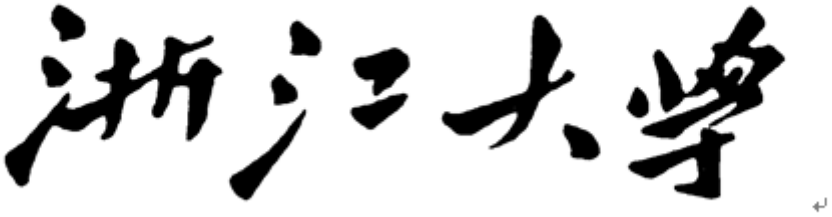

MiniSQL

学号：3220104519
姓名：蔡佳伟

零、实验概述

本次MiniSQL程序我是一个人完成的，共完成了前五个部分和clock_replacer的一个bonus。在此我将会详细介绍各个部分。

一、DISK AND BUFFER POOL MANAGER

bitmap_page

在该部分我通过位运算操作来进行置位、复位，因为位运算的操作非常便捷而又易于理解其中的原理。以下是关键代码。

```
bytes[next_free_page_ >> 3] |= (0x80 >> (next_free_page_ & 0x07));
  // 置位操作：左边是字节 右边是算位数 eg：比如将第1234个位置1 ，字节序：1234 >> 3 = 154;
位序：0x80 >> (1234 & 0x07) =
  // 2 ，那么 1234 放在 M 的下标 154 字节处，把该字节的 2 号位（0~7）置为 1
 bytes[page_offset >> 3] &= ~(0x80 >> (page_offset & 0x07));
 // 复位操作
```

disk_manager

在该部分，我实现了`AllocatePage()`,`DeallocatePage(logical_page_id)`,`IsPageFree(logical_page_id)`,`MapPageId(logical_page_id)`,四个函数。在`AllocatePage()`中，读出每个extent的步骤为如下操作，

```
ReadPhysicalPage(i * (BITMAP_SIZE + 1) + 1, buf);  // 把对应分区的bitmap读取出来
```

因为每个extent的大小为BITMAP_SIZE+1，同时本extent还有一个位图页，所以还要再加一。同时如果所有分区已满，需要新开一个分区。

对于逻辑页面和物理页面的转换，

```
return logical_page_id / BITMAP_SIZE + 2 + logical_page_id;
```

```logical_page_id/BITMAP_SIZE```+1是该页前面的位图页数量，又因为逻辑页号从1开始，所以再加1。

LRU_Replacer

我使用数组和整数来记录访问次数等信息，需要的变量和要求如下：

```
vector<int32_t> frame_used;    // 记录缓冲区每一页帧的访问次数
  vector<bool> frame_isPin;      // 记录缓冲区每一页是否被pin
  size_t number_unpined_frame;  // 记录缓冲区有多少页可以替换
  size_t replacer_size;          // 缓冲区最大容量
```

其中`frame_used[i]`为`-1`表示该页是不在缓存区中不能被选择的空页。对空页的unpin相当于向缓存区添加一个新页，空页不能被pin，也不能被替换（还没有加入缓存区中）。

在`Victim`函数中，被选择的条件是不是空页，unpin，且访问次数更少，选中某节点后需要设置为pin。

`Pin`和`Unpin`函数只需要修改变量的值即可。需要注意的是，如果一个页已经被`Victim`，那么`pin`这一页应该无事发生。这里我在`victim`函数加入了以下处理，可以先把`frame_used[i]`设置为`-1`，因为该页之后可能会被删除，删除之后为空页。

CLOCK_Replacer

我使用一个数组，每个元素含有两个bool值，来记录每个页面信息，同时使用`clock_hand_`来记录时钟周期。

其中第一个bool值表示该页是否被pin，为false表示被pin，为true表示unpin。一开始所有页都被pin，(可以类比于上一部分的空页)，所以都置为false。第二个bool值表示时间周期到了没有，如果轮到这个frame_id时，该变量为true，那就置为false，如果为false（当然第一个变量必须unpin)，代表该页可以被换出，直接在`Victim`中替换并更新第一个变量为false。

所以pin是置两个变量为false，(被pin+被轮到)，unpin是置两个变量为false，(unpin+未被轮到)，这是为了确保刚刚unpin的数据页不会立刻被替换。同时需要注意每次`Victim`需要增加clock_hand，如果该变量不能取出完成替换，则继续增加clock_hand。

buffer_pool_manager

内存池管理模块中，`FetchPage`中，如果该页存在于内存中，我们就pin住，增加访问次数一次，如果不在，如果内存池中没有空页，就要获取一个可替换页进行替换同时pin住，如果内存池中有空页，就使用这一页并且pin住。同时需要更新空页信息，是否dirty等信息，如果要替换页且被替换页是脏的，需要`disk_manager_->WritePage(replace_page_id, pages_[frame_id].GetData());`写回磁盘，来保持数据的一致性。

`NewPage`中，我们也要判断内存中有无空页从而进行操作。

二、RECORD MANAGER
```

