MySQL InnoDB Insert Buffer/Checkpoint/AIO 实现分析

网易杭州研究院：何登成
Email： he.dengcheng@gmail.com
Weibo： 何_登成

1 准备

基于版本：

Mysql 5.5.16

Mysql 5.6.6

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

2.1.1 Insert Buffer for Insert

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

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) 函数 ibuf_bitmap_get_map_page，获取当前 insert 页面对应的 bitmap 管理页面，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

buffered + entry_size + reserved_space <= ibuf_index_page_calc_free_from_bit

ibuf_insert -> ibuf_insert_low -> btr_cur_optimistic_insert ->

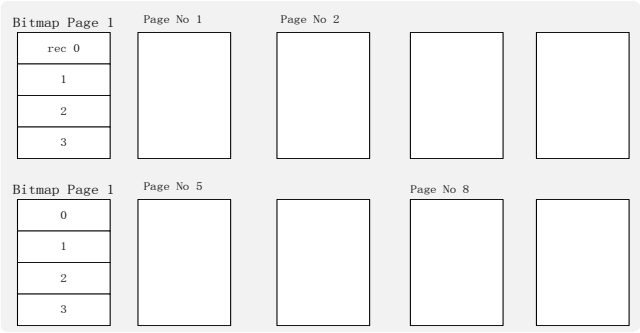
a) 将 entry 插入到 SYS_IBUF_TABLE 系统表之中，该系统表实现了 insert buffer 的管理功能

2.1.2 Ibuf Bitmap page

在上一章节，有一个流程，用于判断本次 Insert 操作是否会导致索引页面 split。若会 split，那么就不能进行 Insert Buffer 优化。并且简单的提到，bitmap 管理页面。此处，专门开出一小节，说一说 InnoDB 是如何管理每个页面的剩余空闲空间的？

InnoDB 采用所谓的 Ibuf Bitmap page 来管理页面剩余空闲空间，这是一个十分经典的算法，据我所知，在 PostgreSQL 中，也有类似的管理方式。

在 InnoDB tablespace 中，每隔 page_size 个页面，就是一个 Ibuf Bitmap page。例如：若 page_size = 16384(16k)，那么 page_no 为 0，16384，32768，…的 page，就是 Ibuf Bitmap page，Bitmap page 的功能，就是管理其后连续的 page_size – 1 个 page 的空间使用率。每个 page，在 Bitmap page 中占用一项(小于 1 byte)。如下图所示：



图表 2-1 Bitmap Page 示意图

图 2.1 中，page_size = 5。page 0，5 为 Bitmap Page，其中的 record 记录了紧随其后的 4 个 page 的剩余空闲空间。例如，Bitmap Page 1 中的 record 0 记录了 page_no = 1 的 page 的剩余空闲空间。而如果想要知道 Page No 8 的剩余空闲空间，定位到 Bitmap Page 1 中的 record 2 即可。

Bitmap Page 中的每一个 record，占用 4 bits，以这 4 bits 来标识其对应的 page 的剩余空闲空间。那么这 4 bits 是如何转换为剩余空闲空间的呢？

4 bits 根据功能，又可以拆分为两份，分别为 2 bits

[0, 1] bits：对应页面的剩余空闲空间

[2, 3] bits：对应页面的 insert buffer 优化占用多少空间

函数 ibuf0ibuf.ic::ibuf_index_page_calc_free_from_bits 给出了剩余空闲空间的计算方式：

```
ut_ad(bits < 4);
if (bits == 3)
    return(4 * UNIV_PAGE_SIZE / IBUF_PAGE_SIZE_PER_FREE_SPACE);
return(bits * (UNIV_PAGE_SIZE / IBUF_PAGE_SIZE_PER_FREE_SPACE));
#define IBUF_PAGE_SIZE_PER_FREE_SPACE  32
```

首先，bits 必须小于 4 (2 bits)；其次，剩余空闲空间必须大于页面大小的 1/32，才能进行 Insert Buffer 优化；最后，剩余空间越小，bits 越精确，当剩余空间大于页面大小的 3/32 时，就已经不能通过 bits 准确计算剩余空间的大小了。

2.1.3 Ibuf Root Page

在一个 InnoDB 系统中，insert buffer 的内存占用是比较大的，最大可以达到 buffer pool 的 1/2。

为了保证 insert buffer 的恢复能在较短时间内完成，insert buffer pages 也会由 dirty page flush 操作写回 disk。系统崩溃时，就能够保证 insert buffer table (SYS_IBUF_TABLE)能够较快恢复。Insert buffer 对应的 page，都在 system tablespace (tablespace 0)中分配，并且 Insert buffer 聚簇表对应的 root page 是恒定的，是 system tablespace 中的第 5 个页面。定义如下：

```
#define FSP_IBUF_TREE_ROOT_PAGE_NO  4
    /*!< insert buffer B-tree root page in tablespace 0 */
```

tablespace 0 的第 5 个 page，就是 Insert buffer 聚簇表的根页面。

在系统崩溃恢复完成之后，重建 SYS_IBUF_TABLE，重建对应的聚簇索引 CLUST_ID。然后将聚簇索引的 root page 设置为[0, 4]即可(space_id = 0，page_no = 4)。

此处为何是重建 Ibuf table？我的理解，因为 Ibuf table 的表定义是不变的，并且表的 root page 也是不变的。因此不需要持久化 Ibuf table 数据字典信息，直接重建最省。

说到这儿，那么表空间的前几个页面，是否也是有特殊用途的呢？答案是肯定的，可以在 Fsp0types.h 文件中找到表空间前几个页面的特殊用途，提取如下所示：

```
/*--------------------------------------*/
#define FSP_XDES_OFFSET             0   /* !< extent descriptor */
#define FSP_IBUF_BITMAP_OFFSET      1   /* !< insert buffer bitmap */
                /* 此页面开始，每隔XDES_DESCRIBED_PER_PAGE个页面，就是一个Bitmap Page*/
#define FSP_FIRST_INODE_PAGE_NO     2   /*!< in every tablespace */
#define FSP_IBUF_HEADER_PAGE_NO     3   /*!< ibuf header page, in tablespace 0, 用于管
理Ibuf中的 page 的分配与释放 */
#define FSP_IBUF_TREE_ROOT_PAGE_NO  4   /*!< ibuf B-tree root page in tablespace 0 */
#define FSP_TRX_SYS_PAGE_NO     5   /*!< transaction system header in tablespace 0 */
#define  FSP_FIRST_RSEG_PAGE_NO     6   /*!< first rb segment  page, in tablespace 0 */
#define FSP_DICT_HDR_PAGE_NO        7   /*!< data dict header  page, in tablespace 0 */
/*--------------------------------------*/
```

2.1.4 Insert Buffer for Purge

在前面的章节中，主要针对的是 buffer insert 操作。在 5.5 之后，Insert Buffer 不仅能够 buffer insert 操作，并且能够 buffer delete mark/purge 等操作。

提到 delete mark 操作，不得不简单说一下 InnoDB 的多版本。为了实现多版本，InnoDB 的索引在进行 delete 操作时，并不是直接将记录从索引中删除，而是仅仅将记录标识为 delete 状态(delete mark)，每条记录上，都有一个 delete bit。

记录何时被真正删除，要等到 InnoDB 的 purge 线程，根据 redo log，回收索引上被标识为 delete bit 的项。

从以上的简单描述可以看出，delete mark 操作并不会删除记录，因此也不会对索引页面的利用率产生影响。但是 purge 操作却是真正的删除数据，会减少索引页面的利用率，甚至将页面删空。空页面会导致索引进行 SMO 操作，而 Insert Buffer 是不支持 SMO 的，因此，必须能够监控这种情况，保证 purge 操作的 buffer 不至于删空整个索引页面，InnoDB 如何实现这个监控？

