From: http://mysql.taobao.org/monthly/2016/02/03/

# 背景

在写压力负载比较重的 MySQL 实例上，InnoDB 可能积累了较长的没有被 purge 掉的 transaction history，导致实例性能的衰减，或者空闲空间被耗尽，下面就来看看它是怎么产生的，或者有没有什么方法来减轻，避免这样的问题出现。

# InnoDB purge 概要

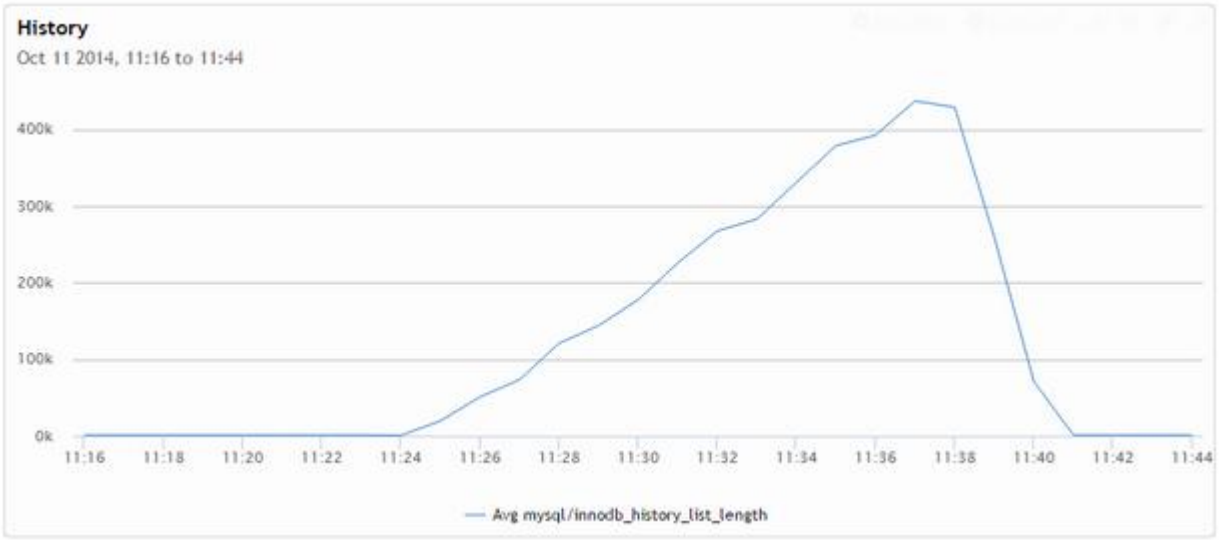
InnoDB 是一个事务引擎，实现了 MVCC 特性，也就是在存储引擎里对行数据保存了多个版本。在对行数据进行 delete 或者 update 更改时，行数据的前映像会保留一段时间，直到可以被删除的时候。

在大部分 OLTP 负载情况下，前映像会在数据操作完成后的数秒钟内被删除掉，但在一些情况下，假设存在一些持续很长时间的事务需要看到数据的前映像，那么老版本的数据就会被保留相当长一段时间。

虽然 MySQL 5.6 版本增加了多个 purge threads 来加快完成老版本数据的清理工作，但在 write-intensive workload 情况下，不一定完全凑效。

