迟、前端连接数、活跃线程数等。
- 存储资源监控是存储节点的监控指标，包括CPU使用率、内存使用量、磁盘空间用量、网络流量（流入流出）、连接数、活跃线程数、IOPS、QPS、TPS、缓冲池用量、临时表创建数量等。



SQL分析与优化

PolarDB-X支持SQL日志的实时查询、可视化分析、设置告警等操作。您仅需在控制台上打开功能开关，即可实时进行SQL日志的审计与分析；也可以基于某些指标定制实时的监测与告警，当关键业务出现异常时可通过多种途径通知告警。

控制台上可以查看当前实例会话，也可以选择终止会话。会话统计页面可查看按客户端统计、按namespace统计等的聚合统计信息。10秒SQL分析功能可以在10秒内对会话进行采样（每秒采样一次），然后对采样结果进行汇总分析，通过分析结果可以很容易看出哪些查询执行次数最多、是否存在慢SQL等。

慢日志查询功能能帮助您快速发现影响数据库性能的关键SQL。慢日志功能提供交互页面，您可以查看慢SQL模板、慢SQL明细，以及每条慢SQL执行次数、耗时、返回行数等关键信息。慢SQL诊断优化还能根据SQL执行计划智能地给出优化方案，例如推荐创建本地索引或全局索引。

为防止少数SQL占用过多资源、影响业务整体运行，SQL限流功能能够限制特定查询的并发度。您可以通过控制台创建限流匹配规则，限制其最大并发度、最长等待队列长度等。

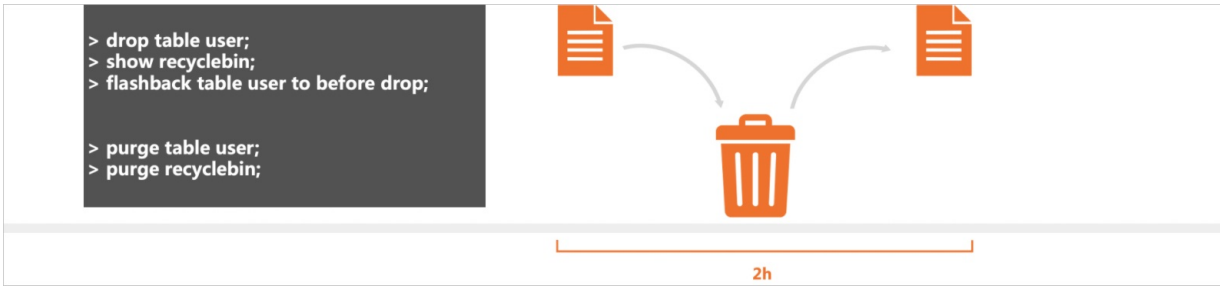
备份与恢复

一致性备份恢复

一致性备份恢复提供实例级任意时间点（精确到秒级）的历史数据恢复能力，基于定期的全量快照备份和增量binlog实现。

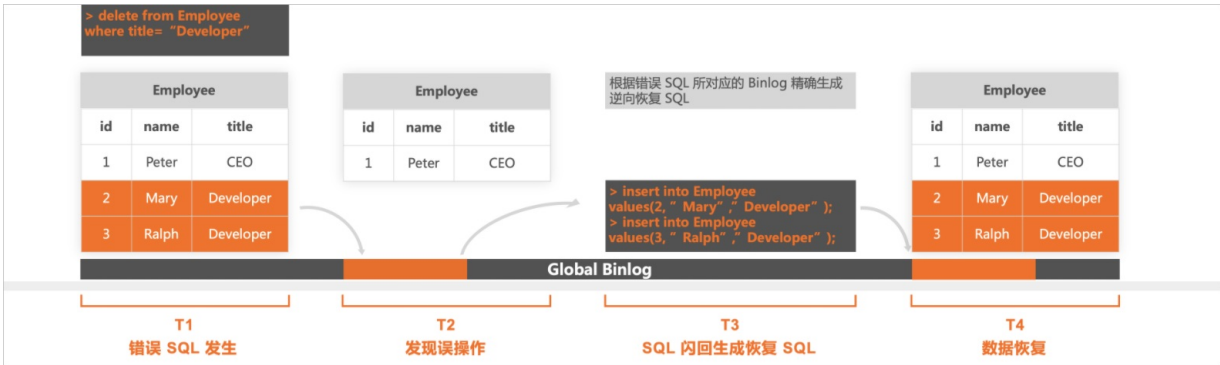
表回收站

当表回收站功能开启时，被删除或清空的表会被暂时放入回收站，若2小时内需要恢复表，可以在**控制台—表回收站**页面中找回。



SQL闪回

SQL闪回功能能够在误操作发生后，帮助您回滚或部分回滚之前的操作。例如，当 `DELETE` 语句误删除数据时，可以通过SQL闪回功能为它生成逆向恢复SQL（即相应的 `INSERT` 语句），从而恢复丢失的数据。

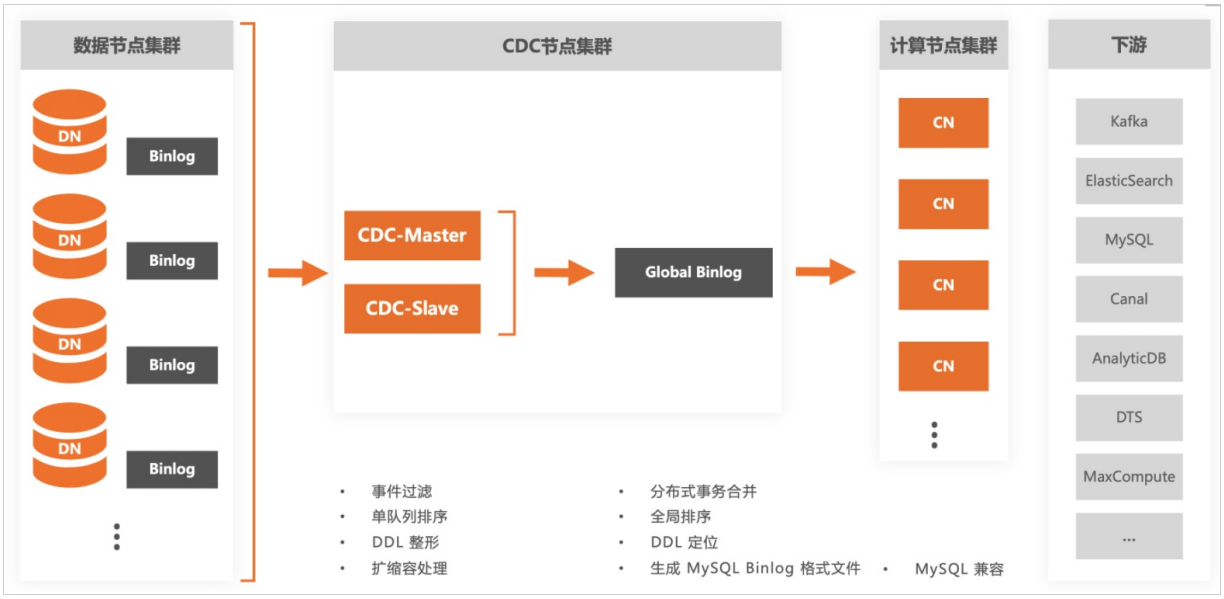


数据导入导出

全局变更日志CDC

PolarDB-X兼容MySQL binlog复制协议。用户可以将PolarDB-X实例看作一个普通的MySQL节点，将其他MySQL节点作为PolarDB-X的同步源端或目标端。

PolarDB-XCDC组件能够提供与MySQL binlog格式兼容的变更日志，并且对外隐藏掉实例扩缩容、分布式事务、全局索引等分布式特性。用户也可以利用canal等开源工具将PolarDB-XB-X的写入数据同步到其他存储中。



MySQL生态工具

PolarDB-X将兼容MySQL以及周边生态作为核心设计目标之一，通讯协议、SQL语法等兼容MySQL、mysqldump、LOAD DATA INFILE等功能可以直接使用。

PolarDB-X也兼容阿里云数据库的生态工具，包括DMS、DTS、DBS、DAS、DataWorks等。

4.5. 解决方案与客户案例

异地多活数据库解决方案

方案背景

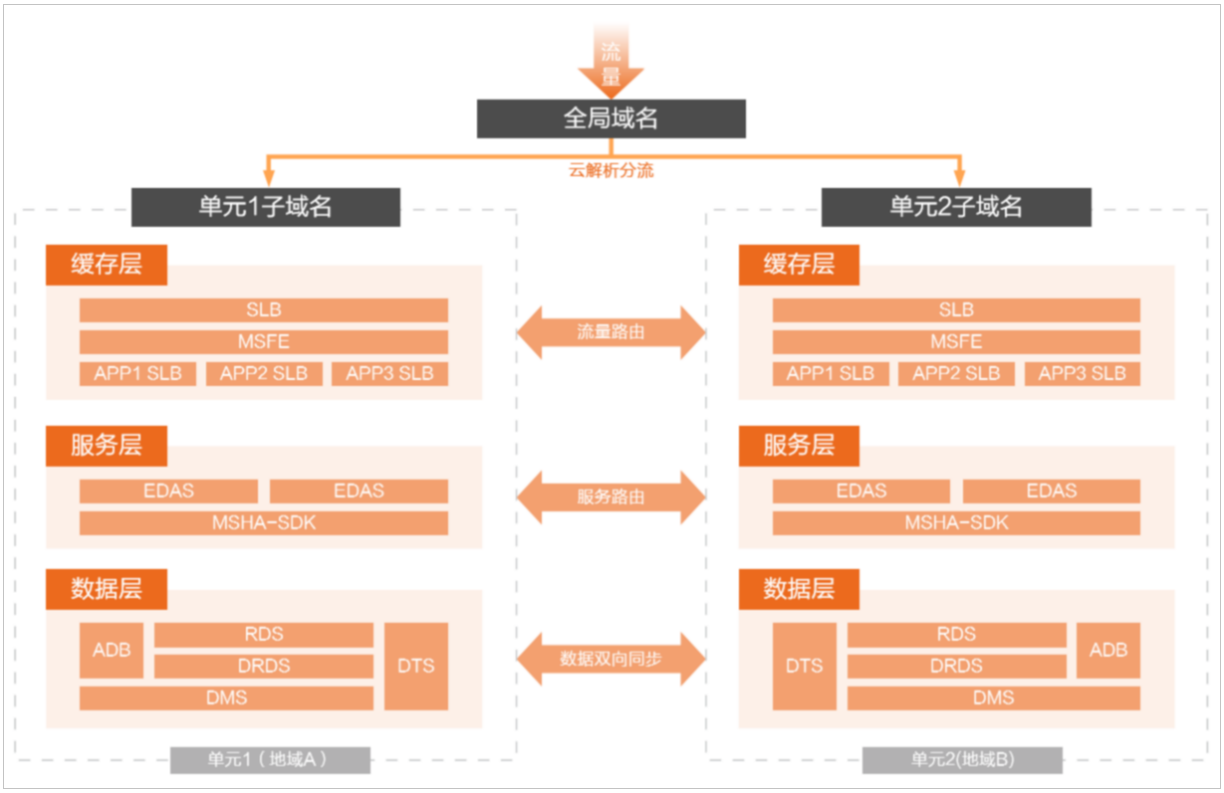
随着云计算的蓬勃发展，越来越多信息系统选择部署在云计算环境下，因此基于云产品为信息系统的服务能力和数据质量提供保障尤为重要。为了防止灾难性的故障如火灾、洪水、地震、区域电力中断或者人为破坏等对信息系统造成不可挽回的损坏，需要构建容灾系统来保障信息系统的可用性和安全性。

2007年，国务院信息化办公室联合银行、电力、民航、铁路、证券等八大重点行业，制定发布了国家标准GB/T20988-2007《信息系统灾难恢复规范》，明确规定了容灾能力的6个等级要求。企业在构建容灾系统时往往会参考国标等级，或者以此作为合规要求。然而，大部分传统容灾方案如同城容灾、同城双活、异地容灾、两地三中心等很难达到国标5-6级要求，同时还存在成本浪费，灾备单元健壮性不足等问题。

异地多活是新一代的容灾解决方案，在保证业务持续高可用的同时还能实现成本优化、地域级水平扩展、持续高可用等能力，本文会着重介绍阿里云主流数据库产品在异地多活场景下的解决方案。

方案架构

异地多活从业务视角来看是通过对业务做自顶向下的流量隔离来实现的，按照某一个分流维度对业务流量进行划分，并路由到不同的地域。整个部署架构分多个地域，每个地域称之为一个单元，其中某个单元又承担着整个多活架构的逻辑中心角色，提供一些中心化的服务能力（如sequence_分发，强一致读服务等），每个单元内的业务架构分为接入层、服务层、数据层：



- **接入层**：业务流量通过租户侧DNS解析后按照权重分配到不同单元的接入层，进入接入层后，通过解析请求header/cookie中的分流标，对比自定义的分流规则，判断请求是否归属本单元，若归属本单元则走到本单元的服务层。否则流量需要转发到对端单元，根据不同的内网连通性可以走upstream或304跳转。
- **服务层**：服务层部署的是业务应用系统，包括中间件等。如果使用了阿里云的中间件如CSB、RPC框架、MQ等，可以通过相应产品的多活接入能力实现在服务层的流量路由和纠错。针对RPC服务，在多活场景中分为：单元化服务、中心化服务、普通服务三类：
  - 单元化服务：每个单元内完全自治的服务，是多活场景下的主要服务类型，会对流量进行单元归属判断，并进行流量纠错。
  - 中心化服务：强依赖中心单元的服务，流量都会被转发到中心单元，通过中心单元的服务访问中心单元的数据层。
  - 普通服务：不做任何改造的服务，在本单元内进行服务调用，并访问本单元的数据层。

  支持多活能力的主要服务类型是单元化服务，但是当业务系统比较复杂时，可能难以实现所有业务模块均按某一个维度划分，或者某些业务服务必须要单点部署以避免分布式部署的一致性问题，此时可由中心化服务和普通服务提供相应能力支持。
- **数据层**：数据层解决数据库跨地域的部署与同步问题，并在灾难发生时对流量切换动作提供相应的数据质量保护策略。针对上层业务不同的服务类型提供UNIT和COPY两种数据同步策略：
  - UNIT类型：每个单元部署独立的数据库系统，单元之间通过DTS进行数据【双向】实时同步，保持每个单元都有全量数据，每个单元均可进行读写操作，读写流量会根据业务定制的分流策略进行单元写保护，这种同步策略用于支持服务层的单元化服务类型，是多活场景的核心同步策略。
  - COPY类型：每个单元部署独立的数据库系统，单元之间通过DTS进行数据【单向】实时同步，保持每个单元都有全量数据，中心单元可进行读写，非中心单元只提供读服务。这种同步策略用于支持中心化服务和普通服务，中心化服务路由回中心执行，普通服务可在单元内进行读。

方案应用场景

本解决方案适用于以下业务场景：

- **容灾能力要求高**：异地多活可以达到国标6级的容灾能力，适合对容灾方面有较高要求的业务、业务流量比较敏感的业务或业务的某些核心系统。
- **流量要求精细化管理**：异地多活支持多种流量管理策略，适合对流量管理有复杂需求的业务，例如按地域就近接入，按用户

信息分配指定数据中心等。

- **业务快速发展**：异地多活实现业务按照单元级别（数据中心级别）实施水平扩展，业务定义好一个单元的服务部署配比后，可以以此为镜像快速部署多个分布全国的单元，实现容量快速扩充。
- **业务读多写少**：读多写少业务可以从业务行为上天然的规避数据异步复制的各类问题，并且业务接入多活的改造成本也较低，是比较适合实施异地多活的业务场景。

方案价值

异地多活的价值具体表现在如下方面：

- **业务即容灾**：传统异地灾备或两地三中心的异地灾备中心常态不提供服务，当发生地域级灾难时难以保证异地灾备中心的可用性，存在切换成功率低的问题，异地多活架构下各个地域数据中心不区分角色，全部常态承载业务流量，时刻保证业务系统健壮，各个数据中心既是业务体系也是容灾系统。
- **业务连续性保障**：异地多活架构下各个数据中心常态承接业务流量，故障发生时只需调拨入口流量即可实现容灾切换，实现分钟级的容灾切换。同时随着参与多活建设的数据中心数量增加，参与调拨流量的比例会相应减少，未参与调拨的业务流量可以实现对容灾切换的**0**感知。
- **业务高速发展支撑**：业务高速发展，受限于单个地域的有限资源，尤其是数据层存在单点性能瓶颈，除异地多活以外的全部容灾架构都只能在主生产中心执行写操作。地域多活实现业务级的流量闭环，各个数据中心均可读写，具备水平扩展能力以及跨地域的快速扩建能力。
- **流量有效隔离**：异地多活本质上是提供了一种自顶向下的流量隔离能力，业务具备在数据中心级别完全隔离的能力，各个数据中心承载的流量大小可灵活调配，在最小隔离数据中心内（例如承载1%流量），业务可灵活进行风险可控的技术演进，例如基础设施升级、新技术验证等。
- **成本有效控制**：在考虑单个地域发生地域级灾难的场景下，不论两地三中心还是异地灾备都需要在灾备中心部署可承载100%流量的系统资源，加上生产中心即达到200%的成本冗余。异地多活通过跨城多数据中心部署，有效分摊各个数据中心成本，实现成本小于200%冗余。

案例-国税总局

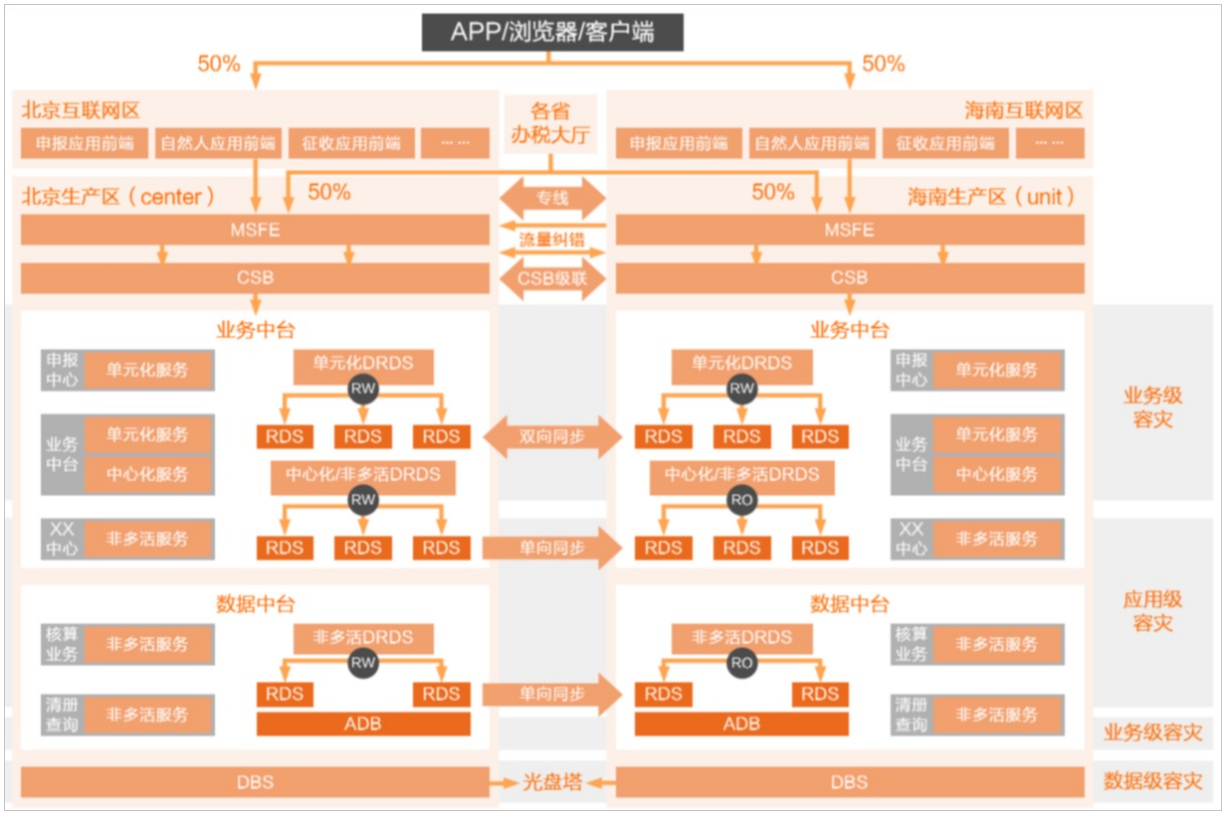
案例背景

个税系统是国家财税信息化领域的重大工程，集中了全国7.8亿自然人的基础信息和3.6亿活跃申报自然人的纳税申报等敏感信息，已成为国家重要战略信息资产，同时也是超大体量的政务云平台。在此背景下，一方面要考虑业务对容灾能力提出的强烈且苛刻的要求，另一方面要考虑超大体量下容灾建设带来的成本浪费问题。因此选择多活方案在支持容灾要求的同时需要解决成本、业务高速发展等问题。

案例架构

客户基于本方案，整合了TP/AP场景的多活能力，借助RDS、PolarDB-X、ADB、DTS、DMS、MSHA等产品，有效实施了异地多活容灾能力，达到国标6级容灾能力要求：

- RDS、PolarDB-X承载TP业务数据，ADB承载AP业务数据。
- MSHA实现多活流量管控和容灾切换动作。
- 通过DTS实现数据的跨域实时同步以及云内同步。
- DMS实现日常运维变更和数据变更管理。
- DBS实现数据在第三方平台的备份功能。



案例效果

- 针对客户不同的业务模块，实施多种分流策略，自然人电子税务局在线业务实施基于自然人档案号的分流，离线业务清册查询实施按地域分流的多活能力。
- 为客户提供国标6级的容灾效果，实现秒级容灾切换并保证数据零丢失。
- MSHA实现多活流量管控和容灾切换动作。
- 客户部署了两单元，常态每个单元承载50%的业务流量，充分利用两单元的资源。
- 借助多活管控灵活的流量分配策略，实现重大业务发布时的灰度放量能力。

案例-联通新客户

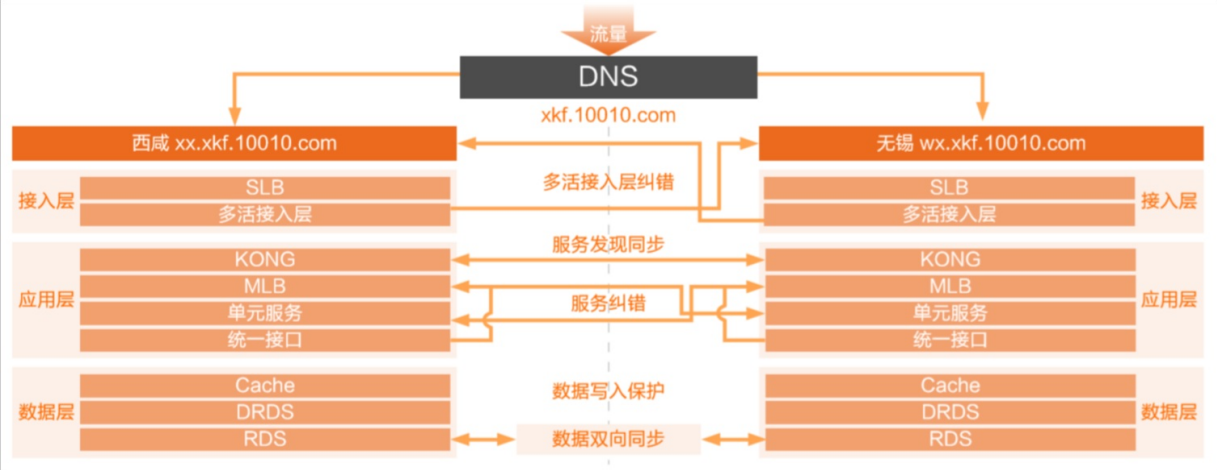
案例背景

联通新客服系统承担着联通全国的客服业务，对持续高可用能力有极高要求，同时也是联通向全站高可用演进的起点，其业务特点以TP业务为主。

案例架构

客户基于此方案，整合RDS、PolarDB-X、DTS、MSHA产品能力，实现了整个新客服系统7个业务中心的多活能力。

- RDS、PolarDB-X承载业务数据并对接多活管控系统。
- DTS实现数据的跨城实时同步和状态上报。
- MSHA实现多活流量管控和容灾切换动作。



案例效果

- 联通新客服系统的接入中心、外呼中心、业务支撑等7个业务实现按地域多活分流。
- 实现多次容灾演练，对多个省份进行切流，秒级完成切换，数据零丢失。
- 客户部署了两单元，常态两个单元均承载业务流量，充分利用两单元的资源。

5.性能白皮书

5.1. TPC-C测试

本文介绍如何使用TPC-C工具测试PolarDB-X数据库的联机交易处理（偏向OLTP能力），您可以按照本文的介绍自行测试进行对比，快速了解数据库系统的性能。

背景信息

TPC-C是业界常用的一套Benchmark，由TPC委员会制定发布，用于评测数据库的联机交易处理（偏向OLTP）能力。主要涉及10张表，包含了NewOrder（新订单的生成）、Payment（订单付款）、OrderStatus（最近订单查询）、Delivery（配送）和StockLevel（库存缺货状态分析）等五类业务事务模型。TPC-C使用tpmC值（Transactions per Minute）来衡量系统最大有效吞吐量（MQTh，Max Qualified Throughput），其中Transactions以NewOrder Transaction为准，即最终衡量单位为每分钟处理的新订单数。

> **说明** 本文中TPC-C的实现基于TPC-C的基准测试，并不能与已发布的TPC-C基准测试结果相比较，本文中的测试并不符合TPC-C基准测试的所有要求。

测试条件

- **测试数据量**

  基于1000 Warehouse，其中主要表的数据量如下：

  - bmsql_order_line表3亿行
  - bmsql_stock表1亿行
  - bmsql_customer、bmsql_history、bmsql_oorder各3000万行

- **测试所用实例规格**

节点规格	节点数
4c16g	2
4c16g	4
8c32g	2
8c32g	4
16c64g	2
16c64g	4

- **测试所用压力机规格**

  ecs.g6.8xlarge（32 vCPU，128 GB内存）

测试方法

1. 准备压力机ECS

   准备一个ECS（建议配置为32 vCPU 128 GB内存，避免在测试高规格集群时，压力机成为瓶颈），后续操作步骤中涉及的数据准备、运行压测等使用的都是这台ECS机器。

   > **说明** 测试所用ECS需要部署在VPC网络内。请记录该VPC的名称和ID，后续的所有实例都将部署在该VPC内。

2. 准备压测所用PolarDB-X实例

   i. 创建PolarDB-X实例，详细操作步骤请参见 创建实例。

   ii. 在实例中创建一个待压测的数据库（本测试中数据库名为tpcc_1000），详细操作步骤请参见创建数据库。

> **说明** 需保证ECS和PolarDB-X实例在同一个VPC中。

3. 调整实例参数

> **说明** 为了在压测场景下达到最佳性能，需要调整部分PolarDB-X计算层实例参数。

i. 修改参数ENABLE_COROUTINE的值为true，XPROTO_MAX_DN_CONCURRENT的值为4000，详细操作步骤请参见 参数设置。

ii. 通过命令行连接到PolarDB-X实例，在同一会话内执行如下SQL语句，关闭日志记录与CPU采样统计：

```
set ENABLE_SET_GLOBAL = true;
set global RECORD_SQL = false;
set global MPP_METRIC_LEVEL = 0;
set global ENABLE_CPU_PROFILE = false;
```

4. 准备压测数据

i. 准备压测工具

> **说明** BenchmarkSQL默认不支持MySQL协议，需要进行工具改造适配（此处提供了基于开源BenchmarkSQL 5.0改动后的源码以及编译后的jar包）。

下载压测工具包benchmarksql.tar.gz，并解压。

```
tar xzvf benchmarksql.tar.gz
```

ii. 压测配置

在ECS上执行如下命令，修改props.mysql配置文件，并填入对应的PolarDB-X实例连接信息：

```
cd benchmarksql/run
vi props.mysql
```

其中， 配置文件以及主要参数解释如下：

```
db=mysql
driver=com.mysql.jdbc.Driver
conn=jdbc:mysql://{HOST}:{PORT}/tpcc?readOnlyPropagatesToServer=false&rewriteBatchedStatements=true&
failOverReadOnly=false&connectTimeout=3000&socketTimeout=90000&allowMultiQueries=true&clobberStreami
ngResults=true&characterEncoding=utf8&netTimeoutForStreamingResults=0&autoReconnect=true
user={USER}
password={PASSWORD}
warehouses=1000
loadWorkers=100
terminals=128
//To run specified transactions per terminal- runMins must equal zero
runTxnsPerTerminal=0
//To run for specified minutes- runTxnsPerTerminal must equal zero
runMins=5
//Number of total transactions per minute
limitTxnsPerMin=0
//Set to true to run in 4.x compatible mode. Set to false to use the
//entire configured database evenly.
terminalWarehouseFixed=true
//The following five values must add up to 100
//The default percentages of 45, 43, 4, 4 & 4 match the TPC-C spec
newOrderWeight=45
paymentWeight=43
orderStatusWeight=4
deliveryWeight=4
stockLevelWeight=4
// Directory name to create for collecting detailed result data.
// Comment this out to suppress.
resultDirectory=my_result_%tY-%tm-%td_%tH%tM%tS
// osCollectorScript=./misc/os_collector_linux.py
// osCollectorInterval=1
// osCollectorSSHAddr=user@dbhost
// osCollectorDevices=net_eth0 blk_sda
```

- conn：连接串配置，需填入主机名{HOST}、端口号{PORT}
- user：用户名
- password：密码
- warehouses：仓库数
- loadWorkers：导入数据并发数
- terminals：压测并发数
- runMins：压测时间

iii. 导入压测数据

在压力机ECS中执行如下命令导入压测数据：

```
cd benchmarksql/run
nohup ./runDatabaseBuild.sh props.mysql &
```

> **说明** 默认按照100并发导入，总共5亿多条记录，整体导入时间在小时级别，建议通过nohup推到后台运行，避免ssh命令行断开导致导入中断。

iv. 验证数据完整性

通过命令行连接到PolarDB-X实例执行以下SQL，若返回结果集均为空，证明数据完整。

```
select a.* from (Select w_id, w_ytd from bmsql_warehouse) a left join (select d_w_id, sum(d_ytd) as
d_ytd_sum from bmsql_district group by d_w_id) b on a.w_id = b.d_w_id and a.w_ytd = b.d_ytd_sum wher
e b.d_w_id is null;
select a.* from (Select d_w_id, d_id, D_NEXT_O_ID - 1 as d_n_o_id from bmsql_district) a left join (
select o_w_id, o_d_id, max(o_id) as o_id_max from bmsql_oorder group by  o_w_id, o_d_id) b on a.d_w_
id = b.o_w_id and a.d_id = b.o_d_id and a.d_n_o_id = b.o_id_max where b.o_w_id is null;
select a.* from (Select d_w_id, d_id, D_NEXT_O_ID - 1 as d_n_o_id from bmsql_district) a left join (
select no_w_id, no_d_id, max(no_o_id) as no_id_max from bmsql_new_order group by no_w_id, no_d_id) b
on a.d_w_id = b.no_w_id and a.d_id = b.no_d_id and a.d_n_o_id = b.no_id_max where b.no_id_max is nul
l;
select * from (select (count(no_o_id)-(max(no_o_id)-min(no_o_id)+1)) as diff from bmsql_new_order gr
oup by no_w_id, no_d_id) a where diff != 0;
select a.* from (select o_w_id, o_d_id, sum(o_ol_cnt) as o_ol_cnt_cnt from bmsql_oorder  group by o_
w_id, o_d_id) a left join (select ol_w_id, ol_d_id, count(ol_o_id) as ol_o_id_cnt from bmsql_order_l
ine group by ol_w_id, ol_d_id) b on a.o_w_id = b.ol_w_id and a.o_d_id = b.ol_d_id and a.o_ol_cnt_cnt
= b.ol_o_id_cnt where b.ol_w_id is null;
select a.* from (select d_w_id, sum(d_ytd) as d_ytd_sum from bmsql_district group by d_w_id) a left
join (Select w_id, w_ytd from bmsql_warehouse) b on a.d_w_id = b.w_id and a.d_ytd_sum = b.w_ytd wher
e b.w_id is null;
```

5. 执行压测

执行如下命令运行TPC-C测试：

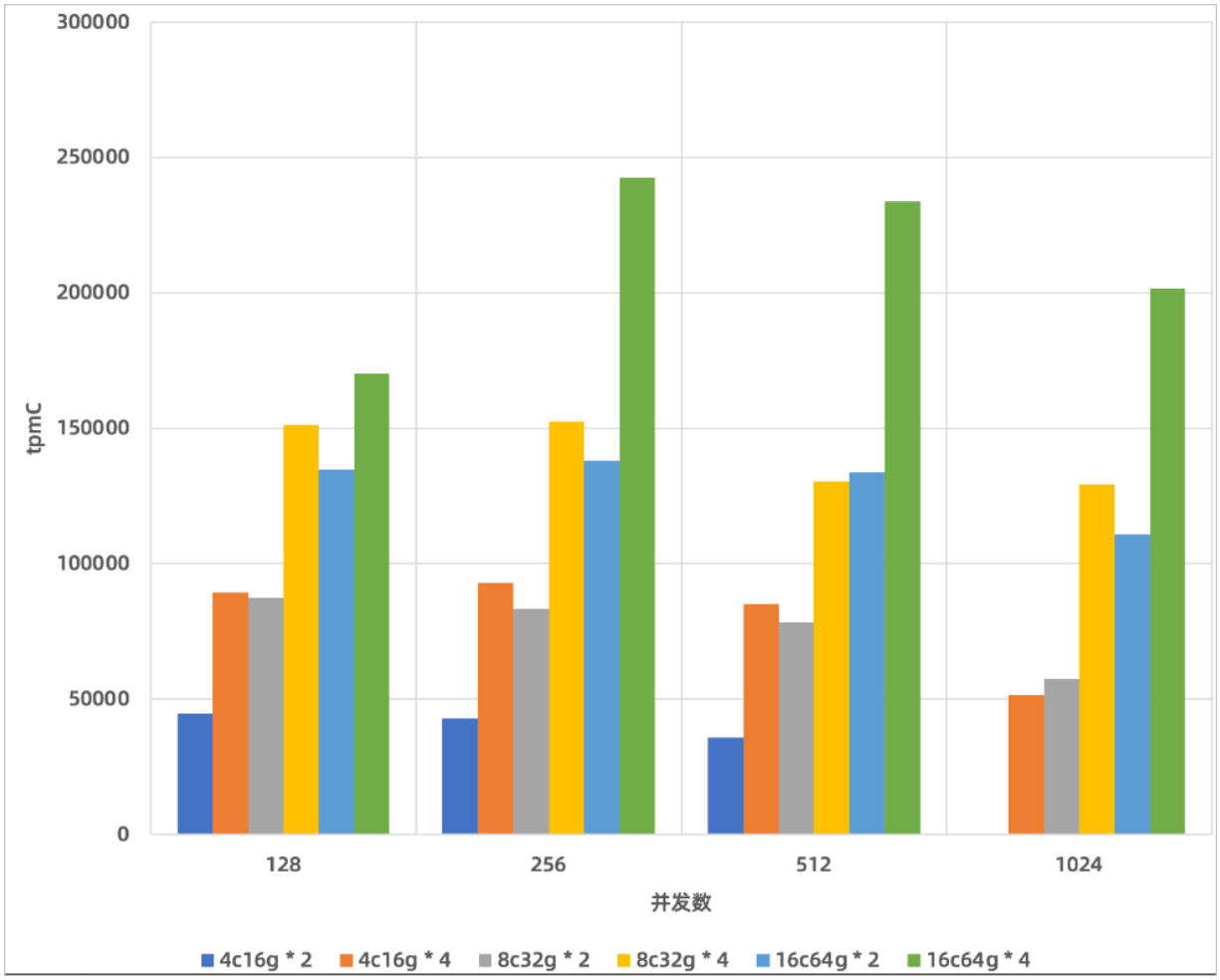
```
cd benchmarksql/run
./runBenchmark.sh props.mysql
```

运行后可以看到实时的tpmC数值，运行结束后会显示平均的tpmC数值。

```
[2022/01/26 22:00:22.995]    Average tpmC: 228425.52    Current tpmC: 235152.00    Memory Usage: 1001MB
/ 3584MB
[2022/01/26 22:00:27.995]    Average tpmC: 228641.71    Current tpmC: 235764.00    Memory Usage: 571MB /
3584MB
[2022/01/26 22:00:32.995]    Average tpmC: 228816.06    Current tpmC: 234744.00    Memory Usage: 101MB /
3584MB
[2022/01/26 22:00:37.995]    Average tpmC: 229017.73    Current tpmC: 236076.00    Memory Usage: 705MB /
3584MB
22:00:38,071 [Thread-132] INFO   jTPCC : Term-00,
22:00:38,071 [Thread-132] INFO   jTPCC : Term-00,
22:00:38,071 [Thread-132] INFO   jTPCC : Term-00, Measured tpmC (NewOrders) = 228974.04
22:00:38,071 [Thread-132] INFO   jTPCC : Term-00, Measured tpmTOTAL = 509473.72
22:00:38,071 [Thread-132] INFO   jTPCC : Term-00, Session Start     = 2022-01-26 21:57:37
22:00:38,072 [Thread-132] INFO   jTPCC : Term-00, Session End       = 2022-01-26 22:00:38
22:00:38,072 [Thread-132] INFO   jTPCC : Term-00, Transaction Count = 1529074
```

测试结果

规格	128并发	256并发	512并发	1024并发
4c16g * 2	44555.79	42998.57	35726.43	/
4c16g * 4	89365.36	93000.84	84905.66	51487.88
8c32g * 2	87574.27	83452.71	78438.62	57450.78
8c32g * 4	151265.13	152423.37	130286.38	129271.17
16c64g * 2	134691.53	138016.43	133600.19	110967.42
16c64g * 4	170364.78	242448.61	233844.02	201670.64



5.2. TPC-H测试

本文详细介绍了PolarDB-X的TPC-H测试设计、测试过程和测试结果。

背景信息

TPC-H是业界常用的一套Benchmark，由TPC委员会制定发布，用于评测数据库的分析型查询能力。TPC-H查询包含8张数据表、22条复杂的SQL查询，大多数查询包含若干表Join、子查询和Group-by聚合等。

> **说明** 本文中TPC-H的实现基于TPC-H的基准测试，并不能与已发布的TPC-H基准测试结果相比较，本文中的测试并不符合TPC-H基准测试的所有要求。

测试条件

- **测试数据量**

  测试基于100G数据量（Scalar Factor = 100），其中主要表数据量如下：

  - LINEITEM表约6亿行
  - ORDERS表1.5亿行
  - PART_SUPP表8000万行

- **测试所用实例规格**

节点规格	节点数	数据集大小
4c32g	6	100 GB

节点规格	节点数	数据集大小
8c64g	6	100 GB
16c128g	6	100 GB

- **测试所用压力机规格**

  ecs.g7.4xlarge（16 vCPU，64 GB内存，存储盘大于200 GB）

测试方法

1. 准备压力机ECS

   准备一个ECS（存储盘要求大于200 GB，需要存放工具生成的csv格式数据集），后续操作步骤中涉及的数据准备、运行压测等使用的都是这台ECS机器。

   > **说明** 测试所用ECS需要部署在VPC网络内。请记录该VPC的名称和ID，后续的所有实例都将部署在该VPC内。

2. 准备压测所用PolarDB-X实例

   i. 创建PolarDB-X实例，详细操作步骤请参见创建实例。

   ii. 在实例中创建一个待压测的数据库（本测试中数据库名为tpch_100g），详细操作步骤请参见创建数据库。

   iii. 在数据库tpch_100g中创建对应的表，方法如下：

```
CREATE TABLE `customer` (
  `c_custkey` int(11) NOT NULL,
  `c_name` varchar(25) NOT NULL,
  `c_address` varchar(40) NOT NULL,
  `c_nationkey` int(11) NOT NULL,
  `c_phone` varchar(15) NOT NULL,
  `c_acctbal` decimal(15,2) NOT NULL,
  `c_mktsegment` varchar(10) NOT NULL,
  `c_comment` varchar(117) NOT NULL,
  PRIMARY KEY (`c_custkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by hash(`c_custkey`) tbpartition by hash(`c_custk
ey`) tbpartitions 4;
CREATE TABLE `lineitem` (
  `l_orderkey` bigint(20) NOT NULL,
  `l_partkey` int(11) NOT NULL,
  `l_suppkey` int(11) NOT NULL,
  `l_linenumber` bigint(20) NOT NULL,
  `l_quantity` decimal(15,2) NOT NULL,
  `l_extendedprice` decimal(15,2) NOT NULL,
  `l_discount` decimal(15,2) NOT NULL,
  `l_tax` decimal(15,2) NOT NULL,
  `l_returnflag` varchar(1) NOT NULL,
  `l_linestatus` varchar(1) NOT NULL,
  `l_shipdate` date NOT NULL,
  `l_commitdate` date NOT NULL,
  `l_receiptdate` date NOT NULL,
  `l_shipinstruct` varchar(25) NOT NULL,
  `l_shipmode` varchar(10) NOT NULL,
  `l_comment` varchar(44) NOT NULL,
  KEY `IDX_LINEITEM_PARTKEY` (`l_partkey`),
  PRIMARY KEY (`l_orderkey`,`l_linenumber`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by RIGHT_SHIFT(`l_orderkey`,6) tbpartition by RIG
HT_SHIFT(`l_orderkey`,6) tbpartitions 4;
CREATE TABLE `orders` (
  `o_orderkey` bigint(20) NOT NULL,
  `o_custkey` int(11) NOT NULL,
  `o_orderstatus` varchar(1) NOT NULL,
  `o_totalprice` decimal(15,2) NOT NULL,
  `o_orderdate` date NOT NULL,
```

```
  `o_orderpriority` varchar(15) NOT NULL,
  `o_clerk` varchar(15) NOT NULL,
  `o_shippriority` bigint(20) NOT NULL,
  `o_comment` varchar(79) NOT NULL,
  PRIMARY KEY (`O_ORDERKEY`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by RIGHT_SHIFT(`O_ORDERKEY`,6) tbpartition by RIG
HT_SHIFT(`O_ORDERKEY`,6) tbpartitions 4;
CREATE TABLE `part` (
  `p_partkey` int(11) NOT NULL,
  `p_name` varchar(55) NOT NULL,
  `p_mfgr` varchar(25) NOT NULL,
  `p_brand` varchar(10) NOT NULL,
  `p_type` varchar(25) NOT NULL,
  `p_size` int(11) NOT NULL,
  `p_container` varchar(10) NOT NULL,
  `p_retailprice` decimal(15,2) NOT NULL,
  `p_comment` varchar(23) NOT NULL,
  PRIMARY KEY (`p_partkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by hash(`p_partkey`) tbpartition by hash(`p_partk
ey`) tbpartitions 4;
CREATE TABLE `partsupp` (
  `ps_partkey` int(11) NOT NULL,
  `ps_suppkey` int(11) NOT NULL,
  `ps_availqty` int(11) NOT NULL,
  `ps_supplycost` decimal(15,2) NOT NULL,
  `ps_comment` varchar(199) NOT NULL,
  KEY `IDX_PARTSUPP_SUPPKEY` (`PS_SUPPKEY`),
  PRIMARY KEY (`ps_partkey`,`ps_suppkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by hash(`ps_partkey`) tbpartition by hash(`ps_par
tkey`) tbpartitions 4;
CREATE TABLE `supplier` (
  `s_suppkey` int(11) NOT NULL,
  `s_name` varchar(25) NOT NULL,
  `s_address` varchar(40) NOT NULL,
  `s_nationkey` int(11) NOT NULL,
  `s_phone` varchar(15) NOT NULL,
  `s_acctbal` decimal(15,2) NOT NULL,
  `s_comment` varchar(101) NOT NULL,
  PRIMARY KEY (`s_suppkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by hash(`s_suppkey`) tbpartition by hash(`s_suppk
ey`) tbpartitions 4;
CREATE TABLE `nation` (
  `n_nationkey` int(11) NOT NULL,
  `n_name` varchar(25) NOT NULL,
  `n_regionkey` int(11) NOT NULL,
  `n_comment` varchar(152) DEFAULT NULL,
  PRIMARY KEY (`n_nationkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 broadcast;
CREATE TABLE `region` (
  `r_regionkey` int(11) NOT NULL,
  `r_name` varchar(25) NOT NULL,
  `r_comment` varchar(152) DEFAULT NULL,
  PRIMARY KEY (`r_regionkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 broadcast;
```

说明 需保证ECS和PolarDB-X实例在同一个VPC中。

3. 调整实例参数

说明 为了在压测场景下达到最佳性能，需要调整PolarDB-X计算层实例参数。

i. 修改参数XPROTO_MAX_DN_CONCURRENT的值为4000，详细操作步骤请参见参数设置。

ii. 通过命令行连接到PolarDB-X实例，在同一会话内执行如下SQL语句，关闭日志记录与CPU采样统计：

```
set ENABLE_SET_GLOBAL = true;
set global RECORD_SQL = false;
set global MPP_METRIC_LEVEL = 0;
set global ENABLE_CPU_PROFILE = false;
```

4. 数据准备

i. 下载脚本tpchData.tar.gz至压力机ECS上，并解压：

```
tar xzvf tpchData.tar.gz
```

修改param.conf配置文件，填入PolarDB-X实例的连接信息：

```
cd tpchData/
vi param.conf
```

```
#!/bin/bash
### remote generating directory
export remoteGenDir=./
### target path
export targetPath=../tpch/tpchRaw
### cores per worker, default value is 1
export coresPerWorker=`cat /proc/cpuinfo| grep "processor"| wc -l`
### threads per worker, default value is 1
export threadsPerWorker=`cat /proc/cpuinfo| grep "processor"| wc -l`
#export threadsPerWorker=1
export hint=""
export insertMysql="mysql -h{HOST} -P{PORT} -u{USER} -p{PASSWORD} -Ac --local-infile tpch_100g -e"
```

具体填入的值包括：

- {HOST}：主机名
- {PORT}：端口号
- {USER}：用户名
- {PASSWORD}：密码

如果希望更高效地生成数据，可调大脚本中threadsPerWorker的值。

ii. 执行脚本，多进程生成100 GB的数据：

```
cd datagen
sh generateTPCH.sh 100
```

可以在tpch/tpchRaw/SF100/目录下查看到生成的数据：

```
ls ../tpch/tpchRaw/SF100/
customer  lineitem  nation  orders  part  partsupp  region  supplier
```

iii. 导入数据到PolarDB-X实例：

```
cd ../loadTpch
sh loadTpch.sh 100
```

iv. 校验数据完整性

通过命令行连接到PolarDB-X实例，查询每张表的数据量是否符合预期：

```
MySQL [tpch_100g]> select (select count(*) from customer) as customer_cnt,
 (select count(*)  from lineitem) as lineitem_cnt,
 (select count(*)  from nation) as nation_cnt,
 (select count(*)  from orders) as order_cnt,
 (select count(*) from part) as part_cnt,
 (select count(*) from partsupp) as partsupp_cnt,
 (select count(*) from region) as region_cnt,
 (select count(*) from supplier) as supplier_cnt;
+--------------+--------------+------------+-----------+----------+--------------+------------+-----
---------+
| customer_cnt | lineitem_cnt | nation_cnt | order_cnt | part_cnt | partsupp_cnt | region_cnt | supp
lier_cnt |
+--------------+--------------+------------+-----------+----------+--------------+------------+-----
---------+
|     15000000 |    600037902 |         25 | 150000000 | 20000000 |     80000000 |          5 |
1000000 |
+--------------+--------------+------------+-----------+----------+--------------+------------+-----
---------+
```

v. 采集统计信息

通过命令行连接到PolarDB-X实例，执行analyze table收集表的统计信息：

```
analyze table customer;
analyze table lineitem;
analyze table nation;
analyze table orders;
analyze table part;
analyze table partsupp;
analyze table region;
analyze table supplier;
```

5. 进行测试

i. 下载测试脚本tpch-queries.tar.gz并解压：

```
tar xzvf tpch-queries.tar.gz
```

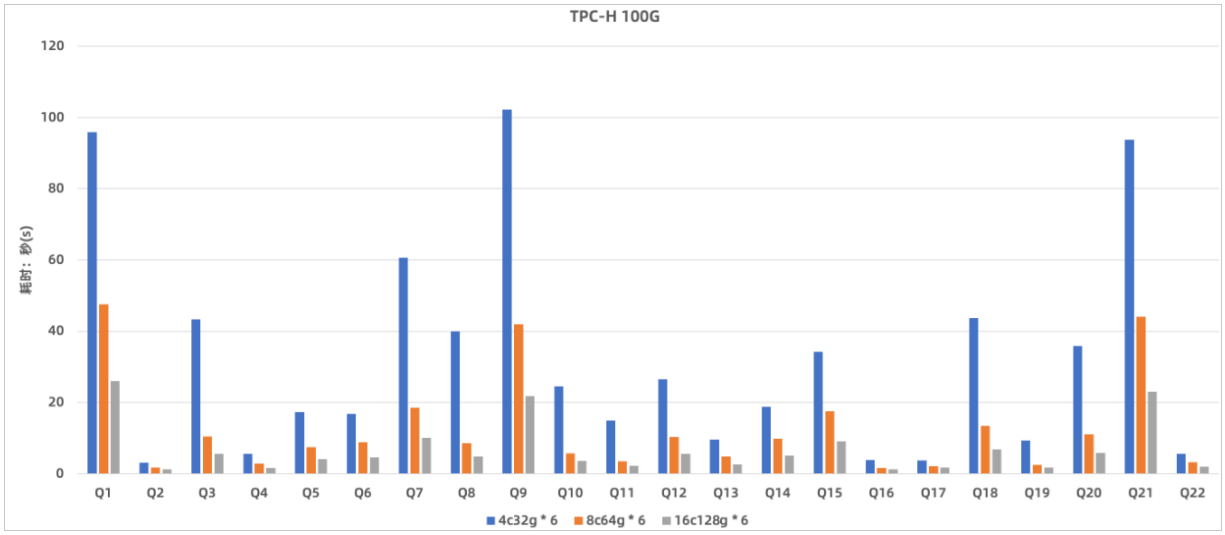
ii. 运行脚本，执行查询并计时：

```
cd tpch-queries
'time' -f "%e" sh all_query.sh {HOST} {USER} {PASSWORD} {DB} {PORT}
```

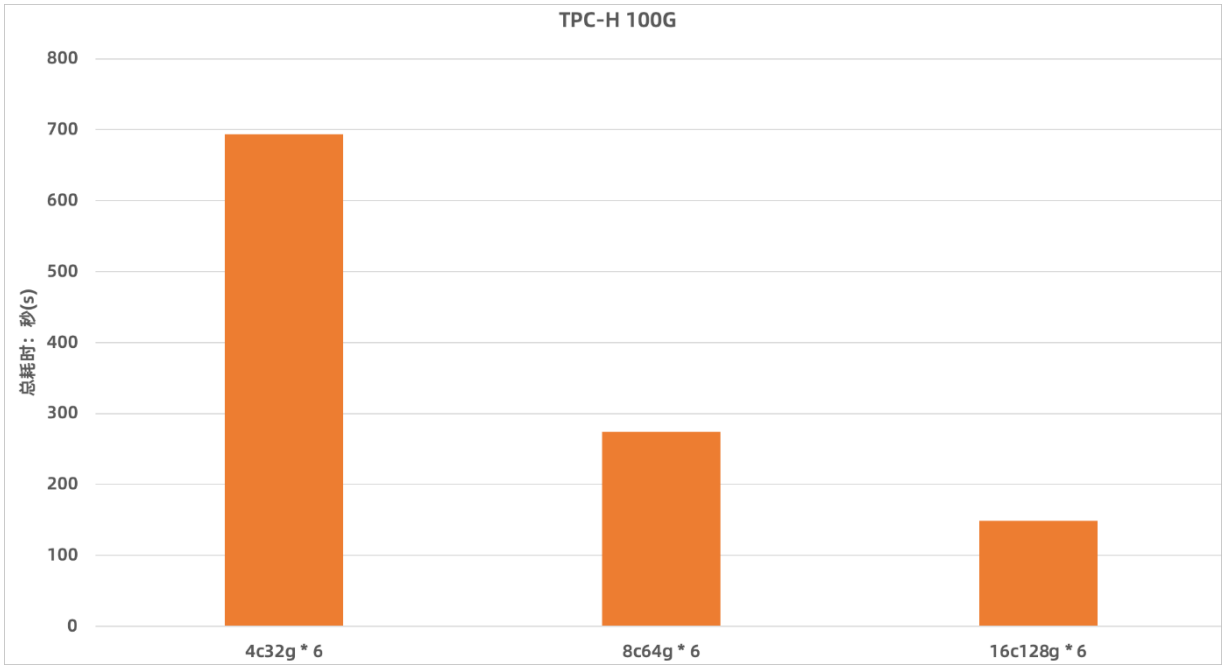
测试结果

以下结果的时间单位为：秒（s）

各个查询在不同规格下的耗时



不同规格下的查询总耗时



Query	4c32g * 6（单位：秒）	8c64g * 6（单位：秒）	16c128g * 6（单位：秒）
Q1	95.88	47.5	26.06
Q2	3.12	1.71	1.29
Q3	43.28	10.48	5.63
Q4	5.54	2.86	1.61
Q5	17.28	7.49	4.09
Q6	16.8	8.77	4.62
Q7	60.64	18.56	10.09
Q8	39.9	8.63	4.87
Q9	102.24	41.89	21.75

Query	4c32g * 6（单位：秒）	8c64g * 6（单位：秒）	16c128g * 6（单位：秒）
Q10	24.51	5.65	3.59
Q11	14.89	3.5	2.21
Q12	26.51	10.31	5.56
Q13	9.6	4.78	2.54
Q14	18.84	9.84	5.14
Q15	34.21	17.53	9.12
Q16	3.79	1.62	1.2
Q17	3.73	2.06	1.69
Q18	43.68	13.39	6.82
Q19	9.28	2.43	1.77
Q20	35.8	11.05	5.86
Q21	93.81	44.07	23.05
Q22	5.6	3.24	1.94
总时间	694.04	273.86	148.29

5.3. Sysbench测试

本文详细介绍了PolarDB-X的Sysbench测试设计、测试过程和测试结果。

背景信息

Sysbench是一款开源的、模块化的、跨平台的多线程性能测试工具，可以执行数据库在CPU、内存、线程、IO等方面的性能测试。本文将验证PolarDB-X在Sysbench多种场景中的性能表现。

测试条件

- **测试数据量**

  Sysbench数据包含16张表，每张表1千万的数据量。

- **测试所用实例规格**

节点规格	节点数
2c8g	2
4c16g	2
4c16g	4
8c32g	2
8c32g	4
16c64g	2
16c64g	4

- **测试所用压力机规格**

ecs.g7ne.8xlarge（32 vCPU，128 GB内存）

测试方法

1. 准备压力机ECS

   需准备一个ECS（建议配置为32 vCPU 128 GB内存，避免在测试高规格集群时，压力机成为瓶颈），后续操作步骤中涉及的数据准备、运行压测等使用的都是这台ECS机器。

   > **说明** 建议将测试所用ECS部署在VPC网络内。请记住该VPC的名称和ID，后续的所有实例都将部署在该VPC内。

2. 准备压测所用PolarDB-X实例

   i. 创建PolarDB-X实例，详细操作步骤请参见创建实例。

   ii. 在实例中创建一个待压测的数据库（本测试中数据库名为sbtest），详细操作步骤请参见创建数据库。

3. 调整实例参数

   > **说明** 为了在压测场景下达到最佳性能，需要调整部分PolarDB-X计算层实例参数。

   i. 修改参数ENABLE_COROUTINE的值为true，XPROTO_MAX_DN_CONCURRENT的值为4000，详细操作步骤请参见参数设置。

   ii. 通过命令行连接到PolarDB-X实例，在同一会话内执行如下SQL语句，关闭日志记录与CPU采样统计：

   ```
   set ENABLE_SET_GLOBAL = true;
   set global RECORD_SQL = false;
   set global MPP_METRIC_LEVEL = 0;
   set global ENABLE_CPU_PROFILE = false;
   ```

4. 压测数据准备

   i. 下载Sysbench压测工具包sysbench.tar.gz，并解压。

   ```
   tar xzvf sysbench.tar.gz
   cd sysbench/
   ```

   安装编译所需依赖库：

   ```
   yum -y install make automake libtool pkgconfig libaio-devel mysql-devel
   ./autogen.sh
   ./configure
   make -j
   make install
   ```

   执行命令 `sysbench --version` ，返回 `sysbench 1.1.0` 证明压测工具包安装成功。

ii. 准备压测配置

创建配置文件sysb.conf，将PolarDB-X连接信息填入配置文件，其中配置文件以及主要参数的解读如下：

```
mysql-host='{HOST}'
mysql-port='{PORT}'
mysql-user='{USER}'
mysql-password='{PASSWORD}'
mysql-db='sbtest'
db-driver='mysql'
percentile='95'
histogram='on'
report-interval='1'
time='60'
rand-type='uniform'
```

参数说明：

- percentile：响应时间采样的百分位；
- histogram：是否展示响应时间分布直方图；
- report-interval：显示实时结果的时间间隔，单位为秒；
- time：压测时长，单位为秒；
- rand-type：随机数的分布模式。

iii. 导入压测数据

```
sysbench --config-file='sysb.conf' --create-table-options='dbpartition by hash(`id`) tbpartition by hash(id) tbpartitions 2'  --tables='16' --threads='16' --table-size='10000000' oltp_point_select prepare
```

参数说明：

- config-file：指定通用配置文件；
- create-table-options：指定建表模式；
- tables：表的数量；
- table-size：单张表的数据量；
- threads：导入时的并发数；
- auto_inc：是否使用auto_increment。

5. 执行压测

以下列举了Sysbench 6种不同场景测试的命令。

> **说明** 命令涉及的相关参数说明如下：
> - db-ps-mode：是否开启prepare模式；
> - skip-trx：是否不开启事务，on表示不开启事务，off表示开启；
> - mysql-ignore-errors：忽略如锁冲突等原因产生的MySQL错误代码；
> - range-size：指定range查询的范围大小。

- 点查：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all' --tables='16' --table-size='10000000' --threads={并发数} oltp_point_select run
```

- 只读：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all' --tables='16' --table-size='10000000' --range-size=5 --threads={并发数} oltp_read_only run
```

- 读写：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all'
--tables='16' --table-size='10000000' --range-size=5 --threads={并发数} oltp_read_write run
```

- 按索引更新：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all'
--tables='16' --table-size='10000000' --threads={并发数} oltp_update_index run
```

- 非索引更新：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all'
--tables='16' --table-size='10000000' --threads={并发数} oltp_update_non_index run
```

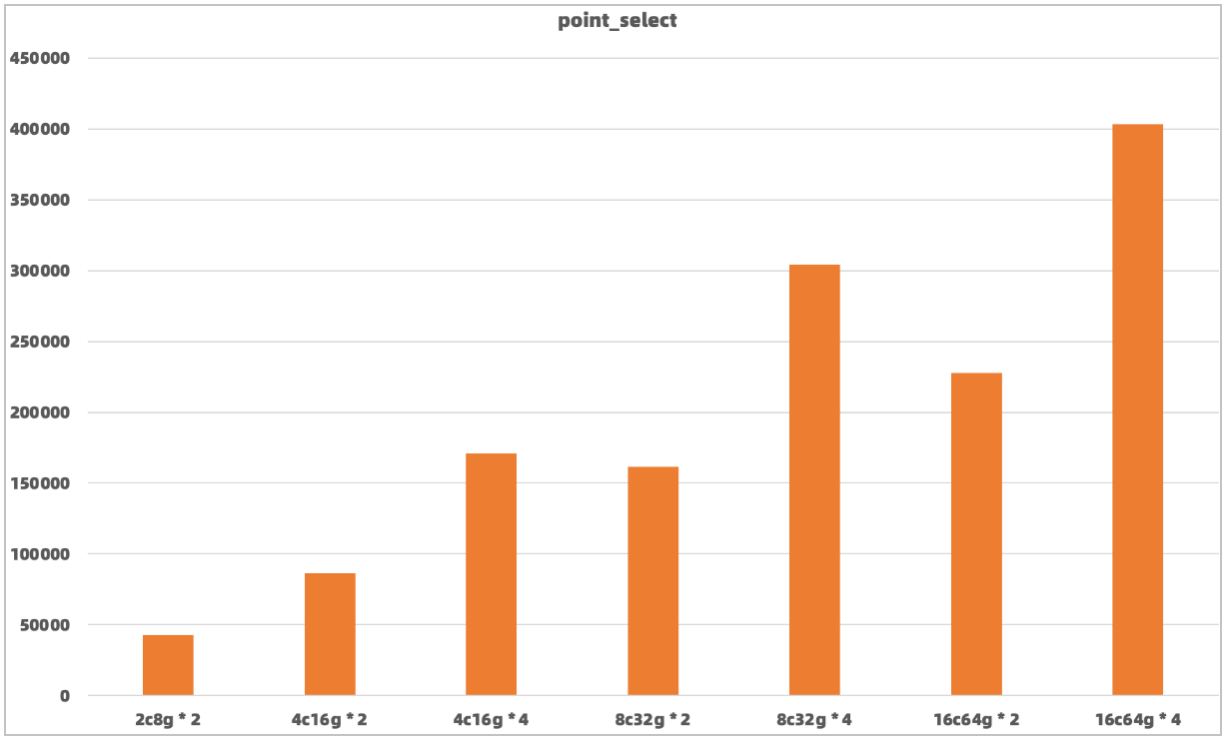
- 写入：

```
sysbench --config-file='sysb.conf' --db-ps-mode='disable' --skip-trx='on' --mysql-ignore-errors='all'
--tables='16' --table-size='10000000' --threads={并发数} oltp_write_only run
```

测试结果

以下结果均以QPS作为衡量性能的标准。

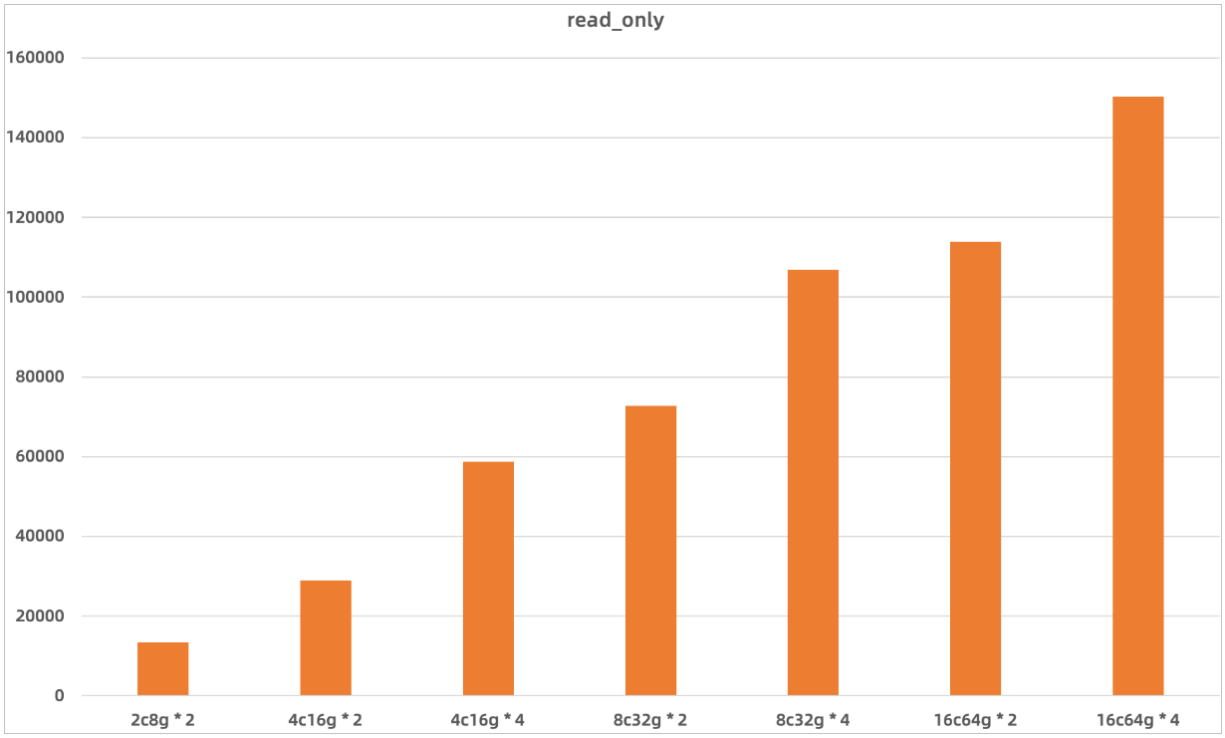
点查场景结果 (point_select)



实例规格	并发数	QPS
2c8g * 2	512	42753.61
4c16g * 2	1024	86413.17
4c16g * 4	1024	170834.72
8c32g * 2	1024	161473.31
8c32g * 4	2048	304003.49
16c64g * 2	2048	227869.71

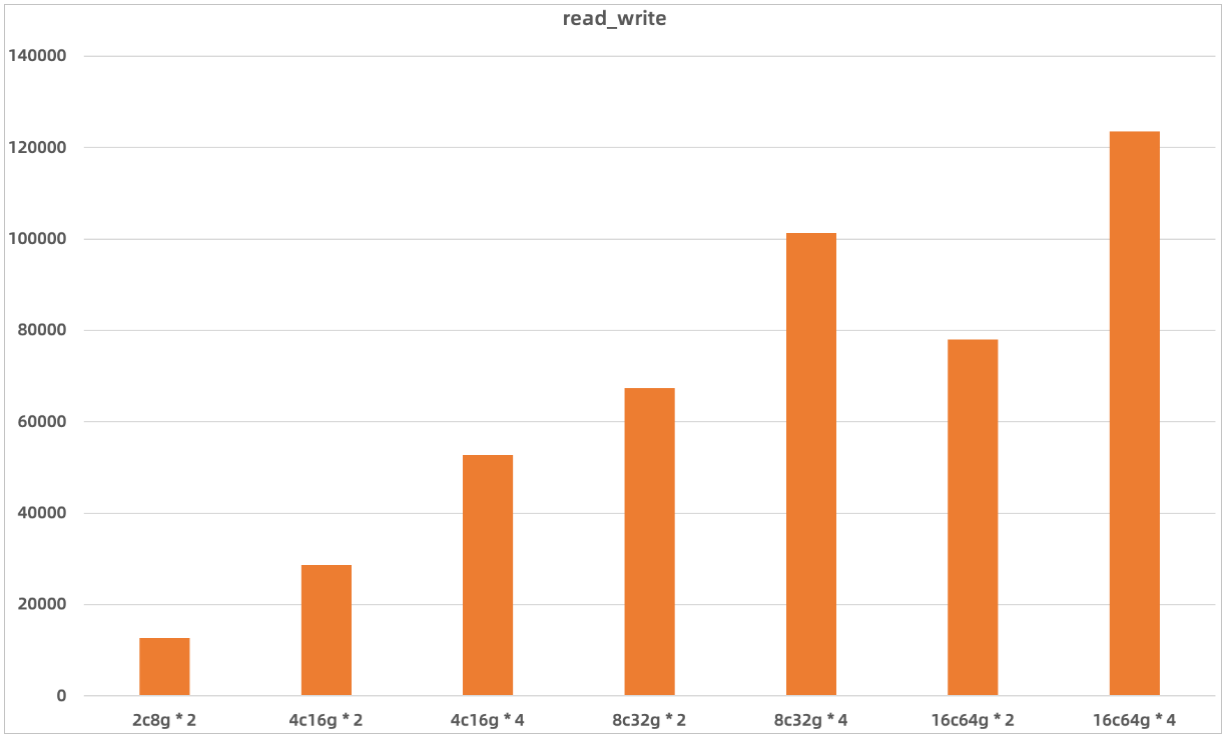
实例规格	并发数	QPS
16c64g * 4	3096	403384.56

只读场景结果 (read_only)



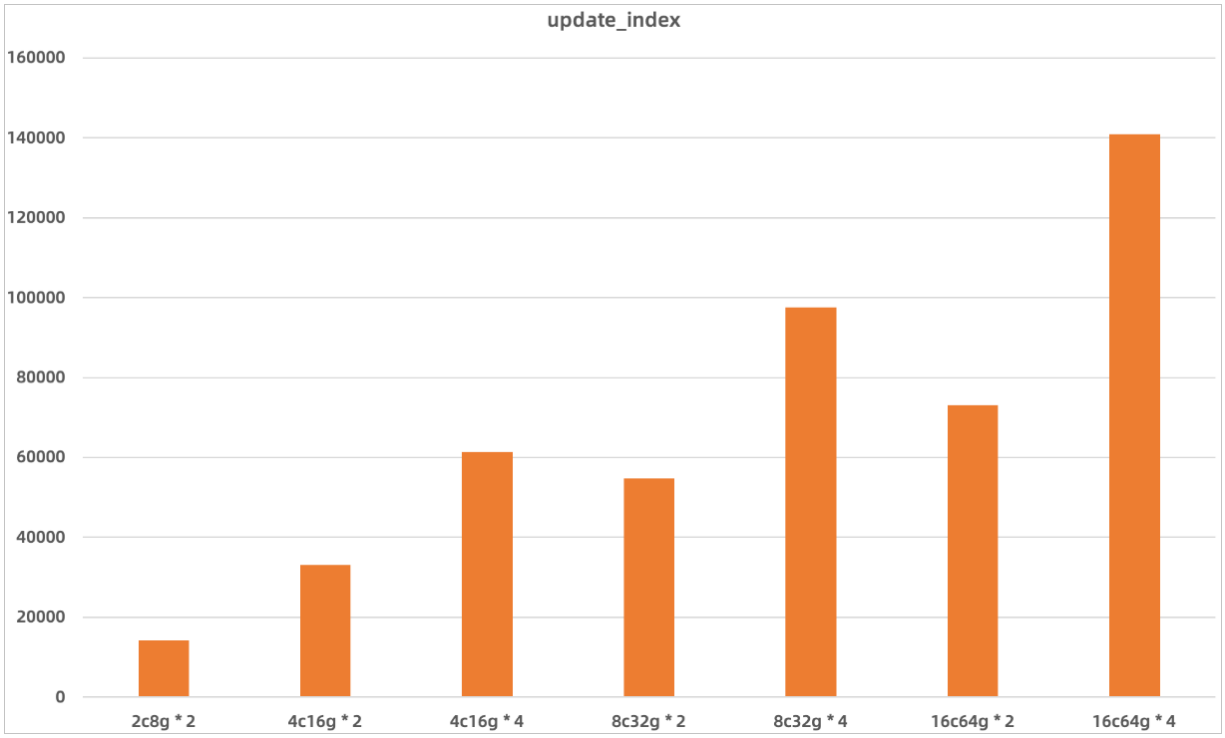
实例规格	并发数	QPS
2c8g * 2	64	13303.04
4c16g * 2	256	28871.19
4c16g * 4	256	58643.49
8c32g * 2	256	72772.71
8c32g * 4	512	106942.17
16c64g * 2	256	113922.87
16c64g * 4	512	150396.33

读写场景结果 (read_write)



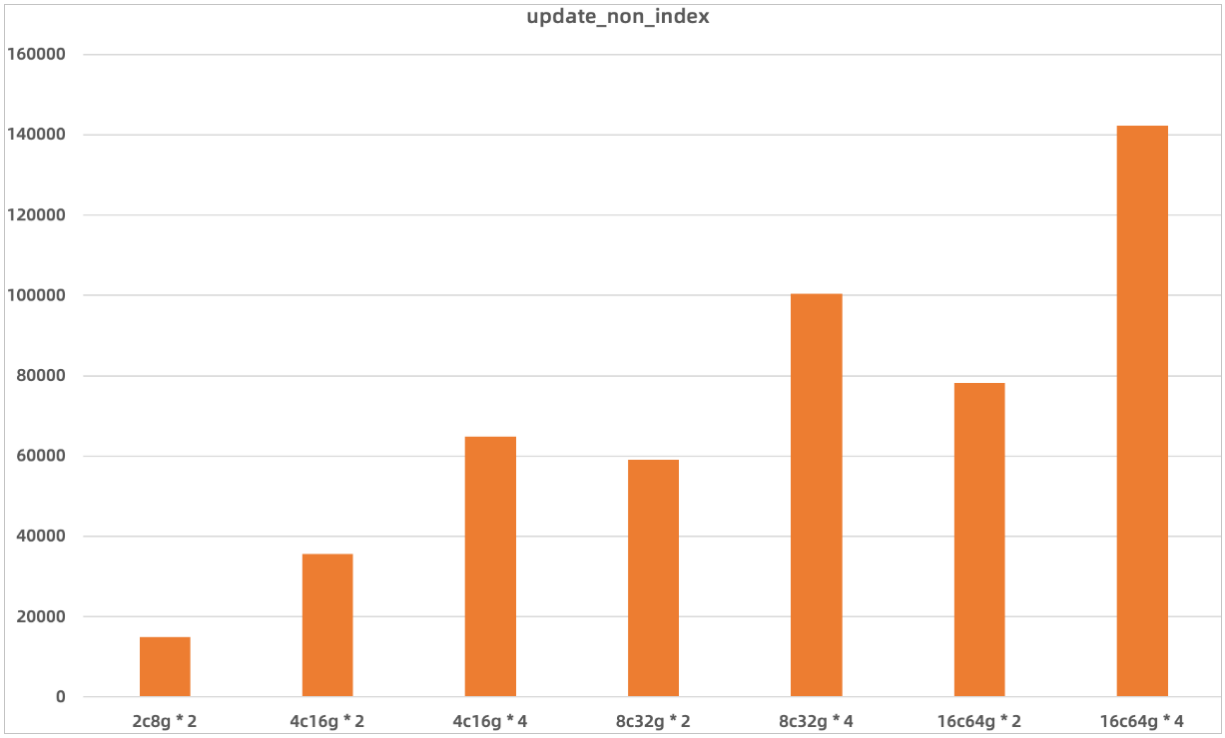
实例规格	并发数	QPS
2c8g * 2	64	12687.14
4c16g * 2	128	28658.71
4c16g * 4	256	52726.33
8c32g * 2	128	67373.28
8c32g * 4	128	101273.77
16c64g * 2	256	77931.34
16c64g * 4	512	123463.22

按索引更新场景结果 (update_index)



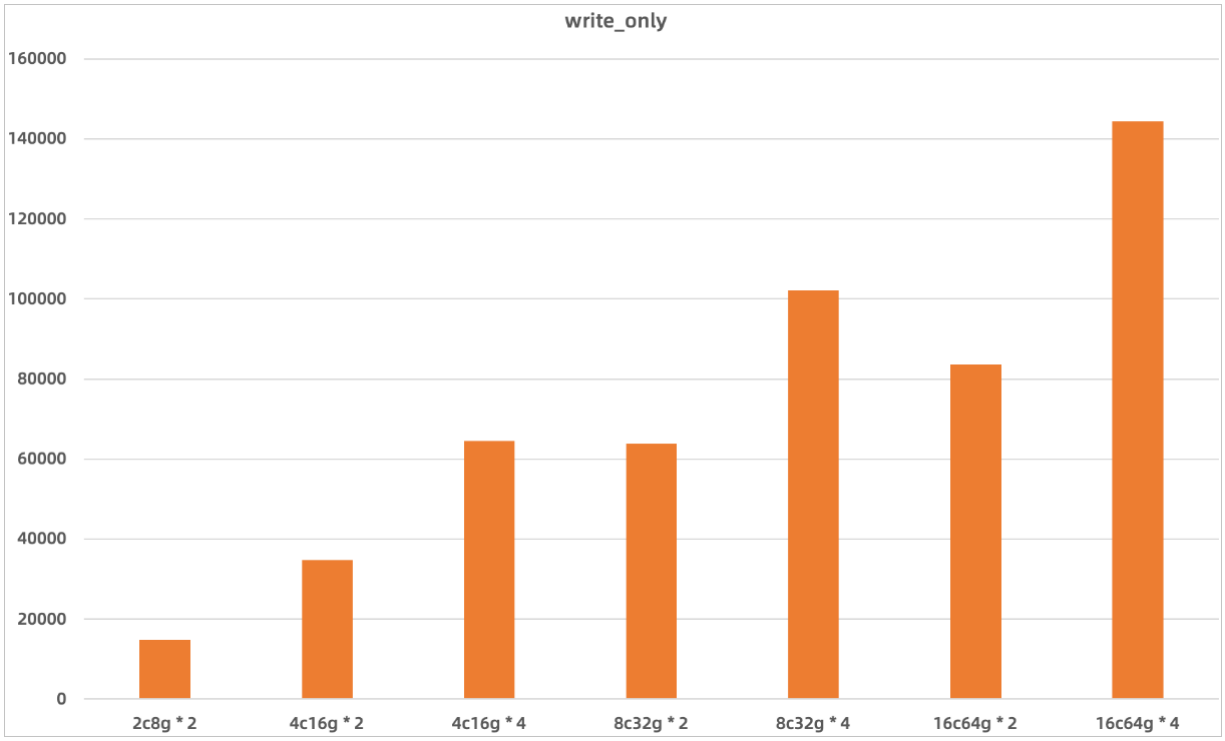
实例规格	并发数	QPS
2c8g * 2	128	14164.71
4c16g * 2	256	33037.09
4c16g * 4	256	61381.59
8c32g * 2	256	54728.10
8c32g * 4	256	97620.78
16c64g * 2	512	73120.56
16c64g * 4	512	140905.00

非索引更新场景结果 (update_non_index)



实例规格	并发数	QPS
2c8g * 2	128	14897.04
4c16g * 2	256	35605.90
4c16g * 4	256	64814.66
8c32g * 2	256	59162.14
8c32g * 4	256	100467.07
16c64g * 2	512	78192.59
16c64g * 4	512	142244.89

写入场景结果 (write_only)



实例规格	并发数	QPS
2c8g * 2	256	14763.85
4c16g * 2	256	34776.46
4c16g * 4	256	64548.93
8c32g * 2	256	63807.99
8c32g * 4	256	102137.14
16c64g * 2	256	83583.16
16c64g * 4	512	144374.46

6.快速入门

6.1. PolarDB-X首次使用流程

本文将帮助您快速了解如何使用PolarDB-X创建实例并使用该实例。

前提条件

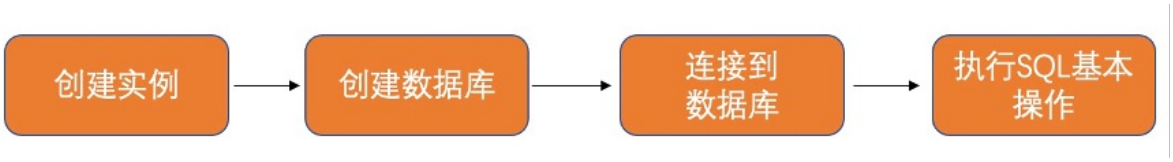
您需要先注册和登录阿里云账号，详情请参见注册和登录阿里云账号。

使用流程

如果您是首次使用PolarDB-X，建议您先阅读以下部分：

- 简介：概述了PolarDB-X的产品概念、产品历史及适用场景等内容。
- 定价：介绍了PolarDB-X的产品定价、计费方式等信息。

通常，从购买PolarDB-X实例到使用该实例，您需要完成下面几个步骤。



1. 创建实例
2. 创建数据库
3. 连接到数据库
4. 执行SQL基本操作

6.2. 创建实例

本文主要介绍如何创建PolarDB-X实例。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 单击页面右上角**创建新实例**。
5. 在购买页面，设置如下参数。

选项	说明
商品类型	您可以选择商品的付费类型为**包年包月**或**按量付费**。存储资源包的购买请参见存储资源包。 **说明** ○ 包年包月为预付费模式，即在新建集群时需要支付费用。适合长期需求，价格比按量付费更实惠。 ○ 按量付费为后付费模式，即按小时扣费。适合短期需求，用完可以立即释放集群，节省费用。 ○ 若您需要创建按量付费的实例，请确保您的阿里云账户余额大于等于100元。

选项	说明
地域	选择实例所在的地域。
部署拓扑	◦ 单可用区：仅需选择主节点可用区，备节点和日志节点将自动部署在同一个可用区。 ◦ 三可用区：选择三可用区，CN、DN、GMS、CDC的主节点将部署于相同的可用区，该可用区称为主可用区。备节点和日志节点将部署于其他不同的可用区。三可用区部署形态下的实例相比单可用区实例具备更高的容灾能力，可以抵御机房级别的故障。 **说明** 主节点、备节点、日志节点的可用区需要互不相同。
主节点可用区	选择主节点所在的可用区。
备节点可用区	选择备节点所在的可用区，仅三可用区部署时生效。
日志节点可用区	选择日志节点所在的可用区，仅三可用区部署时生效。
网络类型	固定为**专有网络**，不可变更。 **说明** 配置**专有网络**前，您需要提前准备相应地域和可用区内的专有网络和虚拟交换机，详情请参见创建和管理专有网络。
专有网络	选择已有VPC，也可单击，在控制台新增VPC。
虚拟交换机	选择虚拟交换机，也可单击，在控制台新增虚拟交换机。
资源组	选择已创建的资源组或者新建资源组。在单个云账号下将一组相关资源进行统一管理的容器即为资源组，一个资源只能归属于一个资源组。
引擎版本	版本固定为**MySQL 5.7**，不可变更。
规格系列	可选择独享规格或通用规格，规格介绍请参见规格说明。
节点规格	PolarDB-X按照节点规格 x 节点个数的方式售卖，您可以通过选择规格配置实例的物理规格CPU核数和内存大小，详情请参见规格说明。 **说明** 计算层节点和存储层节点都使用相同的节点规格。
节点个数	选择目标规格节点的节点个数。 **说明** 节点个数取值范围为2~99。
存储类型	固定为**本地SSD盘**，不可变更。
存储费用	每个节点存储空间默认为3 TB，您购买时无需选择容量，根据实际使用按小时计费。
购买数量	选择需要购买的实例数量。
购买时长	选择实例购买的时长。 **说明** 仅当商品类型为**包年包月**时支持该配置。

6. 选中服务协议，单击**立即购买**。
7. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。
8. 支付成功后，服务开通需要10~15分钟，请您耐心等待。之后您可以返回控制台的**实例列表**页，查看新创建的实例。

6.3. 创建数据库

本文介绍如何在PolarDB-X实例上创建数据库。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 数据库管理**。
6. 在**数据库管理**页面左上角，单击**创建数据库**。
7. 在弹出的对话框中，设置如下参数。

参数	说明
数据库（DB）名称	填写数据库名称。 **说明** 数据库名称需满足如下要求： ○ 由小写字母、数字、下划线（_）或中划线（-）组成，长度需为2~16个字符。 ○ 需以字母开头，以字母或数字结尾。 ○ 不能和已有的数据库名称重复。
支持字符集	可以选择的字符集为**utf8**、**gbk**、**latin1**或**utf8mb4**。
授权账号	选中需要访问本数据库的目标账号。 **说明** ○ 可选的授权账号全部为普通账号，因为高权限账号拥有所有数据库的所有权限，不需要授权。 ○ 若您还未创建任何账号，可以单击**创建新账号**进行创建，具体操作步骤请参见创建账号。
账号权限	选择目标账号需要的权限，可选权限范围为**读写**、**只读**、**仅DDL**和**仅DML**。
备注说明	备注该数据库的相关信息，便于后续数据库管理。该参数为非必填项，最多支持256个字符。

8. 单击**确定**。

6.4. 连接到数据库

本文将介绍如何通过数据管理DMS连接到PolarDB-X。PolarDB-X还支持通过SQL命令行、第三方客户端（MySQL WorkBench、SQLyog、Sequel Pro、Navicat for MySQL）以及符合MySQL官方交互协议的第三方程序代码进行连接。

背景信息

数据管理DMS是一种集数据管理、结构管理、用户授权、安全审计、数据趋势、数据追踪、BI图表、性能与优化和服务器管理于一体的数据管理服务。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 数据库管理**。
6. 在页面右上角，单击**登录数据库**。
7. 在弹出的对话框中，设置如下参数。

参数	说明
数据库类型	默认为**PolarDB-X（原DRDS）**，无需选择。
实例地区	默认为当前实例所在地域，无需选择。 **说明** 若您需要登录其他地域下的PolarDB-X实例，从下拉列表中选择目标实例的所在地域即可。
实例ID	默认为当前实例的实例ID，无需选择。 **说明** 若您需要登录其他PolarDB-X实例，从下拉列表中选择目标实例ID即可。
数据库账号	数据库的授权账号名称。
数据库密码	数据库账号名对应的密码。 **说明** 您可以选中**记住密码**，方便之后再次登录时，无需输入对应的数据库账号和密码即可自动登录。

说明

- 首次通过DMS登录时，管控模式默认为**自由操作**。登录成功后，您还可以通过编辑实例功能来修改管控模式。更多信息，请参见编辑实例和管控模式。
- 配置完登录参数后，您可以单击左下角**测试连接**。如果测试连接失败，请按照报错提示检查录入的集群信息，如账号或密码是否正确。
- 系统会自动尝试往云数据库的白名单中添加DMS的服务器访问地址，若自动添加失败请手动添加。

8. 单击**登录**。

后续步骤

连接到数据库之后，您可以执行一些SQL的基础操作。请参见执行SQL基本操作。

6.5. 执行SQL基本操作

本文介绍如何在PolarDB-X中执行常用的SQL语句。

- 查看数据库
- 创建、查看和删除表
- 增加列、删除列、修改列的数据类型
- 创建、查看和删除本地索引
- 增删改查表数据

查看数据库

查看数据库：

```
SHOW DATABASES;
```

创建、查看和删除表

- 创建表。
  - 创建单表：

    ```
    CREATE TABLE single_tbl(
     id int,
     name varchar(30),
     primary key(id)
    );
    ```

  - 创建一个对目标列（如id列）使用HASH函数进行拆分的表：

    ```
    CREATE TABLE multi_db_single_tbl(
      id int auto_increment,
      name varchar(30),
      primary key(id)
    ) dbpartition by hash(id);
    ```

- 查看建表语句。
  - 查看单表建表语句：

    ```
    SHOW CREATE TABLE single_tbl;
    ```

  - 查看通过HASH函数进行拆分的表的建表语句：

    ```
    SHOW CREATE TABLE multi_db_single_tbl;
    ```

- 删除表。
  - 删除单表：

    ```
    DROP TABLE single_tbl;
    ```

  - 删除通过HASH函数进行拆分的表：

    ```
    DROP TABLE multi_db_single_tbl;
    ```

- 查看数据库中的所有表：

  ```
  SHOW TABLES;
  ```

增加列、删除列、修改列的数据类型

- 增加列：

  ```
  ALTER TABLE multi_db_single_tbl ADD COLUMN textcol text;
  ```

- 修改列的数据类型：

  ```
  ALTER TABLE multi_db_single_tbl MODIFY COLUMN textcol varchar(40);
  ```

- 删除列：

  ```
  ALTER TABLE multi_db_single_tbl DROP COLUMN textcol;
  ```

创建、查看和删除本地索引

- 您可以使用如下两种语句中的任意一种创建本地索引：

  ```
  CREATE INDEX idx_name ON multi_db_single_tbl (name);
  ```

```
ALTER TABLE multi_db_single_tbl ADD INDEX idx_name(name);
```

- 查看本地索引：

```
SHOW INDEX FROM multi_db_single_tbl;
```

- 您可以使用如下两种语句中的任意一种删除本地索引：

```
DROP INDEX idx_name ON multi_db_single_tbl;
```

```
ALTER TABLE multi_db_single_tbl DROP INDEX idx_name;
```

增删改查表数据

- 插入表数据：

```
INSERT INTO multi_db_single_tbl (name) VALUES ('test_name');
INSERT INTO multi_db_single_tbl (name) VALUES ('test_name');
INSERT INTO multi_db_single_tbl (name) VALUES ('test_name'),('test_namexx');
```

- 查询表数据：

```
SELECT * FROM multi_db_single_tbl;
```

- 修改表数据：

```
UPDATE multi_db_single_tbl set name='zzz' WHERE id in (100001,100002,100003,100004);
```

- 删除表数据：

```
DELETE FROM multi_db_single_tbl WHERE id = 100002;
```

7.用户指南

7.1. 规格说明

本文介绍了PolarDB-X的实例规格。

PolarDB-X目前支持两种子系列：通用规格和独享规格（含独占物理机规格）。

- 独享规格：每个PolarDB-X实例会独占所分配到的计算资源（如CPU），而不会与同一服务器上的其他集群共享资源，性能表现更加稳定。
- 通用规格：同一硬件资源下的不同PolarDB-X实例，会互相充分利用彼此空闲的计算资源（如CPU），通过复用计算资源享受规模红利，性价比更高。
- 独占物理机规格：每个PolarDB-X实例会独占所分配到的计算资源（如CPU），数据存储节点会独占整个主机的IOPS资源。

通用规格

系列	规格码	CPU和内存	最大存储	最大连接数	最大IOPS
通用	polarx.x4.medium.2e	2核8G	3072GB	20000	4000
通用	polarx.x4.large.2e	4核16G	3072GB	20000	7000
通用	polarx.x4.xlarge.2e	8核32G	3072GB	20000	12000
通用	polarx.x4.2xlarge.2e	16核64G	3072GB	20000	14000

独享规格

系列	规格码	CPU和内存	最大存储	最大连接数	最大IOPS
独享	polarx.x8.large.2e	4核32G	3072GB	20000	9000
独享	polarx.x8.xlarge.2e	8核64G	3072GB	20000	18000
独享	polarx.x8.2xlarge.2e	16核128G	3072GB	20000	36000
独享	polarx.x8.4xlarge.2e	32核128G	3072GB	20000	36000
独享	polarx.x8.4xlarge.2e	32核256G	3072GB	20000	72000

独占物理机规格

系列	规格码	CPU和内存	最大存储	最大连接数	最大IOPS
独占	polarx.st.8xlarge.25	60核470G	6144GB	20000	120000
独占	polarx.st.12xlarge.25	90核720G	6144GB	20000	140000

规格说明

- 独享和独占物理机规格针对存储节点（DN），所有的计算节点（CN）均为独享型。
- 上述为单节点规格指标，为实现高可用，PolarDB-X实例的最小节点数为2。实例整体性能和容量指标为单节点指标×节点数。

- 2核8G的实例定位入门级，用于测试、体验和极小负载的场景，高负载的生产环境不建议使用。生产环境推荐使用8核32G及以上独享规格。
- 您可以在实例创建时选择主实例节点规格，只读实例将采用与主实例相同的规格和节点数。
- 最大IOPS为理论换算值；最大连接数为应用程序到计算节点（CN）的连接数。

7.2. 实例管理

7.2.1. 实例变配

一个PolarDB-X实例是由多个节点组成的分布式集群。单个节点承担了SQL路由、数据合并、聚合等功能。通过实例变配，您可以变更实例的节点规格，在业务繁忙时进行升配，承载更多的业务流量QPS，在业务空闲时进行降配，避免资源浪费。

注意事项

- 降配过程中由于应用与实例连接会中断，可能在短时间内会产生闪断 ，请确保应用程序具备重连机制。
- 使用长连接时，升配后新增节点无法立即接收到流量，建议灰度重启ECS服务器（即先重启拥有少量业务服务的ECS，观察业务情况，确认没问题后，再重启其他所有ECS）。如果使用Druid作为链接池，可以参考长链接的最佳实践，请参见如何选择应用端链接池。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 在目标实例右侧的**操作**栏中，单击**升配**或**降配**。

5. 在**变配**或**降配**页面，选择目标节点规格。

> **说明** 变配完成后，计算层和存储的节点都会升配或降配到相同的目标节点规格。

6. 选中服务协议，单击**立即购买**。
7. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。

7.2.2. 增加/减少节点

一个PolarDB-X实例是由多个节点组成的分布式集群。PolarDB-X支持通过增加或减少节点来对实例进行扩容和缩容。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在目标实例右侧的**操作**栏中，单击**增加节点**或**删减节点**。

3. 在弹出页面，选择要增加或者减少的节点数量。

> **说明**
> - 删减节点动作会导致实例所能承担的负载上限减少，请全面评估并确认剩余的规格容量足够承担业务所需；
> - 删减节点后，历史备份集会失效，建议删减节点成功后立即备份。

4. 单击**立即购买**。

   执行成功后，在控制台可查看到实例状态变更为**增加节点中**和**删减节点中**。

> **说明** 增加节点需要支付相关费用，减少节点则不需要。

7.2.3. 释放实例

本文介绍如何释放PolarDB-X实例。

前提条件

- 释放实例前，请确保已经删除了该实例下的所有数据库。

> **警告** 数据库删除后，数据库内所有数据将被清空且无法找回，请谨慎操作。

- 仅商品类型为按量付费的实例才支持被释放。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 在目标实例右侧的**操作**栏中，单击图标后，再单击**释放**。

> **警告** 实例释放后不可恢复，请谨慎操作。

5. 在弹出的对话框中，单击**确定**。

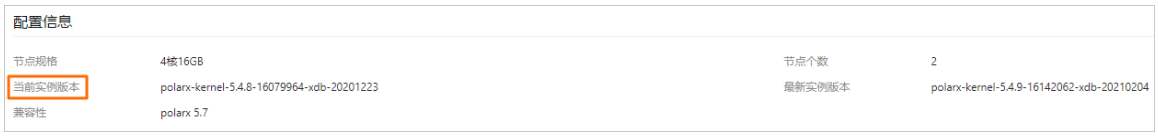
7.2.4. 查看和升级实例版本

本文介绍查看和升级PolarDB-X实例版本的方法。

查看实例版本

您可以通过如下方式查看当前实例的版本：

- 通过控制台查看。
  i. 登录云原生分布式数据库控制台。
  ii. 在页面左上角选择目标实例所在地域。
  iii. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
  iv. 找到目标实例，单击实例ID。
  v. 在**配置信息**区域，您可以查看目标实例的**当前实例版本**信息。



- 通过 `select version()` 命令查看。

  您可以先连接PolarDB-X数据库，再通过执行如下 `select version()` 命令查看目标实例的当前版本：

```
mysql> select version();
+----------------------------+
| VERSION()                  |
+----------------------------+
| 5.6.29-TDDL-5.4.8-16079964 |
+----------------------------+
1 row in set (0.00 sec)
```

> **说明**
> - 其中 `5.4.8-16079964` 即为目标实例的当前版本。
> - 关于如何连接数据库，请参见连接数据库。

升级实例版本

您可以在控制台上升级PolarDB-X实例到最新版本，快速体验实例新特性。

注意事项

- 升级时请不要进行其它操作（如建库、增加只读实例等）。
- 升级过程中会有闪断和少量报错，请在业务低峰期执行。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在**基本信息**页面底部的**配置信息**区域，单击右侧**升级到最新版本**。

> **说明** 升级任务开始后，您可以单击页面右上角的图标查看当前版本升级任务的执行进度。

相关文档

版本发布说明

7.2.5. 重启实例

当PolarDB-X实例出现连接数满或性能问题时，您可以在控制台上手动重启实例。本文将介绍如何重启实例。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在**基本信息**页面，单击右上角**重启实例**。

6. 在弹出的对话框中，单击**确定**。

7.2.6. 实例续费

包年包月实例到期后会导致实例锁定不可用，您可以在实例到期前进行续费，延长实例的使用时间。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 在目标实例右侧的**操作**栏中，单击图标后，再单击**续费**。

5. 在**续费**页面中，选择购买时长。
6. 选中服务协议，单击**立即购买**。
7. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。

7.2.7. 只读实例

本文将介绍如何为主实例添加只读实例。

背景信息

主实例存储节点采用了基于Paxos的三副本高可用架构。而只读实例的存储节点，则通过增加Learner副本的方式来保证物理资源隔离，同时能够基于全局时钟来确保只读查询的强一致性。

注意事项

- 只读实例与主实例必须属于同一个地域，但是可以在不同的可用区。
- 只读实例无法单独存在，必须隶属于某个主实例。若您在主实例上创建了数据库，该数据库也会被复制到只读实例上。同理，若您在主实例上删除数据库时，只读实例上对应的数据库也会被删除。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 在目标实例右侧的**操作**栏中，单击**只读实例**。

   您还可以点击进入实例详情页面，在页面右侧的**只读实例**区域中，单击**添加只读实例**。

5. 在弹出的对话框中，设置如下参数。

参数	说明
商品类型	您可以选择**包年包月**或**按量付费**。 **说明** ○ 包年包月为预付费模式，用于长期生产环境稳定使用，单位价格比按量付费便宜。 ○ 按量付费为后付费模式，通常用于测试或学习了解。 ○ 若您需要创建按量付费的实例，请确保您的阿里云账户余额大于等于100元。
地域	只读实例与主实例需在同一地域。例如主实例所在地域为华东1（杭州），则此时只读实例的地域取值即固定为华东1（杭州），无需选择。
可用区	为只读实例选择可用区。主实例和只读实例可以在不同可用区。
网络类型	固定为**专有网络**，不可变更。
专有网络（VPC）	选择专有网络名称。
交换机（VSwitch）	选择虚拟交换机名称。
MySQL版本	固定为**5.7**，不可变更。
节点规格	按照节点规格×节点个数的方式售卖，您可以通过选择规格配置实例物理规格CPU核数和内存大小。 **说明** 计算层节点和存储层节点都使用相同的节点规格。
节点个数	选择目标规格节点的节点个数，默认2个节点。
存储类型	固定为**本地SSD盘**，不可变更。
存储费用	每个节点存储空间默认为3TB，您购买时无需选择容量，根据实际使用按小时计费，详情请参见存储空间价格。

参数	说明
购买时长	选择实例购买的时长。 说明 仅当商品类型为**包年包月**时支持该配置。
购买数量	选择需要购买的实例数量。

6. 选中服务协议，单击**立即购买**。
7. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。
8. 支付成功后，请耐心等待10~15分钟开通服务。之后您可以返回控制台的**实例列表**页，查看新创建的只读实例。

7.3. 网络与连接

7.3.1. 连接数据库

PolarDB-X支持通过数据管理DMS、MySQL命令行、第三方客户端以及符合MySQL官方交互协议的第三方程序代码进行连接。本文主要介绍如何通过MySQL命令行连接到PolarDB-X数据库。

前提条件

如果您是首次使用PolarDB-X，需要先完成PolarDB-X首次使用流程中的基本操作，其中包括创建实例、创建账号、创建数据库等。

如果您的服务器尚未安装MySQL客户端，请前往MySQL网站下载安装。

通过MySQL命令行连接到数据库

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在**连接信息**区域，找到内外网地址和内网端口信息。

   说明 如果您需要通外网地址进行访问试用，请使用**外网地址**和**外网端口**进行连接。为确保数据库安全，建议您正式访问时使用内网连接地址。

6. 获取**连接信息**后，执行以下命令获取本地对应的公网IP并将其加入实例白名单。关于如何添加白名单，请参考设置白名单。

   ```
   curl 'https://api.ipify.org'
   ```

7. 通过如下MySQL命令行进行连接：

   ```
   mysql -h<连接地址> -P<端口> -u<用户名> -p -D<数据库名称>
   ```

选项	说明	示例
-h	实例的内网连接地址或外网连接地址。	pxc-******************.public.polarx.rds.aliyuncs.com

选项	说明	示例
-P	实例的端口号。 ○ 若使用内网连接，需输入实例的内网端口。 ○ 若使用外网连接，需输入实例的外网端口。 **说明** ○ 此处 -P 为大写字母。 ○ 默认端口为3306。	3306
-u	实例中的账号名称。关于如何创建账号，请参见创建账号。	testuser
-p	以上账号的密码。 **说明** ○ 为保障密码安全，-p 后请不要填写密码，会在执行整行命令后提示您输入密码，输入后按回车即可登录。 ○ 如果填写该参数，-p 与密码之间不能有空格。	passWord123
-D	需要登录的数据库名称。 **说明** ○ 该参数非必填参数。 ○ 可以不输入 -D 仅输入数据库名称。	mysql

通过第三方客户端连接到数据库

PolarDB-X支持通过如下第三方客户端进行连接，您可以去对应的官方网站下载客户端。

- MySQL Workbench（推荐）
- SQLyog
- Sequel Pro
- Navicat for MySQL

说明 第三方GUI客户端可执行基础的数据库操作，包括数据的增删改查和DDL操作，对于工具高级特性，PolarDB-X可能并不支持。

通过第三方程序代码连接到数据库

PolarDB-X支持通过如下符合MySQL官方交互协议的第三方程序代码进行连接：

- JDBC Driver for MySQL (Connector/J)
- Python Driver for MySQL (Connector/Python)
- C++ Driver for MySQL (Connector/C++)
- C Driver for MySQL (Connector/C)

- ADO.NET Driver for MySQL (Connector/NET)
- ODBC Driver for MySQL (Connector/ODBC)
- PHP Drivers for MySQL (mysqli, ext/mysqli, PDO_MYSQL, PHP_MYSQLND)
- Perl Driver for MySQL (DBD::mysql)
- Ruby Driver for MySQL (ruby-mysql)

以下为JDBC Driver for MySQL （Connector/J）程序代码示例。

```
//JDBC
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection("jdbc:mysql://pxc-******************.public.polarx.rds.aliyuncs.com:3306/doc_test","doc_test","doc_test_password");
//...
conn.close();
```

以下为应用端连接池配置示例。

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">
<property name="url" value="jdbc:mysql://pxc-******************.public.polarx.rds.aliyuncs.com:3306/doc_test" />
<property name="username" value="doc_test" />
<property name="password" value="doc_test_password" />
<property name="filters" value="stat" />
<property name="maxActive" value="100" />
<property name="initialSize" value="20" />
<property name="maxWait" value="60000" />
<property name="minIdle" value="1" />
<property name="timeBetweenEvictionRunsMillis" value="60000" />
<property name="minEvictableIdleTimeMillis" value="300000" />
<property name="testWhileIdle" value="true" />
<property name="testOnBorrow" value="false" />
<property name="testOnReturn" value="false" />
<property name="poolPreparedStatements" value="true" />
<property name="maxOpenPreparedStatements" value="20" />
<property name="asyncInit" value="true" />
</bean>
```

> **说明** 推荐使用Druid连接池连接，关于Druid的详细信息请参见Druid Github资源。

7.3.2. 集群地址和只读地址

本文将介绍实例的连接地址及设置项。

背景信息

当您只购买了主实例时，只会有一个连接地址，也叫集群地址（Cluster Endpoint），主要面向在线通用业务场景。在主实例页面上，如果添加了只读实例，每个只读实例页面上会提供一个只读地址（Read Only Endpoint），专注离线拖数、跑批等资源链路隔离场景，确保只读资源可被独享。在HTAP场景下，购买了主实例和只读实例，您可以使用集群地址（Cluster Endpoint）和只读地址（Read Only Endpoint）。

配置集群地址

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 单击主实例名称，打开基本信息界面。

5. 在**连接信息区域**单击集群地址的 图标。

6. 单击**配置管理**，查看集群地址配置项。

配置项说明：

参数	说明
资源配置	若已成功购买只读实例，可以选择将需要分担流量的只读实例加入到主实例资源池中，未加入的将不分担主实例的流量。
只读实例自动加入	控制新购的只读实例是否加入到主实例资源池，默认开启。
智能读写分离	此功能默认为开启，会自动将AP流量路由给多个只读实例去计算，并做MPP并行加速，详细信息请参见读写分离。
只读流量占比	将原本路由给主实例存储的读TP查询按一定的比例规则路由给只读实例存储。
数据一致性读	◦ 强一致性：路由到只读实例的请求，保证一定能够查询到读请求执行前在主实例上已完成更新的数据，提供外部强一致性； ◦ 弱一致性：路由到只读实例的请求，读请求仅访问只读实例上当前的最新数据，会因为主从异步复制的架构产生数据读取延迟。 **说明** 在主从复制延迟的条件下，强一致性读会等待直到读取到已完成更新的数据，默认的等待超时是3s。
只读延迟阈值	超过该阈值后只读流量会发送给其他延迟正常的只读实例，或者根据只读实例可用性决定是否切回主库。
只读可用性	当所有只读实例的延迟超过阈值或者出现HA状态，则认为当前所有只读实例处于不可用的状态，判断是否将只读流量回切主库（建议只读实例个数≥2来提升可用性）。

> 说明
> - 在强一致性读下的等待超时是3s，您可以在**参数设置**界面调整 `loose_replica_read_timeout` 参数控制等待超时时间。
> - 读写分离功能要求PolarDB-X版本需为5.4.13-16415631及以上。

配置只读地址

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 在控制台上，单击只读实例名称，打开基本信息页面。
5. 单击只读地址的 。
6. 单击**配置管理**，查看只读地址配置项。

配置项说明：

- 强一致性：通过只读地址可以查看到主实例上提交的最新数据，满足全局数据查询一致性。如果主备延迟过大，提交的查询会等待。
- 弱一致性：通过只读地址可以查看到当前只读实例上最新数据，如果在主实例和只读实例间数据同步有延迟的话，不会等待，查询到数据立即返回。

7.3.3. 申请和释放外网地址

PolarDB-X实例会默认分配一个内网地址，应用和MySQL客户端可以在阿里云ECS上连接PolarDB-X实例的数据库。如果需要从外网访问PolarDB-X，则需要为PolarDB-X实例申请外网地址。本文将介绍如何申请和释放外网地址。

申请外网地址

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在**基本信息**页面中间的**连接信息**区域，单击右上角**申请外网地址**。

6. 在弹出的对话框中，单击**确定**。

> **说明** 如果您需要通过申请的外网地址来连接实例，您还需要将连接实例的客户端所属的公网IP地址加入到实例的白名单中，详情请参见设置白名单。

释放外网地址

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。

4. 找到目标实例，单击实例ID。
5. 在**基本信息**页面中间的**连接信息**区域，单击右上角**释放外网地址**。

6. 在弹出的对话框中，单击**确定**。

> **说明** 释放外网地址后，将无法继续通过外网地址连接到该实例。同时，为保障数据安全性，建议您在白名单设置中删除对应的公网IP地址，详情请参见设置白名单。

7.4. 账号和安全

7.4.1. 注册和登录阿里云账号

本文介绍如何注册和登录阿里云账号。

注册阿里云账号

1. 您可以通过如下两种方式中的任意一种进入阿里云账号注册页面：
   - 进入阿里云官网，单击右上角的**免费注册**。

   - 直接访问注册页面。
2. 在**欢迎注册阿里云**页面填入您的账号信息并同意隐私条款、服务协议等，完成注册。

登录阿里云账号

阿里云账号的登录入口与子账号不同。

- 阿里云账号的登录入口。

- 子账号的登录入口。

7.4.2. 管理数据库账号

PolarDB-X实例支持高权限账号和普通账号，本文介绍账号的相关管理操作。

账号类型

PolarDB-X实例支持如下两种数据库账号类型：

账号类型	建库数量	说明
高权限账号	500	• 只能通过控制台或API进行创建和管理。 • 一个实例中只能创建一个高权限账号，可以管理所有普通账号和数据库。 • 拥有更多权限，可满足个性化和精细化的权限管理需求，例如可按用户分配不同表的查询权限。 • 拥有实例下所有数据库的所有权限，可以断开任意账号的连接。
普通账号	500	• 可以通过控制台、API或SQL语句进行创建和管理。 • 一个实例可以创建多个普通账号，具体的数量与实例内核有关。 • 需要手动给普通账号授予特定数据库的权限。 • 普通账号不能创建和管理其他账号，也不能断开其他账号的连接。

> **说明**
> - 账号创建后无法切换账号类型，但您可以删除该账号后重新创建一个账号类型不同但名称相同的账号。
> - 如果需要为阿里云账号创建子账号并授权子账号管理某些实例，请参见创建RAM用户。

您可以在控制台上对账号进行如下操作：

- 创建账号
- 重置密码
- 修改普通账号的权限
- 删除账号

创建账号

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。

4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 账号管理**。
6. 单击**创建账号**，在弹出的对话框中，设置如下参数。

参数	说明
账号名	填写账号名称。 **说明** 账号名称需满足如下要求： ○ 由小写字母、数字或下划线组成，长度不超过16个字符。 ○ 以字母开头，以字母或数字结尾。 ○ 不能和已有的账号名重复。
账号类型	您可以选择账号类型为**高权限账号**或**普通账号**。 **说明** 每个PolarDB-X实例最多只允许创建1个高权限账号。
授权数据库	为该账号授予一个或多个数据库的权限。 i. 选中一个或多个数据库，单击 图标将其从左侧**未授权数据库**添加到右侧**已授权数据库**。 ii. 在右侧**已授权数据库**中，为目标数据库选择权限。 **说明** ○ 仅当账号类型为**普通账号**时，支持该配置。 ○ 该参数为非必填项，您也可以在账号创建完成后再给账号授权。 ○ 支持的权限为**读写**、**只读**、**仅DML**、**仅DDL**和**只读+索引**。 ○ 如果要为多个数据库批量设置相同的权限，可以单击**已授权数据库**框右上角的权限对应按钮，如**全部读写**。
密码	设置账号密码。 **说明** 账号密码需满足如下要求： ○ 密码长度需为8~32个字符。 ○ 由大写字母、小写字母、数字、特殊字符中的任意三种组成。 ○ 特殊字符为： @#$%^&+=
确认密码	再次输入密码。
备注说明	备注该账号的相关信息，便于后续账号管理。该参数为非必填项，最多支持256个字符。

重置密码

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 账号管理**。

6. 在**账号管理**页，单击目标账号右侧**操作**栏中的**修改密码**。

7. 在弹出的对话框中，输入新密码并确认密码，单击**确定**。

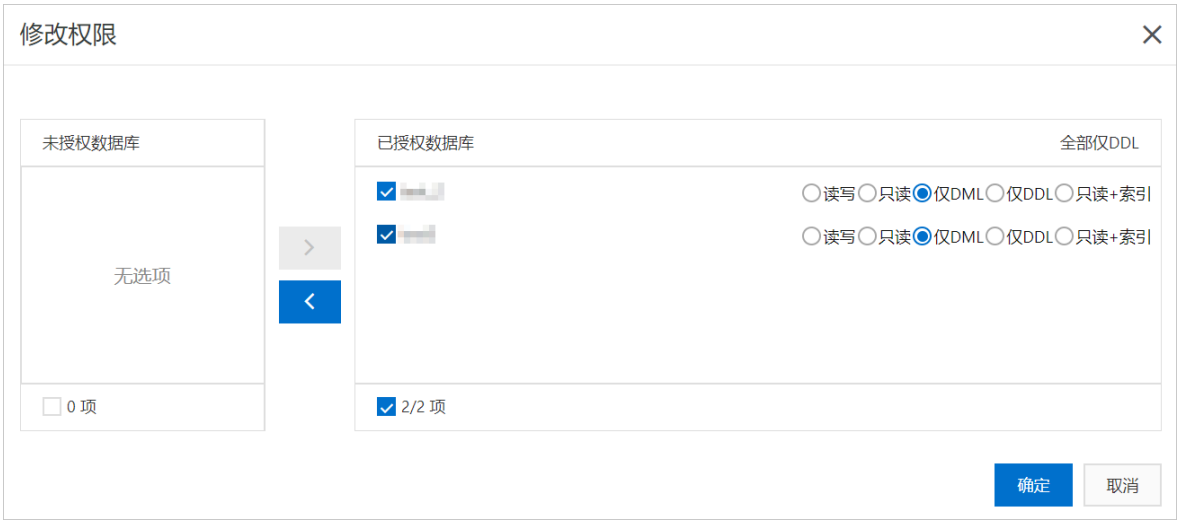
> **说明** 账号密码需满足如下要求：
> - 密码长度需为8~32个字符。
> - 需由大写字母、小写字母、数字、特殊字符中的任意三种组成。
> - 特殊字符为：
>
> @#$%^&+=

修改普通账号的权限

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 账号管理**。
6. 在**账号管理**页，单击目标账号右侧**操作**栏中的**修改权限**。

> **说明** 高权限账号拥有所有数据库的所有权限，因此不需要**修改授权**。

7. 在弹出的对话框中，选中一个或多个数据库，单击 > 图标将其从左侧**未授权数据库**添加到右侧**已授权数据库**。
8. 在右侧**已授权数据库**中，为目标数据库选择权限。



> **说明**
> - 支持的权限为**读写**、**只读**、**仅DML**、**仅DDL**和**只读+索引**。
> - 如果要为多个数据库批量设置相同的权限，可以单击**已授权数据库**框右上角的对应权限按钮，如**全部仅DDL**。

9. 单击**确定**。

删除账号

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 账号管理**。
6. 在**账号管理**页，单击目标账号右侧**操作**栏中的**删除**。

7. 在弹出的对话框中，单击**确定**。

7.4.3. 设置白名单

PolarDB-X实例支持白名单设置，设置白名单并不会影响PolarDB-X实例的正常运行。您可以通过定期维护白名单，保证PolarDB-X实例得到高级别的访问安全保护。本文将介绍白名单的相关操作。

注意事项

- 数据库创建完成后，默认的IP白名单分组内只包含IP地址`127.0.0.1`，表示任何IP地址均无法访问该数据库。出于数据安全考虑，建议仅将您的WEB服务器外网IP或IP段加入白名单。
- 目前支持创建50个IP白名单分组，每个IP白名单分组内最多添加300个IP或IP段。

新增白名单分组

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。

3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 安全管理**。
6. 在**白名单设置**页签，单击左上角**新增白名单分组**。
7. 在**新增白名单分组**面板，配置以下参数。

参数	说明
分组名称	输入白名单分组名称。
组内白名单	输入IP地址。 **说明** ○ 您可以填写IP地址（如192.168.0.1）或IP段（如192.168.0.0/24）。 ○ 每个IP白名单分组最多添加1000个IP或IP段。 ○ 多个IP地址间用英文逗号（,）隔开（如192.168.0.1,192.168.0.0/24）。 ○ 您可以通过输入127.0.0.1禁止任何IP地址访问。

8. 单击**确定**即可。

> **说明** 修改白名单后，需要等待约1分钟才会生效。

修改组内白名单

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 安全管理**。
6. 在**白名单设置**页签，找到目标白名单分组名称，单击右侧**操作**栏中的**配置**。

7. 在**配置白名单**面板，对**组内白名单**进行修改。

参数	说明
组内白名单	输入IP地址。 **说明** ○ 您可以填写IP地址（如192.168.0.1）或IP段（如192.168.0.0/24）。 ○ 每个IP白名单分组最多添加1000个IP或IP段。 ○ 多个IP地址间用英文逗号（,）隔开（如192.168.0.1,192.168.0.0/24）。 ○ 您可以通过输入127.0.0.1禁止任何IP地址访问。

8. 单击**确定**即可。

> **说明** 修改白名单后，需要等待约1分钟才会生效。

删除白名单分组

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 安全管理**。
6. 在**白名单设置**页签，找到目标白名单分组名称，单击右侧**操作**栏中的**删除**。

7. 在弹出的对话框中，单击**确定**即可。

7.4.4. 设置SSL加密

为了提高链路安全性，您可以启用SSL（Secure Sockets Layer）加密，并安装SSL CA证书到需要的应用服务。SSL在传输层对网络连接进行加密，能提升通信数据的安全性和完整性，但会同时增加网络连接响应时间。

前提条件

实例需为PolarDB-X且版本为5.4.10或以上，实例基本信息页可查看实例版本信息。

注意事项

- SSL的证书有效期为1年，请及时更新证书有效期并重新下载配置CA证书，否则使用加密连接的客户端程序将无法正常连接。
- 由于SSL加密的实现原理，启用SSL加密会显著增加CPU使用率，建议您仅在外网链路有加密需求的时候启用SSL加密。内网链路相对较安全，一般无需对链路加密。
- 开启或关闭SSL加密会重启实例，请谨慎操作。

开启SSL加密

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6. 单击**SSL配置**页签，打开**SSL配置**的开关。

> **说明** 开启SSL加密时，默认选择内网地址为**SSL受保护的地址**。您也可以将受保护的地址修改为公网地址，详情请参见修改SSL受保护地址。

7. 在弹出的对话框中，单击**确定**。

> **注意** 开启或关闭SSL加密会重启实例，建议在业务低峰期操作，并确保您的应用具备自动重连机制。

8. 开通成功后，单击**下载CA证书**。

   下载的文件为压缩包，包含如下三个文件：

   - .p7b文件：用于Windows系统中导入CA证书。
   - .pem文件：用于其他系统或应用中导入CA证书。
   - .jks文件：Java中的truststore证书存储文件，密码统一为apsaradb，用于Java程序中导入CA证书链。

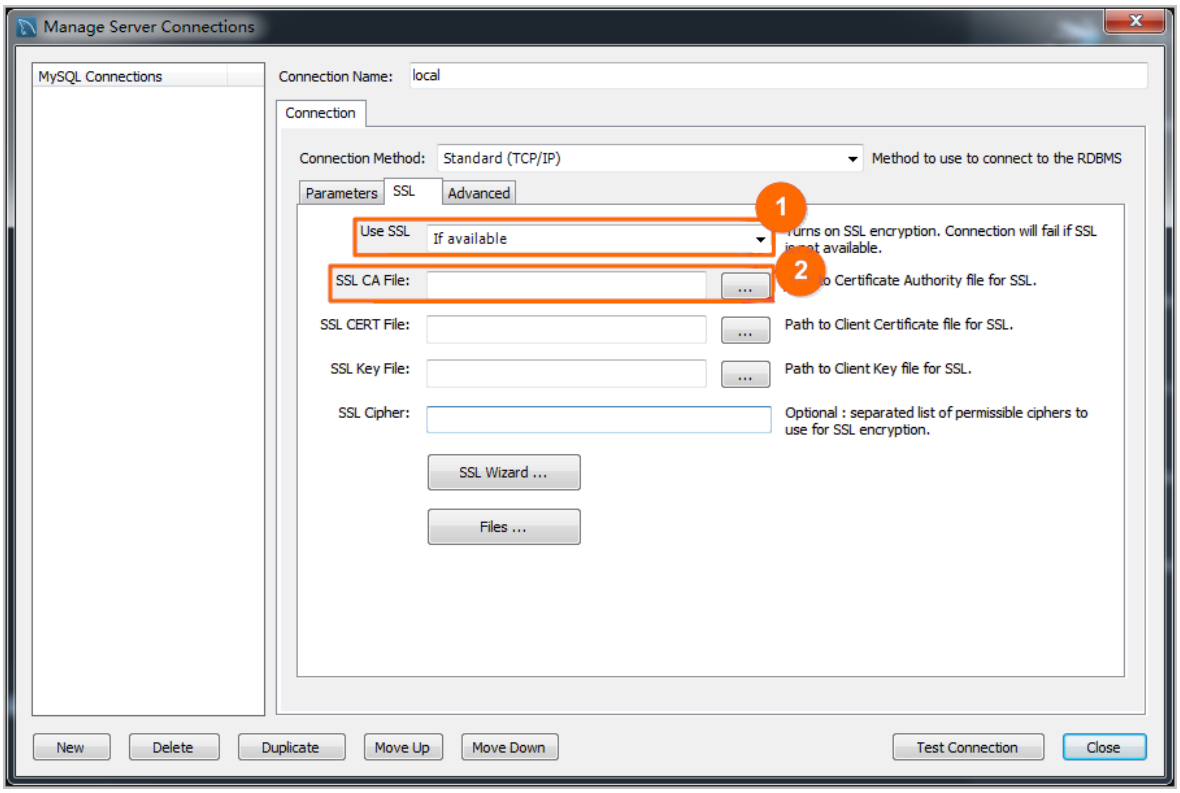
   > **说明** 在Java中使用JKS证书文件时，jdk7和jdk8需要修改默认的jdk安全配置，在应用程序所在主机的 `jre/lib/security/java.security` 文件中，修改如下两项配置：
   >
   > ```
   > jdk.tls.disabledAlgorithms=SSLv3, RC4, DH keySize < 224
   > jdk.certpath.disabledAlgorithms=MD2, RSA keySize < 1024
   > ```
   >
   > 若不修改jdk安全配置，会报如下错误。其它类似报错，一般也都由Java安全配置导致。
   >
   > ```
   > javax.net.ssl.SSLHandshakeException: DHPublicKey does not comply to algorithm constraints
   > ```

配置SSL CA证书

开通SSL加密后，应用或者客户端连接PolarDB-X时需要配置SSL CA证书。本文以MySQL Workbench和Navicat为例，介绍SSL CA证书安装方法。其它应用或者客户端请参见对应产品的使用说明。

MySQL Workbench配置方法

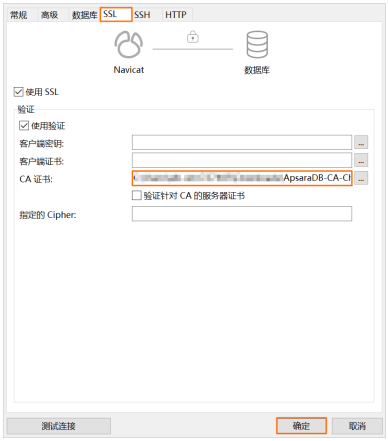
1. 打开MySQL Workbench。
2. 选择**Database > Manage Connections**。
3. 启用**Use SSL**，并导入SSL CA证书。



Navicat配置方法

1. 打开Navicat。
2. 在目标数据库上单击鼠标右键，选择**编辑连接**。

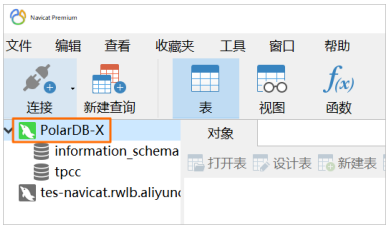
3. 选择SSL页签，选择.pem格式CA证书的路径。



4. 单击**确定**。

> **说明** 如果报 `connection is being used` 错误，是由于之前的会话未断开，请关闭Navicat重新打开。

5. 双击目标数据库测试能否正常连接。



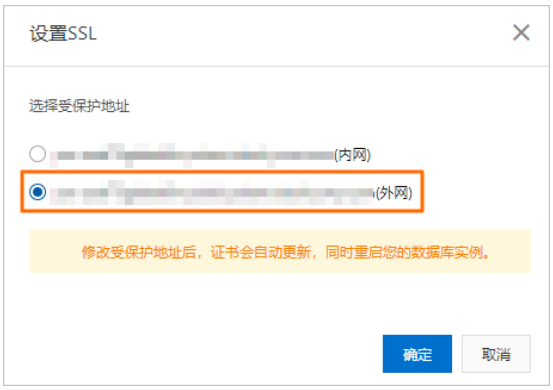
更新证书有效期

> **说明**
> - **更新有效期**操作将会重启实例，建议在业务低峰期操作，并确保您的应用具备自动重连机制。
> - **更新有效期**后需要重新下载及配置CA证书。

修改SSL受保护地址

1. 登录云原生分布式数据库控制台。

2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6.
7. 单击**设置SSL**。
8. 在弹出的对话框中，选择目标地址为受保护的地址。



> **说明**
> - 开通SSL时，默认选择内网地址为**SSL受保护的地址**，您可以将其修改为公网地址。
> - 修改SSL受保护地址操作将会重启实例，建议在业务低峰期操作，并确保您的应用具备自动重连机制。
> - 修改SSL受保护地址后需要重新下载及配置CA证书。

9. 单击**确定**即可。

关闭SSL加密

> **说明**
> - 关闭SSL加密会重启实例，建议您在业务低峰期操作，并确保您的应用具备自动重连机制。
> - SSL加密关闭后，数据库访问性能会有一定程度提升，但安全性上有削弱，故非安全环境下不建议关闭SSL加密。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6.
7. 关闭**SSL配置**的开关。

8. 在弹出的框中对话中单击**确定**即可。

7.4.5. 设置透明数据加密

本文介绍了使用透明数据加密TDE（Transparent Data Encryption） 的方法。

功能介绍

透明数据加密TDE（Transparent Data Encryption）可对数据文件执行实时I/O加密和解密，数据在写入磁盘之前进行加密，从磁盘读入内存时进行解密。TDE不会增加数据文件的大小，开发人员无需更改任何应用程序，即可使用TDE功能。

加密使用的密钥由密钥管理服务（KMS）产生和管理，PolarDB-X不提供加密所需的密钥和证书。PolarDB-X不仅支持使用阿里云自动生成的密钥，也可以使用自带的密钥材料生成数据密钥，然后授权PolarDB-X使用。

前提条件

已开通KMS。如果您未开通KMS，可在开通TDE过程中根据引导开通KMS。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6. 在**TDE**页签单击**未开通**左边的滑块。
7. 选择**使用由阿里云自动生成的密钥**，单击**确定**，开通TDE。
   - 选择**使用由阿里云自动生成的密钥**，单击**确定**，开通TDE。
   - 选择**使用已有自定义密钥**，选择密钥，单击**确定**，开通TDE。

> **说明** 如果没有自定义密钥，需要单击**前往创建**，在密钥管理服务控制台创建密钥并导入自带的密钥材料。详情请参见密钥管理服务。

加密操作

登录数据库，执行如下命令，对要加密的表进行加密。

```
alter table <tablename> encryption='Y';
```

执行如下命令，可直接创建加密表。

```
create table <tablename> <col definition> ENCRYPTION='Y';
```

解密操作

登录数据库，执行如下命令，对TDE加密的表解密。

```
alter table <tablename> ENCRYPTION='N';
```

7.4.6. 三权分立

7.4.6.1. 功能介绍

PolarDB-X新增三权分立模式，您可以将高权限账号拥有的权限分给系统管理员、安全管理员和审计管理员这3个角色，避免因权限高度集中带来的风险，增强数据库的安全性。

风险与解决方案

- **风险**

  传统数据库运维模式下，数据库管理员DBA（Database Administrator）拥有的权限过高且集中，容易在某些场景下给业务带来风险：

  - DBA误判导致系统安全事故。
  - DBA出于某种目的进行非法操作。
  - DBA、第三方外包人员或程序开发人员越权访问敏感信息。

- **解决方案**

  PolarDB-X新增支持三权分立模式，打破传统数据库运维由DBA行使特权的独立控制体系，使得数据库管理员DBA、安全管理员DSA（Department Security Administrator）和审计管理员DAA（Data Audit Administrator）3者的权责更加清晰。其中：

  - 数据库管理员（DBA）：只具备DDL（Data Definition Language）权限。
  - 安全管理员（DSA）：只具备管理角色（Role）或用户（User）以及为其他账号授予权限的权限。
  - 审计管理员（DAA）：只具备查看审计日志的权限。

数据库系统账号的权限对比

下表展示了在默认模式和三权分立模式下，不同数据库系统账号的权限对比。

> **说明**
> - 默认模式下的高权限账号即系统管理员账号。更多关于高权限账号的详情，请参见账号类型。
> - 开启或关闭三权分立模式，仅对系统账号（即高权限账号、系统管理员账号、安全管理员账号和审计管理员账号）的权限有影响，普通账号权限不受模式变更的影响。
> - 三权分立模式下，虽然所有系统账号均不具备DML（Data Manipulation Language）、DQL（Data Query Language）或DAL（Data Administration Language）权限，但安全管理员仍然能够将这些权限授予给普通账号。
> - 表中✔☺表示具备该权限，□表示不具备该权限。

权限		默认模式	三权分立模式		
分类	说明	高权限账号	系统管理员账号	安全管理员账号	审计管理员账号
DDL	• ALTER TABLE • CREATE TABLE • CREATE VIEW • CREATE INDEX • CREATE CCL_RULE • DROP VIEW • DROP INDEX • DROP TABLE • TRUNCATE TABLE	✔☺	✔☺	□	□

权限		默认模式	三权分立模式		
分类	说明	高权限账号	系统管理员账号	安全管理员账号	审计管理员账号
DML	• DELETE • UPDATE • INSERT	✔	✘	✘	✘
DQL	• SELECT • EXPLAIN				
DAL	• SHOW CCL_RULE • SHOW INDEX				
账号或角色相关	账号权限管理 角色权限管理	✔	✘	✔	✘
查看审计日志	查看如下两张表中的审计日志信息： • `information_schema.polardbx_audit_log` • `information_schema.polardbx_ddl_log`	✔	✘	✘	✔

使用限制

三权分立模式下的系统账号（包括系统管理员账号、安全管理员账号和审计管理员账号）存在如下限制：

- 不支持对系统账号执行GRANT ROLE或REVOKE ROLE命令。
- 不支持对系统账号执行GRANT PRIVILEGES或REVOKE PRIVILEGES命令。
- 系统账号的密码只能由对应的账号修改，如系统管理员账号的密码仅能由系统管理员账号修改，不能被其他帐号修改。
- 系统账号均不支持SET DEFAULT ROLE命令。

7.4.6.2. 开启或关闭三权分立

您可以在控制台上开启或关闭三权分立模式。开启后，高权限账号自动转换为系统管理员账号，同时还需创建安全管理员账号和审计管理员账号。本文介绍开启或关闭三权分立的相关操作步骤。

开启三权分立

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6. 单击**账号三权分立**页签，在页面右上角打开**当前账户安全模式**开关。

   > **说明** 开启三权分立模式前，请确保已创建高权限账号。详细操作步骤，请参见创建账号。

7. 在**创建三权分立账号**面板中，设置以下参数。

参数	说明

参数	说明
账号名	填写账号名称。 **说明** 账号名称需满足如下要求： ○ 由小写字母、数字或下划线组成，长度不超过16个字符。 ○ 以字母开头，以字母或数字结尾。 ○ 不能和已有的账号名重复。
账号类型	固定为**安全管理员账号**，无需选择。
密码	设置账号密码。 **说明** 账号密码需满足如下要求： ○ 密码长度需为8~20个字符。 ○ 由大写字母、小写字母、数字、特殊字符中的任意三种组成。 ○ 特殊字符为： @#$%^&+=
确认密码	再次输入密码。
备注说明	备注该账号的相关信息，便于后续账号管理。该参数为非必填项，最多支持256个字符。

8. 单击**下一步**，并设置以下参数。

参数	说明
账号名	填写账号名称。 **说明** 账号名称需满足如下要求： ○ 由小写字母、数字或下划线组成，长度不超过16个字符。 ○ 以字母开头，以字母或数字结尾。 ○ 不能和已有的账号名重复。
账号类型	固定为**审计管理员账号**，无需选择。
密码	设置账号密码。 **说明** 账号密码需满足如下要求： ○ 密码长度需为8~20个字符。 ○ 由大写字母、小写字母、数字、特殊字符中的任意三种组成。 ○ 特殊字符为： @#$%^&+=

参数	说明
确认密码	再次输入密码。
备注说明	备注该账号的相关信息，便于后续账号管理。该参数为非必填项，最多支持256个字符。

9. 单击**确定**即可。

> **说明** 开启后请等待3~5秒，新设置才会生效。

关闭三权分立

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6. 选择**账号三权分立**页签，在页面右上角关闭**当前账户安全模式**开关。
7. 在弹出的对话框中，单击**确定**。
8. 在弹出的**高权限账号密码验证**对话框中，输入高权限账号的密码并单击**确定**。

> **说明** 关闭后请等待3~5秒，新设置才会生效。

7.4.6.3. 管理三权分立系统账号

开启三权分立模式后，您可以在控制台上查看三权分立系统账号的名称或修改密码。本文介绍相关操作步骤。

使用限制

三权分立模式下的系统账号（包括系统管理员账号、安全管理员账号和审计管理员账号）存在如下限制：

- 不支持对系统账号执行GRANT ROLE或REVOKE ROLE命令。
- 不支持对系统账号执行GRANT PRIVILEGES 或REVOKE PRIVILEGES命令。
- 系统账号的密码只能由对应的账号修改，如系统管理员账号的密码仅能由系统管理员账号修改，不能被其他帐号修改。
- 系统账号均不支持SET DEFAULT ROLE命令。

查看账号

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 您可以选择如下任意方式查看三权分立系统账号信息：
   - 方式一：在**安全管理**页面查看。
     a. 在左侧导航栏选择**配置与管理 > 安全管理**。
     b. 单击**账号三权分立**页签，即可查看三权分立模式下的系统管理员、安全管理员和审计管理员账号的账号名称和备注信息等详情。
   - 方式二：在**账号管理**页面查看。
     a. 在左侧导航栏选择**配置与管理 > 账号管理**。

b. 在**账号管理**页面，除了可以查看三权分立模式的系统账号（包括系统管理员、安全管理员和审计管理员）的信息，还可以查看普通账号的信息，如账号类型，所属数据库和拥有的权限等。

重置密码

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击PolarDB-X 2.0页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 安全管理**。
6. 单击**账号三权分立**页签。
7. 在账号列表中，单击目标账号右侧**操作**栏中的**修改密码**。
8. 在弹出的对话框中，输入新密码并确认。

> **说明** 账号密码需满足如下要求：
> - 密码长度需为8~32个字符。
> - 由大写字母、小写字母、数字、特殊字符中的任意三种组成。
> - 特殊字符为：
>
> @#$%^&+=

9. 单击**确定**即可。

7.4.6.4. 权限验证

开启三权分立模式后，高权限账号自动转换为系统管理员账号，同时还需创建安全管理员账号和审计管理员账号，三个账号拥有不同的权限，当您对数据库执行不同操作时，需要对应的管理员进行权限验证。本文将介绍权限验证的相关操作步骤。

系统管理员（高权限账号）权限验证

关闭三权分立模式时，需要通过系统管理员（高权限账号）的验证。关闭三权分立模式的操作步骤，详情请参见关闭三权分立。

安全管理员权限验证

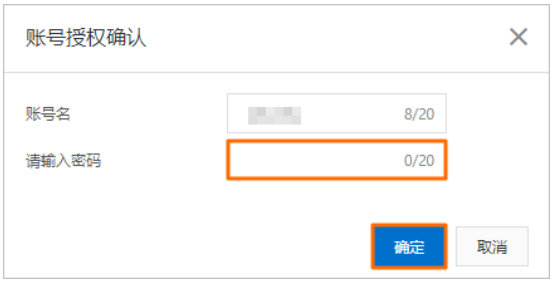
需要由系统管理员进行验证的操作包括：

- 删除普通账号。
- 修改普通账号的权限。
- 修改普通账号的密码。
- 创建数据库时给普通账户授权。

本文以删除普通账号为例介绍相关验证操作步骤。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击PolarDB-X 2.0页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**配置与管理 > 账号管理**。
6. 在**账号管理**页面，找到目标普通账号，单击右侧**操作**栏中的**删除**。
7. 在弹出的 **删除账号**对话框中，单击**确定**。

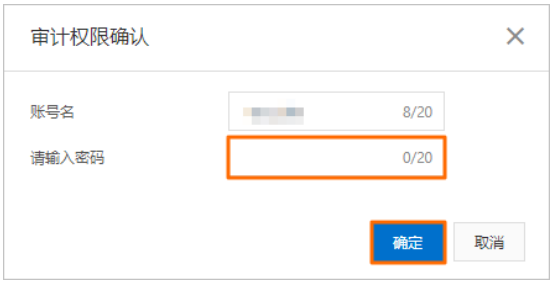
8. 在弹出的**账号授权确认**对话框中，输入安全管理员账号的密码，并单击**确定**。



审计管理员权限验证

在进入**SQL审计与分析**页面查看SQL日志信息时需要经过审计员验证。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏选择**诊断与优化管理 > SQL审计与分析**。
6. 在弹出的**审计权限确认**对话框中，输入审计管理员账号的密码，并单击**确定**。



7.5. 数据导入和导出

7.5.1. 数据同步生态工具

兼容多种数据同步工具

数据同步工具	兼容版本	使用限制
MySQL Slave	>=5.4.11	目前不支持GTID模式复制
canal	>=5.4.11	无
DTS	>=5.4.11	无
debezium	>=5.4.12	不支持快照能力，使用时需要关闭快照"snapshot.mode": "never"
maxwell	>=5.4.12	无
mysql-binlog-connector-java	>=5.4.11	无

数据同步过程中的账号权限问题

- 高权限账号，默认拥有所有权限。
- PolarDB-X从5.4.11版本开始对外提供了完全兼容MySQL的全局binlog能力，为了保障数据安全性，需要通过 `GRANT SELECT ON *.* TO 'user'@'host'` 语句授予账号实例级的SELECT权限， 方可查看和订阅binlog。

- PolarDB-X从5.4.12版本开始，新增了binlog查看和订阅的独立权限。
  - 拥有REPLICATION CLIENT权限 `GRANT REPLICATION CLIENT ON *.* TO 'user'@'host'` ，方可执行以下binlog的查询权限。

    ```
    SHOW MASTER STATUS;
    SHOW BINARY LOGS;
    ```

  - 拥有REPLICATION SLAVE权限 `GRANT REPLICATION SLAVE ON *.* TO 'user'@'host'` ，方可查询binlog event详情和订阅binlog。

    ```
    SHOW BINLOG EVENTS
    ```

7.5.2. 使用DTS导入和导出数据

PolarDB-X提供丰富的数据导入和导出方式，以保持与其他数据系统的互通。本文主要介绍通过DTS导入导出数据的方式。

通过DTS或备份恢复导入数据到PolarDB-X

您可以通过DTS或备份恢复导入数据到PolarDB-X。

- 通过DTS将RDS数据迁移到PolarDB-X。
- 通过DTS将PolarDB数据迁移到PolarDB-X。
- 通过DTS将自建MySQL（公网）数据迁移到PolarDB-X。
- 通过DTS将DB2 LUW数据迁移到PolarDB-X。
- 通过DTS将Oracle (专线) 数据迁移到PolarDB-X。

通过DTS从PolarDB-X导出数据

PolarDB-X通过DTS等方式提供数据同步到下游的能力。

- 在PolarDB-X实例之间同步数据。
- 同步PolarDB-X数据到PolarDB MySQL。
- 同步PolarDB-X数据到RDS。
- 同步 PolarDB-X 数据到自建MySQL（公网）。
- 同步PolarDB-X数据到AnalyticDB MySQL版。
- 同步PolarDB-X数据到AnalyticDB PostgreSQL版。
- 同步PolarDB-X数据到Kafka。
- 同步PolarDB-X数据到Elasticsearch。
- 同步PolarDB-X数据到Datahub。

PolarDB-X增量数据订阅

PolarDB-X提供完全兼容MySQL Binlog体验的增量订阅能力，称为全局日志变更。这项能力可以使消费MySQL Binlog的工具可以无缝对接到PolarDB-X，以便实时获取PolarDB-X的增量数据。

目前已验证的工具列表可参见：数据同步生态工具。

7.5.3. 使用mysqldump导入导出数据

本文介绍了通过mysqldump工具将PolarDB-X数据导入导出的几种常见场景和详细操作步骤。

PolarDB-X支持MySQL官方数据导出工具mysqldump。mysqldump命令的详细说明请参见MySQL 官方文档。

> **说明** mysqldump适合小数据量（低于1000万）的离线导入导出。如果需要完成更大数据量或者实时的数据迁移任务，建议使用DTS进行数据迁移。

mysqldump工具介绍

mysqldump能够导出表结构信息和表内数据，并转化成SQL语句的格式方便用户直接导入，SQL语法如下：

```
DROP TABLE IF EXISTS `table_name`;
CREATE TABLE `table_name` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    ...
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4  dbpartition by hash(`id`);
INSERT INTO `table_name` VALUES (...),(...),...;
INSERT INTO `table_name` VALUES (...),(...),...;
...
```

mysqldump工具导出数据的命令使用方式举例：

```
shell> mysqldump -h ip -P port -u user -pPassword --default-character-set=char-set --net_buffer_length=10240
--no-create-db --no-create-info --skip-add-locks --skip-lock-tables --skip-tz-utc --set-charset  [--hex-blob
] [--no-data] database [table1 table2 table3...] > dump.sql
```

mysqldump的参数说明可通过 `mysqldump --help` 命令查看或查询MySQL 官方文档，常用参数说明如下，请根据实际情况输入：

参数名	说明
ip	PolarDB-X实例的IP。
port	PolarDB-X实例的端口。
user	PolarDB-X的用户名。
password	PolarDB-X的密码，注意前面有个-p，之间没有空格。
char-set	指定的编码。
--hex-blob	使用十六进制格式导出二进制字符串字段。如果有二进制数据就必须使用本选项。影响的字段类型包括BINARY、VARBINARY、BLOB。
--no-data	不导出数据。
table	指定导出某个表。默认导出该数据库所有的表。
--no-create-info	不导出建表信息
--net_buffer_length	传输缓冲区大小。影响Insert语句的长度，默认值1046528。

导出的SQL语句格式文件，有两种方法导入数据库中：

- SOURCE语句导入数据

```
## 1.登录数据库
shell> mysql -h ip -P port -u user -pPassword --default-character-set=char-set
## 2.通过source语句执行文件中的sql语句导入数据
mysql> source dump.sql
```

- mysql命令导入数据

```
shell> mysql -h ip -P port -u user -pPassword --default-character-set=char-set< /yourpath/dump.sql
```

下面从不同场景介绍mysqldump工具的使用实例。

PolarDB-X和MySQL之间数据传输时不推荐导出表结构，因为PolarDB-X包含分库分表功能，CREATE TABLE中的拆分函数等与MySQL不兼容，不兼容的关键字包括：

- DBPARTITION BY hash(partition_key)
- TBPARTITION BY hash(partition_key)

- TBPARTITIONS N
- BROADCAST

如果导出表结构，需在导出的SQL语句文件中修改建表语句，才能正确导入。所以推荐只导出表内数据，手动登录数据库进行建表操作，然后再导入数据。

场景一：从MySQL导入到PolarDB-X

从MySQL导入数据到PolarDB-X，请按照以下步骤进行操作。

1. 从MySQL中导出数据到文件。

   输入以下命令，从MySQL中导出表内数据（不推荐导出表结构），假设导出文件为dump.sql。

   ```
   mysqldump -h ip -P port -u user -pPassword --default-character-set=char-set --net_buffer_length=204800 --no-create-db --no-create-info --skip-add-locks --skip-lock-tables --skip-tz-utc --set-charset --hex-blob database [table1 table2 table3...] > dump.sql
   ```

2. 登录PolarDB-X，手动建立目标表，关于PolarDB-X建表语句的语法请参见CREATE TABLE。如果未加--no-create-info参数，导出的dump.sql文件中包含MySQL端的建表语句，也可在文件中进行修改建表语句。
3. 导入数据文件到PolarDB-X中。您可以通过如下两种方式导入数据文件到PolarDB-X：
   - 通过 `mysql -h ip -P port -u user -pPassword --default-character-set=char-set` 命令登录目标PolarDB-X，执行 `source /yourpath/dump.sql` 命令将数据导入到目标PolarDB-X。
   - 直接通过 `mysql -h ip -P port -u user -pPassword --default-character-set=char-set< /yourpath/dump.sql` 命令将数据导入到目标PolarDB-X。

   > **说明**
   > - 上述两个命令中 `default-character-set` 要设置成实际的数据编码。如果是Windows平台，source命令指定的文件路径需要对分隔符转义。
   > - 第一种方式会把所有的步骤回显到屏幕上，速度略慢，但是可以观察导入过程。
   > - 导入的时候，由于某些PolarDB-X和MySQL实现上的不同，可能会报错，错误信息类似 `ERROR 1231 (HY000): [a29ef6461c00000][10.117.207.130:3306][****]Variable @saved_cs_client can't be set to the value of @@character_set_client` 。此类错误信息并不影响导入数据的正确性。

场景二：从一个PolarDB-X导入到另一个PolarDB-X

假设您之前有一个测试环境的PolarDB-X，测试完毕以后，需要把测试过程中的一些表结构和数据导入到生产环境中的PolarDB-X中，那么可以按照以下步骤进行操作。

1. 从源PolarDB-X中导出数据到文本文件。请参见场景一步骤一。
2. 导入数据文件到PolarDB-X。请参见场景一步骤三。
3. 手动创建Sequence对象。

   mysqldump并不会导出PolarDB-X中的Sequence对象，所以如果在源PolarDB-X中使用了Sequence对象，并且需要在目标PolarDB-X中继续使用相同的Sequence对象，则需要手动在目标PolarDB-X中创建同名的Sequence的对象。具体步骤如下：

   i. 在源PolarDB-X上执行SHOW SEQUENCES，获取当前PolarDB-X中的Sequence对象的状态。

   ii. 在目标PolarDB-X数据库上通过CREATE SEQUENCE命令创建新的Sequence对象。

   Sequence命令详情请参见Sequence。

场景三：从PolarDB-X导出数据到MySQL

从PolarDB-X导出数据到MySQL，和在PolarDB-X之间相互导入数据的过程类似，也分为以下几个步骤。

1. 从源PolarDB-X中导出数据到文本文件。请参见场景一步骤一。
2. 登录MySQL，手动创建目标表。如果导出数据包含建表语句，则需要在导出文件中修改建表语句，删除PolarDB-X中不兼容MySQL的关键字等信息。

   例如PolarDB-X中的某个拆分表：

```
CREATE TABLE `table_name` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    ...
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 dbpartition by hash(`id`);
```

需去掉MySQL不兼容的拆分函数语句，改成：

```
CREATE TABLE `table_name` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    ...
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4;
```

3. 导入数据文件到PolarDB-X。请参见场景一步骤三。

7.5.4. 使用程序进行数据导入

本文将介绍如何通过编写代码的方式，将数据导入到PolarDB-X中。

创建如下一张表：

```
CREATE TABLE `test1` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    `c` char(120) NOT NULL DEFAULT '',
    `pad` char(60) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`),
    KEY `k_1` (`k`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 dbpartition by hash(`id`);
```

从数据库中导出源数据

源数据可以由用户自行生成，也可以从数据库中通过 `mysql -e` 命令的方式导出，PolarDB-X和MySQL都支持该方式，具体方法如下：

```
mysql -h ip  -P port -u usr -pPassword db_name -N -e "SELECT id,k,c,pad FROM test1;" >/home/data_1000w.txt
## 原始数据以制表符分隔，数据格式：188092293    27267211    59775766593-64673028018-...-09474402685    017050514
24-...-54211554755
mysql -h ip  -P port -u usr -pPassword db_name -N -e "SELECT id,k,c,pad FROM test1;" | sed 's/\t/,/g' >/home
/data_1000w.csv
## csv文件格式以逗号分隔，数据格式：188092293,27267211,59775766593-64673028018-...-09474402685,01705051424-...-5
4211554755
```

推荐对字符串进行处理，转变成csv文件格式，方便后续程序读取数据。

在PolarDB-X中创建目标表

源数据不包括建表语句，需要手动在PolarDB-X目标数据库上创建表，示例如下：

```
CREATE TABLE `test1` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    `c` char(120) NOT NULL DEFAULT '',
    `pad` char(60) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`),
    KEY `k_1` (`k`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 dbpartition by hash(`id`);
```

使用程序导入数据到PolarDB-X

您可以自行编写程序，连接PolarDB-X，然后读取本地数据，通过 `Batch Insert` 语句导入到PolarDB-X中。

下面是一个简单的JAVA程序示例：

```
// 需要mysql-connector-java.jar，详情界面：https://mvnrepository.com/artifact/mysql/mysql-connector-java
// 下载链接：https://repo1.maven.org/maven2/mysql/mysql-connector-java/8.0.20/mysql-connector-java-8.0.20.jar
// 注：不同版本的mysql-connector-java.jar，可能Class.forName("com.mysql.cj.jdbc.Driver")类路径不同
// 编译 javac LoadData.java
// 运行 java -cp .:mysql-connector-java-8.0.20.jar LoadData
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class LoadData {
    public static void main(String[] args) throws IOException, ClassNotFoundException, SQLException {
        File dataFile = new File("/home/data_1000w.csv");
        String sql = "insert into test1(id, k, c, pad) values(?, ?, ?, ?)";
        int batchSize = 1000;
        try (
            Connection connection = getConnection("ip", 3306, "db", "usr", "password");
            BufferedReader br = new BufferedReader(new FileReader(dataFile))) {
            String line;
            PreparedStatement st = connection.prepareStatement(sql);
            long startTime = System.currentTimeMillis();
            int batchCount = 0;
            while ((line = br.readLine()) != null) {
                String[] data = line.split(",");
                st.setInt(1, Integer.valueOf(data[0]));
                st.setInt(2, Integer.valueOf(data[1]));
                st.setObject(3, data[2]);
                st.setObject(4, data[3]);
                st.addBatch();
                if (++batchCount % batchSize == 0) {
                    st.executeBatch();
                    System.out.println(String.format("insert %d records", batchCount));
                }
            }
            if (batchCount % batchSize != 0) {
                st.executeBatch();
            }
            long cost = System.currentTimeMillis() - startTime;
            System.out.println(String.format("Take %d second，insert %d records, tps %d", cost/1000, batchCo
unt, batchCount/(cost/1000)));
        }
    }
    /**
     * 获取数据库连接
     *
     * @param host     数据库地址
     * @param port     端口
     * @param database 数据库名称
     * @param username 用户名
     * @param password 密码
     * @return
     * @throws ClassNotFoundException
     * @throws SQLException
     */
    private static Connection getConnection(String host, int port, String database, String username, String
password)
        throws ClassNotFoundException, SQLException {
        Class.forName("com.mysql.cj.jdbc.Driver");
        String url = String.format(
```

```
            "jdbc:mysql://%s:%d/%s?autoReconnect=true&socketTimeout=600000&rewriteBatchedStatements=true", h
ost, port,
            database);
        Connection con = DriverManager.getConnection(url, username, password);
        return con;
    }
}
```