代码流程：

```
ibuf0ibuf.cc::ibuf_insert_low
    // purge 操作在 insert buffer 中被映射为 IBUF_OP_DELETE 操作
    // delete 操作在 insert buffer 中被映射为 IBUF_OP_DELETE_MARK 操作
```

```
    if (op == IBUF_OP_DELETE &&
      (min_n_recs < 2
      || buf_pool_watch_occured(space, page_no)))
        // 若 min_n_recs < 2，则不能进行 purge 的 buffer 操作
        // 因为页面有可能因为本次 purge 而被删空，产生 SMO
```

那么 InnoDB 是如何计算 min_n_recs 的呢？此时则又需要转到我们前面提到过的 ibuf_get_volume_buffered 函数。

ibuf_get_volume_buffered 函数的第一个功能，前面已经提到，统计 insert buffer 中对于同一 page，buffer 了多少空间。其实该函数还有第二个功能，如下：

```
ibuf0ibuf.cc::ibuf_get_volume_buffered
    ibuf_get_volume_buffered_count
        case IBUF_OP_INSERT:
        case IBUF_OP_DELETE_MARK:
            if (ibuf_get_volume_buffered_hash(...))
                (*n_recs)++;
```

min_n_recs 的取值，并不是页面中真实剩余记录的数量，而是页面进入 Insert Buffer 的 Insert/Delete_Mard 操作的数量。Insert 操作，由于有可能不会增加记录数量，因此此处不考虑；而 Delete_Mark 的 buffer，并不会减少记录，一个 Delete_Mark 操作，一定对应于 page 中的一项，因此可以将 recs++。

但是此处又有一个例外：是否每一个 Delete_Mark 操作，都对应于一条不同的记录？这个是不能保证的，一条记录的多次 Insert/Delete_Mark 操作，最终对应的仍旧是一条记录。因此，需要区分不同的 Delete_Mark，操作的记录是否相同，通过函数 ibuf_get_volume_buffered_hash 实现。

ibuf_get_volume_buffered_hash 函数，简单说来，就是将每次看到的 Delete_Mark 操作对应的 data 映射到一个 unsigned long int 值，然后将此值映射到 128 位中的一位，判断此位在已有的 hash 中是否已经设置，若设置，证明记录前面已经存在，此次 count 不能增加；若未设置，证明是不同的记录，增加 count，并更新 hash 取值。

目前，InnoDB 针对 purge 操作的 buffer 有 bug，具体可见网文：InnoDB: Failing assertion: page_get_n_recs(page) > 1。此 bug，目前尚未有 patch 发布，因此在使用 MySQL 5.5 及以上版本时，慎用 Insert Buffer 的新功能。

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。

2.3 Insert Buffer 疑问解析

- Insert Buffer 是否在 crash recovery 时完全导入？

  否。因此第一次读取/写入 Insert Buffer Page 时，可能需要 IO
- Insert Buffer Page 是否进入 buffer pool 的 flush list？

  是。mtr commit 时，会将 Insert Buffer Page 链入 flush list。Insert Buffer Dirty Page 如果不进入 flush list，就无法推进 Checkpoint_LSN.
- Insert Buffer Page 是否进入 buffer pool 的 LRU list？

  猜测：Insert Buffer Page，是不进入 buffer pool 的 LRU list 的，否则就会被 lru 替换策略刷出内存，这样就失去了 Insert Buffer 的意义，待验证。

  经过验证，Insert Buffer Page 会进入 Tablespace 0 的 buffer pool 的 LRU list，Debug 模式下，可见：

  block->page->in_LRU_list = TRUE;

  既然进入 LRU list，就标志着 Insert Buffer Page 也可能被替换到外存。

  猜测原因在于：

  1. Insert Buffer Page 相对较热，LRU 中被替换到外存的概率较小
  2. Insert Buffer Page 如果真的非常冷，那么替换到外存，也无可厚非
  3. 颠覆了我对于 Insert Buffer 的认识
-

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
  - 每 10S，必定调用一次 log_checkpoint，做一次 Checkpoint

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：

buf0flu.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 1S 会在 srv_error_monitor_thread 线程中，调用 buf_flush_stat_update()函数更新。buf_flush_stat_arr[BUF_FLUSH_STAT_N_INTERVAL]数组，保存过去 20S，每秒对应的系统日志总量以及系统 LRU List Flush 写出的脏页总量。每 S 产生的日志量与 LRU List Flush 刷出的脏页数量，将数组内连续的两项相减即可。

buf_flush_stat_arr 数组采用 round-robin 策略维护，每 1S，将数组中最老的项删除，然后插入当前 1S 的记录。

于此同时，InnoDB 中还维护着一个 buf_flush_stat_cur 变量，用于维护上一秒的信息；一个 buf_lru_flush_page_count 变量，用于实时维护系统中的 LRU List Flush 的脏页总量。将 buf_lru_flush_page_count 与 buf_flush_stat_cur.n_flushed 相减，就得到了当前秒内 LRU List Flush 的脏页总量。

```
buf0flu.c::buf_flush_stat_update();
```

3. 计算过去一段时间内，LRU List Flush 的平均速度；LRU List Flush，是将 LRU 链表尾部的脏页写出，释放出足够的空闲页面，供内存页面替换之用。

```
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);
```

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. 根据当前系统页面脏页数量、Redo 日志产生的速度，以及 LRU List Flush 的平均速度，计算本次 Flush List Flush 需要写出的脏页数量。

```
n_flush_req = (n_dirty * redo_avg) / log_capacity;
rate = n_flush_req – lru_flush_avg;
```

n_flush_req：
计算当前 1S 内应该写出的脏页数量；此公式的意义为：当前一共有 n_dirty 个脏页，并且日志产生的速度为 redo_avg，日志总容量为 log_capacity，为了保证日志在回卷前，写完所有的脏页，当前必须写的脏页数量为(n_dirty * redo_avg) / log_capacity

rate：
n_flush_req 为计算出来的当前 1S 需要写出的脏页总数量，但是考虑到写脏页有两个方式：LRU List Flush 与 Flush List Flush，因此当前 Flush List 实际需要写的脏页数量，还需要减去 LRU List Flush 1S 内的平均写脏页数量。

5. Flush List Flush 实际写的 dirty pages 数量，最大是 PCT_IO(100)，200 个 dirty pages。

3.3.3 统计信息维护

3.3.3.1buf_flush_stat_update

维护系统中与日志产生量、LRU List Flush 写脏页数量有关的统计信息，用于 InnoDB 的 Write I/O 流控。

redo：
系统产生的日志总量；

n_flushed：
系统 LRU List Flush 写出的脏页总量；

3.3.3.2buf_lru_stat_update

These statistics are not *'of' LRU* but **'for' LRU**. We keep count of I/O and page_zip_decompress() operations. Based on the statistics we decide if we want to evict from buf_pool->unzip_LRU or buf_pool->LRU.
buf_lru_stat 统计信息，不是 LRU 链表的统计信息，而是为 LRU 替换维护的统计信息。此统计信息结构，包含两个参数：io 与 unzip。io 对应的是 InnoDB 非压缩页面所有的 I/O 操作次数，包括 Write、Read、Read Ahead 等；unzip 对应的是压缩页面的解压次数。根据这两个统计信息，InnoDB 可以判断选择 LRU 链表或者是 unzip_LRU 链表进行页面替换操作。

```
struct buf_LRU_stat_struct
{
    ulint io;       /**< Counter of buffer pool I/O operations. */
                    /**  包括系统非压缩页面所有的 I/O 操作：Read、Write*/
    ulint unzip;    /**< Counter of page_zip_decompress operations. */
                    /**  包括系统压缩页面所有 Decompress 操作*/
};
```

3.3.4 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;
```

选择当前 Dirty Page 的 Neighbors，算法较为简单，就是计算当前 Dirty Page 所属 Extent，属于同一 Extent 的所有 Pages(最大 64 个 Pages)，均为 Neighbor Pages。如此一来，计算出来的 low，high 与当前 Dirty Page offset 的关系为：

low <= offset <= high; range[low, high] = BUF_READ_AHEAD_AREA

      ii. 所有的 pages，在 buffer pool 中同时以 hash 表存储(**Page Hash：用于快速定位页面是否在内存中存在**)，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
      iii. buf_flush_ready_for_flush()函数，判断当前 Dirty Pages 能否被写出(需要持有 buffer pool mutex 与 block mutex)：

