MiniSQL实验报告

MiniSQL实验报告

0 小组分工

- 黄声源: `DISK AND BUFFER POOL MANAGER` & `RECORD MANAGER`
- 何光昭: `INDEX MANAGER`
- 蔡雨谦: `CATALOG MANAGER` & `PLANNER AND EXECUTOR`

注：在实际开发过程中，由于不同模块之间耦合性较高，许多bug是由组员共同修复

1 框架概述

1.1 引言

1.1.1. 实验目的

- 设计并实现一个精简型单用户SQL引擎MiniSQL，允许用户通过字符界面输入SQL语句实现基本的增删改查操作，并能够通过索引来优化性能。
- 通过对MiniSQL的设计与实现，提高学生的系统编程能力，加深对数据库管理系统底层设计的理解。

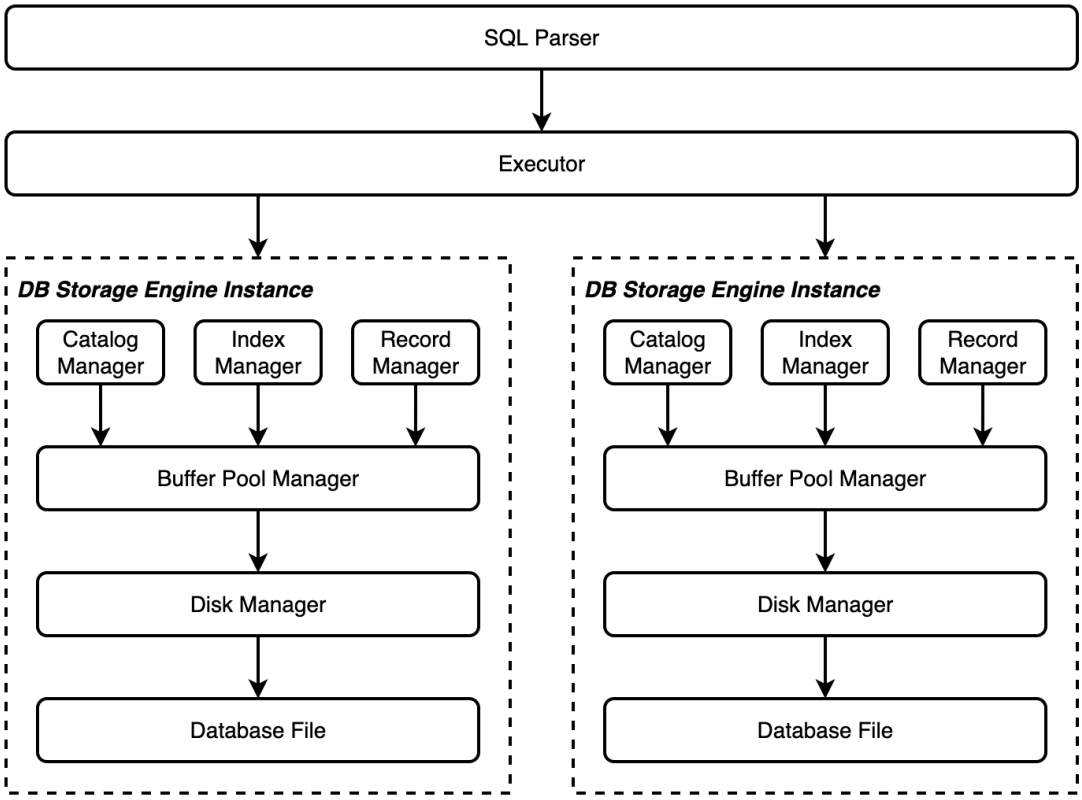
1.1.2. 实验需求

- 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
- 表定义：一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
- 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
- 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。
- 在工程实现上，使用源代码管理工具（如Git）进行代码管理，代码提交历史和每次提交的信息清晰明确；同时编写的代码应符合代码规范，具有良好的代码风格。

1.2 系统架构与模块概述

1.2.1 系统架构概述

- 在系统架构中，解释器`SQL Parser`在解析SQL语句后将生成的语法树交由执行器`Executor`处理。执行器则根据语法树的内容对相应的数据库实例（`DB Storage Engine Instance`）进行操作。
- 每个`DB Storage Engine Instance`对应了一个数据库实例（即通过`CREATE DATABSAE`创建的数据库）。在每个数据库实例中，用户可以定义若干表和索引，表和索引的信息通过`Catalog Manager`、`Index Manager`和`Record Manager`进行维护。目前系统架构中已经支持使用多个数据库实例，不同的数据库实例可以通过`USE`语句切换（即类似于MySQL的切换数据库）。



1.2.2 系统模块概述

1.2.2.1 Disk Manager

- Database File（DB File）是存储数据库中所有数据的文件，其主要由记录（Record）数据、索引（Index）数据和目录（Catalog）数据组成（即共享表空间的设计方式）。与书上提供的设计（每张表通过一个文件维护，每个索引也通过一个文件维护，即独占表空间的设计方式）有所不同。共享表空间的优势在于所有的数据在同一个文件中，方便管理，但其同样存在着缺点，所有的数据和索引存放到一个文件中将会导致产生一个非常大的文件，同时多个表及索引在表空间中混合存储会导致做了大量删除操作后可能会留有有大量的空隙。
- Disk Manager负责DB File中数据页的分配和回收，以及数据页中数据的读取和写入。

1.2.2.2 Buffer Pool Manager

- Buffer Manager 负责缓冲区的管理，主要功能包括：

1. 1. 根据需要，从磁盘中读取指定的数据页到缓冲区中或将缓冲区中的数据页转储（Flush）到磁盘；
   2. 实现缓冲区的替换算法，当缓冲区满时选择合适的数据页进行替换；

3. 记录缓冲区中各页的状态，如是否是脏页（Dirty Page）、是否被锁定（Pin）等；
4. 提供缓冲区页的锁定功能，被锁定的页将不允许替换。

- 为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是数据页（Page），数据页的大小应为文件系统与磁盘交互单位的整数倍。在本实验中，数据页的大小默认为 4KB。

1.2.2.3 Record Manager

- Record Manager 负责管理数据表中记录。所有的记录以堆表（Table Heap）的形式进行组织。Record Manager 的主要功能包括：记录的插入、删除与查找操作，并对外提供相应的接口。其中查找操作返回的是符合条件记录的起始迭代器，对迭代器的迭代访问操作由执行器（Executor）进行。
- 堆表是由多个数据页构成的链表，每个数据页中包含一条或多条记录，支持非定长记录的存储。不要求支持单条记录的跨页存储（即保证所有插入的记录都小于数据页的大小）。堆表中所有的记录都是无序存储的。
- 需要额外说明的是，堆表只是记录组织的其中一种方式，除此之外，记录还可以通过顺序文件（按照主键大小顺序存储所有的记录）、B+树文件（所有的记录都存储在B+树的叶结点中，MySQL中InnoDB存储引擎存储记录的方式）等形式进行组织。

1.2.2.4 Index Manager

- Index Manager 负责数据表索引的实现和管理，包括：索引（B+树等形式）的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。
- B+树索引中的节点大小应与缓冲区的数据页大小相同，B+树的叉数由节点大小与索引键大小计算得到。

1.2.2.5 Catalog Manager

- Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 1. 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。
   2. 表中每个字段的定义信息，包括字段类型、是否唯一等。
   3. 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。

- Catalog Manager 还必需提供访问及操作上述信息的接口，供执行器使用。

1.2.2.6 Planner and Executor

- Planner（执行计划生成器）的主要功能是根据解释器（Parser）生成的语法树，通过Catalog Manager 提供的信息检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与column类型对应等等，随后将这些词语转换成可以理解的各种 c++ 类。解析完成后，Planner根据改写语法树后生成的Statement结构，生成对应的Plannode，并将Plannode交由Executor进行执行。
- Executor（执行器）的主要功能是遍历Planner生成的计划树，将树上的 PlanNode 替换成对应的 Executor，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行。Executor采用的是火山模型，提供迭代器接口，每次调用时会返回一个元组和相应的 RID，直到执行完成。

1.2.2.7 SQL Parser

- 程序流程控制，即“启动并初始化 → ‘接收命令、处理命令、显示命令结果’循环 → 退出”流程。
- 接收并解释用户输入的命令，生成命令的内部数据结构表示，同时检查命令的语法正确性和部分语义正确性，对正确的命令生成语法树，然后调用执行器层提供的函数执行并显示执行结果，对不正确的命令显示错误信息。

1.3 开发环境

本项目完全基于`WSL2--ubuntu`进行开发，通过`vscode`连接至`Linux`子系统，依靠`glog`进行项目调试，通过`git`实现版本管理与多人协作开发。

2 模块功能

2.1 Buffer Pool Manager

Disk Manager和Buffer Pool Manager模块位于架构的最底层，**Buffer Pool Manager**需要利用**LRU**通过**Disk Manager**来完成数据页从磁盘到内存的读取和写入，而数据页的分配和回收又需要通过**Bitmap Page**来完成

2.1.1 Bitmap Page

位图页需要标记一段连续页的分配情况，一个位图有以下属性：

Type	Name
$uint32\_t$	$page\_allocated\_$
$uint32\_t$	$next\_free\_page\_$
$unsigned\ char*$	$bytes$

我们实现了以下函数：

Name	Description
$GetMaxSupportedSize$	返回该位图页最大支持的size
$AllocatePage$	分配一个空闲页
$DeAllocatePage$	回收已经被分配的页
$IsPageFree$	判断给定的页是否是空闲的
$IsPageFreeLow$	为$IsPageFree$提供服务

2.1.2 Disk Manager

Disk Manager需要从磁盘文件中读取数据页或写入数据页，并且需要加锁以应对并发场景

Type	Name
$fstream$	$db\_io\_$
$string$	$file\_name\_$

Type	Name
$recursive\_mutex$	$db\_io\_latch\_$
$bool$	$closed$
$char*$	$meta\_data\_$

