分布式关系型数据库 DRDS

最佳实践

最佳实践

DRDS 与 RDS 按照 CPU 的处理能力、内存容量和磁盘空间等来划分实例的规格，并提供多种不同规格的实例供选择，规格越高代表实例的处理能力越强。

DRDS 实例类型与规格

- **实例类型**：DRDS 为您提供专享实例，支持动态变更配置和其它服务；
- **实例规格**：包含多种规格来提供不同的处理能力。最小规格的配置为 8Core16G（两个 DRDS 节点），参考的最大 QPS 约为 10000。其它规格专享实例的参考 QPS 均为 10000 的整倍数。

DRDS 实例类型与规格的选择

DRDS 属于计算密集型的服务，处理能力主要与 CPU 相关，并以 QPS 为衡量指标。选择实例时，应参照不同规格支持的最大 QPS，并结合估算的业务最大 QPS 来进行选择。例如，某用户估算其业务应用对 DRDS 访问的最大 QPS 为40000，那么应选择规格为 16Core16G 的专享实例。

DRDS 实例除了有丰富的规格供用户选择，还可选择包年包月或按量付费的灵活计费方式，更多信息请参考 DRDS 产品规格和价格说明。

注意：QPS 估算与实际执行的 SQL、数据量、表结构和数据库存储规格相关。本文档的选择策略建立在简单的单条数据查询的基础上，且查询条件中带有拆分键路由到单个分片数据，单条数据容量不超过 1KB。

RDS 实例规格的选择

- 预估 1 到 2 年的数据增长量，判断需要的最大磁盘空间；
- 估算一个实例需要的最大 IOPS。

应根据当前以及未来几年的业务情况对上述指标进行估算，选择需要购买的 RDS 实例数量和单个 RDS 实例的规格。更多关于 RDS 实例规格的选择，请参考 RDS 购买指南。

拆分键即分库/分表字段，是在水平拆分过程中用于生成拆分规则的数据表字段。DRDS 根据拆分键的值将数据表水平拆分到每个 RDS 实例上的物理分库中。

数据表拆分的首要原则，就是要尽可能找到数据表中的数据在业务逻辑上的主体，并确定大部分（或核心的）数据库操作都是围绕这个主体的数据进行，然后可使用该主体对应的字段作为拆分键，进行分库分表。

业务逻辑上的主体，通常与业务的应用场景相关，下面的一些典型应用场景都有明确的业务逻辑主体，可用于

拆分键：

- 面向用户的互联网应用，都是围绕用户维度来做各种操作，那么业务逻辑主体就是用户，可使用用户对应的字段作为拆分键；
- 侧重于卖家的电商应用，都是围绕卖家维度来进行各种操作，那么业务逻辑主体就是卖家，可使用卖家对应的字段作为拆分键；
- 游戏类的应用，是围绕玩家维度来做各种操作，那么业务逻辑主体就是玩家，可使用玩家对应的字段作为拆分键；
- 车联网方面的应用，则是基于车辆信息进行操作，那么业务逻辑主体就是车辆，可使用车辆对应的字段作为拆分键；
- 税务类的应用，主要是基于纳税人的信息来开展前台业务，那么业务逻辑主体就是纳税人，可使用纳税人对应的字段作为拆分键。

以此类推，其它类型的应用场景，大多也能找到合适的业务逻辑主体作为拆分键的选择。

例如，某面向卖家的电商应用，需要对下面的一张单表进行水平拆分：

```
CREATE TABLE sample_order (
  id INT(11) NOT NULL,
  sellerId INT(11) NOT NULL,
  trade_id INT(11) NOT NULL,
  buyer_id INT(11) NOT NULL,
  buyer_nick VARCHAR(64) DEFAULT NULL,
  PRIMARY KEY (id)
)
```

由于确定了业务逻辑主体为卖家，那么选择对应的字段 sellerId 作为拆分键，且只分库不分表，则分布式 DDL 建表语句为：

```
CREATE TABLE sample_order (
  id INT(11) NOT NULL,
  sellerId INT(11) NOT NULL,
  trade_id INT(11) NOT NULL,
  buyer_id INT(11) NOT NULL,
  buyer_nick VARCHAR(64) DEFAULT NULL,
  PRIMARY KEY (id)
) DBPARTITION BY HASH(sellerId)
```

如果确实找不到合适的业务逻辑主体作为拆分键，那么可以考虑下面的方法来选择拆分键：

根据数据分布和访问的均衡度来考虑拆分键，尽量将数据表中的数据相对均匀地分布在不同的物理分库/分表中，适用于大量分析型查询的应用场景（查询并发度大部分能维持为1）；

按照数字（字符串）类型与时间类型字段相结合作为拆分键，进行分库和分表，适用于日志检索类的应用场景。

例如，某日志系统记录用户的所有操作，需要对下面的日志单表进行水平拆分：

```
CREATE TABLE user_log (
  userId INT(11) NOT NULL,
  name VARCHAR(64) NOT NULL,
  operation VARCHAR(128) DEFAULT NULL,
  actionDate DATE DEFAULT NULL
)
```

可以选择用户标识与时间字段相结合作为拆分键，并按照一周七天进行分表，则分布式 DDL 建表语句为：

```
CREATE TABLE user_log (
  userId INT(11) NOT NULL,
  name VARCHAR(64) NOT NULL,
  operation VARCHAR(128) DEFAULT NULL,
  actionDate DATE DEFAULT NULL
) DBPARTITION BY HASH(userId) TBPARTITION BY WEEK(actionDate) TBPARTITIONS 7
```

更多拆分键的选择和分表形式，请参考 DRDS DDL 语句。

注意：无论选择什么拆分键，采用何种拆分策略，都要注意拆分值是否存在热点的问题，尽量规避热点数据来选择拆分键。

DRDS 中的水平拆分有两个层次：分库和分表。每个 RDS 实例上默认会创建8个物理分库，每个物理分库上可以创建一个或多个物理分表。分表数通常也被称为分片数。

一般情况下，建议单个物理分表的容量不超过500万行数据。通常可以预估1到2年的数据增长量，用估算出的总数据量除以总的物理分库数，再除以建议的最大数据量500万，即可得出每个物理分库上需要创建的物理分表数：

```
物理分库上的物理分表数 = 向上取整(估算的总数据量 / (RDS 实例数 * 8) / 5,000,000)
```

因此，当计算出的物理分表数等于1时，分库即可，无需再进一步分表，即每个物理分库上一个物理分表；若计算结果大于1，则建议既分库又分表，即每个物理分库上多个物理分表。

例如，某用户预估一张表在2年后的总数据量大概是1亿行，购买了4个 RDS 实例，那么按照上述公式计算：

```
物理分库上的物理分表数 = CEILING(100,000,000 / ( 4 * 8 ) / 5,000,000) = CEILING(0.625) = 1
```

结果为1，那么只分库即可，即每个物理分库上1个物理分表。

若上述例子中仅购买了1个 RDS 实例，那么按照上述公式计算：

```
物理分库上的物理分表数 = CEILING(100,000,000 / ( 1 * 8 ) / 5,000,000) = CEILING(2.5) = 3
```

结果为3，那么建议既分库又分表，即每个物理分库上3个物理分表。

SQL 优化

DRDS 是一个高效、稳定的分布式关系数据库服务，处理的是分布式关系运算。DRDS 对 SQL 的优化方法与单机关系数据库（例如 MySQL）有所不同，侧重考虑分布式环境中的网络 IO 开销，会尽量将 SQL 中的运算下推到底层各个分库（例如 RDS/MySQL）执行，从而减少网络 IO 开销、提升 SQL 执行效率。

DRDS 提供了一些指令来获取 SQL 的执行信息、辅助 SQL 的优化，例如获取 SQL 执行计划的 EXPLAIN 系列指令、获取 SQL 执行过程和开销的 TRACE 指令等。本文档介绍 DRDS 中 SQL 优化相关的基本概念和常用指令。

执行计划

为了访问数据而产生的一组有序的操作步骤集合，被称为 SQL 执行计划（简称执行计划）。在 DRDS 中，执行计划分为两个层次：DRDS 层的执行计划与 RDS/MySQL 层的执行计划。对执行计划的分析是进行 SQL 优化的有效方法，可以了解 DRDS 或 RDS/MySQL 是否对 SQL 语句生成了最优化的执行计划，是否有优化的空间等，从而为 SQL 优化提供重要的参考信息。

在 SQL 语句执行期间，DRDS 优化器会根据 SQL 语句和相关表的基本信息，判断该 SQL 语句应该在哪些分库上执行，决定在分库上执行的具体 SQL 语句形式，采用何种执行策略、数据合并与计算策略等。这个过程会尽可能达到优化 SQL 语句执行的目的，并产生 DRDS 层的执行计划。而 RDS/MySQL 层的执行计划就是原生的 MySQL 执行计划。

DRDS 提供了一组 EXPLAIN 指令来查看不同层面或不同详尽程度的执行计划。

表 1 是 DRDS 中 EXPLAIN 指令的简要说明，详细信息请参考 DRDS 控制指令。

控制指令	说明	示例
EXPLAIN { SQL }	查看 DRDS 层 SQL 语句的概要执行计划，包括执行的分库、物理语句和整体参数。	EXPLAIN SELECT * FROM test
EXPLAIN DETAIL { SQL }	查看 DRDS 层 SQL 语句的详细执行计划，包括执行语句类型、并发度、返回字段信息、物理表和库分组等。	EXPLAIN DETAIL SELECT * FROM test
EXPLAIN EXECUTE { SQL }	查看底层 RDS/MySQL 的执行计划，等同于 MySQL 的 EXPLAIN 语句。	EXPLAIN EXECUTE SELECT * FROM test

表1 EXPLAIN 指令

DRDS 层执行计划

DRDS 层执行计划的返回结果中，字段含义如表2所示：

字段	说明
GROUP_NAME	DRDS 分库的名字，可以根据后缀识别出是哪个分

	库，其值与 SHOW NODE 指令的结果一致。
SQL	在该分库上执行的 SQL 语句。
PARAMS	当 DRDS 使用 Prepare 协议与 MySQL 通信时，SQL 语句的参数列表。

表2 执行计划的字段含义

其中 SQL 字段的内容有两种形式：

1、 如果 SQL 语句不包含以下部分，则以 SQL 语句的形式显示执行计划：

- 涉及多个分库的聚合函数；
- 涉及多个分片的分布式 Join；
- 复杂子查询。

例如：

```
mysql> EXPLAIN SELECT * FROM test;
+-----------------------------------------------+-----------------------------------------+--------+
| GROUP_NAME                                    | SQL                                     | PARAMS |
+-----------------------------------------------+-----------------------------------------+--------+
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0000_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0001_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0002_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0003_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0004_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0005_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0006_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
| TESTDB_1478746391548CDTCTESTDB_OXGJ_0007_RDS | select `test`.`c1`,`test`.`c2` from `test` | {}     |
+-----------------------------------------------+-----------------------------------------+--------+
8 rows in set (0.04 sec)
```

