# MiniSQL 个人设计报告

1. 个人工作

在本次minisql的设计实验中，我主要负责模块一DISK AND BUFFER POOL MANAGER与模块二RECORD MANAGER的全部设计内容。因而在本minisql个人实验报告中，我便主要分析这两模块的设计原理与具体实现细节；其中第二章为DISK AND BUFFER POOL MANAGER模块部分，第三章为RECORD MANAGER模块部分，第四章则是对此次设计作业的总结与展望。对于第二第三部分的具体设计而言，我们在这仅给出部分重要函数的具体实现代码。

1. DISK AND BUFFER POOL MANAGER模块的设计

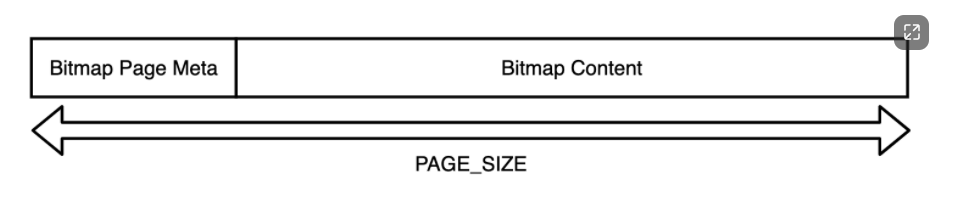
2.0 总体概述

在minisql的总体框架里，Disk Manager和Buffer Pool Manager模块位于架构的最底层。其中Disk Manager模块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与写入[[1]](#footnote-0)；而Buffer POOL Manager模块则更加偏向于将磁盘中的数据页从内存中来回移动到磁盘，在需要时向Disk Manager请求某个数据页，通过Disk Manager的特定映射关系，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模块。这样的交互结构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。

需要注意的是，Buffer Pool Manager中的操作对数据库系统中其他模块是透明的，这意味着其他模块的调用并不知道数据页是否已经在内存中还是需要从磁盘中读取；同样，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。这样的架构便对我们的设计提出挑战，接着，我们便开始介绍DISK AND BUFFER POOL MANAGER模块在实现过程中的具体细节。

2.1 简单位图的实现

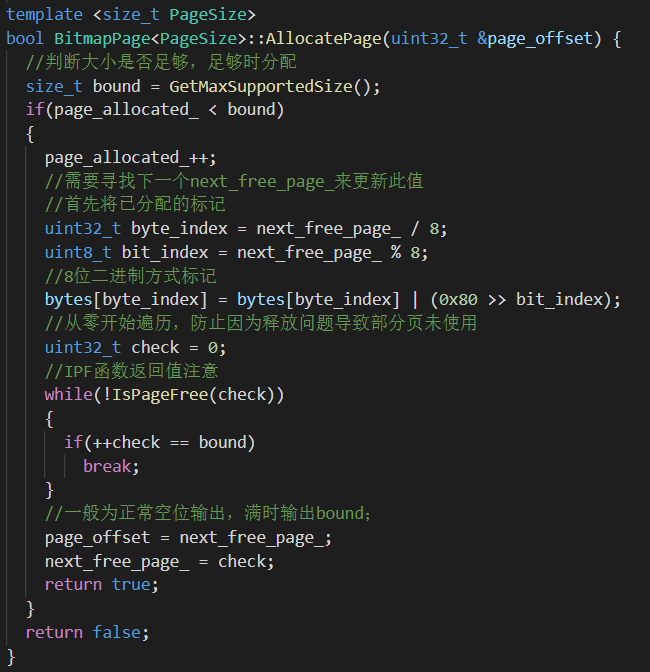
位图页是Disk Manager模块中的一部分，是实现磁盘页分配与回收工作的必要功能组件。其主要有两部分组成（详见下图），一部分（bitmap page meta）用于加速Bitmap内部查找的元信息，可以理解为头信息，包括当前已分配页的数量以及下一空闲的数据页；另一部分（bitmap content）则是存放其存储的具体数据（连续页的使用情况）。另外，位图页与数据页一样，都占据同样的空间大小。



在该模块中，我们主要实现了BitmapPage::AllocatePage、BitmapPage::

DeAllocatePage、BitmapPage::IsPageFree三个主要函数，接下来我便一一展现其中的设计细节。

首先是BitmapPage::AllocatePage函数，AllocatePage函数旨在获得尚未被利用的page，并将其对映的index载于page\_offset中返回；对于整个分配过程，若成功则返回true，反之则为false。我们先给出了代码实现的概况。



在分配空闲page前，我们需要判断已分配的page数量是否已经到达当前bitmap页支持的最大数量；若尚有空闲页，则可以根据类成员next\_free\_page立刻获得。需要特别注意的是，在分配一空闲页后，我们需要相应的修改bitmap的记录信息，也就是除却头信息的部分。为最大化使用page空间，我采用了bit记录的方式，即每一数据页是否被分配用1bit记录，1表示已被分配，0表示尚未被分配，且8bit作为一个char类型存储，以此记录数据页分配的总体情况。

接下来是BitmapPage::DeAllocatePage：释放某一指定数据页函数；对于DeAllocatePage函数，我们给定需要释放的数据页对映的page\_offset，其执行释放操作，在操作成功时返回true，反之则返回false。

在释放page\_offset对映的数据页之前，我们需要判断该数据页是否处于空闲状态（这可以依赖后续实现的IsPageFree函数）。若数据页处于空闲状态，则我们等价于执行了无效释放操作，需要返回false；如果数据页本身处于使用状态，则我们可以进行释放；在修改对映的bitmap记录信息外，需要特别注意，当bitmap本身处于数据页全分配的状态下，我们需要相应的更新next\_free\_page对象的指向（因为产生了新的free page）。以下是具体代码的实现细节：

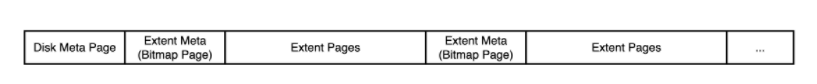


最后是BitmapPage::IsPageFree函数，用于判断给定的page\_offset数据页是否为空闲状态，若出于空闲状态，则返回true；反之则返回false。在本函数的实现中，可以直接调用IsPageFreeLow函数实现，因而没有过多的实现细节。

2.2磁盘数据页管理

在实现bitmap结构后，我们可以通过一个bitmap结构加上一系列支持bitmap管理的数据页来对磁盘文件进行分配和回收。但这样的设计，往往存在支持管理的数据页过少的情况，因而容易产生数据溢出的问题。

然而，我们可以考虑层级组织的方式，来扩充相应结构的管理能力。我们可以把前述的bitmap加一系列数据页的形式作为一个分区，再用一个额外的元信息页对不同的分区进行统一管理（类似于bitmap的结构方式）。这样的层级结构，便可以扩展我们维护的数据页个数，这种结构的实现实例如下：



其中Disk Meta Page 表示数据库文件的第0页，维护分区相关的元信息；接下来的空闲位置便可以存放一系列bitmap加数据页的结构。只要在Disk Meta Page 头信息中记录每一对映分区的相对偏移量，就可实现数据页的统一管理。结构的抽象实现样例见下：



但是，从上面的抽象细节表现中我们可以发现，bitmap页虽然占据了物理页号，存储在了磁盘中，但它不对映相应的逻辑页号，即不作为数据页可以被大规模读写。因而，在本部分函数的具体实现中，我们便需要考虑到上下层级间透明性的交互问题。而在本部分，我们主要需要实现DiskManager::AllocatePage、DiskManager::DeAllocatePage、DiskManager::IsPageFree、DiskManager::

MapPageId四个功能函数。

首先我们展开表述DiskManager::AllocatePage函数，该函数在disk中寻得（分配）一页空闲的数据页并返回分配数据页对映的逻辑页号。以下是具体的代码实现：