我们实现了以下函数：

Name	Description
$ReadPage$	通过逻辑页号读取磁盘中的数据页
$WritePage$	通过逻辑页号写入磁盘中的数据页
$AllocatePage$	从磁盘中分配一个空闲页并返回逻辑页号
$DeAllocatePage$	释放磁盘中逻辑页号对应的物理页
$IsPageFree$	返回逻辑页号对应的数据页是否空闲
$Close$	关闭所有数据库文件和Disk Manager
$GetMetaData$	返回元数据的地址
$GetFileSize$	返回磁盘文件的大小
$ReadPhysicalPage$	根据物理页号在磁盘中读取数据
$WritePhysicalPage$	根据物理页号向磁盘中写入数据
$MapPageId$	将逻辑页号映射为物理页号

2.1.3 LRU替换算法

`LRUReplacer`扩展了抽象类`Replacer`，其中有这些成员：

Type	Name
$list < frame\_id\_t >$	LRU_list
$unordered\_map < frame\_id\_t, list < frame\_id\_t >:: iterator >$	LRU_hash
$recursive\_mutex$	$LRU\_latch\_$

我们实现了如下函数：

Name	Description
$Victim$	找到最近最少被访问的页，将其页帧号输出
Pin	将数据页固定
$UnPin$	将数据页解除固定

Name	Description
$Size$	返回能被替换的数据页的数量

2.1.4 Buffer Pool Manager

Buffer Pool Manager使用了一个哈希表来记录page的逻辑页号和帧页号的映射，以便快速定位到帧页，还需要使用lru_replacer和freelist共同维护可用的页面

Type	Name
$size\_t$	$pool\_size\_$
$Page*$	$pages\_$
$DiskManager*$	$disk\_manager\_$
$unordered\_map < page\_id\_t, frame\_id\_t >$	$page\_table\_$
$Replacer*$	$replacer\_$
$list < frame\_id\_t >$	$free\_list\_$
$recursive\_mutex$	$latch\_$

为此，我们实现了如下函数：

Name	Description
$FetchPage$	根据逻辑页号获取数据页
$UnpinPage$	取消固定一个数据页
$FlushPage$	将数据页转储到磁盘中
$NewPage$	分配一个新的数据页，并将逻辑页号于`page_id`中返回
$DeletePage$	释放一个数据页
$IsPageFree$	判断一个逻辑页号对应的页是否空闲
$AllocatePage$	通过Disk Manager分配数据页
$DeallocatePage$	通过Disk Manager释放数据页

2.2 Record Manager

Record Manager负责管理数据表中所有的记录，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。

2.2.1 序列化与反序列化

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，目的是为了实现数据的持久化。

2.2.1.1 Row

```
/**
 *  Row format:
 * ------------------------------------------
 * | Header | Field-1 | ... | Field-N |
 * ------------------------------------------
 *  Header format:
 * ------------------------------------------
 * | Field Nums | Null bitmap |
 * ------------------------------------------
 *
 */
```

对于Row, 只有两个成员属性，分别是行的唯一标识`RowId`，该行的字段`Field`：

Type	Name
$RowId$	$rid\_$
$vector<Field*>$	$fields\_$

2.2.1.2 Column

引入魔数(MAGIC_NUM)来确保序列化和反序列化的正确性。对于每一个Column都需要有字段名，字段类型，在表中的位置以及与约束相关的信息。

Type	Name
$uint32\_t$	$COLUMN\_MAGIC\_NUM$
$string$	$name\_$
$TypeId$	$type\_$
$uint32\_t$	$len\_$
$uint32\_t$	$table\_ind\_$
$bool$	$nullable\_$
$bool$	$unique\_$

2.2.1.3 Schema

同样有魔数，成员属性如下：

Type	Name
$uint32\_t$	$SCHEMA\_MAGIC\_NUM$
$vector<Column*>$	$columns\_$
$bool$	$is\_manage\_$

对于`Row`，`Column`，`Schema`都有其不同的序列化与反序列化的具体实现，但他们都共同拥有三个最主要的函数：

Name	Description
$SerializeTo$	将成员属性序列化写入字节流
$DeserializeFrom$	从字节流中获取成员属性
$GetSerializedSize$	返回在序列化和反序列化中`buf`指针向前推进了多少个字节

2.2.2 Table Heap

堆表中的每个数据页`table_page`（与课本中的`Slotted-page Structure`给出的结构基本一致，见下图，能够支持存储不定长的记录）都由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成。如图：

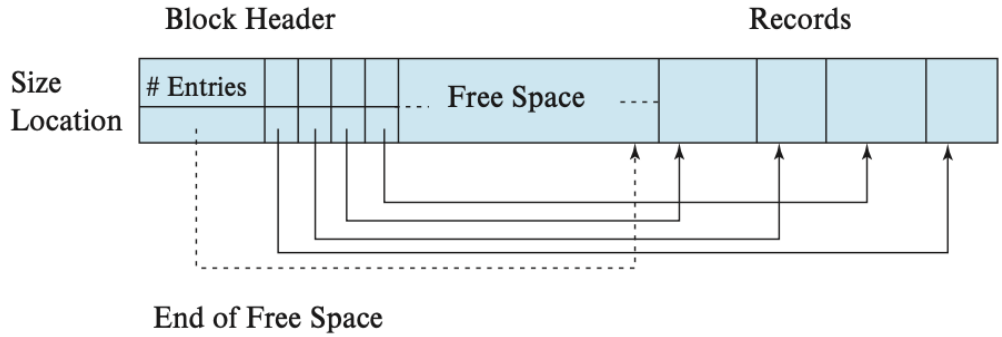


Figure 13.6 Slotted-page structure.

`table_heap`的成员如下：

Type	Name
$BufferPoolManager*$	$buffer\_pool\_manager\_$
$page\_id\_t$	$first\_page\_id\_$
$Schema*$	$schema\_$
$LogManager*$	$log\_manager\_$
$LockManager*$	$lock\_manager\_$

我们实现的函数如下：

Name	Description
$InsertTuple$	向堆表中插入一条记录
$MarkDelete$	标记需要删除记录
$UpdateTuple$	更新一条记录
$ApplyDelete$	从物理意义上删除这条记录

Name	Description
$RollbackDelete$	回滚一次删除
$GetTuple$	根据 `RowId` 获取一条记录
$FreeTableHeap$	销毁整个 `TableHeap` 并释放这些数据页
$Begin$	获取堆表的首迭代器
End	获取堆表的尾迭代器

2.2.3 TableIterator

一个使用迭代器的例子：

```
for (auto iter = table_heap.Begin(); iter != table_heap.End(); iter++) {
    Row &row = *iter;
    /* do some things */
}
```

遍历堆表中每一条记录，成员如下：

Type	Name
$TableHeap*$	$table\_heap\_$
$Row*$	$row\_$

重载的成员函数如下:

Name	Description
$operator ==$	判断两个Iterator是否指向的是同一个Row
$operator!=$	`==` 取反
$operator*$	返回Row
$operator->$	返回Row的指针
$operator =$	拷贝
$operator++$	查找下一行记录

2.3 Index Manager

Index Manager的实现由树节点和树调整两部分组成。在树节点部分，**BPlusTreePage** 是所有树节点的基类，**BPlusTreeInternalPage** 是内部节点页，**BPlusTreeLeafPage** 是叶节点页。节点页存储表的键和/或行 ID（值）。启动、插入和删除 B+ 树中节点所需的函数在 **b_plus_tree.cpp** 中。

2.3.1 BPlusTreePage

一个page class有以下属性

Type	Name
$IndexPageType$	$page\_type\_$
int	$key\_size\_$
Isn_t	$lsn\_$
int	$size\_$
int	$max\_size\_$
$page\_id\_t$	$parent\_page\_id\_$
$page\_id\_t$	$page\_id\_$

我们实现了以下函数：

Name	Description
$IsLeafPage$	return true for leaf node
$IsRootPage$	return true for root node
$SetPageType$	set page to either internal or leaf node
$GetKeySize$	get byte size of a key in current node
$SetKeySize$	set byte size of a key in current node
$GetSize$	get count number of keys in current node
$SetSize$	set count number of keys in current nod
$IncreaseSize$	increase count number of keys by certain amount
$GetMaxSize$	get the maximum size of keys this node can contain
$SetMaxSize$	set the maximum size of keys this node can contain
$GetMinSize$	get the minimal size of keys this node can have without splitting
$GetParentPageId$	get the page id of the parent node
$SetParentPageId$	set the page id of the parent node
$GetPageId$	get the current page id
$SetPageId$	set the current page id

2.3.2 BPlusTreeInternalPage

BPlusTreeInternalPage存储内部节点的键值对。这里的"键"是指索引键，而"值"是指其相应子节点的页面 ID。它只有一个属性：

Type	Name
$char*$	$data$

此 char 数组存储键值对，并通过字节寻址进行索引。它还包含以下方法，以便能够为callers提供搜索、插入和删除功能。

Name	Description
$Init$	set attributes such as page id and parent id
$KeyAt$	return the key at the given index
$SetKeyAt$	set key at given index with given key
$ValueIndex$	return the searched index of given value
$ValueAt$	return the value at the given index
$SetValueAt$	set value at given index with given value
$PairPtrAt$	return pointer to the start of the pair at given index
$PairCopy$	copy certain amount of pairs from one address to another
$Lookup$	return the value for a given key
$PopulateNewRoot$	set the key and value pairs for a new root node
$InsertNodeAfter$	insert a pair of key and value after a given key
$Remove$	remove the pair at given index
$RemoveAndReturnOnlyChild$	return the only child of this node and resize it
$MoveAllTo$	move all pairs inside current node to another during merging
$MoveHalfTo$	move half of the pairs in current node to another during splitting
$MoveFirstToEndOf$	move the first value to recipient node and set its key during redistribution
$MoveLastToFrontOf$	move the last value to recipient node during redistribution
$CopyNFrom$	copy n pairs from source node, called by $MoveHalfTo$

