# 数据库系统夏学期大程

# minisql实验报告

# 一、实验目的

1. 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。
2. 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。

# 二、实验需求

## 2.1 需求概述

### 2.1.1 数据类型

要求支持三种基本数据类型：`integer`，`char(n)`，`float`。

### 2.1.2 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。

### 2.1.3 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。

### 2.1.4 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。

### 2.1.5 开发方式

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。

## 2.2 语法说明

MiniSQL 支持标准的SQL 语句格式，每一条SQL 语句以分号结尾，一条SQL 语句可写在一行或多行。为简化编程，要求所有的关键字都为小写。在以下语句的语法说明中，用黑体显示的部分表示语句中的原始字符串，如create 就严格的表示字符串“create”，其他非黑体显示的有其他的含义，如表名并不是表示字符串 “表名”，而是表示表的名称。

### 2.2.1 创建表语句

该语句的语法如下：

```
create table 表名 (
    列名 类型 ,
    列名 类型 ,
    列名 类型 ,
    primary key ( 列名 )
);
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

### 2.2.2 删除表语句

该语句的语法如下：

```
drop table 表名 ;
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

### 2.2.3 创建索引语句

该语句的语法如下：

```
drop index 索引名 ;
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

### 2.2.4 选择语句

该语句的语法如下：

```
select * from 表名 ;
```

或

```
select * from 表名 where 条件 ;
```

其中“条件”具有以下格式： 列 op 值 and 列 op 值 ... and 列 op 值
op 是算术比较符： =，<，>，< >，<=，>=
若该语句执行成功且查询结果不为空，则按行输出查询结果，第一行为属性名，其余每一行表示一条记录；若查询结果为空，则输出信息告诉用户查询结果为空；若失败，必须告诉用户失败的原因。

### 2.2.4 插入记录语句

该语句的语法如下：

```
insert into 表名 values ( 值1 , 值2 , … , 值n );
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

### 2.2.5 删除记录语句

该语句的语法如下：

```
delete from 表名 ;
```

或

```
delete from 表名 where 条件 ;
```

若该语句执行成功，则输出执行成功信息，其中包括删除的记录数；若失败，必须告诉用户失败的原因。

### 2.2.6 退出MiniSQL 系统语句

该语句的语法如下：

```
quit;
```

### 2.2.7 执行SQL 脚本文件语句

该语句的语法如下：

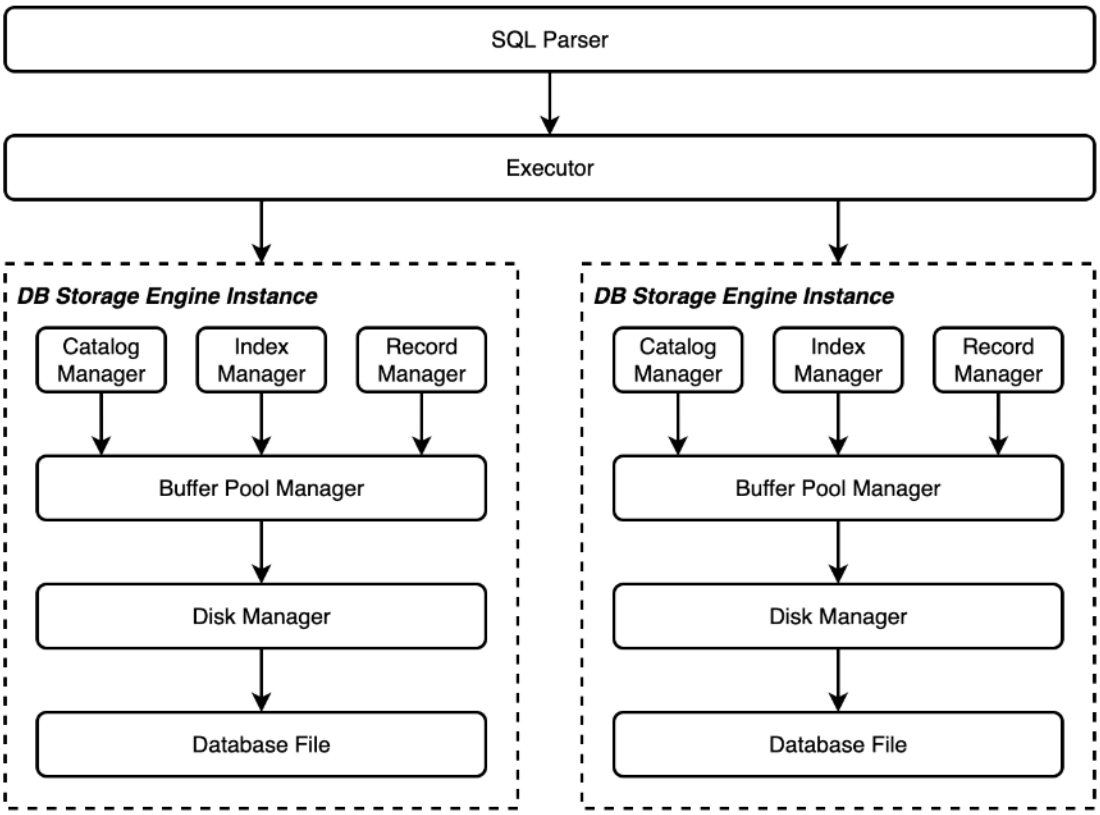
```
execfile 文件路径 ;
```

SQL 脚本文件中可以包含任意多条上述8 种SQL语句，MiniSQL 系统读入该文件，然后按序依次逐条执行脚本中的 SQL 语句。

# 三、实验环境

- 开发语言C++
- 开发系统WSL2, Linux发行版为Ubuntu 20.04
- 编译器g++，版本为9.3.0
- 项目管理工具cmake，版本为3.16.3
- 使用git进行版本控制