序列化和反序列化

在磁盘中数据需要持久性地存储，需要转化成字节流保存起来，所以需要序列化和反序列化。在

table_heap

TableHeap由若干个TablePage组成，不同页之间通过链表连接（即next_page_id）。每个TablePage上会有若干个元组（即Row）不同的Row有其对应的唯一的RowId.而每个Row会有若干个Field,相当于数据库中一条记录，记录会包含若干属性的取值。

对于InsertTuple操作，首先我判断序列化后的大小是否超过了一页能放下的大小，如果是，应该直接返回false。随后取出堆表中的第一页，对于每一页，我都尝试利用当前页的InsertTuple操作（TablePage），如果成功，可以直接返回 true。否则需要继续取下一页。如果已经达到了当前堆表的结尾，即当前堆表所有页都无法放下这个元组，那我需要通过buffer_pool_manager新建一页并完成相关初始化操作，再将元组插入到这页上。

table_iterator

堆表迭代器是从第一页的第一个元组开始，依次遍历堆表中的所有元组。 在设计迭代器时，首先这个类肯定需要对应的table_heap指针。此外还需要知道当前指向哪个元组，因此，我保存了now_page_id和now_row分别表示当前的页号以及当前元组的指针，同时这样也便于后面的运算符重载。需要注意的是，这里不能存当前页面的指针，否则在我们到下一页之前这一页就相当于一直被pin住了，降低了资源的利用率。

除此之外，值得注意的是，每次根据 pageid 取出数据页后，在对页的访问结束后需要unpin这页，并根据是否有对这页进行修改传入dirty page。我的经验是通常情况下如果有FetchPage，那就需要Unpin。后面的B+树以及其他部分均是如此，会有对应的CheckAllPinned函数来检查是否所有的数据页都被取消固定了。

三、INDEX MANAGER

b_plus_tree_page

中间结点`BPlusTreeInternalPage`不存储实际的数据，它只按照顺序存储m个键和$m+1$个指针（这些指针记录的是子结点的`page_id`）。由于键和指针的数量不相等，因此我需要将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。在任何时候，每个中间结点至少是半满的（Half Full）。当删除操作导致某个结点不满足半满的条件，需要通过合并（Merge）相邻两个结点或是从另一个结点中借用（移动）一个元素到该结点中（Redistribute)来使该结点满足半满的条件。当插入操作导致某个结点溢出时，需要将这个结点分裂成为两个结点。

而叶结点`BPlusTreeLeafPage`存储实际的数据，它按照顺序存储m个键和m个值，其中键由一个或多个`Field`序列化得到，在`BPlusTreeLeafPage`类中用模板参数`KeyType`表示；值实际上存储的是`RowId`的值，它在`BPlusTreeLeafPage`类中用模板参数`ValueType`表示。叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。

