

# MySQL 调优相关

## 1 SQL 慢查询相关

### 1.1 如何定位慢查询？

优化 SQL 的前提是能定位到慢 SQL，其方案有如下两种：

- 1) 查看慢查询日志，确定已经执行完的慢查询。
- 2) show processlist 查看正在执行的慢查询。

### 1.2 如何使用慢查询日志？

使用慢查询日志一般分为四步：

- 1) 开启慢查询日志（一般默认是关闭状态）
- 2) 设置慢查询阈值（响应速度是多长时间被定为是慢查询）
- 3) 确定慢查询日志路径（日志文件在哪里）
- 4) 确定慢查询日志的文件名（具体日志文件是哪个），然后对文件内容进行分析。

### 1.3 如何开启慢查询日志？

在 MySQL 命令行下输入下面的命令：

```
mysql> set global slow_query_log = on;  
  
Query OK, 0 rows affected (0.00 sec)
```

默认环境下，慢查询日志是关闭的。

### 1.4 如何设置慢查询的阈值？

设置慢查询时间阈值(响应时间是多长时间是慢查询)

```
mysql> set global long_query_time = 1;

Query OK, 0 rows affected (0.00 sec)
```

如果需要定位到慢查询，一般的方法是通过慢查询日志来查询的，MySQL 的慢查询日志用来记录在 MySQL 中响应时间超过参数 `long_query_time`（单位秒，默认值 10）设置的值并且扫描记录数不小于 `min_examined_row_limit`（默认值 0）的语句

## 1.5 long\_query\_time 的值如何确定呢？

线上业务一般建议把 `long_query_time` 设置为 1 秒，如果某个业务的 MySQL 要求比较高的 QPS，可设置慢查询为 0.1 秒。发现慢查询及时优化或者提醒开发改写。

一般测试环境建议 `long_query_time` 设置的阈值比生产环境的小，比如生产环境是 1 秒，则测试环境建议配置成 0.5 秒。便于在测试环境及时发现一些效率低的 SQL。

甚至某些重要业务测试环境 `long_query_time` 可以设置为 0，以便记录所有语句。并留意慢查询日志的输出，上线前的功能测试完成后，分析慢查询日志每类语句的输出，重点关注 `Rows_examined`（语句执行期间从存储引擎读取的行数），提前优化。

## 1.6 如何知道慢查询日志路径？

慢查询日志的路径默认是 MySQL 的数据目录

```
mysql> show global variables like 'datadir';
```

Variable_name	Value
datadir	/data/mysql/data/3306/

```
1 row in set (0.00 sec)
```

## 1.7 如何知道慢查询日志的文件名？

```
mysql> show global variables like 'slow_query_log_file';
```

```
+-----+-----+
| Variable_name | Value          |
+-----+-----+
| slow_query_log_file | mysql-slow.log |
+-----+-----+
```

```
1 row in set (0.00 sec)
```

打开日志文件，可以对日志文件中的内容进行分析，常用选项说明：

Time: 慢查询发生的时间

User@Host: 客户端用户和 IP

Query\_time: 查询时间

Lock\_time: 等待表锁的时间

Rows\_sent: 语句返回的行数

Rows\_examined: 语句执行期间从存储引擎读取的行数

说明，后续也可以使用 pt-query-digest 或者 mysqldumpslow 等工具对慢查询日志进行分析。

## 1.8 如何查看正在运行的慢 SQL?

有时慢查询正在执行，已经导致数据库负载偏高了，而由于慢查询还没执行完，因此慢查询日志还看不到任何语句。此时可以使用 show processlist 命令判断正在执行的慢查询。show processlist 显示哪些线程正在运行。如果有 PROCESS 权限，则可以看到所有线程。否则，只能看到当前会话的线程。

还有，如果不使用 FULL 关键字，在 info 字段中只显示每个语句的前 100 个字符，如果想看语句的全部内容可以使用 full 修饰 (show full processlist)。