GROUP_NAME 字段中显示的 GROUP 名称可以在 SHOW NODE 的结果中找到：

```
mysql> SHOW NODE;
+----+-----------------------------------------------+------------------+-----------------+--------------------+--------------------+
| ID | NAME                                          | MASTER_READ_COUNT | SLAVE_READ_COUNT | MASTER_READ_PERCENT | SLAVE_READ_PERCENT |
+----+-----------------------------------------------+------------------+-----------------+--------------------+--------------------+
| 0  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0000_RDS |               69 |               0 | 100%               | 0%                 |
| 1  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0001_RDS |               45 |               0 | 100%               | 0%                 |
| 2  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0002_RDS |               30 |               0 | 100%               | 0%                 |
| 3  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0003_RDS |               29 |               0 | 100%               | 0%                 |
| 4  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0004_RDS |               11 |               0 | 100%               | 0%                 |
```

```
| 5  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0005_RDS |              1 |              0 | 100%              | 0%
|
| 6  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0006_RDS |              8 |              0 | 100%              | 0%
|
| 7  | TESTDB_1478746391548CDTCTESTDB_OXGJ_0007_RDS |             50 |              0 | 100%              | 0%
|
+----+----------------------------------------------+----------------+----------------+-------------------+----------
----------+
8 rows in set (0.10 sec)
```

2、无法使用 SQL 语句表示的执行计划，DRDS 使用自定义格式的执行计划来表示。

例如：

```
mysql> EXPLAIN DETAIL SELECT COUNT(*) FROM test;
+-----------------------------------------------+---------------------------------+--------+
| GROUP_NAME                                    | SQL                             | PARAMS |
+-----------------------------------------------+---------------------------------+--------+
| TEST_DB_1478746391548CDTCTEST_DB_OXGJ_0000_RDS | Merge as test
    queryConcurrency:GROUP_CONCURRENT
    columns:[count(*)]
    executeOn: TEST_DB_1478746391548CDTCTEST_DB_OXGJ_0000_RDS
        Query from test as test
            queryConcurrency:SEQUENTIAL
            columns:[count(*)]
            tableName:test
            executeOn: TEST_DB_1478746391548CDTCTEST_DB_OXGJ_0000_RDS
        Query from test as test
            queryConcurrency:SEQUENTIAL
            columns:[count(*)]
            tableName:test
            executeOn: TEST_DB_1478746391548CDTCTEST_DB_OXGJ_0001_RDS
        ... ...
        Query from test as test
            queryConcurrency:SEQUENTIAL
            columns:[count(*)]
            tableName:test
            executeOn: TEST_DB_1478746391548CDTCTEST_DB_OXGJ_0007_RDS
| NULL   |
+-----------------------------------------------+---------------------------------+--------+
1 row in set (0.00 sec)
```

其中，SQL 字段内容中的 executeOn 表示下推的 SQL 语句在哪个分库上执行，分库执行后返回的结果最终由 DRDS 进行合并。

RDS/MySQL 层执行计划

RDS/MySQL 层执行计划的结果与原生 MySQL 执行计划一致，请参考 MySQL 官方文档。

一个 DRDS 逻辑表可能由多个分布在不同分库上的分片所组成，所以查看 RDS/MySQL 层执行计划也有多种方法。

1、查看一个 RDS/MySQL 分片的执行计划。

如果查询条件中带有拆分键，则直接使用 EXPLAIN EXECUTE 指令来查看对应分片上的执行计划。例如：

```
mysql> EXPLAIN EXECUTE SELECT * FROM test WHERE c1 = 1;
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | test  | const | PRIMARY       | PRIMARY | 4       | const |    1 | NULL  |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
1 row in set (0.04 sec)
```

注意：如果 SQL 语句出现了跨分片的情况（例如 SQL 语句的条件中没有带拆分键），则 EXPLAIN EXECUTE 会随机返回一个 RDS/MySQL 分片上的执行计划。

如果要查看一条 SQL 语句在指定分片上的执行计划，可以使用 Hint 的方式来实现。例如：

```
mysql> /!TDDL:node='TESTDB_1478746391548CDTCTESTDB_OXGJ_0000_RDS'*/EXPLAIN SELECT * FROM test;
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
| id | select_type | table | type | possible_keys |  key  | key_len |  ref  | rows | Extra |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
|  1 | SIMPLE      | test  | ALL  | NULL          | NULL  | NULL    | NULL  |    2 | NULL  |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
1 row in set (0.04 sec)
```

2、查看所有 RDS/MySQL 分片的执行计划：

如果确实需要查看 SQL 语句在所有分片上的执行计划，可以利用 SCAN Hint 来实现：

```
mysql> /!TDDL:scan='test'*/EXPLAIN SELECT * FROM test;
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
| id | select_type | table | type | possible_keys |  key  | key_len |  ref  | rows | Extra |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
|  1 | SIMPLE      | test  | ALL  | NULL          | NULL  | NULL    | NULL  |    2 | NULL  |
|  1 | SIMPLE      | test  | ALL  | NULL          | NULL  | NULL    | NULL  |    3 | NULL  |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------+
2 rows in set (0.08 sec)
```

注意：

使用 Hint 方式时，除了分库分表情况下的表名替换，DRDS 不会对 SQL 语句做其它处理，会直接将逻辑 SQL 语句发送到 RDS/MySQL 上执行，结果也不会做任何处理。

通过 EXPLAIN 获取的执行计划是静态分析产生的，并没有真正在数据库中执行。

TRACE 指令

DRDS 中的 TRACE 指令可以跟踪 SQL 的执行过程和各个阶段的执行开销，与执行计划相结合，更有助于对 SQL 进行优化。

TRACE 指令包含两条相关的指令：TRACE 和 SHOW TRACE，需要在一起配合使用。详细的用法信息请参考 DRDS 控制指令。

本文档介绍 DRDS 中 SQL 优化的原则和不同类型 SQL 的优化方法。文档中涉及到的基本概念和指令，请参考文档 SQL 优化基本概念。

SQL 优化的基本原则

在 DRDS 中，可由 RDS/MySQL 执行的 SQL 计算称为可下推计算。可下推计算能够减少数据传输，减少网络层和 DRDS 层的开销，提升 SQL 语句的执行效率。

因此，DRDS SQL 语句优化的基本原则为：**尽量让更多的计算可下推到 RDS/MySQL 上执行。**

可下推计算主要包括：

- JOIN 连接；
- 过滤条件，如 WHERE 或 HAVING 中的条件；
- 聚合计算，如 COUNT，GROUP BY 等；
- 排序，如 ORDER BY；
- 去重，如 DISTINCT；
- 函数计算，如 NOW() 函数等；
- 子查询。

注意：上述列表只是列出可下推计算的各种可能形式，并不代表所有的子句/条件或者子句/条件的组合一定是可下推计算。

不同类型和条件的 SQL 优化有不同的侧重点和方法，下面将针对以下几种情况介绍 SQL 优化的具体方法：

- 单表 SQL 优化
    - 过滤条件优化
    - 查询返回行数优化
    - 分组及排序优化
- JOIN 优化
    - 可下推的 JOIN 优化
    - 分布式 JOIN 优化
- 子查询优化

单表 SQL 优化

单表 SQL 优化有以下几个原则：

- SQL 语句尽可能带有拆分键；
- 拆分键的条件尽可能是等值条件；
- 如果拆分键的条件是 IN 条件，则 IN 后面的值的数目应尽可能少（需要远少于分片数，并且数目不会随业务的增长而增多）；
- 如果 SQL 语句不带有拆分键，那么 DISTINCT、GROUP BY 和 ORDER BY 在同一个 SQL 语句中尽

量只出现一种。

过滤条件优化

DRDS 的数据是按拆分键水平切分的，过滤条件中应尽量包含带有拆分键的条件，可以让 DRDS 根据拆分键对应的值将查询直接下推到特定的分库，避免 DRDS 做全表扫描。

例如，表 test 的拆分键是 c1，过滤条件中若不带有拆分键，会做全表扫描：

```
mysql> SELECT * FROM test WHERE c2 = 2;
+----+----+
| c1 | c2 |
+----+----+
|  2 |  2 |
+----+----+
1 row in set (0.05 sec)
```

对应的执行计划为：

```
mysql> EXPLAIN SELECT * FROM test WHERE c2 = 2;
+-----------------------------------------------+-------------------------------------------------------------------------------+-------
-+
| GROUP_NAME                                    | SQL                                                                           | PARAMS |
+-----------------------------------------------+-------------------------------------------------------------------------------+-------
-+
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0004_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0007_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0005_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0002_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0003_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0006_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0000_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0001_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c2` =
2) | {}     |
+-----------------------------------------------+-------------------------------------------------------------------------------+-------
-+
8 rows in set (0.00 sec)
```

含拆分键的过滤条件的取值范围越小，越有助于提高 DRDS 的查询速度。

例如，对表 test 查询时包含带有拆分键 c1 的范围过滤条件：

```
mysql> SELECT * FROM test WHERE c1 > 1 AND c1 < 4;
+----+----+
| c1 | c2 |
+----+----+
```

```
|  2 |  2 |
|  3 |  3 |
+----+----+
2 rows in set (0.04 sec)
```

对应的执行计划为：

```
mysql> EXPLAIN SELECT * FROM test WHERE c1 > 1 AND c1 < 4;
+-----------------------------------------------+-----------------------------------------------------------------------------------------------------------------+--------+
| GROUP_NAME                                    | SQL                                                                                                             | PARAMS |
+-----------------------------------------------+-----------------------------------------------------------------------------------------------------------------+--------+
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0002_RDS | select `test`.`c1`,`test`.`c2` from `test` where ((`test`.`c1` > 1) AND (`test`.`c1` < 4)) | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0003_RDS | select `test`.`c1`,`test`.`c2` from `test` where ((`test`.`c1` > 1) AND (`test`.`c1` < 4)) | {}     |
+-----------------------------------------------+-----------------------------------------------------------------------------------------------------------------+--------+
2 rows in set (0.00 sec)
```

等值条件会比范围条件执行得更快。例如：

```
mysql> SELECT * FROM test WHERE c1 = 2;
+----+----+
| c1 | c2 |
+----+----+
|  2 |  2 |
+----+----+
1 row in set (0.03 sec)
```

对应的执行计划为：

```
mysql> EXPLAIN SELECT * FROM test WHERE c1 = 2;
+-----------------------------------------------+-----------------------------------------------------------------------------+--------+
| GROUP_NAME                                    | SQL                                                                         | PARAMS |
+-----------------------------------------------+-----------------------------------------------------------------------------+--------+
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0002_RDS | select `test`.`c1`,`test`.`c2` from `test` where (`test`.`c1` = 2) | {}     |
+-----------------------------------------------+-----------------------------------------------------------------------------+--------+
1 row in set (0.00 sec)
```

此外，在向拆分表中插入数据时，插入字段中必须带有拆分键。

例如，向表 test 中插入数据时带有拆分键 c1：

```
mysql> INSERT INTO test(c1,c2) VALUES(8,8);
Query OK, 1 row affected (0.07 sec)
```

查询返回行数优化

DRDS 在执行带有 **LIMIT** [*offset*,] *row_count* 的查询时，实际上是依次将 *offset* 之前的记录读取出来并直接丢弃，这样当 *offset* 非常大的时候，即使 *row_count* 很小，也会导致查询非常缓慢。例如以下的 SQL：

```
SELECT *
FROM sample_order
ORDER BY sample_order.id
LIMIT 10000, 2
```

它虽然只返回第10000与10001两条记录，可它的执行时间为12秒左右，这是因为 DRDS 实际读取的记录数为10002条：

```
mysql> SELECT * FROM sample_order ORDER BY sample_order.id LIMIT 10000,2;
+--------------+------------+--------------+--------------+------------+
| id           | sellerId   | trade_id     | buyer_id     | buyer_nick |
+--------------+------------+--------------+--------------+------------+
| 242012755468 | 1711939506 | 242012755467 | 244148116334 | zhangsan   |
| 242012759093 | 1711939506 | 242012759092 | 244148138304 | wangwu     |
+--------------+------------+--------------+--------------+------------+
2 rows in set (11.93 sec)
```

