ALIBABA CLOUD

阿里云

云原生分布式数据库 PolarDB-X SQL手册

文档版本：20220530

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.SQL使用限制

PolarDB-X 1.0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。本文将介绍相关SQL的使用限制。

SQL大类限制

- 暂不支持自定义数据类型或自定义函数。
- 暂不支持存储过程、触发器、游标。
- 暂不支持临时表。
- 暂不支持BEGIN...END、LOOP...END LOOP、REPEAT...UNTIL...END REPEAT、WHILE...DO...END WHILE等复合语句。
- 暂不支流程控制类语句（如IF或WHILE等）。
- 暂不支持外键。

小语法限制

- DDL
  - `CREATE TABLE tbl_name LIKE old_tbl_name` 不支持拆分表。
  - `CREATE TABLE tbl_name SELECT statement` 不支持拆分表。
  - 暂不支持同时RENAME多表。
  - 暂不支持ALTER TABLE修改拆分字段。
  - 暂不支持跨Schema的DDL（例如 `CREATE TABLE db_name.tbl_name (... )` ）。

  更多关于DDL的信息，请参见DDL。
- DML
  - 暂不支持SELECT INTO OUTFILE、INTO DUMPFILE和INTO var_name。
  - 暂不支持STRAIGHT_JOIN和NATURAL JOIN。
  - 暂不支持在 UPDATE SET 子句中使用子查询。
  - 暂不支持INSERT DELAYED语法。
  - 暂不支持SQL中对于变量的引用和操作（例如 `SET @c=1, @d=@c+1; SELECT @c, @d` ）。
  - 暂不支持在柔性事务中对广播表进行INSERT、REPLACE、UPDATE或DELETE操作。

  更多关于DML的信息，请参见DML。
- 子查询
  - 不支持HAVING子句中的子查询，JOIN ON条件中的子查询。
  - 等号操作行符的标量子查询（The Subquery as Scalar Operand）不支持ROW语法。

  更多关于子查询的信息，请参见子查询。
- 数据库管理
  - SHOW WARNINGS语法不支持LIMIT和COUNT的组合。
  - SHOW ERRORS语法不支持LIMIT和COUNT的组合。
- 运算符

  暂不支持 `':='` 赋值运算符。

更多关于运算符的信息，请参见运算符。

- 函数
  - 暂不支持全文检索函数。
  - 暂不支持XML 函数。
  - 不支持GTID 函数。
  - 不支持企业加密函数。

  更多关于函数的信息，请参见函数简介。
- 关键字
  - 暂不支持MILLISECOND。
  - 暂不支持MICROSECOND。

2.拆分函数使用说明

2.1. 拆分函数概述

PolarDB-X 1.0是一个支持既分库又分表的数据库服务。本文将介绍PolarDB-X 1.0拆分函数的相关信息。

拆分方式

在PolarDB-X 1.0中，一张逻辑表的拆分方式由拆分函数（包括分片数目与路由算法）与拆分键（包括拆分键的 MySQL 数据类型）共同定义。只有当PolarDB-X 1.0使用了相同的拆分函数和拆分键时，才会被认为分库与分表使用了相同的拆分方式。相同的拆分方式让PolarDB-X 1.0可以根据拆分键的值定位到唯一的物理分库和物理分表。当一张逻辑表的分库拆分方式与分表拆分方式不一致时，若SQL查询没有同时带上分库条件与分表条件，则PolarDB-X 1.0在查询过程会进行全分库扫描或全分表扫描操作。

拆分函数对分库、分表的支持情况

拆分函数	说明	是否支持用于分库	是否支持用于分表
HASH	简单取模	是	是
STR_HASH	截取字符串子串	是	是
UNI_HASH	简单取模	是	是
RIGHT_SHIFT	数值向右移	是	是
RANGE_HASH	双拆分列哈希	是	是
MM	按月份哈希	否	是
DD	按日期哈希	否	是
WEEK	按周哈希	否	是
MMDD	按月日哈希	否	是
YYYYMM	按年月哈希	是	是
YYYYWEEK	按年周哈希	是	是
YYYYDD	按年日哈希	是	是

拆分函数对全局二级索引的支持情况

- PolarDB-X 1.0支持全局二级索引，从数据存储的角度看，每个GSI对应一张用于保存索引数据的逻辑表，称为索引表。
- PolarDB-X 1.0还支持创建GSI时指定索引表的拆分方式，并且对拆分函数的支持范围与普通逻辑表相同。创建GSI的详细语法，请参见使用全局二级索引。

拆分函数对数据类型支持情况

拆分函数	数据类型										
	INT	BIGINT	MEDIUMINT	SMALLINT	TINYINT	VARCHAR	CHAR	DATE	DATETIME	TIMESTAMP	其他类型
HASH	√	√	√	√	√	√	√	×	×	×	×
UNI_HASH	√	√	√	√	√	√	√	×	×	×	×
RANGE_HASH	√	√	√	√	√	√	√	×	×	×	×
RIGHT_SHIFT	√	√	√	√	√	×	×	×	×	×	×
STR_HASH	×	×	×	×	×	√	√	×	×	×	×
MM	×	×	×	×	×	×	×	√	√	√	×
DD	×	×	×	×	×	×	×	√	√	√	×
WEEK	×	×	×	×	×	×	×	√	√	√	×
MMDD	×	×	×	×	×	×	×	√	√	√	×
YYYYMM	×	×	×	×	×	×	×	√	√	√	×
YYYYWEEK	×	×	×	×	×	×	×	√	√	√	×
YYYYDD	×	×	×	×	×	×	×	√	√	√	×

拆分函数的语法说明

PolarDB-X 1.0兼容MySQL的DDL表操作语法，并添加了 `drds_partition_options` 的分库分表关键字，具体操作语法如下所示：

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [drds_partition_options]
    [partition_options]
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options]
    [drds_partition_options]
    [partition_options]
    select_statement
drds_partition_options:
    DBPARTITION BY
        { {HASH|YYYYMM|YYYYWEEK|YYYYDD|...}([column])}
        [TBPARTITION BY
            { {HASH|MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|...}(column)}
          [TBPARTITIONS num]
        ]
```

2.2. HASH

本文将介绍HASH函数使用方式。

注意事项

HASH函数的算法是简单取模，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。

路由方式

- 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））。

使用场景

HASH函数主要应用与如下场景：

- 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);
```

2.3. STR_HASH

本文将介绍STR_HASH函数使用方式。

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，则会接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。
- PolarDB-X 1.0的实例版本需为5.3.5或以上，关于实例版本请参见版本说明。

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，某个互联网金融应用是按年月（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。

- 典型的点查场景

  STR_HASH拆分函数适用于使用字符串类型作为拆分键并且是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。

函数定义

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，计算具体的物理分片，具体函数如下所示：

```
STR_HASH( shardKey [, startIndex, endIndex [, valType [, randSeed ] ] ] )
```

参数说明

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）。

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）。
	说明 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 1.0将截取后的子串当作字符串类型来计算路由。 • 1：表示PolarDB-X 1.0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）

参数	说明
randSeed	当子串以字符串类型来计算路由的哈希值时PolarDB-X 1.0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）。

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。

2.4. UNI_HASH

本文将介绍UNI_HASH的使用方式。

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。
- PolarDB-X 1.0实例的版本需为5.1.28-1508068或以上，关于实例版本请参见版本说明。

路由方式

UNI_HASH主要用于以下场景：

- 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。

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：

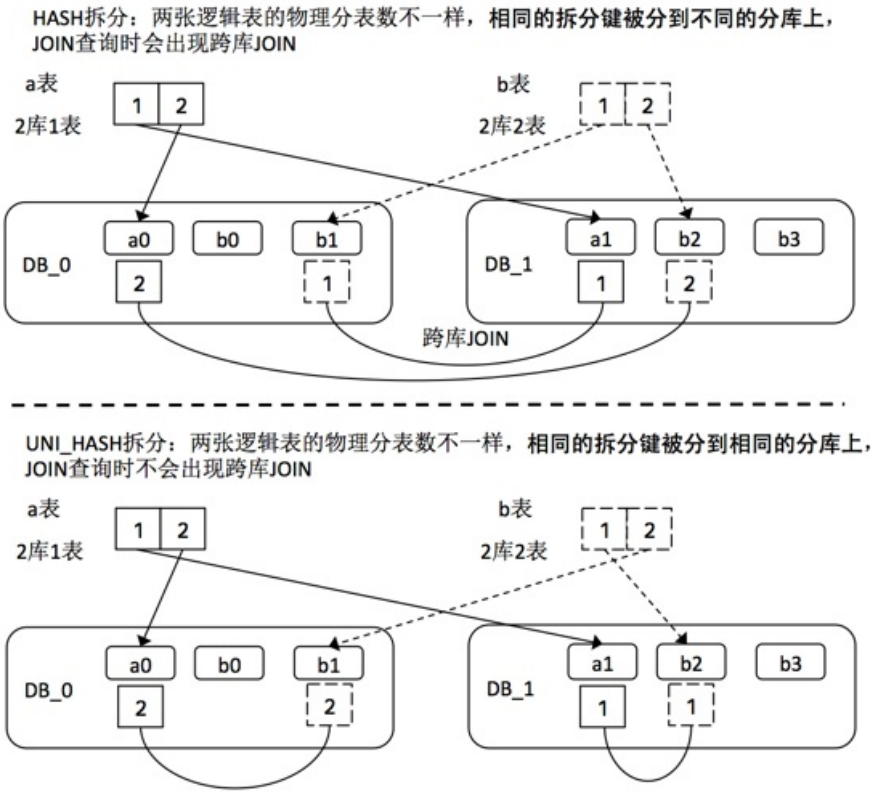
```
create table test_hash_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by UNI_HASH(ID)
tbpartition by UNI_HASH(ID) tbpartitions 4;
```

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。	

对比场景	UNI_HASH	HASH
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：



2.5. RANGE_HASH

本文将介绍RANGE_HASH函数的使用方式。

适用场景

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
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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。

示例

假设PolarDB-X 1.0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;
```

2.6. RIGHT_SHIFT

本文将介绍RIGHT_SHIFT函数的使用方式。

使用限制

- 拆分键的类型必须整数类型。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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。

> **说明** 移位的数目不能超过整数类型所占有的位数目。

使用场景

当拆分键的大部分的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，如果直接对面值进行取余散列，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4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;
```

2.7. MM

本文将介绍MM函数的使用方式。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数而不是分库函数使用。
- 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。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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;
```

2.8. DD

本文将介绍DD函数的使用方式。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数而不是分库函数使用。
- 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。

- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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;
```

2.9. WEEK

本文将介绍WEEK函数的使用方式。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数而不是分库函数使用。
- 按WEEK进行分表，由于一周共有7天，所以各分库的分表数不能超过7。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

根据分库键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。

使用场景

WEEK函数适用于按一周中各个日期进行分表，分表的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

假设需要先按ID对用户进行分库，再将 `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，则您可以使用如下的建表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;
```

2.10. MMDD

本文将介绍MMDD函数的使用方式。

使用限制

- 拆分键的类型必须是DATE、DATETIME或TIMESTAMP中的一种。
- 只能作为分表函数而不是分库函数使用。
- 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。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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;
```

2.11. YYYYDD

本文将介绍YYYYDD函数的使用方式。

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。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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 1.0里已经拥有2个节点，每个节点默认有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。

YYYYDD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 1.0已有16个分库，所以每个分库应该建46张物理分表（732/16=45.75，取整为46，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 46;
```

2.12. YYYYMM

本文将介绍YYYYMM函数的使用方式。

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。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本，关于实例版本请参见版本说明。

路由方式

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，完成路由计算。

例如，`YYYYMM('2012-12-31 12:12:12')` 函数等价于按照 `(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)` 一起使用。

使用示例

假设PolarDB-X 1.0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 1.0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;
```

2.13. YYYYWEEK

本文将介绍YYYYWEEK函数的使用方式。

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。
- PolarDB-X 1.0实例的版本需为5.1.28-1320920或以上版本。关于实例版本请参见版本说明。

路由方式

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，完成路由计算。

例如，`YYYYWEEK('2012-12-31 12:12:12')` 函数等价于按照 `(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 1.0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 1.0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;
```

3.DDL任务管理

3.1. 概述

PolarDB-X 1.0在V5.3.12及以上的实例版本中引入了新的DDL执行引擎，开始支持DDL的任务管理，包括DDL执行过程中的任务状态查看、失败DDL任务的恢复和回滚等。

DDL任务管理中的主要概念

了解以下概念，将帮助您更好地使用DDL任务管理功能：

- DDL任务：一次DDL语句的执行过程对应一个DDL任务。
- 管理语句：对DDL任务进行查看或者操作的PolarDB-X 1.0专有的SQL语句。
- 任务ID：DDL任务的唯一标识，是一个64位有符号的长整型数值。
- 任务状态：DDL任务的内部状态。

关于任务管理语句的语法和用法，请参见任务管理语句。

3.2. 任务管理语句

任务管理语句是PolarDB-X 1.0专有的扩展SQL语句，可用于查看DDL任务的状态、恢复或回滚失败的DDL任务等。本文将详细介绍任务管理语句的语法和用法。

查看任务

您可以在DDL队列中查看正在执行（即非PENDING状态）的任务或失败待处理（即PENDING状态）的任务详情。

> **说明** 已完成（即COMPLETED状态）的任务会被自动清理，无法通过SHOW DDL语句查看。

- 语法

```
SHOW [FULL] DDL
```

参数	说明

参数	说明
FULL	显示DDL任务的所有信息，若不带该参数则结果集中只显示如下常用信息： ○ JOB_ID ○ OBJECT_SCHEMA ○ OBJECT_NAME ○ JOB_TYPE ○ PHASE ○ STATE ○ PROGRESS ○ START_TIME ○ END_TIME ○ ELAPSED_TIME ○ REMARK ○ PHY_PROCESS ○ BACKFILL_PROGRESS

- 结果集中各字段的含义

字段	含义
JOB_ID	DDL任务唯一标识，取值需为64位有符号长整型数值。
PARENT_JOB_ID	该DDL任务的父任务唯一标识，取值需为64位有符号长整型数值。 **说明** 当目标DDL任务为独立无父任务时，该参数取值为0。
SERVER	执行DDL任务的DRDS节点信息。
OBJECT_SCHEMA	DDL任务对象的Schema名称，即当前数据库名称。
OBJECT_NAME	DDL任务对象名称，例如当前执行DDL的表名称。
NEW_OBJECT_NAME	DDL任务新对象名称。 **说明** 仅当DDL任务类型为RENAME TABLE时显示该参数，表示目标表的新名称。
JOB_TYPE	DDL任务类型。
PHASE	DDL任务当前所处的阶段。
STATE	DDL任务当前所处的状态。
PROGRESS	DDL任务执行进度。

字段	含义
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	DDL任务截止到任务查看时已经消耗的时间，单位为毫秒。
DDL_STMT	原始的DDL语句。
REMARK	DDL任务的备注信息。 **说明** 当DDL任务状态为PENDING时，该参数会显示DDL任务失败的原因。

- 示例

  创建一个既分库又分表的拆分表，执行过程中查看状态。

  i. 在一个连接上执行建表DDL:

  ```
  mysql> create table test_mdb_mtb (c1 int not null auto_increment primary key, c2 varc
  har(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64;
  ```

  ii. 在另一个连接上查看DDL任务执行状态:

  ```
  mysql> show full ddl\G
  *************************** 1. row ***************************
      JOB_ID: 1103792075578957824
  PARENT_JOB_ID: 0
      SERVER: 1:102:10.81.69.55
  OBJECT_SCHEMA: ddltest
  OBJECT_NAME: test_mdb_mtb
  NEW_OBJECT_NAME:
    JOB_TYPE: CREATE_TABLE
       PHASE: EXECUTE
       STATE: RUNNING
    PROGRESS: 90%
  START_TIME: 2019-08-29 14:29:58.787
    END_TIME: 2019-08-29 14:30:07.177
  ELAPSED_TIME(MS): 8416
    DDL_STMT: create table test_mdb_mtb (c1 int not null auto_increment primary key, c2
  varchar(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64
      REMARK:
  ```

恢复任务

恢复失败待处理（即PENDING状态）的DDL任务。

说明 恢复任务前，建议您先通过SHOW DDL仔细查看任务中断或者失败的原因，找到并解决导致DDL任务失败的因素后，再执行恢复任务的操作，否则恢复任务可能会遭遇同样的问题导致失败。

- 语法

```
RECOVER DDL { ALL | <job_id> [ , <job_id> ] ... }
```

参数	说明
ALL	恢复所有处于PENDING状态的DDL任务，被恢复的任务会串行执行，请慎用此参数。
job_id	通过SHOW DDL查看到的处于PENDING状态的任务ID。

- 示例

  创建一个既分库又分表的拆分表，任务执行过程中被中断，通过SHOW DDL查看状态和 `job_id` ，然后用RECOVER DDL恢复任务，直至该表创建完成。

  i. 建表DDL任务在执行过程中被中断：

  ```
  mysql> create table test_mdb_mtb (c1 int not null auto_increment primary key, c2 varc
  har(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64;
  ^C^C -- query aborted
  ```

  ii. 查看DDL任务的信息，被中断的DDL任务处于PENDING状态：

  ```
  mysql> show ddl\G
  *************************** 1. row ***************************
       JOB_ID: 1103796219480006656
  OBJECT_SCHEMA: ddltest
  OBJECT_NAME: test_mdb_mtb
    JOB_TYPE: CREATE_TABLE
        PHASE: EXECUTE
        STATE: PENDING
    PROGRESS: 33%
  START_TIME: 2019-08-29 14:46:26.769
    END_TIME: 2019-08-29 14:46:29.691
  ELAPSED_TIME(MS): 2922
    DDL_STMT: create table test_mdb_mtb (c1 int not null auto_increment primary key, c2
  varchar(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64
       REMARK: The job has been interrupted unexpectedly
  ```

  iii. 使用RECOVER DDL恢复该任务：

  ```
  mysql> recover ddl 1103796219480006656;
  Query OK, 0 rows affected (7.28 sec)
  ```

  iv. 通过CHECK TABLE检查该表的一致性：

  ```
  mysql> check table test_mdb_mtb;
  +---------------------------------------+-------+----------+----------+
  | TABLE                                 | OP    | MSG_TYPE | MSG_TEXT |
  +---------------------------------------+-------+----------+----------+
  | ddltest_1562056402230oymk.test_mdb_mtb | check | status   | OK       |
  +---------------------------------------+-------+----------+----------+
  1 row in set (2.24 sec)
  ```

回滚任务

回滚失败待处理（即PENDING状态）的DDL任务。

> **说明** 目前仅对CREATE TABLE和RENAME TABLE两种类型的DDL任务支持回滚。对于其它不支持回滚的DDL任务，建议恢复任务后，再执行其它DDL操作。

- 语法

```
ROLLBACK DDL <job_id> [ , <job_id> ] ...
```

参数	说明
job_id	通过SHOW DDL查看到的处于PENDING状态的任务ID。

- 示例

创建一个既分库又分表的拆分表，任务执行过程中被中断，通过SHOW DDL查看状态和 `job_id` ，然后用ROLLBACK DDL回滚任务。

i. 建表DDL任务在执行过程中被中断：

```
mysql> create table test_mdb_mtb (c1 int not null auto_increment primary key, c2 varc
har(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64;
^C^C -- query aborted
```

ii. 查看DDL任务的信息，被中断的DDL任务处于PENDING状态：

```
mysql> show ddl\G
*************************** 1. row ***************************
     JOB_ID: 1103797850607083520
OBJECT_SCHEMA: ddltest
OBJECT_NAME: test_mdb_mtb
  JOB_TYPE: CREATE_TABLE
      PHASE: EXECUTE
      STATE: PENDING
  PROGRESS: 40%
START_TIME: 2019-08-29 14:52:55.660
  END_TIME: 2019-08-29 14:52:58.885
ELAPSED_TIME(MS): 3225
  DDL_STMT: create table test_mdb_mtb (c1 int not null auto_increment primary key, c2
varchar(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64
     REMARK: The job has been interrupted unexpectedly
```

iii. 使用ROLLBACK DDL回滚该任务：

```
mysql> rollback ddl 1103797850607083520;
Query OK, 0 rows affected (6.42 sec)
```

iv. 回滚成功，该表不存在：

```
mysql> show tables like 'test_mdb_mtb';
Empty set (0.00 sec)
```

取消任务

取消正在执行中（即非PENDING状态）的DDL任务。

- 语法

```
CANCEL DDL <job_id> [ , <job_id> ] ...
```

参数	说明
job_id	通过SHOW DDL查看到的处于非PENDING状态的任务ID。

- 示例

  创建一个既分库又分表的拆分表，任务执行过程中通过CANCEL DDL取消，通过SHOW DDL查看状态和 `job_id` ，后续可以恢复或者回滚该任务。

  i. 在一个连接上执行建表DDL:

  ```
  mysql> create table test_mdb_mtb (c1 int not null auto_increment primary key, c2 varc
  har(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64;
  ```

  ii. 在另一个连接上通过SHOW DDL查看正在执行中的DDL任务:

  ```
  mysql> show ddl\G
  *************************** 1. row ***************************
     JOB_ID: 1103798959568478208
  OBJECT_SCHEMA: ddltest
  OBJECT_NAME: test_mdb_mtb
   JOB_TYPE: CREATE_TABLE
       PHASE: EXECUTE
       STATE: RUNNING
   PROGRESS: 26%
  START_TIME: 2019-08-29 14:57:20.058
   END_TIME: 2019-08-29 14:57:22.284
  ELAPSED_TIME(MS): 2243
   DDL_STMT: create table test_mdb_mtb (c1 int not null auto_increment primary key, c2
  varchar(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64
     REMARK:
  ```

  iii. 执行CANCEL DDL取消该DDL任务的执行:

  ```
  mysql> cancel ddl 1103798959568478208;
  Query OK, 2 rows affected (0.03 sec)
  ```

  iv. 再通过SHOW DDL查看，DDL任务已被取消执行并处于PENDING状态:

```
mysql> show ddl\G
*************************** 1. row ***************************
   JOB_ID: 1103798959568478208
OBJECT_SCHEMA: ddltest
OBJECT_NAME: test_mdb_mtb
 JOB_TYPE: CREATE_TABLE
     PHASE: EXECUTE
     STATE: PENDING
 PROGRESS: 87%
START_TIME: 2019-08-29 14:57:20.058
 END_TIME: 2019-08-29 14:57:28.899
ELAPSED_TIME(MS): 8841
 DDL_STMT: create table test_mdb_mtb (c1 int not null auto_increment primary key, c2
varchar(10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 64
   REMARK: ERR-CODE: [TDDL-4636][ERR_DDL_JOB_ERROR] The job '1103798959568478208' has
been cancelled.
```

删除任务

删除失败待处理（即PENDING状态）的DDL任务，并清理对应的缓存。

> **警告** 请谨慎操作REMOVE DDL删除任务。执行删除PENDING任务的操作后，可能会暴露DDL执行过程中的中间状态，从而影响后续操作。因此，若您不确定PENDING任务是否可以安全删除时，请不要执行REMOVE DDL语句删除任务，建议您优先使用恢复或者回滚任务解除PENDING状态。

- 语法

```
REMOVE DDL { ALL PENDING | <job_id> [ , <job_id> ] ... }
```

参数	说明
ALL PENDING	删除所有处于PENDING状态的任务，同时清理内部缓存。
job_id	通过SHOW DDL查看到的处于PENDING状态的任务ID。

- 示例

  数据库已有两张表，之间建立了参照完整性关系，当尝试删除父表时报错，因为存在参照完整性约束不允许删除，如果此时不想再执行删除表的操作，那么可以删除该DDL任务。

  i. 数据库中存在两张具有参照完整性关系的父子表：

