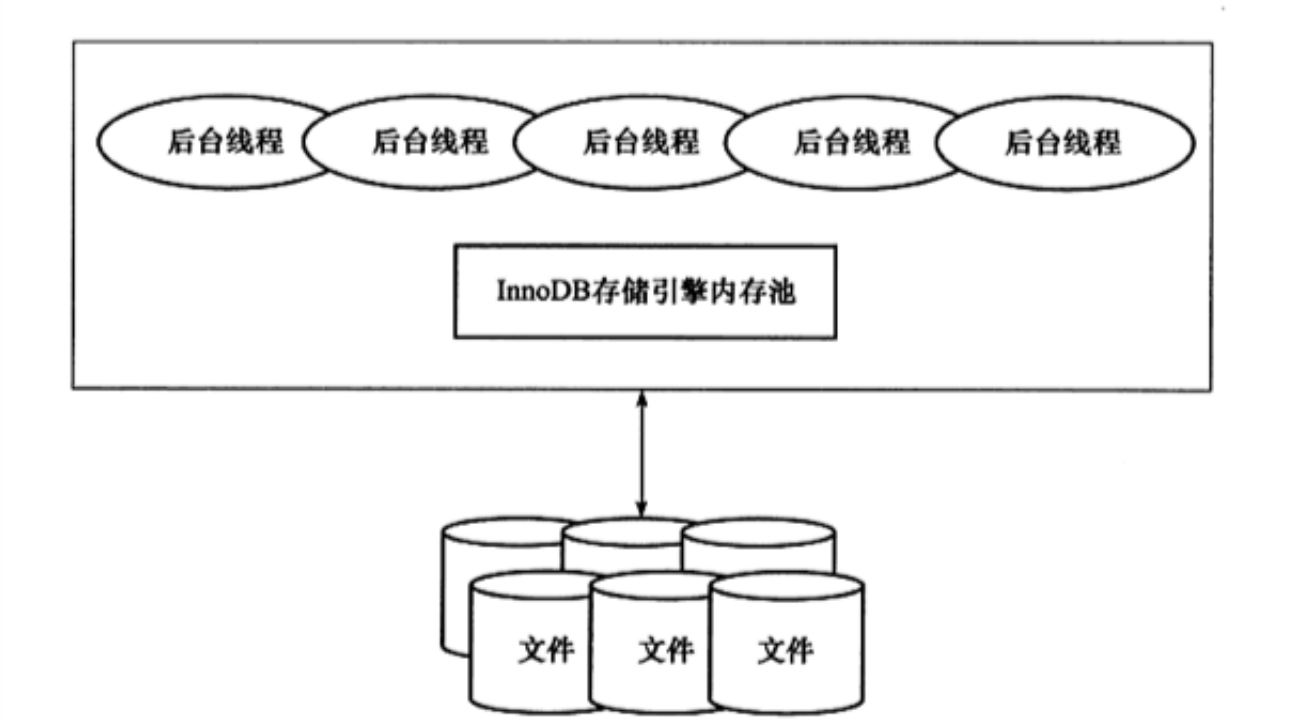
2.1 InnoDB存储引擎概述
2.3 InnoDB体系架构
2.3.1 后台线程
1）Master Thread
2）IO Thread
3）Purge Thread
4）Page Cleaner Thread
2.3.3 内存
1）缓冲池
2）LRU List、Free List、 Flush List
压缩页
脏页
3）重做日志缓冲（redo log buffer）
4） 额外的内存池
2.4 CheckPoint 技术
2.4.1 背景
2.4.2 目的
2.4.3 Checkponit 的类型
2.4.3.1 Sharp Checkpoint
2.4.3.2 Fuzzy Checkpoint
Master Thread Checkponit
FLUSH_LRU_LIST Checkpoint
Async/Sync Flush Checkpoint
Dirty Pagetoo mach Checkpoint
2.5 Master Thread 工作方式
1.0.x版本之前的Master Thread
loop
每一秒的操作
每十秒的操作
后台循环——backgroup loop
刷新循环——flush loop
暂停循环——suspend loop
1.2.x版本之前的Master Thread
innodb_io_capacity参数的引入
innodb_max_dirty_pages_pct默认值的修改
innodb_adaptive_flushing参数的引入
innodb_purge_batch_size参数的引入
1.2.x版本的Master Thread
2.6 InnoDB 关键特性
2.6.1 插入缓冲
1）Insert Buffer
2） Change Buffer
3）Insert Buffer的内部实现
4）merge Insert buffer
2.6.2 两次写
2.6.3 自适应哈希索引
2.6.4 异步 IO

> 本章主要是innodb的体系结构、关键特性以及innodb启动关闭时一些参数的影响

2.1 InnoDB存储引擎概述

InnoDB存储引擎是第一个完整支持ACID事务的MySQL存储引擎，其特点是行锁涉及、支持MVCC、支持外键、提供一致性非锁定读，同时被设计用来最有效地利用以及使用内存和CPU。