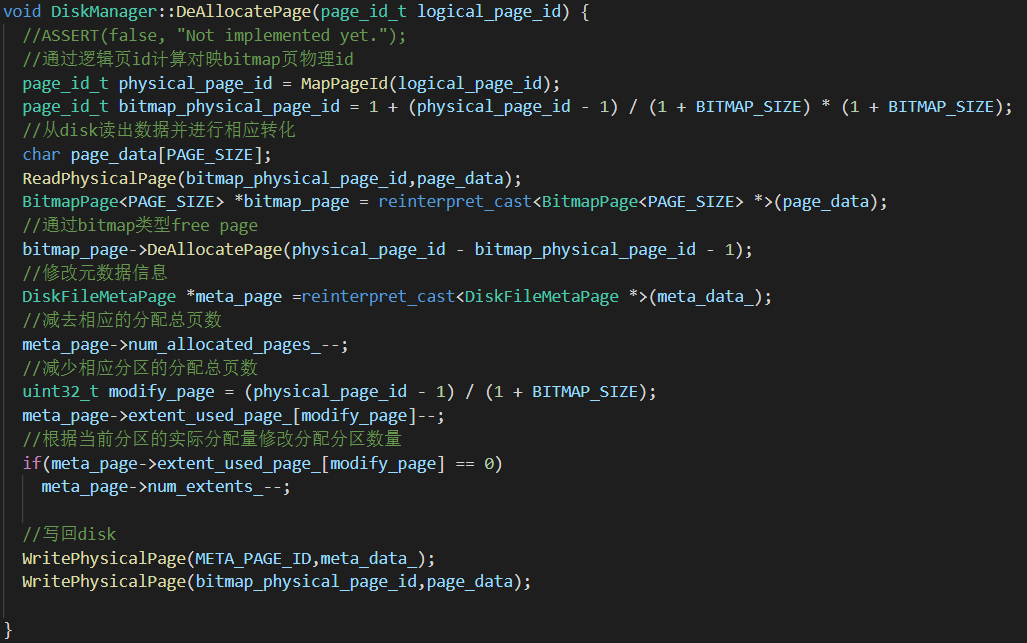
根据提示，为实现AllocatePage函数，我们可以把Disk\_manager类中成员meta-data转换为DiskFileMetaPage类型（类似于BufferPool缓存Page的作用，后续实现中会再次提及）。而在DiskFileMetaPage类中，我们天然得到了num\_allocated\_pages\_（分区总分配数据页之和）、num\_extents\_（当前存有使用数据页的分区数量）、extent\_used\_page\_（每一分区使用数据页数量记录数组）三项极其有用的元信息。

在具体分配数据页前，我们需要通过extent\_used\_page\_数据去寻找留有空闲数据页的bitmap信息。若不存在任何空闲的数据页，则说明disk数据页已被写满，则需要assert程序并返回错误信息；若找到了空闲的数据页，则我们需要根据数组的index计算出对映bitmap的物理页号。在获得了bitmap的物理页号后，我们可以使用框架中ReadPhysicalPage函数根据物理页号读取对映的page信息，并将对映的读取数据转换为BitmapPage结构，便于后续空闲数据页的找寻。

在获得bitmap的具体信息后，我们可以使用2.1部分已实现的BitmapPage::

AllocatePage函数直接获得当前bitmap管理的下一空闲的数据页page\_offset；接着，我们通过这个page\_offset可以直接计算出下一空闲数据页的物理页号，此时我们分配空闲数据页的任务就已完成。但我们分配了新的数据页，因此需要修改对应的page元信息，需要对映自增num\_allocated\_pages\_，对映自增当前分区的extent\_used\_page\_数组值，若当前分区本身没有任何数据页的分配，则需要自增num\_extents\_（新增了一个记录数据页分配的分区）；最后，我们需要使用框架中实现的WritePhysicalPage函数将修改后的结果写回到disk中，保持数据的一致性。

接着是DiskManager::DeAllocatePage函数，该函数接受一逻辑页号作为参数，并通过这一逻辑页号释放对映的page，当然，不能忘记修改对映bitmap信息。以下是具体的代码实现：



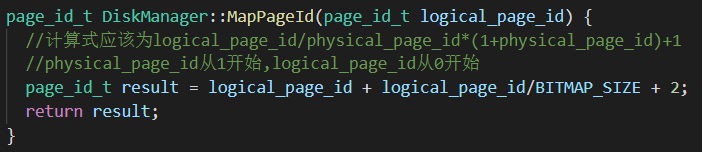
为实现这一函数，我们假设我们已经实现MapPageId函数（实现细节会在本部分的后续提及），我们可以先通过这个函数，将输入的逻辑页号转换为对应的物理页号，并通过这个物理页号反推出管理其的bitmap物理页号。接下来，我们便重复AllocatePage函数中的类似操作，从disk读取bitmap数据并进行对映更多类型转换。区别于AllocatePage函数实现细节的，是我们不再调用BitmapPage::AllocatePage函数用以分配新的空闲页，而是使用BitmapPage::DeAllocatePage函数释放对应的数据页，在此之后，我们的释放工作便已完成。

同样，我们需要去改变bitmap等结构的元信息，首先num\_allocated\_pages\_需要自减，其次当前分区对应的extent\_used\_page\_数组值需要自减，且当当前分区对应的extent\_used\_page\_数组值减至0时，我们需要自减num\_extents\_，因为一个分区的全部分配页都已被释放，因而该分区无需再被分配数量统计。最后，我们也同样需使用WritePhysicalPage函数把修改结果写回disk，保持数据的一致性。

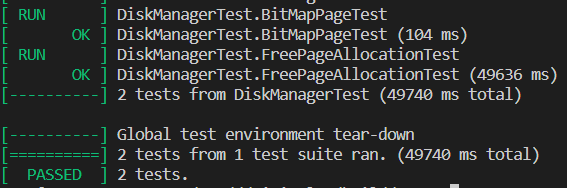
然后是DiskManager::IsPageFree函数，该函数接收一逻辑页号参数，并判断对映的数据页是否处于空闲状态。

为更便捷的实现本函数，我们也需使用到MapPageId函数，快速的将逻辑页号转变为物理页号，再通过对映的函数关系，求出当前页对映的管理bitmap页。接着也是相应的从disk读取bitmap数据，进行必要的类型装换，然后需要调用BitmapPage::IsPageFree函数，去判断对映的数据页是否处于空闲状态，并返回结果。因为本函数只是一个check，没有修改元信息，因而不需要WritePhysicalPage函数的写回。

最后是我们已经提及了很多次的DiskManager::MapPageId函数，该函数接受一逻辑页号作为输入参数，并通过相应的映射关系，计算获得并返回对映的物理页号。以下是具体的代码实现细节：



完成第一第二部分后，我们检验了disk\_manager\_test测试，测试结果如下。可以看出，我们成功的通过了这一part的测试。



2.3 LRU替换策略

Buffer Pool Replacer负责跟踪Buffer Pool中数据页的使用情况，因为Buffer Pool的大小有限，不可能存储disk中的全部信息；因而在Buffer Pool没有空闲页时，我们需要一定的算法决定替换哪一数据页。

在本部分，我们采用LRU替换策略，即最近最少使用替换策略；在此之外，我们也构想与实现了框架内为支持的随机替换策略，作为bonus，我们将在后续章节讨论随机替换策略的实现。

在本部分，我们主要需要实现LRUReplacer::Victim，LRUReplacer::Pin LRUReplacer::Unpin与LRUReplacer::Size四个函数，下面我们便给出其中的具体设计细节。

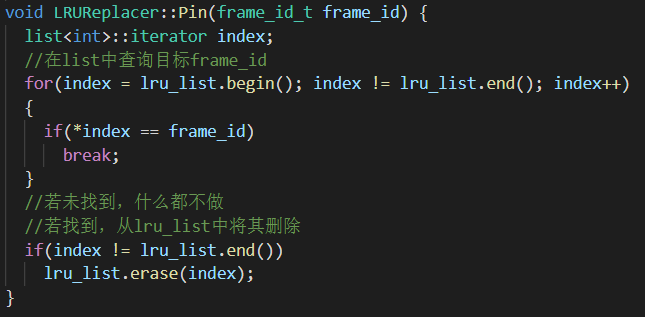
需要特别说明的是，在lru\_replacer.cpp文件中，因为对assert函数的使用需求，我们额外的向其中引用了<assert.h>头文件；为了实现这一替换策略，我们为lru\_replacer类增添了lru\_number（链表记录最大支持个数），lru\_list（链表记录具体数值顺序）两个成员。

接下来先具体分析LRUReplacer::Victim函数，该函数从lru\_list中取出一可被替换页，将对映的index赋值给frame\_id（frame\_id为逻辑页号经映射函数映射后获得的值，具体说明将在本模块下一部分）；若替换取值过程成功则返回true，反之则返回false。

