# Buffer Pool Implementation: InnoDB vs Oracle

网易杭研：何登成
新浪微博：何_登成
邮箱：he.dengcheng@gmail.com

# Outline

- 知识准备
  - 名词解释
  - 并发控制：Latch/Pin/Lock/Mutex
- Buffer Pool (Data Cache)
  - 数据结构
  - 初始化
  - Page/Buffer定位
  - LRU管理
  - Page/Buffer访问控制
  - Log Buffer & Write
  - Dirty Page Write & Checkpoint
  - Crash Recovery/Instance Recovery

# 名词解释

- Buffer Pool vs Data Cache
  - 本PPT，涉及的内容，是InnoDB的Buffer Pool与Oracle的Data Cache之间的实现对比
  - 未涉及InnoDB Change Buffer的实现
  - 未涉及Oracle SGA中其他pool，例如shared pool
  - **本PPT，统一使用Buffer Pool名词指代**
- Buffer vs Block vs Page
  - 本PPT，buffer/block/page对应的均为外存中的一个页面
  - **本PPT，buffer/block/page会交替使用**

# 并发控制

- 并发控制
  - Buffer Pool是全局共享资源，存在竞争，需要并发控制
  - 表中的一个页面，一条记录，同样是共享资源，需要并发控制
  - InnoDB/Oracle系统内部，有其他共享数据结构，同样需要并发控制
- InnoDB vs Oracle
  - InnoDB
    - 根据保护对象的不同，需要采用mutex/latch(rw_lock)/pin/lock等方式
  - Oracle
    - 根据保护对象的不同，需要采用latch/pin/mutex等方式
  - 区别
    - InnoDB/Oracle虽然同时采用了latch/pin/mutex等方式，但是实现/功能有较大不同

# 并发控制(InnoDB)

- **InnoDB mutex**
  - 通过CAS/TAS实现的轻量级原子锁
  - 无锁模式；不可重入；短期持有；deadlock free；

没有锁模式，“无锁/模式”可能会理解错误

  - 保护系统中重要的全局资源，临界区
    - 例如：buffer pool mutex；buffer header mutex；log sys mutex等
- **InnoDB Latch**
  - 通过CAS/TAS实现的读写锁 (RW_Lock)；
  - 短期持有；deadlock free；
  - 保护系统中的共享buffer
    - 例如：page latch

# 并发控制(InnoDB)

- **InnoDB Pin**
  - 一个标识，一个在mutex保护下的count值；不是一个锁
  - 保护内存页面不被替换
    - 例如：page pin count
- ~~**InnoDB Lock (本PPT不讲)**~~
  - 最高层次的锁，实现复杂；
  - 多种锁模式；支持死锁检测
  - 保护用户资源
    - 例如：table lock；row lock；(在buffer pool实现中，无用；本PPT不考虑)

# 并发控制(Oracle)

- **Oracle Latch**
  - 通过CAS/TAS实现的读写锁(RW_Lock)；
  - 短期持有；deadlock free
  - 保护系统中重要的全局资源，临界区
    - 例如：cache buffers chains latch；cache buffer lru chain latch；checkpoint queue latch等
    - 功能与InnoDB mutex类似

- ~~**Oracle Lock(本PPT不涉及)**~~
  - 最高层次的锁，实现复杂；大约 200 bytes
  - 多种锁模式；持有队列、等待队列；支持死锁检测
  - 保护用户资源
    - 例如： table lock；row lock等
    - Oracle Lock，功能与InnoDB Lock类似

# 并发控制(Oracle)

- **Oracle Pin**
  - 轻量级Lock；大约 40 bytes
  - 两种锁模式(S/X)；持有队列、等待队列；支持死锁检测
  - 保护内存对象并发访问；保护对象不被替换出内存
    - 例如：page pin (在cache buffers chains latch保护下获取)
    - **功能 = InnoDB Page Latch + Pin；同时有Latch与Pin的作用**

- ~~**Oracle Mutex(本PPT不使用)**~~
  - 实现类似于Oracle Latch
  - 功能类似于Oracle Latch与Oracle Pin
  - Data Cache中无用，本PPT不考虑

# Buffer Pool功能

- 功能
  - 存储外存页面在内存中的镜像
    - 只读镜像；更新镜像；版本镜像；
  - 只读镜像：InnoDB 与 Oracle一致
    - 非脏页
  - 更新镜像：InnoDB 与 Oracle一致
    - 脏页
  - 版本镜像：Oracle存在，InnoDB不存在
    - Oracle： 页级多版本
    - InnoDB：行级多版本

Why?
Oracle的undo是物理记录(Block)，
InnoDB的undo是逻辑记录(Row)

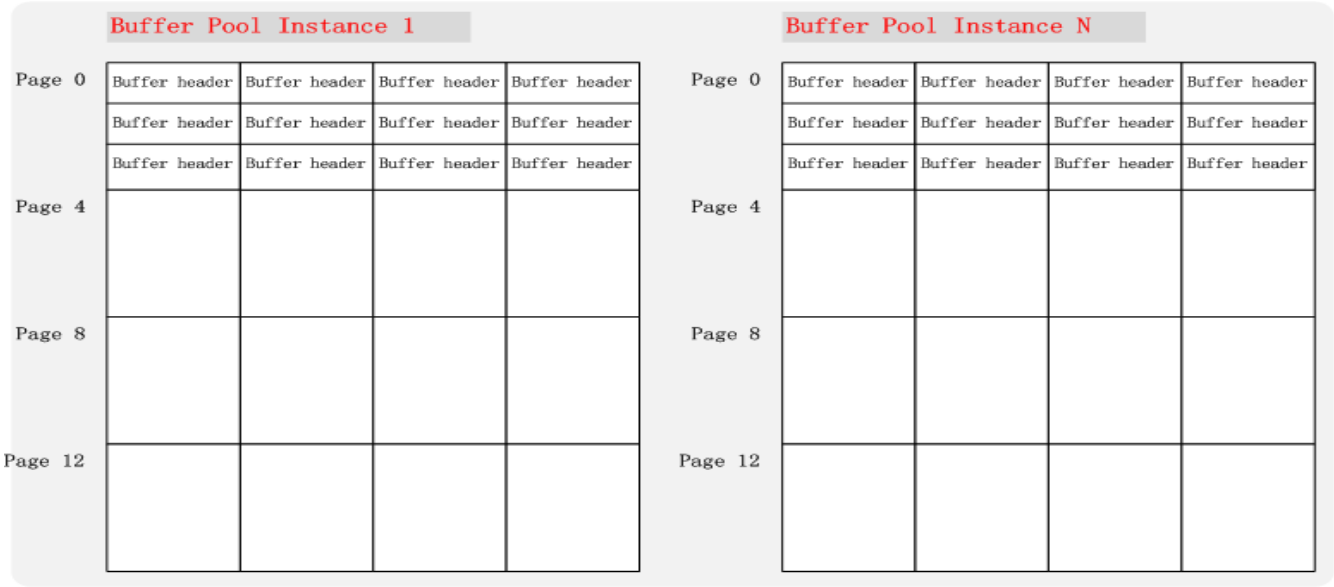
# Buffer Pool Users

- InnoDB Buffer Pool Users
  - Buffer Pool
    - 存储外存页面在内存中的读写镜像；
  - ~~Change Buffer(本PPT不涉及)~~
    - 缓存二级索引(非唯一)上的更新操作
    - **innodb_change_buffer_max_size**
      - default：25%；max：50%

- Oracle Buffer Pool Users
  - ~~shared pool(本文不考虑)~~
  - db_cache(_size) (本PPT对象)
    - 针对default page size的所有对象的cache ——> 最大
  - ~~db_keep_cache(_size)~~
  - ~~db_recycle_cache(_size)~~
  - ~~db_Nk_cache(_size)~~

# Buffer Pool Structure(InnoDB)

- InnoDB Buffer Pool 结构




- **innodb_buffer_pool_size**
  - 定义整个buffer pool的大小(包括change buffer)
- **innodb_buffer_pool_instances**
  - 定义整个buffer pool可被划分为多少实例 (Since MySQL 5.5.4)

# Buffer Pool Structure(InnoDB)

- InnoDB Buffer Pool结构(续)
  - Buffer Pool Header
    - Buffer Pool Instance的头结构
  - 主要成员
    - **mutex**
      - 控制整个buffer pool的并发访问
    - **blocks**
      - buffer pool中所有pages的头结构数组
    - **page_hash**
      - buffer pool中所有pages快速定位的hash table
    - **freed_page_clock**
      - 标识当前buffer pool一共替换了多少页面数(LRU替换算法)
    - ****_list**
      - buffer pool中的各类list