```
buf0flu.c::buf_flush_ready_for_flush();
    // 1. 必须保证页面是脏页，oldest_modification 不为 0
    // 2. 必须保证当前页面不是正在进行 io 操作：io_fix = 0
    if ((block->oldest_modification > ul_dulint_zero) && (block->io_fix == 0))
        // 3. 根据写脏页的类型区别对待：
        // 3.1 若为 Flush List Flush，不会将页面从内存中换出，因此无需
        //     考虑页面当前是否正在被 Pin 住；
        // 3.2 若为 LRU List Flush，会将页面替换出内存，因此 Pin 住的页面
        //     是不能够写出去的；(在 InnoDB 中，Pin 住也意味着被 Latch)
        //     尤其需要注意：InnoDB 中，Pin 住就意味着被 Latch 住
        if (flush_type != BUF_FLUSH_LRU)
            return TRUE;
        else if (block->buf_fix_count == 0)
            return TRUE;
```

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(buf_dblwr_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);
```

vi. buf_flush_buffered_writes 函数，在 MySQL 后续版本中，被 buf_dblwr_flush_buffered_writes 函数替换。

功能上，二者有相似之处，包括同步写 double write；但是有更大的不同之处，由于新的 MySQL 版本支持了真正意义上的 Native AIO，因此采用 Native AIO 写出到数据文件。数据文件读写 AIO，由后台线程处理，write thread 会循环获取所有已完成的 AIO 请求，并作相应的清理工作。(**io_handler_thread**)

```
buf_dblwr_flush_buffered_writes();
    ...
    buf_dblwr_write_block_to_datafile();
        // 分析 IO 的类型以及 IO 的模式，此处为 Write AIO
        fil0fil.cc::fil_io(OS_FILE_WRITE, sync(FALSE), ...);
            os0file.cc::os_aio_func(type(WRITE), mode(OS_AIO_NORMAL));
                // 在 Write AIO Array 的指定 segment 获取一个空闲的 slot
                os_aio_array_reserve_slot();
```

// 提交一个 AIO 处理请求，具体 I/O 何时处理，
// 由文件系统控制。后台线程(**io_handler_thread**)会循环
// 遍历 AIO Array，获取已完成 I/O 操作的页面，做最后的
// 清理工作：释放 slot；将 page 从 flush list 移除；必要时
// flush 所有的数据文件，保证记录完全写出磁盘；
os0file.cc::os_aio_linux_dispatch();
io_submit();

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)

3.3.5 Checkpoint 算法分析

#InnoDB# I/O 流控：InnoDB 中有两个写脏页的操作，分别为 LRU List Flush 与 Flush List Flush。在 MySQL 5.5 中，Flush List Flush 由前端用户线程发起，Flush List Flush 由后台线程进行。二者存在并发。为了实现较为精确的 Write I/O 流控，InnoDB 在进行 Flush List Flush，计算本次写出脏页的数量 n_flush 时，不仅仅需要考虑内存中脏页的数量与 Redo 日志的产生速度，还需要考虑 LRU List Flush 究竟写出了多少脏页 n_flushed，只有将 n_flush 与 n_flushed 相减，才能获得 Flush List Flush 需要写出的脏页数量(1S 内)。

算法存在的问题：

- LRU List Flush 与 Flush List Flush 并发进行，增加了 I/O 流控的复杂度；
- LRU List Flush 与 Flush List Flush 相互干扰，异步 I/O 的合并效率也会降低；

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(可用日志空间的定义，是单个日志文件大小 * 日志文件个数)的 9/10；
日志空间消耗超过 7/8 时，一定要进行异步 Flush Dirty Pages；
日志空间消耗超过 15/16 时，一定要进行同步 Flush Dirty Pages；
日志空间消耗超过 31/32 时，一定要进行异步更新 Checkpoint 信息；
日志空间消耗达到 margin 上限时，一定要进行同步更新 Checkpoint 信息；
以上判断均在 log_checkpoint_margin 函数中完成，1S 中判断一次。
或者是在用户进行 U/I/D 操作时，也需要进行判断。

例如，对于单个日志文件大小为 5M，每个日志组有两个日志文件的情况下：

```
margin                          = 7310501 ~ 7M
log->max_modified_age_async     = 6396689 ~ 6.1M
log->max_modified_age_sync      = 6853595 ~ 6.5M
log->max_checkpoint_age_async   = 7082048 ~ 6.7M
log->max_checkpoint_age         = 7310501 ~ 7.0M
```

注意，日志文件中，需要预留的量是一定的。因此，日志文件越大，文件越多，预留量的比率越小，从而这几个参数占可用日志空间的比率也就越高。

- 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);
Innodb 的异步 I/O，默认情况下使用 linux 原生 aio，libaio。关于异步 I/O 的优势，可参考网文[18][19][43]；libaio 的限制，可见网文[17]。下面详细分析 Innodb 异步 I/O 的处理步骤。

4.1 AIO 处理流程

本小节，以记录插入操作，跟踪 AIO 的处理流程，包括：AIO 使用的初始化；读 AIO；写 AIO；后台线程处理流程等。

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 = 256

```
#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(将 aio 请求进行分发)

os0file.cc::os_aio_func();

os_aio_array_reserve_slot();

os0file.cc::os_aio_linux_dispatch();

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 是提交 IO 任务时用到的，可以完整地描述一个 IO 请求，在 io_submit 调用前，需要通过 io_prep_pread or io_prep_pwrite 填充；

d) io_context 结构，相同 segment 共用一个，标识一个 AIO 能够处理的 I/O 数量，为 segment 中的 slots 数量；因此，io_submit 时，将 slot 中的 AIO 请求，提交到 segment 对应的 io_context 处理；

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

若 mode = OS_AIO_SYNC，采用异步 I/O，则在前面提到的 os_aio_func 处理结束之后，前台用户线程就完成了 I/O 操作，具体 I/O 操作何时进行，完全交由后台的 I/O 线程处理，I/O 处理线程的，主要的功能，在 os_aio_windows/linux_handle 中进行。

os_aio_windows_handle

4. windows 处理异步 I/O 的流程：

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：

```
// arg 为指定等待的 AIO Write Array 中的一个 Segment，一个 Array，会有多个
// write thread 负责；每个 thread 对应一个 segment
srv0start.cc::innobase_start_or_create_for_mysql()
    // 开始 n 个 I/O 处理线程，为每一个线程指定一个 segment n[i] = i
    os_thread_create(io_handler_thread, n+ i, thread_ids + i);
    DECLARE_THREAD(io_handler_thread)(void *arg)
        // 一直循环处理异步 I/O，直至 InnoDB 系统关闭
        // 当然，这个循环不会一直占用 CPU 资源，会进入等待
        // 在系统 I/O 不繁忙时，线程大部分时间，处于等待状态
        // 等待在 io_getevents 函数调用上，见下面的分析
        while(fil0fil.cc::fil_aio_wait());
            os0file.cc::os_aio_linux_handle();
                os_aio_linux_collect();
                    // 一个与 io_context 处理 AIO 数量一致的 io_event 数组
                    // 用于接受 AIO 操作的返回值；
                    events = &array->aio_events[segment * seg_size];
                    // segment 对应的 AIO 处理 io_context
                    io_ctx = array->aio_ctx[segment];
                    // 进入等待，收集完成的 AIO 操作，
                    // AIO 操作的返回值被存储在 events 数组中
                    // ret 说明了完成 events 数组填充的数量
                    ret = io_getevents(io_ctx, 1, seg_size, events, &timeout);
```

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。若 AIO Array 中有多个同时完成的异步 I/O，此函数也是一次处理一个完成 I/O 操作的 Slot，将 Slot 设置为 Free。然后返回上层的 fil_aio_wait 函数，由此函数完成当前 Slot 对应的 I/O 的后期处理(fil0fil.cc::fil_node_completer_io())

AIO 完成之后，后期处理流程：