```
mysql> show create table test_parent\G
*************************** 1. row ***************************
Table: test_parent
Create Table: CREATE TABLE `test_parent` (
`id` int(11) NOT NULL,
`pkey` int(11) NOT NULL,
`col` int(11) DEFAULT NULL,
PRIMARY KEY (`id`,`pkey`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`)
1 row in set (0.01 sec)
mysql> show create table test_child\G
*************************** 1. row ***************************
Table: test_child
Create Table: CREATE TABLE `test_child` (
`id` int(11) DEFAULT NULL,
`parent_id` int(11) DEFAULT NULL,
KEY `parent_id` (`parent_id`),
CONSTRAINT `test_child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `test_parent` (`i
d`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`parent_id`)
1 row in set (0.02 sec)
```

ii. 因为存在参照完整性约束，尝试删除父表报错：

```
mysql> drop table test_parent;
ERROR 4636 (HY000): [f518265d0066000][10.81.69.55:3306][ddltest]ERR-CODE: [TDDL-4636]
[ERR_DDL_JOB_ERROR] Not all physical operations have been done successfully: expected
9,
but done 0. Caused by: 1217:DDLTEST_15620564022300YMK_7WW7_0007:Cannot delete or upda
te a parent row: a foreign key constraint fails on `test_parent`;1217:DDLTEST_1562056
4022
300YMK_7WW7_0000:Cannot delete or update a parent row: a foreign key constraint fails
on `test_parent`;1217:DDLTEST_15620564022300YMK_7WW7_0002:Cannot delete or update a p
are
nt row: a
```

iii. 查看DDL任务：

```
mysql> show ddl\G
*************************** 1. row ***************************
     JOB_ID: 1103806757547171840
OBJECT_SCHEMA: ddltest
OBJECT_NAME: test_parent
  JOB_TYPE: DROP_TABLE
      PHASE: EXECUTE
      STATE: PENDING
  PROGRESS: 0%
START_TIME: 2019-08-29 15:28:19.240
  END_TIME: 2019-08-29 15:28:19.456
ELAPSED_TIME(MS): 216
  DDL_STMT: drop table test_parent
     REMARK: ERR-CODE: [TDDL-4636][ERR_DDL_JOB_ERROR] Not all physical operations hav
e been done successfully: expected 9, but done 0. Caused by: 1217:DDLTEST_1562056402
230OYMK_7WW7_0007:Cannot delete or update a parent row: a foreign key constraint fail
s on `test_pare ...
```

iv. 该DDL任务执行删除表的操作时，违反了参照完整性约束，因此删除操作并未真正执行，此时CHECK TABLE该表仍然是一致的：

```
mysql> check table test_parent;
+---------------------------------------+-------+----------+----------+
| TABLE                                 | OP    | MSG_TYPE | MSG_TEXT |
+---------------------------------------+-------+----------+----------+
| ddltest_1562056402230oymk.test_parent | check | status   | OK       |
+---------------------------------------+-------+----------+----------+
1 row in set (0.05 sec)
```

v. 但由于该表上有PENDING状态的任务存在，因此此时表处于不可访问状态：

```
mysql> show tables like 'test_parent';
Empty set (0.00 sec)
mysql> show create table test_parent;
ERROR 4642 (HY000): [f5185a78b066000][10.81.69.55:3306][ddltest]ERR-CODE: [TDDL-4642]
[ERR_UNKNOWN_TABLE] Unknown table 'ddltest.test_parent'
```

vi. 此时该删除表的任务并没有真正开始执行，表结构仍然一致，虽然看似可以选择回滚失败的DDL操作，但由于DROP TABLE并不允许回滚，回滚操作并不可行，因此必须选择删除该失败的DDL任务：

```
mysql> remove ddl 1103806757547171840;
Query OK, 1 row affected (0.02 sec)
```

vii. 删除该DDL任务后，表恢复正常访问的状态：

```
mysql> show tables like 'test_parent';
+-------------------+
| TABLES_IN_DDLTEST |
+-------------------+
| test_parent       |
+-------------------+
1 row in set (0.01 sec)
```

3.3. 控制参数与行为

您可以通过修改参数设置来改变DDL执行引擎的行为。本文将介绍如何修改DDL执行引擎相关参数。

DDL执行引擎相关参数

目前您可以在PolarDB-X 1.0控制台上自定义如下与DDL执行引擎相关的参数。

参数	影响范围	默认值
ENABLE_ASYNC_DDL	数据库级别、语句级别	TRUE（启用）
PURE_ASYNC_DDL_MODE	数据库级别、会话级别、语句级别	FALSE（禁用）
MAX_TABLE_PARTITIONS_PER_DB	数据库级别、语句级别	128

ENABLE_ASYNC_DDL

- 说明
  - 该参数默认启用，即采用新的DDL执行引擎。
  - 禁用该参数后，PolarDB-X 1.0将使用5.3.12版本之前的DDL执行引擎，PURE_ASYNC_DDL_MODE和MAX_TABLE_PARTITIONS_PER_DB参数将不会生效。建议您提交工单咨询技术支持后，再决定是否禁用该参数。
- 用法
  - 数据库级别：通过PolarDB-X 1.0控制台的参数设置进行调整，整个数据库范围内生效，详情请参见参数设置。
  - 语句级别：通过在DDL语句前增加HINT的方式 `/*+TDDL:cmd_extra(ENABLE_ASYNC_DDL=FALSE)*/` ，可以实现语句级别的控制，仅对该语句生效。

PURE_ASYNC_DDL_MODE

- 说明
  - 该参数仅在 `ENABLE_ASYNC_DDL` 为TRUE时生效。
  - 禁用该参数时，客户端连接PolarDB-X 1.0执行DDL时是同步阻塞的模式，即DDL任务执行完毕后再返回请求结果。客户端与PolarDB-X 1.0连接被中断后，正在执行的DDL任务也可能被中断。
  - 启用该参数后，客户端连接PolarDB-X 1.0执行DDL时是异步模式，即收到DDL请求便立即返回请求结果，而DDL任务继续在后台执行。您可以通过SHOW DDL查看DDL任务的状态，关于如何使用SHOW DDL，详情请参见任务管理语句。
  - 建议您在明确需要启用异步模式规避客户端与PolarDB-X 1.0连接意外中断的场景下将该参数设置为TRUE。否则，为了保证与MySQL执行DDL行为的兼容性，建议您保持该参数的默认值（FALSE）即可。
- 用法
  - 数据库级别：通过PolarDB-X 1.0控制台的参数设置进行调整，整个数据库范围内生效，详情请参见参数设置。
  - 会话级别：
    - 连接PolarDB-X 1.0后，执行 `set PURE_ASYNC_DDL_MODE=true` 或 `set PURE_ASYNC_DDL_MODE=1` 设置会话变量启用异步模式，在当前会话范围内生效。
    - 通过 `set PURE_ASYNC_DDL_MODE=false` 或 `set PURE_ASYNC_DDL_MODE=0` 恢复该会话的默认行为，使用同步模式。

- 语句级别：通过在DDL语句前增加HINT的方式 `/*+TDDL:cmd_extra(PURE_ASYNC_DDL_MODE=TRUE)*/` ，可以实现语句级别的控制，仅对该语句生效。

MAX_TABLE_PARTITIONS_PER_DB

- 说明
  - 该参数仅在 `ENABLE_ASYNC_DDL` 为TRUE时生效。
  - 创建拆分表时，若指定的单个物理库的分表数超过该参数的限制，DDL任务将报错停止执行。

    > **说明** 该参数取值范围为1~65535，默认值为128。

- 用法
  - 数据库级别：通过PolarDB-X 1.0控制台的参数设置进行调整，整个数据库范围内生效，详情请参见参数设置。
  - 语句级别：通过在DDL语句前增加HINT的方式 `/*+TDDL:cmd_extra(MAX_TABLE_PARTITIONS_PER_DB=400)*/` ，可以实现语句级别的控制，仅对该语句生效。

3.4. 注意事项与使用限制

新DDL执行引擎引入了任务管理功能，外部行为与之前版本相比有所变化。本文将介绍相关的注意事项与使用限制。

注意事项

- DDL正常执行成功时，无需关注DDL任务的状态，已成功完成执行的DDL任务会被自动清理。
- 建议DDL执行成功后，立即执行CHECK TABLE检查逻辑表的一致性。
- 通过DDL任务管理语句恢复、回滚或删除DDL任务后，建议执行CHECK TABLE检查逻辑表的一致性。
- DDL执行失败时，会返回导致失败的错误码和错误信息，您可以通过SHOW DDL查看处于PENDING状态的DDL任务失败的原因（即 `REMARK` 字段中记录的信息）。

  > **注意** 建议您在找到并解决导致DDL任务失败的因素后，再尝试执行DDL任务管理语句进行恢复、回滚或删除操作，否则DDL执行可能仍会失败。

- 若DDL执行失败，对应的DDL任务处于PENDING状态时，出于保护目的，目标表会处于不可访问状态（SHOW TABLES等操作无法显示，DML等操作可能会收到 `Unknown table` 或 `doesn't exist` 之类的报错），直到DDL任务通过恢复或回滚等方式使目标表达到一致性的状态后，该表才可以被正常访问。
- 当为CREATE TABLE指定 `IF NOT EXISTS` 或者为DROP TABLE指定 `IF EXISTS` 条件时，一些执行过程中的错误不会导致 DDL失败，但会记录在 `warning` 警告中，请注意DDL执行后是否返回 `warning` 数量的消息（例如 `1 warning` ），并用SHOW WARNINGS语句检查警告，避免遗漏重要的信息。
- 通过DMS等客户端工具执行DDL时，若无法评估DDL需要的执行时间且客户端工具本身带有超时中断连接（客户端与PolarDB-X 1.0之间的连接）的设置，为避免DDL由于超时中断连接而无法被继续执行，您可以启用 `PURE_ASYNC_DDL_MODE` 异步模式，执行DDL后立即返回，并继续通过 `SHOW DDL` 查看DDL任务状态。

使用限制

- 仅支持CREATE TABLE和RENAME TABLE两种DDL回滚操作。
- 不支持对处于PENDING状态的任务执行恢复（RECOVER DDL）和回滚（ROLLBACK DDL）的组合重复操作。

例如，先回滚失败任务，回滚失败后再对任务进行恢复操作。此类组合操作有可能造成逻辑表的不一致状态，如果遇到此类复杂场景，请提交工单。

- `REMOVE DDL` 要在非常确定安全性的前提下谨慎使用，若不确定则不应执行 `REMOVE DDL` ，误用可能造成DDL任务的中间状态暴露，出现逻辑表不一致的情况。如果因为误用 `REMOVE DDL` 产生问题或不确定的状态，请提交工单。
- 默认拆分表的单个物理库允许创建最多128个分表，您也可以通过如下参数调整上限。

```
mysql> create table test_mdb_mtb (c1 int not null auto_increment primary key, c2 varchar(
10), c3 date) dbpartition by hash(c1) tbpartition by hash(c1) tbpartitions 129;
ERROR 4647 (HY000): [f5bd90594800000][30.25.86.55:8527][JICHEN_LOCAL_APP]ERR-CODE: [TDDL-
4647][ERR_TABLE_PARTITIONS_EXCEED_LIMIT] The number of table partitions '129' exceeds the
upper limit '128'. Please specify less table partitions or adjust the value of the parame
ter MAX_TABLE_PARTITIONS_PER_DB.
mysql> /*+TDDL:cmd_extra(MAX_TABLE_PARTITIONS_PER_DB=400)*/create table test_mdb_mtb (c1
int not null auto_increment primary key, c2 varchar(10), c3 date) dbpartition by hash(c1)
tbpartition by hash(c1) tbpartitions 129;
Query OK, 0 rows affected (2.64 sec)
```

- DDL执行引擎内部的任务队列，最多允许堆积65535个PENDING状态的DDL任务，超过此上限则无法执行DDL，需要通过 `REMOVE DDL` 谨慎清理可删除的遗留任务，该数量限制无法通过参数调整上限。

3.5. 最佳实践

本文将介绍对PENDING任务进行适当处理的实践方法。

背景信息

新的DDL任务引擎启用时，当DDL执行失败或者被意外中断后，对应的DDL任务会处于PENDING待处理的状态。此时必须对该PENDING状态进行合适的任务处理，才能解除PENDING状态并恢复正常访问，否则后续的DDL将会被禁止执行并报错。

处理原则

- 您可以通过 `SHOW [FULL] DDL` 语句查看DDL任务的信息和失败原因（即 `REMARK` 字段记录的异常信息）。
- 您也可以参考如下常用的处理方式（仅供参考，请根据实际情况选择最适合的方式）：
  - 分析失败原因，修复或排除导致失败的因素（例如是由于数据问题导致的任务失败，则您可以通过去重等方式订正数据。如果是由于其它约束导致的失败，请确认是否能够去掉约束等）。修复完成后，使用 `RECOVER DDL` 恢复该PENDING任务。
  - 如果导致失败的因素无法解除，并且DDL因失败而不能真正执行，您可以使用REMOVE DDL删除任务（务必确认DDL没有真正执行才可删除，否则可能会造成状态不一致），删除后恢复可访问状态。
  - 如果您想直接删除DDL任务失败的表（比如表中无数据，可以直接删除重建），您可以使用 `REMOVE DDL` 删除任务，然后再执行 `DROP TABLE IF EXISTS` 删除表（务必确认表中无数据，或者数据可以丢弃，并且 `DROP TABLE` 一定要指定 `IF EXISTS` 语法，确保强制删除）。

示例

如下示例展示了对处于PENDING状态的DDL任务进行处理的过程。

1. 建表时没有指定主键，并且插入了带有重复值的数据行（ID=1有两行数据）：

```
mysql> create table test_pending (id int not null, age int) dbpartition by hash(id);
Query OK, 0 rows affected (0.33 sec)
mysql> insert into test_pending values(1,10),(1,20),(2,20),(3,30);
Query OK, 4 rows affected (0.10 sec)
mysql> select * from test_pending order by id;
+------+------+
| id   | age  |
+------+------+
|    1 |   10 |
|    1 |   20 |
|    2 |   20 |
|    3 |   30 |
+------+------+
4 rows in set (0.10 sec)
```

2. 之后想为上述ID加上主键约束，但由于表中已有数据违反了唯一性约束，因此DDL执行失败：

```
mysql> alter table test_pending add primary key (id);
ERROR 4636 (HY000): [f5be83373466000][10.81.69.55:3306][ddltest]ERR-CODE: [TDDL-4636][E
RR_DDL_JOB_ERROR] Not all physical operations have been done successfully: expected 9,
but done 8. Caused by: 1062:DDLTEST_15620564022300YMK_7WW7_0001:Duplicate entry '1' for
key 'PRIMARY' on `test_pending`;.
```

3. 通过 `SHOW FULL DDL` 语句查看任务状态和失败原因，发现其中一个物理表中的数据有重复值导致了物理DDL执行失败：

```
mysql> show full ddl\G
*************************** 1. row ***************************
          JOB_ID: 1106733441212637184
   PARENT_JOB_ID: 0
          SERVER: 1:102:10.81.69.55
   OBJECT_SCHEMA: ddltest
     OBJECT_NAME: test_pending
 NEW_OBJECT_NAME:
        JOB_TYPE: ALTER_TABLE
           PHASE: EXECUTE
           STATE: PENDING
        PROGRESS: 77%
      START_TIME: 2019-09-06 17:17:55.002
        END_TIME: 2019-09-06 17:17:55.273
ELAPSED_TIME(MS): 271
        DDL_STMT: alter table test_pending add primary key (id)
          REMARK: ERR-CODE: [TDDL-4636][ERR_DDL_JOB_ERROR] Not all physical operations
have been done successfully: expected 9, but done 8. Caused by: 1062:DDLTEST_1562056402
230OYMK_7WW7_0001:Duplicate entry '1' for key 'PRIMARY' on `test_pending`;.
```

REMARK字段中的详细信息解释如下：

- `Not all physical operations have been done successfully: expected 9, but done 8.`：该逻辑表的DDL涉及到9个物理DDL的执行，完成了8个，有1个失败了，这个失败的物理DDL导致整个逻辑DDL失败，任务被置于 PENDING状态。
- `Caused by: 1062:DDLTEST_1562056402 230OYMK_7WW7_0001:Duplicate entry '1' for key 'PRIM`

`ARY' on 'test_pending';` ：失败的根源在于 `DDLTEST_1562056402 230OYMK_7WW7_0001` 物理库的 `test_pending` 物理表中有ID字段的重复数据1，导致无法添加主键约束。

4. 此时逻辑表处于不一致的状态：

```
mysql> check table test_pending;
+---------------------------------------+-------+----------+--------------------------
-------------------------------------------------------------------------------+
| TABLE                                 | OP    | MSG_TYPE | MSG_TEXT
|
+---------------------------------------+-------+----------+--------------------------
-------------------------------------------------------------------------------+
| ddltest_1562056402230oymk.test_pending | check | Error    | Table 'DDLTEST_1562056402
230OYMK_7WW7_0001.test_pending' find incorrect columns 'id', please recreate table |
+---------------------------------------+-------+----------+--------------------------
-------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

5. 执行其它DDL也会被禁止，返回错误信息：

```
mysql> drop table test_pending;
ERROR 4644 (HY000): [f5beae39d466000][10.81.69.55:3306][ddltest]ERR-CODE: [TDDL-4644][E
RR_PENDING_DDL_JOB_EXISTS] Another DDL job '1106733441212637184' with operation 'ALTER_
TABLE' is pending on ddltest.test_pending in ddltest. Please use SHOW DDL to check it,
and then recover or rollback it using RECOVER DDL or ROLLBACK DDL, or just remove it us
ing REMOVE DDL if you confirm that the pending job can be discarded.
```

6. 接下来，根据前面所述的常见的处理方式，您可以选择以下几种方式继续进行处理（分别展示它们的效果）：
   - 去重（删除重复的数据）后，恢复DDL任务，继续完成添加主键约束的操作。
     a. 删除重复数据（根据业务需要，仅保留一条数据），删除数据操作可以通过PolarDB-X 1.0执行，也可以根据报错信息，直接连接到PolarDB-X 1.0后端的RDS物理库中操作：

```
mysql> delete from test_pending where id=1 and age=20;
Query OK, 1 row affected (0.07 sec)
mysql> select * from test_pending order by id;
+------+------+
| id   | age  |
+------+------+
|    1 |   10 |
|    2 |   20 |
|    3 |   30 |
+------+------+
3 rows in set (0.02 sec)
```

b. 确认表中已经没有重复数据后，恢复之前PENDING的DDL任务的执行（恢复成功，完成的任务被自动清理，主键添加成功）：

```
mysql> recover ddl 1106733441212637184;
Query OK, 0 rows affected (1.28 sec)
mysql> show full ddl\G
Empty set (0.00 sec)
mysql> show create table test_pending\G
*************************** 1. row ***************************
       Table: test_pending
Create Table: CREATE TABLE `test_pending` (
  `id` int(11) NOT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `auto_shard_key_id` (`id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`)
1 row in set (0.02 sec)
mysql> check table test_pending;
+---------------------------------------+-------+----------+----------+
| TABLE                                 | OP    | MSG_TYPE | MSG_TEXT |
+---------------------------------------+-------+----------+----------+
| ddltest_1562056402230oymk.test_pending | check | status   | OK       |
+---------------------------------------+-------+----------+----------+
1 row in set (0.10 sec)
```

- 直接删除任务，然后删除表（测试数据可以丢弃），后续再根据需要重新创建该表。

```
mysql> remove ddl 1106733441212637184;
Query OK, 1 row affected (0.02 sec)
mysql> drop table if exists test_pending;
Query OK, 0 rows affected (0.44 sec)
mysql> show tables like 'test_pending';
Empty set (0.01 sec)
```

4.DDL

4.1. 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。

注意事项

- PolarDB-X 1.0目前不支持使用DDL语句直接建库，请登录控制台进行创建。关于如何创建数据库，详情请参见创建数据库。
- PolarDB-X 1.0支持全局二级索引 (Global Secondary Index，GSI)，要求MySQL版本为5.7或以上，并且PolarDB-X 1.0实例版本为5.4.1或以上。基本原理请参见全局二级索引。

```
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]]
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}
```

说明 PolarDB-X 1.0DDL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，请参见拆分函数概述 。

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 1.0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。

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)
```

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_PARTI
TION_COUNT | TB_PARTITION_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 1.0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) tbpa
rtition 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_COU
NT | 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_COU
NT | 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_COU
NT | 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_COU
NT | 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_COU
NT | 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)
```

使用主键作为拆分键

当拆分算法不指定任何拆分字段时，系统默认使用主键作为拆分字段。以下示例将介绍如何使用主键当分库和分表键。

- 使用主键当分库键

```
CREATE TABLE prmkey_tbl(
 id bigint not null auto_increment,
 name varchar(30),
 primary key(id)
) dbpartition by hash();
```

- 使用主键当分库分表键

```
CREATE TABLE prmkey_multi_tbl(
 id bigint not null auto_increment,
 name varchar(30),
 primary key(id)
) dbpartition by hash() tbpartition by hash() tbpartitions 3;
```

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调优方法。

```
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) tbpart
itions 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，请参见使用全局二级索引时的注意事项 ，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 | SUB_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_PARTITI
ON_POLICY | TB_PARTITION_COUNT | STATUS |
+--------+---------+------------+------------+-------------+----------------+------------+-
----------------+--------------------+-------------------+-----------------+-----------
----------+--------------------+---------+
| d7     | t_order | 1          | g_i_seller | seller_id   | id, order_id   | NULL       |
seller_id        | HASH               | 8                 |                 | 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，请参见使用全局二级索引时的注意事项，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    | COL
LATION | CARDINALITY | 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       | NUL
L      |           0 |     NULL | NULL   | YES  | GLOBAL     | INDEX    |               |
| t_order      |          1 | g_i_buyer              |            2 | id             | NUL
L      |           0 |     NULL | NULL   |      | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer              |            3 | order_id       | NUL
L      |           0 |     NULL | NULL   | YES  | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer              |            4 | seller_id      | NUL
L      |           0 |     NULL | NULL   | YES  | GLOBAL     | COVERING |               |
| t_order      |          1 | g_i_buyer              |            5 | order_snapshot | NUL
L      |           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_K
EY | 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(`b
uyer_id`) tbpartitions 3 |
+-----------+-------------------------------------------------------------------------------------
--------------------------+
```

4.2. DROP TABLE

您可以使用DROP TABLE语法删除指定表。

语法

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

> **说明** 删除PolarDB-X 1.0数据表与删除原生MySQL数据库表的语法没有区别，系统会自动处理相关物理表的删除操作。更多详情，请参见DROP TABLE Statement。

注意事项

- 若需要删除的表已设置了全局二级索引，执行DROP TABLE命令时，索引表和主表会同时被删除。
- 不支持直接通过DROP TABLE语法删除索引表。您可以使用ALTER TABLE DROP INDEX语句进行删除。详细语法，请参见ALTER TABLE。

示例

您可以通过如下语句删除 `user_log` 表：

```
DROP TABLE user_log;
```

4.3. 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。
- 在包含全局二级索引的表中使用ALTER语法，要求MySQL为5.7及以上版本，PolarDB-X 1.0为5.4.1及以上版本。

变更普通表

> **说明** PolarDB-X 1.0中，修改普通表的表结构语法与修改原生MySQL数据库表结构的语法没有区别。更多详情，请参见ALTER TABLE Statement。

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);
  ```

变更包含全局二级索引的表

列变更

包含全局二级索引的表在进行列变更操作时，使用的语法与普通表的列变更的语法一致，但存在一些限制。详细注意事项请参见ALTER TABLE时的注意事项。

- 下表汇总了使用ALTER TABLE语句变更列的支持情况。

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

- 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}	禁止
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。
ALTER TABLE RENAME INDEX	禁止

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，详情请参见使用全局二级索引时的注意事项。

全局二级索引定义子句详细说明请参见CREATE TABLE。

示例

下面以建立全局唯一索引为例，介绍在建表后如何创建GSI。

- 创建全局二级索引

```
# 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_sna
pshot`) 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。

```
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 |          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
| INDEX_TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT | STATUS |
+---------------------+---------+------------+-----------+-------------+-----------------
------------+------------+-----------------+---------------------+-------------------+
-----------------+---------------------+--------------------+---------+
| ZZY3_DRDS_LOCAL_APP | t_order | 0          | g_i_buyer | buyer_id    | id, order_id, or
der_snapshot | NULL       | 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,
  `buyer_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`) |
+-----------+-----------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------+
```

