Mysql Innodb Insert Buffer/Checkpoint/Aio 实现分析

网易杭研 何登成

1 准备 3

2 INSERT BUFFER 3

2.1 INSERT BUFFER 流程 4

2.2 INSERT BUFFER MERGE 流程 5

2.2.1 主动 Merge 5

2.2.1.1 主动 Merge 原理 5

2.2.1.2 步骤一：异步 I/O 流程 6

2.2.1.3 步骤二：Merge 流程 7

2.2.2 被动 Merge 8

2.2.2.1 被动 Merge-情况一 8

2.2.2.2 被动 Merge-情况二 8

2.2.2.3 被动 Merge-情况三 9

2.2.2.4 被动 Merge-情况四 9

3 CHECKPOINT 10

3.1 CHECKPOINT 原理 10

3.2 CHECKPOINT 触发条件 10

3.3 CHECKPOINT 流程 10

3.3.1 计算脏页比率 10

3.3.2 计算 adaptive flush rate 11

3.3.3 flush dirty pages 算法 11

3.4 CHECKPOINT INFO 更新 13

3.4.1.1 流程一 13

3.4.1.2 流程二 14

3.5 INNODB_FLUSH_METHOD 15

3.5.1 初始化 15

3.5.2 open file 16

3.5.3 flush data 17

3.5.4 未明之处 17

4 INNODB AIO 17

4.1 聚簇索引 IO 18

4.2 UNIQUE 索引 IO 20

4.3 NON-UNIQUE 索引 IO 20

5 PERCONA 版本优化 21

5.1 FLUSH & CHECKPOINT 优化 21

5.1.1 reflex 22

5.1.2 estimate 22

5.1.3 keep_average 23

5.2 INSERT BUFFER & MERGE 优化 24
5.2.1 innodb_ibuf_accel_rate *24*
5.2.2 innodb_ibuf_active_contract *25*
6 参考文献 25

目的：

测试、掌握 Innodb 如何实现 Insert Buffer，如何实现 Checkpoint？如何调用实现 Aio？是否有可优化之处？

1 准备

基于版本：

Mysql 5.5.16

Mysql 5.6.4

测试表结构：

```
mysql> show create table nkeys;
+-------+-----------------------------
| Table | Create Table
+-------+-----------------------------
| nkeys | CREATE TABLE `nkeys` (
  `c1` int(11) NOT NULL,
  `c2` int(11) DEFAULT NULL,
  `c3` int(11) DEFAULT NULL,
  `c4` int(11) DEFAULT NULL,
  `c5` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`),
  UNIQUE KEY `c4` (`c4`),
  KEY `nkey1` (`c3`,`c5`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk |
+-------+-----------------------------
```

并且在 nkeys 表中预先插入 50000 条数据，保证索引有两层。

2 Insert Buffer

Insert Buffer，是 Innodb 处理非唯一索引更新操作时的一个优化。

Insert Buffer，经历多次的版本变迁，其功能越来越强。最早的 Insert Buffer，仅仅实现 Insert 操作的 Buffer，这也是 Insert Buffer 名称的由来。在后续版本中，Innodb 多次对 Insert Buffer 进行增强，到 Innodb 5.5 版本，Insert Buffer 除了支持 Insert，还新增了包括 Update/Delete/Purge 等操作的 buffer 功能，Insert Buffer 也随之更名为 Change Buffer。但是在 Innodb5.5-5.6 的代码之中，Insert Buffer 对应的文件仍旧是 ibuf，所有的函数，也都以 ibuf 前缀命名。

Mysql 5.5 的 Insert Buffer 功能，可参考文档：MYSQL 5.5: InnoDB Change Buffering [1].

2.1 Insert Buffer 流程

insert into nkeys values (20,20,20,20,20);
函数调用流程 (针对与 nkeys 表的 nkey1 索引，其余两个索引，一个主键，一个 Unique，无法使用 insert buffer)：
write_row -> ha_innobase::write_row -> row_insert_for_mysql -> … -> row_insert_entry_low ->

1. 插入 nkey1 索引，准备阶段函数流程

btr0cur.cc::btr_cur_search_to_nth_level ->

2. insert buffer 功能，在 search path 函数中完成

ibuf_should_try ->

a) 判断当前索引，是否可以使用 insert buffer。非主键索引，非唯一索引，可以使用 insert buffer

buf_page_get_gen ->

b) 读取页面，若叶页面不在 buffer pool 中，同时可以进行 insert buffer，则返回 NULL

ibuf_insert -> ibuf_insert_low -> ibuf_entry_build ->
btr_pcur_open(btr_pcur_open_func - > btr_cur_search_to_nth_level) ->
ibuf_get_volume_bufferd ->
ibuf_bitmap_get_map_page -> ibuf_bitmap_page_get_bits -> ibuf_index_page_calc_free_from_bits ->

c) 叶页面不在 buffer pool 之中，进行 insert buffer

d) ibuf_entry_build: 构造 insert buffer 中的记录，记录组织结构如下：

i. 4 bytes：space_id
ii. 1 byte： marker = 0
iii. 4 bytes：page number
iv. type info：
    1. 2 bytes：counter，标识当前记录属于同一页面中的第几条 insert buffer 记录
    2. 1 byte：操作类型：IBUF_OP_INSERT; IBUF_OP_DELETE_MARK; IBUF_OP_DELETE;
    3. 1 byte： Flags. 当前只能是 IBUF_REC_COMPACT
v. entry fields：之后就是索引记录
vi. 由于前 9 个字节[space_id, marker, page_numer, counter]组合，前三个字段，相同页面是一样的，这也保证了相同页面的记录，一定是存储在一起。第四个字段，标识页面中的第几次更新，保证同一页面 buffer 的操作，按照顺序存储。

e) btr_pcur_open: 根据 insert buffer 表 SYS_IBUF_TABLE 的索引 CLUST_IND，进行 search path 找到当前记录应该操作的 insert buffer 页面
(不确定，想来不是，若 SYS_IBUF_TABLE 为内存表，那么也应该不需要 os_aio_ibuf_array 才对)SYS_IBUF_TABLE，CLUST_IND 是内存表与内存索引，因此并

不会被持久化到外存；但是也需要记录日志，当系统崩溃时，需要恢复，并且 merge 到对应的索引中。

f) ibuf_get_volume_bufferd: 在 insert buffer 中已存在的项，同时返回这些项占用的空间大小 buffered。首先遍历当前页面的前页面，比较前页面中的项，若[space_id, page_num]相同，则增加 buffered；然后遍历当前页面的后页面，同样增加相同页面的项。

g) 根据 bitmap，计算索引页面中的空余空间，是否足够存放当前记录，并且不引起页面分裂

i. buffered + entry_size + reserved_space <= ibuf_index_page_calc_free_from_bits

ibuf_insert -> ibuf_insert_low -> btr_cur_optimistic_insert ->

a) 将 entry 插入到 SYS_IBUF_TABLE 系统表之中，该系统表实现了 insert buffer 的管理功能

b)

3.