```
fil0fil.cc::fil_aio_wait();
    os_aio_linux_handle();
    // 处理 OS_FILE_WRITE 操作：
    // 根据是否开启了文件系统缓存，write 操作不一定被持久化到磁盘之中
    fil0fil.cc::fil_node_complete_io();
        // 判断是否禁用文件系统缓存，判断条件：
        // flush_method = SRV_UNIX_O_DIRECT_NO_FSYNC
        if (fil_buffering_disabled(node->space))
            ...
        // 若开启了文件系统缓存，则当前页面的写操作不一定 flush 到磁盘
        // 设置当前 Tablespace 对应的文件需要 flush，并且加入未 flush 链表
```

```
                // 所有链表中的未 flush Tablespace，后续均会调用 flush 进行同步
                else
                    node->space->is_in_unflushed_spaces = true;
                    LIST_ADD(system->unflushed_spaces, node->space);
            // 处理数据文件读写完成之后的后续处理工作
            if (fil_node->space->purpose == FIL_TABLESPACE)
                buf0buf.cc::buf_page_io_complete();
                    // 读操作后续处理：判断页面是否 currupted，计算 checksum
                    if (io_type == BUF_IO_READ)
                        …
                    // 设置当前的 I/O 操作真正完成，此时仍旧持有 page latch；
                    buf_page_set_io_fix(bpage, BUF_IO_NONE);
                    if (io_type == BUF_IO_WRITE)
                        buf0flu.cc::buf_flush_write_complete();
                            // 将页面从 flush list 中移除，当前 page 变为 clean page
                            buf_flush_remove();
                            …
                            // 标识当前 LRU Flush 或者 Flush List Flush 结束
                            os_event_set(buf_pool->no_flush[flush_type]);
                            …

                            buf0dblwr.cc::buf_dblwr_update();
                                buf_dblwr->b_reserved--;
                                // 若 double write 中的所有页面，均已经被写出
                                // 此时可以做一次所有文件的 flush，然后释放 dbl
                                // 全局 flush：通过遍历前面提到的
                                // system->unflushed_spaces 链表获得
                                if (buf_dblwr->b_reserved == 0)
                                    fil_flush_file_spaces(FIL_TABLESPACE);
                                        fil0fil.cc::fil_flush();
            else
                log_io_complete();
```

待当前 Slot 处理完成之后，继续遍历 Aio Array，获取下一个完成的 Slot，直至所有完成的 Slot 全部处理完毕，此时，调用 os_aio_linux_collect 函数，收集下一批完成的 I/O 请求。

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

主要是定位到返回的 control 结构所属的 slot，然后将 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。

6. 在 Linux Native Aio 操作中
   a) os_aio_linux_dispatch 用于用户 AIO 请求的分配与异步处理；
   b) os_aio_linux_handle 等待一个指定的 AIO 线程处理完成；
   c) os_aio_linux_collect 用户在 os_aio_linux_handle 函数没有捕获更多的完成 AIO 时，进行 AIO 请求的等待；

4.2 InnoDB 何时需要进行 File Flush？

在前面提到的 Native AIO 处理流程中，io thread 的功能，只是等待 AIO 完成，进行相应的内存处理，但是并不保证 AIO 对应的 Page 已经被真正持久化到磁盘之中(可能正位于文件系统缓存中)。

所有这些完成 AIO 操作的页面对应的文件，都被存储在内存 system->unflushed_spaces 链表之中。那么，什么情况下，必须 Flush 这些链表中的文件，保证写入必须持久化到磁盘呢？

4.2.1 情况 1：一次性批量的 double write buffer 写出

Flush List Flush，或者是 LRU List Flush，当一次性收集到 double write buffer size 的 pages 数量，或者是预写脏页收集完毕。此时会调用 `buf_dblwr_flush_buffered_writes` 函数，此时会将 buf_dblwr->batch_running 设置为 true，此时，后续的脏页写出必须等待当前这一批 double write 的写完成之后，才能进行。

同步写出 double write 后，将所有的脏页异步的方式写出到磁盘。异步写出所有的脏页之后，Flush List Flush 与 LRU List Flush 线程退出，但是此时 batch write 并未完成，buf_dblwr->batch_running 参数仍旧为 true，新的写操作仍旧不能进行。那么 buf_dblwr->batch_running 参数何时被设置为 false 呢？

此时需要跳转到 io_thread，io_thread 在 fil_io_wait -> buf_page_io_complete -> buf_flush_write_complete -> buf_dblwr_update 函数中，在本批次的 AIO 操作全部完成，最后一个 IO 也写出之后(buf_dblwr->b_reserved == 0)，调用 fil_flush_file_spaces 函数，将所有写所涉及到的文件，flush 到磁盘。此时，可以将 batch_running 设置为 false，接受新的 write aio 请求。

4.2.2 情况 2：InnoDB 做 Checkpoint 之前

InnoDB 的 Checkpoint LSN，取得是 Flush List 中最老更新页面的 oldest_modification，此时要

必须保证 AIO 方式写出的脏页真正写到了磁盘之中。因此，Checkpoint 的第一步，就是将 buffered write flush 到磁盘。

log0log.cc::log_checkpoint();

```
if (srv_unix_file_flush_method != SRV_UNIX_NOSYNC)
        fil_flush_file_spaces();
```

4.2.3 情况 3：Single Page Flush

LRU Flush 的一个特殊情况，用户在 LRU 链表尾部找不到可替换的 Page 时，会尝试 Flush 一个 Dirty Page，并进行替换。此时的操作也是需要 File Flush：

Writes a page to the doublewrite buffer on disk, sync it, then write the page to the datafile and sync the datafile. This function is used for single page flushes.

buf0dblwr.cc::buf_dblwr_write_single_page();

4.3 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。

4.4 InnoDB AIO 的细节

- InnoDB 进行异步 IO 的 Pages，需要持有 Latch 吗？

  答案是肯定的，进行异步 IO 的 Pages，需要持有 S Latch。

  buf0flu.cc::buf_flush_try_neighbors() -> buf0flu.cc::buf_flush_page();

  ```
  // 持有 buf pool mutex 与 buffer header mutex
  ut_ad(buf_pool_mutex_own(buf_pool));
  ut_ad(mutex_own(block_mutex));
  // 设置当前的 IO 状态：BUF_IO_WRITE
  buf_page_set_io_fix(bpage, BUF_IO_WRITE);
  …
  // 获取页面上的 S Latch (如何避免死锁？)
  rw_lock_s_lock_gen(&((buf_block_t*)bpage)->lock, BUF_IO_WRITE);
  ```

```
mutex_exit(block_mutex);
buf_pool_mutex_exit(buf_pool);
…
// 持有页面 S Latch 的情况下，发起页面的异步写 IO 操作 (Latch 何时释放？)
buf0flu.cc::buf_flush_write_block_low();
…
```

- InnoDB 进行异步 IO 的 Pages，如果需要持有 Latch，是否会产生 Latch 死锁？

答：不会产生 Latch 死锁

```
buf_flush_page();
    // 判断页面是否有其他并发使用：buf_fix_count == 0，说明没有并发使用
    // 若存在并发使用，则强制加 S Latch，可能会存在 Latch 死锁的风险
    is_s_latched = (bpage->buf_fix_count == 0);
    if (is_s_latched)
        rw_lock_s_lock_gen();
    if(!is_s_latched)
        // 若页面存在并发使用，强制加 S Latch 存在死锁风险
        // 则先将异步 IO 队列中已有的 Dirty Pages 写出到磁盘
        // 然后在不持有任何 Page S Latch 的前提下，强制加当前 Dirty Page
        // 的 S Latch，发出异步 IO 请求，开始新一次的异步 IO 收集
        buf_dblwr_flush_buffered_writes();
        rw_lock_s_lock_gen();
    buf_flush_write_block_low();
        …
```

- InnoDB 进行异步 IO 的 Pages，如果需要持有 Latch，那么 Latch 何时释放？

答：Latch 由后台的 Write IO 线程负责收集完成的 Write IO，然后释放。除了释放 Latch 之外，还需要做很多的清理工作。