- 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。

```
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。

4.4. 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。

4.5. 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 1.0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。

4.6. CREATE INDEX

本文介绍了如何创建局部索引和全局二级索引。

注意事项

在包含全局二级索引的表中使用ALTER语法，要求MySQL为5.7及以上版本，PolarDB-X 1.0为5.4.1及以上版本。

局部索引

关于局部索引，详情清参见CREATE INDEX Statement。

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，详情请参见使用全局二级索引时的注意事项。

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。

- 创建全局二级索引

```
# 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_sna
pshot`) 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。

```
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 |          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
| INDEX_TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION
_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT | STATUS |
+---------------------+---------+------------+-----------+-------------+-----------------
------------+------------+-----------------+---------------------+-------------------+
-----------------+---------------------+--------------------+---------+
| ZZY3_DRDS_LOCAL_APP | t_order | 0          | g_i_buyer | buyer_id    | id, order_id, or
der_snapshot | NULL       | 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,
  `buyer_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`) |
+-----------+-----------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------+
```

4.7. 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

```
# index_name为需要删除的全局二级索引名。
DROP INDEX index_name ON tbl_name
```

4.8. CREATE VIEW

本文将介绍如何使用CREATE VIEW语句为PolarDB-X 1.0创建视图。

前提条件

PolarDB-X 1.0实例版本需为5.4.5或以上，关于如何查看实例版本，请参见实例版本概览。

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;
```

4.9. DROP VIEW

本文将介绍如何使用DROP VIEW语句删除PolarDB-X 1.0的视图。

前提条件

PolarDB-X 1.0实例版本需为5.4.5或以上，关于如何查看实例版本，请参见实例版本概览。

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;
```

4.10. DDL常见问题

本文汇总了PolarDB-X 1.0上常见的DDL执行问题。

建表的时候执行出错怎么办？

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 1.0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的命令来查看每个分库执行失败的原因。
2. 您可以通过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. 使用 `CHECK TABLE tablename` 指令来查看逻辑表是否创建成功。

   比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. 您可以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怎么办？

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。详情请参见如何处理DDL异常 。

5.DML

5.1. 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 1.0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 1.0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 1.0会将该Hint下推至底层MySQL。
- 暂不支持STRAIGHT_JOIN和NATURAL JOIN。

UNION

PolarDB-X 1.0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 1.0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

5.2. 子查询

本文介绍PolarDB-X 1.0支持的子查询类别及在PolarDB-X 1.0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 1.0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 1.0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 1.0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。

5.3. 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);
```

分布式事务限制

> **说明** 如果您的表是分表，但是事务执行过程中没有跨库（如INSERT或UPDATE带拆分键），则仍视为单库事务。关于分布式事务的更多说明，请参见分布式事务。

开启分布式事务时，不支持如下INSERT命令：

- 表没有定义主键，例如：

```
CREATE TABLE tb(id INT, name VARCHAR(10));
INSERT INTO tb VALUES(1, 'a');
```

- 表没有拆分，主键自增但没有使用Sequence。例如：

```
CREATE TABLE tb(id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(10));
INSERT INTO tb(name) VALUES('a');
```

  您可以指定该主键使用Sequence避免这条限制，例如：

```
CREATE TABLE tb(id INT PRIMARY KEY AUTO_INCREMENT BY GROUP, name VARCHAR(10));
INSERT INTO tb(name) VALUES('a');
```

相关文献

MySQL INSERT 语法。

5.4. 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);
```

- 嵌套NEXTVAL的语法，例如：

```
REPLACE INTO tb(id) VALUES(SEQ1.NEXTVAL + 1);
```

- 包含列名的语法，例如：

```
REPLACE INTO tb(id1, id2) VALUES(1, id1 + 1);
```

分布式事务限制

> **说明** 如果您的表是分表，但是事务执行过程中没有跨库（如INSERT或UPDATE带拆分键），则仍视为单库事务。关于分布式事务的更多说明，请参见分布式事务。

开启分布式事务时，不支持如下REPLACE命令：

- 表没有定义主键，例如：

```
CREATE TABLE tb(id INT, name VARCHAR(10));
REPLACE INTO tb VALUES(1, 'a');
```

- 表没有拆分，主键自增但没有使用Sequence。例如：

```
CREATE TABLE tb(id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(10));
REPLACE INTO tb(name) VALUES('a');
```

  您可以指定该主键使用Sequence，避免这条限制，例如：

```
CREATE TABLE tb(id INT PRIMARY KEY AUTO_INCREMENT BY GROUP, name VARCHAR(10));
REPLACE INTO tb(name) VALUES('a');
```

相关文献

MySQL REPLACE语法。

5.5. 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 1.0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 1.0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;
```

- 默认禁止更新行数超过10000的不可下推的UPDATE，需要通过HINT打开限制，例如：

```
UPDATE t1 SET t1.name = "abc" ORDER BY name LIMIT 10001;
UPDATE t1, t2 SET t1.name = t2.name WHERE t1.id = t2.name LIMIT 10001;
```

> 说明 t1和t2的拆分键为ID。

- 默认禁止执行全表更新操作，需要通过HINT打开限制，详情请参见高危类SQL自动保护。

相关文献

MySQL官方文档，请参见UPDATE。

5.6. 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 1.0的修饰符操作产生影响。

语法限制

与原生MySQL的DELETE语法相比，PolarDB-X 1.0的DELETE语法存在以下限制。

默认禁止删除行数超过10000的不可下推的DELETE，需要通过HINT打开限制，例如：

```
DELETE FROM t1 ORDER BY name LIMIT 10001;
DELETE t1, t2 FROM t1 INNER JOIN t2 INNER JOIN t3 WHERE t1.id=t2.id AND t2.id=t3.name LIMIT
10001;
DELETE FROM t1, t2 USING t1 INNER JOIN t2 INNER JOIN t3 WHERE t1.id=t2.id AND t2.id=t3.name
LIMIT 10001;
```

> **说明** t1、t2和t3的拆分键均为ID。

默认禁止执行全表删除操作，需要通过HINT打开限制，详情请参见高危类SQL自动保护。

5.7. 全局二级索引对DML的限制

本文将介绍PolarDB-X 1.0上全局二级索引对DML的限制。

前提条件

MySQL版本需为5.7或以上，且PolarDB-X 1.0实例版本需为5.4.1或以上，关于如何查看实例版本，请参见实例版本概览。

示例

本文将以下表为例介绍全局二级索引对DML的不同限制：

```
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 h
ash(`seller_id`),
  GLOBAL UNIQUE INDEX `g_i_buyer` (`buyer_id`) COVERING (order_snapshot) dbpartition by has
h(`buyer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
```

写索引失败后，不允许继续执行其他语句或提交事务。

```
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', 'sel
ler_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', 'sel
ler_2');
# 失败后不允许提交事务
COMMIT;
```

6.SHOW

6.1. SHOW HELP

本文介绍了如何使用SHOW HELP命令。

SHOW HELP用于展示PolarDB-X 1.0所有辅助SQL指令及其说明。

```
mysql> show help;
+----------------------------------------+-----------------------------------------------
---------+-----------------------------------------------+
| STATEMENT                              | DESCRIPTION
| EXAMPLE                                   |
+----------------------------------------+-----------------------------------------------
---------+-----------------------------------------------+
| show rule                              | Report all table rule
|                                           |
| show rule from TABLE                   | Report table rule
| show rule from user                       |
| show full rule from TABLE              | Report table full rule
| show full rule from user                  |
| show topology from TABLE               | Report table physical topology
| show topology from user                   |
| show partitions from TABLE             | Report table dbPartition or tbPartition columns
| show partitions 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 profil
ing data | trace select count(*) from user; show trace |
| show trace                             | Report sql execute profiling info
|                                           |
| explain SQL                            | Report sql plan info
| explain select count(*) from user         |
| explain detail SQL                     | Report sql detail plan info
| explain detail select count(*) from user  |
| explain execute SQL                    | Report sql on physical db plan info
| explain 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 sequence test                        |
+----------------------------------------+-----------------------------------------------
---------+-----------------------------------------------+
20 rows in set (0.00 sec)
```

6.2. 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]

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_C
OUNT | TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+--------------+-----------+-----------------+---------------------+---------------
-----+-----------------+--------------------+-------------------+
|    0 | dept_manager |         0 |                 | NULL                | 1
|                  | NULL               | 1                 |
|    1 | emp          |         0 | emp_no          | hash                | 8
| id               | hash               | 2                 |
|    2 | example      |         0 | shard_key       | hash                | 8
|                  | NULL               | 1                 |
+------+--------------+-----------+-----------------+---------------------+---------------
-----+-----------------+--------------------+-------------------+
3 rows in set (0.01 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 | dept_manager |         0 | NULL       |                    0 | SEQ_TEST_148776778
0814RGKKSEQ_TEST_WNJG_0000_RDS   | NULL                                                       | de
pt_manager    | NULL                                | NULL          | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0000_RDS |
|    1 | emp          |         0 | NULL       |                    1 | SEQ_TEST_148776778
0814RGKKSEQ_TEST_WNJG_{0000}_RDS | ((#emp_no,1,8#).longValue().abs() % 8)             | em
p_{0}         | ((#id,1,2#).longValue().abs() % 2) | emp_no      id | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0000_RDS |
|    2 | example      |         0 | NULL       |                    1 | SEQ_TEST_148776778
0814RGKKSEQ_TEST_WNJG_{0000}_RDS | ((#shard_key,1,8#).longValue().abs() % 8).intdiv(1) | ex
ample         | NULL                                | shard_key     | SEQ_TEST_148776778081
4RGKKSEQ_TEST_WNJG_0000_RDS |
+------+--------------+-----------+------------+----------------------+------------------
--------------------------------+------------------------------------------------------------+---
-------------+-------------------------------------+---------------+--------------------
---------------------------+
3 rows in set (0.01 sec)
```

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      |
+------+-------------------------------------------------+------------+
16 rows in set (0.01 sec)
```

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                                           | GROU
P                                        | URL
| USER       | TYPE  | INIT | 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_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0000 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0000_iiab |
10          | 10           |
|    1 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab_2 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0001 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab |
10          | 10           |
|    2 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab_3 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0002 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab |
10          | 10           |
|    3 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab_4 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0003 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab |
10          | 10           |
|    4 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab_5 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0004 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab |
```

```
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab |
10          | 10           |
|    5 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab_6 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0005 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab |
10          | 10           |
|    6 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab_7 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0006 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab |
10          | 10           |
|    7 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab_8 | SEQ_
TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS | jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.
aliyuncs.com:3306/seq_test_wnjg_0007 | jnkinsea0 | mysql | 0    | 24   | 72   | 15
| 5000     | 0            | 1             | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab |
10          | 10           |
+------+----------------------------+-----------------------------------------------+-----
-------------------------------------------+-----------------------------------------------
-----------------------------------+-----------+-------+------+------+------+------------
--+----------+--------------+---------------+-----------------------------------------------
+-------------+--------------+
8 rows in set (0.01 sec)
```

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 1.0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_READ_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。

6.3. 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 1.0的慢SQL。

- `SHOW SLOW`：查看自PolarDB-X 1.0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 1.0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 1.0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 1.0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 1.0发送到RDS的慢SQL。

- `SHOW PHYSICAL_SLOW`：查看自PolarDB-X 1.0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 1.0系统中，当实例重启或者执行时会丢失）；
- `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 1.0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 1.0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 1.0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。

6.4. 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 FULL DB STATUS [LIKE {tablename}]
- SHOW TABLE STATUS [LIKE 'pattern' | WHERE expr]

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 1.0 `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_CONNECTIONS | RT(MS) | RDS_RT(MS) | NET_IN(KB/S) | NET_OUT(KB/S) | THREAD_RUNNING
|
+------+---------+----------+------------------+-----------------+----------------------
-+--------------------+--------+------------+--------------+--------------+--------------
-+
| 1.77 |    1.68 |     0.03 |              0.03 |            0.02 |                   0.00
|                  7 | 157.13 |      51.14 |       134.49 |          1.48 |              1
|
+------+---------+----------+------------------+-----------------+----------------------
-+--------------------+--------+------------+--------------+--------------+--------------
-+
1 row in set (0.01 sec)
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_R
T(MS) | NET_IN(KB/S) | NET_OUT(KB/S) | THREAD_RUNNING | HINT_USED_PER_SECOND | HINT_USED_CO
UNT | AGGREGATE_QUERY_PER_SECOND | AGGREGATE_QUERY_COUNT | TEMP_TABLE_CREATE_PER_SECOND | T
EMP_TABLE_CREATE_COUNT | MULTI_DB_JOIN_PER_SECOND | MULTI_DB_JOIN_COUNT | 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 |          1.21 |              1 |                 0.00 |              54 |
0.00 |                  663 |                          0.00 |                      512 |
0.00 |                516 | 0.09% |   6.96% |       76446 |   21326906 |
+------+---------+----------+------------------+-----------------+----------------------+
-----------------------+--------------------+-------------------------------+--------+-----
-------+--------------+---------------+---------------+----------------------+------------
-----+------------------------------+-----------------------+------------------------------+-
-----------------------+--------------------------+---------------------+-------+---------
+-------------+------------+
1 row in set (0.01 sec)
```

重要列说明：

- **QPS**：应用到PolarDB-X 1.0的QPS，通常称为逻辑QPS；

- RDS_QPS：PolarDB-X 1.0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 1.0的响应时间，通常称为逻辑RT（响应时间）；
- RDS_RT(MS)：PolarDB-X 1.0到RDS/MySQL的响应时间，通常称为物理RT；
- NET_IN(KB/S)：PolarDB-X 1.0收到的网络流量；
- NET_OUT(KB/S)：PolarDB-X 1.0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_RUNNING |
+------+--------------------------+-------------------+------------------+------------+-
-------+----------------+
|    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 1.0DB，此处显示的是PolarDB-X 1.0内部标记，与PolarDB-X 1.0DB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 1.0DB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 1.0DB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 FULL DB STATUS [LIKE {tablename}]

用于查看物理库表容量和性能信息，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 full db status like hash_tb;
+------+--------------------------+-------------------+------------------+-------------
--+------------+--------+----------------+
| ID   | NAME                     | CONNECTION_STRING  | PHYSICAL_DB       | PHYSICAL_TABL
E | SIZE_IN_MB | RATIO  | THREAD_RUNNING |
+------+--------------------------+-------------------+------------------+-------------
```

```
--+------------+--------+----------------+
|    1 | drds_db_1516187088365daui | 100.100.64.1:59077 | TOTAL             |
|     19.875 | 100%   | 3              |
|    2 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | TOTAL
|    3.03125 | 15.25% |                |
|    3 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | hash_tb_00
|   1.515625 | 7.63%  |                |
|    4 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | hash_tb_01
|   1.515625 | 7.63%  |                |
|    5 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | TOTAL
|        2.0 | 10.06% |                |
|    6 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | hash_tb_02
|   1.515625 | 7.63%  |                |
|    7 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | hash_tb_03
|   0.484375 | 2.44%  |                |
|    8 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | TOTAL
|    3.03125 | 15.25% |                |
|    9 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | hash_tb_04
|   1.515625 | 7.63%  |                |
|   10 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | hash_tb_05
|   1.515625 | 7.63%  |                |
|   11 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | TOTAL
|   1.953125 | 9.83%  |                |
|   12 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | hash_tb_06
|   1.515625 | 7.63%  |                |
|   13 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | hash_tb_07
|     0.4375 | 2.2%   |                |
|   14 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | TOTAL
|    3.03125 | 15.25% |                |
|   15 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | hash_tb_08
|   1.515625 | 7.63%  |                |
|   16 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | hash_tb_09
|   1.515625 | 7.63%  |                |
|   17 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | TOTAL
|   1.921875 | 9.67%  |                |
|   18 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | hash_tb_11
|   1.515625 | 7.63%  |                |
|   19 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | hash_tb_10
|    0.40625 | 2.04%  |                |
|   20 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | TOTAL
|    3.03125 | 15.25% |                |
|   21 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | hash_tb_12
|   1.515625 | 7.63%  |                |
|   22 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | hash_tb_13
|   1.515625 | 7.63%  |                |
|   23 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | TOTAL
|      1.875 | 9.43%  |                |
|   24 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | hash_tb_14
|   1.515625 | 7.63%  |                |
|   25 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | hash_tb_15
|   0.359375 | 1.81%  |                |
+------+----------------------------+--------------------+-------------------+--------------
--+------------+--------+----------------+
```

重要列说明：

- **NAME**：代表一个PolarDB-X 1.0DB。此处显示的是PolarDB-X 1.0内部标记，与PolarDB-X 1.0DB名称不同；
- **CONNECTION_STRING**：分库的连接信息；
- **PHYSICAL_DB**：分库名称，`TOTAL`行代表经过LIKE关键字筛选后得到的分库容量的总和。如果没有LIKE，则为全部分库容量的总和；
- **PHYSICAL_TABLE**：分表名称，`TOTAL`行代表经过LIKE关键字筛选后得到的分表容量的总和。如果没有LIKE，则为全部分表容量的总和；
- **SIZE_IN_MB**：分表中数据占用的空间，单位为 MB；
- **RATIO**：单个分表数据量在当前筛选出的分表总数据量中的占比；
- **THREAD_RUNNING**：物理数据库实例当前正在执行的线程情况，含义与MySQL`SHOW GLOBAL STATUS`指令返回值的含义相同。详情请参考 MySQL 文档。

SHOW TABLE STATUS [LIKE 'pattern' | WHERE expr]

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 like 'multi_db_multi_tbl';
+--------------------+--------+---------+------------+------+----------------+-------------
+-----------------+--------------+-----------+----------------+----------------------+----
---------+------------+-----------------+----------+-----------------+---------+
| NAME               | ENGINE | VERSION | ROW_FORMAT | ROWS | AVG_ROW_LENGTH | DATA_LENGTH
| MAX_DATA_LENGTH | INDEX_LENGTH | DATA_FREE | AUTO_INCREMENT | CREATE_TIME          | UPD
ATE_TIME | CHECK_TIME | COLLATION       | CHECKSUM | CREATE_OPTIONS | COMMENT |
+--------------------+--------+---------+------------+------+----------------+-------------
+-----------------+--------------+-----------+----------------+----------------------+----
---------+------------+-----------------+----------+-----------------+---------+
| multi_db_multi_tbl | InnoDB |      10 | Compact    |    2 |          16384 |       16384
|               0 |        16384 |         0 |         100000 | 2017-03-27 17:43:57.0 | NUL
L        | NULL       | utf8_general_ci | NULL     |                |         |
+--------------------+--------+---------+------------+------+----------------+-------------
+-----------------+--------------+-----------+----------------+----------------------+----
---------+------------+-----------------+----------+-----------------+---------+
1 row in set (0.03 sec)
```

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。

和PolarDB-X 1.0的SCAN HINT结合，还可以查看每个物理分表的数据量。具体请参考Hint。

```
mysql> /!TDDL:SCAN='multi_db_multi_tbl'*/show table status like 'multi_db_multi_tbl%';
+----------------------+--------+---------+------------+------+----------------+-----------
--+-----------------+--------------+-----------+----------------+---------------------+----
---------+------------+-----------------+----------+----------------+---------+------------
--+
| Name                 | Engine | Version | Row_format | Rows | Avg_row_length | Data_lengt
h | Max_data_length | Index_length | Data_free | Auto_increment | Create_time         | Upd
ate_time | Check_time | Collation       | Checksum | Create_options | Comment | Block_forma
t |
+----------------------+--------+---------+------------+------+----------------+-----------
--+-----------------+--------------+-----------+----------------+---------------------+----
---------+------------+-----------------+----------+----------------+---------+------------
--+
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    1 |          16384 |       1638
4 |               0 |        16384 |         0 |              2 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
L        | NULL       | utf8_general_ci |     NULL |                |         | Original
|
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       1638
4 |               0 |        16384 |         0 |              1 | 2017-03-27 17:43:57 | NUL
```

```
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_1 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_0 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_1 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_0 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_1 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_0 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_1 | InnoDB |       10 | Compact    |    0 |                0 |        1638
4 |                0 |          16384 |           0 |              1 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
| multi_db_multi_tbl_0 | InnoDB |       10 | Compact    |    1 |            16384 |        1638
4 |                0 |          16384 |           0 |              3 | 2017-03-27 17:43:57 | NUL
L         | NULL          | utf8_general_ci |      NULL |                 |          | Original
|
+----------------------+--------+---------+------------+------+----------------+-----------
--+-----------------+--------------+-----------+----------------+---------------------+----
---------+------------+-----------------+----------+----------------+---------+------------
--+
16 rows in set (0.04 sec)
```

6.5. 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 1.0中的连接与正在执行的SQL等信息：

- 语法

```
SHOW PROCESSLIST
```

- 示例

```
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。
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

```
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 1.0返回的物理连接的ID列为一个字符串，并非一个数字。

6.6. SHOW GLOBAL INDEX

PolarDB-X 1.0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_K
EY | 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_in
t_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_da
tetime_6, c_timestamp_1, c_timestamp_3, c_time, c_time_1, c_time_3, c_time_6, c_year, c_yea
r_4, c_char, c_varchar, c_binary, c_varbinary, c_blob_tiny, c_blob_medium, c_text_tiny, c_t
ext, 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_geometrycollection, 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_bigi
nt_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_tim
estamp_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_tex
t, c_text_medium, c_text_long, c_enum, c_set, c_json, c_point, c_linestring, c_polygon, c_m
ultipoint, c_multilinestring, c_multipolygon, c_geometrycollection, 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, 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_timest
amp_1, c_timestamp_3, c_time, c_time_3, c_time_6, c_year, c_year_4, c_char, c_varchar, c_bi
nary, c_varbinary, c_blob_tiny, c_blob_medium, c_blob_long, c_text_tiny, c_text, c_text_med
ium, c_text_long, c_enum, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_
multilinestring, c_multipolygon, c_geometrycollection, 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

列名	说明
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
STATUS	索引的当前状态，取值范围如下： • CREATING • DELETE_ONLY • WRITE_ONLY • WRITE_REORG • PUBLIC • ABSENT

6.7. SHOW INDEX

您可以使用SHOW INDEX语句查看PolarDB-X 1.0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 | CARDINALIT
Y | 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 |               |
+--------------+------------+-----------+--------------+----------------+-----------+------
-------+----------+--------+------+------------+----------+---------------+
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
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。

列名	说明
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

6.8. SHOW METADATA LOCK

本文将介绍如何在PolarDB-X 1.0上使用SHOW METADATA LOCK语句查询持有锁的事务。

背景信息

PolarDB-X 1.0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 1.0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_WRIT
E | MDL_TRANSACTION |        1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54788 | insert into `full_gs
i_ddl` (id) VALUE (null); |
| 5       |      0 | f88cf71cbc00000 | XXXX_DRDS_LOCAL_APP | full_gsi_ddl | MDL_SHARED_WRIT
E | MDL_TRANSACTION |        1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54789 | insert into `full_gs
i_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

7.DAL

7.1. 账号和权限系统

PolarDB-X 1.0账号和权限系统的用法与MySQL一致，支持 `GRANT` 、 `REVOKE` 、 `SHOW GRANTS` 、 `CREATE USER` 、 `DROP USER` 、 `SET PASSWORD` 等语句。

账号

账号说明

用户名和主机名的组合 `username@'host'` 可以确定一个账号。用户名一样但是主机名不一样则代表不同的账号。例如 `lily@30.9.73.96` 和 `lily@30.9.73.100` 是两个完全不同的账号，这两个账号的密码和权限都可能不一样。

在PolarDB-X 1.0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。

- 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` 。

例如有两个数据库 `dreamdb` 、 `andordb` ，根据上面的规则可知， `dreamdb` 下面包含管理员账号 `dreamdb` ，只读账号 `dreamdb_RO` ； `andordb` 下面包含管理员账号 `andordb` ，只读账号 `andordb_RO` 。

> **说明** PolarDB-X 1.0里通过CREATE USER创建出来的账号只存在于PolarDB-X 1.0，跟RDS没有任何关系，也不会同步到后端的RDS中去。

账号规则

- 管理员账号具有所有权限；
- 只有管理员账号具有创建账号和授权功能；其它账号只能由管理员账号创建，其权限只能由管理员账号授予；
- 管理员账号是跟数据库绑定的，没有其它数据库的权限，连接的时候只能连接自己对应的那个数据库，不能授予其它数据库的权限给某个账号。例如easydb这个管理员账号只能连接easydb数据库，只能授予easydb数据库权限或者easydb数据库中表的权限给某个账号；
- 只读账号只具有SELECT权限。

命名规则

- 区分大小写；
- 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。

密码规则

- 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（@#$%^&+=）。

HOST匹配规则

- 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@'30.9.%.%'，david@'%';
- 假设系统中有两个用户都符合当前登录的用户，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_.xxx'` 和 `david@'30.9.1%.234'` ，在主机 `30.9.127.xxx` 上面登录david，则使用的是 `david@'30.9.12_.xxx'` 这个用户。
- 开启VPC时，主机的IP地址会发生变化。

> 注意 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。

权限

权限级别支持情况

- 数据库层级（支持）
- 表层级（支持）
- 全局层级（暂不支持）
- 列层级（暂不支持）
- 子程序层级（暂不支持）

权限项

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。

权限规则

- 权限是绑定到账号（username@'host'），不是绑定到用户名（username）;
- 授权的时候会判断表是否存在，不存在则报错;
- 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。
- 高级别权限的授予会覆盖低级别权限，移除高级别权限的同时也会移除低级别权限;
- 不支持USAGE授权。

给用户授予多个库权限

5.3.6及以上版本的PolarDB-X 1.0，支持给单个用户授予多个库权限。授权方式如下：

- 您可以在阿里云PolarDB-X 1.0控制台 “账号管理” 页面创建账户和授权，推荐您使用此方法。
- 也可以使用SQL语句CREATE USER和GRANT创建账户和授权。

> **说明** 如果使用SQL语句，需注意：
>
> i. 只有管理员账户可以创建用户和授权。
>
> ii. 管理员只能给用户授予自己库的权限。如果A库管理员创建了一个账户new_user@'%'，要使new_user同时获得访问A库和B库的权限，需要A库管理员和B库管理员分别为其授权。

拥有多个库权限的用户使用

5.3.6及以上版本可以给用户授予多个库权限，如果new_user@'%'拥有了A和B库的SELECT和INSERT权限。使用时有以下限制：

- 如果用户当前登录为A库，要查询B库，需 `use B; SELECT * FROM table_in_B;` ，暂不支持跨库查询，如 `SELECT * FROM B.table_in_B;` 。
- 如果用户当前登录为A库，要写入B库，需 `use B; INSERT INTO table_in_B VALUES('value');` ，暂不支持跨库插入，如 `INSERT INTO B.table_in_B VALUES('value');` 。
- 其他SQL类型同理。

相关命令

创建账号（CREATE USER）

- 语法

