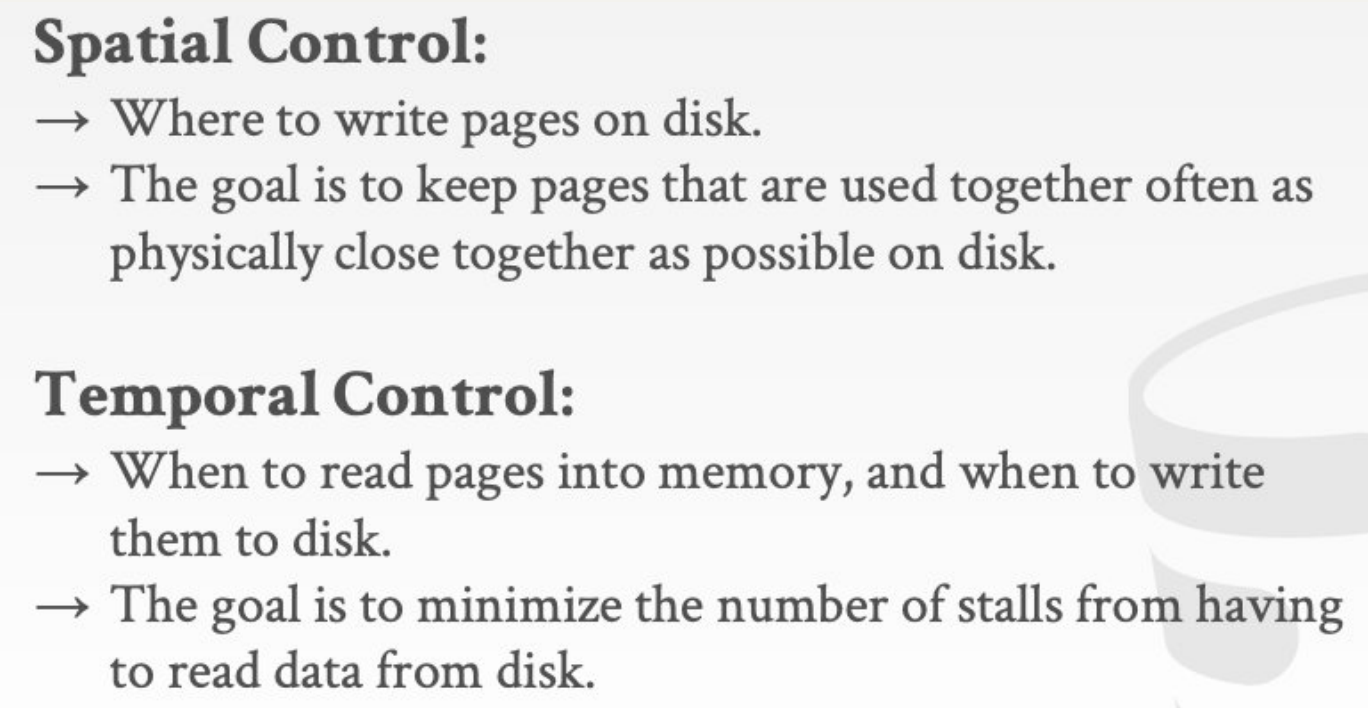
缓存池存在于内存中，用于缓存磁盘中的数据库文件的部分页

缓存池有两大作用，

1. 一是缓存数据库中的数据，因为用户有可能再次访问该页，这样的话就可以直接去内存中查询，提升效率
2. 二是缓存用户的更新，用户对数据库的内容的修改也是先被存储在内存的缓存池里的，我们需要先把修改缓存在内存中，然后集中将修改写回硬盘或者采用其他策略，而不是每次一修改就立刻写回硬盘，这样会大大提升效率

对缓存池有两种控制，分为空间和时间上的控制



1. 空间上的控制是说，对于那些用户经常需要一起使用的数据页，我们把它们在缓存池里集中起来，就像它们在磁盘上的布局一样连续，这样我们到时候直接去缓存池里就能找到所有需要访问的数据
2. 时间上的控制是说，我们要决定何时从磁盘上读数据页，何时往磁盘中写回，我们的目标是最小化用户等待磁盘I/O的时间，

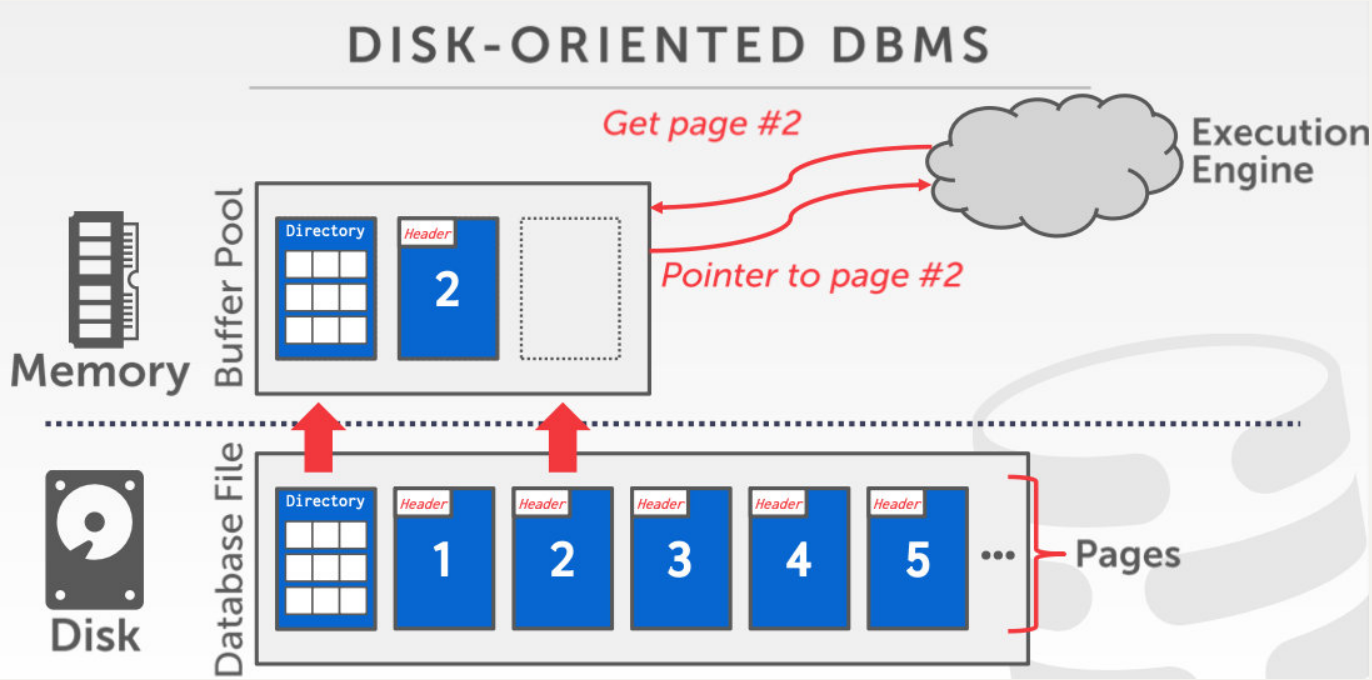
   e.g.

