#1 DISK AND BUFFER POOL MANAGER

#1.1 实验概述

在MiniSQL的设计中，Disk Manager和Buffer Pool Manager模块位于架构的最底层。Disk Manager主要负责数据库文件中数据页的分配和回收，以及数据页中数据的读取和写入。其中，数据页的分配和回收通过位图（Bitmap）这一数据结构实现，位图中每个比特（Bit）对应一个数据页的分配情况，用于标记该数据页是否空闲（`0` 表示空闲，`1` 表示已分配）。当Buffer Pool Manager需要向Disk Manager请求某个数据页时，Disk Manager会通过某种映射关系，找到该数据页在磁盘文件中的物理位置，将其读取到内存中返还给Buffer Pool Manager。而Buffer Pool Manager主要负责将磁盘中的数据页从内存中来回移动到磁盘，这使得我们设计的数据库管理系统能够支持那些占用空间超过设备允许最大内存空间的数据库。

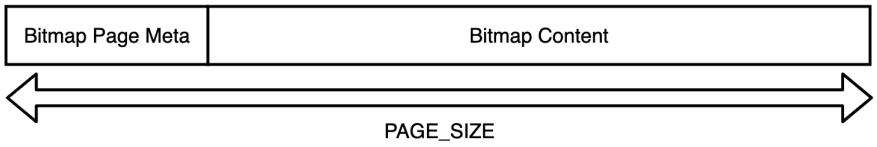
Buffer Pool Manager中的操作对数据库系统中其他模块是透明的。例如，在系统的其它模块中，可以使用数据页唯一标识符 `page_id` 向Buffer Pool Manager请求对应的数据页。但实际上，这些模块并不知道该数据页是否已经在内存中还是需要从磁盘中读取。同样地，Disk Manager中的数据页读写操作对Buffer Pool Manager模块也是透明的，即Buffer Pool Manager使用逻辑页号 `logical_page_id` 向Disk Manager发起数据页的读写请求，但Buffer Pool Manager并不知道读取的数据页实际上位于磁盘文件中的哪个物理页（对应页号 `physical_page_id` ）。

注意：对于下面需要实现的每个模块，我们提供了相关的类与一组公共接口。在编写代码时，请勿修改这些类与公共接口的签名，否则将无法通过测试代码。但你可以选择向类中添加数据成员或是辅助函数以实现所需的功能。为了简化工程，目前我们只考虑单线程下的设计，但在程序中我们仍然保留了并发控制相关的锁，供对感兴趣实现事务、并发控制、故障恢复相关功能模块的同学使用。

#1.2 热身: 实现一个简单的位图页

为了帮助同学们更快地熟悉MiniSQL代码的编写、调试与测试工作，我们设计了一个简单的小任务：实现一个简单的位图页（Bitmap Page），位图页是Disk Manager模块中的一部分，是实现磁盘页分配与回收工作的必要功能组件。位图页与数据页一样，占用 `PAGE_SIZE` （4KB）的空间，标记一段连续页的分配情况。

Bitmap Page由两部分组成，一部分是用于加速Bitmap内部查找的元信息（Bitmap Page Meta），它可以包含当前已经分配的页的数量（ `page_allocated_` ）以及下一个空闲的数据页(`next_free_page_`)，元信息所包含的内容可以由同学们根据实际需要自行定义。除去元信息外，页中剩余的部分就是Bitmap存储的具体数据，其大小 `BITMAP_CONTENT_SIZE` 可以通过 `PAGE_SIZE - BITMAP_PAGE_META_SIZE` 来计算，自然而然，这个Bitmap Page能够支持最多纪录 `BITMAP_CONTENT_SIZE * 8` 个连续页的分配情况。



与Bitmap Page相关的代码位于 `src/include/page/bitmap_page.h` 和 `src/page/bitmap_page.cpp` 中，以下函数需要被实现：

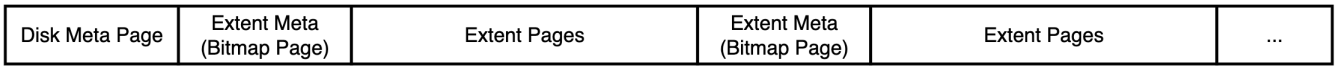
- `BitmapPage::AllocatePage(&page_offset)` ：分配一个空闲页，并通过 `page_offset` 返回所分配的空闲页位于该段中的下标（从 `0` 开始）；
- `BitmapPage::DeAllocatePage(page_offset)` ：回收已经被分配的页；
- `BitmapPage::IsPageFree(page_offset)` ：判断给定的页是否是空闲（未分配）的。

