4.5 MySQL 优化案例 select count(*)

作者：xuty

一、故事背景

项目组联系我说是有一张 500w 左右的表做 select count(*) 速度特别慢。

二、原 SQL 分析

Server version: 5.7.24-log MySQL Community Server (GPL)

SQL 如下，仅仅就是统计 api_runtime_log 这张表的行数，一条简单的不能再简单的 SQL：

```
select count(*) from api_runtime_log;
```

我们先去运行一下这条 SQL，可以看到确实运行很慢，要 40 多秒左右，确实很不正常~

```
mysql> select count(*) from api_runtime_log;
+----------+
| count(*) |
+----------+
|  5718952 |
+----------+
1 row in set (42.95 sec)
```

我们再去看下表结构，看上去貌似也挺正常的~存在主键，表引擎也是 InnoDB，字符集也没问题。

```
CREATE TABLE `api_runtime_log` (
  `BelongXiaQuCode` varchar(50) DEFAULT NULL,
  `OperateUserName` varchar(50) DEFAULT NULL,
  `OperateDate` datetime DEFAULT NULL,
  `Row_ID` int(11) DEFAULT NULL,
  `YearFlag` varchar(4) DEFAULT NULL,
  `RowGuid` varchar(50) NOT NULL,
   ......
  `apiid` varchar(50) DEFAULT NULL,
  `apiname` varchar(50) DEFAULT NULL,
  `apiguid` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`RowGuid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

三、执行计划

通过执行计划，我们看下是否可以找到什么问题点。

```
mysql> explain select count(*) from api_runtime_log \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: api_runtime_log
   partitions: NULL
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 152
          ref: NULL
         rows: 5718952
     filtered: 100.00
        Extra: Using index
```

可以看到，查询走的是 PRIMARY，也就是主键索引。貌似也没有什么问题，走索引了呀！那么是不是真的就没问题呢？

四、原理

为了找到答案，通过 Google 查找 MySQL 下 select count(*) 的原理，找到了答案。这边省略过程，直接上结果。

简单介绍下原理：

- 聚簇索引：每一个 InnoDB 存储引擎下的表都有一个特殊的索引用来保存每一行的数据，称为聚簇索引（通常都为主键），聚簇索引实际保存了 B-Tree 索引和行数据，所以大小实际上约等于为表数据量
- 二级索引：除了聚集索引，表上其他的索引都是二级索引，索引中仅仅存储了对应索引列及主键列

在 InnoDB 存储引擎中，count(*) 函数是先从内存中读取数据到内存缓冲区，然后进行扫描获得行记录数。这里 InnoDB 会优先走二级索引；如果同时存在多个二级索引，会选择 key_len 最小的二级索引；如果不存在二级索引，那么会走主键索引；如果连主键都不存在，那么就走全表扫描！

这里我们由于走的是主键索引，所以 MySQL 需要先把整个主键索引读取到内存缓冲区，这是个从磁盘读写到内存的过程，而且主键索引基本等于整个表数据量（10GB+），所以非常耗时！

那么如何解决呢？

答案就是：建二级索引。

因为二级索引只包含对应的索引列及主键列，所以体积非常小。在 select count(*) 的查询过程中，只需要将二级索引读取到内存缓冲区，只有几十 MB 的数据量，所以速度会非常快。

举个形象的比喻，我们想知道一本书的页数：

- 走聚集索引：从第一页翻到最后一页，知道总页数；
- 走二级索引：通过目录直接知道总页数。

五、验证

创建二级索引后，再次执行 SQL 及查看执行计划。

```
mysql> create index idx_rowguid on api_runtime_log(rowguid);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select count(*) from api_runtime_log;
+----------+
| count(*) |
+----------+
|  5718952 |
+----------+
1 row in set (0.89 sec)

mysql> explain select count(*) from api_runtime_log \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: api_runtime_log
   partitions: NULL
         type: index
possible_keys: NULL
          key: idx_rowguid
      key_len: 152
          ref: NULL
         rows: 5718952
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