2.2 Insert Buffer Merge 流程

2.2.1 主动 Merge

2.2.1.1 主动 Merge 原理

主动 merge 在 Innodb 主线程(srv0srv.c::srv_master_thread)中判断，判断原理很简单易懂：

若过去 1s 之内发生的 I/O，小于系统 I/O 能力的 5%，则主动进行一次 Insert buffer 的 merge 操作。Merge 的页面数为系统 I/O 能力的 5%，读取 page 采用 async io 模式。

每 10s，必定触发一次 insert buffer merge 动作。Merge 的页面数仍旧为系统 I/O 能力的 5%。

函数代码段如下：

```
buf_get_total_stat(&buf_stat);
n_pend_ios = buf_get_n_pending_ios()
    + log_sys->n_pending_writes;
n_ios = log_sys->n_log_ios + buf_stat.n_pages_read
    + buf_stat.n_pages_written;
if (n_pend_ios < SRV_PEND_IO_THRESHOLD
    && (n_ios - n_ios_old < SRV_RECENT_IO_ACTIVITY)) {
    srv_main_thread_op_info = "doing insert buffer merge";
    ibuf_contract_for_n_pages(FALSE, PCT_IO(5));
```

```
            /* Flush logs if needed */
            srv_sync_log_buffer_in_background();
        }
```

n_pend_ios:　　系统目前 pend 的 I/O 操作数

n_ios:　　系统启动到目前为止一共进行的 I/O 操作数

SRV_PEND_IO_THRESHOLD:　　系统 pend 的 I/O 上限

SRV_RECENT_IO_ACTIVITY:　　系统当前一段时间之内的活跃 I/O 数

```
#define SRV_PEND_IO_THRESHOLD     (PCT_IO(3))
#define SRV_RECENT_IO_ACTIVITY    (PCT_IO(5))
#define PCT_IO(p) ((ulong) (srv_io_capacity * ((double) p / 100.0)))
/* Number of IO operations per second the server can do */
UNIV_INTERN ulong  srv_io_capacity          = 200;
```

系统的 I/O 能力，Innodb 默认设置为 200，可以根据自身的系统进行相应的调整

在清楚主动 Merge 操作的原理之后，接下来分析主动 Merge 操作的实现。主动 merge 的实现流程，主要分为两步：

步骤一：　主线程发出异步 I/O 请求，异步读取需要被 merge 的页面

步骤二：　I/O handler 线程，在接收到完成的异步 I/O 之后，进行 merge

2.2.1.2 步骤一：异步 I/O 流程

主线程调用函数 ibuf_contract_for_n_pages 进行索引页面的异步 I/O 读取，进行 insert buffer 的 merge 操作。

函数 ibuf_contract_for_n_pages 流程如下：

srv0srv.c::srv_master_thread ->

ibuf0ibuf.c::ibuf_contract_for_n_pages(系统能力的 5%，200*5% = 10 个 page) ->

ibuf_contract_ext -> btr_pcur_open_at_rnd_pos -> ibuf_get_merge_page_nos ->

- 随机定位一个 insert buffer 的页面，读取页面中所有需要合并的 insert buffer 记录，以及记录对应的 space_id，page_no 至 space_ids，page_nos 数组之中

buf0rea.c::buf_read_ibuf_merge_pages -> buf_read_page_low ->

- 将数组中的(space_id, page_no)组合悉数读出

fil_io -> os_aio_func(type, mode) ->

- 具体 Innodb aio 的流程分析，可见 Innodb Aio 章节。
- 此处，type = OS_FILE_READ; mode = OS_AIO_NORMAL; 使用 os_aio_read_array
  由于 mode != OS_AIO_SYNC，因此此处发出 AIO 命令之后，不需要等待 I/O 操作完成，直接返回即可。
  AIO 完成之后，io_handler_thread 线程将会接收到 I/O 完成的信号

(os_aio_windows/linux_handle 函数)，处理余下的 insert buffer merge 操作，就是接下来将要分析的 步骤二：Merge 流程。

2.2.1.3 步骤二：Merge 流程

Innodb 的 io_handler_thread 线程，在接收到主线程发出的异步 I/O 完成的信号之后，对页面进行 merge 操作。

执行：insert into nkeys values (60,60,60,60,60); 索引 nkey1 会使用 insert buffer，在 insert buffer 操作完成之后，io_handler_thread 线程调度，将记录 merge 到原有页面。

函数调用流程如下：

srv0start.c::io_handler_thread ->

1. innodb 的 io 线程，在数据库启动(innodb/innobase_start_or_create_for_mysql)时创建，通过参数 innodb/innobase_file_io_threads 参数控制 io 线程的数量.

fil0fil.c::fil_aio_wait() ->

2. 等待 asyc io，根据 block 类型进行分发，buf_page_io_complete or log_io_complete ?

os0file.cc::os_aio_linux/windows_handle(&fil_node, &message) ->

3. 调用操作系统相关的方法，完成 aio 操作，并填充 file 头与 block 头

buf0buf.c::buf_page_io_complete(message) ->

4. message 参数，fil_io_wait 传入，buf0buf.h::buf_page_struct 结构，block 通用头结构，其前两个属性为 space:32, offset:32，分别为 0，405，就是 insert buffer 中对应的 nkey1 索引的 page。

ibuf0ibuf.c::ibuf_merge_or_delete_for_page -> ibuf_bitmap_page_get_bits(判断当前页面是否存在 insert buffer 项) -> ibuf_new_search_tuple_build(insert buffer 记录定位) -> btr_pcur_open_on_user_rec(index scan，在 insert buffer 中查找第一条记录) -> page_update_max_trx_id -> ibuf_insert_to_index_page -> ibuf_delete_rec

5. 调用此函数，将 insert buffer 中的修改，merge 到原有 page 之中(0, 405).
   a) 首先判断当前页面是否存在 Insert buffer 项
   b) 根据页面 space_id，page_no 构造 search key 定位到 insert buffer 记录，search key 为 insert buffer 记录的前三个属性(space_id, marker, page_no)
   c) 修改 index page 中的 max_trx_id 系统列
   d) 构造完整索引项，并插入到 index page 之中
   e) 删除 ibuf 中的记录
   f) 设置 ibuf bitmap

buf_pool->n_pend_reads--;

buf_pool->stat.n_pages_read++;

6. 一次 aio merge 操作完成，将 n_pend_reads 参数减减
n_pend_reads 参数，在 buf_page_get_gen -> buf_read_page -> buf_read_page_low -> buf_page_init_for_read 函数中设置。

2.2.2 被动 Merge

上一章节提到的主动 Merge，指的是 Innodb 系统在主线程中，定期主动尝试读取索引的 page，然后将 insert buffer 中的修改 merge 到对应的 page 之中。用户线程无法感知。

而我所谓的被动 Merge，则主要是指在用户线程执行的过程中，由于种种原因，需要将 insert buffer 的修改 merge 到 page 之中。被动 Merge 由用户线程完成，因此用户能够感知到 merge 操作带来的性能影响。

被动 Merge 主要有以下几种情况.

