# 背景

Innodb**为了解决磁盘上磁盘速度和CPU速度不一致的问题**，在操作磁盘上的数据时，先将数据加载至内存中，在内存中对数据页进行操作。

MySQL在启动的时候，会向内存申请一块**连续**的空间，这块空间名为Bufffer Pool，也就是缓冲池，**默认情况下Buffer Pool只有128M**。

# 概述

参考：

page cleaner算法：<http://mysql.taobao.org/monthly/2020/12/03/>

## 传统LRU

LRU算法最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：

1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

## InnoDB LRU

在InnoDB存储引擎中，缓冲池的大小默认16KB，使用LRU算法对缓冲池进行管理。稍有不同的是InnoDB存储引擎对传统的LRU算法做了一些优化，在InnoDB存储引擎中，LRU列表中还加入了midpoint位置。新读取到的页，虽然是新访问的页，但并不是放入到 LRU列表的首部，而是放入LRU列表的midpoint位置。这个算法在InnoDB存储引擎下称为midpoint insertion strategy。

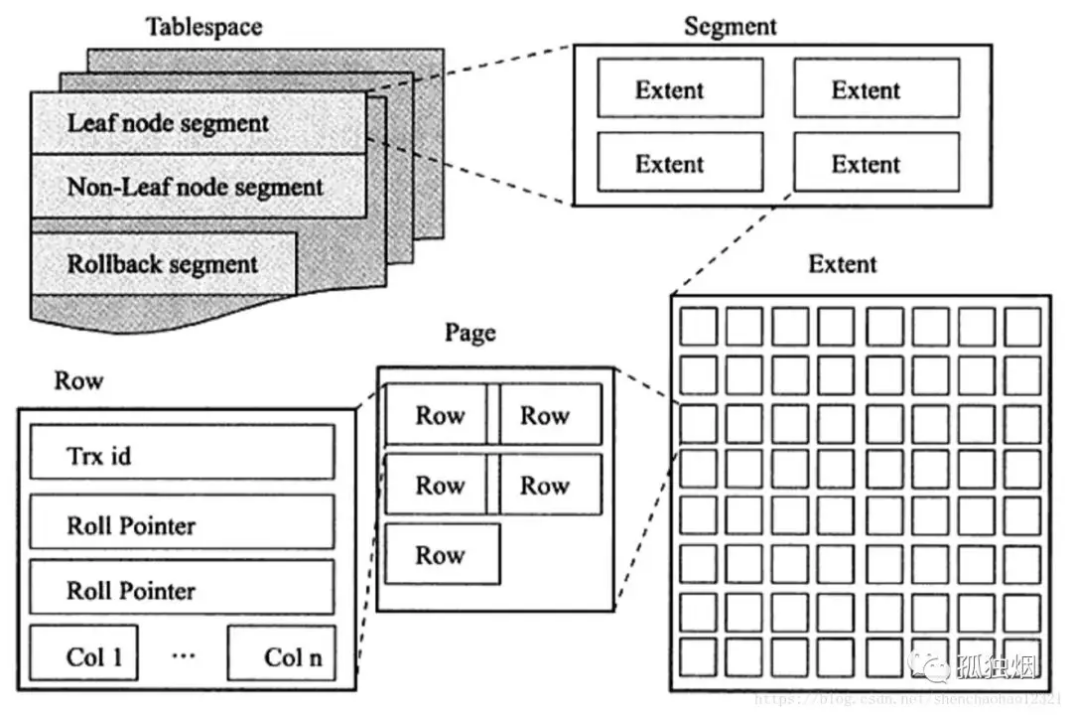
为什么不像传统的LRU算法那样将最新的数据放到链表头部，然后刷新链表尾部的数据呢？

**原因一：**

假设存在一张表tb1，没有任何索引，且单表数据量在千万级别，如果需要执行select \* from tb1操作，那么由于没有索引可用，则会全表扫描，按照传统LRU算法，这些数据页都会通过Buffer Pool加载，然后依次加入到LRU链表头部，由于Buffer Pool大小受限，所以必然会存在内存淘汰，即会清空之前其他查询语句留下来的高频访问的数据页。这样的最终结果就是，Buffer Pool中缓存的全部是低频的数据页，缓存命中率就会大大降低。为了避免这种情况，才引入了midpoint，防止全部高频数据页都丢失。