对应的执行计划为：

```
mysql> EXPLAIN SELECT * FROM sample_order ORDER BY sample_order.id LIMIT 10000,2;
+----------------------------------------------+------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------+-
-------+
| GROUP_NAME                                   | SQL
| PARAMS |
+----------------------------------------------+------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------+-
-------+
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0004_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0007_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0005_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0002_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0003_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0006_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0000_RDS | select
```

```
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0001_RDS | select
`sample_order`.`id`,`sample_order`.`sellerId`,`sample_order`.`trade_id`,`sample_order`.`buyer_id`,`sample_order`.`buyer
_nick` from `sample_order` order by `sample_order`.`id` asc  limit 0,10002 | {}     |
+-----------------------------------------------+-----------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+-
-------+
8 rows in set (0.01 sec)
```

针对上述情况，SQL 优化方向是先查出 id 集合，再通过 IN 匹配真正的记录内容，改写后的 SQL 查询如下：

```
SELECT *
FROM sample_order o
WHERE o.id IN (
    SELECT id
    FROM sample_order
    ORDER BY id
    LIMIT 10000, 2 )
```

这样改写的目的是先用内存缓存 id（前提是 id 数目不多），如果 sample_order 表的拆分键是 id，那么 DRDS 还可以将这样的 IN 查询通过规则计算下推到不同的分库来查询，避免全表扫描和不必要的网络 IO。观察改写后的 SQL 查询效果：

```
mysql> SELECT *
    -> FROM sample_order o
    -> WHERE o.id IN ( SELECT id FROM sample_order ORDER BY id LIMIT 10000,2 );
+--------------+------------+--------------+--------------+------------+
| id           | sellerId   | trade_id     | buyer_id     | buyer_nick |
+--------------+------------+--------------+--------------+------------+
| 242012755468 | 1711939506 | 242012755467 | 244148116334 | zhangsan   |
| 242012759093 | 1711939506 | 242012759092 | 244148138304 | wangwu     |
+--------------+------------+--------------+--------------+------------+
2 rows in set (1.08 sec)
```

执行时间由原来的12秒减少到1.08秒，缩减了一个数量级。

对应的执行计划为：

```
mysql> EXPLAIN SELECT *
    -> FROM sample_order o
    -> WHERE o.id IN ( SELECT id FROM sample_order ORDER BY id LIMIT 10000,2 );
+-----------------------------------------------+-----------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+--------+
| GROUP_NAME                                    | SQL
| PARAMS |
+-----------------------------------------------+-----------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+--------+
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0002_RDS | select
`o`.`id`,`o`.`sellerId`,`o`.`trade_id`,`o`.`buyer_id`,`o`.`buyer_nick` from `sample_order` `o` where (`o`.`id` IN (10002)) | {}
|
| SEQPERF_1478746391548CDTCSEQPERF_OXGJ_0001_RDS | select
```

```
`o`.`id`,`o`.`sellerId`,`o`.`trade_id`,`o`.`buyer_id`,`o`.`buyer_nick` from `sample_order` `o` where (`o`.`id` IN (10001)) | {}
|
+-----------------------------------------------+-----------------------------------------------------------------------------------------------------------+--------+
2 rows in set (0.03 sec)
```

分组及排序优化

在 DRDS 中，如果在一条 SQL 查询中必须同时使用 DISTINCT、GROUP BY 与 ORDER BY，应尽可能保证 DISTINCT、GROUP BY 与 ORDER BY 语句后所带的字段相同，且尽量为拆分键，使最终的 SQL 查询只返回少量数据。这样能够让分布式查询中消耗的网络带宽最小，并且不需要取出大量数据在临时表内进行排序，系统的性能能够达到最优状态。

JOIN 优化

DRDS 的 JOIN 查询分为可下推的 JOIN 和不可下推的 JOIN（即分布式 JOIN）两类，其优化策略各不相同。

可下推的 JOIN 优化

可下推的 JOIN 主要分为以下几类：

- 单表（即非拆分表）之间的 JOIN；
- 参与 JOIN 的表在过滤条件中均带有拆分键作为条件，并且拆分算法相同（即通过拆分算法计算的数据分布在相同分片上）；
- 参与 JOIN 的表均按照拆分键作为 JOIN 条件，并且拆分算法相同；
- 广播表（也称为小表广播）与拆分表之间的 JOIN。

使用 DRDS 时，应尽可能将 JOIN 查询优化成能够在分库上执行的可下推的 JOIN 形式。

以广播表与拆分表之间的 JOIN 为例，应将广播表作为 JOIN 驱动表（将 JOIN 中的左表称为驱动表）。DRDS 的广播表在各个分库都会存放一份同样的数据，当作为 JOIN 驱动表时，该表与分表的 JOIN 可以转化为单库的 JOIN 并进行合并计算，提高查询性能。

例如，有以下的三个表做 JOIN 查询（其中表 sample_area 是广播表，sample_item 和 sample_buyer 是拆分表），查询执行时间约15秒：

```
mysql> SELECT sample_area.name
    -> FROM sample_item i JOIN sample_buyer b ON i.sellerId = b.sellerId JOIN sample_area a ON b.province = a.id
    -> WHERE a.id < 110107
    -> LIMIT 0, 10;
+------+
| name |
+------+
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
```

```
| BJ |
| BJ |
| BJ |
| BJ |
+------+
10 rows in set (14.88 sec)
```

如果调整一下 JOIN 的顺序，将广播表放在最左边作为 JOIN 驱动表，则整个 JOIN 查询在 DRDS 中会被下推为单库 JOIN 查询：

```
mysql> SELECT sample_area.name
    -> FROM sample_area a JOIN sample_buyer b ON b.province = a.id JOIN sample_item i ON i.sellerId = b.sellerId
    -> WHERE a.id < 110107
    -> LIMIT 0, 10;
+------+
| name |
+------+
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
| BJ |
+------+
10 rows in set (0.04 sec)
```

查询执行时间从15秒减少到0.04秒，性能提升非常明显。

注意：广播表在分库上通过同步机制实现数据一致，有秒级延迟。

分布式 JOIN 优化

如果一个 JOIN 查询不可下推（即 JOIN 条件和过滤条件中均不带有拆分键），则需要由 DRDS 完成查询中的部分计算，即分布式 JOIN。

通常将分布式 JOIN 中的表按照数据量大小分为两类：

- 小表：经过条件过滤后，参与 JOIN 计算的中间结果的数据量比较少（一般少于 100 条，或者相较于其它表数据更少）的表；
- 大表：经过条件过滤后，参与 JOIN 计算的中间结果的数据量比较大（一般多于 100 条，或者相较于其它表数据更多）的表。

在 DRDS 层的 JOIN 计算中，大多数情况下采用的 JOIN 算法都是 Nested Loop 及其派生算法（若 JOIN 有排序要求，则使用 Sort Merge 算法）。采用 Nested Loop 算法时，如果 JOIN 中左表的数据量越少，那么 DRDS 对右表做查询的次数就越少，如果右表上建有索引或者表中的数据量也很少，则 JOIN 的速度会更快。因此，在 DRDS 中，分布式 JOIN 的左表被称为驱动表，对分布式 JOIN 的优化应将小表作为驱动表，且让驱动表带有尽可能多的过滤条件。

以下面的分布式 JOIN 为例，查询约需要24秒：

```
mysql> SELECT t.title, t.price
    -> FROM sample_order o,
    ->      ( SELECT * FROM sample_item i WHERE i.id = 242002396687 ) t
    -> WHERE t.source_id = o.source_item_id AND o.sellerId < 1733635660;
+-----------------------------------+--------+
| title                             | price  |
+-----------------------------------|--------+
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
+-----------------------------------+--------+
10 rows in set (23.79 sec)
```

通过初步分析，上述 JOIN 查询是一个 INNER JOIN，并不知道参与 JOIN 计算的中间结果的实际数据量，可以对 o 表与 t 表分别做 COUNT() 查询得到实际数据。

对于 o 表，观察到 WHERE 条件中的 o.sellerId < 1733635660 只与 o 表相关，可以将其提取出来，附加到 o 表的 COUNT() 查询条件中，得到如下的查询结果：

```
mysql> SELECT COUNT(*) FROM sample_order o WHERE o.sellerId < 1733635660;
+----------+
| count(*) |
+----------+
|   504018 |
+----------+
1 row in set (0.10 sec)
```

o 表的中间结果约有50万条记录。类似地，t 表是一个子查询，直接将其抽取出来进行 COUNT() 查询：

```
mysql> SELECT COUNT(*) FROM sample_item i WHERE i.id = 242002396687;
+----------+
| count(*) |
+----------+
|        1 |
+----------+
1 row in set (0.01 sec)
```

t 表的中间结果只有1条记录，所以可确定 o 表为大表，t 表为小表。根据尽量将小表作为分布式 JOIN 驱动表的原则，将 JOIN 查询调整后的查询结果为：

```
mysql> SELECT t.title, t.price
```

```
    -> FROM ( SELECT * FROM sample_item i WHERE i.id = 242002396687 ) t,
    ->      sample_order o
    -> WHERE  t.source_id = o.source_item_id AND o.sellerId < 1733635660;
+----------------------------------+--------+
| title                            | price  |
+----------------------------------|--------+
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
| Sample Item for Distributed JOIN | 239.00 |
+----------------------------------+--------+
10 rows in set (0.15 sec)
```

查询时间从约24秒减少到0.15秒，性能提升非常明显。

子查询优化

在包含子查询的 SQL 优化中，应尽可能将查询下推到具体的分库上执行，并减少 DRDS 层的计算量。要达到这一目标，可以尝试两个方面的优化：

- 将子查询的形式改写为多表 JOIN 形式，并参照 JOIN 优化方法进一步优化；
- 尽量在 JOIN 条件或过滤条件中带上拆分键，有利于 DRDS 将查询下推到特定的分库，避免全表扫描。

以下面的子查询为例：

```
SELECT o.*
FROM sample_order o
WHERE NOT EXISTS
    (SELECT sellerId FROM sample_seller s WHERE o.sellerId = s.id)
```

可将其改写为 JOIN 的形式：

```
SELECT o.*
FROM sample_order o LEFT JOIN sample_seller s ON o.sellerId = s.id
WHERE s.id IS NULL
```

如何排查慢 SQL

DRDS 将执行时间超过1秒的 SQL 定义为慢 SQL。DRDS 中的慢 SQL 分为两种：逻辑慢 SQL 和 物理慢 SQL。

- 逻辑慢 SQL：应用发送到 DRDS 的 慢 SQL；
- 物理慢 SQL：DRDS 发送到 RDS 的 慢 SQL。

实例规格为 2C2G 的实例会记录 5000 条慢 SQL 明细，实例规格为 4C4G 的实例，会记录 10000 条慢 SQL 明细。
DRDS 会滚动删除超过限制数量的慢 SQL 明细。