## 1.9 如何对慢查询进行分析?

工欲善其事，必先利其器”，分析慢查询可以通过 explain、show profile 和 trace 等工具来实现。

## 1.10 如何使用 profile 分析慢查询

在 MySQL 数据库中，通过 profile，能够更清楚地了解 SQL 执行过程的资源使用情况，能让我们知道到底慢在哪个环节。大致使用步骤是：确定这个 MySQL 版本是否支持 profile；确定 profile 是否关闭；开启 profile；执行 SQL；查看执行完 SQL 的

query id; 通过 query id 查看 SQL 的每个状态及耗时时间。

### 第一步：确定是否支持 profile

我们进行第一步，用下面命令来判断当前 MySQL 是否支持 profile:

```
mysql> select @@have_profiling;
```

```
+-----+
| @@have_profiling |
+-----+
| YES              |
+-----+
```

1 row in set, 1 warning (0.00 sec)

从上面结果中可以看出是 YES，表示支持 profile 的。

### 第二步：查看 profiling 是否关闭的

进行第二步，用下面命令判断 profiling 参数是否关闭（默认 profiling 是关闭的）:

```
mysql> select @@profiling;
```

```
+-----+
| @@profiling |
+-----+
| 0           |
+-----+
```

1 row in set, 1 warning (0.00 sec)

结果显示为 0，表示 profiling 参数状态是关闭的。

### 第三步：通过 set 开启 profile

```
mysql> set profiling=1;
```

Query OK, 0 rows affected, 1 warning (0.00 sec)

Tips: set 时没加 global，只对当前 session 有效。

该参数开启后，后续执行的 SQL 语句都将记录其资源开销，如 IO、上下文切换、CPU、Memory 等等。根据这些开销进一步分析当前 SQL 从而进行优化与调整。

### 第四步：执行 SQL 语句

```
mysql> select * from t1 where b=1000;
```

### 第五步：确定 SQL 的 query id

通过 show profiles 语句确定执行过的 SQL 的 query id:

```
mysql> show profiles;
```

```
+-----+-----+-----+
| Query_ID | Duration | Query |
+-----+-----+-----+
| 1        | 0.00063825 | select * from t1 where b=1000 |
+-----+-----+-----+
```

1 row in set, 1 warning (0.00 sec)

## 第六步：查询 SQL 执行详情

通过 show profile for query 可看到执行过的 SQL 每个状态和消耗时间：

```
mysql> show profile for query 1;
```

Status	Duration
starting	0.000115
checking permissions	0.000013
Opening tables	0.000027
init	0.000035
System lock	0.000017
optimizing	0.000016
statistics	0.000025
preparing	0.000020
executing	0.000006
Sending data	0.000294
end	0.000009
query end	0.000012
closing tables	0.000011
freeing items	0.000024
cleaning up	0.000016

15 rows in set, 1 warning (0.00 sec)

通过以上结果，可以确定 SQL 执行过程具体在哪个过程耗时比较长，从而更好地进行 SQL 优化与调整。

## 2 SQL 索引相关

### 2.1 你了解哪类查询可能不走索引的情况？

- 1) 查询条件有隐式转换。（假如字段 id 为 int 类型，但是查询条件中写的是 id='10001'）
- 2) like 查询以%开头。
- 3) 范围查询时，包含的数据量太大。
- 4) 对条件字段做运算及函数操作。（year(hire\_date)='1999'）

### 2.2 MySQL 有哪些排序方式

按照排序原理分，MySQL 排序方式分两种：

- 1) 通过有序索引直接返回有序数据
- 2) 通过 Filesort 进行排序（又分为内存排序和磁盘排序）

我们可以使用 explain 来查看该排序 SQL 的执行计划，重点关注 Extra 字段。如果该字段里显示是 Using index，则表示是通过有序索引直接返回有序数据，如果该字段里显示是 Using filesort，则表示该 SQL 是通过 Filesort 进行的排序。