**原因二：**

InnoDB的表逻辑结构如下图所示：



从InnoDB存储引擎的逻辑存储结构看，所有数据都被逻辑地存放在一个空间中，称之为表空间( tablespace)。表空间又由段(segment)、区( extent)、页(page)组成，页在一些文档中有时也称为块( block)，数据页(page)是放在区(extent)里的。

InnoDB的预读包括线性预读和随机预读。

**线性预读：**当一个区中有连续56个页面(56为默认值)被加载到BufferPool中，会将这个区中的所有页面都加载到BufferPool中。其实挺合理的，毕竟一个区最多才64个页。

**随机预读：**当一个区中随机13个页面(13为默认值)被加载到BufferPool中，会将这个区中所有页面都加载到BufferPool中。随机预读默认是关闭，由变量innodb\_random\_read\_ahead控制。

需要注意的是，预读机制会预读一些额外的页到到BufferPool中。  
 那么，如果这些预读页并不是高频的页呢？  
 如果这些页并不是高频的页，按照上面的算法，也会被加入LRU链表，就会将链表末端一些高频的数据页给淘汰掉，从而导致命中率下降。

**总结：**

为什么不采用朴素LRU算法，因为如果直接读取到页放到LRU首部，那么某些SQL操作可能会是缓冲池中的页被刷出，从而影响缓冲池的效率。常见索引或数据的扫描操作，这类操作需要访问表中的许多页，甚至全部页，而这些页通常来说又仅仅在这次查询操作中需要，并不是活跃的热点数据，如果页被放入LRU列表首部，那么非常可能将所需要的热点数据从LRU列表中删除，而在下一次需要读取该页数据时，InnoDB存储引擎需要再次访问磁盘。

为了解决上面的两个缺点，Innodb将这个链表分为两个部分，也就是所谓的old区和young区。  
 young区在链表的头部，存放经常被访问的数据页，可以理解为热数据。old区在链表的尾部，存放不经常被访问的数据页，可以理解为冷数据。这两个部分的交汇处称为midpoint。

# 配置

## innodb\_old\_blocks\_pct

默认配置下，midpoint位置在LRU列表长度的5/8处。midpoint位置可由参数innodb\_old\_blocks\_pct控制：

mysql> show variables like 'innodb\_old\_blocks\_pct' G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

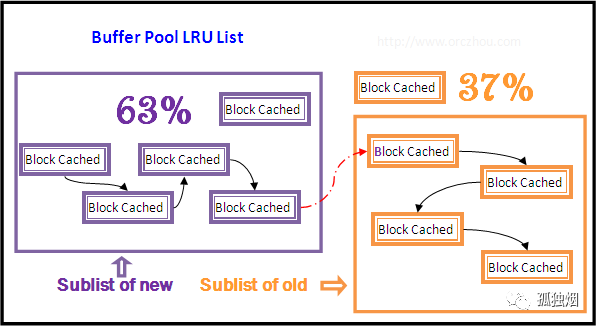
Variable\_name: innodb\_old\_blocks\_pct

Value: 37

row in set (0.01 sec)

从上面的例子中可以看出，参数innodb\_old\_blocks\_pct的默认值是37，表示新读取的页插入到LRU列表尾端的37%的位置（差不多3/8）。

在InnoDB存储引擎中，把midpoint之后的列表称为old列表，之前的列表称为new列表。可以简单的理解为new列表中的页都是最为活跃的热点数据。



注：一般生产的机器，内存比较大。我们会把innodb\_old\_blocks\_pct值调低，防止热数据被刷出内存。

## innodb\_old\_blocks\_time

数据何时在old区，何时进入young区？  
 数据页第一次被加载进BufferPool时在old区头部。当这个数据页在old区，再次被访问到，会做如下判断

如果这个数据页在LRU链表中old区存在的时间超过了1秒，就把它移动到young区。这个存在时间由innodb\_old\_blocks\_time控制

为了解决这个问题，InnoDB存储引擎引入了另一个参数来进一步管理LRU列表，这个参数是innodb\_old\_blocks\_time，用于表示页读取到mid位置后需要等待多久才会被加入到LRU列表的热端。因此当需要执行上述所说的SQL操作时，可以通过下面方式尽量使LRU列表中热点数据不被刷出

mysql> set global innodb\_old\_blocks\_time =1000;

