InnoDB：Change Buffer

Skywalker
软件工程师，主攻数据库系统的设计与实现
关注他

介绍

change buffer（在 MySQL 5.6 之前叫 insert buffer，简称 ibuf ）是 InnoDB 5.5 引入的一种优化策略，若二级索引页不在 buffer pool 中，则将针对二级索引页的操作暂时缓存起来，等到该页从磁盘读到 buffer pool 中时再批量的（batch）apply 这些操作，从而达到减少磁盘 I/O 的目的。具体一点就是：

1. 事务 1 执行写操作（e.g update），但针对的 P1 并不在 buffer pool 中
2. 于是 client 1 将这个操作缓存到 change buffer 里，即添加一个 entry（**ibuf insert**）
3. 事务 2 需要读操作，将 P1 读到 buffer pool 中
4. 将 change buffer 里相关的缓存的操作全部合并（merge）至 P1（**ibuf merge**）
5. 将 P1 返回给用户线程

为什么必须是二级索引页，不能是主键索引页？很简单，因为主键索引要保证唯一性的约束，如果把 insert id=1 缓存起来，再次有要 insert id=1 时再缓存起来，则等 batch apply 时就会出错

change buffer 缓存的操作有三种：

- BTR_INSERT_OP：普通的 insert
- BTR_DELMARK_OP：在用户线程执行 update 和 delete 中，会将记录标记为 delete mark
- BTR_DELETE_OP：purge 线程删除二级索引中 delete mark 的数据行