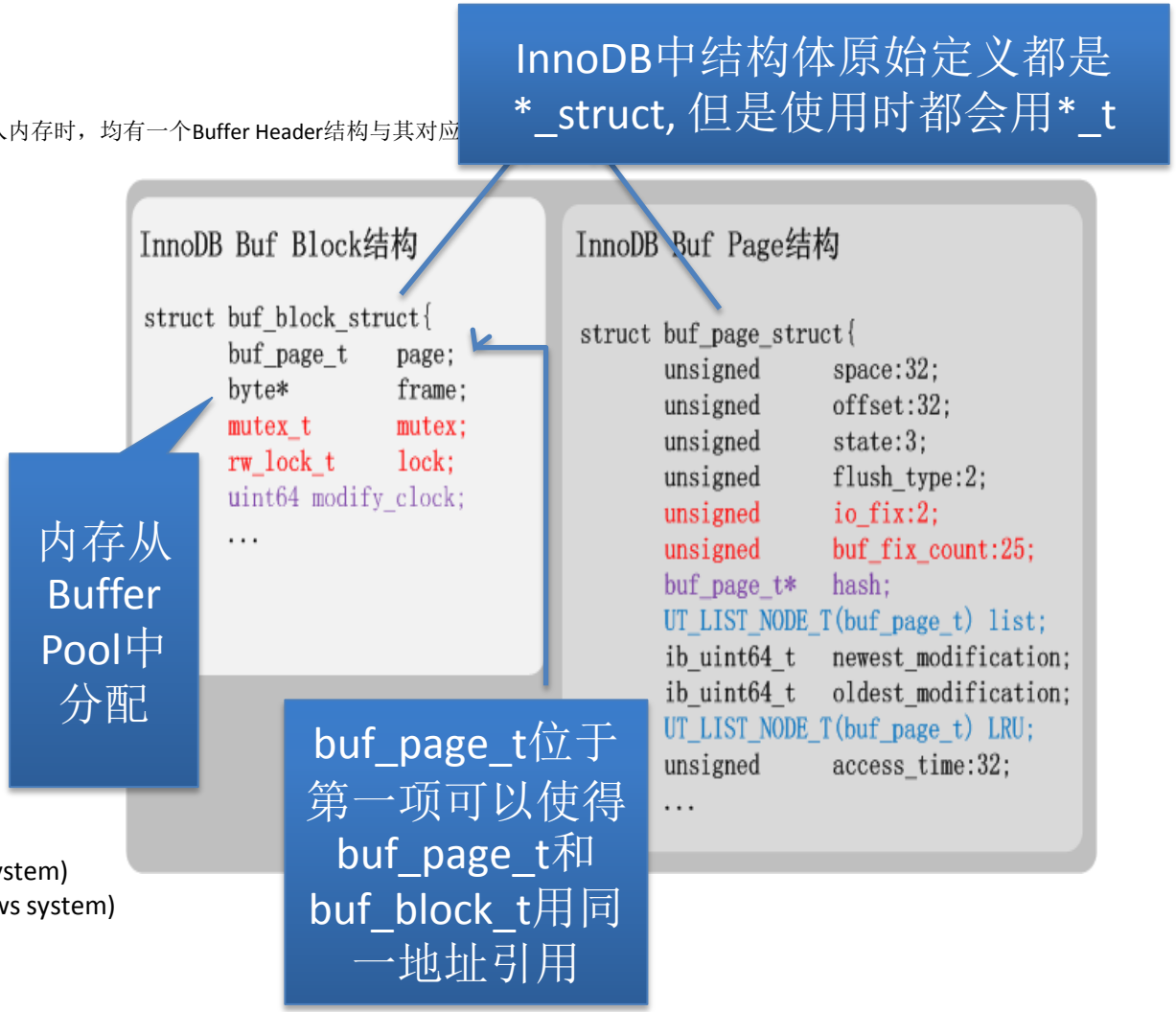
InnoDB Buf Pool结构

```
struct buf_pool_struct{
        mutex_t         mutex;
        byte*           frame_mem;
        byte*           frame_zero;
        byte*           high_end;
        ulint           n_frames;
        buf_block_t*    blocks;
        buf_block_t**   blocks_of_frames;
        ulint           max_size;
        ulint           curr_size;
        hash_table_t*   page_hash;
        ulint           n_pend_reads;
        ulint           freed_page_clock;
        UT_LIST_BASE_NODE_T(buf_block_t) flush_list;
        UT_LIST_BASE_NODE_T(buf_block_t) free;
        UT_LIST_BASE_NODE_T(buf_block_t) LRU;
        buf_block_t*    LRU_old;
        ulint           LRU_old_len;
        ...
};
```

新版本中是buf_page_t

# Buffer Pool Structure(InnoDB)

- InnoDB Buffer Pool 结构(续)
  - Buffer Header
    - 每一个外存页面，读入内存时，均有一个Buffer Header结构与其对应
- Buffer Header
  - **frame**
    - 外存页面镜像
  - **lock**
    - Page Latch
  - **mutex**
    - Buffer Header mutex
  - **modify_clock**
    - 页面逻辑时间戳
  - **io_fix**
  - **buf_fix_count**
    - Page Pin
  - **access_time**
    - 首次访问时间
  - …
- Buffer Header Size
  - ~ 800 bytes (64-bit linux system)
  - ~ 300 bytes (32-bit windows system)



# Buffer Pool Structure(InnoDB)

- InnoDB Buffer Pool 结构(续)
  - 注意事项
    - 1. buffer header结构，不属于buffer pool内存
      - **InnoDB所占用内存要大于innodb_buffer_pool_size大小**
    - 2. buffer pool内存，系统启动时既完全分配，但是不写数据
    - 3. 每一个内存buffer，在指定buffer header之后，永不更改
    - 4. 指定buffer header，可以定位buffer；指定buffer，可以定位到其对应的buffer header
      - buffer_block_struct->frame
      - buf_pool_struct->blocks_of_frames[(buffer – frame_zero) / UNIV_PAGE_SIZE]

# Buffer Pool Structure(Oracle)

- Oracle Buffer Pool(Data Cache)结构
  - 分层结构
    - Block -> Granule -> Working Data Set -> Data Cache
  - Block
    - 与外存页面大小一致；一般为8192 bytes；
  - Granule
    - 若干个Block的集合；例如：1024 个8K blocks = 8 MB
  - Working Data Set
    - 每个Data Cache，被划分为若干个Working Data Sets
    - 每个Working Data Set，跨越Data Cache的所有Granules
  - Data Cache
    - 由大量Granules构成
    - 由若干个Working Data Sets构成

# Buffer Pool Structure(Oracle)

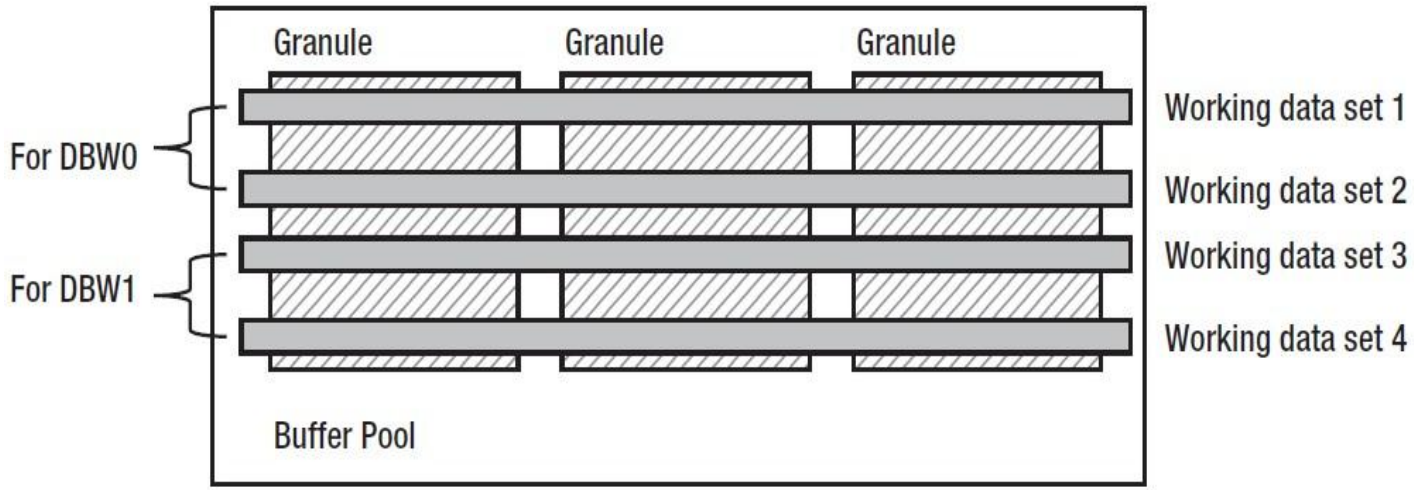
- Oracle Buffer Pool结构(续)
  - Granules结构

| Granule header | | | |
|---|---|---|---|
| Buffer header | Buffer header | Buffer header | Buffer header |
| Buffer header | Buffer header | Buffer header | Buffer header |
| Block buffer | Block buffer | Block buffer | Block buffer |
| Block buffer | Block buffer | Block buffer | Block buffer |

    - Granule分为两部分：buffer headers区域；Block Buffer区域；
    - buffer header size: 150 bytes ~ 250 bytes
  - **Granules意义**
    - 将内存按照块划分，方便内存以Granule级别在Buffer Pool的各Users间移动

# Buffer Pool Structure(Oracle)

- Oracle Buffer Pool结构(续)
  - Working Data Sets




  - Working Data Set意义
    - 增加写并发：每个Working Data Set对应一个DBWR进程
    - 增加读并发：每个Working Data Set，有自己的LRU链表(等)

    - 数量：　　cpu_count/2; cpu_count;
    - DBWR数量：　　max = cpu_count/8 < number of working data sets

# Buffer Pool Structure(总结)

- Multi-Instance Buffer Pool vs Working data Sets
  - 相似之处
    - InnoDB可将Buffer Pool划分为多个Instances (Since MySQL 5.5.4)
    - Oracle可将Data Cache划分为多个Working Data Sets
    - 均可降低内存操作的并发冲突
  - 不同之处
    - InnoDB：将指定的外存页面映射到固定的Buffer Pool Instance

```
// 根据space_id与page_no计算fold，
// 并从fold计算出对应的buffer pool instance Number
Buf0buf.ic::buf_pool_get(space_id, page_no)
        // 将page_no右移6位，去除最低6位，why?
        ignored_offset = page_no>> 6;
        fold = space_id<< 20 + space_id + ignored_offset;
        index = fold % srv_buf_pool_instances;
```

