Mysql InnoDB 源码实现分析(一)

网易杭研-何登成

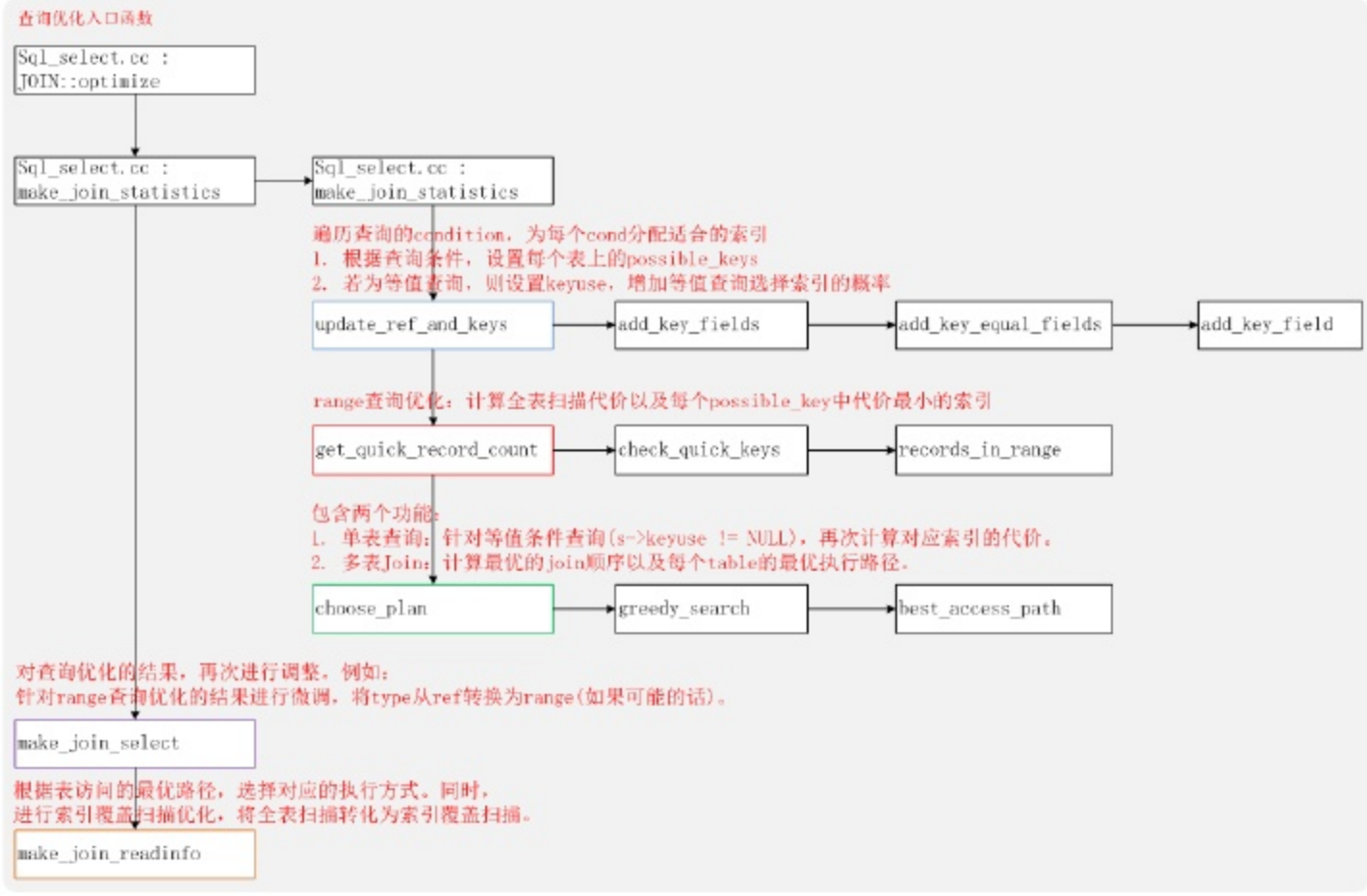
个人简介

- 姓名：何登成
- 工作：
  - 就职于杭州网易研究院，进行自主研发的TNT存储引擎的架构设计/研发工作
- 联系方式
  - 邮箱：he.dengcheng@gmail.com
  - 微博：何_登成
  - 主页：http://hedengcheng.com/

