TDSQL MySQL版（私有云）

TDSQL分布式开发规范

产品文档

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100。

文档目录

TDSQL分布式开发规范

概述

最近更新时间：2021-10-18 17:10:03

文档说明

本手册涵盖TDSQL数据库设计、SQL语句开发编写等内容。设计部分，包括库、表、列、索引的原则。开发部分，包括SQL书写的规范、连接方式等方面的内容。目的是提高应用开发数据库设计效率、减少因为数据库设计不规范导致的问题。

规范级别

本文所有规范会按照【禁止】、【建议】两个级别进行标注，遵守优先级从高到低。

范围

本手册适用于使用TDSQL分布式实例的应用开发人员、数据库应用设计人员、数据库管理员等。
本手册适用于TDSQL10.3.16.2.2版本。

产品术语

最近更新时间：2021-10-18 17:10:19

- 节点：Set或称为数据节点、分片。基于MySQL数据库主从协议联结成若干组。Set 是分布式实例中最小数据单元。每个set 内部都具有一主N备的高可用架构。一个分布式实例是由N个Set组成，每个Set 中存有不同范围的数据，所有set 加到一起是一份全量的数据
- 分片键：根据分片键把一份全量数据进行切分，每份数据称为数据分片
- 分布式实例：Group_Shard，数据分布在n个set 上面。也可以简称为shard
- 集中式实例：No_shard，即非分布式实例，所有数据都在一个set上
- 分片表：即水平拆分表（又名Shard表）；分表需指定一个字段，使用不同的分片算法（hash、list、range），将数据分布到不同的set当中。hash分片算法使用shardkey语法，list和range分片算法采用tdsql_distributed by语法
- 单片表：又名Noshard表，用于存储一些无需分片的表，该表的数据全量存在第一个物理分片（set）中。所有单片表的数据都放在第一个物理分片（set）中。由于单片表默认放置在第一个 set 上，如果在分布式实例中建立了大量的单片表，则可能导致第一个 set 的负载太大
- 广播表：又名小表广播，该表的所有操作都将广播到所有节点（set）中，每个 set 都有该表的全量数据，常用于业务系统的配置表等
- 一级分区表：分片表的同义词

SQL语句语法的使用
DDL语句
创建表

最近更新时间：2021-10-18 17:10:39

TDSQL分布式实例支持创建分表、单表和广播表。分表即自动水平拆分的表（Shard表），水平拆分是基于分表键采用类似于一致性 Hash、Range、List等方式，根据计算后的值分配到不同的节点组中的一种技术方案。可以将满足对应条件的行将存储在相同的物理节点组中。这种场景称为组拆分（Groupshard）,可以迅速提高应用层联合查询等语句的处理效率。TDSQL支持LIST、RANGE、HASH三种类型的一级分区，同时支持支持RANGE、LIST两种格式的二级分区。
【建议】如无特殊要求，建议用户在分布式实例中创建分表进行使用。

一级分区表

在TDSQL中，分表也叫一级分区表。有hash、range、list三种规则。一级hash分区使用shardkey关键字指定拆分键。range和list分区使用tdsql_distributed by语法指定拆分键。

一级HASH分区

- 一级hash分区支持类型
  - DATE，DATETIME
  - TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
  - CHAR，VARCHAR

```
mysql> create table test1 (a int, b int, c char(20),primary key (a,b),unique key u_1(a,c) ) sha
rdkey=a;
Query OK, 0 rows affected (0.07 sec)
```

在分布式实例中，Shardkey对应后端数据库的分区字段，因此必须是主键以及所有唯一索引的一部分，否则无法创建表。详见如下：

```
mysql> create table test1 ( a int, b int, c char(20),primary key (a,b),unique key u_1(a,c),uniq
ue key u_2(b,c) ) shardkey=a;
```

此时有一个唯一索引u_2不包含shardkey，无法创建表，将报如下错误：

```
ERROR 1105 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning fu
nction
```

因为主键索引或者unique key索引需要全局唯一，而要实现全局唯一索引则必须包含shardkey字段。
Shardkey字段设计与使用原则：

- 【建议】Shardkey 字段必须是主键以及所有唯一索引的一部分
- 【建议】Shardkey字段的类型必须为int，bigint，smallint/char/varchar，如果分区键是char或者varchar类型，建议长度不超255
- 【禁止】Shardkey字段的值不能为中文，因为Proxy不会转换字符集，所以不同字符集可能会路由到不同的分区
- 【禁止】不要更新shardkey字段的值
- 【建议】Shardkey=a 需放在SQL语句的最后
- 【建议】访问的数据尽量包含Shardkey字段，否则不带Shardkey字段的SQL语句会路由到所有节点，将消耗较多资源

> ⚠ 注意：
> ：部分分表方案支持“非主键或唯一索引”成为 Shardkey字段，但此类方案会导致数据不一致，因此 TDSQL 默认禁止“非主键或唯一索引”成为 Shardkey字段。

一级RANGE分区

- 一级range分区支持类型
  - DATE，DATETIME，TIMESTAMP
  - TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT
  - CHAR，VARCHAR

```
create table t1(a int key, b int) tdsql_distributed by range(a) (s1 values less than(100), s2 val
ues less than(200));
```

【禁止】避免使用TIMESTAMP类型作为分区键，因为timestamp受到时区的影响，同时只能使用到2038年
【建议】如果分区键是char或者varchar类型，建议长度不超255

一级LIST分区

- 一级list分区支持类型
  - DATE，DATETIME，TIMESTAMP
  - TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT
  - CHAR，VARCHAR

```
create table t2(a int key, b int) tdsql_distributed by list(a) (s1 values in(1,2), s2 values in (3,4
));
```

【禁止】避免使用TIMESTAMP类型作为分区键，因为timestamp受到时区的影响，同时只能使用到2038年
【建议】如果分区键是char或者varchar类型，建议长度不超255

local_table_option选项

在tdsql_distributed by创建分表语法基础上，可以使用local_table_options来指定其他一些选项。

> ⚠ 注意：
> ：local_table_option不是创建表的关键字，只是占位符

```
CREATE TABLE [IF NOT EXISTS] *tbl_name* (*create_definition*,...) [*local_table_options*] TDSQ
L_DISTRIBUTED BY range|list ....
```

- local_table_option例子

```
CREATE TABLE t1 (
a int(11) NOT NULL,
b int(11) DEFAULT NULL,
PRIMARY KEY (a)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
PARTITION BY LIST (a)
(PARTITION p1 VALUES IN (1,2) ENGINE = InnoDB,
PARTITION p2 VALUES IN (3,4) ENGINE = InnoDB)
TDSQL_DISTRIBUTED BY LIST(b) (s1 values in ('100'),s2 values in ('200'));


CREATE TABLE tb_sub_ev (
id int(11) NOT NULL,
purchased date NOT NULL,
PRIMARY KEY (id,purchased)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
PARTITION BY RANGE ( YEAR(purchased))
```

```
SUBPARTITION BY HASH ( TO_DAYS(purchased))
(PARTITION p0 VALUES LESS THAN (1990)
(SUBPARTITION s0 ENGINE = InnoDB,
SUBPARTITION s1 ENGINE = InnoDB),
PARTITION p1 VALUES LESS THAN (2000)
(SUBPARTITION s2 ENGINE = InnoDB,
SUBPARTITION s3 ENGINE = InnoDB))
TDSQL_DISTRIBUTED BY RANGE(id) (s1 values less than ('100'),s2 values less than ('1000'));

CREATE TABLE t1 (
a int(11) NOT NULL,
b int(11) DEFAULT NULL,
PRIMARY KEY (a)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
TDSQL_DISTRIBUTED BY RANGE(a) (s1 values less than ('100'),s2 values less than ('200'));
```

⚠ 注意：

- 分区键不要求是主键/唯一索引的一部分，此时需要业务自己保证唯一性
- 分区键不是主键列时，受限于实现，执行sql中指定force index primary会报错
- 分区键为字符串时，不要使用中文
- 当需要忽略大小写比较时，可以分区函数中使用upper/lower函数，例如tdsql_distributed by range(lower(b))
- tdsql_distributed by ...语法放置于create table ...的末尾

二级分区表

二级分区是将特定条件的数据进行分区处理，目前TDSQL支持Range和List两种格式的二级分区，具体建表语法和MySQL分区语法类似。

二级RANGE分区

- Range支持类型

- DATE，DATETIME，TIMESTAMP

—支持year，month，day函数，函数为空和day函数一样

- TINYINT, SMALLINT, MEDIUMINT, INT , BIGINT

—支持year，month，day函数，此时传入的值转换为年月日，然后和分表信息进行对比

【禁止】避免使用TIMESTAMP类型作为分区键，因为timestamp受到时区的影响，同时只能使用到2038年

如果插入的Hired是Date类型，则插入后查询到的对应的值格式为 'YYYY-MM-DD' ,一级HASH，二级RANGE分区举例如下：

```
MySQL [test]> CREATE TABLE employees_int (
id INT key NOT NULL,
fname VARCHAR(30),
lname VARCHAR(30),
hired date,
separated DATE NOT NULL DEFAULT '9999-12-31',
job_code INT,
store_id INT
)
shardkey=id
PARTITION BY RANGE ( year(hired) ) (
PARTITION p0 VALUES LESS THAN (1991),
PARTITION p1 VALUES LESS THAN (1996),
PARTITION p2 VALUES LESS THAN (2001)
);

MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(10,'a','b','1989-12-01','1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(11,'c','d','1722-08-24','1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(12,'e','f','1994-03-08','1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(13,'g','h','1998-02-09','1880-12-31',1000,2000);

MySQL [test]> select *,year(hired) from employees_int;
+----+-------+-------+-----------+-----------+---------+---------+------------+
| id | fname | lname | hired | separated | job_code | store_id | year(hired) |
+----+-------+-------+-----------+-----------+---------+---------+------------+
| 11 | a | b | 1722-08-24 | 1880-12-31 | 1000 | 2000 | 1722 |
| 10 | c | d | 1989-12-01 | 1880-12-31 | 1000 | 2000 | 1989 |
| 12 | e | f | 1994-03-08 | 1880-12-31 | 1000 | 2000 | 1994 |
| 13 | g | h | 1998-02-09 | 1880-12-31 | 1000 | 2000 | 1998 |
+----+-------+-------+-----------+-----------+---------+---------+------------+
4 rows in set (0.02 sec)
```

