DDB 应用开发者白皮书 V1.0

By 马进

目录

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 V4.5.7 即以上，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，建议在阅读本书之外，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:

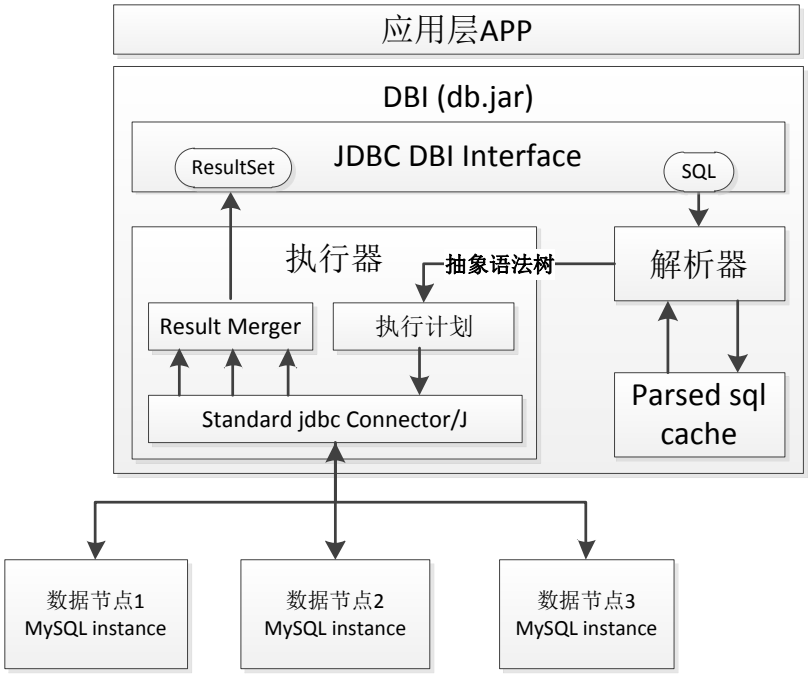

图 1 DBI 模式架构图

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：

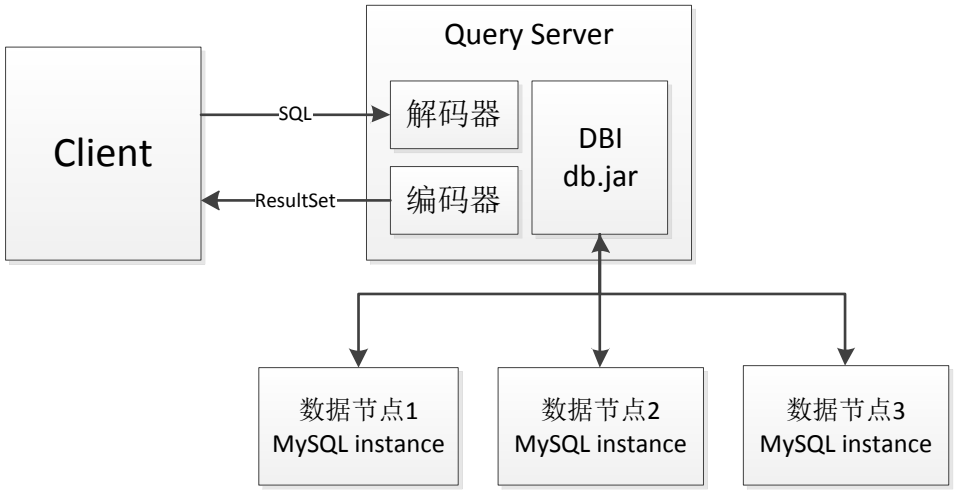

图 2 QS 模式架构图

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

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

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.
- 不支持子查询
- 不支持 UNION/INTERSECT/EXCEPT 等集合操作
- 不支持 INSERT INTO … SELECT 语句
- 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 4.5.7，4.5.7 之前的版本与 4.5.7 差别主要在不支持别名，不支持复杂的 having 条件，不支持透明全局 ID 分配等，具体请参考 DDB 用户手册 SQL 语法说明一章，或直接咨询 DDB 开发组。由于旧版本中存在个别语法上的 BUG，建议使用 DDB 最新稳定版 4.5.7。

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：子查询 IN 操作不支持
- NOT LIKE 操作
- NOT BETWEEN 操作
- 正则表达式运算：RLIKE、REGEXP
- **DDB 目前不支持位运算^、|、&**

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 子句的列表中**
- 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 子句的列表中**
- 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 多表联接

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 字句中表出现的顺序，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，下表说明了两者之间的关系：

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 DDB 流方式保障不会内存溢出

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 OOM，使用方式如下：

```
Statement.setFetchSize(n)
-- PreparedStatement 亦可，n>0,表示每次从 DDB 拉去的结果集大小
-- 这个方法的调用需要在执行 SQL 之前，且 DDB 的流方式与 MySQL 流方式的使用没有关联
```