此外，与该模块相关的测试代码位于 `test/storage/disk_manager_test.cpp` 中。

#1.3 磁盘数据页管理

在实现了基本的位图页后，我们就可以通过一个位图页加上一段连续的数据页（数据页的数量取决于位图页最大能够支持的比特数）来对磁盘文件（DB File）中数据页进行分配和回收。但实际上，这样的设计还存在着一点点的小问题，假设数据页的大小为4KB，一个位图页中的每个字节都用于记录，那么这个位图页最多能够管理32768个数据页，也就是说，这个文件最多只能存储 `4K * 8 * 4KB = 128MB` 的数据，这实际上很容易发生数据溢出的情况。

为了应对上述问题，一个简单的解决思路是，把上面说的一个位图页加一段连续的数据页看成数据库文件中的一个分区（Extent），再通过一个额外的元信息页来记录这些分区的信息。通过这种“套娃”的方式，来使磁盘文件能够维护更多的数据页信息。其主要结构如下图所示：



Disk Meta Page是数据库文件中的第 `0` 个数据页，它维护了分区相关的信息，如分区的数量、每个分区中已经分配的页的数量等等。接下来，每一个分区都包含了一个位图页和一段连续的数据页。在这样的设计下，我们假设Disk Meta Page能够记录 `4K/4=1K` 个分区的信息，那么整个数据库能够维护的数据页的数量以及能够存储的数据数量与之前的设计相比，扩大了1000倍。与Disk Meta Page相关的代码定义在 `src/include/page/disk_file_meta_page.h` 中。

不过，这样的设计还存在着一个问题。由于元数据所占用的数据页实际上是不存储数据库数据的，而它们实际上又占据了数据库文件中的数据页，换而言之，实际上真正存储数据的数据页是不连续的。举一个简单例子，假设每个分区能够支持3个数据页，那么实际上真正存储数据的页只有：2, 3, 4, 6, 7, 8...

物理页号	0	1	2	3	4	5	6	...
职责	磁盘元数据	位图页	数据页	数据页	数据页	位图页	数据页	
逻辑页号	/	/	0	1	2		3	

但实际上，对于上层的Buffer Pool Manager来说，希望连续分配得到的页号是连续的（0, 1, 2, 3...），为此，在Disk Manager中，需要对页号做一个映射（映射成上表中的逻辑页号），这样才能使得上层的Buffer Pool Manager对于Disk Manager中的页分配是无感知的。

因此，在这个模块中，需要实现以下函数，与之相关的代码位于 `src/include/storage/disk_manager.h` 和 `src/storage/disk_manager.cpp` 。

- `DiskManager::AllocatePage()` ：从磁盘中分配一个空闲页，并返回空闲页的**逻辑页号**；
- `DiskManager::DeAllocatePage(logical_page_id)` ：释放磁盘中**逻辑页号**对应的物理页。

- `DiskManager::IsPageFree(logical_page_id)`：判断该**逻辑页号**对应的数据页是否空闲。
- `DiskManager::MapPageId(logical_page_id)`：可根据需要实现。在 `DiskManager` 类的私有成员中，该函数可以用于将逻辑页号转换成物理页号。

此外，为了确保系统的其余部分正常工作，我们将在Disk Manager中提供一些已经实现的功能，如磁盘中数据页内容的读取和写入等等。

#1.4 LRU替换策略

Buffer Pool Replacer负责跟踪Buffer Pool中数据页的使用情况，并在Buffer Pool没有空闲页时决定替换哪一个数据页。在本节中，你需要实现一个基于LRU替换算法的 `LRUReplacer`， `LRUReplacer` 类在 `src/include/buffer/lru_replacer.h` 中被定义，其扩展了抽象类 `Replacer`（在 `src/include/buffer/replacer.h`）中被定义。 `LRUReplacer` 的大小默认与Buffer Pool的大小相同。

因此，在这个模块中，需要实现以下函数，与之相关的代码位于 `src/buffer/lru_replacer.cpp` 中。

- `LRUReplacer::Victim(*frame_id)`：替换（即删除）与所有被跟踪的页相比最近最少被访问的页，将其页帧号（即数据页在Buffer Pool的Page数组中的下标）存储在输出参数 `frame_id` 中输出并返回 `true`，如果当前没有可以替换的元素则返回 `false`；
- `LRUReplacer::Pin(frame_id)`：将数据页固定使之不能被 `Replacer` 替换，即从 `lru_list_` 中移除该数据页对应的页帧。 `Pin` 函数应当在一个数据页被Buffer Pool Manager固定时被调用；
- `LRUReplacer::Unpin(frame_id)`：将数据页解除固定，放入 `lru_list_` 中，使之可以在必要时被 `Replacer` 替换掉。 `Unpin` 函数应当在一个数据页的引用计数变为 `0` 时被Buffer Pool Manager调用，使页帧对应的数据页能够在必要时被替换；
- `LRUReplacer::Size()`：此方法返回当前 `LRUReplacer` 中能够被替换的数据页的数量。