语法

```
SHOW FULL {SLOW | PHYSICAL_SLOW} [WHERE where_condition]
                    [ORDER BY col_name [ASC | DESC], ...]
                    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

解释

SHOW FULL SLOW显示的是逻辑慢 SQL，即应用发送到 DRDS 的 SQL。

其中SHOW FULL SLOW的结果集会包含以下列，其含义如下：

- TRACE_ID: 该 SQL 的唯一标记，同一个逻辑 SQL 以及该逻辑 SQL 产生的物理 SQL 的 TRACE_ID 相同，同时 TRACE_ID 也会以注释的形式发送到 RDS，在 RDS 的 SQL 明细中可以根据 TRACE_ID 找到该 SQL；
- HOST: 发送该 SQL 的客户端的 IP，注意：在 VPC 模式下可能无法获取客户端 IP；
- START_TIME: DRDS 收到这个 SQL 的时间；
- EXECUTE_TIME: DRDS 执行该 SQL 消耗的时间；
- AFFECT_ROW: 该 SQL 返回的记录数或者影响的行数；
- SQL: 执行的语句。

SHOW FULL PHYSICAL_SLOW指的是物理慢 SQL，即 DRDS 发送到 RDS（MySQL）的 SQL。

SHOW FULL PHYSICAL_SLOW 的结果集会包含以下列，其含义如下：

- TRACE_ID: 该 SQL 的唯一标记，同一个逻辑 SQL 以及该逻辑 SQL 产生的物理 SQL 的 TRACE_ID 相同，同时 TRACE_ID 也会以注释的形式发送到 RDS，在 RDS 的 SQL 明细中可以根据 TRACE_ID 找到该 SQL；
- GROUP_NAME: 数据库分组名，分组的目标是管理多组数据完全相同的数据库，比如通过 RDS（MySQL）进行数据复制后的主备数据库，主要用来解决读写分离，主备切换的问题；
- DBKEY_NAME: 执行的分库信息；
- START_TIME: DRDS 开始执行这个 SQL 的时间；
- EXECUTE_TIME: DRDS 执行该 SQL 消耗的时间；
- SQL_EXECUTE_TIME: DRDS 调用 RDS 执行该 SQL 消耗的时间；

- GETLOCK_CONNECTION_TIME: DRDS 从连接池获取连接消耗的时间，该值如果很大，说明 RDS 的连接已被耗尽，一般是慢 SQL 比较多引起，登录到相应的 RDS，结合SHOW PROCESSLIST指令来排查。
- CREATE_CONNECTION_TIME: DRDS 建立 RDS 连接消耗的时间，该值如果很大，很大原因是底层的 RDS 压力比较大或者挂掉了；
- AFFECT_ROW: 该 SQL 返回的记录数或者影响的行数；
- SQL: 执行的语句。

示例

示例一：下面的例子展示了如何一步步定位慢 SQL 在 DRDS 上、DRDS 跟 RDS 之间的执行情况。

更多关于排查慢 SQL 的例子请参考文档**排查 DRDS 慢 SQL**，进行 SQL 调优的例子请参考文档 **SQL 优化方法**。

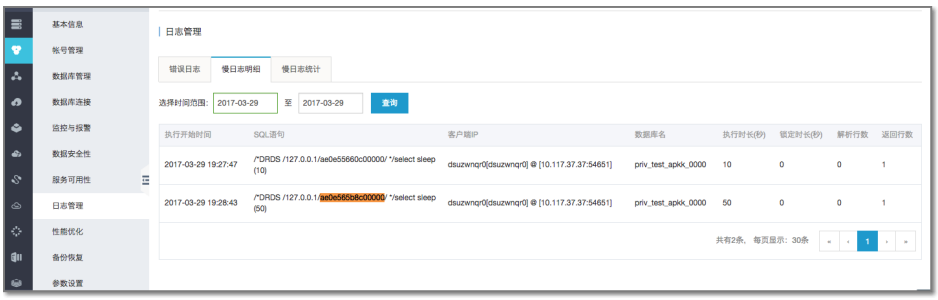
通过一些条件，例如执行时间，SQL 字符串匹配等方式来获取我们想要的慢 SQL；

```
mysql> show full slow where `SQL` like '%select sleep(50)%';
+-----------------+-----------+-------------------------+--------------+------------+------------------+
| TRACE_ID        | HOST      | START_TIME              | EXECUTE_TIME | AFFECT_ROW | SQL              |
+-----------------+-----------+-------------------------+--------------+------------+------------------+
| ae0e565b8c00000 | 127.0.0.1 | 2017-03-29 19:28:43.028 |        50009 |          1 | select sleep(50) |
+-----------------+-----------+-------------------------+--------------+------------+------------------+
1 row in set (0.02 sec)
```

根据逻辑慢 SQL 中获取到的TRACE_ID，执行SHOW FULL PHYSICAL_SLOW指令获取这个 SQL 的物理执行情况；

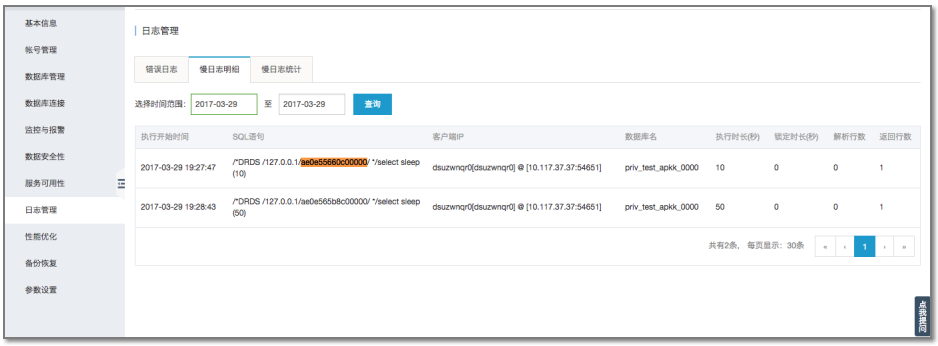
```
mysql> show full physical_slow where trace_id = 'ae0e565b8c00000';
+-----------------+-------------------------------------------------------+-------------------------------------------------------
---+-----------------------+-------------+-----------------+------------------------+-----------------------
+------------+------------------+
| TRACE_ID        | GROUP_NAME                                      | DBKEY_NAME                                    |
START_TIME              | EXECUTE_TIME | SQL_EXECUTE_TIME | GETLOCK_CONNECTION_TIME |
CREATE_CONNECTION_TIME | AFFECT_ROW | SQL              |
+-----------------+-------------------------------------------------------+-------------------------------------------------------
---+-----------------------+-------------+-----------------+------------------------+-----------------------
+------------+------------------+
| ae0e565b8c00000 | PRIV_TEST_1489167306631PJAFPRIV_TEST_APKK_0000_RDS |
rdso6g5b6206sdq832ow_priv_test_apkk_0000_nfup | 2017-03-29 19:27:53.02 |        50001 |           50001 |
0 |                    0 |          1 | select sleep(50) |
+-----------------+-------------------------------------------------------+-------------------------------------------------------
---+-----------------------+-------------+-----------------+------------------------+-----------------------
+------------+------------------+
1 row in set (0.01 sec)
```

可以在 RDS 的 SQL 明细与慢 SQL 中，根据这个TRACE_ID查看这个 SQL 在 RDS 上的执行情况。



示例二：在 RDS 中找到一个慢 SQL，如何找到 DRDS 中的原始 SQL。

在 RDS 的管理控制台 -> 日志管理 -> 慢 SQL 明细页面中，观察到慢 SQL 的TRACE_ID为 ae0e55660c00000。



根据第一步获取到的TRACE_ID，执行SHOW FULL PHYSICAL_SLOW指令获取这个 SQL 的物理执行情况；

```
mysql> show full physical_slow where trace_id = 'ae0e55660c00000';
+-----------------+-----------------------------------------------+-----------------------------------------------
------+-------------------------+-------------+-----------------+-------------------------+------------------
-----+------------+------------------+
| TRACE_ID        | GROUP_NAME                                    | DBKEY_NAME                                    |
START_TIME              | EXECUTE_TIME | SQL_EXECUTE_TIME | GETLOCK_CONNECTION_TIME |
CREATE_CONNECTION_TIME | AFFECT_ROW | SQL              |
+-----------------+-----------------------------------------------+-----------------------------------------------
------+-------------------------+-------------+-----------------+-------------------------+------------------
-----+------------+------------------+
| ae0e55660c00000 | PRIV_TEST_1489167306631PJAFPRIV_TEST_APKK_0000_RDS |
rdso6g5b6206sdq832ow_priv_test_apkk_0000_nfup | 2017-03-29 19:27:37.308 |        10003 |
10001 |                    0 |                    0 |          1 | select sleep(10) |
+-----------------+-----------------------------------------------+-----------------------------------------------
------+-------------------------+-------------+-----------------+-------------------------+------------------
-----+------------+------------------+
1 row in set (0.02 sec)
```

在 DRDS 中，一条 SQL 语句会在 DRDS 和 RDS 节点上逐步执行。任意节点上的执行损耗过大都会导致慢

SQL。

慢 SQL 的一般排查步骤为：

定位慢 SQL；

定位性能损耗节点；

定位性能损耗原因并处理。

说明：排查过程中，建议通过 MySQL 命令行进行连接：mysql -hIP -PPORT -uUSER -pPASSWORD -c 。请务必加上 “-c” ，防止 MySQL 客户端过滤掉注释（默认）从而影响 HINT 的执行。

定位慢 SQL

定位慢 SQL 一般有两种场景：历史信息可从慢 SQL 记录中查询；实时慢 SQL 执行信息可使用 SHOW PROCESSLIST 指令展示。

查看慢 SQL 记录

执行以下指令查询慢 SQL Top 10。此查询针对 DRDS 层面的逻辑 SQL 。一个逻辑 SQL 对应一个或者多个 RDS 库表的 SQL 执行。详情见慢 SQL 明细文档。

```
mysql> SHOW SLOW limit 10;
+-----------------+--------------+-------------------------+--------------+------------+-------------------------------------------+
| TRACE_ID        | HOST         | START_TIME              | EXECUTE_TIME | AFFECT_ROW | SQL                                       |
+-----------------+--------------+-------------------------+--------------+------------+-------------------------------------------+
| ac3133132801001 | 42.120.74.97 | 2017-03-06 15:48:32.330 |       900392 |         -1 | select detail_url, sum(price) from t_item group by detail_url;  |
......
+-----------------+--------------+-------------------------+--------------+------------+-------------------------------------------+
10 rows in set (0.01 sec)
```

查看当前实时 SQL 执行信息

如果当前服务器中正在执行的 SQL 比较慢，可以使用 SHOW PROCESSLIST 指令来查看当前 DRDS 数据库中实时的执行信息。其中 TIME 列代表的是该 SQL 已经执行的时间。

```
mysql> SHOW PROCESSLIST WHERE COMMAND != 'Sleep';
+---------------+-----------+-------------------+-------------+---------+-----------------------------------
---------------------------------+-----------------------------------------------------------------------------------------------------
```