2.2.2.1被动 Merge-情况一

Insert 操作，导致页面空间不足，需要分裂。由于 insert buffer 只能针对单页面，不能 buffer page split，因此引起页面的被动 Merge。
函数判断流程如下：

ibuf0ibuf.cc::ibuf_insert_low

```
do_merge = FALSE;
if (buffered + entry_size + reserved_space <= ibuf_index_page_calc_free_from_bits)
    do_merge = TRUE;
    ibuf_get_merge_page_nos();
func_exit:
if (do_merge)
    buf_read_ibuf_merge_pages();
```

在 btr0cur.cc:: btr_cur_search_to_nth_level 函数中，若判断出 insert buffer 失败，则会将 buf_mode 设置为 BUF_GET，必定读取 page，然后重新进行 search path，读取当前页面。

同理，还有 update 操作导致页面空间不足；purge 导致页面为空等。
换言之，若当前操作引起页面 split or merge，那么就会导致被动 Merge。

2.2.2.2被动 Merge-情况二

insert 操作，由于其他各种原因，insert buffer 优化返回失败，需要真正读取 page 时，也需要进行被动 Merge
代码处理流程如下：

buf0buf.c::buf_page_get_gen

```
if (UNIV_LIKELY(!recv_no_ibuf_operations))
    ibuf_merge_or_delete_for_page(block, space, offset, zip_size, TRUE);
```

参数 recv_no_ibuf_operations 在恢复阶段设置为 TRUE，正常运行阶段设置为 FALSE；
因此正常运行阶段，如果读取了一个 ZIP_PAGE，就需要判断其是否应该做 insert bufferMerge

情况二与情况一的不同之处在于，情况一判断出页面 split 之后，会自动进行一次 Merge，search path restart 时，page 已经在内存之中；情况二，页面仍旧在 disk 上，读取之后，判断页面类型为 ZIP_PAGE，解压之后，进行一次 Merge 操作。

2.2.2.3被动 Merge-情况三

在进行 insert buffer 操作时，发现 insert buffer 已经太大，需要压缩 insert buffer
判断流程如下：

```
if (ibuf->size >= ibuf->max_size + IBUF_CONTRACT_DO_NOT_INSERT)
    ibuf_contract(sync = TRUE);
        ibuf_contract_ext();
            btr_pcur_open_at_rnd_pos(ibuf->index, BTR_SEARCH_LEAF, &pcur, &mtr);
            buf_read_ibuf_merge_pages(sync, space_ids, space_versions, page_nos,
              *n_pages);
```

- ibuf->max_size

```
ibuf->max_size = buf_pool_get_curr_size() / UNIV_PAGE_SIZE
    / IBUF_POOL_SIZE_PER_MAX_SIZE;
```

IBUF_POOL_SIZE_PER_MAX_SIZE = 2;
因此 ibuf 的最大大小为 buf_pool 的 1/2

- IBUF_CONTRACT_DO_NOT_INSERT = 10
  超过 ibuf 最大大小 10 个 page，需要强制进行被动 Merge

- sync = TRUE
  Merge 操作同步 I/O，不允许 Insert 操作进行

- 算法
  在 insert buffer tree 中随机定位一个页面，将该页面中 buffer 的更新全部合并到原有 page 之中，并且返回最终 merge 了多少个 page。

2.2.2.4被动 Merge-情况四

3 Checkpoint

3.1 Checkpoint 原理

关于 Innodb Checkpoint 的原理，此处不准备介绍，推荐 How InnoDB performs a checkpoint [2] 一文，作者详细讲解了 Innodb 的 Checkpoint 原理。

3.2 Checkpoint 触发条件

- 每 1S
  - 若 buffer pool 中的脏页比率超过了 srv_max_buf_pool_modified_pct = 75，则进行 Checkpoint，刷脏页，flush PCT_IO(100)的 dirty pages = 200
  - 若采用 adaptive flushing，则计算 flush rate，进行必要的 flush
- 每 10S
  - 若 buffer pool 中的脏页比率超过了 70%，flush PCT_IO(100)的 dirty pages
  - 若 buffer pool 中的脏页比率未超过 70%，flush PCT_IO(10)的 dirty pages = 20

关于 PCT_IO 宏定义，详情可见 2.2.1.1 主动 Merge 原理 章节。

Innodb 如何计算脏页比率？adaptive flushing 时如何计算 flush rate？如何进行真正的 flush 操作，是否使用 AIO，将在以下章节中一一分析。

3.3 Checkpoint 流程

3.3.1 计算脏页比率

srv0srv.c::srv_master_thread -> buf0buf.c::buf_get_modified_ratio_pct -> buf_get_total_list_len

```
    for (i = 0; i < srv_buf_pool_instances; i++) {
        buf_pool_t*    buf_pool;

        buf_pool = buf_pool_from_array(i);
        *LRU_len += UT_LIST_GET_LEN(buf_pool->LRU);
        *free_len += UT_LIST_GET_LEN(buf_pool->free);
        *flush_list_len += UT_LIST_GET_LEN(buf_pool->flush_list);
    }
ratio = (100 * flush_list_len) / (1 + lru_len + free_len);
```

脏页比率 = 需要被 flush 的页面数 /(使用中的页面数 + 空闲页面数 + 1)

其中，所有的值，在 buf_pool_t 结构中均有统计，无需实际遍历 buffer pool 进行计算

3.3.2 计算 adaptive flush rate

函数流程：

buf0buf.c::buf_flush_get_desired_flush_rate ->

1. 从 buf_pool_t 结构中，获得总 dirty page 的数量

2. 计算最近一段时间之内，redo 日志产生的平均速度

```
redo_avg = (ulint) (buf_flush_stat_sum.redo
    / BUF_FLUSH_STAT_N_INTERVAL
    + (lsn - buf_flush_stat_cur.redo));
```

其中，BUF_FLUSH_STAT_N_INTERVAL = 20S，20S 内的平均 redo 产生速度

```
/** Number of intervals for which we keep the history of these stats.
Each interval is 1 second, defined by the rate at which
srv_error_monitor_thread() calls buf_flush_stat_update(). */
#define BUF_FLUSH_STAT_N_INTERVAL 20
```

flush 的统计信息，每隔 20S 会被 buf_flush_stat_update 函数重置

3. 计算过去一段时间内，flush 的平均速度；与当前需要的 flush 速度

```
lru_flush_avg = buf_flush_stat_sum.n_flushed
        / BUF_FLUSH_STAT_N_INTERVAL
        + (buf_lru_flush_page_count
           - buf_flush_stat_cur.n_flushed);

n_flush_req = (n_dirty * redo_avg) / log_capacity;
```