```
srv0start.cc::DECLARE_THREAD(io_handler_thread);
    while(fil0fil.cc::fil_aio_wait());
        // 一次收集一个完成的异步 IO 请求
        ret = os_aio_linux_handle();
            // 设置当前完成的 Page 对应的文件信息
            fil_node_complete_io();
        buf0buf.cc::buf_page_io_complete();
            …
            buf_pool_mutex_enter(buf_pool);
            mutex_enter(buf_page_get_mutex(bpage));
            // 设置当前页面的异步 IO 操作结束
            buf_page_set_io_fix(bpage, BUF_IO_NONE);
            …
            buf0flu.cc::buf_flush_write_complete();
                // 从 Flush List 脏页链表中移除，并且将页面的
                // 脏页标识 oldest_modification 设置为 0
                buf_flush_remove();
                // 判断一个批量的异步 IO 是否结束，
```

```
                    // 若结束，则可开始新一批次的异步 IO
                    buf0dblwr.cc::buf_dblwr_update();
                        if (buf_dblwr->b_reserved == 0)
                            ...
                // 此处，释放页面上的 S Latch
                // 至此，一次异步 IO 操作结束
                rw_lock_s_unlock_gen(&((buf_block_t*)bpage)->lock, BUF_IO_WRITE);
```

➢

4.5 InnoDB Simulated AIO

在 Linux 系统上，InnoDB 除了可以使用 Linux 自带的 libaio 之外，其内部还实现了一种称之为 Simulated aio 功能，用以模拟系统 AIO 实现(其实，Simulated aio 要早于 linux native aio 使用在 innodb 中，可参考网文[16])。前面的章节，已经分析了 Innodb 对于 Linux 原生 aio 的使用，此处，再简单分析一下 Innodb simulated aio 的实现方式。

以下一段话摘自 Transactions on InnoDB 网站[16]，简单说明了 simulated aio 在 innodb 中的处理方式。

… The query thread simply queues the request in an array and then returns to the normal working. One of the IO helper thread, which is a background thread, then takes the request from the queue and issues a synchronous IO call (pread/pwrite) meaning it blocks on the IO call. Once it returns from the pread/pwrite call, this helper thread then calls the IO completion routine on the block in question …

queue I/O request

```
os0file.c::os_aio_~~func~~();
    // AIO，有 AIO 数组，根据 IO 的类型，分配不同的 AIO 数组
    if (mode == OS_AIO_NORMAL)
        if (type == OS_FILE_READ)     array = os_aio_read_array;
        if (type == OS_FILE_WRITE)    array = os_aio_write_array;
    // 从对应的 AIO 数组中，分配一个 Slot，存储当前 IO 页面
    // 顺序遍历 AIO 数组，分配一个可用的 AIO Slot，Slot 中存储：
    // 文件名；需要写出的 Buf；长度；类型；偏移等等
    os_aio_array_reserve_slot();
    if (array->n_reserved == array_n_slots && !srv_use_native_aio)
        os_aio_simuated_wake_handler_threads();
    os_event_wait(array->not_full);
    slot->… = …;
```

对于用户发起的 read/write aio，simulated aio 仍旧从对应的 array 中寻找空余 slot，然后将 aio request 丢进此空闲 slot，然后返回即可。

do I/O request by backend I/O thread

fil0fil.c::fil_aio_wait -> os0file.c::os_aio_simulated_handle

- windows 下处理异步 I/O 函数为 os_aio_windows_handle；linux 原生 aio 下为 os_aio_linux_handle；而 linux simulated aio 下，对应的函数则是 os_aio_simulated_handle
  - 步骤一，遍历 array，找出请求时间超过 2S 的 I/O，优先处理，防止饥饿。将此 Slot 作为当前这次 I/O 操作的第一个 I/O；

- 步骤二，若步骤一没有定位到，则找出 I/O request 中，page_no 最小的请求。并将此 Slot 作为当前这次 I/O 操作的第一个 I/O；
- 步骤三，再次遍历 array，寻找以步骤一或者步骤二的 Page 开始的连续 pages 的 I/O 请求，连续 Pages 可以一次性写出或者读入(一次处理的连续 Pages 数量有上限，上限是 OS_AIO_MERGE_N_CONSECUTIVE，64 个连续 Pages 作为一组)
- 将步骤三所收集到的连续 Pages，调用底层的读/写操作，一个 I/O 解决；

os_file_read/write -> os_file_read/write_func -> os_file_pread/pwrite -> pread/pwrite

- 调用 pread/pwrite 进行同步读/写
- 在读写结束之后，做些后续处理，标识对应的 slot，I/O 操作完成

总结：

Linux simulated aio，实现简单，基本采用的仍旧是同步 IO 的方式。相对于 Linux native aio，simulated aio 最大的问题在于：每个 I/O 请求，最终都会调用一次 pread/pwrite 进行处理(除非可以进行相邻 page 的合并)，而 Linux native aio，对于一个 array，进行一次异步 I/O 处理即可。

4.6 AIO 层次分析

- InnoDB 层面 (IO 请求)
  SRV_N_PENDING_IOS_PER_THREAD
  一个 InnoDB thread 同时能够处理的异步 I/O 的数量
- File System 层面 (AIO queue)
  fs.aio-max-nr
  文件系统级别，可以同时进行的 aio 的数量上限
- Disk 层面 (IO queue)
  nr_request
  单个 disk，可以并发处理的 I/O 请求的个数
  若当前磁盘为逻辑盘，那么真实的 nr_request = nr_request * disks in raid

注：

彭立勋 (13:42:00):

FusionIO 处理不是普通盘那样，普通磁盘消耗 IO 队列是匀速的，FIO 是抖动式的

何登成 (13:42:50):

难道说要将 aio 队列调小，消除 FIO 的抖动？

彭立勋 (13:43:40):

例如 FusionIO 的处理能力是每 10ms 100 个 IO，IO 队列里有 200 个 IO，FusionIO 会一次拿走 100，10ms 内不再从 IO 队列里拿东西，然后再拿走 100 个

SAS 盘是匀速的，每处理完一个就拿一个

何登成 (13:44:34):

那也没法解释为什么是 10S 抖动一次啊？

彭立勋 (13:44:39):

然后 nr_request 有个算法，如果 IO 队列达到 2/3 时，就认为系统阻塞了，就没响应了

彭立勋 (13:45:49):

所以如果 AIO 队列太长，一次写到磁盘 IO 队列上的东西太多，碰上 FusionIO，就容易出现达到 2/3，虽然很快就消失了，但是还是能看到突然阻塞这种情况

霸爷建议，观察 nr_request 用了多少，AIO 队列长度保持磁盘 IO 队列的 2/3 以内

何登成 (13:47:16):

哦，有点理解了

就是要 aio 的队列最大长度，不超过 nr_request 的 2/3

彭立勋 (13:47:22):

aio_nr 是 FS 级的，nr_request 是 Disk 级的

是的

霸爷就是这意思，只对 FusionIO 有效，SAS 盘不是这样的

SAS 盘尽管往里面堆

4.7 Linux AIO 增强

InnoDB 使用的 linux native aio，仅仅支持以 O_DIRECT 方式打开的文件。针对此情况，大神 Jens Axboe 做了进一步的优化，提供了 Buffered async IO，具体情况可参考其个人主页上的一篇文章：Buffered async IO。

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 的日志量，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));
```