   - 用户修改了一些数据，我们不可以立刻将修改写回硬盘，因为立刻写回的话需要让用户额外更待很长时间，直到完成了对硬盘的写回，SQL语句才

执行完成

- 用户要读数据时，我们不要让用户现场等待从硬盘将数据读入内存，最佳的处理方式是用户发出SQL语句之前，想要读的数据已经从硬盘读到了内存中，这样的预测是确有手段可以做到的

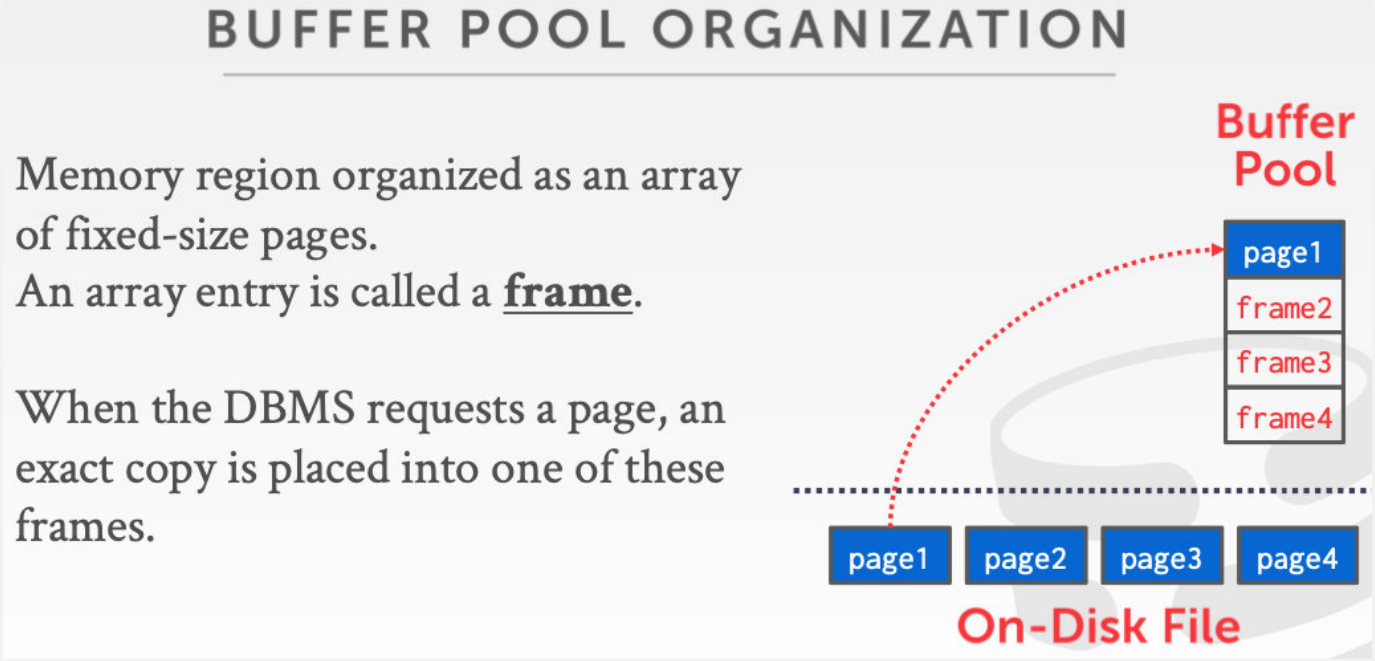
接下来我们先overview一下DBMS读取数据的大致流程，如下图，如果我们想读硬盘中数据库文件的2号页



DBMS的执行引擎会先去缓存池中找，看看2号页是否已经被缓存在其中，如果没有的话，就从硬盘中把索引页读到缓存池里，通过索引页得知我们想要的2号页在硬盘中的位置，然后去硬盘中再把2号页读入缓存池，之后把缓存池中2号页的指针返回给执行引擎，接下来执行引擎再去完成用户想做的增删改查之类的工作

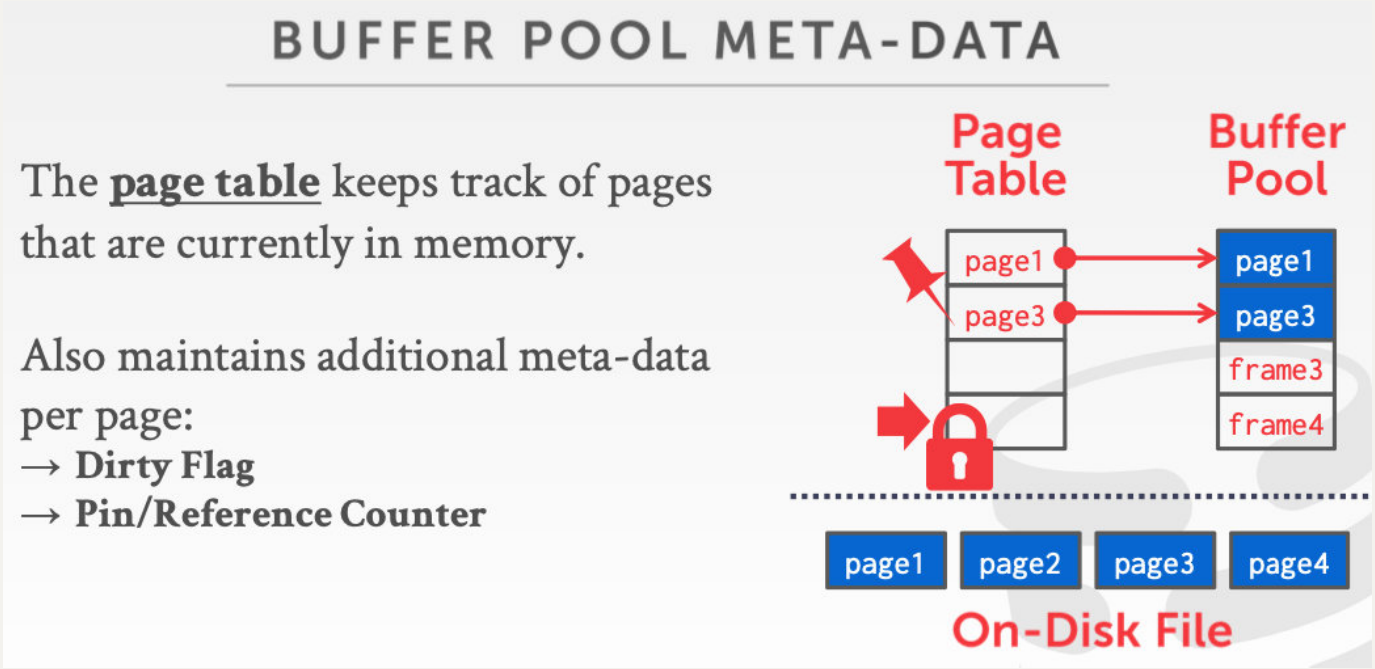
Buffer Pool Manager

缓存池由数组形式的连续存放的一些帧构成，每个帧用于缓存数据库文件的一个页

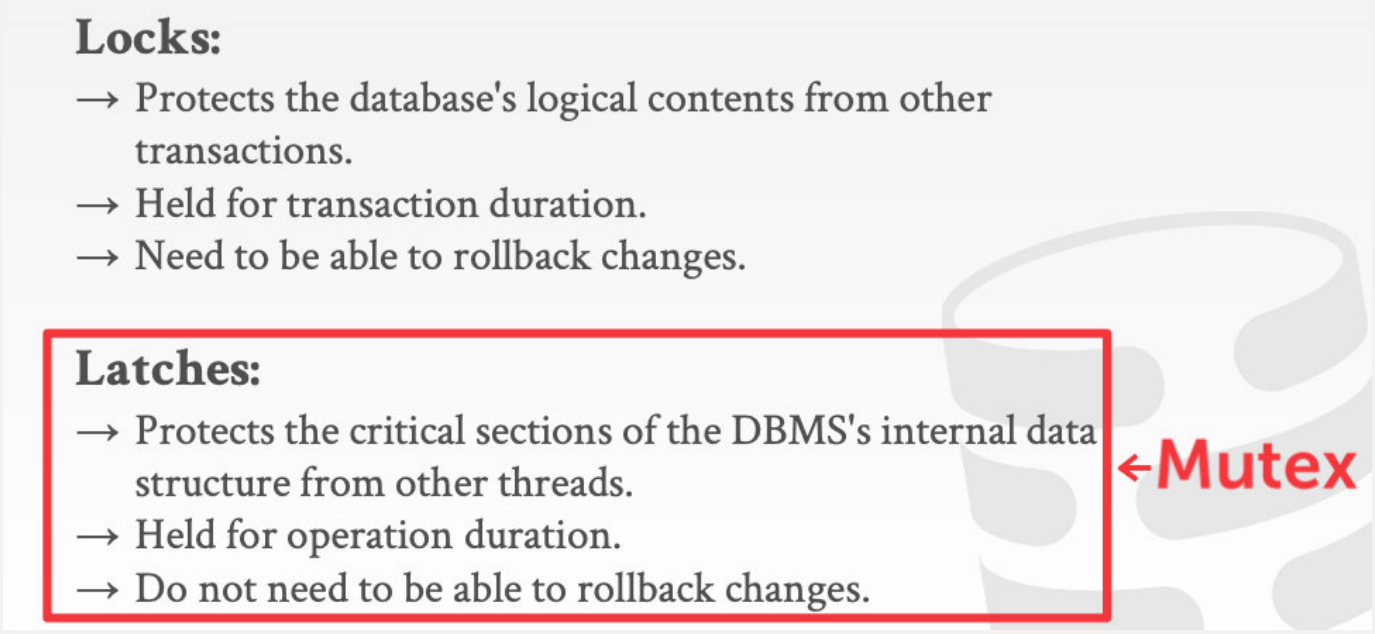


那么我们如何才能知道缓存池里缓存了数据库文件的哪些Page呢?