LRU替换策略已为lru\_list进行了取出替换排序，因而在本函数的实现中，我们只需要取出lru\_list的最后一项即可。

接着是LRUReplacer::Pin 函数，其功能是根据输入的frame\_id参数，标记处于lru\_list中的数据页，使其不能被Victim函数获得；对映于现实中有用户正在使用该数据页的操作，可以维护数据记录的一致性。

在实现过程中，我们对lru\_list采用了最直接的线性搜索，只要发现对映frame\_id的数据页在lru\_list中，我们便把这个元素移出lru\_list；反之，则什么都不做。具体的代码设计实现见下：

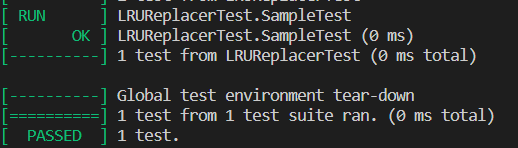


然后是LRUReplacer::Unpin函数，从函数名我们可以推断，此函数的功能与pin函数恰好相反。该函数功能为解除数据页的固定，即将输入参数frame\_id对映的数据页放入lru\_list中，表示其可以在必要时被替换。

在函数具体的实现过程中，我们依然是简单的遍历lru\_list列表，如果找到了对映的frame\_id，则不进行任何操作；如果frame\_id本身不处于lru\_list列表之中，我们就把输入的frame\_id参数放置于lru\_list的最前端，这也对映了LRU策略的实现。结合于我们的Victim函数，每次只取出lru\_list中最后的一个元素，从而最前端的元素往往最后被替换，这也就实现了最近最少替换策略，即最近使用最后被替换策略。

最后是LRUReplacer::Size函数，该函数旨在返回lru\_list当前所含的元素个数，我们只需要直接返回lru\_list.size()函数即可。

完成第三部分后，我们通过lru\_replacer\_test检测程序的正确性，测试结果如下。可以看出，我们成功的通过了LRU替换策略的测试。



2.4 缓冲池管理

在实现了LRU替换算法后，我们便可以实现整个BufferPoolManager结构了。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）[[2]](#footnote-1)。

其实在数据库系统中，所有的内存页面都是一个Page对象，其中大多数都是一段头部信息加上一段连续的实际存储数据。需要特别注意的是，在内存中的Page并不是固定，即在其的整个生命周期中，可能会对映于很多不同（物理页号）的数据页。因而我们也必须为每一个Page分配以一Page\_id加以区别，具体的实现结构我们将在后续讨论。

在本部分，我们需要实现BufferPoolManager::FetchPage、BufferPool-

Manager::NewPage、BufferPoolManager::UnpinPage、BufferPoolManager::

FlushPage、BufferPoolManager::DeletePage五个函数（BufferPoolManager::

FlushAllPages函数在本框架中好像不存在）。

首先，我们来解析一下page的主要结构，Page类主要含有 data\_、page\_id\_、pin\_count\_、is\_dirty\_四个成员。其中data\_记录了当前page中的具体数据，page\_id\_则记录了page当前状态下对映的唯一物理页标识符，pin\_count\_则记录了使用当前page的进程数量，is\_dirty\_则表示了当前page是否被写修改。

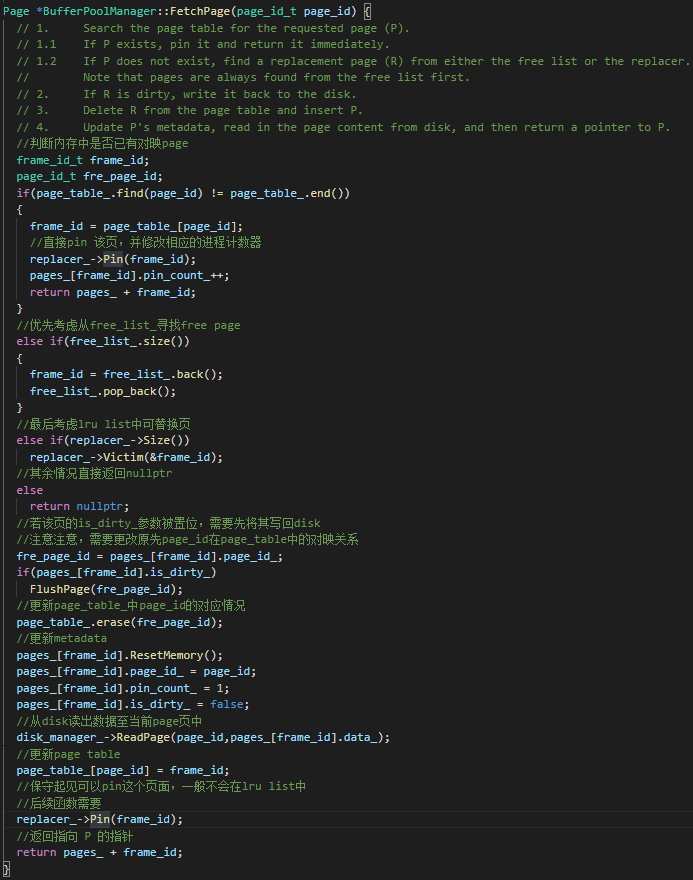
在介绍了Page的具体结构之后，我们还要介绍buffer\_pool\_manager类的具体结构，其主要包括pool\_size\_，pages\_，disk\_manager\_，page\_table\_，replacer\_，free\_list\_六个成员，其中pool\_size\_记录了buffer\_pool支持存储的最大page数量；pages\_则是buffer\_pool中记录的总有page集合；disk\_manager\_则表示前述部分实现的disk manager结构，可用于disk的管理；page\_table\_则是一个映射函数，可以把逻辑页号映射为特殊的frame\_id，replacer\_则是我们实现LRU策略时所需的结构，free\_list\_则记录着buffer pool中不在replacer替换器中的空位。

在了解了page和buffer\_pool的具体结构之后，我们先来分析BufferPoolManager::FetchPage函数。BufferPoolManager::FetchPage接收一个逻辑页号的参数，去获取与之相对应的数据页，若数据页不在内存中，则需要去disk中将其取出。

为获得对映的数据页，我们需要先去遍历内存，判断该数据页是否存在与内存之中。若存在内存之中，我们只需要将这一数据页pin固定后返回即可。若该数据页不在内存中，则我们优先考虑从free\_list中获取空闲的内存page；若free\_list不存在，则我们访问replacer寻求替换。

在获得了空闲page后，我们需要check原page的is\_dirty\_成员，若原page的is\_dirty\_成员被置位，则我们需要调用FlushPage函数（具体实现细节见后）将对映的内容写回disk；经过check操作后，我们需要相应改变内存中该page的记录信息，使之对映于需要fetch的逻辑页号对映的数据页。因此我们需要调用Page::ResetMemory函数重置部分信息，接着将page的page\_id对映到新fetch的逻辑页号，并把pin\_count重置为1（当前只有fetch进程使用了该page），最后重置is\_dirty\_成员；同时，我们需要更新page\_table\_中frame\_id与page\_id的对映关系。

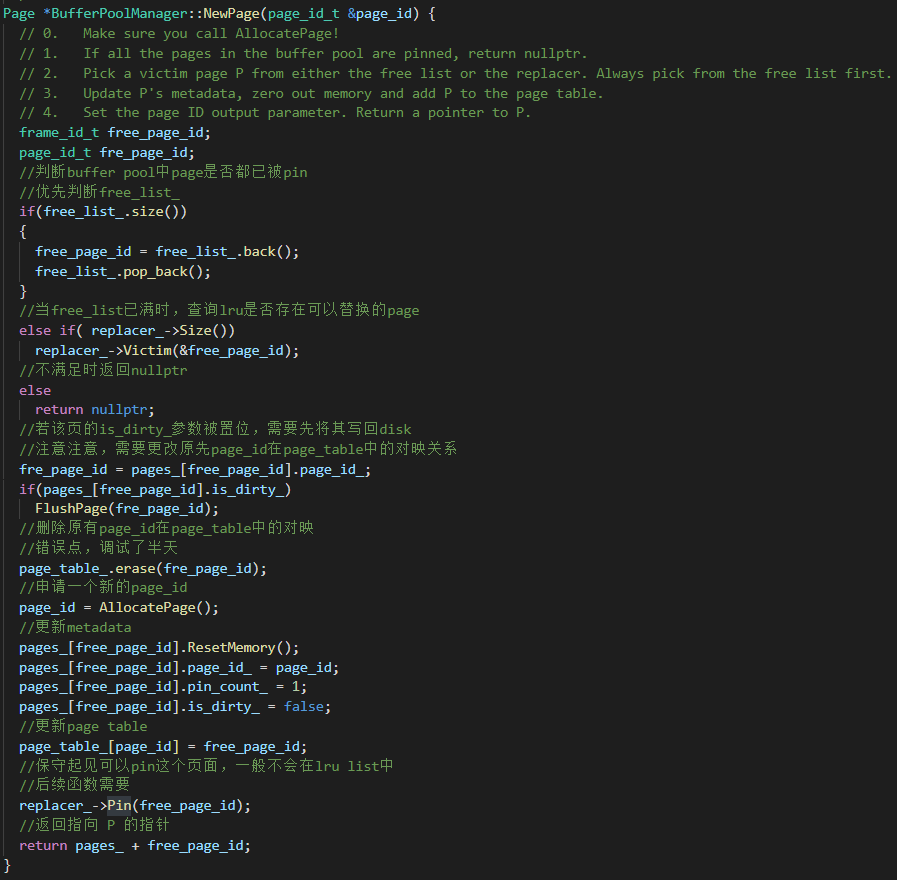
完成上述操作后，我们便可以使用DiskManager::ReadPage函数从disk读取数据至当前内存page中，并使用lru\_replacer::Pin函数将其固定。至此，我们便完成了FetchPage函数的所有步骤。以下是具体的代码设计实现：