# 四、系统架构




# 五、各模块概述

# 5.1 Disk and Buffer Pool Manager

## 5.1.1 实验内容

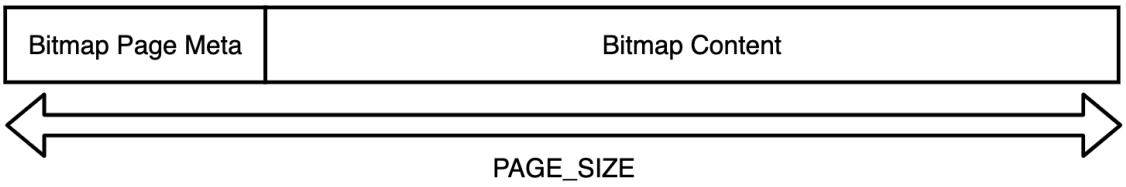
- 实现Disk Manager模块
- 实现Buffer Pool Manager模块

## 5.1.2 具体实现

### 5.1.2.1 数据页管理——位图页(bitmap)

位图页是数据库文件中的一类特殊页，用于管理一段连续的数据页。位图页由两部分组成：元数据和其管理的数据页的分配情况。简便起见，元数据内仅存储当前已分配的页的数量，其管理的数据页的分配情况用一个`char`数组表示。数组内第`i`位为1代表该位图页之后的第`i`页已经被分配，反之该页空闲。

一个位图页如图所示。




此部分需要实现函数：

- `BitmapPage::AllocatePage(&page_offset)`：

  用途：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）。

  实现方式：线性扫描找到第一个未被分配的页，将该页标记为分配状态，同时数据页分配数量+1。

- `BitmapPage::DeAllocatePage(page_offset)`：

  用途：回收已经被分配的页。

  实现方式：将page_offset对应位变为0，同时数据页分配数量-1。

- `BitmapPage::IsPageFree(page_offset)`：

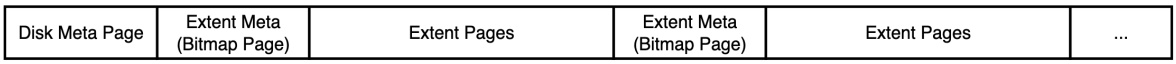
  用途：判断给定的页是否是空闲（未分配）的。

  实现方式：page_offset的分配情况位于数组中第page_offset/8个字节的第page_offset%8位，判断其是否为1。

### 5.1.2.2 磁盘管理

为了实现更大的存储空间，磁盘的所有页需要多个位图页进行管理，因此将一个位图页和它管理的所有数据页视为磁盘中的一个分区，并使用`Disk Meta Page`，即数据库文件中的第`0`个数据页对所有分区进行统一管理。

Disk Meta Page中记录了数据页分配的总页数，分区数以及各分区已分配数据页的数量。磁盘结构如图：




在DiskManager中实现了如下函数：

- `DiskManager::AllocatePage()`：

  用途：从磁盘中分配一个空闲页，并返回空闲页的**逻辑页号**。

  实现方式：在元数据页中线性查找，直到找到一个未满的分区，调用该分区位图页的`AllocatePage`函数。

- `DiskManager::DeAllocatePage(logical_page_id)`：

  用途：释放磁盘中**逻辑页号**对应的物理页。

  实现方式：找到该页号对应的分区，在对应位图页调用`DeAllocatePage`函数。

- `DiskManager::IsPageFree(logical_page_id)`：

  用途：判断该**逻辑页号**对应的数据页是否空闲。

  实现方式：找到该页号对应的分区，在对应位图页调用`IsPageFree`函数。

- `DiskManager::MapPageId(logical_page_id)`：

  用途：该函数可以用于将逻辑页号转换成物理页号。

  实现：磁盘中逻辑页号与物理页号的对应关系如图：

| 物理页号 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 职责 | 磁盘元数据 | 位图页 | 数据页 | 数据页 | 数据页 |
| 逻辑页号 | / | / | 0 | 1 | 2 |

  由图可知，物理页码 = 逻辑页码 + 逻辑页码 / 位图页能管理的最大页数 + 2

### 5.1.2.3 替换策略：LRU & Clock(bonus)

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

当缓存池中没有空闲页时，需要采用一定的替换策略决定替换哪个数据页。可以使用LRU和Clock替换策略。

LRU会将上次使用时间最早，使用次数最少的页面优先替换。Clock会为页表内的每一页分配一个使用位并维护一个“指针”。初始时或页被固定时使用位均为EMPTY， 解除固定时使用位为ACCESSED。需要替换时“指针”循环遍历每一页，遇到ACCESSED页则将其使用位置为UNUSED，遇到UNUSED页则将其作为替换页返回。