因为InnoDB预读单位是64个Pages（Extent），去除低6位，$2^6$=64，可以保证预读不会跨Buffer Pool

    - Oracle：外存页面可在所有Working Data Sets分配存储空间

# Buffer Pool Init(InnoDB)

- InnoDB Buffer Pool初始化
  - 1. 根据**innodb_buffer_pool_size**与**innodb_buffer_pool_instances**参数，划分各buffer pool instance大小(需要去除change buffer大小)；
  - 2. 根据buffer pool instance大小，计算额外需要分配的buffer headers空间；
  - 3. 初始化各buffer pool instance，前部分集中分配buffer headers，后部分为free buffers；并为每个free buffer指定一个buffer header；
  - 4. 初始化buffer headers，包括创建Latch，Mutex；初始化Pin (0)等(buf0buf.c::buf_block_init)；
  - 5. 创建Buffer Pool的Page Hash 哈希表
    - Hash表并发控制通过buffer pool mutex保护；
  - 6. 将所有的free buffers链入buffer pool instance的free list链表，等待分配；

# Buffer Pool Init(Oracle)

- Oracle Buffer Pool初始化
  - 详细流程未知

# Page定位

- Page定位
  - **给定一个外存页面，如何快速判断此页面是否在Buffer Pool中**
  - 均使用Hash表进行快速查找
    - InnoDB
      - hash(space_no, page_id) -> hash bucket number
      - *函数：buf0buf.ic::buf_page_hash_get_low()*
    - Oracle
      - hash(space, file, page_no) -> hash bucket number
      - *同一buffer的各个版本，属于同一hash bucket (***_db_block_max_cr_dba***)*
        - *同一Block，CR块越多，Hash Bucket链表越长*

# Page定位(续)

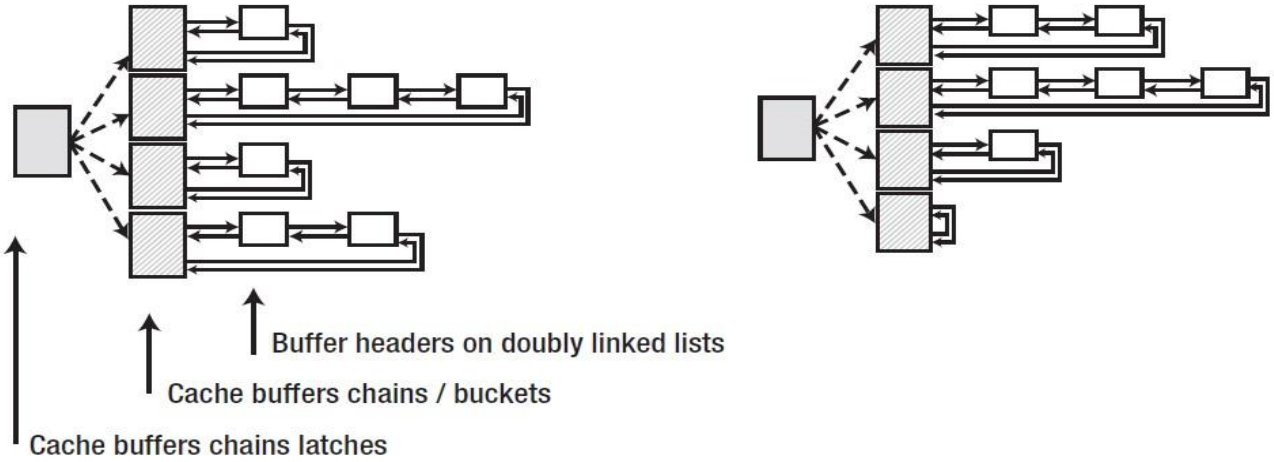
- Page定位(续)
  - **Hash表中buckets数量基本相同**
    - InnoDB
      - find_prime(buf_pool_instance_size * 2); always
      - 小于2倍于buffer pool中buffers数量的最大质数
    - Oracle
      - 8i： db_block_buffers / 4
      - **9i： the same as InnoDB**
      - 10g&11g：power of 2，bigger than db_block_buffers ——> 方便自动扩展
    - **保证每个hash bucket链表较短，平均长度1左右，降低hash冲突概率**

# Page定位(InnoDB)

- InnoDB Page定位
  - Page Hash并发控制
    - Before MySQL 5.6
      - 通过Buffer Pool Mutex保护
      - Page Hash上的所有操作：查询/Insert/Delete 串行执行
      - 缺点：
        - » 高冲突：读读冲突；读写冲突；写写冲突；
        - » 低并发
    - After MySQL 5.6(probably 5.6.4 - )
      - Page Hash上有一个RW Locks数组
      - RW Locks的数量
        - » **参数：page_hash_locks**
      - 优点：
        - » 将集中的mutex拆分为多个RW Locks
        - » RW Lock相对于mutex并发度更高，读操作并行

# Page定位(Oracle)

- Oracle Page定位
  - Page Hash并发控制



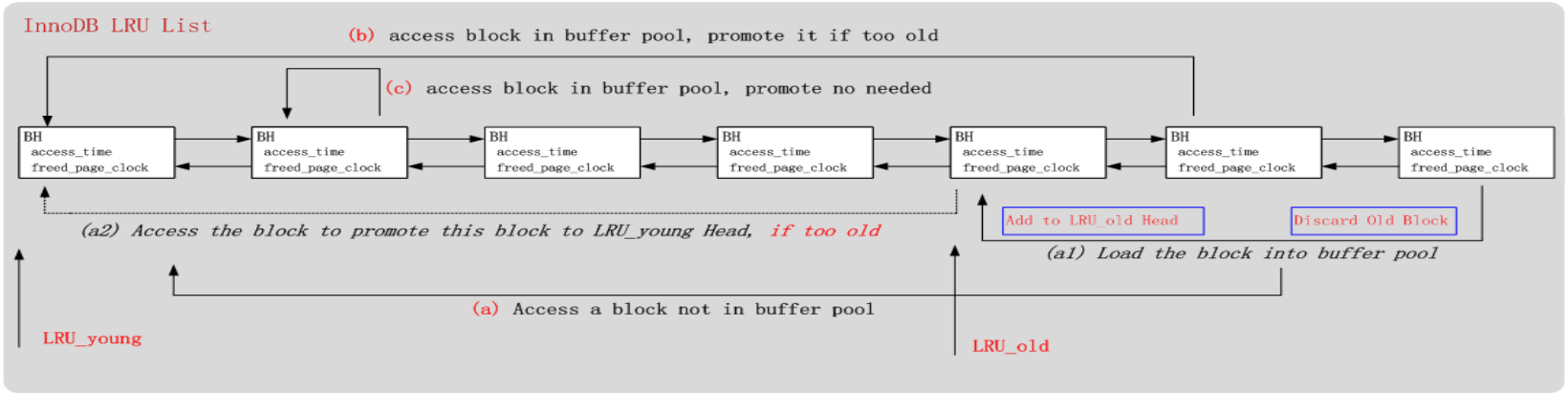

  - 通过分散的cache buffers chains latches保护 Hash Buckets入口
    - **MySQL 5.6.4优化之后，与此类似**
  - 每个latch保护的Hash Buckets数量
    - 9i & before
      - 32 or 128 buckets per latch
    - 10g & 11g
      - fixed 32 buckets per latch

# LRU管理

- LRU
  - *Buffer Pool已满时，读取外存新页面，必须选择一个内存Buffer进行替换*
  - 最近最少使用(Least Recent Used)，最经典的替换策略。InnoDB与Oracle同时采用此策略实现buffer pool的替换；
  - 算法实现
    - 相同之处
      - InnoDB/Oracle同时使用LRU实现Buffer Pool的替换
      - LRU为双向链表，并且分为冷热端两部分
    - 不同之处
      - Oracle有两个LRU链表，InnoDB只有一个
      - Oracle有TCH (Touch Count)配合LRU算法，InnoDB没有
      - …

# LRU管理(InnoDB)

- InnoDB LRU List结构




- Two Parts of LRU List
  - LRU_young
    - 针对频繁访问的buffers
  - LRU_old
    - 针对不经常访问的buffers
    - 参数：**innodb_old_blocks_pct**； [5.. 95]；默认占LRU List的3/8
- Buffer Header
  - **access_time**
    - block第一次被真正访问的时间
  - **freed_page_clock**
    - block上一次移动到LRU_young时，当前Buffer Pool Instance一共evict的blocks数量的快照

# LRU管理(InnoDB)

- LRU List 并发控制
  - 通过buffer pool mutex进行并发访问控制

- Access Block(一)
  - **(a) Access a block not in buffer pool**
    - *Index Page Read(同步读); Linear Read Ahead(异步读); ...*
    - *divided into **two steps***
  - *(a1) Load the block into buffer pool*
    - 1. buffer pool的free list中存在free block，直接使用
    - 2. 不存在free block，从LRU List的LRU_old部分 *Discard一个block*
    - 3. load page至free block，并将buffer header作为新的LRU_old Head
  - *(a2) Access the block*
    - **Rumour**：第一次访问Block，一定将Block从LRU_old提升至LRU_young的头部
    - That's true ??
    - Reason 1：promote only when the block is too old (what is too old? Explained later ...)
    - Reason 2：innodb_old_blocks_time参数

