

分布式数据库 DDB

用户手册 (V 4.5)

编 写 人：王磊、邱似峰、施勇、廖宇俊、柯志锋

编写时间：2012-07-24

部 门 名：网易杭州研究院

审 核 人：

审核时间：

[illegible]

1 前言

1.1 目的

本文档是为了帮助分布式数据库的系统管理员、开发人员以及其他相关技术人员，了解分布式数据库系统。本手册对应的 DDB 版本是 DDB4.5。

1.2 读者

分布式数据库的系统管理员、应用开发人员，以及其他需要了解此系统的相关人员。

1.3 相关文档

《网易_分布式数据库_Concepts》
《分布式数据库 v3.0 详细设计》
《网易_分布式数据库_查询优化及执行计划设计》
《网易_分布式数据库_分布式事务管理器实现设计》
《网易_分布式数据库_DBA 管理工具使用说明》
《网易_分布式数据库_辅助工具使用说明》
《网易_分布式数据库_管理员手册》
《网易_分布式数据库_开发者手册》
《网易_分布式数据库_性能优化》

1.4 术语和定义

<待补充>

策略：

桶：

分区字段：

1.5 常用链接

开发组 wiki: <http://qa.163.org/wiki/ddb?nav=211>.

产品发布公告页: <http://qa.163.org/projects/ddbproduct/news?nav=211>

Svn 地址: <https://svn.163.org/svn/ddbirfs/trunk/blog-db/>

2 系统架构介绍

2.1 开发目的

分布式数据库系统为网易公司基于海量数据的各种前台应用提供统一的数据库管理平台，典型应用包括：点卡计费系统、Blog 系统、POPO 等。这类产品的特点是：采用统一的数据库存储平台、数据库单表存储的数据量大、高并发访问量。随着数据量和访问量的不断增加，依靠提高数据库的服务器硬件性能所能获得的存取性能的提升空间越来越小，如果采用目前较为成熟的商业分布式存储解决方案，如 Oracle 的 RAC，往往需要特殊的硬件环境，如 SAN(Storage Area Network)，并需要支付昂贵的软件使用费。

该平台通过中间件的形式为前台提供一台虚拟的数据库服务器，而中间件在后台实际管理着多个数据库节点，为了解决单张表的数据和访问量过多造成的系统性能瓶颈，该系统允许将单张表分布到不同的数据库节点上进行性能均衡。同时允许后台数据库节点的动态加入，支持在线的可扩展，而不会影响系统的正常服务。

2.2 总体架构

分布式数据库由一台管理服务器（Master）和多台数据库节点服务器（DBN，Database Node）组成。数据根据一定的均衡策略分布在每个 DBN 上，客户机（Client）通过特定数据库访问接口（DBI，Database Interface）透明地访问存储在分布式数据库中的数据。

下图是一个采用分布式数据库的典型三层网站物理架构。应用服务器作为后端数据库的 Client 通过 JDBC 接口访问存储在多个数据库节点上的数据，但是对于 Client 本身来说和访问传统的单台数据库节点没有太大区别，数据访问的复杂性被隐藏在了 DBI 中。

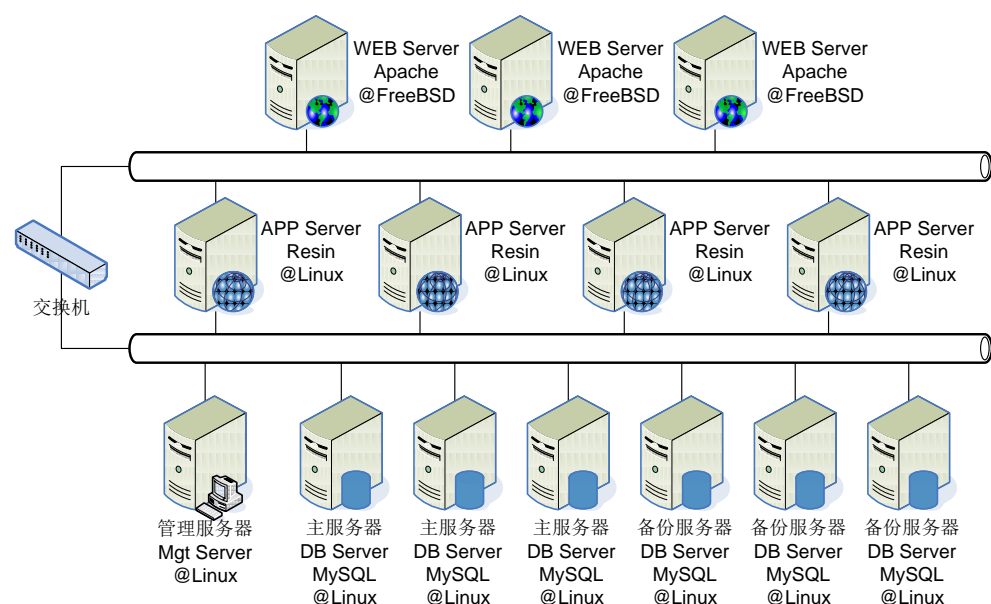


图 1. 采用分布式数据库的三层网站物理架构

如图 2 所示，分布式数据库系统由客户机（Client）、查询服务器（QS）、数据库节点（DBN）、管理服务器（Master）和管理客户机五类节点组成（不考虑数据库备份）。

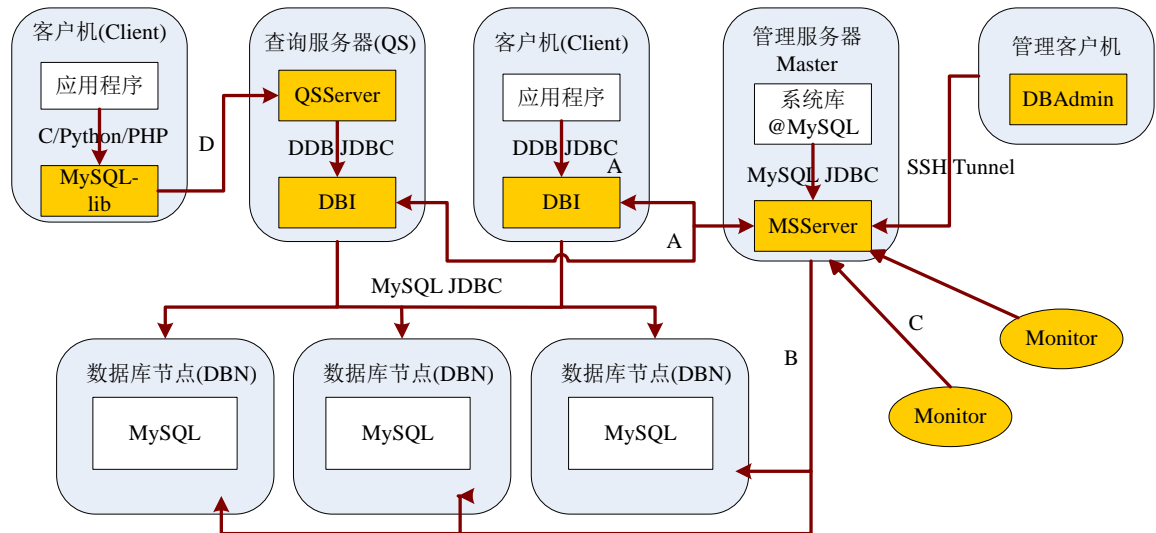


图 2. 分布式数据库的组成架构

分布式数据库系统提供的查询处理功能由 DBI 提供，部署于客户机之上，以 jar 包的形式提供给应用程序使用。DBI 通过 JDBC 驱动程序与部署于后台的数据库节点之上的 MySQL 数据库交互。

由于每个 DBI 都会占用一定的后台数据库连接资源。为了有效地利用这些资源，多个客户机可以通过查询服务器对分布式数据库进行访问，查询服务器作为中间服务器通过 DBI 访问后台数据库节点，这样多个客户机就可以共享这个 DBI 上的连接资源。客户机上的应用程序通过轻量级的 DBI 与查询服务器通信。

分布式数据库中一个重要的组件是管理服务 MSServer，也称为分布式数据库 Master，运行于管理服务器之上，是系统的核心，提供 ID 分配、系统配置管理、资源分配、负载统计、数据迁移、数据状态监控与管理等众多功能。Master 使用系统数据库保存整个分布式数据库的配置、负载、报警等信息。

管理员日常工作主要通过 DBAdmin 这一图形化管理工具完成，DBAdmin 与 Master 通信，可实现系统配置变更、状态或负载查询与统计、数据迁移、控制 DBI 和 Agent 等功能。

Monitor 程序负责监控 Master 是否能正常工作。为提高可靠性，Monitor 程序可以部署在多台节点上。

Agent 运行于每个 DBN 上，负责与 Master 交互。主要用于检测 DBN 上的 MySQL 运行情况、协助 Master 进行数据备份、导出、以及启动和关闭 MySQL 服务。

组成分布式数据库系统的各组件之间的交互关系在图中用深红色线标出，其箭头表示交互过程建立的方向。组件之间的交互简述如下：

DDB JDBC: DBI 提供 JDBC Driver 接口，供应用程序或查询服务器访问分布式数据库时调用。

MySQL JDBC: 这有两个地方：一是 DBI 与 DBN 上的 MySQL 服务器之间通过 JDBC 通信；二是 Master 与系统数据库 MySQL 服务器之间通过 JDBC 通信

SSH Tunnel: 在产品环境中，为保证安全性，DBA 工具主动通过 SSH Tunnel 连接到 Master，管理工具只负责接收用户输入，所有功能实现上都是通过该连接发送给 Master 完成的。

DBI 与 Master 之间的交互是双向的。由 DBI 主动建立的到 Master 的连接实现获取集群配置信息、负载信息上报、分布式事务故障信息上报功能。由 Master 主动建立的到 DBI 的连接实现配政更新通知等功能

注: Monitor 用于检测 Master 是否正常；LWDriver 用于提供多 DDB 的连接；Agent 用于监控数据库节点运行情况。这三个组件的功能已由目前 DDB 的其他模块代替，所以不需要再单独部署。

2.3 分区和负载均衡

为实现数据的分布式存储和负载均衡，需要将数据进行有效的划分，按照一定的策略存储在多个数据库节点上。分布式数据库的分区借鉴了数据库分区的概念和思想。数据的水平分区是数据库用于实现可伸缩性的重要方法。分区实现时的核心问题是如何确定元组存放的位置。数据库中水平分区的方法一般有 **hash** 分区、**range** 分区、**list** 分区等几种。根据目前应用需求，当前分布式数据库系统支持哈希分区。

为实现高效可调整的分区策略，系统采用 **hash** 表和映射表两级映射的方式实现。即首先根据一个固定的 **hash** 算法和特定的字段值将表中的每条记录根据映射到一个 **hash** 桶上，然后使用一个动态可调的 **map** 将多个 **hash** 桶映射到一个存储节点上。通过这样一次间接的映射，完成表记录到存储节点的映射。这个用户计算 **hash** 值的字段称为均衡字段。

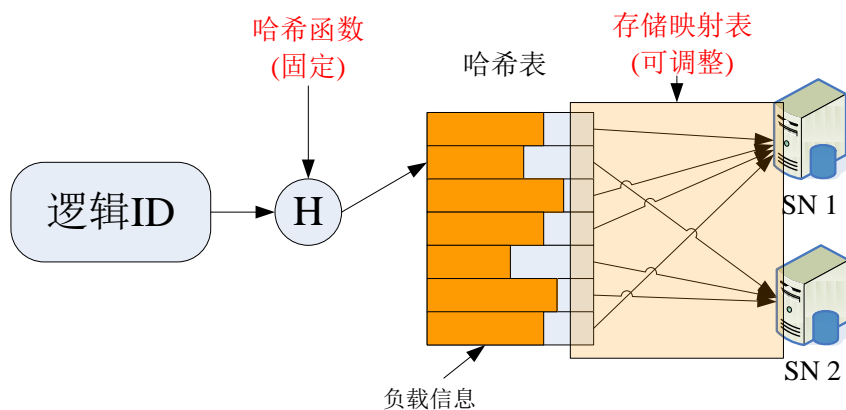


图 3. 分布式数据库的分区策略

采用两级映射的方案，消除了直接使用 **hash** 表映射无法调整和直接使用 **map** 映射占用空间过大的缺点，扩充后端节点时，仅需要修改 **map** 映射，就能完成迁移，结合了 **hash** 的高效性和 **map** 的易管理性。由于分区决定了数据存储位置，调整分区策略的效果就是调整各数据库节点的负载，因此这里说的分区策略与均衡策略是一致的，在术语上可通用。

多个数据表之间可能存在关联关系，如点卡计费系统中的数据表大体可分为三类，一类是用户名相关，一类是卡号相关，另一类是其他表数据。同类数据之间宜采用相同的分区或均衡策略（均衡字段和 **hash** 算法），如用户名相关的表都根据用户名进行均衡。分布式数据库使用均衡策略表示表之间的分类关系，属于同一类的表只要使用相同的均衡策略，在进行负载均衡调整时会以桶为单位被统一处理。

当数据库节点负载过高需要增加节点或者节点之间的负载不够均衡时需要重新调整，这可以通过数据迁移来解决。数据迁移用于在 **DBN** 节点之间迁移数据，并保证迁移结束后 **Client** 仍然可以像过去一样访问分布式数据库。通过数据迁移可以从高负载的 **DBN** 上迁移数据到低负载的 **DBN** 上实现负载均衡。

数据迁移基于数据分区的两级映射思想。表记录通过一个固定的 **hash** 算法和特定的字段值映射到一个 **hash** 桶上，再将桶映射到特定的 **DBN** 上。如果以桶为单位在 **DBN** 之间迁移数据，数据迁移后只需要修改桶到 **DBN** 之间的映射关系，**Client DBI** 就可以通过新的映射关系在 **DBN** 上找到记录。下图给出了通过桶的迁移实现系统负载均衡的原理。

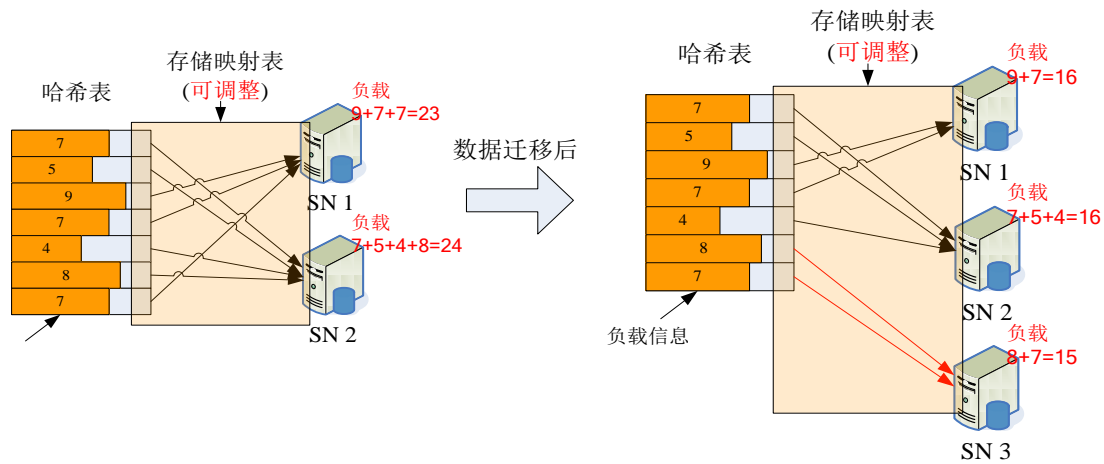


图 4. 通过数据迁移均衡系统负载

两级映射机制为修改映射关系提供了便利，但是数据迁移是数据库管理员 DBA 需要额外关注的事情，关于数据迁移的详细论述，参见本手册下面的章节。

2.4 分布式查询处理

对于应用程序来说，分布式数据库中间件需要提供透明的数据访问接口，即像在一个单独的数据库节点上一样在分布式数据库上执行 SQL 语句。在中间件中采用了分布式查询处理技术实现了这种透明的 SQL 语句执行功能。分布式查询处理的一般流程如下图所示：

系统解析输入 SQL 语句后，根据查询条件访问“存储映射表”，得到需访问的后台存储节点，并生成查询计划。查询计划包括对各 SN 的查询以及对多个 SN 返回结果的进一步处理。如一个 ORDER BY 语句需要首先对各 SN 执行 ORDER BY 语句，并对结果进行合并（使用 MERGE SORT 操作）。

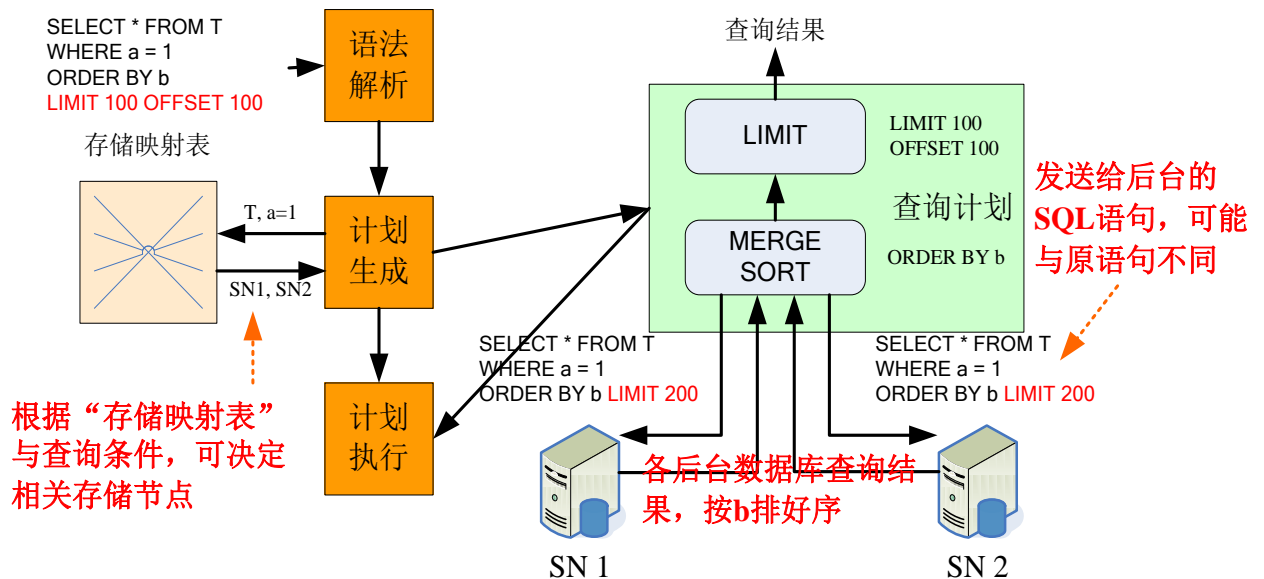


图 5 分布式查询处理的流程

支持全功能的 SQL 查询处理功能是系统是最难的一部分，目前系统支持大部分常用的 SQL 语句，对于一些复杂的查询语句如子查询和多于两表的查询目前还没有支持，不过我们正在不断改进以提供更加全面的查询语句支持。

2.5 系统特征

2.5.1 高效可扩展

分布式数据库的一个重要优势就是在高负载情况下提供稳定的性能表现，并且具备简便的扩展机制。

本系统的高效可扩展性表现在以下几个方面：

- 数据和并发访问的有效均衡。通过将单表数据分布到多个后台数据库节点上一方面可以大大减少每个数据库节点上表记录的数量，另一方面可以将并发的数据库查询分散到各数据库节点上执行，最理想的情况下， n 个数据库节点，每个节点承担 $1/n$ 的查询服务。
- 数据库模式的动态变更。支持在线增加/删除数据库节点、增加/删除均衡策略、增加删除/表、增加/删除/修改表字段、增加/删除表索引、增加/删除 DBI。
- 数据迁移。支持在线和离线数据迁移。
-

2.5.2 数据访问安全性

分布式数据库采用了一些简单的安全控制措施来提高系统的整体安全性。这些措施包括：

- 结合 MySQL 数据库的用户管理功能提供分布式数据库的用户管理和权限控制
- Client DBI、Agent、DBAdmin 访问分布式数据库时都需要用户认证。
- Master 连接 Client DBI 和 Agent 上的服务器时，检查 Master IP 地址的有效性。
- 对分布式数据库用户可以限制允许访问分布式数据库和 DBN 的主机地址。
- 用户名和密码通过网络传递时都进行加密。
- 密码在系统库中存储时进行加密。
- 用户登陆和管理员对分布式数据库的操作内容都记录日志。

2.5.3 数据一致性、完整性和可用性

通过如下机制保证数据的一致性、完整性和可用性：

- 分布式事务处理过程记录日志，保证操作失败时所有数据库节点的数据一致性。
- 通过 DBI 与 Master 的协作处理可能出现的悬挂事务。
- 提供定时数据备份功能，备份整个分布式数据库的数据。结合 MySQL 的 binlog 功能可实现最新数据的恢复。
- 原则上单个节点失效（Master 或 DBN 节点）只影响该节点相关的功能和数据，其他节点仍然可以提供服务。

2.5.4 易维护性

系统专门提供了一个图形界面的管理工具帮助 DBA 进行数据库的日常维护。这些功能包括：

- 数据库模式管理。在线增加/删除数据库节点、增加/删除均衡策略、增加删除/表、增加/删除/修改表字段、增加/删除表索引、增加/删除 DBI。
- 用户管理。增加/删除/修改用户和用户权限。
- 数据迁移。在线迁移和离线迁移，可以定时执行。
- 数据导出。Sql 语句导出和文本数据导出，可定时执行。
- 数据热备份。可定时执行。
- 分区管理。创建和删除分区，支持手动和自动方式，可定时执行。
- 日志和报警管理。对重要操作和事件分门别类记录日志，出现异常情况时向 DBA 进行短信和邮件报警。
- 桶负载统计。查看所有 DBI 上的桶负载情况。

3 分布式数据库系统安装说明

分布式数据库 DDB 的安装相对来说是比较简单的，只需要首先安装好 java 环境、数据库等；然后做一些简单配置后直接运行 DDB 程序即可。整个过程实现了绿色安装，不需要对系统配置做过多修改，这样，配置完系统后移植到其他服务器也就比较方便。

3.1 安装支撑软件

分布式数据库依赖于以下软件：

1. JRE: 分布式数据库需要 1.5 版本的 jre 支持（不支持其他版本的 jre），推荐安装 sun 的 jre 1.5，其它的 jre 未经测试，不推荐使用。
2. MySQL: 在试用时可部署 MySQL 5.0 或 MySQL 5.1 版本，若用于产品环境，由于 MySQL 对分布式事务处理的不足，请务必使用我们修改过的 MySQL，推荐使用 5.0.38 以上版本。

3.1.1 部署 MySQL 的注意事项

1. 对于产品环境，由于需要对 MySQL 进行修改，MySQL 需使用源代码编译的方式安装。目前在发布内容的 doc 目录下包含针对某些 MySQL 版本的 patch 文件，在安装时，请在编译之前使用 patch 命令给源代码打好补丁。
2. 分布式数据库目前主要支持 MySQL 的 innodb 存储引擎（MyISAM 也支持，但没有经过充分测试）。因此在编译 MySQL 时需要确保支持 innodb。在配置 MySQL 服务器时打开对 innodb 的支持。
3. 为了支持更好的支持中文，建议在 my.cnf 中的 [mysqld]，[mysql]，[mysqldump] 的 section 中设置 default-character-set = gbk。
4. 如果需要 redo binlog，同时支持中文，则需要部署 MySQL 5.0.33 以上的版本（之前的版本存在 binlog 多字节编码转义方面的 bug，可能导致 redo binlog 失败）。
5. 将 MySQL 的 innodb 数据文件和 innodb log 文件部署到不同的物理硬盘可以提高约 5% 的写性能。
6. 分布式数据库用户管理功能需要每个 DBN 上的 MySQL 服务器都存在一个默认的系统用户 sys@master_ip，该用户至少需要如下权限：
grant all privileges on DBN.* to 'sys'@'master_ip' identified by 'netease' with grant options;
grant SELECT, INSERT, UPDATE, DELETE on mysql.* to 'sys'@'master_ip' with grant options;
其中 DBN 表示分布式数据库访问的 MySQL 数据库名称。初始的用户口令必须为 netease。
简单起见，可以将通过 grant all privileges on *.* to 'sys'@'master_ip' identified by 'netease' with grant options; 增加系统用户。

在部署完分布式数据库后请及时修改 sys 用户的口令。

3.1.2 MySQL 使用注意事项

提供给分布式数据库使用的 MySQL 在部署和配置时需要注意以下几点：

- 不要使用 MySQL 的 auto increment 功能：MySQL 的 auto increment 功能只能保证在一个节点上生成的 ID 唯一，在分布式数据库环境中一个表可能分布在多个节点上，若使用 MySQL 的 auto increment 功能来生成 ID，则不同节点上生成的 ID 可能

重复。请使用分布式数据库中间件提供的 ID 生成功能，使用方法为调用 `DBCConnection.allocateRecordId` 方法或在插入时指定 `seq` 关键字，然后使用 JDBC 标准的 `getGeneratedKeys` 接口得到中间件为你生成的 ID。

- 不要使用 `begin/commit/rollback` 等 SQL 语句进行事务控制：后果是使中间件对事务当前状态判断错误，从而导致更严重的问题。因此，请务必使用 `Connection` 类提供的接口进行事务控制。
- 不要直接连接到 MySQL 进行数据更新操作：分布式数据库内部使用一个复杂的算法决定记录存储在哪个节点上，若直接连接到 MySQL 进行数据更新操作，如 `INSERT` 一条记录，则可能导致记录存储在错误的节点上，从而无法被应用访问到。
- 主键若为字符串类型，则务必使用大小写敏感的字符串比较：否则可能导致迁移时主键冲突等奇怪问题。使用大小写敏感的字符串比较的方法是在建表时指定使用 `bin` 系统的 `collate`。在使用 `deploy` 工具部署分布式数据库时，将默认使用 `GBK` 字符集和 `gbk_bin collate`，建表语句可不需要修改。

3.2 软件包说明

分布式数据库安装包以 `ddb_XXX.zip` 的形式发布。部署时解压 `ddb_XXX.zip`，将产生一个 `ddb` 目录，该目录包含了运行分布式数据库各部件所需的所有脚本、软件包、配置和说明文件。

`ddb` 目录下包含如下子目录：

子目录或文件	说明
<code>bin</code> 目录	实现分布式数据库各功能模块的 <code>jar</code> 包
<code>lib</code> 目录	所依赖的第三方 <code>jar</code> 包
<code>scripts</code> 目录	运行分布式数据库各种功能的脚本文件
<code>conf</code> 目录	相关配置文件
<code>release_notes</code> 文件	分布式数据库发布版本说明

部署 Master 时，只需要进入到 `scripts` 子目录下，运行相应的脚本。使用分布式数据库 `jdbc` 驱动时，需要在 `classpath` 中指定 `bin` 和 `lib` 目录。

3.2.1 bin 目录

文件	说明	使用场合
<code>common.jar</code>	公共功能模块的 <code>jar</code> 包	DBA 管理工具、 <code>isql</code> 、应用客户端、 <code>master</code>
<code>db.jar</code>	DBI 模块 <code>jar</code> 包	<code>isql</code> 、应用客户端
<code>qs.jar</code>	查询服务器模块 <code>jar</code> 包	查询服务器
<code>exec.jar</code>	通用执行器 <code>jar</code> 包	<code>isql</code>
<code>master.jar</code>	Master 模块 <code>jar</code> 包	<code>master</code>
<code>tool.jar</code>	辅助工具 <code>jar</code> 包	<code>isql</code>
<code>dba.jar</code>	管理工具 <code>jar</code> 包	DBA 管理工具
<code>monitor.jar</code>	实现 <code>monitor</code> 的 <code>jar</code> 包（已停用）	无
<code>benchmark.jar</code>	基准测试程序 <code>jar</code> 包	DDB 内部使用

3.2.2 lib 目录

JAR 包名称	说明
jep-2.4.1.jar	用于查询结果输出
jta-spec1_0_1.jar	用于分布式两阶段提交
log4j-1.2.13.jar	日志打印
netease-commons.jar	网易后台通用支撑包
xstream-1.2.jar	用于 XML 分析
mysql-connector-java-5.0.8-bin.jar	MySQL 数据库客户端 jar 包
commons-cli-1.0.jar	DDB 系统命令行工具中用于命令行参数分析
derby_10.3.2.1.jar	用于执行计划分析
libreadline-java-0.8.0.jar	DDB 系统命令行工具中用于实现上下文键盘助手
jfreechart-1.0.6.jar	Benchmark 框架使用的图形界面包
jython.jar	Jython 语言包
mail.jar	邮件发送工具包
OpenSwing.jar	DBA 管理工具界面包

3.2.3 scripts 目录

文件	说明
master.sh/master.bat	启动 Master 的脚本
killmaster.sh	通过 kill 命令强行关闭 Master 进程的脚本
msctrl	用来启动/关闭/重启 Master 的脚本，通过客户端请求安全关闭 Master
master	Linux 上以系统服务方式启动 Master 的脚本
agent.sh	启动 Agent 的脚本
killagent.sh	关闭 Agent 的脚本
agent	Linux 上以系统服务方式启动 Agent 的脚本
dbadmin.sh/dbadmin.bat	DBA 图形化管理工具的启动脚本

monitor.sh	启动 Monitor 的脚本
killmonitor.sh	关闭 Monitor 的脚本
setclasspath.sh/.bat	设置 Classpath 的脚本，被其他脚本调用
isql.sh/isql.bat	启动 isql 工具的脚本
qs.sh	启动查询服务器的脚本
showver.sh/showver.bat	查看分布式数据库版本信息的脚本
crypt.sh/bat	用于产生密钥文件的脚本
evn 目录	网易博客环境使用的配置信息
loadbalance 目录	负载均衡、在线迁移所使用的脚本
onlineAlterTable	在线修改表结构模块所使用的脚本

3.2.4 conf 目录

文件	说明
create_sysdb.sql	系统库的完整建表脚本
update_sysdb.sql	升级系统库表定义的系统库更新脚本
migsysdb.sh	用于将 ddb1.0 的系统库升级到 ddb2.0
DBClusterConf.xml	Master 配置文件
SysAgent.cnf	Agent 的配置文件
ddb.conf	Master 和 Agent 作为系统服务的配置文件
secret.key	默认的密钥文件
lw.conf	轻量级中间件的配置文件，用于配置查询服务器
monitor.cnf	Monitor 的配置文件
log4j.properties	Log4j 配置文件

3.3 部署 Master

Master 是分布式数据库的核心控制部件，需要首先部署。部署 Master 的一般步骤为：部署软件包，部署系统库，配置 Master，启动 Master。

3.3.1 部署软件包

部署软件包只需要将上面列表中的文件拷贝到指定的安装目录下，假定为 basedir 目录。

3.3.2 部署系统库

Master 上的系统库用来持久化整个分布式数据库的配置信息。系统库采用 MySQL Innodb 存储引擎。首先需要安装 MySQL 数据库，通常是安装到 Master 服务器上，也可以选择安装在 Master 服务器以外的服务器上。安装后启动 MySQL 并创建系统数据库，默认字符集为 GBK，系统库名称可以自行定义，通常为 sysdb 或 xxx_sysdb。这里我们认为是 sysdb，建库命令为：

```
mysql> create database sysdb default character set=gbk;
```

创建完系统库后需要建表，可通过执行

```
mysql> source /basedir/conf/create_sysdb.sql
```

3.3.3 配置 Master

系统库部署完毕后需要修改 `basedir/conf/DBClusterConf.xml` 文件对 Master 进行配置。其中必须要修改的条目有：

`<ip>`：用于指定 Master 服务器绑定的 IP 地址，Client 通过该地址连接分布式数据库

`<sysdb_url>`：Master 访问系统库的 url

`<sysdb_user>`：Master 访问系统库的用户名

`<sysdb_password>`：Master 访问系统库的口令

其他选项都可以采用默认值。

具体 master 配置文件的各参数含义参见附录说明。

3.3.4 启动 Master

在这之后就可以进入到 `basedir/scripts` 目录下运行 `master.sh` 或 `master.bat` 来启动 Master 了。

Master 正常启动后会在该目录下的 `master.log` 中打印：

“开始启动分布式数据库管理服务...

管理服务启动成功，等待连接请求.....”

注意：默认情况下 `basedir/scripts` 目录下的脚本只能在该当前路径是该目录的情况下运行，如果需要在其他目录下运行，需要在该脚本中设置 `setclasspath.sh/bat` 脚本的路径，以及 `setclasspath.sh/bat` 中 `BASEDIR` 路径。

3.3.5 部署 Master 服务

在产品环境中，为实现机器重新启动后自动启动 Master，可将 Master 部署为一个系统服务。当然这个步骤不是必须的。部署方法如下。

部署并配置 Master 服务配置文件——`ddb.conf`

首先，将发布目录下的 `basedir/conf/ddb.conf` 拷贝到 `/etc/netease/backend` 目录下，并根据需要修改参数设置。该文件中各配置含义如下：

参数	默认值	说明
<code>VERBOSE</code>	<code>YES</code>	启动/关闭 Master 服务时是否打印一些调试信息
<code>JAVA_HOME</code>		Jre 安装目录，目的是为了找到 java 可执行程序，若 java 已经在 <code>PATH</code> 中，则不需要设置本参数
<code>MASTER_OPTS</code>		传递给 Master 的参数，请见后续说明
<code>MASTERROOT</code>	<code>/home/ddb</code>	DDB 发布内容根目录 <code>basedir</code> 。

部署并配置 Master 服务启动脚本——`master`

第二步是将 Master 服务启动脚本拷贝到 `/etc/init.d` 目录，并在在启动 Master 服务的 `runlevel` 目录中做好链接。如在一般情况下需要在 `runlevel` 为 2 时启动 Master 服务，则在 `/etc/rc2.d` 目录（有些 Linux 发行版可能是 `/etc/init.d/rc2.d`）下执行：

```
ln -s /etc/init.d/master S99master
```

这样就可以通过 `/etc/init.d/master start` 和 `/etc/init.d/master stop` 命令来启动和关闭 Master 了。当然这需要 root 权限。

3.4 部署 DBAdmin 管理工具

DBAdmin 是一个图形化的客户端管理工具，在 DBA 的本地客户端上基于 Java 虚拟机运行。部署和运行 DBAdmin 很简单，运行 `basedir/scripts/dbadmin.sh` 或 `dbadmin.bat`。通过 DBAdmin 可完成对分布式数据库的所有管理功能。

3.5 部署客户端应用程序

部署客户端应用程序也是比较简单的，分为两种：应用程序和复制客户端工具 `isql`。应用程序即用于连接 DDB 的 java 程序，由于一般内嵌于 Tomcat 等服务器中，所以只要在服务器的 `classpath` 路径中把 DDB 所要用的包都加上就可以；如果是 `isql` 工具，可以运行 `basedir/scripts/isql.sh` 或 `isql.bat`。

3.6 部署主机监控报警工具 Monitor

3.6.1 主机监视报警工具简述

主机监视报警工具（Monitor）用于监视网络上任何一台或多台主机的运行状况，在主机发生故障时可自动向管理员发送电子邮件和手机短信报警。该工具可以被部署在网络上任何一个或多个主机上，但运行 Monitor 的主机和被监视主机最好不要为同一台主机。

3.6.2 Monitor 实现原理

Monitor 采用 Java 实现，实现原理非常简单：对于每个需要被监视的主机，Monitor 都将单独启动一个线程定期向该主机指定的端口发送监视命令字。如果无法与该主机建立连接或该主机返回故障状态，Monitor 都将通过电子邮件和短信向管理员报警。

对于部署在 ZooKeeper 上的 DDB，Monitor 会同时启动对 ZooKeeper 上该 DDB 的 master 列表进行监听。如果该 DDB 有新 master 加入、master 断开或 Leader 切换等事件发生，都会通过电子邮件和短信向管理员发送通知。

3.6.3 被监视主机与 Monitor 的通信接口

如果主机需要被 Monitor 监视，需要实现如下功能：

1. 在指定的端口上监听来自 Monitor 客户端的连接请求
2. 接收来自 Monitor 的监控命令字“CMD_MON”字符串
3. 检查自身运行状态
4. 如果运行正常，则向 Monitor 返回“STA_NOR”字符串（不包含‘\0’结束符）
5. 如果运行异常，需要向管理员报警，则先向 Monitor 返回“STA_ABN”字符串（不包含‘\0’结束符），然后按照顺序发送如下消息内容：

消息内容	类型	取值范围	描述
1 或 2	Byte	1	报警级别，1：表示需要同时发送报警邮件和报警手机短信；2：表示只需要发送报警邮件
故障描述字符串长度 length	int	>=0	注：使用网络字节序
故障描述字符串	Byte[]		数组长度由 length 指定，中文采用 GBK 编码

3.6.4 Monitor 的部署

需要将 mail.jar, activation.jar, log4j-1.2.13.jar 和 monitor.jar 部署到 CLASSPATH 中, 如果要监控部署在 ZooKeeper 上的 DDB, 则还需要 common.jar 和 zookeeper-3.3.3.jar。通过 conf/monitor.cnf 配置相关参数

配置项	说明
name=monitor1	监视器名称, 报警时将显示监视器名称。(可选)
interval=5000	默认的监视命令字发送时间间隔, 如果没有针对主机明确指定时间间隔, 将采用该参数值(可选)
smtp_server=smtp.126.com	SmtP邮件服务器地址(发邮件必需)
mail_to_list=aaa@126.com;bbb@163.com	收件人地址列表, 地址之间用分号间隔(发邮件必需)
mail_cc_list=aaa@126.com;bbb@163.com	抄送地址列表, 地址之间用分号间隔(可选)
mail_sender = master_test@126.com	发件人地址(发邮件必需)
mail_pass = neteasetest	发送邮件需要的口令(发邮件必需)
sms_url=http://sendmsg.reg.163.com:16598/servlet/payment.user.DBSendMessage?SmsGroup=Db59437	提供发送手机短信服务的url(发送短信必需)。
mobile=13533223323;13965674567	管理员手机号码列表, 号码之间用分号间隔(发送短信必需)。
host=ur1;ur2;ur3 或采用如下形式: host=ur1 host=ur2	url有两种形式。普通形式为: ip:port[:interval] 被监视主机参数列表, 每个主机的相关参数包括: 地址, 监听端口, 监视间隔时间(单位毫秒)通过冒号间隔。 zookeeper形式: zookeeper://ip:port[,ip:port]/ddbname:interval[,ddbname:interval]。一个zookeeper式url可以同时指定多个DDB。

说明: 目前短信发送服务由通行证系统提供, 使用前首先需要申请向相关部门提出申请, 注册使用该服务主机的 ip 地址。目前只支持向移动用户发送短信, 每个 ip 地址一天内最多允许发送 200 条短信。可以通过执行 `java -classpath .:monitor.jar:lib/mail.jar:lib/activation.jar:lib/log4j-1.2.13.jar:lib/common.jar:lib/zookeeper-3.3.3.jar com.netease.backend.monitor.Monitor [config file path]` 启动 monitor。默认情况下将从 conf/monitor.cnf 中读取配置信息, 也可以在命令行最后指定配置文件的路径。

3.7 系统建立过程示例

本章以一个示例配置演示建立分布式数据库系统的过程。为简单起见, 本章使用一个最小化配置, 即只使用两个数据库节点和一个应用节点, 且所有组件部署于一台机器上, 同时不包含 Agent 和 Monitor 工具的配置。示例应用的模式定义在 conf/demo.sql 中。

3.7.1 初始化系统库

系统库的地址在 `conf/DBClusterConf.xml` 中指定, 设为 `sysdb`。使用 `mysql` 客户端运行 `conf/create sysdb.sql` 即可完成系统库的初始化, 如下:

```
$ mysql -u root
mysql> source conf/create sysdb.sql
```

3.7.2 启动 MASTER

在初始化好系统库后，就可以启动 MASTER 了，进入到 script 目录，运行：

```
$ ./master.sh &
```

即可。

3.7.3 创建示例应用数据库模式

按上面步骤初始化好的分布式数据库环境已经是可用的了，但只是一个全空的分布式数据库，没有数据库节点，没有任何均衡策略和表定义，下面来创建示例应用所用的数据库模式。

示例应用将包含两个数据库节点 **db1** 和 **db2**，在将数据库节点加入到分布式数据库之前，首先要使用 **mysql** 工具创建好数据库并创建用户 **sys** 和密码为 **netease** 的系统超级用户，如下：

```
$ mysql -u root
mysql> create database db1;
mysql> create database db2;
mysql> grant all privileges on mysql.* to sys@master_host identified by 'netease'
with grant option;
mysql> grant all privileges on db1.* to sys@master_host with grant option;
mysql> grant all privileges on db2.* to sys@master_host with grant option;
完成上述步骤后，使用 isql 工具运行 demo.sql 即可完成模式定义工作：
$ ./isql.sh -p netease -P netease
isql@dbi> source demo.sql;
```

3.7.4 操作示例数据库

我们的示例应用模拟了一个学生选课系统，包含 **Course**、**Student** 和 **Elect** 这三张表，在 **demo.sql** 中同时还插入了一些示例数据。现在可以使用 **isql** 来操作一下了：

看看系统中有哪些表:

```
isql@dbi>> show tables ;
```

NAME	TYPE	POLICY	BF	PKEY
Elect	INNODB	user	sid	SIId,CourseCode
Student	INNODB	user	id	Id
Course	MYISAM	course	code	Code

看看有哪些学生:

```
isql@dbi>> select * from Student;
```

id	name	birthday	gender
----	------	----------	--------


```

+---+-----+-----+-----+
| 1 | Michael | 1984-04-23 | M   |
| 2 | John   | 1985-07-17 | F   |
+---+-----+-----+-----+

```

看看 John 选了哪些课，成绩是多少：

```

isql@dbi>> select Student.name, Course.name, score from Student, Elect, Course
where Student.name = 'John' and sid = id and coursecode = code;
+-----+-----+-----+
| name | name          | score |
+-----+-----+-----+
| John | Algorithms    | 91    |
| John | Operating Systems | 88    |
+-----+-----+-----+

```

4 管理工具 DBA

DBA 管理工具，是网易分布式数据库协助管理、维护和监控分布式数据库的用户图形界面工具。通过 DBA 管理工具，系统管理员可以有效地构建整个分布式数据库系统、维护数据以及监控系统运行状态。

DBA 管理工具的主界面如图所示。



DBA 管理工具启动界面

DBA 管理工具启动后，通过菜单“系统->连接”连接到一个带有管理功能的 Master，就可以对分布式数据库系统进行管理。界面主要功能如下：

1. 系统菜单：启动/关闭 **Master**；配置系统参数；定期清理系统库；计划任务；查看日志；启动 **isql** 界面工具。
2. 系统配置：管理系统配置信息。
3. 数据迁移：执行数据迁移操作；制定数据迁移计划。
4. 数据备份：执行数据备份；制定数据备份计划。
5. 数据导出：执行数据导出；制定数据导出计划。
6. 负载：查看/统计负载信息。
7. 悬挂事务：查看处理上报的悬挂事务。
8. 警报：查看报警信息。
9. **Client** 统计：查看各 **Client** 上的资源信息。
10. 监控统计任务：管理监控统计任务。
11. 统计与分析：针对完成的监控统计结果，进行汇总和分析。

4.1 通过 SSH Tunnel 功能连接到管理服务器

为保证安全性，在产品环境中推荐通过 SSH 提供的 **tunnel**（又称 **port forwarding**）来连接到管理服务器。通过 **SSH tunnel**，可以将远程 **Master** 服务器上用于接受 **DBA** 管理工具连接的端口（**7777**）映射为本地机器的某端口，设为 **P**。建立 **SSH tunnel** 之后，即可在本地启动 **DBA** 管理工具，连接到 **localhost** 的 **P** 端口，即可达到直接连接到远程 **Master** 服务器 **7777** 端口相同的效果。

使用这一模式的优点在于在 **Master** 服务器上只需要开放 **ssh** 服务端口，不需要开放 **7777** 端口。同时 **DBA** 管理工具与 **Master** 之间的通信都自然受 **ssh** 加密通道的保护。相对于在 **Master** 服务器上开放 **7777** 端口，**DBA** 工具直接连接到该端口的方式安全性更高。

不同平台下通过 **SSH tunnel** 连接管理服务器的方法如下：

Linux 平台下使用 **ssh tunnel**

首先建立 **ssh tunnel**（当然，首先在安装好 **ssh**，设置好公钥认证体系啦），一般可使用如下命令：

```
ssh -p <Master 服务器 ssh 服务端口> -L <本地映射端口> :localhost :7777  
<用户名>@<Master 服务器 IP>
```

其中可能比较 **confusing** 的是“<本地映射端口> :localhost :7777”这一段，这里的 **localhost** 并不是指本地机器（即运行 **DBA** 管理工具的机器），而是指对于远程 **Master** 来说是 **localhost**（即 **Master** 机器本身啦）。

运行这一命令，使用 **ssh** 登陆成功后，就可以在 **DBA** 管理工具里连接 **localhost**，端口指定为上述映射的“本地映射端口”，就可以连接上远程的 **Master** 服务器了。

Windows 平台/SSH Secure Shell 中使用 **ssh tunnel**

Windows 平台下有很多 **SSH** 工具，不同的工具用法不用。这里以使用较多的 **SSH Secure Shell** 为例说明如何在 Windows 平台下使用 **ssh tunnel**。

使用 **SSH Secure Shell** 配置 **SSH tunnel** 的步骤如下：

1. 选择“Profiles→Edit Profiles...”，打开“Profiles”对话框
2. 在“Profiles”对话框选中要使用 **tunnel** 的 **profile**
3. 选中“Tunneling”选项卡，点击“Add...”，打开“Add New Ongoing Tunnel”对话框
4. 在“Add New Ongoing Tunnel”对话框内输入以下信息：
 - 1) **Display**：随便取个名字，用来看看而已

- 2) **Type**: 选中 TCP
- 3) **Listen**: 这里输入本地映射端口, 如 7777
- 4) **Destination** (上一个): 输入映射目的服务器地址, 如 localhost (上面已经说过了, 这个是针对远程机器而言的)
- 5) **Destination** (下一个): 输入 7777
5. OK 保存配置

然后, 使用上述已经配置过的 **Profile** 连接到远程 **Master** 服务器, 即可实现 **ssh tunnel**。而后就可以在 **DBA** 管理工具里连接 **localhost**, 端口指定为上述映射的“本地映射端口”连接到 **Master**。

使用网关机中转时的连接方法

一种更为安全的做法是将 **Master** 服务器部署在内网, 从外部连接 **Master** 时必须通过一个网关机中转, 即先 **ssh** 到网关机, 再从网关机上 **ssh** 到 **Master** 服务器。这种模式也是可以使用 **ssh tunnel** 的。

这又有两种方式。

方法一: 在 Master 服务器上开放 7777 端口, 绑定在内网地址上, 并允许在网关机上访问。

这时只需要修改 **ssh tunnel** 的目标机 IP 就可以了, 如在 **Linux** 下可采用如下命令连接到网关机建立 **ssh tunnel**:

```
ssh -p <网关机 ssh 服务端口> -L <本地映射端口> :<Master 服务器内网地址>:7777
<用户名>@<网关机 IP>
```

在 **SSH Secure Shell** 中只需在“Add New Ongoing Tunnel”对话框中修改一下“Destination”即可。

方法二: Master 不开放 7777 端口

这可以通过两步 **ssh tunnel** 实现, 首先在网关机上建立 **ssh tunnel**, 将网关机上的端口 A 映射到 **Master** 的 7777 端口, 再建立本地机器的端口 B 与网关机的端口 A 之间的 **ssh tunnel**。两个 **tunnel** 都建立完成后, 就可以使用 **DBA** 工具连接本地机器的 B 端口来连接到 **Master**。

4.2 通用选项及操作

在说明管理工具各个操作之前, 先要了解管理操作中的一些选项, 这些选项将会影响到众多操作的行为。

1. 忽略通知 **Client** 失败的异常: 管理操作需要 **Master** 通知系统中所有未处于关闭状态的 **Client** 更新配置或操作, 当存在 **Client** 被通知失败, 可使用此选项忽略异常而继续执行下去。
2. 忽略无法连接的 **Client**: 管理操作需要 **Master** 通知系统中所有未处于关闭状态的 **Client** 更新配置或操作, 当存在 **Client** 无法被连接的 **Client** 的时候, 可使用此选项而继续执行下去。
3. 是否在 **DBN** 上执行: 数据库表的创建、修改等操作是否同时在系统中的物理 **DBN** 数据库上操作。若物理 **DBN** 数据库上已经存在相应的数据库表等配置, 可不选此选项而直接在 **Master** 中修改配置, 但需确保 **Master** 和 **DBN** 两端的配置一致。所有涉及表视图的操作都含有此选项。
4. 拒绝用户对表的写操作: 管理操作时所有 **Client** 对指定表的写操作将被禁止。在增删改字段的时候含有此选项。

注意：“忽略通知 Client 失败的异常”包含需选项“忽略无法连接的 Client”。若忽略 Client，必须确保这些 Client 处于关闭状态或者相应操作不会对 Client 造成影响，否则可能导致 Client 和 Master 的数据不一致。

在部分表格结果数据中，有两个通用操作：过滤和分组。

1. 过滤：根据 column 表达式可以对结果进行过滤，支持 <、<=、==、>、>=、!= 条件，以及 &&（与）和 ||（或）连接符。
2. 分组：根据 column 对结果分组，分组后只显示每个组的 count 和用户分组的 column。多个 column 名之间用逗号分隔。
3. 条件为空的时候，任意点击过滤或分组，显示原结果。

4.3 系统菜单

“系统”菜单（Alt-S），主要有以下命令：

1. 连接（Alt-C）：连接到管理服务器，DBA 只有先连接到服务器之后才能做各种管理操作。
2. 启动 MS（Alt-S）：启动 Master，可以让 Master 重新读取系统库中的配置。在 Master 关闭状态下，此功能生效。
3. 关闭 MS（Alt-U）：关闭 Master，在 Master 运行状态下有效。若修改了 Master 的部分系统配置而需要重启 Master，则可以在管理工具中先关闭 MS，再启动 MS 就可以了。
4. 系统参数配置（Alt-P）：可以动态修改部分系统配置参数，主要有 DBN 心跳设置、Client 设置、报警设置、缓存设置(详见)和其它部分设置。所有可配置的参数信息可参见《网易_分布式数据库_安装部署说明.doc》附录。[脚本报警方式](#)细节可见调用脚本进行报警。
5. 系统库定期清理（Alt-R）：可制定系统库中部分表的定期清理任务，可清理的表包括：警报表（alarm）、节点 DBN 负载表（dbn）、分支事务表（xa）。[详见计划任务](#)。
6. 计划任务（Alt-T）：可查看已制定的所有计划任务；可以跳转到具体任务类型界面进行管理。[详见计划任务](#)。
7. 日志查看（Alt-L）：可以查看管理服务器端生成的日志。
8. 启动 ISql（Alt-I）：启动 isql 界面工具，默认使用当前登录用户连接 Master，详见《网易_分布式数据库_辅助工具使用说明.doc》。
9. 退出（Alt-X）：退出 DBA 工具。

4.4 系统配置

DBA 主界面是多个 Tab 页，在系统配置页（Alt-1），可以对部分系统配置进行动态修改。

4.4.1 管理 Client

点击“查看 Client”（Alt-C）按钮，可以查看系统中的 Client 信息，右侧底部按钮命令如下：

1. 刷新（Alt-F）：从 Master 重新读取全部 Client 信息。当 Client 状态有变化时需要手工刷新才能更新显示。
2. 删除（Alt-D）：删除表格中所选的 Client。删除后，Master 所有操作都不会通知此 Client，请确认此 Client 已处于关闭状态。
3. 停止服务：使得 Client 处于停止服务状态，Client 停止服务后，不影响已经开始的查询和事务，但不允许开启新的事务和事务外的查询。
4. 启动服务：使得处于停止服务状态的 Client 重新运行，可以开启新的事务及查询。

5. 设置为关闭 (Alt-O)：使得 Client 处于关闭状态。Master 所有管理操作都不会通知关闭状态的 Client。
6. 连通测试 (Alt-K)：测试指定 Client 或者全部 Client 是否与 Master 连通。
7. 配置 (Alt-N)：设置各个运行状态 Client 的配置参数，包括：最大连接数、缓存 PST 数目、连接池大小等参数配置，以及是否打开日志记录，是否打开只读事务日志记录，是否打开 auto commit 模式的语句的记录功能。各个参数的意义与“系统”菜单中“系统参数配置”相同，但此处设置的参数是指定 Client 特有的，优先级大于系统全局参数。日志记录，是指 Client 会用日志记录执行的语句，可设置不记录只读事务类型的语句及 auto commit 模式的语句。
8. 全局配置 (Alt-G)：强制所有 Client 从 Master 读取并更新表、策略等全局配置信息。当存在 Client 的配置信息与 Master 不一致而影响应用时，可使用此功能。
9. 统计 (Alt-A)：跳至 Client 统计界面。当前选中的 Client 会自动添加为 Client 统计界面中的已选 Client 列表。

提示：原先通过增加指定 Client 以使该 Client 能够访问 Master 的方式已取消，统一增加用户管理，通过用户的 IP 限制来管理 Client 的访问权限。

4.4.2 管理 DBN

系统配置页面，通过左侧“查看 DBN” (Alt-B) 按钮，管理员可以对数据库节点 DBN 进行管理：

1. 刷新 (Alt-F)：从 Master 重新读取所有 DBN 信息。双击列表中一个 DBN 记录，可以查看该节点上的进程 (show processlist)。
2. 关闭 DBN (Alt-U)：关闭一个 DBN 节点，停止其 MySQL 数据库服务，可输入关闭理由。**需要 DBN Agent 的支持。**
3. 启动 DBN (Alt-T)：启动一个 DBN 节点，启动其 MySQL 数据库服务。**需要 DBN Agent 的支持。**
4. 增加 DBN (Alt-A)：增加一个 DBN 节点，分为 MySQL 节点和 Oracle 节点两种类型，均需输入 IP、Port、DBURL 等相关信息，以及一些各数据库类型特有的信息。增加 DBN 时候，若选中在新增的 DBN 上创建系统中已存在用户的选项，则系统会自动在新增 DBN 上创建系统中的用户，在新 DBN 上创建用户的管理用户当然需要有在新 DBN 上创建用户的权限（默认 sys 用户或者手工输入的用户）。Ssh 用户名和端口，是开放给 master 做并发离线迁移用的。对于 MySQL 节点，可以指定自己所从属的主节点；而对于 Oracle 节点，则需要指定节点的 schema 以及表空间。
5. 删除 DBN (Alt-D)：删除选中的 DBN 节点。若 DBN 节点还在被策略使用，不允许删除。
6. 修改 DBN (Alt-M)：修改指定的 DBN 节点信息。
7. 禁用 DBN：设置 DBN 状态为禁用，中间件在分析查询需要访问的 dbn 时若发现某 dbn 已经被禁用则直接抛出异常，为了避免由于某些 dbn 负载过高或锁等待严重导致大量分布式事务被阻塞，分布式数据库整体性能下降。
8. 启用 DBN：重新启用被禁用的 DBN。
9. 测试连接：测试 DBN 上的服务器是否可以接受连接。

从 DDB3.0 开始，提供对 slave 节点的管理功能。在添加一个 slave 节点时，需要额外设置以下几个参数：

1. 从属于节点：用于指定该 slave 节点下挂在哪个 master 节点下。
2. 权重：用于指定在做负载均衡时，该节点能够分配到的查询权重，默认为 1。
3. 不复制的表：可以特别指定一些表，不参与 mysql 的复制。

DDB 提供了一套相对完善的 slave 节点管理机制，可用于重建 slave、启/停 mysql 复制、主从切换等。在做复制管理操作前，请先确定以下准备工作是否完成：

1. 正确设置 DBN 节点的相关参数，包括：ssh 登陆用户名、ssh 登陆端口、mysql 配置文件路径。
2. 正确设置系统参数配置中 DBN 设置页面里的参数，包括：DBN 脚本路径、临时数据目录、ibbackup 目录、innobackup 目录、出错日志、镜像库目录、mysql 本地登录超级用户名、密码、mysql 复制用户名、密码。具体每个参数的含义及说明参见《分布式数据库_管理员手册》附录。
3. 确认 ssh 用户能够正常登录到 dbn 节点。
4. 确认 dbn 节点机器上，各个脚本是否已经存在，并分配了可执行权限。
5. 做建立镜像操作时，从 master 节点导出的临时数据将存放在：临时数据目录（系统参数中配置）+ “/” + master 节点名称（DBN 参数中配置）。因此需要额外确认临时目录是否已存在。

slave 节点管理的目的使得 DBA 管理员能够更简便的对 slave 节点进行日常管理操作，提供的功能包括：

1. 启动复制。启动 slave 节点的 mysql 复制，若选中“是否修改数据库配置使其永久生效”，则会同时修改 mysql 配置文件，使得 mysql 重启之后所做的改动仍有效。
2. 停止复制。停止 slave 节点的 mysql 复制，若选中“是否修改数据库配置使其永久生效”，则会同时修改 mysql 配置文件，使得 mysql 重启之后所做的改动仍有效。
3. 建立镜像。自动对 slave 节点进行重建操作。在做重建操作时，管理员可以指定数据库目录。这样，新的 slave 数据库将存放在指定的目录中。系统将按照下面的策略来决定数据库存放目录：若 mysql 配制文件已存在，则读取配制文件中的目录参数作为实际存放目录；若配置文件不存在，则查看管理员是否指定了存放路径。若没有指定存放路径，则系统把新 slave 数据库存放在：镜像库目录(系统参数中配置)+“/”+ DDB 名称（DBClusterConf.xml 中配置）+ “_” + slave 节点名（DBN 参数中配置）
4. 查看复制状态。列出 slave 节点的复制状态，功能与 show slave status\G 相同
5. 修改复制参数。

Master_User — 用于 mysql 复制的用户名。

Master_Password — 用于 mysql 复制的密码。

Master_Host — 指向的 master 数据库 ip 地址。

Master_Port — 指向的 master 数据库端口。

Master_Log_File — master 数据库当前日志文件名。

Master_Log_Pos — master 数据库当前日志位置。

其中，Master_User 与 Master_Password 作为一组参数必须同时指定。若 Master_User 为空则不做修改操作。Master_Host、Master_Port、Master_Log_File、Master_Log_Pos 作为另一组参数也必须同时指定。若 Master_Host 为空则不做修改操作，若 Master_Host 不为空则其他三个参数也必须同时被指定。

6. 设置自动切换。这会把指定的从节点设置为自动切换节点，即当该从节点对应的主节点失效之后，DDB 会自动将该从节点切换为主节点并接受读写访问，如果存在其他从节点，则其他从节点也会将复制的主节点切换至新的主节点。该功能限制一个主节点只能存在一个自动切换节点，且要求必须在系统参数中配置“MySQL 复制用户名”及“复制用户密码”。当自动进行切换过程中发生异常或者某从节点复制状态不正常等，自动切换会取消。自动切换成功或者失败后，都会取消当前的自动切换配置，并发送报警说明切换结果。
7. 切换主从节点。需要同时选中一个 master 节点和一个 slave 节点，操作完成之后，原来的 slave 将变成 master 节点，原来的 master 节点将变成 slave 节点。与此同时，原来 master 下挂的其他 slave 节点将会自动指向新的 master。进行主从切换时需要指定切换等待时间。该时间用来等待 master 与 slave 节点同步，若指定时间内无法完成同步，则返回失败。

注：其中，启动复制、停止复制、建立镜像三个操作可以同时选中多个 slave 后并发执行。

4.4.3 管理策略

系统配置页面，通过“查看策略”（Alt-P）进行对策略的管理：

1. 刷新（Alt-F）：刷新显示系统中所有策略。提示：**双击列表中一个策略可以显示该策略含有的桶信息。**
2. 增加（Alt-A）：动态增加一个策略，分为 MySQL 和 Oracle 两种类型，一个策略下只能包含一种类型的节点，不能混合。策略桶分布方式有 RR（如 121212）、SR（如 111222）、Man（手工指定）三种，还可以指定桶自动分布所在的 DBN 节点以及桶数目。
3. 删除（Alt-D）：动态删除一个策略。只有在该策略下没有表存在时，才能将它删除。
4. 增加 bucketno（Alt-O）：添加表的 bucketno 字段，或者添加该策略下所有表的 bucketno 字段。可以保存为计划任务执行。拒绝用户对表的写操作，指在对表增加 bucketno 字段的时候，Client 所有对表的写请求将被拒绝。
5. 自定义均衡函数（Alt-H）：除默认的两个均衡函数 DBHash 以及 DBStringHash 外，用户可以通过填写 Java 代码创建满足特定数据分布要求的自定义均衡函数。
 - 1) 增加：创建自定义均衡函数。类名首字母必须大写，输入参数类型支持 long、String 以及多字段均衡的 List<Object>，界面上选择的参数类型须与源码中实现的 hash 函数的参数类型一致。源码输入框中只需要填写 hash 函数的具体实现，其他的类结构代码会自动补充完整。对于 List<Object>的参数类型，DDB 会确保 Object 的基本类型只会是 Long 或者 String。
 - i. 提交编译：提交源码到 master 并由 master 调用 Java 编译器进行编译。
 - ii. 测试：对最新提交编译成功的自定义均衡函数进行测试（参数为 List<Object>的函数不能进行测试）。可以手工输入测试数据或采用自动生成，测试结果中会说明测试值与桶之间的对应关系。**提示：测试结果列表支持双击，可以在弹出窗口看到所选择桶的完整信息。**
 - iii. 保存：将最新提交编译成功的自定义均衡函数保存到系统库。
 - 2) 修改：修改指定的自定义均衡函数的实现代码。**如果该均衡函数已经被均衡策略使用，极有可能导致已存在数据无法访问。**具体操作和“增加”类似。
 - 3) 删除：删除指定的自定义均衡函数。如果该函数已经有均衡策略使用，则不允许删除。

4.4.4 管理表

系统配置页面，通过“查看表”（Alt-T），对数据库表进行管理：

1. 刷新（Alt-F）：刷新显示系统中所有数据库表。提示：**双击列表中一个表可以显示该表的详细信息，可修改表分配的起始 id 和剩余 id。**
2. 表操作
 - 1) 增加（Alt-A）：动态增加一个数据库表。若指定在 DBN 节点上建表，则需要输入建表语句。**注意：**建表语句需要指明必要的字段长度，否则在 DBN 上建表会出错。这里划分为两个标签，分别对应 MySQL 表和 Oracle 表。对于 Oracle 的聚簇表，在选定聚簇时，表所属的策略则必须与聚簇相同。
 - 2) 删除（Alt-D）：动态删除一个表。
 - 3) 修改字段索引：执行 ALTER TABLE 操作，具体可以看下一节。
 - 4) 修改表属性：可以修改表所属模块，修改 ID 相关参数，修改表的均衡策略。
 - 5) 查看 NTSE 属性：查看所选择的 NTSE 表的配置信息。
 - 6) 表记录缓存设置：管理指定表的记录缓存配置。“升级缓存版本号”通过升级表的缓存版本号从而失效表的全部缓存数据，达到强制刷新的效果。缓存 key 字段用于指定表的记录在缓存中的 key 组成，限制为唯一索引字段。如果不指定缓存 key 字段则由于无法生成 key 值而不能缓存该表的数据。字段被指定为表的缓存 key 字段后，字段以及相关关联的唯一索引不能删除，可以先修改缓存配置然后再删除。

“是否缓存表记录”表示是否主动缓存表记录，需指定缓存 **key** 字段后才能开启。修改表记录缓存配置后，可选择是否在修改完成后升级表的缓存版本号。

- 7) 查看在线修改表结构任务：进入在线修改表结构任务的管理界面，详见 [4.4.5 在线修改表结构](#)。

3. 对于自增字段的支持

只支持 MySQL 数据库类型，与 MySQL 语法一致，可以在 CREATE 或 ALTER 操作中设定字段是否自增，每张表只能存在一个自增字段。不允许设定 AUTO_INCREMENT 的起始值。

最后下发到 MySQL 节点上执行的 sql 语句中，AUTO_INCREMENT 关键字都会被去除。

4. 修改字段索引：

1) 支持的操作

字段的增删改、索引的增删、表重命名、表和字段注释的修改。对于 MySQL 表，还支持对表选项信息的修改，例如压缩表选项、表默认字符集等。

2) 提供了两种操作入口，接受标准 SQL 语句的输入框以及简单的鼠标操作输入。

注意：字段删除只能通过鼠标操作，以及修改 MySQL 表选项只能通过 SQL 输入框，其他 ALTER TABLE 操作都可以通过两种操作方式实现。

3) 操作执行：

输入 SQL 语句或者通过鼠标操作完成后，有三种方式让修改生效。

“确定”：该操作会根据设定直接将命令发送到 master，交由 master 到各节点上执行。

“存为任务”：该操作将会存为一个计划任务，可以在弹出的界面中设定启动时间等。到了指定的时间，master 会启动该任务并到各节点执行。计划任务相关信息可以查看 [4.11 计划任务](#)。

“存为在线修改任务”：该操作将会保存为一个在线修改表结构任务。在弹出的界面中，可以设定在线改表任务的相关配置信息。详见 [4.4.5 在线修改表结构](#)。

5. 批量更新数据：批量更新数据的原理是把一条 SQL 所需要更新的记录按照某个基准索引进行分段。分段操作的好处是可以避免发生大量锁等待超时，并且避免单条 sql 语句大批量更新数据之后发生的事务标记清除操作。

用户设定的参数有：

- 1) 进行批量更新操作时的遍历基准索引名称。依据该索引进行数据分段。
- 2) 索引开始 (\geq)，为整型，默认 0。会依据选取的基准索引不同而动态修改，如果基准索引为单字段索引则是数字形式，否则如果是多字段索引则为元组形式
- 3) 索引结束 ($<$)，为整型，默认 1000000000000。会依据选取的基准索引不同而动态修改，如果基准索引为单字段索引则是数字形式，否则如果是多字段索引则为元组形式
- 4) 每批操作执行的主键范围大小，默认 10000。
- 5) 每批执行后睡眠时间与上批执行时间的比值，比值越大，睡眠越久，默认 1.0。
- 6) 操作执行时间长度限制，超过此限制则自动退出。单位：秒。默认 14400（4 小时）。
- 7) 数据操作类型：delete 或者 update。Update 操作需要设定其执行的 set 语句。
- 8) 表名，必须含有主键，并且主键必须为整型。
- 9) where 条件。
6. 分区 (Alt-R)：可以查看、增加、删除指定表的分区，还可以定制自动添加分区的计划。进行增加、删除分区操作，将在所有 DBN 节点上执行。增加、删除分区的操作必须是针对已经分区的表（即 create table 时候指定 partition by ...），否则出错。**注意：**目前只支持按时间水平分区的表。分区计划详见 [4.11 计划任务](#)。
7. 禁用写操作：禁止对表的写操作，表的写操作被禁止后中间件上所有涉及该表的写操作将立即返回异常。
8. 启用写操作：重新启用对表的写操作。**提示：**禁用/启用写操作，可直接点击列表中的选项框完成。

9. 收集统计信息：收集表基本统计信息，包括表总记录数、平均记录大小、数据大小等。可保存为计划任务执行。查看表中索引字段的 **cardinality** 估值。

注意：“操作 DBN 时，是否拒绝用户对表的写操作”的选项，会禁止对表的所有写操作，对于数据量很大的表修改表定义操作会造成 **ddb** 阻塞，可以在进行这类操作时暂时禁止所有 **client** 上对该表的写操作（包括加锁），操作结束后允许写操作。

建表时的建表语句，需要增加额外的 **hint** 来说明一些 **DDB** 的必要信息，具体 **hint** 说明参见附录。

从 **DDB4.0.1** 开始，提供对 **Oracle** 聚簇表的支持，在管理表页面添加了聚簇管理的标签页。提供的功能包括：

1. 刷新 (**Alt-F**)：刷新显示系统中所有聚簇信息。
2. 创建聚簇 (**Alt-A**)：动态创建一个聚簇。必须输入 **SQL** 语句，具体的语法可以参考 **Oracle** 文档。聚簇的均衡策略必须通过末尾的 **hint** 注明。对于散列聚簇，创建聚簇时，该聚簇的索引也自动添加，索引名为聚簇名。而索引聚簇的索引必须手动添加。
3. 创建聚簇索引 (**Alt-I**)：为已存在的聚簇添加索引。需要输入 **SQL** 语句。
4. 删除聚簇 (**Alt-D**)：删除指定聚簇。
5. 删除聚簇索引 (**Alt-U**)：删除指定聚簇的索引。已经有关联表的聚簇的索引不能删，散列聚簇的索引也不允许删除。
6. 详细 (**Alt-T**)：显示指定聚簇字段的详细信息，对聚簇的 **SIZE** 参数进行修改。散列聚簇的 **SIZE** 参数不允许修改。提示：**双击列表中一个聚簇也可以弹出同样的窗口。**

4.4.5 在线修改表结构

在线修改表结构是 **DDB3.0-M3** 版本增加的一个功能。**MySQL** 在执行修改表结构操作时会锁表，因此对于大表操作来说，会影响线上应用。在线修改表结构这个功能的整体思路就是建立一张临时表，在空的临时表上先进行修改表结构操作，然后把数据分段拷贝到临时表，最后用临时表替换线上表。

在线修改表结构分为两部分，一部分是 **jython** 脚本程序，部署在 **scripts/OnlineAlterTable** 目录下；另一部分是调用接口，一般通过 **DBA** 工具或 **isql** 进行调用。在表结构字段及索引对话框中添加完修改表结构操作后，点击存为在线修改表结构任务，把当前操作保存为在线修改表结构任务。点击：表操作—>查看在线修改结构任务菜单，对任务进行管理。分为：

1. 刷新：获取并显示在线改表任务的最新状态。
2. 修改：修改一个在线修改表结构任务的参数，包括拷贝段大小和间隔时间。注意，该功能可以在脚本运行时进行修改，这样就可以动态调整修改表结构任务的执行情况。同时可以通过该功能查看任务的详细信息以及任务的执行进度。
3. 删除：删除一个在线修改表结构任务。
4. 暂停：暂停一个在线修改表结构任务。
5. 启动：启动一个在线修改表结构任务。
6. 撤销：撤销一个在线修改表结构任务。
7. 存为启动任务：设定在线改表任务的启动时间。
8. 日志：查看任务执行日志信息。
9. 调试信息：查看脚本执行时打印的详细调试信息。

目前在线修改表结构任务支持的 **DDL** 语句类型有：

1. 增加列
2. 增加索引
3. 修改列类型 (**alter table modify column**)





4. 删除列
5. 删除索引
6. 修改列 (`alter table modify column`)
7. 除表注释外的各种表选项的修改, 例如压缩表和默认字符集等

在线修改表结构任务, 有几个比较特殊的参数可以单独设定:

1. 拷贝段大小: 从原表将数据复制到 `ghost` 表时, 一次拷贝的记录数
2. 拷贝间隔: 拷贝完一次数据后, 中间的 `sleep` 时间
3. 是否记录 `binlog`: 创建临时表和拷贝数据的 `SQL` 语句, 是否记录到 `binlog` 中

提示: 对于 `Oracle` 类型的表, 不支持在线修改表结构。功能实现细节可以查看 [10.3 在线修改表结构](#)

为了确保在线修改表结构数据的正确性, 增加了数据校验功能。在在线修改表结构任务的详细信息页面, 点击“数据校验”可进入该改表任务的数据校验创建或修改页面。

-   创建数据校验任务。设定校验的各项参数, 然后确定。各项配置参数定义如下:
- 1) 表记录总数: `DDB` 表的记录总数, 即该表分布在各节点上的记录数之和。只需要提供估算值, 可以通过‘表操作’-‘修改表属性’进入表详情界面查询。
 - 2) 比较记录总数: 是指该 `DDB` 表需要进行校验的记录数。
 - 3) 每次比较记录数: 数据校验是一段段数据依次校验的, 该参数设定每个 `DBN` 上每次校验的记录数。该参数越大校验速度越快, 对节点造成的负载也会越大。
 - 4) 校验暂停时间: 在 `DBN` 上每校验一段数据后暂停的时间。该参数越小校验速度越快, 对节点造成的负载也会越大。
 - 5) 唯一字段: 指定表上的值唯一字段, 可以是唯一索引, 也可以是应用保证的数据唯一字段。如果选定的字段值实际并不唯一, 会导致校验失败。
 - 6) 是否自动启动: 如果选择自动启动, 则在任务保存后, 如果关联的在线改表任务已经完成数据拷贝操作, 则立即启动校验, 而如果在线改表任务尚未完成数据拷贝, 则校验任务会在数据拷贝完成后立即自行启动校验。如果不选择自动启动, 用户也可以在合适时间在校验任务配置界面将任务改为自动启动状态。
-   数据校验任务配置。当校验任务创建成功后, 通过在线改表任务界面可以进入校验任务配置管理界面。
- 1) 刷新: 获取校验任务最新的校验进度和任务状态。
 - 2) 启动/停止任务: 当任务处于停止状态, 可以点击“启动”, 校验任务进入自动启动状态; 当任务处于自动启动状态, 点击“停止”, 该任务进入停止状态; 当任务处于校验中状态, 点击“停止”, 该校验任务进入停止状态, 数据校验暂停, 再次启动时, 数据校验会从上上次停止的地方继续进行校验。
 - 3) 删除校验: 删除不再需要的校验任务。如果用户需要对同一个在线改表任务进行多次的校验, 则可以将完成的校验任务删除然后再次添加并启动。
 - 4) 修改校验参数: 在任务处于停止或者自动启动状态, 可以修改任务的表记录总数、比较记录总数、每次比较记录数。在校验中状态, 不允许进行修改。
 - 5) 修改暂停时间: 设置校验任务完成一次“每次比较记录数”所设定的校验量后的暂停时间。该参数在各状态都允许修改, 可在数据校验进行中, 通过设置合理的暂停时间, 调节校验操作对节点造成的压力。

4.4.6 管理视图

系统配置页面, 通过“查看视图” (`Alt-V`), 对视图进行管理:

1. 刷新 (`Alt-F`): 刷新显示系统中所有视图。双击列表中一个表可以显示该表含有的字段信息, 可进行字段操作。
2. 增加视图 (`Alt-A`): 动态增加一个视图。需要输入创建视图语句 `create view view_name as select_statement`, 详见 `mysql reference manual`。视图在创建时需要

保证数据的一致性，即视图中的每条记录只能来源于同一个 DBN 上的表，不能跨越多个 DBN。可通过下面的条件限制：

- 1) 视图所依赖的表必须采用相同的均衡策略
 - 2) 如果均衡策略所含桶数>1，且视图依赖多个表，那么试图创建语句中必须包含这些表在均衡字段上的等值条件。
 - 3) 视图可以不指定（不包含）均衡字段。
 - 4) 对视图的 **select,update,insert** 和 **delete** 通过 **mysql** 本身进行限制，有些视图不支持上述某些操作。
3. 删除视图（Alt-D）：动态删除一个视图。
 4. 修改视图（Alt-M）：利用 **alter view** 修改视图。

4.4.7 用户管理

分布式数据库用户（DDB User）是用于访问、管理整体分布式数据库系统时采用的用户。系统内部定义了三类，分别是

- ◆ **MAN**，普通数据库操作用户，如外部应用服务器或 **isql** 访问分布式数据库采用的用户；
- ◆ **DBA**，数据库管理用户，是系统管理员通过管理工具访问分布式数据库采用的用户，
- ◆ **Agent**用户，即 **DBN Agent**连接 **Master** 采用的用户。系统初始化时候已经默认为 **DBN Agent** 建立了 **Agent** 用户。

每个 **DDB User** 只能属于一个类别，每个类别都具有严格的使用约束，例如，**MAN** 类型的用户只能通过 **isql** 或 **JDBC** 接口访问分布式数据库；**Agent** 用户只允许 **DBN Agent** 使用；**DBA** 用户除了具有 **MAN** 用户的能力之外，还可以通过管理工具管理数据库。

每个 **DDB User** 都具有对各自数据库表的权限。*****代表各个 **DBN** 指定的数据库，**mysql.***代表各个 **DBN** 数据库服务的 **mysql** 表。权限有读（**Select**），写（**Insert、Update、Delete**），授权（**Grant Option**）、所有（**All**），其中“所有”权限包括读、写，但不包括授权。

另外，每个 **DDB User** 都具有一个“允许访问的主机地址”的属性，通过该属性可以限制该用户可以在那些机器上访问分布式数据库。地址可分为三类，允许访问的中间件 **IP** 地址，允许访问的查询服务器 **IP** 地址和允许作为 **DBA** 访问的主机 **IP** 地址。不同类型的 **DDB User** 所必须的允许访问主机地址的种类也不尽相同，**MAN**类型的用户至少需有一个中间件 **IP** 地址，**DBA** 类型的用户至少需有一个 **DBA** 主机 **IP** 地址。**IP** 地址目前支持%通配符，表示允许从任何地址访问，**192.168.%** 允许从任何以 **192.168** 开头的地址访问，当然也可以明确给出所有允许访问的详细地址。

每个 **DDB User** 是与数据库节点上的 **MySQL** 用户相关联的。

操作界面说明

“用户管理”（Alt-U）

1. 刷新（Alt-F）：刷新页面显示的用户列表信息
2. 增加用户（Alt-A）：增加用户，需要提供用户名、密码、允许访问地址列表、用户权限等
3. 修改用户（Alt-M）：修改用户，可修改用户类型及可访问 **IP**，修改密码，修改权限。但默认系统管理用户 **sys** 只能修改主机 **IP** 及访问密码，默认 **Agent** 用户 **agent** 只能修改其密码。
4. 删除用户（Alt-D）：根据用户名删除 **DDB User** 用户
5. 详细信息（Alt-E）：查看用户的详细信息。
6. 上次出错操作：查看、重做、删除用户管理上一次出错操作，在进行用户管理操作时

候若存在以前的出错操作，建议在上次出错操作排除之后进行。

用户过期

为了方便添加临时用户，可以设置用户过期，可以分别对用户权限、用户允许访问的主机地址以及整个用户三个层级设置过期时间。例如创建 **Man** 类型的用户 **A**，设置其权限为*上的读和写，并且设置了写权限的到期时间，则系统会在这个到期时间自动 **revoke** 用户 **A** 在*上的写权限；同理若一个主机地址含有到期时间，则系统中这个到期时间会自动删除这个允许访问的主机地址，这台主机就不允许再访问 **ddb** 了。整个用户过期，则直接 **drop** 掉此用户。

系统用户初始化

注意：系统初始化之前，必须在系统数据库节点上手工建立默认系统管理用户 **sys**，口令为 **netease**，设置用户权限为各个 **DBN** 指定的数据库上所有权限（包括授权）、**DBN** 中 **mysql** 表的 **SELECT, INSERT, UPDATE, DELETE, GRANT OPTION** 以及 **DBN** 上的 **FILE, SHUTDOWN** 等权限。可访问主机地址为 **Master** 服务器的地址，可作为 **DBA** 访问的主机地址为%（全部）。授权语句描述为：

```
grant ALL PRIVILEGES on dbn_database.* to sys@master_ip identified by 'netease'
with grant option;
grant ALL PRIVILEGES on mysql.* to sys@master_ip with grant option;
grant FILE, SHUTDOWN, SELECT, DROP, INSERT, CREATE, RELOAD, SUPER, REPLICATION
CLIENT on *.* to sys@master_ip with grant option;
```

每次 **Master** 在启动时都会检查系统用户信息并做相应的用户初始化工作，这些工作包括：

如果系统中不存在默认系统管理用户，则系统内部自动建立默认管理用户 **sys**，但不会在各个 **DBN** 节点自动建立此用户，需要管理员如上述建立。

如果系统中不存在 **Agent** 类型的 **DDB** 用户，则自动创建名为 **agent** 的 **Agent** 类型的用户，口令为 **netease**。权限为各自 **DBN** 节点指定数据库的 **SELECT,DELETE,INSERT,UPDATE**，以及 **DBN** 上所有数据库的 **FILE,SHUTDOWN**。允许访问主机 **IP** 限定为系统中所有 **DBN** 节点的 **IP** 地址。

注意：第一次启动 **Master** 以后请及时更新 **sys, agent** 用户的密码，并重新绑定 **sys** 用户可访问的 **DBA** 主机 **IP** 地址。

4.4.8 触发器与存储过程

DDB 支持在数据库节点上创建触发器和存储过程以及相关的管理操作。

触发器管理提供的功能包括：

1. 刷新（**Alt-F**）：刷新页面显示的触发器信息。
2. 增加（**Alt-A**）：创建触发器，需要说明数据库类型、触发器的 **SQL** 语句以及触发器所处的节点。选择触发器节点时，只能是同类型的节点。**提示：****SQL** 语句末尾需要通过添加 **/dbntype=...*/**来标注数据库类型，默认为 **MySQL**。
3. 删除（**Alt-D**）：删除指定的触发器记录。会弹出提示选择是否同时在节点上进行触发器的删除。
4. 修改（**Alt-M**）：修改触发器的定义语句或所在的数据库节点。
5. 详细：显示触发器的详细信息。双击触发器列表也可以显示详细信息。

6. 启用 (Alt-E)：设置触发器在节点上有效，等同于在已有节点上将重新创建已经删除的触发器。

7. 禁用 (Alt-B)：设置触发器在节点上无效，等同于在已有节点上删除触发器，但保留记录。

存储过程管理提供的功能包括：

1. 刷新 (Alt-F)：刷新页面显示的存储过程信息。
2. 增加 (Alt-A)：创建存储过程，需要说明数据库类型、存储过程的 SQL 语句以及存储过程所处的节点。选择节点时，只能是同类型的节点。**提示：**SQL 语句末尾需要通过添加 `/*dbntype=...*/` 来标注数据库类型，默认为 MySQL。
3. 删除 (Alt-D)：删除指定的存储过程记录。会弹出提示选择是否同时在节点上进行存储过程的删除。
4. 修改 (Alt-M)：修改存储过程的定义语句或所在的数据库节点。
5. 详细：显示存储过程的详细信息。双击存储过程列表也可以显示详细信息。

4.4.9 日志

点击“日志” (Alt-L)，可以查看系统配置方面的日志记录。

4.5 数据迁移

点击“数据迁移” (Alt-2)，进入数据迁移页面。

数据迁移，分为单线迁移和多桶并发离线迁移。

单线迁移又可分为在线迁移和离线迁移。在线迁移是利用 `select`、`insert` 操作完成记录迁移的，系统会对正在迁移的桶的部分数据加读锁，等这部分数据迁移完马上释放，所以对应用系统访问的影响较小，但迁移速度比较慢，对数据量大的应用不合适。单线离线迁移，系统会禁止应用访问，通过 DBN Agent 执行 `SELECT INTO OUTFILE` 及 `LOAD DATA LOCAL INFILE` 来完成迁移。在线迁移和单线离线迁移，可支持多个桶，但这些桶必须在同一个源数据库节点上。