```
--------------------+-----------+---------------+-----------+
| ID           | USER      | DB              | COMMAND     | TIME    | STATE
| INFO                                                                                                | ROWS_SENT | ROWS_EXAMINED |
ROWS_READ |
+---------------+-----------+-------------------+-------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------
--------------------+-----------+---------------+-----------+
| 0-0-352724126 | ifisibhk0 | test_123_wvvp_0000 | Query       |     13  | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,SUM(`t_item`.`price`) from `t_i |
NULL |          NULL |       NULL |
| 0-0-352864311 | cowxhthg0 | NULL              | Binlog Dump |     17  | Master has sent all binlog to slave;
waiting for binlog to be updated | NULL                                                                                                |
NULL |          NULL |       NULL |
| 0-0-402714795 | ifisibhk0 | test_123_wvvp_0005 | Alter       |     114 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */ALTER TABLE `Persons` ADD `Birthday` date                   |
NULL |          NULL |       NULL |
......
+---------------+-----------+-------------------+-------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------
--------------------+-----------+---------------+-----------+
12 rows in set (0.03 sec)
```

各列的信息如下：

ID：连接标识。

USER：执行该 SQL 的分库用户名。

DB：指定的数据库，如果没有指定则为 NULL。

COMMAND：正在执行的命令类型。SLEEP 代表空闲连接。其它命令详情见 MySQL 线程信息文档。

TIME：SQL 已执行的时间，单位是秒。

STATE：当前的执行状态。详见 MySQL 线程状态文档。

INFO：正在执行的 SQL 语句，有可能因为过长而无法完全显示，此时可以结合业务参数等信息把完整 SQL 推导出来。

在当前的示例中定位到以下的慢 SQL：

ALTER TABLE `Persons` ADD `Birthday` date

定位性能损耗节点

从慢 SQL 记录或者实时 SQL 执行信息中定位到慢 SQL 后，可以执行 TRACE 指令跟踪该 SQL 在 DRDS 和 RDS 上的运行时间，以便定位瓶颈。TRACE 命令会**实际执行** SQL，在执行过程中记录所有节点消耗的时间，并返回执行结果。TRACE 及其他控制指令详情见自定义控制指令文档。

说明： DRDS TRACE 命令需要保持连接的上下文信息，某些 GUI 客户端可能会使用连接池，导致命令不正常。因此建议使用 MySQL 命令行执行。

针对上文定位的慢 SQL，可以执行以下指令：

```
mysql> trace select detail_url, sum(distinct price) from t_item group by detail_url;
+---------------+---------------+
| detail_url    | sum(price)    |
+---------------+---------------+
| www.xxx.com   | 1084326800.00 |
| www.xx1.com   | 1084326800.00 |
| www.xx2.com   | 1084326800.00 |
| www.xx3.com   | 1084326800.00 |
| www.xx4.com   | 1084326800.00 |
| www.xx5.com   | 1084326800.00 |
......
+---------------+---------------+
1 row in set (7 min 2.72 sec)
```

TRACE 指令执行完毕后，可以执行 SHOW TRACE 命令查看结果，根据每个组件的时间消耗来判断慢 SQL 的瓶颈。

```
mysql> SHOW TRACE;
+------+------------+-------------+-----------------------------------------------------------+--------------------------
------------------+---------------+-------------------------+------+-----------------------------------------------------
-------------------------------------------------------------------------------+--------+
| ID   | TIMESTAMP  | TYPE        | GROUP_NAME                                                | DBKEY_NAME
| TIME_COST(MS) | CONNECTION_TIME_COST(MS) | ROWS | STATEMENT
| PARAMS |
+------+------------+-------------+-----------------------------------------------------------+--------------------------
------------------+---------------+-------------------------+------+-----------------------------------------------------
-------------------------------------------------------------------------------+--------+
|    0 |      0.000 | Optimize    | DRDS                                                      | DRDS                                | 2            |
0.00                  |    0 | select detail_url, sum(price) from t_item group by detail_url
| NULL   |
|    1 | 423507.342 | Merge Sorted | DRDS                                                    | DRDS                                |
411307        | 0.00                  |    8 | Using Merge Sorted, Order By (`t_item`.`detail_url` asc )
| NULL   |
|    2 |      2.378 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0003_hbpz | 15            | 1.59                  |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
|    3 |      2.731 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0000_hbpz | 11            | 1.78                  |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
|    4 |      2.933 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0004_hbpz | 15            | 1.48                  |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
```

```
`t_item`.`detail_url` asc  | NULL   |
   |    5 |      3.111 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0001_hbpz | 15          | 1.56                |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
   |    6 |      3.323 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0007_hbpz | 15          | 1.54                |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
   |    7 |      3.496 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0006_hbpz | 18          | 1.30                |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
   |    8 |      3.505 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0005_hbpz | 423507       | 1.97                |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
   |    9 |      3.686 | Query       | TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS |
rdso6g5b6206sdq832ow_test_123_wvvp_0002_hbpz | 14          | 1.47                |    1 | select
`t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by
`t_item`.`detail_url` asc  | NULL   |
   |   10 | 423807.906 | Aggregate    | DRDS                                                | DRDS                                | 1413
| 0.00                |    1 | Aggregate Function (SUM(`t_item`.`price`)), Group By (`t_item`.`detail_url` asc )
| NULL   |
   +------+------------+-------------+-----------------------------------------------------+-------------------------
------------------+--------------+-------------------------+------+-----------------------------------------------------
-----------------------------------------------------------------------------------+--------+
   11 rows in set (0.01 sec)
```

SHOW TRACE 返回的结果中，根据 TIME_COST （单位毫秒）列可以判断哪个节点上的执行时间消耗大。同时可以看到对应的 GROUP_NAME （即 DRDS/RDS 节点），以及 STATEMENT 列信息（即正在执行的 SQL）。通过 GROUP_NAME 是否等于 DRDS 可以判断该慢节点存在于 DRDS 还是 RDS。

在以上结果中，分析可知是 DRDS 节点的 Merge Sorted 动作和 RDS 的 TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS 节点上消耗了大量时间。

定位性能损耗原因并处理

DRDS 慢节点处理

当慢 GROUP_NAME 是 DRDS 时，请检查执行过程中是否存在 Merge Sorted、Temp Table Merge、Aggregate 等计算耗时操作。如果存在的话请参考 SQL 优化文档进行优化。

RDS 慢节点处理

当慢节点在 RDS 时，请检查该 SQL 语句在 RDS 上的**执行计划**。

在 DRDS 中，可以使用 /!TDDL:node={GROUP_NAME}*/ EXPLAIN 来查看某个 RDS 的执行计划。执行计划展示了 RDS 执行该 SQL 的过程信息，包括表间关联及索引信息等。

详细过程如下：

依据 GROUP_NAME 组装
HINT：/!TDDL:node='TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS'*/

将组装好的 HINT 及带 EXPLAIN 前缀的 STATEMENT 拼装成新的 SQL 并执行。EXPLAIN 指令不会真正执行，而只是显示该 SQL 的执行计划信息。

以上文定位的慢节点为例查询执行计划：

```
mysql> /!TDDL:node='TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS'*/ EXPLAIN  select `t_item`.`detail_url`,SUM(distinct `t_item`.`price`) from `t_item` group by `t_item`.`detail_url` order by `t_item`.`detail_url` asc;
+----+-------------+--------+------+---------------+-----+---------+-----+---------+----------------------------------+
| id | select_type | table  | type | possible_keys | key | key_len | ref | rows    | Extra                            |
+----+-------------+--------+------+---------------+-----+---------+-----+---------+----------------------------------+
|  1 | SIMPLE      | t_item | ALL  | NULL          |NULL | NULL    | NULL| 1322263 | Using temporary; Using filesort |
+----+-------------+--------+------+---------------+-----+---------+-----+---------+----------------------------------+
1 row in set (0.01 sec)
```

根据观察发现以上 SQL 在 RDS 中执行时，出现了 Using temporary; Using filesort 现象，说明没有正确的使用索引从而导致执行缓慢。此时可以修正索引问题后重新执行。

如果观察执行计划后仍然无法判断 RDS 执行时间过长的原因，请查阅 RDS 性能调优文档。

数据库连接池是对数据库连接进行统一管理的技术，主要目的是提高应用性能，减轻数据库负载。

资源复用：连接可以重复利用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，同时增进了系统的平稳性。

提高系统响应效率：连接的初始化工作完成后，所有请求可以直接利用现有连接，避免了连接初始化和释放的开销，提高了系统的响应效率。

避免连接泄漏：连接池可根据预设的回收策略，强制回收连接，从而避免了连接资源泄漏。

连接池推荐

将应用和数据库连接进行业务操作，建议使用连接池。如果是 Java 程序，推荐使用 Druid 连接池。

Druid 的 Spring 标准配置

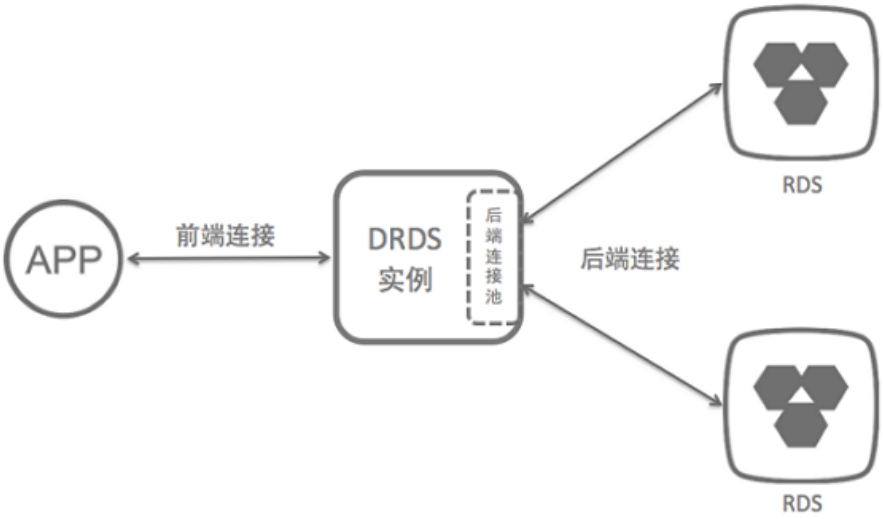
```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-
method="close">
<property name="driverClassName" value="com.mysql.jdbc.Driver" />
<!-- 基本属性 URL、user、password -->
<property name="url"
value="jdbc:mysql://ip:port/db?autoReconnect=true&amp;rewriteBatchedStatements=true&amp;socketTimeout=3
0000&amp;connectTimeout=3000" />
<property name="username" value="root" />
<property name="password" value="123456" />
<!-- 配置初始化大小、最小、最大 -->
<property name="maxActive" value="20" />
<property name="initialSize" value="3" />
<property name="minIdle" value="3" />
<!-- maxWait 获取连接等待超时的时间 -->
<property name="maxWait" value="60000" />

<!-- timeBetweenEvictionRunsMillis 间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
<property name="timeBetweenEvictionRunsMillis" value="60000" />
<!-- minEvictableIdleTimeMillis 一个连接在池中最小空闲的时间，单位是毫秒-->
<property name="minEvictableIdleTimeMillis" value="300000" />
<!-- 检测连接是否可用的 SQL -->
<property name="validationQuery" value="SELECT 'z'" />
<!-- 是否开启空闲连接检查 -->
<property name="testWhileIdle" value="true" />
<!-- 是否在获取连接前检查连接状态 -->
<property name="testOnBorrow" value="false" />
<!-- 是否在归还连接时检查连接状态 -->
<property name="testOnReturn" value="false" />
</bean>
```

当应用程序连接 DRDS 实例执行操作时，从 DRDS 实例的角度看，会有两种类型的连接：

前端连接：由应用程序建立的，到 DRDS 实例中逻辑库的连接；

后端连接：由 DRDS 实例中的节点建立的，到后端 RDS 实例中物理库的连接。



前端连接

前端连接的数量理论上仅受限于 DRDS 实例节点可用的内存大小和网络连接数。但在实际的应用场景中，应用程序连接到 DRDS 实例时，通常会管理有限数量的连接来执行请求的操作，并不会维持很高并发量的持久化长连接（例如，数万个并发的长连接），因此可认为 DRDS 实例能接受的前端连接数量是无限制的。

由于前端连接数量不受限制，可以允许有大量空闲连接存在，因此适用于业务端部署应用程序的服务器数量较多，需要同时保持连接到 DRDS 实例的场景。

注意：虽然前端连接的数量可被认为是无限制的，但从前端连接获取的操作请求是由 DRDS 实例的内部线程通过后端连接实际执行，而内部线程和后端连接的数量有限，因此 DRDS 实例处理请求的整体并发度是有限的。

后端连接

DRDS 实例的每个节点内部都会创建后端连接池，自动管理和维护到 RDS 实例中物理库的后端连接。因此，DRDS 实例中后端连接池的最大连接数与 RDS 实例支持的最大连接数直接相关。可参照以下公式来计算 DRDS 实例中后端连接池的最大连接数：

