MySQL 优化利器 SHOW PROFILE 的实现原理

首页推荐

原创 陈臣 2024-12-23 150

背景

最近碰到一个 case，通过可传输表空间的方式导入一个 4GB 大小的表，耗时 13 分钟。

通过 `PROFILE` 定位，发现大部分耗时竟然是在 `System lock` 阶段。

```
mysql> set profiling=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> alter table sbtest2 import tablespace;
Query OK, 0 rows affected (13 min 8.99 sec)

mysql> show profile for query 1;
+------------------------------+------------+
| Status                       | Duration   |
+------------------------------+------------+
| starting                     |   0.000119 |
| Executing hook on transaction |   0.000004 |
| starting                     |   0.000055 |
| checking permissions         |   0.000010 |
| discard_or_import_tablespace |   0.000007 |
| Opening tables               |   0.000156 |
| System lock                  | 788.966338 |
| end                          |   0.007391 |
| waiting for handler commit   |   0.000041 |
| waiting for handler commit   |   0.011179 |
| query end                    |   0.000022 |
| closing tables               |   0.000019 |
| waiting for handler commit   |   0.000031 |
| freeing items                |   0.000035 |
| cleaning up                  |   0.000043 |
+------------------------------+------------+
15 rows in set, 1 warning (0.03 sec)
```

不仅如此，SQL 在执行的过程中，`show processlist` 中的状态显示的也是 `System lock`。

```
mysql> show processlist;
+----+-----------------+-----------+--------+---------+------+-----------------------+----
| Id | User            | Host      | db     | Command | Time | State                 | Inf
+----+-----------------+-----------+--------+---------+------+-----------------------+----
|  5 | event_scheduler | localhost | NULL   | Daemon  |  818 | Waiting on empty queue | NUL
| 10 | root            | localhost | sbtest | Query   |  648 | System lock           | alt
| 14 | root            | localhost | NULL   | Query   |    0 | init                  | shc
+----+-----------------+-----------+--------+---------+------+-----------------------+----
3 rows in set, 1 warning (0.00 sec)
```