大纲

- 查询优化
- Insert Buffer
- Checkpoint
- Group Commit

查询优化-总流程


查询优化入口函数
Sql_select.cc :
JOIN::optimize
Sql_select.cc :
make_join_statistics
Sql_select.cc :
make_join_statistics
遍历查询的condition，为每个cond分配适合的索引
1. 根据查询条件，设置每个表上的possible_keys
2. 若为等值查询，则设置keyuse，增加等值查询选择索引的概率
update_ref_and_keys
add_key_fields
add_key_equal_fields
add_key_field
range查询优化：计算全表扫描代价以及每个possible_key中代价最小的索引
get_quick_record_count
check_quick_keys
records_in_range
包含两个功能：
1. 单表查询：针对等值条件查询(s->keyuse != NULL)，再次计算对应索引的代价。
2. 多表Join：计算最优的join顺序以及每个table的最优执行路径。
choose_plan
greedy_search
best_access_path
对查询优化的结果，再次进行调整。例如：
针对range查询优化的结果进行微调，将type从ref转换为range(如果可能的话)。
make_join_select
根据表访问的最优路径，选择对应的执行方式。同时，
进行索引覆盖扫描优化，将全表扫描转化为索引覆盖扫描。
make_join_readinfo

查询优化-代价模型

- 总代价模型
  - **COST = CPU Cost + IO Cost**
- CPU Cost
  - mysql上层，处理返回记录所花开销
  - CPU Cost = records / TIME_FOR_COMPARE = records / 5
  - 每5条记录的处理时间，作为 1 cost
- IO Cost
  - 存储引擎层面，读取页面的IO开销。以InnoDB为例：
- 聚簇索引(IO Cost)
  - 全扫描
    - IO Cost = table->stat_clustered_index_size
    - 聚簇索引页面数。一个页面作为 1 cost
  - 范围扫描
    - IO Cost = [(ranges + rows) / total_rows] * 全扫描时间
    - 聚簇索引范围扫描。与返回的记录成比率。

查询优化-代价模型(cont.)

- 二级索引(IO Cost)
  - 索引覆盖扫描
    - 索引覆盖扫描，减少了返回聚簇索引的IO代价
    - keys_per_block = (stats_block_size / 2) / (key_info[keynr].key_length + ref_length + 1)
    - IO Cost = (records + keys_per_block - 1) / keys_per_block
    - 计算range占用多少个二级索引页面，既为索引覆盖扫描的IO Cost
  - 索引非覆盖扫描
    - 索引非覆盖扫描，需要返回聚簇索引读取完整记录，增加IO代价
    - IO Cost = (ranges + rows)
    - ranges为多少个范围，对于IN查询，就会转换为多个索引范围查询
    - rows为范围中一共有多少记录。由于每一条记录都需要返回聚簇索引，因此每一条记录都会产生 1 cost
- 代价模型分析
  - 聚簇索引扫描代价为索引叶节点数量
  - 二级索引覆盖扫描代价较小
  - 二级索引非覆盖扫描，代价巨大
    - 未考虑类似于Oracle中的聚簇因子(Cluster factor)影响

查询优化-关键路径

- range查询优化
  - possible_keys
    - 查询条件中的所有字段，以这些字段打头的所有索引，都可能成为possible_keys
  - records_in_range
    - 针对每一个possible_key，计算给定条件的范围包含的记录数量
    - 根据范围起始值与终止值，做两次search_path，获得path_b，path_e
    - 从根节点出发，计算每一层，path_b与path_e之间包含的记录数量
    - records_in_range = records_in_upper_level(叶页面数) * records_per_leaf
- join查询优化
  - 根据给定的join查询，计算代价最小的查询计划
    - 表的join顺序最优
    - 每张表的执行路径最优
  - 递归穷举所有可能的组合与执行路径
  - optimizer_search_depth
    - 控制递归穷举深度
    - optimizer_search_depth >= join tables —> 执行计划全局最优，代价高
    - optimizer_search_depth < join tables —> 执行计划局部最优，代价低

查询优化-关键路径(cont.)