需要实现的函数：

- `LRUReplacer::Victim(*frame_id)`：

  用途：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`；

  实现方式：维护双向链表`lru_list`和用于实现随机访问的`lru_hash_map`。需要替换时删除`lru_list`的最后一个节点。

- `LRUReplacer::Pin(frame_id)`：

  用途：将数据页固定使之不能被`Replacer`替换

  实现方式：即从`lru_list_`中移除该数据页对应的页帧。

- `LRUReplacer::Unpin(frame_id)`：

  用途：将数据页解除固定

  实现方式：将该页放入`lru_list_`中，使之可以在必要时被`Replacer`替换掉。

- `LRUReplacer::Size()`：

  用途：返回当前`LRUReplacer`中能够被替换的数据页的数量。

#### 5.1.2.4 缓冲池管理

数据库系统中，所有内存页面都由`Page`对象表示，其包含了一段连续的数据（内存页）与其是否脏页，固定页面的线程数等。

`BufferPoolManager`用于为其他模块提供对数据库进行直接管理的接口。一方面，其调用`DiskManager`实现数据页的分配和回收，并实现数据页替换；另一方面，其他模块需要使用数据页时只需调用其内部的`BufferPoolManager`成员，无需直接对磁盘进行操作，从而提高了封装性。

需要实现函数：

- `BufferPoolManager::FetchPage(page_id)`：

  用途：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。

  实现方式：先检查该页是否在内存中。如不在，将该页替换进入内存。之后通过page_table获取该页的frame_id，再通过pages_获取该页指针，最后将该页固定。

- `BufferPoolManager::NewPage(&page_id)`：

  用途：分配一个新的数据页，并将逻辑页号于`page_id`中返回。

  实现方式：调用`AllocatePage`函数分配一页，并替换掉内存中的一页（优先替换`free_list`）。将其添加入page_table并返回该页指针。

- `BufferPoolManager::UnpinPage(page_id, is_dirty)`：

  用途：取消固定一个数据页。

  实现方法：固定数-1，如果`is_dirty`为真，则将其写回磁盘。

- `BufferPoolManager::FlushPage(page_id)`：

  用途：将数据页转储到磁盘中。

  实现方法：获取该页的逻辑页数，调用`disk_manager`的`WritePage`函数。

- `BufferPoolManager::DeletePage(page_id)`：

  用途：释放一个数据页。

  实现方法：调用`disk_manager`的`DeallocatePage`函数。

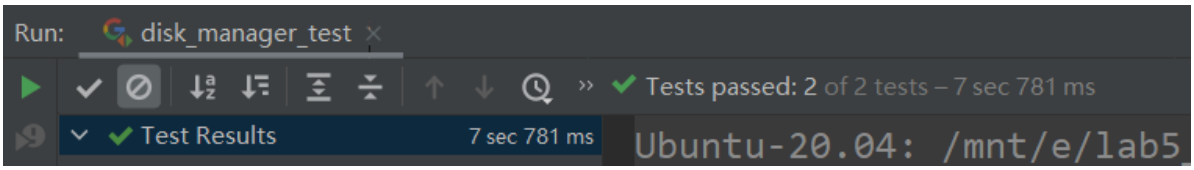
- `BufferPoolManager::FlushAllPages()`：

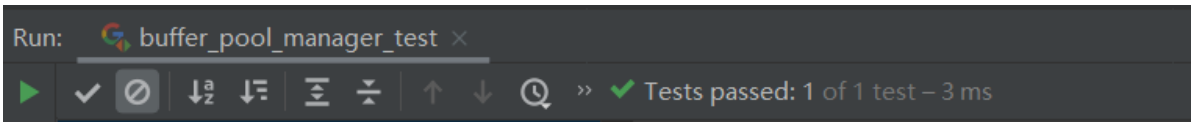
  用途：将所有的页面都转储到磁盘中。

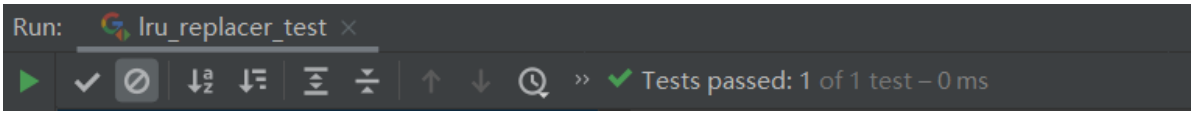
  实现方法：对页表中所有页执行`FlushPage`操作。

# 实验结果

该模块代码通过了所有测试文件，结果如图。




将lru_replacer_test中的LRUReplacer替换为ClockReplacer也可通过测试。

# 5.2 Record Manager

## 5.2.1 模块功能概述

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。与记录有关相关的概念有以下几个：列（`Column`）：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、是否唯一等等；模式（`Schema`）：用于表示一个数据表或是一个索引的结构。一个`Schema`由一个或多个的`Column`构成；域（`Field`）：它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；行（`Row`）：与元组的概念等价，用于存储记录或索引键，一个`Row`由一个或多个`Field`构成。

Record Manager使用堆表来管理记录，并且可以通过序列化和反序列化的操作将数据转化为字节流，向堆表中存储/读取。堆表（`TableHeap`）是一种将记录以无序堆的形式进行组织的数据结构，不同的数据页（`TablePage`）之间通过双向链表连接。堆表中的记录通过`RowId`进行定位。`RowId`记录了该行记录所在的`page_id`和`slot_num`，其中`slot_num`用于定位记录在这个数据页中的下标位置。

堆表中的每个数据页（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，与之相关的代码位于中，表头在页中从左往右扩展，记录了`PrevPageId`、`NextPageId`、`FreeSpacePointer`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。

在本模块中，我们实现了Row, Field, Schema, Column的序列化和反序列化，为上级模块提供了记录的插入、更新、标记删除、物理删除、获取元组以及迭代器遍历记录的相关功能。

