# 概述

日志是记录了MySQL数据库的各种类型活动的数据。DBA可以利用这些日志文件定位故障，优化性能等。

物理日志：

1. InnoDB：redo log、undo log
2. Oracle：redo log、undo log
3. PostgreSQL：WAL（Write Ahead Log前写日志）

逻辑日志：binlog

# 分类

在MySQL中，有4种不同的日志，分别是：错误日志（errorlog）、二进制日志（binlog）、查询日志（log）和慢查询日志（slow query log）。



## 错误日志

### 概述

错误日志记录了当mysqld进程启动和停止时，以及服务器在运行过程中发生任何严重错误时的相关信息。

当数据库出现任何故障导致无法正常使用时，可以首先查看此日志。

### 设置

默认是开启的，可以通过修改my.cnf文件自定义，如：

log\_error=path默认存在$datadir/hostname.err

**查看：**

SHOW VARIABLES LIKE ‘log\_error’\G;

**路径设置：**

可以使用—log-error=[file\_name]选项来指定mysqld（MySQL服务器）保存错误日志文件的位置。如果没有给定file\_name值，mysqld使用错误日志名host\_name.err并默认在参数DATADIR（数据目录）指定的目录中写入日志文件。

在实际应用中，如果数据库启动报错或者crash，或者出现告警信息等，可以通过查看错误日志获取必要的信息。

## 二进制日志

### 概述

二进制文件记录了对MySQL数据库的所有更新操作（其中还包括执行更新操作的时间等额外信息），不包括查询和SHOW这类操作（binlog作用就是备份恢复使用的，所以只需要记录修改操作即可）。

二进制文件默认关闭，需要手动指定参数启动。根据MySQL官方手册的测试数据，开启二进制日志会使性能下降1%，但是考虑到可以使用复制（replication）和point-in-time的恢复，这些性能的损失绝对是可以接受的。

**查看：**

SHOW BINLOG EVENT IN ‘mysqld.00001’\G;

**配置：**

通过参数log-bin[=name]可以开启二进制日志，如果不指定name，默认二进制日志文件名为主机名，后缀名为二进制日志的序列号，所在路径为数据库所在目录（datadir）。

查看datadir：show variables like ‘datadir’;

### 参数

max\_binlog\_size：指定了单个二进制日志文件最大值，如果超过该值，则产生新的二进制日志文件后缀名+1，并记录到.index文件。

binlog\_cache\_size：控制缓冲大小，默认大小32K，基于会话的，因此每开启一个事务就分配一个binlog\_cache\_size大小的缓存，所以不能设置过大。当一个事务的记录大于binlog\_cache\_size时，MySQL会把缓冲中的日志写入一个临时文件中，因此该值又不能设置太小。

通过SHOW GLOBAL STATUS命令查看binlog\_cache\_use、binlog\_cache\_disk\_use的状态，可以判断当前binlog\_cache\_size的设置是否合理。binlog\_cache\_use记录了使用缓冲写二进制日志的次数，binlog\_cache\_disk\_use记录了使用临时文件写二进制日志的次数。

sync\_binlog：表示每写缓冲多少次就要同步到磁盘。如果设置为1，表示采用同步写磁盘的方式来写二进制日志，这时候写操作不使用操作系统的缓冲来写二进制日志。sync\_binlog的默认值为0，如果使用InnoDB存储引擎进行复制，并且想得到最大的可用性，建议将该值设置为ON（对数据库IO系统带来一定影响）。

但是，即使将sync\_binlog设置为1，还是会有一种情况导致问题发生。当使用InnoDB存储引擎时，在一个事务发出COMMIT动作之前，由于sync\_binlog为1，因此会将二进制日志立即写入磁盘。如果这时写入了二进制日志，但是提交还没有发生，并且此时发生了宕机，那么在MySQL数据库下次启动时，由于COMMIT操作并没有发生，这个事务被回滚。但是二进制日志已经记录了该事物信息，不能被回滚。这个问题可以通过参数innodb\_support\_xa设置为1来解决，虽然innodb\_support\_xa与XA事务有关，但是它同时也确保了二进制日志和InnoDB存储引擎文件的同步。

binlog-do-db：表示需要写入哪些库的日志，默认为空，表示需要同步所有库的日志到二进制日志。

binlog-ignore-db：表示需要忽略写入哪些库的日志，默认为空，表示需要同步所有库的日志到二进制日志。

log-slave-update：如果当前数据库是复制中的slave节点，则它不会将从master取得并执行的二进制日志写入自己的二进制文件中。如果需要写入，要设置log-slave-update。如果需要搭建master->slave->slave这种架构的复制，则必须设置该参数。

binlog\_format：记录二进制日志的格式。在MySQL5.1之前，没有这个参数，所有二进制文件的格式都是基于SQL语句（statement）级别的，因此基于这个格式的二进制日志文件的复制（Replication）和Oracle的逻辑Standby有点类似。