- range查询优化
  - IO代价较高，possible_keys越多，随机IO代价越高
  - records_in_range结果不稳定，导致range查询优化的结果不稳定
- join查询优化
  - CPU代价较高
  - join的tables越多，穷举最优执行计划的代价越高
- OLTP使用
  - 更应该关注range查询优化代价，尽量较少possible_keys
- 统计信息持久化
  - Mysql 5.6.2之后，InnoDB支持了统计信息持久化：rows & records_per_key等信息
  - 保证了best_access_path函数的稳定性
  - 对于range查询优化的稳定性无影响，仍旧调用records_in_range统计

查询优化-案例分析(一)

- 案例分析

  select * from TB_IMGOUT_Picture WHERE (2601629 > ID) AND (UserId = 129329421) AND (DeleteState = 0) ORDER BY ID DESC LIMIT 100;

  select * from TB_IMGOUT_Picture force index (IDX_UID_ID) WHERE (2601629 > ID) AND (UserId = 129329421) AND (DeleteState = 0) ORDER BY ID DESC LIMIT 100;

- 错误计划

```
+-----------+------+----------+-------+------+-----------+
|select_type| type |key       |key_len|rows  |Extra      |
+-----------+------+----------+-------+------+-----------+
|SIMPLE     | ref  |IDX_UID_ID|8      |149830|Using where|
+-----------+------+----------+-------+------+-----------+
```

- 正确计划

```
+-----------+-------+----------+-------+------+-----------+
|select_type| type  |key       |key_len|rows  |Extra      |
+-----------+-------+----------+-------+------+-----------+
|SIMPLE     | range |IDX_UID_ID|16     |149830|Using where|
+-----------+-------+----------+-------+------+-----------+
```

查询优化-案例分析(一)

- 错误计划

```
mysql> show status like 'Innodb_rows_read' \G
Variable_name: Innodb_rows_read
      Value: 2309581

select * from TB_IMGOUT_Picture WHERE (2601629 > ID) AND (UserId = 129329421) AND (DeleteState = 0) ORDER BY ID DESC LIMIT 100;
mysql> show status like 'Innodb_rows_read' \G
Variable_name: Innodb_rows_read
      Value: 2490700
```

测试语句(without force index)：访问了2490700-2309581= 181119 行记录

- 正确计划

```
select * from TB_IMGOUT_Picture force index (IDX_UID_ID) WHERE (2601629 > ID) AND (UserId = 129329421) AND (DeleteState = 0) ORDER BY ID DESC LIMIT 100;
mysql> show status like 'Innodb_rows_read' \G
*************************** 1. row ***************************
Variable_name: Innodb_rows_read
      Value: 2490800
```

测试语句(with force index)：访问了2490800-2490700 = 100 行记录

查询优化-案例分析(一)

- 错误计划生成步骤
  - range query optimization：选择聚簇索引(全表扫描)，tab->select->quick->index = 0，why？
  - best_path_access： 选择IDX_UID_ID索引，JT_REF，why？
  - make_join_select

- 正确计划生成步骤
  - range query optimization：选择IDX_UID_ID索引，tab->select->quick ->index = 2
  - best_path_access：选择IDX_UID_ID索引，JT_REF
  - make_join_select：JT_REF -> JT_ALL

```
if (tab->type == JT_REF && tab->quick &&
(uint) tab->ref.key == tab->quick->index &&
tab->ref.key_length < tab->quick->max_used_key_length)
```

  - select_describe: JT_ALL -> JT_RANGE

查询优化-案例分析(一)

- 不足分析
  - 二级索引回聚簇索引代价过大，导致查询优化更倾向于选择聚簇索引。
    - 参考二级索引非覆盖扫描代价模型
  - Access type = ref的优先级要远远由于Access type = range的优先级，导致查询优化更倾向于选择ref方式的访问。
    - 关于access type的不同含义，可参考网文：MySQL Explain - Reference
  - 查询优化没有针对limit语句做特殊路径处理。

查询优化-案例分析(二)