## 5.2.2 各函数具体实现

#### 5.2.2.1 序列化与反序列化

- `Row::SerializeTo(*buf, *schema)`
- `Row::DeserializeFrom(*buf, *schema)`
- `Row::GetSerializedSize(*schema)`
- `Column::SerializeTo(*buf)`
- `Column::DeserializeFrom(*buf, *&column, *heap)`
- `Column::GetSerializedSize()`
- `Schema::SerializeTo(*buf)`
- `Schema::DeserializeFrom(*buf, *&schema, *heap)`
- `Schema::GetSerializedSize()`

这些函数里`SerializeTo`是将类里的成员变量转化为字节流存储在数据页中；`GetSerializedSize`是获取产生字节流的长度，单位是字节；`DeserializeFrom`是根据`schema`将数据从`buf`指向的数据页中反序列为对应的类。这里面`Field`是记录`Row`里的一个字段，序列化和反序列化是由`Row`调用的。同理`Column`也是由`Schema`管理的。

#### 5.2.2.2 堆表及Record Manager相关操作

实现的函数及功能简介如下所示：

- `TableHeap:InsertTuple(&row, *txn)`：

  用法：`row`用引用的方式传入一条记录，`InsertTuple`函数采用First Fit的方式从第一个数据页开始搜索剩余空间能够存储该记录的数据页，然后调用`TablePage`里的`InsertTuple`函数实现数据的插入。插入记录后生成的`RowId`需要通过`row`对象返回（即`row.rid_`）；

- `TableHeap:UpdateTuple(&new_row, &rid, *txn)`：

  用法：将`RowId`为`rid`的记录`old_row`替换成新的记录`new_row`，并将`new_row`的`RowId`通过`new_row.rid_`返回。我们先调用`TablePage::UpdateTuple`，如果插入成功则结束；如果未找到该条记录，则更新失败；如果该记录已被删除，则不更新；如果新纪录长度过长，在当前数据页装不下，则删除该记录，然后重新插入新纪录。

- `TableHeap:ApplyDelete(&rid, *txn)`：

  用法：直接调用`TablePage::Applydelete`从物理意义上删除这条记录

- `TableHeap:GetTuple(*row, *txn)`：获取`RowId`为`row->rid_`的记录；

  用法：通过`row`传入`rowid`，然后根据`rowid`里的page_id和slot_id，在数据页中寻找对应的记录，然后反序列化到`row`。

- `TableHeap:FreeHeap()`：销毁整个`TableHeap`并释放这些数据页；

  用法：这里主要就是调用一些析构函数。

- `TableHeap::Begin()`：获取堆表的首迭代器；
- `TableHeap::End()`：获取堆表的尾迭代器；
- `TableIterator::operator++()`：移动到下一条记录，通过`++iter`调用；
- `TableIterator::operator++(int)`：移动到下一条记录，通过`iter++`调用。

用法：上面这几个迭代器相关的函数就是用来遍历一个表对应的所有记录。

#### 5.2.2.3 堆表操作的优化(bonus)

**Bonus:** 优化堆表（`TableHeap`）以及数据页（`TablePage`）的实现，通过使用额外的空间记录一些元信息来加速`Row`的插入、查找和删除操作。

在性能测试时我们发现，数据的插入操作时制约性能提升的一个重要因素。因为根据框架提供的插入方法，程序需要按照First Fit的方法，从第一个数据页开始，一页一页查找可以容纳该条记录的数据页，这在大量数据插入时需要$O(N)$的时间，性能很不好。所以我们决定采用Next Fit的方法，通过记录每次插入数据所在的数据页，使得下一次插入都从这个数据页开始搜索，可以将这部分耗时降低至$O(1)$。

具体实现方式是在`TableHeap`里定义一个`current_page_id`的成员变量，每次插入结束时都利用`current_page_id = current_page->GetPageId();`更新`current_page_id`的值，使得查找插入位置不用从`first_page_id`搜索，优化插入效率。

### 5.4.3 单元测试结果

元组测试（测试row, schema, field, column四个类的序列化与反序列化操作是否成功）

```
[==========] 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.
```

堆表测试（插入10000条记录，更新400条，删除200条）

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (2008 ms)
[----------] 1 test from TableHeapTest (2008 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (2008 ms total)
[  PASSED  ] 1 test.
```

## 5.3 Index Manager

## 5.3.1 模块功能描述

Index Manager 负责数据表索引的实现和管理，包括：索引的创建和删除，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。本模块采用B+树索引。
B+树种的类主要括`BPlusTree`本身和每一个结点对应的`BplusTreePage`，`BplusTreePage`又分为内部结点`BplusTreeInternalPage`和叶结点`BplusTreeLeafPage`。叶子结点存储`key`以及相应记录的地址，内部节点各层作为索引使用。
当一个结点因插入`key`而包含的`key`过多，则会发生`Split`；相反，当一个结点因删除`key`而包含的`key`过少，则会与兄弟结点调整`key`分配或者删除结点。
插入索引时，`BPlusTree`元素自底向上插入，通过不断`BPlusTreePage`的不断`Split`和向上新建，搭建起自顶向下查询速度较快索引功能。同时与 BufferPool Manager 模块相连，通过`page_id`从内存中获取目标`Page`，进行修改。

## 5.3.2 各函数具体实现

### 5.3.2.1 BPlusTreePage结点实现

#### 5.3.2.1.1 BPlusTreePage 基类

基类函数大多为获取和设置`private`变量的值，略过。

- `int BPlusTreePage::GetMinSize() const`
  结点种类不同，对应的`Minsize`也不同。