该值可以设置为STATEMENT、ROW和MIXED。

1. STATEMENT格式下，记录的是逻辑SQL语句。
2. ROW格式下，记录表的行更改情况。
3. MIXED格式下，MySQL默认采用STATEMENT格式进行二进制文件记录，但是在一些情况下会使用ROW格式，这些情况包括：
4. 表的存储引擎为NDB，这时对表的DML操作都会以ROW格式记录；
5. 使用了UUID()、USER()、CURRENT\_USER()、FOUND\_ROWS()、ROW\_COUNT()等不确定函数；
6. 使用了INSERT DELAY语句；
7. 使用了用户定义函数（UDF）；
8. 使用了临时表（temporary table）。

此外，binlog\_format参数还有对于存储引擎的限制：



在通常情况下，我们将参数binlog\_format设置为ROW，这可以为数据库的恢复和复制带来更好的可靠性。但是不能忽略的一点是，这会带来二进制文件大小的增加，这些语句的ROW格式可能需要更大的容量。而由于复制是采用传输二进制日志方式实现的，因此复制的网络开销也会增加。

要查看二进制日志文件内容，必须通过MySQL提供的工具mysqlbinlog。对于STATEMENT格式的二进制日志文件，在使用mysqlbinlog后，看到的就是执行的逻辑SQL语句。但是，如果使用ROW格式记录，会发现mysqlbinlog的结果变得“不可读”，其实只要加上参数-v或-vv就能清楚地看到执行的具体信息了（-vv会比-v显示更新的类型）。

### 作用

二进制日志的主要作用如下：

1. 恢复（recovery）：某些数据的恢复需要二进制日志，例如，如果需要恢复数据库全量备份的文件，可以通过二进制日志进行point-in-time恢复。
2. 复制（replication）：通过复制和执行二进制日志使得远程的MySQL数据库（一般称为slave或standby）与一台MySQL数据库（一般称为master或primary）进行实时同步。
3. 审计（audit）：用户通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入的攻击。

## 查询日志

### 概述

当客户端连接或断开时，服务器会将信息写入该日志，并记录从客户端收到的每一条SQL语句。当你怀疑客户端的错误并想知道客户端发送给mysqqld的确切消息时，一般查询日志可能非常有用。

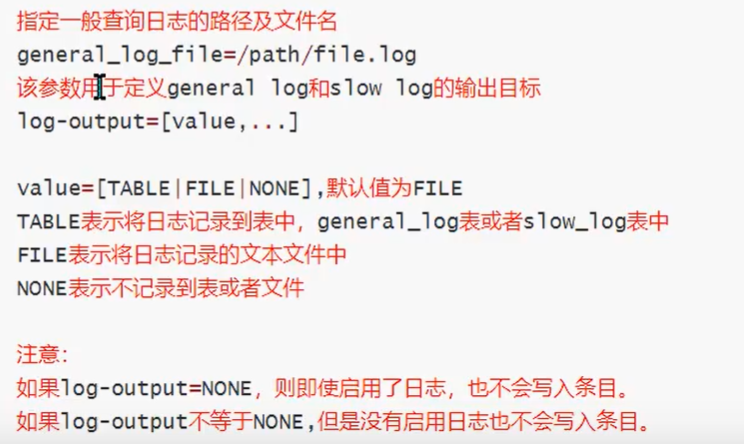
查询日志记录了所有对MySQL数据库的请求信息，不论这些请求是否得到正确的响应。