如果插入的Hired是Int类型，则Proxy首先会转换成对应的Date格式，'YYYY-MM-DD'，然后和分表信息进行对比。语句如下：

```
MySQL [test]> CREATE TABLE employees_int (
id INT key NOT NULL,
fname VARCHAR(30),
lname VARCHAR(30),
hired date,
separated DATE NOT NULL DEFAULT '9999-12-31',
job_code INT,
store_id INT
)
shardkey=id
PARTITION BY RANGE ( year(hired) ) (
PARTITION p0 VALUES LESS THAN (1991),
PARTITION p1 VALUES LESS THAN (1996),
PARTITION p2 VALUES LESS THAN (2001)
);

MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(10,'a','b',19891201,'1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(11,'c','d',17220824,'1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(12,'e','f',19940308,'1880-12-31',1000,2000);
MySQL [test]> insert into employees_int(id,fname,lname,hired,separated,job_code,store_id) val
ues(13,'g','h',19980209,'1880-12-31',1000,2000);

MySQL [test]> select *,year(hired) from employees_int;
+----+-------+-------+-----------+-----------+---------+---------+------------+
| id | fname | lname | hired | separated | job_code | store_id | year(hired) |
+----+-------+-------+-----------+-----------+---------+---------+------------+
| 12 | a | b | 1994-03-08 | 1880-12-31 | 1000 | 2000 | 1994 |
| 11 | c | d | 1722-08-24 | 1880-12-31 | 1000 | 2000 | 1722 |
| 10 | e | f | 1989-12-01 | 1880-12-31 | 1000 | 2000 | 1989 |
| 13 | g | h | 1998-02-09 | 1880-12-31 | 1000 | 2000 | 1998 |
```

```
+----+-------+-------+-----------+-----------+---------+---------+------------+
4 rows in set (0.03 sec)
```

> ⚠ 注意：
>
> ：分区使用小于符号“<”，如果要存储当年数据（例如，2017），需要创建小于往后一年（<2018）的分区，用户只需创建到当前的时间分区。TDSQL会自动增加后续分区，默认往后创建3个分区，以Year为例，TDSQL会自动往后创建3年（2018年、2019年、2020年）的分区，后续也会自动增减。

二级LIST分区

- List支持类型
  - DATE，DATETIME，TIMESTAMP —支持年月日函数
  - TINYINT, SMALLINT, MEDIUMINT, INT , BIGINT

  【禁止】禁止使用TIMESTAMP类型作为分区键，因为timestamp受到时区的影响，同时只能使用到2038年

  【建议】如果分区键是char或者varchar类型，建议长度不超255

  一级HASH，二级List分区举例如下：

```
MySQL [test]> CREATE TABLE customers_1 (
first_name VARCHAR(25) key,
last_name VARCHAR(25),
street_1 VARCHAR(30),
street_2 VARCHAR(30),
city VARCHAR(15),
renewal DATE
) shardkey=first_name
PARTITION BY LIST (city) (
PARTITION pRegion_1 VALUES IN('Beijing', 'Tianjin', 'Shanghai'),
PARTITION pRegion_2 VALUES IN('Chongqing', 'Wulumuqi', 'Dalian'),
PARTITION pRegion_3 VALUES IN('Suzhou', 'Hangzhou', 'Xiamen'),
PARTITION pRegion_4 VALUES IN('Shenzhen', 'Guangzhou', 'Chengdu')
);
```

删除和新增二级分区

删除和新增二级分区的格式和单机MySQL一致，语句如下：

```
MySQL [test]> alter table customers_1 drop partition pRegion_1;
```

```
MySQL [test]> alter table customers_1 add partition (partition pRegion_5 VALUES IN('Wuhan',
'Nanjing', 'Guiyang'));
```

> ⚠ 注意：
> TDSQL不支持除Range和List二级分区以外的其他分区操作，例如，Reorganize。

部分二级分区表创建语法举例

一级RANGE，二级LIST创建语法如下：

```
MySQL [test]> CREATE TABLE tb_sub_r_l (
id int(11) NOT NULL,
order_id bigint NOT NULL,
PRIMARY KEY (id,order_id))
PARTITION BY list(order_id)
(PARTITION p0 VALUES in (2121122),
PARTITION p1 VALUES in (38937383))
TDSQL_DISTRIBUTED BY RANGE(id) (s1 values less than (100),s2 values less than (1000));
Query OK, 0 rows affected, 1 warning (0.35 sec)
```

一级RANGE，二级RANGE创建语法如下：

```
MySQL [test]> CREATE TABLE tb_sub_r_r (
id int(11) NOT NULL,
order_id tinyint NOT NULL,
PRIMARY KEY (id,order_id))
PARTITION BY range (order_id)
(PARTITION p0 VALUES less than (5000),
PARTITION p1 VALUES less than (8000))
TDSQL_DISTRIBUTED BY RANGE(id) (s1 values less than (100),s2 values less than (1000));
```

Query OK, 0 rows affected, 1 warning (0.15 sec)

一级LIST，二级RANGE创建语法如下：

```
MySQL [test]> CREATE TABLE t1_sub_l_r (
a int(11) NOT NULL,
b int(11) DEFAULT NULL,
PRIMARY KEY (a)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
PARTITION BY range (a)
(PARTITION p1 VALUES less than (1000) ENGINE = InnoDB,
PARTITION p2 VALUES less than (2000) ENGINE = InnoDB)
TDSQL_DISTRIBUTED BY LIST(b) (s1 values in ('100'),s2 values in ('200'));
```

一级LIST，二级LIST创建语法如下：

```
MySQL [test]> CREATE TABLE t1_sub_l_l (
a int(11) NOT NULL,
b int(11) DEFAULT NULL,
PRIMARY KEY (a)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
PARTITION BY LIST (a)
(PARTITION p1 VALUES IN (1,2) ENGINE = InnoDB,
PARTITION p2 VALUES IN (3,4) ENGINE = InnoDB)
TDSQL_DISTRIBUTED BY LIST(b) (s1 values in (100),s2 values in (200));
```

二级分区表使用建议

业务应尽量都使用一级分区表。二级分区表使用场景和使用方法建议：

1. 使用前根据业务长期场景合理设计表结构，二级分区适用于表结构创建后长期都不需要DDL变更、需要定期进行分区数据清理和裁剪的场景，比如日志流水表
2. 合理设计二级分区的粒度，二级分区的粒度建议不要划分得太细，避免产生过多的二级子表。比如流水表按月进行二级分区，而不是按天/小时进行分区，避免文件系统上数据文件个数过多
3. 在对二级分区表进行SQL查询时，查询条件需要尽量带上一级分区和二级分区的键值，避免执行查询时需要打开很多的数据文件进行搜索
4. 在对二级分区表进行join查询时，如果查询条件未能带上一级分区和二级分区的键值，操作性能效率较低，建议不要使用
5. 表的主键或唯一索引需要包含分区键，否则无法保证数据唯一性

广播表

如何创建广播表

广播表又名小表广播功能，创建时需要指定noshardkey_allset关键字。创建广播表后，每个节点都有该表的全量数据，且该表的所有操作都将广播到所有物理分片（set）中。
广播表主要用于提升跨节点组（ Set）的Join 操作的性能，常用于配置表等，语句如下：

```
MySQL [test]> create table global_table_a ( a int, b int key) shardkey=noshardkey_allset;
Query OK, 0 rows affected (0.05 sec)
```

广播表使用建议

业务应尽量都使用一级分区表。广播表使用场景和使用方法建议：

1. 表的数据量少
2. 该表需要和分片表进行Join操作，可以使用广播表
3. 该表不需要和分片表进行join操作，但是更新量少（仅在变更时修改，例如版本发布），可以使用广播表

单片表

如何创建单片表

普通表：又名单片表（Noshard表），创建时无须指定shardkey或者tdsql_distributed by关键字。单片表无需拆分且没有做任何特殊处理的表。其语法和MySQL完全一样，所有该类型表的全量数据默认存放在第一个物理节点组（Set）中，具体语句如下：

```
MySQL [test]> create table noshard_table (a int, b int key);
Query OK, 0 rows affected (0.21 sec)
```

> 说明：
> 单片表不支持shardkey，并且单表默认放置在第一个物理节点组（Set）中，如果创建过多单表，可能会导致第一个物理节点组（Set）的负载过大。

单片表的使用建议

业务应尽量都使用一级分区表。单片表使用场景和使用方法建议：

1. 表的数据量少
2. 该表不需要和分片表进行Join操作，但是更新量较大（业务处理中会进行修改），可以使用单表

其他DDL语句操作

最近更新时间：2021-10-18 17:10:49

ALTER，DROP 等其他 DDL 语句操作，绝大部分都与 MySQL 语法完全一致，但如4.1.2节中所述，对于ALTER语句修改shardkey的语句会有以下限制：
不支持ALTER对shardkey改名：

```
MySQL [test]> create table t_ddl(id int primary key not null,name char(8),address char(10)) sh
ardkey=id;
Query OK, 0 rows affected (2.17 sec)

MySQL [test]> insert into t_ddl(id,name,address) values(1,'abcdefg','Shenzhen');
Query OK, 1 row affected (0.05 sec)

MySQL [test]> insert into t_ddl(id,name,address) values(2,'gfedcba','Shanghai');
Query OK, 1 row affected (0.05 sec)

MySQL [test]> select id,name,address from t_ddl;
+----+---------+----------+
| id | name | address |
+----+---------+----------+
| 1 | abcdefg | Shenzhen |
| 2 | gfedcba | Shanghai |
+----+---------+----------+
2 rows in set (0.05 sec)

MySQL [test]> /*sets:allsets */ alter table t_ddl change id stu_num int;
ERROR 3855 (HY000): Column 'id' has a partitioning function dependency and cannot be dropp
ed or renamed.
```