```
CREATE USER user_specification [, user_specification] ...
user_specification: user [ auth_option ]
auth_option: IDENTIFIED BY 'auth#string'
```

- 示例
  - 创建一个名为lily，只能从30.9.73.96登录的账号，密码为123456。

    ```
    CREATE USER lily@30.9.73.96 IDENTIFIED BY '123456';
    ```

  - 创建一个名为david，可以从任意主机登录的账号，密码为空。

    ```
    CREATE USER david@'%';
    ```

删除账号（DROP USER）

- 语法

```
DROP USER user [, user] ...
```

- 示例

  移除账号lily@30.9.73.96：

```
DROP USER lily@30.9.73.96;
```

修改账号密码（SET PASSWORD）

- 语法

```
SET PASSWORD FOR user = password_option
password_option: {
    PASSWORD('auth_string')
}
```

- 示例

  修改账号lily@30.9.73.96的密码为123456。

```
SET PASSWORD FOR lily@30.9.73.96 = PASSWORD('123456')
```

给账号授权（GRANT）

- 语法

```
GRANT
    priv_type[, priv_type] ...
    ON priv_level
    TO user_specification [, user_specification] ...
    [WITH GRANT OPTION]
priv_level: {
  | db_name.*
  | db_name.tbl_name
  | tbl_name
}
user_specification:
    user [ auth_option ]
auth_option: {
    IDENTIFIED BY 'auth#string'
}
```

> **说明** GRANT语句里面的账号如果不存在，同时又没有提供IDENTIFIED BY信息，则报账号不存在异常；如果提供了IDENTIFIED BY信息，则会创建该账号同时授权。

- 示例
  - 在数据库easydb下面，创建一个用户名为david，可以在任意主机登录，具有easydb数据库所有权限的账号。

```
#方法1：先创建账号再授权
CREATE USER david@'%' IDENTIFIED BY 'your#password';
GRANT ALL PRIVILEGES ON easydb.* to david@'%';
#方法2：一条语句完成创建账号和授权两个操作
GRANT ALL PRIVILEGES ON easydb.* to david@'%' IDENTIFIED BY 'your#password';
```

  - 在数据库easydb下面，创建一个用户名为hanson，可以在任意主机登录，具有easydb.employees表所有权限的账号。

```
GRANT ALL PRIVILEGES ON easydb.employees to hanson@'%'
IDENTIFIED BY 'your#password';
```

- 在数据库easydb下面，创建一个用户名为hanson，只能在192.168.3.10登录，具有easydb.emp表的INSERT 和 SELECT权限的账号。

```
GRANT INSERT,SELECT ON easydb.emp to hanson@'192.168.3.10'
IDENTIFIED BY 'your#password';
```

- 在数据库easydb下面创建一个只读账号actro，可以在任意主机登录。

```
GRANT SELECT ON easydb.* to actro@'%' IDENTIFIED BY 'your#password';
```

回收权限（REVOKE）

- 语法
  - 删除账号在某个权限级别下的权限项，具体权限级别由priv_level指定。

```
REVOKE
priv_type
[, priv_type] ...
ON priv_level
```

  - 删除账号在系统内（数据库级别和表级别的）的所有权限项。

```
REVOKE ALL PRIVILEGES, GRANT OPTION
FROM user [, user] ...
```

- 示例
  - 删除hanson@'%'在easydb.emp表的CREATE、DROP、INDEX权限。

```
REVOKE CREATE,DROP,INDEX ON easydb.emp FROM hanson@'%';
```

  - 删掉账号lily@30.9.73.96的所有权限。

```
REVOKE ALL PRIVILEGES,GRANT OPTION FROM lily@30.9.73.96;
```

> **说明** 为了兼容MySQL，需同时写上GRANT OPTION。

查询授权（SHOW GRANTS）

- 语法

```
SHOW GRANTS[ FOR user@host];
```

- 示例

```
SHOW GRANTS FOR user1@host;
```

> **说明** 5.3.6及以上版本，SHOW GRANTS只显示当前用户权限，可在阿里云PolarDB-X 1.0控制台查看所有账号和权限信息。

7.2. 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)
```

7.3. 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，请参见使用全局二级索引。
> - 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，请参见控制参数与行为。

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 00:00:00.000000","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_timestam
p_6":"2000-01-01 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestam
p_1":"2000-01-01 00:00:00.0","c_year":"2000","c_int_32":255},"GSI":{"c_int_32_un":12345
6,"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_timestam
p_6":"2000-01-01 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestam
p_1":"2000-01-01 00:00:00.0","c_blob_tiny":"YeS4reWbvWE=","c_int_32":280}}
|
| `g_i_check` | SUMMARY    | --     | --          | 3 error found (7025/7025 rows check
ed)
|
+-------------+-----------+--------+-------------+-----------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
+
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_timestam
p_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_timestam
p_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,"i
d":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` | ORPHAN     | REPAIRED | (100722)    | {"GSI":{"id":100722,"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_binary":"OTkAAAAAAAAAA==","c_int_32":271}}
|
| `g_i_check` | SUMMARY    | --       | --          | 3 error found (7025/7026 rows check
ed.) 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。

7.4. KILL

您可以使用KILL指令终止一个正在PolarDB-X 1.0上执行的SQL。

前提条件

终止一个PolarDB-X 1.0上正在执行的SQL前，您需要先连接PolarDB-X 1.0之后才可以通过执行KILL语句终止正在执行的SQL，关于如何连接PolarDB-X 1.0，详情请参见快速入门。

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 1.0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 1.0上所有正在执行的物理SQL:

```
KILL 'ALL'
```

7.5. USE

本文将介绍如何通过USE指令在PolarDB-X 1.0中切换当前连接的默认数据库。

背景信息

PolarDB-X 1.0支持访问同一PolarDB-X 1.0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 1.0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。

注意事项

- 切换前，您需要先在控制台里进行Schema的授权，详情请参见账号和权限系统。
- 当切换到新的数据库后，原SQL语句的中HINT语法和Sequence（没有指定Schema的情况）语法也都会默认指向到新的数据库。

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.Sequence

8.1. 概述

本文将为您介绍Sequence的相关概念和支持的类型。

PolarDB-X 1.0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 1.0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 1.0自动维护。

支持的Sequence类型及其特性

PolarDB-X 1.0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

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
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 1.0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。

8.2. 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 1.0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 1.0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 1.0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 1.0生成的Sequence值，那么后续让PolarDB-X 1.0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 1.0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;
   ```

8.3. 显式用法

本文主要介绍了Sequence的显式用法。

创建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
  - 方法一

    ```
    mysql> CREATE SEQUENCE seq1;
    ```

  - 方法二

    ```
    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。
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）。

i. 实例1/库1：

```
mysql> CREATE GROUP SEQUENCE seq2 UNIT COUNT 3 INDEX 0;
```

ii. 实例2/库2：

```
mysql> CREATE GROUP SEQUENCE seq2 UNIT COUNT 3 INDEX 1;
```

iii. 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

```
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。
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。

```
mysql> CREATE SIMPLE SEQUENCE seq4 START WITH 1000 INCREMENT BY 2 MAXVALUE 99999999999 NO
CYCLE;
```

修改Sequence

PolarDB-X 1.0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

- 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。
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。
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。

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。

> **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;
```

查询与获取Sequence

查询Sequence

- 语法

```
SHOW SEQUENCES
```

- 示例

```
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       | 9999
9999999 | N     | SIMPLE |
+------+--------+------------+------------+------------+--------------+------------+-----
--------+-------+--------+
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

```
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

```
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 1.0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

```
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)
```

删除Sequence

- 语法

```
DROP SEQUENCE <name>
```

- 示例

```
mysql> DROP SEQUENCE seq3;
```

8.4. 隐式用法

本文介绍了Sequence的隐式用法。

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 1.0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 1.0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。

创建Group Sequence、Time-based Sequence或Simple Sequence

语法

```
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。

创建单元化Group Sequence

语法

```
CREATE TABLE <name> (
   <column> ... AUTO_INCREMENT [ BY GROUP ] [ UNIT COUNT <numeric value> INDEX <numeric val
ue> ],
   <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。

  ```
  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。

  i. 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);
  ```

  ii. 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);
  ```

  iii. 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。

```
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。

```
mysql> CREATE TABLE tab4 (
col1 BIGINT NOT NULL AUTO_INCREMENT BY SIMPLE,
col2 VARCHAR(16),
PRIMARY KEY(col1)
) DBPARTITION BY HASH(col1);
```

修改Sequence

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）。

查看表信息及相关Sequence类型

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**。

  ```
  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。

  ```
  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。

```
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。

```
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。

```
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
| 9223372036854775807 | N     | SIMPLE  |
+---------------+--------+------------+------------+------------+--------------+-----------
-+---------------------+-------+---------+
8 rows in set (0.01 sec)
```

9.Outline

9.1. 使用说明

本文介绍了Outline的功能及用法。

背景介绍

在使用PolarDB-X 1.0数据库的过程中，可能遇到某些SQL优化器生成的执行计划，并不是期望的结果，或者生成的计划并不是最优的，比如有些Join、Aggregate 函数可以下推到下层RDS执行的，但是并没有下推。Outline功能提供了一种给SQL指定执行计划的方式，您可以通过Hint的方式手工构建SQL的执行计划，并通过Outline的方式将构建的执行计划指定为SQL的执行计划。

Outline功能提供了CREATE，DROP，RESYNC，DISABLE，ENABLE，SHOW指令来创建和管理系统中的Outline，下面对各个指令进行说明。

约束

- 多语句不支持。
- GROUP BY，ORDER BY不支持"?"绑定变量。
- 在参数化匹配模式下，origin_stmt中不能含有常量。
- 在参数化匹配模式下，origin_stmt和target_stmt中含有的绑定变量数必须相等。
- 在完全匹配模式下，target_stmt中不能含有绑定变量。
- 创建OUTLINE时，origin_stmt不能与系统中已经存在的相同。
- 创建OUTLINE时， target_stmt语义必须正确，能正确生成执行计划。

创建Outline

CREATE指令用来创建Outline，创建后默认生效。

```
CREATE outline name ON origin_stmt TO target_stmt
```

参数说明：

- name是指创建的Outline名称；
- origin_stmt是指用来匹配SQL语句。当SQL不含"?"变量时，匹配必须完全相同，为完全匹配模式；
- 当含有"?"变量时，SQL中不能包含常量，并将SQL格式化后来做匹配，为参数化匹配模式；
- target_stmt是指用hint方式指定生成逻辑计划的语句。

示例一：创建一个完全匹配的Outline

```
mysql> create outline t1 on select 1 to select 2;
Query OK, 1 row affected (1.09 sec)
mysql> select 1;
+------+
| ?    |
+------+
|    2 |
+------+
```

可以看到执行的时候select 1语句被替换为select 2语句。

示例二：创建一个参数化匹配的Outline

```
mysql> create outline t2 on select ? to select /*+TDDL:slave()*/ * from ms10 where c1=?;
Query OK, 1 row affected (0.16 sec)
mysql> explain select 1;
+-------------------------------------------------------------------------------+
| LOGICAL PLAN                                                                  |
+-------------------------------------------------------------------------------+
| LogicalView(tables="01.ms10", sql="SELECT * FROM `ms10` WHERE (`c1` = ?)") |
| HitCache:false                                                                |
| UsingOutline: T2                                                              |
+-------------------------------------------------------------------------------+
```

删除Outline

DROP指令用来删除指定的Outline。

```
DROP OUTLINE name  #name是指需要同步的Outline名称
```

重新同步Outline

由于PolarDB-X 1.0实例是由多台Server组成的，创建Outline时，可能会出现同步到其他Server报SYNC error，这时需要重新同步。

```
RESYNC OUTLINE name  #name是指需要同步的Outline名称
```

停用指定Outline

DISABLE指令用来停用指定的Outline。

```
DISABLE OUTLINE name #name是指定的Outline名称
```

启用指定Outline

ENABLE指令用来启用指定的Outline。

```
ENABLE OUTLINE name  #name是指定的Outline名称
```

显示系统中的Outlines

SHOW指令用来显示系统中的Outlines。

```
SHOW OUTLINES
```

9.2. ERROR_CODE说明

本文介绍了常见的ERROR_CODE及含义。

- ERR_ORIGIN_STMT_UNEXPECTED_CONST：在参数化匹配模式下，origin stmt中含有未参数化的常量。

- ERR_PARAM_COUNT_NOT_EQUAL: 在参数化匹配模式下，origin_stmt和target_stmt中含有的绑定变量数不相等。
- ERR_TARGET_STMT_UNEXPECTED_PARAM: 在完全匹配模式下，target_stmt中含有绑定变量。
- ERR_ORIGIN_STMT_CONFLICTED: 创建Outline的origin_stmt与系统中已经存在的Outline的origin_stmt相同。
- ERR_TARGET_STMT_ERROR: 创建Outline的target_stmt生成执行计划失败。

10.Prepare SQL

10.1. Prepare协议使用说明

本文介绍了Prepare协议的概念、用途及在Java中的开启方法。

介绍

PolarDB-X 1.0提供对服务器端预处理语句的支持，支持利用高效的客户端/服务器二进制协议。使用准备好的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。

详细说明

- Prepare协议支持范围
  - Prepare协议目前支持：
    - COM_STMT_PREPARE
    - COM_STMT_EXECUTE
    - COM_STMT_CLOSE
    - COM_STMT_RESET
  - 支持使用Java及其他各种语言。
  - MySQL支持范围参见Prepared Statements。
- 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。
- Prepare协议不支持在MySQL命令行中使用如下方法：

```
mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
mysql> PREPARE stmt2 FROM @s;
mysql> SET @a = 6;
mysql> SET @b = 8;
mysql> EXECUTE stmt2 USING @a, @b;
```

在Java中开启Prepare协议

- 在Java客户端中，如果需要使用Prepare协议，需要强行在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?useSe
rverPrepStmts=true", "xxxxx", "xxxxx");
String sql = "insert into batch values(?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
preparedStatement.setInt(1, 0);
preparedStatement.setString(2, "corona-db");
preparedStatement.executeUpdate();
```

11.Hint

11.1. Hint简介

本文介绍了自定义HINT的用途以及基本语法。

本文适用于PolarDB-X 1.0 5.3及以上版本，其他版本请参见如何使用HINT（5.2及以下版本适用）

简介

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。

PolarDB-X 1.0也提供了特殊的HINT语法。例如已知目标数据在某些分库的分表中，需要直接将SQL下发到该分库执行，就可以使用PolarDB-X 1.0自定义HINT来完成。

PolarDB-X 1.0自定义HINT语法

语法

```
/*+TDDL: hint_command [hint_command ...]*/
/!+TDDL: hint_command [hint_command ...]*/
```

注意事项

- PolarDB-X 1.0自定义HINT支持 `/*+TDDL:hint_command*/` 和 `/!+TDDL:hint_command*/` 两种格式。
- HINT语句位于 `/*` 与 `*/` 或 `/!` 与 `*/` 之间，并且必须以 `+TDDL:` 开头。其中 `hint_command` 是PolarDB-X 1.0自定义HINT命令，与具体的操作相关，多个 `hint_command` 之间使用空格分割。
- 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 1.0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 1.0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 1.0自定义HINT失效。具体请查看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;
```

示例中 `/*+TDDL:scan()*/` 和 `/*+TDDL:node(0) slave()*/` 为PolarDB-X 1.0自定义HINT部分，以 `+TDDL:` 开头。 `scan()` 、 `node(0)` 、 `slave()` 为PolarDB-X 1.0自定义HINT命令，多个HINT命令之间使用空格分割。

在SQL语句中使用HINT

PolarDB-X 1.0支持在DML、DDL、DAL语句中使用HINT，具体语法如下：

- 对于所有支持HINT的语句，允许在语句前指定HINT，例如：

```
/*+TDDL: ... */ SELECT ...
/*+TDDL: ... */ INSERT ...
/*+TDDL: ... */ REPLACE ...
/*+TDDL: ... */ UPDATE ...
/*+TDDL: ... */ DELETE ...
/*+TDDL: ... */ CREATE TABLE ...
/*+TDDL: ... */ ALTER TABLE ...
/*+TDDL: ... */ DROP TABLE ...
/*+TDDL: ... */ SHOW ...
...
```

- 对于DML语句，允许在首个关键字之后指定HINT，例如：

```
SELECT /*+TDDL: ... */  ...
INSERT /*+TDDL: ... */  ...
REPLACE /*+TDDL: ... */  ...
UPDATE /*+TDDL: ... */  ...
DELETE /*+TDDL: ... */  ...
...
```

> **说明** 不同HINT支持的语句范围可能不同，实际支持情况请参考具体HINT命令说明文档。

使用多个HINT

PolarDB-X 1.0支持在HINT语句中使用多个HINT命令，例如：

```
SELECT /*+TDDL:node(0) slave()*/ ...;
```

PolarDB-X 1.0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)*/ ...;
```

PolarDB-X 1.0自定义HINT分类

- 读写分离
- 自定义SQL超时时间
- 指定分库执行SQL
- 扫描全部/部分分库分表

11.2. 读写分离

本文介绍了读写分离类的Hint语法。

本文适用于PolarDB-X 1.0 5.3及以上版本。

PolarDB-X 1.0提供了一种针对应用层透明的读写分离实现。但是由于RDS主实例与只读实例之间数据的同步存在着毫秒级别的延迟，如果在主库中变更以后需要马上读取变更的数据，则需要保证将读取数据的SQL下发到主实例中。针对这种需求，PolarDB-X 1.0提供了读写分离自定义HINT，指定将SQL下发到主实例或者只读实例。

语法

```
/*+TDDL:
    master()
    | slave()
*/
```

在该自定义HINT中可以指定SQL是在主实例上执行还是在只读实例上执行。对于 `/*+TDDL:slave()*/` ，如果一个主RDS实例存在多个只读实例，那么PolarDB-X 1.0会根据所分配的权重随机选择一个只读实例执行SQL语句。

注意事项

- PolarDB-X 1.0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 1.0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 1.0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 1.0自定义HINT失效。具体请查看MySQL 官方客户端命令。

示例

- 指定SQL在主实例上执行：

  ```
  SELECT /*+TDDL:master()*/ * FROM table_name;
  ```

  在SQL第一个关键字之后添加 `/*+TDDL:master()*/` 这个自定义HINT后，这条SQL将被下发到主实例上执行。

- 指定SQL在只读实例上执行：

  ```
  SELECT /*+TDDL:slave()*/ * FROM table_name;
  ```

  在SQL第一个关键字之后添加 `/*+TDDL:slave()*/` 这个自定义HINT后，这条SQL将会根据所分配的权重被随机下发到某个只读实例上执行。

  > **说明**
  > - 此读写分离自定义HINT仅仅针对非事务中的读SQL语句生效，如果SQL语句是写SQL或者SQL语句在事务中，那么还是会下发到RDS的主实例执行。
  > - PolarDB-X 1.0针对 `/*+TDDL:slave()*/` 自定义HINT，会从只读实例中按照权重随机选取一个下发SQL语句执行。若只读实例不存在时，不会报错，而是选取主实例执行。

11.3. 自定义SQL超时时间

本文介绍了自定义SQL超时时间HINT的语法和示例。

本文适用于PolarDB-X 1.0 5.3及以上版本，其他版本请参见如何使用HINT（5.2及以下版本适用）。

在PolarDB-X 1.0中，PolarDB-X 1.0节点与RDS的默认的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 1.0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。

语法

```
/*+TDDL:SOCKET_TIMEOUT(time)*/
```

其中，`SOCKET_TIMEOUT` 的单位是毫秒。通过该HINT用户可以根据业务需要，自由调整SQL语句的超时时间。

注意事项

- PolarDB-X 1.0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 1.0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 1.0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 1.0自定义HINT失效。具体请查看MySQL 官方客户端命令。

示例

设置SQL超时时间为40秒：

```
/*+TDDL:SOCKET_TIMEOUT(40000)*/SELECT * FROM t_item;
```

> **说明** 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。

11.4. 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。

本文适用于PolarDB-X 1.0 5.3及以上版本，其他版本请参见如何使用HINT（5.2及以下版本适用）。

在使用PolarDB-X 1.0的过程中，如果遇到某个PolarDB-X 1.0不支持的SQL语句，可以通过PolarDB-X 1.0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 1.0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 1.0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 1.0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

- 从版本5.4.1开始，PolarDB-X 1.0在拆分表的物理表名中增加了4个字符的随机串，请务必使用SHOW TOPOLOGY命令获取逻辑表拓扑和实际的物理表名。
- 从版本5.4.4 开始，PolarDB-X 1.0提供开关来控制拆分表的物理表名中是否包含随机串，默认为开启，可以在控制台“参数设置”的数据库级别参数中，将“是否启用随机物理表名ENABLE_RANDOM_PHY_TABLE_NAME”改为false来关闭，也可以用HINT来实现语句级别的控制：`/*+TDDL:cmd_extra(ENABLE_RANDOM_PHY_TABLE_NAME=FALSE)*/` 。
- PolarDB-X 1.0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 1.0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 1.0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 1.0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 1.0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 1.0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_147347
1355140LRPRDRDS_TEST_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; ```
```

