ALIBABA CLOUD

Alibaba Cloud

云原生分布式数据库 PolarDB-X

SQL手册

文档版本：20201023

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

DRDS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）。

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，请参见运算符简介。

- 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.分库分表

2.1. 拆分函数概述

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

拆分方式

在DRDS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DRDS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DRDS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DRDS在查询过程会进行全分库扫描或全分表扫描操作。

拆分函数对全局二级索引的支持情况

- DRDS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。
- DRDS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。

拆分函数的语法说明

DRDS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. 拆分函数使用说明

2.2.1. 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);
```

2.2.2. 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。
- DRDS的实例版本需为5.3.5或以上

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DRDS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）。

参数	说明
endIndex	目标子串的结束位置的下标。取值从0开始（即原字符串的第1个字符的下标用0表示），默认值为-1（即不做任何截取）。 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 EF G 。 ◦ 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。

参数	说明
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DRDS将截取后的子串当作字符串类型来计算路由。 • 1：表示DRDS将截取后的子串当作整数类型来计算路由（子串面值的整数不能大于9223372036854775807，也不支持浮点数）
randSeed	当子串以字符串类型来计算路由的哈希值时DRDS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。

2.2.3. 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。
- DRDS实例的版本需为5.1.28-1508068或以上。

路由方式

- 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。	
使用同一个拆分键进行分库分表。	同一个键值分到的分库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。

2.2.4. 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
- DRDS实例的版本需为5.1.28-1320920或以上版本

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DRDS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;
```

2.2.5. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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;
```

2.2.6. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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;
```

2.2.7. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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;
```

2.2.8. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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;
```

2.2.9. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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;
```

2.2.10. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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DRDS里已经拥有8个物理库，现有如下需求：

- 按年天进行分库。
- 同一周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。DRDS已有8个分库，所以每个分库应该建92张物理分表（732/8=91.5，取整为92，分表数最好是分库数的整数倍）。

则您可以使用如下建表DDL：

```
create table test_yyyydd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYDD(create_time)
tbpartition by YYYYDD(create_time) tbpartitions 92;
```

2.2.11. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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DRDS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。DRDS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;
```

2.2.12. 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。
- DRDS实例的版本需为5.1.28-1320920或以上版本，。

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DRDS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。DRDS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

3.1. DDL任务管理

3.1.1. 任务管理语句

任务管理语句是DRDS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
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。

字段	含义
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。
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 varchar(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 varchar(10), c
3 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 varchar(10), c
  3 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 varchar(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(1
0), 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 bee
n 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` (`id`) 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_D
DL_JOB_ERROR] Not all physical operations have been done successfully: expected 9,
but done 0. Caused by: 1217:DDLTEST_1562056402230OYMK_7WW7_0007:Cannot delete or updat
e a parent row: a foreign key constraint fails on `test_parent`;1217:DDLTEST_15620564022
30OYMK_7WW7_0000:Cannot delete or update a parent row: a foreign key constraint fails on `t
est_parent`;1217:DDLTEST_1562056402230OYMK_7WW7_0002:Cannot delete or update a pare
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 have 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 fails 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_U
NKNOWN_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)
```

3.1.2. 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DRDS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，DRDS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DRDS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DRDS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DRDS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DRDS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DRDS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DRDS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DRDS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。

3.2. 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

- DRDS目前不支持使用DDL语句直接建库，请登录云原生分布式数据库控制台进行创建。关于如何创建数据库，详情请参见创建数据库。
- DRDS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DRDS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
```

```
 | 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 DRDS DDL语法基于MySQL语法，以上主要列出了差异部分，详细语法请参见MySQL 文档。

分库分表子句和参数

- `DBPARTITION BY hash(partition_key)`：指定分库键和分库算法；
- `TBPARTITION BY { HASH(column) | {MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(column)`（可选）：默认与 `DBPARTITION BY` 相同，指定物理表使用什么方式映射数据；
- `TBPARTITIONS num`（可选）：每个库上的物理表数目（默认为1），如不分表，就不需要指定该字段。
- 拆分函数的详细介绍，请参见拆分函数概述 。

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`：DRDS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_tb
l |
|   1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_single_tb
l |
|   2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_single_tb
l |
|   3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_single_tb
l |
|   4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_single_tb
l |
|   5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_single_tb
l |
|   6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_single_tb
l |
|   7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_single_tb
l |
+------+-------------------------------------------------------------------+--------------------+
8 rows in set (0.01 sec)
```

分库分表

您可以使用如下拆分方式进行分库分表：

- 使用哈希函数做拆分
- 使用双字段哈希函数做拆分
- 使用日期做拆分

> **说明** 以下示例均假设已经建好的分库数为8。

使用哈希函数做拆分

建一张表，既分库又分表，每个库含有3张物理表，分库拆分方式为按照ID列进行哈希，分表拆分方式为按照bid列进行哈希。您可以先根据ID列的值进行哈希运算，将表中数据分布在多个子库中，每个子库中的数据再根据bid列值的哈希运算结果分布在3个物理表中。