您可以根据实际应用场景编写程序，设置合适的batch size和多线程导入，来提高导入的性能。

7.5.5. 使用Batch Tool工具导入导出数据

本文介绍了通过Batch Tool工具导入导出数据的方法。

工具介绍

Batch Tool工具是PolarDB-X团队开发的专为PolarDB-X数据库提供数据导入导出服务的工具。

Batch Tool工具采用JAVA语句实现，核心是生产者消费者模型，支持多线程操作，提供批量导出、批量导入、批量删除、批量更新等功能。数据以csv文件格式传输，方便用户进行数据交互。

Batch Tool工具的命令用法和参数如下：

```
usage: BatchTool [-?] [-batchsize <arg>] [-con <consumer count>] [-cs
           <charset>] -D <database> [-dir <directory>] [-f <from>] [-F
           <filenum>] [-func] -h <host> [-header] [-i] [-in] [-L <line>]
           [-lastSep] [-lb] [-local] [-maxConn <arg>] [-minConn <arg>]
           [-noesc] [-np] [-O <order by type>] -o <operation> [-OC <ordered
           column>] -p <password> [-P <port>] [-para] [-pre <prefix>] [-pro
           <producer count>] [-quote <auto/force/none>] [-readsize <arg>]
           [-rfonly] [-ringsize <arg>] -s <sep> [-t <table>] -u <user> [-w
           <where>]
     -?,--help                                Help message.
     -batchsize,--batchSize <arg>             Batch size of emitted tuples.
     -con,--consumer <consumer count>         Configure number of consumer
                                              threads.
     -cs,--charset <charset>                  Define charset of files.
     -D,--database <database>                 Database to use.
     -dir,--dir <directory>                   Directory path including files to
                                              import.
     -f,--from <from>                         Source file(s), separated by ; .
     -F,--filenum <filenum>                   Fixed number of exported files.
     -func,--sqlfunc                          Use sql function to update.
     -h,--host <host>                         Connect to host.
     -header,--header                         Whether the header line is column
                                              names.
     -H,--historyFile <filename>              history file name for resuming from breakpoint
     -i,--ignoreandresume                     Flag of insert ignore & resume from breakpoint
     -in,--wherein                            Using where ... in (...)
     -L,--line <line>                         Max line limit of exported files.
     -lastSep,--withLastSep                   Whether line ends with separator.
     -lb,--loadbalance                        If using load balance.
     -local,--localmerge                      o local merge sort.
     -maxConn,--maxConnection <arg>           Max connection number limit.
     -minConn,--minConnection <arg>           Mim connection number limit.
     -noesc,--noescape                        Don't escape values.
     -np,--noparttion                         No use of parttion.
     -O,--orderby <order by type>             asc or desc.
     -o,--operation <operation>               Batch operation type: export /
                                              import / delete / update.
     -OC,--orderCol <ordered column>          col1;col2;col3.
     -p,--password <password>                 Password to use when connecting to
                                              server.
     -P,--port <port>                         Port number to use for connection.
     -para,--paraMerge                        Using parallel merge when doing
                                              order by export.
     -pre,--prefix <prefix>                   Export file name prefix.
     -pro,--producer <producer count>         Configure number of producer
                                              threads.
     -quote,--quoteMode <auto/force/none>     The mode of how field values are
                                              enclosed by double-quotes when
                                              exporting table. Default value is
                                              auto.
     -readsize,--readSize <arg>               Read block size in MB.
     -rfonly,--rfonly                         Only read and process file, no sql
                                              execution.
     -ringsize,--ringBufferSize <arg>         Ring buffer size.
     -s,--sep <sep>                           Separator between fields
                                              (delimiter).
     -t,--table <table>                       Target table.
     -tps,--tpsLimit <arg>                    Tps limit
     -u,--user <user>                         User for login.
     -w,--where <where>                       Where condition: col1>99 AND
                                              col2<100 ...
```

参数说明

常用参数说明如下：

- -o：批处理操作，包括export、import、delete、update四个选项。
- -t：指定目标表名，只能为单个表。
- -s：指定分隔符，可以为字符或字符串。
- -f：指定源文件，多个文件名之间使用分号“;”分隔。
- -OC：指定导出时排序使用的列名，多个列之间使用分号“;”分隔。
- -cs：指定文本文件的字符集，默认为utf-8。
- -lastSep：文件每行是否以分隔符结尾。
- -quote：指定导出或导入时引号包围模式，包括以下三个可选值：
  - auto：默认模式，将根据字段值是否包含特殊字符（如分隔符、换行符等）来添加双引号；
  - force：强制每个字段值都添加双引号；
  - none：强制不添加双引号（适用于已知表字段类型都是数值型、或字符串型字段中不包含特殊字符的情况）。
- -header：首行是否为字段名。
- -i：是否开启insert ignore与断点续传。
- -pre：指定导出文件名的前缀。
- -F：指定导出文件数量。

工具获取

Batch Tool工具的jar包，单击下载：Batch_tool工具。

使用示例

以编译生成的batch-tool.jar为例，查看参数说明：

```
java -jar batch-tool.jar -?
```

- 批量导出数据

```
## 1.默认导出（文件数等于表的分片数）
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o export -t table_name
-s ,
## 2.导出文件数=3 （-F:指定文件数）
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o export -t table_name
-s , -F 3
## 3.指定单个文件最大行数=10000 （-L:指定单文件行数）
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o export -t table_name
-s , -L 10000
## 4.带where条件 若条件带有空格，需要使用引号 （-w:where条件语句）
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o export -t table_name
-s , -w "id < 150 and id > 120"
```

- 批量导入数据（需手动创建目标表，Batch Tool只包含数据传输）

```
## 1.多个文件用分号 （;） 分隔
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o import -t table_name
-s , -f "table_name_0;table_name_1;table_name_2;table_name_3"
## 2.默认根据拆分键进行sharding插入，如不采用，打开-np开关即可
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o import -t table_name
-s , -f "file0;file2" -np
## 3.指定生产者、消费者线程（-pro:生产者线程，读取文件线程；-con:消费者线程，导入数据线程）
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o import -t table_name
-s , -f "table_name_0;table_name_1" -np -pro 16 -con 16
## 4.打开insert ignore和断点续传
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o import -t table_name
-s , -f "table_name_0;table_name_1" -i
```

- 批量删除数据（删除数据库中包含文件中的数据，原理：构建DELETE语句，根据表结构填充文件中的数据）

```
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o delete -t table_name
-s , -f "file0"
```

- 批量更新数据（更新数据库中包含文件中的数据，原理：构建UPDTATE语句，根据表结构填充文件中的数据）

```
java -jar batch-tool.jar -h 127.0.0.1 -u testUser -p testPasswd -P 3306 -D db_name -o update -t table_name
-s , -f "file0"
```

7.6. 数据迁移和同步

7.6.1. PolarDB-X 1.0升级至PolarDB-X 2.0

PolarDB-X提供了评估升级工具，帮助您将数据库从PolarDB-X 1.0迁移至PolarDB-X 2.0。

概述

PolarDB-X推出的评估升级功能，面向PolarDB-X 1.0用户提供一站式升级评估和迁移服务，涵盖数据库迁移全生命周期，包含成本、规格、新老版本兼容性及数据迁移同步与一致性校验、割接支持与性能优化。

PolarDB-X评估升级功能可以安全平滑地将PolarDB-X 1.0升级到2.0最新版本，结合云端丰富的数据平台产品特性以及用户业务特性给出数据库和应用改造方案，助力企业的源数据库迁移上云。

费用

评估升级功能可免费使用。

相关概念

- 源实例：即要升级的PolarDB-X 1.0实例。
- 源数据库（源库）：即源实例中要进行迁移的数据库。
- 目标实例：要升级迁移至的PolarDB-X 2.0实例。
- 目标数据库（目标库）：即目标实例中创建好的用于迁移至的数据库。

注意事项

- 当前评估升级功能以面向呼和浩特和中国香港地域发布。其他地域若有升级需求，请提交工单联系阿里云技术支持。
- 当前仅PolarDB-X 1.0 V5.4.9及以上的实例支持升级至PolarDB-X 2.0。
- 当前评估升级工具支持的是库级别的数据迁移功能。每个升级任务仅包含一个数据库的数据迁移。
- 开始升级后，将会对全量数据进行迁移。该过程对源库的资源有一定的消耗，因此建议在对应业务的低峰期进行。
- 由于PolarDB-X 1.0不支持全局Binlog，因此该升级过程中不支持迁移源库中由DDL操作造成的数据变动。若您在升级过程中有DDL需求，请提交工单联系阿里云技术支持。

升级前准备

在开始评估升级前，请在源实例中：

- 将源实例底层RDS的binlog_format参数设置为ROW，binlog_row_image设置为FULL。否则预检查阶段会提示报错，且无法成功启动数据迁移任务。
- 待迁移的表需具有主键约束或非空唯一键约束，否则预检查阶段会提示报错，且无法成功启动数据迁移任务。
- 根据业务情况在源实例库表中删除SIMPLE sequence（若有）。

此外，请使用**相同阿里云账号**在**相同地域**购买PolarDB-X 2.0实例作为目标实例，且：

- 已在目标实例中创建账号。
- 已在目标实例中创建数据库作为目标数据库，且目标数据库的字符集与源数据库的字符集保持一致。

  例如，若您PolarDB-X 1.0实例中的数据库字符集为utf8mb4，请在PolarDB-X 2.0实例中创建好字符集同为utf8mb4的数据库。

> **说明** 您可以执行如下命令，查看PolarDB-X 1.0实例中数据库的字符集。
>
> ```
> SELECT default_character_set_name FROM information_schema.SCHEMATA WHERE schema_name = '...'
> ```

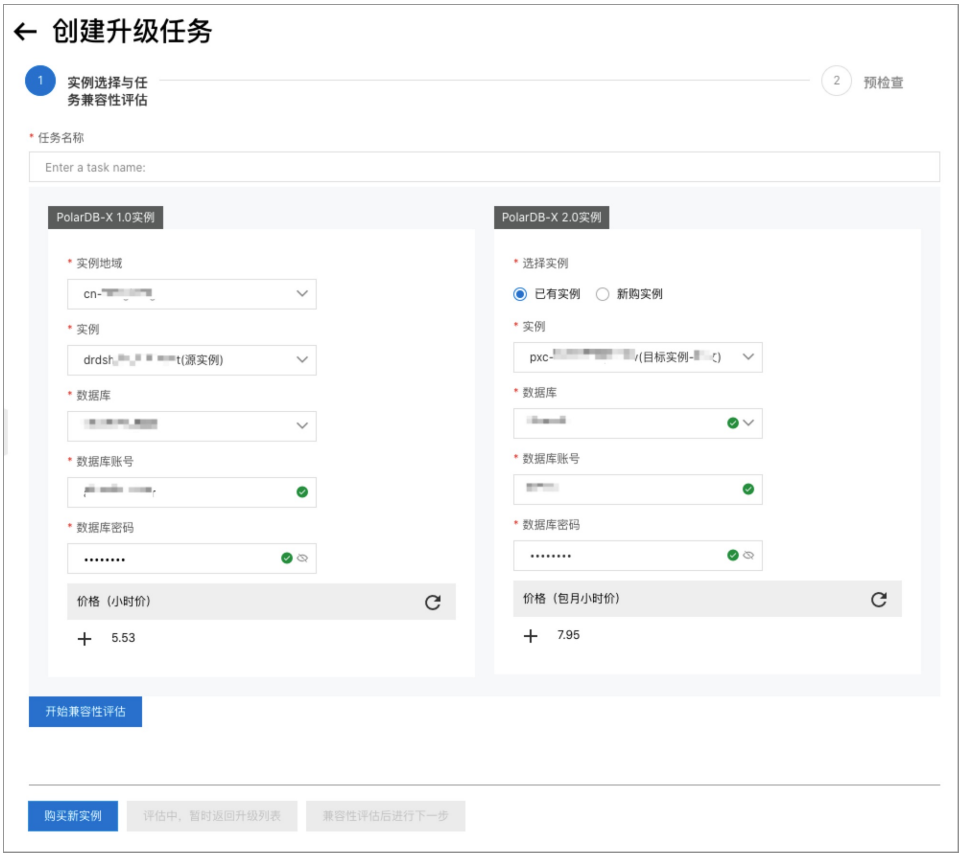
- 创建的目标数据库需授权给创建的账号，确保账号有目标数据库的读写权限。
- 迁移过程中，评估升级工具不会覆盖目标数据库中已有的表。但请确认目标数据库中不存在与源数据库中的表相同名称的表。否则预检查阶段会提示报错，且无法成功启动数据迁移任务。

步骤1：创建升级任务

> **说明**
>
> 评估升级工具提供了库级别的数据迁移功能。每个升级任务仅包含一个数据库的数据迁移。因此若您有多个数据库，请创建多个升级任务。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 右侧导航栏单击**评估升级**。
5. 单击**创建升级任务**。
6. 在打开的页面中，填写如下内容。

配置项		说明
任务名称		为本次评估与升级任务自定义一个名称。
PolarDB-X 1.0实例	实例地域	选择源PolarDB-X 1.0实例所在的地域。 源PolarDB-X 1.0实例与目标PolarDB-X 2.0实例需要位于同一地域下。
	实例	选择源PolarDB-X 1.0实例。 当选定实例地域后，该地域下的所有PolarDB-X 1.0的实例将展示在**实例**下拉列表中。
	数据库	选择PolarDB-X 1.0实例下要迁移的源数据库。 当选定实例后，该实例下的所有数据库将展示在**数据库**下拉列表中。
	数据库账号、数据库密码	对已选定的数据库拥有读写权限的账号及对应的密码。
PolarDB-X 2.0实例	选择实例	选择与PolarDB-X 1.0实例处于同一地域下的PolarDB-X 2.0实例。 当选定源实例地域后，该地域下所有PolarDB-X 2.0实例将展示在**实例**下拉列表中。 **说明** 您也可选择**新购实例**，并设置想要购买的实例的规格和节点数，了解对应的价格。
	数据库	选择PolarDB-X 2.0实例下要迁移至的目标数据库。 当选定实例后，该实例下的所有数据库将展示在**数据库**下拉列表中。
	数据库账号、数据库密码	对已选定的数据库有读写权限的账号及对应的密码。

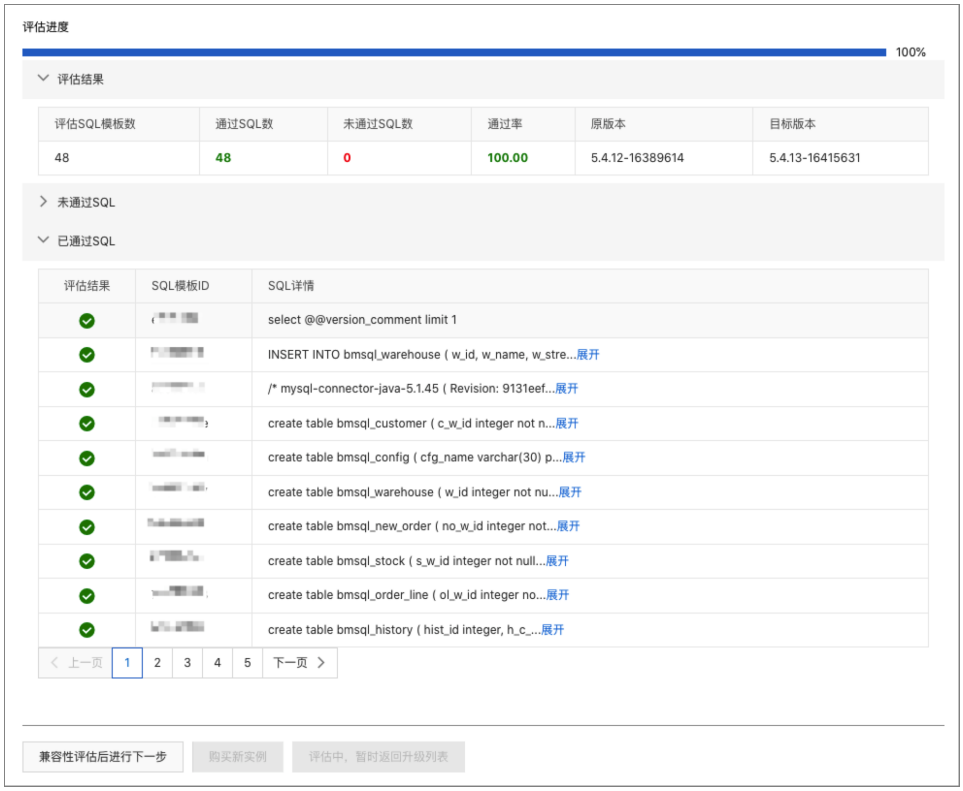


7. 配置完源PolarDB-X 1.0实例和目标PolarDB-X 2.0实例后，您可以了解1.0和2.0实例的价格区别。
8. 价格确认无误后，单击**开始兼容性评估**，进入下一步。

步骤2：兼容性评估

升级任务创建完成后，将进行SQL兼容性评估。由于PolarDB-X 1.0与2.0的SQL语法略有差异，因此需要进行兼容性评估，评估源库的SQL语句是否完全兼容PolarDB-X 2.0的SQL语法。若不兼容，则需要对其进行针对性的改造。

当单击**开始兼容性评估**后，评估升级工具将开始评估源库和目标库的兼容性情况。



若存在未通过的SQL，您可根据PolarDB-X 2.0的SQL语法说明进行SQL改造。您也可先进行下一步，然后在迁移开始并校验完成后，对相关SQL进行改造。

步骤3：预检查

兼容性评估完成后，单击**兼容性评估后进行下一步**，进入预检查阶段。

预检查项有：

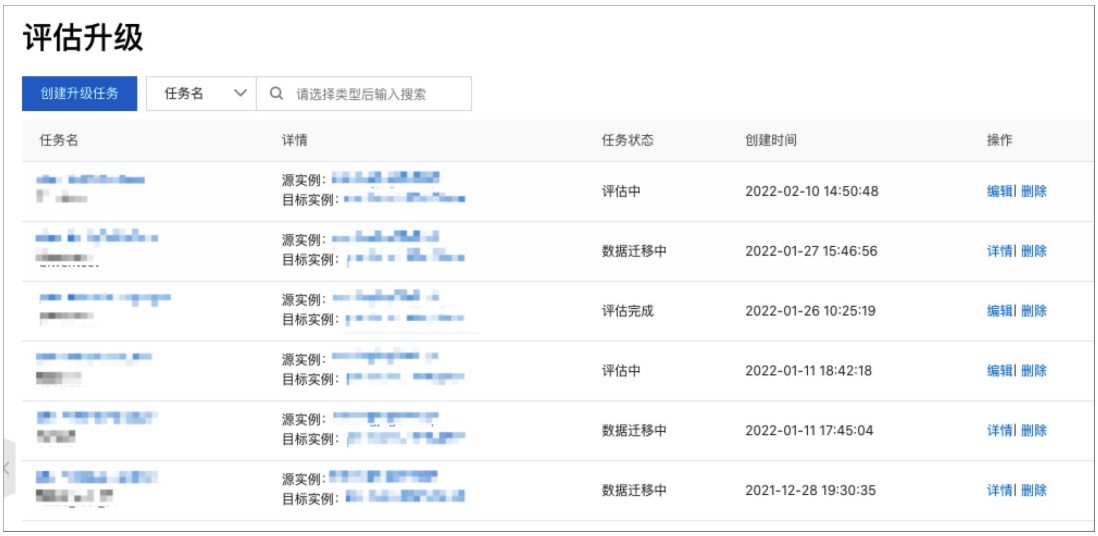
- **CONNECTIVITY**：检查源实例和目标实例的用户名和密码是否正确，以及两者的网络连通性。若失败，请检查实例的用户名和密码是否正确。
- **BINLOG_CONFIG**：检查源实例底层RDS的binlog设置。若失败，请确保已将源实例底层RDS的binlog_format参数设置为ROW，binlog_row_image设置为FULL。
- **TABLE_META**：检查源实例库表的主键情况。若失败，请检查源实例待迁移的表，确保其需具有主键约束或非空唯一键约束。
- **CONFLICT**：检查目标库中是否已存在与源库中同名的表。若失败，请修改目标库中对应同名表的名称。
- **SEQUENCE**：检查源库是否存在SIMPLE sequence。由于PolarDB-X 2.0不支持SIMPLE Sequence，因此需要在源库根据业务情况删除此Sequence。

请确保所有预检查项均通过，否则无法开始升级。

步骤4：开始升级

1. 当预检查的所有结果均通过后，单击**开始升级**。

   开始升级后，请返回升级任务列表，查看任务状态。

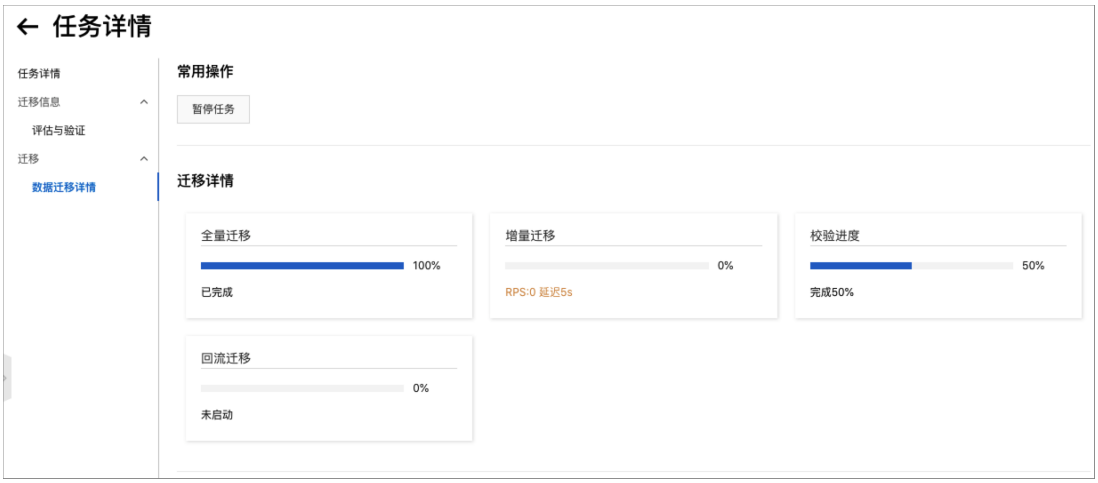


2. 开始升级后，任务状态将变为**迁移准备中**，包括建表和一些资源的准备。

   若出现错误提示，或者一直处于该状态下无法开始迁移，请提交工单联系阿里云技术支持。

3. 迁移准备完成后，任务状态将变为**数据迁移中**。该阶段中将对全量数据和增量数据进行迁移。

   当任务状态将变为**数据迁移中**后，单击**操作**列下的**详情**，并进入**数据迁移详情**页，查看具体的状态。



   i. 首先开始**全量迁移**。开始迁移后，源库中待迁移对象的存量数据，将全部迁移到目标库中。

   ii. 全量迁移的进度达到100%后，将立即自动开始**增量迁移**，将全量迁移开始后源库中更新的数据持续同步至目标库。

   增量迁移是一个动态持续进行的过程，在业务正式切换到目标库以前，是不会停止的。

   iii. 增量迁移开始后，当源库和目标库中的数据大致追平后（表现为增量迁移的延迟在30秒以内），将自动开始**数据校验**，校验源库和目标库中的数据一致性，确保两端数据是动态一致的。

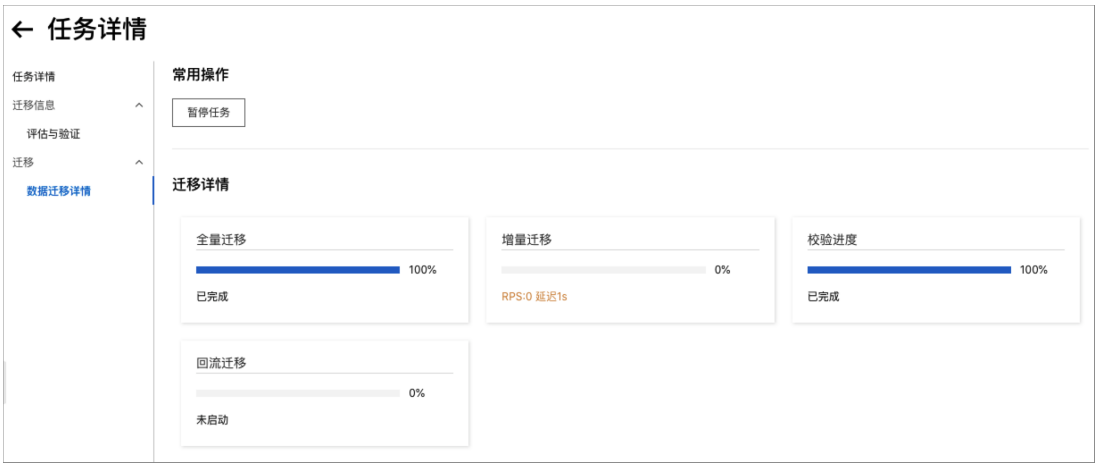
> **说明** 以上三种迁移状态若出现错误提示，或者长时间没有进度变化，请提交工单联系阿里云技术支持。

4. 当**数据校验**进度达到100%后，前往**评估与验证**页，查看是否有评估未通过的SQL，并进行针对性改造。

步骤5：开启回流迁移，并切换业务至PolarDB-X 2.0

当增量迁移的延迟在10秒内，且数据校验达到100%，SQL兼容性评估全部通过后，表明源库和目标库的数据除增量迁移的延迟外，已经达到一致。

增量迁移的延迟小于10秒且校验进度达到100%



SQL兼容性评估全部通过

当业务应用连接的数据库的升级任务都达到以上状态时，可将业务从源实例切换至目标实例。

业务切换前，为了防止风险的发生（例如发现迁移的数据有误、新灌入的数据不符合预期等），建议开启**回流迁移**功能，将切换后新写入目标数据库的业务数据同步到源数据库，从而及时实现可逆的操作。

> **说明** 当前您需要提交工单申请开启回流迁移。

开启回流迁移后，可按照如下步骤将业务的应用连接从源实例切换至目标实例：

> **警告** 切换操作会导致业务中断，直至重新连接至目标实例后恢复。请在业务低峰期进行切换。

1. 停止应用，检查源数据库是否还有连接，保证源数据库连接已经完全停止。

   > **说明**
   > - 您可以检查源数据库的会话数（连接数），当会话值逐渐降低为0时，表示应用已经完全停止。
   > - 请确保源库和目标库都无回流迁移之外的同时写入，包括触发器、任务、业务写入等，否则无法保证两侧数据一致性。
   > - 由于增量迁移存在一定的延迟（10秒内），因此为了保证源库和目标库的数据完全一致，应用的停止时间需长于该延迟时间。

2. 修改应用内的连接地址配置。您可前往目标实例的**基本信息**页获取新的连接地址信息。
3. 重新启动应用，观察目标数据库的连接情况。

   当切换至PolarDB-X 2.0实例后，新的业务数据将灌入对应数据库中。若您开启了回流迁移功能，切换后新产生的数据将同时增量同步至PolarDB-X 1.0实例对应的数据库中。

7.7. 参数设置

PolarDB-X支持查看计算层和存储层的相关参数说明，您还可以在控制台上自定义修改这些参数值。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**配置与管理 > 参数设置**，您可以选择修改**计算层**或**存储层**的参数。

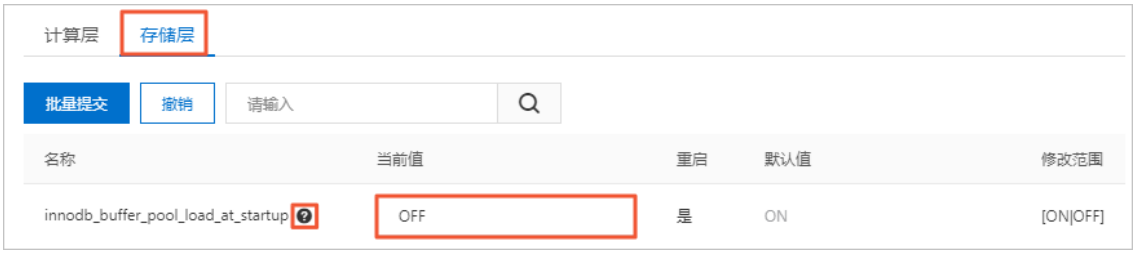
   修改计算层参数

   i. 在**计算层**页签下找到目标参数，在**当前值**对话框中输入新的参数值。

   修改存储层参数

   i. 单击**存储层**页签。

   ii. 找到目标参数，在**当前值**对话框中输入新的参数值。

   > **说明**
   > - 请输入目标参数右侧**修改范围**栏规定的参数值，否则当您**批量提交**时会出现错误提示。
   > - 您可以单击参数名称后面的图标查看目标参数的详细说明。
   > - 全量参数说明请参见变量。

6. 单击页面左上角**批量提交**。

   > **说明** 对于是否重启栏显示为是的参数，**批量提交**新的**运行参数值**后，您需要重启实例新参数值才会生效。

7. 在弹出的对话框中单击**确定**即可。

7.8. 监控与告警

7.8.1. 计算资源监控

为方便您掌握实例的运行状态，PolarDB-X提供监控查询功能。您可以在控制台上查看计算资源监控和存储资源监控信息。其中计算资源监控展示了实例计算层资源的性能数据，本文将介绍如何查看计算资源监控信息。

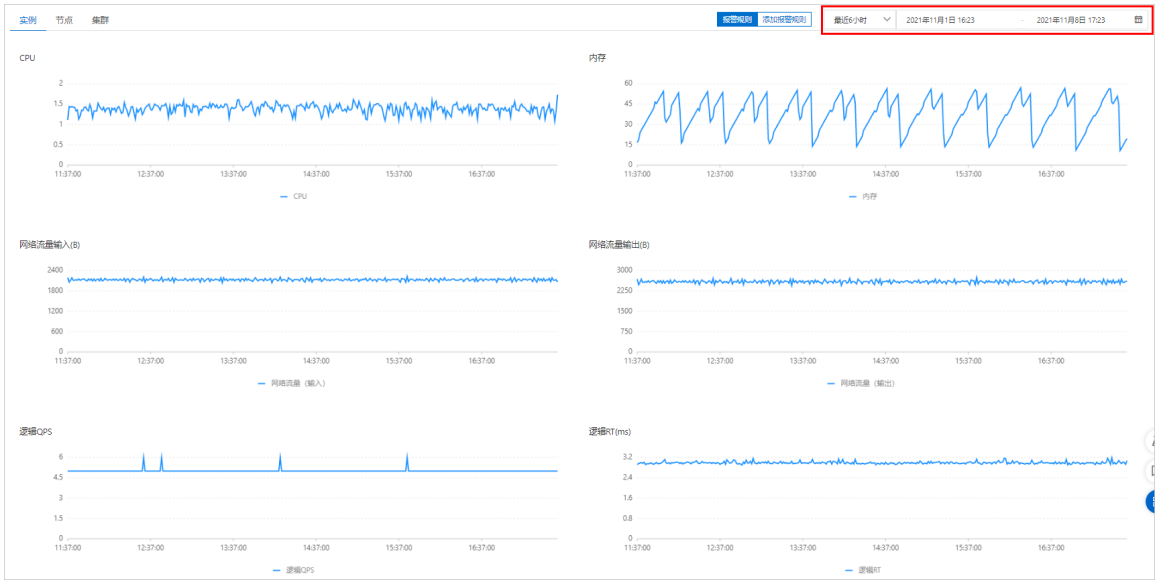
操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**监控与报警 > 计算资源监控**。
6. 在**计算资源监控**页面，您可以查看实例、节点或集群的以下监控项。

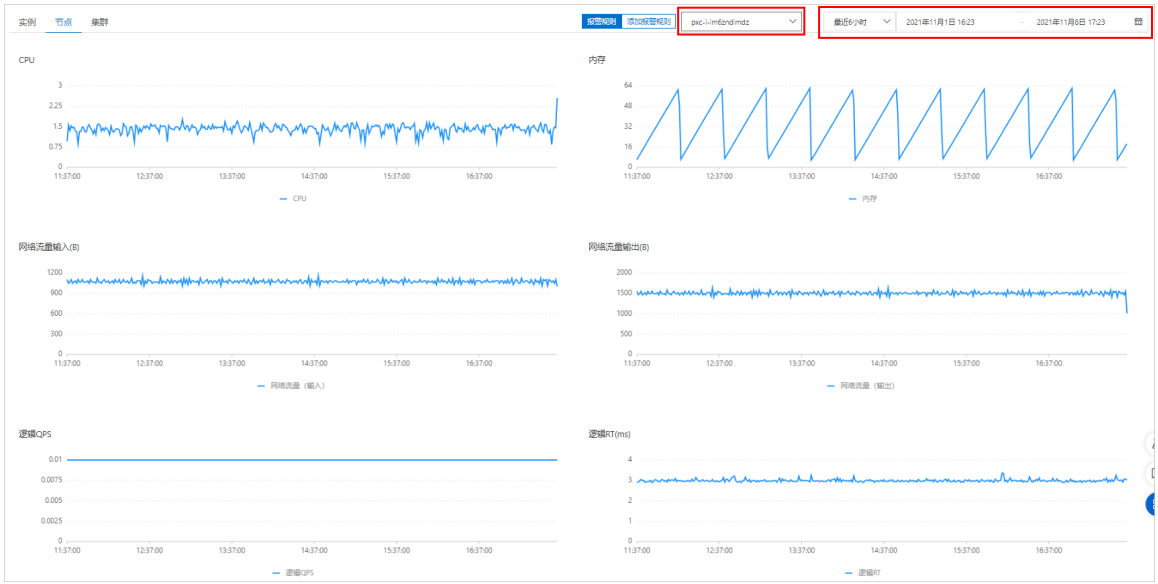
监控项	指标	说明
CPU	cpu	实例计算节点CPU使用率的平均值。
内存	old	实例计算节点JVM Old Generation的内存使用率。内存使用率波动属于正常现象。
网络流量输入	netin	实例计算节点的网络输入流量的总和，单位为Kbps。存储节点返回数据到计算节点，会产生网络输入流量。
网络流量输出	netout	实例计算节点的网络输出流量的总和，单位为Kbps。计算节点发送物理SQL到存储节点，计算节点返回数据到应用，均会产生网络输出流量。
逻辑QPS	lqps	实例每秒处理的SQL语句数目的总和。
逻辑RT	lrt	实例对于每条SQL的平均响应时间。
前端连接数	con	应用到实例的连接总数。
活跃线程数	active	实例中用来执行SQL的线程数。

> **说明** 您可以自定义上述监控项的采集周期，最多支持查询7天内的监控数据。

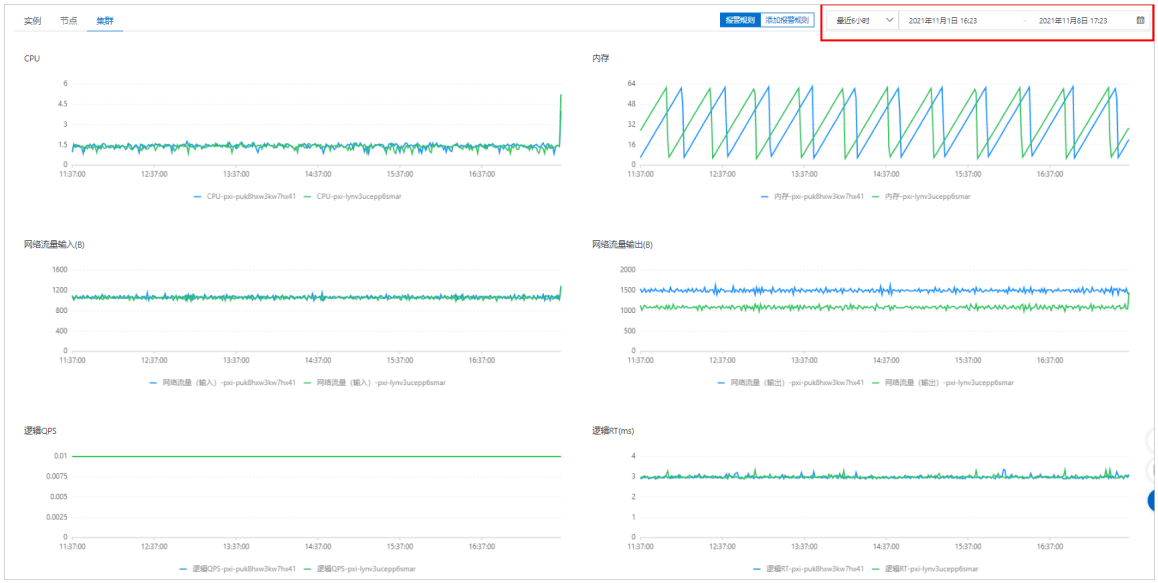
- 查看实例性能监控：单击**实例**页签，设置查询的时间段，单击**确定**。



- 查看节点性能监控：单击**节点**页签，从页面中间的下拉框中选择目标节点，并设置查询的时间段，单击**确定**。



- 查看集群性能监控：单击**集群**页签，设置查询的时间段，单击**确定**。



7.8.2. 存储资源监控

为方便您掌握实例的运行状态，PolarDB-X提供监控查询功能。您可以在控制台上查看计算资源监控和存储资源监控信息。其中存储资源监控展示了实例存储层资源的性能数据，本文将介绍如何查看存储资源监控信息。

操作步骤

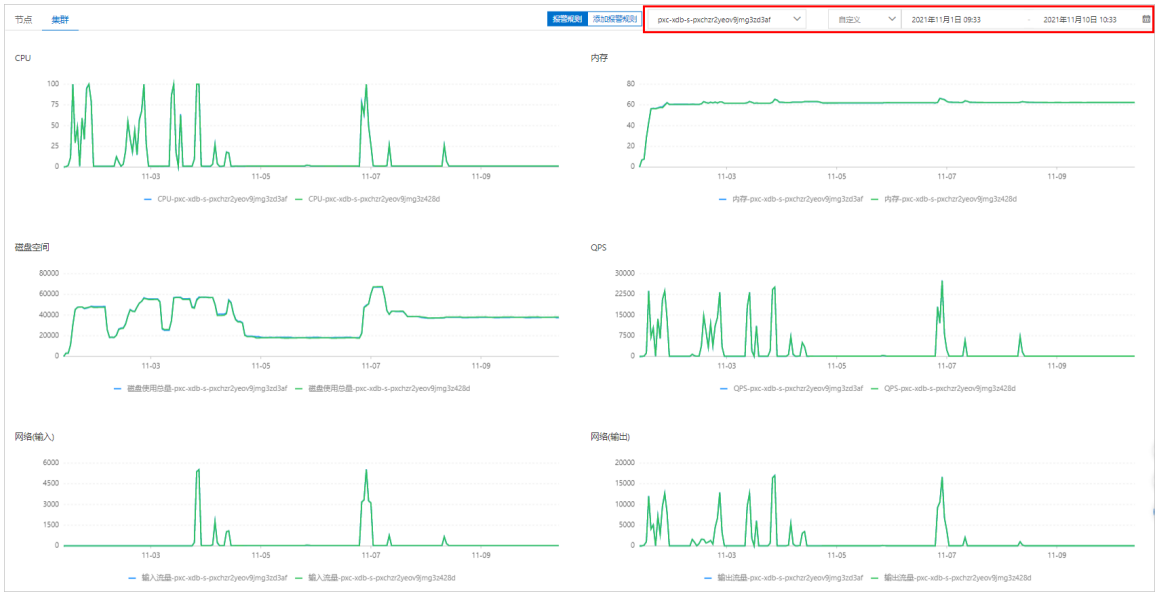
1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**监控与报警 > 存储资源监控**。
6. 在**存储资源监控**页面，您可以查看节点或集群的以下监控项。

监控项	指标	说明
CPU	cpu	CPU使用率的平均值。
内存	mem	JVM Old Generation的内存使用率。内存使用率波动属于正常现象。
磁盘空间	ins_size	磁盘空间总使用量，单位为MByte。
	data_size	数据空间使用量，单位为MByte。
	log_size	实例存储节点的日志空间使用量，单位为MByte。
	tmp_size	临时文件空间使用量，单位为MByte。
	other_size	系统文件空间使用量，单位为MByte。
网络流量	recv_k	每秒输入流量，单位为Kbps。
	sent_k	每秒输出流量，单位为Kbps。
连接数	active_session	当前活跃连接数。
	total_session	当前总连接数。
IOPS	iops	每秒读写次数，包括每秒读写总次数、每秒读次数、每秒写次数。

监控项	指标	说明
QPS	qps	每秒请求数。
TPS	tps	每秒事务数。
内存缓冲池	ibuf_dirty_ratio	缓冲池的脏块率。
	ibuf_read_hit	缓冲池的读命中率。
	ibuf_use_ratio	缓冲池的利用率。
InnoDB读写数量	inno_data_read	实例存储节点每秒从存储引擎读取的数据量。
	inno_data_written	实例存储节点每秒往存储引擎写入的数据量。
InnoDB缓冲池请求数	ibuf_request_r	缓冲池的每秒读次数。
	ibuf_request_w	缓冲池的每秒写次数。
InnoDB日志写	Innodb_log_write_requests	实例存储节点日志的每秒写请求数量和每秒同步数据到磁盘的次数。
	Innodb_os_log_fsyncs	实例存储节点日志的每秒同步数据到磁盘的次数。
临时表	tb_tmp_disk	实例存储节点每秒创建的临时表数量。
操作	com_delete	每秒Delete数。
	com_insert	每秒Insert数。
	com_insert_select	每秒Insert_Select数。
	com_replace	每秒Replace数。
	com_replace_select	每秒Replace_Select数。
	com_select	每秒Select数。
	com_update	每秒Update数。

- 查看节点监控项：单击**节点**页签，在下拉框中选择目标存储节点并设置查询的时间段，单击**确定**。

- 查看集群监控项：单击**集群**页签，在下拉框中选择目标集群并设置查询的时间段，单击**确定**。



7.8.3. 配置告警

您可以在控制台上配置计算资源监控指标和存储资源监控指标的告警规则。本文将介绍如何配置实例的告警规则。

背景信息

为方便您掌握实例的运行状态，PolarDB-X对接云监控提供告警配置功能。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**监控与报警 > 计算资源监控**或者**监控与报警 > 存储资源监控**。
6. 点击页面上方的**设置报警规则**按钮，前往云监控页面配置对应指标的告警规则。关于告警规则配置，请参考云监控文档：创建报警规则。

7.9. 诊断与优化

7.9.1. SQL洞察和审计

7.9.1.1. 搜索

搜索功能可查询并导出SQL语句对应的各种信息，如数据库、状态、执行时间等。本文介绍如何使用搜索功能。

前提条件

实例已开启DAS专业版，详情请参见DAS专业版。

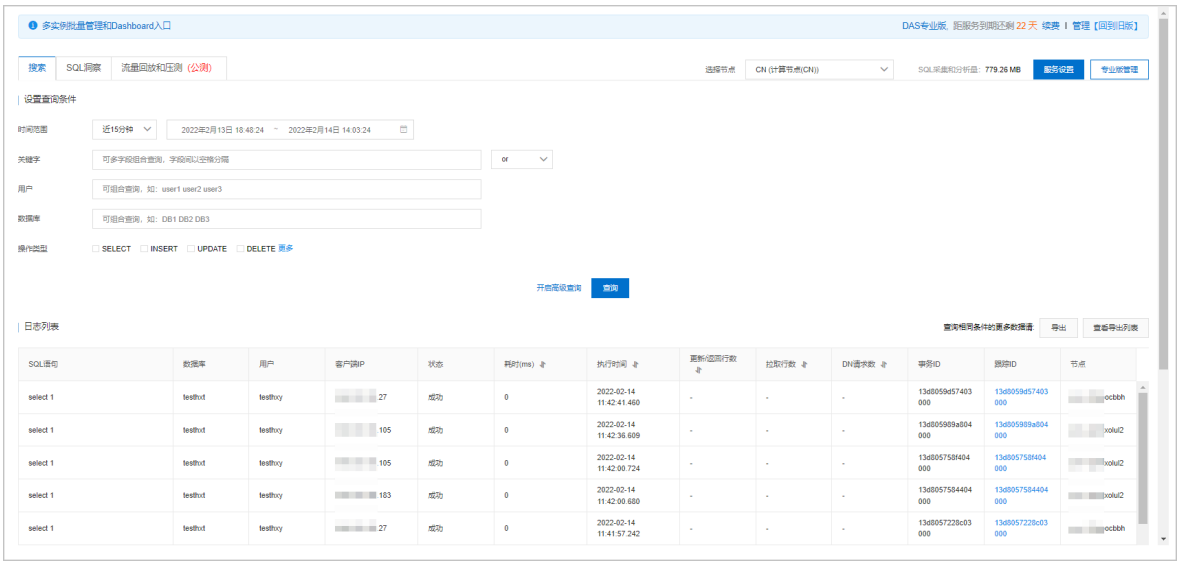
操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > SQL洞察与压测（新版）**。
6. 在右侧页面单击**搜索**页签，设置如下参数作为查询条件。

参数	说明
时间范围	根据实际需要选择查询的时间范围。 在线查询时间范围最多为24小时。这是因为SQL洞察记录所有数据库行为，会记录大量SQL语句，在线查询选择时间范围过大，会导致长时间没有返回查询结果，甚至查询超时。 **说明** 如果需要查询更大时间范围的SQL记录，请您导出后进行查询。导出功能会异步导出日志，适合大时间范围内的查询。
关键字	可多字段组合查询，不支持模糊查询。字段间以空格分隔。 **说明** 查询的关键字至少包含4个字符。
用户	可组合查询用户名，如：user1 user2 user3。
数据库	可组合查询数据库名，如：DB1 DB2 DB3。
操作类型	选择需要查询的操作类型，可多选。

> **说明** 您可**开启高级查询**输入更多信息以缩小搜索范围。

7. 单击**查询**，即可看到**日志列表**中**SQL语句**的具体信息。



> **说明**
>
> - 单击**导出**，勾选需要的**导出字段**来**导出SQL日志记录**，通过设置**导出时间范围**可以导出更大时间范围的SQL记录。
> - 单击**查看导出列表**，可查看最近导出任务列表。

7.9.1.2. SQL洞察

SQL洞察功能可对SQL进行健康情况诊断和性能问题排查。本文介绍了SQL洞察的操作步骤。

前提条件

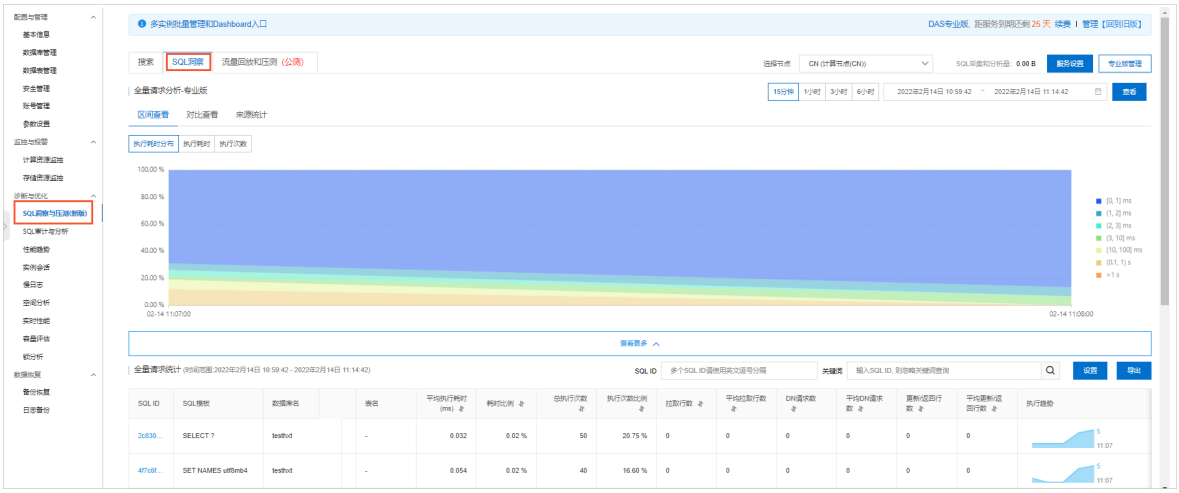
实例已开启DAS专业版，详情请参见DAS专业版。

背景信息

开启SQL洞察功能可以记录所有DQL、DML和DDL操作信息，这些信息通过数据库内核输出，对系统CPU消耗极低。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > SQL洞察与压测（新版）**。
6. 在右侧页面单击**SQL洞察**页签。



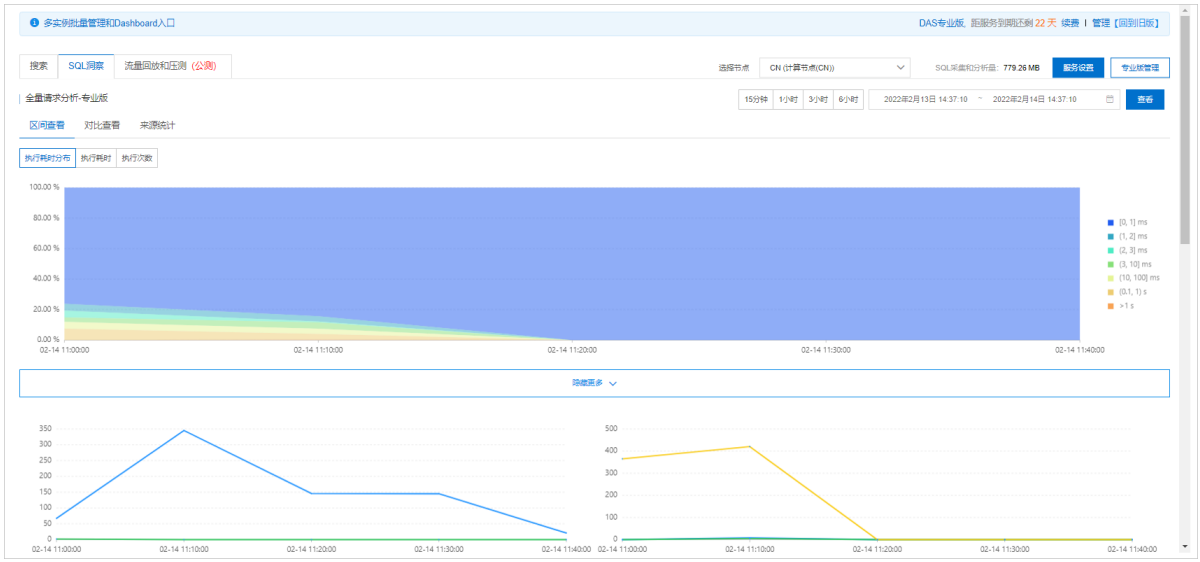
7. 根据需要选择对应的功能项：
   - **区间查看**：选择需要查看SQL洞察结果的时间范围，您可以查看选定时间范围内所有SQL的**执行耗时分布**、**执行耗时**、**执行次数**。您还可以在**全量请求统计**区域内查看选定时间范围内所有SQL的详细信息，并可以将其导出到本地。

     > **说明** 单次最多可导出1000万行数据。

   - **对比查看**：选择需要对比SQL洞察结果的时间点，您可以查看所有SQL**执行耗时分布**、**执行耗时**、**执行次数**的对比结果。您还可以在**请求对比列表**区域内查看详细的对比结果。
   - **来源统计**：选择需要统计SQL来源的时间范围，你可以查看选定时间范围内所有SQL的来源信息。

SQL洞察信息说明

- **执行耗时分布**：展示了选定时间范围内，所有SQL的执行耗时分布，将执行耗时分为了7个区间，每分钟计算一次，分别是：
  - `[0,1]ms` 是指满足`0ms ≤ 执行耗时 ≤ 1ms` SQL执行次数占比。
  - `(1,2]ms` 是指`1ms < 执行耗时 ≤ 2ms` SQL执行次数占比。
  - `(2,3]ms` 是指`2ms < 执行耗时 ≤ 3ms` SQL执行次数占比。
  - `(3,10]ms` 是指`3ms < 执行耗时 ≤ 10ms` SQL执行次数占比。
  - `(10,100]ms` 是指`10ms < 执行耗时 ≤ 100ms` SQL执行次数占比。
  - `(0.1,1]s` 是指`0.1s < 执行耗时 ≤ 1s` SQL执行次数占比。
  - `>1s` 是指`1s < 执行耗时` SQL执行次数占比。



> **说明** 若一个实例的**执行耗时分布**越接近蓝色，代表这个实例的SQL健康情况越好，越接近橙色和红色，代表这个实例的SQL健康情况越差。

- **执行耗时**：通过执行耗时，可以方便地查看在选定的时间范围内，SQL的执行耗时。
- **请求对比列表**：**对比查看**页签的**请求对比列表**展示了选定时间范围内每一类SQL的SQL文本、耗时比例、平均执行耗时和执行趋势等信息。

> **说明** 耗时比例 =（该类SQL执行耗时 × 执行次数）/（所有SQL执行耗时 × 总执行次数）× 100%。耗时比例越高的SQL，占用资源越多。

- **SQL ID**：单击**SQL ID**，您可查看该类SQL性能趋势、SQL样本等。
- **SQL样本**：通过**SQL样本**可查看该SQL是由哪个应用端发起的。

> **说明** SQL样本的编码采用UTF-8字符集。

7.9.2. SQL审计与分析

7.9.2.1. 简介

PolarDB-X联合日志服务推出SQL审计与分析功能，将SQL审计日志投递到日志服务中，实现了日志的实时查询、可视化分析、告警、投递、加工等操作。本文介绍SQL审计日志相关的信息。

功能优势

- 操作简单：仅需在控制台上打开功能开关，即可实时进行SQL日志的审计与分析。
- 性能无损：日志服务能够实时拉取PolarDB-X实例下各数据库的SQL审计日志文件，对实例本身性能无影响。
- 实时分析：依托日志服务，提供实时日志分析、开箱即用的报表中心和下钻分析的功能，帮助您准确地了解数据库执行状况、性能和安全问题。
- 实时告警：支持基于特定指标定制实时的监测与告警，当关键业务出现异常时可通过多种途径通知告警。

注意事项

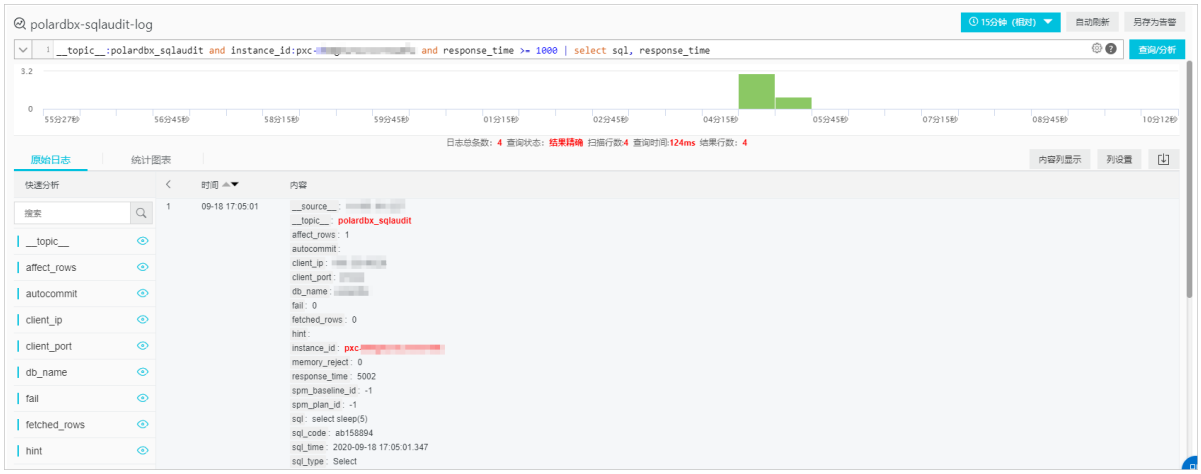
- SQL审计日志默认保存45天。
- 日志服务会不定期更新SQL日志审计功能，专属日志库的索引与默认报表也会自动更新，请勿在日志服务控制台上随意删除或修改PolarDB-X的Project、Logstore、索引和仪表盘等配置。
- PolarDB-X SQL审计与分析功能本身免费使用，但日志服务会对存储空间、读取流量、请求数量、数据加工、数据投递等进行收费，详情请参见日志服务产品定价。
- SQL审计与分析功能默认关闭，您可以在PolarDB-X控制台上随时开启SQL审计与分析。
- 功能关闭后将不再向日志服务写入数据，但您仍可以查看历史数据，这部分历史数据会产生存储和索引费用。您可以删除所有数据来停用日志服务，日志停止服务后将不再计费。

应用场景

- 问题SQL排查

  开启SQL审计与分析功能后，您可以通过快速检索SQL日志排查定位问题。例如您可以使用如下命令查询响应时间超过1s的所有SQL：

```
__topic__:polardbx_sqlaudit and instance_id:pxc-*************** and response_time >= 1000 | select sql, response_time
```

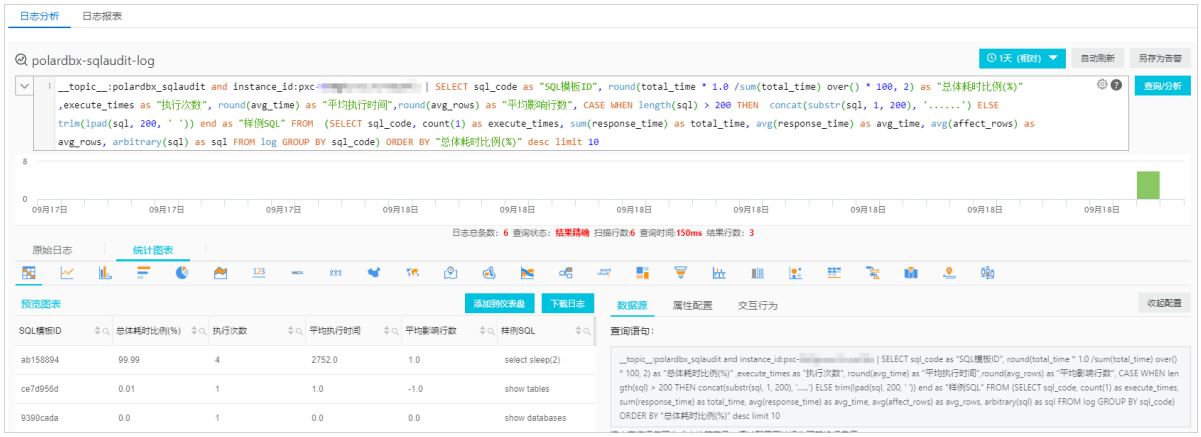


如上图所示，您可以在查询结果查看目标SQL的执行时间、执行该SQL的客户端IP地址等信息。

- 高代价SQL模板分析

  大多数应用中，SQL通常是基于若干模板动态生成的，只是参数不同。例如，您可以使用如下命令获取目标数据库中的高代价SQL模板信息。

```
__topic__:polardbx_sqlaudit and instance_id:pxc-**************** | SELECT sql_code as "SQL模板ID", round(total_time * 1.0 /sum(total_time) over() * 100, 2) as "总体耗时比例(%)" ,execute_times as "执行次数", round(avg_time) as "平均执行时间",round(avg_rows) as "平均影响行数", CASE WHEN length(sql) > 200 THEN  concat(substr(sql, 1, 200), '......') ELSE trim(lpad(sql, 200, ' ')) end as "样例SQL" FROM  (SELECT sql_code, count(1) as execute_times, sum(response_time) as total_time, avg(response_time) as avg_time, avg(affect_rows) as avg_rows, arbitrary(sql) as sql FROM log GROUP BY sql_code) ORDER BY "总体耗时比例(%)" desc limit 10
```



如上图所示，查询结果中包括SQL模板的ID、该模板SQL占总体SQL的耗时比例、执行次数、平均执行时间、平均影响行数以及样例SQL等信息。您可以基于该分析结果，找到应用中代价最高的SQL模板，并对其进行优化。

- 日志报表统计

  为了便于分析问题，PolarDB-X SQL审计与分析功能结合日志服务，提供了开箱即用的日志报表功能，帮助您实时诊断分析数据库的执行状况、性能和潜在安全问题等，详情请参见日志报表。

7.9.2.2. 开启SQL审计与分析

PolarDB-X SQL日志审计功能默认关闭，您可以在控制台上手动开启。

前提条件

- 已注册阿里云账号，并完成实名认证，详情请参见。
- 已开通日志服务。

  首次登录日志服务控制台时，根据页面提示开通日志服务。
- 目标PolarDB-X实例下已创建数据库。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏，单击**诊断与优化 > SQL审计与分析**。
6. 在页面右上角，打开**当前数据库SQL审计日志状态**开关即可。

执行结果

SQL审计功能开启后，相同地域下的PolarDB-X数据库的审计日志会写入同一个日志服务的Logstore中。同时，日志服务为您创建以下默认配置。

> **警告** 日志服务会不定期更新与升级SQL日志审计功能，专属日志库的索引与默认报表也会自动更新，因此请勿随意删除或修改日志服务为您默认创建的Project、Logstore、索引和仪表盘等设置。

默认配置项	配置内容
Project	默认为您创建Project为 `polardbx-sqlaudit-`**地域**`ID-`**阿里云账户**`ID` ，例如 `polardbx-sqlaudit-cn-hangzhou-123456789` 。
Logstore	默认为您创建Logstore为 `polardbx-sqlaudit-log` 。
地域	相同地域下的PolarDB-X数据库的审计日志会写入同一个日志服务的Logstore中。例如，华东1（杭州）地域中所有可用区的PolarDB-X数据库审计日志都会写入日志服务的Logstore `polardbx-sqlaudit-cn-hangzhou-`**阿里云账户**`ID` 中。
Shard	默认创建5个分区（Shard），并开启管理Shard功能。
索引	日志服务自动为您创建索引。 **警告** 您的索引设置会随着功能升级自动更新，请勿手动修改或删除索引。
日志存储时间	默认保存45天，45天以上的日志自动删除。
仪表盘	PolarDB-X默认创建如下三个仪表盘： • 运营中心 • 性能中心 • 安全中心 关于仪表盘的更多信息，请参见日志报表。 **说明** 请勿修改默认的仪表盘配置，默认仪表盘会随着功能升级自动更新。您也可以自定义创建新的仪表盘，详情请参见创建仪表盘。

7.9.2.3. 日志字段

本文将介绍PolarDB-X审计日志所包含的日志字段详情。

字段名称	字段值类型	字段说明
__topic__	string	日志主题，格式为 `polardbx_sqlaudit` 。
instance_id	string	PolarDB-X实例ID。

字段名称	字段值类型	字段说明
db_name	string	数据库名称。
client_ip	string	访问PolarDB-X实例的客户端IP地址。
client_port	int	访问PolarDB-X实例的客户端端口。
user	string	执行SQL的用户名。
sql	string	被执行的SQL语句。
response_time	int	响应时间，单位：毫秒（ms）。
affect_rows	int	• 若执行的是DML，表示受影响的行数。 • 若执行的是查询语句，表示返回结果的行数。
trace_id	string	SQL执行的TRACE ID。
fail	int	SQL执行是否出错，取值范围如下： • 0：未出错 • 1：出错
hint	string	SQL执行使用的HINT。
table_name	string	SQL执行涉及到的表。若涉及多个表，则多个表名称之间以英文逗号（,）分隔。
sql_type	string	SQL类型，取值范围如下： • SELECT • INSERT • DELETE • UPDATE • SET • ALTER • CREATE • DROP • TRUNCATE • REPLACE • OTHER
sql_type_detail	string	SQL解析器的名称。
sql_time	datetime	SQL开始执行的时间，格式为`yyyy-MM-dd HH:mm:ss.SSS`（UTC时间）。
sql_code	string	模板SQL的哈希值。
transaction_id	string	事务ID。
autocommit	int	SQL执行时`autocommit`是否打开，取值范围如下： • 0：未打开 • 1或NULL：打开

字段名称	字段值类型	字段说明
transaction_policy	string	SQL执行时使用的事务策略，取值范围如下 • 空：未开启事务 • TSO：执行PolarDB-X TSO事务（默认取值） • BEST_EFFORT：执行PolarDB-X 2PC事务
workload_type	string	SQL执行时的负载类型，取值范围如下： • TP：事务类型的负载 • AP：分析类型的负载

7.9.2.4. 日志分析

PolarDB-X支持SQL审计与分析功能，依托日志服务产品，提供强大的日志分析能力。本文将介绍常见场景的SQL日志分析语句及示例。

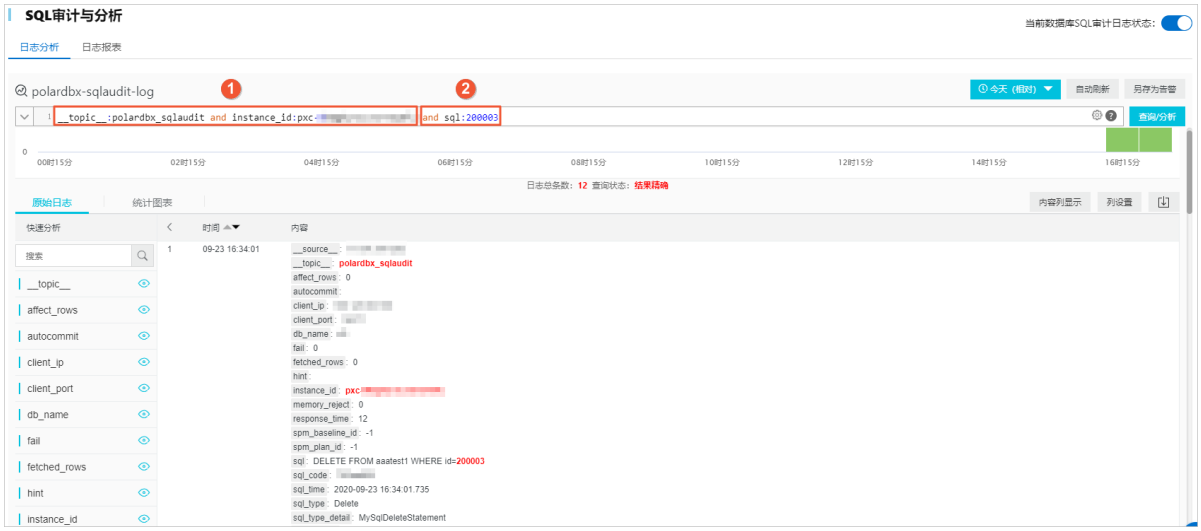
前提条件

已开启SQL审计与分析功能。

注意事项

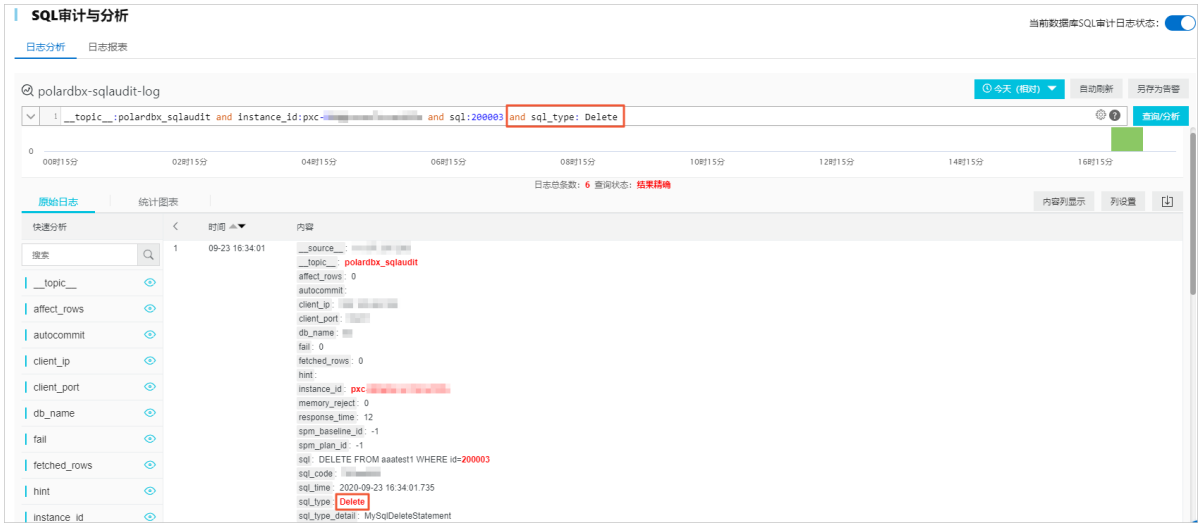
- 相同地域下的所有PolarDB-X数据库的审计日志都会写入同一个日志服务的Logstore中，因此PolarDB-X的SQL审计与分析搜索框内会默认为您带上 `__topic__` 的过滤条件，保证您搜索到的SQL日志都属于同一地域下的PolarDB-X数据库。本文提供的所有查询语句，都需要在已有的 `__topic__` 过滤条件后追加使用。

  例如图中1部分的语句为默认过滤条件，序号2部分的语句为追加的过滤条件。



- 您可以单击**原始日志**各个字段后的详细内容，自动生成包含对应字段查询语句。

  例如您可以单击 `sql_type` 后的 `Delete` ，查看所有包含 `Delete` 字段的SQL语句。



快速定位SQL

您可以使用以下命令快速定位问题SQL。

- 模糊搜索

  例如，您可以使用如下命令查询包含关键字为 `200003` 的SQL语句：

  ```
  and sql: 200003
  ```

- 字段搜索

  依赖预置的索引字段，PolarDB-X SQL审计还支持根据字段搜索。例如您可以使用如下命令查询DROP类型的SQL：

  ```
  and sql_type:Drop
  ```

- 多条件搜索

  您可以通过 `and` 或 `or` 等关键字实现多条件的搜索。例如您可以使用如下命令查询针对 `id=200003` 行进行的所有DELETE语句：

  ```
  and sql: 200003 and sql_type: Delete
  ```

- 数值比较搜索

  索引字段中的 `affect_rows` 和 `response_time` 是数值类型，支持比较操作符。例如您可以使用如下命令查询 `response_time` 大于5秒的DROP语句：

  ```
  and response_time > 5 and sql_type: Drop
  ```

  或者使用如下命令查询删除100行以上数据的SQL语句：

  ```
  and affect_rows  > 100 and sql_type: Delete
  ```

SQL执行状况分析

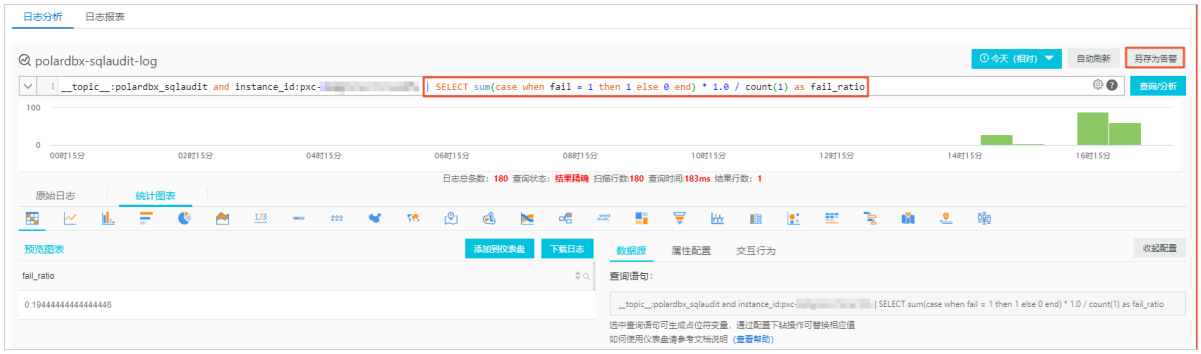
您可以使用以下命令查看SQL执行状况。

- SQL执行失败率

  您可以使用如下命令查询SQL执行的失败率：

  ```
  | SELECT sum(case when fail = 1 then 1 else 0 end) * 1.0 / count(1) as fail_ratio
  ```

  查询结果如下图所示：



> **说明** 若您的业务对SQL错误率敏感，您可以在此查询结果的基础上，单击页面右上角的**另存为告警**并根据业务需要自定义告警信息。

- SQL累计查询行数

  您可以使用如下命令查询SELECT语句累计查询的行数：

  ```
  and sql_type: Select | SELECT sum(affect_rows)
  ```

- SQL类型分布

  您可以使用如下命令查询SQL类型的分布情况：

  ```
  | SELECT  sql_type, count(sql) as times GROUP BY sql_type
  ```

- SQL独立用户IP分布

  您可以使用如下命令查询SQL独立用户的IP地址分布：

  ```
  | SELECT  user, client_ip, count(sql) as times GROUP BY user, client_ip
  ```

SQL性能分析

您可以使用以下命令查看SQL性能分析详情。

- SELECT平均耗时

  您可以使用如下命令查询SELECT语句的平均耗时：

  ```
  and sql_type: Select | SELECT avg(response_time)
  ```

- SQL执行耗时分布

  您可以使用如下命令查询SQL执行耗时分布：

  ```
  and response_time > 0 | select   case  when response_time <= 10 then '<=10毫秒'  when response_time > 10 a
  nd response_time <= 100 then '10~100毫秒'  when response_time > 100 and response_time <= 1000 then '100毫秒
  ~1秒'  when response_time > 1000 and response_time <= 10000  then '1秒~10秒'  when response_time > 10000 an
  d response_time <= 60000  then '10秒~1分钟'  else '>1分钟' end as latency_type,  count(1) as cnt group by l
  atency_type order by latency_type DESC
  ```

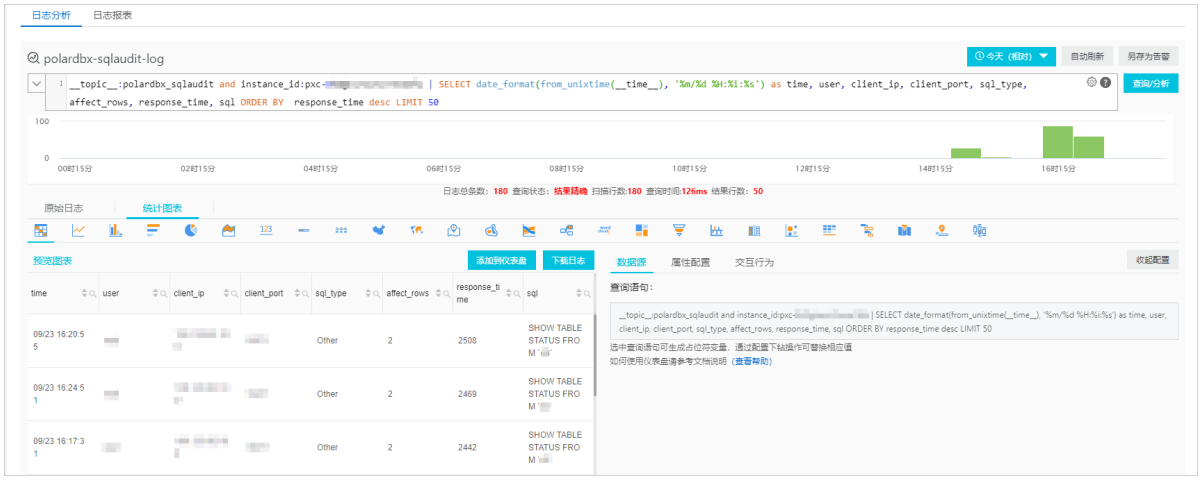
  > **说明** 上述查询将 `response_time` 按照 `<=10毫秒` 、`10~100毫秒` 、`100毫秒~1秒` 和 `1秒~10秒` 这4个时间段对SQL执行耗时进行分布，您也可以自定义 `response_time` 的时间段，获取更加精细的结果。

- 慢SQL Top 50

  您可以使用如下命令查询系统排名前50的慢SQL：

  ```
  | SELECT date_format(from_unixtime(__time__), '%m/%d %H:%i:%s') as time, user, client_ip, client_port, sql
  _type, affect_rows, response_time, sql ORDER BY  response_time desc LIMIT 50
  ```

  查询结果如下图所示，结果中包含SQL执行时间、执行的用户名、IP地址、端口号、SQL类型、影响行数、执行时间以及具体的SQL语句等信息。

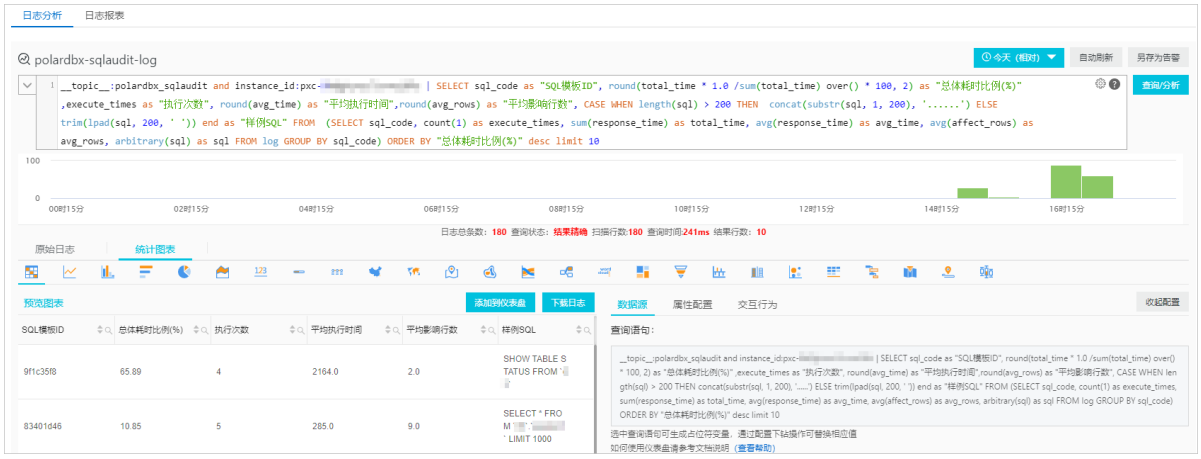


- 高代价SQL模板Top 10

  在大多数应用中，SQL通常基于若干模板动态生成的，只是参数不同。您可以使用如下命令通过模板ID找到应用中高代价的SQL模板：

```
| SELECT sql_code as "SQL模板ID", round(total_time * 1.0 /sum(total_time) over() * 100, 2) as "总体耗时比例(%)" ,execute_times as "执行次数", round(avg_time) as "平均执行时间",round(avg_rows) as "平均影响行数", CASE WHEN length(sql) > 200 THEN  concat(substr(sql, 1, 200), '.....') ELSE trim(lpad(sql, 200, ' ')) end as "样例SQL" FROM  (SELECT sql_code, count(1) as execute_times, sum(response_time) as total_time, avg(response_time) as avg_time, avg(affect_rows) as avg_rows, arbitrary(sql) as sql FROM log GROUP BY sql_code) ORDER BY "总体耗时比例(%)" desc limit 10
```

  统计结果中包括SQL模板ID，该模板SQL占总体SQL的耗时比例、执行次数、平均执行时间、平均影响行数以及样例SQL等信息。



> **说明** 上述查询是按照**总体耗时比例%**排序，您也可以根据**平均执行时间**或**执行次数**进行排序帮助排查问题。

- 事务平均执行时长

  对于相同事务内的SQL，预置的 `trace_id` 字段前缀相同，后缀为 `'-' + 序号` ；非事务的SQL的 `trace_id` 中则不包含 `'-'` 。因此，您可以使用如下命令对事务SQL的性能进行相关分析。

> **说明** 由于事务分析涉及前缀匹配操作，查询效率会低于其它类型的查询操作。

  - 查询事务的平均执行耗时

    您可以使用如下语句查询事务的平均执行耗时：

```
 | SELECT  sum(response_time) / COUNT(DISTINCT substr(trace_id, 1, strpos(trace_id, '-') - 1)) where strpos(trace_id, '-') > 0
```

- 慢事务Top 10

  您可以按照事务的执行时间排序查询慢事务的列表：

  ```
  | SELECT substr(trace_id, 1, strpos(trace_id, '-') - 1) as "事务ID" , sum(response_time) as "事务耗时" wh
  ere strpos(trace_id, '-') > 0 GROUP BY substr(trace_id, 1, strpos(trace_id, '-') - 1) ORDER BY "事务耗时"
  DESC LIMIT 10
  ```

  在此基础上，您可以使用如下命令，根据查到的慢事务ID搜索该事务下的所有SQL用于分析执行慢的具体原因：

  ```
   and trace_id: db3226a20402000*
  ```

- 大批量操作事务Top 10

  您可以使用如下命令按照事务内SQL影响的行数排序，查询大批量操作的事务列表：

  ```
  | SELECT substr(trace_id, 1, strpos(trace_id, '-') - 1) as  "事务ID" , sum(affect_rows) as "影响行数" whe
  re strpos(trace_id, '-') > 0 GROUP BY substr(trace_id, 1, strpos(trace_id, '-') - 1) ORDER BY "影响行数"
  DESC LIMIT 10
  ```

SQL安全性分析

您可以使用以下命令查看SQL安全性分析情况。

- 错误SQL类型分布

  您可以使用如下命令查看错误SQL类型分布：

  ```
  and fail > 0 | select sql_type, count(1) as "错误次数" group by sql_type
  ```

- 高危SQL列表

  PolarDB-X中的高危SQL是指DROP或TRUNCATE类型的SQL（您也可以根据业务需求增加更多条件自定义高危SQL)。

  您可以使用如下命令查询包含DROP或TRUNCATE类型的SQL列表：

  ```
  and sql_type: Drop OR sql_type: Truncate
  ```

- 大批量删除SQL列表

  您可以使用如下命令大批量删除SQL列表：

  ```
  and affect_rows > 100 and sql_type: Delete | SELECT date_format(from_unixtime(__time__), '%m/%d %H:%i:%s')
  as time, user, client_ip, client_port, affect_rows, sql ORDER BY  affect_rows desc LIMIT 50
  ```

7.9.2.5. 日志报表

PolarDB-X 支持SQL审计与分析功能，依托日志服务，提供了运营中心、性能中心、安全中心等开箱即用的报表，方便您快速查看了解PolarDB-X数据库的执行状况、性能指标、潜在安全问题等情况。

前提条件

已开启SQL审计与分析功能。

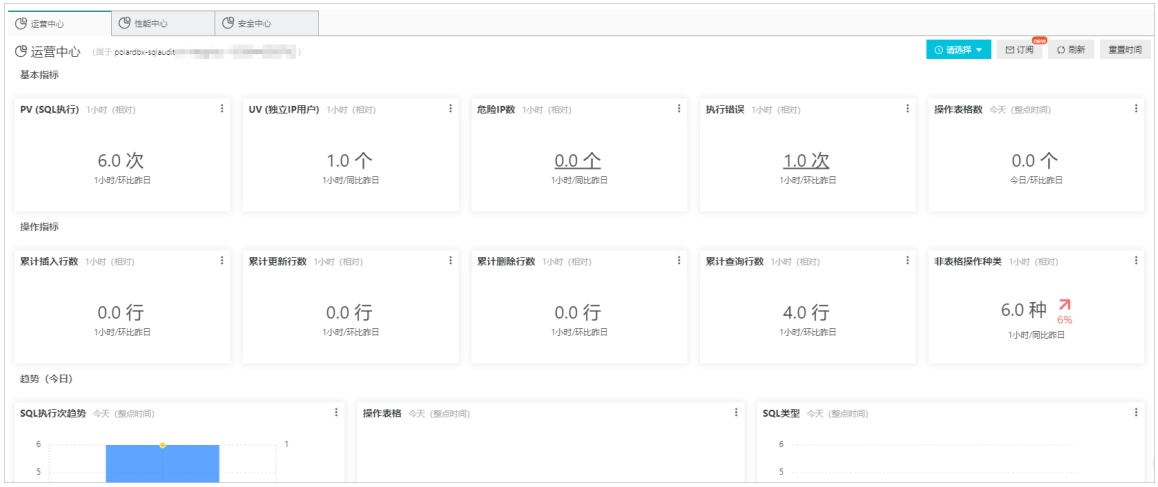
注意事项

由于相同地区的PolarDB-X数据库的审计日志均写入日志服务同一个Logstore中，查看当前PolarDB-X实例下的报表数据时，默认为您添加基于`__topic__:polardbx_sqlaudit and instance_id:xxxxxxxxx`的过滤条件，表示查看当前实例下的所有数据库的日志数据。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。

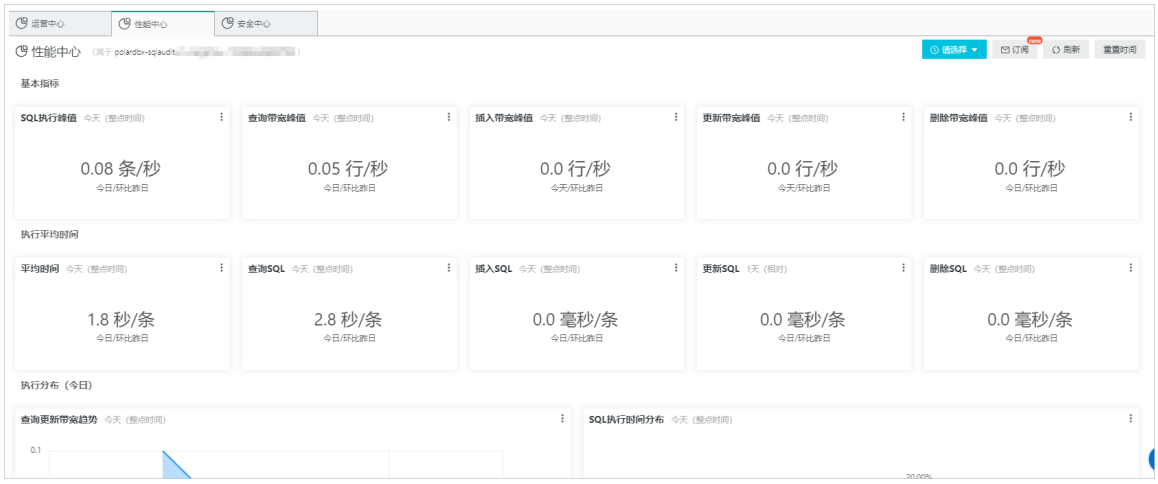
5. 在左侧导航栏，单击**诊断与优化 > SQL审计与分析**。
6. 在**SQL审计与分析**页面，单击**日志报表**页签，您可以通过单击不同页签查看**运营中心**、**性能中心**和**安全中心**的详情。
   - **运营中心**：展示了目标PolarDB-X实例下所有数据库的SQL执行指标、分布、趋势等信息。



分类	图表	类型	默认时间范围	描述
基本指标	PV（SQL执行）	单值	1小时（相对）	SQL执行的次数。
	UV（独立IP用户）	单值	1小时（相对）	独立的用户及IP数量。
	危险IP数	单值	1小时（相对）	危险IP的数量。 **说明** 更多关于危险IP的详情，请参见安全检测函数。
	执行错误	单值	1小时（相对）	执行错误的SQL数量。
	操作表格数	单值	1小时（相对）	SQL操作的表格总数。
操作指标	累计插入行数	单值	1小时（相对）	插入操作累计插入的数据行数。
	累计更新行数	单值	1小时（相对）	更新操作累计更新的数据行数。
	累计删除行数	单值	1小时（相对）	删除操作累计删除的数据行数。
	累计查询行数	单值	1小时（相对）	查询操作累计返回的数据行数。
	非表格操作种类	单值	1小时（相对）	非表格操作的SQL种类，例如 `show variables like`
趋势	SQL执行趋势	柱状图	1小时（相对）	SQL执行的趋势分布以及对应的错误SQL的分布趋势。
	操作表格	流图	1小时（相对）	SQL操作表格的分布情况。
	SQL类型	流图	1小时（相对）	SQL类型的按照时间的分布情况。
分布	操作用户分布	饼图	1小时（相对）	执行SQL用户的分布情况。
	SQL执行类型分布	饼图	1小时（相对）	当前时间范围内SQL类型的比例。
	操作最多的表格 Top 50	表格	1小时（相对）	操作最多的表格列表，包括表格的名称以及对应的读、删、改、插的次数。

分类	图表	类型	默认时间范围	描述
	执行分布（世界）	地图	1小时（相对）	执行SQL的客户端IP在世界地图上的分布情况。
	执行分布（中国）	地图	1小时（相对）	执行SQL的客户端IP在中国地图上的分布情况。

- **性能中心**：展示了目标PolarDB-X实例下所有数据库的具体性能指标，例如SQL执行峰值、SQL执行的平均时间、慢SQL（即执行时间超过1s的SQL）的具体分布与来源等。



分类	图表	类型	默认时间范围	描述
基本指标	SQL 执行峰值	单值	1小时（相对）	每秒SQL执行条数的峰值。
	查询带宽峰值	单值	1小时（相对）	每秒查询SQL返回行数的峰值。
	插入带宽峰值	单值	1小时（相对）	每秒插入SQL插入的行数峰值。
	更新带宽峰值	单值	1小时（相对）	每秒更新SQL更新的行数峰值。
	删除带宽峰值	单值	1小时（相对）	每秒删除SQL删除的行数峰值。
执行平均时间	平均时间	单值	1小时（相对）	SQL平均的执行时间。
	查询SQL	单值	1小时（相对）	平均每秒查询SQL执行的条数。
	插入SQL	单值	1小时（相对）	平均每秒插入SQL执行的条数。
	更新 SQL	单值	1小时（相对）	平均每秒更新SQL执行的条数。
	删除 SQL	单值	1小时（相对）	平均每秒删除SQL执行的条数。
执行分布	查询更新带宽趋势	折线图	1小时（相对）	查询SQL、更新SQL操作行数随时间的分布情况。
	SQL执行时间分布	饼图	1小时（相对）	SQL执行时间的分布情况。
	慢SQL表格分布	饼图	1小时（相对）	慢SQL的表格分布情况。
	慢SQL用户分布	饼图	1小时（相对）	慢SQL的用户分布情况。
	慢SQL类型分布	饼图	1小时（相对）	慢SQL的类型分布情况

分类	图表	类型	默认时间范围	描述
慢SQL分布	慢SQL列表Top 50	表格	1小时（相对）	慢SQL的列表，包括： ■ SQL开始执行的时间点 ■ 客户端（IP、城市、网络） ■ SQL执行时间 ■ PolarDB-X实例ID ■ 数据库 ■ 表格 ■ 用户 ■ 影响行数 ■ SQL类型 ■ 具体SQL语句
高代价 SQL模板	SQL模板执行时间Top 20	表格	1小时（相对）	按照高代价SQL模板统计该模板 SQL的执行情况，包括： ■ SQL模板ID ■ 总体耗时比例 ■ 执行次数 ■ 平均执行时间（毫秒） ■ 平均影响行数 ■ 样例SQL
事务SQL	事务执行影响行数Top 20	表格	1小时（相对）	事务影响行数的Top 20列表，包括： ■ 事务ID ■ 影响行数
	事务执行时间Top 20	表格	1小时（相对）	事务执行时间的Top 20列表，包括： ■ 事务ID ■ 执行时间（毫秒）

- **安全中心**：展示了目标PolarDB-X实例下所有数据库的失败SQL和危险SQL（DROP或RUNCATE类型的SQL），以及大批量（影响行数超过100行）删除或修改事件的详情、分布和趋势等。

分类	图表	类型	默认时间范围	描述
	错误数	单值	1小时（相对）	失败SQL的执行次数。

分类	图表	类型	默认时间范围	描述
安全指标	大批量删除事件	单值	1小时（相对）	大批量删除事件的 SQL数量。
	大批量修改事件	单值	1小时（相对）	大批量修改事件的SQL数量。
	危险SQL执行	单值	1小时（相对）	危险SQL的数量。
	危险IP数	单值	1小时（相对）	危险IP的数量。 说明 更多关于危险IP的详情，请参见安全检测函数。
错误分布	错误操作类型分布	面积图	1小时（相对）	失败SQL的类型分布。
	出错客户端外网分布	地图	1小时（相对）	失败SQL的客户端在中国地图上的分布。
	错误最多的客户端	表格	1小时（相对）	失败SQL的客户端列表，包括： ■ 客户端（IP、城市、网络） ■ 错误次数 ■ 主要错误（查询、插入有、更新、删除、其它） ■ 出错样例
危险SQL情况	危险SQL 执行列表	表格	1小时（相对）	危险SQL的列表，包括： ■ SQL开始执行的时间点 ■ 客户端（IP、城市、网络） ■ SQL ■ PolarDB-X实例ID ■ 数据库 ■ 表格 ■ 用户
大批量事务	大批量删除事件 Top 50	表格	1小时（相对）	大批量删除SQL的列表，包括： ■ 最早执行时间 ■ 最近执行时间 ■ PolarDB-X实例ID ■ 数据库 ■ 表格 ■ 执行次数 ■ 平均删除行数 ■ 平均时长（秒） ■ SQL

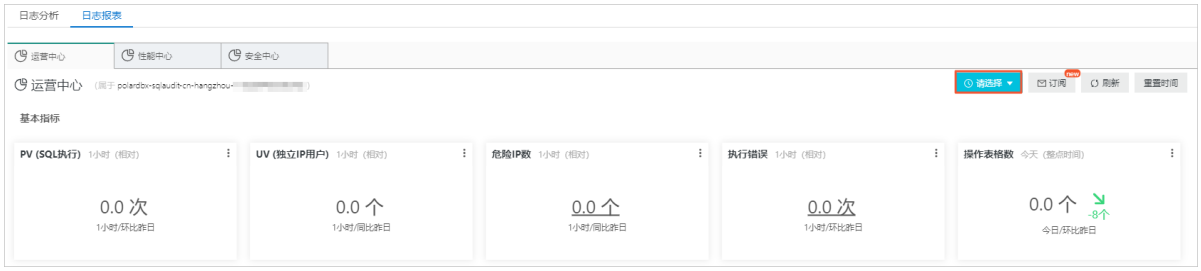
分类	图表	类型	默认时间范围	描述
	大批量修改事件 Top 50	表格	1小时（相对）	大批量修改 SQL 的列表，包括： ■ 最早执行时间 ■ 最近执行时间 ■ PolarDB-X实例ID ■ 数据库 ■ 表格 ■ 执行次数 ■ 平均更新行数 ■ 平均时长（秒） ■ SQL

修改数据统计时间

日志报表页面的所有图表都是基于不同时间段（默认为过去1小时内的）的数据统计结果，您可以根据业务需求修改目标页签下的所有图表或单一图表的数据统计时间范围。

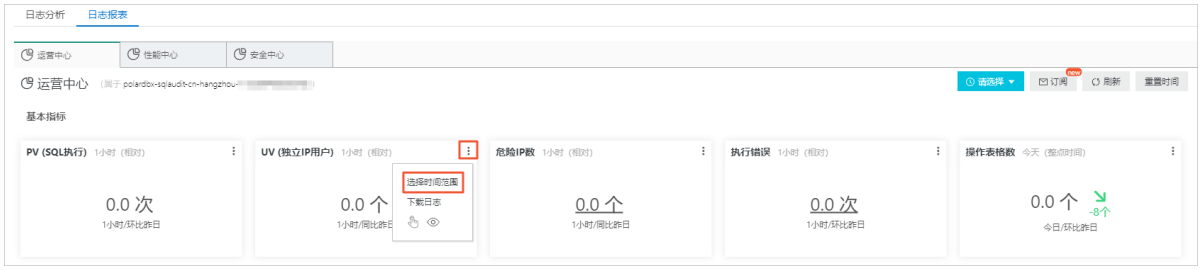
- 修改目标页签下所有图表的数据统计时间

  在目标页签右上角，单击**请选择**，在弹出的页面中修改当前页面所有图表的数据统计时间。



- 修改目标页签下单一图表的数据统计时间

  将鼠标放置在目标图表右侧的⋮图标后，单击**选择时间范围**，在弹出的页面中修改当前图表的数据统计时间。



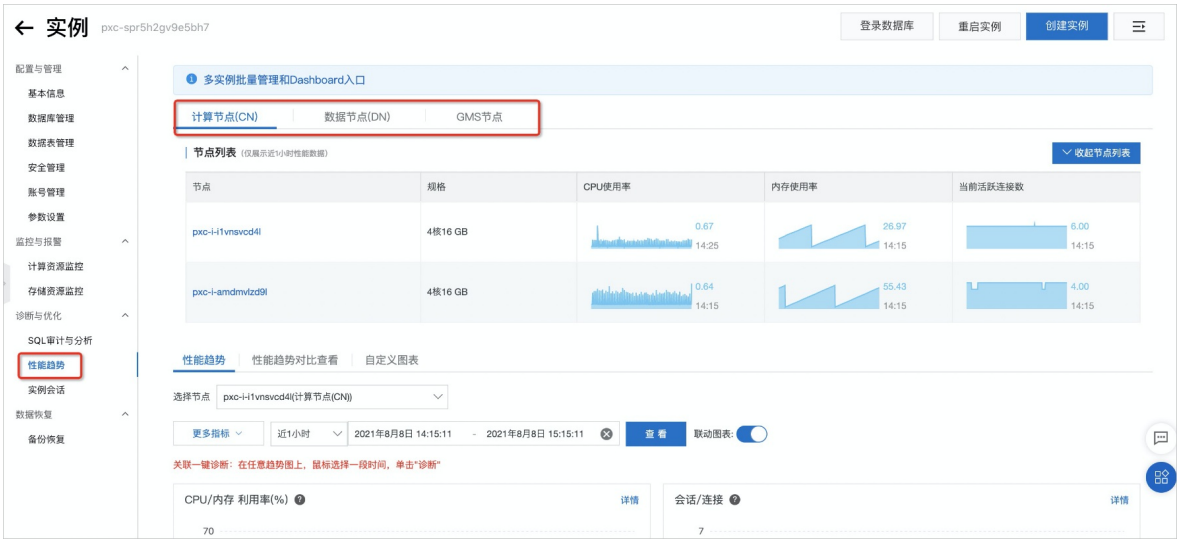
7.9.3. 性能趋势

本文介绍了PolarDB-X性能趋势的查看方式。

PolarDB-X集成DAS的性能趋势功能，提供计算节点（CN）、存储节点（DN）、GMS节点的监控指标查看能力。同时支持多种查看性能趋势方式，不仅支持性能趋势区间查看，也支持性能趋势对比查看和自定义性能趋势查看。

性能趋势查看

1. 登录云原生分布式数据库控制台。
2. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
3. 在页面左上角选择目标实例所在地域。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 性能趋势**。



6. 单击页面上的页签分别查看计算节点（CN）、存储节点（DN）和GMS节点的监控指标。具体指标请参见性能指标说明。
7. 节点列表展示了节点的规格和CPU使用率、内存使用率等。
8. 单击**性能趋势**页签，选择节点和时间范围，单击**查看**，可查看相关性能趋势图表。
9. 单击**性能趋势对比查看**页签。设置任意两个时间段，单击**查看**可查看时间段内的性能对比趋势图表。
10. 单击**自定义性能趋势**页签，您可以根据业务需要，自定义多个性能监控大盘，将需要的多个性能监控指标在同一个图标中进行展示，便于问题排查和分析。

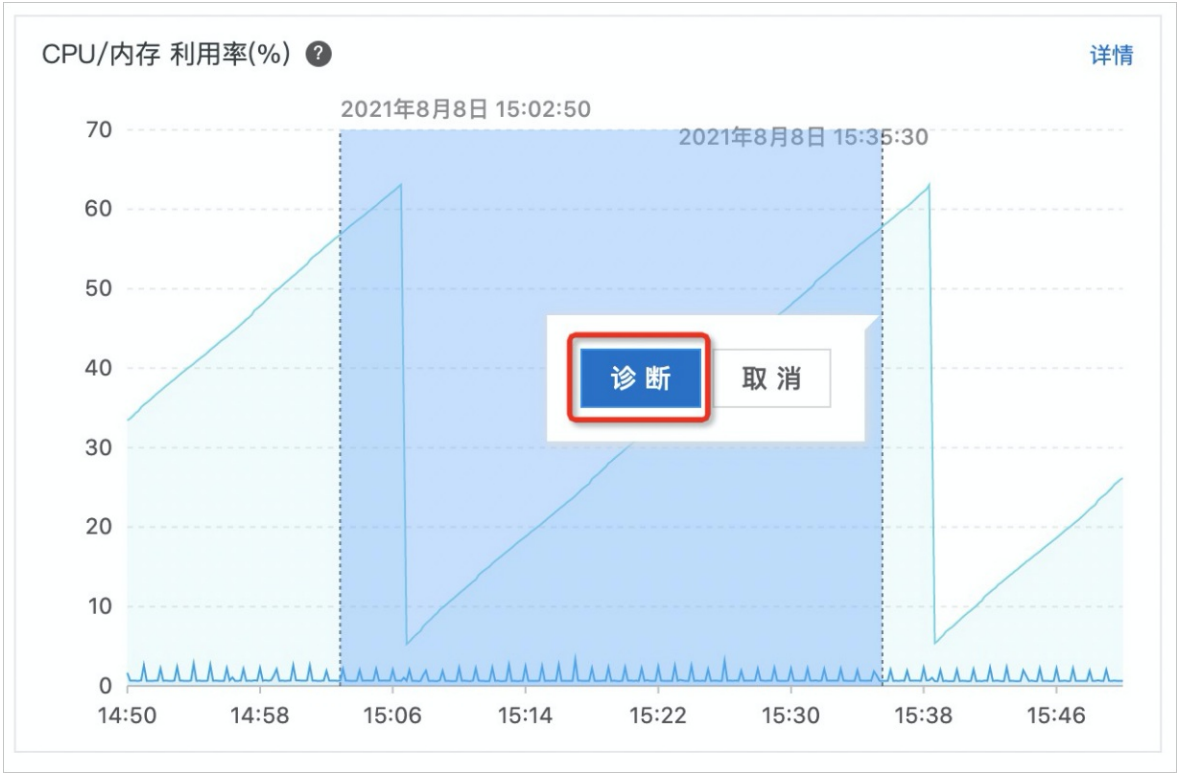
> **说明** 如果您是首次使用，需要先创建监控大盘，详情可参见 监控大盘。

执行性能诊断

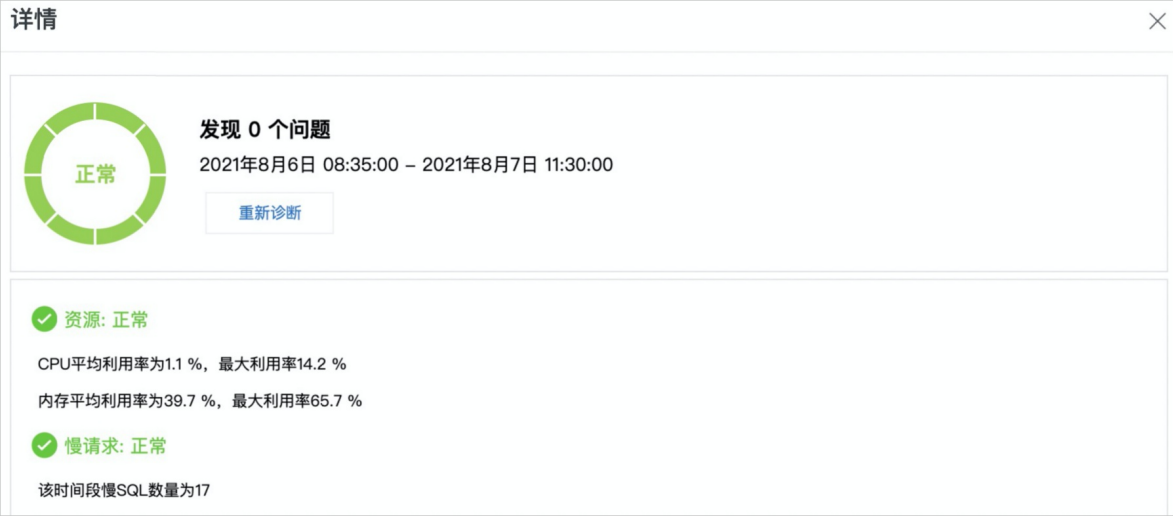
除了性能趋势查看外，在任意节点的趋势图上，按住鼠标拖动选择一段时间，即可针对该节点在这段时间范围内的资源利用率，慢SQL状况进行诊断。

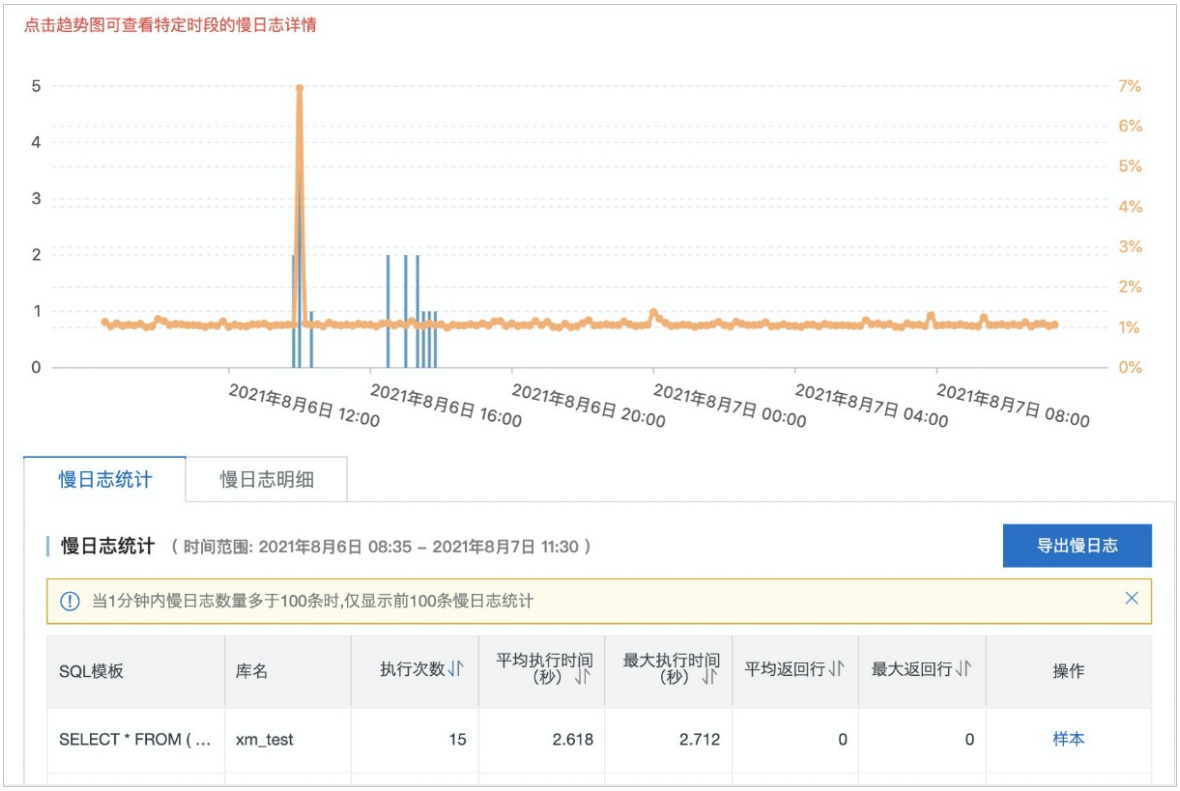
> **说明** 目前仅计算节点和存储节点支持性能诊断，GMS节点不支持。

1. 登录云原生分布式数据库控制台。
2. 在**实例列表**页，单击PolarDB-X 2.0页签。
3. 在页面左上角选择目标实例所在地域。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 性能趋势**。
6. 在任意节点的趋势图上，按住鼠标拖动选择一段时间，点击诊断按钮。



7. 在诊断详情页待诊断任务完成即可查看资源利用率和慢SQL的异常情况。





性能指标说明

计算节点

指标	单位	含义
polardbx.cpu_usage	%	CPU使用率的平均值。
polardbx.mem_usage	%	JVM的内存使用率，内存使用率波动属于正常现象。
polardbx.active_connection	Count	连接总数。
polardbx.running_thread	Count	活跃连接总数。
polardbx.network_in_bytes	Byte	网络输入流量的总和。
polardbx.network_out_bytes	Byte	网络输出流量的总和。
polardbx.logic_qps	Per Second	每秒处理的逻辑SQL语句数目的总和。
polardbx.physical_qps	Per Second	每秒处理的物理SQL语句数目的总和。
polardbx.logic_rt	Ms	逻辑SQL平均响应时间。
polardbx.physical_rt	Ms	物理SQL平均响应时间。
polardbx.slow_request_count	Per Second	逻辑慢SQL数量。
polardbx.physical_slow_reuquest_count	Per Second	物理慢SQL数量。

存储节点与GMS节点

指标	单位	含义
mysql.tps	Per Second	每秒事务数。
mysql.qps	Per Second	每秒请求数。
mysql.total_session	Count	当前全部会话。
mysql.active_session	Count	当前活跃会话。
mysql.bytes_received	KByte	平均每秒从所有客户端接收到的字节数。
mysql.bytes_sent	KByte	平均每秒发送给所有客户端的字节数。
mysql.tb.tmp.disk	Count	MySQL执行语句时在硬盘上自动创建的临时表的数量。
mysql.insert_ps	Per Second	平均每秒insert语句执行次数。
mysql.select_ps	Per Second	平均每秒select语句执行次数。
mysql.update_ps	Per Second	平均每秒update语句执行次数。
mysql.delete_ps	Per Second	平均每秒delete语句执行次数。
mysql.replace_ps	Per Second	平均每秒replace语句执行次数。
mysql.innodb_data_written	KByte	InnoDB平均每秒写字节数。
mysql.innodb_data_read	KByte	InnoDB平均每秒读字节数。
mysql.innodb_buffer_pool_reads_requests	Count	InnoDB平均每秒从Buffer Pool读取页的次数（逻辑读）。
mysql.innodb_bp_dirty_pct	%	InnoDB Buffer Pool脏页比率，计算公式：Innodb_buffer_pool_pages_dirty / Innodb_buffer_pool_pages_data * 100%。
mysql.innodb_bp_hit	%	InnoDB Buffer Pool读缓存命中率，计算公式：(Innodb_buffer_pool_read_requests - Innodb_buffer_pool_reads) /Innodb_buffer_pool_read_requests * 100%。
mysql.innodb_bp_usage_pct	%	InnoDB Buffer Pool使用率，计算公式：innodb_buffer_pool_pages_data / (innodb_buffer_pool_pages_data +innodb_buffer_pool_pages_free) * 100%。
mysql.innodb_log_writes	Per Second	Innodb平均每秒物理写Redo Log File次数。
mysql.innodb_os_log_fsyncs	Per Second	平均每秒向日志文件完成的fsync()写数量。
mysql.innodb_rows_deleted	Per Second	InnoDB平均每秒删除的行数。
mysql.innodb_rows_read	Per Second	InnoDB平均每秒读取的行数。
mysql.innodb_rows_inserted	Per Second	InnoDB平均每秒插入的行数。
mysql.innodb_rows_updated	Per Second	InnoDB平均每秒更新的行数。

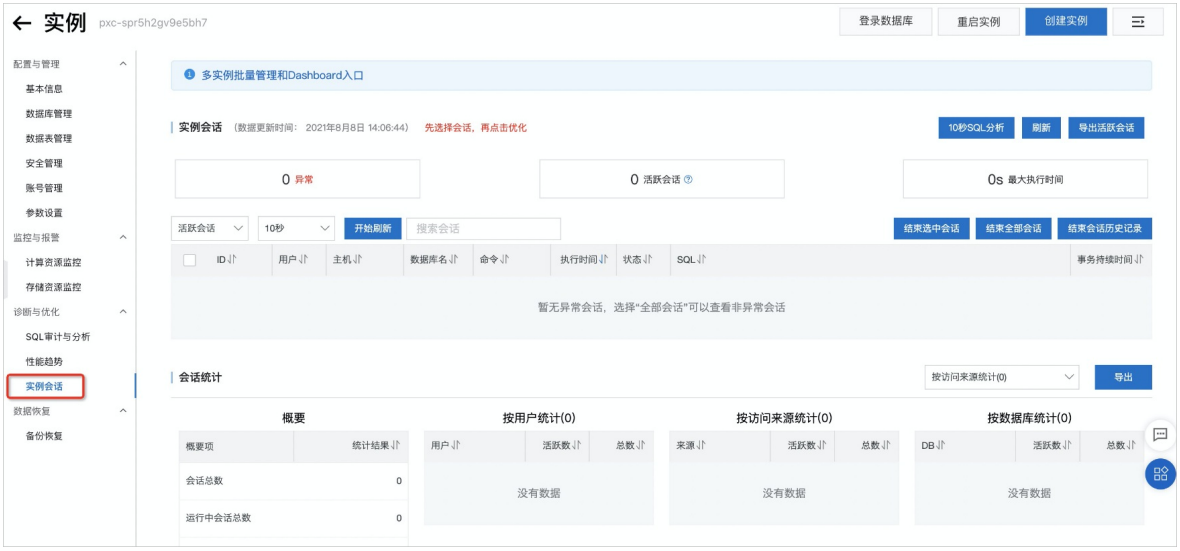
指标	单位	含义
mysql.mem_usage	%	MySQL实例内存使用率（占操作系统总数）。
mysql.cpu_usage	%	MySQL服务进程CPU使用率（阿里云数据库最高100%）。
mysql.data.size	MByte	数据空间。
mysql.tmp.size	MByte	临时表空间。
mysql.other.size	MByte	系统空间。
mysql.instance.size	MByte	MySQL实例总空间使用量。
mysql.log.size	MByte	日志空间。
mysql.iops	Count	MySQL读写次数。

7.9.4. 实例会话

PolarDB-X支持查看实例的会话及其统计信息。当您的实例突然出现CPU飙升、活跃会话升高或响应时间飙升的告警时，PolarDB-X提供10秒SQL分析的功能帮助您对10秒内的实时SQL进行分析。

会话管理

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 实例会话**。



6. 在**实例会话**页签，您可查看正在执行的会话和全部连接，也可**终结选中会话**。

7. 在**会话统计**页签，您可查看**客户端统计**、**按namespace统计**等的统计信息。

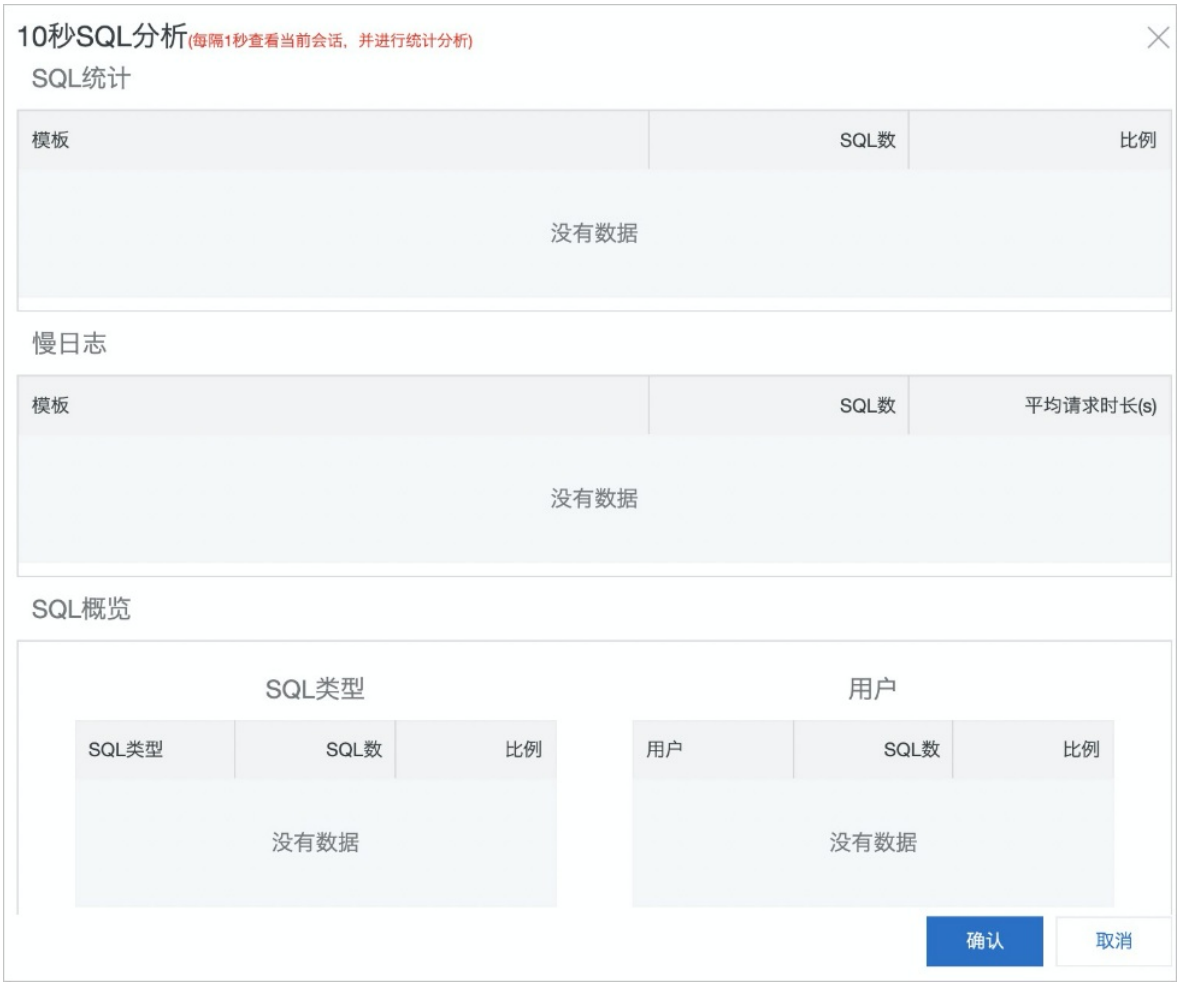
> **说明** 统计信息可以帮助您发现总会话数、 活跃会话总数、 会话执行最长耗时（秒）、 超过3秒的会话数、 没有走索引的查询等会话信息。

10秒SQL分析

该功能帮助您对实例10秒内的SQL进行实时分析。在10秒中，每隔一秒执行一次 `show processlist` ，然后将所有的结果集进行统计分析，您可看到在这10秒中，哪些SQL执行的次数最多，是否存在慢SQL等。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 实例会话**。
6. 单击**10秒SQL分析**按钮。

7. 10秒后即可查看在这10秒中，哪些SQL执行的次数最多，是否存在慢SQL等。



7.9.5. 慢日志

本文介绍了PolarDB-X的慢日志查询功能。

PolarDB-X提供慢日志查询功能，并且将CPU利用率和慢日志数量的视角结合在一起，能帮助您快速发现影响数据库性能的关键SQL。慢日志功能提供交互页面，您可以查看格式化的慢SQL模板、慢SQL明细，以及每条慢SQL执行次数、耗时、返回行数等关键信息。同时，您可以对慢SQL进行诊断优化。在分析慢SQL执行计划的基础上，为您推荐出包括局部索引和全局二级索引在内的合适索引方案，并提供优化后的预期收益明细。

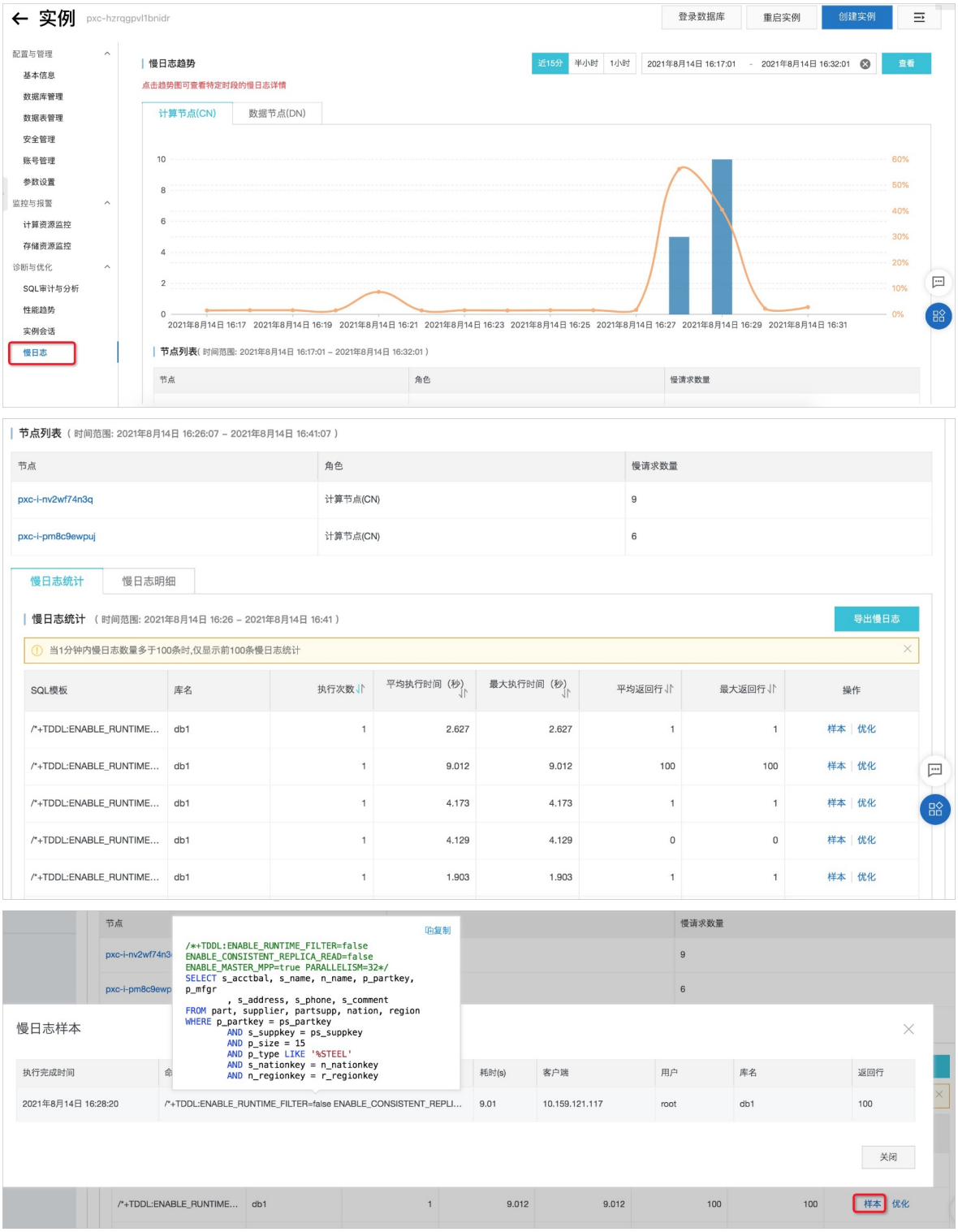
计算节点（CN）慢SQL和数据节点（DN）慢SQL的定义如下：

- 计算节点（CN）慢SQL：即逻辑慢SQL，客户端发送到PolarDB-X的慢SQL。
- 数据节点（DN）慢SQL：即物理慢SQL，PolarDB-X发送到存储层节点的慢SQL，不支持优化操作。

每个PolarDB-X节点最多保存30天慢日志，超过时间将会被自动删除。

慢日志查看

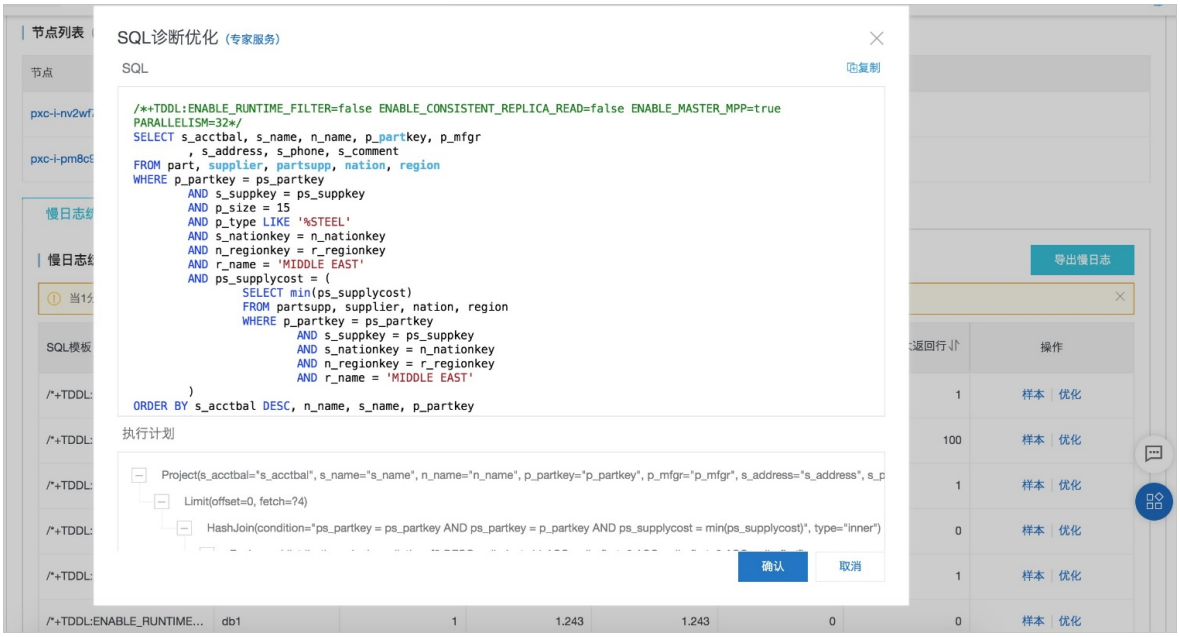
1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 慢日志**。
6. 单击页面上方**计算节点（CN）**或**数据节点（DN）**页签，可以分别查看计算节点慢SQL和数据节点慢SQL。

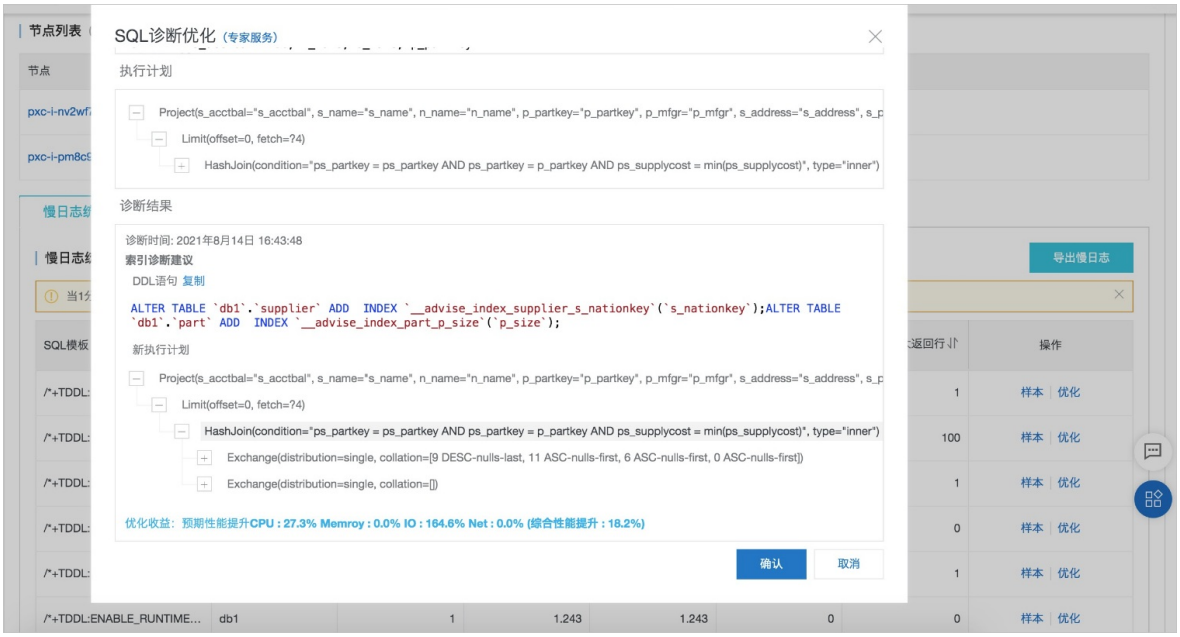


慢日志优化

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 慢日志**。
6. 在下方慢日志统计页签中查看慢SQL模版列表，单击**优化**按钮（目前仅支持CN节点的慢日志优化）。

7. 查看慢SQL优化建议。





7.9.6. 空间分析

PolarDB-X空间分析功能支持查看实例的空间使用概况、空间剩余可用天数，以及数据库中某个表的空间使用情况、空间碎片、空间异常诊断等。本文介绍如何使用PolarDB-X空间分析功能。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 空间分析**。
6. 在**空间分析**页面，您可直观的查看该实例的**空间概况**和**数据空间**页签中的**异常列表**、**空间变化趋势**等信息，详细说明见如下表。

参数	说明
空间概况	您可直观的查看空间近一周日均增长量，空间可用天数、已用空间等信息。
数据空间	您可查看每个数据空间和单个表空间的使用情况，单击表名可查看对应表的字段、索引、拆分方式、物理表空间等信息。
异常列表	◦ 实例的空间使用率检查：检查实例空间使用率是否大于90%。 ◦ 实例的剩余空间检查：检查实例空间剩余可用天数是否不足7天。
空间变化趋势	您可查看数据空间、已用空间、日志空间和临时空间数据趋势。
表空间	您可查看每张表的数据空间，索引空间，GSI 空间，碎片率等信息。单击表名可查看对应表的字段、索引、拆分方式、物理表空间等信息。

7.9.7. 实时性能

PolarDB-X提供实例级别的实时性能，帮助您在压测或者大促期间，实时确认数据库性能。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 实时性能**。
6. 进入**实时性能**详情页，查看相关性能指标。

详情页相关操作说明：

- 单击**多实例批量管理和Dashboard入口**，可跳转至DAS控制台。
- 单击**参数说明**，可查看相关指标的具体描述。
- 单击**计算节点(CN)**、**数据节点(DN)**、**GMS节点**或**集群**页签，可分别查看相关节点的性能信息。
- 单击**选择节点**下拉列表可查看不同节点的性能信息。

> **说明** PolarDB-X支持**实时图表**和**实时表格**两种视图。

7.9.8. 容量评估

您可以通过容量评估功能查看容量评估建议、性能容量情况、存储容量的使用情况和剩余可用时间。该功能还能通过机器学习和容量算法，帮助您预测未来存储空间的使用情况。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 容量评估**。
6. 在PolarDB-X实例的容量评估页面，您可以在**容量建议**和**容量评估**区域查看存储空间的性能及具体的使用情况。

详情页相关操作说明：

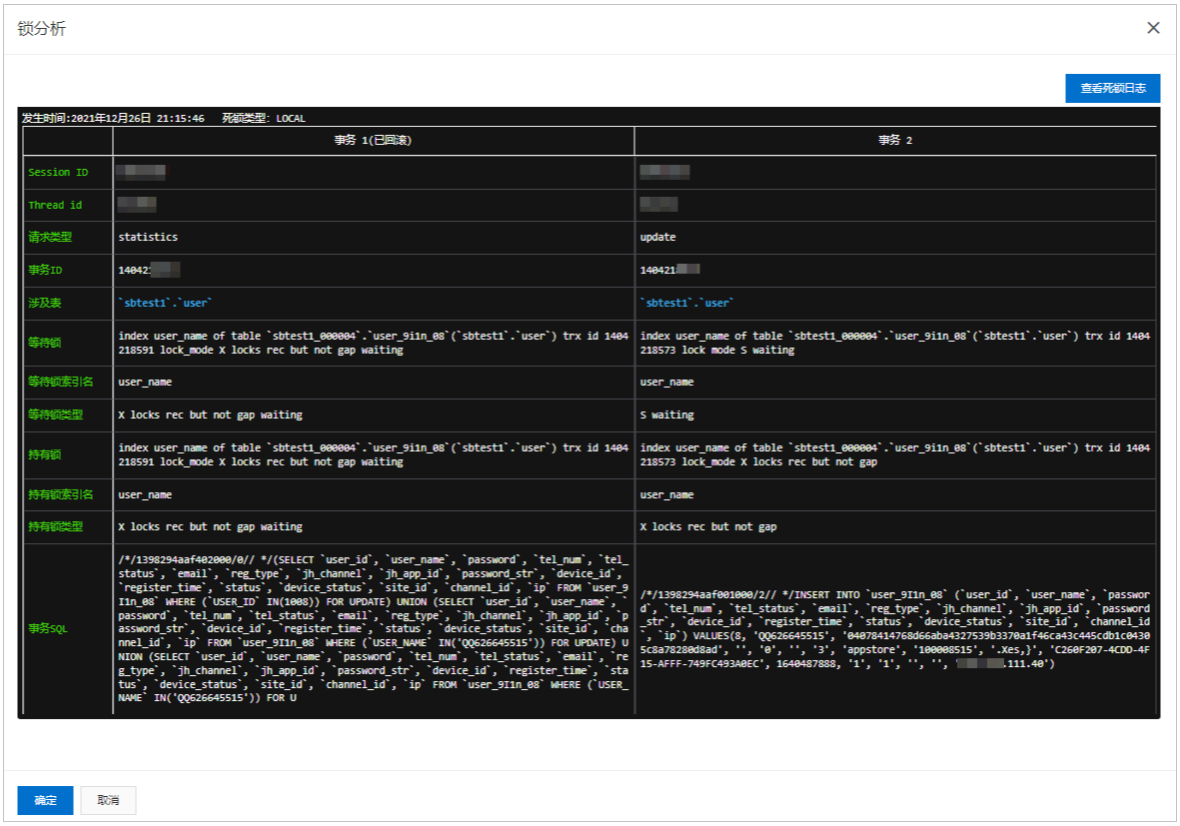
- 单击上方**多实例批量管理和Dashboard入口**，可跳转至DAS控制台。
- 单击**计算节点(CN)**或**数据节点(DN)**，可分别查看计算节点或数据节点的容量信息。
- 单击**节点**下拉列表可查看不同节点的信息。

7.9.9. 锁分析

本文介绍了锁分析的具体操作步骤。

操作步骤

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**诊断与优化 > 锁分析**。
6. 在**锁分析**页面，单击左侧**立即诊断**。
7. 在新生成的死锁诊断列表右侧，单击**详情**栏中的**查看详情**。
8. 在锁分析对话框中，查看死锁诊断详情列表，您也可以单击**查看死锁日志**，查看最近一次的死锁日志详情。



7.10. 备份与恢复

7.10.1. 备份数据

PolarDB-X支持自动备份及手动备份，方便您恢复历史数据。 本文介绍数据备份的相关功能。

功能介绍

备份类型

PolarDB-X的备份由数据备份和日志备份组成。

- **数据备份**：系统对数据进行备份，并生成备份集，目前采用的是物理备份。
- **日志备份**：也叫增量备份，备份了记录数据变更的binlog日志。开启日志备份后，基于“数据备份+日志备份”，您可以恢复时间范围内任意时间点（精确至秒）的数据。例如，实例中创建了2021年01月01日00:00:01的数据备份集以及该时间之后的日志，则可以恢复2021年01月01日00:00:01以来任意时间点（精确至秒）的数据。

备份方式

PolarDB-X支持自动备份与手动备份两种方式。

- **自动备份**：系统自动定期触发的数据备份
  - 自动备份策略默认开启，包括数据备份和日志备份；
  - 数据备份默认为每天备份1次，日志备份持续进行，无需设置频率。您可以设置自动执行备份的时间段和周期。具体请参见设置自动备份。
- **手动备份**：用户在控制台主动触发的备份。具体请参考手动创建备份。手动备份仅支持数据备份，日志备份持续进行，无需手动触发。

> **说明**
> - 出于数据安全考虑，自动备份的频率为每周至少2次。
> - 仅当前一次的手动备份完成后才能进行下一次的手动备份。

备份存放位置

- **数据备份**：存放于阿里云提供的备份空间，不占用实例的存储空间，默认保留30天、
- **日志备份**：暂存于实例的存储空间，超过一定时间后（默认7小时）或者一定的空间后（实例存储空间的30%）会上传至阿里云提供的备份空间，默认保留7天。

  您可以设置本地日志备份的保留策略，避免日志备份占用过多存储空间，详情请参见本地日志备份设置。

> **说明** 备份空间的地域是PolarDB-X实例所在的地域，但可用区不一定是实例所在的可用区。目前暂不支持数据备份与日志备份的跨地域存储。

备份影响

PolarDB-X备份在数据节点的备节点执行，不占用主节点CPU，备节点不承接业务流量，因此不影响实例性能。

计费方式

- **数据备份**：当前免费。
- **日志备份**：当前仅收取使用实例存储空间的费用，计费方式和PolarDB-X存储空间的价格与计费方式一致，详情请参见存储空间价格。

注意事项

- 备份过程中不要进行实例的扩容，升降配等操作，避免备份失败。
- 备份过程中不要执行DDL命令，避免因锁表到导致备份失败。
- 数据备份、日志备份暂不支持下载。

设置自动备份

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 备份恢复**。
6. 单击**备份策略**。

7. 在**数据备份设置**页签，设置以下参数。

参数	说明
开启备份	开启或关闭自动备份。 **说明** 自动备份默认为开启状态。
备份集保留时间	设置自动备份的具体保留天数，取值范围为7~730天。 **说明** 超过保留天数的备份会被自动删除。
备份开始时间段	设置自动备份开始的时间，可以设置为任意时段，以小时为单位，建议设置为业务低峰期时间。
备份周期	设置自动备份的周期。 **说明** 出于安全考虑，建议自动备份的频率为每周至少2次。
日志备份保留时间	日志备份文件的保留天数。 ○ 取值范围为0~730天。 ○ 保留天数为0时，表示不保留日志备份，即关闭日志备份功能。

8. 单击**确定**即可。

> 说明
> - 自动备份默认为开启状态。
> - 超过保留天数的备份会被自动删除。
> - 出于安全考虑，建议自动备份的频率为每周至少2次。

本地日志备份设置

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 备份恢复**。
6. 单击**备份策略**。
7. 在**本地设置**页签，设置以下参数：

参数	说明
保留时长	日志备份在实例存储空间的保留时长。默认值为7小时。表示开启日志备份功能后，本地只保存最近7个小时内的日志备份文件，7个小时之前的日志备份将被清理。取值：0~168。
空间使用率不超过	本地空间使用率=本地日志备份（Binlog）大小/实例总可用空间大小，默认值为30%，表示本地日志备份空间使用率大于30%时，系统会从最早日志备份（Binlong）开始清理。取值：0~50。

参数	说明
可用空间保护	该功能默认开启，表示当实例总空间使用率超过80%或实例剩余可用空间不足5GB时，会强制从最早的日志备份（Binlog）开始清理，直到总空间使用率降到80%以下且实例剩余可用空间大于5GB。

手动创建备份

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 备份恢复**。
6. 单击**创建备份**。
7. 在弹出的对话框中，单击**确定**即可。

7.10.2. 恢复数据

PolarDB-X支持通过备份恢复历史数据。本文介绍恢复数据的相关操作步骤。

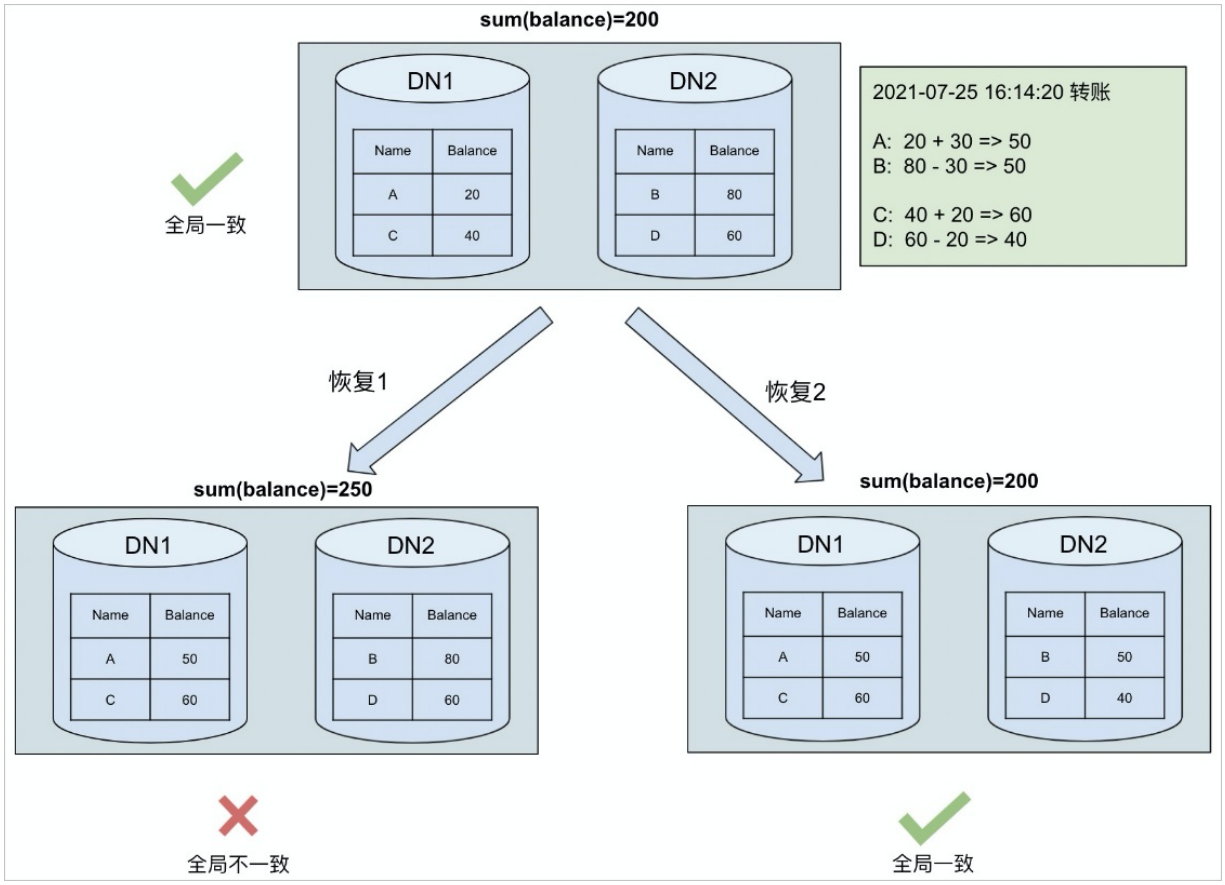
恢复方式

PolarDB-X支持按备份集恢复和按时间点恢复两种数据恢复方式。

- 按备份集恢复：依赖数据备份恢复数据，仅支持将数据恢复至数据备份的时刻。
- 按时间点恢复：依赖数据备份和日志备份恢复数据，支持将数据恢复至备份时间范围内任意时间点（精确至秒）。例如，实例已创建了2021年01月01日00:00:01的数据备份集以及该时间之后的日志，则可以恢复2021年01月01日00:00:01以来任意时间点（精确至秒）的数据。

全局一致性

PolarDB-X是一款分布式数据库，数据存放在多个数据节点（DN）上，当分布式事务存在的情况下，恢复后的实例需要保证多个数据节点间的数据一致性。下图通过转账测试给出了全局一致性的示例：



PolarDB-X中存放了一张用户的账户余额表，该表的数据分布在两个数据节点中，总的账户金额是200元。业务持续地使用分布式事务在不同的账号间进行转账。

某一时刻（2021-07-25 16:14:20），账户C向账户A转账了30元，账户D向账户C转账了20元。如果将数据恢复至该时刻，全局一致性要求恢复出的数据要么是转账前的状态，要么是转账完成后的状态，即账户的总金额需仍为200元（如恢复2），不能出现转账过程中的数据状态（如恢复1），恢复1中账户总金额为250元，数据是全局不一致的。

恢复数据的位置

PolarDB-X当前仅支持恢复数据至新实例，恢复完成后建议通过DTS等工具将数据导回原实例。

- 新实例的白名单设置、备份设置、参数设置和当前实例保持一致。
- 新实例内的数据与备份文件或指定恢复时间点时的数据一致。
- 新实例带有备份文件或时间点当时的账号信息。
- 新实例的拓扑与原实例一致，即节点个数相同。

价格和计费方式

由于数据是恢复到新实例上，因此需要收取新实例的费用。

注意事项

恢复数据需要原实例需要满足如下条件：

- 如果要按时间点进行恢复，需要确保日志备份功能已开启（即在设置数据备份策略时，**日志备份保留时间**需大于0）。
- 若要按备份集恢复，则原实例必须至少有一个物理备份集。

按时间点恢复

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。

4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 备份恢复**。
6. 单击**恢复**。

7. 在恢复实例页面，设置以下参数。

参数	说明
恢复实例	需要恢复的原实例ID。
恢复类型	固定为**克隆**，无需选择。
恢复时间	选择需要恢复的时间点。 **说明** 最早仅支持恢复到第一个备份集的创建开始时间。
地域和可用区	默认与原实例所在地域和可用区保持一致，无需选择。
网络类型	固定为**专有网络**，不可变更。
专有网络（VPC）	请确保PolarDB-X与需要连接的ECS创建于同一个VPC，否则它们无法通过内网互通，无法发挥最佳性能。 **说明** 配置专有网络前，您需要提前准备相应地域和可用区内的专有网络和虚拟交换机，详情请参见创建和管理专有网络。
交换机（VSwitch）	实例的专有网络交换机信息（VSwitch）。
MySQL版本	默认与原实例版本一致，无需选择。
节点规格	PolarDB-X 按照节点规格x节点个数的方式售卖，您可以通过选择规格配置实例物理规格CPU核数和内存大小，详情请参见规格说明。
节点个数	默认与原实例在目标时间点时拥有的节点个数保持一致，不可变更。
存储类型	默认与原实例存储类型一致，无需选择。
存储费用	每个节点存储空间默认为3 TB，购买时无需选择容量，可根据实际使用按小时计费。
购买时长	选择实例购买的时长。 **说明** 仅当原实例的商品类型为**包年包月**时支持该配置。
购买数量	固定为1，不可变更。

8. 选中服务协议，单击**立即购买**。
9. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。
10. 支付成功后，请耐心等待10~15分钟开通服务。之后您可以返回控制台的**实例列表**页，查看新创建的实例。

按备份集恢复

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。

4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 备份恢复**。
6. 在**备份恢复**页面，单击**恢复**。

7. 在恢复实例页面，设置以下参数。

参数	说明
恢复实例	需要恢复的原实例ID，自动生成，无需填写。
恢复类型	固定为**备份集**，无需选择。
备份集	恢复的数据备份集，默认选中，无需选择。
地域和可用区	默认与原实例所在地域和可用区保持一致，无需选择。如果当前可用区无库存，建议更换其他可用区。
网络类型	固定为**专有网络**，不可变更。
专有网络（VPC）	实例的专有网络（VPC）信息。请确保PolarDB-X与需要连接的ECS创建于同一个VPC，否则它们无法通过内网互通，无法发挥最佳性能。 **说明** 配置专有网络前，您需要提前准备相应地域和可用区内的专有网络和虚拟交换机，详情请参见创建和管理专有网络。
交换机（VSwitch）	实例的专有网络交换机信息（VSwitch）。
MySQL版本	默认与原实例版本一致，无需选择。
节点规格	PolarDB-X按照节点规格×节点个数的方式售卖，您可以通过选择规格配置实例物理规格CPU核数和内存大小，详情请参见规格说明。
节点个数	默认与原实例的节点个数保持一致，不可变更。
存储类型	默认与原实例存储类型一致，无需选择。
存储费用	每个节点存储空间默认为3TB，购买时无需选择容量，可根据实际使用按小时计费。
购买时长	选择实例购买的时长。 **说明** 仅当原实例的商品类型为**包年包月**时支持该配置。
购买数量	固定为1，不可变更。

8. 选中服务协议，单击**立即购买**。
9. 在**支付**页面，确认待支付订单和支付方式，单击**支付**。
10. 支付成功后，请耐心等待10~15分钟开通服务。之后您可以返回控制台的**实例列表**页，查看新创建的实例。

7.10.3. 日志备份

本文介绍了在日志备份界面下载全局Binlog文件的方法。

1. 登录云原生分布式数据库控制台。
2. 在页面左上角选择目标实例所在地域。
3. 在**实例列表**页，单击**PolarDB-X 2.0**页签。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**数据恢复 > 日志备份**。
6. 页面展示了已经备份好的所有Binglog文件，在操作列单击**下载**，可下载对应的Binglog文件。

> **说明**
> - Binglog文件大小约为500M。
> - 默认保存周期为15天。
> - 如果在保存周期内下载链接失效，请刷新页面，会生成新的下载链接。

7.11. 数据订阅

本文介绍了如何通过Binlog文件实现数据订阅功能。

功能介绍

PolarDB-X提供了全局Binlog来支持消费或订阅数据，关于全局变更日志的详细内容，请参见全局日志变更。

PolarDB-X的全局变更日志兼容MySQL Binlog文件格式，在数据订阅方式上也完全兼容MySQL Dump协议，您可以像使用单机MySQL一样来订阅PolarDB-X的事务日志。

PolarDB-X全局Binlog剔除了内部细节，避免了实例内部发生的变化对数据订阅链路的影响。PolarDB-X通过一系列的协议和算法来保证全局Binlog的服务能力，确保实例内部发生的各种变更（如HA切换、增删节点、执行Scale Out或分布式DDL等操作）不会影响数据订阅链路的正常工作。

使用限制

- 暂不支持Gtid（Global Transaction Identifier）模式下的数据订阅方式。
- 仅当事务策略指定为TSO时（即更高强度的一致性保证），才支持对分布式事务的合并。

> **说明** PolarDB-X的默认事务策略即为TSO，建议您保持默认策略即可。

数据订阅源端支持的SQL语句

1. 连接PolarDB-X数据库，详细操作步骤请参见连接数据库。
2. 在已连接PolarDB-X实例的客户端中，您可以参考如下命令，查询对应的Binlog文件：
   - 查看PolarDB-X全局Binlog文件列表，语法如下：