多桶并发离线迁移，根据桶所在源数据库分组，不同源数据库上的数据可以并发迁移到目的数据库。迁移过程：从源数据库导出数据到 `myisam` 临时表，将临时表文件通过 `scp` 命令拷贝到目的数据库节点，目的数据库从临时表文件导入数据到数据库表，再删除源数据库上数据。由于采用了临时表，此操作不需要 DBN Agent 的支持。

4.5.1 ~~单线迁移~~

提示：DDB4.0 版本中不再提供该功能。

1. 刷新 (Alt-F)：刷新系统中已完成或进行中的在线迁移和单线离线迁移。
2. 执行 (Alt-G)：执行一个迁移。
3. 删除 (Alt-D)：删除选中的迁移操作。
4. 查看迁移日志 (Alt-L)：查看服务端迁移操作执行情况。

注意：“在线迁移”模式基本不影响应用访问，但“离线迁移”模式在迁移过程中，所有对该迁移桶的操作将被禁止，并且支持离线迁移的表都必须含有 `bucketno` 字段或者均衡字段为数值型。

4.5.2 多桶并发离线迁移

多桶并发离线迁移，界面上各个按钮操作与单线迁移类似。

4.5.3 在线迁移

由于原有 **ddb** 扩容方式都存在缺点，所以开发了一种基于 **mysql** 复制机制的在线迁移功能。具体的迁移实现细节可以参见后面数据维护章节，建议数据库管理员在执行迁移任务前先通读迁移原理。

在线迁移界面说明：

1. 新建：新建一个在线迁移任务
2. 开始迁移：开始执行一个之前建立的迁移任务。源节点或目标节点有重叠的任务不能同时执行，目标节点仍在使用的任务不能迁移
3. 停止迁移：停止一个正在执行的迁移任务。目前只允许在准备阶段中停止迁移，线上阶段是不允许暂停的
4. 详细：查看一个迁移任务的详细信息
5. 删除：删除一个迁移任务
6. 查看迁移日志：查看服务端迁移操作执行情况

一个具体的迁移任务界面中，包含下面几部分：

1. 源数据库：源数据库可以多个，可直接从系统中已存在的 **dbn** 中选取
2. 目的数据库：目的数据库可以多个，可直接从系统中已存在的 **dbn** 中选取。另外还需要指定每个目的数据库的数据库目录。源数据库和目的数据库，不能同时为多个，即只能执行一对多迁移（包括一对一）和多对一迁移
3. 迁移策略列表：详细列出了将要迁移的所有均衡策略，以及每个策略在迁移完成后桶分布情况
4. 设置策略：可以调整哪些策略需要迁移，哪些策略仍旧保留在源节点上
5. 设置桶对应关系：可以对迁移策略进行微调，指定哪些桶保留在源节点上，以及每个目标节点的桶分布策略
6. 同步超时时间：迁移过程中，必须等待 **mysql** 的 **slave** 节点复制完全跟上主节点之后再继续进行线上迁移操作。这个时候，会根据设定的同步超时时间，在时间段内不停地轮询从节点查看是否复制同步完成。如果同步超时时间内从节点仍然没有跟上主节点，则迁移操作失败。默认值是 **1000** 秒，对于负载较高的节点，最好把该值调大
7. 源节点切换间隔：多对一迁移的情况下，目标节点的 **MySQL** 复制进程在多个源节点之间来回切换，此参数控制切换频率。一对多迁移此参数无意义
8. 批量删除每批记录数：删除脏数据及过期数据时，调用 **DDB** 批量更新接口来完成。控制批量更新的数据分段长度
9. 批量删除睡眠比例：删除脏数据及过期数据时，调用 **DDB** 批量更新接口来完成。控制批量更新的每段数据执行完成后间隔时间比例
10. 任务所处阶段：表示当前任务所处阶段
11. 迁移状态：表示当前迁移任务的执行状态
12. 忽略通知 **Client** 失败的异常：把迁移后的桶对应关系通知所有客户端时，如果通知失败的处理机制
13. 忽略无法连接的 **Client**：把迁移后的桶对应关系通知所有客户端时是否忽略失效客户端
14. 忽略数据合并时主键冲突：多对一迁移时，在从多个源节点汇总数据过程中有可能产生主键冲突，定义是否忽略该冲突。一对多迁移此参数无意义
15. 分阶段执行：开始迁移操作。
16. 修改当前步骤：每个步骤有三种状态，未开始、准备阶段完成、执行完成。相应的，每个迁移步骤实际上需要执行两步，准备操作阶段和线上操作阶段。每个阶段执行完

成之后，会停下来等待管理人员确认是否成功。管理人员也可以手工修改当前迁移步骤的执行阶段

17. 存为计划：存为计划任务
18. 保存：保存迁移任务

4.5.4 迁移历史记录

可以查看到执行过的迁移操作的信息。

4.5.5 迁移计划

查看及管理所有迁移计划，参见[计划任务](#)。

4.6 悬挂事务

点击“悬挂事务”（Alt-6），进入悬挂事务页。

1. 刷新（Alt-F）：执行上次查询操作。
2. 查询（Alt-Q）：查询符合条件的上报分支事务。
3. 选中一行点击“查看”按钮（Alt-E），或者双击表格中一行，查看事务具体信息。
4. 删除（Alt-T）：根据事务提交时间删除记录。
5. 修改（Alt-M）：修改一个事务的状态，供管理员处理好事务之后使用。
6. 处理（Alt-P）：可手工提交或者回滚事务。
7. 检查（Alt-J）：检查指定事务的具体信息。
8. 日志（Alt-L）：查看 Master 端处理悬挂事务的日志记录。
9. 过滤：显示/隐藏过滤与分组功能面板。

4.7 警报

系统报警信息在“警报”（Alt-7）页显示/查询。

1. 刷新（Alt-F）：刷新显示上次查询结果，或者上次无查询的情况下显示所有报警信息。
2. 查询（Alt-Q）：查询报警信息，输入一定查询条件（不输入代表全部）可以查询符合条件的所有报警信息。
3. 查看（Alt-E）：对话框显示报警信息，需要先选中表格中一行；也可以直接双击表格中一行弹出显示对话框。
4. 删除（Alt-T）：删除特定时间之前的记录。
5. 日志（Alt-L）：查看 Master 端警报日志记录。
6. 过滤：显示/隐藏过滤与分组功能面板。

提示：在系统配置、负载、分支事务和警报表格中，可以通过点击列名来对显示结果排序，按住 Ctrl 再点击另一个列名可以再细化排序哦！

4.8 Client 统计

Client 统计页面，用户查看各个运行着的 Client 的资源情况，主要包括：

1. 连接信息：连接的基本信息、活跃连接的信息、隐式关闭连接的信息、悬挂关闭连接的信息。
2. Statement 信息：Statement 基本状态信息、活跃 Statement 的信息、隐式关闭 Statement 的信息、悬挂关闭 Statement 的信息。
3. PreparedStatement 信息：PreparedStatement 基本状态信息、活跃 PreparedStatement 的信息、隐式关闭 PreparedStatement 的信息、悬挂关闭 PreparedStatement 的信息。
4. Ops 信息：各个 Client 上的操作信息。各个参数意义详见《网易_分布式数据库_辅助工具使用说明.doc》SHOW OPS 命令。
5. 连接池信息：包括普通连接池和 XA 连接池。
6. 用户登录记录：Client 登录 Master 的用户信息记录。
7. 更多 Client 统计信息：包括 ddb 连接数、到 dbn 的连接数、创建的 statement 数目等。
8. 线程堆栈信息：查看 Client 上所有线程的堆栈信息。
9. 占用连接的线程堆栈信息：查看 Client 上占用连接的线程的堆栈信息。

清理悬挂连接：可以手工清理 Client 上指定超时时间的悬挂连接，使之释放资源。

异常关闭资源：所有 Client 上非正常关闭的资源列表。

保存结果：保存右侧结果页中的查询结果。

过滤：显示/隐藏过滤与分组功能面板。

提示：隐式关闭资源是指资源被垃圾收集器回收而关闭的资源；悬挂关闭是指资源悬挂超时被处理而关闭的资源。

4.9 监控统计任务

制定相应的监控统计任务，获取结果，为统计与分析做准备。

1. 刷新 (Alt-F)：获取系统中所有监控统计任务。
2. 增加任务 (Alt-A)：弹出任务增加对话框，制定一个监控统计任务。
3. 修改任务 (Alt-M)：修改原来的一个监控统计任务。
4. 删除任务 (Alt-D)：删除一个监控统计任务，若任务含有统计数据则一起删除。
5. 启动任务：启动一个监控统计任务，任务中的 Client 将统计各个计数。
6. 生成结果：对运行中的监控统计任务，获取一次当前的统计结果并保存到 master 系统库，Client 上各个计数不清零会继续累加。一个运行中的监控任务可随时获取一次结果。
7. 停止任务：停止一个监控统计任务，任务中的 Client 上的计数将清零。
8. 任务结果 (Alt-R)：弹出对话框，查看任务已经获取的结果描述信息，对结果的统计分析需要在“统计与分析”页面进行。
9. 查看日志 (Alt-L)：查看统计任务的日志。

4.10 统计与分析

针对监控统计任务返回的统计结果数据，可进行统计与分析。

4.10.1 统计

按钮功能如下：

1. 执行 (Alt-E)：执行一条统计命令。
2. 上一条 (Alt-P)：命令历史记录中上一条统计命令。
3. 下一条 (Alt-N)：命令历史记录中下一条统计命令。
4. 常用命令 (Alt-C)：弹出常用命令对话框，可增删改常用命令，将命令读入统计命令输入框。
5. 保存结果 (Alt-V)：把当前执行的命令及结果保存为文件。
6. 读取结果 (Alt-L)：可读取已保存的结果文件。
7. 帮助 (Alt-H)：显示语句帮助信息。

系统支持的统计命令：

- **help**：显示帮助信息；
- **简单统计命令语言**：（参见《网易_分布式数据库_辅助工具使用说明.doc》）
- **show stat ddb** 查询统计结果中的 DDB 语句级统计信息，语法：

```
show stat ddb[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

where：只选择涉及满足条件的统计信息，格式为：**WHERE <cond>[(and | or) <cond> ...]**，各个条件之间可用 **AND** 或 **OR** 连接，常用的条件有：

- **tables (like 'tbl_name')**：只选择涉及到 **table** 表的统计信息
- **client_id (= Num)**：只选择某个 **Client** 上运行的中间件的统计信息
- **count (= | < | > | <= | >= Num)**：只选择相应执行次数的统计信息
- **time (= | < | > | <= | >= Num)**：只选择符合总执行时间条件的统计信息
- **type (= 'sql_type')**：只选择对应类型的 **sql** 语句 (**SELECT**, **UPDATE**,...) 的统计信息

group_by：按属性名分组，可根据多个属性名分组，格式为：**GROUP BY <attr>[,<attr>,...]**，常用的分组属性有：

- **tables** 按照表名分组
- **client_id** 按照客户端 **Client** 分组
- **sql_sig** 按照语句签名分组
- **task_id** 按照统计任务分组
- **type** 按照不同类型的 **sql** 语句 (**SELECT**, **UPDATE**,...) 进行分组

having：只选择涉及满足条件的统计信息，跟 **where** 类似

order_by：根据属性进行排序，支持多个属性排序，**ORDER BY <attr> [desc | asc]>[,<attr> [desc | asc]>,...]**，常用排序的属性有：

- **count**：执行次数
- **time**：总执行时间
- **avg_time**：平均执行时间
- **type**：不同类型的 **sql** 语句 (**SELECT**, **UPDATE**,...)
- **mysql_count**：执行 DDB 语句时导致的 **MySQL** 执行次数
- **mysql_time**：执行 DDB 语句时导致的 **MySQL** 执行时间
- **dbn_count**：执行 DDB 语句时访问的后台数据库节点数
- **rows**：执行 DDB 语句时最底层操作扫描过的行数

limit：LIMIT number 只查询指定数目的记录数

label_by：LABEL BY <attr> 按照指定属性分标签页显示，常用的有：

- **tables** 按照表名分标签页
- **client_id** 按照客户端 **Client** 分标签页
- **sql_sig** 按照语句签名分标签页
- **task_id** 按照统计任务分标签页
- **type**：按照不同类型的 **sql** 语句 (**SELECT**, **UPDATE**,...) 分标签页
- **dbn** 按照 **dbn id** 分组，只对 **show mysql ss** 结果有效

- **show stat extended ddb** 查询统计结果中的 DDB 语句级统计信息，除在命令结果中

显示更多的属性外，与 SHOW STAT DDB 命令完全相同。

- **show stat mysql** 查询统计结果中的 MySQL 语句级统计信息，语法：

```
show stat mysql[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

所有条件与 **show stat ddb** 命令相同，增加了几个属性：

where dbn (= Num): 只选择对应 dbn 上的统计信息

group by dbn 按照 dbn id 分组

order by:

- **rnd_next**: 执行 MySQL 语句时 Handler_read_rnd_next 增加次数（即表扫描经过的行数）
- **avg_rnd_next**: 执行 MySQL 语句时 Handler_read_rnd_next 平均增加
- **read_next**: 执行 MySQL 语句时 Handler_read_next 增加次数（即索引扫描经过的行数）
- **avg_read_next**: 执行 MySQL 语句时 Handler_read_next 平均增加次数

label by dbn 按照 dbn id 分组，只对 show mysql ss 结果有效

- **show stat extended mysql** 查询统计结果中的 MySQL 语句级统计信息，除在命令结果中显示更多的属性外，与 SHOW STAT MYSQL 命令完全相同。

- **show stat ops**: 查询统计结果中的基本操作统计信息。语法：

```
show stat ops[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 where 条件有: task_id, client_id

常用的 label_by 有: task_id, client_id

- **show stat index**: 查询统计结果中的索引使用情况统计信息。语法：

```
show stat index[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 where 条件有: task_id, client_id, tablename, used_count

常用的 order_by 条件有: task_id, client_id, used_count

常用的 label_by 有: task_id, client_id, tablename

- **show stat column**: 查询统计结果中的属性使用情况统计信息。语法：

```
show stat column[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 where 条件有: task_id, client_id, tablename, refer_count

常用的 order_by 条件有: task_id, client_id, refer_count

常用的 label_by 有: task_id, client_id, tablename

- **show stat explain**: 查询统计结果中的 MySQL 执行计划统计信息。语法：

```
show stat explain[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 where 条件有: tablename, select_type, keyname, task_id

常用的 group_by 条件有: sql_sig, explain_sig, seq, tablename

常用的 order_by 条件有: count, rows

常用的 label_by 有: task_id, tablename

- **show stat bucket**: 查询统计结果中的桶级统计信息。语法：

```
show stat bucket[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 **where** 条件有: **task_id, client_id, policy, bucket, db_url**

常用的 **group_by** 条件有: **task_id, client_id, policy, bucket, db_url**

常用的 **order_by** 条件有: **policy, bucket, read_count, update_count, insert_count**

常用的 **label_by** 有: **task_id, client_id, policy, db_url**

- **show stat table memcached**: 查询统计结果中基于表的 **memcached** 操作统计信息。
语法:

```
show stat table memcached[ WHERE where][ GROUP BY group_by][ HAVING
having][ ORDER BY order_by][ LIMIT limit][ LABEL BY label_by]
```

常用的 **where** 条件有: **task_id, result_id, client_id, tablename**

常用的 **order_by** 条件有: **tablename, get_count, get_miss_count, set_count, set_datasize, del_count**

常用的 **label_by** 有: **task_id, result_id, client_id, tablename**

- **show stat mcv**: 查询统计结果中的 **DDB** 语句常数和参数值的 **MCV** 统计。语法:

```
show stat mcv[ WHERE where][ GROUP BY group_by][ HAVING having][ ORDER BY
order_by][ LIMIT limit][ LABEL BY label_by]
```

- 系统库查询语句:

show tables: 查看系统库中的表信息。

desc[ribe] <table>: 查看表定义。

select 语句: 直接从系统库中查询结果。

4.10.2 分析

根据统计结果提供了如下分析功能: (参见《网易_分布式数据库_辅助工具使用说明.doc》)

1. 分析得到忘记加均衡字段条件的语句
忘加均衡字段的语句, 会被发送到底层多个数据库节点, 而只有一个数据库节点返回结果, 造成系统性能下降。
2. 分析从不使用的索引
多余的索引会造成额外的开销, 从不使用的索引应该删除。
3. 发现需要增加的索引
分析每个语句执行时是否有效地使用了索引, 如果没有使用索引或索引效率不高则根据使用字段情况给出创建索引的建议。
4. 发现区分度不高的索引
发现在索引字段上区分度不高的索引。如果索引在表中的区分度不高, 将可能导致通过索引查找效率比全表扫描没有太大提高, 而索引又会带来 **insert** 的开销。应该避免使用这样的索引。
5. 分析被 **Mysql** 忽略的索引
通过系统记录的索引信息发现被包含的索引, 从而被 **Mysql** 忽略, 应该简化。
6. 发现可进行 **Using Index** 优化的潜在索引
执行 **select** 语句时如果语句返回的字段全部包含在查询所使用的索引中, 则语句执行时只需要从索引中获得返回值而无须访问实际的表记录, 这样可以提高查询执行效率。这类查询可称为 **Using index** 查询。分析当前的非 **Using index** 查询中, 在索引中增加长度最小的字段或增加最小的索引长度 (增加索引带来的 **insert** 代价最低), 就可以成为 **Using index** 查询的 **select** 语句。
7. 分析可能出现死锁的 **DDB** 和 **Mysql** 语句
根据 **DDB** 和 **mysql** 操作抛出的锁等待超时异常和死锁异常给出相互之间可能出现死

锁的操作。

8. 比较两次统计的结果

对两次访问统计的结果进行比较，统计和分析前后两次执行过程中系统负载、负载分布、执行效率的差别。统计的具体内容包括

- 表访问情况比较，包括：**select/update/delete/insert** 语句和所有语句的执行次数和执行时间的比较，表访问次数和执行时间所占比例的比较
- 每个 **sql** 语句的执行次数、执行总时间、平均执行时间、执行次数比例、执行总时间比例的比较，并且给出增加和减少的 **sql** 语句

4.11 计划任务

计划任务可用于安排定期的管理操作或者指定比较耗时的操作在系统负载较轻的时间执行。通过菜单“系统”->“计划任务”弹出计划对话框，可以管理系统中所有的计划任务。

计划对话框按钮功能如下：

1. 刷新（Alt-F）：从 **Master** 重新读取全部计划信息。
2. 增加（Alt-A）：增加一个计划。
3. 修改（Alt-M）：修改一个计划。
4. 删除（Alt-D）：删除一个计划。
5. 暂停（Alt-P）：暂停一个调度中的计划。
6. 重启（Alt-R）：重启启动一个被暂停的计划，计划重启后可能已经过期。
7. 组合（Alt-S）：选择列表中多个计划，可组合这些计划为一个新的计划。
8. 详细（Alt-E）：弹出窗口，查看计划详细信息。

对计划的说明：

1. 计划的名称，需唯一，可以由数字，字母，下划线组成，必须以字母或划线开头，规则和 **java** 的变量名规则一样。
2. 执行时间表达式
 - 1) 一个完整的表达式由至少 6 个（可能 7 个）有空格分隔的时间元素，分别代表：秒、分钟、小时、天（月）、月、天（星期）、年（可选）。其中每个元素可以是一个值（如 6），一个连续区间（9-12），一个间隔时间（8-18/4、/表示每隔 4 小时），一个列表（1,3,5），通配符*。由于“月份中的日期”和“星期中的日期”两个元素互斥，须要对其中一个设置为‘?’。

◆ 字段	◆ 允许值	◆ 允许的特殊字符
◆ 秒	◆ 0-59	◆ , - * /
◆ 分	◆ 0-59	◆ , - * /
◆ 小时	◆ 0-23	◆ , - * /
◆ 日期	◆ 1-31	◆ , - * / ? L
◆ 月	◆ 1-12 或 JAN-DEC	◆ , - * /
◆ 星期	◆ 1-7 或 SUN-SAT	◆ , - * / ? L
◆ 年（可选）	◆ 空 , 或者 1970-2099	◆ , - * /

- 2) “/”字符用来指定数值的增量。例如：在子表达式（分钟）里的“0/15”表示从第 0 分钟开始，每 15 分钟；在子表达式（分钟）里的“3/20”表示从第 3 分钟开始，每 20 分钟（它和“3, 23, 43”）的含义一样。

- 3) “?”字符仅被用于天(月)和天(星期)两个子表达式,表示不指定值。当2个子表达式其中之一被指定了值以后,为了避免冲突,需要将另一个子表达式的值设为“?”
- 4) “L”字符仅被用于天(月)和天(星期)两个子表达式,它是单词“LAST”的缩写,L在天(月)表达式代表一个月的最后一天;而L在天(星期)表达式代表一个星期的最后一天(星期六),如果“L”前面有具体内容,含义就不同了,“6L”表示这个月的最后一个星期五。
- 5) “#”字符只能用于天(星期)表达式,n#XXX,表示每月的第n个星期XXX
3. 计划开始时间,支持以下语法:
 - 1) 一个完整的日期的时间,形如“2007-08-30 12:00:00”。
 - 2) 一个完整的时间,形如“12:00:00”,这时认为日期是今天。
 - 3) 简写的时间和日期,形如“2 am tomorrow”,其中时间部分可使用数字表示24时制的小时、数字加am或pm表示12时制的上午或下午几点。
 - 4) 从当前时间向后隔多久时间,格式为“+数字 单位”,其中单位可以是h(小时)或m(分钟),如“+5 m”表示在当前时间5分钟之后开始,“+2 h”表示在当前时间2小时之后开始。
4. 计划结束时间,语法与开始类似,但以+开始的时间表示从开始时间之后隔多久时间,如“+2 h”表示统计在开始时间之后2小时结束,即统计历时2小时。
5. 任务语句:是形如“job_name1 statement1”的任务名称和具体执行语句的组合,多个组合之间用逗号分隔。即一个计划可以有多个任务,这些任务会依次执行。任务名称在一个计划中必须唯一。任务执行具体语句,目前支持:(参见《网易_分布式数据库_辅助工具使用说明.doc》)
 - 1) 模式定义与修改语句: CREATE/DROP POLICY、CREATE /DROP TABLE、CREATE/DROP INDEX、ALTER TABLE、CREATE/DROP/ALTER VIEW
 - 2) 分区管理语句: ADD/DROP PARTITIONS
 - 3) 增加 bucketno 语句: ADD BUCKETNO
 - 4) 用户与权限管理语句: ADD/DROP USER、GRANT/REVOKE、SET PASSWORD/QUOTA、ADD/REMOVE HOST
 - 5) 数据迁移语句: SINGLE MIGRATE、CONCURRENT MIGRATE、STOP ONLINE MIGRATE
 - 6) 备份导出语句: BACKUP、EXPORT、EXPORT SYSDB
 - 7) 监控任务语句: CREATE/DROP STAT TASK、START/STOP STAT TASK、GET STAT RESULT
 - 8) 收集表统计信息语句: COLLECT TABLE STAT
 - 9) 清理系统库语句: CLEAN SYS TABLES
 - 10) Use 命令
 - 11) 选项设置语句: execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients 选项的 set 语句

5 交互式 SQL 工具

本章详细介绍了交互式 SQL 工具(即 isql)使用方法,可用于查询或更新分布式数据库中数据或进行模式定义及其它管理工作。

交互式 SQL 工具为一个用于操作分布式数据库的命令程序,支持如下功能:

- ◆ 在分布式数据库中执行各类 DML 语句
- ◆ 数据库节点管理: 查看系统中的数据库节点列表及增加、删除、禁用、启用某个数据

- ◆ 库节点等操作
- ◆ 用户与权限管理：查看用户列表及某用户的权限，增加、删除用户及修改用户权限等操作
- ◆ 模式定义：增加、删除均衡策略，增加、删除表或修改表定义等操作
- ◆ 导入导出与迁移：导出分布式数据库模式定义或数据，导入数据，迁移数据等操作
- ◆ 运行状态监控：监控数据库中间件中连接池、语句及准备语句等资源使用情况，各类操作统计信息等
- ◆ 性能统计与分析：进行性能统计，分析统计结果
- ◆ 计划任务管理：增加、修改、删除、暂停、重启计划任务

5.1 使用交互式 SQL 工具

交互式 SQL 工具使用 `isql.sh` (Linux 系统) 或 `isql.bat` (Windows 系统) 脚本启动。一般的用法形如：

```
./isql.sh -h 主机 -o 端口 -u 用户名 -p 密码 -l logdir
```

这时将以指定的用户名和密码连接到指定的分布式数据库，此时操作模式为中间件模式(下一节说明)，只能执行 DML 操作。

从 4.0 版本开始，为支持多 DDB 和 zookeeper 式的 url，增加了 `-url` 参数，用于指定一个或者多个 ddb 的连接参数，用法为：

```
./isql.sh -url url1[;url2:url3;...] [-U 管理员用户名 -P 管理员密码 [-O 管理端口]]
```

url 的形式有两种，普通形式：

```
host:port?user=username&password=pass&logdir=dir&key=k
```

或 zookeeper 形式（具体可以查看[高可用性实现](#)）：

```
zookeeper://host:port[,host:port]/ddbname[?params][,ddbname[?params]]
```

这两种形式的 url 可以单独使用或者通过分号进行混合，若指定了 `-url` 参数，其余的中间件模式参数例如 `-h` `-o` `-u` `-p` 等都将忽略。

当且仅当 `-url` 参数值只与 1 个 DDB 关联时，可以通过 `-U` `-P` 参数指定管理员的用户名和密码，这样可同时使用中间件以及管理员模式。如果此时采用的是普通形式的 url，还需要通过 `-O` 参数指定管理端口，而 zookeeper 式的则不需要，若指定了管理端口也会被忽略。

如果 `-url` 参数值与多个 DDB 关联，例如有分号隔开了多个 url 或者 zookeeper 式的 url 中指定了多个 ddbname，则 `-U` `-P` `-O` 参数都会被忽略，只能开启中间件模式。

管理员用户一般使用如下命令连接到分布式数据库进行管理员操作：

```
./isql.sh -h 主机 -O 管理端口 -U 管理员用户名 -P 管理员密码 -l logdir
```

5.1.1 操作模式

根据调用 `isql.sh` 或 `isql.bat` 时指定的参数不同，交互式 SQL 工具可能工作在以下五种模式。每种模式需要部署的 jar 包和能够进行的数据库操作都不同。

中间件

当没有使用 **-d** 参数指定驱动模式（或指定 **-d ddb**）时，且使用 **-u** 和 **-p** 参数指定了用户名密码后或 **-url** 指定连接字符串时，交互式 **SQL** 工具即工作在中间件模式，这时将使用 **-u** 和 **-p** 指定的用户名密码连接到分布式数据库。另外如果使用 **-U** 和 **-P** 参数指定了管理员用户名和密码，则没有使用 **-u** 和 **-p** 指定普通用户管理员密码，交互式 **SQL** 工具也将工具中中间件模式，不过这时是使用 **-U** 和 **-P** 参数指定的用户名和密码。这时用户主要可以进行各类 **DML** 操作、查看系统中数据库节点、均衡策略或表的定义、修改自己的密码等操作。由于这些 **DML** 操作都会通过分布式数据库中间件来执行，因此称为中间件模式。

中间件模式需要部署以下 **jar** 包：**commons-cli-xxx.jar**、**mysql-connector-xxx.jar**、**netease-commons.jar**、**common.jar**、**db.jar**、**exec.jar**、**tool.jar**

并行节点操作

当交互式 **SQL** 工具工作在中间件模式时，可使用 **USE** 命令连接到一个或多个从属于分布式数据库系统的后端数据库节点（参见 5.1.3 节），称为并行节点操作模式。这一模式下所有语句都将通过 **MySQL JDBC** 驱动并行发送到所选择的数据库节点上执行。

注意此时交互式 **SQL** 工具同时工作在中间件模式下，所有中间件模式提供的命令仍然是可用的。但对那些中间件模式和并行节点操作模式都提供的命令（如 **DML** 操作，**SHOW TABLES**）等则会优先使用并行节点操作模式提供的命令。

管理员

当没有使用 **-d** 参数指定驱动模式（或指定 **-d ddb**）时，且使用 **-U** 和 **-P** 参数指定了管理员用户名密码后，交互式 **SQL** 工具即工作在管理员模式。这时用户可以进行各类分布式数据库管理操作，如管理数据库节点、均衡策略、建表、修改表定义等等。

用户可以同时指定 **-u/-p** 和 **-U/-P** 参数，这样交互式 **SQL** 工具同时工作在中间件模式和管理员模式，可以同时进行这两个模式支持的操作。

管理员模式需要部署以下 **jar** 包：**commons-cli-xxx.jar**、**mysql-connector-xxx.jar**、**netease-commons.jar**、**common.jar**、**db.jar**、**exec.jar**、**tool.jar**

系统数据库操作

当交互式 **SQL** 工具工作模式包含管理员模式时，可使用 **USE SYSDB** 命令运行在系统数据库模式，可以操作 **master** 的系统数据库。这一模式下所有语句都将被发送到 **master** 的系统数据库上执行。

注意此时交互式 **SQL** 工具同时工作在管理员模式下，所有管理员模式提供的命令仍然是可用的。

轻量级

当指定 **-d ddblw** 参数时交互式 **SQL** 工具即工作在轻量级模式，这时用户可使用轻量级 **JDBC** 驱动进行各类 **DML** 操作。

轻量级模式需要部署以下 jar 包：`commons-cli-xxx.jar`、`netease-commons.jar`、`common.jar`、`exec.jar`、`tool.jar`、`db-lw.jar`

5.1.2 命令行参数说明

调用 `isql.sh` 或 `isql.bat` 时使用 `-help` 参数可查看各命令行参数的说明。

5.1.3 使用交互式 SQL 并行操作多个数据库节点

交互式 SQL 工具还可用于并行操作从属于分布式数据库的多个数据库节点，这些操作直接通过 MySQL 的 JDBC 驱动进行，不通过分布式数据库中间件。

当交互式 SQL 工具工作在中间件模式时，可以使用如下命令指定要操作的多个数据库节点：

```
use (ALL | dbns)
```

若指定 `ALL` 则表示操作所有数据库节点，否则会操作在“`dbns`”中指定的那些数据库节点，“`dbns`”为一个数据库节点名称的列表。系统中所有数据库节点的列表可使用 `SHOW DBNS` 命令查看。

当执行上述命令后，接下来的所有操作都会并行的发送到上述命令选中的所有数据库节点中执行。

使用“`use dbi`”命令可以再切换回来，使所有数据库操作通过分布式数据库中间件执行。

5.1.4 调试模式

当交互式 SQL 工具工作在中间件模式下时，可以设置选项 `debug` 为 `true`，使中间件在执行语句时打印出更多的调试信息，包括在各个数据库节点上执行的语句、事务相关操作、写分布式事务日志、执行计划等，方便观察分布式数据库中间件查询处理的流程。这些信息以 `log4j` 的 `INFO` 级别打印。

5.2 命令参考

5.2.1 一般性说明

所有命令和选项都不区分大小写。

5.2.2 通用命令执行框架

交互式 SQL 工具是基于网易通用 Java 类库中的通用命令执行框架开发的。这一执行框架使用插件机制，每个插件提供一组命令。在 2.1.1 节中所说的各种工作模式实际上分别都对应于一个插件，如下：

◆ 中间件模式：对应于 `dbi` 插件

- ◆ 管理员模式：对应于 **dba** 插件
- ◆ 并行节点模式：对应于 **mdbn** 插件
- ◆ 系统数据库操作模式：对应于 **sysdb** 插件
- ◆ 轻量级模式：对应于 **lw** 插件

交互式 SQL 工具启动时会根据命令行参数的不同自动加载不同的插件，若指定了 **-u** 和 **-p** 参数时交互式 SQL 工具会自动加载 **dbi** 插件，这样中间件模式提供的命令就会自动可用。如果同时指定 **-u/-p** 和 **-U/-P** 参数，则交互式 SQL 工具会自动加载 **dbi** 和 **dba** 插件，这时中间件模式和管理员模式提供的命令都同时可用，即可以认为交互式 SQL 工具同时处于中间件模式和管理员模式下。

在任何时候，都可以用 **SHOW PLUGINS** 命令显示当前已经加载的插件。除上述插件外，还会显示一个名为 **core** 的插件，这一插件是通用命令执行框架用于完成其基本功能的插件，提供一些查看插件列表、命令列表、执行脚本等基本命令（见 5.2.5 节），如 **SHOW PLUGINS** 命令即是该插件提供的命令。

命令的分派

交互式 SQL 工具提供的每条命令都有一个标识字符串，标识字符串即该命令的前几个单词，这几个单词可以将一条命令与其它命令区分开。如建表语句 **CREATE TABLE** 命令的标识字符串就是“**CREATE TABLE**”。同一个插件提供的命令的标识字符串都是不同的，但某个命令的标识字符串可能会是另一个命令标识字符串的前缀，如 **dbi** 插件提供命令 **DESC** 和 **DESC POLICY**，这时将使用最大匹配原则，即：

```
DESC POLICY some_policy;
```

命令会被分派给 **DESC POLICY** 命令，而：

```
DESC some_table;
```

命令会被分派给 **DESC** 命令。

可以用 **SHOW COMMANDS** 命令查看当前所有可用的命令，输入中的 **NAME** 列即为命令的标识字符串。

注意不同的插件可能会提供具有相同字符串的命令，如 **dbi** 插件和 **mdbn** 插件都提供 **SELECT** 命令。这时将优先使用排名在前的插件提供的 **SELECT** 命令。所谓排名即是在 **SHOW PLUGINS** 输出中的排名。当在中间件模式下使用 **USE** 命令切换到并行节点操作模式时，交互式中间件会自动将 **mdbn** 插件设为首选插件，即排名仅在 **core** 插件之后，这时的 **SELECT** 语句将会调用 **mdbn** 插件提供的 **SELECT** 命令，即在各节点上并行执行。

但这时也可以强制调用 **dbi** 插件提供的 **SELECT** 命令，方法是在命令之前使用如下的格式显式指定插件名称：

```
插件名称: 命令
```

如强制调用 **dbi** 插件提供的 **SELECT** 命令可用：

```
dbi:SELECT * FROM ...
```

5.2.3 通用选项

在说明各具体的命令之前，先要了解交互式 SQL 工具的一些选项，这些选项将会影响到众多命令的行为。

可以使用以下命令查看所有选项当前的值及其说明：

```
SHOW OPTIONS;
```

根据交互式 SQL 工具所处的操作模式的不同，系统提供的选项也会有所差别。

要修改选项的值要使用 SET 命令，语法为：

```
SET option_name value;
```

如设置选项 **force** 的值为 **true** 可用如下命令：

```
SET force true;
```

交互式 SQL 工具提供的所有选项详细说明如下：

force

是否强制执行所有命令。为一布尔类型选项，默认为 **false**，这时交互式 SQL 工具要执行某些操作（如直接操作数据库节点插入数据）时会要求用户确认。设置为 **true** 时交互式 SQL 工具会强制性执行所有命令，不要求用户确认，这在定时执行一个脚本时可以防止由于要求用户确认导致操作不能继续执行。

quiet

是否输出命令的结果。为一布尔类型选项，默认为 **false**，这时交互式 SQL 工具在执行众多命令时都会输出命令结果，如 **SELECT** 语句输出查询结果，**INSERT** 等输出更新了多少行等。设置为 **true** 可指定交互式 SQL 工具不输出任何结果。通常在执行一个 SQL 脚本，又不需要查看每条命令的输出结果时可设置 **quiet** 为 **true**。

delimiter

命令结束符。为一字符选项，默认为分号；，即交互式 SQL 工具在遇到分号时（不包括用引号括起的字符串中的分号）时认为一条命令结果并执行这一命令。可设置为其它任何字符，但不能是用来括起字符串的单引号'和双引号”。

ignore_error

在执行一条命令出错时是否继续执行后续命令。默认为 **false**。

charset

交互式 SQL 工具进行文件操作时使用的字符集。默认为 **UTF-8**。可设置为其它任何当前系统支持的字符集。这一选项主要用于指定执行脚本命令（**SOURCE**）时用于读取脚本所用的字符集，或导入导出数据时的字符集。

autocommit

自动提交模式。默认为 **true**，即每个对数据库的 **DML** 操作都作为一个单独的事务。注意 **autocommit** 只影响中间件或轻量级模式下对数据库进行的 **DML** 操作，不影响管理员模式下进行的模式定义等操作，这些操作都是作为单独的事务执行的。

execute_on_dbn

执行一些管理员操作（如修改表模式等）时是否在数据库节点上执行。默认为 **true**。

这一参数只有在交互式 SQL 工具工作在管理员模式下可用。

ignore_failed_clients

执行某些需要通知所有连接到分布式数据库的中间件程序的操作（如修改表模式等）时是否忽略通知出错的客户端。默认为 **false**。

这一参数只有在交互式 SQL 工具工作在管理员模式下可用，且此选项已经包含下面一个选项 **ignore_unreachable_clients**，即当 **ignore_failed_clients** 设置为 **true** 的时候，**ignore_unreachable_clients** 选项必定为 **true**。

ignore_unreachable_clients

执行某些需要通知所有连接到分布式数据库的中间件程序的操作（如修改表模式等）时是否忽略无法连接的客户端。默认为 **false**。

这一参数只有在交互式 SQL 工具工作在管理员模式下可用。

debug

执行语句时打印出更多的调试信息，包括在各个数据库节点上执行的语句、事务相关操作、写分布式事务日志、执行计划等，方便观察分布式数据库中间件查询处理的流程。这些信息以 **log4j** 的 **INFO** 级别打印。

这一参数只有在交互式 SQL 工具工作在中间件模式下可用。

prepare

当该参数设为 **true** 时将使用 **PREPARE** 的方式执行给定的 SQL 语句，详细说明见 5.2.6 节。

5.2.4 结果重定向

交互式 SQL 工具的输出结果可以被重定向到文件中。所有交互式 SQL 工具接受的命令格式如下：

```
COMMAND [(>> | >>>) file];
```

由于在 SQL 命令中 **>** 表示逻辑大于比较，为方便解析，使用 **>>** 和 **>>>** 来表示重定向。其中 **>>** 表示覆盖指定的文件，**>>>** 表示追加到指定文件的末尾。

5.2.5 基本命令

HELP

HELP 命令打印交互式 SQL 工具帮助信息或某条命令的详细说明。语法：

```
HELP [command];
```

参数说明：

- ◆ **command**: 显示指定命令的详细说明，若省略则显示一般性的交互式 SQL 工具帮助信息。

命令返回值为一个字符串，其中包含命令的语法，简要说明和详细使用说明。

SHOW COMMANDS

SHOW COMMANDS 命令显示所有可用命令或指定插件提供的命令。语法：

```
SHOW COMMANDS [FOR plugin];
```

参数说明：

- ◆ **plugin**: 显示指定插件提供的命令。交互式 SQL 工具提供的命令根据相关性被划分为多个组，每个组称为一个插件。系统中可能有的插件有：
- ◆ **core**: 无论工作在什么模式下，交互式 SQL 工具都会加载一个名为 **core** 的插件，本节中的这些基本命令即是由这一插件提供的。
- ◆ **dbi**: 当交互式 SQL 工具工作在中间件模式下时，则会加载 **dbi** 插件，中间件模式提供的经由分布式数据库中间件执行的命令即是由 **dbi** 插件提供。
- ◆ **mdbn**: 当使用 **USE** 命令切换到并行节点操作模式，使用交互式 SQL 工具并行操作多个数据库节点时，则还会动态加载 **mdbn** 插件，这一插件提供的命令使用 MySQL JDBC 驱动执行，直接操作数据库节点。当使用“**USE dbi**”命令切回到中间件模式时 **mdbn** 插件将被自动卸载。
- ◆ **dba**: 当交互式 SQL 工具工作在管理员模式下时，会加载 **dba** 插件，管理员模式提供的所有命令都由这一插件实现。
- ◆ **sysdb**: 当使用 **USE SYSDB** 命令切换到系统数据库模式操作 **master** 系统库的时候，则还会动态加载 **sysdb** 插件，这一插件提供的命令使用 MySQL JDBC 驱动执行，直接操作 **master** 系统数据库。当使用“**USE**”命令切回到别的模式时 **sysdb** 插件将被自动卸载。
- ◆ **lw**: 当交互式 SQL 工具工作在轻量级模式下时，会加载 **lw** 插件，实现轻量级模式提供的命令。

命令返回值为一个表格，每行为一个命令的信息，包含以下各列：

- ◆ **PLUGIN**: 命令所属的插件。当指定了 **FOR plugin** 时不显示这一列。
- ◆ **NAME**: 命令名。
- ◆ **SYNTAX**: 命令语法。
- ◆ **DESCRIPTION**: 命令功能的简单描述。

命令语法使用如下表示法：

- ◆ 全大写单词表示关键字。
- ◆ 全小写单词表示命令中的参数。
- ◆ |表示逻辑或。
- ◆ +表示出现一次或多次。

SHOW OPTIONS

SHOW OPTIONS 命令显示所有可用选项或指定插件提供的选项。语法:

```
SHOW OPTIONS [FOR plugin];
```

参数说明:

- ◆ **plugin**: 显示指定插件提供的选项，若不指定则显示所有选项。

命令返回值为一个表格，每行为一个命令的信息，包含以下各列:

- ◆ **PLUGIN**: 选项所属的插件。当指定了 **FOR plugin** 时不显示这一列。
- ◆ **NAME**: 参数名。
- ◆ **VALUE**: 参数的当前值
- ◆ **DESCRIPTION**: 参数的简单描述。

SHOW PLUGINS

SHOW PLUGINS 命令显示当前已经加载的插件信息。语法:

```
SHOW PLUGINS;
```

命令返回值为一个表格，每行表示一个插件的显示，包含以下各列:

- ◆ **NAME**: 插件名称。
- ◆ **#COMMANDS**: 插件提供的命令个数。
- ◆ **#OPTIONS**: 插件提供的选项个数。
- ◆ **DESCRIPTION**: 插件功能的简单描述。

USAGE

USAGE 命令显示指定命令的简要语法信息。语法:

```
USAGE command;
```

参数说明:

- ◆ **command**: 显示指定命令的简要语法信息。

命令返回值为一个字符串，内容为命令的语法信息。

EXECFILE

EXECFILE 执行一个命令脚本。语法:

```
EXECFILE [-q] file;
```

参数说明:

- ◆ **-q**: 安静 (**quite**) 模式，即在执行脚本过程中不输出任何结果。
- ◆ **file**: 要执行的脚本路径。

EXECFILE 命令是通用命令执行器提供的用于执行一个脚本的命令，与中间件模式提供的 SOURCE 命令具有类似功能，但 SOURCE 命令更灵活，功能更强，一般情况下都建议使

用 **SOURCE** 命令。

命令返回值为空。

EXIT

EXIT 命令退出交互式 **SQL** 工具。语法：

```
EXIT;
```

PREFER

PREFER 命令设置一个插件为首选插件。语法：

```
PREFER plugin;
```

参数说明：

◆ **plugin**: 要设置为首选插件的插件名，这一插件必须是已经被加载的插件。

首选插件即是在 **SHOW PLUGINS** 结果中排名仅在 **core** 插件之后的插件。设置首选插件主要是为了处理多个插件同时提供同一命令时的命令分派行为，此时排名在前的插件提供的命令将优先调用。

注意这一命令是通用命令执行框架提供的命令，一般情况下交互式 **SQL** 工具已经自动为你设置好的首选插件，在使用交互式 **SQL** 工具时不建议使用直接 **PREFER** 命令来调整插件排名。

命令返回值为空。

USE

USE 命令切换交互式 **SQL** 的工作模式。语法：

```
USE SERIAL (dbi | ALL | dbns | dba | sysdb | DBNS FOR (TABLE|POLICY) name)
```

参数说明：

- ◆ **SERIAL**: 连接到多个数据库节点时在这些节点上的操作顺序执行。
- ◆ **dbi**: 切换到中间件模式，必需在启动交互式 **SQL** 工具时指定了普通用户密码后才可使用本参数。
- ◆ **ALL**: 切换到并行节点操作模式，并行操作所有节点。
- ◆ **dbns**: 切换到并行节点操作模式，并行操作给出的节点。**dbns** 为数据库节点名称列表，空格分隔。
- ◆ **sysdb**: 切换到系统库操作模式。
- ◆ **DBNS FOR (TABLE|POLICY) name**: 连接到指定表或均衡策略使用的所有数据库节点。

操作模式：中间件、管理员、轻量级、并行节点操作

命令返回值为空。

SHOW USED DBNS

SHOW USED DBNS 命令显示并行节点操作模式时操作的数据库节点。语法：

```
SHOW USED DBNS;
```

命令返回值为一个表格，每一行为一个数据库节点的信息，包含以下各列：

- ◆ NAME: 节点名。
- ◆ URL: 连接该数据库时使用的 JDBC URL。

操作模式：并行操作操作

SHOW DDBS

SHOW DDBS 命令显示当前连接系统中所有的 DDB 环境。语法：

```
SHOW DDBS;
```

命令返回值为一个表格，每一行为一个 DDB（Master）的信息，包含以下各列：

- ◆ NAME: DDB 名称。
- ◆ MASTER_HOST: Master 主机地址。
- ◆ MASTER_PORT: DBI 连接端口。
- ◆ DBA_PORT: DBA 连接端口。

操作模式：中间件、管理员、轻量级

注意：管理员与轻量级模式下，此命令最多返回一条记录，因为只有中间件模式下才能指定-m 参数连接多个 DDB。

SHOW CURRENT DDB

SHOW CURRENT DDB 命令显示当前使用的 DDB。语法：

```
SHOW CURRENT DDB;
```

命令返回值为一条记录，表示当前使用的 DDB 信息，列信息与 [SHOW DDBS](#) 命令返回值相同。

操作模式：中间件、管理员、轻量级

注意：管理员与轻量级模式下，因为最多只有一个 DDB，故此命令与 [SHOW DDBS](#) 效果相同。

USE DDB

USE DDB 命令设置当前默认的 DDB 环境。语法：

```
USE DDB ddb_name;
```

命令返回值为空。

操作模式：中间件、管理员、轻量级

5.2.6 数据访问与修改

DML 语句

DML 语句指 **SELECT**、**INSERT**、**UPDATE**、**DELETE** 这四条 SQL 语句。中间件、并行节点操作和轻量级模式都提供这四条语句，但具体的支持能力有所不同：

- ◆ 中间件模式：不支持某些高级特性，如子查询等，具体参见《开发者手册》。
- ◆ 并行节点操作模式：支持所有 MySQL JDBC 支持语句，具体应参考 MySQL 手册。
- ◆ 轻量级模式：与中间件模式相同。

在中间件或轻量级模式下，命令返回值对于 **SELECT** 语句返回使用表格表示的结果集，对于更新语句返回更新记录数。

在并行节点操作模式下，对每个节点先返回形如“**Results of** 数据库节点名”的标识，然后返回命令结果。

事务控制语句

事务控制语句指 **BEGIN**、**COMMIT**、**ROLLBACK**、**AUTOCOMMIT** 这四条用于控制事务开始与结束的 SQL 语句。中间件、并行节点操作和轻量级模式都提供这四条语句，各语句功能如下：

- ◆ **BEGIN**：开始一个事务并设置当前事务状态为非自动提交模式。
- ◆ **COMMIT**：提交一个事务，若当前为自动提交模式则这一命令没有效果。提交事务后自动提交状态不变。
- ◆ **ROLLBACK**：回滚一个事务，若当前为自动提交模式则这一命令没有效果。回滚事务后自动提交状态不变。
- ◆ **AUTOCOMMIT**：设置当前事务状态为自动提交模式。

命令返回值为空。

以 PREPARE 方式执行 SQL 语句

由于中间件在处理 **PREPARE** 和非 **PREPARE** 模式时的处理流程有很大不同，因此交互式 SQL 工具提供以 **PREPARE** 方式执行 SQL 语句的方法，方便测试使用 **PREPARE** 模式时的查询处理功能。

要使交互式 SQL 工具以 **PREPARE** 方式执行 SQL 语句，首先需要设置参数 **prepare** 的值为 **true**，即使用如下命令：

```
SET prepare true;
```

然后输入包含的参数值的 SQL 语句执行，但要使用特殊的标识 **/*?*/** 来指定语句中的参数位置。交互式 SQL 工具会自动分析这一标识，提供出相关的 **PREPARE** 语句，并设置相应的参数并执行。如若输入以下语句：

```
SELECT * FROM Blog WHERE UserID = 123/*?*/ AND AllowView <= 10000/*?*/ ORDER BY PublishTime DESC LIMIT 5;
```


则交互式 SQL 工具会发现这一语句中包含两个参数，分析出 PREPARE 语句如下：

```
SELECT * FROM Blog WHERE UserID = ? AND AllowView <= ? ORDER BY PublishTime DESC  
LIMIT 5;
```

然后设置好两个参数的值并执行。

注意/*?*/标志与参数值之间不能有空格。

EXPLAIN

EXPLAIN 命令显示给定 SQL 语句的执行策略。语法：

```
EXPLAIN sql;
```

参数说明：

◆ **sql**: 执行策略

返回值为一个表格，只有一个名为 **PLAN** 的列，表示 SQL 语句的执行策略。

工作模式：中间件、轻量级

相关选项：无

CLOSE CONNECTIONS

CLOSE CONNECTIONS 命令关闭数据库连接。语法：

```
CLOSE CONNECTIONS;
```

交互式 SQL 工具在以中间件、轻量级等模式操作数据库时，默认是在进行第一次操作时得到到数据库的连接，并保持连接供以后的操作使用。当需要关闭这些连接时可以使用本命令，这样执行下一次操作时将会重新获取连接。

返回值为空。

工作模式：中间件、轻量级、并行节点操作

5.2.7 数据库节点管理

START DBN

START DBN 命令启动指定名称的数据库节点。语法：

```
START DBN name;
```

参数说明：

◆ **name**: 数据库节点名称

命令返回值为空。

可使用 SHOW DBNS 命令查看当前系统中的数据库节点信息。注意使用这一命令需要部署对应的 Agent。

工作模式：管理员

相关选项：无

STOP DBN

STOP DBN 命令关闭指定名称的数据库节点。语法：

```
STOP DBN name [reason];
```

参数说明：

- ◆ **name**: 数据库节点名
- ◆ **reason**: 一个任意的字符串，表示关闭原因，可以省略。注意若 **reason** 若不仅仅是一个单词，则使用单引号。

命令返回值为空。

可使用 SHOW DBNS 命令查看当前系统中的数据库节点信息。注意使用这一命令需要部署对应的 Agent。

工作模式：管理员

相关选项：无

ADD DBN

ADD DBN 命令增加一个数据库节点。语法：

```
ADD DBN name 'url' [user password] [SSHUSER sshuser] [SSHPORT sshport] [without  
init user] [PARENTID parentid] [CONFIGFILE 'config_file']  
[SCHEMA schema] [TABLESPACE tablespace]
```

参数说明：

- ◆ **name**: 数据库节点名。不能与系统中已有数据库节点重复。
- ◆ **url**: 使用 JDBC 连接该数据库时使用的 URL，如 'jdbc:mysql://192.168.0.1:3306/db'，需要使用单引号。
- ◆ **user password**: 分布式数据库用于操作该节点的超级用户名和密码，若省略则使用系统默认系统用户。
- ◆ **sshuser**: 使用 SSH 登录到指定数据库服务器时使用的用户名。
- ◆ **sshport**: 使用 SSH 登录到指定数据库服务器时使用的端口。
- ◆ **without init user**: 不在此节点上创建系统中已有用户。若不指定 **without init user**，则分布式数据库在增加数据库节点的同时会将系统中已有用户信息加入到新增节点中。
- ◆ **parentid**: 添加节点作为从节点，其主节点 ID。
- ◆ **config_file**: MYSQL 节点配置文件名，如 /home/ddb/mysql/my1.cnf。
- ◆ **schema**: Oracle 节点配置，设置连接所访问的 schema，如果不设置，则默认为 url 中末尾所标识的数据库名。
- ◆ **tablespace**: 默认的表空间，默认为 Definition.DEFAULT_TABLESPACE_NAME。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

DROP DBN

DROP DBN 命令删除一个数据库节点。语法：

```
DROP DBN name;
```

参数说明：

◆ **name**: 数据库节点名。

命令返回值为空。

可使用 SHOW DBNS 命令查看当前系统中的数据库节点信息。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

DISABLE DBN

DISABLE DBN 命令禁用一个数据库节点。语法：

```
DISABLE DBN name;
```

参数说明：

◆ **name**: 数据库节点名。

命令返回值为空。

禁用某数据库节点后，在执行涉及到这一数据库节点的查询时将立即抛出异常。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

ENABLE DBN

ENABLE DBN 命令启用一个原被禁用的数据库节点。语法：

```
ENABLE DBN name;
```

参数说明：

◆ **name**: 数据库节点名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SHOW DBNS

SHOW DBNS 命令显示分布式数据库中已有的数据库节点列表，包括各节点的名称和 URL。语法：

```
SHOW DBNS;
```

命令返回值为一个表格，按 URL 排序，每一行表示一个数据库节点的信息，包含以下各列：

- ◆ ID: 标志。
- ◆ NAME: 节点名。
- ◆ URL: 连接该数据库时使用的 JDBC URL。
- ◆ DBNTYPE: 数据库节点类型
- ◆ STATUS: 节点运行状态，ALIVE 或者 DEAD。
- ◆ ENABLED: 是否在启用状态。
- ◆ MASTER/SLAVE: 属于 Master 节点或者 Slave 节点。
- ◆ AUTO_SWITCH: 如果是从节点，显示是否属于自动切换节点。

工作模式：管理员、中间件、轻量级

相关选项：无

SHOW DBNS FOR POLICY

SHOW DBNS FOR POLICY 命令显示分布式数据库中指定策略所在的数据库节点列表，包括各节点的名称和 URL。语法：

```
SHOW DBNS FOR POLICY name;
```

参数说明：

- ◆ name: 策略名

命令返回值与 [SHOW DBNS](#) 命令相同。

工作模式：管理员、中间件、轻量级

相关选项：无

SHOW DBNS FOR TABLE

SHOW DBNS FOR TABLE 命令显示分布式数据库中指定表所在的数据库节点列表，包括各节点的名称和 URL。语法：

```
SHOW DBNS FOR TABLE name;
```

参数说明：

- ◆ name: 表名

命令返回值与 [SHOW DBNS](#) 命令相同。

工作模式：管理员、中间件、轻量级

相关选项：无

SHOW DBNS FOR RECORD

SHOW DBNS FOR RECORD 命令显示分布式数据库中指定表所在的数据库节点列表，即各节点的 URL。语法：

```
SHOW DBNS FOR RECORD TABLE=table_name, {VALUE=value[, value2, ...] | VALUE=(v1, v2)[, (v3, v4), ...]};
```

参数说明:

- ◆ **table_name**: 表名
- ◆ **value**: 均衡字段参数值

工作模式: 管理员、中间件、轻量级

相关选项: 无

SHOW MASTER DBNS

SHOW MASTER DBNS 命令显示分布式数据库中所有数据库 **Master** 节点列表, 包括各节点的名称和 URL。语法:

```
SHOW MASTER DBNS;
```

命令返回值为一个表格, 按 **URL** 排序, 每一行表示一个数据库节点的信息, 包含以下各列:

- ◆ **ID**: 标志。
- ◆ **NAME**: 节点名。
- ◆ **URL**: 连接该数据库时使用的 JDBC URL。
- ◆ **STATUS**: 节点运行状态, **ALIVE** 或者 **DEAD**。
- ◆ **ENABLED**: 是否在启用状态。
- ◆ **SLAVE_COUNT**: **Slave** 节点个数。

工作模式: 管理员、中间件、轻量级

相关选项: 无

SHOW SLAVE DBNS

SHOW SLAVE DBNS 命令显示分布式数据库中所有 **Slave** 节点列表或特定 **Master** 节点下的 **Slave** 节点列表, 包括各节点的名称和 URL。语法:

```
SHOW SLAVE DBNS [FOR name];
```

参数说明:

- ◆ **name**: **Master** 节点的名称

命令返回值为一个表格, 按 **URL** 排序, 每一行表示一个数据库节点的信息, 包含以下各列:

- ◆ **ID**: 标志。
- ◆ **NAME**: 节点名。
- ◆ **URL**: 连接该数据库时使用的 JDBC URL。
- ◆ **ENABLED**: 是否在启用状态。
- ◆ **STATUS**: 节点运行状态, **ALIVE** 或者 **DEAD**。
- ◆ **MASTER**: 复制的 **master** 节点名称。
- ◆ **REP_DELAY**: 落后 **Master** 的时间长度。
- ◆ **VALID_DELAY**: 容许的复制落后时间。
- ◆ **WEIGHT**: **Slave** 节点的权重。

◆ **IGNORE_TABLES:** Slave 节点复制忽略的表名列表。

工作模式：管理员、中间件、轻量级

相关选项：无

START SLAVES

START SLAVES 命令启动指定名称的 Slave 数据库节点的复制。语法：

```
START SLAVES name[, name2, ...] [WITHOUT MODIFY CONFIG];
```

参数说明：

- ◆ **name:** 数据库节点名称
- ◆ **WITHOUT MODIFY CONFIG:** 不修改数据库配置文件

命令返回值为空。

工作模式：管理员

相关选项：无

STOP SLAVES

STOP SLAVES 命令停止指定名称的 Slave 数据库节点的复制。语法：

```
STOP SLAVES name[, name2, ...] [WITHOUT MODIFY CONFIG];
```

参数说明：

- ◆ **name:** 数据库节点名称
- ◆ **WITHOUT MODIFY CONFIG:** 不修改数据库配置文件

命令返回值为空。

工作模式：管理员

相关选项：无

SWITCH SLAVE

SWITCH SLAVE 命令切换指定名称的从节点为主节点。语法：

```
SWITCH SLAVE name EXPIRED time;
```

参数说明：

- ◆ **name:** 从节点名称。
- ◆ **time:** 等待主从节点同步时间，等同步后再切换，若超过这个时间不能同步，则返回失败，单位毫秒。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

BUILD SLAVES

BUILD SLAVES 命令重建指定名称的从节点。语法：

```
BUILD SLAVES name[ WITHLINK 'linkdir' ][, name2 [WITHLINK 'linkdir2'], ...];
```

参数说明：

- ◆ **name**: 从节点名称。
- ◆ **linkdir**: 将镜像库新建的数据库目录建成链接符号，linkdir 为需要链接的实际目录。

命令返回值为空。

工作模式：管理员

相关选项：无

SET DBN DIRTY

SET DBN DIRTY 命令设置节点是否含有脏数据。语法：

```
SET DBN DIRTY name[,name2,name3,...] (TRUE | FALSE) [ WITH POLICY  
ply_name[,ply_name2, ply_name3,...]];
```

参数说明：

- ◆ **name**: 节点名称，多个节点用逗号连接，节点名最好用双引号括起。
- ◆ **ply_name**: 策略名称，指定操作对象为策略下的表。若无 WITH POLICY 项，操作对象为整个 DBN 下所有表。只有在 DIRTY 设置为 TRUE 的情况下才能设置 POLICY。若 DIRTY 设置为 FALSE，则 DBN 下所有表 DIRTY 属性都设置为 FALSE，不能指定 POLICY。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SET SLAVE AUTO SWITCH

SET SLAVE AUTO SWITCH 命令设置指定的一个从节点为自动切换节点，当该从节点所对应的主节点失效后，该从节点自动切换为主节点。语法：

```
SET SLAVE AUTO SWITCH name TRUE | FALSE;
```

参数说明：

1. **name**: 节点名称，必须是 slave 节点

命令返回值为空。

工作模式：管理员

相关选项：无

5.2.8 用户与权限管理

ADD USER

ADD USER 命令增加一个用户。语法：

```
ADD USER user [TYPE type] PASSWORD 'password';
```

参数说明：

- ◆ **user**: 用户名。不能与系统中已有用户名重复
- ◆ **type**: 用户类型，可以是 DBA 或 MAN。DBA 表示管理员用户，可以通过管理员工具对分布式数据库进行管理，MAN 为普通用户，只能通过 JDBC 驱动访问分布式数据库。若不指定则为 MAN 类型。
- ◆ **password**: 密码。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn

DROP USER

DROP USER 命令删除一个用户。语法：

```
DROP USER user;
```

参数说明：

- ◆ **user**: 用户名。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW USERS

SHOW USERS 命令列出分布式数据库中所有用户的名称及类型。语法：

```
SHOW USERS;
```

命令返回值为一个表格，按用户名排序，每一行表示一个用户的信息，包含以下各列：

- ◆ **NAME**: 用户名。
- ◆ **TYPE**: 用户类型。

工作模式：管理员

相关选项：无

GRANT

GRANT 命令给某用户分配权限。语法：


```
GRANT privileges ON [database.]table TO user [WITH GRANT OPTION] [/*
clear_client_conn_pool */];
```

参数说明:

- ◆ **privileges:** 权限
- ◆ **database:** 数据库节点 MySQL 中的 **database**, 通常情况下不需要指定, 这时相当于指定各数据库节点上构成分布式数据库那些 **database**。
- ◆ **table:** 要分配权限的表, 所有表用*表示。
- ◆ **user:** 用户名。该用户必需是已存在的用户, 若不存在, 需要先使用 **ADD USER** 命令增加用户并设置密码。
- ◆ **WITH GRANT OPTION:** 该用户还可以将他的部分权限再分配给其它用户。
- ◆ **clear_client_conn_pool:** 修改后清空 Client 上的连接池。默认为不清空。

其中 **privileges** 为一个列表, 列表中的每项可以是如下的值, 多项之间用,分隔:

- ◆ **READ:** 读权限, 即执行 **SELECT** 语句的权限
- ◆ **WRITE:** 写权限, 即执行 **INSERT/UPDATE/DELETE** 语句的权限
- ◆ **ALL PRIVILEGES:** 所有权限, 包含了 **READ** 和 **WRITE** 权限, 还包括所有其它权限, 如创建/删除表等等

命令返回值为空。

工作模式: 管理员

相关选项: **execute_on_dbn**

REVOKE

REVOKE 命令回收权限。语法:

```
REVOKE privileges ON [database.]tables FROM user [/* clear_client_conn_pool */];
```

参数说明:

- ◆ **privileges:** 权限列表, 格式参见 **GRANT** 命令
- ◆ **database:** 数据库节点 MySQL 中的 **database**, 通常情况下不需要指定, 这时相当于指定各数据库节点上构成分布式数据库那些 **database**。
- ◆ **tables:** 要分配权限的表, 所有表用*表示。
- ◆ **user:** 用户名。该用户必需是已存在的用户。
- ◆ **clear_client_conn_pool:** 修改后清空 Client 上的连接池。默认为不清空。

命令返回值为空。

工作模式: 管理员

相关选项: **execute_on_dbn**

SET PASSWORD

SET PASSWORD 命令设置当前用户或指定用户的密码。语法:

```
SET PASSWORD 'password' [FOR user];
```

参数说明:

- ◆ **password:** 密码。需要使用'括起。

◆ **FOR user:** 设置指定用户的密码，若省略则设置当前用户的密码。

命令返回值为空。

只有管理员才可以指定其它用户的密码，普通用户只能修改自己的密码且不能指定 **FOR user**（即使指定的 **user** 是自己）。

工作模式：管理员、中间件

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW GRANTS

SHOW GRANTS 命令显示当前用户或指定用户的权限。语法：

```
SHOW GRANTS [FOR user];
```

参数说明：

◆ **FOR user:** 显示指定用户的权限。若省略则显示当前用户的权限。

命令返回值为字符串，内容包含该用户权限的所有 **GRANT** 语句。

工作模式：管理员

相关选项：无

ADD HOST

ADD HOST 命令增加可以使用某用户的 IP 地址。语法：

```
ADD HOST hosts TO type OF user;
```

参数说明：

- ◆ **hosts:** 要增加的 IP 地址列表。每个 IP 地址都需要用单引号”括起，多项之间用,分隔，每项可以是：
 - ◆ 一个完整的 IP 地址，形如 192.168.0.1。
 - ◆ 最后分量是%的 IP 地址段，形如 192.168.0.%、192.168.%。
 - ◆ %: 表示所有 IP。
- ◆ **type:** IP 地址类型，共有三种类型：
 - ◆ **DBA:** 指定可以使用 DBA 工具连接到 master 的 IP 地址，只有 DBA 类型的用户才能指定这一类型的 IP 地址。
 - ◆ **CLIENT:** 指定可以通过分布式数据库中间件 JDBC 驱动或轻量级 JDBC 驱动连接到分布式数据库的 IP 地址。
 - ◆ **QS:** 当使用轻量级 JDBC 驱动连接到分布式数据库时，指定它可以连接的查询服务器 IP 地址。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

REMOVE HOST

REMOVE HOST 命令删除可以使用某用户的 IP 地址。语法：

```
REMOVE HOST hosts FROM type OF user;
```

其中各参数的含义同 ADD HOST 命令。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW QUOTA

SHOW QUOTA 命令显示当前用户或指定用户的查询配额。语法：

```
SHOW QUOTA [OF user];
```

参数说明：

◆ **user**: 显示指定用户的查询配额，若省略则显示当前用户的查询配额。

命令返回值为用户的查询配额。

工作模式：管理员

相关选项：无

SET QUOTA

SET QUOTA 命令设置当前用户或指定用户的查询配额。语法：

```
SET QUOTA [OF user] TO value [SLAVEQUOTA TO value2];
```

参数说明：

◆ **user**: 设置指定用户的查询配额，若省略则设置当前用户的查询配额。

◆ **value**: 新的查询配额的值，必须为正整数。

◆ **SLAVEQUOTA**: 当查询语句指定 SLAVEONLY 的 hint 时候（即强制只查询 Slave 节点），用户的配额。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SET DESC

SET DESC 命令设置当前用户或指定用户的描述信息。语法：

```
SET DESC [OF user] 'desc';
```

参数说明：

◆ **user**: 设置指定用户的描述信息，若省略则设置当前用户的描述信息。

◆ **desc**: 新的描述信息，需要用单引号括起。

命令返回值为空。

工作模式：管理员

相关选项：无

SET ROLE

SET ROLE 命令管理员用户的管理权限。语法：

```
SET ROLE [OF user] 'roles';
```

参数说明：

- ◆ **user**: 设置指定用户的管理权限，若省略则设置当前用户的管理权限。
- ◆ **roles**: 新的权限列表，需要用单引号括起。权限总共分为四种角色：

角色包含的功能	功能分类
系 统 配 置 Client 、 DBN、策略、 表、视图、用 户、日志、系 统配置参数	提供只读和管理两种权限
系统维护数据 迁移、备份、 导出、迁移、 清理系统库、 悬挂事务管 理、报警设 置、启动停止 分布式数据 库、计划任务	只提供管理权限
监 控 统 计 Client 统计、 统计任务查看 和统计结果分 析	管理权限在只读权限的基础上 增加统计任务管理
用户管理用户 权限信息	提供只读和管理两种权限

权限列表由四个数字组成，每个数字依次代表系统配置、系统维护、监控统计和用户管理四种角色的功能，每个数字为 0（无权限）或 1（只读权限）或 2（管理权限），数字之间用逗号分隔。例如：命令 **SET ROLE OF admin '1,0,0,0'**，代表设置 **admin** 的管理权限为对系统配置的功能有只读权限，对其它功能都没有权限。

命令返回值为空。

工作模式：管理员

相关选项：无

5.2.9 模式定义

SHOW TABLES

SHOW TABLES 命令显示系统中所有表的基本信息。语法：

```
SHOW TABLES;
```

对于中间件、管理员或轻量级模式，按表名排序，命令输出结果为一个表格，每行表示一个表的信息，包含以下各列：

- ◆ NAME: 表名
- ◆ TYPE: 类型，其中 VIEW 表示是视图，其它的值表示表所用的存储引擎
- ◆ POLICY: 表使用的均衡策略名
- ◆ MODEL: 表所属的模块
- ◆ BF: 均衡字段
- ◆ PKEY: 主键
- ◆ BUCKETNO: 是否含有 bucketno 字段
- ◆ WRITEABLE: 表是否允许写操作
- ◆ NEED_CHECK_DUP_KEY: Insert 操作时候是否需要检查键值重复
- START_ID: 起始 ID，只有管理员模式下才输出这一列
- ◆ REMAIN_ID: 剩余 ID 数，只有管理员模式下才输出这一列
- ◆ COMMENT: 表的注释
- ◆ ID_ASSIGN_TYPE: ID 分配策略，MSB(master 统一批量分配)，TSB(基于时间戳的 ID 分配)

对于并行节点操作模式，对每个节点先返回形如“Results of 数据库节点名”的标识，然后返回后端数据库节点运行“SHOW TABLES”命令的结果，为一个只包含一列，即表名的表格。

工作模式：中间件、轻量级、并行节点操作、管理员

相关选项：无

SHOW TABLES FOR POLICY

SHOW TABLES FOR POLICY 命令显示指定均衡策略中所有表的基本信息。语法：

```
SHOW TABLES FOR POLICY policy_name;
```

参数说明：

- ◆ policy_name: 均衡策略名。

命令返回值与 SHOW TABLES 命令类似，但没有 POLICY 列。

工作模式：中间件、轻量级、管理员

相关选项：无

SHOW TABLES FOR DBN

SHOW TABLES FOR DBN 命令显示使用指定数据库节点的所有表的基本信息。语法：

```
SHOW TABLES FOR DBN name;
```

参数说明：

◆ **name**: 数据库节点名。

命令返回值与 **SHOW TABLES** 命令相同。

工作模式：中间件、轻量级、管理员

相关选项：无

SHOW TABLES FOR MODEL

SHOW TABLES FOR MODEL 命令显示属于指定模块的所有表的基本信息。语法：

```
SHOW TABLES FOR MODEL mdl_name;
```

参数说明：

◆ **mdl_name**: 模块名。

命令返回值与 **SHOW TABLES** 命令相同。

工作模式：中间件、轻量级、管理员

相关选项：无

SHOW VIEWS

SHOW VIEWS 命令显示系统中的视图信息。语法：

```
SHOW VIEWS;
```

命令返回值为一个表格，按视图名排序，每一行表示一个视图的信息，包含以下各列：

- ◆ **NAME**: 视图名。
- ◆ **POLICY**: 视图使用的均衡策略。
- ◆ **DBNTYPE**: 数据库节点类型
- ◆ **BF**: 均衡字段。
- ◆ **TABLES**: 视图所依赖的基本表列表。
- ◆ **SQL**: 视图定义。

工作模式：中间件、轻量级、管理员

相关选项：无

SHOW POLICIES

SHOW POLICIES 命令显示系统中所有均衡策略的基本信息。语法：

```
SHOW POLICIES;
```

输出结果为一个表格，按名称排序，每一行表示一个均衡策略的信息，包含以下各列：

- ◆ **NAME**: 策略名称。
- ◆ **DBNTYPE**: 数据库节点类型

- ◆ BUCKETS: 桶数。
- ◆ DBNS: 均衡策略分布使用的数据库节点。
- ◆ TABLES: 使用该均衡策略的表列表。
- ◆ COMMENT: 均衡策略的注释

工作模式: 中间件、轻量级、管理员

相关选项: 无

DESC

DESC 命令显示指定表的结构信息。语法:

```
DESC name;
```

参数说明:

- ◆ name: 表名。

输出结果为一个表格，每一行表示表中一个属性的信息，包含以下各列:

- ◆ FIELD: 属性名。
- ◆ TYPE: 类型。
- ◆ PRI KEY: 为 YES 表示该属性是主键的一部分。
- ◆ BALANCE: 若该属性是均衡字段则显示该字段在均衡字段列表中的位置，否则为空。
- ◆ EXPR: 若该属性是 Oracle 表的虚拟子度，则显示表达式，否则为空。
- ◆ COMMENT: 字段的注释说明

工作模式: 中间件、轻量级、管理员

相关选项: 无

SHOW INDEX FOR

SHOW INDEX FOR 命令显示 Master 上表定义的索引信息。语法:

```
SHOW INDEX FOR tbl_name;
```

按表名排序，命令输出结果为一个表格，每行表示一个表索引的信息，包含以下各列:

- ◆ NAME: 索引名。
- ◆ COLUMNS: 索引包含的字段列表。
- ◆ PKEY: 是否主键。
- ◆ UNIQUE: 是否唯一。
- ◆ TYPE: 索引的类型
- ◆ MAX_CARDINALITY: 索引中最大的字段区分度。

工作模式: 中间件、轻量级、管理员

相关选项: 无

SHOW CREATE TABLE

SHOW CREATE TABLE 命令显示指定表的建表语句。语法:

```
SHOW CREATE TABLE name;
```

参数说明：

◆ **name**: 表名。

返回值为建表语句。

工作模式：中间件、轻量级

相关选项：无

DESC POLICY

DESC POLICY 命令显示指定均衡策略的定义。语法：

```
DESC POLICY name;
```

参数说明：

◆ **name**: 均衡策略名。

返回值为一个表格，按桶号排序，其中每一行为一个桶的信息，包含以下各列：

- ◆ **NO**: 桶号，从 0 开始。
- ◆ **DBN**: 桶所在的数据库节点名。
- ◆ **URL**: 桶所在数据库节点的 URL。

工作模式：中间件、轻量级、管理员

相关选项：无

CREATE POLICY

CREATE POLICY 命令创建一个均衡策略。语法：

```
CREATE POLICY name dbntype (algorithm)[\ (dbns\)] [n] [comment 'string']
```

参数说明：

- ◆ **name**: 均衡策略名称。
- ◆ **dbntype**: 数据库节点类型，只能为 **oracle|mysql**，不区分大小写。
- ◆ **algorithm**: 均衡策略桶分布算法，可以是：
 - ◆ **RR**: 采用 Round Robin 的方式在 **dbns** 中分布各个桶，如若 **dbns** 为 "db1 db2 db3"，**n** 为 9，则按 RR 方式生成的均衡策略各个桶所在的数据库节点分别为 "db1 db2 db3 db1 db2 db3 db1 db2 db3"
 - ◆ **SR**: 采用 Serial 的方式在 **dbns** 中分布各个桶，如若 **dbns** 为 "db1 db2 db3"，**n** 为 9，则按 SR 方式生成的均衡策略各个桶所在的数据库节点分别为 "db1 db1 db1 db2 db2 db2 db3 db3 db3"
 - ◆ **MAN**: 采用 Manually(人工指定)的方式在 **dbns** 中分布各个桶，如若 **dbns** 为 "db1 db2 db3"，则按 MAN 方式会生成三个桶，分别位于 **db1**、**db2** 和 **db3**
- ◆ **dbns**: 放置均衡策略各桶的数据库节点名称列表，使用空格分隔。若省略 **dbns**，则表示使用系统中所有数据库节。
- ◆ **n**: 均衡策略包含桶数。当使用 **MAN** 均衡策略桶分布算法时，均衡策略各个桶所在节点由 **dbns** 严格给出，不需要指定 **n**。
- ◆ **comment**: 均衡策略的注释说明

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

DROP POLICY

DROP POLICY 命令删除一个均衡策略。语法：

```
DROP POLICY name;
```

参数说明：

◆ **name**: 要删除的均衡策略名称。

只有在均衡策略不被系统中任何表使用时才可以删除。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CHANGE POLICY COMMENT

CHANGE POLICY COMMENT 命令修改策略的注释。语法：

```
CHANGE POLICY COMMENT name 'string';
```

参数说明：

◆ **name**: 需要修改的均衡策略名称。

◆ **String**: 均衡策略的新注释。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CREATE TABLE

CREATE TABLE 命令创建一张表。语法：

```
CREATE TABLE name (  
    ...  
) ... /*BF=balance_field POLICY=policy MODEL=mdl_name  
ASSIGNIDTYPE = MSB|TSB withPersist*/;
```

参数说明见附录。

分布式数据库支持大部分通用的 SQL 建表命令语法，可以用 INDEX 或 KEY 子句来指定索引。建表语句中可以包括 MySQL 中特有的用于将数据库对象名称括起的字符`。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

DROP TABLE

DROP TABLE 命令删除指定的数据库表。语法：

```
DROP TABLE name;
```

参数说明：

◆ **name**: 表名。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

CREATE INDEX

CREATE INDEX 命令创建一个索引，可以是表索引或者是聚簇索引。语法：

MySQL 表：

```
CREATE [UNIQUE] INDEX index_name [USING index_type] ON table_name (col_name, ...)
[/*options*/];
```

Oracle 表：

```
CREATE [UNIQUE|BITMAP] INDEX index_name ON table_name (col_name, ...);
```

参数说明：

- ◆ **index_name**: 索引名称。
- ◆ **index_type**: MySQL 支持的索引类型。
- ◆ **table_name**: 表名。
- ◆ **col_name**: 索引属性名。注：分布式数据库中间件暂不支持 MySQL 中的前缀索引语法。
- ◆ **options**: 一些执行策略和选项，为一个逗号分隔的列表，其中每项可以是：
- ◆ **WITH RENAME**: 在修改之前先将表重命名（表 T 重命名为 T），修改完成后再重命名回来。指定 WITH RENAME 的目的是为了防止阻塞应用。这是由于在修改表模式时 MySQL 会对被修改的表加读锁，这时其它修改该表的操作将被无限期阻塞，直接表修改操作完成为止。如果指定了 WITH RENAME，则同时也意味着指定了 DBN FIRST。
- ◆ **DBN FIRST**: 先在数据库节点上执行 ALTER TABLE 操作，再修改分布式数据库中的模式定义
- ◆ **WITH MYISAM TEMP**: 不直接运行 ALTER TABLE，而是先将数据拷贝到具有相同模式的 MyISAM 类型的临时表中（对于表 T 临时表名为 T_Temp），然后删除清空原表数据并修改表模式，最后将数据拷贝回来。在修改数据量很大的 InnoDB 类型的表时推荐使用这一方法，可防止 InnoDB 数据文件大幅增长
- ◆ **WITH LOCK**: 在分布数据库中间件上对指定表加读锁，这样即可允许读操作继续访问被修改的表，又可即时拒绝写操作，不会造成应用阻塞。理论上指定 WITH LOCK 选

项后即不再需要指定 **WITH RENAME**。

对于 **MySQL** 的表，本命令为 **ALTER TABLE** 命令的一种。上述命令等价于：

```
ALTER TABLE table_name ADD [UNIQUE] INDEX ...
```

该命令也可以作为创建 **Oracle** 索引聚簇的索引

```
CREATE INDEX index_name ON CLUSTER cluster_name [TABLESPACE tbsp_name ...];
```

参数说明：

- ◆ **index_name**: 索引名称。
- ◆ **cluster_name**: 索引聚簇名。
- ◆ **tbsp_name**: 表空间。

对于 **Oracle** 的索引聚簇，必须创建索引后才可以 在聚簇上建表。

命令返回值为空。

工作模式：管理员

相关选项：**execute_on_dbn**、**ignore_failed_clients**、**ignore_unreachable_clients**

DROP INDEX

DROP INDEX 命令删除一个索引，可以是表的索引或者索引聚簇的索引。语法：

```
DROP INDEX index_name ON table_name [/*options*/];
```

参数说明：

- ◆ **index_name**: 索引名称。
- ◆ **table_name**: 表名。
- ◆ **options**: 一些执行策略和选项，格式参见 **CREATE INDEX** 命令。

本命令是 **ALTER TABLE** 命令的一种。上述命令等价于：

```
ALTER TABLE table_name DROP INDEX index_name [/*options*/];
```

对于 **Oracle** 的表索引或者索引聚簇的索引，支持的语法为

```
DROP INDEX index_name [FORCE];
```

参数说明：

- ◆ **index_name**: 索引名称。

命令返回值为空。

工作模式：管理员

相关选项：**execute_on_dbn**、**ignore_failed_clients**、**ignore_unreachable_clients**

ALTER TABLE

ALTER TABLE 命令修改指定表的定义，包括增加/删除字段、增加/删除索引、修改或重命名字段、重命名表名等。语法：

```
ALTER TABLE table_name alter_operation, ... [/*options*/];
alter_operation ::=
    ADD [COLUMN]...
  | DROP [COLUMN]...
  | ADD (INDEX|KEY)...
  | DROP (INDEX|KEY)...
  | CHANGE [COLUMN]...
  | MODIFY...
  | RENAME [TO] ...
```

具体的 **ALTER TABLE** 语法请参见 **MySQL** 手册。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients、online

CHANGE COLUMN COMMENT

CHANGE COLUMN COMMENT 命令修改一个表指定字段的注释。语法：

```
CHANGE COLUMN COMMENT tbl_name cln_name 'string'
```

参数说明：

- ◆ **tbl_name**: 字段所在表的名称。
- ◆ **cln_name**: 字段的名称。
- ◆ **string**: 字段的新注释。

出于性能的考虑，该命令在执行时只修改系统库中所保存的字段注释，而不会在 **DBN** 上执行，也不通知 **client** 进行修改，这会使得数据不一致。

命令返回值为空。

工作模式：管理员

相关选项：无

CHANGE TABLE MODEL

CHANGE TABLE MODEL 命令修改指定表所属的模块。语法：

```
CHANGE TABLE MODEL tbl_name [mdl_name]
```

参数说明：

- ◆ **tbl_name**: 需要修改所属模块的表名称。
- ◆ **mdl_name**: 修改后表所属的模块名称，不能包含中文字符，置空则表不属于任何模块。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CREATE VIEW

CREATE VIEW 命令创建一个视图。语法：

```
CREATE VIEW name AS SELECT ... /*BF=balance_field POLICY=policy*/;
```

参数说明：

- ◆ **name**: 视图名。
- ◆ **balance_field**: 均衡字段名称，必须是视图中存在的字段，且为整数或字符串类型。
- ◆ **policy**: 视图所使用的均衡策略名称，必须与关联的表的均衡策略一致，可省略，默认为关联表的策略。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

DROP VIEW

DROP VIEW 命令删除指定的视图。语法：

```
DROP VIEW name;
```

参数说明：

- ◆ **name**: 视图名。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

ALTER VIEW

ALTER VIEW 命令修改指定视图的定义。语法：

```
ALTER VIEW view_name AS SELECT ... /* BF=balance_field POLICY=policy */;
```

参数说明：

- ◆ **view_name**: 原视图名，不能修改。
- ◆ **balance_field**: 均衡字段名称，必须是视图中存在的字段，且为整数或字符串类型。
- ◆ **policy**: 视图所使用的均衡策略名称，必须与关联的表的均衡策略一致，可省略，默认为关联表的策略。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

ADD BUCKETNO

ADD BUCKETNO 命令增加一个表或者一个策略下表的 **bucketno** 字段。语法:

```
ADD BUCKETNO (tbl_name | OF policy_name) [/ * WITH LOCK */];
```

参数说明:

- ◆ **tbl_name**: 表名。
- ◆ **policy_name**: 均衡策略名称。
- ◆ **with lock**: 增加字段过程中对表加写锁。

命令返回值为空。

工作模式: 管理员

相关选项: **execute_on_dbn**、**ignore_failed_clients**、**ignore_unreachable_clients**

SHOW PARTITIONS

SHOW PARTITIONS 命令查看一个表的水平分区。语法:

```
SHOW PARTITIONS tbl_name [ON db_name];
```

参数说明:

- ◆ **tbl_name**: 表名。
- ◆ **db_name**: 数据库节点名称, 不指定则随机。

命令返回值为表分区字段描述加一个表格, 表格中每行包含一个分区信息, 包含以下两列:

- ◆ **NAME**: 分区名。
- ◆ **VALUES LESS THAN**: 分区间隔最大值, 小于这个值且大于等于前一个分区(若存在)间隔值的属于这个分区。

工作模式: 管理员

相关选项: 无

ADD PARTITIONS

ADD PARTITIONS 命令直接在 DBN 节点增加表的时间水平分区。语法:

```
ADD PARTITIONS tbl_name count;
```

参数说明:

- ◆ **tbl_name**: 表名。
- ◆ **count**: 维持的未来分区数目。若已经有大于 **count** 个未来的分区, 则不会继续添加; 若未来的分区个数小于 **count**, 则会补到 **count** 个未来分区。

此外, 也可以用 **ALTER TABLE tbl_name ADD PARTITION ...** 命令来增加分区。

命令返回值为空。

工作模式: 管理员

相关选项：无

DROP PARTITIONS

DROP PARTITIONS 命令直接在 DBN 节点删除表的水平分区。语法：

```
DROP PARTITIONS tbl_name count;
```

参数说明：

- ◆ **tbl_name**: 表名。
- ◆ **count**: 删除个数，从第一个分区开始删除。

此外，也可以用 ALTER TABLE tbl_name DROP PARTITION ...命令来删除分区。

命令返回值为空。

工作模式：管理员

相关选项：无

DISABLE WRITE TABLE

DISABLE WRITE TABLE 命令禁用 Client 对指定表的写操作。语法：

```
DIABLE WRITE TABLE tbl_name;
```

参数说明：

- ◆ **tbl_name**: 表名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

ENABLE WRITE TABLE

ENABLE WRITE TABLE 命令启用 Client 对指定表的写操作。语法：

```
ENABLE WRITE TABLE tbl_name;
```

参数说明：

- ◆ **tbl_name**: 表名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

DISABLE WRITE POLICY

DISABLE WRITE POLICY 命令禁用 Client 对指定策略下所有表的写操作。语法：

```
DIABLE WRITE POLICY ply_name;
```

参数说明：

◆ **ply_name**: 策略名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

ENABLE WRITE POLICY

ENABLE WRITE POLICY 命令启用 Client 对指定策略下所有表的写操作。语法：

```
ENABLE WRITE POLICY ply_name;
```

参数说明：

◆ **ply_name**: 策略名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SET TABLE IDASSIGNMENT

SET TABLE IDASSIGNMENT 命令设置指定表的起始剩余分配 ID。语法：

```
SET          TABLE          IDASSIGNMENT          tbl_name
[startid=value][,remainid=value][,assigncount=value][,notifyclient=true|false]
```

参数说明：

- ◆ **tbl_name**: 表名。
- ◆ **Startid**: 分配起始 ID
- ◆ **remainid**: 剩余 ID 数
- ◆ **assigncount**: 每次分配的 ID 数。以上 3 个选项最少要有 1 个。
- ◆ **notifyclient**: 是否通知 Client 将剩余 ID 数设为 0
- ◆ **options**: startid 或 remainid 的等值表达式，两个之间用逗号分隔，如“startid=100,remainid=9000”。如果 startid 没有变化，则 notifyclient 参数无作用，下面的两个选项也没有作用。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

REFRESH TABLE STARTID

REFRESH TABLE STARTID 命令记录所有表的 start_id 到系统库，并强制所有 client 更新

表的 **start_id**。迁移时删除脏数据会根据这些 **id** 开始删除。语法：

```
REFRESH TABLE STARTID;
```

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CHANGE TABLE POLICY

CHANGE TABLE POLICY 命令修改表的均衡策略。将使用均衡策略 **p1** 的表 **t1** 改为使用均衡策略 **p2**，需满足如下条件之一：

- ◆ **p1** 和 **p2** 具有相同的桶数，并且两个策略下每个对应的 **DBN** 都相同；
- ◆ **p1** 和 **p2** 具有不同的桶属数，但是两个策略的所有桶都位于一个相同的 **DBN** 上，且表不含有 **bucketno** 字段。

语法：

```
CHANGE TABLE POLICY tbl_name policy_name;
```

参数说明：

- ◆ **tbl_name**: 表名。
- ◆ **policy_name**: 新的策略名。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CREATE TRIGGER

CREATE TRIGGER 命令创建一个触发器。语法：

```
CREATE [DEFINER = { user | CURRENT_USER }] TRIGGER trigger_name trigger_time  
trigger_event ON tbl_name FOR EACH ROW trigger_stmt /*DBNTYPE=type*/
```

参数说明（详见 **MYSQL** 手册）：

- ◆ **trigger_name**: 触发器名，须以 **DDB_** 开头便于区别。
- ◆ **trigger_time**: **BEFORE** | **AFTER**。
- ◆ **trigger_event**: **INSERT** | **UPDATE** | **DELETE**。
- ◆ **tbl_name**: 表名。
- ◆ **trigger_stmt**: 执行语句。
- ◆ **type**: 数据库节点类型，只允许为 **Oracle** 或 **MySQL**，不区分大小写。

命令返回值为空。

工作模式：管理员

相关选项：无

注意：

- ◆ 此命名只会在 **Master** 建立触发器，而不会在 **DBN** 上正式建立，需要利用 **ALTER TRIGGER trigger_name ADD DBN...**命令正式建立到 **DBN** 上。
- ◆ 创建 **TRIGGER** 的时候，由于 **trigger_stmt** 一般都含有 **ISQL** 默认的命令结束符分号“;”，故需要利用 **SET [DELIMITER](#)** 命令重新设置 **ISQL** 的命令结束符。

DROP TRIGGER

DROP TRIGGER 命令删除一个触发器。语法：

```
DROP TRIGGER [IF EXISTS] trigger_name;
```

参数说明：

- ◆ **trigger_name**: 触发器名。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn

ALTER TRIGGER

ALTER TRIGGER 命令修改一个触发器。语法：

```
ALTER TRIGGER trigger_name alter_operation
alter_operation ::=
    ADD DBN "db-name1", "db-name2", ...
  | REMOVE DBN "db-name1", "db-name2", ...
  | SET DESC "desc"
  | SET ENABLED (TRUE | FALSE)
  | SET SQL_DEFINE create_trigger_stmt
```

参数说明：

- ◆ **trigger_name**: 触发器名。
- ◆ **alter_operation**: 可以为如下任意一个：
- ◆ **ADD DBN**: 增加触发器所在的 **DBN** 节点
- ◆ **REMOVE DBN**: 删除触发器所在的 **DBN** 节点
- ◆ **SET DESC**: 设置触发器的描述信息
- ◆ **SET ENABLED**: 启用、禁用触发器
- ◆ **SET SQL_DEFINE**: 利用 [CREATE TRIGGER](#) 命令设置触发器语句，不能修改触发器名称和触发器所在的表。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn

SHOW TRIGGERS

SHOW TRIGGERS 命令查看所有或特定的触发器。语法：

```
SHOW TRIGGERS [LIKE tbl_name] | [FOR trigger_name];
```

参数说明：

- ◆ **tbl_name**: 触发器所在的表名，查看建在指定表上的触发器。
- ◆ **trigger_name**: 触发器名，查看指定的触发器。

对于中间件、管理员或轻量级模式，按触发器名排序，命令输出结果为一个表格，每行表示一个触发器的信息，包含以下各列：

- ◆ **NAME**: 触发器名
- ◆ **TABLE**: 所在表
- ◆ **DBNS**: 所在 DBN 列表
- ◆ **DBNTYPE**: 数据库节点类型
- ◆ **USED**: 是否启用
- ◆ **DESCRIPTION**: 描述信息
- ◆ **SQL**: 创建语句

对于并行节点操作模式，将直接调用 MYSQL 的 SHOW TRIGGERS [LIKE tbl_name]命令返回结果。

工作模式：中间件、轻量级、并行节点操作、管理员

相关选项：无

CREATE PROCEDURE

CREATE PROCEDURE 命令创建一个存储过程。语法：

```
CREATE PROCEDURE sp_name... /*DBNTYPE=type*/
```

参数说明（详见 MYSQL 手册）：

- ◆ **sp_name**: 存储过程名，须以 DDB_开头便与区别。
- ◆ **type**: 数据库节点类型，只允许为 Oracle 或 MySQL，不区分大小写。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

注意：

- ◆ 此命名只会在 Master 建立存储过程，而不会在 DBN 上正式建立，需要利用 ALTER PROCEDURE sp_name ADD DBN...命令正式建立到 DBN 上。
- ◆ 创建 PROCEDURE 的时候，由于存储过程语句一般都含有 ISQL 默认的命令结束符分号“;”，故需要利用 SET [DELIMITER](#) 命令重新设置 ISQL 的命令结束符。

DROP PROCEDURE

DROP PROCEDURE 命令删除一个存储过程。语法：

```
DROP PROCEDURE [IF EXISTS] sp_name;
```

参数说明：

- ◆ **sp_name**: 存储过程名。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

ALTER PROCEDURE

ALTER PROCEDURE 命令修改一个存储过程。语法：

```
ALTER PROCEDURE sp_name alter_operation
alter_operation ::=
    ADD DBN "db-name1","db-name2",...
  | REMOVE DBN "db-name1","db-name2",...
  | SET DESC "desc"
  | SET SQL_DEFINE create_procedure_stmt
```

参数说明：

- ◆ sp_name: 存储过程名。
- ◆ alter_operation: 可以为如下任意一个：
- ◆ ADD DBN: 增加存储过程所在的 DBN 节点
- ◆ REMOVE DBN: 删除存储过程所在的 DBN 节点
- ◆ SET DESC: 设置存储过程的描述信息
- ◆ SET SQL_DEFINE: 利用 [CREATE PROCEDURE](#) 命令设置存储过程语句，不能修改存储过程名称。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW PROCEDURES

SHOW PROCEDURES 命令查看所有或特定的存储过程。语法：

```
SHOW PROCEDURES [FOR sp_name];
```

参数说明：

- ◆ sp_name: 存储过程名，查看指定的存储过程。

对于中间件、管理员或轻量级模式，按存储过程名排序，命令输出结果为一个表格，每行表示一个存储过程的信息，包含以下各列：

- ◆ NAME: 存储过程名
- ◆ DBNS: 所在 DBN 列表
- ◆ DBNTYPE: 数据库节点类型
- ◆ DESCRIPTION: 描述信息
- ◆ SQL: 创建语句

工作模式：中间件、轻量级、管理员

相关选项：无

SHOW ONLINE ALTER TABLE

SHOW ONLINE ALTER TABLE 命令查看所有或特定的在线修改表结构任务。语法：

```
SHOW ONLINE ALTER TABLE [OF task_id];
```

参数说明：

◆ task_id: 任务 id。

命令输出结果为一个表格，每行表示一个任务信息，包含以下各列：

- ◆ ID: 在线修改表结构任务 ID
- ◆ CREATE_TIME: 在线修改表结构任务创建时间
- ◆ LAST_UPDATE: 任务状态最后更新时间
- ◆ TABLE_NAME: 所修改的表名
- ◆ SQL: 执行的 SQL 语句
- ◆ TRUNK_SIZE: 拷贝段大小
- ◆ SLEEP_TIME: 间隔时间
- ◆ STATUS: 任务状态

工作模式：管理员

相关选项：无

START ONLINE ALTER TABLE

START ONLINE ALTER TABLE 命令启动一个在线修改表结构任务。语法：

```
START ONLINE ALTER TABLE task_id;
```

参数说明：

◆ task_id: 任务 id。

工作模式：中间件、轻量级、管理员

相关选项：无

STOP ONLINE ALTER TABLE

STOP ONLINE ALTER TABLE 命令停止一个在线修改表结构任务。语法：

```
STOP ONLINE ALTER TABLE task_id ;
```

参数说明：

◆ task_id: 任务 id。

工作模式：中间件、轻量级、管理员

相关选项：无

CANCEL ONLINE ALTER TABLE

CANCEL ONLINE ALTER TABLE 命令撤销一个在线修改表结构任务。语法：

```
CANCEL ONLINE ALTER TABLE task_id;
```

参数说明：

◆ **task_id**: 任务 id。

工作模式：中间件、轻量级、管理员

相关选项：无

DROP ONLINE ALTER TABLE

DROP ONLINE ALTER TABLE 命令启动一个在线修改表结构任务。语法：

```
DROP ONLINE ALTER TABLE tasks;
```

参数说明：

◆ **task_id**: 任务 id 列表。

工作模式：中间件、轻量级、管理员

相关选项：无

SET ONLINE ALTER TABLE

SET ONLINE ALTER TABLE 命令设置一个在线修改表结构任务的命令参数，可以在任务正在执行时操作。语法：

```
SET ONLINE ALTER TABLE ID=... TRUNKSIZE=... SLEEPTIME=...;
```

参数说明：

◆ **id**: 任务 id。

◆ **TRUNKSIZE**: 拷贝段大小。

◆ **SLEEPTIME**: 间隔时间。

工作模式：中间件、轻量级、管理员

相关选项：无

CREATE CLUSTER

CREATE CLUSTER 命令创建一个聚簇。语法：

```
CREATE CLUSTER name (  
    ...  
) ... SIZE integer[K|M]/*POLICY=policy */;
```

参数说明：

◆ **name**: 聚簇名。

◆ **integer**: 聚簇的 SIZE，必须为正整数。

◆ **policy**: 表所使用的均衡策略名称，必须是系统中已有的 Oracle 均衡策略。

分布式数据库支持 Oracle 的索引聚簇以及散列聚簇，具体语法可以参考 Oracle 手册。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

ALTER CLUSTER

ALTER CLUSTER 命令对聚簇进行修改，目前只支持对聚簇 SIZE 参数的修改。语法：

```
ALTER CLUSTER name SIZE integer[K|M];
```

参数说明：

- ◆ name: 聚簇名。
- ◆ integer: 聚簇的 SIZE，需为正整数。

当前只支持对聚簇 SIZE 的修改，SIZE 默认的单位为 BYTE，具体的最大限制由节点定义。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW DBNCLUSTERS

SHOW DBNCLUSTERS 命令显示系统中所有聚簇的基本信息。语法：

```
SHOW DBNCLUSTERS;
```

对于中间件、管理员或轻量级模式，按聚簇名排序，命令输出结果为一个表格，每行表示一个聚簇的信息，包含以下各列：

- ◆ NAME: 聚簇名。
- ◆ TYPE: 类型，INDEX CLUSTER 表示索引聚簇，HASH CLUSTER 表示散列聚簇。
- ◆ TABLES: 表名，指建立在该聚簇上的表，以逗号相隔，为空则表示还没有添加表。
- ◆ POLICY: 聚簇使用的均衡策略名。
- ◆ SIZE: 聚簇的 SIZE 参数值。
- ◆ INDEX_NAME: 聚簇上的索引的名称，对于索引聚簇，该索引需要通过 CREATE INDEX 命令添加，而散列聚簇则在创建的同时会自动补充上索引名。

工作模式：中间件、轻量级、管理员

相关选项：无

DESC DBNCLUSTER

DESC DBNCLUSTER 命令显示指定聚簇的字段定义信息。语法：

```
DESC DBNCLUSTER name;
```

参数说明：

- ◆ name: 聚簇名。

输出结果为一个表格，每一行表示表中一个属性的信息，包含以下各列：

- ◆ **SEQ**: 字段顺序编号。
- ◆ **NAME**: 字段名。
- ◆ **DATA_TYPE**: 数据类型，指创建聚簇时所指定的数据类型。

工作模式：中间件、轻量级、管理员

相关选项：无

DROP CLUSTER

DROP CLUSTER 命令对聚簇进行删除。语法：

```
DROP CLUSTER name [INCLUDING TABLES];
```

参数说明：

- ◆ **name**: 聚簇名。

INCLUDING TABLES 表示是否将关联的表一并删除。当聚簇上存在关联表时，如果没有使用 **INCLUDING TABLES** 选项，则会返回异常并拒绝执行。

命令返回值为空。

工作模式：管理员

相关选项：execute_on_dbn、ignore_failed_clients、ignore_unreachable_clients

SHOW NTSE TABLE PARAM

SHOW NTSE TABLE PARAM 命令查看指定 NTSE 表的表配置。语法：

```
SHOW NTSE TABLE PARAM FOR name;
```

参数说明：

- ◆ **name**: NTSE 表名。

命令的输出结果为一个表格，包含以下字段。

- ◆ **KEY**: 配置参数名
- ◆ **VALUE**: 配置参数值

工作模式：管理员

相关选项：无

SHOW NTSE COLUMN PARAM

SHOW NTSE COLUMN PARAM 命令查看指定 NTSE 表的字段配置。语法：

```
SHOW NTSE COLUMN PARAM FOR name;
```

参数说明：

- ◆ **name:** NTSE 表名。

命令的输出结果为一个表格，包含以下字段。

- ◆ **COLUMN:** 字段名
- ◆ **KEY:** 配置参数名
- ◆ **VALUE:** 配置参数值

工作模式：管理员

相关选项：无

5.2.10 导入导出与迁移

SHOW SCHEMA

SHOW SCHEMA 命令导出分布式数据库的模式定义。语法：

```
SHOW SCHEMA;
```

命令结果包括数据库节点、均衡策略和表定义，不包括权限等其它信息。

工作模式：中间件、轻量级

相关选项：无

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。

命令输出了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

BACKUP

BACKUP 命令使用 MySQL 的 Hot-Backup 工具备份所有存储节点中的指定数据。语法：

```
BACKUP [OPTIONS] (all | tables);
```

参数说明：

- ◆ **OPTIONS**: 选项。可用如下选项：
- ◆ **--exclude-mysam**: 不同时备份 `mysam` 表，默认为同时备份 `mysam` 表，若同时备份 `mysam` 表，则不允许“`compress`”和“`sync-client`”选项。
- ◆ **--compress**: 压缩备份的数据。
- ◆ **--execute-scripts**: 备份完成后执行在 **Agent** 中配置的脚本。
- ◆ **--clear-old-data n**: 备份完成后删除 `n` 天前的备份数据。
- ◆ **--sync-client**: 备份完成时通知所有客户端停止写操作，完成后再开放写操作，使用这一选项可以保证备份的数据反映的是备份结束时全局一致状态（仅对 `InnoDB` 类型的表可做此保证）。
- ◆ **all**: 备份所有表。
- ◆ **tables**: 指定要备份的一个或多个表，为一个逗号分隔的列表，每一项为一个表名。

命令返回值为空。

工作模式：管理员

相关选项：`ignore_failed_clients`、`ignore_unreachable_clients`

EXPORT

EXPORT 命令使用 MySQL 的 `mysqldump` 工具导出所有存储节点中的指定数据。语法：

```
EXPORT [OPTIONS] (all | tables)
```

参数说明：

- ◆ **OPTIONS**: 选项。可用如下选项：
- ◆ **--where='where-condition'**: 条件语句。
- ◆ **--csv**: 导出为 `CSV` 格式，若不指定这一选项则导出为 `INSERT` 语句。
- ◆ **--fields-terminated-by='...'**、**--fields-enclosed-by='...'**、**--lines-terminated-by='...'**: 当指定格式为 `csv` 文本时候的间隔符号，在“`LOAD DATA INFILE`”导入时候有用。
- ◆ **--compress**: 压缩导出的数据。
- ◆ **--execute-scripts**: 导出完成后执行在 **Agent** 中配置的脚本。
- ◆ **--clear-old-data n**: 导出完成后删除 `n` 天前的备份数据。
- ◆ **--ignore-check**: 检查导出有效性失败后是否仍然继续后续的导出操作。
- ◆ **--ignore-error**: 导出失败后是否继续执行脚本并删除旧数据。
- ◆ **all**: 导出所有表。
- ◆ **tables**: 指定要导出的一个或多个表，为一个逗号分隔的列表，每一项为一个表名。

EXPORT 命令与 DUMP 的区别是 EXPORT 是将数据导出在服务器端，而 DUMP 则是将数据导出在客户端。

命令返回值为空。

工作模式：管理员

相关选项：无

EXPORT SYSDB

EXPORT SYSDB 命令使用 MySQL 的 `mysqldump` 工具导出系统库到分布式数据库 MASTER 所在服务器上。语法：

```
EXPORT SYSDB TO dir;
```

选项说明：

- ◆ **dir**: 导出到指定目录。需要注意的是 **dir** 是分布式数据库 MASTER 所在服务器中的路径

命令返回值为空。

工作模式：管理员

相关选项：无

SINGLE MIGRATE

SINGLE MIGRATE 命令以桶为单位顺序进行数据迁移。语法：

```
SINGLE MIGRATE (bucketnos OF policy TO des_db [/* ONLINE select_order */]) | (WITH task_id);
```

参数说明：

- ◆ **bucketnos**: 桶号列表，已逗号分隔，每项为桶的数字号
- ◆ **policy**: 策略名称
- ◆ **des_db**: 目的节点 DBN 名称
- ◆ **task_id**: 执行系统中 id 为 **task_id** 的迁移操作

选项说明：

- ◆ **ONLINE**: 在线迁移，从源数据库 **select** 记录再 **insert** 到目的数据库。默认为 **OFFLINE**，通过 Agent 执行 **SELECT INTO OUTFILE** 及 **LOAD DATA LOCAL INFILE** 来完成迁移。
- ◆ **select_order**: **NO_ORDER** | **ASCEND_ORDER** | **DESCEND_ORDER**，当类型为在线迁移时候有作用，对 **select** 结果的排序，减少出错。默认为 **NO_ORDER**。

命令返回值为空，在线迁移会在标准输出打印迁移进度。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

CONCURRENT MIGRATE

CONCURRENT MIGRATE 命令实现并行的离线数据迁移，多个源数据库并行执行基于 **myisam** 临时表的数据迁移。语法：

```
CONCURRENT MIGRATE (bucketnos OF policy TO des_db [WITH_RESET] [, ...]) | (WITH task_ids);
```

参数说明：

- ◆ **bucketnos**: 桶号列表，已逗号分隔，每项为桶的数字号
- ◆ **policy**: 策略名称
- ◆ **des_db**: 目的节点 DBN 名称
- ◆ **WITH_RESET**: 重建源数据库表，先 **truncate** 源表，再从临时表选择数据插入
- ◆ **task_ids**: 执行 **task_ids** 列表中的系统中存在的迁移操作

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SHOW MIGRATE

SHOW MIGRATE 命令显示系统中的所有或者某类迁移操作（利用复制的在线迁移除外）。

语法：

```
SHOW MIGRATE [OF type];
```

参数说明：

- ◆ **type**: SINGLE | CONCURRENT

命令返回值为一个表格，每行包含一个迁移记录的信息，包含以下各列：

- ◆ **ID**: 迁移记录的 ID，唯一标识。
- ◆ **TYPE**: 迁移类型，分为 **ONLINE**（在线顺序迁移）、**OFFLINE**（离线顺序迁移）、**CONCURRENT**（并发离线迁移）。
- ◆ **POLICY**: 迁移数据所属的策略名。
- ◆ **BUCKETNO**: 迁移的桶号列表。
- ◆ **SRC_DB**: 迁移的源数据库。
- ◆ **DES_DB**: 迁移的目的数据库。
- ◆ **STATUS**: 迁移的状态，主要有未开始、成功完成、失败完成、执行中等。
- ◆ **START_TIME**: 迁移开始的时间。
- ◆ **END_TIME**: 迁移结束时间。
- ◆ **FINISH**: 迁移完成记录数，针对在线迁移有效。
- ◆ **TOTAL**: 总共需要迁移的记录数，针对在线迁移有效。

工作模式：管理员

相关选项：无

DROP MIGRATE

DROP MIGRATE 命令删除系统中的指定的迁移操作。语法：

```
DROP MIGRATE task_ids;
```

参数说明：

- ◆ **task_ids**: 迁移任务 id，以逗号分隔

命令返回值为空。

工作模式：管理员

相关选项：无

STOP SINGLE MIGRATE

STOP SINGLE MIGRATE 命令停止正在进行的顺序在线迁移操作。语法：

```
STOP SINGLE MIGRATE;
```

命令返回值为空。

工作模式：管理员

相关选项：无

STOP CONCURRENT MIGRATE

STOP CONCURRENT MIGRATE 命令停止系统中出错的并发离线迁移操作，使得系统和 Client 恢复到正常状态。语法：

```
STOP CONCURRENT MIGRATE;
```

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SHOW ONLINE MIGRATE

SHOW ONLINE MIGRATE 命令显示系统中的所有或指定的利用复制进行的在线迁移操作。语法：

```
SHOW ONLINE MIGRATE [OF task_id];
```

参数说明：

◆ **task_id**: 迁移任务 ID，只显示指定的迁移任务。

命令返回值为一个表格，每行包含一个迁移记录的信息，包含以下各列：

- ◆ **ID**: 迁移记录的 ID，唯一标识。
- ◆ **SOURCE_DB**: 源数据库名称列表。
- ◆ **DESTINATION_DB**: 目的数据库名称列表。
- ◆ **ADDITIONAL_DB**: 额外辅助数据库名称列表。
- ◆ **STATUS**: 当前迁移状态。
- ◆ **POLICY**: 迁移的策略名称列表。
- ◆ **VALID_START_TIME**: 允许的迁移开始时间。
- ◆ **VALID_END_TIME**: 允许的迁移结束时间。
- ◆ **REP_TIMEOUT**: 等待复制同步的超时时间。
- ◆ **STEPS**: 迁移步骤。

工作模式：管理员

相关选项：无

ONLINE MIGRATE

ONLINE MIGRATE 命令创建一个基于复制的在线迁移任务。语法：

```
ONLINE MIGRATE [OPTIONS] POLICY policy_names FROM db_info,[db_info2,...] TO
db_info,[db_info2,...] MIGSCALE =(1|2|3|...)
```

参数说明：

- ◆ **OPTIONS**: 选项。可用如下选项：
- ◆ **--rep_timeout=value**: 等待复制同步的超时时间，正整数，单位为秒，默认为 1000 秒。若在此时间内镜像数据库不能同步源数据库，则迁移失败。
- ◆ **policy_names**: 分裂策略名称列表，用逗号分隔。
- ◆ **db_info**: 指定数据库，格式为：
db_name WITH DATA_DIR data_dir
- ◆ **db_name**: 数据库名称。
- ◆ **data_dir**: 数据库的数据目录
- ◆ **FROM**: 指定迁移源数据库列表。
- ◆ **TO**: 指定迁移目的数据库列表。
- ◆ **MIGSCALE**: 扩容比例。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

START ONLINE MIGRATE

START ONLINE MIGRATE 命令启动一个在线迁移任务，任务将在 Maser 后台执行，可以通过查看迁移日志（[SHOW LOG migrate](#)）来了解详情。同一时间只能运行一个在线迁移任务。语法：

```
START ONLINE MIGRATE task_id;
```

参数说明：

- ◆ **task_id**: 在线迁移任务的 ID。

命令返回值为空。

工作模式：管理员

相关选项：无

STOP ONLINE MIGRATE

STOP ONLINE MIGRATE 命令停止一个在线迁移任务，可以通过查看迁移日志（[SHOW LOG migrate](#)）来了解详情。语法：

```
STOP ONLINE MIGRATE task_id;
```

参数说明：

◆ **task_id**: 在线迁移任务的 ID。

命令返回值为空。

工作模式: 管理员

相关选项: 无

DROP ONLINE MIGRATE

DROP ONLINE MIGRATE 命令删除在线迁移任务。语法:

```
DROP ONLINE MIGRATE task_ids;
```

参数说明:

◆ **task_ids**: 在线迁移任务的 ID 列表, 以逗号分隔。

命令返回值为空。

工作模式: 管理员

相关选项: 无

SLICEDMASSUPDATE

SLICEDMASSUPDATE 命令根据主键分批对表进行 Update 或 Delete 数据的操作, 每批执行后会等待一段时间再执行下一批, 从而不会一直锁住表而对应用产生较大影响。注意: 执行此操作的话, 表必须含有主键, 并且主键必须为整型。语法:

```
slicedMassUpdate [OPTIONS] table condition;
```

参数说明:

- ◆ **OPTIONS**: 选项。可用如下选项:
- ◆ **--start value**: 开始 (\geq) 的主键值, 为整型, 默认 0。
- ◆ **--end value**: 结束 ($<$) 的主键值, 为整型, 默认 10000000000000。
- ◆ **--slice-size value**: 每批操作执行的主键范围大小, 默认 10000。
- ◆ **--sleep-ratio value**: 每批执行后睡眠时间与上批执行时间的比值, 比值越大, 睡眠越久, 默认 1.0。
- ◆ **--duration value**: 操作执行时间长度限制, 超过此限制则自动退出。单位: 秒。默认 14400 (4 小时)。
- ◆ **--index value**: 进行批量更新操作时的遍历基准索引名称。依据该索引进行数据分段。
- ◆ **--delete | update set_clause**: 数据操作类型: delete 或者 update。Update 操作需要执行 set 语句, 用双引号括起。
- ◆ **table**: 表名, 必须含有主键, 并且主键必须为整型。
- ◆ **condition**: where 条件, 用双引号括起。

命令返回值为空。

工作模式: 管理员

相关选项: 无

5.2.11 运行状态监控

SHOW CLIENTS

SHOW CLIENTS 命令显示当前系统中维护的所有客户端信息。语法：

```
SHOW CLIENTS;
```

命令返回值为一个表格，按客户端类型和 IP 排序，包含以下各列：

- ◆ ID: 客户端编号。
- ◆ NAME: 客户端名称。
- ◆ IP: 客户端 IP。
- ◆ PORT: 客户端用于监听 MASTER 连接请求的端口。
- ◆ TYPE: 客户端类型，其中 QS 表示为查询服务器，CLIENT 表示为一般中间件。
- ◆ STATUS: 客户端当前状态。
- ◆ CONNECT TIME: 客户端与 MASTER 建立连接的时间（通常就是客户端的启动时间）。
- ◆ LAST ACTIVE: 客户端最近一次访问 MASTER 的时间。
- ◆ USERS: 使用该客户端的用户列表。

工作模式：管理员

相关选项：无

ADD MONITOR CLIENTS

ADD MONITOR CLIENTS 命令增加一个或多个客户端参与监控。语法：

```
ADD MONITOR CLIENTS clients;
```

参数说明：

- ◆ clients: 客户端列表，为一个逗号分隔的列表，其中每一项可以是：
- ◆ 一个数字: 表示要增加的客户端 ID
- ◆ 'ip[:port]': 表示要增加的客户端的 IP 或 IP 与端口。若没有指定端口，则会增加所有指定 IP 的客户端，可能有多个。注意 IP 或 IP:PORT 必须用单引号括起。
- ◆ 一个字符串: 表示要增加的客户端的名称，由于客户端名称不要求唯一，因此可能会增加多个客户端

命令返回值为空。

通过 ADD MONITOR CLIENTS 增加了某些客户端之后，本节以下的所有监控命令（SHUTDOWN 除外）默认都会在这些客户端上操作（假如没有使用"FOR clients"子句另外指定要操作的客户端）。可以使用 SHOW MONITOR CLIENTS 命令显示当前已经指定参与监控的客户端列表。

工作模式：管理员

相关选项：无

REMOVE MONITOR CLIENTS

REMOVE MONITOR CLIENTS 命令从监控客户端集合中删除一个或多个客户端。语法：

```
REMOVE MONITOR CLIENTS clients;
```

参数说明：

◆ **clients**: 要删除的客户端列表，格式与 **ADD MONITOR CLIENTS** 命令相同

命令返回值为空。

工作模式：管理员

相关选项：无

SHOW MONITOR CLIENTS

SHOW MONITOR CLIENTS 命令显示当前已经指定参与监控的客户端列表。语法：

```
SHOW MONITOR CLIENTS;
```

显示结果为一个表格，格式与 **SHOW CLIENTS** 命令结果相同。本节以下的所有监控命令（**SHUTDOWN** 除外）默认都会在这些客户端上操作（假如没有使用“**FOR clients**”子句另外指定要操作的客户端）。

工作模式：管理员

相关选项：无

SHOW OPS

SHOW OPS 命令显示指定客户端的操作统计信息。语法：

SHOW OPS [FOR clients]

参数说明：

◆ **clients**: 另外指定要操作的客户端列表，格式与 **ADD MONITOR CLIENTS** 命令相同。若不指定“**FOR clients**”，则会操作前先用 **ADD MONITOR CLIENTS** 命令指定的监控客户端集。当前的监控客户端集可用 **SHOW MONITOR CLIENTS** 命令查看。

显示结果为一个表格，包含以下各列：

- ◆ **CLIENT**: 客户端 ID。
- ◆ **Operation**: 操作名称。
- ◆ **Count**: 操作执行次数。
- ◆ **Time(ms)**: 操作总执行时间，单位为毫秒。
- ◆ **Avg(ms)**: 操作平均执行时间，单位为毫秒。

包含以下操作：

- ◆ **CONN**: 所有应用要求分布式数据库中间件进行的连接相关操作（不是指 **DDB** 执行语句过程中对后端数据库的连接操作），包括以下子操作：
 - ◆ **CONN_GET**: 建立连接操作
 - ◆ **CONN_FREE**: 关闭连接操作
- ◆ **TXN**: 所有应用要求分布式数据库中间件进行的事务操作（注：**AUTO COMMIT** 模式下只访问一个后台节点的语句在分布式数据库中间件中并不开始事务），包括以下子操作：
 - ◆ **TXN_BEGIN**: 开始事务操作
 - ◆ **TXN_END**: 结束事务操作，包括以下子操作：
 - ◆ **TXN_COMMIT**: 提交事务操作

- ◆ **TXN_ROLLBACK**: 回滚事务操作
- ◆ **PS**: 所有应用要求分布式数据库中间件进行的 **PreparedStatement** 相关操作，包括以下子操作：
 - ◆ **PS_GET**: 创建一个 **PreparedStatement**
 - ◆ **PS_FREE**: close 一个 **PreparedStatement**
 - ◆ **PS_CACHE**: 语法树缓存相关操作。DDB 缓存符合条件的 **PreparedStatement** 对应的优化后的语法树以降低中间件在语法解析和查询优化方面的开销。
 - ◆ **PS_CACHE_HIT**: 命中语法树缓存
 - ◆ **PS_CACHE_MISS**: 语法树缓存不命中
 - ◆ **PS_CACHE_PRUNE**: 语句由于不符合被缓存条件而没有放到语法树缓存中
- ◆ **SQL**: 所有应用要求分布式数据库中间件进行的 **SQL** 操作，包括以下子操作：
 - ◆ **EXECUTE**: 不指定操作类型的执行操作（即 JDBC 中直接调用 **execute()**，而非调用 **executeQuery()**或 **executeUpdate()**）
 - ◆ **QUERY**: 查询操作（即 JDBC 中调用 **executeQuery()**）
 - ◆ **MODIFY**: 更新操作（即 JDBC 中调用 **executeUpdate()**），包括以下子操作：
 - ◆ **INSERT**: 通过 **executeUpdate()**执行 **INSERT** 语句
 - ◆ **UPDATE**: 通过 **executeUpdate()**执行 **UPDATE** 语句
 - ◆ **DELETE**: 通过 **executeUpdate()**执行 **DELETE** 语句
 - ◆ **LOGGING**: 中间件写分布式事务日志操作，一个分布式要写两次日志
 - ◆ **RECOVER**: 分布式事务故障恢复操作。注意：分布式数据库中间件启动时都要进行故障恢复，因此如果系统显示分布式事务故障恢复次数为 1，则实际上并没有分布式事务故障产生
- ◆ **CONN_POOL**: 所有连接分布式数据库中间件执行过程中对连接到后端数据库节点的连接池进行的操作，包括以下子操作：
 - ◆ **CP_GET**: 获取一个普通连接，普通连接用于执行 **AUTO COMMIT** 模式下的非分布式事务
 - ◆ **CP_FREE**: 释放一个普通连接
 - ◆ **XACP_GET**: 获取一个 **XA** 连接，**XA** 连接用于执行分布式事务（注：所有非 **AUTO COMMIT** 模式执行的事务都被认为是分布式事务，即使它实际上只访问了一个数据库节点）
 - ◆ **XACP_FREE**: 释放一个 **XA** 连接
- ◆ **THREAD_POOL**: 分布式数据库中间件内部用于执行对某个后端数据库节点操作的线程池的所有相关操作，包括以下子操作：
 - ◆ **TP_EXEC**: 普通执行线程（执行 **SQL** 操作的线程）的使用情况
 - ◆ **TXN_TP_EXEC**: 事务处理执行线程（执行事务提交/回滚等操作的线程）的使用情况
- ◆ **DBN_CONN**: 后端数据库节点连接相关操作，包括以下子操作：
 - ◆ **DBN_CONN_GET**: 创建到某个后端数据库节点的连接
 - ◆ **DBN_CONN_FREE**: 关闭到某个后端数据库节点的连接
- ◆ **DBN_TXN**: 对后端数据库节点进行的所有事务相关操作，包括以下子操作：
 - ◆ **DBN_BEGIN**: 开始事务
 - ◆ **DBN_PREPARE**: 使用二阶段提交时准备事务
 - ◆ **DBN_END**: 结束事务，包括以下子操作：
 - ◆ **DBN_COMMIT**: 提交事务
 - ◆ **DBN_ROLLBACK**: 回滚事务
 - ◆ **DBN_SQL**: 对后端数据库节点进行的所有 **SQL** 操作，包括以下子操作：
 - ◆ **DBN_EXECUTE**: 不指定操作类型的执行操作（即 JDBC 中直接调用 **execute()**，而非调用 **executeQuery()**或 **executeUpdate()**）
 - ◆ **DBN_QUERY**: 查询操作（即 JDBC 中调用 **executeQuery()**）
 - ◆ **DBN_MODIFY**: 更新操作（即 JDBC 中调用 **executeUpdate()**），包括以下子操作：
 - ◆ **DBN_INSERT**: 通过 **executeUpdate()**执行 **INSERT** 语句
 - ◆ **DBN_UPDATE**: 通过 **executeUpdate()**执行 **UPDATE** 语句

◆ **DBN_DELETE**: 通过 `executeUpdate()` 执行 **DELETE** 语句

工作模式: 管理员

相关选项: 无

RESET OPS

SHOW OPS 命令重置指定客户端的操作统计信息。语法:

```
RESET OPS [FOR clients]
```

参数说明:

◆ **clients**: 另外指定要操作的客户端列表, 格式参见 **SHOW OPS** 命令。

命令返回值为空。

工作模式: 管理员

相关选项: 无

SHOW RESOURCE STATUS

SHOW RESOURCE STATUS 命令显示指定客户端的数据库资源的使用情况汇总信息。语法:

```
SHOW RESOURCE STATUS [FOR clients];
```

参数说明:

◆ **clients**: 另外指定要操作的客户端列表, 格式参见 **SHOW OPS** 命令。

命令返回值为一个表格, 包含以下各列:

- ◆ **CLIENT**: 客户端 ID。
- ◆ **RESOURCE**: 资源类型, 其中 **Connection** 表示 DDB 连接, **Statement** 表示 DDB Statement、**PreparedStatement** 表示 DDB PreparedStatement。
- ◆ **CREATE**: 资源创建次数。
- ◆ **CLOSE**: 资源关闭次数。
- ◆ **IMPLICIT_CLOSE**: 隐式关闭次数, 即没有调用 **close** 关闭而是在垃圾收集时才关闭。
- ◆ **HANGING_CLOSE**: 悬挂关闭次数, 即很久没有使用, 被悬挂资源检测线程强行关闭。

工作模式: 管理员

相关选项: 无

SHOW ACTIVE

SHOW ACTIVE 命令显示指定客户端的指定类型的活跃资源信息。语法:

```
SHOW ACTIVE OF type [FOR clients];
```

参数说明:

- ◆ **type**: 指定要查看的资源类型，可以是：
- ◆ **CONN**: DDB 连接。
- ◆ **STMT**: DDB Statement。
- ◆ **PS**: DDB PreparedStatement。
- ◆ **clients**: 另外指定要操作的客户端列表，格式参见 **SHOW OPS** 命令。

命令返回值为一个表格，每行包含一个活跃资源的信息，包含以下各列：

- ◆ **CLIENT**: 资源所在客户端的 ID。
- ◆ **PID**: 父资源，对于 DDB 连接资源，PID 一定是空，对于 DDB Statement 和 PreparedStatement 资源，PID 表示其所属 DDB 连接资源的 ID。
- ◆ **ID**: 资源的 ID。各类资源的 ID 是从系统启动开始顺序编号的，从 1 开始，不会重用。
- ◆ **STATUS**: 资源的使用状态，可为 **Idle** 或 **Active**。其中 **Active** 表示资源正在用于执行数据库操作，**Idle** 表示不在用于执行数据库操作。
- ◆ **CREATE TIME**: 资源创建时间。
- ◆ **DURATION**: 资源保持在当前状态下的持续时间，单位为秒。因此如资源状态为 **Active**，则 **DURATION** 表示当前正在执行的数据库操作执行了多久了，如资源状态为 **Idle**，则表示资源已经多长时间没有用来执行数据库操作了。
- ◆ **OP COUNT**: 利用该资源进行的数据库操作次数，次数是在操作执行之前统计的，因此即使一个操作还没完成也已经被计入计数。连接资源的操作次数包含该连接所有 **Statement** 或 **PreparedStatement** 资源操作，还包括事务提交回滚等操作。
- ◆ **EXTRA**: 关于资源的其它额外信息，目前的实现中如果资源是被使用轻量级 **JDBC** 连接的客户端占用则会显示客户端的 IP，否则为空。
- ◆ **OP INFO**: 利用该资源进行的最近 5 次数据库操作的 SQL 语句，用;分隔。

工作模式：管理员

相关选项：无

SHOW IC

SHOW IC 命令显示指定客户端的指定类型的资源被隐式关闭的信息。语法：

```
SHOW IC OF type [FOR clients];
```

参数说明：

- ◆ **type**: 指定要查看的资源类型，具体格式参见 **SHOW ACTIVE** 命令
- ◆ **clients**: 另外指定要操作的客户端列表，格式参见 **SHOW OPS** 命令。

命令返回值为一个表格，格式与 **SHOW ACTIVE** 命令类似，但没有 **STATUS** 列。

资源被隐式关闭是指该资源应用没有调用 **close** 方法关闭，但后来被 **Java** 虚拟机垃圾回收。

SHOW HC

SHOW HC 命令显示指定客户端的指定类型资源中由于长时间不处于活动状态，被悬挂资源检测线程清理的信息。语法：

```
SHOW HC OF type [FOR clients];
```

参数说明：

- ◆ **type**: 指定要查看的资源类型，具体格式参见 **SHOW ACTIVE** 命令

◆ **clients**: 另外指定要操作的客户端列表，格式参见 **SHOW OPS** 命令。

命令返回值为一个表格，格式与 **SHOW ACTIVE** 命令类似，但没有 **STATUS** 列。

工作模式：管理员

相关选项：无

SHOW DBN CONN

SHOW DBN CONN 命令显示指定客户端连接到后面数据库的底层连接池信息。语法：

```
SHOW DBN CONN [FOR clients];
```

参数说明：

◆ **clients**: 另外指定要操作的客户端列表，格式参见 **SHOW OPS** 命令。

命令返回值为一个表格，包含以下各列：

- ◆ **CLIENT**: 客户端 ID。
- ◆ **XA?**: 该连接是否为 **XA** 连接。
- ◆ **#**: 一个临时编码，方便看总共有多少个连接，每次 **SHOW** 时都重新编码，不一定相同。
- ◆ **USER**: 使用该连接的用户。
- ◆ **LWIP**: 如果是查询服务器上的数据库连接且被某轻量级驱动的客户端占用则显示该客户端的 IP，否则为空。
- ◆ **STATUS**: 连接状态。可为以下值：
- ◆ **Free**: 连接空闲。
- ◆ **Active**: 连接正在用于执行数据库操作
- ◆ **Idle**: 连接已经分配给某应用使用，但当前不在执行数据库操作
- ◆ **DURATION**: 资源保持在当前状态下的持续时间，单位为秒。因此如资源状态为 **Active**，则 **DURATION** 表示当前正在执行的数据库操作执行了多久了，如资源状态为 **Idle**，则表示资源已经多长时间没有用来执行数据库操作了。
- ◆ **URL**: 连接到哪个数据库。
- ◆ **SQL**: 该连接最后一次执行的数据库操作。

工作模式：管理员

相关选项：无

SHOW ACTIVE DBN CONN

SHOW ACTIVE DBN CONN 语句显示指定客户端连接到后面数据库的底层连接池中当前正被占用的连接信息。语法：

```
SHOW ACTIVE DBN CONN [FOR clients];
```

参数说明：

◆ **clients**: 另外指定要操作的客户端列表，格式参见 **SHOW OPS** 命令。

命令返回值为一个表格，格式与 **SHOW DBN CONN** 相同。

工作模式：管理员

相关选项：无

KILL HANGING CONN

KILL HANGING CONN 命令杀死指定客户端连接到后面数据库的底层连接中超过指定时间未执行过任何操作，但系统显示当前正被占用的连接。语法：

```
KILL HANGING CONN [minutes] [FOR clients];
```

参数说明：

- ◆ **minutes**: 指定超时时间，单位为分钟。若不指定则默认为 30 分钟。
- ◆ **clients**: 另外指定要操作的客户端列表，格式参见 SHOW OPS 命令。

命令返回值为字符串，内容包含每个客户端上各杀死了多少个连接。

工作模式：管理员

相关选项：无

SHOW TRACE

SHOW TRACE 语句显示指定客户端上所有线程的堆栈信息。语法：

```
SHOW TRACE [FOR clients];
```

参数说明：

- ◆ **clients**: 另外指定要操作的客户端列表，格式参见 SHOW OPS 命令。

命令返回值为字符串，内容为每个客户端上每个线程的堆栈。

工作模式：管理员

相关选项：无

SHOW ACTIVE CONN TRACE

SHOW ACTIVE CONN TRACE 语句显示指定客户端上占用连接的线程的堆栈信息。语法：

```
SHOW ACTIVE CONN TRACE [FOR clients];
```

参数说明：

- ◆ **clients**: 另外指定要操作的客户端列表，格式参见 SHOW OPS 命令。

命令返回值为字符串，内容为每个客户端上当前打开的 DDB 连接的线程的堆栈。

工作模式：管理员

相关选项：无

STOP CLIENTS

STOP CLIENTS 命令安全的关闭一个或多个分布式数据库中间件。语法：

```
STOP CLIENTS clients;
```

参数说明：

◆ **clients**: 指定要操作的客户端列表，格式参见 SHOW OPS 命令。

命令返回值为空。

安全关闭中间件指的是等待当前已经开始的语句和事务结束，并拒绝开始新的事务或不在事务中的语句。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

START CLIENTS

START CLIENTS 命令启动一个或多个中间件，继续提供服务。语法：

```
STOP CLIENTS clients;
```

参数说明：

◆ **clients**: 指定要操作的客户端列表，格式参见 SHOW OPS 命令。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

5.2.12 性能统计与分析

SHOW STAT TASKS

SHOW STAT TASKS 命令显示已有统计任务。语法：

```
SHOW STAT TASKS;
```

显示结果为一个表格，包含各统计任务的 ID、状态、具体进行的统计操作和参与统计的客户端。

命令返回值为一个表格，每行包含一个统计任务的信息，包含以下各列：

- ◆ **ID**: 统计任务的 ID，唯一标识。
- ◆ **STATUS**: 任务状态，分为 IDLE、RUNNING、FINISH、ERROR 等。
- ◆ **ACTIONS**: 统计任务指定的操作选项，参见 CREATE STAT TASK 命令的 actions 参数。
- ◆ **CLIENTS**: 统计任务指定的需要统计的 Client 列表。

工作模式：管理员

相关选项：无

CREATE STAT TASK

CREATE STAT TASK 命令创建一个统计任务。语法：

```
CREATE STAT TASK (DO actions ON clients | LIKE another_task)
```

参数说明：

- ◆ **actions**: 具体要进行的统计操作。为一个逗号分隔的列表，其中每一项可以是：
- ◆ **BUCKET OF (policies)**: 指定均衡策略的桶负载，**policies** 为一个逗号分隔列表，每一项为一个均衡策略的名称。
- ◆ **OPS**: 基本操作。
- ◆ **DDB**: 分布式数据库中间件的语句级统计。
- ◆ **MYSQL [EXPLAIN explain_policy][HS hs_policy]**: MySQL 语句级统计。
explain_policy 为执行 EXPLAIN 收集 MySQL 执行计划的策略，可以是 **disable**（不统计执行策略）、**first**（同类语句第一次执行时统计执行策略）或 **each**（每条语句执行前都统计执行策略），若省略则默认为 **first**。
hs_policy 为通过对比语句执行前后“show status”命令结果差异统计语句执行时的行操作情况的策略，可以是 **disable**（不统计）、**each**（每次执行都统计）或“**sample interval**”（每执行 **interval** 次同类语句统计一次），若省略则默认为“**sample 10**”。
- ◆ **COLUMN**: 统计表字段使用情况。
- ◆ **INDEX**: 统计索引使用情况。
- ◆ **MCV**: 统计分布式数据库中间件语句参数和常数值的 MCV（最常出现的值）
- ◆ **clients**: 在参与统计的客户端。格式参见 ADD MONITOR CLIENTS 命令。
- ◆ **LIKE another_task**: 以另一个已有统计任务为模板创建新统计任务，即新任务与这一任务拥有相同的统计操作和客户端列表。

命令返回值为空。

工作模式：管理员

相关选项：无

DROP STAT TASK

DROP STAT TASK 命令删除一个统计任务。语法：

```
DROP STAT TASK task_id;
```

参数说明：

- ◆ **task_id**: 统计任务 ID。

命令返回值为空。

工作模式：管理员

相关选项：无

START STAT TASK

START STAT TASK 命令启动一个统计任务。语法：

```
START STAT TASK task_id;
```

参数说明：

◆ **task_id**: 统计任务 ID。

命令返回值为空。

工作模式：管理员

相关选项：无

STOP STAT TASK

START STAT TASK 命令停止一个统计任务。语法：

```
STOP STAT TASK task_id;
```

参数说明：

◆ **task_id**: 统计任务 ID。

命令返回值为空。

工作模式：管理员

相关选项：无

GET STAT RESULT

GET STAT RESULT 命令产生统计任务的一个统计结果。语法：

```
GET STAT RESULT OF task_id [DESC BY description];
```

参数说明：

◆ **task_id**: 统计任务 ID。

◆ **description**: 统计结果的描述信息

命令返回值为空。

工作模式：管理员

相关选项：无

SHOW STAT RESULTS

SHOW STAT RESULTS 命令显示一个或多个统计任务生成的统计结果列表。语法：

```
SHOW STAT RESULTS [FOR task_ids];
```

参数说明：

◆ **task_ids**: 统计任务 ID 列表，由逗号分隔，每项为统计任务 ID。

命令返回值为一个表格，每行包含一个统计结果的信息，包含以下各列：

◆ **TASK_ID**: 统计结果对应的统计任务 ID。

◆ **RESULT_ID**: 统计结果的 ID，在同一个统计任务下需唯一。

◆ **PRODUCE_TIME**: 统计结果产生的时间。

◆ **START_TIME**: 统计任务开始的时间。

◆ **DESCRIPTION:** 统计结果的描述信息。

工作模式：管理员

相关选项：无

REMOVE STAT RESULT

REMOVE STAT RESULT 命令删除统计任务的一个统计结果。语法：

```
REMOVE STAT RESULT result_ids OF task_id;
```

参数说明：

- ◆ **result_ids:** 统计结果 ID 列表，用逗号分隔，每项为统计结果 ID。
- ◆ **task_id:** 统计任务 ID。

命令返回值为空。

工作模式：管理员

相关选项：无

ALTER STAT RESULT

ALTER STAT RESULT 命令修改统计任务的一个统计结果的描述信息。语法：

```
ALTER STAT RESULT result_id OF task_id DESC BY new_description;
```

参数说明：

- ◆ **result_id:** 统计结果 ID。
- ◆ **task_id:** 统计任务 ID。
- ◆ **new_description:** 新描述信息。

命令返回值为空。

工作模式：管理员

相关选项：无

SET SHOW STAT RESULTS

SET SHOW STAT RESULTS 命令设置分析统计时要参与的统计结果。语法：

```
SET SHOW STAT RESULTS result_ids OF task_id[, result_ids OF task_id];
```

参数说明：

- ◆ **result_ids:** 统计结果列表，用逗号分隔，其中每一项为一个统计结果的 ID。
- ◆ **task_id:** 一个统计任务的 ID。

执行此命令设置要处理的统计结果列表之后，本节以下统计命令除 SHOW STAT TABLE、COLLECT TABLE STAT、SHOW TABLE STAT 外都只会对这些统计结果进行处理。

命令返回值为空。

工作模式：管理员

相关选项：无

SHOW STAT DDB

SHOW STAT DDB 命令查询统计结果中的 DDB 语句级统计信息。语法：

```
SHOW STAT DDB [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 LABEL BY 子句外，其余子句被转化为对 `ddb_stmt_stats` 系统表的查询。`label_by` 为一个属性列表，只用于在图形化的 SQL 客户端中将结果分开显示为多个标签页，在命令行的交互式 SQL 工具中将被忽略。

命令返回值为一个表格，每行包含一个 DDB 语句统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ `id`: 记录标识，无实际意义。
- ◆ `count`: 语句被执行次数。
- ◆ `time`: 语句总执行时间，单位为毫秒。
- ◆ `avg_time`: 语句每次平均执行时间，单位为毫秒。
- ◆ `avg_dbn_count`: 语句每次执行平均访问的 DBN 数目，若为 2，表示语句每次执行平均被发送到 2 个数据库节点分布执行。
- ◆ `rows`: 语句执行涉及到的记录总行数。
- ◆ `avg_rows`: 语句执行设计到的平均记录行数，公式为总行数 `rows`/执行次数 `count`。
- ◆ `sql_sig`: 语句签名，用于标记同一类语句。

当含有 GROUP BY 子句的时候，返回结果表格列将不显示 `id`、`sql_sig` 列，但会增加显示 GROUP BY 条件中指定的属性的列。比如，含有 GROUP BY `client_id` 子句，则显示结果表格的列为 `client_id`、`count`、`time`、`avg_time`、`avg_dbn_count`、`rows`、`avg_rows`。

GROUP BY 可选的属性为：`client_id`（客户端 ID）、`task_id`（统计任务 ID）、`result_id`（统计结果 ID）、`sql_sig`（语句签名）、`explain_sig`（语句 explain 结果）、`type`（语句类型）、`tables`（语句涉及的表名）。

工作模式：管理员

相关选项：无

SHOW STAT EXTENDED DDB

SHOW STAT EXTENDED DDB 命令查询统计结果中的 DDB 语句级统计信息，除在命令结果中显示更多的属性外，与 SHOW STAT DDB 命令完全相同。

命令返回值为一个表格，每行包含一个 DDB 语句统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ `id`: 记录标识，无实际意义。
- ◆ `task_id`: 统计任务的 ID。
- ◆ `result_id`: 统计结果的 ID。
- ◆ `client_id`: 客户端 ID。
- ◆ `type`: 语句类型，如 SELECT、INSERT、DELETE 等。
- ◆ `tables`: 语句涉及的表名。

- ◆ **count**: 语句被执行次数。
- ◆ **time**: 语句总执行时间, 单位为毫秒。
- ◆ **avg_time**: 语句每次平均执行时间, 单位为毫秒。
- ◆ **mysql_count**: 语句在 **Mysql** 上被执行次数。
- ◆ **mysql_time**: 语句在 **Mysql** 上的总执行时间, 单位为毫秒。
- ◆ **max_mysql_time**: 语句在 **Mysql** 上执行的最长一次时间, 单位毫秒。
- ◆ **nonintrinsic_time**: 非本质的执行时间, 主要是建立连接 (包括实际建立连接或从连接池获取) 的时间。
- ◆ **dbn_count**: 执行时候访问的后台数据库节点总数。
- ◆ **avg_dbn_count**: 语句每次执行平均访问的 **DBN** 数目, 若为 2, 表示语句每次执行平均被发送到 2 个数据库节点分布执行。
- ◆ **rows**: 语句执行涉及到的记录总行数。
- ◆ **avg_rows**: 语句执行设计到的平均记录行数, 公式为总行数 **rows**/执行次数 **count**。
- ◆ **valid_dbn_count**: 实际返回结果的后台数据库节点数目。
- ◆ **single_dbn_count**: 访问多个后台节点却只有一个返回结果情况的出现次数。
- ◆ **lockwait_timeout_count**: 锁等待超时的次数。
- ◆ **deadlock_count**: 死锁的次数。
- ◆ **memcached_get_count**: 尝试从 **memcached** 获取的记录数量。
- ◆ **memcached_get_miss_count**: 从 **memcached** 获取数据时的没有命中的记录数量。
- ◆ **memcached_set_count**: 通过 **SET** 操作向 **memcached** 放进去的记录数量。
- ◆ **memcached_set_datasize**: 通过 **SET** 操作向 **memcached** 放进去的数据记录的总大小, 具体是指值的大小, 不包括 **key** 的大小。
- ◆ **memcached_del_count**: 通过 **DELETE** 操作让 **memcached** 删除的记录数量。
- ◆ **sql_sig**: 语句签名, 用于标记同一类语句。
- ◆ **explain_sig**: 语句 **explain** 信息。

当含有 **GROUP BY** 子句的时候, 返回结果表格列将不显示 **id**、**task_id**、**result_id**、**client_id**、**type**、**tables**、**sql_sig**、**explain_sig** 列, 但会增加显示 **GROUP BY** 条件中指定的属性的列。

GROUP BY 可选的属性为: **client_id** (客户端 ID)、**task_id** (统计任务 ID)、**result_id** (统计结果 ID)、**sql_sig** (语句签名)、**explain_sig** (语句 **explain** 结果)、**type** (语句类型)、**tables** (语句涉及的表名)。

工作模式: 管理员

相关选项: 无

SHOW STAT MYSQL

SHOW STAT MYSQL 命令查询统计结果中的 **MYSQL** 语句级统计信息。语法:

```
SHOW STAT MYSQL [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 **LABEL BY** 子句外, 其余子句被转化为对 **ddb_mysql_stats** 系统表的查询。**label_by** 为一个属性列表, 只用于在图形化的 **SQL** 客户端中将结果分开显示为多个标签页, 在命令行的交互式 **SQL** 工具中将被忽略。

命令返回值为一个表格, 每行包含一个 **MYSQL** 语句统计信息, 当不含 **GROUP BY** 子句时候包含以下各列:

- ◆ **count**: 语句被执行次数。

- ◆ **time**: 语句总执行时间，单位为毫秒。
- ◆ **avg_time**: 语句每次平均执行时间，单位为毫秒。
- ◆ **read_key**: 语句执行通过等值条件进行索引扫描操作次数。
- ◆ **avg_read_key**: 语句执行通过等值条件进行索引扫描操作平均次数
- ◆ **read_next**: 语句执行在索引中从左向右扫描过的行数。
- ◆ **avg_read_next**: 语句执行在索引中从左向右扫描过的平均行数
- ◆ **read_rnd**: 语句执行请求读入基于一个固定位置的一行的次数，基本上由排序操作导致。
- ◆ **avg_read_rnd**: 语句执行请求读入基于一个固定位置的一行的平均次数。
- ◆ **sql_sig**: 语句签名，用于标记同一类语句。
- ◆ **explain_sig**: 语句 **explain** 信息。

当含有 **GROUP BY** 子句的时候，返回结果表格列将不显示 **sql_sig**、**explain_sig** 列，但会增加显示 **GROUP BY** 条件中指定的属性的列。

GROUP BY 可选的属性为：**client_id**（客户端 ID）、**task_id**（统计任务 ID）、**result_id**（统计结果 ID）、**sql_sig**（语句签名）、**explain_sig**（语句 **explain** 结果）、**type**（语句类型）、**tables**（语句涉及的表名）、**dbn**（语句执行所在的 **dbn** 节点 ID）。

工作模式：管理员

相关选项：无

SHOW STAT EXTENDED MYSQL

SHOW STAT EXTENDED MYSQL 命令查询统计结果中的 **MYSQL** 语句级统计信息，除在命令结果中显示更多的属性外，与 **SHOW STAT MYSQL** 命令完全相同。

命令返回值为一个表格，每行包含一个 **MYSQL** 语句统计信息，除了包含 **SHOW STAT MYSQL** 命令结果的列外，还包含以下各列：

- ◆ **id**: 记录标识，无实际意义。
- ◆ **task_id**: 统计任务的 ID。
- ◆ **result_id**: 统计结果的 ID。
- ◆ **client_id**: 执行语句的客户端 ID。
- ◆ **type**: 语句类型。
- ◆ **tables**: 语句涉及到的表名。
- ◆ **columns**: 语句涉及到的列名。
- ◆ **select_columns**: **SELECT** 语句选择的列。
- ◆ **dbn**: 语句执行所在的 **dbn** 节点 ID。
- ◆ **lockwait_timeout_count**: 执行时锁等待超时的次数。
- ◆ **deadlock_count**: 死锁的次数。
- ◆ **read_first**: 语句执行从索引中读取第一个入口项的次数。
- ◆ **avg_read_first**: 语句执行从索引中读取第一个入口项的平均次数。
- ◆ **read_prev**: 语句执行在索引中从右向左扫描过的行数。
- ◆ **avg_read_prev**: 语句执行在索引中从右向左扫描过的平均行数。
- ◆ **original_sql**: 未被优化过的原始语句，即 **Client** 发送到 **DDB** 的语句。

工作模式：管理员

相关选项：无

SHOW STAT OPS

SHOW STAT OPS 命令查询统计结果中的基本操作统计信息。语法：

```
SHOW STAT OPS [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 LABEL BY 子句外，其余子句被转化为对 **opt_stats** 系统表的查询。**label_by** 为一个属性列表，只用于在图形化的 SQL 客户端中将结果分开显示为多个标签页，在命令行的交互式 SQL 工具中将被忽略。

命令返回值为一个表格，每行包含一个基本操作统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ **task_id**: 统计任务的 ID。
- ◆ **result_id**: 统计结果的 ID。
- ◆ **client_id**: 客户端 ID。
- ◆ **name**: 基本操作的名称。
- ◆ **count**: 基本操作总执行次数。
- ◆ **time**: 操作总耗时，单位毫秒。
- ◆ **avg_time**: 每次操作平均耗时，单位毫秒。
- ◆ **parent**: 当前基本操作所属父类操作，如 **CONN_GET**（连接获取）、**CONN_FREE**（连接释放）的父类操作都属于 **CONN**（连接）。

当含有 GROUP BY 子句的时候，返回结果表格列将不显示 **task_id**、**result_id**、**client_id**、**parent** 列，但会增加显示 GROUP BY 条件中指定的属性的列。

GROUP BY 可选的属性只有 **name**（基本操作的名称）。

工作模式：管理员

相关选项：无

SHOW STAT INDEX

SHOW STAT INDEX 命令查询统计结果中的索引使用情况统计信息。语法：

```
SHOW STAT INDEX [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 LABEL BY 子句外，其余子句被转化为对 **index_stats** 系统表的查询。**label_by** 为一个属性列表，只用于在图形化的 SQL 客户端中将结果分开显示为多个标签页，在命令行的交互式 SQL 工具中将被忽略。

命令返回值为一个表格，每行包含一个索引使用情况统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ **task_id**: 统计任务的 ID。
- ◆ **result_id**: 统计结果的 ID。
- ◆ **client_id**: 客户端 ID。
- ◆ **tablename**: 索引所属表名。
- ◆ **indexname**: 索引名。
- ◆ **used_count**: 总使用次数。

当含有 GROUP BY 子句的时候，返回结果表格列将不显示 **task_id**、**result_id**、**client_id**、**tablename**、**indexname** 列，但会增加显示 GROUP BY 条件中指定的属性的列。

GROUP BY 可选的属性有 **tablename**（索引所属表名）、**indexname**（索引名）。

工作模式：管理员

相关选项：无

SHOW STAT COLUMN

SHOW STAT COLUMN 命令查询统计结果中的属性使用情况统计信息。语法：

```
SHOW STAT COLUMN [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 LABEL BY 子句外，其余子句被转化为对 **column_stats** 系统表的查询。**label_by** 为一个属性列表，只用于在图形化的 SQL 客户端中将结果分开显示为多个标签页，在命令行的交互式 SQL 工具中将被忽略。

命令返回值为一个表格，每行包含一个属性使用情况统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ **task_id**: 统计任务的 ID。
- ◆ **result_id**: 统计结果的 ID。
- ◆ **client_id**: 客户端 ID。
- ◆ **tablename**: 字段所属表名。
- ◆ **columnname**: 字段名。
- ◆ **refer_count**: 字段被涉及到的次数。

当含有 GROUP BY 子句的时候，返回结果表格列将不显示 **task_id**、**result_id**、**client_id**、**tablename**、**columnname** 列，但会增加显示 GROUP BY 条件中指定的属性的列。

GROUP BY 可选的属性有 **tablename**（字段所属表名）、**columnname**（字段名）。

工作模式：管理员

相关选项：无

SHOW STAT EXPLAIN

SHOW STAT EXPLAIN 命令查询统计结果中的 MySQL 执行计划统计信息。语法：

```
SHOW STAT EXPLAIN [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 LABEL BY 子句外，其余子句被转化为对 **stmt_explain_map** 和 **mysql_stmt_stats** 系统表的联合查询。**label_by** 为一个属性列表，只用于在图形化的 SQL 客户端中将结果分开显示为多个标签页，在命令行的交互式 SQL 工具中将被忽略。

命令返回值为一个表格，每行包含一个 MySQL 执行计划统计信息，当不含 GROUP BY 子句时候包含以下各列：

- ◆ **tablename**: 表名。
- ◆ **select_type**: SELECT 的类型，可能有以下几种：
 1. SIMPLE，简单的 Select（没有使用 UNION 或子查询）。
 2. PRIMARY，最外层的 Select。
 3. UNION，第二层，在 Select 之后使用了 UNION。

4. **DEPENDENT UNION**, **UNION** 语句中的第二个 **Select**, 依赖于外部子查询。
 5. **SUBQUERY**, 子查询中的第一个 **Select**。
 6. **DEPENDENT SUBQUERY**, 子查询中的第一个 **SUBQUERY** 依赖于外部的子查询。
 7. **DERIVED**, 派生表 **Select** (**FROM** 子句中的子查询)。
- ◆ **type**: 表连接类型, 以下是常见的类型, 效率最高到最低依次为
 3. **system**, 表只有一行记录(等于系统表), 是 **const** 表连接类型的一个特例。
 4. **const**, 表中最多只有一行匹配的记录。
 5. **eq_ref**, 从该表中会有一行记录被读取出来。
 6. **ref**, 该表中所有符合检索值的记录都会被取出来。
 7. **range**, 只有在给定范围的记录才会被取出来, 利用索引来取得一条记录。
 8. **index**, 连接类型跟 **ALL** 一样, 不同的是它只扫描索引树。
 9. **ALL**, 将对该表做全部扫描。
 - ◆ **possible_keys**: 搜索表记录时可能使用到的索引。
 - ◆ **keyname**: 搜索表记录时实际上要用的索引。
 - ◆ **key_length**: 搜索表记录时使用索引的长度。
 - ◆ **ref**: 显示了哪些字段或者常量被用来和 **key** 配合从表中查询记录出来。
 - ◆ **extra**: 附加信息。
 - ◆ **count**: 语句执行次数。
 - ◆ **rows**: 检索的记录数。
 - ◆ **avg_rows**: 检索的平均记录数。
 - ◆ **max_rows**: 检索的最大记录数。
 - ◆ **sql_sig**: 语句签名, 用户标记同一类语句。

当含有 **GROUP BY** 子句的时候, 返回结果表格列将不显示 **tablename**、**sql_sig** 列, 但会增加显示 **GROUP BY** 条件中指定的属性的列。

GROUP BY 可选的属性有 **sql_sig** (语句签名)、**seq** (**explain** 返回结果的记录序号)。

工作模式: 管理员

相关选项: 无

SHOW STAT MCV

SHOW STAT MCV 命令查询统计结果中的 **DDB** 语句常数和参数值的 **MCV** 统计。语法:

```
SHOW STAT MCV [WHERE where] [GROUP BY group by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

默认的显示结果为各类语句中各个参数和常数位置的 **MCV**。**label_by** 为一个属性列表, 只用于在图形化的 **SQL** 客户端中将结果分开显示为多个标签页, 在命令行的交互式 **SQL** 工具中将被忽略。

命令返回值为一个表格, 每行包含一个参数值 **MCV** 统计信息, 包含以下各列:

- ◆ **sql_sig**: 语句签名。
- ◆ **seq**: 参数所在语句中的序号。如签名语句为 **SELECT * FROM a WHERE (c = #) AND (b = #)**, 则 **seq** 为 1 代表参数 **c**, 为 2 代表参数 **b**。
- ◆ **mcv**: 列表, 每个元素为 (参数值: 比例), 多个之间用分号间隔。如显示为 **1:30.000; 5:70.0000**, 表示参数值为 1 的比例为 30%, 为 5 的比例为 70%。

语句 **GROUP BY** 可选属性范围为 **sql_sig** (语句签名)、**seq** (参数序号)、**dbn** (执行语句的 **dbn** 的 ID)、**tables** (语句涉及到的表名)、**stmt_id** (**DDB** 语句级统计信息的记录 id)、

explain_sig（**explain** 结果信息）、**type**（语句类型）。为了统计，系统会在 **GROUP BY** 中自动添加其中的属性 **sql_sig**、**seq**、**tables**、**stmt_id**、**explain_sig**、**type**。

工作模式：管理员

相关选项：无

SHOW STAT BUCKET

SHOW STAT BUCKET 命令查询统计结果中的桶级统计信息。语法：

```
SHOW STAT BUCKET [WHERE where] [GROUP BY group_by]
[HAVING having][ORDER BY order_by][LIMIT limit][LABEL BY label_by];
```

除 **LABEL BY** 子句外，其余子句被转化为对 **bucket_stats** 系统表的查询。**label_by** 为一个属性列表，只用于在图形化的 **SQL** 客户端中将结果分开显示为多个标签页，在命令行的交互式 **SQL** 工具中将被忽略。

命令返回值为一个表格，每行包含一个统计信息，当不含 **GROUP BY** 子句时候包含以下各列：

- ◆ **policy**: 策略名称。
- ◆ **bucket**: 桶号。
- ◆ **db_url**: 桶所在数据库地址。
- ◆ **read_count**: 桶被读取总次数，**SELECT** 操作。
- ◆ **update_count**: 桶被更新总次数，**UPDATE**、**DELETE** 操作。
- ◆ **insert_count**: 桶被插入总次数，**INSERT** 操作。

当含有 **GROUP BY** 子句的时候，返回结果表格列将不显示 **policy**、**bucket**、**db_url** 列，但会增加显示 **GROUP BY** 条件中指定的属性的列。

GROUP BY 可选的属性有 **policy**、**bucket**、**db_url**。

工作模式：管理员

相关选项：无

SHOW STAT TABLE MEMCACHED

SHOW STAT TABLE MEMCACHED 命令查询统计结果中基于表的缓存操作统计信息。语法：

```
SHOW STAT TABLE MEMCACHED [ WHERE where] [ GROUP BY group_by] [ HAVING having] [ ORDER
BY order_by][ LIMIT limit][ LABEL BY label_by] ;
```

除 **LABEL BY** 子句外，其余子句被转化为对 **table_memcached_stats** 系统表的查询。**label_by** 为一个属性列表，只用于在图形化的 **SQL** 客户端中将结果分开显示为多个标签页，在命令行的交互式 **SQL** 工具中将被忽略。

命令返回值为一个表格，每行包含一个统计信息，当不含 **GROUP BY** 子句时候包含以下各列：

- ◆ **task_id**: 统计任务的 ID。
- ◆ **result_id**: 统计结果的 ID。
- ◆ **client_id**: 客户端 ID。

- ◆ **tablename**: 表名。
- ◆ **get_count**: 尝试从 memcached 获取的记录数量。
- ◆ **get_miss_count**: 从 memcached 获取数据时的没有命中的记录数量。
- ◆ **set_count**: 通过 SET 操作向 memcached 放进去的记录数量。
- ◆ **set_datasize**: 通过 SET 操作向 memcached 放进去的数据记录的总大小，具体是指值的大小，不包括 key 的大小。
- ◆ **del_count**: 通过 DELETE 操作让 memcached 删除的记录数量。

当含有 GROUP BY 子句的时候，返回结果表格列将不显示 task_id、result_id、client_id、列。

GROUP BY 可选的属性只有 tablename。

工作模式：管理员

相关选项：无

SHOW STAT ANALYSIS

SHOW STAT ANALYSIS 命令对统计结果进行指定的分析并给出建议。语法：

```
SHOW STAT ANALYSIS types;
```

参数说明：

- ◆ **types**: 要进行的分析，为一个逗号分隔的列表，其中每一项可以是：
- ◆ **forget bf**: 分析忘了加均衡字段等值条件的语句，即那些没有加均衡字段等值条件，但执行时却只有一个节点返回结果的语句。
- ◆ **forget index on bf**: 分析所属策略分布在多个 DBN 上且均衡字段上没有索引的表。
- ◆ **unused index**: 分析从来没有使用过的索引，即在 EXPLAIN 结果中从来没有被选中用于执行查询的索引。
- ◆ **index advice m**: 建议需要增加的索引，给出索引长度小于 m 的索引，m 小于等于零表示不限制长度。
- ◆ **lowcard index m**: 分析区分度不高的索引，给出区分度 cardinality 值小于 m 的索引。
- ◆ **using index m n**: 建议在已有索引基础上通过扩展一些属性实现 Using index 的优化。m 为索引最终最多包含的属性数，n 为索引最终包含的属性的最大长度。
- ◆ **deadlock ddb**: 统计执行时出现死锁的 DDB 语句。
- ◆ **deadlock mysql**: 统计执行时出现死锁的 MySQL 语句。
- ◆ **ignored index**: 作为备选索引但最终未被 MySQL 优化器采用的索引。

Forget bf 分析结果返回一个表格，每行包含一个可能忘加均衡字段等值条件的语句信息，包含以下各列：

- ◆ **sql_sig**: 语句签名。
- ◆ **count**: 执行次数。
- ◆ **dbn_count**: 语句执行发送到的 DBN 节点数目。
- ◆ **single_dbn_count**: 语句执行发送到多个 DBN 节点却只有一个返回结果的情况的出现次数。
- ◆ **balance_field**: 建议的均衡字段。

Forget index on bf 分析结果返回一个表格，每一行是一个所属策略分布在多个 DBN 上且均衡字段上没有建立索引的表的名称的信息：

- ◆ **Table**: 表名。
- ◆ **access_count**: 表的操作次数。

- ◆ **dbn_count**: 语句执行发送到的平均 DBN 节点数目。

Unused index 分析结果返回一个表格，每一行包含一个未被使用的索引的信息，包含以下各列：

- ◆ **INDEX_NAME**: 索引名称。
- ◆ **TABLE**: 所属表。
- ◆ **COLUMNS**: 索引所在的字段列表。
- ◆ **BF**: 是否是均衡字段。
- ◆ **UNIQUE**: 是否唯一。
- ◆ **COUNTS**: 被使用次数。

Index advice 分析结果返回一个表格，每一行包含一个建议的索引信息，包含以下各列：

- ◆ **TABLE**: 表名。
- ◆ **INDEX_NAME**: 建议索引名。
- ◆ **INDEX_LENGTH**: 索引长度。
- ◆ **DESC**: 描述信息。
- ◆ **STMT_EXPLAIN**: explain 结果。
- ◆ **MAX_SCAN_ROWS**: 最大检索记录数。

Lowcard index 分析结果返回一个表格，每一行表示 **cardinality** 小于指定值的索引信息，包含以下各列：

- ◆ **TABLE**: 表名。
- ◆ **INDEX_NAME**: 索引名。
- ◆ **COLUMN**: 索引字段名。
- ◆ **SEQ_IN_INDEX**: 索引字段在索引中的序号。
- ◆ **CARDINALITY**: cardinality 实际值。
- ◆ **TABLE_ROWS**: 表估算记录数。
- ◆ **ROWS_PER_VALUE**: 每个值平均记录数。

Using index 分析结果返回一个表格，每一行包含一个可能扩展的索引信息，各个列与 **index advice** 结果表格类似，增加了 **EXTEND_COLUMN_COUNT**（扩展后索引包含属性的数目）和 **EXTEND_LENGTH**（扩展后索引包含的属性的长度）两列。

Deadlock ddb、deadlock mysql 分析结果都为一个表格，包含以下各列：

- ◆ **table**: 表名。
- ◆ **sql**: 死锁 sql 语句。
- ◆ **explain**: explain 结果。
- ◆ **tables**: 关联的表。
- ◆ **deadlock**: 死锁信息。

Ignored index 分析结果返回值与 **using index** 相同。

工作模式：管理员

相关选项：无

DIFF STAT

DIFF STAT 命令比较两个统计结果之间的差异。语法：

```
DIFF STAT ((result_id1, result_id2 of task_id) | (result_id1 of task_id1,
result_id2 of task_id2));
```

参数说明：

- ◆ **result_id1**: 前一次统计结果
- ◆ **result_id2**: 后一次统计结果
- ◆ **task_id**: 统计任务 ID

命令返回值为 6 个表格，分别是统计结果 1 的表统计信息、统计结果 2 的表统计信息、结果 2 对比结果 1 的表统计信息，统计结果 1 的 **sql** 语句统计信息、统计结果 2 的 **sql** 语句统计信息，结果 2 对比结果 1 的 **sql** 语句统计信息。

表统计信息记录包含以下各列：

- ◆ **table**: 表名。
- ◆ **count**: 涉及到该表的执行次数。
- ◆ **count_ratio**: 该表执行次数占有所有表执行总次数的百分比。
- ◆ **time**: 该表所有操作的执行总时间，单位毫秒。
- ◆ **time_ratio**: 该表操作执行时间占总时间百分比。
- ◆ **select_count**: SELECT 操作次数。
- ◆ **select_time**: SELECT 操作占用时间。
- ◆ **insert_count**: INSERT 操作次数。
- ◆ **insert_time**: INSERT 操作占用时间。
- ◆ **update_count**: UPDATE 操作次数。
- ◆ **update_time**: UPDATE 操作占用时间。
- ◆ **delete_count**: DELETE 操作次数。
- ◆ **delete_time**: DELETE 操作占用时间。

Sql 语句统计信息记录包含以下各列：

- ◆ **sql_sig**: 语句签名。
- ◆ **count**: 此类语句执行次数。
- ◆ **count_ratio**: 占有所有语句执行总次数的百分比。
- ◆ **time**: 执行总时间，单位毫秒。
- ◆ **time_ratio**: 占有所有语句执行总时间百分比。
- ◆ **avg_time**: 平均执行一次所耗费的时间，单位毫秒。

对比信息表格以 +, - 表示增加或减少的百分比。

工作模式：管理员

相关选项：无

COLLECT TABLE STAT

COLLECT TABLE STAT 收集所有表或某些表的统计信息。语法：

```
COLLECT TABLE STAT [tbl_name,...] | [ OF policy,...] [/* analyze_table
without_notify_clients */];
```

参数说明：

- ◆ **tbl_name**: 指定需要统计的表名列表，没有表名代表系统中所有表。
- ◆ **policy**: 指定 **policy** 名列表，收集 **policy** 下所有表的统计信息。
- ◆ **analyze_table**: 统计表之前对表进行分析，需要对表加 **read lock**。
- ◆ **without_notify_clients**: 统计表之后不用立即通知 **client** 更新。

命令返回值为空。

工作模式：管理员

相关选项：ignore_failed_clients、ignore_unreachable_clients

SHOW TABLE STAT

SHOW TABLE STAT 显示所有表或某些表的基本统计信息。语法：

```
SHOW TABLE STAT [tbl_name,...] | [ OF policy,...];
```

参数说明：

- ◆ **tbl_name**: 指定需要查看的表名列表，没有表名代表系统中所有表。
- ◆ **policy**: 指定 **policy** 名列表，查看 **policy** 下所有表的统计信息。

命令返回值为一个表格，每行包含一个表的基本统计信息，包含以下各列：

- ◆ **NAME**: 表名。
- ◆ **POLICY**: 表所属策略名。
- ◆ **DATA_LENGTH**: 估算数据大小。
- ◆ **ROWS**: 估算记录数。
- ◆ **AVG_ROW_LENGTH**: 估算平均每行记录的长度。
- ◆ **INDEX_LENGTH**: 估算索引大小。
- ◆ **STAT_TIME**: 统计时间。

工作模式：中间件、管理员

相关选项：无

5.2.13 计划管理

SHOW PLANS

SHOW PLANS 显示所有计划信息。语法：

```
SHOW PLANS;
```

命令返回值为一个表格，每行包含一个计划的信息，包含以下各列：

- ◆ **NAME**: 计划名称，唯一。
- ◆ **TYPE**: 类型，计划任务的类型，主要包含模式修改（**SCHEMA_MODIFY**）、数据导出（**DATA_DUMP**）、数据备份（**DATA_BACKUP**）、数据迁移（**DATA_MIGRATION**）等。
- ◆ **TIME**: 计划执行的时间表达式，参见 [ADD PLAN](#) 命令的 **cron_expression**。
- ◆ **JOB**: 计划执行的语句，参见 [ADD PLAN](#) 命令。
- ◆ **LAST EXECUTE TIME**: 上次执行时间。
- ◆ **NEXT EXECUTE TIME**: 下次执行时间。

工作模式：管理员

相关选项：无

ADD PLAN

ADD PLAN 命令创建一个计划。语法：

```
ADD PLAN name START 'cron_expression' [FROM 'start_time'] [TO 'end_time'] JOB
job_name1 statement1 [, JOB job_name2 statement2...]
```