# LRU管理(InnoDB)

- Access Block(二)
  - (c) Access a too old buffer in buffer pool
    - 移动buffer至LRU_young
  - (d) Access a not too old buffer in buffer pool
    - 保持buffer在LRU List中不动
    - **不是每次访问，均会引起buffer在LRU List中的移动**

# LRU管理(InnoDB)

- 如何判断一个LRU链表中的Block是否够Young？(*MySQL 5.5.25*)
  - **算法**(*buf0buf.ic::buf_page_peek_if_young()/buf_page_peek_if_too_old()*)

```
Buf0buf.ic::buf_page_peek_if_young();

  Return((buf_pool->freed_page_clock & ((1UL << 31) - 1))
    < ((ulint) bpage->freed_page_clock
    + (buf_pool->curr_size
      * (BUF_LRU_OLD_RATIO_DIV - buf_pool->LRU_old_ratio)
      / (BUF_LRU_OLD_RATIO_DIV * 4))));
```

  - **算法解读**

    若从page上一次移动到buf pool的LRU_young以来，buf pool在此期间evict的page数量，不足LRU_young list长度的1/4，那么说明本page足够年轻，无需移动；否则， 本次访问需要移动page到LRU_young；

  - **算法功能**
    - **第一次访问**

      bpage->freed_block_clock等于0，因此判断一定失败，必定移动Block至LRU_young；
      同时设置bpage->freed_block_clock等于当前buf_pool->freed_block_clock；

    - **后续访问**

      bpage->freed_block_clock不等于0，因此判断不一定失败，因此不一定移动Block；

# LRU管理(InnoDB)

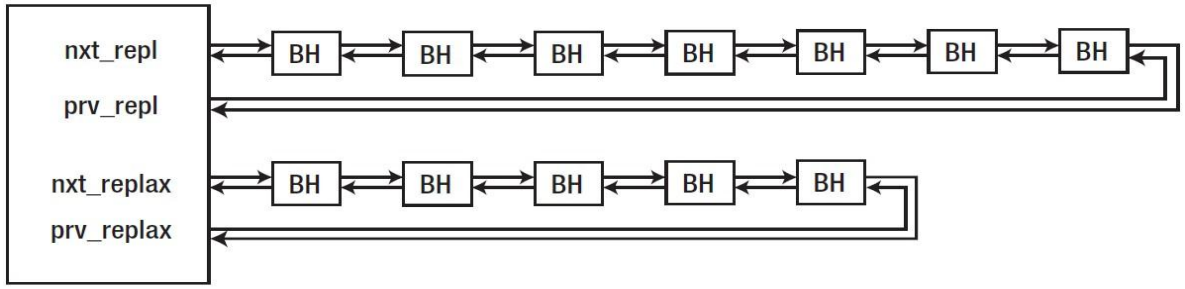
- 全表扫描影响
  - 一个大的全表扫描，可能替换LRU链表中的所有Page，导致LRU失效；
- innodb_old_blocks_time
  - **正面意义**
    - 引入此参数，消除全表扫描带来的影响；
  - **负面意义**
    - 由于此参数的判断处理，不考虑Page类型，因此也会影响到索引页面；
- Block访问，新流程
  - 第一次访问
    - 只设置access_time，不提升；
  - 后续访问
    - 若访问时间与access_time时间间隔超过innodb_old_blocks_time，提升Block；

# LRU管理(InnoDB)

- Buffer Pool Dump/Restore
  - 将buffer pool中的内容dump到外存，加速crash recovery
- Buffer Pool Dump
  - dump内容
    - 针对Buffer Pool中的每一个page，dump page的[space_id, page_no]组合
  - dump流程
    - 获取buffer pool mutex
    - 遍历LRU List，读取其中page的[space_id, page_no]，存入内存数组
    - 释放buffer pool mutex
    - 将数组中的内容写入磁盘文件
- Buffer Pool Restore
  - restore流程
    - 读取dump文件，进入内存数组
    - 将内存数组[space_id, page_no]组合排序，保证顺序读取磁盘页面
    - 按照内存数组排序之后的顺序，顺序读取磁盘页面(async)

# LRU管理(Oracle)

- LRU/TCH/TIM
  - LRU
    - Oracle同样使用LRU替换策略
    - 分 REPL_MAIN 与 REPL_AUX 两个LRU Lists

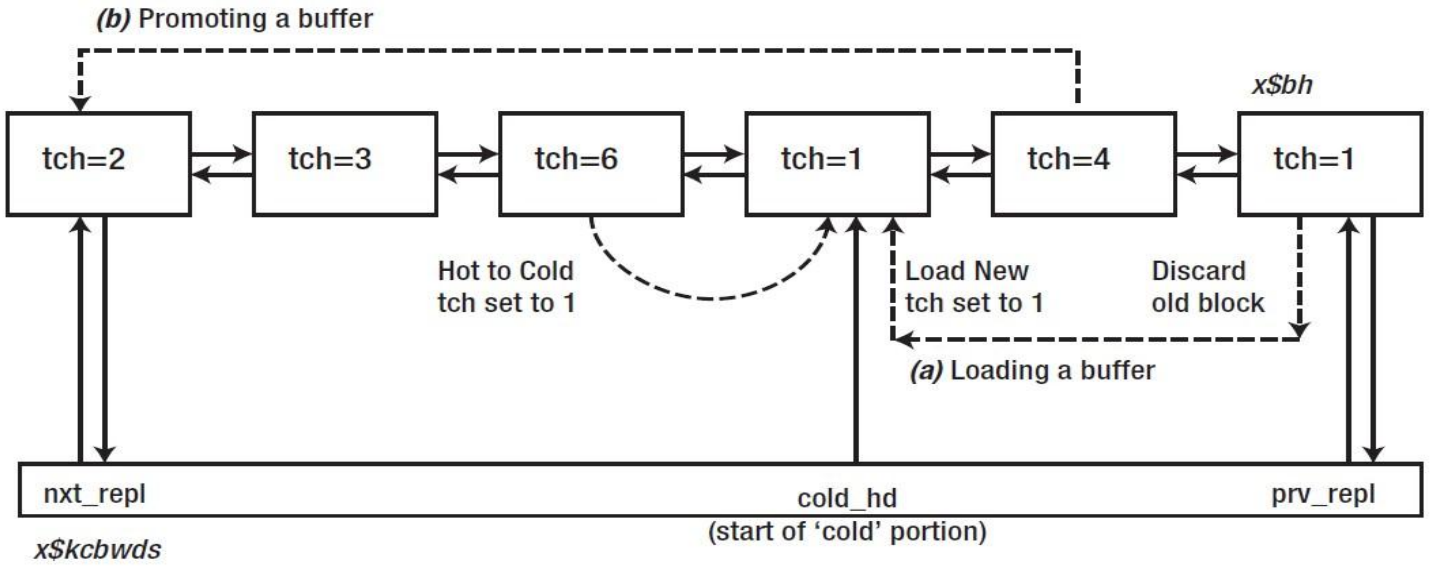



  - TCH
    - Touch Count，每个buffer header上，有一个TCH计数，标识此block被访问的次数(非精确)
  - TIM
    - Timestamp，每个buffer header上，维护一个timestamp字段，标识上一次访问此block，并且导致TCH修改的时间

- TCH
  - 维护
    - 本次访问时间，若与TIM相隔3s以上，则增加TCH，同时修改TIM为当前时间；否则不变
  - 功能
    - TCH配合LRU使用，用于减少页面访问导致的LRU List整理的次数

# LRU管理(Oracle)

- Oracle REPL_MAIN结构




- Two Parts of LRU List too
  - cold_hd指针，将REPL_MAIN List分为两部分
  - Oracle cold_hd = InnoDB LRU_old
  - 参数： **_db_percent_hot_default**；默认：50%

# LRU管理(Oracle)

- Block Access (**without REPL_AUX**)
  - (a) Access a block not in Buffer Pool
    - 1. Finding a free buffer in Working Data Set
    - 2. If not find, finding a reusable block in LRU List end
    - 3. Loading the block in current buffer, and link to cold_hd
    - 以上逻辑，与InnoDB的方案类似
      - 但是**没有主动尝试提升到MRU_head(LRU_young)的动作**
  - (b) Access a block in Buffer Pool
    - 应用更新TCH的算法进行判断，是否必要更新TCH与TIM
    - 不会尝试移动block在REPL_MAIN链表中的位置
      - Compared to InnoDB？

# LRU管理(Oracle)

- TCH in REPL_MAIN
  - TCH在REPL_MAIN中是如何处理的？
  - 替换时
    - REPL_MAIN List尾部的buffer，TCH > 1，那么将TCH减半，并提升至MRU_head
    - 只有这一种情况，会提升buffer
  - buffer跨越cold_hd时
    - *Maybe：将buffer的TCH降为1*

# LRU管理(Oracle)

- REPL_AUX
  - 解决的问题
    - 在REPL_MAIN List尾部，寻找可替换buffer时，可能碰到以下情况：
      - buffer clean，TCH = 1，not pinned ——> 最优，直接可用
      - buffer clean，TCH = 1，pinned ——> 不可替换
      - buffer clean，TCH > 1，not pinned ——> 提升至MRU_head
      - buffer dirty，TCH = 1， not pinned ——> flush dirty page，然后可用
      - buffer dirty，其他情况 ——> flush dirty page，然后可用
      - …
    - 在以上各情况下，只有情况1能够顺利找到可用buffer，其余均需要search next
    - REPL_AUX List的引入，就是为了解决此问题，*在REPL_AUX List中的buffer，TCH一定 <= 1 (可能dirty)*
  - REPL_AUX使用(free buffer search)
    - 1. Finding a free buffer in Working Data Set
    - **2. If not find, finding buffer in REPL_AUX List (discardable buffer)**
    - 3. If not find, finding a reusable block in REPL_MAIN List end
  - REPL_AUX List
    - size：half of **_db_percent_hot_default**；默认：25%