这个状态其实有很大的误导性。

接下来我们从 `SHOW PROFILE` 的基本用法出发，从源码角度分析它的实现原理。

最后在分析的基础上，看看 case 中的表空间导入操作为什么大部分耗时是在 `System lock` 阶段。

SHOW PROFILE 的用法

陈臣

关注

10 文章 4 粉丝 1K+ 浏览量

获得了 14 次点赞

内容获得 1 次评论

获得了 16 次收藏

热门文章

没想到，JDBC 驱动会偷偷修改 sql_mode 的会话值
2024-03-11 270浏览

SHOW PROCESSLIST 最多能显示多长的 SQL？
2024-06-17 263浏览

基于案例分析 MySQL权限认证中的具体优先原则
2024-10-28 199浏览

如何将 performance_schema 中的 TIMER 字段转换为日期时间
2024-01-04 190浏览

Redis 内存突增时，如何定量分析其内存使用情况
2024-09-23 177浏览

最新文章

基于源码分析 SHOW GLOBAL STATUS 的实现原理
2025-01-06 74浏览

基于案例分析 MySQL权限认证中的具体优先原则
2024-10-28 199浏览

Redis 内存突增时，如何定量分析其内存使用情况
2024-09-23 177浏览

如何让 MGR 不从 Primary 节点克隆数据？
2024-07-22 92浏览

SHOW PROCESSLIST 最多能显示多长的 SQL？
2024-06-17 263浏览

目录

下面通过一个示例来看看 `SHOW PROFILE` 的用法。

```
# 开启 Profiling
mysql> set profiling=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

# 执行需要分析的 SQL
mysql> select count(*) from slowtech.t1;
+----------+
| count(*) |
+----------+
|  1048576 |
+----------+
1 row in set (1.09 sec)

# 通过 show profiles 查看 SQL 对应的 Query_ID
mysql> show profiles;
+----------+------------+---------------------------------+
| Query_ID | Duration   | Query                           |
+----------+------------+---------------------------------+
|        1 | 1.09378600 | select count(*) from slowtech.t1 |
+----------+------------+---------------------------------+
1 row in set, 1 warning (0.00 sec)

# 查看该 SQL 各个阶段的执行耗时情况，其中，1 是该 SQL 对应的 Query_ID
mysql> show profile for query 1;
+------------------------------+----------+
| Status                       | Duration |
+------------------------------+----------+
| starting                     | 0.000157 |
| Executing hook on transaction  | 0.000009 |
| starting                     | 0.000020 |
| checking permissions         | 0.000012 |
| Opening tables               | 0.000076 |
| init                         | 0.000011 |
| System lock                  | 0.000026 |
| optimizing                   | 0.000013 |
| statistics                   | 0.000033 |
| preparing                    | 0.000032 |
| executing                    | 1.093124 |
| end                          | 0.000025 |
| query end                    | 0.000013 |
| waiting for handler commit   | 0.000078 |
| closing tables               | 0.000048 |
| freeing items                | 0.000076 |
| cleaning up                  | 0.000037 |
+------------------------------+----------+
17 rows in set, 1 warning (0.01 sec)
```

如果指定 all 还会输出更详细的统计信息，包括 CPU、上下文切换、磁盘IO、IPC（进程间通信）发送/接受的消息数量、页面故障次数、交换次数等。

需要注意的是，这里的统计信息是针对整个进程的，不是单个 SQL 的。如果在执行上述 SQL 的同时还有其它 SQL 在执行，那么这些数据就不能用来评估该 SQL 的资源使用情况。

```
mysql> show profile all for query 1\G
...
*************************** 11. row ***************************
             Status: executing
           Duration: 0.825417
           CPU_user: 1.486951
         CPU_system: 0.007982
  Context_voluntary: 0
Context_involuntary: 553
       Block_ops_in: 0
      Block_ops_out: 0
      Messages_sent: 0
  Messages_received: 0
  Page_faults_major: 0
  Page_faults_minor: 24
              Swaps: 0
    Source_function: ExecuteIteratorQuery
        Source_file: sql_union.cc
        Source_line: 1678
...
17 rows in set, 1 warning (0.00 sec)
```

SHOW PROFILE 的实现原理

SHOW PROFILE 主要是在 `sql_profile.cc` 中实现的。它的实现主要分为两部分：

1. 数据的采集。
2. 数据的计算。

下面我们分别从这两个维度来看看 SHOW PROFILE 的实现原理。

- 背景
- SHOW PROFILE 的用法
- SHOW PROFILE 的实现原理
  - 数据的采集
  - 数据的计算
- 表空间导入操作为什么大部分耗时是在 ...
- 总结
- 参考资料

数据的采集

数据的采集实际上是通过“埋点”来实现的。不同阶段对应的“埋点”地址可通过 `show profile source` 查看。

```
mysql> show profile source for query 1;
+------------------------------+----------+-------------------------+--------------------
| Status                       | Duration | Source_function         | Source_file
+------------------------------+----------+-------------------------+--------------------
| starting                     | 0.000157 | NULL                    | NULL
| Executing hook on transaction  | 0.000009 | launch_hook_trans_begin | rpl_handler.cc
| starting                     | 0.000020 | launch_hook_trans_begin | rpl_handler.cc
| checking permissions         | 0.000012 | check_access            | sql_authorization.c
| Opening tables               | 0.000076 | open_tables             | sql_base.cc
| init                         | 0.000011 | execute                 | sql_select.cc
| System lock                  | 0.000026 | mysql_lock_tables       | lock.cc
| optimizing                   | 0.000013 | optimize                | sql_optimizer.cc
| statistics                   | 0.000033 | optimize                | sql_optimizer.cc
| preparing                    | 0.000032 | optimize                | sql_optimizer.cc
| executing                    | 1.093124 | ExecuteIteratorQuery    | sql_union.cc
| end                          | 0.000025 | execute                 | sql_select.cc
| query end                    | 0.000013 | mysql_execute_command   | sql_parse.cc
| waiting for handler commit     | 0.000078 | ha_commit_trans         | handler.cc
| closing tables               | 0.000048 | mysql_execute_command   | sql_parse.cc
| freeing items                | 0.000076 | dispatch_sql_command    | sql_parse.cc
| cleaning up                  | 0.000037 | dispatch_command        | sql_parse.cc
+------------------------------+----------+-------------------------+--------------------
17 rows in set, 1 warning (0.00 sec)
```