参数说明：

- ◆ **name**: 计划的名称，需唯一，可以由数字，字母，下划线组成，必须以字母或划线开头，规则和 java 的变量名规则一样。
- ◆ **cron_expression**: 执行时间表达式
- ◆ 一个完整的表达式由至少 6 个（可能 7 个）有空格分隔的时间元素，分别代表：秒、分钟、小时、天（月）、月、天（星期）、年（可选）。其中每个元素可以是一个值（如 6），一个连续区间（9-12），一个间隔时间（8-18/4、/表示每隔 4 小时），一个列表（1,3,5），通配符*。由于“月份中的日期”和“星期中的日期”两个元素互斥，须要对其中一个设置为‘?’。
- ◆ 字段允许值允许的特殊字符秒 0-59, - */分 0-59, - */小时 0-23, - */日期 1-31, - */? L 月 1-12 或 JAN-DEC, - */星期 1-7 或 SUN-SAT, - */? L 年(可选)空，或者 1970-2099, - */"/"字符用来指定数值的增量。例如：在子表达式（分钟）里的“0/15”表示从第 0 分钟开始，每 15 分钟；在子表达式（分钟）里的“3/20”表示从第 3 分钟开始，每 20 分钟（它和“3, 23, 43”）的含义一样。
- ◆ “?”字符仅被用于天（月）和天（星期）两个子表达式，表示不指定值。当 2 个子表达式其中之一被指定了值以后，为了避免冲突，需要将另一个子表达式的值设为“?”
- ◆ “L”字符仅被用于天（月）和天（星期）两个子表达式，它是单词“LAST”的缩写，L 在天（月）表达式代表一个月的最后一天；而 L 在天（星期）表达式代表一个星期的最后一天（星期六），如果“L”前面有具体内容，含义就不同了，“6L”表示这个月的最后一个星期五。
- ◆ “#”字符只能用于天（星期）表达式，n#XXX，表示每月的第 n 个星期 XXX
- ◆ **start_time**: 计划开始时间，支持以下语法：
- ◆ 一个完整的日期的时间，形如“2007-08-30 12:00:00”。
- ◆ 一个完整的时间，形如“12:00:00”，这时认为日期是今天。
- ◆ 简写的时间和日期，形如“2 am tomorrow”，其中时间部分可使用数字表示 24 时制的小时、数字加 am 或 pm 表示 12 时制的上午或下午几点。
- ◆ 从当前时间向后隔多久时间，格式为“+数字 单位”，其中单位可以是 h(小时)或 m(分钟)，如“+5 m”表示在当前时间 5 分钟之后开始，“+2 h”表示在当前时间 2 小时之后开始。
- ◆ **end_time**: 计划结束时间，语法与 start_time 类似，但以+开始的时间表示从开始时间之后隔多久时间，如“+2 h”表示统计在开始时间之后 2 小时结束，即统计历时 2 小时。
- ◆ **job_name1 statement1**: 任务名称以及任务具体执行语句。任务名称在一个计划中必须唯一。一个计划中可以有多个任务，在执行的时候会依次执行。
- ◆ 任务 **statement**: 目前支持的语句
- ◆ 模式定义与修改语句：CREATE/DROP POLICY、CREATE /DROP TABLE、CREATE/DROP INDEX、ALTER TABLE、CREATE/DROP/ALTER VIEW
- ◆ 分区管理语句：ADD/DROP PARTITIONS
- ◆ 增加 bucketno 语句：ADD BUCKETNO
- ◆ 用户与权限管理语句：ADD/DROP USER、GRANT/REVOKE、SET PASSWORD/QUOTA、ADD/REMOVE HOST
- ◆ 数据迁移语句：SINGLE MIGRATE、CONCURRENT MIGRATE、STOP ONLINE MIGRATE
- ◆ 备份导出语句：BACKUP、EXPORT、EXPORT SYSDB
- ◆ 监控任务语句：CREATE/DROP STAT TASK、START/STOP STAT TASK、GET STAT RESULT

- ◆ 收集表统计信息语句：COLLECT TABLE STAT
- ◆ 清理系统库语句：CLEAN SYS TABLES
- ◆ 选项设置语句：execute_on_dbn 、 ignore_failed_clients 、 ignore_unreachable_clients 选项的 set 语句
- ◆ Use 命令语句：通用执行器 use 命令，设置不同模式

命令返回值为空。

工作模式：管理员

相关选项：无

DROP PLAN

DROP PLAN 删除一个计划。语法：

```
DROP PLAN name;
```

参数说明：

- ◆ name: 计划名称。

命令返回值为空。

工作模式：管理员

相关选项：无

PAUSE PLAN

PAUSE PLAN 暂停一个计划。语法：

```
PAUSE PLAN name;
```

参数说明：

- ◆ name: 计划名称。

命令返回值为空。

工作模式：管理员

相关选项：无

RESUME PLAN

RESUME PLAN 重启被暂停的计划。语法：

```
RESUME PLAN name;
```

参数说明：

- ◆ name: 计划名称。

命令返回值为空。

工作模式：管理员

相关选项：无

ALTER PLAN

ALTER PLAN 修改计划，包括增加一个任务，删除一个任务，设置执行时间。语法：

```
ALTER PLAN name (ADD JOB job_name statement) | (DROP JOB job_name) | (SET TIME 'cron_expression' [FROM 'start_time'] [TO 'end_time']);
```

参数说明：

- ◆ **name**: 计划名称
- ◆ **job_name**: 任务名称
- ◆ **statement**: 任务执行语句
- ◆ **cron_expression**、**start_time**、**end_time**: 与 ADD PLAN 同

命令返回值为空。

工作模式：管理员

相关选项：无

5.2.14 其它管理命令

CLEAN SYS TABLES

CLEAN SYS TABLES 命令清理系统库中的表。语法：

```
CLEAN SYS TABLES tbl_name time [, tbl_name time,...];
```

参数说明：

- ◆ **tbl_name**: 系统库中表名称，可以为 **alarm**、**dbn**、**xa**。
- ◆ **time**: 在此时间产生的记录将被删除。支持以下语法：（可参考 ADD PLAN 命令中的时间格式）
- ◆ 一个完整的日期的时间，形如“2007-08-30 12:00:00”。
- ◆ 一个完整的时间，形如“12:00:00”，这时认为日期是今天。
- ◆ 从当前时间向前隔多久时间，格式为“-数字 单位”，其中单位可以是 **d**（天）或 **h**（小时）或 **m**（分钟），如“-7 d”表示当前时间 7 天之前，“-50 m”表示在当前时间 50 分钟之前，“-2 h”表示在当前时间 2 小时之前。

命令返回值为空。

工作模式：管理员

相关选项：无

SHOW LOG

SHOW LOG 命令显示 Master 端的日志文件。语法：

```
SHOW LOG [OPTIONS] type
```

参数说明：

- ◆ **OPTIONS:** 选项。可用如下选项：
- ◆ **--direction=head | tail:** 文件读取顺序，**head** 从头开始，**tail** 从尾部开始，默认为 **tail**。
- ◆ **--chunk-size=num:** 一块数据的大小，单位为 **K bytes**，默认为 **4K bytes**。
- ◆ **--chunk-count=num:** 读取的数据块数目，默认为 **1**。那么一次读取的总字节数为 **chunk-size * chunk-count**。
- ◆ **type:** 日志类型，为：
- ◆ **root:** 根日志，记录所有操作，文件最大。
- ◆ **config:** 配置日志，记录所有配置相关操作。
- ◆ **migrate:** 数据迁移日志，记录数据迁移操作。
- ◆ **maintain:** 数据维护日志，记录数据备份、导出操作。
- ◆ **alarm:** 警报日志，记录所有报警信息。
- ◆ **stattask:** 统计任务日志，记录统计任务相关操作。
- ◆ **xa:** 悬挂事务日志，记录悬挂事务处理操作。
- ◆ **request:** **Client** 及 **Agent** 请求日志。
- ◆ **plan:** 计划任务日志。
- ◆ **replication:** **DBN** 复制相关日志。

命令在标准输出打印出指定的日志文件，使用[结果重定向](#)可把结果输出为文件。

工作模式：管理员

相关选项：无

ALTER GLOBAL_PS_CACHE

ALTER GLOBAL_PS_CACHE 命令可修改 **Client** 上语法树缓存的全局配置参数，若 **Client** 没有单独设置过相关参数，则统一从全局配置参数读取。语法：

```
ALTER GLOBAL_PS_CACHE SET options;
```

参数说明：

- ◆ **options:** 参数选项，是{参数=值}对的组合（多个参数对之间用逗号分隔）。可用如下选项：
- ◆ **prepare_param_limit:** **Client** 上缓存 **preparedStatement** 时对其参数数量的限制，参数数量大于该值的 **preparedStatement** 不被缓存。默认值为 **10**。
- ◆ **cache_sql_size_limit:** **Client** 上缓存语法树的 **SQL** 语句长度限制，**SQL** 语句长度超过该值不被缓存。默认为 **256**。
- ◆ **parseTree_cache_size:** **PS** 语句解析树缓存大小限制，解析树满的时候利用 **LRU** 策略替换。默认为 **1024**。

工作模式：管理员

相关选项：**ignore_failed_clients**、**ignore_unreachable_clients**

ALTER CLIENT_PS_CACHE

ALTER CLIENT_PS_CACHE 命令可修改特定 **Client** 上语法树缓存的配置参数。语法：

```
ALTER CLIENT_PS_CACHE client_id SET options;
```

参数说明：

- ◆ **client_id:** 指定 **Client** 的 ID。

- ◆ **options**: 参数选项，是{参数=值}对的组合（多个参数对之间用逗号分隔）。可用如下选项：
- ◆ **cache_sql_size_limit**: 含义与 ALTER GLOBAL_PS_CACHE 命令中的相同，但可以设置为-1（代表使用全局配置的值）。
- ◆ **parseTree_cache_size**: 含义与 ALTER GLOBAL_PS_CACHE 命令中的相同，但可以设置为-1（代表使用全局配置的值）。

工作模式：管理员

相关选项：无

BASH

BASH 命令可执行本地的脚本命令。语法：

```
BASH "commands";
```

参数说明：

- ◆ **commands**: 脚本命令。

工作模式：管理员

相关选项：无

6 使用 DDB JDBC 客户端

6.1 通过标准 JDBC 进行连接

加载驱动请用：

```
Class.forName("com.netease.backend.db.DBDriver")
```

语句。

建立连接请用：

```
DriverManager.getConnection(url, user, pass)
```

或

```
DriverManager.getConnection(url, null)
```

语句。若使用的是后二种形式，则必须在 url 中使用参数 **user** 和 **password** 指定用户名和密码。

6.1.1 连接字符串格式

连接 url 格式如下：

```
<master host>[:<master port>][?<options>]
```

其中<master host>为分布式数据库 MASTER 的 IP 或域名，<master port>为 MASTER 监听端口，可以省略，若不指定则为 8888。

url 可包含一些参数，参数内容放在?号之后。每个参数形如“key=value”形式，若有多个参数则选项之间用&分隔。这些参数只对当前连接生效。可用的参数有：

- ◆ user: 用户名。
- ◆ password: 密码。
- ◆ chkdupkey: 执行更新操作时是否启动全局索引唯一性检查功能，默认为 true。指定为 false 可以提高更新操作的性能，但可能会导致存储在不同节点上的数据之间包含重复数据，因此除了数据装载等应用可以确信不会违反索引唯一性，不建议将 chkdupkey 设为 false。该参数只对当前连接有效。
- ◆ debug: 执行操作时通过 log4j 打印出更多的反映中间件查询处理过程的日志信息
- ◆ timeout: 设置语句执行超时时间，单位为秒，只对当前连接生效。若不指定则使用 MASTER 上统一配置的执行超时时间，默认配置为 120 秒。
- ◆ maxconn: 设置最大 DDB 连接数限制，对当前进程有效。若不指定则使用 Master 上统一配置的连接数，默认配置为 1024。
- ◆ pspoolsize: 每个 DBN 连接的最大 PreparedStatement 缓冲数量。若不指定则使用 Master 上统一配置的参数，默认配置为 50。
- ◆ coonpoolsize: 为每个 DBN 分配的普通连接池最大连接数限制。若不指定则使用 Master 上统一配置的参数，默认配置为 50。
- ◆ xapoolsize: 为每个 DBN 分配的 XA 连接池最大连接数限制。若不指定则使用 Master 上统一配置的参数，默认配置为 50。
- ◆ logsql: 是否记录执行的 sql 语句日志，进程内所有连接有效，默认配置为 false。
- ◆ logreadonly: 记录只读事务日志开关，开启 sql 日志时才能生效，默认配置为 true。
- ◆ logautocommit: 记录 autocommit 模式的语句开关，开启 sql 日志时才能生效，默认配置为 true。
- ◆ cachesqlsize: 缓存语法树的 sql 语句长度限制，若不指定则使用 Master 上统一配置的参数，默认配置为 256。
- ◆ parsetreecachesize: 语法树缓存个数限制，若不指定则使用 Master 上统一配置的参数，默认配置为 1024。
- ◆ maxconditions: SQL 语句中最大 in/or 条件个数
- ◆ maxoffset: SQL 语句中最大 offset 值
- ◆ logdir: 分布式数据库中间件 JDBC 驱动运行时需要记录分布式事务日志，这一选项指定日志所有目录。若不指定，则默认由 MASTER 统一设置。
- ◆ logfile: 指定分布式事务日志文件的基本名，日志文件会有多个，最终的文件名形如 logfile_1.log, logfile_2.log...等。
- ◆ key: 密钥文件的完整路径。使用 JDBC 驱动时必须指定这一系统变量。
- ◆ name: 指定一个反映使用中间件程序用途或标识的字符串，便于管理员区分各使用 DDB 程序的用途。可以不设置。
- ◆ lwip: 轻量级中间件 ip 地址。
- ◆ safelyshutdown: 进程退出时是否安全关闭 DDB Driver，可能导致退出时间较长。
- ◆ readtimeout: 连接首次连接 DDB Master 时的读数据超时时间。
- ◆ defaultxa: 当开启事务时，是否默认使用 xa 连接，具体用法参见性能优化章节中的说明。

url 示例:

```
DriverManager.getConnection("127.0.0.1?user=xxx&password=xxx", null);
DriverManager.getConnection("127.0.0.1:8889?user=xxx&password=xxx");
DriverManager.getConnection("127.0.0.1:8889?timeout=60", "xxx", "xxx");
```

6.1.2 设置 JAVA 环境变量

上面 URL 格式中提到的各种连接参数，也可以在 JAVA 环境变量中进行设置。这时，java 的启动命令应写为：“java -Dprop=value”。

URL 连接字符串中设置参数的使用优先级高于 JAVA 环境变量中的参数。

6.1.3 同时连接多个 DDB

DDB 允许一个客户端进程中同时访问多个 DDB 数据库，目前支持两种方法：

方法一：对每个 DDB 分别建立一个连接。

采用这种方法可以对两个 DDB 进行独立访问，该方法存在的问题是：对两个 DDB 的操作无法用同一个事务保护。如下例所示：

```
import com.netease.backend.db.DBConnection;
Connection conn1 =
DriverManager.getConnection("127.0.0.1:8889?user=xxx&password=xxx");
Connection conn2 =
DriverManager.getConnection("127.0.0.1:8889?user=xxx&password=xxx");
Statement stmt1 = conn1.createStatement("select * from table1");
Statement stmt2 = conn2.createStatement("select * from table2");
```

方法二：使用同一个连接访问多个 DDB。

建立连接时在 url 中指定多个 DDB 的连接信息，DDB 连接信息之间通过分号间隔，建立连接后可以通过在 sql 语句中指定数据库名称来访问对应的 DDB，如下例所示：

```
import com.netease.backend.db.DBConnection;
Connection conn =
DriverManager.getConnection("127.0.0.1:8889?user=xxx&password=xxx;192.168.80.67:8887?user=xxx&password=xxx");
Statement stmt1 = conn.createStatement("select * from ddb1.table1");
Statement stmt2 = conn.createStatement("select * from ddb2.table2");
Statement stmt3 = conn.createStatement("select ddb1.table1.col1,
ddb2.table2.col2 from ddb1.table1, ddb2.table2");
```

其中 ddb1 是地址为 127.0.0.1:8889 的 DDB 名称，ddb2 为 192.168.80.67:8889 的 DDB 名称，具体访问的 DDB 名称可以向 DBA 获取。

使用这种方法可以在同一个事务中操作两个不同 DDB 上的数据。

注意：目前只有 select 语句支持一个语句中同时访问多个 DDB，其他类型的语句每条语句只能访问一个 DDB。

6.2 客户端启动

DDB 客户端目前是非轻量级的，以下将对客户端尤其是其启动过程做详细的描述，对于启动过程中可能出现的异常情况和解决办法将做特别说明。

6.2.1 术语与概念

客户端

使用 DDB 的应用程序称为 DDB 的客户端，在以下的说明中将统称为客户端或 **Client**。

AServer

AServer 完成与 DDB Master 服务器 (MServer) 的所有交互操作。AServer 对于应用是不可见的，并不对外部提供对其进行管理的操作接口。当 **Client** 试图首次获取 DDB 的连接时，AServer 将自动启动，当 **Client** 退出时，AServer 将自动关闭。

DBI

DBI 是 DDB Master 服务器 (MServer) 的客户端，负责在 DBN (数据库节点) 上执行 SQL 语句。AServer 可能包含多个 DBI，用于完成与 MServer 的交互操作。

6.2.2 DBI 配置管理

配置文件

DBI 有单独的配置并且持久化为对应的配置文件，名称为“<name>.meta”，其中“name”为 DBI 的名字（同 DDB 的名字）。

SQL 日志文件

DBI 存在多个日志文件，名称为“<name>_<no>.log”，其中“no”为自增的序列号，日志文件的个数在配置文件中指定。

配置文件位置

当前进程内所有的 DBI 的配置文件和 SQL 日志文件位于同一目录，该目录的默认值是 AServer 的配置目录（如后面章节所述）。当 **Client** 首次连接 DDB，可以在 DDB 连接 URL 里通过指定“logdir=xxx”参数指定该目录。如果首次连接时指定的是连接多个 DDB 的 URL，而这些 DDB 的连接参数里多次设置了该参数，那么将按序把第一个 DDB 连接参数列表里包含的“logdir”参数的值视为有效值。当 **Client** 进程未退出，以后再连接 DDB 时，如果再次指定了“logdir”参数，将被忽略。

创建或加载配置文件

当 **Client** 试图通过某个符合 DDB 连接协议的 URL 去获得 DDB 连接，而且是首次指定该 URL (AServer 将首次去连接对应的 MServer) 时，AServer 将读取该 DBI 的配置以完成 DBI 的初始化。

如果一个配置项在配置文件中存在，则以本地配置为准，如果不存在，则读取 **master** 端保存的全局配置信息。当数据库管理人员单独选择一个客户端修改其配置时，被修改过的配置项将被保存在客户端本地配置文件中。

创建和加载配置文件时可能出现以下异常情况：

- “DBI 配置加载错误: xxx”、“DBI 配置保存错误: xxx”：
更详细的错误信息可能是“DBI 配置文件目录未锁定: xxx”或 I/O 错误信息等。
解决方法：重启应用并检查配置文件目录的 I/O 权限等。

更新配置

Client 可以在指定的 DDB 连接 URL 里添加更多的参数，这些参数将重载已有的 DBI 配置里对应的参数。显然，如果 **Client** 非首次连接某 DDB，且此时连接的 URL 相对上次提供了不同的参数列表，将导致重载并持久化所指定的参数，同时，先前所获得的连接将也会自动使用这些配置，因为某 DBI 的配置实例当前进程内是唯一的。

创建和加载配置文件时可能出现以下异常情况：

- “DBI 配置保存错误：xxx”：
更详细的错误信息可能是“DBI 配置文件目录未锁定：xxx”或 I/O 错误信息等。
解决方法：重启应用并检查配置文件目录的 I/O 权限等。

配置文件内容

<更多.....>

6.2.3 DBI 的注册（ID 分配）

ID 与签名

在 Client 同时连接多个 DDB 时，每个 DBI 可以由 DBI 的名称（即 DDB 的名称）和唯一 ID 确定。

DDB（或者 MServer）使用唯一 ID 来标志所连接的 DBI（或 MServer 的 Client）。注册（申请 ID）要求 DBI 发送其签名，它由类型、版本、主机地址、服务主机地址（通常同主机地址）、服务端口、客户端所在文件目录六部分组成。若 DBI 的签名发生改变，该 ID 将无效。

分配或验证 ID

若 Client 通过某个符合 DDB 连接协议的 URL 获得 DDB 连接时，如配置管理所述，将通过本地配置文件或通过 MServer 加载 DBI 的配置，其中“id=xxx”为配置的一个属性，即该 DBI 的 ASID。

DBI 获得配置后完成上下文的初始化时，将首先去 MServer 注册，可能会获取新的 ID 或验证保存在配置文件中的已有 ID 是否仍然有效，此时可能出现以下“DBI 端 ID 分配或验证失败！”异常，它包含以下情况：

- 已没有足够的 ID 可分配：
目前每个 DDB 可管理的 DBI 的 ID 的个数是 1024 个（0 号 ID 不予使用），因此可管理 1023 个 DBI，已分配的 DBI 的 ID 会被回收，但为了保证正确性，可能不会及时被回收掉，因此仍然可能会出现 ID 不够用的情况。
解决方法：需 DBA 处理，如手动删除一些可能无效的 DBI。
- 版本不匹配：
DBI 或 MServer 的版本变更都可能导致该问题，因为 MServer 有可接受的 DBI 的版本范围，这会在详细的错误信息里给出。
解决方法：将客户端升级到最新版本。
- ID[xxx]分配出错：
如果是分配新的 ID，可能会出现“不能分配该 ID 或以无可用的 ID 分配！”、“持久化(增加) DBI 信息失败！”错误，分别指无可用的 ID 分配或保存 DBI 的信息失败；
解决方法：DBA 手动删除废弃的已分配 ID 并检查系统库是否出错；
如果是验证已有的 ID，可能会出现“DBI[XXX]的签名不匹配！”错误，可能是由于复制配置文件目录（通常即为日志文件目录“/log”）如复制 ISQL 或使用 DDB 的应用程序所导致。
解决方法：删除配置目录并重启应用。

6.2.4 AServer 启动

在 Client 加载并运行时，AServer 将作为进程内部的服务器启动，应用并不需要关注 AServer 的行为，Client 只需要与 AServer 交互，而 AServer 与 MServer 的交互对于 Client 或

应用而言是完全透明的。

如上节所述，Client 首次连接 DDB 时指定 DBI 的配置文件目录，如果不指定 AServer 将使用默认的目录“/log”，配置文件的目录以后在当前进程中将不可修改！

为了避免多个 AServer 的进程共享使用同一份 DBI 配置文件可能出现的错误，AServer 启动时将试图锁定配置文件目录，并在进程退出时自动解决锁定。因此 AServer 启动时可能出现一下异常情况：

- **DBI 配置管理器初始化失败：**
更多的错误信息可能有“DBI 配置文件目录可能已被其他 AServer 所使用！”或“DBI 配置管理器初始化 I/O 错误：xxx”。
解决方法：检查是否在同一目录下启动多个 ISQL 或应用程序，如果是则说明有错；或在 DDB 连接 URL 里指定配置文件目录；或检查配置文件目录的 I/O 权限等。

此外，AServer 启动时将根据系统属性确定是否启用 JMX 远程管理服务，如果出现问题，请查看对应章节进行解决。

6.3 其他

6.3.1 获取 DBN 连接信息

为了提高使用 DDB 的灵活性，方便开发人员直接访问 DBN，在 DBConnection 中提供了获取 DBN 连接信息的接口。

获默认 DDB 或指定 DDB 的所有 DBN 的 url，url 中不带用户名和口令参数：

```
String[] getAllDbnUrl() throws SQLException;
String[] getAllDbnUrl(String ddbName) throws SQLException;
```

获默认 DDB 或指定 DDB 的所有 DBN 的 url，url 中带用户名和口令参数：

```
String[] getAllDbnUrlWithUser throws SQLException;
String[] getAllDbnUrlWithUser(String ddbName) throws SQLException;
```

获默认 DDB 或指定 DDB 中表所在 DBN 的 url，url 中不带用户名和口令参数：

```
String[] getAllDbnUrlByTable(String tableName) throws SQLException;
String[] getAllDbnUrlByTable(String ddbName, String tableName) throws
SQLException;
```

获默认 DDB 或指定 DDB 中表所在 DBN 的 url，url 中带用户名和口令参数：

```
String[] getAllDbnUrlByTableWithUser(String tableName) throws SQLException;
String[] getAllDbnUrlByTableWithUser(String ddbName, String tableName) throws
SQLException;
```

获默认 DDB 或指定 DDB 中表所在 DBN 的 url，能够具体指定均衡字段值进行查询。url 中不带用户名和口令参数：

```
List<String> getDbnByKeyValue(String tableName, List<Object> keyValues) throws
SQLException;
List<String> getDbnByKeyValue(String ddbName, String tableName, List<Object>
keyValues) throws SQLException;
```

6.3.2 客户端版本限制

DDB 的更新和完善一直都在进行中，有较大功能升级后，有时客户端也必须升级到相应的

版本才能与新版本的 **Master** 互通。为避免客户端版本过于落后引发一些莫名的错误，DDB 中加入对客户端版本的限制，**Master** 会拒绝不兼容版本客户端的连接请求。

master.jar 包含一个文件 **acceptable-client-version.txt**，标识 **master** 可接收的客户端版本列表。**client** 端用于版本校验的版本号则硬编码在 **db.jar** 的代码中。

当前 DDB 的版本号由 3 位数字组成，版本限制只检查前两位。如果 **master.jar** 包中不存在 **acceptable-client-version.txt** 文件，则 **Master** 不进行版本限制。

DDB4 的通信协议有比较大的改动，与 DDB3 的 **dbi** 已经无法兼容，也无法进行版本校验操作。如果客户端在版本校验过程中没能通过版本限制，连接 **Master** 会返回类似异常信息：

Client version check failed, master[192.168.84.22:8889]的版本为4.2.1，可接受dbi版本为[4.1, 4.2]，当前dbi版本为4.0

7 JMX 远程管理控制与统计

DDB 通过 Java Management Extensions (JMX)方式向外暴露服务的管理控制、以及信息（如统计信息等）的查询接口，方便 DBA 和开发人员使用。

目前，DDB 的 JMX 支持对开发人员和用户提供以下功能：

- 能够方便的将已有服务注册、发布成远程管理服务的基础服务；
- DBI 端的各类统计信息的远程查询接口；
- 提供更多的基于 JMX 的 DDB 管理接口（下阶段）。

7.1 JMX 基础服务

基于 JMX ModelBean 的方式注册和导出需要进行远程调用的服务，无需指定标准的 MBean，同时可以在 JMX 客户端可以通过该服务所实现的任意接口类进行调用，JMX Client 基于 JDK 动态代理，可根据 MBean 的接口类型获得远端 MBean 的直观调用。

服务端和客户端的接口定义：

```
// com.netease.mx.JMXServerManager
// JMX Server Side
<T> void manage(T mBean, String domain, String key) throws JMException, IOException;
/** ----- */
// com.netease.mx.JMXClientManager
// JMX Client Side
<T> T getMBean(Class<T> mBeanInterface, String domain, String key) throws IOException, JMException
```

7.1.1 以 JMX 方式发布服务为远程服务

首先要获得 JMX 服务管理器。

```
JMXServerManager m = JMXServerManager.getInstance();
// m = JMXServerManager.getInstance(host, port);
```

JMX 服务管理器将使用 JVM 的默认环境变量“**com.sun.management.jmxremote.port**”来确定服务监听端口，如果不存在该环境变量，将使用默认监听端口“**JMXServerManager.DEFAULT_SERVICE_PORT**”（1099）作为服务监听端口。

在 DDB 中，虽然通过调用内部的“JMXServerManager”完成远程服务的发布，但处理方式有所不同，即如果环境变量“com.sun.management.jmxremote.port”未指定，则认为禁用 DDB 的 JMX 服务，在禁用 JMX 时如果要通过 JMX 发布其他服务时将会导致“IllegalStateException”。

其次，将服务以 JMX 的方式发布为远程服务，这要求该服务至少实现一个接口，在客户端或 JConsole 中工具中将根据其实现的所有接口（即包含其实现的所有接口）发布远程接口服务。

```
// construct or get the service reference first
// ServiceInterface service = ...
// domain name such as "ddb.dbi" and service object key name such as "conn-stats"
m.manage(service, domain, key);
```

最后，在关闭应用时进行后续处理（DDB 会自动完成）。

```
// on exit
m.dispose();
```

7.1.2 JMX 远程服务的 RMI 调用 URL

以 JMX 方式发布的服务的名称定义为

```
service:jmx:rmi:///jndi/rmi://<host>:<port>/<domain>
```

其中，“domain”通常会定义为“xxx.yyy.zzz”的方式，而转换成 URL 之后会变成“/xxx/yyy/zzz”的方式，所有在这个域下发布的服务都可以通过 URL

```
service:jmx:rmi:///jndi/rmi://<host>:<port>/xxx/yyy/zzz
```

进行 RMI 调用。

比如在 DDB 的某个在线的 DBI 节点的服务的将发布在域“ddb.as.xxxx”下，其中“ddb.as”为在 AServer 上发布的 JMX 服务所在域的前缀或默认域（未指定名称时），而“xxxx”为该 AServer 的名称，虽然 AServer 可能包含多个 DBI（即连接到多个 DDB），但各 DBI 通过 JMX 发布的服务将共享同该域，因此也拥有相同的远程 RMI 调用 URL。

通过如下方式在 JVM 启动的系统属性中指定名字：

```
-Dnetease.ddb.jmxremote.name=<name>
```

名字加上前缀“ddb.as”将构成完成的域名“ddb.as.xxxx”，若不指定则域名为“ddb.as”。

同样，通过如下方式指定 JMX-RMI 远程服务监听端口：

```
-Dnetease.ddb.jmxremote.port=<port>
```

因此，而 DDB 的 DBI 端的 JMX-RMI 服务连接 URL 将为

```
service:jmx:rmi:///jndi/rmi://<host>:<port>/ddb/as[/<name>]
```

7.1.3 以 JConsole 等方式或撰写 JMX-RMI 客户端进行调用远程服务

1. JConsole 等工具

通过 JConsole 等 JMX-RMI 工具直接查看 DDB 某个 DBI 所发布的统计信息，其 JMX-RMI 连接 URL 为

```
service:jmx:rmi:///jndi/rmi://<host>:<port>/ddb/as[/<name>]
```

2. 撰写 JMX-RMI 客户端进行调用

```
JMXClientManager m = JMXClientManager.getInstance(host, port);
// "ServiceInterface" is the interface class name of service exported via JMX
```

```
// domain name such as "ddb.dbi" and service object key name such as "conn-stats"
ServiceInterface s = m.getMBean(ServiceInterface.class, domain, key);
// call service interface as normal
// StringMatrix m = s.getConnStat();
// ...
// m.dispose(); // on exit
// ...
```

7.2 DDB 中 DBI 以 JMX 形式发布的统计服务

7.2.1 统计服务类型和使用

目前在 DDB 的 DBI 端以 JMX 方式发布的统计服务（包括 **key**）有

- ConnStatsService (stats-conn)，连接（包括 XA 连接）统计信息；
- OptStatsService (stats-opt)，操作统计信息，包括 DBI（连接，Stmt，PS，Transaction 等），DBN，连接池，线程池等计数；
- ResourcesStatsService (stats-res)，包括连接、Stmt 和 PS 的更详细的资源状态信息；
- ThreadStackTraceService (trace)，线程堆栈（包括活动连接堆栈、DBN、DBI 连接线程堆栈等）
- MiscStatsService (stats-misc)，其他统计信息。

其中，每种类型的统计服务实现 2 个接口，如存在实现了 ConnStatsService 和 ConnStatsStrService 的服务，提供两类对等的接口，如：

```
// ConnStatsService
// ...
StringMatrix getConnStats();
// ...
// ConnStatsStrService
// ...
String getConnStatsStr();
// ...
```

其中：

- 1、前者用户远程调用（JMX Client，包含相关的 class），而后者则没有自定义类，可以在 JConsole 等工具中直接调用。
- 2、如果直接在 JConsole 中调用前类接口，将可能获得“java.rmi.UnmarshalException”异常。

7.2.2 统计服务接口方法的具体定义（待补充）

<略>

7.3 配置和默认监听端口

7.3.1 配置 DDB 的 JMX 远程管理服务

DDB 使用了自定义的一些系统变量来配置 JMX 远程管理服务，即为了避免和 JVM 本身的配置产生冲突。

在 JVM 启动时通过如下系统变量对 DDB 的 JMX 远程管理服务进行配置：

启用 JMX 并配置 RMI 服务监听端口：

```
-Dnetase.ddb.jmxremote.port=<port>
```

配置通过 JMX 发布的服务所在的域的名字（可选）：

```
-Dnetase.ddb.jmxremote.name=<name>
```

7.3.2 JVM 相关的 JMX 配置

JRE 本身会开启 JMX 服务（即注册一些平台相关的可远程调用的监控服务），但并不会开启监听（JDK6 及以上版本默认本地监听，即本机可使用 JConsole 等工具进行直接附加到进程并监听），可以通过以下环境变量配置（或在 JRE 的 lib 目录下的相关配置文件中配置，但会被此变量重载）。

JDK5 及以下版本开启本地监听（JDK6 及以上默认开启）：

```
-Dcom.sun.management.jmxremote=<true | false>
```

开启远程监听（默认开启 SSL 和认证等）：

```
-Dcom.sun.management.jmxremote.port=<port>
```

另外还有其他一些参数配置认证等，具体参见：

<http://download-llnw.oracle.com/javase/6/docs/technotes/guides/management/agent.html>

使用示例（常用）：

```
-Dcom.sun.management.jmxremote.port=1099
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

7.3.3 注意事项

一般情况下，DDB 的 JMX 远程管理服务的配置和运行与 JVM 本身的 JMX 的配置不存在冲突，但如果两者指定了相同的端口，将可能会导致以下错误：

```
java.io.IOException: Cannot bind to URL [rmi://localhost:1099/ddb/as]:
javax.naming.NoPermissionException [Root exception is java.rmi.ServerException:
RemoteException occurred in server thread; nested exception is:
java.rmi.AccessException: Cannot modify this registry]
```

这并不是端口占用的错误，因为 JMX 服务端口是可以在线程甚至进程间共享的，如果启用了 JVM 本身的 JMX 远程管理服务同时又取消了认证，JVM 所开启的 JMX 服务监听端口将被配置为只读，即不能绑定新的 RMI 远程访问 URL。

因此，要避免上述情况出现。

8 SQL 语法说明

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
- ◆ 支持根据单个属性/表达式进行 HAVING

- ◆ 支持根据单个或多个属性/表达式/聚集函数进行 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)

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

DDB 支持的 SQL 语法都是主要面向 MySQL 数据库引擎，如果使用其他数据库，可能会有微小差异，具体参见本手册的数据库引擎单独说明。

下面分章节对各个 SQL 语句语法进行详细说明。

8.1 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。

8.1.1 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
- ◆ 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
- ◆ 对于任意 FROM 子句中相邻出现的表，即会进行 JOIN 的表，在 WHERE 条件中必须直接或者间接的出现它们的连接条件。

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, 没有直接 join 条件, 但是可以推导出 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';

```

8.1.2 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和排序
- ◆ 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** 子句的投影列表中
- ◆ 支持对聚集函数进行排序, 但对应的聚集函数必须出现在 **SELECT** 子句的投影列表中
- ◆ 分组操作不能应用在聚集函数上

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);

```

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投影列表中
select id from score group by score order by max(score);

```

8.1.3 DISTINCT

分布式数据库支持 **DISTINCT**，并可与聚集函数联合使用，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，表示对整个投影列表做 **distinct**，返回记录投影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**。

另外，**DISTINCT** 操作符不能被应用在聚集函数上。

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;
```

8.1.4 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

8.1.5 聚集函数

支持 MAX、MIN、SUM、COUNT、AVG

8.1.6 LIMIT/OFFSET

分布式数据库支持 **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;
```

8.1.7 FORCE INDEX

分布式数据库支持 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
```

8.1.8 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 ...
```

8.2 INSERT

分布式数据库支持 INSERT 语句，语法为：

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

中间件不支持以下复杂的 INSERT 语句：

- ◆ 不支持 INSERT INTO ... SELECT 语句
- ◆ 不支持 INSERT INTO ... SET ...语句

当 SQL 语句中包含 ON DUPLICATE KEY UPDATE 时，update 的列不能是均衡字段。

但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：

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。

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。

8.2.1 全局 ID 分配

分布式数据库中间件 JDBC 驱动提供了分配全局 ID 的功能以替代 AUTO_INCREMENT 功能。使用全局 ID 分配功能的方法有两种。一是调用 com.netease.backend.db.DBConnection.allocateRecordId 函数，如下所示：


```
import com.netease.backend.db.DBConnection;
Connection conn = DriverManager.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。

第二种方法是在插入语句中使用特殊的 **seq** 标志，若要得到生成的 ID 的值，则可以用 **getGeneratedKeys** 方法，如下所示：

```
Connection conn = DriverManager.getConnection(url, user, pass);
Statement s = conn.createStatement();
s.executeUpdate("insert into test(seq, 'a')");
ResultSet rs = s.getGeneratedKeys();
rs.next();
long id = rs.getLong(1);
```

8.2.2 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。

8.3 REPLACE

分布式数据库支持简单的单条记录 REPLACE 语句，语法为：

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

8.4 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]];
```

8.5 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]];
```

8.6 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**。如果用户想获取存储过程的返回值，可以把

8.7 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`。

8.7.1 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 **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。

在原有的基础上支持多字段均衡的 **FORWARD BY**：

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, TABLE1.value2...), (TABLE2.value1, TABLE2.value2...)) */
```

其中 **TABLE1.value1, TABLE1.value2...**为表 **TABLE1** 的所有均衡字段；**TABLE2.value1, TABLE2.value2...**为表 **TABLE2** 的所有均衡字段。

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，好处是能够绕过 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**。但另一方面，**Direct forward** 在使用时也有些限制，因为这个功能不经过 DDB 语法解析，所以导致中间件不能对这类语句进

行一些必要的自动转换。因此，DDB 进行在线迁移过程当中，中间件就会屏蔽所有的 **Direct forward** 语句。另外，在语句统计时，也会忽略掉 **Direct forward** 的语句。

8.7.2 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。

8.8 MEMCACHED

分布式数据库提供 **memorycache** 中存储记录主键到记录数据的 **key-value** 键值对，可直接使用主键进行数据查找。用户可设定当前 **SQL** 语句是否强制使用 **memcached**，即该语句所涉及的表指定了缓存 **memcached key** 字段，不管表是否开启“缓存表记录”开关，均强制使用 **memcached**。

使用方法如下：

在语句前加上

/*MEMCACHED = TRUE*/ 强制使用 **memcached**

/*MEMCACHED = FALSE*/ 强制不使用 **memcached**

例如：

/*MEMCACHED = TRUE*/**注意：** **LOAD BALANCE** 只能用于 **SELECT * FROM table;**语句

```
/*MEMCACHED = FALSE*/SELECT * FROM table WHERE id = 1;
```

使语句最终使用 **memcached** 还必须满足以下条件：

1. 该 **sql** 语句所涉及的表指定了缓存 **memcached key** 字段。（多表 **join** 的情况较复杂，详情见 15.2.1 节中“分析是否使用 **memcached** 的流程图”）
2. 该 **sql** 语句符合操作 **memcached** 条件。（详情见 15.2.1 节中“使用 **memcached** 流程”）

8.9 JDBC 标准符合度说明

分布式数据库中间件 **JDBC** 驱动支持 **JDBC 3.0** 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。

8.9.1 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()

8.9.2 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

8.9.3 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

8.9.4 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

8.9.5 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 columnName) throws SQLException
- ◆ public Clob getClob(int column) throws SQLException
- ◆ public Clob getClob(String columnName) 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

8.9.6 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)

8.9.7 java.sql.Blob 类

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

8.10 示例程序

下面程序演示了使用标准 jdbc 接口操作 DDB 的方法。

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

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();
    }
}
```

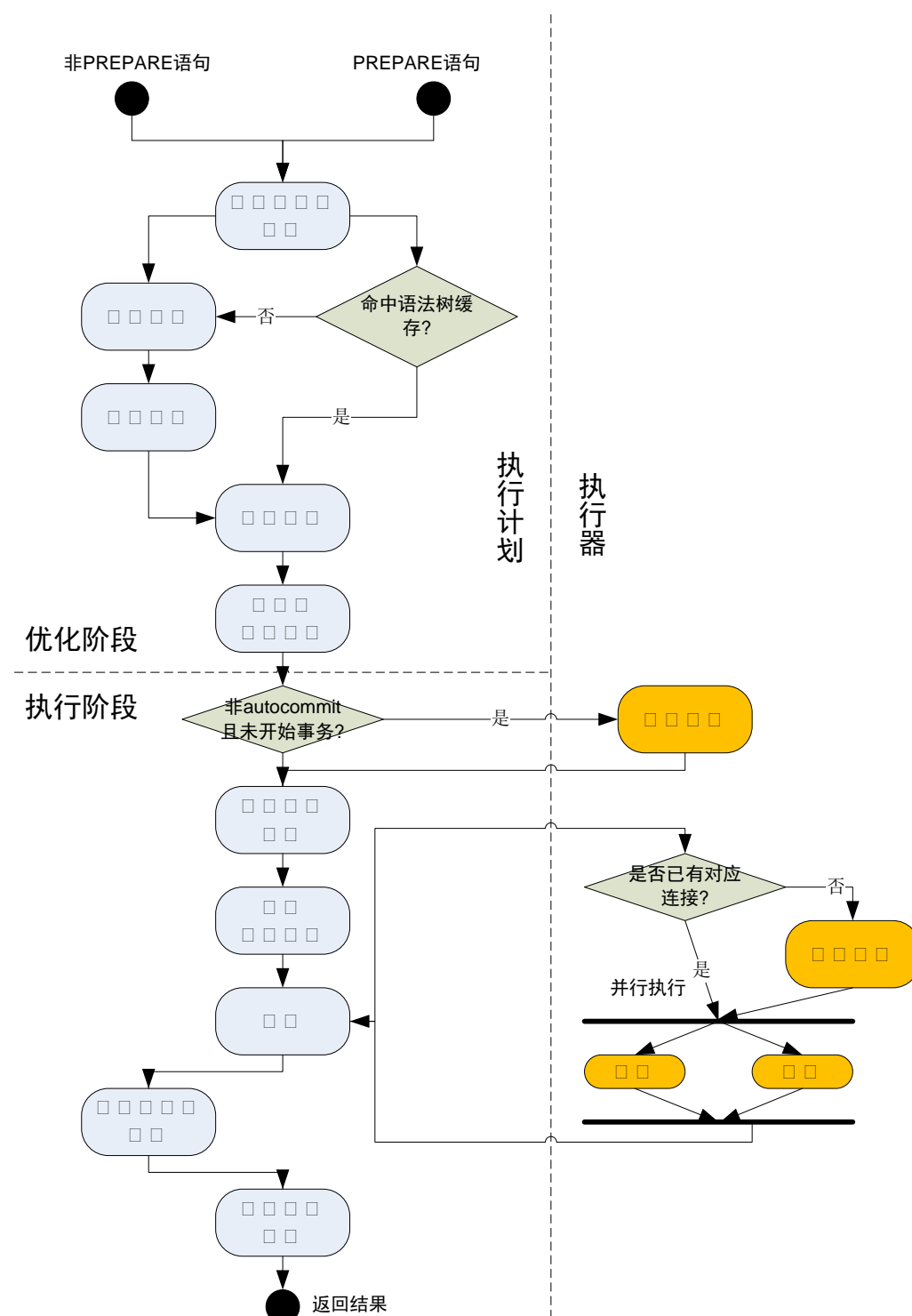
```
        if (pstmt != null) {  
            pstmt.close();  
        }  
        if (con != null) {  
            con.close();  
        }  
    }  
}
```

9 SQL 语句执行相关说明

9.1 语句执行流程

9.1.1 查询处理流程

中间件查询处理（这里的查询是泛指，也包括更新语句）的大致流程见下图：



如图所示，中间件的查询处理大致可分为优化和执行两个阶段，参与中间件查询处理的主要有执行计划和执行器两个组件，其中以执行计划组件为主。

优化阶段的任务是生成执行计划，主要包括语法解析、代数优化和生成计划这三个步骤：

语法解析：词法分析与文法分析，解析用户给出的语句生成语法树

代数优化：对语法树进行一些等价变换处理，如消除重复条件，条件的推导等，为计划生成作准备。这里也包括语义分析，如验证查询使用的表或字段都存在等

生成计划：生成执行计划

为提高性能，对 **PREPARE** 语句中间件采用了优化语法树缓存机制，即使用哈希表缓存了常用的 **PREPARE** 语句对应的经代数优化之后的语法树（进行语句统计时缓存优化前的语法树），因此对 **PREPARE** 语句，先要去语法树缓存中看对应的语法树是否存在，若存在则跳过语法解

析和代数优化这两个步骤（进行语句统计只跳过语法解析）。

为防止在优化阶段进行过程中分布式数据库的配置变化，在优化阶段开始时会锁定数据库配置，优化完成后再解锁。

执行阶段的任务是按照执行计划执行语句得到查询结果。对于不需要中间件进行联接操作的查询，主要包括目标节点计算、执行和结果后续处理这三个步骤：

目标节点计算：根据语句条件和数据库配置计划要操作的数据库节点。如语句中指定了均衡字段等值条件则只需要操作一个数据库节点，如果有多个均衡字段等值条件的 **OR** 条件则需要操作可能要操作多个数据库节点（这时还要进行语句拆分），若没有均衡字段等值条件则需要操作表所在的所有节点等。

执行：在需要操作的数据库节点上执行语句。这一功能主要由执行器模块完成，主要流程是先获取到这些数据库节点的连接，然后开启多个子线程并行执行（即使只需要访问一个节点时也使用子线程执行，这时为了使中间件的超时控制总是有效）

结果后续处理：中间件对各节点结果进行后续处理，如要操作多个节点的 **ORDER BY** 语句中间件需要将各节点结果进行归并排序等。

执行过程中需要锁定查询涉及的表，以防止这一期间这些表的定义发生变化。

对于 **PREPARE** 语句，优化阶段即对应于创建 **PREPARE** 阶段，即在用户调用 **Connection.prepareStatement()** 函数时完成，执行阶段在用户调用 **PreparedStatement.executeQuery()** 等函数时完成。

9.1.2 事务处理流程

DDB 支持事务处理的相关 SQL 语句：

```
begin/start transaction
commit
rollback
set autocommit=true/false
```

与 MySQL 一样，DDB 不支持嵌套事务，如果连续开启两个事务，而前一个事务不提交或回滚，则会抛错。

下面详细说明事务处理流程：

对于非 **autocommit** 模式，用户在调用 **Connection.setAutoCommit(false)** 时并不开启事务，而是在此后执行第一步语句时开启全局事务，这时这一全局事务的 ID 就确定了。但这时并不会在所有数据库节点上都开启一个分支事务，只有数据库操作真正要访问到某个节点时才在该节点上开启分支事务。这一事务将持续到用户调用 **commit/rollback** 或 **setAutoCommit(true)** 时才结束。

对于 **autocommit** 模式，一般情况下不会开启全局事务，但如果是需要中间件进行联接的查询，则在执行前自动开启一个事务，执行完成后立即提交这一事务。

分布式事务使用两阶段提交协议，即包含 **PREPARE** 和 **COMMIT** 两个阶段。为了处理在 **PREPARE** 成功后 **COMMIT** 之前发生故障的情况，中间件对分布式事务会记录分布式事务日志。具体来说，分布式事务的各个分支都 **PREPARE** 成功后，中间件会写一条 **PREPARE** 日志，并使用 **fsync** 保证这一日志被真正写出到磁盘上。日志内容主要是参与事务各数据库节点的 ID。然后在各个分支都 **COMMIT** 成功后，中间件会写一条 **DONE** 日志，这一日志只是写到日志缓冲区中，不一定立即写到磁盘上。

中间件使用多种技术来减少分布式事务的开销。首先是日志的成组提交，即在写 **PREPARE** 日志时并不是在主线程中直接调用 **fsync** 将日志刷出到磁盘上，而是等待一个定时刷日志线程来刷日志（默认是每 50ms 刷一次）。这样，即使对于具有很高并发度的应用，为写日志也不过每秒调用 20 次左右的 **fsync**。其次是只对真正需要的分布式事务才写日志。具体来说，如果一个分布式事务涉及的数据库节点中有超过一个的节点上对事务型表（当前专指 InnoDB 类型的表）进行了更新操作时才需要写分布式事务日志，并且对这些节点才进行两阶段提交，其它节点上只进行一阶段提交。比如以下情况都不需要写分布式事务日志和进行两阶段提交：

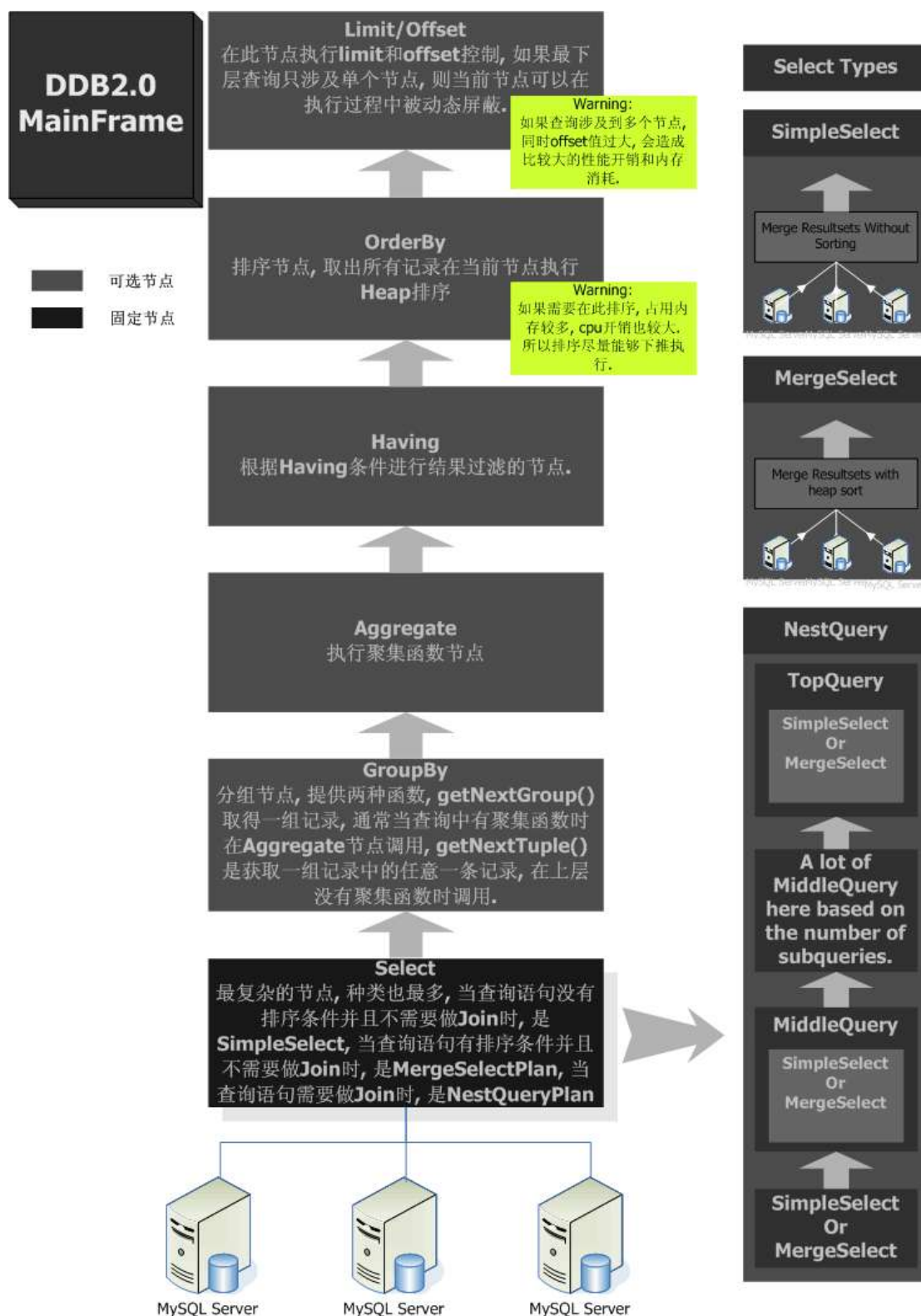
- 用户设置 **autocommit** 为 **false** 开启事务，然后执行了多条语句，但这些语句只操作了一个数据库节点；
- 用户设置 **autocommit** 为 **false** 开启事务，然后执行了多条语句，这些语句操作了多个数据库节点，但只有一个节点上的数据被修改；
- 用户设置 **autocommit** 为 **false** 开启事务，然后执行了多条语句，这些语句修改了多个数据库节点，但被修改的表都是 **MyISAM** 类型的

事务回滚不需要写日志。

9.2 SQL 语句执行计划

DDB 中运行 SQL 语句，中间件做的事情，基本流程就是把 SQL 语句按节点进行拆分，发往正确的数据库节点，运行后把结果汇总起来。整个语句拆分和结果汇总的执行流程，我们称之为 SQL 语句执行计划。不同于 MySQL 等数据库引擎提供的执行计划，DDB 的执行计划更加侧重于考虑如何有效正确的在不同的数据库节点之间分派任务并汇总结果。

下面是 DDB 执行计划总体框架图，细节在子章节中详细展开。



9.2.1 使用 explain 查看执行计划

语句在不同节点上执行, 同一个 SQL 语句, 在 DDB 中运行和在单个节点上运行, 效率可能相差千万倍。搞清楚一个 SQL 语句的执行计划, 对如何编写高性能的应用程序非常有帮助, 为此, DDB 专门提供了一个 explain 命令用于查看语句执行计划。

Explain 命令的用法如下：

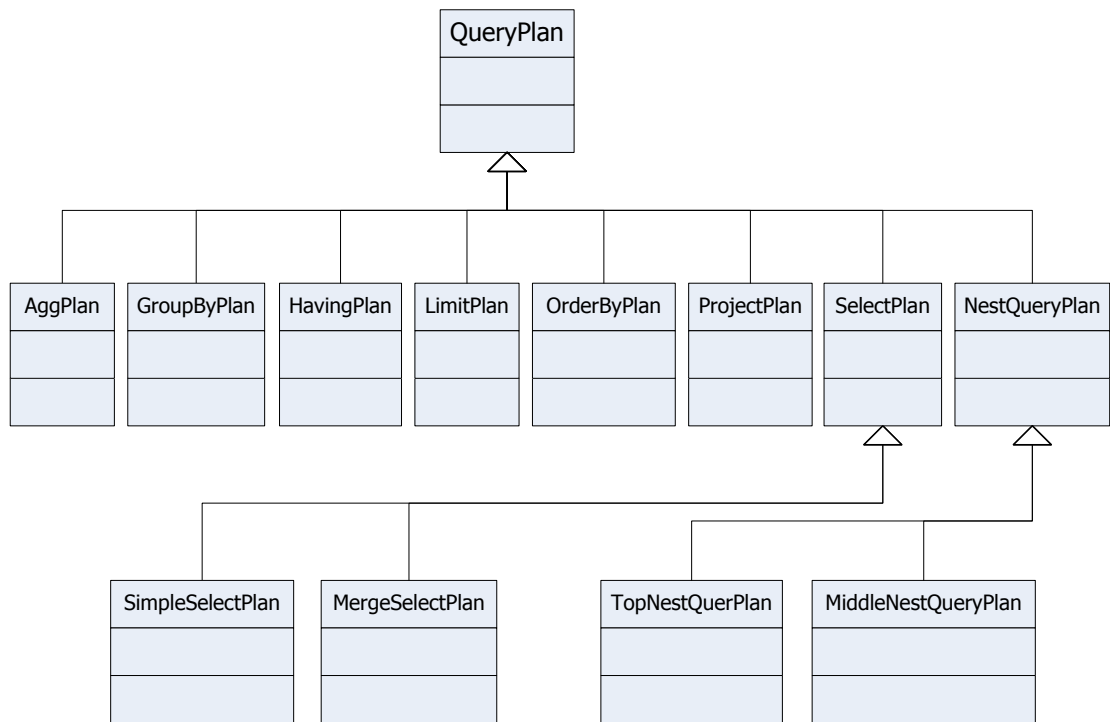
```
isql@dba>> explain select * from m3a;
+-----+
| PLAN                                     |
+-----+
| SIMPLE-SELECT                           |
| SQL: SELECT m3a.id, m3a.value, m3a.value2 FROM m3a |
| Dest Node:                             |
|   dbn2[jdbc:mysql://localhost:3306/dbn2]      |
|   fortest[jdbc:mysql://127.0.0.1:3306/fortest]  |
| Return records without sorting.           |
+-----+
6 rows in set, execute time: 31 ms
total time: 31 ms.
```

上面的执行计划是最简单的一种情况，没有涉及到表连接，没有额外的查询条件，下面说明一下输出信息的含义。

- ◆ **SIMPLE-SELECT**。表示这是一个简单查询，与此对应的是 **MERGE-SELECT**。对于 **MERGE-SELECT** 的详细说明和例子，在后面章节会详细讨论。
- ◆ **SQL**。此处显示了实际下发给数据库节点的 SQL 语句。
- ◆ **Dest Node**。语句执行的目标节点。上面的语句，由于没有指定任何的 **WHERE** 约束条件，因此会在该表所属策略下的所有数据库节点上执行该语句，然后进行结果汇总。对于简单查询来说，还有一种情况，就是对均衡字段指定等值条件，这样的话，**SQL** 语句就会只发送往正确的数据库节点，目标节点个数为 1。相比之下，在一个数据库节点上执行要比在所有节点上执行对系统的影响小，所以指定均衡字段的等值条件是 **DDB** 推荐的用法，应该尽量加上此类条件。
- ◆ **Return records without sorting**。由于该 SQL 语句中没有指定排序及分组条件，**DDB** 在获取到结果集之后，只需要简单合并输出给用户就可以，不需要做额外的操作。

9.2.2 查询计划分类

执行计划一共有 8 大类节点, 10 小类, 其关系图如下



9.2.3 简单查询计划（SIMPLE-SELECT）

简单查询计划出现在以下的场合：

- ◆ DIRECT FORWARD 模式。这种模式下，DDB 不做语法解析，直接把语句下发给数据库节点，把执行结果返回给用户。
- ◆ 语句不需要做复杂的跨节点表连接操作，并且满足：没有分组及排序条件；或者没有排序条件并且分组条件的对象是均衡字段。
- ◆ 出现在复杂查询中，作为复杂查询的一部分，负责对一个数据库节点进行操作。

最典型的简单查询计划的例子，已经在介绍 **explain** 命令的小节中给出。下面再介绍几种比较特殊的情况。

情况一：SQL 语句按节点分拆。

```
isql@dba>> explain select * from m3a where id = 1 or id = 2 or id=3;
+-----+
|                                           PLAN
|
+-----+
|                                           SIMPLE-SELECT
|                                           Splitted
|                                           As:
```

```

|      jdbc:mysql://localhost:3306/dbn2 --- SELECT m3a.id, m3a.value, m3a.value2
FROM m3a WHERE (id = 1) OR (id = 3) |
|      jdbc:mysql://127.0.0.1:3306/fortest --- SELECT m3a.id, m3a.value,
m3a.value2 FROM m3a WHERE (id = 2)      |
|                                     Return      records      without      sorting.
|
+-----+
-----

```

上面的例子显示，对于不同的节点，实际执行的 SQL 语句是不一样的。这是 DDB 额外增加的一种优化，如果用户输入的 SQL 语句的查询条件，可以被分拆，例如上面的语句，id 是均衡字段，查询条件的连接关系是 OR，则可以根据 DDB 记录的桶映射关系，把查询条件进行拆分，这样每个节点的 SQL 语句就不会包含明显不属于该节点的查询条件。这样的优化措施，可以减少 MySQL 数据库节点的执行代价。

情况二：语句中包含 DISTINCT 关键字。

DDB 在实际处理时，把 DISTINCT 看作与 GROUP BY 等同，因此，包含 DISTINCT 的查询计划就会变成如下的形式：

```

isql@dba>> explain select distinct id from m3a;
+-----+
| PLAN                                     |
+-----+
| SIMPLE-SELECT                           |
| SQL: SELECT id FROM m3a GROUP BY id      |
| Dest Node:                             |
|      dbn2[jdbc:mysql://localhost:3306/dbn2] |
|      forttest[jdbc:mysql://127.0.0.1:3306/fortest] |
|      Return records without sorting.    |
+-----+

```

9.2.4 合并结果集查询计划 (MERGE-SELECT)

合并结果集查询计划，含义是在从数据库拿到结果及之后，需要对结果集进行额外处理，然后返回给用户。这里指的额外处理，主要指排序和分组。

合并结果集查询计划出现在以下的场合：

- ◆ 语句不需要做复杂的跨节点表连接操作，并且有排序条件，或者有分组条件而且分组条件不包含均衡字段。
- ◆ 出现在复杂查询中，作为复杂查询的一部分，负责对一个或多个数据库节点进行操作。

合并结果集查询计划的 explain 输出如下：

```

isql@dba>> explain select * from m3a group by value;
+-----+
-----+
| PLAN                                     |
|

```

```

+-----+
| GROUP                                     |
|   Group By: value,                       |
|   /\                                     |
|  /||\                                    |
|   ||                                     |
| MERGE-SELECT                             |
|   SQL: SELECT m3a.id, m3a.value, m3a.value2 FROM m3a GROUP BY value ORDER BY |
|   value ASC |                           |
|   Dest Node:                             |
|                                           |
|                                           |
|                                           |
|                                           |
|           Order      by:      value      ASC,           with      heap      sort. |
|                                           |
+-----+

```

上面的语句中，由于 **group by** 的条件是一个非均衡字段，所以就产生了 **MERGE-SELECT**。

注意一下，上面显示的实际执行 **SQL** 语句中，加了个额外条件：“**ORDER BY value ASC**”。这是 **DDB** 执行的一种优化，利用底层数据库节点进行排序操作，然后对排序之后的结果集进行分组合并，**CPU** 代价和内存代价都会小得多。这种优化在 **DDB** 中称之为排序下推。

合并结果集查询计划同样有可能会产生 **SQL** 语句按节点拆分，方式与简单查询计划相同。

9.2.5 嵌套查询计划

嵌套查询是比较复杂的一种情况，出现在多表查询中。如果查询条件使得 **SQL** 语句无法在一个节点中一次执行完成，就需要把 **SQL** 语句进行分隔，分多次在不同节点上分批执行。下面通过实际例子说明嵌套查询的执行流程：

```

isql@dba>> explain select m3a.id,m3b.id from m3a, m3b where m3a.id = m3b.id;
+-----+
| PLAN                                     |
+-----+
| PROJECT                                 |
|   Project record to: m3a.id,m3b.id,     |
|   /\                                     |
|  /||\                                    |
|   ||                                     |
| TOP-INNER-QUERY                         |
|   SQL:SELECT m3b.id FROM m3b WHERE ? = m3b.id |
|   Input columns from outer query:      |
|   m3a.id                               |
|   /\                                     |
|  /||\                                    |
|   ||                                     |
| SIMPLE-SELECT                           |

```

```

| SQL: SELECT m3a.id FROM m3a |
| Dest Node: |
| dbn2[jdbc:mysql://localhost:3306/dbn2] |
| fortest[jdbc:mysql://127.0.0.1:3306/fortest] |
| Return records without sorting. |
+-----+

```

上面的 SQL 语句由于 **m3a**、**m3b** 两张表分属于不同的均衡策略，所以被分割成一个嵌套查询来执行。通过查询计划，能够形象的说明嵌套查询执行流程：

- ◆ 最底层是 SIMPLE-SELECT。这是一个简单查询：SELECT m3a.id FROM m3a。先从 m3a 表中把所有的 id 查询出来。
- ◆ 最上层是一个 TOP-INNER-QUERY。对于每一个 m3a 表中查询得到的 id 值，都执行一条语句：SELECT m3b.id FROM m3b WHERE ? = m3b.id

如果查询的表超过两张，而且彼此之间都没有均衡字段的等值关系或分属不同均衡策略，则查询计划会出现三层或以上，这时就会有“INNER-QUERY”出现，充当中间层的角色。

需要特别提醒的是，嵌套查询是非常消耗系统资源的，需要执行的 SQL 语句次数跟最外层的查询返回记录数相等。考虑到性能，应使得外层查询返回的结果数较少，同时对内层查询能够利用索引从而能够很快完成。

9.2.6 聚集函数查询计划 (AGGREGATE)

聚集函数执行计划的执行结果通常只有一条结果 只有当聚集条件同时参与了分组时才有可能有多条结果。目前支持 MAX, MIN, COUNT, SUM, AVG。

聚集函数查询计划出现的场合有：

- ◆ 查询中包含的均衡策略多于 1 个。
- ◆ 不是对均衡字段进行分组的情况。

下面的语句由于不是对均衡字段进行分组，所以需要做聚集函数查询计划。

```

isql@dba>> explain select min(id) from m3a;
+-----+
| PLAN |
+-----+
| AGGREGATE |
| Do: MIN(id) |
| /\ |
| /|\ |
| || |
| SIMPLE-SELECT |
| SQL: SELECT MIN(id) FROM m3a |
| Dest Node: |
| dbn2[jdbc:mysql://localhost:3306/dbn2] |
| fortest[jdbc:mysql://127.0.0.1:3306/fortest] |
| Return records without sorting. |
+-----+

```


9.2.7 分组查询计划 (GROUP)

分组查询计划的出现条件与聚集函数查询计划完全相同。具体例子已经在合并结果集查询计划中说明。

9.2.8 Having 查询计划

Having 查询计划依据 SQL 语句中是否包含 **having** 条件来决定是否出现。

9.2.9 Limit 查询计划 (LIMIT/OFFSET)

如果 SQL 语句中出现了 limit 条件，就会产生 Limit 查询计划。DDB 中的 limit、offset，除了在单个节点上执行的情况，否则都不会直接下发给数据库去执行，而是经过修改之后再发送到数据库节点。

```
isql@dba>> explain select * from m3a limit 1 offset 1;
-----+-----
|                                                                                               PLAN
|
+-----+-----
|                                                                                               LIMIT/OFFSET
|
|   This plan will be dynamicly set disable/enable while running based on the
underlying plan. |
|                                                                                               /\
|                                                                                               /||\
|                                                                                               ||
|                                                                                               SIMPLE-SELECT
SQL:      SELECT    m3a.id,     m3a.value,     m3a.value2   FROM    m3a    LIMIT    2
|
|                                                                                               Dest                                Node:
|                                                                                               dbn2[jdbc:mysql://localhost:3306/dbn2]
|                                                                                               fortest[jdbc:mysql://127.0.0.1:3306/fortest]
|
|                                                                                               Return          records          without          sorting.
|
+-----+-----
```

仔细查看上面的执行计划可以发现，用户的 SQL 语句，条件是 `limit 1 offset 1`，而

实际下发时，自动变成了 `LIMIT 2`。这是因为在多节点情况下，返回的数据集需要合并，这时下发给数据库节点执行的 SQL 语句的 `limit` 条件，就变成了原语句的 `limit` 值加上 `offset` 值。DDB 对返回的多个结果集合并之后再 `offset` 操作。

9.2.10 排序查询计划 (SORT)

如果 SQL 语句中出现了 `order by` 条件，就会产生排序查询计划。

```
isql@dba>> explain select id from m3a order by value;
+-----+
| PLAN                                     |
+-----+
| PROJECT                                 |
|   Project record to: id,                |
|   /\                                   |
|  /||\                                  |
|   ||                                   |
| MERGE-SELECT                           |
|   SQL: SELECT id, value FROM m3a ORDER BY value ASC |
|   Dest Node:                            |
|     dbn2[jdbc:mysql://localhost:3306/dbn2] |
|     fortest[jdbc:mysql://127.0.0.1:3306/fortest] |
|   Order by: value ASC, with heap sort.    |
+-----+
```

通过上面的例子我们发现，DDB 会对排序查询做一个特殊处理。如果 SQL 语句包含排序条件，而排序条件又没有出现在查询的结果列中，则 DDB 会自动把排序条件添加到查询列中。例子中的实际执行 SQL，查询结果列中自动增加了 `value`。这样做的原因是 DDB 在做结果集合并时，需要参考排序条件，而如果底层节点没有返回排序条件的值，合并是无法进行的。

9.2.11 更新查询计划 (UPDATE)

更新查询计划描述了对数据库中数据产生更新的 SQL 语句的执行计划，包括 `INSERT`、`UPDATE`、`DELETE` 三种语句。举个例子：

```
isql@dba>> explain insert into m3a values(1,'1','1');
+-----+
| PLAN                                     |
+-----+
| UPDATE                                 |
|   SQL: INSERT INTO m3a(id,value,value2) values(1,'1','1') |
|   Dest Nodes:                            |
|     jdbc:mysql://localhost:3306/dbn2      |
+-----+
```

更新查询计划同样有可能会产生 SQL 语句按节点拆分，`UPDATE` 语句和 `DELETE` 语句，拆分方式与简单查询计划相同。`INSERT` 语句，语句拆分的情况主要出现在 `INSERT` 多值。

```
isql@dba>> explain insert into m3a values(1,'1','1'),(2,'2','2'),(3,'3','3');
+-----+
-----+
|                                           PLAN
|
+-----+
-----+
|                                           UPDATE
|
|                                           Splitted           As:
|
|      jdbc:mysql://localhost:3306/dbn2 --- INSERT INTO m3a(id,value,value2)
values(1,'1','1'),(3,'3','3') |
|      jdbc:mysql://127.0.0.1:3306/fortest --- INSERT INTO m3a(id,value,value2)
values(2,'2','2')           |
+-----+
-----+
```

9.3 SQL 语句自动优化

DDB 中间件，在执行 SQL 语句时，会自动帮助用户进行 SQL 语句优化，前面在介绍语句执行计划的章节中，已经或多或少涉及了一些优化策略，本章节再专门针对语句优化这个主题，做一些补充说明。

9.3.1 语句整体优化

情况一：

中间件支持如下一些类型的查询：

```
select now();
select abs(-1) from Student;
```

这类语句的查询列中不包含任何表的字段，仅仅是标量函数，而其后可以有 from 关键字，也可以没有，所以需要在这一步中判断出是否所有的查询列都不包含任何的字段，以便对于这类语句单独处理，例如在发往数据库的时候，只需在任何一个节点执行，就能满足要求。

情况二：

有些出现在from之后的表可能是没有意义的，比如以下两类语句：

```
select t1.a, t2.b from t1, t2, t3, t4
select t1.a, t2.b from t1, t2, t3, t3 where t1.a = t2.b
```

可以将它们分别优化为：

```
select t1.a, t2.b from t1, t2
select t1.a, t2.b from t1, t2 where t1.a = t2.b
```

9.3.2 简单查询条件优化

类型	实例	限定条件	优化方法	优化结果
----	----	------	------	------

独立的数值表达式	Where 100 Where true Where false	可以是任意数值	不需要	Where 100 Where true Where false
独立的参数	Where ?	无	不需要	Where ?
独立的数学运算表达式	Where 100+200 Where 100*?	仅限数值和参数之间的数学运算	能在执行前完成的计算都提前进行	Where 300 Where 100*?
Comparison 表达式 (>, >=, <, <=, !=, =, like, is null, is not null)	Where userid = 2; Where name != 'Tom'; Where userid < 2 + 5; Where name like '%tom'; Where id is not null; Where User.id + 1 > User.id.Age;	如果 comparison 两边的表达式都属于同一张表, 那么写法没有限制. 如果两边的表达式属于两张表, 那么必须是简单的字段之间做 comparison, 不允许其他带有 +, -, *, / 的操作, 或者标量函数和聚集函数操作.	如果有数值之间的算术表达式, 可以计算出表达式的值.	Where userid = 2; Where name != 'Tom'; Where userid < 7; Where name like '%tom'; Where id is not null; Where User.id + 1 > User.id.Age;
In, not in	Where id in (1, 2, 3, 4) Where id not in (1, 2, 3, 4)	只支持一元的 in 和 not in 操作.	转化为 and 或者 or 连接的 comparison 条件.	Where id = 1 or id = 2 or id = 3 or id = 4; Where id != 1 and id != 2 and id != 3 and id != 4;
Between, not between	Where id between 1 and 4; Where id not between 1 and 4;	无	转化为 and 或者 or 连接的 comparison 条件.	Where id >= 1 and id <= 4; Where id < 1 or id > 4;
标量函数	Where length(name) > 5; Where length(name) = ?; Where length(name) > 1 + 2;	标量函数只能针对单表的单个或多个字段进行, 并且 comparison 的另一个表达式必须是数值或者参数, 数值表达式或同表字段的标量函数.	如果有数值之间的算术表达式, 可以计算出表达式的值.	Where length(name) > 5; Where length(name) = ?; Where length(name) > 3;
聚集函数	Where max(Score) = 90; Where max(score) = ?; Where max(score) > 1 + 2;	聚集函数只能针对单表的单个字段进行, 并且 comparison 的另一个表达式必须是数值或者参数或者数值表达式.	如果有数值之间的算术表达式, 可以计算出表达式的值.	Where max(Score) = 90; Where max(score) = ?; Where max(score) > 3;

9.3.3 AND 条件优化

实际使用中, 很多 and 条件由程序自动生成, 这种自动生成的 and 条件通常不会经过开发者的审查和优化, 就直接交付给中间件执行, 因此这些 and 条件可能会包含一些冗余, 不简练的信息, 也可能出现一些冲突的信息, 有些时候按照这种原始的没有经过优化的条件执行, 不会发挥 ddb 的最佳性能, 同时也会存在一些表面上看起来无法支持, 但是经过优化后, 可以支持的条件类型, 下表是一些常见的可能出现的情况.

待优化类型	实例	存在问题
条件冗余	...where ID1 = 10 and ID2 > 200 and ID1 = 10;	有重复的条件, 多了内存消

	...where ID1 > 10 and ID2 > 20;	耗, 以及遍历时的时间消耗, 可以通过优化减少无用条件, 减少内存消耗.
	...where ID1 = 10 and true;	
	...where ID1 = 10 and 1 < 2;	
	...where ID > 5 and ID is not null;	
条件冲突	...where User.ID > 5 and Blog.UserID = User.ID and Blog.UserID = 2;	冲突条件如果不经优化, 会发往很多个 dbn, 带来不必要的网络开销以及 db 的负载, 经过优化为 where false, 只需要从任一 db 获得查询列的 metadata 就足够.
	...where ID > 5 and ID is null;	
条件隐藏信息	Select * from User, Blog, Music where User.ID >= Blog.UserID and Blog.UserID >= Music.UserID and Music.UserID >= User.ID order by Music.UserID	这种语句可以明显的推导出 User.ID = Blog.UserID = Music.UserID, 排序条件是 Music.UserID, 但是第一个执行的子查询是表 User 的, 如果把排序条件替换成 User.ID, 则可以实现排序下推, 减轻中间件排序的开销.
	Select * from User, Blog where User.ID = Blog.UserID and Blog.UserID = 5 order by User.ID;	根据条件可以推导出 User.ID = 5, 是有固定值的, 因此排序条件可以删除, 减少发往 mysql 的性能开销, 在执行第一个 User 表的子查询时, 可以把 User.ID = 5 条件一起发过去, 增大区分度, 减少 join 的次数.

上表中提到的情况, 都是可以被优化的。具体优化方法, DDB 中间件借用了有向图来完成, 实现机理比较复杂, 就不再介绍。如果对这方面有兴趣, 可以查阅《网易_分布式数据库_查询优化及执行计划设计》或直接向 DDB 开发组成员咨询。

9.3.4 结果集流水线

DDB 支持流水线操作。只需要通过 Statement 的 setFetchSize(int size)接口, 就可以在大多数情况下使流水线生效。

流水线失效的情况是, 当用户执行的语句有 order by 条件或者 group by 条件, 并且不满足排序下推的条件, 这时需要在中间件取出所有的结果进行排序, 相当于中间件还是把所有的数据都保存了起来, 流水线实际上失去意义。

9.3.5 语句缓存

DDB 引入了复杂的查询优化算法, 容易导致中间件 CPU 性能降低。为了避免那些出现频率较高的 prepare 语句每次调用时都进行 sql 的解析和查询优化, 对已经语句解析和查询优化后的 prepare 语法树进行了缓存。

9.4 数据表主键 ID（PID）分配

DDB 对于分布式存储的数据库表提供统一的 PID 生成服务，保证各 DBN 上表数据 PID 的全局唯一性。

目前表 PID 的生成方式包括以下两种，集中式的 PID 自增方式和基于时间戳的分布式 PID 生成方式。

9.4.1 集中式的 PID 自增方式

指 PID 以自增的方式由全局唯一的 Master 生成。

PID 的分配范围为 $0 \sim 2^{63}-1$ 。

从实现的角度来说，最终的 PID 生成由 DBI 完成，所有的 DBI 同步于 Master，即每次从 Master 申请一批（一般为 1000，可配置）某个表的 PID 资源，然后仍然按照自增的方式分配给 Client 使用，当分配完成后，将同步地去 Master 获得一批新的 PID 资源。

注：

Client 应用根据其类型和需求对每次 DBI 获得 PID 资源的数量进行合理的配置：

- 如果值太小将可能导致 DBI 频繁与 Master 进行同步从而影响应用的效率；
- 如果值过大则可能造成 PID 资源的浪费。

9.4.2 基于时间戳的分布式 PID 生成方式

与集中式的 PID 分配方式不同，分布式的 PID 分配可以完全分布式的由 DBI 完成。

PID 定义

基于时间戳的分布式 PID 生成策略所生成的 PID 格式如下，由三部分组成，定义如下（格式为十进制）：

```
<  timeStamp  ><   id   ><   sn   >
|<-- ? digits -->|<-- 3 digits -->|<-- 3 digits -->|
```

其中：

- timeStamp 指分配表 PID 时 DBI 的时间戳（单位：毫秒（ms））；
- id 指 DBI 的 ID，DBI 作为 Master 的客户端，拥有唯一的 ID，可分配的 ID 范围为 1～999（共 999 个可用 ID，0 号 ID 保留）；
- sn 是自增的序列号，范围为 0～999（共 1000 个序列号值）。

功能与特性

为了保证 PID 分配的正确性，即保证 PID 分配的全局唯一性，以下是 DDB 所保证的：

- 每个当前可用的 DBI 拥有唯一的 PID，保证各 DBI 分配的 PID 不会冲突。
- Master 保证各 DBI 上 PID 分配的一致性，即保证同步给各 DBI 的时间戳一致而且单调递增的。因此除了保证各 DBI 时间戳基本一致外，Master 能避免如下情况：
如果某 DBI 已经分配了部分 PID 资源，然后该 DBI 的 UID 被回收之后被另外一个 DBI

所使用时，此时新的 DBI 所获得的时间戳一定大于先前 DBI 所获得的最大时间戳。

- DBI 上保证分配 PID 时所采用的时间戳与 Master 的基准时间戳在一定可接受差别内，并保证每次分配 PID 时所使用的时间戳是单调递增的，这能保证：①当前进程内分配的 PID 一定是递增的；②重启后该 DBI 上用于 PID 分配的时间戳一定大于上次所使用过的最大时间戳。
- DBI 保证用于 PID 生成的序列号字段在时间戳前缀确定情况下是单调递增的。

通过上述措施，DDB 除了保证了 PID 分配的正确性外，也提供如下的特性：

- 分布式的方式下 PID 分配的效率更高，应用在分配 ID 时并发度和效率更高。
- 分布式的 PID 分配对 Master 的依赖性更低，不存在单点故障，系统可用性更高。
- 既没有 PID 资源的可能浪费，也基本上保证了理论上不可用尽的 PID 资源。
- PID 中包含了分配该 PID 的时间，该值以较高地精度保证了与当前实际时间戳的值相等，因此对某些需要记录创建时间的数据表，省却了创建时间字段，可以通过其前缀直观地获得；
- 对于某单节点的应用而言，保证应用所获得的 PID 根据分配时的时间先后排序，同时也较高精度地保证了应用在 DDB 不同节点所获得的 PID 按分配时间排序，该精度保证源于 PID 所包含时间与全局时间戳的基本相等（更详细的介绍参见下两节）。

PID 生成过程

本节将简要描述 PID 的生成过程，以使应用开发人员和 DBA 更好地了解 PID 的生成方式，解决可能遇到的问题。

1、术语与定义

- **PID 基准时间戳（Master 时间戳、基准时间戳）**
指 Master 的时间戳，精度为毫秒（ms），即相对于 1970 年 1 月 1 日（UTC 标准时间）的毫秒数，在 DBI 会被转换为纳秒（ns）以方便计算。该值是理论的绝对正确值，或者认为其是正确的参考值。
- **PID 基准时间差（时间差）**
指某次同步时 DBI 与 Master 时间戳差值（参见 PID 基准时间差的计算过程一节），单位为纳秒（ns）。在使用该时间差的过程中，认为其是正确的，即与使用该值时真正的时间差是一致的。在使用该值时，如果可能已经不一致，DDB 有相应的措施保证其偏差在可以接受的范围内才使用。
- **PID 基值**
该值即为 PID 基准时间戳的实际值，即用于 PID 生成时的实际值，通常根据 PID 基准时间差和本地时间戳计算得出。该值嵌入 DBI 的 ID 和自增的 ID 即为最终生成的 PID。该值与实际值可能有差别，不一致时的处理方式即基于 PID 基准时间差不一致时的处理过程。

2、PID 生成过程

一次完整的 PID 的生成过程包含三个步骤

- i DBI 与 Master 同步获得 PID 基准时间戳，并计算 PID 基准时间差；
- ii DBI 根据 PID 基准时间差和本地时间戳计算 PID 基值；

iii DBI 根据 PID 基值生成 PID

以下将对部分过程做说明。

3、PID 基准时间差（时间差）的计算过程

为了保证 DBI 和 Master 之间时间差的计算的精确，DBI 采用如下的方式获取并计算其值。

1. 记录当前时间点（ns，非时间戳）作为过程开始时间 `startTime`;
2. 向 Master 请求时间戳 `msTimeStamp`;
3. 记录过程结束时间（ns，非时间戳）作为过程结束时间 `endTime`

计算差值（其中“toNanos”指将毫秒（ms）转换为纳秒（ns））：