默认文件名为：主机名.log。

默认情况下，一般查询日志是被禁用的。如果需要开启，可以使用一下参数：

general\_log={0|1} //0表示禁用，1表示开启

默认情况下，系统会在数据目录下创建host\_name.log命令的一般查询日志。如果要自己指定，可以使用以下参数：



### 区别

**二进制日志与查询日志区别：**

二进制日志不包含只查询数据的语句，查询日志记录了客户端的所有语句。

## 慢查询日志

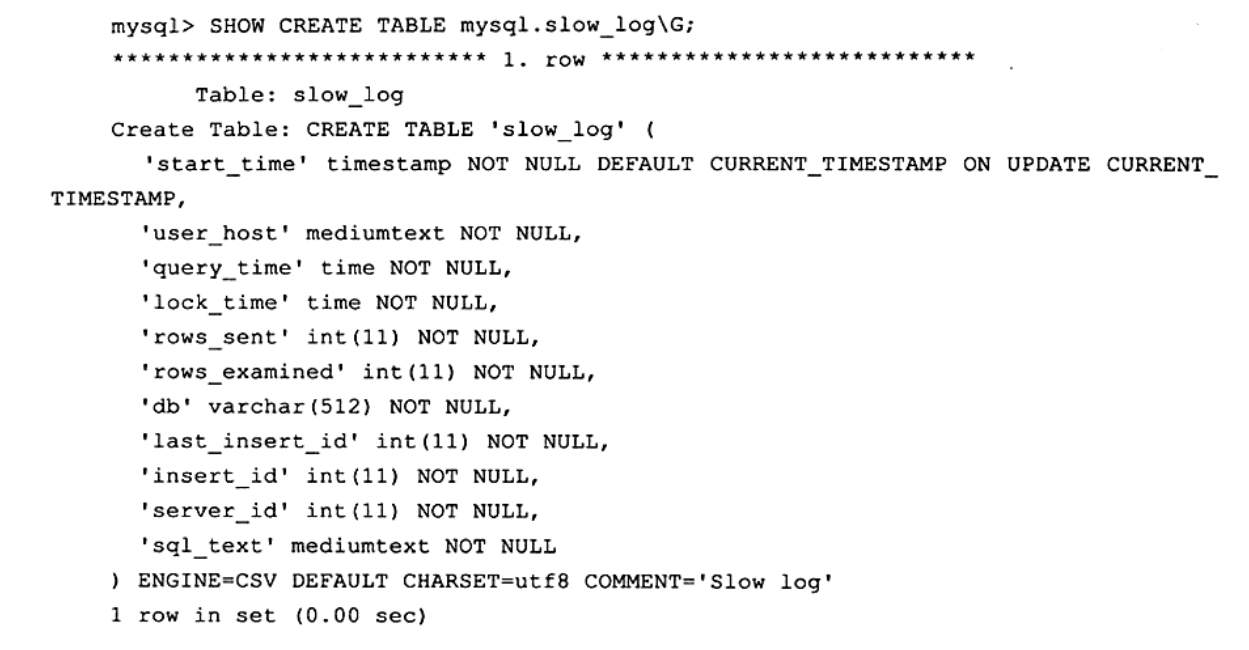
### 概述

慢查询日志记录了所有执行时间超过参数long\_query\_time（单位：秒，运行时间等于long\_query\_time的情况不会被记录）设置值并且扫描记录数不小于min\_examinied\_row\_limit的所有SQL语句的日志（注意，获得表锁定的时间不算做指定时间）。

慢查询日志默认关闭，需要手动开启。



MySQL5.1开始，支持将慢查询日志记录到表中，这样用户查询就更加方便直观了。慢查询表在MySQL架构下，名称为slow\_log，表结构定义如下：



### 参数

set slow\_query\_log：是否开启慢查询日志，通过set slow\_query\_log=on;开启。

long\_query\_time：阈值。long\_query\_time默认为10秒，最小为0，从MySQL5.1开始，精度可以到微秒。

min\_examinied\_row\_limit：允许扫描的最小行数。

另一个和慢查询日志相关的参数是long\_queries\_not\_using\_indexes，如果运行SQL未命中索引，则MySQL数据库同样会将这条SQL语句记录到慢查询日志中（这样方便后续优化）。

MySQL5.6新增一个参数log\_throttle\_queries\_not\_using\_indexes，用来表示每分钟允许记录到slow log且未命中索引的SQL语句次数。默认为0，表示没有限制。在生产环境中，如果没有命中索引，则此类SQL语句会频繁地被记录到slow log，从而导致慢查询日志大小不断增加，可通过修改此参数配置。