可以看到添加二级索引后，确实速度明显变快，而且执行计划也变成了走二级索引。至此这个问题其实已经解决了，就是由于表上缺少二级索引导致。

六、深入测试

为了进一步验证上述的推论，所以就做了如下的测试。

测试过程如下：

1. 通过 sysbench 创建了一张 500W 的测试表 sbtest1，表上仅仅包含一个主键索引，表大小为 1125MB；
2. 调整部分 MySQL 参数，重启 MySQL，保证目前 innodb buffer pool (内存缓冲区) 中为空，不缓存任何数据；
3. 执行 select count(*)，理论上走主键索引，查看当前内存缓冲区中缓存的数据量（理论上会缓存整个聚簇索引）；
4. 在测试表 sbtest1 上添加二级索引，索引大小为 55MB；
5. 再次重启 MySQL，保证内存缓冲区为空；
6. 再次执行 select count(*)，理论上走二级索引；
7. 再次查看内存缓冲区中缓存的数据量（理论上只会缓存二级索引）。

测试结果如下：

1. 聚簇索引

查询当前内存缓冲区状态，结果为空证明不缓存测试表数据。

```
mysql> select * from sys.innodb_buffer_stats_by_table where object_schema = 'test';
Empty set (1.92 sec)

mysql>  select count(*) from test.sbtest1;
+----------+
| count(*) |
+----------+
|  5188434 |
+----------+
1 row in set (5.52 sec)
```

再次查看内存缓冲区，发现缓存了 sbtest1 表上 1G 多的数据，基本等于整个表数据量。

```
mysql> select * from sys.innodb_buffer_stats_by_table where object_schema = 'test'
    \G;
*************************** 1. row ***************************
object_schema: test
  object_name: sbtest1
    allocated: 1.08 GiB
         data: 1.01 GiB
        pages: 71081
 pages_hashed: 0
    pages_old: 28119
  rows_cached: 5189798
```

最后我们再来看下执行计划，确实走的是主键索引，放在最后执行是为了避免影响缓冲区。

```
mysql> explain  select count(*) from test.sbtest1 \G;
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sbtest1
   partitions: NULL
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 5117616
     filtered: 100.00
        Extra: Using index
```

2. 二级索引

创建二级索引 idx_id，查看 sbtest1 表上主键索引与二级索引的数据量。

```
mysql> create index idx_id on sbtest1(id);
Query OK, 0 rows affected (12.97 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT sum(stat_value) pages ,index_name ,
(round((sum(stat_value) * @@innodb_page_size)/1024/1024)) as MB
  FROM mysql.innodb_index_stats
  WHERE table_name = 'sbtest1'
  AND database_name = 'test'
  AND stat_description = 'Number of pages in the index'
  GROUP BY index_name;
+-------+------------+------+
| pages | index_name | MB   |
+-------+------------+------+
| 72000 | PRIMARY    | 1125 |
|  3492 | idx_id     |   55 |
+-------+------------+------+
```

重启 MySQL，再次查看缓冲区同样为空，证明没有缓存测试表上的数据。

```
mysql> select * from sys.innodb_buffer_stats_by_table where object_schema = 'test';
Empty set (1.49 sec)

mysql> select count(*) from test.sbtest1;
+----------+
| count(*) |
+----------+
```

```
|  5188434 |
+----------+
1 row in set (2.92 sec)
```

再次查看内存缓冲区，发现仅仅缓存了 sbtest1 表上的 50M 数据，约等于二级索引的数据量。

```
mysql> select * from sys.innodb_buffer_stats_by_table where object_schema = 'test'
    \G;
*************************** 1. row ***************************
object_schema: test
  object_name: sbtest1
    allocated: 49.48 MiB
         data: 46.41 MiB
        pages: 3167
 pages_hashed: 0
    pages_old: 1575
rows_cached: 2599872
```

最后确认下执行计划，确实走的是二级索引。

```
mysql> explain select count(*) from test.sbtest1 \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sbtest1
   partitions: NULL
         type: index
possible_keys: NULL
          key: idx_id
      key_len: 4
          ref: NULL
         rows: 5117616
     filtered: 100.00
        Extra: Using index
```