其中，MySQL 中的 Filesort 并不一定是在磁盘文件中进行排序的，也有可能在内存中排序，内存排序还是磁盘排序取决于排序的数据大小和 sort\_buffer\_size 配置的大小。

如果 “排序的数据大小” < sort\_buffer\_size: 内存排序

如果 “排序的数据大小” > sort\_buffer\_size: 磁盘排序

## 2.3 如何对 Order by 语句进行优化

- 1) 假如是单个字段，直接添加索引。
- 2) 假如是多个字段，对多个字段按使用顺序添加组合索引(复合索引)。
- 3) 对于先等值查询再排序的语句，可以通过在条件字段和排序字段添加联合索引来优化此类排序语句。
- 4) 去掉 select 列表中不需要返回的字段。
- 5) 修改数据库参数值(例如 sort\_buffer\_size,max\_length\_for\_sort\_data)

对于 group by 语句的优化，如果只要分组，没有排序需求的话，可以加 order by null 禁止排序。

## 2.4 如何对 MySQL 的分页查询进行优化

让排序时返回的字段尽可能少，所以可以让排序和分页操作先查出主键，然后根据主键查到对应的记录，例如：

```
select * from t1 f inner join (select id from t1 order by id limit 99000,2)g on f.id = g.id;
```

可通过如下表进行实验：

```
use test;                                /* 使用 test 这个 database */
drop table if exists t1;                  /* 如果表 t1 存在则删除表 t1 */

CREATE TABLE `t1` (                      /* 创建表 t1 */
  `id` int(11) NOT NULL auto_increment,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
```

```

`create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT
'记录创建时间',
`update_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP COMMENT '记录更新时间',
PRIMARY KEY (`id`),
KEY `idx_a` (`a`),
KEY `idx_b` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

```

drop procedure if exists insert_t1; /* 如果存在存储过程 insert_t1, 则
删除 */
delimiter ;; /* 设置分隔符为;;, 下一次遇到分隔符则执行语句 */

create procedure insert_t1()          /* 创建存储过程 insert_t1 */
begin
    declare i int;                    /* 声明变量 i */
    set i=1;                          /* 设置 i 的初始值为 1 */
    while(i<=100000)do                /* 对满足 i<=100000 的值进行 while
循环 */
        insert into t1(a,b) values(i, i); /* 写入表 t1 中 a、b 两个字段, 值都
为 i 当前的值 */
        set i=i+1;                    /* 将 i 加 1 */
    end while;
end;;

delimiter ;                          /* 创建批量写入 100000 条数据到表 t1 的存储过程
insert_t1 */
call insert_t1();                     /* 运行存储过程 insert_t1 */

```

## 2.5 如何进行 Join 优化?

- 1) 关联字段加索引
- 2) 小表驱动大表
- 3) 使用临时表

说明:

有时因为某条关联查询只是临时查一次, 如果再去添加索引可能会浪费资源, 那么有什么办法优化呢?

创建临时表, 把驱动表数据放到临时表, 然后在临时表中的关联字段上添加索引, 然后通过

临时表来做关联查询。

## 3 锁应用相关(作业)

### 3.1 为什么使用锁？

MySQL 中，锁就是协调多个用户或者客户端并发访问某一资源的机制，保证数据并发访问时的正确性。

### 3.2 MySQL 中的锁是如何分类的？

根据加锁的范围，MySQL 中的锁可分为三类：

- 1) 全局锁
- 2) 表级锁
- 3) 行锁

### 3.3 全局锁如何理解及应用

MySQL 全局锁会关闭所有打开的表，并使用全局读锁（简称：FTWRL）锁定所有表。其命令为：