Query OK, 0 rows affected (0.01 sec)

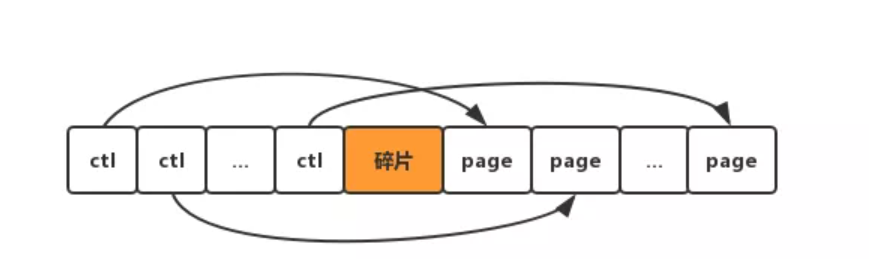
如果用户预估自己的热点数据不知63%，可以执行下面语句设置

mysql> set global innodb\_old\_blocks\_pct =20;

Query OK, 0 rows affected (0.01 sec)

# 原理

缓冲池：



如图所示，有三部分组成:

ctl: 俗称控制体，里头有一个指针指向缓存页，还有一个成员变量存储着所谓的一些所谓的控制信息，例如该页所属的表空间编号、页号

page:缓存页，就是磁盘上的页加载进Bufffer Pool后的结构体

碎片：每个控制体都有一个缓存页。最后内存中会有一点点的空间不足以容纳一对控制体和缓存页，于是碎片就诞生的！

## 概述

如果系统一直在进行数据库的增删改操作，数据库内部的基本流程就是：

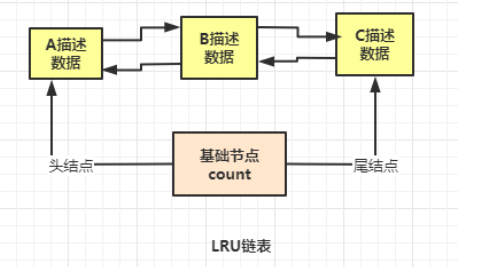
我们还拿redis类做类比，以便更好的帮助大家明白其原理。Flush的作用其实类似redis的key设置的过期时间，所以一般情况下，redis内存不会不够使用，但是总有特殊的情况，问题往往就是在这种极端和边边角角的情况下产生的。

如果redis的内存不够使用了，是不是自己还有一定的淘汰策略？最基本的准则就是淘汰掉不经常使用到的key。Buffer Pool也类似，它也会有内存不够使用的情况，它是通过LRU链表来维护的。LRU即Least Recently Uesd（最近最少使用）。

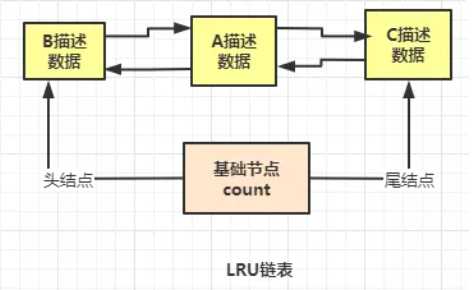
MySql会把最近使用最少的缓存页数据刷入到磁盘去，那MySql如何判断出LRU数据的呢？为此MySql专门设计了LUR链表，还引入了另一个概念：缓存命中率

**# 缓存命中率** 可以理解为缓存被使用到的频率，举个例子来说：现在有两个缓存页，在100次请求中A缓存页被命中了20次，B缓存页被命中了2次，很显然A缓存页的命中率更高，这也就意味着A在未来还会被使用到的可能性比较大，而B就会被MySQL认为基本不会被使用到；

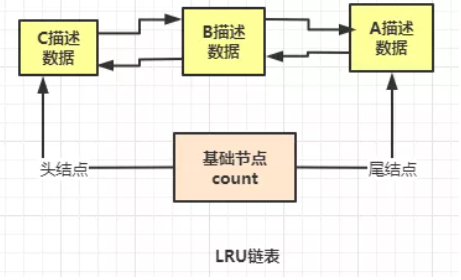
说到这里，那LRU究竟是怎么工作的。假设MySQL在将数据加载到缓存池的时候，他会将被加载进来的缓存页按照被加载进来的顺序插入到LRU链表的头部（就是链表的头插法），假设MySQL现在先后分别加载A、B、C数据页到缓存页A、B、C中，然后LRU的链表大致是这样子的。