11.5. 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。

本文适用于PolarDB-X 1.0 5.3及以上版本，其他版本请参见如何使用HINT（5.2及以下版本适用）。

除了可以将SQL单独下发到一个或多个分库执行，PolarDB-X 1.0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 1.0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 1.0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 1.0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 1.0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 1.0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 JO
IN 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。

11.6. 高危类SQL自动保护

为避免因误操作导致数据丢失，PolarDB-X 1.0默认禁止执行高危类SQL，例如全表删除（即不带 `WHERE` 或 `LIMIT` 条件的 `DELETE` 语句）、全表更新（即不带 `WHERE` 或 `LIMIT` 条件的 `UPDATE` 语句）等语句。但您也可以通过HINT语句避开上述自动保护，强制执行全表删除或更新操作。

语法

您可以在 `UPDATE` 或 `DELETE` 语句中加上如下HINT语句，执行全表删除或更新：

```
/!TDDL:FORBID_EXECUTE_DML_ALL=false*/
```

示例

- 若 `DELETE` 语句中未加任何 `WHERE` 或 `LIMIT` 条件，执行时会被拦截且出现如下错误提示：

```
DELETE FROM tt;
ERR-CODE: [TDDL-4620][ERR_FORBID_EXECUTE_DML_ALL] Forbid execute DELETE ALL or UPDATE ALL
sql. More: [http://
example.aliyundoc.com/faq/faqByFaqCode.html?faqCode=TDDL-4620]
```

在上述语句中加上如下HINT则可成功执行：

```
/!TDDL:FORBID_EXECUTE_DML_ALL=false*/DELETE FROM tt;
Query OK, 10 row affected (0.21 sec)
```

- 若 `UPDATE` 语句中未加任何 `WHERE` 或 `LIMIT` 条件，执行时会被拦截且出现如下错误提示：

```
UPDATE tt SET id = 1;
ERR-CODE: [TDDL-4620][ERR_FORBID_EXECUTE_DML_ALL] Forbid execute DELETE ALL or UPDATE ALL
sql. More: [http://example.aliyundoc.com/faq/faqByFaqCode.html?faqCode=TDDL-4620]
```

在上述语句中加上如下HINT则可成功执行：

```
/! TDDL:FORBID_EXECUTE_DML_ALL=false*/UPDATE tt SET id = 1;
Query OK, 10 row affected (0.21 sec)
```

11.7. INDEX HINT

PolarDB-X 1.0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

- MySQL版本需为5.7或以上，且PolarDB-X 1.0内核小版本需为5.4.1或以上。
- INDEX HINT仅对SELECT语句生效。

注意事项

PolarDB-X 1.0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 1.0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 1.0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 1.0自定义HINT失效，详情请参见MySQL官方客户端命令。

语法

PolarDB-X 1.0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
```

11.8. 如何使用HINT（5.2及以下版本适用）

本文介绍了在5.2及以下版本适用HINT的方法。

基本语法

```
 /!TDDL:hint command*/
```

PolarDB-X 1.0自定义HINT是借助于 MySQL 注释实现的，也就是PolarDB-X 1.0的自定义HINT语句位于 `/!` 与 `*/` 之间，并且必须以 `TDDL:` 开头。其中 `hint command` 是PolarDB-X 1.0自定义HINT命令，与具体的操作相关。

例如下面的SQL语句通过PolarDB-X 1.0的自定义HINT展示每个分库的表名。

```
/!TDDL:SCAN*/SHOW TABLES;
```

该SQL语句中 `/!TDDL:SCAN*/` 为PolarDB-X 1.0自定义HINT部分，以 `TDDL:` 开头， `SCAN` 为PolarDB-X 1.0自定义HINT命令。

读写分离

由于RDS主实例与只读实例之间数据的同步存在着毫秒级别的延迟，如果在主库中变更后需要马上读取变更的数据，则需要保证将读取数据的SQL下发到主实例中。针对这种需求，PolarDB-X 1.0提供了读写分离自定义HINT，指定将SQL下发到主实例或者只读实例。

语法

```
/!TDDL:MASTER|SLAVE*/
```

在该自定义HINT中可以指定SQL是在主实例上执行还是在只读实例上执行。对于 `/!TDDL:SLAVE*/` 这个自定义HINT，如果一个主RDS实例存在多个只读实例，那么PolarDB-X 1.0会根据所分配的权重随机选择一个只读实例执行SQL语句。

示例

- 指定SQL在主实例上执行：

```
/!TDDL:MASTER*/SELECT * FROM table_name;
```

在SQL语句前添加 `/!TDDL:MASTER*/` 这个自定义HINT后，这条SQL将被下发到主实例上执行。

- 指定SQL在只读实例上执行：

```
/!TDDL:SLAVE*/SELECT * FROM table_name;
```

在SQL语句前添加 `/!TDDL:SLAVE*/` 这个自定义HINT后，这条SQL将会根据所分配的权重被随机下发到某个只读实例上执行。

> **注意**
>
> - 此读写分离自定义HINT仅仅针对非事务中的读SQL语句生效，如果SQL语句是写SQL或者SQL语句在事务中，那么还是会下发到RDS的主实例执行。
> - PolarDB-X 1.0针对 `/!TDDL:SLAVE*/` 自定义HINT，会从只读实例中按照权重随机选取一个下发SQL语句执行。若只读实例不存在时，不会报错，而是选取主实例执行。

只读实例延迟切断

正常情况下，如果给PolarDB-X 1.0数据库的RDS主实例配置了只读实例，并且给主实例和只读实例都设置了读流量，那么PolarDB-X 1.0会根据读写比例将SQL下发到主实例或者是只读实例执行。但是如果主实例与只读实例的异步数据复制存在较大的延迟，将SQL下发到只读实例执行就会导致出错或者返回错误结果。

只读实例延时切断会根据主备复制最大延时时间判断将所执行的SQL下发到主实例还是只读实例。

语法

```
/!TDDL:SQL_DELAY_CUTOFF=time*/
```

在自定义HINT中指定 `SQL_DELAY_CUTOFF` 的值，当备库的 `SQL_DELAY` 值（MySQL主备复制延迟）达到或超过 `time` 的值（单位秒）时，查询语句会被下发到主实例。

示例

- 指定主备复制延迟时间为5秒：

```
/!TDDL:SQL_DELAY_CUTOFF=5*/SELECT * FROM table_name;
```

在SQL语句指定了 `SQL_DELAY_CUTOFF` 的值为5，当备库的 `SQL_DELAY` 值达到或超过5秒时，查询语句会下发到主实例执行。

- 配合其他自定义HINT使用：

```
/!TDDL:SLAVE AND SQL_DELAY_CUTOFF=5*/SELECT * FROM table_name;
```

备库延迟切断注释也可以配合其他注释使用，该SQL查询请求默认会被下发到只读实例，但是当出现主备复制延时达到或超过5秒时，会下发到主实例。

自定义SQL超时时间

在PolarDB-X 1.0中，PolarDB-X 1.0节点与RDS的默认的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 1.0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。

语法

```
/!TDDL:SOCKET_TIMEOUT=time*/
```

其中，`SOCKET_TIMEOUT` 的单位是毫秒。通过该 HINT 用户可以根据业务需要，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;
```

> **说明** 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。

指定分库执行SQL

在使用PolarDB-X 1.0的过程中，如果遇到某个PolarDB-X 1.0不支持的SQL语句，可以通过PolarDB-X 1.0提供的自定义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，也可以使用该自定义HINT，直接将SQL语句下发到分库中执行。

指定分库执行SQL自定义HINT有两种使用方式，即通过分片名指定SQL在分库上执行或者通过分库键值指定SQL在分库上执行。其中分片名是PolarDB-X 1.0中分库的唯一标识，可以通过 `SHOW NODE` 控制指令得到。

语法

```
/!TDDL:NODE='node_name'*/
```

`node_name` 为分片名，通过这个PolarDB-X 1.0自定义HINT，就可以将SQL下发到 `node_name` 对应的分库中执行。

```
/!TDDL:NODE IN ('node_name'[,'node_name1','node_name2'])*/
```

使用 `in` 关键字指定多个分片名，将SQL下发到多个分库上执行，括号内分片名之间使用逗号分隔。

> **说明** 使用该自定义HINT时，PolarDB-X 1.0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。

```
/!TDDL:table_name.partition_key=value [and table_name1.partition_key=value1]*/
```

在这个PolarDB-X 1.0自定义HINT中 `table_name` 为逻辑表名，该表是一张拆分表， `partition_key` 是拆分键， `value` 为指定的拆分键的值。在该自定义注释中，可以使用 `and` 关键字指定多个拆分表的拆分键。通过这个PolarDB-X 1.0自定义HINT，PolarDB-X 1.0会计算出SQL语句应该在哪些分库和分表上执行，进而将SQL语句下发到相应的分库。

示例

对于名为 `drds_test` 的PolarDB-X 1.0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 1.0的自定义HINT指定分库执行SQL语句了。

- 指定SQL在第0个分库上执行：

  ```
  /!TDDL:NODE='DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS'*/SELECT * FROM table_nam
  e;
  ```

- 指定SQL在多个分库上执行：

  ```
  /!TDDL:NODE  IN('DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS','DRDS_TEST_147347135
  5140LRPRDRDS_TEST_VTLA_0006_RDS')*/SELECT * 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。

- 查看某个分库的执行计划：

  ```
  /!TDDL:NODE='DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS'*/EXPLAIN SELECT * FROM t
  able_name;
  ```

  这条SQL语句将会展示 `SELECT` 语句在分片 `DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS` 中的执行计划。

- 通过键值指定SQL在分库上执行：

  对于 `UPDATE` 语句，PolarDB-X 1.0不支持 `SET` 子句中的子查询，由于 `UPDATE` 语句在PolarDB-X 1.0中必须指定拆分键，所以可以使用PolarDB-X 1.0的自定义HINT将该语句直接下发到分库上执行。

  比如有两张逻辑表，分别是t1和t2，它们都是分库分表，建表语句如下：

  ```
    CREATE TABLE `t1` (
      `id` bigint(20) NOT NULL,
      `name` varchar(20) NOT NULL,
      `val` varchar(20) DEFAULT NULL,
      PRIMARY KEY (`id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`nam
  e`) tbpartitions 3
    CREATE TABLE `t2` (
      `id` bigint(20) NOT NULL,
      `name` varchar(20) NOT NULL,
      `val` varchar(20) DEFAULT NULL,
      PRIMARY KEY (`id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`nam
  e`) tbpartitions 3
  ```

需要执行的语句是：

```
UPDATE t1 SET val=(SELECT val FROM t2 WHERE id=1) WHERE id=1;
```

这条语句直接在PolarDB-X 1.0上执行会报不被支持的错误，但是可以给这条语句加上PolarDB-X 1.0的自定义HINT，再提交到PolarDB-X 1.0执行。具体SQL语句如下：

```
/!TDDL:t1.id=1 and t2.id=1*/UPDATE t1 SET val=(SELECT val FROM t2 WHERE id=1) WHERE id=1;
```

这条语句会被下发到 `t1` 的 `ID` 为1的分库上执行。通过explain命令可以看到执行这条SQL语句的执行计划：

```
  mysql> explain /!TDDL:t1.id=1 and t2.id=1*/UPDATE t1 SET val=(SELECT val FROM t2 WHERE
id=1) WHERE id=1\G
  *************************** 1. row ***************************
  GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
         SQL: UPDATE `t1_2` AS `t1` SET `val` = (SELECT val FROM `t2_2` AS `t2` WHERE `id
` = 1) WHERE `id` = 1
      PARAMS: {}
  *************************** 2. row ***************************
  GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
         SQL: UPDATE `t1_1` AS `t1` SET `val` = (SELECT val FROM `t2_1` AS `t2` WHERE `id
` = 1) WHERE `id` = 1
      PARAMS: {}
  *************************** 3. row ***************************
  GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
         SQL: UPDATE `t1_0` AS `t1` SET `val` = (SELECT val FROM `t2_0` AS `t2` WHERE `id
` = 1) WHERE `id` = 1
      PARAMS: {}
  3 rows in set (0.00 sec)
```

从 `explain` 命令的结果集可以看到，SQL语句被改写成3条语句下发到分库上执行。还可以继续指定分表键值，将SQL执行范围缩小到一张分表：

```
  mysql> explain /!TDDL:t1.id=1 and t2.id=1 and t1.name='1'*/UPDATE t1 SET val=(SELECT va
l FROM t2 WHERE id=1) WHERE id=1\G
  *************************** 1. row ***************************
  GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
         SQL: UPDATE `t1_1` AS `t1` SET `val` = (SELECT val FROM `t2_1` AS `t2` WHERE `id
` = 1) WHERE `id` = 1
      PARAMS: {}
  1 row in set (0.00 sec)
```

说明 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 1.0计算出的两个键值对应的分库不一致，就会报错。

扫描全部分库分表

除了可以将SQL单独下发到一个或多个分库执行，PolarDB-X 1.0还提供了扫描全部分库与分表的自定义HINT。通过这个自定义HINT，您可以一次将SQL下发到每一个分库执行。比如通过这个自定义HINT，可以查看某个分库上的所有分表。还可以通过这个自定义HINT，查看某个逻辑表的每个分库的分表数据量。

扫描全部分片的PolarDB-X 1.0自定义HINT有两种使用方式，第一种方式是将SQL语句下发到每个分库执行，第二种方式是将SQL语句下发到每个分库上对某个逻辑表进行操作。

- 将SQL下发到全部分库上执行：

```
/!TDDL:SCAN*/
```

- 对某个逻辑表进行操作：

```
/!TDDL:SCAN='table_name'*/
```

其中 `table_name` 是PolarDB-X 1.0数据库的某个逻辑表名。该自定义HINT是为分库分表提供的，请尽量确保 `table_name` 为分库分表。

12.函数

12.1. 函数

本文展示了PolarDB-X 1.0支持的函数及部分不支持函数的列表。

PolarDB-X 1.0支持的函数分为日期时间函数、字符串函数、转换函数、聚合函数、数学函数、比较函数、位函数、控制流程函数、信息函数、加密和压缩函数以及其他函数； JSON函数和地理信息函数的下推执行。

以下函数出现在WHERE条件、UPDATE语句中，PolarDB-X 1.0不支持：
LAST_INSERT_ID()，CONNECTION_ID()，CURRENT_USER()，CURRENT_USER
DATABASE()，SCHEMA()，USER()，VERSION()。

与MySQL5.7相比，PolarDB-X 1.0不支持以下几类函数：

- 全文检索函数
- XML 函数
- GTID 函数
- 企业加密函数

已经支持的几类函数中，有如下函数不支持：

类别	函数名	描述
日期时间函数	CONVERT_TZ()	Convert from one time zone to another
	GET_FORMAT()	Return a date format string
	LOCALTIME(), LOCALTIME	Synonym for NOW()
	LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
字符串函数	FIND_IN_SET()	Return the index position of the first argument within the second argument
	LOAD_FILE()	Load the named file
	MATCH	Perform full-text search
	SOUNDS LIKE	Compare sounds
	BIT_AND(	Return bitwise AND
	BIT_OR()	Return bitwise OR
	BIT_XOR()	Return bitwise XOR
	GROUP_CONCAT()	Return a concatenated string
	STD()	Return the population standard deviation

聚合函数	STDDEV()	Return the population standard deviation
	STDDEV_POP()	Return the population standard deviation
	STDDEV_SAMP()	Return the sample standard deviation
	VAR_POP()	Return the population standard variance
	VAR_SAMP()	Return the sample variance
	VARIANCE()	Return the population standard variance
数学函数	RADIANS()	Return argument converted to radians
信息函数	BENCHMARK()	Repeatedly execute an expression
	CHARSET()	Return the character set of the argument
	COERCIBILITY()	Return the collation coercibility value of the string argument
	COLLATION()	Return the collation of the string argument
	FOUND_ROWS()	For a SELECT with a LIMIT clause, the number of rows that would be returned were there no LIMIT clause
	ROW_COUNT()	The number of rows updated
	ASYMMETRIC_DECRYPT()	Decrypt ciphertext using private or public key
	ASYMMETRIC_DERIVE()	Derive symmetric key from asymmetric keys
	ASYMMETRIC_ENCRYPT()	Encrypt cleartext using private or public key
	ASYMMETRIC_SIGN()	Generate signature from digest
	ASYMMETRIC_VERIFY()	Verify that signature matches digest
	CREATE_ASYMMETRIC_PRIV_KEY()	Create private key

加密和压缩函数	CREATE_ASYMMETRIC_PUB_KEY()	Create public key
	CREATE_DH_PARAMETERS()	Generate shared DH secret
	CREATE_DIGEST()	Generate digest from string
	DECODE() (deprecated 5.7.2)	Decodes a string encrypted using ENCODE()
	DES_DECRYPT() (deprecated 5.7.6)	Decrypt a string
	DES_ENCRYPT() (deprecated 5.7.6)	Encrypt a string
	ENCODE() (deprecated 5.7.2)	Encode a string
	ENCRYPT() (deprecated 5.7.6)	Encrypt a string
	OLD_PASSWORD()	Return the value of the pre-4.1 implementation of PASSWORD
	PASSWORD() (deprecated 5.7.6)	Calculate and return a password string
	RANDOM_BYTES()	Return a random byte vector
	SHA1(), SHA()	Calculate an SHA-1 160-bit checksum
	SHA2()	Calculate an SHA-2 checksum
	VALIDATE_PASSWORD_STRENGTH()	Determine strength of password
	ANY_VALUE()	Suppress ONLY_FULL_GROUP_BY value rejection
	DEFAULT()	Return the default value for a table column
	GET_LOCK()	Get a named lock
	INET_ATON()	Return the numeric value of an IP address
	INET_NTOA()	Return the IP address from a numeric value
	INET6_ATON()	Return the numeric value of an IPv6 address
	INET6_NTOA()	Return the IPv6 address from a numeric value

其他函数	IS_FREE_LOCK()	Whether the named lock is free
	IS_IPV4()	Whether argument is an IPv4 address
	IS_IPV4_COMPAT()	Whether argument is an IPv4-compatible address
	IS_IPV4_MAPPED()	Whether argument is an IPv4-mapped address
	IS_IPV6()	Whether argument is an IPv6 address
	IS_USED_LOCK()	Whether the named lock is in use; return connection identifier if true
	MASTER_POS_WAIT()	Block until the slave has read and applied all updates up to the specified position
	NAME_CONST()	Causes the column to have the given name

12.2. 日期时间函数

本文介绍了PolarDB-X 1.0支持及不支持的日期时间函数。

支持函数

PolarDB-X 1.0支持如下日期时间函数：

函数名	描述
ADDDATE()	Add time values (intervals) to a date value
ADDTIME()	Add time
CURDATE()	Return the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Return the current time
DATE()	Extract the date part of a date or datetime expression
DATE_ADD()	Add time values (intervals) to a date value

函数名	描述
DATE_FORMAT()	Format date as specified
DATE_SUB()	Subtract a time value (interval) from a date
DATEDIFF()	Subtract two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Return the name of the weekday
DAYOFMONTH()	Return the day of the month (0-31)
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
EXTRACT()	Extract part of a date
FROM_DAYS()	Convert a day number to a date
FROM_UNIXTIME()	Format Unix timestamp as a date
HOUR()	Extract the hour
LAST_DAY()	Return the last day of the month for the argument
MAKEDATE()	Create a date from the year and day of year
MAKETIME()	Create time from hour, minute, second
MICROSECOND()	Return the microseconds from argument
MINUTE()	Return the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Return the name of the month
NOW()	Return the current date and time
PERIOD_ADD()	Add a period to a year-month
PERIOD_DIFF()	Return the number of months between periods
QUARTER()	Return the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format
SECOND()	Return the second (0-59)
STR_TO_DATE()	Convert a string to a date

函数名	描述
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments
SUBTIME()	Subtract times
SYSDATE()	Return the time at which the function executes
TIME()	Extract the time portion of the expression passed
TIME_FORMAT()	Format as time
TIME_TO_SEC()	Return the argument converted to seconds
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Subtract an interval from a datetime expression
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()	Return a Unix timestamp
UTC_DATE()	Return the current UTC date
UTC_TIME()	Return the current UTC time
UTC_TIMESTAMP()	Return the current UTC date and time
WEEK()	Return the week number
WEEKDAY()	Return the weekday index
WEEKOFYEAR()	Return the calendar week of the date (1-53)
YEAR()	Return the year
YEARWEEK()	Return the year and week

不支持函数

与MySQL5.7相比，PolarDB-X 1.0暂不支持如下函数：

函数名	描述
CONVERT_TZ()	Convert from one time zone to another
GET_FORMAT()	Return a date format string
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()

另外，暂不支持不带参数的`UNIX_TIMESTAMP()`函数，可用`UNIX_TIMESTAMP(NOW())`替换。

12.3. 字符串函数

本文介绍了PolarDB-X 1.0支持及不支持的字符串函数。

支持函数

PolarDB-X 1.0支持如下字符串函数：

函数名	描述
ASCII()	Return numeric value of left-most character
BIN()	Return a string containing binary representation of a number
BIT_LENGTH()	Return length of argument in bits
CHAR()	Return the character for each integer passed
CHAR_LENGTH()	Return number of characters in argument
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenate with separator
ELT()	Return string at index number
EXPORT_SET()	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
FIELD()	Return the index (position) of the first argument in the subsequent arguments
FORMAT()	Return a number formatted to specified number of decimal places
FROM_BASE64()	Decode to a base-64 string and return result

函数名	描述
HEX()	Return a hexadecimal representation of a decimal or string value
INSERT()	Insert a substring at the specified position up to the specified number of characters
INSTR()	Return the index of the first occurrence of substring
LCASE()	Synonym for LOWER()
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits set
MID()	Return a substring starting from the specified position
NOT LIKE	Negation of simple pattern matching
NOT REGEXP	Negation of REGEXP
OCT()	Return a string containing octal representation of a number
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Whether string matches regular expression
REPEAT()	Repeat a string the specified number of times

函数名	描述
REPLACE()	Replace occurrences of a specified string
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Whether string matches regular expression
RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
SOUNDEX()	Return a soundex string
SPACE()	Return a string of the specified number of spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string
TRIM()	Remove leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase
WEIGHT_STRING()	Return the weight string for a string

不支持函数

与MySQL5.7相比，PolarDB-X 1.0暂不支持如下字符串函数：

函数名	描述
FIND_IN_SET()	Return the index position of the first argument within the second argument
LOAD_FILE()	Load the named file
MATCH	Perform full-text search
SOUNDS LIKE	Compare sounds

12.4. 转换函数

本文介绍了PolarDB-X 1.0支持的转换函数。

PolarDB-X 1.0支持如下转换函数：

函数名	描述
BINARY	Cast a string to a binary string
CAST()	Cast a value as a certain type
CONVERT()	Cast a value as a certain type

CONVERT函数仅支持CONVERT(expr USING transcoding_name)形式。如需使用CONVERT(expr,type)，请使用CAST(expr AS type)代替。

12.5. 聚合函数

本文介绍了PolarDB-X 1.0支持及不支持的聚合函数。

支持函数

PolarDB-X 1.0支持如下聚合函数：

函数名	描述
AVG()	Return the average value of the argument
COUNT()	Return a count of the number of rows returned
COUNT(DISTINCT)	Return the count of a number of different values
MAX()	Return the maximum value
MIN()	Return the minimum value
SUM()	Return the sum

不支持函数

与MySQL5.7相比，PolarDB-X 1.0暂不支持如下聚合函数：

函数名	描述
BIT_AND()	Return bitwise AND
BIT_OR()	Return bitwise OR
BIT_XOR()	Return bitwise XOR
GROUP_CONCAT()	Return a concatenated string

函数名	描述
STD()	Return the population standard deviation
STDDEV()	Return the population standard deviation
STDDEV_POP()	Return the population standard deviation
STDDEV_SAMP()	Return the sample standard deviation
VAR_POP()	Return the population standard variance
VAR_SAMP()	Return the sample variance
VARIANCE()	Return the population standard variance

12.6. 数学函数

本文介绍了PolarDB-X 1.0支持及不支持的数学函数。

背景信息

支持函数

PolarDB-X 1.0支持如下数学函数：

函数名	描述
ABS()	Return the absolute value
ACOS()	Return the arc cosine
ASIN()	Return the arc sine
ATAN()	Return the arc tangent
ATAN2(), ATAN()	Return the arc tangent of the two arguments
CEIL()	Return the smallest integer value not less than the argument
CEILING()	Return the smallest integer value not less than the argument
CONV()	Convert numbers between different number bases
COS()	Return the cosine
COT()	Return the cotangent
CRC32()	Compute a cyclic redundancy check value
DEGREES()	Convert radians to degrees

函数名	描述
EXP()	Raise to the power of
FLOOR()	Return the largest integer value not greater than the argument
LN()	Return the natural logarithm of the argument
LOG()	Return the natural logarithm of the first argument
LOG10()	Return the base-10 logarithm of the argument
LOG2()	Return the base-2 logarithm of the argument
MOD()	Return the remainder
PI()	Return the value of pi
POW()	Return the argument raised to the specified power
POWER()	Return the argument raised to the specified power
RAND()	Return a random floating-point value
ROUND()	Round the argument
SIGN()	Return the sign of the argument
SIN()	Return the sine of the argument
SQRT()	Return the square root of the argument
TAN()	Return the tangent of the argument
TRUNCATE()	Truncate to specified number of decimal places

不支持函数

与MySQL5.7相比，PolarDB-X 1.0暂不支持如下数学函数：

函数名	描述
RADIANS()	Return argument converted to radians

12.7. 比较函数

本文介绍了PolarDB-X 1.0支持的比较函数。

PolarDB-X 1.0支持如下比较函数：

函数名	描述
COALESCE()	Return the first non-NULL argument
GREATEST()	Return the largest argument
IN()	Check whether a value is within a set of values
INTERVAL()	Return the index of the argument that is less than the first argument
ISNULL()	Test whether the argument is NULL
LEAST()	Return the smallest argument
NOT IN()	Check whether a value is not within a set of values
STRCMP()	Compare two strings

12.8. 位函数

本文介绍了位函数BIT_COUNT()。

PolarDB-X 1.0支持一个位函数，即BIT_COUNT()，其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)
```