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

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了3张分表。

```
mysql> show topology from multi_db_multi_tbl;
+------+-----------------------------------------------------------------+----------------------+
| ID   | GROUP_NAME                                      | TABLE_NAME         |
+------+-----------------------------------------------------------------+----------------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl
_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl
_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl
_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl
_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl
_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl
_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl
_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl
_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl
_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl
_09 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl
_10 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl
_11 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl
_12 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl
_13 |
```

```
|  14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl
_14 |
|  15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl
_15 |
|  16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl
_16 |
|  17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl
_17 |
|  18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl
_18 |
|  19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl
_19 |
|  20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl
_20 |
|  21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl
_21 |
|  22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl
_22 |
|  23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl
_23 |
+------+-------------------------------------------------------------------+----------------------+
24 rows in set (0.01 sec)
```

查看该逻辑表的拆分规则，可以看出分库分表的拆分方式均为哈希，分库的拆分键为ID，分表的拆分键为bid。

```
mysql> show rule from multi_db_multi_tbl;
+------+--------------------+-----------+-----------------+--------------------+--------------------+---------------
---+--------------------+--------------------+
| ID   | TABLE_NAME         | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COU
NT | 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DRDS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) tbpartitio
n by RANGE_HASH(buyer_id, order_id, 10) tbpartitions 3;
```

> 说明
> - 两个拆分键皆不能修改。
> - 插入数据时如果发现两个拆分键指向不同的分库或分表时，插入会失败。

使用日期做拆分

除了可以使用哈希函数做拆分算法，您还可以使用日期函数 `MM` 、 `DD` 、 `WEEK` 或 `MMDD` 来作为分表的拆分算法，具体步骤请参见如下示例。

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为根据 `actionDate` 列，按照一周七天来拆分（ `WEEK(actionDate)` 计算的是 `DAY_OF_WEEK` ）。

比如 `actionDate` 列的值是2017-02-27，这天是星期一， `WEEK(actionDate)` 算出的值是2，该条记录就会被存储到 `2（2 % 7 = 2）` 这张分表（位于某个分库，具体的表名是 `user_log_2` ）；比如 `actionDate` 列的值是2017-02-26，这天是星期天， `WEEK(actionDate)` 算出的值是1，该条记录就会被存储到 `1（1 % 7 = 1）` 这张分表（位于某个分库，具体的表名是 `user_log_1` ）。

```
CREATE TABLE user_log(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by WEEK(actionDate) tbpartitions 7;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了7张分表（一周7天）。

```
mysql> show topology from user_log;
+------+--------------------------------------------------------------------+------------+
| ID   | GROUP_NAME                                                         | TABLE_NAME |
+------+--------------------------------------------------------------------+------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_0 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_1 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_2 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_3 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_4 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_5 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_6 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_0 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_1 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_2 |
|   10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_3 |
|   11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_4 |
|   12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_5 |
|   13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_6 |
...
|   49 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_0 |
|   50 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_1 |
|   51 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_2 |
|   52 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_3 |
|   53 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_4 |
|   54 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_5 |
|   55 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_6 |
+------+--------------------------------------------------------------------+------------+
56 rows in set (0.01 sec)
```

> **说明** 由于返回结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `WEEK` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log;
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
|    0 | user_log   |         0 | userId          | hash               | 8                  | actionDate      | week               | 7
|
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
1 row in set (0.00 sec)
```

查看给定分库键和分表键参数时， SQL 被路由到哪个物理分库和该物理分库下的哪张物理表。

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为根据 `actionDate` 列，按照一年 12 个月进行拆分（ `MM(actionDate)` 计算的是 `MONTH_OF_YEAR` ）。

比如 `actionDate` 列的值是2017-02-27， `MM(actionDate)` 算出的值是02，该条记录就会被存储到 `02（02 % 12 = 02）` 这张分表（位于某个分库，具体的表名是 `user_log_02` ）。比如 `actionDate` 列的值是2016-12-27， `MM(actionDate)` 算出的值是12，该条记录就会被存储到 `00（12 % 12 = 00）` 这张分表（位于某个分库，具体的表名是 `user_log_00` ）。

```
CREATE TABLE user_log2(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MM(actionDate) tbpartitions 12;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了12张分表（1年有12个月）。

```
mysql> show topology from user_log2;
+------+-----------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                                            | TABLE_NAME   |
+------+-----------------------------------------------------------------------+--------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_03 |
```

```
|  4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_04 |
|  5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_05 |
|  6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_06 |
|  7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_07 |
|  8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_08 |
|  9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_09 |
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_10 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_11 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_00 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_01 |
| 14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_02 |
| 15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_03 |
| 16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_04 |
| 17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_05 |
| 18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_06 |
| 19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_07 |
| 20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_08 |
| 21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_09 |
| 22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_10 |
| 23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_11 |
...
| 84 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_00 |
| 85 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_01 |
| 86 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_02 |
| 87 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_03 |
| 88 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_04 |
| 89 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_05 |
| 90 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_06 |
| 91 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_07 |
| 92 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_08 |
| 93 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_09 |
| 94 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_10 |
| 95 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_11 |
+------+-----------------------------------------------------------------+--------------+
96 rows in set (0.02 sec)
```

> **说明** 由于返回结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MM` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log2;
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
|   0 | user_log2 |       0 | userId        | hash           | 8            | actionDate     | mm           | 12
|
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
1 row in set (0.00 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一个月31天进行拆分（函数 `DD(actionDate)` 计算的是 `DAY_OF_MONTH` ）。

比如 `actionDate` 列的值是2017-02-27， `DD(actionDate)` 算出的值是27，该条记录就会被存储到 `27（27 % 31 = 27）` 这张分表（位于某个分库，具体的表名是 `user_log_27` ）。