```
DRDS 实例后端连接池的最大连接数 = 向下取整(RDS 实例最大连接数 / RDS 实例物理分库数 / DRDS 实例节点数)
```

例如，某用户搭配购买了如下规格的 RDS 实例和 DRDS 实例：

- 1个 RDS 实例，包含8个物理分库，规格为 4Core16G，最大连接数为 4000；
- 1个 DRDS 专享实例，规格为 32Core32G（每 2Core2G 为1个 DRDS 节点，即该实例包含16个 DRDS 节点）。

按照上述公式可计算出 DRDS 实例中后端连接池的最大连接数：

```
DRDS 实例后端连接池的最大连接数 = FLOOR(4000 / 8 / 16) = FLOOR(31.25) = 31
```

注意：

上述公式计算的结果为 DRDS 实例中后端连接池的最大连接数的上限。实际使用中，为了减轻 RDS 实例的连接压力，留出一定的缓冲余地，DRDS 实例会适当调整后端连接池的最大连接数，使其小于上限值。

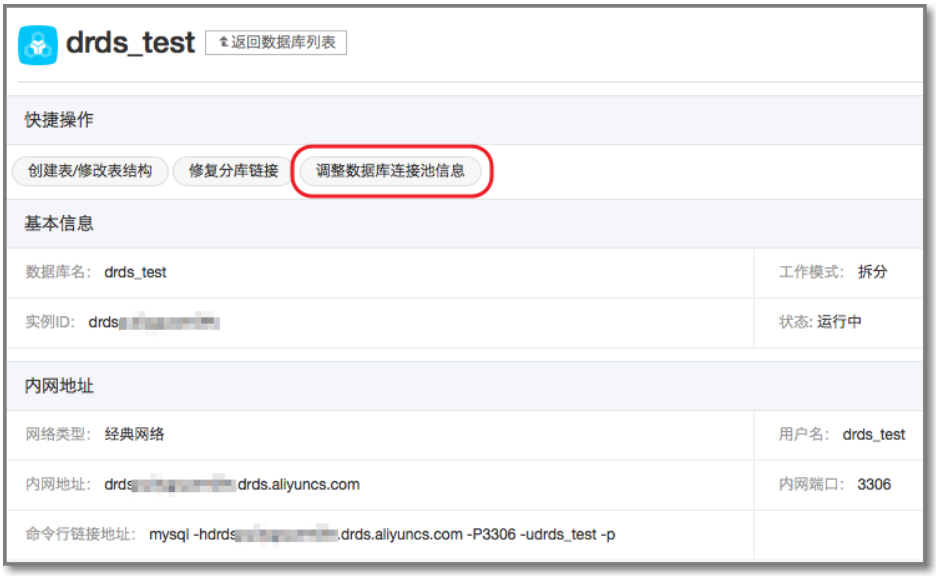
建议 DRDS 实例中的数据库创建在专用的 RDS 实例上，即专用 RDS 实例上不应创建其它应用或其它 DRDS 实例的数据库。

调整后端连接数

若增大 RDS 实例的最大连接数后，想调整 DRDS 实例中后端连接池的最大连接数，请按以下步骤操作。

1. 登录 DRDS 控制台，在数据库列表页选择相应的数据库。

   在数据库**基本信息**页，单击**调整数据库连接池信息**。DRDS 控制台会自动进行调整。



前后端连接的关系

在应用程序与 DRDS 实例建立了前端连接并发出 SQL 语句的执行请求后，DRDS 实例节点会异步地处理请求，并通过内部后端连接池获取后端连接，在一个或多个物理库上执行经过优化处理的 SQL 语句。

由于 DRDS 实例节点内部为异步的处理流程，前端连接和后端连接并无绑定关系，因此在通常的短事务和简单查询情况下，少量的后端连接有能力处理并发量较高的前端连接所带来的大量请求。这也是在 DRDS 中应关注 QPS 指标，而非并发连接数的原因。

虽然前端连接数量可被认为近乎无限制，但 DRDS 实例节点内部的后端连接池所维护的最大连接数是有限的（参见上文“后端连接”一节的说明），因此在实际的应用场景中，需要注意以下几点：

应用程序中应尽量避免长的（或称为大的）事务，以免其长时间未提交或回滚而占用后端连接，造成后端连接池紧张或达到上限，降低整体的并发处理能力，增加响应时间；

应监控并优化或消除在 DRDS 中执行的慢查询，以免其长时间执行而占用后端连接，造成后端连接池紧张或达到上限。这种情况下 DRDS 实例或 RDS 实例会面临更大的处理压力，导致整体的并发处理能力下降，增加响应时间，还可能因为执行超时导致 SQL 执行失败率上升。对于慢查询的排查和优化，请参考 排查 DRDS 慢 SQL 和 SQL 优化；

若在正常使用连接和执行查询的情况下，对后端连接的使用仍然达到了 DRDS 实例中后端连接池的最大连接数，请联系客服协助解决。

数据库性能主要可以从响应时间（RT）和容量（QPS）两个指标进行衡量。RT 指标反映的是单个 SQL 的性能，这类性能问题可以通过 SQL 优化等方法进行解决。DRDS 升配则主要通过扩充容量来提升性能，适用于低延时高 QPS 类型的数据库访问业务。

DRDS 实例性能取决于 DRDS 本身和 RDS 的性能表现，任一 DRDS 或者 RDS 节点性能不足都会导致整体性能出现瓶颈。本文主要说明如何观察 DRDS 实例的性能指标，并通过升配来解决性能不足的问题。RDS 的性能判断及升配方法请参考 RDS 文档。

判断 DRDS 实例性能瓶颈

DRDS 实例的 QPS 和 CPU 性能是正相关的。当 DRDS 性能出现瓶颈时，主要表现为实例的 CPU 利用率居高不下。

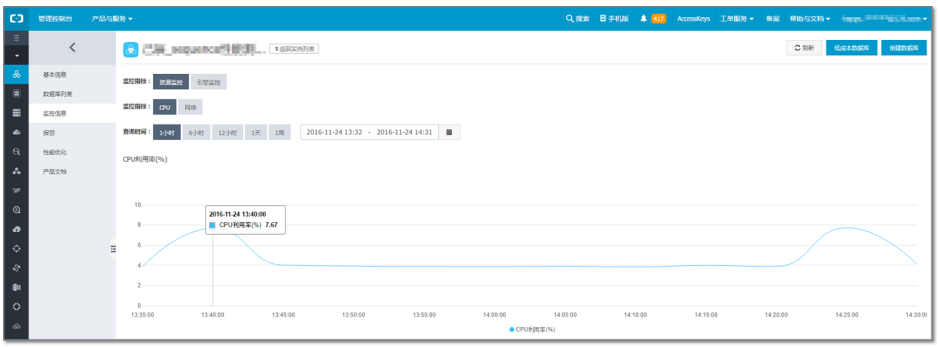
观察 CPU 利用率监控项

在 DRDS 控制台左侧菜单栏选择**实例列表**。

单击需要查看的实例名称进入实例基本信息页。

在左侧菜单栏选择**监控信息**。

如果发现 CPU 利用率**超出90%**或**持续超出80%**，则意味着当前实例性能出现瓶。在 RDS 不存在瓶颈的情况下，可以判断当前的 DRDS 实例规格无法满足业务的 QPS 性能需求，需要通过升配解决。



更多性能相关的业务监控场景及配置 DRDS CPU 利用率报警的方法请参考 DRDS 实例监控。

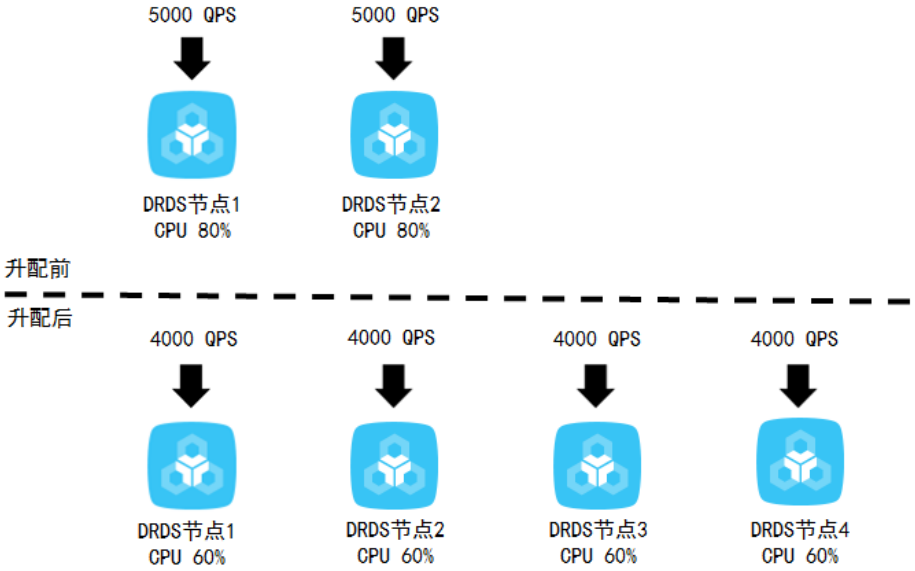
DRDS 升配

QPS 是衡量 DRDS 实例规格的重要指标。每种实例规格对应一定的 QPS 参考值，具体请参考文档 DRDS 实例规格。

注意：有些特殊的 SQL 语句在 DRDS 层面需要更多的计算（如临时表排序、聚合计算等），此时每个 DRDS 实例可以支撑的 QPS 相比规格中的标准值会有所下降。

DRDS 升配以增加处理节点，均摊 QPS 的方式来提高实例的处理性能。由于 DRDS 节点本身是无状态的，因此这种升配方式对 DRDS 实例的性能会有线性的提升。

例如业务 A 需要1.5万左右的 QPS 性能，当前 DRDS 实例规格为 4C4G，两个节点， QPS 只能达到1万。通过观察发现 DRDS 的 CPU 占用一直处于高位后，升配到 8C8G，升配后实例节点约各承担4000的 QPS。此时性能满足了用户的需求，同时 CPU 利用率也下降到合理位。如下图所示：



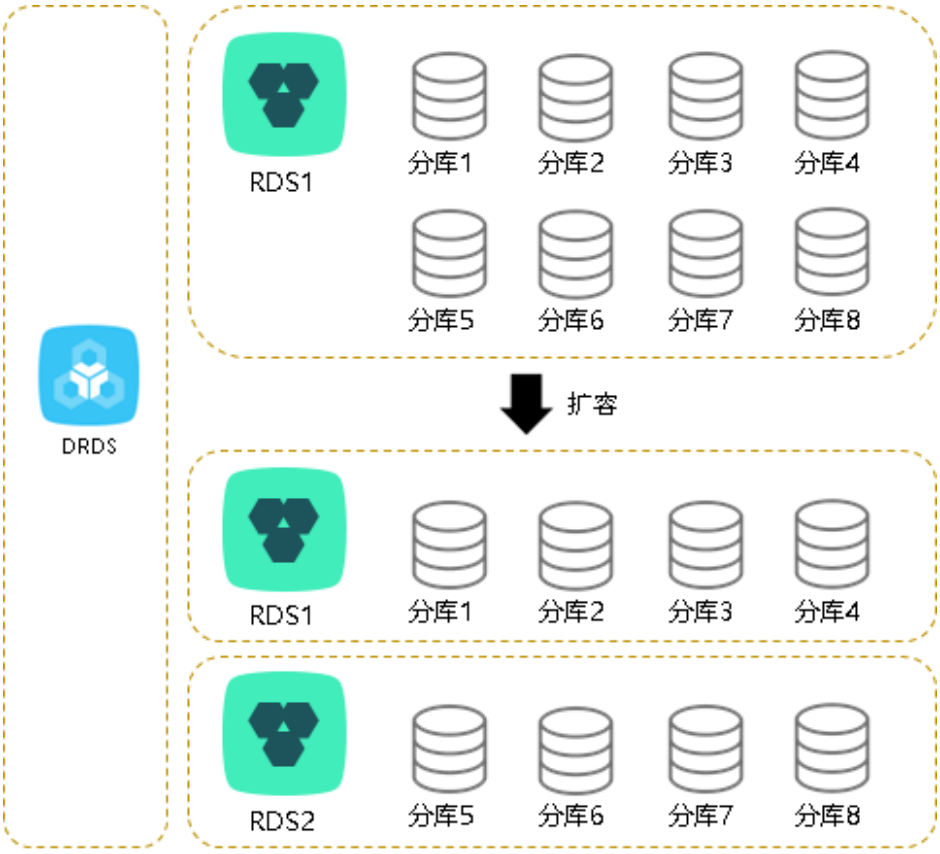
实例升配的具体操作请参考 DRDS 实例变配。

什么是扩容

DRDS 平滑扩容是指通过增加 RDS 的数量以提升整体性能。当 RDS 的 IOPS、CPU、磁盘容量等指标到达瓶

颈，并且 SQL 优化、RDS 升配已无法解决瓶颈（例如磁盘已升至顶配）时，可通过 DRDS 水平扩容增加 RDS 数量，提升 DRDS 数据库的容量。

DRDS 平滑扩容通过迁移分库到新 RDS 来降低原 RDS 的压力。例如，扩容前8个库的压力集中在一个 RDS 实例上，扩容后8个库分别部署在两个 RDS 实例上，单个 RDS 实例的压力就明显降低。如下图所示：



说明：扩容多次后，如果出现 RDS 数量和分库数量相等的情况，需要创建另外一个 DRDS 和预期容量 RDS 的数据库，再进行数据迁移以达到更大规模数据容量扩展的目标。此过程较复杂，推荐创建 DRDS 数据库时要考虑未来2-3年数据的增长预期，做好 RDS 数量规划。