```
(endTime + startTime) / 2 - toNanos(msTimeStamp)
```

注：

这里计算出的差值并不是 DBI 的时间戳和 Master 的时间戳的差值，单独看这个值没有意义，因此“`System.nanoTime()`”衡量的是时间段，仅与 CPU 的时钟频率有关，而 Master 返回的是时间戳，即具体的时间点。

但这个值可以用于：

- 下次同步时的所用同样方式算出的差值与当前差值的差（精度为纳秒（ns）），用于衡量 DBI 的时钟频率是否与 Master 的时钟频率（假定其正确或以其为参考值）一致。
- 计算 PID 基值，即在两次同步周期之间，基于 DBI 所流逝的时间与 Master 流逝的时间一致的前提，计算出接近于当前点的 PID 基准时间戳估计值作为 PID 基值。
如果 DBI 的时钟频率异常，见“PID 基准时间戳同步”一节的“同步过程”部分的问题分析。

4、PID 基准时间戳同步

（1）同步过程

目前，DBI 会周期性地与 Master 同步以获得 Master 的时间戳作为 PID 基准时间戳。

同步过程由每个 DBI 发起，而不是由 Master 进行通知。理论上来说，应该由 Master 发起与当前连接的各 DBI 的定期同步，因为只认为 Master 的时间戳和时钟频率是绝对正确的。

在实际的设计中，基于 DBI 的时钟频率发生错误的可能性很低、误差可以接受或忽略的前提，采用 DBI 端发起周期性及按需的同步。

- 每次同步所获得 Master 的时间戳作为新的 PID 基准时间戳（虽然可以在上次获得的 PID 基准时间戳的基础上加上同步周期，但可能会出现偏差）；
- DBI 可以按需同步。

可能出现的问题：

▪ DBI 时钟频率过慢

时钟频率过慢产生的影响不大，只会导致 PID 基值所采用的时间戳落后于 PID 基准时间戳，且最大偏差不会大于实际的同步周期。但在非常极端的情况下，即 DBI 时钟频率太慢，而 PID 分配请求量又非常大，将可能经常会出现 PID 资源不够导致 PID 分配等待过久或失败的情况，如正常情况下每个 DBI 每毫秒(ms)可分配约 1000 个 PID，如果时钟频率慢 100 倍，将导致每毫秒只能分配最多约 10 个 PID。

▪ DBI 时钟频率过快

如果在一个同步周期（正确的）内的时钟频率过快导致的与正确时间戳的偏差在可接受范围内，不会有问题。

反之，则会较多次数的出现同步获得的 PID 基准时间戳小于已用于 PID 分配的 PID 基值，即预支了未来的 PID 资源。如果在当前应用进程中，结合“PID 基值生成过程”的异常情况处理，仍然可以保证 PID 分配的正确性，但是将不能保证可能出现的全局错误导致的 PID 重复冲突，即因为此次预支了未来的 PID 资源，当 DBI 重启或 DBI 的 UID 释放后分配给其他应用启动时会导致 PID 重复的冲突，此时的解决方法为：禁用此 DBI 的 UID 一段时间即可。

（2）失败处理

若 DBI 与 Master 的基准时间戳同步失败，将会影响 PID 的生成方式。这种影响是间接的，因为生成 PID 基值（见“PID 基值生成过程”）时会检查 DBI 本地时间戳与上次同步的 PID 基准时间戳的关系，确认是否在接受的范围之内。

假设 DBI 本地时钟稳定，同时基于 Master 的基准时间戳是可信赖的，PID 基准时间戳同步失败时，将可直接使用 PID 基准时间戳加上上次同步的时间点与现在的时间点的差值来生成新的 PID 基值。如果没有记录上次同步的时间点与生成 PID 基值的时间点的差值，那可以忽略差值，而使用“上次的 PID 基准时间戳+同步周期”做为新的 PID 基准时间戳估计值。它能保证：

- 新生成的 PID 基值大于以往的 PID 基值的最大值，不会出现 DBI 端 PID 冲突；
- 与真实的 PID 基准时间戳的误差在可接受范围之内（同步周期），是保证全局 PID 不冲突的必要条件。

它可能存在如下问题：

但如果结合可能出现的 DBI 时钟频率不正确的问题，将会出现不一样的结果。比如 DBI 时钟频率过快，上述的处理方式将导致估计修正值“上次的 PID 基准时间戳+同步周期”反而会导致其与真实的 PID 基准时间戳的偏差越来越大。

这种可能性发生基于 DBI 时钟频率误差很大的情况，基本不会发生，因此忽略。

5、PID 基准时间差变化过大的处理

若 DBI 本地计算得到的 PID 基准时间差与上次所计算得的值相比偏差较大，在认为 PID 基准时间戳绝对争取的情况下，可能由 DBI 的时钟的不稳定所导致。参见“同步过程”一节的处理。

偏差过大时需要及时发现并报警以及及时处理。

允许的最大差值是可配置的，具体参见“DBI 端的 PID 生成相关配置”一节。

6、PID 基值生成过程

（1）生成过程

PID 的基值根据 PID 基准时间差和自上次以来流逝的时间计算得知。即在两次同步周期之间，基于 DBI 所流逝的时间与 Master 流逝的时间一致的前提，计算出接近于当前点的 PID 基准时间戳估计值作为 PID 基值。

在计算 PID 基值时会先判断 DBI 当前和上次与 Master 同步时的时间差值（称为 δ ）是否超出可接受的范围。如果 DBI 的本地时钟自上次同步以来是正确的，那么 δ 的取值范围应该是 $[0, \text{DBI 与 Master 的同步周期}]$ ，超过同步周期没有意义。

如果 DBI 的本地时钟出现异常，将导致 δ 出现异常，如果只允许 δ 的值在 $[0, \text{DBI 与 Master}$

的同步周期), 那同样能得出 PID 的基值的最大误差即为 DBI 与 Master 的同步周期。

(2) 异常情况处理

如果偏差超出可接受的范围, 将采取如下方式生成 PID 基值:

当前的 PID 基准时间戳+同步周期

如果该值不大于上次的 PID 基值 (如 PID 基值时间戳服务失败的情况), 将使用如下方式生成 PID 基值:

上次的 PID 基值+1

正确性说明:

如果上次获得 PID 基值是基于 PID 基准时间戳生成, 那其值一定不大于“上次的 PID 基准时间戳+同步周期”, 当然也更不会大于“当前的 PID 基准时间戳+同步周期”, 因此可采用后者来生成 PID 基值。否则, 说明在 PID 基准时间戳未更新的情况下, 连续出现此种情况, 因此采用“上次的 PID 基值+1”, 这两种情况下的处理方式都能在保证 PID 正确性的基础上获得最接近实际时间戳的有效 PID 基值。

这种解决方法:

- 可以保证当前进程中生成的 PID 的唯一性;
- 在绝大部分情况下将落后 PID 基值所对应的真实的时间戳落后于真实的 PID 基准时间戳, 避免预支未来的 PID 资源导致可能造成的全局 PID 冲突。

配置

1、表 PID 配置

表可以使用不同的 PID 分配策略, 主要有以下参数:

- **genType:**
表采用何种 PID 分配策略, 默认为全局配置, 该值在建表时指定后不能再修改, 若修改可能导致 PID 冲突!
- **numLimit:**
采用集中式的 PID 策略时 DBI 一次从 Master 分配得的最大 PID 数量, 默认值 1000。

注:

在使用 ISQL 工具建表时使用如下 hint 指定表的 PID 类型:

```
/*assignidtype=msb|tsb*/
```

其中, “msb”为集中式的自增方式, “tsb”为基于时间戳的分布式 PID 生成方式。

2、DBI 端 PID 生成相关配置

(1) 配置定义

基于时间戳的分布式 PID 分配方式相关配置部分用于 DBI, 其他用于配置 DBI 和 Master 之间的时间戳同步。为了简化, 目前这些配置 DBI 不能就修改, 而使用从 MSever 获得的统一配置。

- **SyncPeriod**
指 DBI 获取 Master 的时间戳作为其 PID 分配的基准时间戳的同步周期。

- **MLTD (Minimal Legal Time Deviation)**

衡量 DBI 的时钟频率是否与 Master 的时钟频率一致，需结合同步周期进行配置。通常情况下该差值保持恒定，若出现较大偏差，说明 DBI 的本地时钟有较大的偏差。虽然绝大部分情况下不会影响 PID 分配的正确性，但这种情况需要即时获知并向 DBA 报警以解决。

- **TTF**

PID 资源的时效性，该值用于保证 DBI 上用于衡量生成 PID 基值与生成 PID 时真实的 PID 基准时间戳的差值的可允许范围（绝对时间精度）。

当生成 PID 基值时，PID 的生成将根据其最后组成部分即序列号部分的自增完成。因此该值用于保证自增序列号时生成的 PID 值与真实的 PID 基准时间戳不至于差别太大而影响 PID 的时效性，这也保证了 PID 分配严格按时间序（原因见上节所述）。

如果该值设为最大，那将使 PID 生成的效率最高，但 PID 的时效性可能较差，对于不要求时效性的应用可以采用此配置；

该值结合 syncPeriod、MLTD 构成了 PID 包含的时间戳与真实的 PID 基准时间戳的可能最大差值，然而，在绝大部分正常情况下，TTF 即该差值。

该值不可配置，DDB 内确定为 1ms；

其他默认配置：

- **ignoreSyncError**

是否忽略同步错误，默认认为是。若出现同步失败但忽略的情况，将可能会对生成的 PID 产生影响。

(2) 如何配置

- **syncPeriod**

主要考虑配置合适的值，基于 DBI 与 Master 之间交互的可能情况以及给 Master 带来的负载。更重要的是，该值同样代表了 PID 基值与计算 PID 基值时真实的 PID 基准时间戳的最大差值，要考虑到 AServer 重启所花的时间等，主要用于避免 DBI 时钟不正常而后重启后或其 UID 分配给其他 DBI 所可能导致的 PID 重复冲突的问题。默认值为 15min。

- **MLTD**

避免 DBI 的时钟频率与 Master 的相差过大，涉及到与 syncPeriod 同样的问题。另外，也是 PID 的绝对时间精度的重要保证之一。如同步周期为 5 分钟，结合同步可能产生的计算误差，可配置差值为 5s（1%×同步周期+误差）或更大。

默认值为 2s。

常见异常情况处理

本节描述 PID 生成时常见的异常情况和处理方法。

<待添加>

9.5 两阶段提交机制

9.5.1 两阶段提交模型概念

事务管理器和资源管理器

通用的分布式事务架构由应用程序、事务管理器和资源管理器三部分组成。其结构图如下所示：

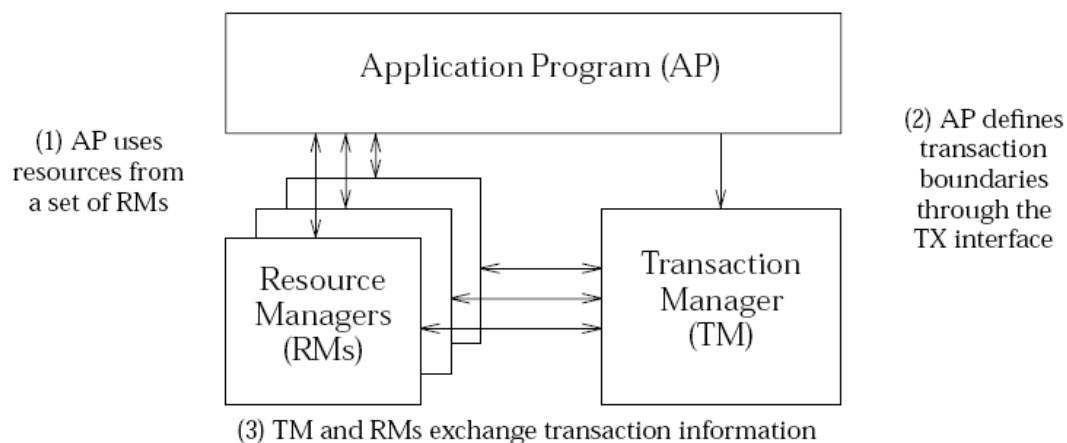


图-1 分布式事务框架图

分布式事务管理器用于管理一组分布式事务。其功能主要包括：协调分布式事务的提交或回滚、提供事务容错保证。其框架图如下：

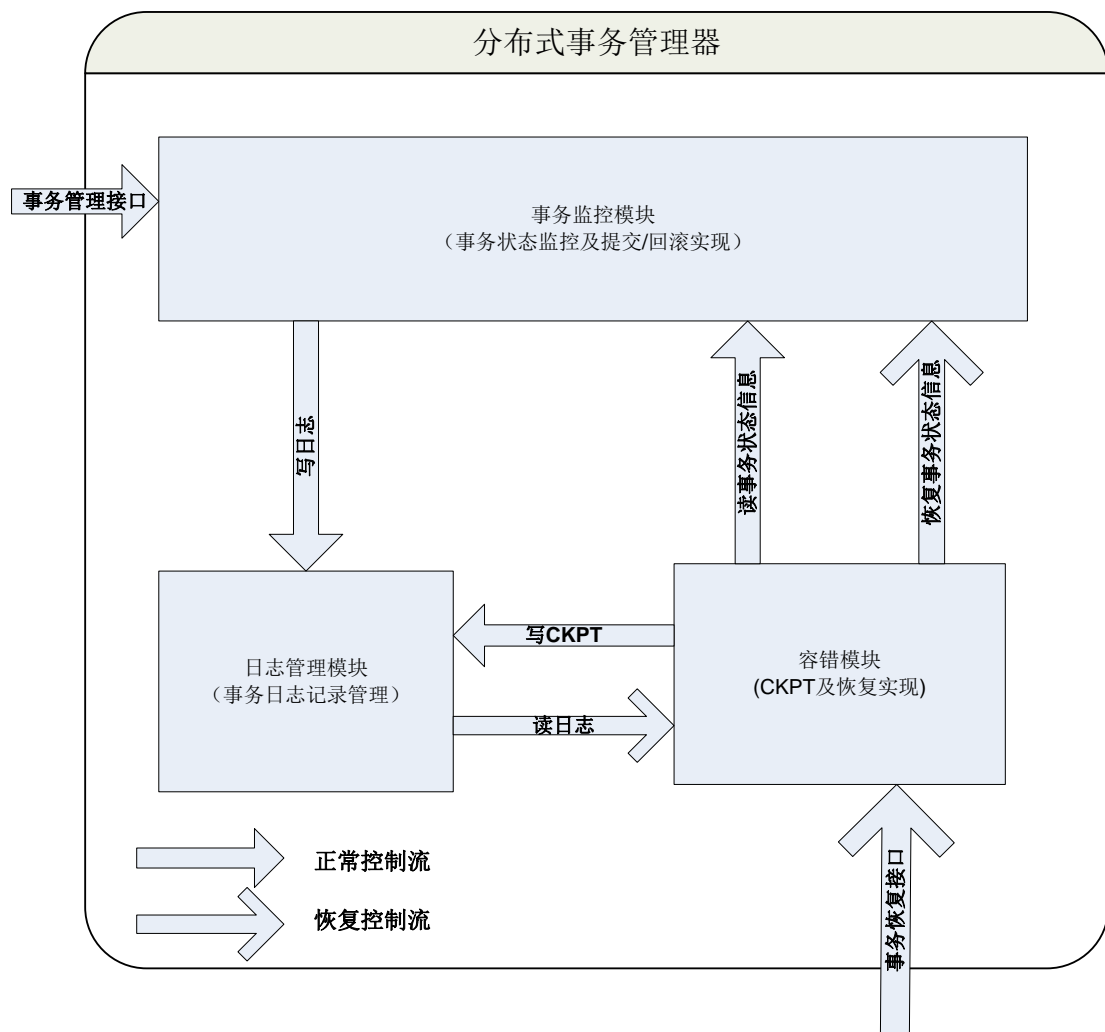


图-2 事务管理器框架图

资源管理器一般指的就是数据库节点。

两阶段提交协议描述

两阶段提交协议实现分布式事务提交/回滚，保证分布式事务的原子性。在本章，简述经典的二阶段提交实现。

提交

成功的两阶段提交由以下顺序步骤组成：

- **准备** 调用事务资源列表中的每个资源管理器，要求它们投票。
- **决定** 如果所有资源管理器投票赞成，永久性的写入事务提交的日志纪录。
- **提交** 调用列表中的每个资源管理器，告诉它提交决定。
- **完成** 当所有资源管理器都对提交消息做出了应答，在日志中写入提交完成纪录，释放事务。

回滚

如果任一资源管理器在准备阶段投反对票，或者没有应答，必须回滚事务。

回滚步骤如下：

- **撤销** 反向读事务日志，对每条记录发出 **undo**，调用写该记录的资源管理器来执行 **undo** 操作。
- **广播** 在每个保存点（特别在保存点 0）调用参与事务的所有资源管理器。
- **中止** 在日志上写入事务中止纪录。
- **完成** 在日志写一条完成记录表明中止结束，释放事务。

错误情况讨论

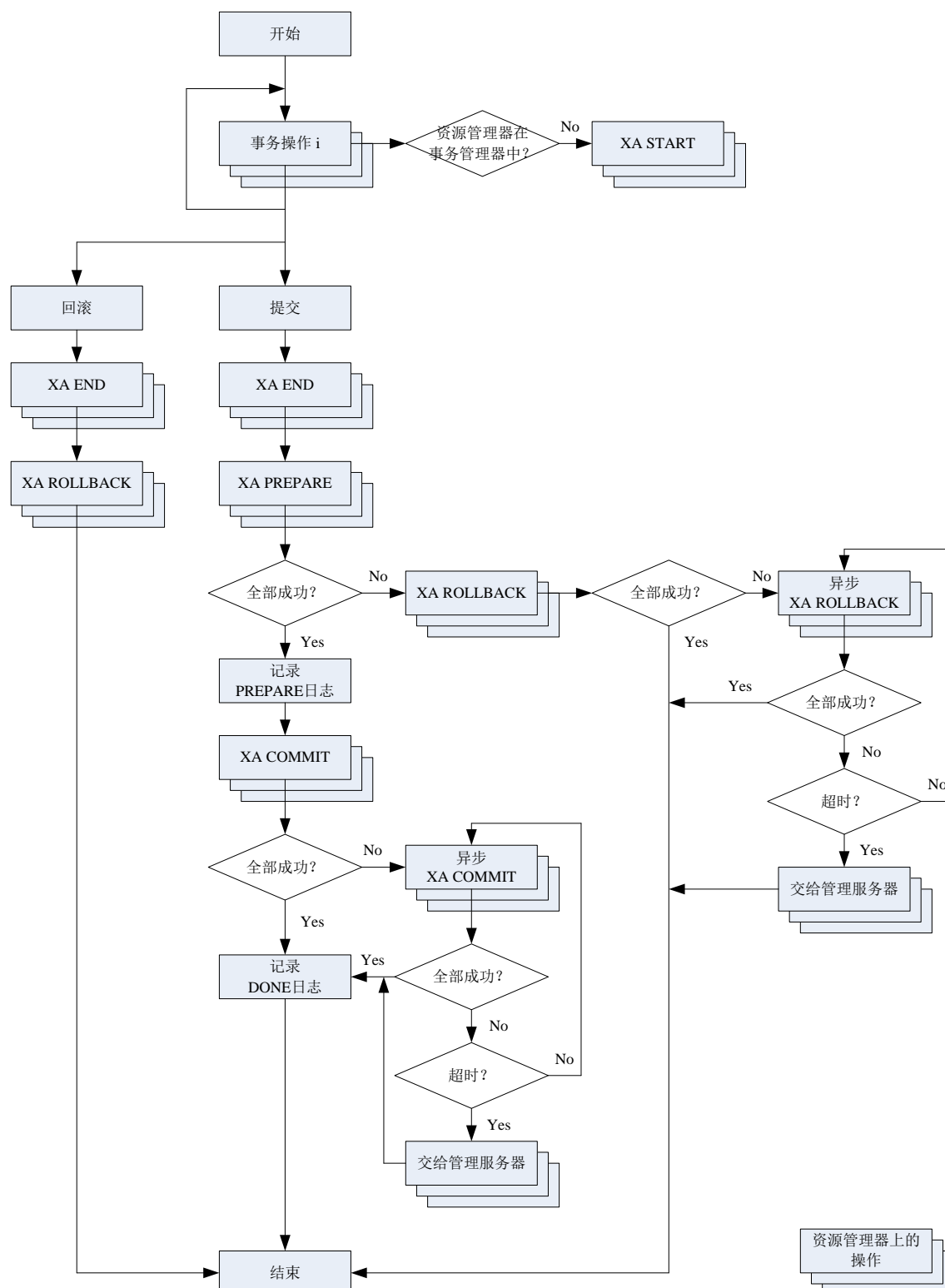
资源管理器在二阶段时的容错，通过事务管理器的超时和 **PREPARE_COMMIT_ROLLBACK** 机制保障。

事务管理器在二阶段时的容错，主要通过如下机制保证：

1. 每个资源管理器在 **PREPARE** 成功之前，不写日志。因此，事务管理器在 **PREPARE** 之前出错，不会造成事务悬挂。
2. 资源管理器在 **PREPARE** 之后，定时询问事务管理器端的相应事务活动状态，如询问不到，回滚事务，防止事务管理器在 **PREPARE** 之后出错。
3. 在 **COMMITTING** 开始时，事务管理器强制写日志，防止 **COMMITTING** 开始之后事务管理器的出错。
4. 在 **COMMITTING** 阶段，事务管理器调用资源管理器提交失败，创建异步线程反复提交，直到成功。这里为了处理提交过程中的各种出错情况，包括事务管理器端，资源管理器端和网络。
5. 当所有提交都完成时，事务管理器惰性写完成日志。表示事务完成。
6. 对于 **ROLLBACK** 不成功或 **ROLLBACK** 时事务管理器出错情况，由资源管理器通过定期询问或超时机制保障。

9.5.2 DDB 分布式事务管理

DDB 分布式事务处理实现遵循两阶段协议标准。通用的分布式事务架构由应用程序、事务管理器和资源管理器三部分组成。MySQL 从 5.0 版本开始支持 **XA Transaction** 的资源管理器（**Resource Management**）接口，分布式数据库 DBI 则实现 **XA Transaction** 的事务管理器（**Transaction Management**）功能。在 DBI 实现的事务管理器中，一个典型的分布式事务流程如下图所示（一些优化操作可能不包含图中的某些步骤，见后面的描述）：



容错机制讨论

分布式数据库中间件实现经典的两阶段提交（Two-Phase Commit, 2PC）协议，保证事务的原子性。为支持分布式事务故障处理，分布式数据库中间件使用日志记录分布式事务的准备（prepare）及提交操作。

以分布式事务日志为基础，分布式数据库中间件采用多种手段，提供了较完善的分布式事务故障处理解决方案。对于事务管理器失效之外的分布式事务故障，系统首先重试失败的操

作，试图解决暂时性故障。若重试不能解决，系统将创建一个事务处理异步线程处理失败的操作，并立即返回应用事务成功与否，防止应用长时间被阻塞。该事务处理异步线程将提交/回滚事务动作一段时间，若不成功，则将失败的操作信息发送给管理服务器进行统一处理（通常这可以解决后台数据库节点与应用服务器节点之间网络链路失效导致的故障）。

对于事务管理器失效故障，由于在系统使用日志记录了分布式事务操作信息，在重启事务管理器后，对于日志中有 **PREPARE** 日志但没有 **DONE** 日志的事务，系统会自动进行日志 redo 操作，解决故障。由于在事务管理器失效至重启之前，管理服务器可能同时也在试图解决资源管理器中的悬挂事务问题，因此为了简化恢复的工作，并保证正确处理悬挂事务，redo 的操作交由管理服务器处理。

若在极端情况下，上述所有措施都不奏效，只要分布式事务日志不丢失，系统管理员仍可以通过分析日志和查询各后台数据库节点分布式事务状态得出正确的操作。

性能优化

分布式事务处理器另一关键技术是要尽量提高事务处理的性能。为此，分布式事务的开始操作被设计成“延迟”式的，即在开始一个全局分布式事务时系统并不立即在每个后台数据库节点上开始一个局部事务，而是在事务中的操作第一次真正需要访问某个数据库节点时才在此节点上开始局部事务。这一方法最大限度减少了分布式事务操作数量和参与分布式事务的数据库节点数，提高了分布式事务处理性能。其次，分布式事务的准备、提交或回滚等操作都通过多个事务子线程并发执行。

为了提高事务处理的性能，还优化了分布式事务提交时的处理方式。通常，分布式事务在提交时总是采用两阶段提交策略，并在 **prepare** 和 **commit** 成功时分别记录日志（**PREPARE** 日志需要同步到磁盘，**DONE** 日志不需要）。但在某些特殊的情况下，某些分支甚至整个事务都可以优化为一阶段提交：对于只读或只访问 **MyISAM** 表的分支，“一定”采用一阶段提交；如果不“一定”采用一阶段提交的分支数不多于一个，则整个分布式事务都采用一阶段提交；对于采用一阶段提交的分布式事务不记录事务日志。

9.5.3 存在的问题

MySQL 对 XA Transaction 的支持并不完善，存在以下两个问题：

1) 一个分布式事务准备成功后，若事务管理器断开，正确的做法是数据库应该保持该事务在 **prepared** 状态，因为事务管理器可能已经通知其它参与该分布式事务的数据库节点提交这一事务，但目前 MySQL 采取的做法是回滚这一事务。为此，我们分析了 MySQL 事务处理相关代码，并进行了修改，制作了专门的 MySQL 补丁。经测试修改后能够解决上述问题。

2) 一个分布式事务准备成功后，若该连接关闭或数据库异常退出，则服务重启后可以使用新的连接来提交事务，但在 **binlog** 中却不会有记录，即会导致 **binlog** 与数据不同步，使系统无法与 **binlog** 复制机制共用。MySQL 的 **binlog** 实现较为复杂，该问题暂时无法解决。

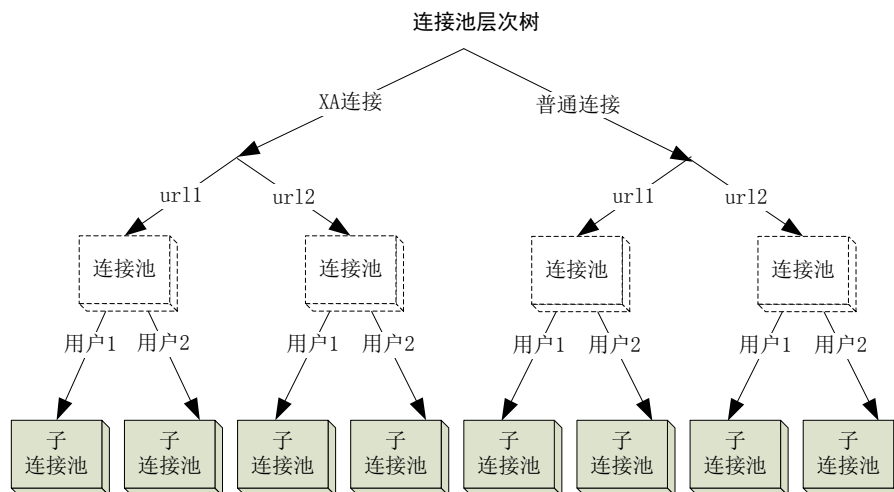
9.6 连接池管理

中间件到各后端数据库节点的连接使用连接池进行管理。连接分为以下两类：

XA 连接：执行包含在事务之中的数据库操作时使用 **XA** 连接，如上所述，这有可能是由于 **autocommit** 为 **false** 或需要在中间件进行联接操作。

普通连接：要进行数据库操作不在事务之中时使用普通连接。

对每类连接，每个在查询处理中曾经使用过的数据库节点都对应一个连接池。但这一连接池实际上并不包含连接，真正包含连接的是该节点上某用户对应的子连接池。即每个子连接池管理某指定数据库节点上某指定用户的 **XA** 连接或普通连接。这样，中间件中的连接池就构成了如下图所示的连接池层次树。



当上层程序需要获取连接时，系统首先检查对应的连接池是否存在，若不存在则创建一个空的连接池。然后检查对应连接池中是否有空闲连接，若有则返回一个空闲连接，否则新建一个连接。上层程序释放连接时是先将该连接作为空闲连接缓存中连接池中，只有空闲连接长时间未被使用（这一时间可以配置）时才会被关闭。

连接池中的连接可能被执行器中的事务对象或逻辑语句对象持有。语句对象当上层使用执行器提供的逻辑语句对象执行数据库操作时，对每个要操作的数据库节点，逻辑语句对象首先获取一个连接并增加连接的引用次数。如果该逻辑连接已经使用过到某数据库的连接，则重用这一连接，否则会从连接池中获取一个新连接。如果在事务之中，则这些连接还会被加入到事务中，再次增加引用次数。逻辑语句对象关闭时减少其所用的所有连接的引用次数，类似的事务结束时也减少其所用的所有连接的引用次数。如果某连接的引用次数为 0，则会被放回连接池。

查询处理上层总是在每次执行时创建一个新的逻辑语句对象，使用该逻辑语句对象执行产生的结果集被关闭后就关闭语句对象。而在默认不使用游标的情况下，中间件在执行查询时总是先算出了所有结果后再返回给上层应用，这时所有底层的结果集都已经被关闭。

因此对于用户来说如果应用没有使用事务和游标，则总是在语句执行开始时从连接池中申请连接，语句执行结果后就释放这些连接。如果使用了事务，则是在语句第一次访问到某节点时从连接池中申请连接，保持这些连接到事务结束后全部释放。如果使用了游标并且中间件的流水线生效时，则在 `executeQuery` 返回时底层的结果集可能没有关闭，这时应用取完了结果集中的所有结果或关闭结果集时连接才会释放，此外应用关闭的连接或 `Statement/PreparedStatement` 时也会释放连接。

为防止过多的连接占用过多资源影响到系统性能，连接池具有大小限制。注意这一大小限制是限定某数据库节点对应的连接池，而不是该节点上某用户对应的子连接池。子连接池本身没有大小限制，属于同一个节点的所有子连接池大小总和不能超过上述限制。当子连接池中没空闲连接且总大小达到上限时，连接池管理模块首先会通知同一数据库节点的其它子连接池释放一些空闲连接为该子连接池腾出一些空间。**XA** 连接池和普通连接池的大小分别独立设置。

9.7 用户 quota 配额限制

对于关系数据库来说，对一张大表进行一次表扫描，会消耗节点大量的 CPU 资源及 IO 资源。如果要对大表进行跨节点的表连接操作，消耗的代价会更大。为了避免用户由于不小心输错一个 SQL 语句，导致资源大量消耗，影响到其他用户的正常操作，DDB 引入了用户 quota 配额限制的概念。

用户配额分两种：一般节点配额和 MySQL 复制从节点配额。把这两种配额区别开是因为在从节点上，往往会允许跑一些资源消耗较大的应用，而且此类操作也不会影响到正常的线上服

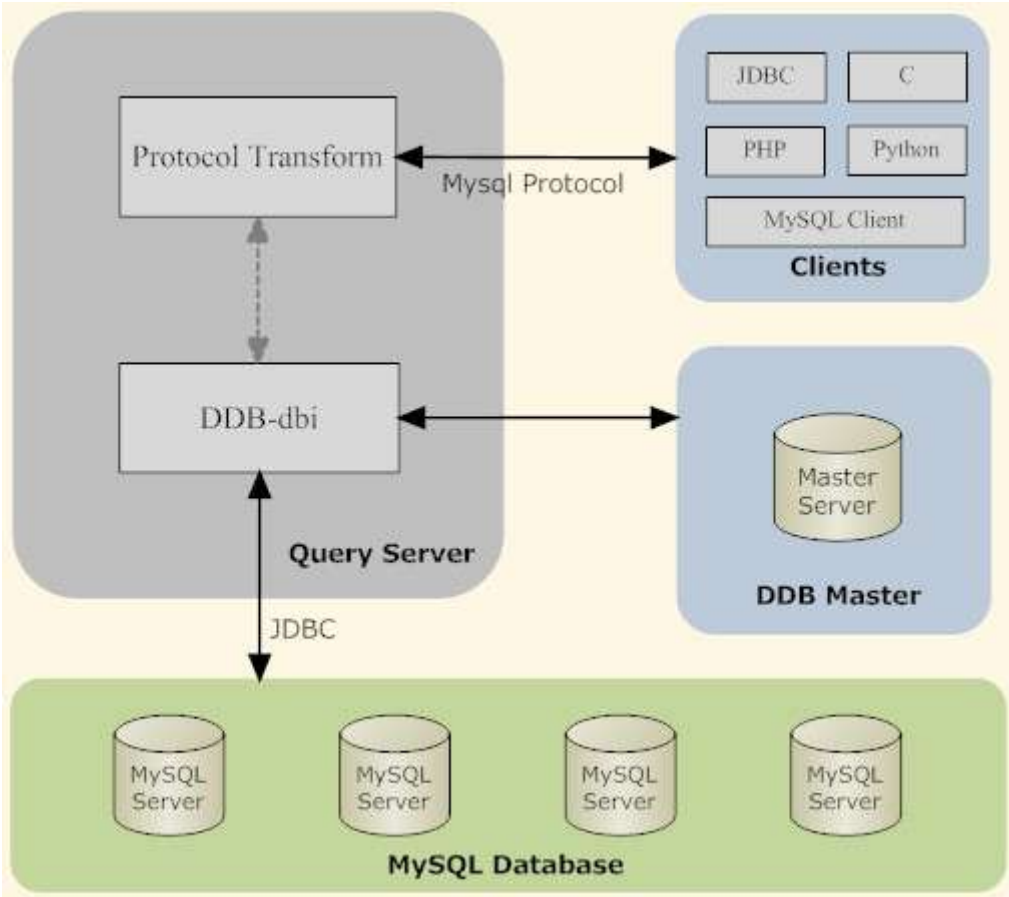
务。

DDB 对用户配额进行限制按如下流程进行：

- **Master** 定期执行计划任务，收集数据库节点上每张表大小并进行汇总。
- 对 DDB 中间件上执行的每一条 SQL 语句，进行 **quota** 配额判断，决定该语句是否允许被执行。
 - 如果是 **DIRECT FORWARD** 语句，忽略。
 - 判断语句是否有 **limit-offset** 条件，如果有，则判断 **limit** 值+**offset** 值是否小于配额，成立则允许执行。
 - 判断语句所涉及的表的记录数，如果小于配额，则允许执行。
 - 如果语句中不包含 **WHERE** 条件，检查查询的列
 - ◆ 如果查询列都是 **min**、**max** 函数，并且函数的列存在索引，则允许执行
 - ◆ 其他情况都说明用户配额超出，抛出异常
 - 包含了 **WHERE** 条件，如果是 **AND** 条件，需要有一个条件满足下面的判断；如果是 **OR** 条件，需要全部满足下面的判断。
 - ◆ 如果是列、聚集函数、标量函数、运算表达式，则判断表达式是否是索引，其他任何情况都认为符合用户配额限制。

10 查询服务器

查询服务器是独立于 DDB 管理服务器（**master**）和底层数据库节点服务器的一台单独服务器。是否搭建一台查询服务器并不是使用 DDB 必须的步骤，DDB 原来的设计初衷就是能够让客户端直接访问底层数据库节点进行操作。而建立查询服务器后，将实现一个三层架构，客户端可以访问查询服务器，查询服务器负责操作各个数据库节点。采用查询服务器后的三层架构示意图如下：



查询服务器三层架构示意图

采用基于查询服务器的三层架构，有如下好处：

把 DDB 的适用范围扩展到非 JAVA 领域。目前的 DDB 代码采用 JAVA 开发，dbi 端由于需要内嵌到客户端程序中，这客观上强制要求 DDB 客户端程序必须使用 JAVA 开发。采用三层架构后，客户端与查询服务器之间，采用标准 MySQL 服务器通信协议进行交互。对于 C 编程语言，可以采用 MySQL-C API；PHP 编程语言，可以采用 PDO 或 MySQLi；Python 语言，可以采用 MySQLdb。

重用现有的 MySQL 客户端工具。用户可以根据喜好使用自己以前熟悉的客户端工具，如 MySQL 标准命令行客户端工具或 PhpMyAdmin 等第三方工具（未经测试）。

共享资源。一些数据库资源，例如 MySQL 服务器 TCP 连接或 DDB 执行计划等，创建时会消耗额外资源。把他们集成到某几台专门的查询服务器中，可以有效实现资源共享。尤其是与 MySQL 服务器之间的网络链接资源，每台 MySQL 服务器都有其固定最大值，维护过多客户端连接会导致 MySQL 服务器性能下降，采用查询服务器后能有效解决这个问题，使 DDB 实现真正线性无限制扩展。

10.1 服务器通信协议

查询服务器与客户端之间的交互，采用 MySQL 标准通信协议，具体描述参见：http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol

查询服务器目前已支持的通信命令见下表：

命令名称	说明	路径方向
Handshake	握手命令，连接刚建立时由查询服务器发往客户端	Server -> Client

Authentication	认证命令，客户端把用户名密码等信息发送到服务器用于认证	Client -> Server
ChangeUser	客户端发送新用户名、密码、默认数据库名称到服务器，用于修改当前值	Client -> Server
InitDB	用户设置默认数据库名称	Client -> Server
Ping	实现 ping 功能	Client -> Server
Query	执行 SQL 语句	Client -> Server
Quit	客户端退出命令	Client -> Server
StmtClose	关闭一个服务器上缓存的 Preparestatement 句柄	Client -> Server
StmtExecute	执行 Preparestatement 语句	Client -> Server
StmtFetch	从服务器获取一批上次 Preparestatement 执行后的结果集数据	Client -> Server
StmtPrepare	在服务器上创建一个 Preparestatement 句柄	Client -> Server
StmtReset	重置 Preparestatement 句柄，常用于清空 blob 数据	Client -> Server
StmtSendLongData	发送 Preparestatement 中的 blob 参数数据	Client -> Server
Error	执行错误返回包	Server -> Client
OK	执行成功返回包	Server -> Client
ResultSet	返回查询语句结果集	Server -> Client
PrepareOK	服务器上创建 Preparestatement 句柄成功	Server -> Client

10.2 服务器内部实现

整个查询服务器，最主要的模块有两个：通信协议转换和 SQL 语句处理。

通信协议处理转换模块把客户端发送过来的数据包进行解码之后转交给 SQL 语句处理模块进行处理，然后把查询结果集等返回信息编码后发送给客户端。

SQL 语句处理模块其实代码相对简单，我们可以把 SQL 语句处理看作在一个标准 DDB 客户端中进行。需要处理的 SQL 语句被提交给 DDB 的 dbi 模块，dbi 模块负责 SQL 语法解析，然后生成执行计划，并在底层数据库上执行 SQL 语句，最后进行可能需要的结果集合并操作。所有的复杂操作其实都是在 dbi 中实现，对上层保持透明。

对于 Preparestatement 来说，实现稍微复杂一些。当客户端需要打开一个 Preparestatement 时，查询服务器会创建一个 DDB 的 preparestatement 句柄，由 dbi 负责解析 SQL 语句，然后返回给客户端语句中的参数个数等信息。当客户端发送参数值时，查询服务器会把参数值缓存起来，在真正执行时才发送给 dbi。从创建 preparestatement 开始，查询服务器就缓存 DDB 的 preparestatement 句柄，直到客户端显式关闭 preparestatement 或者客户端连接断开。

查询服务器采用异步通信模式，基于 Mina 开发。<http://mina.apache.org/>。采用异步通信架构后，一台查询服务器可以同时为多台客户端应用程序提供服务，并发性能大为提高。具体测试数据见性能测试报告的相关章节。

对一张大数据量的表进行遍历操作时，需要把所有表数据获取到客户端。为了避免此过程导致查询服务器内存溢出，专门提供了流模式进行数据发送。开启流模式后，查询服务器内部做了两个操作：打开 ddb 的流模式获取数据开关，通过调用 setfetchsize 函数，ddb 将分批从底层数据库节点获取数据；以流模式来发送数据到客户端，当第一批数据被成功发送到服务器

操作系统缓存后，才开始发送第二批数据。

一般的 MySQL 服务器，客户端程序在运行 SQL 语句失败后，都能够拿到对应的错误号（ErrorNo）和 SQL 状态号（SQLState）。目前查询服务器简单起见，所有的错误号都是 1，SQL 状态号都是 10000。MySQL 的错误号和状态号，虽然比起 Oracle 稍微简洁一点，但也算是千奇百怪无所不有，估计等以后有空了或用户实在受不了提出需求了再慢慢的完善这块内容吧。

在连接 MySQL 数据库后，服务器会返回其版本号，一些客户端工具如 MySQL 命令行客户端在每次连接之后都会显示服务器的版本号。相应的，如果连接了查询服务器，返回的版本信息将如以下形式：5.1.47-DDBQS-DDB-4.1.1-68419。其中 5.1.47 仿照目前 MySQL 服务器的最新版本号，目前没有提供外部接口供修改。一些客户端程序包如 JDBC 之类，会根据这个版本号做判断，不同版本号在代码实现上有些细微差别。目前经过测试，返回 5.1.47 是没有问题的。DDB-4.1.1-68419 这串数字，从 db.jar 中提取，其实就是 DDB 代码的版本号。每次发布新版本时，都会进行升级。

10.3 服务器返回参数列表

根据 MySQL 的通信协议，在每次连接服务器后，在握手包中服务器都会返回一串标识符，用于说明当前服务器能够提供的功能。下面把查询服务器所返回的标识符进行详细说明。其实在通常情况下，不需要太关注这块内容，因为查询服务器目前的行为已经能够满足大多数的客户端需求。对于一些客户端程序的特殊功能，可能需要查找下表来确定查询服务器是否能够提供支持。

参数	是否支持	说明
CLIENT_LONG_PASSWORD	是	new more secure passwords
CLIENT_FOUND_ROWS	否	Found instead of affected rows
CLIENT_LONG_FLAG	是	Get all column flags
CLIENT_CONNECT_WITH_DB	是	One can specify db on connect。查询服务器要求必须在连接时指定数据库名称
CLIENT_NO_SCHEMA	否	Don't allow database.table.column。允许客户端在 SQL 语句中指定数据库名称，这个名称即 DDB 名称，用于多 DDB 查询
CLIENT_COMPRESS	否	Can use compression protocol
CLIENT_ODBC	否	Odbc client
CLIENT_LOCAL_FILES	否	Can use LOAD DATA LOCAL
CLIENT_IGNORE_SPACE	否	Ignore spaces before '('
CLIENT_PROTOCOL_41	是	New 4.1 protocol
CLIENT_INTERACTIVE	否	This is an interactive client
CLIENT_SSL	否	Switch to SSL after handshake
CLIENT_IGNORE_SIGPIPE	否	IGNORE sigpipes
CLIENT_TRANSACTIONS	是	Client knows about transactions
CLIENT_RESERVED	否	Old flag for 4.1 protocol
CLIENT_SECURE_CONNECTION	是	New 4.1 authentication

CLIENT_MULTI_STATEMENTS	否	multi-stmt support
CLIENT_MULTI_RESULTS	否	multi-results

10.4 服务器安装配置说明

运行查询服务器非常简单，直接运行启动脚本 `qs.sh/qs.bat` 就可以。在运行之前，需要关注两个服务器配置文件。

Master 端增加了对查询服务器的监控报警，部署查询服务器时，需要先在 **DDB** 中添加类型为 **MAN** 的查询服务器监控用户。用户名限定为“**qs**”，并需要在“允许访问的中间件”地址列表中添加 **Master** 的地址。

10.4.1 log4j.properties

没什么好多讲，**log4j** 的配置文件而已。需要修改的项目只是 `log4j.category.QueryServer` 的输出级别。如果设定为 **DEBUG** 级别，所有来往的通信包内容都会被打印，这对调试非常有帮助，只是开启 **DEBUG** 后输出内容会非常庞大，因此建议在线上部署时把输出级别调为 **INFO**。设定为 **INFO** 级别后，输出的内容只是服务器启动或关闭的信息，另外，如果服务器运行过程中产生了非预期异常，也会打印出来。对于整个服务器运行过程中产生的异常错误，我们大致把它们分为两类：一种是正常错误，例如 **insert** 时的 **duplicate key** 异常，所有的 **SQLException** 异常只是作为错误返回包返回给客户端，错误内容不会输出到服务器的日志中。另一种是非预期异常，比方说运行中的空指针异常，这些错误除了以错误返回包返回给客户端之外，还会记录在服务器日志中。

10.4.2 QSServerConf.xml

这是服务器参数配置文件，简单的参数配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<qs_conf>
  <server>
    <!--查询服务器绑定的 IP 地址-->
    <ip>127.0.0.1</ip>
    <!--查询服务器绑定的端口-->
    <port>6666</port>
    <!--BLOB 参数的最大长度-->
    <blob_length>209715200</blob_length> <!-- 200M -->
    <!--以 preparestatement 方式 fetch 查询结果时，每次 fetch 最记录数-->
    <max_pstmt_fetch_size>5000</max_pstmt_fetch_size>
    <!--以流方式发送查询结果，可避免服务器内存溢出，true/false-->
    <use_stream_fetch>true</use_stream_fetch>
    <!--流方式每次发送记录数-->
    <stream_fetch_size>100</stream_fetch_size>
    <ddb> (1)
      <name>ddb_demo</name>
      <url>127.0.0.1:8888</url>
    </ddb>
    <ddb>
      <name>ddb_demo2</name>
      <url>127.0.0.1:8889</url>
    </ddb>
  </server>
  <support> (2)
```



```

<!--type: string/regex-->
<!--behavior:
execute: 执行 sql 语句
refuse: 拒绝执行
ignore: 忽略, 不抛错不执行
query: 查询 sql 语句并返回结果
query_dbn_cache: 随机获取一个底层数据库节点进行查询, 并缓存查询结果以备以后使用
insert: 插入数据并返回可能的唯一 ID 值
-->
<sql type="string" behavior="query_dbn_cache">show session
variables</sql>
<sql type="string" behavior="query_dbn_cache">show collation</sql>
<sql type="string" behavior="ignore">set character_set_results =
NULL</sql>
<sql type="string" behavior="ignore">set
sql_mode='STRICT_TRANS_TABLES'</sql>
<sql type="regex" behavior="query">(?i)^select .*</sql>
<sql type="regex" behavior="execute">(?i)^update .*</sql>
<sql type="regex" behavior="execute">(?i)^delete .*</sql>
<sql type="regex" behavior="execute">(?i)^call .*</sql>
<sql type="regex" behavior="insert">(?i)^insert .*</sql>
<sql type="regex" behavior="insert">(?i)^replace .*</sql>
<sql type="regex" behavior="ignore">(?i)set\s+names\s+.*</sql>
<!--这一项必须放在最后-->
<sql type="regex" behavior="refuse">.*</sql>
</support>
</qs_conf>

```

额外说明

(1): **ddb** 配置项。每组配置项有两个参数: **ddb** 名称和连接 **url**, 可同时配置多组 **ddb**。对于一个查询服务器来说, 不可能能够动态获知周边有多少个 **ddb** 可以连接。因此, 必须在配置文件中配置, 说明每个 **ddb** 的名称和 **master** 连接 **url**。为了统一起见, **ddb** 名称最好和该 **ddb** 的真实名称一致, 客户端在连接查询服务器时, 会给出需要连接的 **ddb** 名称, 查询服务器将根据这个名称在 **ddb** 列表中进行查找匹配, 获取其真实连接 **url** 之后再连接到 **master** 服务器进行用户认证。

(2): **support** 配置项。这个配置项将直接决定查询服务器的行为。客户端提交给查询服务器的 **SQL** 语句, 究竟应该如何执行, 都需要在这里配置。一般来说, 这组配置项不用改动, 如果在运行过程中出现了没法解决的问题, 可以回过头来尝试一下修改这个配置项。以上面的参数配置文件为例来一步步说明:

如果用户提交了 **show session variables** 命令, 则查询服务器将随机选取一个底层节点来执行这条命令, 然后把执行结果返回给客户端。为了提高性能, 这个执行结果将被缓存, 如果以后有同样的命令, 则直接返回结果给客户端。之所以要直接选取底层数据库节点, 是因为目前 **dbi** 还没有支持所有的 **SQL** 语句, 像 **show session variables** 之类的命令, **dbi** 是无法识别的, 因此只能直接提交给底层节点。由于 **query_dbn_cache** 是随机选取一个底层节点, 不做任何 **ddb** 层面处理, 因此配置时需要格外小心, 不要把一些正常的如 **select**、**insert**、**delete**、**update** 之类的句子放到里面。

如果用户提交了 **show collation** 命令, 则查询服务器行为同上。

如果用户提交了 **set character_set_results = NULL** 命令, 则查询服务器直接忽略。

如果用户提交了 **set sql_mode='STRICT_TRANS_TABLES'** 命令, 则查询服务器直接忽略。

如果用户提交了以 **select** 开头的 **SQL** 语句, 则查询服务器把它提交给 **dbi** 进行查询, 并把查询结果返回给客户端。

如果用户提交了以 **update**、**delete**、**call** 开头的 **SQL** 语句, 则查询服务器把它提交给 **dbi** 执行。

如果用户提交了以 **insert**、**replace** 开头的 SQL 语句，则查询服务器把它提交给 **dbi** 执行，并把新生成的唯一 **ID** 返回给客户端。

如果用户提交了 **set names** 命令，则查询服务器直接忽略。

除此之外的所有其他 SQL 命令，查询服务器均拒绝执行，并返回错误包给客户端。

10.5 使用 Isql 命令

查询服务器支持 DDB 中交互式 SQL 工具常用的命令，用户可以通过通用的 MySQL Client 等终端工具连接查询服务器并执行，具体支持的命令如下表，详细的命令说明可以查阅[交互式 SQL 工具](#)一章。

命令	语法	描述
show commands	SHOW COMMANDS	Show all commands.
show tables	SHOW TABLES	Show all tables in the system.
show tables for policy	SHOW TABLES FOR POLICY policy_name	Show tables for the specified policy.
show tables for dbn	SHOW TABLES FOR DBN name	Show tables which use the specified database node.
show views	SHOW VIEWS	Show all views in the system.
show create table	SHOW CREATE TABLE table	Show definition of a table
show index for	SHOW INDEX FOR tbl_name	Show indexes for one table.
show dbns	SHOW DBNS	Show all database nodes in the system.
show dbns for policy	SHOW DBNS FOR POLICY policy_name	Show dbns for the specified policy.
show dbns for table	SHOW DBNS FOR TABLE table_name	Show dbns which have the specified table.
show policies	SHOW POLICIES	Show all policies in the system.
show ddb	SHOW DDBS	Show ddb informations in system.
show current ddb	SHOW CURRENT DDB	Show current ddb information.
show triggers	SHOW TRIGGERS [LIKE tbl_name] [FOR trigger_name]	Show all triggers or specified triggers.
show procedures	SHOW PROCEDURES [FOR sp_name]	Show all procedures or specified procedures.
desc	DESC name	Describe a table.
desc policy	DESC POLICY name	Describe a policy.

10.6 客户端使用说明

10.6.1 连接查询服务器

理论上所有的已有 MySQL 客户端工具或程序，都能无缝的连接到新查询服务器上。与之前不同的是，在连接时必须同时给出数据库名称。其实此处的数据库名称即 **ddb** 名称，因为查询服务器必须根据 **ddb** 名称查找到对应连接 **ddb** 的 **url**，然后在 **master** 服务器上认证客户端给出的用户名密码，否则无法在连接时做用户名密码认证操作。

MySQL 命令行客户端连接查询服务器的命令如下：


```
mysql -h 127.0.0.1<查询服务器监听 ip 地址> -P 6666<查询服务器监听端口> -utest<DDB 用户名> -ptest<DDB 密码> ddb_demo<DDB 数据库名称>
```

JDBC 方式连接查询服务器的连接 URL 如下：

```
jdbc:mysql://127.0.0.1:6666/ddb_demo?user=test&password=test
```

客户端通过查询服务器进行 SQL 语句操作时，支持同时连接多个 DDB。DDB 名称之间使用 '\$\$' 进行连接，但每个 DDB 的认证用户名密码必须相同。

MySQL 命令行客户端连接多 DDB 的命令如下：

```
mysql -h 127.0.0.1 -P 6666 -utest -ptest ddb_demo$$ddb_demo2
```

JDBC 方式连接查询服务器的连接 URL 如下：

```
jdbc:mysql://127.0.0.1:6666/ddb_demo$$ddb_demo2?user=test&password=test
```

10.6.2 多字符集问题

由于多了查询服务器这一层转换，所以字符集问题就比原来直连数据库节点来的更复杂一点。在讨论下面的问题之前，必须先牢记解决字符集这个头痛问题最简单的办法：把底层数据库节点字符集、查询服务器字符集、客户端字符集统统设为统一字符集。

客户端连接 MySQL 服务器，有三种字符集设置方式：`character_set_client`、`character_set_connection`、`character_set_results`，或者统一用 `set names` 命令来设置。这些命令的实现方式，都是把客户端当前字符集名称发往 MySQL 服务器，由服务器负责进行编码转换。

仔细查看前面给出的查询服务器配置模版，可以发现我们把 `set names` 和 `set character_set` 命令都设置为 `ignore` 方式，即查询服务器接收到该命令时，直接将其忽略。这是由于 ddb 内部使用了比较复杂的底层连接池缓冲机制，无法为每个查询服务器客户端连接单独设定字符集。因此，在使用查询服务器的三层架构中，字符集设置规则如下：客户端程序使用的字符集与查询服务器使用的字符集必须相同，而底层的 MySQL 服务器可以使用任意编码方式。

10.6.3 SQL 语句接口

1. 支持大部分常用 DML 命令。

```
SELECT ...
INSERT ...
UPDATE ...
DELETE ...
REPLACE ...
CALL ...
```

2. 所有的 DDL 语句均不支持。

3. 支持事务。

```
COMMIT
ROLLBACK
SET AUTOCOMMIT=true/false
```

4. 支持 `preparestatement`。需要注意的是，在提交预处理的 SQL 语句后(`prepare`)获得的参数描述信息(`param_metadata`)或者结果集字段描述信息(`resultset_metadata`)都是不正确的或者说是无效的。有效的结果集字段信息必须是在提交执行后(`execute`)得到的 `result_set` 中读取。
5. 支持游标。游标分为两种：传统意义的游标，JDBC 下通过设置 `setFetchSize=Integer.MIN` 打开，python 或 C 语言中通过 `useResultSet` 函数打开。这种方式的游标不需要服务器提供额外支持，只是客户端的行为；另一种是

Preparestatement 下的游标，JDBC 下通过 `setFetchSize` 接口并在连接 `url` 中设置 `useCursorFetch` 打开，此种类型游标不需要客户端取完所有查询结果，使用起来更方便。两种游标在查询服务器中均已支持。

6. 支持 SQL 语句中包含 `hint`。DDB 的 SQL 语法中有一套独特 `hint` 语法，用于做负载均衡或 `direct forward`，这在查询服务器中已支持。特别说明一下，在 `mysql` 客户端下使用 `hint` 无效，这是因为客户端自动把 `hint` 部分截掉了，没有发给服务器，导致服务器无法获知 `hint` 信息。
7. 能够获取已分配自增长字段 `id`。按照 MySQL 通信协议，`insert` 命令成功后，在 `ok` 返回包中会包含已分配的自增长字段 `id`。查询服务器遵照了 MySQL 协议，把 DDB 所分配的 `id` 返回给客户端，客户端可以通过自己编程语言的相应接口获取。JDBC 中的获取接口为：`Statement.getGeneratedKeys()`。在 MySQL 客户端下，只能通过 `select last_insert_id()` 语句获取自增长 `id`，这个语句在 DDB 中不支持，因此 MySQL 客户端无法获取自增长 `id`。

10.6.4 Python 编程语言接口支持度说明

Python 语言已经过测试能够正常连接查询服务器，其中语言接口测试覆盖度说明如下：

包、模块或类	方法	测试是否覆盖
MySQLdb	<code>connect()</code>	是
MySQLdb.connections	<code>affected_rows()</code>	是
	<code>autocommit()</code>	是
	<code>begin()</code>	是
	<code>change_user()</code>	否
	<code>character_set_name()</code>	是
	<code>close()</code>	是
	<code>commit()</code>	是
	<code>cursor()</code>	是
	<code>dump_debug_info()</code>	否
	<code>errno()</code>	是
	<code>error()</code>	是
	<code>escape()</code>	否
	<code>escape_string()</code>	否
	<code>field_count()</code>	是
	<code>get_character_set_info()</code>	是
	<code>get_host_info()</code>	是
	<code>get_proto_info()</code>	是
	<code>get_server_info()</code>	是
	<code>info()</code>	是
	<code>insert_id()</code>	是
	<code>kill()</code>	否
	<code>next_result()</code>	否
	<code>ping()</code>	否
	<code>query()</code>	是
	<code>rollback()</code>	是
	<code>select_db()</code>	是
	<code>set_character_set()</code>	是
	<code>set_server_option()</code>	否
	<code>set_sql_mode()</code>	否
	<code>show_warnings()</code>	否

	shutdown()	否
	sqlstate()	是
	stat()	否
	store_result()	是
	thread_id()	是
	use_result()	是
	warning_count()	是
_mysql.result	data_seek()	是
	fetch_row()	是
	field_flags()	是
	num_fields()	是
	num_rows()	是
	row_seek()	是
	row_tell()	是
MySQLdb.cursors	callproc()	否
	close()	是
	execute()	是
	executemany()	否
	fetchall()	是
	fetchmany()	是
	fetchone()	是
	nextset()	否
	scroll()	是
	next()	是

共 55 个接口，其中 15 个未测试，40 个已测试

10.6.5 PHP 语言接口支持度说明

PHP 语言已经过测试能够正常连接查询服务器，其中语言接口测试覆盖度说明如下：

类名	方法	测试是否覆盖
PDO	beginTransaction	是
	commit	是
	construct	是
	errorCode	是
	errorInfo	是
	exec	是
	getAttribute	是
	getAvailableDrivers	是
	lastInsertId	是
	prepare	是
	query	是
	quote	是
	rollBack	是
	setAttribute	是
PDOStatement	bindColumn	是

	bindParam	是
	bindValue	是
	closeCursor	是
	columnCount	是
	debugDumpParams	是
	errorCode	是
	errorInfo	是
	execute	是
	fetch	是
	fetchAll	是
	fetchColumn	是
	fetchObject	是
	getAttribute	否
	getColumnMeta	是
	nextRowset	否
	rowCount	是
	setAttribute	否
	setFetchMode	是

PDO 共 33 个接口，其中 3 个未测试覆盖，30 个已测试覆盖。

类名	方法	是否覆盖
MySQLi	affected_rows	是
	autocommit	是
	change_user	否
	character_set_name	是
	client_info	是
	client_version	是
	close	是
	commit	是
	connect_errno	是
	connect_error	是
	construct	是
	debug	否
	dump_debug_info	否
	errno	是
	error	是
	field_count	是
	get_charset	是
	get_client_info	是
	get_connection_stats	否
	host_info	是
	protocol_version	是

	server_info	是
	server_version	否
	get_warnings	是
	info	否
	init	是
	insert_id	是
	kill	否
	more_results	否
	multi_query	否
	next_result	否
	options	否
	ping	是
	poll	否
	prepare	是
	query	是
	real_connect	是
	real_escape_string	否
	real_query	否
	reap_async_query	否
	rollback	是
	select_db	是
	set_charset	否
	set_local_infile_default	否
	set_local_infile_handler	否
	sqlstate	否
	ssl_set	否
	stat	否
	stmt_init	否
	store_result	是
	thread_id	否
	thread_safe	否
	use_result	否
	warning_count	否
MySQLi_STMT	affected_rows	是
	attr_get	否
	attr_set	否
	bind_param	是
	bind_result	是
	close	是
	data_seek	是
	errno	是
	error	是
	execute	是

	fetch	是
	field_count	是
	free_result	是
	get_warnings	否
	insert_id	否
	num_rows	否
	param_count	否
	prepare	否
	reset	是
	result_metadata	是
	send_long_data	是
	sqlstate	否
	store_result	是
MySQLi Result	current_field	是
	data_seek	是
	fetch_all	是
	fetch_array	是
	fetch_assoc	是
	fetch_field_direct	是
	fetch_field	是
	fetch_fields	是
	fetch_object	是
	fetch_row	是
	field_count	是
	field_seek	是
	free	是
	lengths	否
	num_rows	是

MySQLi 共 92 个接口，其中 35 个未测试覆盖，57 个已测试覆盖。

10.6.6 对二进制数据类型的支持

在 MySQL 的协议中，客户端将二进制类型的数据（`tinyblob`，`blob`，`mediumblob`，`longblob`）发送到服务器（`insert`，`update`）有多种方式，不同客户端驱动在实现的时候采用的方式也不尽相同。

查询服务器支持其中两种方式：客户端使用服务器端（**server side**）预处理直接发送二进制数据；或根据 SQL 语言标准将二进制内容转换为字符串，将二进制值作为字符串发送到服务器。通过查询服务器读取二进制类型数据，则是完全兼容，不需要特殊处理。

服务器端预处理是指先将带占位符的 `sql` 语句发送到服务器进行解析，然后发送不同参数值到服务器执行，以减少服务器端对 `sql` 语句的重复解析。不同客户端驱动对服务器端预处理方式的支持会有较大不同，有些还是伪预处理。尤其需要注意的是以下两种客户端驱动：

Python MySQLdb：不支持服务器端预处理。支持参数化处理，底层的实现是在执行的将参

数填入 **sql** 语句中，直接发送完整的 **sql** 命令到服务器端，减少开发人员手工进行 **sql** 语句组装。

Java Connector/J：支持服务器端预处理，但必须在连接 **url** 中指定连接参数 **useServerPrepStmts = true**，否则 **Connection.prepareStatement()** 得到的并不是真正的服务器端预处理，也是简单的内部 **sql** 语句组装。

将二进制值转换为字符串是指将二进制内容的每 1 个字节转换为 2 个 16 进制的字符，并在字符串单引号前加上转换标识字母 **x** 或 **X**。这是标准 **SQL** 语言支持的二进制处理方式。由于转换标识的存在，限制了这种方式只适用于手工直接进行字符串组装，而不能使用伪预处理。**Java** 语言下简单实现如下：

```
/** 转换函数实现 */
public class HexUtil {
    private final static byte[] HEX DIGITS = new byte[] { (byte) '0',
        (byte) '1', (byte) '2', (byte) '3', (byte) '4', (byte) '5',
        (byte) '6', (byte) '7', (byte) '8', (byte) '9', (byte) 'A',
        (byte) 'B', (byte) 'C', (byte) 'D', (byte) 'E', (byte) 'F' };

    public static String toHexString(byte[] data) {
        byte[] tmp = new byte[data.length * 2];
        for(int i = 0; i < data.length, i++) {
            tmp[i*2] = HEX DIGITS[(data[i] & 0xff) / 16];
            tmp[i*2 + 1] = HEX DIGITS[(data[i] & 0xff) % 16];
        }
        return new String(tmp);
    }
}

/** 使用示例 */
byte[] content = getContent();
String strContent = HexUtil.toHexString(content);
Statement st = connection.createStatement();
//必须记得加上 x/x
st.execute("insert into blob(id, content) values(" + id + ", x'" + strContent
+ "')");
```

11 数据库维护高级主题

11.1 在线数据迁移

对数据库管理人员来说，日常维护 **DDB** 时，碰到的最复杂艰巨的任务可能就是数据迁移。由于数据库 **SQL** 语言本身的语义丰富性，并且应用在实际生产的高并发访问环境中，导致使用简单的分批数据迁移变的问题多多。**DDB** 目前提供的数据迁移方式，基于 **MySQL** 数据库提供的主从复制，能够保证只在非常短的一段时间内对线上操作进行阻塞，且整个迁移过程，对数据库的性能影响微乎其微。但不管怎么说，在线迁移仍旧是一件复杂的工作，建议数据库管理人员在实际产品环境操作之前，先认真通读本章，了解迁移的内部原理。

11.1.1 数据迁移流程

DDB 中提供的在线迁移，总的来说分为两种模式。

- 一对多迁移。把一个源数据库节点中的数据迁移到多个其他目标数据库中，迁移过程中源数据库节点中的数据将按照管理员指定的策略进行分裂。这种模式其实也包括了

一对一迁移的情况，即把一个源数据库中的某些策略或一个策略中的某些桶的数据迁移到其他节点上。

- 多对一迁移。把多个源数据库节点中的数据迁移到一个目标数据库中，迁移过程中源数据库节点中的数据将全部合并到新节点中。

上面所说的源节点，必须是 DDB 系统目前正在使用的节点，迁移完成后，源节点中的数据可被完全迁走，也可能仍旧保留了一部分数据。如果源节点中仍旧保留有数据，则在迁移完成后，将继续在 DDB 中提供服务，否则的话，迁移完成后源节点将从均衡策略的桶对应关系中剔除。目标节点数据库，必须是一个全新的节点，迁移之前不应存在任何有用的其他数据。DDB 系统目前不支持往一个正在提供服务的节点上迁移数据。

下面详细说明两种迁移模式的执行流程。

一对多迁移

假设迁移的源节点是 A，上面存储了 b1, b2, ..., bn 等 n 个桶，现要将 bk+1, bk+2, ..., bn 等(n - k)个桶迁移到节点 B，把 b1, b2, ..., bk 等 k 个桶迁移到 C 节点。

- 开始准备阶段操作。
- 将 A 节点中的 mysql 数据库数据使用 InnoDB backup 工具备份出来，拷贝到节点 B、节点 C 上。
- 启动 mysql，并开启 mysql 复制，等待同步完成。
- 记录每个表的当前分配 id。并通知所有 dbi，强行从最新的 id 之后开始分配。
- 停止 mysql 复制
- 节点 B 最终存放的桶数据是 bk+1, bk+2, ..., bn 等(n - k)个，节点 C 存放的数据是 b1, b2, ..., bk 等 k 个桶。因此先把节点 B 和节点 C 上需要保留的数据导出到 MyIsam 临时表，然后 drop 掉原表。
- 在节点 B 和 C 上重建原表，此时过滤所有触发器建表语句，触发器暂时不加。
- 在节点 B 和 C 上把临时表数据重新导回。这个过程可以使得 innodb 数据文件瘦身（前提是使用 innodb 多表空间）。
- 在节点 B 和 C 上恢复所有触发器。
- 开启节点 B 和 C 到节点 A 的 mysql 复制，等待同步完成。
- 开始线上阶段操作。
- 在 A 节点的 mysql 上执行 set readonly=true。
- 停止 B、C 到 A 的 replication。
- 通知所有中间件，把对 bk+1, bk+2, ..., bn 操作指向到 B，对 b1, b2, ..., bk 的操作指向 C。
- 系统访问 C 时，若查询语句没有指定均衡字段等值条件，则加上条件 bucketno in (b1, b2, ..., bk)，访问 B 时若没有指定均衡字段等值条件，则加上条件 bucketno in (bk+1, bk+2, ..., bn)，其中若均衡字段是整型 bucketno 也可用"abs(均衡字段%桶数)"代替。
- 在 A 节点的 mysql 上执行 set readonly=false。
- 根据前面统计的表已分配 id，批量删除 replication 同步时引入的少量脏数据。
- 中间件恢复到正常模式，不自动加 bucketno 条件。
- 清空 B、C 的所有复制相关参数，并置 server-id 为 1，迁移完成。

多对一迁移

A 为线上节点（包含 1,2,3 共三个桶），B 为线上节点（包含 4,5,6 共三个桶），C 为空闲节点，现将两个节点的数据迁移到 C 节点。

- 开始准备阶段操作
- 记录所有表的已分配最大 ID，以便以后删除脏数据

- c) 通知所有的 DDB 客户端，对源节点涉及到迁移的相关数据的更新删除操作，增加桶号的额外限制条件，防止经过 MySQL 复制后修改了从节点上不属于自身的其他数据
- d) 多对一复制，考虑到源节点的数据量比较大，全部拷贝到 **slave** 节点之后再整理数据时有可能导致从节点硬盘空间不足，所以采用按照源节点依次导入数据的方式建立从节点
- e) 在 C 节点上首先建立 A 节点的 **Slave**，称为 C1
- f) 把 C1 的数据导出到 **MyIsam** 临时表，然后导入 1、2、3 号桶的数据到 C1 中，最后清空 **MyIsam** 临时表。
- g) 同理，建立 B 节点的 **Slave**--C2，把 C2 的数据导出到临时表，然后导入 4、5、6 号桶的数据到 C1。这个过程中，C2 数据库的文件目录应位于一个其他的位置，最后导入完毕后，关闭 C2 数据库并删掉 C2 的文件夹，但要保存 **ibbackup_binlog_info** 文件，以备以后建立复制关系
- h) 根据 **ibbackup_binlog_info** 启动 C1 到 A 的复制，待同步后停止复制
- i) 根据 **ibbackup_binlog_info** 启动 C1 到 B 的复制，待同步后停止复制
- j) 重复步骤 3 和 4，直到 30 秒钟（这个时间可以配置）内 C1 分别完成到 A 和 B 的同步各一次为止。这个过程涉及到在多个源服务器之间来回切换，切换时应先停止复制，然后记录 **Exec_Master_Log_Pos** 位置，作为下一次该源节点 **master** 日志读取的起始位置
- k) 至此，在线迁移的 **prepare** 阶段完成，等待 DBA 发出继续操作的命令
- l) 在 **prepare** 完成到正式开始节点切换这段时间内，**slave** 节点还是要不停地做 **master** 节点间的来回切换，切换时间间隔为 30 秒钟切换一次
- m) 开始线上阶段操作
- n) 在两个源节点上执行 **set global readonly=true**，拒绝各种写操作
- o) 确认从节点对两个源节点都已复制完全同步
- p) 停止 C 节点的 **MySQL** 复制
- q) 修改 **Master** 和 **DBI** 上的桶映射关系，并将 **DBI** 上涉及 A,B,C 的相关语句（包括 **select**）加上 **bucketno** 条件
- r) 在两个源节点 A、B 上执行 **set readonly=false**，开启系统对源节点的写操作
- s) 批量删除 A、B、C 上的脏数据
- t) **DBI** 上涉及 A,B,C 的语句去掉 **bucketno** 条件
- u) 清空 C 的所有复制相关参数，并置 **server-id** 为 1，整个迁移过程完成

删除过期数据

上面讨论的迁移情况，都是针对源节点的数据被完全迁出，迁移完成之后源节点不再被使用。实际操作中，还有一种情况是只是迁移出部分的数据，或者是一部分的表，或者是表中一部分的桶数据。这时就会涉及到源节点上执行删除已被迁出的过期数据的问题。

在迁移过程中，删除过期数据被安排在删除脏数据的同时进行。需要提醒的是，依据被删除过期数据的规模大小，删除操作可能会执行较长的时间。在这个过程中，不会阻塞线上的应用请求操作，但是会影响到底层数据库的性能。**DDB** 采用已有的“批量删除”接口来实现删除过期数据，每次删除一定数目的记录后，会暂停一段时间，这样做的好处有：

- 减轻删除操作对数据库的负载压力
- 把整个删除过程分批进行，可以避免锁住所有的过期数据，造成不必要的线上操作锁等待超时
- 可以减少删除过程产生的数据库节点回滚段额外磁盘消耗及清空回滚段时间
- 可以有效减少底层数据库节点“块清除”带来的负面影响。

在开发阶段，已经对删除过期数据做过仿真模拟测试，只要参数配置合理，对线上的影响并不大。但是，迁移完成后源节点数据文件并不能被马上“瘦身”，如果想要马上获得空余磁盘空间，依靠这种方法是不行的。

数据库管理人员需要知道，删除过期数据这个操作并不是必须的，可以依靠迁移策略来避免，举个例子来说明。假设 A 节点有桶数据 1、2、3，现在需要把 1、2 的数据迁往 B，自身保

留 3 号桶的数据。一种做法是创建一个一对一复制，把 1、2 的数据迁往 B，然后做删除过期数据操作，把 A 节点的 1、2 桶数据都删除；另一种做法是增加一个额外节点 C，创建一对多复制，把 1、2 数据迁往 B 节点、3 数据迁往 C 节点，迁移结束后，节点 A 不再提供服务，而由 C 替代。个人建议，究竟采用哪种方式，需要按照过期数据的规模决定，如果过期数据不多，可以直接在源节点上进行删除操作，如果量比较大，还是以第二种方法为好。

11.1.2 迁移操作配置方法

由于在线迁移操作比较复杂，下面举例完整的说明如何创建一个迁移操作。

1. 配置 DDB 系统参数。在线迁移功能需要用到的系统参数都集中在“DBN 复制设置”页中。另外，在线迁移实际操作中，调用了大量的 `bash` 脚本，而 `bash` 脚本的路径必须预先配置好。在“其他设置”页面中配置 DDB 脚本所在路径。注意：DDB 脚本路径应该配置到脚本所在的根目录，在线迁移的脚本都在脚本目录的 `loadbalance` 子目录下，系统在调用在线迁移脚本时会自动在脚本目录下增加 `loadbalance` 这一级子目录。
2. 增加 DBN。在线迁移的目的节点，大多数情况下都是 DDB 中不存在的，因此需要先把节点添加到 DDB 中。增加 DBN 时，需要注意的是，如果该节点当时还不存在，应该把“同时在节点上创建用户”选项去掉，否则会增加失败。除了一般的 DBN 配置，`ssh` 用户名、`ssh` 端口、数据库配置文件名也必须同时指定
3. 在源节点和目标节点的相应服务器上，分别部署迁移 `bash` 脚本，位置必须与参数配置中的路径相符
4. 检查节点之间的 `ssh` 命令通道是否执行通畅。其中包括需要保证 `master` 节点到各数据库节点 `ssh` 命令执行成功；以及数据库源节点到目标节点 `scp` 命令执行成功
5. 到在线迁移页面中，增加一个新的在线迁移任务
6. 选择源节点、目的节点、迁移策略、定义桶对应关系。在桶对应关系界面中，可以详细自定义哪些桶被迁移到目标节点，哪些桶仍旧保留在源节点上
7. 其他一些同步超时时间等参数，可以使用默认值
8. 启动在线迁移任务，启动之后，可以查看迁移日志，跟踪执行情况
9. 当前步骤执行成功后，迁移任务会停在 `prepare` 阶段，等待管理员确认正确性
10. 确认操作正确后，继续执行迁移任务，通知所有的 `client` 修改桶映射关系，执行成功后最后完成一个迁移任务

具体在线迁移界面的详细说明，请参见管理工具章节的相关说明。

11.1.3 非标准均衡函数的处理

一般来说，DDB 应用都采用标准的哈希方法来生成桶号，如果是数字类型的均衡字段值，直接对桶数取模，如果是字符串类型的均衡字段值，则调用 `java` 的字符串哈希函数后对桶数取模。但同时，DDB 支持其他类型的均衡函数，这会对在线迁移产生影响，下面讨论之。

用户自定义均衡函数

如果采用了用户自定义均衡函数，则桶号的生成规则完全由用户控制，因此在线迁移时，DDB 完全不知道均衡字段值与桶号的对应关系。为了解决这个问题，用户必须在迁移之前，手工在每个节点添加存储过程，存储过程名称与用户自定义均衡函数名称必须相同。存储过程的内部实现应与自定义均衡函数完全一致，即同样的均衡字段值，通过 `java` 版用户自定义均衡函数运算出来的结果应与存储过程版均衡函数运算结果完全一致，这点必须注意。

存储过程书写时，要注意参数与 `java` 版均衡函数有些不同。举例来说，如果原先均衡函数如下形式：

```
public int UserHash(long value) {
    return ((int)value) % bucketCount;
```

```
}

```

存储过程必须多一个传入参数 **bucketCount**:

```
CREATE FUNCTION UserHash (i integer, bucketCount integer) RETURNS integer
RETURN i % bucketCount;
```

bucketCount 由 DDB 在线迁移模块负责传入正确的数值，代表该均衡函数某个策略的桶数。参数名称并不必要一定是 **bucketCount**，也可以是其他名称，但必须是参数列表中的最后一个。

在添加存储过程时，有可能 MySQL 服务器会抛错不允许执行，这时需要设置要在程序里调用存储过程，必须把服务器的数据库配置文件模版加上：
log_bin_trust_function_creators=1

采用 DDB 组编译的 MySQL 内嵌字符串均衡函数

为了解决字符串类型均衡策略无法实现迁移的限制，DDB 编译了一个内嵌到 MySQL 服务器的字符串均衡函数（**strjavahash**）。该均衡函数使用与 java 字符串哈希函数一样的处理机制把均衡字段值转换为桶号，这样在线迁移执行时就可以调用该函数来计算桶号。

要使用 **strjavahash** 函数，必须使用 DDB 组提供的 MySQL 安装包来安装 MySQL 服务器，并且在创建均衡策略时，把均衡函数选为：**DBStringHash**。

11.1.4 对于出错情况的讨论

数据迁移同日常的 DDB 操作不同，期间涉及到大量 **bash** 脚本等同外部环境打交道的操作，完备的出错机制对于在线迁移是有必要的。另外，数据库管理人员也应该知道在出现错误之后应该怎么处理。好在线迁移有完备的日志机制，管理员可以清楚地知道目前所处的阶段及错误原因。

在迁移准备阶段产生错误，这是最常见的，好在这个阶段产生的任何问题都不会对 DDB 系统产生负作用。如果发生了错误，可以有两种方式来解决：通过日志查找到错误原因，以手工处理的方式解决问题，确保完成了所有准备阶段的任务后，在管理工具中手工修改任务的所处阶段，从未开始改为准备阶段完成；或者是查找到错误原因进行相应处理后，直接在管理工具中选择执行当前阶段，DDB 自动为当前任务再做一次准备操作。有一种比较特殊的情况是多对一迁移时的主键冲突，如果出现这种问题，可以选择修改任务的“忽略主键冲突”属性。

在线上阶段产生错误，一般比较少见，但每种情景的处理方式都不同，需要分开讨论。

1. 正如前面迁移流程中所描述的，线上阶段开始时，会先锁住源节点数据库，然后等待同步完成。这样做的目的是为了保证在迁移过程中绝对不会有数据丢失发生。等待同步完成的最长时间限制，是可以在迁移任务的参数配置中修改的，这个参数好比一把双刃剑，如果太长，这个期间的线上应用都会被卡住，如果太短，可能从节点还来不及同步完成。如果在指定的同步最长时间之内没有完成节点同步，则任务执行将会失败。如果发生这种情况，问题不大，只需要适当加大同步时间最大值后，重启任务就可以。如果是多对一迁移，还要额外注意，因为系统需要轮询每个从节点，查看是否同步完成，因此，同步最长时间起码需要大于轮询间隔×从节点个数，目前轮询间隔设为 5 秒。
2. 在线迁移，需要把新的桶映射关系等配置通知所有的客户端。如果在迁移任务中配置了不忽略通知客户端失败的异常，则有可能在此阶段发生任务执行失败的情况。在线迁移如果遇到这种情况，会自动先把之前停掉的 MySQL 重新启动，这样就相当于回

到了线上阶段的初始状态。DBA 管理人员如果发现这个错误，可以直接重新启动线上阶段的操作。通知客户端出错到线上阶段重新执行这段时间，不会造成数据丢失或其他错误，但因为某些客户端可能已经迁移通知成功，所以会自动增加查询语句的桶号条件，这可能会对性能产生轻微影响，建议发现这个错误后尽快进行重做。

3. 删除脏数据/过期数据发生错误。对于一般的脏数据删除，执行时间还是比较短的，而如果涉及到删除过期数据，则需要依据过期数据的数据量而定，有可能执行时间比较长。这期间，最有可能发生的是删除数据时因为锁超时而错误。DDB 在批量删除数据时，增加了锁超时重试机制，但这也不能保证完全避免出错。如果发生了问题，可以重新执行线上阶段的操作。同样，建议发现这个问题后尽快进行处理。

11.1.5 存在的应用限制

下面讨论在线迁移功能本身存在的一些限制。

1. 对 **DIRECT FORWARD** 语句产生影响。在线迁移执行过程中，有两种情况需要对 SQL 语句进行修正：**(a)** 删除脏数据过程中，为了保证脏数据不会影响到查询语句的结果，会在所有的查询语句中增加额外均衡字段等值条件；**(b)** 多对一迁移时，为了避免一个源节点上执行的 SQL 语句影响到从节点中不属于自身的数据，会对所有的更新删除语句加额外均衡字段等值条件。对于 DDB 中的 **DIRECT FORWARD** 语句，由于本身就不做语法解析，因此无法对其进行动态增加查询条件，为了保证数据严格正确，目前迁移过程中，在上述的两种情况下，如果发现有 **DIRECT FORWARD** 语句，则不允许执行。
2. 表结构的限制。这个问题比较复杂，讲清楚这个事情先说明一下对脏数据/过期数据的处理。删除脏数据过程调用了 DDB 已有的批量更新接口，依据一个基准索引进行数据分段操作。获取基准索引的规则如下：
 - a) 检查表中的主键，如果主键是单字段数字型，则采用该主键
 - b) 检查表中的主键，如果主键是多字段并且所有字段为数字，则采用主键的第一个字段
 - c) 遍历表中的所有索引，规则同上。返回一个单字段数字型索引或者多字段的所有字段为数字的索引
 - d) 如果还是没有找到符合条件的字段，则不允许执行迁移操作
3. 表分配 ID 的限制。目前在线迁移中删除脏数据阶段，为了避免全表扫描数据，会在执行迁移之前记录表主键第一个字段的最大 ID 值，删除数据时会依据此 ID 值往后扫描数据。这客观上就要求表记录 ID 值是按照递增的顺序进行分配。此点请数据库管理员特别注意，务必在迁移前仔细查看，否则可能导致有脏数据没有被清除。
4. 非超级用户。为了保证数据不丢失，会在切换桶关系的刹那暂时锁住源节点的数据更新操作，通过在源节点执行 **set global read_only** 来实现。但这条语句，只对非超级用户的连接起作用，因此，在线迁移时，请确保没有分配 MySQL 超级权限角色的客户端在执行操作。
5. 均衡字段为数字。在删除脏数据和重建目标节点过程中，要么表上有专门的 **bucketno** 字段用于直接判别，要么表上的均衡字段为数字，可以通过数字取模的方式获取桶号，要么是表采用的是自定义均衡函数。否则的话，无法判别哪些数据属于脏数据。

11.2 离线数据迁移

离线数据迁移通过在源数据库上执行 `insert...select` 首先将整个桶的数据导出到本地临时 `MyISAM` 表中，然后将包含迁移数据的 `MyISAM` 临时表 `scp` 到目的数据库服务器上。通过在目的数据库上执行 `insert...select` 将临时表中的数据导入到目的数据库中，最后在源数据库上执行 `delete` 操作删除桶数据。数据迁移最耗时的操作莫过于从源数据库导出数据和向目的数据库中导入数据，采用 `insert...select` 的导出和导入效率远远高于直接使用 `select` 和 `insert`。另外采用 `insert...select` 可以避免使用 `mysqldump` 可能带来的编码不一致问题。

导出迁移数据到临时表时，如果表使用了 `bucketno` 字段，可以使用该字段来选择迁移数据，否则只能通过使用 `abs(bf) % x in (buckets)` 来选择迁移数据，对于没有使用 `bucketno` 字段的表如果均衡字段类型不为整型，将无法通过 `insert...select` 语句获取迁移的桶记录，因此无法进行迁移。

在离线迁移过程中，允许用户继续操作正在迁移的表，但是这些操作是受限的。首先，迁移期间不允许用户访问正在迁移的桶记录，一旦用户操作涉及这些记录将会抛出异常。其次，在 `MySQL` 中执行 `insert...select` 语句将导致对执行 `select` 的表加记录锁的 `S` 锁，这将导致来自用户的所有对该表的写操作被锁定，特别的，如果一个写操作首先执行，耗时很长，也有可能后来执行的 `insert...select` 等锁超时。为此我们在数据迁移过程中会禁止所有迁移表上的写操作，用户的写表操作将会抛出异常。

离线数据迁移统一由 `Master` 发起，并且由 `Master` 控制每个 `DBN` 上的数据导出和导入，不需要 `Agent` 的参与。由于迁移的过程中需要将临时表从源数据库服务器 `scp` 到目的数据库服务器，`Master` 需要具有到所有 `DBN` 服务器的 `ssh` 权限。一个表上的迁移可以被定义为一个迁移任务，作为一个数据迁移的最小执行单位。迁移任务中可以定义迁移多个桶。离线迁移时 `Master` 采用多线程机制可同时并行执行多个迁移任务。为了保证访问源数据库的效率，每个源数据库对应一个迁移线程，迁移任务按照源数据库被分配给这些迁移线程执行，迁移线程内部将顺序执行分配到的任务。每个迁移任务都维护了一个自身的执行状态，`DBA` 可以通过管理工具实时监控各迁移任务的执行状态。