七、案例总结

从上述这个测试结果可以看出，和之前的推论基本吻合。

如果 select count(*) 走的是主键索引，那么会缓存整个表数据，大量查询时间会花费在读取表数据到缓冲区。

如果存在二级索引，那么只需要读取索引页到缓冲区即可，速度自然快。

另：项目上由于磁盘性能层次不齐，所以当遇上这种情况时，性能较差的磁盘更会放大这个问题；一张超级大表，统计行数时如果走了主键索引，后果可想而知~

八、优化建议

此次测试过程中我们仅仅模拟是百万数据量，此时我们通过二级索引统计表行数，只需要读取几十 M 的数据量，就可以得到结果。

那么当我们的表数据量是上千万，甚至上亿时呢。此时即便是最小的二级索引也是 几百 M、过 G 的数据量，如果继续通过二级索引来统计行数，那么速度就不会如此迅速了。

这个时候可以通过避免直接 select count(*) from table 来解决，方法较多，例如：

1. 使用 MySQL 触发器 + 统计表实时计算表数据量；
2. 使用 MyISAM 替换 InnoDB，因为 MyISAM 自带计数器，坏处就不多说了；
3. 通过 ETL 导入表数据到其他更高效的异构环境中进行计算；
4. 升级到 MySQL 8 中，使用并行查询，加快检索速度。

当然，什么时候 InnoDB 存储引擎可以直接实现计数器的功能就好了！

4.6 MySQL OOM 故障应如何下手

作者：孙祚龙

引言

前阵子处理这样一个案例，某客户的实例 mysqld 进程内存经常持续增加导致最终被 OOM killer。作为 DBA 肯定想知道有哪些原因可能会导致 OOM（内存溢出）。

先介绍下这位朋友：OOM-killer

OOM Killer(Out of Memory Killer) 当系统内存严重不足时 linux 内核采用的杀掉进程，释放内存的机制。

OOM Killer 通过检查所有正在运行的进程，然后根据自己的算法给每个进程一个 badness 分数，拥有最高 badness 分数的进程将会在内存不足时被杀掉。

它打分的算法如下:

- 某一个进程和它所有的子进程都占用了很多内存的将会打一个高分。
- 为了释放足够的内存来解决这种情况，将杀死最少数量的进程（最好是一个进程）。
- 内核进程和其他较重要的进程会被打成相对较低的分。

上面打分的标准意味着，当 OOM killer 选择杀死的进程时，将选择一个使用大量内存，有很多子进程且不是系统进程的进程。

简单来讲，oom-killer 的原则就是损失最小、收益最大，因此它会让杀死的进程数尽可能小、释放的内存尽可能大。在数据库服务器上，MySQL 被分配的内存一般不会小，因此容易成为 oom-killer 选择的对象。

“ 既然发生了 OOM，那必然是内存不足，内存不足这个问题产生原因很多。

首先第一个就是 MySQL 自身内存的规划有问题，这就涉及到 mysql 相应的配置参数。

另一个可以想到的原因就是一般部署 MySQL 的服务器，都会部署很多的监控和定时任务脚本，而这些脚本往往缺少必要的内存限制，导致在高峰期的时候占用大量的内存，导致触发 Linux 的 oom-killer 机制，最终 MySQL 无辜躺枪牺牲。 ”