2.3 InnoDB体系架构



2.3.1 后台线程

InnoDB存储引擎是多线程的模型，因此其后台有多个不同的后台线程，负责处理不同的任务。

1）Master Thread

作为一个核心的后台线程，主要负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲（INSERT BUFFER）、UNDO页的回收等。

2）IO Thread

在InnoDB存储引擎汇总，使用了大量的AIO来处理IO请求，这样可以极大提高数据库的性能。而IO Thread的工作主要是负责这些IO请求的回调。共有四类IO Thread，分别是：write、read、insert buffer 以及 log IO Thread。

3）Purge Thread

事务被提交后，其所使用的undolog可能不再需要，因此需要PurgeThread来回收已使用并分配的undo页。目前purge以及可以独立到单独的线程中国进行，从而减轻MasterThread的工作，从而提高CPU的使用率以及提升存储引擎的性能。

4）Page Cleaner Thread

将对脏页的刷新操作独立于MasterThread，减轻MasterThread的工作以及对于用户查询线程的阻塞，进一步提高InnoDB存储引擎的性能。

2.3.3 内存

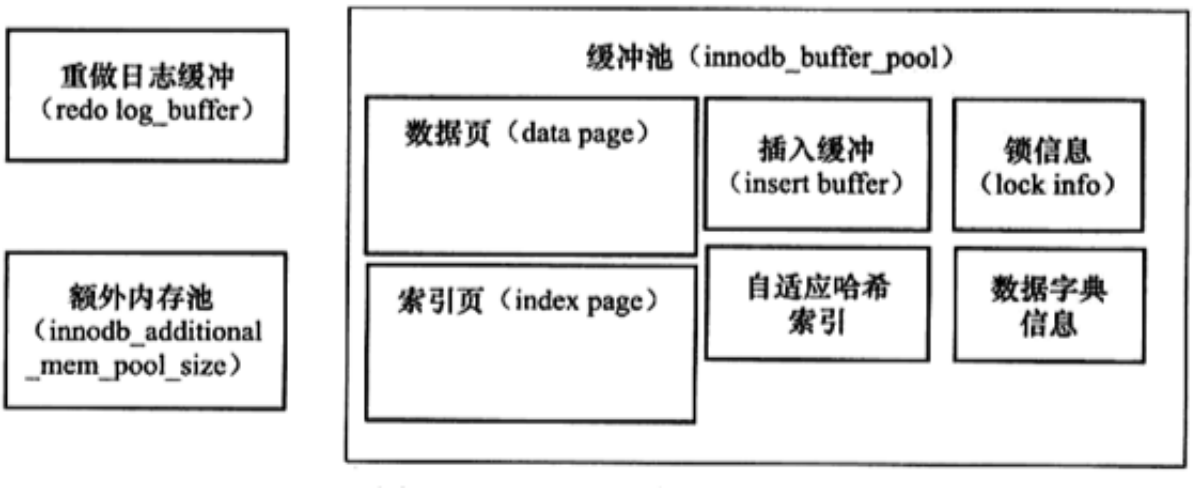
1）缓冲池

InnoDB存储引擎是基于磁盘存储的，并将其中给的记录按照页的方式进行管理。因此可将其视为基于磁盘的数据库系统。在数据库系统中，由于CPU速度与磁盘速度之间的鸿沟，基于磁盘的数据库系统通常使用缓冲池技术提高数据库的整体性能。

缓冲池是一块内存区域，通过内存的速度来弥补较慢的磁盘速度。

对于数据库页的读操作，会将内放在缓冲池中，称之为将页FIX在缓冲池中，下次读取相同页时，可以在缓冲池中命中。

对于数据库页的修改操作，先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。刷新机制成为Checkpoint。



如上图，缓冲池中缓存的数据页类型有：索引页、数据页、插入缓存、自适应哈希索引、InnoDB存储的锁信息、数据字典信息等。

2）LRU List、Free List、 Flush List

InnoDB存储引擎如何对内存区域进行管理?

数据库中的缓冲池通过LRU算法进行管理。即最频繁使用的页在LRU列表的前段，最少使用的页在LRU列表的尾端。当缓冲池不能存放读取到的页时，将首先释放LRU列表中尾端的页。

InnoDB中，缓冲池的页的默认大小为16KB，同样使用LRU算法对缓冲池进行管理。不同的是，InnoDB对传统LRU进行了优化。在LRU列表中加入了midpoint位置，对于新读取到的页，放入LRU列表的midpoint位置。这种算法在InnoDB中被称为midpoint insertion strategy。默认在LRU列表长度的5/8处，由参数**innodb_old_blocks_pct**%控制。midpont之之后的列表成为old列表，之前的称为new列表，new列表中都是最为活跃的热点数据。

为什么进行优化呢？因为如果直接将读取到的页放入LRU的首部，那么某些SQL操作可能回事缓冲池中的页被刷新出，从而影响缓冲池的效率。这类操作通常为索引或者数据的扫描操作，会访问表中许多页甚至全部的页，而这些也有只是当次查询中需要的，而不是活跃的热点数据。这会导致热点数据页会从LRU列表中被移除，而下次需要读取该页时，需要再次访问磁盘。