     ```
     SHOW BINARY LOGS;
     ```

     > **说明**
     > - 执行上述SQL需要有SUPER或REPLICATION CLIENT权限，权限操作请参见账号权限管理。
     > - 更多详情，请参见SHOW BINARY LOGS。

     返回结果如下：

     ```
     +---------------+-----------+
     | LOG_NAME      | FILE_SIZE |
     +---------------+-----------+
     | binlog.000001 |         4 |
     +---------------+-----------+
     ```

   - 查看PolarDB-X作为主Master角色的Binlog信息，语法如下：

```
SHOW MASTER STATUS;
```

> **说明**
> - 执行上述SQL需要有SUPER或REPLICATION CLIENT权限，权限操作请参见账号权限管理。
> - 更多详情，请参见SHOW MASTER STATUS。

返回结果如下：

```
*************************** 1. row ***************************
             FILE: binlog.000001
         POSITION: 4
     BINLOG_DO_DB:
 BINLOG_IGNORE_DB:
EXECUTED_GTID_SET:
```

- 查看全局Binlog文件中的具体事件信息，语法如下：

```
SHOW BINLOG EVENTS
```

> **说明**
> - 执行上述SQL需要有REPLICATION SLAVE权限，权限操作请参见账号权限管理。
> - 更多详情，请参见SHOW BINLOG EVENTS。

返回结果如下：

```
+---------------+------+------------+-----------+-------------+-------------------------------------
-----------------+
| LOG_NAME      | POS  | EVENT_TYPE | SERVER_ID | END_LOG_POS | INFO
|
+---------------+------+------------+-----------+-------------+-------------------------------------
-----------------+
| binlog.000001 |  745 | Query      |         1 |         807 | BEGIN
|
| binlog.000001 |  807 | Table_map  |         1 |         868 | table_id: 258 (transfer_test.accounts)
|
| binlog.000001 |  868 | Write_rows |         1 |         912 | table_id: 258 flags: STMT_END_F
|
| binlog.000001 |  912 | Table_map  |         1 |         973 | table_id: 367 (transfer_test.accounts)
|
| binlog.000001 |  973 | Write_rows |         1 |        1017 | table_id: 367 flags: STMT_END_F
|
| binlog.000001 | 1017 | Table_map  |         1 |        1078 | table_id: 366 (transfer_test.accounts)
|
| binlog.000001 | 1078 | Write_rows |         1 |        1122 | table_id: 366 flags: STMT_END_F
|
| binlog.000001 | 1122 | Table_map  |         1 |        1183 | table_id: 365 (transfer_test.accounts)
|
| binlog.000001 | 1183 | Write_rows |         1 |        1227 | table_id: 365 flags: STMT_END_F
|
| binlog.000001 | 1227 | Table_map  |         1 |        1288 | table_id: 365 (transfer_test.accounts)
|
| binlog.000001 | 1288 | Write_rows |         1 |        1332 | table_id: 365 flags: STMT_END_F
|
| binlog.000001 | 1332 | Xid        |         1 |        1363 | COMMIT /* xid=1 */
|
| binlog.000001 | 1363 | Rows_query |         1 |        1441 | 67770739906675434241308139189744701440
0000000000132369 |
+---------------+------+------------+-----------+-------------+-------------------------------------
-----------------+
```

数据订阅目标端支持的SQL语句

如果数据订阅目标端是标准MySQL，目前支持MySQL的Replicate指令。

- 在数据订阅目标端设置需要同步的源端数据源信息。

```
CHANGE MASTER TO option [, option] ... [ channel_option ]
option: {
    MASTER_BIND = 'interface_name'
  | MASTER_HOST = 'host_name'
  | MASTER_USER = 'user_name'
  | MASTER_PASSWORD = 'password'
  | MASTER_PORT = port_num
  | PRIVILEGE_CHECKS_USER = {'account' | NULL}
  | REQUIRE_ROW_FORMAT = {0|1}
  | REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF}
  | ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | uuid}
  | MASTER_LOG_FILE = 'source_log_name'
  | MASTER_LOG_POS = source_log_pos
  | MASTER_AUTO_POSITION = {0|1}
  | RELAY_LOG_FILE = 'relay_log_name'
  | RELAY_LOG_POS = relay_log_pos
  | MASTER_HEARTBEAT_PERIOD = interval
  | MASTER_CONNECT_RETRY = interval
  | MASTER_RETRY_COUNT = count
  | SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}
  | MASTER_DELAY = interval
  | MASTER_COMPRESSION_ALGORITHMS = 'value'
  | MASTER_ZSTD_COMPRESSION_LEVEL = level
  | MASTER_SSL = {0|1}
  | MASTER_SSL_CA = 'ca_file_name'
  | MASTER_SSL_CAPATH = 'ca_directory_name'
  | MASTER_SSL_CERT = 'cert_file_name'
  | MASTER_SSL_CRL = 'crl_file_name'
  | MASTER_SSL_CRLPATH = 'crl_directory_name'
  | MASTER_SSL_KEY = 'key_file_name'
  | MASTER_SSL_CIPHER = 'cipher_list'
  | MASTER_SSL_VERIFY_SERVER_CERT = {0|1}
  | MASTER_TLS_VERSION = 'protocol_list'
  | MASTER_TLS_CIPHERSUITES = 'ciphersuite_list'
  | MASTER_PUBLIC_KEY_PATH = 'key_file_name'
  | GET_MASTER_PUBLIC_KEY = {0|1}
  | NETWORK_NAMESPACE = 'namespace'
  | IGNORE_SERVER_IDS = (server_id_list)
}
channel_option:
    FOR CHANNEL channel
server_id_list:
    [server_id [, server_id] ... ]
```

- 开启主备同步

```
START {SLAVE | REPLICA}
```

- 停止主备同步

```
STOP {SLAVE | REPLICA}
```

- 重置主备同步，需要先停止主备同步

```
RESET {SLAVE | REPLICA} [ALL] [channel_option]
channel_option:
    FOR CHANNEL channel
```

> **说明** 如果目标端是PolarDB-X，目前暂时不支持相关Replicate指令。

8.开发指南

8.1. 开发限制

本文介绍了PolarDB-X数据库开发过程中的使用限制。

PolarDB-X高度兼容MySQL协议和语法，但由于分布式数据库和单机数据库存在较大的架构差异，存在SQL使用限制。本文将介绍数据库开发过程中的使用限制。

常见标识符限制

类型	最大字符长度	取值范围
Database	32	大小写字母、数字和下划线（_）。
Sequence	128	任意符合Unicode编码方式的字符。
Table	64	
Column		
Partition		
Partition Key		
View		
Sequence		
Variables		
Constraint		

资源使用限制

资源	类型	数量
逻辑资源	Database	最多包含32个数据库。
	Table	每个数据库最多包含8192张表。
	Partition	每张表最多包含8192个分区。
	Column	每张表最多包含1017列。
	Sequence	每个数据库最多支持创建16384个序列。
	View	每个数据库最多支持创建8192个视图。
	Global Index	每张表最多支持创建32个全局索引。
	User	单个数据库最大支持2048个用户，用户名最大长度255。
物理资源	数据库中单个计算节点最多支持的连接数	默认不要超过20000。
	数据库中最多支持增加的计算节点个数	一次性默认最多支持增加99个节点，如需添加更多，请联系阿里云技术支持。
	数据库中最多支持增加的存储节点个数	

SQL语法使用限制

SQL语法	操作	使用约束
自定义操作	自定义函数	暂不支持。
	自定义类型	暂不支持。
	存储过程	暂不支持。
	触发器	暂不支持。
	游标	暂不支持。
	视图	支持。
DDL	CREATE TABLE ... LIKE ...	暂不支持拆分表。
	CREATE TABLE ... SELECT ...	暂不支持拆分表。
	RENAME TABLE	暂不支持同时RENAME多表。
	ALTER TABLE	暂不支持ALTER TABLE修改拆分字段类型。
	Merge/Split/Add/Drop 二级分区表	暂不支持。
	Merge/Split/Add/Drop 索引分区表	暂不支持。
	外键	暂不支持。
DML	STRAIGHT_JOIN	暂不支持。
	NATURAL JOIN	暂不支持。
	INSERT DELAYED	暂不支持。
	对变量的引用和操作（例如SET @c=1，@d=@c+1；SELECT @c，@d）	暂不支持。
	LOAD XML	暂不支持。
DQL	HAVING子句中包含子查询	暂不支持。
	JOIN ON子句中包含子查询	暂不支持。
	等号操作行符的标量子查询（The Subquery as Scalar Operand）	暂不支持。
数据库管理	SHOW WARNINGS	暂不支持LIMIT和COUNT的组合。
	SHOW ERRORS	暂不支持LIMIT和COUNT的组合。
	HELP	暂不支持。
运算符	:=	暂不支持。
函数	全文检索函数	暂不支持。
	XML函数	暂不支持。
	GTID函数	暂不支持。
类型	空间类型（GEOMETRY/LINESTRING...）	暂不支持。
	Json类型	暂不支持做分区键。

SQL语法	操作	使用约束
关键字	MILLISECOND	暂不支持。
	MICROSECOND	暂不支持。

8.2. SQL概述

本文简单介绍了PolarDB-X语法及操作。

阿里云分布式数据库PolarDB-X是一款分布式关系数据库，高度兼容MySQL语法，但由于分布式数据库和单机数据库架构的差异，又有着自身的语法特点。所以本章节除了介绍数据类型、运算符和函数等基本元素，还会从下面五种类别介绍PolarDB-X的语法。

- 数据定义语言DDL（Data Definition Language）：对数据库中资源进行定义、修改和删除，如新建表和删除表等。
- 数据操作语言DML（Data Manipulation Language）：用以改变数据库中存储的数据内容，即增加、修改和删除数据。
- 数据查询语言DQL（Data Query Language）：也称为数据检索语言，用以从表中获得数据，并描述怎样将数据返回给程序输出。
- 事务控制语言TCL（Transaction Control Language）：保证数据库的完整性、一致性，在同一个事务中的TCL语句要么同时成功，要么同时失败。
- 数据控制语句DAL（Data Access Language）：相关运维指令。可以配合这些运维指令，协助DBA操作和优化数据库。

8.3. 元数据库和数据字典

云原生分布式数据库PolarDB-X的元数据库为information_schema库，兼容MySQL的元数据库。查询元数据库可以直接在JDBC连接中使用SQL语句进行查询。

Information_schema视图	兼容情况
SCHEMATA	兼容
TABLES	兼容
COLUMNS	兼容
STATISTICS	兼容
COLUMN_STATISTICS	直方图信息为PolarDB-X格式
ENGINES	兼容
KEYWORDS	兼容
COLLATIONS	兼容
TABLE_CONSTRAINTS	兼容
PROCESSLIST	兼容

Information_schema视图	兼容情况
SESSION_VARIABLES	兼容
GLOBAL_VARIABLES	兼容
INNODB_LOCKS	兼容
INNODB_TRX	兼容
INNODB_BUFFER_PAGE	兼容
INNODB_BUFFER_POOL_STATS	兼容
INNODB_BUFFER_PAGE_LRU	兼容
INNODB_LOCK_WAITS	兼容
USER_PRIVILEGES	兼容
PARTITIONS	仅支持PolarDB-X分区表
WORKLOAD	PolarDB-X HTAP负载信息
GLOBAL_INDEXES	PolarDB-X全局二级索引
METADATA_LOCK	PolarDB-X MDL锁信息
TABLE_GROUP	PolarDB-X表组信息
TABLE_DETAIL	PolarDB-X分区表各分区存储信息
LOCALITY_INFO	PolarDB-X Locality信息
PHYSICAL_PROCESSLIST	PolarDB-X CN到DN的Processlist信息
PLAN_CACHE	PolarDB-X计划缓存信息
STATISTIC_TASK	PolarDB-X统计信息任务信息
CCL_RULE	PolarDB-X CCL规则信息

Information_schema视图	兼容情况
CCL_TRIGGER	PolarDB-X CCL触发信息
SCHEMA_PRIVILEGES	未兼容
TABLE_PRIVILEGES	未兼容
INNODB_TEMP_TABLE_INFO	未兼容
INNODB_SYS_INDEXES	未兼容
INNODB_SYS_FIELDS	未兼容
INNODB_CMP_PER_INDEX_RESET	未兼容
INNODB_FT_DEFAULT_STOPWORD	未兼容
INNODB_FT_INDEX_TABLE	未兼容
INNODB_FT_INDEX_CACHE	未兼容
INNODB_SYS_TABLESPACES	未兼容
INNODB_METRICS	未兼容
INNODB_SYS_FOREIGN_COLS	未兼容
INNODB_CMPMEM	未兼容
INNODB_SYS_COLUMNS	未兼容
INNODB_SYS_FOREIGN	未兼容
INNODB_SYS_TABLESTATS	未兼容
INNODB_FT_CONFIG	未兼容
INNODB_SYS_VIRTUAL	未兼容
INNODB_CMP	未兼容

Information_schema视图	兼容情况
INNODB_FT_BEING_DELETED	未兼容
INNODB_CMP_PER_INDEX	未兼容
INNODB_CMPMEM_RESET	未兼容
INNODB_CMP_RESET	未兼容
INNODB_FT_DELETED	未兼容
INNODB_SYS_TABLES	未兼容
INNODB_SYS_DATAFILES	未兼容
PROFILING	未兼容
REFERENTIAL_CONSTRAINTS	未兼容
SESSION_STATUS	未兼容
TABLESPACES	未兼容
EVENTS	未兼容
TRIGGERS	未兼容
ROUTINES	未兼容
COLUMN_PRIVILEGES	未兼容
FILES	未兼容
KEY_COLUMN_USAGE	未兼容
OPTIMIZER_TRACE	未兼容
PARAMETERS	未兼容
CHARACTER_SETS	未兼容

Information_schema视图	兼容情况
COLLATION_CHARACTER_SET_APPLICABILITY	未兼容
PLUGINS	未兼容

SCHEMATA

SCHEMATA表提供了关于数据库的信息。包含如下列：

- CATALOG_NAME：库所属的catalog名，值固定为def.
- SCHEMA_NAME：库名字
- DEFAULT_CHARACTER_SET_NAME：库默认character set名字
- DEFAULT_COLLATION_NAME：库默认collation名字
- SQL_PATH：值固定为NULL

TABLES

TABLES表提供关于数据库表的信息。包含如下列：

- TABLE_CATALOG：表所属的catalog名，值固定为def.
- TABLE_SCHEMA：表所属的库名
- TABLE_NAME：表名
- TABLE_TYPEBASE：表类型，普通表为TABLE，视图为VIEW，INFORMATION_SCHEMA表为SYSTEM VIEW
- ENGINE：数据库存储引擎
- VERSION：版本
- ROW_FORMAT：行格式
- TABLE_ROWS：表行数估算值
- AVG_ROW_LENGTH：平均行长度
- DATA_LENGTH：主表空间估算值
- MAX_DATA_LENGTH：最大表空间值，默认为NULL
- INDEX_LENGTH：表索引空间估算值
- DATA_FREE：表空间空闲值
- AUTO_INCREMENT：下一个AUTO_INCREMENT值
- CREATE_TIME：表创建时间
- UPDATE_TIME：表更新时间
- CHECK_TIME：表校验时间，值固定为NULL
- TABLE_COLLATION：表默认collation值
- CHECKSUM：表CHECKSUM
- CREATE_OPTIONS：建表语句可选项
- TABLE_COMMENT：表注释

COLUMNS

COLUMNS表提供关于数据库列的信息。包含如下列：

- TABLE_CATALOG：列所属表所属的catalog名，值固定为def.
- TABLE_SCHEMA：列所属表所属的库名

- TABLE_NAME：列所属表名
- COLUMN_NAME：列名
- ORDINAL_POSITION：列在表中的顺序
- COLUMN_DEFAULT：列默认值
- IS_NULLABLE：列是否可以为Null
- DATA_TYPE：列类型（无精度）
- CHARACTER_MAXIMUM_LENGTH：列最大长度（单位字符）
- CHARACTER_OCTET_LENGTH：列最大长度（单位字节）
- NUMERIC_PRECISION：列数字精度
- NUMERIC_SCALE：列数字scale
- DATETIME_PRECISION：列datetime精度
- CHARACTER_SET_NAME：列character set名
- COLLATION_NAME：列collation名
- COLUMN_TYPE：列类型（包含精度）
- COLUMN_KEY：列索引信息
- EXTRA：列额外信息
- PRIVILEGES：列权限信息
- COLUMN_COMMENT：列注释
- GENERATION_EXPRESSION：生成列

STATISTICS

STATISTICS表提供关于数据库索引的信息。包含如下列：

- TABLE_CATALOG：索引所属表所属的catalog名，值固定为def.
- TABLE_SCHEMA：索引所属表所属的schema名
- TABLE_NAME：索引所属表所属的表名.
- NON_UNIQUE：索引是否唯一
- INDEX_SCHEMA：索引所属表所属的schema名
- INDEX_NAME：索引名
- SEQ_IN_INDEX：列所属索引序号
- COLUMN_NAME：列名
- COLLATION：列名collation信息
- CARDINALITY：列Cardinality值
- SUB_PART：索引前缀
- PACKED：索引PACKED信息
- NULLABLE：列是否可以为NULL
- INDEX_TYPE：索引类型
- COMMENT：索引注释（非列维度）
- INDEX_COMMENT：索引注释

有关Information_schema的更多信息，请参见Mysql官网。

8.4. 数据类型

8.4.1. 数值类型

本文介绍了PolarDB-X支持的数值类型。

PolarDB-X数值类型按精确度可以划分为如下两类：

- 精确数据类型包括：
  - 整数数据类型：TINYINT，SAMLLINT，MEDIUMINT，INTEGER和BIGINT。
  - 定点数据类型：DECIMAL和NUMERIC。
- 近似数据类型包括FLOAT、REAL、DOUBLE和PRECISION。

整体与MySQL保持一致，详细信息请参见MySQL整数类型文档。

8.4.2. 字符串类型

本文介绍了PolarDB-X支持的字符串类型。

PolarDB-X支持如下字符串类型：

- CHAR，VARCHAR
- BINARY，VARBINARY
- BLOB，TEXT
- ENUM
- SET

详细信息可参考MySQL 字符串类型文档。

8.4.3. Collation类型

字符集（Character Set）是一组字符符号及编码方式的组合，collation是建立在某一字符集上的字符排序规则。本文汇总了PolarDB-X支持的collation类型。

PolarDB-X支持下表所列的collation类型。关于collation类型的详细信息，请参见Collations。

字符集	collation
utf8	utf8_general_ci
	utf8_bin
	utf8_unicode_ci
utf8mb4	utf8mb4_general_ci
	utf8mb4_bin
	utf8mb4_unicode_ci
utf16	utf16_general_ci
	utf16_bin
	utf16_unicode_ci
ascii	ascii_general_ci
	ascii_bin
binary	binary
	latin1_swedish_ci

字符集	collation
latin1	latin1_german1_ci
	latin1_danish_ci
	latin1_bin
	latin1_general_ci
	latin1_general_cs
	latin1_spanish_ci
gbk	gbk_chinese_ci
	gbk_bin

8.4.4. 日期和时间类型

本文介绍了PolarDB-X支持的日期时间类型。

PolarDB-X支持如下日期时间类型：

- DATE
- DATETIME
- TIMESTAMP
- TIME
- YEAR

详细信息可参考MySQL日期时间类型文档。

8.4.5. Json类型

PolarDB-X支持JSON类型。

与MySQL不同，PolarDB-X支持的JSON类型暂不支持作为分区键。

详细信息请参见MySQL Json类型。

8.5. 运算符

8.5.1. 逻辑运算符

本文介绍了PolarDB-X支持的逻辑运算符。

PolarDB-X支持如下逻辑运算符：

运算符	描述
AND, &&	逻辑与
NOT, !	逻辑非
||, OR	逻辑或
XOR	逻辑异或

8.5.2. 算术运算符

本文介绍了PolarDB-X支持的算术运算符。

PolarDB-X支持如下算术运算符：

操作符	描述
/，DIV	除法
%，MOD	取余
+	加法
*	乘法
-	减法

8.5.3. 比较运算符

本文介绍了PolarDB-X支持的比较运算符。

SELECT语句中的条件语句经常要使用比较运算符。通过这些比较运算符，可以判断表中的哪些记录是符合条件的。比较结果为真，则返回1，为假则返回0，比较结果不确定则返回NULL。

PolarDB-X支持如下比较运算符：

函数名	描述
=	等于
<>, !=	不等于
>	大于
<	小于
<=	小于等于
>=	大于等于
BETWEEN	在两值之间 >=min&&<=max
NOT BETWEEN	不在两值之间
IN	在集合中
NOT IN	不在集合中
<=>	比较两个值是否相等，也可以比较NULL值 **说明** 两个操作码均为NULL时，其所得值为1；而当一个操作码为NULL时，其所得值为0。
LIKE	模糊匹配
REGEXP或RLIKE	正则式匹配
IS NULL	为空
IS NOT NULL	不为空

8.5.4. 位运算符

本文介绍了PolarDB-X支持的位运算符。

PolarDB-X支持如下位运算符：

运算符	描述
&	与运算符
~	非运算符
|	取反运算符
^	异或运算符
<<	左移运算符
>>	右移运算符

8.5.5. 赋值运算符

本文介绍了PolarDB-X支持及不支持的赋值运算符。

PolarDB-X支持 '=' 赋值运算符，一般在UPDATE的SET部分出现。

PolarDB-X暂不支持 ':=' 赋值运算符。

8.5.6. 运算符优先级

本文介绍了PolarDB-X中运算符的优先级。

PolarDB-X操作符的优先级由高到低，如下所示：

优先级	运算符
15	!
14	-(负号), ~
13	^
12	*，/，%，MOD
11	+，-
10	<<,>>
9	&
8	|
7	=(比较运算符等于)，<=>,>，>=，<，<=，<>，!=，IS，LIKE，REGEXP，IN
6	BETWEEN
5	NOT
4	AND, &&
3	XOR
2	OR, ||

优先级	运算符
1	=（赋值运算符）

特殊说明

IN/NOT IN与=优先级

在MySQL 5.7.19中执行如下SQL:

```
mysql> select binary 'a' = 'a' in (1, 2, 3);
+--------------------------------+
| binary 'a' = 'a' in (1, 2, 3) |
+--------------------------------+
|                              1 |
+--------------------------------+
1 row in set, 1 warning (0.01 sec)
mysql> show warnings;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'a' |
+---------+------+----------------------------------------+
1 row in set (0.00 sec)
mysql> select 1 in (1, 2, 3) = 'a';
+----------------------+
| 1 in (1, 2, 3) = 'a' |
+----------------------+
|                    0 |
+----------------------+
1 row in set, 1 warning (0.00 sec)
mysql> show warnings;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'a' |
+---------+------+----------------------------------------+
1 row in set (0.00 sec)
```

可见，在MySQL中，IN/NOT IN的优先级高于=（比较运算符），在PolarDB-X中，严格按照以上优先级实现，在优先级相同的情况下，采用左结合的方式。

8.6. 函数

8.6.1. 拆分函数

8.6.1.1. HASH

本文将介绍HASH函数使用方式。

描述

- 若分库和分表使用不同拆分键进行HASH时，则根据分库键的键值直接按分库数取余。如果键值是字符串，则字符串会先被换算成哈希值再进行路由计算。

  例如HASH(8)等价于8%D（D是分库数目），而HASH("ABC")等价于hashcode("ABC").abs()%D（D是分库数目）。

- 若分库和分表都使用同一个拆分键进行HASH时，则根据拆分键的键值按总的分表数取余。

  例如有2个分库，每个分库4张分表，那么0库上保存分表0~3，1库上保存分表4~7。某个键值为15，那么根据该路由方式，则该键值15将被分到1库的表7上（（15 % (2 * 4) =7））。

注意事项

HASH函数要求拆分列的值的自身分布均衡才能保证哈希均衡。

使用限制

拆分键的数据类型必须是整数类型或字符串类型。

使用场景

HASH函数主要适用于如下场景：

- 需要按用户ID或订单ID进行分库的场景；
- 拆分键是字符串类型的场景。

示例

假设需要对ID列按HASH函数进行分库不分表，则您可以使用如下DDL语句进行建表：

```
create table test_hash_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by HASH(ID);
```

8.6.1.2. STR_HASH

本文将介绍STR_HASH函数使用方式。

描述

STR_HASH函数通过指定字符串的开始位置下标与结束下标，以截取拆分键的字符串的某段子串，然后将其作为字符串（或整数）输入进行分库分表的路由计算具体的物理分片，函数如下所示：

```
STR_HASH( shardKey [, startIndex, endIndex [, valType [, randSeed ] ] ] )
```

参数	说明
shardKey	拆分键列的列名。
startIndex	目标子串的开始位置的下标。取值从0开始（即原字符串的第1个字符的下标用0表示），默认值为-1（即不做任何截取）。

参数	说明

endIndex	目标子串的结束位置的下标。取值从0开始（即原字符串的第1个字符的下标用0表示），默认值为-1（即不做任何截取）。
	startIndex和endIndex时需注意如下几种取值情况：
	• 当startIndex == j && endIndex = k (j>=0, k>=0 ,k>j)时，表示截取原字符串的[j, k)区间的字符串作为子串。例如：
	  ○ 对于字符串ABCDEFG，子串区间[1,5)的值是BCDE。
	  ○ 对于字符串ABCDEFG,，子串区间[2,2)的值是''。
	  ○ 对于字符串ABCDEFG，子串区间[4,100)的值是EFG。
	  ○ 对于字符串 ABCDEFG，子串区间[100,105)的值是''。
	• 当startIndex == -1 && endIndex = k (k>=0)时，表示截取原字符串最后k个字符作为子串，原字符串不足k个字符则直接获取整个字符串。
	• 当startIndex = k && endIndex == -1 (k>=0)时，表示截取原字符串开头k个字符作为子串，原字符串不足k个字符则直接获取整个字符串。
	• 当startIndex == -1 && endIndex == -1时， 表示不做任何截取，子串与原字符串完全一致。

valType	表示截取后的子串在计算分库分表时所使用的类型，取值范围如下： • 0（默认值）：表示PolarDB-X将截取后的子串当作字符串类型来计算路由。 • 1：表示PolarDB-X将截取后的子串当作整数类型来计算路由（子串面值的整数不能大于9223372036854775807，也不支持浮点数）
randSeed	当子串以字符串类型来计算路由的哈希值时PolarDB-X所使用的随机种子的值，通常不用需要填写，仅当用于使用默认值随机种子（randSeed=31）的STR_HASH在实际业务中出现路由不均衡的场景，达到用哈希均衡数据的目的。该参数默认值为31，可取其他值（如131，13131，1313131等）。 **说明** • 仅当valType取值为0时，才支持配置该参数。 • 该参数调整后您需要手动将所有数据导出来，再将新的拆分算法导入数据（即您需要对数据进行重分布）。

注意事项

使用STR_HASH做拆分的表仅适用于点查场景，如果在业务中范围查询，则会直接触发全表扫描导致慢查询。

使用限制

- 拆分键的数据类型需为字符串类型（CHAR或VARCHAR）。
- 不支持在建表完成后再调整STR_HASH的参数。

使用场景

- 实现一个分表（或分库）只对应一个拆分表键的取值（字符串类型）的精准路由效果。例如，某个应用是按年月（YYYYMM）分库，然后按订单号分表，该应用的订单号有个特点，就是订单号的最后3位字符串是一个整数，其取值范围是000~999。该应用的需求是需要在一个物理分库内，要将订单号后3位的每一个数值只单独路由到一个物理分表。那么，该应用分库采用YYYYMM，然后分表采用拆分函数STR_HASH，每个库1024个分表，就可以达到效果。具体的SQL语句如下：

```
create table test_str_hash_tb (
    id int NOT NULL AUTO_INCREMENT,
    order_id varchar(30) NOT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYMM(`create_time`)
tbpartition by STR_HASH(`order_id`, -1, 3, 1) tbpartitions 1024;
```

应用采用这样建表SQL，原因是分表的字符串截取后3位并转换为整数（整数范围是000~999）后再取模做分表路由（共1024个分表），其路由结果能保证每一个物理分表只对一个拆分建的取值。而原来PolarDB-X默认拆分函数HASH无法达到这样的效果，是因为字符串经过hashCode计算后的整数是不可预知的，有可能会出现一个物理分表要对应多个不同拆分建的取值。

- STR_HASH拆分函数适用于使用字符串类型作为拆分键并且绝大部分都是点查的场景，如根据ID查交易订单、物流订单等。

使用示例

假设order_id的类型为VARCHAR(32)，现在需要将order_id作为拆分键，计划分4个库，分8个表。

- 假设需要使用order_id的最后4位的字符串作为整数来计算分库分表路由，则您可以使用如下SQL进行建表。

```
create table test_str_hash_tb (
    id int NOT NULL AUTO_INCREMENT,
    order_id varchar(32) NOT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by STR_HASH(`order_id`, -1, 4, 1)
tbpartition by STR_HASH(`order_id`, -1, 4, 1) tbpartitions 2;
```

- 假设需要截取order_id的第3个字符（即startIndex=2）与第7个字符（即endIndex=7）之间子串来计算分库分表路由，则您可以使用如SQL进行建表。

```
create table test_str_hash_tb (
    id int NOT NULL AUTO_INCREMENT,
    order_id varchar(32) NOT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by STR_HASH(`order_id`, 2, 7)
tbpartition by STR_HASH(`order_id`, 2, 7) tbpartitions 2;
```

- 假设需要截取order_id的前5个字符串作为子串来计算分库分表路由，则您可以使用如SQL进行建表。

```
create table test_str_hash_tb (
    id int NOT NULL AUTO_INCREMENT,
    order_id varchar(32) NOT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by STR_HASH(`order_id`, 5, -1)
tbpartition by STR_HASH(`order_id`, 5, -1) tbpartitions 2;
```

常见问题

Q：`dbpartition by STR_HASH(order_id)` 与 `dbpartition by HASH(order_id)` 有什么区别？

A：两者虽然都是直接根据字符串取值做分库分表的哈希路由，但是两者的分库分表的路由算法实现不一样。前者支持用户建表时自行设定截取子串相关参数，且在根据字符串的哈希值计算分库分表路由时是基于UNI_HASH算法进行计算；而后者是只对字符串的哈希值做简单取模。

8.6.1.3. UNI_HASH

本文将介绍UNI_HASH的使用方式。

描述

- 使用UNI_HASH分库时，根据分库键的键值直接按分库数取余。如果键值是字符串，则字符串会被计算成哈希值再进行计算，完成路由计算，例如HASH('8')等价于8 % D（D是分库数目）。
- 分库和分表都使用同一个拆分键进行UNI_HASH时，先根据分库键键值按分库数取余，再均匀散布到该分库的各个分表上。

注意事项

UNI_HASH算法是简单取模，要求拆分列的值的自身分布均衡才能保证哈希均衡。

使用限制

拆分键的数据类型必须是整数类型或字符串类型。

使用场景

- 适合于需要按用户ID或订单ID进行分库的场景。
- 适合于拆分键是整数或字符串类型的场景。
- 两张逻辑表需要根据同一个拆分键进行分库，两张表的分表数不同，又经常会按该拆分键进行JOIN的场景。

使用示例

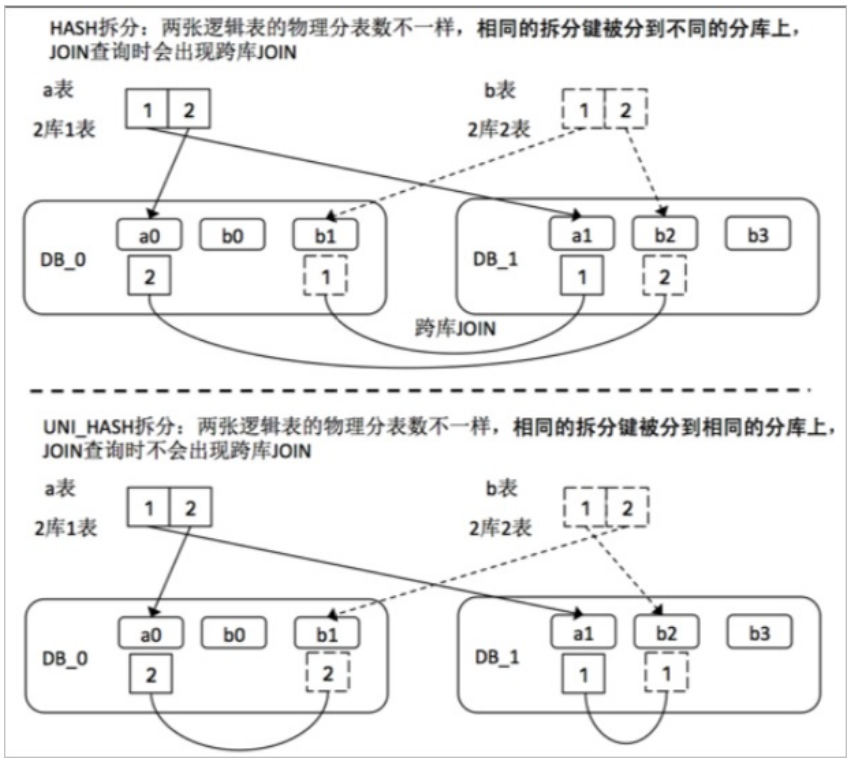
假设需要对ID列按UNI_HASH函数进行分库分表，每库包含4张表，则您可以使用如下DDL语句进行建表：

```
create table test_hash_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by UNI_HASH(ID)
tbpartition by UNI_HASH(ID) tbpartitions 4;
```

与HASH的比较

对比场景	UNI_HASH	HASH
分库不分表。	此时两个函数的路由方式一样，都是根据分库键的键值按分库数取余。	
使用同一个拆分键进行分库分表。	同一个键值分到的分库的路由结果不会随着分表数的变化而改变。	同一个键值分到的分库会随着分表数的变化而改变。
两张逻辑表需要根据同一个拆分键进行分库分表，但分表数不同。	当两张表按该拆分键进行JOIN时，不会出现跨库JOIN的情况。	当两张表按该拆分键进行JOIN时，会出现跨库JOIN的情况。

假设有2个物理分库（DB_0和DB_1），2张逻辑表（a和b），其中a表每库1张分表，b表每库2张分表。下图展示了分别使用HASH和UNI_HASH进行拆分后，a表和b表进行JOIN的情景：



8.6.1.4. RANGE_HASH

本文将介绍RANGE_HASH函数的使用方式。

描述

根据任一拆分键后N位计算哈希值，然后再按分库数取余，完成路由计算。N为函数第三个参数。

例如，RANGE_HASH(COL1, COL2, N)，计算时会优先选择COL1，截取其后N位进行计算。 COL1不存在时再选择COL2。

使用限制

- 拆分键的类型必须是字符类型或数字类型，两个拆分键类型必须保持一致。
- 两个拆分键皆不能修改。
- 折分键暂时不支持做范围查询。
- 插入数据时两个拆分键的后N位需确保一致。
- 字符串长度需不少于N位。

使用场景

适用于需要有两个拆分键，并且查询时仅有其中一个拆分键值的场景。

使用示例

假设PolarDB-X里已经分了8个物理库，现在需要按买家ID和订单ID对订单表进行分库；查询时条件仅有买家ID或订单ID，那么您可以使用如下DDL语句构建订单表：

```
create table test_order_tb (
    id int,
    buyer_id varchar(30) DEFAULT NULL,
    order_id varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8
    dbpartition by RANGE_HASH(buyer_id,order_id, 10)
    tbpartition by RANGE_HASH (buyer_id,order_id, 10) tbpartitions 3;
```

8.6.1.5. RIGHT_SHIFT

本文将介绍RIGHT_SHIFT函数的使用方式。

描述

根据分库键的键值（键值必须是整数）有符号地向右移二进制指定的位数（位数可通过DDL指定），然后将得到的整数值按分库（表）数目取余。

使用限制

拆分键的类型必须整数类型。

使用场景

当拆分键大部分键值的低位部分区分度比较低而高位部分区分度比较高时，则适用于通过此拆分函数提高散列结果的均匀度。

例如有4个拆分键的键值，分别为0x0100、0x0200、0x0300和0x0400， 这4个值的第8位部分都是0。通常一些业务后N位可能只是一些业务上的标志位，如果直接对键值进行取余散列，其散列效果可能会比较差。但如果通过RIGHT_SHIFT（shardKey, 8）将拆分键的值进行二进制右移8位，则分别变成了0x01、0x02、0x03和0x04，这样的散列效果就会比较均匀（若分4个库，刚好可以每个值对应一个分库）。

使用示例

假设需要将ID作为拆分键，并将ID的值向右移二进制8位的值作为哈希值，则可以如下建表：

```
create table test_hash_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by RIGHT_SHIFT(id, 8)
tbpartition by RIGHT_SHIFT(id, 8) tbpartitions 4;
```

8.6.1.6. MM

本文将介绍MM函数的使用方式。

描述

根据分库键时间值的月份数进行取余运算并得到分表下标。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数使用，不能作为分库函数使用。
- 按MM进行分表，由于一年的月份只有12个月，所以各分库的分表数不能超过12。

使用场景

MM函数适用于按月份数进行分表，分表的表名即为月份数。

使用示例

假设需要先按ID对用户进行分库，再将create_time列按月份进行分表，使得每个月份能够对应一张物理表，则您可以使用如下的建表DDL:

```
create table test_mm_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(id)
tbpartition by MM(create_time) tbpartitions 12;
```

8.6.1.7. DD

本文将介绍DD函数的使用方式。

描述

根据分库键时间值日期的天数进行取余运算并得到分表下标。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数使用，不能作为分库函数使用。
- 按DD进行分表，由于一个月中日期（DATE_OF_MONTH）的取值范围是1~31，所以各分库的分表数不能超过31。

使用场景

DD函数适用于按日期的天数进行分表，分表的表名即为日期的天数。

使用示例

假设需要先按ID对用户进行分库，再将create_time列按日期进行分表，使得每个日期能够对应一张物理表，则您可以使用如下的建表DDL:

```
create table test_dd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(id)
tbpartition by DD(create_time) tbpartitions 31;
```

8.6.1.8. YYYYDD

本文介绍YYYYDD函数的使用方式。

描述

根据分库键时间值的年份与一年的天数计算哈希值，然后再按分库数进行取余，完成路由计算。

例如，`YYYYDD('2012-12-31 12:12:12')` 函数等价于按照(2012x366+366)%D（D是分库数目）公式计算出2012-12-31是2012年的第366天。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 使用YYYYDD函数前，需要先确定所需的总物理分表数，您可以通过确定循环周期（如2年）来确定总的物理分表数。因为YYYYDD函数仅支持为循环周期内的每一天创建一张独立分表。

- 当日期经过一个循环周期后（如2012-03-01经过一个2年的循环周期后是2014-03-01），同一个日期有可能被路由到同一个分库分表，具体被分到哪个分表受实际的分表数目影响。

使用场景

YYYYDD函数适用于需要按年份与一年的天数进行分库的场景。建议结合该函数与`tbpartition by YYYYDD(ShardKey)`命令一起使用。

使用示例

假设PolarDB-X里已经拥有8个物理库，现有如下需求：

- 按年天进行分库。
- 同一天的数据都能落在同一张分表，且两年以内的每一天都能单独对应一张分表。
- 查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

上述需求中提到两年以内的每一天都需对应一张分表（即一天一张表），由于一年最多有366天，所以需要创建732（366x2=732）张物理分表才能满足上述需求。PolarDB-X已有8个分库，所以每个分库应该建92张物理分表（732/8=91.5，取整为92，分表数最好是分库数的整数倍）。

则您可以使用如下建表DDL：

```
create table test_yyyydd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYDD(create_time)
tbpartition by YYYYDD(create_time) tbpartitions 92;
```

8.6.1.9. WEEK

本文介绍WEEK函数的使用方式。

描述

根据分库键的时间值所对应的一周之中的日期进行取余运算并得到分表下标。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数使用，但不能作为分库函数。

使用场景

WEEK适用于按周数的日期进行分表，分表表名的下标分别对应一周中的各个日期（星期一到星期天）。

使用示例

假设先按ID对用户进行分库，再需要对create_time列按周进行分表，并且每周7天（星期一到星期天）各对应一张物理表，则应该使用如下的建表DDL：

```
create table test_week_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(name)
tbpartition by WEEK(create_time) tbpartitions 7;
```

> **说明** 当按WEEK进行分表时，由于一周共有7天，所以各分库的分表数不能超过7张。

8.6.1.10. MMDD

本文介绍MMDD函数的使用方式。

描述

根据分库键时间值在一年中所对应的日期进行取余运算并得到分表下标。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数使用，但不能作为分库函数。
- 按MMDD进行分表，由于一年最多只有366天，所以各个分库的分表数目不能超过366。

使用场景

MMDD函数适用于按一年中的日期进行分表，分表的表名下标就是一年中的某个日期。

使用示例

假设需要先按ID对用户进行分库，再将create_time列按一年中的日期（包括月份与日期）进行建表，使得一年中每一天的日期都能对应一张物理表，则您可以使用如下的建表DDL:

```
create table test_mmdd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(name)
tbpartition by MMDD(create_time) tbpartitions 366;
```

8.6.1.11. YYYYMM

本文介绍YYYYMM函数的使用方式。

描述

根据拆分键时间值的年份与月份计算哈希值，然后再按分库数进行取余。

例如，`YYYYMM('2012-12-31 12:12:12')`等价于按照(2012x12+12)%D（D是分库数目）公式计算出2012-12-31是2012年的第12个月。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 使用YYYYMM函数前，需要先确定所需的总物理分表数，您可以通过确定循环周期（如2年）来确定总的物理分表数。因为YYYYMM函数仅支持为循环周期内的每一个月创建一张独立分表。
- 当月份经过一个循环周期后（如2012-03经过一个2年的循环周期后是2014-03），相同月份有可能被路由到同一个分库分表，具体被分到哪个分表受实际的分表数目影响。

使用场景

适合于需要按年份与月份进行分库的场景，建议结合该函数与`tbpartition by YYYYMM(ShardKey)`命令一起使用。

使用示例

假设PolarDB-X里已经拥有8个物理库，现有如下需求：

- 按年月进行分库。

- 同一个月的数据都能落在同一张分表，且两年以内的每一个月都能单独对应一张分表。
- 查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

YYYYMM分库函数即可满足上述要求。上述需求中提到两年以内的每个月都需对应一张分表（即一个月一张表），由于一年有12个月，所以需要创建24（12x2=24）张物理分表才能满足上述需求。PolarDB-X已有8个分库，所以每个分库应该建3（24/8=3）张物理分表。

则您可以使用如下建表DDL:

```
create table test_yyyymm_tb (
        id int,
        name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYMM(create_time)
tbpartition by YYYYMM(create_time) tbpartitions 3;
```

8.6.1.12. YYYYWEEK

本文将介绍YYYYWEEK函数的使用方式。

描述

根据分库键时间值的年份与一年的周数计算哈希值，然后再按分库数进行取余。

例如，`YYYYWEEK('2012-12-31 12:12:12')`等价于按照(2013x54+1)%D（D是分库数目）公式计算出2012-12-31是2013年的第1周。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 使用YYYYWEEK函数前，需要先确定所需的总物理分表数，您可以通过确定循环周期（如2年）来确定总的物理分表数。因为YYYYWEEK函数仅支持为循环周期内的每一周创建一张独立分表。
- 当周数经过一个循环周期后（如2012年第1周经过一个2年的循环周期后是2014年第1周），相同周数有可能被路由到同一个分库分表，具体被分到哪个分表受实际的分表数目影响。

使用场景

YYYYWEEK函数适用于需要按年份与一年的周数进行分库的场景。建议结合该函数与`tbpartition by YYYYWEEK(ShardKey)`命令一起使用。

使用示例

假设PolarDB-X里已经拥有8个物理库，现有如下需求：

- 按年周进行分库。
- 同一周的数据都能落在同一张分表，且两年以内的每一周都能单独对应一张分表。
- 查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

YYYYWEEK分库函数即可满足上述要求。上述需求中提到两年以内的每一周都需对应一张分表（即一周一张表），由于一年最多有53周，所以需要创建106（53x2=106）张物理分表才能满足上述需求。PolarDB-X已有8个分库，所以每个分库应该建14张物理分表（106/8=13.25，取整为14，分表数最好是分库数的整数倍）。

则您可以使用如下建表DDL:

```
create table test_yyyyweek_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYWEEK(create_time)
tbpartition by YYYYWEEK(create_time) tbpartitions 14;
```

8.6.2. 日期时间函数

本文介绍了PolarDB-X支持的日期时间函数。

支持函数

PolarDB-X支持如下日期时间函数：

函数名	描述	示例

ADDDATE(d,n)	计算起始日期d加上n天的日期	`SELECT ADDDATE("2017-06-15", INTERVAL 10 DAY);`
		返回：
		`->2017-06-25`

ADDTIME(t,n)	n是一个时间表达式，时间t加上时间表达式n	• 加5秒：
		`SELECT ADDTIME('2011-11-11 11:11:11', 5);`
		返回：
		`->2011-11-11 11:11:16 (秒)`
		• 添加 2 小时10 分钟5 秒：
		`SELECT ADDTIME("2020-06-15 09:34:21", "2:10:5");`
		返回：
		`-> 2020-06-15 11:44:26`

CURDATE()	返回当前日期	`SELECT CURDATE();`
		返回：
		`-> 2018-09-19`

CURRENT_DATE()	返回当前日期	`SELECT CURRENT_DATE();`
		返回：
		`-> 2018-09-19`

CURRENT_TIME()	返回当前时间	`SELECT CURRENT_TIME();`
		返回：
		`-> 19:59:02`

CURRENT_TIMESTAMP()	返回当前日期和时间	`SELECT CURRENT_TIMESTAMP();`
		返回：
		`-> 2018-09-19 20:57:43`

CURTIME()	返回当前时间	`SELECT CURTIME();`
		返回：
		`-> 19:59:02`

函数名	描述	示例

DATE()	从日期或日期时间表达式中提取日期值	`SELECT DATE("2017-06-15");`
		返回：
		`-> 2017-06-15`

DATEDIFF(d1,d2)	计算日期d1和d2之间相隔的天数	`SELECT DATEDIFF('2001-01-01','2001-02-02');`
		返回：
		`-> -32`

DATE_ADD(d，INTERVAL expr type)	计算起始日期d加上一个时间段后的日期，type值可以是：
	• MICROSECOND
	• SECOND
	• MINUTE
	• HOUR
	• DAY
	• WEEK
	• MONTH
	• QUARTER
	• YEAR
	• SECOND_MICROSECOND
	• MINUTE_MICROSECOND
	• MINUTE_SECOND
	• HOUR_MICROSECOND
	• HOUR_SECOND
	• HOUR_MINUTE
	• DAY_MICROSECOND
	• DAY_SECOND
	• DAY_MINUTE
	• DAY_HOUR
	• YEAR_MONTH
		• `SELECT DATE_ADD("2017-06-15", INTERVAL 10 DAY);`
		返回：
		`-> 2017-06-25`
		• `SELECT DATE_ADD("2017-06-15 09:34:21", INTERVAL 15 MINUTE);`
		`-> 2017-06-15 09:49:21`
		• `SELECT DATE_ADD("2017-06-15 09:34:21", INTERVAL -3 HOUR);`
		返回：
		`->2017-06-15 06:34:21`

DATE_FORMAT(d,f)	按表达式f的要求显示日期d	`SELECT DATE_FORMAT('2011-11-11 11:11:11','%Y-%m-%d %r');`
		返回：
		`-> 2011-11-11 11:11:11 AM`

DATE_SUB(date,INTERVAL expr type)	函数从日期减去指定的时间间隔	Orders表中OrderDate字段减去2天： `SELECT OrderId,DATE_SUB(OrderDate,INTERVAL 2 DAY) AS OrderPayDate FROM Orders`

DAY(d)	返回日期值d的日期部分	`SELECT DAY("2017-06-15");`
		返回：
		`-> 15`

函数名	描述	示例

DAYNAME(d)	返回日期d是星期几，如Monday，Tuesday	`SELECT DAYNAME('2011-11-11 11:11:11');`
		返回：
		`->Friday`

DAYOFMONTH(d))	计算日期d是本月的第几天	`SELECT DAYOFMONTH('2011-11-11 11:11:11');`
		返回：
		`->11`

DAYOFWEEK(d)	日期d今天是星期几，1星期日，2星期一，以此类推	`SELECT DAYOFMONTH('2011-11-11 11:11:11');`
		返回：
		`->6`

DAYOFYEAR(d)	计算日期d是本年的第几天	`SELECT DAYOFMONTH('2011-11-11 11:11:11');`
		返回：
		`->315`

EXTRACT(type FROM d)	从日期d中获取指定的值，type指定返回的值。
	type可取值为：
	• MICROSECOND
	• SECOND
	• MINUTE
	• HOUR
	• DAY
	• WEEK
	• MONTH
	• QUARTER
	• YEAR
	• SECOND_MICROSECOND
	• MINUTE_MICROSECOND
	• MINUTE_SECOND
	• HOUR_MICROSECOND
	• HOUR_SECOND
	• HOUR_MINUTE
	• DAY_MICROSECOND
	• DAY_SECOND
	• DAY_MINUTE
	• DAY_HOUR
	• YEAR_MONTH
		`SELECT DAYOFMONTH('2011-11-11 11:11:11');`
		返回：
		`->11`

函数名	描述	示例

FROM_DAYS(n)	计算从0000年1月1日开始n天后的日期	`SELECT FROM_DAYS(1111)`
		返回：
		`-> 0003-01-16`

HOUR(t)	返回t中的小时值	`SELECT HOUR('1:2:3')`
		返回：
		`-> 1`

LAST_DAY(d)	返回给定日期的那一月份的最后一天	`SELECT LAST_DAY("2017-06-20");`
		返回：
		`-> 2017-06-30`

LOCALTIME()	返回当前日期和时间	`SELECT LOCALTIME()`
		返回：
		`-> 2018-09-19 20:57:43`

LOCALTIMESTAMP()	返回当前日期和时间	`SELECT LOCALTIMESTAMP()`
		返回：
		`-> 2018-09-19 20:57:43`

MAKEDATE(year, day-of-year)	基于给定年份year和所在年中的天数序号day-of-year返回一个日期	`SELECT MAKEDATE(2017, 3);`
		返回：
		`-> 2017-01-03`

MAKETIME(hour, minute, second)	组合时间，参数分别为小时、分钟、秒	`SELECT MAKETIME(11, 35, 4);`
		返回：
		`-> 11:35:04`

MICROSECOND(date)	返回日期所对应的微秒数	`SELECT MICROSECOND("2017-06-20 09:34:00.000023");`
		返回：
		`-> 23`

MINUTE(t)	返回t中的分钟值	`SELECT MINUTE('1:2:3');`
		返回：
		`-> 2`

函数名	描述	示例

MONTHNAME(d)	返回日期当中的月份名称，如 November	`SELECT MONTHNAME('2011-11-11 11:11:11');`
		返回：
		`-> November`

MONTH(d)	返回日期d中的月份值，1到12	`SELECT MONTH('2011-11-11 11:11:11');`
		返回：
		`->11`

NOW()	返回当前日期和时间	`SELECT NOW()`
		返回：
		`-> 2018-09-19 20:57:43`

PERIOD_ADD(period, number)	为年-月组合日期添加一个时段	`SELECT PERIOD_ADD(201703, 5);`
		返回：
		`-> 201708`

PERIOD_DIFF(period1, period2)	返回两个时段之间的月份差值	`SELECT PERIOD_DIFF(201710, 201703);`
		返回：
		`-> 7`

QUARTER(d)	返回日期d是第几季度，返回1到4	`SELECT QUARTER('2011-11-11 11:11:11');`
		返回：
		`-> 4`

SECOND(t)	返回t中的秒钟值	`SELECT SECOND('1:2:3')`
		返回：
		`-> 3`

SEC_TO_TIME(s)	将以秒为单位的时间s转换为时分秒的格式	`SELECT SEC_TO_TIME(4320);`
		返回：
		`-> 01:12:00`

STR_TO_DATE(string, format_mask)	将字符串转变为日期	`SELECT STR_TO_DATE("August 10 2017", "%M %d %Y");`
		返回：
		`-> 2017-08-10`

函数名	描述	示例
SUBDATE(d,n)	日期d减去n天后的日期	`SELECT SUBDATE('2011-11-11 11:11:11', 1);` 返回： `->2011-11-10 11:11:11`（默认是天）
SUBTIME(t,n)	时间t减去n秒的时间	`SELECT SUBTIME('2011-11-11 11:11:11', 5);` 返回： `->2011-11-11 11:11:06`（秒）

SYSDATE()	返回当前日期和时间	`SELECT SYSDATE();`
		返回：
		`-> 2018-09-19 20:57:43`

TIME(expression)	提取传入表达式的时间部分	`SELECT TIME("19:30:10");`
		返回：
		`-> 19:30:10`

TIME_FORMAT(t,f)	按表达式f的要求显示时间t	`SELECT TIME_FORMAT('11:11:11','%r');`
		返回：
		`-> 11:11:11 AM`

TIME_TO_SEC(t)	将时间t转换为秒	`SELECT TIME_TO_SEC('1:12:00');`
		返回：
		`-> 4320`

TIMEDIFF(time1, time2)	计算时间差值	`SELECT TIMEDIFF("13:10:11", "13:10:10");`
		返回：
		`-> 00:00:01`

TIMESTAMP(expression, interval)	单个参数时，函数返回日期或日期时间表达式；有2个参数时，将参数加和	`SELECT TIMESTAMP("2017-07-23", "13:10:11");`
		返回：
		`-> 2017-07-23 13:10:11`

TO_DAYS(d)	计算日期d距离0000年1月1日的天数	`SELECT TO_DAYS('0001-01-01 01:01:01');`
		返回：
		`-> 366`

函数名	描述	示例

WEEK(d)	计算日期d是本年的第几个星期，范围是0到53	`SELECT WEEK('2011-11-11 11:11:11');`
		返回：
		`-> 45`

WEEKDAY(d)	日期d是星期几，0表示星期一，1表示星期二	`SELECT WEEKDAY("2017-06-15");`
		返回：
		`-> 3`

WEEKOFYEAR(d)	计算日期d是本年的第几个星期，范围是0到53	`SELECT WEEKOFYEAR('2011-11-11 11:11:11');`
		返回：
		`-> 45`

YEAR(d)	返回年份	`SELECT YEAR("2017-06-15");`
		返回：
		`-> 2017`

YEARWEEK(date, mode)	返回年份及第几周（0到53），mode中0表示周天，1表示周一，以此类推	`SELECT YEARWEEK("2017-06-15");`
		返回：
		`-> 201724`

8.6.3. 字符串函数

本文介绍了PolarDB-X支持及不支持的字符串函数。

支持函数

PolarDB-X支持如下字符串函数：

函数名	描述	示例
ASCII(s)	返回字符串s的第一个字符的ASCII码	返回CustomerName字段第一个字母的ASCII码： `SELECT ASCII(CustomerName) AS NumCodeOfFirstChar FROM Customers;`
CHAR_LENGTH(s)	返回字符串s的字符数	返回字符串RUNOOB的字符数： `SELECT CHAR_LENGTH("RUNOOB") AS LengthOfString;`
CHARACTER_LENGTH(s)	返回字符串s的字符数	返回字符串RUNOOB的字符数： `SELECT CHARACTER_LENGTH("RUNOOB") AS LengthOfString;`

函数名	描述	示例
CONCAT(s1,s2...sn)	字符串s1,s2等多个字符串合并为一个字符串	合并多个字符串： `SELECT CONCAT("SQL ", "Runoob ", "Gooogle ", "Facebook") AS ConcatenatedString;`
CONCAT_WS(x, s1,s2...sn)	同CONCAT(s1,s2,...)函数，但是每个字符串之间要加上x，x可以是分隔符	合并多个字符串，并添加分隔符： `SELECT CONCAT_WS("-", "SQL", "Tutorial", "is", "fun!") AS ConcatenatedString;`
FIELD(s,s1,s2...)	返回第一个字符串s在字符串列表(s1,s2...)中的位置	返回字符串c在列表值中的位置： `SELECT FIELD("c", "a", "b", "c", "d", "e");`
FIND_IN_SET(s1,s2)	返回在字符串s2中与s1匹配的字符串的位置	返回字符串c在指定字符串中的位置： `SELECT FIND_IN_SET("c", "a,b,c,d,e");`

FORMAT(x,n)	函数可以将数字x进行格式化"#,###.##", 将x保留到小数点后n位，最后一位四舍五入。	格式化数字"#,###.##"形式：
		`SELECT FORMAT(250500.5634, 2);`
		返回：
		`-- 250,500.56`

INSERT(s1,x,len,s2)	字符串s2替换s1的x位置开始长度为len的字符串	从字符串第一个位置开始的6个字符替换为runoob：
		`SELECT INSERT("example.com", 1, 6, "runoob");`
		返回：
		`-- example.com`

LOCATE(s1,s)	从字符串s中获取s1的开始位置	• 获取st在字符串myteststring中的位置：
		`SELECT LOCATE('st','myteststring');`
		返回：
		`-- 5`
		• 返回字符串abc中b的位置：
		`SELECT LOCATE('b', 'abc');`
		返回：
		`-- 2`

LCASE(s)	将字符串s的所有字母变成小写字母	字符串RUNOOB转换为小写：
		`SELECT LCASE('RUNOOB');`
		返回：
		`-- runoob`

函数名	描述	示例

LEFT(s,n)	返回字符串s的前n个字符	返回字符串runoob中的前两个字符：
		`SELECT LEFT('runoob',2);`
		返回：
		`-- ru`

LOWER(s)	将字符串s的所有字母变成小写字母	字符串RUNOOB转换为小写：
		`SELECT LOWER('RUNOOB');`
		`-- runoob`

LPAD(s1,len,s2)	在字符串s1的开始处填充字符串s2，使字符串长度达到len	将字符串xx填充到abc字符串的开始处：
		`SELECT LPAD('abc',5,'xx')`
		返回：
		`-- xxabc`

LTRIM(s)	去掉字符串s开始处的空格	去掉字符串RUNOOB开始处的空格：
		`SELECT LTRIM("RUNOOB") AS LeftTrimmedString;`
		返回：
		`-- RUNOOB`

MID(s,n,len)	从字符串s的n位置截取长度为len的子字符串，同SUBSTRING(s,n,len)	从字符串RUNOOB中的第2个位置截取3个字符：
		`SELECT MID("RUNOOB", 2, 3) AS ExtractString;`
		返回：
		`-- UNO`

POSITION(s1 IN s)	从字符串s中获取s1的开始位置	返回字符串abc中b的位置：
		`SELECT POSITION('b' in 'abc');`
		返回：
		`-- 2`

REPEAT(s,n)	将字符串s重复n次	将字符串runoob重复三次：
		`SELECT REPEAT('runoob',3;`
		返回：
		`-- runoobrunoobrunoob`

REPLACE(s,s1,s2)	将字符串s2替代字符串s中的字符串s1	将字符串abc中的字符a替换为字符x：
		`SELECT REPLACE('abc','a','x');`
		返回：
		`--xbc`

函数名	描述	示例

REVERSE(s)	将字符串s的顺序反过来	将字符串abc的顺序反过来：
		`SELECT REVERSE('abc');`
		返回：
		`-- cba`

RIGHT(s,n)	返回字符串s的后n个字符	返回字符串runoob的后两个字符：
		`SELECT RIGHT('runoob',2);`
		返回：
		`-- ob`

RPAD(s1,len,s2)	在字符串s1的结尾处添加字符串s2，使字符串的长度达到len	将字符串xx填充到abc字符串的结尾处：
		`SELECT RPAD('abc',5,'xx');`
		返回：
		`-- abcxx`

RTRIM(s)	去掉字符串s结尾处的空格	去掉字符串RUNOOB的末尾空格：
		`SELECT RTRIM("RUNOOB") AS RightTrimmedString;`
		返回：
		`-- RUNOOB`

SPACE(n)	返回n个空格	返回10个空格： `SELECT SPACE(10);`

STRCMP(s1,s2)	比较字符串s1和s2，如果s1与s2相等返回0，如果s1>s2返回1，如果s1<s2，返回-1	比较字符串：
		`SELECT STRCMP("runoob", "runoob");`
		返回：
		`-- 0`

SUBSTR(s, start, length)	从字符串s的start位置截取长度为length的子字符串	从字符串RUNOOB中的第2个位置截取3个字符：
		`SELECT SUBSTR("RUNOOB", 2, 3) AS ExtractString;`
		返回：
		`-- UNO`

SUBSTRING(s, start, length)	从字符串s的start位置截取长度为length的子字符串	从字符串RUNOOB中的第2个位置截取3个字符：
		`SELECT SUBSTRING("RUNOOB", 2, 3) AS ExtractString;`
		返回：
		`-- UNO`

函数名	描述	示例

SUBSTRING_INDEX(s, delimiter, number)	返回从字符串s的第number个出现的分隔符delimiter之后的子串。
	如果number是正数，返回第number个字符左边的字符串。
	如果number是负数，返回第（number的绝对值（从右边数））个字符右边的字符串。
		• `SELECT SUBSTRING_INDEX('a*b','*',1);`
		返回：
		`-- a`
		• `SUBSTRING_INDEX('a*b','*',-1);`
		返回：
		`-- b`
		• `SUBSTRING_INDEX(SUBSTRING_INDEX('a*b*c*d*e','*',3),'*',-1);`
		返回：
		`-- c`

TRIM(s)	去掉字符串s开始和结尾处的空格	去掉字符串RUNOOB的首尾空格： `SELECT TRIM('RUNOOB') AS TrimmedString;`

UCASE(s)	将字符串转换为大写	将字符串runoob转换为大写：
		`SELECT UCASE("runoob");`
		返回：
		`-- RUNOOB`

UPPER(s)	将字符串转换为大写	将字符串runoob转换为大写：
		`SELECT UPPER("runoob");`
		返回：
		`-- RUNOOB`

不支持函数

与MySQL5.7相比，PolarDB-X暂不支持如下字符串函数：

函数名	描述
LOAD_FILE()	加载文件
MATCH	全文检索
SOUNDS LIKE	同音字符串比较

8.6.4. 转换函数

本文介绍了PolarDB-X支持的转换函数。

PolarDB-X支持如下转换函数：

函数名	描述	示例

函数名	描述	示例

BINARY	将字符串s转换为二进制字符串	`SELECT BINARY "RUNOOB";`
		输出：
		`-> RUNOOB`

CAST(x AS type)	转换数据类型	字符串日期转换为日期：
		`SELECT CAST("2017-08-29" AS DATE);`
		输出：
		`-> 2017-08-29`

CONVERT(s USING cs)	函数将字符串s的字符集变成cs	• `SELECT CHARSET('ABC')`
		输出：
		`->utf-8`
		• `SELECT CHARSET(CONVERT('ABC' USING gbk))`
		输出：
		`->gbk`

8.6.5. 聚合函数

本文介绍了PolarDB-X支持及不支持的聚合函数。

支持函数

PolarDB-X支持如下聚合函数：

函数名	描述
AVG(expression)	返回一个表达式的平均值，expression是一个字段。
COUNT(expression)	返回查询的记录总数，expression参数是一个字段或者*号。
COUNT(DISTINCT expression)	返回去重后查询的记录总数。
MAX(expression)	返回字段expression中的最大值。
MIN(expression)	返回字段expression中的最小值。
SUM(expression)	返回字段expression加起来的值。
BIT_OR(expression)	以或方式聚合expression。
BIT_XOR(expression)	以异或方式聚合expression。
BIT_AND(expression)	以与方式聚合expression。
GROUP_CONCAT(expression))	拼凑expression。

不支持函数

与MySQL5.7相比，PolarDB-X暂不支持如下聚合函数：

函数名	描述
STD()，STDDEV()	返回标准差。
STDDEV_POP()	计算总体标准偏离，并返回总体变量的平方根。
STDDEV_SAMP()	计算累积样本标准偏离，并返回样本变量的平方根。

8.6.6. 数学函数

本文介绍了PolarDB-X支持的数学函数。

PolarDB-X支持如下数学函数：

函数名	描述	示例

ABS(x)	返回x的绝对值。	返回-1的绝对值：
		`SELECT ABS(-1);`
		返回：
		`--1`

ACOS(x)	求x的反余弦值（参数是弧度）。	`SELECT ACOS(0.25);`
ASIN(x)	求x的反正弦值（参数是弧度）。	`SELECT ASIN(0.25);`
ATAN(x)	求x的反正切值（参数是弧度）。	`SELECT ATAN(2.5);`
ATAN2(n, m)	求n/m的反正切值（参数是弧度）。	`SELECT ATAN2(-0.8, 2);`
AVG(expression)	返回一个表达式的平均值，expression是一个字段。	返回Products表中Price字段的平均值： `SELECT AVG(Price) AS AveragePrice FROM Products;`

CEIL(x)	返回大于或等于x的最小整数 。	`SELECT CEIL(1.5);`
		返回：
		`--2`

CEILING(x)	返回大于或等于x的最小整数。	`SELECT CEILING(1.5);`
		返回：
		`--2`

COS(x)	求余弦值（参数是弧度）。	`SELECT COS(2);`
COT(x)	求余切值（参数是弧度）。	`SELECT COT(6);`

函数名	描述	示例
COUNT(expression)	返回查询的记录总数，expression参数是一个字段或者*号。	返回Products表中productID字段共有多少条记录： `SELECT COUNT(ProductID) AS NumberOfProducts FROM Products;`

DEGREES(x)	将弧度转换为角度。	`SELECT DEGREES(3.1415926535898);`
		返回：
		`-- 180`

n DIV m	整除，n为被除数，m为除数。	计算10除以5：
		`SELECT 10 DIV 5;`
		返回：
		`-- 2`

EXP(x)	返回e的x次方。	计算e的三次方：
		`SELECT EXP(3);`
		返回：
		`-- 20.085536923188`

FLOOR(x)	返回小于或等于x的最大整数。	小于或等于1.5的整数：
		`SELECT FLOOR(1.5);`
		返回：
		`-- 1`

GREATEST(expr1, expr2, expr3, ...)	返回列表中的最大值。	返回以下数字列表中的最大值：
		`SELECT GREATEST(3, 12, 34, 8, 25);`
		返回：
		`-- 34`
		返回以下字符串列表中的最大值：
		`SELECT GREATEST("Google", "Runoob", "Apple");`
		返回：
		`-- Runoob`

函数名	描述	示例

LEAST(expr1, expr2, expr3, ...)	返回列表中的最小值。	返回以下数字列表中的最小值：
		`SELECT LEAST(3, 12, 34, 8, 25);`
		返回：
		`-- 3`
		返回以下字符串列表中的最小值：
		`SELECT LEAST("Google", "Runoob", "Apple");`
		返回：
		`-- Apple`

LN	返回数字的自然对数，以e为底。	返回2的自然对数：
		`SELECT LN(2);`
		返回：
		`-- 0.6931471805599453`

LOG(x)或LOG(base, x)	返回自然对数（以e为底的对数），如果带有base参数，则base为指定带底数。	• `SELECT LOG(20.085536923188)`
		返回：
		`-- 3`
		• `SELECT LOG(2, 4);`
		返回：
		`-- 2`

LOG10(x)	返回以10为底的对数。	`SELECT LOG10(100) ;`
		返回：
		`-- 2`

LOG2(x)	返回以2为底的对数。	返回以2为底6的对数：
		`SELECT LOG2(6);`
		返回：
		`-- 2.584962500721156`

MAX(expression)	返回字段expression中的最大值。	返回数据表Products中字段Price的最大值： `SELECT MAX(Price) AS LargestPrice FROM Products;`
MIN(expression)	返回字段expression中的最小值。	返回数据表Products中字段Price的最小值： `SELECT MIN(Price) AS MinPrice FROM Products;`

函数名	描述	示例

MOD(x,y)	返回x除以y以后的余数。	5除以2的余数：
		`SELECT MOD(5,2);`
		返回：
		`-- 1`

PI()	返回圆周率（3.141593）。	`SELECT PI();`
		返回：
		`--3.141593`

POW(x,y)	返回x的y次方。	2的3次方：
		`SELECT POWER(2,3);`
		返回：
		`-- 8`

POWER(x,y)	返回x的y次方。	2的3次方：
		`SELECT POWER(2,3);`
		返回：
		`-- 8`

RADIANS(x)	将角度转换为弧度。	180度转换为弧度：
		`SELECT RADIANS(180);`
		返回：
		`-- 3.1415926535898`

RAND()	返回0到1的随机数。	`SELECT RAND();`
		返回：
		`--0.93099315644334`

ROUND(x)	返回离x最近的整数。	`SELECT ROUND(1.23456;`
		返回：
		`--1`

SIGN(x)	返回x的符号，x是负数、0、正数分别返回-1、0和1。	`SELECT SIGN(-10);`
		返回：
		`-- (-1)`

SIN(x)	求正弦值（参数是弧度）。	`SELECT SIN(RADIANS(30));`
		返回：
		`-- 0.5`

函数名	描述	示例

SQRT(x)	返回x的平方根。	25的平方根：
		`SELECT SQRT(25);`
		返回：
		`-- 5`

SUM(expression)	返回指定字段的总和。	计算OrderDetails表中字段Quantity的总和： `SELECT SUM(Quantity) AS TotalItemsOrdered FROM OrderDetails;`

TAN(x)	求正切值（参数是弧度）。	`SELECT TAN(1.75);`
		返回：
		`-- -5.52037992250933`

TRUNCATE(x,y)	返回数值x保留到小数点后y位的值（与ROUND最大的区别是不会进行四舍五入）。	`SELECT TRUNCATE(1.23456,3);`
		返回：
		`-- 1.234`

8.6.7. 比较函数

本文介绍了PolarDB-X支持的比较函数。

PolarDB-X支持如下比较函数：

函数名	描述
COALESCE()	返回参数中的第一个非空表达式（从左向右）。
GREATEST()	返回列表中的最大值。
IN()	检查特定的值或者字段在IN罗列的表达式中。
ISNULL()	判断表达式是否为NULL。
LEAST()	返回列表中的最小值。
STRCMP()	比较字符串s1和s2，如果s1与s2相等返回0，如果s1>s2返回1，如果s1<s2 返回-1。

8.6.8. 位函数

本文介绍了PolarDB-X支持的位函数。

位函数分为两类，一是标量函数，二是聚合函数。

PolarDB-X支持的标量位函数

函数名	描述
|	按位或
^	按位异或

函数名	描述
&	按位与
BIT_COUNT()	统计二进制数中1的个数

PolarDB-X支持的聚合位函数

函数名	描述
BIT_OR()	按位或
BIT_XOR()	按位异或
BIT_AND()	按位与

示例

BIT_COUNT()返回参数对应的二进制数中1的个数；若参数为NULL，则返回NULL。

```
mysql> SELECT BIT_COUNT(29), BIT_COUNT(b'101010');
+--------------+----------------------+
| BIT_COUNT(29) | BIT_COUNT(b'101010') |
+--------------+----------------------+
|            4 |                    3 |
+--------------+----------------------+
1 row in set (0.00 sec)
mysql> SELECT BIT_COUNT(NULL);
+-----------------+
| BIT_COUNT(NULL) |
+-----------------+
|            NULL |
+-----------------+
1 row in set (0.00 sec)
mysql> SELECT 3 & 2;
+-------+
| 3 & 2 |
+-------+
|     2 |
+-------+
1 row in set (0.01 sec)
```

8.6.9. 流程控制函数

本文介绍了PolarDB-X支持的流程控制函数。

PolarDB-X支持如下流程控制函数：

函数名	描述
CASE expression WHEN condition1 THEN result1 WHEN condition2 THEN result2 ... WHEN conditionN THEN resultN ELSE resultEND	CASE表示函数开始，END表示函数结束。如果condition1成立，则返回result1，如果condition2成立，则返回result2，当全部不成立则返回result，而当有一个成立之后，后面的就不执行了。
IF(expr,v1,v2)	如果表达式expr成立，返回结果v1；否则，返回结果v2。
IFNULL(v1,v2)	如果v1的值不为NULL，则返回v1，否则返回v2。
NULLIF(expr1, expr)	比较两个字符串，如果字符串expr与expr2相等，返回NULL，否则返回expr1。

8.6.10. 信息函数

信息函数用于返回动态的数据库信息，本文介绍了PolarDB-X支持及不支持的信息函数。

支持函数

PolarDB-X支持如下信息函数：

函数名	描述
CONNECTION_ID()	返回唯一的连接ID
CURRENT_USER(), CURRENT_USER	返回当前用户
DATABASE()	返回当前数据库
LAST_INSERT_ID()	返回最近生成的AUTO_INCREMENT值
SCHEMA()	和DATABASE()同义
SESSION_USER()	和 USER()同义
SYSTEM_USER()	和 USER()同义
USER()	返回当前用户
VERSION()	返回当前版本号
BENCHMARK()	重复执行一个表达式
CHARSET()	返回当前字符集
COLLATION()	返回当前Collation
FOUND_ROWS()	返回上次查询结果集的记录数
ROW_COUNT()	返回上一条SQL语句，对表数据进行修改操作后影响的记录数

不支持函数

与 MySQL5.7 相比，PolarDB-X暂不支持如下信息函数：

函数	描述
COERCIBILITY()	返回字符串参数的整序可压缩性值

8.6.11. 加密和压缩函数

本文主要介绍PolarDB-X支持的加密和压缩函数。

支持的加密和压缩函数

PolarDB-X目前支持mysql 5.7的大部分未过时的加密和压缩函数，具体信息见下表：

函数名	描述
AES_DECRYPT	使用AES算法解密
AES_ENCRYPT	使用AES算法加密
RANDOM_BYTES	返回随机字节向量

函数名	描述
MD5	计算MD5 128位校验和
SHA1, SHA	计算SHA-1 160位校验和
SHA2	计算SHA2校验和

对于加密和压缩函数，如果需要存储字节类型的返回结果，推荐使用VARBINARY或BLOB字段，避免尾部空白字符的移除或字符集转换问题（如CHAR/VARCHAR/TEXT类型）。

AES_DECRYPT(crypt_str, key_str [, init_vector])

根据输入的密文crypt_str、密钥key_str以及初始向量init_vector（可选参数），返回解密后的明文结果。具体解密算法与使用方法可参考AES_ENCRYPT函数。

AES_ENCRYPT(str, key_str [, init_vector])

根据输入的明文str、密钥key_str以及初始向量init_vector（可选参数），返回加密后的密文结果。在加密时具体使用的AES算法由系统变量block_encryption_mode决定，该变量取值的格式为 `aes-keylen-mode` ，其中keylen为密钥的位长度（合法取值为128/192/256），mode为加密模式，PolarDB-X支持以下六种加密模式：

加密模式	是否需要初始向量
ECB	否
CBC	是
CFB1	是
CFB8	是
CFB128	是
OFB	是

对于需要初始向量的加密模式，初始向量必须大于等于16字节（超出16字节部分将自动截断）；对于不需要初始向量的加密模式，初始向量参数将被自动忽略。

示例：

```
mysql> SET block_encryption_mode = 'aes-128-ofb';
mysql> SET @iv = RANDOM_BYTES(16);
mysql> SET @key = SHA2('secret key', 224);
mysql> set @crypto = AES_ENCRYPT('polardb-x', @key, @iv);
mysql> select @crypto;
+----------------------------+
| @crypto                    |
+----------------------------+
|  ß÷s,(ýýÂåîA}ýO           |
+----------------------------+
mysql> SELECT AES_DECRYPT(@crypto, @key, @iv);
+---------------------------------+
| AES_DECRYPT(@crypto, @key, @iv) |
+---------------------------------+
| polardb-x                       |
+---------------------------------+
```

RANDOM_BYTES(len)

返回len字节长度的随机二进制字符串，长度len的合法取值为1~1024。

示例：

```
mysql> select HEX(RANDOM_BYTES(16));
+----------------------------------+
| HEX(RANDOM_BYTES(16))            |
+----------------------------------+
| C83CF8A2499F407E15F34F6E32948CEA |
+----------------------------------+
```

MD5(str)

计算MD5 128位检验和。

示例：

```
mysql> select MD5('polardb-x');
+----------------------------------+
| MD5('polardb-x')                 |
+----------------------------------+
| fa4900656bcd39dc90024e733fa4531f |
+----------------------------------+
```

SHA1(str), SHA(str)

计算SHA-1 160位校验和。该函数安全性强于MD5。

示例：

```
mysql> select SHA1('polardb-x');
+------------------------------------------+
| SHA1('polardb-x')                        |
+------------------------------------------+
| a2e83af051f032b500f13c369976298208d821d1 |
+------------------------------------------+
```

SHA2(str, hash_length)

计算SHA-2族散列算法结果，参数hash_length决定了检验和结果的位数，合法取值包括：224、256、384、512和0（等价于256）。该函数安全性强于MD5和SHA1。

示例：

```
mysql> select SHA2('polardb-x', 384);
+----------------------------------------------------------------------------------------------------------------+
| SHA2('polardb-x', 384)                                                                                         |
+----------------------------------------------------------------------------------------------------------------+
| 20222037666be5234d9af3c391f9c3a1a3e39b910f3f8081c32d972acca890c818d6c70025ff6c6d4b648bd91d66a3fe |
+----------------------------------------------------------------------------------------------------------------+
```

8.6.12. 窗口函数

传统的Group By函数会按照分组后的查询结果进行聚合计算，且每个分组只输出一条数据。但与传统的Group By函数不同，窗口函数（也称OLAP函数）可以为每个分组返回多个值，且不会影响记录的数量。本文介绍如何使用窗口函数。

使用限制

- 窗口函数仅支持用于SELECT语句中。
- 窗口函数禁止与单独的聚合函数混合使用。

  例如，在如下语句中，SUM为聚合函数，且未与OVER关键字组合，因此您无法使用如下语句进行查询：

  ```
  SELECT SUM(NAME),COUNT() OVER(...) FROM SOME_TABLE
  ```

  若需实现如上查询，您可以使用如下语句代替：

```
SELECT SUM(NAME),WIN1 FROM (SELECT NAME,COUNT() OVER(...) AS WIN1 FROM SOME_TABLE) alias
```

语法

```
function OVER ([[partition by column_some1] [order by column_some2] [RANGE|ROWS BETWEEN start AND end]])
```

参数	说明

`function`	该部分指定了窗口函数中支持的函数，取值范围如下：
	• 可以在窗口函数中结合OVER关键字使用如下聚合函数：
	  ◦ `SUM()`
	  ◦ `COUNT()`
	  ◦ `AVG()`
	  ◦ `MAX()`
	  ◦ `MIN()`
	• 专用窗口函数如下：
	  ◦ `ROW_NUMBER()`
	  ◦ `RANK()`
	  ◦ `DENSE_RANK()`
	  ◦ `PERCENT_RANK()`
	  ◦ `CUME_DIST()`
	  ◦ `FIRST_VALUE()`
	  ◦ `LAST_VALUE()`
	  ◦ `LAG()`
	  ◦ `LEAD()`
	  ◦ `NTH_VALUE()`
	说明
	• 当使用专用窗口函数 `RANK()` 或 `DENSE_RANK()` 时，窗口函数中的 `order by` 部分不可省略。更多专用窗口函数的介绍，请参见Window Function Descriptions。
	• 支持如下专用窗口函数：
	  ◦ `PERCENT_RANK()`
	  ◦ `CUME_DIST()`
	  ◦ `FIRST_VALUE()`
	  ◦ `LAST_VALUE()`
	  ◦ `LAG()`
	  ◦ `LEAD()`
	  ◦ `NTH_VALUE()`

`[partition by column_some1]`	该部分指定了窗口函数的分区规范，用于将输入行分散到不同的分区中，过程和GROUP BY子句的分散过程相似。 **说明** `partition by` 部分不支持引用复杂表达式，如您可以引用 `column_some1` ，但不可以引用 `column_some1 + 1` 。

参数	说明
`[order by column_some2]`	该部分指定了窗口函数的排序规范，用于确定输入数据行在窗口函数中执行的顺序。 **说明** `order by` 部分不支持引用复杂表达式，如您可以引用 `column_some2` ，但不可以引用 `column_some2 + 1` 。

`[RANGE|ROWS BETWEEN start AND end]`	该部分指定了窗口函数的窗口区间，支持按照计算列值的范围（即RANGE）或计算列的行数（即ROWS）等两种模式来定义区间。
	您可以使用 `BETWEEN start AND end` 指定边界的可取值，其中：
	• `start` 取值范围如下：
	  ◦ `CURRENT ROW` ：当前行
	  ◦ `N PRECEDING` ：前N行
	  ◦ `UNBOUNDED PRECEDING` ：直到第1行
	• `end` 取值范围如下：
	  ◦ `CURRENT ROW` ：当前行
	  ◦ `N FOLLOWING` ：后N行
	  ◦ `UNBOUNDED FOLLOWING` ：直到最后1行

使用示例

假设已有如下原始数据：

```
| year | country | product    | profit |
|------|---------|------------|--------|
| 2001 | Finland | Phone      |     10 |
| 2000 | Finland | Computer   |   1500 |
| 2001 | USA     | Calculator |     50 |
| 2001 | USA     | Computer   |   1500 |
| 2000 | India   | Calculator |     75 |
| 2000 | India   | Calculator |     75 |
| 2001 | India   | Calculator |     79 |
```

- 您可以使用如下聚合函数来统计每个国家的总利润：

```
select
    country,
    sum(profit) over (partition by country) sum_profit
from test_window;
```

返回结果如下：

```
| country | sum_profit |
|---------|------------|
| India   |        229 |
| India   |        229 |
| India   |        229 |
| USA     |       1550 |
| USA     |       1550 |
| Finland |       1510 |
| Finland |       1510 |
```

- 您可以使用如下专用窗口函数将数据按照国家分组，并将国家内的产品按利润由小到大排名：

```
select
    'year',
    country,
    product,
    profit,
    rank() over (partition by country order by profit) as rank
from test_window;
```

返回结果如下：

```
| year | country | product    | profit | rank |
|------|---------|------------|--------|------|
| 2001 | Finland | Phone      |     10 |    1 |
| 2000 | Finland | Computer   |   1500 |    2 |
| 2001 | USA     | Calculator |     50 |    1 |
| 2001 | USA     | Computer   |   1500 |    2 |
| 2000 | India   | Calculator |     75 |    1 |
| 2000 | India   | Calculator |     75 |    1 |
| 2001 | India   | Calculator |     79 |    3 |
```

- 您可以使用如下带有ROWS命令的语句，查询根据当前窗口的每行数据计算利润部分的总和：

```
select
    'year',
    country,
    profit,
    sum(profit) over (partition by country order by 'year' ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT RO
W) as sum_win
from test_window;
```

返回结果如下：

```
+------+---------+--------+-------------+
| year | country | profit |   sum_win   |
+------+---------+--------+-------------+
| 2001 | USA     |     50 |          50 |
| 2001 | USA     |   1500 |        1550 |
| 2000 | India   |     75 |          75 |
| 2000 | India   |     75 |         150 |
| 2001 | India   |     79 |         229 |
| 2000 | Finland |   1500 |        1500 |
| 2001 | Finland |     10 |        1510 |
```

8.6.13. JSON函数

本文介绍了PolarDB-X支持的JSON函数。

概览

PolarDB-X目前支持mysql 5.7的所有json函数，可分为以下五大类：

类别	函数名
JSON创建	JSON_ARRAY
	JSON_OBJECT
	JSON_QUOTE
	JSON_CONTAINS
	JSON_CONTAINS_PATH

类别	函数名
JSON搜索	JSON_EXTRACT
	->
	->>
	JSON_KEYS
	JSON_SEARCH
JSON修改	JSON_APPEND
	JSON_ARRAY_APPEND
	JSON_ARRAY_INSERT
	JSON_INSERT
	JSON_MERGE
	JSON_MERGE_PATCH
	JSON_MERGE_PRESERVE
	JSON_REMOVE
	JSON_REPLACE
	JSON_SET
	JSON_UNQUOTE
JSON属性	JSON_DEPTH
	JSON_LENGTH
	JSON_TYPE
	JSON_VALID
JSON工具	JSON_PRETTY
	JSON_STORAGE_SIZE

JSON_ARRAY([val [, val] ...])

- 描述：返回包含所有参数值的JSON数组（数组可为空）。
- 示例：

```
mysql> SELECT JSON_ARRAY(123, "polardb-x", NULL, TRUE);
+-----------------------------------------+
| JSON_ARRAY(123, 'polardb-x', NULL, true) |
+-----------------------------------------+
| [123,"polardb-x",null,true]              |
+-----------------------------------------+
```

JSON_OBJECT([key, val [, key, val] ...])

- 描述：根据参数中键值对来创建一个JSON对象，如果key为NULL或参数个数不为2的倍数，将返回相应错误提示。
- 示例：

```
mysql> SELECT JSON_OBJECT('id', 123, 'name', 'polardb-x');
+---------------------------------------------+
| JSON_OBJECT('id', 123, 'name', 'polardb-x') |
+---------------------------------------------+
| {"name":"polardb-x","id":123}               |
+---------------------------------------------+
```

JSON_QUOTE(string)

- 描述：将JSON值用双引号括起来，并对内部特殊处理字符进行转义，返回utf8mb4编码的字符串结果。若参数为NULL，返回值为NULL。该函数通常用于生成一个合法的JSON字符串字面量。
- 示例：

```
mysql> SELECT JSON_QUOTE(null), JSON_QUOTE('"abc"');
+------------------+----------------------+
| JSON_QUOTE(NULL) | JSON_QUOTE('"abc"') |
+------------------+----------------------+
| NULL             | ""abc""              |
+------------------+----------------------+
```

JSON_CONTAINS(target, candidate [, path])

- 描述：判断给定的candidate JSON文档是否包含了target JSON文档；如果给定了path参数，则是判断candidate能否找到包含target的路径path，结果返回1或0。如果只关心candidate是否存在路径path，可以使用JSON_CONTAINS_PATH函数。
- 示例：

```
mysql> SET @json_doc = '{"a": 123, "b": null, "c": {"d": 456}}';
mysql> SELECT JSON_CONTAINS(@json_doc, '123', '$.a');
+----------------------------------------+
| JSON_CONTAINS(@json_doc, '123', '$.a') |
+----------------------------------------+
|                                      1 |
+----------------------------------------+
mysql> SELECT JSON_CONTAINS(@json_doc, 'null', '$.b');
+-----------------------------------------+
| JSON_CONTAINS(@json_doc, 'null', '$.b') |
+-----------------------------------------+
|                                       1 |
+-----------------------------------------+
mysql> SELECT JSON_CONTAINS(@json_doc, '123', '$.b');
+----------------------------------------+
| JSON_CONTAINS(@json_doc, '123', '$.b') |
+----------------------------------------+
|                                      0 |
+----------------------------------------+
mysql> SELECT JSON_CONTAINS(@json_doc, '{"d": 456}', '$.c');
+-----------------------------------------------+
| JSON_CONTAINS(@json_doc, '{"d": 456}', '$.c') |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ...)

- 判断给定的json_doc JSON文档是否存在一条或多条路径path，结果返回1或0。参数one_or_all的取值规定如下：
  - 'one'：只要json_doc存在参数中的至少一条路径，就返回1，否则返回0；
  - 'all'：只有json_doc包含参数中的所有路径时，才返回1，否则返回0。
- 示例：

```
mysql> SET @json_doc = '{"a": 123, "b": null, "c": {"d": 456}}';
mysql> SELECT JSON_CONTAINS_PATH(@json_doc, 'one', '$.a', '$.e');
+-------------------------------------------------+
| JSON_CONTAINS_PATH(@json_doc, 'one', '$.a', '$.e') |
+-------------------------------------------------+
|                                               1 |
+-------------------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@json_doc, 'all', '$.a', '$.e');
+-------------------------------------------------+
| JSON_CONTAINS_PATH(@json_doc, 'all', '$.a', '$.e') |
+-------------------------------------------------+
|                                               0 |
+-------------------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@json_doc, 'one', '$.c.d');
+-----------------------------------------------+
| JSON_CONTAINS_PATH(@json_doc, 'one', '$.c.d') |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

JSON_EXTRACT(json_doc, path[, path] ...)

- 描述：从JSON文档中解析出路径path对应的部分。
- 示例：

```
mysql> SELECT JSON_EXTRACT('[123, 456, [789, 1000]]', '$[1]');
+-------------------------------------------------+
| JSON_EXTRACT('[123, 456, [789, 1000]]', '$[1]') |
+-------------------------------------------------+
| 456                                             |
+-------------------------------------------------+
mysql> SELECT JSON_EXTRACT('[123, 456, [789, 1000]]', '$[0]', '$[1]');
+-----------------------------------------------------------+
| JSON_EXTRACT('[123, 456, [789, 1000]]', '$[0]', '$[1]') |
+-----------------------------------------------------------+
| [123,456]                                                 |
+-----------------------------------------------------------+
mysql> SELECT JSON_EXTRACT('[123, 456, [789, 1000]]', '$[0]', '$[2]');
+-----------------------------------------------------------+
| JSON_EXTRACT('[123, 456, [789, 1000]]', '$[0]', '$[2]') |
+-----------------------------------------------------------+
| [123,[789,1000]]                                          |
+-----------------------------------------------------------+
```

column->path

- 描述：->操作符提供了与JSON_EXTRACT函数相同的功能，返回path对应的列值。
- 示例：

```
mysql> SELECT JSON_OBJECT('id', 123, 'name', 'polardb-x')->"$.name" as NAME;
+--------------+
| NAME         |
+--------------+
| "polardb-x" |
+--------------+
```

column->>path

- 描述：->>操作符在->操作符的基础上，对返回结果去除了双引号并反转义（参考JSON_UNQUOTE函数）。该操作符与以下两个函数组合等价：
  - JSON_UNQUOTE(JSON_EXTRACT(column, path))
  - JSON_UNQUOTE(column -> path)

- 示例：

```
mysql> SELECT JSON_OBJECT('id', 123, 'name', 'polardb-x')->"$.name" as NAME;
+--------------+
| NAME         |
+--------------+
| "polardb-x" |
+--------------+
```

JSON_KEYS(json_doc[, path])

- 以JSON数组的形式返回json_doc对象最外层的key值列表。
- 示例：

```
mysql> SELECT JSON_KEYS('{"a": 123, "b": {"c": 456}}');
+-------------------------------------------+
| JSON_KEYS('{"a": 123, "b": {"c": 456}}') |
+-------------------------------------------+
| ["a","b"]                                 |
+-------------------------------------------+
```

JSON_SEARCH(json_doc, one_or_all, search_str [, escape_char[, path] ...])

- 返回给定字符串search_str在json_doc中的路径，未找到则返回NULL。search_str中支持包含如LIKE中使用的%和_通配符作为模糊匹配。参数escape_char指定了转义字符，而参数one_or_all的取值规定如下：
  - 'one'：返回第一个匹配的路径值（哪一条路径第一个匹配到是不确定的）；
  - 'all'：以数组形式返回所有路径值（数组元素顺序是不确定的）。

  > **说明** PolarDB-X暂不支持**路径通配符

- 示例：

```
mysql> SET @json_doc = '["abc", [{"k1": 123}, "def"], {"k2": "abc"}, {"k3": null}]';
mysql> SELECT JSON_SEARCH(@json_doc, 'one', 'abc');
+-----------------------------------+
| JSON_SEARCH(@json_doc, 'one', 'abc') |
+-----------------------------------+
| "$[0]"                            |
+-----------------------------------+
mysql> SELECT JSON_SEARCH(@json_doc, 'all', 'abc');
+-----------------------------------+
| JSON_SEARCH(@json_doc, 'all', 'abc') |
+-----------------------------------+
| ["$[0]","$[2].k2"]                |
+-----------------------------------+
mysql> SELECT JSON_SEARCH(@json_doc, 'all', 'xyz');
+-----------------------------------+
| JSON_SEARCH(@json_doc, 'all', 'xyz') |
+-----------------------------------+
| NULL                              |
+-----------------------------------+
mysql> SELECT JSON_SEARCH(@json_doc, 'all', 'def', NULL, '$[*]');
+-------------------------------------------------+
| JSON_SEARCH(@json_doc, 'all', 'def', NULL, '$[*]') |
+-------------------------------------------------+
| "$[1][1]"                                       |
+-------------------------------------------------+
mysql> SELECT JSON_SEARCH(@json_doc, 'all', '%a%');
+-----------------------------------+
| JSON_SEARCH(@json_doc, 'all', '%a%') |
+-----------------------------------+
| ["$[0]","$[2].k2"]                |
+-----------------------------------+
```

JSON_APPEND(json_doc, path, val [, path, val] ...)

即JSON_ARRAY_APPEND函数。

> **说明** 该函数在MySQL5.7中已过时，在MySQL8.0中被移除。

JSON_ARRAY_APPEND(json_doc, path, val [, path, val] ...)

- 描述：将val追加到json_doc指定JSON数组末尾，并返回修改后的json_doc。当有多对(path, val)参数时，该函数会从左到右依次计算每一对参数，并将追加完前一对参数后的结果作为下一次计算的输入。
- 示例：

```
mysql> SET @json_doc = '{"a": 1, "b": [2, 3], "c": 4}';
mysql> SELECT JSON_ARRAY_APPEND(@json_doc, '$.b', 'x');
+-----------------------------------------+
| JSON_ARRAY_APPEND(@json_doc, '$.b', 'x') |
+-----------------------------------------+
| {"a":1,"b":[2,3,"x"],"c":4}             |
+-----------------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@json_doc, '$.c', 'y');
+-----------------------------------------+
| JSON_ARRAY_APPEND(@json_doc, '$.c', 'y') |
+-----------------------------------------+
| {"a":1,"b":[2,3],"c":[4,"y"]}           |
+-----------------------------------------+
```

JSON_ARRAY_INSERT(json_doc, path, val [, path, val] ...)

- 描述：将val插入到json_doc中JSON数组的指定位置，并返回修改后的json_doc。如果val插入位置超出了数组大小，则会追加为最后一个元素。当有多对(path, val)参数时，该函数会从左到右依次计算每一对参数，并将插入完前一对参数后的结果作

为下一次计算的输入。

- 示例：

```
mysql> SET @json_doc = '["a", {"b": [1, 2]}, [3, 4]]';
mysql> SELECT JSON_ARRAY_INSERT(@json_doc, '$[1]', 'x');
+-----------------------------------------+
| JSON_ARRAY_INSERT(@json_doc, '$[1]', 'x') |
+-----------------------------------------+
| ["a","x",{"b":[1,2]},[3,4]]               |
+-----------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@json_doc, '$[10]', 'x');
+-----------------------------------------+
| JSON_ARRAY_INSERT(@json_doc, '$[10]', 'x') |
+-----------------------------------------+
| ["a",{"b":[1,2]},[3,4],"x"]               |
+-----------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@json_doc, '$[1].b[1]', 'x');
+-----------------------------------------------+
| JSON_ARRAY_INSERT(@json_doc, '$[1].b[1]', 'x') |
+-----------------------------------------------+
| ["a",{"b":[1,"x",2]},[3,4]]                   |
+-----------------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@json_doc, '$[0]', 'x', '$[3][1]', 'y');
+-----------------------------------------------------------+
| JSON_ARRAY_INSERT(@json_doc, '$[0]', 'x', '$[3][1]', 'y') |
+-----------------------------------------------------------+
| ["x","a",{"b":[1,2]},[3,"y",4]]                           |
+-----------------------------------------------------------+
```

JSON_INSERT(json_doc, path, val [, path, val] ...)

- 描述：将val插入到json_doc的指定位置。当有多对(path, val)参数时，该函数会从左到右依次计算每一对参数，并将插入完前一对参数后的结果作为下一次计算的输入。插入val时的行为，将区分以下两种情况：
  - 如果指定path处本身已存在值，则会忽略当前这轮处理的(path, val)对，不会覆盖原本已存在的值；
  - 如果指定path处不存在值，则分为以下三种情况：
    - 如果指定path处是对象内不存在的成员，则直接在对象内添加对应的键值对；
    - 如果指定path处是超出数组下标的元素，则会将val添加到该数组末尾（标量值会自动包装为只含一个元素的数组）；
    - 其他情况，则会忽略当前这轮处理的(path, val)对，不采取任何操作。

  请参考JSON_REPLACE函数、JSON_SET函数来比较三者用法的不同。

- 示例：

```
mysql> SET @json_doc = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_INSERT(@json_doc, '$.a', 10, '$.c', '[true, false]');
+-----------------------------------------------------------+
| JSON_INSERT(@json_doc, '$.a', 10, '$.c', '[true, false]') |
+-----------------------------------------------------------+
| {"a":1,"b":[2,3],"c":"[true, false]"}                     |
+-----------------------------------------------------------+
```

JSON_MERGE(json_doc, json_doc [, json_doc] ...)

即JSON_MERGE_PRESERVE函数（注：该函数在MySQL5.7中已过时，在MySQL8.0.3中被移除）。

JSON_MERGE_PATCH(json_doc, json_doc [, json_doc] ...)

- 合并多个JSON文档，且会对相同的key进行去重。该函数将从左到右依次合并两个json_doc（下称为doc1与doc2），合并规则如下：
  i. 如果doc1不是对象类型，则合并结果即为doc2；
  ii. 如果doc2不是对象类型，则合并结果也为doc2；

iii. 如果doc1与doc2都是对象类型，则合并结果是包含了以下成员的对象：

- key只存在于doc1、而不存在于doc2的所有doc1成员；
- key只存在于doc2、而不存在于doc1，且值不是null字面量的所有doc2成员；
- 对于key既存在于doc1、又存在于doc2且在doc2中值不为null字面量的成员，会将key对应的两个value再按上述规则进行递归合并。

根据上述规则可知，如果有重复key则会保留doc2中对应的值。

- 示例：

```
mysql> SELECT JSON_MERGE_PATCH('{"name": "polardb-x"}', '{"id": 123}');
+-----------------------------------------------------------+
| JSON_MERGE_PATCH('{"name": "polardb-x"}', '{"id": 123}') |
+-----------------------------------------------------------+
| {"name":"polardb-x","id":123}                             |
+-----------------------------------------------------------+
mysql> SELECT JSON_MERGE_PATCH('{"a":1, "b":2}', '{"b":null}');
+-------------------------------------------------+
| JSON_MERGE_PATCH('{"a":1, "b":2}', '{"b":null}') |
+-------------------------------------------------+
| {"a":1}                                         |
+-------------------------------------------------+
mysql> SELECT JSON_MERGE_PATCH('{ "a": 1, "b":2 }', '{ "a": 3, "c":4 }',
    -> '{ "a": 5, "d":6 }');
+-----------------------------------------------------------------------------------+
| JSON_MERGE_PATCH('{ "a": 1, "b":2 }', '{ "a": 3, "c":4 }', '{ "a": 5, "d":6 }') |
+-----------------------------------------------------------------------------------+
| {"a":5,"b":2,"c":4,"d":6}                                                         |
+-----------------------------------------------------------------------------------+
```

JSON_MERGE_PRESERVE(json_doc, json_doc [, json_doc] ...)

- 描述：

  合并多个JSON文档，会保留所有的值而不会根据相同key去重。合并规则如下：

  - 两个数组的合并结果为一个数组；
  - 两个对象的合并结果为一个对象；
  - 标量类型值会自动包装为数组类型进行合并；
  - 一个对象和一个数组合并，则会把对象包装为含单个元素数组并进行合并。

- 示例：（可以对比与JSON_MERGE_PATCH函数的不同）

```
mysql> SELECT JSON_MERGE_PRESERVE('{"name": "polardb-x"}', '{"id": 123}');
+-------------------------------------------------------------+
| JSON_MERGE_PRESERVE('{"name": "polardb-x"}', '{"id": 123}') |
+-------------------------------------------------------------+
| {"name":"polardb-x","id":123}                               |
+-------------------------------------------------------------+
mysql> SELECT JSON_MERGE_PRESERVE('{"a":1, "b":2}', '{"b":null}');
+---------------------------------------------------+
| JSON_MERGE_PRESERVE('{"a":1, "b":2}', '{"b":null}') |
+---------------------------------------------------+
| {"a":1,"b":[2,null]}                              |
+---------------------------------------------------+
mysql> SELECT JSON_MERGE_PRESERVE('{ "a": 1, "b":2 }', '{ "a": 3, "c":4 }',
    ->  '{ "a": 5, "d":6 }');
+-------------------------------------------------------------------------------------+
| JSON_MERGE_PRESERVE('{ "a": 1, "b":2 }', '{ "a": 3, "c":4 }', '{ "a": 5, "d":6 }') |
+-------------------------------------------------------------------------------------+
| {"a":[1,3,5],"b":2,"c":4,"d":6}                                                     |
+-------------------------------------------------------------------------------------+
```

JSON_REMOVE(json_doc, path [, path] ...)

- 描述：删除json_doc下指定路径的元素。当有多个path参数时，该函数会从左到右依次计算路径并删除元素，并将前一次删除后的结果作为下一次计算的输入。若指定路径不存在，不会报错。
- 示例：

```
mysql> SET @json_doc = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_REMOVE(@json_doc, '$[1]');
+---------------------------------+
| JSON_REMOVE(@json_doc, '$[1]') |
+---------------------------------+
| ["a","d"]                       |
+---------------------------------+
```

JSON_REPLACE(json_doc, path, val [, path, val] ...)

- 描述：用val替换json_doc的指定位置的值，只会替换已存在的值。当有多对(path, val)参数时，该函数会从左到右依次计算每一对参数，并将计算完前一对参数后的结果作为下一次计算的输入。若指定路径不存在，不会报错。
- 示例：

```
mysql> SET @json_doc = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_REPLACE(@json_doc, '$.a', 10, '$.c', '[true, false]');
+-----------------------------------------------------------+
| JSON_REPLACE(@json_doc, '$.a', 10, '$.c', '[true, false]') |
+-----------------------------------------------------------+
| {"a":"10","b":[2,3]}                                      |
+-----------------------------------------------------------+
```

JSON_SET(json_doc, path, val [, path, val] ...)

- 描述：在json_doc的指定位置插入或更新值val。该函数会从左到右依次计算每一对参数，并将计算完前一对参数后的结果作为下一次计算的输入。将区分以下两种情况：
  - 如果指定path处本身已存在值，会用参数val的值覆盖原本已存在的值；
  - 如果指定path处不存在值，则分为以下三种情况：
    - 如果指定path处是对象内不存在的成员，则直接在对象内添加对应的键值对；
    - 如果指定path处是超出数组下标的元素，则会将val添加到该数组末尾（标量值会自动包装为只含一个元素的数组）；
    - 其他情况，则会忽略当前这轮处理的(path, val)对，不采取任何操作。
- 示例：

```
mysql> SET @json_doc = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_SET(@json_doc, '$.a', 10, '$.c', '[true, false]');
+-------------------------------------------------+
| JSON_SET(@j, '$.a', 10, '$.c', '[true, false]') |
+-------------------------------------------------+
| {"a":10,"b":[2,3],"c":[true,false]}             |
+-------------------------------------------------+
```

JSON_UNQUOTE(json_val)

- 描述：去除JSON值外围的双引号，对转义字符进行反转义，返回utf8mb4编码的字符串结果。
- 示例：

```
mysql> SELECT JSON_UNQUOTE('"abc"');
+------------------------+
| JSON_UNQUOTE('"abc"') |
+------------------------+
| abc                    |
+------------------------+
mysql> SELECT JSON_UNQUOTE('"a\\tbc"');
+---------------------------+
| JSON_UNQUOTE('"a\\tbc"') |
+---------------------------+
| a    bc                     |
+---------------------------+
mysql> SELECT JSON_UNQUOTE('"\\t\\u0032"');
+-------------------------------+
| JSON_UNQUOTE('"\\t\\u0032"') |
+-------------------------------+
|     2                          |
+-------------------------------+
```

JSON_DEPTH(json_doc)

- 描述：

  返回json_doc的最大深度，规则如下：

  - 空数组、空对象以及标量类型，深度均为1；
  - 只有单个元素的数组深度为1；
  - 对象中如果所有成员的value深度均为1，则自身的深度为2。

- 示例：

```
mysql> SELECT JSON_DEPTH('[10, {"a": 20}]');
+--------------------------------+
| JSON_DEPTH('[10, {"a": 20}]') |
+--------------------------------+
|                              3 |
+--------------------------------+
```

JSON_LENGTH(json_doc [, path])

- 描述：

  返回json_doc的长度，规则如下：

  - 标量类型的长度为1；
  - 数组类型的长度为数组元素个数；
  - 对象类型的长度为对象成员个数；
  - 结果不包含内部嵌套的数组或对象长度。

- 示例：

```
mysql> SELECT JSON_LENGTH('{"a": 1, "b": {"c": 30}}');
+-----------------------------------------+
| JSON_LENGTH('{"a": 1, "b": {"c": 30}}') |
+-----------------------------------------+
|                                       2 |
+-----------------------------------------+
```

JSON_TYPE(json_val)

- 描述：返回参数json_val的json类型。
- 示例：

```
mysql> SET @json_obj = '{"a": [10, true]}';
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a'));
+-----------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a')) |
+-----------------------------------------+
| ARRAY                                   |
+-----------------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a[0]'));
+--------------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a[0]')) |
+--------------------------------------------+
| INTEGER                                    |
+--------------------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a[1]'));
+--------------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@json_obj, '$.a[1]')) |
+--------------------------------------------+
| BOOLEAN                                    |
+--------------------------------------------+
```

JSON_VALID(val)

- 描述：判断参数值val是否为合法的json格式。
- 示例：

```
mysql> SELECT JSON_VALID('hello'), JSON_VALID('"hello"');
+--------------------+----------------------+
| JSON_VALID('hello') | JSON_VALID('"hello"') |
+--------------------+----------------------+
|                  0 |                    1 |
+--------------------+----------------------+
```

JSON_PRETTY(json_doc)

- 描述：通过添加换行与缩进，以更直观的方式打印出json_doc。
- 示例：

```
mysql> SET @json_doc = '["abc", [{"k1": 123}, "def"], {"k2": "abc"}, {"k3": null}]';
mysql> SELECT JSON_PRETTY(@json_doc);
+-----------------------------------------------------------------------------+
| JSON_PRETTY(@json_doc)                                                      |
+-----------------------------------------------------------------------------+
| [
    "abc",
    [
        {
            "k1":123
        },
        "def"
    ],
    {
        "k2":"abc"
    },
    {
    }
] |
+-----------------------------------------------------------------------------+
```

JSON_STORAGE_SIZE(json_doc)

- 描述：返回json_doc以二进制形式存储的字节大小。
- 示例：

```
mysql> SET @json_doc = '[999, "polardb-x", [1, 2, 3], 888.88]';
mysql> SELECT JSON_STORAGE_SIZE(@json_doc) AS Size;
+------+
| Size |
+------+
|   48 |
+------+
```

8.6.14. 其他函数

本文介绍了PolarDB-X支持的其他函数。

PolarDB-X还支持如下函数：

函数名	描述
RAND()	返回0到1的随机数
RELEASE_ALL_LOCKS()	释放所有锁
RELEASE_LOCK()	释放指定名称的锁
UUID()	返回UUID
UUID_SHORT()	返回值是一个unsigned long long类型的UUID
GET_LOCK()	获取锁
IS_USED_LOCK()	检查锁是否正在被使用
IS_FREE_LOCK()	检查锁是否被释放

PolarDB-X不支持如下函数：

函数名	描述
ANY_VALUE()	选择被分到同一组的数据里第一条数据的指定列值作为返回数据。
DEFAULT(col_name)	返回字段col_name的默认值。
INET_ATON()	返回IPv4地址对应的数值。
INET_NTOA()	返回数值对应的IPv4地址。
INET6_ATON()	返回IPv6地址对应的数值。
INET6_NTOA()	返回数值对应的IPv6地址。
IS_IPV4()	判断是否为一个IPv4地址。
IS_IPV4_COMPAT()	判断expr是否为一个IPv4兼容地址。
IS_IPV4_MAPPED()	判断expr是否为一个IPv4映射地址。
IS_IPV6()	判断是否是IPV6地址。
MASTER_POS_WAIT()	待当前从库达到这个位置后返回，返回期间执行的事务个数。
NAME_CONST()	根据指定的列名创建一列。

8.7. DDL语句

8.7.1. 分库分表语法

8.7.1.1. CREATE DATABASE

CREATE DATABASE语句用于创建数据库，并可以指定数据库的默认属性（如数据库默认字符集，校验规则等）。

语法

```
create_database_stmt:
    CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] database_name [database_option_list]
database_option_list:
    database_option [database_option ...]
database_option:
    [DEFAULT] {CHARACTER SET | CHARSET} [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  |  LOCALITY = locality_option}
  | [{PARTITION_MODE|MODE} = partition_mode_option]
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
partition_mode_option:
     {'auto'|'partitioning'}
    |{'drds'|'sharding'}
```

参数说明

参数	说明
database_name	指定要修改属性的数据库名称。如果不指定，会对当前默认数据库进行修改。
CHARSET charset_name	指定要修改的字符集。
COLLATE collation_name	指定校对规则。
LOCALITY	创建数据库时指定该库的存储位置。

MODE	指定逻辑库所使用的分区模式，支持两种模式：
	• auto/partitioning：自动分区模式
	  ◦ 不手工指定分区定义时，自动按主键分区；
	  ◦ 全库只能使用新的CREATE TABLE语法（如partition byhash/Range/List）进行建表。
	• drds/sharding：DRDS模式，全库只能使用DRDS模式的CREATE TABLE语法（如dbpartition by/tbpartition by）进行建表。
	注意
	• 创建数据库不指定mode，默认是“drds/sharding”；
	• 数据库的分区模式不可更改，创建数据库时一旦指定，不能变更。

示例

- 创建数据库test，并指定字符集为UTF8。

```
create database test default CHARACTER SET=UTF8 mode=auto;
Query OK, 1 row affected (3.36 sec)
show create database test;
+----------+--------------------------------------------+
| DATABASE | CREATE DATABASE                            |
+----------+--------------------------------------------+
| test     | CREATE DATABASE `test` /* MODE = 'auto' */ |
+----------+--------------------------------------------+
1 row in set (0.06 sec)
```

- 在实例中创建一个数据库，并通过以下命令指定其存储位置为*polardbx-storage-0-master*节点。

```
CREATE DATABASE db1 LOCALITY='dn=polardbx-storage-0-master';
```

> **说明**
> - 如果在创建数据库时未指定数据库的存储位置，系统将默认在所有存储节点中均匀分布数据库。
> - 数据库中分表的存储位置与该库的存储位置保持一致，以实现分表上的数据隔离。

创建成功后，您可以通过以下语句查看数据库的存储位置信息。

```
SHOW CREATE DATABASE db1;
```

返回结果如下所示：

```
+----------+-----------------------------------------------------------------------------+
| DATABASE | CREATE DATABASE                                                             |
+----------+-----------------------------------------------------------------------------+
| db1      | CREATE DATABASE `db1` /* LOCALITY = "dn=polardbx-storage-0-master" */  |
+----------+-----------------------------------------------------------------------------+
1 row in set
```

8.7.1.2. DROP DATABASE

DROP DATABASE语句用于删除数据库。

语法

```
drop_database_stmt:
    DROP DATABASE [IF EXISTS] database_name;
```

参数说明

参数	说明
IF EXISTS	用于防止当数据库不存在时发生错误。
database_name	指定待删除的数据库名。

示例

删除数据库test。

```
mysql>drop database test;
Query OK, 0 rows affected (0.03 sec)
mysql>drop database if exists test;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

8.7.1.3. CREATE TABLE

本文主要介绍使用DDL语句进行建表的语法、子句、参数和基本方式。

语法

```
CREATE [SHADOW] TABLE [IF NOT EXISTS] tbl_name
    (create_definition, ...)
    [table_options]
    [drds_partition_options]
create_definition:
    col_name column_definition
  | mysql_create_definition
  | [UNIQUE] GLOBAL INDEX index_name [index_type] (index_sharding_col_name,...)
      [global_secondary_index_option]
      [index_option] ...
# 全局二级索引相关
global_secondary_index_option:
    [COVERING (col_name,...)]
    [drds_partition_options]
# 分库分表子句
drds_partition_options:
    DBPARTITION BY db_partition_algorithm
    [TBPARTITION BY table_partition_algorithm [TBPARTITIONS num]]
    [LOCALITY=locality_option]
db_sharding_algorithm:
    HASH([col_name])
  | {YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
table_sharding_algorithm:
    HASH(col_name)
  | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
# 可以在创建单表时指定该表的存储位置
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
# 以下为MySQL DDL语法
index_sharding_col_name:
    col_name [(length)] [ASC | DESC]
index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
index_type:
    USING {BTREE | HASH}
```

> **说明** PolarDB-X DDL语法基于MySQL语法，以上主要列出了差异部分，详细语法请参见MySQL 文档。

分库分表子句和参数

- `DBPARTITION BY hash(partition_key)`：指定分库键和分库算法。
- `TBPARTITION BY { HASH(column) | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(column)`（可选）：默认与 `DBPARTITION BY` 相同，指定物理表使用什么方式映射数据。

- `TBPARTITIONS num`（可选）：每个库上的物理表数目（默认为1），如不分表，就不需要指定该字段。
- 拆分函数的详细介绍，请参见拆分函数 。

全局二级索引定义子句

- `[UNIQUE] GLOBAL`：定义全局二级索引，UNIQUE GLOBAL代表全局唯一索引。
- `index_name`：索引名，也是索引表的名称。
- `index_type`：索引表中分库分表键上局部索引的类型，支持范围请参见MySQL 文档。
- `index_sharding_col_name,...`：索引列，包含且仅包含索引表的全部分库分表键。详情请参见全局二级索引。
- `global_secondary_index_option`：PolarDB-X全局二级索引的扩展语法。
  - `COVERING (col_name,...)`：覆盖列，索引表中除索引列以外的其他列，默认包含主键和主表的分库分表键。详情请参见全局二级索引。
  - `drds_partition_options`：索引表的分库分表子句，详情请参见分库分表子句和参数。
- `index_option`：索引表中分库分表键上局部索引的属性，详情请参见MySQL 文档。

全链路压测影子表子句

`SHADOW`：创建全链路压测影子表，表名必须以`_test_`为前缀，前缀后的表名部分必须与关联的正式表名一致，且正式表必须先于影子表创建。

LOCALITY

通过LOCALITY指定单表的存储位置。

> **说明**
> - 通过LOCALITY语法指定了数据库或单表的位置之后，不支持再修改该库或该表的存储位置。
> - 数据库中单表的存储位置不受该库存储位置的影响。创建单表时若未指定存储位置，则会被随机放置在一个存储节点上，且后续该PolarDB-X实例上创建的所有未指定存储位置的单表，均会被放置在该存储节点上。

单库单表

建一张单库单表，不做任何拆分。

```
CREATE TABLE single_tbl(
 id bigint not null auto_increment,
 name varchar(30),
 primary key(id)
);
```

查看逻辑表的节点拓扑，可以看出只在0库创建了一张单库单表的逻辑表。

```
mysql> show topology from single_tbl;
+------+--------------------------------------------------------------------+-------------+
| ID   | GROUP_NAME                                                         | TABLE_NAME |
+------+--------------------------------------------------------------------+-------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | single_tbl |
+------+--------------------------------------------------------------------+-------------+
1 row in set (0.01 sec)
```

假设已有一个PolarDB-X实例，该实例上已有*polardbx-storage-0-master*和*polardbx-storage-1-master*两个存储节点，其中*polardbx-storage-1-master*上已创建了一个数据库db1。现需要在db1库中创建一张表，并指定其的存储位置为*polardbx-storage-0-master*，语法如下：

```
mysql>CREATE TABLE tb1 (id int) LOCALITY='dn=polardbx-storage-0-master';
```

创建成功后，您可以通过如下语句查看该表的拓扑结构：

```
mysql>SHOW TOPOLOGY FROM tb1;
```

返回结果如下：

```
+----+------------------+------------+
| ID | GROUP_NAME       | TABLE_NAME |
+----+------------------+------------+
| 0  | DB1_000000_GROUP | tb1        |
+----+------------------+------------+
1 row in set
```

> **说明** 以上返回结果表示tb1表位于DB1_000000_GROUP分库。

分库不分表

假设已经建好的分库数为8，建一张表，只分库不分表，分库方式为根据ID列哈希。

```
CREATE TABLE multi_db_single_tbl(
  id bigint not null auto_increment,
  name varchar(30),
  primary key(id)
) dbpartition by hash(id);
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了1张分表，既只做了分库。

```
mysql> show topology from multi_db_single_tbl;
+------+-------------------------------------------------------------------+---------------------+
| ID   | GROUP_NAME                                                        | TABLE_NAME          |
+------+-------------------------------------------------------------------+---------------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_single_tbl |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_single_tbl |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_single_tbl |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_single_tbl |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_single_tbl |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_single_tbl |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_single_tbl |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_single_tbl |
+------+-------------------------------------------------------------------+---------------------+
8 rows in set (0.01 sec)
```

分库分表

您可以使用如下拆分方式进行分库分表：

- 使用哈希函数做拆分
- 使用双字段哈希函数做拆分
- 使用日期做拆分

> **说明** 以下示例均假设已经建好的分库数为8。

使用哈希函数做拆分

建一张表，既分库又分表，每个库含有3张物理表，分库拆分方式为按照ID列进行哈希，分表拆分方式为按照bid列进行哈希。您可以先根据ID列的值进行哈希运算，将表中数据分布在多个子库中，每个子库中的数据再根据bid列值的哈希运算结果分布在3个物理表中。

```
CREATE TABLE multi_db_multi_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 primary key(id)
) dbpartition by hash(id) tbpartition by hash(bid) tbpartitions 3;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了3张分表。

```
mysql> show topology from multi_db_multi_tbl;
+------+-------------------------------------------------------------------+-----------------------+
| ID   | GROUP_NAME                                                        | TABLE_NAME            |
+------+-------------------------------------------------------------------+-----------------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_09 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_10 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_11 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_12 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_13 |
|   14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_14 |
|   15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_15 |
|   16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_16 |
|   17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_17 |
|   18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_18 |
|   19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_19 |
|   20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_20 |
|   21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_21 |
|   22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_22 |
|   23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_23 |
+------+-------------------------------------------------------------------+-----------------------+
24 rows in set (0.01 sec)
```

查看该逻辑表的拆分规则，可以看出分库分表的拆分方式均为哈希，分库的拆分键为ID，分表的拆分键为bid。

```
mysql> show rule from multi_db_multi_tbl;
+------+--------------------+-----------+------------------+---------------------+---------------------+-----
-------------+---------------------+---------------------+
| ID   | TABLE_NAME         | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_P
ARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+--------------------+-----------+------------------+---------------------+---------------------+-----
-------------+---------------------+---------------------+
|    0 | multi_db_multi_tbl |         0 | id               | hash                | 8                   | bid
| hash                | 3                   |
+------+--------------------+-----------+------------------+---------------------+---------------------+-----
-------------+---------------------+---------------------+
1 row in set (0.01 sec)
```

使用双字段哈希函数做拆分

- 使用要求拆分键的类型必须是字符类型或数字类型。
- 路由方式根据任一拆分键后N位计算哈希值，以哈希方式完成路由计算。N为函数第三个参数。例如 `RANGE_HASH(COL1, COL2, N)` ，计算时会优先选择COL1，截取其后N位进行计算。COL1不存在时按COL2计算。

- 适用场景适合于需要有两个拆分键，并且仅使用其中一个拆分键值进行查询时的场景。假设用户的PolarDB-X里已经分了8个物理库， 现业务有如下的场景：
  - 一个业务想按买家ID和订单ID对订单表进行分库。
  - 查询时条件仅有买家ID或订单ID。

此时可使用以下DDL对订单表进行构建：

```
create table test_order_tb (
 id bigint not null auto_increment,
 seller_id varchar(30) DEFAULT NULL,
 order_id varchar(30) DEFAULT NULL,
 buyer_id varchar(30) DEFAULT NULL,
 create_time datetime DEFAULT NULL,
 primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by RANGE_HASH(buyer_id, order_id, 10) tbpartition by RANGE_
HASH(buyer_id, order_id, 10) tbpartitions 3;
```

> **说明**
> - 两个拆分键皆不能修改。
> - 插入数据时如果发现两个拆分键指向不同的分库或分表时，插入会失败。

使用日期做拆分

除了可以使用哈希函数做拆分算法，您还可以使用日期函数 `MM` 、 `DD` 、 `WEEK` 或 `MMDD` 来作为分表的拆分算法，具体步骤请参见如下示例。

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为根据 `actionDate` 列，按照一周七天来拆分（ `WEEK(actionDate)` 计算的是 `DAY_OF_WEEK` ）。

比如 `actionDate` 列的值是2017-02-27，这天是星期一， `WEEK(actionDate)` 算出的值是2，该条记录就会被存储到 `2`（`2 % 7 = 2`） 这张分表（位于某个分库，具体的表名是 `user_log_2` ）；比如 `actionDate` 列的值是2017-02-26，这天是星期天， `WEEK(actionDate)` 算出的值是1，该条记录就会被存储到 `1`（`1 % 7 = 1`） 这张分表（位于某个分库，具体的表名是 `user_log_1` ）。

```
CREATE TABLE user_log(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by WEEK(actionDate) tbpartitions 7;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了7张分表（一周7天）。

```
mysql> show topology from user_log;
+------+-------------------------------------------------------------------+------------+
| ID   | GROUP_NAME                                                        | TABLE_NAME |
+------+-------------------------------------------------------------------+------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_0 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_1 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_2 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_3 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_4 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_5 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_6 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_0 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_1 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_2 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_3 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_4 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_5 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_6 |
...
|   49 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_0 |
|   50 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_1 |
|   51 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_2 |
|   52 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_3 |
|   53 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_4 |
|   54 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_5 |
|   55 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_6 |
+------+-------------------------------------------------------------------+------------+
56 rows in set (0.01 sec)
```

说明 由于返回结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `WEEK` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log;
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+---------------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+---------------------+--------------------+
|    0 | user_log   |         0 | userId          | hash               | 8                  | actionDate
| week               | 7                  |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+---------------------+--------------------+
1 row in set (0.00 sec)
```

查看给定分库键和分表键参数时， SQL被路由到哪个物理分库和该物理分库下的哪张物理表。

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为根据 `actionDate` 列，按照一年12个月进行拆分（ `MM(actionDate)` 计算的是 `MONTH_OF_YEAR` ）。

比如 `actionDate` 列的值是2017-02-27， `MM(actionDate)` 算出的值是02，该条记录就会被存储到 `02（02 % 12 = 02）` 这张分表（位于某个分库，具体的表名是 `user_log_02` ）。比如 `actionDate` 列的值是2016-12-27， `MM(actionDate)` 算出的值是12，该条记录就会被存储到 `00（12 % 12 = 00）` 这张分表（位于某个分库，具体的表名是 `user_log_00` ）。

```
CREATE TABLE user_log2(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MM(actionDate) tbpartitions 12;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了12张分表（1年有12个月）。

```
mysql> show topology from user_log2;
+------+--------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                                         | TABLE_NAME   |
+------+--------------------------------------------------------------------+--------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_09 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_10 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_11 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_00 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_01 |
|   14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_02 |
|   15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_03 |
|   16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_04 |
|   17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_05 |
|   18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_06 |
|   19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_07 |
|   20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_08 |
|   21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_09 |
|   22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_10 |
|   23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_11 |
...
|   84 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_00 |
|   85 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_01 |
|   86 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_02 |
|   87 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_03 |
|   88 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_04 |
|   89 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_05 |
|   90 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_06 |
|   91 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_07 |
|   92 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_08 |
|   93 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_09 |
|   94 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_10 |
|   95 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_11 |
+------+--------------------------------------------------------------------+--------------+
96 rows in set (0.02 sec)
```

> **说明** 由于返回结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MM` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log2;
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
|    0 | user_log2  |         0 | userId          | hash               | 8                  | actionDate
| mm                 | 12                 |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
1 row in set (0.00 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一个月31天进行拆分（函数 `DD(actionDate)` 计算的是 `DAY_OF_MONTH` ）。

比如 `actionDate` 列的值是2017-02-27， `DD(actionDate)` 算出的值是27，该条记录就会被存储到 `27`（`27 % 31 = 27`） 这张分表（位于某个分库，具体的表名是 `user_log_27` ）。

```
CREATE TABLE user_log3(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by DD(actionDate) tbpartitions 31;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了31张分表（按每个月有31天处理）。

```
mysql> show topology from user_log3;
+------+-------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                                        | TABLE_NAME   |
+------+-------------------------------------------------------------------+--------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_09 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_10 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_11 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_12 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_13 |
|   14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_14 |
|   15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_15 |
|   16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_16 |
|   17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_17 |
|   18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_18 |
|   19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_19 |
|   20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_20 |
|   21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_21 |
|   22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_22 |
|   23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_23 |
|   24 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_24 |
|   25 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_25 |
|   26 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_26 |
|   27 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_27 |
|   28 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_28 |
|   29 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_29 |
|   30 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_30 |
...
|  237 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_20 |
|  238 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_21 |
|  239 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_22 |
|  240 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_23 |
|  241 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_24 |
|  242 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_25 |
|  243 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_26 |
|  244 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_27 |
|  245 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_28 |
|  246 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_29 |
|  247 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_30 |
+------+-------------------------------------------------------------------+--------------+
248 rows in set (0.01 sec)
```

说明 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `DD` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log3;
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
|    0 | user_log3  |         0 | userId          | hash               | 8                  | actionDate
| dd                 | 31                 |
+------+------------+-----------+-----------------+--------------------+--------------------+-------------
-----+--------------------+--------------------+
1 row in set (0.01 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一年365天进行拆分，路由到365张物理表（ `MMDD(actionDate) tbpartitions 365` 计算的是 `DAY_OF_YEAR % 365` 。

比如 `actionDate` 列的值是2017-02-27， `MMDD(actionDate)` 算出的值是58，该条记录就会被存储到58这张分表（位于某个分库，具体的表名是 `user_log_58` ）。

```
CREATE TABLE user_log4(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 365;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了365张分表（按每年有365天处理）。

```
mysql> show topology from user_log4;
+------+--------------------------------------------------------------------+---------------+
| ID   | GROUP_NAME                                                         | TABLE_NAME    |
+------+--------------------------------------------------------------------+---------------+
...
| 2896 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_341 |
| 2897 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_342 |
| 2898 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_343 |
| 2899 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_344 |
| 2900 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_345 |
| 2901 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_346 |
| 2902 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_347 |
| 2903 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_348 |
| 2904 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_349 |
| 2905 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_350 |
| 2906 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_351 |
| 2907 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_352 |
| 2908 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_353 |
| 2909 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_354 |
| 2910 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_355 |
| 2911 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_356 |
| 2912 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_357 |
| 2913 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_358 |
| 2914 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_359 |
| 2915 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_360 |
| 2916 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_361 |
| 2917 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_362 |
| 2918 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_363 |
| 2919 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_364 |
+------+--------------------------------------------------------------------+---------------+
2920 rows in set (0.07 sec)
```

> **说明** 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MMDD` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log4;
+------+------------+-----------+------------------+---------------------+---------------------+-------------
-----+---------------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+------------------+---------------------+---------------------+-------------
-----+---------------------+--------------------+
|    0 | user_log4  |         0 | userId           | hash                | 8                   | actionDate
| mmdd                | 365                 |
+------+------------+-----------+------------------+---------------------+---------------------+-------------
-----+---------------------+--------------------+
1 row in set (0.02 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一年365天进行拆分，路由到10张物理表（ `MMDD(actionDate) tbpartitions 10` 计算的是 `DAY_OF_YEAR % 10` 。

```
CREATE TABLE user_log5(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 10;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了10张分表（按照一年365天进行拆分，路由到10张物理表）。

```
mysql> show topology from user_log5;
+------+-------------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                                              | TABLE_NAME   |
+------+-------------------------------------------------------------------------+--------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_09 |
...
|   70 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_00 |
|   71 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_01 |
|   72 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_02 |
|   73 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_03 |
|   74 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_04 |
|   75 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_05 |
|   76 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_06 |
|   77 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_07 |
|   78 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_08 |
|   79 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_09 |
+------+-------------------------------------------------------------------------+--------------+
80 rows in set (0.02 sec)
```

> **说明** 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MMDD` 进行拆分，路由到10张物理表，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log5;
+------+------------+-----------+------------------+---------------------+--------------------+-------------
-----+---------------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+------------------+---------------------+--------------------+-------------
-----+---------------------+--------------------+
|    0 | user_log5  |         0 | userId           | hash                | 8                  | actionDate
| mmdd                | 10                  |
+------+------------+-----------+------------------+---------------------+--------------------+-------------
-----+---------------------+--------------------+
1 row in set (0.01 sec)
```

广播表

子句 `BROADCAST` 用来指定创建广播表。广播表是指将这个表复制到每个分库上，在分库上通过同步机制实现数据一致，有秒级延迟。这样做的好处是可以将JOIN操作下推到底层的RDS（MySQL），来避免跨库JOIN。关于如何使用广播表来做SQL优化，详情请参见SQL调优指南。

```
CREATE TABLE brd_tbl(
  id bigint not null auto_increment,
  name varchar(30),
  primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 BROADCAST;
```

其他MySQL建表属性

您在分库分表的同时还可以指定其他的MySQL建表属性，例如：

```
CREATE TABLE multi_db_multi_tbl(
  id bigint not null auto_increment,
  name varchar(30),
  primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(id) tbpartition by hash(id) tbpartitions 3;
```

全局二级索引

本小节介绍如何在建表时定义全局二级索引：

- 定义全局二级索引
- 定义全局唯一索引

> **说明** 以下示例均假设已经建好的分库数为8。

定义全局二级索引

示例

```
CREATE TABLE t_order (
 `id` bigint(11) NOT NULL AUTO_INCREMENT,
 `order_id` varchar(20) DEFAULT NULL,
 `buyer_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 `order_snapshot` longtext DEFAULT NULL,
 `order_detail` longtext DEFAULT NULL,
 PRIMARY KEY (`id`),
 GLOBAL INDEX `g_i_seller`(`seller_id`) dbpartition by hash(`seller_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
```

其中：

- 主表：`t_order` 只分库不分表，分库的拆分方式为按照 `order_id` 列进行哈希。
- 索引表：`g_i_seller` 只分库不分表，分库的拆分方式为按照 `seller_id` 列进行哈希，未指定覆盖列。
- 索引定义子句：`GLOBAL INDEX `g_i_seller`(`seller_id`) dbpartition by hash(`seller_id`)` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键 `order_id` 上的局部索引，和 `seller_id` 、 `id` 、 `order_id` 上的GSI，其中 `seller_id` 为索引表的拆分键， `id` 和 `order_id` 为默认的覆盖列（主键和主表的拆分键）。

> **说明** 关于GSI的限制与约定，请参见如何使用全局二级索引，关于SHOW INDEX详细说明，请参见SHOW INDEX 。

```
mysql> show index from t_order;
+---------+------------+-------------------------+--------------+-------------+-----------+-------------+---
-------+--------+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME                | SEQ_IN_INDEX | COLUMN_NAME | COLLATION | CARDINALITY | SU
B_PART | PACKED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+---------+------------+-------------------------+--------------+-------------+-----------+-------------+---
-------+--------+------+------------+----------+---------------+
| t_order |          0 | PRIMARY                 |            1 | id          | A         |           0 |
NULL | NULL   |      | BTREE      |          |               |
| t_order |          1 | auto_shard_key_order_id |            1 | order_id    | A         |           0 |
NULL | NULL   | YES  | BTREE      |          |               |
| t_order |          1 | g_i_seller              |            1 | seller_id   | NULL      |           0 |
NULL | NULL   | YES  | GLOBAL     | INDEX    |               |
| t_order |          1 | g_i_seller              |            2 | id          | NULL      |           0 |
NULL | NULL   |      | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_seller              |            3 | order_id    | NULL      |           0 |
NULL | NULL   | YES  | GLOBAL     | COVERING |               |
+---------+------------+-------------------------+--------------+-------------+-----------+-------------+---
-------+--------+------+------------+----------+---------------+
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息。详细说明请参见SHOW GLOBAL INDEX。

```
 mysql> show global index from t_order;
+--------+---------+------------+------------+-------------+---------------+------------+------------------
+---------------------+--------------------+-----------------+---------------------+--------------------+--
------+
| SCHEMA | TABLE   | NON_UNIQUE | KEY_NAME   | INDEX_NAMES | COVERING_NAMES | INDEX_TYPE | DB_PARTITION_KEY
| DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT | S
TATUS |
+--------+---------+------------+------------+-------------+---------------+------------+------------------
+---------------------+--------------------+-----------------+---------------------+--------------------+--
------+
| d7     | t_order | 1          | g_i_seller | seller_id   | id, order_id  | NULL       | seller_id
| HASH                | 8                  |                 | NULL                | NULL               | P
UBLIC |
+--------+---------+------------+------------+-------------+---------------+------------+------------------
+---------------------+--------------------+-----------------+---------------------+--------------------+--
------+
```

查看索引表的结构，索引表包含主表的主键、分库分表键和默认的覆盖列，主键列去除了 `AUTO_INCREMENT` 属性，并且去除了主表中的局部索引。

```
mysql> show create table g_i_seller;
+------------+-------------------------------------------------------------+
| Table      | Create Table                                                |
+------------+-------------------------------------------------------------+
| g_i_seller | CREATE TABLE `g_i_seller` (
 `id` bigint(11) NOT NULL,
 `order_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `auto_shard_key_seller_id` (`seller_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`seller_id`) |
+------------+-------------------------------------------------------------+
```

定义全局唯一索引

```
CREATE TABLE t_order (
 `id` bigint(11) NOT NULL AUTO_INCREMENT,
 `order_id` varchar(20) DEFAULT NULL,
 `buyer_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 `order_snapshot` longtext DEFAULT NULL,
 `order_detail` longtext DEFAULT NULL,
 PRIMARY KEY (`id`),
 UNIQUE GLOBAL INDEX `g_i_buyer`(`buyer_id`) COVERING(`seller_id`, `order_snapshot`)
   dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_id`) tbpartitions 3
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
```

其中：

- 主表：`t_order` 只分库不分表，分库的拆分方式为按照 `order_id` 列进行哈希。
- 索引表：`g_i_buyer` 只分库且分表，分库和分表的拆分方式均为按照 `buyer_id` 列进行哈希，覆盖列包含 `seller_id` 和 `order_snapshot` 。
- 索引定义子句：`UNIQUE GLOBAL INDEX `g_i_buyer`(`buyer_id`) COVERING(`seller_id`, `order_snapshot`) dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_id`) tbpartitions 3` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键 `order_id` 上的局部索引，和 `buyer_id` 、 `id` 、 `order_id` 、 `seller_id` 和 `order_snapshot` 上的GSI，其中 `buyer_id` 为索引表的拆分键， `id` 和 `order_id` 为默认的覆盖列（主键和主表的拆分键）， `seller_id` 和 `order_snapshot` 为显示指定的覆盖列。

> **说明** 关于GSI的限制与约定，请参见如何使用全局二级索引，关于SHOW INDEX详细说明，请参见SHOW INDEX 。

```
mysql> show index from t_order;
+--------------+------------+-------------------------+--------------+----------------+-----------+---------
----+----------+--------+------+------------+----------+---------------+
| TABLE        | NON_UNIQUE | KEY_NAME                | SEQ_IN_INDEX | COLUMN_NAME    | COLLATION | CARDINAL
ITY | SUB_PART | PACKED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+--------------+------------+-------------------------+--------------+----------------+-----------+---------
----+----------+--------+------+------------+----------+---------------+
| t_order_dthb |          0 | PRIMARY                 |            1 | id             | A         |
0 |     NULL | NULL   |      | BTREE      |          |               |
| t_order_dthb |          1 | auto_shard_key_order_id |            1 | order_id       | A         |
0 |     NULL | NULL   | YES  | BTREE      |          |               |
| t_order      |          0 | g_i_buyer               |            1 | buyer_id       | NULL      |
0 |     NULL | NULL   | YES  | GLOBAL     | INDEX    |               |
| t_order      |          1 | g_i_buyer               |            2 | id             | NULL      |
0 |     NULL | NULL   |      | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer               |            3 | order_id       | NULL      |
0 |     NULL | NULL   | YES  | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer               |            4 | seller_id      | NULL      |
0 |     NULL | NULL   | YES  | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer               |            5 | order_snapshot | NULL      |
0 |     NULL | NULL   | YES  | GLOBAL     | COVERING |               |
+--------------+------------+-------------------------+--------------+----------------+-----------+---------
----+----------+--------+------+------------+----------+---------------+
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息。详细说明请参见SHOW GLOBAL INDEX。

```
mysql> show global index from t_order;
+--------+---------+------------+-----------+-------------+-----------------------------------------+-------
-----+------------------+--------------------+--------------------+------------------+---------------------
+--------------------+---------+
| SCHEMA | TABLE   | NON_UNIQUE | KEY_NAME  | INDEX_NAMES | COVERING_NAMES                          | INDEX_
TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY
| TB_PARTITION_COUNT | STATUS |
+--------+---------+------------+-----------+-------------+-----------------------------------------+-------
-----+------------------+--------------------+--------------------+------------------+---------------------
+--------------------+---------+
| d7     | t_order | 0          | g_i_buyer | buyer_id    | id, order_id, seller_id, order_snapshot | NULL
| buyer_id         | HASH               | 8                  | buyer_id         | HASH                | 3
| PUBLIC |
+--------+---------+------------+-----------+-------------+-----------------------------------------+-------
-----+------------------+--------------------+--------------------+------------------+---------------------
+--------------------+---------+
```

查看索引表的结构，索引表包含主表的主键、分库分表键、默认覆盖列和GSI定义中指定的覆盖列，主键列去除了 `AUTO_INCREMENT` 属性，并且去除了主表中局部索引，全局唯一索引默认会创建一份数据表来实现全局的唯一性支持。

```
mysql> show create table g_i_buyer;
+-----------+-----------------------------------------------------------------------------------------------------------
---------+
| Table     | Create Table
|
+-----------+-----------------------------------------------------------------------------------------------------------
---------+
| g_i_buyer | CREATE TABLE `g_i_buyer` (
  `id` bigint(11) NOT NULL,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext,
  PRIMARY KEY (`id`),
  UNIQUE KEY `auto_shard_key_buyer_id` (`buyer_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_id`) tbparti
tions 3 |
+-----------+-----------------------------------------------------------------------------------------------------------
---------+
```

8.7.1.4. DROP TABLE

您可以使用DROP TABLE语法删除指定表。

语法

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

> **说明** 通过PolarDB-X删除表，系统会自动对相关物理表进行删除操作。

注意事项

- 当删除包含全局二级索引的表时，索引表会和主表同时被删除。
- 禁止单独删除索引表，如需删除索引，请使用 `ALTER TABLE DROP INDEX` 语句。
- `DROP TABLE` 语句一次执行仅支持删除一张表，如果同时删除多张表会收到报错提示。

示例

您可以通过如下语句删除 `user_log` 表：

```
DROP TABLE user_log;
```

8.7.1.5. ALTER TABLE

您可以通过ALTER TABLE语法改变表的结构，如增加列、增加索引、修改数据定义等。

注意事项

不支持通过ALTER TABLE语法修改拆分字段。

语法

> **说明** ALTER TABLE用于改变表的结构，如增加列、增加索引和修改数据定义。详细语法请参见MySQL修改表语法

```
ALTER [ONLINE|OFFLINE] [IGNORE] TABLE tbl_name
    [alter_specification [, alter_specification] ...]
    [partition_options]
```

示例

- 增加列

  在“user_log”表中增加一列“idcard”，示例如下：

```
ALTER TABLE user_log
    ADD COLUMN idcard varchar(30);
```

- 增加局部索引

  在“user_log”表中为“idcard”列增加一个名为“idcard_idx”的索引，示例如下：

```
ALTER TABLE user_log
    ADD INDEX idcard_idx (idcard);
```

- 重命名局部索引

  将“user_log”表中“idcard_idx”索引命修改为“idcard_idx_new”，示例如下：

```
ALTER TABLE user_log
    RENAME INDEX `idcard_idx` TO `idcard_idx_new`;
```

- 删除局部索引

  删除“user_log”表中的“idcard_idx”索引，示例如下：

```
ALTER TABLE user_log
    DROP INDEX idcard_idx;
```

- 修改字段

  将“user_log”表中“idcard”列（字段类型为varchar）的长度由30改为40，语法示例如下：

```
ALTER TABLE user_log
    MODIFY COLUMN idcard varchar(40);
```

全局二级索引

PolarDB-X支持全局二级索引 (Global Secondary Index, GSI)，基本原理请参见全局二级索引。

列变更

使用全局二级索引的表，对列的修改，语法和普通表的一致。

> **说明** 当修改的表包含全局二级索引时，对列的修改有额外的限制，关于GSI的限制与约定，详情请参见如何使用全局二级索引。

索引变更

语法

```
ALTER TABLE tbl_name
    alter_specification # 全局二级索引相关变更仅支持一条alter_specification
alter_specification:
  | ADD GLOBAL {INDEX|KEY} index_name # 全局二级索引必须显式指定索引名
      [index_type] (index_sharding_col_name,...)
      global_secondary_index_option
      [index_option] ...
  | ADD [CONSTRAINT [symbol]] UNIQUE GLOBAL
      [INDEX|KEY] index_name # 全局二级索引必须显式指定索引名
      [index_type] (index_sharding_col_name,...)
      global_secondary_index_option
      [index_option] ...
  | DROP {INDEX|KEY} index_name
  | RENAME {INDEX|KEY} old_index_name TO new_index_name
global_secondary_index_option:
    [COVERING (col_name,...)] # Covering Index
    drds_partition_options # 包含且仅包含index_sharding_col_name中指定的列
# 指定索引表拆分方式
drds_partition_options:
    DBPARTITION BY db_sharding_algorithm
    [TBPARTITION BY {table_sharding_algorithm} [TBPARTITIONS num]]
db_sharding_algorithm:
    HASH([col_name])
  | {YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
table_sharding_algorithm:
    HASH(col_name)
  | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
# 以下为MySQL DDL语法
index_sharding_col_name:
    col_name [(length)] [ASC | DESC]
index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
index_type:
    USING {BTREE | HASH}
```

`ALTER TABLE ADD GLOBAL INDEX` 系列语法用于在建表后添加GSI，该系列语法在MySQL语法上新引入了GLOBAL关键字，用于指定添加的索引类型为GSI。

`ALTER TABLE { DROP | RENAME } INDEX` 语法同样可以对GSI进行修改，目前建表后创建GSI存在一定限制。关于GSI的限制与约定，详情请参见如何使用全局二级索引。

全局二级索引定义子句详细说明请参见CREATE TABLE。

示例

- **建表后添加全局二级索引**

  下面以建立全局唯一索引为例，介绍在建表后如何创建GSI。

```
# 创建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
# 创建全局二级索引
ALTER TABLE t_order ADD UNIQUE GLOBAL INDEX `g_i_buyer` (`buyer_id`) COVERING (`order_snapshot`) dbpartiti
on by hash(`buyer_id`);
```

- 主表：” t_order “只分库不分表，分库的拆分方式为按照” order_id “列进行哈希。
- 索引表：” g_i_buyer “只分库不分表，分库的拆分方式为按照” buyer_id “列进行哈希，指定覆盖列为” order_snapshot “。
- 索引定义子句：`GLOBAL INDEX `g_i_seller` ON t_order (`seller_id`) dbpartition by hash(`seller_id`)` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键order_id上的局部索引，和buyer_id、id、order_id和order_snapshot上的GSI，其中buyer_id为索引表的拆分键，id和order_id为默认的覆盖列（主键和主表的拆分键），order_snapshot显式指定的覆盖列。

> **说明** 关于GSI的限制与约定，详情请参见如何使用全局二级索引，SHOW INDEX详细说明，请参见SHOW INDEX。

```
mysql> show index from t_order;
+---------+------------+-----------+--------------+---------------+-----------+-------------+----------+-
-------+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME  | SEQ_IN_INDEX | COLUMN_NAME    | COLLATION | CARDINALITY | SUB_PART |
PACKED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+---------+------------+-----------+--------------+---------------+-----------+-------------+----------+-
-------+------+------------+----------+---------------+
| t_order |          0 | PRIMARY   |            1 | id             | A         |           0 |     NULL |
NULL   |      | BTREE      |          |               |
| t_order |          1 | l_i_order |            1 | order_id       | A         |           0 |     NULL |
NULL   | YES  | BTREE      |          |               |
| t_order |          0 | g_i_buyer |            1 | buyer_id       | NULL      |           0 |     NULL |
NULL   | YES  | GLOBAL     | INDEX    |               |
| t_order |          1 | g_i_buyer |            2 | id             | NULL      |           0 |     NULL |
NULL   |      | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_buyer |            3 | order_id       | NULL      |           0 |     NULL |
NULL   | YES  | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_buyer |            4 | order_snapshot | NULL      |           0 |     NULL |
NULL   | YES  | GLOBAL     | COVERING |               |
+---------+------------+-----------+--------------+---------------+-----------+-------------+----------+-
-------+------+------------+----------+---------------+
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息，详情请参见SHOW GLOBAL INDEX。

```
mysql> show global index from t_order;
+---------------------+---------+------------+-----------+-------------+-----------------------------+---
---------+------------------+---------------------+--------------------+------------------+---------------
------+--------------------+--------+
| SCHEMA              | TABLE   | NON_UNIQUE | KEY_NAME  | INDEX_NAMES | COVERING_NAMES              | IN
DEX_TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_P
OLICY | TB_PARTITION_COUNT | STATUS |
+---------------------+---------+------------+-----------+-------------+-----------------------------+---
---------+------------------+---------------------+--------------------+------------------+---------------
------+--------------------+--------+
| ZZY3_DRDS_LOCAL_APP | t_order | 0          | g_i_buyer | buyer_id    | id, order_id, order_snapshot | NU
LL       | buyer_id         | HASH                | 4                  |                  | NULL
| NULL               | PUBLIC |
+---------------------+---------+------------+-----------+-------------+-----------------------------+---
---------+------------------+---------------------+--------------------+------------------+---------------
------+--------------------+--------+
```

查看索引表的结构，索引表包含主表的主键、分库分表键、默认的覆盖列和自定义覆盖列，主键列去除了AUTO_INCREMENT属性，并且去除了主表中的局部索引，全局唯一索引默认在索引表的所有分库分表键上创建一个唯一索引，以实现全局唯一约束。

```
mysql> show create table g_i_buyer;
+-----------+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------+
| Table     | Create Table
|
+-----------+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------+
| g_i_buyer | CREATE TABLE `g_i_buyer` (`id` bigint(11) NOT NULL, `order_id` varchar(20) DEFAULT NULL, `bu
yer_id` varchar(20) DEFAULT NULL, `order_snapshot` longtext, PRIMARY KEY (`id`), UNIQUE KEY `auto_shard_ke
y_buyer_id` (`buyer_id`) USING BTREE) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`buyer_id`)
|
+-----------+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------+
```

- **删除全局二级索引**

  删除名为g_i_seller的GSI，相应的索引表也将被删除。

  ```
  # 删除索引
  ALTER TABLE `t_order` DROP INDEX `g_i_seller`;
  ```

- **重命名索引**

  默认情况下限制对GSI的重命名。关于GSI的限制与约定，详情请参见全局二级索引使用。

8.7.1.6. TRUNCATE TABLE

本文介绍了如何清空表数据。

语法

```
TRUNCATE [TABLE] tbl_name
```

详细语法参考 TRUNCATE TABLE Statement。

注意事项

使用TRUNCATE TABLE，需要有DROP权限。

8.7.1.7. RENAME TABLE

本文介绍了对表做重命名的方法。

语法

```
RENAME TABLE
    tbl_name TO new_tbl_name
```

注意事项

- PolarDB-X不支持同时RENAME多张表。
- 考虑到稳定性和性能，目前不支持重命名包含全局二级索引的表。
- 不支持单独重命名索引表。如需重命名索引，请参见ALTER TABLE中RENAME INDEX推荐的方法。

8.7.1.8. CREATE INDEX

PolarDB-X支持创建局部索引和全局二级索引 (Global Secondary Index, GSI)，同时支持删除这两种索引。

局部索引

关于局部索引，详情请参见CREATE INDEX Statement。

全局二级索引

关于全局二级索引基本原理，请参见全局二级索引。

语法

```
CREATE [UNIQUE]
    GLOBAL INDEX index_name [index_type]
    ON tbl_name (index_sharding_col_name,...)
    global_secondary_index_option
    [index_option]
    [algorithm_option | lock_option] ...
# 全局二级索引特有语法，具体说明请参见CREATE TABLE文档
global_secondary_index_option:
    [COVERING (col_name,...)]
    drds_partition_options
# 分库分表子句，具体说明请参见CREATE TABLE文档
drds_partition_options:
    DBPARTITION BY db_sharding_algorithm
    [TBPARTITION BY {table_sharding_algorithm} [TBPARTITIONS num]]
db_sharding_algorithm:
    HASH([col_name])
  | {YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
table_sharding_algorithm:
    HASH(col_name)
  | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
 # 以下为MySQL DDL语法
index_sharding_col_name:
    col_name [(length)] [ASC | DESC] # length参数仅用于在索引表拆分键上创建局部索引
index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
index_type:
    USING {BTREE | HASH}
algorithm_option:
    ALGORITHM [=] {DEFAULT|INPLACE|COPY}
lock_option:
    LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
```

`CREATE GLOBAL INDEX` 系列语法用于在建表后添加GSI，该系列语法在MySQL语法上新引入了GLOBAL关键字，用于指定添加的索引类型为GSI。目前建表后创建GSI存在一定限制，关于GSI的限制与约定，详情请参见如何使用全局二级索引。

关于全局二级索引定义子句详细说明，请参见CREATE TABLE。

示例

下面以建立普通全局二级索引为例，介绍在建表后创建GSI。