可以采用下图的Page Table从而使用索引的机制，并且我们可以像page 3那样pin住page table的某个entry，告诉DBMS，这个对应的页之后还会用到，不可以把它从缓存池里拿走，也可以锁住page table的某个entry，这相当于告诉DBMS，这个entry槽位过一会儿要被占用，会有新的页被存入缓存池



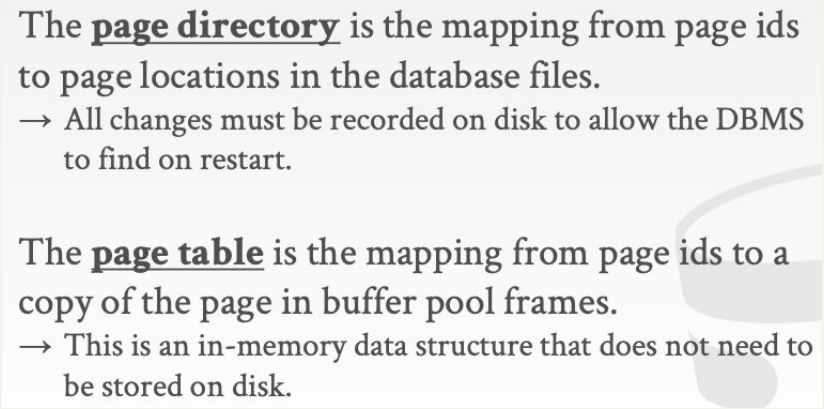
在数据库中，有关于锁，有如下的辨析:



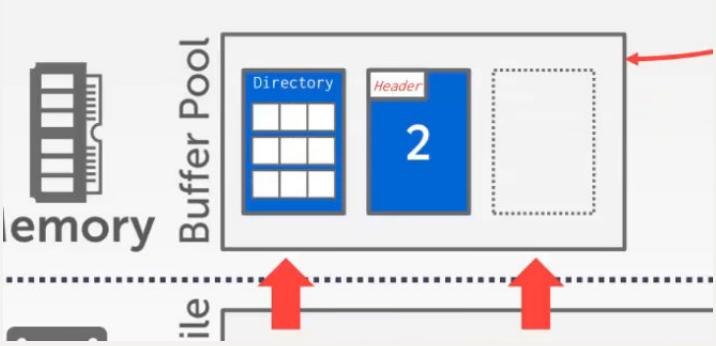
当我们想锁住数据库的某个表，锁住的时候其他线程都不能读这个表，这种锁叫lock，而latch是具体的、底层的锁，我们为了lock住某个表，需要加特定的latch，比如说我们把和这个表相关的所有的数据页都加上锁，我们给这些数据页上的锁就叫latch

总结一下就是说，Lock是逻辑上的抽象的概念，是high-level的，Latch是实现这个抽象的概念的具体手段，又名Mutex

数据库中page table和page direcory也有如下的概念辨析：

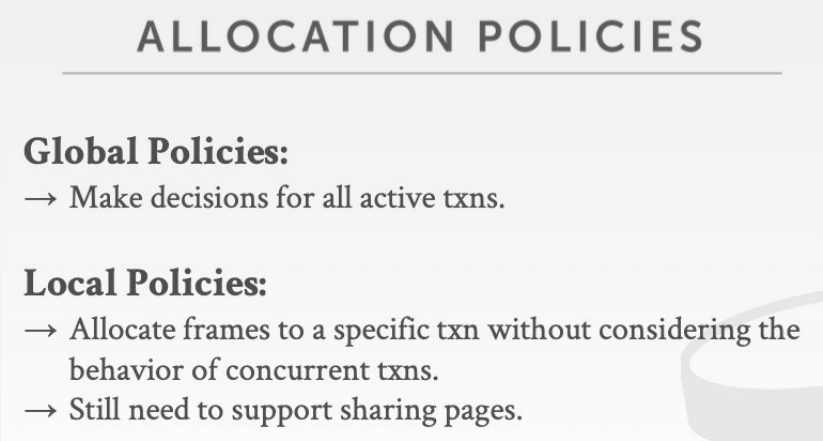


前面overview DBMS读数据的流程时提到的索引页就是page directory，如下图所示：



它是磁盘上的一个目录，记录着“x号页在磁盘的哪里”，而page table索引的是那些被缓存的页都被放到了缓存池的哪个位置

将从磁盘中缓存的页放置在缓存池的哪个位置，需要一些策略，分为如下两种：



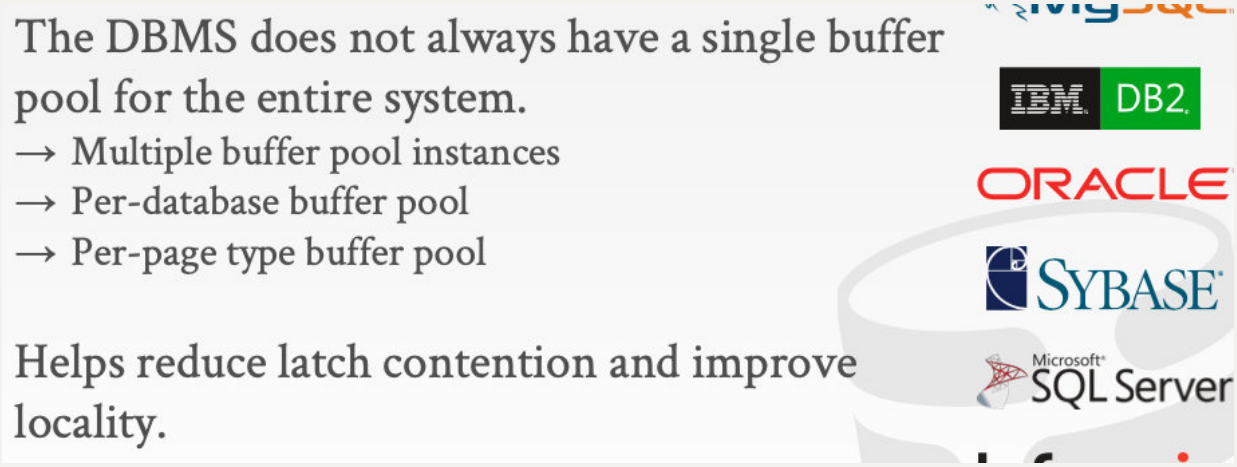
- 全局策略：数据库同时有很多用户在使用，在并发的运行很多事务，我们可以统一的安排将这些的事务所需要的所有数据页缓存到哪里
- 本地策略：为每一个事务需要缓存的文件页有单独的管理策略

下面介绍**提升缓存池性能的一些策略**

- Multiple Buffer Pools，多缓存池

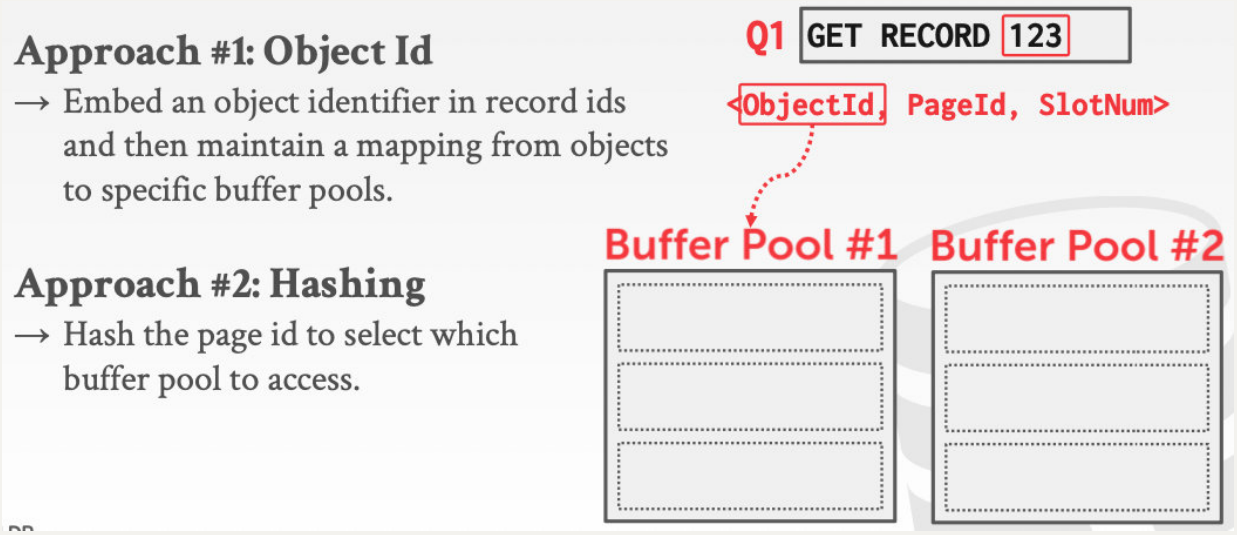
  数据库文件的页的用途各不相同，有些页用来作为索引页，有些页用于存储数据，有些页用于存储元数据，所以很多数据库采取这样的策略：索引页有专门的缓存池，存数据的页有专门的缓存池，存元数据的页也是，或者也可以不分类但依然使用多个缓存池来缓存磁盘里的页