不支持将shardkey字段长度缩至实际长度以下：

```
MySQL [test]> create table t_ddl_char(name char(8) not null primary key,address varchar(10))
shardkey=name;
Query OK, 0 rows affected (1.72 sec)
```

```
MySQL [test]> insert into t_ddl_char(name,address) values('abcdefg','Shenzhen');
Query OK, 1 row affected (0.07 sec)

MySQL [test]> insert into t_ddl_char(name,address) values('gfedcba','Shanghai');
Query OK, 1 row affected (0.02 sec)

MySQL [test]> select name,address from t_ddl_char;
+---------+----------+
| name | address |
+---------+----------+
| gfedcba | Shanghai |
| abcdefg | Shenzhen |
+---------+----------+
2 rows in set (0.11 sec)

MySQL [test]> /*sets:allsets */ alter table t_ddl_char modify name char(12);
Query OK, 2 rows affected (5.02 sec)

MySQL [test]>/*sets:allsets */ alter table t_ddl_char modify name char(4);
ERROR 1265 (HY000): Data truncated for column 'name' at row 1
```

支持修改shardkey的类型，例如从char(10)修改至varchar(100)：

```
MySQL [test]> show create table t_ddl_char\G;
*************************** 1. row ***************************
Table: t_ddl_char
Create Table: CREATE TABLE `t_ddl_char` (
`name` varchar(100) COLLATE utf8_bin NOT NULL,
`address` varchar(10) COLLATE utf8_bin DEFAULT NULL,
PRIMARY KEY (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin shardkey=name
1 row in set (0.01 sec)

MySQL [test]> select name,address from t_ddl_char;
+---------+----------+
| name | address |
+---------+----------+
```

```
| abcdefg | Shenzhen |
| gfedcba | Shanghai |
+---------+----------+
2 rows in set (0.00 sec)

MySQL [test]> /*sets:allsets */ alter table t_ddl_char modify name varchar(100);
Query OK, 2 rows affected (5.22 sec)
```

【建议】线上系统的DDL变更请通过赤兔管理控制台的online-ddl模块进行

DML语句

最近更新时间：2021-10-18 17:10:58

本节主要介绍 DML 语句中常用的Select（查询）、Insert（插入）、Replace（替换）、Update（更新）及Delete（删除）指令。

SELECT指令

执行Select指令时，建议在条件中增加Shardkey字段，语句如下。

> 说明：
> Proxy根据该字段的Hash值，将SQL指令请求路由至对应的数据库实例进行处理；否则SQL指令将发送到集群所有的数据库实例，Proxy再进行数据库返回的结果集进行聚合，将影响执行效率。

```
MySQL [test]> create table test1(a int not null primary key,b int,c char(10)) shardkey=a;
Query OK, 0 rows affected (2.64 sec)

MySQL [test]> insert into test1(a,b,c) values(2,3,'record2');
Query OK, 1 row affected (0.04 sec)

MySQL [test]> insert into test1(a,b,c) values(3,4,'record3');
Query OK, 1 row affected (0.03 sec)

MySQL [test]> select b,c from test1 where a=3;
+------+---------+
| b | c |
+------+---------+
| 4 | record3 |
+------+---------+
1 row in set (0.00 sec)
```

INSERT/REPLACE指令

执行Insert/Replace命令时，字段必须包含Shardkey，否则系统会拒绝执行SQL命令，因为Proxy无法判断SQL语句发送的后端数据库节点位置。语句显示如下：

```
MySQL [test]> insert into test1 (b,c) values(10,"record3");
ERROR 683 (HY000): Proxy ERROR: Get shardkeys return error: insert/replace must contain sha
rdkey column

MySQL [test]> insert into test1 (a,c) values(40,"records5");
Query OK, 1 row affected (0.03 sec)


MySQL [test]> truncate table test1;
Query OK, 0 rows affected (2.18 sec)

--重新插入数据后：
MySQL [test]> select a,b,c from test1;
+---+------+---------+
| a | b | c |
+---+------+---------+
| 3 | 4 | record3 |
| 2 | 3 | record2 |
+---+------+---------+
2 rows in set (0.03 sec)

MySQL [test]> replace into test1 (b,c) values(10,"record3");
ERROR 683 (HY000): Proxy ERROR: Get shardkeys return error: insert/replace must contain sha
rdkey column

MySQL [test]> replace into test1(a,b,c) values(3,40,"record1");
Query OK, 2 rows affected (0.03 sec)

MySQL [test]> select a,b,c from test1;
+---+------+---------+
| a | b | c |
+---+------+---------+
| 3 | 40 | record1 |
| 2 | 3 | record2 |
+---+------+---------+
2 rows in set (0.00 sec)
```

DELETE/UPDATE指令

执行Delete/Update命令时，为了安全考虑，分表和广播表执行该 SQL指令的时候必须带“ where ”条件，否则系统拒绝执行该SQL命令。语句如下：

```
MySQL [test]> delete from test1;
ERROR 658 (HY000): Proxy ERROR: Join internal error: delete query has no where clause

MySQL [test]> delete from test1 where a=2;
Query OK, 1 row affected (0.01 sec)
```

【建议】：为了防止用户误操作，建议尽量不要使用全表的Update/Delete指令；如必须使用该指令，可在SQL语句中增加where 1条件：

```
MySQL [test]> delete from test1 where 1;
Query OK, 1 row affected (0.01 sec)

MySQL [test]> select * from test1;
Empty set (0.01 sec)
```

JOIN和子查询功能

最近更新时间：2021-10-18 17:11:07

对于分布式实例，数据水平拆分在各个节点上。为了提高性能，应用层应优先优化表结构和SQL语句，使数据尽量采用不跨节点的方式存储。

- 如果Join 相关的表有分表键相等条件（如下示例），由于分表的一致性原则，会让这部分数据自动存储到同一物理节点，此时相当于单机Join，数据处理效率将更高，因此我们推荐使用join方式替代子查询。
- 如果涉及到跨物理节点数据，此时 Proxy 会先从其他节点拉取数据并缓存，由于涉及到网络数据传输，将降低数据处理效率。

推荐join方式

分表之间

如果分表之间带有分表键相等的条件，则相当于单机Join。语句如下：

```
MySQL [test]> create table test1(a int not null primary key,b int,c char(20)) shardkey=a;

Query OK, 0 rows affected (2.64 sec)

MySQL [test]> create table test2(a int not null primary key,b int,c char(20)) shardkey=a;
Query OK, 0 rows affected (2.28 sec)

MySQL [test]> insert into test1(a,b,c) values(1,2,'test1_record1');
Query OK, 1 row affected (0.01 sec)

MySQL [test]> insert into test1(a,b,c) values(2,3,'test1_record2');
Query OK, 1 row affected (0.03 sec)

MySQL [test]> insert into test2(a,b,c) values(1,200,'test2_record1');
Query OK, 1 row affected (0.01 sec)

MySQL [test]> insert into test2(a,b,c) values(2,300,'test2_record2');
Query OK, 1 row affected (0.07 sec)

MySQL [test]> select a,b,c from test1;
+---+------+--------------+
```

```
| a | b | c |
+---+------+--------------+
| 1 | 2 | test1_record1 |
| 2 | 3 | test1_record2 |
+---+------+--------------+
2 rows in set (0.01 sec)

MySQL [test]> select a,b,c from test2;
+---+------+--------------+
| a | b | c |
+---+------+--------------+
| 1 | 200 | test2_record1 |
| 2 | 300 | test2_record2 |
+---+------+--------------+
2 rows in set (0.01 sec)

MySQL [test]> select t1.a,t1.b,t1.c,t2.a,t2.b,t2.c from test1 t1 join test2 t2 where t1.a=t2.a;
+---+------+--------------+---+------+--------------+
| a | b | c | a | b | c |
+---+------+--------------+---+------+--------------+
| 1 | 2 | test1_record1 | 1 | 200 | test2_record1 |
| 2 | 3 | test1_record2 | 2 | 300 | test2_record2 |
+---+------+--------------+---+------+--------------+
2 rows in set (0.01 sec)

MySQL [test]> select t1.a,t1.b,t1.c,t2.a,t2.b,t2.c from test1 t1 left join test2 t2 on t1.a<t2.a wh
ere t1.a=1;
+---+------+--------------+------+------+--------------+
| a | b | c | a | b | c |
+---+------+--------------+------+------+--------------+
| 1 | 2 | test1_record1 | 2 | 300 | test2_record2 |
+---+------+--------------+------+------+--------------+
1 row in set (0.00 sec)
```

分表和广播表

跨分片的分表与广播表，效果相当于单机 Join。语句如下：

```
MySQL [test]> create table global_test(a int key, b int) shardkey=noshardkey_allset;
Query OK, 0 rows affected (0.07 sec)

MySQL [test]> insert into global_test(a, b) values(1,1),(2,2);
Query OK, 2 rows affected (0.05 sec)

MySQL [test]> select a,b from global_test;
+---+------+
| a | b |
+---+------+
| 1 | 1 |
| 2 | 2 |
+---+------+
2 rows in set (0.02 sec)

MySQL [test]> select * from test1;
+---+------+--------------+
| a | b | c |
+---+------+--------------+
| 1 | 2 | test1_record1 |
| 2 | 3 | test1_record2 |
+---+------+--------------+
2 rows in set (0.00 sec)

MySQL [test]> select t1.a,t1.b,t1.c,tg.a,tg.b from test1 t1,global_test tg where t1.a=tg.a;
+---+------+--------------+---+------+
| a | b | c | a | b |
+---+------+--------------+---+------+
| 1 | 2 | test1_record1 | 1 | 1 |
| 2 | 3 | test1_record2 | 2 | 2 |
+---+------+--------------+---+------+
2 rows in set (0.00 sec)
```

复杂SQL查询

对于不满足推荐方式的SQL语句，因需要做跨节点的数据交互，将会导致性能变差，可能影响以下查询活动：

- 包含子查询的查询。
- 带有Having条件的查询。
- 需要进行多个排序/分组/去重的查询，例如：Count (Distinct ID)。
- 多表的Join查询，且参与查询的各表的分区字段(Shardkey)不相等，或者同时涉及不同类型的表（例如，单表和分表）。
  对于此类复杂查询，可以通过条件下推，将从后端数据库中抽取出参与查询的数据，并存放在本地临时表中，通过临时表中的数据进行计算。