# LRU管理(Oracle)

- REPL_AUX(维护)
  - 消费buffers
    - load new block，从REPL_AUX取出buffer使用
    - construct cr-block，从REPL_AUX取出buffer使用
  - 产生buffers
    - 后台进程，逐步将REPL_MAIN List尾部的可替换buffer移至REPL_AUX
    - 一个block，超过 **_db_block_max_cr_dba** 限制的cr block，可以直接放入REPL_AUX
  - 目的
    - 后台进程尽量保证REPL_AUX List有足够的buffers；前台用户进程能够快速获取可用buffer

# LRU管理(总结)

- InnoDB vs Oracle
  - LRU List
    - Single LRU List vs Two LRU Lists
  - TCH (Touch Count)
    - No TCH vs TCH
  - Promote a Buffer
    - InnoDB: every time access, judge if promote needed
    - Oracle: promote only in LRU end and TCH > 1

# Buffer访问控制

- Buffer访问并发控制
  - 访问一个buffer(page)，如何保证并发访问的正确性？
  - InnoDB
    - Buffer Pool Mutex
      - 保护整个buffer pool
    - Page Latch
      - 保护page的内存镜像
    - Buffer Header Mutex
      - 保护buffer header结构
    - Page Pin
      - 保护page不被替换出内存
      - io_fix/buf_fix_count
  - Oracle
    - cache buffers chains latch
    - cache buffers lru chain latch
    - buffer pin

# Buffer访问控制(InnoDB)

- InnoDB Buffer访问控制
  - 1. 加buffer pool mutex，search page hash
  - 2. 若不存在，load page into buffer pool (涉及到I/O，需要释放buffer pool mutex，并重新获取)
    - 同样在buffer pool mutex的保护下将buffer加入LRU list
  - 3. 获取buffer header上的mutex
  - 4. 修改buffer header上的pin count(先获取pin，然后才是加Latch)
  - 5. 释放buffer pool mutex与buffer header mutex
  - 6. 根据页面访问类型，获取页面上的Page Latch(S/X mode)
  - 7. 在持有Latch与Pin的情况下，操作页面
  - 8. 页面操作完成，释放Page Latch，同时释放pin count (*mtr0mtr.c::mtr_commit*)
    - 若为Unique Scan/ Insert... ，释放Pin没问题，不会再次访问Page；
    - 若为Range Scan/ Table Scan...，释放Pin后，Page可能被替换，此时通过buffer header上modify_clock判断是否发生替换
    - *函数：buf0lru.c::buf_LRU_block_remove_hashed_page -> buf_block_modity_clock_inc();*

# Buffer访问控制(Oracle)

- Oracle Buffer访问控制
  - 1. 获取cache buffers chains latch，查询Page Hash
  - 2. 若不存在，load page into buffer pool (涉及到I/O，需要释放cache buffers chains latch ，并重新获取)
    - 此时同时需要cache buffers lru chain latch保护，加入REPL_MAIN List
  - 3. 获取buffer 上的Pin (S/X mode)(轻量级Lock)，并增加pin count
    - 等待事件：buffer busy waits；read by other session；
  - 4. 释放cache buffers chains latch
  - 5. 操作buffer
  - 6. 重新获取cache buffers chains latch，根据情况释放Pin
    - 参数：_db_handles_cached (session可持有pin数量)； _cursor_db_buffers_pinned (sesssion可持有pin buffers数量)
    - 保留pin目的：防止页面被替换出内存；保留的Buffer Header仍旧有效
  - 7. 释放cache buffers chains latch

# Buffer访问控制(Oracle)

- Oracle Buffer访问控制(Pin)

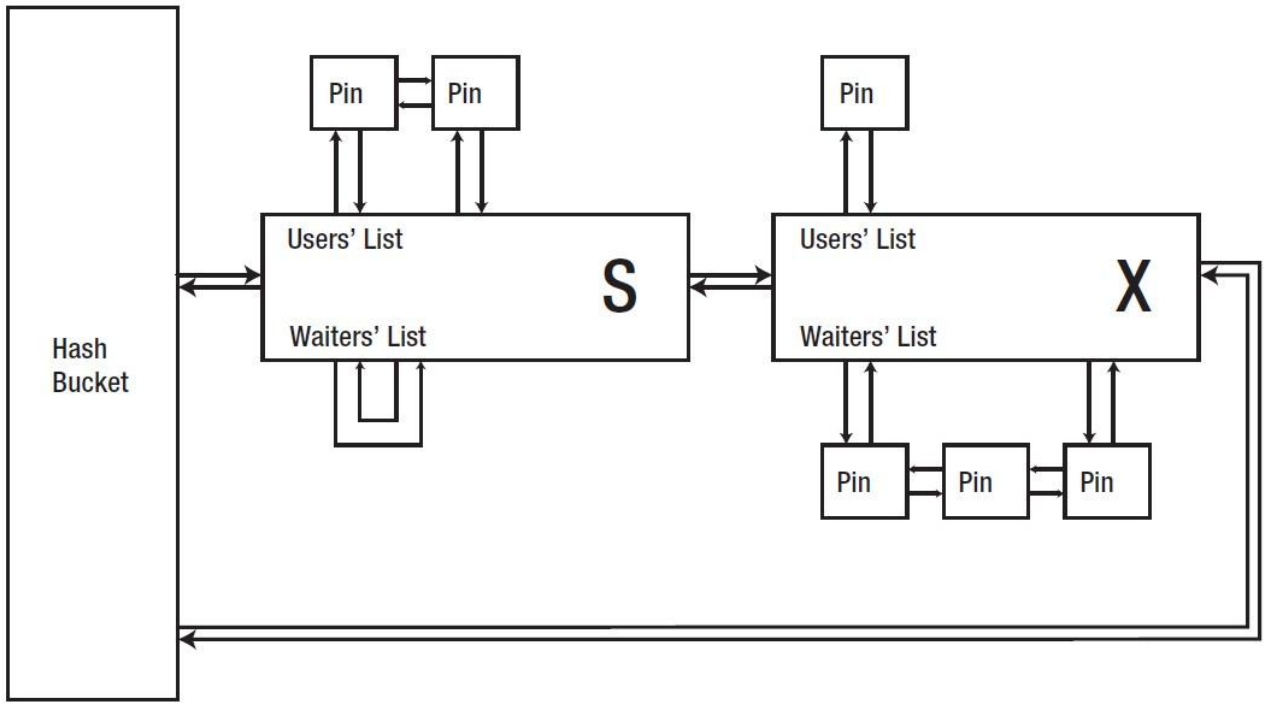



*Figure 5-7. A single hash chain with a pair of pinned buffers*

# Read Page into Buffer

- Read Page into Buffer
  - 难点
    - 多个用户，同时读取一个不在内存中的page，如何做到只有一个用户真正的进行I/O？
    - **InnoDB与Oracle同时使用加标识的方式**
  - InnoDB
    - 第一个读线程，在buffer pool mutex的保护下申请新的buffer header，并且存入Page Hash；
    - buffer header上进行标识，标识当前正在进行read (bh->io_fix = BUF_IO_READ)
      - I/O过程中，短暂的加X Latch；I/O完成之后释放
    - 其他读线程，读取Page Hash，定位到buffer header，发现io_fix设置，则等待线程1读取完成(标识复位)
      - buf0buf.c::buf_page_get_gen()
  - Oracle
    - 同样的逻辑，分配buffer header，链入Page Hash，在buffer header上设置标识
      - 过程中，持有Pin锁
    - 其他读进程，判断此标识，等待读取结束
      - 等待事件：**read by other session**

# Buffer访问控制(总结)

- 并发控制
  - Page Hash并发控制
    - InnoDB： buffer pool mutex (single)
    - Oracle： cache buffers chains latch (multiple)
    - Oracle更高
  - LRU List并发控制
    - InnoDB： buffer pool mutex (single)
    - Oracle：cache buffer lru chain latch (single)
    - Oracle更好(InnoDB仍旧使用唯一的buffer pool mutex)

# Buffer访问控制(总结)

- 并发控制(续)
  - Buffer(Page)
    - InnoDB： Page Latch + buffer mutex + pin count
    - Oracle： oracle pin
      - 轻量级的Lock，结合Latch/mutex/pin于一体
    - InnoDB vs Oracle
      - 优势
        - » InnoDB：传统实现，简洁易懂；有足够的理论基础
        - » Oracle：轻量级Lock；Pin可长期/多个持有；维护持有队列/等待队列；支持死锁/锁超时检测
      - 劣势
        - » InnoDB：多级并发控制策略；Latch与Pin分开维护，增加泄漏风险
        - » Oracle：新颖实现，缺少理论支撑；Pin的操作需要cache buffers chains latch保护

