

DDB 应用开发者白皮书 V1.1

By 马进

目录

一. 导语	2
1. DDB 架构简介	2
2. 技术支持与反馈	5
二. DDB SQL 语法规则	6
1. SELECT	7
2. INSERT	14
3. REPLACE	17
4. UPDATE	18
5. DELETE	18
6. 存储过程	19
三. Hint 语法说明	20
1. DIRECT FORWARD	20
2. LOAD BALANCE	21
四. DDB 命令行	22
1. QS 命令行	22
2. DDB 命令行	25
3. 实时统计	28
五. 连接池范例	31
六. 规范与边界	32

一. 导语

网易分布式数据库 DDB 作为杭研最老的产品之一，几乎伴随了网易所有大型互联网产品的成长，8 年间帮助众多产品和用户解决了大数据，分库分表的问题。本书目的在于帮助这些产品和用户更快更方便地熟悉并适应基于 DDB 的应用开发。

本书一些内容提取自 DDB 用户手册，并加以整理。由于手册内容过于详实，不太适合初级用户阅读，本书则立足于应用开发者关心的问题，在简要介绍 DDB 架构基础上，重点记录了 DDB 的 SQL 语法规则（支持什么，不支持什么），便于开发者随时查阅。

本书使用的 DDB 版本为 V4.6 及以上，老版本 DDB 在 SQL 语法上有若干 BUG 和限制，对使用旧版本的开发者，在阅读本书之外，需要对使用的 SQL 进行充分测试。

1. DDB 架构简介

本节仅对应用开发端需要关注的 DDB 架构进行阐述，如下所示：

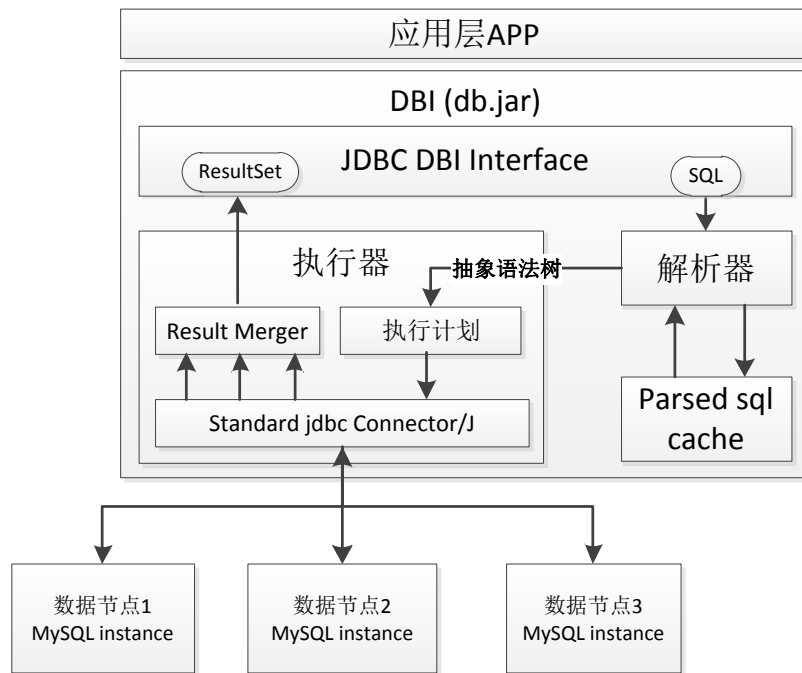


图 1 DBI 模式架构图

应用 APP 通过 DDB 提供的 JDBC 驱动 DBI (Database Interface) 来访问 DDB，就像 MySQL 应用需要通过官方提供的 JDBC 驱动 Connection/J 访问 MySQL 一样。

DBI 与标准 Connection/J 不一样的地方在于，Connection/J 的功能仅是按照 MySQL 通信协议将应用 SQL 提交给 MySQL Server 执行，SQL 的解析和执行均在 MySQL Server 内部完成。而 DBI 的工作则要复杂许多，如上图所示，首先它接收应用方 SQL，经过语法解析生成一个抽象语法树，再根据抽象语法树生成分布式执行计划，由 DBI 的执行器按照分布式执行计划生成下发给每个数据节点（单个 MySQL Server）的 SQL，并将数据节点返回的结果合并后返回给应用 APP，如有 SQL：

```
select * from Blog order by id;
```

DBI 接收到该 SQL 后，首先会通过语法解析生成一个 SELECT 语法树，DBI 执行器根据这颗 SELECT 语法树生成分布式执行计划：将 select 语句下发给每个数据节点，在执行器获得所有数据节点的结果集后，做一次归并排序，将最终结果集返回给应用 App。

由于 DBI 中的语法解析，生成执行计划以及执行器执行都在应用端，会占用应用端一部分处理器和内存资源，一般情况下，SQL 越复杂，DBI 的负载比重会越大，另外，合理构建 SQL 语句可以有效控制 DBI 的代价，具体请参照本书第二章高效 SELECT 语句这一节。

使用 DDB 提供的 JDBC 驱动 DBI 访问 DDB 存在一定的缺陷：

- ◆ 仅支持 JAVA 语言
- ◆ 使用 DBI 需要应用端部署 DBI 的 jar 包：db.jar，netease-commons.jar，common.jar 等，DDB 升级需要与应用沟通重启停服，给运维和升级工作带来困难，这也直接导致了目前 DDB 产品线从 3.3-4.5 都有，给技术支持造成了很大的不便。
- ◆ 由于 DBI 部署在应用端，每次应用重启都会伴随着 DBI 模块的重启，由于 DBI 内部维护了一些文件锁，瞬间重启可能由于操作系统的文件锁没来得及释放而抛异常，总而言之，DBI 比标准 JDBC 复杂很多，它与应用端一起运维会提升应用运维的复杂度。

为了弥补 DBI 模式的各种缺陷，我们开发了一个全新的 QS 使用模式，如下图所示：

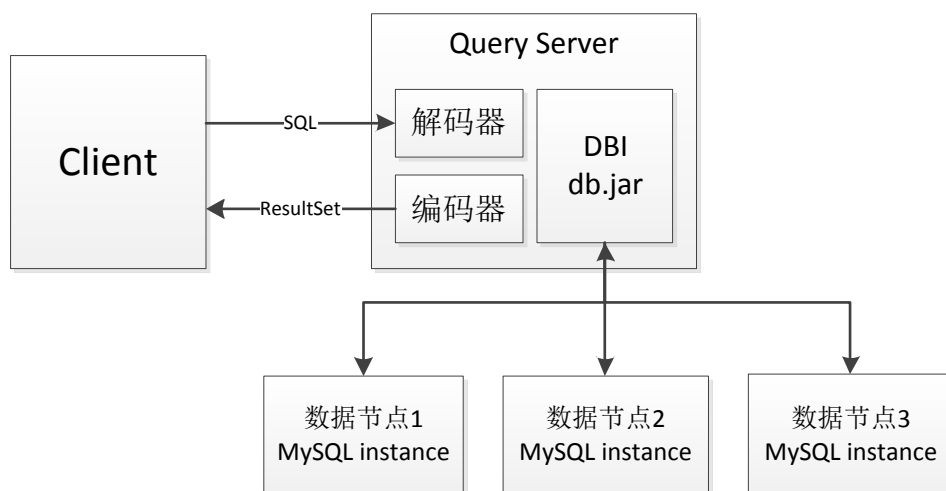


图 2 QS 模式架构图

在 DDB 的这种使用模式中，首先要为应用部署一个或多个查询服务器，查询服务器负责 MySQL 标准 JDBC 驱动到 DDB 的 JDBC 驱动的转换，应用通过标准 JDBC Connection/J 访问查询服务器 QS，QS 遵照 MySQL 通信协议对应用请求进行解码，并调用 DBI 的相关方法完成 SQL 请求，最后将结果集按照 MySQL 通信协议编码返回给应用端。