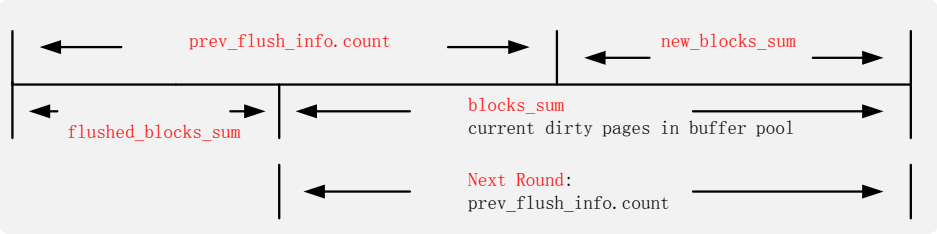
5.1.3 keep_average

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 MySQL 5.6.6 Flushing 优化

InnoDB 自带的 flushing 策略有问题，Percona 的 XtraDB 对其做了优化，提出了 estimate 与 keep_average 两种新的 flushing 策略。在 MySQL 5.6.6-labs 版本中，Oracle InnoDB 团队提出了自己的新的 flushing 算法[43]，不同于已有的 Percona 算法，以下对此新算法作简要分析。关于新 flushing 算法的测试结果，可见[41][42]。
首先，在 MySQL 5.6-labs 版本中，dirty pages 的 flushing 操作，从 InnoDB 主线程中移出，新开一个 buf_flush_page_cleaner_thread (page_cleaner)线程来进行。仍旧是每次休眠 1s，然后进行一次 flushing 尝试。

6.1 处理流程

新的 flushing 算法的处理流程如下：

```
buf0flu.cc::buf_flush_page_cleaner_thread();
    // 回收 buffer pool LRU 链表，此处不考虑
    page_cleaner_flush_LRU_tail();
    // 根据用户指定的参数，判断是否需要进行 dirty pages 的 flushing
    // 每 1S 调用一次此函数
    page_cleaner_flush_pages_if_needed();
        // 根据用户指定的 innodb_flushing_avg_loops 参数(默认取值为 30)，
        // 计算前 innodb_flushing_avg_loops 个 flushing 的平均速度
        // 1. avg_page_rate:      平均的 dirty pages 的 flushing 速度
        // 2. lsn_avg_rate:       平均的日志产生速度
        if (++n_iterations >= srv_flushing_avg_loops)
            …
        // 计算目前系统中的最老未 flush 的 dirty page 的日志年龄
        age = cur_lsn – oldest_lsn;
        // 根据系统中的脏页数量，计算本次需要 flushing 的页面数量：
        // 1. 计算系统中脏页面占用的比率[0, 100]
        // 2. 若用户未指定脏页的 lower water mark – innodb_max_dirty_pages_pct_lwm
        //     [0, 99]默认值为 50. 则采用原有的 srv_max_buf_pool_modified_pct 参数判断
        //     超过此参数则返回 100.
        // 3. 若用户指定了 lower water mark，并且脏页比率超过此值，则返回
        //     dirty_pct * 100 / (srv_max_buf_pool_modified_pct + 1)；否则返回 0
        //
        // 注意：
        // 根据 innodb_max_dirty_pages_pct_lwm 与 innodb_max_buf_pool_modified_ptc
        // 两参数的设置情况，最终的返回值主要与后者相关，后者设置的越小，计算出
        // 的 flushing 量越大，甚至会超过 100 (超过 innodb_io_capacity)。但是，由于后者
```

```
// 默认设置为 75，因此在默认设置下，返回值的范围是[0, 132]，一次 flush 的量，
// 并不会比 innodb_io_capacity 多多少。
pct_for_dirty = af_get_pct_for_dirty();
// 根据系统中最老日志的年龄，计算本次需要 flushing 的页面数量：
// 1. 根据系统参数 innodb_adaptive_flushing_lwm，默认取值为 10，区间为[0, 70]
//    计算对应的 lower water mark
//    lsn age：  af_lwm = (srv_adaptive_flushing_lwm * log_get_capacity()) / 100
// 2. 若当前最老日志的年纪小于 lower water mark age，则不需要根据日志 flushing
// 3. 获得系统的异步日志刷新阀值：max_async_age，此值的具体含义可见 3.4.1.2
//    章节，近似等于日志空间的 7/8
// 4. 若当前系统的日志年纪小于异步刷新阀值，并且系统参数
//    innodb_adaptvie_flushing 未开启，则不需要根据日志 flushing
// 5. 不满足以上条件，需要根据日志进行 flushing，返回值 pct_for_lsn 的计算如下：
//        lsn_age_factor = (age * 100) / max_async_age; 取值区间近似为(0, 114]
//        pct_for_lsn =  srv_max_io_capacity / srv_io_capacity
//                       * (lsn_age_factor * sqrt(lsn_age_factor)) / 7.5;
//        pct_for_lsn 的取值范围近似为(0, 600]
//        其中，innodb_max_io_capacity 参数为硬件能够支撑的最大 IOPS，默认值
//        4000，innodb_io_capacity 默认值为 400
// 注意：
// 1.基于日志的 flushing，一次 flushing 的速度与日志的产生速度正相关，
//日志产生的速度越快，需要 flushing 的 dirty pages 数量也越多，速度也更快。
// 2. 基于日志的 flushing，其返回值有可能超过 100，也就是一次 flush 的量
// 超过 innodb_io_capacity，甚至是超过 innodb_max_io_capacity 的系统参数。
// 3. 在默认参数环境下，基于日志的 flush 策略，其速度有可能远远大于基于
// dirty pages 的策略。其主要目的是为了保证日志回收足够快，从而不至于出现
// 由于日志空间不足而导致的 async 甚至是 sync flushing 出现。
pct_for_lsn = af_get_pct_for_lsn(age);
// 真正需要 flushing 的页面数量，是根据脏页数量/日志年龄 计算出来的
// 本次 flushing 页面数量的较大值
pct_total = ut_max(pct_for_dirty, pct_for_lsn);
// 根据本次计算出来的 flushing 页面数量，与上次 flush 的平均速度，再次
// 取平均值，获得最终需要 flushing 的页面数量
n_pages = (PCT_IO(pct_total) + avg_page_rate) / 2;
// 当然，需要根据用户设置的 innodb_max_io_capacity 参数，进行一次微调
// 保证一次 flushing 的页面数量，不会超过硬件 IO 能力的处理上限
if (n_pages > srv_max_io_capacity)
      n_pages = srv_max_io_capacity;
// 开始进行本次真正的 flushing 操作
page_cleaner_do_flush_batch();
```

6.2 新增参数

MySQL 5.6.6-labs 版本中，针对新的 flush 策略，新增了如下几个参数，具体参数的功能及意

义，在上面的流程分析中已经详细给出。

- innodb_adaptive_flushing_lwm
- innodb_max_dirty_pages_pct_lwm
- innodb_max_io_capacity
- innodb_io_capacity
- innodb_flushing_avg_loops

6.3 Checkpoint 改进算法分析

在本章的 Checkpoint 算法分析章节，我们看到了 MySQL 5.5 中 InnoDB Checkpoint 算法的分析以及存在的问题，主要是 LRU List Flush 与 Flush List Flush 两个操作相互并发干扰。

因此在 MySQL 5.6.6 的优化中，将这两个写脏页的操作，统一到 page cleaner 线程中先后处理，将并发写脏页的操作转换为顺序写脏页。如此一来，Flush List Flush 的 Write I/O 流控，无需考虑 LRU List Flush 究竟有多少并发 Write 操作，只需要根据系统当前阶段的脏页数量以及日志的产生速度，计算最终的 Flush 脏页数量即可。

7 已知问题点

7.1 日志文件回卷

7.1.1 问题描述

InnoDB 的 log 文件，大小恒定，通过 innodb_log_file_size 参数可以设定。日志文件顺序写，当写到最后之后，就会覆盖写 log 文件中最老的日志。

要覆盖写最老的日志，必须得有一个前提条件：覆盖的最老日志对应的 dirty pages 已经被 flush 到 disk，否则，就会导致更新数据丢失的问题。

对于小日志文件，或者是大量更新操作的 InnoDB 系统，日志文件消耗的非常快，轻易可能就写玩一圈。而 Dirty pages flush & Checkpoint 的机制不一定能够保证对应的脏页已经 flush 到 disk。此时，为了保证系统的安全性，当前正在进行的所有的更新操作必须等待 Checkpoint 点推进之后，才能继续写日志，提交更新。