```
{
  if(IsLeafPage()){
    return max_size_/2;
  }
  if(IsRootPage()){
    return IsLeafPage() ? 1 : 2;
  }
  return (max_size_ - 1) / 2 + 1;
}
```

#### 5.3.2.1.2 BPlusTreeLeafTree实现

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::Init(page_id_t page_id, page_id_t parent_id, int max_size)`
  用途：初始化`BPlusTreeLeafTree`。
  实现方法：赋予结点的`page_id`，父结点的`page_id`，以及最大尺寸`max_size`。

- `page_id_t B_PLUS_TREE_LEAF_PAGE_TYPE::GetNextPageId() const`
  用途：获取下一兄弟结点的id。

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::SetNextPageId(page_id_t next_page_id)`
  用途：设置下一兄弟结点的id。

- `int B_PLUS_TREE_LEAF_PAGE_TYPE::KeyIndex(const KeyType &key, const KeyComparator &comparator) const`

用途：找到`key`的索引位置
实现方法：利用`KeyComparator`找到第一个大于输入`key`的数组值，并返回其位于数组中位置走位索引。

- `KeyType B_PLUS_TREE_LEAF_PAGE_TYPE::KeyAt(int index) const`
  用途：获取数组指定位置的`key`

- `const MappingType &B_PLUS_TREE_LEAF_PAGE_TYPE::GetItem(int index)`
  用途：获取数组指定位置的`MappingType`,即键值对。

- `int B_PLUS_TREE_LEAF_PAGE_TYPE::Insert(const KeyType &key, const ValueType &value, const KeyComparator &comparator)`
  用途：插入新数据
  实现方法：利用`KeyIndex`找到对应位置并插入输入`key`和`value`，并更新`PageSize`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveHalfTo(BPlusTreeLeafPage *recipient)`
  用途：移动一般数据到目标页
  实现方法：调用`CopyNFrom`函数，并更新`PageSize`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::CopyNFrom(MappingType *items, int size)`
  用途：复制部分数据至此页种
  实现方法：内存移动，并更新`PageSize`
  `std::copy(items, items+size, array_+GetSize());`

- `bool B_PLUS_TREE_LEAF_PAGE_TYPE::Lookup(const KeyType &key, ValueType &value, const KeyComparator &comparator) const`
  用途：修改输入`key`的`value`

- `int B_PLUS_TREE_LEAF_PAGE_TYPE::RemoveAndDeleteRecord(const KeyType &key, const KeyComparator &comparator)`
  用途：删除一个数据
  实现方法：删除含key的数组元素，内存移动填充数组中的空缺，更新`size`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveAllTo(BPlusTreeLeafPage *recipient)`
  用途：转移所有数据至目标数组
  实现方法：同`MoveHalfTo`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveFirstToEndOf(BPlusTreeLeafPage *recipient)`
  用途：移动数组第一个元素至目标页数组的最后
  实现方法：调用`CopyLastFrom`并更新`size`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::CopyLastFrom(const MappingType &item)`
  用途：复制输入元素至数组最后

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveLastToFrontOf(BPlusTreeLeafPage *recipient)`
  用途：移动数组最后一个元素至目标页数组的第一个
  实现方法：调用`CopyFirstFrom`并更新`size`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::MoveAllToFrontOf(BPlusTreeLeafPage *recipient)`
  用途：移动数组所有元素至目标页数组的第一个
  实现方法：反复调用`CopyFirstFrom`并更新`size`

- `void B_PLUS_TREE_LEAF_PAGE_TYPE::CopyFirstFrom(const MappingType &item)`
  用途：复制输入的元素到数组的第一个元素并更新`size`

### 5.3.2.2 BPlusTreeInternalPage的实现

许多函数类似BPlusTreeLeafPage的实现，以下值选择差别较大的重要函数实现进行阐释。

- `void B_PLUS_TREE_INTERNAL_PAGE_TYPE::PopulateNewRoot(const ValueType &old_value, const KeyType &new_key, const ValueType &new_value)`
  用途：建立新的根结点
  实现方法：

```
array_[0].second = old_value;
array_[1]= {new_key,new_value};
SetSize(2);
```

- `int B_PLUS_TREE_INTERNAL_PAGE_TYPE::InsertNodeAfter(const ValueType &old_value, const KeyType &new_key,const ValueType &new_value) {`
  用途：在含指定键的数组元素后插入输入元素
  实现方法：利用`ValueIndex`找到指定位置移动数组再插入，并更新`size`
- `ValueType B_PLUS_TREE_INTERNAL_PAGE_TYPE::RemoveAndReturnOnlyChild()`
  用途：删出唯一的一个数组元素，并输出其`value`
  再Merge部分，BPlusTreeInternalPage相较BPlusTreeLeafPage不同的是，
  BPlusTreeInternalPage需要修改子结点的`ParentPageId`并设置为脏页。以下为例子。
- `void B_PLUS_TREE_INTERNAL_PAGE_TYPE::CopyLastFrom(const MappingType &pair, BufferPoolManager *buffer_pool_manager)`
  用途：复制输入元素到数组最后一个元素
  实现方法：