```
CREATE TABLE user_log3(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by DD(actionDate) tbpartitions 31;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了31张分表（按每个月有31天处理）。

```
mysql> show topology from user_log3;
+------+-------------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                      | TABLE_NAME   |
+------+-------------------------------------------------------------------------+--------------+
|   0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_00 |
|   1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_01 |
|   2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_02 |
|   3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_03 |
|   4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_04 |
|   5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_05 |
|   6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_06 |
|   7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_07 |
|   8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_08 |
|   9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_09 |
```

```
|  9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_09 |
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_10 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_11 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_12 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_13 |
| 14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_14 |
| 15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_15 |
| 16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_16 |
| 17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_17 |
| 18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_18 |
| 19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_19 |
| 20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_20 |
| 21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_21 |
| 22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_22 |
| 23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_23 |
| 24 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_24 |
| 25 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_25 |
| 26 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_26 |
| 27 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_27 |
| 28 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_28 |
| 29 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_29 |
| 30 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_30 |
...
| 237 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_20 |
| 238 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_21 |
| 239 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_22 |
| 240 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_23 |
| 241 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_24 |
| 242 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_25 |
| 243 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_26 |
| 244 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_27 |
| 245 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_28 |
| 246 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_29 |
| 247 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_30 |
+------+-----------------------------------------------------------------+--------------+
248 rows in set (0.01 sec)
```

说明 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `DD` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log3;
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
|    0 | user_log3 |       0 | userId        | hash           | 8            | actionDate     | dd            | 31
|
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
1 row in set (0.01 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一年365天进行拆分，路由到365张物理表（ `MMDD(actionDate) tbpartitions 365` 计算的是 `DAY_OF_YEAR % 365` 。

比如 `actionDate` 列的值是2017-02-27， `MMDD(actionDate)` 算出的值是58，该条记录就会被存储到58这张分表（位于某个分库，具体的表名是 `user_log_58` ）。

```
CREATE TABLE user_log4(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 365;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了365张分表（按每年有365天处理）。

```
mysql> show topology from user_log4;
+------+--------------------------------------------------------------------+---------------+
| ID   | GROUP_NAME                                                         | TABLE_NAME    |
+------+--------------------------------------------------------------------+---------------+
...
| 2896 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_341 |
| 2897 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_342 |
| 2898 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_343 |
| 2899 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_344 |
| 2900 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_345 |
| 2901 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_346 |
| 2902 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_347 |
| 2903 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_348 |
| 2904 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_349 |
| 2905 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_350 |
| 2906 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_351 |
| 2907 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_352 |
| 2908 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_353 |
| 2909 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_354 |
| 2910 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_355 |
| 2911 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_356 |
| 2912 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_357 |
| 2913 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_358 |
| 2914 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_359 |
| 2915 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_360 |
| 2916 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_361 |
| 2917 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_362 |
| 2918 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_363 |
| 2919 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_364 |
+------+--------------------------------------------------------------------+---------------+
2920 rows in set (0.07 sec)
```

说明 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MMDD` 进行拆分，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log4;
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
|    0 | user_log4 |       0 | userId        | hash           | 8             | actionDate     | mmdd           | 365
|
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
----------------+--------------------+
1 row in set (0.02 sec)
```

建一张表，既分库又分表，分库方式为根据 `userId` 列哈希，分表方式为按照一年365天进行拆分，路由到10张物理表（ `MMDD(actionDate) tbpartitions 10` 计算的是 `DAY_OF_YEAR % 10` 。

```
CREATE TABLE user_log5(
 userId int,
 name varchar(30),
 operation varchar(30),
 actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 10;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了10张分表（按照一年365天进行拆分，路由到10张物理表）。

```
mysql> show topology from user_log5;
+------+-------------------------------------------------------------------+--------------+
| ID   | GROUP_NAME                                                        | TABLE_NAME   |
+------+-------------------------------------------------------------------+--------------+
|    0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_00 |
|    1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_01 |
|    2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_02 |
|    3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_03 |
|    4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_04 |
|    5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_05 |
|    6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_06 |
|    7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_07 |
|    8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_08 |
|    9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_09 |
...
|   70 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_00 |
|   71 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_01 |
|   72 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_02 |
|   73 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_03 |
|   74 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_04 |
|   75 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_05 |
|   76 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_06 |
|   77 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_07 |
|   78 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_08 |
|   79 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_09 |
+------+-------------------------------------------------------------------+--------------+
80 rows in set (0.02 sec)
```

说明 由于返回的结果较长，这里用...做了省略处理。

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 `userId` ，分表的拆分方式为按照时间函数 `MMDD` 进行拆分，路由到10张物理表，分表的拆分键为 `actionDate` 。