现在又来了一个请求，假设查询到的数据是已经被缓存在缓存页B中，这时候 MySQL就会将B缓存页对应的描述信息插入到LRU链表的头部，如下图：

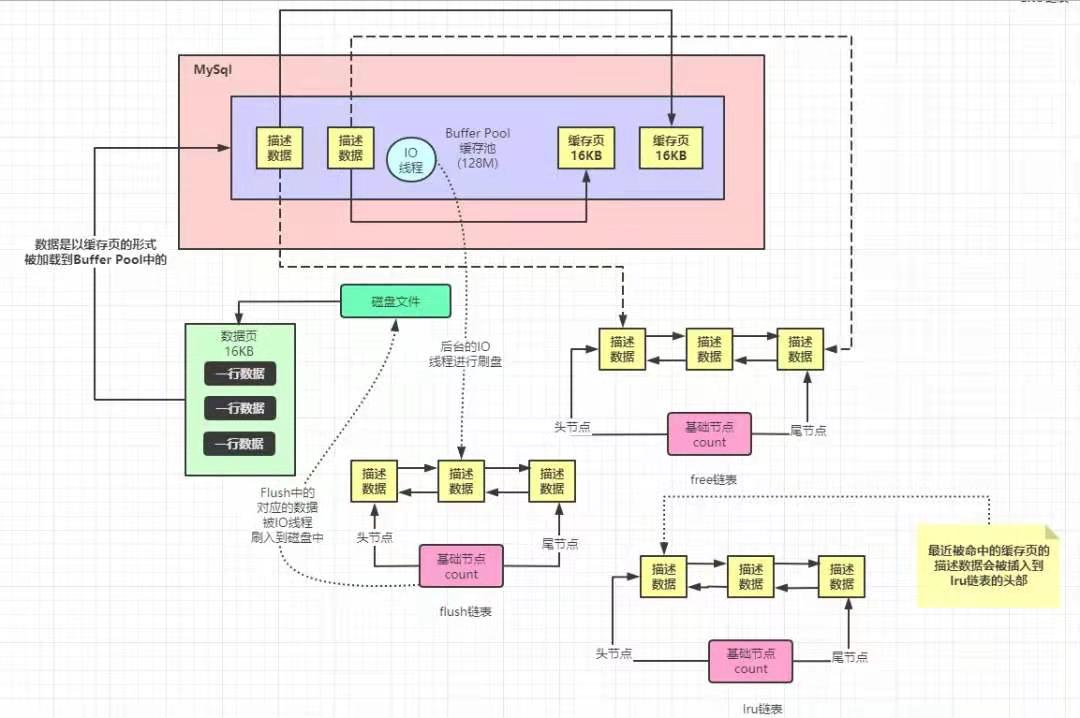


然后又来了一个请求，数据是已经被缓存在了缓存页C中，然后LRU会变成这样子：



说到底，每次查询数据的时候如果数据已经在缓存页中，那么就会将该缓存页对应的描述信息放到LRU链表的头部，如果不在缓存页中，就去磁盘中查找，如果查找到了，就将其加载到缓存中，并将该数据对应的缓存页的描述信息插入到LRU链表的头部。也就是说最近使用的缓存页都会排在前面，而排在后面的说明是不经常被使用到的。

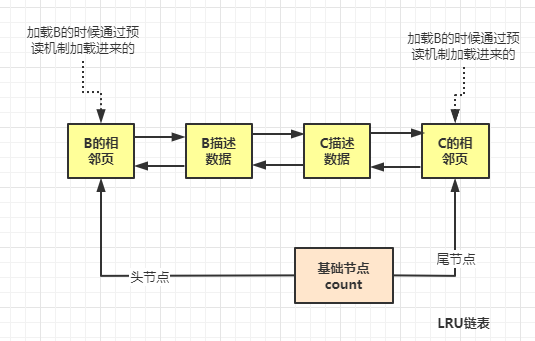
最后，如果Buffer Pool不够使用了，那么 MySQL就会将LRU链表中的尾节点刷入到磁盘中，用来给Buffer Pool腾出内存空间。来个整体的流程图给大家看下