其中，BUF_FLUSH_STAT_N_INTERVAL = 20S 不变，计算的仍旧是过去 20S 内的平均 flush 速度

4. 若当前所需速度 > 20S 内的平均速度，则 adaptive flushing 会尝试进行一次 flush 操作。flush 的 dirty pages 数量仍旧是 PCT_IO(100)，200 个 dirty pages。

3.3.3 flush dirty pages 算法

函数流程：

buf0flu.c::buf_flush_list -> buf_flush_start -> buf_flush_batch -> buf_flush_end -> buf_flush_common

1. 首先，判断当前是否有正在进行的相同类型的 flush (buf_flush_start)，有则直接退出

2. **buf_flush_batch** -> buf_flush_flush_list_batch -> buf_flush_page_and_try_neighbors ->

buf_flush_try_neighbors -> buf_flush_page -> buf_flush_buffered_writes ->

a) 从 flush_list 的最后一个页面开始，向前遍历页面 & flush

b) 对于 a)中的 page，尝试 flush，并且尝试 flush 该 page 的 neighbors pages (buf_flush_try_neighbors)

i. 首先计算可选的 neighbors 范围。所谓 neighbors 范围，指的是 space_id 相同，page_no 不同的 page，只有这些 page 才是连续的。

```
buf_flush_area      = ut_min(
                    ut_min(64, ut_2_power_up((b)->curr_size / 32)),
                    buf_pool->curr_size / 16);

low = (offset / buf_flush_area) * buf_flush_area;
high = (offset / buf_flush_area + 1) * buf_flush_area;
```

low 为当前 page，neighbors 的范围既为 buf_flush_area，最大 64 neighbors 为当前 page 开始，page_no 连续递增的 buf_flush_area 个 pages

ii. 所有的 dirty pages，在 buffer pool 中同时以 hash 表存储，根据(space_id, [page_no_low, page_no_high])到 hash 表中进行查找，若存在，则 flush 此 dirty page

3. buf_flush_page -> buf_flush_write_block_low -> log_write_up_to ->

a) flush 脏页之前，必须保证脏页对应的日志已经写回日志文件(log_write_up_to)

b) 判断是否需要使用 double write

i. 若不需要 double write 保护，直接调用 fil_io 进行 flush 操作，设置 type = OS_FILE_WRITE；mode = OS_AIO_SIMULATED_WAKE_LATER

ii. 若需要 double write 保护，则调用 buf_flush_post_to_doublewrite_buf 函数

1. 写到 double write 就算完成，退出 buf_flush_page

4. buf_flush_batch -> buf_flush_buffered_writes

a) buf_flush_batch 函数，在完成 2，3 步骤，batch flush 之后，调用 buf_flush_buffered_writes 函数进行真正的 write 操作

b) buf_flush_buffered_writes：将 double write memory 写出到 disk

i. 我的测试中，有 7 个 dirty pages，每个 page 大小为 16k = 16384，因此 doublewrite buffer 的大小为 16384 * 7 = 114688

ii. doublewrite buffer 的写，为同步写，调用 fil_io(OS_FILE_WRITE, TRUE)

iii. 同步写之后，调用 fil_flush 函数，将 doublewrite buffer 中的内容 flush 到 disk

1. windows：

```
FlushFileBuffers(file);
```

2. linux：

```
os_file_fsync(file);  or
fcntl(file, F_FULLFSYNC, NULL);
```

iv. 在 doublewrite buffer 被成功 flush 到 disk 之后，对应的 dirty pages 不会再丢失数据。此时再将 doublewrite buffer 对应的 dirty pages 写出到 disk

1. `fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER, FALSE);`
2. 写 dirty pages，采用非同步写 AIO

v. 在 dirty pages 都完成异步 IO 之后，调用 buf_flush_sync_datafiles 函数，将所有的异步 IO 操作，flush 到磁盘

```
/* Wake simulated aio thread to actually post the writes to the operating system */
os_aio_simulated_wake_handler_threads();
/* Wait that all async writes to tablespaces have been posted to the OS */
os_aio_wait_until_no_pending_writes();
/* Now we flush the data to disk (for example, with fsync) */
fil_flush_file_spaces(FIL_TABLESPACE);
```

5. 标识当前 flush 操作结束(buf_flush_end)

6. 收集当前 flush 操作的统计信息(buf_flush_common)

3.4 Checkpoint info 更新

在完成 Checkpoint 流程中的 flush dirty pages 之后，Innodb Checkpoint 的大部分流程已经完成，只余下最后的修改 Checkpoint Info 信息。

3.4.1.1流程一

更新 Checkpoint Info 流程一，流程一每 10S 调用一次：
srv_master_thread -> log0log.c::log_checkpoint ->
log_buf_pool_get_oldest_modification ->

- 读取系统中，最老的日志序列号。实现简单，读取 lsn flush list 中最老日志对应的 lsn 即可

log_write_up_to(oldest_lsn, LOG_WAIT_ALL_GROUPS, TRUE) ->

- 将日志 flush 到 oldest_lsn

log_groups_write_checkpoint_info -> log_group_checkpoint ->
fil_io(OS_FILE_WRITE | OS_FILE_LOG, FALSE) ->

- 遍历所有日志组，分别更新每个日志组对应的 Checkpoint Info
- 构造 Checkpoint Info，使用 os_aio_log_array 进行异步写 I/O 操作

3.4.1.2流程二

更新 Checkpoint Info 流程二，在 I/O 较为繁忙的系统中，流程二每 1S 调用一次：

srv_master_thread -> log0log.ic::log_free_check -> log0log.c::log_check_margins -> log_checkpoint_margin -> log_preflush_pool_modified_pages -> log_checkpoint

- 读取当前日志系统中的最老日志序列号 lsn
  根据 oldest_lsn 与 log->lsn(current lsn)之间的差距，判断日志空间是否足够，是否需要进行 flush dirty pages 操作

- 读取当前日志系统中最老的 Checkpoint Lsn
  根据 last_checkpoint_lsn 与 log->lsn 之间的差距，判断是否需要向前推进检查点