同时为了解决这个问题，引入了**innodb_old_blocks_time**，用于表示也在mid位置等待多久才会被加入LRU列表的热端。

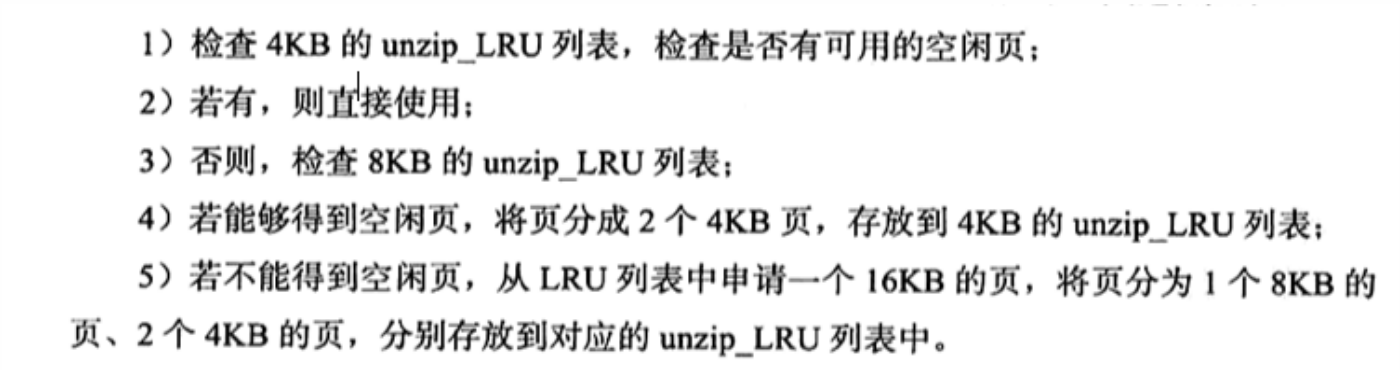
LRU列表用来管理已读取的页，数据库刚启动时，页都在Free列表中。读取页时，先查找Free列表是否有空闲页，有则将页从中删除，加入LRU列表，否则淘汰LRU列表中末尾的页。

当页从LRU的old加入new时，成为 page made young，而由于时间没到无法将页移动到new时，称为 page not made young。

自适应哈希索引、Lock信息、InsertBuffer等页不需要LRU算法维护。

压缩页

InnoDB支持压缩页，将页从16k压缩到1KB、2KB、4KB、8KB。每个都有单独的列表，申请4KB页时的逻辑为：



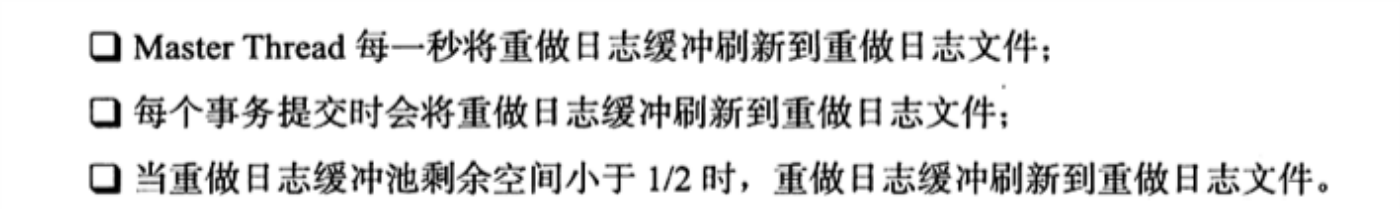
脏页

LRU列表中也被修改后，被称为脏页，也就是缓冲池中数据与磁盘产生了不一致。会通过Checkpoint机制将脏页刷新回磁盘，而Flush列表中的页即为脏页列表，但脏页存在于Flush列表与LRU列表，分别用来管理页的刷新与缓冲池中页的可用性。

3）重做日志缓冲（redo log buffer）

InnoDB先将重做日志信息放入缓冲区，然后按照一定频率刷新到重做日志文件。重做日志缓冲大小默认为8MB，通过innodb_log_buffer_size控制。

有三种情况会进行刷新：



4） 额外的内存池

InnoDB中，对内存的管理是通过内存堆的方式进行的，但在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。如缓冲池中的帧缓冲以及对应的缓冲控制对象，这些对象记录了一下诸如LRU、锁、等待等信息。