接着分析BufferPoolManager::NewPage函数，该函数分配一个新数据页，并通过参数page\_id返回对映的逻辑页号。若分配成功则返回指向对映数据页的指针；若不存在空闲数据页或分配失败，则直接返回nullptr指针。

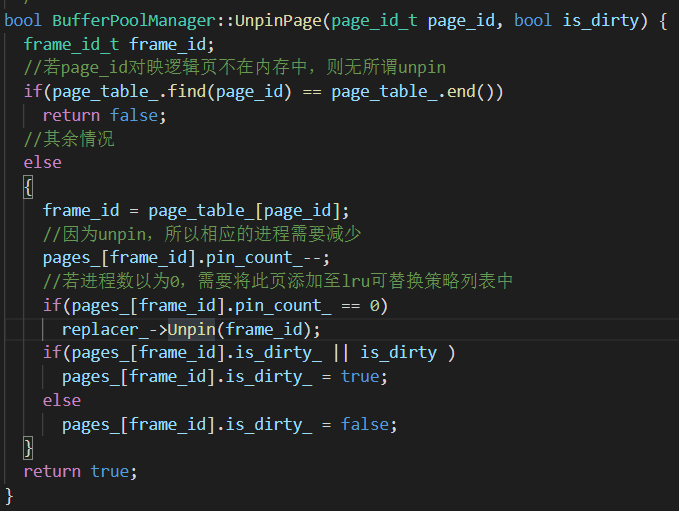
在寻找可以分配的page时，我们优先寻找free\_list列表，若其中不存在可以分配的内存页，则我们去搜索replacer替换器，寻找可以进行替换的page。若还是不存在，则函数直接返回nullptr，表示内存已满且所有page都在使用中，不允许替换。

若我们通过free\_list或replacer途径获得了可以替换或填入的内存页，同理于Fetch函数，我们需要先去check对映page的is\_dirty\_属性，若该页为脏页，则我们需要进行同Fetch的FlushPage操作。在此之后，我们同样需要修改page\_table\_的对映关系，pages\_数组中对映逻辑页的元信息，以及在最后对相应的内存页使用pin函数使其固定（具体实现细节与Fetch函数一致，在此便不再赘述）。以下是具体的代码实现细节：



接着我们继续分析BufferPoolManager::UnpinPage函数，该函数功能为取消一个内存数据页的固定。其接收需要unpin的逻辑页号，以及is\_dirty\_是否修改记录两个参数；显然，在unpin的时候，我们需要特别关注is\_dirty\_函数。

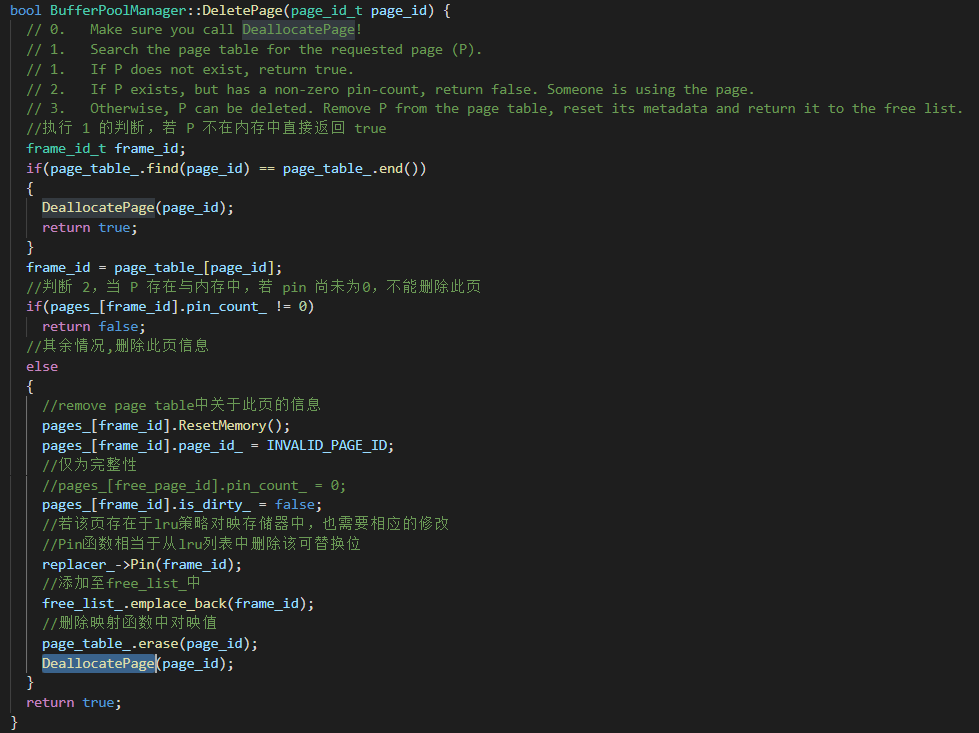
在函数的实现过程中，我们需要先去判断给定的逻辑页号是否在内存中，若不在内存中，则直接返回false表示unpin失败即可；若在内存中，我们也只能使page\_id对映的frame\_id在pages\_数组中相应的成员pin\_count作自减（可能该页被多个进程同时使用）；在此基础上，我们需要判断pin\_count成员是否为0，若其值已减少至0，则我们可以调用下层的lru\_replacer::Unpin函数，真正的释放该内存数据页，使其可以被替换。需要注意的是，无论该数据页是否可以被底层释放，我们都需要去更新其is\_dirty\_参数，使is\_dirty\_成员一直处于最新的更新状态（方便FlushPage函数的正确执行）。以下是具体的代码设计实现：



我们就继续分析BufferPoolManager::FlushPage函数，该函数接收一个逻辑页号为参数，并实现将这个逻辑页号对映的数据页转储到disk中。若转储成功，则返回true，反之则返回false。

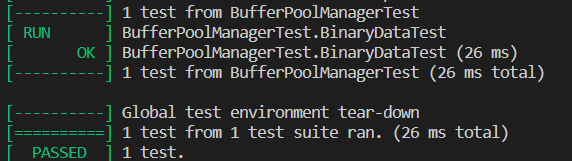
在具体实现的过程中，我们同样需要先判断给定的逻辑页号是否在内存page集合中，若不存在，我们直接返回false即可；若存在，且is\_dirty\_成员被置位，此时我们则需要调用DiskManager::WritePage函数将对映的数据写回到disk中，保证数据的一致性；同时，要把is\_dirty\_成员重置。若is\_dirty\_成员未被置位，则我们就不需要多余的操作，直接返回true即可。

最后是我们的BufferPoolManager::DeletePage函数，此函数接收一个逻辑页号作为输入参数，并彻底释放其对映的数据页；若释放成功，则返回true，反之则返回false。以下是具体的代码设计细节：