以 `executing` 为例，它对应的“埋点”地址是 `sql_union.cc` 文件的第 1677 行，该行对应的代码是：

```
  THD_STAGE_INFO(thd, stage_executing);
```

其它的“埋点”地址也类似，调用的都是 `THD_STAGE_INFO` ，唯一不一样的是 stage 的名称。

THD_STAGE_INFO 主要会做两件事情：

1. 采集数据。
2. 将采集到的数据添加到队列中。

下面我们结合代码看看具体的实现细节。

```
void QUERY_PROFILE::new_status(const char *status_arg, const char *function_arg,
                               const char *file_arg, unsigned int line_arg) {
  PROF_MEASUREMENT *prof;
  ...
  // 初始化 PROF_MEASUREMENT，初始化的过程中会采集数据。
  if ((function_arg != nullptr) && (file_arg != nullptr))
    prof = new PROF_MEASUREMENT(this, status_arg, function_arg,
                                base_name(file_arg), line_arg);
  else
    prof = new PROF_MEASUREMENT(this, status_arg);
  // m_seq 是阶段的序号，对应 information_schema.profiling 中的 SEQ。
  prof->m_seq = m_seq_counter++;
  // time_usecs 是采集到的系统当前时间。
  m_end_time_usecs = prof->time_usecs;
  // 将采集到的数据添加到队列中，这个队列在查询时会用到。
  entries.push_back(prof);
  ...
}
```

继续分析 `PROF_MEASUREMENT` 的初始化逻辑。

```
PROF_MEASUREMENT::PROF_MEASUREMENT(QUERY_PROFILE *profile_arg,
                                   const char *status_arg,
                                   const char *function_arg,
                                   const char *file_arg, unsigned int line_arg)
    : profile(profile_arg) {
  collect();
  set_label(status_arg, function_arg, file_arg, line_arg);
}

void PROF_MEASUREMENT::collect() {
  time_usecs = (double)my_getsystime() / 10.0; /* 1 sec was 1e7, now is 1e6 */
#ifdef HAVE_GETRUSAGE
  getrusage(RUSAGE_SELF, &rusage);
#elif defined(_WIN32)
  ...
#endif
}
```

PROF_MEASUREMENT 在初始化时会调用 `collect` 函数， `collect()` 函数非常关键，它会做两件事情：

1. 通过 `my_getsystime()` 获取系统的当前时间。
2. 通过 `getrusage(RUSAGE_SELF, &rusage)` 获取当前进程（注意是进程，不是当前 SQL）的资源使用情况。

   `getrusage` 是一个用于获取进程或线程资源使用情况的系统调用。它返回进程在执行期间所消耗的资源信息，包括 CPU 时间、内存使用、页面故障、上下文切换等信息。

PROF_MEASUREMENT 初始化完毕后，会将其添加到 entries 中。entries 是一个队列（`Queue<PROF_MEASUREMENT> entries`）。这个队列，会在执行 `show profile for query N` 或者 `information_schema.profiling` 时用到。

说完数据的采集，接下来我们看看数据的计算，毕竟“埋点”收集的只是系统当前时间，而我们在 `show profile for query N` 中看到的Duration 是一个时长。

数据的计算

当我们在执行 `show profile for query N` 时，实际上查询的是 `information_schema.profiling`，此时，会调用 `PROFILING::fill_statistics_info` 来填充数据。