# Table Scan

- Table Scan(全表扫描)
  - InnoDB
    - 无区别对待
    - 新增参数： **innodb_old_blocks_time**
      - 第一次访问page，不提升至LRU_young，只设置access_time
      - 第二次访问page，当前时间与access_time差值大于此参数，提升；否则不提升
  - Oracle
    - 对大小表区别对待
    - 8i & 9i
      - **small**： < 2% data cache; **large**: >= 2% data cache
      - small： load至REPL_MAIN的cold_hd位置，但是TCH设置为0
      - large： load至REPL_MAIN的LRU List尾部，循环利用
    - 10g & 11g
      - **small**： < 2%; **medium**： about 10%; **large**： about 25%
      - small： normal op
      - medium： almost like small in 8i&9i
      - large： almost like large in 8i&9i

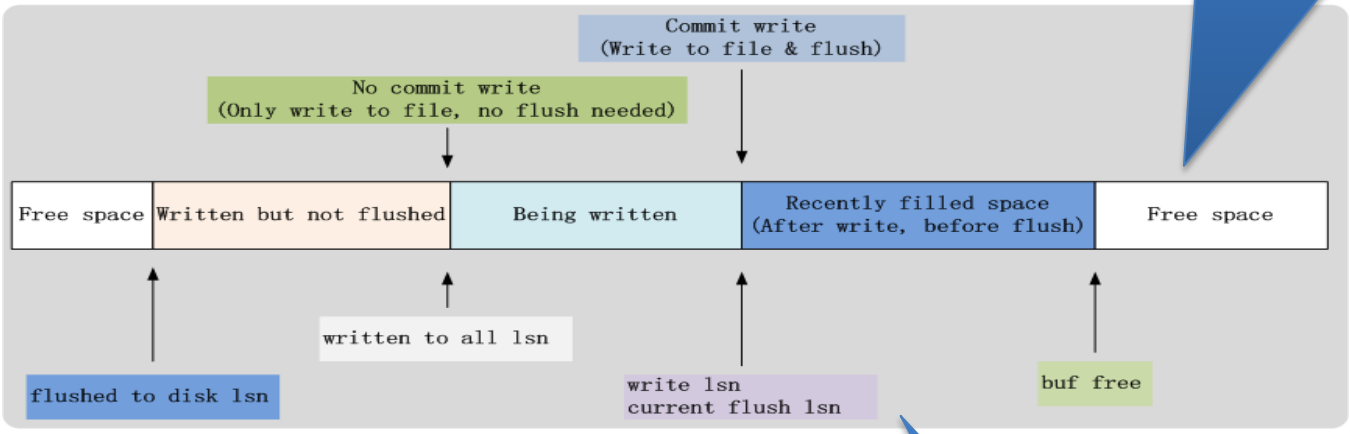
# Log

- Log
  - WAL (Write After Logging)
  - 日志先行，将日志回刷磁盘之后，才能将对应的脏页写回磁盘

- Log 实现简介
  - 相同之处
    - InnoDB 与 Oracle均在内存维护Log Buffer
    - 定期将Log Buffer中的日志写回磁盘
    - Log Buffer空间，循环使用
    - Log Block Size，均为512 Bytes = Disk Sector Size
  - 不同之处
    - InnoDB：**单一Public Log Buffer**； Oracle：**多Public Log Buffers**； 多Private Log Buffers 支持
    - InnoDB：用户/后台线程写日志； Oracle：单独LGWR进程写日志
    - InnoDB：不存在log wastage； Oracle：存在Log wastage

# Log Buffer & Write(InnoDB)

- Log Buffer 并发控制
  - 全局唯一 log_sys的mutex
  - mutex在写文件时持有；在flush文件时释放；
- Log Buffer 结构




为何free space不足时需要移动整个buffer，把空闲空闲移到尾部？

- Important Pointers
  - flushed_to_disk_lsn：此指针之前的日志已经flush到磁盘
  - written_to_all_lsn：此指针之前的日志已经写文件，但是未flush
  - write_lsn/current_flush_lsn：此指针之前的日志，正在写文件
  - buf free：log buffer的空闲起始位置

猜测是因为InnoDB都采用指针引用和拷贝内容，如果循环使用，如果一个区垮了首尾，就需要两次操作

# Log Buffer & Write(InnoDB)

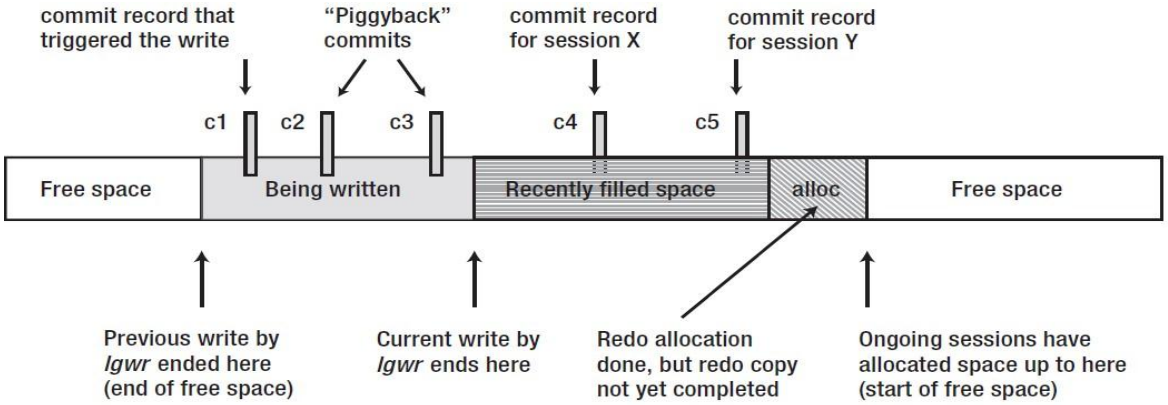
- Log Write触发
  - 事务提交/回滚
    - 参数：innodb_flush_log_at_trx_commit
    - 事务提交，一定写日志，此参数控制写完是否flush
  - log buffer的log free指针超过max_buf_free

```
#define OS_FILE_LOG_BLOCK_SIZE            512
#define UNIV_PAGE_SIZE                    (2 * 8192)

/* A margin for free space in the log buffer before a log entry is catenated */
#define LOG_BUF_WRITE_MARGIN              (4 * OS_FILE_LOG_BLOCK_SIZE)

/* Margins for free space in the log buffer after a log entry is catenated */
#define LOG_BUF_FLUSH_RATIO 2


#define LOG_BUF_FLUSH_MARGIN              (LOG_BUF_WRITE_MARGIN + 4 * UNIV_PAGE_SIZE)

Log_sys->max_buf_free = log_sys->buf_size / LOG_BUF_FLUSH_RATIO - LOG_BUF_FLUSH_MARGIN;
```

    - **InnoDB在写完日志之后，均会检查log buffer前面的空闲空间，若前面的空闲空间超过max_buf_free的一半，就会将log buffer内容向前移动**
  - 后台线程，1S检查一次
  - 用户线程，在修改页面时进行检查

# Log Buffer & Write(Oracle)

- Log Buffer 并发控制
  - redo copy latch
    - 控制redo copy到log buffer
  - redo allocation latch
    - 控制log buffer空间分配
- Log Buffer结构




  - 与InnoDB基本一致

# Log Buffer & Write(Oracle)

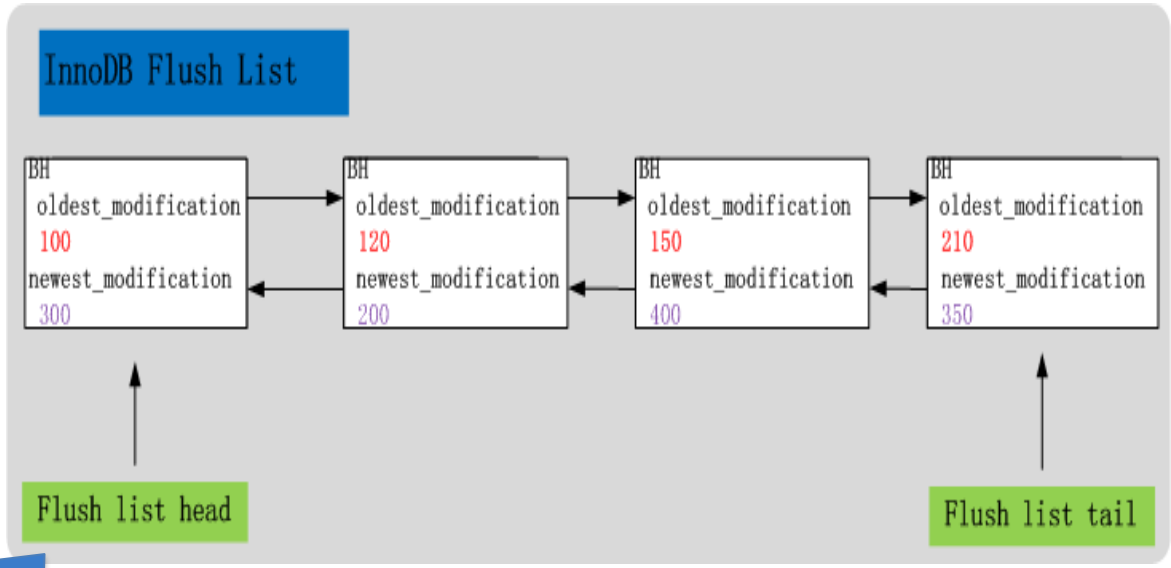
- Log Write触发
  - 每3s唤醒一次LGWR进程
  - 当Log Buffer达到1/3满
  - 当Log Buffer中的日志超过1MB
  - 事务提交/回滚
    - PL/SQL优化：PL/SQL内部的提交，不一定持久化
      - 参数：commit_write
        - [batch | immediate] [wait | nowait]