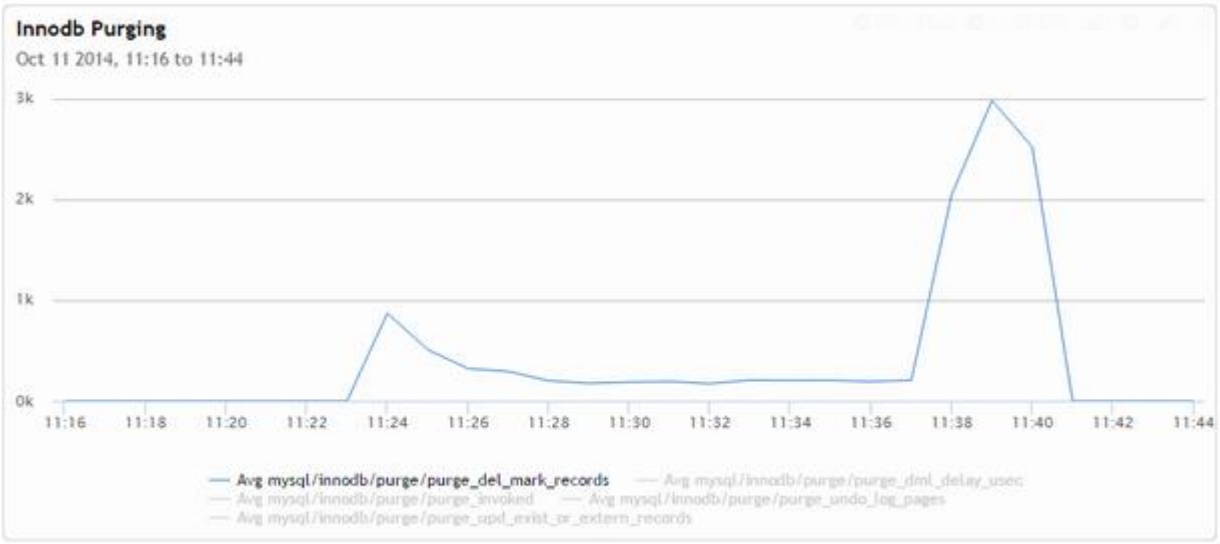
# 测试案例

Peter Zaitsev 使用 sysbench 的 update 进行的测试，无论是 innodb_purge_threads=1 还是 8 的时候，显示的 transaction history 快速增长的情况，如下图所示:



transaction history 增长情况

下面看一下同步测试过程中 purge 的速度(可以通过 `I_S.innodb_metrics` 进行查询)：




InnoDB purge 情况

显示在并发 process 的过程中，purge thread 其实处在饥饿状态，待 sysbench 结束，purge 线程满载运行清理工作。

对于这个测试结果，这里需要说明下：

1. 对于 Peter Zaitsev 的测试，其实主要是为了说明 transaction history 的情况，如果是用 sysbench 进行小事务的 OLTP 测试，并不会产生这么明显的 transaction history 增长而 purge thread 跟不上的情况，或者他在测试的时候，对 sbtest 表进行了全表查询吧，或者设置了 RR 级别，不过这只是猜测。
2. 对于 undo page 大部分被 cache 在 buffer pool 的情况下，purge thread 还是比较快的，但如果因为 buffer pool 的不足而导致 undo page 被淘汰到 disk 上的情况，purge 操作就会被受限 IO 情况， 而导致跟不上。

# 问题分析

我们来看下出现 transaction history 增长最常见的两种场景：

**大查询** 如果你在一张大表上发起一个长时间运行的查询，比如 mysqldump，那么 purge 线程必须停下来等待查询结束，这个时候 transaction undo 就会累积。如果 buffer pool 中 free page 紧张，undo page 还会被置换到 disk 上，加剧 purge 的代价。

**MySQL 重启** 即使 transaction history 并没有急剧增加，但 MySQL 重启操作，buffer pool 的重新预热，还是导致 purge 变成 IO 密集型操作。不过 MySQL 5.6 提供了 InnoDB buffer pool 的 dump 和 reload 方法，可以显著减轻 purge 的 IO 压力。