用户可以通过参数long\_query\_io将超过指定逻辑IO次数的SQL语句记录到slow\_log中，该值默认为100。

为了兼容原MySQL数据库的运行方式，还添加了参数slow\_query\_type，用来表示启用slow log的方式。，可选的值为：

0：表示不把SQL语句记录到slow log

1：表示根据运行时间将SQL语句记录到slow log

2：表示根据逻辑IO次数将SQL语句记录到slow log

3：表示根据运行时间及逻辑IO次数将SQL语句记录到slow log

### 分析

#### mysqldumpslow

如果在慢查询日志中搜索，则会非常不方便，分析起来很麻烦，MySQL提供了mysqldumpslow工具帮助分析慢查询日志。

指令：mysqldumpslow \*\*\*-slow.log

如果希望得到执行时间最长的5条SQL语句，操作如下：

mysqldumpslow -s -al -n 5 \*\*\*-slow.log

#### pt-query-digest

pt-query-digest是用于分析MySQL慢查询的一个工具，先对查询语句的条件进行参数化，然后对参数化以后的查询进行分组统计，统计出各查询的执行时间、次数、占比等，同时把分析结果输出到文件中，我们可以借助分析结果找出问题进行优化。

pt-query-digest是percona-toolkit工具包下的一个工具，如果MySQL是使用Linux\_5.7.22版本的bin包安装的话，默认会安装percona-toolkit，可以直接使用pt-query-digest命令来分析慢查询日志。

### 分布式数据库实践

分布式数据库的慢查询包括两种：计算节点proxy慢查询、数据节点DB慢查询和DB锁等待日志。

#### Proxy慢查询

Proxy慢查询开关、阈值、日志文件路径都可以通过配置文件设置。Proxy慢查询日志关键内容分析：

TotalExecTime：从应用接收SQL到返回应用响应的总时间

SQL：执行的SQL语句

MsgToExecTime：从消息线程发送到执行线程的时间

ParserSQLTime：解析SQL语法的时间

PlanTreeCreateTime：创建执行计划的时间

GetGTIDTime：获取GTID的时间

FreeGTIDTime：释放GTID时间

PlanTreeExecTime：执行计划树的执行时间

SubSQL[N]：第N条子查询

ExecTime：子查询的执行时间

FinishTime：proxy处理子查询的时间

g1 num:1,duration:104577us：执行子查询的某个分片的执行时间

DB connection\_id:2534578,duration:5467us：DB上执行时间

**第一部分：**

Port：proxy的连接实例端口

Session：proxy为当前客户端分配的dialogid

TransSerial：通过set @@transaction\_serial\_number语句为事务设置的流水号

LinkIP：前端客户端的IP

LinkPort：前端客户端的端口

UserName：dbproxy的连接实例用户

ProxyName：dbproxy的名称，与OMM页面一致

ClusterName：连接实例绑定的集群名称，与OMM页面一致

TotalExecTime：语句执行的总时间，以执行线程收到OS转发的语句消息为开始，以执行线程最终给客户端回响应时为结束

BeginTs：语句开始执行的时间，以执行线程收到OS转发的语句消息为开始

EndTs：语句执行结束的时间，以执行线程最终给客户端回响应时为结束

SQL：客户端下发的原始语句

MsgToExecTime：客户端语句从OS转发到执行线程的时间，执行线程消息积压在其中会有体现

ParserSQLTime：dbproxy做语法解析占用的时间

PlanTreeCreateTime：dbproxy创建执行计划的时间

GetGTIDTime：dbproxy申请GTID占用的时间，开始和结束统计的时间均在gtmproxy代理线程，申请GTID可能存在重试，但该时间只记录最后一次申请GTID的时间

FreeGTIDTime：dbproxy释放GTID占用的时间，开始和结束统计的时间均在gtmproxy代理线程，释放GTID可能存在重试，但该时间只记录最后一次释放GTID的时间

PlanTreeExecTime：执行计划树的总时间，以创建计划树结束为开始，计划树执行结束为结束

**第二部分：**

SubSQL[N]：执行计划中子语句的编号

ExecTime：执行计划中子语句的执行总耗时，开始和结束的时间统计均在执行线程，执行线程消息积压、路由线程消息积压、worker线程忙无法及时处理epoll事件以及查询活跃gtid重试耗时均会有体现

BeginTs[xxxx]：执行计划中子语句的开始执行时间

FinishTime：子语句执行结束，执行线程调用SQLNode的Finish方法的耗时

QueryGTID：查询活跃GTID的耗时，开始与结束时间在gtmproxy代理线程设置。如果存在查询活跃，该值记录最后一次查询活跃GTID的耗时。如果存在查询活跃，对应于子语句会多次执行，这会在每个group统计数据中的num及duration中有所体现，这些统计会做累加