离线迁移的具体流程如下：

1. `Master` 通知所有 `Client` 准备离线迁移。`Client` 检查各离线迁移任务是否有效。
2. `Master` 通知所有 `Client` 开始离线迁移。`Client` 对各迁移任务所涉及的表加写锁，同时记录所有迁移的桶。此后 `Client` 上所有访问这些桶或写这些表的操作都将抛出异常。通知完毕后 `Master` 修改自身状态为正在迁移。
3. `Master` 根据所有迁移任务中的源数据库启动对应的多个迁移线程并行地执行迁移任务。
4. 迁移线程顺序地执行被分配的迁移任务。在执行某个迁移任务时，首先将源数据库中的迁移桶数据通过 `insert...select` 导出到一个 `MyISAM` 临时表中，然后 `scp` 该临时表到目的数据库服务器的 `MySQL` 数据目录，接着在目的数据库上通过 `insert...select` 将临时表中的数据导入到表中，最后从源数据库上删除迁移的表记录。
5. `Master` 等待所有迁移线程执行完毕后通知 `Client` 迁移结束。`Client` 修改迁移桶的映射，从原来的指向源数据库改为指向目的数据库，同时允许迁移表上的写操作。`Master` 在通知完毕后修改自身配置中的桶映射，并恢复正常运行状态。

迁移任务执行过程中可能出现如下几种状态：

- 未开始：迁移尚未开始
- 请求迁移：`Master` 正在向 `Client` 发送迁移请求
- 请求失败：`Master` 向 `Client` 请求数据迁移失败，需要重新执行作为补救
- 调度中：迁移开始后任务迁移线程的处理队列中尚未开始执行
- 执行中：迁移任务正在被迁移线程执行
- 执行失败：执行不成功，需要进行补救以保证数据访问一致性
- 已执行：任务已经执行完毕，但整个迁移操作尚未结束

- 反馈中：Master 正在通知 Client 迁移结束
- 反馈失败：Master 通知 Client 迁移结束失败，需要重新执行作为补救
- 成功完成：任务最终成功完成（结束状态）
- 失败结束：任务最终执行失败但不需要补救措施（结束状态）

上述状态中“请求失败”，“执行失败”，“反馈失败”都是需要进行补救的任务状态，任务处于这些状态时 Client 和 Master 都处于中间状态，Client 上的用户操作仍然是受限的，需要重新对这些任务进行迁移，直到任务“成功完成”或“失败结束”。

11.3 在线修改表结构

11.3.1 实现流程

在线修改表结构实现包括两部分，jython 脚本执行在单个节点上的操作，java 端则进行统一调用并负责更新系统库以及 Client 端的配置信息。

假设需要执行在线改表操作的表名为 OriginalTbl，对于单个节点，基本执行流程如下：

1. 例行检查，检查将要创建的表以及触发器等是否在节点中存在同名的对象，包括镜像表 __oak_OriginalTbl，用来保存数据拷贝进度的临时表 __oak_tmp_OriginalTbl，原表切换后的表 __arc_OriginalTbl，Insert 触发器 OriginalTbl_AI_oak，更新触发器 OriginalTbl_AU_oak，删除触发器 OriginalTbl_AD_oak，数据拷贝存储过程 OriginalTbl_SP_oak 等。如果有同名对象已经存在，则抛出异常并返回。
2. 创建在线修改表结构所需要的表、触发器和存储过程等，并在原表和镜像表的共有索引中选定一个唯一索引作为数据拷贝的依据。根据所选定的唯一索引，将最小值和最大值保存到临时表中。镜像表采用 CREATE TABLE LIKE OriginalTbl 的形式创建，然后在镜像表上执行所指定的 ALTER TABLE 语句。
3. 循环调用拷贝数据的存储过程，把数据按照选定的索引从小到大一段一段地从原表拷贝到镜像表中，而且只拷贝原表与镜像表共有同名字段的值。当不存在比临时表中所记录的最大值还要大的记录时，表示旧数据都已经拷贝完成，而拷贝过程中的新的数据则已经由触发器自动同步到镜像表中。
4. 执行表切换操作。原表 OriginalTbl 重命名为 __arc_OriginalTbl，镜像表 __oak_OriginalTbl 重命名为 OriginalTbl，新的表结构启用。接着删除拷贝数据的存储过程 OriginalTbl_SP_oak 和临时表 __oak_tmp_OriginalTbl。出于安全的考虑，旧表 __arc_OriginalTbl 不会被删除，需要由 DBA 确认没有问题后，再到节点上通过手工删除。在下次对同一张表进行在线改表操作时，如果旧表还没有被删除，则会在检查阶段抛出异常，提示同名表已经存在。

11.3.2 配置和执行限制

1. jython 脚本配置
脚本配置比较简单，只需要保留发布包的结构，各 jython 脚本都在 scripts/onlineAlterTable 文件夹中即可。jython 脚本被访问过一次后，就会在内存中保留定义，如果在 master 持续运行阶段调用过 jython 脚本，如果想通过修改 jython 脚本满足特殊要求，则需要在 jython 脚本修改后重启 master，清除内存中旧的脚本信息，新的 jython 脚本才能生效。
2. 执行限制
除去上面有提到的不能存在同名表或者触发器等，在线修改表结构还有如下限制：
表类型需为 innodb；
原表和镜像表需存在至少一个相同的唯一索引；
表重命名操作和 CHANGE COLUMN 操作不支持。

11.3.3 特殊情况处理

特殊情况主要分为两种，一种是操作异常，另一种则是任务执行过程中的 **master** 重启。

对于操作异常，包括例行检查中抛出的错误或者其他阶段抛出的错误，通过日志或者调试信息都可以找到异常的原因。如果是例行检查中抛出的错误，可以根据异常提示到节点进行修正，然后再次启动任务。而如果是数据拷贝阶段等出现异常，建议执行撤销操作或者到节点上进行手工清理，然后重做任务，避免产生错误数据。

对于 **master** 重启，目前支持的情况有：

1. 数据拷贝过程中，**master** 需要重启。可以先把任务暂停，**master** 重启后，暂停的任务可以继续启动。重启后的任务的实现流程中，例行检查是如果应该存在的表、触发器等不存在则抛出异常，不进行表等对象的创建，直接进入数据拷贝阶段并且是从上次暂停的地方开始。
2. 数据拷贝过程中，**master** 异常重启，任务没有暂停。**master** 启动后，该任务处于“拷贝中”的状态，可以再次启动该任务，数据拷贝会从上次停止的地方继续进行下去。或者执行撤销操作，不需要到节点进行手工清理。
3. 数据拷贝结束后，**master** 重启。**master** 启动后，启动该任务，则进入表切换操作，**master** 重启没有影响。
4. 对于其他异常情况，建议根据实际情况到节点进行手工清理操作，然后重做任务，避免产生数据错误。

11.4 镜像管理和负载均衡

MySQL 本身提供了复制机制（**replication**），可以自动把一个 **mysql** 数据库上的操作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，实现负载均衡，提高分布式数据库可扩展性。

DDB 把 MySQL 的复制机制也整合进来，通过 DDB 进行统一分配管理。这样的话，镜像库的数据就能够同时为线上应用提供服务，把一些不重要的只读操作分流到镜像库。并且，在线上库停机维护时能够很方便的暂时把应用切换到镜像库机器上。通过集成镜像管理，数据库管理员能够方便的进行建立镜像库等操作，而如果手工直接创建镜像库，相比较起来还是比较繁琐的。

11.4.1 主要功能列表

1. 提供有效的负载均衡功能

通过在 DDB 中引入镜像节点对 DBN 节点的只读操作分流。

支持一个 **Master** 节点对应多个 **Slave** 节点的情况。可根据一定的策略和权重分流只读操作。

2. 可以让用户指定不同的语句执行策略

只能在 **Slave** 上执行 **hint: slave_only**（**Slave** 不可用则失败）

优先在 **Slave** 上执行 **hint: slave_prefer**（先在 **Slave** 上执行，**Slave** 不可用在 **Master** 上执行）

根据均衡策略执行 **hint: load_balance**（根据均衡策略选在 **Slave** 或 **Master** 上执行）

只能在 **Master** 上执行 `no hint or hint: master_only`（对实时性要求高的语句）

SQL 语句格式：

```
/* LOADBALANCE(type = loadBalanceType [, delay = delayTime]) */ select ...
```

`loadBalanceType` 有四种类型：

```
* MASTERONLY (DEFAULT)
* LOADBALANCE
* SLAVEONLY
* SLAVEPREFER
```

`delayTime` 是整形，默认值为 `Integer.MaxInt`, `delayTime` 参数是可选的

3. 提供 mysql 复制管理

选择性的表复制（所有 DBN 相同）。可选择只复制写操作不多，而读操作频繁的表进行复制。

支持自动化重建某些 **Slave** 节点

停止 / 启动 **Slave** 节点上的复制

查看 **Slave** 的复制状态

Slave 复制落后报警和自动失效

4. 提供数据库节点管理

停止所有 **Client** 对某个 DBN（**Master or Slave**）的数据访问服务

动态删除和增加 **Slave** 节点

动态切换主从节点

设置节点访问权重、复制配置参数

11.4.2 相关脚本说明

DDB 的复制管理依赖于 `bash` 脚本实现，所以在实际应用时，必须保证在目标机器上已经部署有整套 `bash` 脚本，脚本目录可以在系统参数配置中指定。

脚本清单

名称	执行位置	脚本说明
<code>build-slave.sh</code>	从节点上运行	重建镜像库，调用 <code>mysql-backup.sh</code> 、 <code>recover-mirror.sh</code> 、 <code>rep-from-backup.sh</code> 来实现相关功能
<code>change-unrep.sh</code>	从节点上运行	变更一个从节点的不复制表，并修改 <code>mysql</code> 配置文件

common.sh		定义其它脚本用到的公用函数
create-cnf.sh		根据给定的传入参数自动生成 mysql 配置脚本
mirror_template		镜像库 mysql 配置模板文件
mysql-backup.sh	主节点上运行	数据库备份到本地目录，然后上传到镜像数据库节点
mysql-control.sh		定义用于 mysql 数据库操作的公用函数
mysql-start.sh		用于启动 mysql 的脚本。由于 mysqld_safe 运行后， bash 脚本不会自动退出，所以专门增加了这个脚本
mysql-start-proxy.sh		启动 mysql 的脚本
parse-cnf.sh		用于操作 mysql 配置文件
recover-mirror.sh	从节点上运行	根据备份数据恢复镜像库
rep-from-backup.sh	从节点上运行	在新建立的从节点数据库上开启复制
reset-cnf.sh		把一个 mysql 配置文件中的所有复制相关选项恢复成默认值
start-slave.sh	从节点上运行	启动一个 slave 节点，并修改配置文件
stop-slave.sh	从节点上运行	停止一个 slave 节点，并修改配置文件
switch-rep.sh	master 上运行	镜像库与主数据库之间作主从切换

所有复制管理用到的环境配置文件清单：

名称	文件说明
rep.pass	定义复制管理中用到的用户名/密码信息

其中几个主要脚本的执行流程如下。

建立镜像库脚本

按照 DDB 中的配置在数据库节点机器上建立 **mysql** 数据库，并启动复制。执行流程：

- 考虑到此时镜像节点可能正在对外提供服务，所以先查看 **slave** 节点是否现在正被 **ddb** 使用。若正在使用则先阻止 **client** 对该镜像库的访问。（在 **master** 端执行）
- 在复制主节点上使用 **innobackup** 工具备份出 **mysql** 数据库文件
- 把备份数据库文件使用 **scp** 上传到复制从节点机器上
- 若镜像节点的 **mysql** 配置文件不存在，则创建新配置文件
- 关闭镜像库
- 若原有 **mysql** 目录已存在，则先备份（**mysql** 目录中存有镜像库的用户权限信息）
- 删除原有镜像库目录
- 对备份数据库文件使用 **innobackup** 的 **apply log** 恢复成原始格式
- 把恢复后的数据文件拷贝到镜像库目录
- 恢复原有 **mysql** 目录
- 启动镜像库 **mysql** 服务
- 回滚悬挂事务
- 创建两个系统用户：本地超级用户----用于在本地登陆 **mysql**；**ddb** 超级用户-----用于 **ddb**

在 **master** 端登陆镜像库做日常维护操作

- n) 在 **master** 上建立用于复制的帐户
- o) 设置镜像库上的复制参数，并启动复制
- p) 在新 **slave** 节点上分配所有 **ddb** 中已有用户的权限（在 **master** 端执行）
- q) 恢复 **ddb** 对该节点的访问（在 **master** 端执行）

启动 mysql 复制脚本

启动镜像库的复制。执行流程：

- a) 执行 **mysql** 的“**START SLAVE**”命令
- b) 然后修改 **mysql** 配置文件中的 **skip-slave-start**

停止 mysql 复制脚本

停止镜像库的复制。执行流程：

- a) 执行 **mysql** 的“**STOP SLAVE**”命令
- b) 然后修改 **mysql** 配置文件中的 **skip-slave-start**

修改镜像库不复制表脚本

修改镜像库上不复制表。执行流程：

- a) 考虑到此时镜像节点可能正在对外提供服务，所以先查看 **slave** 节点是否现在正被 **ddb** 使用。若正在使用则先阻止 **client** 对该镜像库的访问。（在 **master** 端执行）
- b) 停止 **mysql** 服务
- c) 修改 **mysql** 配置文件的 **replicate-ignore-table**
- d) 重新启动 **mysql** 服务
- e) 恢复 **ddb** 对该节点的访问（在 **master** 端执行）

Master/Slave 主从切换脚本

用于把指定的 **slave** 节点与其 **master** 节点做主从切换，需要同时修改其它所有 **slave** 节点的复制指向。执行流程：

- a) 禁止 **ddb** 对原 **master** 的访问。屏蔽 **ddb** 中诊断线程对所有该 **master** 下 **slave** 节点的轮询。（在 **master** 端执行）
- b) 确认所有 **slave** 节点都已同步，并且新 **master** 上没有“不复制表”
- c) 检查新 **slave** 节点的 **mysql** 配置文件，把 **replicate-ignore-table**、**skip-slave-start** 相关项全部清空。并重启数据库，使配置文件中的相关修改生效
- d) 把原 **master** 数据库设为只读，防止修改 **slave** 节点复制指向时有 **sql** 语句被漏掉
- e) 重启所有 **slave** 复制，因为 **master** 重启后 **slave** 的复制可能处于异常状态
- f) 再次检查每个 **slave** 节点，确认在修改配置及重启期间 **slave** 能够跟上 **master**
- g) 停止所有 **slave** 复制
- h) 清空新 **master** 节点中的复制信息，使之成为主节点
- i) 修改所有 **slave** 节点的指向，指向新的 **master** 节点
- j) 恢复原 **master** 数据库为非只读
- k) 恢复 **ddb** 对原 **master** 的访问，恢复对所有 **slave** 节点的轮询（在 **master** 端执行）

12 高可用性实现

这要从 DDB 的架构谈起。众所周知，高可用性的目的是避免单点故障。在 DDB 架构中，数据被分片到各台不同的数据库服务器，所有的 DDB 客户端和数据库服务器，都需要 DDB 的 master 服务器来做统一管理。如果数据库服务器出现问题，并没有 DDB 层面的解决方案，用户可以通过 DRBD 或者 MySQL 复制机制，在数据库服务器出现故障时，自动切换到备份机。而如果是 DDB 的 master 出现故障，问题就要严重得多，可能导致整个 DDB 系统瘫痪，下面讨论的主要对象就是 master 高可用性。

12.1 高可用性实现原理

12.1.1 基于 zookeeper 实现高可用性

Zookeeper 是 Apache 组织下的一个开源项目，主要是基于分布式的架构，几台服务器同时对外提供服务，从而实现高可用性。Zookeeper 服务器具体介绍信息，可参见官方网站：<http://hadoop.apache.org/zookeeper/>。

DDB 用到了 zookeeper 以下方面特性：

1. 存储 master 的实时运行状态信息。因为 zookeeper 本身是高可用性的，所以不必担心数据丢失。另外，只有一份 zookeeper 配置信息，所以可用作在多个 DDB 的 master 之间进行协调。目前 master 的运行状态信息，存储在 /ddb/ddb_demo(DDB 名称)/master 路径下。
2. 存储 zookeeper 服务器列表。这些信息同样存储在 zookeeper 内，方便 DDB 的 master 和客户端读取。目前 zookeeper 服务器列表，存储在：/serverlist 下，具体内容为：127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183，列表中的每项代表一台 zookeeper 服务器的监听 ip 地址和端口。需要提醒一下，这些信息并不是 DDB 自动维护的，需要 DBA 人员在搭建 zookeeper 服务器环境时手工添加。
3. Zookeeper 自身的通知机制。客户端可以通过在 zookeeper 上注册一个 watch，一旦用户关心的数据发生改变，zookeeper 服务器负责给相关用户发出通知。
4. 作为通用接口给其他第三方软件提供接口。因为 zookeeper 的存取接口是标准的，因此其他系统可以不依赖于 DDB 提供的接口，直接在 zookeeper 上获取需要的信息。

12.1.2 Master 高可用性实现流程

Master 在高可用性模式下，至少应该有两台服务器同时运行。服务器有可能处在下面两种状态下：

1. **Leader 角色。**leader 角色下的 master 服务器，实际对外提供服务。启动流程如下：
 - 1) master 服务器在启动时，首先连接到 zookeeper 服务器，并在服务器上注册自己。注册信息包括路径和内容两部分。注册信息路径是下面的形式：/ddb/ddb_demo/master/master-0000000266。其中 ddb_demo 是 DDB 名称，每个 DDB 数据库都有一个名称用于唯一标识自己。master-0000000266 表示当前的 master 服务器进程实例，每次 master 注册时，都会从 zookeeper 服务器获取一个递增 ID 来唯一标记自己这个进程实例。注册信息内容，目前的格式为：127.0.0.1:8888-7777。分别代表 master 服务器的监听 ip 地址、监听 dbi 端口号、监听 dba 端口号。
 - 2) 获取 zookeeper 服务器列表。这个列表存储在 zookeeper 服务器的 /serverlist 目录下。因为 zookeeper 服务器，可能是整个公司的所有 DDB 公用一套环境，因此，如果 zookeeper 配置改变的话，可以方便的修改这个列表，每次 DDB 的 master 登录到

- zookeeper** 服务器时，都能发现变更并自动更新 **DDB** 配置。当然，前提是原来的 **zookeeper** 服务器中至少有一台的地址没有改变，否则 **master** 根本无法连接到新 **zookeeper** 环境。
- 3) 启动监听器，在 **zookeeper** 服务器上注册 **watch**，监听 **/ddb/ddb_demo/master** 目录下的数据是否有改变。初始启动时，通过获取 **/ddb/ddb_demo/master** 目录下的所有数据列表来判断自己是否应该处于 **leader** 角色。判断依据为：根据前面提到的 **master** 服务器进程 ID(例如：0000000266)，查看自己的 ID 是否是所有列表中最小的 ID，如果是则把自己状态切换为 **leader** 角色。
 - 4) 启动监听器，在 **zookeeper** 服务器上注册 **watch**，监听 **/serverlist** 目录下的数据是否有改变。如果数据改变则更新 **master** 相应配置。
2. **Stand-by** 角色。**stand-by** 角色下的 **master** 服务器，实际上不对外提供服务，而只是作为 **leader** 服务器的备份机，一旦 **leader** 服务器出现故障，将切换为 **leader** 角色。同样的，**stand-by** 服务器在启动时，同样需要经过上面的四个步骤。只是在判断自己 ID 时，发现自己的 ID 不是所有 ID 列表中最小，则把自己角色定位为 **stand-by**。

上面说的 **leader** 角色和 **stand-by** 角色，在某些场景下将进行转换。如果 **zookeeper** 服务器无法正常监听到处于 **leader** 角色下的 **master** 服务器，比方说 **master** 通讯故障或进程退出，则 **zookeeper** 服务器上的该 **master** 注册信息将消失。处于 **stand-by** 状态下的 **master** 检测到这一变化，自动把自己的角色转换为 **leader**。

为了实现 **Master** 高可用性，**DDB** 的客户端同样需要做一些修改。客户端在连接 **DDB** 时，不再直接指定 **master** 的 ip 地址和端口号，而是指定 **zookeeper** 地址以及需要连接的 **DDB** 数据库名称。之后，客户端能够自动连接到当前处于 **leader** 角色的 **master** 服务器。**DDB** 客户端同样需要在 **zookeeper** 上增加监听器，如果新的 **master** 服务器处于 **leader** 角色，**zookeeper** 服务器将通知 **DDB** 客户端。客户端采用与 **master** 一样的 **leader** 选举策略，把 ID 号最小的 **master** 服务器作为 **leader**，这样可以保证能够登录到正确的新 **master** 服务器。

还有一种情况，**master** 本身运行没有问题，但是与 **zookeeper** 服务器的通讯中断，这可能由于网络问题或者 **zookeeper** 服务器本身的问题导致。如果通讯中断，**master** 会切换到重试状态下，如果在重试时间内能够恢复通讯，则 **master** 恢复到 **leader** 或 **stand-by** 角色；如果超过重试时间，仍没有恢复通讯，则 **master** 将退出关闭。这主要考虑到如果是 **master** 与 **zookeeper** 服务器的网络出现问题，而其他 **leader** 角色下的 **master** 通讯正常，则如果长时间不关闭故障下的 **master**，有可能导致两个 **leader** 角色的 **master** 同时对外提供服务。

leader 角色和 **stand-by** 角色互换时，需要再说明一下。**stand-by** 角色下，除了 **DDB** 基本配置信息，其他所有日常服务都没有启动。这样可以保证在切换到 **leader** 时，只要重新载入基本配置信息，在进程中就不会再有过期信息存在。但接下来继续考虑下面的情况：**leader** 角色切换到 **stand-by** 角色，然后再切换回 **leader** 角色。这种情况下可能就有过期数据存在。**DDB** 目前的做法是在 **leader** 切换到 **stand-by** 角色时，采用退出进程的方式。让外部脚本来重启 **master** 进程，这样保证所有以前的信息都被完全清除，而服务器重启后运行在 **stand-by** 角色下。

12.2 相关配置

如果需要启用 **master** 高可用性，需要在管理工具中配置以下参数：

1. 是否启用高可用性模式。用于控制 **master** 以普通方式运行或是连接到 **zookeeper** 服务器。
2. **Zookeeper** 地址配置，格式为：127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183。如

果 **zookeeper** 服务器地址有变动，**DDB** 会自动更新这个参数。

如果 **DDB** 启用了高可用性模式，则客户端应该以下面格式的 **url** 来连接 **master**：新的 **url**，以 **zookeeper://** 开头，**host:port** 是 **ZooKeeper** 本身的地址，**ddbname** 是需要连接的那个 **DDB** 的全局唯一标识符，**params** 是额外的那些参数，比如：
[zookeeper://host:port\[,host:port\]/ddbname\[?params\]\[,ddbname\[?params\]\]](#)

12.3 搭建 zooKeeper 服务器环境

ZooKeeper 本身是一个高可用性的集群，它可以在只有 2 个节点的情况下跑，不过如果要能容纳一个节点的失败，则至少需要 3 个节点，一般推荐的节点数为 $1+2*n$ ，**n** 是能够承受的节点失败数量。

具体配置的话，列几个要点：

每个节点需要一份自己的配置，比如 **zoo1.cfg**，并需要在节点启动之前，导出环境变量 **ZOO_CFG=zoo1.cfg**。这个文件中必需要写明每个其他节点的地址信息，如：

```
server.1=localhost:2881:3881
server.2=localhost:2882:3882
server.3=localhost:2883:3883
```

另外也要写明 **datadir** 和 **clientPort**：

```
# the directory where the snapshot is stored.
dataDir=/var/zookeeper/server1/data

# the port at which the clients will connect
clientPort=2181
```

在上面提到的 **dataDir** 目录中，需要有一个文件，名字叫 **myid**，没有后缀名，内容就是简单的一个数字，表明自己在前面的节点信息中，是处于那个位置。比如填 1，那么就说明自身是 **server.1**

每个节点配置好后，就可以启动了，顺序可以随机。

启动之后，管理员注意需要初始化一些信息：

1. **/serverlist** 节点，需要把目前正确的 **ZooKeeper** 服务器连接字符串保存到这个节点信息里。
2. 创建好 **/ddb/<ddbname>/master** 这个路径上的三个节点，节点数据无所谓，关键要有，因为 **master** 注册的话，是直接在最后一个节点上创建子节点的。

目前就这些需要配置。

13 均衡策略最佳实践

本章讲述如何设置均衡策略的最佳实践。所谓最佳实践者，即 **E 文之 Best Practice** 也，也就是说本章只是说设置均衡策略时最好要这样做，如果不这样也不一定会导致出大问题，但对此系统是不做保证的。

均衡策略设置主要要处理如下四个问题：

1. 如何决定每个表应当采用什么均衡字段？

2. 如何决定哪些表应采用相同的均衡策略，顺序如何？
3. 什么时候该用 `bucketno` 字段，什么时候不该用？
4. 如何决定均衡策略应当包含多少个 `bucket`？

现一一解答之。

13.1 如何决定每个表应当采用什么均衡字段？

每个表应采用什么均衡字段是由对这个表的访问模式决定的。在通常情况下，均衡字段的选取可由如下的规则决定：

规则 1: 对表中的某字段 A，当且仅当应用在访问（包括 SELECT / DELETE / UPDATE）这个表时经常指定 A 上的等值条件时，字段 A 才是均衡字段的一个好的候选人。

如果根据上述规则，有多个均衡字段候选者而根据上述规则无法判断其优劣时，则要使用到如下的规则：

规则 2: 当依据规则 1 产生多个候选均衡字段时，倾向于使用 `distinct` 值比较多的那个。

比如说，设有一个表 `student(name, dept, age...)`，系统在访问这个表时总是指定 `name` 和 `dept` 上的等值条件，如：

```
SELECT * FROM student WHERE name = '张三' and dept = '计算机';
```

这个时候根据规则 1，`name` 和 `dept` 都是均衡字段的候选者，难分高下。但一般来说，`name` 的 `distinct` 值总是比 `dept` 的要大，因此根据规则 2，应该选中 `name` 做均衡字段，而非 `dept`。

规则 2 背后的依据是：分布式数据库系统是通过均衡字段的值进行哈希计算来决定数据的存储位置，若均衡字段的 `distinct` 值太少，则哈希值分布不均衡的概率较大，从而影响到系统的负载均衡实现。

实际上还有第三条规则，**当多个表需要使用相同的均衡策略时，表的均衡字段选取时就要统一考虑，不能各管各**，这个在下面再讲。

13.2 如何决定哪些表应采用相同的均衡策略，顺序如何？

这个问题的简单回答是：**不同的表尽量不要使用同一个的均衡策略，除非这些表之间存在外键引用关系。**

要注意“不同的表尽量不要使用同一个的均衡策略”指的是对不同的表，不要指定相同名称的均衡策略，但你完全可以指定不同名称，但其内容（即包含的 `bucket` 数和各 `bucket` 的 `url` 都相同）完全相同的均衡策略。

当多个表之间存在外键引用关系，则采用同一个均衡策略通常是个好主意，有助于提高系统性能。但为这些表指定均衡字段时需要统一考虑，规则如下：

规则 1: 这些表之间被引用的表称为主表。对于主表，均衡字段要求是被引用的属性，对于其它的表，均衡字段要求是引用的属性。

需要注意的是使用相同均衡策略的表之中要求有且只有一个主表，且这个表在配置时需要

最先指定，即在 `DBClusterSetup.xml` 中需要在这个表的定义放在使用与之相同均衡策略的其它表之前，通过配置助手工具时也需要在创建使用与之相同均衡策略的其它表之前创建这个表。

如设有以下的表：

```
CREATE TABLE User (  
    Id BIGINT PRIMARY KEY,  
    Name VARCHAR(50)  
);  
  
CREATE TABLE Blog (  
    Id BIGINT PRIMARY KEY,  
    UserId FOREIGN KEY REFERENCES User (Id),  
    Title TEXT,  
    ...  
);  
  
CREATE TABLE Music (  
    Id BIGINT PRIMARY KEY,  
    UserId FOREIGN KEY REFERENCES User (Id),  
    Artist VARCHAR(100),  
    ...  
);
```

为这些表使用同一个均衡策略是比较好的，其中 `User` 作为主表，根据上述规则，`User` 表的均衡字段为 `Id`，`Blog` 和 `Music` 表的均衡字段为 `UserId`。

但即使多个表之间有相互引用关系，不使用同一个均衡策略也是可以的。在有些时候，这一点甚至是达不到的，如设再有一个表 `BlogComment`，引用 `Blog` 的 `Id` 字段，这个时候，若考虑 `Blog` 和 `BlogComment` 这两个表，则要使用同一均衡策略时 `Blog` 的均衡字段应为 `Id`。但考虑 `User` 和 `Blog` 表，则要使用同一均衡策略时 `Blog` 的均衡字段应为 `UserId`。在这种情况下，选择 `Blog` 和 `BlogComment` 使用同一均衡字段或选择 `User` 和 `Blog` 使用同一均衡字段都是可行的。

13.3 什么时候该用 bucketno 字段

在使用 DBA 管理工具建表时有个选项是是否使用 `bucketno` 字段，这一字段是什么？什么时候需要用这一字段？

其实 `bucketno` 字段的唯一用途是记录数据库中某一行从属于均衡策略中的那个桶。确定是否需要 `bucketno` 的原则也很简单：

1. 若指定了一个表要使用 `bucketno` 字段，则在 `CREATE TABLE` 时必须增加一个名为 `bucketno`，类型为 `SMALLINT` 的列

2. 若某个表的均衡字段是整数类型，则不需要使用 `bucketno` 字段
3. 若某个表的均衡字段是字符串类型，且该表是所属均衡策略的主表，则最好使用 `bucketno` 字段
4. 如果需要对某个均衡策略做数据迁移，采用这个均衡策略的所有表都需要使用 `bucketno` 字段或均衡字段为整数类型

举个例子，如果在使用其它的数据库时你要创建一个表 `student`，建表语句如下：

```
CRATE TABLE student (  
    Name VARCHAR(30),  
    Age INT,  
    PRIMARY KEY (Name)  
);
```

在使用分布式数据库时，就要在 `CREATE TABLE` 语句里加一个 `bucketno` 列，即使用分布式数据库时的建表语句是这样的：

```
CRATE TABLE student (  
    Name VARCHAR(30),  
    Age INT,  
    bucketno SMALLINT,  
    PRIMARY KEY (Name)  
);
```

并且在配置分布式数据库时要指定这个表使用了 `bucketno` 列。当然，若 `student` 表与其它表共享均衡策略且不是主表时，则不需要修改 `CREATE TABLE` 语句。

13.4 如何决定均衡策略应当包含多少个 bucket?

推荐值为存储使用该均衡策略的表的数据库目标节点数（即预计最终会使用的节点数）的 10 倍到 20 倍。如若分布式数据库现使用 3 个节点，将来预计可能增加到 6 个节点，而每个表都分布在各个节点上，则均衡策略中的 `bucket` 数可以指定为 60 个（60 是个相当的数字，即是 3 的整数倍，也是 4、5 和 6 的整数倍，这样，当将来系统从 3 个节点一直扩充到 6 个节点时，都能实现完美的均衡）。

13.5 常见问题及其解决方法

我想改一个均衡策略怎么办？

将数据导出（`isql` 中使用 `dump` 命令），修改均衡策略（注意需要停止服务，重启 `master`），然后导入数据（使用 `isql` 的 `load data infile` 命令）。

我建表的时候忘了加 `bucketno` 列了，怎么办？

DBAdmin 工具提供了增加 `bucketno` 字段的功能。另外，对于均衡字段为整型的表即使没有 `bucketno` 也可以进行数据迁移，迁移时会自动计算记录所在桶号。

14 性能优化指南

14.1 建索引

这一点与分布式数据库无关，但却是最重要的。分布式数据库的高性能依赖于底层数据库的高性能，而对底层数据库性能来说，建立需要的索引是最重要的。分布式数据库提供了语句级统计功能，方便检查是否建立的需要的索引。关于语句级统计信息的收集参见第 3.2 节。如果发现某条语句执行时间较长，需要特别注意，有可能就是因为没有建索引。

14.2 在均衡字段上指定等值条件

在均衡字段上指定等值条件可以使操作只发送到一个后台数据库节点。若不指定，则操作需要发送到每个后台节点，可能导致性能大幅下降。因此如果可能，请在均衡字段上指定等值条件。

14.3 减少使用分布式事务

分布式事务也会导致系统性能下降。为减少分布式事务，尽可能使用 `AUTO COMMIT` 模式，并指定均衡字段上的等值条件。如果事务包含多条语句，则尽可能要求各语句均衡字段上的条件的值都相同。

14.4 使用 PreparedStatement

尽可能的使用 `PreparedStatement`，减少语法分析，执行计划生成带来的 CPU 开销。

14.5 当不需要所有数据时，指定 LIMIT

一方面由于 `MySQL JDBC` 的限制，一方面为简化分布式数据库执行过程，分布式数据库在执行查询时总是一次性取出所有结果。因此，如果一个表比较大，而应用只需要访问前几条记录（如进行分页显示时），则不要忘了加 `LIMIT` 子句限制结果返回条数。

14.6 设定较大的 Java 虚拟机内存

分布式数据库客户端由于包含了很多 `cache` 机制，执行时也需要非常频繁的分配和释放对象，对于高并发的应用（如达到或超过 300 个连接时），若 `Java` 虚拟机内存设定过小，则

可能导致 JVM 进行频率的垃圾收集，导致性能大幅下降。

如何确定 JVM 内在设定是否过小？

在启动 Java 程序时可以加 `-verbose:gc -XX:+PrintGCDetails` 查看 GC 动作

对于已经启动的 Java 程序，可在 `top` 中查看。若 `top` 显示的 CPU 统计信息中 `us` 值较低，`sys` 或 `ni` 值较高，则很可能是垃圾收集过于频繁导致

14.7 设定合适的连接池

连接池设得过小将导致执行时获取不了可用连接，从而被迫等待。一般来说，连接池的大小应与应用与分布式数据库系统建立的连接数相当。在运行时，可通过分布式数据库收集的操作统计信息中的“`Get connection`”一项查看连接池使用情况。若获取连接的平均时间较大，则需要考虑增大连接池。

但连接池也不宜设得过大，否则将占用大量资源。虽没有进行过测试，但不建议将连接池设成大于 400 的值。

14.8 使用非 XA 连接来执行事务

当在 DDB 中开启事务时，由于 DDB 无法判断是否是一个 XA 事务，所以会默认使用 XA 连接来执行 SQL 语句。对于 MySQL 来说，使用 XA 连接执行语句和使用一般连接相比，性能会有 40%-50% 的下降。因此，我们鼓励用户自行判断是否需要使用 xa 连接。

可以使用下面两种方法来使用普通连接：

1. 连接数据库时，在 url 中增加参数：`defaultxa=false`。这样的话，所有本次连接中的事务执行时，都会使用普通链接；
2. 在开始事务时，使用 DDB 特殊语法来做单独控制：
`((DBConnection)conn).setAutoCommit(false,false)`；有效范围直到下次使用 `setAutoCommit(true)` 关闭为止。

如果用户使用了普通连接，而本次事务中包含一个以上数据库节点的非只读分支，执行时会抛出异常。

15 集成 Memcached 缓存

目前使用 DDB 作为后台数据存储的产品中，为了提高响应时间，都会再部署 Memcached 集群，并在应用层维护数据库和缓存数据的一致。为方便产品开发，DDB 增加对 Memcached 的支持，将分布式数据库访问、数据缓存以及一致性维护等一并隐藏在 DDB 提供的数据访问接口后面。应用通过 JDBC 接口即可访问数据，而不用关心数据是从底层节点还是 Memcached 中读取到的，也不用处理数据更新后 Memcached 与底层节点之间的一致性问题。

DDB 采用 `spymemcached` 访问 Memcached 集群，与博客 DAO 框架的方案类似。Memcached 中存储记录主键到记录数据的 `key-value` 键值对。如果用户直接使用主键进行数据查找(`where` 条件中有指定)，则采用最简单的方式，直接通过主键 `key` 查找 Memcached，如果找不到再到数据库中查找并把记录插入到 Memcached 中。如果用户的 `sql` 通过二级索引来查询，`dbi` 通过存储的表 `schema` 信息，自动先转换成查找记录主键的 `sql` 语句，在数据库上找到主键后，再在 Memcached 中查找对应的数据。对于事务以及复杂的查询等，DDB 都做了相应

的处理。目前暂不支持在 Memcached 中缓存二级索引到主键 id 的键值对。

通过 DDB 管理工具可以配置 Memcached 集群的地址、Memcached 连接数限制、以及 spymemcached 使用的连接配置项等。

15.1 配置和使用

15.1.1 系统参数配置

通过 DBA 管理工具-‘系统参数配置’-‘缓存设置’界面可以对 DDB 的 Memcached 使用进行配置，可配置项包括：

- ◆ 是否使用 Memcached 缓存：表示该 DDB 是否开启缓存功能；
- ◆ Memcached 服务器列表：所有 memcached 服务器的地址，要求是 ip:port 的形式，多个服务器地址用分号隔开；
- ◆ 连接操作队列长度：每个 memcached 连接异步操作队列可缓存的操作任务长度；
- ◆ 连接缓冲区大小：每个 memcached 连接使用的数据缓存区大小；
- ◆ 压缩阈值：当存入的 value 值超过该大小，先进行数据压缩；
- ◆ 操作超时时间：进行数据存取操作的超时时间；
- ◆ 缓存记录失效时间：memcached 记录的失效时间，0 表示不失效；
- ◆ 连接池最大连接数：每个 client 缓存的 memcached 连接最大值；
- ◆ 获取连接超时时间：从连接池获取 memcached 连接的等待超时时间；
- ◆ 连接空闲超时时间：空闲连接清理时间；
- ◆ 每条查询语句所存入缓存记录数限制：如果一次用户查询的记录数超过该阈值，则超过的部分中未命中 memcached 的将不再存入 memcached；
- ◆ 每次读取缓存记录数限制：用户需要读取大量缓存记录时可分批进行，每批不超过该阈值；

15.1.2 表关联的相应配置参数

还有一系列控制参数，用于控制单张表的缓存相关行为。包括：

- ◆ 该表是否启用 memcached 缓存
- ◆ 缓存 KEY 字段
- ◆ 缓存版本号

具体含义参见 DBA 管理工具章节以及本章后续章节的说明。

15.1.3 SQL 语句 hint

在用户的 SQL 语句中，可以使用 `/*MEMCACHED = true/false*/` 来控制单条 SQL 语句是否操作 memcached。具体的用例参见 SQL 语法详细说明章节。

DDB 在实现时，定义了一系列的参数用于控制何时何地进行 memcached 操作。究其原因，是因为每个应用都千差万别，哪些数据适合进入 memcached，哪些数据不适合进入 memcached，我们不可能事先作出判断。这些参数正是为了给用户以最大的灵活性，让用户自己决定何时操作 memcached。

实际使用时，用户为了开启 memcached 缓存功能，首先需要在系统参数中配置正确的 memcached 服务器地址及其他参数，并配置打开 memcached 缓存开关。然后，单独为每一张需要进行缓存的数据表做配置，选择合适的缓存 KEY 字段并打开 memcached 缓存开关。这些操作都完成后，DDB 就会自动把表中的数据放入 memcached 并维护数据一致性。

SQL 语句中的 hint 为用户提供更大的灵活性，如果用户在 SQL 语句中给出明确 hint，则 DDB 将无视数据表的缓存开关，直接按照用户要求进行缓存操作。但是这个选项需要慎用，举例来说：如果数据表没有开启缓存，而用户的查询语句中配置了 `memcached=true`，则 DDB 将会尝试从 memcached 获取数据，但如果用户没有在新语句中配置开启缓存更新，则 DDB 不会更新 memcached 中的数据，将造成数据不一致。

需要说明的是，即使所有的 memcached 开关都被打开，对一条查询语句来说，仍然有可能不会从缓存中读取数据。这是由于这条语句不适合从 memcached 中获取数据，具体原因和情

况列表参见下面章节的详细说明。

15.2 实现方式说明

15.2.1 表记录缓存方式

存入 Memcached 的数据记录由两部分组成：**KEY** 和数据记录。其中，**KEY** 由三部分组成：表名、版本号、缓存 **key** 字段值。从这个结构可以看到，如果多个 DDB 合用一个 Memcached 集群，就需要注意表名相同的情况。版本号主要用于实现失效指定表所有缓存记录的功能。缓存 **key** 字段是指用户指定的，用于在缓存中标识一条唯一记录的一个或多个字段，可以是有主键或者唯一键限制的字段。表只有在设定缓存 **key** 字段后才可能将记录存入缓存。

目前能够作为缓存 **KEY** 的字段，可以是主键或者唯一索引字段。一张表上满足这个要求的字段可能不止一个，这时 DBA 人员就需要仔细考虑哪个索引更适合。对于 memcached 缓存来说，**KEY** 的作用只是为了更快定位到缓存记录值，并不存储实际有用信息。在插入 memcached 时，即使是数值类型的 **KEY** 字段，也会被转化为字符串类型，实际内存开销是较大的。因此，实际选择时，需要挑选一个总长度最短的索引作为缓存 **KEY**。

把缓存中数据失效，是实际产品运行过程中经常会碰到的需求。产生这个需求的原因可能是多方面的：用户可能需要对一张数据表进行大规模更新，这时使用 DDB 来维护 memcached 数据一致性就不太划算，因为 DDB 需要额外从数据库中查出更新主键并进行 memcached 删除操作，这时就可以直接操作数据库，然后把这张表的缓存数据整个失效掉。

DDB 在集成 memcached 设计时，就已经考虑到表结构修改时需要让现有缓存数据仍然有效。

1. 当删除一张表时，不需要对缓存做额外操作，因为经过一段时间 lru 后，原表中的记录会被自然替换出缓存；
2. 表删除字段后，缓存中已经存在的记录不用删除，记录中多出来的字段会在返回给上层时被去除；
3. 和数据库类似，增加字段后，DDB 可以给缓存中已经存在的记录填充有效的新增字段默认值，保持与数据库一致；
4. 表字段类型修改后，DDB 会自动发现前后类型不匹配，把原有数据自动转换成当前新字段类型。

15.2.2 记录数据压缩

对于 memcached 缓存系统来说，每条记录占用的内存越小，意味着整个缓存集群能够缓存更多的记录，对系统的性能提升就越明显。因此，DDB 在实现时，充分考虑了如何对各种数据类型进行有效的压缩。

压缩分几个层面。对于整条记录，我们目前采用 gzip 进行压缩。考虑到对于小数据进行压缩可能性价比不高，目前提供了系统参数接口用于设置压缩大小阈值。当大于阈值时，才使用 gzip 进行压缩。

对于数值类型，我们参考 google 的 protobuf，采用 varint 进行压缩。具体原理说明如下：

Base 128 Varints

32 位整数使用 4 字节存储，32 位的整数值 1 同样要使用 4 个字节，比较浪费空间。Varint 采用变长字节的方式存储整数，将高位为 0 的字节去掉，节约空间

高位为 0 的字节去掉以后，用来存储整数的每一个字节，其最高有效位（most significant bit）用作标识位，0 表示这是整数的最后一个字节，1 表示不是最后一个字节；其他 7 位用于存储整数的数值。字节序采用 little-endian

示例：

整数 1，Varint 的二进制值为 0000 0001。因为 1 个字节就足够，所以最高有效位为 0，后 7 位则为 1 的原码形式

整数 300，Varint 需要 2 字节表示，二进制值为 1010 1100 0000 0010。第一个字节最高有效位设为 1，最后一个字节最高有效位设为 0。解码过程如下：

- a). 首先每个字节去掉最高有效位，得到：010 1100 000 0010
- b). 按照 little-endian 方式处理字节序，得到：000 0010 010 1100
- c). 二进制值 100101100 即为 300

ZigZag 编码

Varint 对于无符号整数有效，对负数无法进行压缩，protocol buffer 对有符号整数采用 ZigZag 编码后，再以 varint 形式存储

对 32 位有符号数，ZigZag 编码算法为 $(n \ll 1) \oplus (n \gg 31)$ ，对 64 位有符号数的算法为 $(n \ll 1) \oplus (n \gg 63)$

注意：32 位有符号数右移 31 位后，对于正数所有位为 0，对于负数所有位为 1

编码后的效果是 $0 \Rightarrow 0$, $-1 \Rightarrow 1$, $1 \Rightarrow 2$, $-2 \Rightarrow 3$, $2 \Rightarrow 4 \dots$ ，即将无符号数编码为有符号数表示，这样就能有效发挥 varint 的优势了

各 Java 类型的编码方式

Java 类型	编码方式
BOOLEAN, BYTE, SHORT, INTEGER, LONG	采用上述有符号的 varint 方式编码
FLOAT	用固定的 32 位表示
DOUBLE	用固定的 64 位表示
STRING, BYTE_ARRAY	采用定长数据类型表示，这类数据在开始位置使用一个 varint 表示数据的字节长度，后面接着是数据值
BIGINTEGER, BIGDECIMAL	通过 toString 方法转为字符串，然后采用 STRING 的方式编码，反序列化时由 string 创建相应的类对象
DATE, TIME, TIMESTAMP	通过 getTime() 获取毫秒数以 long 的形式保存，反序列化时再通过毫秒数创建相应的类对象
NULL	不写入内容

序列化后的结构

Object 列表序列化为 key-value pair 的字节流形式，key 里面包含 2 个东西，一个是该 Object 对应的 tag，另一个是 Object 的类型。这 2 个部分都是正整数，使用 $(tag \ll 5) \mid object_type$ 方式生成一个正整数，然后使用 base 128 varint 方式表示。key 后面跟着是属性值编码后得到的值。由于序列化后的值都可以计算出明确的长度，各 key-value 无须间隔符。对于 Object 为 null 的，key 后无 value，即紧跟在后面的下一个 Object 的 key 值。

反序列化

反序列化时，需要从第一个字节开始，根据最高有效位，解码出第一个 varint 即为第一个 key。key 最后 5 位是对象类型，key 右移 5 位得到 tag，根据对象类型可以知道接下来的 value 应该怎么样解码。依次下去，可将整个 List<Object> 以及 int[] tags 还原。

解码时返回的 key 如果为 0 表示数据末尾，所以上文中设定的类型编号从 1 开始，保留 0

google protocol buffer encoding

<http://code.google.com/apis/protocolbuffers/docs/encoding.html>

15.2.3 缓存使用情况统计

目前 **memcached** 系统中提供了下面几种状态统计信息：

- **Global hitrate**，总体查询命中率
- **Hitrate per slab**，每个 **slab** 的查询命中率（**memcached** 以 **slab** 形式来组织内存管理）
- **Evictions**，被替换出的记录条数

数据库管理员如果单凭这几个统计信息来判断缓存运行情况，其实是比较困难的。总体查询命中率有利于判断 **memcached** 集群内存是否足够，如果命中率偏低，则可能需要增加内存。**Evictions** 用于反映记录被替换出去的次数，但这个值跟运行时间有关，不能真实反映实际情况。鉴于上述情况，**DDB** 专门增加了几个缓存相关的统计功能。

缓存相关的统计分析功能

首先需要创建一个监控统计任务，运行一段时间用于收集客户端的执行情况，然后执行 **show stat table memcached** 命令，显示基于表的 **memcached** 操作统计信息。具体操作方式参见 **DBA 管理工具** 章节的说明。

task...	result_id	client_id	tablename	get_count	get_miss_count	set_count	set_datasize	del_count
1	1	1	t44a	6	2	2	8	3
1	1	1	t44b	0	0	0	0	0

从图中可以看到，统计信息以表为维度进行归类显示。分别展示了统计期间每张表执行 **get**、**get miss**、**set**、**delete** 的次数，并且显示了每张表在开启统计这段时间内实际插入 **memcached** 的数据长度（该长度以 **byte** 为单位，不包含缓存 **KEY** 的大小）。

通过查看统计分析结果，可以有效帮助数据库管理人员决定 **memcached** 缓存策略。如果一张表 **get miss** 次数很高，说明这张表的数据特征并不存在特别的热点数据，如果占用的内存很大，则可以考虑不对这张表的数据进行缓存。

缓存相关的 Client 统计

Client 统计功能并不需要开启统计任务，可以实时进行查看。在 **DBA 管理工具** 的 **Client** 统计页面中，选择一个客户端进行查看，结果如下：

v MEMCACHED	0	0	
v GET	0	0	
> GET_HIT	0	0	
> GET_MISS	0	0	
> SET	0	0	
> DEL	0	0	
> MEMCACHED_TIMEOUT	6	12347	2057

Client 统计信息中显示了缓存 **get**、**get miss**、**set**、**del**、**memcached** 操作超时的次数和平均操作时间。

15.3 SQL 操作 memcached 详细讨论

使用 **memcached** 的方式包括：

1. 不符合使用 **memcached** 条件，直接查询数据库；
2. 语句条件中包含缓存 **key** 字段等值关系，则可以直接从 **SQL** 语句中获得缓存 **key** 值，再通过该 **key** 查询 **memcached**，对于未命中的再查询数据库(只支持单表)；

3. 语句条件中不包含缓存 **key** 字段等值关系，需先查询数据库获得相应的缓存 **key** 值，再通过该 **key** 查询 **memcached**，对于未命中的再查询数据库；
4. 多表 **join** 的情况下，以表为单位进行分析，一条语句中可以有 1、3 混用的情况。
例如：

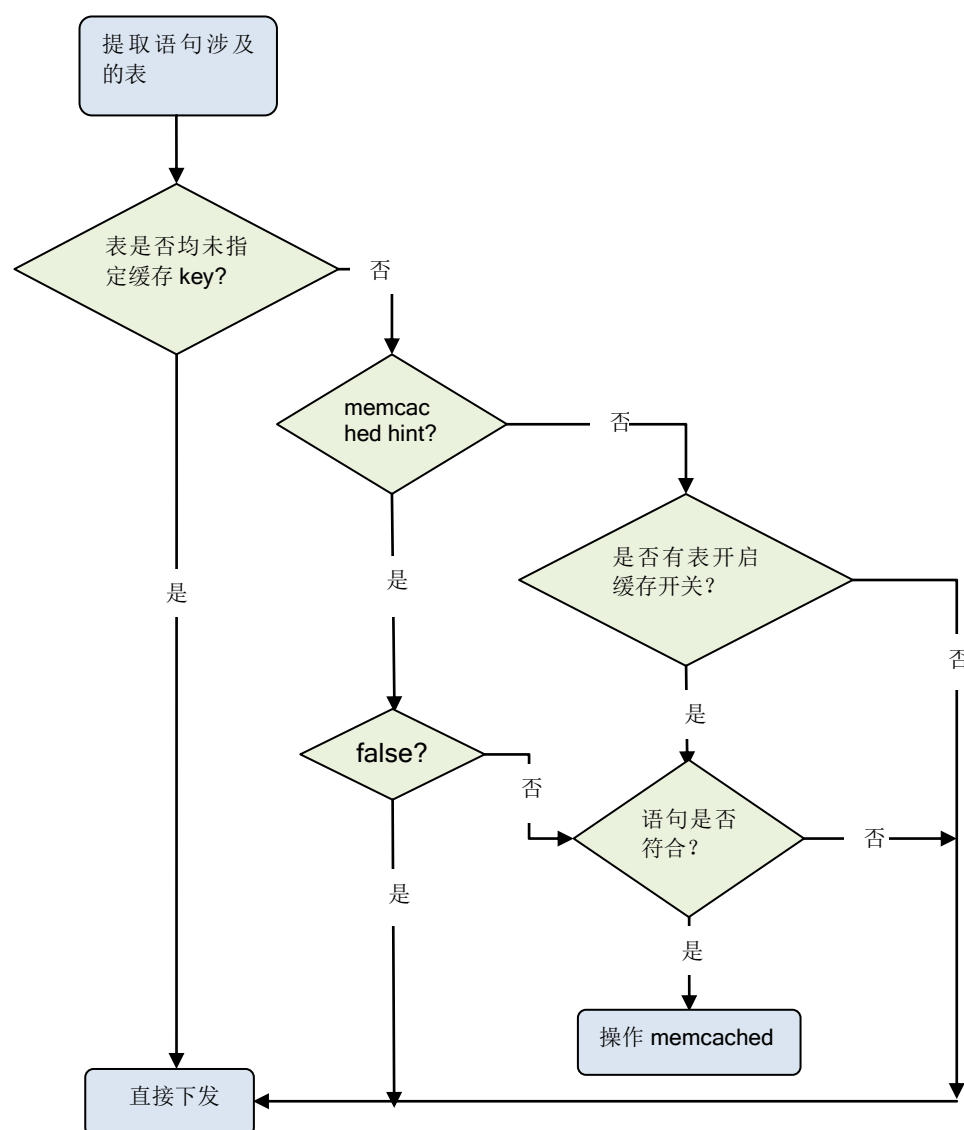
```
/*MEMCACHED = TRUE*/SELECT * FROM T1,T2 WHERE T1.id = T2.id;_ (T1 指定了缓存 key 字段，T2 没有。则 T1 使用 3 方式，T2 使用 1 方式)
```

15.3.1 Select 语句

使用 **memcached** 的流程：

1. 提取语句相关表，判断表是否指定了缓存 **key** 字段，在 SQL 语句没有明确说明是否使用 **memcached** 的情况下，判断数据表上有没有设置数据缓存到 **memcached**（详见下面的使用 **memcached** 的流程图）；
2. 如果语句包含：**FOR UPDATE**、**LOCK IN SHARE MODE**，则直接下发到节点执行；
3. 若查询的列中包含聚集函数，则直接下发节点查询后返回结果；
4. 若查询的列中包含函数，则直接下发节点查询后返回结果；
5. 若语句中包含 **group by**、**distinct**，则无法使用 **memcached**，直接下发节点查询后返回结果；
6. 查询字段中若只包含缓存 **key** 字段集或其子集，则直接下发节点查询后返回结果；
7. 如果语句条件包含缓存 **key** 字段等值关系（**Condition** 只有一层且与其他条件由 **and** 连接且该 SQL 语句不是多表 **join** 的），则先查找 **memcached**，若查找不到则下发节点执行（命中 **memcached** 情况：由语句中获得的缓存 **key** 值查询记录，该记录需进行正确性的判断才能最终决定其是否为用户查询的记录；不命中：则直接带所有条件查询数据库 所以无需判断正确性）；
8. 若均不是上述情况，则生成 SQL 语句：**select id from table where ...**，如果原语句包含 **order by**，则 **sql** 中也必须包含 **order by**，做排序下推，（实际操作是克隆原语法树，将查询字段改为所要操作的缓存 **key** 字段）把语句下发到数据库节点执行，获取缓存 **key** 字段列表；
9. 根据上面获取到的缓存 **key** 字段组成缓存 **key** 值列表，使用 **getMulti** 接口在 **memcached** 中查找记录值，对于不能在 **memcached** 中获取到的数据，生成 SQL 语句到数据库节点查询：**select column from table where id in (v1,v2,v3...)**。使用 **setMulti** 接口，将未命中的记录插入到 **memcached** 中。
10. 把最终结果 **merge** 后返回。

分析是否使用 **memcached** 的流程见下图：



15.3.2 Update、Delete 语句

使用 memcached 的流程：

1. 提取语句相关表，判断表是否指定了缓存 **key** 字段，在 SQL 语句没有明确说明是否使用 **memcached** 的情况下，判断数据表上有没有设置数据缓存到 **memcached**。
2. 如果语句条件包含缓存 **key** 字段等值关系（**Condition** 只有一层且与其他条件由 **and** 连接）若包含则提取缓存 **key** 值；
3. 若不是上述情况，则生成 SQL 语句：**select id from table where ...**，其中查询字段为该表的缓存 **key** 字段，**where** 条件为原语句的条件，把语句下发执行，获取缓存 **key** 字段列表；
4. 根据上面获取到的缓存 **key** 字段组成缓存 **key** 值列表，使用 **deleteMulti** 接口，删除 **memcached** 中对于的记录。
5. 执行原语句。

15.3.3 Insert 语句

使用 memcached 的流程：

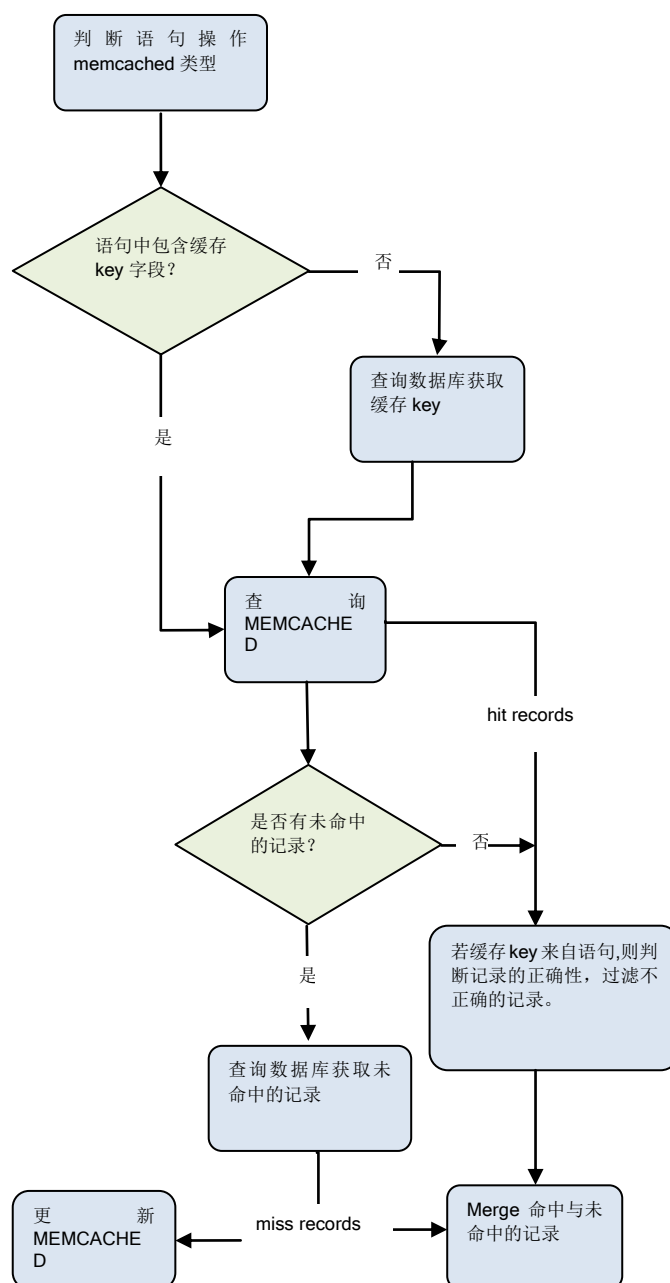
1. 提取语句相关表，判断表是否指定了缓存 key 字段，在 SQL 语句没有明确说明是否使用 memcached 的情况下，判断数据表上有没有设置数据缓存到 memcached。
2. 若语句不为 **replace** 或不包含 **on duplicate key** 则直接下发。
3. 如果语句包含 **on duplicate key**，则判断 **update** 的字段是否为缓存 key 字段集或子集，若不是则先执行原语句（因为该语句对数据库中对应记录的改动不会影响到缓存 key 值的变化，因此执行原语句之后仍可从 memcached 中找到对于的记录），再根据返回的操作数判断是插入新的记录还是变动了原记录。若是插入新记录，则无需从数据库中查询缓存 key 值去更新 memcached。
4. 若均不是上述情况，则生成 SQL 语句：**select ... from table where ...**，其中查询字段为该表的缓存 key 字段，**where** 条件为所插入记录值组成的主键等值关系，把语句下发执行，获取缓存 key 字段列表；
5. 根据上面获取到的缓存 key 字段组成缓存 key 值列表，使用 **deleteMulti** 接口，删除 memcached 中对应的记录。
6. 执行原语句。

15.4 memcached 执行计划

此处通过 memcached 执行计划流程图以及执行计划的 **explain** 展开分析。同时对更新 memcached 的时机进行详细的说明，以及多线程并发情况下 memcached 与数据库的数据不一致性的分析。

15.4.1 Select 语句执行计划

执行计划流程见下图：



◆ 单表（test1.id 为缓存 key 字段）

1. MEM_WITH_UNIQUEKEY

SQL 语句条件包含缓存 key 字段等值关系，可直接从语句中获得缓存 key 值。

例如：

```

isql@dbi>> explain /*memcached = true*/select * from test1 where id = 1 and age = 1;
+-----+
| PLAN                                     |
+-----+
| MEMCACHED-QUERY:                       |

```

```

| MERGE-RECORDS
|   Merge records from memcached or MySQL.
|   /\
|  /||\
|   ||
| SELECT-MISS-RECORDS (If no such Records in memcached, we will get it from
MySQL.)
|   TABLE: test1
|   SQL: SELECT test1.id, test1.name, test1.age, test1.num FROM test1 WHERE
(test1.id = ?) AND (age = 1) AND (id = 1) |
|   /\
|  /||\
|   ||
| QUERY-MEMCACHED
|   Get records from memcached according to memKeys.
|   /\
|  /||\
|   ||
| MEMKEY-FROM-SQL
|   Gets memKeys from SQL.
|   Input memcached key fields from SQL: test1.id = 1
+-----+
21 rows in set, execute time: 6 ms
total time: 43 ms.

```

2. MEMCACHED

SQL 语句条件不包含缓存 **key** 字段等值关系，需先查询数据库获取缓存 **key** 列表。

例如：

```

isql@dbi>> explain /*memcached = true*/ select * from test1 where age = 1;
+-----+
| PLAN
+-----+
| MEMCACHED-QUERY:
|
| MERGE-RECORDS
|   Merge records from memcached or MySQL.
|   /\
|  /||\
|   ||
| SELECT-MISS-RECORDS (If no such Records in memcached, we will get it from
MySQL.)
|   TABLE: test1
|   SQL: SELECT test1.id, test1.name, test1.age, test1.num FROM test1 WHERE
test1.id = ? |
|   /\
|  /||\
|   ||
| QUERY-MEMCACHED
|   Get records from memcached according to memKeys.
|   /\
|  /||\
|   ||
| MEMKEY-FROM-MySQL
|   Gets memKeys from SQL.
|   SQL: SELECT test1.id FROM test1 WHERE (age = 1)

```

```

| SIMPLE-SELECT
|   SQL: SELECT test1.id FROM test1 WHERE (age = 1)
|   Dest Node:
|     dbn2[jdbc:mysql://127.0.0.1:3306/test1]
|     dbn1[jdbc:mysql://127.0.0.1:3306/dbn1]
|   Return records without sorting.
+-----+
27 rows in set, execute time: 21 ms
total time: 38 ms.

```

上面的执行计划分别为一级索引和二级索引，下面说明一下输出信息的含义。

- a) **MEMKEY-FROM-SQL**: 表示语句中包含缓存 **key** 字段等值关系,可从语句中直接获得查询 **memcached** 的 **key** 值。
- b) **MEMKEY-FROM-MySQL**: 当 **a** 不满足时, 需要构造查询缓存 **key** 字段的语法树(实现为克隆原语句的语法树, 将查询字段替换为缓存 **key** 字段), 查询数据库获得缓存 **key** 值。
- c) **QUERY MEMCACHED**: 根据上面获得的缓存 **key** 值查询 **memcached** 获取相应的记录。
(注意: 对于直接从 **SQL** 语句中获得主键而查出的记录, 需判断该记录是否满足原语句的查询条件)
- d) **SELECT-MISS-RECORDS**: 当查询 **memcached** 有为命中的情况, 则需要对未命中的缓存 **key** 生成查询对应记录的语法树, 查询数据库获得记录。
- e) **MERGE-RECORDS**: 对上述命中与未命中的记录进行合并, 变根据原语句中查询的字段通过分析运算, 整合成最终结果。由于整个查询计划有可能运行在非事务状态下, 可能两次数据库查询产生数据不一致, 所以在 **merge** 时还需要考虑不一致情况。

◆ 多表 join (T1、T2 表/*memcached = true*/ T1.id、T2.id 为缓存 **key** 字段)

1. T1:MEMCACHED, T2:NO_MEMCACHED

T1 使用 **memcached** 二级索引, T2 不使用 **memcached**, 语句总体还是使用 **memcached** 的(下面讨论的混用情况均如此)。有两种情况导致 T2 不使用 **memcached**, 一种是 T2 表没有指定缓存 **key** 字段, 另外一种是指定了缓存 **key** 字段, 但在 **SQL** 语句查询字段中 T2 表所涉及的字段只在缓存 **key** 字段集中。

下面为第二种情况例子:

```

isql@dbi>> explain /*memcached = true*/ select T1.age, T2.id from T1, T2 where
T1.id = T2.id;
+-----+
| PLAN
+-----+
| MEMCACHED-QUERY:
|
| MERGE-RECORDS
|   Merge records from memcached or MySQL.
|   /\
|  /||\
|   ||
| SELECT-MISS-RECORDS (If no such Records in memcached, we will get it from
MySQL.)
|   TABLE: T1
|   SQL: SELECT T1.id, T1.name, T1.age, T1.num FROM T1 WHERE T1.id = ?
|   /\
|  /||\
|   ||
| QUERY-MEMCACHED
|   Get records from memcached according to memKeys.
|   /\
|  /||\

```

```

|  ||
| MEMKEY-FROM-MySQL
|   Gets memKeys from SQL.
|   SQL: SELECT T1.id, T2.id FROM T1, T2 WHERE (T1.id = T2.id)
|   PROJECT
|     Project record to: T1.id,T2.id,
|       /\
|     /||\
|     ||
| TOP-INNER-QUERY
|   SQL:SELECT T2.id FROM T2 WHERE ? = T2.id
|   Input columns from outer query:
|   T1.id
|     /\
|   /||\
|   ||
| SIMPLE-SELECT
|   SQL: SELECT T1.id FROM T1
|   Dest Node:
|     dbn2[jdbc:mysql://127.0.0.1:3306/test1]
|     dbn1[jdbc:mysql://127.0.0.1:3306/dbn1]
|   Return records without sorting.
+-----+
39 rows in set, execute time: 13 ms
total time: 35 ms.

```

上面的 explain 中查询 memcached 的缓存 key 字段只有使用 memcached 的 T1 表的缓存 Key 字段。其中不使用 memcached 的表则保留它原有查询字段。

2. T1:MEMCACHED, T2:MEMCACHED

T1 使用 memcached 二级索引, T2 使用 memcached 二级索引

例子:

```

isql@dbi>> explain /*memcached = true*/ select T1.age, T2.age from T1, T2 where
T1.id = T2.id;
+-----+
| PLAN
+-----+
| MEMCACHED-QUERY:
|
| MERGE-RECORDS
|   Merge records from memcached or MySQL.
|   /\
|  /||\
|  ||
| SELECT-MISS-RECORDS (If no such Records in memcached, we will get it from
MySQL.)
|   TABLE: T1
|   SQL: SELECT T1.id, T1.name, T1.age, T1.num FROM T1 WHERE T1.id = ?
|   TABLE: T2
|   SQL: SELECT T2.id, T2.name, T2.age, T2.num FROM T2 WHERE T2.id = ?
|   /\
|  /||\
|  ||
| QUERY-MEMCACHED
|   Get records from memcached according to memKeys.
|   /\
|  /||\

```

```

|  |
|  |
| MEMKEY-FROM-MySQL
|   Gets memKeys from SQL.
|   SQL: SELECT T1.id, T2.id FROM T1, T2 WHERE (T1.id = T2.id)
|   PROJECT
|     Project record to: T1.id,T2.id,
|       /\
|     /\
|     /\
|     /\
|   TOP-INNER-QUERY
|     SQL:SELECT T2.id FROM T2 WHERE ? = T2.id
|     Input columns from outer query:
|     T1.id
|       /\
|     /\
|     /\
|     /\
|   SIMPLE-SELECT
|     SQL: SELECT T1.id FROM T1
|     Dest Node:
|       dbn2[jdbc:mysql://127.0.0.1:3306/test1]
|       dbn1[jdbc:mysql://127.0.0.1:3306/dbn1]
|     Return records without sorting.
+-----+
41 rows in set, execute time: 3 ms
total time: 32 ms.

```

上面的 explain 中查询 memcached 的缓存 key 字段为 T1、T2 表的缓存 key 字段。

- 设置系统参数 `recordCountInBatch` 作为分批查询 memcached 每批的数据量。
 - 设置系统参数 `maxRecordsIntoMem` 限制插入 memcached 的最大数据量，即超过该值之后就不再将其中未命中的记录插入 memcached。（注意：此处的 `maxRecordsIntoMem` 为该语句所涉及的记录数而非未命中 memcached 的记录数。）
- 分批查询的流程（针对二级索引）：

1. 根据 `PSPProducer.getMemKeysPS()` 获得的查询缓存 key 的语法树从数据库中获取主键值。
2. 用 `getNextTuple()` 将主键值存入列表中，数量达到 `recordCountInBatch` 停止。
3. 以表为单位查询 memcached 获取记录，对于未命中的 key，经 `PSPProducer.getRecordsPS()` 获取查询未命中记录的语法树查询数据库获取相应的记录（当所获得的累计缓存 key 数小于系统参数 `maxRecordsIntoMem`，则将这些未命中的记录插入到 memcached 中）。最后将命中与未命中的记录进行 merge。当语句中所涉及的表均获得该批记录后，将所有记录代入原语句查询字段进行运算合并获取最终的结果集。（注：在原语句不开事务的情况下，各个表之间的记录可能与之前从数据库中获取的缓存 key 不对应，因此当一张表的某条记录不存在时得删除其余表相应的那条记录）
4. 用户通过 `MemcachedSelectPlan.getNextTuple()` 获取最终记录，当为空的时候，回到第二步执行。直到所有主键获取完。

● 生成结果集

由上述流程可是 ddb 集成 memcached 的结果集并不是直接从数据库中获得的，而是先获取各个表对应的记录，然后手工整合成用户所需的结果集。这里就涉及到 `DBResultSet` 对象的构造，以及 `DBResultMetaData` 的生成。

下面为查询结果类型转换规则：

Memcached 只支持 `+` `-` `*` `/` 四则运算以及比较运算，以下是各类型经过四则运算之后的结果数

数据类型转化规则:

数据类型分五种:

- a) 浮点型: FLOAT, DOUBLE, REAL
- b) 整型: INTERGER, TINYINT, SMALLINT, BIGINT, BOOLEAN, BIT
- c) DECIMAL, NUMERIC
- d) 非数型: DATE, TIME, TIMESTAMP, CHAR, VARCHAR, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, BLOB, CLOB

各类型经 + - * / 四则运算后的类型:

- 1. a 类型与任何类型进行 + - * / 结果为类型 DOUBLE
- 2. b 类型与 b 类型进行 + - * 结果为类型 BIGINT
- 3. b 类型与 b 类型进行 / 结果为类型 DECIMAL
- 4. d 类型与 d (或 b) 类型进行 + - * / 结果为类型 DECIMAL
- 5. e 类型与任何类型进行 + - * / 直接抛异常

各类型经比较运算后的类型均为 BIGINT。

根据上述规则生成 DBResultMetaData, 在根据 DBResultMetaData 将记录进行计算整合, 最终获得所需的 DBResultSet。

15.4.2 Update、Delete 语句执行计划

1. MEM_WITH_UNIQUEKEY

SQL 语句条件包含缓存 key 字段等值关系, 可直接从语句中获得缓存 key 值。

例如:

```
isql@dbi>> explain /*memcached = true*/ update T1 set name = 'a' where id = 1;
+-----+
| PLAN                                     |
+-----+
| UPDATE                                  |
|   SQL: UPDATE T1 SET name = 'a' WHERE id = 1 |
|   Dest Nodes:                           |
|   jdbc:mysql://127.0.0.1:3306/dbn1      |
|   /\                                     |
|   /||\                                  |
|   ||                                     |
| UPDATE-MEMCACHED                        |
|   delete records in memcached according to memKeys. |
|   /\                                     |
|   /||\                                  |
|   ||                                     |
| MEMKEY-FROM-SQL                          |
|   Gets memKeys from SQL.                 |
|   Input memcached key fields from SQL: T1.id = 1 |
+-----+
15 rows in set, execute time: 12 ms
total time: 19 ms.
```

2. MEMCACHED

SQL 语句条件不包含缓存 key 字段等值关系, 需生成相应的查询缓存 key 字段的语法树, 查询数据库获得缓存 key 值。为了保证获得的缓存 key 值与执行原语句操作数据所涉及的记录一致 (若不一致可能会导致 memcached 中的数据与数据库中的不一致), 所以在这种情况下开启事务。

例如:

```

isql@dbi>> explain /*memcached = true*/ update T1 set name = 'a' where name =
'b';
+-----+
| PLAN                                     |
+-----+
| UPDATE                                  |
|   SQL: UPDATE T1 SET name = 'a' WHERE name = 'b' |
|   Dest Nodes:                           |
|     jdbc:mysql://127.0.0.1:3306/test1    |
|     jdbc:mysql://127.0.0.1:3306/dbn1    |
|     /\                                  |
|     /||\                               |
|     ||                                 |
|   UPDATE-MEMCACHED                      |
|     delete records in memcached according to memKeys. |
|     /\                                  |
|     /||\                               |
|     ||                                 |
|   MEMKEY-FROM-MySQL                     |
|     Gets memKeys from MySQL.             |
|     SQL: SELECT T1.id FROM T1 WHERE name = 'b' |
|     SIMPLE-SELECT                      |
|       SQL: SELECT T1.id FROM T1 WHERE name = 'b' |
|       Dest Node:                        |
|         dbn2[jdbc:mysql://127.0.0.1:3306/test1] |
|         dbn1[jdbc:mysql://127.0.0.1:3306/dbn1] |
|       Return records without sorting.      |
+-----+
22 rows in set, execute time: 1 ms
total time: 13 ms.

```

在查询缓存 **key** 字段的时候，为了防止所查的数据过大导致内存溢出需要开启游标，分批获取数据更新 **memcached**。

15.4.3 Replace、Insert...on duplicate key update...执行计划

Insert 语句只有在 **replace** 以及 **on duplicate key** 的时候才可能会对数据库中的原记录有所改动需，因此需删除 **memcached** 中相应的记录以保证与数据库的数据一致。需要注意的是若该语句在事务中，符合使用 **memcached** 的普通 **Insert** 语句，需要获取语句中所插入记录对应的缓存 **key**，放到事务中一个专门的 **id** 列表中，以便 **select** 时判断是否需要加入缓存。

◆ Replace

例如：

```

isql@dbi>> explain /*memcached = true*/ replace into T1 (id, name, age, num)
values (1, 1, 1, 1);
+-----+
| PLAN                                     |
+-----+
| UPDATE                                  |
|   SQL: /*memcached = true*/ replace into T1 (id, name, age, num) values (1, |
|   1, 1, 1) |
|   Dest Nodes:                           |
|     jdbc:mysql://127.0.0.1:3306/dbn1    |
|     /\                                  |
|     /||\                               |
|     ||                                 |
|   UPDATE-MEMCACHED                      |
|     delete records in memcached according to memKeys. |

```



```

      /\
     /|\
    ||
MEMKEY-FROM-MySQL
  Gets memKeys from MySQL.
  SQL: SELECT T1.id FROM T1 WHERE (T1.id = 1)
      SIMPLE-SELECT
        SQL: SELECT T1.id FROM T1 WHERE (T1.id = 1)
        Dest Node:
          dbn2[jdbc:mysql://127.0.0.1:3306/test1]
          dbn1[jdbc:mysql://127.0.0.1:3306/dbn1]
        Return records without sorting.
+-----+
21 rows in set, execute time: 9 ms
total time: 26 ms.

```

先查询数据库获取与插入记录冲突的记录，更新 **memcached** 的方式与上节 **update** 一样。

◆ On duplicate key

1. Update memKey（与 replace 类似）
2. Update no_memKey
update 的字段不在缓存 key 字段中

```

isql@dbi>> explain /*memcached = true*/ insert into T1 (id, name, age, num) values
(1, 1, 1, 1) on duplicate key update name = 'a';
+-----+
| PLAN
+-----+
| UPDATE-MEMCACHED
|   delete records in memcached according to memKeys.
|   /\
|  /|\
|  ||
MEMKEY-FROM-MySQL
  Gets memKeys from MySQL.
  SQL: SELECT T1.id FROM T1 WHERE (T1.id = 1)
      SIMPLE-SELECT
        SQL: SELECT T1.id FROM T1 WHERE (T1.id = 1)
        Dest Node:
          dbn2[jdbc:mysql://127.0.0.1:3306/test1]
          dbn1[jdbc:mysql://127.0.0.1:3306/dbn1]
        Return records without sorting.
      /\
     /|\
    ||
| UPDATE
|   SQL: /*memcached = true*/ insert into T1 (id, name, age, num) values (1,
1, 1, 1) on duplicate key update name = 'a' |
|   Dest Nodes:
|     jdbc:mysql://127.0.0.1:3306/dbn1
+-----+
21 rows in set, execute time: 1 ms
total time: 26 ms.