这里介绍一下如何查看 buffer pool 中 undo page 的 cache 情况，percona 的版本上提供了 `I_S.innodb_rseg` 记录 undo 的分配和使用情况：

```
mysql> select sum(curr_size)*16/1024 undo_space_MB from
innodb_rseg;
+---------------+
| undo_space_MB |
+---------------+
|     1688.4531 |
+---------------+
1 row in set (0.00 sec)
mysql> select count(*) cnt, count(*)*16/1024 size_MB,
page_type from innodb_buffer_page group by page_type;
+--------+-----------+-------------------+
| cnt    | size_MB   | page_type         |
+--------+-----------+-------------------+
|     55 |    0.8594 | EXTENT_DESCRIPTOR |
|      2 |    0.0313 | FILE_SPACE_HEADER |
|    108 |    1.6875 | IBUF_BITMAP       |
|  17186 |  268.5313 | IBUF_INDEX        |
| 352671 | 5510.4844 | INDEX             |
|     69 |    1.0781 | INODE             |
|    128 |    2.0000 | SYSTEM            |
|      1 |    0.0156 | TRX_SYSTEM        |
|   6029 |   94.2031 | UNDO_LOG          |
|  16959 |  264.9844 | UNKNOWN           |
+--------+-----------+-------------------+
10 rows in set (1.65 sec)
```

从这两个 information_schema 下的两张表可以看到：undo space 使用的总大小是 1.7G，而 buffer pool 中 cached 不足 100M。

# InnoDB 优化方法

在一定的写压力情况下，并发进行一些大查询，transaction history 就会因为 undo log 无法 purge 而一直增加。

InnoDB 提供了两个参数 `innodb_max_purge_lag`，`innodb_max_purge_lag_delay` 来调整，即当 `trx_sys->rseg_history_len` 超过了设置的 `innodb_max_purge_lag`，

就影响 DML 操作最大 delay 不超过 `innodb_max_purge_lag_delay` 设置的时间，以 microseconds 来计算。
其核心计算代码如下：

```
/*********************************************************
**********//**
Calculate the DML delay required.
@return delay in microseconds or ULINT_MAX */
static
ulint
trx_purge_dml_delay(void)
/*=====================*/
{
    /* Determine how much data manipulation language
(DML) statements
    need to be delayed in order to reduce the lagging of
the purge
    thread. */
    ulint    delay = 0; /* in microseconds; default: no
delay */

    /* If purge lag is set (ie. > 0) then calculate the
new DML delay.
    Note: we do a dirty read of the trx_sys_t data
structure here,
    without holding trx_sys->mutex. */

    if (srv_max_purge_lag > 0) {
        float    ratio;

        ratio = float(trx_sys->rseg_history_len) /
srv_max_purge_lag;

        if (ratio > 1.0) {
            /* If the history list length exceeds the
            srv_max_purge_lag, the data manipulation
            statements are delayed by at least 5000
            microseconds. */
            delay = (ulint) ((ratio - .5) * 10000);
        }

        if (delay > srv_max_purge_lag_delay) {
            delay = srv_max_purge_lag_delay;
        }
```