```
FLUSH TABLES WITH READ LOCK;
```

可以使用下面命令解锁：

```
UNLOCK TABLES;
```

当执行 FTWRL 后，所有的表都变成只读状态，数据更新或者字段更新将会被阻塞。

案例分析：



session1	session2
FLUSH TABLES WITH READ LOCK; Query OK, 0 rows affected (0.00 sec)	
select * from t14 limit 1; ... 1 row in set (0.00 sec) (能正常返回结果)	select * from t14 limit 1; ... 1 row in set (0.00 sec) (能正常返回结果)
insert into t14(a,b) values(2,2); ERROR 1223 (HY000): Can't execute the query because you have a conflicting read lock (报错)	insert into t14(a,b) values(2,2);/* sql1 */ (等待)
UNLOCK TABLES;	insert into t14(a,b) values(2,2);/* sql1 */ Query OK, 1 row affected (5.73 sec) ( <b>session1</b> 解锁后, 在等待的 <b>sql1</b> 马上执行成功)

全局锁一般用在整个库（包含非事务引擎表）做备份（例如 mysqldump）时。也就是说，在整个备份过程中，整个库都是只读的，其实这样风险挺大的。如果是在主库备份，会导致业务不能修改数据；而如果是在从库备份，就会导致主从延迟。

好在 mysqldump 包含一个参数 --single-transaction，可以在一个事务中创建一致性快照，然后进行所有表的备份。因此增加这个参数的情况下，备份期间可以进行数据修改。但是需要所有表都是事务引擎表。所以这也是建议使用 InnoDB 存储引擎的原因之一。

### 3.4 表锁是如何应用的？

其中表锁又分为表读锁和表写锁，命令分别是：

表读锁：