需要注意的是`BPlusTreePage`内存的大小固定是`4096Btye`，它其实是`Page`的`data_`成员，Page是物理内存中存储真实数据的一层封装，保留了一些pin_cout, dirty_flag信息。所以当我们从`BufferPoolManager`中`FetchPage`后，要用`reinterpret_cast`将`Page.GetData()`强转为需要的`BPlusTreePage`类型。

b_plus_tree_index

`leafpage`的`size`最大只能到`maxsize-1`，到`maxsize`时立马会发生分裂。

而`internalpage` size最大可以到`maxsize`，在下一次插入时，如果像`leafpage`一样插入之后再分裂的话会溢出，可能会造成无法预知的错误。所以要提前准备另一块足够的空间，先插入，再拷贝。

每一次从`bufferpool`中`fetch`一个`page`，最后都要`unpin`还回去，如果fetch，unpin不匹配，那么这个页就永远固定在`bufferpool`中无法驱逐了。在每次插入，查找，删除完成后，`bufferpool`可以写一个函数检查一下`page_id`0的页`pin_count`是否都为0。

删除时，当页的大小 < `minsize`时，可以选择向左/向右借或者合并， 先往左边或者先往右边，或者先借还是先合并顺序都不重要。只要保证满足发生借或者合并后，节点与兄弟节点大小满足大于等于`minsize`即可。假设我们找到节点的sibling，如果`sibling size` > `minsize`，那么直接借一个，不会影响父节点的大小。如果`left sibling size` = `minsize`，合并两个节点，总是删除合并节点中的右边节点，然后再父结点中递归删除右节点的索引。

index_iterator

与堆表实现迭代器类似，这里就不一一赘述。

四、CATALOG MANAGER

Catalog manager部分主要为执行器Executor提供了公共接口以供执行器获得目录信息并生成执行计划，并维护数据库的所有模式信息，包括数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。表中每个字段的定义信息，包括字段类型、是否唯一等。数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

主要函数有以下：

- `CatalogManager`

用来初始化日志信息，如果是第一次创建，则析构的时候将所有信息序列化到catalogmetapage的data中，并将该page写入磁盘。如果不是第一次创建，则从磁盘中读取CATALOG_META_PAGE中的信息

- `CreateTable`

利用tablename建一个表并输出tableinfo，同时要把表信息序列化到CATALOG_META_PAGE中，以便析构的时候一起写入磁盘

- `GetTable`

根据tablename找到tableinfo

- `GetTables`

获取所有表的tableinfo

- `CreateIndex`

根据tablename，indexname，indexkeys创建索引，并存储在indexinfo中；同时，在创建结束之后，需要先利用keymap中的索引将每一个row进行重组，组成一个新的row，利用这个row对B+树进行插入

- `GetIndex`

利用tablename和indexname获取indexinfo

- `GetTableIndexes`

获取对应tablename的所有indexinfo

- `DropTable`

删除对应tablename的所有信息（包括磁盘中）

- `DropIndex`

根据tablename和indexname删掉所有关于这个index的信息（包括磁盘中）

- `FlushCatalogMetaPage`

这个函数是用在catalogmanager的析构函数中，指的是每次退出数据库引擎时都会将当前被序列化到CATALOG_META_PAGE中的信息写入磁盘中，以保证持久化的功能

- `LoadTable` & `LoadIndex`

将对应pageid中的Table和index的所有相关信息加载出来以便使用

- `GetTable`

根据tableid获取tableinfo

五、PLANNER AND EXECUTOR

本部分主要执行了以下函数：

该函数主要执行数据库中表的创建。在选择了数据库的前置条件下，遍历抽象语法结构获取创建表的表名、相关属性信息以及主键约束，利用这些信息构造数据库可以理解的对象，例如`Column` `Schema`。调用当前选择的数据库的`Catalog Manager`来创建表，同时对于约束是Unique、主键的属性需要创建索引。