- 若需要 flush dirty pages，调用函数 log_preflush_pool_modified_pages
  log_preflush_pool_modified_pages -> buf_flush_list(ULINT_MAX, new_oldest)
  - new_oldest 参数，指定当前将 dirty pages flush 到何 lsn？sync 参数指定当前 flush 操作是否为同步操作？由函数 log_checkpoint_margin 计算，代码如下：

```
oldest_lsn = log_buf_pool_get_oldest_modification();
age = log->lsn - oldest_lsn;
if (age > log->max_modified_age_sync) {
     /* A flush is urgent: we have to do a synchronous preflush */
     sync = TRUE;
     advance = 2 * (age - log->max_modified_age_sync);
} else if (age > log->max_modified_age_async) {
     /* A flush is not urgent: we do an asynchronous preflush */
     advance = age - log->max_modified_age_async;
} else {
     advance = 0;
}
ib_uint64_t   new_oldest = oldest_lsn + advance;
if (checkpoint_age > log->max_checkpoint_age) {
     /* A checkpoint is urgent: we do it synchronously */
     checkpoint_sync = TRUE;
     do_checkpoint = TRUE;
}
```

  - log->max_modified_age_(a)sync; log->max_checkpoint_age
    以上参数用于控制是否需要进行 log flush，以及是否需要进行 Checkpoint。
    参数的计算，在 log0log.c::log_calc_max_ages 函数中完成，代码较为简单，如下所示：

```
     margin = smallest_capacity - free;
     margin = margin - margin / 10;   /* Add still some extra safety */
     log->log_group_capacity = smallest_capacity;
```

```
log->max_modified_age_async = margin-margin / LOG_POOL_PREFLUSH_RATIO_ASYNC;
log->max_modified_age_sync = margin - margin / LOG_POOL_PREFLUSH_RATIO_SYNC;
log->max_checkpoint_age_async = margin-margin/LOG_POOL_CHECKPOINT_RATIO_ASYNC;
log->max_checkpoint_age = margin;

#define LOG_POOL_CHECKPOINT_RATIO_ASYNC    32
#define LOG_POOL_PREFLUSH_RATIO_SYNC       16
#define LOG_POOL_PREFLUSH_RATIO_ASYNC      8
```

简单来说，margin 近似认为是 Innodb 系统可用的日志空间的 9/10；
日志空间消耗超过 7/8 时，一定要进行异步 Flush 日志；
日志空间消耗超过 15/16 时，一定要进行同步 Flush 日志；
日志空间消耗超过 31/32 时，一定要进行异步 Flush Buffer Pool；
日志空间消耗达到 margin 上限时，一定要进行同步 Flush Buffer Pool

以上判断均在 log_checkpoint_margin 函数中完成，1S 中判断一次。

- 若需要向前推进检查点，调用函数 log_checkpoint，log_checkpoint 函数的流程，在前一章节中已经分析。

3.5 innodb_flush_method

无论是数据文件，还是日志文件，在完成 write 操作之后，最后都需要 flush 到 disk。

是否 flush？如何进行 flush？日志文件与数据文件的 flush 操作有何不同？通过参数 innodb_flush_method 控制。

关于 innodb_flush_method 这个参数的意义及设置，网上有大量的文档。具体可参考 innodb_flush_method 与 File I/O 与 SAN vs Local-disk：：innodb_flush_method performance benchmarks 两文。

接下来我主要从源码层面简单分析以下 innodb_flush_method 参数的使用(在 innodb 5.5-5.6 中，此参数的名字修改为 innobase_file_flush_method)。

3.5.1 初始化

函数处理流程：
srv0start.cc::innobase_start_or_create_for_mysql

```
if (srv_file_flush_method_str == NULL) {
    /* These are the default options */
    srv_unix_file_flush_method = SRV_UNIX_FSYNC;
    srv_win_file_flush_method = SRV_WIN_IO_UNBUFFERED;
```

```
} else if (0 == ut_strcmp(srv_file_flush_method_str, "fsync")) {
        srv_unix_file_flush_method = SRV_UNIX_FSYNC;
} else if (0 == ut_strcmp(srv_file_flush_method_str, "O_DSYNC")) {
        srv_unix_file_flush_method = SRV_UNIX_O_DSYNC;
} else if (0 == ut_strcmp(srv_file_flush_method_str, "O_DIRECT")) {
        srv_unix_file_flush_method = SRV_UNIX_O_DIRECT;
} else if (0 == ut_strcmp(srv_file_flush_method_str, "littlesync")) {
        srv_unix_file_flush_method = SRV_UNIX_LITTLESYNC;
} else if (0 == ut_strcmp(srv_file_flush_method_str, "nosync")) {
        srv_unix_file_flush_method = SRV_UNIX_NOSYNC;
}
```

根据用户指定的 srv_file_flush_method_str 的不同，设置 srv_unix_file_flush_method 的不同取值，innodb 内部，通过判断此参数，来确定以何种模式 open file，以及是否 flush write。

简单起见，此处只拷贝了 linux 部分处理代码，未包括 windows 部分。

3.5.2 open file

Innodb 系统启动阶段，设置完成 srv_unix_file_flush_method 参数之后，可以进行 I/O 操作，I/O 操作的总入口为函数 fil0fil.c::fil_io，相信大家已经看过上面的分析之后，对此函数不会陌生。

fil0fil.c::fil_io 函数中，处理了 file open 的过程，函数流程如下：
fil0fil.c::fil_io -> fil_node_perpare_for_io -> fil_node_open_file ->

```
if (space->purpose == FIL_LOG) {
    node->handle = os_file_create(innodb_file_log_key, node->name, OS_FILE_OPEN,
                        OS_FILE_AIO, OS_LOG_FILE, &ret);
} else if (node->is_raw_disk) {
    node->handle = os_file_create(innodb_file_data_key, node->name,
                        OS_FILE_OPEN_RAW, OS_FILE_AIO, OS_DATA_FILE, &ret);
} else {
    node->handle = os_file_create(innodb_file_data_key, node->name, OS_FILE_OPEN,
                        OS_FILE_AIO, OS_DATA_FILE, &ret);
}
```

根据当前文件类型不同，底层依赖的硬件环境不同，调用 os_file_create 宏定义 open 对应的文件。os_file_create 宏定义对应的函数是 os_file_create_func.

os_file_create_func 函数处理 open file 的流程：

- Log file 将 O_DSYNC 转化为 O_SYNC，O_DSYNC 设置只对 data file 有用

```
if (type == OS_LOG_FILE && srv_unix_file_flush_method == SRV_UNIX_O_DSYNC) {
        create_flag = create_flag | O_SYNC;
file = open(name, create_flag, os_innodb_umask);
```

- Data file 与 O_DIRECT 组合，需要禁用底层 os file cache

```
/* We disable OS caching (O_DIRECT) only on data files */
if (type != OS_LOG_FILE && srv_unix_file_flush_method == SRV_UNIX_O_DIRECT)
        os_file_set_nocache(file, name, mode_str);
```

3.5.3 flush data

fil0fil.c::fil_io 函数打开 file 之后，可以进行 file 的 write 与必要的 flush 操作，write 操作在前面的章节中已经分析，本章主要看 srv_unix_file_flush_method 参数对于 flush 操作的影响。