```
mysql> show rule from user_log5;
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
-----------------+--------------------+
| ID   | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
-----------------+--------------------+
|   0 | user_log5 |       0 | userId         | hash            | 8             | actionDate      | mmdd            | 10
|
+------+-----------+-----------+-----------------+--------------------+--------------------+-----------------+----
-----------------+--------------------+
1 row in set (0.01 sec)
```

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;
```

其他MySQL建表属性

您在分库分表的同时还可以指定其他的MySQL建表属性，例如：

```
CREATE TABLE multi_db_multi_tbl(
 id bigint not null auto_increment,
 name varchar(30),
 primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(id) tbpartition by hash(id) tbpartitions 3
;
```

全局二级索引

本小节介绍如何在建表时定义全局二级索引：

- 定义全局二级索引
- 定义全局唯一索引

> 说明 以下示例均假设已经建好的分库数为8。

定义全局二级索引

示例

```
CREATE TABLE t_order (
 `id` bigint(11) NOT NULL AUTO_INCREMENT,
 `order_id` varchar(20) DEFAULT NULL,
 `buyer_id` varchar(20) DEFAULT NULL,
 `seller_id` varchar(20) DEFAULT NULL,
 `order_snapshot` longtext DEFAULT NULL,
 `order_detail` longtext DEFAULT NULL,
 PRIMARY KEY (`id`),
 GLOBAL INDEX `g_i_seller`(`seller_id`) dbpartition by hash(`seller_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`order_id`);
```

其中：

- 主表：`t_order` 只分库不分表，分库的拆分方式为按照 `order_id` 列进行哈希。
- 索引表：`g_i_seller` 只分库不分表，分库的拆分方式为按照 `seller_id` 列进行哈希，未指定覆盖列。
- 索引定义子句：`GLOBAL INDEX `g_i_seller`(`seller_id`) dbpartition by hash(`seller_id`)` 。

通过 `SHOW INDEX` 查看索引信息，包含拆分键 `order_id` 上的局部索引，
和 `seller_id` 、 `id` 、 `order_id` 上的GSI，其中 `seller_id` 为索引表的拆分键， `id` 和 `order_id` 为默认的覆盖列（主键和主表的拆分键）。

```
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 | 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  | GLOB
AL     | 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  | GLO
BAL     | 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_PA
RTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY | TB_PARTITION_POLI
CY | 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。

```
mysql> show index from t_order;
+--------------+------------+------------------------+--------------+---------------+-----------+-------------+-----
-----+--------+------+------------+----------+---------------+
| TABLE      | NON_UNIQUE | KEY_NAME             | SEQ_IN_INDEX | COLUMN_NAME   | COLLATION | CARDIN
ALITY | SUB_PART | PACKED | NULL | INDEX_TYPE | COMMENT  | INDEX_COMMENT |
+--------------+------------+------------------------+--------------+---------------+-----------+-------------+-----
-----+--------+------+------------+----------+---------------+
| t_order_dthb |       0 | PRIMARY              |            1 | id          | A       |         0 |    NULL | NULL   |    | BTR
EE     |       |          |
| t_order_dthb |       1 | auto_shard_key_order_id |            1 | order_id     | A       |         0 |    NULL | NULL
| YES  | BTREE     |       |          |
| t_order     |       0 | g_i_buyer            |            1 | buyer_id     | NULL    |         0 |    NULL | NULL   | YES |
GLOBAL    | INDEX    |          |
| t_order     |       1 | g_i_buyer            |            2 | id           | NULL    |         0 |    NULL | NULL   |    | GLO
BAL    | COVERING |          |
| t_order     |       1 | g_i_buyer            |            3 | order_id     | NULL    |         0 |    NULL | NULL   | YES |
GLOBAL    | COVERING |          |
| t_order     |       1 | g_i_buyer            |            4 | seller_id    | NULL    |         0 |    NULL | NULL   | YES |
GLOBAL    | COVERING |          |
| t_order     |       1 | g_i_buyer            |            5 | order_snapshot | NULL    |         0 |    NULL | NULL   | Y
ES  | 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_PA
RTITION_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,
  `seller_id` varchar(20) DEFAULT NULL,
  `order_snapshot` longtext,
  PRIMARY KEY (`id`),
  UNIQUE KEY `auto_shard_key_buyer_id` (`buyer_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`buyer_id`) tbpartition by hash(`buyer_i
d`) tbpartitions 3 |
+-----------+-------------------------------------------------------------------------------------------------------------+
```

3.3. CREATE VIEW

本文将介绍如何使用CREATE VIEW语句为DRDS创建视图。

前提条件

DRDS实例版本需为5.4.5或以上。

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;
```