12.9. 流程控制函数

本文介绍了PolarDB-X 1.0支持的流程控制函数。

PolarDB-X 1.0支持如下流程控制函数：

函数名	描述
CASE	Case operator

函数名	描述
IF()	If/else construct
IFNULL()	Null if/else construct
NULLIF()	Return NULL if expr1 = expr2

12.10. 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 1.0支持及不支持的信息函数。

支持函数

PolarDB-X 1.0支持如下信息函数：

函数名	描述
CONNECTION_ID()	Return the connection ID (thread ID) for the connection
CURRENT_USER(), CURRENT_USER	The authenticated user name and host name
DATABASE()	Return the default (current) database name
LAST_INSERT_ID()	Value of the AUTOINCREMENT column for the last INSERT
SCHEMA()	Synonym for DATABASE()
SESSION_USER()	Synonym for USER()
SYSTEM_USER()	Synonym for USER()
USER()	The user name and host name provided by the client
VERSION()	Return a string that indicates the MySQL server version

不支持函数

与 MySQL5.7 相比，PolarDB-X 1.0暂不支持如下信息函数：

函数	描述
BENCHMARK()	Repeatedly execute an expression
CHARSET()	Return the character set of the argument
COERCIBILITY()	Return the collation coercibility value of the string argument
COLLATION()	Return the collation of the string argument

函数	描述
FOUND_ROWS()	For a SELECT with a LIMIT clause, the number of rows that would be returned were there no LIMIT clause
ROW_COUNT()	The number of rows updated

12.11. 加密和压缩函数

本文主要介绍PolarDB-X 1.0支持和不支持的加密和压缩函数。

支持的加密和压缩函数

PolarDB-X 1.0支持如下加密和压缩函数。

函数名	描述
AES_ENCRYPT()	Encrypt using AES
AES_DECRYPT()	Decrypt using AES
MD5()	Calculate MD5 checksum
UNCOMPRESS()	Uncompress a string compressed
UNCOMPRESSED_LENGTH()	Return the length of a string before compression

不支持的加密和压缩函数

对比MySQL5.7，PolarDB-X 1.0暂不支持如下加密和压缩函数。

函数名	描述
ASYMMETRIC_DECRYPT()	Decrypt ciphertext using private or public key
ASYMMETRIC_DERIVE()	Derive symmetric key from asymmetric keys
ASYMMETRIC_ENCRYPT()	Encrypt cleartext using private or public key
ASYMMETRIC_SIGN()	Generate signature from digest
ASYMMETRIC_VERIFY()	Verify that signature matches digest
CREATE_ASYMMETRIC_PRIV_KEY()	Create private key
CREATE_ASYMMETRIC_PUB_KEY()	Create public key
CREATE_DH_PARAMETERS()	Generate shared DH secret
CREATE_DIGEST()	Generate digest from string
DECODE() (deprecated 5.7.2)	Decodes a string encrypted using ENCODE()

函数名	描述
DES_DECRYPT() (deprecated 5.7.6)	Decrypt a string
DES_ENCRYPT() (deprecated 5.7.6)	Encrypt a string
ENCODE() (deprecated 5.7.2)	Encode a string
ENCRYPT() (deprecated 5.7.6)	Encrypt a string
OLD_PASSWORD()	Return the value of the pre-4.1 implementation of PASSWORD
PASSWORD() (deprecated 5.7.6)	Calculate and return a password string
RANDOM_BYTES()	Return a random byte vector
SHA1(), SHA()	Calculate an SHA-1 160-bit checksum
SHA2()	Calculate an SHA-2 checksum
VALIDATE_PASSWORD_STRENGTH()	Determine strength of password

12.12. 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。

前提条件

PolarDB-X 1.0实例版本需为5.4.8及以上。关于如何查看实例版本，请参见查看实例版本。

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 sta
rt 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。
	• 仅实例版本为5.4.9或以上（若您的实例版本低于5.4.9，请升级版本）的PolarDB-X 1.0实例，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 PRECEDI
NG and CURRENT ROW) 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 |
```

12.13. 其他函数

本文介绍了PolarDB-X 1.0支持的其他函数。

PolarDB-X 1.0还支持如下函数：

函数名	描述
RAND()	Return a random floating-point value
RELEASE_ALL_LOCKS()	Releases all current named locks
RELEASE_LOCK()	Releases the named lock
SLEEP()	Sleep for a number of seconds
UUID()	Return a Universal Unique Identifier (UUID)
UUID_SHORT()	Return an integer-valued universal identifier
VALUES()	Defines the values to be used during an INSERT
ANY_VALUE()	Suppress ONLY_FULL_GROUP_BY value rejection
DEFAULT()	Return the default value for a table column
GET_LOCK()	Get a named lock
INET_ATON()	Return the numeric value of an IP address
INET_NTOA()	Return the IP address from a numeric value
INET6_ATON()	Return the numeric value of an IPv6 address
INET6_NTOA()	Return the IPv6 address from a numeric value
IS_FREE_LOCK()	Whether the named lock is free
IS_IPV4()	Whether argument is an IPv4 address
IS_IPV4_COMPAT()	Whether argument is an IPv4-compatible address
IS_IPV4_MAPPED()	Whether argument is an IPv4-mapped address
IS_IPV6()	Whether argument is an IPv6 address
IS_USED_LOCK()	Whether the named lock is in use; return connection identifier if true
MASTER_POS_WAIT()	Block until the slave has read and applied all updates up to the specified position
NAME_CONST()	Causes the column to have the given name

12.14. 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 1.0新增支持通过Grouping Sets、Rollup和Cube扩展来实现这一目的。此外，PolarDB-X 1.0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。

前提条件

PolarDB-X 1.0实例版本需为5.4.10及以上。关于如何查看实例版本，请参见查看实例版本。

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(`i
d`);
```

  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 1.0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 1.0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 1.0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 1.0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 1.0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 1.0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 1.0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Bit map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 |
+------+------+------+----------+------+------+------+------------+
```

13.运算符

13.1. 逻辑运算符

本文介绍了PolarDB-X 1.0支持的逻辑运算符。

PolarDB-X 1.0支持如下逻辑运算符：

运算符	描述
AND, &&	Logical AND
NOT, !	Negates value
||, OR	Logical OR
XOR	Logical XOR

13.2. 算术运算符

本文介绍了PolarDB-X 1.0支持的算术运算符。

PolarDB-X 1.0支持如下算术运算符：

操作符	描述
DIV	Integer division
/	Division operator
-	Minus operator
%, MOD	Modulo operator
+	Addition operator
*	Multiplication operator
-	Change the sign of the argument

13.3. 比较运算符

本文介绍了PolarDB-X 1.0支持的比较运算符。

PolarDB-X 1.0支持如下比较运算符：

运算符	描述
BETWEEN ... AND ...	Check whether a value is within a range of values

运算符	描述
=	Equal operator
<=>	NULL-safe equal to operator
>	Greater than operator
>=	Greater than or equal operator
IS	Test a value against a boolean
IS NOT	Test a value against a boolean
IS NOT NULL	NOT NULL value test
IS NULL	NULL value test
<	Less than operator
<=	Less than or equal operator
LIKE	Simple pattern matching
NOT BETWEEN ... AND ...	Check whether a value is not within a range of values
!=, <>	Not equal operator
NOT LIKE	Negation of simple pattern matching

13.4. 位运算符

本文介绍了PolarDB-X 1.0支持的位运算符。

PolarDB-X 1.0支持如下位运算符：

运算符	描述
&	Bitwise AND
~	Bitwise inversion
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift

13.5. 赋值运算符

本文介绍了PolarDB-X 1.0支持及不支持的赋值运算符。

PolarDB-X 1.0支持 '=' 赋值运算符，一般在UPDATE的SET部分出现。

PolarDB-X 1.0暂不支持 ':=' 赋值运算符。

13.6. 运算符优先级

本文介绍了PolarDB-X 1.0中运算符的优先级。

PolarDB-X 1.0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, ||
1	=（赋值运算符）

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 1.0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。

14.数据类型

14.1. 数据类型

本文介绍了PolarDB-X 1.0支持的数据类型。

PolarDB-X 1.0支持四种主要数据类型：

- 数值类型
- 字符串类型
- 日期时间类型
- JSON 数据类型

不支持的数据类型：空间数据类型

关于数据类型的详细信息可参考MySQL 数据类型文档。

14.2. 数值类型

本文介绍了PolarDB-X 1.0支持的数值类型。

数值类型按精确度可以划分为两类：

- 精确数据类型
  - 整数数据类型TINYINT，SAMLLINT，MEDIUMINT，INTEGER，BIGINT
  - 定点数据类型DECIMAL，NUMERIC
- 近似数据类型FLOAT，REAL，DOUBLE PRECISION

整体与MySQL保持一致，详细信息可参考MySQL 整数类型文档。

14.3. 字符串类型

本文介绍了PolarDB-X 1.0支持的字符串类型。

PolarDB-X 1.0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。

14.4. 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 1.0支持的collation类型。

> **说明** 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	latin1_swedish_ci
	latin1_german1_ci
	latin1_danish_ci
	latin1_bin
	latin1_general_ci
	latin1_general_cs
	latin1_spanish_ci
gbk	gbk_chinese_ci
	gbk_bin

14.5. 日期和时间类型

本文介绍了PolarDB-X 1.0支持的日期时间类型。

PolarDB-X 1.0支持如下日期时间类型：

- DATE
- DATETIME
- TIMESTAMP

- TIME
- YEAR

> **说明** 与MySQL不同，PolarDB-X 1.0支持的TIME类型范围为'00:00:00' ~ '23:59:59'。

详细信息可参考MySQL日期时间类型文档。

15.实用SQL语句

15.1. TRACE

本文介绍了TRACE语句的用法。

TRACE语句用于查看具体SQL的执行情况。TRACE [SQL]和SHOW TRACE要结合使用。

> **说明** TRACE SQL和EXPLAIN SQL的区别在于TRACE SQL会实际执行该语句。

示例

查看select 1这条语句的执行情况。

```
mysql> trace select 1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.03 sec)
mysql> show trace;
+------+----------+----------------+-----------------------------------+---------------+---
----------------------+------+-----------+---------+
| ID   | TYPE     | GROUP_NAME     | DBKEY_NAME                        | TIME_COST(MS) | CO
NNECTION_TIME_COST(MS) | ROWS | STATEMENT | PARAMS |
+------+----------+----------------+-----------------------------------+---------------+---
----------------------+------+-----------+---------+
|    0 | Optimize | DRDS           | DRDS                              | 3             | 0.
00                     |    0 | select 1  | NULL    |
|    1 | Query    | TDDL5_00_GROUP | db218249098_sqa_zmf_tddl5_00_3309 | 7             | 0.
15                     |    1 | select 1  | NULL    |
+------+----------+----------------+-----------------------------------+---------------+---
----------------------+------+-----------+---------+
2 rows in set (0.01 sec)
```

15.2. 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 1.0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。

前提条件

PolarDB-X 1.0实例版本需为5.4.12及以上。关于如何查看实例版本，请参见查看实例版本。

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 = value
3 ] [ ,FAST_MATCH = { 0 , 1 }]
```

参数说明

参数		是否必选	说明
	`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
限流规则匹配参数	`[ filter_options ]`	可选	过滤条件，支持如下两种条件： ○ 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。

参数		是否必选	说明
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 1.0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 1.0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                       | TIM
  E | STATE | INFO                  | SQL_TEMPLATE_ID |
  +----+---------------+-----------------+----------+-------------------------------+----
  --+-------+-----------------------+-----------------+
  |  2 | polardbx_root | ***.*.*.*:62787 | polardbx | Query                         |
  0 |       | show full processlist | NULL            |
  |  1 | polardbx_root | ***.*.*.*:62775 | polardbx | Query(Waiting-selectrulereal) |   1
  2 |       | select 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 c
  oncurrency 0 of ccl 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 c
  oncurrency 0 of ccl 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。

参数	说明
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。
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
```

15.3. 跨Schema

PolarDB-X 1.0实例中通常有多个Schema，PolarDB-X 1.0支持通过SQL语法进行跨Schema的查询，效果与MySQL的跨Schema查询类似。

注意事项

- 若要使用跨Schema的查询语法，需要在写SQL语句时为具体的TableName增加其对应的SchemaName的前缀（如 `xxx_tbl` -> `yyy_db` . `xxx_tbl` ），从而指定表 `xxx_tbl` 所属的Schema。这与MySQL的跨Schema查询的语法完全兼容。
- 不支持CREATE、ALTER、DROP SEQUENCE语句的跨Schema用法。
- PolarDB-X 1.0实例需为5.3.8-15517870或以上版本。
- 使用跨Schema查询前，请先完成Schema的访问授权，授权相关的语法请参见账号和权限系统。

基本概念

- Schema：PolarDB-X 1.0实例中的一个数据库，它可能是一个带水平拆分的库，也可能是没做水平拆分的库。
- SchemaName：PolarDB-X 1.0数据库库名，同一个实例内的数据库库名具有唯一性。
- Table: PolarDB-X 1.0数据库中的一张表，它可能是一张带水平拆分的表，也可能是没做水平拆分的表。
- TableName: PolarDB-X 1.0数据库中一张表的表名，同一个数据库内的表名具有唯一性。

使用示例

假设在某一PolarDB-X 1.0实例中，您创建了3个不同的数据库，每个数据库内分别有一张表，且每张表均有各自对应的Sequence，各个数据库、表和Sequence的详情如下所示：

SchemaName	TableName	Sequence
`new_db`	`new_tbl`	`AUTO_SEQ_new_tbl`
`trade_db`	`trade_tbl`	`AUTO_SEQ_trade_tbl`
`user_db`	`user_tbl`	`AUTO_SEQ_user_tbl`

假设您当前登录控制台使用的SchemaName为 `trade_db` ，则您可以使用如下SQL语句进行跨Schema查询：

- 跨Schema的使用示例（SELECT）

  您可以使用如下SQL在 `trade_tbl` 与 `user_tbl` 数据库间进行跨Schema关联聚合查询：

  ```
  SELECT COUNT(DISTINCT u.user_id)
  FROM `trade_tbl` AS t
  INNER JOIN `user_db`.`user_tbl` AS u ON t.user_id=u.user_id
  WHERE u.user_id >= 10000
  GROUP BY t.title
  ```

- 跨Schema的使用示例（INSERT）

  您可以使用如下SQL语句，将数据插入到 `new_db` 库中 `new_tbl` 表中：

  ```
  INSERT INTO `new_db`.`new_tbl` (user_id, title) VALUES ( null, 'test' );
  ```

- 跨Schema的使用示例3（分布式事务）

  在分布式事务中，您可以使用如下SQL语句，分别对表 `new_tbl` 与表 `user_tbl` 进行更新或删除，并合并提交：

```
SET AUTOCOMMIT=off;
SET drds_transaction_policy = 'XA';
UPDATE `new_db`.`new_tbl` SET name='abc' WHERE use_id=1;
DELETE FROM `user_db`.`user_tbl` WHERE user_id=2;
COMMIT;
```

- 跨Schema的使用示例（SEQUENCE）

  若要显式使用Sequence进行跨Schema的INSERT操作，则您需要先在显式Sequence名字前加上SchemaName的前缀（如 `xxx_seq` -> `yyy_db` . `xxx_seq`

```
/* 该 SQL将使用`new_db`库的`AUTO_SEQ_new_tbl`作为Sequence并进行插入操作 */
INSERT INTO `new_db`.`new_tbl` (id, name) values ( null, 'test_seq' );
```

```
/* 该 SQL将使用`new_db`库的`AUTO_SEQ_new_tbl`作为Sequence进行插入操作，注意这里的Sequence指定了SchemaName */
INSERT INTO `new_db`.`new_tbl` (id, name) values ( `new_db`.AUTO_SEQ_new_tbl.nextval, 'test_seq' );
```

- 跨Schema的使用示例（SHOW CREATE TABLE）

  您可以使用如下SQL语句在当前Schema中去查询其它Schema（如 `new_db`

```
SHOW CREATE TABLE `new_db`.`new_tbl`;
```

支持跨Schema查询的SQL类型

- SELECT
- INSERT
- REPLACE
- UPDATE
- DELETE
- 分布式事务
- Sequence
- DAL
- USE

15.4. 多语句

本文将介绍PolarDB-X 1.0多语句的相关信息。

PolarDB-X 1.0支持多语句（即用英文分号（;）分割的SQL语句）功能。

```
mysql> SELECT * FROM t1; SELECT * FROM t2; SELECT NOW().
```

> **说明**
>
> - 上述语句中通过修改MySQL客户端的--delimiter参数将SQL分隔符设置为英文句号（.），避免客户端将SQL按照英文分号（;）进行拆分。
> - PolarDB-X 1.0执行以上SQL时，会按照英文分号（;）对语句进行分割，并从左至右按顺序依次执行。

15.5. EXPLAIN和执行计划

本文着重介绍PolarDB-X 1.0执行计划中各个操作符的含义，以便用户通过查询计划了解SQL执行流程，从而有针对性的进行SQL调优。

执行计划介绍

与多数数据库系统类似，PolarDB-X 1.0在处理SQL时，会通过优化器生成执行计划，该执行计划由关系操作符构成一个树形结构，反映PolarDB-X 1.0如何执行SQL语句。不同的是，PolarDB-X 1.0本身不存储数据，更侧重考虑分布式环境中的网络IO开销，将运算下推到各个分库（如RDS/MySQL）执行，从而提升SQL执行效率。用户可通过EXPLAIN命令查看SQL的执行计划。

文中示例均基于如下表结构：

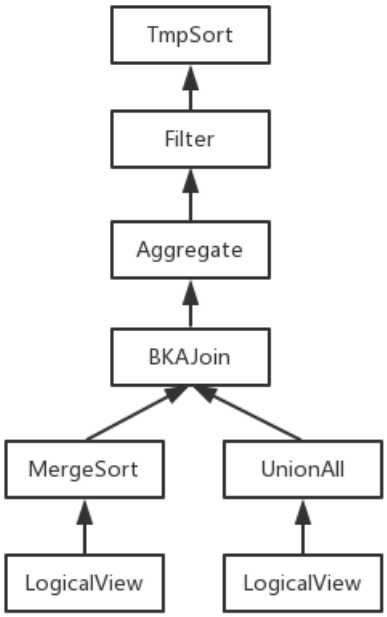
```
CREATE TABLE `sbtest1` (
  `id`  INT(10) UNSIGNED NOT NULL,
  `k`   INT(10) UNSIGNED NOT NULL DEFAULT '0',
  `c`   CHAR(120)        NOT NULL DEFAULT '',
  `pad` CHAR(60)         NOT NULL DEFAULT '',
  KEY `xid` (`id`),
  KEY `k_1` (`k`)
) dbpartition BY HASH (`id`) tbpartition BY HASH (`id`) tbpartitions 4
```

如下示例展示了PolarDB-X 1.0执行计划的树形结构。

```
mysql> explain 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;
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------+
| TmpSort(sort="cnt ASC", offset=?2, fetch=?3)
|
|   Filter(condition="cnt > ?1")
|
|     Aggregate(group="k", cnt="COUNT()")
|
|       BKAJoin(id="id", k="k", c="c", pad="pad", id0="id0", k0="k0", c0="c0", pad0="pad0",
condition="id = k", type="inner")                       |
|         MergeSort(sort="k ASC")
|
|           LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT
* FROM `sbtest1` WHERE (`id` > ?) ORDER BY `k`")        |
|         UnionAll(concurrent=true)
|
|           LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT
* FROM `sbtest1` WHERE ((`k` > ?) AND (`k` IN ('?')))") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-------------------------------------------------------+
9 rows in set (0.01 sec)
```

PolarDB-X 1.0 EXPLAIN的结果总体分为两部分：执行计划和其他信息。

- 执行计划：以缩进形式表示操作符之间的 "父-子" 关系。示例中，Filter是TmpSort的子操作符，同时是Aggregate的父操作符。从真正执行的角度看，每个操作符均从其子操作符中获取数据，经当前操作符处理，输出给其父操作符。为方便理解，将以上执行计划转换为更加直观的树形结构：



- 其他信息：除执行计划外，EXPLAIN结果中还会有一些额外信息，目前仅有一项 `HitCache` 。需要说明的是，PolarDB-X 1.0会默认开启PlanCache功能， `HitCache` 表示当前SQL是否命中PlanCache。 开启PlanCache后，PolarDB-X 1.0会对SQL做参数化处理，参数化会将SQL中的大部分常量用 `?` 替换，并构建一个参数列表。在执行计划中的体现就是，LogicalView的SQL中会有 `?` ，在部分操作符中会有类似 `?2` 的字样，这里的 `2` 表示其在参数列表中的下标，后续会结合具体的例子进一步阐述。

EXPLAIN语法

EXPLAIN用于查看SQL语句的执行计划，语法如下：

```
EXPLAIN
{LOGICALVIEW | LOGIC | SIMPLE | DETAIL | EXECUTE | PHYSICAL | OPTIMIZER | SHARDING
 | COST | ANALYZE | BASELINE | JSON_PLAN | ADVISOR}
 {SELECT statement | DELETE statement | INSERT statement | REPLACE statement| UPDATE statem
ent}
```

操作符介绍

LogicalView

LogicalView是从底层数据源获取数据的操作符。从数据库的角度来看，使用 `TableScan` 命名更符合常规，但考虑到PolarDB-X 1.0本身不存储数据，而是通过SQL从底层数据源获取，因此，该操作符中会记录下推的SQL语句和数据源信息，这更像一个 "视图"。该 "视图" 中的SQL，通过优化器的下推，可能包含多种操作，如投影、过滤、聚合、排序、连接和子查询等。

以下通过示例说明EXPLAIN中LogicalView的输出信息及其含义：