Name	Description
$CopyLastFrom$	copy the first value from source node to the last value, called by $MoveFirstToEndOf$
$CopyFirstFrom$	copy the last value from source node to the first value, called by $MoveLastToFrontOf$

2.3.3 BPlusTreeLeafPage

与 **BPlusTreeInternalPage** 类似，**BPlusTreeLeafPage** 存储键值对，键是表键，值是表中特定记录行的 **RowID**。它具有以下属性：

Type	Name
$page\_id\_t$	$next\_page\_id\_$
$char*$	$data$

我们实现了以下函数：

Name	Description
$Init$	set attributes such as page id and parent id
$SetNextPageId$	get the next sibling leaf page id during index iterating
$KeyAt$	return the key at the given index
$SetKeyAt$	set key at given index with given key
$ValueAt$	return the value at the given index
$SetValueAt$	set value at given index with given value
$PairPtrAt$	return pointer to the start of the pair at given index
$PairCopy$	copy certain amount of pairs from one address to another
$GetItem$	get the pair at given index
$Insert$	insert a pair into current leaf node
$Lookup$	return the value for a given key
$RemoveAndDeleteRecord$	remove and delete record given certain key
$MoveAllTo$	move all pairs inside current node to another during merging
$MoveHalfTo$	move half of the pairs in current node to another during splitting
$MoveFirstToEndOf$	move the first value to recipient node and set its key during redistribution

Name	Description
$MoveLastToFrontOf$	move the last value to recipient node during redistribution
$CopyNFrom$	copy n pairs from source node, called by $MoveHalfTo$
$CopyLastFrom$	copy the first value from source node to the last value, called by $MoveFirstToEndOf$
$CopyFirstFrom$	copy the last value from source node to the first value, called by $MoveLastToFrontOf$

2.3.4 BPlusTree

BPlutTree 类包含 B+ 树的所有逻辑部分，即使用内部或叶节点内的方法调整节点和内部对。以下是此类中的属性：

Type	Name
$index\_id\_t$	$index\_id\_$
$page\_id\_t$	$root\_page\_id\_$
$BufferPoolManager*$	$buffer\_pool\_manager\_$
$KeyManager$	$processor\_$
int	$leaf\_max\_size\_$
int	$internal\_max\_size\_$

以下是我们为此部分实现的功能：

Name	Description
$BPlusTree$	constructor of a new tree, initializes attributes and updates index root page
$IsEmpty$	return true if the root doesn't exist
$Insert$	insert a new pair of key and value into the tree
$Remove$	remove a pair of key and value from the tree
$GetValue$	return the found value of given key
$FindLeafPage$	either return the leaf page which the key is on or the left most leaf page for generating index iterator
$Begin$	get the starting iterator, which points at the left most pair in the leaf nodes
$Begin$	@overload, get the iterator pointing the pair which contains given key

Name	Description
End	get the ending iterator, which marks the stopping point of iterating through the leaf pairs
$Check$	seek through buffer pool manager to find any page that is not unpinned and returns error
$Destroy$	destroy the entire current tree
$StartNewTree$	add the first pair of key and value into a new tree
$InsertIntoLeaf$	insert a new pair of key and value into leaf node and update internal nodes
$InsertIntoParent$	insert a new pair of key and value into internal nodes
$Split$	create a new sibling node, move half of pairs to it and update parents
$CoalesceOrRedistribute$	check the size of current and sibling nodes and decide whether to coalesce or redistribute during deletion
$Coalesce$	merge two nodes together and keep updating parent node if necessary
$Redistribute$	move the first of last node of current node to sibling node and update corresponding keys and values
$AdjustRoot$	if the last element inside root node is deleted, there could be two possibilities, either the root should be removed or the entire tree is empty. This method adjust the root in such conditions
$UpdateRootPageId$	when root page id is updated, this method is called to change the root id inside index root page
$FindLeftOrRightMostLeaf$	we implemented this method for End which can return the last leaf page address

2.3.5 IndexIterator

在这一部分中，我们只需要实现三种方法：

Name	Description
$operator*$	returns the pair of key and value the current iterator is pointing at
$operator++$	returns the next iterator, if this is the last pair inside this node, go to next sibling leaf node
$operator==$	if both the page id and the index are the same, then return true, otherwise return false

2.4 Catalog Manager

2.4.1功能介绍

该模块具有的功能如下：

功能	描述
create table	创建新的表，以tableInfo的形式存在，将序列化信息加入到当前的目录元信息的页和缓存池的页中，并更新记录着<tableName,tableId>和<tableId,tableInfo>的两张maps。
get table	从当前的目录元信息中获取指定表名的表，即通过两张maps由tableName找到tableInfo。
get tables	获取当前目录元信息的所有表。
load table	从缓存池中获取指定id的页，将页上的数据反序列化得到表的元信息后加入目录元信息的指定页中。
create index	创建新的索引，以indexInfo的形式存在，将序列化信息加入到当前的目录元信息的页和缓存池的页中，并更新信息到记录着<tableName,<indexName,indexId>>和<indexId,indexInfo>的两张maps中。
get index	获取当前目录元信息中指定表中的指定索引。
get table indexes	获取指定表的全部索引。
drop index	删除指定索引，当被删除索引的表中没有其他索引时，删除<tableName,<indexName,indexId>>map中的该表。
load index	从缓存池中获取指定id的页，将页上的数据反序列化得到索引的元信息后加入目录元信息的指定页中。
flush catalog page	将目录元页刷新一遍，把当前的目录数据储存进磁盘。

2.4.2对外接口

该模块提供的供上层模块调用的接口如下：

- void CatalogMeta::SerializeTo(char *buf) const
- CatalogMeta *CatalogMeta::DeserializeFrom(char *buf)
- uint32_t CatalogMeta::GetSerializedSize()
- CatalogMeta::CatalogMeta()
- CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager,LogManager *log_manager, bool init)
- dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema, ransaction *txn, TableInfo *&table_info)

- dberr_t CatalogManager::GetTable(const string &table_name, TableInfo *&table_info)
- dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables)
- dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string &index_name,const std::vector<std::string> &index_keys, Transaction *txn, IndexInfo *&index_info, const string &index_type)
- dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string &index_name, IndexInfo *&index_info)
- dberr_t CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes)
- dberr_t CatalogManager::DropTable(const string &table_name)
- dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name)
- dberr_t CatalogManager::FlushCatalogMetaPage()
- dberr_t CatalogManager::LoadTable(const table_id_t table_id, const page_id_t page_id)
- dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t page_id)

2.5 Planner and Executor

2.5.1功能介绍

该模块具有的功能如下：

功能	描述
create database	当输入的指令为创建数据库时（以下省略该条件），创建数据库。
use database	使用指定数据库
show databases	显示全部数据库。
drop database	删除数据库。
create table	根据sql语句的条件创建表。
show tables	显示全部表。
drop table	删除表。
create index	创建索引。
show indexes	显示全部索引。
drop index	删除索引。
execute file	执行指定文件中的语句。
quit	退出minisql。
seq-scan	按顺序遍历表中的元组
index-scan	按索引遍历元组
insert	插入元组
update	更新元组
delete	删除元组

2.5.2对外接口

该模块提供的供上层模块调用的接口如下：

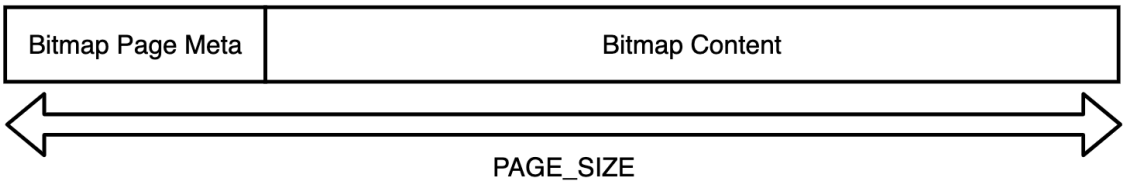
- ExecuteEngine::ExecuteEngine()
- std::unique_ptr<AbstractExecutor>ExecuteEngine::CreateExecutor(ExecuteContext *exec_ctx, const AbstractPlanNodeRef &plan)
- dberr_t ExecuteEngine::ExecutePlan(const AbstractPlanNodeRef &plan, std::vector <Row> *result_set, Transaction *txn, ExecuteContext *exec_ctx)
- dberr_t ExecuteEngine::Execute(pSyntaxNode ast)
- void ExecuteEngine::ExecuteInformation(dberr_t result)
- dberr_t ExecuteEngine::ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context)
- dberr_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast, ExecuteContext *context)
- void SeqScanExecutor::Init()
- bool SeqScanExecutor::Next(Row *row, RowId *rid)
- void IndexScanExecutor::Init()
- bool IndexScanExecutor::Next(Row *row, RowId *rid)
- void InsertExecutor::Init()
- bool InsertExecutor::Next(Row *row, RowId *rid)
- void UpdateExecutor::Init()
- bool UpdateExecutor::Next(Row *row, RowId *rid)
- Row UpdateExecutor::GenerateUpdatedTuple(const Row &src_row)
- void ValuesExecutor::Init()
- bool ValuesExecutor::Next(Row *row, RowId *rid)
- void DeleteExecutor::Init()
- bool DeleteExecutor::Next(Row *row, RowId *rid)

3 具体实现

3.1 Buffer Pool Manager

3.1.1 Bitmap Page

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_`）。



除去元数据外的剩余部分就是一个`unsigned char`数组来管理内存，一个`char`类型在内存中是8个bit，正好对应了8个数据页的分配情况，每次需要allocate/deallocate一个数据页的时候，就将那个数据页对应的bit置为1/0。

```
auto byte_index = page_offset / 8;
auto bit_index = page_offset % 8;
bytes[byte_index] |= 1 << bit_index;
```

并且遍历查找bitmap中的空bit设置为`next_free_page_`。

3.1.2 Disk Manager

为使磁盘文件能够维护更多的数据页信息，我们把位图页和一段连续的数据页看成一个分区(Extent)，再由一个磁盘元数据页记录这些分区的信息。

Disk Meta Page	Extent Meta (Bitmap Page)	Extent Pages	Extent Meta (Bitmap Page)	Extent Pages	...