3.4. DROP VIEW

本文将介绍如何使用DROP VIEW语句删除DRDS的视图。

前提条件

DRDS实例版本需为5.4.5或以上。

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;
```

3.5. DDL常见问题

本文汇总了DRDS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. DRDS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_mult
i_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。

4.DML

4.1. INSERT

您可以使用INSERT语句往表中插入数据。

语法

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
{VALUES | VALUE} (value_list) [, (value_list)]
[ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
SET assignment_list
[ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
SELECT ...
[ON DUPLICATE KEY UPDATE assignment_list]

value_list:
value [, value] ...

value:
{expr | DEFAULT}

assignment_list:
assignment [, assignment] ...

assignment:
col_name = value
```

语法限制

不支持使用以下语法。

- INSERT IGNORE ON DUPLICATE KEY UPDATE语法，例如：

```
INSERT IGNORE INTO tb (id) VALUES(7) ON DUPLICATE KEY UPDATE id = id + 1;
```

- PARTITION语法，例如：

```
INSERT INTO tb PARTITION (p0) (id) VALUES(7);
```

- 嵌套NEXTVAL的语法，例如：

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

- 包含列名的语法，例如：

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

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。

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。

4.2. REPLACE

您可以使用REPLACE语法往表中插入行或替换表中的行。

语法

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
{VALUES | VALUE} (value_list) [, (value_list)]

REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
SET assignment_list

REPLACE [LOW_PRIORITY | DELAYED]
[INTO] [schema_name.]tbl_name
[(col_name [, col_name] ...)]
SELECT ...

value_list:
value [, value] ...

value:
{expr | DEFAULT}

assignment_list:
assignment [, assignment] ...

assignment:
col_name = value
```

语法限制

不支持使用以下语法。

- PARTITION语法，例如：

```
REPLACE INTO tb PARTITION (p0) (id) VALUES(7);
```

- 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。

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。

4.3. UPDATE

您可以使用UPDATE语法修改表中符合条件的行。

语法

- 单逻辑表

```
UPDATE [LOW_PRIORITY] [IGNORE] [schema_name.]tbl_name
   SET assignment_list
   [WHERE where_condition]

value:
   {expr | DEFAULT}

assignment:
   col_name = value

assignment_list:
   assignment [, assignment] ...
```

- 多逻辑表

```
UPDATE [LOW_PRIORITY] [IGNORE] table_references
  SET assignment_list
  [WHERE where_condition]
```

> 说明
> - UPDATE支持如下修饰符：
>   - 若设置LOW_PRIORITY，UPDATE操作将在该表没有任何读操作之后执行。
>   - 若设置IGNORE，将会忽略更新过程中的错误，即更新不会被错误中断。
> - UPDATE语句中的修饰符均会原样下推至存储层MySQL，不会对DRDS的修饰符操作产生影响。

语法限制

与原生MySQL的UPDATE语法相比，DRDS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。

相关文献

MySQL UPDATE语法。

4.4. DELETE

您可以使用DELETE语句删除表中符合条件的行。

语法

下述DELETE语句表示从 `tbl_name` 中删除满足 `where_condition` 的行，并返回删除的行数；若没有WHERE条件，将删除表中所有的数据。

- 单逻辑表

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM [schema_name.]tbl_name
  [WHERE where_condition]
```

- 多逻辑表

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
  tbl_name[.*] [, tbl_name[.*]] ...
  FROM table_references
  [WHERE where_condition]

DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
  FROM [schema_name.]tbl_name[.*] [, [schema_name.]tbl_name[.*]] ...
  USING table_references
  [WHERE where_condition]
```

说明

- DELETE支持如下修饰符：
  - 若设置LOW_PRIORITY，DELETE操作将在该表没有任何读操作之后执行。
  - 若设置IGNORE，则会忽略删除过程中产生的错误。
  - QUICK，与MySQL存储引擎相关，详情请参见MySQL文档。
- DELETE语句中的修饰符均会原样下推至存储层MySQL，不会对DRDS的修饰符操作产生影响。

语法限制

与原生MySQL的DELETE语法相比，DRDS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 1000
1;
```

说明 t1、t2和t3的拆分键均为ID。

4.5. 全局二级索引对DML的限制

本文将介绍DRDS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DRDS实例版本需为5.4.1或以上。

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

- INSERT SELECT、REPLACE SELECT、UPDATE和DELETE的更新行数不超过10000行

```
# INSERT SELECT 插入行数超过 10000
INSERT INTO t_order SELECT * FROM t_order_bak WHERE id BETWEEN 0 AND 20000;
# DELETE 删除行数超过 10000
DELETE FROM t_order WHERE id BETWEEN 0 AND 20000;
```

- INSERT IGNORE、INSERT ON DUPLICATE KEY UPDATE、REPLACE可能报主键冲突

```
INSERT INTO t_order(order_id, buyer_id, seller_id) VALUES('order_1', 'buyer_1', 'seller_1');
# IGNORE 仍有可能报主键冲突
INSERT IGNORE INTO t_order(order_id, buyer_id, seller_id) VALUES('order_2', 'buyer_1', 'seller_1');
```

- 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', 'seller_1');
# 失败不允许继续执行
INSERT IGNORE INTO t_order(order_id, buyer_id, seller_id) VALUES('order_2', 'buyer_2', 'seller_2');
# 失败后不允许提交事务
COMMIT;
```

5.DAL

5.1. SHOW

5.1.1. 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DRDS中的连接与正在执行的SQL等信息：

- 语法