因此用户需要指定参与查询的表的条件，避免因抽取大量数据而导致性能受损。相关语句如下：

```
mysql> create table test1 ( a int key, b int, c char(20) ) shardkey=a;
Query OK, 0 rows affected (1.56 sec)

mysql> create table test2 ( a int key, d int, e char(20) ) shardkey=a;
Query OK, 0 rows affected (1.46 sec)

mysql> insert into test1 (a,b,c) values(1,2,"record1"),(2,3,"record2");
Query OK, 2 rows affected (0.02 sec)

mysql> insert into test2 (a,d,e) values(1,3,"test2_record1"),(2,3,"test2_record2");
Query OK, 2 rows affected (0.02 sec)

MySQL [test]> select t1.a,t1.b,t1.c,t2.a,t2.d,t2.e from test1 t1 join test2 t2 on t1.b=t2.d;
+---+------+---------+---+------+--------------+
| a | b | c | a | d | e |
+---+------+---------+---+------+--------------+
| 2 | 3 | record2 | 1 | 3 | test2_record1 |
| 2 | 3 | record2 | 2 | 3 | test2_record2 |
+---+------+---------+---+------+--------------+
2 rows in set (0.00 sec)

MySQL [test]> select t1.a,t1.b,t1.c from test1 t1 where t1.a in (select a from test2);
+---+------+---------+
| a | b | c |
+---+------+---------+
| 1 | 2 | record1 |
| 2 | 3 | record2 |
```

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

MySQL [test]> select t1.a,t1.b,t1.c from test1 t1 where exists (select t2.a,t2.d,t2.e from test2 t
2 where t2.a=t1.b);
+---+------+---------+
| a | b | c |
+---+------+---------+
| 1 | 2 | record1 |
+---+------+---------+
1 row in set (0.00 sec)

MySQL [test]> select t1.a, count(1) from test1 t1 where exists (select t2.a,t2.d,t2.e from test2 t
2 where t2.a=t1.a) group by t1.a;
+---+----------+
| a | count(1) |
+---+----------+
| 1 | 1 |
| 2 | 1 |
+---+----------+
2 rows in set (0.03 sec)

MySQL [test]> select distinct count(1) from test1 t1 where exists (select t2.a,t2.d,t2.e from test
2 t2 where t2.a=t1.a) group by t1.a;
+----------+
| count(1) |
+----------+
| 1 |
+----------+
1 row in set (0.02 sec)

MySQL [test]> select count(distinct t1.a) from test1 t1 where exists (select t2.a,t2.d,t2.e from t
est2 t2 where t2.a=t1.a);
+--------------------+
| count(distinct t1.a) |
+--------------------+
| 2 |
+--------------------+
```

```
1 row in set (0.00 sec)
```

预处理协议

最近更新时间：2021-10-18 17:11:14

TDSQL 支持预处理协议，使用方式与单机 MySQL 相同，例如：

- PREPARE Syntax
- EXECUTE Syntax

二进制协议的支持：

- COM_STMT_PREPARE
- COM_STMT_EXECUTE

举例如下：

```
MySQL [test]> create table test1(a int not null primary key,b int) shardkey=a;
Query OK, 0 rows affected (1.71 sec)

MySQL [test]> insert into test1(a,b) values(5,6),(3,4),(1,2);
Query OK, 3 rows affected (0.06 sec)
Records: 3 Duplicates: 0 Warnings: 0

MySQL [test]> select a,b from test1;
+---+------+
| a | b |
+---+------+
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |
+---+------+
3 rows in set (0.02 sec)

mysql> prepare ff from "select a,b from test1 where a=?";
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> set @aa=3;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> execute ff using @aa;
+---+------+
| a | b |
+---+------+
| 3 | 4 |
+---+------+
1 row in set (0.06 sec)
```

说明：

目前TDSQL只对Prepare/Execute命令做语法兼容，从性能角度的话，在分布式下建议用户尽量不要使用该种方式，直接使用文本协议。

注释透传功能

最近更新时间：2021-10-18 17:11:20

注释透传指支持透传 SQL语句到对应的一个或者多个物理分片（Set），并透传到分表键（Shardkey）对应的分片（Set）中的操作方式。对于分布式实例，Proxy 会对 SQL进行语法解析，但有比较严格的限制，如果用户想在某个物理分片（set）中执行SQL语句，可以使用该功能。
具体语法如下：

```
/*sets:set_1*/
/*sets:set_1,set_2*/ （set名字可以通过/*proxy*/show status查询）
/*sets:allsets */
```

透传功能演示如下：

```
MySQL [test]> create table test1 ( a int key , b int, c char(20) ) shardkey=a;
Query OK, 0 rows affected (1.71 sec)

MySQL [test]> select count(*) from test1;
+----------+
| count(*) |
+----------+
| 300 |
+----------+
1 row in set (0.12 sec)

MySQL [test]> select count(*) from test1;
+----------+
| count(*) |
+----------+
| 300 |
+----------+
1 row in set (0.11 sec)

MySQL [test]> /*sets:allsets */ select count(*) from test1;
+----------+-----------------+
| count(*) | info |
+----------+-----------------+
```

```
| 150 | set_1619374020_1 |
| 150 | set_1619508344_3 |
+---------+-----------------+
2 rows in set (0.02 sec)

MySQL [(none)]> /*proxy*/ show status;
+--------------------------+-----------------------------------------+
| status_name | value |
+--------------------------+-----------------------------------------+
| cluster | group_1619373877_13 |
| set_1619374020_1:ip | 10.0.0.17:4007;s1@10.0.0.16:4007@1@IDC1@0 |
| set_1619374020_1:alias | s1 |
| set_1619374020_1:hash_range | 0---31 |
| set_1619508344_3:ip | 10.0.0.17:4008;s1@10.0.0.16:4008@1@IDC1@0 |
| set_1619508344_3:alias | s2 |
| set_1619508344_3:hash_range | 32---62 |
| set | set_1619374020_1,set_1619508344_3 |
+--------------------------+-----------------------------------------+
8 rows in set (0.00 sec)

MySQL [test]> /*sets:set_1619374020_1*/ select count(*) from test1;
+---------+-----------------+
| count(*) | info |
+---------+-----------------+
| 150 | set_1619374020_1 |
+---------+-----------------+
1 row in set (0.04 sec)

MySQL [test]> /*set_1619508344_3*/ select count(*) from test1;
+----------+
| count(*) |
+----------+
| 150 |
+----------+
1 row in set (0.11 sec)


MySQL [test]> delete from test1;
```

```
ERROR 913 (HY000): Proxy ERROR:Join internal error: delete query has no where clause