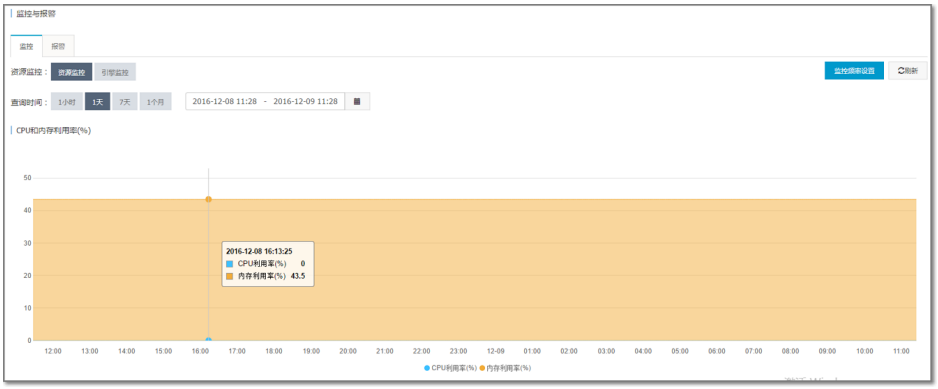
判断是否需要扩容

DRDS 是否需要进行平滑扩容，可以通过观察 RDS 的三个指标进行判断：IOPS、CPU、磁盘空间。在 RDS 控制台可以查看这些指标，详情 请参见 RDS 相关文档。

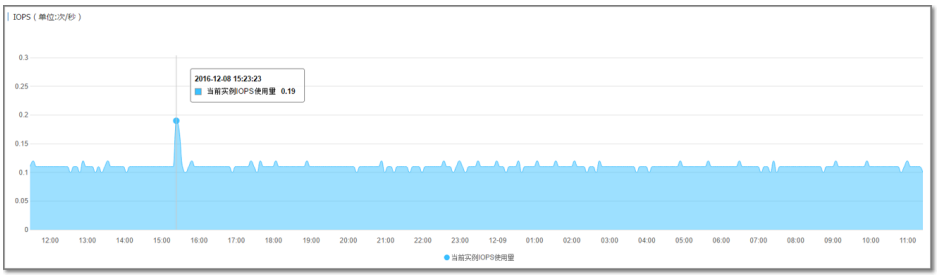
在 RDS 控制台也可以针对这些指标设置专门的报警项，报警策略及默认阈值见下图。

CPU 及 IOPS 指标

RDS 平台 CPU 监控信息。



RDS 对 IO 的频率做了限制，体现在 IOPS 这个指标上。



对于以上两个指标，如果发现任何一个指标长期保持在80%以上或频繁收到报警信息，请考虑通过以下步骤来解决：

尝试 SQL 优化。CPU 利用率过高的问题通常都可以通过这一步解决，详情请参考 MySQL CPU 使用率高的解决方法及MySQL IOPS 使用率高的解决方法。

SQL 优化无法解决问题时，可以升高 RDS 的相关规格。详情请参考 RDS 升配文档。

当 CPU 和 IOPS 超标时，可以通过设置只读库的方式来分担主库负荷。注意读写分离会影响读一致性。详情请参见设置读写分离文档。

当以上步骤无法很好地解决问题时，请考虑进行 DRDS 扩容。

磁盘空间

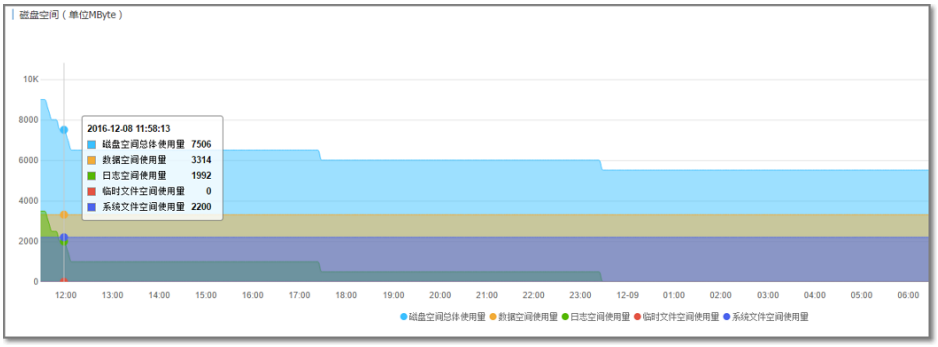
RDS 的磁盘空间有以下几种：

数据空间：数据所占用的空间。随着数据的插入，空间占用量会一直增长。磁盘存储容量的余量建议保持在30%以上。

系统文件空间：包括共享表空间、错误日志文件等。

Binlog 文件空间：这是数据库运行过程中产生的空间。更新事务越多，空间占用量就越大。

判断是否需要扩容主要关注**数据空间**即可。当数据空间将要或预期要超出磁盘容量时，可以通过扩容的方式将数据分散到多个 RDS。



扩容风险及注意事项

DRDS 扩容流程分为 **配置**>**迁移**>**切换**>**清理** 四个步骤。具体请参考**扩容操作**文档。

在扩容前请注意以下事项：

如果需要新增5个或5个以上 RDS 实例，需要事先提工单，以防后端迁移资源不足造成迁移不成功。

源 RDS 实例扩容过程中会有读压力，请尽量在源 RDS 低负载时操作。

扩容期间请勿在控制台提交 DDL 任务或连接 DRDS 直接执行 DDL SQL，否则会导致扩容任务失败。

扩容需要源库表中有主键，如果没有需要事先加好主键。

扩容的切换动作会将读写流量切换到新增的 RDS 上，切换过程大约持续3~5分钟，建议在停业务的情况下进行切换。

在执行切换前，扩容动作不会对 DRDS 产生任何影响。因此在切换前都可以通过回滚来放弃本次扩容。

扩容操作对数据库有一定压力，建议在业务低谷期操作。

DDL 原理简介

DRDS 的 DDL 指令会在所有分表上执行对应的 DDL 操作。失败的情况可以分为两类：

1. DDL 在分库执行失败。DDL 在任意分库执行出错都可能导致各分表结构不一致。
2. 执行长时间无响应。在对大表执行 DDL 操作时，有可能由于分库的执行时间过长导致 DDL 长时间无响应。

分库执行报错的原因多种多样，如建表时表已存在、加列时列已存在等各类冲突、磁盘空间不足等。

长时间无响应一般是由分库的执行时间过长导致的。以 MySQL（RDS 原理与之类似）为例，DDL 的耗时大部分取决于该操作是 In-Place（直接在源表修改）还是 Copy Table（拷贝表数据）。In-Place 只需要修改元数据就可以了，而 Copy Table 需要重建整张表数据，此外还涉及日志和 buffer 操作。

各类操作与这两项因素的关系详见 MySQL 官方文档 Summary of Online Status for DDL Operations。

判断 DDL 操作是 In-place 还是 Copy Table 操作，可以查看操作结束后 “rows affected” 这一项的返回值。

示例：

改变某列的默认值（非常快，完全不会影响表数据）：

Query OK, 0 rows affected (0.07 sec)

增加一个索引（需要花些时间 ，但是 0 rows affected 说明表数据没有被复制）：

Query OK, 0 rows affected (21.42 sec)

改变某列的数据类型（耗费大量时间并且需要重建表中的所有数据行）：

Query OK, 1671168 rows affected (1 min 35.54 sec)

因此，执行一个大表 DDL 操作前，可以先通过以下步骤判定这是一个快速或慢速操作：

1. 复制表结构生成一张克隆表。
2. 插入一些数据。
3. 在克隆表上执行这个 DDL 操作。
4. 检查操作完成后 “rows affected” 值是否是0。一个非0的值意味着该操作需要重建整张表，这时可能需要考虑在流量低谷去执行该操作。

失败处理

DRDS DDL 操作会将所有 SQL 分发到所有分库上并行执行。任一分库上执行失败不会影响其他分库。另外，DRDS 还提供了 CHECK TABLE 指令来检测分表结构的一致性。因此，失败的 DDL 操作可以重新执行，已经执行成功的分库上失败报错并不会影响其他分库。只要保证最终所有分表结构一致即可。

DDL 失败处理步骤

1. 使用 **CHECK TABLE** 指令检查表结构。如果返回结果只有一行且为状态正常则可认为**表状态一致**。

此时进行步骤2，否则进行步骤3。

2. 使用 **SHOW CREATE TABLE** 指令检查表结构。如果显示的表结构符合 DDL 执行后的预期则可认为 **DDL 执行成功**，否则继续进行步骤3。
3. 使用 **SHOW PROCESSLIST** 指令观察所有当前执行的 SQL 状态。如有仍在执行的 DDL 操作，请等候其执行完成后再进行步骤1、2，检查表结构是否符合预期，否则进行步骤4。
4. 在 DRDS 上重新执行 DDL 操作。如果出现 **Lock conflict** 的报错请进行步骤5，否则进行步骤3。
5. 使用 **RELEASE DBLOCK** 指令释放 DDL 操作锁，然后进行步骤4。

详细操作如下：

检查表结构一致性

使用 CHECK TABLE 指令检查表结构，当返回结果只有一行且显示状态 OK 时，表明**表结构一致**。

注意：如果在 DMS 上执行 CHECK TABLE 没有返回结果，请在命令行下重试。