2.4 CheckPoint 技术

2.4.1 背景

前面了解到，一条DML语句会先变更缓冲池中的数据，导致缓冲池中数据与磁盘不一致，需要将新版本的页刷新到磁盘上。

事务数据库中使用Write Ahead Log 策略，即事务提交时，先写重做日志，再修改页。从而避免以下两种情况：

1）每次变更都刷新到磁盘上，会导致性能极度下降。

2）在从缓冲池将页的新版本刷新到磁盘时发生宕机，会丢失数据

理论上如果：缓冲池可以缓存数据库中所有数据；重做日志可以无限增大；接受宕机后恢复的时间较长。那么完全可以通过重做日志可以恢复整个数据库系统中数据到宕机时刻，

2.4.2 目的

但上述情况以目前的条件无法做到，因此，出现了Checkpoint（检查点）技术 。它解决了一下三个问题

1）缩短数据库的恢复时间；

2）缓冲池不够用时，将脏页刷新到磁盘；

3）重做日志不可用时，刷新脏页；

发生宕机时，只需要恢复Checkpoint之后的页即可，之前的页已经刷新回磁盘了。

缓冲池不够用时，会根据LRU算法溢出最少使用的页，此页如果为脏页，会强制执行Checkpoint，将脏页刷新回数据库。

重做日志为了降低成本、方便管理，一般都是环形使用的。其可以重用部分都是已经被刷新到磁盘中的，且宕机后恢复时也无需进行恢复。如果在记录重做日志时，需要覆盖未被刷新的部分，需要强制产生Checkpoint，刷新到磁盘。

2.4.3 Checkponit 的类型

Checkpoint 所做的事情是将缓冲池中的脏页刷新到磁盘。我们需要了解，每次刷新多少页到磁盘（What），每次从哪里取脏页（Where），以及什么时间触发Checkpoint（When）。

InnoDB中有两类Checkpoint：Sharp Checkpoint、Fuzzy Checkpoint。

2.4.3.1 Sharp Checkpoint

Sharp Checkpoint 发生在数据库关闭时，将所有脏页刷新到磁盘，这是默认的方式，对应的参数为inn0db_fast_shutdown = 1。

运行时也可以使用Sharp Checkpoint，但会影响数据库的可用性，因此在InnoDB内部使用时，只刷新部分脏页。

2.4.3.2 Fuzzy Checkpoint

InnoDB中会发生以下四种Fuzzy Checkpoint，分别为：

1）Master Thread Checkponit

2）FLUSH_LRU_LIST Checkpoint

3）Async/Sync Flush Checkpoint

4）Dirty Pagetoo mach Checkpoint

Master Thread Checkponit

以异步的形式，按照一定频率刷新一定比例的页到磁盘中。

FLUSH_LRU_LIST Checkpoint

InnoDB中需要保证有一定数量的空闲页可以使用，对应参数为innodb_lru_scan_depth，默认为1024。如果没有这么多空闲页，会冲LRU列表尾端移除对应数量的页，这些页中如果存在脏页，那么需要进行Checkpoint。

Async/Sync Flush Checkpoint

在重做日志文件不可用的情况下，需要强制将一些页刷新回磁盘，而脏页是从脏页列表中选取的。已经写入重做日志的LSN（log sequence number）记为redo_lsn，已经刷新回磁盘最新页的LSN记为checkpoint_lsn。此时：

checkponit_age = redo_lsn - checkpoint_lsn；

async_water_mark = 75% * total_redo_log_file_size;

sync_water_mark = 90% * total_redo_log_file_size;

刷新规则如下：

当 checkponit_age< async_water_mark时，无需刷新；

当async_water_mark<checkponit_age<sync_water_mark时，触发Async Flush，使刷新后满足checkponit_age< async_water_mark；

当 checkponit_age >sync_water_mark 时，触发Sync Flush，使刷新后满足checkponit_age< async_water_mark；这种情况一般不会出现，除非设置的重做日志文件太小，并且在进行类型LOAD DATA 的 BULK INSERT 操作。

Dirty Pagetoo mach Checkpoint

脏页数量太多时，会触发强制Checkpoint，为了确保缓存池中有足够的可用页，参数为 innodb_max_dirty_page_pct，值为75时，便是脏页数量不可操作75%，否则进行强制刷新。

2.5 Master Thread 工作方式

Master Thread拥有最高的线程级别。其内部由多个循环（loop）组成：

1. 主循环（loop）
2. 后台循环（backgroup loop）
3. 刷新循环（flush loop）
4. 暂停循环（suspend loop）

Master Thread会根据数据库的运行状态在这四个循环中进行切换

1.0.x版本之前的Master Thread

loop

loop被称为主循环，因为大多数的操作都是在这个循环中，其中有两大部分的操作——每一秒的操作和每十秒的操作

每一秒的操作

每一秒的操作包括：

- 日志缓冲刷新到磁盘，即使这个事务还没有提交（总是）
- 合并插入缓冲（可能）
- 至多刷新100个缓冲池中的脏页到磁盘（可能）
- 如果当前用户没有活动，则切换到backgroup loop

操作解释：

即使某个事务还没有提交，InnoDB存储引擎仍然会每秒将日志缓冲的内容刷新到磁盘的日志文件中。这就是为什么再大的事务提交的时间也是很短的。

合并插入缓冲并不是每秒都会发生的。InnoDB存储引擎会判断当前一秒内发生的IO次数是否小于5次，如果小于5次，InnoDB认为当前IO的压力很小，可以执行插入缓冲操作。

刷新100个脏页也不是每秒都会发生的。InnoDB会判断挡墙缓冲池中脏页的比例，是否超过了配置文件中的innodb_max_dirty_pages_pct这个参数，如果超过了这个阈值，就会认为需要做磁盘同步的操作。