  这样可以减少锁的争用，因为在做这种优化之前，内存中只有一个缓存池，线程在读写缓存池数据时要先获取缓存池的锁，使用多个缓存池可以减少锁的争用，这和xv6中可以使用多个哈希桶管理内存里的磁盘块缓冲区从而减少锁争用是一个道理

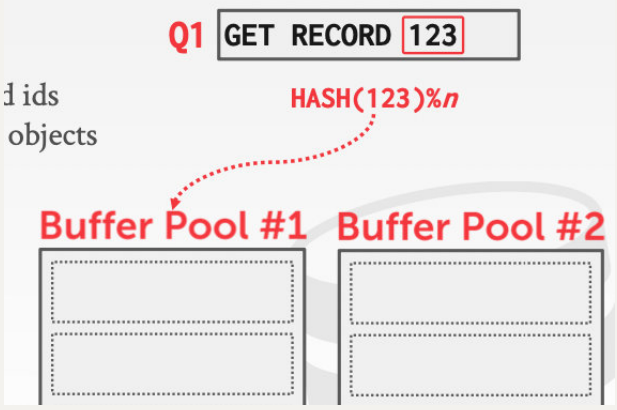


接下来看一下多缓存池的具体实现

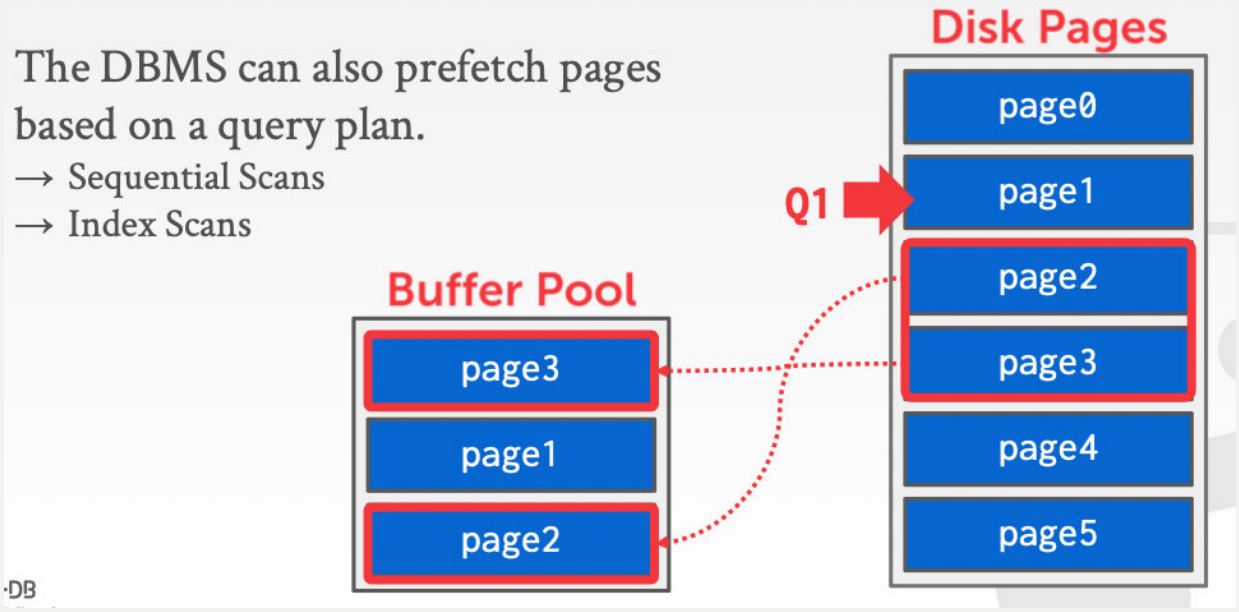
我们可以人为记录某条数据被放在了哪个缓存池，维护一个数据的id和缓存池号间的映射，如下图



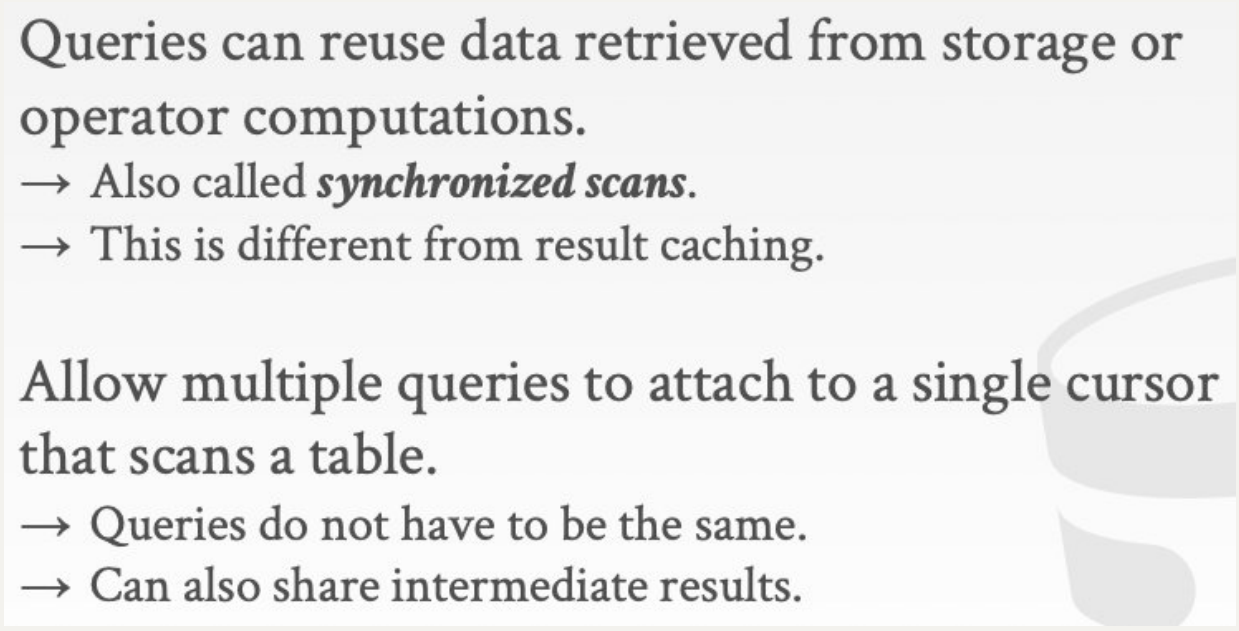
或者直接对数据的id进行哈希运算，得出这条数据在哪个缓存池

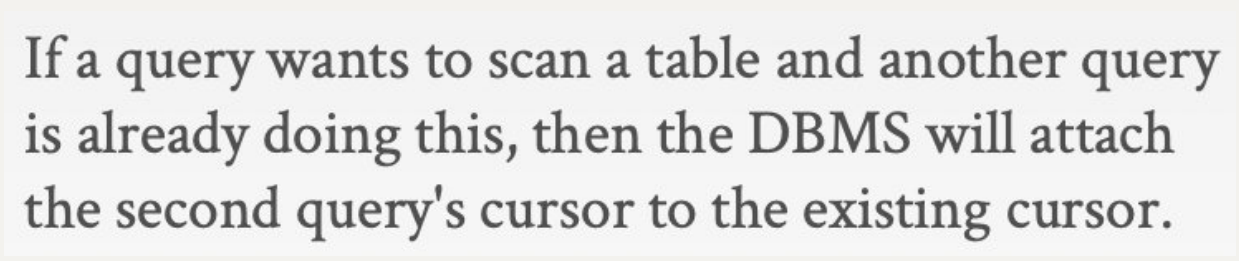


- Pre-fetching，预先拉取，如果我们要把数据库的表从第0条到最后一条全遍历一遍，即全表扫描，就可以预先把未来要读的数据页从磁盘读入内存，大致如下图，详细过程可参考15-445的sildes