# Dirty Page Flush

- 目的
  - 目的1：加快crash recovery速度 (*checkpoint*)
    - 写回磁盘的页面，其页面LSN之前的redo日志，可以不做
  - 目的2：增加内存可用buffers数量
    - 脏页不可替换，只有写出之后，才能替换为其他外存页
- 方法
  - InnoDB
    - 针对目的1：Flush List Flush
    - 针对目的2：LRU List Flush
  - Oracle：
    - 针对目的1：checkpoint queue flush
    - 针对目的2：REPL List Flush

# Dirty Page Flush(InnoDB)

- **Flush List Flush**
  - InnoDB维护一个Flush List链表，保存所有的Dirty Pages
  - Dirty Pages，在链表中按照其第一次修改的时间排序
  - 并发控制
    - before MySQL 5.5
      - buffer pool->mutex
    - After MySQL 5.5
      - flush_list_mutex
- Flush List结构
  - oldest_modification
    - 首次修改时间
  - newest_modification
    - 最新修改时间
  - 排序
    - oldest_modification




因为用首次修改时间排序，Flush时沿着链表刷新才能保证刷新至指定LSN。否则可能有oldest_modification很早，而newest_modification很晚的，按newest_modification刷新会漏掉这些页面没有刷写回去

# Dirty Page Flush(InnoDB)

- Flush List Flush (续)

- 处理流程
  - 从Flush List Head开始，flush部分dirty pages到磁盘；
  - Before MySQL 5.6.2
    - 由**InnoDB Master Thread**负责；
  - After
    - 由**Page Cleaner**线程负责；

- Flush策略
  - innodb_adaptive_flushing；innodb_adaptive_checkpoint (Percona XtraDB)
  - 每1S；每10S；每0.1S；
  - …

- Flush约束
  - 写回磁盘的dirty page，必须保证日志已经flush到newest_modification lsn；
  - 写回磁盘的dirty page，被修改为clean page，从Flush List移除；
  - 并不从内存LRU List摘除，因此不会释放内存空间；

# Dirty Page Flush(InnoDB)

- **LRU List Flush**
  - 将LRU List的尾部的dirty page写回磁盘，并释放空间
- LRU List Flush触发
  - Before MySQL 5.6.2
    - 用户线程，在load new page to buffer pool之后，判断目前buffer pool中是否有足够空闲页面；
  - After MySQL 5.6.2
    - 后台page cleaner线程，每1s唤醒一次，进行LRU List Flush
    - 优势：
      - 减少LRU List Flush对于用户响应的影响
      - 能够一次释放更多的buffers
- 并发控制
  - 在buffer pool mutex保护下进行LRU List Flush
- 约束
  - 被写出的dirty page，一定是no latch, no pin

# Dirty Page Flush(InnoDB)

- LRU List Flush (续)
  - 用户线程处理 (Before MySQL 5.6.2)
    - BUF_LRU_FREE_SEARCH_LEN
      - 5 + 2 * **BUF_READ_AHEAD_AREA** (64) = 133
    - BUF_FLUSH_FREE_BLOCK_MARGIN
      - 5 + BUF_READ_AHEAD_AREA = 69
    - BUF_FLUSH_EXTRA_MARGIN
      - (BUF_FLUSH_FREE_BLOCK_MARGIN / 4 + 100) / srv_buf_pool_instance
      - 117 / srv_buf_pool_instance

    - 需要Flush的dirty pages数量
      - **BUF_FLUSH_FREE_BLOCK_MARGIN + BUF_FLUSH_EXTRA_MARGIN – n_replaceable**
      - **其中：n_replaceable (当前可用的Pages数量)**
        - » buffer pool free list中的pages数量
        - » LRU List尾部BUF_LRU_FREE_SEARCH_LEN内可替换page数量

    - 持有buffer pool mutex，进行Flush

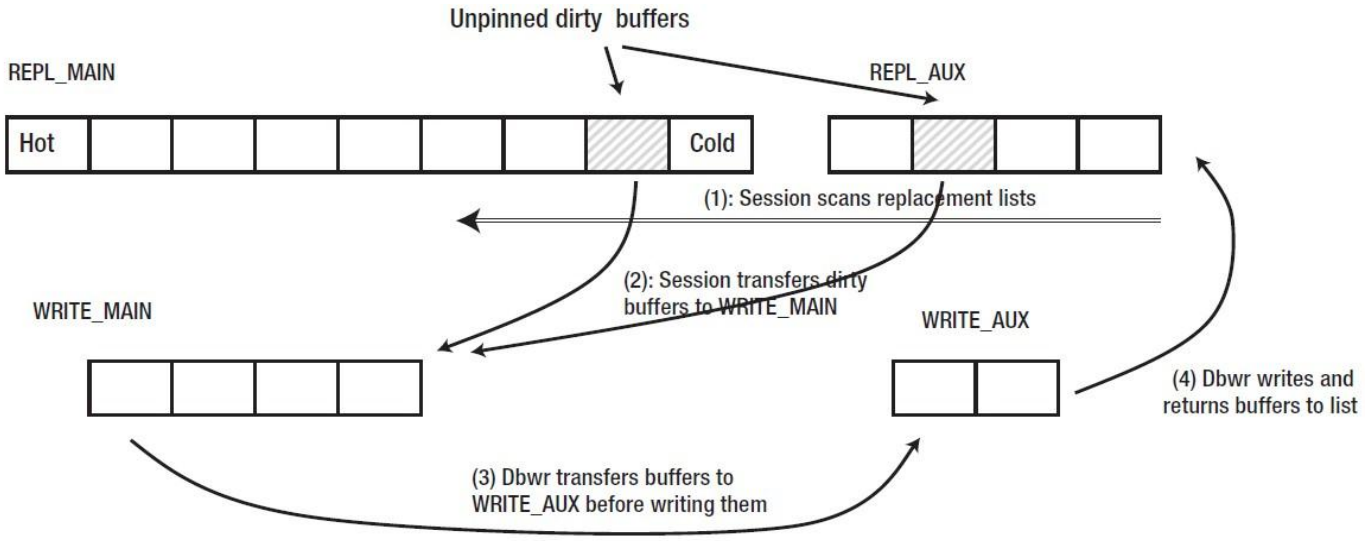
  - Page Cleaner线程处理 (After MySQL 5.6.2)
    - 每1s唤醒一次，进行LRU List Flush
    - 持有buffer pool mutex
    - **innodb_lru_scan_depth default: 1024**
      - 控制从尾部遍历LRU List的长度
      - 分批：PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE (100)

# Dirty Page Flush(Oracle)

- **Checkpoint Queue Flush**
  - 每个working data set，包含两个Checkpoint queue
    - 两个queue，减少冲突，增加并发访问性能
    - 一个queue在flush时，用户进程可以访问另为一个queue
  - 内存buffer第一次修改，加入任意一个Checkpoint queue
    - 按照LRBA排序；LRBA(Low Redo Block Address) = oldest_modification in InnoDB
- 功能
  - 从checkpoint queue的head开始，沿着queue写部分dirty pages
    - **BWR：Block Written Records，每写出一个dirty page，将page最新的change scn写入日志**
  - 由后台DBWR进程负责，每3s唤醒一次
- 并发控制
  - 每个Checkpoint queue，都有一个对应的Checkpoint queue latch
- 约束
  - 与InnoDB一致，写回磁盘的dirty page，必须保证其对应的所有日志已经写回磁盘
  - 不会释放内存空间；在dirty page写出之后，将page从Checkpoint queue移除，标识为clean

# Dirty Page Flush(Oracle)

- **REPL List Flush**




- 功能
  - 将REPL(MAIN/AUX)中的脏页写回磁盘，释放足够可用/可替换空间
- 实现
  - 新增WRITE_MAIN，WRITE_AUX List
  - 用户进程，在分配不到空间时，负责将REPL List中的dirty page移至WRITE_MAIN
    - 唤醒DBWR进程，等待时间：free buffer waits
  - 后台进程，定期检查REPL链表尾部，将dirty pages移入WRITE_MAIN
  - DBWR进程，将WRITE_MAIN链表中的dirty page移入WRITE_AUX，从AUX链表写出，并且移回REPL_AUX链表

# Checkpoint

- Checkpoint
  - 降低系统crash recovery所需时间
    - 将dirty pages写回磁盘，最终的目标是推进Checkpoint
  - 标识当前系统可以从哪条日志开始进行恢复
    - InnoDB：Checkpoint为一个LSN (Log Sequence Number) 值
    - Oracle：Checkpoint为一个SCN (System Change Numer) 值

- Checkpoint主要种类
  - **Full Checkpoint**
    - Checkpoint时，将内存中所有的脏页写回磁盘
    - 优点：实现简单
    - 缺点：耗时长；对系统性能影响大
  - **Incremental(Fuzzy) Checkpoint**
    - 按照页面首次修改的时间，每次将部分脏页写回磁盘，逐步推进Checkpoint
    - 优点：Checkpoint可调优；对系统性能影响较小；
    - 缺点：实现复杂
  - **InnoDB与Oracle均选择Incremental Checkpoint**
    - InnoDB只支持这一种Checkpoint方法
    - Oracle支持更多其他类型Checkpoint