使用 QS 模式可以解决 DBI 模式的以上缺陷，因为 DBI 驱动模块位于 QS 中，应用重启与 DBI 模块再无关联，而 DDB 的运维升级也无需应用感知。通过 HAProxy 的分流机制，还可以做到不停服升级。另外，QS 与应用通过 MySQL 通信协议交互，真正做到了多语言支持。除此之外，QS 还具备以下几点优势：

- ◆ 在现有服务满足不了应用规模时，可以通过 HAProxy 做到水平线性扩展
- ◆ 可以通过 QS 的运维命令（show processlist 等）查看 DBI 驱动的各种状态，以及连接数等信息，便于运维
- ◆ 通过标准 MySQL JDBC 或 MySQL 命令行连入 DDB，能够为应用提供更好的移植性和更小的侵入
- ◆ 可以通过 QS 做一些 SQL 层面的统计，如 SQL 的慢语句模式，落入数据节点的分布情况等

QS 模式的代价在于需要申请额外的 QS 服务器，以及应用请求多次在网络中过度带来的风险，即便如此，QS 模式的使用依旧能够保持 DDB 整体服务的 SLA。因此强烈推荐用户使用 QS 模式。

另外, 目前云计算 DDB 日渐完善, QS 模式的使用可以帮助应用无缝接入云计算 DDB, 以实现 DDB 的一键部署和更好地控制成本。QS 的更多情况可以向 DDB 开发组寻求技术支持。

2. 技术支持与反馈

网易公司内部适用

DDB 用户交流群: 1132484

DDB DOC 平台: <http://doc.hz.netease.com/display/ddb/Home>

DDB FAQ: <http://doc.hz.netease.com/display/ddb/FAQ>

DDB 需求收集: <http://doc.hz.netease.com/display/ddb/Requirements>

二. DDB SQL 语法规则

分布式数据库支持常用的 SQL DML 语句的基本功能。分布式数据库支持的 SQL 语句简单说明如下：

- ◆ 支持单表或多表联接查询
- ◆ 支持单个记录 INSERT、REPLACE 语句
- ◆ 支持 UPDATE、DELETE 语句
- ◆ 支持根据单个或多个属性/表达式进行 GROUP BY
- ◆ 支持简单和复杂的 HAVING 条件
- ◆ 支持简单和复杂的 ORDER BY
- ◆ 支持 MIN、MAX、COUNT、SUM、AVG 这几个聚集函数
- ◆ 支持 DISTINCT，并可与聚集函数联合使用
- ◆ 支持 LIMIT/OFFSET，其值可以是整数也可以是算术表达式
- ◆ 支持 FOR UPDATE、LOCK IN SHARE MODE
- ◆ 支持 FORCE INDEX
- ◆ 支持多值插入语句 INSERT INTO ... VALUES (row1)(row2)
- ◆ 支持 SELECT 中使用别名
- ◆ 暂不支持子查询（V4.8 及以上支持）
- ◆ 暂不支持 union 和 union all（V4.8 及以上支持）
- ◆ 分布式数据库的 SQL 支持功能主要限制说明如下：
 - ◆ 在支持多表连接查询时，连接的顺序是按照 SQL 语句中表的出现顺序进行，因此 SQL 语句中相邻的表之间需要有直接或者间接的 JOIN 条件。
- ◆ 不支持联表 UPDATE 和 DELETE

DDB 支持的 SQL 语法面向 MySQL 数据库引擎（是 mysql sql 语法的子集），对其他数据库如 Oracle，在 DDB4.0 版本之前做过支持，但由于长时间未有产品使用，在 DDB4.0 版本之后不再支持。

本书描述的 SQL 语法规则适用于 DDB 最新稳定版 4.6，下面分章节对各个 SQL 语句语法进行详细说明。

1. SELECT

分布式数据库支持单表和多表联接查询，支持 SQL 标准中定义的 GROUP BY/HAVING、ORDER BY、LIMIT/OFFSET 等子句的基本功能，另外还支持 MySQL 提供的 FOR UPDATE/LOCK IN SHARE MODE 特殊功能。详细的语法如下：

```
SELECT select_expr [FROM table_references] [WHERE where_condition]
[GROUP BY col_name, ...] [HAVING where_definition]
[ORDER BY col_name [ASC|DESC], ...]
[LIMIT row_count [OFFSET offset]]
[FOR UPDATE | LOCK IN SHARE MODE]
```

分布式数据库提供的 SELECT 语句功能及限制详细说明如下。

1.1 查询条件

分布式数据库支持以下查询条件：

- ◆ 算术运算符：+、-、*、/、%
- ◆ 比较运算符：<、<=、=、>=、>、<>、LIKE、BETWEEN
- ◆ 逻辑操作符：AND、OR、NOT
- ◆ 列表 IN/NOT IN 操作：形如“id IN (1,2,3)”，注：4.8 版本以后支持 in 子查询
- ◆ NOT LIKE 操作
- ◆ NOT BETWEEN 操作
- ◆ 正则表达式运算：RLIKE、REGEXP
- ◆ 支持位运算^、|、&等

1.2 函数查询

DDB 支持基本的聚集函数和标量函数查询。

支持的聚集函数包括 MIN、MAX、COUNT、SUM 和 AVG。

与 mysql 相比，DBI 聚集函数的查询结果在数据精度上略有不同：无论聚集字段为何种类型，COUNT 的精度总是 JVM 中的 LONG，在 MySQL 中为 BigInt，SUM 和 AVG 的精度总是 JVM 中的 BigDecimal，由于 MySQL 中没有 BigDecimal 对应的精度，所以

DBI 返回的 SUM 和 AVG 一般与 MySQL 返回的结果总是数值相等，精度不等。在对 DBI 聚集函数结果进行字符串处理时需要注意这一点。

DDB 理论上支持 mysql 所有的标量函数，如 abs, length 等，因为对标量函数，DBI 无需多余的处理，只需要将函数直接下发给 mysql 获取结果集即可。

1.3 分组与排序

分布式数据库支持最基本的分组和排序功能，详细说明如下：

- ◆ 支持对多个列进行分组(group by)和排序(order by)
- ◆ 支持按复杂表达式进行分组和排序，但此时分组或排序表达式**必须出现在 SELECT 字段列表中**
- ◆ 支持对聚集函数和标量函数进行排序，但对应的聚集函数或表达式**必须出现在 SELECT 字段列表中**
- ◆ 聚集函数和标量函数不能应用在分组操作上

如以下的分组与排序语句是支持的：

```
select sum(score) from score group by id;
select id, score from score order by score desc;
-- 联接语句也支持分组和排序
select name, sum(score) from users, score where users.id = score.id and
users.id > 0 group by users.name;
select name, score from users, score where users.id = score.id and users.id >
0 order by score desc;
-- 按复杂表达式进行分组和排序
select 100 - score from score order by 100 - score;
select id / 2, sum(score) from score group by id / 2;
-- 按聚集函数排序
select id, max(score) from score group by score order by max(score);
select *, max(score) from score group by score order by max(score);
```

但如下的分组和排序语句不被支持：

```
-- 按复杂表达式排序但该表达式不在投影列表中
select score from score order by 100 - score;
-- 参加排序的聚集函数不在 select 列表中
select id from score group by score order by max(score);
select * from score group by score order by max(score);
```

在构建分组和排序的 SQL 语句时，对出现在 group by 和 order by 后面的聚集函数，标量函数以及表达式，必须同时出现在 SELECT 子句的列表中，否则会在生成执行计划