在具体实现函数的过程中，我们同样需要判断逻辑页号对映数据页是否在内存page集合中。若不存在于page集合中，则我们直接调用DeallocatePage函数即可，随后返回true表示成功释放（虽然好像也没做什么）。若逻辑页号对映的数据页存在与内存之中，则我们需要先判断该页的pin\_count成员是否为0；若pin\_count成员不为0，说明还有其他进程在使用该page，因而我们不能直接从内存中释放它，因而需要终止函数，返回false。另外，如果pin\_count成员已被减为0，我们则可以直接修改（重置）pages\_数组中对映的记录信息，如将pages\_中page\_id成员直接赋为INVALID\_PAGE\_ID，变成无效数据；将is\_dirty\_成员重置（因为本函数是释放一内存数据页，因而不需要将数据页的信息写回至disk，这也给予上层根据实际情况调用的灵活性）等等。最后，我们可以对逻辑页号对映的frame\_id进行lru\_replacer::Pin函数操作，等价于将其从LRU替换策略的lru\_list中移出（不能被其余分配数据页替换）。在此之后，我们更新page\_table\_中page\_id的对映关系，并调用DeallocatePage函数释放对应的数据页。

完成第四部分后，我们通过buffer\_pool\_manager\_test检测程序的正确性，测试结果如下。可以看出，我们成功的通过了缓冲池管理的测试。



1. RECORD MANAGER模块的设计

3.0 总体概述

在minisql的设计中，Record Manager负责管理数据表中所有的记录（精确到行），它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。

对于record记录而言，我们存有以下几个相关概念（对映于数据库中表的构成）：1. 列，用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等；2. 模式，用于表示一个数据表或是一个索引的结构，其中一个模式由一个或多个列组成；3. 域，对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；4. 行，与元组的概念等价，用于存储记录或索引键，其中一行由一个或多个域组成。

3.1记录与模式

在minisql的数据表存储中，我们为了方便统一处理，都需要提供对映的函数将上面提到的行，域，列，模式四个类型转化为字节流对象，以写入数据页中。另外，为了能够从磁盘中恢复这些对象，我们同样需要能够提供一种反序列化的方法，把字节流对象转换为我们需要的对象类型。

总而言之，序列化和反序列化操作实际上是将数据库系统中的对象（包括记录、索引、目录等）进行内外存格式转化的过程，前者将内存中的逻辑数据（即对象）通过一定的方式，转换成便于在文件中存储的物理数据，后者则从存储的物理数据中恢复出逻辑数据，两者的目的都是为了实现数据的持久化。

为了区分上述不同的对象类型，我们引入MAGIC\_NUM加以区分（MAGIC\_NUM一般位于元信息中），它在序列化时被写入到字节流的头部并在反序列化中被读出以验证我们在反序列化时生成的对象是否符合预期。

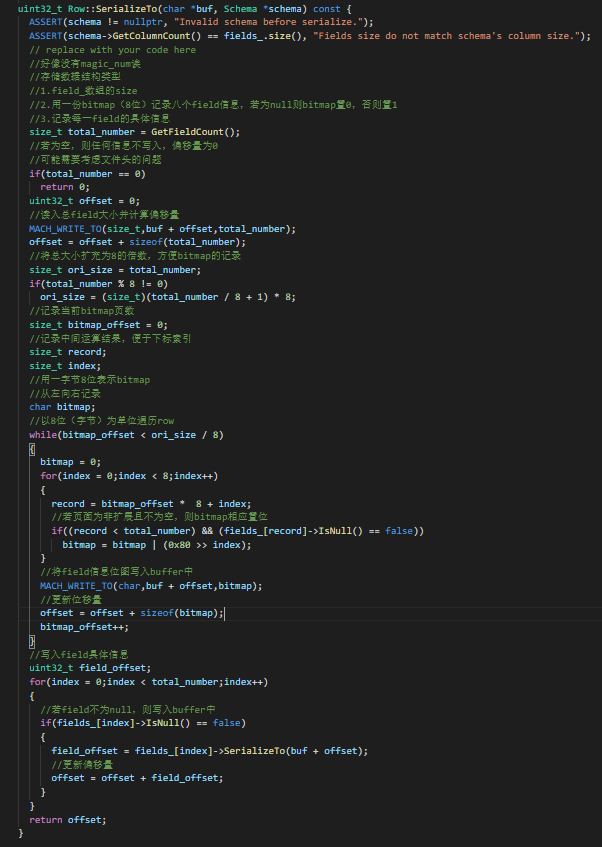
在本部分，我们需要实现Row::SerializeTo、Row::DeserializeFrom、Row::

GetSerializedSize、Column::SerializeTo、Column::DeserializeFrom、Column::

GetSerializedSize、Schema::SerializeTo、Schema::DeserializeFrom、Schema::

GetSerializedSize九个函数，下面我们一一进行具体分析。

首先我们分析Row::SerializeTo函数，该函数旨在将row对象转换为字节流方便存储，其接收buffer、模式两个参数，并返回buffer在存入row对象对映的字节流时产生偏移的大小。以下是具体代码的设计实现：



在实现row对象到字节流的转换时，我们先check输入的schema指针是否为空，若为空，则需要终止程序并显示相应的错误；同样，我们也要判断当前行对象的域总大小是否与模式中的列宽一致，若不一致，我们也不允许信息的填入。另外，本框架下的row对象好像没有对应的magic\_num。

在判断正确头信息过后，我们只需要计算row对象各成员的占位宽大小之和即可，用以记录在row对象填入的过程中buffer产生的偏移。而row对象只有rowid（下一部分详细介绍）与fields（记录当前row对象下各个field的数值）两个成员，其中rowid并不需要额外的记录，因为从数据页中读取数据这个过程，便已经将rowid唯一确定；因而对于row对象而言，我们只需要记录field\_数组。而对于field\_数组而言，我们需要先用MACH\_WRITE\_TO函数记录field\_数组的大小，接着，我们也不直接记录对映的field数据，而是使用类似于bitmap的结构，先记录每一field的具体填入情况，用于表示数据库对映表中null元素的存在。

因为8bit更加符合C语言的使用，因而在记录field bitmap前，我们需要扩展field number至8的倍数，这样每8位可以用一个char类型的bitmap进行记录。对于扩展序列，我们只用与先前bitmap同样的方法，使用1bit记录一个field是否为null；对于扩充部分与null元素直接对映取0即可，我们便可构建相应的field\_数组bitmap。最后，我们再用MACH\_WRITE\_TO写入具体的field数据即可，需要注意的是，对于null数据，我们无需再次处理。

接下来我们分析Row::DeserializeFrom函数，该函数旨在实现SerializeTo函数的逆过程，即把相应的文字流文件转变为对映的row对象，接收buffer、schema两个参数，并返回读出数据后buffer的偏移量。

在实现该函数时，我们需要进行与SerializeTo函数同样的判断，若schema指针为null或fields\_数组不为空时，我们需要暂停函数并给出错误信息。在判断之后，我们可以使用SerializeTo函数的逆过程，用MACH\_READ\_FROM函数读取buffer中的信息，并直接赋予row对象中相同的类型。在这一部分，我们需要额外的空间来记录bitmap的信息，在这里我简单的使用了一个二维数组，这可以大大简便我们后续的操作。

读取完总数据大小及对映bitmap分布后，我们开始读取真实的记录数据。对映于bitmap，我们需要对写入的数据分类；若本身bitmap对映值为0，则说明数据本身为null，这时我们在调用Field::DeserializeFrom函数解析时，需要将参数设置为false；相反，若数据本身不为null时，则需将对映参数设置为true。除了把读取数据负载在fields\_数组的末端，我们需要在每次读取前判断此次读取数据的类型，这时我们就要用到参数schema指针，schema指针中每一column的类型便是我们应该读到的数据类型。在这里我们可以用Schema::GetColumn函数获得当前列，并用GetType函数获得其对映的数据类型，并将这个类型与TypeId枚举类型进行比对，从而介导相应的Field构造函数。以下是具体代码的实现：

最后，对于row对象而言，我们可以谈谈Row::GetSerializedSize函数，该函数接收schema为唯一参数，并返回记录当前row对象需要的buffer偏移量。

在函数实现的过程中，我们同样需要先判断schema指针参数不为null，并且schema对映的column数量与fields\_.size()保持一致；接着，我们只需要计算需要存储的数据对映的类型字节数即可；当然，对于fields\_类型而言，我们需要额外的bitmap来记录数据是否为null。