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**	

然而由于上层调用不需要知道物理分页，这时我们就需要将物理分页映射为上层可感的逻辑分页：

```
page_id_t DiskManager::MapPageId(page_id_t logical_page_id) {
  return logical_page_id + logical_page_id / BITMAP_SIZE + 2;
}
```

从磁盘中分配数据页的方法是先寻找到有空闲页的分区，定位到其bitmap，再通过bitmap定位到具体的物理页，如果没有分区有空闲页则新建一个分区。

相应地，要修改磁盘元数据和bitmap。

回收数据页的方法只需找到分区所在的bitmap，修改磁盘元数据和bitmap即可。

3.1.3 LRU替换算法

LRU(least recently used)最近最少使用算法，使用一个`list`来标记所有最近使用的页的帧页号(buffer pool manager中的frame)，如果有一个页最近使用了，则将帧页号其插入到队列的尾，当我需要替换数据页时，该队列的头即是最近最少使用的page，它将被替换，返回帧页号给上层。

- `Pin`：从队列中移除该页帧。

- `UnPin`：放入队列中，仅在上层Buffer Pool Manager中一个数据页的引用计数为0时被调用。

`LRU_hash`是加速`LRU_list`查找用的。

3.1.4 Buffer Pool Manager

上层通过Buffer Pool Manager与底层数据进行通信。

`Page`类型的数组`pages_`用来存放内存中实际的页，数组的下标即是页的帧页号(frame_id)，另外有一个哈希表`page_table_`用来存其对应的逻辑页号，便于与下层函数之间进行通信。

- `FetchPage`：先在`page_table_`中找是否有这个逻辑页号，如果找到了则fetch成功，在lru中pin住并返回该页；如果没找到则说明不在内存中，需要从`free_list_`或者`replacer_`中找到可用的页并通过Disk Manager从磁盘中获取该页。
- `NewPage`：先在`free_list_`和`replacer_`寻找空闲页，如果没找到则返回空指针，找到则通过Disk Manager为其匹配一个逻辑页号并返回这个逻辑页号。
- `UnPinPage`：当其他函数要用到某一页时会在lru用中pin住并使其pin_count加一，Unpin则是使其pin_count减一，当pin_count为0时可以在lru中unpin
- `FlushPage`：根据哈希表找到帧页号对应的逻辑页号，将逻辑页号传给Disk Manager进行刷盘（写入数据）。
- `DeletePage`：reset该页占有的内存，相应地更新该用到的资源，如freelist，lru，哈希表。

3.2 Record Manager

3.2.1 序列化与反序列化

- `SerializeTo`：通过`MACH_WRITE`宏写入内存`buf`本质上是使用c++的语法`reinterpret`，需要注意的是要记录每一次写进内存的偏移量`offset`以免不同的值写入同一个地址发生错误，最后返回`offset`。
- `DeserializeFrom`：是序列化的逆过程，本质也是使用`reinterpret`，不再赘述
- `GetSerializedSize`：直接计算得出（反）序列化的内存偏移量，主要用途是检测（反）序列化是否出错。

3.2.2 Table Heap

table heap中以双向链表的方式存放了许多table page，table page中又有许多row，每一个row都一个一个rowId与之唯一对应。

- `InsertTuple`：遍历堆表，沿着`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`TablePage`（*First Fit* 策略），向该page中插入tuple；如果没有找到page，则`buffer_pool_manager_->NewPage`进行插入，注意需要更新相关的一些参数。

其他函数较为简单，基本都是调用下层buffer pool manager或者table page中的同名函数。

3.2.3 TableIterator

完全由自己定义以及实现，我的做法是每一个迭代器包含两个指针，一个指向堆表一个指向row，遍历的时候就在堆表中遍历每一个page的每一个row，这也是++操作符的核心，其他的运算符都比较简单，需要注意的是==操作符需要参考之前在table heap中定义的`begin`和`end`函数，需要保证遍历到最后能够`==`end迭代器，比如我的实现中，由于`++`如果没找到下一个row则返回的是一个`INVALID_ROWID`，而`end`迭代器最后为`nullptr`，所以我在`==`的重载中将这两种视为相等返回`true`。

其他细节部分请参考源码：

```
1 bool TableIterator::operator==(const TableIterator &itr) const {
```

```
  if (row_ == nullptr && itr.row_ == nullptr) {
    return true;
  } else if (row_ == nullptr || itr.row_ == nullptr) {
    if (row_ == nullptr) {
      return itr.row_->GetRowId() == INVALID_ROWID;
    } else {
      return row_->GetRowId() == INVALID_ROWID;
    }
  }
  return row_->GetRowId() == itr.row_->GetRowId();
}

// ++iter
TableIterator &TableIterator::operator++() {
  // return *this;
  // 若为空或非法则返回非法
  if (row_ == nullptr || row_->GetRowId() == INVALID_ROWID) {
    delete row_;
    row_ = new Row(INVALID_ROWID);
    return *this;
  }
  TablePage *table_page =
      reinterpret_cast<TablePage *>(table_heap_->buffer_pool_manager_->FetchPage(this->row_->GetRowId().GetPageId()));
  RowId *next_rid = new RowId();
  table_page->RLatch();
  // 若当页能找到row
  if (table_page->GetNextTupleRid(row_->GetRowId(), next_rid)) {
    delete row_;
    row_ = new Row(*next_rid);
    table_heap_->GetTuple(row_, nullptr);
    table_heap_->buffer_pool_manager_->UnpinPage(table_page->GetPageId(), false);
    table_page->RUnlatch();
    return *this;
  } else {
    // 当页没找到,找下一页, 直到找到可用的
    auto next_page_id = table_page->GetNextPageId();
    while (next_page_id != INVALID_PAGE_ID) {
      table_page->RUnlatch();
      table_heap_->buffer_pool_manager_->UnpinPage(table_page->GetPageId(), false);
      table_page = reinterpret_cast<TablePage *>(table_heap_->buffer_pool_manager_->FetchPage(next_page_id));
      table_page->RLatch();
      if (table_page->GetFirstTupleRid(next_rid)) {
        // 找到了可用的
        delete row_;
        row_ = new Row(*next_rid);
        table_heap_->GetTuple(row_, nullptr);
        table_heap_->buffer_pool_manager_->UnpinPage(table_page->GetPageId(), false);
        table_page->RUnlatch();
        return *this;
      } else {
```

```
52          // 没有找到，继续找
53          next_page_id = table_page->GetNextPageId();
54        }
55      }
56      // 没有可用的tuple，返回非法
57      delete row_;
58      row_ = new Row(INVALID_ROWID);
59      table_heap_->buffer_pool_manager_->UnpinPage(table_page->GetPageId(),
    false);
60      table_page->RUnlatch();
61      return *this;
62    }
63  }
```

3.3 Index Manager

3.3.1 BPlusTree Insertion

当将一对给定的键和值插入树中时，我们首先检查树是否为空。如果是，我们首先通过从buffer pool manager获取一个新页来初始化一个新树，并将其reinterpret为根的叶子。然后我们将该对插入根节点并更新根索引页。如果树已经存在，我们插入到这棵树的叶子中。这是通过搜索节点级别并进入正确的叶节点来完成的。然后我们将该对插入到叶节点内的正确位置，这可能涉及向后移动现有对以适合此对。然后我们检查生成的叶节点的大小。如果大小大于叶节点的最大大小，我们分配一个新页面并将其对的第一部分移动到新节点中。移动对后，我们将最小的键插入节点内，并将新同级的页面 ID 插入父节点。然后，如果父节点已满，它会继续拆分并插入其父节点，直到找到一个大小小于最大大小的节点，或者将根节点拆分为两个。

3.3.2 BPlusTree Deletion

删除过程与插入过程相反。给定要删除的键，我们首先从查找key level和level开始，直到leaf page level。

如果在任何叶节点中都找不到key，我们直接返回。

如果在叶节点中找到key，我们将继续删除过程。我们首先删除包含它们的叶节点中的键值对，并在必要时更新其父级。然后我们检查生成的节点的大小是否小于其最小大小。如果否，则删除完成。如果是，那么我们必须合并或重新分配。

这是通过检查同级节点的大小来完成的，如果它大于最小大小，那么我们可以从同级节点借用一个并更新父键。

但是，如果同级节点内的节点数量已经等于最小大小，我们必须将它们合并在一起，并在父节点内减少一对键值。如果父节点现在的最小大小对小于最小大小，则必须再次检查是重新分发还是合并。在找到一个节点超过最小节点或根节点只有一对之前，将执行此操作。然后，我们只需删除根，并将其唯一的子项设置为根索引页中的新根。