- srv_unix_file_flush_method = SRV_UNIX_NOSYNC
  无论是 log file，还是 data file，一定只 write，但不 flush
- srv_unix_file_flush_method = SRV_UNIX_O_DSYNC
  Log file： 不 flush

```
if (srv_unix_file_flush_method != SRV_UNIX_O_DSYNC
        && srv_unix_file_flush_method != SRV_UNIX_NOSYNC)
        fil_flush(group->space_id);
```

  当然，其他情况下，Log file 是否一定 flush？还与参数 srv_flush_log_at_trx_commit 的设置有关
  Data file：
- srv_unix_file_flush_method = SRV_UNIX_LITTLESYNC
  Data file： 不 flush
- srv_unix_file_flush_method =

3.5.4 未明之处

innodb_flush_method 参数，在使用系统 native aio 时，好像对于 data file 完全无影响，还需要进一步的理解与调研。

4 Innodb Aio

insert into nkeys values (71,71,71,71,71);

4.1 聚簇索引 IO

insert 操作，读取聚簇索引页面，函数调用流程：
buf_page_get_gen -> buf_read_page -> buf_read_page_low -> fil_io ->

os_aio_func(type, mode) ->

1. type = OS_FILE_READ; mode = OS_AIO_SYNC; 使用 os_aio_sync_array (其余的 array 包括：os_aio_read_array; os_aio_write_array; os_aio_ibuf_array; os_aio_log_array)
   a) 每个 aio array，在系统启动时调用 os0file.c::os_aio_init 函数初始化

```
os_aio_init(io_limit,
        srv_n_read_io_threads,
        srv_n_write_io_threads,
        SRV_MAX_N_PENDING_SYNC_IOS);
```

   b) io_limit： 每个线程可以并发处理多少 pending I/O
   windows -> io_limit = SRV_N_PENDING_IOS_PER_THREAD = 32
   linux -> io_limit = 8 * SRV_N_PENDING_IOS_PER_THREAD = 8 * 32

```
#define SRV_N_PENDING_IOS_PER_THREAD OS_AIO_N_PENDING_IOS_PER_THREAD = 32
```

   c) srv_n_read_io_threads
   处理异步 read I/O 线程的数量
   innobase_read_io_threads/innodb_read_io_threads：通过参数控制
   因此系统可以并发处理的异步 read page 请求为：
   io_limit * innodb_read_io_threads

```
os_aio_read_array = os_aio_array_create(n_read_segs * n_per_seg, n_read_segs);
```

   异步 I/O 主要包括两大类：
   一、预读 page
   需要通过异步 I/O 方式进行
   二、主动 Merge
   Innodb 主线程对需要 merge 的 page 发出异步读操作，在 read_thread 中进行实际 merge 处理

   注：如何确定将哪些 read io 请求分配给哪些 read thread？
   1. 首先，每个 read thread 负责 os_aio_read_array 数组中的一部分。
   例如：thread0 处理 read_array[0, io_limit-1]；thread1 处理 read_array[io_limit, 2*io_limit - 1]，以此类推

   2. os_aio_array_reserve_slot 函数中实现了 array 的分配策略(array 未满时)。
   给定一个 Aio read page，[space_id, page_no]，首先计算 local_seg(local_thd):
   local_seg = (offset >> (UNIV_PAGE_SIZE_SHIFT + 6)) % array->n_segments;
   然后从 read_array 的 local_seg * io_limit 处开始向后遍历 array，直到找到一个空闲 slot。
   一来保证相邻的 page，能够尽可能分配给同一个 thread 处理，提高 aio(merge

io request)性能；
二来由于是循环分配，也基本上保证了每个 thread 处理的 io 基本一致。

d) srv_n_write_io_threads
处理异步 write I/O 线程的数量
innobase_write_io_threads/innodb_write_io_threads：通过参数控制
因此系统可以并发处理的异步 write 请求为：
io_limit * innodb_write_io_threads
超过此限制，必须将已有的异步 I/O 部分写回磁盘，才能处理新的请求。

e) SRV_MAX_N_PENDING_SYNC_IOS
同步 I/O array 的 slots 个数，同步 I/O 不需要处理线程

f) log thread，ibuf thread 个数均为 1

os_aio_array_reserve_slot ->

2. 在 aio array 中定位一个空闲 array，aio 前期准备工作
   a) array 已满
      i. native aio： os_wait_event(array->not_full); native aio，等待 not_full 信号
      ii. 非 native aio：os_aio_simulated_wake_handler_threads；模拟唤醒
   b) array 未满
      i. WIN_ASYNC_IO(windows AIO)
         设置 OVERLAPPED 结构，使用的是 Windows Overlapped I/O [5,6]
         ResetEvent(slot->handle)

      ii. LINUX_NATIVE_AIO(Linux AIO)
         设置 iocb 结构
         然后根据 type 判断： io_prep_pread or io_prep_pwrite [7]

3. 进行 aio 操作
   a) type = OS_FILE_READ
      i. use native aio
         1. windows: ReadFile

         2. Linux: os_aio_linux_dispatch(array, slot);
            a) 将 async io 请求发送至 linux kernel
            b) 调用 io_submit 函数进行 aio 发送

```
iocb = &slot->control;
io_ctx_index = (slot->pos * array->n_segments) / array->n_slots;
ret = io_submit(array->aio_ctx[io_ctx_index], 1, &iocb);
```

            c) iocb 结构在 slot 之中；io_context 结构，相同 segment 共用一个

      ii. use simulated aio

b) type = OS_FILE_WRITE
   i. use native aio
      1. windows: WriteFile
      2. Linux: os_aio_linux_dispatch(array, slot)
   ii. use simulated aio

os_aio_windows_handle

4. 特殊处理流程，windows 下，若 mode = OS_AIO_SYNC，则需要调用 os_aio_windows_handle 函数等待 aio 结束
   a) 判断当前是否为 sync_array
      i. 若是，等待指定的 slot aio 操作完成： WaitForSingleObject
      ii. 若不是，等待 array 中所有的 aio 操作完成： WaitForMultipleObjects

   b) 获取 aio 操作的结果
      i. GetOverlappedResult
   c) 最后释放当前 slot
      i. os_aio_array_free_slot

os_aio_linux_handle

5. 分析完 os_aio_windows_handle 函数，接着分析 Linux 下同样功能的函数：os_aio_linux_handle
   a) 无限循环，遍历 array，直到定位到一个完成的 I/O 操作(slot->io_already_done)为止
   b) 若当前没有完成的 I/O，同时有 I/O 请求，则进入 os_aio_linux_collect 函数
      i. os_aio_linux_collect：从 kernel 中收集更多的 I/O 请求
         1. 调用 io_getevents 函数，进入忙等，等待超时设置为 OS_AIO_REAP_TIMEOUT

```
/** timeout for each io_getevents() call = 500ms. */
#define OS_AIO_REAP_TIMEOUT   (500000000UL)
```