```
mysql> explain select * From sbtest1 where id > 1000;
+-------------------------------------------------------------------------------------------------
----------------------------+
| LOGICAL PLAN
|
+-------------------------------------------------------------------------------------------------
----------------------------+
| UnionAll(concurrent=true)
|
|   LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FROM
`sbtest1` WHERE (`id` > ?)") |
| HitCache:false
|
+-------------------------------------------------------------------------------------------------
----------------------------+
3 rows in set (0.00 sec)
```

LogicalView的信息由三部分构成：

- tables：底层数据源对应的表名，以 `.` 分割，其前是分库对应的编号，其后是表名及其编号，对于连续的编号，会做简写，如 `[000-127]` ，表示表名编号从 `000` 到 `127` 的所有表。
- shardCount：需要访问的分表总数，该示例中会访问从 `000` 到 `127` 共128张分表。
- sql：下发至底层数据源的SQL模版。这里显示的并非真正下发的SQL语句，PolarDB-X 1.0在执行时会将表名替换为物理表名；另外，SQL中的常量 `10` 被 `?` 替换，这是因为PolarDB-X 1.0默认开启了PlanCache功能，对SQL做了参数化处理。

UnionAll

UnionAll是 `UNION ALL` 对应的操作符，该操作符通常有多个输入，表示将多个输入的数据UNION在一起。以上示例中，LogicalView之上的UnionAll表示将所有分表中的数据进行UNION。

UnionAll中的 `concurrent` 表示是否并行执行其子操作符，默认为true。

UnionDistinct

与UnionAll类似，UnionDistinct是 `UNION DISTINCT` 对应的操作符。示例如下：

```
mysql> explain select * From sbtest1 where id > 1000 union distinct select * From sbtest1 w
here id < 200;
+-------------------------------------------------------------------------------------------
------------------------------+
| LOGICAL PLAN
|
+-------------------------------------------------------------------------------------------
------------------------------+
| UnionDistinct(concurrent=true)
|
|   UnionAll(concurrent=true)
|
|     LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FRO
M `sbtest1` WHERE (`id` > ?)") |
|   UnionAll(concurrent=true)
|
|     LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FRO
M `sbtest1` WHERE (`id` < ?)") |
| HitCache:false
|
+-------------------------------------------------------------------------------------------
------------------------------+
6 rows in set (0.02 sec)
```

MergeSort

MergeSort，归并排序操作符，通常有多个子操作符。PolarDB-X 1.0中实现了两种排序：基于有序数据的归并排序和对无序数据的内存排序。示例如下：

```
mysql> explain select *from sbtest1 where id > 1000 order by id limit 5,10;
+-------------------------------------------------------------------------------------------
-----------------------------------------------------+
| LOGICAL PLAN
|
+-------------------------------------------------------------------------------------------
-----------------------------------------------------+
| MergeSort(sort="id ASC", offset=?1, fetch=?2)
|
|   LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FROM
`sbtest1` WHERE (`id` > ?) ORDER BY `id` LIMIT (? + ?)") |
| HitCache:false
|
+-------------------------------------------------------------------------------------------
-----------------------------------------------------+
3 rows in set (0.00 sec)
```

MergeSort操作符包含三部分内容：

- sort：表示排序字段以及排列顺序，`id ASC`表示按照`id`字段递增排序，`DESC`表示递减排序。
- offset：表示获取结果集时的偏移量，同样由于对SQL做了参数化，示例中的`offst`表示为`?1`，其

中 `?` 表示这是一个动态参数，其后的数字对应参数列表的下标。示例中SQL对应的参数为 `[1000, 5, 10]` ，因此， `?1` 实际对应的值为 `5` 。

- fetch：表示最多返回的数据行数。与 `offset` 类似，同样是参数化的表示，实际对应的值为 `10` 。

Aggregate

Aggregate是聚合操作符，通常包含两部分内容：Group By字段和聚合函数。示例如下：

```
mysql> explain select k, count(*) from sbtest1 where id > 1000 group by k;
+-------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| LOGICAL PLAN
|
+-------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| Aggregate(group="k", count(*)="SUM(count(*))")
|
|   MergeSort(sort="k ASC")
|
|     LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT `k`,
COUNT(*) AS `count(*)` FROM `sbtest1` WHERE (`id` > ?) GROUP BY `k` ORDER BY `k`") |
| HitCache:true
|
+-------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
4 rows in set (0.00 sec)
```

Aggregate 包含两部分内容：

- group：表示GROUP BY字段，示例中为 `k` 。
- 聚合函数： `=` 前为聚合函数对应的输出列名，其后为对应的计算方法。示例中的 `count(*)="SUM(count(*))"` ，第一个 `count(*)` 对应输出的列名，随后的 `SUM(count(*))` 表示对其输入数据中的 `count(*)` 列进行 `SUM` 运算得到最终的 `count(*)` 。

由此可见，PolarDB-X 1.0将聚合操作分为两部分，首先将聚合操作下推至底层数据源做局部聚合，最终在PolarDB-X 1.0层面对局部聚合的结果做全局聚合。另外，PolarDB-X 1.0的最终聚合是基于排序做的，因此，会在优化器阶段为其添加一个 `Sort` 子操作符，而 `Sort` 操作符又进一步通过下推Sort转换为 `MergeSort` 。

如下为 `AVG` 聚合函数的例子：

```
mysql> explain select k, avg(id) avg_id from sbtest1 where id > 1000 group by k;
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
----------------------------+
| LOGICAL PLAN|
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
----------------------------+
| Project(k="k", avg_id="sum_pushed_sum / sum_pushed_count")|
|   Aggregate(group="k", sum_pushed_sum="SUM(pushed_sum)", sum_pushed_count="SUM(pushed_cou
nt)")|
|     MergeSort(sort="k ASC")|
|       LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT `k`
, SUM(`id`) AS `pushed_sum`, COUNT(`id`) AS `pushed_count` FROM `sbtest1` WHERE (`id` > ?)
GROUP BY `k` ORDER BY `k`")|
| HitCache:false|
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
----------------------------+
5 rows in set (0.01 sec)
```

PolarDB-X 1.0会将 `AVG` 聚合函数转换为 `SUM / COUNT` ，再分别根据 `SUM` 和 `COUNT` 的下推规则，将其转换为局部聚合和全局聚合。您可自行尝试了解其他聚合函数的执行计划。

> **说明** PolarDB-X 1.0会将 `DISTINCT` 操作转换为 `GROUP` 操作，示例如下：

```
mysql> explain select distinct k from sbtest1 where id > 1000;
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------+
| Aggregate(group="k")
|
|   MergeSort(sort="k ASC")
|
|     LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT `
k` FROM `sbtest1` WHERE (`id` > ?) GROUP BY `k` ORDER BY `k`") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------+
4 rows in set (0.02 sec)
```

TmpSort

TmpSort，表示在内存中对数据进行排序。与MergeSort的区别在于，MergeSort可以有多个子操作符，且每个子操作符返回的数据都已经排序。TmpSort仅有一个子操作符。

TmpSort对应的查询计划信息与MergeSort一致，请参考MergeSort。

Project

Project表示投影操作，即从输入数据中选择部分列输出，或者对某些列进行转换（通过函数或者表达式计算）后输出，当然，也可以包含常量。以上 `AVG` 的示例中，最顶层就是一个 `Project` ，其输出 `k` 和 `sum_pushed_sum / sum_pushed_count` ，后者对应的列名为 `avg_id` 。

```
mysql> explain select '你好', DRDS', 1 / 2, CURTIME();
+-----------------------------------------------------------------------------------------+
| LOGICAL PLAN                                                                            |
+-----------------------------------------------------------------------------------------+
| Project(你好, DRDS="_UTF-16'你好, DRDS'", 1 / 2="1 / 2", CURTIME()="CURTIME()") |
|                                                                                         |
| HitCache:false                                                                          |
+-----------------------------------------------------------------------------------------+
3 rows in set (0.00 sec)
```

可见，Project的计划中包括每列的列名及其对应的列、值、函数或者表达式。

Filter

Filter表示过滤操作，其中包含一些过滤条件。该操作符对输入数据进行过滤，若满足条件，则输出，否则丢弃。如下是一个较复杂的例子，包含了以上介绍的大部分操作符。

```
mysql> explain select k, avg(id) avg_id from sbtest1 where id > 1000 group by k having avg_
id > 1300;
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------+
| LOGICAL PLAN  |
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------+
| Filter(condition="avg_id > ?1")  |
|   Project(k="k", avg_id="sum_pushed_sum / sum_pushed_count")  |
|     Aggregate(group="k", sum_pushed_sum="SUM(pushed_sum)", sum_pushed_count="SUM(pushed_c
ount)")  |
|       MergeSort(sort="k ASC")  |
|         LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT `
k`, SUM(`id`) AS `pushed_sum`, COUNT(`id`) AS `pushed_count` FROM `sbtest1` WHERE (`id` > ?
) GROUP BY `k` ORDER BY `k`") |
| HitCache:false  |
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------+
6 rows in set (0.01 sec)
```

在以上 `AVG` 示例的SQL基础上添加了 `having avg_id > 1300` ，执行计划最上层添加了一个Filter操作符，用于过滤所有满足 `avg_id > 1300` 的数据。

有读者可能会问，WHERE中的条件为什么没有对应的Filter操作符呢？在PolarDB-X 1.0优化器的某个阶段，WHERE条件的Filter操作符的确是存在的，只是最终将其下推到了LogiacalView中，因此可以在LogicalView的 `sql` 中看到 `id > 1000` 。

NlJoin

NlJoin，表示NestLoop Join操作符，即使用NestLoop方法进行两表Join。PolarDB-X 1.0中实现了两种JOIN策略：NlJoin和BKAJoin，后者表示Batched Key Access Join，批量键值查询，会从左表取一批数据，构建一个IN条件拼接在访问右表的SQL中，从右表一次获取一批数据。

```
mysql> explain select a.* from sbtest1 a, sbtest1 b where a.id = b.k and a.id > 1000;
+-----------------------------------------------------------------------------------------------------
----------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------------
----------------------------------+
| Project(id="id", k="k", c="c", pad="pad")
|
|   NlJoin(id="id", k="k", c="c", pad="pad", k0="k0", condition="id = k", type="inner")
|
|     UnionAll(concurrent=true)
|
|       LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * F
ROM `sbtest1` WHERE (`id` > ?)")  |
|     UnionAll(concurrent=true)
|
|       LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT `k`
FROM `sbtest1` WHERE (`k` > ?)") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------------
----------------------------------+
7 rows in set (0.03 sec)
```

NlJOIN的计划包括三部分内容：

- 输出列信息：输出的列名，示例中的JOIN会输出5列 `id="id", k="k", c="c", pad="pad", k0="k0"` 。
- contition：连接条件，示例中连接条件为 `id = k` 。
- type：连接类型，示例中是INNER JOIN，因此其连接类型为 `inner` 。

BKAJoin

BKAJoin（Batched Key Access Join），表示通过批量键值查询的方式进行JOIN，即从左表取一批数据，构建一个IN条件拼接在访问右表的SQL中，从右表一次获取一批数据进行JOIN。

```
mysql> explain select a.* from sbtest1 a, sbtest1 b where a.id = b.k order by a.id;
+-----------------------------------------------------------------------------------------------
-------------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-------------------------------------+
| Project(id="id", k="k", c="c", pad="pad")
|
|   BKAJoin(id="id", k="k", c="c", pad="pad", id0="id0", k0="k0", c0="c0", pad0="pad0", con
dition="id = k", type="inner")       |
|     MergeSort(sort="id ASC")
|
|       LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * F
ROM `sbtest1` ORDER BY `id`")        |
|     UnionAll(concurrent=true)
|
|       LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * F
ROM `sbtest1` WHERE (`k` IN ('?'))") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-------------------------------------+
7 rows in set (0.01 sec)
```

BKAJoin的计划内容与NlJoin相同，这两个操作符命名不同，旨在告知执行器以何种方法执行JOIN操作。另外，以上执行计划中右表的LogicalView中 `'k' IN ('?')` 是优化器构建出来的对右表的IN查询模板。

LogicalModifyView

如上文介绍，LogicalView表示从底层数据源获取数据的操作符，与之对应的，LogicalModifyView表示对底层数据源的修改操作符，其中也会记录一个SQL语句，该SQL可能是INSERT、UPDATE或者DELETE。

```
mysql> explain update sbtest1 set c='Hello, DRDS' where id > 1000;
+-----------------------------------------------------------------------------------------------
-------------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-------------------------------------+
| LogicalModifyView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="UPDATE `sb
test1` SET `c` = ? WHERE (`id` > ?)") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-------------------------------------+
2 rows in set (0.03 sec)
mysql> explain delete from sbtest1 where id > 1000;
+-----------------------------------------------------------------------------------------------
-------------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-------------------------------+
| LogicalModifyView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="DELETE FRO
M `sbtest1` WHERE (`id` > ?)") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-------------------------------+
2 rows in set (0.03 sec)
```

LogicalModifyView查询计划的内容与LogicalView类似，包括下发的物理分表，分表数以及SQL模版。同样，由于开启了PlanCache，对SQL做了参数化处理，SQL模版中的常量会用 `?` 替换。

PhyTableOperation

PhyTableOperation表示对某个物理分表执行一个操作。该操作符目前仅用于INSERT INTO ... VALUES ...。

```
mysql> explain insert into sbtest1 values(1, 1, '1', '1'),(2, 2, '2', '2');
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------+
| LOGICAL PLAN
|
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------+
| PhyTableOperation(tables="SYSBENCH_CORONADB_1526954857179TGMMSYSBENCH_CORONADB_VGOC_0000_
RDS.[sbtest1_001]", sql="INSERT INTO ? (`id`, `k`, `c`, `pad`) VALUES(?, ?, ?, ?)", params=
"`sbtest1_001`,1,1,1,1") |
| PhyTableOperation(tables="SYSBENCH_CORONADB_1526954857179TGMMSYSBENCH_CORONADB_VGOC_0000_
RDS.[sbtest1_002]", sql="INSERT INTO ? (`id`, `k`, `c`, `pad`) VALUES(?, ?, ?, ?)", params=
"`sbtest1_002`,2,2,2,2") |
|
|
| HitCache:false
|
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------+
4 rows in set (0.00 sec)
```

示例中，INSERT插入两行数据，每行数据对应一个PhyTableOperation操作符，PhyTableOperation操作符的内容包括三部分：

- tables：物理表名，仅有唯一一个物理表名。
- sql：SQL模版，该SQL模版中表名和常量均被参数化，用 `?` 替换，对应的参数在随后的params中给出。
- params：SQL模版对应的参数，包括表名和常量。

其他信息

Hit Cache

PolarDB-X 1.0会默认开PlanCache 功能，Hit Cache用于告知用户当前查询是否命中PlanCache。如下示例，第一次运行Hit Cache为false，第二次运行为true。

```
mysql> explain select * From sbtest1 where id > 1000;
+-----------------------------------------------------------------------------------------------
-----------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-----------------------------+
| UnionAll(concurrent=true)
|
|   LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FROM
`sbtest1` WHERE (`id` > ?)") |
| HitCache:false
|
+-----------------------------------------------------------------------------------------------
-----------------------------+
3 rows in set (0.01 sec)
mysql> explain select * From sbtest1 where id > 1000;
+-----------------------------------------------------------------------------------------------
-----------------------------+
| LOGICAL PLAN
|
+-----------------------------------------------------------------------------------------------
-----------------------------+
| UnionAll(concurrent=true)
|
|   LogicalView(tables="[0000-0031].sbtest1_[000-127]", shardCount=128, sql="SELECT * FROM
`sbtest1` WHERE (`id` > ?)") |
| HitCache:true
|
+-----------------------------------------------------------------------------------------------
-----------------------------+
3 rows in set (0.00 sec)
```

16.错误代码

本文档列出了PolarDB-X 1.0返回的常见错误码及解决方法。

TDDL-4006 ERR_TABLE_NOT_EXIST

数据表不存在。

示例：

```
ERR-CODE: [TDDL-4006][ERR_TABLE_NOT_EXIST] Table '*****' doesn't exist.
```

该错误表示PolarDB-X 1.0数据表不存在，或者由于未知原因PolarDB-X 1.0无法加载数据表的元数据信息。

如果出现该错误，请提交工单。

TDDL-4007 ERR_CANNOT_FETCH_TABLE_META

PolarDB-X 1.0无法加载数据表的元数据信息。

示例：

```
ERR-CODE: [TDDL-4007][ERR_CANNOT_FETCH_TABLE_META] Table '*****' metadata
cannot be fetched because Table '*****.*****' doesn't exist.
```

该错误表示PolarDB-X 1.0尝试读取数据表的元数据信息失败。可能的错误原因如下：

- 未创建数据表。
- 分库上的表名被人为删除或改名。
- 无法连接RDS MySQL。

出现该错误时，首先请检查表名是否存在，然后确认PolarDB-X 1.0后端所有RDS MySQL状态是否正常。

如果确定表名被人为删除或改名，可以通过RDS MySQL的数据恢复功能修复。如果仍无法修复，请提交工单。

TDDL-4100 ERR_ATOM_NOT_AVALILABLE

PolarDB-X 1.0后端RDS MySQL暂时不可用。

示例：

```
ERR-CODE: [TDDL-4100][ERR_ATOM_NOT_AVALILABLE] Atom : ***** isNotAvailable
```

如果PolarDB-X 1.0探测到后端某个RDS MySQL实例状态异常，会临时阻止访问该实例并提示TDDL-4100错误。当遇到该错误，请检查PolarDB-X 1.0后端所有RDS MySQL是否异常，并尝试进行恢复。

当RDS MySQL实例从异常状态恢复后，PolarDB-X 1.0将自动解除不可用状态，恢复应用正常访问。

TDDL-4101 ERR_ATOM_GET_CONNECTION_FAILED_UNKNOWN_REASON

未知原因的PolarDB-X 1.0后端连接获取失败。

示例：

```
ERR-CODE: [TDDL-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.
```

PolarDB-X 1.0在处理请求时会向后端RDS MySQL异步创建连接。如果无法在等待时间内完成RDS MySQL连接创建，而异步任务又尚未返回错误原因，PolarDB-X 1.0会向应用返回TDDL-4101错误。

该错误通常是由后端RDS MySQL异常导致的。如果排除RDS MySQL异常后仍然出现该错误，请提交工单。

TDDL-4102 ERR_ATOM_GET_CONNECTION_FAILED_KNOWN_REASON

已知原因的PolarDB-X 1.0后端连接获取失败。

示例：

```
ERR-CODE: [TDDL-4102][ERR_ATOM_GET_CONNECTION_FAILED_KNOWN_REASON] Get
connection for db '*****' failed because wait millis 5000, active 0,
maxActive 5
```

PolarDB-X 1.0获取后端RDS MySQL连接时出错，错误原因已经在 `ERR-CODE` 消息中给出。

常见PolarDB-X 1.0后端连接失败的原因如下：

- 后端RDS MySQL连接数已满；
- RDS MySQL连接超时；
- RDS MySQL拒绝连接。

如果排除后端RDS MySQL问题后仍然出现该错误，请提交工单。

TDDL-4103 ERR_ATOM_CONNECTION_POOL_FULL

PolarDB-X 1.0后端RDS MySQL连接池已满。

示例：

```
ERR-CODE: [TDDL-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.
```

该错误表示PolarDB-X 1.0后端连接池已满。导致TDDL-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或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RDS性能分析报告与DBA建议优化业务SQL语句。
- 使用PolarDB-X 1.0读写分离将跨库查询转发至读库处理。
- 升级更高规格的RDS实例，提升后端处理能力。
- 请提交工单调整PolarDB-X 1.0后端连接数。

TDDL-4104 ERR_ATOM_CREATE_CONNECTION_TOO_SLOW

PolarDB-X 1.0后端RDS MySQL连接创建太慢。

示例：

```
ERR-CODE: [TDDL-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.
```

PolarDB-X 1.0向后端RDS MySQL异步创建连接时，如果短时间内创建大量连接，或者RDS MySQL建立连接速度太慢，会出现等待超时。

该问题通常是由后端RDS MySQL压力过大或异常导致的，建议使用PolarDB-X 1.0读写分离，或者升级更高规格的RDS实例，减轻后端处理压力。

如果排除RDS MySQL问题后仍然出现该错误，请提交工单。

如果问题是由于短时间内创建大量连接导致，请提交工单调整PolarDB-X 1.0最小连接数。

TDDL-4105 ERR_ATOM_ACCESS_DENIED

PolarDB-X 1.0后端RDS MySQL拒绝创建连接。

示例：

```
ERR-CODE: [TDDL-4105][ERR_ATOM_ACCESS_DENIED] DB '*****' Access denied for
user '*****'@'*****'. AppName:*****, Env:*****, UnitName:null. Please
contact DBA to check.
```

该错误表明PolarDB-X 1.0通过用户名和密码连接RDS MySQL时被拒绝访问。

如果不小心在后端RDS MySQL上修改了由PolarDB-X 1.0自动创建的用户名或密码，就会导致PolarDB-X 1.0提示TDDL-4105错误。修复该问题需要手工订正PolarDB-X 1.0的用户名或密码，请提交工单进行修改。

此外，后端RDS实例欠费或到期后会拒绝所有访问请求，也会导致PolarDB-X 1.0产生TDDL-4105错误。如果遇到这种情况，请及时给RDS实例续费。

TDDL-4106 ERR_ATOM_DB_DOWN

PolarDB-X 1.0后端RDS MySQL无法连接。

示例：

```
ERR-CODE: [TDDL-4106][ERR_ATOM_DB_DOWN] DB '*****' cannot be connected.
AppName:*****, Env:*****, UnitName:null. It seems a very real possibility
that this DB IS DOWN. Please contact DBA to check.
```

该错误表示PolarDB-X 1.0向后端RDS MySQL创建连接超时或没有响应。遇到该错误的通常原因是RDS MySQL故障，请提交工单。

TDDL-4108 ERR_VARIABLE_CAN_NOT_SET_TO_NULL_FOR_NOW

变量（variable）不允许被设置为 `NULL` 。

示例：

```
ERR-CODE: [TDDL-4108][ERR_VARIABLE_CAN_NOT_SET_TO_NULL_FOR_NOW] System
variable ***** can''t set to null for now;
```

部分MySQL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 1.0会提示TDDL-4108错误。

如果遇到该错误，请检查传递的变量（variable）值，并参考MySQL官方文档改正，详情请参见 MySQL官网。

TDDL-4200 ERR_GROUP_NOT_AVALILABLE

PolarDB-X 1.0分库暂时不可用。

示例：

```
ERR-CODE: [TDDL-4200][ERR_GROUP_NOT_AVALILABLE] The TDDL Group ***** is
running in fail-fast status, caused by this SQL:***** which threw a fatal
exception as *****.
```

当分库包含的RDS MySQL实例出现访问异常，并且分库下没有其他可用实例时，PolarDB-X 1.0会将分库置于 `fail-fast` 状态并提示TDDL-4200错误。

通常该错误是由于RDS MySQL故障导致的。请根据包含的RDS MySQL实例异常信息定位和解决问题。当故障RDS MySQL实例恢复后，PolarDB-X 1.0将自动取消 `fail-fast` 状态。

如果RDS MySQL故障解决后仍然出现TDDL-4200错误，请提交工单。

TDDL-4201 ERR_GROUP_NO_ATOM_AVALILABLE

PolarDB-X 1.0分库内暂时没有可用RDS MySQL实例。

示例：

```
ERR-CODE: [TDDL-4201][ERR_GROUP_NO_ATOM_AVALILABLE] All weights of DBs in
Group '******' is 0. Weights is: *****.
```

当分库包含的RDS MySQL实例全都不可用，或者处于 `fail-fast` 状态时，PolarDB-X 1.0会提示TDDL-4201错误。

通常该错误是由于RDS MySQL故障导致。请检查后端RDS MySQL实例状态以定位和解决问题。如果故障解决后仍然出现TDDL-4201错误，请提交工单。

TDDL-4202 ERR_SQL_QUERY_TIMEOUT

PolarDB-X 1.0查询超时。

示例：

```
ERR-CODE: [TDDL-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.
```

该错误表示SQL语句在后端RDS MySQL实例上的执行时间超过PolarDB-X 1.0设置的 `socketTimeout` 参数限制。默认的PolarDB-X 1.0超时（socketTimeout）时间设置是900秒。

建议优化SQL语句，以及在后端RDS MySQL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语句仍然执行较慢，可以参考如下PolarDB-X 1.0 Hint语法临时设置超时时间：