```
# 创建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
# 创建全局二级索引
ALTER TABLE t_order ADD UNIQUE GLOBAL INDEX `g_i_buyer` (`buyer_id`) COVERING (`order_snapshot`) dbpartition
by hash(`buyer_id`);
```

- 主表：” t_order “只分库不分表，分库的拆分方式为按照” order_id “列进行哈希。
- 索引表：” g_i_buyer “只分库不分表，分库的拆分方式为按照” buyer_id “列进行哈希，指定覆盖列为” order_snapshot “。
- 索引定义子句：`GLOBAL INDEX `g_i_seller` ON t_order (`seller_id`) dbpartition by hash(`seller_id`)` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键order_id上的局部索引，和seller_id、id和order_id上的GSI，其中seller_id为索引表的拆分键，id和order_id为默认的覆盖列（主键和主表的拆分键）。

> **说明** 关于GSI的限制与约定，详情请参见如何使用全局二级索引，SHOW INDEX详细说明，请参见SHOW INDEX。

```
mysql> show index from t_order;
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME   | SEQ_IN_INDEX | COLUMN_NAME | COLLATION | CARDINALITY | SUB_PART | PACK
ED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
| t_order |          0 | PRIMARY    |            1 | id          | A         |           0 |     NULL | NULL
|      | BTREE      |          |               |
| t_order |          1 | l_i_order  |            1 | order_id    | A         |           0 |     NULL | NULL
| YES  | BTREE      |          |               |
| t_order |          1 | g_i_seller |            1 | seller_id   | NULL      |           0 |     NULL | NULL
| YES  | GLOBAL     | INDEX    |               |
| t_order |          1 | g_i_seller |            2 | id          | NULL      |           0 |     NULL | NULL
|      | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_seller |            3 | order_id    | NULL      |           0 |     NULL | NULL
| YES  | GLOBAL     | COVERING |               |
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息，详情请参见SHOW GLOBAL INDEX。

```
mysql> show global index from t_order;
+--------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+--------------------+--------------------+-----------------+---------------------+----------
----------+--------+
| SCHEMA              | TABLE   | NON_UNIQUE | KEY_NAME   | INDEX_NAMES | COVERING_NAMES | INDEX_TYPE | DB_P
ARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTIT
ION_COUNT | STATUS |
+--------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+--------------------+--------------------+-----------------+---------------------+----------
----------+--------+
| ZZY3_DRDS_LOCAL_APP | t_order | 1          | g_i_seller | seller_id   | id, order_id   | NULL       | sell
er_id        | HASH                | 4                  |                  | NULL                | NULL
| PUBLIC |
+--------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+--------------------+--------------------+-----------------+---------------------+----------
----------+--------+
```

查看索引表的结构，索引表包含主表的主键、分库分表键、默认的覆盖列和自定义覆盖列，主键列去除了 `AUTO_INCREMENT` 属性，并且去除了主表中的局部索引。

```
mysql> show create table g_i_seller;
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------+
| Table      | Create Table
|
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------+
| g_i_seller | CREATE TABLE `g_i_seller` (
  `id` bigint(11) NOT NULL,
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `auto_shard_key_seller_id` (`seller_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`seller_id`) |
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------+
```

8.7.1.9. DROP INDEX

本文介绍了如何删除局部索引和全局二级索引。

局部索引

删除局部索引的方法和MySQL中一致，请参考DROP INDEX 文档。

全局二级索引

语法

```
DROP INDEX index_name ON tbl_name
```

> **说明** index_name为需要删除的全局二级索引名。

8.7.1.10. CREATE VIEW

本文将介绍如何使用CREATE VIEW语句为PolarDB-X创建视图。

语法

```
CREATE
    [OR REPLACE]
    VIEW view_name [(column_list)]
    AS select_statement
```

示例

```
# 先建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
# 创建视图
create view t_detail as select order_id,order_detail from t_order;
# 查询视图
select * from t_detail;
```

8.7.1.11. DROP VIEW

本文将介绍如何使用DROP VIEW语句删除PolarDB-X的视图。

语法

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

示例

```
# 创建视图
create view v as select 1;
# 删除视图
drop view v;
```

8.7.2. 分区表语法

8.7.2.1. CREATE DATABASE

CREATE DATABASE语句用于创建数据库，并可以指定数据库的默认属性（如数据库默认字符集，校验规则等）。

语法

```
create_database_stmt:
    CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] database_name [database_option_list]
database_option_list:
    database_option [database_option ...]
database_option:
    [DEFAULT] {CHARACTER SET | CHARSET} [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  |  LOCALITY = locality_option}
  | [{PARTITION_MODE|MODE} = partition_mode_option]
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
partition_mode_option:
     {'auto'|'partitioning'}
    |{'drds'|'sharding'}
```

参数说明

参数	说明
database_name	指定要修改属性的数据库名称。如果不指定，会对当前默认数据库进行修改。
CHARSET charset_name	指定要修改的字符集。
COLLATE collation_name	指定校对规则。
LOCALITY	创建数据库时指定该库的存储位置。

MODE	指定逻辑库所使用的分区模式，支持两种模式：
	• auto/partitioning：自动分区模式
	  ◦ 不手工指定分区定义时，自动按主键分区；
	  ◦ 全库只能使用新的CREATE TABLE语法（如partition byhash/Range/List）进行建表。
	• drds/sharding：DRDS模式，全库只能使用DRDS模式的CREATE TABLE语法（如dbpartition by/tbpartition by）进行建表。
	注意
	• 创建数据库不指定mode，默认是“drds/sharding”；
	• 数据库的分区模式不可更改，创建数据库时一旦指定，不能变更。

示例

- 创建数据库test，并指定字符集为UTF8。

```
create database test default CHARACTER SET=UTF8 mode=auto;
Query OK, 1 row affected (3.36 sec)
show create database test;
+----------+--------------------------------------------+
| DATABASE | CREATE DATABASE                            |
+----------+--------------------------------------------+
| test     | CREATE DATABASE `test` /* MODE = 'auto' */ |
+----------+--------------------------------------------+
1 row in set (0.06 sec)
```

- 在实例中创建一个数据库，并通过以下命令指定其存储位置为*polardbx-storage-0-master*节点。

```
CREATE DATABASE db1 LOCALITY='dn=polardbx-storage-0-master';
```

> 说明
> - 如果在创建数据库时未指定数据库的存储位置，系统将默认在所有存储节点中均匀分布数据库。
> - 数据库中分表的存储位置与该库的存储位置保持一致，以实现分表上的数据隔离。

创建成功后，您可以通过以下语句查看数据库的存储位置信息。

```
SHOW CREATE DATABASE db1;
```

返回结果如下所示：

```
+----------+-----------------------------------------------------------------------------+
| DATABASE | CREATE DATABASE                                                             |
+----------+-----------------------------------------------------------------------------+
| db1      | CREATE DATABASE `db1` /* LOCALITY = "dn=polardbx-storage-0-master" */  |
+----------+-----------------------------------------------------------------------------+
1 row in set
```

8.7.2.2. DROP DATABASE

DROP DATABASE语句用于删除数据库。

语法

```
drop_database_stmt:
    DROP DATABASE [IF EXISTS] database_name;
```

参数说明

参数	说明
IF EXISTS	用于防止当数据库不存在时发生错误。
database_name	指定待删除的数据库名。

示例

删除数据库test。

```
mysql>drop database test;
Query OK, 0 rows affected (0.03 sec)
mysql>drop database if exists test;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

8.7.2.3. CREATE TABLEGROUP

CREATE TABLEGROUP语句用于在数据库中创建一个表组。

定义

表组是表的属性，在PolarDB-X中允许用户创建表组，然后将多个表和表组关联起来，同一个表组里的表必须是采用相同的分区策略，并且同一个表组的所有分区表，在涉及分区变更操作时，必须保持同步。例如要对表组的一个表的某个分区进行分裂，那么需要通过分裂整个表组某个分区来实现这个目的。

语法

```
create_tablegroup_stmt:
    CREATE tablegroup [IF NOT EXISTS] tablegroup_name
```

示例

创建一个名为test_tg的表组。

```
create tablegroup test_tg;
```

创建好表组后可以通过 `show tablegroup` 命令查看表组的详细信息。例如，查看名为test_tg的表组信息。

```
show tablegroup where TABLE_GROUP_NAME='test_tg'
```

8.7.2.4. DROP TABLEGROUP

DROP TABLEGROUP语句用于删除表组，仅当表组为空时（也就是表组不包括任何的表）才允许删除表组。

语法

```
drop_tablegroup_stmt:
    DROP tablegroup [IF EXISTS] tablegroup_name
```

示例

删除一个名为test_tg的表组。

```
drop tablegroup test_tg;
```

8.7.2.5. ALTER TABLEGROUP

本文介绍了ALTER TABLEGROUP语句的用法。

分区组分裂

Range分区

- 支持将分区分裂成指定范围的多个分区。

  语法：

  ```
  ALTER TABLEGROUP tg_name split PARTITION identifier INTO
   (partition_definition,partition_definition, [, partition_definition] ...);
   partition_definition:
      PARTITION partition_name
          [VALUES
              {LESS THAN {(value_list) | MAXVALUE}
              |
              IN (value_list)}]
  ```

  示例：

  假设表组tg1中的p1的定义为partition p1 values less than(20)，可以通过以下语句将p1分裂成p10，p11，p12。

  ```
  ALTER TABLEGROUP tg1 split PARTITION p1 into
  (partition p10 values less than (8),
  partition p11 values less than(15),
  partition p12 values less than(20))
  ```

- 支持将一个Range在其范围内按任意点一分为二。

  语法：

  ```
  ALTER TABLEGROUP identifier split PARTITION identifier at(number) into
   (PARTITION identifier,
    PARTITION identifier);
  ```

示例：

假设表组tg1中的p1的定义为partition p1 values less than(20)，若需要将p1分区在9这个点一分为2，即p11：[0,9)，p12：[9,20)，SQL如下：

```
ALTER TABLEGROUP tg1 split PARTITION p1 at(9) into (partition p11, partition p12)
```

该SQL也等价于：

```
ALTER TABLEGROUP tg1 split PARTITION p1 into
(partition p11 values less than (9),
partition p12 values less than(20));
```

List分区

- 语法：

```
ALTER TABLEGROUP tg_name split PARTITION partition_name into
 (partition_definition,partition_definition, [, partition_definition] ...);
```

- 示例：

  假设表组tg1中的p1的定义为partition p1 values in(1, 2, 3, 4, 5, 6)，可以通过以下语法将p1分裂成p10，p11，p12。

```
ALTER TABLEGROUP tg1 split PARTITION p1 into
(partition p10 values in (1,3,5),
partition p11 values in (2,4),
partition p12 values in (6))
```

Hash分区

- 支持用分区键的Hash range的中值作为分裂点，将一个分区切分为两个分区。

  语法：

```
ALTER TABLEGROUP identifier split PARTITION identifier;
```

  示例：

```
ALTER TABLEGROUP tg1 split PARTITION p1;
```

- 支持将热点值提取到单独的分区。

```
ALTER TABLEGROUP tgName extract to PARTITION [newPartName] by hot value(xxx)
```

key分区

- 支持将热点值提取到单独的分区。

  语法：

```
ALTER TABLEGROUP tgName extract to PARTITION [newPartName] by hot value(xxx[,..,xxx]);
```

  示例：

  假设拆分键为(a,b),key拆分方式，将拆分键值为(a=88,b=10)的数据，提取到一个单独的新分区p20中。

```
ALTER TABLEGROUP tg1 extract to PARTITION p20 by hot value(88,10);
```

- 支持将热点散列到多个分区。

  语法：

  N为将热点值散列为N个分区。

```
ALTER TABLEGROUP tgName split into [newPartNamePrefix] PARTITIONS N by hot value(xxx[,..,xxx]);
```

  假设拆分键为(a，b)，key拆分方式。假设a=88对应的数据很多，将a=88的数据散列到5个分区的语法为 `ALTER TABLEGROUP tgName split into hot88_ PARTITIONS 5 by hot value(88)` ，这个语法会将a=88的数据散列到5个新的分区，hot88_1，hot88_2，hot88_3，hot88_4，hot88_5，这5个分区是按照b的hash空间五等分切分的。

- 支持表级别的操作，N表示将热点值散列为N个分区。

```
ALTER tableName split into [newPartNamePrefix] PARTITIONS N by hot value(xxx[,..,xxx]);
```

> **说明** 表级别的操作要求表所在的TABLEGROUP只有当前表一个表，没有其他表。

分区合并

语法：

将多个分区（两个或者两个以上）合并成一个分区。

```
ALTER TABLEGROUP tg_name MERGE PARTITIONS partition_name,...,partition_name TO partition_name;
```

示例：

```
ALTER TABLEGROUP tbl_tg MERGE PARTITIONS p2,p3 to p23;
```

分区迁移

语法：

将分区迁移到指定的DN节点。

```
ALTER TABLEGROUP tg_name MOVE PARTITIONS partition_name,...,partition_name TO dn_id;
```

示例：

```
ALTER TABLEGROUP tg_name MOVE PARTITIONS p2,p4 to 'dn-0' ;
```

增加分区

对于Range/List策略的分区组，支持增加分区。

```
ALTER TABLEGROUP tg_name ADD PARTITION (partition_definition [,partition_definition] ...)
```

删除分区

对于Range/List策略的分区组，支持删除分区。

```
ALTER TABLEGROUP tg_name DROP PARTITION partitiion_name [,partition_name] ...
```

重命名分区

```
ALTER TABLEGROUP tg_name RENAME PARTITION old_part_name to new_part_name[, old_part_name to new_part_name]
```

修改分区值

只支持对List/List column分区策略的分区修改分区值。

```
ALTER TABLEGROUP tg_name MODIFY partition_name add/drop values in (value_list)
```

8.7.2.6. CREATE TABLE

本文主要介绍使用DDL语句创建分区表的语法、子句、参数和基本方式。

注意事项

使用分区表语法之前，请务必确认创建当前的逻辑库时模式指定为自动分区模式（mode='auto'），非自动分区模式不允许使用分区表的建表语法。您也可以通过 `SHOW CREATE DATBASE db_name` 语法查看当前逻辑库的建表模式。示例如下：

```
CREATE DATABASE part_db mode='auto';
Query OK, 1 row affected (4.29 sec)
SHOW CREATE DATABASE part_db;
+----------+-----------------------------------------------+
| DATABASE | CREATE DATABASE                               |
+----------+-----------------------------------------------+
| part_db  | CREATE DATABASE `part_db` /* MODE = 'auto' */ |
+----------+-----------------------------------------------+
1 row in set (0.18 sec)
```

创建数据库的语法详情，请参见CREATE DATABASE。

语法

```
CREATE [PARTITION] TABLE [IF NOT EXISTS] tbl_name
    (create_definition, ...)
    [table_options]
    [table_partition_definition]
    [local_partition_definition]
create_definition:
    col_name column_definition
  | mysql_create_definition
  | [UNIQUE] GLOBAL INDEX index_name [index_type] (index_sharding_col_name,...)
      [global_secondary_index_option]
      [index_option] ...
index_sharding_col_name:
    col_name [(length)] [ASC | DESC]
index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
index_type:
    USING {BTREE | HASH}
# 全局二级索引相关
global_secondary_index_option:
    [COVERING (col_name,...)]
    [partition_options]
table_options:
    table_option [[,] table_option] ...
table_option: {
# 指定tablegroup
    TABLEGROUP [=] value,...,}
# 分区表类型定义
table_partition_definition:
        single
  |    broadcast
  | partition_options
# 分区策略定义
partition_options:
    PARTITION BY
          HASH({column_name | partition_func(column_name)})
        | KEY(column_list)
        | RANGE{({column_name | partition_func(column_name)})
        | RANGE COLUMNS(column_list)}
        | LIST{({column_name | partition_func(column_name)})
        | LIST COLUMNS(column_list)} }
    partition_list_spec
# 分区函数定义
partition_func:
    YEAR
  | TO_DAYS
  | TO_SECOND
```

```
  | UNIX_TIMESTAMP
  | MONTH
# 三种分区类型定义
partition_list_spec:
    hash_partition_list
  | range_partition_list
  | list_partition_list
# Hash/Key分区定义
hash_partition_list:
    PARTITIONS partition_count
# Range/Range Columns分区定义
range_partition_list:
    range_partition [, range_partition ...]
range_partition:
    PARTITION partition_name VALUES LESS THAN {(expr | value_list)} [partition_spec_options]
# List/List Columns分区定义
list_partition_list:
    list_partition [, list_partition ...]
list_partition:
    PARTITION partition_name VALUES IN (value_list) [partition_spec_options]
partition_spec_options:
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string']
        [{CHARSET | CHARACTER SET} [=] charset]
        [COLLATE [=] collation]
        [TABLEGROUP [=] table_group_id]
        [LOCALITY [=] locality_option]
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
local_partition_definition:
    LOCAL PARTITION BY RANGE (column_name)
    [STARTWITH 'yyyy-MM-dd']
    INTERVAL interval_count [YEAR|MONTH|DAY]
    [EXPIRE AFTER expire_after_count]
    [PRE ALLOCATE pre_allocate_count]
    [PIVOTDATE pivotdate_func]
    [DISABLE SCHEDULE]
pivotdate_func:
    NOW()
  | DATE_ADD(...)
  | DATE_SUB(...)
```

> **说明** PolarDB-X DDL语法基于MySQL语法，以上主要列出了差异部分，详细语法请参见MySQL 文档。

名词解释

- 分区键：分区表中用于进行水平切分的一个列或多个列。
- 分区列：水平切分后，参与分区路由及计算的列，它通常是分区键的一部分，一个分区键可以含有一个或多个分区列。
- 向量分区键：由一个或多个分区列组成的分区键。
- 单列分区键：由一个分区列组成的分区键。
- 前缀分区列：若一个向量分区键由N（N>1)个分区列组成，它的前K（1<=K<=N）个分区列便组成这个向量分区键的前缀分区列。
- 分区函数：将分区列作为一个函数的输入参数，并将该函数的输出结果作为原始值参与路由计算，该函数被称为分区函数。
- 分区裁剪：根据分区定义及查询条件，最大限度地过滤不需要扫描的分区的查询优化手段。
- 热点分裂：当向量分区键的前缀分区列存在访问热点或分布不均衡时，允许使用下一个分区列对热点分区进行分裂，以达到负载均衡效果。

默认自动分区

- 建表SQL在不指定分区键的情况下，PolarDB-X默认会按主键（如果表没有指定主键，则使用隐式主键）并使用KEY分区进行默认分区。

  默认分区的分区数目=实例创建时逻辑节点数×8。例如，PolarDB-X实例创建时，逻辑节点是2，那默认分区数目就是16。
- 除了主表默认会按主键自动分区，主表中所有索引也会默认以索引列与主键列作为分区键并进行自动分区。

如下示例是标准的MySQL建表语法，主键是id，索引列为name：

```
CREATE TABLE auto_part_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 primary key(id),
 index idx_name (name)
);
```

若使用 `SHOW CREATE TABLE` 语句查询该建表语句，显示标准的MySQL建表语法，自动隐藏所有分区信息：

```
show create table auto_part_tbl;
+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| TABLE         | CREATE TABLE
|
+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| auto_part_tbl | CREATE TABLE `auto_part_tbl` (
    `id` bigint(20) NOT NULL AUTO_INCREMENT,
    `bid` int(11) DEFAULT NULL,
    `name` varchar(30) DEFAULT NULL,
    PRIMARY KEY (`id`),
    INDEX `idx_name` (`name`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8 |
+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.06 sec)
```

若使用 `SHOW FULL CREATE TABLE` 查询该建表语句，则会显示上述主表及其索引表的所有分区信息：

```
show full create table auto_part_tbl;
+---------------+-----------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------+
| TABLE         | CREATE TABLE
|
+---------------+-----------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------+
| auto_part_tbl | CREATE PARTITION TABLE `auto_part_tbl` (
    `id` bigint(20) NOT NULL AUTO_INCREMENT,
    `bid` int(11) DEFAULT NULL,
    `name` varchar(30) DEFAULT NULL,
    PRIMARY KEY (`id`),
    GLOBAL INDEX /* idx_name_$a870 */ `idx_name` (`name`) PARTITION BY KEY (`name`, `id`) PARTITIONS 16,
    LOCAL KEY `_local_idx_name` (`name`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8
PARTITION BY KEY(`id`)
PARTITIONS 16
/* tablegroup = `tg108` */ |
+---------------+-----------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------+
1 row in set (0.03 sec)
```

从返回信息中可以看到：

- 主表auto_part_tbl默认按id进行了KEY分区，分区数是16；
- 主表的索引idx_name默认使用了全局索引，全局索引的分区键是 `name`，`id`，分区数也是16。

单表与广播表

PolarDB-X允许创建表时通过指定关键字SINGLE来创建单表（不进行任何分区的表），示例如下：

```
CREATE TABLE single_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 primary key(id)
) SINGLE;
```

PolarDB-X允许创建表时通过指定关键字BROADCAST来创建广播表（该表将在所有DN节点上有一份数据完全相同的拷贝），示例如下：

```
CREATE TABLE broadcast_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 primary key(id)
) BROADCAST;
```

分区类型介绍

PolarDB-X允许创建表时通过指定分区子句的语法，来创建符合业务需求的分区表。PolarDB-X支持三大类型的分区：

- **Hash类型**：基于用户指定的分区列或分区函数表达式的值，使用内置的一致性哈希算法计算其哈希值并进行分区路由的策略。按是否支持使用分区函数表达式或使用多个分区列作为分区键，Hash分区策略又可以细分为**Key分区**和**Hash分区**两种分区策略。
- **Range类型**：基于用户指定的分区列或分区函数表达式的值，通过比较计算来确定数据位于哪些预定义分区的范围并进行分

区路由的策略。按是否支持使用分区函数表达式或使用多个分区列作为分区键，Range分区策略又可以细分为**Range Columns分区**和**Range分区**两种分区策略。

- **List类型**：与Range分区策略类似，基于用户指定的分区列或分区函数表达式的值，通过比较计算来确定数据位于哪些预定义分区的取值集合并进行分区路由的策略。按是否多个分区列作为分区键以及其使用方式的不同，List类型也分为**List Columns分区**和**List分区**两种分区策略。

Hash类型

PolarDB-X的Hash类型分区可细分为Hash分区与Key分区两种类型。Hash分区与Key分区是原生MySQL的标准分区语法之一，PolarDB-X为提供灵活强大的分区管理能力（例如分裂、合并与迁移等）以及考虑支持向量分区键下热点散裂，PolarDB-X不仅在语法上尽量兼容了MySQL的Hash分区与Key分区的建表语法（PolarDB-X仅兼容了语法，分区路由的实现与MySQL并非一致），但是对于Key分区与Hash分区的路由行为重新进行了定义。Key分区与Hash分区使用区别如下表所示：

Key分区策略与Hash分区策略对比

分区策略	分区键支持	是否支持分区函数	语法示例	特点与限制	路由描述（点查）
Key（默认的分区策略）	单列分区键	否	PARTITION BY KEY(c1)	• 不支持热点分裂； • 最多有1个分区列参与路由计算；	1. 按c1列的值，使用一致性哈希算法计算其哈希c1_hash； 2. 按哈希值 c1_hash进行分区路由。
	向量分区键	否	PARTITION BY KEY(c1,c2,...,cn)	• 支持热点分裂； • 建表后默认仅第1个分区列c1实际参与路由计算，热点分裂时可以使用c2,...,cn等更多的分区列； • 最多允许n个分区列同时参与路由计算； • 分区列数目n默认不能超过5个。	1. 将(c1,c2,...,cn)这N个列的值组成一组分区键向量，并使用一致性哈希算法为向量的各个列的值计算其哈希值，从而将一个分区键向量映射成一个哈希值向量 (c1_hash,c2_hash,...,cn_hash) ； 2. 将哈希值向量 (c1_hash,c2_hash,...,cn_hash) 按range路由并找到目标分区。

Hash	单列分区键	否	PARTITION BY HASH(c1)	• 不支持热点分裂；
				• 一个分区列参与路由
				• 分区函数目前仅支持5个
				  ◦ YEAR
				  ◦ TO_DAYS
				  ◦ TO_SECOND
				  ◦ MONTH
				  ◦ UNIX_TIMESTAMP
					PARTITION BY HASH(c1)与PARTITION BY KEY(c1)完全等同 ，其路由算法与PARTITION BY KEY(c1)完全一致。

		是	PARTITION BY HASH(YEAR(c1))		1. 按c1列的值，使用YEAR分区函数计算其对应的年份 year； 2. 将年份year使用一致性哈希算法计算其哈希值year_hash； 3. 按哈希值year_hash进行分区路由。
	向量分区键	否	PARTITIONBY HASH(c1,c2,...,cn)	• 不支持热点分裂； • 建表后，n个分区列将同时与路由； • 分区列数目n默认不能超过5个。	1. 将(c1,c2,...,cn)这N个分区列的值组成一组向量，并使用一致性哈希算法，计算该向量对应哈希值hashVal； 2. 按哈希hashVal进行分区路由。

示例1-1：Key分区

Key分区也是PolarDB-X的默认分区方式。 Key分区支持向量分区键。例如，用户想按用户名字name列与用户id两个列作为分区键进行分区，预建分区数目可以指定为8，可以使用以下的语法建表：

```
CREATE TABLE key_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 birthday datetime not null,
 primary key(id)
)
PARTITION BY KEY(name, id)
PARTITIONS 8;
```

根据KEY分区的特点，如上所示向量分区键的分区表，当它进行路由计算时，默认只使用向量分区键的第1个分区列（name）进行路由。因此，业务查询SQL的WHERE条件表达式中，只需要含有第1个分区列的等值条件，即可命中分区裁剪的优化，如下所示：

```
##（命中分区裁剪，只需扫描一个分区）
SELECT id from key_tbl where name='Jack';
```

若第1个分区列name存在分布不均衡或者出现数据热点，您也可以通过分区分裂操作（详细请参考ALTER TABLEGROUP），并使用下一个分区列（如id）进行分区分裂，从而解决数据不均衡的问题。

如果一个由N个分区列组成的向量分区键，若它实际路由使用到的分区列数目是前K个（1<=K<=N），则查询SQL的WHERE条件表达式只需要包含由这前K个分区列组成的前缀分区裂即可命中分区裁剪。

示例1-2：Hash分区

如果您想按用户id列作为分区键进行水平分区，可以使用Hash分区进行建表，分区数目可以指定为8，建表语法示例如下：

```
CREATE TABLE hash_tbl(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 birthday datetime not null,
 primary key(id)
)
partition by hash(id)
partitions 8;
```

Hash分区支持使用分区函数表达式（例如YEAR/TO_DAYS/...等）来将时间类型转换成整数类型。因此，如果您想按用户出生日期birthday列进行分区，并且预建的Hash分区数目是8，也可以使用如下语句建表：

```
CREATE TABLE hash_tbl_todays(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 birthday datetime not null,
 primary key(id)
)
PARTITION BY HASH(TO_DAYS(birthday))
PARTITIONS 8;
```

目前PolarDB-X的分区函数仅支持以下的函数列表：

- YEAR
- TO_DAYS
- TO_SECOND
- MONTH
- UNIX_TIMESTAMP

因此，使用分区函数的分区键的类型必须是时间类型（DATE/DATETIME/TIMESTAMP），其它类型不支持使用分区函数。

示例1-3：Hash分区扩展

PolarDB-X对Hash分区的语法进行扩展，让Hash分区支持使用向量分区键 （原生MySQL的标准分区语法，可使用如下语句：

```
CREATE TABLE hash_tbl2(
 id bigint not null auto_increment,
 bid int,
 name varchar(30),
 birthday datetime not null,
 primary key(id)
)
PARTITION BY HASH(name, birthday)
PARTITIONS 8;
```

与Key分区不同， Hash分区使用向量分区键，这样的分区表在分区路由计算时，是所有分区列同时参与哈希值计算与路由计算，所以，它会要求查询SQL的WHERE条件表达式必须要含包所有的分区列的等值条件才能命中分区裁剪，如下示例的SQL1可以命中hash_tbl2的分区裁剪，而SQL2则不能命中hash_tbl2的分区裁剪：

```
## SQL1（命中分区裁剪，只扫描一个分区）：
SELECT id from hash_tbl2 where name='Jack' and birthday='1990-11-11';
##SQL2（没命中分区裁剪，全分区扫描）：
SELECT id from hash_tbl2 where name='Jack';
```

Hash分区由于一开始就直接使用所有的分区键进行哈希值计算，所以理论上，它比使用向量分区键的Key分区会打散得更均衡，但它不再支持使用下一个列（因为已经没有列可用）进行热点散列。

相关限制

- 数据类型限制
  - 整数类型： BIGINT/BIGINT UNSINGED/INT UNSINGED/INT/MEDIUMINT/MEDIUMINT UNSINGED/SMALLINT/SMALLINT UNSINGED/TINYINT/TINYINT UNSINGED;
  - 时间类型：DATETIME/DATE/TIMESTAMP;
  - 字符串类型：CHAR/VARCHAR。
- 语法限制
  - Hash分区的单列分区键支持使用分区函数，且分区键类型必须为时间类型;
  - Hash分区的向量分区键不允许使用分区函数，不支持热点分列;
  - 默认最大分区数目不允许超过8192个;
  - 默认最大分区列数目不允许超过5个。

数据均匀性

- Key分区与Hash分区内置的一致性Hash散列算法是经过业界广泛测试的、冲突概率低且性能良好的散列算法 MurmurHash3。
- 基于MurmurHash3的特性，一般情况下，当分区键不同取值的数目N大于3000时，Key分区与Hash分区的数据分布才会相对均衡，且N的值越大，数据分布也将越均衡。

Range类型

PolarDB-X的Range类型分区，可细分为Range分区与Range Columns分区两种。同样， Range分区和Range Columns分区属于原生MySQL的标准分区语法。这两种分区的使用区别如下表所示。

Range Columns分区策略与Range分区策略对比

分区策略	分区键支持	是否支持分区函数	语法示例	特点与限制	路由描述（点查）
Range Columns	单列分区键&向量分区键	否	PARTITION BY RANGE COLUMNS (c1,c2,...,cn) (PARTITION p1 VALUES LESS THAN (1,10,...,1000), PARTITION p2 VALUES LESS THAN (2,20,...,2000) ...)	支持热点分裂（例如c1有热点值88，可以使用c2进行分区分裂解决热点）。	1. 将(c1,c2,...,cn)这N个列的值组成一组分区键向量; 2. 根据这个分区键向量(c1,c2,...,cn)按二分查找算法判断它属于哪个预定义分区，并最终路由到目标分区。

分区策略	分区键支持	是否支持分区函数	语法示例	特点与限制	路由描述（点查）

Range	单列分区键	是	PARTITION BY RANGE(YEAR(c1)) (PARTITION p1 VALUES LESS THAN (2019), PARTITION p2 VALUES LESS THAN (2021) ...)	• 不支持热点分裂；
				• 分区函数目前仅支持5个
				  ◦ YEAR
				  ◦ TO_DAYS
				  ◦ TO_SECOND
				  ◦ MONTH
				  ◦ UNIX_TIMESTAMP
					1. 按c1列的值，使用YEAR分区函数计算其对应的年份year；
					2. 年份year按二分查找算法判断它属于哪个预定义分区，并路由到目标分区。

示例2-1：Range Columns分区

Range Columns分区支持使用向量分区键，但它不支持使用分区函数。例如，业务可以按订单ID与订单日期进行Range分区，可以使用以下的建表语法：

```
CREATE TABLE orders(
 order_id int,
 order_time datetime not null)
PARTITION BY range columns(order_id,order_time)
(
  PARTITION p1 VALUES LESS THAN (10000,'2021-01-01'),
  PARTITION p2 VALUES LESS THAN (20000,'2021-01-01'),
  PARTITION p3 VALUES LESS THAN (30000,'2021-01-01'),
  PARTITION p4 VALUES LESS THAN (40000,'2021-01-01'),
  PARTITION p5 VALUES LESS THAN (50000,'2021-01-01'),
  PARTITION p6 VALUES LESS THAN (MAXVALUE,MAXVALUE)
);
```

> **注意** 目前Range Columns分区不支持使用TIMESTAMP/TIME等与时区相关的类型作为分区键。

示例2-2：Range分区

Range分区仅支持单列分区健，但对于时间类型的分区列，它支持使用分区函数（例如YEAR/TO_DAYS/TO_SECOND/MONTH等）来将时间类型转换成整数类型。

> **注意** Range分区是不支持直接使用字符串类型作为分区列。

例如，业务想按订单的日期order_time列进行Range分区，并且每个季度一个分区，建表语法如下所示：

```
CREATE TABLE orders_todays(
 id int,
 order_time datetime not null)
PARTITION BY RANGE(to_days(order_time))
(
  PARTITION p1 VALUES LESS THAN (to_days('2021-01-01')),
  PARTITION p2 VALUES LESS THAN (to_days('2021-04-01')),
  PARTITION p3 VALUES LESS THAN (to_days('2021-07-01')),
  PARTITION p4 VALUES LESS THAN (to_days('2021-10-01')),
  PARTITION p5 VALUES LESS THAN (to_days('2022-01-01')),
  PARTITION p6 VALUES LESS THAN (MAXVALUE)
);
```

> **注意** 目前Range分区仅支持使用整数类型作为分区键，且分区键仅支持1个分区列。

相关限制

- 数据类型限制

- 整数类型： BIGINT/BIGINT UNSINGEDINT/INT/INT UNSINGED/MEDIUMINT/MEDIUMINT UNSINGED/SMALLINT/SMALLINT UNSINGED/TINYINT/TINYINT UNSINGED;
- 时间类型：DATETIME/DATE;
- 字符串类型：CHAR/VARCHAR。

- 语法限制
  - Range Columns分区与Range分区都不支持使用NULL值作为边界值;
  - Range Columns分区目前不支的使用TIMESTAMP类型;
  - Range分区仅支持整数类型，若分区键使用TIMESTAMP类型，必须配套使用分区函数UNIX_TIMESTAMP确保时区一致;
  - Range的分区不支持热点分列;
  - 查询时，NULL值查询会被当作最小值进行分区路由;
  - 默认最大分区数目不允许超过8192个;
  - 默认最大分区列数目不允许超过5个。

List类型

与Range类型类似，PolarDB-X将List分区策略进一步细分为List分区与List Columns分区两种类型。List分区与List Columns分区属于原生MySQL的标准分区语法。List Columns分区与List分区的使用区别如下表所示：

List Columns分区策略与List分区策略对比

分区策略	分区键支持	是否支持分区函数	语法示例	特点与限制	路由描述（点查）
List Columns	单列分区键&向量分区键	否	PARTITION BY LIST COLUMNS (c1,c2,...,cn) (PARTITION p1 VALUES IN ((1,10,...,1000), (2,20,...,2000)), PARTITION p2 VALUES IN ((3,30,...,3000), (3,30,...,3000)), ...)	不支持热点分裂	1. 将(c1,c2,...,cn)这N个列的值组成一组分区键向量; 2. 根据这个分区键向量(c1,c2,...,cn)按二分查找算法判断它属于哪个预定义分区，并最终路由到目标分区。
List	单列分区键	是	PARTITION BY LIST(YEAR(c1)) (PARTITION p1 VALUES IN (2018,2019), PARTITION p2 VALUES IN (2020,2021) ...)		不支持热点分裂

示例3-1：List Columns分区

List Columns分区支持使用向量分区键。例如，业务可以按订单的国家country与城市city进行List Columns分区， 建表语法如下所示：

```
CREATE TABLE orders_region(
 id int,
 country varchar(64),
 city varchar(64),
 order_time datetime not null)
PARTITION BY LIST COLUMNS(country,city)
(
  PARTITION p1 VALUES IN (('China','Hangzhou'), ('China','Beijing')),
  PARTITION p2 VALUES IN (('United States','NewYork'),('United States','Chicago')),
  PARTITION p3 VALUES IN (('Russian','Moscow'))
);
```

> 注意 目前List Columns分区不支持使用TIMESTAMP/TIME等与时区相关的类型作为分区键。

示例3-2：List分区

List 分区只支持单列分区健，但对于时间类型的分区列，它支持分区函数表达式（例如YEAR/TO_DAYS/TO_SECONDS/MONTH等）来将时间类型转换成整数类型。

例如，业务想按订单日期order_time的年份进行List分区，则可以使用以下语法建表：

```
CREATE TABLE orders_years(
 id int,
 country varchar(64),
 city varchar(64),
 order_time datetime not null)
PARTITION BY LIST(YEAR(order_time))
(
  PARTITION p1 VALUES IN (1990,1991,1992,1993,1994,1995,1996,1997,1998,1999),
  PARTITION p2 VALUES IN (2000,2001,2002,2003,2004,2005,2006,2007,2008,2009),
  PARTITION p3 VALUES IN (2010,2011,2012,2013,2014,2015,2016,2017,2018,2019)
);
```

> 注意　目前List仅支持使用整数类型作为分区键，另外要值得注意的是，也不支持直接使用字符串类型作为分区列。

相关限制

- 数据类型限制
  - 整数类型： BIGINT/BIGINT UNSINGEDINT/INT/INT UNSINGED/MEDIUMINT/MEDIUMINT UNSINGED/SMALLINT/SMALLINT UNSINGED/TINYINT/TINYINT UNSINGED;
  - 时间类型：DATETIME/DATE;
  - 字符串类型：CHAR/VARCHAR。
- 语法限制
  - List Columns分区目前还不支的使用TIMESTAMP类型;
  - List分区仅支持整数类型;
  - List Columns分区与List分区均不支持热点分裂;
  - 默认最大分区数目不允许超过8192个;
  - 默认最大分区列数目不允许超过5个。

数据类型说明

各分区策略分区列数据类型的支持情况

数据类型		Hash类型			Range类型		List类型	
		HASH		Key	Range	Range Columns	List	List Columns
		单区分列	多区分列					
数值类型	TINYINT	支持	支持	支持	支持	支持	支持	支持
	TINYINT UNSIGNED	支持	支持	支持	支持	支持	支持	支持
	SMALLINT	支持	支持	支持	支持	支持	支持	支持
	SMALLINT UNSIGNED	支持	支持	支持	支持	支持	支持	支持
	MEDIUMINT	支持	支持	支持	支持	支持	支持	支持
	MEDIUMINT UNSIGNED	支持	支持	支持	支持	支持	支持	支持
	INT	支持	支持	支持	支持	支持	支持	支持

数据类型		Hash类型			Range类型		List类型	
		HASH		Key	Range	Range Columns	List	List Columns
		单区分列	多区分列					
	INT UNSIGNED	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持
	BIGINT	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持
	BIGINT UNSIGNED	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持	✅支持
时间类型	DATE	✅（该类型的分区列支持使用分区函数）	✅支持	✅支持	✅（该类型的分区列支持使用分区函数）	✅支持	✅（该类型的分区列支持使用分区函数）	✅支持
	DATETIME	✅（该类型的分区列支持使用分区函数）	✅支持	✅支持	✅（该类型的分区列支持使用分区函数）	✅支持	✅（该类型的分区列支持使用分区函数）	✅支持
	TIMESTAMP	✅（该类型的分区列支持使用分区函数）	✅支持	✅支持	❌不支持	❌不支持	❌不支持	❌不支持
字符串类型	CHAR	✅支持	✅支持	✅支持	❌不支持	✅支持	❌不支持	✅支持
	VARCHAR	✅支持	✅支持	✅支持	❌不支持	✅支持	❌不支持	✅支持

关于分区列的数据类型与路由计算的说明

分区表的路由计算是直接依赖于分区列的数据类型的，特别是 Key分区与Hash分区。因此，使用不同数据类型的分区列，其哈希值算法或比较算法（比如，大小写是否敏感）的实现不一样，会产生不一样的路由行为（MySQL的分区路由算法也是类型强相关的）。

如下所示，假如tbl_int表是分区列，类型是int且分区数是1024, 而tbl_bigint分区列类型是bigint且分区数目也是1024。它们虽然都是整数类型，但是由于数据类型不一样，对于同一个查询值（12345678）的路由结果也不一样：

```
show create table tbl_int;
+---------+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| TABLE   | CREATE TABLE
|
+---------+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| tbl_int | CREATE TABLE `tbl_int` (
 `a` int(11) NOT NULL,
 KEY `auto_shard_key_a` USING BTREE (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`a`)
PARTITIONS 1024 |
+---------+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
1 row in set (0.02 sec)
show create table tbl_bigint;
+------------+-------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------+
| TABLE      | CREATE TABLE
|
+------------+-------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------+
| tbl_bigint | CREATE TABLE `tbl_bigint` (
 `a` bigint(20) NOT NULL,
 KEY `auto_shard_key_a` USING BTREE (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`a`)
PARTITIONS 1024 |
+------------+-------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------+
1 row in set (0.10 sec)mysql> create table if not exists tbl_bigint(a bigint not null)
    -> partition by key(a) partitions 1024;
Query OK, 0 rows affected (28.41 sec)
explain select * from tbl_int where a=12345678;
+-----------------------------------------------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN                                                                                           |
+-----------------------------------------------------------------------------------------------------------------+
| LogicalView(tables="tbl_int[p260]", sql="SELECT `a` FROM `tbl_int` AS `tbl_int` WHERE (`a` = ?)") |
| HitCache:false                                                                                                  |
| Source:PLAN_CACHE                                                                                               |
| TemplateId: c90af636                                                                                            |
+-----------------------------------------------------------------------------------------------------------------+
4 rows in set (0.45 sec)
explain select * from tbl_bigint where a=12345678;
+-----------------------------------------------------------------------------------------------------------------
-+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------------
-+
| LogicalView(tables="tbl_bigint[p477]", sql="SELECT `a` FROM `tbl_bigint` AS `tbl_bigint` WHERE (`a` = ?)")
|
| HitCache:false
|
| Source:PLAN_CACHE
|
| TemplateId: 9b2fa47c
|
+-----------------------------------------------------------------------------------------------------------------
-+
4 rows in set (0.02 sec)
```

分区列大小写、字符集及校验集的说明

> **说明** 分区列的字符集（charset）及校验集（collation）对分区表的路由算法会产生直接影响。比如，指定分区路由是否需要忽略大小写。如果分区表的校验集是区分大小写的，则分区路由在哈希与比较过程中便会区分大小写；如果分区表的校验集不区分大小写，则分区路由在哈希与比较过程中不会区分大小写。默认情况下，字符类型的分区列会使用字符集utf8及不区分区大小写的校验集 utf8_general_ci。

示例1

如果用户需要让分区表在路由时区分分区列的大小写，在建表时将分区表的校验集设置为区分大小写的校验集即可（如utf8_bin）。如下所示，分区表tbl_varchar_cs的分区表采用了CHARACTER SET utf8 COLLATE utf8_bin，所以对于大小写不同的两个字符串 'AbcD' 与 'abcd'，分区表会将它们路由不同的分区：

```
show create table tbl_varchar_cs;
+---------------+---------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
| TABLE         | CREATE TABLE
|
+---------------+---------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
| tbl_varchar_cs | CREATE TABLE `tbl_varchar_cs` (
 `a` varchar(64) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
 KEY `auto_shard_key_a` USING BTREE (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8
PARTITION BY KEY(`a`)
PARTITIONS 64 |
+---------------+---------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-----------------+
1 row in set (0.07 sec)
explain select a from tbl_varchar_cs where a in ('AbcD');
+-------------------------------------------------------------------------------------------------
--------------+
| LOGICAL EXECUTIONPLAN
|
+-------------------------------------------------------------------------------------------------
--------------+
| LogicalView(tables="tbl_varchar_cs[p29]", sql="SELECT `a` FROM `tbl_varchar_cs` AS `tbl_varchar_cs` WHERE
(`a` IN(?))") |
| HitCache:false
|
| Source:PLAN_CACHE
|
| TemplateId: 2c49c244
|
+-------------------------------------------------------------------------------------------------
--------------+
4 rows in set (0.11 sec)
explain select a from tbl_varchar_cs where a in ('abcd');
+-------------------------------------------------------------------------------------------------
--------------+
| LOGICAL EXECUTIONPLAN
|
+-------------------------------------------------------------------------------------------------
--------------+
| LogicalView(tables="tbl_varchar_cs[p11]", sql="SELECT `a` FROM `tbl_varchar_cs` AS `tbl_varchar_cs` WHERE
(`a` IN(?))") |
| HitCache:true
|
| Source:PLAN_CACHE
|
| TemplateId: 2c49c244
|
+-------------------------------------------------------------------------------------------------
--------------+
4 rows in set (0.02 sec)
```

示例2

如果需要让分区表在路由时忽略分区列的大小写，在建表时将分区表的校验集设置为不区分大小写的校验集即可（如utf8_general_ci）。如下所示，分区表tbl_varchar_ci的分区表采用了CHARACTER SET utf8 COLLATE utf8_general_ci，所以对于大小写不同的两个字符串 'AbcD' 与 'abcd'，分区表则会将它们路由到同样的分区：

```
show create table tbl_varchar_ci;
+----------------+------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| TABLE          | CREATE TABLE
|
+----------------+------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| tbl_varchar_ci | CREATE TABLE `tbl_varchar_ci` (
 `a` varchar(64) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
 KEY `auto_shard_key_a` USING BTREE (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8
PARTITION BY KEY(`a`)
PARTITIONS 64 |
+----------------+------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
1 row in set (0.06 sec)
explain select a from tbl_varchar_ci where a in ('AbcD');
+-----------------------------------------------------------------------------------------------------------
-------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-------------+
| LogicalView(tables="tbl_varchar_ci[p4]", sql="SELECT `a` FROM `tbl_varchar_ci` AS `tbl_varchar_ci` WHERE (
`a` IN(?))") |
| HitCache:false
|
| Source:PLAN_CACHE
|
| TemplateId: 5c97178e
|
+-----------------------------------------------------------------------------------------------------------
-------------+
4 rows in set (0.15 sec)
explain select a from tbl_varchar_ci where a in ('abcd');
+-----------------------------------------------------------------------------------------------------------
-------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-------------+
| LogicalView(tables="tbl_varchar_ci[p4]", sql="SELECT `a` FROM `tbl_varchar_ci` AS `tbl_varchar_ci` WHERE (
`a` IN(?))") |
| HitCache:true
|
| Source:PLAN_CACHE
|
| TemplateId: 5c97178e
|
+-----------------------------------------------------------------------------------------------------------
-------------+
4 rows in set (0.02 sec)
```

分区列字符集与校验集的变更

由于分区表的路由算法与数据类型是相关的，如果分区列的字符集与校验集被修改，将会导致全表所有数据的重分布。所以请慎重地修改分区列的数据类型。

关于分区列的类型截断与类型转换的说明

分区列类型截断

SQL在查询或插入时，若指定分区列的常量表达式超过了分区列类型所能表达的范围，PolarDB-X会先产生类型截断，然后再使用类型截断后的值进行路由计算。

例如：tbl_smallint表分区列类型是smallint，smallint的正常取值范围是[-32768, 32767]。因此，如果insert的值超过smallint的范围（例如，12745678或-12345678），则会被截断类型的最大值或最小值（32767或-32768），如下所示 。

```
show create table tbl_smallint;
+--------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------+
| TABLE        | CREATE TABLE
|
+--------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------+
| tbl_smallint | CREATE TABLE `tbl_smallint` (
    `a` smallint(6) NOT NULL,
    KEY `auto_shard_key_a` USING BTREE (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`a`)
PARTITIONS 128 |
+--------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------+
1 row in set (0.06 sec)
set sql_mode='';
Query OK, 0 rows affected (0.00 sec)
insert into tbl_smallint values (12345678),(-12345678);
Query OK, 2 rows affected (0.07 sec)
select * from tbl_smallint;
+---------+
| a       |
+---------+
| -32768 |
|  32767 |
+---------+
2 rows in set (3.51 sec)
explain select * from tbl_smallint where a=12345678;
+-----------------------------------------------------------------------------------------------------------------
-------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------------
-------+
| LogicalView(tables="tbl_smallint[p117]", sql="SELECT `a` FROM `tbl_smallint` AS `tbl_smallint` WHERE (`a`
= ?)") |
| HitCache:false
|
| Source:PLAN_CACHE
|
| TemplateId: afb464d5
|
+-----------------------------------------------------------------------------------------------------------------
-------+
4 rows in set (0.16 sec)
explain select * from tbl_smallint where a=32767;
+-----------------------------------------------------------------------------------------------------------------
-------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------------
-------+
| LogicalView(tables="tbl_smallint[p117]", sql="SELECT `a` FROM `tbl_smallint` AS `tbl_smallint` WHERE (`a`
= ?)") |
| HitCache:true
|
| Source:PLAN_CACHE
```

```
|
| TemplateId: afb464d5
|
+-------------------------------------------------------------------------------------------------------------------
-------+
4 rows in set (0.03 sec)
```

同理，如果查询的常量值超过类型范围，也会按截断后的值进行路由。因此，对于tbl_smallint表，a=12345678与a=32767的分区路由结果是完全相同的。

分区列类型转换

SQL在查询或插入时，若指定分区列的常量表达式与分区列的类型不一致，PolarDB-X会先对常量表达式进行隐式类型转换，然后再使用类型转换后的值进行路由计算。但是，类型转换也有可能会出现转换失败的情况，例如，字符串 "abc" 无法转换为整型。

对于分区列出现类型转换及其失败的情况，PolarDB-X按DQL、DML、DDL会有不同的行为：

- DQL (特指WHERE条件表达式中的分区列的类型转换)
  - 类型转换成功：按类型转换后的值进行分区路由
  - 类型转换失败：该分区列条件会被直接忽略，走全表扫描
- DML（特指Insert/Replace）
  - 类型转换成功：按类型转换后的值进行分区路由
  - 类型转换成功：按类型转换后的值进行分区路由类型转换失败：直接报错，拒绝执行
- DDL（特指分区表相关的DDL，如建表、分裂等）
  - 类型转换成功：直接报错，拒绝执行，DDL不允许出现类型转换
  - 类型转换失败：直接报错，拒绝执行

参数说明

参数	说明
CHARSET | CHARACTER SET	指定表中列的默认字符集，可使用字符集如下： • utf8 • utf8mb4 • gbk
COLLATE	指定表中列的默认字符序，可使用字符序如下： • utf8_bin • utf8_general_ci • utf8_unicode_ci • gbk_bin • gbk_chinese_ci • utf8mb4_general_ci • utf8mb4__general_cs • utf8mb4_bin • utf8mb4_unicode_ci
TABLEGROUP	用于指定分区表所属于的表组。若不指定，会自动查找或创建与之分区方式完全一致的表组。
LOCALITY	用于指定分区表的所在DN节点。

与MySQL分区表的语法差异

差异点	MySQL	PolarDB-X
分区键包含主键	强制要求	不要求
Key分区	路由算法：按分区数目取模	路由算法：一致性哈希算法
Hash分区	1. 路由算法：按分区数目取模 2. 不支持多分区列 3. 支持LINEAR HASH分区 4. 单列分区键下，Hash(col)与Key(col)的路由算法不同	1. 路由算法：一致性哈希算法 2. 支持多分区列（扩展语法） 3. 不支持LINEAR HASH分区 4. 单列分区键下，Hash(col)与Key(col)的路由算法相同
分区函数	支持。PARTITION BY HASH(expr(col)) ..., expr可以是常见的计算表达式，如YEAR(col) + 1	有限地支持。PARTITION BY HASH(expr(col))，expr仅限于以下5个函数，且不允许表达式中出现其它计算操作（如+,-,*,/ 等）： • YEAR • TO_DAYS • TO_SECOND • MONTH • UNIX_TIMESTAMP
分区列类型	Key分区支持所有数据类型	Key分区共支持整数类型、时间类型与字符类型三类
分区列字符集	支持所有常见字符集	仅支持3种字符集 • utf8 • utf8mb4 • gbk
二级分区	支持	暂不支持

8.7.2.7. DROP TABLE

您可以使用DROP TABLE语法删除指定表。

语法

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

> **说明** 通过PolarDB-X删除表，系统会自动对相关物理表进行删除操作。

注意事项

- 当删除包含全局二级索引的表时，索引表会和主表同时被删除。
- 禁止单独删除索引表，如需删除索引，请使用 `ALTER TABLE DROP INDEX` 语句。
- `DROP TABLE` 语句一次执行仅支持删除一张表，如果同时删除多张表会收到报错提示。

示例

您可以通过如下语句删除 `user_log` 表：

```
DROP TABLE user_log;
```

8.7.2.8. ALTER TABLE

您可以通过ALTER TABLE语法改变表的结构，如增加列、增加索引、修改数据定义等。

注意事项

不支持通过ALTER TABLE语法修改拆分字段。

语法

> **说明** ALTER TABLE用于改变表的结构，如增加列、增加索引和修改数据定义。详细语法请参见MySQL修改表语法。

```
ALTER [ONLINE|OFFLINE] [IGNORE] TABLE tbl_name
    [alter_specification [, alter_specification] ...]
  [partition_options]
  [local_partition_alter_options]
```

示例

- 增加列

  在“user_log”表中增加一列“idcard”，示例如下：

  ```
  ALTER TABLE user_log
      ADD COLUMN idcard varchar(30);
  ```

- 增加索引

  在“user_log”表中为“idcard”列增加一个名为“idcard_idx”的索引，示例如下：

  ```
  ALTER TABLE user_log
      ADD INDEX idcard_idx (idcard);
  ```

- 删除索引

  删除“user_log”表中的“idcard_idx”索引，示例如下：

  ```
  ALTER TABLE user_log
      DROP INDEX idcard_idx;
  ```

- 重命名索引

  将“user_log”表中“idcard_idx”索引命修改为“idcard_idx_new”，示例如下：

  ```
  ALTER TABLE user_log
      RENAME INDEX `idcard_idx` TO `idcard_idx_new`;
  ```

- 修改字段

  将“user_log”表中“idcard”列（字段类型为varchar）的长度由30改为40，语法示例如下：

  ```
  ALTER TABLE user_log
      MODIFY COLUMN idcard varchar(40);
  ```

全局二级索引

PolarDB-X支持全局二级索引 (Global Secondary Index，GSI)，基本原理请参见全局二级索引。

列变更

使用全局二级索引的表，对列的修改，语法和普通表的一致。

> **说明** 当修改的表包含全局二级索引时，对列的修改有额外的限制，关于GSI的限制与约定，详情请参见如何使用全局二级索引。

索引变更

语法

```
ALTER TABLE tbl_name
```

```
    alter_specification # 全局二级索引相关变更仅支持一条alter_specification
alter_specification:
  | ADD GLOBAL {INDEX|KEY} index_name # 全局二级索引必须显式指定索引名
      [index_type] (index_sharding_col_name,...)
      global_secondary_index_option
      [index_option] ...
  | ADD [CONSTRAINT [symbol]] UNIQUE GLOBAL
      [INDEX|KEY] index_name # 全局二级索引必须显式指定索引名
      [index_type] (index_sharding_col_name,...)
      global_secondary_index_option
      [index_option] ...
  | DROP {INDEX|KEY} index_name
  | RENAME {INDEX|KEY} old_index_name TO new_index_name
# 关于全局二级索引特有语法，详情请参见CREATE TABLE文档
global_secondary_index_option:
    [COVERING (col_name,...)] # Covering Index
    partition_options # 包含且仅包含 index_sharding_col_name 中指定的列
# 指定索引表拆分方式
partition_options:
    PARTITION BY
          HASH({column_name | partition_func(column_name)})
        | KEY(column_list)
        | RANGE{({column_name | partition_func(column_name)})
        | RANGE COLUMNS(column_list)}
        | LIST{({column_name | partition_func(column_name)})
        | LIST COLUMNS(column_list)} }
    partition_list_spec
# 分区函数定义
partition_func:
    YEAR
  | TO_DAYS
  | TO_SECOND
  | UNIX_TIMESTAMP
  | MONTH
# 分区列表定义
partition_list_spec:
    hash_partition_list
  | range_partition_list
  | list_partition_list
# Hash/Key分区表列定义
hash_partition_list:
    PARTITIONS partition_count
# Range/Range Columns分区表列定义
range_partition_list:
    range_partition [, range_partition ...]
range_partition:
    PARTITION partition_name VALUES LESS THAN {(expr | value_list)} [partition_spec_options]
# List/List Columns分区表列定义
list_partition_list:
    list_partition [, list_partition ...]
list_partition:
    PARTITION partition_name VALUES IN (value_list) [partition_spec_options]
partition_spec_options:
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string' ]
# 以下为MySQL DDL语法
index_sharding_col_name:
    col_name [(length)] [ASC | DESC]
index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
```

```
index_type:
    USING {BTREE | HASH}
```

`ALTER TABLE ADD GLOBAL INDEX` 系列语法用于在建表后添加GSI，该系列语法在MySQL语法上新引入了GLOBAL关键字，用于指定添加的索引类型为GSI。

`ALTER TABLE { DROP | RENAME } INDEX` 语法同样可以对GSI进行修改，目前建表后创建GSI存在一定限制。关于GSI的限制与约定，详情请参见如何使用全局二级索引。

全局二级索引定义子句详细说明请参见CREATE TABLE。

示例

- **建表后添加全局二级索引**

  下面以建立全局唯一索引为例，介绍在建表后如何创建GSI。

```
# 创建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
partition by key(`order_id`)
partitions 4;
# 创建全局二级索引
ALTER TABLE t_order ADD UNIQUE GLOBAL INDEX `g_i_buyer` (`buyer_id`) COVERING (`order_snapshot`) partition
by key(`buyer_id`) partitions 4;
```

  - 主表：” t_order “只分库不分表，分库的拆分方式为按照” order_id “列进行哈希。
  - 索引表：” g_i_buyer “只分库不分表，分库的拆分方式为按照” buyer_id “列进行哈希，指定覆盖列为” order_snapshot “。
  - 索引定义子句：`UNIQUE GLOBAL INDEX `g_i_buyer`(`buyer_id`) COVERING (order_snapshot) partition by key(`buyer_id`) partitions 4` 。

  通过 `SHOW INDEX` 查看索引信息，包含拆分键order_id上的局部索引，和buyer_id、id、order_id和order_snapshot上的GSI，其中buyer_id为索引表的拆分键，id和order_id为默认的覆盖列（主键和主表的拆分键），order_snapshot显式指定的覆盖列。

  > **说明** 关于GSI的限制与约定，详情请参见如何使用全局二级索引，SHOW INDEX详细说明，请参见SHOW INDEX。

```
show index from t_order;
+--------------------+------------+-----------+--------------+-------------+-----------+-------------+----
------+--------+------+------------+---------+---------------+
| Table              | Non_unique | Key_name  | Seq_in_index | Column_name | Collation | Cardinality | Sub
_part | Packed | Null | Index_type | Comment | Index_comment |
+--------------------+------------+-----------+--------------+-------------+-----------+-------------+----
------+--------+------+------------+---------+---------------+
| t_order_syes_00000 |          0 | PRIMARY   |            1 | id          | A         |           0 |
NULL | NULL   |      | BTREE      |         |               |
| t_order_syes_00000 |          1 | l_i_order |            1 | order_id    | A         |           0 |
NULL | NULL   | YES  | BTREE      |         |               |
+--------------------+------------+-----------+--------------+-------------+-----------+-------------+----
------+--------+------+------------+---------+---------------+
2 rows in set (0.05 sec)
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息，详情请参见SHOW GLOBAL INDEX。

```
show global index;
+--------+---------+------------+-----------------+-------------+------------------------------+----------
--+------------------+---------------------+--------------------+------------------+---------------------+
--------------------+--------+
| SCHEMA | TABLE   | NON_UNIQUE | KEY_NAME        | INDEX_NAMES | COVERING_NAMES               | INDEX_TYP
E | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY |
TB_PARTITION_COUNT | STATUS |
+--------+--------+------------+-----------------+-------------+------------------------------+-----------
-+------------------+---------------------+--------------------+------------------+---------------------+-
-------------------+--------+
| d1     | t_order | 0          | g_i_buyer_$c1a0 | buyer_id    | id, order_id, order_snapshot | NULL
|                  |                     | NULL               |                  |                     | N
ULL               | PUBLIC |
+--------+---------+------------+-----------------+-------------+------------------------------+----------
--+------------------+---------------------+--------------------+------------------+---------------------+
--------------------+--------+
1 row in set (0.04 sec)
```

查看索引表的结构，索引表包含主表的主键、分库分表键、默认的覆盖列和自定义覆盖列，主键列去除了AUTO_INCREMENT属性，并且去除了主表中的局部索引，全局唯一索引默认在索引表的所有分库分表键上创建一个唯一索引，以实现全局唯一约束。

```
show create table g_i_buyer;
+-----------------+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------+
| TABLE           | CREATE TABLE
|
+-----------------+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------+
| g_i_buyer_$c1a0 | CREATE TABLE `g_i_buyer_$c1a0` (
    `id` bigint(11) NOT NULL,
    `order_id` varchar(20) DEFAULT NULL,
    `buyer_id` varchar(20) DEFAULT NULL,
    `order_snapshot` longtext,
    UNIQUE KEY `auto_shard_key_buyer_id` USING BTREE (`buyer_id`),
    KEY `_gsi_pk_idx_` USING BTREE (`id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8
PARTITION BY KEY(`buyer_id`)
PARTITIONS 4 |
+-----------------+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------+
1 row in set (0.10 sec)
```

- **删除全局二级索引**

  删除名为g_i_seller的GSI，相应的索引表也将被删除。

```
# 删除索引
ALTER TABLE `t_order` DROP INDEX `g_i_seller`;
```

- **重命名索引**

  默认情况下限制对GSI的重命名。关于GSI的限制与约定，详情请参见全局二级索引使用。

8.7.2.9. TRUNCATE TABLE

本文介绍了如何清空表数据。

语法

```
TRUNCATE [TABLE] tbl_name
```

详细语法参考 TRUNCATE TABLE Statement。

注意事项

使用TRUNCATE TABLE，需要有DROP权限。

8.7.2.10. RENAME TABLE

本文介绍了对表做重命名的方法。

语法

```
RENAME TABLE
    tbl_name TO new_tbl_name
```

注意事项

- PolarDB-X不支持同时RENAME多张表。
- 考虑到稳定性和性能，目前不支持重命名包含全局二级索引的表。
- 不支持单独重命名索引表。如需重命名索引，请参见ALTER TABLE中RENAME INDEX推荐的方法。

8.7.2.11. CREATE INDEX

PolarDB-X支持创建局部索引和全局二级索引 (Global Secondary Index，GSI)，同时支持删除这两种索引。

局部索引

关于局部索引，详情请参见CREATE INDEX Statement。

全局二级索引

关于全局二级索引基本原理，请参见全局二级索引。

语法

```
CREATE [UNIQUE]
    GLOBAL INDEX index_name [index_type]
    ON tbl_name (index_sharding_col_name,...)
    global_secondary_index_option
    [index_option]
    [algorithm_option | lock_option] ...
# 全局二级索引特有语法，具体说明请参见CREATE TABLE文档
global_secondary_index_option:
    [COVERING (col_name,...)]
    [partition_options]
# 分区策略定义
partition_options:
    PARTITION BY
          HASH({column_name | partition_func(column_name)})
        | KEY(column_list)
        | RANGE{({column_name | partition_func(column_name)})
        | RANGE COLUMNS(column_list)}
        | LIST{({column_name | partition_func(column_name)})
        | LIST COLUMNS(column_list)} }
    partition_list_spec
# 分区函数定义
partition_func:
    YEAR
  | TO_DAYS
  | TO_SECOND
  | UNIX_TIMESTAMP
  | MONTH
# 分区列表定义
partition_list_spec:
    hash_partition_list
  | range_partition_list
  | list_partition_list
# Hash/Key分区表列定义
hash_partition_list:
    PARTITIONS partition_count
# Range/Range Columns分区表列定义
range_partition_list:
    range_partition [, range_partition ...]
range_partition:
    PARTITION partition_name VALUES LESS THAN {(expr | value_list)} [partition_spec_options]
# List/List Columns分区表列定义
list_partition_list:
    list_partition [, list_partition ...]
list_partition:
    PARTITION partition_name VALUES IN (value_list) [partition_spec_options]
```

`CREATE GLOBAL INDEX` 系列语法用于在建表后添加GSI，该系列语法在MySQL语法上新引入了GLOBAL关键字，用于指定添加的索引类型为GSI。目前建表后创建GSI存在一定限制，关于GSI的限制与约定，详情请参见如何使用全局二级索引。

关于全局二级索引定义子句详细说明，请参见CREATE TABLE。

示例

下面以建立普通全局二级索引为例，介绍在建表后创建GSI。

```
# 先创建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 partition by hash(`order_id`) partitions 16;
# 再创建全局二级索引
CREATE GLOBAL INDEX `g_i_seller` ON t_order (`seller_id`) partition by hash(`seller_id`) partitions 16;
```

- 主表：” t_order “是分区表，分库的拆分方式为按照” order_id “列进行哈希。
- 索引表：” g_i_seller “按照” seller_id “列进行哈希，指定覆盖列为” order_snapshot “。
- 索引定义子句：`GLOBAL INDEX `g_i_seller` ON t_order (`seller_id`) partition by hash(`seller_id`) partitions 16` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键order_id上的局部索引，和seller_id、id和order_id上的GSI，其中seller_id为索引表的拆分键，id和order_id为默认的覆盖列（主键和主表的拆分键）。

> **说明** 关于GSI的限制与约定，详情请参见如何使用全局二级索引，SHOW INDEX详细说明，请参见SHOW INDEX。

```
show index from t_order;
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME   | SEQ_IN_INDEX | COLUMN_NAME | COLLATION | CARDINALITY | SUB_PART | PACK
ED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
| t_order |          0 | PRIMARY    |            1 | id          | A         |           0 |     NULL | NULL
|      | BTREE      |          |               |
| t_order |          1 | l_i_order  |            1 | order_id    | A         |           0 |     NULL | NULL
| YES  | BTREE      |          |               |
| t_order |          1 | g_i_seller |            1 | seller_id   | NULL      |           0 |     NULL | NULL
| YES  | GLOBAL     | INDEX    |               |
| t_order |          1 | g_i_seller |            2 | id          | NULL      |           0 |     NULL | NULL
|      | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_seller |            3 | order_id    | NULL      |           0 |     NULL | NULL
| YES  | GLOBAL     | COVERING |               |
+---------+------------+------------+--------------+-------------+-----------+-------------+----------+-----
---+------+------------+----------+---------------+
```

通过 `SHOW GLOBAL INDEX` 可以单独查看GSI信息，详情请参见SHOW GLOBAL INDEX。

```
show global index from t_order;
+---------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+---------------------+--------------------+-----------------+---------------------+----------
----------+--------+
| SCHEMA              | TABLE   | NON_UNIQUE | KEY_NAME   | INDEX_NAMES | COVERING_NAMES | INDEX_TYPE | DB_P
ARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTIT
ION_COUNT | STATUS |
+---------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+---------------------+--------------------+-----------------+---------------------+----------
----------+--------+
| ZZY3_DRDS_LOCAL_APP | t_order | 1          | g_i_seller | seller_id   | id, order_id   | NULL       | sell
er_id        | HASH                | 4                  |                 | NULL                | NULL
| PUBLIC |
+---------------------+---------+------------+------------+-------------+---------------+------------+-----
-------------+---------------------+--------------------+-----------------+---------------------+----------
----------+--------+
```

查看索引表的结构，索引表包含主表的主键、分库分表键、默认的覆盖列和自定义覆盖列，主键列去除了 `AUTO_INCREMENT` 属性，并且去除了主表中的局部索引。

```
show create table g_i_seller;
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| Table      | Create Table
|
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| g_i_seller | CREATE TABLE `g_i_seller` (
  `id` bigint(11) NOT NULL,
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `auto_shard_key_seller_id` (`seller_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 partition by hash(`seller_id`) partitions 16 |
+------------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
```

8.7.2.12. DROP INDEX

本文介绍了如何删除局部索引和全局二级索引。

局部索引

删除局部索引的方法和MySQL中一致，请参考DROP INDEX 文档。

全局二级索引

语法

```
DROP INDEX index_name ON tbl_name
```

> **说明** index_name为需要删除的全局二级索引名。

8.7.2.13. CREATE VIEW

本文将介绍如何使用CREATE VIEW语句为PolarDB-X创建视图。

语法

```
CREATE
    [OR REPLACE]
    VIEW view_name [(column_list)]
    AS select_statement
```

示例

```
# 先建表
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
# 创建视图
create view t_detail as select order_id,order_detail from t_order;
# 查询视图
select * from t_detail;
```

8.7.2.14. DROP VIEW

本文将介绍如何使用DROP VIEW语句删除PolarDB-X的视图。

语法

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

示例

```
# 创建视图
create view v as select 1;
# 删除视图
drop view v;
```

8.7.3. 秒级加字段（Instant Add Column）

本文介绍了什么是秒级加字段功能及如何使用。

传统方法给表中增加列时，存储节点上需要重建所有物理分片数据，会占用大量系统资源。PolarDB-X新增的秒级加字段（Instant Add Column）功能，在加列操作时只需变更表定义信息，无需修改已有数据，帮助您快速给任意大小的表增加列。

前提条件

- 此功能仅支持PolarDB-X 2.0版本的实例，并且要求实例版本为5.4.13-16415631及以上；
- 该功能默认关闭，使用时需要显式开启。

使用限制

- 不支持新增主键列；
- 不支持对具有ROW_FORMAT = COMPRESSED属性的表使用该功能；

> **说明** 执行 `SHOW CREATE TABLE` 语句，可查看表是否具有ROW_FORMAT = COMPRESSED属性。

- 不支持对具有全文索引的表使用该功能；
- 同一条 `ALTER TABLE` 语句中只能包含增加列操作，包含任何其它操作时则不支持。

开启方法

执行以下语句，开启秒级加字段功能：

```
SET enable_set_global=TRUE;
SET global support_instant_add_column=ON;
```

参数说明：

参数名称	级别	参数说明
support_instant_add_column	Global	秒级加字段功能的开关： • ON：开启秒级加字段功能； • OFF：关闭秒级加字段功能（默认值）。

使用方法

当 `ALTER TABLE` 语句为下列情况时，该语句使用了秒级加字段功能：

- 未指定算法，示例如下：

```
ALTER TABLE test ADD COLUMN col2 int;
```

- 将算法指定为**DEFAULT**，示例如下：

```
ALTER TABLE test ADD COLUMN col3 int, ALGORITHM=DEFAULT;
```

- 将算法指定为**INSTANT**，示例如下：

```
ALTER TABLE test ADD COLUMN col1 int, ALGORITHM=INSTANT;
```

> **说明**
>
> 如果指定了其它算法，例如INPLACE或COPY，则不能使用该功能，PolarDB-X将采用普通方式执行增加列操作。示例如下：

```
ALTER TABLE test ADD COLUMN col4 int, ALGORITHM=INPLACE;
ALTER TABLE test ADD COLUMN col5 int, ALGORITHM=COPY;
```

8.8. DML语句

8.8.1. INSERT

您可以使用INSERT语句往表中插入数据。

语法

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
{VALUES | VALUE} (value_list) [, (value_list)]
[ON DUPLICATE KEY UPDATE assignment_list]
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
SET assignment_list
[ON DUPLICATE KEY UPDATE assignment_list]
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
SELECT ...
[ON DUPLICATE KEY UPDATE assignment_list]
value_list:
value [, value] ...
value:
{expr | DEFAULT}
assignment_list:
assignment [, assignment] ...
assignment:
col_name = value
```

语法限制

不支持使用以下语法。

- INSERT IGNORE ON DUPLICATE KEY UPDATE语法，例如：

```
INSERT IGNORE INTO tb (id) VALUES(7) ON DUPLICATE KEY UPDATE id = id + 1;
```

- PARTITION语法，例如：

```
INSERT INTO tb PARTITION (p0) (id) VALUES(7);
```

- 嵌套NEXTVAL的语法，例如：

```
INSERT INTO tb(id) VALUES(SEQ1.NEXTVAL + 1);
```

- 包含列名的语法，例如：

```
INSERT INTO tb(id1, id2) VALUES(1, id1 + 1);
```

相关文献

MySQL INSERT 语法。

8.8.2. REPLACE

您可以使用REPLACE语法往表中插入行或替换表中的行。

语法

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
{VALUES | VALUE} (value_list) [, (value_list)]
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
SET assignment_list
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
SELECT ...
value_list:
value [, value] ...
value:
{expr | DEFAULT}
assignment_list:
assignment [, assignment] ...
assignment:
col_name = value
```

语法限制

不支持使用以下语法。

- PARTITION语法，例如：

```
REPLACE INTO tb PARTITION (p0) (id) VALUES(7);
```

- 包含列名的语法，例如：

```
REPLACE INTO tb(id1, id2) VALUES(1, id1 + 1);
```

相关文献

MySQL REPLACE语法。

8.8.3. UPDATE

您可以使用UPDATE语法修改表中符合条件的行。

语法

- 单逻辑表

```
UPDATE [LOW_PRIORITY] [IGNORE] [schema_name.]tbl_name
    SET assignment_list
    [WHERE where_condition]
value:
    {expr | DEFAULT}
assignment:
    col_name = value
assignment_list:
    assignment [, assignment] ...
```

- 多逻辑表

```
UPDATE [LOW_PRIORITY] [IGNORE] table_references
    SET assignment_list
    [WHERE where_condition]
```

> **说明**
> - UPDATE支持如下修饰符：
> - 若设置LOW_PRIORITY，UPDATE操作将在该表没有任何读操作之后执行。
> - 若设置IGNORE，将会忽略更新过程中的错误，即更新不会被错误中断。
> - UPDATE语句中的修饰符均会原样下推至存储层MySQL，不会对PolarDB-X的修饰符操作产生影响。

语法限制

与原生MySQL的UPDATE语法相比，PolarDB-X的UPDATE语法存在以下限制。

不支持在SET子句中使用子查询（相关子查询和非相关子查询），例如：

```
UPDATE t1 SET name = (SELECT name FROM t2 WHERE t2.id = t1.id) WHERE id > 10;
```

> **说明** t1和t2的拆分键为ID。

相关文献

MySQL UPDATE语法。

8.8.4. DELETE

您可以使用DELETE语句删除表中符合条件的行。

语法

下述DELETE语句表示从 `tbl_name` 中删除满足 `where_condition` 的行，并返回删除的行数；若没有WHERE条件，将删除表中所有的数据。

- 单逻辑表

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM [schema_name.]tbl_name
    [WHERE where_condition]
```

- 多逻辑表

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    tbl_name[.*] [, tbl_name[.*]] ...
    FROM table_references
    [WHERE where_condition]
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    FROM [schema_name.]tbl_name[.*] [, [schema_name.]tbl_name[.*]] ...
    USING table_references
    [WHERE where_condition]
```

> **说明**
> - DELETE支持如下修饰符：
> - 若设置LOW_PRIORITY，DELETE操作将在该表没有任何读操作之后执行。
> - 若设置IGNORE，则会忽略删除过程中产生的错误。
> - QUICK，与MySQL存储引擎相关，详情请参见MySQL文档。
> - DELETE语句中的修饰符均会原样下推至存储层MySQL，不会对PolarDB-X的修饰符操作产生影响。

8.8.5. SOURCE

本文介绍PolarDB-X两种导入数据命令。

mysql命令导入

使用mysql命令导入语法格式：

```
mysql -u用户名    -p密码    <  要导入的数据库数据(runoob.sql)
```

source命令导入

source命令导入数据库需要先登录到数库终端：

```
mysql> create database abc;      # 创建数据库
mysql> use abc;                  # 使用已创建的数据库
mysql> set names utf8;           # 设置编码
mysql> source /home/abc/abc.sql  # 导入备份数据库
```

8.8.6. EXPORT

本文介绍了PolarDB-X两种导出数据语法。

mysqldump

PolarDB-X支持MySQL官方数据导出工具mysqldump。mysqldump命令的详细说明请参见mysqldump。

- **语法**

```
      mysqldump -h ip -P port -u user -ppassword --default-character-set=char-set --net_buffer_length=102
40 --no-create-db --skip-add-locks --skip-lock-tables --skip-tz-utc --set-charset  [--hex-blob] [--no-data
] database [table1 table2 table3...] > dump.sql
```

- **参数说明**

参数名称	说明	是否必选
ip	PolarDB-X实例的IP。	是
port	PolarDB-X实例的端口。	否
user	PolarDB-X实例的用户名。	是
password	PolarDB-X实例的密码。 **说明** 密码前存在-p，之间没有空格。	是
char-set	指定的编码。	是
--hex-blob	使用十六进制格式导出二进制字符串字段。如果有二进制数据就必须使用本选项。影响的字段类型包括BINARY、VARBINARY、BLOB。	否
--no-data	不导出数据。	否
table	指定导出某个表。默认导出该数据库所有的表。	否

SELECT ... INTO OUTFILE

- **语法**

```
SELECT ... INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        [export_options]
export_options:
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
```

- **示例**

```
SELECT customer_id, firstname, surname INTO OUTFILE '/exportdata/customers.txt'
  FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
  LINES TERMINATED BY '\n'
  FROM customers;
```

- **限制**

  目前OUTFILE和mysql语法一致，导出的文件会保存在执行该SQL会话对应的PolarDB-X计算节点。

8.8.7. LOAD DATA

本文介绍在PolarDB-X中使用load data进行数据导入的相关限制和注意事项。

注意事项

- 使用load data进行数据导入时，load data语句并非一个事务，因此各种原因（如数据库宕机等）可能导致部分数据插入，剩余部分未插入。
- 客户端需要开启local-infile。

语法

```
LOAD DATA
    [LOCAL]
    INFILE 'file_name'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_mask [, col_name_or_mask] ...)]
```

参数说明

参数名称	说明
LOAD DATA [LOCAL] INFILE	文件位于服务端还是client端。
file_name	使用相对路径时，为相对于客户端启动时的路径。

参数名称	说明
REPLACE	导入数据时，遇到主键重复则强制用当前数据覆盖已有数据。
IGNORE	导入数据时，遇到主键重复则自动忽略该行。
[FIELDS] TERMINATED BY 'string'	定义每行数据的分隔符，默认为\t。
[FIELDS] ENCLOSED BY 'char'	每列数据的包围符。例如，某一列数据为"test"，定义enclosed by '"'后，导入数据时先将"test"前后的"移除，然后再导入数据。
[LINES] TERMINATED BY 'string'	定义行分隔符，默认为\n。
IGNORE number LINES	导入数据时忽略开始的某几行。例如，`IGNORE 1 LINES`，导入数据时忽略第一行数据。
(col_name_or_mask [, col_name_or_mask] ...)	1. 设置导入的列，如果不设置，默认按照表中的列顺序来导入数据。 2. 掩盖掉文件中的某些列，使文件中对应列失效，例如，`table test(x int, y int)`，导入文件有三列，导入时使用`(x, @name, y)`则会忽略文件中的第二列，使用第一列填充x，使用第二列填充y。

示例

创建测试表：

```
CREATE TABLE test ( a int(11) NOT NULL DEFAULT '0',  b varchar(8) NOT NULL,  PRIMARY KEY (a)  ) DBPARTITION by hash(a);
```

本地待导入文件：

```
x,y
test1,2
test2,3
test3,4
test4,5
test5,6
test7,8
test8,9
```

load data语句：

```
LOAD DATA LOCAL INFILE '~/test.txt' IGNORE INTO TABLE test FIELDS TERMINATED BY ',' LINES STARTING BY 'test' TERMINATED BY '\n' IGNORE 1 LINES;
```

结果如下：

```
mysql> select * from test order by a;
+------+------+
| a    | b    |
+------+------+
|    1 | 2    |
|    2 | 3    |
|    3 | 4    |
|    4 | 5    |
|    5 | 6    |
|    7 | 8    |
|    8 | 9    |
+------+------+
7 rows in set (0.02 sec)
```

8.9. DQL语句

8.9.1. SELECT

SELECT用于从一个或多个表中查询数据。

语法

```
SELECT
    [ALL | DISTINCT]
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [FOR UPDATE]
```

SELECT子句说明：

- select_expr：表示查询的列，SELETE必须至少有一个select_expr。
- table_references：表示从哪些表中获取数据。
- WHERE子句：指定查询条件，即从表中获取满足 `where_condition` 的行，若没有指定，则获取所有行。
- GROUP BY子句：支持列名，表达式以及输出列中的位置引用。
- HAVING子句：与WHERE类似，不同点在于可以使用聚合函数。
- ORDER BY：指定排序，支持列名、表达式以及输出列中的位置引用，同时支持指定排序方向，ASC（升序）或 DESC（降序）。
- LIMIT/OFFSET：限定输出结果集的偏移量和大小。LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1)： 为了与PostgreSQL兼容，MySQL也支持句法： LIMIT # OFFSET #。
- FOR UPDATE：对查询结果所有行加排他锁，以阻止其他事务的并发修改，或阻止在某些事务隔离级别时的并发读取。

注意事项

- 不支持在HAVING中使用适用于WHERE子句的表达式，如下SQL1应改写为SQL2。

  SQL1：

  ```
  SELECT col_name FROM tbl_name HAVING col_name > 0;
  ```

  SQL2：

  ```
  SELECT col_name FROM tbl_name WHERE col_name > 0;
  ```

- HAVING子句可以引用聚合函数，但WHERE子句不可以。

  ```
  SELECT user, MAX(salary) FROM users
    GROUP BY user HAVING MAX(salary) > 10;
  ```

- LIMIT若有两个参数，第一个参数表示返回第一行的偏移量，第二个参数表示返回的行数；若仅有一个参数，则表示返回的行数，默认偏移量为0。
- GROUP BY子句不支持ASC和DESC。
- 同时存在GROUP BY和ORDER BY时，ORDER BY后面的表达式必须在SELECT表达式或GROUP BY表达式中，如不支持以下SQL。

  ```
  SELECT user FROM users GROUP BY age ORDER BY salary;
  ```

- 暂不支持ORDER BY子句中使用聚合函数以及包含聚合函数的表达式，可将表达式作为 `select_expr` ，并赋予别名，在ORDER BY子句中引用该别名。

- 暂不支持以空字符串作为别名。

JOIN

PolarDB-X支持在SELECT语句的table_references中使用如下JOIN语法：

```
table_references:
    escaped_table_reference [, escaped_table_reference] ...
escaped_table_reference:
    table_reference
  | { OJ table_reference }
table_reference:
    table_factor
  | join_table
table_factor:
    [schema_name.]tbl_name [[AS] alias] [index_hint_list]
  | table_subquery [AS] alias
  | ( table_references )
join_table:
    table_reference [INNER | CROSS] JOIN table_factor [join_condition]
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_condition
join_condition:
    ON conditional_expr
  | USING (column_list)
index_hint_list:
    index_hint [, index_hint] ...
index_hint:
    USE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
  | IGNORE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
  | FORCE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
index_list:
    index_name [, index_name] ...
```

使用JOIN语句时，应考虑如下因素：

- JOIN、CROSS JOIN与INNER JOIN语法是等价的，这种设定与MySQL保持一致。
- 如果INNER JOIN没有ON条件，其与”逗号“连接是等价的，均表示笛卡尔积。如下两条SQL等价：

  ```
  SELECT * FROM t1 INNER JOIN t2 WHERE t1.id > 10
  SELECT * FROM t1, t2 WHERE t1.id > 10
  ```

- `USING(column_list)` 指定连接两表中都存在的列名，PolarDB-X会按照这些列构建等值条件。如下两条SQL等价：

  ```
  a LEFT JOIN b USING(c1, c2)
  a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2
  ```

- JOIN的优先级高于”逗号“操作符，对于连接表达式t1, t2 JOIN t3会转换为(t1, (t2 JOIN t3)，而不是((t1, t2) JOIN t3)。
- 外连接LEFT / RIGHT JOIN必须有ON条件。
- index_hint用于告知MySQL使用哪个索引，PolarDB-X会将该Hint下推至底层MySQL。
- 暂不支持STRAIGHT_JOIN和NATURAL JOIN。

UNION

PolarDB-X支持如下UNION语法：

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

> **说明** 对于UNION中的每个SELECT，PolarDB-X暂不支持使用多个同名的列。例如以下SQL的SELECT中存在重复的列名，暂不支持。

```
SELECT id, id, name FROM t1 UNION SELECT pk, pk, name FROM t2;
```

相关文档

- MySQLSELECT语法
- MySQLJOIN语法
- MySQLUNION语法

8.9.2. 子查询

本文介绍PolarDB-X支持的子查询类别及在PolarDB-X中使用子查询的相关限制和注意事项。

使用限制

相比原生MySQL，PolarDB-X在子查询使用上增加了如下限制：

- 不支持在HAVING子句中使用子查询，示例如下：

```
SELECT name, AVG( quantity )
FROM tb1
GROUP BY name
HAVING AVG( quantity ) > 2* (
   SELECT AVG( quantity )
   FROM tb2
 );
```

- 不支持在JOIN ON子句中使用子查询，示例如下：

```
SELECT * FROM tb1 p JOIN tb2 s on (p.id=s.id and p.quantity>All(select quantity from tb3))
```

- 等号操作行符的标量子查询（The Subquery as Scalar Operand）不支持ROW语法。示例如下：

```
select * from tb1 where row(id, name) = (select id, name from tb2)
```

- 不支持在UPDATE SET子句中使用子查询，示例如下：

```
UPDATE t1 SET c1 = (SELECT c2 FROM t2 WHERE t1.c1 = t2.c1) LIMIT 10
```

注意事项

PolarDB-X中部分子查询仅能以APPLY的方式执行，查询效率低下。在实际使用中请尽量避免如下例子中的低效SQL：

- WHERE条件中OR与子查询共存时，执行效率会依外表数据情况大幅降低。示例如下：

```
高效：select * from tb1 where id in (select id from tb2)
高效：select * from tb1 where id in (select id from tb2) and id>3
低效：select * from tb1 where id in (select id from tb2) or  id>3
```

- 关联子查询（Correlated Subqueries）的关联项中带函数或非等号运算符。示例如下：

```
高效：select * from tb1 a where id in
      (select id from tb2 b where a.name=b.name)
低效：select * from tb1 a where id in
      (select id from tb2 b where UPPER(a.name)=b.name)
低效：select * from tb1 a where id in
      (select id from tb2 b where a.decimal_test=abs(b.decimal_test))
低效：select * from tb1 a where id in
      (select id from tb2 b where a.name!=b.name)
低效：select * from tb1 a where id in
      (select id from tb2 b where a.name>=b.name)
```

- 关联子查询（Correlated Subqueries）关联项与其它条件的逻辑运算符为OR。示例如下：

```
高效：select * from tb1 a where id in
      (select id from tb2 b where a.name=b.name
                                and b.date_test<'2015-12-02')
低效：select * from tb1 a where id in
      (select id from tb2 b where a.name=b.name
                                or b.date_test<'2015-12-02')
低效：select * from tb1 a where id in
      (select id from tb2 b where a.name=b.name
                                or b.date_test=a.date_test)
```

- 标量子查询（The Subquery as Scalar Operand）带关联项。示例如下：

```
高效：select * from tb1 a where id >
        (select id from tb2 b where b.date_test<'2015-12-02')
低效：select * from tb1 a where id >
        (select id from tb2 b where a.name=b.name
                                  and b.date_test<'2015-12-02')
```

- 跨关联层子查询。示例如下：
  - SQL多层关联，每层子查询关联项仅与直接上层关联，此类高效。

```
高效：select * from tb1 a where id in(select id from tb2 b
        where a.name=b.name and
        exists (select name from tb3 c where b.address=c.address))
```

  - SQL多层关联，但 `表c` 的子查询关联项中与 `表a` 的列进行了关联，此类低效。

```
低效：select * from tb1 a where id in(select id from tb2 b
        where a.name=b.name and
        exists (select name from tb3 c where a.address=c.address))
```

> **说明** 上述示例中，`表a` 和 `表b` 、`表b` 和 `表c` 为直接层级关联，`表a` 和 `表c` 间为跨层关联。

- 子查询中包含GROUP BY，请确保GROUP BY的分组列包含关联项。示例如下：
  - SQL子查询中包含聚合函数和关联项，关联项 `b.pk` 包含于分组列 `pk` 之中，此类高效。

```
高效：select * from tb1 a where exists
    (select pk from tb2 b
                where a.pk=b.pk and  b.date_test='2003-04-05'
                group by pk);
```

  - SQL子查询中包含聚合函数和关联项，关联项 `b.date_test` 不包含于分组列 `pk` 之中，此类低效。

```
低效：select * from tb1 a where exists
    (select pk from tb2 b
                where a.date_test=b.date_test and b.date_test='2003-04-05'
                group by pk);
```

支持的子查询

PolarDB-X目前支持如下类别的子查询：

- Comparisons Using Subqueries

  Comparisons Using Subqueries指带有比较运算符的子查询，这类子查询最为常见。
  - 语法

```
non_subquery_operand comparison_operator (subquery)
comparison_operator：  =  >  <  >=  <=  <>  !=  <=> like
```

- 示例

```
select * from tb1 WHERE 'a' = (SELECT column1 FROM t1)
```

> **说明** 目前仅支持子查询在比较运算符的右边。

- Subqueries with ANY、ALL、IN/NOT IN、EXISTS/NOT EXISTS
  - 语法

```
operand comparison_operator ANY (subquery)
operand comparison_operator ALL (subquery)
operand IN (subquery)
operand NOT IN (subquery)
operand EXISTS (subquery)
operand NOT EXISTS (subquery)
comparison_operator:=  >  <  >=  <=  <>  !=
```

  - 示例
    - ANY：如果子查询返回的任意一行满足ANY前的表达式，返回TRUE，否则返回FALSE。
    - ALL：如果子查询返回所有行都满足ALL前的表达式，返回TRUE，否则返回FALSE。
    - IN：在子查询前使用时，IN等价于 `=ANY` 。示例如下：

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 IN    (SELECT s1 FROM t2);
```

    - NOT IN：NOT IN在子查询前使用时，等价于 `<>ALL` 。示例如下：

```
SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2);
```

    - EXISTS：如果子查询返回任意行，EXISTS子查询结果为TRUE；如果子查询返回空值，EXISTS子查询结果为FALSE。示例如下：

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

> **说明** 如果EXISTS子查询中包含任意行，即使只包含NULL的行值，WHERE条件也会返回TRUE。

    - NOT EXISTS：如果子查询返回任意行，NOT EXISTS子查询结果为FALSE；如果子查询返回空值，NOT EXISTS子查询结果为TRUE。

- Row Subqueries
  - Row Subqueries支持如下比较运算符：

```
comparison_operator: =  >  <  >=  <=  <>  !=  <=>
```

  - 示例

```
SELECT * FROM t1
  WHERE (col1,col2) = (SELECT col3, col4 FROM t2 WHERE id = 10);
SELECT * FROM t1
  WHERE ROW(col1,col2) = (SELECT col3, col4 FROM t2 WHERE id = 10);
```

  以上两个SQL是等价的，只有同时满足以下条件时，t1表的数据行才会返回：
    - 子查询（ `SELECT col3, col4 FROM t2 WHERE id=10` ）仅返回一行记录，返回多行会报错。
    - 子查询返回的 `col3` ， `col4` 结果与主表中 `col1` ， `col2` 的值需一一对应。

- Correlated Subqueries

  Correlated Subqueries指子查询中包含对外层查询表的引用。示例如下：

```
SELECT * FROM t1
  WHERE column1 = ANY (SELECT column1 FROM t2
                       WHERE t2.column2 = t1.column2);
```

示例子查询SQL中并没有包含表t1及其列名column2，此时会向上一层寻找表t1的引用。

- Derived Tables（Subqueries in the FROM Clause）

  Derived Tables指在FROM子句中的子查询。

  - 语法

    ```
    SELECT ... FROM (subquery) [AS] tbl_name ...
    ```

  - 示例

    a. 数据准备：

    使用如下语法创建表t1：

    ```
    CREATE TABLE t1 (s1 INT, s2 CHAR(5), s3 FLOAT);
    INSERT INTO t1 VALUES (1,'1',1.0);
    INSERT INTO t1 VALUES (2,'2',2.0);
    ```

    使用如下查询并得到查询结果为 `2, '2', 4.0` 。

    ```
    SELECT sb1,sb2,sb3
      FROM (SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1) AS sb
      WHERE sb1 > 1;
    ```

    b. 查询需求：获取分组数据SUM后的平均值。

    若直接使用如下SQL则会报错，无法执行：

    ```
    SELECT AVG(SUM(s1)) FROM t1 GROUP BY s1;
    ```

    此时可使用如下Derived Tables子查询，并得到查询结果为 `1.5000` ：

    ```
    SELECT AVG(sum_s1)
      FROM (SELECT SUM(s1) AS sum_s1
            FROM t1 GROUP BY s1) AS t1;
    ```

    > **说明**
    > - Derived Tables必须拥有一个别名（如示例中的 `t1` ）。
    > - Derived Tables可以返回一个标量、列、行或表。
    > - Derived Tables不可以成为Correlated Subqueries，即不能包含子查询外部表的引用。

8.9.3. Grouping Sets、Rollup和Cube扩展

在关系型数据库中，通常需要使用多个 `SELECT + UNION` 语句来实现按照多组维度的结果分组，PolarDB-X新增支持通过Grouping Sets、Rollup和Cube扩展来实现这一目的。此外，PolarDB-X还支持在SELECT命令或HAVING子句中使用GROUPING函数和GROUPING_ID函数，来帮助解释使用上述扩展时的结果。本文将介绍相关语法和示例。

注意事项

- 本文介绍的所有GROUP BY相关的扩展语法，均不支持查询下推至 `LogicalView` 算子中执行。关于查询下推，请参见查询改写与下推。
- 本文示例中所用测试数据信息如下：

  使用如下语句创建一张 `requests` 表：

```
CREATE TABLE requests (
  `id` int(10) UNSIGNED NOT NULL,
  `os` varchar(20) DEFAULT NULL,
  `device` varchar(20) DEFAULT NULL,
  `city` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8 dbpartition BY hash(`id`) tbpartition BY hash(`id`);
```

在 `requests` 表中使用如下语句插入测试所需的数据：

```
INSERT INTO requests (id, os, device, city) VALUES
(1, 'windows', 'PC', 'Beijing'),
(2, 'windows', 'PC', 'Shijiazhuang'),
(3, 'linux', 'Phone', 'Beijing'),
(4, 'windows', 'PC', 'Beijing'),
(5, 'ios', 'Phone', 'Shijiazhuang'),
(6, 'linux', 'PC', 'Beijing'),
(7, 'windows', 'Phone', 'Shijiazhuang');
```

GROUPING SETS扩展

- **功能介绍**

  GROUPING SETS是GROUP BY子句的扩展，可以生成一个结果集，该结果集实际上是基于不同分组的多个结果集的串联（与UNION ALL运算结果类似），但UNION ALL运算和GROUPING SETS扩展并不会消除合并结果集中的重复行。

- **语法**

```
GROUPING SETS (
  { expr_1 | ( expr_1a [, expr_1b ] ...) |
    ROLLUP ( expr_list ) | CUBE ( expr_list )
  } [, ...] )
```

  说明 GROUPING SETS扩展可包含一个或多个由半角逗号（,）分隔表达式（如 `expr_1` 或 `(expr_1a [, expr_1b ] ...)` ）的任意组合，以及带半角圆括号（()）的表达式列表（如 `( expr_list )` ），其中：

  - 每个表达式都可用于确定结果集的分组方式。
  - GROUPING SETS内也支持嵌套使用ROLLUP或者CUBE。

- **示例**

- 通过GROUPING SETS扩展对数据进行分组查询，语法如下：

```
select os,device, city ,count(*)
from requests
group by grouping sets((os, device), (city), ());
```

上述语句等效于如下语句：

```
select os, device, NULL, count(*)
from requests group by os, device
union all
select NULL, NULL, NULL, count(*)
from requests
union all
select null, null, city, count(*)
from requests group by city;
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| windows | PC     | NULL         |        3 |
| linux   | PC     | NULL         |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| ios     | Phone  | NULL         |        1 |
| NULL    | NULL   | Shijiazhuang |        3 |
| NULL    | NULL   | Beijing      |        4 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

> **说明** 未在分组集中使用的表达式，会用NULL充当占位符，便于对这些未在分组集使用的结果集进行操作，例如结果 `city` 列中显示为NULL的行。

- 通过在GROUPING SETS中嵌套ROLLUP来对数据进行分组，语法如下：

```
select os,device, city ,count(*) from requests
group by grouping sets((city), ROLLUP(os, device));
```

上述语句等效于如下语句：

```
select os,device, city ,count(*) from requests
group by grouping sets((city), (os), (os, device), ());
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| NULL    | NULL   | Shijiazhuang |        3 |
| NULL    | NULL   | Beijing      |        4 |
| windows | PC     | NULL         |        3 |
| linux   | PC     | NULL         |        1 |
| ios     | Phone  | NULL         |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| windows | NULL   | NULL         |        4 |
| linux   | NULL   | NULL         |        2 |
| ios     | NULL   | NULL         |        1 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

- 通过在GROUPING SETS中嵌套CUBE扩展来对数据进行分组，语法如下：

```
select os,device, city ,count(*) from requests
group by grouping sets((city), CUBE(os, device));
```

上述语句等效于如下语句：

```
select os,device, city ,count(*) from requests
group by grouping sets((city), (os), (os, device), (), (device));
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| NULL    | NULL   | Beijing      |        4 |
| NULL    | NULL   | Shijiazhuang |        3 |
| windows | PC     | NULL         |        3 |
| ios     | Phone  | NULL         |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| linux   | PC     | NULL         |        1 |
| windows | NULL   | NULL         |        4 |
| ios     | NULL   | NULL         |        1 |
| linux   | NULL   | NULL         |        2 |
| NULL    | PC     | NULL         |        4 |
| NULL    | Phone  | NULL         |        3 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

- 通过GROUP BY、CUBE和GROUPING SETS组合产生GROUPING SETS，示例如下：

```
select os,device, city, count(*)
from requests
group by os, cube(os,device), grouping sets(city);
```

上述语句等效于如下语句：

```
select os,device, city, count(*)
from requests
group by grouping sets((os,device,city),(os,city),(os,device,city));
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| linux   | Phone  | Beijing      |        1 |
| windows | Phone  | Shijiazhuang |        1 |
| windows | PC     | Shijiazhuang |        1 |
| linux   | PC     | Beijing      |        1 |
| windows | PC     | Beijing      |        2 |
| ios     | Phone  | Shijiazhuang |        1 |
| linux   | NULL   | Beijing      |        2 |
| windows | NULL   | Shijiazhuang |        2 |
| windows | NULL   | Beijing      |        2 |
| ios     | NULL   | Shijiazhuang |        1 |
+---------+--------+--------------+----------+
```

ROLLUP扩展

- **功能介绍**

  ROLLUP扩展生成一系列有总计的分层组，每个分层组都有小计。该层次结构的顺序由ROLLUP表达式列表中给定的表达式的顺序确定。该层次结构的顶部是列表中最左侧的项。每个连续项都会沿右侧在该层次结构中向下移动，最右侧的项是最低级别。

- **语法**

```
ROLLUP ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

说明

- 每个表达式都会用于确定结果集的分组方式。如果采用带圆括号形式的表达式，例如 `( expr_1a, expr_1b, ... )` ，则 `expr_1a` 和 `expr_1b` 返回的值组合定义层次结构的单个分组级别。
- 对于列表中的第一项，例如 `expr_1` 或 `( expr_1a, expr_1b, ...)` 的组合，PolarDB-X将为每个唯一值返回一个小计。对于列表中的第二项，例如 `expr_2` 或 `( expr_2a, expr_2b, ...)` 的组合，PolarDB-X将为第二项的每个分组中的每个唯一值返回一个小计，依此类推。最后，PolarDB-X将为整个结果集返回一个总计。
- 对于小计行，将为小计包含的各项返回NULL。

- **示例**
  - 通过ROLLUP对 `(os, device, city)` 按层级聚合的方式产生GROUPING SETS，语法如下：

```
select os,device, city, count(*)
from requests
group by rollup (os, device, city);
```

上述语句等效于如下语句：

```
select os,device, city, count(*)
from requests
group by os, device, city with rollup;
```

也等效于如下语句：

```
select os,device, city, count(*)
from requests
group by grouping sets ((os, device, city),(os, device),(os),());
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| windows | PC     | Beijing      |        2 |
| ios     | Phone  | Shijiazhuang |        1 |
| windows | PC     | Shijiazhuang |        1 |
| linux   | PC     | Beijing      |        1 |
| linux   | Phone  | Beijing      |        1 |
| windows | Phone  | Shijiazhuang |        1 |
| windows | PC     | NULL         |        3 |
| ios     | Phone  | NULL         |        1 |
| linux   | PC     | NULL         |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| windows | NULL   | NULL         |        4 |
| ios     | NULL   | NULL         |        1 |
| linux   | NULL   | NULL         |        2 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

- 通过ROLLUP对 `os, (os,device), city` 按层级聚合的方式产生GROUPING SETS，语法如下：

```
select os,device, city, count(*)
from requests
group by rollup (os, (os,device), city);
```

上述语句等效于如下语句：

```
select os,device, city, count(*)
from requests
group by os, (os,device), city with rollup;
```

也等效于如下语句：

```
select os,device, city, count(*)
from requests
group by grouping sets ((os, device, city),(os, device),(os),());
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| windows | PC     | Beijing      |        2 |
| windows | PC     | Shijiazhuang |        1 |
| linux   | PC     | Beijing      |        1 |
| linux   | Phone  | Beijing      |        1 |
| windows | Phone  | Shijiazhuang |        1 |
| ios     | Phone  | Shijiazhuang |        1 |
| windows | PC     | NULL         |        3 |
| linux   | PC     | NULL         |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| ios     | Phone  | NULL         |        1 |
| windows | NULL   | NULL         |        4 |
| linux   | NULL   | NULL         |        2 |
| ios     | NULL   | NULL         |        1 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

CUBE扩展

- **功能介绍**

  CUBE扩展与ROLLUP扩展类似，但与生成分组并基于ROLLUP表达式列表中从左到右的项列表生成层次结构的ROLLUP扩展不同，CUBE是基于CUBE表达式列表中所有项的每个排列生成分组和小计。因此，与对同一表达式列表执行的ROLLUP相比，CUBE结果集会包含更多的行。

- **语法**

```
CUBE ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

> **说明**
>
> - 每个表达式都会用于确定结果集的分组方式。如果采用带半角圆括号的形式，例如 `( expr_1a, expr_1b, ...)`，则 `expr_1a` 和 `expr_1b` 返回的值组合定义单个组。
> - 对于列表中的第一项，例如 `expr_1` 或 `( expr_1a, expr_1b, ...)` 的组合，PolarDB-X将为每个唯一值返回一个小计。对于列表中的第二项，例如 `expr_2` 或 `( expr_2a, expr_2b, ...)` 的组合，PolarDB-X在为每个唯一值返回一个小计的同时，还将为第一项和第二项的每个唯一组合返回一个小计。如果存在第三项，PolarDB-X则会为第三项的每个唯一值、第三项和第一项组合的每个唯一值、第三项和第二项组合的每个唯一值以及第三项、第二项和第一项组合的每个唯一值返回一个小计。最后，再将为整个结果集返回一个总计。
> - 对于小计行，将为小计包含的各项返回NULL。

- **示例**

- 通过CUBE枚举 `(os, device, city)` 的所有可能列为GROUPING SETS，语法如下：

```
select os,device, city, count(*)
from requests
group by cube (os, device, city);
```

上述语句等效于如下语句：

```
select os,device, city, count(*)
from requests
group by grouping sets ((os, device, city),(os, device),(os, city),(device,city),(os),(device),(city),()
);
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| linux   | Phone  | Beijing      |        1 |
| windows | Phone  | Shijiazhuang |        1 |
| windows | PC     | Beijing      |        2 |
| ios     | Phone  | Shijiazhuang |        1 |
| windows | PC     | Shijiazhuang |        1 |
| linux   | PC     | Beijing      |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| windows | PC     | NULL         |        3 |
| ios     | Phone  | NULL         |        1 |
| linux   | PC     | NULL         |        1 |
| linux   | NULL   | Beijing      |        2 |
| windows | NULL   | Shijiazhuang |        2 |
| windows | NULL   | Beijing      |        2 |
| ios     | NULL   | Shijiazhuang |        1 |
| linux   | NULL   | NULL         |        2 |
| windows | NULL   | NULL         |        4 |
| ios     | NULL   | NULL         |        1 |
| NULL    | Phone  | Beijing      |        1 |
| NULL    | Phone  | Shijiazhuang |        2 |
| NULL    | PC     | Beijing      |        3 |
| NULL    | PC     | Shijiazhuang |        1 |
| NULL    | Phone  | NULL         |        3 |
| NULL    | PC     | NULL         |        4 |
| NULL    | NULL   | Beijing      |        4 |
| NULL    | NULL   | Shijiazhuang |        3 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

- 通过CUBE枚举 `(os, device),(device, city)` 所有可能列为GROUPING SETS，语法如下：

```
select os,device, city, count(*)
from requests
group by cube ((os, device), (device, city));
```

上述语句等效于如下语句：

```
select os,device, city, count(*)
from requests
group by grouping sets ((os, device, city),(os, device),(device,city),());
```

返回结果如下：

```
+---------+--------+--------------+----------+
| os      | device | city         | count(*) |
+---------+--------+--------------+----------+
| linux   | Phone  | Beijing      |        1 |
| windows | Phone  | Shijiazhuang |        1 |
| windows | PC     | Beijing      |        2 |
| windows | PC     | Shijiazhuang |        1 |
| linux   | PC     | Beijing      |        1 |
| ios     | Phone  | Shijiazhuang |        1 |
| linux   | Phone  | NULL         |        1 |
| windows | Phone  | NULL         |        1 |
| windows | PC     | NULL         |        3 |
| linux   | PC     | NULL         |        1 |
| ios     | Phone  | NULL         |        1 |
| NULL    | Phone  | Beijing      |        1 |
| NULL    | Phone  | Shijiazhuang |        2 |
| NULL    | PC     | Beijing      |        3 |
| NULL    | PC     | Shijiazhuang |        1 |
| NULL    | NULL   | NULL         |        7 |
+---------+--------+--------------+----------+
```

GROUPING和GROUPING_ID函数

- **功能介绍**
  - **GROUPING函数**

    在GROUP BY子句使用GROUPING SETS、ROLLUP、或CUBE扩展时，GROUPING SETS结果中会使用NULL来充当占位符，导致无法区分占位符NULL与数据中真正的NULL。此时，您可以使用PolarDB-X提供的GROUPING函数来作区分。

    GROUPING函数接受一个列名作为参数，如果结果对应行使用了参数列做聚合，则结果返回0，此时意味着NULL来自输入数据。如果结果对应行未使用参数列做聚合，则返回1，此时意味着NULL来自GROUPING SETS结果中的占位符。

  - **GROUPING_ID函数**

    GROUPING_ID函数简化了GROUPING函数，用于确定ROLLBACK、CUBE或GROUPING SETS扩展的结果集中行的小计级别。GROUPING函数仅采用一个列表达式并返回一个值来指示行是否为给定列的所有值的小计。因此，当解释具有多个分组列的查询的小计级别时，可能需要多个 GROUPING函数。GROUPING_ID函数接受ROLLBACK、CUBE或GROUPINGSETS扩展中已使用的一个或多个列表达式，并返回单个整数，该整数可用于确定其中哪一列已聚合小计。

- **语法**
  - **GROUPING函数**

```
SELECT [ expr ...,] GROUPING( col_expr ) [, expr ] ...
FROM ...
GROUP BY { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr
  [, ...] ) [, ...]
```

> **说明** GROUPING函数采用单个参数，该参数必须是GROUP BY子句中ROLLUP、CUBE或GROUPING SETS扩展的表达式列表中指定的维度列的表达式。

- **GROUPING_ID函数**

```
SELECT [ expr ...,]
  GROUPING_ID( col_expr_1 [, col_expr_2 ] ... )
  [, expr ] ...
FROM ...
GROUP BY { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr_1
  [, col_expr_2 ] [, ...] ) [, ...]
```

- **示例**

  通过GROUPING_ID函数将多个列名作为参数，并将参数列的GROUPING结果按照Bitmap的方式组成整数，语法如下：

```
select a,b,c,count(*),
grouping(a) ga, grouping(b) gb, grouping(c) gc, grouping_id(a,b,c) groupingid
from (select 1 as a ,2 as b,3 as c)
group by cube(a,b,c);
```

  返回结果如下：

```
+------+------+------+----------+------+------+------+------------+
| a    | b    | c    | count(*) | ga   | gb   | gc   | groupingid |
+------+------+------+----------+------+------+------+------------+
|    1 |    2 |    3 |        1 |    0 |    0 |    0 |          0 |
|    1 |    2 | NULL |        1 |    0 |    0 |    1 |          1 |
|    1 | NULL |    3 |        1 |    0 |    1 |    0 |          2 |
|    1 | NULL | NULL |        1 |    0 |    1 |    1 |          3 |
| NULL |    2 |    3 |        1 |    1 |    0 |    0 |          4 |
| NULL |    2 | NULL |        1 |    1 |    0 |    1 |          5 |
| NULL | NULL |    3 |        1 |    1 |    1 |    0 |          6 |
| NULL | NULL | NULL |        1 |    1 |    1 |    1 |          7 |
+------+------+------+----------+------+------+------+------------+
```

8.10. TCL语句

本文介绍了PolarDB-X支持的TCL语句及其用法。

数据库事务（Database Transaction）是指作为单个逻辑工作单元执行的一系列操作。事务处理可以用来维护数据库的完整性，保证成批的SQL操作全部执行或全部不执行。

默认情况下，PolarDB-X以开启自动提交（autocommit）的模式运行，也就是每条SQL语句单独构成一个事务（出于性能考虑，跨分片的DML语句默认未开启分布式事务，仅对更新GSI、更新广播表、更新拆分键的DML语句默认开启分布式事务）。您可以通过执行 `SET AUTOCOMMIT=0` 或者显式事务的方式手动开启多条语句组成的交互式事务。

显示事务是由用户自定义或用户指定的事务。通过START TRANSACTION，或BEGIN（被作为START TRANSACTION的别名得到支持）语句显示开始，以COMMIT或ROLLBACK语句显示结束。

语法

```
START TRANSACTION
  [transaction_characteristic [, transaction_characteristic] ...]
transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | ISOLATION LEVEL {REPEATABLE READ | READ COMMITTED}
  | READ WRITE
  | READ ONLY
}
BEGIN
COMMIT
ROLLBACK
SET autocommit = {0 | 1}
```

参数说明

参数	说明
START TRANSACTION [READ ONLY | READ WRITE]	启动新的事务，随后执行的DML语句（即INSERT、UPDATE、DELETE等）直到事务提交时才会生效。 • READ ONLY子句表示事务以只读方式开启，事务内不允许执行修改操作。 • READ WRITE子句表示事务以读写方式开启，默认为此种模式。
START TRANSACTION WITH CONSISTENT SNAPSHOT	启动新的事务，如果没有显式指定隔离级别，则事务的隔离级别为REPEATABLE READ。
START TRANSACTION ISOLATION LEVEL {REPEATABLE READ | READ COMMITTED}	启动新的事务，并为事务设置指定的隔离级别。
BEGIN	BEGIN被作为START TRANSACTION的别名得到支持。
COMMIT	提交当前事务。
ROLLBACK	回滚当前事务。
SET autocommit = {0 | 1}	为当前会话（session）关闭/开启自动提交（autocommit）模式。

8.11. DAL语句

8.11.1. DDL管理语句

8.11.1.1. SHOW DDL

一条DDL语句会以一个DDL任务方式在PolarDB-X内部执行，使用SHOW DDL命令可以查看当前PolarDB-X实例中的DDL任务。

语法

```
SHOW [FULL] DDL;
```

示例

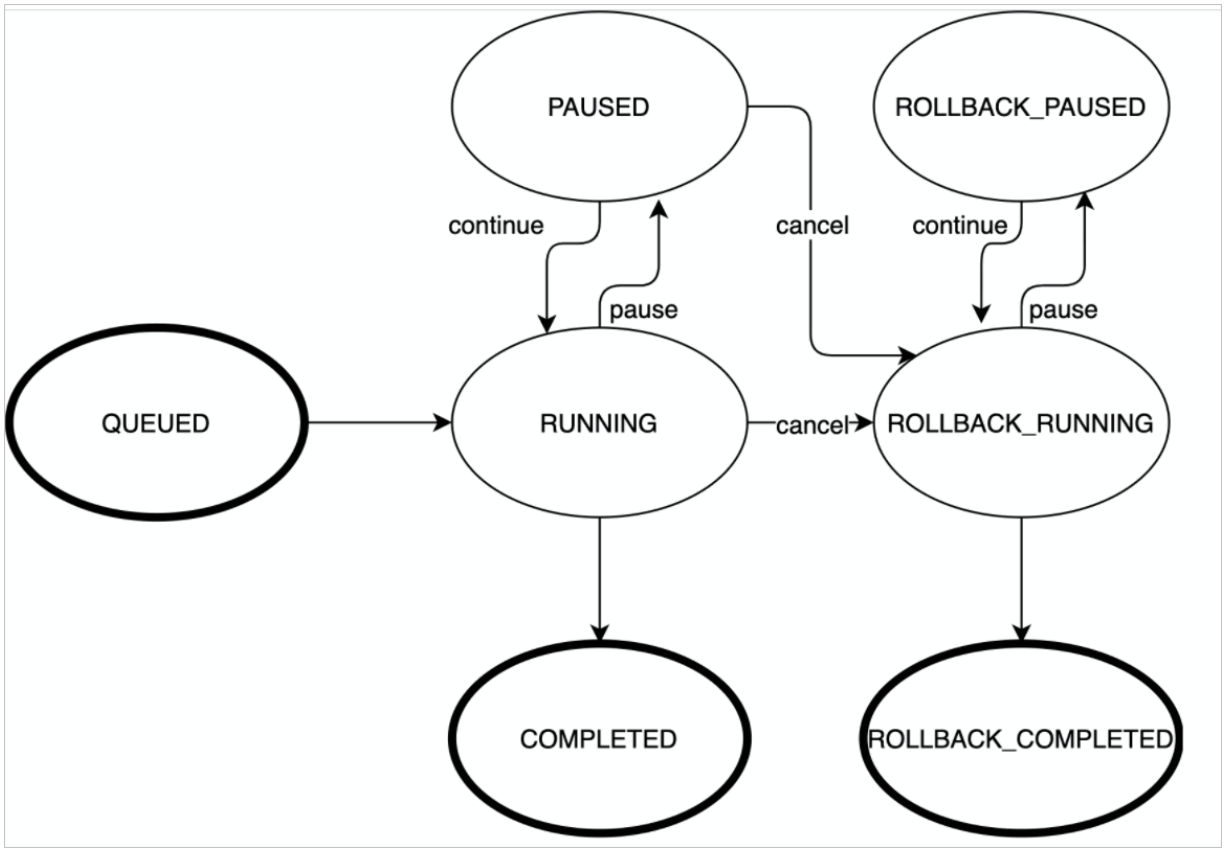
```
mysql> show ddl\G;
*************************** 1. row ***************************
           JOB_ID: 1359947811109470208
    OBJECT_SCHEMA: d1
      OBJECT_NAME: t2
           ENGINE: DAG
         DDL_TYPE: ALTER_TABLE
            STATE: PAUSED
BACKFILL_PROGRESS: 0%
 PHY_DDL_PROGRESS: 100%
         PROGRESS: 20%
       START_TIME: 2021-08-05 11:01:25.291
         END_TIME: 2021-08-05 11:02:27.020
 ELAPSED_TIME(MS): 61729
      PHY_PROCESS:
       CANCELABLE: true
1 row in set (0.03 sec)
```

返回参数说明

参数名称	说明
JOB_ID	DDL任务唯一标识，取值需为64位有符号长整型数值。
OBJECT_SCHEMA	DDL任务对象的Schema名称。
OBJECT_NAME	DDL任务对象名称，例如当前执行DDL的表名称。
ENGINE	DDL任务引擎类型，默认为DAG。
DDL_TYPE	DDL任务类型，例如，`CREATE_TABLE`。
STATE	DDL任务当前所处的状态，具体请参见DDL状态。
BACKFILL_PROGRESS	DDL任务中，数据回填的进度。
PHY_DDL_PROGRESS	DDL任务中，当前执行的一批物理DDL的进度。
PROGRESS	DDL任务的总体进度。
START_TIME	DDL任务开始执行的时间。
END_TIME	DDL任务结束执行的时间。
ELAPSED_TIME(MS)	DDL任务截止到任务查看时已经消耗的时间，单位：毫秒。
PHY_PROCESS	执行中的物理DDL的状态。
CANCELABLE	DDL任务是否可通过`CANCEL DDL`命令取消。

DDL状态

SHOW DDL命令返回的STATE字段标示了DDL任务的状态，状态转移图如下所示：



绝大多数情况，DDL任务会按照 `QUEUED->RUNNING->COMPLETED` 的流程执行成功。

异常情况下，DDL任务可能会自动回滚，此时的流程为 `QUEUED->RUNNING->ROLLBACK_RUNNING->ROLLBACK_COMPLETED` 。

特殊情况下，DDL任务的状态还可以通过命令干预，上图中箭头上展示的是可用于修改DDL任务状态的命令。

8.11.1.2. SHOW DDL RESULT

可以通过 `SHOW DDL RESULT` 命令查看近期执行过的DDL任务的结果。

语法

```
SHOW DDL RESULT;
```

示例

```
mysql> show ddl result\G;
*************************** 1. row ***************************
        JOB_ID: 1359590227445809152
   SCHEMA_NAME: test
   OBJECT_NAME: test_table
      DDL_TYPE: CREATE_TABLE
   RESULT_TYPE: SUCCESS
RESULT_CONTENT: SUCCESS
1 row in set (0 min 0.83 sec)
```

返回参数说明

参数名称	说明
JOB_ID	DDL任务唯一标识，取值需为64位有符号长整型数值。
SCHEMA_NAME	DDL任务对象的Schema名称。
OBJECT_NAME	DDL任务对象名称，例如，当前执行DDL的表名称。
DDL_TYPE	DDL任务类型，例如， `CREATE_TABLE` 。
RESULT_TYPE	DDL任务执行结果的类型。
RESULT_CONTENT	DDL任务执行结果的详细信息。

8.11.1.3. CANCEL DDL

`CANCEL DDL` 命令可用于取消状态为RUNNING或PAUSED的DDL任务。取消之后，当前DDL任务已发生的变更将会回滚，数据表将恢复到DDL任务执行之前的状态。

语法

JOB_ID可通过 `SHOW DDL` 命令获取。

```
CANCEL DDL JOB_ID;
```

示例

```
mysql> cancel ddl 1359628009396502528;
Query OK, 1 row affected (0.03 sec)
```

8.11.1.4. PAUSE DDL

`PAUSE DDL` 命令可用于暂停当前状态为RUNNING或ROLLBACK_RUNNING的DDL任务。

语法

JOB_ID可通过 `SHOW DDL` 命令获取。

```
PAUSE DDL JOB_ID;
```

示例

```
mysql> pause ddl 1359630950316638208;
Query OK, 1 row affected (0.04 sec)
-- 再次
mysql> show ddl\G;
*************************** 1. row ***************************
           JOB_ID: 1359630950316638208
    OBJECT_SCHEMA: d2
      OBJECT_NAME: t1
           ENGINE: DAG
         DDL_TYPE: CREATE_TABLE
            STATE: PAUSED
         PROGRESS: 0%
       START_TIME: 2021-08-04 14:02:19.781
         END_TIME: 2021-08-04 14:02:35.276
 ELAPSED_TIME(MS): 15495
      PHY_PROCESS:
BACKFILL_PROGRESS: 0%
1 row in set (0.02 sec)
```

8.11.1.5. CONTINUE DDL

`CONTINUE DDL` 命令用于控制状态为PAUSED或ROLLBACK_PAUSED的DDL任务继续执行。

语法

JOB_ID可通过 `SHOW DDL` 命令获取。

```
CONTINUE DDL JOB_ID;
```

示例

```
mysql> continue ddl 1359630057722609664;
Query OK, 1 row affected (0.04 sec)
```

8.11.2. SHOW

8.11.2.1. SHOW HELP

本文介绍了如何使用SHOW HELP命令。

SHOW HELP用于展示PolarDB-X所有辅助SQL指令及其说明。

```
mysql> show help;
+----------------------------------------+-------------------------------------------------------------+-------
---------------------------------------+
| STATEMENT                              | DESCRIPTION                                                 | EXAMPL
```

```
E                                        |
+-----------------------------------------+-----------------------------------------------------------+-------
--------------------------------------+
| show rule                                | Report all table rule                                     |
|
| show rule from TABLE                     | Report table rule                                         | show r
ule from user                           |
| show full rule from TABLE                | Report table full rule                                    | show f
ull rule from user                      |
| show topology from TABLE                 | Report table physical topology                            | show t
opology from user                       |
| show partitions from TABLE               | Report table dbPartition or tbPartition columns           | show p
artitions from user                     |
| show broadcasts                          | Report all broadcast tables                               |
|
| show datasources                         | Report all partition db threadPool info                   |
|
| show node                                | Report master/slave read status                           |
|
| show slow                                | Report top 100 slow sql                                   |
|
| show physical_slow                       | Report top 100 physical slow sql                          |
|
| clear slow                               | Clear slow data                                           |
|
| trace SQL                                | Start trace sql, use show trace to print profiling data | trace
select count(*) from user; show trace |
| show trace                               | Report sql execute profiling info                         |
|
| explain SQL                              | Report sql plan info                                      | explai
n select count(*) from user             |
| explain detail SQL                       | Report sql detail plan info                               | explai
n detail select count(*) from user      |
| explain execute SQL                      | Report sql on physical db plan info                       | explai
n execute select count(*) from user     |
| show sequences                           | Report all sequences status                               |
|
| create sequence NAME [start with COUNT] | Create sequence                                           | create
sequence test start with 0             |
| alter sequence NAME [start with COUNT]  | Alter sequence                                            | alter
sequence test start with 100000         |
| drop sequence NAME                       | Drop sequence                                             | drop s
equence test                            |
| show db status                           | Report size of each physical database                     | show d
b status                                |
| show full db status                      | Report size of each physical table, support like          | show f
ull db status like user                 |
| show ds                                  | Report simple partition db info                           | show d
s                                       |
| show connection                          | Report all connections info                               | show c
onnection                               |
| show trans                               | Report trans info                                         | show t
rans/show trans limit 2                 |
| show full trans                          | Report all trans info                                     | show f
ull trans/show full trans limit 2       |
| show stats                               | Report all requst stats                                   | show s
tats                                    |
| show stc                                 | Report all requst stats by partition                      | show s
tc                                      |
| show htc                                 | Report the CPU/LOAD/MEM/NET/GC stats                      | show h
tc                                      |
| show ccl_rules                           | Report the ccl rule info                                  | show c
cl_rules                                |
| clear ccl_rules                          | Clear the ccl rules                                       | clear
```

```
ccl_rules                              |
| show ccl_triggers                        | Report the ccl trigger info                              | show c
cl_triggers                            |
| clear ccl_triggers                       | Clear the ccl triggers                                   | clear
ccl_triggers                           |
+-----------------------------------------+----------------------------------------------------------+-------
--------------------------------------+
```

8.11.2.2. 规则和拓扑查询语句

本文介绍了规则和拓扑类查询语句。

- SHOW RULE [FROM tablename]
- SHOW FULL RULE [FROM tablename]
- SHOW TOPOLOGY FROM tablename
- SHOW PARTITIONS FROM tablename
- SHOW BROADCASTS
- SHOW DATASOURCES
- SHOW NODE

SHOW RULE [FROM tablename]

使用说明：

- `show rule`：查看数据库下每一个逻辑表的拆分情况；
- `show rule from tablename`：查看数据库下指定逻辑表的拆分情况。

```
mysql> show rule;
+----+------------+-----------+------------------+---------------------+--------------------+---------------
---+---------------------+--------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_K
EY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+----+------------+-----------+------------------+---------------------+--------------------+---------------
---+---------------------+--------------------+
|  0 | k_1        |         0 | k                | hash                | 40                 | k
| hash                | 2                  |
|  1 | k_2        |         0 | k                | hash                | 40                 | k
| hash                | 2                  |
|  2 | sbtest1    |         0 | id               | hash                | 40                 | id
| hash                | 2                  |
|  3 | t1         |         0 | id               | hash                | 40                 | id
| hash                | 4                  |
+----+------------+-----------+------------------+---------------------+--------------------+---------------
---+---------------------+--------------------+
4 rows in set (0.05 sec)
```

重要列详解：

- **BROADCAST**：是否为广播表（0：否，1：是）；
- **DB_PARTITION_KEY**：分库的拆分键，没有分库的话，值为空；
- **DB_PARTITION_POLICY**：分库的拆分策略，取值包括哈希或YYYYMM、YYYYDD、YYYYWEEK等日期策略；
- **DB_PARTITION_COUNT**：分库数；
- **TB_PARTITION_KEY**：分表的拆分键，没有分表的话，值为空；
- **TB_PARTITION_POLICY**：分表的拆分策略，取值包括哈希或MM、DD、MMDD、WEEK等日期策略；
- **TB_PARTITION_COUNT**：分表数。

SHOW FULL RULE [FROM tablename]

查看数据库下逻辑表的拆分规则，比SHOW RULE指令展示的信息更加详细。

```
mysql> show full rule;
+----+------------+-----------+------------+----------------------+----------------------+-----------------
---------------------------------+------------------+----------------------------------------+-------------
---+--------------------+
| ID | TABLE_NAME | BROADCAST | JOIN_GROUP | ALLOW_FULL_TABLE_SCAN | DB_NAME_PATTERN      | DB_RULES_STR
| TB_NAME_PATTERN    | TB_RULES_STR                          | PARTITION_KEYS | DEFAULT_DB_INDEX    |
+----+------------+-----------+------------+----------------------+----------------------+-----------------
---------------------------------+------------------+----------------------------------------+-------------
---+--------------------+
|  0 | k_1        |         0 | NULL       |                    1 | TEST1_{000000}_GROUP | ((#k,1,80#).long
Value().abs() % 80).intdiv(2)     | k_1_cewR_{00}     | ((#k,1,80#).longValue().abs() % 80)     | k
| TEST1_SINGLE_GROUP |
|  1 | k_2        |         0 | NULL       |                    1 | TEST1_{000000}_GROUP | ((#k,1,80#).long
Value().abs() % 80).intdiv(2)     | k_2_1xsQ_{00}     | ((#k,1,80#).longValue().abs() % 80)     | k
| TEST1_SINGLE_GROUP |
|  2 | sbtest1    |         0 | NULL       |                    1 | TEST1_{000000}_GROUP | ((#id,1,80#).lon
gValue().abs() % 80).intdiv(2)    | sbtest1_wO5k_{00} | ((#id,1,80#).longValue().abs() % 80)    | id
| TEST1_SINGLE_GROUP |
|  3 | t1         |         0 | NULL       |                    1 | TEST1_{000000}_GROUP | ((#id,1,160#).lo
ngValue().abs() % 160).intdiv(4) | t1_EMrC_{000}     | ((#id,1,160#).longValue().abs() % 160) | id
| TEST1_SINGLE_GROUP |
+----+------------+-----------+------------+----------------------+----------------------+-----------------
---------------------------------+------------------+----------------------------------------+-------------
---+--------------------+
```

重要列详解：

- **BROADCAST**：是否为广播表（0：否，1：是）；
- **JOIN_GROUP**：保留字段，暂时无意义。
- **ALLOW_FULL_TABLE_SCAN**：分库分表在没有指定分表键值的情况下是否允许查询数据，如果配置为true，此时需要扫描每一个物理表来查找出符合条件的数据，简称为全表扫描；
- **DB_NAME_PATTERN**：DB_NAME_PATTERN中{}之间的0为占位符，执行SQL时会被DB_RULES_STR计算出的值替代，并保持位数。比如，DB_NAME_PATTERN的值为SEQ_{0000}_RDS，DB_RULES_STR的值为[1,2,3,4]，则会产生4个DB_NAME，分别为SEQ_0001_RDS、SEQ_0002_RDS、SEQ_0003_RDS、SEQ_0004_RDS；
- **DB_RULES_STR**：具体的分库规则；
- **TB_NAME_PATTERN**：TB_NAME_PATTERN中{}之间的0为占位符，执行SQL时会被TB_RULES_STR计算出的值替代，并保持位数。比如，TB_NAME_PATTERN的值为table_{00}，TB_RULES_STR的值为[1,2,3,4,5,6,7,8]，则会产生8张表，分别为table_01、table_02、table_03、table_04、table_05、table_06、table_07、table_08；
- **TB_RULES_STR**：分表规则；
- **PARTITION_KEYS**：分库和分表键集合，对于既分库又分表的情形，分库键在前，分表键在后；
- **DEFAULT_DB_INDEX**：单库单表存放的分库。

SHOW TOPOLOGY FROM tablename

查看指定逻辑表的拓扑分布，展示该逻辑表保存在哪些分库中，每个分库下包含哪些分表。

```
mysql> show topology from emp;
+------+-------------------------------------------------+------------+
| ID   | GROUP_NAME                                      | TABLE_NAME |
+------+-------------------------------------------------+------------+
|    0 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS | emp_0      |
|    1 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS | emp_1      |
|    2 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS | emp_0      |
|    3 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS | emp_1      |
|    4 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS | emp_0      |
|    5 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS | emp_1      |
|    6 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS | emp_0      |
|    7 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS | emp_1      |
|    8 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS | emp_0      |
|    9 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS | emp_1      |
|   10 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS | emp_0      |
|   11 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS | emp_1      |
|   12 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS | emp_0      |
|   13 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS | emp_1      |
|   14 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS | emp_0      |
|   15 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS | emp_1      |
+------+-------------------------------------------------+------------+
```

SHOW PARTITIONS FROM tablename

查看分库分表键集合，分库键和分表键之间用逗号分割。如果最终结果有两个值，说明是既分库又分表的情形，第一个是分库键，第二个是分表键。如果结果只有一个值，说明是分库不分表的情形，该值是分库键。

```
mysql> show partitions from emp;
+------------+
| KEYS       |
+------------+
| emp_no,id |
+------------+
1 row in set (0.00 sec)
```

SHOW BROADCASTS

查看广播表列表。

```
mysql> show broadcasts;
+------+------------+
| ID   | TABLE_NAME |
+------+------------+
|    0 | brd2       |
|    1 | brd_tbl    |
+------+------------+
2 rows in set (0.01 sec)
```

SHOW DATASOURCES

查看底层存储信息，包含数据库名、数据库分组名、连接信息、用户名、底层存储类型、读写权重、连接池信息等。

```
mysql> show datasources;
+------+----------------------------+-------------------------------------------------------+----------------------
----------------------------+-----------------------------------------------------------------------------------------
---+-----------+-------+------+------+------+--------------+----------+--------------+---------------+------
-------------------------------------+-------------+--------------+
| ID   | SCHEMA                     | NAME                                                  | GROUP
| URL                                                                                                  | USER      | TYPE   | INI
T | MIN  | MAX  | IDLE_TIMEOUT | MAX_WAIT | ACTIVE_COUNT | POOLING_COUNT | ATOM
| READ_WEIGHT | WRITE_WEIGHT |
+------+----------------------------+-------------------------------------------------------+----------------------
----------------------------+-----------------------------------------------------------------------------------------
---+-----------+-------+------+------+------+--------------+----------+--------------+---------------+------
-------------------------------------+-------------+--------------+
|    0 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0000_iiab_1 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0000_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
00 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0000_iiab | 10          | 10           |
|    1 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab_2 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0001_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
01 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab | 10          | 10           |
|    2 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab_3 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0002_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
02 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab | 10          | 10           |
|    3 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab_4 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0003_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
03 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab | 10          | 10           |
|    4 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab_5 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0004_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
04 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab | 10          | 10           |
|    5 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab_6 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0005_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
05 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab | 10          | 10           |
|    6 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab_7 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0006_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
06 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab | 10          | 10           |
|    7 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab_8 | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0007_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_00
07 | jnkinsea0 | mysql | 0    | 24   | 72   | 15           | 5000     | 0            | 1             | rds1u
r80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab | 10          | 10           |
+------+----------------------------+-------------------------------------------------------+----------------------
----------------------------+-----------------------------------------------------------------------------------------
---+-----------+-------+------+------+------+--------------+----------+--------------+---------------+------
-------------------------------------+-------------+--------------+
```

重要列详解：

- **SCHEMA**：数据库名；
- **GROUP**：数据库分组名。分组的目标是管理多组数据完全相同的数据库，比如通过 RDS（MySQL）进行数据复制后的主备数据库。主要用来解决读写分离、主备切换的问题；
- **URL**：底层RDS（MySQL）的连接信息；
- **TYPE**：底层存储类型，目前只支持RDS（MySQL)；
- **READ_WEIGHT**：读权重。在主实例的读压力比较大的时候，可以通过读写分离功能将读流量进行分流，减轻RDS主实例的压力。PolarDB-X会自动识别读写流量，引导写流量进入RDS主实例，读流量则按配置的权重流向所有RDS实例；
- **WRITE_WEIGHT**：写权重。

SHOW NODE

查看物理库的读写次数（历史累计数据）、读写权重（历史累计数据）。

```
mysql> show node;
+------+-------------------------------------------------+------------------+------------------+----------
-----------+--------------------+
| ID   | NAME                                            | MASTER_READ_COUNT | SLAVE_READ_COUNT | MASTER_RE
AD_PERCENT | SLAVE_READ_PERCENT |
+------+-------------------------------------------------+------------------+------------------+----------
-----------+--------------------+
| 0    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |               12 |                0 | 100%
| 0%                |
| 1    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS |                0 |                0 | 0%
| 0%                |
| 2    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS |                0 |                0 | 0%
| 0%                |
| 3    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS |                0 |                0 | 0%
| 0%                |
| 4    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS |                0 |                0 | 0%
| 0%                |
| 5    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS |                0 |                0 | 0%
| 0%                |
| 6    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS |                0 |                0 | 0%
| 0%                |
| 7    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS |                0 |                0 | 0%
| 0%                |
+------+-------------------------------------------------+------------------+------------------+----------
-----------+--------------------+
8 rows in set (0.01 sec)
```

重要列详解：

- **MASTER_COUNT**：RDS主实例处理的读写查询次数（历史累计数据）；
- **SLAVE_COUNT**：RDS备实例处理的只读查询次数（历史累计数据）；
- **MASTER_PERCENT**：RDS主实例处理的读写查询占比（注意该列显示的是累计的实际数据占比，并不是用户配置的百分比）；
- **SLAVE_PERCENT**：RDS备实例处理的读写查询占比（注意该列显示的是累计的实际数据占比，并不是用户配置的百分比）。

> **说明**
> - 事务中的只读查询会被发送到RDS主实例；
> - 由于 `MASTER_PERCENT` ， `SLAVE_PERCENT` 这两列代表的是历史累计数据，更改读写权重的配比后，这几个数值并不能立即反应最新的读写权重配比，需累计一段比较长的时间才行。

8.11.2.3. 慢SQL相关

本文介绍了慢SQL相关的SHOW语句。

- SHOW [FULL] SLOW [WHERE expr] [limit expr]
- SHOW [FULL] PHYSICAL_SLOW [WHERE expr] [limit expr]
- CLEAR SLOW

SHOW [FULL] SLOW [WHERE expr] [limit expr]

执行时间超过1秒的SQL语句是慢SQL，逻辑慢SQL是指应用发送到PolarDB-X的慢SQL。

- `SHOW SLOW` ：查看自PolarDB-X启动或者上次执行 `CLEAR SLOW` 以来最慢的100条逻辑慢SQL;

> **说明** 此处记录的是最慢的100个，缓存在PolarDB-X 系统中，当实例重启或者执行 `CLEAR SLOW` 时会丢失。

- `SHOW FULL SLOW`：查看实例启动以来记录的所有逻辑慢SQL（持久化到PolarDB-X的内置数据库中）。该记录数有一个上限（具体数值跟购买的实例规格相关），PolarDB-X会滚动删除比较老的慢SQL语句。实例的规格为4C4G时，最多记录10000条慢SQL语句（包括逻辑慢SQL和物理慢SQL）；实例的规格为8C8G时，最多记录20000条慢SQL语句（包括逻辑慢SQL和物理慢SQL），其它规格依此类推。

示例：

```
mysql> show slow where execute_time > 1000 limit 1;
+-----------+---------------------+--------------+------------+-----------+
| HOST      | START_TIME          | EXECUTE_TIME | AFFECT_ROW | SQL       |
+-----------+---------------------+--------------+------------+-----------+
| 127.0.0.1 | 2016-03-16 13:02:57 |         2785 |          7 | show rule |
+-----------+---------------------+--------------+------------+-----------+
1 row in set (0.02 sec)
```

重要列详解：

- **HOST**：来源IP；
- **START_TIME**：执行开始时间；
- **EXECUTE_TIME**：执行时间；
- **AFFECT_ROW**：对于DML语句是影响行数；对于查询语句是返回的记录数。

SHOW [FULL] PHYSICAL_SLOW [WHERE expr] [limit expr]

执行时间超过1秒的SQL语句是慢SQL，物理慢SQL是指PolarDB-X发送到RDS的慢SQL。

- `SHOW PHYSICAL_SLOW`：查看自PolarDB-X启动或者上次执行 `CLEAR SLOW` 以来最慢的100条物理慢SQL（注意，这里记录的是最慢的100个，缓存在PolarDB-X系统中，当实例重启或者执行**CLEAR SLOW**时会丢失）；
- `SHOW FULL PHYSICAL_SLOW`：查看实例启动以来记录的所有物理慢SQL（持久化到PolarDB-X的内置数据库中）。该记录数有一个上限（具体数值跟购买的实例规格相关），PolarDB-X会滚动删除比较老的慢SQL语句。实例的规格如果是4C4G，最多记录10000条慢SQL语句（包括逻辑慢SQL和物理慢SQL）；实例的规格如果是8C8G，最多记录20000条慢SQL语句（包括逻辑慢SQL和物理慢SQL），其它规格依此类推。

示例：

```
mysql> show physical_slow;
+---------------+-----------------------------------+---------------------+--------------+-----------------
-+-------------------------+-----------------------+------------+-----------------+
| GROUP_NAME     | DBKEY_NAME                         | START_TIME          | EXECUTE_TIME | SQL_EXECUTE_TIME
| GETLOCK_CONNECTION_TIME | CREATE_CONNECTION_TIME | AFFECT_ROW | SQL             |
+---------------+-----------------------------------+---------------------+--------------+-----------------
-+-------------------------+-----------------------+------------+-----------------+
| TDDL5_00_GROUP | db218249098_sqa_zmf_tddl5_00_3309 | 2016-03-16 13:05:38 |         1057 |             1011
|                      0 |                     0 |          1 | select sleep(1) |
+---------------+-----------------------------------+---------------------+--------------+-----------------
-+-------------------------+-----------------------+------------+-----------------+
1 row in set (0.01 sec)
```

重要列详解：

- **GROUP_NAME**：数据库分组；
- **START_TIME**：执行开始时间；
- **EXECUTE_TIME**：执行时间；
- **AFFECT_ROW**：对于DML语句是影响行数；对于查询语句是返回的记录数。

CLEAR SLOW

清空自PolarDB-X启动或者上次执行 `CLEAR SLOW` 以来最慢的100条逻辑慢SQL和最慢的100条物理慢SQL。

示例：

```
mysql> clear slow;
Query OK, 0 rows affected (0.00 sec)
```

> **说明** `SHOW SLOW` 和 `SHOW PHYSICAL_SLOW` 展示的是最慢的100个SQL，如果长时间未执行 `CLEAR SLOW` ，可能是非常老的SQL，一般执行过SQL优化之后，建议执行 `CLEAR SLOW` 后，等待系统运行一段时间，再查看慢SQL的优化效果。

8.11.2.4. 统计信息查询

本文介绍了用于查询实时统计信息的语句。

- SHOW [FULL] STATS
- SHOW DB STATUS
- SHOW TABLE STATUS
- SHOW TABLE INFO [name]

SHOW [FULL] STATS

查看整体的统计信息，这些信息都是瞬时值。 注意不同版本的PolarDB-X `SHOW FULL STATS` 的结果是有区别的。

示例：

```
mysql> show stats;
+------+---------+----------+-------------------+------------------+------------------------+---------------
-----+--------+------------+--------------+---------------+----------------+---------------+---------------+
--------------+
| QPS  | RDS_QPS | SLOW_QPS | PHYSICAL_SLOW_QPS | ERROR_PER_SECOND | MERGE_QUERY_PER_SECOND | ACTIVE_CONNECT
IONS | RT(MS) | RDS_RT(MS) | NET_IN(KB/S) | NET_OUT(KB/S) | THREAD_RUNNING | DDL_JOB_COUNT | BACKFILL_ROWS |
CHECKED_ROWS |
+------+---------+----------+-------------------+------------------+------------------------+---------------
-----+--------+------------+--------------+---------------+----------------+---------------+---------------+
--------------+
| 0.00 |    0.00 |     0.00 |              0.00 |             0.00 |                    0.00 |
1 |   0.00 |       0.00 |         0.00 |          0.00 |              1 |             0 |             0 |
0 |
+------+---------+----------+-------------------+------------------+------------------------+---------------
-----+--------+------------+--------------+---------------+----------------+---------------+---------------+
--------------+
mysql> show full stats;
+------+---------+----------+-------------------+------------------+----------------------+-----------------
-------+--------------------+------------------------------+--------+------------+--------------+-----------
----+----------------+----------------------+-----------------+----------------------------+----------------
-------+------------------------------+-------------------------+--------------------------+----------------
-----+-------+---------+-------------+------------+
| QPS  | RDS_QPS | SLOW_QPS | PHYSICAL_SLOW_QPS | ERROR_PER_SECOND | VIOLATION_PER_SECOND | MERGE_QUERY_PER_
SECOND | ACTIVE_CONNECTIONS | CONNECTION_CREATE_PER_SECOND | RT(MS) | RDS_RT(MS) | NET_IN(KB/S) | NET_OUT(KB
/S) | THREAD_RUNNING | HINT_USED_PER_SECOND | HINT_USED_COUNT | AGGREGATE_QUERY_PER_SECOND | AGGREGATE_QUERY
_COUNT | TEMP_TABLE_CREATE_PER_SECOND | TEMP_TABLE_CREATE_COUNT | MULTI_DB_JOIN_PER_SECOND | MULTI_DB_JOIN_C
OUNT | CPU   | FREEMEM | FULLGCCOUNT | FULLGCTIME |
+------+---------+----------+-------------------+------------------+----------------------+-----------------
-------+--------------------+------------------------------+--------+------------+--------------+-----------
----+----------------+----------------------+-----------------+----------------------------+----------------
-------+------------------------------+-------------------------+--------------------------+----------------
-----+-------+---------+-------------+------------+
| 1.63 |    1.68 |     0.03 |              0.03 |             0.02 |                 0.00 |
0.00 |                  6 |                         0.01 | 157.13 |      51.14 |       134.33 |          1.2
1 |              1 |                 0.00 |              54 |                        0.00 |
663 |                         0.00 |                     512 |                      0.00 |                 51
6 | 0.09% |   6.96% |       76446 |   21326906 |
+------+---------+----------+-------------------+------------------+----------------------+-----------------
-------+--------------------+------------------------------+--------+------------+--------------+-----------
----+----------------+----------------------+-----------------+----------------------------+----------------
-------+------------------------------+-------------------------+--------------------------+----------------
-----+-------+---------+-------------+------------+
1 row in set (0.01 sec)
```

重要列说明：

- **QPS**：应用到PolarDB-X的QPS，通常称为逻辑QPS；
- **RDS_QPS**：PolarDB-X到RDS的QPS，通常称为物理QPS；
- **ERROR_PER_SECOND**：每秒的错误数，包含SQL语法错误，主键冲突，系统错误，连通性错误等各类错误总和；
- **VIOLATION_PER_SECOND**：每秒的主键或者唯一键冲突；
- **MERGE_QUERY_PER_SECCOND**：通过分库分表，从多表中进行的查询；
- **ACTIVE_CONNECTIONS**：正在使用的连接；
- **CONNECTION_CREATE_PER_SECCOND**：每秒创建的连接数；
- **RT(MS)**：应用到PolarDB-X的响应时间，通常称为逻辑RT（响应时间）；
- **RDS_RT(MS)**：PolarDB-X到RDS/MySQL的响应时间，通常称为物理RT；
- **NET_IN(KB/S)**：PolarDB-X收到的网络流量；
- **NET_OUT(KB/S)**：PolarDB-X输出的网络流量；
- **THREAD_RUNNING**：正在运行的线程数；

- **HINT_USED_PER_SECOND**：每秒带HINT的查询的数量；
- **HINT_USED_COUNT**：启动到现在带HINT的查询总量；
- **AGGREGATE_QUERY_PER_SECCOND**：每秒聚合查询的频次；
- **AGGREGATE_QUERY_COUNT**：聚合查询总数（历史累计数据）；
- **TEMP_TABLE_CREATE_PER_SECCOND**：每秒创建的临时表的数量；
- **TEMP_TABLE_CREATE_COUNT**：启动到现在创建的临时表总数量；
- **MULTI_DB_JOIN_PER_SECCOND**：每秒跨库JOIN的数量；
- **MULTI_DB_JOIN_COUNT**：启动到现在跨库JOIN的总量。

SHOW DB STATUS

用于查看物理库容量/性能信息，所有返回值为实时信息。 容量信息通过MySQL系统表获得，与真实容量情况可能有差异。

示例：

```
mysql> show db status;
+------+---------------------------+--------------------+------------------+------------+--------+---------
-------+
| ID   | NAME                      | CONNECTION_STRING  | PHYSICAL_DB      | SIZE_IN_MB | RATIO  | THREAD_R
UNNING |
+------+---------------------------+--------------------+------------------+------------+--------+---------
-------+
|    1 | drds_db_1516187088365daui | 100.100.64.1:59077 | TOTAL            |  13.109375 | 100%   | 3
|
|    2 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 |   1.578125 | 12.04% |
|
|    3 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 |     1.4375 | 10.97% |
|
|    4 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 |     1.4375 | 10.97% |
|
|    5 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 |     1.4375 | 10.97% |
|
|    6 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 |   1.734375 | 13.23% |
|
|    7 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 |   1.734375 | 13.23% |
|
|    8 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 |   2.015625 | 15.38% |
|
|    9 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 |   1.734375 | 13.23% |
|
+------+---------------------------+--------------------+------------------+------------+--------+---------
-------+
```

重要列说明：

- **NAME**：代表一个PolarDB-X DB，此处显示的是PolarDB-X内部标记，与PolarDB-X DB名称不同；
- **CONNECTION_STRING**：分库的连接信息；
- **PHYSICAL_DB**：分库名称， `TOTAL` 行代表一个PolarDB-X DB中所有分库容量的总和；
- **SIZE_IN_MB**：分库中数据占用的空间，单位为MB；
- **RATIO**：单个分库数据量在当前PolarDB-X DB总数据量中的占比；
- **THREAD_RUNNING**：物理数据库实例当前正在执行的线程情况，含义与MySQL语句 `SHOW GLOBAL STATUS` 返回值的含义相同，详情请参考 MySQL 文档。

SHOW TABLE STATUS

获取表的信息，该指令聚合了底层各个物理分表的数据。

示例：

```
mysql> SHOW TABLE STATUS;
+---------+--------+---------+------------+------+----------------+-------------+-----------------+---------
-----+-----------+----------------+--------------------+-------------+------------+--------------------+---
-------+----------------+---------+
| NAME    | ENGINE | VERSION | ROW_FORMAT | ROWS | AVG_ROW_LENGTH | DATA_LENGTH | MAX_DATA_LENGTH | INDEX_LE
NGTH | DATA_FREE | AUTO_INCREMENT | CREATE_TIME         | UPDATE_TIME | CHECK_TIME | COLLATION          | CH
ECKSUM | CREATE_OPTIONS | COMMENT |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+---------
-----+-----------+----------------+--------------------+-------------+------------+--------------------+---
-------+----------------+---------+
| sbtest1 | InnoDB |      10 | Dynamic    |    0 |              0 |     1310720 |               0 |
0 |         0 |              0 | 2021-07-20 15:39:37 | NULL        | NULL       | utf8mb4_general_ci | NULL
|                |         |
| t1      | InnoDB |      10 | Dynamic    |    0 |              0 |     2621440 |               0 |      262
1440 |         0 |         200000 | 2021-07-26 20:11:15 | NULL        | NULL       | utf8mb4_general_ci | NU
LL     |                |         |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+---------
-----+-----------+----------------+--------------------+-------------+------------+--------------------+---
-------+----------------+---------+
```

重要列详解：

- **NAME**：表名称；
- **ENGINE**：表的存储引擎；
- **VERSION**：表的存储引擎的版本；
- **ROW_FORMAT**：行格式，主要是Dynamic、Fixed、Compressed这三种格式。动态（Dynamic）行的行长度可变，例如VARCHAR或BLOB类型字段；固定（Fixed）行是指行长度不变，例如CHAR和INTEGER类型字段；
- **ROWS**：表中的行数；
- **AVG_ROW_LENGTH**：平均每行包括的字节数；
- **DATA_LENGTH**：整个表的数据量（单位：字节）；
- **MAX_DATA_LENGTH**：表可以容纳的最大数据量；
- **INDEX_LENGTH**：索引占用磁盘的空间大小；
- **CREATE_TIME**：表的创建时间；
- **UPDATE_TIME**：表的最近更新时间；
- **COLLATION**：表的默认字符集和字符排序规则；
- **CREATE_OPTIONS**：指表创建时的其他所有选项。

SHOW TABLE INFO [name]

获取各个分表的数据量信息。

示例：