MySQL [test]> /*sets:allsets*/delete from test1;
Query OK, 300 rows affected (0.04 sec)
```

禁止：

非必要情况下，避免普通用户使用透传功能对数据进行增删改。因为透传SQL语句进行写操作时，因Proxy不解析SQL语句，所以如果往两个及以上节点组（Set）进行透传写操作，系统将不使用分布式事务，可能导致数据不一致，因此对于写操作建议一次透传一个节点组（set）

应用架构设计
设计通则

最近更新时间：2021-10-25 15:56:26

TDSQL是一款针对OLTP场景设计的分布式数据库产品，适用于高并发的联机访问。此类场景下，数据库的使用思路是尽量降低数据库的CPU和IO负载，将没有必要由数据库进行的计算上移到应用层处理，通过应用层横向扩展提升性能，使数据库最基本的增删改查能满足业务要求。

- 【不建议】不在TDSQL数据库端做运算，比如：md5()、sha()等
- 【不建议】不在数据库中存储图片
- 【建议】不在应用程序端显示的加锁
- 【不建议】不使用未经验证的，不明确的新功能
- 【建议】OLTP 应用尽量避免大事务，OLTP 和 OLAP 应用分离，尽量不要在 OLTP 数据库上进行全文检索、统计查询操作
- 【建议】应用需要有失败重连
- 【建议】对于有连接池的前端程序，必须根据业务需要配置初始、最小、最大连接数，超时时间（最大 1 秒）以及连接回收机制（最大 3600 秒）
- 【建议】新项目上线或旧项目重构前，一定要做数据量/访问量/数据增量/访问增长的评估，预先估计那些表会是性能瓶颈，如果访问量大，需要合理的前端架构，缓存应用数据减少数据库压力
- 【建议】一个数据表中，如果有部分字段需要频繁更新，而其他字段不需要，那么建议把它们拆分到不同表中，以提升更新的效率
- 【建议】程序端日志必须记录连接数据库的标准 MySQL 错误号以及所连接的数据库信息（比如 IP 和 PORT，数据库用户名），用于后续排查错误
- 【建议】读写分离，尽可能减小主库的负载，提升架构的整体性能

读写分离

最近更新时间：2021-10-18 17:11:38

读写分离的模式

读写分离的基本原理是让主节点 (Master) 处理事务性操作，例如：增加（INSERT）、更新（UPDATE）、删除（DELETE）操作，让从节点 (Slave) 处理查询（SELECT）操作，将读写功能分开处理，以降低主节点处理数据的压力。
TDSQL 默认支持读写分离功能，架构中的每个从机都支持只读能力，如果配置有多个从机，将由网关集群自动分配到低负载从机上，以支撑大型应用程序的读取流量。
支持以下4种模式的读写分离操作。

- 创建只读账号：您仅需要在创建帐号时，标记为只读帐号，系统将根据只读策略向将读请求发往从机；只读策略可以根据主从延迟等维度进行灵活配置
- 配置Proxy参数：开启语法解析的配置，通过语法解析过滤出用户选择的读请求，默认把读请求直接发给备机
- slave注释：在编程过程中，通过增加Slave注释模式标记，将指定的SQL指令发往备机。即在SQL中添加/*slave*/标记，该SQL会发送给备机，常用于编程阶段将特殊的读逻辑嵌入代码

> 说明：
> 系统还支持/*slave:slaveonly*/,/*slave:20*/,/*slave:slaveonly,20*/三种形式。数值表示备机（Slave）应该满足的延迟；Slaveonly表示在没有符合条件的备机（Slave）时，不会将查询发送给主节点

- 全局自动读写分离：该配置会自动将 SQL 中的所有读请求发向从机，且能识别事务、存储过程中的读语法并灵活处理。当然如果从机延迟较大，全局自动读写分离并不具备应对策略，读到的数据可能会有延迟，若使用该功能，性能有大约50%的下降。该功能默认未开启。

> 说明：
> 读写分离由此为应用提高总的读取吞吐量。通过多种只读方案的组合，可以配置出复杂的只读方案，以满足您各种业务需求和开发的灵活性。

- 【建议】推荐采用只读账号或Slave注释的方式实现读写分离
- 【建议】对于主从延迟严格敏感的 select 语句，可以从主库读取
- 【建议】当主库CPU使用率大于60%时，可考虑将部分对数据一致性要求没有那么高的业务开启读写分离

读写分离故障的影响

- proxy故障对读写分离的影响：非全部proxy故障无影响
- DB节点故障，对读写分离的影响：
  主库故障触发主从切换，影响只读
  备库故障，根据只读账号属性不同，直接返回读异常或路由到主节点
- 其他事件、故障 对读写分离的影响：只看结果的话，如果是造成了实例异常(无法访问或延迟超过阈值，同2；否则无影响)

数据库安全规范

最近更新时间：2021-10-25 15:56:43

- 【不建议】严禁TDSQL服务器的机器开启公网网卡
- 【不建议】严禁在TDSQL服务器上跑其他应用程序
- 【不建议】分离TDSQL服务器生产环境、开发测试环境，保证数据库环境一致，特别是字符集
- 【不建议】严禁应用程序通过 root 账户连接数据库
- 【不建议】严禁应用用户的 host 设置为'%'，必须限制到网段或者具体 IP
- 【不建议】数据库应用账户只允许访问应用数据库，常规情况应用账号只需要 select，insert，update，delete，create temporary tables，execute权限。生产环境DDL仅由指定人员实施。对于读写分离读应用及统计账户，单独建立只读账户
- 【不建议】设置复杂的数据库账号密码，禁止使用简单密码，禁止在数据库中存储任何形式的明文密码

设计规范
数据库设计

最近更新时间：2021-10-25 15:57:14

库

- 【建议】TDSQL服务器默认存储引擎必须设置为 InnoDB
- 【建议】InnoDB 存储引擎，事物隔离级别使用 Read Committed
- 【建议】创建数据库时必须显式指定字符集，并且字符集只能是 utf8mb4
- 【建议】同一个数据库中所有表、字段必须使用相同的字符集，应保证应用程序连接、数据库、表、字段字符集一致
- 【建议】库的名称必须控制在 32 个字符以内
- 【建议】必须统一用英文小写字母命名数据库名,不得使用中文命名，不得使用符号（下划线除外）
- 【建议】不同应用放到不同 db 中
- 【建议】库的名称格式：业务系统名称_子系统名，同一模块使用的表名尽量使用统一前缀

表

- 【建议】表名 <= 32 个字符 ，只能使用字母小写、数字和_
- 【建议】表必须有主键，且主键值禁止被更新
- 【建议】建表必须显式指定存储引擎engine=innodb
- 【建议】必须有表级别comment
- 【不建议】活跃表中不建议使用 blob、text、varchar(>255) 等大字段
- 【建议】多个表中同一类型与含义字段类型与属性必须相同
- 【不建议】不使用外键做数据一致性保证，从业务上控制
- 【建议】对重要数据 userid、orderid 等业务关键特性字段，除了从应用层面控制数据唯一性，从数据库层面增加唯一性检查，最终保证数据完整性
- 【不建议】对于财务货币数据，不建议使用float和double浮点类型，这两种浮点类型无法确保精度，很容易产生误差。建议使用DECIMAL类型或者将对应金额转换成分或厘，使用整数类型（如int或者bigint）去存储。
- 【建议】备份表名字格式 原表名*年月日 例如： test*20210510
- 【建议】建表必须显式指定字符集default charset=utf8mb4
- 【建议】表包含创建时间字段 create_time和最后更新时间字段 update_time
- 【不建议】不建议使用 MySQL已有的关键字，MySQL关键字参考：https://dev.mysql.com/doc/refman/8.0/en/keywords.html

列

- 【建议】名称必须控制在 32 个字符 只能使用字母小写、数字和_
- 【建议】添加列级别 comment
- 【建议】必须添加字段 NOT NULL 属性，业务可以根据需要定义 DEFAULT 值
- 【不建议】不建议列级别自定义字符集与校对规则, 使用表级默认
- 【建议】业务中选择性很少的状态 status、类型 type 等字段推荐使用 tinytint 或者smallint
- 【不建议】不建议使用 enum，set，使用 tinyint 或 smallint 代替，因为增加新的枚举值属于ddl操作
- 【建议】多个数据表中都有相同属性字段时，其属性需保持一致，减少类型转换开销
- 【建议】仅仅当字符数量可能超过20000个的时候，才可以使用TEXT类型来存放字符类 数据，因为所有TDSQL 数据库都会使用UTF8MB4字符集。所有使用TEXT 类型的字段必须和原表进行分拆，与原表主键单独组成另外一个表进行存放
- 【不建议】不在数据库中存储图片
- 【建议】varchar 长度尽量小于 100，活跃表中不要使用varchar(>255) 等大字段
- 【建议】业务中 IPV4 地址字段推荐使用 int 类型，不推荐用 char(15)
- 【建议】尽量使用字段固有属性，不要用 varchar 代替 int 或 datetime

索引

- 【建议】名称规则 == 前缀 + 后缀
  1.前缀
  主键 pk_
  唯一键 uniq_
  普通索引 idx_
  2.后缀
  一律使用字段小写的名称
  多个字段使用_连接
- 【建议】单个索引中每个索引记录的长度不能超过 64KB，索引列太长，可以使用 prefix 列创建 index
- 【建议】在建立索引时，多考虑建立联合索引，并把区分度最高的字段放在最前面。如列 userid 的区分度可由 select count(distinct userid)计算出来
- 【建议】建表或加索引时，保证表里互相不存在冗余索引。对于TDSQL来说，如果表里已经存在 key(a,b)，则 key(a)为冗余索引，需要删除

SQL编写

最近更新时间：2021-10-25 15:57:22

DML语句

- 【不建议】SELECT 语句必须指定具体字段名称，禁止写成select *，因为select * 会造成不必要的IO和网络开销，无法使用覆盖索引，表结构变更业务会受影响
- 【不建议】insert 语句指定具体字段名称，不要写成 insert into t1 values(…)，因为后期如果表字段做了变更，但应用层没有及时更新的话，系统会报错
- 【建议】事务涉及的表必须全部是 innodb 表。否则一旦失败不会全部回滚，且易造成主从库同步中断
- 【不建议】事务中，不要使用SELECT … FOR UPDATE 语法，它会扩大意向锁范围，较大程度影响数据库的并发事务效率
- 【建议】单个事务操作的行数不超过5000行
- 【建议】除静态表或小表（XXX行以内），DML 语句必须有 where 条件，且使用索引查找
- 【建议】生产环境禁止使用 hint，如 sql_no_cache，force index，ignore key，straightjoin 等。因为 hint 是用来强制 SQL 按照某个执行计划来执行，但随着数据量变化我们无法保证自己当初的预判是正确的
- 【建议】where 条件里等号左右字段类型必须一致 ，否则无法利用索引
- 【建议】WHERE 子句中禁止只使用全模糊的 LIKE 条件进行查找，必须有其他等值或范围查询条件，否则无法利用索引
- 【建议】事务里批量更新数据需要控制数量，进行必要的 sleep，做到少量多次
- 【建议】SELECT|UPDATE|DELETE|REPLACE 要有 WHERE 子句，且 WHERE 子句的条件必需使用索引查找
- 【不建议】索引列不要使用函数或表达式，否则无法利用索引
- 【建议】减少使用 or 语句，可将 or 语句优化为 union，然后在各个 where 条件上建立索引。如 where a=1 or b=2 优化为 where a=1… union …where b=2, key(a),key(b)
- 【建议】要返回 MySQL 自增序列的 ID 值，可以考虑使用函数 LAST_INSERT_ID()，此函数只能返回同一个 SESSION 最近一次对有 AUTO_INCREMENT 属性表 INSERT 的 ID 值
- 【建议】使用 covering index 提高性能（index key 包含所要查询的数据）
- 【建议】分页查询，当 limit 起点较高时，可先用过滤条件进行过滤，比如：
  select a,b,c from t1 limit 10000,20;
  ==> select a,b,c from t1 where id>10000 limit 20;
  具体条件需要根据 sql调整

多表连接

- 【不建议】不建议在业务的更新类 SQL 语句中使用 join，比如 update t1 join t2…
- 【建议】建议将子查询 SQL 拆开结合程序多次查询，或使用 join来代替子查询

- 【建议】线上环境，多表 join 不要超过 3个表
- 【建议】在多表 join 中，尽量选取结果集较小的表作为驱动表，来 join 其他表
- 【建议】Join 操作，确保第二个表（探针表的关联列存在 index）
- 【建议】Join 操作，尽量确保 group by 或 order by 子句中列只参考一个表中的列

事务

- 【建议】单个事务中 INSERT|UPDATE|DELETE|REPLACE 语句操作的行数控制在 5000行 以内
- 【建议】事务里包含 SQL 不超过 5 个（支付业务除外）。因为过长的事务会导致锁数据较久，MySQL 内部缓存、连接消耗过多等问题
- 【建议】事务里更新语句尽量基于主键或 unique key，如 update … where id=XX; 否则会产生间隙锁，内部扩大锁定范围，导致系统性能下降，产生死锁
- 【建议】尽量把一些典型外部调用移出事务，如调用 webservice，访问文件存储等，从而避免事务过长
- 【建议】对于 MySQL 主从延迟严格敏感的 select 语句，请开启事务强制访问主库
- 【建议】事务中，不要使用 SELECT … FOR UPDATE 语法，它会扩大意向锁范围，较大程度影响数据库的并发事务效率

分布式事务说明

由于事务操作的数据通常跨多个物理节点，在分布式数据库中，类似方案即称为分布式事务。TDSQL支持普通分布式事务协议和XA分布式事务协议。TDSQL默认支持分布式事务，且对客户端透明，使用户像使用单机事务一样方便。
TDSQL 分布式事务采用两阶段提交算法（2PC）保证事务的原子性（Atomicity）和一致性（Consistency），隔离级别配置为 Read committed, Repeatable read，或 Serializable。
分布式事务使用方法：