每十秒的操作

每十秒的操作包括：

- 刷新100个脏页到磁盘（可能）
- 合并至多5个插入缓冲（总是）
- 将日志缓冲刷新到磁盘（总是）
- 删除无用的undo页（总是）
- 刷新100个或者10个脏页到磁盘（总是）

操作解释：

InnoDB会先判断过去十秒钟磁盘IO的操作是否小于200次，如果是则认为此事有足够的磁盘IO能力，因此将100个脏页进行刷新。

接着进行合并插入缓冲，但是这个和之前的一秒操作不同，这次的操作是总会执行的！

然后进行日志刷新操作，也是总会执行。

之后对进一步执行full purge操作，即删除不用的Undo页。对表进行update，delete操作时，原先的行并不是直接被删除，而是被标记为删除，是因为一致性读的关系，所以需要保留版本信息。在full purge过程中，InnoDB会判断当前事务系统中已被删除的行是否可以删除，如果可以，每次最多尝试收回20个undo页
最后InnoDB会判断缓冲池中脏页的比例，如果有超过70%的脏页就刷新100个脏页到磁盘，如果脏页比例小于70%，则只需要刷新10个的脏页到磁盘

后台循环——backgroup loop

若当前没有用户活动（数据库空闲时）或数据库关闭时，就会切换到这个循环，其中有以下操作：

1. 删除无用的undo页（总是）
2. 合并20个插入缓冲（总是）
3. 跳回到主循环（总是）
4. 不断刷新100个页直到符合条件（可能，也有可能跳转到flush loop中完成）

刷新循环——flush loop

刚刚上面提到了，backgroup loop中的刷新操作可能会跳到flush loop中完成，如果该loop中也没有事情可做了，那么就会切换到suspend loop，将Master Thread挂起，等待事件的发生

暂停循环——suspend loop

如果flush loop中也没有事情可做了，那么就会切换到suspend loop，将Master Thread挂起，等待事件的发生。

Master thread 伪代码

```
oid master_thread() {
    loop:
    for(int i=0; i<10; i++) {
        thread_sleep(1)    // sleep 1秒
        do log buffer flush to dish

        if (last_one_second_ios < 5% innodb_io_capacity) {
            do merget 5% innodb_io_capacity insert buffer
        }

        if (buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct) {  // 如果缓冲池中的脏页比例大于innodb_max_dirty_pages_pct(默认是75时)
            do buffer pool flush 100% innodb_io_capacity dirty page  // 刷新全部脏页到磁盘
        } else if (enable adaptive flush) {    // 如果开户了自适应刷新
            do buffer pool flush desired amount dirty page // 通过判断产生redo log的速度决定最合适的刷新脏页的数量
        }

        if (no user activetuy) {
            goto background loop
        }
    }

    if (last_ten_second_ios < innodb_io_capacity) {  // 如果过去10内磁盘IO次数小于设置的innodb_io_capacity的值（默认是200）
        do buffer pool flush 100%  innodb_io_capacity dirty page
    }

    do merge 5% innodb_io_capacity insert buffer  // 合并插入缓冲是innodb_io_capacity的5%（10）（总是）
    do log buffer flush to dish
    do flush purge

    if (buf_get_modified_ratio_pct > 70%) {
        do buffer pool flush 100% innodb_io_capacity dirty page
    } else {
        do buffer pool flush 10% innodb_io_capacity dirty page
    }
    goto loop

    backgroud loop:   // 后台循环
    do full purge     // 删除无用的undo页 （总是）
    do merger 5% innodb_io_capacity insert buffer  // 合并插入缓冲是innodb_io_capacity的5%（10）（总是）
    if not idle:      // 如果不空闲，就跳回主循环，如果空闲就跳入flush loop
    goto loop:    // 跳到主循环
    else:
        goto flush loop
    flush loop:  // 刷新循环
```

```
    do buf_get_modified_ratio_pct pool flush 100% innodb_io_capacity dirty page // 刷新200个脏页到磁盘
    if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct ) // 如果缓冲池中的脏页比例大于innodb_max_dirty_pages_pct的值（默认75%）
        goto flush loop            // 跳到刷新循环，不断刷新脏页，直到符合条件
        goto suspend loop          // 完成刷新脏页的任务后，跳入suspend loop
    suspend loop:
    suspend_thread()               //master线程挂起，等待事件发生
    waiting event
    goto loop;
}
```

1.2.x版本之前的Master Thread

innodb_io_capacity参数的引入

问题：

在InnoDB存储引擎1.0.x版本之前的Master Thread中，我们会发现，InnoDB对IO其实是有限制的，当缓冲池向磁盘刷新的时候都做了一定的硬编码。但是如今固态磁盘快速发展，这种硬编码就很大程度上限制了innoDB对磁盘IO的性能
什么意思呢？举个例子：刚刚我们谈到InnoDB每次最多刷新100个脏页和合并20个插入缓冲，如果在写入密集的程序中，每秒可能会产生大于100个的脏页，大于20个的插入缓冲，这时候Master Thread似乎会“忙不过来”，或者说即使磁盘有一秒钟处理超过100个脏页和超过20个插入缓冲的能力，由于硬编码的原因，导致也只能每次刷新100和20个。这就是弊端！
解决：