```
  array_[GetSize()] = pair;
  IncreaseSize(1);
  Page* child_page = buffer_pool_manager->FetchPage(pair.second);
  BPlusTreePage* tmp_node = reinterpret_cast<BPlusTreePage*>(child_page->GetData());
  tmp_node->SetParentPageId(this->GetPageId());
  buffer_pool_manager->UnpinPage(child_page->GetPageId(), true);
  return;
```

### 5.3.2.3 BPlusTree的实现

- `BPLUSTREE_TYPE::BPlusTree(index_id_t index_id, BufferPoolManager *buffer_pool_manager, const KeyComparator &comparator,int leaf_max_size, int internal_max_size):`

  `index_id_(index_id),buffer_pool_manager_(buffer_pool_manager),comparator_(comparator),leaf_max_size_(leaf_max_size),internal_max_size_(internal_max_size)`
  用途：构造函数
  实现方法：

```
  IndexRootsPage *root_index_page =
          reinterpret_cast<IndexRootsPage *>(buffer_pool_manager_-
>FetchPage(INDEX_ROOTS_PAGE_ID));
  if(root_index_page->GetRootId(index_id_, &(root_page_id_)) == false) {
    root_page_id_ = INVALID_PAGE_ID;
  }
  buffer_pool_manager_->UnpinPage(INDEX_ROOTS_PAGE_ID, false);
  buffer_pool_manager_->UnpinPage(root_page_id_, true);
```

- `bool BPLUSTREE_TYPE::GetValue(const KeyType &key, std::vector<ValueType> &result, Transaction *transaction)`
  用途：找到对应 `key` 的唯一 `value`，返回是否成功找到
  实现方法：利用 `FindLeafPage` 和 `B_PLUS_TREE_LEAF_PAGE_TYPE::Lookup` 函数找到该 `value`。
- `bool BPLUSTREE_TYPE::Insert(const KeyType &key, const ValueType &value, Transaction *transaction)`
  用途：插入新元素
  实现方法：

```
  //new tree
  if(IsEmpty()==true){
    StartNewTree(key,value);
    return true;
  }
  //insert node
  return InsertIntoLeaf(key, value);
```

- `void BPLUSTREE_TYPE::StartNewTree(const KeyType &key, const ValueType &value)`
  用途：建立树的第一个Page
  实现方法：利用 `BufferPoolManager::NewPage` 获得第一个页，再用初始化该叶结点并插入数据，设置脏页。
- `bool BPLUSTREE_TYPE::InsertIntoLeaf(const KeyType &key, const ValueType &value, Transaction *transaction)`
  用途：找到对应叶结点并插入数据，判断是否需要 `Split` 操作
  实现方法：同前方法插入，如果插入后 `size` 大于 `MaxSize`，则 `Split`。
- `template<typename N>`
  `N *BPLUSTREE_TYPE::Split(N *node)`
  用途：分裂结点并更新父结点。
  实现方法：通过 `MoveHalfTo`、`SetNextPageId` 等函数实现分裂，`InsertIntoParent` 插入父结点。叶结点和内部节点略有不同，在于更新子结点。

- `void BPLUSTREE_TYPE::InsertIntoParent(BPlusTreePage *old_node, const KeyType &key, BPlusTreePage *new_node,Transaction *transaction)`
  用途：插入数据至父结点相应位置
  实现方法：分父结点为根结点、内部节点两种情况。若为根结点，`PopulateNewRoot`并相应更新；若为你内部节点，则和`InsertIntoLeaf`相似，并相应更新。
- `void BPLUSTREE_TYPE::Remove(const KeyType &key, Transaction *transaction)`
  用途：删除指定元素
  实现方法：调用`RemoveAndDeleteRecord`删除，根据size判断是否需要要更新数据，再调用`CoalesceOrRedistribute`判断是调整还是删除页。
- `bool BPLUSTREE_TYPE::CoalesceOrRedistribute(N *node, Transaction *transaction)`
  用途：判断删除操作后是调整还是删除页
  实现方法：如果时根结点，`AdjustRoot`判断是否该删除。如果叶结点或内部节点，通过父结点找到其兄弟结点，如果两结点`size`的和大于`MaxSize`，则调整页，反之，则删除页。
- `bool BPLUSTREE_TYPE::Coalesce(N **neighbor_node, N **node,BPlusTreeInternalPage<KeyType, page_id_t, KeyComparator> **parent, int index,Transaction *transaction)`
  用途：合并至兄弟结点并删除页。
  实现方法：设置更新`next_page_id`等数据及父结点后调用`MoveAllTo`函数完成合并，再删除页。根据目标页再父结点子结点数组中位置的不同，合并的方法会有些许差别。
- `void BPLUSTREE_TYPE::Redistribute(N *neighbor_node, N *node, int index)`
  用途：重新分配两个结点的元素。
  实现方法：更具结点类型，以及结点再父结点的子结点数组中位置的不同，分配方法有所差别。叶结点位于第一个则调用`MoveFirstToEndOf`，其他则调用`MoveLastToFrontOf`；内部结点位于第一个则还需要改变父结点数组的第二个元素。
- `bool BPLUSTREE_TYPE::AdjustRoot(BPlusTreePage *old_root_node)`
  用途：更新根结点并返回是否该删除
  实现方法：如果时叶结点，删；如果时内部结点，只有一个子结点时，删除并以该子结点为新的根结点。
- `INDEXITERATOR_TYPE BPLUSTREE_TYPE::Begin()`
  用途：找到最左的叶结点并构建索引迭代器
  实现方法：利用`FindLeafPage(KeyType(),true)`找到最左的叶结点，再`return INDEXITERATOR_TYPE(first_node,0,buffer_pool_manager_);`
- `INDEXITERATOR_TYPE BPLUSTREE_TYPE::Begin(const KeyType &key)`
  用途：找到目标叶结点并构建索引迭代器
  实现方法：利用`FindLeafPage(key)`找到叶结点，再`return INDEXITERATOR_TYPE(leaf_page,0,buffer_pool_manager_);`
- `INDEXITERATOR_TYPE BPLUSTREE_TYPE::End()`
  用途：找到最后一个叶结点并构建索引迭代器
  实现方法：利用`FindLeafPage(keyKeyType(),false,tru)`找到叶结点，再`return INDEXITERATOR_TYPE(leaf_page,0,buffer_pool_manager_);`
- `Page *BPLUSTREE_TYPE::FindLeafPage(const KeyType &key, bool leftMost,bool rightMost)`
  用途：找到目标键的叶结点，或找到最左、最右的叶结点
  实现方法：利用`BufferPoolManager::FetchPage`获取当前页，再反复调用`Lookup`获得目标`LeafPage`
- `void BPLUSTREE_TYPE::UpdateRootPageId(int insert_record)`
  用途：在标题页中更新/插入根页 ID
  实现方法：`insert_record`为0，调用`IndexRootsPage::Update`更新；`insert_record`为1，调用`IndexRootsPage::Insert`插入。

### 5.3.2.4 索引迭代器的实现

- `INDEX_TEMPLATE_ARGUMENTS INDEXITERATOR_TYPE::IndexIterator(BPlusTreeLeafPage<KeyType, ValueType, KeyComparator>  *leaf_page, int index_, BufferPoolManager *buffer_pool_manager)`
  用途：构造函数
- `INDEX_TEMPLATE_ARGUMENTS INDEXITERATOR_TYPE::~IndexIterator()`
  用途：析构函数，设置脏页
  实现方法：
- `INDEX_TEMPLATE_ARGUMENTS const MappingType &INDEXITERATOR_TYPE::operator*()`
  用途：重载`*`，获取叶结点
  实现方法：`return leaf_page->GetItem(index_);`
- `INDEXITERATOR_TYPE &INDEXITERATOR_TYPE::operator++()`
  用途：重载`++`，移动到下一元素
  实现方法：如果是页最后，`FetchPage`转到下一页