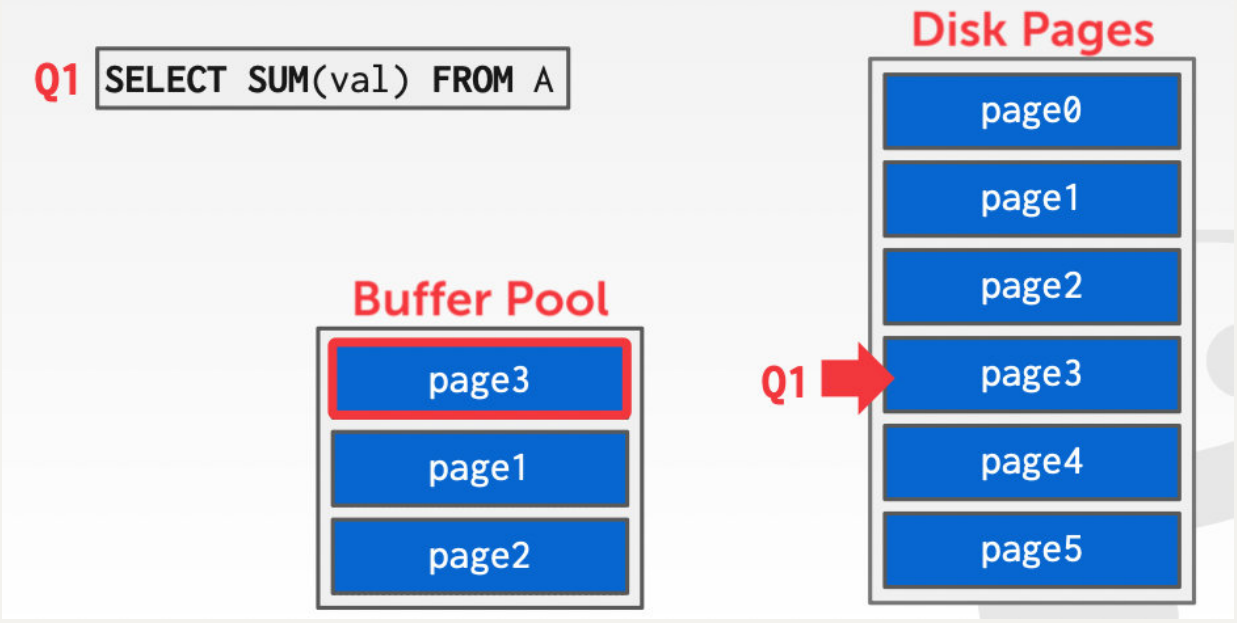
- Scan Sharing，共享扫描，如果两个SQL语句在读同一个表，如果A语句需要a和b join的结果，B语句也需要a和b join的结果，这两个语句几乎同时到达数据库，如果没有优化，数据库会给A做join操作，然后把结果给A，之后再给B做join操作，然后把结果给B，也就是分开处理两个语句，使用了scan sharing的优化策略之后，如果两个SQL想要同样的结果，就可以把扫描表的结果/中间结果共享，这种策略和result catching不同，result catching是先执行语句A，把结果记下来，等到B语句执行的时候，把记下来的结果给B，但是scan sharing策略是支持A语句和B语句并发执行，在它们并发进行的过程中DBMS有能力推断出它们是在读同一张表



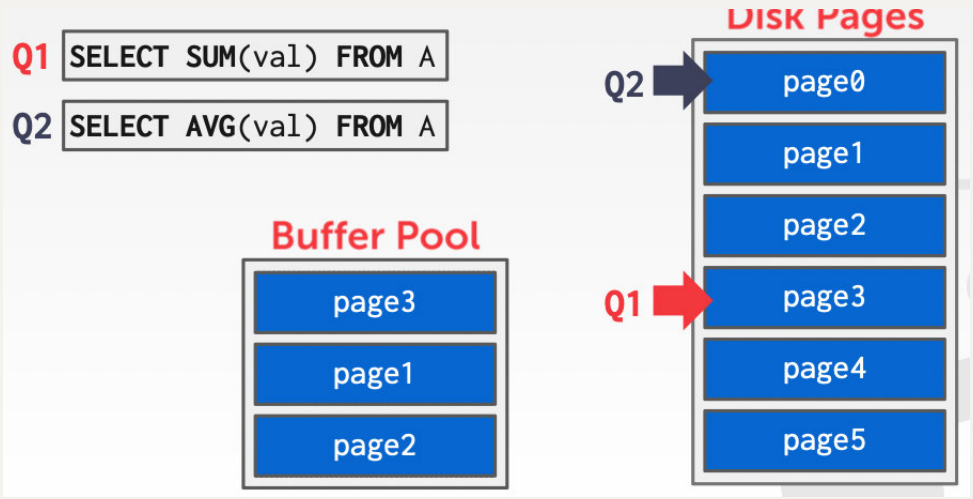


这个策略的图解如下：

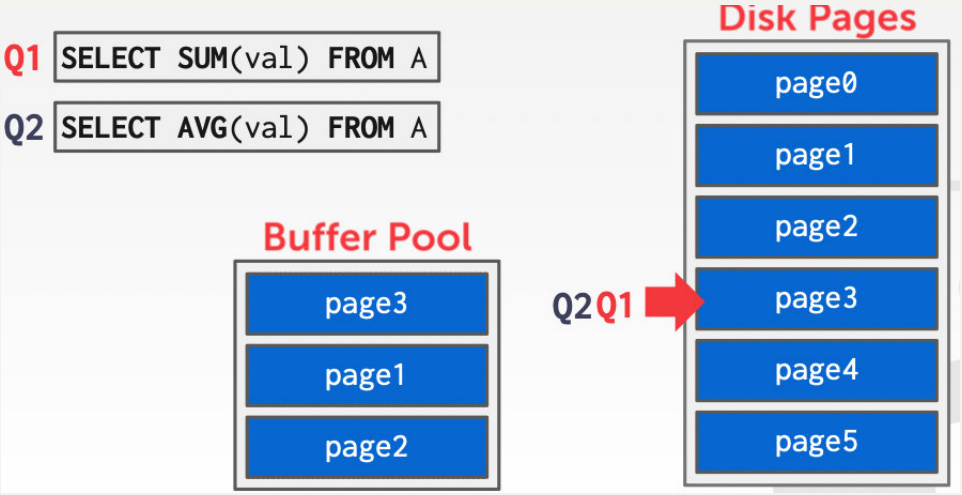
假设有Q1语句做某个字段的求和，Q2语句做该字段的求平均值，都是全表扫描，Q1先到达数据库，然后全表扫描，并且扫描到3号页的时候把0号页踢出了缓存池，



这时Q2语句到达数据库，Q2原本的逻辑应该是把0号页拉取到缓存池中然后开始全表扫描，



但这个时候DBMS会推断出这两个语句其实干的是一件事，之后DBMS会将Q1和Q2的执行合二为一：Q2语句先跟着Q1一起扫描3/4/5号页，扫描完5号页之后，Q1的任务完成了，然后Q2再回过头来扫描0/1/2号页