# Incremental Checkpoint(InnoDB)

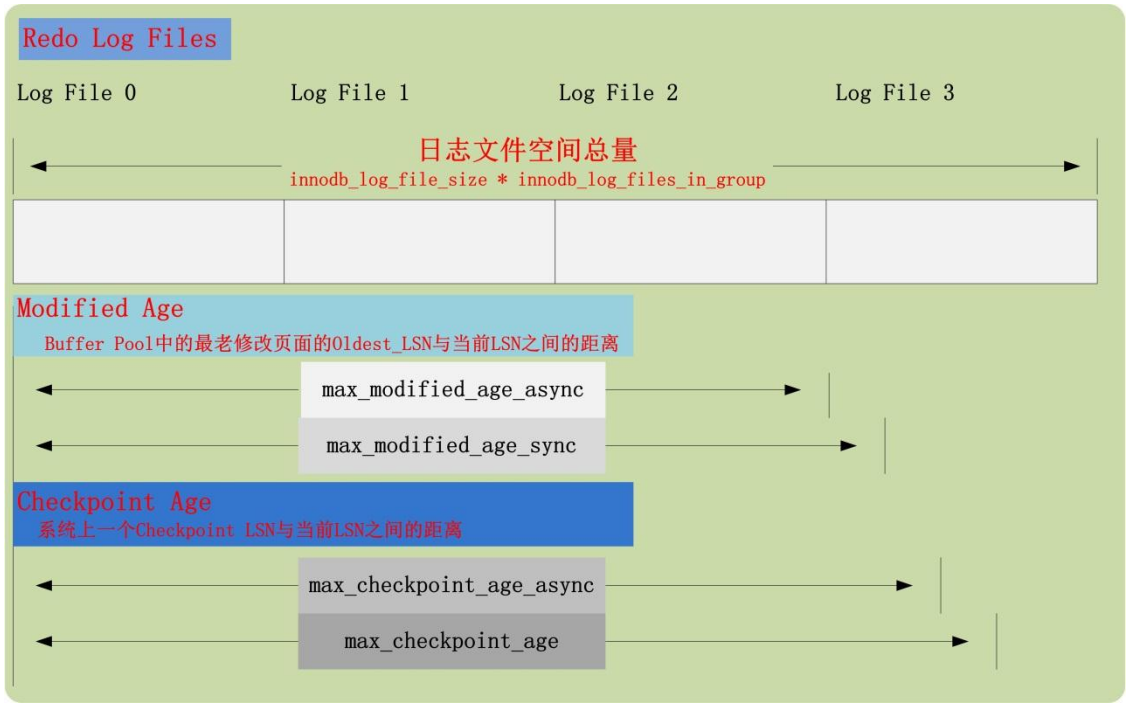
- Checkpoint信息
  - InnoDB Checkpoint，是一个LSN (Log Sequence Number)，指向日志文件的特定位置
  - 标识小于此Checkpoint LSN的脏页，已经写出磁盘，无需Redo
  - 是系统crash recovery的起点
  - Checkpoint LSN，保存在每个日志组日志文件第一页之中
    - *log0recv.c::recv_find_max_checkpoint()*
    - **LOG_CHECKPOINT_1 = OS_FILE_LOG_BLOCK_SIZE**
- Checkpoint更新
  - 将buffer pool Flush List中，第一个dirty page的oldest_modification (最小) 作为新的Checkpoint LSN；
  - 在此之前，必须保证新的Checkpoint LSN之前的dirty pages全部Flush倒磁盘；

# Incremental Checkpoint(InnoDB)

- 日志文件空间总量
  - 可用的日志文件空间总量 = 单个日志文件大小 * 日志组中的日志文件个数
  - innodb_log_file_size
  - innodb_log_files_in_group
    - default = 2;
- Modified age
  - Modified age = log->lsn – oldest_lsn
  - Modified age，定义为上次Flush List Flush以来，系统一共产生多少新的日志
- **Checkpoint age**
  - Checkpoint age = log->lsn – log->last_checkpoint_lsn
  - Checkpoint age，定义为上次写出Checkpoint LSN以来，系统一共产生多少新的日志
- 日志文件使用基本规则
  - **Checkpoint age <= 日志文件空间总量**
  - 否则，日志被覆盖，无法恢复

# Incremental Checkpoint(InnoDB)

- Checkpoint触发点
  - Flush List Flush
  - 更新操作，修改页面之前
- Checkpoint更新条件
  - **超过max_modified_age_async**
    - 异步Flush List Flush
  - **超过max_modified_age_sync**
    - 同步Flush List Flush
  - **超过max_checkpoint_age_async**
    - 异步Checkpoint
  - **超过max_checkpoint_age**
    - 同步Checkpoint
- 注意
  - 两个max_..._sync参数达到时，系统短暂无法进行U/D/I操作



# Incremental Checkpoint(Oracle)

- Incremental Checkpoint in Oracle
  - Oralce最主要的Checkpoint类别之一
  - Oracle的Checkpoint信息，为SCN (System Change Number)
  - 系统crash recovery的起点
  - Checkpoint SCN，存储于Oracle的Control File之中
    - 并发控制：control file enqueue (lock)
- Checkpoint触发
  - 每3s，由后台CKPT进程，更新Checkpoint信息

# Other Checkpoints(Oracle)

- **Media recovery checkpoint**
- **Thread checkpoint**
  - 在log file切换时，进行以上两种Checkpoint
  - 将当前系统的dirty pages全部写出磁盘 (*逐步写出，慢*)
  - CKPT进程修改每个data file，写入checkpoint scn，标识前一个日志文件无需Redo
  - media recovery checkpoint scn，备份恢复起点
- **Interval checkpoint**
  - 参数log_checkpoint_interval/log_checkpoint_timeout
- **Object reuse checkpoint**
  - Truncate table时，根据object queue，做对象级别checkpoint，写出dirty pages
    - object queue，同一对象的所有pages (clean/dirty)的链表
- **Object checkpoint**
  - Drop table时，根据object queue，做对象级别checkpoint
- **Tablespace checkpoint**
  - 根据file queue，做文件级别checkpoint
    - file queue，同一tablespace dirty pages的链表
- …

# InnoDB Checkpoint增强(想法)

- Flush_LSN
  - InnoDB数据文件的第一页中记录着Flush LSN
  - 标识数据文件中，所有page_lsn小于此Flush LSN的脏页均以写回磁盘
    - Media Recovery Checkpoint LSN？
  - Flush LSN，仅在系统正常关闭时更新，目前无实际功能

- **InnoDB Checkpoint增强**
  - Media Recovery Checkpoint
    - 通过InnoDB Incremental Checkpoint，实现Media Recovery Checkpoint，在redo文件切换时维护
  - 意义
    - 实现更为简洁的物理备份/恢复
    - 物理备份：　Copy数据文件
    - 恢复：　　从所有数据文件中，最小的Media Recovery Checkpoint开始恢复即可
  - 优势
    - 备份过程中，无需锁表
    - 备份操作，可以在外部实现
  - 难点
    - Flush Lsn的维护
    - 数据文件Copy时的Partial Write问题
      - XtraBackup方案：嵌入InnoDB代码内部，Copy每一个Page，都对Page进行Check

# Crash Recovery (简介)

- Crash recovery
  - 系统崩溃后，重新恢复到一致点
    - 崩溃前，所有提交事务做的修改，一定持久化
    - 崩溃前，所有未完成事务做的修改，一定回滚
  - Oracle称之为Instance recovery
- Crash recovery流程
  - 首先，根据日志，向前redo (roll forward)
  - 然后，将未提交事务，向后undo (roll backward)

# Crash Recovery (InnoDB)

- InnoDB crash recovery流程
  - 起点
    - log file第一个文件中的Checkpoint LSN
  - Redo阶段
    - 日志文件，一次遍历
    - 优化
      - Redo batch apply
        - » 收集redo，按照操作page分组，存入Hash Table
        - » 当Hash Table Size达到上限时，提取其中的每个page，batch apply
      - Red Black Tree(红黑树)
        - » 恢复时，page的oldest_modification不一定有序，引入红黑树，加速排序过程
  - Undo阶段
    - 将最后未提交的事务，undo

# Instance Recovery (Oracle)

- Oracle instance recovery 流程
  - 起点
    - control file中的Incremental checkpoint SCN
  - Redo阶段
    - 优化
      - 日志文件，两次遍历
      - 第一遍
        - » 根据BWR日志，构建每个page被写出的最大SCN，存入Hash Table
      - 第二遍
        - » 读取日志并应用，但是过滤掉日志SCN小于Hash Table中保存的最大SCN，这些日志无需应用
        - » 目的：减少最终需要读取的page数量
  - Undo阶段
    - 将最后未提交的事务，undo

# 参考资料


- *Jonathan Lewis*. Oracle.Core:Essential.Internals.for.DBAs.and.Developers
- *Andrey S. Nikolaev*. Exploring mutexes the oracle rdbms retrial spinlocks
- *MySQL*. MySQL Product Archives
- *MySQL*. InnoDB Startup Options and System Variables
- *何登成*. InnoDB Buffer Pool管理调研
- *何登成*. InnoDB Crash Recovery & Rollback Segment源码实现分析

# 下期预告

- InnoDB如何记录Undo/Redo？
- InnoDB的Rollback Segment是如何实现的？
- InnoDB的事务如何实现Roll Forward/Roll Backward？
- InnoDB的Purge操作，主要有哪些功能？
- InnoDB的完整Crash Recovery流程是什么？
- InnoDB的Mini Transaction又有何功能？

# Any Question？

- 联系方式
  - 微博：@何_登成
  - 邮箱：he.dengcheng@gmail.com
  - 博客：hedengcheng.com

谢谢大家！