```

对于 **on duplicate key update** 的字段不在缓存 **key** 字段中，这先执行原语句，再根据返回的操作数判断是插入新的记录还是变动了原记录。若是插入新记录，则无需从数据库中查询缓存 **key** 值去更新 **memcached**（更新 **memcached** 的方式与上节 **update** 一样）。

15.4.4 分批处理数据

用户更新新数据库时，如果查询条件是非缓存 KEY，则 DDB 会自动通过现有查询条件查找到相应的缓存 KEY，通过 KEY 值删除 memcached 中相应记录。这个过程，有可能因为返回的记录条数过多而导致客户端内存溢出。针对这种情况，DDB 采取分批开游标的方式来避免占用客户端过多内存。

另外，在操作 memcached 缓存时，如果一次更新的记录过多，也有可能导致操作超时。在这里 DDB 也做了分批次处理操作。

15.5 关于缓存数据一致性的讨论

Memcached 缓存能够给系统带来吞吐量和响应时间的极大提升，但话说回来，memcached 和底层数据库毕竟是两套完全隔离的系统。另外，考虑到 memcached 本身是不支持事务特性的，就导致两者之间的数据一致性问题变得非常复杂，常常给外部应用日常使用带来困扰。下面依不同情况分别进行讨论。

15.5.1 目前执行策略

不开启事务的情况下比较简单，查询语句在返回用户最终结果前把 memcached 未命中的记录插入缓存。更新语句在最后更新数据库之前清除 memcached 缓存相关记录。

开启事务的情况下，相对复杂一些。一般情况下，有三种位置可以选择来作为更新 memcached 的时机：

```
memcached 的操作分两种情况。举一个典型客户端 SQL 操作流程：
sql: ... <----- //更新位置 A:在每条 SQL 执行完之后更新 memcached
sql2: ...
...
<----- //更新位置 B:在接收到用户的 commit/rollback 请求后马上更新
memcached
commit/rollback
<----- //更新位置 C:在数据库执行完 commit/rollback 后更新 memcached
```

目前 DDB 在 A 位置进行 memcached 插入操作，在 B 位置进行 memcached 删除操作。另外，DDB 会在事务中保存一个局部队列，用于缓存当前事务中插入、更新、删除的 id 列表。如果查询语句发现 memcached 未命中之后，从数据库获取的记录 id 在这个列表中，则不会去插入 memcached 缓存。这样做的含义就是凡是本次事务操作更新所涉及到的记录，都不会插入缓存。至于为什么这么做的原因，会在下面讨论中给出。

15.5.2 事务隔离级别

事务隔离级别定义为四个等级：

Read Uncommitted: 允许脏读取，但不允许更新丢失。

Read Committed: 允许不可重复读取，但不允许脏读取。

Repeatable Read: 禁止不可重复读取和脏读取，但是有时可能出现幻影数据。

Serializable: 提供严格的事务隔离，它要求事务序列化执行。

目前 DDB 的 memcached 操作策略提供的事务隔离级别为 **Read Committed**。

前面说过，DDB 在执行查询语句时，凡是本次事务操作更新所涉及到的记录，都不会插入缓存。所有插入缓存的记录，都是其他已提交事务所反映的更新。这就保证了 **Read Committed** 事务隔离级别。

目前 DDB 无法保证可重复读的事务隔离级别。当用户的 SQL 语句中直接给出缓存 KEY 的值时，DDB 会从语句中获取 KEY 值，到缓存中进行查找。如果用户第一次查找时能够从缓存

获取到记录，而用户第二次查找时无法从缓存获取记录，则会到数据库中获取记录值。在有并发更新事务，并且另一个事务已提交的情况下，两个记录无法保证一定是相同的。

15.5.3 事务执行原子性

执行原子性是事务的一个基本特性，要求事务中的操作要么全部成功，要么全部失败，不允许出现执行一半的不一致状态。DDB 集成 memcached 后，memcached 和底层数据库就作为一个整体，需要提供事务一致性保证。

前面提到过，DDB 插入到缓存中的数据，都是其他事务已经提交的记录。这样的话，对于插入操作来说，就不需要考虑事务原子性。因为多插入或者少插入都不会影响系统整体数据一致性。

DDB 在删除 memcached 数据记录时，保证是在实际数据库执行 commit 操作之前。实际效果就是只可能在 memcached 上多删除数据，而不会少删除数据。对于一个缓存系统来说，多删除数据并不会影响数据一致性。

综上所述，DDB 在实际操作时，采用的是一种更严格的数据同步方式。既然无法在两个独立系统中做到完全意义上的原子操作，就尽可能多删除一些数据，从而保证了 memcached 与底层的数据一致性。

15.5.4 并发情况下的讨论

前面提到，DDB 在执行完查询语句时，马上把未命中的记录插入缓存，而执行更新语句删除缓存时，尽量延后到事务 commit 之前。这是为了避免在并发情况下产生数据不一致。只要删除缓存的操作在插入缓存操作之后，就能够保证最终 memcached 状态是正确的。

但需要指出的是，在某些极端并发情况下，仍然会产生 memcached 与底层数据不一致。

假设线程 1 的执行顺序如下：

- 1) 用户 SQL 语句要求删除数据库某条记录；
- 2) 从数据库中获取该记录的缓存 KEY；
- 3) 根据 KEY 值删除 memcached 中的相应记录；

线程 2 的执行顺序如下：

- a) 用户 SQL 语句需要查询数据库同一条记录；
- b) 发现缓存中没有该条记录；
- c) 从数据库中获取该记录的值；
- d) 插入到 memcached 缓存中；

如果在(c)和(d)之间，插入了(3)操作，则最终 memcached 中该条记录还是存在的，与实际情况不符，造成了数据不一致。此种情况其实是很难避免的。如果考虑在数据库查询记录值时使用 for update 关键字加写锁，虽然可以避免这种情况，但会带来额外的死锁风险。

另外还有一种情况要说明一下。当用户更新的数据量很大，均保存在内存中会导致内存溢出，所以在查询 memcached key 值的时候开启游标，分批获取记录，当整个事务中待更新的 memcached key 数量大于阈值时就直接在执行计划中去分批删除 memcached 中相应的记录，不会再放到更新 id 列表，也不会再做延后删除行为。由于大批量数据更新在 OLTP 中出现的概率非常低，所以不管是处于性能还是数据一致性考虑，都建议直接更新数据库后做缓存失效操作。(该最大阈值目前定为 100000，用户不可配置)

15.6 存在的限制

并发操作下有可能产生数据不一致，原因已经在上面分析过。这是目前最大的问题，其实也是缓存系统实际使用过程中普遍遇到的问题。DDB 尽量把不一致降到最低。

DDB 使用 `LOADBALANCE` (`slaveonly`、`slaveprefer`、`loadbalance`、`masteronly`) 模式下, 有可能在 `slave` 节点上进行数据查询后, 数据被缓存到 `memcached` 中, 这种过期数据以后有可能被当做 `master` 节点上的正确数据来更新到 `master` 上。这时应用在使用时, 需要自己判断是否需要在适当时候加入缓存 `hint` 语法来控制数据在 `memcached` 中的存取。

DDB 在 `DIRECT FORWARD` 模式下不做语法解析, 因此也不会去操作 `memcached`。

当多台机器组成 `memcached` 集群时, 客户端可以通过一致性 `hash` 算法获取正确的 `memcached` 服务器节点进行操作。但这个算法有一个缺陷, 如果某台机器网络出现短暂间隙, 无法正常连接, 则会选取一致性 `hash` 环上的下一个节点进行操作。如果这时进行了一些 `memcached` 记录删除操作, 之后原来那台机器网络又恢复了, 那么那些删除操作就无法体现在原节点上, 会造成数据不一致。这是一致性 `hash` 本身的问题, DDB 目前还没有解决方案。

DDB 修改表结构后客户端通知不到的情况。考虑以下场景: DBA 在一张表上增加一个字段, `master` 负责把新的表结构通知到所有客户端, 假设客户端 `a` 没有被通知到。以后客户端 `a` 依然会用老版本的表结构来查询数据库, 再把获取到的数据记录存储到 `memcached` 中。这时 `memcached` 中的记录会少一个字段。当其他客户端读取这条 `memcached` 记录时, 如果发现少一个字段, 会自动做兼容性处理, 把该字段的值使用默认值进行填充, 这就可能导致与实际结果不符合。所以, 以后 DDB 集成 `memcached` 后, 如果发现有客户端通知失败的情况, 建议 DBA 手工关闭客户端 `a`。这个事情确实比较麻烦, 不过预计 DDB4 加入客户端有效性定期检查后, 通知客户端失败的情况会大大减少。

16 支持 Oracle 数据库引擎

DDB 分布式数据库自从开发以来, 一直针对的 `MySQL` 存储引擎。`MySQL` 数据库具有开源、配置简单等优点, 因此, 基于 `MySQL` 的 DDB 已经成功的应用于网易博客项目中。到目前为止, 较好的满足了博客目前的应用需求。但是, 相比较而言, `Oracle` 有着更成熟的系统解决方案及更高的性能, 尤其对于需要高可靠性的应用环境来说, `Oracle` 就显得更有吸引力。基于上述考虑, DDB4.0 版本中集成了 `Oracle` 存储引擎。

DDB 项目的设计与开发, 是围绕着 `MySQL` 来进行的, 在博客系统上线之后, 更是针对博客的特定需求, 在 DDB 上增加了很多额外的功能。集成 `Oracle` 存储引擎的目标, 是希望在多个底层单独 `Oracle` 数据库的基础上, 对外提供一个统一的 DDB 接口, 方便应用程序开发。在 `Oracle` 数据库基础上开发分布式数据库, 可以有很多种实现方式, 但是考虑到现在 DDB 系统已经在网易的很多已有产品上部署使用, 所以在集成过程中, 尽量保持了对外的接口及使用方法不变。

分布式数据库集群, 大致可分为三类: 高性能计算集群、负载均衡集群 (LB) 和高可用性集群 (HA)。`Oracle` 企业版本本身提供了一种分布式解决方案 `Oracle Real Application Clusters` (`RAC`), `RAC` 的侧重点是 LB 和 HA, `RAC` 解决方案中没有单点故障失效 (`one point failure`) 问题, 能够很快的从一台机器的失效中恢复过来, 因此具有很高的可靠性。但是 `RAC` 数据库系统中, 数据并不是严格的分区到单独的节点上, 节点之间的数据交互非常频繁, 使得 `RAC` 并不适合使用在超大规模数据量及高并发的环境下。而 DDB 的解决方案, 很好的考虑到了 Web2.0 应用中数据分布的特点及要求高并发、高性能的特点。综上所述, DDB 集成 `Oracle` 存储引擎, 不同于 `RAC` 的特点, 而是为了能够提供更高的整体性能及快速响应时间。

16.1 搭建基于 Oracle 的 DDB

流程和标准 DDB 搭建流程大致相同, 可按下面步骤进行:

1. 创建 DDB 用户。通过 DDB 管理工具 DBAdmin 来完成，设置方式与标准方式完全相同。
2. 初始化 Oracle 数据库节点。这个过程是标准流程没有的，使用 `conf/node_oracle_init.sql` 来完成。该脚本主要完成的工作：创建供 DDB 系统使用的超级用户（`ddbsys`），该用户具有 `dba` 权限但是不具备 `ADMINISTER DATABASE TRIGGER` 权限；创建 DDB 系统专用 `shema`（`ddbsys`）；创建 DDB 系统专用的域角色（`DDB_DOMAIN_*`）；创建 DDB 系统专用的全局角色（`DDB_GLOBAL_*`）；创建 DDB 系统所使用的 IP 认证表（`USER_AUTHORIZATION`）；创建登陆触发器 `TRG_LOGON` 用于 IP 限制。
3. 在 DDB 系统中添加 Oracle 数据库节点。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的节点类型必须是 Oracle 类型。
4. 在 DDB 系统中添加均衡策略。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的策略类型必须是 Oracle 类型。
5. 在 DDB 系统中添加表。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的表类型必须是 Oracle 类型。

16.2 用户管理

16.2.1 用户 IP 限制

在 DDB 管理工具中增加用户时，可以设定新增用户的 IP 地址。只有在 IP 地址列表中的客户端使用该用户登陆时才被允许。

IP 限制这个功能，是 DDB 基于 MySQL 开发时提出的功能，而 Oracle 数据库本身不支持客户端登录的 IP 判断。为了实现数据库端的 IP 限制，DDB 在 Oracle 节点上部署了一套辅助系统：

- 创建 DDB 系统所使用的 IP 认证表，`USER_AUTHORIZATION`。其中保存了每个用户对应的用户名及允许访问的 IP 地址。
- DDB 管理工具在修改用户 IP 时，同时更新 `USER_AUTHORIZATION` 表。
- 创建登陆触发器 `TRG_LOGON` 用于 IP 限制。每个用户登陆时，都判断其登陆 IP 是否在 IP 认证表中。由于 DDB 在访问 `oracle` 节点时，其实 DBA 所使用的用户都是 `ddbsys`，因此，必须保证 `ddbsys` 用户必须不具备 `ADMINISTER DATABASE TRIGGER` 权限，因为此类权限会限制登陆触发器正常执行。

16.2.2 用户表模式（schema）共享

DDB 系统中的用户表模式概念与 Oracle 数据库有本质不同。DDB 中可以创建多个用户，但所有用户所见到的都是同一个表模式，即 DDB 实现的是单 `schema`，而不是每个用户对应一个 `schema`。而 Oracle 数据库是使用每个用户独立对应一个表模式的概念。

为了把 Oracle 数据库的表模式模型转换到 DDB 中，我们做了如下变通。首先在节点初始化脚本中创建 DDB 专用 `schema`：DDBSYS。每个通过 DDB 登录的用户，都会在创建数据库连接时自动执行命令：`ALTER SESSION SET CURRENT_SCHEMA = DDBSYS`。而在 DBA 管理工具中创建表时，所有的表都是建立在 DDBSYS 表模式空间中。

16.2.3 数据库对象权限

DDB 系统中支持的用户数据库对象权限有三种。

单张数据表上的权限

这种权限类型比较简单，只要在 DDB 创建用户时，同时也在数据库上把该张表的权限也赋给用户即可。

域用户权限

域用户权限能够对 DDB 中的所有表进行操作，对应权限列表中的*。其实域用户的意思就是能够对 DDBSYS 表模式下的所有表进行操作。这类用户有专门的一组 DDB 用户角色来与之对应，在 DDB 增加或删除表时，都会去更新这组用户角色的相应表权限。

全局用户权限

全局用户权限能够对 Oracle 数据库中的所有表进行操作，对应权限列表中的*.*。这类用户有专门的一组 DDB 用户角色来与之对应。

16.3 数据库对象管理

16.3.1 管理数据库节点

在执行完数据库节点初始化脚本之后，可以使用管理工具来新增节点。新增节点时必须指定的参数有：

- 数据库节点名称
- IP 地址
- 端口号
- jdbc 连接 URL
- 状态测试语句。

另外有些参数可以选填。

- **Schema**：一般不需要修改，就是用系统默认的 DDBSYS 即可。
- **表空间**：新增节点时可以指定这个节点默认的 DDB 建表表空间。指定之后，在该节点创建用户时，会自动修改用户的默认表空间为所指定的表空间。新增节点时也可以先不指定表空间，而在新建数据表时去指定表空间。如果在建立数据表时也没有指定表空间，则使用 Oracle 数据库的默认系统表空间。

还有些参数不需要填。

Ssh 用户名、ssh 端口、负载均衡权重。这些参数都是为 MySQL 节点提供的。

16.3.2 管理均衡策略

与标准模式相同，不再赘述。如果选定了所创建的均衡策略类型为 Oracle 数据库，则在选择该策略所对应的数据库节点时，就只能选择 Oracle 数据库节点。

16.3.3 管理数据库表

如果选定了所创建的表类型为 Oracle 数据库，则在选择均衡策略时，就只能选择 Oracle 数据库类型的策略。

支持的字段类型

DDB 支持了 Oracle 数据库所有的常见字段类型。

数据类型	对应的 jdbc 类型
------	-------------

CHAR	Types.CHAR
NCHAR	Types.CHAR
VARCHAR2	Types.VARCHAR
NVARCHAR2	Types.VARCHAR
RAW	Types.VARCHAR
NUMERIC	Types.NUMERIC
DECIMAL	Types.DECIMAL
INTEGER	Types.INTEGER
SMALLINT	Types.SMALLINT
FLOAT	Types.FLOAT
DOUBLE PRECISION	Types.DOUBLE
REAL	Types.REAL
NUMBER	Types.CHAR
DATE	Types.DATE
TIMESTAMP	Types.TIMESTAMP
BLOB	Types.BLOB
CLOB	Types.CLOB
NCLOB	Types.CLOB
BINARY_FLOAT	Types.FLOAT
BINARY_DOUBLE	Types.DOUBLE

DDB 中有一个问题是均衡字段类型。均衡字段只允许是整数或字符串类型，不能是浮点数类型。这样，DDB 中集成 Oracle 时，就不能只是提供 NUMBER 类型，因为 NUMBER 类型无法判别该字段是整型还是浮点数。因此，如果是均衡字段，则该字段的类型是数字时，只能使用 INTEGER。INTEGER 在 Oracle 存储时，虽然使用 NUMBER 来保存，但是如果插入浮点数，则自动会把小数值截断。

表模式支持度说明

支持聚簇表

DDB 支持 Oracle 中的索引聚簇表和散列聚簇表。如果需要创建聚簇表，可以首先在“查看表—> 聚簇”页面中创建聚簇，然后在创建表时选择相应的已存在聚簇。

支持触发器和存储过程

DDB 支持创建 Oracle 触发器和存储过程。如果需要在 DDB 中调用存储过程，可以首先在管理工具中增加触发器，然后在 isql 工具或直接通过 JDBC 的“call”接口调用。

支持对数据表分区

DDB 中支持对 Oracle 的数据表进行分区。可以通过“查看表—> 分区”进行操作。功能包括创建分区、查看分区、删除分区、使用计划任务自动添加分区。

特殊索引类型

DDB 中支持 Oracle 的一些特殊索引类型。包括函数索引、降序索引、B 树索引、hash 索引、B 树索引压缩、位图索引。可以在 DDB 管理工具中直接输入对应的关键字，其实 DDB 中并不记录此类信息，而是把操作语句直接下发给 Oracle 数据库。

支持虚拟字段

在 DDB 管理工具中建表时，可以把某一字段设置成虚拟字段。

支持的数据表类型

目前 DDB 支持的数据表类型包括：堆表、索引组织表、索引聚簇表、散列聚簇表、有序散列聚簇表。

语法及限制

创建表

```
CREATE TABLE tbl_name (create_definition,...) [table_options] [TABLESPACE
tablespace_name];
create_definition:
    column_definition|virtual_column_definition,...
    |outline_constraint

column_definition:
column_name datatype[(length[,...])] [inline_constraint]

virtual_column_definition:
column_name datatype[(length[,...])] [GENERATED ALWAYS] AS (column_expression)
[VIRTUAL] [inline_constraint]

inline_constraint:
    |[NOT] NULL
    |UNIQUE [USING INDEX...]
    |PRIMARY KEY [USING INDEX...]

outline_constraint:
    |UNIQUE(column_name,...) [USING INDEX...]
    |PRIMARY KEY(column_name,...) [USING INDEX...]

table_options:
    |ORGANIZATION HEAP|INDEX
    |CLUSTER cluster_name(column_name,...)
```

限制规则：

- 1、聚簇表不能再指定表空间：在 `table_options` 检测到'CLUSTER'和'TABLESPACE'同时出现，则抛 `SQLException`；
- 2、索引组织表不能添加聚簇选项：在 `table_options` 检测到'ORGANIZATION INDEX'和'CLUSTER'同时出现，则抛 `SQLException`；
- 3、索引组织表必须有主键：在 `table_options` 检测到'ORGANIZATION INDEX'，但前面并没有提取到'PRIMARY KEY'，则抛 `SQLException`；
- 4、暂不支持临时表：如果在 `TABLE` 前检测到'GLOBAL TEMPORARY'，则抛 `SQLException`；
- 5、暂不支持外部组织表：如果在 `ORGANIZATION` 后检测到的是 `EXTERNAL`，则抛出 `SQLException`；
- 6、'USING INDEX'不允许创建索引：在 `column_definition` 里检测到'USING INDEX'，如果紧接着的是'('，则抛 `SQLException`；

聚簇相关

```
CREATE CLUSTER cluster_name (column_name datatype [SORT], ..) [cluster options];
cluster options:
    | SIZE integer [K|M|G|T|P|E]
    | TABLESPACE tablespace_name
    | INDEX
    | [SINGLE TABLE] HASHKEYS integer [HASH IS expr]

--IF: 索引聚簇
CREATE INDEX index_name ON CLUSTER cluster_name [cluster_index_options];
cluster_index_options:
    | REVERSE
    | TABLESPACE tablespace_name

--删除聚簇
DROP CLUSTER name [INCLUDING TABLES];

--修改聚簇, 只支持修改 SIZE
ALTER CLUSTER name SIZE integer [K|M|G|T|P|E]
```

限制规则:

- 1、聚簇索引不能为 BITMAP 或 UNIQUE: 'INDEX'前包含'BITMAP'|'UNIQUE', 则不可能是聚簇索引

创建索引

```
CREATE [UNIQUE|BITMAP] INDEX index_name ON table_name (index_expr [ASC|DESC], ..)
[index_options];
index_options:
    | REVERSE
    | TABLESPACE tablespace_name
    | COMPRESS integer | NOCOMPRESS
```

限制规则:

- 1、位图索引不能压缩: 如果在'INDEX'前检测到'BITMAP', 且 index_options 里有 'COMPRESS', 则抛 SQLException;
- 2、位图索引不能 REVERSE: 如果在'INDEX'前检测到'BITMAP', 且 index_options 里有 'REVERSE', 则抛 SQLException;

表字段增加、修改、删除

```
ALTER TABLE tbl_name ADD (column_definition | virtual_column_definition,...);

ALTER TABLE tbl_name MODIFY (column_definition | virtual_column_definition,...);

ALTER TABLE tbl_name DROP (column_name[,column_name...]);
ALTER TABLE tbl_name DROP COLUMN column_name;

column_definition:
column_name datatype[(length[,...])] [inline_constraint]

virtual_column_definition:
column_name datatype[(length[,...])] [GENERATED ALWAYS] AS (column_expression)
[VIRTUAL] [inline_constraint]
```

限制规则：

- 1、字段的 **ADD,MODIFY** 操作可以混合或重复出现；
- 2、字段的 **DROP** 操作不能重复，也不能与其他操作联合；
- 3、主键、唯一键的 **ADD,DROP** 可以重复，也可以混合；
- 4、字段的 **ADD,MODIFY** 操作和主键、唯一键的 **ADD,DROP** 操作可以联合；
- 5、字段重命名、表重名不能重复，也不能与其他操作联合；

字段、表的重命名

```
ALTER TABLE tbl_name RENAME TO new_name;
ALTER TABLE tbl_name RENAME COLUMN old_name TO new_name;
```

分区的增加、删除

```
ALTER TABLE tbl_name ADD PARTITION...;
ALTER TABLE tbl_name DROP PARTITION...;
```

表限制的增加、删除

```
ALTER TABLE tbl_name ADD PRIMARY KEY(column_name,...);
ALTER TABLE tbl_name ADD UNIQUE(column_name,...);
ALTER TABLE tbl_name DROP PRIMARY KEY;
ALTER TABLE tbl_name DROP UNIQUE(column_name,...);
```

注释

```
COMMENT ON TABLE tbl_name IS 'string'
COMMENT ON COLUMN tbl_name.column_name IS 'string'
```

16.4 SQL 语法说明

DBI 端默认只加载 MySQL 的 JDBC 驱动，如果需要连接 Oracle，则同时需要加载 Oracle 驱动。通过设置 `jvm` 参数来控制：`-Dloadoracle=true`

Oracle 数据库和 MySQL 数据库的 SQL 语法，是有一些微小差异的。DDB 在集成 Oracle 数据库时，遵循的规则是：提供给用户的接口采用原来 MySQL 的形式，在实际 Oracle 节点上执行前，DDB 自动把语法转换到 Oracle 语法。

正则表达式语法

对于正则表达式语法不同，实际 DDB 对外提供的接口使用统一的形式：

```
SELECT * from table WHERE column REGEXP '^S'。
```

实际执行时，对于 Oracle 的类型，转换成：

```
SELECT * from table WHERE REGEXP_LIKE(column, '^S')
```

MERGE INTO 语法

在 MySQL 中，对于有唯一性索引限制的表，插入记录有两种方式：

```
REPLACE [INTO] tbl_name [(col_name,...)] VALUES ({expr | DEFAULT},...), (...), ...
INSERT ... on duplicate key update ...
```

Oracle 中没有 `replace` 和 `on duplicate key update` 的语法，而是相应增加了一种 `merge` 类型的 SQL 语句：

```
MERGE INTO table_name
USING (table|view|sub_query) ON (join condition)
WHEN MATCHED THEN
UPDATE SET col1=val1,col2....
WHEN NOT MATCHED THEN
INSERT (column_list) VALUES (column_values);
```

DDB 在实际下发给 Oracle 执行时，把原 SQL 语句改成 `MERGE INTO` 的语法。

数据表连接方式

Oracle 支持下面的三种表连接方式：

```
SELECT /*+ use_nl(t1 t2) */ * FROM t1, t2 WHERE ...
SELECT /*+ use_hash(t1 t2) */ * FROM t1, t2 WHERE ...
SELECT /*+ use_merge(t1 t2) */ * FROM t1, t2 WHERE ...
```

DDB 的 SQL 接口规定，表连接方式的三种 `hint`，只能写在 SQL 语句的开头。在实际执行时把 `hint` 还原到 SQL 语句中。因此 DDB 对表连接支持的语法如下：

```
/*+ use_nl(t1 t2) */ SELECT * FROM t1, t2 WHERE ...
/*+ use_hash(t1 t2) */ SELECT * FROM t1, t2 WHERE ...
/*+ use_merge(t1 t2) */ SELECT * FROM t1, t2 WHERE ...
```

Limit-Offset 语法

Oracle 中的 `limit-offset` 语法如下：

```
SELECT * FROM
(SELECT rownum r , * FROM table WHERE rownum < 20) t2 WHERE t2.r >= 10;
```

相比起来，还是 MySQL 中的语法简洁明了，而且为了接口统一起见，决定 DDB 只提供原先的 `limit-offset` 方式。在实际语句执行时，把 `limit-offset` 转换成 Oracle 语法。

WHERE 条件中的真伪判断

DDB 中的语法为:

```
SELECT * FROM table WHERE true
SELECT * FROM table WHERE false
```

实际执行时转换成 Oracle 语法:

```
SELECT * FROM table WHERE 1 = 1
SELECT * FROM table WHERE 1 <> 1
```

日期时间格式

DDB 在每次链接到 Oracle 数据库时, 会自动执行设置默认时间字符串格式的 SQL 语句:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'yyyy-mm-dd hh24:mi:ss';
ALTER SESSION SET NLS_TIMESTAMP_FORMAT = 'yyyy-mm-dd hh24:mi:ss';
```

因此, 请用户也根据此种格式操作日期时间字段类型。

16.5 其他限制

目前 DDB 移植到 Oracle 数据库, 并没有实现 DDB 原有的全部功能, 除了上面章节提到的一些限制之外, 其他的主要限制还有:

- 不支持视图。
- 不支持 DDB 自带的数据迁移。
- 不支持统计任务中的数据库节点统计。
- 不支持在线修改表结构功能。
- 不支持数据导入导出功能。
- 不支持 MySQL 数据库的一些特有语法, 如 **lock in share mode**。所有 MySQL 中的特殊语法, 能够自动转换到 Oracle 的都已经列在 SQL 语法说明一节中列出。

17 支持神通数据库引擎

神通数据库是由天津神舟通用数据技术有限公司开发的一款拥有完全自主知识产权国产商用数据库, 目前在航天等多个国内领域中得到广泛应用。通过 DDB 的支持, 神通数据库能够实现数据库节点水平线性扩展, 从而能够在大规模数据场景以及高性能并发访问领域提供解决方案。DDB 对神通数据库提供了日常基本应用支持, 能够通过 DDB 进行统一的节点管理、表模式管理, 通过 DDB 提供的中间件客户端, 能够对分散在各节点中的数据进行查询汇总。DDB 的分析统计功能及神通数据库的一些独特特性, 目前版本中还没有支持。

17.1 搭建基于神通数据库的 DDB

流程和标准 DDB 搭建流程大致相同, 可按下面步骤进行:

1. 创建 DDB 用户。通过 DDB 管理工具 DBAdmin 来完成, 设置方式与标准方式完全相同。
2. 初始化神通数据库节点。这个过程是标准流程没有的, 使用 `conf/node_oscar_init.sql`

来完成。该脚本主要完成的工作：创建供 DDB 系统使用的超级用户（`ddbsys`）；创建 DDB 系统专用 `shema(ddbsys)`；创建 DDB 系统专用的域角色（`DDB_DOMAIN_*`）；创建 DDB 系统所使用的几个存储过程（`UPDATE_DDB_USER_ROLES_PRIVILEGES`、`CREATE_DDB_SYS_USER`、`CREATE_DDB_USER_ROLES`、）。

3. 在 DDB 系统中添加神通数据库节点。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的节点类型必须是 **Oscar** 类型。
4. 在 DDB 系统中添加均衡策略。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的策略类型必须是 **Oscar** 类型。
5. 在 DDB 系统中添加表。通过 DDB 管理工具 DBAdmin 来完成，需要注意增加的表类型必须是 **Oscar** 类型。

17.2 用户管理

用户管理在操作方式和用户界面上，与标准用户管理完全相同，不需要增加神通数据库专用用户。但在一些实现细节上会有差异，需要说明一下。

17.2.1 用户 IP 限制

在 DDB 管理工具中增加用户时，可以设定新增用户的 IP 地址。只有在 IP 地址列表中的客户端使用该用户登陆时才被允许。

IP 限制这个功能，是 DDB 基于 MySQL 开发时提出的功能，而神通数据库本身不支持客户端登录的 IP 判断。因此，DDB 在整合神通数据库时并没有基于神通数据库节点去实现这个功能，而是在客户端登陆管理服务器（**master**）时进行了判断，如果所使用的用户的 IP 地址与实际登陆客户端的 IP 地址不符，则不允许登陆。

17.2.2 用户表模式（schema）共享

DDB 系统中的用户表模式概念与神通数据库有本质不同。DDB 中可以创建多个用户，但所有用户所见到的都是同一个表模式，即 DDB 实现的是单 **schema**，而不是每个用户对应一个 **schema**。而神通数据库是使用每个用户独立对应一个表模式的概念。

为了把神通数据库的表模式模型转换到 DDB 中，我们做了如下变通。首先在节点初始化脚本中创建 DDB 专用 **schema: DDBSYS**。每个通过 DDB 登录的用户，都会在创建数据库连接时自动执行命令：**SET SEARCH_PATH=DDBSYS**。而在 DBA 管理工具中创建表时，所有的表都是建立在 DDBSYS 表模式空间中。

17.2.3 数据库对象权限

DDB 系统中支持的用户数据库对象权限有两种。

单张数据表上的权限

这种权限类型比较简单，只要在 DDB 创建用户时，同时也在数据库上把该张表的权限也赋给用户即可。

全局用户权限

全局用户权限能够对 DDB 中的所有表进行操作，对应权限列表中的 ***.*** 或者 *****。这两种用户只在 MySQL 版本的 DDB 中有区别，在神通版本的 DDB 中是一样的。其实全局用户的意思

就是能够对 DDBSYS 表模式下的所有表进行操作。这类用户有专门的一组 DDB 用户角色来与之对应，在 DDB 增加或删除表时，都会去更新这组用户角色的相应表权限。

17.3 数据库对象管理

17.3.1 管理数据库节点

在执行完数据库节点初始化脚本之后，可以使用管理工具来新增节点。新增节点时必须指定的参数有：

- 数据库节点名称
- IP 地址
- 端口号
- jdbc 连接 URL
- 状态测试语句。

另外有些参数可以选填。

- **Schema:** 一般不需要修改，就是用系统默认的 DDBSYS 即可。
- **表空间:** 新增节点时可以指定这个节点默认的 DDB 建表表空间。指定之后，在该节点创建用户时，会自动修改用户的默认表空间为所指定的表空间。新增节点时也可以先不指定表空间，而在新建数据表时去指定表空间。如果在建立数据表时也没有指定表空间，则使用神通数据库的默认系统表空间。

还有些参数不需要填。

- **Ssh 用户名、ssh 端口、负载均衡权重。**这些参数都是为 MySQL 节点提供的。

17.3.2 管理均衡策略

与标准模式相同，不再赘述。如果选定了所创建的均衡策略类型为神通数据库，则在选择该策略所对应的数据库节点时，就只能选择神通数据库节点。

17.3.3 管理数据库表

与标准模式相同，不再赘述。如果选定了所创建的表类型为神通数据库，则在选择均衡策略时，就只能选择神通数据库类型的策略。

支持的字段类型

DDB 支持了神通数据库所有的常见字段类型。

数据类型	对应的 jdbc 类型
CHAR	Types.CHAR
VARCHAR	Types.VARCHAR
TEXT	Types.VARCHAR
BIT	Types.BIT
TINYINT	Types.TINYINT
SMALLINT	Types.SMALLINT
INT	Types.INTEGER
BIGINT	Types.BIGINT
DECIMAL	Types.DECIMAL
NUMERIC	Types.NUMERIC
REAL	Types.REAL
DOUBLE PRECISION	Types.DOUBLE

FLOAT	Types.FLOAT
BINARY	Types.BINARY
VARBINARY	Types.VARBINARY
DATE	Types.DATE
BOOLEAN	Types.BOOLEAN
BLOB	Types.BLOB
CLOB	Types.CLOB
TIME	Types.TIME
TIMESTAMP	Types.TIMESTAMP

`interval` 类型、数组类型在当前版本中都没有支持

DDB 中创建神通数据表

```
CREATE TABLE tbl_name (create_definition,...) [table_options] [TABLESPACE
tablespace_name];
create_definition:
    column_definition,...
    |outline_constraint

column_definition:
column_name datatype[(length[,...])] [inline_constraint]

inline_constraint:
    |UNIQUE USING INDEX index_name ...
    |PRIMARY KEY [USING INDEX...]

outline_constraint:
    |UNIQUE(column_name,...) USING INDEX index_name ...
    |PRIMARY KEY(column_name,...) [USING INDEX...]
```

表字段增加、删除、修改

```
ALTER TABLE tbl_name ADD [COLUMN] { column_definition } |
{ (column_definition ,...) };

column_definition:
column_name datatype[(length[,...])] [inline_constraint]

inline_constraint:
    |UNIQUE USING INDEX index_name ...
    |PRIMARY KEY [USING INDEX index_name...]

ALTER TABLE tbl DROP [COLUMN] column_name [RESTRICT | CASCADE] --这个操作会导致
有关联的索引都被删除

ALTER TABLE tbl ALTER [COLUMN] TYPE column_name datatype[(length[,...])]

ALTER TABLE tbl ALTER [COLUMN] column { SET DEFAULT constant_expression | DROP
DEFAULT }

ALTER TABLE tbl ALTER [COLUMN] column { SET | DROP } NOT NULL
```

主键等约束的增加、删除

```
ALTER TABLE tbl ADD [CONSTRAINT constraint_name] PRIMARY KEY (column_name[,...])
[USING INDEX index_name] ...

ALTER TABLE tbl ADD [CONSTRAINT constraint_name] UNIQUE (column_name[,...]) USING
INDEX index_name...

ALTER TABLE tbl DROP CONSTRAINT constraint_name|idx_name

ALTER TABLE tbl DROP PRIMARY KEY
```

表和字段的重命名

```
ALTER TABLE tbl RENAME TO new table

ALTER TABLE tbl RENAME [COLUMN] column TO new_column
```

索引

```
CREATE [UNIQUE] INDEX idx_name ON tbl_name [USING BTREE|BITMAP] ( column_name
[ASC|DESC] [,...]) [index_options] ...

CREATE [UNIQUE] INDEX idx_name ON tbl_name [USING BTREE|BITMAP] FROM { idx_name
| idx_name, .. } [index_options] ...

index_options:
    | REVERSE
    | TABLESPACE tablespace_name
    | COMPRESS integer | NOCOMPRESS

DROP INDEX idx_name [RESTRICT | CASCADE]
```

表管理的限制

目前通过 DDB 管理神通数据库表时，有两个限制。

限制 1:

建表时的唯一索引，必须通过 **USING INDEX** 指定索引名。

限制 2:

在整个 DDB 中，不能存在同名的索引，即使是分属不同的数据表也不行。

17.4 SQL 语法说明

17.4.1 与标准 DDB 语法的差异

DDB 的 DML-SQL 语句语法，是参照 MySQL 数据库设计的。每种数据库引擎，在 SQL 语法上都会有些不同，并且都会提供些自己数据库的特性。DDB 在实现 SQL 语句上，很大程度依赖于底层的数据节点。因此，在 DDB 移植到神通数据库时，会产生一些细小的差异，罗列如下：

- 不支持 **insert** 多值。语法为 **insert into ... values (...),(...),(....)**。这种语法是 **MySQL** 特有的，其他数据库均不支持。
- 不支持正则表达式。目前 **DDB** 中的 **Rlike**、**REGEXP** 关键字都不支持。
- 没有 **on duplicate key** 和 **replace into**。如果在往数据表中插入数据时发生主键冲突的情况，必须由应用程序解决。
- 不支持 **LOCK IN SHARE MODE**。目前只支持 **SELECT FOR UPDATE**，此种语句对表记录施加的是排它锁，读共享锁目前不支持。
- 不支持 **FORCE INDEX**。不能对某条 **SQL** 语句强制指定使用某个索引，这样，**SQL** 语句的实际执行情况，将完全依赖于数据库节点所生成的执行计划。

17.4.2 大对象操作

神通数据库在操作 **Blob** 类型的字段时，有一种特有的处理方法：

```
Blob blob = ( (OscarJdbc2Connection) con).createBlob();
```

而在 **DDB** 中，所有的数据库真实连接，对于用户都是不可见的，使用 **DDB** 驱动所创建的数据库连接，经过了 **DDB** 的封装。这样的话，用户就无法拿到真正的神通数据库 **JDBC** 连接，从而无法创建 **Blob** 对象。因此，**DDB** 提供了其他标准方法来操作大对象。

A. 通过 **setBinaryStream** 方式来赋值。

```
pstmt = con.prepareStatement("insert into testblog values(1,?)");
File file = new File("d:\\1.txt");
FileInputStream fis = new FileInputStream(file);
pstmt.setBinaryStream(1, fis, (int)file.length());
pstmt.executeUpdate();
```

B. 通过 **setBytes** 方法来赋值。

```
pstmt = con.prepareStatement("insert into testblog values(1,?)");
pstmt.setBytes(1,...);
pstmt.executeUpdate();
```

C. 读取 **Blob** 数据，有三种方式：

通过 **ResultSet** 的 **getBlob** 方法。

通过 **ResultSet** 的 **getBytes** 方法。

通过 **ResultSet** 的 **getBinaryStream**。

17.4.3 两阶段提交

两阶段提交在神通数据库上有点特殊，需要单独讲一下。

OscarXA 包

神通数据库的 **jdbc** 接口，本身没有提供对两阶段提交的支持，为此，**DDB** 单独开发了一个专为神通数据库提供的 **JDBC** 包（**OscarXa.jar**）。**OscarXa** 封装了普通的 **Oscar** 数据库 **JDBC** 连接，使之看上去能够提供标准 **XA** 接口。

OscarXA 包含有下面的 **java** 类：

- **OscarConnectionWrapper**。继承自标准 **JDBC-Connection** 接口，用于封装普通的 **Oscar-JDBC** 连接。调用 **close** 方法时，实际上不是真正去关闭数据库连接。真正关闭数据库连接必须由 **OscarXAConnection** 发出。
- **OscarXAConnection**。集成了标准的 **XAConnection**, **XAResource** 接口。
- **OscarXADataSource**。用于创建 **OscarXAConnection**。
- **OscarXAException**。异常封装。
- **OscarXid**。定义两阶段提交协议中专用的 **Xid**。

如果要实现两阶段提交，**OscarXAConnection** 的调用流程为：

- A. 调用 **start** 方法，开启两阶段事务。
- B. 调用 **end** 方法，结束两阶段事务。
- C. 调用 **prepare** 方法，实际提交两阶段事务。
- D. 调用 **commit** 方法，一般的两阶段提交不做任何操作。如果是两阶段提交中的 **commit-one-phase** 情况，则需要在该方法中提交事务。

局限性

真正的两阶段提交，必须由数据库引擎来提供支持，数据库必须完全的实现 **prepare** 和 **commit** 两个方法。目前由 **DDB** 上层提供的两阶段提交机制，在某些情况下会造成数据不一致。这里面主要的问题出在 **prepare** 方法里实际调用了数据库 **commit** 命令。假设下面的情况：对数据库节点 **A**、**B**、**C** 发出两阶段提交命令，在 **prepare** 阶段，**A**、**B** 都正常完成，而 **C** 节点抛出异常，此时由于 **A**、**B** 上都已经把事务完全提交掉，因此无法再次 **rollback**。最后的结果是 **C** 节点数据没有正常更新，而 **A**、**B** 节点的更新事务都已提交，造成事务状态不一致。

17.5 其他限制

目前 **DDB** 移植到神通数据库，并没有实现 **DDB** 原有的全部功能，除了上面章节提到的一些限制之外，其他的主要限制还有：

- 不支持视图。
- 不支持存储过程和触发器。
- 不支持 **DDB** 自带的数据迁移。
- 不支持统计任务中的数据库节点统计。
- 不支持在交互式 **SQL** 工具 (**isql**) 中新增神通数据库节点、建立神通均衡策略、创建修改神通数据库表。
- 不支持在线修改表结构功能。
- 不支持数据导入导出功能。

18 DDB 访问异常速查

18.1 ResultSet 超时异常

异常信息:

com.mysql.jdbc.CommunicationsException: Communications link failure due to underlying exception:

** BEGIN NESTED EXCEPTION **

java.io.EOFException

MESSAGE: Can not read response from server. Expected to read 4 bytes, read 0 bytes before connection was unexpectedly lost.

STACKTRACE:

java.io.EOFException: Can not read response from server. Expected to read 4 bytes, read 0 bytes before connection was unexpectedly lost.

Last packet sent to the server was 2467890 ms ago.

at com.mysql.jdbc.MysqlIO.reuseAndReadPacket(MysqlIO.java:2622)

...

产生原因: DDB 执行 `select` 语句时获得多个 dbn 上的 `ResultSet`, 如果没有排序操作, 会从一个 dbn 的 `ResultSet` 取完所有结果后再依次从其他 dbn 的 `ResultSet` 取记录, 知道全部取完为止。如果结果集特别大, 可能导致某个 dbn 上的 `ResultSet` 打开后很久都没有被读取, 这样的连接会被 MySQL 服务器关闭, 当再去读取结果时便出现这个异常。通常出现在镜像库访问。

解决办法:

设置 MySQL 的 `net_read_timeout` 和 `net_write_timeout` 为足够长, 默认为 30 和 60 秒。

对语句加 `order by`, 保证每个 dbn 的结果集都有机会被及时访问 (应用不一定接受)。

18.2 MySQL 连接数过多异常:

异常信息: Too many connections

产生原因和解决办法: MySQL 服务器支持的并发连接数是有限的, 通常设置为 800。一旦并发连接数超过这个限制, 新的连接将被拒绝, 抛出这个异常。出现这个问题通常是 MySQL 服务器上的连接释放不掉, 而不是 MySQL 服务器设置的连接数限制过小。连接释放不掉的原因很多:

- a) MySQL 服务器过载, 语句执行被卡住了, 可通过 `show processlist` 查看, 造成阻塞的语句可以根据情况 `kill` 掉;
- b) 语句本身有问题 (索引不合适、锁竞争激烈) 需要较长时间才能执行完, 通过 `show processlist` 查看语句执行情况, 需要优化应用、增加索引等。
- c) MySQL 服务器上出现较多悬挂事务, 可以通过 `xa recover` 命令查看那些总是存在的悬挂事务, 通常是 Master 的诊断线程挂了, 无法自动回滚, 需要重启 Master 恢复。
- d) Client 所在服务器过载, 导致执行变慢, 无法及时释放连接, 表现为无法连到该服务器上查看状态或进行统计, 通常需要重启服务器。
- e) Client 在执行数据库操作的间隙中调用其他服务被卡住了, 用管理工具查看占用连接的线程堆栈。需要解决具体服务的问题。
- f) Client 上访问数据库的线程数过多, 事务中间又有较长其他操作时间, 导致 MySQL 服务器的连接数不够用。通常是应用设计不合理, 可视情况临时增加 MySQL 的并发连接数上限。

18.3 DDB 连接过多

错误信息: Caused by: java.sql.SQLException: Too many total ddb connections [max 1024]

产生原因和解决办法: Client 上允许同时建立的最大默认 DDB 连接数是 1024, 一旦超出则报该异常。打开连接数过多可能有以下几种情况:

- a) **Client** 建立和使用连接后忘记关闭。可以通过管理工具查看 **Client** 的统计状态来检查，可以查看处于 **idle** 状态的连接最近执行的语句和当前占用连接的线程堆栈，帮助定位问题。该问题属于程序 **bug**，需要修正访问程序来解决。
- b) **Client** 使用了大量的长连接，数量超过 1024。尽量避免这种数据库访问方式，DDB 中已经内置了 MySQL 连接池，建立 DDB 连接的开销并不大，不建议使用长连接。

18.4 连接 Master 超时

异常信息：java.sql.SQLException: Client login DDB failed: Read timed out

产生原因：Client 在第一次连接 DDB 时需要从 Master 读取配置信息，传输过程中可能有超时情况发生。原因可能有：

- a) 配置信息过大而网络过慢或不稳定
- b) 多个 Client 同时连 Master 时 Master 处理不过来，例如博客应用服务器同时重启时
- c) Master 与 Client 版本不一致，造成通信协议不一致

解决方法：前面两种情况可在 Client 连接 DDB 时采用超时重试的办法，后者需要更新 Client 的版本。

18.5 DBI 上下文加载错误

异常信息：java.sql.SQLException: DBI[127.0.0.1:8888/]初始化失败, DBI 上下文加载错误

产生原因：第一次建立 DDB 连接时需要连接 Master，之间的交互协议较为复杂，一旦交互失败，将导致该异常。原因可能有如下几种：

- a) 对应地址的 Master 不存在：Connection Refused.
- b) 用户没有权限访问 DDB：用户名或密码验证失败！，用户凭证[user/password]验证不通过：User syss does not exist / Password is not matched.
- c) Client 版本不兼容：版本不匹配：DBI 版本是[4.3.1-82637]，而 MServer 版本是[4.4.1]，同时支持的 DBI 的版本范围是[4.4]

解决办法：访问正确的 Master 地址；使用正确的用户和口令；更新 Client 版本和 Master 一致。

18.6 AServer 未启动完成

错误信息：

java.sql.SQLException: DBI[127.0.0.1:8888/]初始化失败, AServer 未启动完成！

Caused by: com.netease.backend.db.common.config.ConfigException: AServerConfig 配置管理服务初始化失败：配置文件目录可能已被占用！

产生原因：每个 client 进程都需要独占一个日志目录用于存放配置和日志信息，默认是./log，可以在连接 url 中进行设置，host:port?logdir=xxx。如果日志目录已经被一个 client 进程锁定，则第二个启动的 client 进程则会抛出该异常。

Caused by: com.netease.backend.db.common.config.ConfigException: AServerConfig 配置管理服务初始化 I/O 错误：Permission denied

产生原因：配置文件目录存在但没有读或者写权限

Caused by: com.netease.backend.db.common.config.ConfigException: AServerConfig 配置管理服务初始化 I/O 错误：No such file or directory

产生原因：配置文件目录不存在且无创建 log 目录权限

解决办法：在连接 url 中设置正确的配置文件目录 host:port?logdir=xxx，并分配权限，然后重启应用。

18.7 对镜像库执行写操作

异常信息: java.sql.SQLException: The MySQL server is running with the --read-only option so it cannot execute this statement

产生原因: 镜像库 MySQL 通常以 read only 方式运行, 除管理员以外的帐号不允许写操作, 试图执行写操作会返回上述异常。

解决办法: 避免写镜像库, 如果确实有需要, 给对应的用户分配足够的权限。

18.8 Client 访问错误过多而被 MySQL 服务器禁止访问

错误信息:

java.sql.SQLException: null, message from server: "Host 'club-6.space.163.org' is blocked because of many connection errors; unblock with 'mysqladmin flush-hosts'"

产生原因: MySQL 服务器的防攻击功能在发现来自某个主机的请求执行错误次数过多时, 会将来自该主机的访问 block 掉, 禁止后续访问。该异常通常发生在线上环境。

解决办法: 由 DBA 在对应的 MySQL 服务器上执行 'mysqladmin flush-hosts', 消除限制, 避免 Client 上产生过多无法被 MySQL 服务器处理的请求。

18.9 DBN 被禁用

错误信息: Operation is denied since it involves disused dbn.

产生原因: 有以下两种情况下 DBN 会被禁用

- a) DBA 维护数据库时避免某个 DBN 被外界访问时手工禁用 DBN
- b) Master 检测到连接某个 DBN 超时, 为避免 Client 被连接 DBN 操作卡住, 自动禁用该 DBN

解决办法: DBA 手工解禁 DBN, 对于后者 Master 检测到可以 DBN 可以重新连接后自动解禁, 由于通知每个 Client 比较耗时, 禁用和解禁也需要一些时间。

18.10 获取 MySQL 连接超时

错误信息: Get resource from pool 'blogmirror' time out

产生原因和解决办法: 执行语句申请 MySQL 连接池资源时如果空闲连接耗尽, 而等待连接池的请求数小于连接池上限则进行等待, 等待超时仍无法获得连接则抛出该异常。造成这种问题的根本原因通常是连接不能及时被释放掉, 具体可能由以下几种情况:

- a) MySQL 服务器过载, 语句执行被卡住了, 可通过 show processlist 查看, 造成阻塞的语句可以根据情况 kill 掉;
- b) 语句本身有问题 (索引不合适、锁竞争激烈) 需要较长时间才能执行完, 通过 show processlist 查看语句执行情况, 需要优化应用、增加索引等。
- c) MySQL 服务器上出现较多悬挂事务, 可以通过 xa recover 命令查看那些总是存在的悬挂事务, 通常是 Master 的诊断线程挂了, 无法自动回滚, 需要重启 Master 恢复。
- d) Client 所在服务器过载, 导致执行变慢, 无法及时释放连接, 表现为无法连到该服务器上查看状态或进行统计, 通常需要重启服务器。
- e) Client 在执行数据库操作的间隙中调用其他服务被卡住了, 用管理工具查看占用连接的线程堆栈。需要解决具体服务的问题。
- f) Client 上访问数据库的线程数过多, 事务中间又有较长其他操作时间, 导致连接池不够用。通常是应用设计不合理, 可视情况临时增加连接池上限。

18.11 MySQL 连接池耗尽

错误信息:

com.netease.backend.db.common.exceptions.DDBConnFullException: Get null from pool

产生原因和解决办法：执行语句申请 MySQL 连接池资源时如果空闲连接耗尽，而等待连接池的请求数大于连接池上限则抛出该异常。造成这种问题的根本原因通常是连接不能及时被释放掉，具体可能由以下几种情况：

- g) MySQL 服务器过载，语句执行被卡住了，可通过 `show processlist` 查看，造成阻塞的语句可以根据情况 kill 掉；
- h) 语句本身有问题（索引不合适、锁竞争激烈）需要较长时间才能执行完，通过 `show processlist` 查看语句执行情况，需要优化应用、增加索引等。
- i) MySQL 服务器上出现较多悬挂事务，可以通过 `xa recover` 命令查看那些总是存在的悬挂事务，通常是 Master 的诊断线程挂了，无法自动回滚，需要重启 Master 恢复。
- j) Client 所在服务器过载，导致执行变慢，无法及时释放连接，表现为无法连到该服务器上查看状态或进行统计，通常需要重启服务器。
- k) Client 在执行数据库操作的间隙中调用其他服务被卡住了，用管理工具查看占用连接的线程堆栈。需要解决具体服务的问题。
- l) Client 上访问数据库的线程数过多，事务中间又有较长其他操作时间，导致连接池不够用。通常是应用设计不合理，可视情况临时增加连接池上限。

18.12 调用 DDB 不支持的 JDBC 接口

错误信息： `java.sql.SQLException: Not supported function!`

产生原因： Client 调用了 DDB 不支持的 JDBC 接口

解决办法： 避免调用这类接口，如果确实需要，可以让 DDB 开发人员实现对应接口。

18.13 更新被禁止写操作的表

错误信息： `SQLException :Write operation on Table 'UserRelation' is disabled.`

产生原因： DBA 通过管理工具可以禁用某个表上的写操作，另外 DBA 在修改表定义时也可以选择禁用上面的写操作。写操作被禁用时试图该表会抛出该异常。

解决办法： DBA 通过管理工具解禁该表上的写操作。

18.14 系统找不到 secret.key 文件

错误信息：

`com.netease.backend.db.client.ASEException: DBI 上下文加载错误, En/Decrypting KEY error
Caused by: java.net.MalformedURLException: no protocol: ./secret.key`

产生原因： 给定的 `secret.key` 路径不存在或不可读

解决办法： 指定可读的 `secret.key` 路径

18.15 写日志错误

错误信息：

`java.sql.SQLException: Save log meta information of ddb 'logdir/xalog.meta' failed.
logdir/xalog.meta (No such file or directory)`

产生原因： url 中指定的 `logdir`（或默认 `logdir`）不存在

解决办法： 创建对应目录或 url 中指定存在的目录

18.16 DDB Driver 正在被关闭

错误信息： `DBSQLException: DDB has been shutdown.`

产生原因： Client 进程退出时，DDB Driver 正在进行清理操作，此时访问 DDB 时会抛出该异常。

解决办法： 这类异常应该可以无视。

18.17 MySQL 锁等待超时

错误信息: SQLException: Lock wait timeout exceeded; try restarting transaction

产生原因和解决办法:

- a) 对某些表记录的并发写操作过多。通过研究发生该错误的语句找到原因，通过优化应用来解决
- b) 更新操作没有使用索引，每次都锁全表。可通过增加合适索引来解决
- c) 悬挂事务造成。可能是 Master 的悬挂事务处理线程异常，可通过重启 Master 解决。

18.18 DDB Master 与 MySQL 上的表定义不一致

错误信息: com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: Unknown column 'UserRelation.InviteTime' in 'field list'

产生原因: DDB Master 与 MySQL 上的表定义不一致

解决办法: 修改 Master 配置或 MySQL 表定义，保持一致。

18.19 两个 Client 进程使用了相同的日志文件

错误信息: ERROR com.netease.backend.db.DBConnection - Recover from xa log failed:
java.lang.RuntimeException: javax.transaction.SystemException: LogException occurred in xaLog.open() LogFileManager.open: unable to obtain lock on file
D:\apache-tomcat-6.0.18\log\ddb_blog_mirror_xalog_1.log

产生原因: 每个 Client 都会独自使用一个分布式事务日志，当一个 Client 进程试图访问已经被另一个 Client 进程使用的日志文件时，将报上述异常。

解决办法: 给每个 Client 指定不同的日志目录或日志名称。

18.20 语句条件个数超过限制

错误信息: Caused by error on dbi: exceed max condition number:2001

产生原因: DDB3.0M3 版本中增加了对语句条件中由 or 或 and 连接的条件个数的限制，用来避免条件过多造成系统负载突然升高，默认上线为 2000 个。当语句条件数超过限制就会报该异常。

解决办法: 尽量避免语句中条件个数过多。如果当前限制无法满足应用需要，可以通过管理工具修改限制条件数。

18.21 键值重复

错误信息: Duplicate entry '7' for key 1

产生原因和解决办法: 有以下三种情况

- d) 键值由 DDB 产生，可能是操作失误导致 DDB 中该表的起始分配 id 已经小于表中实际记录值的最大值。需要将 DDB 中该表起始 id 设置为一个足够大的安全值。
- e) 键值由应用程序自己产生，则需要检查键值产生器的逻辑正确性，或其他人为因素。
- f) 将允许重复的键设置成了 primary key 或 unique key。可通过修改表定义解决。

18.22 表 ID 消耗殆尽

错误信息: Master has no id to allocate.

产生原因: Master 上为该表保留的 id 全部分配完毕，继续申请新的 id 会报该异常。

解决办法: DBA 通过管理工具修改该表的剩余 id 数量。

18.23 内存溢出

错误信息: `Java.lang.OutOfMemoryError: Java heap space`

产生原因和解决办法: 和 DDB 相关的原因有两种

- g) 没有用游标返回超大结果集, 导致结果集全部装入 `Client` 内存。可以通过控制返回结果集大小 (分段读取) 或使用游标解决。
- h) 语句中存在数量巨大的 `and` 或 `or` 条件并且条件嵌套层数过多, 导致 DDB Driver 进行语法分析时函数嵌套层次过多造成栈溢出。必须避免过多条件层次嵌套。

19 使用限制和存在的问题

19.1 使用限制

19.1.1 节点数据库限制

分布式数据库基于 MySQL 数据库节点实现。由于 MySQL 从 5.0 开始才支持 XA Transaction (分布式事务), 因此分布式数据库只能使用 MySQL5.0 以上版本 (含 5.0)。如果需要使用分区功能, 则需要 MySQL5.1 以上版本。

目前, DDB 在原有基础上, 还新增了对 Oracle 数据库和神通奥斯卡数据库的支持。但是提供的功能不如 MySQL 全面, 比如统计分析、数据迁移等功能没有。

19.1.2 编程接口限制

目前只提供了 Java 语言的编程接口, 不支持 C/C++ 等其他编程语言。只是部分实现了 JDBC 接口。支持一些常用的 SQL 语句, 不支持子查询、集合 `union` 操作。

19.2 存在问题

- MySQL 目前版本 (5.1.12) 还不是特别稳定, 在使用了分布式事务后有时会出现自动重启。
- MySQL 没有支持完整的 XA Transaction 标准, 在我们修改了 MySQL 的相关代码后虽然解决了连接关闭后自动回滚的问题, 但在启动新的连接来提交悬挂事务后, 提交的事务操作不被记录 binlog 的问题仍然存在, 这将导致 binlog 与数据库状态不一致, 影响通过 binlog 进行数据恢复和复制。

20 性能测试报告

20.1 测试环境配置

一共两台测试服务器:

服务器 db-34 配置: 8 核 CPU, 开启超线程, MemTotal: 41282532 KB;

服务器 db-35 配置: 8 核 CPU, 开启超线程, MemTotal: 49556448 KB。

DDB 管理服务器 (`master`)、DDB 系统库 (`sysdb`) 和测试客服端 (`client`) 搭在同一台服务器 db-34 上, 数据库节点 (`dbn`) 搭在另一台服务器 db-35 上。由于测试过程中管理服务器和系统库基本不产生负载, 所以对测试结果基本不产生影响。

1. 客服端的 DDB 配置中设置连接池最大连接数、XA 连接池最大连接数、Mysql 连接最大并发数均设为 500。
2. 数据库节点的配置文件：

```
[mysql]
port                =3307
socket              = /mnt/ddb/1/dbn1/mysql.sock
default-character-set = gbk

[mysqld_safe]
user                = ddb
nice                = 0

[mysqld]
bind-address        = 172.17.3.32
port                = 3307
pid-file            = /mnt/ddb/1/dbn1/mysqld.pid
socket              = /mnt/ddb/1/dbn1/mysqld.sock
basedir             = /usr/local/mysql
datadir             = /mnt/ddb/1/dbn1
innodb_data_home_dir = /mnt/ddb/1/dbn1
innodb_log_group_home_dir = /mnt/ddb/1/dbn1
tmpdir              = /mnt/ddb/1/dbn1/tmp
log-error            = /mnt/ddb/1/dbn1/mysqld.log
log_slow_queries    = /mnt/ddb/1/dbn1/mysql-slow.log
# log_bin            = /mnt/ddb/1/dbn1/mysql-bin.log

user                = ddb
language            = /usr/local/mysql/share/mysql/english
table_cache         = 128
long_query_time     = 4
max_connections     = 800
query_cache_type    = 0
default-character-set = gbk
default-storage-engine = innodb
skip-external-locking
expire_logs_days    = 1
server-id           = 1
max_binlog_size     = 50M
max_allowed_packet  = 16M
innodb_buffer_pool_size = 1G
innodb_data_file_path = ibdata1:10M:autoextend
innodb_autoextend_increment = 8
innodb_log_files_in_group = 2
innodb_log_file_size = 128M
innodb_lock_wait_timeout = 5

innodb_flush_method = O_DIRECT
innodb_flush_log_at_trx_commit = 2
innodb_file_per_table = 1
wait_timeout = 600

[mysqldump]
quick
quote-names
max_allowed_packet  = 16M
default-character-set = gbk
```

20.2 测试工具 DbiBench、Perf-simple

DbiBench 是专门开发用于客户端性能测试的平台，可以自定义性能测试 SQL 语句、负载线程数、运行时间等，能够实现运行吞吐量、单位响应时间等统计功能。另外，为了避免在高并发测试情况下，短时间集中向数据库获取连接请求过多对数据库造成压力，DbiBench 还实现了预热功能（ramp-up），能够在一段时间内缓慢增加负载，预热过程不参与到最后的测试数据

统计。

本次测试数据库表结构:

```
CREATE TABLE `Blog` (
  `ID` bigint(20) NOT NULL,
  `UserID` bigint(20) default NULL,
  `Title` varchar(255) default NULL,
  `Abstract` varchar(2000) default NULL,
  `Content` mediumtext,
  `AllowView` smallint(6) default NULL,
  `PublishTime` bigint(20) default NULL,
  `AccessCount` int(11) default NULL,
  `CommentCount` int(11) default NULL,
  PRIMARY KEY (`ID`),
  KEY `IDX_BLOG_UID_PUBTIME` (`UserID`, `PublishTime`, `AllowView`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

本次测试所涵盖的 SQL 语句:

1. *delete from Blog where ID = AUTO_GET_ID_KEYWORD*
2. *insert into Blog values()*
3. *update Blog set CommentCount = 1 where ID= AUTO_GET_ID_KEYWORD*
4. *select * from Blog where ID = AUTO_GET_ID_KEYWORD*
5. *select * from Blog where ID = AUTO_GET_ID_KEYWORD order by id*
6. *select * from Blog where ID = AUTO_GET_ID_KEYWORD group by id*

Perf-simple 是一套服务器服务器负载实时监控脚本，运行时通过调用 linux 的 top、vmstat、sar 等命令来收集系统运行情况。本次测试的采样周期为 10 秒。

20.3 测试用例设计

本次测试主要关注以下三方面:

1. DDB 客户端 CPU 利用率和 JDBC 客户端 CPU 和内存利用率对比。
2. DDB 使用 FORWARDBY 后的性能变化。
3. 对于简单的单节点事务操作，使用 xa 连接和使用非 xa 连接执行的性能对比。

基于以上原则，本次运行的测试用例如下:

1. 5 线程并发压力下，使用 DDB 连接数据库
2. 50 线程并发压力下，使用 DDB 连接数据库
3. 200 线程并发压力下，使用 DDB 连接数据库
4. 5 线程并发压力下，使用 DDB 的 FORWARDBY 语法连接数据库
5. 50 线程并发压力下，使用 DDB 的 FORWARDBY 语法连接数据库
6. 200 线程并发压力下，使用 DDB 的 FORWARDBY 语法连接数据库
7. 5 线程并发压力下，使用 JDBC 连接数据库
8. 50 线程并发压力下，使用 JDBC 连接数据库
9. 200 线程并发压力下，使用 JDBC 连接数据库

20.4 测试结果

runtime: 运行时间(second) ,每个小场景运行600s

Thread number: 并发线程数

Transaction type: 事务类别

no 不开启事务

tx 开启普通事务

isXa:

true 使用XA事务

false 使用非 XA 事务

Run type: 运行类别

prepare 以preparestatement方式运行, 每次循环 不重用ps实例

prepareReuse 以preparestatement方式运行, 每次循环 重用ps实例

statement 以statement方式运行

Total runned sqls: 在 runtime 时间里运行的总 sql 语句数

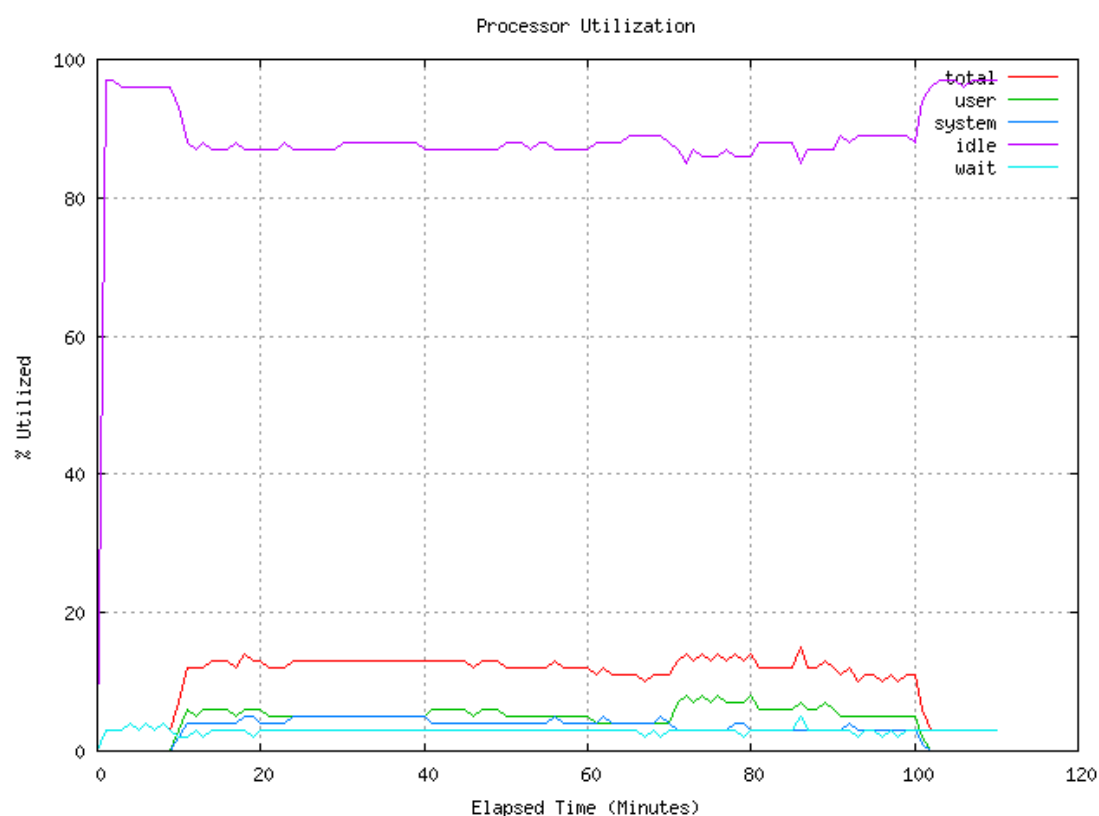
Avg response time: 平均响应时间(ms)

forwardby: 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。

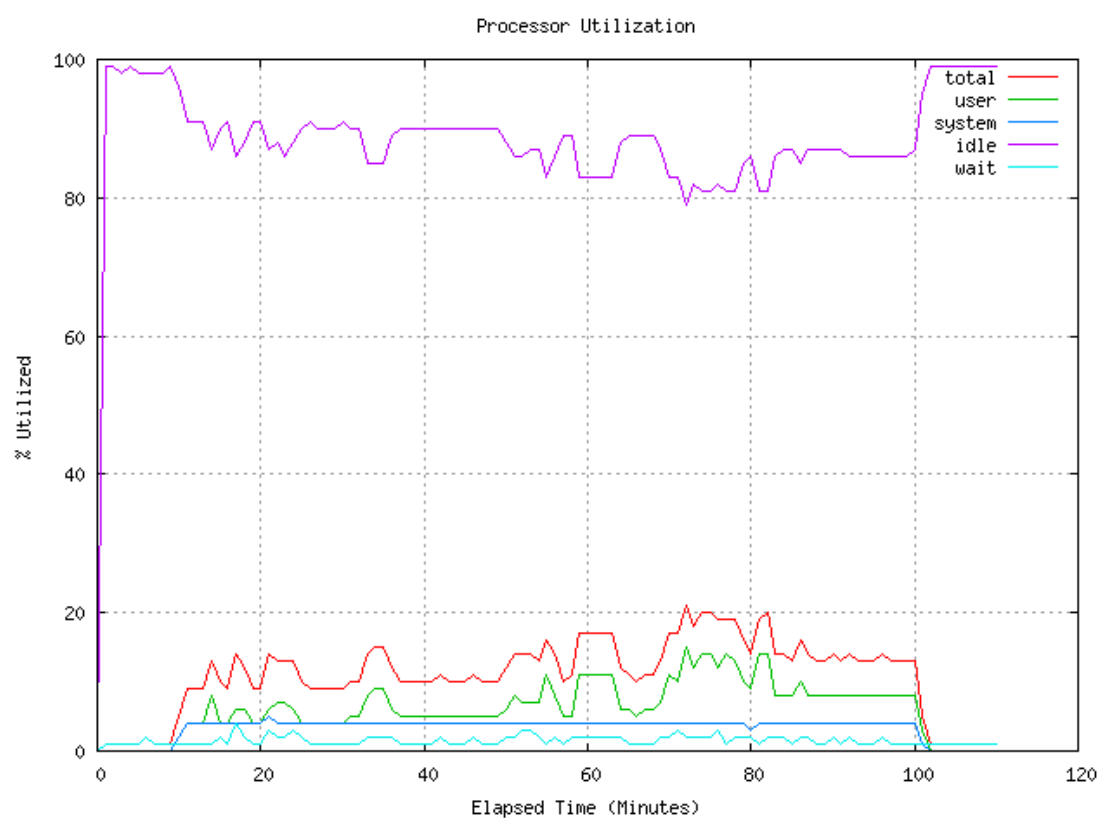
20.4.1 DDB 客户端运行测试结果

Thread number	Transaction type	Is xa	Run type	Total runned sqls (600s)	Avg response time (ms)
5	tx	true	Statement	3287477	0.90689
			Prepare	3393661	0.87972
			prepareReuse	3579099	0.83295
		false	Statement	4117676	0.72260
			Prepare	4269907	0.69793
			prepareReuse	4501735	0.66155
	no	\	Statement	8693105	0.34208
			Prepare	8926266	0.3332
			prepareReuse	9427633	0.31548
50	tx	True	Statement	17655534	1.69248
			Prepare	17591178	1.70478
			prepareReuse	18353383	1.62808
		false	Statement	22616002	1.32101
			Prepare	22095950	1.36040
			prepareReuse	22539254	1.32283
	no	\	Statement	23161895	1.09154
			Prepare	23187739	1.28986
			prepareReuse	23310048	1.28434
200	Tx	true	Statement	11918732	10.03543
			Prepare	14313045	6.44219
			prepareReuse	12984709	5.89361
		false	Statement	12732749	9.31941
			Prepare	13232312	9.15864
			prepareReuse	14320626	9.33261
	no	\	Statement	22241943	5.53399
			Prepare	23331254	5.16955
			prepareReuse	23092912	5.11822

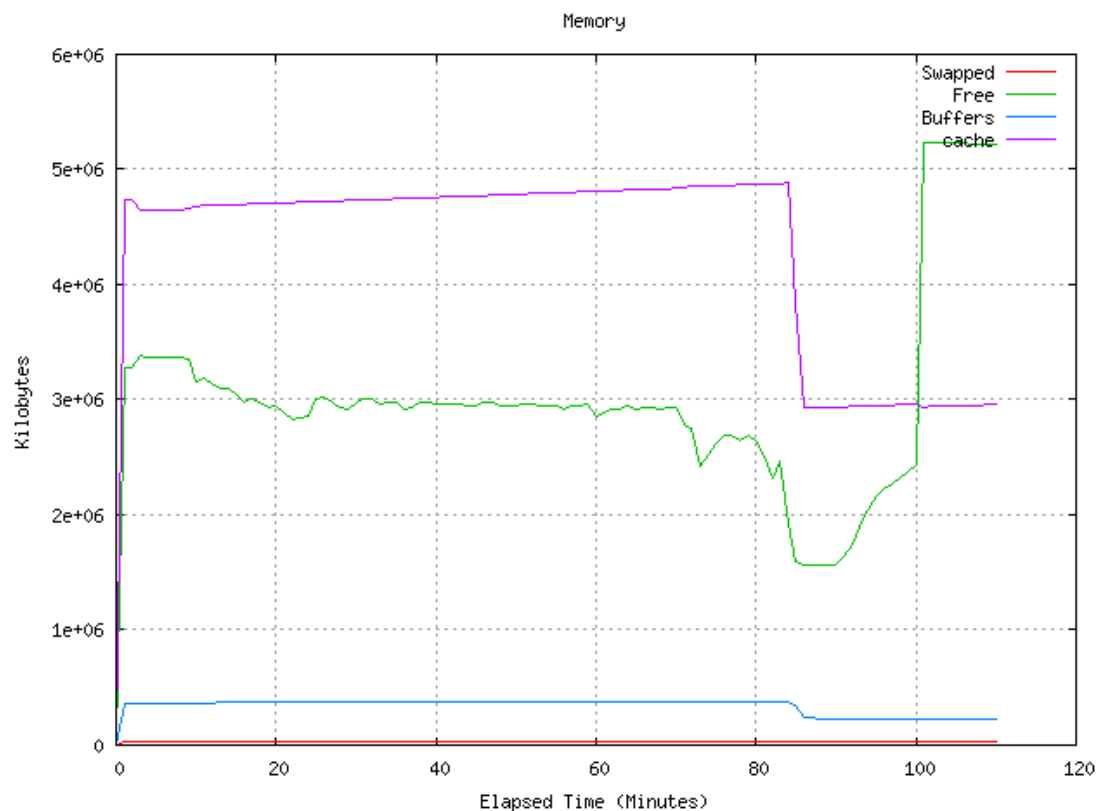
			se		
--	--	--	----	--	--



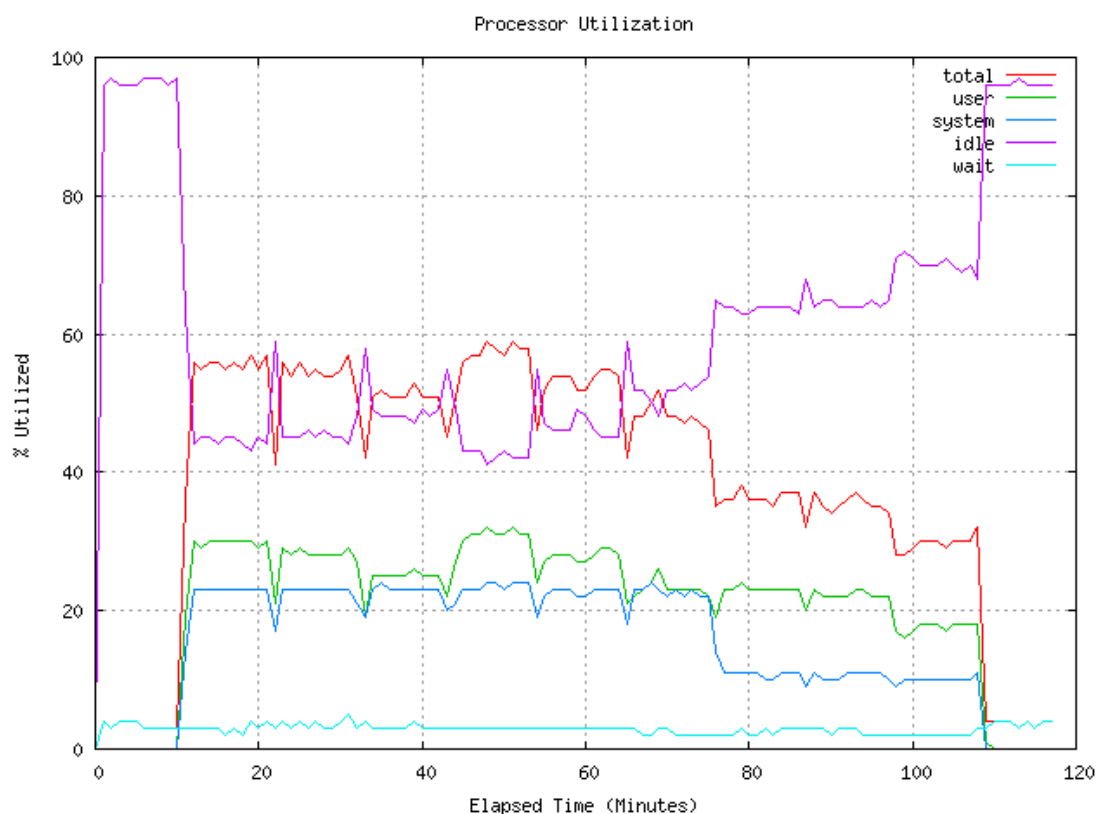
连接 ddb 跑 5 个线程时服务器 ddb-34 的 cpu 利用率



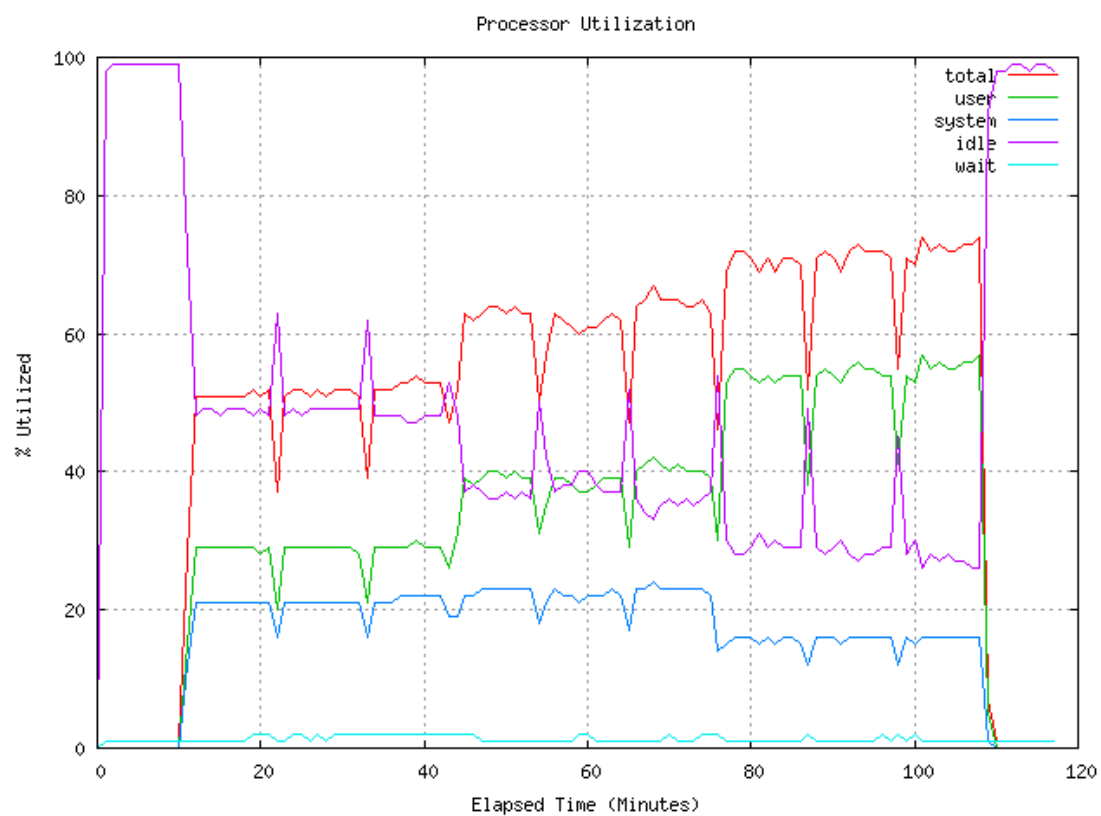
连接 ddb 跑 5 个线程时服务器 ddb-35 的 cpu 利用率



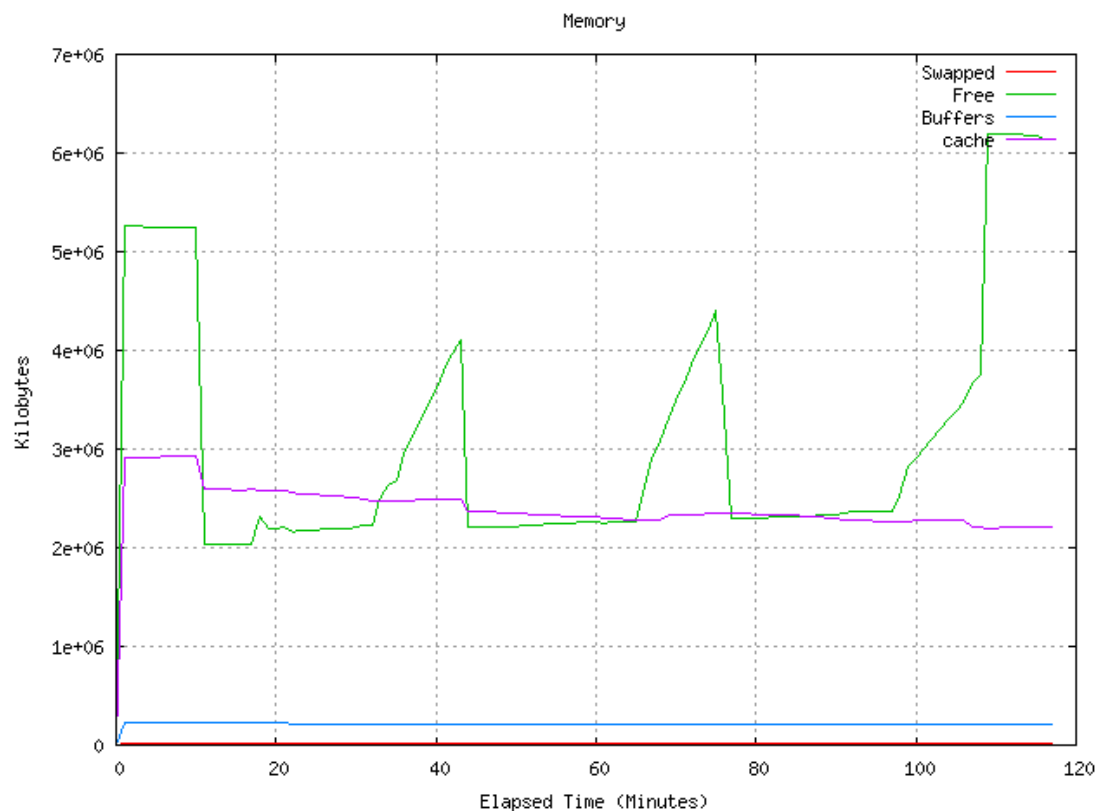
连接 ddb 跑 5 个线程时服务器 ddb-34 的内存占用率



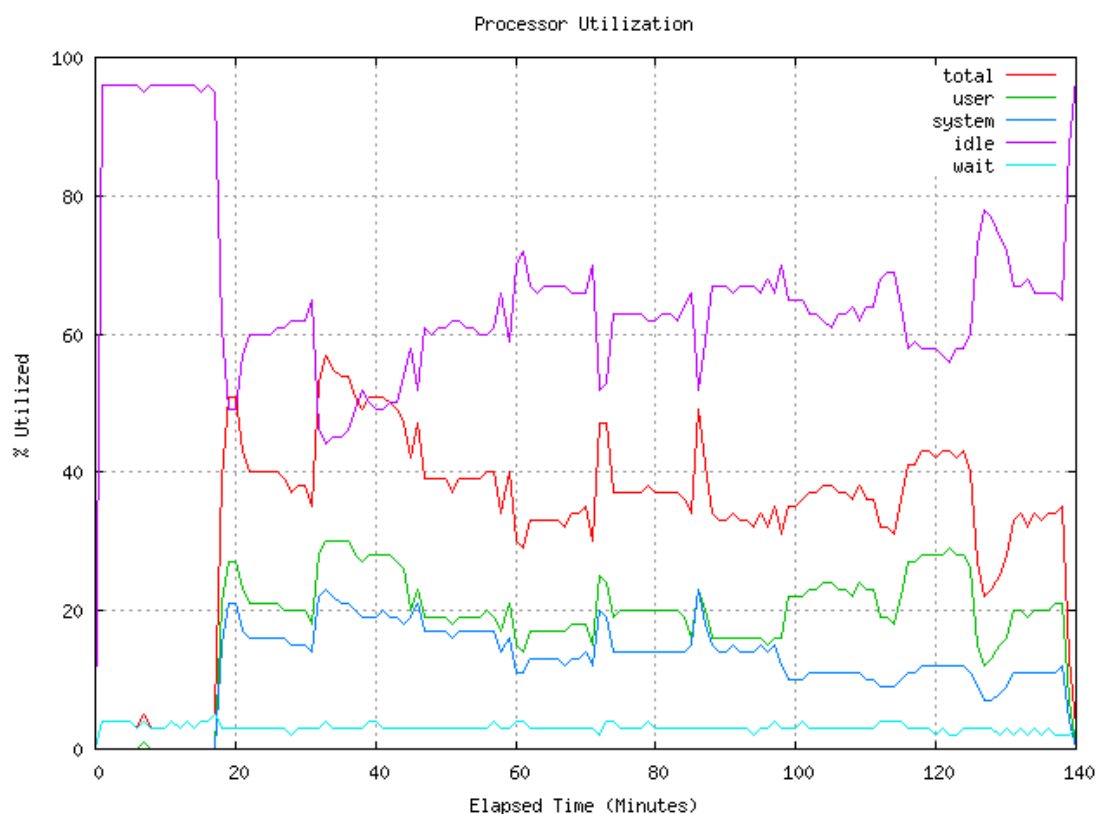
连接 ddb 跑 50 个线程时服务器 ddb-34 的 cpu 利用率



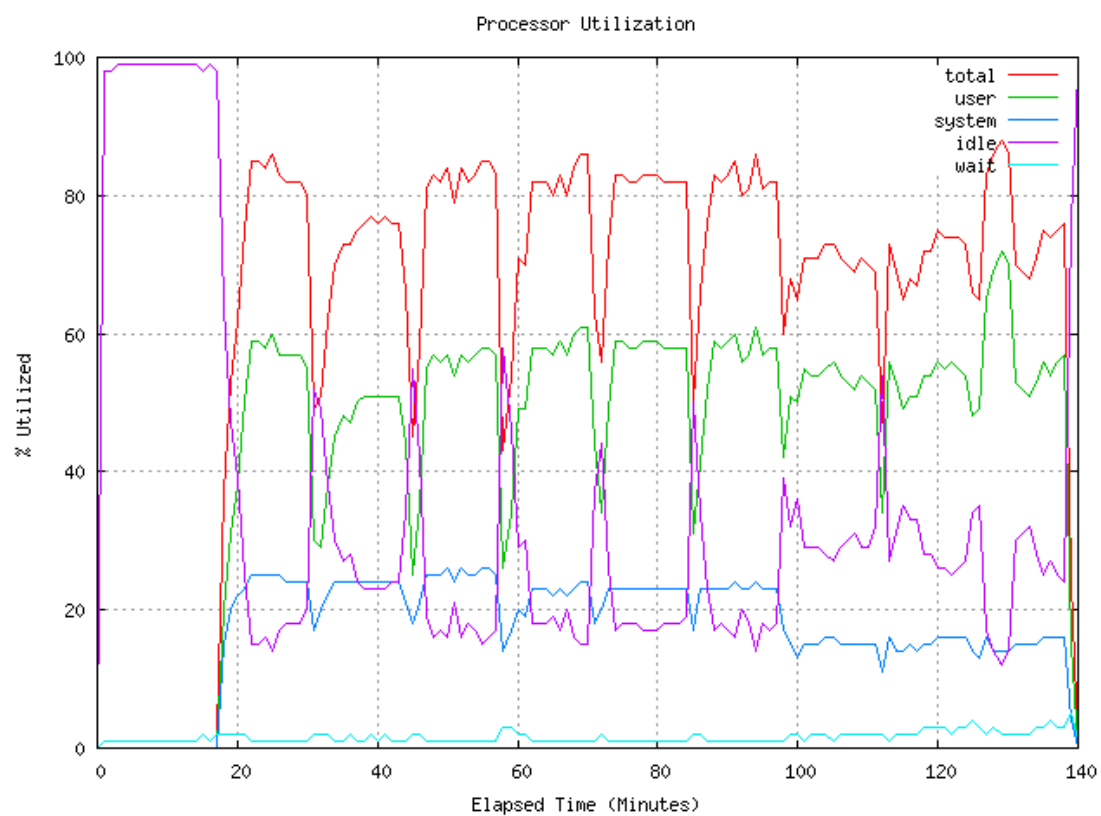
连接 ddb 跑 50 个线程时服务器 ddb-35 的 cpu 利用率



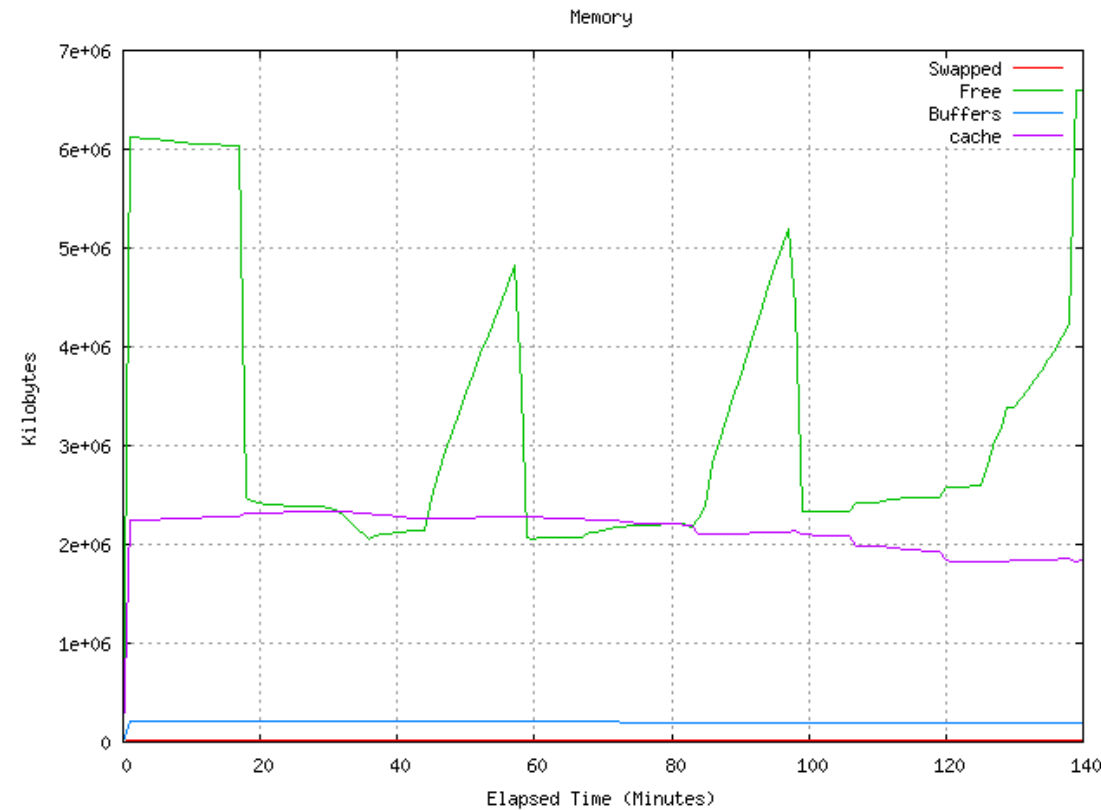
连接 ddb 跑 50 个线程时服务器 ddb-34 的内存占用率



连接 ddb 跑 200 个线程时服务器 ddb-34 的 cpu 利用率



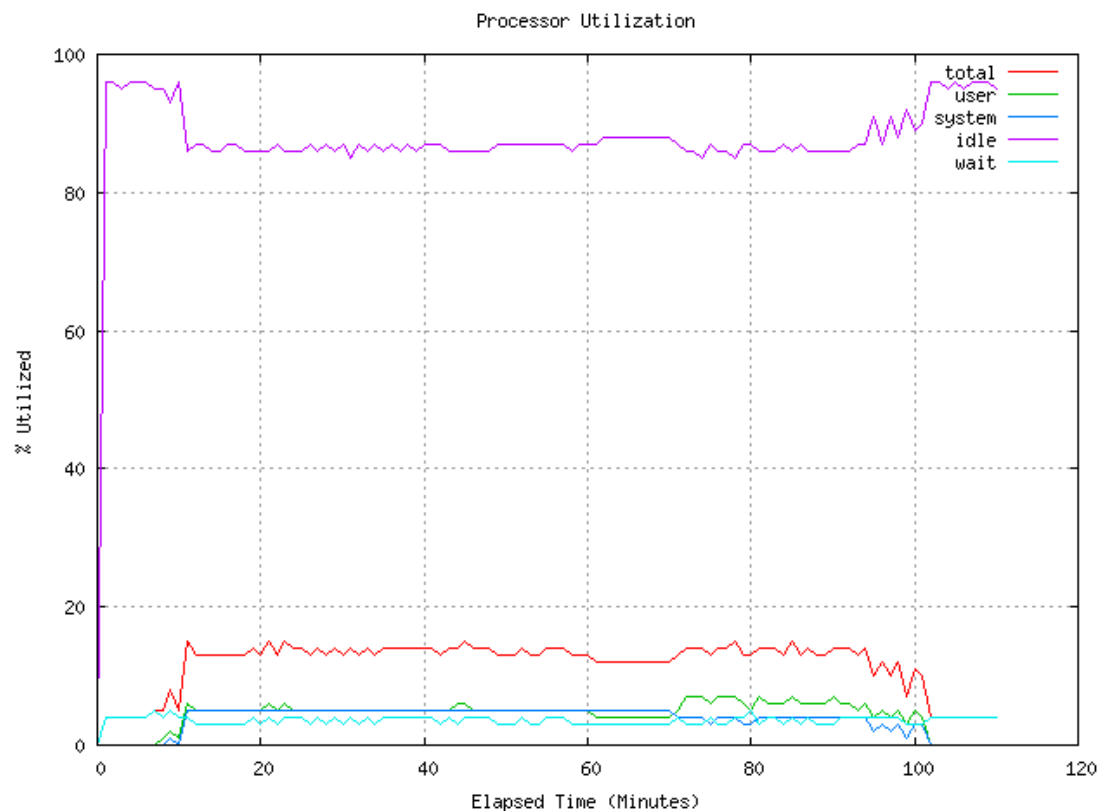
连接 ddb 跑 200 个线程时服务器 ddb-35 的 cpu 利用率



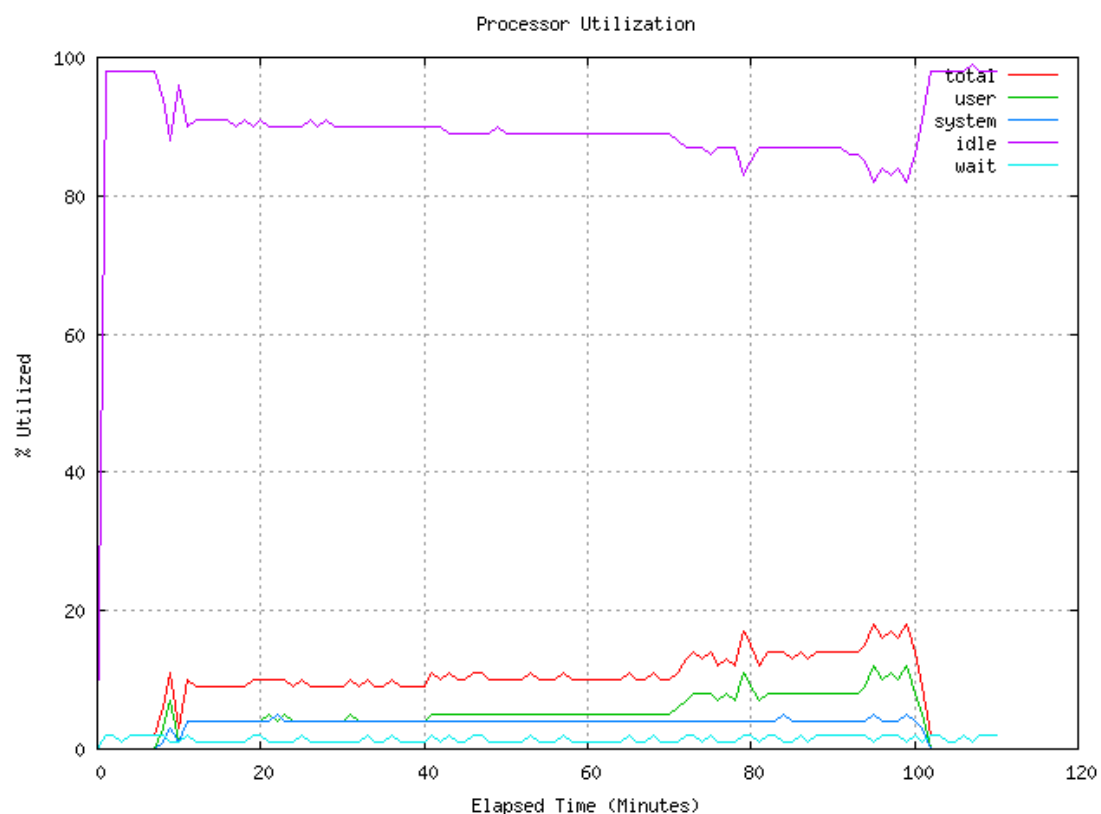
20.4.2 DDB 采用 forwardby 方式测试结果

Thread number	Transaction type	Is xa	Run type	Total runned sqls (600s)	Avg response time (ms)
5	tx	true	Statement	3627339	0.82047
			Prepare	3631951	0.81996
			prepareReuse	3796920	0.78428
		false	Statement	4571202	0.64966
			Prepare	4672075	0.63628
			prepareReuse	4834064	0.61512
	no	\	Statement	8960866	0.33150
			Prepare	10277865	0.28856
			prepareReuse	7239434	0.41099
50	tx	True	Statement	18649450	1.60085
			Prepare	18600670	1.60809
			prepareReuse	19237268	1.55828
		false	Statement	24331906	1.22549
			Prepare	24309997	1.22956
			prepareReuse	25134743	1.18470
	no	\	Statement	28002928	0.86583