GroupTIme：子语句在具体每个Group上的执行耗时分析

g1 num:1,duration:104577us：子语句在具体每个group上的执行耗时，以语句准备从执行线程发往路由线程为开始，以执行收到该group的响应为结束。如果该值与DB统计项的duratio相差较大，可能由于执行线程消息积压、路由线程消息积压、worker线程忙无法及时处理epoll事件导致

sqltoRoute num:xxxx,duration:xxxxus：语句从执行线程下发到路由，路由线程最终收到请求并处理的耗时。开始的时间统计在执行线程，结束时间统计在路由线程，路由线程消息积压会在该统计项中体现

getTask num:xxxx,duration:xxxxus：为SQL获取一个空闲task的时间，其中包括获取后端链路的时间：如果有空闲链路，则获取空闲链路；如果没有空闲链路，则包含调用mysql创建链路同步接口conn\_mysql\_real\_connect\_dyntime的时间

runSql num:xxxx,duration:xxxxus：调用mysql\_real\_query\_start接口将SQL下发给DB实例的耗时，注意该时间并非完全执行的时间，除非语句的执行结果能从接口函数的返回值全部返回，否则是异步过程

addEpoll num:xxxx,duration:xxxxus：mysql\_real\_query\_start是异步接口，如果语句执行响应不能一次执行，则需要将端口加入epoll进行监听。该统计项统计加入epoll事件的耗时。

DB connection\_id:2534578,duration:5467us,begin Ts:xxxx：connection\_id是和DB之间MySQL链路的thread\_id，duration是语句真正在DB的执行时间，这个事件比较大，说明语句确实在DB执行慢。BeginTs是理由下发语句到DB的开始时间。

handleEpoll num:xxxx,duration:xxxxus：listener线程监听到epoll事件后，将发生事件的socket分离的时间。

worker num:xxxx,duration:xxxxus：listener线程监听到epoll事件后，会将相应的task放入队列，并通过条件变量唤醒一个空闲worker线程，该统计项统计空闲worker线程被唤醒后，拿到task做后续处理的时间，一般是对语句执行结果做处理的时间。需要注意的是，当worker线程全忙，放入task队列的任务可能没有worker线程做及时的处理，等待空闲worker线程的时间不会再慢查询日志中记录。

restoExec num:xxxx,duration:xxxxus：路由线程或worker线程将语句执行结果处理结束后，给执行线程发消息，通知执行线程向客户端回响应。这部分统计开始时间在路由或worker线程，统计结束时间在执行线程，所以路由线程或执行线程消息积压会在这部分体现。

#### DB慢查询

##### 日志格式

# Time: 2018-12-18T05:55:15.941477Z

# User@Host: root[root] @ localhost [] Id: 53

# Query\_time: 2.000479 Lock\_time: 0.000000 Rows\_sent: 1 Rows\_examined: 0

#Req\_wait\_time:0.0 Pre\_dispatch\_time:0.000058 Parse\_time:0.000127 Execute\_time:0.019889

#Exec\_prep\_time=0.000009 open\_table\_time=0.006844 Mdl\_req\_time=0.00001 Innodblock\_req\_time:0.00000

#Order\_commit\_time:0.012669 Flush\_time:0.011767 Sync\_time:0.000044 Commit\_time:0.000044 Ack\_wait\_time:0.000005 Commit\_queue\_wait\_time:0.000000

SET timestamp=1560131835;

insert into t1 values(10);