- sql语句

  select * from nkeys where nkeys.c3 not in (select distinct c3 from ncopy);

  select * from nkeys where nkeys.c3 not in (select c3 from ncopy group by c3);

- 语句二-错误计划

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	nkeys	ALL	NULL	NULL	NULL	NULL	32727	Using where
2	DEPENDENT SUBQUERY	ncopy	index	NULL	ncopy_idx	10	NULL	1	Using index

- 语句一-正确计划

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	nkeys	ALL	NULL	NULL	NULL	NULL	32727	Using where
2	DEPENDENT SUBQUERY	ncopy	index_subquery	ncopy_idx	ncopy_idx	5	func	2	Using index;

- 不同之处
  - type，possible_keys，ref ...
  - 执行时间：语句一 0.93s；语句二 39min 29.93s

查询优化-案例分析(二)

- 查询优化 与 执行路径

语句一：

```
sql_select.cc::make_join_statistics();
    update_ref_and_keys();
        add_key_fields();
            // Subquery优化：将条件下降到subquery中
            // 针对语句一，就是将nkeys.c3条件下降
            if (cond->type() == Item::FUNC_ITEM &&
                cond->functype() == TRIG_COND_FUNC)
                // cond_func->functype() == EQ_FUNC
                // 下降的查询条件为等值条件查询
                add_key_fields();
                ...
                    // 判断当前条件是否不是out filed
                    // 此处，判断失败，因为条件是out
                    is_local_field();
```

语句二：

```
sql_select.cc::make_join_statistics();
    // conds == NULL，因此不需要进入函数
    // 同时也意味着没有可以下降的查询条件
    if (conds || outer_join)
        update_ref_and_keys();
```

语句一：

```
sql_select.cc::sub_select
    while (rc == NESTED_LOOP_OK)
        // nkeys进行全表扫描，读取下一项
        error = info->read_record(info);
        rc = evaluate_join_record();
            Item_func_not::val_int();
                ...
                item_subselect.cc::
                subselect_single_select_engine::exec();
                ...
                // 根据nkeys读取的记录，构造ncopy表的
                // 查询条件，调用index_read函数读取ncopy
                // 表中满足查询条件的记录项
                ha_innobase::index_read(buf, ...);
```

语句二：

```
sql_select.cc::sub_select
    while(rc == NESTED_LOOP_OK)
        // nkeys进行全表扫描，读取下一项
        error = info->read_record(info);
        // 根据nkeys取出项，判断ncopy表
        rc = evaluate_join_record();
            item_cmpfunc.cc::Item_func_not::val_int();
                ...
                item_subselect.cc::
                subselect_single_select_engine::exec();
                ...
                sub_select();
                    while(rc == NESTED_LOOP_OK)
                        // 读取ncopy表项，索引全扫描
                        error = inof->read_record(info);
                        rc = evaluate_join_record();
                            // 将ncopy表中读取的每一项，
                            // 与nkeys表前面读取的项比较
                            // 遇见相同项，则>= nkeys项返回
                            // 完成一次not in的检测
                            if (join->having &&
                                join->having->val_int() == 0)
                                error = -1;
```

查询优化-案例分析(二)

- 分析
  - 查询优化过程中
    - 语句一的not in被优化为join，查询条件下降；语句二没有优化
  - 执行过程中
    - 语句一
      - 针对外表中的每一项，调用index_read方法查找匹配的项，一次即可
      - 内表一次index_init/index_end调用
      - 只有外表一个while循环
    - 语句二
      - 针对外表中的每一项，内表开始一个全索引扫描，结束条件是找到第一个 >= 外表项的记录
      - 内表多次index_init/index_end调用
      - 针对外表的每一项，内表开始一个新的while循环
  - 慎用not in，最好是用join改写；若一定要用，保证内表中无group by等复杂的操作

查询优化-案例分析(其他)

- sql语句一
  - select * from nkeys where (c3,c5) >= (1,1);
  - 无法进行索引扫描
  - 分析
    - mysql查询优化在update_ref_and_keys调用，根据condition分配合适索引时，只支持**MULTI_EQUAL_FUNC**，无法处理多值 >= 的条件。