```
SHOW PROCESSLIST
```

- 示例

```
mysql> SHOW PROCESSLIST\G
     ID: 1971050
   USER: admin
   HOST: 111.111.111.111:4303
     DB: drds_test
COMMAND: Query
   TIME: 0
  STATE:
   INFO: show processlist
1 row in set (0.01 sec)
```

参数	说明
ID	本次连接的ID，为一个Long型数字。
USER	建立此连接所使用的用户名。
HOST	建立此连接的机器的IP与端口。
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。

参数	说明
STATE	目前无意义，恒为空值。
INFO	○ 当COMMAND为Query时，为此连接上正在执行的SQL的内容。 说明 当不带FULL参数时，最多返回正在执行的SQL的前 30 个字符。当带FULL参数时，最多返回正在执行的SQL的前1000个字符。 ○ 当COMMAND为Sleep时，为空值，无意义。

SHOW PHYSICAL_PROCESSLIST

您可以使用如下指令查看所有正在执行的物理SQL信息：

- 语法

```
SHOW PHYSICAL_PROCESSLIST
```

说明 当SQL比较长的时候，使用 SHOW PHYSICAL_PROCESSLIST 语句返回得到的SQL会被截断，这时可以使用 SHOW FULL PHYSICAL_PROCESSLIST 语句获取完整SQL。

- 示例

```
mysql> SHOW PHYSICAL_PROCESSLIST\G
*************************** 1. row ***************************
        ID: 0-0-521414
      USER: tddl5
        DB: tddl5_00
   COMMAND: Query
      TIME: 0
     STATE: init
      INFO: show processlist
*************************** 2. row ***************************
        ID: 0-0-521570
      USER: tddl5
        DB: tddl5_00
   COMMAND: Query
      TIME: 0
     STATE: User sleep
      INFO: /*DRDS /88.88.88.88/b67a0e4d8800000/ */ select sleep(1000)
2 rows in set (0.01 sec)
```

> 说明
> - 返回结果中每一列的含义与MySQL的 `SHOW PROCESSLIST` 指令等价，详情请参见 SHOW PROCESSLIST Syntax。
> - 但与MySQL不同，DRDS返回的物理连接的ID列为一个字符串，并非一个数字。

5.1.2. SHOW GLOBAL INDEX

DRDS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           | COVE
RING_NAMES
| INDEX_TYPE | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT | TB_PARTITION_KEY |
TB_PARTITION_POLICY | TB_PARTITION_COUNT | STATUS   |
+--------------------+--------------------+------------+----------------------------+----------------------------
+-----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------+------------+----------------+-------
-------------+------------------+----------------+-------------------+------------------+----------+
| XXXX_DRDS_LOCAL_APP | full_gsi_ddl_renamed | 1        | g_i_c_ddl_c_blob_long_renamed | c_blob_long
| id, c_bit_1, c_bit_8, c_bit_16, c_bit_32, c_bit_64, c_tinyint_1, c_tinyint_1_un, c_tinyint_4, c_tinyint_4_un, c_
tinyint_8, c_tinyint_8_un, c_smallint_16, c_smallint_16_un, c_mediumint_1, c_mediumint_24, c_mediumint_
24_un, c_int_1, c_int_32, c_int_32_un, c_bigint_1, c_bigint_64, c_bigint_64_un, c_decimal, c_decimal_pr, c_fl
oat, c_float_pr, c_float_un, c_double, c_double_pr, c_double_un, c_date, c_datetime, c_datetime_3, c_dat
etime_6, c_timestamp_1, c_timestamp_3, c_time, c_time_1, c_time_3, c_time_6, c_year, c_year_4, c_char, c
_varchar, c_binary, c_varbinary, c_blob_tiny, c_blob_medium, c_text_tiny, c_text, c_text_medium, c_text_l
ong, c_enum, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_multilinestring, c_multipolyg
on, 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_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_time
stamp_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_bina
ry, c_varbinary, c_blob_tiny, c_blob_medium, c_blob_long, c_text_tiny, c_text, c_text_medium, c_text_lon
g, c_enum, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, 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_u
n, 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_tinyi
nt_4_un, c_tinyint_8, c_tinyint_8_un, c_smallint_16, c_mediumint_1, c_mediumint_24, c_mediumint_24_un,
c_int_1, c_int_32, c_int_32_un, c_bigint_1, c_bigint_64, c_bigint_64_un, c_decimal, c_decimal_pr, c_float, c_
float_pr, c_float_un, c_double, c_double_pr, c_double_un, c_date, c_datetime, c_datetime_3, c_datetime_
6, c_timestamp_1, c_timestamp_3, c_time, c_time_3, c_time_6, c_year, c_year_4, c_char, c_varchar, c_binar
y, c_varbinary, c_blob_tiny, c_blob_medium, c_blob_long, c_text_tiny, c_text, c_text_medium, c_text_long,
c_enum, c_set, c_json, c_point, c_linestring, c_polygon, c_multipoint, c_multilinestring, c_multipolygon, c_
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_i
d
| HASH      | seller_id       | HASH              | 4               | seller_id       | HASH              | 2               | CREATI
NG |
+--------------------+---------------------+-----------+-----------------------------+---------------------------
+-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------+-----------+-----------------+-------
-------------+------------------+-----------------+--------------------+------------------+----------+
4 rows in set (0.01 sec)
```