时抛异常，做这样的限制是为了减少 DBI 驱动不必要的投影开销，使用时需要留意。

1.4 多表连接

表联接主要功能及其限制说明如下：

- ◆ 支持多表线性联接
- ◆ 联接条件若包含多个子条件，则必须为 AND 关系，DDB 不支持笛卡尔连接，必须包含至少一个连接条件
- ◆ 联接条件可以是等值条件，不等值条件，大于条件或者小于条件
- ◆ DDB 的连接遵循 FROM 子句中表出现的顺序，由于 DDB 不支持笛卡尔连接，要求对于任意 FROM 子句中相邻出现的表（相邻即 join），在 WHERE 条件中必须**直接或者间接**出现它们的连接条件。所谓间接连接条件，是指能够通过等值推导间接推出两表的连接条件

如以下的联接语句是支持的：

```
select * from users, score where users.id = score.id and users.name = 'aaa';
-- 联接条件与外表条件间的顺序可以互换
select * from users, score where users.name = 'aaa' and score.score > 80 and users.id = score.id;
-- 联接外表没有指定选择条件
select * from users, score where users.id = score.id;
-- 多个联接条件
select * from users, score where users.id = score.id and users.name = score.subject and users.name = 'aaa';
-- 联接条件为不等值条件
select * from users, score where users.id <> score.id and users.name = 'aaa';
-- 多表联接
Select * from users, score, class where user.id = score.id and score.cid = class.id and class.name = 'No.1';
-- 多表联接,虽然相邻的 users 和 score 表没有直接的 join 条件,但是可以根据 score.id=class.id 和 user.id=class.id 推导出 user.id=score.id 的 join 条件
Select * from users, score, class where user.id = class.id and score.id = class.id and class.name = 'No.1';
```

但如下的联接语句不被支持：

```
-- 多个联接条件不能为 or 关系
select * from users, score where users.id > score.id or users.id < score.id and users.name = 'aaa';
-- 联接条件包含算术表达式
select * from users, score where users.id + 1 = score.id and
```

```
users.name = 'aaa';  
-- 相邻的 users 和 score 两表没有直接或间接的连接条件  
Select * from users, score, class where user.id = class.id and  
score.name = class.name and class.name = 'No.1';
```

1.5 DISTINCT

分布式数据库支持 DISTINCT，并可与聚集函数联合使用。DDB 中 distinct 的使用分以下两种情况：

1. 当 distinct 没有出现在聚集函数中，那么 distinct 只能出现在 select 关键字后面，表示对整个 select 列表做 distinct，返回记录列表中任意字段的值存在差异都将返回该记录，而不是对列表中某个字段单独做 distinct。distinct 全字段的实现方法是将所有列转化为 group by 列表来执行去重（除非表中有完全相同的两行，否则全字段上的 distinct 没有意义，完全相同的两行会被 group by 为一行）。
2. 当 distinct 出现在聚集函数中时，DBI 会把 distinct 的字段加到 group by 末尾下发到 mysql 中，并在 DBI 内部再做一次去重处理，通过 mysql 的 group by 操作可以很好地减少 DBI 内部去重的量。由于 DBI 的这种 distinct 转化为 group by 下发的优化，目前 DBI 不支持多个聚合函数中包含 distinct。

另外 DDB 不支持 distinct 既出现在聚合函数内，也出现在聚合函数外。

如以下语句是支持的：

```
select distinct id from score;  
select distinct id, score from score;  
select id, count(distinct score) from score;
```

但下面的语句不被支持：

```
select id, distinct score from score;  
select id, count(distinct score), sum(distinct score) from score;
```

1.6 LIMIT/OFFSET

DDB 支持 LIMIT/OFFSET，值可以是整数也可以是算术表达式，如：

```
select * from score order by score desc limit 1;  
select * from score order by score desc limit 1 offset 1;  
select * from score order by score desc limit 1+1 offset 2*3;
```

从 DDB4.2 起，开始支持 LIMIT m,n 的 SQL 语法，m 相当于 offset 值，n 相当于 limit 值：

```
select * from score order by score desc limit 1,2;
```

DDB 不推荐应用开发者使用过大的 offset，原因见高性能 SELECT 语句一节。

1.7 FORCE INDEX

DDB 支持 MySQL 特有的 FORCE INDEX 功能，如：

```
select * from users force index (name) where users.name = 'aaa';  
-- 联接语句中也支持 FORCE INDEX  
select * from users force index (name), score where users.name = 'aaa' and  
users.id = score.id
```

1.8 SQL_CACHE 和 SQL_NO_CACHE

SQL_CACHE 和 SQL_NO_CACHE 关键字用于指定语句在执行之后，结果集是否在 MySQL 上缓存，对于一张表中的数据不经常被修改，而查询操作又非常频繁的情况下，缓存结果集能够极大提升数据库性能及响应时间。

对于最终 MySQL 数据库上是否缓存了查询结果集，还跟 query_cache_type 参数有关，下表说明了两者的关系：

query_cache_type	说明
0/OFF	不缓存任何结果集
1/ON（默认）	缓存所有的查询，除了那些明确指定 SQL_NO_CACHE 的语句
2/DEMAND	只缓存那些指定了 SQL_CACHE 参数的语句

具体 SQL 语句语法如下：

```
select [distinct | SQL_CACHE | SQL_NO_CACHE] *|column1|column2 ... from  
table_name ...
```

1.9 高性能 SELECT 语句

DDB 一直以来的设计宗旨，都是希望帮助应用开发者像使用普通数据库一样使用 DDB。因此 DDB 的实现非常强调“透明性”，然而透明的弊端，就是可能会导致开发者在不经意间写出一些运行起来非常低效的 SQL，这种现象尤其反映在 SELECT 语句中（因为 DDB 支持的 SELECT 语句较为复杂）。这里总结一些构建高效 SELECT 语句的技巧，若开发者或 DBA 发现某个 SELECT 语句运行慢地不符合预期时，请参照本节内容。

a) WHERE 条件中尽量带入均衡字段的判等条件

DDB4.5.7 版本中引入了单节点查询优化，当 WHERE 条件中包含均衡字段的判等条件时，该 SQL 有很大可能落在一个数据节点上，而进入单节点查询优化路径的 SQL 因为省去了 DBI 的各种处理，比常规 SELECT 吞吐率和响应时间能够提升 1.5-5 倍不等。

b) 相互 JOIN 的表尽量使用同一个均衡策略，并在均衡字段上连接

对常规联接 SELECT，DDB 会先从第一张表中 select 出符合条件的所有数据，再一行行带入下一张表的 select 中，以此类推。在这种实现下，若两表连接，第一张表 select 出 1000 行数据，则会产生 1000 条 SQL 带入第二张表，以获取最终结果集。因此常规联表操作在第一张表的结果集较大时是非常耗时的。

若相互 JOIN 的表都使用同一个均衡策略，且在均衡字段上连接，如表 A,B 都是用均衡策略 user，A 的均衡字段为 a，B 的均衡字段为 b，构建 SQL 语句如：

```
select * from A, B where A.a = B.b and A.kkk = 'ok'
```

由于同一种均衡策略下数据分布相同，而且又都在均衡字段上连接，则不会产生任何跨数据节点的联接操作，对这种 SQL，DDB 会把连接下发给数据节点，直接获取最终结果集。

非均衡字段上的联接会给客户端带来比较大的开销，且非常耗时，线上 SELECT 最好都使用均衡字段上的联接，对没办法在均衡字段上联接的 SELECT，建议构建以联接字段作为均衡字段的冗余表（保守估计差距在 2-10 倍）。

对线下 SQL，在构造联接 SQL 时应当将小表放在 FROM 子句前列。

c) group by 子句中尽量包含均衡字段

DDB 中，不同数据节点上不会出现相同的均衡字段值，利用这一特性，当 SELECT 语句中出现 group by 包含均衡字段时，group by 以及基于 group by 的聚合函数都会直接下发给数据节点中做分组和聚合，而无需在 DBI 端再做 Merge。

group by 包含均衡字段可以极大节省 DBI 端开销，推荐应用开发者和 DBA 在为表选择均衡字段时充分考虑。

d) 聚合函数+distinct 尽量做在均衡字段上

与 group by 尽量包含均衡字段的原则相同，聚合函数+distinct 均衡字段也可以直接下发给数据节点，无需 DBI 再做 Merge，如表 A 均衡字段为 a:

```
select count(distinct a) from A
```