## 问题

这里的麻烦指的是就是 MySQL本身的预读机制带来的问题

# 预读机制  
 MySQL 在从磁盘加载数据的的时候，会将数据页的相邻的其他的数据页也加载到缓存中。  
 # MySQL 为什么要这么做   
 因为根据经验和习惯，一般查询数据的时候往往还会查询该数据相邻前后的一些数据，有人可能会反问：一个数据页上面不是就会存在该条数据相邻的数据吗？这可不一定，某条数据可能很大，也可能这条数据是在数据页在头部，也可能是在数据页的尾部，所以 MySQL 为了提高效率，会将某个数据页的相邻的数据页也加载到缓存池中。



上图能够看到B的相邻也被加载到了C描述数据的前面，而实际上C的命中率比B的相邻页高多了，这就是LRU本身带来的问题。

# 哪些情况会触发预读机制  
 1、有一个参数是innodb\_read\_ahead\_threshold，他的默认值是56，意思就是如果顺序的访问了一个区里的多个数据页，访问的数据页的数量超过了这个阈值，此时就会触发预读机制，把下一个相邻区中的所有数据页都加载到缓存里去（这种就是：线性预读）  
 2、如果Buffer Pool里缓存了一个区里的13个连续的数据页，而且这些数据页都是比较频繁会被访问的，此时就会直接触发预读机制，把这个区里的其他的数据页都加载到缓存里去（这种就是：随机预读）随机预读是通过：innodb\_random\_read\_ahead来控制的，默认是OFF即关闭的（MySQL 5.5已经基本飞起该功能，应为他会带来不必要的麻烦，这里也不推荐大家开启，说出来的目的是让大家了解下有这么个东西）

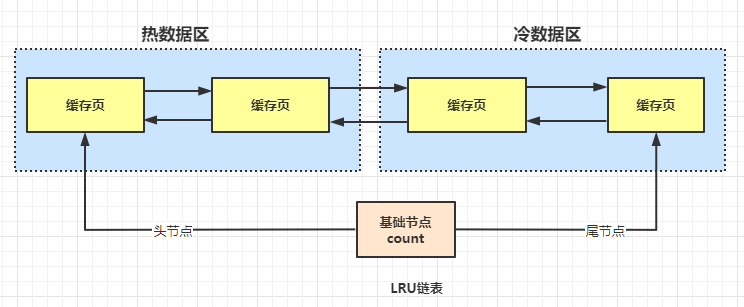
还有一种情况是SELECT \* FROM students 这种直接全表扫描的，会直接加载表中的所有的数据到缓存中，这些数据基本是加载的时候查询一次，后面就基本使用不到了，但是加载这么多数据到链表的头部就将其他的经常命中的缓存页直接全挤到后面去了。

以上种种迹象表明，预读机制带来的问题还是蛮大的，既然这么大，那 MySQL为什么还要进入预读机制呢，说到底还是为了提高效率，\*\*一种新的技术的引进，往往带来新的挑战，下面我们就一起来看下 MySQL是如何解决预加载所带来的麻烦的。

## 改进

基于冷热数据分离的LRU链表

所谓的冷热分离，就是将LRU链表分成两部分，一部分是经常被使用到的热数据，另一部分是被加载进来但是很少使用的冷数据。通过参数innodb\_old\_blocks\_pct 参数控制的，默认为37，也就是37% 。用图表示大致如下：



数据在从磁盘被加载到缓存池的时候，首先是会被放在冷数据区的头部，然后在一定时间之后，如果再次访问了这个数据，那么这个数据所在的缓存页对应描述数据就会被放转移到热数据区链表的头部。

那为什么说是在一定的时间之后呢，假设某条数据刚被加载到缓存池中，然后紧接着又被访问了一次，这个时候假设就将其转移到热数据区链表的头部，但是以后就再也不会被使用了，这样子是不是就还是会存在之前的问题呢？