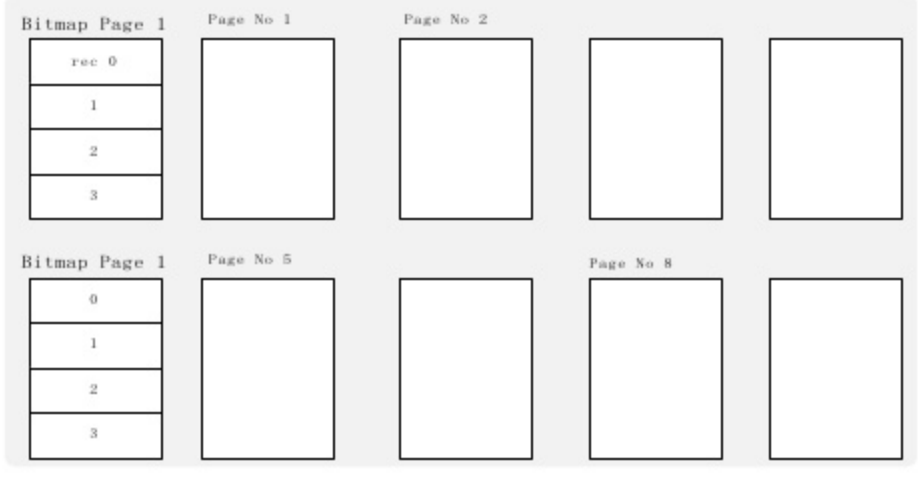
- sql语句二
  - select * from *** where id = floor(rand());
  - 无法进行索引扫描
  - 分析
    - Mysql查询优化对于rand函数有特殊处理，若查询条件中有rand，则不进行查询优化。
    - 原因在于rand函数取值，在查询优化时并没有计算出来，mysql无法优化这种情况。
    - 其他函数，例如：floor，log，pow都无此问题。

Insert Buffer

- Insert Buffer功能
  - 将非Unique索引的修改(U/D/I/P) buffer起来(前提是对应的页面不在buffer pool中)，减少随机IO
  - Insert Buffer最大可达到buffer pool的1 /2
- Insert Buffer表
  - Insert Buffer对应于系统中的SYS_IBUF_TABLE表，持久化在系统表空间(tablespace 0)中
  - tablespace 0的第五个page，就是Insert Buffer的root page(恒定不变)
- Insert Buffer记录
  - Insert Buffer记录的type为DATA_BINARY。解析出来的格式如下：
    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
  - 分析
    - 前三个字段，同一表的同一页面是一致的，保证同一页面的更新放在一起；
    - 第四个字段，以counter开始，保证同一页面的更新，按顺序存放。

Insert Buffer-空间管理

- Insert Buffer限制
  - 必须保证buffer的操作，不会引起页面的SMO(split or empty)
  - 因此，必须能够监控页面的空间利用率
- IBuf Bitmap
  - 用bitmap的方式管理tablespace中的所有页面的剩余空闲空间
  - tablespace中，每隔page_size个页面，就是一个IBuf Bitmap page (page 0, 16384, ...)。
- IBuf Bitmap record
  - 每个record，占4 bits
  - [0,1] 对应page的剩余空闲空间
    - 1 -> 剩余空闲空间大于1/32 page
    - 2 -> 剩余空闲空间大于2/32 page
    - 3 -> 剩余空闲空间大于3/32 page
  - [2,3] 对应page的insert buffer优化空间



- 空间判断
  - 已优化空间 + 当前记录空间 < 剩余空闲空间 -> 可以进行Insert Buffer优化

Insert Buffer-Merge

- 主动Merge
  - 系统主动(srv0srv.c::srv_master_thread)将Insert Buffer中的修改merge到原有页面之中
  - 触发条件
    - 过去1s内发生的I/O，小于系统能力的5%
    - 每10s，必定触发一次主动merge
    - merge数量为系统I/O能力的5% -> 10个Insertb Buffer page
    - srv_io_capacity = 200?
  - merge流程
    - 主线程发出异步I/O请求，异步读取需要被merge的页面
    - I/O handler线程，在异步I/O完成之后，进行merge
  - page选择
    - 随机定位10个Insert Buffer page
    - 读取page中的所有[space_id, page_no]到pages数组
    - 将pages数组中的所有pages，通过AIO读取
  - Insert Buffer Merge
    - 在AIO完成之后，根据[space_id, page_no]构造search_key (space_id, 0, page_no)，定位page的insert buffer记录，并进行merge

Insert Buffer-Merge(cont.)