```
mysql> show table info  sbtest1;
+----+--------------+----------------+------------+
| ID | GROUP_NAME   | TABLE_NAME     | SIZE_IN_MB |
+----+--------------+----------------+------------+
|  0 | test1_000000 | sbtest1_wo5k_00 | 0.01562500 |
|  1 | test1_000000 | sbtest1_wo5k_01 | 0.01562500 |
|  2 | test1_000005 | sbtest1_wo5k_10 | 0.01562500 |
|  3 | test1_000005 | sbtest1_wo5k_11 | 0.01562500 |
|  4 | test1_000010 | sbtest1_wo5k_20 | 0.01562500 |
|  5 | test1_000010 | sbtest1_wo5k_21 | 0.01562500 |
|  6 | test1_000015 | sbtest1_wo5k_30 | 0.01562500 |
|  7 | test1_000015 | sbtest1_wo5k_31 | 0.01562500 |
|  8 | test1_000020 | sbtest1_wo5k_40 | 0.01562500 |
|  9 | test1_000020 | sbtest1_wo5k_41 | 0.01562500 |
| 10 | test1_000025 | sbtest1_wo5k_50 | 0.01562500 |
| 11 | test1_000025 | sbtest1_wo5k_51 | 0.01562500 |
| 12 | test1_000030 | sbtest1_wo5k_60 | 0.01562500 |
| 13 | test1_000030 | sbtest1_wo5k_61 | 0.01562500 |
| 14 | test1_000035 | sbtest1_wo5k_70 | 0.01562500 |
| 15 | test1_000035 | sbtest1_wo5k_71 | 0.01562500 |
+----+--------------+----------------+------------+
```

重要列详解：

- **ID**：标识；
- **GROUP_NAME**：分库名；
- **TABLE_NAME**：物理分表名；
- **SIZE_IN_MB**：表大小；

8.11.2.5. SHOW PROCESSLIST

本文介绍如何使用SHOW PROCESSLIST和SHOW PHYSICAL_PROCESSLIST语句。

SHOW PROCESSLIST

您可以使用如下语句查看PolarDB-X中的连接与正在执行的SQL等信息：

- 语法

```
SHOW PROCESSLIST
```

- 示例

```
mysql> SHOW PROCESSLIST\G
     ID: 1971050
   USER: admin
   HOST: 111.111.111.111:4303
     DB: drds_test
COMMAND: Query
   TIME: 0
  STATE:
   INFO: show processlist
1 row in set (0.01 sec)
```

参数	说明
ID	本次连接的ID，为一个Long型数字。
USER	建立此连接所使用的用户名。
HOST	建立此连接的机器的IP与端口。

参数	说明
DB	此连接所访问的数据库名称。
COMMAND	目前有如下两种取值： ○ Query：当前连接正在执行SQL语句。 ○ Sleep：当前连接正处于空闲状态。
TIME	连接处于当前状态持续的时间。 ○ 当COMMAND为Query时，代表此连接上正在执行的SQL已经执行的时间。 ○ 当COMMAND为Sleep时，代表此连接空闲的时间。
STATE	目前无意义，恒为空值。
INFO	○ 当COMMAND为Query时，为此连接上正在执行的SQL的内容。 **说明** 当不带FULL参数时，最多返回正在执行的SQL的前 30 个字符。当带FULL参数时，最多返回正在执行的SQL的前1000个字符。 ○ 当COMMAND为Sleep时，为空值，无意义。

SHOW PHYSICAL_PROCESSLIST

您可以使用如下指令查看所有正在执行的物理SQL信息：

- 语法

```
SHOW PHYSICAL_PROCESSLIST
```

说明 当SQL比较长的时候，使用 `SHOW PHYSICAL_PROCESSLIST` 语句返回得到的SQL会被截断，这时可以使用 `SHOW FULL PHYSICAL_PROCESSLIST` 语句获取完整SQL。

- 示例

```
mysql> SHOW PHYSICAL_PROCESSLIST\G
*************************** 1. row ***************************
           ID: 0-0-521414
         USER: tddl5
           DB: tddl5_00
      COMMAND: Query
         TIME: 0
        STATE: init
         INFO: show processlist
*************************** 2. row ***************************
           ID: 0-0-521570
         USER: tddl5
           DB: tddl5_00
      COMMAND: Query
         TIME: 0
        STATE: User sleep
         INFO: /*DRDS /88.88.88.88/b67a0e4d8800000/ */ select sleep(1000)
2 rows in set (0.01 sec)
```

说明

- 返回结果中每一列的含义与MySQL的 `SHOW PROCESSLIST` 指令等价，详情请参见 SHOW PROCESSLIST Syntax。
- 但与MySQL不同，PolarDB-X返回的物理连接的ID列为一个字符串，并非一个数字。

8.11.2.6. SHOW GLOBAL INDEX

PolarDB-X支持使用全局二级索引，本文将介绍如何使用SHOW GLOBAL INDEX命令查看已创建或创建中的全局二级索引。

语法

```
SHOW GLOBAL {INDEX | INDEXES} [FROM [schema_name.]tbl_name]
```

`schema_name` 和 `tbl_name` 是可选的，用于过滤表名或查看其它数据库上表的信息。

```
show global index; # 查询当前数据库上所有表的全局二级索引信息
show global index from xxx_tb; # 查询当前数据库上 xxx_tb 的全局二级索引信息
show global index from xxx_db.xxx_tb; # 查询 xxx_db 上 xxx_tb 的全局二级索引信息（跨库查询）
```

示例

```
mysql> show global index;
+---------------------+----------------------+------------+-------------------------------+-----------------
-----------+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+------------+------------------+---------
------------+--------------------+------------------+---------------------+--------------------+----------+
| SCHEMA              | TABLE                | NON_UNIQUE | KEY_NAME                      | INDEX_NAMES
| COVERING_NAMES
| INDEX_TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION
_POLICY | TB_PARTITION_COUNT | STATUS   |
+---------------------+----------------------+------------+-------------------------------+-----------------
-----------+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+------------+------------------+---------
------------+--------------------+------------------+---------------------+--------------------+----------+
| XXXX_DRDS_LOCAL_APP | full_gsi_ddl_renamed | 1          | g_i_c_ddl_c_blob_long_renamed | c_blob_long
| id, c_bit_1, c_bit_8, c_bit_16, c_bit_32, c_bit_64, c_tinyint_1, c_tinyint_1_un, c_tinyint_4, c_tinyint_4_
un, c_tinyint_8, c_tinyint_8_un, c_smallint_16, c_smallint_16_un, c_mediumint_1, c_mediumint_24, c_mediumint
_24_un, c_int_1, c_int_32, c_int_32_un, c_bigint_1, c_bigint_64, c_bigint_64_un, c_decimal, c_decimal_pr, c_
float, c_float_pr, c_float_un, c_double, c_double_pr, c_double_un, c_date, c_datetime, c_datetime_3, c_datet
ime_6, c_timestamp_1, c_timestamp_3, c_time, c_time_1, c_time_3, c_time_6, c_year, c_year_4, c_char, c_varch
ar, c_binary, c_varbinary, c_blob_tiny, c_blob_medium, c_text_tiny, c_text, c_text_medium, c_text_long, c_en
um, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_multilinestring, c_multipolygon, c_geom
etrycollection, c_geometory                            | NULL       | c_blob_long      | HASH
| 4                  | c_blob_long      | HASH                | 3                  | PUBLIC   |
| XXXX_DRDS_LOCAL_APP | full_gsi_ddl_renamed | 1          | g_i_c_ddl_c_mediumint_1       | c_mediumint_1
| id, c_bit_1, c_bit_8, c_bit_16, c_bit_32, c_bit_64, c_tinyint_1, c_tinyint_1_un, c_tinyint_4, c_tinyint_4_
un, c_tinyint_8, c_tinyint_8_un, c_smallint_16, c_smallint_16_un, c_mediumint_24, c_mediumint_24_un, c_int_1
, c_int_32, c_int_32_un, c_bigint_1, c_bigint_64, c_bigint_64_un, c_decimal, c_decimal_pr, c_float, c_float_
pr, c_float_un, c_double, c_double_pr, c_double_un, c_date, c_datetime, c_datetime_3, c_datetime_6, c_timest
amp_1, c_timestamp_3, c_time, c_time_1, c_time_3, c_time_6, c_year, c_year_4, c_char, c_varchar, c_binary, c
_varbinary, c_blob_tiny, c_blob_medium, c_blob_long, c_text_tiny, c_text, c_text_medium, c_text_long, c_enum
, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_multilinestring, c_multipolygon, c_geomet
rycollection, c_geometory, c_smallint_1, c_timestamp_6 | NULL       | c_mediumint_1     | HASH
| 4                  | c_mediumint_1    | HASH                | 3                  | PUBLIC   |
| XXXX_DRDS_LOCAL_APP | full_gsi_ddl_renamed | 1          | g_i_c_ddl_c_smallint_16_un    | c_smallint_16_un
```

```
| XXXX_DRDS_LOCAL_APP | full_gsi_ddl_renamed | 1          | g_i_c_ddl_c_smallint_16_un     | c_smallint_16_un
, c_time_1 | id, c_bit_1, c_bit_8, c_bit_16, c_bit_32, c_bit_64, c_tinyint_1, c_tinyint_1_un, c_tinyint_4, c
_tinyint_4_un, c_tinyint_8, c_tinyint_8_un, c_smallint_16, c_mediumint_1, c_mediumint_24, c_mediumint_24_un,
c_int_1, c_int_32, c_int_32_un, c_bigint_1, c_bigint_64, c_bigint_64_un, c_decimal, c_decimal_pr, c_float, c
_float_pr, c_float_un, c_double, c_double_pr, c_double_un, c_date, c_datetime, c_datetime_3, c_datetime_6, c
_timestamp_1, c_timestamp_3, c_time, c_time_3, c_time_6, c_year, c_year_4, c_char, c_varchar, c_binary, c_va
rbinary, c_blob_tiny, c_blob_medium, c_blob_long, c_text_tiny, c_text, c_text_medium, c_text_long, c_enum, c
_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_multilinestring, c_multipolygon, c_geometryc
ollection, c_geometory                                          | NULL       | c_smallint_16_un | HASH
| 4                  | c_smallint_16_un | HASH                | 3                  | PUBLIC    |
| XXXX_DRDS_LOCAL_APP | t_order              | 0          | g_i_seller                     | seller_id
| id, order_id
| HASH       | seller_id        | HASH                | 4                  | seller_id        | HASH
| 2                  | CREATING |
+--------------------+----------------------+------------+--------------------------------+-----------------
-----------+-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------+------------+------------------+---------
------------+--------------------+------------------+------------------+--------------------+----------+
4 rows in set (0.01 sec)
```

列名说明

列名	说明
SCHEMA	库名
TABLE	表名
NON_UNIQUE	是否为唯一约束全局二级索引，取值范围如下： • 1：普通全局二级索引 • 0：唯一约束全局二级索引
KEY_NAME	索引名
INDEX_NAMES	索引列
COVERING_NAMES	覆盖列
INDEX_TYPE	索引类型，取值范围如下： • NULL（即未指定） • BTREE • HASH
DB_PARTITION_KEY	分库拆分键
DB_PARTITION_POLICY	分库拆分函数
DB_PARTITION_COUNT	分库数量
TB_PARTITION_KEY	分表拆分键
TB_PARTITION_POLICY	分表拆分函数
TB_PARTITION_COUNT	分表数

列名	说明
STATUS	索引的当前状态，取值范围如下： • CREATING • DELETE_ONLY • WRITE_ONLY • WRITE_REORG • PUBLIC • ABSENT

8.11.2.7. SHOW INDEX

您可以使用SHOW INDEX语句查看PolarDB-X表上的局部索引和全局索引信息。

语法

```
SHOW {INDEX | INDEXES | KEYS}
    {FROM | IN} tbl_name
    [{FROM | IN} db_name]
    [WHERE expr]
```

示例

```
mysql> show index from t_order;
+--------------+------------+-----------+--------------+----------------+-----------+-------------+----------
-+--------+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME  | SEQ_IN_INDEX | COLUMN_NAME    | COLLATION | CARDINALITY | SUB_PART | PA
CKED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+--------------+------------+-----------+--------------+----------------+-----------+-------------+----------
-+--------+------+------------+----------+---------------+
| t_order |          0 | PRIMARY   |            1 | id             | A         |           0 |     NULL | NU
LL   |      | BTREE      |          |               |
| t_order |          1 | l_i_order |            1 | order_id       | A         |           0 |     NULL | NU
LL   | YES  | BTREE      |          |               |
| t_order |          0 | g_i_buyer |            1 | buyer_id       | NULL      |           0 |     NULL | NU
LL   | YES  | GLOBAL     | INDEX    |               |
| t_order |          1 | g_i_buyer |            2 | id             | NULL      |           0 |     NULL | NU
LL   |      | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_buyer |            3 | order_id       | NULL      |           0 |     NULL | NU
LL   | YES  | GLOBAL     | COVERING |               |
| t_order |          1 | g_i_buyer |            4 | order_snapshot | NULL      |           0 |     NULL | NU
LL   | YES  | GLOBAL     | COVERING |               |
+--------------+------------+-----------+--------------+----------------+-----------+-------------+----------
-+--------+------+------------+----------+---------------+
6 rows in set (0.01 sec)
```

列名说明

列名	说明
TABLE	表名
NON_UNIQUE	是否为唯一约束全局二级索引，取值范围如下： • 1：普通全局二级索引 • 0：唯一约束全局二级索引
KEY_NAME	索引名

列名	说明
SEQ_IN_INDEX	索引列在索引中的序号，取值从1开始。
COLUMN_NAME	索引列名。
COLLATION	排序方式，取值范围如下： • A：升序 • D：降序 • NULL：不排序
CARDINALITY	预计的唯一值数目
SUB_PART	索引前缀（NULL索引前缀为整个列）。
PACKED	字段压缩信息（NULL表示没有压缩）。
NULL	是否允许空。
INDEX_TYPE	索引类型，取值范围如下： • NULL（即未指定） • BTREE • HASH
COMMENT	索引信息，取值范围如下： • NULL：局部索引 • INDEX：全局二级索引的索引列 • COVERING：全局二级索引的覆盖列
INDEX_COMMENT	其他信息

8.11.2.8. SHOW METADATA LOCK

本文将介绍如何在PolarDB-X上使用SHOW METADATA LOCK语句查询持有锁的事务。

背景信息

PolarDB-X在创建全局二级索引时使用了内建的METADATA LOCK，保证事务以及数据的一致性。在已有表上建立全局二级索引通常需要较长的时间，若此时同时存在持有锁的事务在运行则可能出现SCHEMA变更等待事务完成的情况。此时您可以使用SHOW METADATA LOCK语句查询持有锁的事务以及对应正在执行的SQL语句，方便您排查阻塞SCHEMA变更的长时间事务。

> **说明** PolarDB-X支持Online Schema Change，添加全局二级索引过程中，会发生4次元数据版本切换，其中有两次会先获取METADATA LOCK的写锁加载元数据完成后立即解锁，其余的时间均不会持有METADATA LOCK的写锁。

语法

```
SHOW METADATA {LOCK | LOCKS} [schema_name[.table_name]]
```

`schema_name` 和 `tbl_name` 是可选的，用于过滤显示的数据库名或表名。

```
show metadata lock; # 显示该节点上所有持有metadata lock的连接
show metadata lock xxx_db; # 显示该节点上 xxx_db 中所有持有metadata lock的连接
show metadata lock xxx_db.tb_name; # 显示该节点上 xxx_db 中 tb_name 上所有持有metadata lock的连接
```

示例

```
mysql> show metadata lock;
+---------+--------+----------------+--------------------+-------------+-----------------+-------------
---+----------+----------------------------------------+-------------------------------------------------+
| CONN_ID | TRX_ID | TRACE_ID       | SCHEMA             | TABLE       | TYPE            | DURATION
| VALIDATE | FRONTEND                             | SQL                                             |
+---------+--------+----------------+--------------------+-------------+-----------------+-------------
---+----------+----------------------------------------+-------------------------------------------------+
| 4       |      0 | f88cf71cbc00001 | XXXX_DRDS_LOCAL_APP | full_gsi_ddl | MDL_SHARED_WRITE | MDL_TRANSACTI
ON |        1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54788 | insert into `full_gsi_ddl` (id) VALUE (null); |
| 5       |      0 | f88cf71cbc00000 | XXXX_DRDS_LOCAL_APP | full_gsi_ddl | MDL_SHARED_WRITE | MDL_TRANSACTI
ON |        1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54789 | insert into `full_gsi_ddl` (id) VALUE (null); |
+---------+--------+----------------+--------------------+-------------+-----------------+-------------
---+----------+----------------------------------------+-------------------------------------------------+
2 rows in set (0.00 sec)
```

说明 该语句仅用于显示已持有锁的连接，不显示等待锁的连接。

列名说明

列名	说明
CONN_ID	持有锁的连接ID
TRX_ID	持有锁的事务ID
TRACE_ID	持有锁的SQL的跟踪 ID
SCHEMA	库名
TABLE	表名
TYPE	持有锁类型
DURATION	持有锁的周期
VALIDATE	是否有效
FRONTEND	前端连接信息
SQL	持有锁的SQL语句

8.11.2.9. SHOW TABLEGROUP

用于查看当前数据库实例的表组信息，包括每个表组的分区情况，每个表组里都包括哪些表等信息。

语法

```
show_tablegroup_stmt:
    SHOW tablegroup [WHERE where_condition]
```

示例

查找库名为 `test_db` 下的所有 `tablegroup` 信息。

```
show tablegroup where schema_name='test_db';
```

8.11.3. SET

8.11.3.1. SET语句变量设置

您可以通过SET语句设置各类变量，包括用户自定义的变量、session变量和global变量。

语法

```
SET variable = expr [, variable = expr] ...
variable: {
    user_var_name
  | {GLOBAL | @@GLOBAL.} system_var_name
  | [SESSION | @@SESSION. | @@] system_var_name
}
```

注意事项

使用SET GLOBAL设置全局变量时，PolarDB-X会将其进行持久化，实例重启后仍然生效。此外，SET GLOBAL设置成功后，所有已有的连接都会生效。

示例

```
mysql> SET @foo='bar';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @foo;
+------+
| @foo |
+------+
| bar  |
+------+
1 row in set (0.01 sec)
mysql> SET @@time_zone='+09:00';
Query OK, 0 rows affected (0.01 sec)
mysql> SELECT @@time_zone;
+--------------+
| @@time_zone |
+--------------+
| +09:00       |
+--------------+
1 row in set (0.00 sec)
mysql> SET GLOBAL time_zone='+09:00';
Query OK, 0 rows affected (0.04 sec)
mysql> SELECT @@GLOBAL.time_zone;
+--------------------+
| @@global.time_zone |
+--------------------+
| +09:00             |
+--------------------+
1 row in set (0.02 sec)
```

8.11.3.2. SET CHARACTER SET

您可以使用SET CHARACTER SET语句设置当前客户端与服务端通信使用的字符集。该语句会将 `character_set_client` 和 `character_set_results` 设置为给定值，将 `character_set_connection` 设置为 `character_set_database` 的值。

语法

```
SET {CHARACTER SET | CHARSET}
    {'charset_name' | DEFAULT}
```

其中，`charset_name`可不加引号。

示例

```
mysql> SHOW SESSION VARIABLES LIKE 'character\_set\_%';
+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| character_set_client     | utf8mb4 |
| character_set_connection | utf8mb4 |
| character_set_database   | utf8    |
| character_set_filesystem | binary  |
| character_set_results    | utf8mb4 |
| character_set_server     | utf8    |
| character_set_system     | utf8    |
+--------------------------+---------+
7 rows in set (0.01 sec)
mysql> SET CHARSET 'big5';
Query OK, 0 rows affected (0.06 sec)
mysql> SHOW SESSION VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | big5   |
| character_set_connection | utf8   |
| character_set_database   | utf8   |
| character_set_filesystem | binary |
| character_set_results    | big5   |
| character_set_server     | utf8   |
| character_set_system     | utf8   |
+--------------------------+--------+
7 rows in set (0.21 sec)
```

8.11.3.3. SET NAMES

您可以使用SET NAMES语句设置字符集。该语句会
将`character_set_client`，`character_set_connection`和`character_set_results`设置为给定的字符集。

语法

```
SET NAMES {'charset_name'
    [COLLATE 'collation_name'] | DEFAULT}
```

其中，`charset_name`和`collation_name`可不加引号，当用户给出`collation_name`时，字符序也将被设置为给定值。

示例

```
mysql> SHOW SESSION VARIABLES LIKE 'character\_set\_%';
+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| character_set_client     | utf8mb4 |
| character_set_connection | utf8mb4 |
| character_set_database   | utf8    |
| character_set_filesystem | binary  |
| character_set_results    | utf8mb4 |
| character_set_server     | utf8    |
| character_set_system     | utf8    |
+--------------------------+---------+
7 rows in set (0.03 sec)
mysql> SET NAMES gb18030 COLLATE gb18030_chinese_ci;
Query OK, 0 rows affected (0.02 sec)
mysql> SHOW SESSION VARIABLES LIKE 'character\_set\_%';
+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| character_set_client     | gb18030 |
| character_set_connection | gb18030 |
| character_set_database   | utf8    |
| character_set_filesystem | binary  |
| character_set_results    | gb18030 |
| character_set_server     | utf8    |
| character_set_system     | utf8    |
+--------------------------+---------+
7 rows in set (0.02 sec)
```

8.11.4. CHECK TABLE

本文介绍了CHECK TABLE语句的用法。

CHECK TABLE用于对数据表进行检查，主要用于DDL建表失败的情形。

- 对于拆分表，检查底层物理分表是否有缺失的情况，底层的物理分表的列和索引是否一致；
- 对于单库单表，检查表是否存在。

语法

```
CHECK TABLE tbl_name
```

示例

```
mysql>  check table tddl_mgr_log;
+-------------------------+-------+----------+----------+
| TABLE                   | OP    | MSG_TYPE | MSG_TEXT |
+-------------------------+-------+----------+----------+
| TDDL5_APP.tddl_mgr_log | check | status   | OK       |
+-------------------------+-------+----------+----------+
1 row in set (0.56 sec)
mysql> check table tddl_mg;
+-------------------+-------+----------+---------------------------------------+
| TABLE             | OP    | MSG_TYPE | MSG_TEXT                              |
+-------------------+-------+----------+---------------------------------------+
| TDDL5_APP.tddl_mg | check | Error    | Table 'tddl5_00.tddl_mg' doesn't exist |
+-------------------+-------+----------+---------------------------------------+
1 row in set (0.02 sec)
```

8.11.5. CHECK GLOBAL INDEX

您可以使用CHECK GLOBAL INDEX语句检查主表和索引表的数据是否完全一致，并修订不一致的数据。

语法

```
CHECK GLOBAL INDEX gsi_name [ON tbl_name] [extra_cmd]
```

参数	说明
gsi_name	需要校验的全局二级索引名。
tbl_name	全局二级索引所在的主表，非必选。若您输入具体主表名称，将会检查全局二级索引表和主表间的索引关系是否正确。
extra_cmd	保留的额外指令，目前支持如下指令： • -：不指定任何关键字时，即表示仅检查全局二级索引。 • SHOW：显示指定GSI表的最近一次校验或订正的结果。 • CORRECTION_BASED_ON_PRIMARY：对全局二级索引表中的数据进行订正，以主表为基准。

说明

- 对全局二级索引表中的数据进行校验或订正时会占用一定的系统资源，特别是订正操作会对主表或索引表内数据分批加锁并订正，建议您在业务低谷期进行操作。更多关于GSI的使用方式，请参见全局二级索引。
- 对大表的GSI校验可能会花费较多的时间，可以使用HINT指定PURE_ASYNC_DDL_MODE，以纯异步模式执行DDL语句。

示例

- 您可以使用如下语句进行校验：

  ```
  mysql> CHECK GLOBAL INDEX `g_i_check`;
  ```

  - 若校验没有发现错误，则返回如下结果：

    ```
    +-------------+------------+--------+-------------+-----------------------------+
    | GSI_TABLE   | ERROR_TYPE | STATUS | PRIMARY_KEY | DETAILS                     |
    +-------------+------------+--------+-------------+-----------------------------+
    | `g_i_check` | SUMMARY    | --     | --          | OK (7025/7025 rows checked) |
    +-------------+------------+--------+-------------+-----------------------------+
    1 row in set (1.40 sec)
    ```

- 若校验发现错误，则返回如下结果：

```
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------+
| GSI_TABLE   | ERROR_TYPE | STATUS | PRIMARY_KEY | DETAILS
|
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------+
| `g_i_check` | ORPHAN     | FOUND  | (100722)    | {"GSI":{"id":100722,"c_timestamp_6":"2000-01-01 00:0
0:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_binary
":"OTkAAAAAAAAAA==","c_int_32":271}}
|
| `g_i_check` | CONFLICT   | FOUND  | (108710)    | {"Primary":{"id":108710,"c_timestamp_6":"2000-01-01
00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_ye
ar":"2000","c_int_32":255},"GSI":{"c_int_32_un":123456,"id":108710,"c_timestamp_6":"2000-01-01 00:00:00.
000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_year":"2000
","c_int_32":255}} |
| `g_i_check` | MISSING    | FOUND  | (100090)    | {"Primary":{"id":100090,"c_timestamp_6":"2000-01-01
00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_bl
ob_tiny":"YeS4reWbvWE=","c_int_32":280}}
|
| `g_i_check` | SUMMARY    | --     | --          | 3 error found (7025/7025 rows checked)
|
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------+
4 rows in set (1.92 sec)
```

> **说明** 如果数据存在多种错误，同一行数据会报多个ERROR_TYPE值。

- 您可以使用如下语句进行订正：

```
mysql> CHECK GLOBAL INDEX g_i_check CORRECTION_BASED_ON_PRIMARY;
```

返回结果如下：

```
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------+
| GSI_TABLE   | ERROR_TYPE | STATUS | PRIMARY_KEY | DETAILS
|
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------+
| `g_i_check` | SUMMARY    | --     | --          | Done. Use SQL: { CHECK GLOBAL INDEX `g_i_check` SHOW;
} to get result. |
+-------------+------------+--------+-------------+-----------------------------------------------------
-----------------+
1 row in set (1.40 sec)
```

- 您可以使用如下语句查看最近一次校验或订正的报告：

```
mysql> CHECK GLOBAL INDEX `g_i_check` SHOW;
```

返回结果如下：

```
+-------------+------------+----------+-------------+-----------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------+
| GSI_TABLE   | ERROR_TYPE | STATUS   | PRIMARY_KEY | DETAILS
|
+-------------+------------+----------+-------------+-----------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------+
| `g_i_check` | MISSING    | REPAIRED | (100090)    | {"Primary":{"id":100090,"c_timestamp_6":"2000-01-01
00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_blob
_tiny":"YeS4reWbvWE=","c_int_32":280}}
|
| `g_i_check` | CONFLICT   | REPAIRED | (108710)    | {"Primary":{"id":108710,"c_timestamp_6":"2000-01-01
00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_year
":"2000","c_int_32":255},"GSI":{"c_int_32_un":123456,"id":108710,"c_timestamp_6":"2000-01-01 00:00:00.0000
00","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_year":"2000","c_i
nt_32":255}} |
| `g_i_check` | ORPHAN     | REPAIRED | (100722)    | {"GSI":{"id":100722,"c_timestamp_6":"2000-01-01 00:0
0:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_binary":
"OTkAAAAAAAAAAA==","c_int_32":271}}
|
| `g_i_check` | SUMMARY    | --       | --          | 3 error found (7025/7026 rows checked.) Finish time:
2020-01-13 14:41:51.0
|
+-------------+------------+----------+-------------+-----------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-------------+
4 rows in set (0.02 sec)
```

列名说明

列名	说明
GSI_TABLE	全局二级索引名。
ERROR_TYPE	错误类型，取值范围如下： • MISSING：索引丢失 • ORPHAN：孤儿索引 • CONFLICT：索引数据不一致 • ERROR_SHARD：数据分片位置错误 • SUMMARY：结果汇总
STATUS	状态，取值范围如下： • FOUND：发现错误 • REPAIRED：已修复
PRIMARY_KEY	主键。
DETAILS	错误的详细信息。

8.11.6. KILL

您可以使用KILL指令终止一个正在PolarDB-X上执行的SQL。

语法

KILL语法支持以下几种用法。

- 您可以通过如下语句终止此连接正在执行的逻辑SQL与物理SQL，并断开该连接。

```
KILL PROCESS_ID
```

> **说明**
> - 您可以通过 `SHOW [FULL] PROCESSLIST` 语句查看 `PROCESS_ID` 。
> - PolarDB-X不支持 `KILL QUERY` 语句。

- 您可以通过如下语句终止一个特定的物理SQL:

```
KILL 'PHYSICAL_PROCESS_ID'
```

示例

```
mysql> KILL '0-0-521570';
Query OK, 0 rows affected (0.01 sec)
```

> **说明**
> - 您可以通过 `SHOW PHYSICAL_PROCESS_ID` 语句查看 `PHYSICAL_PROCESS_ID` 。
> - 由于 `PHYSICAL_PROCESS_ID` 列为一个字符串，并非一个数字，因此在该语句中 `PHYSICAL_PROCESS_ID` 需要使用英文单引号（''）括起来。

- 您可以使用如下语句终止PolarDB-X上所有正在执行的物理SQL:

```
KILL 'ALL'
```

8.11.7. USE

本文将介绍如何通过USE指令在PolarDB-X中切换当前连接的默认数据库。

背景信息

PolarDB-X支持访问同一PolarDB-X实例下的多个不同的数据库，就如同单机MySQL的跨数据库查询。 通常，PolarDB-X登录时需要指定一个DB_NAME作为默认数据库。您可以使用USE语句动态切换当前Schema，方便同时管理多个数据库。

语法

```
USE db_name
```

示例

您可以使用如下语法将当前的默认库切换到 `NEW_DB` 。

```
USE NEW_DB
```

8.11.8. EXPLAIN

该语句用于解释SQL语句的执行计划，包括SELECT、DELETE、INSERT、REPLACE或UPDATE语句。

语法

获取SQL计划信息：

```
EXPLAIN
{LOGICALVIEW | LOGIC | SIMPLE | DETAIL | EXECUTE | PHYSICAL | OPTIMIZER | SHARDING
 | COST | ANALYZE | BASELINE | JSON_PLAN | ADVISOR}
 {SELECT statement | DELETE statement | INSERT statement | REPLACE statement| UPDATE statement}
```

示例

- explain语句：展示基本的SQL执行计划，该执行计划是算子组成，主要体现SQL在CN上的整个执行过程。

```
mysql> explain select count(*) from lineitem group by L_ORDERKEY;
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| Project(count(*)="count(*)")
|
|   HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))")
|
|     Gather(concurrent=true)
|
|       LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_ORDERKEY`, CO
UNT(*) AS `count(*)` FROM `lineitem` AS `lineitem` GROUP BY `L_ORDERKEY`") |
| HitCache:false
|
|
| TemplateId: 5819c807
|
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
```

其中，`HitCache` 标记该查询是否命中 `PlanCache`，取值为 `false` or `true`；`TemplateId` 表示对该计划的标识，具有全局唯一性。

- explain logicalview语句：展示LogicalView所表示的下推SQL在DN上的执行计划。

```
mysql> explain LOGICALVIEW select  mysql> explain select logialview count(*) from lineitem group by L_ORDE
RKEY;
+-----------------------------------------------------------+
| LOGICAL EXECUTIONPLAN                                     |
+-----------------------------------------------------------+
| Project(count(*)="count(*)")                              |
|   HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))")   |
|     Gather(concurrent=true)                               |
|       LogicalView                                         |
|         MysqlAgg(group="L_ORDERKEY", count(*)="COUNT()") |
|           MysqlTableScan(name=[ads, lineitem])            |
| HitCache:true                                             |
| Source:PLAN_CACHE                                         |
| TemplateId: 5819c807
```

- explain execute语句：表示下推SQL在mysql的执行情况，这个语句和mysql的explain语句同义。通过该语句可以查看下推SQL在DN上有没有使用索引，有没有做全表扫描。

```
mysql> explain EXECUTE  select  count(*) from lineitem group by L_ORDERKEY;
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------------------------------------------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref | rows | filt
ered | Extra                                           |
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------------------------------------------+
| 1  | SIMPLE      | lineitem | NULL       | index | PRIMARY       | PRIMARY | 8       | NULL | 1    | 100
| Using index; Using temporary; Using filesort |
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------------------------------------------+
1 row in set (0.24 sec)
```

- explain sharding语句：展示当前查询在DN上扫描的物理分片情况。

```
mysql> explain sharding  select  count(*) from lineitem group by L_ORDERKEY;
+---------------+-----------------------------------+-------------+-----------+-----------+
| LOGICAL_TABLE | SHARDING                          | SHARD_COUNT | BROADCAST | CONDITION |
+---------------+-----------------------------------+-------------+-----------+-----------+
| lineitem      | [000000-000003].lineitem_[00-15] | 16          | false     |           |
+---------------+-----------------------------------+-------------+-----------+-----------+
1 row in set (0.04 sec)
```

- explain cost语句：相对于explain语句，除了展示执行计划以外，还会显示各个算子基于统计信息估算的代价，以及这条查询被优化器识别的WORKLOAD。

```
mysql> explain COST  select  count(*) from lineitem group by L_ORDERKEY;
+-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------+
| Project(count(*)="count(*)"): rowcount = 2508.0, cumulative cost = value = 2.4867663E7, cpu = 112574.0,
memory = 88984.0, io = 201.0, net = 4.75, id = 182
|
|   HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))"): rowcount = 2508.0, cumulative cost = value = 2.
4867662E7, cpu = 112573.0, memory = 88984.0, io = 201.0, net = 4.75, id = 180
|
|     Gather(concurrent=true): rowcount = 2508.0, cumulative cost = value = 2.4860069E7, cpu = 105039.0, m
emory = 29796.0, io = 201.0, net = 4.75, id = 178
|
|       LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_ORDERKEY`, CO
UNT(*) AS `count(*)` FROM `lineitem` AS `lineitem` GROUP BY `L_ORDERKEY`"): rowcount = 2508.0, cumulative
cost = value = 2.4860068E7, cpu = 105038.0, memory = 29796.0, io = 201.0, net = 4.75, id = 109 |
| HitCache:true
|
| Source:PLAN_CACHE
|
| WorkloadType: TP
|
| TemplateId: 5819c807
```

- explain analyze语句：相对于explain cost语句，除了显示各个算子基于统计信息估算的代价以外，该语句可以收集真实运行过程中算子输出的rowCount等信息。

```
mysql> explain ANALYZE  select  count(*) from lineitem group by L_ORDERKEY;
+-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+
| Project(count(*)="count(*)"): rowcount = 2508.0, cumulative cost = value = 2.4867663E7, cpu = 112574.0,
memory = 88984.0, io = 201.0, net = 4.75, actual time = 0.001 + 0.000, actual rowcount = 2506, actual memo
ry = 0, instances = 1, id = 182
|
|   HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))"): rowcount = 2508.0, cumulative cost = value = 2.
4867662E7, cpu = 112573.0, memory = 88984.0, io = 201.0, net = 4.75, actual time = 0.000 + 0.000, actual r
owcount = 2506, actual memory = 0, instances = 1, id = 180
|
|     Gather(concurrent=true): rowcount = 2508.0, cumulative cost = value = 2.4860069E7, cpu = 105039.0, m
emory = 29796.0, io = 201.0, net = 4.75, actual time = 0.000 + 0.000, actual rowcount = 0, actual memory =
0, instances = 0, id = 178
|
|       LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_ORDERKEY`, CO
UNT(*) AS `count(*)` FROM `lineitem` AS `lineitem` GROUP BY `L_ORDERKEY`"): rowcount = 2508.0, cumulative
cost = value = 2.4860068E7, cpu = 105038.0, memory = 29796.0, io = 201.0, net = 4.75, actual time = 0.030
+ 0.025, actual rowcount = 10000, actual memory = 0, instances = 0, id = 109 |
| HitCache:true
|
| Source:PLAN_CACHE
|
| TemplateId: 5819c807
|
+-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+
7 rows in set (1.08 sec)
```

- explain physical语句：展示查询在运行过程中执行模式、各个执行片段（Fragment）的依赖关系和并行度。该查询被识别为单机单线程计划模式（TP_LOCAL），执行计划被分为三个片段Fragment-0、Fragment-1和Fragment-2，先做预聚合再做最终的聚合计算，每个片段的执行度可以不同。

```
mysql> explain physical   select  count(*) from lineitem group by L_ORDERKEY;
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| PLAN
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| ExecutorMode: TP_LOCAL
|
| Fragment 0 dependency: [] parallelism: 4
|
| Gather(concurrent=true)
|
|   LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_ORDERKEY`, COUNT(
*) AS `count(*)` FROM `lineitem` AS `lineitem` GROUP BY `L_ORDERKEY`") |
| Fragment 1 dependency: [] parallelism: 8
|
| LocalBuffer
|
|   RemoteSource(sourceFragmentIds=[0], type=RecordType(INTEGER L_ORDERKEY, BIGINT count(*)))
|
| Fragment 2 dependency: [0, 1] parallelism: 8
|
| Project(count(*)="count(*)")
|
|   HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))")
|
|     RemoteSource(sourceFragmentIds=[1], type=RecordType(INTEGER L_ORDERKEY, BIGINT count(*)))
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
11 rows in set (0.10 sec)
```

- explain advisor语句：主要是会基于统计信息，分析当前查询的执行计划，给用户推荐可以加速查询的全局二级索引。

```
mysql> explain advisor   select  count(*) from lineitem group by L_ORDERKEY \G;
*************************** 1. row ***************************
IMPROVE_VALUE: 4.4%
  IMPROVE_CPU: 340.8%
  IMPROVE_MEM: 0.0%
   IMPROVE_IO: 1910.0%
  IMPROVE_NET: 0.0%
 BEFORE_VALUE: 2.48676627E7
   BEFORE_CPU: 112573.7
   BEFORE_MEM: 88983.8
    BEFORE_IO: 201
   BEFORE_NET: 4.7
  AFTER_VALUE: 2.38256249E7
    AFTER_CPU: 25536
    AFTER_MEM: 88983.8
     AFTER_IO: 10
    AFTER_NET: 4.7
 ADVISE_INDEX: ALTER TABLE `ads`.`lineitem` ADD GLOBAL INDEX `__advise_index_gsi_lineitem_L_ORDERKEY`(`L_O
RDERKEY`) DBPARTITION BY HASH(`L_ORDERKEY`) TBPARTITION BY HASH(`L_ORDERKEY`) TBPARTITIONS 4;
     NEW_PLAN:
Project(count(*)="count(*)")
  HashAgg(group="L_ORDERKEY", count(*)="SUM(count(*))")
    Gather(concurrent=true)
      IndexScan(tables="[000000-000003].lineitem__what_if_gsi_L_ORDERKEY_[00-15]", shardCount=16, sql="SEL
ECT `L_ORDERKEY`, COUNT(*) AS `count(*)` FROM `lineitem__what_if_gsi_L_ORDERKEY` AS `lineitem__what_if_gsi
_L_ORDERKEY` GROUP BY `L_ORDERKEY`")
         INFO: GLOBAL_INDEX
1 row in set (0.13 sec)
```

8.11.9. SQL限流

为应对突发的数据库请求流量、资源消耗过高的语句访问以及SQL访问模型的变化等问题，PolarDB-X提供了节点级别的SQL限流功能来限制造成上述问题的SQL执行，从而保证实例的持续稳定运行。本文介绍如何使用SQL限流功能。

创建限流规则

- **语法**

```
CREATE CCL_RULE [ IF NOT EXISTS ] `ccl_rule_name`
ON `database`.`table`
TO '<usename>'@'<host>'
FOR { UPDATE | SELECT | INSERT | DELETE }
[ filter_options ]
with_options
filter_options:
    [ FILTER  BY KEYWORD('KEYWORD1', 'KEYWORD2',…) ]
    [ FILTER  BY TEMPLATE('template_id') ]
 with_options:
    WITH MAX_CONCURRENCY = value1 [ , WAIT_QUEUE_SIZE = value2 ] [ , WAIT_TIMEOUT = value3 ] [ ,FAST_MATCH
= { 0 , 1 }]
```

参数说明

参数		是否必选	说明
限流规则匹配参数	`ccl_rule_name`	必选	限流规则的名称。 **说明** 为避免名称与SQL关键字冲突，建议在规则名称前后各加一个反引号（`）。
	`database`.`table`	必选	数据库和数据表的名称，支持使用星号（*）表示任意匹配。 **说明** 为避免名称与SQL关键字冲突，建议在库表名称前后各加一个反引号（`）。
	'<usename>'@'<host>'	必选	账号名称。其中Host部分支持用百分号（%）来表示任意匹配。
	UPDATE | SELECT | INSERT | DELETE	必选	SQL语句类型。当前支持UPDATE、SELECT、INSERT和DELETE类型。 **说明** 每条限流规则仅支持传入一种类型的SQL语句。

参数		是否必选	说明
	[filter_options]	可选	过滤条件，支持包括如下两种条件： ○ 关键词（KEYWORD）：查看限流规则时，关键词列表会在查询结果中被转化为 ["kwd1","kw2","kw3"...] 的字符串形式，最多支持512个字符。 **说明** ■ 若关键字是SQL语句中的参数值，匹配时大小写敏感。 ■ 若关键字是SQL语句中的其他词，匹配时大小写不敏感。 ○ 模版（TEMPLATE）：模版编号是SQL日志中的 sql_code 值，该值是参数化后的SQL语句（SQL模版）以16进制表示的哈希值。您可以通过SHOW FULL PROCESSLIST和EXPLAIN命令查看模版编号。
限流规则行为控制参数	with_options	必选	WITH选项中支持如下4个参数来控制限流规则的行为： ○ MAX_CONCURRENCY：匹配到该限流规则的SQL语句的最大并发度，超过后进入等待队列。 取值范围：[0~2^{31} - 1]，默认值为0。 ○ WAIT_QUEUE_SIZE：超过并发度后的最大等待队列长度。当等待队列长度超过该值后，SQL语句将报错。在队列中的语句仍然占用了线程资源，排队过多时也可能导致内存耗尽。 取值范围：[0~2^{31} - 1]，默认值为0。 ○ WAIT_TIMEOUT：SQL语句在等待队列中的最长等待时间，超过该等待时间后，SQL语句将报错。 取值范围：[0~2^{31} - 1]，单位为秒，默认值为600。 ○ FAST_MATCH：是否开启Cache来加速匹配。开启后，PolarDB-X会将模版编号作为Cache key的一部分，匹配结果作为value进行缓存，来加速匹配速度。 取值范围：0表示关闭，1表示开启，默认开启。 **说明** ○ 创建限流规则时，需从上述4个行为控制参数中至少选择一个传入。 ○ 当MAX_CONCURRENCY为默认值（0）时，可能会使匹配到的所有SQL返回错误。此时，建议您显式指定该参数为非0的值。 ○ PolarDB-X是分布式云原生数据库，计算层由多个节点组成，因此每个节点的并发度之和是整个实例的并发数最大值。在负载不均衡的情况下，整个实例的受限制SQL并发数可能无法达到最大并发数。

说明 仅当一个SQL语句满足所有的匹配参数条件时，才会根据该规则的WITH选项进行限流。

- **限流结果**

  一条SQL匹配到该规则后，根据限流规则中WITH选项里配置的参数，会出现如下几种结果：

  ○ **RUN（可运行）**

  若并发度还未达到最大并发度（即MAX_CONCURRENCY参数值），该SQL正常执行不会被限流。

- **WAIT（等待中）**

  若并发度已经达到最大并发度，但等待队列长度还未达到最大长度（即WAIT_QUEUE_SIZE参数值），该SQL进入等待状态，直到进入可运行（RUN）状态，或者等待超时（WAIT_TIMEOUT）状态。

  您可以通过如下命令查看由于匹配到限流规则而等待的SQL语句：

  ```
  mysql> SHOW FULL PROCESSLIST;
  ```

  返回结果示例如下：

  ```
  +----+---------------+-----------------+----------+-------------------------------+------+-------+------
  -----------------+-----------------+
  | ID | USER          | HOST            | DB       | COMMAND                       | TIME | STATE | INFO
  | SQL_TEMPLATE_ID |
  +----+---------------+-----------------+----------+-------------------------------+------+-------+------
  -----------------+-----------------+
  |  2 | polardbx_root | ***.*.*.*:62787 | polardbx | Query                         |    0 |       | show
  full processlist | NULL            |
  |  1 | polardbx_root | ***.*.*.*:62775 | polardbx | Query(Waiting-selectrulereal) |   12 |       | selec
  t 1               | 9037e5e2        |
  +----+---------------+-----------------+----------+-------------------------------+------+-------+------
  -----------------+-----------------+
  2 rows in set (0.08 sec)
  ```

  从上述查询结果可以看出：SQL语句 `select 1` 由于限流规则 `selectrulereal` 而处于等待（Waiting）状态。

- **WAIT_TIMEOUT（等待超时）**

  SQL语句进入等待状态后，当等待时间超过最长等待时间（即WAIT_TIMEOUT参数值）时，该语句将会返回错误。

  例如，设置了一条最长等待时间为10秒的限流规则，执行 `SELECT sleep(11)` 语句时会因为等待超时而报错，示例如下：

  ```
  ERROR 3009 (HY000): [11a07e23fd800000][30.225.180.55:8527][polardbx]Exceeding the max concurrency 0 of c
  cl rule selectrulereal after waiting for 10060 ms
  ```

- **KILL（结束）**

  并发度和等待队列长度均已经达到最大值，客户端将收到提示超过最大并发度的报错，报错信息中会包含匹配上的限流规则的名称。

  例如，在并发度和等待队列长度均已经达到最大值后执行 `SELECT 1;` 命令，会出现如下报错：

  ```
  ERROR 3009 (HY000): [11a07c4425c00000][**.***.***.**:8527][polardbx]Exceeding the max concurrency 0 of c
  cl rule selectrulereal
  ```

  上述结果表示：该SQL语句 `SELECT 1;` 由于超出了限流规则 `selectrulereal` 设置的最大并发度而执行失败。

- **示例**

  假设需要创建一条名为 `selectrule` 的规则，用于限制由 `'ccltest'@'%'` 用户发起的，包含 `cclmatched` 关键字的，且对任意表执行SELECT操作的SQL语句，同时将最大并发度设置为10。

  规则创建语句如下：

  ```
  CREATE CCL_RULE IF NOT EXISTS `selectrule` ON *.* TO 'ccltest'@'%'
  FOR SELECT
  FILTER BY KEYWORD('cclmatched')
  WITH MAX_CONCURRENCY=10;
  ```

查看限流规则

- **语法**

- **查看指定限流规则**

  语法如下：

  ```
  SHOW CCL_RULE `ccl_rule_name1` [, `ccl_rule_name2` ]
  ```

- **查看所有限流规则**

  语法如下：

  ```
  SHOW CCL_RULES
  ```

- **示例**

  使用如下命令查看当前数据库下所有的限流规则：

  ```
  mysql> SHOW CCL_RULES \G
  ```

  返回结果如下：

  ```
  *************************** 1. row ***************************
                      NO.: 1
                RULE_NAME: selectrulereal
                  RUNNING: 2
                  WAITING: 29
                   KILLED: 0
          MATCH_HIT_CACHE: 21374
              TOTAL_MATCH: 21406
        ACTIVE_NODE_COUNT: 2
  MAX_CONCURRENCY_PER_NODE: 1
  WAIT_QUEUE_SIZE_PER_NODE: 100
              WAIT_TIMEOUT: 600
                FAST_MATCH: 1
                  SQL_TYPE: SELECT
                      USER: ccltest@%
                     TABLE: *.*
                  KEYWORDS: ["SELECT"]
                TEMPLATEID: NULL
              CREATED_TIME: 2020-11-26 17:04:08
  ```

  参数说明

参数	说明
NO.	匹配优先级，数字越小，优先级越高。
RULE_NAME	限流规则名称。
RUNNING	匹配到该限流规则且正常执行的SQL语句数量。
WAITING	匹配到该限流规则且正在等待队列里的查询数量。
KILLED	匹配到该限流规则且被KILL的SQL语句数量。
MATCH_HIT_CACHE	匹配到该限流规则且命中Cache的SQL语句数量。
TOTAL_MATCH	匹配到该限流规则的总次数。
ACTIVE_NODE_COUNT	计算层中启用了SQL限流的节点数。
MAX_CONCURRENCY_PER_NODE	每个计算节点的并发度。
WAIT_QUEUE_SIZE_PER_NODE	每个计算节点上等待队列的最大长度。
WAIT_TIMEOUT	SQL语句在等待队列的最大等待时间。

参数	说明
FAST_MATCH	是否启动缓存加速匹配速度。
SQL_TYPE	SQL语句类型。
USER	用户名。
TABLE	数据库表。
KEYWORDS	关键词列表。
TEMPLATEID	SQL模版的编号。
CREATED_TIME	创建时间（本地时间），格式为 `yyyy-MM-dd HH:mm:ss` 。

删除限流规则

> **说明** 被删除的限流规则会立即失效，此时该规则下等待队列中的SQL语句全部会被正常执行。

- 删除指定限流规则：

```
DROP CCL_RULE [ IF EXISTS ] `ccl_rule_name1` [, `ccl_rule_name2`, ...]
```

- 删除所有限流规则：

```
CLEAR CCL_RULES
```

慢SQL限流

- 一键开启

  按语句类型分，默认为SELECT类型，相同语句类型的命令，有更新作用。语法结构如下：

```
SLOW_SQL_CCL GO [ SQL_TYPE [MAX_CONCURRENCY] [SLOW_SQL_TIME] [MAX_CCL_RULE]]
```

  - SQL_TYPE取值：ALL，SELECT，UPDATE，INSERT，默认为SELECT。
  - MAX_CONCURRENCY默认值为CPU核数的一半。
  - SLOW_SQL_TIME默认值为系统参数SLOW_SQL_TIME的值。
  - MAX_CCL_RULE的默认值为1000。

  动作：
  - 遍历整个实例的session，识别出该语句类型慢SQL的TemlateId。
  - 创建针对慢SQL的限流触发器，名称为：_SYSTEM_SLOW_SQL_CCL_TRIGGER_{SQL_TYPE}_。
  - 传递慢SQL的TemplateId给限流触发器，由限流触发器创建限流规则。
  - Kill所有该语句类型的慢TemplateId查询。

- 一键关闭

  删除由SLOW_SQL_CCL创建的限流触发器，连带着会删除由限流触发器创建的限流规则。语法结构如下：

```
SLOW_SQL_CCL BACK
```

- 查看限流情况。语法结构如下：

```
SLOW_SQL_CCL SHOW
```

  plan_cache和ccl_rules里以模版ID作为join key的一次inner join。

```
mysql> slow_sql_ccl show \G
*************************** 1. row ***************************
                    NO.: 1
                 SCHEMA: ads
            TEMPLATE_ID: 1ef9108b
                    SQL: SELECT *
FROM zsh_one_case
LIMIT ?#ads.zsh_one_case#version:1
              RULE_NAME: AUTO__SYSTEM_SLOW_SQL_CCL_TRIGGER_SELECT__ads_1ef9108b_070a1261
                RUNNING: 0
                WAITING: 0
                 KILLED: 1
            TOTAL_MATCH: 1
      ACTIVE_NODE_COUNT: 1
MAX_CONCURRENCY_PER_NODE: 0
           CREATED_TIME: 2021-06-03 00:00:00
1 row in set (0.02 sec)
```

- 如何干预慢SQL的阈值?
  - 设置SLOW_SQL_CCL GO中的语法。
  - 在一键开启SQL限流之前，设置用户变量slow_sql_time。如下：

```
set @slow_sql_time=2000;
slow_sql_ccl go;
```

  - 在控制台上设置系统参数SLOW_SQL_TIME。

> **说明** 后面的设置方式会被前面的设置方式所覆盖。

8.12. Prepared语句

本文介绍了Prepare协议的概念、用途及在Java中的开启方法。

Prepare协议介绍

Prepare协议分为两种模式：

- 二进制模式：该模式基于高效的客户端/服务器二进制协议，是程序开发中最常用的模式；
- 文本模式：该模式基于SQL语法来实现，包括PREPARE/EXECUTE/DEALLOCATE PREPARE三种语句。

PolarDB-X对这两种模式均提供支持。使用预处理语句和占位符来获取参数值具有以下优势：

- 每次执行时解析语句的开销都较小。通常情况下，数据库应用程序处理大量几乎相同的语句，只改变Prepare语句中的变量值，这样可以大幅度提升SQL执行效率。
- 防止SQL注入攻击。

二进制模式

二进制Prepare协议支持使用JDBC及其他各种语言，MySQL支持范围可参见Prepared Statements。PolarDB-X的支持情况如下：

- Prepare协议支持范围：
  - COM_STMT_PREPARE
  - COM_STMT_EXECUTE
  - COM_STMT_CLOSE
  - COM_STMT_RESET
- Prepare协议SQL支持范围：支持所有DML语句，例如：SELECT、UPDATE、DELETE、INSERT等。
- Prepare协议SQL不支持范围：不支持DML以外其他SQL语句，例如：SHOW、SET等。

在Java中开启Prepare协议

- 在JAVA客户端中，如果需要使用Prepare协议，需要显式在URL连接串中增加useServerPrepStmts=true参数，如果不指定此参数，则PreparedStatement默认会走普通查询。
- 如：jdbc:mysql://xxxxxx:3306/xxxxxx?**useServerPrepStmts=true**

Java使用示例：

```
Class.forName("com.mysql.jdbc.Driver");
Connection connection =  DriverManager.getConnection("jdbc:mysql://xxxxxx:3306/xxxxxx?useServerPrepStmts=true", "xxxxx", "xxxxx");
String sql = "insert into batch values(?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
preparedStatement.setInt(1, 0);
preparedStatement.setString(2, "polardb-x");
preparedStatement.executeUpdate();
```

文本模式

首先通过PREPARE语句给预处理语句preparable_stmt指定名称stmt_name，其中stmt_name不区分大小写，并且preparable_stmt只能为单语句。

```
PREPARE stmt_name FROM preparable_stmt;
```

接着通过EXECUTE语句执行指定的预处理语句，如果预处理语句包含参数占位符的话，必须用USING子句指定用户定义变量作为参数。

```
EXECUTE stmt_name [USING @var_name [, @var_name] ...];
```

最后通过DEALLOCATE PREPARE语句来释放清理预处理语句。

```
DEALLOCATE PREPARE stmt_name;
```

示例如下：

```
mysql> PREPARE stmt2 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
mysql> SET @a = 6;
mysql> SET @b = 8;
mysql> EXECUTE stmt2 USING @a, @b;
+-------------+
| hypotenuse |
+-------------+
|          10 |
+-------------+
mysql> DEALLOCATE PREPARE stmt2;
```

8.13. 通过LOCALITY指定库或表的存储位置

PolarDB-X支持通过LOCALITY来指定数据库或表的存储位置，以实现数据隔离或数据的均匀分布。本文介绍如何使用LOCALITY。

前提条件

实例版本需为5.4.10或以上。关于如何查看实例版本，请参见查看实例版本。

语法

您可以在创建数据库或表的时候，通过LOCALITY来指定库表的存储位置，具体语法如下：

> **说明** 通过LOCALITY语法指定了数据库或单表的位置之后，不支持再修改该库或该表的存储位置。

- 创建数据库时指定该库的存储位置，语法如下：

```
CREATE DATABASE db_name [LOCALITY=locality_option];
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
```

- 创建单表时指定该表的存储位置，语法如下：

```
CREATE TABLE tbl_name (create_definition, ...) [LOCALITY=locality_option];
locality_option:
    'dn=storage_inst_id_list'
storage_inst_id_list:
    storage_inst_id[,storage_inst_id_list]
```

示例

- 在创建数据库时指定存储位置以实现数据隔离。
  i. 假设已有一个PolarDB-X实例，您可以通过如下命令查看实例中的存储节点：

```
mysql> SHOW STORAGE;
```

返回结果如下：

```
+--------------------------+--------------------+------------+-----------+----------+-------------+
| STORAGE_INST_ID          | LEADER_NODE        | IS_HEALTHY | INST_KIND | DB_COUNT | GROUP_COUNT |
+--------------------------+--------------------+------------+-----------+----------+-------------+
| polardbx-storage-1-master | 1.1.1.1:3308       | true       | MASTER    | 2        | 6           |
| polardbx-storage-meta     | 1.1.1.1:3306       | true       | META_DB   | 2        | 2           |
| polardbx-storage-0-master | 1.1.1.1:3306       | true       | MASTER    | 2        | 8           |
+--------------------------+--------------------+------------+-----------+----------+-------------+
```

> **说明**
> - `polardbx-storage-1-master`，`polardbx-storage-0-master` 为存储节点，用于存储用户数据。
> - `polardbx-storage-meta` 为元数据的存储节点，不用于存储用户数据。

  ii. 在该实例中创建一个数据库，并通过如下命令指定其存储位置为 `polardbx-storage-0-master` 节点：

```
mysql> CREATE DATABASE db1 LOCALITY='dn=polardbx-storage-0-master';
```

> **说明**
> - 若在创建数据库时未指定数据库的存储位置，系统将默认在所有存储节点中均匀分布数据库。
> - 数据库中分表的存储位置与该库的存储位置保持一致，以实现分表上的数据隔离。
> - 数据库中单表的存储位置不受该库存储位置的影响。创建单表时若未指定存储位置，则会被随机放置在一个存储节点上，且后续该PolarDB-X实例上创建的所有未指定存储位置的单表，均会被放置在该存储节点上。

  iii. 创建成功后，您可以通过如下语句查看数据库的存储位置信息：

```
mysql> SHOW CREATE DATABASE db1;
```

返回结果如下：

```
+----------+-----------------------------------------------------------------------+
| DATABASE | CREATE DATABASE                                                       |
+----------+-----------------------------------------------------------------------+
| db1      | CREATE DATABASE `db1` /* LOCALITY = "dn=polardbx-storage-0-master" */  |
+----------+-----------------------------------------------------------------------+
1 row in set
```

  iv. 您还可以通过如下命令查看创建在该数据库下的逻辑分库和物理分库信息：

```
mysql> SHOW DS;
```

返回结果如下：

```
+----+----------------------------+-----+------------------+------------+---------+
| ID | STORAGE_INST_ID            | DB  | GROUP            | PHY_DB     | MOVABLE |
+----+----------------------------+-----+------------------+------------+---------+
| 0  | polardbx-storage-0-master | db1 | DB1_000000_GROUP | db1_000000 | 1       |
| 1  | polardbx-storage-0-master | db1 | DB1_000001_GROUP | db1_000001 | 1       |
| 2  | polardbx-storage-0-master | db1 | DB1_P00000_GROUP | db1_p00000 | 1       |
| 3  | polardbx-storage-0-master | db1 | DB1_SINGLE_GROUP | db1_single | 0       |
+----+----------------------------+-----+------------------+------------+---------+
4 rows in set
```

> **说明** 以返回结果中ID为0的一行为例，表示在存储节点 `polardbx-storage-0-master` 中的数据库db1下创建了逻辑分库DB1_000000_GROUP，物理分库db1_000000。

- 在创建单表时指定存储位置以实现数据的均匀分布。

> **说明** PolarDB-X的表分为两种，一种是分布式表，一种是单表。创建单表时若未指定存储位置，则会被随机放置在一个存储节点上，且后续再创建的单表，均会被放置在该存储节点上，造成数据分布不均匀的情况。通过LOCALITY来指定单个表存放于其他的存储节点，可以有效解决这一问题，实现数据的均匀分布。关于分布式表的信息，请参考拆分函数。

假设已有一个PolarDB-X实例，该实例上已有 `polardbx-storage-0-master` 和 `polardbx-storage-1-master` 两个存储节点，其中 `polardbx-storage-1-master` 上已创建了一个数据库 `db1` 。现需要在 `db1` 库中创建一张表，并指定其的存储位置为 `polardbx-storage-0-master` ，语法如下：

```
mysql>CREATE TABLE tb1 (id int) LOCALITY='dn=polardbx-storage-0-master';
```

创建成功后，您可以通过如下语句查看该表的拓扑结构：

```
mysql>SHOW TOPOLOGY FROM tb1;
```

返回结果如下：

```
+----+------------------+------------+
| ID | GROUP_NAME       | TABLE_NAME |
+----+------------------+------------+
| 0  | DB1_000000_GROUP | tb1        |
+----+------------------+------------+
1 row in set
```

> **说明** 以上返回结果表示tb1表位于DB1_000000_GROUP分库。

8.14. 权限管理

8.14.1. 账号权限管理

本文介绍了账号权限管理的相关操作。

PolarDB-X账号和权限系统的用法与MySQL 5.7一致，支持GRANT、REVOKE、SHOW GRANTS、CREATE USER、DROP USER、SET PASSWORD等语句，目前支持库级和表级权限的授予，全局级别和列级别权限暂时不支持。

创建账号

语法：

```
CREATE USER [IF NOT EXISTS] user IDENTIFIED BY 'password';
```

其中，user通过用户名和主机名的组合 `'username'@'host'` 确定了一个账号，账号规则如下：

- username为创建的用户名，用户名遵循以下规则；
  - 大小写敏感；
  - 长度必须大于等于4个字符，小于等于20个字符；
  - 必须以字母开头；
  - 字符可以包括大写字母、小写字母、数字。
- host指定了创建用户可以在哪台主机上登录，用户名一样但是主机名不一样也代表不同的账号，需满足以下规则：
  - HOST必须是纯IP地址，可以包含_和%通配符（_代表一个字符，%代表0个或多个字符）。含有通配符的HOST需要加上单引号，例如lily@'0.9.%.%'，david@'%';
  - 假设系统中有两个用户都符合当前准备登录的用户，则以最长前缀匹配（不包含通配符的最长IP段）的那个用户为准。例如系统有两个用户david@'30.9.12_.234'和david@'30.9.1%.234'，在主机30.9.127.234上面登录david，则使用的是david@'30.9.12_.234'这个用户；
  - 开启VPC时，主机的IP地址会发生变化。为避免账号和权限系统中的配置无效，请将HOST配置为‘%’来匹配任意IP。
- password为用户密码，需满足以下规则：
  - 长度必须大于等于6个字符，小于等于20个字符；
  - 字符可以包括大写字母、小写字母、数字、特殊字符（@#$%^&+=）。

示例：

```
mysql> CREATE USER 'user1'@'127.0.0.1' IDENTIFIED BY '123456';
mysql> CREATE USER IF NOT EXISTS 'user2'@'%' identified by '123456';
```

在PolarDB-X控制台创建完数据库之后，系统会自动在该数据库下创建两个系统账号：管理员账号和只读账号。这两个账号是系统内置的，不能删除，不能修改其权限，规则如下：

- 管理员账号的名字跟数据库名一致，比如数据库名是easydb，管理员账号的名字就叫easydb；
- 只读账号的名字是数据库名加上_RO后缀，比如数据库名是easydb，只读账号的名字就叫easydb_RO。

修改账号密码

语法：

```
SET PASSWORD FOR user = PASSWORD('auth_string')
```

示例：

```
mysql> SET PASSWORD FOR 'user1'@'127.0.0.1' = PASSWORD('654321');
```

删除账号

语法：

```
DROP USER user;
```

示例：

```
mysql> DROP USER 'user2'@'%';
```

授予账号权限

语法：

```
GRANT privileges ON database.table TO user;
```

其中，privileges为具体权限类型，数据库权限级别从高到低依次是：全局级别权限（暂不支持）、数据库级别权限、表级别权限、列级别权限。PolarDB-X目前支持和表相关联的8个基本权限项：CREATE、DROP、ALTER、INDEX、INSERT、DELETE、UPDATE、SELECT。

- TRUNCATE操作需要有表上的DROP权限；
- REPLACE操作需要有表上的INSERT和DELETE权限；
- CREATE INDEX 和 DROP INDEX操作需要有表上的INDEX权限；
- CREATE SEQUENCE需要有数据库级的创建表（CREATE）权限；
- DROP SEQUENCE需要有数据库级的删除表（DROP）权限；
- ALTER SEQUENCE需要有数据库级的更改表（ALTER）权限；
- INSERT ON DUPLICATE UPDATE语句需要有表上的INSERT和UPDATE权限。

示例：

```
mysql> GRANT SELECT,UPDATE ON `db1`.* TO 'user1'@'127.0.0.1';
```

查看账号权限

语法：

```
SHOW GRANTS [FOR user];
```

可以使用current_user()来获取当前用户。

示例：

```
mysql> SHOW GRANTS FOR 'user1'@'127.0.0.1';
+-----------------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'                      |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'           |
| GRANT SELECT, UPDATE ON db1.* TO 'user1'@'127.0.0.1' |
+-----------------------------------------------------+
mysql> SHOW GRANTS FOR current_user();
+-----------------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'                      |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'           |
| GRANT SELECT, UPDATE ON db1.* TO 'user1'@'127.0.0.1' |
+-----------------------------------------------------+
```

回收账号权限

语法：

```
REVOKE privileges ON database.table TO user;
```

示例：

```
mysql> REVOKE UPDATE ON db1.* FROM 'user1'@'127.0.0.1';
mysql> SHOW GRANTS FOR 'user1'@'127.0.0.1';
+----------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'               |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'    |
| GRANT SELECT ON db1.* TO 'user1'@'127.0.0.1' |
+----------------------------------------------+
```

8.14.2. 角色权限管理

本文介绍角色权限管理相关语法级示例。

PolarDB-X兼容原生MySQL 8.0基于角色的权限控制，请参见基于角色的权限控制。

创建角色

语法：

```
CREATE ROLE role [, role]...
```

role同user一样也由Name和Host这两部分组成，其中：

- Name不能为空；
- Host需满足如下规则：
  - 必须是纯IP地址，可以包含下划线（_）和百分号（%），但这两个符号仅代表2个普通字符，并不具备通配符意义；
  - Host留空等于%，但也是精准匹配，不具备通配符意义。

示例：

```
mysql> CREATE ROLE 'role_ro'@'%', 'role_write';
```

删除角色

语法：

```
DROP ROLE role [, role] ...
```

示例：

```
mysql> DROP ROLE 'role_ro'@'%';
```

授予角色

将权限授予角色

语法：

```
GRANT priv_type [, priv_type] ... ON priv_level TO role [, role]... [WITH GRANT OPTION]
```

示例：

```
mysql> GRANT ALL PRIVILEGES ON db1.* TO 'role_write';
```

将角色授予用户

语法：

```
GRANT role [, role] ...
    TO user_or_role [, user_or_role] ...
    [WITH ADMIN OPTION]
```

说明：

- 执行该命令必须满足如下条件的其中之一：
  - 当前用户有CREATE_USER权限；
  - 当前用户对Role有admin权限；
- 如果包含WITH ADMIN OPTION选项，则目标用户对该Role拥有admin权限；
- 将角色授予用户并不代表此用户已拥有该角色下的权限，您还需要通过 `SET DEFAULT ROLE` 语句和 `SET ROLE` 语句为用户设置需要激活的角色。

示例：

```
mysql> GRANT 'role_write' TO 'user1'@'127.0.0.1';
```

设置默认角色

语法：

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ] ...
```

执行该命令必须满足如下条件的其中之一：

- 语句中所提到的Role已通过GRANT命令授予给目标用户；
- 当前用户为目标用户，或当前用户有CREATE_USER权限。

示例：

```
mysql> SET DEFAULT ROLE 'role_write' TO 'user1'@'127.0.0.1';
```

设置当前连接角色

语法：

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

> **说明**
> - 若选择执行 `SET ROLE DEFAULT` ，则当前激活的角色为 `SET DEFAULT ROLE` 命令中选择的角色；
> - 通过该语法激活的角色仅对使用当前连接的用户生效。

示例：

```
mysql> SET ROLE 'role_write';;
```

查看角色权限

语法：

```
SHOW GRANTS
    [FOR user_or_role
        [USING role [, role] ...]]
```

示例：

```
mysql>  SHOW GRANTS FOR 'role_write'@'%';
+-----------------------------------------------+
| GRANTS FOR 'ROLE_WRITE'@'%'                   |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'role_write'@'%'        |
| GRANT ALL PRIVILEGES ON db1.* TO 'role_write'@'%' |
+-----------------------------------------------+
mysql> SHOW GRANTS FOR 'user1'@'127.0.0.1' USING 'role_write';
+-----------------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'                      |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'           |
| GRANT ALL PRIVILEGES ON db1.* TO 'user1'@'127.0.0.1' |
| GRANT 'role_write'@'%' TO 'user1'@'127.0.0.1'       |
+-----------------------------------------------------+
-- 以user1的会话执行
mysql> SELECT CURRENT_ROLE();
+------------------+
| CURRENT_ROLE()   |
+------------------+
| 'role_write'@'%' |
+------------------+
```

回收角色

回收角色的权限

语法：

```
REVOKE priv_type [, priv_type] ... ON priv_level FROM role [, role]...
```

示例：

```
mysql> REVOKE ALL PRIVILEGES ON db1.* FROM 'role_write';
mysql> SHOW GRANTS FOR 'role_write'@'%';
+---------------------------------------+
| GRANTS FOR 'ROLE_WRITE'@'%'           |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'role_write'@'%' |
+---------------------------------------+
```

回收用户的权限

语法：

```
REVOKE role [, role ] ... FROM user_or_role [, user_or_role ] ...
```

示例：

```
mysql> SHOW GRANTS FOR 'user1'@'127.0.0.1';
+-----------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'                |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'     |
| GRANT SELECT ON db1.* TO 'user1'@'127.0.0.1'  |
| GRANT 'role_write'@'%' TO 'user1'@'127.0.0.1' |
+-----------------------------------------------+
mysql> REVOKE 'role_write' FROM 'user1'@'127.0.0.1';
mysql> SHOW GRANTS FOR 'user1'@'127.0.0.1';
+-----------------------------------------------+
| GRANTS FOR 'USER1'@'127.0.0.1'                |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'127.0.0.1'     |
| GRANT SELECT ON db1.* TO 'user1'@'127.0.0.1'  |
+-----------------------------------------------+
```

8.14.3. 三权分立下的权限管理

本文介绍了三权分立下的三类管理员相关权限。

PolarDB-X在开启三权分立模式后，默认的高权限账号将划分为系统管理员账号、安全管理员账号和审计管理员账号，具体功能介绍与使用方法可参考三权分立。

下面将以上述三个类型管理员的身份来演示不同类型的SQL操作，其中三类管理员的用户名分别如下所示：

- 系统管理员：admin_dba
- 安全管理员：admin_security
- 审计管理员：admin_audit

系统管理员

只有系统管理员具备DDL相关权限：

```
mysql> SELECT USER();
+--------------------------+
| USER()                   |
+--------------------------+
| admin_dba@10.159.164.179 |
+--------------------------+
mysql> create database priv_test;
Query OK, 1 row affected (0.09 sec)
mysql> use priv_test;
Database changed
mysql> create table test (id int primary key, value int);
Query OK, 0 rows affected (1.23 sec)
```

但系统管理员不具备DML/DQL/DAL以及权限管理功能：

```
mysql> select * from test;
ERROR 5108 (HY000): [130b87654f001000][10.2.57.181:3028][priv_test]ERR-CODE: [TDDL-5108][ERR_CHECK_PRIVILEGE
_FAILED_ON_TABLE] User admin_dba@'10.159.164.179' does not have 'SELECT' privilege on table 'TEST'. Database
is PRIV_TEST.
mysql> insert into test values (1, 123);
ERROR 5108 (HY000): [130b877647c01000][10.2.57.181:3028][priv_test]ERR-CODE: [TDDL-5108][ERR_CHECK_PRIVILEGE
_FAILED_ON_TABLE] User admin_dba@'10.159.164.179' does not have 'INSERT' privilege on table 'TEST'. Database
is PRIV_TEST.
mysql> CREATE USER 'user1'@'%' IDENTIFIED BY '123456';
ERROR 5110 (HY000): [130b877e6f001000][10.2.57.181:3028][priv_test]ERR-CODE: [TDDL-5110][ERR_CHECK_PRIVILEGE
_FAILED] User admin_dba@'%' does not have 'CREATE ACCOUNT' privilege.
```

安全管理员

安全管理员同样不具备DML/DQL/DAL权限，但支持账户或角色的权限管理以及将DML/DQL/DAL权限授予给普通账号：

```
mysql> SELECT USER();
+--------------------------------+
| USER()                         |
+--------------------------------+
| admin_security@10.159.164.119 |
+--------------------------------+
mysql> use priv_test;
Database changed
mysql> select * from test;
ERROR 5108 (HY000): [130b8a31af401000][10.57.23.233:3028][priv_test]ERR-CODE: [TDDL-5108][ERR_CHECK_PRIVILEG
E_FAILED_ON_TABLE] User admin_security@'10.159.164.119' does not have 'SELECT' privilege on table 'TEST'. Da
tabase is PRIV_TEST.
mysql> CREATE USER 'user1'@'%' IDENTIFIED BY '123456';
Query OK, 0 rows affected (0.08 sec)
mysql> GRANT SELECT,INSERT,UPDATE ON priv_test.* TO 'user1'@'%';
Query OK, 0 rows affected (0.06 sec)
-- 使用user1账户登录
mysql> SELECT USER();
+---------------------+
| USER()              |
+---------------------+
| user1@10.159.164.29 |
+---------------------+
mysql> show grants;
+-----------------------------------------------------------+
| GRANTS FOR 'USER1'@'%'                                    |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'%'                         |
| GRANT SELECT, INSERT, UPDATE ON priv_test.* TO 'user1'@'%' |
+-----------------------------------------------------------+
mysql> insert into test values (1, 123);
Query OK, 1 row affected (0.02 sec)
mysql> select * from test;
+------+-------+
| id   | value |
+------+-------+
|    1 |   123 |
+------+-------+
```

安全管理员不具备DDL相关权限：

```
mysql> drop table test;
ERROR 5108 (HY000): [130b8a1b9dc01000][10.2.57.181:3028][priv_test]ERR-CODE: [TDDL-5108][ERR_CHECK_PRIVILEGE
_FAILED_ON_TABLE] User admin_security@'10.159.164.59' does not have 'DROP' privilege on table 'TEST'. Databa
se is PRIV_TEST.
```

审计管理员

审计管理员只具备查看审计日志的权限：

```
mysql> SELECT USER();
+----------------------------+
| USER()                     |
+----------------------------+
| admin_audit@10.159.164.209 |
+----------------------------+
mysql> select USER_NAME,HOST,PORT,AUDIT_INFO,ACTION,TRACE_ID from polardbx_audit_log where SCHEMA = 'priv_te
st';
+----------------+----------------+-------+-----------------------------------------------------+------------
-+------------------+
| USER_NAME      | HOST           | PORT  | AUDIT_INFO                                          | ACTION
| TRACE_ID         |
+----------------+----------------+-------+-----------------------------------------------------+------------
-+------------------+
| admin_dba      | 10.159.164.239 | 26245 | create table test (id int primary key, value int)   | CREATE
| 130b83120e003000 |
| admin_security | 10.159.164.239 | 37537 | create table test2 (id int primary key, value int) | CREATE
| 130b839700402000 |
| admin_audit    | 10.159.164.89  | 51128 | create table test2 (id int primary key, value int) | CREATE
| 130b83ea42404000 |
| admin_dba      | 10.159.164.119 | 15923 | CREATE USER 'user1'@'%' IDENTIFIED BY '123456'      | CREATE
| 130b8658c9c03000 |
| admin_dba      | 10.159.164.119 | 15923 | CREATE USER 'user1'@'%' IDENTIFIED BY '123456'      | CREATE
| 130b866b49c03000 |
| admin_dba      | 10.159.164.179 | 24559 | CREATE USER 'user1'@'%' IDENTIFIED BY '123456'      | CREATE
| 130b877e6f001000 |
| admin_security | 10.159.164.119 | 44965 | create table test2 (id int primary key, value int) | CREATE
| 130b87c6f6002000 |
| admin_security | 10.159.164.119 | 44965 | CREATE USER 'user1'@'%' IDENTIFIED BY '123456'      | CREATE_USER
| 130b87ee65402000 |
| admin_security | 10.159.164.119 | 44965 | CREATE USER 'user1'@'%' IDENTIFIED BY '123456'      | CREATE
| 130b87ee65402000 |
| admin_security | 10.159.164.119 | 44965 | GRANT SELECT,UPDATE ON priv_test.* TO 'user1'@'%'   | GRANT
| 130b88a7b0402000 |
| admin_security | 10.159.164.59  | 21156 | drop table test                                     | DROP
| 130b8a1b9dc01000 |
+----------------+----------------+-------+-----------------------------------------------------+------------
-+------------------+
```

8.15. Replication

特性

- 兼容MySQL Binlog文件格式和Dump协议。PolarDB-X的全局Binlog是在DN节点的物理Binlog基础之上生成的，剔除了分布式事务的细节，只保留了单机事务的特性。同时，全局Binlog兼容MySQL Binlog文件格式，在数据订阅方式上也完全兼容MySQL Dump协议，您可以像使用单机MySQL一样来订阅PolarDB-X的事务日志。
- 保证分布式事务的完整性和有序性。全局Binlog不是将物理Binlog简单地汇总到一起，而是通过合并模块和归并模块保证了分布式事务的完整性和有序性，从而实现高数据一致性。例如，在转账场景下，基于全局Binlog能力，接入PolarDB-X的下游MySQL，可以在任何时刻查询到一致的余额。
- 提供7x24小时服务能力，运维简单。全局Binlog剔除了PolarDB-X的内部细节（此时您可以将PolarDB-X看作一个单机MySQL），来避免实例内部发生的变化对数据订阅链路的影响。PolarDB-X通过一系列的协议和算法来保证全局Binlog的服务能力，确保实例内部发生的各种变更（如HA切换、增删节点、执行Scale Out或分布式DDL等操作）不会影响数据订阅链路的正常工作。

使用限制

- 暂不支持Gtid（Global Transaction Identifier）模式下的数据订阅方式。
- 仅当事务策略指定为TSO时（即更高强度的一致性保证），才支持对分布式事务的合并。

数据订阅源端支持的SQL语句

- 查看PolarDB-X全局Binlog文件列表。

```
SHOW BINARY LOGS
```

> **说明** 执行上述SQL语句需要有SUPER或REPLICATION CLIENT权限。权限操作请参见账号权限管理。

- 查看PolarDB-X作为主Master角色的Binlog信息。

```
SHOW MASTER STATUS
```

> **说明** 执行上述SQL语句需要有SUPER或REPLICATION CLIENT权限。权限操作请参见账号权限管理。

- 查看全局Binlog文件中的具体事件信息。

```
SHOW BINLOG EVENTS
   [IN 'log_name']
   [FROM pos]
   [LIMIT [offset,] row_count]
```

> **说明** 执行上述SQL语句需要有SUPER或REPLICATION CLIENT权限。权限操作请参见账号权限管理。

数据订阅目标端支持的SQL语句

如果数据订阅目标端是标准MySQL，目前支持MySQL的Replicate指令。

- 在数据订阅目标端设置需要同步的源端数据源信息。

```
CHANGE MASTER TO option [, option] ... [ channel_option ]
option: {
    MASTER_BIND = 'interface_name'
  | MASTER_HOST = 'host_name'
  | MASTER_USER = 'user_name'
  | MASTER_PASSWORD = 'password'
  | MASTER_PORT = port_num
  | PRIVILEGE_CHECKS_USER = {'account' | NULL}
  | REQUIRE_ROW_FORMAT = {0|1}
  | REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF}
  | ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | uuid}
  | MASTER_LOG_FILE = 'source_log_name'
  | MASTER_LOG_POS = source_log_pos
  | MASTER_AUTO_POSITION = {0|1}
  | RELAY_LOG_FILE = 'relay_log_name'
  | RELAY_LOG_POS = relay_log_pos
  | MASTER_HEARTBEAT_PERIOD = interval
  | MASTER_CONNECT_RETRY = interval
  | MASTER_RETRY_COUNT = count
  | SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}
  | MASTER_DELAY = interval
  | MASTER_COMPRESSION_ALGORITHMS = 'value'
  | MASTER_ZSTD_COMPRESSION_LEVEL = level
  | MASTER_SSL = {0|1}
  | MASTER_SSL_CA = 'ca_file_name'
  | MASTER_SSL_CAPATH = 'ca_directory_name'
  | MASTER_SSL_CERT = 'cert_file_name'
  | MASTER_SSL_CRL = 'crl_file_name'
  | MASTER_SSL_CRLPATH = 'crl_directory_name'
  | MASTER_SSL_KEY = 'key_file_name'
  | MASTER_SSL_CIPHER = 'cipher_list'
  | MASTER_SSL_VERIFY_SERVER_CERT = {0|1}
  | MASTER_TLS_VERSION = 'protocol_list'
  | MASTER_TLS_CIPHERSUITES = 'ciphersuite_list'
  | MASTER_PUBLIC_KEY_PATH = 'key_file_name'
  | GET_MASTER_PUBLIC_KEY = {0|1}
  | NETWORK_NAMESPACE = 'namespace'
  | IGNORE_SERVER_IDS = (server_id_list)
}
channel_option:
    FOR CHANNEL channel
server_id_list:
    [server_id [, server_id] ... ]
```

- 开启主备同步

```
START {SLAVE | REPLICA}
```

- 停止主备同步

```
STOP {SLAVE | REPLICA}
```

- 重置主备同步，需要先停止主备同步

```
RESET {SLAVE | REPLICA} [ALL] [channel_option]
channel_option:
    FOR CHANNEL channel
```

> **说明** 如果目标端是PolarDB-X，目前暂时不支持相关Replicate指令。

8.16. Sequence

8.16.1. 概述

本文将为您介绍Sequence的相关概念和支持的类型。

PolarDB-X全局唯一数字序列（64位数字，对应MySQL中Signed BIGINT类型，以下简称为Sequence）的主要目标是为了生成全局唯一和有序递增的数字序列，常用于主键列、唯一索引列等值的生成。

基本概念

了解以下概念，将帮助您更好地选用Sequence类型：

- 连续：如果本次取值为n，下一次取值一定是n + 1，则是连续的；如果下一次取值不能保证为n + 1，则是非连续的；
- 单调递增：如果本次取值为n，下一次取值一定是一个比n大的数，则是单调递增的；
- 单点：存在单点故障风险；
- 宏观上单调递增，微观上非单调递增：类似于 `1、3、2、4、5、7、6、8、.....` 这样的序列，这个序列从宏观是看是递增的，微观上非单调递增。
- 单元化能力：能够跨实例或跨库分配全局唯一数字序列。

用法

PolarDB-X中的Sequence主要有两类用法：

- 显式Sequence：通过Sequence DDL语法创建和维护，可以独立使用；通过 `select seq.nextval` 获取序列值， `seq` 是具体Sequence的名字。
- 隐式Sequence，在为主键定义AUTO_INCREMENT后，用于自动填充主键，由PolarDB-X自动维护。

支持的Sequence类型及其特性

PolarDB-X目前共支持如下4种Sequence类型：

类型（缩写）	全局唯一	连续	单调递增	同一连接内单调递增	非单点	数据类型	可读性	单元化能力
Group Sequence（GROUP）	是	否	否	是	是	所有整型	好	否
单元化 Group Sequence（GROUP）	是	否	否	是	是	所有整型	好	是
Time-based Sequence（TIME）	是	否	宏观上单调递增，微观上非单调递增	是	是	仅支持BIGINT	差	否
Simple Sequence（SIMPLE）	是	是	是	是	否	所有整型	好	否

Group Sequence（GROUP，默认使用）

全局唯一的Sequence，产生的值是自然数序列，但是不保证连续和单调递增。如果未指定Sequence类型，PolarDB-X默认使用Group Sequence。

实现原理：采用多个节点产生值来保证高可用，每次取出一段值，如果该段值没有取完（比如连接断掉等情形），就会产生跳跃段。

- 优点：全局唯一，不会产生单点问题，性能非常好。
- 缺点：产生的序列不连续，可能会有跳跃段；不会严格从起始值开始取值；不能循环。

单元化 Group Sequence（GROUP）

以Group Sequence为基础，扩展了单元化能力，能够跨实例或跨库实现全局唯一，但同样不保证连续和单调自增。当单元化Group Sequence仅配置一个单元时，等价于普通的Group Sequence。

- 优点：具备Group Sequence的所有优点，且扩展了单元化能力。
- 缺点：产生的序列不连续，可能会有跳跃段；不会严格从起始值开始取值；不能循环。

基本原理与Group Sequence相同；通过扩展参数选项，支持自定义单元数量和单元索引：

- 单元数量决定了单元化 Group Sequence的全局唯一数字序列分配空间；
- 每个单元（由单元索引指定）占用全局唯一数字序列分配空间中的一个子集；
- 不同单元（指定了不同的单元索引）占用的子集之间不重叠（即不会分配相同的Sequence值）；
- 属于同一个全局唯一数字序列分配空间的每个单元化Group Sequence，必须指定相同的单元数量和不同的单元索引。

> **说明**
>
> 单元化 Group Sequence从以下版本开始提供支持：
>
> - V5.2：V5.2.7-1606682（2018.4.27）
> - V5.3：V5.3.3-1670435（2018.8.15）

Time-based Sequence（TIME）

基于时间戳+节点编号+序列号组合而成的一种Sequence，保证全局唯一和宏观自增；这种Sequence值的更新不依赖于数据库，也不需要持久化到数据库，仅在数据库中保留名称和类型信息，性能很好；产生的是类似于 `776668092129345536`、`776668098018148352`、`776668111578333184`、`776668114812141568`、`.....` 这样的序列值。

- 优点：全局唯一、性能很好。
- 缺点：产生的序列不连续，起始值、步长、最大值、是否循环这些参数对于Time-based Sequence无意义。

> **说明**
>
> - 用于表中自增列时，必须使用BIGINT类型；
> - Time-based Sequence从以下版本开始提供支持：
>   - V5.2：V5.2.8-15432885（2018.11.27）
>   - V5.3：V5.3.6-15439241（2018.11.29）

Simple Sequence（SIMPLE）

仅Simple Sequence支持自定义步长、最大值和循环/非循环利用。

- 优点：全局唯一、连续、单调递增，并具备最大值和循环利用等特性。
- 缺点：单点，性能较差，存在瓶颈，需要谨慎使用。

每产生一个值都要进行一次持久化操作。

使用场景

这几种Sequence都保证全局唯一，均可以应用在主键列和唯一索引列。

- 大部分场景下建议选用Group Sequence；
- 如果有跨实例或跨库分配全局唯一数字序列的需求，可以选用单元化Group Sequence；
- 如果业务上能接受整体趋势上的宏观自增，不介意微观上的不保证自增，且不想依赖数据库的分配机制，则Time-based Sequence可能是合适的选择；
- 如果业务强依赖连续的Sequence值，此时只能使用Simple Sequence（注意Simple Sequence的性能问题）。

以创建一个起始值是100000，步长为1的Sequence为例说明。

- 如果采用 **Simple Sequence**，则会严格产生 `100000`、`100001`、`100002`、`100003`、`100004`、`.....`、`200000`、`200001`、`200002`、`200003`、`.....` 这样的序列（全局唯一、连续、单调递增）。Simple Sequence 会保证持久化，即使发生单点问题，服务重启后依然会在断点继续产生 Sequence 值，中间不会产生跳跃段。Simple Sequence 的机理是每产生一个值都要进行一次持久化操作，因此性能并不是很好。

- 如果采用Group Sequence或单元化 Group Sequence，产生的序列有可能是 `200001、200002、200003、200004、100001、100002、100003、.....` 这样的序列。

> **说明**
> - Group Sequence起始值并不会严格从设定的参数（本例中是100000）开始，但保证比该参数大。本例中是从200001开始取值的。
> - Group Sequence保证全局唯一，但是会有跳跃段。比如Group Sequence的某个节点失效，或者某个连接只取了一部分值，然后该连接被关闭了，都会产生跳跃段。该例中200004和100001之间产生了跳跃段。
> - 单元化Group Sequence跨实例或跨库的全局唯一性，必须通过指定相同的单元数量和不同的单元索引来保证。

8.16.2. 使用限制

本文将介绍使用Sequence过程中的注意事项及问题处理的方法。

限制与注意事项

在使用Sequence时，您需要注意如下事项：

- 转换Sequence类型时，必须指定START WITH起始值。
- 单元化Group Sequence不支持作为源或目标的类型转换，也不支持起始值以外的参数修改。
- 属于同一个全局唯一数字序列分配空间的每个单元化Group Sequence ，必须指定相同的单元数量和不同的单元索引。
- 在PolarDB-X非拆分模式库（即后端仅关联一个已有的RDS物理库）、或拆分模式库中仅有单表（即所有表都是单库单表，且无广播表）的场景下执行INSERT时， PolarDB-X会自动优化并直接下推语句，绕过优化器中分配Sequence值的部分。此时 `INSERT INTO ... VALUES (seq.nextval, ...)` 这种用法不支持，建议使用后端RDS/MySQL自增列机制代替。
- 如果将指定分库的Hint用在INSERT语句上，比如INSERT INTO ... VALUES ... 或INSERT INTO ... SELECT ...，且目标表使用了Sequence，则PolarDB-X会绕过优化器直接下推语句，使Sequence不生效，目标表最终会使用后端RDS/MySQL表中的自增机制生成id。
- 必须对同一个表采用一种统一的方式分配自增id：或者依赖于Sequence，或者依赖于后端RDS/MySQL表的自增列；应避免两种机制混用，否则很可能会造成id冲突（INSERT时产生重复id）的情况，且难于排查。
- 将Time-based Sequence用于表中自增列时，该列必须使用BIGINT类型。

如何处理主键冲突

如果直接在RDS中写入了数据，而对应的主键值不是PolarDB-X生成的Sequence值，那么后续让PolarDB-X自动生成主键写入数据库，可能会和这些数据发生主键冲突，您可以通过如下步骤解决此问题：

1. 通过 `SHOW SEQUENCES` 来查看当前已有Sequence。AUTO_SEQ_ 开头的Sequence是隐式Sequence（创建表时加上AUTO_INCREMENT参数的字段产生的Sequence）。

   请在命令行输入如下代码：

   ```
   mysql> SHOW SEQUENCES;
   ```

   返回结果如下：

   ```
   +--------------------+-------+--------------+------------+-----------+-------+-------+
   | NAME               | VALUE | INCREMENT_BY | START_WITH | MAX_VALUE | CYCLE | TYPE  |
   +--------------------+-------+--------------+------------+-----------+-------+-------+
   | AUTO_SEQ_xkv_t_item | 0     | N/A          | N/A        | N/A       | N/A   | GROUP |
   | AUTO_SEQ_xkv_shard  | 0     | N/A          | N/A        | N/A       | N/A   | GROUP |
   +--------------------+-------+--------------+------------+-----------+-------+-------+
   2 rows in set (0.04 sec)
   ```

2. 若xkv_t_item表有冲突，并且xkv_t_item表主键是ID，那么从PolarDB-X获取这个表最大主键值。

   请在命令行输入如下代码：

   ```
   mysql> SELECT MAX(id) FROM xkv_t_item;
   ```

   返回结果如下：

```
+------------+
| MAX(id)    |
+------------+
| 8231       |
+------------+
1 row in set (0.01 sec)
```

3. 更新Sequence表中对应的值，这里更新成比8231要大的值，比如9000，更新完成后，后续插入语句生成的自增主键将不再报错。

   请在命令行输入如下代码：

```
mysql> ALTER SEQUENCE AUTO_SEQ_xkv_t_item START WITH 9000;
```

8.16.3. 显式用法

8.16.3.1. 创建Sequence

本文主要介绍如何创建各种类型的Sequence。

Group Sequence

语法

```
CREATE [ GROUP ] SEQUENCE <name>
[ START WITH <numeric value> ]
```

参数说明

参数	说明
START WITH	Group Sequence 的起始值，若未指定，则默认起始值为100001。

示例

创建一个Group Sequence：

- 方法一

  请在命令行输入如下代码：

```
mysql> CREATE SEQUENCE seq1;
```

- 方法二

  请在命令行输入如下代码：

```
mysql> CREATE GROUP SEQUENCE seq1;
```

单元化 Group Sequence

语法

```
CREATE [ GROUP ] SEQUENCE <name>
[ START WITH <numeric value> ]
[ UNIT COUNT <numeric value> INDEX <numeric value> ]
```

参数说明

参数	说明
START WITH	单元化Group Sequence的起始值，默认起始值依赖于单元数量和单元索引；若单元数量和单元索引未被指定或为默认值，则默认起始值为100001。

参数	说明
UNIT COUNT	单元化Group Sequence的单元数量，默认值为1。
INDEX	单元化Group Sequence的单元索引，取值范围为 [0, 单元数量 - 1]，默认值为0。

> **说明**
> - 如果未指定类型关键字，则默认类型为 Group Sequence。
> - Group Sequence 和单元化Group Sequence是非连续的。START WITH参数对于它们仅具有指导意义，Group Sequence 和单元化Group Sequence不会严格按照该参数作为起始值，但是保证起始值比该参数大。
> - 可以将Group Sequence看作单元化Group Sequence的一个特例，即UNIT COUNT = 1 且 INDEX = 0 时的单元化Group Sequence。

示例

创建包含3个单元的全局唯一数字序列（将3个同名的、指定了相同单元数量和不同单元索引的单元化Group Sequence，分别用于3个不同的实例或库，组成一个全局唯一数字序列）。

1. 实例1/库1：

```
mysql> CREATE GROUP SEQUENCE seq2 UNIT COUNT 3 INDEX 0;
```

2. 实例2/库2：

```
mysql> CREATE GROUP SEQUENCE seq2 UNIT COUNT 3 INDEX 1;
```

3. 实例3/库3：

```
mysql> CREATE GROUP SEQUENCE seq2 UNIT COUNT 3 INDEX 2;
```

Time-based Sequence

语法

```
CREATE TIME SEQUENCE <name>
```

> **注意** 存储Time-based Sequence值的列必须为BIGINT类型。

示例

创建一个Time-based Sequence：

请在命令行输入如下代码：

```
7mysql> CREATE TIME SEQUENCE seq3;
```

Simple Sequence

语法

```
CREATE SIMPLE SEQUENCE <name>
[ START WITH <numeric value> ]
[ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ][ CYCLE | NOCYCLE ]
```

参数说明

参数	说明
START WITH	Simple Sequence的起始值，若未指定，则默认起始值为1。

参数	说明
INCREMENT BY	Simple Sequence每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1。
MAXVALUE	Simple Sequence允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值，即9223372036854775807。
CYCLE 或 NOCYCLE	两者只能选择其一，代表当Simple Sequence增长到最大值后，是否允许继续循环（即从START WITH重新开始）使用该Simple Sequence。若未指定，则默认值为NOCYCLE。

示例

创建一个Simple Sequence，起始值是1000，步长为2，最大值为99999999999，增长到最大值后不继续循环。

请在命令行输入如下代码：

```
mysql> CREATE SIMPLE SEQUENCE seq4 START WITH 1000 INCREMENT BY 2 MAXVALUE 99999999999 NOCYCLE;
```

8.16.3.2. 修改Sequence

本文主要介绍如何对Sequence的各种类型进行修改。

PolarDB-X支持对Sequence的各种类型进行如下修改：

- 修改Simple Sequence的参数：起始值、步长、最大值、循环或非循环。
- 修改Group Sequence或单元化 Group Sequence的参数：起始值。
- 不同类型Sequence间的转换（单元化Group Sequence除外）。

注意事项

在对Sequence的类型进行修改时，您需要注意如下事项：

- Group Sequence和单元化Group Sequence是非连续的。START WITH参数对于它们仅具有指导意义，Group Sequence和单元化Group Sequence不会严格按照该参数作为起始值，但是保证起始值比该参数大。
- 单元化Group Sequence不支持转换到其它类型或修改单元化相关的参数。
- 对于Simple Sequence，如果修改Sequence时指定了START WITH，则会立即生效，下次取Sequence值时会从新的START WITH值开始。比如原先Sequene增长到100，这时把START WITH值改成了200，那么下一次获取的Sequence值就从200开始。
- 修改START WITH的参数值时，需要仔细评估已经产生的Sequence值，以及生成新Sequence值的速度，防止产生冲突。如非必要，请谨慎修改START WITH参数值。

Group Sequence

语法

```
ALTER SEQUENCE <name> [ CHANGE TO SIMPLE | TIME ]
START WITH <numeric value>
[ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ]
[ CYCLE | NOCYCLE ]
```

参数说明

参数	说明
START WITH	Sequence的起始值，无默认值，若未指定则忽略该参数，在转换类型时必须指定。

参数	说明
INCREMENT BY	仅在将Group Sequence转换为Simple Sequence时有效，是Simple Sequence每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1。
MAXVALUE	仅在将Group Sequence转换为Simple Sequence时有效，是Simple Sequence允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值，即9223372036854775807。
CYCLE 或 NOCYCLE	仅在将Group Sequence转换为Simple Sequence时有效，两者只能选择其一，代表当Simple Sequence值增长到最大值后，是否允许继续循环（即从START WITH重新开始）使用该Simple Sequence，若未指定，则默认值为NOCYCLE。

说明 当修改的目标类型为TIME时，不支持上述参数。

单元化Group Sequence

语法

```
ALTER SEQUENCE <name>
START WITH <numeric value>
```

参数说明

参数	说明
START WITH	单元化Group Sequence的起始值，无默认值，若未指定则忽略该参数。

说明 单元化Group Sequence 不支持转换到其它类型或修改单元化相关的参数。

Time-based Sequence

语法

```
ALTER SEQUENCE <name>[ CHANGE TO GROUP | SIMPLE ]
START WITH <numeric value>
[ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ]
[ CYCLE | NOCYCLE ]
```

参数说明

参数	说明
START WITH	Sequence的起始值，无默认值，若未指定则忽略该参数，在转换类型时必须指定。
INCREMENT BY	Simple Sequence每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1，将Simple Sequence转换为Group Sequence时该参数无效。
MAXVALUE	Simple Sequence允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值，即9223372036854775807，将Simple Sequence转换为Group Sequence时该参数无效。

参数	说明
CYCLE或NOCYCLE	两者只能选择其一，代表当Simple Sequence值增长到最大值后，是否允许继续循环（即仍从START WITH开始）使用该Simple Sequence，若未指定，则默认值为NOCYCLE，将Simple Sequence转换为Group Sequence时该参数无效。

Simple Sequence

语法

```
ALTER SEQUENCE <name> [ CHANGE TO GROUP | TIME ]
START WITH <numeric value>
[ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ]
[ CYCLE | NOCYCLE ]
```

参数说明

参数	说明
START WITH	Sequence的起始值，无默认值，若未指定则忽略该参数，在转换类型时必须指定。
INCREMENT BY	Simple Sequence每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1，将Simple Sequence转换为Group Sequence时该参数无效。
MAXVALUE	Simple Sequence允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值，即9223372036854775807，将Simple Sequence转换为Group Sequence时该参数无效。
CYCLE 或 NOCYCLE	两者只能选择其一，代表当Simple Sequence值增长到最大值后，是否允许继续循环（即仍从START WITH开始）使用该Simple Sequence，若未指定，则默认值为NOCYCLE，将Simple Sequence转换为Group Sequence时该参数无效。

> **说明** 当修改的目标类型为TIME时，不支持上述参数。

不同类型Sequence间的转换

在对Sequence的不同类型进行转换时，您需要了解如下事项：

- 通过 `ALTER SEQUENCE` 的 `CHANGE TO <sequence_type>` 子句实现。
- `ALTER SEQUENCE` 如果指定了 `CHANGE TO` 子句，则强制必须加上START WITH参数，避免忘记指定起始值而造成取值时得到重复值；若没有CHANGE TO（可选参数），则不强制。
- 不支持单元化Group Sequence作为源或目标的类型转换。

示例

- 将Simple Sequence seq4的起始值改为3000，步长改为5，最大值改为1000000，增长到最大值后改为继续循环。语句如下：

  ```
  mysql> ALTER SEQUENCE seq4 START WITH 3000 INCREMENT BY 5 MAXVALUE 1000000 CYCLE;
  ```

- 将Group Sequence转换为Simple Sequence。

  ```
  mysql> ALTER SEQUENCE seq1 CHANGE TO SIMPLE START WITH 1000000;
  ```

8.16.3.3. 查询与获取Sequence

本文主要介绍如何查询Sequence类型以及获取Sequence值。

查询Sequence

语法

```
SHOW SEQUENCES
```

示例

请在命令行输入如下代码：

```
mysql> SHOW SEQUENCES;
```

返回结果如下：

```
+------+--------+------------+------------+------------+--------------+------------+-------------+-------+--
------+
| NAME | VALUE  | UNIT_COUNT | UNIT_INDEX | INNER_STEP | INCREMENT_BY | START_WITH | MAX_VALUE   | CYCLE | T
YPE   |
+------+--------+------------+------------+------------+--------------+------------+-------------+-------+--
------+
| seq1 | 100000 | 1          | 0          | 100000     | N/A          | N/A        | N/A         | N/A   | G
ROUP  |
| seq2 | 400000 | 3          | 1          | 100000     | N/A          | N/A        | N/A         | N/A   | G
ROUP  |
| seq3 | N/A    | N/A        | N/A        | N/A        | N/A          | N/A        | N/A         | N/A   | T
IME   |
| seq4 | 1006   | N/A        | N/A        | N/A        | 2            | 1000       | 99999999999 | N     | S
IMPLE |
+------+--------+------------+------------+------------+--------------+------------+-------------+-------+--
------+
4 rows in set (0.00 sec)
```

> **说明** 返回结果中的TYPE列，显示的是Sequence类型的缩写。

获取显式Sequence值

语法

```
[<schema_name>.]<sequence name>.NEXTVAL
```

示例

- 方法一

  请在命令行输入如下代码：

  ```
  mysql> SELECT sample_seq.nextval FROM dual;
  ```

  返回结果如下：

  ```
  +--------------------+
  | SAMPLE_SEQ.NEXTVAL |
  +--------------------+
  |             101001 |
  +--------------------+
  1 row in set (0.04 sec)
  ```

- 方法二

  请在命令行输入如下代码：

```
mysql> INSERT INTO some_users (name,address,gmt_create,gmt_modified,intro) VALUES ('sun',sample_seq.nextval,now(),now(),'aa');
```

> 说明
> - 该方法是把sample_seq.nextval当做一个值写入了 SQL中。
> - 如果建表时已经指定了AUTO_INCREMENT参数，INSERT时不需要指定自增列，可以让PolarDB-X自动维护。

批量获取Sequence值

语法

批量获取Sequence值的语法如下：

```
SELECT [<schema_name>.]<sequence name>.NEXTVAL FROM DUAL WHERE COUNT = <numeric value>
```

示例

请在命令行输入如下代码：

```
mysql> SELECT sample_seq.nextval FROM dual WHERE count = 10;
```

返回结果如下：

```
+--------------------+
| SAMPLE_SEQ.NEXTVAL |
+--------------------+
|             101002 |
|             101003 |
|             101004 |
|             101005 |
|             101006 |
|             101007 |
|             101008 |
|             101009 |
|             101010 |
|             101011 |
+--------------------+
10 row in set (0.04 sec)
```

8.16.3.4. 删除Sequence

本文主要介绍如何删除已经创建的Sequence。

语法

删除Sequence的语法如下：

```
DROP SEQUENCE <name>
```

示例

请在命令行输入如下代码：

```
mysql> DROP SEQUENCE seq3;
```

8.16.4. 隐式用法

8.16.4.1. 创建 Sequence

在为拆分表或广播表的主键定义AUTO_INCREMENT后，Sequence可以用于自动填充主键，由PolarDB-X自动维护。

扩展标准建表语法，增加了自增列的Sequence类型，如果未指定类型关键字，则默认类型为GROUP。PolarDB-X自动创建的、跟表相关联的Sequence名称，都是以 `AUTO_SEQ_` 为前缀，后面加上表名。

Group Sequence、Time-based Sequence或Simple Sequence

创建**Group Sequence**、**Time-based Sequence**或**Simple Sequence**作为自增列的表语法如下：

```
CREATE TABLE <name> (
   <column> ... AUTO_INCREMENT [ BY GROUP | SIMPLE | TIME ],
   <column definition>,
   ...
) ... AUTO_INCREMENT=<start value>
```

> **说明** 如果指定了 `BY TIME`，即Time-based Sequence，则该列类型必须为BIGINT。

单元化Group Sequence

创建**单元化Group Sequence**的表语法如下：

```
CREATE TABLE <name> (
   <column> ... AUTO_INCREMENT [ BY GROUP ] [ UNIT COUNT <numeric value> INDEX <numeric value> ],
   <column definition>,
   ...
) ... AUTO_INCREMENT=<start value>
```

示例

示例一：默认创建一张使用Group Sequence作为自增列的表。

请在命令行输入如下代码：

```
mysql> CREATE TABLE tab1 (
col1 BIGINT NOT NULL AUTO_INCREMENT,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

示例二：创建3张同名的、使用相同单元数量和不同单元索引的单元化Group Sequence作为自增列的表，分别用于3个不同的实例或库。

1. 实例1/库1

   请在命令行输入如下代码：

   ```
   mysql> CREATE TABLE tab2 (
   col1 BIGINT NOT NULL AUTO_INCREMENT UNIT COUNT 3 INDEX 0,
   col2 VARCHAR(16),
   PRIMARY KEY(col1)
   ) DBPARTITION BY HASH(col1);
   ```

2. 实例2/库2

   请在命令行输入如下代码：

```
mysql> CREATE TABLE tab2 (
col1 BIGINT NOT NULL AUTO_INCREMENT UNIT COUNT 3 INDEX 1,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

3. 实例3/库3

   请在命令行输入如下代码：

```
mysql> CREATE TABLE tab2 (
col1 BIGINT NOT NULL AUTO_INCREMENT UNIT COUNT 3 INDEX 2,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

示例三：创建一张使用Time-based Sequence作为自增列的表。

请在命令行输入如下代码：

```
mysql> CREATE TABLE tab3 (
col1 BIGINT NOT NULL AUTO_INCREMENT BY TIME,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

示例四：创建一张使用Simple Sequence作为自增列的表。

请在命令行输入如下代码：

```
mysql> CREATE TABLE tab4 (
col1 BIGINT NOT NULL AUTO_INCREMENT BY SIMPLE,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

8.16.4.2. 修改 Sequence

本文主要介绍如何对表相关的Sequence类型进行修改。

暂不支持通过 `ALTER TABLE` 来修改对应Sequence的类型，但您可以参见如下语法通过 `ALTER TABLE` 修改起始值：

```
ALTER TABLE <name> ... AUTO_INCREMENT=<start value>
```

> **说明**
>
> - 如果想要修改表相关的Sequence类型，需要通过 `SHOW SEQUENCES` 指令查找出Sequence的具体名称和类型，然后再用 `ALTER SEQUENCE` 指令去修改。
> - 使用Sequence后，请谨慎修改 `AUTO_INCREMENT` 的起始值（仔细评估已经产生的Sequence值，以及生成新Sequence值的速度，防止产生冲突）。

8.16.4.3. 查看表信息及相关Sequence类型

本文主要介绍如何对使用Sequence类型作为自增列的表信息以及相关Sequence类型进行查看。

SHOW CREATE TABLE

当表为拆分表或者广播表时，显示自增列Sequence的类型。

查看已创建的表语法如下：

```
SHOW CREATE TABLE <name>
```

说明

- `SHOW CREATE TABLE` 仅显示相关Sequence的类型，并不显示Sequence详细信息，如需查看，请使用 `SHOW SEQUENCES` 命令。
- 关联了单元化Group Sequence的表并不显示单元数量和单元索引，因此不能将 `SHOW CREATE TABLE` 显示的DDL直接用于创建具备同样**单元化Group Sequence**能力的表。
- 如果需要创建具备同样单元化能力的表，必须使用 `SHOW SEQUENCES` 查看单元数量和单元索引，然后参照 `CREATE TABLE` 的语法修改通过 `SHOW CREATE TABLE` 获取的建表DDL。

示例

示例一：建表时指定 `AUTO_INCREMENT` ，但没有指定Sequence类型关键字，则默认使用**Group Sequence**。

请在命令行输入如下代码：

```
mysql> SHOW CREATE TABLE tab1;
```

返回结果如下：

```
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
| Table | Create Table
|
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
| tab1   | CREATE TABLE `tab1` (
`col1` bigint(20) NOT NULL AUTO_INCREMENT BY GROUP,
`col2` varchar(16) DEFAULT NULL,
PRIMARY KEY (`col1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`col1`) |
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
1 row in set (0.02 sec)
```

示例二：建表时为 `AUTO_INCREMENT` 指定了单元数量和单元索引，使用**单元化 Group Sequence**，但 `SHOW CREATE TABLE` 时并不显示单元数量和单元索引，不能将此DDL用于创建具备同样**单元化 Group Sequence**能力的表。

请在命令行输入如下代码：

```
mysql> SHOW CREATE TABLE tab2;
```

返回结果如下：

```
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
| Table | Create Table
|
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
| tab2   | CREATE TABLE `tab2` (
`col1` bigint(20) NOT NULL AUTO_INCREMENT BY GROUP,
`col2` varchar(16) DEFAULT NULL,
PRIMARY KEY (`col1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`col1`) |
+-------+-----------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

示例三：建表时为 `AUTO_INCREMENT` 指定了 `BY TIME` ，即Time-based Sequence类型。

请在命令行输入如下代码：

```
mysql> SHOW CREATE TABLE tab3;
```

返回结果如下：

```
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| Table | Create Table
|
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| tab3  | CREATE TABLE `tab3` (
`col1` bigint(20) NOT NULL AUTO_INCREMENT BY TIME,
`col2` varchar(16) DEFAULT NULL,
PRIMARY KEY (`col1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`col1`) |
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

示例四：建表时为 `AUTO_INCREMENT` 指定了 `BY SIMPLE` ，即Simple Sequence类型。

请在命令行输入如下代码：

```
mysql> SHOW CREATE TABLE tab4;
```

返回结果如下：

```
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| Table | Create Table
|
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| tab3  | CREATE TABLE `tab4` (
`col1` bigint(20) NOT NULL AUTO_INCREMENT BY TIME,
`col2` varchar(16) DEFAULT NULL,
PRIMARY KEY (`col1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`col1`) |
+-------+----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

SHOW SEQUENCES

建表后相关的Sequence名称和详细信息，可通过 `SHOW SEQUENCES` 查看。

请在命令行输入如下代码：

```
mysql> SHOW SEQUENCES;
```

返回结果如下：

```
+---------------+--------+------------+------------+------------+-------------+------------+---------------
------+-------+--------+
| NAME          | VALUE  | UNIT_COUNT | UNIT_INDEX | INNER_STEP | INCREMENT_BY | START_WITH | MAX_VALUE
| CYCLE | TYPE   |
+---------------+--------+------------+------------+------------+-------------+------------+---------------
------+-------+--------+
| seq1          | 100000 | 1          | 0          | 100000     | N/A          | N/A        | N/A
| N/A   | GROUP  |
| seq2          | 400000 | 3          | 1          | 100000     | N/A          | N/A        | N/A
| N/A   | GROUP  |
| seq3          | N/A    | N/A        | N/A        | N/A        | N/A          | N/A        | N/A
| N/A   | TIME   |
| seq4          | 1006   | N/A        | N/A        | N/A        | 2            | 1000       | 99999999999
| N     | SIMPLE |
| AUTO_SEQ_tab1 | 100000 | 1          | 0          | 100000     | N/A          | N/A        | N/A
| N/A   | GROUP  |
| AUTO_SEQ_tab2 | 400000 | 3          | 1          | 100000     | N/A          | N/A        | N/A
| N/A   | GROUP  |
| AUTO_SEQ_tab3 | N/A    | N/A        | N/A        | N/A        | N/A          | N/A        | N/A
| N/A   | TIME   |
| AUTO_SEQ_tab4 | 2      | N/A        | N/A        | N/A        | 1            | 1          | 92233720368547
75807 | N     | SIMPLE |
+---------------+--------+------------+------------+------------+-------------+------------+---------------
------+-------+--------+
8 rows in set (0.01 sec)
```

8.17. TTL表功能

8.17.1. 什么是TTL功能

本文介绍了TTL的功能和原理。

功能介绍

TTL（Time to Live，生存周期）功能，支持在创建表的DDL语句中指定local_partition_definition语法，创建一个TTL表。TTL表会将每个物理表按照时间进行分区，并通过定时任务进行管理，能够让冷数据在PolarDB-X中按照计划失效，减少对存储空间的占用。

适用场景

某些业务场景下，业务数据的增长很快，并且业务数据的热度随着时间推移会有明显的降低。这时候，如果数据一直存储在PolarDB-X中，既会占用存储空间，也会降低正常业务查询的效率，此种场景即可使用TTL表进行数据管理。

使用限制

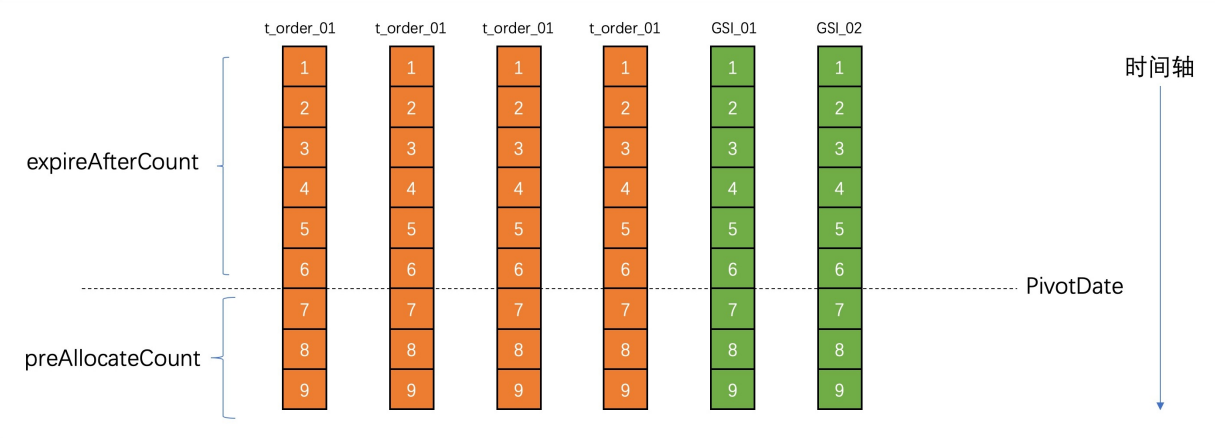
- 仅PolarDB-X的5.4.13及以上版本支持该功能；
- TTL功能仅支持在分区表中使用。

基本原理

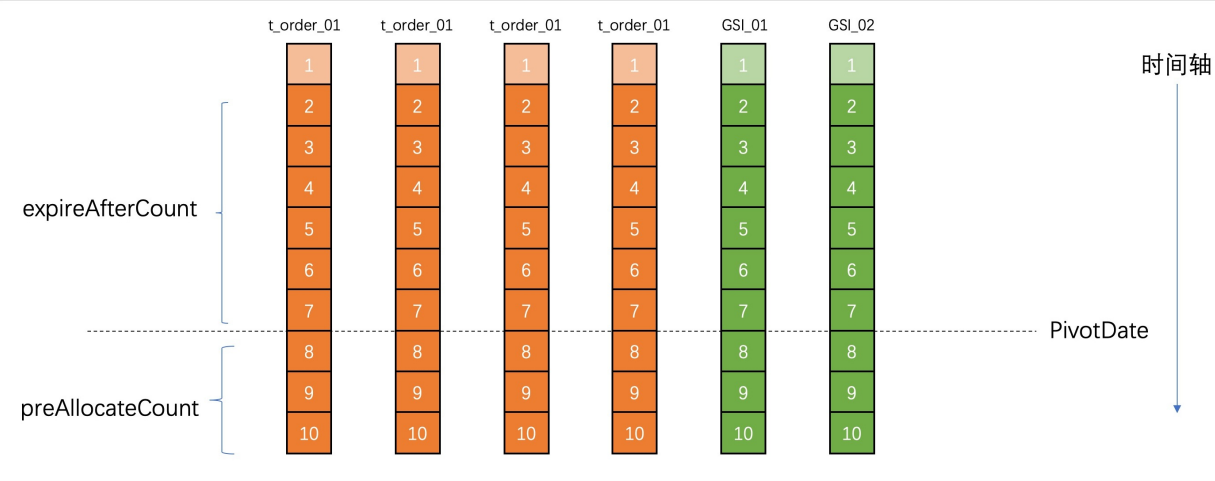
TTL表的每个物理表都会按照时间做Range分区，所有GSI的物理表跟主表的分区方式一致。

随着时间的推移，新的分区会被创建，失效的分区会被删除，如下图所示：

> **说明** PolarDB-X默认会自动创建新分区和删除失效分区。您也可以在建表时禁用自动功能，通过DDL语句来手动创建或删除失效分区。



一个周期后，创建的新分区和失效的分区，如下图所示：



8.17.2. 创建TTL表

本文介绍了TTL表相关的CREATE TABLE语法。

若在建表时指定local_partition_definition语法，则将创建一张TTL表，TTL表会将每个物理表按照时间进行分区。

如果需要将t_order的物理表按照gmt_modified列进行时间分区，每个分区间隔一个月，每个分区12个月后过期，提前创建3个分区，创建语法如下：

```
CREATE TABLE t_order (
    id bigint NOT NULL AUTO_INCREMENT,
    gmt_modified DATETIME NOT NULL,
    PRIMARY KEY (id, gmt_modified)
)
PARTITION BY HASH(id)
PARTITIONS 16
LOCAL PARTITION BY RANGE (gmt_modified)
INTERVAL 1 MONTH
EXPIRE AFTER 12
PRE ALLOCATE 3;
```

参数说明

参数	是否必填	示例	说明
LOCAL PARTITION BY RANGE (column_name)	是	LOCAL PARTITION BY RANGE (gmt_modified)	指定TTL表的物理时间分区列。在物理表上数据会以此列做时间分区。

参数	是否必填	示例	说明
STARTWITH 'yyyy-MM-dd'	否	STARTWITH '2022-01-01'	初始时间分区。TTL表会把“初始时间分区”作为第一个分区。
INTERVAL interval_count MONTH	是	INTERVAL 2 MONTH	时间分区的间隔，支持的时间粒度为YEAR、MONTH、DAY。
EXPIRE AFTER expire_after_count	否	EXPIRE AFTER 6	指定分区失效的时间。 INTERVAL 2 MONTHEXPIRE AFTER 6 表示每隔2个月划分一个物理时间分区每经过6倍物理时间分区（12个月），失效1个分区。
PRE ALLOCATE pre_allocate_count	否	PRE ALLOCATE 2	指定提前创建的分区数量。 示例表示提前创建2个物理时间分区。
PIVOTDATE pivotdate_func	否	PIVOTDATE NOW()	指定基准时间，一般用默认值。
DISABLE SCHEDULE	否	DISABLE SCHEDULE	禁用自动分区滚动。 如果开启的话，过期分区会自动被删除，新分区会自动提前创建。

语法限制

- TTL表支持的时间分区列类型为：date、datetime;
- 所有的唯一键（包括主键）必须包含TTL表的local partition by range时间分区列;
- TTL表最多支持32个最大物理时间分区。

查看TTL表的分区元数据

TTL表创建后，可以通过 `information_schema.local_partitions` 视图查看物理表上的分区信息。每个物理时间分区（Local Partition）会存储小于 `PARTITION_DESCRIPTION` 的数据。比如下面的例子中，'2021-11-15'的数据会存储到分区：p20211201。

每个物理时间分区（Local Partition）的失效时间会通过 `PARTITION_COMMENT` 显示。

```
select * from information_schema.local_partitions where table_name = 't_create_lxok' and table_schema='local
_partition'\G;
*************************** 1. row ***************************
         TABLE_SCHEMA: local_partition
           TABLE_NAME: t_create_lxok
       PARTITION_NAME: p20211101
     PARTITION_METHOD: RANGE COLUMNS
 PARTITION_EXPRESSION: `gmt_modified`
PARTITION_DESCRIPTION: '2021-11-01'
    PARTITION_COMMENT: expire:2022-11-01
*************************** 2. row ***************************
         TABLE_SCHEMA: local_partition
           TABLE_NAME: t_create_lxok
       PARTITION_NAME: p20211201
     PARTITION_METHOD: RANGE COLUMNS
 PARTITION_EXPRESSION: `gmt_modified`
PARTITION_DESCRIPTION: '2021-12-01'
    PARTITION_COMMENT: expire:2022-12-01
*************************** 3. row ***************************
         TABLE_SCHEMA: local_partition
           TABLE_NAME: t_create_lxok
       PARTITION_NAME: pmax
     PARTITION_METHOD: RANGE COLUMNS
 PARTITION_EXPRESSION: `gmt_modified`
PARTITION_DESCRIPTION: MAXVALUE
    PARTITION_COMMENT:
7 rows in set (13.19 sec)
```

校验物理分区的完整性

校验所有物理表（包括GSI）的物理时间分区（Local Partition）的完整性。

```
check table t_order with local partition;
```

```
mysql> check table t_order with local partition;
+-------------------------+--------+-----------------+------------------+
| TABLE                   | STATUS | PARTITION_COUNT | PARTITION_DETAIL |
+-------------------------+--------+-----------------+------------------+
| local_partition.t_order | OK     | 17              | -                |
+-------------------------+--------+-----------------+------------------+
1 row in set (0.10 sec)
```

8.17.3. 变更TTL表

本文介绍了TTL表相关的ALTER TABLE语法。

变更TTL表的定时任务

```
ALTER TABLE tbl_name
    local_partition_alter_options
local_partition_alter_options:
        ALLOCATE LOCAL PARTITION
  |    EXPIRE LOCAL PARTITION [local_partition_name]
  | REMOVE LOCAL PARTITIONING
  | local_partition_definition
local_partition_definition:
        LOCAL PARTITION BY RANGE (column_name)
    [STARTWITH 'yyyy-MM-dd']
    INTERVAL interval_count MONTH
    [EXPIRE AFTER expire_after_count]
    [PRE ALLOCATE pre_allocate_count]
    [PIVOTDATE pivotdate_func]
    [DISABLE SCHEDULE]
pivotdate_func:
        NOW()
  |    DATE_ADD(...)
  | DATE_SUB(...)
```

创建新的物理时间分区

此命令多次调用不会无限制创建新分区，只会创建预定时间内的新分区。

```
ALTER TABLE t_order ALLOCATE LOCAL PARTITION;
```

删除过期的物理时间分区

此命令多次调用不会无限制删除已有分区，只会删除过期分区。

- 可以指定物理时间分区名，可通过information_schema.local_partitions获取

  ```
  ALTER TABLE t_order EXPIRE LOCAL PARTITION p20210401;
  ```

- 不指定分区名时，删除所有已过期的分区

  ```
  ALTER TABLE t_order EXPIRE LOCAL PARTITION;
  ```

  > **说明** 未失效分区不允许删除，会报错。

TTL表转换成普通表

```
ALTER TABLE t_order REMOVE LOCAL PARTITIONING;
```

> **说明** TTL表转换成普通表需要进行数据搬迁，请在业务低峰期进行操作。

普通表转换成TTL表

```
ALTER TABLE t_order
  LOCAL PARTITION BY RANGE (gmt_modified)
  STARTWITH '2021-01-01'
  INTERVAL 1 MONTH
  EXPIRE AFTER 12
  PRE ALLOCATE 6;
```

> **说明** 普通表转换成TTL表需要进行数据搬迁，请在业务低峰期进行操作。

8.17.4. TTL表的定时任务

本文介绍了TTL表创建、查看及删除定时任务的方法。

在TTL表中创建定时任务成功后，每隔一段时间定时任务会扫描表中的所有物理时间分区。如果需要创建新分区，会被自动创建。如果需要删除过期分区，会被自动删除。过期时间和提前创建时间由建表时的local_partition_definition指定。

创建定时任务

```
CREATE SCHEDULE
    FOR partition_type
    ON [schemaName.]tableName
    CRON 'cron_expr'
    [TIMEZONE 'time_zone']
partition_type:
        LOCAL_PARTITION
cron_expr:
    '0 15 10 ? * *'    #每天上午10：15分过期
  | '0 15 10 ? * 6L'   #每月最后一个周五的早上10：15过期
  | '0 11 11 11 11 ?' #每个11月11日的早上11：11分过期
  | ...
time_zone:
    '+08:00'
  | '+00:00'
  | ...
```

查看TTL定时任务

创建TTL表时，会默认创建一个定时任务，用于物理时间分区（Local Partition）的调度。如果创建表时指定了DISABLE SCHEDULE语法，则不会自动创建该定时任务。

可以通过INFORMATION_SCHEMA.LOCAL_PARTITIONS_SCHEDULE视图来查看当前有哪些定时任务。

> **说明** 该定时任务只是定期扫描物理时间分区（Local Partition），具体的创建或删除分区的时间是由建表时的参数决定的。

```
SELECT * FROM INFORMATION_SCHEMA.LOCAL_PARTITIONS_SCHEDULE\G;
*************************** 1. row ***************************
     SCHEDULE_ID: 21
     SCHEMA_NAME: local_partition
      TABLE_NAME: t_order
          STATUS: ENABLED
   SCHEDULE_EXPR: 0 0 12 1/5 * ?
SCHEDULE_COMMENT: at 12:00 every 5 days
       TIME_ZONE: +00:00
  LAST_FIRE_TIME: 1970-01-01 08:00:00
  NEXT_FIRE_TIME: 2021-11-01 12:00:00
1 row in set (0.03 sec)
```

查看定时任务执行结果

```
show schedule result 21\G;
*************************** 1. row ***************************
SCHEDULE_ID: 21
  TIME_ZONE: +00:00
  FIRE_TIME: 2021-11-08 04:20
 START_TIME:
FINISH_TIME:
      STATE: QUEUED
     REMARK: NULL
 RESULT_MSG: NULL
*************************** 2. row ***************************
SCHEDULE_ID: 21
  TIME_ZONE: +00:00
  FIRE_TIME: 2021-11-08 04:19
 START_TIME: 2021-11-08 04:19:50
FINISH_TIME: 2021-11-08 04:19:50
      STATE: SUCCESS
     REMARK: NULL
 RESULT_MSG: NULL
```

删除定时任务

```
DROP SCHEDULE schedule_id;
```

示例

示例1

从每月第一天开始，每5天的中午12点过期，时区为+00:00。

```
CREATE SCHEDULE
FOR LOCAL_PARTITION
ON `local_partition`.`t_order`
CRON '0 0 12 1/5 * ?'
TIMEZONE '+00:00';
```

示例2

从每月第一天开始，每5天的中午12点过期。

```
CREATE SCHEDULE
FOR LOCAL_PARTITION
ON `local_partition`.`t_order`
CRON '0 0 12 1/5 * ?'
;
```

示例3

删除定时任务526。

```
DROP SCHEDULE 526;
```

8.18. 变量

8.18.1. 计算层变量

PolarDB-X是存储计算分离架构，这里主要罗列了常见CN变量（其中部分变量控制台可配置）。

变量名	是否重启	默认值	范围	备注
PLAN_CACHE	否	TRUE	[TURE|FALSE]	计划缓存的开关。
ENABLE_RECYCLEBIN	否	FALSE	[TURE|FALSE]	开启回收站的开关。
SHOW_TABLES_CACHE	否	FALSE	[TURE|FALSE]	是否对 `show tables` 结果做缓存。
MERGE_CONCURRENT	否	FALSE	[TURE|FALSE]	全并行执行开关，主要影响DDL和全下推简单查询的并行执行度。
MERGE_UNION	否	FALSE	[TURE|FALSE]	默认关闭，打开意味着物理SQL不做union优化，且将串行执行下推的物理SQL。
MERGE_UNION_SIZE	否	-1	[0-10000]	通过union合并物理SQL的数量，默认结合连接池可用连接数自适应推导。
TABLE_META_CACHE_EXPIRE_TIME	否	300	[0-180000]	元数据缓存过期时间。
COLUMN_LABEL_INSENSITIVE	否	TRUE	[FALSE|TRUE]	返回列大小写是否敏感。
RECORD_SQL	否	TRUE	[FALSE|TRUE]	审计日志开关。
SOCKET_TIMEOUT	否	900000	[0~3600000]	物理SQL超时时间。
TRANSACTION_POLICY	否	TSO	[XA|TSO|TSO_READONLY]	事务策略。
SHARE_READ_VIEW	否	FALSE	[TRUE|FALSE]	共享ReadView开关。
ENABLE_TRX_SINGLE_SHARD_OPTIMIZATION	否	TRUE	[TRUE|FALSE]	事务单分片优化开关。
GET_TSO_TIMEOUT	否	10	[1-1800]	获取TSO时间戳超时时间。
MAX_TRX_DURATION	否	28800	[1-180000]	事务物理超时时间。

变量名	是否重启	默认值	范围	备注
TRANSACTION_ISOLATION	否	REPEATABLE_READ	[READ_UNCOMMITTED|READ_COMMITTED|REPEATABLE_READ|SERIALIZABLE]	事务隔离级别。
GROUP_CONCURRENT_BLOCK	否	TRUE	[TRUE|FALSE]	非MPP模式下分库级别执行策略。
SEQUENTIAL_CONCURRENT_POLICY	否	FALSE	[TRUE|FALSE]	非MPP模式下单并发执行策略。
DML_SKIP_DUPLICATE_CHECK_FOR_PK	否	TRUE	[TRUE|FALSE]	DML过程中是否跳过主键的冲突检查。
DML_SKIP_CRUCIAL_ERR_CHECK	否	FALSE	[TRUE|FALSE]	DML过程中是否允许有DML报错的事务继续提交。
DML_USE_RETURNING	否	TRUE	[TRUE|FALSE]	是否使用returning优化。
BROADCAST_DML	否	FALSE	[TRUE|FALSE]	是否允许广播表的写入不走分布式事务。
SEQUENCE_STEP	否	10000	[1-10000000]	SEQUENCE步长，默认为10万。
MERGE_DDL_TIMEOUT	否	0	[1-10000000]	DDL物理连接超时时间，默认是0，不超时。
MERGE_DDL_CONCURRENT	否	FALSE	[FALSE|TRUE]	DDL是否采用全并行模式，默认库级并发。
SLOW_SQL_TIME	否	1000	[1-180000]	慢SQL阈值。
LOAD_DATA_BATCH_INSERT_SIZE	否	1024	[1-180000]	LOAD DATA每次batch insert的记录条数。
LOAD_DATA_CACHE_BUFFER_SIZE	否	60	[1-180000]	LOAD DATA缓存大小，默认60Mb。主要用途为流控。
MAX_ALLOWED_PACKET	否	16777216	[4194304-33554432]	最大包大小。
KILL_CLOSE_STREAM	否	FALSE	[FALSE|TRUE]	是否开启物理连接流式早停功能。

变量名	是否重启	默认值	范围	备注
ALLOW_SIMPLE_SEQUENCE	否	FALSE	[FALSE|TRUE]	是否允许使用simple sequence。
MAX_PARAMETERIZED_SQL_LOG_LENGTH	否	5000	[1-1000000]	参与参数化SQL日志打印的最大长度。
FORBID_EXECUTE_DML_ALL	否	TRUE	[TRUE|FALSE]	是否禁止全表删除/更新。
GROUP_SEQ_CHECK_INTERVAL	是	60	[1-36000]	检查插入显式值的周期/间隔，单位秒。
JOIN_BLOCK_SIZE	是	300	[1-100000]	在非动态裁剪下BKAJOIN执行时的IN Values个数。
LOOKUP_JOIN_MAX_BATCH_SIZE	是	6400	[1-100000]	BKAJOIN执行时的最大IN Values个数。
LOOKUP_JOIN_MIN_BATCH_SIZE	是	100	[1-100000]	BKAJOIN执行时的最大IN Values个数。
PURGE_TRANS_INTERVAL	是	300	[1-180000]	事务日志清理间隔。
PURGE_TRANS_BEFORE	是	1800	[1-180000]	清理多久之前的事务日志。
ENABLE_BACKGROUND_STATISTIC_COLLECTION	否	TRUE	[TRUE|FALSE]	是否允许统计数据采集。
GENERAL_DYNAMIC_SPEED_LIMITATION	否	-1	[-1-10000000]	数据回填、校验动态限速调整，-1为默认限制。
PARALLELISM	否	-1	[1-1024]	单机并行的并行度，默认是由规格推导出来。
LOGICAL_DB_TIME_ZONE	否	SYSTEM	[SYSTEM|±HH:mm]	数据库时区。
MPP_PARALLELISM	否	-1	[1-1024]	MPP执行模式的并发度，默认是由规格推导出来。
DATABASE_PARALLELISM	否	0	[0-1024]	单个查询在一个DN上允许同时下发SQL的数量，用户计算Scan的并发度。

变量名	是否重启	默认值	范围	备注
POLARDBX_PARALLELISM	否	0	[0-1024]	单个查询在一个CN允许的最大并发度，默认是CPU核数。
MPP_METRIC_LEVEL	否	3	[0|1|2|3]	计算过程中统计信息收集程度，级别越高说明收集粒度越细。
ENABLE_COMPLEX_DML_CROSS_DB	否	TRUE	[TRUE|FALSE]	是否支持跨库复杂DML。
PER_QUERY_MEMORY_LIMIT	是	-1	[-1-922337203685477580 7]	查询级内存池大小限制，默认是全局连接池的三分之一。
ENABLE_SPILL	否	FALSE	[FALSE|TRUE]	临时表落盘的开关。
CONN_POOL_MIN_POOL_SIZE	否	20	[0-10]	物理分库链接数最小值。
CONN_POOL_MAX_POOL_SIZE	否	60	[1-1600]	物理分库链接数最大值。
CONN_POOL_MAX_WAIT_THREAD_COUNT	否	0	[-1-8192]	单分库最大等待建连数（DRUID）。
CONN_POOL_IDLE_TIMEOUT	否	30	[1-60]	物理空闲链接超时时间。
CONN_POOL_BLOCK_TIMEOUT	否	5000	[1000-60000]	物理连接池获取连接的最大等待时间。
CONN_POOL_XPROTO_MAX_POOLED_SESSION_PER_INST	否	512	[1-8192]	单个存储节点最大缓存会话数（私有协议）。
XPROTO_MAX_DN_CONCURRENT	否	500	[1-8192]	单个存储节点最大并发请求数（私有协议）。
XPROTO_MAX_DN_WAIT_CONNECTION	否	32	[1-8192]	单个存储节点最大请求等待数（私有协议）
MERGE_SORT_BUFFER_SIZE	否	2048	[1024-81920]	TableScan层做归并排序使用的缓存大小，默认2Mb。

变量名	是否重启	默认值	范围	备注
WORKLOAD_TYPE	否		[AP|TP]	是否指定查询的workload，默认基于代价智能识别负载。
EXECUTOR_MODE	否		[MPP|TP_LOCAL|AP_LOCAL]	是否指定查询的执行模式，默认基于workload选择执行模式。
ENABLE_MASTER_MPP	否	FALSE	[TRUE|FALSE]	是否在主实例上开启MPP能力。
LOOKUP_JOIN_BLOCK_SIZE_PER_SHARD	是	50	[1-100000]	在裁剪下BKAJOIN执行时的单分片约定的IN Values个数。
ENABLE_RUNTIME_FILTER	否	TRUE	[TRUE|FALSE]	Runtime Filter的开关。
FEEDBACK_WORKLOAD_AP_THRESHOLD	否	FALSE	[TRUE|FALSE]	针对AP查询的HTAP FEEDBACK开关。
FEEDBACK_WORKLOAD_TP_THRESHOLD	否	FALSE	[TRUE|FALSE]	针对TP查询的HTAP FEEDBACK开关。
MASTER_READ_WEIGHT	否	-1	[0-100]	查询流量按照(100-MASTER_READ_WEIGHT)设置比例路由给只读DN。
SHOW_ALL_PARAMS	否	FALSE	[TRUE|FALSE]	是否显示SHOW全部变量。
ENABLE_SET_GLOBAL	否	FALSE	[TRUE|FALSE]	启动SET GLOBAL语句开关。
FORCE_READ_OUTSIDE_TX	否	FALSE	[TRUE|FALSE]	事务内是否在一个分库上强制开启多个连接。
ENABLE_COROUTINE	是	FALSE	[TRUE|FALSE]	是否开启wisp协程。
TRUNCATE_TABLE_WITH_GSI	否	FALSE	[TRUE|FALSE]	是否允许truncate包含GSI的表。
DDL_ON_GSI	否	FALSE	[TRUE|FALSE]	是否允许直接在GSI表做DDL。
DML_ON_GSI	否	FALSE	[TRUE|FALSE]	是否允许直接在GSI表做DML。
ENABLE_HASH_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否与允许HashJoin节点的生成。

变量名	是否重启	默认值	范围	备注
ENABLE_BKA_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许BKAJoin节点生成。
ENABLE_NL_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许NLJoin节点生成。
ENABLE_SEMI_NL_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin转成NLJoin来实现。
ENABLE_SEMI_HASH_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin转成HashJoin来实现。
ENABLE_SEMI_BKA_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin转成BKAJoin来实现。
ENABLE_SEMI_SORT_MERGE_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin转成MergeJoin来实现。
ENABLE_MATERIALIZED_SEMI_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin转成MaterializedJoin来实现。
ENABLE_SEMI_JOIN_REORDER	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将SemiJoin参与CBO Reorder优化。
ENABLE_HASH_AGG	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许生成HashAgg节点生成。
ENABLE_PARTIAL_AGG	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许将Agg拆分成二阶段来实现。
ENABLE_SORT_AGG	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许生成SortAgg节点生成。
ENABLE_PUSH_PROJECT	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Project PushDown。
ENABLE_PUSH_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Join PushDown。
ENABLE_PUSH_AGG	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Agg PushDown。
ENABLE_CBO_PUSH_AGG	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Agg透传Join。
ENABLE_PUSH_SORT	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Sort PushDown。

变量名	是否重启	默认值	范围	备注
ENABLE_STATISTIC_FEEDBACK	否	TRUE	[TRUE|FALSE]	是否支持统计信息修正的Feedback。
ENABLE_CBO_PUSH_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Join透传优化。
ENABLE_SORT_JOIN_TRANSPOSE	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否允许Sort透传Join。
CHUNK_SIZE	否	1024	[1-10240]	设置执行器每次计算的批次大小。
ENABLE_SORT_MERGE_JOIN	否	TRUE	[TRUE|FALSE]	优化查询计划过程中是否禁止MergeJoin节点生成。
ENABLE_BKA_PRUNING	否	TRUE	[TRUE|FALSE]	是否开启BKAJoin的裁剪功能。
ENABLE_SPM	否	TRUE	[TRUE|FALSE]	是否开启执行计划管理。
ENABLE_EXPRESSION_VECTORIZATION	否	TRUE	[TRUE|FALSE]	是否开启表达式的向量化计算。
FORCE_DDL_ON_LEGACY_ENGINE	否	TRUE	[TRUE|FALSE]	是否启用新DDL引擎。
PURE_ASYNC_DDL_MODE	否	FALSE	[TRUE|FALSE]	是否以非阻塞的形式执行DDL任务，开启意味着客户端执行DDL后立即返回，通过show [full] ddl查看执行的情况。
DDL_JOB_REQUEST_TIMEOUT	否	90000	[1-9223372036854775807]	设置DDL执行的最大超时时间，默认是25天。
LOGICAL_DDL_PARALLELISM	否	1	[1-10240]	配置逻辑DDL执行的并发度，设置为1意味着是串行执行DDL任务。
ENABLE_BROADCAST_RANDOM_READ	否	TRUE	[TRUE|FALSE]	是否开启广播表随机读优化。
STORAGE_DELAY_THRESHOLD	否	16777216s	[0-16777216]	查询过程中CN感知DN的延迟程度。如果延迟超过设置阈值，则路由给只读DN的只读流量会根据DELAY_EXECUTION_STRATEGY参数的设置策略执行。

变量名	是否重启	默认值	范围	备注
DELAY_EXECUTION_STRATEGY	否	CONTINUE_TO_SLAVE	[CONTINUE_TO_SLAVE | CHANGE_TO_MASTER | THROW_EXCEPTION]	当出现只读DN延迟超过阈值情况时，可设置如下三个取值： • CONTINUE_TO_SLAVE：路由给只读DN的只读流量可继续路由给该延迟超过阈值的只读DN； • CHANGE_TO_MASTER：只读流量会优先路由到延迟正常的只读DN，若所有只读DN延迟都超过阈值，则将流量路由到主实例的DN； • THROW_EXCEPTION：若所有只读DN延迟都超过阈值，路由给只读DN的查询会直接报错。
STORAGE_BUSY_THRESHOLD	否	100	[0-16777216]	当有多个只读DN时，如果某个只读DN上的活跃线程数超过设置阈值，则认为当前DN处于繁忙状态，只读流量路由时会避开该只读DN。如果所有的只读DN都处于繁忙状态，那么只读流量会随机路由给一个只读DN。
ENABLE_CONSISTENT_REPLICA_READ	否	TRUE	[TURE|FALSE]	一致性读需求的开关项。 TRUE：路由给只读DN的流量可以满足一致性读的需求，读到已提交最新鲜的数据。 FALSE：路由给只读DN的流量读取数据的新鲜程度依赖于主备之间的同步延迟。
SUPPORT_INSTANT_ADD_COLUMN	否	OFF	[ON|OFF]	秒级加字段（Instant Add Column）功能的开关，通过 `SET GLOBAL` 语句开启。

8.18.2. 存储层变量

PolarDB-X是存储计算分离架构，存储层（DN）的变量名和含义与MySQL对齐，这里主要罗列了常见的DN变量（控制台可配置），其他变量可以参考MySQL变量。

名称	是否重启	默认值	修改范围	备注
loose_enable_gts	是	1	[0|1]	全局时间戳开关
loose_gts_lease	是	5000	[0-10000]	全局时间戳的租约时间
loose_performance-schema-instrument	否	wait/lock/metadata/sql/mdl=ON	.*	用于获取MySQL运行时的MDL锁信息
performance_schema	否	ON	[ON,OFF]	用于监控MySQL服务器的执行细节

名称	是否重启	默认值	修改范围	备注
binlog_rows_query_key_content	否	ON	[ON,OFF]	打印sql到binlog日志
loose_polarx_max_allowed_packet	否	16777216	(0-2147483648)	限制私有协议下Server接受和发送的数据包大小
innodb_buffer_pool_load_at_startup	是	ON	[ON|OFF]	是否启动之后,重新加载buffer pool
bulk_insert_buffer_size	否	4194304	[0-4294967295]	用来缓存批量插入数据的时候临时缓存写入数
show_old_temporals	否	OFF	[ON|OFF]	
ft_query_expansion_limit	是	20	[0-1000]	指定MyISAM引擎表使用with query expansion进行全文搜索的最大匹配数
innodb_old_blocks_time	否	1000	[0-1024]	用来表示页读取到mid位置后，需要等待多久才会被加入到LRU列表的热端
innodb_stats_sample_pages	否	8	[1-4294967296]	控制采集精度
thread_stack	是	262144	[131072-18446744073709551615]	每个连接线程被创建时，MySQL给它分配的内存大小
lc_time_names	否	en_US	[ja_JP|pt_BR|en_US]	控制用于显示日期和月份名称及缩写的语言
innodb_thread_concurrency	否	0	[0-1000]	并发数限制

名称	是否重启	默认值	修改范围	备注
default_time_zone	是	SYSTEM	[SYSTEM|-12:00|-11:00|-10:00|-9:00|-8:00|-7:00|-6:00|-5:00|-4:00|-3:00|-2:00|-1:00|\+0:00|\+1:00|\+2:00|\+3:00|\+4:00|\+5:00|\+5:30|\+5:45|\+6:00|\+6:30|\+7:00|\+8:00|\+9:00|\+10:00|\+11:00|\+12:00|\+13:00]	时区设置
old_passwords	否	0	[0|2]	
optimizer_search_depth	否	62	[0-62]	在多表关联场景下，控制优化器递归深度
innodb_compression_level	否	6	[0-9]	
loose_innodb_log_optimize_ddl	否	OFF	[ON|OFF]	
max_sort_length	否	1024	[4-8388608]	
slave_pending_jobs_size_max	否	1073741824	[1024-18446744073709551615]	
innodb_online_alter_log_max_size	否	134217728	[134217728-2147483647]	
key_cache_block_size	否	1024	[512-16384]	
mysql_native_password_proxy_users	否	OFF	[ON|OFF]	
innodb_adaptive_max_sleep_delay	否	150000	[1-1000000]	
innodb_purge_rseg_truncate_frequency	否	128	[1-128]	
query_alloc_block_size	否	8192	[1024-16384]	

名称	是否重启	默认值	修改范围	备注
innodb_lock_wait_timeout	否	50	[1-1073741824]	
innodb_purge_threads	是	1	[1-32]	
innodb_compression_failure_threshold_pct	否	5	[0-100]	
innodb_compression_pad_pct_max	否	50	[0-70]	
binlog_rows_query_log_events	否	ON	[ON|OFF]	
innodb_stats_persistent_sample_pages	否	20	[0-4294967295]	
innodb_ft_total_cache_size	是	640000000	[32000000-1600000000]	
innodb_flush_method	是	O_DIRECT	[fsync|O_DSYNC|littlesync|nosync|O_DIRECT|O_DIRECT_NO_FSYNC]	
eq_range_index_dive_limit	否	10	[0-4294967295]	
loose_max_execution_time	否	0	[0-4294967295]	
loose_optimizer_trace_features	否	greedy_search=on,range_optimizer=on,dynamic_range=on,repeated_subselect=on	.*	
rds_reserved_connections	否	512	[0-512]	
connect_timeout	否	10	[1-3600]	
innodb_purge_batch_size	是	300	[1-5000]	
div_precision_increment	否	4	[0-30]	

名称	是否重启	默认值	修改范围	备注
avoid_temporal_upgrade	否	OFF	[ON|OFF]	
innodb_sync_array_size	是	1	[1-64]	
sync_binlog	否	1	[0-2147483647]	
innodb_stats_method	否	nulls_equal	[nulls_equal|nulls_unequal|nulls_ignored]	
lock_wait_timeout	否	31536000	[1-1073741824]	
net_read_timeout	否	30	[1-184467440737095516 15]	
innodb_deadlock_detect	否	ON	[ON|OFF]	
innodb_write_io_threads	是	4	[1-64]	
end_markers_in_json	否	OFF	[ON|OFF]	
ngram_token_size	是	2	[0-20]	
loose_innodb_numa_interleave	是	ON	[ON|OFF]	
max_binlog_stmt_cache_size	否	184467440737095475 20	[4096-184467440737095475 20]	
innodb_checksum_algorithm	否	crc32	[innodb|crc32|none|strict_innodb|strict_crc32|strict_none]	
query_cache_type	是	0	[0|1|2]	
innodb_ft_enable_diag_print	否	OFF	[ON|OFF]	
innodb_ft_enable_stopword	否	ON	[ON|OFF]	

名称	是否重启	默认值	修改范围	备注
innodb_io_capacity	否	20000	[0-18446744073709551615]	
slow_launch_time	否	2	[1-1024]	
innodb_table_locks	否	ON	[ON|OFF]	
innodb_stats_persistent	否	ON	[ON|OFF]	
tmp_table_size	否	2097152	[262144-134217728]	
disconnect_on_expired_password	是	ON	[ON|OFF]	
default_storage_engine	是	InnoDB	[InnoDB|innodb]	
net_retry_count	否	10	[1-4294967295]	
innodb_ft_cache_size	是	8000000	[1600000-80000000]	
binlog_cache_size	否	2097152	[4096-16777216]	
innodb_max_dirty_pages_pct	否	75	[0-99]	
query_cache_limit	否	1048576	[1-1048576]	
innodb_disable_sort_file_cache	否	ON	[ON|OFF]	
innodb_lru_scan_depth	否	1024	[100-18446744073709551615]	
innodb_ft_result_cache_limit	否	2000000000	[1000000-4294967295]	
long_query_time	否	1	[0.03-31536000]	
interactive_timeout	否	7200	[10-86400]	

名称	是否重启	默认值	修改范围	备注
innodb_read_io_threads	是	4	[1-64]	
transaction_prealloc_size	否	4096	[1024-131072]	
open_files_limit	是	65535	[1-184467440737095516 15]	
innodb_open_files	是	3000	[10-4294967295]	
max_heap_table_size	否	67108864	[16384-184467440737095475 2]	
automatic_sp_privileges	否	ON	[ON|OFF]	
explicit_defaults_for_timestamp	否	OFF	[ON|OFF]	
ft_max_word_len	是	84	[10-4294967295]	
innodb_autoextend_increment	否	64	[1-1000]	