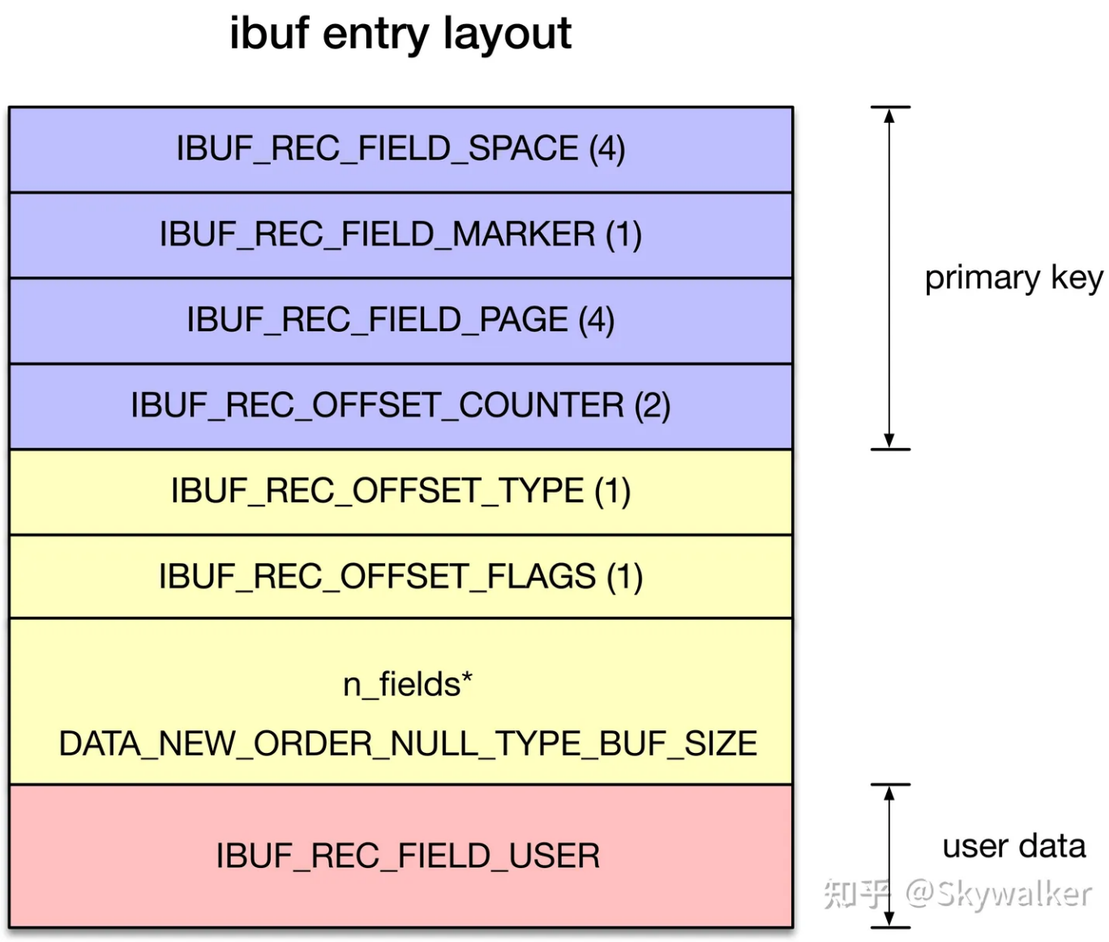
组织形式（B-tree）

change buffer 本质是一块写缓存，组织形式是 B-tree，存在于系统表空间中。其根节点位于系统表空间的第四页面（**FSP_IBUF_TREE_ROOT_PAGE_NO**）

ibuf entry layout

其缓存的每一个操作叫做一个 entry，物理结构是（详见 ibuf_entry_build）：

- IBUF_REC_FIELD_SPACE：对应二级索引页的 space id
- IBUF_REC_FIELD_MARKER：用于区分新旧版本的 entry 格式，目前默认值为 0
- IBUF_REC_FIELD_PAGE_NO：对应二级索引页的 page no
- IBUF_REC_OFFSET_COUNTER：对于同一个二级索引页，其 entry 的递增序号（**非单调递增**，详见下文）
- IBUF_REC_OFFSET_TYPE：缓存的操作的类型，IBUF_OP_INSERT / IBUF_OP_DELETE_MARK / IBUF_OP_DELETE
- IBUF_REC_OFFSET_FLAGS：待操作的用户记录格式，REDUNDANT / COMPACT
- IBUF_REC_FIELD_USER：用户记录



ibuf entry counter

entry counter 的存在是为了与 space_id 和 page_no 一起构成 entry 的主键，在 change buffer 里对同一二级索引页的 entry，其 entry counter 是递增的

在 change buffer 中插入 entry 时，先定位到待插入的位置（btr_pcur_open）：

- search tuple：（space_id, page_no, 0xFFFF）为主键
- mode PAGE_CUR_LE（<=）模式搜索 B-tree

0xFFFF 是最大的 entry counter（IBUF_REC_OFFSET_COUNTER 域为两个字节），所以 cursor 会定位到对应于同一二级索引页的具有最大 counter 的 entry，记为 max_counter。max_counter+1 即为待插入 entry 的 counter

但在每一次 ibuf merge ，清空了该二级索引页的所有 entry 后，再插入针对该索引页的新的 ibuf entry，**counter 又从 0 开始**

Change Buffer 的约束：不能引起 index SMO

需要注意的是如果一个操作可能引起二级索引页的 SMO，则该操作不能缓存在 change buffer 中。这个约束可以理解，比如针对 P1 缓存了三个 entry：entry1 / entry2 / entry3，在 ibuf merge 时，如果 entry2 使得 P1 发生分裂，那么 entry3 无法正确的 merge 至分裂后的 P1。因此 change buffer 在缓存新的操作时，新的操作不能引发这两种情况发生：

- 索引页的空间被填满（索引页分裂）
- 索引页内只剩下一条记录（索引页合并）

1. 追踪每个页的剩余空间

如果得知一个操作是否会引起二级索引页的溢出？这需要我们追踪每个页的剩余f空间。通过 ibuf bitmap page 来实现（Skywalker：InnoDB：Tablespace management（1）），ibuf bitmap page 用 4 bits 描述每个页：

- IBUF_BITMAP_FREE（2 bits）：描述页的空闲空间范围
- IBUF_BITMAP_BUFFERED（1 bit）：ibuf 中是否缓存着这个页的操作
- IBUF_BITMAP_IBUF（1 bit）：该页是否是 ibuf B-tree 的节点