所以 MySQL通过innodb\_old\_blocks\_time来设置数据被加载到缓存池后的多少时间之后再次被访问，才会将该数据转移到热数据区链表的头部，该参数默认是1000单位为：毫秒，也就是1秒之后，如果该数据又被访问了，那么这个时候才会将该数据从LRU链表的冷数据区转移到热数据区。

现在再回头看下上面的问题

# 通过预加载（加载相邻数据页）进来的数据  
 这个时候就很好理解了，反正数据会被放在LRU链表的冷数据区的（注意：这里说的放在链表中的数据都是指的是<缓存页中的数据所对应的描述数据>），当在指定时候之后，如果某些缓存页被访问了那么就将该缓存页的描述数据放到热数据区链表的头部  
 # 全表扫描加载进来的数据页  
 和上面一样，数据都是先在冷数据区，然后在一定时间之后，再次被访问到的数据页才会转移到热数据区的链表的头结点，所以这也就很好的解决了全表扫描所带来的问题

再来思考下 Buffer Pool 内存不够的问题

# Buffer Pool 内存空间不够使用了怎么办？也就是说没有足够使用的空闲的缓存页了。  
 这个问题在这个时候就显得非常简单了，直接将链表冷数据区的尾节点的描述数据多对应的缓存页刷到磁盘即可。

但是这样子还不是足够完美，为什么这么说，刚刚我们一直在讨论的是冷数据区的数据被访问，然后在一定规则之下会被加载到热数据链表的头部，但是现在某个请求需要访问的数据就在热数据区，那是不是直接把该数据所在的缓存页对应的描述数据转移到热数据区链表头部呢？

很显然不是这样子的，因为热数据区的数据本身就是会被频繁访问的，这样子如果每次访问都去移动链表，势必造成性能的下降（影响再小极端情况下也可能会不可控），所以 MySQL针对热数据区的数据的转移也有相关的规则。

该规则就是：如果被访问的数据所在的缓存页在热数据区的前25%，那么该缓存页对应的描述数据是不会被转移到热数据链表的头部的，只有当被访问的缓存页对应的描述数据在热数据区链表的后75%，该缓存页的描述数据才会被转移到热数据链表的头部。

举个例子来说，假设热数据区有100个缓存页（这里的缓存页还是指的是缓存页对应的描述数据，再强调下，链表中存放的是缓存页的描述数据，为了方便有时候会直接说缓存页。希望朋友们注意），当被访问的缓存页在前25个的时候，热数据区的链表是不会有变化的，当被访问的缓存页在26~100（也就是数据在热数据区链表的后75%里面）的时候，这个时候被访问的缓存页才会被转移到链表的头部。

到此为止， MySQL对于LUR 链表的优化就堪称完美了。是不是看到这里瞬间感觉很多东西都明朗了，好了，对于 LRU 链表我们就讨论到这里了。

## Free List

LRU列表用来管理已经读取的数据，但当数据库刚启动时，LRU列表时空的，即没有任何页。这时页都存放在free列表中。当需要从缓冲池中分页时，首先从free表中查找是否有可用的空闲页，若有则将该页从free列表中删除，放入到LRU列表中。否则，根据LRU算法，淘汰LRU列表末尾的页，将该内存空间分配给新的页。当页从LRU列表的old部分加入到new部分时，称此时发生的操作为page made young，而因为innodb\_old\_blocks\_time 的设置而导致页没有从old部分移到new部分的操作称为page not made young。可以通过命令 show engine innodb status 来观察LRU列表及Free列表的使用情况和运行状态。

mysql> show engine innodb status G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Type: InnoDB

Name:

Status:

=====================================

2019-09-18 16:53:18 0x7f5fa5f3b700 INNODB MONITOR OUTPUT

=====================================

Per second averages calculated from the last 10 seconds

----------------------

BUFFER POOL AND MEMORY

----------------------

Total large memory allocated 137428992

Dictionary memory allocated 687641

Buffer pool size 8191

Free buffers 7063

Database pages 1099

Old database pages 238

Modified db pages 0

Pending reads 0

Pending writes: LRU 0, flush list 0, single page 0

Pages made young 0, not young 0

0.00 youngs/s, 0.00 non-youngs/s

Pages read 842, created 257, written 7490

0.00 reads/s, 0.00 creates/s, 0.00 writes/s

Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not 0 / 1000

1 row in set (0.00 sec)