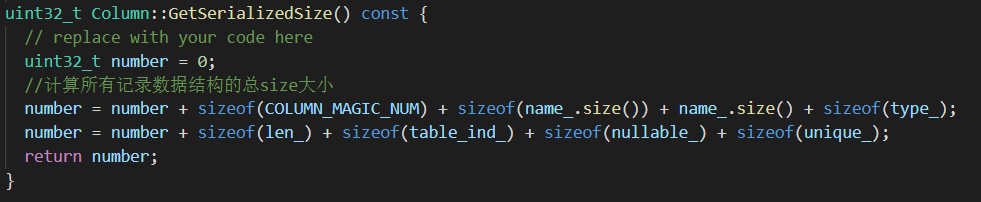
解析完row对象部分的函数，我们继续解析column对象的相关函数，Column::SerializeTo、Column::DeserializeFrom与Column::GetSerializedSize。

首先看Column::SerializeTo函数，与row对象中Row::SerializeTo函数类似，只接收buffer一个参数，并返回buffer的偏移量一个整数值，我们只需要根据column类型中成员的数据类型记录偏移量即可。在column类中，需要记录的成员有：magic\_num（魔数），name\_（列名称），type\_（列类型），len\_（支持记录的长度，本身为整数类型），table\_ind\_（对映表中的位置--偏移量），nullable\_（数据是否可以为null）与unique（数据是否唯一），其中只有name\_需要特殊考虑。对于name\_的存放，我们需要先存放name\_的长度，再一一记录name\_中出现字节即可。

相同的顺序，我们继续分析Column::DeserializeFrom函数，类似于row的同名函数，该函数接收buffer与column两个参数，并返回读出数据后buffer对映的偏移量。

在具体的函数实现中，我们再次判断了非空，并保证了buffer中记录的魔数对映于column类型，否则数据读取失败，我们必须中断函数，显示报错信息。接下来，我们根据存放数据的顺序，一一读出字节流，并将其转变为对映的数据类型。当然，和SerializeTo函数相同，我们只需要对name\_对象进行字符一一读取的操作即可；另外类型都可以通过MACH\_READ\_FROM函数直接读取获得。最后，我们需要用上述获得的一系列数据重新生成一个column对象并赋给输入参数。

最后也是分析Column::GetSerializedSize函数，没有输入参数，返回将column对象装入buffer后会产生的计算偏移量。以下是函数的具体实现细节：

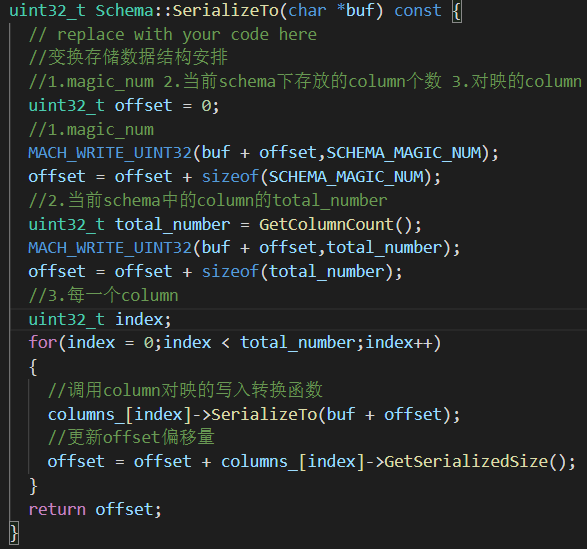


对于偏移量的计算，我们只需要根据将会放入buffer的类型进行对映的字节叠加即可，返回这些类型变量的字节总和就是我们要求取的偏移量size。

最后我们分析schema对象，其中我们需要实现Schema::SerializeTo、Schema::DeserializeFrom与Schema::GetSerializedSize三个类似于前述对象的函数。

也是按顺序先看Schema::SerializeTo函数，该函数接受buffer为唯一参数，返回载入schema对象后buffer指针的偏移量大小。

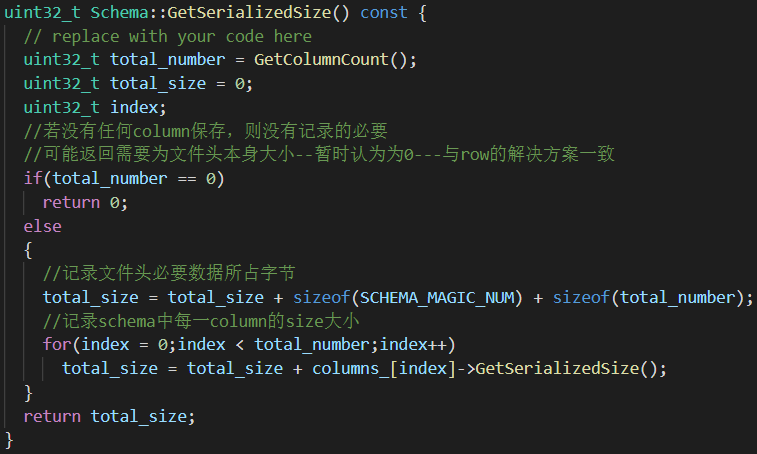
类似于先前的两种对象，在该函数中，schema需要记录1.magic\_num，2.当前模式下的column总数，3.每一个column；当然，在记录每一个column时，我们可以简单的调用Column::SerializeTo函数，而对于它们各自相应的偏移量，我们也可以使用GetSerializedSize函数简单获得。以下是具体代码的实现细节：



接下来是Schema::DeserializeFrom函数，该函数接受了buffer和schema两个输入参数并返回读出相应schema对象数据后buffer的偏移量。

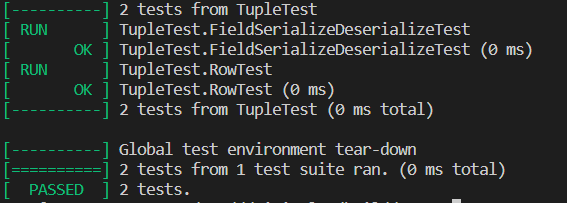
一如既往的，我们需要判断接收的schema指针是否为空、魔数类型是否与schema对象一致。在此基础上，我们执行SerializeTo函数的反向操作，使用MACH\_READ\_FROM函数将各个数据一一取出并转换为相应的类型；注意，column对象需要一一取出与保存。最后我们建立一个新的schema类接受读出的数据，并使用拷贝函数将其赋值给输入参数的引用，这样我们便完成了schema类型的读出工作，返回其对映的偏移量即可。

最后是Schema::GetSerializedSize函数，类似于row对象和column对象，该函数需要返回若将当前schema对象写入buffer中时会产生的偏移量大小。以下是具体代码的设计实现：



在具体实现中，我们只需要记录各个需要写入buffer的变量的类型字节大小即可，其累加和就是我们要得到的结果。

完成上面三个对象的字节流序列化转换后，我们通过tuple\_test检测程序的正确性，测试结果如下。可以看出，我们成功的通过了记录与模式的测试。



3.2 通过堆表管理记录

3.2.1 RowId

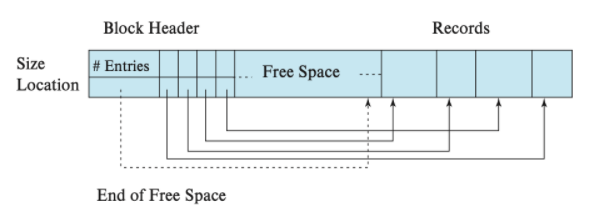
对于数据表中的每一行记录，都对映着一个唯一的标识符RowId。RowId同时具有逻辑和物理意义，在物理意义上，它是一个64bit整数，是每行记录的唯一标识；而在逻辑意义上，它的高32位存储的是该RowId对应记录所在数据页的page\_id（逻辑页号），低32位存储的是该RowId在page\_id对应的数据页中对应的是第几条记录[[3]](#footnote-2)。

RowId可以在索引时便于与叶节点中的存储键值形成一定的映射关系，从而实现通过索引键key的查找，得到相应记录对映的RowId，从而在数据页中定位到相应的数据；另一方面，在堆表的实现中，RowId中存储的逻辑信息可以快速地定位到其对应的记录位于物理文件的哪个位置。

3.2.2 堆表

堆表是一种将记录以无序堆的形式进行组织的数据结构，不同数据页之间通过双向链表链接，而堆表就依赖于RowId的定位。

堆表的具体结构见下，堆表中每个数据页都由表头、空闲空间和已插入数据三部分构成。其中表头在页中从左往右扩展，记录PrevPageId（前一pageid）、NextPageId（后一pageid）、FreeSpacePointer（空闲位置定位指针）以及每条记录在TablePage中的偏移和长度；插入的记录在页中从右向左扩展，每次插入记录时会将FreeSpacePointer的位置向左移动。