**参考如下解释：**

# Time：当前打印的时间

# User@Host：当前用户

id：connection\_id

# Query\_time：查询总时间，从事件响应到执行结束的时间

Lock\_time：“锁”时间，mysql源码定义，时间从解析命令开始打获取到需要的锁的时间，也包括缓存表的时间等，并非单独获取锁的时间

Rows\_sent：返回记录数

Rows\_examined：扫描的行数

#Req\_wait\_time：线程池队列等待时间，单位为ms

Pre\_dispatch\_time：出线程池队列到命令开始解析的时间

Parse\_time：命令解析时间

Execute\_time：命令执行时间

#Exec\_prep\_time：执行前准备时间

open\_table\_time：打开缓存表和数据字典的时间，包含获取MDL锁时间

Mdl\_req\_time：获取MDL锁的时间

Innodblock\_req\_time：获取innodb锁的时间

#Order\_commit\_time：提交的总时间

Flush\_time：写binlog到进入缓存，sync redo落盘

Sync\_time：sync binlog落盘

Ack\_wait\_time：备机响应的时间

Commit\_time：存储引擎提交时间以及等备机响应

Commit\_queue\_wait\_time：提交队列中等待的时间

SET timestamp：表示当前的时间戳

insert into t1 values(10)：当前慢查询日志所执行的语句

##### 日志分析

###### 参数分析

1. #Time

该时间是当前打印的时间，使用该时间减去Query\_time，可以近似得到收到该SQL的时间。

1. id：connection\_id

该ID就是mysql的thread\_id，也就是链接ID，可以通过该ID在锁等待日志，binlog二进制日志，系统表（如innodb\_trx）中，根据该ID查询相关时刻的信息。

如可以根据该ID在锁等待日志innodb\_lock\_wait.log日志中搜索当前时刻同一个ID的锁等待日志，根据锁等待日志再做具体分析。

1. Rows\_sent：返回记录数 && Rows\_examined：扫描的行数

对于扫描行数过多而返回结果集的慢查询，首先肯定是查询SQL，然后可能存在两种场景：

1. 实际业务就是需要扫描那么多行的记录，这种无法优化
2. 实际业务可能不需要扫描过多记录，那么可以通过查询条件优化或者增加索引等，进行优化，避免过多的全表扫描
3. #Req\_wait\_time：线程池队列等待时间

如果Req\_wait\_time时间过长，那么很大可能是线程池队列发生了阻塞，线程池队列发生阻塞的可能性有两个：

1. 线程池配置thread\_pool\_size\*thread\_pool\_oversubscribe过小，导致大量并发来不及处理，该问题需要及时调整线程池大小；
2. 线程池配置能够满足90%以上的业务场景，但是某一批异常的业务中每一个SQL执行耗时非常长，导致线程池资源被长时间占用，来不及处理新的SQL。该问题可能是业务上的SQL很慢，具体可以根据该时刻其余的慢SQL。

注：DB存在线程池的监控日志，可以通过该日志确认线程池发生了阻塞。

1. open\_table\_time：打开缓存表和数据字典的时间，包含获取MDL锁时间

该阶段时间慢，可能在Mdl\_req\_time没有打印出长时间，但是在这个open\_table\_time阶段打印出了长时间。

具体原因可能是MDL锁时间消耗，也可能是打开缓存表的时间消耗。

目前遇到过的open\_table\_time阶段慢的问题，主要是truncate table和copy table引起的，可以通过一键诊断或者慢查询体制分析该时刻是否同步有两个以上的操作。

1. Mdl\_req\_time：获取MDL锁的时间

这个锁很明确就是MDL锁产生的时间间隔。

1. Innodblock\_req\_time：获取innodb锁的时间

这个阶段慢，主要就是innodb的行锁产生了锁等待冲突，可以进一步根据锁等待日志innodb\_lock\_wait.log排查请求事务和阻塞事务。

如果请求事务和阻塞事务都有gtm\_gtid，那么可以根据gtm\_gtid在相应的binlog中找到DML记录，进而对比请求的SQL和阻塞的SQL是否真的产生行锁冲突，冲突在哪一个主键记录、或者唯一索引记录上。

当前如果没有gtm\_gtid，也可以根据事物的thd\_id在binlog中查找，只不过要对时间点进行严格的校验，确保是同一个事务。

1. Flush\_time：写binlog进入缓存，sync redo落盘

Flush阶段慢的原因，基本上都是磁盘的IO较高导致的，可以查看该时刻是否IO较高，是否存在copy table命令等。

注：在某银行卡中心生产环境中每天凌晨2-5点左右flush\_time慢就是copy table导致的。

1. Sync\_time：sync binlog落盘

该阶段慢，主要原因是sync binlog落盘较慢，存在两种可能：

1. sync的binlog很大，导致sync时间过长
2. Binlog所在磁盘的IO产生了毛刺，导致sync动作较慢
3. Commit\_time：存储引擎提交时间以及备机响应

这个阶段时间慢，很大可能是等待备机响应慢，具体原因可能是：

1. 大的binlog事务，网络存在抖动；
2. 等待备机响应慢，其Ack\_wait\_time时间可能较大，也可能不大，不影响该问题的分析。

###### 主要影响因素

常见的影响慢查询的有：

1. 表锁时间较长
2. 扫描记录函数较多，如100万行记录
3. 线程池积压，队列等待时间过长
4. open\_table\_time时间较长
5. MDL锁等待时间较长，一般可能与DDL阻塞了
6. InnoDB行锁等待时间较长
7. Flush时间和sync时间较长，磁盘IO可能存在压力
8. Ack\_wait时间较长，可能是大事务，或者网络丢包，或者备机宕机等