```
if (index_==leaf_page->GetSize()-1){//页的最后一个
    page_id_t next_page_id = leaf_page->GetNextPageId();
    if (next_page_id!=INVALID_PAGE_ID){//还有下一页
      index_ = 0;
      //leaf_page变成下一页
      Page* page = buffer_pool_manager->FetchPage(next_page_id);
      leaf_page = reinterpret_cast<LeafPage *>(page->GetData());
    }
    else{
      index_++;
    }
  }
  else{
    index_++;
  }
  return *this;
```

- `INEXITERATOR_TYPE::operator==(const IndexIterator &itr) const`
  用途：重载`==`
  实现方法：`page_id`和`_index`都相等
- `bool INDEXITERATOR_TYPE::operator!=(const IndexIterator &itr) const`
  用途：重载`!=`
  实现方法：`page_id`和`_index`不都相等

## 5.3.3单元测试结果

B+树测试

```
/mnt/d/Programs/database/minisql/build/test/b_plus_tree_test
chy@LAPTOP-G9LNOE8J:/mnt/d/Programs/database/minisql/build/test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (19 ms)
[----------] 1 test from BPlusTreeTests (19 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (20 ms total)
[  PASSED  ] 1 test.
```

B+树索引测试

```
chy@LAPTOP-G9LNOE8J:/mnt/d/Programs/database/minisql/build/test$ /m
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (8 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (7 ms)
[----------] 2 tests from BPlusTreeTests (16 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (16 ms total)
[  PASSED  ] 2 tests.
```

索引迭代器测试