在堆表中插入一条记录时，我们只是简单的使TablePage构成的链表依次查找，直到找到第一个能够容纳该记录的page；当需要从堆表中删除指定RowId的数据时，我们只是先通过mask进行标记，直到后序某个时间段再对其进行逻辑删除。对于更新操作，如果TablePage可以容纳下更新后的数据，我们便直接更新；如果不能容纳，我们则根据具体的问题返回错误信息。另外，在本部分我们还需要实现tablepage相应iterator的设计。

本模块需要实现TableHeap:InsertTuple、TableHeap:UpdateTuple、TableHeap:

ApplyDelete、TableHeap:GetTuple、TableHeap:FreeHeap（本框架中好像不存在）、TableHeap::Begin、TableHeap::End七个函数以及TableIterator的设计。

我们首先来看TableHeap:InsertTuple函数，本函数接收row对象（一行数据），与txn指针（暂无作用）为两个参数，返回插入语句本身的运行结果；插入成功则返回true，失败则返回false。以下是具体代码的设计实现：



在尝试插入行数据时，我们需要先判断该数据是否超过page大小，若单行数据超过了page大小，我们直接返回false。接着我们从tablepage的第一页开始遍历，寻找第一个可以装载下该行数据大小的page；在寻找到可能的page后，我们需要将它从disk中取出，并使用tablepage::InsertTuple函数尝试插入。若插入成功，因为disk取新页的过程本身会固定住页面，因而此时我们需要将对映的page Unpin；若插入因为其他的原因失败，我们同样需要Unpin当前page，因为我们即将搜寻下一个page，搜寻下一个可能的page时，我们只需使用框架已经实现的GetNextPageId函数即可。

若在搜寻的过程中我们得到page\_id为INVALID\_PAGE\_ID无效值（当前数据页都无空间存放），此时我们则需要开辟新的数据页来存放对映数据。同样我们需要使用tablepage中的buffer\_pool\_manager（我们在第一模块实现）的NewPage函数将对映的逻辑页号数据页取至内存中，并转化为TablePage的类型；当然，如果分配本身失败，我们也直接返回false；若分配成功，我们可以调用table\_page::Init函数初始化新分配的page。

再开始一系列操作之前，我们可以使用WLatch函数为当前界面上锁，保持写过程的一致性；然后我们可以调用框架已经实现的tablepage::InsertTuple函数，向page插入我们的数据，在插入完成之后，需要使用WUnlatch函数为写操作解锁。因为在Fetch操作时我们默认锁住了界面，所以在最后我们需要使用buffer\_pool\_manager::UnpinPage函数对其解固定，保证使用的一致性；最后我们需要使用框架已经实现的SetNextPage函数更新数据页元信息，使我们新开辟的数据页加入到table\_page的双向链表中。最后，也不能忘记将fetch的当前页通过Unpin函数解固定。

接着是TableHeap:UpdateTuple函数，该函数接受需要更新的新对象row，更新的目标地址rowid与txn指针三个参数，返回此次操作的成功与否。

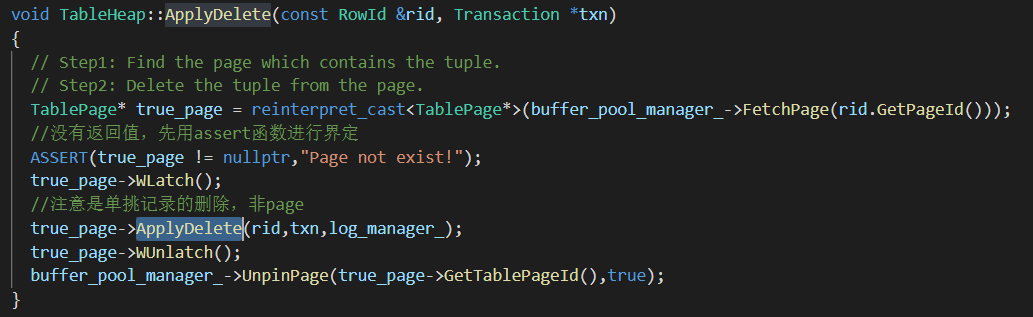
当然，我们需要先通过判断保证输入row与rowid对线的有效性，如果接收到了nullptr指针或INVALID\_ROWID，我们可以直接返回false并退出。在确保有效性后，我们可以通过rowid的前32位，从内存读取对映的数据页并转化为table\_page的格式，并通过已有的rowid信息新建一row对象。同时，我们可以使用GetTuple函数（后续会给出实现细节）读出对映的数据，接着我们便为当前页上写锁，使用TablePage::UpdateTuple函数尝试更新数据；注意，为了区分相同返回值下产生的不同错误，我们对架构实现的TablePage::UpdateTuple函数进行了一定的更改。

更改后，若函数返回0，则表示更新成功；若函数返回1或-1则表示对映tuple已被删除或slot\_num数值已越界；若函数返回2，则表示当前page页位置不足，不能发生替换。而本函数只需要根据返回值是否为0，作出true返回或false返回的判断。因为不清楚高层想要的架构支持模式，本函数只是将返回值进行分类，而并没有采取相应的操作。以下是具体代码的设计细节：



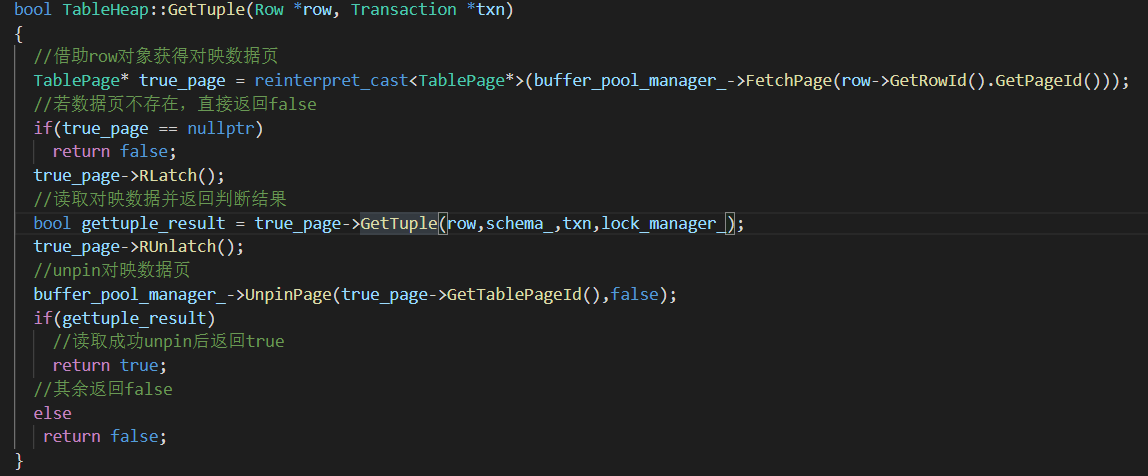
接着是TableHeap:ApplyDelete函数，其接收rowid与txn指针两个参数，根据给定的rowid信息，从物理意义上删除其，根据操作的结果返回true or false。

一如既往的，我们需要根据rowid从disk中读取出相应数据页，同样，我们需要判断这个数据页的有效性；在保证有效性的基础上，我们只需要为当前page上写锁，并调用TablePage::ApplyDelete函数进行对映位置数据的删除，最后在对映于fetch操作，对对映数据页进行解固定即可。以下是具体函数的代码实现细节：



接下来我们继续分析TableHeap:GetTuple函数，该函数接收对映的row对象（取数据依赖于rowid的记录功能）与txn指针，根据操作的结果返回true or false。

类似于ApplyDelete函数，在本函数中，我们同样需要取出对映数据页，并进行有效性判断；区别于上写锁，在获得row对象数据时，我们只需要使用读锁即可。另外，我们只需要使用TablePage:: GetTuple函数从page中取得对映数据即可，最后，不要忘记Unpin解固定相应界面即可。以下是具体代码设计细节：



最后是TableHeap::Begin与TableHeap::End函数，这两个函数主要是为连接迭代器而实现，其中end不接受任何参数，begin接受一txn指针，两者均返回TableIterator对映的迭代器。

其中对于Begin函数而言，我们只需要获得table\_page对映的第一个数据页逻辑页号，进而我们可以从disk或内存中找到相应的page并转变为TablePage类型。接着，我们只需要调用TablePage::GetFirstTupleRid函数，就可以获得对映逻辑页下的第一个rowid（这需要与迭代器相契合），最后，解固定因fetch函数带来的固定状态；构造一新迭代器对象返回即可。