Bonus：除LRU Replacer外，实现一种新的缓冲区替换算法（如Clock Replacer）。

#1.5 缓冲池管理

在实现Buffer Pool的替换算法 `LRUReplacer` 后，你需要实现整个 `BufferPoolManager`，与之相关的代码位于 `src/include/buffer/buffer_pool_manager.h` 和 `src/buffer/buffer_pool_manager.cpp` 中。Buffer Pool Manager负责从Disk Manager中获取数据页并将它们存储在内存中，并在必要时将脏页面转储到磁盘中（如需要为新的页面腾出空间）。

数据库系统中，所有内存页面都由 `Page` 对象（ `src/include/page/page.h` ）表示，每个 `Page` 对象都包含了一段连续的内存空间 `data_` 和与该页相关的信息（如是否是脏页，页的引用计数等等）。注意， `Page` 对象并不作用于唯一的数据页，它只是一个用于存放从磁盘中读取的数据页的容器。这也就意味着同一个 `Page` 对象在系统的整个生命周期内，可能会对应很多不同的物理页。 `Page` 对象的唯一标识符 `page_id_` 用于跟踪它所包含的物理页，如果 `Page` 对象不包含物理页，那么 `page_id_` 必须被设置为 `INVALID_PAGE_ID`。每个 `Page` 对象还维护了一个计数器 `pin_count_`，它用于记录固定(Pin)该页面的线程数。Buffer Pool Manager将不允许释放已经被固定的 `Page`。每个 `Page` 对象还将记录它是否脏页，在复用 `Page` 对象之前必须将脏的内容转储到磁盘中。

在 `BufferPoolManager` 的实现中，你需要用到此前已经实现的 `LRUReplacer` 或是其它的 `Replacer`，它将被用于跟踪 `Page` 对象何时被访问，以便 `BufferPoolManager` 决定在Buffer Pool中没有空闲页可以用于分配时替换哪个数据页。

因此，在这个模块中，需要实现以下函数：

- `BufferPoolManager::FetchPage(page_id)`：根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取；
- `BufferPoolManager::NewPage(&page_id)`：分配一个新的数据页，并将逻辑页号于 `page_id` 中返回；
- `BufferPoolManager::UnpinPage(page_id, is_dirty)`：取消固定一个数据页；
- `BufferPoolManager::FlushPage(page_id)`：将数据页转储到磁盘中；
- `BufferPoolManager::DeletePage(page_id)`：释放一个数据页；
- `BufferPoolManager::FlushAllPages()`：将所有的页面都转储到磁盘中。

对于 `FetchPage` 操作，如果空闲页列表（ `free_list_` ）中没有可用的页面并且没有可以被替换的数据页，则应返回 `nullptr`。 `FlushPage` 操作应该将页面内容转储到磁盘中，无论其是否被固定。

#1.6 模块相关代码

- `src/include/page/bitmap_page.h`
- `src/page/bitmap_page.cpp`
- `src/include/storage/disk_manager.h`
- `src/storage/disk_manager.cpp`
- `src/include/buffer/lru_replacer.h`
- `src/buffer/lru_replacer.cpp`
- `src/include/buffer/buffer_pool_manager.h`
- `src/buffer/buffer_pool_manager.cpp`
- `test/buffer/buffer_pool_manager_test.cpp`
- `test/buffer/lru_replacer_test.cpp`
- `test/storage/disk_manager_test.cpp`

#1.7 开发提示

1. 推荐在**夏学期第2周前**完成本模块的设计。
2. 我们推荐使用 `glog` 模块（ `glog` 是谷歌提供的日志模块，代码已在 `thirdparty` 目录下提供）来打印必要的调试和日志信息来代替 `printf`。在程序入口处调用 `google::InitGoogleLogging` 初始化日志模块后，即可通过 `LOG` 宏（通过 `#include "glog/logging.h"` 引入）来打印不同级别的日志，下面是一个示例：

C++ | 复制代码

```cpp
#include "glog/logging.h"

void InitGoogleLog(char *argv) {
  FLAGS_logtostderr = true;
  FLAGS_colorlogtostderr = true;
  google::InitGoogleLogging(argv);
}

int main(int argc, char **argv) {
    InitGoogleLog(argv[0]);
    LOG(INFO) << "This is an info log!";
    LOG(WARNING) << "This is a warning log!" << std::endl;
    LOG(ERROR) << "This is an error log!";
    return 0;
}
```

#1.8 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为 `0` ；
2. 请勿将代码发布到公共Github存储库上。