         2. 若 io_getevents 函数返回 ret > 0，说明有完成的 I/O，进行一些设置，最主要是将 slot->io_already_done 设置为 TRUE

```
slot->io_already_done = TRUE;
```

         3. 若系统 I/O 处于空闲状态，那么 io_thread 线程的主要时间，都在 io_getevents 函数中消耗。

4.2 Unique 索引 IO

与聚簇索引 IO 完全一致，因为二者都必须读取页面，不能进行 Insert Buffer 优化。

4.3 Non-unique 索引 IO

与聚簇索引 IO 不一致，区分流程在于函数 buf_page_get_gen

buf_page_get_gen

```
if (mode == BUF_GET_IF_IN_POOL || mode == BUF_PEEK_IF_IN_POOL || mode ==
BUF_GET_IF_IN_POOL_OR_WATCH)
    return NULL;
```

对于 non-unique 索引，此时 mode = BUF_GET_IF_IN_POOL；若 page 在 buffer pool 中，则返回 page，否则不立即读取 page，进行 insert buffer 优化。

由于不会调用 buf_read_page 函数，因此不会产生物理 IO。那么 non-unique 索引的页面何时会读入 buffer pool，与 insert buffer 进行 merge 呢？详见 主动 Merge 章节。

5 Percona 版本优化

关于 Percona XtraDB 的优化，推荐一篇十分好的文章：XtraDB: The Top 10 enhancements [11]. 该文详细列举了 XtraDB 对于原生 InnoDB 引擎做的最重要的 10 个优化，虽然文章是 2009 年 8 月写的，但是主要优化都已经存在了，每一个都值得一读。

当然，在本文中，我接下来主要讨论 XtraDB 在 Checkpoint 与 Insert Buffer 两个方面做的优化。Checkpoint 与 Insert Buffer 优化，都属于 XtraDB 优化中的一个大类：I/O 优化，可见网文：Improved InnoDB I/O Scalability [12].
增加 innodb_io_capacity 选项。原生 innodb 中的参数 srv_io_capacity，写死的是 200，XtraDB 中增加此选项，用户可以根据系统的硬件不同而设置不同的 io_capacity。但是，io_capacity 与系统的实际 I/O 能力还是有所区别，网文与其中的讨论：MYSQL 5.5.8 and Percona Server: being adaptive [13]给出了 fusion-io 下，innodb_io_capacity 选项具体如何设置更为合理。

源代码，参考的版本包括 Percona-Server-5.5.15-rel21.0; Percona-Server-5.5.18-rel23.0; Percona-Server-5.1.60;

5.1 Flush & Checkpoint 优化

XtraDB 对于 Flush & Checkpoint 的优化，主要在于新增了系统变量：innodb_adaptive_checkpoint
此变量可设置的值包括：none, reflex, estimate, keep_average，分别对应于 0/1/2/3；同时改变量要与 Innodb 自带的 innodb_adaptive_flushing 变量配合使用
关于每种设置的不同含义，[12]中有详尽介绍，此处给出简单说明：

- none
  原生 innodb adaptive flushing 策略
- reflex
  与 innodb 基于 innodb_max_dirty_pages_pct 的 flush 策略类似。不同之处在于，原生 innodb 是根据 dirty pages 的量来 flush；而此处根据 dirty pages 的 age 进行 flush。每次 flush 的 pages 根据 innodb_io_capacity 计算
- estimate

与 reflex 策略类似，都是基于 dirty page 的 age 来 flush。不同之处在于，每次 flush 的 pages 不再根据 innodb_io_capacity 计算，而是根据[number of modified blocks], [LSN progress speed]和[average age of all modified blocks]计算

- keep_average

  原生 Innodb 每 1S 触发一次 dirty page 的 flush，此参数降低了 flush 的时间间隔，从 1S 降低为 0.1S

注：在最新的 Percona XtraDB 版本中，reflex 策略已经被废弃；estimate，keep_average 策略的算法，或多或少也与网文中提到的有所出入，应该是算法优化后的结果，具体算法参考以下的几个小章节。

5.1.1 reflex

此策略，Percona XtraDB 5.1.60 版本中存在，但是在 5.5 版本中被删除，源代码级别彻底删除，可能并无太多的意义，功能与 estimate 重合，不建议使用。

5.1.2 estimate

函数处理流程：

```
// 1. innodb_adaptive_checkpoint 参数必须与 innodb_adaptive_flushing 同时设置
if (srv_adaptive_flushing && srv_adaptive_flushing_method == 1)
    // 2. 获取当前最老 dirty page 的 lsn
    oldest_lsn = buf_pool_get_oldest_modification();
    // 3. 若当前未 flush 的 dirty pages，超过 Checkpoint_age 的 1/4，则进行 flush
    if ((log_sys->lsn) – oldest_lsn) > (log_sys->max_checkpoint_age/4)
        // 4. estimate flush 策略
```

- 遍历 buffer pool 的 flush_list 链表，统计以下信息
  - n_blocks：　链表中的 page 数量
  - level：　链表所有 page 刷新的紧迫程度的倒数

    ```
    level += log_sys->max_checkpoint_age –
                (lsn – oldest_modification);
    ```

    最新修改的 page，level 贡献越大，紧迫程度越小；越老的 page，紧迫程度越大。

    关于 log_sys->max_checkpoint_age 的功能，可参考 Checkpoint Info 更新-流程二 章节。

- 需要 flush 的 dirty pages 数量 bpl，计算公式如下：

  ```
  bpl = n_blocks * n_blocks * (lsn – lsn_old) / level;
  ```

  其中：lsn_old 为上一次 flush 时记录下的 lsn

- 调用 buf_flush_list 函数，进行 flush

```
buf_flush_list(bpl, oldest_lsn + (lsn – lsn_old));
```

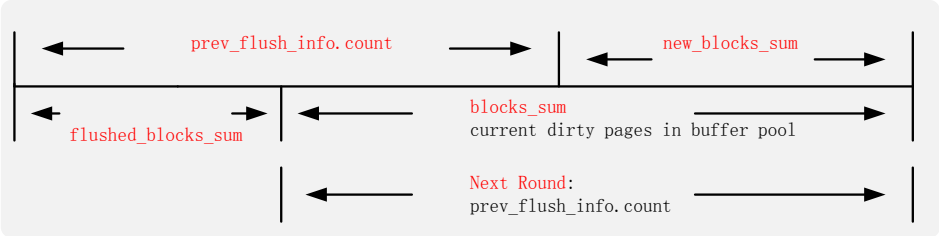
5.1.3 keep_average

keep_average 策略，将原生的 Innodb，每 1S flush 一次 dirty pages，改为每 0.1S 做一次。

```
if (srv_adaptive_flushing && srv_adpative_flushing_method == 2)
    next_itr_time -= 900;
```