```
begin; # 开启事务
... # 跨set的增、删、改、查等非DDL操作
commit; # 提交事务
```

【建议】分布式事务的性能不如单机事务，性能会有一定的损耗。如需使用，需要进行实际测试结果来决定是否使用。

排序和分组

- 【建议】减少使用 order by，和业务沟通能不排序就不排序，或将排序放到程序端去做。order by、group by、distinct 这些语句较为耗费 CPU，数据库的 CPU 资源是极其宝贵的

- 【建议】order by、group by、distinct 这些 SQL 尽量利用索引直接检索出排序好的数据。如 where a=1 order by 可以利用 key(a,b)
- 【建议】包含了 order by、group by、distinct 这些查询的语句，where 条件过滤出来的结果集请保持在 1000 行以内，否则导致 IO/CPU 过高

开发限制项

最近更新时间：2021-10-18 17:12:01

- 禁止使用未经验证的，不明确的新功能
- 禁止在TDSQL数据库中存储图片
- 禁止在TDSQL数据库端做运算，比如：md5()、sha()等
- 对于财务货币数据，禁止使用float和double浮点类型，这两种浮点类型无法确保精度，很容易产生误差。建议使用DECIMAL类型或者将对应金额转换成分或厘，使用整数类型（如int或者bigint）去存储
- 禁止使用关联子查询，如 update t1 set … where name in(select name from user where…);效率极其低下
- 禁用 procedure、function、trigger、views、event、外键约束。因为他们消耗数据库资源，降低数据库实例可扩展性，推荐都在程序端实现
- 禁止联表更新语句，如 update t1,t2 where t1.id=t2.id…
- 禁止使用TIMESTAMP 类型，因为timestamp受到时区的影响，同时只能使用到2038年
- 禁止使用BLOB类型，如果数据页不足以存储整行数据，则InnoDB选择最长的字段，将其存储到一个单独的页中，我们称这样的页为“overflow page”，类似ORACLE中的“行迁移”。频繁读写会降低数据库I/O性能，造成数据页空洞浪费硬盘空间
- 禁止创建没有主键的表
- 禁止DEFAULT NULL，建议NOT NULL 设置默认值
- 禁止使用外键
- 只使用InnoDB存储引擎（默认），禁止使用MyISAM引擎
- 禁止在数据库中存储图片、二进制文件等大数据，会影响数据库的性能和空间
- 禁止使用 MySQL已有的关键字，MySQL关键字参考：https://dev.mysql.com/doc/refman/8.0/en/keywords.html
- 禁止使用全文索引，全文索引不适用于OLTP场景
- 禁止使用join关联太多的表。对于TDSQL来说，是存在join buffer cache的，缓存的大小可以由join_buffer_size参数进行设置。对于同一个SQL多关联（join）一个表，就会多分配一个关联缓存，如果在一个SQL中关联的表越多，所占用的内存也就越大。如果程序中大量的使用了多表关联的操作，同时join_buffer_size设置的也不合理的情况下，就容易造成服务器内存溢出的情况，就会影响到服务器数据库性能的稳定性。同时对于关联操作来说，会产生临时表操作，影响查询效率。生产环境禁止超过5个表的join
- 禁止使用order by rand() 进行随机排序，会把表中所有符合条件的数据装载到内存中，然后在内存中对所有数据根据随机生成的值进行排序，并且可能会对每一行都生成一个随机值，如果满足条件的数据集非常大，就会消耗大量的CPU和IO及内存资源。推荐在程序中获取一个随机值，然后从数据库中获取数据的方式
- WHERE从句中禁止对列进行函数转换和计算，对列进行函数转换或计算时会导致无法使用索引。 如 where length(name)='ABC'或 where user_id+100=2000
- 禁止使用TIMESTAMP类型作为分区键，因为timestamp受到时区的影响，同时只能使用到2038年

性能建议
如何使用分表

最近更新时间：2021-10-18 17:12:08

业内常见分表规则

关系型数据库是一个二维模型，数据的切分通常就需要找到一个分表字段以确定拆分维度，再通过定义规则来实现数据库的拆分。业内的几种常见的分表规则如下：

- 基于某字段求模（Hash）
  将求模后字段的特定范围分散到不同库中。
- 基于日期或字段范围（Range）
  如按年拆分，2020 年一个分表，2021 年一个分表 。
  如按用户 ID 划分，0~~1000 一个分表，1001~~2000 一个分表。
- 基于枚举值列表（List）
- 按满足某些固定条件的数据分散到不同库中。

> ⚠ 注意：
> LIST和RANGE类似，区别在于LIST是枚举值列表的集合，RANGE是连续的区间值的集合。

各分表规则适用场景

RANGE规则适用场景

- 业务表数据量非常大，表存在日期或时间类型的字段
- 该字段通常不会被更新，很多查询语句都包含该字段
- 选该字段为分区键可以使得各个分区上的数据分配的比较均衡
- 可能需要定期按时间清理历史数据
  业务场景举例：用户登录日志表
- 用户每次登录都会记录登录日志
- 用户登录日志保存一年，1年后按照时间删除或者归档
- 以login_time（登录时间）为分表键

LIST规则适用场景

- 业务表数据量非常大，存在可以按分区键取值的列表进行分区
- 同范围分区一样，各分区的列表值不能重复
- 该字段通常不会被更新，很多查询语句都包含该字段

- 每一行数据必须能找到对应的分区列表，否则数据插入失败
  业务场景举例：城市常驻人口信息表
- 以city_district（城市行政区：0 东区 1西区 2南区 3北区）为分表键
- 将不同行政区的人口信息存入对应分区进行管理

HASH规则适用场景

无论是 Time、Range ，大部分业务场景都容易导致比较严重的数据倾斜，即分片之间负载和数据容量严重不均衡。例如，在大部分数据库系统中，数据有明显的冷热特征——例如当前的订单被访问的概率比半年前的订单要高的多。而采用 Time 分表或 range 分表，就意味着很容易出现大部分热数据将会被路由在少数几分片中，而剩下的分片设备性能却被白白浪费掉了。而采用某个字段求模（Hash）的方案进行分表就不会出现这种问题，因为 Hash 算法的原理能够基本保证数据相对均匀的分散在不同的物理设备中。
业务场景举例：商品订单表

- 以user_id（用户ID）进行HASH分区
- 用户ID是该表的主键

数据rebalance

概念介绍：

假设现在有一个表A，create table A(a int key, b int) local_table_options distributed by range(a) (s1 values less than(100)); 其数据分布在set1上: distributed by range(a) (s1 values less than(100)); 现在想在尽量不影响业务的情况下，把该表的数据进行重分布，使得重分布后的A表其数据分布为：distributed by range(a) (s1 values less than(50) , s2 values less than(100)); 即将原先分布在set1上的数据重分布到set1和set2上。rebalance工具要求尽量不影响业务，故讨论决定大表(记录大于等于30w)使用多源同步，小表根据是否有自增列决定是采用musqldump+insert into或者insert...select的方式导入。

工具配置文件介绍：

工具位于 proxy/bin/rebalance/ 目录下，共有四个文件，其中autoincre_table.sh和noautoincre_table.sh脚本为依赖脚本，被rebalance.sh脚本调用，rebalance.config为配置文件。

配置文件rebalance.config介绍：

database="rebalance" #数据库名
table="t1" #需要重分布的表名
rebalance="TDSQL_DISTRIBUTED BY RANGE(b) (s1 values less than ('200'));" #重分布语句，用于建新表
divide="300000" #暂时不用
ip="9.30.17.168" #数据库ip
use="test" #数据库用户

passwd="test123" #数据库用户密码
port="15064" #连接数据库的proxy端口

使用介绍：

首先将配置文件与3个脚本文件放在同一个目录下，按要求填写好rebalance.config并保存。然后运行rebalance.sh脚本即可。重分布完成后可以发现在相应数据库中指定的表A被新表A（按照指定重分布语句进行构造，数据与旧表一致）接替工作，旧表的表结构和数据会保存为A_bak表的形式存在。
重分布方案步骤如下：

1. 通过proxy端口进行mysqldump获得A表的建表语句，通过字符匹配修改，结合输入参数，形成分表B的建表语句，通过proxy新建该分表B表。
2. 通过proxy对A表的记录数进行判断：select count(*) from A;当记录数小于30w行时，进入步骤3；当记录数大于等于30w时，进入步骤4。
3. 小数据量的表处理方法：锁表，mysqldump导出数据，字符串替换将sql更换为导入B表。执行sql文件insert into到分表B（如果没有自增列，可以使用proxy提供的insert…select来实现），通过rename的方式将A更名为A_bak1,B更名为A，解锁。
4. 大数据量的表处理方法：调用多源同步的接口创建同步任务（srcid为相应setid，dstid为groupid，采用精确匹配找到相应的源表A和目标表B），全量同步A和分表B，监控延迟，当延迟小于等于4s时认为同步快要结束，此时锁表，再次监控延迟，当延迟小于等于4s时执行rename，将A更名为A_bak1,B更名为A，解锁，停止多源同步任务。

执行计划解读

最近更新时间：2021-10-18 17:12:13

如何查看执行计划

在企业的应用场景中，为了知道优化SQL语句的执行，需要查看SQL语句的具体执行过程，以加快SQL语句的执行效率。

- 可以使用explain+SQL语句来模拟优化器执行SQL查询语句，从而知道TDSQL是如何处理sql语句的：

```
MySQL [test]> explain select id,fname,lname from employees where id=20;
+----+-------------+-------+-----------+------+---------------+------+---------+------+------+----------+-------
-----------------------+-------------------------------------------------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
Extra | info |
+----+-------------+-------+-----------+------+---------------+------+---------+------+------+----------+-------
-----------------------+-------------------------------------------------------------------------------------------+
| 1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | no matching ro
w in const table | set_1619374020_1, explain select id,fname,lname from `test`.`employees` where (id = 20) |
+----+-------------+-------+-----------+------+---------------+------+---------+------+------+----------+-------
-----------------------+-------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

- 查看执行计划，SQL不会真正执行

```
MySQL [test]> select a,b from test1;
+---+------+
| a | b |
+---+------+
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |
+---+------+
3 rows in set (0.00 sec)

MySQL [test]> explain delete from test1 where a=3;
+----+-------------+-------+-----------+-------+---------------+---------+---------+-------+------+----------+--
```

```
----------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+--
----------+
| 1 | DELETE | test1 | NULL | range | PRIMARY | PRIMARY | 4 | const | 1 | 100.00 | Using where
|
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+--
----------+
1 row in set, 1 warning (0.00 sec)

MySQL [test]> select a,b from test1;
+---+------+
| a | b |
+---+------+
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |
+---+------+
3 rows in set (0.00 sec)
```