列名说明

列名	说明
SCHEMA	库名
TABLE	表名
NON_UNIQUE	是否为唯一约束全局二级索引，取值范围如下： • 1：普通全局二级索引 • 0：唯一约束全局二级索引
KEY_NAME	索引名
INDEX_NAMES	索引列
COVERING_NAMES	覆盖列
INDEX_TYPE	索引类型，取值范围如下： • NULL（即未指定） • BTREE • HASH
DB_PARTITION_KEY	分库拆分键
DB_PARTITION_POLICY	分库拆分函数
DB_PARTITION_COUNT	分库数量
TB_PARTITION_KEY	分表拆分键
TB_PARTITION_POLICY	分表拆分函数
TB_PARTITION_COUNT	分表数

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

5.1.3. SHOW INDEX

您可以使用SHOW INDEX语句查看DRDS表上的局部索引和全局索引信息。

语法

```
SHOW {INDEX | INDEXES | KEYS}
    {FROM | IN} tbl_name
    [{FROM | IN} db_name]
    [WHERE expr]
```

示例

```
mysql> show index from t_order;
+-------------+------------+----------+-------------+---------------+----------+------------+----------+-------
--+------+------------+----------+---------------+
| TABLE   | NON_UNIQUE | KEY_NAME  | SEQ_IN_INDEX | COLUMN_NAME   | COLLATION | CARDINALITY | SUB
_PART | 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    | CO
VERING |           |
| 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 | GLOB
AL    | 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

列名	说明
CARDINALITY	预计的唯一值数目
SUB_PART	索引前缀（NULL索引前缀为整个列）。
PACKED	字段压缩信息（NULL表示没有压缩）。
NULL	是否允许空。
INDEX_TYPE	索引类型，取值范围如下： • NULL（即未指定） • BTREE • HASH
COMMENT	索引信息，取值范围如下： • NULL：局部索引 • INDEX：全局二级索引的索引列 • COVERING：全局二级索引的覆盖列
INDEX_COMMENT	其他信息

5.1.4. SHOW METADATA LOCK

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

背景信息

DRDS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 DRDS支持Online Schema Change，添加全局二级索引过程中，会发生4次元数据版本切换，其中有两次会先获取METADATA LOCK的写锁加载元数据完成后立即解锁，其余的时间均不会持有METADATA LOCK的写锁。

语法

```
SHOW METADATA {LOCK | LOCKS} [schema_name[.table_name]]
```

`schema_name` 和 `tbl_name` 是可选的，用于过滤显示的数据库名或表名。

```
show metadata lock; # 显示该节点上所有持有metadata lock的连接
show metadata lock xxx_db; # 显示该节点上 xxx_db 中所有持有metadata lock的连接
show metadata lock xxx_db.tb_name; # 显示该节点上 xxx_db 中 tb_name 上所有持有metadata lock的连接
```

示例

```
mysql> show metadata lock;
+---------+--------+----------------+--------------------+-------------+-----------------+----------------+------
----+-------------------------------------+-----------------------------------------------+
| CONN_ID | TRX_ID | TRACE_ID       | SCHEMA             | TABLE       | TYPE            | DURATION       | VALIDATE
| FRONTEND                    | SQL                                 |
+---------+--------+----------------+--------------------+-------------+-----------------+----------------+------
----+-------------------------------------+-----------------------------------------------+
| 4       |      0 | f88cf71cbc00001 | XXXX_DRDS_LOCAL_APP | full_gsi_ddl | MDL_SHARED_WRITE | MDL_TRANS
ACTION |       1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54788 | insert into `full_gsi_ddl` (id) VALUE (null); |
| 5       |      0 | f88cf71cbc00000 | XXXX_DRDS_LOCAL_APP | full_gsi_ddl | MDL_SHARED_WRITE | MDL_TRANS
ACTION |       1 | XXXX_DRDS_LOCAL_APP@127.0.0.1:54789 | insert into `full_gsi_ddl` (id) VALUE (null); |
+---------+--------+----------------+--------------------+-------------+-----------------+----------------+------
----+-------------------------------------+-----------------------------------------------+
2 rows in set (0.00 sec)
```

> **说明** 该语句仅用于显示已持有锁的连接，不显示等待锁的连接。

列名说明

列名	说明
CONN_ID	持有锁的连接ID
TRX_ID	持有锁的事务ID
TRACE_ID	持有锁的SQL的跟踪 ID
SCHEMA	库名
TABLE	表名
TYPE	持有锁类型
DURATION	持有锁的周期
VALIDATE	是否有效
FRONTEND	前端连接信息
SQL	持有锁的SQL语句

5.2. KILL

您可以使用KILL指令终止一个正在DRDS上执行的SQL。

前提条件

终止一个DRDS上正在执行的SQL前，您需要先连接DRDS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DRDS，详情请参见步骤四：连接DRDS。

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` 。
> - DRDS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DRDS上所有正在执行的物理SQL:

```
KILL 'ALL'
```

5.3. USE

本文将介绍如何通过USE指令在DRDS中切换当前连接的默认数据库。

背景信息

DRDS支持访问同一DRDS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，DRDS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
```

6.Sequence

7.Outline

8.Hint

8.1. INDEX HINT

DRDS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DRDS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

DRDS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DRDS自定义HIN的SQL时，请在登录命令中加上-c参数，否则由于DRDS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DRDS自定义HINT失效，详情请参见MySQL官方客户端命令。

语法

DRDS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
```

9.函数

9.1. 加密和压缩函数

本文主要介绍DRDS支持和不支持的加密和压缩函数。

支持的加密和压缩函数

DRDS支持如下加密和压缩函数。

函数名	描述
AES_ENCRYPT()	Encrypt 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，DRDS暂不支持如下加密和压缩函数。

函数名	描述
AES_DECRYPT()	Decrypt using AES
AES_ENCRYPT()	Encrypt using AES
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()
DES_DECRYPT() (deprecated 5.7.6)	Decrypt a string
DES_ENCRYPT() (deprecated 5.7.6)	Encrypt a string

函数名	描述
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

10.运算符

11.数据类型

12.Prepare SQL

13.实用 SQL 语句

13.1. 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，将会检查索引和主表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：进行索引订正，以主表为基准。

> 说明
> - 对全局二级索引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分批加锁校验并订正，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纯异步DDL模式执行，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":"OTkAAAAAAAAAAA==","c_int_32":271}}
|
| `g_i_check` | CONFLICT   | FOUND  | (108710)    | {"Primary":{"id":108710,"c_timestamp_6":"2000-01-01 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_year":"2000","c_int_32":255},"GSI":{"c_int_32_un":123456,"id":108710,"c_timestamp_6":"2000-01-01 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_year":"2000","c_int_32":255}} |
| `g_i_check` | MISSING    | FOUND  | (100090)    | {"Primary":{"id":100090,"c_timestamp_6":"2000-01-01 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0","c_blob_tiny":"YeS4reWbvWE=","c_int_32":280}}
|
| `g_i_check` | SUMMARY    | --     | --          | 3 error found (7025/7025 rows checked)
|
+-------------+-----------+--------+------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
4 rows in set (1.92 sec)
```

说明 如果数据存在多种错误，同一行数据会报多个ERROR_TYPE。

- 您可以使用如下语句进行订正：

```
mysql> CHECK GLOBAL INDEX g_i_check CORRECTION_BASED_ON_PRIMARY;
```

返回结果如下：

```
+-------------+------------+--------+-------------+-------------------------------------------------------------------------------
-------+
| GSI_TABLE   | ERROR_TYPE | STATUS | PRIMARY_KEY | DETAILS                                                                       |
+-------------+------------+--------+-------------+-------------------------------------------------------------------------------
-------+
| `g_i_check` | SUMMARY    | --     | --          | Done. Use SQL: { CHECK GLOBAL INDEX `g_i_check` SHOW; }
to get result. |
+-------------+------------+--------+-------------+-------------------------------------------------------------------------------
-------+
1 row in set (1.40 sec)
```

- 您可以使用如下语句查看最近一次校验或订正的报告：

```
mysql> CHECK GLOBAL INDEX `g_i_check` SHOW;
```

返回结果如下：

```
+-------------+-----------+----------+------------+-----------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
| GSI_TABLE   | ERROR_TYPE | STATUS   | PRIMARY_KEY | DETAILS
|
+-------------+-----------+----------+------------+-----------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
| `g_i_check` | MISSING   | REPAIRED | (100090)   | {"Primary":{"id":100090,"c_timestamp_6":"2000-01-01
00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.0"
,"c_blob_tiny":"YeS4reWbvWE=","c_int_32":280}}
|
| `g_i_check` | CONFLICT   | REPAIRED | (108710)   | {"Primary":{"id":108710,"c_timestamp_6":"2000-01-0
1 00:00:00.000000","c_timestamp_3":"2000-01-01 00:00:00.000","c_timestamp_1":"2000-01-01 00:00:00.
0","c_year":"2000","c_int_32":255},"GSI":{"c_int_32_un":123456,"id":108710,"c_timestamp_6":"2000-01-0
1 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":"OTkAAAAAAAAAAA==","c_int_32":271}}
|
| `g_i_check` | SUMMARY   | --       | --         | 3 error found (7025/7026 rows checked.) Finish time: 2020
-01-13 14:41:51.0
|
+-------------+-----------+----------+------------+-----------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------+
4 rows in set (0.02 sec)
```

列名说明

列名	说明
GSI_TABLE	全局二级索引名。

列名	说明
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。

14.多语句

本文将介绍DRDS多语句的相关信息。

DRDS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。
> - DRDS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.跨Schema

DRDS实例中通常有多个Schema，DRDS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。
- DRDS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：DRDS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：DRDS数据库库名，同一个实例内的数据库库名具有唯一性。
- Table: DRDS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: DRDS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DRDS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
- Sequences
- DAL
- USE