关于 DDB 流方式的更多内容请参照 DDB 用户手册。

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 … SELECT 语句
- 不支持 INSERT INTO … SET …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 MSB（Master 统一批量分配）类型和 TSB（基于时间戳的 ID 分配）类型，使用 MSB 的话，DBI Client 每次分配完固定个数的自增长 ID 后会向 Master 申请新一批 ID，当 Master 处于不可用状态时，如宕机或升级，会导致分配全局 ID 功能不可用，TSB 则无此问题。关于 MSB 和 TSB 的更多内容请参照 DDB 用户手册。

在应用代码中，使用 DDB DBI 的全局 ID 分配功能方法有两种：

1. 调用 com.netease.backend.db.DBConnection.allocateRecordId 函数，如下所示:

```
import com.netease.backend.db.DBConnection;
Connection conn = DriverManger.getConnection(url, user, pass);
long id = ((DBConnection)conn).allocateRecordId(<table>);
```

其中<table>为表名。分布式数据库保证生成的 ID 对指定的表名不重复，对不同的表名生成的 ID 则可能重复。获取到 ID 通过 insert 指定自增长 ID 列插入到 DDB 中。

2. 第一种生成全局 ID 方式属于 DDB 老版本的遗留做法，在目前 DDB 最新稳定版 4.5.7 中，可以使用标准 JDBC 的先插入，再通过接口获取全局自增 ID 的方法：

```
Connection conn = DriverManger.getConnection(url, user, pass);
Statement s = conn.createStatement();
s.executeUpdate("insert into test(seq, 'a')");
-- DDB4.5.7 中 seq 可不写，不写则与使用标准 JDBC 完全一致。一些旧版本必须写 seq。
ResultSet rs = s.getGeneratedKeys();
rs.next();
long id = rs.getLong(1);
```

推荐用户使用第二种方式获取全局自增长 ID，一来开销较小，二来应用代码可移植性较强，三来也可以更好地控制 DDB 对用户代码的侵入。

当使用 QS 时，同样有两种对应的方法获取自增长 ID：

1. 对应第一种方法，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。

```
import com.mysql.jdbc.Driver;
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManger.getConnection(url, user, pass);
ResultSet res = conn.createStatement().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。

总结一下，4.5.7 的 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会被拆分为：

```
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 defaultxa=0/1 命令设置连接是否开启两阶段事务，若设置 defaultxa=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。