3.3.3 Index Iterator

Index Iterator主要由三部分组成：查找开始迭代器、定义获取下一个迭代器的逻辑以及查找结束迭代器以标记迭代停止。

对于第一部分，我们只需逐级向下查看树，每次都选择最小的键。这将使我们能够在整个树中找到最小的键，这将是我们迭代的起点。

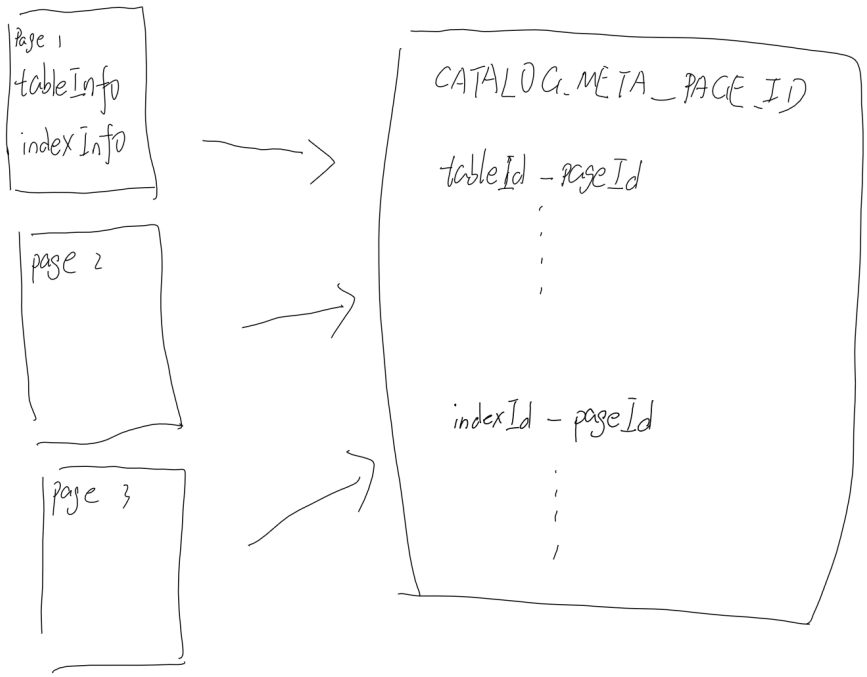
对于第二部分，由于叶节点彼此链接，我们首先检查当前对是否是当前节点内的最后一个。如果为 false，我们只需通过添加 pair_size 并保持相同的节点 id 来进入下一对。如果为 true，则我们转到下一个叶节点并选择其第一对作为我们的目的地。但是，如果下一个叶节点不存在，这意味着我们在迭代结束时接收，对于这种情况，我们选择保留相同的节点 ID，但将配对地址添加一pair_size以便索引现在等于节点大小。

对于最后一部分，当迭代结束时，迭代器应在索引等于大小的最后一个节点处停止。我们使用类似于查找最左边节点的函数来查找树内最右边的节点，并将其大小用作索引。通过这样做，当我们比较迭代器的节点 id 和索引时，结束迭代器将匹配最后一个可能的迭代器。

3.4 Catalog Manager

3.4.1Catalog Meta

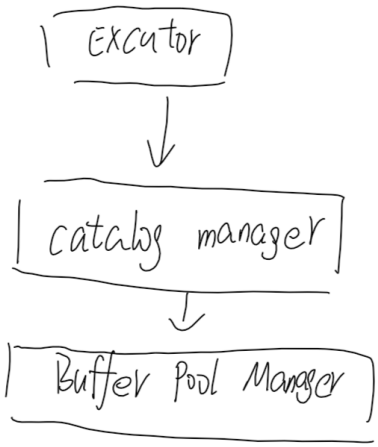
catalog简单来讲就是用来管理表和索引的数据信息的。表和索引的数据以tableInfo和indexInfo的类存储，这个类中包含表和索引的元信息metadata以及一些需要反序列化存储的信息。每张表和索引将存储在一张单独的page中，这个page存储的是对应表或索引序列化后的数据。而为了记录每张表和索引储存在那张page上，还需要一张总page来记录每张表和索引与各自page的对应关系，这张总page上的数据即为catalogMeta,其信息以序列化的形式储存在数据库的CATALOG_META_PAGE_ID数据页中，CATALOG_META_PAGE_ID的值为零。



tableInfo,tableId,tableName储存在两张map中，即<tableName,tableId>和<tableId,tableInfo>，这样可以通过表的名字或表的id来获取表的信息。同样的，索引有类似的两张map，<tableName, <indexName,indexId>>和<indexId,indexInfo>，不同的地方在于，索引依赖表存在，所以查找索引时需要知道在那张表上找。

3.4.2Catalog Manager

catalogManager用来管理目录元信息和数据页中表和索引的信息。当其初始化时，会加载元数据，把tableInfo和indexInfo放在内存中等待使用。通过catalogManager类里的方法可以对表与索引进行操作，包括创建、删除等。这些方法可以由上层模块调用，获取表与索引中的信息。



catalogManager可以调用buffer pool manager的方法，获取与存储各种元数据；同时会被executor调用，进行表与索引的增删。

3.4.3主要函数实现

Catalog Manager

```
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager,
                               LogManager *log_manager, bool init)
    : buffer_pool_manager_(buffer_pool_manager), lock_manager_(lock_manager), log_manager_(log_manager) {
  if(init)//初始化
  {
    //LOG(INFO)<<"init";
    catalog_meta_=CatalogMeta::NewInstance();
    next_table_id_=catalog_meta_->GetNextTableId();
    next_index_id_=catalog_meta_->GetNextIndexId();
  }
  else//已有catalogManager
  {
    //LOG(INFO)<<"call deserializeFrom";
    char* buf = buffer_pool_manager->FetchPage(CATALOG_META_PAGE_ID)->GetData();//获取目录元页的序列化信息
    catalog_meta_=CatalogMeta::DeserializeFrom(buf);//将序列化信息反序列化获得目录元信息
    next_table_id_=catalog_meta_->GetNextTableId();
    next_index_id_=catalog_meta_->GetNextIndexId();
    //将表信息载入目录
    for(map<table_id_t,page_id_t>::iterator it=catalog_meta_->table_meta_pages_.begin();it!=catalog_meta_->table_meta_pages_.end();it++)
    {
      dberr_t result=LoadTable(it->first,it->second);
      if(result!=DB_SUCCESS)
      {
        cout<<"Error:Loadtable failed.Table already exist."<<endl;
      }
    }
    //将索引信息载入目录
    for(map<table_id_t,page_id_t>::iterator it=catalog_meta_->index_meta_pages_.begin();it!=catalog_meta_->index_meta_pages_.end();it++)
    {
      dberr_t result=LoadIndex(it->first,it->second);
      if(result!=DB_SUCCESS)
      {
        cout<<"Error:Loadindex failed.Index already exist."<<endl;
      }
    }
    buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, false);//取消固定目录元页
  }
}
```

建表

```cpp
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema,
                                    Transaction *txn, TableInfo *&table_info) {
  if(table_names_.find(table_name)!=table_names_.end())//如果表名重复则返回相应结果
  {
    return DB_TABLE_ALREADY_EXIST;
  }
  else
  {
    table_id_t table_id=catalog_meta_->GetNextTableId();
    table_info=TableInfo::Create();
    TableHeap* table_heap=TableHeap::Create(buffer_pool_manager_,INVALID_PAGE_ID,schema,log_manager_,lock_manager_);
    auto root_page_id=table_heap->GetFirstPageId();
    TableMetadata* table_meta=TableMetadata::Create(table_id,table_name,table_heap->GetFirstPageId(),schema);//创建表元数据
    page_id_t page_id;
    Page *table_meta_page=buffer_pool_manager_->NewPage(page_id);
    char *buf=table_meta_page->GetData();
    table_meta->SerializeTo(table_meta_page->GetData());//将从缓存池获取的反序列信息序列化
    table_info->Init(table_meta,table_heap);

    //目标，将新表的信息存入表的map和目录
    tables_[table_id]=table_info;
    table_names_[table_name]=table_id;
    catalog_meta_->table_meta_pages_[table_id]=page_id;

    buffer_pool_manager_->UnpinPage(page_id,true);//取消固定
    FlushCatalogMetaPage();//写进磁盘
    return DB_SUCCESS;
  }
}
```

建立索引

```cpp
dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string &index_name,
                                    const std::vector<std::string> &index_keys, Transaction *txn,
                                    IndexInfo *&index_info, const string &index_type) {
  if(index_names_[table_name].find(index_name)!=index_names_[table_name].end())
  {
    return DB_INDEX_ALREADY_EXIST;
  }
  else if(table_names_.find(table_name)==table_names_.end())
  {
    return DB_TABLE_NOT_EXIST;
  }
  else
  {
    index_info=IndexInfo::Create();
    page_id_t table_id=table_names_[table_name];
    TableInfo *table_info=tables_[table_id];
    page_id_t index_id=catalog_meta_->GetNextIndexId();
    vector<uint32_t> key_map;
    auto schema=table_info->GetSchema();
    for(vector<string>::const_iterator it =index_keys.begin();it!=index_keys.end();it++)
    {
      uint32_t column_index;
      dberr_t result=schema->GetColumnIndex(it->data(),column_index);
      if(result==DB_SUCCESS)
      {
        key_map.push_back(column_index);
      }
      else
      {
        return DB_COLUMN_NAME_NOT_EXIST;
      }
    }
    IndexMetadata *meta_data=IndexMetadata::Create(index_id,index_name,table_id,key_map);
    page_id_t page_id;
    Page *index_meta_page=buffer_pool_manager_->NewPage(page_id);
    char* buf=index_meta_page->GetData();
    meta_data->SerializeTo(buf);
    index_info->Init(meta_data,table_info,buffer_pool_manager_);
    index_names_[table_name][index_name]=index_id;
    indexes_[index_id]=index_info;
    catalog_meta_->index_meta_pages_[index_id]=page_id;
    buffer_pool_manager_->UnpinPage(page_id,true);
    return DB_SUCCESS;
  }
}
```