以下，我们就来分析一下 InnoDB 是如何判断这个情况的，并且在检测到该情况之后，如何实现 Checkpoint 点的推进，释放出多少可用的日志空间？

7.1.2 InnoDB 处理

在 Checkpoint 章节，我们提到，srv_master_thread 函数中，每个 1S，会调用 log_free_check

函数，判断当前是否有足够的空闲 log 空间？是否需要进行 Checkpoint 以释放 log 空间？

其实，log_free_check 函数不仅仅在 srv_master_thread 函数中会调用，而且会在所有的 DML 操作时调用。包括 insert(row_ins_index_entry_low), delete(), update(), purge(row_purge_remove_clust_if_poss_low), … DML 前调用此函数的目的，就是为了检查当前是否有足够的空闲日志空间，是否需要做 Checkpoint。log_free_check 函数很简单，如下：

```
log_free_check(void)
{
    if (log_sys->check_flush_or_checkpoint) {
        log_check_margins();
    }
}
```

判断参数 log_sys->check_flush_or_checkpoint 是否设置，若设置，则调用函数 log_check_margins，否则什么都不做。现在，最重要的变成了了解 log_sys->check_flush_or_checkpoint 参数何时会被设置为 TRUE。

7.1.2.1 check_flush_or_checkpoint

- check_flush_or_checkpoint = TRUE
  - log0log.c::log_init
    初始化时设置为 TRUE
  - log0log.c::log_close

```
if (log->buf_free > log->max_buf_free)
    check_flush_or_checkpoint = TRUE;
```

若 log buffer 当前空闲起始位置，已经超过定义的 max_buf_free(一般来说是 log buffer 的一半左右)，则将 check_flush_or_checkpoint 设置为 TRUE。

```
if (!oldest_lsn ||
    lsn – oldest_lsn > log->max_modified_age_async ||
    checkpoint_age > log->max_checkpoint_age_async)
        check_flush_or_checkpoint = TRUE;
```

log_close 函数在每次写完日志(undo & redo)之后，都会调用：

mtr_commit -> mtr_log_reserve_and_write -> log_close

- check_flush_or_checkpoint = FALSE

  在 log_free_check -> log_check_margins -> log_checkpoint_margin 调用中，若不需要做同步 Checkpoint (checkpoint_sync = FALSE)，则将 check_flush_or_checkpoint 参数设置回 FALSE。

7.1.3 改进方案

7.2 数据文件扩展

7.2.1 问题描述

在网文 InnoDB Insert 抖动问题极其改进 [20] 中，作者详细的分析了数据文件扩展对于 DML 操作带来的问题。摘录其中的分析一节如下：

> *在 InnoDB 里，扩展表空间的操作是在语句执行过程中，由执行线程直接调用的。*
>
> *尤其是对于一些表每行比较大，则会出现每插入几条记录就需要扩展表空间。*
>
> *虽然有 insert buffer 和 write ahead logging 策略保证在执行线程中不直接操作表数据文件，但扩展表空间的操作会导致更新的 tps 出现瞬间低点。现象如下图。实际上整体 TPS 也受此影响。*

接下来我们看看 InnoDB 内部是如何实现数据文件扩展的。

7.2.2 InnoDB 处理

构造一个 Insert 导致数据文件扩展的用例，得到如下的函数调用流程：

ha_innobase::write_row -> … -> row_ins_index_entry_low -> btr0cur.c::btr_cur_pessimistic_insert -> fsp0fsp.c::fsp_reserve_free_extents -> fsp_try_extend_data_file

正如以上所示，数据文件的扩展是在用户线程中完成的

fsp0fsp.c::fsp_reserve_free_extents 函数(预约空闲空间)分析：

1. 获取读写 tablespace header 的权限

```
rw_lock_t *latch = fil_space_get_latch(space, &flags);
mtr_x_lock(latch, mtr);
fsp_header_t *space_header = fsp_get_space_header(space, …);
```

读写 tablespace header 前，需要获取 tablespace 的 rw_lock，并加 x 锁

2. 计算 tablespace 上有多少空闲 extents

```
// 链入 free list 链表的 extents 个数
n_free_list_ext = flst_get_len(space_header + FSP_FREE, mtr);
// extent 分配上限，大于这个 extent number 的 extents 都是 free 的
free_limit = mtr_read_ulint(space_header + FSP_FREE_LIMIT, MLOG_4BYTES, …);
// extent 分配上限之上，有多少 free extents
n_free_up = (size – free_limit) / FSP_EXTENT_SIZE;
// free extents 总数量
n_free = n_free_list_ext + n_free_up;
```

InnoDB 中的空闲 extents，由两部分组成：

一部分是已初始化，但是未写过的 extents——free_limit 控制；

另一部分是曾经分配过，但是被完全回收的 extents——链入 free_list 链表。

这两部分组成了当前 tablespace 剩余的 free extents 的总数量。

3. 计算预约 extents 数量。

a) 若预约 extents 数量大于 n_free，则需要调用下面的 fsp_try_extend_data_file 函数，扩展数据文件

b) 若预约 extents 的数量小于 n_free，则调用 fil_space_reserve_free_extents，预约 extents，并且返回。

4. 读写 tablespace header 的权限，何时释放？

fsp0fsp.c::fsp_reserve_free_extents 函数通过 mtr_x_lock 对 tablespace 加锁，用于读写 tablespace 的 header page。

在数据文件扩展完毕，并且修改 header page 之后，按理说应该及时释放此 latch，但是我在流程中一直未能定位到 unlock 操作的主动调用。

那么这个 latch 是何时释放的呢？

首先再来看看 mtr_x_lock 函数：

mtr_x_lock -> mtr0mtr.ic::mtr_x_lock_func

```
rw_lock_x_lock_func(lock, 0, file, line);
// x lock成功之后，将lock对象存入到mtr的memo对象之中(dyn hash)
mtr_memo_push(mtr, lock, MTR_MEMO_X_LOCK);
```

在以下函数调用中，将会遍历 mtr 中的加锁链表，然后逐个释放：

row_ins_index_entry_low -> mtr0mtr.c::mtr_commit -> mtr_memo_pop_all -> mtr_memo_slot_release -> pfs_rw_lock_x_unlock_func -> sync0rw.ic::rw_lock_x_unlock_func

在 mtr_memo_pop_all 函数中，将会遍历 mtr 的 memo，释放其中的各个 lock。

对于前面提到的 tablespace header page lock，也同样在此时释放。

注：由于 tablespace header page lock 一直持有到此时才释放，同时中间又涉及到了数据文件扩展的同步 I/O，导致整个 InnoDB 系统在这段时间之内，不能够进行任何 DML 操作。这也是数据文件扩展时，整个系统的更新事务数量急剧下降的原因所在。

fsp0fsp.c::fsp_try_extend_data_file 函数(扩展数据文件)分析：

1. 计算当前 tablespace 大小

```
size = mtr_read_ulint(header + FSP_SIZE, MLOG_4BYTES, mtr);
```

tablespace 的第一个 page 的第 8 个字节开始的 4 bytes，记录了当前 tablespace 的大小

关于 File space header 的数据结构，可参考 fsp0fsp.c 文件，也可以参考 Appendix A.