- 不支持 WHERE 子句中包含子查询

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
- 不支持 WHERE 子句中包含子查询

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

/*FORWARDBY (TABLENAME=TABLE1, TABLE2, … ; BFVALUE=value1,value2,…) */

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=value1, value2,…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:

```
/* FORWARDBY(TABLENAME=T1, T2)*/ select * from T1, T2 where T1.ID = T2.ID and T1.Score > 90
/* FORWARDBY(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 lsql 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。

对于每张表支持用括号标识的多个均衡字段以逗号隔开，单均衡字段则可选用括号（即单均衡字段兼容原设计）：

/*FORWARDBY(TABLENAME=TABLE1，TABLE2，…；BFVALUE=(TABLE1.value1),(TABLE2.value2…)) */

其中 TABLE1.value1 为表 TABLE1 的均衡字段；TABLE2.value1 为表 TABLE2 的均衡字段。

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．附录

1. DBI 接口总结

DDB 的 JDBC 驱动 DBI 支持 JDBC 3.0（JAVA）标准接口中的下列方法。调用未在下列给出的 JDBC 方法时将抛出异常。

java.sql.Driver 类

public Connection connect(String url, Properties info) throws SQLException
public Boolean acceptsURL(String url) throws SQLException
public int getMajorVersion()
public int getMinorVersion()
public boolean jdbcCompliant()

java.sql.Connection 类

public Statement createStatement( ) throws SQLException
public PreparedStatement prepareStatement(String sql) throws SQLException
public void setAutoCommit(boolean autoCommit) throws SQLException
public boolean getAutoCommit() throws SQLException
public void commit() throws SQLException
public void rollback() throws SQLException
public void close() throws SQLException
public boolean isClosed() throws SQLException
public boolean isReadOnly() throws SQLException
public boolean setReadOnly() throws SQLException
public void clearWarnings() throws SQLException
public SQLWarning getWarnings() throws SQLException

java.sql.Statement 类

public ResultSet executeQuery(String sql) throws SQLException
public int executeUpdate(String sql) throws SQLException
public void close() throws SQLException
public boolean execute(String sql) throws SQLException
public ResultSet getResultSet() throws SQLException
public int getUpdateCount() throws SQLException
public Connection getConnection() throws SQLException
public int getFetchSize() throws SQLException
public ResultSet getGeneratedKeys() throws SQLException
public int getMaxRows() throws SQLException
public SQLWarning getWarnings() throws SQLException
public void clearWarnings() throws SQLException

public boolean getMoreResults() throws SQLException
public int getResultSetConcurrency() throws SQLException
public int getResultSetType() throws SQLException
public boolean getMoreResults(int current) throws SQLException
public int getResultSetHoldability() throws SQLException
public void setFetchSize(int rows) throws SQLException
public void setMaxRows(int max) throws SQLException

java.sql.PreparedStatement 类

注：未列出 PreparedStatement 继承自 Statement 的接口。

public ResultSet executeQuery() throws SQLException
public int executeUpdate() throws SQLException
public boolean execute() throws SQLException
public void setNull(int parameterIndex, int sqlType) throws SQLException
public void setBigDecimal(int parameterIndex, BigDecimal x) throws SQLException
public void setBinaryStream(int index, InputStream is, int size) throws SQLException
public void setBlob(int parameterIndex, boolean x) throws SQLException
public void setBoolean(int parameterIndex, boolean x) throws SQLException
public void setByte(int parameterIndex, byte x) throws SQLException
public void setBytes(int parameterIndex, byte[] x) throws SQLException
public void setDate(int parameterIndex, Date x) throws SQLException
public void setDouble(int parameterIndex, double x) throws SQLException
public void setFloat(int parameterIndex, float x) throws SQLException
public void setInt(int parameterIndex, int x) throws SQLException
public void setLong(int parameterIndex, long x) throws SQLException
public void setShort(int parameterIndex, short x) throws SQLException
public void setString(int parameterIndex, String x) throws SQLException
public void setTime(int parameterIndex, Time x) throws SQLException
public void setTimestamp(int parameterIndex, Timestamp x) throws SQLException
public void setObject(int index, Object value) throws SQLException

java.sql.ResultSet 类

public void close() throws SQLException
public BigDecimal getBigDecimal(int column) throws SQLException
public BigDecimal getBigDecimal(String columnName) throws SQLException
public InputStream getBinaryStream(int column) throws SQLException
public InputStream getBinaryStream(String columnName) throws SQLException
public Blob getBlob(int column) throws SQLException
public Blob getBlob(String columnNames) throws SQLException
public Clob getClob(int column) throws SQLException
public Clob getClob(String columnNames) throws SQLException
public boolean getBoolean(int column) throws SQLException

public boolean getBoolean(String columnName) throws SQLException
public byte getByte(int column) throws SQLException
public byte getByte(String columnName) throws SQLException
public byte getBytes(int column) throws SQLException
public byte getBytes(String columnName) throws SQLException
public Date getDate(int column) throws SQLException
public Date getDate(String columnName) throws SQLException
public double getDouble(int column) throws SQLException
public double getDouble(String columnName) throws SQLException
public float getFloat(int column) throws SQLException
public float getFloat(String columnName) throws SQLException
public int getInt(int column) throws SQLException
public int getInt(String columnName) throws SQLException
public long getLong(int column) throws SQLException
public long getLong(String columnName) throws SQLException
public DBResultSetMetaData getMetaData() throws SQLException
public Object getObject(int column) throws SQLException
public Object getObject(String columnName) throws SQLException
public short getShort(int column) throws SQLException
public short getShort(String columnName) throws SQLException
public String getString(int column) throws SQLException
public String getString(String columnName) throws SQLException
public Time getTime(int column) throws SQLException
public Time getTime(String columnName) throws SQLException
public Timestamp getTimestamp(int column) throws SQLException
public Timestamp getTimestamp(String columnName) throws SQLException
public boolean next() throws SQLException

java.sql.ResultSetMetaData 类

public int getColumnCount()
public String getTableName(int column)
public int getColumnType(int column)
public String getColumnName(int column)
public String getColumnTypeName(int column)

java.sql.Blob 类

public long length()
public byte[] getBytes(long pos, int length)
public InputStream getBinaryStream()

2. DBI 示例程序

下面程序演示了使用标准 jdbc 接口操作 DDB 的方法。

```
 try {
       Class.forName("com.netease.backend.db.DBDriver");
     } catch (ClassNotFoundException e) {
       throw e;
     } // 加载 DBI JDBC 驱动

     PreparedStatement pstmt = null;
     Connection con = null;
     ResultSet rs = null;

     try {
       // 建立连接
       String url = "127.0.0.1?key=d:\\secret.key";
       con = DriverManager.getConnection(url, username, password);

       // 向表中插入数据
       pstmt     =     con.prepareStatement("insert     into     testtable
values(seq,?,?)");
       pstmt.setLong(1, 1);
       pstmt.setString(2, "1");
       pstmt.setString(3, "1");
       pstmt.executeUpdate();
       ResultSet rs = pstmt.getGeneratedKeys();
       if(rs.next()){
         System.out.println("新生成的 id: " + rs.getLong(1));
       }
       rs.close();
       rs = null;
       pstmt.close();
       pstmt = null;

       // 对表中的数据进行查询
       pstmt = con
           .prepareStatement("select * from testtable where id = ?");
       pstmt.setLong(1, 1);
       rs = pstmt.executeQuery();
       while (rs.next()) {
         System.out.println(rs.getString("value"));
       }

     } catch (SQLException e) {
       throw e;
     } finally {
       if (rs != null) {
         rs.close();
       }
       if (pstmt != null) {
         pstmt.close();
       }
       if (con != null) {
         con.close();
       }
     }
```