- 被动Merge
  - 情况一
    - I/U操作，导致page split，此时需要将page读取上来，先进行merge
    - 同理，purge导致page empty，也需要进行被动merge
  - 情况二
    - Insert buffer优化由于种种原因，返回失败，需要真正读取page
  - 情况三
    - Insert Buffer占用空间达到上限，需要进行merge
    - ibuf->size >= ibuf->max_size + IBUF_CONTRACT_DO_NOT_INSERT
    - 超过Insert Buffer最大空间的10个page，必须进行同步merge
    - 此时的merge是同步操作，merge时不允许新的insert操作
    - 在Insert Buffer中随机定位一个page，将page中的更新全部merge到原有page

Insert Buffer-Percona优化

- 优化目的
  - 目的一：加快Insert Buffer的Merge与内存回收速度
    - 原因：原生InnoDB，Insert Buffer Pages回收十分缓慢。因此增加两个参数用于控制回收速度。
    - innodb_ibuf_accel_rate　　控制每次回收的ibuf page数量
    - innodb_ibuf_active_contract　　控制是否提前回收
  - 目的二：减少Insert Buffer的内存开销
    - 原因：原生InnoDB，Insert Buffer内存开销能达到buffer pool的 ½
    - innodb_ibuf_max_size　　控制Insert Buffer的大小
  - 两目标相辅相成，只有实现了目标一，才能实现目标二
  - 各参数的具体含义及设置，可参考：Improved InnoDB I/O Scalability

Checkpoint

- 原理
  - 可参见 How InnoDB performs a checkpoint 一文
- 触发条件(原生InnoDB)
  - 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
  - PCT_IO
    - #define PCT_IO(p) ((ulong) (srv_io_capacity * ((double) p / 100.0)))

Checkpoint-Adaptive Flushing

- Adaptive flushing(原生InnoDB)
  - 获取当前系统内，dirty pages的总数量： n_dirty
  - 计算过去BUF_FLUSH_STAT_N_INTERVAL 内，redo产生的平均速度：redo_avg
  - 计算过去BUF_FLUSH_STAT_N_INTERVAL 内，dirty page flush的平均速度：lru_flush_avg
  - 计算当前所需的dirty page flush的速度：n_flush_req = (n_dirty * redo_avg) / log_capacity;
  - 若n_flush_req > lru_flush_avg，则需要进行一次adaptive flushing
  - #define BUF_FLUSH_STAT_N_INTERVAL 20
    - InnoDB维护着过去20s内的统计信息，每1s对应一个interval，20s有20个interval，过期的interval定时清除
    - srv_error_monitor_thread线程调用buf_flush_stat_update函数，定期(1s)收集新的统计信息，并清除旧的

Checkpoint-Flush Dirty

- Flush Dirty Pages
  - 遍历flush_list链表，从最后一个page开始，向前遍历
  - 对于读取的每一个dirty page，尝试同时flush该page的neighbor pages(space_id相同的page)
    - 首先计算neighbor pages范围，low为当前page，up为连续的n(max = 64)个pages
    - 根据范围中的(space_id, page_no)到buffer pool的dirty pages hash表中查询
  - flush page前，必须保证对应的log已经被回刷
  - 判断是否需要使用double write
    - 直接将脏页写回
  - 需要使用double write
    - 将需要flush的dirty pages拷贝到一整块double write空间中
    - 同步写double write buffer & flush
    - 异步写dirty pages
  - flush结束，收集统计信息