删表

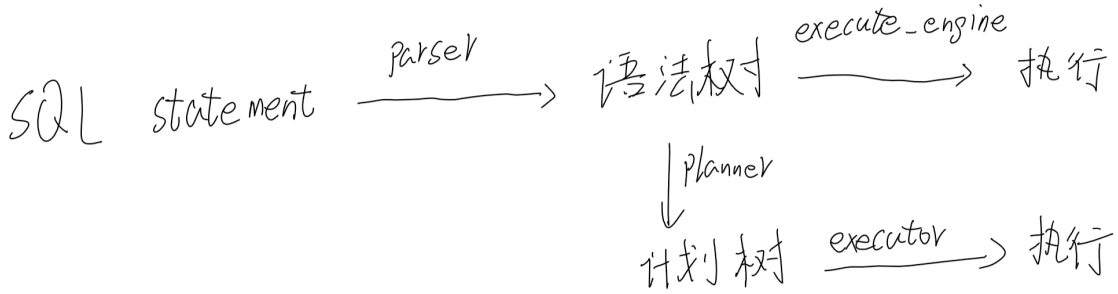
```
dberr_t CatalogManager::DropTable(const string &table_name) {
  if(table_names_.find(table_name)==table_names_.end())//要删除的表不存在
  {
    return DB_TABLE_NOT_EXIST;
  }
  else
  {
    page_id_t table_id=table_names_[table_name];
    //删除maps和目录中该表的信息
    tables_.erase(table_id);
    table_names_.erase(table_name);
    catalog_meta_->table_meta_pages_.erase(table_id);
    //删除该表下的索引
    if(index_names_.find(table_name)!=index_names_.end())
    {
      for(unordered_map<string,index_id_t>::iterator it=index_names_[table_name].begin();it!=index_names_[table_name].end();it++)
      {
        catalog_meta_->index_meta_pages_.erase(it->second);
        indexes_.erase(it->second);
      }
      index_names_.erase(table_name);
    }
    FlushCatalogMetaPage();
    return DB_SUCCESS;
  }
}
```

删除索引

```
dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name) {
  if(table_names_.find(table_name)==table_names_.end())//指定的表里没有要删除的索引
  {
    return DB_TABLE_NOT_EXIST;
  }
  else
  {
    if(index_names_.find(table_name)==index_names_.end()||index_names_.find(table_name)->second.find(index_name)==index_names_.find(table_name)
    {
      return DB_INDEX_NOT_FOUND;
    }
    else
    {
      page_id_t index_id=index_names_[table_name][index_name];
      IndexInfo *index_info=indexes_[index_id];
      //删除索引信息
      index_info->GetIndex()->Destroy();
      indexes_.erase(index_id);
      index_names_[table_name].erase(index_name);
      //当表没有索引时，将该表从索引map中删去
      if(index_names_[table_name].empty())//表没有索引了
      {
        index_names_.erase(table_name);
      }
      catalog_meta_->index_meta_pages_.erase(catalog_meta_->index_meta_pages_.find(index_id));
      FlushCatalogMetaPage();
      return DB_SUCCESS;
    }
  }
}
```

3.5 Planner and Executor

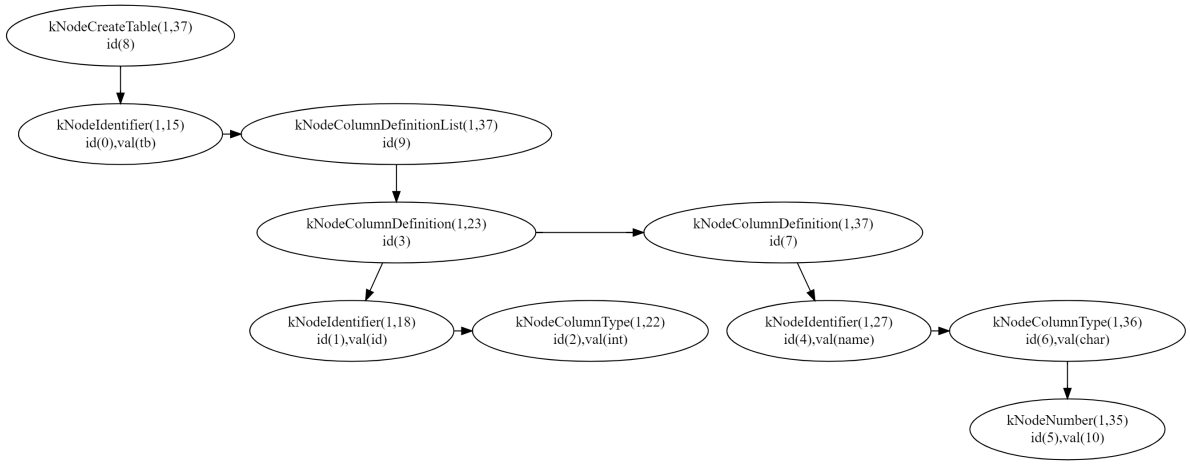
这个模块负责执行输入的sql语句。首先解释器parser用输入的SQL语句生成语法树，然后简单的语法树可以直接处理，复杂的语法树例如插入、删除等交由planner生成计划树，将一系列计划交给executor执行。



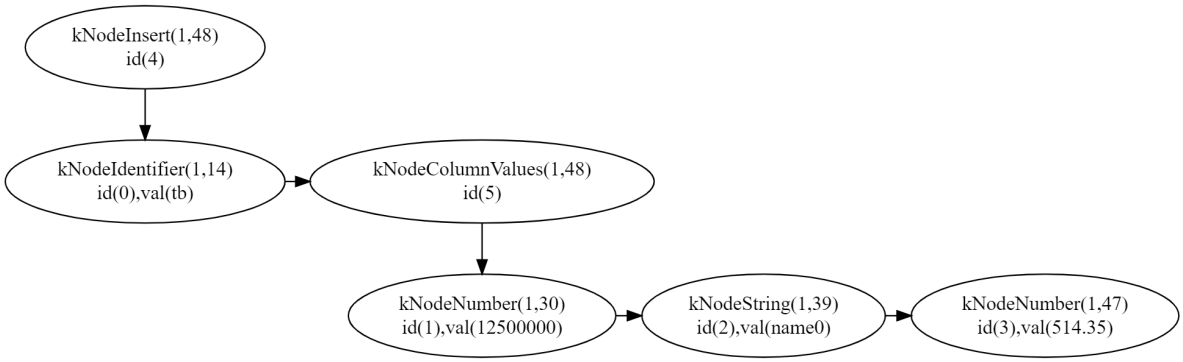
3.5.1语法树

语法树的节点包含id,type,value等信息以及child,next等指针，child指向的是下一级节点，next指向同级节点，例如：

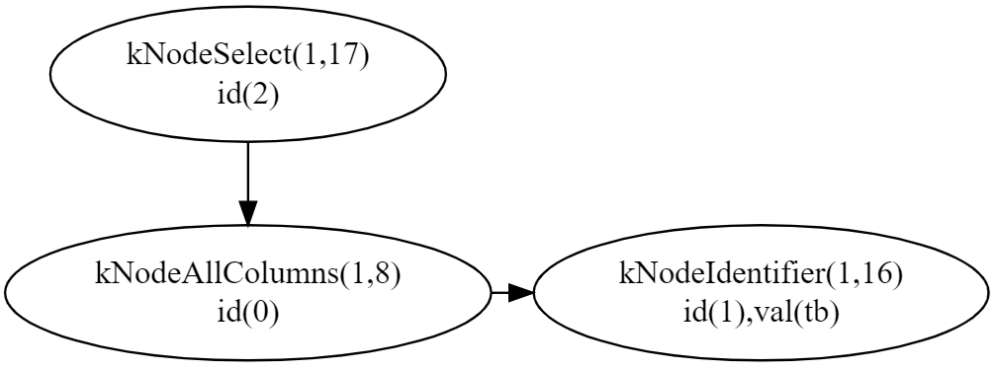
create table tb(id int,name char(16) unique,balance float,primary key(id));



insert into tb values(12500000，“name0”,514.35);



select * from tb;



3.5.2语句示例

parser模块支持的sql语句，即可以生成语法树的语句包括：

- create database <database name>
- drop database <database name>
- show databases
- use <database name>
- create table <name> (<column name> <column type>,.....)
- drop table <table name>
- show tables

- select <column name> from <table name> where <column name>=<value>.....
- insert into <table name> values(<value>,.....)
- .....

3.5.3计划执行

生成语法树后会向executeEngine模块传递语法树的根节点，executeEngine首先会根据根节点的类型进行判断，如果根节点是select,insert等较复杂的类型，则传入planner生成计划树，再将计划树的根节点传给executor执行；反之直接由executeEngine执行，如数据库、表等的添加与删除。

本实验中executor模块采用的算子执行模型为火山模型，即iterator model。每个算子有两个方法：init和next，init进行初始化，next向下层算子请求下一条数据。next的返回类型为bool，当返回false时说明没有下一条数据，执行结束。

本实验共有6个执行算子：

name	function
seqscan	对表进行顺序扫描，每次next返回满足条件的一行。
indexscan	对表按索引进行遍历，每次next返回满足条件的一行。
insert	向表中插入一行，数据从valueExecutor获得，还需要处理唯一性约束等问题。
update	更新表中的行，从seqscan获得，需要处理唯一性约束等问题。更新分为删除和添加两个步骤处理。
value	从计划树中得到需要使用的数据。
delete	删除符合条件的行，这些行由seqscan获取。

3.5.4主要函数实现

执行建库语句

```
dberr_t ExecuteEngine::ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateDatabase" << std::endl;
#endif
  string database_name = ast->child_->val_;//从语法树的结点获得数据库的名字
  //如果同名数据库存在返回失败。
  if (dbs_.find(database_name) == dbs_.end()) {
    dbs_.emplace(database_name, new DBStorageEngine("./"+database_name));
    cout << "Create database " << database_name << " successfully." << endl;
    return DB_SUCCESS;
  }
  else {
    cout << "Error:Database " << database_name << " already exists." << endl;
    return DB_FAILED;
  }
}
```

执行删库语句

```
dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropDatabase" << std::endl;
#endif
  string database_name = ast->child_->val_;//从语法树的结点获得数据库的名字
  //如果该数据库不存在则返回失败
  if (dbs_.find(database_name) != dbs_.end())
  {
    if(current_db_==database_name)
    {
      current_db_="";
    }//drop当前使用的db的话current_db_也要改变
    remove(("./database/" + database_name + ".db").c_str());
    dbs_.erase(database_name);
    cout << "Drop database " << database_name << " successfully." << endl;
    return DB_SUCCESS;
  }
  else
  {
    cout << "Error:Database " << database_name << " doesn't exist." << endl;
    return DB_FAILED;
  }
}
```