可见当前Buffer pool size共有8191页，即8191\*16K缓冲池。Free buffers表示当前free 列表中页的数量，Database pages代表LRU列表中页的数量。可能的情况是Free buffers和 Database pages的数量只和不等于Buffer pool size，因为缓冲池中的页还可能会被分配给自适应哈希，lock 信息，Insert Buffer 等页，而这部分不需要LRU 算法维护，因此不在LRU 列表中。

Pages made young显示了LRU列表中页移动到前端的次数，因此该服务在运行阶段没有改变innodb\_old\_blocks\_time的值，因此not young为0，youngs/s，non-youngs/s表示每秒这两类操作的次数。Buffer pool hit rate命中率为100%，说明缓冲池运行良好，通常该值不应该小于95%，若小于此值，用户需要考试是否是全表扫描引起的 LRU列表被污染的问题。

从 InnoDB 1.2 版本开始，还可以通过表INNODB\_BUFFER\_POOL\_STATS来观察缓冲池的运行状态，还可以通过INNODB\_BUFFER\_PAGE\_LRU来观察每个LRU列表中每个页的具体信息。

InnoDB存储引擎从1.0.x版本开始支持压缩页的功能，即将原本16K的页压缩为1KB, 2KB, 4KB和8KB，由于页的大小发生了变化，LRU列表也有了些许改变，对于非16KB的页，是通过unzip\_LRU列表进行管理的。通过命令show engine innodb statsu可观察如下：

mysql> show engine innodb status G;

......

Pages read 842, created 257, written 7522

0.00 reads/s, 0.00 creates/s, 0.00 writes/s

No buffer pool page gets since the last printout

Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s

LRU len: 1099, unzip\_LRU len: 0

I/O sum[0]:cur[0], unzip sum[0]:cur[0]

可以看到LRU列表中一共 1099页，而 unzip\_LRU 为0，注意，这里LRU列表页包含了unzip\_LRU中的页。

对于压缩页的列表，每个表的压缩比率可能各不相同。那unzip\_LRU是怎样从缓冲池中分配内存呢？

首先，在unzip\_LRU列表中对不同压缩页大小的页进行分别管理，其次，通过伙伴算法进行内存的分配。例如对需要从缓冲池中申请页为 4KB 的大小，过程如下：

1、检查4KB 的unzip\_LRU列表，检查是否有可用的空闲页

2、若有，直接使用

3、否则，检查8KB 的unzip\_LRU列表

4、若能够得到空闲页，将分页成2个4KB页，存放4KB 的unzip\_LRU列表

5、若不能得到空闲页，从LRU列表中申请一个16KB的页，将页氛围一个8KB 的页，2个4KB的页，分别存放对应的unzip\_LRU列表中。

同样可以通过information\_schema架构下的表INNODB\_BUFFER\_PAGE\_LRU来观察unzip\_LRU列表中的页。

mysql> select table\_name, space, page\_number, compressed\_size from innodb\_buffer\_page\_lru where compressed\_size <>0;

Empty set (0.01 sec)

## Flush List

在LRU列表中的页被修改后，称该页为脏页（dirty page），即缓冲池中页和磁盘上的数据产生了不一致。这时数据库会通过CHECKPOINT机制将脏页刷新回磁盘，而Flush列表中的页即为脏页列表。需要注意的是，脏页即存在于LRU列表中，页存在于Flush列表中。**LRU列表用来管理缓冲池中页的可用性，Flush列表用来管理将页面刷新回磁盘，二者互不影响**。

同LRU列表一样，Flush列表也可以通过命令show engine innodb status来查看，前面的例子中Modified db pages 0就显示了脏页数据为0。information\_schema架构下并没有类似 INNODB\_BUFFER\_PAGE\_LRU的表来显示脏页的数量及脏页的类型，但正如前面所述的那样，脏页同样存在于LRU列表中，故用户可以通过元数据表INNODB\_BUFFER\_PAGE\_LRU 来查看，唯一不同的是需要加入oldest\_modification > 0的查询条件：

1mysql> select table\_name, space, page\_number, page\_type from INNODB\_BUFFER\_PAGE\_LRU where oldest\_modification > 0;

2Empty set (0.08 sec)

如上，查询没有脏页。