下面我们看看该函数的实现逻辑。

```
int PROFILING::fill_statistics_info(THD *thd_arg, Table_ref *tables) {
  DBUG_TRACE;
  TABLE *table = tables->table;
  ulonglong row_number = 0;

  QUERY_PROFILE *query;
  // 循环 history 队列，队列中的元素是 QUERY_PROFILE，每一个查询对应一个 QUERY_PROFILE。
  // 队列的大小由参数 profiling_history_size 决定，默认是 15。
  void *history_iterator;
  for (history_iterator = history.new_iterator(); history_iterator != nullptr;
       history_iterator = history.iterator_next(history_iterator)) {
    query = history.iterator_value(history_iterator);

    ulong seq;

    void *entry_iterator;
    PROF_MEASUREMENT *entry, *previous = nullptr;
    // 循环每个查询中的 entries，entries 存储了每个阶段的系统当前时间。
    for (entry_iterator = query->entries.new_iterator();
         entry_iterator != nullptr;
         entry_iterator = query->entries.iterator_next(entry_iterator),
    previous = entry, row_number++) {
entry = query->entries.iterator_value(entry_iterator);
seq = entry->m_seq;

if (previous == nullptr) continue;

if (thd_arg->lex->sql_command == SQLCOM_SHOW_PROFILE) {
  if (thd_arg->lex->show_profile_query_id ==
      0) /* 0 == show final query */
  {
    if (query != last) continue;
  } else {
    // 如果记录中的 Query_ID 跟 show profile for query query_id 中的不一致，则继续判断下一条
    if (thd_arg->lex->show_profile_query_id != query->profiling_query_id)
      continue;
  }
}

restore_record(table, s->default_values);
// query->profiling_query_id 用来填充 information_schema.profiling 中的 QUERY_ID
table->field[0]->store((ulonglong)query->profiling_query_id, true);
// seq 用来填充 information_schema.profiling 中的 SEQ
table->field[1]->store((ulonglong)seq,
                       true);
      // status 用来填充 information_schema.profiling 中的 STATE
      // 注意，这里是上一条记录的 status，不是当前记录的 status
      table->field[2]->store(previous->status, strlen(previous->status),
                             system_charset_info);
      // 当前记录的 time_usecs 减去上一条记录的 time_usecs 的值，换算成秒，用来填充 information_sch
      my_decimal duration_decimal;
      double2my_decimal(
          E_DEC_FATAL_ERROR,
          (entry->time_usecs - previous->time_usecs) / (1000.0 * 1000),
          &duration_decimal);

      table->field[3]->store_decimal(&duration_decimal);
#ifdef HAVE_GETRUSAGE
      my_decimal cpu_utime_decimal, cpu_stime_decimal;
      // 当前记录的 ru_utime 减去上一条记录的 ru_utime，用来填充 information_schema.profiling 中
      double2my_decimal(
          E_DEC_FATAL_ERROR,
          RUSAGE_DIFF_USEC(entry->rusage.ru_utime, previous->rusage.ru_utime) /
              (1000.0 * 1000),
          &cpu_utime_decimal);
```

```
      ...
      table->field[4]->store_decimal(&cpu_utime_decimal);
...

  return 0;
}
```

可以看到，`information_schema.profiling` 中的第三列（STATE，对应 `show profile for query N` 中的 Status）存储的是上一条记录的 status（阶段名），而第四列（DURATION）的值等于当前记录的采集时间（entry->time_usecs）减去上一条记录的采集时间（previous->time_usecs）。

所以，我们在 `show profile for query N` 中看到的 Duration 实际上通过下一个阶段的采集时间减去当前阶段的采集时间得到的，并不是 `show profile source` 中函数（Source_function）的执行时长。

这种实现方式在判断操作当前状态和分析各个阶段耗时时存在一定的误导性。

回到开头的 case。

表空间导入操作为什么大部分耗时是在 System lock 阶段？

表空间导入操作是在 `mysql_discard_or_import_tablespace` 函数中实现的。

下面是该函数简化后的代码。

```
bool Sql_cmd_discard_import_tablespace::mysql_discard_or_import_tablespace(
    THD *thd, Table_ref *table_list) {
  ...
  THD_STAGE_INFO(thd, stage_discard_or_import_tablespace);
  ...
  if (open_and_lock_tables(thd, table_list, 0, &alter_prelocking_strategy)) {
    return true;
  }
  ...
  const bool discard =
      (m_alter_info->flags & Alter_info::ALTER_DISCARD_TABLESPACE);
  error = table_list->table->file->ha_discard_or_import_tablespace(discard,
                                                                   table_def);
  THD_STAGE_INFO(thd, stage_end);
  ...
  return true;
}
```