IBUF_BITMAP_FREE 2 bits 的计算方式是

```
UNIV_INLINE
ulint ibuf_index_page_calc_free_bits(ulint page_size, ulint max_ins_size) {
  // max_ins_size 是这个页中目前的全部剩余空间，IBUF_PAGE_SIZE_PER_FREE_SPACE 是 32
  // page_size / IBUF_PAGE_SIZE_PER_FREE_SPACE 就是 512 bytes，可以看出这里只是粗糙
  // 的统计 max_ins_size 是 512 bytes 的几倍，max_ins_size / 512：
  //  - 大于 3（max_ins_size > 2048）：IBUF_BITMAP_FREE 记录 3
  //  - 3（1536 < max_ins_size < 2048）：IBUF_BITMAP_FREE 记录 3
  //  - 2（1024 < max_ins_size < 1536）：IBUF_BITMAP_FREE 记录 2
  //  - 1（512  < max_ins_size < 1024）：IBUF_BITMAP_FREE 记录 1
  //  - 0（max_ins_size < 512）：IBUF_BITMAP_FREE 记录 0
  n = max_ins_size / (page_size / IBUF_PAGE_SIZE_PER_FREE_SPACE);

  if (n == 3) {
    n = 2;
  }

  if (n > 3) {
    n = 3;
  }

  return (n);
}
```

在每次 insert / update / delete 后会更新 IBUF_BITMAP_FREE（ibuf_update_free_bits_if_full / ibuf_update_free_bits_low）。

2. 防止页的分裂

只有在缓存 IBUF_OP_INSERT 时才需要防止页的分裂发生。这里涉及到一个函数 **ibuf_get_volume_buffered**。该函数用于计算对于特定的二级索引页（设为 P_x），change buffer 里缓存的操作 merge 完会导致此页增长的数据量是多少（affect the available space on the page），然后与该页的剩余空间做比较。因为 IBUF_BITMAP_FREE 最大就是 3，可见能缓存的操作最多只能占用 2048 bytes

首先把 pcur 在 ibuf B-tree 中定位到缓存的关于 P_x 的最大的 ibuf record。采用的依然是 B-tree 的函数，search tuple 是（P_x space id， P_x page no，0xFFFF），search mode 是 PAGE_CUR_LE，latch mode 是 BTR_MODIFY_PREV（x-latch 左兄弟节点和当前节点） 或 BTR_MODIFY_TREE（x-latch 左兄弟节点、当前节点和右兄弟节点）。定位完成 pcur 指向 之后从 pcur 开始：

1. 首先逆向搜索，直至得到的 ibuf entry 不再是关于 P_x。若直到左兄弟节点的 infimum record 还未停止，则放弃（根据 latch order，无法再 latch 左兄弟的左兄弟）
2. 然后正向搜索，直至得到的 ibuf entry 不再是关于 P_x。若直到右兄弟节点的 supremum record 还未停止，则放弃（认为缓存的操作足够多了，会引起 P_x 的分裂）

放弃的意思是认为 change buffer 缓存的操作已经够多了，不能再缓存了

> 其实上述第二步很奇怪，pcur 理应处在 P_x 的最大的 ibuf entry（最大的意思是 ibuf entry counter 最大），为何还要正向搜索？（待解决）

```
ibuf_insert_low {
  ...
  /* Find out the volume of already buffered inserts for the same index
  page */
  min_n_recs = 0;
  buffered = ibuf_get_volume_buffered(&pcur,
```

```
            op == IBUF_OP_DELETE? &min_n_recs: NULL, &mtr);


if (op == IBUF_OP_INSERT) {
  ulint       bits = ibuf_bitmap_page_get_bits(
              bitmap_page, page_id, page_size, IBUF_BITMAP_FREE,
              &bitmap_mtr);

  // 若 ibuf_get_volume_buffered 返回 UNIV_PAGE_SIZE，那么 if 里一定是 TRUE
  if (buffered + entry_size + page_dir_calc_reserved_space(1) >
    // 根据 IBUF_FREE_BITS 计算 Page 内的剩余空间，0 / 512 / 1024 / 2048 Bytes
    // change buffer 里缓存的针对 Page X 的最大容量不能超过 2048 Bytes
    ibuf_index_page_calc_free_from_bits(page_size, bits)) {
    /* Release the bitmap page latch early. */
    ibuf_mtr_commit(&bitmap_mtr);

    /* It may not fit */
    do_merge = TRUE;

    ibuf_get_merge_page_nos(FALSE, btr_pcur_get_rec(&pcur), &mtr,
           space_ids, page_nos, &n_stored);
    goto fail_exit;
  }
}
```