Checkpoint-Percona优化

- innodb_adaptive_checkpoint
  - 新增此参数，控制不同的Checkpoint方式
  - 可取的值包括： none, reflex, estimate, keep_average，对应于0/1/2/3
  - 必须与innodb_adaptive_flushing参数配合使用
- none
  - 采用原生InnoDB adaptive flushing策略
- reflex
  - 与innodb_max_dirty_pages_pct策略类似。不同在于，原生InnoDB根据dirty pages的量来flush；此处根据dirty pages的age进行flush。flush的pages数量根据innodb_io_capacity计算。
  - 新版Percona XtraDB中，reflex策略已废弃。
- estimate
  - 与reflex策略类似。不同在于，flush的pages数量不根据innodb_io_capacity计算。
- keep_average
  - 原生InnoDB每1s触发一次flush，此策略降低flush的时间间隔，从1s降低为0.1s。

Checkpoint-estimage实现

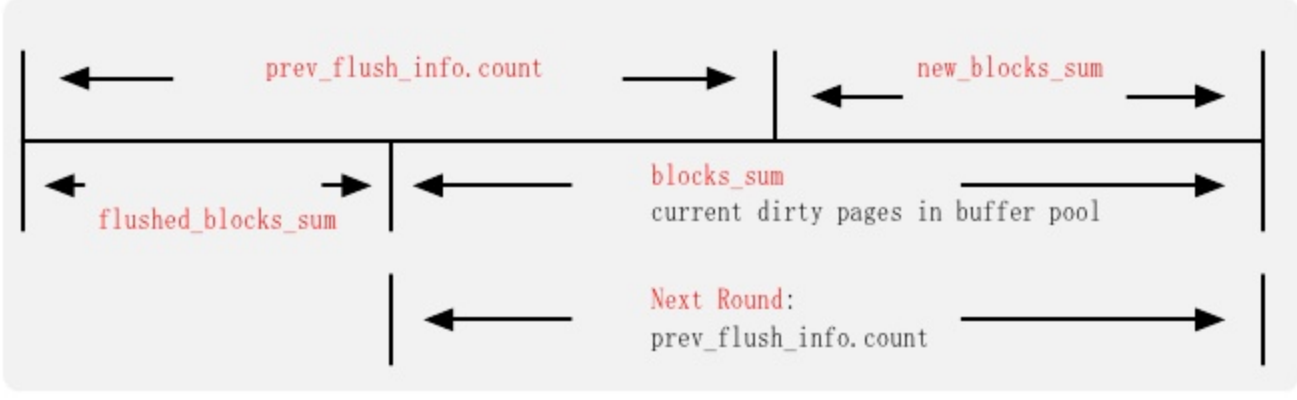
- estimate流程
  - 获取当前最老的dirty page的lsn：oldest_lsn
  - 当前未flush的日志量，超过max_checkpoint_age的1/4，则进行flush
    - if ((log_sys->lsn) – oldest_lsn) > (log_sys->max_checkpoint_age/4)
    - max_checkpoint_age：系统启动时设置(log_calc_max_ages)，近似等于系统可用log空间
  - 遍历buffer pool的flush_list链表，统计以下信息
    - n_blocks：dirty pages数量
    - level：+= log_sys->max_checkpoint_age – (lsn – oldest_lsn)
    - 每个dirty page，存在于内存中不被flush的max_age = max_checkpoint_age，(lsn – oldest_lsn)为每个page的current_age，max_age – current_age = 剩余的age。page越早被修改，剩余age越小，level越小。
  - 计算需要flush的pages数量
    - bpl = n_blocks * n_blocks * (lsn – lsn_old) / level
    - lsn_old：上次flush时记录下的最新lsn
  - 调用buf_flush_list函数
    - buf_flush_list(bpl, oldest_lsn + (lsn – lsn_old))
    - flush bpl个pages，同时必须flush到指定的lsn

Checkpoint-estimate分析

- estimate分析
  - level?
    - 越老的page，其剩余的age越小 -> level贡献越小
    - 越老的page，其flush的紧迫程度越大 -> level反比
  - bpl?
    - bpl与level成反比
    - bpl与系统中dirty pages的数量相关
    - bpl与上次flush以来log的生成速度成正比
- 举例说明
  - oldest_lsn的age为1/4 max (max = 1000)；并且日志均为上次flush以来新生成
  - 若每条日志修改同一个page
    - n_blocks = 1； level = ¾ max
    - bpl = 1 * 1 * ¼ / ¾ = 0
  - 若每条日志修改不同page
    - n_blocks = 250； level = (750 + 751 + ... + 1000) = 1766528
    - bpl = 250 * 250 * 250 / level = 8