对上述 SELECT，DBI 会将 count(distinct a)直接下发，并将所有数据节点的结果集依次返回。

e) 尽量不要使用过大的 offset

DDB 作为分布式数据库中间件，不推荐用户使用较大的 offset，因为任何分布式数据中间件对 limit n offset m 的常规做法，都是转化为 limit n+m 下发给数据节点，再在中间件层做排序筛选，因此 offset 的大小会直接反映到数据节点中的 limit 大小。当 offset 过大时会严重影响 SQL 执行效率，甚至导致 OOM。

对应用中的“翻页”业务，建议用户通过记录上一页尾的排序值，并将其作为下一页的 WHERE 子句条件带入 SELECT 来达到与 offset 同样的效果。

f) order by 聚合函数会对客户端 CPU 产生较大开销

DDB 中，order by 一般字段，会先将 order by 下发给数据节点，DBI 在获取了所有数据节点的结果集后再做一次归并排序，这种归并排序的开销较小。

若 order by 聚合函数，因为数据节点没有全局聚合结果，本地排序没有意义，只有在 DBI 上做全局的堆排序，才能保障排序的正确性。这种情况下的堆排序开销较大。

是否需要在聚合上排序取决于应用逻辑，应用开发者在 SQL 选型时需要予以考虑。

g) 对过大结果集，通过设置流方式保障 QS 不会内存溢出

由于 DDB 一般包含多个数据节点，相比普通数据库而言，更容易出现过大结果集导致内存溢出的问题。为此 DDB 通过标准 JDBC 提供了流方式结果集机制(**stream resultset**)来避免 QS 节点 OOM，使用方法，是在执行 **select sql** 之前先执行一条指令：

```
set useStreamResult = true;
```

上述 SQL 会将连接中的 **useStreamResult** 这项环境变量设置为 **true**，之后的查询中 QS 会启用 JDBC 的 **Stream result** 机制，可以有效防止内存溢出，需要注意的是，使用这个机制是有风险的：在网络不稳定或出现后端数据库瞬断的情况下，可能造成连接池中部分物理连接完全不可用，这种情况下需要重启才能解决（源于 JDBC 的 BUG）。所以这个机制要慎用，最好在应用层面杜绝可能会引发 QS OOM 的 SQL。OLTP 应用的 SQL 场景一般是不会引发 OOM 的，需要注意的是部分使用 QS 做 OLAP 的应用场景。具体的系统边界请参考本书最优一章“系统边界”

2. INSERT

DDB 支持 INSERT 语句，语法为：

```
INSERT [IGNORE] INTO table_name [(col_name, ...)] VALUES (expr|DEFAULT|seq,...)
[ON DUPLICATE KEY UPDATE col_name = expr, ... ]
```

DDB 不支持以下复杂的 INSERT 语句：

- ◆ 不支持 INSERT INTO ... SET ...语句
- ◆ 不支持 INSERT INTO ... SELECT 语句
- ◆ 支持 INSERT IGNORE 语句，前提是表均衡字段必须是主键或主键中的一个字段，否则该表无法保障数据在主键上全局唯一，进而无法保障 ignore 语义
- ◆ 当 SQL 语句中包含 ON DUPLICATE KEY UPDATE 时，update 的列不能是均衡字段。因为均衡字段的更新会带来数据迁移，会为 DDB 的运维带来风险

当数据库表上建立有唯一索引时，如果向其中插入相同的数据，会返回用户提示插入失败。这个问题在 DDB 上需要分情况讨论：

1. 如果唯一性索引就是均衡字段，则情况比较简单，DDB 依赖于底层数据库节点来保证数据唯一性。

2. 如果唯一性索引不是均衡字段，则 DDB 只能保证同一个节点上数据唯一，跨节点是无法保证的。如果用户需要这个限制，可以打开数据表上的插入数据唯一性检查选项。打开此选项后 DDB 会在每次插入前在所有节点上进行查询，以此来保证数据唯一。当节点数量较多时，性能下降比较厉害，因此默认 DDB 不开启这个选项。
3. 在唯一性索引不是均衡字段的情况下，还有一种方法保障字段全局唯一：为每个唯一性字段建一张冗余表，并将冗余表的均衡字段设为唯一性字段，在 insert 这张表的数据时，需要用一个分布式事务保证数据插入所有的冗余表中，例如有表 A 中有非均衡字段的唯一性索引字段 a,b,c，为表 A 建冗余表 A1, A2, A3，其中 A1 仅包含均衡字段 a，A2 仅包含均衡字段 b，A3 仅包含均衡字段 c，在向表 A 中插入数据时，做如下事务：

```
begin;
insert into A(a, b, c) values(1, 2, 3);
insert into A1(a) values(1);
insert into A2(b) values(2);
insert into A3(c) values(3);
commit;
```

通过冗余表的均衡字段全局唯一来保障特定数据列的全局唯一性，这种应用模式下，冗余表相当于一个二级索引。而且由于每个 insert 都是按均衡字段进行插入，其性能要优于先 select 后 insert 的方案，目前众多大产品，如邮件，易信都采用了这种方案，这种方案的劣势在于需要应用配合构建冗余表。

2.1 全局自增 ID 分配

DDB 与 MySQL 一样支持在建表时指定单个自增长字段（每个表至多一个），与 mysql 不同的是，DDB 的自增长字段需要定义在建表语句后的 hint 中，如下斜体所示：

```
isql@dba>> show create table inctest;
CREATE TABLE `inctest` (
  `id` bigint(20) NOT NULL,
  `uid` bigint(20) NOT NULL DEFAULT '0',
  KEY `uid` (`uid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 /* BF=id, POLICY=user, STARTID=13001,
AUTO_INCREMENT_COLUMN=id, ASSIGNIDTYPE=MSB */;
```

ASSIGNIDTYPE 标注了全局自增长 ID 的类型，目前可选有 USB（ID 信息存在数据库

中批量分配) 类型和 TSB (基于时间戳的 ID 分配) 类型, 使用 USB 的话, QS 每次分配完固定个数的自增长 ID 后会向系统库申请新一批 ID, 默认一批 ID 为 1000 个, 因为申请方式为批量申请, 难以保证 ID 的自增性, TSB 是根据时间戳信息生成自增 ID, 自增性较 USB 较好, 但是字段很长。

有两种方法通过 QS 获取自增长 ID:

1. 可以使用 `select allocate_record_id from table` 或 `select allocate_record_ids from table` 语句作为 SQL 向 QS 发起 query 请求, 前者只获取一个自增字段值, 后者获取一批, 批量获取的数量是个随机数(一般<1000), 当获取的自增字段数量不够时, 可以多次调用 `allocate_record_ids` 直到获取到想要的数量为止。

```
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection(url, user, pass);
Statement stmt = conn.createStatement();
ResultSet res = stmt.executeQuery("select allocate_record_id from table1");
long id = 0;
If (res.next())
    id = res.getLong(1);
```

2. 也可以直接向 QS 插入自增长字段值, 此时自增长字段设为 `seq` 或不写, 之后通过 jdbc 的 `getGenerateKeys()` 获取插入的自增长字段值, 这点和 DBI 的用法一致, 需要特别注意的是, 当插入批量数据时, `getGenerateKeys()` 得到的批量自增长 ID 是不准的, 若有获取批量自增长 ID 需求, 必须使用第一种方式。

总体来说, DDB 在插入自增长字段时支持和 MySQL 一样的写法: 在不指定自增长字段时, DDB 会为语句自动添值, 但是如果自增长字段是表的均衡字段, 则必须指定自增长字段为 `seq`, 否则会抛异常。