###### 特殊场景

truncate table导致insert/update/select慢查询，但是根据慢查询分析找不到具体原因：

1. 对于insert的影响，主要是在sys\_mutex锁的影响
2. 对于update和select的影响，可能还是由于内部资源相关的锁导致的，具体原因还需分析

##### 问题分析

慢查询的具体表现：

1. 部分业务场景存在超时逻辑，如授权业务存在5秒超时，业务超时后，会主动断链
2. 迁移业务可能没有超时逻辑，业务侧可能只是简单反馈下业务慢，如之前业务正常运行只需要50min，现在突然需要80min，这种是什么原因？
3. GoldenDB系统上出现锁等待时间较长的告警，是什么原因？
4. 行方的监控系统统计的慢查询日志增长较高，是什么原因？

以上问题，归根结底都是慢查询导致。

###### 大事务引起的慢查询

1. 在A时间点业务反馈执行SQL语句超时，则在对应group的主DB节点查询慢查询日志，使用grep+时间戳进行过滤；
2. 通过proxy可以查看到链路报错的thread\_id，改thread\_id就是慢查询第二行的ID号，再根据thread\_id找到对应慢查询日志；
3. 查看慢查询日志，如果commit\_time时间较长，则继续查看同一个时间前后是否存在Ack\_wait\_time时间较长的慢查询，如果存在，则一定是主机等待备机时间较长引起的；
4. 继续查看A时刻binlog文件的大小，查看A时刻前后binlog文件有没有超过200M大小的文件，如果有，则一定是大事务导致的慢查询；
5. 最后解析binlog文件，根据每一个事务的pos信息，查看该binlog中大事务binlog多大，是什么类型的事务，此处可能是insert/update/delete事务，再确认操作的库表名，联系业务侧修改大事务的问题。

###### 等待备机响应过长(非大事务)引起的慢查询

1. 在A时间点业务反馈执行SQL语句超时，则在对应group的主DB节点查询慢日志，使用grep+时间戳过滤；
2. 通过proxy可以查看链路报错的thread\_id，该thread\_id就是慢查询第二行的ID，再根据thread\_id找到出问题对应的慢查询日志；
3. 查看慢查询日志，如果Commit\_time时间较长，则继续查看同一个时间前后是否存在Ack\_wait\_time时间较长的慢查询，如果存在，则一定是主机等待备机时间较长导致的；
4. 继续查看A时刻binlog文件的大小，查看A时刻前后binlog文件有没有超过200M大小的文件，如果没有，则一定不是大事务导致的慢查询；
5. 继续查看mysqld1.log错误日志，查看A时刻是否有semi\_sync插件超时或者高低水位切换的日志，如果有，则可能存在备机异常，导致等待备机响应时间较长；
6. 如果mysqld1.log错误日志在A时刻没有异常日志，则可能是网络丢包导致的，继续查看网络包是否异常，可以从行方的监控系统查看。

###### 线程池积压引起的慢查询

1. 在A时间点出现慢查询，查看DB上对应点的慢查询体制，发现如果Req\_wait\_time时间较长，达到总时间的50%以上，则有可能是DB线程池消息积压引起的慢查询；
2. 继续查看DB节点下的tool/db/threadpool.log监控记录，查看A时刻线程池的低优先级队列（Low\_queue\_events）是否存在积压，等待时间（wait\_timeout）是否较长；
3. 确认如果是线程池积压，则查看DB上的线程池配置（thread\_pool\_size\*thread\_pool\_oversubscribe），如果是配置不足，则可以动态修改调整大小，确保生效；
4. 确保如果非线程池配置问题，则查看A时刻是否存在大量时间较长的慢查询日志，比如存在大量的select大结果集语句，则可能是由于大量的select查询语句占用了较多的线程池资源，导致线程池存在积压。此时可以过滤出执行较长的SQL语句交由相应的业务进行分析，一般可能都是由于没有索引或者索引失效引起的问题。