所以在1.2.x之前，引入如下参数：
innodb_io_capacity——表示磁盘IO的吞吐量

1. 在合并插入缓冲时，合并插入缓冲的数量为innodb_io_capacity的5%
2. 在刷新脏页时，刷新脏页的数量为innodb_io_capacity

innodb_max_dirty_pages_pct默认值的修改

问题：

之前innodb_max_dirty_pages_pct默认值是90，意味着脏页占缓冲池的90%，但是这个值太大了，超过90%才会刷新100个脏页。如果有很大的内存或者数据库服务的压力很大，这时候刷新脏页的速度反而会降低。
解决：

在经过不断的调试之后，将该值的默认值调成了75

innodb_adaptive_flushing参数的引入

该值影响每秒刷新脏页的数量
原来的刷新规则是：大于innodb_max_dirty_pages_pct才会刷新100个脏页，小于这个值不刷新。
解决：

随着innodb_adaptive_flushing参数的引入，InnoDB会通过一个函数来判断需要刷新脏页最适合数量，这个函数的原理是通过判断产生重做日志的速度来决定最适合刷新的脏页数量
所以现在当脏页比例小于innodb_max_dirty_pages_pct时也会刷新一定数量的脏页

innodb_purge_batch_size参数的引入

问题：

之前每次进行full purge操作时，每次都是最多回收20个undo页
解决：

innodb_purge_batch_size决定回收undo页的数量，可以动态调整，默认值为20

1.2.x版本的Master Thread

```
if (InnoDB is idle) {
    srv_master_do_idle_tasks();    // 每10秒操作
} else {
    srv_master_do_active_tasks();    // 每秒操作
}
```

对于刷新页的操作，从Master 线程单独分离到一个Page Clean Thread，减轻主线程负担，提高并发性。

2.6 InnoDB 关键特性

插入缓冲（Insert Buffer）

两次写（Double Write）

自适应哈希索引（Adaptive Hash Index）

异步IO（Async IO）

刷新邻接页（Flush Neighbor Page）

2.6.1 插入缓冲

1）Insert Buffer

在进行插入操作时，数据页的存放是按照主键a 进行顺序存放的，但对于非聚集索引叶子节点的插入不再是顺序的，这时需要离散的访问非聚集索引页，导致插入性能的下降。某些情况下，辅助索引的插入依然是比较顺序的，如时间顺序的索引。

为了提高非聚集索引插入的性能，InnoDB设计了Insert Buffer，对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是优先插入到缓冲池中非聚集索引页中，如果不在，则先放入Insert Buffer中，然后再以一定频率进行Insert Buffer与辅助索引页子节点的合并操作，这通常能够将多个插入合并到一个操作中。

Insert Buffer 需要满足一下两个条件：

1. 索引是辅助索引
2. 索引不是唯一的（插入缓冲中，并不会会去判断插入记录的唯一性）

2） Change Buffer

InnoDB对DML操作均进行了缓存：

INSERT： Insert Buffer

DELETE： Delete Buffer

UPDATE： Purge Buffer

均只适用非唯一的辅助索引。

对一条记录进行UPDATE操作可能分为两个过程：

1. 将记录标记为已删除；对应Delete buffer
2. 真正将记录删除。对应Purge buffer

通过参数进行控制启用Change buffer： inserts, deletes,purges,changes,all,none。

changes 表示启用inserts 和 deletes

all表示启用所有

none表示都不启用

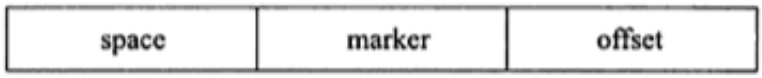
3）Insert Buffer的内部实现

Insert Buffer 具体是什么，内部如何实现?

InsertBuffer 是一颗B+树，全局唯一，负责对所有表的所有辅助索引进行Insert Buffer，默认存放在共享表空间（ibdatal）。

当试图通过ibd文件恢复表中数据时，会导致Check TABLE，就是因为表中的辅助索引数据还在Insert Buffer中，需要在ibd文件恢复后，进行 PEPAIR TABLE操作，重建表中所有辅助索引。

Insert Buffer B+树的非叶子节点存放的是查询的 键值（search key）共9个字节：



space，四个字节，表示插入记录所在表的表空间id。

marker，一个字节，兼容老版本Insert Buffer。

offset，四个字节，表示页所在的偏移量。

Insert Buffer 叶子节点存放的数据：

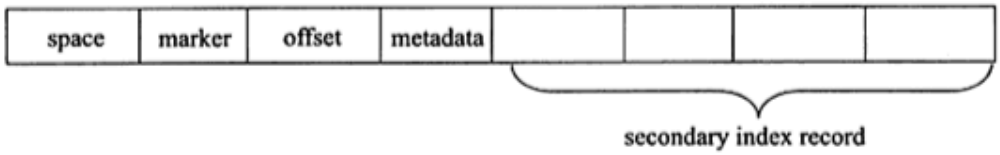


图 2-4　Insert Buffer 叶子节点中的记录

space、marker、page_no 字段和之前非叶节点中的含义相同，一共占用 9 字节。第 4 个字段 metadata 占用 4 字节，其存储的内容如表 2-2 所示。