2.2 批量插入数据

DDB 支持批量 INSERT 数据, 语法: 如“INSERT ... VALUES (row1),(row2)”

在实际执行时, 会先根据语句中的均衡字段值对整条语句进行拆分, 按照目标数据库节点进行归类, 每个目标节点一条语句。因此, 如果用户输入了以下 SQL 语句:

```
"INSERT ... VALUES (row1),(row2),(row3),(row4)"
```


则实际执行时有可能被拆分为：

```
DBN1: "INSERT ... VALUES (row1), (row2)"
DBN2: "INSERT ... VALUES (row3), (row4)"
```

同时，DDB 在底层节点执行时，会启动两阶段分布式事务，来保证整个事务的原子性。

对查询服务器用户，可以通过 `set defaulttxa=0/1` 命令设置连接是否开启两阶段事务，若设置 `defaulttxa=0`，则默认采用普通的一阶段事务进行插入操作，在这种场景下不能严格保证多个数据节点上的数据一致性。一阶段事务与 XA 事务相比的优势在于能够提供非常可观的吞吐率。

3. REPLACE

DDB 支持 REPLACE 语句，语法为：

```
REPLACE INTO table_name [(col_name, ...)] VALUES (expr|DEFAULT|seq,...)
```

DDB 支持的 REPLACE 语法特性与 MySQL 一致：当插入的数据与表已有数据在唯一索引或主键索引上有冲突时，REPLACE 会将已有冲突行更新为 REPLACE 的数据行。另外对 DDB 而言，行冲突的前提是行的均衡字段落在同一个 DBN 上。

由于 REPLACE 操作依赖全局唯一性索引，与 INSERT IGNORE 一样，只有在表均衡字段是主键或主键中的一个字段时，DDB 才能保障 REPLACE 的语义完整性。

DDB 的 REPLACE 同样支持全局自增 ID 分配和批量插入数据，在返回结果上，DDB REPLACE 返回的修改行个数=更新行数*2 + 插入行个数。示例如下：

```
isql@dba>> replace into usertest2 values(6,'zxr',1112),(111,'kkk',999);
3 rows modified, execute time: 138 ms
-- (6,'zxr',1112)为更新, (111,'kkk',999)为插入, 返回修改行数为3

isql@dba>> show create table inctest;
CREATE TABLE `inctest` (
  `id` bigint(20) NOT NULL,
  `uid` bigint(20) NOT NULL DEFAULT '0',
  KEY `uid` (`uid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 /* BF=id, POLICY=user, STARTID=13001,
AUTO_INCREMENT_COLUMN=id, ASSIGNIDTYPE=MSB */;
```

```
isql@dba>> replace into inctest(uid) values(99),(100),(101),(102);
4 rows modified, execute time: 209 ms
Generated keys
+-----+
| id    |
+-----+
| 13006 |
| 13007 |
| 13008 |
| 13009 |
+-----+
-- 这种用法结果与 insert 一样
```

4. UPDATE

分布式数据库支持的 UPDATE 语句语法为:

```
UPDATE table_name SET col_name1 = expr [col_name2 = expr, ...] WHERE
where_condition [LIMIT row_count [OFFSET offset]];
```

DDB 对 UPDATE 语句限制如下:

◆ 不支持 UPDATE 均衡字段

与 insert on duplicate key update 的限制相同, 更新均衡字段会为 DDB 系统引入风险, 因为一个普通的 UPDATE 语句就可能造成数万行的数据迁移, 而且还要在一个事务中完成, 无疑会为系统带来极大开销。

为避免业务遇到更新均衡字段的需求, 建议用户在创建表模式时选择一个在业务上无需更新并且具有较均匀的分布特性的字段作为均衡字段。如全局自增 ID。

对确实需要更新均衡字段的情况, 目前需要用户用一个 delete+insert 的事务来实现更新均衡字段的业务。

◆ 不支持联表 UPDATE 语句

DDB 虽然支持联表查询, 但不支持联表 UPDATE。这个功能考虑在 4.5.7 以后的版本中支持。直观地说, DDB 目前不支持 UPDATE 子句中包含多个表。

5. DELETE

分布式数据库支持的 DELETE 语句语法为:

```
DELETE FROM table_name WHERE where_condition [LIMIT row_count [OFFSET offset]];
```

DDB 对 DELETE 语句限制如下：

- ◆ 不支持联表 DELETE 语句，与 UPDATE 语句限制一样

6. 存储过程

DDB 使用 CALL 关键字来调用存储过程。语法如下：

```
call procedure_name(param1, param2.....)
```

按以上方法调用存储过程时，将在 DDB 的所有数据库节点上调用该存储过程。如果用户想只在某几个个别节点上执行，可以使用 **direct forward** 来实现：

```
/* FORWARDBY (TABLENAME=T1, T2; BFVALUE=1,2) */call procedure_name(param1, param2.....)
```

目前 DDB 对存储过程的支持还比较简单，还不支持 JDBC 接口中的 CallableStatement。

三. Hint 语法说明

用户可以通过 **hint** 来指定语句的具体执行方式，以提高性能。**hint** 可以混合使用，通过逗号间隔，例如 `/* hint1, hint2*/ select * from test_tbl`。

1. DIRECT FORWARD

当用户确定一条 **sql** 语句可以直接发往 **mysql** 而获得正确的结果时，可以不经中间件处理直接发往节点。

使用方法:

在语句前加上

```
/*FORWARDDBY (TABLENAME=TABLE1, TABLE2, ... ; BFVALUE=value) */
```

其中 **TABLENAME=TABLE1, TABLE2..**是这条查询语句所涉及到的表，用来检查语句涉及的表是否属于相同的均衡策略，**BFVALUE=value** 是语句使用的均衡字段值，用于进行选择执行的 **DBN** 节点，**BFVALUE** 参数是可选的。例如:

```
/* FORWARDDBY (TABLENAME=T1, T2)*/ select * from T1, T2 where T1.ID = T2.ID and T1.Score > 90
/* FORWARDDBY (TABLENAME=T1, T2; BFVALUE=1,2)*/ select * from T1, T2 where T1.ID = T2.ID and (T1.ID=1 or T1.ID=2) and T1.Score > 90
```

在 **Isql** 中使用 **FORWARD BY**，如果 **hint** 中有分号，则需要通过 [DELIMITER](#) 命令修改命令结束符。

目前 **FORWARD BY** 只能用于下发一个节点的场景，如果 **BFVALUE** 指定了多个均衡字段的值，或者不指定均衡字段值而获取到多个 **DBN** 节点，那么只会选择第一个节点下发。对于希望每个节点都执行的场景，可以通过 **show dbns for table tablename** 获取 **DBN** 节点个数，再通过连续的一组均衡字段值达到依次下发的效果。

使用 **FORWARD BY** 时，语句必须满足以下限制：

- 多表查询时涉及的表属于相同的均衡策略

Direct forward 这个功能，好处是能够绕开 **DDB** 中间件的一些限制。例如，目前 **DDB**

是不支持嵌套查询语句的，如果用户确实有嵌套查询语句的需求，而且又能够保证在单台节点上运行返回的结果是正确的，那就可以使用 **Direct forward**。

2. LOAD BALANCE

MySQL 本身提供了复制机制（**replication**），可以自动把一个 **mysql** 数据库上的内容复制到另一台机器的 **mysql** 镜像节点上。基于 **mysql** 的复制机制，我们可以把一些对时间精度要求不高的只读操作分流到镜像机器上，实现 **DDB** 负载均衡，提高分布式数据库可扩展性。

实现负载均衡的 **SQL** 查询语法如下：

```
/*LOADBALANCE (TYPE=hint,delay=hint)*/ select .....
```