而对于end函数而言，直接返回rowid为INVALID\_ROWID对映的新构造迭代器即可。

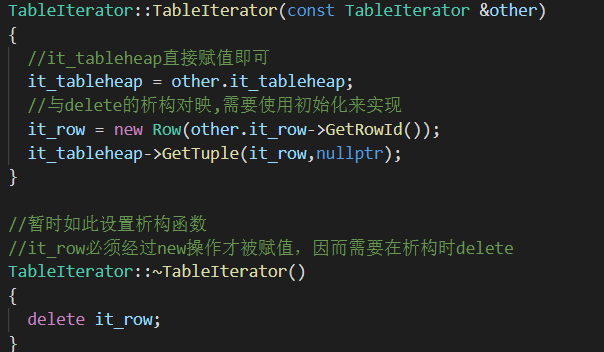
完成了table\_heap的实现，接下来我们可以继续分析其对映迭代器的实现细节。

首先，我们需要为迭代器增加合适的类对象，根据要求，迭代器可以遍历所有的row对象数据，因而我们为其增添了TableHeap\* it\_tableheap成员与Row\* it\_row成员，其中it\_tableheap可以帮助我们跨页的获得row对象数据，而row对象则记录当前的指向。

对于构造函数而言，我们需要接受tableheap和rowid两个参数，具体的代码实现见下：

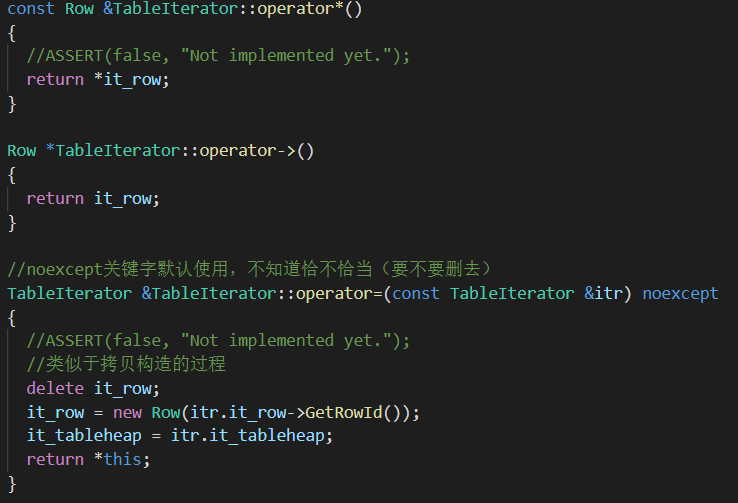


首先，我们也是判断传入参数，特别是tableheap的有效性，如果为空指针，则直接返回nullptr。接着，我们只需要做相应的赋值操作即可；注意，构造函数中指针对象的赋值需要用new特别考虑。

接着是拷贝构造函数，类似与一般拷贝构造函数，我们只需要在考虑到指针特殊性的情况下进行拷贝构造即可。然后析构函数也是对应于一般的析构函数，只不过其中需要考虑到new row对象后必要的delete释放，防止内存泄露。以下是其对映的具体代码实现：

接下来是运算符的重载，我们首先关注“==”；重载“==”时，我们只需要考虑迭代器（当前类）相等的判断条件即可，根据给出的note，我们选择了类中当前row对象的数值作为相等判断的标椎；相对应的“!=”运算符的重载也可以基于此实现。以下是具体代码的设计细节：

随后是“\*”与“->”的重载，对应于遍历所有记录的要求，我们迭代器返回的类型，只需要满足于row指针即可，因而在“\*”符号的重载中，我们直接返回\*row对象即可，而对于->而言，我们则直接返回row指针成员。而对于“=”赋值符号的重载，我们也只需要借鉴于拷贝构造函数，进行对映成员的分配与赋值即可。以下是具体的函数实现：



最后是重载自增操作，首先我们考虑前置自增，后置自增的实现可以依赖于其完成。

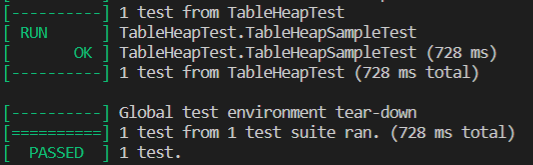
在实现前置自增的过程中，我们同样需要判断row对象中rowid的有效性，如果迭代器已经读至table\_page的尾部，这时对应于INVALID\_ROWID，我们就不能继续实现自增操作了。判断结束后，我们需要根据row对象获得其rowid，从而获得当前的page\_id，并把对映的数据页保存至内存中。读数据前，我们需要位改页上读锁。通过GetNextTupleRid函数尝试，如果下一rowid还在本页，则我们已经获得新的rowid，因而可以生成对映的row对象并返回；如果下一rowid已不再本页，那我们需要从本数据页对映的逻辑页号出发，或者依赖于GetNextPageId函数，去搜寻下一可能的数据页。

在搜寻的过程中，我们只需要判断一个新的page的第一个rowid（因为原有为最后一项rowid，因而其next rowid一定记录于某一page的首项）是否符合我们的需要，如果符合，则搜寻过程结束；如果仍不满足，则我们需要继续遍历。若最后获得的page\_id变成了INVALID\_PAGE\_ID，说明我们已经读完了所有的记录，因而我们可以返回相应的INVALID\_ROWID数值用于构造无效row对象，这样无效的iterator对象恰好对映了table\_heap访问时end函数的要求。

而对于后置自增而言，我们只需要先拷贝构造原有的类，再对原有类进行前置自增操作，最后返回原有类的拷贝即可。以下是具体函数代码的实现细节：



完成上面TableHeap类的具体实现与对映的Iterator后，我们通过table\_heap\_test检测程序的正确性，测试结果如下。可以看出，我们成功的通过了通过堆表管理记录的测试。



1. 总结与展望

只能说minisql还是有一定工作量的，在没有任何头绪之前，总感觉基于框架实现minisql的工作遥不可及；先别说能不能真正理解框架，去尝试实现一部分功能，单单环境的缺陷对于我这种小白而言，便是一道天堑。当然，也是感谢实验指导书中细心的教导，我成功的安装了wsl，并开始了我的minisql实现之路。

在本次合作中，我主要负责最后底层部分的实现，也就是关于buffer\_pool\_manager与table\_heap的内容，在实现的过程中，也是发现这两部分有很大的联系，因而这样连续的设计要求，也对我编程的连续性，特别是理解与填充方面，有了很大的帮助。

一开始对于框架还是仅存畏惧，哪怕最简单的一个热身---实现一个bitmap，我都不能很好的理解其中奥秘；但到后来我发现，我所畏惧的，都来源于我对框架的未知，其实minisql给定的框架已经实现了绝大部分的主要内容，我们要做的，更多的是拼接一些已有函数，从而实现新的功能函数。因而，在读懂框架的前提下，一步一个脚印慢慢编写、慢慢向前走，也没有想象中的那么复杂。

当然，最具有挑战的，翻到成了一些类的设计，如tablepage对映iterator的成员设计，这对于尚未掌握C++的我又提出了一大挑战；但细细品味实验指导的文字，去发掘一些函数实现的返回值，就能发现它在每每细微之处提醒你设计的方向。当然，最后我也是实现了这些设计，并根据自己的理解，简化了不必要的成员，把一个类作简。

从一开始对minisql的恐惧，到现在一个人完成了第一第二模块，还是颇具成就感，也算是对数据库学习的一个交代吧。当然，我深知能完成这一模块，极大程度上还是依赖于框架的存在，要是连框架都没有，仅仅依赖于需求的表达，我估计自始至终都处于minisql的萌芽阶段。

除此之外，我也要谢谢一些同学的帮助，特别是在minisql开始编写的初期，如果没有他们的指导，我也不能马上理解我的任务所在，只能傻傻地顶着代码发呆。

或许过程是煎熬的，但挥洒汗水后总是收获满满。

1. 来源于：语雀 https://www.yuque.com/yingchengjun/minisql/ldgod4 [↑](#footnote-ref-0)
2. 来源于：语雀 https://www.yuque.com/yingchengjun/minisql/ldgod4 [↑](#footnote-ref-1)
3. 来源于：语雀 https://www.yuque.com/yingchengjun/minisql/cnz3sh [↑](#footnote-ref-2)