- `ExecuteDropTable`

该函数主要执行数据库中表的删除。在选择了数据库的前置条件下，从语法树中获取删除的表名，调用`Catalog Manager`删除表。

- `ExecuteShowIndexes`

该函数主要执行数据库中索引的展示。在选择了数据库的前置条件下，调用当前数据库的`Catalog Manager`的`GetTables`函数获取所有的表信息。遍历所有的表信息，获取表明的最大长度；对于每一个表，遍历器所有索引信息，获取索引名的最大长度。这两个长度主要是用于后续的格式化输出。最后遍历每一个表，对于每一个表遍历其所有索引，按照格式进行输出。

- `ExecuteCreateIndex`

该函数主要执行数据库中索引的创建。首先我们取出`catalog`，查询当前是否已经有这个表了。如果没有，根据语法树的结构遍历属性的定义,并找出是否有 unique 属性，以及可能的类型size(char类型)。随后遍历主键的定义，将这些属性加入到主键的容器中。结束遍历后，根据刚刚遍历的结果初始化Column对象，并基于此创建TableSchema对象。随后调用catalog中的CreateTable函数创建表。然后还要基于主键和unique属性的元素创建索引，调用 catalog中的CreateIndex即可，遍历主键的定义，将这些属性加入到主键的容器中。

- `ExecuteDropIndex`

该函数主要执行数据库中索引的删除。在选择了数据库的前置条件下，通过语法树获取需要删除的索引名称，调用当前数据库的`Catalog Manager`的`GetTables`函数获取当前数据库的所有表信息。对于每一个表，调用`Catalog Manager`的`GetTableIndexes`函数获取该表的所有索引信息，对于每一个索引，如果名称与待删除的索引相同，则调用`Catalog Manager`的`DropIndex`函数删除索引，同时标记

删除成功。

- `ExecuteExecfile`

该函数主要执行文件的读入（多SQL语句的输入）。在选择了数据库的前置条件下，解析语法树获得文件的名称，然后使用`freopen`函数改变输入流定向至对应文件，此时后续的读入都会从文件中读取。为了计算执行文件所花费的总时间，我额外给`Engine`添加了一个public的成员变量`file_start_time`作为文件执行的初始时间，具体在`execute_engine.h`文件中定义。在minisql的主入口`main.cpp`中，需要修改`InputCommand`函数——当文件读取完之后重定向回终端。同时文件结束时，记录结束时间，输出运行总时间。

- `ExecuteQuit`

该函数主要执行数据库退出操作，只需要简单地返回`DB_QUIT`即可。注意不能使用exit(0)跳出环境，因为这样会导致不能正常调用对象的析构函数。

六、项目测试通过情况

我的MiniSQL在注释掉`executor_test.cpp`中xx的情况下，通过了所有测试，且成功完成了十万条数据插入并查询的测试，也按照验收流程完成了所有的操作。

`clock_replacer_test`的测试在CMU 15-445中取出（其实和`LRU_Replacer`中的测试点差距不大），并进行了适当的修改，并通过了测试。

七、其他

代码部署在github私有仓库，同时我还写了一个文档作为开发历程的记录，也记录了需要执行的操作。在代码文件中即为environment.md，在此把内容一并上传。

```
配置环境完成 2024.4.21
我使用的是vscode+wsl的形式
part1：配置wsl
    wsl之前已经配置好，网上教程不少
part2：clone代码框架
    进入vscode主页打开wsl：ubuntu
    打开终端 在zju内网下 ```sudo git clone
https://git.zju.edu.cn/zjucsdb/minisql.git```
    以下各步骤如果提示没有权限的话就加一个```sudo``` 然后输入密码就可以啦
    ```cd minisql```
    ```mkdir build```
    ```cd build```
    ```sudo cmake ..```
    在这几步我都没有遇到问题
part3：修改一下框架
    在我的测试中，如果直接按语雀上输入```make -j4```会提示报错，具体是_cv变量找不到
    这个时候如果直接修改/minisql/src/include/concurrency/lock_manager.h
    为它添加头文件```#include<condition_variable>```会提示没有保存权限，无法修改
    我尝试过使用vim相关的指令但没有成功
    解决方法是通过```cd xx```进入项目根目录（即一步步进入concurrency文件夹）
    采用```sudo chmod 777 -R lock_manager.h```获得保存权限，之后就可以修改了
    助教gg在钉钉群内说修改过后即可通过，但我这里还出现了问题，
是/executor/plans/abstract_plan.h的智能指针缺少头文件
    这个时候我先进入根目录，修改保存权限，加入了```#include<memory>```然后再编译就解决了问
题！
part4：完成
```