其中，**type** 为负载均衡类型，目前支持的类型为：

- ◆ 只能在 **Slave** 上执行，**TYPE=slaveonly**（**Slave** 不可用则失败）
- ◆ 优先在 **Slave** 上执行，**TYPE=slaveprefer**（先在 **Slave** 上执行，**Slave** 不可用在 **Master** 上执行）
- ◆ 根据均衡策略执行，**TYPE=loadbalance**（根据均衡策略选在 **Slave** 或 **Master** 上执行）
- ◆ 只能在 **Master** 上执行，**SQL** 中不指定 **LOADBALANCE** 语句或 **TYPE= masteronly**（对实时性要求高的语句）

其中，**delay** 为可选项，表示用户指定的 **slave** 节点最大延时时间限制。若不指定延时时间，则只要 **slave** 节点有效（可连接并且复制正常）即可提供服务。

四. DDB 命令行

DDB 在 QS 模式下,支持通过 MySQL 命令行访问 DDB,并支持基本的 `show processlist`, `kill`, `kill query` 命令,除此之外,DDB 自身设计了一些 `show` 命令来查看 DDB 的一些内部状态。QS 一般用于 OLTP 应用,为了保证相应时间和服务质量,对 QS 的内存做了限制,因此不建议直接在 QS 上直接做大规模的数据导入导出,一来 DDB 的数据导入因为要计算 hash 值,比较占 CPU,容易对线上服务造成影响,二来数据导出时有 OOM 的风险。为了解决这个问题,我们一方面为 QS 开发了流方式模式避免内存溢出,一方面提供了一个 DDB 专用的命令行工具 `isql`,`isql` 访问 DDB 时通过获取 DDB 的元数据,会直接作用于 QS 后端的数据库节点,因此不会对 QS 节点造成任何影响。

1. QS 命令行

通过 `show commands` 查看 QS 支持的命令,除了基本的 DML 语句外, QS 支持一些 DDB 特有的 `show` 命令:

show (full) processlist:

```
mysql> show processlist;
```

Id	User	Host	db	Command	Time	State	Info
1638	qs	10.120.146.133	ddb_demo	Query	2	Dispatching	sql:SELECT ID,
1639	qs	10.120.146.133	ddb_demo	Query	0	Dispatching	sql:SELECT ID,
1640	qs	10.120.146.133	ddb_demo	Query	0	Dispatching	sql:SELECT ID,
1641	qs	10.120.146.133	ddb_demo	Query	0	Dispatching	sql:SELECT t1.I
1642	qs	10.120.146.133	ddb_demo	Query	2	Dispatching	sql:SELECT t1.I
2119	qs	10.120.146.131	ddb_demo	Query	5	Dispatching	sql:SELECT MAX(

与 MySQL 不同的是, QS 的 `db` 列表示连接访问的 DDB name,另外, QS 对 SQL 的处理过程与 MySQL 大相径庭,因此 `show processlist` 返回 `State` 列的枚举值没有与 MySQL 保持一致,在 QS 中 `State` 列的枚举值包含:

- **Decoding:** 表示连接中有新请求,处于解码阶段,由于 SQL 还未被解码出来, `Info` 列的信息为空
- **Parsing:** 表示 SQL 处于语法解析阶段,从这个阶段开始 `Info` 列的信息为 SQL 字符串
- **Planning:** 表示 QS 正在为 SQL 生成执行计划
- **Connecting dbns:** SQL 在生成相应执行计划后,开始从连接池中获取到底层 DBN 的连接,若连接池中数目不够用,可能导致 SQL 长时间卡在这个状态

- Dispatching: 成功获取到各个 DBN 的连接后, QS 将 SQL 下发到各个 DBN 上执行, 在 MySQL 压力较大的情况下, Status 大部分会处于这个状态
- Coding: SQL 执行结束后, 将结果或结果集编码返回给客户端
- Writing net: 编码过程结束后, QS 将序列化结果写入网络的过程

QS 返回的 Command 列与 MySQL 也有些许差别, 出了标准的 Sleep, Query, Update, Insert 命令外, QS 中的 Command 列还包含以下三个枚举值:

- Mina: 当 State 处于 Decoding, Coding 以及 Writing net 三种状态时, Command 会显示为 Mina, 表示当前 SQL 处理过程独立于 DDB, 由 Mina 框架调用
- Isql: 所有的 QS 运维命令, 以及支持的 isql 命令在处理过程中 Command 都会显示为 Isql, Isql 命令下的 Status 和 Info 都为空
- Closing: 当连接断开, QS 正在回收该连接相关联的资源时, Command 显示为 Closing

总体来说, QS 的 show processlist 命令与 MySQL 相似, 除了通过 MySQL 命令行查看外, 也可以帮助第三方应用实时监控 QS 的 SQL 运行状态。

可以通过 show full processlist 查看满长度的 SQL。

show connection_pool:

show connection_pool 命令可以实时查询当前 QS 内部连接池的使用状况, MySQL 命令行下的结果如下所示:

```
mysql> show connection_pool;
```

Type	User	DBN	Active	Idle
NORMAL	qs	jdbc:mysql://10.120.146.129:3306/dbn2	23	4
NORMAL	qs	jdbc:mysql://10.120.146.129:3306/dbn1	27	4
NORMAL	qs	jdbc:mysql://10.120.146.130:3306/dbn3	11	1
NORMAL	qs	jdbc:mysql://10.120.146.130:3306/dbn4	12	4
XA	qs	jdbc:mysql://10.120.146.129:3306/dbn2	37	5
XA	qs	jdbc:mysql://10.120.146.129:3306/dbn1	35	4
XA	qs	jdbc:mysql://10.120.146.130:3306/dbn3	43	7
XA	qs	jdbc:mysql://10.120.146.130:3306/dbn4	59	0

8 rows in set (0.00 sec)

DDB 中连接池组织为三层树形结构, 从上到下依次为: 是否 XA 连接, 用户名, DBN URL。

DBAdmin 中设置的连接池上限是指树形结构中每个叶子节点上子连接池的大小, 从上

图中我们可以清晰地看到，在默认连接池上限为 100 下，当前每个子连接池的活跃连接数和空闲连接数。

该功能可以实时查阅 QS 对每个 DBN 的连接压力，以及当前的连接池配置是否合理：若发现 Active 连接数与连接池上限基本持平，应当及时调整连接池上限，防止 SQL 执行因获取不到物理连接而长时间阻塞。

show status:

QS 支持的 show status 命令与 MySQL 完全无关，中 uptime 表示 QS 从启动开始的运行时长，单位为秒。其他结果按字面意思解释，不再赘述。

```
mysql> show status;
+-----+-----+
| Name                                | Value |
+-----+-----+
| dbn_connections                     | 83    |
| dbn_connections_idle                | 9     |
| dbn_connections_active              | 67    |
| dbn_xa_connections                  | 186   |
| dbn_xa_connections_idle             | 42    |
| dbn_xa_connections_active           | 29    |
| ddb_connections                     | 21    |
| ddb_connections_impl_close          | 1     |
| ddb_connections_hang_close          | 2     |
| ddb_statements                      | 8     |
| ddb_statements_impl_close           | 1     |
| ddb_statements_hang_close           | 0     |
| ddb_pre pares                       | 32    |
| ddb_pre pares_impl_close            | 0     |
| ddb_pre pares_hang_close            | 0     |
| dbn_pre pares                       | 2468  |
| uptime                              | 185801|
+-----+-----+
17 rows in set (0.01 sec)
```

show max_connections:

查询 QS 支持的最大连接数，默认 1000，Admin 用户可以通过 set max_connections=? 来设置这个变量。

show slow_log:

QS 支持打印慢语句，并可以动态调整慢语句的阈值，通过 show slow_log 查询慢语句阈值，Admin 用户通过 set slow_log=? 来设置慢语句阈值。slow_log 阈值默认 2000ms。

当 slow_log 设置为 0 时，QS 会将所有成功处理的 sql 打印在 qs.log 中，当 slow_log

设置为-1 时，不会打印任何慢语句。

show commands:

查看 QS 支持的所有 DML 之外的命令。

其他有用的 **show** 命令:

show create table tableName: 查看建表语句

show policies: 查看所有均衡策略

show tables: 查看所有表信息

show dbns: 查看所有数据节点信息

show dbns for policy policyName: 查看某均衡策略的所有数据节点信息

show dbns for table tableName: 查看某表的所有数据节点信息

show version: 查看 DDB 和 JDBC 的版本信息

2. DDB 命令行

DDB 命令行设计的初衷在于不通过 QS 的大批量数据导入导出，但本身 isql 工具支持 DBI 端所有的命令，因此 DDB 命令行在命令集上甚至比 QS 命令行要多，DDB 命令行支持的命令以及使用方法可以通过老版的 DDB 用户手册查看，这里重点说明一下 DDB 命令行的数据导入导出命令。

DDB 命令行的使用方式与 MySQL 命令行非常类似。

通过 `./ddb -H` 查看参数。一般使用方式如下：

```
./ddb -h127.0.0.1 -P6000 -uqs -pqs -Dddb_demo
```

DUMP

DUMP 命令导出分布式数据库中所有表或指定的表的数据。语法：

```
DUMP [OPTIONS] [(ALL|tables)] [INTO 'file'];
```

参数说明:

- ◆ **OPTIONS:** 选项, 可以是:
- ◆ **-t 'dir':** 导出为 CSV 格式, 每张表对应一个.txt 文件, 输出到 **dir** 指定的目录中。若不指定则导出为 INSERT 语句。
- ◆ **-w 'condition':** 只导出满足条件的记录。
- ◆ **-s 'select':** 指定查询语句。使用这一选项时即指定了完整的查询, 因此不能同时指定 **-w**、**-o**、**-l** 这些选项, 也不能指定 **ALL** 或 **tables**。指定查询语句时导出为 CSV 格式。
- ◆ **-o 'order_by':** 按 **order_by** 对导出的记录进行排序, 其中 **order_by** 可为 **PRIMARY** 表示按主键排序
- ◆ **-l records:** 最多只导出 **records** 条记录
- ◆ **-p:** 并行导出模式, 对每一张要导出的数据库表, 使用多个线程同时从存储该表的数据库节点导出数据, 但不同表的数据不会交错, 总是从各节点上导出完成一张表后再处理下一张表。注意若指定了 **-o** 选项则 **-p** 选项会被忽略。
- ◆ **-r row_delimiter:** 记录与记录之前的分隔符, 默认为"**\n**", **Windows** 上的换行符 "**\r\n**"用<lrcf>指定。支持字符串形式, 该分隔符不能在字段值中出现。
- ◆ **-a attr_delimiter:** 属性值之间的分隔符, 若不指定则为 **TAB** 键, 空格用<space>表示, **TAB** 键用<tab>表示。支持字符串形式, 该分隔符不能在字段值中出现。
- ◆ **-c charset:** 输入和输出文件使用的字符集, 若不指定则默认是 **core.charset** 的值。
- ◆ **ALL:** 导出所有表的数据。
- ◆ **tables:** 导出指定的表的数据, 为一个表名的列表, 用逗号分隔。
- ◆ **INTO 'file':** 在导出为 **INSERT** 语句时将结果写出到指定文件中, 否则会输出结果到标准输出。

命令输出除了导出的数据以外还包括标准输出显示的导出进度(若数据量大的话)。

工作模式: 中间件

相关选项: 无

SOURCE

SOURCE 命令执行一个 SQL 脚本。语法：

```
SOURCE [-q] [-b batch_size] 'file';
```

参数说明：

- ◆ **-q**: 安静模式，不输出结果。
- ◆ **-b batch_size**: 批量大小，即每隔 **batch_size** 条语句提交一次。若不指定批量大小，则在设置了 **autocommit** 选项时每执行一条语句都提交，否则不会提交事务。

命令在安静模式下不输出结果，不指定安静模式则按照各个脚本语句输出。

工作模式：中间件、轻量级

相关选项：无

LOAD

LOAD 命令导入 CSV 格式的数据到指定的表中。语法：

```
LOAD options 'file' [REPLACE] INTO table[(column_list)];
```

参数说明：

- ◆ **options**: 选项。可用以下选项：
- ◆ **-r row_delimiter**: 记录与记录之前的分隔符，默认为"**\n**"，Windows 上的换行符 "**\r\n**"用**<lrcf>**指定。支持字符串形式，该分隔符不能在字段值中出现。
- ◆ **-a attr_delimiter**: 属性值之间的分隔符，若不指定则为 TAB 键，空格用**<space>**表示，TAB 键用**<tab>**表示。支持字符串形式，该分隔符不能在字段值中出现。
- ◆ **-c charset**: 输入和输出文件使用的字符集，若不指定则默认是 **core.charset** 的值。
- ◆ **-t dir**: 临时文件目录，默认为当前目录下的 **load.temp** 子目录。
- ◆ **-e**: 出错时继续执行。
- ◆ **--skip-load**: 只执行输入分割操作，不加载数据。
- ◆ **--smart**: 智能模式，当输入中包含中文且中文乱码导致分割失败时，可使用此模式增加分割程序的智能程度。

- ◆ **REPLACE**: 遇到重复键值时替换原有记录，若不指定则跳过这一记录。
- ◆ **column_list**: 指定文件中各列对应的列名，当文件中包含的列数少于表中的属性数，或文件中的列与表中属性的顺序不一致时可以利用这一功能详细指定文件各列对应的表中属性。

导入过程为先根据均衡字段的值将输入分割为多个文件，表所在的每个数据库节点对应一个文件，然后使用 MySQL 的"LOAD DATA LOCAL INFILE"命令导入数据。

分割产生的结果文件名为 **dbn/file.txt**，其中 **file** 为输入文件名，**dbn** 为数据库节点名。

当使用智能模式或出错时继续执行时，分析过程中产生的错误信息或警告信息将被输入到临时文件目录中的 **load_xxx.log** 文件中，其中 **xxx** 为当前时间。

工作模式：中间件

相关选项：无

3. 实时统计

DDB 有实时的计数统计：保存固定时间间隔内的 **query**、**insert**、**update/delete** 以及 **transation** 数目，由此可以计算相应 **qps**、**ips**、**udps** 以及 **tps**。实时计数的命令格式为：

count [all/transaction/query/insert/update] (from [uptime] to [downtime])

其中 **uptime** 和 **downtime** 分别表示统计开始时间和结束时间，单位为 **QS** 统计时间间隔（统计周期），默认 1 分钟。**downtime** 必须小于 **uptime**，**uptime** 上限为 1440，若统计周期采用 1 分钟，则最多可以保存查询 24 小时内的实时计数。

例如输入 **count all from 10 to 5**，QS 返回的结构如下所示：

```
mysql> count all from 10 to 5;
```

Time	Query	Insert	U/D	Commit	Rollback	Trasaction
09:35:30	32340(41)	862(256)	518(112)	213(201)	0(0)	1165(301)
09:36:30	61573(15)	1463(132)	860(52)	337(98)	0(0)	1928(145)
09:37:30	27589(29)	617(221)	491(104)	225(140)	0(0)	932(268)
09:38:30	23648(56)	609(316)	406(109)	163(225)	0(0)	841(369)
09:39:30	19665(53)	418(625)	301(207)	127(243)	0(0)	623(629)
09:40:30	23357(48)	603(430)	396(142)	161(305)	0(0)	811(490)

6 rows in set (0.00 sec)

返回结构为当前时间之前的 10 分钟到 5 分钟之间，每分钟的所有统计数据，由于 QS 默认的统计周期为 1 分钟，所以 from 10 to 5 即表示过去 10 分钟到 5 分钟之间。