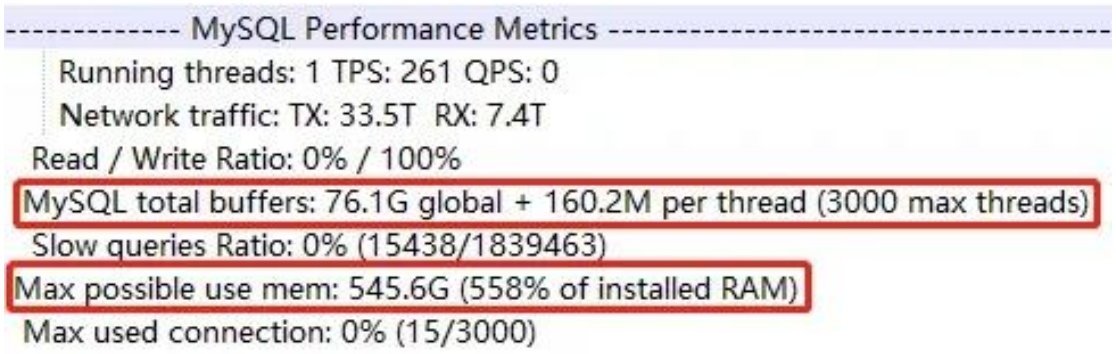
all-important：MySQL 自身内存规划

说到 MySQL 自身的内存规划，最先想到的就是 MySQL 中各种 buffer 的大小，innodb buffer pool 就是最鹤立鸡群的那个。innodb_buffer_pool_size 参数的大小究竟如何设置，才能保证 MySQL 的性能呢？在官网文档中可以找到这个参数的一些描述：

A larger buffer pool requires less disk I/O to access the same table data more than once. On a dedicated database server, you might set the buffer pool size to 80% of the machine's physical memory size.

意思是在专用数据库服务器上，可以将 innodb_buffer_pool_size 设置为计算机物理内存大小的 80%。在许许多多前辈的的经验中了解到，此参数的值设置为物理内存的 50%~80% 颇为合理。

举个栗子：



innodb buffer pool 分配 76G，每个连接线程最大可用 160M，最大有 3000 连接数，最大可能使用内存总量 545G，但是这台实例所在服务器的物理内存仅仅有 97G，远超物理内存总量。结果可想而知，这个实例在运行中经常被 oom-killer 杀死，想必原因之一即是因为一开始 MySQL 自身的内存规划欠妥。

innodb buffer pool 缓存数据的作用相信大家都懂，比如这个 case 中，可以发现该实例为写密集，读请求很少，innodb buffer 对性能改善作用不大，80% 的内存没必要，完全可以降低到物理内存的 50%。

“ 以上是对 OOM 发生原因的一些见解，那思考一下还有没有其他的原因会导致内存溢出的情况呢？不知道大家对内存泄漏是否了解，有没有可能 MySQL 因为内存泄漏堆积演变为内存溢出，最终 oom-killer ... ”

知识补给站：内存泄漏

内存泄漏（Memory Leak）是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

内存泄漏缺陷具有隐蔽性、积累性的特征，比其他内存非法访问错误更难检测。因为内存泄漏的产生原因是内存块未被释放，属于遗漏型缺陷而不是过错型缺陷。此外，内存泄漏通常不会直接产生可观察的错误症状，而是逐渐积累，降低系统整体性能，极端的情况下可能使系统崩溃。

关于 valgrind 工具

Valgrind 是一个用于构建动态分析工具的工具框架。它提供了一组工具，每个工具都执行某种调试、分析或类似的任务，以帮助您改进程序。Valgrind 的体系结构是模块化的，因此可以轻松地创建新工具，而不会影响现有的结构。

标配了许多有用的工具：

1. Memcheck 是内存错误检测器。
2. Cachegrind 是一个缓存和分支预测探查器。
3. Callgrind 是一个生成调用图的缓存分析器。
4. Helgrind 是线程错误检测器。
5. DRD 还是线程错误检测器。
6. Massif 是堆分析器。
7. DHAT 是另一种堆分析器。
8. SGcheck 是一种实验性工具，可以检测堆栈和全局阵列的溢出。
9. BBV 是一个实验性 SimPoint 基本块矢量生成器。

关于内存泄漏，我们需要使用 valgrind 的默认工具，也就是 memcheck 工具。

Memcheck 是内存错误检测器。它可以检测以下和内存相关的问题：

- 使用未初始化的内存
- 读取/写入已释放的内存
- 读取/写入 malloc 块的末端
- 内存泄漏
- 对 malloc/new/new[]与 free/delete/delete[] 的不匹配使用
- 双重释放内存

Valgrind Memcheck 工具的用法如下：

```
valgrind --tool=memcheck ./a.out
```

从上面的命令可以清楚地看到，主要的命令是“ Valgrind”，而我们要使用的工具由选项“ --tool”指定。上面的“ a.out ” 表示我们要在其上运行 memcheck 的可执行文件。此外还可以使用其他的命令行选项，以满足我们的需要。运行的程序结束后，会生成这个进程的内存分析报告。