名称	是否重启	默认值	修改范围	备注
sql_mode	否		(\s*|REAL_AS_FLOAT|PIPES_AS_CONCAT|ANSI_QUOTES|IGNORE_SPACE|ONLY_FULL_GROUP_BY|NO_UNSIGNED_SUBTRACTION|NO_DIR_IN_CREATE|POSTGRESQL|ORACLE|MSSQL|DB2|MAXDB|NO_KEY_OPTIONS|NO_TABLE_OPTIONS|NO_FIELD_OPTIONS|MYSQL323|MYSQL40|ANSI|NO_AUTO_VALUE_ON_ZERO|NO_BACKSLASH_ESCAPES|STRICT_TRANS_TABLES|STRICT_ALL_TABLES|NO_ZERO_IN_DATE|NO_ZERO_DATE|ALLOW_INVALID_DATES|ERROR_FOR_DIVISION_BY_ZERO|TRADITIONAL|HIGH_NOT_PRECEDENCE|NO_ENGINE_SUBSTITUTION|PAD_CHAR_TO_FULL_LENGTH|NO_AUTO_CREATE_USER) (,NO_AUTO_CREATE_USER|,REAL_AS_FLOAT|,PIPES_AS_CONCAT|,ANSI_QUOTES|,IGNORE_SPACE|,ONLY_FULL_GROUP_BY|,NO_UNSIGNED_SUBTRACTION|,NO_DIR_IN_CREATE|,POSTGRESQL|,ORACLE|,MSSQL|,DB2|,MAXDB|,NO_KEY_OPTIONS|,NO_TABLE_OPTIONS|,NO_FIELD_OPTIONS|,MYSQL323|,MYSQL40|,ANSI|,NO_AUTO_VALUE_ON_ZERO|,NO_BACKSLASH_ESCAPES|,STRICT_TRANS_TABLES|,STRICT_ALL_TABLES|,NO_ZERO_IN_DATE|,NO_ZERO_DATE|,ALLOW_INVALID_DATES|,ERROR_FOR_DIVISION_BY_ZERO|,TRADITIONAL|,HIGH_NOT_PRECEDENCE|,NO_ENGINE_SUBSTITUTION|,PAD_CHAR_TO_FULL_LENGTH)*	
innodb_stats_transient_sample_pages	否	8	[1-4294967295]	

名称	是否重启	默认值	修改范围	备注
innodb_random_read_ahead	否	OFF	[ON|OFF]	
session_track_state_change	否	OFF	[ON|OFF]	
table_open_cache	否	2000	[1-524288]	表文件句柄高速缓存的大小
range_optimizer_max_mem_size	否	8388608	[0-18446744073709551615]	
innodb_status_output	否	OFF	[ON|OFF]	
innodb_log_compressed_pages	否	OFF	[ON|OFF]	
slave_net_timeout	否	60	[15-300]	
delay_key_write	否	ON	[ON|OFF|ALL]	
query_cache_wlock_invalidate	否	OFF	[ON|OFF]	
max_points_in_geometry	否	65536	[3-1048576]	
max_prepared_stmt_count	否	16382	[0-1048576]	
wait_timeout	否	86400	[1-31536000]	
query_cache_min_res_unit	否	1024	[512-18446744073709551608]	结果缓存配置
innodb_print_all_deadlocks	否	OFF	[OFF|ON]	
loose_thread_pool_size	否	32	[1-1024]	分组的数量，默认值：32。线程池中的线程被平均分到多个组中进行管理

名称	是否重启	默认值	修改范围	备注
binlog_stmt_cache_size	否	32768	[4096-16777216]	
transaction_isolation	否	READ-COMMITTED	[READ-UNCOMMITTED|READ-COMMITTED|REPEATABLE-READ|SERIALIZABLE]	事务隔离级别策略
innodb_buffer_pool_dump_at_shutdown	否	ON	[ON|OFF]	
query_prealloc_size	否	8192	[8192-1048576]	
key_cache_age_threshold	否	300	[100-4294967295]	
loose_rds_kill_connections	否	20	[0, 184467440737095516 15]	
transaction_alloc_block_size	否	8192	[1024-131072]	
optimizer_trace_limit	否	1	[0-4294967295]	
metadata_locks_cache_size	是	1024	[1-1048576]	
optimizer_prune_level	否	1	[0|1]	
innodb_max_purge_lag	否	0	[0-4294967295]	
innodb_buffer_pool_dump_pct	否	25	[1-100]	
innodb_max_dirty_pages_pct_lwm	否	0	[0-99]	
max_sp_recursion_depth	否	0	[0-255]	
innodb_status_output_locks	否	OFF	[ON|OFF]	

名称	是否重启	默认值	修改范围	备注
updatable_views_with_limit	否	YES	[YES|NO]	
binlog_row_image	否	full	[full|minimal]	
innodb_change_buffer_max_size	否	25	[0-50]	
innodb_optimize_fulltext_only	否	OFF	[ON|OFF]	
loose_opt_rds_last_error_gtid	否	ON	[ON|OFF]	
innodb_ft_max_token_size	是	84	[10-84]	
innodb_max_undo_log_size	否	1073741824	[10485760-18446744073709551615]	
slave_parallel_type	否	LOGICAL_CLOCK	DATABASE,LOGICAL_CLOCK	
loose_rds_check_core_file_enabled	否	ON	[ON|OFF]	
innodb_adaptive_hash_index	否	ON	[ON|OFF]	
innodb_sync_spin_loops	否	30	[0-4294967295]	
net_write_timeout	否	60	[1-18446744073709551615]	
flush_time	否	0	[0-31536000]	
lower_case_table_names	是	1	[0|1]	
sha256_password_proxy_users	否	OFF	[ON|OFF]	

名称	是否重启	默认值	修改范围	备注
character_set_filesystem	否	binary	[utf8|latin1|gbk|binary]	
innodb_flush_sync	否	ON	[ON|OFF]	
tls_version	是	TLSv1,TLSv1.1,TLSv1.2	[TLSv1,TLSv1.1,TLSv1.2|TLSv1,TLSv1.1]	
key_cache_division_limit	否	100	[1-100]	
delayed_insert_timeout	否	300	[1-3600]	
preload_buffer_size	否	32768	[1024-1073741824]	
innodb_read_ahead_threshold	否	56	[0-1024]	
loose_optimizer_switch	否	index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,subquery_materialization_cost_based=on,use_index_extensions=on	.*	
concurrent_insert	否	1	[0|1|2]	并发插入功能设置
block_encryption_mode	否	"aes-128-ecb"	["aes-128-ecb"|"aes-192-ecb"|"aes-256-ecb"|"aes-128-cbc"|"aes-192-cbc"|"aes-256-cbc"]	
slow_query_log	否	ON	[ON|OFF]	记录慢日志
net_buffer_length	否	16384	[1024-1048576]	

名称	是否重启	默认值	修改范围	备注
query_cache_size	否	3145728	[0-104857600]	
delayed_insert_limit	否	100	[1-4294967295]	
innodb_large_prefix	否	OFF	[ON|OFF]	
innodb_monitor_disable	否		all	
innodb_adaptive_flushing_lwm	否	10	[0-70]	
innodb_log_checksums	否	ON	[ON|OFF]	
delayed_queue_size	否	1000	[1-4294967295]	
session_track_gtids	否	OFF	[OFF|OWN_GTID|ALL_GTIDS]	
innodb_thread_sleep_delay	否	10000	[0-1000000]	
loose_rds_set_connection_id_enabled	否	ON	[ON|OFF]	
innodb_old_blocks_pct	否	37	[5-95]	
innodb_ft_sort_pll_degree	是	2	[1-16]	
log_slow_admin_statements	否	OFF	[ON|OFF]	
innodb_stats_on_metadata	否	OFF	[ON|OFF]	
stored_program_cache	否	256	[16-524288]	
group_concat_max_len	否	1024	[4-18446744073709547 52]	

名称	是否重启	默认值	修改范围	备注
innodb_sort_buffer_size	是	1048576	[65536-67108864]	
innodb_page_cleaners	是	1	[1-64]	
innodb_spin_wait_delay	否	6	[0-4294967295]	
myisam_sort_buffer_size	否	262144	[262144-16777216]	
innodb_rollback_segments	否	128	[1-128]	
innodb_commit_concurrency	是	0	[0-1000]	
innodb_concurrency_tickets	否	5000	[1-4294967295]	
table_definition_cache	否	512	[400-524288]	
auto_increment_increment	否	1	[1-65535]	
binlog_checksum	是	CRC32		
max_seeks_for_key	否	1844674407370950000	[1-18446744073709551615]	
sync_relay_log	否	1	[0-2147483647]	
max_length_for_sort_data	否	1024	[0-838860]	
back_log	是	3000	[0-65535]	
max_error_count	否	64	[0-65535]	
innodb_io_capacity_max	否	40000	[0-18446744073709551615]	

名称	是否重启	默认值	修改范围	备注
innodb_strict_mode	否	OFF	[ON|OFF]	
binlog_order_commits	是	OFF		
min_examined_row_limit	否	0	[0-4294967295]	
innodb_ft_min_token_size	是	3	[0-16]	
innodb_stats_auto_recalc	否	ON	[ON|OFF]	
max_connect_errors	否	100	[0-4294967295]	
session_track_schema	否	ON	[ON|OFF]	
join_buffer_size	否	262144	[128-4294967295]	
innodb_change_buffering	否	all	[none|inserts|deletes|changes|purges|all]	
optimizer_trace_max_mem_size	否	16384	[0-4294967295]	
innodb_autoinc_lock_mode	是	2	[0|1|2]	
innodb_rollback_on_timeout	是	OFF	[OFF|ON]	
loose_opt_rds_enable_show_slave_lag	否	ON	[ON|OFF]	
max_write_lock_count	否	102400	[1-102400]	
master_verify_checksum	是	OFF		
innodb_ft_num_word_optimize	否	2000	[0-10000]	

名称	是否重启	默认值	修改范围	备注
max_join_size	否	18446744073709551615	[1-18446744073709551615]	
loose_validate_password_length	否	8	[1-12]	
log_throttle_queries_not_using_indexes	否	0	[0-4294967295]	
innodb_max_purge_lag_delay	否	0	[0-10000000]	
loose_optimizer_trace	否	enabled=off,one_line=off	.*	
loose_thread_handling	是	one-thread-per-connection	[one-thread-per-connection|pool-of-threads]	
default_week_format	否	0	[0-7]	
innodb_cmp_per_index_enabled	否	OFF	[ON|OFF]	
show_compatibility_56	否	OFF	[ON|OFF]	
host_cache_size	否	644	[0-65535]	host缓存大小
low_priority_updates	否	0	[0|1]	
auto_increment_offset	否	1	[1-65535]	自增列的自增步长
range_alloc_block_size	否	4096	[4096-18446744073709551615]	
ft_min_word_len	是	4	[1-3600]	
sort_buffer_size	否	262144	[32768-4294967295]	排序申请的buffer
max_allowed_packet	否	1073741824	[16384-1073741824]	限制Server接受的数据包大小

名称	是否重启	默认值	修改范围	备注
thread_cache_size	否	256	[0-16384]	执行线程缓存
optimizer_trace_offset	否	-1	[0-4294967295]	
character_set_server	是	utf8	[utf8|latin1|gbk|gb18030|utf8mb4]	数据库级的字符集
innodb_adaptive_flushing	否	ON	[ON|OFF]	自适应刷新脏页开关
log_queries_not_using_indexes	否	OFF	[ON|OFF]	没有使用索引的查询就不会记录到慢日志去
innodb_monitor_enable	否		all	
table_open_cache_instances	是	16	[1-64]	控制 table cache instance的个数
innodb_flush_neighbors	否	1	[0|1|2]	用来控制buffer pool刷脏页时是否把脏页邻近的其他脏页一起刷到磁盘
innodb_buffer_pool_instances	是	1	[1-64]	缓冲池实例的大小
innodb_data_file_purge	否	OFF	[ON|OFF]	是否启用异步清除策略。
innodb_data_file_purge_all_at_shutdown	否	OFF	[ON|OFF]	正常关机时全部清理。
innodb_data_file_purge_immediate	否	OFF	[ON|OFF]	取消数据文件的链接但不清理。
innodb_data_file_purge_interval	否	100	[1-1073741824]	清理时间间隔。单位：ms。
innodb_data_file_purge_max_size	否	512	[1-1073741824]	每次清理单个文件大小的最大值。单位：MB。
hotspot	否	OFF	[ON|OFF]	热点更新的开关

名称	是否重启	默认值	修改范围	备注
hotspot_lock_type	否	OFF	[ON|OFF]	热点更新锁类型

8.19. 透明分布式

8.19.1. 主键拆分

PolarDB-X新增按主键类型自动拆分表功能，简易地将分布式技术引入到普通DDL语法，您只需要执行简单的修改，系统将根据主键和索引键自动选择拆分键和拆分方式，完成从单机数据库到分布式数据库的切换。

前提条件

- PolarDB-X版本为5.4.9及以上。
- 创建数据库时使用了**drds模式**，即“MODE”选择为“drds/sharding”，创建数据库时的模式选择，请参见CREATE DATABASE。

注意事项

- 主键拆分表仅支持在建表时指定主键，不支持对已有的表添加或删除主键。
- 主键拆分表的非LOCAL索引必须指定索引名。
- 拆分规则变更后，主键拆分表将变成普通表（即不再适用原主键拆分表中的自动拆分规则或索引转换规则）。如何变更拆分规则，请参见变更表类型及拆分规则。

语法

在CREATE TABLE语法中新增了PARTITION关键字，同时，在创建索引的子句中新增了LOCAL、GLOBAL和CLUSTERED关键字，以适应主键拆分表。

```
CREATE PARTITION TABLE [IF NOT EXISTS] tbl_name
                (create_definition, ...)
                [table_options]
                [drds_partition_options]
                create_definition:
                col_name column_definition
                | mysql_create_definition
                | [UNIQUE] [LOCAL | GLOBAL | CLUSTERED] INDEX index_name [index_type] (index_col_name,...)
                [global_secondary_index_option]
                [index_option] ...
```

- LOCAL：强制指定为本地索引。
- GLOBAL：全局二级索引。
- CLUSTERED：聚簇索引。

> **说明** 关于CREATE TABLE的详细语法介绍，请参见CREATE TABLE。

如果不想修改DDL，想使用普通单库单表的DDL直接创建主键拆分表，您可以通过在SQL命令行中设置用户变量的方式开启主键拆分，方法如下：

1. 执行**set @auto_partition=1;**命令，开启自动主键拆分。
2. 执行CREATE TABLE语句创建表（无需附加PARTITION关键字），该动作将被视为创建主键拆分表。
3. 执行**set @auto_partition=0;**命令，关闭自动主键拆分。

自动拆分规则

- 如果目标表没有指定主键，PolarDB-X会启用隐式主键并将其作为拆分键，该主键为BIGINT类型的自增主键，且对用户不可见。

- 如果目标表指定了主键，PolarDB-X会使用该主键作为拆分键。如果为复合主键，则选择复合主键的第一列作为拆分键。
- 自动拆分仅拆分数据库，不拆分数据表，且拆分算法根据主键类型自动选择：

主键类型	拆分算法
TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT、CHAR、VARCHAR	HASH
DATE、DATETIME、TIMESTAMP	YYYYDD
BIT、FLOAT、DOUBLE、TIME、YEAR、BLOB、ENUM、DECIMAL、BINARY、TEXT、SET、GEOMETRY	不支持拆分

索引转换规则

- 如果指定了LOCAL关键字，即强制指定索引为本地索引。
- 对主键拆分表执行创建索引操作时，如果未指定LOCAL关键字，该操作将被自动地转变为创建无覆盖列（covering）的全局二级索引，并且按索引列的第一列进行自动拆分。如果需要建立普通的局部索引，您需要指定LOCAL关键字。
- 创建全局二级索引和聚簇索引时，会创建一个带 `_local_` 前缀的本地索引。如果删除全局二级索引，PolarDB-X会自动同步删除对应的本地索引。
- 主键拆分表可以不指定全局二级索引、聚簇索引的拆分方式，PolarDB-X会根据自动拆分原则对索引键的第一列执行拆分。

下述语句及其注释为您展示了索引的转换规则。

```
CREATE PARTITION TABLE `t_order` (
  `x` int,
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  LOCAL INDEX `l_seller` using btree (`seller_id`), -- 强制指定为本地索引
  UNIQUE LOCAL INDEX `l_order` using btree (`order_id`), -- 强制指定为本地唯一索引
  INDEX `i_seller` using btree (`seller_id`), -- 会被替换为GSI，自动拆分
  UNIQUE INDEX `i_order` using btree (`order_id`), -- 会被替换为UGSI，自动拆分
  GLOBAL INDEX `g_seller` using btree (`seller_id`), -- 自动拆分
  UNIQUE GLOBAL INDEX `g_order` using btree (`order_id`), -- 自动拆分
  CLUSTERED INDEX `c_seller` using btree (`seller_id`), -- 自动拆分聚簇
  UNIQUE CLUSTERED INDEX `c_order` using btree (`order_id`) -- 自动拆分聚簇
);
```

执行**show create table t_order;**命令，查看表结构信息。

```
+----------+----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------+
| Table    | Create Table
|
+----------+----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------+
| t_order | CREATE PARTITION TABLE `t_order` (
  `x` int(11) DEFAULT NULL,
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  UNIQUE LOCAL KEY `l_order` USING BTREE (`order_id`),
  UNIQUE LOCAL KEY `_local_i_order` USING BTREE (`order_id`),
  UNIQUE LOCAL KEY `_local_g_order` USING BTREE (`order_id`),
  UNIQUE LOCAL KEY `_local_c_order` USING BTREE (`order_id`),
  LOCAL KEY `l_seller` USING BTREE (`seller_id`),
  LOCAL KEY `_local_i_seller` USING BTREE (`seller_id`),
  LOCAL KEY `_local_g_seller` USING BTREE (`seller_id`),
  LOCAL KEY `_local_c_seller` USING BTREE (`seller_id`),
  UNIQUE CLUSTERED KEY `c_order` USING BTREE (`order_id`) DBPARTITION BY HASH(`order_id`),
  CLUSTERED INDEX `c_seller` USING BTREE(`seller_id`) DBPARTITION BY HASH(`seller_id`),
  UNIQUE GLOBAL KEY `g_order` USING BTREE (`order_id`) DBPARTITION BY HASH(`order_id`),
  GLOBAL INDEX `g_seller` USING BTREE(`seller_id`) DBPARTITION BY HASH(`seller_id`),
  UNIQUE GLOBAL KEY `i_order` USING BTREE (`order_id`) DBPARTITION BY HASH(`order_id`),
  GLOBAL INDEX `i_seller` USING BTREE(`seller_id`) DBPARTITION BY HASH(`seller_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4   |
+----------+----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------+
1 row in set (0.06 sec)
```

主键拆分表的DDL限制

如需对主键拆分表执行下述DDL操作，存在一些限制。

DDL类别	DDL子句	说明与限制
CREATE INDEX	无	• 如果使用了 `[UNIQUE] LOCAL INDEX` 子句，PolarDB-X将创建本地索引，且会自动同步到聚簇索引表中。 • 如果使用了 `[UNIQUE] INDEX` 子句，PolarDB-X将其转换为对应的全局二级索引，同时，自动创建一个带 `_local_` 前缀的本地索引并同步到聚簇索引表中。 • 如果使用了 `[UNIQUE] GLOBAL | CLUSTERED INDEX` 子句，PolarDB-X会自动创建一个带 `_local_` 前缀的本地索引并同步到聚簇索引表中。 说明 如果未指定拆分方式则会自动拆分。
ALTER TABLE	ADD {INDEX | KEY} [index_name] [index_type] (key_part,...) [index_option] ...	
	ADD [COLUMN] (col_name column_definition,...)	• PolarDB-X将自动添加对应的列到聚簇全局二级索引（CGSI）和带唯一约束的聚簇全局二级索引（UCGSI）。 • 支持回滚操作。
	DROP [COLUMN] col_name	不允许删除主键、主表拆分键、索引表拆分键和复合UNIQUE约束中的列。

DDL类别	DDL子句	说明与限制
	CHANGE [COLUMN] old_col_name new_col_name column_definition [FIRST | AFTER col_name]	• 不允许重命名主键、主表拆分键、索引表拆分键。 • 不允许重命名复合UNIQUE约束中的列。 说明 其它列的限制可通过执行Hint：/*+TDDL: cmd_extra(ALLOW_LOOSE_ALTER_COLUMN_WITH_GSI=true)*/命令解除。更多信息，请参见如何使用HINT。 • 如果仅需修改default信息，推荐使用SET DEFAULT或DROP DEFAULT子句。
	MODIFY [COLUMN] col_name column_definition [FIRST | AFTER col_name]	
	ALTER TABLE tbl_name ALTER [COLUMN] col_name { SET DEFAULT {literal | (expr)} | DROP DEFAULT }	支持回滚操作。 说明 如果列的默认值为current_timestamp，则不支持回滚操作。

8.19.2. 自动分区模式下的主键拆分

本文主要介绍了在自动分区模式下主键拆分表的自动分区规则和索引变换规则。

在自动分区模式的数据库中，如不手动指定分区方式，默认按主键进行拆分。主键拆分的基本特性和限制请先参见主键拆分。

前提条件

- PolarDB-X版本为5.4.13及以上。
- 创建数据库时选择了**自动分区模式**，即“MODE”选择为“auto/partitioning”，关于创建数据库时的模式选择，请参见CREATE DATABASE。

注意事项

- 除去创建表时指定为单表或广播表，其他表都默认创建为分区表，如不手动指定分区方式，默认按主键进行拆分。
- 主键拆分表仅支持在建表时指定主键，不支持对已有的表添加或删除主键。如果建表时没有指定主键，则会自动创建隐式主键。
- 主键拆分表的索引除非第一列不支持自动分区或指定为LOCAL索引，否则自动创建为全局二级索引。
- 分区策略变更后，主键拆分表将变成普通表（即不再适用原主键拆分表中的自动分区策略或索引转换规则）。如何变更分区策略，请参见变更表类型及分区策略。
- 自动分区数据库中，GSI名称不再要求唯一，建表语句中也不再要求强制指定GSI名称。所有GSI表会携带随机后缀，使用 `show full create table` 可以看到这些随机后缀。

语法

自动分区模式下的主键拆分会默认开启，建表语句无需添加特殊关键字，建表语法参见 CREATE TABLE。

下面通过几个例子展示分区效果，其中 `show create table` 会显示表的精简结构信息，而 `show full create table` 会显示完整的表结构。

```
CREATE TABLE `tb` (
    ->   `x` int NOT NULL AUTO_INCREMENT,
    ->   `y` int NOT NULL,
    ->   `z` float NOT NULL,
    ->   `d` int NOT NULL,
    ->   PRIMARY KEY (x,y,z,d)
    -> );
Query OK, 0 rows affected (0.42 sec)
show create table tb;
+-------+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
-----------------------------------------------------------+
| TABLE | CREATE TABLE
|
```

```
+--------+----------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------+
| tb     | CREATE TABLE `tb` (
        `x` int(11) NOT NULL AUTO_INCREMENT,
        `y` int(11) NOT NULL,
        `z` float NOT NULL,
        `d` int(11) NOT NULL,
        PRIMARY KEY (`x`, `y`, `z`, `d`),
        LOCAL KEY `auto_shard_key_x_y_d` USING BTREE (`x`, `y`, `d`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 |
+--------+----------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------+
1 row in set (0.01 sec)
show full create table tb;
+--------+----------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
| TABLE | CREATE TABLE                                                                              |
+--------+----------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
| tb     | CREATE PARTITION TABLE `tb` (
        `x` int(11) NOT NULL AUTO_INCREMENT,
        `y` int(11) NOT NULL,
        `z` float NOT NULL,
        `d` int(11) NOT NULL,
        PRIMARY KEY (`x`, `y`, `z`, `d`),
        LOCAL KEY `auto_shard_key_x_y_d` USING BTREE (`x`, `y`, `d`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`x`,`y`,`d`)
PARTITIONS 16
/* tablegroup = `tg8` */ |
+--------+----------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
1 row in set (0.00 sec)
CREATE TABLE `t_order` (
    -> `x` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> `order_id` varchar(20) DEFAULT NULL,
    -> `seller_id` varchar(20) DEFAULT NULL,
    -> INDEX (`seller_id`),
    -> UNIQUE INDEX (`order_id`)
    -> );
Query OK, 0 rows affected (0.84 sec)
show create table `t_order`;
+----------+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------+
| TABLE    | CREATE TABLE
|
+----------+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------+
| t_order | CREATE TABLE `t_order` (
        `x` int(11) NOT NULL AUTO_INCREMENT,
        `order_id` varchar(20) DEFAULT NULL,
        `seller_id` varchar(20) DEFAULT NULL,
        PRIMARY KEY (`x`),
        INDEX `i_0` (`seller_id`),
```

```
        INDEX `i_0` (`seller_id`),
        UNIQUE INDEX `i_1` (`order_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 |
+----------+------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------+
1 row in set (0.01 sec)
show full create table `t_order`;
+----------+------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
| TABLE    | CREATE TABLE          |
+----------+------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
| t_order | CREATE PARTITION TABLE `t_order` (
        `x` int(11) NOT NULL AUTO_INCREMENT,
        `order_id` varchar(20) DEFAULT NULL,
        `seller_id` varchar(20) DEFAULT NULL,
        PRIMARY KEY (`x`),
        GLOBAL INDEX /* i_0_$cff4 */ `i_0` (`seller_id`) PARTITION BY KEY (`seller_id`, `x`) PARTITIONS 16,
        UNIQUE GLOBAL INDEX /* i_1_$1782 */ `i_1` (`order_id`) PARTITION BY KEY (`order_id`) PARTITIONS 16,
        UNIQUE LOCAL KEY `_local_i_1` (`order_id`),
        LOCAL KEY `_local_i_0` (`seller_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`x`)
PARTITIONS 16
/* tablegroup = `tg8` */ |
+----------+------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------+
1 row in set (0.00 sec)
```

自动分区规则

- 如果目标表没有指定主键，PolarDB-X会启用隐式主键并将其作为拆分键，该主键为BIGINT类型的自增主键，且对用户不可见，使用`show full create table`可以查看完整的分区细节。
- 如果目标表指定了主键，PolarDB-X会使用该主键作为拆分键。如果为复合主键，会使用复合主键的所有列作为拆分键。
- 对于复合主键，按顺序作为KEY拆分方式的参数，如果遇到不支持的类型则跳过该列，如果第一列就不支持则会报错，必须手动指定为SINGLE表才能创建成功，SINGLE表建表语句参见单表与广播表。
- 自动分区使用分区表拆分算法，且拆分算法根据主键类型自动选择：

主键类型	拆分算法
bit, float, double, time, year, tinyblob, blob, mediumblob, longblob, enum, decimal, binary, varbinary, tinytext, text, mediumtext, longtext, set, geometry	不支持
其他	KEY分区

索引转换规则

主键拆分表上创建索引，除非明确指定为LOCAL索引，否则会自动转换为全局二级索引，具体转换规则如下：

- 如果指定了LOCAL关键字，即强制指定索引为本地索引。
- 对主键拆分表执行创建索引操作时，如果未指定LOCAL关键字，该操作将被自动地转变为创建无覆盖列（covering）的全局二级索引，并且索引列会依据上述的自动分区规则进行自动分区。如果需要建立普通的局部索引，您需要指定LOCAL关键字。
- 在对全局二级索引生成拆分算法时，会使用全部的索引键进行拆分，同时，对于非UNIQUE约束，会附带上主键作为拆分算法参数。
- 创建全局二级索引和聚簇索引时，会创建一个带 _local_ 前缀的本地索引。如果删除全局二级索引，PolarDB-X会自动同步删除对应的本地索引。
- 主键拆分表可以不指定全局二级索引、聚簇索引的分区方式，PolarDB-X会根据自动分区原则对索引键执行分区。

下述语句及其注释为您展示了索引的转换规则。

```
CREATE PARTITION TABLE `t_order` (
    ->   `x` int,
    ->   `order_id` varchar(20) DEFAULT NULL,
    ->   `seller_id` varchar(20) DEFAULT NULL,
    ->   LOCAL INDEX `l_seller` using btree (`seller_id`), -- 强制指定为本地索引
    ->   UNIQUE LOCAL INDEX `l_order` using btree (`order_id`), -- 强制指定为本地唯一索引
    ->   INDEX `i_seller` using btree (`seller_id`), -- 会被替换为GSI，自动分区
    ->   UNIQUE INDEX `i_order` using btree (`order_id`), -- 会被替换为UGSI，自动分区
    ->   GLOBAL INDEX `g_seller` using btree (`seller_id`), -- 自动分区
    ->   UNIQUE GLOBAL INDEX `g_order` using btree (`order_id`), -- 自动分区
    ->   CLUSTERED INDEX `c_seller` using btree (`seller_id`), -- 自动分区聚簇
    ->   UNIQUE CLUSTERED INDEX `c_order` using btree (`order_id`) -- 自动分区聚簇
    -> );
Query OK, 0 rows affected (1.49 sec)
show create table `t_order`;
+---------+----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
----------------------------------------------------+
| TABLE   | CREATE TABLE                                     |
+---------+----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
----------------------------------------------------+
| t_order | CREATE TABLE `t_order` (
        `x` int(11) DEFAULT NULL,
        `order_id` varchar(20) DEFAULT NULL,
        `seller_id` varchar(20) DEFAULT NULL,
        UNIQUE CLUSTERED INDEX `c_order` USING BTREE (`order_id`),
        CLUSTERED INDEX `c_seller` USING BTREE (`seller_id`),
        UNIQUE INDEX `g_order` USING BTREE (`order_id`),
        INDEX `g_seller` USING BTREE (`seller_id`),
        UNIQUE INDEX `i_order` USING BTREE (`order_id`),
        INDEX `i_seller` USING BTREE (`seller_id`),
        UNIQUE LOCAL KEY `l_order` USING BTREE (`order_id`),
        LOCAL KEY `l_seller` USING BTREE (`seller_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 |
+---------+----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
----------------------------------------------------+
1 row in set (0.01 sec)
show full create table `t_order`;
+---------+----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| TABLE   | CREATE TABLE
|
+---------+---------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
| t_order | CREATE PARTITION TABLE `t_order` (
        `x` int(11) DEFAULT NULL,
        `order_id` varchar(20) DEFAULT NULL,
        `seller_id` varchar(20) DEFAULT NULL,
        `_drds_implicit_id_` bigint(20) NOT NULL AUTO_INCREMENT,
        PRIMARY KEY (`_drds_implicit_id_`),
        UNIQUE CLUSTERED INDEX /* c_order_$1ba0 */ `c_order` USING BTREE (`order_id`) PARTITION BY KEY (`ord
er_id`) PARTITIONS 16,
        CLUSTERED INDEX /* c_seller_$1e39 */ `c_seller` USING BTREE (`seller_id`) PARTITION BY KEY (`seller_
id`, `_drds_implicit_id_`) PARTITIONS 16,
        UNIQUE GLOBAL INDEX /* g_order_$d57f */ `g_order` USING BTREE (`order_id`) PARTITION BY KEY (`order_
id`) PARTITIONS 16,
        GLOBAL INDEX /* g_seller_$6ed5 */ `g_seller` USING BTREE (`seller_id`) PARTITION BY KEY (`seller_id`
, `_drds_implicit_id_`) PARTITIONS 16,
        UNIQUE GLOBAL INDEX /* i_order_$ab2f */ `i_order` USING BTREE (`order_id`) PARTITION BY KEY (`order_
id`) PARTITIONS 16,
        GLOBAL INDEX /* i_seller_$2b4b */ `i_seller` USING BTREE (`seller_id`) PARTITION BY KEY (`seller_id`
, `_drds_implicit_id_`) PARTITIONS 16,
        UNIQUE LOCAL KEY `l_order` USING BTREE (`order_id`),
        UNIQUE LOCAL KEY `_local_i_order` USING BTREE (`order_id`),
        UNIQUE LOCAL KEY `_local_g_order` USING BTREE (`order_id`),
        UNIQUE LOCAL KEY `_local_c_order` USING BTREE (`order_id`),
        LOCAL KEY `l_seller` USING BTREE (`seller_id`),
        LOCAL KEY `_local_i_seller` USING BTREE (`seller_id`),
        LOCAL KEY `_local_g_seller` USING BTREE (`seller_id`),
        LOCAL KEY `_local_c_seller` USING BTREE (`seller_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`_drds_implicit_id_`)
PARTITIONS 16
/* tablegroup = `tg11` */ |
+---------+---------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

主键拆分表的DDL约束

同普通库的主键拆分表约束相同，请参考主键拆分表的DDL限制。

8.19.3. 聚簇索引

PolarDB-X新增支持聚簇索引功能，用于自动维护全局二级索引（GSI）中的覆盖列，保证聚簇索引表和主表的实时同步，所有查询均不用回表，避免因回表带来的额外开销。本文介绍如何创建并使用聚簇索引。

前提条件

PolarDB-X内核小版本需为5.4.9或以上。

注意事项

- 聚簇索引是一种特殊的全局二级索引，相关行为和限制请参考全局二级索引。
- 聚簇索引的覆盖列默认包含主表的所有列，并在主表的列发生变更时，自动同步修改聚簇索引表，保证聚簇索引表和主表的实时同步。
- 聚簇索引表也会和主表的本地索引保持同步。

语法

您可以在建表或加索引的语句中，通过 `CLUSTERED` 关键字指定创建的索引为聚簇索引。

- CREATE TABLE:

```
CREATE [SHADOW] TABLE [IF NOT EXISTS] tbl_name
    (create_definition, ...)
    [table_options]
    [drds_partition_options]
create_definition:
    [UNIQUE] CLUSTERED INDEX index_name [index_type] (index_col_name,...)
      [drds_partition_options]
      [index_option] ...
```

> **说明** 仅在主键拆分表中可省略拆分规则即 `[drds_partition_options]` 部分。

- CREATE INDEX:

```
CREATE [UNIQUE]
    CLUSTERED INDEX index_name [index_type]
    ON tbl_name (index_col_name,...)
    [drds_partition_options]
    [index_option] ...
```

> **说明** 仅在主键拆分表中可省略拆分规则即 `[drds_partition_options]` 部分。

- ALTER TABLE:

```
ALTER TABLE tbl_name
    alter_specification
```

其中 `alter_specification` 支持如下规则：

```
alter_specification:
  | ADD [UNIQUE] CLUSTERED {INDEX|KEY} index_name
      [index_type] (index_col_name,...)
      [drds_partition_options]
      [index_option] ...
```

> **说明**
> - 聚簇索引相关变更（即 `alter_specification` 部分）仅支持使用一条变更规则。
> - 聚簇索引必须显式指定索引名。
> - 仅在主键拆分表中可省略拆分规则（即 `[drds_partition_options]` 部分）。

使用示例

假设已使用如下语句在PolarDB-X数据库中创建了一张 `t_order` 表：

```
CREATE PARTITION TABLE `t_order` (
    ->   `t` timestamp null default CURRENT_TIMESTAMP,
    ->   `x` int default 3,
    ->   `order_id` varchar(20) DEFAULT NULL,
    ->   `seller_id` varchar(20) DEFAULT NULL
    -> );
```

您可以使用如下语句为 `t_order` 表添加聚簇索引：

```
CREATE CLUSTERED INDEX `c_i` ON `t_order` (seller_id, x)
```

添加成功后，您可以使用如下语句查看主表结构，来确认聚簇索引的定义：

```
SHOW CREATE TABLE t_order;
```

返回结果如下：

```
+---------+-------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
-------------------------------------+
| Table   | Create Table
|
+---------+-------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
-------------------------------------+
| t_order | CREATE PARTITION TABLE `t_order` (
  `t` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `x` int(11) DEFAULT '3',
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  LOCAL KEY `_local_c_i` (`seller_id`, `x`),
  CLUSTERED INDEX `c_i`(`seller_id`, `x`) DBPARTITION BY HASH(`seller_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4   |
+---------+-------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
-------------------------------------+
1 row in set (0.08 sec)
```

您还可以通过如下语句查看聚簇索引表结构：

```
SHOW CREATE TABLE c_i;
```

从如下返回结果中，可以看到聚簇索引表包含了主表所有的列：

```
+-------+---------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
---------------------------------------------+
| Table | Create Table
|
+-------+---------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
---------------------------------------------+
| c_i   | CREATE TABLE `c_i` (
  `t` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `x` int(11) DEFAULT '3',
  `order_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  KEY `auto_shard_key_seller_id` USING BTREE (`seller_id`),
  KEY `i_seller_id_x` USING BTREE (`seller_id`, `x`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4  dbpartition by hash(`seller_id`) |
+-------+---------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
---------------------------------------------+
1 row in set (0.03 sec)
```

8.19.4. 变更表类型及拆分规则

PolarDB-X新增支持变更表的类型（即在单表、拆分表和广播表三者间进行相互转换），和变更拆分表的拆分规则（包括拆分函数或拆分列）。本文介绍相关语法和示例。

前提条件

- 仅内核小版本为5.4.8或以上的PolarDB-X实例支持变更拆分表的拆分规则。
- 仅内核小版本为5.4.10或以上的PolarDB-X实例支持变更表的类型（即在单表、拆分表和广播表三者间进行相互转换）。

如何查看实例版本，请参见查看实例版本。

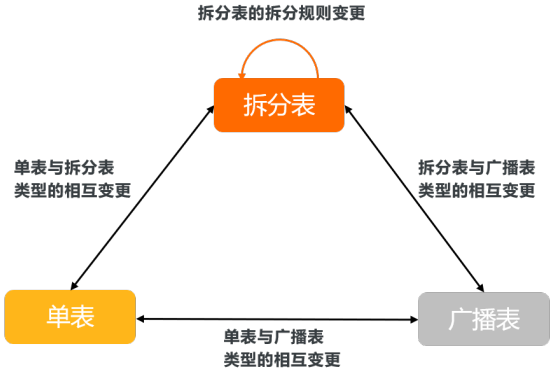
注意事项

- 暂不支持变更带有GSI的拆分表的拆分规则。
- 表属性变更后，主键拆分表将变成普通表（即不再适用原主键拆分表中的自动拆分规则或索引转换规则）。更多详情，请参见主键拆分。
- 若单表设置了自增列，在变更为广播表或拆分表时，需提前为该表创建Sequence。如何创建Sequence，请参见创建Sequence。
- 本文中关于变更拆分表、广播表和单表的表类型示例，均在单表 `t_order` 的基础上进行变更， `t_order` 表的创建语句如下：

```
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT BY GROUP,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

表类型

PolarDB-X实例支持3种类型的表：拆分表、广播表和单表。您可以通过ALTER TABLE语句在拆分表、广播表和单表间进行转换，同时还能对拆分表的拆分规则进行变更。



- **拆分表**

  使用 `drds_partition_options` 子句进行拆分的表。

  `drds_partition_options` 可以是如下分库或分表子句：

```
drds_partition_options:
    DBPARTITION BY db_partition_algorithm
    [TBPARTITION BY table_partition_algorithm [TBPARTITIONS number]]
```

  其中：

- `db_partition_algorithm` 支持如下函数：

```
db_partition_algorithm:
    HASH([col_name])
  | {YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
```

- `table_partition_algorithm` 支持如下函数：

```
table_partition_algorithm:
    HASH(col_name)
  | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(col_name)
  | UNI_HASH(col_name)
  | RIGHT_SHIFT(col_name, n)
  | RANGE_HASH(col_name, col_name, n)
```

> **说明** 更多关于拆分函数的信息，请参见拆分函数。

- **广播表**

  通过 `BROADCAST` 子句创建的表，系统会将该表复制到每个分库上，并通过分布式事务实现数据一致性。

- **单表**

  未进行任何拆分或未指定 `BROADCAST` 子句的表。

单表或广播表变为拆分表

- **语法**

```
ALTER TABLE table_name drds_partition_options;
```

> **说明**
> - 更多关于 `drds_partition_options` 的信息，请参见表类型。
> - 若单表设置了自增列，在变更为拆分表时，需提前为该表创建Sequence。如何创建Sequence，请参见创建Sequence。

- **示例**

  因业务扩展，单表 `t_order` 无法承载日益增长的数据。此时，您可以使用如下语句将该单表变更为拆分表（以 `order_id` 为拆分键）：

```
ALTER TABLE t_order dbpartition BY hash(`order_id`);
```

单表或拆分表变为广播表

- **语法**

```
ALTER TABLE table_name BROADCAST;
```

> **说明** 若单表设置了自增列，在变更为广播表时，需提前为该表创建Sequence。如何创建Sequence，请参见创建Sequence。

- **示例**

  您可以使用如下语句将单表或拆分表 `t_order` 变更为广播表：

```
ALTER TABLE t_order BROADCAST;
```

广播表或拆分表变为单表

- **语法**

```
ALTER TABLE table_name SINGLE;
```

- **示例**

  您可以使用如下语句将广播表或拆分表 `t_order` 变更为单表：

```
ALTER TABLE t_order SINGLE;
```

变更拆分表的拆分规则

- **语法**

```
ALTER TABLE tbl_name drds_partition_options;
```

- **示例**

  假设已使用如下语句在PolarDB-X数据库中创建了一张拆分表 `t_order` （根据 `order_id` 列进行库级拆分）：

```
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition BY hash(`order_id`);
```

  现需要对 `t_order` 表的拆分规则作出如下变更：

  - 根据 `order_id` 列进行库级拆分。
  - `buyer_id` 列做表级拆分。
  - 每个分库包含3个分表。

  您可以使用如下语句实现上述变更：

```
ALTER TABLE t_order dbpartition BY hash(order_id) tbpartition BY hash(buyer_id) tbpartitions 3;
```

相关文档

拆分规则变更后，您可以通过如下命令查看表的拆分规则或拓扑结构：

- 查看表拆分规则，请参见规则和拓扑查询语句。
- 查看表拓扑结构，请参见规则和拓扑查询语句。

常见问题

Q：为什么有时拆分键变更的DDL任务会执行失败？此时该如何处理？

A：实例崩溃或唯一索引存在冲突等因素会导致拆分规则变更的DDL任务执行失败。但这不会损坏原表任何数据，也不会阻塞正常的DML和查询语句执行。当拆分键变更的DDL任务执行失败时，您可以通过 `ROLLBACK DDL` 命令回滚该任务，然后再次尝试变更。关于 `ROLLBACK DDL` 命令的详情。

> **说明** 暂不支持通过 `RECOVER DDL` 命令恢复执行失败的拆分键变更任务。

8.19.5. 变更表类型及分区策略

本文介绍了变更表类型及分区策略（包括拆分函数或分区列）的相关语法和示例。

前提条件

- 仅适用于分区模式为partitioning的逻辑库（请参见CREATE DATABASE）；
- 仅支持对内核小版本为5.4.13或以上的PolarDB-X实例变更表的类型和分区策略。
- 仅内核小版本为5.4.13或以上的PolarDB-X实例支持对带有GSI的分区表进行分区变更。

如何查看实例版本，请参见查看实例版本。

注意事项

- 表属性变更后，主键分区表将变成普通表（即不再适用原主键分区表中的自动分区策略或索引转换规则）。更多详情，请参见自动分区模式下的主键拆分。
- 本文中关于变更分区表、广播表和单表的表类型示例，均在单表 `t_order` 的基础上进行变更， `t_order` 表的创建语句如下：

```
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT BY GROUP,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

表类型

PolarDB-X实例支持3种类型的表：分区表、广播表和单表。您可以通过ALTER TABLE语句将表的类型在分区表、广播表和单表之间进行转换，同时还能对分区表的分区策略进行变更。

- **分区表**

  使用 `partition_options` 分区子句进行创建的表。

  `partition_options` 可以是如下子句：

```
  partition_options:
    PARTITION BY
          HASH({column_name | partition_func(column_name)})
        | KEY(column_list)
        | RANGE{({column_name | partition_func(column_name)})
        | RANGE COLUMNS(column_list)}
        | LIST{({column_name | partition_func(column_name)})
        | LIST COLUMNS(column_list)} }
    partition_list_spec
```

  > **说明** 更多关于分区策略的信息，请参见分区类型介绍。

- **广播表**

  通过 `BROADCAST` 子句创建的表，系统会将该表复制到每个分库上，并通过分布式事务实现数据一致性，更多详情，请参见单表与广播表。

- **单表**

  通过 `SINGLE` 子句创建的表。更多详情，请参见单表与广播表。

单表或广播表变为分区表

- **语法**

```
ALTER TABLE table_name drds_partition_options;
```

> 说明
>
> 更多关于 `partition_options` 的信息，请参见CREATE TABLE。

- **示例**

  因业务扩展，单表 `t_order` 无法承载日益增长的数据。此时，您可以使用如下语句将该单表变更为分区表（以 `order_id` 为分区键，采用KEY分区策略）：

  ```
  ALTER TABLE t_order PARTITION BY KEY(`order_id`);
  ```

  如需指定分区数量，可以使用如下语句：

  ```
  ALTER TABLE t_order PARTITION BY KEY(`order_id`) PARTITIONS 8;
  ```

单表或分区表变为广播表

- **语法**

  ```
  ALTER TABLE table_name BROADCAST;
  ```

- **示例**

  您可以使用如下语句将单表或分区表 `t_order` 变更为广播表：

  ```
  ALTER TABLE t_order BROADCAST;
  ```

广播表或分区表变为单表

- **语法**

  ```
  ALTER TABLE table_name SINGLE;
  ```

  或者：

  ```
  ALTER TABLE table_name REMOVE PARTITIONING;
  ```

- **示例**

  您可以使用如下语句将广播表或分区表 `t_order` 变更为单表：

  ```
  ALTER TABLE t_order SINGLE;
  ```

  或者：

  ```
  ALTER TABLE t_order REMOVE PARTITIONING;
  ```

变更分区表的分区策略

- **语法**

  ```
  ALTER TABLE tbl_name partition_options;
  ```

- **示例1**

  假设已使用如下语句在PolarDB-X数据库中创建了一张分区表 `t_order` （根据 `order_id` 列进行KEY分区）：

```
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
PARTITION BY KEY(`order_id`);
```

现需要对 `t_order` 表的分区策略作出如下变更：

- 根据 `order_id` 列以及 `buyer_id` 进行KEY分区。
- 共包含8个分区。

您可以使用如下语句实现上述变更：

```
ALTER TABLE t_order PARTITION BY KEY(order_id, buyer_id) PARTITIONS 8;
```

- **示例2**

  假设已使用如下语句在PolarDB-X数据库中创建了一张分区表 `t_order` （根据 `id` 列进行RANGE分区）：

```
CREATE TABLE t_order (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `l_i_order` (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
PARTITION BY RANGE(`id`)
(
  PARTITION p1 VALUES LESS THAN (100),
  PARTITION p2 VALUES LESS THAN (1000),
  PARTITION P3 VALUES LESS THAN MAXVALUE
);
```

  现需要对 `t_order` 表的分区策略作出如下变更：

  - 根据 `order_id` 列以及 `buyer_id` 进行KEY分区；
  - 总共包含16个分区。

  您可以使用如下语句实现上述变更：

```
ALTER TABLE t_order PARTITION BY KEY(order_id, buyer_id) PARTITIONS 16;
```

相关文档

分区变更后，您可以通过如下命令查看表的分区规则或拓扑结构：

- 查看表分区规则，请使用 `SHOW CREATE TABLE tablename` 。
- 查看表拓扑结构，请参见SHOW TOPOLOGY FROM tablename。

常见问题

Q：为什么有时分区变更的DDL任务会执行失败？此时该如何处理？

A：实例崩溃或唯一索引存在冲突等因素会导致分区变更的DDL任务执行失败。但这不会损坏原表任何数据，也不会阻塞正常的DML和查询语句执行。当分区变更的DDL任务执行失败时，您可以通过 `CANCEL DDL` 命令回滚该任务，然后再次尝试变更。关于 `CANCEL DDL` 命令的详情，请参见CANCEL DDL。

8.20. 常见问题

8.20.1. DDL常见问题

本文汇总了PolarDB-X上常见的DDL执行问题。

创建表执行出错

DDL的执行是一个分布式处理过程，出错可能导致各个分片表结构不一致，所以需要进行手动清理，详细操作步骤如下：

1. PolarDB-X会提供基本的错误描述信息，比如语法错误等。如果错误信息太长，则会提示您使用 `SHOW WARNINGS` 来查看每个分库执行失败的原因。
2. 执行 `SHOW TOPOLOGY` 命令来查看物理表的拓扑结构。

```
SHOW TOPOLOGY FROM multi_db_multi_tbl;
+------+-----------------+-----------------------+
| ID   | GROUP_NAME      | TABLE_NAME            |
+------+-----------------+-----------------------+
|    0 | corona_qatest_0 | multi_db_multi_tbl_00 |
|    1 | corona_qatest_0 | multi_db_multi_tbl_01 |
|    2 | corona_qatest_0 | multi_db_multi_tbl_02 |
|    3 | corona_qatest_1 | multi_db_multi_tbl_03 |
|    4 | corona_qatest_1 | multi_db_multi_tbl_04 |
|    5 | corona_qatest_1 | multi_db_multi_tbl_05 |
|    6 | corona_qatest_2 | multi_db_multi_tbl_06 |
|    7 | corona_qatest_2 | multi_db_multi_tbl_07 |
|    8 | corona_qatest_2 | multi_db_multi_tbl_08 |
|    9 | corona_qatest_3 | multi_db_multi_tbl_09 |
|   10 | corona_qatest_3 | multi_db_multi_tbl_10 |
|   11 | corona_qatest_3 | multi_db_multi_tbl_11 |
+------+-----------------+-----------------------+
12 rows in set (0.21 sec)
```

3. 执行 `CHECK TABLE tablename` 语句来查看逻辑表是否创建成功。例如下面的例子展示了multi_db_multi_tbl的某个物理分表没有创建成功时的情况。

```
mysql> check table multi_db_multi_tbl;
+-------------------------------------------------+-------+----------+----------------------------------
-----------------------------------------+
| TABLE                                           | OP    | MSG_TYPE | MSG_TEXT
|
+-------------------------------------------------+-------+----------+----------------------------------
-----------------------------------------+
| andor_mysql_qatest. multi_db_multi_tbl | check | Error    | Table 'corona_qatest_0. multi_db_multi_tbl
_02' doesn't exist |
+-------------------------------------------------+-------+----------+----------------------------------
-----------------------------------------+
1 row in set (0.16 sec)
```

4. 使用幂等的方式重新执行建表或删表操作，该操作会创建或删除剩余的物理表。

```
CREATE TABLE IF NOT EXISTS table1
(id int, name varchar(30), primary key(id))
dbpartition by hash(id);
DROP TABLE IF EXISTS table1;
```

建索引失败或加列失败

建索引失败或加列失败的处理方法跟上面建表失败的处理类似，详情请参见如何正确处理DDL异常。

8.20.2. 如何使用全局二级索引

PolarDB-X支持全局二级索引，本文将在分库分表语法下介绍如何创建、使用全局二级索引功能。

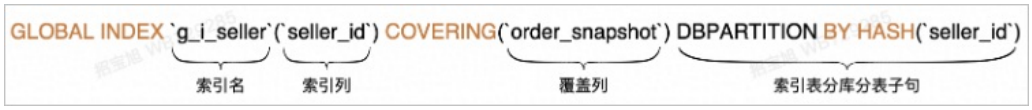
使用限制

如果在分区表下，该文档依然试用，只不过创建语法需要参考CREATE INDEX。

创建GSI

PolarDB-X对MySQL DDL语法进行了扩展，增加定义GSI的语法。使用方式与在MySQL上创建索引一致。

- 建表时定义GSI



- 建表后添加GSI

> **说明**
> - 索引名：作为索引表的名字，用于创建索引表。
> - 索引列：索引表的分库分表键，即索引分库分表子句中用到的所有列。
> - 覆盖列：索引表中的其他列，默认包含主键和主表的全部分库分表键。
> - 索引分库分表子句：索引表的分库分表算法，与CREATE TABLE中分库分表子句的语法一致。
> - 上述是在分库分表下的创建GSI语法，如果是在分区表下GSI语法只需要可以参考CREATE INDEX。

示例：

```
# 建表时定义GSI
CREATE TABLE t_order (
 `id` bigint(11) NOT NULL AUTO_INCREMENT,
 `order_id` varchar(20) DEFAULT NULL,
 `buyer_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 `order_snapshot` longtext DEFAULT NULL,
 `order_detail` longtext DEFAULT NULL,
 PRIMARY KEY (`id`),
 GLOBAL INDEX `g_i_seller`(`seller_id`) COVERING (`id`, `order_id`, `buyer_id`, `order_snapshot`) dbpartitio
n by hash(`seller_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
# 添加GSI
CREATE UNIQUE GLOBAL INDEX `g_i_buyer` ON `t_order`(`buyer_id`)
    COVERING(`seller_id`, `order_snapshot`)
    dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_id`) tbpartitions 3
```

使用GSI

GSI创建完成后，可以通过如下方式指定查询使用的索引表：

- 通过HINT指定索引

  您可以选择以下两种HINT语句中的任意一种指定使用目标索引进行查询。

```
FORCE INDEX({index_name})
```

示例：

```
SELECT a.*, b.order_id
 FROM t_seller a
   JOIN t_order b FORCE INDEX(g_i_seller) ON a.seller_id = b.seller_id
 WHERE a.seller_nick="abc";
```

```
/*+TDDL:INDEX({table_name/table_alias}, {index_name})*/
```

示例：

```
/*+TDDL:index(a, g_i_buyer)*/ SELECT * FROM t_order a WHERE a.buyer_id = 123
```

> **说明** 如果查询需要使用索引中未包含的列，则首先查询索引表取得所有记录的主键和主表分库分表键，然后回查主表中取得缺少列的值，详细说明请参见。

- 直接查询索引表

  如果索引表中包含了查询需要的所有列，可以直接查询索引表获得结果。

- 索引选择

  对于带有全局二级索引的主表查询，PolarDB-X会自动选择出优化器认为代价最低的索引表（目前只支持覆盖索引选择）。下面SQL查询的主表是t_order，带有seller_id等值过滤条件，同时涉及的id、order_snapshot和seller_id等列被全局二级索引g_i_seller覆盖。选择了覆盖索引g_i_seller既可以不回表，又可以明确减少分表的扫描数目（seller_id是g_i_seller的拆分键）。通过EXPLAIN可以看到PolarDB-X优化器确实选择了g_i_seller。

```
EXPLAIN SELECT t_order.id,t_order.order_snapshot FROM t_order WHERE t_order.seller_id = 's1';
IndexScan(tables="g_i_seller_sfL1_2", sql="SELECT `id`, `order_snapshot` FROM `g_i_seller` AS `g_i_seller`
WHERE (`seller_id` = ?)")
```

- IGNORE INDEX与USE INDEX

  您可以通过以下HINT指定优化器使用或不使用某些索引。

```
IGNORE INDEX({index_name},...)
```

示例：

```
SELECT t_order.id,t_order.order_snapshot FROM t_order IGNORE INDEX(g_i_seller) WHERE t_order.seller_id =
's1';
```

```
USE INDEX({index_name},...)
```

示例：

```
SELECT t_order.id,t_order.order_snapshot FROM t_order USE INDEX(g_i_seller) WHERE t_order.seller_id = 's
1';
```

注意事项

- 创建GSI过程的约束
  - 不支持在单表或广播表上创建GSI。
  - 不支持在无主键的表上创建GSI。
  - 不支持在UNIQUE GSI中通过任何方式使用前缀索引。
  - 创建索引表时必须指定索引名。
  - 创建索引表时必须指定分库或分库加分表组合的规则，不允许仅指定分表规则或不指定任何拆分规则。
  - 索引表的INDEX列必须包含全部拆分键。
  - GSI定义子句中，索引列与覆盖列不可重复。
  - 索引表默认包含主表的全部主键和拆分键，如果没有显式包含在索引列中，默认添加到覆盖列。

- 对主表中的每个局部索引，如果引用的所有列均包含在索引表中，默认添加该局部索引到索引表。
- 对GSI的每个索引列，如果没有已经存在的索引，默认单独创建一个索引。
- 对包含多个索引列的GSI，默认创建一个联合局部索引，包含所有索引列。
- 索引定义中，索引列的length参数仅用于在索引表拆分键上创建局部索引。
- 建表后创建GSI时，会在GSI创建结束时自动进行数据校验，只有通过校验，创建GSI的DDL语句才能执行成功。

> **说明** 您也可以使用CHECK GLOBAL INDEX对索引数据进行校验或订正。

- Alter Table过程的约束

语句	是否支持变更主表拆分键	是否支持变更主表主键（也即索引表主键）	是否支持变更本地唯一索引列	是否支持变更索引表拆分键	是否支持变更Unique Index列	是否支持变更Index列	是否支持变更Covering列
ADD COLUMN	无该场景	不支持	无该场景	无该场景	无该场景	无该场景	无该场景
ALTER COLUMN SET DEFAULT和ALTER COLUMN DROP DEFAULT	不支持	不支持	支持	不支持	不支持	不支持	不支持
CHANGE COLUMN	不支持	不支持	支持	不支持	不支持	不支持	不支持
DROP COLUMN	不支持	不支持	仅当唯一键中只有1列时支持	不支持	不支持	不支持	不支持
MODIFY COLUMN	不支持	不支持	支持	不支持	不支持	不支持	不支持

> **说明**
> - 考虑到全局二级索引的稳定性和性能情况，目前禁止直接使用DROP COLUMN命令删除全局二级索引中的列。如需删除全局二级索引中的某些列，您可以先使用DROP INDEX删除对应的全局二级索引，再重新创建一个新的二级索引，或提交工单联系技术支持进行删除。
> - 以上对列的分类存在重叠（如Index列包含索引表拆分键，Covering列包含主表拆分键、主键以及指定的列），若存在支持情况冲突情况，不支持的优先级高于支持。

下表汇总了使用ALTER TABLE语句变更索引的支持情况

语句	是否支持
ALTER TABLE ADD PRIMARY KEY	支持
ALTER TABLE ADD [UNIQUE/FULLTEXT/SPATIAL/FOREIGN] KEY	支持，您可以同时在主表和索引表上添加局部索引，索引名称不可与GSI重复。
ALTER TABLE ALTER INDEX index_name {VISIBLE | INVISIBLE}	支持，仅在主表执行（禁止变更GSI状态）。
ALTER TABLE {DISABLE | ENABLE} KEYS	支持，仅在主表执行（禁止变更GSI状态）。
ALTER TABLE DROP PRIMARY KEY	禁止
ALTER TABLE DROP INDEX	仅支持删除普通索引或全局二级索引。

语句	是否支持
ALTER TABLE DROP FOREIGN KEY fk_symbol	支持，仅在主表执行。
ALTER TABLE RENAME INDEX	禁止

> **说明** 考虑到全局二级索引的稳定性和性能情况，目前禁止直接使用DROP COLUMN命令重命名全局二级索引。如需修改全局二级索引名，您可以先使用DROP INDEX删除对应的全局二级索引，再重新创建一个新的二级索引，或提交工单联系技术支持进行修改。

- Alter GSI Table的约束条件
  - 不支持在索引表上执行DDL、DML语句。
  - 不支持带有NODE HINT的DML语句更新主表、索引表。
- 在包含GSI的表上使用其他DDL时的约束

语句	是否支持
DROP TABLE	支持
DROP INDEX	支持
TRUNCATE TABLE	不支持
RENAME TABLE	不支持
ALTER TABLE RENAME	不支持

> **说明**
>
> - 考虑主表与索引表的数据一致性，目前禁止执行TRUNCATE TABLE语句 。如需清空主表与索引表数据，您可以使用DELETE语句删除对应的数据，或强制使用HINT `/*+TDDL:CMD_EXTRA(TRUNCATE_TABLE_WITH_GSI=TRUE)*/` 。
> - 考虑到全局二级索引的稳定性和性能情况，目前禁止直接使用RENAME TABLE或ALTER TABLE RENAME命令重命名全局二级索引。如需修改全局二级索引名，您可以先使用DROP INDEX删除全局二级索引，修改表名后再重新创建新的二级索引，或提交工单联系技术支持进行修改。

- 在包含GSI的表上使用DML语句时的约束
  - 不支持在索引表上执行DML语句。
  - 在主表上执行DML语句的有如下限制：写索引失败后，不允许继续执行其他语句或提交事务。

```
CREATE TABLE t_order(
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `order_id` varchar(20) DEFAULT NULL,
  `buyer_id` varchar(20) DEFAULT NULL,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext DEFAULT NULL,
  `order_detail` longtext DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `l_i_order` (`order_id`),
  GLOBAL INDEX `g_i_seller` (`seller_id`) dbpartition by hash(`seller_id`) tbpartition by hash(`seller_id`
),
  GLOBAL UNIQUE INDEX `g_i_buyer` (`buyer_id`) COVERING (order_snapshot) dbpartition by hash(`buyer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
SET DRDS_TRANSACTION_POLICY='XA';
INSERT INTO t_order(order_id, buyer_id, seller_id) VALUES('order_1', 'buyer_1', 'seller_1');
# 失败
INSERT IGNORE INTO t_order(order_id, buyer_id, seller_id) VALUES('order_2', 'buyer_1', 'seller_1');
# 失败不允许继续执行
INSERT IGNORE INTO t_order(order_id, buyer_id, seller_id) VALUES('order_2', 'buyer_2', 'seller_2');
# 失败后不允许提交事务
COMMIT;
```

8.20.3. 如何使用HINT

本文介绍了HINT的语法级示例。

HINT作为一种SQL补充语法，在关系型数据库中扮演着非常重要的角色。它允许用户通过相关的语法影响SQL的执行方式，对SQL进行特殊的优化。同样，PolarDB-X也提供了特殊的HINT语法。

语法

```
/*+TDDL: hint_command [hint_command ...]*/
/!+TDDL: hint_command [hint_command ...]*/
```

> **说明** 如果使用 /*+TDDL:hint_command*/ 格式，在使用MySQL官方命令行客户端执行带有PolarDB-X自定义HINT的SQL时，请在登录命令中加上-c参数。否则，由于PolarDB-X自定义HINT是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致PolarDB-X自定义HINT失效。详情请参见MySQL 官方客户端命令。

示例

```
# 查询每个分库中的物理表名
/*+TDDL:scan()*/SHOW TABLES;
# 将查询下发到RDS只读实例的0000分库上
/*+TDDL:node(0) slave()*/SELECT * FROM t1;
# 强制指定workload为AP
/*+TDDL:WORKLOAD=AP*/SELECT * FROM t1;
```

PolarDB-X支持在HINT语句中使用多个HINT命令：

```
SELECT /*+TDDL:node(0) slave()*/ ...;
```

PolarDB-X不支持通过以下方式使用多个HINT命令：

```
# 不支持单条SQL语句中包含多个HINT语句
SELECT /*+TDDL:node(0)*/ /*+TDDL:slave()*/ ...;
# 不支持HINT语句中包含重复的HINT命令
SELECT /*+TDDL:node(0) node(1)*/ ...;
```

8.20.4. INDEX HINT

PolarDB-X支持全局二级索引（Global Secondary Index，简称GSI），您可以通过INDEX HINT命令指定从GSI中获取查询结果。

使用限制

INDEX HINT仅对SELECT语句生效。

注意事项

PolarDB-X自定义HINT支持 `/*+TDDL:hint_command*/` 和 `/!+TDDL:hint_command*/` 两种格式。若使用 `/*+TDDL:hint_command*/` 格式时，在使用MySQL 官方命令行客户端执行带有PolarDB-X自定义HIN的SQL时，请在登录命令中加上-c参数，否则由于PolarDB-X自定义HINT是以MySQL注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致PolarDB-X自定义HINT失效，详情请参见MySQL官方客户端命令。

语法

PolarDB-X支持如下两种语法INDEX HINT：

- `FORCE INDEX()`：语法与MySQL FORCE INDEX相同，若指定的索引不是GSI，则会将FORCE INDEX下发到MySQL上执行。

```
# FORCE INDEX()
tbl_name [[AS] alias] [index_hint]
index_hint:
    FORCE INDEX({index_name})
```

- `INDEX()`：通过表名和索引名组合或表在当前查询块中的别名和索引名组合来使用指定的GSI。

```
# INDEX()
/*+TDDL:
    INDEX({table_name | table_alias}, {index_name})
*/
```

> **说明** 上述语句在以下情况中不会生效：
> - 查询中不存在指定的表名或别名。
> - 指定的索引不是指定表上的GSI。

示例

```
CREATE TABLE t_order (
 `id` bigint(11) NOT NULL AUTO_INCREMENT,
 `order_id` varchar(20) DEFAULT NULL,
 `buyer_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 `order_snapshot` longtext DEFAULT NULL,
 `order_detail` longtext DEFAULT NULL,
 PRIMARY KEY (`id`),
 GLOBAL INDEX `g_i_seller`(`seller_id`) dbpartition by hash(`seller_id`),
 UNIQUE GLOBAL INDEX `g_i_buyer` (`buyer_id`) COVERING(`seller_id`, `order_snapshot`)
  dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_id`) tbpartitions 3
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
```

- 在FROM子句中通过FORCE INDEX指定使用 `g_i_seller`

```
SELECT a.*, b.order_id
   FROM t_seller a
     JOIN t_order b FORCE INDEX(g_i_seller) ON a.seller_id = b.seller_id
   WHERE a.seller_nick="abc";
```

- 通过INDEX加上表的别名指定使用 `g_i_buyer`

```
/*+TDDL:index(a, g_i_buyer)*/ SELECT * FROM t_order a WHERE a.buyer_id = 123
```

8.20.5. 如何自定义SQL超时时间

在PolarDB-X中，PolarDB-X节点与RDS的默认SQL执行超时时间是900秒（可以调整），但是对于某些特定的慢SQL，其执行时间可能超过了900秒 。针对这种慢SQL，PolarDB-X提供了调整超时时间的自定义HINT。通过这个自定义HINT可以任意调整SQL执行时长。

注意事项

- PolarDB-X自定义HINT支持 `/*+TDDL:hint_command*/` 和 `/!+TDDL:hint_command*/` 两种格式。
- 如果使用 `/*+TDDL:hint_command*/` 格式，在使用MySQL官方命令行客户端执行带有PolarDB-X自定义HINT的SQL时，请在登录命令中加上-c参数。否则，由于PolarDB-X自定义HINT是以MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致PolarDB-X自定义HINT失效。具体请参见MySQL 官方客户端命令

语法

PolarDB-X自定义SQL超时时间HINT的语法如下：

```
/*+TDDL:SOCKET_TIMEOUT(time)*/
```

其中，SOCKET_TIMEOUT的单位是毫秒。通过该HINT您可以根据业务需要，自由调整SQL语句的超时时间。

示例

设置SQL超时时间为40秒：

```
/*+TDDL:SOCKET_TIMEOUT(40000)*/SELECT * FROM t_item;
```

超时时间设置得越长，占用数据库资源的时间就会越长。如果同一时间长时间执行的SQL过多，可能消耗大量的数据库资源，从而导致无法正常使用数据库服务。所以，对于长时间执行的SQL语句，尽量对SQL语句进行优化。

8.20.6. 指定分库执行SQL

本文介绍了指定分库执行SQL的HINT语法和示例。

在使用PolarDB-X的过程中，如果遇到某个PolarDB-X不支持的SQL语句，可以通过PolarDB-X提供的 `NODE HINT` ，直接将SQL下发到一个或多个分库上去执行。此外如果需要单独查询某个分库或者已知分库的某个分表中的数据，也可以使用 `NODE HINT` ，直接将SQL语句下发到分库中执行。

语法

`NODE HINT` 支持通过分片名指定SQL在分库上执行。其中分片名是PolarDB-X中分库的唯一标识，可以通过 `SHOW NODE` 语句得到。

通过分库名指定SQL在分库上执行分两种使用方式，分别是指定SQL在某个分库上执行和指定SQL在多个分库上执行。

> **注意** 如果在目标表包含Sequence的INSERT语句上使用了指定分库的HINT，那么Sequence将不生效。更多相关信息，请参考使用限制。

- 指定SQL在某个分库上执行：

  ```
  /*+TDDL:node('node_name')*/
  ```

  `node_name` 为分片名，通过这个PolarDB-X自定义HINT，就可以将SQL下发到 `node_name` 对应的分库中执行。

- 指定SQL在多个分库上执行：

  ```
  /*+TDDL:node('node_name'[,'node_name1','node_name2'])*/
  ```

  在参数中指定多个分片名，将SQL下发到多个分库上执行，分片名之间使用逗号分隔。

> 说明
> - 使用该自定HINT 时，PolarDB-X会将SQL直接下发到分库上执行，所以在SQL语句中，表名必须是该分库中已经存在的表名。
> - `NODE HINT` 支持 DML、DDL、DAL语句。

注意事项

- PolarDB-X自定义HINT支持 `/*+TDDL:hint_command*/` 和 `/!+TDDL:hint_command*/` 两种格式。
- 如果使用 `/*+TDDL:hint_command*/` 格式，在使用MySQL官方命令行客户端执行带有PolarDB-X自定义HINT的SQL时，请在登录命令中加上 `-c` 参数。否则，由于PolarDB-X自定义HINT是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致PolarDB-X自定义HINT失效。具体请查看MySQL 官方客户端命令。

示例

对于名为 `drds_test` 的PolarDB-X数据库， `SHOW NODE` 的结果如下：

```
mysql> SHOW NODE\G
*************************** 1. row ******************
                ID: 0
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS
  MASTER_READ_COUNT: 212
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 2. row ******************
                ID: 1
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0001_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 3. row ******************
                ID: 2
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0002_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 4. row ******************
                ID: 3
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0003_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 5. row ******************
                ID: 4
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0004_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 6. row ******************
                ID: 5
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0005_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 7. row ******************
                ID: 6
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0006_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
*************************** 8. row ******************
                ID: 7
              NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0007_RDS
  MASTER_READ_COUNT: 29
   SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
 SLAVE_READ_PERCENT: 0%
8 rows in set (0.02 sec)
```

可以看到每个分库都有 `NAME` 这个属性，这就是分库的分片名。每个分片名都唯一对应一个分库名，比如 `DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0003_RDS` 这个分片名对应的分库名是 `drds_test_vtla_0003` 。得到了分片名，就可以使用PolarDB-X的自定义HINT指定分库执行SQL语句了。

- 指SQL在第0个分库上执行：

```
SELECT /*TDDL:node('DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS')*/ * FROM table_name;
```

- 指定SQL在多个分库上执行：

```
SELECT /*TDDL:node('DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS','DRDS_TEST_1473471355140LRPRDRDS_T
EST_VTLA_0006_RDS')*/ * FROM table_name;
```

这条SQL语句将在 `DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS` 、 `DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0006_RDS` 这两个分片上执行。

- 查看SQL在第0个分库上物理执行计划：

```
/*TDDL:node('DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS')*/ EXPLAIN SELECT * FROM table_name; ```
```

8.20.7. 扫描全部/部分分库分表

本文介绍了扫描全部/部分分库分表的HINT语法和示例。

除了可以将SQL单独下发到一个或多个分库执行，PolarDB-X还提供了扫描全部/部分分库与分表的 `SCAN HINT` 。使用 `SCAN HINT` ，您可以一次将SQL下发到每一个分库执行，比如查看某个分库上的所有分表，或者查看某个逻辑表的每张物理表中的数据量等。

通过 `SCAN HINT` ，可以指定四种执行SQL的方式：

1. 在所有分库的所有分表上执行；
2. 在指定分库的所有分表上执行；
3. 在指定分库分表上执行，根据条件计算物理表名称；
4. 在指定分库分表上执行，显式指定物理表名；

`SCAN HINT` 支持 DML、DDL和部分DAL语句。

语法

```
# SCAN HINT
# 将SQL语句下发到所有分库的所有分表上执行
SCAN()
# 将SQL语句下发到指定分库的所有分表上执行
SCAN(NODE="node_list")                # 指定分库
# 将SQL语句下发到指定分库分表上执行，根据条件计算物理表名称
SCAN(
  [TABLE=]"table_name_list"           # 逻辑表名
  , CONDITION="condition_string"      # 使用TABLE和CONDITION中的内容计算物理库表名称
  [, NODE="node_list"] )              # 过滤通过CONDITION计算出的结果，仅保留指定物理库
# 将SQL语句下发到指定分库分表上执行，显式指定物理表名
SCAN(
  [TABLE=]"table_name_list"           # 逻辑表名
  , REAL_TABLE=("table_name_list")    # 物理表名，对所有物理库使用相同的物理表名
  [, NODE="node_list"] )              # 过滤通过CONDITION计算出的结果，仅保留指定物理库
# 物理/逻辑表名列表
table_name_list:
    table_name [, table_name]...
# 物理库列表，支持GROUP_KEY和GROUP的序号，可以通过`SHOW NODE`语句获得
node_list:
    {group_key | group_index} [, {group_key | group_index}]...
# 支持SQL WHERE的语法，需要为每一张表设置条件，如：t1.id = 2 and t2.id = 2
condition_string:
    where_condition
```

注意事项

- PolarDB-X自定义HINT支持 `/*+TDDL:hint_command*/` 和 `/!+TDDL:hint_command*/` 两种格式。
- 如果使用 `/*+TDDL:hint_command*/` 格式，在使用MySQL官方命令行客户端执行带有PolarDB-X自定义HINT的SQL时，请在登录命令中加上 `-c` 参数。否则，由于PolarDB-X自定义HINT是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致PolarDB-X自定义HINT失效。具体请查看MySQL 官方客户端命令。

示例

- 在所有分库的所有分表上执行：

  ```
  SELECT /*+TDDL:scan()*/ COUNT(1) FROM t1
  ```

  执行后会下发SQL语句到 `t1` 的所有物理表上执行，并将结果集合并后返回。

- 在指定分库的所有分表上执行：

  ```
  SELECT /*+TDDL:scan(node='0,1,2')*/ COUNT(1) FROM t1
  ```

  执行后会首先计算出 `t1` 在0000, 0001, 0002分库上的所有物理表，然后下发SQL语句并将结果集合并后返回。

- 按条件在指定分表上执行：

  ```
  SELECT /*+TDDL:scan('t1', condition='t1.id = 2')*/ COUNT(1) FROM t1
  ```

  执行后会首先计算出逻辑表 `t1` 满足 `condition` 条件的所有物理表，然后下发SQL语句并将结果集合并后返回。

- 按条件在指定分表上执行，有JOIN的情况：

  ```
  SELECT /*+TDDL:scan('t1, t2', condition='t1.id = 2 and t2.id = 2')*/ * FROM t1 a JOIN t2 b ON a.id = b.id
  WHERE b.name = "test"
  ```

  执行后会首先计算出逻辑表 `t1` `t2` 满足 `condition` 条件的所有物理表，然后下发 SQL 语句并将结果集合并后返回。
  注意：使用该自定义注释需要保证两张表的分库和分表数量一致，否则PolarDB-X计算出的两个键值对应的分库不一致，就会报错。

- 在指定分库分表上执行，显式指定物理表名：

```
SELECT /*+TDDL:scan('t1', real_table=("t1_00", "t1_01"))*/ COUNT(1) FROM t1
```

执行后会下发SQL语句到所有分库的 `t1_00``t1_01` 分表上，合并结果集后返回。

- 在指定分库分表上执行，显式指定物理表名，有JOIN的情况：

```
SELECT /*+TDDL:scan('t1, t2', real_table=("t1_00,t2_00", "t1_01,t2_01"))*/ * FROM t1 a JOIN t2 b ON a.id =
b.id WHERE b.name = "test";
```

执行后会下发SQL语句到所有分库的 `t1_00` `t2_00` `t1_01` `t2_01` 分表上，合并结果集后返回。

9.最佳实践

9.1. 如何选择实例规格

实例规格越高代表实例的性能越强，本文介绍了选择实例规格的方法。

PolarDB-X实例主要由计算节点和存储节点联合提供服务，单个节点按照CPU/MEM来划分实例的多种规格，多个节点一起组成PolarDB-X实例。实例规格请参见规格说明。

节点规格类型

系列	规格码	CPU和内存	最大存储	最大连接数	最大IOPS	特点
通用	polarx.x4.medium.2e	2核8G	3072GB	20000	4000	定位入门级，用于测试、体验和极小负载的场景。
	polarx.x4.large.2e	4核16G	3072GB	20000	7000	CPU和MEM配比为1:4，复用计算资源享受规模红利，性价比高。
	polarx.x4.xlarge.2e	8核32G	3072GB	20000	12000	
	polarx.x4.2xlarge.2e	16核64G	3072GB	20000	14000	
独享	polarx.x8.large.2e	4核32G	3072GB	20000	9000	CPU和MEM配比为1:8，独占分配到的计算资源（如CPU），性能表现更加稳定。
	polarx.x8.xlarge.2e	8核64G	3072GB	20000	18000	
	polarx.x8.2xlarge.2e	16核128G	3072GB	20000	36000	
	polarx.x8.4xlarge.2e	32核128G	3072GB	20000	36000	
	polarx.x8.4xlarge.2e	32核256G	3072GB	20000	72000	
独占	polarx.st.8xlarge.25	60核470G	6144GB	20000	120000	独占物理机规格，可以有更好的资源使用保障。
	polarx.st.12xlarge.25	90核720G	6144GB	20000	140000	

实例规格=节点数×节点规格 (计算节点+存储节点)

举例如下：

polarx.x8.xlarge.2e独享规格，节点数为2个，性能数据如下：存储6TB (3072GB×2)、连接数40000 (20000×2)、最大IOPS 36000 (18000×2)。

按照存储容量选择

按照业务的存储空间估算逻辑：

1. 业务的数据存储会随着时间而持续增加，可以预估1~2年内的业务增长量，判断需要的最大存储空间。
2. PolarDB-X的数据存储分为：数据空间、系统文件空间、日志空间等，比较建议单节点的存储使用量保持在70%以下。

示例：

当前业务的存储空间为1500GB，每天新增约10GB，按照1年的业务预估来看，总计约5150GB的存储。按照使用量70%来计算，预估需要5150GB / 0.7 = 7357GB的存储空间诉求，如果按照独享规格polarx.x8.xlarge.2e (节点存储上限3TB)，最后判断需要CEILING(7357GB/3072GB) = CEILING(2.39) = 3个节点。

按照并发量选择

按照业务的并发量的估算逻辑：

1. PolarDB-X的节点规格资源限制，包含CPU、MEM、连接数、IOPS等。在面向事务型场景下，一般比较常见是CPU瓶颈为主，可通过业务的QPS预期进行估算和推导。
2. 按照常见的sysbench/TPC-C的偏交易混合读写场景，单core估算可支持的QPS为1000~3000，按照独享规格polarx.x8.xlarge.2e单节点预估可支持1~2万的QPS。

   > **说明** 业务的流量模型和通用benchmark会有比较多的差异，单节点的QPS仅供估算参考，比较建议基于业务流量进行实际压测。

3. 常规的峰值流量，PolarDB-X建议单节点的资源使用量保持在70%以下。

示例：

当前业务的QPS峰值预估为10万QPS，预留70%的资源余量，预计需要支持14万QPS的资源，按照PolarDB-X单节点支持2万的能力来估算，预估需要7个节点。

按照多维度组合选择

示例：

当前业务的QPS峰值预估为10万QPS，当前业务的存储空间为1500GB，每天新增约10GB，按照1年的业务预估来看，总计约5150GB的存储。

建议的选择逻辑：

1. 分布式数据库由多个节点组成，会有类似的木桶效应，比如突发流量导致个别节点达到资源瓶颈，会导致整体实例出现部分慢SQL的现象。因此，节点规格推荐独享型，建议生产环境8核64G起步，默认存储空间有3072GB（3TB）。
2. 分别按照存储容量和并发量分别估算需要的节点数和CPU规格，比如例子中需要CPU 56核、存储7357GB，可以按照最小覆盖原则进行计算。存储空间最小需要3个节点覆盖，PolarDB-X提供了存储包的按量付费模式，存储需要的节点数可以作为下限，上限可以选择CPU核数的最小覆盖，可以选择7个节点的8核64G或4个节点的16核128G。
3. 业务流量如果包含报表分析的场景，因涉及更多数据计算的代价，建议选择4个节点的16核128G，优先大节点规格，提高木桶边的上限。另外的场景下，建议选择7个节点的8核64G，更多的节点数可以支撑更大的存储空间，未来实例规格的升配也优先建议升配单个节点规格。

9.2. 如何选择升配与扩容

本文介绍了选择升配与扩容的逻辑和方法。

升配与增加节点

PolarDB-X中，扩容包含两种方式：

1. 升配，指节点数不变，升级已有节点的CPU、内存、IOPS等规格。

   升配优先会选择原地升配，例如，将节点由4C升配为8C，如果当前节点所在宿主机有空余的CPU资源，则会通过调整参数的方式，将4C升配为8C，此种形式的升配耗时比较短，特别是数据节点的升配，不涉及数据的迁移；如果节点宿主机没有足够的资源做原地升配，则会进行节点的迁移，相对会消耗更多的时间。

   一般情况下，节点规格相对较小的时候，能更容易做到原地升配。相应的，降配操作可以做到原地降配。

2. 增加节点，指节点的规格不变，增加新的CN与DN节点。

   增加DN节点会对数据做重新的负载均衡，因此涉及到数据的迁移，所耗时间与数据量成比例。相应的，减少节点的操作也会涉及到数据的迁移。

   可以看出，大多情况下，升配可以在原地完成，是一种更轻量的操作；增加节点一定涉及到数据迁移，是一个更复杂的操作。

   因此选择升配还是扩容，遵循以下原则：

i. 业务突发流量、高峰期等引起的资源不足，优先选择升配，可以更快的解决性能瓶颈。同时，业务高峰过去之后，也更方便使用降配来节省资源。

ii. 日常开发、运维等过程中计划内的增加资源，优先选择增加节点，将单节点规格保持在相对较小的状态，便于应急情况下进行原地升配。

iii. 升配不会增加磁盘容量，磁盘容量是瓶颈的情况下，选择增加节点。

何时选择扩容

对于在线交易类业务，一般推荐日常负载不超过水位线的30%，以应付突发的一些流量（例如促销、甚至业务代码出现BUG等）。业务应该根据自己的业务特点（例如是否有周期性波动，是否有大促等）来对安全水位线进行调整。

CN的负载一般重点关注以下指标：

1. CPU使用率
2. 活跃线程数 (runing thread)
3. 响应时间(逻辑RT、物理RT)

DN的负载一般重点关注以下指标：

1. CPU使用率
2. IOPS使用率
3. 活跃链接数 (active session)

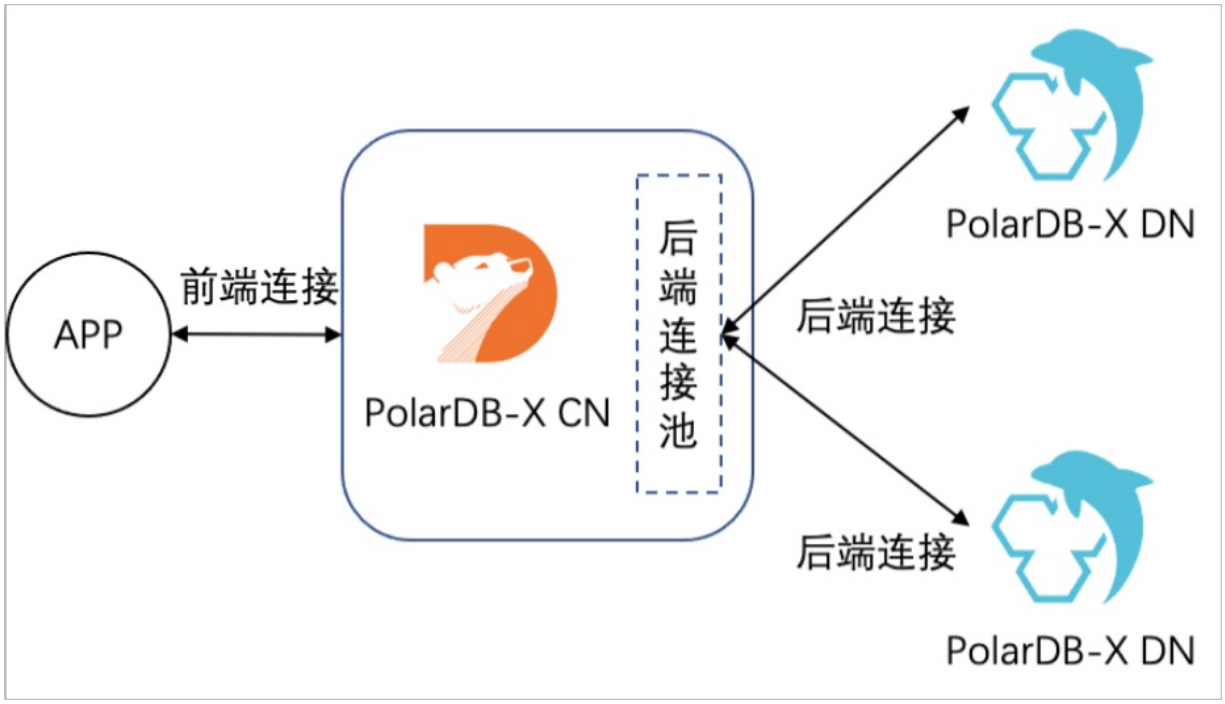
对于磁盘空间成为瓶颈的场景，考虑到增加节点需要消耗一定的时间，因此增加节点需要提前进行，一般推荐磁盘容量超过70%就应该进行增加节点的操作。

9.3. 如何选择应用端链接池

背景

当应用程序连接PolarDB-X实例执行操作时，从PolarDB-X实例的角度看，会有如下两种类型的连接：

- 前端连接：由应用程序建立的，到PolarDB-X计算节点（CN）中逻辑库的连接。
- 后端连接：由PolarDB-X计算节点建立的，到后端数据节点（DN）中物理库的连接。



其中后端连接由CN管理，通过私有协议实现TCP连接与后端连接解绑，对用户透明。前端连接由用户创建和管理，本文主要讨论前端连接管理的最佳实践。

> **说明** 为了简化描述，下文中的 “连接” 均代指 “前端连接”。

QPS/RT与连接数的关系

每秒查询请求数 (Query Per Second，QPS) 和响应时间 (Response Time，RT) 是衡量应用对数据库性能需求的基本指标，QPS代表应用并发访问数据库的需求，RT代表单条语句的处理性能。RT的高低与执行的SQL是否复杂和扫描的数据量紧密相关，OLTP系统中查询RT较低，通常以毫秒为单位。

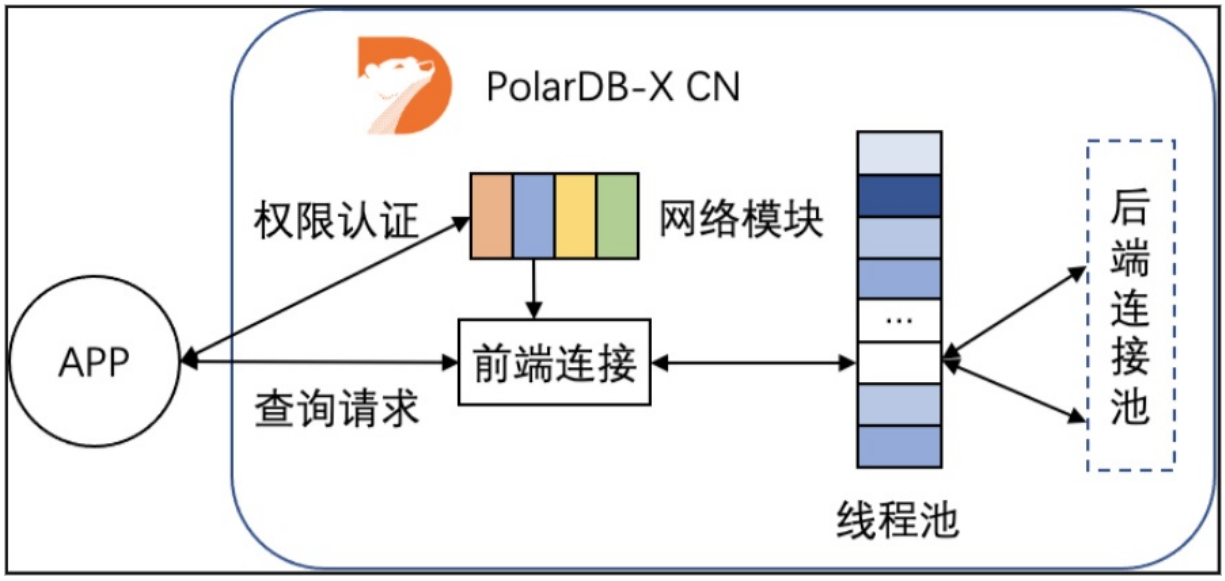
PolarDB-X兼容MySQL协议，请求在单个连接上串行执行，不同连接上的请求可以并行执行，由此得到以下公式：

- 单个连接的QPS上限 = 1000/RT
- 应用访问单个CN的QPS上限 = 单个连接的QPS上限×连接数

按照平均RT为5ms计算，单个连接的QPS上限为200，假设应用预估需要的QPS为5000，则至少需要建立25个连接。

连接数限制

PolarDB-X中前端连接仅与网络连接绑定，连接的数量理论上仅受限于CN可用的内存大小和网络连接数。但在实际的场景中，应用创建连接是为了执行查询，连接数需要与执行线程的数量相匹配才能达到最佳性能。



如上图所示，应用发起连接请求后，首先由网络模块进行权限认证，认证通过则创建连接对象。与 MariaDB相似，PolarDB-X计算节点收到后续查询请求后，会尝试从线程池中分配一个执行线程，用于处理查询请求。单个CN节点默认维护一个1024大小的线程池，如果并发查询数量超过线程池大小，后续请求会排队等待。由此可以得到以下公式：

- 应用访问单个CN的QPS上限 = 单个连接的QPS上限×MIN（连接数, 线程池大小）
- 应用访问数据库的QPS上限 = 单个连接的QPS上限×MIN（连接数, 线程池大小）×CN数量

我们通过以下两个示例来进行公式的实际应用：

- 示例1

  问：查询的平均RT为10ms，理想情况下两节点CN能提供多少QPS？

  答：平均RT为10ms，则单个连接的QPS上限为100。理想情况下（CPU不成为瓶颈），包含2个CN节点的PolarDB-X实例默认能够提供的QPS上限为 100×1024×2 = 204800。这里需要注意，CN节点能够同时处理的查询数量与CN规格和查询复杂度有关，实际场景下通常不能做到1024个线程完全并行，QPS上限会低于204800。

- 示例2

  问：在CN规格为16C的PolarDB-X实例上压测，CPU刚好跑满时，某查询的平均RT为5ms。仅考虑CN节点，支撑40w的QPS应当如何选择实例和设置应用端连接池？

  答：平均RT为5ms，则单个连接的QPS上限为200。应用端连接池大小设置为400000/200 = 2000可以将额外开销降到最低。保持单个CN节点上的并行度不超过1024，需要两个16C节点构成的32C的 PolarDB-X实例。

连接池

数据库连接池是对数据库连接进行统一管理的技术，主要目的是提高应用性能，减轻数据库负载。

- 提高系统响应效率：连接的初始化工作完成后，所有请求可以直接利用现有连接，避免了连接初始化和释放的开销，提高了系统的响应效率。
- 资源复用：连接可以重复利用，避免了频繁创建、释放连接引入的性能开销。在减少系统消耗的基础上，增强了系统的平稳性。
- 避免连接泄漏：连接池可根据预设的回收策略，强制回收连接，从而避免了连接资源泄漏。

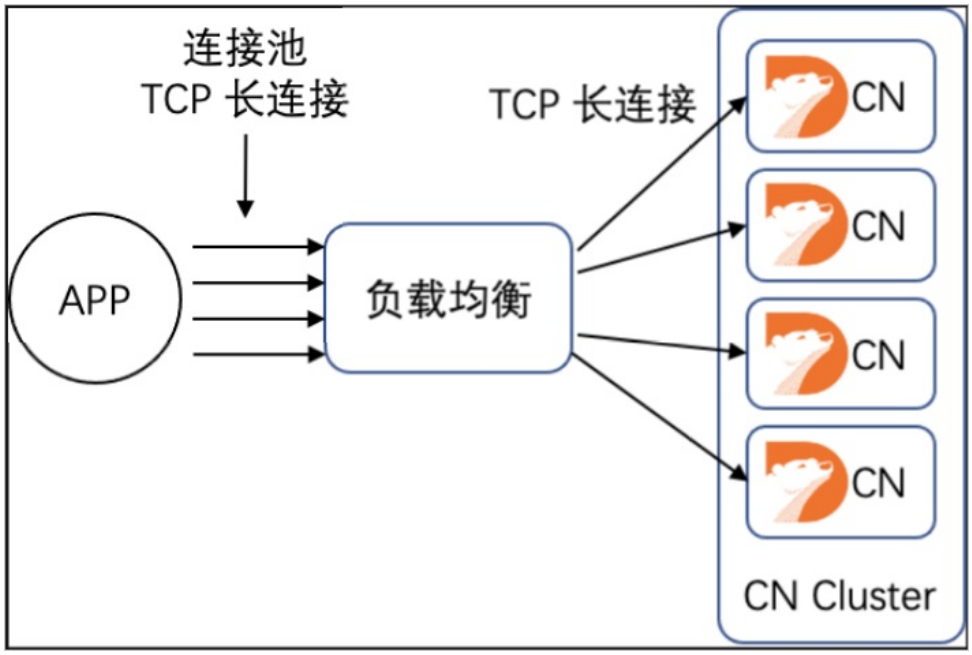
如果是Java程序，推荐使用 Druid 连接池，版本要求1.1.11及以上。

Druid的Spring标准配置如下：

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="
close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- 基本属性 URL、user、password -->
        <property name="url" value="jdbc:mysql://ip:port/db?autoReconnect=true&rewriteBatchedStatements=true
&socketTimeout=30000&connectTimeout=3000" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
        <!-- 配置初始化大小、最小、最大 -->
        <property name="maxActive" value="20" />
        <property name="initialSize" value="3" />
        <property name="minIdle" value="3" />
        <!-- maxWait 获取连接等待超时的时间 -->
        <property name="maxWait" value="60000" />
        <!-- timeBetweenEvictionRunsMillis 间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <!-- minEvictableIdleTimeMillis 一个连接在池中最小空闲的时间，单位是毫秒-->
        <property name="minEvictableIdleTimeMillis" value="300000" />
        <!-- 检测连接是否可用的 SQL -->
        <property name="validationQuery" value="select 'z' from dual" />
        <!-- 是否开启空闲连接检查 -->
        <property name="testWhileIdle" value="true" />
        <!-- 是否在获取连接前检查连接状态 -->
        <property name="testOnBorrow" value="false" />
        <!-- 是否在归还连接时检查连接状态 -->
        <property name="testOnReturn" value="false" />
        <!-- 是否在固定时间关闭连接。增加此参数可以均衡后端服务节点参数 -->
        <property name="phyTimeoutMillis" value="600000" />
        <!-- 是否在固定SQL使用次数之后关闭连接，增加此参数可以均衡后端服务节点参数-->
        <property name="phyMaxUseCount" value="10000" />
    </bean>
```

注意事项

- 连接池与负载均衡



连接池模式（TCP 长连接）效率更高，但部分场景下对分布式负载均衡不友好，可能导致CN负载不均匀

i. 突发创建连接，导致分布不均

如果应用存在突发创建大量连接的情况，负载均衡设备无法及时刷新统计信息，可能出现部分CN上连接较多，结合连接池化，最终导致部分CN压力高于其他CN，影响系统总体性能。

ii. 负载均衡探活异常，导致分布不均

负载均衡通过主动探活来判断CN节点是否正常，当探活出现偶发异常时，可能导致部分CN上连接较少，结合连接池化，最终导致部分CN压力低于其他CN，影响系统总提性能。

Druid连接池增加了phyTimeoutMillis/phyMaxUseCount参数，定期（例如执行10000次或者10分钟）刷新连接池中的连接，可以在解决上述问题的同时保持性能基本不变，建议默认添加这两个配置。

- 应用线程数与连接池

  应用程序访问数据库的一种常见模式，是在应用程序中创建多个线程，每个线程获取一个到数据库的连接并执行查询。为了减少创建/释放线程的开销，通常会使用“线程池”来管理线程，线程池的一个重要参数是“最大线程数”，需要根据实际情况调整。

  理想情况下，查询的RT波动不大，可以应用上文介绍的公式，根据RT计算出合理的连接池大小，并按照“每个线程一个数据库连接”的思路确定最大线程数。实际场景中，查询RT受到热点、锁、数据倾斜等多种因素的影响，可能出现突发RT增长，甚至部分连接失去响应。如果完全按照理想情况连接池/线程池，可能由于部分慢查询耗尽连接池/线程池，导致应用失去响应，影响关联系统。因此，建议按照“理想情况”的1.5到2倍来设置最大连接数/线程数。

9.4. 如何快速定位及解决数据库问题

本文介绍了数据库发生故障时的快速判断方法和解决办法。

如何定位系统瓶颈是否在数据库上

- **通过Processlist来判断**

  执行以下语句，显示PolarDB-X上所有正在执行的SQL语句。

  ```
  SHOW PROCESSLIST WHERE INFO IS NOT NULL
  ```

  一般情况下，语句堆积会伴随着数据库卡慢一起出现，因此如果该语句的显示结果中没有大量执行时间大于0的语句，则基本可以断定问题不在数据库层面，反之，则说明数据库可能存在瓶颈。

- **通过堆栈信息来判断**

应用与数据库之间通过TCP协议进行交互，如果数据库层出现瓶颈，则会产生应用将请求通过socket发送给了数据库，但是数据库不返回结果的情况，此时socket会阻塞在read方法上。因此我们可以通过应用当前的堆栈信息来判断是否在数据库层面发生了阻塞。本文以Java应用为例说明：

i. 通过jstack命令dump堆栈信息。

ii. 在dump出的信息中搜索mysql驱动等待请求返回的堆栈，内容如下：

```
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at com.mysql.jdbc.util.ReadAheadInputStream.fill(ReadAheadInputStream.java:101)
        at com.mysql.jdbc.util.ReadAheadInputStream.readFromUnderlyingStreamIfNecessary(ReadAheadInput
Stream.java:144)
        at com.mysql.jdbc.util.ReadAheadInputStream.read(ReadAheadInputStream.java:174)
        - locked <0x00000002eb8f2d98> (a com.mysql.jdbc.util.ReadAheadInputStream)
        at com.mysql.jdbc.MysqlIO.readFully(MysqlIO.java:3183)
        at com.mysql.jdbc.MysqlIO.reuseAndReadPacket(MysqlIO.java:3659)
        at com.mysql.jdbc.MysqlIO.reuseAndReadPacket(MysqlIO.java:3649)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:4090)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:972)
        at com.mysql.jdbc.MysqlIO.readAllResults(MysqlIO.java:2497)
        at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2870)
        at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2806)
```

如果有大量的线程的堆栈情况如上例所示，则代表大量线程阻塞在等待数据库返回，说明瓶颈可能在数据库层面，反之，则应重点排查应用本身是否存在瓶颈。

数据库问题快速处置

在通过上述方法判断数据库存在瓶颈之后，推荐依次使用以下方法进行快速恢复。

方法一：KILL所有语句

如果Processlist中显示堆积了很多SQL，建议立即KILL掉所有正在执行的语句，PolarDB-X提供了如下指令进行这个操作：

```
KILL "ALL"
```

该语句会KILL掉计算节点与数据节点之间的每一个连接，从而达到结束掉所有语句的效果。

方法二：重启应用

执行方法一后，等待一段时间如果再次产生语句堆积，建议重启应用，避免应用因为处于某种错误的状态，不断的重试高代价的SQL。

方法三：SQL限流

方法2依然无法解决问题之后，建议使用PolarDB-X的CCL_RULES（限流功能）。

1. 执行 `SHOW FULL PROCESSLIST` 命令，找到占比比较高的SQL的模板ID。

```
+----+---------------+-----------------+----------+-------------------------------+------+-------+------
-----------------+-----------------+
| ID | USER          | HOST            | DB       | COMMAND                       | TIME | STATE | INFO
| SQL_TEMPLATE_ID |
+----+---------------+-----------------+----------+-------------------------------+------+-------+------
-----------------+-----------------+
|  2 | polardbx_root | ***.*.*.*:62787 | polardbx | Query                         |    0 |       | show
full processlist | NULL            |
|  1 | polardbx_root | ***.*.*.*:62775 | polardbx | Query(Waiting-selectrulereal) |   12 |       | selec
t 1              | 9037e5e2        |
+----+---------------+-----------------+----------+-------------------------------+------+-------+------
-----------------+-----------------+
2 rows in set (0.08 sec)
```

2. 通过模板ID对该类型的SQL进行限流，例如：

```
CREATE CCL_RULE IF NOT EXISTS `test` ON *.* TO 'ccltest'@'%'
FOR SELECT
FILTER BY TEMPLATE('9037e5e2')
WITH MAX_CONCURRENCY=10;
```

方法四：重启数据库

以上方法都无效的情况下，请重启数据库。

9.5. 如何正确处理DDL异常

本文介绍处理DDL异常的常用方法。

概述

作为一款分布式数据库，PolarDB-X中的一条DDL语句背后隐藏着复杂的数据处理流程。例如创建一张拆分表时，实际上会在多个数据节点上创建很多张物理MySQL表。PolarDB-X的DDL处理框架拥有一定的容错能力，会保证DDL的正确性和一致性。但在特殊情况下，需要手动处理DDL异常。

处理步骤

下面以创建全局唯一二级索引为例，介绍处理DDL异常的步骤。创建全局唯一二级索引时，会校验列中数据的全局唯一性，如果发现列中数据不唯一，则会导致DDL的执行失败。此时DDL的状态可能变为“PAUSED”或“ROLLBACK_PAUSED”。

1. **查看DDL的执行状态**

   执行 `SHOW DDL` 语句，查看DDL的执行状态。只有当DDL状态为“PAUSED”或“ROLLBACK_PAUSED”时，才需要手动处理。

```
mysql> show ddl\G;
*************************** 1. row ***************************
           JOB_ID: 1359992239576580096
    OBJECT_SCHEMA: d1
      OBJECT_NAME: t1
           ENGINE: DAG
         DDL_TYPE: ALTER_TABLE
            STATE: PAUSED
BACKFILL_PROGRESS: 0%
 PHY_DDL_PROGRESS: 100%
         PROGRESS: 80%
       START_TIME: 2021-08-05 13:57:57.852
         END_TIME: 2021-08-05 13:58:30.804
 ELAPSED_TIME(MS): 32952
      PHY_PROCESS:
       CANCELABLE: true
1 row in set (0.03 sec)
```

2. **判断错误原因**

   通常DDL异常发生时，会直接返回错误信息。有时DDL的错误信息可能难以获取（例如DDL是异步执行的），可以通过 `SHOW DDL RESULT` 语句查看近期执行过的DDL。在本示例中，从返回的错误信息中看到出现了UNIQUE KEY的冲突。

```
mysql> alter table t1 add unique global INDEX `idx_c2`(`c2`) DBPARTITION BY HASH(`c2`) tbpartition by ha
sh(c2) tbpartitions 3;
ERROR 3009 (HY000): [12dfa9bc5c800000][30.39.226.95:8527][d1]Failed to execute the DDL task. Caused by:
ERR-CODE: [TDDL-5321][ERR_GLOBAL_SECONDARY_INDEX_BACKFILL_DUPLICATE_ENTRY] Duplicated entry '100018' for
key 'PRIMARY'
mysql> show ddl result\G;
*************************** 1. row ***************************
        JOB_ID: 1359992239576580096
   SCHEMA_NAME: d1
   OBJECT_NAME: t1
      DDL_TYPE: ALTER_TABLE
   RESULT_TYPE: ERROR
RESULT_CONTENT: Failed to execute the DDL task. Caused by: ERR-CODE: [TDDL-5321][ERR_GLOBAL_SECONDARY_IN
DEX_BACKFILL_DUPLICATE_ENTRY] Duplicated entry '100018' for key 'PRIMARY'
```

3. **恢复或取消DDL**

   大多数情况下，DDL出错时都能够自动恢复或取消。特殊情况需要手动处理DDL异常。各个DDL的容错策略可能不同，例如 `CREATE TABLE` 语句的容错策略是：自动尝试恢复，多次失败则自动取消。

   可以手动恢复或取消DDL任务。

   - 手动恢复：使用 `CONTINUE DDL` 语句可以恢复DDL任务。
   - 手动取消：使用 `CANCEL DDL` 语句可以取消DDL任务。有些DDL任务可能是无法取消的，可以通过 `SHOW DDL` 语句返回的“CANCELABLE”字段查看是否可以取消。

```
-- 处理完UNIQUE KEY的数据冲突后，可以使用continue ddl语句继续执行
mysql> continue ddl 1359992239576580096;
Query OK, 1 row affected (1.63 sec)
-- 如果不想再继续添加这个全局唯一二级索引，则可以直接取消任务
mysql> cancel ddl 1359992239576580096;
Query OK, 1 row affected (0.03 sec)
```

4. **检查表一致性和正确性**

   DDL执行成功后，建议使用以下语句检查表的一致性和表结构的正确性：

   - 使用 `CHECK TABLE` 语句检查表的一致性：如果返回结果为OK，说明表已经处于一致性状态，可以正常使用；如果返回报错信息，提示表处于不一致状态，请联系阿里云技术支持。
   - 使用 `SHOW CREATE TABLE` 语句查看表结构的正确性。

```
mysql> check table t1;
+-------+-------+----------+----------+
| TABLE | OP    | MSG_TYPE | MSG_TEXT |
+-------+-------+----------+----------+
| d1.t1 | check | status   | OK       |
+-------+-------+----------+----------+
1 row in set (0.05 sec)
mysql> show create table t1\G;
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
    `c1` bigint(20) NOT NULL AUTO_INCREMENT BY GROUP,
    `c2` bigint(20) DEFAULT NULL,
    `c3` bigint(20) DEFAULT NULL,
    PRIMARY KEY USING BTREE (`c1`),
    UNIQUE GLOBAL KEY `idx_c2` (`c2`) COVERING (`c1`) DBPARTITION BY HASH(`c2`) TBPARTITION BY HASH(`c2`
) TBPARTITIONS 3
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4  dbpartition by hash(`c1`) tbpartition by hash(`c1`) tbparti
tions 3
1 row in set (0.02 sec)
```

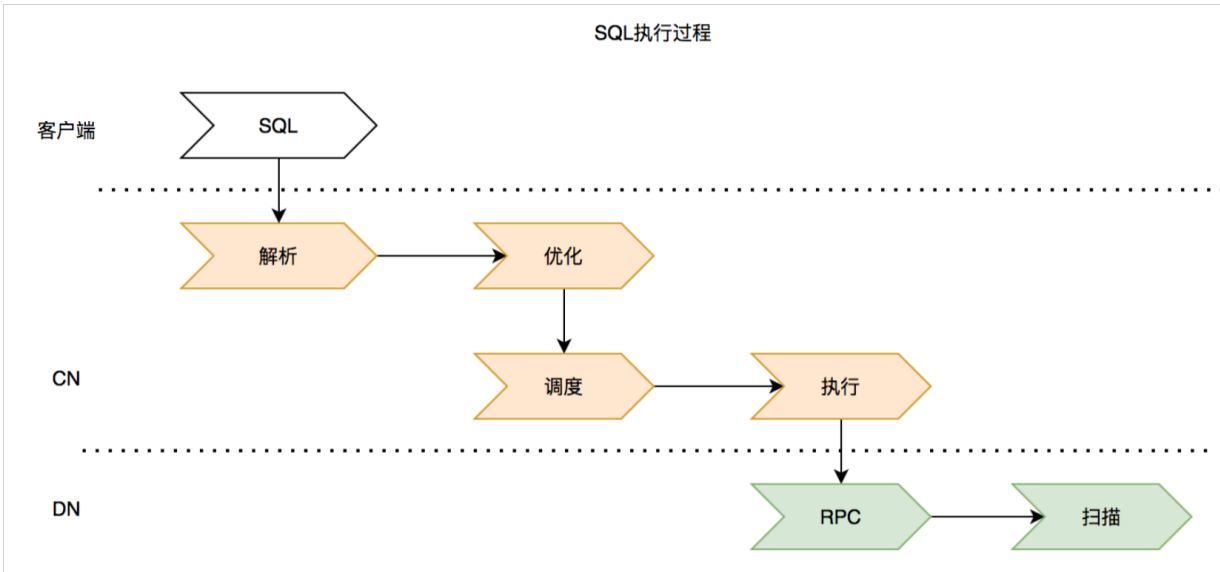
9.6. 如何分析及优化慢SQL

本文介绍了分析和解决慢SQL的方法。

从数据库角度看，每个SQL执行都需要消耗一定I/O资源，SQL执行的快慢，决定资源被占用时间的长短。假如有一条慢SQL占用了30%的资源共计1分钟。那么在这1分钟时间内，其他SQL能够分配的资源总量就是70%，如此循环，当资源分配完的时候，所有新的SQL执行将会排队等待。所以往往一条慢SQL会影响到整个业务。

PolarDB-X中SQL的执行过程

在分析慢SQL之前，需要先了解SQL的执行过程。一条SQL的执行共涉及到三层，包含两次网络数据交换：客户端到计算节点（CN）、计算节点到数据节点（DN）。



执行过程示意图说明：

- 橘色部分为不可下推的SQL语句的执行流程，包括解析优化到调度执行两大步骤。
- 浅绿色部分为可以下推的SQL语句的执行流程，主要过程是将数据节点（DN）的数据进行扫描和计算，并将数据通过私有RPC返回给上层计算节点继续做计算。

结合PolarDB-X数据库的原理和SQL语句的执行过程，发现影响DQL语句执行效率的主要因素包括：

1. 数据量
   - SQL执行后返回给客户端的数据量的大小；
   - DN层数据量越大需要扫描的I/O次数越多，DN节点的IO更容易成为瓶颈；
2. 取数据的方式
   - 数据在缓存中还是在磁盘上；
   - 是否通过DN上的全局索引快速寻址；
   - 是否结合谓词条件命中全局索引加速扫描；
3. 数据加工的方式
   - 排序、子查询、聚合、关联等，一般需要先把数据取到临时表中，再对数据进行加工；
   - 对于数据量比较多的计算，会消耗大量计算节点的CPU资源，让数据加工变得更加缓慢；
   - 是否选择了合适的关联算法。

优化思路

1. 将数据存放在更快的地方。

   某条查询涉及到大表，无法进一步优化，如果返回的数据量不大且变化频率不高但访问频率很高，此时应该考虑将返回的数据放在应用端的缓存当中或者Redis这样的缓存当中，以提高存取速度。

2. 关联操作尽量将计算下推
   - 如果业务大部分都是大表关联操作，那么尽可能设计合理的分片策略。将分片的Key与关联条件的Join Key对齐，PolarDB-X就会把这部分关联操作转化成可下推的物理SQL下推，避免单表扫描带来的压力；

- 如果涉及到的慢SQL是典型的大小表关联，而小表的数据量不大，且变化频率不高的，那么可以把小表做成广播表，这样任何表和广播表做关联，都会被优化成可下推的物理SQL；
- 如果关联涉及到的表分片策略无法变更，可以以Join Key为表创建全局二级索引，这样关联操作可以下推，在DN的索引表上做关联。

3. 减少数据扫描
   - 尽量在查询中加入一些可以提前过滤数据的谓词条件，比如按照时间过滤数据等，可以减少数据的扫描量，对查询更友好；
   - 在扫描大表数据时是否可以命中索引，减少回表代价，避免全表扫描。
4. 执行计划中选择更合理的数据加工算法

   比如关联算法有HashJoin、SortMergeJoin、BkaJoin等，每种关联算法都有特定的场景，执行计划选择不合理，会影响到查询代价。

定位慢SQL

- 通过控制台定位具体的慢SQL，可以参考慢日志和性能趋势。
- 通过命令 `show slow` 和 `show physical_slow` 查看逻辑和物理慢SQL，通过指令只能定位到最近100条的慢SQL。

```
mysql> show slow;
+------------------+---------------+-----------+---------------------+--------------+------------+--------
------------+
| TRACE_ID         | USER          | HOST      | START_TIME          | EXECUTE_TIME | AFFECT_ROW | SQL
|
+------------------+---------------+-----------+---------------------+--------------+------------+--------
------------+
| 12f812912f400000 | polardbx_root | 127.0.0.1 | 2021-08-24 12:59:38 |         6475 |          0 | show ph
ysical_slow |
| 12f8128383400000 | polardbx_root | 127.0.0.1 | 2021-08-24 12:59:19 |         1899 |         -1 | show ph
ysical_slow |
+------------------+---------------+-----------+---------------------+--------------+------------+--------
------------+
2 rows in set (0.01 sec)
mysql> show physical_slow;
Empty set (0.01 sec)
```

分析慢SQL

慢SQL产生的原因主要包括以下三大类：

- 业务问题：明显的数据倾斜、不合理的分片策略设置、数据返回过多等和业务使用相关问题；
- 系统问题：流量太大，资源成为瓶颈或者网络抖动造成的问题；
- 执行问题：如选错索引，选错Join类型或顺序等问题。

从业务角度分析

1. 如果慢SQL本身是简单查询，但扫描返回的数据本身是十几万或者上百万行，这个是符合预期的，需要从业务上判断是否真的需要返回那么多条数据；
2. 如果某类SQL出现慢SQL只是针对于特定分区键的值和区间，可以通过 `show info from T` 语句，确认是否有数据倾斜问题，是否可以调整分片策略。

```
mysql> show table info from test_tb;
+----+------------+-----------------+------------+
| ID | GROUP_NAME | TABLE_NAME      | SIZE_IN_MB |
+----+------------+-----------------+------------+
|  0 | ads_000000 | test_tb_o2ud_00 | 0.01562500 |
|  1 | ads_000000 | test_tb_o2ud_01 | 0.01562500 |
|  2 | ads_000000 | test_tb_o2ud_02 | 0.01562500 |
|  3 | ads_000000 | test_tb_o2ud_03 | 0.01562500 |
|  4 | ads_000000 | test_tb_o2ud_04 | 0.01562500 |
+----+------------+-----------------+------------+
```

从系统角度分析

1. 慢SQL出现时会伴随着业务流量上涨，可以先去控制台查看对应时间点的计算和存储上的资源监控，判断是否存在资源被打满，如果是由于业务流量上涨，导致资源被打满，出现慢sql是合理现象，可以考虑做资源扩容或者采取SQL限流。
2. 如果从慢日志查看不到慢SQL，但是从业务上确实受到慢SQL影响，且已经排除掉了是业务自身问题，这个时候可以查看是否是网络问题，确认下客户端和服务端是否处于一个VPC网络或者通过抓包进一步分析。

从执行角度分析

PolarDB-X提供了丰富的explain指令，来排查是否是查询本身导致的慢SQL。

- 查看逻辑执行计划

  执行 `explain` 语句，查看逻辑执行计划。

```
mysql> explain select nation, o_year, sum(amount) as sum_profit from ( select n_name as nation, extract(ye
ar from o_orderdate) as o_year, l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
from part, supplier, lineitem, partsupp, orders, nation where s_suppkey = l_suppkey and ps_suppkey = l_sup
pkey and ps_partkey = l_partkey and p_partkey = l_partkey and o_orderkey = l_orderkey and s_nationkey = n_
nationkey and p_name like '%yellow%' ) as profit group by nation, o_year order by nation, o_year desc limi
t 1;
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+
| Limit(offset=0, fetch=?2)
|
|   SortAgg(group="nation,o_year", $f2="SUM(amount)")
|
|     Project(nation="N_NAME", o_year="EXTRACT", amount="__*__ - PS_SUPPLYCOST * L_QUANTITY")
|
|       HashJoin(condition="PS_PARTKEY = L_PARTKEY AND P_PARTKEY = L_PARTKEY AND PS_SUPPKEY = L_SUPPKEY AN
D PS_SUPPKEY = S_SUPPKEY", type="inner")
|
|         MemSort(sort="N_NAME ASC,EXTRACT DESC")
|
|           BKAJoin(condition="O_ORDERKEY = L_ORDERKEY", type="inner")
|
|             Project(S_SUPPKEY="S_SUPPKEY", S_NATIONKEY="S_NATIONKEY", N_NATIONKEY="N_NATIONKEY", N_NAME=
"N_NAME", L_ORDERKEY="L_ORDERKEY", L_PARTKEY="L_PARTKEY", L_SUPPKEY="L_SUPPKEY", L_QUANTITY="L_QUANTITY",
__*__="__*__")
|
|               HashJoin(condition="L_SUPPKEY = S_SUPPKEY", type="inner")
|
|                 Gather(concurrent=true)
|
|                   LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_O
RDERKEY`, `L_PARTKEY`, `L_SUPPKEY`, `L_QUANTITY`, (`L_EXTENDEDPRICE` * (? - `L_DISCOUNT`)) AS `__*__` FROM
`lineitem` AS `lineitem`")
|
|                 BKAJoin(condition="N_NATIONKEY = S_NATIONKEY", type="inner")
|
|                   Gather(concurrent=true)
|
|                     LogicalView(tables="[000000-000003].supplier_[00-15]", shardCount=16, sql="SELECT `S
_SUPPKEY`, `S_NATIONKEY` FROM `supplier` AS `supplier`")
|
|                   Gather(concurrent=true)
|
|                     LogicalView(tables="[000000-000003].nation_[00-15]", shardCount=16, sql="SELECT `N_N
```

```
|                    LogicalView(tables="[000000-000003].nation_[00-15]", shardCount=16, sql="SELECT `N_N
ATIONKEY`, `N_NAME` FROM `nation` AS `nation` WHERE (`N_NATIONKEY` IN (...))")
|
|             Gather(concurrent=true)
|
|               LogicalView(tables="[000000-000003].orders_[00-15]", shardCount=16, sql="SELECT `O_ORDERKE
Y`, EXTRACT(YEAR FROM `O_ORDERDATE`) AS `EXTRACT` FROM `orders` AS `orders` WHERE (`O_ORDERKEY` IN (...))"
)
|
|         Gather(concurrent=true)
|
|           LogicalView(tables="[000000-000003].part_[00-15],partsupp_[00-15]", shardCount=16, sql="SELECT
`part`.`P_PARTKEY`, `partsupp`.`PS_PARTKEY`, `partsupp`.`PS_SUPPKEY`, `partsupp`.`PS_SUPPLYCOST` FROM `par
t` AS `part` INNER JOIN `partsupp` AS `partsupp` ON ((`part`.`P_PARTKEY` = `partsupp`.`PS_PARTKEY`) AND (`
part`.`P_NAME` LIKE ?)) WHERE (`partsupp`.`PS_PARTKEY` = `part`.`P_PARTKEY`)") |
| HitCache:false
|
| Source:PLAN_CACHE
```

> **说明** LogicalView里的shardCount参数表示需访问的分表总数。

根据执行计划分析SQL执行过程是否符合预期，主要关注全局二级索引、聚合优化和执行和JOIN优化和执行算法选择是否合理，执行计划的优劣直接影响到执行快慢。如果确认是优化过程中的执行计划不合理的话，可以按照下面三种方式去解决：

i. 执行计划的生成是基于统计信息的，如果统计信息缺失和过期，有可能导致优化出来的执行计划不是最佳的。可以通过 `analyze table` 语句手动触发统计信息收集。

```
mysql> show statistic;  //可以查看统计信息是否被正确收集
+---------------+------------+-------------+-------------+
| table_name    | table_rows | column_name | cardinality |
+---------------+------------+-------------+-------------+
| tpch_orders   |       8400 | NULL        |        NULL |
| tpch_lineitem |      10000 | NULL        |        NULL |
| lineitem      |       8492 | NULL        |        NULL |
| tpch_part     |       8054 | NULL        |        NULL |
| aa_lineitem   |       8490 | l_id        |        9143 |
| part          |       8054 | NULL        |        NULL |
| not_rt        |       8446 | NULL        |        NULL |
| aa_orders     |       8308 | o_id        |        9095 |
| partsupp      |       8609 | NULL        |        NULL |
| aa_partsupp   |       8667 | ps_id       |        9424 |
| tpch_partsupp |       8596 | NULL        |        NULL |
| tpch_customer |       8572 | NULL        |        NULL |
| orders        |       8423 | NULL        |        NULL |
| aa_customer   |       9080 | c_id        |        9476 |
| customer      |       8472 | NULL        |        NULL |
+---------------+------------+-------------+-------------+
15 rows in set (0.01 sec)
mysql> analyze table lineitem; //指定表名手动收集统计信息
+--------------+---------+----------+----------+
| TABLE        | OP      | MSG_TYPE | MSG_TEXT |
+--------------+---------+----------+----------+
| ads.lineitem | analyze | status   | OK       |
+--------------+---------+----------+----------+
1 row in set (1.66 sec)
```

ii. 在统计信息合理的基础上，优化器生成的执行计划可能还不是最优的。因为优化器的优化过程实际上是利用动态规划算子寻找相对最优解的过程，存在一定的不确定性。所以可以通过HINT force index手动制定最佳执行计划。

iii. 分析执行计划的基础上，还可以通过增加全局二级索引达到优化执行计划的效果。可参考智能索引推荐。

- 查看物理执行计划

  执行 `explain execute` 语句，查看下推SQL在DN上执行的计划，主要用于确认在DN上的执行是否选择了合适的局部索引。同样可以通过 `force index` 指令强制指定索引，选择合适的局部索引可以加快下推SQL在DN上的扫描效率。

```
mysql> explain execute  select * from lineitem;
+----+-------------+----------+------------+------+---------------+-----+---------+-----+------+----------
+-------+
| id | select_type | table    | partitions | type | possible_keys | key | key_len | ref | rows | filtered
| Extra |
+----+-------------+----------+------------+------+---------------+-----+---------+-----+------+----------
+-------+
| 1  | SIMPLE      | lineitem | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1    | 100
| NULL  |
+----+-------------+----------+------------+------+---------------+-----+---------+-----+------+----------
+-------+
1 row in set (0.02 sec)
mysql> explain execute  select L_LINENUMBER from lineitem;
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref | rows | filt
ered | Extra       |
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------+
| 1  | SIMPLE      | lineitem | NULL       | index | NULL          | PRIMARY | 8       | NULL | 1    | 100
| Using index |
+----+-------------+----------+------------+-------+---------------+---------+---------+-----+------+-----
-----+-------------+
1 row in set (0.04 sec)
```

- 分析查询耗时

  如果一条SQL每次执行都很慢，我们可以通过 `explain analyze` 语句分析查询到底慢在哪个算子上。

```
mysql> explain analyze select L_SUPPKEY, count(*) from lineitem group by L_SUPPKEY;
+----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| HashAgg(group="L_SUPPKEY", count(*)="SUM(count(*))"): rowcount = 1000.0, cumulative cost = value = 2.486
3085E7, cpu = 108049.0, memory = 35480.0, io = 201.0, net = 4.75, actual time = 0.000 + 0.000, actual rowc
ount = 1000, actual memory = 0, instances = 1, id = 519
|
|   Gather(concurrent=true): rowcount = 1000.0, cumulative cost = value = 2.4860051E7, cpu = 105039.0, mem
ory = 11881.0, io = 201.0, net = 4.75, actual time = 0.000 + 0.000, actual rowcount = 0, actual memory = 0
, instances = 0, id = 517
|
|     LogicalView(tables="[000000-000003].lineitem_[00-15]", shardCount=16, sql="SELECT `L_SUPPKEY`, COUNT
(*) AS `count(*)` FROM `lineitem` AS `lineitem` GROUP BY `L_SUPPKEY`"): rowcount = 1000.0, cumulative cost
= value = 2.486005E7, cpu = 105038.0, memory = 11881.0, io = 201.0, net = 4.75, actual time = 0.004 + 0.00
2, actual rowcount = 4892, actual memory = 0, instances = 0, id = 453 |
```

  参数说明：

  - rowcount：基于统计信息预估的算子输出行数
  - actual rowcount：真实执行过程中算子实际输出函数
  - actual time: 真实执行过程中算子实际耗时

  通过 `explain analyze` 语句可以准确分析出查询耗时的原因，从而有针对性的解决。如果是并发度不够，可以通过调整并发度加速查询。此外还可以通过trace指令，准确知道真实的下推物理SQL和耗时，明确耗时长是发生在CN层还是DN层。

```
mysql> trace select L_SUPPKEY, count(*) from lineitem group by L_SUPPKEY limit 1;
+-----------+----------+
```

```
| L_SUPPKEY | count(*) |
+-----------+----------+
|         1 |       12 |
+-----------+----------+
1 row in set (0.21 sec)
mysql> show trace;
+----+----------------+-----------+-------+-----------------+-----------------------------------------
---------------------------------+--------------+-------------------------+--------------------+--------
-------------+------+-----------------------------------------------------------------------------------
-----------------------------------------+----------------------+
| ID | NODE_IP        | TIMESTAMP | TYPE  | GROUP_NAME       | DBKEY_NAME
| TIME_COST(MS) | CONNECTION_TIME_COST(MS) | TOTAL_TIME_COST(MS) | CLOSE_TIME_COST(MS) | ROWS | STATEMENT
| PARAMS               |
+----+----------------+-----------+-------+-----------------+-----------------------------------------
---------------------------------+--------------+-------------------------+--------------------+--------
-------------+------+-----------------------------------------------------------------------------------
-----------------------------------------+----------------------+
|  0 | 192.168.31.105 |     0.000 | Query | ADS_000003_GROUP | dskey_ads_000003_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000003 | 3            | 0.00                    | 3                  | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_13`] |
|  1 | 192.168.31.105 |    -0.000 | Query | ADS_000003_GROUP | dskey_ads_000003_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000003 | 4            | 0.00                    | 7                  | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_12`] |
|  2 | 192.168.31.105 |     0.000 | Query | ADS_000003_GROUP | dskey_ads_000003_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000003 | 2            | 0.00                    | 2                  | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_14`] |
|  3 | 192.168.31.105 |     0.000 | Query | ADS_000003_GROUP | dskey_ads_000003_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000003 | 3            | 0.00                    | 3                  | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_15`] |
|  4 | 192.168.31.105 |     0.000 | Query | ADS_000001_GROUP | dskey_ads_000001_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000001 | 5            | 0.00                    | 14                 | 2
|  746 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_04`] |
|  5 | 192.168.31.105 |     0.000 | Query | ADS_000001_GROUP | dskey_ads_000001_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000001 | 4            | 0.00                    | 15                 | 1
|  682 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_05`] |
|  6 | 192.168.31.105 |     0.000 | Query | ADS_000001_GROUP | dskey_ads_000001_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000001 | 9            | 0.00                    | 15                 | 1
|  516 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_06`] |
|  7 | 192.168.31.105 |     0.000 | Query | ADS_000001_GROUP | dskey_ads_000001_group#polardbx-storage-1-m
aster#127.0.0.1-3306#ads_000001 | 7            | 0.00                    | 15                 | 0
|  310 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_07`] |
|  8 | 192.168.31.105 |     0.000 | Query | ADS_000000_GROUP | dskey_ads_000000_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000000 | 2            | 0.00                    | 3                  | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_00`] |
|  9 | 192.168.31.105 |     0.000 | Query | ADS_000000_GROUP | dskey_ads_000000_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000000 | 5            | 0.00                    | 7                  | 1
```

```
|  884 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_02`] |
| 10 | 192.168.31.105 |     0.000 | Query | ADS_000002_GROUP | dskey_ads_000002_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000002 | 2             | 0.00                      | 5                    | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_08`] |
| 11 | 192.168.31.105 |     0.000 | Query | ADS_000000_GROUP | dskey_ads_000000_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000000 | 6             | 0.00                      | 11                   | 0
|  917 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_01`] |
| 12 | 192.168.31.105 |     0.000 | Query | ADS_000000_GROUP | dskey_ads_000000_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000000 | 7             | 0.00                      | 10                   | 1
|  837 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_03`] |
| 13 | 192.168.31.105 |     0.000 | Query | ADS_000002_GROUP | dskey_ads_000002_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000002 | 3             | 0.00                      | 3                    | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_09`] |
| 14 | 192.168.31.105 |     0.000 | Query | ADS_000002_GROUP | dskey_ads_000002_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000002 | 2             | 0.00                      | 2                    | 0
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_10`] |
| 15 | 192.168.31.105 |     0.000 | Query | ADS_000002_GROUP | dskey_ads_000002_group#polardbx-storage-0-m
aster#127.0.0.1-3306#ads_000002 | 1             | 0.00                      | 3                    | 1
|    0 | /*DRDS /127.0.0.1/12f940ed62400000/0// */SELECT `L_SUPPKEY`, COUNT(*) AS `count(*)`
FROM ? AS `lineitem`
GROUP BY `L_SUPPKEY` | [`lineitem_MgSG_11`] |
+----+----------------+-----------+-------+------------------+-----------------------------------------
---------------------------------+---------------+--------------------------+---------------------+--------
-------------+------+-----------------------------------------------------------------------------------------
-----------------------------------------+----------------------+
16 rows in set (0.05 sec)
```

9.7. 如何限流慢SQL

本文介绍了如何对慢SQL进行有效限流。

在数据库会话或者慢日志中发现大量慢SQL，大量占用数据库资源，同时活跃会话数、CPU使用率、IOPS、内存使用率等监控指标一项或者多项处于高位。分析后发现这些慢SQL不属于核心业务，是优化不足的烂SQL，为保障核心业务的稳定运行，此时我们需要对其进行限流。

相关限流语法，请参见SQL限流。

SQL限流的运维操作步骤

1. 在实例会话页面或使用如下语句发现慢SQL。

```
select *
  from information_schema.processlist
 where COMMAND!= 'SLEEP'
   and TIME>= 1000
 order by TIME DESC;
```

2. 分析慢SQL，请参见如何分析及优化慢SQL。
3. 创建限流规则，可使用SQL命令，或者实例会话里SQL限流功能上的白屏化操作。
4. 从以下几方面观察限流规则效果。

- 监控指标恢复情况；
- 业务侧反馈；
- `show ccl_rules` 查看每个限流规则的限流情况的统计信息；
- 查看会话和SQL日志。

5. 创建索引、修改SQL、增加资源等。
6. 关闭限流规则，使用 `DROP CCL_RULE` 或者 `CLEAR CCL_RULES` 语句。

如下案例说明了如何对发现的慢SQL进行限流，您可以参照案例中的限流规则，修改后使用。

案例1：慢SQL属于同一个SQL模版

某DBA收到了数据库资源某指标处于高位的报警，查看数据库慢日志和会话后均发现有如下的慢SQL：

```
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------+-----------------+
| ID     | USER          | HOST               | DB                 | COMMAND | TIME | STATE | INFO
| SQL_TEMPLATE_ID |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------+-----------------+
| 951494 | userxxxxxxxxx | 222.0.0.1:33830    | analy_db           | Query   |   40 |       | select * from
bmsql_oorder where `o_id` > 12 | 65c92c88        |
| 952468 | userxxxxxxxxx | 222.0.0.1:33517    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where `o_id` > 10 | 65c92c88        |
| 953468 | userxxxxxxxxx | 222.0.0.1:33527    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where `o_id` > 23 | 65c92c88        |
| 954468 | userxxxxxxxxx | 222.0.0.1:33537    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where `o_id` > 25 | 65c92c88        |
| 955468 | userxxxxxxxxx | 222.0.0.1:33547    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where `o_id` > 27 | 65c92c88        |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------+-----------------+
```

可见，这些慢SQL属于同一个SQL模版（模版ID为65c92c88）：

```
select * from bmsql_oorder where `o_id` > ?
```

bmsql_oorder为一个数据量较大的表，而且列o_id上没有索引，显然这个一个未经优化的SQL，占尽了数据库资源影响了其他重要SQL的正常执行。这是一个非常适合利用模版ID去做SQL限流的场景。

创建限流规则

- 如果这个SQL模版的SQL不应该在当时执行，而且应该在业务低峰期执行，则我们可以创建SQL限流规则不让它执行：

```
CREATE CCL_RULE `KILL_CCL`       //限流规则名称为KILL_CCL
    ON `analy_db`.`*`              //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'         //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                     //&匹配是SELECT类型的SQL语句
  FILTER BY TEMPLATE '65c92c88'  //&匹配模版ID
  WITH MAX_CONCURRENCY = 0;      //设置单节点并发度为0，不允许匹配到的SQL执行
```

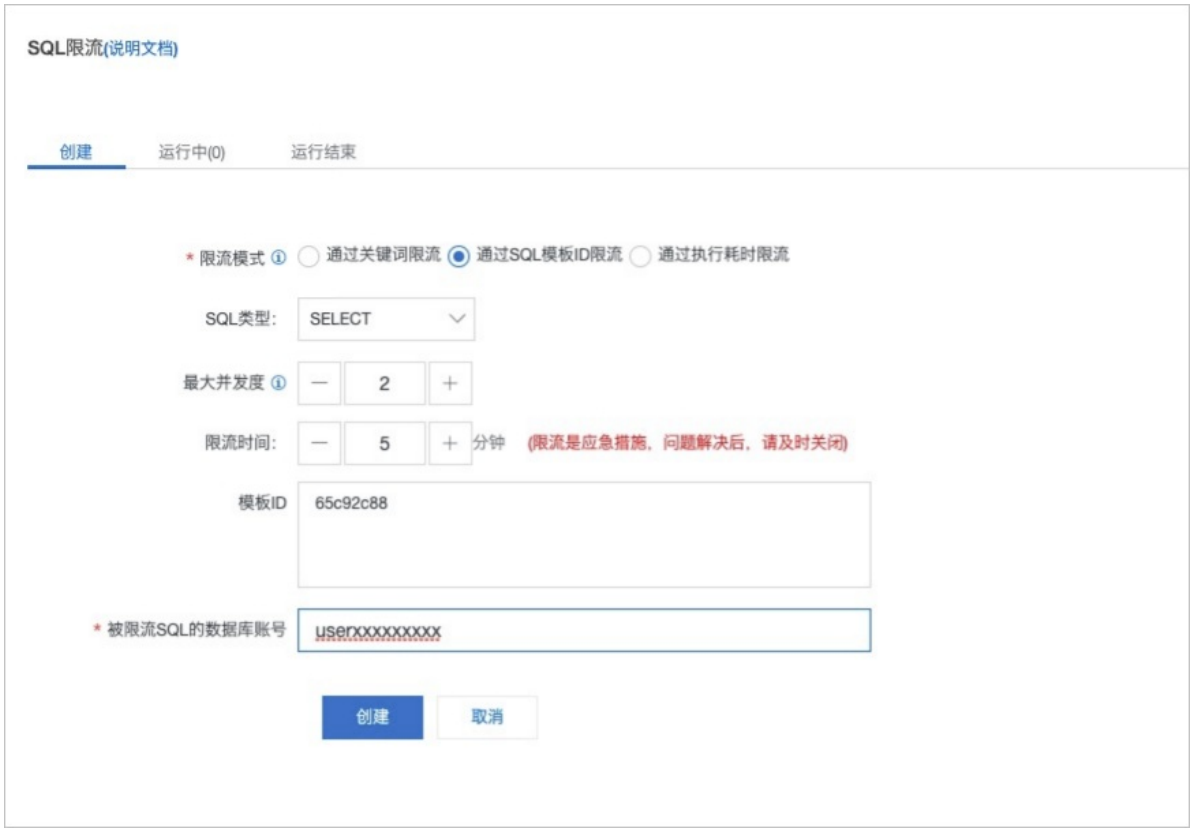
客户端再次执行这类SQL的时候将会返回报错信息：

```
ERROR 3009 (HY000): [13172dbaf2801000][10.93.159.222:3029][analy_db]Exceeding the max concurrency 0 per no
de of ccl rule KILL_CCL
```

- 如果允许这个SQL模版的SQL少量执行，只要不占尽数据库资源就行，则我们可以在创建限流规则的时候设置一定的并发度：

```
CREATE CCL_RULE `KILL_CCL_2`        //限流规则名称为KILL_CCL
    ON `analy_db`.`*`               //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'          //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                      //&匹配是SELECT类型的SQL语句
  FILTER BY TEMPLATE '65c92c88'  //&匹配模版ID 65c92c88
  WITH MAX_CONCURRENCY = 2;       //允许单个节点可以同时有两个这样的SQL在执行
```

也可使用实例会话页面里的SQL限流功能，进行如下操作：



- 如果希望这个SQL模版的SQL执行的时候可以慢，但尽量不要出错，则可以设置等待队列和等待超时时间（默认为600秒）：

```
CREATE CCL_RULE `QUEUE_CCL_2`        //限流规则名称为KILL_CCL
    ON `analy_db`.`*`               //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'          //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                      //&匹配是SELECT类型的SQL语句
  FILTER BY TEMPLATE '65c92c88'  //&匹配模版ID
  WITH MAX_CONCURRENCY = 2, WAIT_QUQUE_SIZE=20, WAIT_TIMEOUT=500; //单节点并发度为2，单节点等待队列长度为20，等待超时时间为500秒
```

创建完后，可以通过 `show ccl_rules` 指令查询各个限流规则的实际效果，比如当前匹配到某个限流规则的正在执行的SQL的数量、被限流报错的SQL数量、总匹配成功次数等。如果想放开被限流SQL，比如在增加了某个索引后，被限流SQL的执行效率变高了，则可以通过 `drop ccl_rule` 命令来关闭指定限流规则，或者使用 `clear ccl_rules` 来关闭所有的限流规则。

当然上面的SQL也可以通过关键字来限流，将SQL语句上的关键字做拆分，我们得到关键字列表：

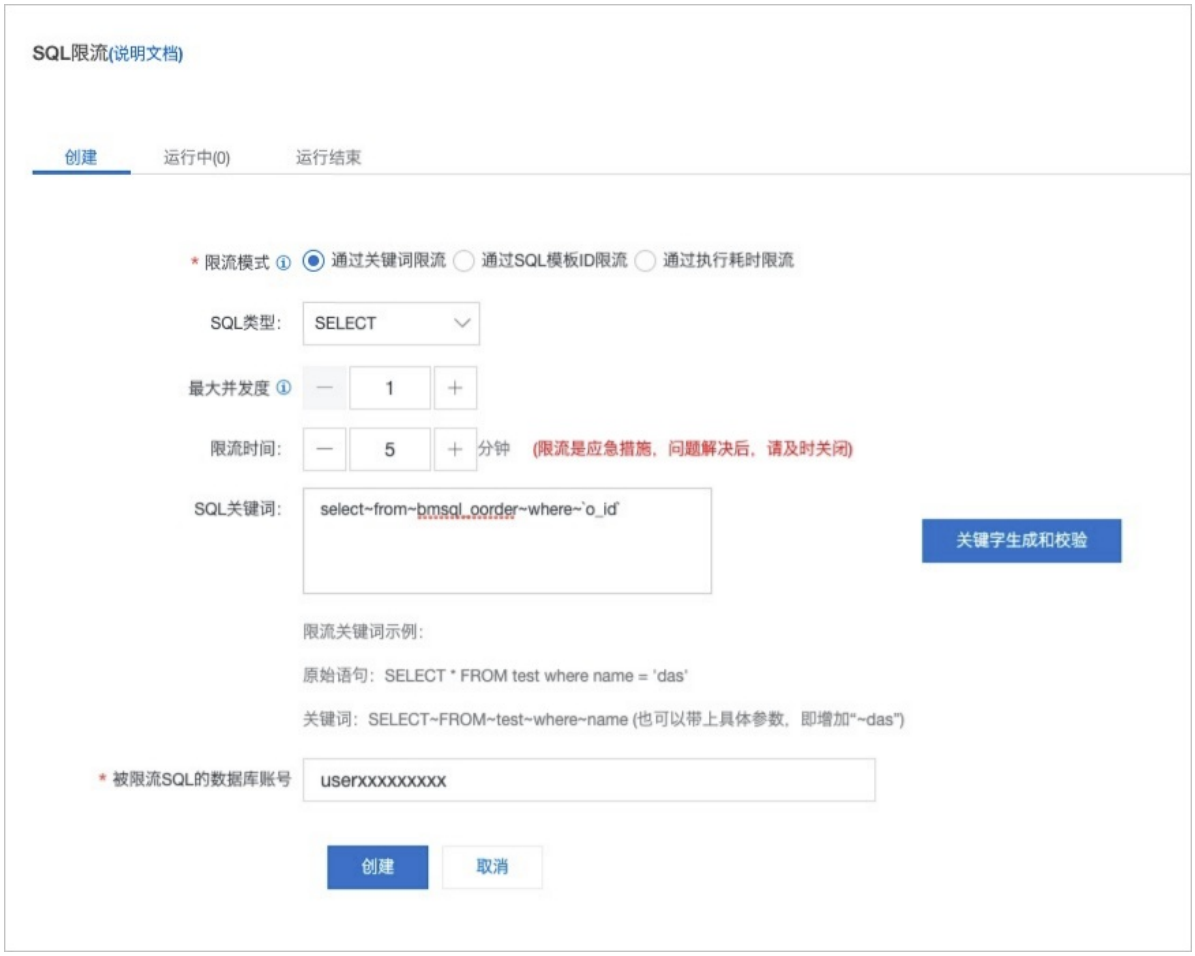
- select
- from
- bmsql_oorder
- where
- `o_id`

创建限流规则：

```
CREATE CCL_RULE `KILL_CCL`        //限流规则名称为KILL_CCL
    ON `analy_db`.`*`               //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'          //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                      //&匹配是SELECT类型的SQL语句
  FILTER BY KEYWORD('select','from','bmsql_oorder','where','`o_id`')  //&匹配模版ID
  WITH MAX_CONCURRENCY = 0;       //设置单节点并发度为0，不允许匹配到的SQL执行
```

在能获取到模版ID（在SQL日志、explain命令、会话中）的情况下，我们还推荐使用更精准的基于模版ID的限流。

也可使用实例会话页面里的SQL限流功能，操作如下：



案例2：慢SQL都是同一个SQL

某DBA收到了数据库资源某指标处于高位的报警，查看数据库慢日志和会话后均发现有如下的慢SQL：

```
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
| ID     | USER          | HOST               | DB                 | COMMAND | TIME | STATE | INFO
| SQL_TEMPLATE_ID |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
| 951494 | userxxxxxxxxx | 222.0.0.1:33830    | analy_db           | Query   |   40 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
| 952468 | userxxxxxxxxx | 222.0.0.1:33517    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
| 953468 | userxxxxxxxxx | 222.0.0.1:33527    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
| 954468 | userxxxxxxxxx | 222.0.0.1:33537    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
| 955468 | userxxxxxxxxx | 222.0.0.1:33547    | analy_db           | Query   |   43 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
```

bmsql_oorder中的符合o_carrier_id = 2条件的有较多记录，导致了慢SQL，如果使用模版ID限流，则会影响o_carrier_id不是2的SQL语句，如果使用关键字限流则会影响类似如下的正常SQL:

```
select * from bmsql_oorder where o_carrier_id = 2 limit 1;
select * from bmsql_oorder where o_carrier_id = 2 and o_c_id = 1;
```

限流具体的SQL，可以使用模版ID加关键字的方法，创建如下限流规则：

```
CREATE CCL_RULE `KILL_CCL`       //限流规则名称为KILL_CCL
    ON `analy_db`.`*`              //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'         //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                     //&匹配是SELECT类型的SQL语句
  FILTER BY TEMPLATE '438b00e4'  //&匹配模版ID 438b00e4
  FILTER BY KEYWORD('o_carrier_id','2')  //&匹配参数关键字
  WITH MAX_CONCURRENCY = 0;      //设置单节点并发度为0，不允许匹配到的SQL执行
```

如果使用PolarDB-X的CN内核版本为5.4.11以上，且该SQL不在prepare模式下执行，还可以使用如下高阶语法进行限流：

```
CREATE CCL_RULE `KILL_CCL`       //限流规则名称为KILL_CCL
    ON `analy_db`.`*`              //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'         //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                     //&匹配是SELECT类型的SQL语句
  FILTER BY QUERY 'select * from bmsql_oorder where o_carrier_id = 2'  //&匹配SQL语句
  WITH MAX_CONCURRENCY = 0;      //设置单节点并发度为0，不允许匹配到的SQL执行
```

案例3: 慢SQL集包含多个SQL模版

某DBA收到了数据库资源某指标处于高位的报警，查看数据库慢日志和会话后均发现有如下的慢SQL:

```
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
| ID     | USER          | HOST               | DB                 | COMMAND | TIME | STATE | INFO
| SQL_TEMPLATE_ID |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
| 951494 | userxxxxxxxxx | 222.0.0.1:33830    | analy_db           | Query   |   40 |       | select * from
bmsql_oorder where o_carrier_id = 2 | 438b00e4        |
| 952468 | userxxxxxxxxx | 222.0.0.1:33517    | analy_db           | Query   |   43 |       | select * from
bmsql_order_line where ol_o_id = 2  | 57a572f9        |
| 953468 | userxxxxxxxxx | 222.0.0.1:33527    | analy_db           | Query   |   43 |       | select * from
bmsql_new_order where no_w_id = 2   | de6eefdb        |
+--------+---------------+--------------------+--------------------+---------+------+-------+--------------
--------------------------------------+-----------------+
```

此种情况较为复杂，如果一条明显执行效率很高的SQL也成了慢SQL，则不排除是由于网络抖动或者服务节点异常等原因导致运行效率降低从而产生大批量的慢SQL，也可能是由于真正的烂SQL完全耗尽了资源，导致原本正常的SQL也成了慢SQL，需要通过SQL分析具体原因，不在本文的讨论范围内。假设已经确定了需要限流的慢SQL，我们则可以针对每个模版ID创建一个限流规则。但随着限流规则增加，匹配效率会略有降低，当PolarDB-X的CN内核版本为5.4.11以上时，我们推荐使用多模版限流：

```
CREATE CCL_RULE `KILL_CCL`       //限流规则名称为KILL_CCL
    ON `analy_db`.`*`              //&匹配analy_db下的所有表上执行的SQL
  TO 'userxxxxxxxxx'@'%'         //&匹配来自userxxxxxxxxx用户的SQL
    FOR SELECT                     //&匹配是SELECT类型的SQL语句
  FILTER BY TEMPLATE('438b00e4','57a572f9','de6eefdb')  //&匹配中其中一个模版ID，则该匹配项算匹配成功
  WITH MAX_CONCURRENCY = 0;      //设置单节点并发度为0，不允许匹配到的SQL执行
```

如果确定会话中的慢SQL是都是需要限流的烂SQL，且PolarDB-X的CN内核版本为5.4.11以上时，可以开启慢SQL限流。也可以使用实例会话页面里的SQL限流功能，进行如下操作：

总结

SQL限流为应急措施，可在数据库由于烂SQL导致效率降低的时候，起到快速恢复的作用。对烂SQL进行限流后，用户需要将注意力集中在如何优化烂SQL上，并在合适的时机清空SQL限流规则。

9.8. 如何异步删除大表

本文介绍了异步删除大表的方法。

背景信息

PolarDB-X底层存储节点，默认使用InnoDB引擎时，直接删除大表后会触发表文件的删除，导致POSIX文件系统出现严重的稳定性问题，因此InnoDB会启动一个后台线程来异步清理数据文件。当删除单个表空间时，会将对应的数据文件先重命名为临时文件，然后清除线程将异步、缓慢地清理文件。

> **说明** 提供清除文件日志来保证DDL语句的原子性。

操作步骤

1. 使用如下命令查看实例全局变量设置：

```
SHOW GLOBAL VARIABLES LIKE '%data_file_purge%';
```

返回结果如下：

```
+----------------------------------------+-------+
| Variable_name                          | Value |
+----------------------------------------+-------+
| innodb_data_file_purge                 | ON    |
| innodb_data_file_purge_all_at_shutdown | OFF   |
| innodb_data_file_purge_dir             |       |
| innodb_data_file_purge_immediate       | OFF   |
| innodb_data_file_purge_interval        | 100   |
| innodb_data_file_purge_max_size        | 128   |
| innodb_print_data_file_purge_process   | OFF   |
+----------------------------------------+-------+
```

参数说明如下：

参数	说明
innodb_data_file_purge	是否启用异步清除策略。
innodb_data_file_purge_all_at_shutdown	正常关机时全部清理。
innodb_data_file_purge_dir	临时文件目录。
innodb_data_file_purge_immediate	取消数据文件的链接但不清理。
innodb_data_file_purge_interval	清理时间间隔。单位：ms。
innodb_data_file_purge_max_size	每次清理单个文件大小的最大值。单位：MB。
innodb_print_data_file_purge_process	是否打印文件清理工作进程。

可以使用如下命令设置参数：

```
set global INNODB_DATA_FILE_PURGE = on;
set global INNODB_DATA_FILE_PURGE_INTERVAL = 100;
set global INNODB_DATA_FILE_PURGE_MAX_SIZE = 128;
```

> **说明** 默认情况下PolarDB-X并不支持直接使用set global指令设置参数，建议在控制台上进行的存储层参数设置。

2. 使用如下命令查看清理进度：

```
select * from information_schema.innodb_purge_files;
```

返回结果如下：

```
+--------+---------------------+--------------------+--------------+------------------------+---------
-----+
| log_id | start_time          | original_path      | original_size | temporary_path          | current_
size |
+--------+---------------------+--------------------+--------------+------------------------+---------
-----+
|      0 | 2021-05-14 14:40:01 | ./file_purge/t.ibd |     146800640 | ./#FP_210514 14:40:01_9 |     7969
1776 |
+--------+---------------------+--------------------+--------------+------------------------+---------
-----+
```

参数说明如下：

参数	说明
start_time	清理操作的开始时间。
original_path	表数据文件的原始路径。
original_size	表数据文件的原始大小，单位：byte。
temporary_path	清理中的临时文件路径。
current_size	待清理的剩余临时文件大小，单位：byte。

9.9. 如何支持热点更新场景

背景介绍

数据库中更新的模式为lock -> update -> unlock，当对数据库中的同一条记录有大量修改请求时，会造成大量的锁争抢与锁等待。请求量增加会导致TPS下降，延迟飙升。例如，秒杀场景中对于商品库存的扣减。

您可以选择在数据库内核中进行批处理，即对该条记录进行的更新操作使用组提交，更新的模式更改为lock-> group update -> unlock，从而减少锁争抢。结合流水线处理等优化，可以大大提高该场景的TPS，详情可见测试结果。

使用方法

1. 开启hotspot相关功能。在控制台的参数设置页面打开以下两个开关项，或者使用 `set global` 命令（非必须情况不建议使用set global指令）。

```
set global hotspot=on;
set global hotspot_lock_type=on
```