表 2-2　metadata 字段存储的内容

名　　称	字　　节
IBUF_REC_OFFSET_COUNT	2
IBUF_REC_OFFSET_TYPE	1
IBUF_REC_OFFSET_FLAGS	1

IBUF_REC_OFFSET_COUNT，用来排序每个记录进入Insert buffer的顺序，但通过顺序回放才能得到记录的正确值。

从第五列开始，就是实际插入记录的字段，所以叶子节点有额外13个字节的开销。

为了确保每次merge Insert buffer 页的操作成功，通过Insert Buffer Bitmap来标记每个辅助索引页的可用空间

在每256个区，也就是16284个辅助索引页中，都会有一个Insert buffer bitmap 来追踪这些辅助索引页，其位于16384个页中的第二个页中，也就是在段中第一个区的第二页。

每个辅助索引页标识都在insert buffer bitmap中占用四位，组成情况如下：

名　　称	大小（bit）	说　　明
IBUF_BITMAP_FREE	2	表示该辅助索引页中的可用空间数量，可取值为： ❑ 0 表示无可用剩余空间 ❑ 1 表示剩余空间大于 1/32 页（512 字节） ❑ 2 表示剩余空间大于 1/16 页 ❑ 3 表示剩余空间大于 1/8 页
IBUF_BITMAP_BUFFERED	1	1 表示该辅助索引页有记录被缓存在 Insert Buffer B+ 树中
IBUF_BITMAP_IBUF	1	1 表示该页为 Insert Buffer B+ 树的索引页

4）merge Insert buffer

Insert buffer 中的记录何时合并（merge）到真正的辅助索引上呢?

有以下三种情况：

1. 辅助索引页被读取到缓冲池中
2. Insert Buffer Bitmap 页追踪到该辅助索引页以无可用空间
3. Master Thread

对于第一种情况，正常执行select 操作时，需要通过insert buffer bitmap ，判断该辅助索引页是够有记录存放与Insert buffer B+树中。若有，则将B+树中的该页的记录插入到辅助索引页中。

对于第二种情况，Insert buffer bitmap 在插入辅助索引记录时检测到对于辅助索引页空间小于1/32，则会强制进行一个合并操作，将对应的页的记录以及待插入的记录都合并到辅助索引页中。

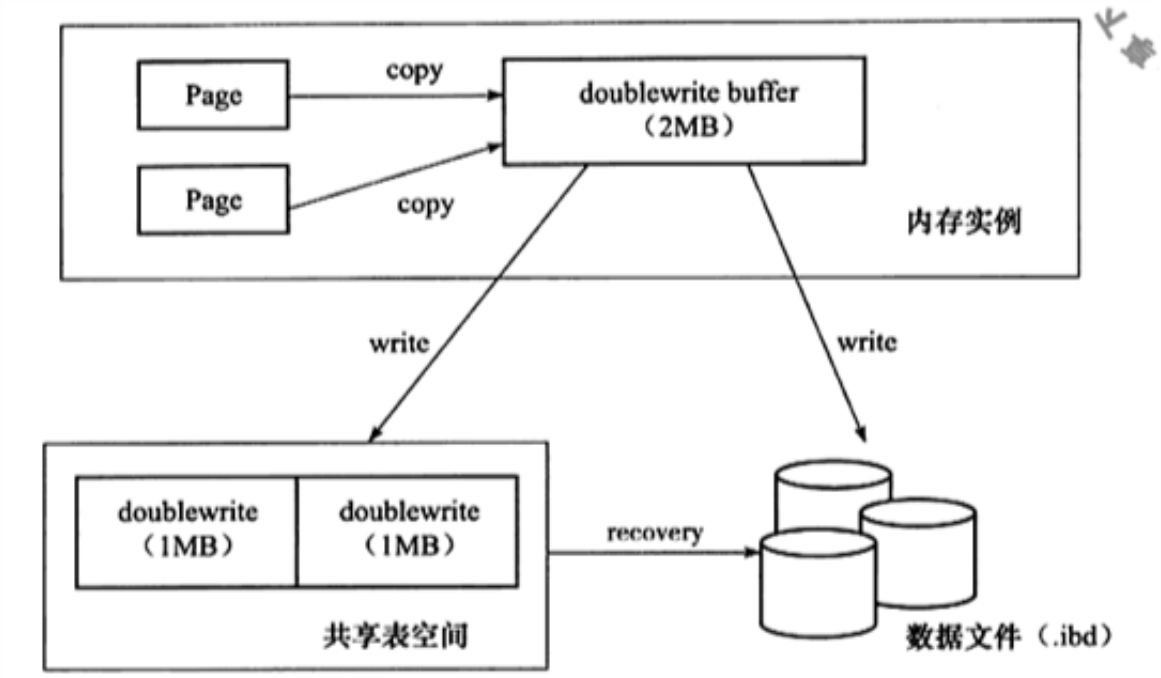
Master Thread 每秒或者每十秒会进行一次Merge Insert Buffer的操作。通过srv_innodb_io_capactiy的百分比决定真正要合并的页数。合并时innodb会随机选择树上的一个页，读取该页中的space以及之后所需要数量的页。