查看数据库

```
dberr_t ExecuteEngine::ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteShowDatabases" << std::endl;
#endif
  //没有数据库时提示
  if(dbs_.empty())
  {
    cout<<"No database."<<endl;
    return DB_SUCCESS;
  }
  else//设置输出格式
  {
    int width=0;
    for(unordered_map<string,DBStorageEngine*>::iterator it = dbs_.begin();it!=dbs_.end();it++)
    {
      width=width>it->first.length()?width:it->first.length();
    }
    cout << "+" << setw(width+3) << setfill('-')<<"+"<<endl;
    for(unordered_map<string,DBStorageEngine*>::iterator it =dbs_.begin();it!=dbs_.end();it++)
    {
      cout<<"|"<<setw(width)<<setfill(' ')<<it->first<<setw(width+1)<<setfill(' ')<<"|"<<endl;
    }
    cout << "+" << setw(width+3) << setfill('-')<<"+"<<endl;
    return DB_SUCCESS;
  }
}
```

执行建表语句

```cpp
dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateTable" << std::endl;
#endif
  if (dbs_.find(current_db_) != dbs_.end()&&ast!=nullptr) {
    string table_name=ast->child_->val_;
    TableInfo *table_info;
    //检查表是否存在
    if(dbs_[current_db_]->catalog_mgr_->GetTable(table_name,table_info)==DB_SUCCESS)
    {
      cout<< "Error:Table already exist."<<endl;
      return DB_FAILED;
    }
    else
    {
      SyntaxNode *node = ast->child_->next_->child_;
      vector<Column*>columns;
      vector<vector<string> > unique_indexes;
      vector<string> primary_keys;
      //遍历语法树，找到全部列
      for(uint32_t column_index=0;node!=nullptr&&node->type_==kNodeColumnDefinition;column_index++,node=node->next_)
      {
        string column_name=node->child_->val_;
        string column_type=node->child_->next_->val_;

        bool unique=false;
        //unique index
        string str=(node->val_==nullptr)?"":node->val_;
        if(str=="unique")
        {
          unique=true;
          vector<string> unique_index;
          unique_index.push_back(node->child_->val_);
          unique_indexes.push_back(unique_index);
        }

        //table
        if(column_type=="int")
        {
          Column* column=new Column(column_name, kTypeInt, column_index, true, unique);
          columns.push_back(column);
        }
        else if(column_type=="float")
        {
          Column* column=new Column(column_name, kTypeFloat, column_index, true, unique);
          columns.push_back(column);
        }
        else if(column_type=="char")
        {
          uint32_t length = atoi(node->child_->next_->child_->val_);
          Column* column=new Column(column_name, kTypeChar, length, column_index, true, unique);
          columns.push_back(column);
        }
        else
        {
          cout<<"Error:Invalid type."<<endl;
          return DB_FAILED;
        }
      }
      Schema *schema = new Schema(columns,true);
      if(dbs_[current_db_]->catalog_mgr_->CreateTable(table_name,schema,nullptr,table_info)!=DB_SUCCESS)
      {
        return DB_FAILED;
      }
      //继续遍历语法树，找主码
      //primary_key_index
      if(node!=nullptr)
      {
        SyntaxNode* PK_node=node->child_;
        while(PK_node!=nullptr)
        {
          string primary_key=PK_node->val_;
          primary_keys.push_back(primary_key);
          PK_node=PK_node->next_;
        }
        if(!primary_keys.empty())
        {
          string pk_index_name=table_name+"_pk_index";
          IndexInfo *index_info;
          dberr_t result=dbs_[current_db_]->catalog_mgr_->CreateIndex(table_name,pk_index_name,primary_keys,nullptr,index_info,"bptree");
          if(result!=DB_SUCCESS)
          {
            cout<<"Error:Primary key index "<<pk_index_name<<" create fail."<<endl;
            return DB_FAILED;
          }
        }
      }
      //unique_index
```

```
    for(auto it=unique_indexes.begin();it!=unique_indexes.end();it++)
    {
      string uq_index_name=table_name+"_uq_"+(*it)[0];
      IndexInfo *index_info;
      dberr_t result=dbs_[current_db_]->catalog_mgr_->CreateIndex(table_name,uq_index_name,*it,nullptr,index_info,"bptree");
      if(result!=DB_SUCCESS)
      {
        cout<<"Error:Unique index "<<uq_index_name<<" create fail."<<endl;
        return DB_FAILED;
      }
    }
  }
  cout<<"Create table successfully."<<endl;
  return DB_SUCCESS;
}
else
{
  cout << "Error:No using database." << endl;
  return DB_FAILED;
}
}
```

执行删表语句

```
dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropTable" << std::endl;
#endif
  if (dbs_.find(current_db_) != dbs_.end()) {
    string table_name = ast->child_->val_;
    dberr_t result = dbs_[current_db_]->catalog_mgr_->DropTable(table_name);
    if (result == DB_FAILED) {
      cout << "Error:Drop failed." << endl;
      return DB_FAILED;
    }
    else if (result == DB_TABLE_NOT_EXIST) {
      cout << "Error:Table " << table_name << " not exist." << endl;
      return DB_FAILED;
    }
    else {
      cout << "Drop table " << table_name << " successfully." << endl;
      return DB_SUCCESS;
    }
  }
  else {
    cout << "Error:No using database." << endl;
    return DB_FAILED;
  }
}
```

执行创建索引语句

```cpp
dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateIndex" << std::endl;
#endif
  SyntaxNode *node = ast->child_;
  if(dbs_.find(current_db_)!=dbs_.end()&&ast!=nullptr)
  {
    string index_name=node->val_;
    string table_name=node->next_->val_;
    TableInfo *table_info;
    IndexInfo *index_info;
    if(dbs_[current_db_]->catalog_mgr_->GetTable(table_name,table_info)!=DB_SUCCESS)
    {
      cout<<"Error:Table not exist."<<endl;
      return DB_FAILED;
    }
    else if(dbs_[current_db_]->catalog_mgr_->GetIndex(table_name,index_name,index_info)==DB_SUCCESS)
    {
      cout<<"Error:Index already exist."<<endl;
      return DB_FAILED;
    }
    else
    {
      node=node->next_->next_;
      if(node->type_!=kNodeColumnList)
      {
        cout<<"Error:Index key not exist."<<endl;
        return DB_FAILED;
      }
      node=node->child_;
      vector<string> index_keys;
      while(node!=nullptr)
      {
        index_keys.push_back(node->val_);
        node=node->next_;
      }
      vector<string>::iterator it;
      for(it=index_keys.begin();it!=index_keys.end();it++)
      {
        uint32_t column_index;//初始化
        if(table_info->GetSchema()->GetColumnIndex(*it,column_index)==DB_COLUMN_NAME_NOT_EXIST)
        {
          cout<<"Eror:Column not exist."<<endl;
          return DB_FAILED;
        }
        if(table_info->GetSchema()->GetColumn(column_index)->IsUnique())
        {
          break;
        }
      }
      if(it==index_keys.end())//无unique索引
      {
        cout<<"Error:At least one unique index shuold exist."<<endl;
        return DB_FAILED;
      }
      IndexInfo *index_info_1;
      dberr_t result=dbs_[current_db_]->catalog_mgr_->CreateIndex(table_name,index_name,index_keys,nullptr,index_info_1,"bptree");
      if(result!=DB_SUCCESS)
      {
        cout<<"Error: Create index fail."<<endl;
        return DB_FAILED;
      }
      else
      {
        uint32_t *column_index=new uint32_t[index_keys.size()];
        for(int i=0;i<index_keys.size();i++)
        {
          table_info->GetSchema()->GetColumnIndex(index_keys[i],column_index[i]);
        }
        auto table_heap= table_info->GetTableHeap();
        for(auto iter = table_heap->Begin(nullptr);iter!=table_heap->End();++iter)
        {
          vector<Field>fields;
          for(int i =0;i<index_keys.size();i++)
          {
            fields.push_back((*iter->GetField(column_index[i])));
          }
          Row row(fields);
          RowId rid = iter->GetRowId();
          index_info->GetIndex()->InsertEntry(row,rid,nullptr);
        }
        cout<<"Create index successfully."<<endl;

        return DB_SUCCESS;
      }
    }
  }
  else
  {
```

```
    cout<<"Error:No using database."<<endl;
    return DB_FAILED;
  }
}
```