2. 切换事务类型为XA，并在业务的update语句中添加inventory hint。

```
BEGIN;
SET drds_transaction_policy=xa; // 若默认事务策略是XA，则无须重复设置，否则每次均需设置事务策略为XA以使用热点更新hint
UPDATE /*+ commit_on_success rollback_on_fail target_affect_row(number)*/ table_reference
    SET assignment_list
    [WHERE where_condition];
COMMIT | ROLLBACK; #取决于更新成功还是失败
```

注意事项

- where条件应为主键或唯一键的等值条件，且不支持带有全局索引的表（可包含本地索引）。
- 若已开启共享read_view，则应当先将共享read_view进行关闭，而后使用热点更新的能力。
- 热点更新hint的使用场景为单分片事务，无法在跨库场景中使用。
- 用户需使用COMMIT | ROLLBACK以提交或回滚事务。

Inventory Hint各参数含义

- commit_on_success（必选）

  如果该语句成功，则进行提交，连同该语句之前的未提交语句一并提交。

- rollback_on_fail

  如果该语句失败，则进行回滚，连同该语句之前的未提交语句一并回滚。

- target_affect_row(number)

  校验更新的行数是否符合预期，若不符合则更新失败。

示例

- 添加commit_on_success以使用组提交等针对热点更新场景的优化（id为主键，使用如下语句对id=1的记录进行更新时，若更新成功则自动提交）

```
BEGIN;
SET drds_transaction_policy=xa;
UPDATE /*+ commit_on_success*/ table_test SET c = c - 1 WHERE id = 1;
COMMIT | ROLLBACK;
```

- 使用rollback_on_fail，可使得更新失败时自动进行回滚

```
BEGIN;
SET drds_transaction_policy=xa;
UPDATE /*+ commit_on_success rollback_on_fail*/ table_test SET c = c - 1 WHERE id = 1;
COMMIT | ROLLBACK;
```

- 使用target_affect_row(number)，使得该update语句的预期更新行数为number，若不为number，则更新失败

```
BEGIN;
SET drds_transaction_policy=xa;
UPDATE /*+ commit_on_success rollback_on_fail target_affect_row(1)*/ table_test SET c = c - 1 WHERE id = 1
;
COMMIT | ROLLBACK;
```

- 在带有热点更新hint的update语句前，可对同一个物理库中的表进行DML操作

```
BEGIN;
SET drds_transaction_policy=xa;
INSERT into table_test_2 values (1,1);
UPDATE /*+ commit_on_success rollback_on_fail target_affect_row(1)*/ table_test SET c = c - 1 WHERE id = 1
;
COMMIT | ROLLBACK;
```

查看inventory hint是否生效

- 使用命令 `show global status like "%Group_update%"` 查看组提交状态，Group_update_leader_count一直增加则说明触发了热点组提交的优化逻辑。

```
mysql> show global status like "%Group_update%";
+---------------------------------------+--------+
| Variable_name                         | Value  |
+---------------------------------------+--------+
| Group_update_fail_count               | 54     |
| Group_update_follower_count           | 962869 |
| Group_update_free_count               | 2      |
| Group_update_group_same_count         | 0      |
| Group_update_gu_leak_count            | 0      |
| Group_update_ignore_count             | 0      |
| Group_update_insert_dup               | 0      |
| Group_update_leader_count             | 168292 |
| Group_update_lock_fail_count          | 0      |
| Group_update_mgr_recycle_queue_length | 0      |
| Group_update_recycle_queue_length     | 0      |
| Group_update_reuse_count              | 23329  |
| Group_update_total_count              | 2      |
+---------------------------------------+--------+
13 rows in set (0.01 sec)
```

- 使用 `show physical full processlist` 查看update的状态，是否出现hotspot字样

```
mysql> show physical full processlist where command != 'Sleep';
+-------+------+-----+---------+-------------+---------+------+-------------------------+-----------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------+
| Group | Atom | Id  | User    | db          | Command | Time | State                   | Info
|
+-------+------+-----+---------+-------------+---------+------+-------------------------+-----------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------+
|     0 |    0 |  56 | diamond | test_000001 | Query   |    0 | hotspot wait for commit | /*DRDS /127.0.0.
1/12e774cab8800000-128/0// */UPDATE  /*+COMMIT_ON_SUCCESS ROLLBACK_ON_FAIL TARGET_AFFECT_ROW(1) */ `test_h
otline_lZTr` AS `test_hotline` SET `b` = (`b` + 1) WHERE (`a` = 1) |
|     0 |    0 | 822 | diamond | test_000001 | Query   |    0 | query end               | /*DRDS /127.0.0.
1/12e774c4e9400000-563/0// */UPDATE  /*+COMMIT_ON_SUCCESS ROLLBACK_ON_FAIL TARGET_AFFECT_ROW(1) */ `test_h
otline_lZTr` AS `test_hotline` SET `b` = (`b` + 1) WHERE (`a` = 1) |
|     0 |    0 | 831 | diamond | test_000001 | Query   |    0 | hotspot wait for commit | /*DRDS /127.0.0.
1/12e774c551000000-509/0// */UPDATE  /*+COMMIT_ON_SUCCESS ROLLBACK_ON_FAIL TARGET_AFFECT_ROW(1) */ `test_h
otline_lZTr` AS `test_hotline` SET `b` = (`b` + 1) WHERE (`a` = 1) |
|     0 |    0 | 838 | diamond | test_000000 | Query   |    0 | starting                | show full proces
slist
|
+-------+------+-----+---------+-------------+---------+------+-------------------------+-----------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------+
4 rows in set (0.33 sec)
```

热点更新测试

测试表定义

```
CREATE TABLE sbtest(id INT UNSIGNED NOT NULL PRIMARY KEY, c BIGINT UNSIGNED NOT NULL);
```

测试语句

```
UPDATE /*+ COMMIT_ON_SUCCESS ROLLBACK_ON_FAIL TARGET_AFFECT_ROW(1) */ sbtest SET c=c+1 WHERE id = 1;
```

测试工具

sysbench

机器规格

4C8G×2（两节点）

测试结果

场景	1线程	4线程	8线程	16线程	32线程	64线程	128线程	256线程	512线程
热点更新	298	986	1872	3472	6315	10138	13714	15803	23262
普通更新	318	423	409	409	412	428	448	497	615

- 以上结果的单位为TPS，即每秒处理的交易数（Transaction per second）
- 热点更新的TPS与机器规格、并发请求数、更新语句有关，测试结果仅供参考

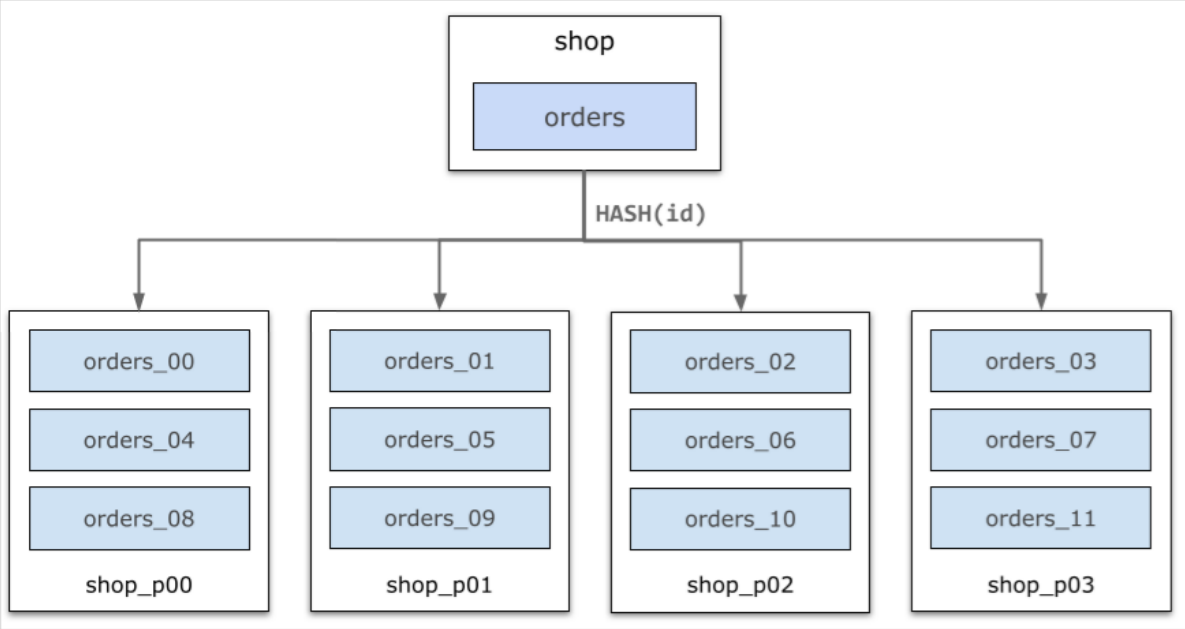
9.10. 如何分析数据分布不均衡

本文将介绍如何分析和处理数据倾斜的问题。

概述

PolarDB-X是由阿里巴巴自主研发的云原生分布式数据库，在物理资源上是由多个节点所组成的分布式集群。通过数据分区的方式，可以将数据分布到集群中的多个存储节点，发挥多个节点的存储和计算能力。

当存储节点的数据分布不均匀，大部分数据集中在一两个节点时，将导致节点负载过高、查询缓慢，甚至造成节点故障，这种现象称之为数据倾斜。这类问题无法通过扩容来解决，本文将介绍如何分析和处理数据倾斜的问题。



问题分析

数据倾斜问题可以按照分库级、分表级、分区级的思路由浅入深进行分析排查。

分库级数据倾斜

执行 `show db status` 语句，能够显示当前数据库中的所有物理库的数据大小，部分参数说明如下：

- PHYSICAL_DB：物理库名
- SIZE_IN_MB：数据大小
- RATIO：数据比例

示例：

```
MySQL polardbx_root@127.1:test_polarx> show db status;
+----+--------------------------+-------------------+--------------------------+------------+--------+---
-------------+
| ID | NAME                     | CONNECTION_STRING  | PHYSICAL_DB              | SIZE_IN_MB | RATIO  | TH
READ_RUNNING |
+----+--------------------------+-------------------+--------------------------+------------+--------+---
-------------+
| 1  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | TOTAL                    |  0.875     | 100%   | 1
|
| 2  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | hehe_000000              |  0.203125  | 23.21% |
|
| 3  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | hehe_000001              |  0.203125  | 23.21% |
|
| 4  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | hehe_000002              |  0.203125  | 23.21% |
|
| 5  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | hehe_000003              |  0.203125  | 23.21% |
|
| 6  | hehe@polardbx-polardbx    | 100.82.20.151:3306 | hehe_single              |  0.0625    | 7.14%  |
|
+----+--------------------------+-------------------+--------------------------+------------+--------+---
-------------+
6 rows in set
```

在数据倾斜的情况下，多个物理库的“SIZE_IN_MB"和"RATIO”会相差较大。对于其中数据量较多的分库，可以通过分表级的信息进一步分析。

分表级数据倾斜

执行 `show table status` 语句，查看当前库的所有数据表大小。部分参数说明如下：

- ROWS：近似的数据行数
- DATA_LENGTH: 近似的数据量

```
MySQL polardbx@127.1:test_polarx> show table status;
+----------+--------+---------+------------+------+----------------+-------------+-----------------+--------
------+-----------+----------------+---------------------+-------------+------------+--------------------+--
--------+-----------------+----------+
| NAME     | ENGINE | VERSION | ROW_FORMAT | ROWS | AVG_ROW_LENGTH | DATA_LENGTH | MAX_DATA_LENGTH | INDEX_L
ENGTH | DATA_FREE | AUTO_INCREMENT | CREATE_TIME         | UPDATE_TIME | CHECK_TIME | COLLATION          | C
HECKSUM | CREATE_OPTIONS | COMMENT |
+----------+--------+---------+------------+------+----------------+-------------+-----------------+--------
------+-----------+----------------+---------------------+-------------+------------+--------------------+--
--------+-----------------+----------+
| test_tb  | InnoDB | 10      | Dynamic    | 0    | 0              | 131072      | 0               | 131072
| 0         | 100000         | 2021-08-19 07:40:07 | <null>      | <null>     | utf8mb4_general_ci | <null>
|                |         |
| test_tb1 | InnoDB | 10      | Dynamic    | 0    | 0              | 65536       | 0               | 65536
| 0         | 100000         | 2021-08-19 07:52:24 | <null>      | <null>     | utf8mb4_general_ci | <null>
|                |         |
+----------+--------+---------+------------+------+----------------+-------------+-----------------+--------
------+-----------+----------------+---------------------+-------------+------------+--------------------+--
--------+-----------------+----------+
2 rows in set
```

执行 `show table info from $TABLE` 语句，查看分表级的数据大小，示例如下：

```
MySQL polardbx@127.1:test_polarx> show create table test_tb\G
***************************[ 1. row ]***************************
Table        | test_tb
Create Table | CREATE TABLE `test_tb` (
        `id` int(11) DEFAULT NULL,
        `c1` bigint(20) DEFAULT NULL,
        `c2` varchar(100) DEFAULT NULL,
        KEY `auto_shard_key_id` USING BTREE (`id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4  dbpartition by hash(`id`) tbpartition by hash(`id`) tbpartition
s 2
MySQL polardbx@127.1:test_polarx> show table info from test_tb;
+----+--------------------+----------------+------------+
| ID | GROUP_NAME         | TABLE_NAME     | SIZE_IN_MB |
+----+--------------------+----------------+------------+
| 0  | test_polarx_000000 | test_tb_hg6z_0 | 0.03125    |
| 1  | test_polarx_000000 | test_tb_hg6z_1 | 0.03125    |
| 2  | test_polarx_000001 | test_tb_hg6z_2 | 0.03125    |
| 3  | test_polarx_000001 | test_tb_hg6z_3 | 0.03125    |
| 4  | test_polarx_000002 | test_tb_hg6z_4 | 0.03125    |
| 5  | test_polarx_000002 | test_tb_hg6z_5 | 0.03125    |
| 6  | test_polarx_000003 | test_tb_hg6z_6 | 0.03125    |
| 7  | test_polarx_000003 | test_tb_hg6z_7 | 0.03125    |
+----+--------------------+----------------+------------+
8 rows in set
```

test_tb表的拆分是 `dbpartition by hash(id)` 和 `tbpartition by hash(id) tbpartitions 2` ，因此有4个分库，8个分表。以上的 `show table info from test_tb` 命令中， SIZE_IN_MB即每个分表的数据大小。

如果分表之间的数据容量相差较多，那么即发生了分表的数据倾斜，可能是由于tbpartition by的拆分不当导致的。

分区级数据倾斜

对于PolarDB-X的分区表来说，支持更灵活的数据拆分方式，即LIST/HASH/RANGE分区，以及灵活的分区分裂、合并、迁移。

对于分区表来说，同样支持通过 `show table info from $TABLE` 命令查询每个分表的物理大小。

除此之外，分区表还支持通过 `select * from information_schema.table_detail where logical_table='test_tb'` 查询分区级的详细信息，部分参数说明如下：

- PARTITION_NAME： 分区名
- TABLE_ROWS：分区的数据行数
- DATA_LENGTH： 分区的数据大小
- PERCENT ： 分区的数据比例

```
+-------------+-----------------+---------------+----------------+---------------+----------------+--------
----+-------------+--------------+-------------------------------------------------+-------------------------
---------+
| SCHEMA_NAME | TABLE_GROUP_NAME | LOGICAL_TABLE | PHYSICAL_TABLE | PARTITION_SEQ | PARTITION_NAME | TABLE_R
OWS | DATA_LENGTH | INDEX_LENGTH | BOUND_VALUE                                     | PERCENT
|
+-------------+-----------------+---------------+----------------+---------------+----------------+--------
----+-------------+--------------+-------------------------------------------------+-------------------------
---------+
| partdb_test | tg73             | test_tb       | test_tb_00000  | 0             | p1             | 0
| 16384       | 16384        | [MINVALUE, -6917529027641081843)             | 0.00%|
-+    |
| partdb_test | tg73             | test_tb       | test_tb_00001  | 1             | p2             | 1
| 16384       | 16384        | [-6917529027641081843, -4611686018427387893) | 9.09%|
--+  |
| partdb_test | tg73             | test_tb       | test_tb_00002  | 2             | p3             | 1
| 16384       | 16384        | [-4611686018427387893, -2305843009213693943) | 9.09%|
--+  |
| partdb_test | tg73             | test_tb       | test_tb_00003  | 3             | p4             | 0
| 16384       | 16384        | [-2305843009213693943, 7)                    | 0.00%|
-+    |
| partdb_test | tg73             | test_tb       | test_tb_00004  | 4             | p5             | 6
| 16384       | 16384        | [7, 2305843009213693957)                     | 54.55%|
---+ |
| partdb_test | tg73             | test_tb       | test_tb_00005  | 5             | p6             | 2
| 16384       | 16384        | [2305843009213693957, 4611686018427387907)   | 18.18%|
---+ |
| partdb_test | tg73             | test_tb       | test_tb_00006  | 6             | p7             | 1
| 16384       | 16384        | [4611686018427387907, 6917529027641081857)   | 9.09%|
--+  |
| partdb_test | tg73             | test_tb       | test_tb_00007  | 7             | p8             | 0
| 16384       | 16384        | [6917529027641081857, 9223372036854775807)   | 0.00%|
-+    |
+-------------+-----------------+---------------+----------------+---------------+----------------+--------
----+-------------+--------------+-------------------------------------------------+-------------------------
---------|
8 rows in set
```

在以上示例中，分区p5的数据量明显多于其他分区，存在数据倾斜。

解决方案

数据倾斜通常是由于数据拆分的方式不当造成的，常见原因如下：

- 使用了不恰当的拆分函数，例如UNI_HASH，但拆分键不具备均匀分布的特征；
- 拆分键的区分度过低，例如HASH分区，按照省份拆分，但省份实际较少，容易造成数据不均；
- 某些拆分键存在较多的数据，例如订单表按照卖家id进行拆分，部分的大卖家可能存在较多的数据。

拆分方式调整

对于拆分方式选择不当导致的数据倾斜问题，通常需要调整拆分方式，包括以下两方面：

- 调整拆分函数：分库分表可以选择HASH/UNI_HASH/STR_HASH等拆分函数；分区表可采用HASH/KEYS/RANGE/RANGE COLUMN等拆分方式；
- 调整拆分键：
  - 选择较为均匀，不存在热点的拆分键；
  - 选择区分度较高的拆分键，避免HASH结果不均匀；
  - 大部分查询都通过拆分键做等值查询，尽量避免查询多个分片。

在选择好数据拆分方式之后，可以通过如下方法对数据表进行调整：

- 重建表：重建另一个新的表，将旧表的数据导入。

> **说明** 此方法需要先停止业务写入。

- 在线调整分区：通过变更表类型及拆分规则在线修改分区方式；无需停止业务写入，但此过程仍然需要重写全表数据，开销较大，需要在业务低峰期执行。

示例：用户发现test_tb表存在数据倾斜，原因在于数据拆分键使用不当，因此可以通过以下语句将拆分键调整成hash(order_id)：

```
ALTER TABLE test_tb dbpartition BY hash(`order_id`);
```

分区调整

在PolarDB-X中，实现了更灵活的基于分区表的数据分布，因此可以实现分区级的分裂及迁移，解决数据倾斜问题。分区调整能够解决的场景主要是分区过大导致的数据倾斜，不适用于拆分函数选择不当等问题。

以Range分区举例：

1. 建表时指定两个分区，p0和p1，其范围分别是 [-inf, 1000), [1000, 2000);
2. 发现分区p0数据过多，存在数据倾斜，因此将分区p0进行分裂，使其分布到多个节点；
3. 默认新建的分区会创建到数据量最少的节点上，如果不满足需求，可另外进行分区迁移。

```
CREATE TABLE `table_range` (
        `id` int(11) DEFAULT NULL
) PARTITION BY RANGE(`id`)
(PARTITION p0 VALUES LESS THAN (1000),
 PARTITION p1 VALUES LESS THAN (2000)
) /* tablegroup = `tg110` */ ;
ALTER TABLEGROUP tg110 SPLIT PARTITION p0 INTO
(partition p0_1 values less than (500),
 partition p0_2 values less than (1000) );
```

> **注意** SPLIT PARTITION操作的对象是TABLEGROUP而非单个表，通过 `SHOW FULL CREATE TABLE` 命令可以查看一个表所属的tablegroup。

9.11. 如何高效使用IN查询

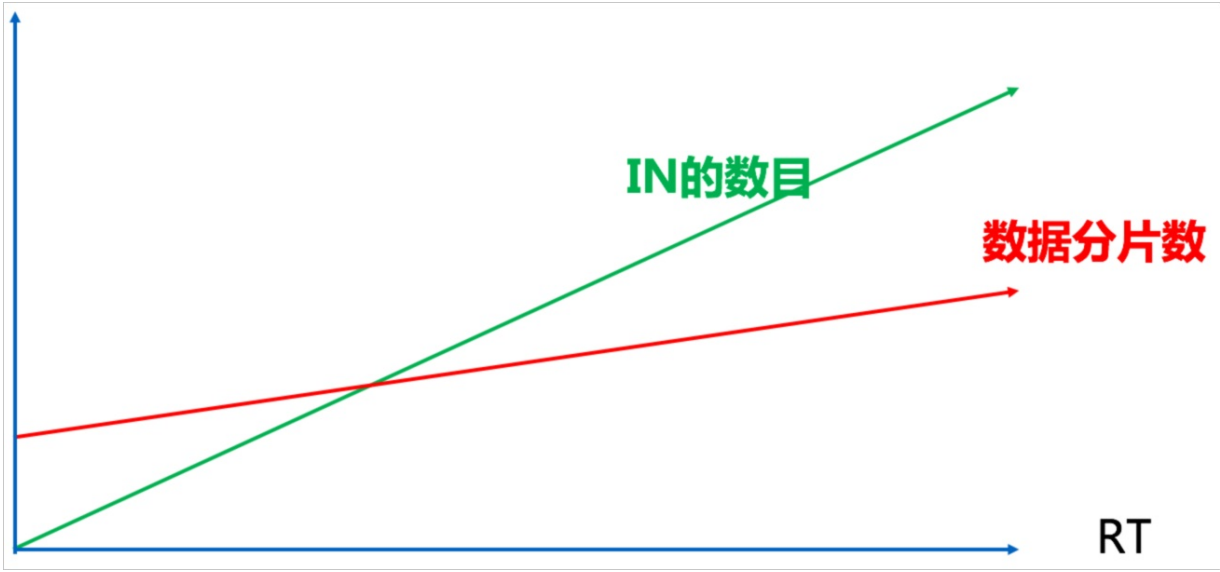
本文将介绍如何在PolarDB-X中做IN查询时，选择最佳的Values个数。

功能介绍

实际场景中经常需要根据一些常量指标做IN查询，其中IN的字段是分区键。例如在电商场景中，所有订单都会记录到订单表Order，此表按照订单ID进行拆分，一个买家经常会根据已购买的订单列表，查询这些订单的具体信息。假设用户已购买的订单数是2，那么会产生2个值的IN条件查询，理论上查询会路由到两个2分片。查询SQL示例：

```
SELECT * FROM ORDER WHERE ORDER_ID IN (id1,id2)
```

随着用户购买的订单数增加，查询订单信息的IN值数量也会增加，这样一次查询很可能会路由到所有的分片，导致RT变高。下图展示了IN值数量、扫描分片数和RT之间的关系。



为了尽可能避免随着IN值数量增加，导致物理SQL膨胀和扫描压力，PolarDB-X在内核版本5.4.8-16069335（包含）之后引入了基于IN值做动态分区裁剪的能力。

继续以上述场景为例假设Order表分片数量是128，IN查询的数量128个，那么一次查询的SQL为：

```
SELECT * FROM ORDER WHERE ORDER_ID IN (id1,id2,id3....id128)
```

如果ID足够离散，可能会分散到所有的分片，需要查询最多128个分片，每个分片的物理查询没有做IN值的裁剪，每个物理查询都会携带128个IN值条件下推给MySQL，过多的IN条件也会加大MySQL执行压力。查询示例如下：

```
SELECT * FROM ORDER WHERE ORDER_ID_1 IN (id1,id2,id3....id128);
SELECT * FROM ORDER WHERE ORDER_ID_2 IN (id1,id2,id3....id128);
SELECT * FROM ORDER WHERE ORDER_ID_3 IN (id1,id2,id3....id128);
.....
SELECT * FROM ORDER WHERE ORDER_ID_128 IN (id1,id2,id3....id128);
```

在支持IN分区裁剪的版本上，首先计算层会根据条件计算分片，引入IN值的动态分片裁剪，下发给MySQL的物理查询上就会只包含属于该分片的ID条件，裁剪掉了多余的IN值条件，因此IN查询的RT和吞吐都会有一定的提升。查询示例如下：

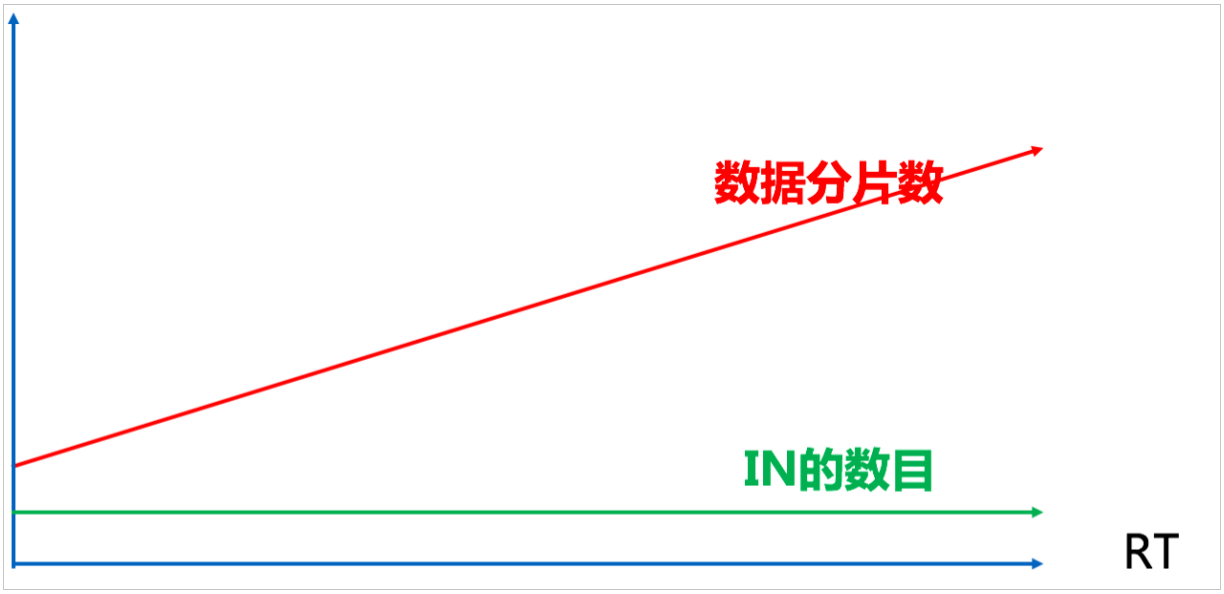
```
SELECT * FROM ORDER WHERE ORDER_ID_1 IN (id1);
SELECT * FROM ORDER WHERE ORDER_ID_2 IN (id2,id12);
SELECT * FROM ORDER WHERE ORDER_ID_3 IN (id3,id4,id5);
.....
SELECT * FROM ORDER WHERE ORDER_ID_32 IN (id100....id128);
```

另外，PolarDB-X内部针对跨分片的查询会有一个Parallel Query执行，例如涉及32个分片，针对每个用户查询，会有节点CPU内核数大小的并发度，例如分布式下单个节点规格为16core时，默认并发数就是16个，即32个分片会分成2批才能执行完成。

结合以往经验，在IN查询的业务中，阿里云建议：

- IN的值的数目远小于分片数，这样可以避免每次都做全分片查询；
- IN的值的数目不会随着业务的发展而增长，这样可以避免随着业务变化而导致性能下降；
- 兼顾RT和吞吐的话，建议IN的值的数量在8~32之间。

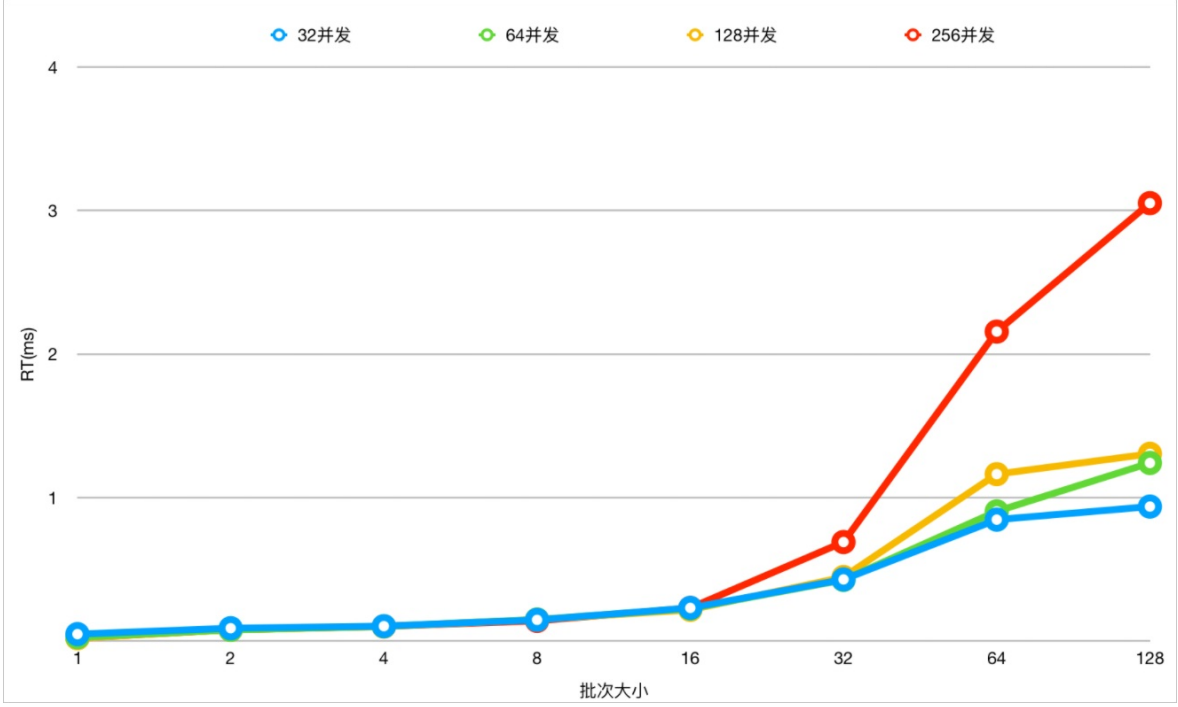
满足上述最佳经验后，涉及到IN查询的业务可以做到面向并发场景下的线性扩展，而RT也不会有明显抖动。线性扩展举例：例如分布式16core能跑1万个IN并发，扩展到32core之后就能跑2万个并发。



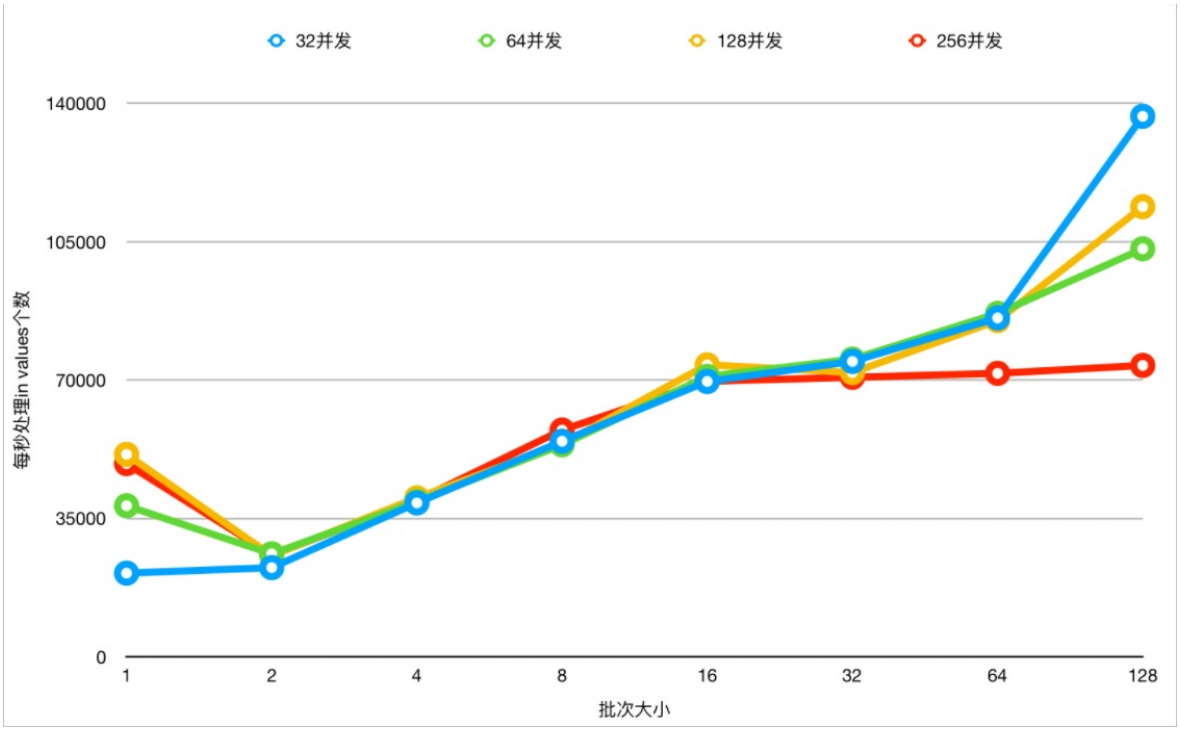
比对测试

在兼顾RT和吞吐的场景下，确定合理的IN查询的值的数量。在规格2×16C64G的节点，针对一张分表数量为64，分表记录数为百万级别的表在不同值数量、不同并发下做测试。在内核版本5.4.8-16069335（包含）之后针对IN查询进一步完善了动态裁剪分表的能力，下发的物理SQL也会裁剪掉多余的Values，下面是比对测试的结果。

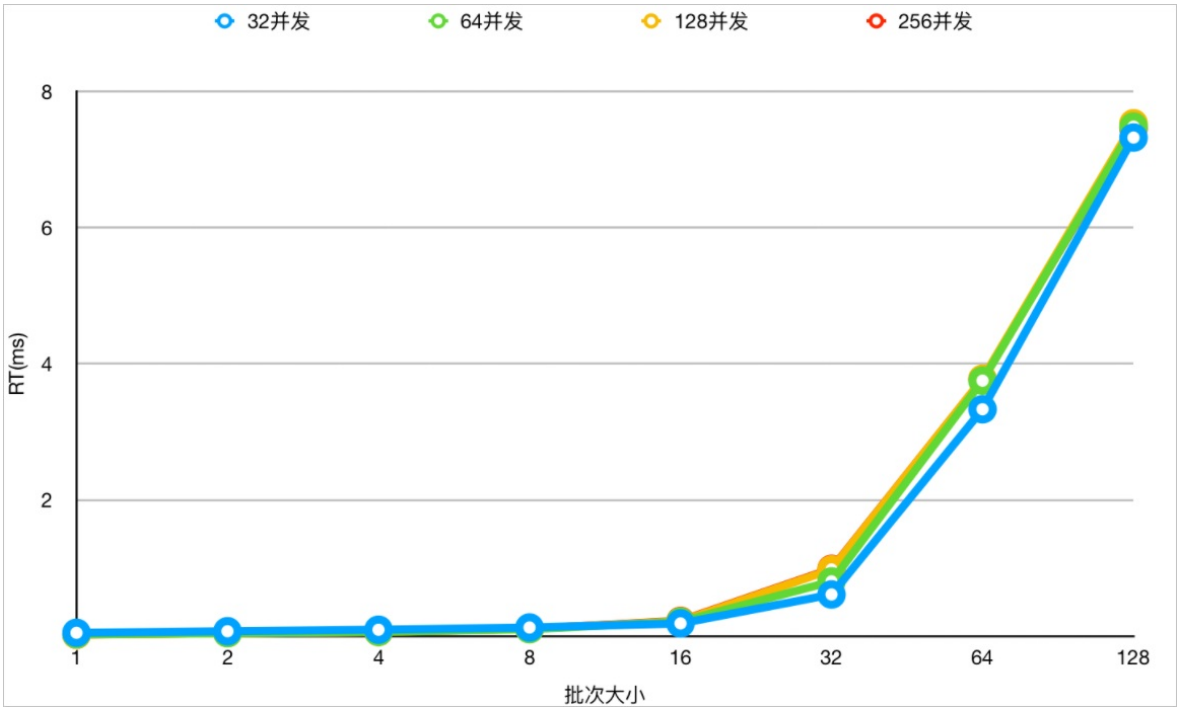
1. 在不同并发下，不同Values值数量下测试，开启IN查询动态裁剪能力下，查看RT变化。



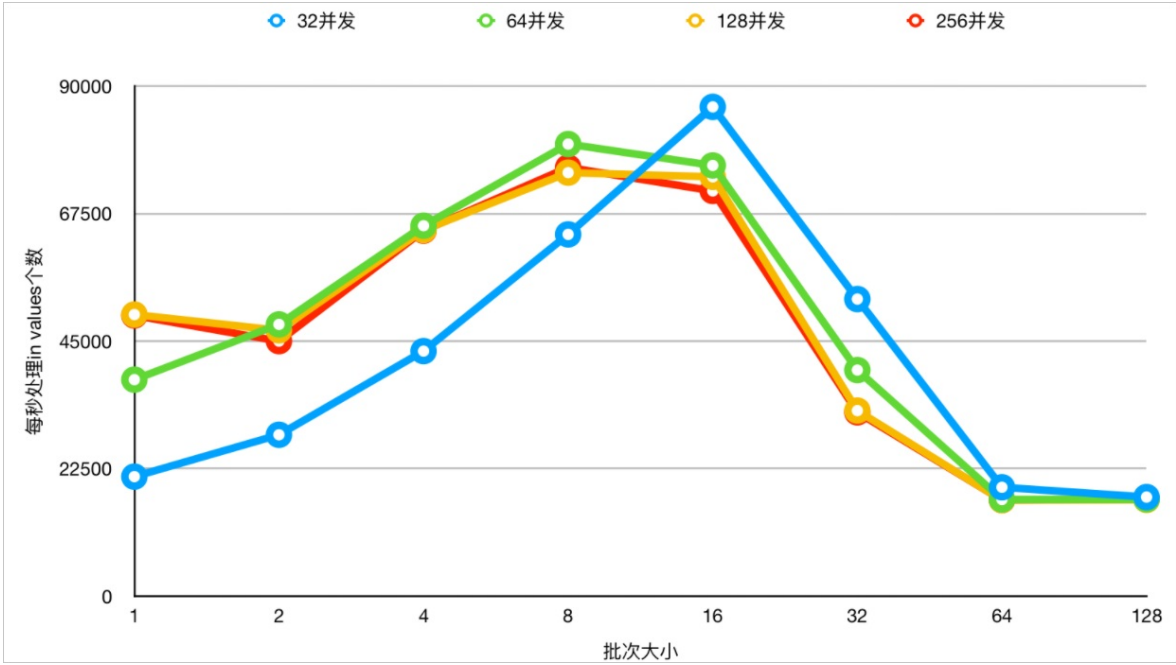
2. 在不同并发下，不同Values值数量下测试，开启IN查询动态裁剪能力下，查看吞吐变化。



3. 在不同并发下，不同Values值数量下测试，关闭IN查询动态裁剪能力下，查看RT变化。



4. 在不同并发下，不同Values值数量下测试，关闭IN查询动态裁剪能力下，查看吞吐变化。



通过测试对比，可以得到以下结论：

- 兼顾RT和吞吐时，建议IN的值的数量在8~32之间，基本对齐分布式Parallel Query的默认并发度（单节点的CPU内核数）。
- 在内核版本5.4.8-16069335（包含）之后，在开启IN查询的动态裁剪能力下，吞吐和RT都有明显的优势，推荐您将内核版本升级至5.4.8及以上版本。

9.12. 如何优化Batch Insert

Batch Insert语句是常见的数据库写入数据的方式，PolarDB-X兼容MySQL协议和语法，Batch Insert语法为：

```
INSERT [IGNORE] [INTO] table_name(column_name, ...) VALUES (value1, ...), (value2, ...), ...;
```

影响Batch Insert性能的主要因素包括：

1. batch size
2. 并行度
3. 分片数目
4. 列数目
5. GSI的数目
6. sequence数目

对于分片数目、列数目、GSI数目、sequence数目等内需因素，根据实际需求进行设置，并且常常会和读性能相互影响，例如GSI数目较多情况下，写入性能肯定会下降，但是对读性能有提升。本文不详细讨论这些因素的影响，主要聚焦于batch size和并行度的合理设置。

测试环境

本文档的测试环境见下表：

环境	参数
PolarDB-X版本	polarx-kernel_5.4.11-16279028_xcluster-20210802
节点规格	16核64GB
节点个数	4

测试的表用例：

```
CREATE TABLE `sbtest1` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    `c` char(120) NOT NULL DEFAULT '',
    `pad` char(60) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`),
    KEY `k_1` (`k`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4;
```

Batch特性：BATCH_INSERT_POLICY=SPLIT

PolarDB-X针对数据批量写入，为保障更好的并发性，对Batch Insert进行了优化，当单个Batch Insert语句大小超过256K时，PolarDB-X会将Batch Insert语句动态拆分成多个小Batch，多个小Batch之间串行执行，这个特性称为SPLIT。

通过BATCH_INSERT_POLICY=SPLIT的机制，在保障最佳性能的同时，减少PolarDB-X并行执行Batch Insert的代价，尽可能规避分布式下多节点的负载不均衡。

相关参数：

1. BATCH_INSERT_POLICY，可选SPLIT/NONE，默认值为SPLIT，代表默认启用动态拆分Batch。
2. MAX_BATCH_INSERT_SQL_LENGTH，默认值256，单位KB。代表触发动态拆分Batch的SQL长度阈值为256K。
3. BATCH_INSERT_CHUNK_SIZE_DEFAULT，默认值200。代表触发动态拆分Batch时，每个拆分之后的小Batch的批次大小。

关闭BATCH_INSERT_POLICY=SPLIT机制，可通过如下hint语句 `/*+TDDL:CMD_EXTRA(BATCH_INSERT_POLICY=NONE)*/` 。此参数的目标是关闭BATCH_INSERT_POLICY策略，这样才可以保证batch size在PolarDB-X执行时不做自动拆分，可用于验证batch size为2000、5000、10000下的性能，从测试的结果来看batch size超过1000以后提升并不明显。

单表的性能基准

在分布式场景下单表只会在一个主机上，其性能可以作为一个基础的性能基线，用于评测分区表的水平扩展的能力，分区表会将数据均匀分布到多台主机上。

测试方法为对PolarDB-X中的单表进行Batch Insert操作，单表的数据只会存在一个数据存储节点中，PolarDB-X会根据表定义将数据写入到对应的数据存储节点上。

场景一：batch size

参数配置：

- 并行度：16
- 列：4
- gsi：无
- sequence：无

测试项	batch size	1	10	100	500	1000	2000	5000	10000
PolarDB-X【单表】	性能（行每秒）	5397	45653	153216	211976	210644	215103	221919	220529

场景二：并行度

参数配置：

- batch size：1000
- 列：4
- gsi：无
- sequence：无

测试项	thread	1	2	4	8	16	32	64	128
PolarDB-X【单表】	性能（行每秒）	22625	41326	76052	127646	210644	223431	190138	160858

测试总结

对于单表的测试，推荐batch size为1000，并行度为16~32时整体性能比较好。在测试batch size为2000、5000、10000时，需要添加hint参数来关闭SPLIT特性，从测试的结果来看batch size超过1000以后提升并不明显。示例：

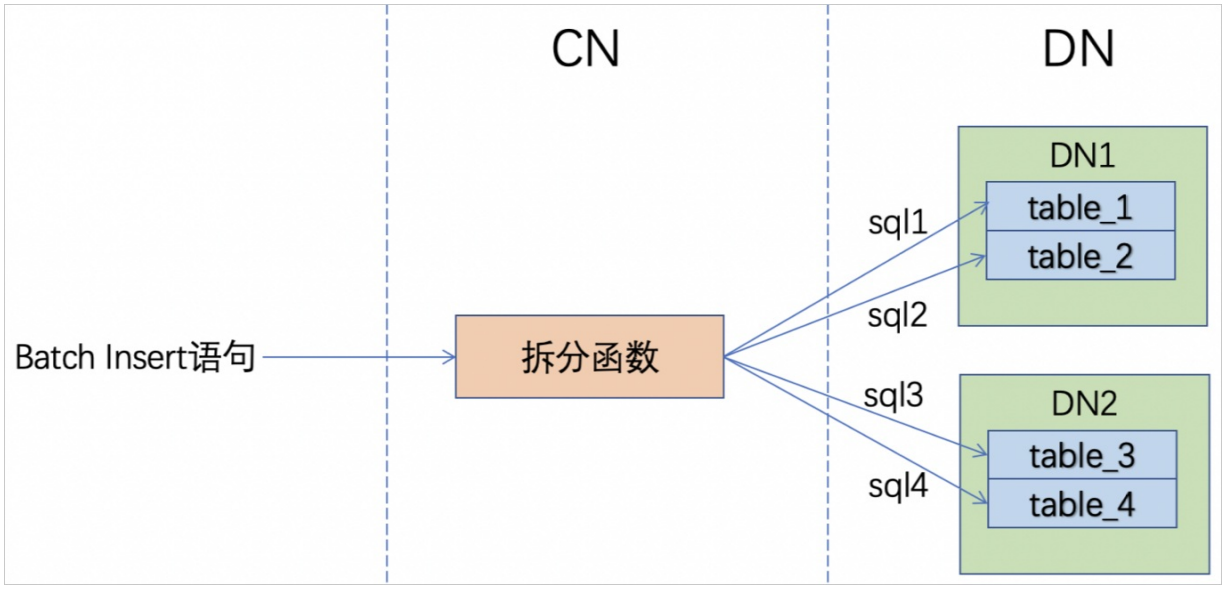
```
/*+TDDL:CMD_EXTRA(BATCH_INSERT_POLICY=NONE)*/
```

分区表的性能基准

Batch size和并行度都会影响Batch Insert的性能，下面对这两个因素分开进行测试分析。

场景一：batch Size

在数据分片的情况下，由于包含拆分函数，Batch Insert语句会经过拆分函数分离values，下推到物理存储上的batch size会改变，示意图如下图所示。



所以数据分片下，PolarDB-X的Batch Insert语句可以更大一些，或者尽量将同一个物理表的数据放到一个Batch Insert语句中，保证拆分函数分离values后下推到单个数据分片上的batch size较合适，以提升存储节点的性能。

- 子场景一 (BATCH_INSERT_POLICY=SPLIT)

  参数配置：

  - BATCH_INSERT_POLICY：开启
  - 并行度：32
  - 分片数：32
  - 列：4
  - gsi：无
  - sequence：无

测试项	batch size	1	10	100	500	1000	2000	5000	10000
PolarDB-X【单表】	性能（行每秒）	12804	80987	229995	401215	431579	410120	395398	389176

> **说明** batch size >= 2000时，会触发BATCH_INSERT_POLICY策略。

- 子场景二 (BATCH_INSERT_POLICY=NONE)

  参数配置：

  - BATCH_INSERT_POLICY：关闭
  - 并行度：32
  - 分片数：32
  - 列：4
  - gsi：无
  - sequence：无

测试项	batch size	1000	2000	5000	10000	20000	30000	50000
PolarDB-X【分片数：32】	性能（行每秒）	431579	463112	490350	526751	549990	595026	685500

  总结：

  i. `BATCH_INSERT_POLICY=SPLIT`，batch size为1000，整体性能为每秒43w行，相当于单表的两倍；

  ii. `BATCH_INSERT_POLICY=NODE`，本测试的values是随机分布，拆分函数是Hash，所以分布到每个分片上的数据基本均匀，理论情况下分片数×1000的batch size下性能会比较好，在最大batch size为50000时，整体性能为每秒68万行。

场景二：并行度

判断并行度合适的标准是将PolarDB-X数据节点的CPU利用率压满或将IOPS打满，以达到较好性能，因为Batch Insert语句基本无计算，所以PolarDB-X计算节点开销不大，主要开销在PolarDB-X数据节点。并行度过小或者过大都会影响性能，影响并行度的值的因素：节点个数、节点规格（核数和CPU）、线程池压力等，所以并行度很难得出一个确切的数字，推荐通过实践环境进行测试，找出适合该环境的最佳并行度。

- 子场景一：测试4节点下，batch size为1000的不同并行度。

  参数配置：

  - batch size：1000
  - 列：4
  - gsi：无
  - sequence：无

测试项	thread	1	2	4	8	16	32	64	80	96
PolarDB-X【分片数：32】	性能（行每秒）	40967	80535	151415	246062	367720	431579	478876	499918	487173

  总结：该配置下，64～80并发时性能达到峰值，大概50w行每秒。

- 子场景二：不同节点个数下的并行度测试

  参数配置：

  - 2节点：2CN×2DN
  - batch size：20000
  - 列：4
  - gsi：无
  - sequence：无

测试项	thread	4	8	12	16
PolarDB-X【分片数：16】	性能（行每秒）	159794	302754	296298	241444

参数配置：

- 3节点：3CN×3DN
- batch size：20000
- 列：4
- gsi：无
- sequence：无

测试项	thread	9	12	15	18
PolarDB-X【分片数：24】	性能（行每秒）	427212	456050	378420	309052

参数配置：

- 4节点：4CN×4DN
- batch size：20000
- 列：4
- gsi：无
- sequence：无

测试项	thread	16	32	40	64
PolarDB-X【分片数：32】	性能（行每秒）	464612	549990	551992	373268

总结：节点数增加，最佳性能的并行度也需要增大。2节点下峰值为8并发30万、3节点下峰值为12并发45万、4节点下峰值为32并发55万，整体随着节点数的性能提升线性率为0.9~1左右。

- 子场景三：不同节点规格下的并行度测试

参数配置：

- batch size：20000
- 列：4
- gsi：无
- sequence：无

测试项	thread	4	8	10	12	16
PolarDB-X【4核16GB】	性能（行每秒）	165674	288828	276837	264873	204738

测试项	thread	8	10	12	16
PolarDB-X【8核32GB】	性能（行每秒）	292780	343498	315982	259892

测试项	thread	16	32	40	64
PolarDB-X【16核64GB】	性能（行每秒）	464612	549990	551992	373268

总结：节点规格升级，最佳性能的并行度也需要增大。4核16GB下峰值为8并发28w、8核32GB下峰值为10并发34w、16核64GB下峰值为32并发55w，整体随着节点规格的性能提升线性率为0.5~0.6左右。

测试总结

- 默认情况下，batch size建议为1000，并发度建议16~32并发，整体实例的并发性能和资源负载会比较优。
- 追求数据批量导入的最大速度，可增大batch size，建议batch size = 分片数 * [100~1000]，比如20000~50000。对应的单条Batch语句的大小控制2MB ~ 8MB（默认最大包大小为16MB），同时需要通过hint设置BATCH_INSERT_POLICY=NONE。注意点：单条sql语句过大时，分布式下单个计算节点的压力会偏重，首先会带来一定的内存消耗风险，其次可能会出现多个节点之间的压力不均衡。
- Batch的批量导入，消耗更多的是IOPS的资源，CPU/MEM不是主要瓶颈。因此，如果需要做资源升配来提升性能时，可以优先考虑扩容节点数，其次再考虑升配单节点规格。
- 线下文本数据的批量导入，建议使用PolarDB-X配套的导入导出工具Batch Tool，请参见使用Batch Tool工具导入导出数据。

9.13. 如何优化数据导入导出

数据库实际应用场景中经常需要进行数据导入导出，本文将介绍如何使用数据导入导出工具。

测试环境

本文档的测试环境要求如下表：

环境	参数
PolarDB-X版本	polarx-kernel_5.4.11-16282307_xcluster-20210805
节点规格	16核64GB
节点个数	4个

测试用表如下：

```
CREATE TABLE `sbtest1` (
    `id` int(11) NOT NULL,
    `k` int(11) NOT NULL DEFAULT '0',
    `c` char(120) NOT NULL DEFAULT '',
    `pad` char(60) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`),
    KEY `k_1` (`k`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 dbpartition by hash(`id`);
```

导入导出工具介绍

PolarDB-X常见的数据导出方法有：

- mysql -e命令行导出数据
- musqldump工具导出数据
- select into outfile语句导出数据（默认关闭）
- Batch Tool工具导出数据（PolarDB-X配套的导入导出工具）

PolarDB-X常见的数据导入方法有：

- source语句导入数据
- mysql命令导入数据
- 程序导入数据
- load data语句导入数据
- Batch Tool工具导入数据（PolarDB-X配套的导入导出工具）

MySQL原生命令使用示例

mysql -e命令可以连接本地或远程服务器，通过执行sql语句，例如select方式获取数据，原始输出数据以制表符方式分隔，可通过字符串处理改成 `','` 分隔，以csv文件方式存储，方法示例：

```
mysql -h ip  -P port -u usr -pPassword db_name -N -e "SELECT id,k,c,pad FROM sbtest1;" >/home/data_1000w.txt
## 原始数据以制表符分隔，数据格式：188092293    27267211    59775766593-64673028018-...-09474402685    01705051424-...-54211554755
mysql -h ip  -P port -u usr -pPassword db_name -N -e "SELECT id,k,c,pad FROM sbtest1;" | sed 's/\t/,/g' >/home/data_1000w.csv
## csv文件以逗号分隔，数据格式：188092293,27267211,59775766593-64673028018-...-09474402685,01705051424-...-54211554755
```

原始数据格式适合load data语句导入数据，使用方法可参考：LOAD DATA 语句，示例如下：

```
LOAD DATA LOCAL INFILE '/home/data_1000w.txt' INTO TABLE sbtest1;
## LOCAL代表从本地文件导入，local_infile参数必须开启
```

csv文件数据适合程序导入，具体方式可查看使用程序进行数据导入。

mysqldump工具使用示例

mysqldump工具可以连接到本地或远程服务器，详细使用方法请参见使用mysqldump导入导出数据。

- 导出数据示例：

  ```
  mysqldump -h ip  -P port -u usr -pPassword --default-character-set=utf8mb4 --net_buffer_length=10240 --no-tablespaces --no-create-db --no-create-info --skip-add-locks --skip-lock-tables --skip-tz-utc --set-charset  --hex-blob db_name [table_name] > /home/dump_1000w.sql
  ```

  mysqldump导出数据可能会出现的问题及解决方法，这两个问题通常是mysql client和mysql server版本不一致导致的。

  i. 问题：mysqldump: Couldn't execute 'SHOW VARIABLES LIKE 'gtid_mode''

     解决方法：添加“--set-gtid-purged=OFF”参数关闭gtid_mode。

  ii. 问题：mysqldump: Couldn't execute 'SHOW VARIABLES LIKE 'ndbinfo_version''

     解决方法：查看mysqldump --version和mysql版本是否一致，使用和mysql版本一致的mysql client。

  导出的数据格式是SQL语句方式，以Batch Insert语句为主体，包含多条SQL语句， `INSERT INTO `sbtest1` VALUES (...),(...)` ，“net_buffer_length”参数将影响batch size大小。

- SQL语句格式合适的导入数据方式：

  ```
  方法一：souce语句导入数据
  source /home/dump_1000w.sql
  方法二：mysql命令导入数据
  mysql -h ip  -P port -u usr -pPassword --default-character-set=utf8mb4 db_name < /home/dump_1000w.sql
  ```

Batch Tool工具使用示例

Batch Tool是阿里云内部开发的数据导入导出工具，支持多线程操作。

- 导出数据：

  ```
  ## 导出“默认值=分片数”个文件
  java -jar batch-tool.jar -h ip  -P port -u usr -pPassword -D db_name -o export -t sbtest1 -s ,
  ## 导出整合成一个文件
  java -jar batch-tool.jar -h ip  -P port -u usr -pPassword -D db_name -o export -t sbtest1 -s , -F 1
  ```

- 导入数据：

```
## 导入32个文件
java -jar batch-tool.jar -hpxc-spryb387va1ypn.polarx.singapore.rds.aliyuncs.com  -P3306 -uroot -pPassw0rd
-D sysbench_db -o import -t sbtest1 -s , -f "sbtest1_0;sbtest1_1;sbtest1_2;sbtest1_3;sbtest1_4;sbtest1_5;s
btest1_6;sbtest1_7;sbtest1_8;sbtest1_9;sbtest1_10;sbtest1_11;sbtest1_12;sbtest1_13;sbtest1_14;sbtest1_15;s
btest1_16;sbtest1_17;sbtest1_18;sbtest1_19;sbtest1_20;sbtest1_21;sbtest1_22;sbtest1_23;sbtest1_24;sbtest1_
25;sbtest1_26;sbtest1_27;sbtest1_28;sbtest1_29;sbtest1_30;sbtest1_31" -np -pro 64 -con 32
## 导入1个文件
java -jar batch-tool.jar -h ip  -P port -u usr -p password -D db_name -o import -t sbtest1 -s , -f "sbtest
1_0" -np
```

导出方法对比

测试方法以PolarDB-X导出1000w行数据为例，数据量大概2GB左右。

方式	数据格式	文件大小	耗时	性能（行/每秒）	性能（MB/S）
mysql -e命令 导出原始数据	原始数据格式	1998MB	33.417s	299248	59.8
mysql -e命令导出csv格式	csv格式	1998MB	34.126s	293031	58.5
mysqldump工具（net-buffer-length=10KB）	sql语句格式	2064MB	30.223s	330873	68.3
mysqldump工具（net-buffer-length=200KB）	sql语句格式	2059MB	32.783s	305036	62.8
batch tool工具 文件数=32（分片数）	csv格式	1998MB	4.715s	2120890	423.7
batch tool工具 文件数=1	csv格式	1998MB	5.568s	1795977	358.8

总结：

1. mysql -e命令和mysqldump工具原理上主要是单线程操作，性能差别并不明显。
2. Batch Tool工具采用多线程方式导出，并发度可设置，能够极大提高导出性能。

导入方法对比

测试方法以PolarDB-X导入1000w行数据为例，源数据是上一个测试中导出的数据，数据量大概2GB左右。

方式	数据格式	耗时	性能（行/每秒）	性能（MB/S）
source语句（net-buffer-length=10KB）	sql语句格式	10m24s	16025	3.2
source语句（net-buffer-length=200KB）	sql语句格式	5m37s	29673	5.9
mysql命令导入（net-buffer-length=10KB）	sql语句格式	10m27s	15948	3.2
mysql命令导入（net-buffer-length=200KB）	sql语句格式	5m38s	29585	5.9

方式	数据格式	耗时	性能（行/每秒）	性能（MB/S）
load data语句导入	原始数据格式	4m0s	41666	8.3
程序导入batch-1000thread-1	csv格式	5m40s	29411	5.9
程序导入batch-1000thread-32	csv格式	19s	526315	105.3
batch tool工具文件数=32（分片数）	csv格式	19.836s	504133	100.8
batch tool工具文件数=1	csv格式	10.806s	925411	185.1

总结：

1. source语句和mysql命令导入方式，都是单线程执行SQL语句导入，实际是Batch Insert语句的运用，Batch size大小会影响导入性能。Batch size和mysqldump导出数据时的“net-buffer-length”参数有关。建议优化点如下：
   - 推荐将“net-buffer-length”参数设置大，不超过256K，以增大batch size大小，来提高插入性能。
   - 使用第三方工具，例如mysqldump，进行mydumper（备份）和myloader（导入）等，可多线程操作。
2. load data语句是单线程操作，性能优于mysql命令和source语句。
3. 程序导入灵活性较好，可自行设置合适的batch size和并发度，可以达到较好性能。推荐batch大小为1000，并发度为16~32。
4. Batch Tool工具支持多线程导入，且贴合分布式多分片的操作方式，性能优异。

总结

1. PolarDB-X兼容MySQL运维上常用的数据导入导出方法，但这些方法大多为MySQL单机模式设计，只支持单线程操作，性能上无法充分利用所有分布式资源。
2. PolarDB-X提供Batch Tool工具，非常贴合分布式场景，在多线程操作下，能够达到极快的数据导入导出性能。

9.14. 如何优化数据全量抽取

本文介绍了在应用内通过代码高效抽取数据的方法。

简介

数据抽取是指通过代码或者数据导出工具，从PolarDB-X中批量读取数据的操作。主要包括以下场景：

- 通过数据导出工具将数据全量抽取到下游系统。PolarDB-X支持多种数据导出工具，更多内容请参考数据导入导出。
- 在应用内处理数据或者批量的将查询结果返回给用户浏览时，不能依赖外部工具，必须在应用内通过代码完成数据全量抽取。

本文主要介绍在应用内通过代码高效抽取数据的方法，根据是否一次性读取全量数据，分为全量抽取和分页查询。

全量抽取场景

全量抽取使用的SQL通常不包含表的拆分键，以全表扫描的方式执行，随着读取数据量的增加，数据抽取操作的执行时间线性增长。为了避免占用过多网络/连接资源，可以使用HINT直接下发查询语句，从物理分片中拉取数据。以下示例采用JAVA代码编写，完整使用方法参考如何使用HINT。

```
public static void extractData(Connection connection, String logicalTableName, Consumer<ResultSet> consumer)
    throws SQLException {
    final String topology = "show topology from {0}";
    final String query = "/*+TDDL:NODE({0})*/select * from {1}";
    try (final Statement statement = connection.createStatement()) {
        final Map<String, List<String>> partitionTableMap = new LinkedHashMap<>();
        // Get partition id and physical table name of given logical table
        try (final ResultSet rs = statement.executeQuery(MessageFormat.format(topology, logicalTableName)))
{
            while (rs.next()) {
                partitionTableMap.computeIfAbsent(rs.getString(2), (k) -> new ArrayList<>()).add(rs.getStrin
g(3));
            }
        }
        // Serially extract data from each partition
        for (Map.Entry<String, List<String>> entry : partitionTableMap.entrySet()) {
            for (String tableName : entry.getValue()) {
                try (final ResultSet rs = statement
                    .executeQuery(MessageFormat.format(query, entry.getKey(), tableName))) {
                    // Consume data
                    consumer.accept(rs);
                }
            }
        }
    }
}
```

分页查询场景

向用户展示列表信息时，需要分页来提高页面的加载效率，避免返回过多冗余信息，用于处理分页显示需求的查询，称为分页查询。关系型数据库没有直接提供分段返回表中数据的能力，高效的实现分页查询，还需要结合数据库本身的特点来设计查询语句。

以MySQL为例，分页查询最直观的实现方法，是使用limit offset，pageSize来实现，例如如下查询：

```
select * from t_order where user_id = xxx order by gmt_create, id limit offset, pageSize
```

因为gmt_create可能重复，所以order by时应加上id，保证结果顺序的确定性。

> **说明** 该方案在表规模较小的时候，能够正常运行。当t_order表增长到十万级，随着页数增加，执行速度明显变慢，可能降到几十毫秒的量级，如果数据量增长到百万级，则耗时达到秒级，数据量继续增长，耗时最终会变得不可接受。

问题分析

假设我们在user_id, gmt_create上创建了局部索引，由于只有user_id上的条件，每次需要扫描的总数据量为offset + pageSize，随着offset的增大逐渐接近全表扫描，导致耗时增加。并且在分布式数据库中，全表排序的吞吐无法通过增加DN数量来提高。

改进方案1

每次获取下一页记录时，指定从上次结束的位置继续往后取，这样不需要设置offset，能够避免出现全表扫描的情况。看一个按id进行分页查询的例子：

```
select * from t_order where id > lastMaxId order by id limit pageSize
```

第一次查询不指定条件，后续查询则传入前一次查询的最大id，在执行时，数据库首先在索引上定位到lastMaxId的位置，然后连续返回pageSize条记录即可，非常高效。

> **说明** 当id为主键或者唯一键时，改进方案1可以达到分页查询的效果，也有不错的性能。但缺点也比较明显，当id上有重复值时，可能会漏掉部分记录。

改进方案2

MySQL支持通过 Row Constructor Expression实现多列比较的语义（PolarDB-X同样支持）。

```
(c2,c3) > (1,1)
等价于
c2 > 1 OR ((c2 = 1) AND (c3 > 1))
```

因此，可以用下面的方法实现分页查询语义：

```
select * from t_order
where user_id = xxx and (gmt_create, id) > (lastMaxGmtCreate, lastMaxId)
order by user_id, gmt_create, id limit pageSize
```

第一次查询不指定条件，后续查询则传入前一次查询的最大gmt_create和id，通过Row Constructor Expression正确处理gmt_create存在重复的情况。

> **说明** 示例中，为了提高查询性能，我们在user_id和gmt_create上建立联合索引，并在order by中加入user_id提示优化器可以通过索引来消除排序。由于Row Constructor Expression包含null值会导致表达式求值结果为null，当存在null值时需要使用OR表达式。PolarDB-X目前只在Row Constructor Expression仅包含拆分键时才将其用于分区裁剪，其他场景同样需要使用OR表达式。

结合上述分析，给出一个PolarDB-X上分页查询的最佳实践：

```
-- lastMaxGmtCreate is not null
select * from t_order
where user_id = xxx
and (
      (gmt_create > lastMaxGmtCreate)
      or ((gmt_create = lastMaxGmtCreate) and (id > lastMaxId))
    )
order by user_id, gmt_create, id limit pageSize
-- lastMaxGmtCreate is null
select * from t_order
where user_id = xxx
and (
      (gmt_create is not null)
      or (gmt_create is null and id > lastMaxId)
    )
order by user_id, gmt_create, id limit pageSize
```

9.15. 如何使用Blob对象

本文介绍了如何使用Blob对象。

Java的MySQL各版本驱动（包括5.1.x和8.0.x），在实现PreparedStatement.setBlob方法时都有一些缺陷（无论连接的是mysql server还是PolarDB-X，都存在缺陷），对于一些用于存储二进制格式（例如图片、视频等）的Blob对象，会有概率性的报出语法错误，本文列出一些需要注意的地方。

场景一：PolarDB-X版本<5.4.9

对于低版本的PolarDB-X，其自身不支持SQL中直接携带与连接串字符集不符的二进制信息，需要在客户端转义成如下列所示的十六进制字符串。

```
insert into t1 values (0xaabbccdd,x'aabbccdd');
```

方法1：对于Java用户，可以使用PreparedStatement.setBytes方法替换setBlob方法，mysql驱动会将byte[]转义成十六进制字符串。

方法2：对于Java用户，如果不方便使用方法1（例如使用Hibernate等框架，无法控制框架使用哪个set方法），请联系阿里云技术支持，我们会提供一个定制的mysql驱动包，该驱动包内会完成转义。

方法3：对于其他语言用户，请在应用程序中自行完成转义。

场景二：5.4.9<=PolarDB-X版本<5.4.13

该版本的PolarDB-X，额外支持了_binary前缀。因此，除了场景一中的方法继续适用之外，还有以下方法可以使用：

方法4：对于Java用户，可以修改SQL语句，例如原语句：

```
insert into t1 values (?)
```

修改为：

```
insert into t1 values (_binary?)
```

方法5：对于Java用户，可以使用8.0.26版本的mysql驱动，该驱动在setBlob方法内会自动加上_binary前缀。

场景三：PolarDB-X版本>=5.4.13

该版本比较彻底的兼容了mysql对于二进制信息的处理，上述场景一和场景二的方法继续适用，还有以下方法可以使用：

方法6：使用utf8/utf8mb4连接数据库，例如对于Java用户，jdbcurl中加入参数：

```
useUnicode=true&characterEncoding=utf8
```

对于其他语言用户，可以在建完连接后，执行：

```
set names utf8mb4;
```

实际上，mysql驱动对于setBlob的实现是有问题的，即使是使用官方mysql server，也必须在连接使用utf8/utf8mb4编码的情况下，才能很好的支持setBlob这种使用方法；对于gbk等编码，都有一定的概率报语法错误。

9.16. 偏高并发场景的实践和优化

本文介绍了如何判断查询语句是否为“点查”，以及如何将查询优化为“点查”。

“点查”是应用访问OLTP数据库的一种常见方式，特点是返回结果前只扫描表中的少量数据，在淘宝上查看订单/商品信息对应到数据库上的操作就是点查。PolarDB-X对点查的响应时间（Response Time, RT）和资源占用做了较多优化，能够支持较高的吞吐，适合高并发读取场景使用。

什么是点查

顾名思义，“点查”是指只扫描少量数据的查询。注意这里说的是“扫描少量数据”而不是“返回少量数据”，比如 `select * from t1 order by c1 limit 1` 虽然只返回了一条数据，但如果c1上没有索引，需要先扫描t1上所有数据排序后才能返回结果，不符合“点查”的定义。

单机数据库中，最常见的点查是按照主键（Primary Key, PK）查询数据，通过扫描主键索引快速得到结果，平均只需要扫描logn条记录。如果通过其他条件查询，可以增加局部二级索引（Local Secondary Index，LSI），首先扫描局部二级索引得到主键，然后回表查出完整记录。特殊场景下，如果局部二级索引中包含了查询涉及的所有列，则回表的步骤也可以省略。

PolarDB-X是一个分布式数据库，为了将数据分散到不同数据节点（Data Node，DN）上，引入了分区表的概念，预先将数据切分成多个分区，然后建立分区和DN的映射，其中切分数据需要选取一个或多个列作为切分维度，这些列因此被称为“分区键”。分布式数据库中，查询性能除了与扫描的数据量线性相关，还与扫描的分片数量正相关，因此“点查”的定义还需要加上“扫描少量分区”。

PolarDB-X具备透明分布式能力，默认使用主键作为分区键，按照PK查询时首先定位到数据所在的分区，然后通过分区上的主键索引得到结果，性能最高。如果通过其他条件查询，可以增加全局二级索引（Global Secondary Index，GSI）。使用GSI优化查询的原理与LSI相同，首先查到主键然后回表获得完整记录，主要区别在于GSI本身也是一张分区表，数据与主表保存在不同DN上，回表操作大概率需要经过网络，回表代价高于单机数据库。因此，PolarDB-X支持创建聚簇索引来消除回表，达到与主键查询相同的性能。

注意事项

LSI和GSI本质上是以额外存储空间和写入开销为代价，换取查询性能的方案，使用时需要谨慎评估对写入性能的影响。索引表与主表的数据分布不同，为了保证GSI的数据与主表强一致，所有涉及GSI的写入操作都默认被包装在分布式事务中。相比没有GSI的场景，写入RT会增加2~3倍，同时由于索引表和主表混合并行写入，高并发写入场景下产生分布式死锁的概率会增加。综上所述，建议每张逻辑表上创建不超过3个GSI。

如何识别点查

如上所述，分布式数据库中的点查，是指扫描少量分片和数据的查询。通过查看执行计划，可以确认一个查询语句扫描的分片数，更多执行计划介绍请参见执行计划介绍。以下为一个点查的示例：

```
> explain select c_custkey, c_name, c_address from customer where c_custkey = 42;
+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| LogicalView(tables="TEST1_000002_GROUP.customer_IVgG_10", sql="SELECT `c_custkey`, `c_name`, `c_address` F
ROM `customer` AS `customer` WHERE (`c_custkey` = ?)") |
+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------+
```

EXPLAIN EXECUTE用于汇总展示DN上的执行计划，由此可以判断查询在DN上是否命中正确的索引。DN节点基于MySQL实现，执行计划与MySQL相同，更多介绍参考 MySQL 官方文档。以下展示一个点查的示例：

```
> explain execute select c_custkey, c_name, c_address from customer where c_custkey = 42;
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+-----
-----+-----------------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref   | rows | filt
ered | Extra           |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+-----
-----+-----------------+
| 1  | SIMPLE      | customer | NULL       | const | PRIMARY       | PRIMARY | 4       | const | 1    | 100
| Using pk access |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+-----
-----+-----------------+
```

如何将查询优化为点查

不是所有查询语句都可以优化为点查，例如没有任何条件的数据抽取查询 `select * from t1` ，不合理的分页查询 `select * from t1 where c1 = 1 limit 100000, 10` ，参数数量随着业务增长而增长的IN查询等。能够优化为点查的语句可以概括为以下两类：

1. 固定范围扫描的查询：条件中包含等值条件（或可以简化为等值条件），小范围BETWEEN AND条件，参数数量固定的IN条件的查询；
2. 结果行数固定的TopN查询：例如 `select * from t1 where c1 > 42 limit 10` 和 `select * from t1 order by c1 limit 10` 和 `select * from t1 order by c1 limit 10` 。

对于这两类查询，优化的思路是添加合适的索引，将全表扫描转化为索引扫描，示例如下：

```
CREATE TABLE `customer` (
  `c_custkey` int(11) NOT NULL,
  `c_name` varchar(25) NOT NULL,
  `c_address` varchar(40) NOT NULL,
  `c_nationkey` int(11) NOT NULL,
  `c_phone` varchar(15) NOT NULL,
  `c_acctbal` decimal(15,2) NOT NULL,
  `c_mktsegment` varchar(10) NOT NULL,
  `c_comment` varchar(117) NOT NULL,
  PRIMARY KEY (`c_custkey`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 dbpartition by hash(`c_custkey`) tbpartition by hash(`c_custkey`) tbp
artitions 4;
> explain select * from customer where c_phone = "11";
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
---------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
---------------------------+
| Gather(concurrent=true)
|
|   LogicalView(tables="[000000-000003].customer_[00-15]", shardCount=16, sql="SELECT `c_custkey`, `c_name`,
`c_address`, `c_nationkey`, `c_phone`, `c_acctbal`, `c_mktsegment`, `c_comment` FROM `customer` AS `customer
` WHERE (`c_phone` = ?)") |
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
---------------------------+
```

customer表上只有主键索引，因此虽然c_phone指定了等值条件，依然需要扫描全部分片，可以通过添加GSI来优化。

```
> create global index g_i_phone on customer(c_phone) dbpartition by hash(c_phone);
> explain select * from customer where c_phone = "11";
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------------------------------------+
| LOGICAL EXECUTIONPLAN
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------------------------------------+
| Project(c_custkey="c_custkey", c_name="c_name", c_address="c_address", c_nationkey="c_nationkey", c_phone=
"c_phone", c_acctbal="c_acctbal", c_mktsegment="c_mktsegment", c_comment="c_comment")
|
|   BKAJoin(condition="c_custkey = c_custkey", type="inner")
|
|     IndexScan(tables="TEST1_000000_GROUP.g_i_phone_2CSp", sql="SELECT `c_custkey`, `c_phone` FROM `g_i_pho
ne` AS `g_i_phone` WHERE (`c_phone` = ?)")
|
|     Gather(concurrent=true)
|
|       LogicalView(tables="[000000-000003].customer_[00-15]", shardCount=16, sql="SELECT `c_custkey`, `c_na
me`, `c_address`, `c_nationkey`, `c_acctbal`, `c_mktsegment`, `c_comment` FROM `customer` AS `customer` WHER
E ((`c_phone` = ?) AND (`c_custkey` IN (...)))") |
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-------------------------------------------------+
```

添加GSI后，查询变为索引表上的点查加回表，回表操作只访问一个分片（执行计划中回表显示为主表上的全表扫描，这是因为确定需要扫描的主表分片依赖索引表的查询结果，explain阶段无法确定）。

```
> drop index g_i_phone on customer;
> create clustered index g_i_phone on customer(c_phone) dbpartition by hash(c_phone);
> explain select * from customer where c_phone = "11";
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------
-----------+
| LOGICAL EXECUTIONPLAN
|
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------
-----------+
| IndexScan(tables="TEST1_000000_GROUP.g_i_phone_fHmZ", sql="SELECT `c_custkey`, `c_name`, `c_address`, `c_n
ationkey`, `c_phone`, `c_acctbal`, `c_mktsegment`, `c_comment` FROM `g_i_phone` AS `g_i_phone` WHERE (`c_pho
ne` = ?)") |
+-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------
-----------+
```

使用聚簇索引代替GSI后，由于索引表中包含了主表上的所有列，不再需要回表，执行计划变为索引表上的点查。

以上示例阐述了通过索引优化点查性能的一般过程，其中的关键点是根据查询特征找到适合添加索引的列。对于包含多个条件比较复杂的查询，可以通过PolarDB-X内置的索引推荐功能来找到合适的LSI和GSI，详情请参考智能索引推荐。

9.17. 偏分析场景的实践和优化

PolarDB-X是一款以TP为主的HTAP数据库，也支持一定场景的分析需求。而典型的分析场景一般有以下几类特征：

- 少量的写或者更新请求，大多数是读请求；
- 每次查询都从数据库中读取大量的行，但是同时又仅需要少量的列；
- 大多数查询都是比较复杂的查询，查询的并发不会很大，但单个查询需要高吞吐量；
- 对于简单查询，允许一定的延迟；
- 分析场景上分布式事务可能不是必须的；
- 大部分查询中往往会涉及到事实表和维表的关联，是典型的大小表关联场景；
- 查询结果明显小于源数据，即数据被过滤或聚合后能够被盛放在单台服务器的内存中；
- 分析的数据往往是最近的业务数据，历史数据可以被清理或者被归档。

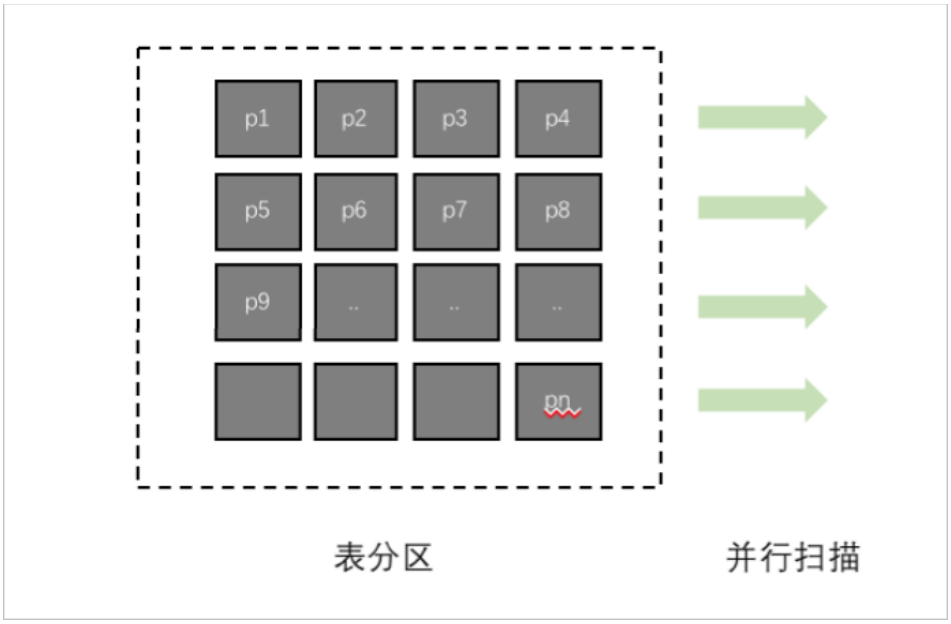
依据上述对分析场景的归纳，分析场景做性能优化除了要沿用TP数据库的优化思路，还会有自身不一样的优化思路。这主要会体现在结构设计和查询优化两个方面。

结构设计

在结构设计上主要包括如何选择表类型、分区键、主键以及聚簇，使表的性能达到最优。

设计为分区表或者广播表

1. 广播表会在集群的每个数据节点都存储一份数据，建议广播表的数据量不宜太大，每张广播表存储的数据不超过20万行，这样在大表和广播表做关联时，可以计算下推，让关联贴近数据层做计算，避免大表数据拉取到计算节点做计算。
2. 其他业务数据尽可能做成分区表，可以充分利用分布式系统的查询能力。理论上表的分区数量越多越好，这样多个分区表可以做并行扫描。存储层更易做到水平扩展，存储千万条甚至上亿条数据。不过实际使用中建议一个分区表的数量在500w~5000w之间。



选择合适的分区键

PolarDB-X默认按照主键做分拆，主要为了降低您使用分布式数据库的成本。同时我们也支持通过指定分区键建分区表，在分析场景建议您根据如下依据选择的分区键：

1. 尽可能选择参与JOIN的字段作为分区键，这样做的目的还是为了关联条件下推，避免数据被拉取到计算层做计算。
2. 尽可能选择值分布均匀的字段作为分区键，这样可以避免由于分布式不均导致出现计算长尾现象，严重托慢大查询性能。

合理设计二级分区

PolarDB-X支持二级分区。当数据量过大或者有数据倾斜时，二级分区的选择至关重要，如果数据量大的表中没有二级分区或者二级分区切分不合理，也会影响性能。如果业务明确有增量数据导入需求，主要是对最近数据的报表分析，那么建议用日期格式做二级分区，避免对历史过期数据的扫描。

```
//直接用col先做一级分区
PARTITION BY HASH(col)
SUBPARTITION BY LIST (ds)
//ds转换后的月做分区
SUBPARTITION TEMPLATE (
   PARTITION p1 VALUES LESS THAN ('2021-08-00'),
   PARTITION p2 VALUES LESS THAN ('2021-09-00'),
)
```

合理设计索引

如果业务已经按照关联字段，合理的设计了分区键。但依然还有部分复杂查询涉及到对该表的其他列做关联，无法做到关联查询下推，此时可以考虑基于该非分区键的列做全局二级索引。这样复杂查询对该表做关联，可以转化成与该全局二级索引做关联。同时了为了避免回表的代价，对于分析场景建议所有的全局二级索引都建成聚簇索引。

查询优化

在分析场景中，由于会涉及比较大的数据，且对简单查询的延迟有一定的容忍度，推荐您采用MPP执行模式，既利用多个计算节点（CN）的计算资源承担复杂计算。一般只在只读实例默认开启MPP能力，如果您可以允许在主实例做分析需求，请联系阿里云技术支持。

在查询过程中，PolarDB-X首先会基于优化器选择合适的分布式执行计划，然后将计划调度到各个计算节点，充分发挥整个集群的计算资源加速查询。这个过程生成的分布式执行计划完全是基于统计信息做代价选择，所以及时的信息采集至关重要；同时由于优化器生成的计划不一定是最优的，所以这里也给到您在SQL编写和优化时的经验。

收集统计信息

PolarDB-X会及时定时收集统计信息，如果发现PolarDB-X生成的分布式执行计划不是最优的。可以通过ANALYZE TABLE手动对某个表做统计信息收集。

SQL编写技巧

- 去掉不必要的列

  由于分析场景大多数是高吞吐的，所以应该去除返回过程中不必要的列，减少对带宽的压力。在编写SQL时一定要确认业务需要返回的列，不要直接使用星号（*）进行查询。

```
//不合适写法
select * from T1 where a1>100 and a2<1000;
//更合适写法，只需要返回业务关心的列
select a1, a2 from T2 where a1>100 and a2<1000;
```

- 基于局部索引做过滤

  很多分析场景都期望用时间做二级分区，这样做大数据扫描的时候可以把时间做过滤条件，过滤掉绝大多数历史数据。

```
select a1,c2 from T1 where time >='2010-01-01 00:00:00';
```

  为了避免全部扫描，目前默认会在这个分区列上做局部索引。同样的在很多高吞吐的扫描场景下，可以考虑基于过滤条件做局部索引。

- 避免低效的SQL语法

  如果表记录数非常大，扫描会很慢，直接导致查询缓慢。所以在SQL编写过程中我们需要注意以下几点：

  第一，避免索引失效

  i. 不在索引列上做任何操作，计算、函数、类型转换（自动或手动），会导致索引失效而转向全表扫描。

```
mysql> explain execute select * from staffs where name= 'hu';
+----+-------------+--------+------------+------+-----------------------+-----------------------+-----
----+-------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys         | key                   | key_
len | ref   | rows | filtered | Extra |
+----+-------------+--------+------------+------+-----------------------+-----------------------+-----
----+-------+------+----------+-------+
|  1 | SIMPLE      | staffs | NULL       | ref  | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74
| const |    1 |      100 | NULL  |
+----+-------------+--------+------------+------+-----------------------+-----------------------+-----
----+-------+------+----------+-------+
1 row in set , 1 warning (0.00 sec)
//在索引列上做了其他操作，导致索引失效
mysql> explain execute select * from staffs where left(name,4)= 'hu';
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filte
red | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
|  1 | SIMPLE      | staffs | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  198 |
100 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
1 row in set , 1 warning (0.00 sec)
```

  ii. 在使用不等于（！=或<>）的时候，无法使用索引导致全表扫描。

  iii. is null，is not null也无法使用索引。

```
mysql>  explain execute select * from staffs where name is null ;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+-------
---+------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filter
ed | Extra            |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+-------
---+------------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL | NULL
| Impossible WHERE |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+-------
---+------------------+
1 row in set
```

iv. like以通配符开头，mysql索引失效会进行全表扫描的操作。

```
mysql>  explain exeucte select * from staffs where name like '%hu' ;
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filte
red | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
|  1 | SIMPLE      | staffs | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  198 |    11
.11 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+------
----+-------------+
1 row in set
mysql>  explain execute select * from staffs where name like 'hu%' ;
+----+-------------+--------+------------+-------+-----------------------+-----------------------+----
-----+------+------+----------+-----------------------+
| id | select_type | table  | partitions | type  | possible_keys         | key                   | key
_len | ref  | rows | filtered | Extra                 |
+----+-------------+--------+------------+-------+-----------------------+-----------------------+----
-----+------+------+----------+-----------------------+
|  1 | SIMPLE      | staffs | NULL       | range | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74
| NULL |    1 |      100 | Using index condition |
+----+-------------+--------+------------+-------+-----------------------+-----------------------+----
-----+------+------+----------+-----------------------+
1 row in set
```

第二，尽量少用like，like操作一般不会很高效，尽量使用范围条件到达目的。比如between...and...

第三，多表关联场景下：

i. 尽量包含分区列条件。如果不包含，则尽量通过WHERE条件过滤掉多余的数据。

ii. outer join的on和where作用域不同。on是作用于join的过程，where是作用于join之后的结果，所以应该将能在join的时候提前过滤的条件写在on上，也可以写在join表的子查询里，这样可以减少join原始表的数据量。

数据倾斜的检查和处理

如果出现查询异常缓慢，或者资源利用率不均匀的情况，则需要确认是否出现了数据倾斜。一般解决倾斜有三种策略：

1. 通过 `show info from table` 检查某个分片在各个节点上的数据分布计数，如果各节点上的数据分布明显不均匀，则可以考虑对该表的分区键进行调整。
2. 如果是出现了严重Join Key热点问题，将倾斜的Key用单独的逻辑来处理。例如两边的Key中有大量NULL数据导致了倾斜，则需要在Join前先过滤掉NULL数据或者补上随机数，然后再进行Join，示例如下。

```
SELECT * FROM A JOIN B ON CASE WHEN A.value IS NULL THEN CONCAT('value',RAND() ) ELSE A.value END = B.va
lue;
```

在实际场景中，如果您发现已经数据倾斜，但无法获取导致数据倾斜的Key信息，可以使用如下方法查看数据倾斜。

```
--执行如下语句查询数据倾斜。
SELECT * FROM a JOIN b ON a.key=b.key;
--您可以执行如下SQL，查看Key的分布，判断执行Join操作时是否会有数据倾斜。
SELECT left.key, left.cnt * right.cnt FROM
(select key, count(*) AS cnt FROM a GROUP BY key) LEFT
JOIN
(SELECT key, COUNT(*) AS cnt FROM b GROUP BY key) RIGHT
ON left.key=right.key;
```

3. 如果Group By Key出现了热点问题，可以考虑对SQL进行改写，添加随机数，把长Key进行拆分。例如：

```
SELECT Key,COUNT(*) AS Cnt FROM TableName GROUP BY Key;
//优化成以下SQL，先对热点做打散预聚合，再做最终聚合
-- 假设长尾的Key已经找到是KEY001。
SELECT a.Key
  , SUM(a.Cnt) AS Cnt
FROM (
  SELECT Key
    , COUNT(*) AS Cnt
FROM TableName
GROUP BY Key,
  CASE
    WHEN Key = 'KEY001' THEN rand() % 50
    ELSE 0
   END
) a
GROUP BY a.Key;
```

调整执行策略

按照上述策略调整后，查询性能依然不理想且计算和存储资源都未到达瓶颈，这个时候可以调整下执行策略。主要有两种方式去调整：

1. 加大并发度，您可以通过HINT `/*+TDDL:MPP_PARALLELISM=4*/` 指定MPP执行器并行度。

```
mysql> /*+TDDL:TDDL:MPP_PARALLELISM=4*/ select a.k, count(*) cnt from sbtest1 a, sbtest1 b where a.id =
b.k and a.id > 1000 group by k having cnt > 1300 or
der by cnt limit 5, 10;
```

2. 通过HINT指定特定的算法，如何调整更好的聚合算法和关联算法，请参见聚合和关联。

9.18. 混合负载HTAP的实践和优化

背景信息

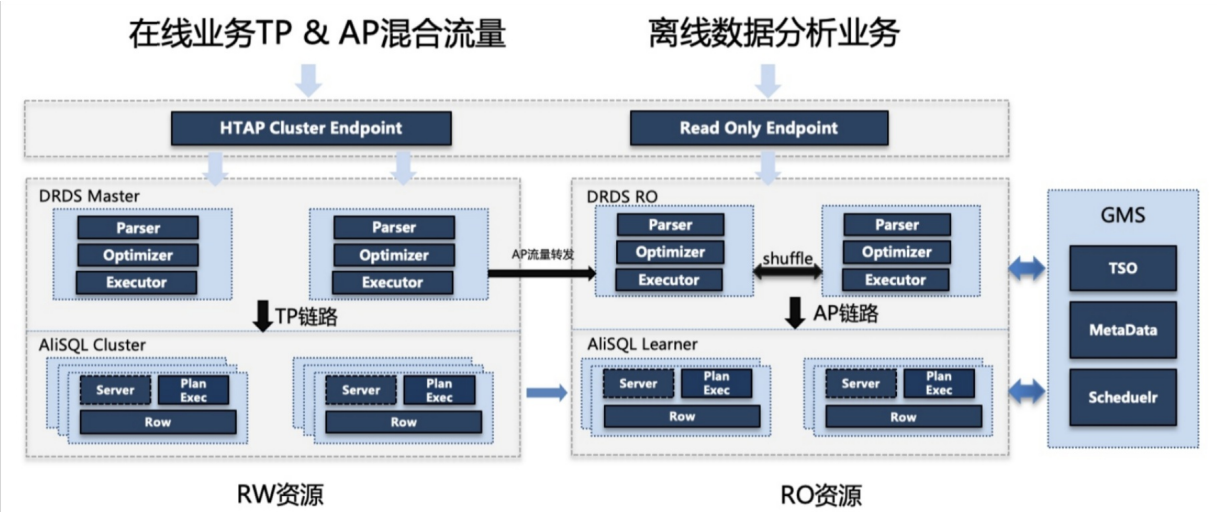
本文主要提供数据库上云后OLTP+OLAP一体化架构的最佳实践，既HTAP。主要面对的业务应用范围：

1. 混合型HTAP数据库需求：如ORACLE数据库改造上云，云上数据库方案选型；
2. OLTP系统查询慢，存在分析型场景和瓶颈的客户；
3. 读写分离需求。

PolarDB-X HTAP架构更多信息，请参见混合负载HTAP。

HTAP集群

您购买的PolarDB-X主实例，主要面向在线通用业务场景。如果业务针对同一份数据有分析、专注离线拖数、跑批等场景，您可以在PolarDB-X主实例上购买多个只读实例。



业务如果有在线HTAP混合流量或者读写分离的需求，推荐使用集群地址。PolarDB-X内部会基于智能路由或者读写权重将部分流量转发给只读实例；业务上只有离线数据分析需求时，推荐使用只读地址，只读地址会直接访问只读实例，只读地址的流量会采用MPP加速。关于连接地址信息，请参见集群地址和只读地址。

路由

智能路由

PolarDB-X优化器会基于代价分析出查询物理扫描行数、CPU、内存、IO、网络等核心资源消耗量，将请求区分为TP与AP负载。当您在集群地址上开启了智能路由，会主动识别SQL的工作负载类型来做路由，比如将识别为AP负载的流量路由给只读实例。您可以通过explain cost指令查看SQL工作负载类型的识别情况。例如以下查询，该查询涉及到物理扫描行数rowcount很小，计算资源（CPU&Memory）也消耗比较少，所以这个查询被识别为TP负载。

```
mysql> explain cost  select a.k, count(*) cnt from sbtest1 a, sbtest1 b where a.id = b.k and a.id > 1000 gro
up by k having cnt > 1300 order by cnt limit 5, 10;
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
------------------------------------------------------------+
| TopN(sort="cnt ASC", offset=?2, fetch=?3): rowcount = 1.0, cumulative cost = value = 2.8765038E7, cpu = 37
.0, memory = 64.0, io = 3.0, net = 5.75, id = 163602178
|
|   Filter(condition="cnt > ?1"): rowcount = 1.0, cumulative cost = value = 2.8765026E7, cpu = 26.0, memory
= 47.0, io = 3.0, net = 5.75, id = 163602177
|
|     HashAgg(group="k", cnt="COUNT()"): rowcount = 1.0, cumulative cost = value = 2.8765025E7, cpu = 25.0,
memory = 47.0, io = 3.0, net = 5.75, id = 163602171
|
|       BKAJoin(condition="k = id", type="inner"): rowcount = 1.0, cumulative cost = value = 2.8765012E7, cp
u = 12.0, memory = 18.0, io = 3.0, net = 5.75, id = 163602169
|
|         Gather(concurrent=true): rowcount = 1.0, cumulative cost = value = 2.3755003E7, cpu = 3.0, memory
= 0.0, io = 1.0, net = 4.75, id = 163602164
|
|           LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `id`, `k` FROM
`sbtest1` AS `sbtest1` WHERE (`id` > ?)"): rowcount = 1.0, cumulative cost = value = 2.3755002E7, cpu = 2.0,
memory = 0.0, io = 1.0, net = 4.75, id = 163601451         |
|         Gather(concurrent=true): rowcount = 1.0, cumulative cost = value = 5003.0, cpu = 3.0, memory = 0.0
, io = 1.0, net = 0.0, id = 163602167
|
|           LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `k` FROM `sbtes
t1` AS `sbtest1` WHERE ((`k` > ?) AND (`k` IN (...)))"): rowcount = 1.0, cumulative cost = value = 5002.0, c
pu = 2.0, memory = 0.0, io = 1.0, net = 0.0, id = 163601377 |
|
| WorkloadType: TP
|
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
------------------------------------------------------------+
```

工作负载的识别，对于HTAP路由至关重要。这里也允许您通过HINT WORKLOAD_TYPE指定工作负载。同样以上述查询为例，可以将查询负载强制指定为AP。

```
mysql> explain cost /*+TDDL:WORKLOAD_TYPE=AP*/ select a.k, count(*) cnt from sbtest1 a, sbtest1 b where a.id
= b.k and a.id > 1000 group by k having cnt > 1300 order by cnt limit 5, 10;
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
------------------------------------------------------------+
| TopN(sort="cnt ASC", offset=?2, fetch=?3): rowcount = 1.0, cumulative cost = value = 2.8765038E7, cpu = 37
.0, memory = 64.0, io = 3.0, net = 5.75, id = 163602178
|
|   Filter(condition="cnt > ?1"): rowcount = 1.0, cumulative cost = value = 2.8765026E7, cpu = 26.0, memory
= 47.0, io = 3.0, net = 5.75, id = 163602177
|
|     HashAgg(group="k", cnt="COUNT()"): rowcount = 1.0, cumulative cost = value = 2.8765025E7, cpu = 25.0,
memory = 47.0, io = 3.0, net = 5.75, id = 163602171
|
|       BKAJoin(condition="k = id", type="inner"): rowcount = 1.0, cumulative cost = value = 2.8765012E7, cp
u = 12.0, memory = 18.0, io = 3.0, net = 5.75, id = 163602169
|
|         Gather(concurrent=true): rowcount = 1.0, cumulative cost = value = 2.3755003E7, cpu = 3.0, memory
= 0.0, io = 1.0, net = 4.75, id = 163602164
|
|           LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `id`, `k` FROM
`sbtest1` AS `sbtest1` WHERE (`id` > ?)"): rowcount = 1.0, cumulative cost = value = 2.3755002E7, cpu = 2.0,
memory = 0.0, io = 1.0, net = 4.75, id = 163601451         |
|         Gather(concurrent=true): rowcount = 1.0, cumulative cost = value = 5003.0, cpu = 3.0, memory = 0.0
, io = 1.0, net = 0.0, id = 163602167
|
|           LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `k` FROM `sbtes
t1` AS `sbtest1` WHERE ((`k` > ?) AND (`k` IN (...)))"): rowcount = 1.0, cumulative cost = value = 5002.0, c
pu = 2.0, memory = 0.0, io = 1.0, net = 0.0, id = 163601377 |
|
| WorkloadType: AP
|
|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
------------------------------------------------------------+
```

规则路由

除了基于代价的智能路由以外，我们也支持基于读写规则的路由。允许您在控制台参数管理上设置读写分离权重参数MASTER_READ_WEIGHT，默认值为100，可配置值区间[0, 100]。如果配置为Weight=60，意味着占60%的流量会继续在主实例执行，40%的剩余流量会路由到只读实例执行，如果只读实例有多个会进行自动分配。

智能路由和规则路由这两者关系是解耦的，具体关系请查看下表格。

智能路由规则	规则路由 (MASTER_READ_WEIGHT)	路由结果
开启	以代价的读写分离为主规则路由建议保持默认值为100	• 事务和写操作流量，全部路由给主实例； • 识别为AP的查询流量，全部路由给只读实例； • 识别为TP的查询流量，按照（100-MASTER_READ_WEIGHT）路由给只读实例。
关闭	以规则的读写分离为主规则路由的可选范围：[0-100]	• 事务和写操作流量，全部路由给主实例； • 识别为TP/AP的查询流量，一律按照（100-MASTER_READ_WEIGHT）路由给只读实例。

执行模式

目前PolarDB-X支持了三种执行模式：

- 单机单线程（TP_LOCAL）：查询过程中，是单线程计算，TP负载的查询涉及到的扫描行数比较少，往往会采样这种执行模式，比如基于主键的点查。
- 单机并行（AP_LOCAL）：查询过程中，会利用节点的多核资源做并行计算，您在没有购买只读实例的前提下，针对AP负载的查询，往往会采样这种执行模式，一般也称之为Parallel Query模式。
- 多机并行（MPP）：您若购买了只读实例，针对AP负载的查询，可以协调只读实例上多个节点的多核做分布式多机并行加速。

为了让您可以准确知道执行模式，我们在原有EXPLAIN指令上，扩展出了EXPLAIN PHYSICAL。例如以下查询，通过指令可以查看到当前查询采样的是MPP模式，此外还可以获取到每个执行片段的并发数。

```
mysql> explain physical select a.k, count(*) cnt from sbtest1 a, sbtest1 b where a.id = b.k and a.id > 1000
group by k having cnt > 1300 or
der by cnt limit 5, 10;
+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------+
| PLAN
|
+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------+
| ExecutorType: MPP
|
| The Query's MaxConcurrentParallelism: 2
|
| Fragment 1
|
|     Shuffle Output layout: [BIGINT, BIGINT] Output layout: [BIGINT, BIGINT]
|
|     Output partitioning: SINGLE [] Parallelism: 1
|
|     TopN(sort="cnt ASC", offset=?2, fetch=?3)
|
|   Filter(condition="cnt > ?1")
|
|     HashAgg(group="k", cnt="COUNT()")
|
|       BKAJoin(condition="k = id", type="inner")
|
|         RemoteSource(sourceFragmentIds=[0], type=RecordType(INTEGER_UNSIGNED id, INTEGER_UNSIGNED k))
|
|         Gather(concurrent=true)
|
|           LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `k` FROM `sbtes
t1` AS `sbtest1` WHERE ((`k` > ?) AND (`k` IN (...)))") |
| Fragment 0
|
|     Shuffle Output layout: [BIGINT, BIGINT] Output layout: [BIGINT, BIGINT]
|
|     Output partitioning: SINGLE [] Parallelism: 1 Splits: 16
|
|     LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `id`, `k` FROM `sbtes
t1` AS `sbtest1` WHERE (`id` > ?)")                         |
+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------+
```

工作负载和执行模式有一定的耦合关系，AP工作负载会采用多机并行计算。同样的也允许您通过HINT EXECUTOR_MODE指定执行模式。假如主实例空闲资源很多，可以考虑强制设置为单机或者多机并行模式来加速。

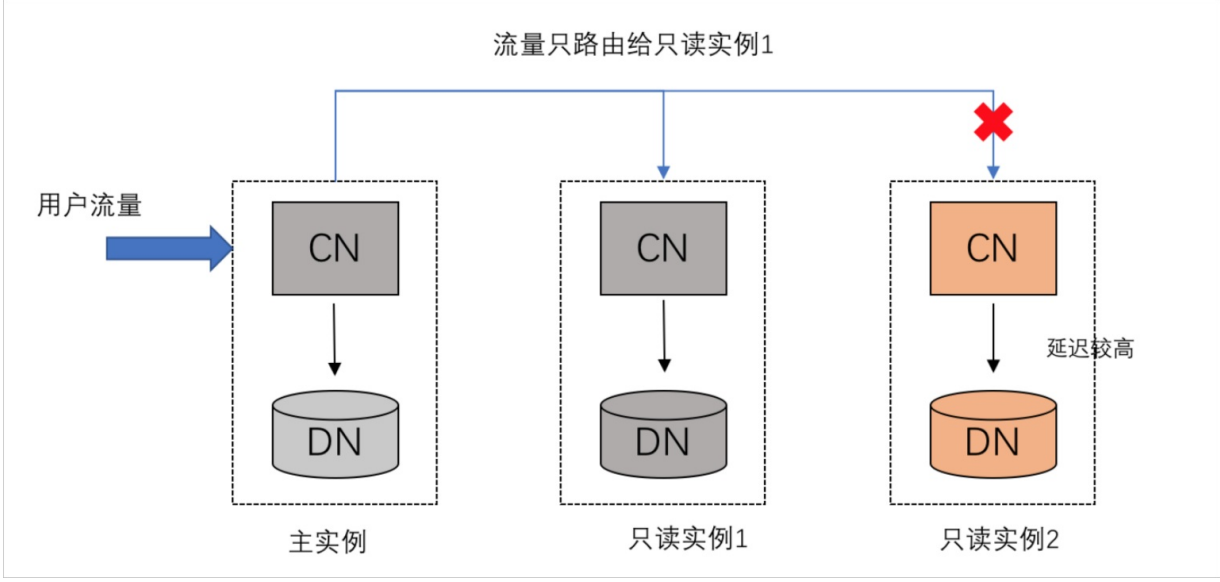
```
mysql> explain physical /*+TDDL:EXECUTOR_MODE=AP_LOCAL*/select a.k, count(*) cnt from sbtest1 a, sbtest1 b w
here a.id = b.k and a.id > 1000 group by k having cnt > 1300 order by cnt limit 5, 10;
|
+-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------+
| ExecutorMode: AP_LOCAL
|
| Fragment 0 dependency: [] parallelism: 4
|
| BKAJoin(condition="k = id", type="inner")
|
|   Gather(concurrent=true)
|
|     LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `id`, `k` FROM `sbtes
t1` AS `sbtest1` WHERE (`id` > ?)")               |
|   Gather(concurrent=true)
|
|     LogicalView(tables="[000000-000003].sbtest1_[00-15]", shardCount=16, sql="SELECT `k` FROM `sbtest1` AS
`sbtest1` WHERE ((`k` > ?) AND (`k` IN (...)))") |
| Fragment 1 dependency: [] parallelism: 8
|
| LocalBuffer
|
|   RemoteSource(sourceFragmentIds=[0], type=RecordType(INTEGER_UNSIGNED id, INTEGER_UNSIGNED k, INTEGER_UNS
IGNED k0))                                        |
| Fragment 2 dependency: [0, 1] parallelism: 8
|
| Filter(condition="cnt > ?1")
|
|   HashAgg(group="k", cnt="COUNT()")
|
|     RemoteSource(sourceFragmentIds=[1], type=RecordType(INTEGER_UNSIGNED id, INTEGER_UNSIGNED k, INTEGER_U
NSIGNED k0))                                      |
| Fragment 3 dependency: [0, 1] parallelism: 1
|
| LocalBuffer
|
|   RemoteSource(sourceFragmentIds=[2], type=RecordType(INTEGER_UNSIGNED k, BIGINT cnt))
|
| Fragment 4 dependency: [2, 3] parallelism: 1
|
| TopN(sort="cnt ASC", offset=?2, fetch=?3)
|
|   RemoteSource(sourceFragmentIds=[3], type=RecordType(INTEGER_UNSIGNED k, BIGINT cnt))
|
+-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------+
```

在多机并行MPP执行模式的并发度是根据物理扫描行数、实例规格和计算所涉及到表的分表数来计算出来的，整体的并行度要考虑高并发场景，所以并行度的计算会偏保守，您可以通过上述EXPLAIN PHYSICAL指令查看并行度。当然也同样支持HINT MPP_PARALLELISM强制指定并行度，

```
/*+TDDL:EXECUTOR_MODE=MPP MPP_PARALLELISM=8*/select a.k, count(*) cnt from sbtest1 a, sbtest1 b where a.id =
b.k and a.id > 1000 group by k having cnt > 1300 order by cnt limit 5, 10;
```

调度策略

假设您购买了多个只读实例并加入到集群地址中，您通过集群地址的查询SQL路由到只读实例的流量，会被均匀调度到只读实例多个节点上执行，调度会考虑各个节点的资源负载，确保各个节点的负载压力差不多。比如PolarDB-X会将只读实例延迟作为调度参考指标，避免将流量调度到延迟较大的只读实例上执行。



反馈机制

基于统计信息智能识别负载存在一定的误差。在负载识别出错的情况下，PolarDB-X能基于查询最终真实扫描的物理行数和执行时间，重新修正工作负载，这个过程我们称之为负载的自适应反馈。根据执行计划管理一文，PolarDB-X的执行计划会被管理起来，同样其工作负载也会被管理起来，用户可以通过以下语句，查看执行计划管理中各个计划的工作负载。

```
baseline [Select Statemtnt]
```

比如当一条TP负载的简单查询，执行时间和真实扫描行数都大于阈值，这个查询就会被修正为AP工作负载，写入到执行计划管理状态中，同理反之。除了上述提到了自适应反馈能力以外，我们也可以通过以下语句，手动修正计划管理中的工作负载。

```
baseline fix sql /*+TDDL:WORKLOAD_TYPE=AP*/ [Select Statemtnt]
```

执行计划管理中的计划负载被合理修正后，同类型的查询会从执行计划管理中获取正确的工作负载标记。

一致性读

从集群地址和只读地址一文可了解，业务流量直连只读地址，提供了可配置的全局一致性读能力。业务流量是通过集群地址，路由给只读实例，默认就会开启全局一致性读能力。全局一致性读机制，可以确保业务在主实例数据写入成功后就一定可以在只读库读到写入的数据，避免因传统读写分离架构，数据复制延迟带来的数据写后读不一致问题。

如果业务使用的集群地址，对数据一致性读并没有强烈诉求，例如业务可以忽略当前只读实例复制延迟带来的数据影响，我们也支持在参数管理页面上配置ENABLE_CONSISTENT_REPLICA_READ，或者通过HINT对某个查询关闭一致性读能力。

```
/*+TDDL:ENABLE_CONSISTENT_REPLICA_READ=false*/ [Select Statemtnt]
```

常见问题

- Q： 在集群地址中开启了智能路由后，是否还有必要配置MASTER_READ_WEIGHT的规则路由？

  A： PolarDB-X智能路由主要的机制是将查询SQL中偏AP的复杂查询路由到只读实例中来减轻主实例的压力，如果此时TP简单查询并发很高，占用了主实例比较大的资源，也可以额外开启规则路由，将部分TP流量分摊到只读实例中。

- Q： 集群地址的读写分离，是否还兼容传统按比例的读写模式，对应的区别点和优势分别是什么？

  A： PolarDB-X支持智能路由和规则路由两种模式，基于规则路由可以兼容传统的读写分离模式。PolarDB-X的读写分离优势在于引入了一致性读的机制，可以很好的规避只读实例复制延迟带来的数据写后读的查询一致性问题。

- Q： 智能路由识别的SQL负载类型，是否有办法查询，如果识别不正确的情况下对应的影响是什么，以及如何修正？

  A： PolarDB-X可以通过执行计划管理的功能查询历史执行过的SQL模板和执行计划，针对智能路由自身会基于实际执行后的反馈来对SQL负载类型进行修正。同时也允许业务通过执行计划管理的方式，基于hint的方式强制设置工作负载类型来做修改。

- Q： 基于集群地址的HTAP模式，和传统的OLTP + DTS + OLAP组合的解决方案相比，有什么特点和优势？

  A： PolarDB-X的HTAP模式，基于数据库原生的多副本能力，简化了外置DTS同步的运维复杂性和同步成本，同时引入一致性读机制，并结合MPP的多机并行计算能力，可以很好满足业务对数据在线计算的实时性和扩展性的要求。

9.19. 如何支持超大事务

本文档介绍了PolarDB-X对超大事务的支持情况。

在分布式数据库中，超大事务一般满足以下条件中的一个或多个：

1. 事务修改的数据涉及多个分片；
2. 事务修改的数据量比较大；
3. 事务执行的SQL语句比较多。

下面通过以下场景的测试，以这三个维度为切入点，介绍PolarDB-X对超大事务的支持情况。

测试所用实例规格

使用以下PolarDB-X实例进行测试：

PolarDB-X版本	polarx-kernel_5.4.11-16301083_xcluster-20210805
节点规格	4核16GB
节点个数	4

创建一个规格为4C16G的ECS连接实例进行测试，该ECS与实例位于同一网段下，由同一个虚拟交换机连接。

执行以下命令，创建如下表：

```
CREATE TABLE `tb` (
    `id` bigint(20) NOT NULL,
    `c` longblob/char(255)/char(1),
    PRIMARY KEY (`id`)
);
```

该表只有两列，一列为 `id` （bigint类型，主键），一列为 `c` （在不同的场景下，分别为longblob/char(255)/char(1)类型）。针对分片数，主要考虑以下三种设定：

- 1分片，即单库单表的情形，PolarDB-X对于单分片上的事务提交会优化为一阶段提交；
- 8分片，以id为拆分键对上述实验表拆分成8个分片，每个存储节点（DN）2 个分片；
- 16分片，以id为拆分键对上述实验表拆分成16个分片，每个存储节点（DN）4 个分片；

场景一

在该场景下，执行SQL语句数量中等，每条语句携带数据量较大，事务写入数据量较大。

执行SQL语句数量	2048
每条语句携带的数据量	约256KB ~ 8MB
数据修改总量	512MB ~ 16GB
数据修改条数	2048

测试过程

测试中，每一条SQL语句形如：

```
INSERT INTO `tb` VALUES (id, c)
```

数据表中 `c` 列的数据类型为longblob， `c` 的大小从256KB到8MB不等。即每个事务写入2048条数据时，数据写入总量从512MB到16GB不等。实际的写入量比这个值稍大，因为bigint类型的id也占用一定的空间。

> **注意** 受计算节点（CN）的参数“MAX_ALLOWED_PACKET”的影响，每个查询的请求包大小不应超过“MAX_ALLOWED_PACKET”个字节。该值默认为16MB，可通过参数设置修改，即每条SQL语句携带的数据量应小于参数“MAX_ALLOWED_PACKET”。如果您通过JDBC连接PolarDB-X数据库，在发送的包大于“MAX_ALLOWED_PACKET”个字节时，会收到“CommunicationsException: Communications link failure”的报错。

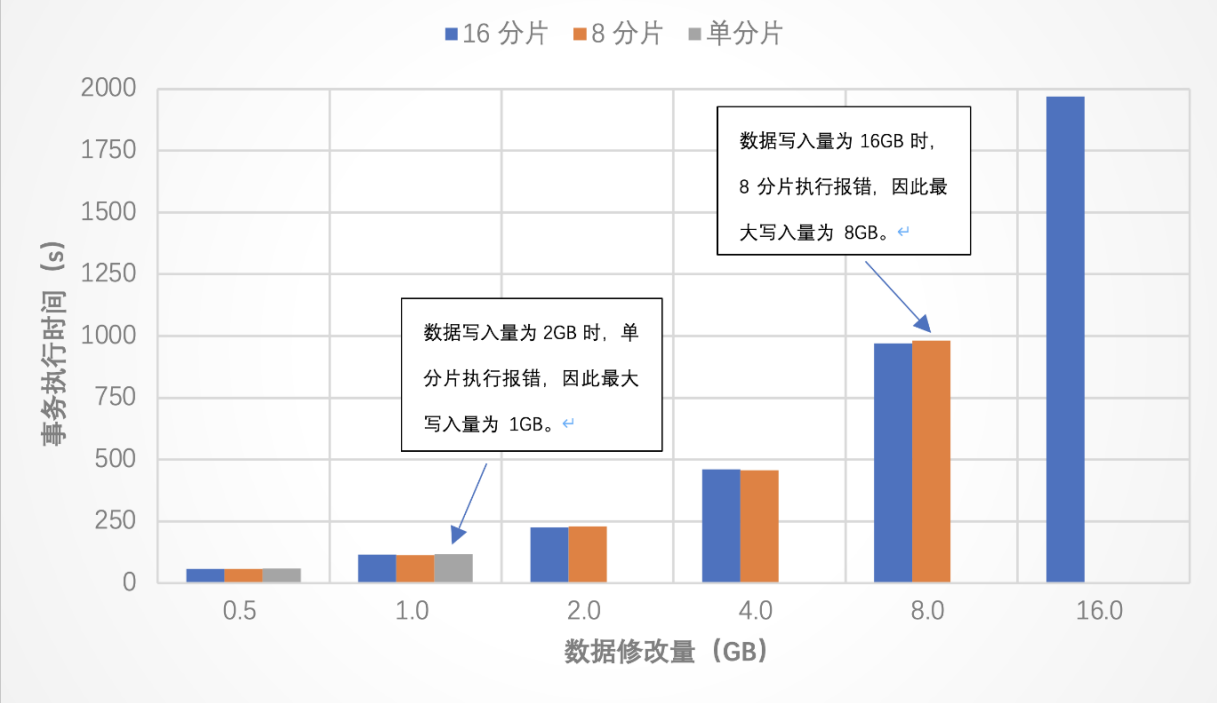
在单表的情况下，当 `c` 的大小为1MB，事务总写入量略大于2GB 时，执行该事务会报错：

```
ERR-CODE: [TDDL-4614][ERR_EXECUTE_ON_MYSQL] Error occurs when execute on GROUP ...: Multi-statement transact
ion required more than 'max_binlog_cache_size' bytes of storage; increase this mysqld variable and try again
```

受DN的参数“MAX_BINLOG_CACHE_SIZE”的限制，事务执行过程中，引起的binlog写入量不应超过该值。对于分布式事务，每个分片对应一个分支事务，每个分支事务引起的binlog写入量不应超过该值。以本测试场景为例，如果事务的每条语句都是INSERT语句，那么每个分片的数据写入量不应超过2GB，事务总的数据写入量不能超过（分片数量×2GB），例如8分片的情况下，写入总量在16GB时也会触发这个报错。但是，上述讨论并不意味着只要您的事务数据写入量小于（分片数量×2GB），就一定能执行成功。

> **注意** DN的参数“MAX_BINLOG_CACHE_SIZE”无法修改，默认值为2147483648，即2GB。实际上，由于DN受MySQL的限制，即使这个参数可以修改，“MAX_BINLOG_CACHE_SIZE”最大也不应超过4GB。

测试结果



测试结论

- 无论是否开启事务，每条SQL语句携带的数据量受CN的参数“MAX_ALLOWED_PACKET”的限制，不能超过该值。
- 对于每个事务，在每个分片上执行的语句所引起的binlog写入量受DN的参数“MAX_BINLOG_CACHE_SIZE”的影响，不能超过该值。例如，在INSERT场景下，binlog需要记录插入的值，则事务对每个分片插入的数据量不应超过该值。
- 如果想支持更大的事务，比如在一个事务中插入更多的数据，请把数据表划分到更多的分片上。

场景二

在该场景下，执行SQL语句数量较少，每条语句携带数据量较小，事务修改数据量较大。

执行SQL语句数量	1
每条语句携带的数据量	约256KB

数据修改总量	256MB ~ 8GB
数据修改条数	2^20 ~ 2^25

测试过程

在本测试中，数据表中 `c` 列的数据类型为char(255)。首先在数据表中导入2^25条数据，数据的 `id` 为0, 1, 2, ..., 2^26-1，每条数据的 `c` 为 "aa...aa"（255个a的字符串）。然后开启事务，执行以下语句：

```
UPDATE `tb` SET c = "bb...bbb" (255 个 b 的字符串) where id < ?
```

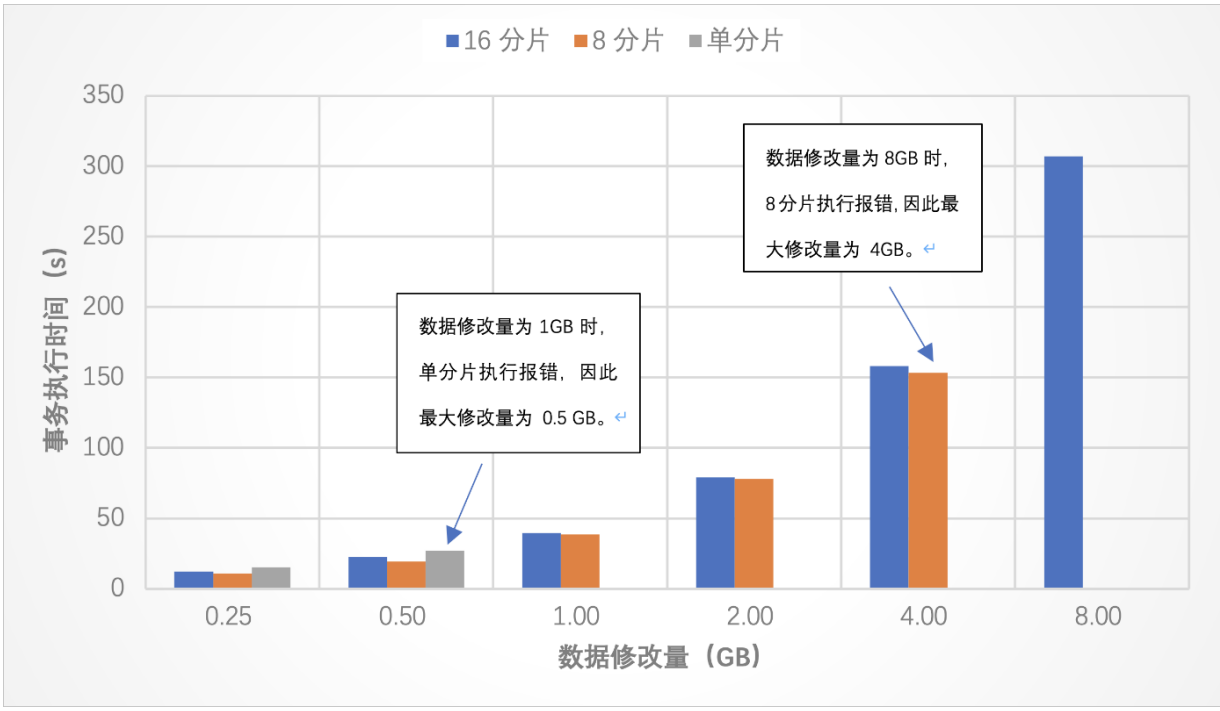
假设语句中的 `?` 为 `x`，则会修改共 `x` 条数据，每条数据修改约256字节的数据，数据总修改量为 `x * 2^8` 字节。本测试中 `x` 取值从2^20到2^25不等，数据修改总量从512MB到8GB不等。

在单表的情况下，当 `x` 为2^22，事务修改的数据量约为1GB时，执行该事务会和场景一样报错：

```
ERR-CODE: [TDDL-4614][ERR_EXECUTE_ON_MYSQL] Error occurs when execute on GROUP ...: Multi-statement transact
ion required more than 'max_binlog_cache_size' bytes of storage; increase this mysqld variable and try again
```

对于UPDATE语句，binlog需要记录修改前和修改后的值，即每条数据修改了256字节的数据，binlog需记录至少512字节的数据。在单表情况下，事务修改量约为1GB时，引起binlog写入量超过了2GB，就触发了这个报错。同理，对于分布式事务，如果事务执行的都是UPDATE语句，那么每个分片上的数据修改量不应超过1GB。例如，在本测试中，8分片的场景下，在一个事务中UPDATE的数据为8GB时，也会触发此报错。

测试结果



测试结论

- 与场景一类似，对于每个事务，在每个分片上所引起的binlog写入量受DN的参数“MAX_BINLOG_CACHE_SIZE”的影响，不能超过该值。例如，在事务语句全为UPDATE的场景下，binlog需要记录修改前的值和修改后的值，则事务在每个分片上修改的数据量不应超过该值的一半。在本场景下，单分片的事务修改的数据量最多不应超过1GB，8分片的事务修改的数据量最多不应超过4GB。
- 如果想支持更大的事务，比如在一个事务中修改更多的数据，请把数据表划分到更多的分片上。

场景三

在该场景下，执行SQL语句数量较多，每条语句携带数据量较小，事务写入数据量较小。

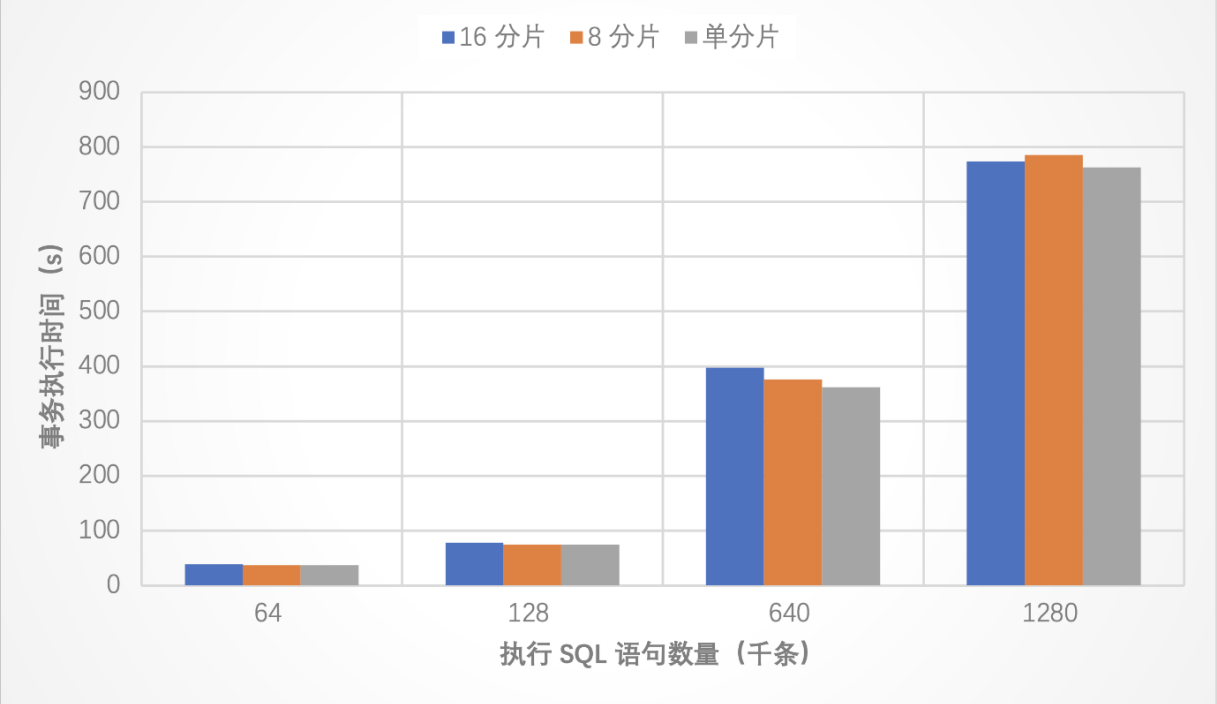
执行SQL语句数量	64,000 ~ 1,280,000
每条语句携带的数据量	若干字节
数据修改总量	32B
数据修改条数	32

测试过程

在本测试中，数据表中 `c` 列的数据类型为char(1)。首先往数据表插入32条数据，数据的 `id` 为0 ~ 31， `c` 为 "a"。然后开启事务，反复执行以下语句 `x` 次，即每个事务会执行 `64x` 条SQL:

```
UPDATE `tb` SET c = "b" where id = 0;
UPDATE `tb` SET c = "a" where id = 0;
UPDATE `tb` SET c = "b" where id = 1;
UPDATE `tb` SET c = "a" where id = 1;
...
UPDATE `tb` SET c = "b" where id = 31;
UPDATE `tb` SET c = "a" where id = 31;
```

测试结果



测试结论

本场景下，一个事务内执行128万条SQL仍未达到瓶颈，不同分片设定下都能正常执行。因此，在多数情况下，首先触发事务瓶颈的更可能是事务数据写入、删除、修改量。

场景四

在该场景下，我们测试分片数对事务的性能影响。

执行SQL语句数量	1
每条语句携带的数据量	若干字节
数据修改总量	8KB ~ 8MB

数据修改条数	8K ~ 8M
分片数	1 ~ 2048

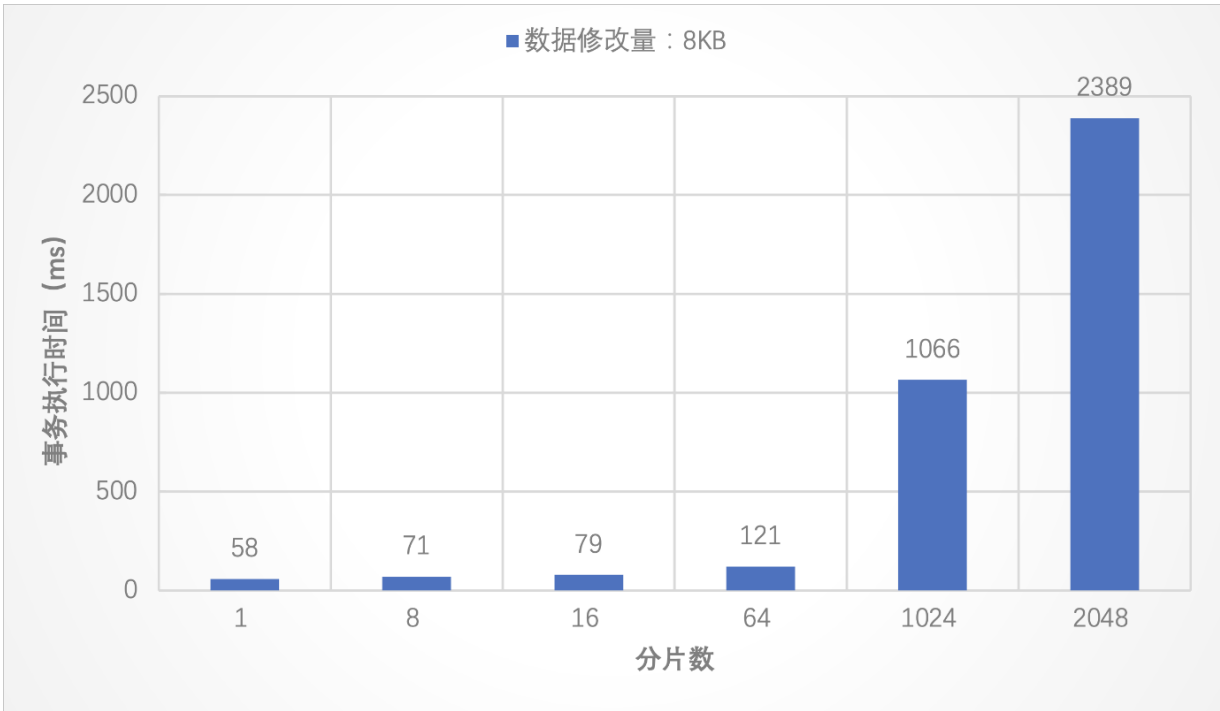
测试过程

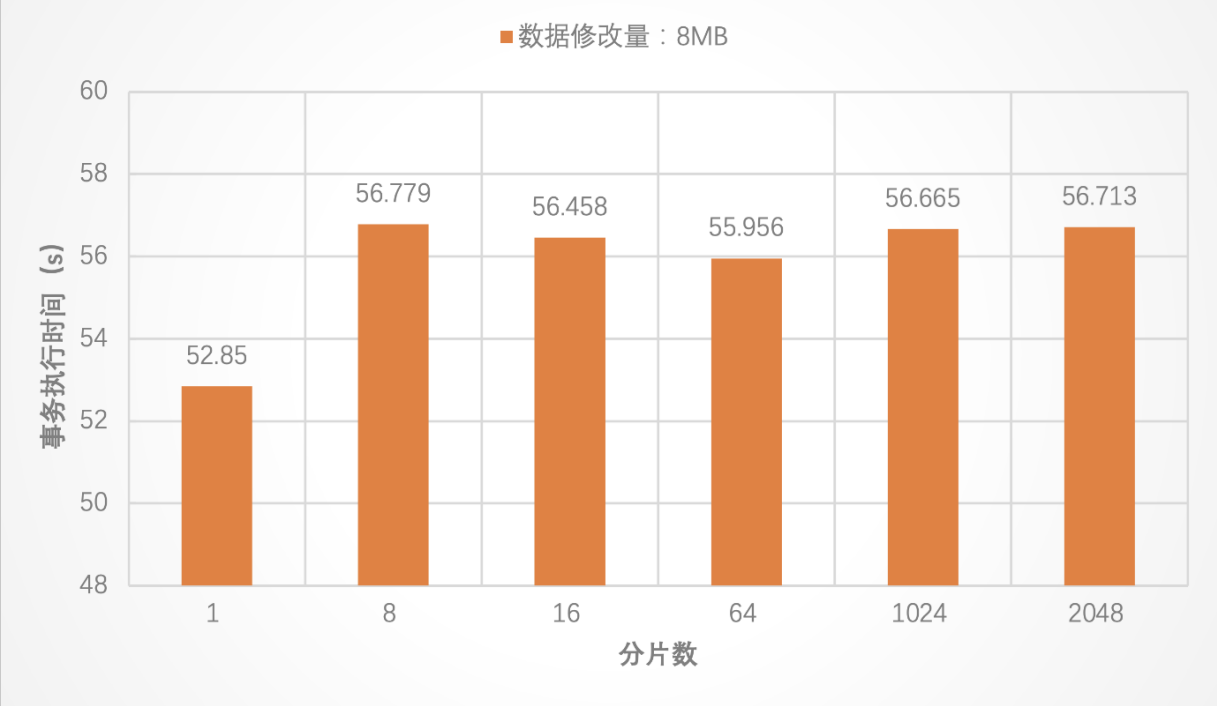
在本场景中，数据表中 `c` 列的数据类型为char(1)。首先往数据表插入x条数据，数据的 `id` 为0 ~ x，`c` 为 "a"。然后开启事务，执行以下语句，即每个事务会修改x条记录，修改x字节数据：

```
UPDATE `tb` SET c = "b" where id < x
```

每个事务执行一次该SQL，数据修改量为x字节，数据修改条数为x。考虑x=8K和8M的情况。

测试结果





测试结论

- 当数据修改量在8KB时，由于事务修改的数据量较小，SQL语句的执行时间较短，事务的执行时间受事务提交时间的影响较大。在该场景下，事务执行时间随分片数增多而增大。具体而言，在分片数为1～64时，当分片数增长为原来的n倍时，事务执行的时间比原来的n倍少得多。这是因为分片数较少时，虽然事务提交时间有一定的影响，但事务内DML语句的执行也占用了较多的时间。而当分片数在1024～2048的情况下，事务执行时间则基本全部由事务提交时间构成。相比之下，DML语句的执行时间可以忽略不计。因此，此时事务执行时间与分片数近似成倍数增加的关系。
- 当数据修改量在8MB时，事务执行时间则不再随分片数增多而明显增大，这时事务执行时间主要由DML的执行时间组成，事务提交时间可以忽略不计。
- 总而言之，根据前文的结论，当要支持更大的事务时，建议将表划分到更多的分片上。但分片数越多，事务提交的时间也会越长。在多分片小事务的场景下，事务执行时间甚至主要由事务提交时间组成。因此，您可以参考本文在不同场景下的结论，根据具体的业务场景，选择合适的分片数，以获取更好的事务性能。

场景五

该场景下测试CN的计算压力较大的时候，对事务的影响。

测试过程

在本场景中，把逻辑表tb划分到16个分片进行实验。数据表中，`c`列的数据类型为char(255)。首先往数据表插入2^26条数据，数据的`id`为0～2^26-1，`c`和`id`相同。接着，创建另一个表tb2，表结构和数据与tb完全一致。此时，每个表的大小约为16GB。最后，创建一个临时表tmp，其表结构和tb也相同，然后开启事务，执行下面的语句：

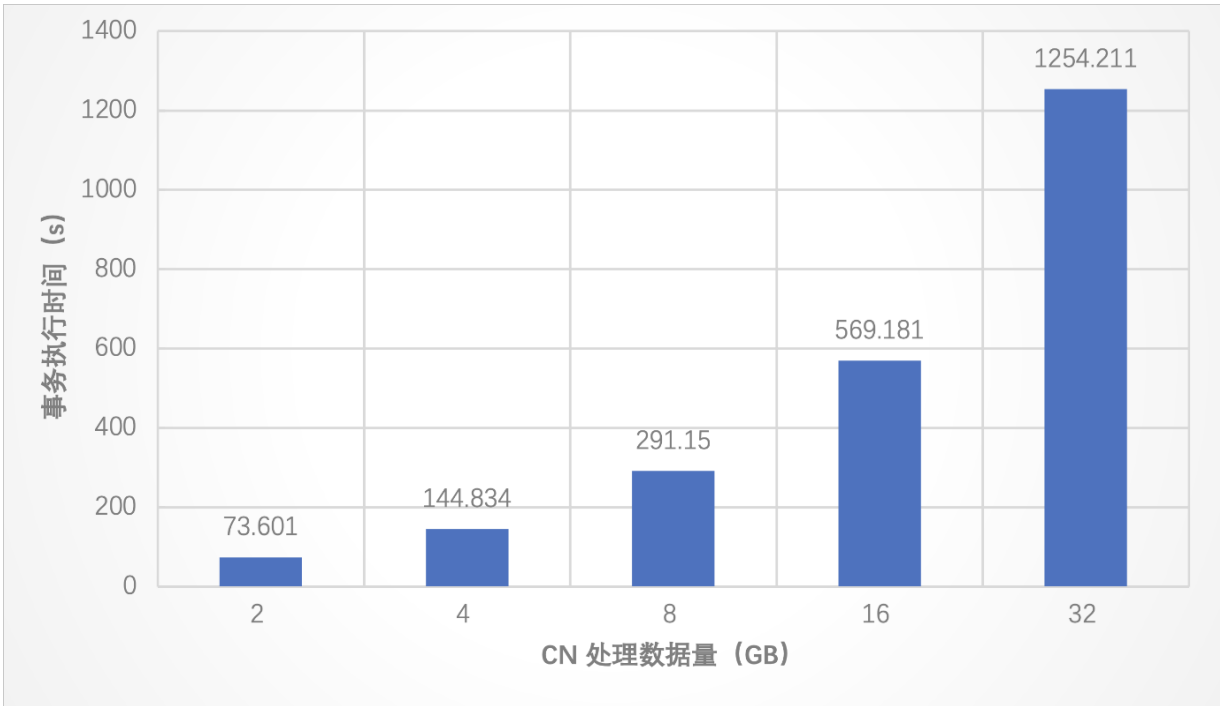
```
INSERT INTO tmp
    SELECT tb.id, tb.c
  FROM tb, tb2
  WHERE tb.c = tb2.c AND tb.id > x AND tb2.id > x
```

通过explain可以看到这条SQL的执行计划：

```
LogicalInsert(table="tmp", columns=RecordType(BIGINT id, CHAR c))
  Project(id="id", c="c")
    HashJoin(condition="c = c", type="inner")
      Gather(concurrent=true)
        LogicalView(tables="[000000-000015].tb_VjuI", shardCount=16, sql="SELECT `id`, `c` FROM `tb` AS `tb`
WHERE (`id` > ?)")
      Gather(concurrent=true)
        LogicalView(tables="[000000-000015].tb2_IfrZ", shardCount=16, sql="SELECT `c` FROM `tb2` AS `tb2` WH
ERE (`id` > ?)")
```

即把两个表id>x的部分拉到CN做hash join。通过x来控制拉取的数据量，即CN需要计算的数据量。当x=0，会拉取所有的数据到CN做hash join，此时，CN总共需要处理约32GB的数据。值得一提的是，单个CN节点的内存只有16GB。

测试结果



测试结论

单个事务导致CN端的计算压力较大、计算量较多时（比如事务内有复杂的join且这些join无法下推，必须要在CN处理的场景），PolarDB-X对大事务的支持仍然稳定。具体表现为，在需要处理的数据量呈倍数增长时，事务执行不报错，且执行时间也呈线性增长。

总结

- 无论是否开启事务，每条SQL语句自身携带的数据量受CN的参数“MAX_ALLOWED_PACKET”的限制，不能超过该值。
- 对于每个事务，在每个分片上执行的语句所引起的binlog写入量受DN的参数“MAX_BINLOG_CACHE_SIZE”的影响，不能超过该值。
  - 该参数值默认为2GB，受MySQL的限制，最大值为4GB，以INSERT为例，如果一个事务中全是INSERT语句，由于binlog需要记录插入的值，数据写入总量最大不超过：分片数×4GB。
  - 如果想支持更大的事务，比如在一个事务中插入/删除/修改更多的数据，请把数据表划分到更多的分片上。
- 在事务修改数据量较少的情况下，分片数对事务执行时间的影响较大；在事务修改数据量较多的情况下，分片数对事务执行时间的影响则不大。因此，您可以参考本文在不同场景下的结论，根据具体的业务场景，选择合适的分片数，以获取更好的事务性能。
- 单个事务导致CN端的计算压力较大、计算量较多时，PolarDB-X对大事务的支持仍然稳定。

10.错误码

本文档列出了PolarDB-X返回的常见错误码及解决方法。

PXC-1305 ERR_UNKNOWN_SAVEPOINT

PXC-1094 ERR_UNKNOWN_THREAD_ID

PXC-4006 ERR_TABLE_NOT_EXIST

PXC-4007 ERR_CANNOT_FETCH_TABLE_META

PXC-4018 ERR_INVALID_DDL_PARAMS

PXC-4100 ERR_ATOM_NOT_AVALILABLE

PXC-4101 ERR_ATOM_GET_CONNECTION_FAILED_UNKNOWN_REASON

PXC-4102 ERR_ATOM_GET_CONNECTION_FAILED_KNOWN_REASON

PXC-4103 ERR_ATOM_CONNECTION_POOL_FULL

PXC-4104 ERR_ATOM_CREATE_CONNECTION_TOO_SLOW

PXC-4105 ERR_ATOM_ACCESS_DENIED

PXC-4106 ERR_ATOM_DB_DOWN

PXC-4108 ERR_VARIABLE_CAN_NOT_SET_TO_NULL_FOR_NOW

PXC-4200 ERR_GROUP_NOT_AVALILABLE

PXC-4201 ERR_GROUP_NO_ATOM_AVALILABLE

PXC-4202 ERR_SQL_QUERY_TIMEOUT

PXC-4203 ERR_SQL_QUERY_MERGE_TIMEOUT

PXC-4400 ERR_SEQUENCE

PXC-4401 ERR_MISS_SEQUENCE

PXC-4403 ERR_MISS_SEQUENCE_TABLE_ON_DEFAULT_DB

PXC-4404 ERR_SEQUENCE_TABLE_META

PXC-4405 ERR_INIT_SEQUENCE_FROM_DB

PXC-4407 ERR_OTHER_WHEN_BUILD_SEQUENCE

PXC-4408 ERR_SEQUENCE_NEXT_VALUE

PXC-4500 ERR_PARSER

PXC-4501 ERR_OPTIMIZER

PXC-4502 ERR_OPTIMIZER_MISS_ORDER_FUNCTION_IN_SELECT

PXC-4504 ERR_OPTIMIZER_SELF_CROSS_JOIN

PXC-4506 ERR_MODIFY_SHARD_COLUMN

PXC-4508 ERR_OPTIMIZER_NOT_ALLOWED_SORT_MERGE_JOIN

PXC-4509 ERR_OPTIMIZER_ERROR_HINT

PXC-4510 ERR_CONTAINS_NO_SHARDING_KEY

PXC-4511 ERR_INSERT_CONTAINS_NO_SHARDING_KEY

PXC-4515 ERR_CONNECTION_CHARSET_NOT_MATCH

PXC-4517 ERR_MODIFY_SYSTEM_TABLE

PXC-4520 ERR_DML_WITH_SUBQUERY

PXC-4521 ERR_INSERT_SHARD

PXC-4523 ERR_TODNF_LIMIT_EXCEED

PXC-4524 ERR_TOCNF_LIMIT_EXCEED

PXC-4526 ERR_FUNCTION_NOT_FOUND

PXC-4527 ERR_MODIFY_SHARD_COLUMN_ON_TABLE_WITHOUT_PK

PXC-4595 ERR_UNKNOWN_TZ

PXC-4600 ERR_FUNCTION

PXC-4602 ERR_CONVERTOR

PXC-4603 ERR_ACCROSS_DB_TRANSACTION

PXC-4604 ERR_CONCURRENT_TRANSACTION

PXC-4606 ERR_QUERY_CANCLED

PXC-4610 ERR_CONNECTION_CLOSED

PXC-4612 ERR_CHECK_SQL_PRIV

PXC-4614 ERR_EXECUTE_ON_MYSQL

PXC-4616 ERR_UNKNOWN_DATABASE

PXC-4620 ERR_FORBID_EXECUTE_DML_ALL

PXC-4633 ERR_DB_STATUS_READ_ONLY

PXC-4707 ERR_OUT_OF_MEMORY

PXC-4709 ERR_IVENTORY_HINT_NOT_SUPPORT_CROSS_SHARD

PXC-4994 ERR_FLOW_CONTROL

PXC-4998 ERR_NOT_SUPPORT

PXC-5001 ERR_TRANS

PXC-5002 ERR_TRANS_UNSUPPORTED

PXC-5003 ERR_TRANS_LOG

PXC-5006 ERR_TRANS_COMMIT

PXC-5008 ERR_TRANS_TERMINATED

PXC-5010 ERR_TRANS_CONTINUE_AFTER_WRITE_FAIL

PXC-5108 ERR_CHECK_PRIVILEGE_FAILED_ON_TABLE

PXC-5119 ERR_FILE_CANNOT_BE_CREATE

PXC-5302 ERR_GLOBAL_SECONDARY_INDEX_UNSUPPORTED

PXC-5306 ERR_GLOBAL_SECONDARY_INDEX_INSERT_DUPLICATE_VALUES

PXC-5308 ERR_GLOBAL_SECONDARY_INDEX_MODIFY_UNIQUE_KEY

PXC-5310 ERR_GLOBAL_SECONDARY_INDEX_ONLY_SUPPORT_XA

PXC-5313 ERR_GLOBAL_SECONDARY_INDEX_MODIFY_GSI_TABLE_WITH_DDL

PXC-5316 ERR_GLOBAL_SECONDARY_INDEX_INDEX_AND_SHARDING_COLUMNS_NOT_MATCH

PXC-5317 ERR_GLOBAL_SECONDARY_INDEX_CONTINUE_AFTER_WRITE_FAIL

PXC-5321 ERR_GLOBAL_SECONDARY_INDEX_BACKFILL_DUPLICATE_ENTRY

PXC-8007 ERR_ABANDONED_TASK

PXC-8008 ERR_EXECUTE_SPILL

PXC-8011 ERR_OUT_OF_SPILL_SPACE

PXC-8012 ERR_OUT_OF_SPILL_FD

PXC-8102 ERR_PAGE_TOO_LARGE

PXC-8103 ERR_NO_NODES_AVAILABLE

PXC-9301 ERR_DUPLICATED_PARTITION_NAME

PXC-9305 ERR_PARTITION_NAME_NOT_EXISTS

PXC-10004 ERR_X_PROTOCOL_RESULT

PXC-1305 ERR_UNKNOWN_SAVEPOINT

- 描述：指定名称的SAVEPOINT不存在。
- 示例：`ERR-CODE: [PXC-1305][ERR_UNKNOWN_SAVEPOINT] SAVEPOINT ***** does not exist`
- 说明：在PolarDB-X上执行`ROLLBACK TO SAVEPOINT`或者`RELEASE SAVEPOINT`命令时，如果指定的SAVEPOINT名称不存在，会提示PXC-1305错误。

  建议检查SAVEPOINT命令返回的名称是否和使用的名称一致。

PXC-1094 ERR_UNKNOWN_THREAD_ID

- 描述：KILL命令指定的会话ID不存在。
- 示例：`ERR-CODE: [PXC-1094][ERR_UNKNOWN_THREAD_ID] Unknown thread id: *****`
- 说明：在PolarDB-X上执行`KILL`命令取消执行的SQL语句时，如果指定的会话ID不存在，或者对应的SQL语句已经结束执行，会提示PXC-1094错误。

  建议使用`SHOW PROCESSLIST`命令查看正在执行的SQL语句会话ID，并只针对返回的ID执行`KILL`命令。

PXC-4006 ERR_TABLE_NOT_EXIST

- 描述：PolarDB-X数据表不存在。
- 示例：`ERR-CODE: [PXC-4006][ERR_TABLE_NOT_EXIST] Table '*****' doesn't exist.`
- 说明：该错误码表示PolarDB-X数据表不存在，或者由于未知原因，PolarDB-X无法加载数据表的元数据信息。

PXC-4007 ERR_CANNOT_FETCH_TABLE_META

- 描述：PolarDB-X无法加载数据表的元数据信息。
- 示例：`ERR-CODE: [PXC-4007][ERR_CANNOT_FETCH_TABLE_META] Table '*****' metadata cannot be fetched because Table '*****.*****' doesn't exist.`
- 说明：该错误码表示PolarDB-X尝试读取数据表的元数据信息失败。可能的错误原因如下：
  - 数据表未创建。
  - 维护的元数据库信息不一致。
  - 表被删除或者改名。

  出现该错误时，首先检查表名是否存在，或者执行`check table`命令确认PolarDB-X维护的元数据库信息是否一致。如果确定表被人为删除或改名，可以通过PolarDB-X提供的数据恢复功能修复。如果仍无法修复，请联系技术支持。

PXC-4018 ERR_INVALID_DDL_PARAMS

- 描述：PolarDB-X执行DDL过程失败。
- 示例：`ERR-CODE: [PXC-4018][ERR_INVALID_DDL_PARAMS] invalid '*****'.`
- 说明：该错误表示用户执行DDL的参数有错误，请检查参数正确性。如果确认参数正确，请联系技术支持。

PXC-4100 ERR_ATOM_NOT_AVALILABLE

- 描述：PolarDB-X后端数据节点暂时不可用。
- 示例：`ERR-CODE: [PXC-4100][ERR_ATOM_NOT_AVALILABLE] Atom : ***** isNotAvailable`
- 说明：如果PolarDB-X探测到后端某个数据节点状态异常，会临时阻止访问该实例并提示PXC-4100错误。

  当遇到该错误，请检查PolarDB-X后端所有数据节点是否异常。当后端数据节点从异常状态恢复后，PolarDB-X将自动解除不可用状态，恢复应用正常访问。

PXC-4101 ERR_ATOM_GET_CONNECTION_FAILED_UNKNOWN_REASON

- 描述：PolarDB-X计算节点和数据节点连接获取失败。
- 示例：`ERR-CODE: [PXC-4101][ERR_ATOM_GET_CONNECTION_FAILED_UNKNOWN_REASON] Get connection for db '*****' from pool failed. AppName:*****, Env:*****, UnitName:null. Message from pool: wait millis 5000, active 0, maxActive 5. You should look for the following logs which contains the real reason.`

- 说明：PolarDB-X在处理请求时会向数据节点异步创建连接。如果无法在等待时间内完成数据节点连接创建，而异步任务又尚未返回错误原因，PolarDB-X会向应用返回PXC-4101错误。

  该错误通常是由后端数据节点异常导致的。如果排除数据节点问题后仍然出现该错误，请联系技术支持。

PXC-4102 ERR_ATOM_GET_CONNECTION_FAILED_KNOWN_REASON

- 描述：已知原因的PolarDB-X后端连接获取失败。
- 示例：`ERR-CODE: [PXC-4102][ERR_ATOM_GET_CONNECTION_FAILED_KNOWN_REASON] Get connection for db '*****' failed because wait millis 5000, active 0, maxActive 5`
- 说明：PolarDB-X计算节点获取数据节点连接时出错，错误原因已经在ERR-CODE消息中给出。常见PolarDB-X数据节点连接失败的原因如下：
  - 后端数据节点连接数已满
  - 计算节点到数据节点的连接超时
  - 数据节点拒绝连接