- **在只读的DB上，无法查看写SQL的执行计划**

```
MySQL [test]> explain delete from employees where id=20;
ERROR 1290 (HY000): The MySQL server is running with the --read-only option so it cannot e
xecute this statement
```

执行计划各个字段的含义

以下面的执行计划为例，说明各字段的含义：

```
MySQL [test]> explain select id,fname,lname from employees where id=20\G;
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: NULL
partitions: NULL
```

```
type: NULL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: NULL
filtered: NULL
Extra: no matching row in const table
info: set_1619374020_1, explain select id,fname,lname from `test`.`employees` where (id = 20)
```

- id：执行行顺序，按1,2,3,4…进行排序。在所有组中，id值越大，优先级越高，越先执行。id如果相同，可以认为是一组，从上往下顺序执行
- select_type：select的类型

select_type Value	Meaning
SIMPLE	Simple SELECT (not using UNION or subqueries)
PRIMARY	Outermost SELECT
UNION	Second or later SELECT statement in a UNION
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query
UNION RESULT	Result of a UNION.
SUBQUERY	First SELECT in subquery
DEPENDENT	First SELECT in subquery, dependent on outer query
SUBQUERY	
DERIVED	Derived table
UNCACHEABLE	A subquery for which the result cannot be cached and must
SUBQUERY	be re-evaluated for each row of the outer query
UNCACHEABLE	The second or later select in a UNION that belongs to an
UNION	uncacheable subquery (see UNCACHEABLE SUBQUERY)

关于select type，参考以下示例：

```
--sample:简单的查询，不包含子查询和union
explain select * from emp;

--primary:查询中若包含任何复杂的子查询，最外层查询则被标记为Primary
explain select staname,ename supname from (select ename staname,mgr from emp) t join em
p on t.mgr=emp.empno ;

--union:若第二个select出现在union之后，则被标记为union
explain select * from emp where deptno = 10 union select * from emp where sal >2000;

--dependent union:跟union类似，此处的depentent表示union或union all联合而成的结果会受外部表影
响
explain select * from emp e where e.empno in ( select empno from emp where deptno = 10 uni
on select empno from emp where sal >2000)

--union result:从union表获取结果的select
explain select * from emp where deptno = 10 union select * from emp where sal >2000;

--subquery:在select或者where列表中包含子查询
explain select * from emp where sal > (select avg(sal) from emp) ;

--dependent subquery:subquery的子查询要受到外部表查询的影响
explain select * from emp e where e.deptno in (select distinct deptno from dept);

--DERIVED: from子句中出现的子查询，也叫做派生类，
explain select staname,ename supname from (select ename staname,mgr from emp) t join em
p on t.mgr=emp.empno ;

--UNCACHEABLE SUBQUERY：表示使用子查询的结果不能被缓存
explain select * from emp where empno = (select empno from emp where deptno=@@sort_bu
ffer_size);

--uncacheable union:表示union的查询结果不能被缓存
```

- table：输出记录的表，对应行正在访问哪一个表，表名或者别名，可能是临时表或者union合并结果集
- partitions：符合的分区
- type：显示的是访问类型，访问类型表示以何种方式去访问数据，例如全表扫描

```
--all:全表扫描，一般情况下出现这样的sql语句而且数据量比较大的话那么就需要进行优化。
explain select * from emp;

--index：全索引扫描这个比all的效率要好，主要有两种情况，一种是当前的查询时覆盖索引，即我们需要的数据在索引中就可以索取，或者是使用了索引进行排序，这样就避免数据的重排序
explain select empno from emp;

--range：表示利用索引查询的时候限制了范围，在指定范围内进行查询，这样避免了index的全索引扫描，适用的操作符： =, <>, >, >=, <, <=, IS NULL, BETWEEN, LIKE, or IN()
explain select * from emp where empno between 7000 and 7500;

--index_subquery：利用索引来关联子查询，不再扫描全表
explain select * from emp where emp.job in (select job from t_job);

--unique_subquery:该连接类型类似与index_subquery,使用的是唯一索引
explain select * from emp e where e.deptno in (select distinct deptno from dept);

--index_merge：在查询过程中需要多个索引组合使用，没有模拟出来

--ref_or_null：对于某个字段即需要关联条件，也需要null值的情况下，查询优化器会选择这种访问方式
explain select * from emp e where e.mgr is null or e.mgr=7369;

--ref：使用了非唯一性索引进行数据的查找
create index idx_3 on emp(deptno);
explain select * from emp e,dept d where e.deptno =d.deptno;

--eq_ref ：使用唯一性索引进行数据查找
explain select * from emp,emp2 where emp.empno = emp2.empno;

--const：这个表至多有一个匹配行，
explain select * from emp where empno = 7369;
```

```
--system：表只有一行记录（等于系统表），这是const类型的特例，平时不会出现
```

- possible_keys：优化器可能使用到的索引
- key：优化器实际选择的索引
- key_len：表示索引中使用的字节数，可以通过key_len计算查询中使用的索引长度
- ref：显示索引的哪一列被使用了，如果可能的话，是一个常数
- rows：优化器预估的记录数量，根据表的统计信息及索引使用情况，大致估算出找出所需记录需要读取的行数
- filtered：该 filtered 列指示将按表条件过滤的表行的估计百分比。 最大值为100，这意味着不会对行进行过滤。 值从100开始减少表示过滤量增加
- Extra：额外的显示选项
- info：网关下推，记录了实际发往的set名称和sql信息，info这个一列信息是分布式实例执行计划特有的

DML语句的调整建议

最近更新时间：2021-10-18 17:12:21

DML执行时约束的开销

约束会对DML操作的性能产生影响：

- 完整性约束：时间会耗费在验证约束列的数据值是否合法上。
- 其他类型约束：数据也需要做相应的检查。

> ⚠ 注意：
> 在执行大容量数据的插入或变更前，备份原始数据，并且暂时禁用所有与所影响的数据表有关的约束，数据加载完成后重启约束。

DML执行时维护索引所需的开销

表中数据变更时，表上所有的参与索引都必须实时的进行更新，这会产生大量的系统开销，严重降低系统的执行性能。

> ⚠ 注意：
> 在大型DML批操作中，在更改数据表之前，删除全部索引。在操作完成后，重新建立索引。

如何删除大表

最近更新时间：2021-10-18 17:12:28

如何删除大表

在生产环境有可能有删除某个大表的需求，因为大表占用的大量磁盘空间，如果直接drop table可能会导致TDSQL实例挂起。建议采用以下方式进行：

- 如果业务是按照时间进行清理，比如月度、季度则建议使用二级分区，则在业务低峰期，利用删除分区的操作来完成数据清理
- 如果不能使用二级分区，则在业务低峰期，以小事务批量删除（一次delele操作的行数限制在5000行以内）

> **注意：**
> 数据完成删除后，各应用根据数据变化的频率，决定统计信息的收集频率。需要注意的是，统计信息的收集也需要在业务低峰期进行。另外，innodb引擎在表数据量变化超过10%后，也会在系统低负载期间自动进行统计信息更新。