```
/*TDDL:SOCKET_TIMEOUT=900000*/ SELECT * FROM dual;
```

其中 `SOCKET_TIMEOUT` 设置的单位是毫秒。

关于PolarDB-X 1.0 Hint，详情请参见自定义SQL超时时间。

如果需要永久调整PolarDB-X 1.0超时设置，请提交工单。

TDDL-4203 ERR_SQL_QUERY_MERGE_TIMEOUT

分布式查询超时。

示例：

```
ERR-CODE: [TDDL-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 *****
```

PolarDB-X 1.0执行分布式查询超时，默认的超时设置是900秒。

产生TDDL-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：

- 尽量在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RDS MySQL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，降低数据合并阶段的消耗。

如果优化后的SQL语句仍然执行较慢，可以使用下面的Hint语法临时设置PolarDB-X 1.0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 1.0 Hint，详情请参见自定义SQL超时时间。

如果仍然出现TDDL-4203错误，请提交工单。

TDDL-4400 ERR_SEQUENCE

处理Sequence（全局唯一序列）失败。

示例：

```
ERR-CODE: [TDDL-4400][ERR_SEQUENCE] Sequence : All dataSource faild to get
value!
```

该错误表示处理Sequence出错，错误信息在 `Sequence :` 中给出。

导致该错误的常见原因是RDS MySQL故障，无法访问Sequence有关的数据表。建议先检查后端RDS MySQL状态。如果排除RDS MySQL故障后仍然发生错误，请提交工单。

TDDL-4401 ERR_MISS_SEQUENCE

Sequence不存在。

示例：

```
ERR-CODE: [TDDL-4401][ERR_MISS_SEQUENCE] Sequence '*****' is not found
```

命令中使用的Sequence名称不存在。建议用 `SHOW SEQUENCES` 命令检查PolarDB-X 1.0中所有已创建的Sequence名称，并且选择正确的Sequence名称。

如果使用的Sequence尚不存在，可以用 `CREATE SEQUENCE` 语法创建：

```
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TDDL-4401错误，请提交工单。

关于Sequence，详情请参见Sequence。

TDDL-4403 ERR_MISS_SEQUENCE_TABLE_ON_DEFAULT_DB

Sequence使用的数据表不存在。

示例：

```
ERR-CODE: [TDDL-4403][ERR_MISS_SEQUENCE_TABLE_ON_DEFAULT_DB] Sequence table
is not in default db.
```

无法在后端的数据库里访问名称为 `sequence` 或者 `sequence_opt` 的数据表。请提交工单。

TDDL-4404 ERR_SEQUENCE_TABLE_META

Sequence数据表结构错误。

示例：

```
ERR-CODE: [TDDL-4404][ERR_SEQUENCE_TABLE_META] the meta of sequence table
is error, some columns missed
```

Sequence相关数据表（如 `sequence` 或 `sequence_opt` ）中缺少相应的字段。请提交工单。

TDDL-4405 ERR_INIT_SEQUENCE_FROM_DB

初始化Sequence错误。

示例：

```
ERR-CODE: [TDDL-4405][ERR_INIT_SEQUENCE_FROM_DB] init sequence manager
error: *****
```

在初始化需要访问的Sequence时出错，错误信息在 `init sequence manager error:` 后给出。

建议先检查后端RDS MySQL状态。如果排除RDS MySQL故障后仍然提示TDDL-4405错误，请提交工单。

TDDL-4407 ERR_OTHER_WHEN_BUILD_SEQUENCE

访问Sequence数据表出错。

示例：

```
ERR-CODE: [TDDL-4407][ERR_OTHER_WHEN_BUILD_SEQUENCE] error when build
sequence: *****
```

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。

建议先检查后端RDS MySQL状态。如果排除RDS MySQL故障后仍然提示TDDL-4407错误，请提交工单。

DDL-4408 ERR_SEQUENCE_NEXT_VALUE

获取Sequence值出错。

示例：

```
ERR-CODE: [TDDL-4408][ERR_SEQUENCE_NEXT_VALUE] error when get sequence's
next value, sequence is: *****, error: *****
```

使用PolarDB-X 1.0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TDDL-4408错误的原因一般来自后端的RDS MySQL故障。建议先检查后端RDS MySQL状态和访问压力。如果排除RDS MySQL故障后仍然提示TDDL-4408错误，请提交工单。

TDDL-4500 ERR_PARSER

解析SQL语句失败。

示例：

```
ERR-CODE: [TDDL-4500][ERR_PARSER] not support statement: '*****'
```

PolarDB-X 1.0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 1.0兼容的SQL语法标准及MySQL规范。

关于SQL标准语法，请参见SQL标准语法。

关于PolarDB-X 1.0兼容的SQL语法，请参见SQL使用限制。

如果您的SQL语句符合上述语法仍然提示TDDL-4500错误，请提交工单。

TDDL-4501 ERR_OPTIMIZER

优化器转换SQL语句失败。

示例：

```
ERR-CODE: [TDDL-4501][ERR_OPTIMIZER] optimize error by: Unknown column
'*****' in 'order clause'
```

PolarDB-X 1.0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TDDL-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TDDL-4501错误，请提交工单。

TDDL-4502 ERR_OPTIMIZER_MISS_ORDER_FUNCTION_IN_SELECT

`ORDER BY` 包含的函数列在 `SELECT` 子句中不存在。

示例：

```
ERR-CODE: [TDDL-4502][ERR_OPTIMIZER_MISS_ORDER_FUNCTION_IN_SELECT] Syntax
Error: orderBy/GroupBy Column ***** is not existed in select clause`
```

当SQL语句中的 `ORDER BY` 子句包含函数列（例如RAND（））时，PolarDB-X 1.0要求同样的函数列必须也在 `SELECT` 子句中出现，否则提示TDDL-4502错误。

建议在 `SELECT` 子句中添加相应的函数列。

TDDL-4504 ERR_OPTIMIZER_SELF_CROSS_JOIN

相同表JOIN的条件不足。

示例：

```
ERR-CODE: [TDDL-4504][ERR_OPTIMIZER_SELF_CROSS_JOIN] self cross join case,
add shard column filter on right table
```

PolarDB-X 1.0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TDDL-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。

TDDL-4506 ERR_MODIFY_SHARD_COLUMN

禁止更新拆分键。

示例：

```
ERR-CODE: [TDDL-4506][ERR_MODIFY_SHARD_COLUMN] Column '*****' is a sharding
key of table '*****', which is forbidden to be modified.
```

PolarDB-X 1.0禁止使用UPDATE语句修改拆分键（sharding key）的值。由于更新操作很可能会改变数据所在的分片，PolarDB-X 1.0无法保证操作的原子性与数据的一致性。

建议将对应 `UPDATE` 语句修改为相同效果的 `INSERT+DELETE` 语句。

TDDL-4508 ERR_OPTIMIZER_NOT_ALLOWED_SORT_MERGE_JOIN

无法执行合并排序JOIN。

示例：

```
ERR-CODE: [TDDL-4508][ERR_OPTIMIZER_NOT_ALLOWED_SORT_MERGE_JOIN] sort merge
join is not allowed when missing equivalent filter
```

如果SQL语句中需要JOIN的数据表分别来自不同的RDS MySQL实例，PolarDB-X 1.0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，否则将提示TDDL-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。

TDDL-4509 ERR_OPTIMIZER_ERROR_HINT

Hint语法错误。

示例：

```
ERR-CODE: [TDDL-4509][ERR_OPTIMIZER_ERROR_HINT] Hint Syntax Error:
unexpected operation: *****.
```

该错误表示SQL语句中的Hint语法无法被PolarDB-X 1.0解析。更多关于Hint语法信息，请参见Hint简介。

TDDL-4510 ERR_CONTAINS_NO_SHARDING_KEY

缺少拆分键（sharding key）条件。

示例：

```
ERR-CODE: [TDDL-4510][ERR_CONTAINS_NO_SHARDING_KEY] Your SQL contains NO
SHARDING KEY '*****' for table '*****', which is not allowed in DEFAULT.
```

如果PolarDB-X 1.0拆分表没有开启全表扫描（full-table scan）功能，则访问该表时必须在 `WHERE` 子句中包含拆分键条件。否则，将提示TDDL-4510错误。

PolarDB-X 1.0在建表时默认开启全表扫描功能。如果手工关闭全表扫描，建议确认与该表有关的SQL语句都已添加拆分键条件。

TDDL-4511 ERR_INSERT_CONTAINS_NO_SHARDING_KEY

`INSERT` 语句缺少拆分键 (sharding key)。

示例：

```
ERR-CODE: [TDDL-4511][ERR_INSERT_CONTAINS_NO_SHARDING_KEY] Your INSERT SQL
contains NO SHARDING KEY '*****' for table '*****'.
```

当INSERT语句的目标是一张PolarDB-X 1.0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，将提示TDDL-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。

TDDL-4515 ERR_CONNECTION_CHARSET_NOT_MATCH

输入字符集不匹配。

示例：

```
ERR-CODE: [TDDL-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.
```

PolarDB-X 1.0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TDDL-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 x` 命令修改当前连接字符集。如果是Java程序JDBC方式连接PolarDB-X 1.0，请设置数据库连接参数 `characterEncoding` 。

TDDL-4516 ERR_TRUNCATED_DOUBLE_VALUE_OVERFLOW

浮点数转换为整数溢出。

示例：

```
ERR-CODE: [TDDL-4516][ERR_TRUNCATED_DOUBLE_VALUE_OVERFLOW] Truncated
incorrect DOUBLE value '*****' over column[*****]'s value range.
```

PolarDB-X 1.0将浮点数转换成对应类型的整数时，结果超过了整数类型的取值范围。建议检查SQL语句中的字段类型和输入参数。

TDDL-4517 ERR_MODIFY_SYSTEM_TABLE

禁止修改系统表。

示例：

```
ERR-CODE: [TDDL-4517][ERR_MODIFY_SYSTEM_TABLE] Table '*****' is PolarDB-XSYSTEM
TABLE, which is forbidden to be modified.
```

PolarDB-X 1.0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TDDL-4517错误。

限制的系统表包括 `sequence` 、 `sequence_opt` 、 `txc_undo_log` 和 `__DRDS__SYSTEM__LOCK__` ，请避免在业务或数据库设计中使用这些表名。

TDDL-4518 ERR_VALIDATE

元数据校验失败。

示例：

```
ERR-CODE: [TDDL-4518][ERR_VALIDATE] Object 'optest1' not found
```

SQL发送到PolarDB-X 1.0计算节点上，会基于现有的元信息作校验，这个异常说明查询的表或者列等信息不符合现有元信息要求。

TDDL-4600 ERR_FUNCTION

错误的函数调用。

示例：

```
ERR-CODE: [TDDL-4600][ERR_FUNCTION] function compute error by Incorrect
parameter count in the call to native function '*****'
```

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，并且使用正确的参数个数和类型调用函数。

TDDL-4601 ERR_EXECUTOR

SQL执行过程出错。

示例：

```
ERR-CODE: [TDDL-4601][ERR_EXECUTOR] only one column is supported in
distinct aggregate
```

该错误代表PolarDB-X 1.0在执行SQL语句过程中出现了意外错误。这类错误通常与后端RDS MySQL异常状态有关。建议先检查后端RDS MySQL，如果排除RDS MySQL故障后仍然发生此类错误，请提交工单。

TDDL-4602 ERR_CONVERTOR

错误的类型转换。

示例：

```
ERR-CODE: [TDDL-4602][ERR_CONVERTOR] convertor error by Unsupported convert:
[*****]
```

该错误表明PolarDB-X 1.0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。

TDDL-4603 ERR_ACCROSS_DB_TRANSACTION

跨库事务失败。

示例：

```
ERR-CODE: [TDDL-4603][ERR_ACCROSS_DB_TRANSACTION] Transaction accross db is
not supported in current transaction policy, transaction node is: {0}, but
this sql execute on: *****.
```

PolarDB-X 1.0默认仅支持单库事务，即事务中的所有SQL语句都必须按规则转发到相同RDS MySQL分库执行。否则，将提示TDDL-4603错误。

TDDL-4604 ERR_CONCURRENT_TRANSACTION

嵌套事务失败。

示例：

```
ERR-CODE: [TDDL-4604][ERR_CONCURRENT_TRANSACTION] Concurrent query is not
supported on transaction group, transaction group is: {0}.
```

PolarDB-X 1.0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TDDL-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。

TDDL-4606 ERR_QUERY_C

当前执行的SQL被取消。

示例：

```
ERR-CODE: [TDDL-4606][ERR_QUERY_CANCLED] Getting connection is not allowed
when query has been cancled, group is *****
```

使用 `KILL x` 取消某条正在执行的SQL语句时，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。

TDDL-4607 ERR_INSERT_WHEN_UPDATE

以 `INSERT DELETE` 方式执行 `UPDATE` 语句时出错。

示例：

```
ERR-CODE: [TDDL-4607][ERR_INSERT_WHEN_UPDATE] Insert new values error,
table is: *****, old Values: *****, new Values: *****
```

开启允许更新分库键（Sharding-key）功能后，PolarDB-X 1.0可以将 `UPDATE` 分库键值的SQL语句按照 `INSERT+DELETE` 方式转换执行。如果执行失败，则提示TDDL-4607错误。

该错误通常是因为后端RDS MySQL故障导致的。建议首先检查RDS MySQL状态，如果排除RDS MySQL故障后仍然发生错误，请提交工单。

TDDL-4610 ERR_CONNECTION_CLOSED

连接已经关闭。

示例：

```
ERR-CODE: [TDDL-4610][ERR_CONNECTION_CLOSED] connection has been closed
```

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TDDL-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。

TDDL-1305 ERR_UNKNOWN_SAVEPOINT

指定名称的 `SAVEPOINT` 不存在。

示例：

```
ERR-CODE: [TDDL-1305][ERR_UNKNOWN_SAVEPOINT] SAVEPOINT ***** does not exist
```

在PolarDB-X 1.0上执行 `ROLLBACK TO SAVEPOINT x` 或者 `RELEASE SAVEPOINT x` 命令时，如果指定的 `SAVEPOINT` 名称不存在，会提示TDDL-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。

TDDL-1094 ERR_UNKNOWN_THREAD_ID

`KILL` 命令指定的会话ID不存在。

示例：

```
ERR-CODE: [TDDL-1094][ERR_UNKNOWN_THREAD_ID] Unknown thread id: *****
```

在PolarDB-X 1.0上执行 `KILL x` 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TDDL-1094错误。

建议使用 `SHOW PROCESSLIST` 命令查看正在执行的SQL语句会话ID，并只使用返回的ID执行 `KILL` 操作。

TDDL-4612 ERR_CHECK_SQL_PRIV

由于权限不够，SQL语句无法执行。

示例：

```
ERR-CODE: [TDDL-4612][ERR_CHECK_SQL_PRIV] check user ***** on db ***** sql
privileges failed.
```

PolarDB-X 1.0支持账号和授权，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 1.0将提示TDDL-4612错误。

建议检查用户拥有的PolarDB-X 1.0权限。如果权限不足，请在PolarDB-X 1.0控制台设置。

TDDL-4613 ERR_INSERT_SELECT

执行 `INSERT ... SELECT` 语句出错。

示例：

```
ERR-CODE: [TDDL-4613][ERR_INSERT_SELECT] insert error, table is: *****,
values: *****
```

PolarDB-X 1.0支持将跨库的 `INSERT ... SELECT` 语句拆分成独立的SELECT和INSERT语句批量执行。如果执行失败，则提示TDDL-4613错误。

该错误通常是因为后端RDS MySQL故障导致的。建议首先检查RDS MySQL状态，如果排除RDS MySQL故障后仍然发生错误，请提交工单。

TDDL-4614 ERR_EXECUTE_ON_MYSQL

SQL语句在RDS MySQL上执行报错。

示例：

```
ERR-CODE: [TDDL-4614][ERR_EXECUTE_ON_MYSQL] Error occurs when execute on GROUP '*****': Dup
licate entry '*****' for key 'PRIMARY'
```

PolarDB-X 1.0在后端RDS MySQL数据库上执行SQL语句报错，末尾包含了从RDS MySQL返回的原始错误信息，例如：

- `Duplicate entry '*****' for key 'PRIMARY'` 表示写入RDS MySQL数据表发生了主键冲突。
- `The table '*****' is full` 表示RDS MySQL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RDS MySQL中出现了死锁，通常是数据写入存在较多事务冲突导致的。

建议参考TDDL-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RDS MySQL问题后仍然发生TDDL-4614错误，请提交工单。

TDDL-4615 ERR_CROSS_JOIN_SIZE_PROTECTION

分布式JOIN返回的数据行数超出限制。

示例：

```
ERR-CODE: [TDDL-4615][ERR_CROSS_JOIN_SIZE_PROTECTION] across join table size protection, ch
eck your sql or enlarge the limination size .
```

当PolarDB-X 1.0以嵌套循环（Nested-loop）方式执行分布式JOIN语句时，如果从右表返回大量数据，将严重占据内存，影响PolarDB-X 1.0服务的稳定性。因此，PolarDB-X 1.0限制这种情况下右表返回的数据行数最多不能超过5000。如果超出这一限制，PolarDB-X 1.0会提示TDDL-4615错误。

建议优化SQL语句避免从右表返回大量数据，或者让PolarDB-X 1.0能够使用更好的算法（如合并排序Sort-merge Join）执行分布式JOIN。

如果需要针对某个具体SQL调整该值，建议按以下原则操作：

- 如果单条记录超过100K，不建议调整该值。
- 如果单条记录不超过100K，可以适当调整。建议不要设置太大的值，否则会有爆内存的风险。
- 假设单条记录是100K，参与分布式JOIN计算的内存就需要500M（100K*5000）。如果有多个连接都在运行该SQL，很容易引起爆内存，例如有5个连接同时在运行该SQL，就需要2.5G（500M*5）的内存。
- 如果确实需要调整，请在需要调整的SQL前面加HINT，例如调整成5100：`/*!TDDL:MAX_ROW_RETURN_FROM_RIGHT_INDEX_NESTED_LOOP=5100*/SQL`。
- 如果需要全局修改这一限制，请提交工单。

TDDL-4616 ERR_UNKNOWN_DATABASE

错误的数据库。

示例：

```
ERR-CODE: [TDDL-4616][ERR_UNKNOWN_DATABASE] Unknown database '*****'
```

PolarDB-X 1.0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 1.0提供的数据库名称不一致，将返回TDDL-4616错误。

建议修改DDL语句中的数据库名称，确保与PolarDB-X 1.0数据库名称一致。

TDDL-4617 ERR_SUBQUERY_LIMIT_PROTECTION

子查询返回的数据行数超出限制。

示例：

```
ERR-CODE: [TDDL-4617][ERR_SUBQUERY_LIMIT_PROTECTION] The number of rows returned by the sub
query exceeds the maximum number of 20000.
```

当PolarDB-X 1.0执行含子查询（Sub-query）的SQL语句时，如果子查询返回大量数据，会严重占据内存，影响PolarDB-X 1.0服务的稳定性。因此，PolarDB-X 1.0限制这种情况下子查询返回的数据行数最多不能超过20000。如果超出这一限制，PolarDB-X 1.0会返回TDDL-4617错误。

建议优化SQL语句中的子查询避免返回大量数据。或者设法将子查询改写成JOIN形式，让PolarDB-X 1.0能够使用更好的算法（如合并排序Sort-merge Join）执行。

如果需要修改这一限制，请提交工单。

TDDL-4800 ERR_SET_TXCID

执行 `SET TXC_ID x` 命令失败。

示例：

```
ERR-CODE: [TDDL-4800][ERR_SET_TXCID] set txc_id failed: *****
```

具体原因请参见GTS文档什么是全局事务服务GTS。

TDDL-4801 ERR_TXCID_NULL

执行 `SELECT LAST_TXC_ID` 返回NULL。

示例：

```
ERR-CODE: [TDDL-4801][ERR_TXCID_NULL] txc_xid is null: *****
```

具体原因请参见GTS文档什么是全局事务服务GTS。

TDDL-4802 ERR_SELECT_LAST_TXCID

执行 `SELECT LAST_TXC_ID` 命令失败。

示例：

```
ERR-CODE: [TDDL-4802][ERR_SELECT_LAST_TXCID] select last_txc_xid failed: *****
```

具体原因请参见GTS文档什么是全局事务服务GTS。

TDDL-4994 ERR_FLOW_CONTROL

流量控制。

示例：

```
ERR-CODE: [TDDL-4994][ERR_FLOW_CONTROL] [*****] flow control by *****
```

该错误代表PolarDB-X 1.0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TDDL-4994错误，请提交工单。

TDDL-4998 ERR_NOT_SUPPORT

不支持的特性。

示例：

```
ERR-CODE: [TDDL-4998][ERR_NOT_SUPPORT] ***** not support yet!
```

该错误表示使用的SQL语法或者功能PolarDB-X 1.0尚不支持。

如果这些SQL语法或者功能对您十分重要，请提交工单。

TDDL-5001 ERR_TRANS

一般性的事务错误。

示例：

```
ERR-CODE: [TDDL-5001][ERR_TRANS] Too many lines updated in statement.
```

请参考错误信息处理，`Too many lines updated in statement`表示事务中的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 1.0 Hint语句`/*TDDL:UNDO_LOG_LIMIT={number}*/`调整限制值。

`Deferred execution is only supported in Flexible or XA Transaction`表示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 1.0 Hint语句`/*TDDL:DEFER*/`提交后置执行语句之前，请先用`SET drds_transaction_policy = ***`命令更改事务策略。

其他错误信息，请提交工单。

TDDL-5002 ERR_TRANS_UNSUPPORTED

事务中的语法或功能尚不支持。

示例：

```
ERR-CODE: [TDDL-5002][ERR_TRANS_UNSUPPORTED] Table without primary keys is not
supported.
```

功能在PolarDB-X 1.0事务中尚不支持，如果此功能很重要，请提交工单。

TDDL-5003 ERR_TRANS_LOG

无法访问事务日志。

示例：

```
ERR-CODE: [TDDL-5003][ERR_TRANS_LOG] Failed to update transaction state: *****
```

为保证分布式事务的原子性，PolarDB-X 1.0在事务中会访问后端RDS MySQL上的事务日志。如果PolarDB-X 1.0在读写事务日志时出错，将返回TDDL-5003错误。

产生TDDL-5003错误的原因通常来自后端的RDS MySQL故障。建议检查PolarDB-X 1.0后端RDS MySQL状态和访问压力。如果排除RDS MySQL问题后仍然产生TDDL-5003错误，请提交工单。

TDDL-5004 ERR_TRANS_NOT_FOUND

事务ID不存在。

示例：

```
ERR-CODE: [TDDL-5008][ERR_TRANS_TERMINATED] Current transaction was killed
or timeout. You may need to set a longer timeout value.
```

如果事务在执行中被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 = ***命令修改事务的执行时间上限，单位是毫秒。

TDDL-5006 ERR_TRANS_COMMIT

事务提交过程中出错。

示例：

```
ERR-CODE: [TDDL-5006][ERR_TRANS_COMMIT] Failed to commit primary group *****:
*****, TRANS_ID = *****
```

PolarDB-X 1.0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TDDL-5006错误的原因通常来自后端的RDS MySQL故障。建议检查PolarDB-X 1.0后端RDS MySQL状态和访问压力。如果排除RDS MySQL问题后仍然产生TDDL-5006错误，请提交工单。

TDDL-5007 ERR_TRANS_PARAM

事务参数不正确。

示例：

```
ERR-CODE: [TDDL-5007][ERR_TRANS_PARAM] Illegal timeout value: *****
```

命令中指定事务参数不正确，例如 `SET drds_transaction_timeout = ***` 指定的参数值是负数。

TDDL-5008 ERR_TRANS_TERMINATED

事务已Kill或超时中止。

示例：

```
ERR-CODE: [TDDL-5008][ERR_TRANS_TERMINATED] Current transaction was killed
or timeout. You may need to set a longer timeout value.
```

如果事务在执行中被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 = ***` 事务的执行时间上限，单位是毫秒。