但是scan sharing也有一定的风险，如果我们的Q2语句限定是取前100个数据的平均值，像下面这样

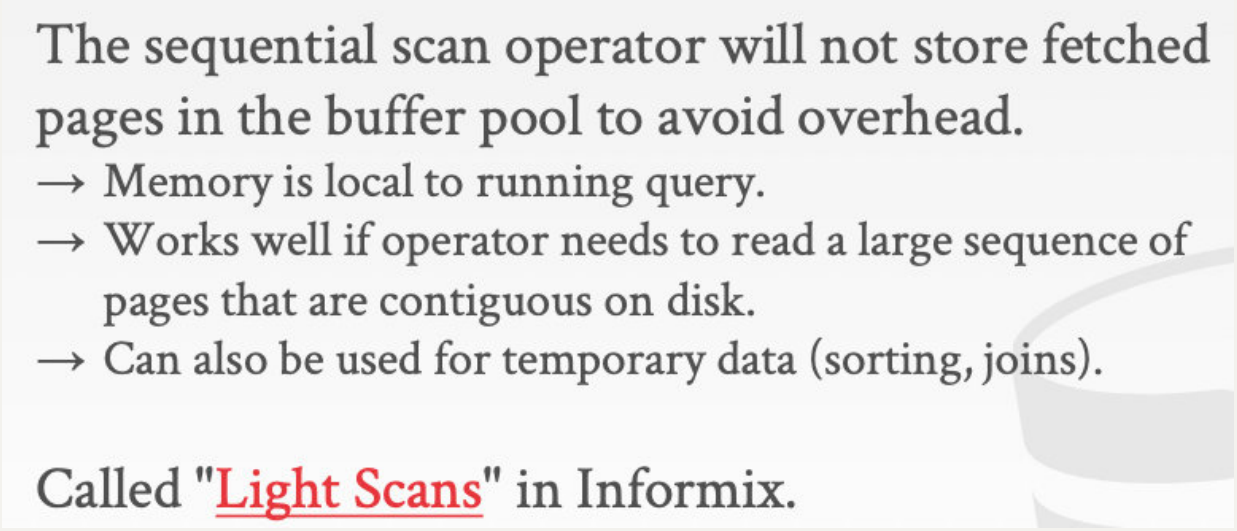
有可能只需要扫描0/1/2号页，按照上面的策略的话，就会出问题，我们无法控制数据库执行引擎不去做scan sharing的优化，因此如果像这个场景中需要强制从特定的位置开始扫描数据库文件的话，需要在SQL语句里面加入"order by"这样的关键字

- Buffer Pool Bypass，bypass，是旁路/绕开的意思，简单的概括就是“不进内存池”

  有些数据我们可能只需要用一次，后面也不会用了，我们可以将这种数据直接从磁盘放入内存当中没有池化的区域，之后把这块内存交给执行引擎使用，执行引擎使用完之后就把这块内存释放或者垃圾回收

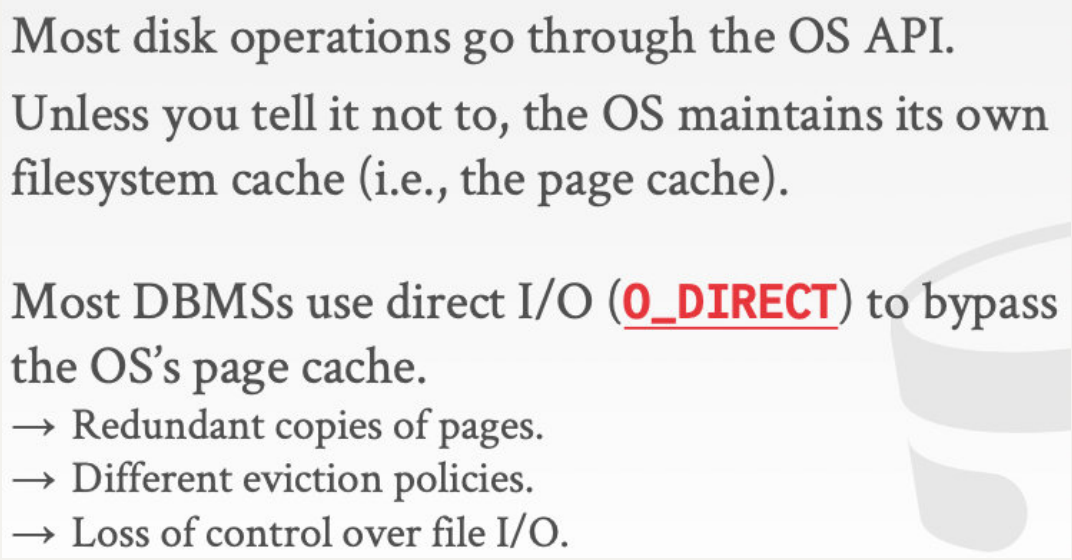
  这么做的话，就没有了将来将这种数据从缓存池淘汰导致的开销，如果SQL语句需要读取磁盘上连续的一大片区域，那么就有可能是上文所说的只使用一次数据的情况，便可以采取这样的优化策略，这种优化策略又名"light scans"，即“轻扫描”

  数据库的join操作的结果，其他的查询可能以后永远都用不到，因此我们也没必要把这些结果放到缓存池里缓存，把结果返回给用户之后就可以释放掉所占用的内存区域了



- 操作系统对于文件的页也是有缓存的，除了DBMS有缓存池，操作系统读写磁盘的时候也对磁盘的数据块也有缓存（比如xv6的buffer cache layer），但是DBMS一般会绕过操作系统提供的缓存机制，不需要操作系统缓存数据库文件，因为和DBMS维护的缓存池相比，操作系统对自己提供的磁盘缓存区没有很明确的淘汰策略，即每个缓存块保留多久，何时淘汰，甚至可以说OS的页缓存的淘汰机制很差，并且OS提供的缓存的DBMS的缓存池如果都存储了同样的数据，那就是很大的冗余(redundant)

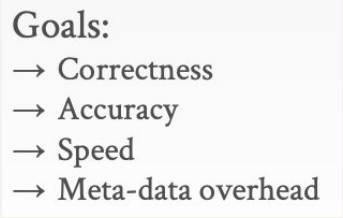
  因此DBMS需要读磁盘时，在操作系统提供的磁盘读写api中加入`O_DIRECT`参数，直接和磁盘进行I/O，绕过操作系统的页缓存



Replacement Policies

接下来讨论DBMS缓存池的替换策略

如果缓存池满了，DBMS的执行引擎还需要把新的数据缓存到缓存池里，那就一定要踢出缓存池中已经有的一部分数据，替换策略就在研究“踢掉谁”和“怎么踢”



DBMS的缓存池替换策略要保证它的正确性（不能把正在使用的页踢出），准确性，速度（不能花费太长时间用于决定踢出哪一页，即相关置换算法的复杂度不能太高），并且注意维护元数据带来的开销（比如说，元数据记录的是哪个页在缓存池里，哪个页不在缓存池里，把一些页踢出之后，要更新元数据，更新元数据的时候也不宜有太大开销）

下面介绍经典的替换策略，**LRU**，即著名的“最近最少使用”

一个比较简单的实现LRU的方法就是，给缓存池里每个被缓存的页都维护一个时间戳，用于记录执行引擎上次读这一页是在什么时候，当需要从缓存池踢出一个页时，就找到时间戳最久远的页然后踢出，但如果真的这么实现的话，我们需要在每次想驱逐缓存池里的页时，把缓存池里的页都遍历一遍，然后找到时间戳最久远的那个，开销会很大，因此有如下的时间复杂度更低的LRU实现方式：

clock

这是一个没有使用时间戳的近似LRU算法，每次驱逐的页未必一定是距离上次被访问时间上最久远的，因为精确地找出时间上最久远的页面带来的开销一般都不小，不妨就模糊地找出时间相对比较久远的页然后踢出

clock算法将缓存池中所有页在逻辑上做了一个环形的排列（通过例如链表这种数据结构来实现），每个页都有一个标志位`ref`，DBMS执行引擎每访问一次缓存池里的某页，这页的`ref`就会变成1，与此同时，还有一个“时针”绕着缓存页的环形排列旋转，如果被时针指到的页的`ref`是1，那就将`ref`清空为0，但是不驱逐这个页，如果被指到的页的`ref`是0，那么就把它驱逐（因为从时针上次指到它，到这次指到它，这一个周期里，没有任何线程读过这个页），而且还要注意，clock算法踢出页面的时候缓存池不一定是满的，这可以理解为转动的时针周期性地找出最近没用过的页面然后驱逐

CLOCK

Approximation of LRU that does not need a separate timestamp per page.