可以看到，该函数实际调用的是 THD_STAGE_INFO(thd, stage_discard_or_import_tablespace)。

只不过，在调用 THD_STAGE_INFO(thd, stage_discard_or_import_tablespace) 后，调用了 open_and_lock_tables。

而 open_and_lock_tables 最后会调用 THD_STAGE_INFO(thd, stage_system_lock)。

这也就是为什么上述函数中虽然调用了 THD_STAGE_INFO(thd, stage_discard_or_import_tablespace)，但 `show profile` 和 `show processlist` 的输出中却显示 `System lock`。

但基于对耗时的分析，我们发现这么显示其实并不合理。

在开头的 case 中，虽然 `System lock` 阶段显示的耗时是 788.966338 秒，但实际上 `open_and_lock_tables` 这个函数只消耗了 0.000179 秒，真正的耗时是来自 `table_list->table->file->ha_discard_or_import_tablespace`，其执行时间长达 788.965481 秒。

为什么这个函数需要执行这么久呢？主要是表空间在导入的过程中会检查并更新表空间中的每个页，包括验证页是否损坏、更新表空间 ID 和 LSN、处理 Btree 页（如设置索引 ID、清除 delete marked 记录等）、将页标记为脏页等。表越大，检查校验的时候会越久。

如此来看，针对表空间导入操作，将其状态显示为 `discard_or_import_tablespace` 更能反映操作的真实情况。

总结

1. 在 `SHOW PROFILE` 中显示的每个阶段的耗时，实际上是由下一个阶段的采集时间减去当前阶段的采集时间得出的。

   每个阶段的采集时间是通过在代码的不同路径中植入 `THD_STAGE_INFO(thd, stage_xxx)` 实现的，采集的是系统当前时间。

2. 这种实现方式在判断操作当前状态（通过 SHOW PROCESSLIST）和分析各个阶段耗时（通过 SHOW PROFILE ）时存在一定的误导性，主要是因为预定义的阶段数量是有限的。

   在 MySQL 8.4 中，共定义了 98 个阶段，具体的阶段名可在 `mysqld.cc` 中的 `all_server_stages` 数组找到。

3. 在表空间导入操作中，虽然大部分耗时显示为 `System lock` 阶段，但实际上，使用 `discard_or_import_tablespace` 来描述这一过程会更为准确。

参考资料

1. https://dev.mysql.com/doc/refman/8.4/en/show-profile.html
2. https://dev.mysql.com/doc/refman/8.4/en/performance-schema-query-profiling.html
3. https://dev.mysql.com/worklog/task/?id=5522

墨力计划 mysql

「喜欢这篇文章，您的关注和赞赏是给作者最好的鼓励」

关注作者 赞赏

评论

分享你的看法，一起交流吧～

相关阅读

【干货】2024年下半年墨天轮最受欢迎的50篇技术文章+文档
墨天轮编辑部 1562次阅读 2025-02-13 10:42:44

MySQL性能分析的“秘密武器”，深度剖析SQL问题
szrsu 680次阅读 2025-01-23 09:59:26

2025年1月“墨力原创作者计划”获奖名单公布
墨天轮编辑部 348次阅读 2025-02-13 15:07:02

MySQL 主从节点切换指导
CuiHulong 302次阅读 2025-01-23 11:50:29

[MYSQL] 忘记root密码时, 不需要重启也能强制修改了!
大大刺猬 286次阅读 2025-02-06 11:12:15

mysql 内存使用率高问题排查
蔡璐 266次阅读 2025-02-06 10:02:23

MySQL 9.2.0 中的更新（2025-01-21，创新版本）
通讯员 151次阅读 2025-01-22 09:54:21

MySQL基础高频面试题－划重点、敲难点
锁钥 146次阅读 2025-02-03 07:52:28

MySQL 底层数据&日志刷新策略解读
CuiHulong 133次阅读 2025-02-11 10:56:13

[MYSQL] mysql主从延迟案例(有索引但无主键)
大大刺猬 107次阅读 2025-01-21 13:53:21