###### copy table引起的慢查询

在copy table执行的阶段，会持有相关表的MDL锁，也会造成系统的IO很高，可能会对系统中的读写操作产生影响。

特殊地，copy table正常不会写入慢查询日志，除非设置log\_slow\_admin\_statements配置，该配置默认是关闭的，可以动态设置为开启，这样copy table等管理类SQL命令就会写入慢查询日志。

1. 如果是MDL锁引起的慢查询，则慢查询日志中的Mdl\_req\_time值一定比较大，占比在50%以上；
2. 如果是IO引起的慢查询，则具体分析IO过高引起慢查询；

一般情况下，行方在执行copy table的时候是没有业务下发的，因此如果copy table对业务产生了影响，则需要联系业务侧解决。

###### 行锁等待引起的慢查询

1. 慢查询日志中，如果是行锁等待时间过长的，则Innodblock\_req\_time值比较大，占比超过50%以上；
2. 同时需要查看锁等待日志，可以查看问题时刻，在锁等待日志中有没有相应的thread\_id的锁等待记录，锁等待记录req\_trx\_id表示请求锁事务的thread\_id，blk\_trx\_id表示阻塞锁事务的thread\_id；
3. 如果是锁等待引起的慢查询，则需要具体分析为什么事务会长时间持有锁记录，如果是业务逻辑，也可能是IO原因导致的SQL执行慢引起的。

###### Select查询记录过大引起的慢查询

慢查询日志中，如果发现是一个select语句，其Execute\_time时间较长，在更细的时间区间上没有长时间的阶段，且Row\_examine数值较大，如可能达到几十万，甚至百万级别，则该语句的慢查询是由于select查询扫描记录过大引起的。

###### IO过高引起的慢查询

1. 慢查询日志中，如果发现commit语句或者自动提交的insert/update/delete语句中的Flush\_time和Sync\_time时间较长，达到总时间的50%以上，则该SQL的慢查询是由于IO过高引起的，此时，可以通过监控平台查看该节点对应时刻的IO监控，是否存在IO过高；
2. 如果是IO过高，可以继续分析binlog日志和mysqld1.log错误日志，确实是由于copy table导致，还是由于大批量的数据导入导致的；

对于大批量的数据导入则可以通知业务侧减小导入并发，减轻数据库的IO压力。

###### open table时间过长引起的慢查询

###### 其他未知原因引起的慢查询

#### DB锁等待

##### 配置

DB锁等待日志配置：

innodb\_lock\_wait\_log=ON

innodb\_lock\_wait\_collect\_time=2000 --单位1ms

##### 分析

经过一次锁等待日志格式化后的锁等待日志格式如下：

[2020-06-10T01:00:00.1]||||0||||#WARN DESC=lock\_wait\_time:more than 200ms

req\_thd\_id:2, req\_trx\_id:123456, req\_trx\_seq:0, req\_gtm\_gtid:0, req\_sql:[update t1 set id=1 where name=’x’]

blk\_thd\_id:3, blk\_trx\_id:123457, blk\_trx\_seq:0, blk\_gtm\_gtid:0, blk\_key\_len:[211,2]

说明：

第一行：[时间戳]+固定格式+请求事务交易流水号+固定格式+#WARN DESC=+锁等待时间

第二行：

req\_thd\_id：请求事务thread\_id

req\_trx\_id：请求事务trx\_id

req\_trx\_seq：请求事务流水号，字符串最大长度32

req\_gtm\_gtid：请求事务gtm\_gtid，字符串最大长度32

req\_sql：请求事务SQL语句，字符串最大长度2048

第三行：

blk\_thd\_id：阻塞事务thread\_id

blk\_trx\_id：阻塞事务trx\_id

blk\_trx\_seq：阻塞事务流水号，字符串最大长度32

blk\_gtm\_gtid：阻塞事务gtm\_gtid，字符串最大长度32

blk\_key\_len：阻塞事务的索引键值，字符串最大长度256

# 分布式数据库实践

## TDSQL

## GoldenDB

### 系统日志

#### OMM操作日志

记录所有OMM下发的操作信息。

#### Moni日志

使用dbmoni命令时产生的日志，记录程序的启动/停止，日志文件为dbmoni.log，在安装用户的$HOME/log目录下。

#### 程序运行日志

程序运行日志在安装用户的$HOME/log目录下，日志文件达到100MB后自动切换到下一个文件，前一个文件会更名为“模块名-日期-序号.log”。

### 慢日志

#### Proxy慢日志

#### DB慢日志

#### DB锁等待日志

## TiDB

## OceanBase