```
lock tables t14 read;
```

表写锁：

```
lock tables t14 write;
```

其中：

对表执行 lock tables xxx read （表读锁）时，本线程和其它线程可以读，本线程写会报错，其它线程写会等待。

对表执行 lock tables xxx write （表写锁）时，本线程可以读写，其它线程读写都会阻塞。

案例分析：

表读锁分析

session1	session2
lock tables t14 read; Query OK, 0 rows affected (0.00 sec)	
select id,a,b from t14 limit 1; ... 1 row in set (0.00 sec) (能正常返回结果)	select id,a,b from t14 limit 1; ... 1 row in set (0.00 sec) (能正常返回结果)
insert into t14(a,b) values(3,3); ERROR 1099 (HY000): Table 't14' was locked with a READ lock and can't be updated (报错)	insert into t14(a,b) values(3,3);/* sql2 */ (等待)
unlock tables; Query OK, 0 rows affected (0.00 sec)	insert into t14(a,b) values(3,3);/* sql2 */ Query OK, 1 row affected (10.97 sec) ( <b>session1</b> 解锁后， <b>sql2</b> 立马写入成功)

表写锁分析：

session1	session2
lock tables t14 write; Query OK, 0 rows affected (0.00 sec)	
select id,a,b from t14 limit 1; ... 1 row in set (0.00 sec) (能正常返回结果)	select id,a,b from t14 limit 1;/* sql3 */ (等待)
unlock tables; Query OK, 0 rows affected (0.01 sec)	select id,a,b from t14 limit 1;/* sql3 */ ... 1 row in set (7.16 sec) ( <b>session1</b> 解锁后, <b>sql3</b> 马上返回查询结果)
lock tables t14 write; Query OK, 0 rows affected (0.00 sec)	
delete from t14 limit 1; Query OK, 1 row affected, 1 warning (0.00 sec) (能正常执行删除语句)	delete from t14 limit 1;/* sql4 */ (等待)
unlock tables; Query OK, 0 rows affected (0.00 sec)	delete from t14 limit 1;/* sql4 */ Query OK, 1 row affected, 1 warning (14.94 sec) ( <b>session1</b> 解锁后, <b>sql4</b> 立马执行成功)

表锁使用场景：

事务需要更新某张大表的大部分或全部数据。

事务涉及多个表, 比较复杂, 可能会引起死锁, 导致大量事务回滚, 可以考虑表锁避免死锁。

### 3.5 如何理解元数据锁？

在 MySQL 中, DDL 是不属于事务范畴的。如果事务和 DDL 并行执行同一张表时, 可能会出现事务特性被破坏、binlog 顺序错乱等 bug。为了解决这类问题, 从 MySQL 5.5.3 开始, 引入了元数据锁 (Metadata Locking, 简称: MDL 锁)

从上面我们知道，MDL 锁的出现解决了同一张表上事务和 DDL 并行执行时可能导致数据不一致的问题。

如下例：

session1	session2	session3
select id,a,b,sleep(100) from t14 limit 1;/* sql5 */		
	alter table t14 add column c int;/* sql6 */ (等待)	select id,a,b from t14 limit 1;/* sql7 */ (等待)
select id,a,b,sleep(100) from t14 limit 1;/* sql5 */ ... 1 row in set (1 min 40.00 sec) (100 秒后 sql5 返回 结果)	alter table t14 add column c int;/* sql6 */ Query OK, 0 rows affected (1 min 33.98 sec) Records: 0 Duplicates: 0 Warnings: 0 (session1 的查询语 句执行完成后，sql6 立马执行完毕)	select id,a,b from t14 limit 1;/* sql7 */ ... 1 row in set (1 min 26.65 sec) (session1 的查询语 句执行完成后，sql7 立马执行完毕)

上面的实验中，我们在 session1 查询了表 t14 的数据，其中使用了 sleep(100)，表示在 100 秒后才会返回结果；然后在 session2 执行 DDL 操作时会等待（原因是 session1 执行期间会对表 t14 加一个 MDL，而 session2 又会跟 session1 争抢 MDL）；而 session3 执行查询时也会继续等待。因此如果 session1 的语句一直没结束，其它所有的查询都会等待。这种情况下，如果这张表查询比较频繁，很可能短时间把数据库的连接数打满，导致新的连接无法建立而报错，如果是正式业务，影响是非常恐怖的。

当然如果出现这种情况，假如你还有 session 连着数据库，可以 kill 掉 session1 中的语句或者终止 session2 中的 DDL 操作，可以让业务恢复。但是出现这种情况的根源其实是：session1 中有长时间未提交的事务。因此对于开发来说，在工作中应该尽量避免慢查询、尽量保证事务及时提交、避免大事务等，当然对于 DBA 来说，也应该尽量避免在业务高峰执行 DDL 操作。

### 3.6 如何理解 MySQL 中的行锁？

MySQL 5.5 之前的默认存储引擎是 MyISAM，5.5 之后改成了 InnoDB。InnoDB 后来居上最主要的原因就是：

InnoDB 支持事务：适合在并发条件下要求数据一致的场景。

InnoDB 支持行锁：有效降低由于删除或者更新导致的锁定。

InnoDB 实现了以下两种类型的行锁：

共享锁 (S)：允许一个事务去读一行，阻止其它事务获得相同数据集的排他锁；

排他锁 (X)：允许获得排他锁的事务更新数据，阻止其它事务取得相同数据集的共享读锁和排他写锁。

对于普通 select 语句，InnoDB 不会加任何锁，事务可以通过以下语句显式给记录集加共享锁或排他锁：

共享锁 (S)：select \* from table\_name where ... lock in share mode;

排他锁 (X)：select \* from table\_name where ... for update。

### 3.7 你了解事务的哪些隔离级别？

MySQL 的 4 种隔离级别：

Read uncommitted (读未提交)：在该隔离级别，所有事务都可以看到其它未提交事务的执行结果。可能会出现脏读。

Read Committed (读已提交，简称：RC)：一个事务只能看见已经提交事务所做的改变。因为同一事务的其它实例在该实例处理期间可能会有新的 commit，所以可能出现幻读。

Repeatable Read (可重复读，简称：RR)：这是 MySQL 的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。消除了脏读、不可重复读，默认也不会出现幻读。

Serializable (串行)：这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。

这里解释一下脏读和幻读：

脏读：读取未提交的事务。

幻读：一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据。

## 4 总结(Summary)

### 4.1 参考

《深入浅出 MySQL》（第 2 版）