**buffer pool**

我们将构建一个面向磁盘的数据库系统，这样的一个数据库系统的主要存储位置是磁盘。首先我们要在存储管理系统中实现一个缓冲池(**buffer pool**),缓冲池负责物理页面和主存之间数据的来回移动。缓冲池允许数据库管理系统处理大于系统可用内存的数据。缓冲池的操作对数据库的其他部分是透明的。例如，系统利用唯一的标识符**page\_id\_t**向缓冲池请求一个页面，系统不知道这个页面是已经在内存中了还是需要从磁盘中将该页面取出。

我们的实现将是线程安全(**thread-safe**)的。由于多个线程可能同时访问内部数据结构，因此我们需要确保它们的关键部分受到**锁存器(latches)**的保护(在操作系统中被称为锁，即lock)。

在本部分，我们将实现如下功能：

* **LRU换页策略**
* **缓冲池管理器实例**
* **并行的缓冲池管理器**

**一．LRU换页策略**

这一组件用于追踪缓冲池中每一页的使用情况。我们将在**src/include/buffer/lru\_replacer.h**以及与其相关的实现文件**src/buffer/lru\_replacer.cpp**中实现类**LRUReplacer**。**LRUReplacer**继承自抽象类**Replacer(src/include/buffer/replacer.h)**，该文件包含各函数的功能说明。

**LRUReplacer**的最大页数与缓冲池的大小相同，这是因为**LRUReplacer**包含**BufferPoolManager**中所有帧的占位符。然而，在任意时刻，并非所有帧一定都在**LRUReplacer**中。初始化**LRUReplacer**中没有任何帧。只有新的取消占用(占用=固定)(**unpinned**)的帧将被考虑放入**LRUReplacer**中。

我们将实现如下方法：

* **Victim(frame\_id\_t\*)** : 在**Replacer**追踪的元素中，删除最近最少访问的对象，在输出参数中存储该对象的内容并返回**True**。如果**Replacer**是空的，返回**False**。
* **Pin(frame\_id\_t)** :在将一个页面固定到**BufferPoolManager**的帧后该方法应该被调用。该方法将从**LRUReplacer**中移除该被固定到**BufferPoolManager**的帧。
* **Unpin(frame\_id\_t)** : 当一个页面的**pin\_count**变为0后该方法应该被调用。该方法将包含取消固定的页面的帧移入**LRUReplacer**中。
* **Size()** : 该方法返回目前在**LRUReplacer**中帧的数量。

**二．缓冲池管理实例**

接下来，我们实现实现缓冲池管理功能(**BufferPoolManagerInstance**)。**BufferPoolManagerInstance**负责从**DiskManager**中获取数据库页面，并将其存储在主存中。在明确指示或者需要换出旧页面为新页面腾出空间时，**BufferPoolManagerInstance**可以将脏页写入磁盘。注意，**DiskManager**负责将数据从磁盘中读出以及将数据写入磁盘。

数据库系统中所有在主存中的页面都用**Page**对象表示。**BufferPoolManagerInstance**不需要理解这些页的内容。但是作为系统开发者，我们需要明确，**Page**对象就是缓冲池中存储内容的容器，因此它不和某个特定页面相关联。也就是说，每个**Page**对象都包含一块内存，**DiskManager**将它从磁盘中读取的物理页面的内容复制到该位置。当**BufferPoolManagerInstance**从磁盘中读写数据时，它将重用相同的**Page**对象来存储数据。这意味着在系统的生命周期内，相同的**Page**对象可能包含不同的物理页面。**Page**对象的标识符(**page\_id**)跟踪它包含的物理页面。如果一个**Page**对象不包含任何物理页面，那么它的**page\_id**必须设定为**INVALID\_PAGE\_ID**。

每个**Page**对象还维护一个计数器，用于统计固定了该页面的线程数。**BufferPoolManagerInstance**不允许释放一个被固定住的页面。每个**Page**对象还时刻追踪它是否是脏页，我们需要记录在一个页面被取消固定前它是否被修改。在**Page**对象可被重用前，**BufferPoolManagerInstance**必须将脏**Page**的内容写回磁盘。

**BufferPoolManagerInstance**将使用之前实现的**LRUReplacer**类。它将使用**LRUReplacer**来跟踪**Page**对象的访问时间，这样当**BufferPoolManagerInstance**需要为磁盘中的一个新物理页面腾出一个空闲帧时，它可以确定应该将哪个帧移出。

我们将实现头文件src/include/buffer/buffer\_pool\_manager\_instance.h以及源文件src/buffer/buffer\_pool\_manager\_instance.cpp中的如下函数：

* **FetchPgImp(page\_id)**
* **UnpinPgImp(page\_id, is\_dirty)**
* **FlushPgImp(page\_id)**
* **NewPgImp(page\_id)**
* **DeletePgImp(page\_id)**
* **FlushAllPagesImpl()**

如果在空闲列表中没有可用的页面并且所有的页面都被占用，**FetchPgImp**应该返回NULL。不论页面的状态如何(pin或unpin)，**FlushPgImp**都应该刷新页面。

对于**UnpinPgImp**，参数**is\_dirty**用于页面被占用时追踪该页是否被修改过。

注意：**Pin**和**Unpin**在**LRUReplacer**和**BufferPoolManagerInstance**中有着相反的含义，在**LRUReplacer**中，一个页面被pin意味着我们不应该删除该页面,因为该页面正在被使用。这意味着我们应该将该页面从**LRUReplacer**中移除。另一方面，在**BufferPoolManagerInstance**中一个页面被pin意味着我们想使用该页面，那么我们不应将该页从缓冲池中移除。

**三．并行的缓冲池管理**

之前实现的单例缓冲池管理实例为了实现线程安全需要锁存器。但当若干个线程与缓冲池交互争夺一个锁存器时，这会导致性能问题。一个潜在的解决方案是为数据库系统分配多个缓冲池，每个缓冲池都有自己的锁存器。

**ParallelBufferPoolManager**是有着多个**BufferPoolManagerInstance**的类。对于每个操作，**ParallelBufferPoolManager**选取一个**BufferPoolManagerInstance**实例并调用该实例。

我们依据给定的page id来决定使用哪个**BufferPoolManagerInstance**。如果我们有**num\_instances**个**BufferPoolManagerInstance**，那么我们要用某种方式将给定的page id映射到[0,num\_instances)之间，在这里我们使用模运算**page\_id mod num\_instances**将给定的page\_id映射到该区间。

当**ParallelBufferPoolManager**初始化的时候，他的起始索引应该是0。此后每创建一个新的页面，都应从起始索引开始尝试每个**BufferPoolManagerInstance**直到某个成功。然后将起始索引加1。

创建**BufferPoolManagerInstances**时，构造函数应使用uint32\_t num\_instances和uint32\_t instance\_index，来保证页面id被正确地创建。

我们将实现头文件src/include/buffer/parallel\_buffer\_pool\_manager.h和源文件src/buffer/parallel\_buffer\_pool\_manager.cpp中的如下函数：

* **ParallelBufferPoolManager(num\_instances, pool\_size, disk\_manager, log\_manager)**
* **~ParallelBufferPoolManager()**
* **GetPoolSize()**
* **GetBufferPoolManager(page\_id)**
* **FetchPgImp(page\_id)**
* **UnpinPgImp(page\_id, is\_dirty)**
* **FlushPgImp(page\_id)**
* **NewPgImp(page\_id)**
* **DeletePgImp(page\_id)**
* **FlushAllPagesImpl()**