```
mysql> check table `xxxx`;
+---------------------------+-------+----------+----------+
| TABLE                     | OP    | MSG_TYPE | MSG_TEXT |
+---------------------------+-------+----------+----------+
| TDDL5_APP.xxxx            | check | status   | OK       |
+---------------------------+-------+----------+----------+
1 row in set (0.05 sec)
```

检查表结构

使用 SHOW CREATE TABLE 指令检查表结构，如果**表结构一致**且**表结构无误**时，可认为 DDL 已执行成功。

```
mysql> show create table `xxxx`;
+---------+-----------------------------------------------------------------------------------------------------------------------------------------
--------+
| Table   | Create Table                                                                                                                            |
+---------+-----------------------------------------------------------------------------------------------------------------------------------------
--------+
|  xxxx   | CREATE TABLE `xxxx` (
`id` int(11) NOT NULL DEFAULT '0',
`NAME` varchar(1024) NOT NULL DEFAULT '',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`id`) tbpartitions 3
|
+---------+-----------------------------------------------------------------------------------------------------------------------------------------
--------+
1 row in set (0.05 sec)
```

观察当前正在执行的 SQL 语句

有些 DDL 执行速度过慢，发现 DDL 长时间无响应后，可执行 SHOW PROCESSLIST 指令观察所有当前执行的 SQL 状态。

```
mysql> SHOW PROCESSLIST WHERE COMMAND != 'Sleep';
+--------------+-----------+-------------------+------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------------
----------------------+-----------+--------------+-----------+
| ID           | USER      | DB                | COMMAND    | TIME    | STATE
| INFO                                                                                | ROWS_SENT | ROWS_EXAMINED |
ROWS_READ |
+--------------+-----------+-------------------+------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------------
----------------------+-----------+--------------+-----------+
| 0-0-352724126 | ifisibhk0 | test_123_wvvp_0000 | Query      |      15 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,SUM(`t_item`.`price`) from `t_i |
NULL |          NULL |      NULL |
| 0-0-352864311 | cowxhthg0 | NULL               | Binlog Dump |      13 | Master has sent all binlog to slave;
waiting for binlog to be updated | NULL                                                                                                 |
NULL |          NULL |      NULL |
| 0-0-402714566 | ifisibhk0 | test_123_wvvp_0005 | Query      |      14 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,`t_item`.`price` from `t_i      |
NULL |          NULL |      NULL |
| 0-0-402714795 | ifisibhk0 | test_123_wvvp_0005 | Alter      |     114 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */ALTER TABLE `Persons` ADD `Birthday` date                    |
NULL |          NULL |      NULL |
......
+--------------+-----------+-------------------+------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------------
----------------------+-----------+--------------+-----------+
12 rows in set (0.03 sec)
```

TIME 列代表该指令已经执行的秒数。发现过慢指令后，例如图中，可使用 **KILL '0-0-402714795'** 指令来取消慢指令。

注意： DRDS 中一个逻辑 SQL 对应多条分库指令，因此为了停止一个逻辑 DDL 可能需要 Kill 多条指令。从 SHOW PROCESSLIST 结果集的 INFO 列判断该指令归属的逻辑 SQL。

Lock conflict 报错处理

DRDS 执行 DDL 操作先会加库级锁，操作完后再释放掉。KILL DDL 操作很可能会导致该锁没有释放，此时再执行 DDL 会有以下报错：

```
Lock conflict , maybe last DDL is still running
```

此时执行 **RELEASE DBLOCK** 释放该锁即可。指令取消及锁释放后，选择业务低谷甚至停止期间，重新执行该 DDL。

其它问题

DMS 或其它客户端无法显示修改后的表结构

为了兼容某些客户端从系统表（如 COLUMNS 或 TABLES）中获取表结构的功能，DRDS 在用户0分库 RDS 里建立了一个影子库，影子库名与用户的 DRDS 逻辑库名一致。存储了所有用户库里的表结构等信息。

DMS 获取 DRDS 表结构是从影子库系统表中获取的。在处理异常 DDL 过程中，由于种种原因可能会出现用户库表结构正常修改，但是影子库中的表结构却未修改的现象。此时需要用户连接到影子库，在该库中重新对表进行一次 DDL 操作即可。

注意： CHECK TABLE 不会检测影子库表结构与用户库是否一致。

DRDS 支持高效的数据扫描方式，并支持在全表扫描时使用聚合函数进行统计汇总。

常见的扫描场景如下：

没有分库分表： DRDS 会把原 SQL 传递到后端 MySQL 执行。这种情况下 DRDS 支持任何聚合函数。

非全表扫描：SQL 经过 DRDS 路由后，发送到单个 MySQL 库上执行。比如说拆分键在 WHERE 中是等于关系时，就会出现非全表扫描。此时同样可以支持任何聚合函数。

全表扫描：目前支持的聚合函数有 COUNT、MAX、MIN、SUM。另外在全表扫描时同样支持 LIKE、ORDER BY 、LIMIT 以及 GROUP BY 语法。

并行的全表扫描：如果需要从所有库导出数据，可以通过 SHOW 指令查看表拓扑结构，针对分表并行处理。详见下文。

通过 HINT 来进行表遍历

执行 SHOW TOPOLOGY FROM TABLE_NAME 指令获取表拓扑结构。

```
mysql:> SHOW TOPOLOGY FROM DRDS_USERS;
+------+------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+------------------+--------------+
| 0 | DRDS_00_RDS | drds_users |
| 1 | DRDS_01_RDS | drds_users |
+------+------------------+--------------+
```

```
2 rows in set (0.06 sec)
```

非分库分表的表默认存储在第0个分库。

针对 TOPOLOGY 进行单表遍历。

- 第0个分库运行当前 SQL

/!TDDL:node='DRDS_00_RDS'*/ SELECT * FROM DRDS_USERS;

- 第1个分库运行当前 SQL

/!TDDL:node='DRDS_01_RDS'*/ SELECT * FROM DRDS_USERS;

注意：推荐每次扫描前执行 **SHOW TOPOLOGY FROM TABLE_NAME** 获取最新的表拓扑结构。

并行扫描

DRDS 支持 mysqldump 指令导出数据。但如果想要更快地扫描数据，可以针对每个分表开启多个会话的方式并行加速扫描。

```
mysql> SHOW TOPOLOGY FROM LJLTEST;
+------+----------------+------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+----------------+------------+
| 0 | TDDL5_00_GROUP | ljltest_00 |
| 1 | TDDL5_00_GROUP | ljltest_01 |
| 2 | TDDL5_00_GROUP | ljltest_02 |
| 3 | TDDL5_01_GROUP | ljltest_03 |
| 4 | TDDL5_01_GROUP | ljltest_04 |
| 5 | TDDL5_01_GROUP | ljltest_05 |
| 6 | TDDL5_02_GROUP | ljltest_06 |
| 7 | TDDL5_02_GROUP | ljltest_07 |
| 8 | TDDL5_02_GROUP | ljltest_08 |
| 9 | TDDL5_03_GROUP | ljltest_09 |
| 10 | TDDL5_03_GROUP | ljltest_10 |
| 11 | TDDL5_03_GROUP | ljltest_11 |
+------+----------------+------------+
12 rows in set (0.06 sec)
```

如上所示该表有四个分库，每个分库有三个分表。使用以下的 SQL 对 TDDL5_00_GROUP 库上的分表进行操作：

```
/!TDDL:node='TDDL5_00_GROUP'*/ select * from ljltest_00;
```

注意： HINT 中的 TDDL5_00_GROUP 与 SHOW TOPOLOGY 指令结果中的 GROUP_NAME 列相对应。另外 SQL 中的表名为分表名。

此时可开启最多12个会话（分别对应12张分表）并行处理数据。

DRDS 可以配合阿里云全局事务服务 GTS 来实现多个分库下的分布式事务。

DRDS 事务分类

- 在 DRDS 上，可以将事务分为两类：
  - 单机事务：所有的事务操作都落在同一个 RDS 分库；
  - 分布式事务：事务的操作涉及到多个 RDS 分库。
- DRDS 目前默认支持单机事务。若需要使用分布式事务，可以申请开通 DRDS 的跨库事务的功能。

开启 DRDS 分布式事务

前提条件

- 需要开启 GTS 分布式事务的 DRDS 实例必须是专享实例（共享 DRDS 实例暂不支持开通 GTS），并且创建至少一个 DRDS 数据库。
- DRDS 所属区域在 GTS 支持的范围内。目前 GTS 支持的区域包括华北2、华东1、华东2、华南1。

操作步骤

登录 DRDS 控制台，在左侧导航栏单击**数据库列表**。

在需要开启 GTS 的数据库所在行，向右滑动分布式事务服务列的滑块。

在弹出的确认对话框中，单击**确定**提交申请。

DRDS 后台会对申请进行审核。

- 审核通过，数据库列表中该数据库的分布式事务服务为开启状态。
- 审核未通过，请根据反馈的原因进行处理，如有需要请**重新申请**。

通过审核后，提交工单给 DRDS 团队，工单内容说明“**请重启 DRDS 实例**”。

重启完成后，该 DRDS 上的 GTS 服务生效。

配置 GTS 接入

在申请分布式事务成功后，就可以配置 GTS 接入了。配置 GTS 接入分为两种情况：

- 单个 DRDS 实例的 GTS 接入
- 多个 DRDS 实例的 GTS 接入

单个 DRDS 实例的 GTS 接入

对于单个 DRDS 实例的 GTS 接入，只需要在客户端代码里，在需要开始分布式事务的地方加上 select last_txc_xid() 语句即可开启分布式事务。之后的使用方式和传统单机数据库事务完全一致。

您既可以用手工的方式处理事务的提交和回滚，也可以将管理交给 Spring 事务管理器来管理。

手工处理事务的典型 SQL 语句步骤如下：

i. set autocommit=false //开启事务
ii. select last_txc_xid() //注册一个 GTS 事务
iii. insert/update/delete 等业务 SQL
iv. commit 或者 rollback //全局提交或回滚
v. set autocommit=true //恢复自动提交

使用 Spring 事务管理器来处理事务的典型代码如下：

```
@Transactional
public void update(JdbcTemplate jdbcTemplate) {
//开启一个 GTS 分布式事务
jdbcTemplate.execute("select last_txc_xid()");
// insert/update/delete 等业务 SQL
}
```

多个 DRDS 实例的 GTS 接入

如果需要将多个 DRDS 实例接入 GTS，则需要在第一个 DRDS 实例中加上 select last_txc_xid() 语句，再将其他 DRDS 实例和第一个实例的 txc-xid 进行关联。

下面以两个 DRDS 实例（datasource1 和 datasource2）的 GTS 接入举例说明。多个实例的接入方法类似。

1. 在 datasource1 上执行 select last_txc_xid ()， 接入 GTS，并得到 xid；
2. 在 datasource2 上执行 set txc_xid = xid，其中 xid 为 1 中获取的 xid。

以上两步就可以把 dataource1、datasource2 放到一个分布式事务里了。

完成上述 GTS 配置之后，DRDS 的多个分库中的数据会保持强一致，根据您业务代码的配置，要么同时提交，要么同时回滚。