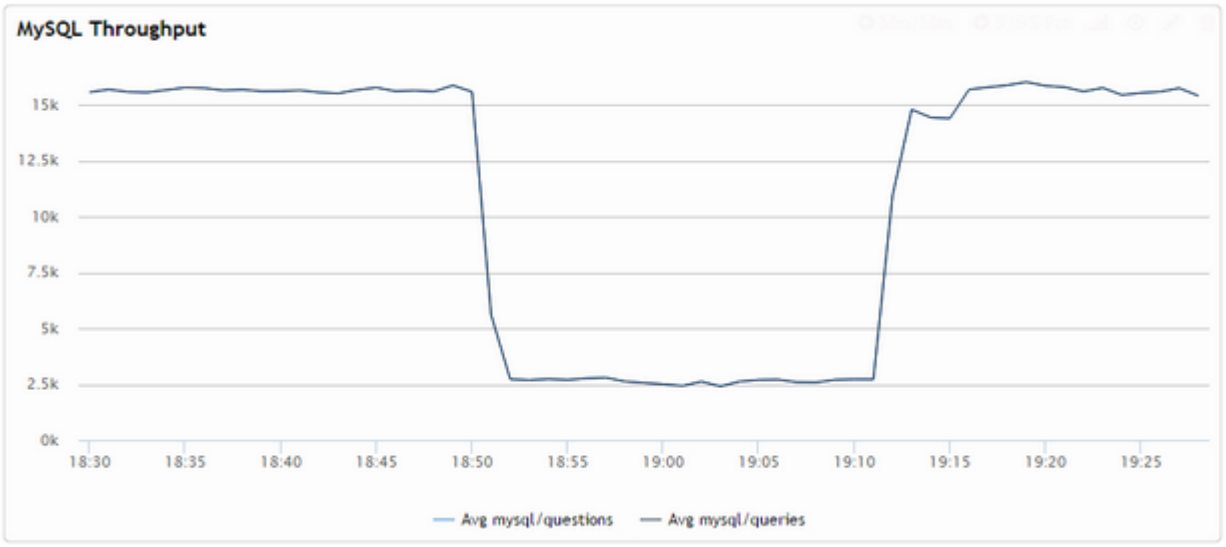
```
            MONITOR_SET(MONITOR_DML_PURGE_DELAY, delay);
        }

        return(delay);
}
```

但这两个参数设计有明显的两个缺陷：

**缺陷 1：针对 total history length** 假设 transaction history 中保留两类 records，一类是是马上可以被 purge 的，一类是因为 active transaction 而不能 purge 的。但大多数时间，我们期望的是 purgable history 比较小，而不是整个 history。

**缺陷 2：针对大小而非变化** `trx_sys->rseg_history_len` 是一个当前 history 的长度，而不是一个 interval 时间段内 undo 的增长和减少的变化情况，导致 `trx_sys->rseg_history_len` 一旦超过 `innodb_max_purge_lag` 这个设定的值，就对 DML 产生不超过 `innodb_max_purge_lag_delay` 的时间 delay，一旦低于这个值马上 delay 时间就又恢复成 0。
在对系统的吞吐监控的时候，会发现系统抖动非常厉害，而不是一个平滑的曲线。类似于下图：




Purge 造成系统抖动

# InnoDB purge 设计思路

针对 InnoDB 的 purge 功能，可以从以下几个因素来综合考虑：

1. 增加默认 purge thread 的个数；
2. 测量 purgable history 长度而不是总的长度；

3. 针对变化进行调整 delay 数值，以应对 shrinking；
4. 基于 undo space 的大小，而不是事务的个数；
5. 调整 undo page 在 buffer pool 中的缓存策略，类似 insert buffer；
6. 针对 undo page 使用和 index page 不同的预读策略。

以上 6 条可以针对 purge 线程进行一些改良。

# 当前调优方法

在当前的 MySQL 5.6 版本上，我们能做哪些调整或者调优方法，以减少 transaction history 增加带来的问题呢？

**监控** 监控 `trx_sys` 的 `innodb_history_list_length`，为它设置报警值，及时关注和处理。
**调整参数** 如果你的实例是写压力比较大的话，调整 `innodb_purge_threads=8`，增加并发 purge 线程数。 谨慎调整 `innodb_max_purge_lag` 和 `innodb_max_purge_lag_delay` 参数，依据现在的设计，可能你的实例的吞吐量会急剧的下降。
**purge 完之后再 shutdown** 大部分的 case 下，MySQL 实例重启后，会发现 purge 的性能更差，因为 undo page 未命中的原因，并且是 random IO 请求。 如果是正常 shutdown，就等 purge 完成再 shutdown；如果是 crash，就启动后等 purge 完成再接受业务请求。

**预热** 使用 MySQL 5.6 提供的 `innodb_buffer_pool_dump_at_shutdown=on` 和 `innodb_buffer_pool_load_at_startup=on` 进行预热，把 undo space page 预热到 buffer pool 中。