truncate一次失败后如何处理

【建议】使用透传方式进行二次truncate

Proxy错误码及错误信息汇总

最近更新时间：2021-10-18 17:12:33

Proxy错误码及错误信息显示如下：

```
#define ER_PROXY_GRAM_ERROR_BEGIN 600

#define ER_PROXY_SANITY_ERROR 601 // "Sanity error: %s"
#define ER_PROXY_SQL_TYPE_NOT_SUPPORT 602 // sql 类型不支持
#define ER_PROXY_SQL_NOT_SUPPORT_ORDERBY_1 603 // order by index is negative
#define ER_PROXY_SQL_NOT_SUPPORT_ORDERBY_2 604 // order by index is too big
#define ER_PROXY_SQL_NOT_SUPPORT_ORDERBY_3 605 // 不支持到 order by 用法
#define ER_PROXY_SQL_NOT_SUPPORT_GROUPBY_1 606 // group by index is negative
#define ER_PROXY_SQL_NOT_SUPPORT_GROUPBY_2 607 // group by index is too big
#define ER_PROXY_SQL_NOT_SUPPORT_GROUPBY_3 608 // 不支持的 group by 用法
#define ER_PROXY_GET_AUTO_ID_FAILED 609 // get auto id back with error
#define ER_PROXY_TEANS_ROLLED_BACK 610 // 事务已经被回滚
#define ER_PROXY_ONE_SET 611 // 当前 sql 应该被发往一个后端，但是不是
#define ER_PROXY_CLIENT_HS_ERROR 612 // 解析客户端握手包出错
#define ER_PROXY_ACCESS_DENIED_ERROR 613 // the length of readu_auth_switch_result is no
t 20，不应该出现
#define ER_PROXY_TRANS_NOT_ALLOWED 614 // 事务中不允许执行的命令
#define ER_PROXY_TRANS_READ_ONLY 615 // 只读事务中不允许执行的命令
#define ER_PROXY_TRANS_ERROR_DIFFENT_SET 616 // 非 xa 事务中，只读 sql 使用了多个后端
#define ER_PROXY_STRICT_ERROR 617 // strict 模式下，一次仅允许修改一个 set
#define ER_PROXY_SC_TOO_LONG 618 // 后端断开时间过长，断开链接
#define ER_PROXY_START_TRANS_FAILED 619 // 开启新的 xa 事务失败
#define ER_PROXY_SC_RETRY 620 // server 已经 close，请重试上一条 sql
#define ER_PROXY_SC_TRANS_IN_ROLLBACK_ONLY 621 // server 已经 close，当前事务处于 rollba
ck
#define ER_PROXY_SC_COMMIT_LATER 622 // server 已经 close，事务会在稍后提交
#define ER_PROXY_SC_ROLLBACL_LATER 623 // server 已经 close，事务会在稍后回滚
#define ER_PROXY_SC_IN_COMMIT_OR_ROLLBACK 624 // server 在事务提交/回滚阶段 close
#define ER_PROXY_SC_NEED_ROLLBACK 625 // server 已经 close，需要首先会滚当前事务
#define ER_PROXY_SC_STATE_WILL_ROLLBACK 626 // server 已经 close，将会会滚
#define ER_PROXY_XA_UNSUPPORT 627 // xa 目前不支持的命令
#define ER_PROXY_XA_INVALID_COMMAND 628 // xa 命令不合法
```

```
#defien ER_PROXY_XA_GTID_INIT_ERROR 629 // gtid log 初始化失败
#define ER_PROXY_XA_GET_SET_IP_PORT_FAILED 630 // 获取 set 地址失败
#define ER_PROXY_XA_UPDATE_GTID_LOG_FAILED 631 // 更新 gtid log 失败
#define ER_PROXY_MYSQL_PARSER_ERROR 632 // mysql 解析失败，返回详细错误信息
#define ER_PROXY_MYSQL_PARSER_UNEXPECTED_ERROR 633 // mysql 解析失败，unexpected e
rror
#define ER_PROXY_MYSQL_PARSER_NOT_SUPPORTED 634 // mysql 不支持的命令
#define ER_PROXY_ILLEGAL_ID 635 // kill id 不合法
#define ER_PROXY_NOT_SUPPORT_CURSOR 636 // CURSOR_TYPE_READ_ONLY 不支持
#define ER_PROXY_UNKNOWN_PREPARE_HANDLER 637 // 执行的 prepare 不明确
#define ER_PROXY_SET_PARA_FAIL 638 // Set parameters failed
#define ER_PROXY_SUBPARTITION_TABLE_TOO_MANY_DEAL 639 // 只能够处理一个二级分区表
#define ER_PROXY_NS_AND_SHARD_TABLE_DENY 640 // can not deal with noshard and shard t
able
#define ER_PROXY_NS_AND_GLOBAL_TABLE_DENY 641 // Can not deal with noshard and global
table
#define ER_PROXY_NO_SUBPARTITION_ROUTE 642 // 没有获取到二级分区表的路由信息
#define ER_PROXY_LOCK_MORE_TABLE 643 // 一次只可以锁定一张二级分区表
#define ER_PROXY_GET_ROUTER_LOCK_FAIL 644 // 获取路由锁失败
#define ER_PROXY_PART_NAME_EMPTY 645 // part name 为空
#define ER_PROXY_SUB_PART_TABLE_IS_NONE 646 // 没有耳机分区表
#define ER_PROXY_ALTER_PART_TYPE_TO_RANGE 647 // "Table has list type, alter use range typ
e"
#define ER_PROXY_ALTER_PART_TYPE_TO_LIST 648 // "Table has range type, alter use list type"
#define ER_PROXY_PART_NAME_ILLEGAL 649 // 分区名不合法
#define ER_PROXY_DROP_ALL_PARTITION_FAIL 650 // 删除所有分区失败，尝试直接删除表
#define ER_PROXY_GET_OLD_PART_NUM_FAIL 651 // 获取表的分片数失败
#define ER_PROXY_EMPTY_SQL 652 // empty sql，不会返回给客户端
#define ER_PROXY_ERROR_SHARDKEY 653 // sk 必须为某一列
#define ER_PROXY_ERROR_SUB_SHARDKEY 654 // 二级分表键失败
#define ER_PROXY_SQLUSE_NOT_SUPPORT 655 // proxy 不支持这种用法
#define ER_PROXY_DBFW_WHITE_LIST_DENY 656 // 不在白名单，被防火墙拒绝
#define ER_PROXY_DBFW_DENY 657 // 防火墙拒绝
#define ER_PROXY_INCORRECT_ARGS 658 // stmt 参数不正确
#define ER_PROXY_SYSTABLE_UNSUPPORT_NON_READ_SQL 659 // 不支持非只读sql访问系统表
#define ER_PROXY_TABLE_NOT_EXIST 660 // 表不存在
#define ER_PROXY_SHARD_JOIN_UNSUPPORT_TYPE 661 // shard join 不支持的用法
#define ER_PROXY_RECURSIVE_JOIN_DENY 662 // 递归 join 不支持
```

```
#define ER_PROXY_JOIN_INTERNAL_ERROR 663 // join 异常
#define ER_PROXY_SQL_TOO_COMPLEX 664 // sql 太复杂，groupshard 不支持
#define ER_PROXY_INVALID_ARG_FOR_GTID_STATE 665 // gtid_state() 参数不合法
#define ER_PROXY_CANT_SET_GLOBAL_AUTOCOMMIT_GS 666 // Global autocommit cannot be set in groupshard
#define ER_PROXY_INVALID_VALUE_FOR_AUTOCOMMIT 667 // autocommit 值设置不合法
#define ER_PROXY_XID_ERROR 668 // xid 不合法
#define ER_PROXY_XID_GENERAT_FAILED 669 // xid 不能由用户指定
#define ER_PROXY_CANT_EXEC_IN_INTER_TRANS 670 // "The command cannot be executed in internal transction"
#define ER_PROXY_XID_TIME_ERROR 671 // "Unexpected time part of xid"
#define ER_PROXY_XID_TIMEDIFF_TOO_LONG 672 // "timediff > 1800s, it's not safe to execute boost"
#define ER_PROXY_SAVEPOINT_NOT_EXIST 673 // SAVEPOINT 不存在
#define ER_PROXY_SC_TRANS_IN_ROLLED 674 // 事务已经会滚，由于 serevr 已经 close
#define ER_PROXY_CANT_BOOST_IN_TRANS 675 // 事务中不允许执行 SQLCOM_BOOST
#define ER_PROXY_TRANS_EXPECTED 676 // "A transaction is expected, this maybe a bug"
#define ER_PROXY_EXTERNAL_TRANS 677 // 外部 xa 中不允许执行
#define ER_PROXY_AUTO_INC_FAIL 678 // "Deal auto inc failed"
#define ER_PROXY_CHECK_JOIN_FAIL 679 // "Check join failed"
#define ER_PROXY_TABLE_TYPE_NOT_MATCH 680 // "Do not support shard-table operations in noshard instance"
#define ER_PROXY_UNSUPPORT_NS_IN_INSERT 681 // "Do not support noshard and noshard_allset in insert sql"
#define ER_PROXY_ALTER_SEQ_ID_FAIL 682 // Alter seq id failed
#define ER_PROXY_ALTER_ID_ILLEGAL 683 // Alter seq id is illegal
#define ER_PROXY_CANT_CHANGE_STEP 684 // "Current table use zk to get auto inc, do not support to change step: \'%s\'"
#define ER_PROXY_ALTER_STEP_FAIL 685 // Alter step failed
#define ER_PROXY_TOO_MUCH_TABLES 686 // "Too much tables, exceed the maximum value"
#define ER_PROXY_TABLE_EXISTED 687 // 表已经存在
#define ER_PROXY_CREATE_STABLE_FAILED 688 // "Complex sql can not used to create shard tables"
#define ER_PROXY_DDL_DENY 689 // "DDL can not handle noshard and global table"
#define ER_PROXY_SHADKEY_ERROR 690 // "SQL should not relate to subpartition tables"
#define ER_PROXY_NO_SK 691 // reject nosk
#define ER_PROXY_COMBINE_SQL_KEY 692 // "Something went wrong:%s"
#define ER_PROXY_GET_SK_ERROR 693 // sk 获取失败
```

```
#define ER_PROXY_SHOW_FAILED 684 // "%s, see /*proxy*/ help"
#define ER_PROXY_SET_FAILED 695 // "%s, see /*proxy*/ help"
#define ER_PROXY_UNLOCK_FORMAT_ERROR 696 // sql 格式不正确
#define ER_PROXY_UNLOCK_ROUTER_FAIL 697 // "Unlock failed"
#define ER_PROXY_LOCK_ROUTER_FAIL 698 // "Lock failed"
#define ER_PROXY_PROXY_CMD_FAIL 699 // 不支持的/*proxy*/ 命令
#define ER_PROXY_PROCESS_RULE_FILE_FAILED 700 // dump_error
#define ER_PROXY_GET_AUTO_NUM_ERROR 701 // "Get auto num failed"
#define ER_PROXY_SEQUENCE_NOT_EXIST 702 // sequence 不存在
#define ER_PROXY_SEQUENCE_ERROR 703 // sequence 不合法
#define ER_PROXY_SEQUENCE_ALREADY_EXIST 704 // Sequence 已经存在

#define ER_PROXY_GRAM_ERROR_END 705
#define ER_PROXY_SYSTEM_ERROR_BEGIN 900

#define ER_PROXY_SLICING 901 // slice 被修改，可能在扩容阶段，拒掉当前 sql
#define ER_PROXY_NO_DEFAULT_SET 902 // set 为空
#define ER_PROXY_GET_ADDRESS_FAILED 903 // 还未初始化完成，获取后端地址失败，稍后重试
#define ER_PROXY_SQL_SIZE_ERROR_IN_GET_CANDIDATE_ADDRESS 904 // 获取后端地址出错（发往后端个数不正确）
#define ER_PROXY_GET_ADDRESS_ERROR 905 // 获取后端地址出错
#define ER_PROXY_CANDIDATE_ADDRESS_EMPTY 906 // 未获取到后端地址
#define ER_PROXY_SOCK_ERROR 907 // 当前 sql 不允许发送到后端
#define ER_PROXY_CANT_GET_SOCK 908 // socket 获取失败
#define ER_PROXY_GET_SET_SOCK_FAIL 909 // socket 获取失败
#define ER_PROXY_CONNECT_ERROR 910 // 后端连接失败
#define ER_PROXY_NO_SQL_ASSIGN_TO_SET 911 // an unbelievable error
#define ER_PROXY_STATUS_ERROR 912 // group 状态异常，断开链接
#define ER_PROXY_CONN_BROKEN_ERROR 913 // server close，sql 状态不正常
#define ER_PROXY_UNKNOWN_ERROR 914 // proxy 未知错误（可能为异常引起）
#define ER_PROXY_SQL_RETRY 915 // sql 还未提交或回顾
#define ER_PROXY_XA2PC_ABORT 916 // 2pc 失败，事务将会会滚:
#define ER_PROXY_XA2PC_COMMIT 917 // 2pc 失败，后续提交
#define ER_PROXY_XA2PC_UNCERTAIN 918 // 2pc 失败，结果未知
#define ER_PROXY_ERROR_END 919
```

⚠ 注意：

其中错误码为900以上的为系统错误，将会通过监控平台进行告警。