2. 计算需要扩展的 free extents 的个数

a) extent 的大小

#define FSP_EXTENT_SIZE (1 << (20 - UNIV_PAGE_SIZE_SHIFT))

64 个 pages

b) 系统 tablespace，一次扩展 8 个 extents = 512 个 pages

c) 用户 tablespace

i. 第一次，扩展一个 extent

ii. 表大小小于 32 个 extents 时，扩展一个 extent

iii. 表大小大于 32 个 extents 时，扩展四个 extents

3. 调用函数 fil0fil.c::fil_extend_space_to_desired_size 扩展数据文件

a) 获取扩展权限：fil_mutex_enter_and_prepare_for_io

```
mutex_enter->(&fil_system->mutex);
```

b) 一次扩展 64 个 pages，若大于 64，则分为多次扩展，while 循环

```
buf_size = ut_min(64, size_after_extend - start_page_no) * page_size;
buf2 = mem_alloc(buf_size + page_size);
```

```
buf = ut_align(buf2, page_size);
// 将分配的空间全部设置为0
memset(buf, 0, buf_size);
```

c) 调用 os_aio 函数进行 64 个 pages 的扩展

```
// sync io，同步将 buf 写入 tablespace 文件
os_aio(OS_FILE_WRITE, OS_AIO_SYNC, buf, …);
```

d) 扩展成功，释放权限：fil_node_complete_io

e) 最后，调用 fil_flush 函数，将 write flush 到 disk

4. 修改 File space header，更新对应的 8-12 bytes

```
mlog_write_ulint(header + FSP_SIZE, new_size, MLOG_4BYTES, mtr);
```

7.2.3 改进方案

7.2.3.1改进一

同样可以参考网文 InnoDB Insert 抖动问题极其改进 [20]，作者提出了一种预先分配的策略。能够手动预先分配足够的空间，从而避免用户线程扩展数据文件产生的性能问题。
同时，以上的改进可以结合监控与报警使用。在函数 fsp_reserve_free_extents 中，实时监控 n_free，若 n_free 小于特定值，则发出报警，用户收到报警之后，可以手动扩展数据文件。与 Oracle 的方案类似。

7.2.3.2改进二

在函数 fsp_reserve_free_extents 中，实时监控 n_free，若 n_free 小于特定值，此时不是发出报警，而是唤醒一个后台线程，执行扩展数据文件的工作，而用户线程可以直接退出。不影响用户线程的性能。

7.3 软中断的影响

上面提到的两个问题点，是 InnoDB 引擎内部的，其实 mysql 在运行过程中，还有一个极大的与操作系统与硬件相关的问题点——(软)中断的处理。关于软中断的原理以及 Linux 下软中断的实现，可参考[25][26]。
在 mysql 系统中，最主要的软中断有两类：Disk 软中断；网卡软中断。而软中断带来的问题是：所有的软中断由一个 CPU 集中处理，导致单个 CPU 利用率增加，同时处理软中断的性能下降。一个典型的例子如下：

```
mpstat -P ALL 2 3
Average:      CPU    %user   %nice    %sys %iowait    %irq   %soft  %steal  %idle    intr/s
Average:      all     1.07    0.00    0.37    0.12    0.08    0.04    0.00   98.31   1555.33
Average:        0     6.28    0.00    0.83    0.99    0.66    0.17    0.00   91.07   1555.00
Average:        1     0.00    0.00    0.17    0.00    0.00    0.00    0.00   99.83      0.00
Average:        2     0.00    0.00    0.17    0.00    0.00    0.00    0.00   99.83      0.00
```

Average:	3	0.82	0.00	1.31	0.00	0.00	0.00	0.00	97.88	0.00

上例中，intr/s 列，表示 CPU 在 internal 时间段(2S)内，每秒 CPU 接收的中断的次数(intr/total)*100；%soft 列，表示在 internale 时间段内，软中断时间(softirq/total)*100。可以看出，每秒总得中断次数为 1555.33，而 CPU 0 就处理了 1555 个，中断的处理分配极度不均衡。

解决的方案也很明确，将软中断处理分布到系统所有的 CPU 中即可。

关于网卡软中断对于 MYSQL 数据库性能的影响及解决方案，可参考网文 MYSQL 数据库网卡软中断不平衡问题及解决方案 [22]。除此之外，Google 与 Facebook 都曾经针对网卡软中断，做过相应的优化[23][27].

而对于 Disk 软中断，据我所知 **2002 年一本漫画闯天涯**同学，也对 Linux 做了优化，能够保证 Disk 软中断的处理在所有 CPU 中平均分布：block: improve rq_affinity placement。

关于中断(包括软硬中断)，Linux 系统已经做了大量的优化，简单归纳起来，主要有以下一些：

- 高级可编程中断控制(Advanced Programmable Interrupt Controller, APIC) [30]
- 中断亲和力(SMP IRP Affinity) [28][29]
- IRQ Balance [32]
- MSI & MSI-X [34][35][36]
- 网卡软中断处理[22][23][27]
- 磁盘软中断处理[37]
- 中断监控 [33]

8 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] http://code.google.com/p/google-mysql-tools/wiki/InnodbIoPerfConfig Innodb IO Performance Config (talk about new system variables added by google)
[15] http://code.google.com/p/google-mysql-tools/wiki/InnodbIoPerformance Innodb IO Performance (google's mysql team)
[16] http://blogs.innodb.com/wp/2010/04/innodb-performance-aio-linux/ InnoDB now supports native AIO on linux
[17] http://lse.sourceforge.net/io/aio.html Kernel Asynchronous I/O (AIO) Support for Linux
[18] http://blog.yufeng.info/archives/741 Linux 下异步 I/O(libaio)的使用以及性能
[19] http://www.ibm.com/developerworks/cn/linux/l-async/ 使用异步 I/O 大大提高应用程序的性能
[20] http://dinglin.iteye.com/blog/1317874 InnoDB Insert 抖动问题极其改进
[21] http://www.quora.com/Domas-Mituzas/answers/XFS What underlying file system works best with a SQL database?
[22] http://blog.yufeng.info/archives/2037 MYSQL 数据库网卡软中断不平衡问题及解决方案
[23] http://lwn.net/Articles/328339/ Software receive package steering
[24] http://blog.sina.com.cn/s/blog_3dbab2840100j4ey.html mpstat 使用详解
[25] http://blog.csdn.net/yuanyufei/article/details/776263 linux 软中断的读书笔记
[26] http://bbs.ednchina.com/BLOG_ARTICLE_135152.HTM Linux 中软中断机制分析
[27] http://www.facebook.com/note.php?note_id=39391378919 Scaling memcached at Facebook
[28] http://www.alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux SMP affinity and proper interrupt handling in linux
[29] http://www.cs.uwaterloo.ca/~brecht/servers/apic/SMP-affinity.txt SMP IRQ Affinity
[30] www.osdever.net/tutorials/pdf/apic.pdf Advanced Programmable Interrupt Controller
[31] http://www.alexonlinux.com/what-is-direct-io-anyway What is direct I/O anyway?
[32] http://www.irqbalance.org/documentation.html IRQ balance documentation

[33] http://blog.yufeng.info/archives/1062 itop 更方便的了解 Linux 下的中断情况
[34] http://www.pcisig.com/specifications/conventional/msi-x_ecn.pdf msi-x ecn
[35] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/PCI/MSI-HOWTO.txt;hb=HEAD Linux MSI-Howto
[36] http://www.alexonlinux.com/msi-x-the-right-way-to-spread-interrupt-load MSI-X the right way to spread interrupte load
[37] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=bcf30e75b773b60379338768677a1301ef602ff9 Block: improve rq_affinity placement
[38] http://axboe.livejournal.com/1718.html Buffered async IO
[39] http://dimitrik.free.fr/blog/archives/2012/01/mysql-performance-linux-io.html MySQL Performance: Linux I/O
[40] http://dimitrik.free.fr/blog/archives/2012/04/mysql-performance-improved-adaptive-flushing-in-56labs.html MySQL Performance: Improved Adaptive Flushing in 5.6-labs
[41] http://dimitrik.free.fr/blog/archives/04-01-2012_04-30-2012.html#142 MySQL Performance: 5.5 and 5.6-labs @TPCC-like
[42] http://blogs.innodb.com/wp/2012/04/new-flushing-algorithm-in-innodb/ New flushing algorithm in InnoDB
[43] http://rdc.taobao.com/blog/cs/?p=1583 Linux 异步 IO 编程实例分析