Checkpoint-keep_average实现

- keep_average流程
  - 将原生InnoDB，每1s flush一次的策略，改为每0.1s flush一次



  - 需要刷新的pages数量
    - n_flush = blocks_sum * (lsn – lsn_old) / log_sys->max_modified_age_async
  - n_flush微调整
    - if (flushed_blocks_sum > n_pages_flushed_prev) { n_flush -= (flushed_blocks_sum – n_pages_flushed_prev); }
    - n_pages_flushed_prev：　上一次keep_average策略flush的pages数量
    - flushed_blocks_sum：　两次keep_average flush间，一共flush的pages数量
  - max_modified_age_async
    - 与max_checkpoint_age类似，近似等于日志空间的 7/8
    - 到达此限制，系统将进行异步日志flush

Checkpoint-keep_average分析

- keep_average分析
  - n_flushed？
    - 与当前系统中的dirty pages数量成正比
    - 与上次flush以来日志的产生速度成正比
    - 与max_modified_age_async成反比
  - n_flushed调整？
    - n_pages_flushed_prev为上次keep_average策略flush的pages数量
    - flushed_blocks_sum为两次keep_average策略之间，真正flush的pages数量
    - 若上次flush以来，系统已经经过了其他flush，则适当的减少本次flush pages的数量

Group Commit-问题

- Group Commit问题？
  - 何为Group Commit？
    - 参考网文：MySQL/InnoDB和Group Commit(1)
    - Group Commit的目的是为了减少fsync代价
  - 原生InnoDB支持Group Commit
  - 开启binlog后，不支持Group Commit
    - 保证InnoDB日志与Binlog日志的事务提交顺序一致性
    - 如何保证？
      开启二阶段事务，prepare阶段，获取prepare_commit_mutex，写prepare日志；
      返回Mysql上层写binlog；
      回到InnoDB，写Commit日志，完成之后，释放mutex
    - 正是由于引入了prepare_commit_mutex，Group Commit功能被禁用

Group Commit-MariaDB方案

- 参考文献
  - WL#116：Efficient group commit for binary log
  - WL#132，WL#164
- 实现分析
  - InnoDB prepare阶段，采用原本的Group Commit，一次fsync。事务prepare完成之后返回
  - mysql上层，将prepare事务加入一个queue
    - 第一个加入queue的负责余下操作；queue中其余事务等待
  - queue中的一个事务执行binlog group commit，一次fsync
  - 完成binlog group commit之后，queue中第一个事务，调用InnoDB层面提供的commit_ordered方法
    - commit_ordered方法，写commit日志，但是不fsync
  - 完成commit_ordered调用的线程，被唤醒
  - 唤醒线程进入InnoDB commit阶段
    - 由于commit日志已经写入，此处可以进行group commit，一次fsync

Group Commit-分析

- 顺序保证
  - prepare阶段，不需要顺序保证
  - 同一queue中的事务，无论是binlog，还是其InnoDB commit日志，都由queue中的第一个线程完成，与进入queue的顺序一致。从而保证了同一queue中，binlog与commit日志的顺序一致性。
  - 不同queue之间，在第一个queue完成之前，第二个queue只可加入新事务，但是不能写日志；从而保证不同queue之间不会交叉。
  - 每个queue，收集的事务越多，group commit的作用越明显。
  - 上一个queue完成之后，立即进行下一个queue的操作。
- 问题分析
  - 一组事务，全部prepare，并且写binlog完成，但是最终未commit？
  - 存储引擎层面，Handler需要实现最新的commit_ordered函数接口。

参考资料

- 参考资料
  - MySQL查询优化实现分析
  - MySQL InnoDB Insert Buffer/Checkpoint/Aio实现分析
  - MariaDB & Percona XtraDB Group Commit实现简要分析

Q & A

谢谢大家！