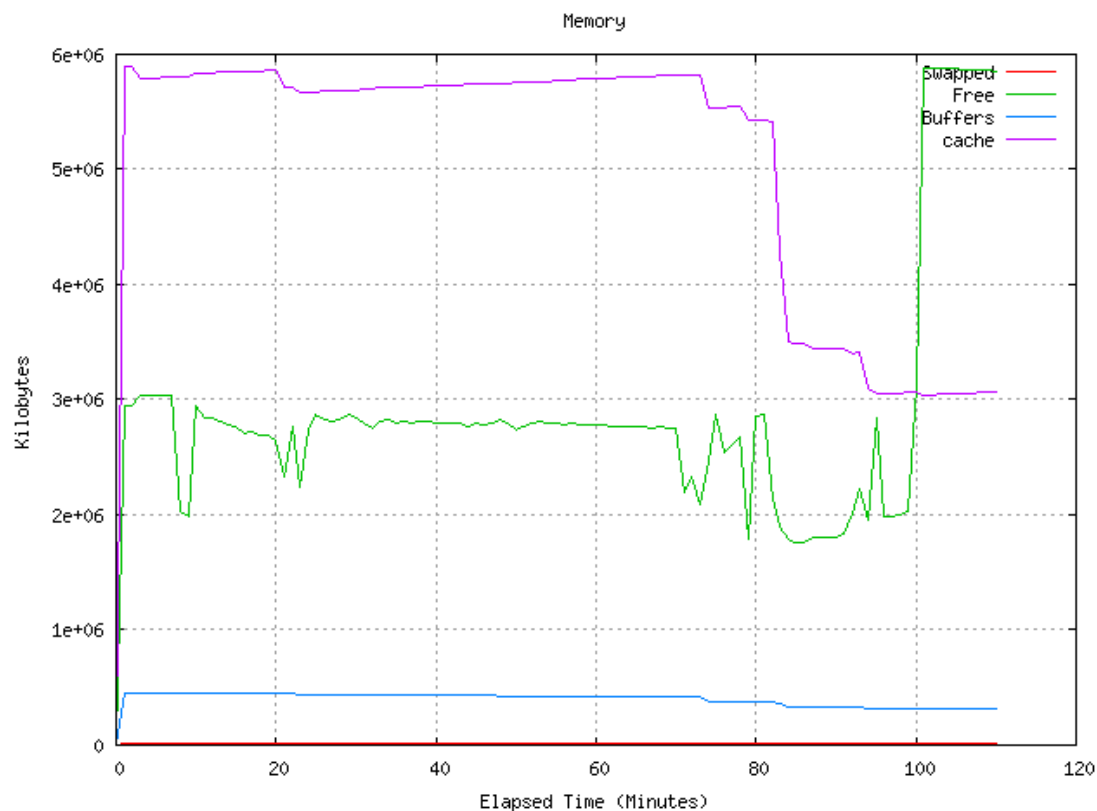
			Prepare	27531909	1.08542
			prepareReuse	27155928	1.10013
200	Tx	true	Statement	11890729	9.95978
			Prepare	17164032	5.96423
			prepareReuse	15004535	5.43241
		false	Statement	14415760	8.28993
			Prepare	15329166	7.99867
			prepareReuse	17249947	7.54630
	no	\	Statement	27389147	4.65342
			Prepare	27996333	4.39733
			prepareReuse	29067883	3.86753



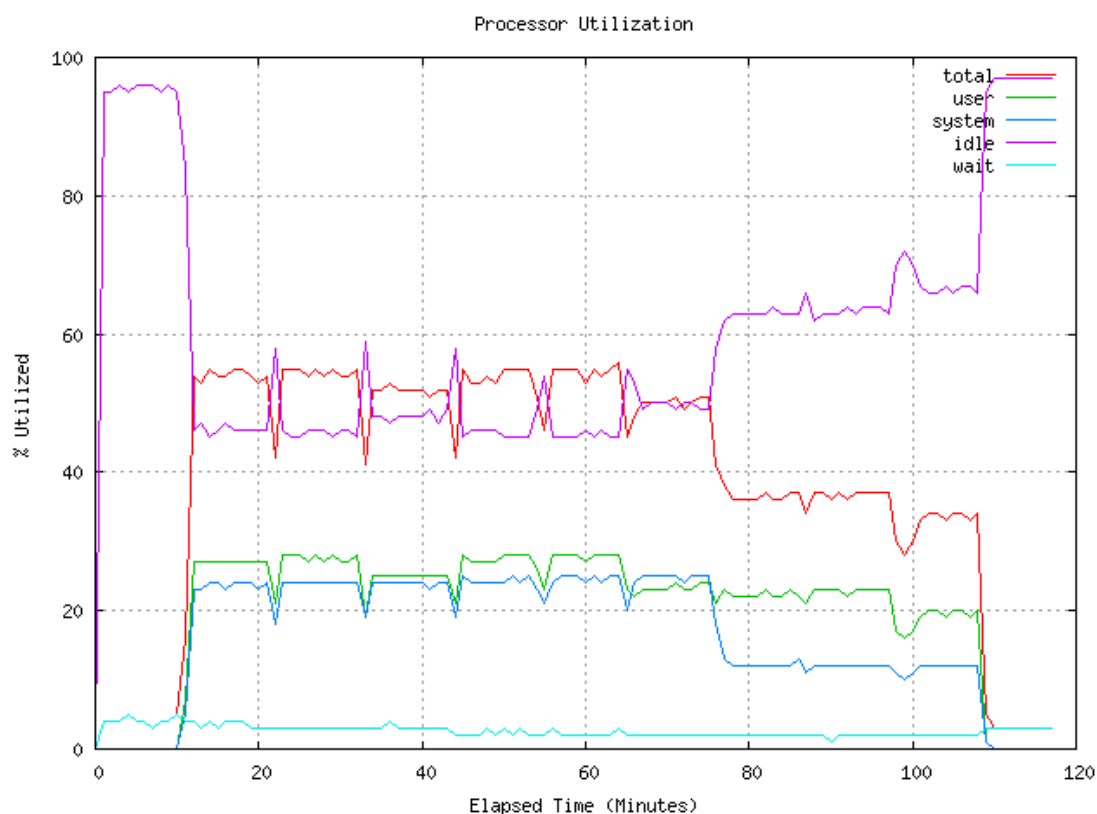
连接 ddb 跑 5 个线程时服务器 ddb-34 的 cpu 利用率(forwardby)



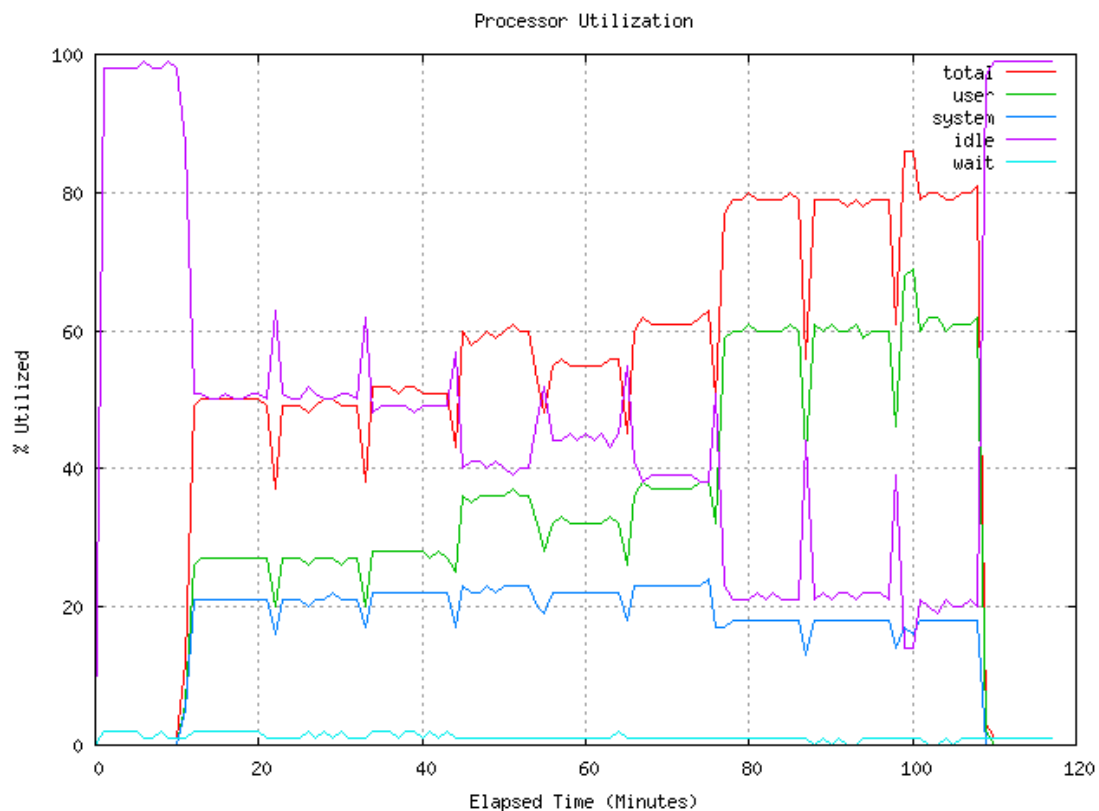
连接 ddb 跑 5 个线程时服务器 ddb-35 的 cpu 利用率(forwardby)



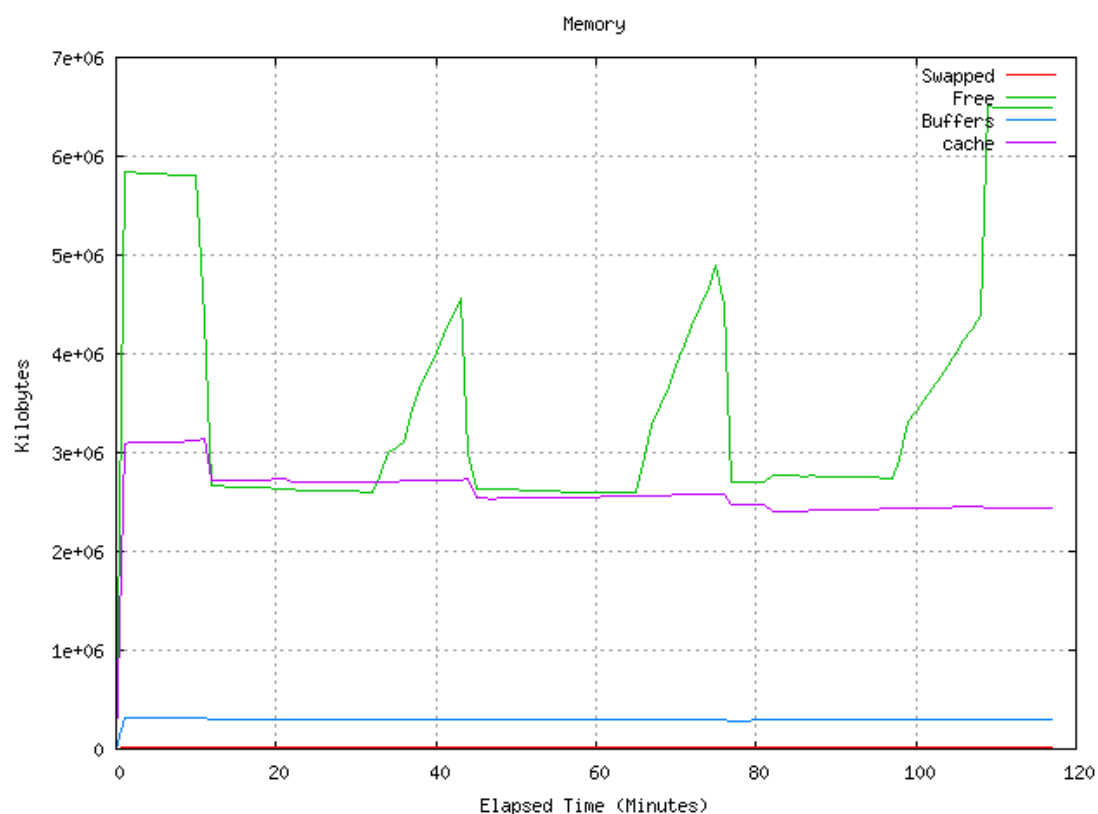
连接 ddb 跑 5 个线程时服务器 ddb-34 的内存利用率(forwardby)



连接 ddb 跑 50 个线程时服务器 ddb-34 的 cpu 利用率(forwardby)

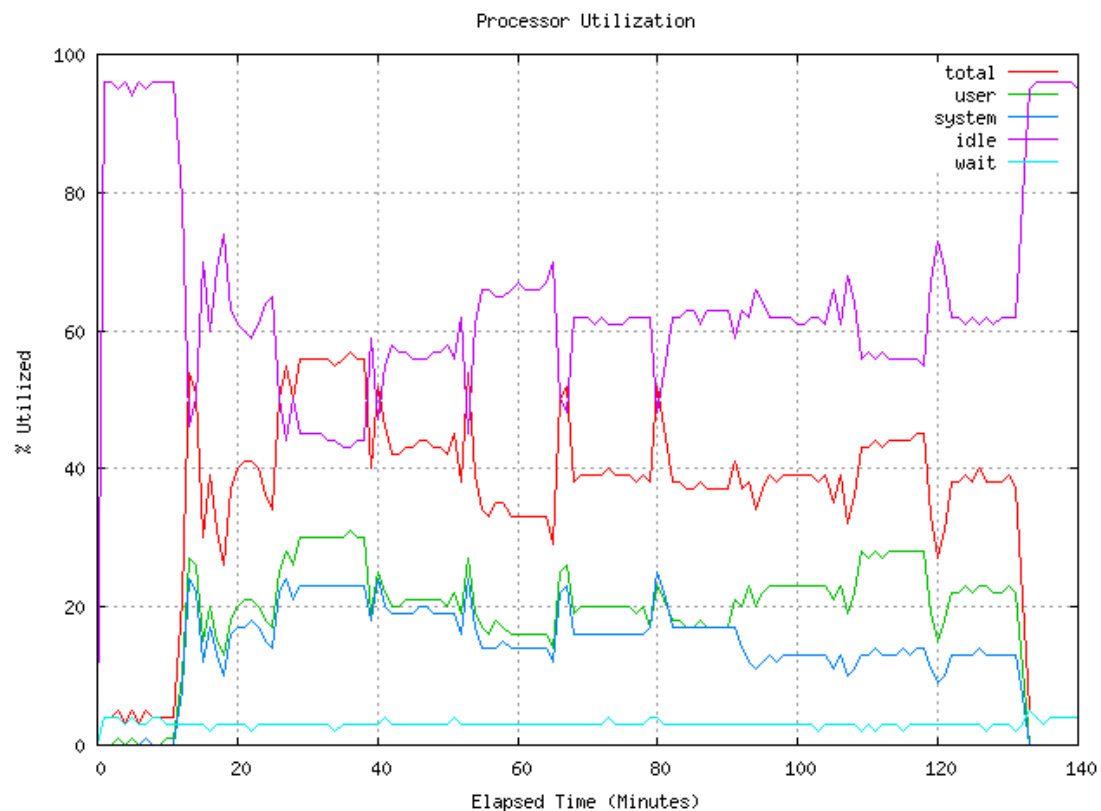


连接 ddb 跑 50 个线程时服务器 ddb-35 的 cpu 利用率(forwardby)

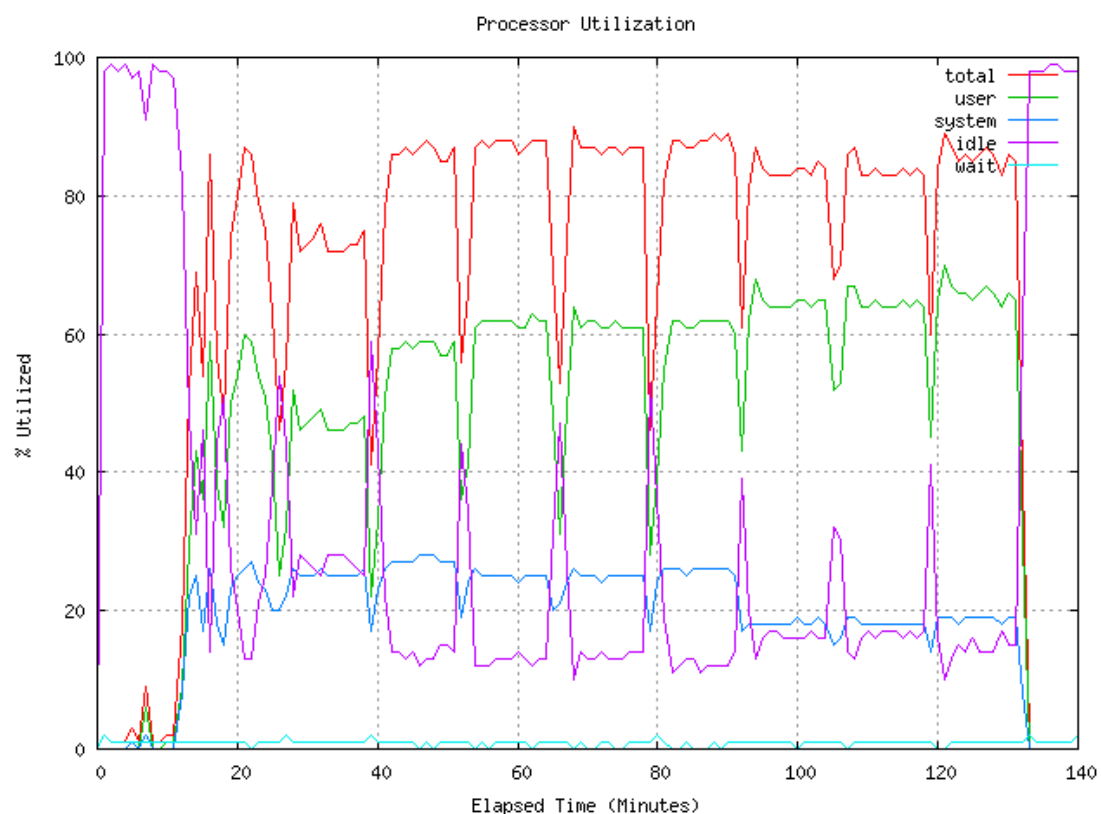


接 ddb 跑 50 个线程时服务器 ddb-34 的内存利用率(forwardby)

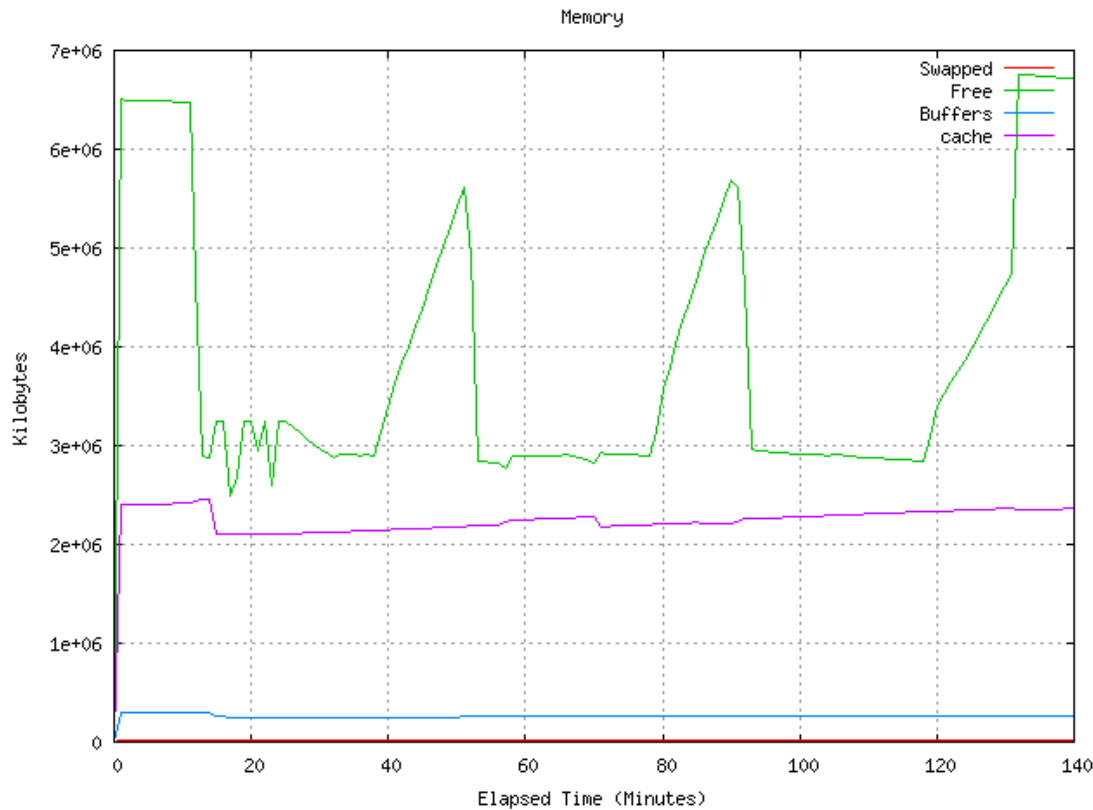
连



连接 ddb 跑 200 个线程时服务器 ddb-34 的 cpu 利用率(forwardby)



连接 ddb 跑 200 个线程时服务器 ddb-35 的 cpu 利用率(forwardby)



连接 ddb 跑 200 个线程时服务器 ddb-34 的内存利用率(forwardby)

20.4.3 JDBC 客户端运行测试结果

Thread number	Transaction type	Run type	Total runned Sqls (600s)	Avg response time (ms)
5	jdbc-tx	Statement	5485142	0.54192
		Prepare	5424338	0.54936
		prepareReuse	5524586	0.53915
	jdbc-no	Statement	13006857	0.22700
		Prepare	12540349	0.23579
		prepareReuse	13161462	0.22450
50	jdbc-tx	Statement	26096816	1.13949
		Prepare	25111164	1.51911
		prepareReuse	25768929	1.15663
	jdbc-no	Statement	26642737	1.12262
		Prepare	35496235	0.79698
		prepareReuse	26124190	1.14367
200	jdbc-tx	Statement	14266457	8.38375
		Prepare	19675604	5.04640
		prepareReuse	14833055	5.68342
	jdbc-no	Statement	23642417	5.05152
		Prepare	24424200	4.97267
		prepareReuse	26487809	4.72996

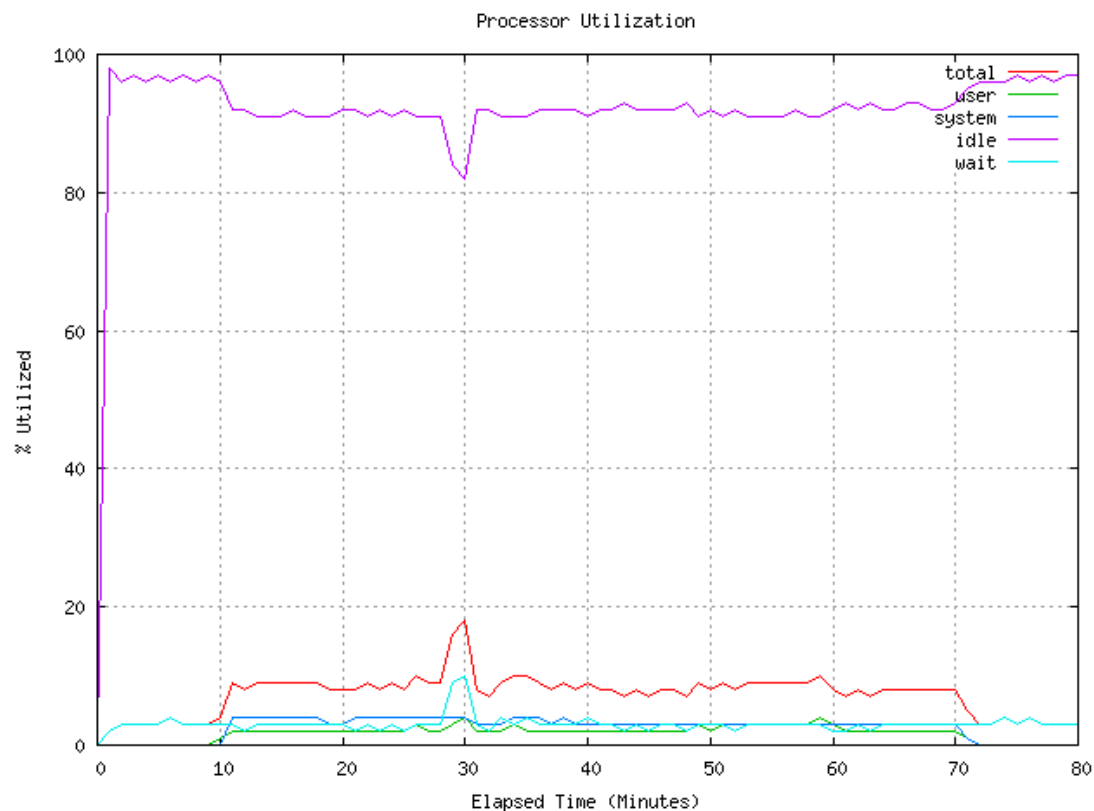


图 7-1 jdbc 直接连 mysql 跑 5 个线程时服务器 ddb-34 的 cpu 利用率

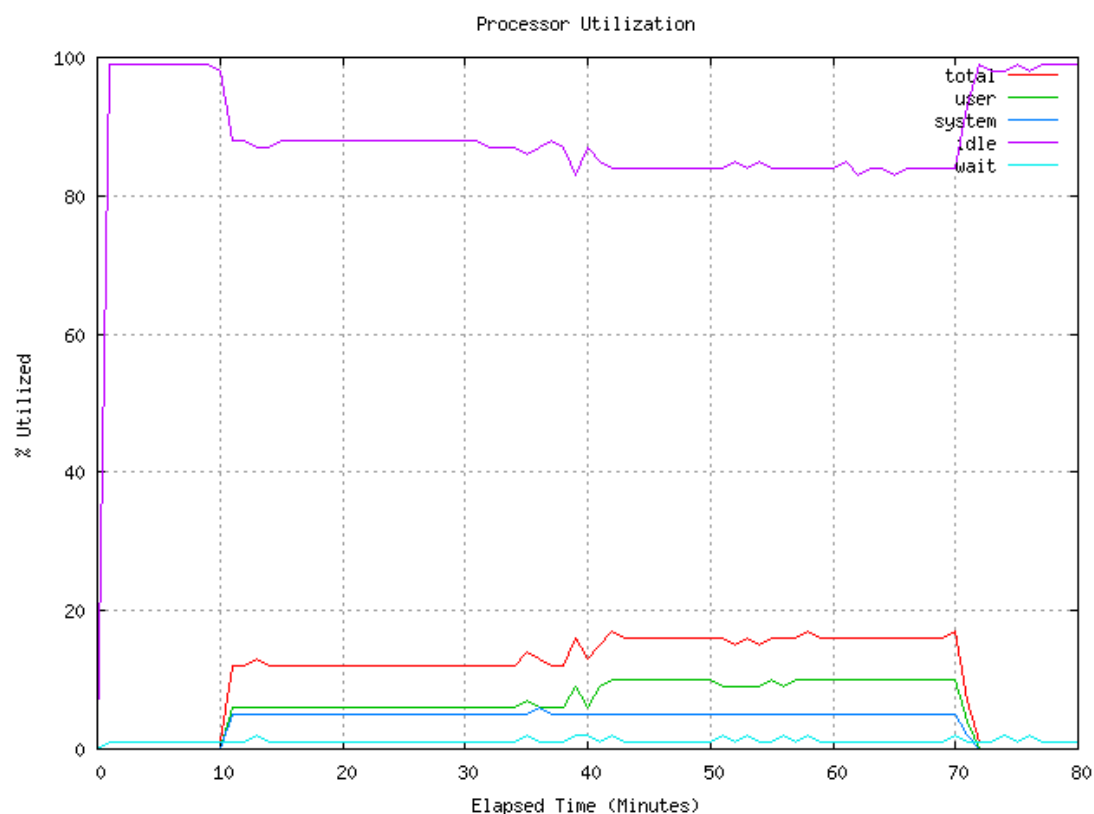
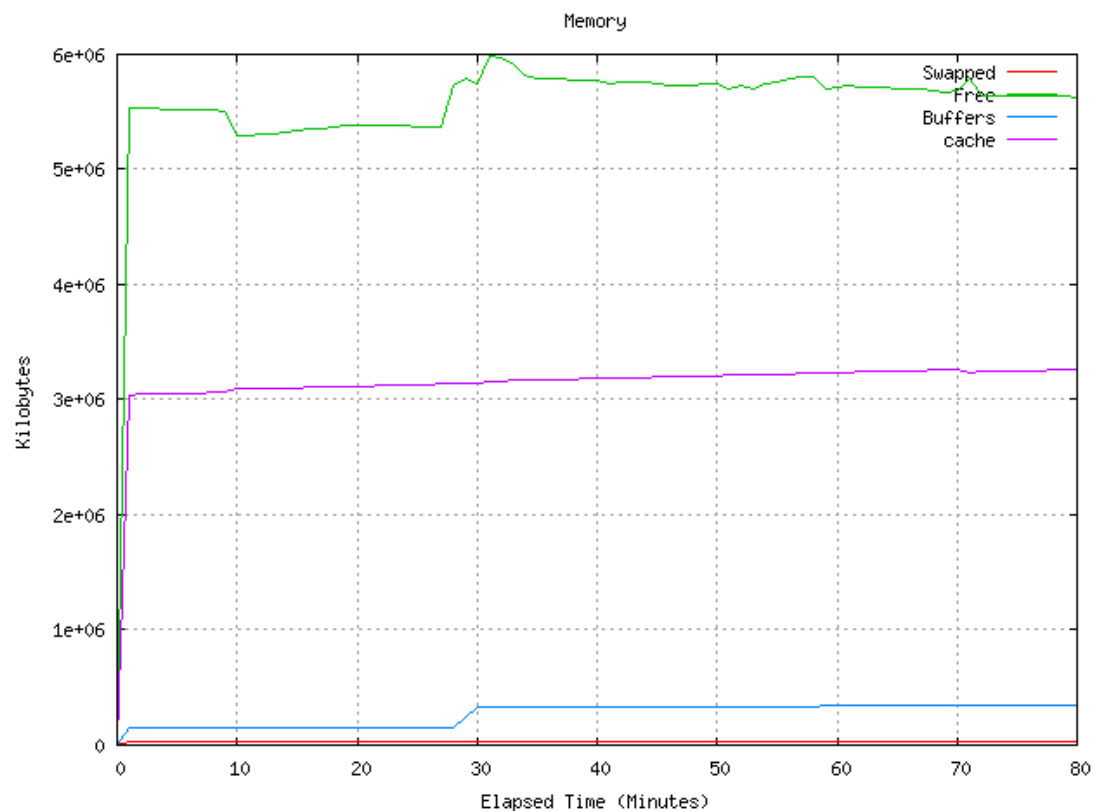


图 7-2 jdbc 直接连 mysql 跑 5 个线程时服务器 ddb-35 的 cpu 利用率



jdbc 直接连 mysql 跑 5 个线程时服务器 ddb-34 的内存占用率

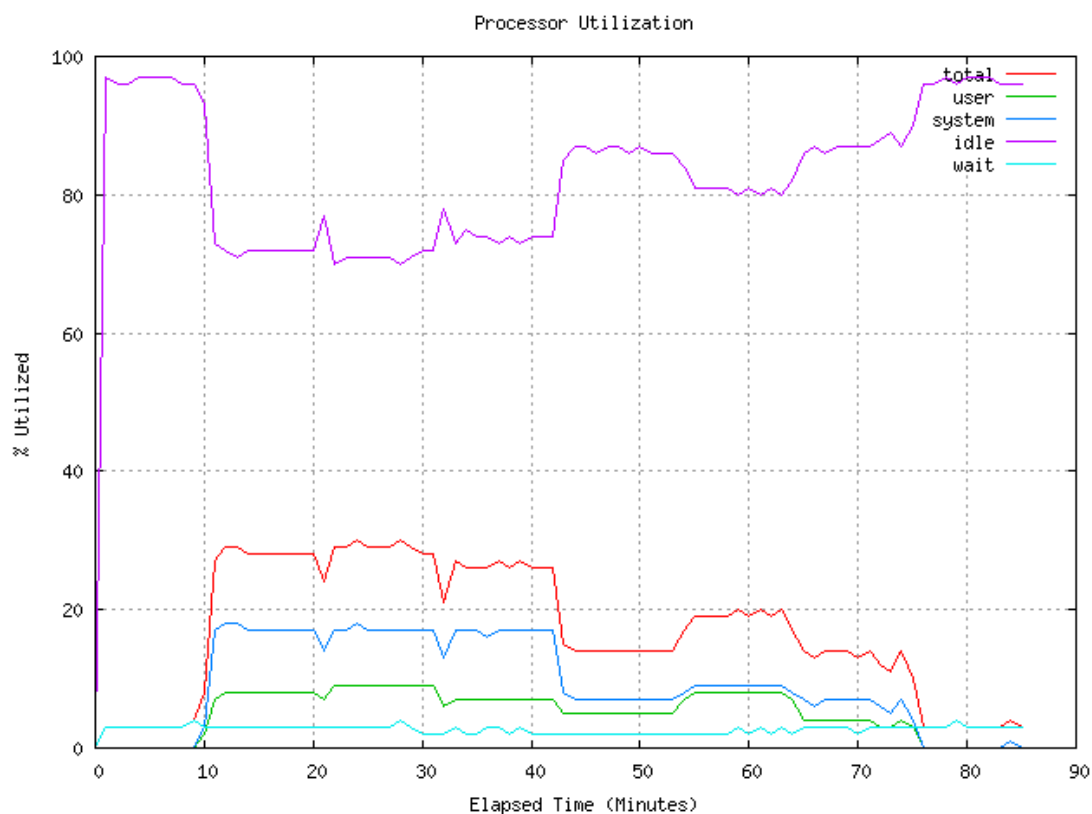


图 8-1 jdbc 直接连 mysql 跑 50 个线程时服务器 ddb-34 的 cpu 利用率

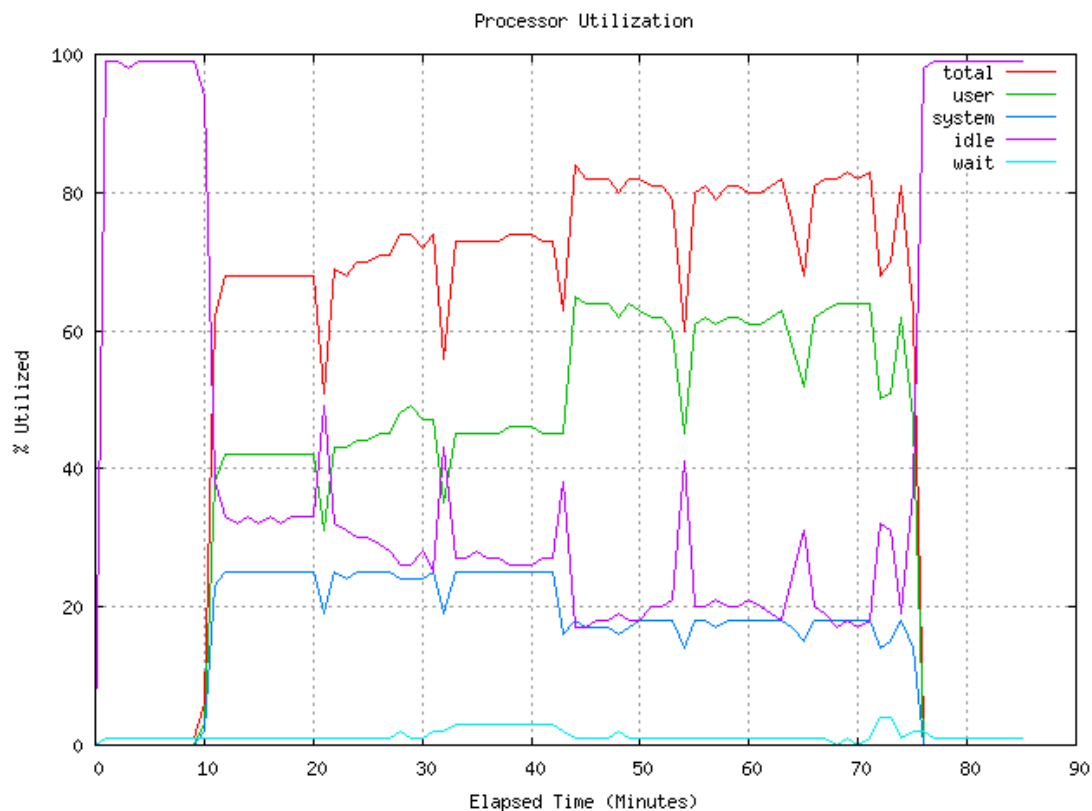
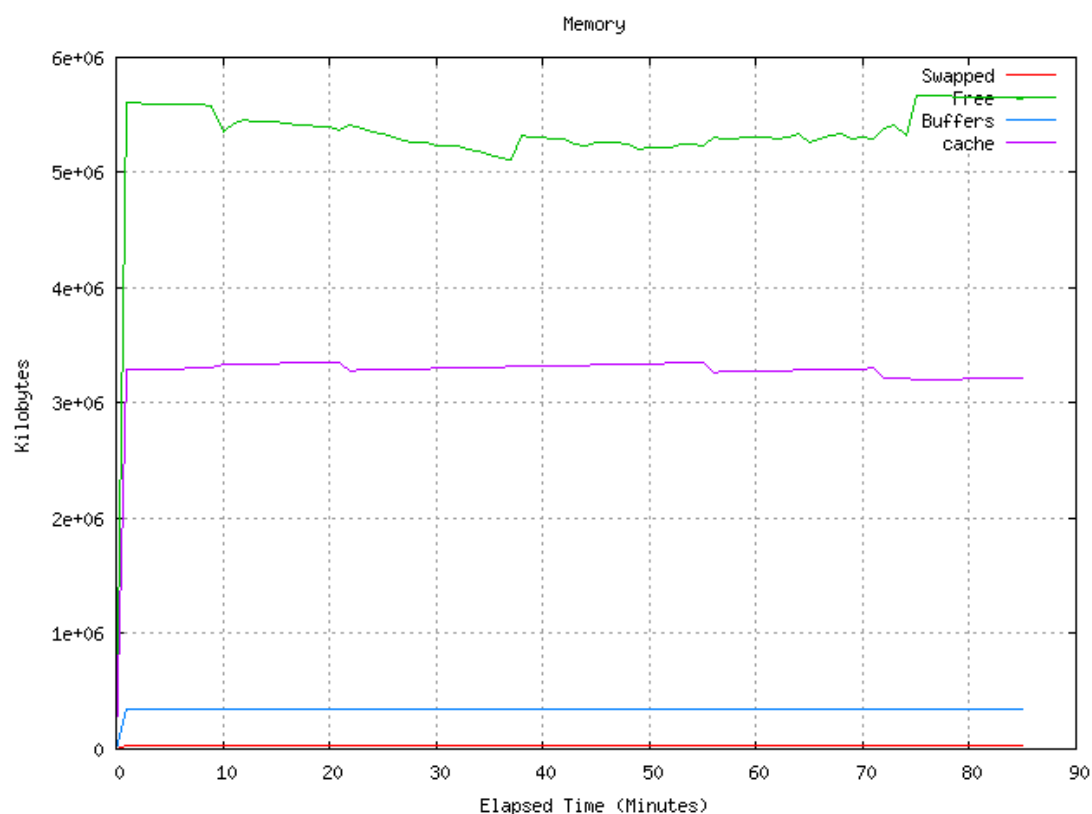


图 8-2 jdbc 直接连 mysql 跑 50 个线程时服务器 ddb-35 的 cpu 利用率



jdbc 直接连 mysql 跑 50 个线程时服务器 ddb-34 的内存占用率

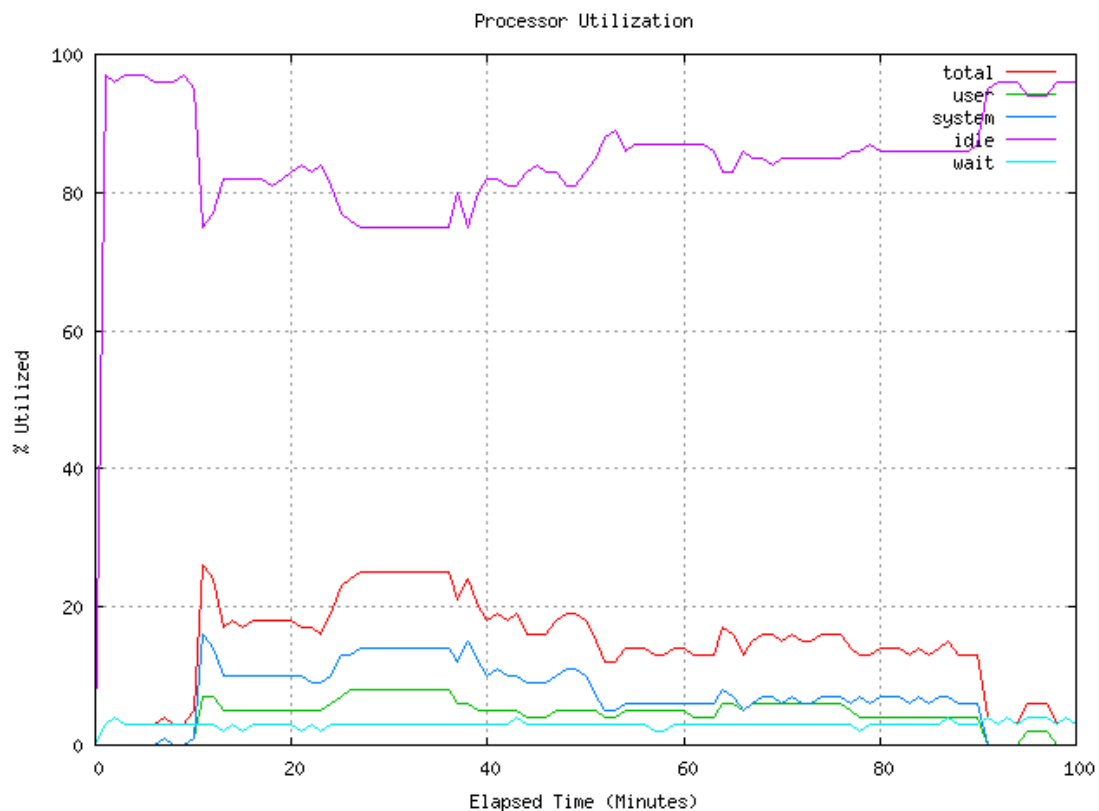


图 9-1 jdbc 直接连 mysql 跑 200 个线程时服务器 ddb-34 的 cpu 利用率

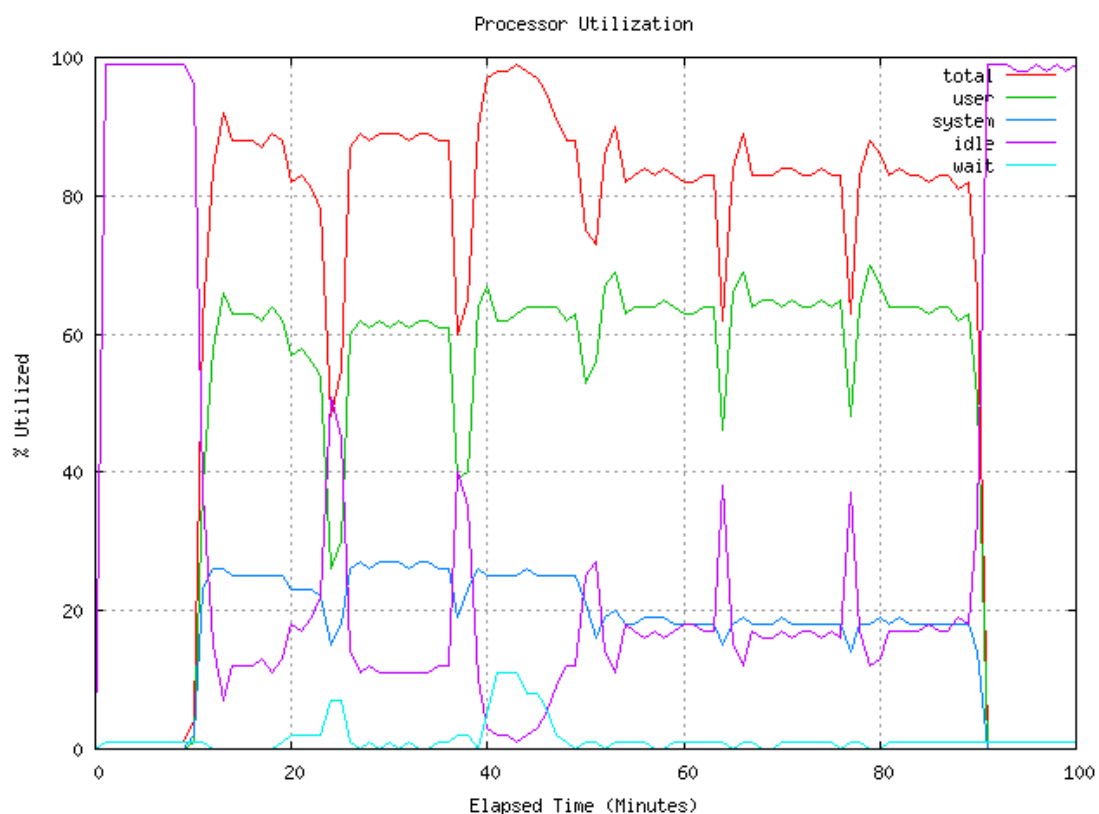
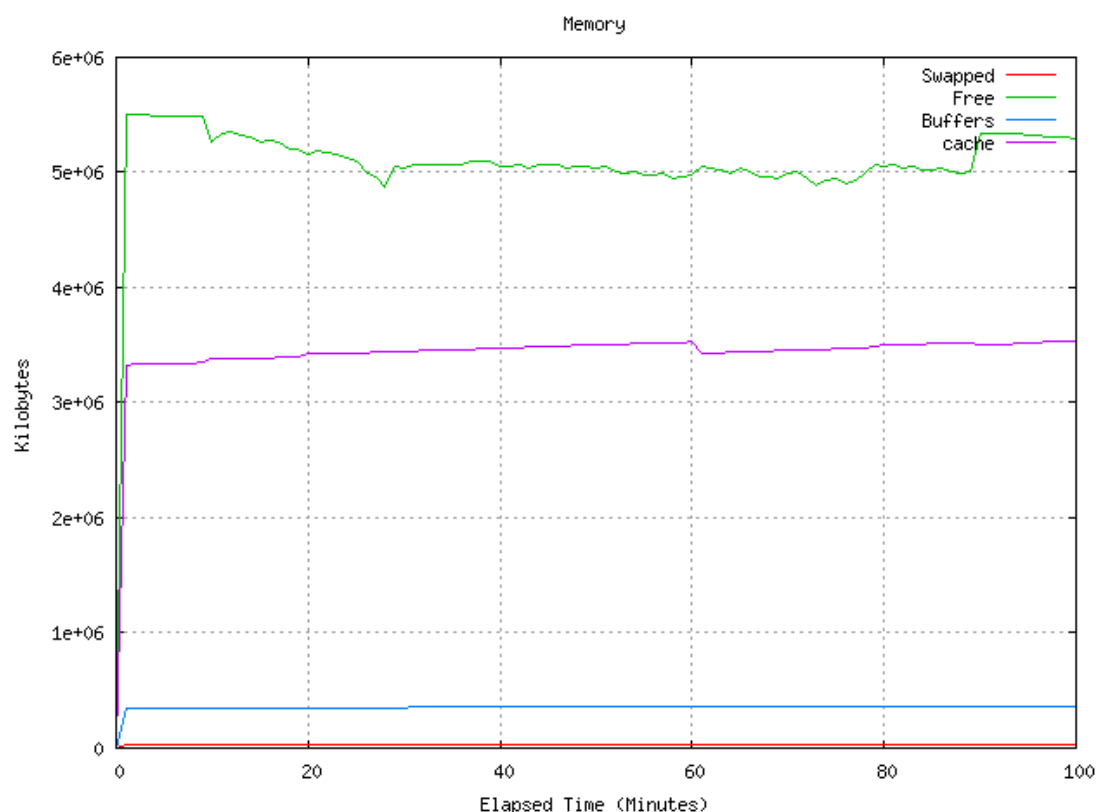


图 9-2 jdbc 直接连 mysql 跑 200 个线程时服务器 ddb-35 的 cpu 利用率



jdbc 直接连 mysql 跑 200 个线程时服务器 ddb-34 的内存占用率

20.5 测试结论

首先要明确的是，DDB 的设计初衷是为了能够实现数据库节点的线性性能提升，只要用户合理制定均衡策略，是完全能够实现数据库节点性能线性提升的。使用了 DDB 之后，相比于直接使用 JDBC 连接，对 MySQL 数据库来说，并不会产生任何额外影响，因此本次测试中数据库节点并不是重点关注对象。

测试数据说明，使用了 DDB 之后，客户端性能会有一定程度下降，但这究竟是不是应用在做系统选型时需要重点考虑的问题呢？这个问题可以这样看，例举测试数据中的 DDB 50 线程并发测试，在大约每秒产生 3 万次请求时，CPU 占用率大约 50%。如果用户线上运行时，每台应用服务器每秒发出的数据库请求远小于 3 万，基本可以不考虑 DDB 带来的额外代价。

对于客户端来说，我们重点关注 CPU 利用率和内存占用率，接下来分别讨论之。

20.5.1 DDB 与 JDBC 的 CPU 占用率对比

这次测试过程中，在高并发情况下，主要的系统性能瓶颈在数据库节点 CPU。为了公平进行 DDB 和 JDBC 的 CPU 耗用情况对比，我们主要查看单位吞吐量下 CPU 占用率这个指标，即 CPU 总占用率/吞吐量。

- 1) DDB 较 JDBC 开启事务时，单位吞吐量下 CPU 占用率提高
 - 5 个线程: statement: 99% prepare: 102% prepareReuse: 48%
 - 50 个线程: statement: 124% prepare: 98% prepareReuse: 56%
 - 200 个线程: statement: 117% prepare: 126% prepareReuse: 108%
- 2) DDB 较 JDBC 不开启事务时，单位吞吐量下 CPU 占用率提高
 - 5 个线程: statement: 181% prepare: 82% prepareReuse: 92%
 - 50 个线程: statement: 191% prepare: 175% prepareReuse: 140%
 - 200 个线程: statement: 176% prepare: 175% prepareReuse: 176%

注：为了公平起见，这里 DDB 的数据采集自开启非 xa 连接优化选项下的数据。

20.5.2 DDB 与 JDBC 的内存占用率

在线程数 50 线程以及 200 线程的情况下，DDB 的内存占用情况抖动较剧烈，会有 2G 左右大小的内存起伏。而相比之下，JDBC 在运行时就稳定的多，耗费内存稳定在 500M 左右，基本没有上下抖动发生。

20.5.3 Forwardby 语法的性能差别

语句使用 Forwardby 时，DDB 将不做语法解析，直接下发到节点执行，因此相比普通 SQL 性能有略微提升。

1) DDB 较 Forwardby，在开启事务情况下单位吞吐量下 CPU 占用率提高

5 个线程：statement: 10% prepare: 7% prepareReuse: 6%

50 个线程：statement: 6% prepare: 4% prepareReuse: 1%

200 个线程：statement: 14% prepare: 3% prepareReuse: 0%

2) DDB 较 Forwardby，在不开启事务情况下单位吞吐量下 CPU 占用率提高

5 个线程：statement: -11% prepare: -1% prepareReuse: -30%

50 个线程：statement: 21% prepare: 12% prepareReuse: 0%

200 个线程：statement: 23% prepare: 38% prepareReuse: 16%

20.5.4 使用非 xa 连接的优化选项

具体在什么时候可以开启非 xa 连接选项，以及开启这个选项后的作用，参见用户手册的相关章节介绍。DDB 在开启非 xa 优化选项后，系统整体吞吐量有明显提升：

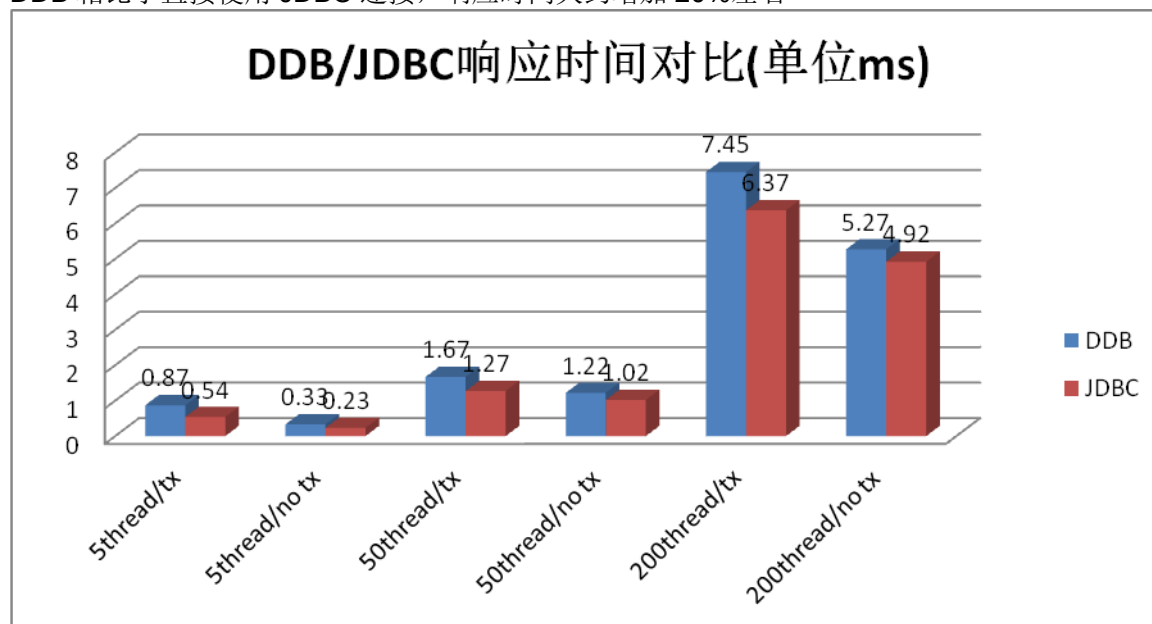
5 个线程：statement: 28% prepare: 25% prepareReuse: 25%

50 个线程：statement: 28% prepare: 25% prepareReuse: 23%

200 个线程：statement: 6% prepare: -8% prepareReuse: 10%

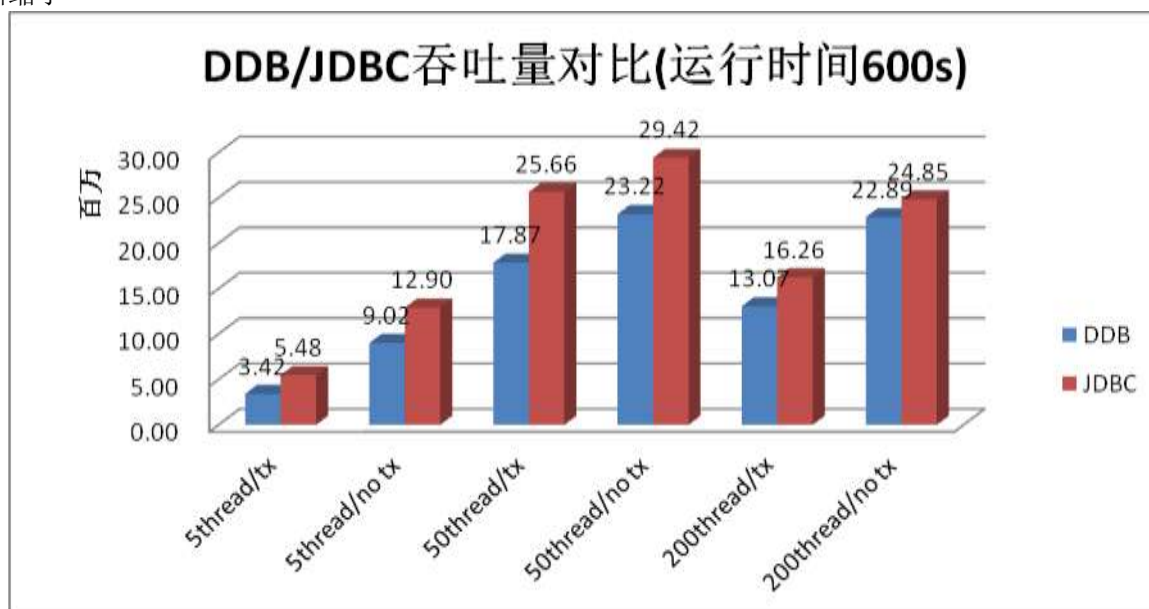
20.5.5 对比结论

DDB 相比于直接使用 JDBC 连接，响应时间大约增加 20%左右



DDB 相比于直接使用 JDBC 连接，吞吐量大约下降 30%左右，随着系统压力增大，差距

逐渐缩小



21 附录

21.1 附录 A—master 配置文件说明

DBClusterConf.xml 的参数包括用于 Master（管理服务器）和用于所有 Client 的两类参数。下面对这些参数分别进行说明。

21.1.1 用于 Master 配置参数

名称（标签）	类别	取值范围	默认值	说明
name	String		null	分布式数据库名称，如： netease
ip	String	非空	null	Master 服务器绑定的 ip 地址
port	int	>1000	8888	Master 服务器监听端口
dbn_hb_interval	long	>500	10000	DBN 心跳间隔时间，单位：毫秒
dbn_report_interval	long	<=0 表示不上报；>=2000	300000	上报 DBN 系统负载情况的时间间隔，单位：毫秒
dead_check_interval	long	>200	5000	检查 AS 和 DBN 心跳是否停止的时间间隔，单位：毫秒
dead_assure_interval	long	>5000	30000	当心跳间隔大于该时间时宣布 DBN 死亡，单位：毫

				秒
socket_timeout	int	1000~60000	10000	Socket 等待超时时间，单位：毫秒
connect_timeout	int	>=0	5000	连接服务器超时时间，单位：毫秒
mig_unit	int	10~10000	1000	数据迁移粒度，即一个迁移事务中迁移的最大主表记录数
xab_retry_times	int	0~10	3	Master 提交分支事务失败后重试提交的次数
xab_commit_interval	long	>1000	2000	Master 提交分支事务失败后两次重试提交的时间间隔，单位：毫秒
xa_check_interval	long	>=5000	30000	检查 DBN 超时悬挂事务的间隔时间，单位：毫秒
xab_timeout	long	>=10000	60000	分支事务悬挂超时时间，单位：毫秒。
sysdb_url	String	非空	null	sysdb 系统数据库 url
sysdb_user	String	非空	null	访问 sysdb 的用户名
sysdb_password	String		null	访问 sysdb 的口令
dba_mobile	String	手机号码列表，中间用分号分隔	空	用于短信报警的手机号码
sms_url	String			提供手机短信发送功能的服务地址
email_to_dba	String	电子邮件地址列表，中间用分号分隔	空	用于邮件报警的发送地址
email_cc_dba	String	电子邮件地址列表，中间用分号分隔	空	用于邮件报警的抄送地址
smtp_host	String		空	用于发送报警邮件的 smtp server
email_address	String	电子邮件地址	空	报警邮件发件人地址
email_password	String		空	和发件人地址对应的口令
pid	String	Master pid	Master.pid	用于标识 Master 已经正常

		文件的完整路径		启动
ddb_exec_dir	String		空	数据库节点机器上的 Bash 脚本位置，如：~/ddb/scripts/loadbalance
temp_dir	String		空	在做重建 slave 操作时，主节点上存放导出数据的临时目录，如：~/tmp
ibbackup_dir	String		空	在做重建 slave 操作时的 ibbackup 工具所在位置，如：~/ddb/scripts/loadbalance
innobackup_dir	String		空	在做重建 slave 操作时的 innobackup 工具所在位置，如：~/ddb/scripts/loadbalance
log_file	String		空	执行脚本时的日志文件，如：~/error.log
mirror_dir	String		空	镜像库机器上存放镜像数据库的目录，如：~/mirror
super_dbn_user	String		空	用于本地登录 mysql 的超级用户名
super_dbn_pass	String		空	用于本地登录 mysql 的超级用户密码
rep_dbn_user	String		空	用于 mysql 复制的用户名
rep_dbn_pass	String		空	用于 mysql 复制的密码
ssh_id_file	String		空	Ssh 登陆认证文件
check_rep_interval	String		空	复制延时诊断线程的检查间隔时间（秒），如：10
rep_fail_times	String		空	复制延时发现 slave 超时的重试次数，如：2
备注： 用红色标记的属性可以在系统运行时通过 DBA 工具进行修改，因此不需要在 DDBClusterConf.xml 文件中指定，这些参数的值将被保存在系统库中。				

21.1.2 用于所有 Client 的配置参数，对应标签<client>

名称（标签）	类别	取值范围	默认值	说明
buffer_size	int	1, 2, 4	4	日志 BUFFER 大小(单位 KB)

max_blocks_per_file	int	200, 400, 800	200	每个文件的最大日志块
log_file_dir	String	绝对路径	当前目录	日志文件的路径, 如 C:\lmgr
log_file_name	String	文件名	mylog	日志文件名称, 如最大文件数为 2, 日志文件就是 mylog_1.log, mylog_2.log
max_log_files	int	2~32	16	最多日志文件数。事务管理器采用循环日志, 日志的总大小 = max_log_files * max_blocks_per_file * blockSize
report_interval	long	>2000	30000	Client 向 Master 上报桶负载的时间间隔, 单位: 毫秒
use_daemon	boolean	True/false	true	Client 上线程是否以 daemon 方式运行
wait_conn_timeout	long	5000~60000	60000	连接等待超时时间
max_conns_per_pool	int	连接数	100	每个连接池的最大连接数
max_conns_per_xa_pool	int	XA 连接数	100	每个 XA 连接池的最大连接数
conn_idle_timeout	long	300000~600000	600000	连接空闲超时(单位: 毫秒)
max_pst_per_conn	int	50~150	100	每个连接最多缓存的 PREPAREDSTATEMENT 数
async_timeout	int	60~180	120	异步连接超时时间(单位: 秒)
async_interval	int	2~10	2	异步连接间隔(单位: 秒)
async_xidlist_max_size	int	100~200	100	CLIENT 端保存的异步 XID 数最大值
async_thread_interval	int	5000~15000	5000	异步线程唤醒时间(单位: 毫秒)
buffer_flush_interval	int	10~50	50	刷磁盘间隔时间(单位: 毫秒)
immediate_flush	boolean	true/false	false	立即刷新磁盘(低并发度应用建议采用)
exec_timeout	int	120	[5,28800] 5 秒到 8 小时	每条 SQL 语句执行或 COMMIT/ROLLBACK 时的最长允许时间,单位: 秒
备注: 用红色标记的属性可以在系统运行时通过 DBA 工具进行修改, 因此不需要在 DDBCusterConf.xml 文件中指定, 这些参数的值将被保存在系统库中。				

21.1.3 配置文件示例

```
<?xml version="1.0" encoding="UTF-8"?>

<cluster>

  <master>
    <name>ddb_demo</name>
    <ip>127.0.0.1</ip>
    <port>8888</port>
    <dba_port>7777</dba_port>
    <dbn_hb_interval>10000</dbn_hb_interval>
    <dbn_report_interval>300000</dbn_report_interval>
    <dead_check_interval>500000</dead_check_interval>
    <dead_assure_interval>60000</dead_assure_interval>
    <xab_check_interval>30000</xab_check_interval>
    <xab_timeout>60000</xab_timeout>
    <xab_retry_times>3</ xab_retry_times >
    <xab_interval>2000</xab_interval>
    <socket_timeout>3000</socket_timeout>
    <connect_timeout>5000</connect_timeout>
    <mig_unit>1000</mig_unit>
    <sysdb_url>jdbc:mysql://localhost:3306/sysdb_ddb_demo</sysdb_url>
    <sysdb_user>root</sysdb_user>
    <sysdb_password></sysdb_password>
    <pid>/home/db/master.pid</pid>
  </master>

  <client>
    <buffer_size>4</buffer_size>
    <max_blocks_per_file>200</max_blocks_per_file>
    <log_file_dir>log</log_file_dir>
    <log_file_name>howl</log_file_name>
    <max_log_files>16</max_log_files>
    <report_interval>1800000</report_interval>
    <use_daemon>true</use_daemon>
    <max_conns_per_pool>100</max_conns_per_pool>
    <max_conns_per_xa_pool>100</max_conns_per_xa_pool>
    <wait_conn_timeout>60000</wait_conn_timeout>
    <conn_idle_timeout>600000</conn_idle_timeout>
    <max_pst_per_conn>100</max_pst_per_conn>
    <async_timeout>120</async_timeout>
    <async_interval>2</async_interval>
    <async_xidlist_max_size>100</async_xidlist_max_size>
    <async_thread_interval>5000</async_thread_interval>
    <buffer_flush_interval>50</buffer_flush_interval>
    <immediate_flush>false</immediate_flush>
    <exec_timeout>120</exec_timeout>
  </client>

</cluster>
```

21.2 附录 B—建表语句注释格式说明

由于分布式数据库需要指定均衡字段、均衡策略等一般数据库中没有的信息，因此若需要通过 **CREATE TABLE** 语句指定分布式数据库的表定义（主要用于方便部署），则需要在 **CREATE TABLE** 语句中增加一些注释，否则就需要生成 **DBClusterSetup.xml** 文件描述。

注释要放在 **CREATE TABLE** 语句的最后，并使用“/*”和“*/”括起，即位置如下：

```
CREATE TABLE (
    // 字段定义、primary key 等
) /*注释在此*/;
```

注释由多个“属性=值”对组成，对于对之间使用空白分隔，系统支持的属性及值说明如下：

1. **bf=<均衡字段名>**：指定表的均衡字段，若指定多个均衡字段，则需要用括号括起，例如 **bf=(type, id)**。
2. **policy=<均衡策略名>**：指定均衡策略
3. **startid=<起始 ID>**：指定表的起始 ID，若不指定，则默认为 1
4. **remained=<剩余 ID>**：指定表的剩余 ID 数，若不指定，则默认为 64 位整数的最大值。
5. **model**：表所属于的模块名称，不能包含中文字符。（可不填）
6. **assignidtype**：ID 分配策略，有 2 种，**MSB(master 统一批量分配)**，**TSB(基于时间戳的 ID 分配)**。可不填，默认值为系统参数配置中所设置的默认 ID 分配策略。
7. **withPersist**：只针对 **MEMORY** 内存表起作用，表明此内存表需要持久化；若无此参数表明不持久化。持久化 **MEMORY** 表，需要创建一个对应的 **INNODB** 表，利用三个触发器同步 **MEMORY** 和 **INNODB** 表，并建立一个存储过程，用于在启动 **MySQL** 的时候从 **INNODB** 表读取数据到 **MEMORY** 表。（可不填）

以下是一个包含注释的建库脚本示例：

```
2. CREATE TABLE users (
3.     id INT(10) NOT NULL,
4.     name VARCHAR(30) NOT NULL,
5.     birth DATETIME NULL,
6.     bucketno SMALLINT,
7.     PRIMARY KEY (id),
8.     INDEX(name)
9. ) ENGINE = INNODB /*bf=id policy=user*/;
10.
11. CREATE TABLE score (
12.     id INT(10) NOT NULL,
13.     subject VARCHAR(30) NOT NULL,
14.     score INT(3) NOT NULL,
15.     date DATETIME NOT NULL,
16.     INDEX(id),
17.     FOREIGN KEY (id) REFERENCES users(id)
18. ) ENGINE = MYISAM /*bf=id policy=user*/;
```

上述建库脚本创建了两个表：**users** 和 **score**。**users** 和 **score** 都使用均衡策略 **user**，该均衡策略包含 10 个 **bucket**。**users** 表的均衡字段为 **id**，**score** 表的均衡字段为 **id**。

21.3 附录 C—调用脚本进行报警

DDB 发布目录中 **scripts/sendAlarm.sh** 是进行脚本报警时会调用的脚本(不支持 windows

平台),当发生某项报警且该项报警设置了通过脚本发送报警信息时,则会调用该脚本并将 JSON 格式的报警内容作为参数传递进去(\$1)。用户可根据需要扩展这个脚本,例如再调用 **python** 脚本或者 **Java** 程序都可以,但不能删除该脚本或者修改该脚本的名称。

DDB 会获取调用脚本的返回码和标准错误输出,用户可以在末尾定义返回码,如果返回码不为 0,则认为脚本执行失败, **master** 端记录相应的日志,并获取错误输出加入到日志中。脚本被错误删除或者其他方面的问题导致调用脚本失败,都会有相应的日志记录。

各项报警的 JSON 格式列表如下:

报警类型	JSON 格式说明	备注
sysdb 故障报警	<pre>{ "alarmType":1, "ddbName":"ddb_demo", "detail": { "sysdbUrl":"jdbc:mysql://172.21.0.35:4332/db35", "eventType":2 } }</pre>	eventType: 1- 系统库失效, 2- 系统库恢复
Master 故障报警	<pre>{ "alarmType":2, "ddbName":"ddb_demo", "detail": { "desc":"assign id failed." } }</pre>	
悬挂事务故障报警	<pre>{ "alarmType":4, "ddbName":"ddb_demo", "detail": { "dbns":[{"name":"dbn1","url":"jdbc:mysql://172.21.0.35:4332/dbn1"}, {"name":"dbn2","url":"jdbc:mysql://172.21.0.35:4332/dbn2"}] } }</pre>	
DBN 故障报警	<pre>{ "alarmType":5, "ddbName":"ddb_demo", "detail": { "dbns":[{"name":"dbn1","url":"jdbc:mysql://172.21.0.35:4332/dbn1"}, {"name":"dbn2","url":"jdbc:mysql://172.21.0.35:4332/dbn2"}], "eventType":2 } }</pre>	eventType: 1-dbn 失效, 2-dbn 恢复
slave 超时报警	<pre>{ "alarmType":10, "ddbName":"ddb_demo", "detail": { "dbns":[{ "delay":140, "name":"dbn2", </pre>	eventType: 1-slave 节点复制延时超时, 2-slave 节点复制延时恢复

	<pre> "url":"jdbc:mysql://172.21.0.35:4332/dbn2" }, { "delay":130, "name":"dbn1", "url":"jdbc:mysql://172.21.0.35:4332/dbn1" }], "eventType":1 } } </pre>	
计划任务报警	<pre> { "alarmType":11, "ddbName":"ddb_demo", "detail": { "planName":"plan1" } } </pre>	
slave 复制失败报警	<pre> { "alarmType":12, "ddbName":"ddb_demo", "detail": { "dbns":[{"name":"dbn1","url":"jdbc:mysql://172.21.0.35:4332/dbn1"}, {"name":"dbn2","url":"jdbc:mysql://172.21.0.35:4332/dbn2"}], "eventType":1 } } </pre>	eventType: 1-slave 节点复制失败, 2-slave 节点复制恢复
DBN Processlist 监控报警	<pre> { "alarmType":13, "ddbName":"ddb_demo", "detail": { "desc":"process list description" } } </pre>	
QS 监控报警	<pre> { "alarmType":14, "ddbName":"ddb_demo", "detail": { "qsList":["172.21.0.9:6000","172.21.0.10:6000"], "eventType":1 } } </pre>	eventType: 1- 查询服务器失效, 2- 查询服务器恢复
自动主从切换报警	<pre> { "alarmType":15, "ddbName":"ddb_demo", "detail": { "desc":"自动主从切换完成" } } </pre>	
在线改表	<pre> { "alarmType":16, </pre>	

报警	<pre> "ddbName":"ddb_demo", "detail": { "desc":"在线改表任务完成" } </pre>	
----	--	--

21.4 附录 D—系统库升级

21.4.1 DDB3 升级到 DDB4

本节主要说明由 DDB3 升级到 DDB4 的过程中，DDB 系统库所要进行的升级工作。具体操作步骤如下：

1. 关闭 master；
2. 将现有系统库导出一个备份，作为新系统库初始数据并启动；
3. 在新系统库上执行更新脚本 `update_sysdb.sql`；
4. 启动升级脚本 `./migsysdb.sh old_sysdb new_sysdb`。（sysdb 语法：`user[:password]@host:port/database`）；
5. 修改 `DBClusterConf.xml` 指向新系统库并启动 master。

至此，系统库升级完成。建议保留旧系统库，如果升级失败，可以切换回旧系统库。

21.4.2 DDB4.3 升级到 DDB4.4

DDB4.4 增加了对缓存的支持，由 DDB4.3 升级到 DDB4.4 需要对系统库进行升级，步骤如下：

1. 在系统库上执行 DDB4.4 的更新脚本 `update_sysdb.sql`；
2. 在系统库上执行 `UPDATE column_info SET field_number = number`；