在上述正向 / 逆向遍历的过程中，对于每一个 ibuf entry，计算其对于 P_x 的可能占用的空间（ibuf_get_volume_buffered_count），并累加该值。计算方式依据 ibuf entry type，如果是 IBUF_OP_DELETE_MARK 则无影响，若是 IBUF_OP_INSERT 则计算其占据的空间：

```
...
ut_ad(ibuf_op == IBUF_OP_INSERT);

get_volume_comp : {
  dtuple_t *entry;
  ulint volume;
  dict_index_t *dummy_index;
  mem_heap_t *heap = mem_heap_create(500);

  entry = ibuf_build_entry_from_ibuf_rec(mtr, rec, heap, &dummy_index);

  volume = rec_get_converted_size(dummy_index, entry, 0);

  ibuf_dummy_index_free(dummy_index);
  mem_heap_free(heap);

  return (volume + page_dir_calc_reserved_space(1));
}
```

3. 防止页的合并

计算 apply 完 change buffer 里的缓存，该索引页的剩余记录使多少。依然通过函数 ibuf_get_volume_buffered_count。以参数 n_rec 传递出来

```
static ulint ibuf_get_volume_buffered(
...
lint *n_recs,           /*!< in/out: minimum number of records on the
                            page after the buffered changes have been
                            applied, or NULL to disable the counting */
)
```

计算方式就是遇到 IBUF_OP_DELETE 就把 n_recs --，遇到 IBUF_OP_INSERT或 IBUF_OP_DELETE_MARK 要注意：

- ibuf entry 中的 user data 在 hashtable 中若没找到，说明这个 user data 是第一次遇到，插到 hashtable 中并把 n_recs ++。原因如下：

> Inserts can be done by updating a delete-marked record. Because delete-mark and insert operations can be pointing to the same records, we must not count duplicates.

如果要缓存的操作是 IBUF_OP_DELETE 且 n_recs < 2，说明这个操作可能导致页面变空，则不缓存到 ibuf 中

> 但奇怪的一点是 ibuf_get_volume_buffered 里虽然统计了所有 ibuf entry，根据 ibuf entry type 把 n_recs 增加或减少，但 n_recs 初始值是 0。难道不应该是对应二级索引页目前的记录数？（待解决）

Change Buffer 的写流程

当需要把一个新操作缓存在 change buffer 中时，在 "2. 防止页的分裂" 中定位完 pcur 后会拿到此时 pcur 指向的 ibuf entry 的 counter，这个 counter + 1 作为新操作的 counter，再根据 user data 等构建出 ibuf entry 直接插到 pcur 指向的页即可

缓存 purge 操作的特殊性

change buffer 会缓存三种操作：insert / delete mark / delete，前两种是用户事务的操作，最后一种是 purge 线程的操作。

首先我们需要知道在准备 purge 一个二级索引记录之前，均会判断这个记录是否可以被删除（row_purge_del_mark -> row_purge_remove_sec_if_poss）。拿到该二级索引记录中保存的主键，找到主键索引记录（row_search_index_entry），如果该主键索引记录存在某个 old version 满足下述三点，则该二级索引记录不能被删除（row_purge_poss_sec）：

1. old version 不是 delete mark
2. old version 的 ROW_TRX_ID 大于 purge view，即可能被活跃的 reader 访问
3. old version 与该二级索引记录的主键和二级索引列均相同