“ OK，工具有了，这就如同摸金校尉拿到了洛阳铲，宝藏还会远吗~ 还不快找几块地挖掘试试？”

测试

1. 使用 valgrind 的 memcheck 工具启动 mysql：

```
valgrind --tool=memcheck --leak-check=full --show-reachable=yes                    -
    -log-file=/tmp/valgrind-mysql.log /usr/local/mysql/bin/mysqld                  --
    defaults-file=/etc/my.cnf --user=root
```

2. 利用 sysbench 模拟负载；

3. 进程结束后查看检测报告：

```
==29326== LEAK SUMMARY:
==29326==    definitely lost: 0 bytes in 0 blocks
==29326==    indirectly lost: 0 bytes in 0 blocks
==29326==      possibly lost: 549,072 bytes in 1,727 blocks
==29326==    still reachable: 446,492,944 bytes in 54 blocks
==29326==         suppressed: 0 bytes in 0 blocks
==29326==
==29326== For counts of detected and suppressed errors, rerun with: -v
==29326== ERROR SUMMARY: 339 errors from 339 contexts (suppressed: 0 from 0)
```

在报告的最后的总结中发现程序退出时有部分内存未释放，而且存在潜在的内存泄漏。通过向上查看具体的信息，分析后发现主要集中在 performance_schema，偶然发现了一个疑点，那我们完全禁用掉 performance_schema 呢？

```
==9954== LEAK SUMMARY:
==9954==    definitely lost: 0 bytes in 0 blocks
==9954==    indirectly lost: 0 bytes in 0 blocks
==9954==    possibly lost: 0 bytes in 0 blocks
==9954==    still reachable: 32 bytes in 1 blocks
==9954==         suppressed: 0 bytes in 0 blocks
==9954==
==9954== For counts of detected and suppressed errors, rerun with: -v
==9954== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

发现程序退出时几乎没有内存未释放，也不存在潜在的内存泄漏。三次测试过后，发现结果是一致的。这是什么原因？

“ 大家都知道 MySQL 的 performance schema 用于监控 MySQL server 在一个较低级别的运行过程中的资源消耗、资源等待等情况，但它为什么可能会导致内存泄漏呢，看来关于 ps 还有不少待挖掘的宝藏哦~ ”

总结

1. 注意 MySQL 自身的内存规划，为保证 MySQL 的性能，innodb buffer pool 大小设置要合理，可以根据实例读写负载的情况适当调整 buffer pool 的大小。并且 innodb buffer 与连接会话内存的总和尽量不要超过系统物理内存。
2. 调整 oom_score_adj 参数（/proc/<pid>/oom_score_adj），将 MySQL 被 oom-killer 锁定的优先级降低。这个参数值越小，越不容易被锁定。
3. 加强内存的监控和报警，一旦报警，DBA 应该迅速介入，选择性 Kill 掉一些占用较多内存的连接。
4. 在开启 performance_schema 时，会有额外的内存开销，通过 valgrind-memcheck 内存分析工具发现，较大概率发生内存泄漏。它有可能也会导致 OOM，在场景中若不需要 performance_schema 可以完全禁用，或需要尽量只开启必要的 instrument。

4.7 MySQL 字符集转换

作者：xuty

一、背景

开发联系我，说开发库上有一张视图查询速度很慢，9000 条数据要查 10s，要求我这边协助排查优化。

二、问题 SQL

Server version: 5.7.24-log MySQL Community Server (GPL)

这个 SQL 非常简单，定义如下，其中就引用了 view_dataquality_analysis 这张视图，后面跟了两个 where 条件，并且做了分页。

```
SELECT *
FROM view_dataquality_analysis
WHERE modelguid = '710adae5-1900-4207-9864-d53ee3a81923'
  AND configurationguid = '6845d000-cda4-43ea-9fd3-9f9f1f22f95d' limit 20;
```

我们先去开发库上运行一下这条 SQL，下图中可以看到确实运行很慢，要 8s 左右。