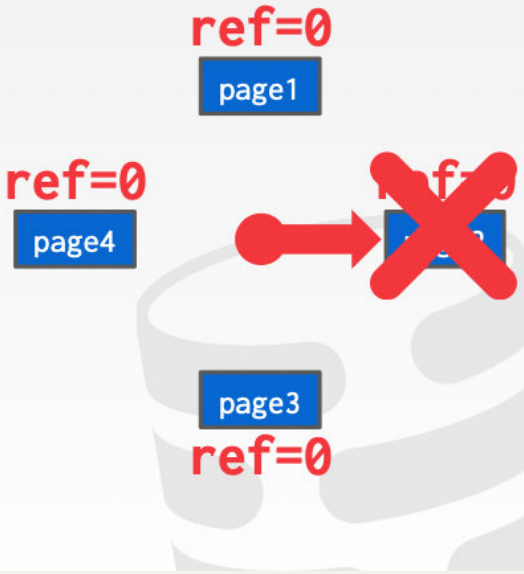
→ Each page has a reference bit.

→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

→ Upon sweeping, check if a page's bit is set to 1.

→ If yes, set to zero. If no, then evict.



（其实《操作系统导论》中页面置换算法那一章里更加详细地介绍过这个近似LRU算法）

DBMS的基于LRU思想的缓存池替换策略也有一些问题，比如说对于sequential flooding（像全表扫描时从第0页读到第n页这种场景）性能很差，因为当前的语句是要做全表扫描，扫描完了之后也许就不会再访问被扫描过的某些数据了，要读第n页时，在缓存池中的第n-1页是最近刚被访问过的，最不可能被驱逐的，但有可能之后不会有线程再访问第n-1页，此时在缓存池中的第n-1, n-2, n-3页...，这里面哪个以后会被访问都不好说

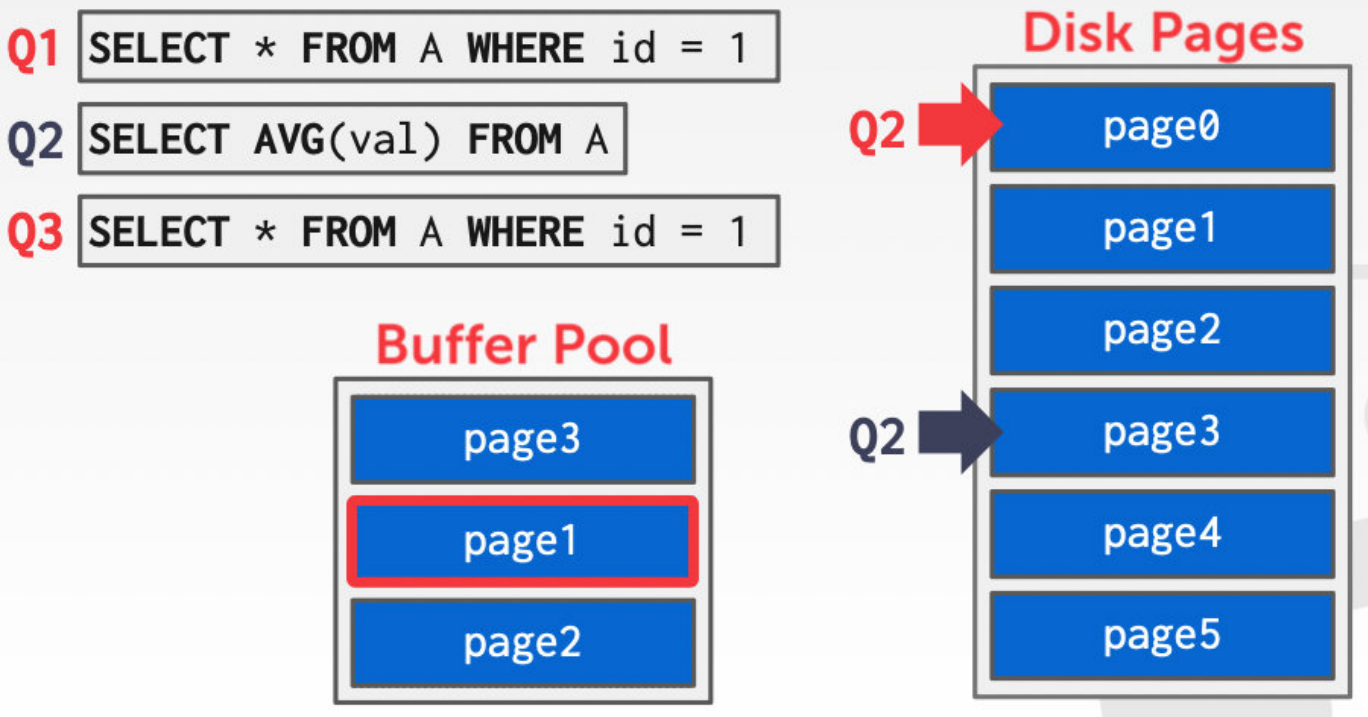
LRU and CLOCK replacement policies are susceptible to sequential flooding.

→ A query performs a sequential scan that reads every page.

→ This pollutes the buffer pool with pages that are read once and then never again.

In some workloads the most recently used page is the most unneeded page.

我们结合具体的场景来分析这种情况



如上图所示，用户发出的第一条SQL语句Q1是想要查询id为1的那条数据，这种查询单条数据的行为也叫点查询，想要查找的数据位于page 0当中，因为有可能在后面还会用到这条id为1的数据，所以page 0属于热点页，但如果在Q1被执行完之后，接下来要执行的是需要进行全表扫描的Q2语句，Q2会读很多页，因此读到后面的时候，LRU策略必定会把最早进入缓存池的page 0给驱逐，但是page 0是热点页，有可能Q2执行完之后，又有语句想访问id为1的那条数据进而访问page 0，就像上图的Q3语句，这时就需要再把page 0从磁盘重新读回缓冲区，开销就很大（这个场景发生时的具体过程还需参考15-445的slides）

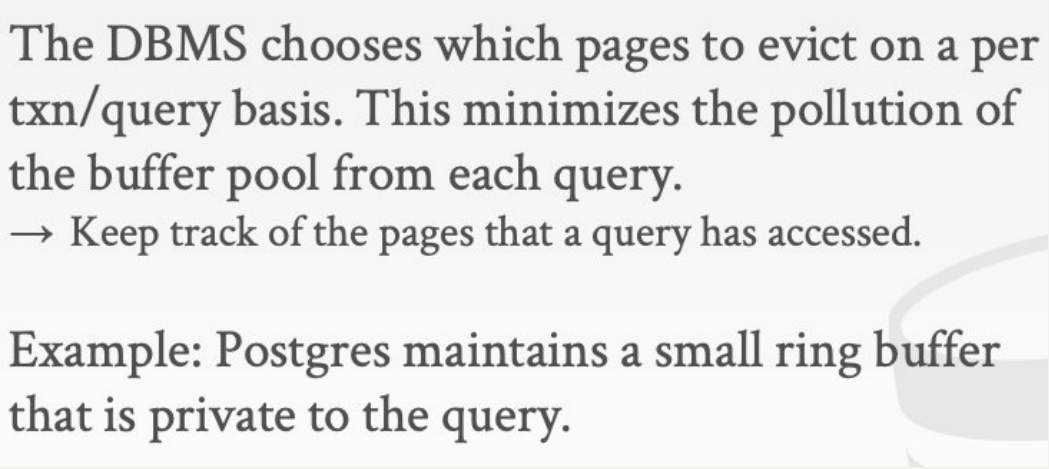
因此，考虑到单纯的LRU替换策略有一定的不足之处，我们需要对它加一点优化，有如下几种策略：

- LRU-K

> Track the history of last *K* references to each page as timestamps and compute the interval between subsequent accesses.
>
> The DBMS then uses this history to estimate the next time that page is going to be accessed.