其中第2、3点很好理解，表示如果有 reader 需要拿到这个 old version，可能会通过该二级索引记录检索，但第 1 点的原因还不明确。

但如果使用 change buffer，上述的检查是不够的。比如这个场景：

1. 用户线程：delete mark (1, A)，然后数据页被淘汰出 buffer pool ...
2. purge 线程：delete (1, A)；通过 row_purge_poss_sec 的检查，因为新的 (1, A) 在下一步才会出现。准备把操作缓存到 change buffer
3. 用户线程：insert (1, A)，准备把操作缓存到 change buffer
4. 先把 insert (1, A) 缓存到 change buffer
5. 再把 delete (1, A) 缓存到 change buffer

则在 merge 的时候 insert (1, A) 可能会重用原来被 delete mark 的记录 (1, A)，即 "delete unmark"；导致 purge 直接删除 (1, A) 后，相当于事务 insert (1, A) 丢失

解决的办法很简单，当发现有 purge 线程准备缓存操作到 change buffer 中时，第 4 步中放弃缓存 insert。那么怎么发现 purge 线程准备缓存操作到 change buffer？首先在 buffer pool 中保存着这样一些 buf_block_t 结构体，它们不在 buffer chunk 中而是单独的内存区域，即 buf_pool->watch 数组，buf_block_t::state 初始状态是 BUF_BLOCK_POOL_WATCH。然后：

- 当 purge 线程从 buffer pool 中拿索引页时，如果不在，则在 buf_pool->watch 申请一个空闲的 buf_block_t，并设置其 page_id，还有 state 为 BUF_BLOCK_ZIP_PAG，并放到 page_hash 中

```
bpage->state = BUF_BLOCK_ZIP_PAGE;
bpage->id = page_id;
bpage->buf_fix_count = 1;

ut_d(bpage->in_page_hash = TRUE);
HASH_INSERT(buf_page_t, hash, buf_pool->page_hash,
    page_id.fold(), bpage);
```

这样的话在第 4 步再去检查这个页是否被设置为 watch，即在 buf_pool->watch 数组中（buf_page_get_also_watch），如果是的话直接放弃

参考

- innodb purge--二级索引
- innodb purge--聚集索引
- 关于InnoDB中mvcc和覆盖索引查询的困惑?
- MySQL · 引擎特性 · Innodb change buffer介绍
- InnoDB多版本(MVCC)实现简要分析
- MySQL · 社区动态 · MySQL5.6.26 Release Note解读
- MySQL数据库InnoDB存储引擎 Insert Buffer实现机制详解
- MySQL · 社区动态 · MySQL5.6.26 Release Note解读

编辑于 2022-11-16 14:12 · IP 属地北京

Innodb　MySQL　数据库

发布一条带图评论吧

还没有评论，发表第一个评论吧

文章被以下专栏收录

推荐阅读

一文详解InnoDB最核心组件Buffer Pool（一）

前文笔者通过一条语句的执行，从整体上讲解了 InnoDB存储引擎的架构，让大家对一条SQL语句的执行过程中，都有哪些操作有了完备的了解。InnoDB存储引擎处理一条数据，无论是查询还是修改，都...

南山的架构笔记

一文详解InnoDB最核心组件Buffer Pool（三）

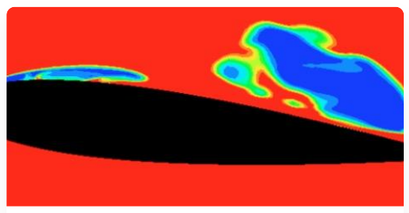
前面笔者用了两篇文章，讲解InnoDB最核心组件Buffer Pool的部分知识点，对Buffer Pool的内部结构有了一定的了解。 第一讲主要引入了缓存页的概念。 一文详解InnoDB最核心组件Buffer Pool（...

南山的架构笔记

一文详解InnoDB最核心组件Buffer Pool（二）

前文我们已经讲了Buffer Pool最基础的数据存储单元，缓存页。缓存页里存储的就是一行一行的数据，同时每个缓存页都对应了一个描述数据。那MySQL启动的时候，是如何初始化Buffer Pool的？又...

南山的架构笔记



interPhaseChangeFoam求解器解析

海兮吾槊