采用```sudo make -j4```完成配置，我这里是没有报错啦

终端输入```./bin/main```回车就弹出minisql了，不过只能quit，因为还没写别的操作它不认识

part5：测试

进入build文件夹，```make lru_replacer_test```，还是如果没权限就开sudo

还是build目录下，```./test/lru_replacer_test```就可以测试~

怎么把zjugit的代码弄到github自己的仓库里

https://blog.csdn.net/m0_55546349/article/details/121786789

如何提交本地的更改到那个仓库

```
git add .
git commit -m "commit message" commit message就是批注
git push origin main
```

此外：安装clangd可以避免全是红色9+的错误 方便调试和看着舒心

步骤如下：

https://zhuanlan.zhihu.com/p/592802373

完成：

4.26 开始尝试#1 1.2 disk_manager_test bitmap_page.cpp

5.6 #1 1.2 disk_manager_test 测试点1通过 开始尝试#1 1.3 disk_manager_test disk_manager.cpp

5.10 #1 完成disk_manager.cpp 开始尝试#1 1.4 lru_replacer_test lru_replacer.cpp

5.15 #1 1.4 lru_replacer_test 测试点通过 开始尝试#1 1.5 buffer_pool_manager_test buffer_pool_manager.cpp

5.16 #1 1.5 buffer_pool_manager.cpp完成 不过似乎没有过测试点 不管了开始#2 2.1

5.31 #2 2 程序全部完成 但是测试点一个都没过 tuple_test和table_heap_test

6.2 #2 2.1 tuple_test测试点通过

6.3 #1 disk_manager_test 测试点2通过 是之前环境的问题 在mnt目录下重新配了下 之前是直接wsl的目录下

发现过程：使用cerr对不通过的测试点进行排查 发现是"xx.db"那句话的导入出了问题 在网上复制了正确同学的代码 还是这个地方有问题 观看同学正确的环境 发现有区别 于是开始尝试新环境

差 #1 buffer_pool_manager_test和#2 table_heap_test

6.4 #1 buffer_pool_manager测试点通过 #2 table_heap_test测试点通过

6.6 #3 b_plus_tree_index_test测试点和index_iterator_test测试点通过 b_plus_tree_test查询那一块有问题

6.10 #3 b_plus_tree_test测试点通过 #4 catalog_test通过 前四部分全部过完

6.13 #5 executor_test过掉 但是手动insert插入表格会有问题 很奇怪

6.15 #5 executor_test又过不掉了

6.17 #5 注释掉建数据库代码以后 过掉全部测试 至少插入678条数据没有问题(使用测试的那个table) 但是删除以后再插入插入4-5条后就会报错然后直接弹出 报错是b_plus_tree.cpp第471行 不注释掉那一段代码(execute_engine.cpp)会根本没办法运行 magic_num那一段报错 catalog.cpp

注释掉以后 按照验收流程运行所有测试 在676个insert情况下 所有测试没有问题

100000条数据居然跑成功了 截图见微信 挺抽象的 不过电脑跑的爆烫还有点卡

minisql_ the end. 正式结束吧 复习其他考试