2.6.2 两次写

> insert bufer 带来了innodb存储引擎性能上的提升，doublewrite带来了数据页的可靠性。

innodb在写入某个页到表中时，如果发生了宕机，会存在数据丢失的情况，这是无法用重做日志恢复的，因为重做日志是对页的物理操作，如果这个页本身发生了损坏，是无法对其进行重做的。为此需要一个页的副本，在应用重做日志之前，先恢复这个页，再进行重做。

doublewrite由两部分构成：内存中的doublewrite buffer，大小为2MB，另一部分是物理磁盘中共享表空间中连续的128个页，即2个区，大小为2MB。对缓冲池脏页刷新时，通过memcpy将脏页复制到内存中的doublewrite buffer，然后通过doublewrite buffer再分两次，每次1MB顺序的写入共享表空间的物理磁盘，然后通过fsync函数，同步磁盘。



2.6.3 自适应哈希索引

InnoDB存储引擎会监控表上各个索引页的查询，如果观察到建立哈希索引可以带来速度提升，那么就会建立哈希索引，称之为自适应哈希索引（AHI）。AHI通过缓冲池的B+数页构造而来，建立的速度很快，而且不需要对整张表构建哈希索引。

有三个要求：

1. 对页的连续访问模式是一样的，如对于联合索引（a,b），访问模式是指查询的条件一样：where a = XXX;where a = XXX and b = XXX。如果两类查询条件交替执行，不会构造AHI。
2. 以该模式访问了100次
3. 页通过该模式访问了N次，其中N = 页中的记录数/16。

注意：哈希索引只能用于等值的查询，对于其他诸如范围查询，无法使用。

2.6.4 异步 IO

innodb存储引擎通过异步io（AIO）的方式处理磁盘操作。

当一条查询语句需要扫描多个索引页，也就是进行多个io操作时，会一次发出多个io请求，然后等待所有操作的完成。

同时AIO的优势是可以进行IO Merge操作，当访问的多个页相邻时，可以直接进行合并，一次性读取这些页。

2.6.5 刷新邻接页

工作原理：当刷新一个脏页时，innodb会检测该页所在区的所有页，如果是脏页，会一起刷新。对于机械硬盘有着显著优势，但对于固态硬盘，我们可以通过innodb_flush_neighbors，控制是否开启该特性。

存在一个问题：如果页不怎么脏，被刷新后，页又很快变成了脏页，这种情况应该怎么办?

2.7 启动、关闭与恢复

innodb引擎的启动与关闭，是Mysql实例在启动过程中对Innodb存储引擎的处理过程。

innodb关闭时，参数innodb_fast_shutdown影响着innodb的行为，默认值为1：

- 0 表示在 MySQL 数据库关闭时，InnoDB 需要完成所有的 full purge 和 merge insert buffer，并且将所有的脏页刷新回磁盘。这需要一些时间，有时甚至需要几个小时来完成。如果在进行 InnoDB 升级时，必须将这个参数调为 0，然后再关闭数据库。
- 1 是参数 innodb_fast_shutdown 的默认值，表示不需要完成上述的 full purge 和 merge insert buffer 操作，但是在缓冲池中的一些数据脏页还是会刷新回磁盘。
- 2 表示不完成 full purge 和 merge insert buffer 操作，也不将缓冲池中的数据脏页写回磁盘，而是将日志都写入日志文件。这样不会有任何事务的丢失，但是下次 MySQL 数据库启动时，会进行恢复操作（recovery）。

当innodb 发生宕机时，后续启动之后，就要进行数据恢复，innodb_force_recovery，会影响整个innodb存储引擎恢复的状况，默认值为0，表示需要恢复是，进行所有恢复操作。但有时候用户知道怎么进行恢复，可以通过参数，避免恢复时间过长，而是通过用户自行恢复，如可以把表删除，从备份中重新导入数据到表，这些操作会远远快于回滚。

- 1(SRV_FORCE_IGNORE_CORRUPT)：忽略检查到的 corrupt 页。
- 2(SRV_FORCE_NO_BACKGROUND)：阻止 Master Thread 线程的运行，如 Master Thread 线程需要进行 full purge 操作，而这会导致 crash。
- 3(SRV_FORCE_NO_TRX_UNDO)：不进行事务的回滚操作。

- 4(SRV_FORCE_NO_IBUF_MERGE)：不进行插入缓冲的合并操作。
- 5(SRV_FORCE_NO_UNDO_LOG_SCAN)：不查看撤销日志（Undo Log），InnoDB 存储引擎会将未提交的事务视为已提交。
- 6(SRV_FORCE_NO_LOG_REDO)：不进行前滚的操作。