执行删除索引语句

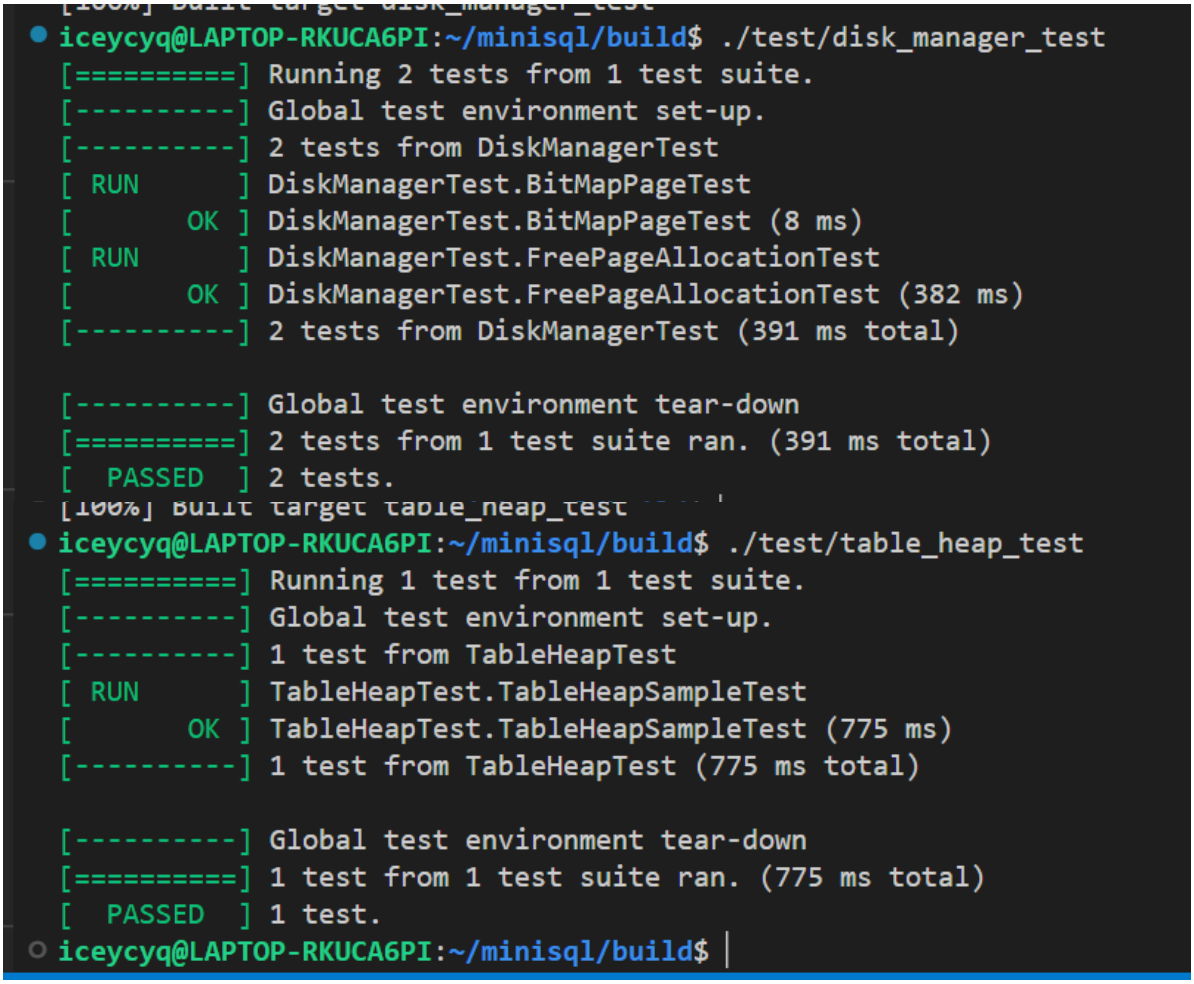
```
dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropIndex" << std::endl;
#endif
  if (dbs_.find(current_db_) != dbs_.end()) {
    string index_name_drop=ast->child_->val_;
    vector<TableInfo*> tables;
    dbs_[current_db_]->catalog_mgr_->GetTables(tables);
    if(tables.empty())
    {
      cout<<"Error:No tables."<<endl;
      return DB_FAILED;
    }
    else {
      for (auto it = tables.begin(); it != tables.end(); it++)
      {
        string table_name= (*it)->GetTableName();
        vector<IndexInfo*> indexes;
        dbs_[current_db_]->catalog_mgr_->GetTableIndexes(table_name, indexes);
        for (vector<IndexInfo *>::iterator it2 = indexes.begin(); it2 != indexes.end(); it2++)
        {
          string index_name= (*it2)->GetIndexName();
          if (index_name == index_name_drop)
          {
            dbs_[current_db_]->catalog_mgr_->DropIndex(table_name, index_name_drop);
            cout << "Drop index " << index_name_drop << " successfully." << endl;
            return DB_SUCCESS;
          }
        }
      }
      cout << "Error:Index " << index_name_drop << " not found." << endl;
      return DB_FAILED;
    }
  }
  else {
    cout << "Error:No using database." << endl;
    return DB_FAILED;
  }
}
```

4 测试

4.1Program Test

```
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/buffer_pool_manager_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (0 ms)
[----------] 1 test from BufferPoolManagerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/lru_replacer_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (0 ms)
[----------] 1 test from LRUReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/catalog_test
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (46 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (38 ms)
[----------] 3 tests from CatalogTest (84 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (84 ms total)
[  PASSED  ] 3 tests.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/b_plus_tree_index_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (39 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (17 ms)
[----------] 2 tests from BPlusTreeTests (57 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (57 ms total)
[  PASSED  ] 2 tests.
```

```
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/b_plus_tree_test
[==========] Running 2 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from MyBPlusTreeTest
[ RUN      ] MyBPlusTreeTest.MySampleTest
[       OK ] MyBPlusTreeTest.MySampleTest (749 ms)
[----------] 1 test from MyBPlusTreeTest (749 ms total)

[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (78 ms)
[----------] 1 test from BPlusTreeTests (78 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test suites ran. (828 ms total)
[  PASSED  ] 2 tests.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/index_iterator_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (30 ms)
[----------] 1 test from BPlusTreeTests (30 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (30 ms total)
[  PASSED  ] 1 test.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/index_roots_page_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from PageTests
[ RUN      ] PageTests.IndexRootsPageTest
[       OK ] PageTests.IndexRootsPageTest (0 ms)
[----------] 1 test from PageTests (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/tuple_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TupleTest
[ RUN      ] TupleTest.FieldSerializeDeserializeTest
[       OK ] TupleTest.FieldSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (0 ms)
[----------] 2 tests from TupleTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```


iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/disk_manager_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from DiskManagerTest
[RUN] DiskManagerTest.BitMapPageTest
[OK] DiskManagerTest.BitMapPageTest (8 ms)
[RUN] DiskManagerTest.FreePageAllocationTest
[OK] DiskManagerTest.FreePageAllocationTest (382 ms)
[----------] 2 tests from DiskManagerTest (391 ms total)
[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (391 ms total)
[PASSED] 2 tests.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$ ./test/table_heap_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TableHeapTest
[RUN] TableHeapTest.TableHeapSampleTest
[OK] TableHeapTest.TableHeapSampleTest (775 ms)
[----------] 1 test from TableHeapTest (775 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (775 ms total)
[PASSED] 1 test.
iceycyq@LAPTOP-RKUCA6PI:~/minisql/build$

4.2Main Test

```
minisql > create database db;
[INFO] Sql syntax parse ok!
Create database db successfully.
minisql > show databases;
[INFO] Sql syntax parse ok!
+----+
|db  |
+----+
minisql > use db;
[INFO] Sql syntax parse ok!
Use database db successfully.
minisql > create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
[INFO] Sql syntax parse ok!
Create table successfully.
minisql > show tables;
[INFO] Sql syntax parse ok!
+---------+
|account  |
+---------+
minisql > insert into account values(12500000, "name00000", 514.35);
1 rows affected (0.005829s)
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12500000 | name00000 | 514.349976 |
+----------+-----------+------------+
1 rows affected (0.000303s)
minisql > delete from account where name = "name00000";
minisql > select * from account;
empty set
0 rows affected (0.000159s)
```

```
minisql > create index idx01 on account(name);
Create bptree index idx01 OK
1 rows affected (0.003932s)
minisql > show indexes;
+-----------------+
| Index           |
+-----------------+
| idx01           |
| account_name    |
| account_PRIMARY |
+-----------------+
3 rows affected (0.00022s)
```

```
minisql > drop index idx01;
Drop index idx01 OK
1 rows affected (0.002007s)
minisql > show indexes;
+-----------------+
| Index           |
+-----------------+
| account_name    |
| account_PRIMARY |
+-----------------+
+----------+----------+-----------+
| id       | name     | balance   |
+----------+----------+-----------+
| 12500000 | name00000 | 514.349976 |
| 12500001 | name00001 | 103.139999 |
| 12500002 | name00002 | 981.859985 |
| 12500003 | name00003 | 926.510010 |
+----------+----------+-----------+
```

```
minisql > select name from account where balance > 600;
+-----------+
| name      |
+-----------+
| name00002 |
| name00003 |
+-----------+
2 rows affected (0.000471s)
```

5心得与体会

早在大一时就听说了ZJU的特色大程miniSQL，如今也算是完成了这一挑战，做出自己的DBMS并看着他能实现诸多的功能实在有一种成就感，这是其他大程比不了的。

本次大程我们选择了使用助教提供的现有框架，补充主要函数以实现完整项目，虽然省去了部分造轮子的时间，但是通过略显简单的实验大纲和如此庞大的源代码想要理解框架也不是一件容易的事，需要掌握许多知识。

本次项目中由我负责的模块是Buffer Pool Manager 和 Record Manager，是整个项目运行的最底层，我深知在底层bug往往是最难发现且影响最大的，这也导致了我在实现函数时十分谨慎，争取每个函数都不留bug（不幸的是还是有部分bug在后期才发现，如TableIterator的==操作符和row的序列化中都有部分bug）。并且作为小组中最先开始此项目的成员，我在理解框架时得到了组员们的许多帮助，同样，我在队友进行Index Manager时也会向队友解释缓冲池的原理，序列化的原理等。自我认为本人对项目还有一个比较大的贡献在于倡导使用`glog`进行调试，通过在程序中加LOG和breakpoint的方式可以比较方便的定位bug并进行修复。

通过前两个模块的编写，我在实践中弄懂了bitmap的意义，lru算法的原理，缓冲池的原理以及清楚了数据在数据库中是如何实现存储的，这对我整个数据库的学习都起到了很大的帮助。除了对于数据库的理解加深，我还更熟悉了C++的使用和调试以及如何使用git进行团队合作开发，这都是实际生产中不可不学的知识。

记忆比较深刻的是在验收的前几天一起debug的时候被一个问题卡到了凌晨四点，最后发现是test有问题，希望下一次数据库开课时，框架中能修复那个bug。

总的来说，miniSQL虽然难但是收获很大，回想起写大程的一幕幕，有因为修不好bug而沉默破防两天，有写完commit之后的放松愉悦，有debug定位到自己负责的模块的问题时的紧张尴尬，有一起熬夜一起为一个bug苦恼一起为一次commit兴奋的奇妙体验，miniSQL带来的不仅仅是能力的提升，更是一个学生从只会刷题的做题家变化成为有实际项目开发经验的入门级别程序员😊！