Innodb，每次将 netx_itr_time 加 1000ms，然后 sleep 这 1000ms 时间。进入 keep_average 策略，将 next_itr_time 减去 900，那么下一次也就只会 sleep 100ms 时间。

接下来则是分析 keep_average 策略如何计算当前需要 flush 多少 dirty pages。下图能够较为清晰的说明 keep_average 策略：



图表 5-1 Percona keep_average flush 策略

图中名词解释：

prev_flush_info.count

上一次 flush 前，buffer pool 中的 dirty pages 数量

new_blocks_sum

上次记录 prev_flush_info.count 之后，系统新产生的 dirty pags

blocks_sum

当前系统，buffer pool 中的 dirty pages 数量

flushed_blocks_sum

```
flushed_blocks_sum = new_blocks_sum + prev_flush_info.count – blocks_sum;
```

上次的 flush 数量+循环间其余 flush 的数量

Next Round

本次 flush dirty pages 前，记录新的 prev_flush_info.count = blocks_sum

n_pages_flushed_prev

此参数未标出，表示上次 keep_average 策略成功 flush 了多少 dirty pages

计算本次应该 flush 的量：

```
n_flush = blocks_sum * (lsn – lsn_old) / log_sys->max_modified_age_async;
```

公式分析：

系统中的 dirty pages，有多少比率需要在此时被 flush

log_sys->max_modified_age_async

日志异步 flush 的临界值，若当前系统的 lsn 间隔大于此值，则启动异步 flush。

此参数为原生 Innodb 所有，Percona 此处借用来计算。

关于 log_sys->max_modified_age_async 参数在 Innodb 中的作用及设置，
可参考 Checkpoint Info 更新-流程二 章节。

flush 量微调：

```
if (flushed_blocks_sum > n_pages_flushed_prev)
    n_flush -= (flushed_blocks_sum - n_pages_flushed_prev);
```

若 flushed_blocks_sum > n_pages_flushed_prev，两次之间的实际 flush 量大于上次 keep_average 的 flush 量，那么本次 keep_average 需要 flush 量应相应减少。

5.2 Insert Buffer & Merge 优化

Percona XtraDB 除了优化 Checkpoint 策略，同时也对 Insert Buffer 的 Merge 操作做了部分优化。

优化 Insert Buffer 的目的，主要有两个：

- 目的一：加快 Insert Buffer 的 Merge 与内存的回收
  通过前面的分析可以发现，Insert Buffer 每次 Merge 操作，最多 Merge 一个 Insert Buffer 页面，繁忙的系统，很容易就达到了 Insert Buffer 可用内存的上限。

  针对目的一，Percona 增加了两个系统参数：innodb_ibuf_accel_rate 与 innodb_ibuf_active_contract

- 目的二：减少 Insert Buffer 的内存开销
  原生 Innodb 的 Insert Buffer，内存消耗上限为 Buffer pool 内存的一半。而在现阶段的应用中，大内存随处可见。因此有必要降低 Insert Buffer 的内存上限。

  针对目的二，Percona 增加了一个系统参数：innodb_ibuf_max_size

分析可看出，优化 Insert Buffer 的两个目的是相辅相成的。只有实现了目的一，才有减少 Insert Buffer 内存开销的可能。

5.2.1 innodb_ibuf_accel_rate

此系统参数的使用十分简单，参考 主动 Merge 章节，Percona 只是将每次 Merge 操作的 pages 数量，由宏定义 PCT_IO 修改为 PCT_IBUF_IO。而 PCT_IBUF_IO 的定义如下：

```
#define PCT_IBUF_IO(pct)    \
        ((ulint) (srv_io_capacity * srv_ibuf_accel_rate * ((double) pct / 10000.0)))
```

因此，如果想加快 Merge，只需要设置较大的 innodb_ibuf_accel_rate 参数即可。

5.2.2 innodb_ibuf_active_contract

与 innodb_ibuf_accel_rate 参数类似，innodb_ibuf_active_contract 参数的处理也是十分简单。只是在 ibuf0ibuf.c::ibuf_contract_after_insert 函数中，增加了一行判断：

```
if (!srv_ibuf_active_contract) {
    // #define IBUF_CONTRACT_ON_INSERT_NON_SYNC  0
    if (size < max_size + IBUF_CONTRACT_ON_INSERT_NON_SYNC) {
        return;
    }
}
```

绿色部分为 Innodb 的原生代码，每次 Insert 操作导致 Insert Buffer 的索引页面 split，产生 SMO 时，都会调用。若 Insert Buffer 的大小未超出上限，则不进行 Merge；

Percona 增加 innodb_ibuf_active_contract 参数之后，哪怕 Insert Buffer 未超上限，Insert Buffer split SMO 之后都会调用 Merge 功能。

6 参考文献

[1] http://blogs.innodb.com/wp/2010/09/mysql-5-5-innodb-change-buffering/ Mysql 5.5: InnoDB Change Buffering

[2] http://www.xaprb.com/blog/2011/01/29/how-innodb-performs-a-checkpoint/ How InnoDB performs a checkpoint

[3] http://www.percona.com/doc/percona-server/5.1/scalability/innodb_io.html?id=percona-server:features:innodb_io_51&redirect=2 Improved InnoDB I/O Scalability

[4] http://www.facebook.com/note.php?note_id=408059000932 InnoDB fuzzy checkpoints

[5] http://en.wikipedia.org/wiki/Overlapped_I/O Windows Overlapped I/O

[6] http://tinyclouds.org/iocp-links.html Asynchronous I/O in Windows for Unix Programmers

[7] http://tiaozhanshu.com/libaio-api.html Linux 下原生异步 I/O 接口 Libaio 的用法

[8] http://blog.csdn.net/zhaiwx1987/article/details/7165100 由 percona5.5 参数 innodb_adaptive_flushing_method 想到的….

[9] http://themattreid.com/wordpress/2012/01/06/san-vs-local-disk-innodb_flush_method-performance-benchmarks/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+Themattreid+%28TheMattReid+-+MySQL+DBA%29 SAN vs Local-disk :: innodb_flush_method performance benchmarks

[10] http://www.orczhou.com/index.php/2009/08/innodb_flush_method-file-io/ innodb_flush_method 与 File I/O

[11] http://www.mysqlperformanceblog.com/2009/08/13/xtradb-the-top-10-enhancements/ XtraDB: The Top 10 enhancements

[12] http://www.percona.com/docs/wiki/percona-xtradb:patch:innodb_io Improved InnoDB I/O Scalabilty
[13] http://www.mysqlperformanceblog.com/2010/12/20/mysql-5-5-8-and-percona-server-being-adaptive/ MYSQL 5.5.8 and Percona Server: being adaptive
[14]