这个方法具有一点统计学的思想，它不只关注某个页上一次被访问是何时，而是关注这个页之前的k次访问都是在何时，从而发现规律，比如说判断出这个页是不是在被周期性地访问，因此，在上文提到的两次点查询中间夹一次全表扫描的场景中，我们使用LRU-K策略的话，page 0就不会被轻易驱逐

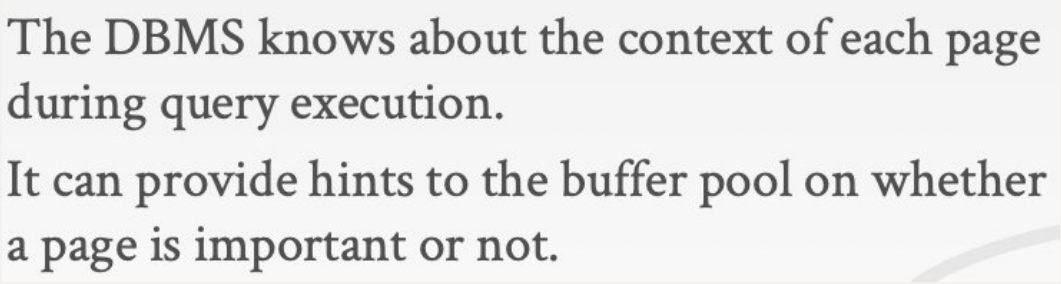
- Localization，本地化



笔者个人认为，本地化是将每一条SQL语句/每个事务的驱逐页面的需求分开处理，并且将驱逐页面的结果彼此之间隔离开来，互不影响，这听起来很抽象，实则属于上文介绍的allocation policies中的local policies，还是拿刚才说的场景举例，DBMS可以记住page 0是语句Q1所使用过的，Q2语句执行过程中需要驱逐页面时，就会知道page 0是语句Q1所使用的，先不驱逐page 0，只驱逐那些单独Q2自己使用的页

- Priority Hints，优先级提醒

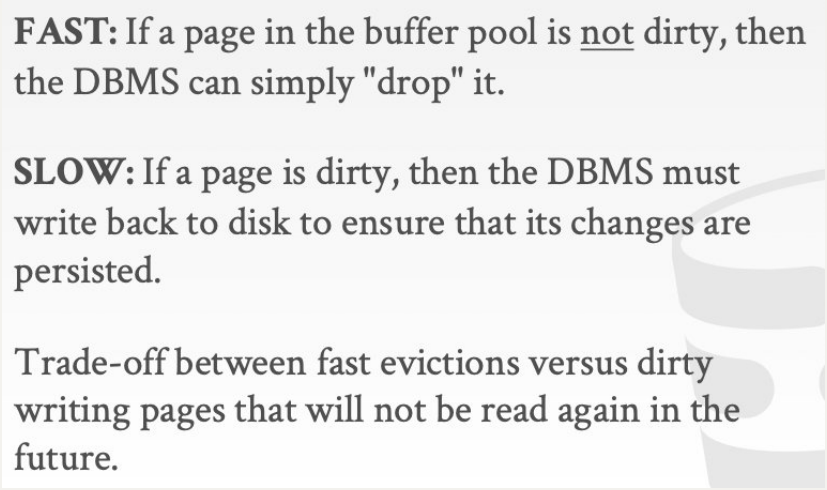
这是说DBMS执行引擎在运行时可以给缓存池一些指示，类似于“告诉缓存池这个页很重要，后面还要被用到，不要轻易清理”，至于怎么判断出来某个页重不重要，可以通过机器学习算法或者其他的很高级的策略来实现



前面说的都和从磁盘读数据有关，接下来讲的和把内存里数据的修改写回磁盘有关

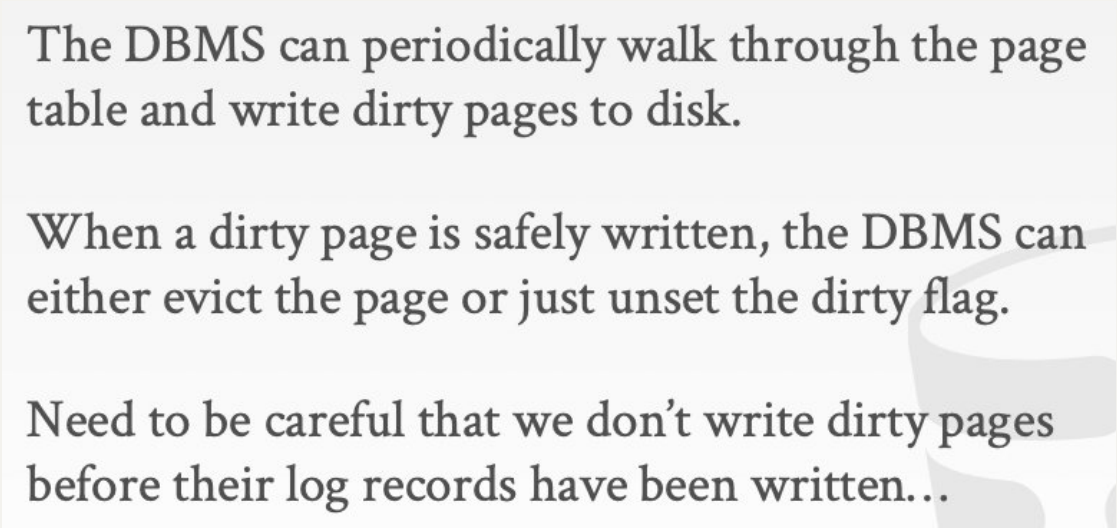
- Dirty Pages，脏页

脏页说的就是在内存里面被修改了但还没写回磁盘的页，DBMS在清理内存池时，如果发现某个页不是脏页，也就是自从它被放入内存后从来没被修改过，那就可以在需要淘汰它时直接淘汰它，但如果该页是脏页，那当DBMS需要把它驱逐出缓存池的时候，还要把修改写回磁盘



脏页的存在催生了WAL策略，即write ahead log，修改缓存池的数据之后不会立刻写回磁盘，否则会导致过多的磁盘I/O开销，一般会找一个机会集中写回，但是如果修改了缓存池里的数据后断电了，为了解决这个问题，可以额外再记个日志，专门记录哪些东西在缓存池里被修改了但还没写回磁盘，我们可以在用户修改数据库时，不将缓存池的脏页立刻写回磁盘，但日志必须要先被立刻写回磁盘

- Background Writing，异步地在背后将脏页写回

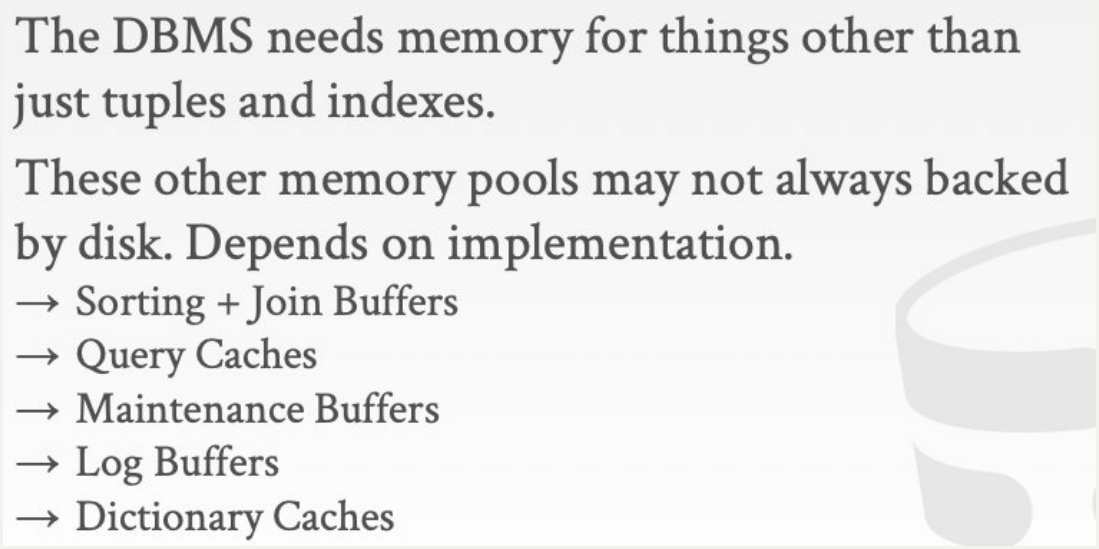


每隔一段时间集中地把缓存池里的脏页写回磁盘，当一个脏页已经被安全地写回了磁盘时，那么这个脏页就可以被驱逐，或者把缓存池中这个页的脏页标志位清除

Other Memory Pools

DBMS不仅需要缓存磁盘上的文件数据，还需要缓存其他的东西，比如说热点的SQL语句的执行结果，WAL策略中的日志，etc.

这本质上是因为磁盘I/O的开销太大，因此需要使用内存作为磁盘读/写时的缓存



总之，让DBMS自己去处理缓存，永远要比让OS来处理缓存要好，因为只有DBMS知道缓存的每一页是干什么用的，什么时候被用过，以后有没有可能被再用

Reference:

https://www.bilibili.com/video/BV1VP4y1H7TL/?spm_id_from=333.788