  如果排除后端数据节点问题后仍然出现该错误，请联系技术支持。

PXC-4103 ERR_ATOM_CONNECTION_POOL_FULL

- 描述：PolarDB-X后端数据节点连接池已满。
- 示例：`ERR-CODE: [PXC-4103][ERR_ATOM_CONNECTION_POOL_FULL] Pool of DB '*****' is full. Message from pool: wait millis 5000, active 5, maxActive 5. AppName:*****, Env:*****, UnitName:null.`
- 说明：该错误表示PolarDB-X后端连接池已满。导致PXC-4103错误的常见原因如下：
  - 应用SQL语句执行比较慢，占用单个连接的时间过长，导致连接数不够；
  - 应用端没有关闭数据库连接，导致连接泄露；
  - 有很多跨库查询（例如聚合统计类查询，未带分库条件的查询）同时执行，占用大量连接。

  解决方法建议如下：
  - 尽量使用框架访问数据库，如Spring JDBC、MyBatis等；
  - 按性能分析报告与DBA建议优化业务SQL语句；
  - 使用PolarDB-X读写分离将跨库查询转发至读库处理；
  - 升级更高规格的PolarDB-X，提升后端处理能力；
  - 联系技术支持调整PolarDB-X后端连接数。

PXC-4104 ERR_ATOM_CREATE_CONNECTION_TOO_SLOW

- 描述：PolarDB-X后端数据节点连接创建太慢。
- 示例：`ERR-CODE: [PXC-4104][ERR_ATOM_CREATE_CONNECTION_TOO_SLOW] Get connection for db '*****' from pool timeout. AppName:*****, Env:*****, UnitName:null. Message from pool: wait millis 5000, active 3, maxActive 5.`
- 说明：PolarDB-X向后端数据节点异步创建连接时，如果在短时间创建大量连接，或者数据节点建立连接速度太慢，会出现等待超时。该问题通常是由于后端数据节点压力过大或异常导致的，建议使用PolarDB-X读写分离，或者升级更高规格，减轻后端处理压力。

  如果排除后端数据节点问题后仍然出现该错误，请联系技术支持。如果问题是由短时间创建大量连接导致，建议联系技术支持调整PolarDB-X最小连接数。

PXC-4105 ERR_ATOM_ACCESS_DENIED

- 描述：PolarDB-X后端数据节点拒绝创建连接。
- 示例：`ERR-CODE: [PXC-4105][ERR_ATOM_ACCESS_DENIED] DB '*****' Access denied for user '*****'@'*****'. AppName:*****, Env:*****, UnitName:null. Please contact DBA to check.`
- 说明：该错误表明PolarDB-X通过用户名和密码连接数据节点时被拒绝访问，请联系技术支持。

PXC-4106 ERR_ATOM_DB_DOWN

- 描述：PolarDB-X后端数据节点无法连接。
- 示例：`ERR-CODE: [PXC-4106][ERR_ATOM_DB_DOWN] DB '*****' cannot be connected. AppName:*****, Env:*****, Unit`

```
Name:null. It seems a very real possibility that this DB IS DOWN. Please contact DBA to check.
```

- 说明：该错误表明PolarDB-X向后端数据节点创建连接超时或者没有响应。遇到该错误的通常原因是数据节点故障，请联系技术支持。

PXC-4108 ERR_VARIABLE_CAN_NOT_SET_TO_NULL_FOR_NOW

- 描述：变量（variable）不允许被设置为NULL。
- 示例：`ERR-CODE: [PXC-4108][ERR_VARIABLE_CAN_NOT_SET_TO_NULL_FOR_NOW] System variable ***** can''t set to null for now;`
- 说明：有些数据节点变量（variable）不允许用`SET var = x`语句设置成NULL值。遇到这种情况，PolarDB-X会提示PXC-4108错误。

PXC-4200 ERR_GROUP_NOT_AVALILABLE

- 描述：PolarDB-X下的某个数据节点暂时不可用。
- 示例：`ERR-CODE: [PXC-4200][ERR_GROUP_NOT_AVALILABLE] The TDDL Group ***** is running in fail-fast status, caused by this SQL:***** which threw a fatal exception as *****.`
- 说明：当分库包含的数据节点出现访问异常，并且分库下没有其他可用数据节点时，PolarDB-X会将分库置于fail-fast状态并提示PXC-4200错误。

  通常该错误是由于数据节点故障导致的。请根据包含的数据节点异常信息定位和解决问题。当故障数据节点恢复后，PolarDB-X将自动取消fail-fast状态。

  如果数据节点故障解决后仍然出现PXC-4200错误，请联系技术支持。

PXC-4201 ERR_GROUP_NO_ATOM_AVALILABLE

- 描述：PolarDB-X分库内暂时没有可用数据节点。
- 示例：`ERR-CODE: [PXC-4201][ERR_GROUP_NO_ATOM_AVALILABLE] All weights of DBs in Group '*****' is 0. Weights is: *****.`
- 说明：当分库包含的数据节点全都不可用，或者处于fail-fast状态时，PolarDB-X会提示PXC-4201错误。

  通常该错误是由于数据节点故障导致的。请检查后端数据节点状态以定位和解决问题。如果故障解决后仍然出现PXC-4201错误，请联系技术支持。

PXC-4202 ERR_SQL_QUERY_TIMEOUT

- 描述：PolarDB-X查询超时。
- 示例：`ERR-CODE: [PXC-4202][ERR_SQL_QUERY_TIMEOUT] Slow query leads to a timeout exception, please contact DBA to check slow sql. SocketTimout:*** ms, Atom:*****, Group:*****, AppName:*****, Env:*****, UnitName:null.`
- 说明：该错误表示SQL语句在后端数据节点上的执行时间超过PolarDB-X设置的socketTimeout参数限制。默认的PolarDB-X超时（socketTimeout）时间设置是900秒。

  建议优化SQL语句，以及在后端数据节点上创建适合的索引以提升SQL语句的执行性能。如果优化后的SQL语句仍然较慢，可以参见如下Hint语法临时设置PolarDB-X的超时时间：`/*TDDL:SOCKET_TIMEOUT=900000*/ SELECT * FROM dual;`，其中SOCKET_TIMEOUT设置的单位是毫秒。关于PolarDB-X的Hint用法，详情请参见如何自定义SQL超时时间。如果需要永久调整PolarDB-X超时设置，请联系技术支持。

PXC-4203 ERR_SQL_QUERY_MERGE_TIMEOUT

- 描述：PolarDB-X分布式查询超时。
- 示例：`ERR-CODE: [PXC-4203][ERR_SQL_QUERY_MERGE_TIMEOUT] Slow sql query leads to a timeout exception during merging results, please optimize the slow sql. The the default timeout is *** ms. DB is *****`
- 说明：PolarDB-X执行分布式查询超时，默认的超时设置是900秒。产生PXC-4203错误表示SQL语句扫描了多个分库的数据并且执行时间超过900秒。

  建议进行如下优化：

  - 在WHERE条件中添加分库键（Sharding key）条件，将SQL语句优化成单库执行；
  - 检查是否可以在后端数据节点上创建适合的索引，提升扫描各个分库数据的性能；

- 设法消除分布式查询中的跨库JOIN，数据重排序等耗时操作，降低PolarDB-X数据合并阶段的消耗。

如果优化后的SQL语句仍然较慢，可以使用如下Hint语法临时设置PolarDB-X的超时时间：`/*TDDL:SOCKET_TIMEOUT=900000*/ SELECT * FROM dual;`，其中SOCKET_TIMEOUT设置的单位是毫秒。关于PolarDB-X的Hint用法，详情请参见如何自定义SQL超时时间。

PXC-4400 ERR_SEQUENCE

- 描述：处理Sequence（全局唯一序列）失败。
- 示例：`ERR-CODE: [PXC-4400][ERR_SEQUENCE] Sequence : All dataSource faild to get value!`
- 说明：处理Sequence出错，错误信息在`Sequence :`中给出。导致PXC-4400的常见原因是数据节点故障，无法访问Sequence有关的数据表。

  建议先检查后端数据节点状态。如果排除数据节点故障后仍然发生错误，请联系技术支持。

PXC-4401 ERR_MISS_SEQUENCE

- 描述：Sequence不存在。
- 示例：`ERR-CODE: [PXC-4401][ERR_MISS_SEQUENCE] Sequence '*****' is not found`
- 说明：命令中使用的Sequence名称不存在。建议用`SHOW SEQUENCES`命令检查PolarDB-X中所有已创建的Sequence名称，并且选择正确的Sequence使用。

  如果使用的Sequence尚不存在，可以用如下语句创建：

```
CREATE SEQUENCE <sequence name> [ START WITH <numeric value> ]
[ INCREMENT BY <numeric value> ] [ MAXVALUE <numeric value> ]
[ CYCLE | NOCYCLE ]`
```

  如果使用的Sequence已经存在，但是仍然提示PXC-4401错误，请联系技术支持。关于PolarDB-X的Sequence用法，详情请参见概述。

PXC-4403 ERR_MISS_SEQUENCE_TABLE_ON_DEFAULT_DB

- 描述：Sequence使用的数据表不存在。
- 示例：`ERR-CODE: [PXC-4403][ERR_MISS_SEQUENCE_TABLE_ON_DEFAULT_DB] Sequence table is not in default db.`
- 说明：无法在PolarDB-X后端的数据库里访问名称叫sequence或者sequence_opt的数据表，请联系技术支持。

PXC-4404 ERR_SEQUENCE_TABLE_META

- 描述：Sequence数据表结构错误。
- 示例：`ERR-CODE: [PXC-4404][ERR_SEQUENCE_TABLE_META] the meta of sequence table is error, some columns missed`
- 说明：Sequence相关数据表（如sequence或sequence_opt）中缺少相应的字段，请联系技术支持。

PXC-4405 ERR_INIT_SEQUENCE_FROM_DB

- 描述：初始化Sequence错误。
- 示例：`ERR-CODE: [PXC-4405][ERR_INIT_SEQUENCE_FROM_DB] init sequence manager error: *****`
- 说明：初始化需要访问的Sequence时出错，错误信息在init sequence manager error:后给出。建议先检查PolarDB-X后端数据节点状态。如果排除数据节点故障后仍然提示PXC-4405错误，请联系技术支持。

PXC-4407 ERR_OTHER_WHEN_BUILD_SEQUENCE

- 描述：访问Sequence数据表出错。
- 示例：`ERR-CODE: [PXC-4407][ERR_OTHER_WHEN_BUILD_SEQUENCE] error when build sequence: *****`
- 说明：在访问Sequence相关数据表（如sequence或sequence_opt）时发生错误。错误信息在error when build sequence:后给出。

  建议先检查PolarDB-X后端数据节点状态。如果排除数据节点故障后仍然提示PXC-4407错误，请联系技术支持。

PXC-4408 ERR_SEQUENCE_NEXT_VALUE

- 描述：获取Sequence值出错。
- 示例：`ERR-CODE: [PXC-4408][ERR_SEQUENCE_NEXT_VALUE] error when get sequence's next value, sequence is: ****, error: *****`
- 说明：使用PolarDB-X自增主键，或者使用`<sequence name>.NEXTVAL`语法手工获取全局唯一ID时发生错误。错误原因在error:提示后给出。

  产生PXC-4408错误的原因是后端数据节点故障。建议先检查PolarDB-X后端数据节点状态和访问压力。如果排除数据节点故障后仍然提示PXC-4408错误，请联系技术支持。

PXC-4500 ERR_PARSER

- 描述：解析SQL语句失败。
- 示例：`ERR-CODE: [PXC-4500][ERR_PARSER] not support statement: '*****'`
- 说明：PolarDB-X支持符合SQL-92标准的SQL语法，以及MySQL支持的语法扩展与函数。请检查执行的SQL语句是否符合PolarDB-X兼容的SQL标准及MySQL规范。

  关于SQL标准语法，请参见SQL标准语法。关于SQL兼容性，请参见SQL语法使用限制。关于MySQL 5.6 SQL语法，请参见MySQL 5.6 SQL语法。如果您的SQL语句符合上述语法规则仍然提示PXC-4500错误，请联系技术支持。

PXC-4501 ERR_OPTIMIZER

- 描述：优化器转换SQL语句失败。
- 示例：`ERR-CODE: [PXC-4501][ERR_OPTIMIZER] optimize error by: Unknown column '*****' in 'order clause'`
- 说明：PolarDB-X优化器的工作是转换SQL语句到内部语法树。如果SQL语句中出现逻辑错误，优化器转换就会失败，产生PXC-4501错误。

  建议按照optimize error by:后的提示检查和调整您的SQL语句。如果调整SQL语句后仍然提示PXC-4501错误，请联系技术支持。

PXC-4502 ERR_OPTIMIZER_MISS_ORDER_FUNCTION_IN_SELECT

- 描述：ORDER BY包含的函数列在SELECT子句中不存在。
- 示例：`ERR-CODE: [PXC-4502][ERR_OPTIMIZER_MISS_ORDER_FUNCTION_IN_SELECT] Syntax Error: orderBy/GroupBy Column ***** is not existed in select clause`
- 说明：当SQL语句中的ORDER BY子句包含函数列（例如RAND()）时，PolarDB-X要求同样的函数列必须也在SELECT子句中出现，否则提示PXC-4502错误。

  建议在SELECT子句中添加相应的函数列。

PXC-4504 ERR_OPTIMIZER_SELF_CROSS_JOIN

- 描述：相同表JOIN的条件不足。
- 示例：`ERR-CODE: [PXC-4504][ERR_OPTIMIZER_SELF_CROSS_JOIN] self cross join case, add shard column filter on right table`
- 说明：PolarDB-X在执行相同表的JOIN时，如果WHERE子句只包含其中一张左表（或右表）的拆分字段（sharding column）条件，会提示PXC-4504错误。

  建议调整SQL语句，在WHERE子句中补全JOIN左表（或右表）的拆分字段条件。

PXC-4506 ERR_MODIFY_SHARD_COLUMN

- 描述：禁止更新拆分键。
- 示例：`ERR-CODE: [PXC-4506][ERR_MODIFY_SHARD_COLUMN] Column '*****' is a sharding key of table '*****', which is forbidden to be modified.`
- 说明：这是禁止修改拆分键（sharding key）的异常，PolarDB-X目前禁止修改包含GSI的表的分区键。

  建议将对应UPDATE语句修改为相同效果的INSERT+DELETE语句。

PXC-4508 ERR_OPTIMIZER_NOT_ALLOWED_SORT_MERGE_JOIN

- 描述：无法执行合并排序JOIN。
- 示例：`ERR-CODE: [PXC-4508][ERR_OPTIMIZER_NOT_ALLOWED_SORT_MERGE_JOIN] sort merge join is not allowed when m`

```
issing equivalent filter
```

- 说明：如果SQL语句中需要JOIN的数据表分别来自不同的数据节点，PolarDB-X会优先选择合并排序（Sort-merge Join）算法。该算法要求JOIN的左表与右表必须包含字段相等的关联条件，否则PolarDB-X将提示PXC-4508错误。

  建议调整SQL语句，在JOIN或WHERE部分添加相应的关联条件。

PXC-4509 ERR_OPTIMIZER_ERROR_HINT

- 示例：`ERR-CODE: [PXC-4509][ERR_OPTIMIZER_ERROR_HINT] Hint Syntax Error: unexpected operation: *****.`
- 说明：该错误表示SQL语句中的Hint语法无法被PolarDB-X解析。更多关于Hint语法信息，请参见如何使用HINT。

PXC-4510 ERR_CONTAINS_NO_SHARDING_KEY

- 描述：缺少拆分键（sharding key）条件。
- 示例：`ERR-CODE: [PXC-4510][ERR_CONTAINS_NO_SHARDING_KEY] Your SQL contains NO SHARDING KEY '*****' for table '*****', which is not allowed in DEFAULT.`
- 说明：禁止全表扫描（full-table scan）功能，PolarDB-X在建表时默认开启全表扫描功能。如果手动关闭全表扫描，建议确认与该表有关的SQL语句都已添加拆分键条件。

PXC-4511 ERR_INSERT_CONTAINS_NO_SHARDING_KEY

- 描述：INSERT语句缺少拆分键 (sharding key）。
- 示例：`ERR-CODE: [PXC-4511][ERR_INSERT_CONTAINS_NO_SHARDING_KEY] Your INSERT SQL contains NO SHARDING KEY '*****' for table '*****'.`
- 说明：当INSERT语句的目标是一张PolarDB-X拆分表时，必须在插入数据中包含拆分键的值（拆分键是自增主键例外）。否则，PolarDB-X将提示PXC-4511错误。

  如果遇到该错误，建议修改INSERT语句补充缺少的拆分键值。

PXC-4515 ERR_CONNECTION_CHARSET_NOT_MATCH

- 描述：输入的字符串与数据库的字符集不匹配。
- 示例：`ERR-CODE: [PXC-4515][ERR_CONNECTION_CHARSET_NOT_MATCH] Caused by MySQL's character_set_connection doesn't match your input charset. Partition DDL can only take ASCII or chinese column name. If you want use chinese table or column name, Make sure MySQL connection's charset support chinese character. Use "set names xxx" to set correct charset.`
- 说明：PolarDB-X支持用中文字符命名表名及字段名。在执行含有中文字符的SQL语句时，如果数据库连接的字符集设置（character_set_connection）不支持中文（如 latin1），会提示PXC-4515错误。

  您可以使用 `SHOW VARIABLES LIKE 'character_set_connection'` 查询MySQL客户端当前的连接字符集，使用 `SET NAMES` 命令修改当前连接字符集。如果是Java程序JDBC方式连接PolarDB-X，请设置数据库连接参数characterEncoding。

PXC-4517 ERR_MODIFY_SYSTEM_TABLE

- 描述：禁止修改系统表。
- 示例：`ERR-CODE: [PXC-4617][ERR_MODIFY_SYSTEM_TABLE] Table '*****' is PolarDB-XSYSTEM TABLE, which is forbidden to be modified.`
- 说明：PolarDB-X内部维护了一些系统表，使用SQL语句更新其中的数据会提示PXC-4517错误。限制的系统表包括sequence、sequence_opt、txc_undo_log等，请避免在业务或数据库设计中使用这些表名。

PXC-4520 ERR_DML_WITH_SUBQUERY

- 描述：不支持在DML使用子查询语句。
- 示例：`ERR-CODE: [PXC-4520][ERR_DML_WITH_SUBQUERY] DO NOT support UPDATE/DELETE with subQuery`
- 说明：目前PolarDB-X中禁止在DML语句中包含子查询语句，遇到此类问题建议先从业务上改写SQL，避免使用子查询。

PXC-4521 ERR_INSERT_SHARD

- 描述：Insert过程中一条记录被路由到了多个分片。
- 示例：`ERR-CODE: [PXC-4521][ERR_INSERT_SHARD] Cannot decide which group to insert`

- 说明：Insert过程中一条记录被路由到了多个分片，导致不知道这条记录被插入到哪个分片，出现这类问题，请联系技术支持。

PXC-4523 ERR_TODNF_LIMIT_EXCEED

- 描述：where后面的过滤条件太多。
- 示例：`ERR-CODE: [PXC-4523][ERR_TODNF_LIMIT_EXCEED] toDnf has exceed the limit size`
- 说明：PolarDB-X会对用户SQL的查询条件（即where子句）进行CNF/DNF的条件范式转换，并进行条件推导与优化，为了优化稳定性，限制了条件数2000。用户可以将DNF_REX_NODE_LIMIT参数调整大。

PXC-4524 ERR_TOCNF_LIMIT_EXCEED

- 描述：where后面的过滤条件太多。
- 示例：`ERR-CODE: [PXC-4524][ERR_TODNF_LIMIT_EXCEED] toCnf has exceed the limit size`
- 说明：PolarDB-X会对用户SQL的查询条件（即where子句）进行CNF/DNF的条件范式转换，并进行条件推导与优化，为了优化稳定性，限制了条件数2000。用户可以将CNF_REX_NODE_LIMIT参数调整大。

PXC-4526 ERR_FUNCTION_NOT_FOUND

- 描述：该函数不支持调用。
- 示例：`ERR-CODE: [PXC-4526][ERR_FUNCTION_NOT_FOUND] No match found for function signature`
- 说明：该错误表明在SQL语句中使用了错误的语法或不支持的函数。建议仔细检查SQL语句中的函数调用部分，使用正确的参数个数和类型调用函数。

PXC-4527 ERR_MODIFY_SHARD_COLUMN_ON_TABLE_WITHOUT_PK

- 描述：不允许在无主键的表上修改分库键。
- 示例：`ERR-CODE: [PXC-4527][ERR_MODIFY_SHARD_COLUMN_ON_TABLE_WITHOUT_PK]`
- 说明：PolarDB-X目前不允许在无主键的表上修改分库键。

PXC-4595 ERR_UNKNOWN_TZ

- 描述：设置错误的时区。
- 示例：`ERR-CODE: [PXC-4595][ERR_UNKNOWN_TZ]`
- 说明：建议检查时间的语法和格式是否设置正确。

PXC-4600 ERR_FUNCTION

- 描述：错误的函数调用。
- 示例：`ERR-CODE: [PXC-4600][ERR_FUNCTION] function compute error by Incorrect parameter count in the call to native function '*****'`
- 说明：该错误表明在SQL语句中使用了错误的语法或参数调用函数。建议仔细检查SQL语句中的函数调用部分，使用正确的参数个数和类型调用函数。

PXC-4602 ERR_CONVERTOR

- 描述：错误的类型转换。
- 示例：`ERR-CODE: [PXC-4602][ERR_CONVERTOR] convertor error by Unsupported convert: [*****]`
- 说明：该错误表明PolarDB-X在执行SQL时进行数据类型转换失败。请检查SQL语句中是否存在需要隐式类型转换的数据，并且尽量使用相同类型进行比较和计算。

PXC-4603 ERR_ACCROSS_DB_TRANSACTION

- 描述：错误的类型转换。
- 示例：`ERR-CODE: [PXC-4602][ERR_CONVERTOR] convertor error by Unsupported convert: [*****]`
- 说明：该错误表明PolarDB-X在执行SQL时进行数据类型转换失败。请检查SQL语句中是否存在需要隐式类型转换的数据，并且尽量使用相同类型进行比较和计算。

PXC-4604 ERR_CONCURRENT_TRANSACTION

- 描述：嵌套事务失败。
- 示例：`ERR-CODE: [PXC-4604][ERR_CONCURRENT_TRANSACTION] Concurrent query is not supported on transaction group, transaction group is: {0}.`
- 说明：PolarDB-X不支持嵌套事务，如果在同一个数据库连接里尝试同时开启2个以上事务，将提示PXC-4604错误。

  建议在应用开发时避免使用嵌套事务，或者使用应用层的事务框架防止产生嵌套事务。

PXC-4606 ERR_QUERY_CANCLED

- 描述：当前执行的SQL被取消。
- 示例：`ERR-CODE: [PXC-4606][ERR_QUERY_CANCLED] Getting connection is not allowed when query has been cancled, group is *****`
- 说明：使用 `KILL` 取消某条SQL语句的执行时，被取消的SQL语句会返回该错误。如果经常出现这一情况，请排查是否有客户端或程序在执行KILL命令。

PXC-4610 ERR_CONNECTION_CLOSED

- 描述：连接已经关闭。
- 示例：`ERR-CODE: [PXC-4610][ERR_CONNECTION_CLOSED] connection has been closed`
- 说明：当事务中的SQL语句执行出错，或者被 `KILL` 命令取消后，重复使用同一个数据库连接执行其他SQL语句会提示PXC-4610错误。

  建议在该情况下关闭连接，重新获取一个新的数据库连接。

PXC-4612 ERR_CHECK_SQL_PRIV

- 描述：由于权限不够，SQL语句无法执行。
- 示例：`ERR-CODE: [PXC-4612][ERR_CHECK_SQL_PRIV] check user ***** on db ***** sql privileges failed.`
- 说明：PolarDB-X的新版本支持为账号授权，类似MySQL账号权限体系，只有拥有对应类型权限的账号才能执行该SQL语句。如果账号权限不足，PolarDB-X将提示PXC-4612错误。

  建议检查用户拥有的PolarDB-X权限。如果权限不足，请在PolarDB-X控制台设置。关于PolarDB-X账号与权限设置，请参见管理数据库账号。

PXC-4614 ERR_EXECUTE_ON_MYSQL

- 描述：SQL语句在DN上执行报错。
- 示例：

```
ERR-CODE: [PXC-4614][ERR_EXECUTE_ON_MYSQL] Error occurs when execute on  GROUP '*****': Duplicate entry '******' for key 'PRIMARY'
PolarDB-X在后端数据节点数据库上执行SQL语句报错，末尾包含了从数据节点返回的原始错误信息，例如：
Duplicate entry '*****' for key 'PRIMARY'
表示写入数据节点数据表发生了主键冲突。
The table '*****' is full
表示数据节点使用的临时表已满，需要调整临时表空间或优化SQL语句。
Deadlock found when trying to get lock;
表示在数据节点中出现了死锁，通常是数据写入存在较多事务冲突导致的。
```

- 说明：建议参见PXC-4614提供的原始错误信息排查问题。更多关于SQL语句错误信息请参见MySQL 5.6文档。如果排除应用或数据节点问题后仍然发生PXC-4614错误，请联系技术支持。

PXC-4616 ERR_UNKNOWN_DATABASE

- 描述：错误的数据库。
- 示例：`ERR-CODE: [PXC-4616][ERR_UNKNOWN_DATABASE] Unknown database '*****'`
- 说明：PolarDB-X允许在DDL语句中指定数据库名称。如果指定的数据库名称与PolarDB-X提供的数据库名称不一致，将返回PXC-4616错误。

  建议修改DDL语句中的数据库名称，确保与PolarDB-X数据库名称一致。

PXC-4620 ERR_FORBID_EXECUTE_DML_ALL

说明：PolarDB-X禁止不带where条件执行delete和update操作。

PXC-4633 ERR_DB_STATUS_READ_ONLY

- 示例：`ERR-CODE: [PXC-4633][ERR_DB_STATUS_READ_ONLY] Database is read-only, only read sql are supported`
- 说明：该错误码表示数据库只有读权限，请检查下当前账号的权限是否受限。

PXC-4707 ERR_OUT_OF_MEMORY

- 描述：临时表使用内存超限。
- 示例：`ERR-CODE: [PXC-4707][ERR_OUT_OF_MEMORY]`
- 说明：为了保证数据库稳定性，PolarDB-X限制了每条查询的临时表使用内存量，说明用户当前的查询涉及到的数据比较多，可以打开SPILL或者调大当前查询临时表内存限制。

PXC-4709 ERR_IVENTORY_HINT_NOT_SUPPORT_CROSS_SHARD

- 示例：`ERR-CODE: [PXC-4709][ERR_IVENTORY_HINT_NOT_SUPPORT_CROSS_SHARD]`
- 说明：热点秒杀功能要求是单分片事务，如果涉及到多分片事务，会出现该错误。请调整业务逻辑，确保在单分片事务下使用热点功能。

PXC-4994 ERR_FLOW_CONTROL

- 描述：流量已达上限。
- 示例：`ERR-CODE: [PXC-4994][ERR_FLOW_CONTROL] [*****] flow control by *****`
- 说明：该错误代表PolarDB-X处理SQL请求已达到内部流量上限，当前请求被拒绝。建议检查SQL请求量是否存在异常峰值。如果观察到SQL请求量下降后，仍然大量提示PXC-4994错误，请联系技术支持。

PXC-4998 ERR_NOT_SUPPORT

- 描述：不支持的特性。
- 示例：`ERR-CODE: [PXC-4998][ERR_NOT_SUPPORT] ***** not support yet!`
- 说明：该错误代表使用的SQL语法或者功能PolarDB-X尚不支持。如果这些SQL语法或者功能对您十分重要，请联系技术支持。

PXC-5001 ERR_TRANS

- 描述：一般性的事务错误。
- 示例：`ERR-CODE: [PXC-5001][ERR_TRANS] Too many lines updated in statement.`
- 说明：请参见错误信息处理。
  - `Too many lines updated in statement`

    事务中的UPDATE语句更新行数超出限制（1000），建议检查UPDATE语句的WHERE条件。如果需要在事务中执行大批量数据更新，可以使用PolarDB-X `Hint/*TDDL:UNDO_LOG_LIMIT={number}*/` 调整限制值。
  - `Deferred execution is only supported in Flexible or XA Transaction`

    后置执行功能仅仅在柔性事务与XA事务策略下可用。在用PolarDB-X的 `Hint/*TDDL:DEFER*/` 语句提交后置执行功能之前，请先用 `SET drds_transaction_policy = ***` 命令更改PolarDB-X事务策略。
  - 其他错误信息，请联系技术支持。

PXC-5002 ERR_TRANS_UNSUPPORTED

- 描述：事务中的语法或功能尚不支持。
- 示例：`ERR-CODE: [PXC-5002][ERR_TRANS_UNSUPPORTED] Table without primary keys is not supported.`
- 说明：该功能在PolarDB-X事务中尚不支持。如果此功能很重要，请联系技术支持。

PXC-5003 ERR_TRANS_LOG

- 描述：无法访问事务日志。
- 示例：`ERR-CODE: [PXC-5003][ERR_TRANS_LOG] Failed to update transaction state: *****`
- 说明：为保证分布式事务的原子性，PolarDB-X在事务中会访问后端数据节点上的事务日志。如果PolarDB-X在读写事务日志

时出错，将返回PXC-5003错误。

产生PXC-5003错误的原因通常来自后端的数据节点故障。建议检查PolarDB-X后端数据节点状态和访问压力。如果排除数据节点问题后仍然产生PXC-5003错误，请联系技术支持。

PXC-5006 ERR_TRANS_COMMIT

- 描述：事务提交过程中出错。
- 示例：`ERR-CODE: [PXC-5006][ERR_TRANS_COMMIT] Failed to commit primary group *****: *****, TRANS_ID = *****`
- 说明：PolarDB-X在提交事务分支过程中出错，TRANS_ID对应的事务将被自动回滚。产生PXC-5006错误的原因通常来自后端的数据节点故障。建议检查PolarDB-X后端数据节点状态和访问压力。如果排除数据节点问题后仍然产生PXC-5006错误，请联系技术支持。

PXC-5008 ERR_TRANS_TERMINATED

- 描述：事务已被Kill或超时中止。
- 示例：`ERR-CODE: [PXC-5008][ERR_TRANS_TERMINATED] Current transaction was killed or timeout. You may need to set a longer timeout value.`
- 说明：如果PolarDB-X事务在执行中被Kill或者超时（执行时间超出drds_transaction_timeout值），则出现该错误。如果是事务超时导致报错，建议使用`SET drds_transaction_timeout = ***`命令修改PolarDB-X事务的执行时间上限，单位是毫秒。

PXC-5010 ERR_TRANS_CONTINUE_AFTER_WRITE_FAIL

- 描述：写入失败后，不允许继续进行事务操作。
- 示例：`ERR-CODE: [PXC-5010][ERR_TRANS_CONTINUE_AFTER_WRITE_FAIL] Cannot continue or commit transaction after writing failed`
- 说明：如果PolarDB-X事务中涉及到分布式事务，执行失败后，不允许继续进行事务，这个时候需要前端主动执行rollback，重试事务操作。

PXC-5108 ERR_CHECK_PRIVILEGE_FAILED_ON_TABLE

说明：当前账号没有对当前表的操作权限，请检查权限。

PXC-5119 ERR_FILE_CANNOT_BE_CREATE

说明：PolarDB-X默认关闭对`SELECT INTO OUTFILE`语句的支持，如有需要请联系技术支持。

PXC-5302 ERR_GLOBAL_SECONDARY_INDEX_UNSUPPORTED

说明：当前表不支持创建全局二级索引。可能是如下原因导致的：1. 当前表是单表或者广播表；2. 全局二级索引的列不包含分区键。其他情况请联系技术支持。

PXC-5306 ERR_GLOBAL_SECONDARY_INDEX_INSERT_DUPLICATE_VALUES

说明：在数据写入到全局二级索引表过程中存在主键冲突，请根据报错信息提示的记录值，确认是否存在主键冲突。

PXC-5308 ERR_GLOBAL_SECONDARY_INDEX_MODIFY_UNIQUE_KEY

说明：在执行DML过程中，全局二级索引表存在唯一键冲突，请根据报错信息提示的记录值，确认是否存在唯一键冲突。

PXC-5310 ERR_GLOBAL_SECONDARY_INDEX_ONLY_SUPPORT_XA

说明：PolarDB-X只有在XA/TSO分布式事务下，才支持全局二级索引，如果产生此报错，可能业务之前调整过默认事务策略，请调整回XA/TSO分布式事务，再创建全局二级索引。

PXC-5313 ERR_GLOBAL_SECONDARY_INDEX_MODIFY_GSI_TABLE_WITH_DDL

说明：PolarDB-X默认不支持对全局二级索引表做DDL操作，如有需要请联系技术支持。

PXC-5316 ERR_GLOBAL_SECONDARY_INDEX_INDEX_AND_SHARDING_COLUMNS_NOT_MATCH

说明：PolarDB-X在创建全局二级索引时，要求索引表字段必须包含主表分区键，因为在回表过程中需要根据分区键定位到具体的分片和具体记录。如果产生此报错，请检查DDL语句中索引表字段是否包含了分区键。

PXC-5317 ERR_GLOBAL_SECONDARY_INDEX_CONTINUE_AFTER_WRITE_FAIL

- 示例：`ERR-CODE: [PXC-5317][ERR_GLOBAL_SECONDARY_INDEX_CONTINUE_AFTER_WRITE_FAIL] Cannot continue or commit transaction after writing global secondary index failed`
- 说明：在包含GSI的表上执行DML语句时，如果产生此报错，则不允许继续提交包含该失败DML语句的事务。需要修改业务代码，DML报错后需要回滚事务之后重试。

PXC-5321 ERR_GLOBAL_SECONDARY_INDEX_BACKFILL_DUPLICATE_ENTRY

说明：在创建全局二级索引过程中，数据回填出现了索引表主键冲突，请确认索引表的主键是否存在相同值。

PXC-8007 ERR_ABANDONED_TASK

说明：查询过慢或者由于不明原因卡住超过2小时，数据库系统会终止查询，出现该异常。请先优化查询，确保不会出现超慢查询，若无法解决，请联系技术支持。

PXC-8008 ERR_EXECUTE_SPILL

说明：查询过程中，数据过大会触发临时表落盘，这个错误是在数据落盘过程中出现的异常，请联系技术支持。

PXC-8011 ERR_OUT_OF_SPILL_SPACE

说明：查询过程中，数据过大会触发临时表落盘，如果临时表生产的文件过多，超过了系统允许的可落盘的最大磁盘空间就会出现此错误。请先优化查询，减少计算过程中对临时表的依赖。若无法解决请联系技术支持。

PXC-8012 ERR_OUT_OF_SPILL_FD

说明：查询过程中，数据过大会触发临时表落盘，如果临时表生产的文件过多，超过了系统允许的文件句柄个数就会出现此错误。请联系技术支持，确保是否存在文件句柄泄漏，若没有泄漏，可以适当调大文件句柄个数限制。

PXC-8102 ERR_PAGE_TOO_LARGE

说明：在MPP并计算过程中，数据是按批在多个计算节点网络交互，如果一个批的数据过大超过RPC限制最大值，就会出现此错误，可以尝试调小默认CHUNK_SIZE值。

PXC-8103 ERR_NO_NODES_AVAILABLE

说明：在MPP并行计算过程中可能有计算节点出现故障，导致执行调度之初没有计算节点可调度，请确认计算节点服务是否正常，如果计算节点服务都正常仍然出现报此类错误，请联系技术支持。

PXC-9301 ERR_DUPLICATED_PARTITION_NAME

说明：在执行分区表相关的DDL操作过程中，使用了相同的分区表名。

PXC-9305 ERR_PARTITION_NAME_NOT_EXISTS

说明：在执行分区表相关的DDL操作过程中，提示分区表名不存在。请检查下分表名是否拼写正常，使用`show create table`和`check table`检查表元数据是否一致，如果不一致，可能是元数据维护信息不一致导致。请联系技术支持。

PXC-10004 ERR_X_PROTOCOL_RESULT

- 示例：`ERR-CODE: [PXC-10004][ERR_X_PROTOCOL_RESULT] Should use chunk2chunk to fetch data`
- 说明：PolarDB-X计算节点和数据节点采用的是私有RPC通信，出现这类异常主要是计算节点和数据节点建立的连接异常，具体原因有很多种，需要查看异常信息，如果无法定位具体问题，请联系技术支持。

11.PolarDB-X 1.0

11.1. 版本变更记录

11.1.1. 实例版本概览

本文主要介绍如何查看PolarDB-X 1.0实例当前版本号及各系列实例版本的概览。

查看实例版本

您可以通过如下方式查看当前实例的版本号：

- 通过控制台查看。
  i. 登录云原生分布式数据库控制台。
  ii. 在页面左上角选择目标实例所在地域。
  iii. 在**实例列表**页，找到目标实例，单击实例ID。
  iv. 在**配置信息**区域，您可以查看目标实例的当前版本号。

- 通过 `select version()` 命令查看。

  您可以先连接PolarDB-X 1.0数据库，再通过执行如下 `select version()` 命令查看目标实例的当前版本号：

```
mysql> select version();
+----------------------------+
| VERSION()                  |
+----------------------------+
| 5.6.29-TDDL-5.4.6-15972343 |
+----------------------------+
1 row in set (0.00 sec)
```

> **说明**
> - 其中 `5.4.6-15972343` 即为当前目标实例的版本号。
> - 关于如何连接数据库，请参见连接数据库。

升级实例版本

您可以在控制台上升级PolarDB-X 1.0实例到最新版本，快速体验实例新特性，具体操作步骤请参见升级版本。

实例版本概览

- V5.4.x系列实例版本说明
- V5.3.x系列实例版本说明
- V5.2.x系列实例版本说明
- V5.1.x系列实例版本说明

11.1.2. V5.4.x系列实例版本说明

本文汇总了PolarDB-X 1.0实例V5.4.x系列的版本说明。

发布日期：2021-11-03

大版本：V5.4.12

小版本：16359407

类别	说明
缺陷修复	修复数据库名包含特殊字符时，通过DTS导数据产生的报错问题。

发布日期：2021-10-28

大版本：V5.4.12 hotfix

小版本：16349923

类别	说明
新增	明确set指令的支持边界，set指令暂不支持包含表达式的计算。
优化	优化死锁检测逻辑，确保陷入到死锁的事务可以被快速中止。
缺陷修复	修复sql_mode设置语句携带表达式时，导致物理建连失败的问题。
	修复跨分片场景中包含limit的物理SQL由于受SQL cache的影响，存在查询结果不符合预期的问题。

发布日期：2021-09-14

大版本：V5.4.12

小版本：16315258

类别	说明
新增	优化器支持Cascades Style物理属性传递。
	支持AES加解密函数。
	新增自动KILL阻塞DDL的长事务的机制。
	支持UPDATE拆分键为表的首列且字段类型为时间的场景。
优化	优化go-sql-driver在prepare协议下，兼容0000-00-00 00:00:00时间的问题。
	优化创建聚簇索引时索引名大小写敏感的问题。
	优化部分场景下INSERT SELECT的事务可见性问题。
	优化DDL分表下的全并行执行策略。
	优化Create/Drop Table过程中，逻辑表的可见性。
	优化复杂DML过程中，由于数据过多易出现执行堆栈溢出问题。
缺陷修复	修复date_sub函数计算异常问题。
	修复高精度Decimal在特殊情况下解码出错的问题。
	修复点查命中带subpart的索引时，可能触发返回过多数据的问题。
	修复广播表上包含子查询的可下推单表DELETE，子查询中指定库名导致报错的问题。

发布日期：2021-07-05

大版本：V5.4.11

小版本：16251897

类别	说明
新增	支持select into outfile语句，可以按指定分隔符和格式导出数据。
	支持show table info from <tablename>语句，可以查看各分片的数据量。
	支持基于采样的Histogram with TopN处理数据倾斜的情况。
	支持SPM根据不同的参数空间选择不同plan的能力。
	单表和广播表支持随机物理表名。
	优化器支持Cascades搜索空间剪枝。
	支持AliSQL秒杀热点的语法。
	支持information_schema buffer相关的系统表：INNODB_BUFFER_POOL_STATS、INNODB_BUFFER_PAGE_LRU 和 INNODB_BUFFER_PAGE。
	新增GENERAL_DYNAMIC_SPEED_LIMITATION参数用于动态调整GSI回填校验和scaleout的限速。
优化	优化统计信息的处理过程，增加TopN用于处理数据倾斜的情况。
缺陷修复	修复的问题如下： • 兼容已下线的事务超时系统变量drds_transaction_timeout。 • load data导数据过程中对齐mysql对空字符的处理语义。

发布日期：2021-05-26

大版本：V5.4.10

小版本：16219368

类别	说明
修复问题	修复的问题如下： • 部分场景下未能正确填充Sequence。 • 特定字符串Collation条件下，GSI表数据不一致。 • NOW函数在涉及事务的部分场景下的更新不及时。

V5.4.10-16202941 日期2021-05-07

类别	说明
修复问题	修复部分场景下物理SQL丢失 `traceId` 的问题。

V5.4.10-16193390 日期2021-04-26

类别	说明
新增特性	新增支持通过 `show global variables like '%ssl%'` 查看SSL连接状态。
修复问题	• 修复当同时使用Join Hint和Limit SQL语句时，可能出现的栈溢出问题。 • 修复当通过ALTER TABLE添加全局二级索引（GSI）或执行ScaleOut命令时，由于数据回填中使用了默认的 `sql_mode` 导致报错的问题。 • 修复特定场景下，执行RENAME TABLE命令时会修改物理分片表名的问题。 • 修复部分场景下使用INSERT命令同时插入多行数据后， `last_insert_id` 取值不符合预期的问题。

V5.4.10-16179583 日期2021-04-09

类别	说明
新增特性	• 新增支持GROUPING SETS、ROLLUP和CUBE语法，用于按照多组维度对查询结果进行分组，详情请参见Grouping Sets、Rollup和Cube扩展。 • 新增支持0~6位时间精度设置，更多详情，请参见Fractional Seconds in Time Values。 • 新增支持包含全局二级索引的索引推荐。 • 新增支持在READ COMMITTED隔离级别下使用TSO事务。 • 新增限制只读实例下的Sequence访问。
修复问题	• 修复当通过同一连接重建库时，导致元数据异常的问题。 • 修复 `sql_mode` 值不合法导致SQL语句执行时会报错的问题。 • 修复在MySQL高可用切换过程中，低版本的Java驱动会断连的问题。 • 修复当对多表执行带有AGG子查询的DML命令时会报错的问题。 • 修复当多表UPDATE或DELETE时，相同表重复出现导致的 `affected rows` 不正确的问题。 • 修复ALLOW_READ事务策略下，在广播表写入失败后事务依然能够提交的问题。 • 修复相关联子查询条件推导结果中缺少OR条件，导致分区裁剪结果不正确的问题。 • 修复当多语句（即用英文分号（;）分割的SQL语句）中包含SET AUTOCOMMIT命令时，执行结果不正确的问题。 • 修复PREPARE协议下异常传递丢失的问题及元数据获取报错的问题。 • 修复某些情况下，当执行包含Interval函数的语句时，物理SQL会报错的问题。 • 修复全局二级索引更新过程中报错，导致事务未释放的问题。 • 修复子查询全表扫描误判，导致全表扫描不合理的问题。 • 修复函数名加反引号的用法会导致报错的问题。 • 修复建表时由于索引定义引用了不存在字段，导致空指针异常（NullPointerException，简称NPE）的问题。 • 修复为单表创建独立的关联Sequence后，当执行INSERT命令时，Sequence会不生效的问题。 • 修复无法在某些客户端上执行SHOW VARIABLES语句的问题。 • 修复部分情况下，SHOW PROCESSLIST无法查询数据的问题。 • 修复时间类型与函数的兼容性问题，提升对时间类型数据的处理效率。 • 修复监控中的连接数显示不正确的问题。 • 修复ALTER TABLE语句中CONVERT TO CHARACTER SET不能同时修改GSI索引表的问题。 • 修复部分PARSER报错信息不明确的问题。
优化性能	• 优化占用大内存SQL的执行，提升SQL执行速度。 • 优化HashJoin、NLJoin和SemiHashJon的执行方式，使其能按流式方式执行，提升执行速度。 • 优化查询优化器的算法，用于消除JOIN、FILTER、基于BKA（Batched Key Access）算法的IN条件中的重复条件。 • 优化部分子查询的执行速度。

V5.4.9-16142062 日期2021-02-25

类别	说明
新增特性	默认禁止执行未包含条件的UPDATE或DELETE语句。
修复问题	修复特殊情况下，无法正确提示非法SQL的语法错误的问题。

V5.4.9-16124261 日期2021-02-05

类别	说明
新增特性	• 新增支持ALTER TALBE ADD PRIMARY KEY语法。 • 新增支持单表下推场景下的变量赋值语法。

类别	说明
修复问题	• 修复MIN_ALLOCATE_SIZE参数设置时会出错的问题。 • 修复当列名定义与引用的大小写不同时，会导致GSI创建失败的问题。

V5.4.9-16116300 日期2021-01-26

类别	说明
新增特性	• 兼容MySQL Collation特性，新增支持若干Collation来确保SQL语句字符串排序的正确性，详情请参见Collation类型。 • 新增支持全部MySQL窗口函数，详情请参见窗口函数。 • 新增支持局部索引智能推荐功能，详情请参见智能索引推荐。 • 新增支持DELETE语句为单表指定别名。 • 新增支持AlTER VIEW语法。 • 新增支持SQL中使用 `_binary 'data...'` 表示二进制数据。 • 新增支持JOIN和DML语句中的自动全局二级索引选择。 • 新增支持事务中的复杂查询。 • PREPARE协议新增支持UNSIGNED数据类型。
修复问题	• 修复某些条件下复杂JOIN结果不正确的问题。 • 修复当窗口函数中OVER部分的参数为空时的报错问题。 • 修复执行BIT_OR函数会报错的问题。 • 修复在无主键的表上修改拆分键，报错信息会不准确的问题。 • 修复当查询结果中单行数据量超过16 MB时，查询结果的数据包未做切分导致报错的问题。 • 修复主键数据类型为UNSIGNED BIGINT的全局二级索引表，全局二级索引的数据回填进度不正确的问题。 • 修复部分情况下当物理SQL包含别名时，SQL执行会报错的问题。 • 修复ROUND函数的计算结果不正确的问题。 • 修复LOCATE、INSTR等字符串搜索函数的计算结果不正确的问题。 • 修复部分情况下IF函数无法下推的问题。
优化性能	• 优化广播表的读请求执行策略，避免读请求被路由到单个节点而出现热点访问。 • 优化PREPARE STATEMENT模型，提升PREPARE阶段的性能。 • 优化字符存储方式，提升系统处理字符类型时处理效率。 • 优化只读事务和单分片写事务的性能。

V5.4.8-16069335 日期2020-12-03

类别	说明
新增特性	• 支持窗口函数，详情请参见窗口函数。 • 支持分布式死锁的检测。 • 支持 `ROW_COUNT()` 和 `FOUND_ROWS()` 函数。 • 支持 `MATCH AGAINST` 函数（暂不支持WITH QUERY EXPANSION语法）。 • 支持向量化开关参数ENABLE_EXPRESSION_VECTORIZATION，您可通过自定义HINT命令开关该功能。 • 支持GLOBAL_INDEXES和METADATA_LOCK两张系统表，方便管理与维护全局二级索引。 • 支持SET TRANSACTION ISOLATION LEVEL语法，用于设置单条语句或单个事务的隔离级别。 • 支持START TRANSACTION READ ONLY语法，用于设置单个事务的读写属性。 • 支持LookUpJoin值裁剪，提升JOIN性能。

类别	说明
修复问题	• 修复事务内可能会多次获取相同分表上的连接，导致报错的问题。 • 修复因删除某数据库后又创建了同名的新数据库，所产生的脏数据导致SQL执行时的空指针异常（NullPointerException，简称NPE）问题。 • 修复部分情况下异步日志消费线程无法退出问题。 • 修复部分异常情况下引发的连接泄露问题。 • 修复部分SQL执行失败后，可能导致连接断开的问题。 • 修复与PolarProxy读写分离不兼容的问题。 • 修复可能产生重复CONNECTION ID的问题。 • 修复全局唯一索引报错信息中的索引名称与实际使用索引名称不对应的问题。 • 修复带GSI的DDL限制不严格的问题。 • 修复部分情况下可能存在的Stack Overflow问题。 • 修复ORDER BY列序号结果可能不正确的问题。 • 修复建表时，若表名长度超出MySQL限制，报错信息与MySQL报错信息不一致的问题。 • 修复CREATE TABLE LIKE语句执行时的NPE问题。 • 修复在只读实例上使用包含 `auto_increment_increment` 变量的命令会出现的NPE问题。 • 修复GSI DDL任务恢复后可能会导致不同随机后缀物理表遗留的问题。 • 修复DDL命令执行过程中偶发性的死锁问题。 • 修复当使用CHECK TABLE检查单表时，系统只检查部分分库的问题。 • 修复当存在大量RDS实例或分库时，DDL执行过程中偶发性的执行卡住问题。 • 修复部分ORDER BY场景下，物理SQL转换错误的问题。 • 修复跨库单表情况下，执行计划转换错误的问题。 • 修复部分跨库单表情况下，数据库名称在物理SQL命令中没有被去除的问题。 • 修复若时间类拆分函数中输入了NULL值，计算路由时会报错的问题。 • 修复Memory Pool泄漏的问题。 • 修复INSERT ON DUPLICATE KEY UPDATE中多个VALUES与同一行冲突时， `affected rows` 不正确的问题。 • 修复若表中包含多个UNIQUE KEY，INSERT ON DUPLICATE KEY UPDATE中多个VALUES与相互重叠的多行冲突时，导致变更行选择不正确的问题。 • 修复当多表UPDATE或DELETE时，相同表重复出现导致的 `affected rows` 不正确的问题。 • 修复TRACE结果中未打印UPDATE或DELETE语句参数的问题。 • 修复执行INSERT SELECT之后，可能出现的相同事务后续SELECT看不到新插入数据的问题。 • 修复部分场景下在子查询中引用了包含别名的表，导致UPDATE语句报错的问题。 • 修复DATETIME直方图序列化导致范围查询代估算不准确的问题。 • 修复在超长SQL执行时，因全链路压测判断导致PARSE缓慢的问题。
优化性能	• 优化IN VALUES类型语句执行计划的裁剪算法。 • 优化LookUpJoin的执行方式，统一按照流式方式执行。 • 优化全局事务日志表，通过MySQL分区表实现，消除因事务日志表清理造成的性能抖动。 • 优化执行器逻辑，避免出现SQL执行时会长时间卡住的问题。 • 优化代价估算模型，新增流式算子代价估算模型。 • 优化RENAME TABLE执行前的检查，提高实例整体性能。 • 提高DDL执行时的稳定性，减少DDL执行失败的概率。 • 优化INSERT冲突检测使用的物理SQL，若VALUES中存在重复情况可以提前去重。 • 在使用自增主键且UNIQUE KEY包含全部拆分键场景下，优化INSERT冲突检测使用的分区裁剪策略，避免全表扫描。

V5.4.7-16027425 日期2020-10-15

类别	说明
修复问题	• 修复升级实例版本后，无法通过MySQL上通过连接串登录PolarDB-X 1.0数据库的问题。 • 升级DRUID版本，并修复任务计数器泄露导致PolarDB-X 1.0无法与后端MySQL建立连接的问题。 • 修复EXPLAIN并发处理的相关问题。 • 修复当加载新建数据库配置出现偶发性报错并无法恢复相关配置时，导致新建数据库不可用的问题。 • 修复在单表上执行INSERT语句时，`LAST_INSERT_ID`结果异常的问题。 • 修复在非AUTO_INCREMENT的列上显式指定SEQUENCE时报错的问题。 • 修复REPLACE SELECT语句中包含AGG命令时，导致报错的问题。 • 修复使用KILL CONNECTION命令时，小概率出现的MDL锁泄露问题。 • 修复在区分大小写的MySQL存储引擎中从5.4.3或之前版本升级到5.4.4或以上版本时，导致的GSI元数据异常问题。 • 修复在混用utf8和utf8mb4连接编码场景下，当物理连接中使用SET NAMES失败时，小概率出现的编码异常问题。 • 修复ANALYZE表中JSON字段异常的问题。

V5.4.7-16000638 日期2020-09-14

类别	说明

类别	说明

新增特性	• 支持在DELETE语句中使用子查询，子查询的子句范围与SELECT相同。
	• 支持跨分片DELETE和UPDATE中使用ORDER BY、LIMIT或ORDER BY LIMIT命令。
	• 支持跨分片DELETE和UPDATE多表中的数据。
	• 支持在多表删除中删除广播表中的数据（目标表列表中可以包含广播表）。
	• 支持在UPDATE语句中使用子查询，子查询的子句范围与SELECT相同，且不支持在SET子句中使用子查询。
	• 支持UPDATE更新逻辑表的拆分键字段。
	• 支持在多表更新中修改广播表（广播表中的列可以出现在SET中赋值语句的左侧）。
	• 支持在VALUES从句中不指定拆分键或唯一键的值，或指定值为DEFAULT。
	• 支持通过INSERT ON DUPLICATE KEY UPDATE修改主键、唯一键或拆分键的值。
	• 支持在VALUES从句中设置唯一键中任意一列的值为NULL。
	• 支持在BATCH INSERT语句中两行的主键或唯一键重复。
	• 支持索引表中包含ON UPDATE CURRENT_TIMESTAMP或DEFAULT CURRENT_TIMESTAMP的列。
	• 支持索引表中包含类型为TIMESTAMP且未指定DEFAULT VALUE或指定了CURRENT_TIMESTAMP为DEFAULT VALUE的列。
	• 支持如下向量化的表达式：
	  ◦ 基本类型（如SHORT、INT、LONG、FLOAT、DOUBLE）的ADD（+）、MINUS（-）、MULTIPLY（*）、MOD（%）运算。
	  ◦ 基本类型的逻辑运算（如AND、OR、NOT）。
	  ◦ 基本类型的IS运算。
	  ◦ 控制流表达式（如CASE WHEN、IF、IF NULL）。
	  ◦ COALESCE表达式。
	• 支持GROUP JOIN算子。
	• 支持ENUM数据类型。
	• 支持下推算子代价估算。
	• 支持自动判断SQL负载类型。
	• 支持通过EXPLAIN LOGICVIEW展示下推算子。
	• 支持通过INFOMATION_SCHEMA.WORKLOAD展示实时负载。
	• 支持在INFOMATION_SCHEMA中显示PROCESSLIST、CONSTRATINS和VARIABLES信息。
	• 支持将SQL负载类型及代价信息打印到SQL日志中。
	• 支持在创建全局二级索引时强制要求索引必须包含索引名及分库拆分规则，并对未包含上述规则的二级索引创建语句给出准确报错。
	• 支持GROUP SEQUENCE自动抬高功能。
	• 支持通过LOAD DATA语法导入数据。

修复问题	• 修复聚合函数下唯一键判断错误可能导致的索引使用错误等问题。 • 修复非Java语言PREPARE模式下使用SELECT子查询可能报错的问题。 • 修复若干MPP模式下的潜在问题 （如部分对象无法序列化）。 • 修复对ZEROTIME、ZEROTIMESTAMP、ZERODATE等的支持问题。 • 修复建表存在重复列时不会报错的问题。 • 修复TINYINT返回长度概率性出错的问题。 • 修复全局二级索引锁丢失问题。 • 修复垂直拆分模式下通过ALTER TABLE加减列不生效的问题。 • 修复自定义auto_increment_increment参数后，getGeneratedKeys不准确的问题。 • 修复使用SEQUENCE语句时有可能访问到只读实例的问题。 • 修复STR_TO_DATE指定格式返回结果不正确的问题。 • 修复多种表达式的类型错误和执行错误问题。

类别	说明
优化性能	• 优化部分HASHJOIN执行计划，使之匹配BKAJOIN算法，提升查询性能。 • 优化MergeSort实现，解决归并排序过程中保持过多连接的问题。 • 优化复杂查询逻辑，使用基于CBO代价估算实现复杂查询的识别。 • 优化MemoryNotEnoughException错误提示，并在提示中显示USAGE和LIMIT信息。 • 优化TIME-BASED SEQUENCE批量插入的性能。 • 禁止同一拆分列在分库分表上使用不同的拆分函数。

V5.4.6-15946373 日期2020-07-13

类别	说明
新增特性	• 新增支持CUBE、ROLLUP、GROUPING和GROUPING SETS语法。 • 新增支持一个查询同时包含多个带DISTINCT关键字的聚合函数。 • 新增支持需要回表场景下全局二级索引的自动选择。 • 新增支持SHOW TRACE、SHOW SLOW使用WHERE和ORDER BY语法。 • 新增支持对视图的SHOW COLUMN和SHOW INDEX操作。 • 新增支持Functional Dependency。
修复问题	• 修复First Value导致聚合函数无法下推问题。 • 修复PREPARE协议下内存对象未被释放的问题。 • 修复PREPARE协议下执行SET语句抛异常的问题。 • 修复PREPARE协议下非Java户端可能会产生的错误问题。 • 修复部分场景下包含全局二级索引表上执行INSERT，由于主键校验异常导致报错的问题。 • 修复全局二级索引特殊情况下回滚失败的问题。 • 修复全局二级索引校验在错误超限时的报错信息。 • 修复包含全局二级索引的表在SHOW CREATE TABLE时缺失Sequence类型的问题。 • 修复DROP正在创建中的全局二级索引的报错信息。 • 修复SET开始的多语句，后续语句转义问题。 • 修复部分解析器语法问题。 • 修复标识符、别名等包含英文反引号（`）的问题。 • 修复TableMeta主键误报问题。 • 修复部分场景下元数据刷新导致UPDATE执行报错的问题。 • 修复RR隔离级别的一处连接管理问题。 • 修复SHOW STATS中ACTIVE_CONNECTIONS不准确的问题。 • 修复子查询部分场景下报错的问题。 • 修复OPTIMIZE TABLE有概率被路由到备库的问题。 • 修复部分模式下SELECT ... FOR UPDATE语句会走备库问题。 • 修复特殊情况下EXPLAIN OPTIMIZER报错的问题。
优化性能	• 优化MPP优化阶段，统一SMP及MPP的CBO算法。 • 优化XA分布式事务提交的性能。 • 优化窗口函数不支持的报错。 • 优化GROUP BY和ORDER BY中子查询不支持的报错。 • 禁用CREATE TABLE SELECT STATEMENT语法。

V5.4.5-15917814 日期2020.06.11

类别	说明
修复问题	• 修复部分场景下物理SQL中包含指定GSI的USE INDEX导致报错的问题。 • 修复包含GSI的表，在ON DUPLICATE KEY UPDATE中引用时间函数表达式导致报错的问题。 • 修复 `information_schema.statistics` 索引值为NULL问题。 • 修复 `information_schema.statistics` 大小写问题。 • 修复DML语句表名大小写处理异常，导致GSI创建失败的问题。 • 修复非Java语言使用PREPARE模式访问PolarDB-X 1.0可能出现的NPE问题。

V5.4.5-15897682 日期2020.05.18

类别	说明
新增特性	• 新增支持视图的创建、修改和删除。 • 新增支持单表并行查询。 • 新增支持在主键为任意类型的表上创建全局二级索引。 • 新增支持INFOMATION SCHEMA的复杂查询。 • 新增支持设置SESSION事务隔离级别。 • 新增支持EXPLAIN支持前置COMMENT（如 `/* COMMENT*/EXPLAIN …` ）。 • 新增支持热点补丁语法。 • 新增支持MySQL协议中 `COM_RESET_CONNECTION` 命令。 • 新增支持SHOW TRANS命令显示 `PROCESS_ID` 。
修复问题	• 修复MySQL 8.0 PREPARE模式下若干问题。 • 修复 `SHOW STATS` 中类型转换错误。 • 修复数字结尾的逻辑表名导致执行计划展示异常的问题。 • 修复 `EXPLAIN OPTIMIZER` 在部分场景下报错的问题。 • 修复特殊表结构导致 `UPDATE` 返回值数据类型异常的问题。 • 修复添加多列全局二级索引时，局部索引名过长导致报错的问题。 • 修复部分类型返回问题。 • 修复LIMIT、OFFSET过大值导致的溢出问题。 • 修复部分列子查询APPLY执行器问题。 • 修复高并发大批量多语句DML执行有概率被内存池分配限制并产生报错的问题。 • 修复 `SELECT FROM DUAL WHERE NOT EXISTS…` 报错问题。
优化性能	优化全局二级索引多索引表点写性能。

V5.4.4-15862314 日期 2020.04.07

类别	说明
修复问题	• 修复包含列子查询的INSERT SELECT在部分场景下报错的问题。 • 修复部分场景下物理SQL丢失别名的问题。 • 修复单逻辑表场景下应用执行计划失效的问题。 • 修复部分场景下执行计划缓存导致的权限问题。 • 修复拆分键大小写导致BKAJoin报错的问题。 • 修复BKAJoin不支持MergeUnion的问题。 • 修复部分带HINT的DELETE报错的问题。 • 修复AsyncDDL初始化过慢导致SPM初始化异常的问题。

V5.4.4-15846983 日期 2020.03.20

类别	说明
新增特性	• 支持非递归公共表表达式（CTE）语法。 • `SHOW PHYSICAL_PROCESSLIST` 指令支持WHERE、ORDER BY、LIMIT和ORDER BY LIMIT。 • 支持BASELINE HELP指令。 • 支持 `EXPLAIN JSON_PLAN [SQL]` 指令，输出JSON格式的物理执行计划。 • LIKE运算符支持ESCAPE关键字。 • 支持通过开关控制DDL生成的物理表名是否带随机串。 • 支持协程开关。
修复问题	• 修复HAVING中子查询重名情况下列别名匹配错误问题。 • 修复子查询中 `GROUP BY` 匹配别名错误问题。 • 修复非等号子查询关联项部分场景下的顺序问题。 • 修复枚举类型的等值过滤问题。 • 修复使用高版本JDBC驱动读取 `TINTINT(1)/BIT(1)` 显示为二进制问题。 • 修复TIMESTAMP固定显示3位秒级以下精度问题。 • 修复部分场景下TIMESTAMP和TINYINT互转问题。 • 修复 `SHOW STATS` 中并发处理可能导致的物理库QPS不准确的问题。 • 修复窗口函数优化部分场景下转化失败报错的问题。 • 解决跨库时存在同名表时解析优化表名混淆问题。 • 修复拆分键为复合主键一部分时，不为拆分键自动创建索引的问题。 • 修复 `SHOW TABLES` 和 `INFORMATION_SCHEMA` 查询结果包含当前用户无权限表的问题。 • 修复通过 `ALTER TABLE` 变更 `AUTO_INCREMENT` 字段时报SEQUENCE已存在的问题。 • 修复 `INFORMATION_SCHEMA.STATISTICS` 查询结果不完整的问题。 • 修复表名中带有英文句点（.）时建表和删表失败的问题。 • 修复 `SHOW COLUMNS` 不支持跨SCHEMA查询的问题。 • 修复EXPLAIN OPTIMIZER异常处理机制导致查询报错的问题。 • 修复UPDATE不支持别名使用英文反引号（`）的问题。 • 修复多表UPDATE部分场景下丢失别名导致的报错问题。 • 修复添加全局二级索引时，索引表中缺少主表上部分局部索引的问题。
优化性能	• 优化XA、2PC分布式事务流程，不再支持柔性事务。 • 优化DECIMAL数据类型实现。 • 优化事务内扫描多分片SQL的并发执行策略。 • 优化SEMIJOIN类子查询性能。

V5.4.3-15791082 日期2020.01.16

类别	说明
新增特性	• 新增DML语句的CPU统计及SQL日志指标的打印。 • 新增SQL日志增加arows属性（逻辑SQL的所有物理DML的影响行数）。
修复问题	• 修复产生一个物理分片的SQL查询的ptc与frows统计错误问题的问题。 • 修复JOIN HINT与回表算子的兼容性问题。 • 修复LIMIT存在时CBO异常下推Filter的问题。 • 修复USE、IGNORE INDEX指定无效索引时错误下推到物理SQL中的问题。 • 修复Filter下推对子查询ApplyCache的支持。

类别	说明
优化性能	• 优化启动时统计信息加载效率。 • 优化SET NAMES指令对utf8mb4兼容编码的处理效率。

V5.4.3-15788850 日期2020.01.13

类别	说明
新增特性	• 新增全局二级索引自动索引选择，详情请参见使用全局二级索引。 • 新增通过USE INDEX和IGNORE INDEX限制候选的全局二级索引。 • 新增通过UPDATE语句更新全局二级索引表的拆分键。 • 新增全局二级索引校验，详情请参见CHECK GLOBAL INDEX。 • 新增支持在BKAJOIN中进行动态分区裁剪。 • SHOW PROFILE命令增加显示计划的并行度及TRACEID。 • 新增支持 `SET tx_isolation = '…'` 语法。
修复问题	• 修复跨库INSERT SELECT的偶发性内存不足（OOM）问题。 • 修复统计信息列大小写敏感问题。 • 修复部分下推SQL EXPLAIN EXECUTE返回结果为空的问题。 • 修复LENGTH、IF、IFNULL函数的结果类型推导。
优化性能	• 改进DDL任务引擎的稳定性、易用性和性能。 • 优化单条SQL运行过程中的CPU统计准确性。

V5.4.2-15749091 日期 2019.11.28

类别	说明
修复问题	• 修复主表带AGG，以及有PROJECT中的子查询时，关联项ID丢失问题。 • 修复 `show stc` 指令丢失 `dbname` 和 `ipport` 信息问题。

V5.4.2-15744202 日期 2019.11.22

类别	说明
新增特性	• PolarDB-X 1.0支持RDS MySQL 8.0与PolarDB MySQL 8.0实例。 • 新增对SQL进行CPU及内存消耗分析功能，可通过 `SHOW PROFILE [CPU|MEMORY] [FOR QUERY X]` 命令分析。 • SHOW PROCESSLIST新增SQL的CPU消耗与内存消耗信息。
修复问题	• 修复TIME类型及相关函数兼容问题。 • 修复一些类型相关的推导问题。
优化性能	• 优化BATCH INSERT性能。 • 优化全局二级索引数据回填并行化，提升回填速度。

V5.4.1-15731181 日期 2019.11.07

类别	说明
新增特性	• 新增支持创建全局二级索引（CREATE GLOBAL INDEX）、创建全局唯一索引（CREATE UNIQUE GLOBAL INDEX）、查看全局二级索引语法（SHOW GLOBAL INDEX）。详情请参见使用全局二级索引。 • 新增动态调整PolarDB-X 1.0连接时区功能。 • 新增支持逻辑RENAME TABLE，提高RENAME TABLE性能。 • 新增CREATE SHADOW TABLE语法，用于创建和关联全链路压测影子表。 • 新增UPDATE、DELETE对DERIVED子查询的支持。 • 新增CBO、JOIN与AGG下推与交换能力。 • 新增SHOW METADATA LOCK查看DRDS METADATA LOCK信息。 • 新增支持PLAN MANAGEMENT运维指令。 • 新增支持CREATE TABLE WITH GLOBAL INDEX和CREATE GLOBAL INDEX的回滚。 • 新增CREATE TABLE和DROP TABLE关联未完成全局二级索引的检查并提示。 • 新增支持指定JOB ID执行SHOW DDL。 • 新增支持非等值的条件推导。 • 新增PARALLEL QUERY对更多算子并行的支持。
修复问题	• 修复回收站启用后，表名中包含特殊符号导致RENAME TABLE报错的问题。 • 修复PLAN MANAGEMENT线程池抛ABORT异常问题。 • 修复HashAggCursor处理不同类型输入HASHCODE不准确问题。 • 修复BETWEEN与NOT BETWEEN处理问题。 • 修复HashAggExec内存统计问题。 • 修复写入广播表时的函数计算出错问题。 • 修复使用高版本JDBC驱动连接MANAGE PORT报错的问题。 • 修复UPDATE、DELETE条件中包含库名导致报错的问题。 • 修复UPDATE、DELETE SET部分有重名列导致报错的问题。 • 修复 `LEFT JOIN + IS NULL` 导致分区裁剪异常的问题。 • 修复INSERT SELECT偶发的不支持错误的问题。 • 修复全局二级索引表名、拆分键、DDL回滚表名中包含特殊字符问题。 • 修复SQL中HINT和COMMENT多层嵌套时解析异常。 • 修复建表语句DEFAULT后加COLLATE的解析错误问题。 • 修复非法日期时间格式解析的问题。 • 修复CONVERT函数转换HEX到字符乱码的问题。 • 修复SUBSTRING、SUBSTRINGINDEX、JSONEXTRACT等函数传递NULL参数异常的问题。 • 修复 `CONNECTION_ID()` 与SHOW RPOCESSLIST不一致的问题。 • 修复子查询表名同外表一致时出现的异常。 • 修复子查询转化过程中CASE WHEN函数的类型问题。 • 修复PROJECT中多个子查询下推时的转换问题。 • 修复包含JOIN的列重名问题。 • 修复部分列找不到误报未表找不到的问题。 • 修复查询时ORDER BY不存在的列导致的StackOverflow问题。 • 修复WHERE条件中存在CASE WHEN且WHEN表达式存在AND条件下的下推错误问题。 • 修复当JOIN左右表存在不可下推的PROJECT时，进行 `JOIN_PROJECT_TRANSPOSE` 导致的问题。 • 修复当TIMESTAMP类型作为拆分键时会因时区导致分片路由错误的问题。

类别	说明
优化性能	• 优化规则自动清理任务执行。 • 优化INFORMATION_SCHEMA查询性能。 • 优化BKAJOIN COST估算算法。 • 优化DDL后更新统计信息、BASELINE逻辑。 • 优化两阶段AGG逻辑，支持GROUP BY列。 • 优化PolarDB-X 1.0事务日志的清理策略，减少清理时间，降低对后端存储的访问压力。 • 优化ADDTIME、SUBTIME、DATEADD、DATESUB、INTERVAL等函数中微秒精度计算。 • 优化SQL审计日志减少网络抖动的影响。 • 优化器内支持创建DOUBLE类型的LITERAL。 • 优化下推的GROUP BY查询避免生成子查询。 • 优化聚合函数内部实现，内存占用更低、性能更优。 • 优化CBO对常量表达式的选择率估计。 • 优化DDL相关解析，增强错误检查能力。 • 优化部分涉及时间类型的表达式计算中对异常时间的处理，兼容MySQL行为。

11.1.3. V5.3.x系列实例版本说明

本文汇总了PolarDB-X 1.0实例V5.3.x系列的版本说明。

v5.3.12-15708775 日期 2019.10.12

类别	说明
新增特性	新增Chunk-Based执行器，查询优化复杂查询。
	新增SQL执行计划管理（Plan Management）功能。
	新增新的DDL任务引擎，支持DDL任务的查看、恢复和回滚等功能。
	新增通过FORCE INDEX HINT方式指定查询使用某个索引的功能。
	新增非SCALAR子查询中ROW函数支持。
	新增DATE_FORMAT、INTERVAL等时间函数转义字符完整支持。
优化性能	主实例默认开启基于代价的优化（CBO），自动选择JOIN顺序和算子实现。
	优化更多类型的子查询（含SemiJoin）及下推。
	合并事务指令与应用业务SQL执行，减少后端数据库网络通讯，提升事务性能。
	优化内部线程池的参数配置。
	优化DRUID连接池在后端RDS异常情况下快速恢复。
	修复SQL解析的算术优先级问题。
	修复HASH JOIN部分场景下的连接泄漏问题。
	修复一元运算符物理SQL报错。
	修复LOCK TABLE单库无法正确执行问题。
	修复PolarDB-X 1.0对SAVE POINT的支持。
	修复MD5函数和HEX函数与MySQL不兼容的问题。

类别	说明
修复问题	修复 `SELECT USER(), DATABASE() LIMIT 1` 报错的问题。
	修复MERGE_UNION=FALSE并发行为不符合预期的问题。
	修复多个AND的日期范围查询，查询路由结果不正确的问题。
	修复RENAME TABLE时忽略目标库名的问题。
	修复管理员账户无法查看所有用户权限的问题。
	修复主实例并行度代价会低概率出现计算不准的问题。
	修复清理脚本导致日志滚动异常。
	修复UPDATE、DELETE单表语句中存在表名引用时，下推物理SQL报错的问题。
	修复不含ORDER BY的LIMIT语句因为内存限制的报错问题。
	修复多个AND条件查询结果不一致的问题。
	修复UPDATE包含子查询时的报错信息。
	修复DISTINCT和ORDER BY中同时出现函数的解析问题。

v5.3.11-15663020 日期 2019.08.20

类别	说明
修复问题	修复物理RT监控不准的问题。
	调整工作线程数配置。

V5.3.11-15622313 日期 2019.07.04

类别	说明
优化性能	优化分析型只读实例基于CBO进行JOIN重排及物理执行策略。
	优化分析型只读实例全表扫描速度。
	优化INSERT、REPLACE、UPDATE和DELETE语句的性能。
修复问题	修复并发KILL命令导致用户无法链接PolarDB-X 1.0的问题。
	修复原表自增主键存在0值，同步到PolarDB-X 1.0报错问题。
	修复多表UPDATE报错。
	修复用户存在视图时，SHOW TABLES报错问题。
	修复表名过长管控无法显示以及无法删除的问题。
	修复多库权限的账号无法同时使用Navicat的问题。
	修复COMPRESS协议下用户查询的结果集过大时卡死的问题。
	修复AVG函数参数为TIMESTAMP类型时的查询报错。
	修复ROUND函数返回类型问题。

类别	说明
	修复SHOW NODE读写次数超过整型上限问题。
	修复部分DDL以及语句解析报错问题。
	修复建带库名的广播表报错的问题。
	修复特殊场景下物理SQL优化导致分片计算异常的问题。
	修复ADMIN账户SHOW GRANTS无法看到全部用户权限的问题。
	修复多列IN只有一列有值时的报错问题。
	修复单表LOCK TABLES导致的系统异常。
	修复INSERT中拆分键有函数，EXPLAIN报错问题。

V5.3.10-15585948 日期 2019.05.23

类别	说明
新增特性	新增对PARALLEL QUERY的支持。
	新增SCALAR关联子查询支持UNSIGNED类型。
	支持ALTER SEQUENCE支持跨SCHEMA。
	支持SEQUENCE的NEXTVAL时的带有WHERE查询（如 `SELECT test.NEXTVAL FROM DUAL WHERE COUNT=10` 。
优化性能	优化带拆分键的查询性能。
	优化连接池特性，增加KEEP ALIVE支持。
修复问题	修复DESC带有SCHEMA名称（如 `DESC DB.ORDER` ）时，执行报错的问题。
	修复只读实例执行SHOW TABLE STATUS报错问题。
	修复XA RECOVER只读实例导致部分事务回滚的问题。
	修复时间类型拆分函数枚举日期缺失的问题。
	修复DISTINCT与ORDER BY子句混用执行报错问题。
	修复复杂查询下的 `COLUMN 'S' NOT FOUND` 问题。
	修复复杂查询下的 `COLUMN 'ID' IS AMBIGUOUS` 问题。
	修复 `CURRENT_DATE` 为列名时建索引报错问题。
	补全相关SHOW HELP帮助说明。

V5.3.9-15557587 日期 2019.04.20

类别	说明
	支持并行DDL。
	支持重构的系统变量（如SELECT）。
	支持物化子查询结果。

类别特性	说明
	SHOW STATUS支持事务的统计。
	支持SELECT后的AGGREGATE和表达式的嵌套使用。
	支持展示EXPLAIN MySQL物理执行计划。
优化性能	优化IN、NOT_IN子查询在某些场景下的性能。
修复问题	修复ALTER SEQUENCE时默认行为为GROUP，而非AUTO_INCREMENT。
	修复Convertor error by Unsupported问题。
	修复 `syntax error 'CHARSET=utf8 BLOCK_FORMAT=ENCRYPTED COMMENT` 问题。
	修复单表场景下ALTER TABLE ADD自增主键错误创建了SEQUENCE的问题。
	修复一些开启事物场景下导致报错的问题，提升事物兼容性。
	修复数据库插入YEAR类型非法字段成功导致的SELECT语句出错问题。
	修复CREATE TABLE ... PARTITION BY RANGE COLUMNS报错问题。
	修复索引表规则存在同名脏数据时，创建包含二级索引的表异常的问题。
	修复流量型只读实例上默认创建系统表导致报错的问题。
	修复DECIMAL函数的一些问题。

V5.3.8-15517870 日期 2019.03.05

类别	说明
新增特性	支持映射规则。
	支持SHOW FULL PROCESSLIST语句携带WHERE条件。
	支持如下跨Schema（数据库）操作： • 支持跨Schema查询。 • 支持跨chema使用BUSHY JOIN。 • 支持跨Schema的DML语法。 • 支持跨Schema的XA事务。 • 支持跨Schema的DAL语法。 • 支持跨Schema使用SEQUENCE。 • 支持USE DB_NAME进行跨Schema访问。
	增加对PREPARE协议支持。
修复问题	优化SHOW TABLES指令，解决以往SHOW TABLES执行缓慢的问题。
	修复以时间拆分列的取值为 `NOW()` 函数时可能的跨分片错误问题。
	修复 `CONVERT(column USING gbk)` 报错 `"Unknown datatype name 'gbk'"` 的问题。
	修复INTERVAL表达式不支持字段的问题（如 `DATE_FORMAT(DATE_ADD(MAX(table.column),interval table.column DAY),'%Y-%m-%d')` ）。
	修复INSERT里面有CASE WHEN报NPE问题。

类别	说明
	修复 `Hint: Allow full table scan = false` 不生效问题。

V5.3.7-15460044 日期 2018.12.28

类别	说明
新增特性	支持JOIN下推顺序调整，优化JOIN性能。
	支持根据表行数调整内存JOIN顺序。
	支持逻辑库锁功能。
	支持显示PREPARE和EXECUTE语法。
	增加HINT支持表间并行执行模式。
性能优化	优化由于白名单或密码问题导致登录失败的提示信息。
	优化多表JOIN，物理SQL生成多余子查询导致性能问题。
	优化执行器获取连接时，可能多次执行SET NAMES utf8mb4的问题。
	优化SHOW VARIABLES性能。
修复问题	修复SHOW FULL PROCESSLIST时SQL显示不全问题。
	修复SQL函数（如 `~1` ）报错问题。
	修复导入数据库报错，以及导入后建表报错问题。
	修复多表JOIN（如 `（index (16) must be less than size (16)）` ）报错问题。
	修复SCAN HINT对LIMIT处理不符合预期的问题。

V5.3.6-15447056 日期 2018.12.13

类别	说明
修复问题	修复用户自建账号查询information_schema报错，影响DMS及部分GUI工具使用的问题。
	修复Filter-Join结构执行计划下推不够精确，导致部分查询性能下降的问题。
	修复特殊场景下生成的物理SQL中包含冗余子查询，导致未能使用MySQL索引的问题。
	修复UNION优化导致部分场景下生成异常物理SQL的问题。
	修复相关联子查询生成物理SQL时的一个别名问题。
	修复特殊场景下LIMIT穿透JOIN 下推后查询结果异常的问题。
	修复SHOW CREATE TABLE未显示分表算法的问题。
	修复建表时无法在UNI_HASH中指定字符串列的问题。

V5.3.6-15439241 日期 2018.11.29

类别	说明
	支持为单个账户授予多个数据库权限，使用USE DATABASE切换数据库并执行SQL。

类别	说明
新增特性	支持SELECT语句中，全部的列子查询和WHERE中的子查询。
	支持Apply子查询算子，允许嵌套执行无法转为JOIN的子查询。
	支持HashBKAJoin算子。
	支持下推执行多表UPDATE、DELETE 语句。
	支持下推执行地理信息函数。
	支持异构索引表加列顺序检测，避免DDL导致索引表同步异常。
	支持IPv6。
优化性能	优化禁止使用BINARY类型作为拆分键，避免数据插入时自动补0导致查询异常。
	优化全链路压测PARSER性能。
修复问题	修复NODE HINT + LIMIT下发错误SQL，导致查询异常的问题。
	修复部分条件未能下推，导致LEFT JOIN性能低问题。
	修复部分场景下BKA JOIN + LIMIT执行异常的问题。
	修复Simple Sequence设置最大值时，越界的问题。
	修复CREATE TABLE对已存在表且带有IF NOT EXISTS关键字时报错问题。

V5.3.5-15423348 日期 2018.11.20

类别	说明
修复问题	修复并发策略选择导致部分场景下排序结果异常的问题。
	修复NODE HINT与LIMIT兼容性问题导致部分场景下结果异常的问题。
	修复事务策略清理机制变化导致部分场景下DTS任务异常的问题。
	修复BATCH INSERT在部部分场景下执行异常的问题。
	修复BATCH INSERT特殊转义字符处理报错的问题。

V5.3.5-15409153 日期 2018.10.30

类别	说明
新增特性	支持完整的GROUP_CONCAT语法。
	支持STR_HASH哈希函数，兼容PolarDB-X 1.0实例5.2版本所有HASH函数。
	支持UPDATE语句中使用别名。
	支持ON DUPLICATE KEY UPDATE语法中指定表名。
	支持下推执行JSON相关函数。
	支持CONVERT(X, SIGNED) 语法。
	支持JOIN USING语法。

类别	说明
优化	大幅降低BATCH INSERT内存占用，增强数据导入稳定性。
	优化INSERT并发策略，解决后端连接占用过高的问题。
修复问题	修复物理SQL中包含子查询时，JOIN重复列名报错问题。
	修复OUTLINE相关细节问题。
	修复TO_DAYS函数时间遇到闰年返回值与MySQL不一致的问题。
	修复建表主键、拆分键列名为关键字导致报错的问题。
	修复INSERT SELECT特殊语法报错的问题。
	修复ORDER BY、GROUP BY中同时包含函数导致执行计划异常的问题。
	修复BIGINT 列在结果集中表示为DECIMAL的问题。
	修复SHOW FULL TABLES FROM LIKE报错的问题。
	修复SHOW DATABASE语句中包含换行符导致报错的问题。

V5.3.4-15378085 日期 2018.09.20

类别	说明
新增特性	支持同步更新广播表，基于XA事务多写，无需异步数据同步。
	支持全局事务策略等多种动态参数，可通过控制台设置。
	使用XA作为默认事务策略。
	支持 `SET @@SESSION.BINLOG_ROWS_QUERY_LOG_EVENTS` 。
修复问题	修复BIT类型问题。
	修复BKAJoin+UNION优化导致JOIN查询报错的问题。
	修复SCALAR子查询的类型匹配问题。
	修复SEMI、ANTI执行器中CURRENT()方法未实现的问题。
	修复不支持CURRENT_TIMESTAMP特殊写法的问题。

V5.3.3-1670435 日期 2018.08.15

类别	说明
新增特性	支持PolarDB-X 1.0的XA事务。
	支持单元化Group Sequence。
性能优化	优化MysqlDump兼容性问题处理，支持MySQL可执行HINT。
	优化MysqlDump兼容性问题处理，支持MySQL单行注释 `"--"` 。
	修复存在HINT时，没有正确设置ThreadLocal变量导致NPE的问题。
	修复特殊场景下子查询条件上拉失败，导致全表扫描的问题。

类别	说明
修复问题	修复子查询中拆分键条件提取失败导致全表扫描的问题。
	修复计算物理分片时没有考虑 `BETWEEN … AND` 的情况，导致全表扫描的问题。
	修复可直接下推RDS的For update SQL可能走了备库的问题。
	修复 `SELECT @@session.tx_read_only` 返回值与MySQL不一致的问题。
	修复算术表达式在某些情况下计算结果不正确的问题。
	修复处理0000-00-00 00:00:00时间戳的取值报错的问题。
	修复NESTLOOP JOIN在数据较大的情况下可能导致内存不足（OOM）的问题。
	修复存在DRDS HINT在部分查询场景会导致NPE的问题。
	修复开启事务时，PolarDB-X 1.0与RDS之间进行变量传递引发的报错问题。
	修复SQL前加入多段PolarDB-X 1.0的HINT时SQL解析器会报错的问题。
	修复物理慢SQL输出不正确的物理表表名的问题。
	修复因算子计算量过大，引发协程过度占用CPU，并影响IO响应的问题。
	修复UNIX_TIMESTAMP()函数不支持不带函数的形式，且带参数之后结果均为NULL的问题。
	修复UNI_HASH对于bigInt unsigned及int unsigned的路由不正确的问题。
	修复INSERT带HINT指定物理分库物理分表报表会表不存在的问题。
	修复单库单表建表时错误走Group Sequence的问题。
	修复Druid连接池在某些特殊情况下的Fail-Fast恢复异常所导致 `"because null"` 报错，并引发连接泄漏问题。
	修复SQL的LIMIT子句被参数化导致物理SQL查询报错的问题，取消LIMIT子句参数化。
	优化精确选择BKAJoin算法的规则，解决不可下推JOIN内存占用过高的问题。
	修复INSERT VALUE中包含转意字符时，下发SQL报错的问题。
	修复LEFT JOIN条件推导不全，导致分片计算报错的问题。
	修复驱动表较大时BKAJoin的StackOverFlow的问题。
	修复因Union查询优化没及时关闭连接导致连接泄漏问题。
	修复特殊拆分规则导致拆分键大小写不匹配的问题。
	修复类型相关的元数据存在的内存泄漏问题。
	修复参数化缓存在SQL执行后没及时清空占用内存的问题。
	修复涉及到UNI_HASH、RANGE_HASH、RIGHT_SHIFT三个拆分函数的跨库JOIN查询执行计划不正确下推并导致报错的问题。
	修复SQL执行后返回JDBC的库名的meta信息不正确导致DMS或DTS操作报错的问题。
	修复XA事务校验set drds_transaction_policy的语句必须放在事务开始前，否则报错的问题。

V5.3.2-1645122 日期 2018.07.05

类别	说明
修复问题	修复带有Outline的SQL在高并发场景因无法生效而产生不合预期的执行计划的问题。
	修复事务中单个物理库的SELECT因使用并行查询策略导致拿物理连接报错的问题。

V5.3.2-1623342 日期 2018.05.25

类别	说明
新增特性	完善原生事务对XA事务的支持。
	INSERT、UPDATE、DELETE语句支持5.2和5.3版本的HINT。
修复问题	修复结果集元信息中表名不正确的问题。
	修复部分数据类型精度和范围异常的问题。
	修复使用Direct HINT且仅指定部分逻辑表名时报错的问题。
	SHOW NODE只统计读请求比例。
	修复分库不分表情况下，关键字作为表名时查询异常的问题。
	修复多UNION查询异常。
	修复单表 `INSERT last_insert_id` 返回0的问题。

11.1.4. V5.2.x系列实例版本说明

本文汇总了PolarDB-X 1.0实例V5.2.x系列的版本说明。

V-5.2.8-15738106 日期 2020.04.25

类别	说明
修复问题	修复删库操作导致元数据刷新失败的稳定性问题。

V5.2.8-15432885 日期 2018.11.27

类别	说明
新增特性	支持2PC分布式事务策略。
	兼容分布式事务中的INSERT、REPLACE ... SELECT语句。
优化性能	减少清理PolarDB-X 1.0事务日志的周期和数据保留时间，降低RDS存储空间占用。
修复问题	修复使用UNION时可能出现错误的问题。
	修复多语句UPDATE返回的影响行数为0的问题。
	修复Connection because null问题。
	修复对于特殊查询SQL鉴权失败问题。
	修复SHOW FULL TRANS LIMIT x命令的问题。
	修复只有当MySQL版本为5.6或以上才允许执行XA事务的问题。

V5.2.7-1638709 日期 2018.06.25

类别	说明
修复问题	修复某些情况下会看到Cannot find post-values for some PK的报错。
	修复SHOW TABLE STATUS的平均行数的精度显示问题。

V5.2.7-1623038 日期 2018.5.24

类别	说明
修复问题	修复指定分库HINT不支持以时间函数为拆分键。
	修复SHOW TABLE STATUS字段数值越界问题。
	修复添加表权限表名大小写问题。
	修复YYYYDD拆分函数在建表时跨年报错问题。
	修复YYYYMM拆分函数对NULL值异常问题。
	修复建表时TIMESTAMP类型精度问题。

V5.2.7-1606682 日期 2018.04.27

类别	说明
新增特性	支持PolarDB-X 1.0原生分布式事务，无须依赖其它外部组件。
	支持SHOW TRANS命令查看事务状态。
	支持SSL加密功能。
	支持单元化Group Sequence。
修复问题	修复物理连接池初始化失败后导致的连接泄漏的问题。
	修复当物理连接池初始化失败会导致的频繁反复建连接的现象。
	修复临时表排序在处理NULL值会报错的问题。
	解决分库分表键取值如 `"_utf8'xxx'"` 前缀类型会报错的问题。
	修复MySQL Driver因默认的sharedSendBuffer过大导致物理连接占用内存过多并在数据导入等场景引发FGC的问题。

V5.2.6-1556951 日期 2018.02.06

类别	说明
修复问题	修复事务及GTS事务中的读SQL会被路由到备库的问题。
	修复PolarDB-X 1.0自定义HINT不能同时作用于多张逻辑表（如JOIN的SQL语句）的问题。
	修复DDL在部分并发场景下执行会报 `"Lock wait timeout"` 异常，导致DDL无法正常完成的问题。
	修复一个数据库在被多个PolarDB-X 1.0实例同时引用时会出现的实例间建表或删表结果不一致的问题。
	修复因低版本的Fastjson安全缺陷造成的PolarDB-X 1.0安全风险问题。

类别	说明
	修复PolarDB-X 1.0的 `LAST_INSERT_ID()` 返回值情况与MySQL不完全一致的问题。

11.1.5. V5.1.x系列实例版本说明

本文汇总了PolarDB-X 1.0实例V5.1.x系列的版本说明。

V5.1.28-1508068 日期 2017.12.15

类别	说明
新增特性	支持查询一张逻辑表下的各个物理分表的数据分布情况。
	新增拆分函数UNI_HASH能让分库数一样的两张表的JOIN保持为非跨库JOIN，不再受到分表数目的影响。
	增加支持ISNULL()函数。
	支持只读库因同步延迟过大自动切断流量并回流主库的功能。
性能优化	优化权限相关的SQL错误提示，与MySQL保持一致。
	优化执行器的列查找性能，INSERT、REPLACE的吞吐提升14%。
	优化JVM的启动参数，CPU消耗降低5%。
	优化SQL类型识别的算法性能。
	优化跨库查询归并排序的算法性能。
	DDL建表的组合索引的第一列包含拆分键时将不再重新单独为拆分列自动建索引。
修复问题	修复关联子查询中引用的外部字段在直接下推时拼错的问题。
	修复GTS事务超过100记录列时，新的事务更新后无法回滚的问题。
	修复查询结果单个字段值过大会导致连接被意外关闭的问题。
	修复若干PolarDB-X 1.0后端连接池的一些连接泄漏问题和部分连接异常场景下不能自动恢复的问题。
	修复超长HINT报数组越界的问题。
	修复建表时TIMESTAMP类型的列指定为NULL DEFAULT NULL导致无法建表的问题。
	修正部分错误码的英文翻译。
	修复STR_TO_DATE()函数返回的时间类型与MySQL不一致的问题。
	修复通过PREPARE协议调用存储过程时传NULL值会报类型转换不正确的异常。
	修复频繁设置连接编码会导致的内存泄漏问题。
	修复在使用GTS事务之下执行InsertSelect出现读可见性不一致的问题。
	修复带DISTINCT的JOIN语句在新版本PolarDB-X 1.0报Column is ambiguous的问题。
	修复 `"((x) IS TRUE) IS TRUE"` 语句执行报错的问题。
	解决DISTINCT在部分跨库JOIN场景下会出现报错的问题。
	修复分库分表情况下聚合函数查询带LIMIT时返回结果不正确的问题。

类别	说明
	解决BigDecimal类型在类型转换过程会消耗大量CPU的问题。
	修复SHOW FULL STATS连接数不准确的问题。
	修复SHOW FULL STATS CPU，内存不准确的问题。
	修复当前端连接过多且因流控被堵塞在读取时，JVM的临时内存缓存出现堆积和泄漏的问题。
	修复个别多语句处理场景中包含ROLLBACK语句可能导致的NPE错误。
	修复 `SHOW VARIABLES LIKE"autocommit"` 显示的值不正确的问题。
	修复ResultSet.getObject方法获取tinyint（1）类型的数据，返回结果类型跟MySQL不一致的问题。

V5.1.28-1408022 日期 2017.09.15

类别	说明
新增特性	支持逻辑连接的SHOW PROCESSLIST与KILL。
	支持 `INSERT INTO … SELECT < sequence_name >.nextVal [ , … ] FROM DUAL` 语法。
	支持OPTIMIZE TABLE语法。
	增加对ODBC的支持。
	增加ANSI_QUOTE类型SQL_MODE的支持。
	GROUP BY、ORDER BY、HAVING子句中出现的函数不再必须出现在SELECT中。
	分布式事务中，支持 `SELECT last_txc_xid(timeoutMillisecond)` 指定超时时间。
优化性能	优化复杂LIMIT查询的性能。
	优化str2date函数对时分秒的支持。
	优化JSON_SET、JSON_OBJECT、JSON_EXTRACT、JSON_UNQUOTE、JSON_REMOVE、JSON_MERGE、JSON_KEYS、JSON_REPLACE、JSON_INSERT函数在复杂SQL中的支持。
	优化对Navicat的兼容性。
修复问题	修复SHOW FULL STATS的元信息类型错误的问题。
	修复单库单表下 `INSERT INTO … ON DUPLICATE KEY UPDATE …` 语句报错的问题。
	修复因实例过期续费再次使用时，建表失败的问题。
	修复PolarDB-X 1.0分库分表条件中拆分键为整数类型的负数条件查询报错的问题。
	修复执行时表找不到的问题。
	修复PolarDB-X 1.0单库单表时因LOCK、UNLOCK语句导致数据无法导出的问题。
	修复部分场景下中文表名时乱码的问题。
	修复分布式事务中FOR UDPATE查询返回数据错误的问题。

V5.1.28-1349132 日期 2017.08.10

类别	说明
修复问题	修复由于新增关键字DB，导致无法使用DB作为列名或表名，并且影响Navicat 从PolarDB-X 1.0导入或导出数据的问题。
	修复字符串类型拆分键的哈希值出现负最大值时会导致路由结果不正确的问题。
	修复SHOW FULL SLOW指令在某些情况下会报SLOW表不存在的问题。
	修复Server Prepare协议下，有小概率出现权限校验错误的问题。
	修复使用拆分函数RIGHT_SHIFT和RANGE_HASH的拆分表在INSERT时，某些情况下出现的多语句无法执行的问题。

V5.1.28-1320920 日期 2017.07.18

类别	说明
新增特性	支持RENAME TABLE功能。
	支持创建拆分表时新规则函数和多列规则。
	支持REPLACE INTO SELECT的用法。
	支持能够展示分库容量的SHOW DB STATUS命令。
	支持KILL ALL命令批量中断慢SQL执行。
	PolarDB-X 1.0控制台支持RDS 5.7时使用非拆分模式建库。
	升、降配后能够自动调整内部连接池的大小。
	专有网络vswitch支持跨可用区。
	通过PolarDB-X 1.0控制台切换至VPC网络后，自动清理PolarDB-X 1.0白名单。
	改进PolarDB-X 1.0控制台在一些场景下的显示内容和提示信息。
修复问题	修复高并发场景下偶发的客户端与PolarDB-X 1.0连接意外中断（MySQL server has gone away）的问题。
	解决PolarDB-X 1.0频繁执行DDL时，偶发出现的DDL执行中会挂起而无法恢复的问题，并提升了DDL的执行性能。
	修复通过OpenAPI创建实例时返回400错误码的问题。
	解决子账号通过DMS连接或登录PolarDB-X 1.0的问题。
	修复PolarDB-X 1.0控制台建库过程中，系统表创建失败导致的PolarDB-X 1.0实例无法释放的问题。
	修复无法在PolarDB-X 1.0控制台中从VPC网络切换回经典网络的问题。

V5.1.27-1282781 日期 2017.06.26

类别	说明
	默认禁止全表删除与全表更新的高危操作，可以通过加HINT临时跳过此限制。
	账号和权限功能调整（用户名长度范围改为支持2~20个字符，且允许出现英文下划线（_）字符）。
	SHOW SLOW和SHOW FULL SLOW指令显示结果中添加用户信息。
新增特性	

类别	说明
新增特性	单表查询支持LIMIT 0优化。
	DELETE语句的ORDER BY部分支持子查询。
	在SQL日志中添加标识由哪个帐号执行的信息。
修复问题	修复不含分号的语句被当成多语句执行的问题。
	修复SELECT语句常量列设置别名导致提示列找不到的问题。
	长度超过2000字节的SQL语句不会被缓存，避免占据内存导致性能下降。

V5.1.27-1217986 日期 2017.05.23

类别	说明
修复问题	修复PolarDB-X 1.0建表时库名带英文引号（''）报错的问题。
	修复PolarDB-X 1.0建表时TIMESTAMP类型带 `on update current_timestamp` 子句的报错问题。
	修复PolarDB-X 1.0控制台删除只读账号后不生效的问题。
	修复SQL强制指定权重为0的只读实例进行查询的报错问题。
	修复PolarDB-X 1.0默认打参数化日志导致性能下降的问题。
	修复Group Sequence更新系统SEQUENCE表日期字段无效的问题。

V5.1.26-1502740 日期 2017.02.24

类别	说明
新增特性	分布式账号和权限系统发布，支持企业级数据库帐号和资源管理。
	新增Time-based Sequence。
	扩展SEQUENCE语法，支持创建不同类型的SEQUENCE（默认为Group Sequence），支持修改SEQUENCE的类型。
优化性能	优化SIMPLE SEQUENCE性能。

V5.1.26-1037257 日期 2017.02.16

类别	说明
新增特性	支持自助优化连接池参数。
修复问题	修复激活访问RDS无法授权问题。
	修复RDS使用MySQL 5.7无法建库问题。
	修复PolarDB-X 1.0实例平滑扩展出错问题。

V5.1.26-970482 日期 2017.01.10

类别	说明
	支持自助优化连接池参数。

类别	说明
新增特性	支持自助切换PolarDB-X 1.0网络类型为VPC新功能。
	控制台显示分库容量比例，方便确定库容方案。
	控制台支持自建MySQL或者RDS平滑导入数据到PolarDB-X 1.0。

V5.1.26-9024000 日期 2016.12.13

类别	说明
新增特性	控制台支持接入分布式全局事务，可以通过PolarDB-X 1.0控制台开通分布式事务。

V5.1.26-866978 日期 2016.11.24

类别	说明
新增特性	新增CPU报警，第一时间发现CPU瓶颈，升配解决资源瓶颈。
	增加慢SQL明细，随时查看实例慢SQL。
	PolarDB-X 1.0控制台增加实例监控，支持监控CPU、网络、物理QPS、逻辑QPS、连接数、活跃线程数等性能。

V5.1.25-791778 日期 2016.10.09

类别	说明
新增特性	支持服务端开启限流保护实例（默认关闭）。
	支持中文表名和列名。
	支持大小写敏感Collation类型。
	支持 `LIMIT row_count OFFSET` 语法。
	分布式事务升级到2.0.17，支持超时时间、隔离级别动态配置。
优化性能	优化第一次访问PolarDB-X 1.0的性能。
	UPDATE和部分查询语句局部性能优化。
修复问题	修复子查询中多重JOIN列查找问题。
	修复SET指令导致连接污染问题。
	修复SQL语句中含有 `Long.MIN_VALUE` 导致溢出的问题。
	修复数字类型的列错传了一个含有非常大的科学计数法数据的子字符串导致内存溢出的问题。
	修复了唯一序列相关的几个问题。
	修复了含有GROUP BY函数的子查询报错的问题。
	修复VARBINARY类型的列在系统编码是GBK的情况下，取数据的结果会和数据库中的不一致的问题。
	修复LEFT JOIN中含有子查询时丢失条件导致查询结果不正确的问题。
	修复了部分场景下长SQL会导致内存溢出的问题。

类别	说明
	包年包月实例变配前，需要检查用户是否创建了RAM角色。
	修复了部分场景下分布式事务内存溢出问题。

V5.1.24-653395 日期 2016.07.26

类别	说明
新增特性	SEQUENCE支持全局有序自增，并提供增量、最大值等功能和相应语法的支持。
	支持TO_DAYS函数。
	支持包年包月购买实例。
	接入RAM主子账户。
优化性能	完善库存不足、数据导入失败的告警信息。
	数据导入状态查询增加自动重试。
修复问题	修复USE语句含有 `";"` 解析出错的问题。
	修复SELECT 、UPDATE 等语句在WHERE条件中含有单个条件且是永假式，WHERE条件会丢掉的问题。
	修复升降配检查实例数上限的问题。
	修复包年包月到期锁定共享实例的问题。
	修复升降配失败时无法告警的问题。
	数据导入移除对源表的外键检查。
	数据导入检测目标表是否包含外键。

V5.1.24-826 日期 2016.05.24

类别	说明
新增特性	数据拆分列的字段类型增加支持BigDecimal类型。
	支持Now(3)、Now(2)、Now(1) 函数。
	增加只读账户OpenAPI。
	增加修复RDS后端链接功能。
优化性能	VPC实例显示VPC地址。
修复问题	修复高并发访问PolarDB-X 1.0后端连接长时间不会自动回收的问题。
	修复后端连接池在主备切换后会出现概率性无法恢复的问题。
	修复SEQUENCE的缓存清除不干净，导致SEQUENCE不可用的问题。
	修复INSERT SELECT语句在广播表存在的情况下不会下推分片执行的问题。
	修复BigDecimal类型的浮点数精度不会显示为指数的问题。
	修复部分存储过程调用会导致连接进入不可恢复状态。

类别	说明
	修复 `SET @session.sql_log_bin=0` 语句会导致连接池污染的问题。
	修复开启TXC功能后SQL报错信息会显示异常的问题。
	修复部分DDL执行过慢问题。
	修复部分Count（Distinct）语句出现数据重复的问题。

V5.1.23-783 日期 2016.04.12

类别	说明
新增特性	控制台新增PolarDB-X 1.0实例级别监控。
	控制台增PolarDB-X 1.0诊断建议，提前发现实例异常。
	控制台新增PolarDB-X 1.0实例数据库拓扑。
	控制台新增PolarDB-X 1.0数据库概览。
	提供 `SHOW FULL SLOW | PHYSICAL_SLOW` 指定，支持慢SQL明细的查询。
	支持指定单个RDS上的分库数量。
优化性能	变配失败后立即修复实例，不需要手动修复。
	建库等待时间延长至半小时，解决等待RDS数据库创建超时问题。
	数据库列表页面删库，自动刷新。

V5.1.23-742 日期 2016.03.22

类别	说明
新增特性	支持Now(3)、Now(6) 函数。
	支持非拆分模式下自定义函数执行。
	支持非拆分表的 `SELECT sql_calc_found_rows` 与 `SELECT found_rows` 语法。
	读写分离，支持多个只读实例之间的failover机制。
	支持异常连接快速回收机制（如识别RDS锁定状态、访问失败等）。
	支持CHECK TABLE语法，检查所有分片表结构一致性。
	增加应用状态统计信息SHOW STATS。
	支持YYYYMM分区。
优化性能	优化DDL执行过慢问题。
	完善INSERT SELECT语法的下推优化。
	全局ID优化为大小写不敏感。
	慢SQL统计优化，增加连接获取时间和SQL执行时间，支持过滤条件分析。
	修复PREPARE协议对于大文本对象解析的问题。

修复问题类别	说明
	修复多表JOIN连接条件推导无效的问题。

V5.1.22-675 日期 2016.02.23

类别	说明
新增特性	建库输入库名后实时进行校验。
	共享实例迁移数据库到专享实例，自动刷新RDS白名单。
	OpenAPI建库刷新RDS白名单。
	使用RDS高权限账号建库、扩容。
	升降配过滤掉状态不正常的RDS实例。
	数据导入支持DECIMAL类型和DATE类型的表主键。
	数据导入切换时自动修改SEQUENCE。
	设置实例读写比例时刷新只读实例的白名单。
	自动恢复失败的数据导入任务。

V5.1.22-614 日期 2015.12.08

类别	说明
新增特性	支持查看一个表非精确数据量。
	HINT支持过滤条件。
	SCAN HINT支持EXPLAIN语句，支持使用EXPLAIN查看所有分片上的执行计划。
	支持MySQLDump。
	支持MySQL地理字段类型geometry。
	支持INSERT SELECT语句。
	支持 `GROUP_CONCAT` 函数。
	支持GROUP BY语法。
修复问题	修复只读账号不支持SET语句的问题。
	修复DELETE语句中存在嵌套子查询，子查询中列无法找到的问题。
	修复在 `SELECT * FROM (UNION) WHERE` 语句中，WHERE条件丢失的问题。
	修复建分库分表时，若同时存在分库分表字段，且分库分表字段一个带英文反引号（`），一个不带，SQL解析列名错误的问题。
	修复SQL中存在大量OR条件时内存占用过多的问题。
	修复当SQL中GROUPBY、DISTINCT、AGGREGATE和LIMIT混用时结果错误的问题。
	修复LEFT OUTTER JOIN中如果广播表在左边结果集错误的问题。
	修复 `SET sql_mode=""` 会报错的问题。

V5.1.22-580 日期 2015.10.27

类别	说明
新增特性	控制台新增PolarDB-X 1.0数据库只读账号管理。
	公有云支持域名作连接串。
性能优化	只读账户密码修改支持短信确认。
	开启读写分离时，支持根据Slave延迟控制是否走只读实例。
	建表时自动为分库键创建索引。
	按时间和ID做双字段分区时，优化针对分库键的GROUP BY语句。
	针对Top N的ORDER BY LIMIT使用内存堆排。
	针对DML类型优化为单库下推模式。
	BEGIN、ROLLBACK、COMMIT语句分号精确识别。
	支持实例间并行执行DDL语句。
	支持 `SET AUTOCOMMIT=on/off` 。
	CREATE TABLE IF NOT EXIST语句优化容错。
	针对广播表优化TRUNCATE语句，自动下发到所有分库上执行。
修复问题	修复DMS截断DBPARTATION语句的问题。
	修复批量INSERT出现fatal exception时的空指针异常。
	提前判断MySQL变量名称，防止传入错误的系统变量名，导致链接失效。
	修复系统编码为GBK时，VARBINARY类型getBytes结果与数据库不一致的问题
	修复DDL锁记录冲突问题。
	修复虚表LEFT JOIN问题。
	修复三表做分库键上的JOIN时，分库键的值只被传递一次的问题。
	修复当 `DATE( gmt_modified )<=?` 语句右侧值为DATE类型时的空指针异常问题。

V5.1.21-513 日期 2015.09.07

类别	说明
新增特性	支持VPC和商业化实例的OpenAPI。
	增加实例列表中显示网络类型。
	跳转到DMS页面时自动填写可用VPC实例连接信息。
	支持查看PolarDB-X 1.0实例机器IP列表。

V5.1.21-478 日期 2015.08.06

类别	说明
新增特性	如果RDS有白名单，会提示将相关IP加入RDS白名单。
	PolarDB-X 1.0数据库初始化增加分布式事务日志表。
	PolarDB-X 1.0数据表拆分键自动加索引。
	支持金融云用户购买实例。
优化性能	PolarDB-X 1.0建库失败时自动清理垃圾数据。
	PolarDB-X 1.0导入数据库失败时，自动清理配置，支持重复导入。
	DDL时过滤带Schema的语句
	优化数据导入功能，增加 `DRDS InstanceId` 参数。
修复问题	修复聚石塔购买页面错误。

V5.1.21-464 日期 2015.07.21

类别	说明
新特性	PolarDB-X 1.0开始商业化，增加按量实例购买入口，支持选配、询价、批量下单等功能。
	实例生命周期管理，包括创建、欠费、续费、欠费释放等。
	支持主动释放实例。
	数据导入候选表列表中过滤视图。
	支持MySQL Compress压缩协议的读和写。
	支持数据平滑导入。
	支持读写分离。
优化性能	完善MySQL Prepare协议支持。

11.2. 技术白皮书

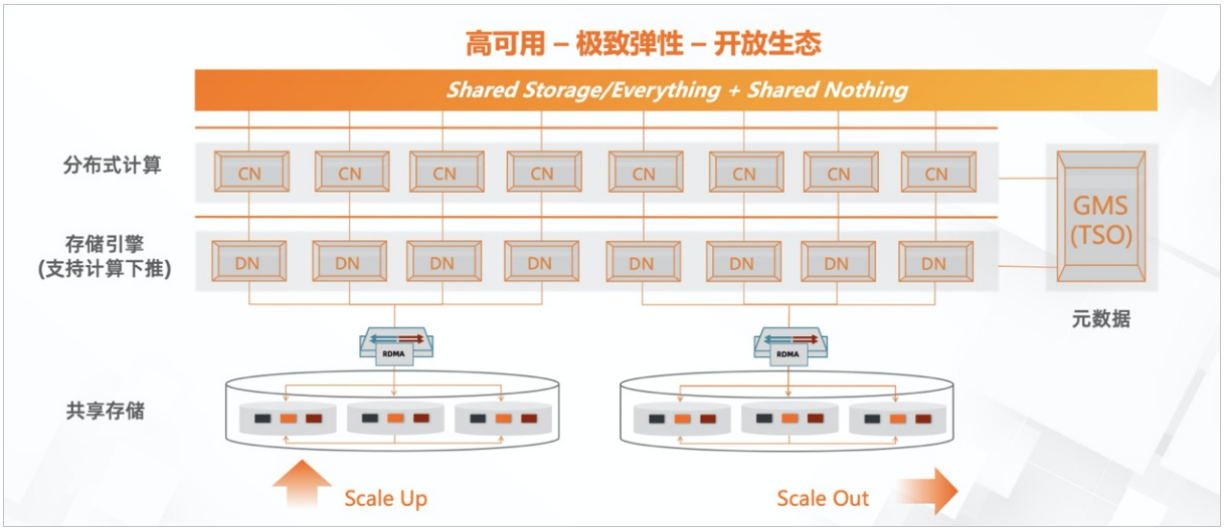
11.2.1. 技术架构

架构演进理念

当前，分布式领域有3大技术方向：Sharding技术，NewSQL原生分布式技术，云原生DB技术。每种分布式都有其独特的优势和特点。PolarDB-X 1.0的架构继承了DRDS和X-DB技术的稳定性，结合了PolarDB的云原生技术，融入了NewSQL对于分布式数据一致性的能力，为用户提供新的“云原生+分布式”的产品体验。

整体架构

整体架构如下：



部署架构

- 支持VPC、IP白名单、非对称账号密码、TDE等方式，确保数据服务安全。
- 使用独享高性能物理资源、实例间充分隔离，支持多可用区实例，确保数据服务稳定。
- 支撑运维系统采用多地域隔离部署，核心数据服务SLA与运维管控SLA解绑，确保运维体系稳定。

架构优势

- 高可用

  经过阿里多年双11验证的X-DB(X-Paxos共识协议能力)，提供数据强一致，保证节点故障切换时RPO=0。另外支持多样化的部署和容灾能力，比如基于Paxos强同步的同城三机房、三地五中心，另外搭配binlog异步复制的两地三中心、异地灾备、异地多活等。尤其在异地长距离传输上，基于Batching & Pipelining进行网络优化来提升性能。

- 高兼容

  PolarDB-X 1.0主要兼容MySQL，包括SQL、函数类型等，技术上引入全局时间授时服务，提供全局一致性的分布式事务能力，通过TSO+2PC提供数据库完整的ACID能力，满足分布式下的Read-Commited/Repeatable-Read的隔离级别。同时在分布式事务的基础上，提供全局二级索引能力，通过事务多写保证索引和主表数据强一致的同时，引入基于代价的CBO优化器实现索引选择。除此以外，在元数据和生态对接层面，PolarDB-X 1.0基于Online DDL的技术提供了分布式下元数据的一致性。同时硬件层面，兼容主流国产操作系统和芯片认证，比如麒麟、鲲鹏、海光等。

- 高扩展

PolarDB-X 1.0基于Share-Nothing的架构支持水平扩展，同时支持数据库在线扩缩容能力，在OLTP场景下可支持千万级别的并发、以及PB级别的数据存储规模，同样在OLAP场景下，引入MPP并行查询技术，扩展机器后查询能力可线性提升，满足TPC-H等的复杂报表查询诉求。

- 极致弹性

  PolarDB-X 1.0结合PolarDB云原生的技术，可以基于PolarDB的共享存储+RDMA网络优化能力，提供秒级备份、极速弹性、以及存储按需扩展的能力。基于共享存储的基础上，结合分布式的多点写入能力，可以在不迁移数据的前提下提供秒级弹性的能力，给到用户完全不一样的弹性体验。

- 开放生态

  PolarDB-X 1.0全面拥抱和坚定MySQL的开源生态，做到代码完全自主可控的同时满足分布式MySQL的兼容性，架构做到简单开放，只要具备一定MySQL背景的同学即可完成持续运维。除此以外，PolarDB-X 1.0和阿里云的数据库生态有完整的闭环对接，支持如DTS/DBS/DMS等，可打通阿里云的整个大生态。

11.2.2. 技术原理

扩展性原理

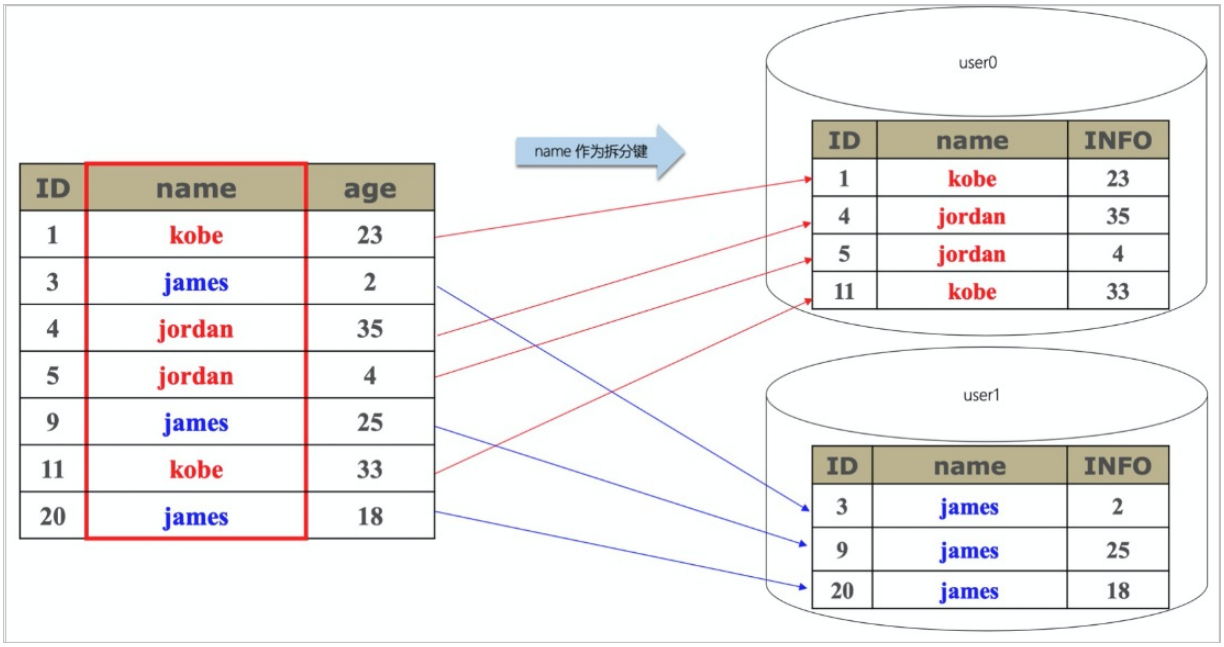
扩展性本质在于分而治之，PolarDB-X 1.0计算资源通过水平拆分（分库分表）和垂直拆分，将数据分散到多个存储资源MySQL以实现获取数据读写并发和存储容量分散的效果。

水平拆分（分库分表）

PolarDB-X 1.0具备数据水平拆分的能力，将数据库数据按某种规则分散存储到多个稳定的MySQL数据库上。这些MySQL数据库可分布于多台机器乃至跨机房，对外服务（增删改查）尽可能保证如同单MySQL数据库体验。拆分后，在MySQL上物理存在的数据库称为分库，物理的表称为分表（每个分表数据是完整数据的一部分）。PolarDB-X 1.0通过在不同MySQL实例上挪动分库，实现数据库扩容，提升PolarDB-X 1.0数据库总体访问量和存储容量。

您可以通过一定的计算或路由规则放置数据，实现将数据分散到多个存储资源MySQL中。PolarDB-X 1.0具备丰富的算法来应对各种场景。

数据拆分原理如下图所示：



计算扩展性

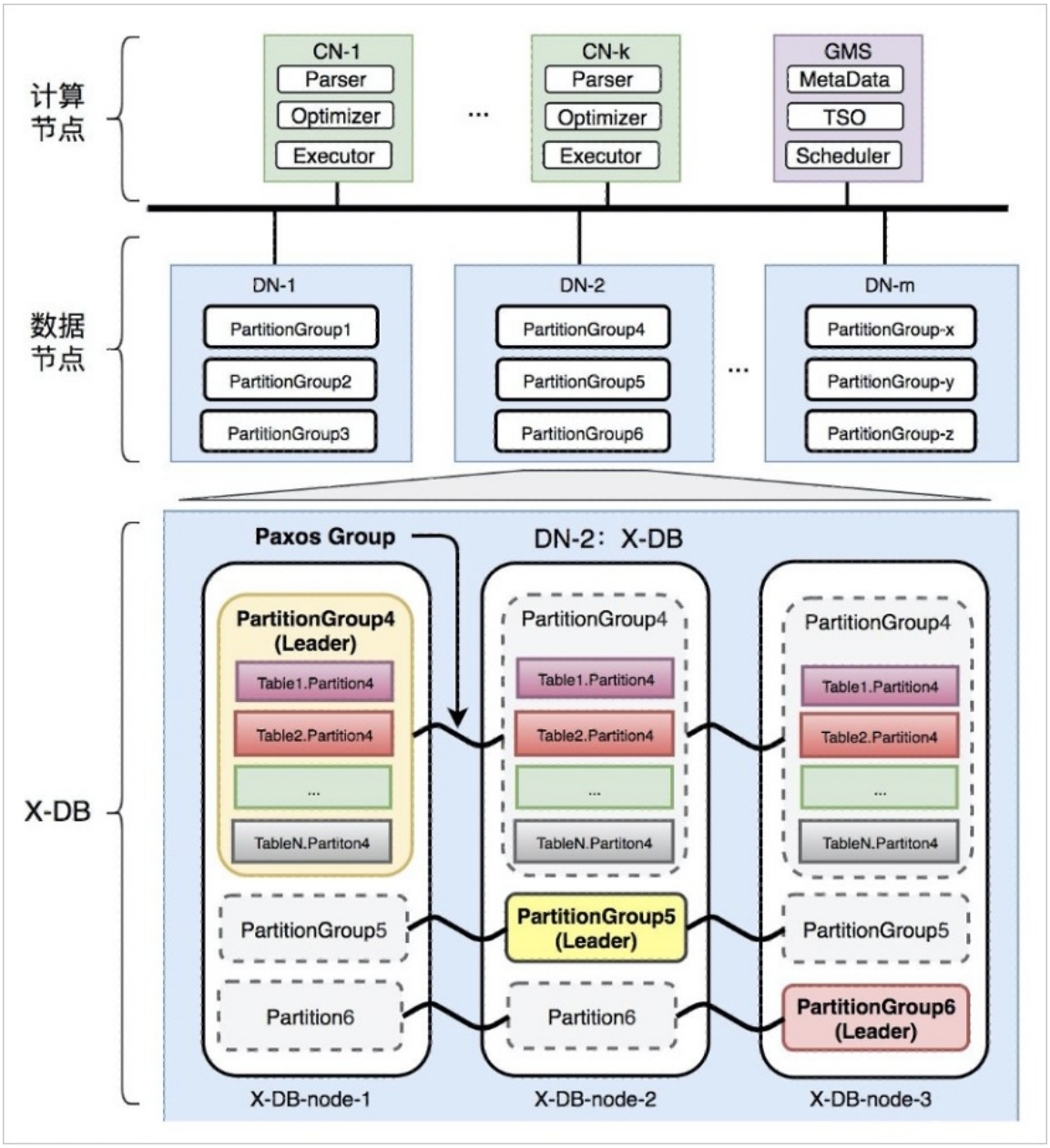
无论是水平拆分还是垂直拆分，PolarDB-X 1.0常常碰到需要对远超单机容量数据进行复杂计算的需求，例如需要执行多表JOIN、多层嵌套子查询、Grouping、Sorting、Aggregation等组合的SQL操作语句。

针对这类在线数据库上复杂SQL的处理， PolarDB-X 1.0额外扩展了单机并行处理器（Symmetric Multi-Processingy，简称SMP）和多机并行处理器（DAG）。前者完全集成在PolarDB-X内核中；而对于后者，PolarDB-X 1.0构建了一个计算集群，能够在运行时动态获取执行计划并进行分布式计算，通过增加节点提升计算能力。

平滑扩容

PolarDB-X 1.0扩容通过增加RDS/MySQL实例数，将原有的分库迁移到新的RDS/MySQL实例上，达到扩容的目标。PolarDB-X 1.0采用基于存储计算分离的Shared-Nothing架构，最大限度地发挥了云数据库的弹性扩展能力。

架构图如下：



如上图所示，PolarDB-X 1.0整个架构核心可分为3个部分： GMS(Global Meta Service)：负责管理分布式下库、表、列、分区等的元数据，以及提供TSO服务； CN(Compute Node)：负责提供分布式的SQL引擎与强一致事务引擎； DN(Data Node)：提供数据存储服务，负责存储数据与副本数据的强一致复制。

PolarDB-X 1.0存储层使用的是X-DB。因此，每一个DN节点就是一个X-DB实例。X-DB是在MySQL的基础之上基于X-Paxos打造的具备跨可用区跨地域容灾的高可用数据库，使用InnoDB存储引擎并完全兼容MySQL语法，它能提供Schema级别的多点写与Paxos Group的Leader调度的能力。PolarDB-X 1.0的分区副本的高可用与故障容灾就是建立在X-DB基础上的。

PolarDB-X 1.0扩容原理步骤如下：

1. 创建扩容计划

   选择新增加RDS/MySQL，并选定需要迁移到新RDS/MySQL实例上的分库，提交任务后系统自动在目标RDS/MySQL上创建数据库和账号，并提交任务进行数据迁移同步。

2. 全量迁移

   系统选择当前时间之前的一个时间点，将这个时间点之前的数据进行全量的数据复制迁移。

3. 增量数据同步

   完成全量迁移后，基于全量迁移开始之前时间点的增量变更日志进行增量同步，最终原分库和目标分库数据实时同步。

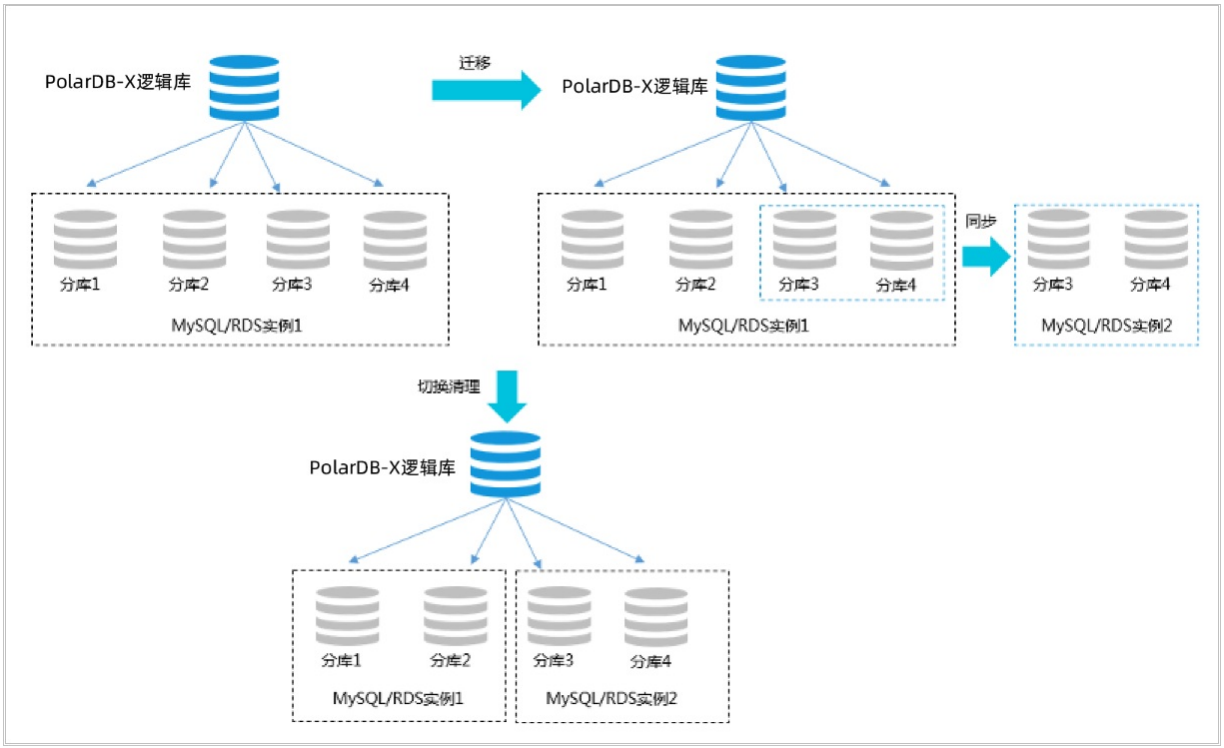
4. 数据校验

   增量达到准实时同步后，系统自动做全数据校验，并且订正因为同步延迟造成的不一致数据。

5. 应用停写和路由切换

   校验完成后，并且增量依然维持准实时同步，业务选定时间进行切换，为确保数据严格一致，建议应用停服（也可以不停，但可能面临同一条数据高并发写入覆盖问题），引擎层进行分库规则的路由切换，将后续流量转向新库，切换过程秒级完成。

扩容示意图下：



为了保证数据本身的安全，便于扩容回滚，在路由规则切换完成后，数据同步依然会运行，直到数据运维人员确认服务正常后在控制台主动发起旧分库数据的清理。

整个扩容过程对上层的业务正常服务几乎没有影响（如果RDS/MySQL实例规格过小或者压力过大则可能造成部分影响），切换时如果应用不停服，建议操作选择在数据库访问低谷期进行，降低同一条数据并发更新覆盖的概率。

读写分离

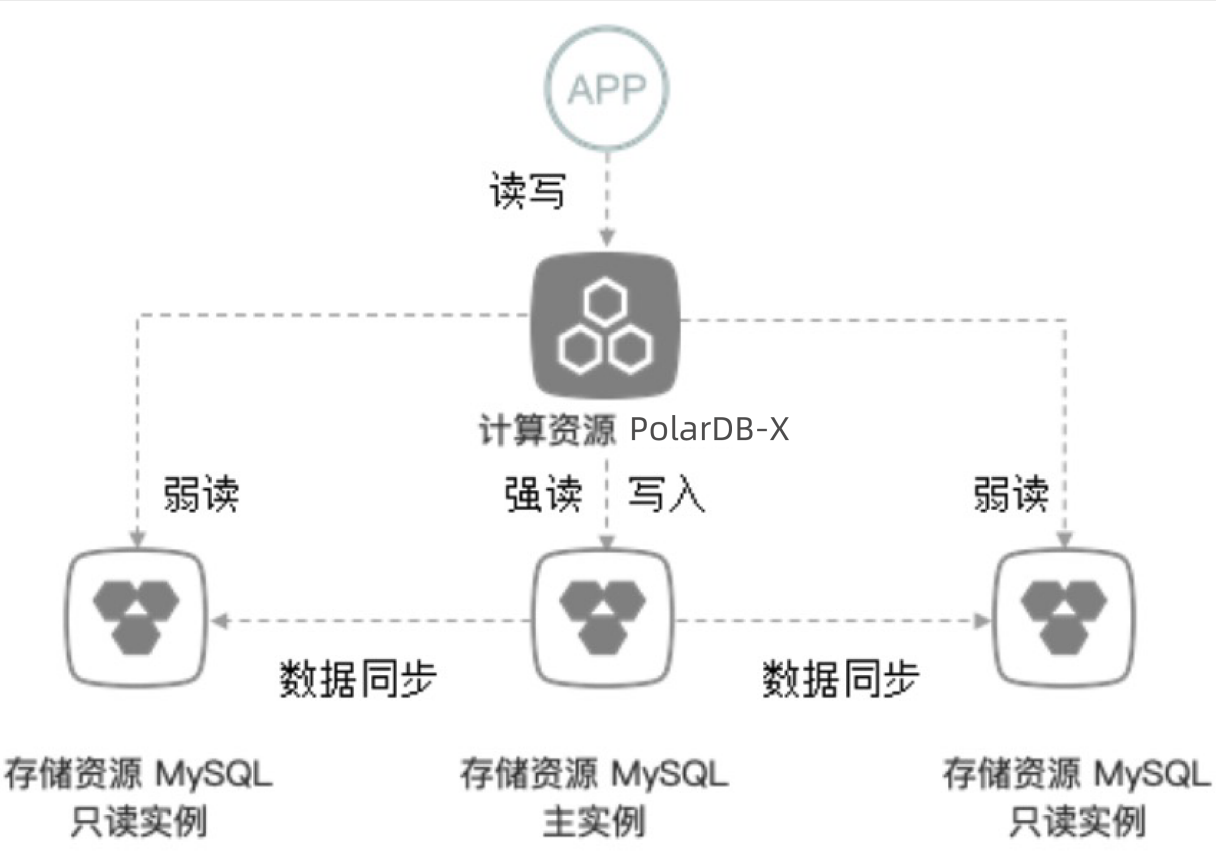
PolarDB-X 1.0的读写分离功能是基于RDS/MySQL只读实例所做的一种相对透明读流量切换策略。当PolarDB-X 1.0存储资源MySQL主实例的读请求较多、读压力比较大时，您可以通过读写分离功能对读流量进行分流，减轻存储层的读压力。

业务应用在能够忍受只读实例相对于主实例数据同步延迟的前提下，不需要修改代码，即可在PolarDB-X 1.0控制台中增加RDS/MySQL只读实例和调整读权重，将读流量按照需要的比例在RDS/MySQL主实例与多个RDS/MySQL只读实例之间调整，写操作和事务操作则统一走RDS/MySQL主实例。需要注意的是，主RDS/MySQL实例和只读RDS/MySQL存在数据同步延迟，并且在发生大的DDL或者数据订正时，有可能导致分钟级别以上的延迟，所以需要业务忍受该情况所带来的影响。添加只读实例可以使读性能线性提升。例如在初始有一个只读实例的情况下，挂载一个只读实例，读性能提升至原来两倍，挂载2个只读实例，则读性能为单个主库读性能的三倍。

读写分离流量分配与扩展

PolarDB-X 1.0读写分离功能采用了对应用透明的设计。在不修改应用程序任何代码的情况下，只需在控制台中调整读权重，即可实现将读流量按自定义的权重比例在存储资源MySQL/RDS主实例与多个存储资源只读实例之间进行分流，而写流量则不做分流全部到指向主实例。

添加只读实例可以使读性能线性提升。例如在初始有一个只读实例的情况下，挂载一个只读实例，读性能提升至原来2倍，挂载2个只读实例为单个主库的3倍。设置读写分离后，从存储资源MySQL主实例读取属于强读（即实时强一致读）；而只读实例上的数据是从主实例上异步复制而来存在毫秒级的延迟，因此从只读实例读取属于弱读（即非强一致性读）。您可以通过Hint指定那些需要保证实时性和强一致性的读SQL到主实例上执行。示意图如下：



读写分离对事务的支持

读写分离仅对显式事务（即需要显式提交或回滚的事务）以外的读请求（即查询请求）有效，写请求和显式事务中的读请求（包括只读事务）均在主实例中执行，不会被分流到只读实例。

常见的读、写请求SQL语句包括：

- 读请求：SELECT、SHOW、EXPLAIN、DESCRIBE。
- 写请求：INSERT、REPLACE、UPDATE、DELETE、CALL。

非拆分模式的读写分离

PolarDB-X 1.0的读写分离可以在非拆分模式下独立使用。

PolarDB-X 1.0控制台上创建PolarDB-X 1.0数据库时，在选定一个数据库实例的情况下，可以选择将底层数据库实例下的一个逻辑数据库直接引入PolarDB-X 1.0做读写分离，不需要做数据迁移。

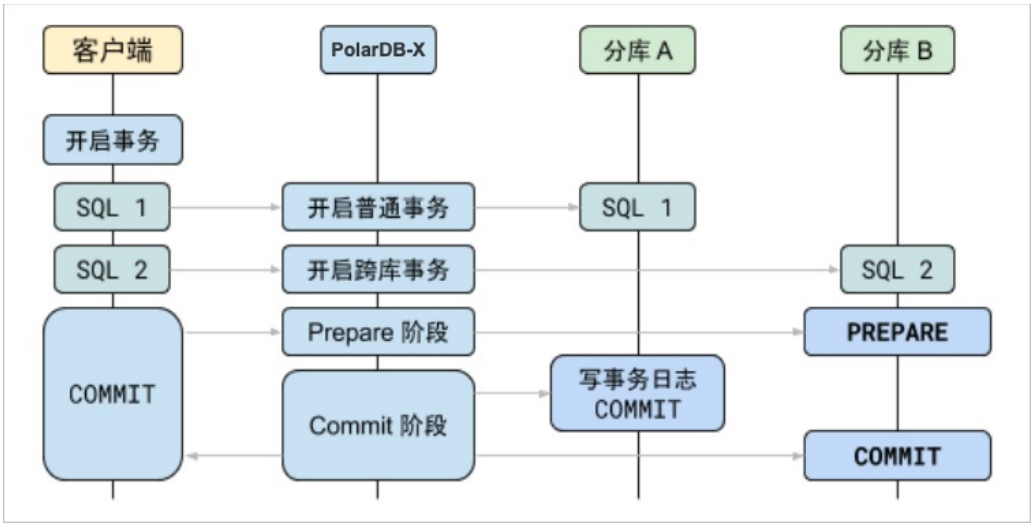
分布式事务

分布式事务通常使用二阶段提交来保证事务的原子性（Atomicity）和一致性（Consistency）。

二阶段事务会将事务分为以下两个阶段：

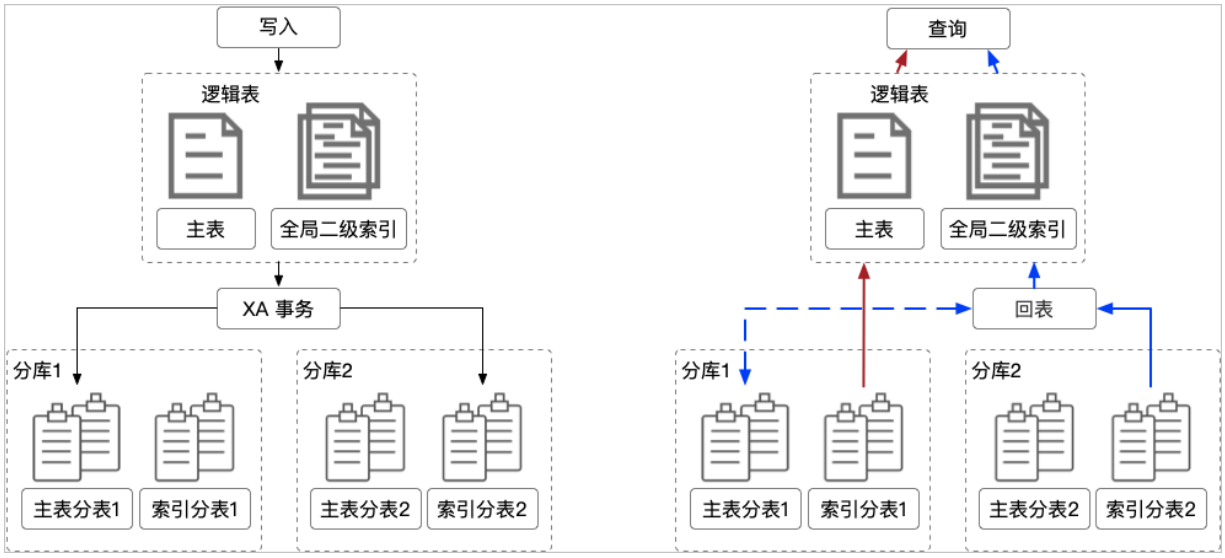
- 准备（PREPARE）阶段：在PREPARE阶段，数据节点会准备好所有事务提交所需的资源（例如加锁、写日志等）。
- 提交（COMMIT）阶段：在COMMIT阶段，各个数据节点才会真正提交事务。

当提交一个分布式事务时，PolarDB-X 1.0服务器会作为事务管理器的角色，等待所有数据节点（MySQL服务器）PREPARE成功，之后再向各个数据节点发送COMMIT请求。



全局二级索引

全局二级索引（Global Secondary Index，GSI）支持按需增加拆分维度，提供全局唯一约束。每个GSI对应一张索引表，使用XA多写保证主表和索引表之间数据强一致。



全局二级索引支持如下功能：

- 增加拆分维度。
- 支持全局唯一索引。
- XA多写，保证主表与索引表数据强一致。
- 支持覆盖列，减少回表操作，避免额外开销。
- Online Schema Change，添加GSI不锁主表。
- 支持通过HINT指定索引，自动判断是否需要回表。

11.2.3. 解决方案与客户案例

异地多活场景下的数据库方案

方案背景

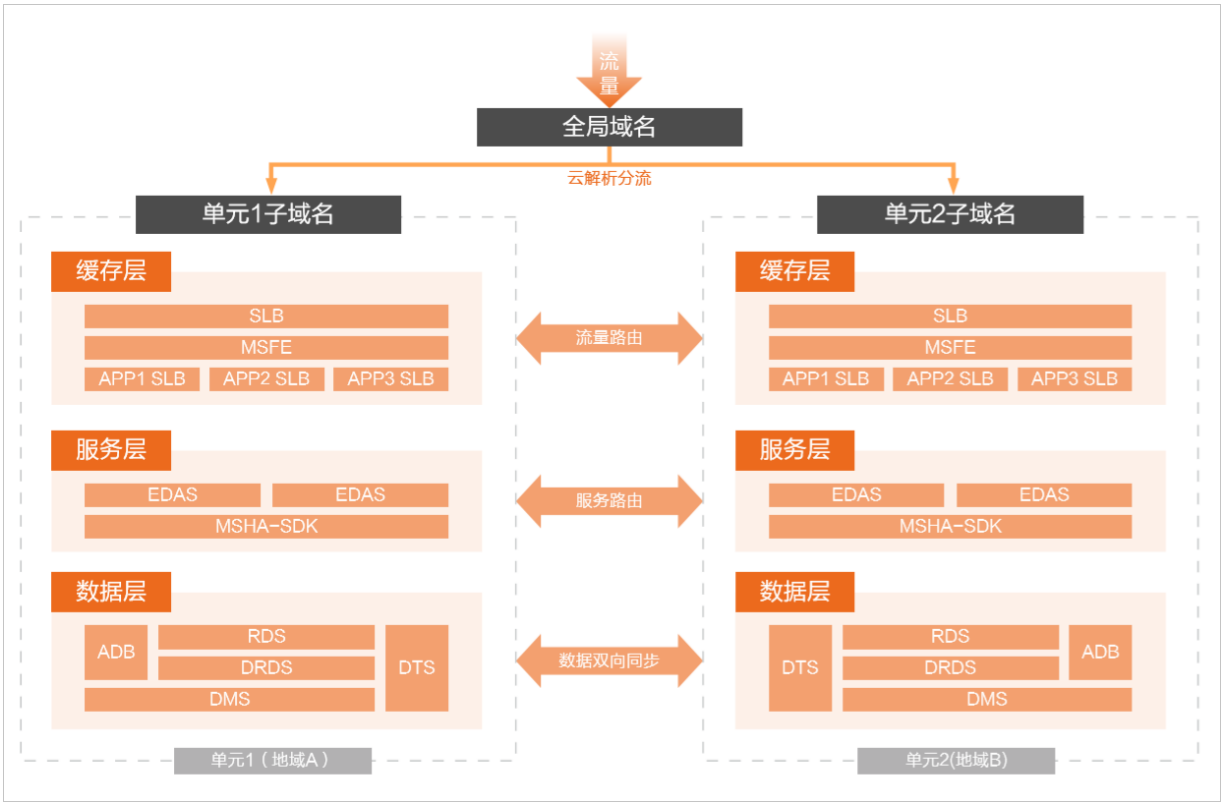
随着云计算的蓬勃发展，越来越多信息系统选择部署在云计算环境下，因此基于云产品为信息系统的服务能力和数据质量提供保障尤为重要。为了防止灾难性的故障如火灾、洪水、地震、区域电力中断或者人为破坏等对信息系统造成不可挽回的破坏，需要构建容灾系统来保障信息系统的可用性和安全性。

2007年，国务院信息化办公室联合银行、电力、民航、铁路、证券等八大重点行业，制定发布了国家标准GB/T20988-2007《信息系统灾难恢复规范》，明确规定了容灾能力的6个等级要求。企业在构建容灾系统时往往会参考国标等级，或者以此作为合规要求。然而，大部分传统容灾方案如同城容灾、同城双活、异地容灾、两地三中心等很难达到国标5-6级要求，同时还存在成本浪费，灾备单元健壮性不足等问题。

异地多活是新一代的容灾解决方案，在保证业务持续高可用的同时还能实现成本优化、地域级水平扩展、持续高可用等能力，本文会着重介绍阿里云主流数据库产品在异地多活场景下的解决方案。

方案架构

异地多活从业务视角来看是通过对业务做自顶向下的流量隔离来实现的，按照某一个分流维度对业务流量进行划分，并路由到不同的地域。整个部署架构分多个地域，每个地域称之为一个单元，其中某个单元又承担着整个多活架构的逻辑中心角色，提供一些中心化的服务能力（如sequence_分发，强一致读服务等）。每个单元内的业务架构分为接入层、服务层、数据层：



- 接入层

  业务流量通过租户侧DNS解析后按照权重分配到不同单元的接入层，进入接入层后，通过解析请求header/cookie中的分流标，对比自定义的分流规则，判断请求是否归属本单元，若归属本单元则走到本单元的服务层。否则流量需要转发到对端单元，根据不同的内网连通性可以走upstream或304跳转。

- 服务层

  服务层部署的是业务应用系统，包括中间件等。如果使用了阿里云的中间件如CSB、RPC框架、MQ等，可以通过相应产品的多活接入能力实现在服务层的流量路由和纠错。针对RPC服务，在多活场景中分为：单元化服务、中心化服务、普通服务三类：

  - 单元化服务：每个单元内完全自治的服务，是多活场景下的主要服务类型，会对流量进行单元归属判断，并进行流量纠错。
  - 中心化服务：强依赖中心单元的服务，流量都会被转发到中心单元，通过中心单元的服务访问中心单元的数据层。
  - 普通服务：不做任何改造的服务，在本单元内进行服务调用，并访问本单元的数据层。

  支持多活能力的主要服务类型是单元化服务，但是当业务系统比较复杂时，可能难以实现所有业务模块均按某一个维度划分，或者某些业务服务必须要单点部署以避免分布式部署的一致性问题，此时可由中心化服务和普通服务提供相应能力支持。

- 数据层

数据层解决数据库跨地域的部署与同步问题，并在灾难发生时对流量切换动作提供相应的数据质量保护策略。针对上层业务不同的服务类型提供UNIT和COPY两种数据同步策略：

UNIT类型：每个单元部署独立的数据库系统，单元之间通过DTS进行数据【双向】实时同步，保持每个单元都有全量数据，每个单元均可进行读写操作，读写流量会根据业务定制的分流策略进行单元写保护，这种同步策略用于支持服务层的单元化服务类型，是多活场景的核心同步策略。

COPY类型：每个单元部署独立的数据库系统，单元之间通过DTS进行数据【单向】实时同步，保持每个单元都有全量数据，中心单元可进行读写，非中心单元只提供读服务。这种同步策略用于支持中心化服务和普通服务，中心化服务路由回中心执行，普通服务可在单元内进行读。

方案应用场景

本解决方案适用于以下业务场景：

- 容灾能力要求高

  异地多活可以达到国标6级的容灾能力，适合对容灾方面有较高要求的业务、业务流量比较敏感的业务或业务的某些核心系统。

- 流量要求精细化管理

  异地多活支持多种流量管理策略，适合对流量管理有复杂需求的业务如按地域就近接入，按用户信息分配指定数据中心等。

- 业务快速发展

  异地多活实现业务按照单元级别（数据中心级别）实施水平扩展，业务定义好一个单元的服务部署配比后，可以以此为镜像快速部署多个分布全国的单元，实现容量快速扩充。

- 业务读多写少

  读多写少业务可以从业务行为上天然的规避数据异步复制的各类问题，并且业务接入多活的改造成本也较低，是比较适合实施异地多活的业务场景。

方案价值

异地多活的价值具体表现在如下方面：

- 业务即容灾

  传统异地灾备或两地三中心的异地灾备中心常态不提供服务，当发生地域级灾难时难以保证异地灾备中心的可用性，存在切换成功率低的问题，异地多活架构下各个地域数据中心不区分角色，全部常态承载业务流量，时刻保证业务系统健壮，各个数据中心既是业务体系也是容灾系统。

- 业务连续性保障

  异地多活架构下各个数据中心常态承接业务流量，故障发生时只需调拨入口流量即可实现容灾切换，实现分钟级的容灾切换。同时随着参与多活建设的数据中心数量增加，参与调拨流量的比例会相应减少，未参与调拨的业务流量可以实现对容灾切换的“0”感知。

- 业务高速发展支撑

  业务高速发展，受限于单个地域的有限资源，尤其是数据层存在单点性能瓶颈，除异地多活以外的全部容灾架构都只能在主生产中心执行写操作。地域多活实现业务级的流量闭环，各个数据中心均可读写，具备水平扩展能力以及跨地域的快速扩建能力。

- 流量有效隔离

  异地多活本质上是提供了一种自顶向下的流量隔离能力，业务具备在数据中心级别完全隔离的能力，各个数据中心承载的流量大小可灵活调配，在最小隔离数据中心内（例如承载1%流量），业务可灵活进行风险可控的技术演进，例如基础设施升级、新技术验证等。

- 成本有效控制

  在考虑单个地域发生地域级灾难的场景下，不论两地三中心还是异地灾备都需要在灾备中心部署可承载100%流量的系统资源，加上生产中心即达到200%的成本冗余。异地多活通过跨城多数据中心部署，有效分摊各个数据中心成本，实现成本小于200%冗余。

成功案例之联通新客户

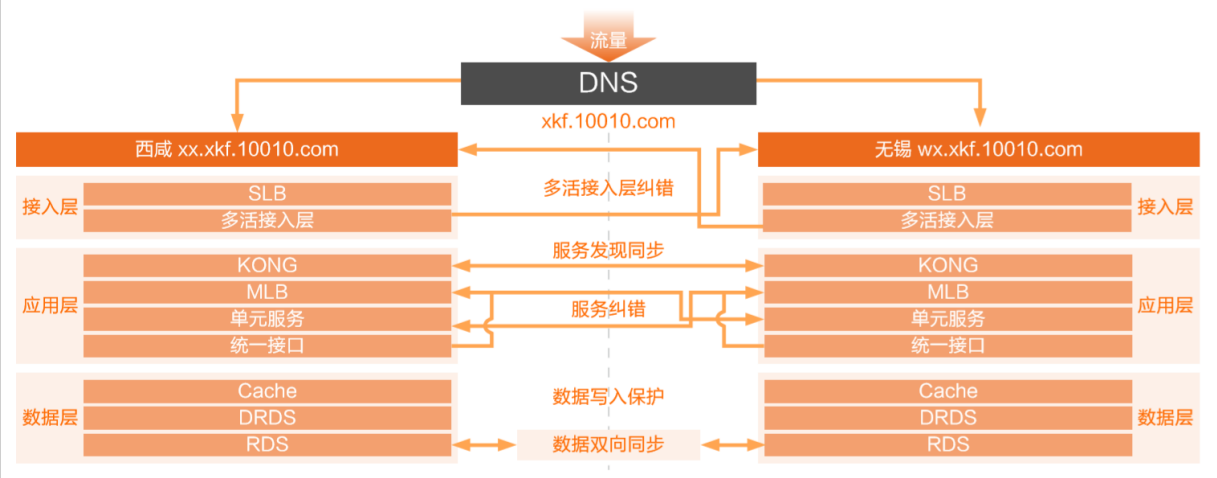
案例背景

联通新客服系统承担着联通全国的客服业务，对持续高可用能力有极高要求，同时也是联通向全站高可用演进的起点，其业务特点以TP业务为主。

案例架构

客户基于此方案，整合RDS、PolarDB-X 1.0、DTS、MSHA产品能力，实现了整个新客服系统7个业务中心的多活能力。

- RDS、PolarDB-X 1.0承载业务数据并对接多活管控系统。
- DTS实现数据的跨城实时同步和状态上报。
- MSHA实现多活流量管控和容灾切换动作。



案例效果

- 联通新客服系统的接入中心、外呼中心、业务支撑等7个业务实现按地域多活分流。
- 实现多次容灾演练，对多个省份进行切流，秒级完成切换，数据0丢失。
- 客户部署了两单元，常态两个单元均承载业务流量，充分利用两单元的资源。

11.3. API参考