图中所示，Time 表示统计的时间点，每次统计点过后所有计数会清 0，以进入下一个统计周期，括号内的数字表示 response time，图中所示每分钟 query 数目为 2W 左右，每分钟 transaction 为 1K 左右。

若 count 后面没有跟 from to, 则返回过去一个统计周期内的统计数据，如输入 count all, 返回如下：

```
mysql> count all;
+-----+-----+-----+-----+-----+-----+-----+
| Time      | Query      | Insert     | U/D       | Commit     | Rollback   | Trasaction |
+-----+-----+-----+-----+-----+-----+-----+
| 09:57:30  | 16521(54)  | 402(745)   | 287(249)  | 129(460)   | 0(0)       | 573(814)   |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

count query from 10 to 5 和的结果为：

```
mysql> count query from 10 to 5;
+-----+-----+-----+
| Time      | Query | Avg_rt |
+-----+-----+-----+
| 09:51:30  | 18520 | 48     |
| 09:52:30  | 21748 | 41     |
| 09:53:30  | 21290 | 39     |
| 09:54:30  | 18597 | 45     |
| 09:55:30  | 17026 | 48     |
| 09:56:30  | 22938 | 37     |
| QPS       | 333   | 43     |
+-----+-----+-----+
7 rows in set (0.00 sec)
```

另外，count (from [uptime] to [downtime]) [all/transaction/query/insert/update]这种写法也是支持的，count from 10 to 5 transaction 返回结果如下：

```
mysql> count from 10 to 5 transaction;
```

Time	AutoCommit	Commit	Rollback	total	Avg_rt
09:57:30	444	129	0	573	814
09:58:30	565	176	0	741	419
09:59:30	567	173	0	740	538
10:00:30	650	199	0	849	498
10:01:30	728	163	0	891	343
10:02:30	484	130	0	614	644
TPS				12	542

```
7 rows in set (0.00 sec)
```

QS 也支持 `count [type] from uptime` 的命令，这时候 `to` 缺省为 0，即当前时间。

`count` 的统计周期可以在 QS 配置文件中定义，单位 `ms`，默认配置如下：

```
<count_interval>60000</count_interval>
```

五. 连接池范例

```
<property name="driverClassName" value="com.mysql.jdbc.Driver" />
<!-- 基本属性 url、user、password -->
<property name="url" value="jdbc:mysql://ip:port/db?autoReconnect=true&
rewriteBatchedStatements=true&socketTimeout=30000&connectTimeout=3000" />
<property name="username" value="root" />
<property name="password" value="123456" />
<!-- 配置初始化大小、最小、最大 -->
<property name="maxActive" value="500" />
<property name="initialSize" value="10" />
<property name="minIdle" value="501" />
<!-- maxWait 获取连接等待超时的时间,单位 ms -->
<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="180000" />
<!-- minEvictableIdleTimeMillis 一个连接在池中最小空闲的时间, 单位是毫秒-->
<property name="minEvictableIdleTimeMillis" value="600000" />
<property name="numTestsPerEvictionRun" value="5" />
<property name="validationQuery" value="SELECT 1" />
<property name="testWhileIdle" value="true" />
<property name="testOnBorrow" value="false" />
<property name="testOnReturn" value="false" />
```

要点:

1. `maxIdle > maxActive > initialSize`, `maxIdle` 设大防止频繁创建连接
2. 需要将 `testWhileIdle` 设置为 `true`, 表示每 `timeBetweenEvictionRunsMillis` 段时间, 对 `numTestsPerEvictionRun` 个连接通过 `validationQuery` 的语句进行探活, 如果探活失败, 将该连接从连接池中剔除
3. QS 的连接超时时间为 7 个小时, 如果一个连接连续 7 个小时没有任何请求, QS 会将这个连接关闭, 这就要求应用连接池在 7 小时内对一个连接至少探活一次, 从而计算出 `numTestsPerEvictionRun` 应当取的值。

例如本例中没 3 分钟探活 5 个连接, 7 个小时可以探活 700 个连接 `> maxActive`

六. 规范与边界

DDB 是基于 MySQL（老版本中也可以基于 Oracle）开发的分库分表中间件，在 DDB 系统设计中，一直以来秉承以尽量做到透明使用为原则，因此 DDB 在使用上，尤其在 QS 模式下，与访问 MySQL 是十分相似的。

但是也必须认识到，DDB 不像 MySQL 是一个标准化的产品，作为一个分布式数据库，DDB 中的 SQL 兼容性首先无法做到 100%兼容，在 SQL 执行效率上，也会有些限制，对 DDB SQL 语法和性能上的限制，请参考前几章说明，这里不再赘述。

DDB 系统边界

为了帮助应用合理使用 DDB，也方便在出现问题后有据可查，在此对 DDB 的系统边界说明如下：

1. 每个 DDB 发布的版本，都会携带相应的应用开发者白皮书，白皮书中描述了这个版本 DDB 支持的 SQL 类型以及各种使用方法，如申请自增 ID 方法，设置流方式方法，白皮书描述内容外的功能和 SQL 不保障兼容，另外一些 SQL 可能会有性能问题，如非均衡字段上的 join。
2. DDB 的 QS 组件是 JAVA 程序，为了保障 QS 的响应速度，对 QS 使用的内存做了限制，默认 2G，QS 使用内存越大，full gc 所带来的卡顿越大，这里并不建议产品管理员自己改变 QS 使用的内存大小，对一些特殊的 OLAP 应用，可以因地制宜。
3. QS 是个轻量级的代理服务器，在应用使用不当时可能出现 OOM 的情况，为了避免这种情况的发生，假设应用访问单个 QS 的最高并发为 n ，则规定如下：
 - a) DDB 的单个 SQL 长度不要超过 $2048 / (n * 10)$ ，单位 mb
 - b) DDB select 的结果集大小不能超过 $2048 / n$ ，单位 mb

为了方便计算，对单个 SQL 长度限制和 select 结果集限制定义在 5m 和 100m，如果应用使用的 SQL 超出上述限制，造成 QS OOM，之间造成的 DDB 不可用，不算做 DDB 的可用性问题，责任由应用方和 SQL 审查方承担。

对 SQL 结果集大小的限制其实不能一概而论，如果应用确实有些大结果集查询，可以使用 QS 的流方式，具体请参考 select 一节中的流方式设置方法（注意使用流方式可能

带来的风险)，未来 DDB 也会通过底层游标的方法对这种大结果集的场景做出优化。另外，即便应用 SQL 结果集超出 100m，也不见得一定造成 OOM（如果应用并发读 20 同时在执行这种 SQL，就很容易 OOM 了），具体场景还要具体分析，这里将 select 结果集的大小边界设为 100m，也是为了帮助应用规避问题。

DDB 使用规范与责任划分：

1. 遵守杭研发布的数据库命名规范（内部适用）。
2. 在部署 DDB 时，如果选择老版本，遇到了 BUG，且这些 BUG 在已经发布的较新版本中得到了修复，DDB 开发组不负责任，所以如果选择老版本，需要产品管理员对新版本的 **release notes** 进行研读。
3. DDB 上的 SQL 需要现在测试环境中经过测试才准上线，**且要求测试环境中的分库在 2 个或 2 个以上**。对通过应用测试，但是线上执行错误的 SQL，属于 DDB 开发组全责。因没有经过测试，并且执行的 SQL 属于白皮书描述的支持的 SQL 范畴，而在线上 SQL 执行出错，应用方与 DDB 开发组各承担一半责任，或经过测试，但测试中没有发现错误，导致线上执行错误的情况，应用方与 DDB 开发组同样各承担一半责任。
4. 在应用线上或测试中发现的 BUG，由开发组和应用协商解决方案，一般是采用其他 SQL 绕过或通过 hotfix 或升级来解决。
5. 因使用的 SQL 不当，导致的性能以及相关的问题，原则上 DDB 开发组不负责任。