```
chy@LAPTOP-G9LNOE8J:/mnt/d/Programs/database/minisql/build/test
ams/database/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 (12 ms)
[----------] 1 test from BPlusTreeTests (12 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (12 ms total)
[  PASSED  ] 1 test.
```

# 5.4 Catalog Manager

## 5.4.1 模块功能概述

Catalog Manager 负责管理和数据库的表和索引相关信息，包括数据库中所有表的字段，元组信息，以及索引的所属表，所建立的字段。`catalogmeta`在数据库首次创建时被初始化，其对应的逻辑页页号为`1`，被加载到内存时，反序列化为`catalogmeta`，随后根据其信息从数据库文件中加载所有的表和索引信息，构建`TableInfo`和`IndexInfo`信息置于内存中。此外`catalogmeta`还为上层模块提供创建/删除表，创建/删除索引等操作的接口。在内存中对表和索引进行操作时需要对`catalogmeta`进行修改，修改后的`catalogmeta`由`bufferpoolmanager`写回磁盘中。

## 5.4.2 各函数具体实现

### 5.4.2.1 序列化与反序列化

- `CatalogMeta::SerializeTo(*buf)`
- `CatalogMeta::GetSerializedSize()`
- `CatalogMeta::DeserializeFrom(*buf, *heap)`
- `IndexMetadata::SerializeTo(*buf)`
- `IndexMetadata::GetSerializedSize()`
- `IndexMetadata::DeserializeFrom(*buf, *&index_meta, *heap)`
- `TableMetadata::SerializeTo(*buf)`
- `TableMetadata::GetSerializedSize()`
- `TableMetadata::DeserializeFrom(*buf, *&table_meta, *heap)`
- `IndexInfo::Init(*index_meta_data, *table_info, *buffer_pool_manager)`

与5.2.2.1节类似，`SerializeTo`，`GetSerializedSize`，`DeserializeFrom`函数功能相同，只需要将对应类的成员变量转化为字节流以及从字节流恢复即可。

### 5.4.2.2 表和索引的管理（Catalog Manager类的实现）

- `dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema,Transaction *txn, TableInfo *&table_info)`

  用途：创建一个表。

  实现方式：首先获取新建表的`id`并为其元信息分配一页；再调用`TableMetadata::Create`和`TableHeap::Create`创建该表的元信息 和堆表；然后创建`table_info`并初始化，最后更新`catalogmeta`中表名、页号等信息的散列表即可，新表的信息写入参数表中的 `table_info`，返回值为是否创建成功。

- `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)`

  用途：在给定列上创建索引。

  实现方式：`index_keys`为所需创建索引的列的名称，由此可得到这些列的序号，作为`arrtibute_key`创建相对应的`indexmeta`。然后创建`IndexInfo`并初始化，更新`catalogmeta`中index的相关信息即可。

- `CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager, LogManager *log_manager, bool init)`

  用途：构造函数

  实现方式：如果`init`为真，则新建一个`catalogmanager`，否则获取逻辑页0数据，反序列化得到`catalogmanager`并根据其中的页号与

  表、索引的对应关系，从相应页中反序列化得到表和索引的详细信息，重新构建相关哈希表。

- `dberr_t CatalogManager::DropTable(const string &table_name)`

  用途：删除表

  实现方式：从表名的哈希表中删除该表，并释放对应数据页。

- `dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name)`

  用途：删除索引

  实现方式：从索引名的哈希表中删除该索引，并释放对应数据页。

- `dberr_t CatalogManager::GetTable(const table_id_t table_id, TableInfo *&table_info)`

  用途：根据表名获取表信息

  实现方式：查询哈希表。

- `dberr_t CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const`

  用途：获取一个表的所有信息。

  实现方式：查询哈希表。

- `dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string &index_name, IndexInfo *&index_info) const`

  用途：根据表名和索引名获取索引信息。

  实现方式：查询哈希表。

- `dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables)`

  用途：获取所有表信息。

  实现方式：通过迭代器遍历哈希表。

- `void IndexInfo::Init(IndexMetadata *meta_data, TableInfo *table_info, BufferPoolManager *buffer_pool_manager)`

  用途：初始化`IndexInfo`。

  实现方式：首先初始化`meta_data`和`table_info`，再调用`Schema::ShallowCopySchema`创建自身`key_schema_`，最后调用`IndexInfo::CreateIndex`创建索引。

- `Index *IndexInfo::CreateIndex(BufferPoolManager *buffer_pool_manager)`

  用途：创建索引。

  实现方式：调用索引的构造函数，从`MemHeap`中分配内存构造索引。

#### 5.4.2.3 CRC校验算法(bonus)

为了确保序列化和反序列化时的正确性，我们引入了CRC校验算法验证数据的正确性。我们在每次序列化时对写入的数据调用CRC校验算法计算出来一个值，一并写入到数据页中。在反序列化时根据读取出来的数据重新利用CRC算法计算，并与从数据页中提取出来的值进行对比，如果一致则可以证明数据正确。

CRC算法的核心内容如下所示：

```
uint16_t TableMetadata::crc16(char *addr, uint32_t num, uint16_t crc) const
```

```
{
  int i;
  for (; num > 0; num--)              /* Step through bytes in memory */
  {
    crc = crc ^ (*addr++ << 8);     /* Fetch byte from memory, XOR into CRC top
byte*/
    for (i = 0; i < 8; i++)             /* Prepare to rotate 8 bits */
    {
      if (crc & 0x8000)            /* b15 is set... */
        crc = (crc << 1) ^ POLY;    /* rotate and XOR with polynomic */
      else                          /* b15 is clear... */
        crc <<= 1;                  /* just rotate */
    }                               /* Loop for 8 bits */
    crc &= 0xFFFF;                  /* Ensure CRC remains 16-bit value */
  }                                 /* Loop until num=0 */
  return(crc);                      /* Return updated CRC */
}
```

该方法的正确性也可以通过下面的单元测试结果得到验证。

### 5.4.3 单元测试结果

发现可以通过所有的测试样例。

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTestOperation
[       OK ] CatalogTest.CatalogTableTestOperation (29 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (23 ms)
[----------] 4 tests from CatalogTest (53 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (54 ms total)
[  PASSED  ] 4 tests.
```

## 5.5 SQL Executor

### 5.5.1 模块功能概述

Executor（执行器）的主要功能是根据解释器（Parser）生成的语法树，通过Catalog Manager 提供的信息生成执行计划，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，最后通过执行上下文`ExecuteContext`。在本节中，我们实现了`ExecuteEngine`中所有的执行函数，它们被声明为`private`类型的成员，即所有的执行过程对上层模块是隐藏的，上层模块只需要调用`ExecuteEngine::execute()`并传入语法树结点即可无感知地获取到执行结果。

### 5.5.2 各函数具体实现

- `ExecuteEngine::ExecuteEngine()`

  用法：构造函数通过读取`/mnt/d/database_name.txt`中记录的已存在数据库名称，然后从对应的文件中读取数据库实例，构建`DBStorageEngine`。

- `ExecuteEngine::~ExecuteEngine()`

  用法：析构函数通过将已创建的数据库名称写入到`/mnt/d/database_name.txt`中，然后将已创建的数据库实例写到对应的文件中。

- `ExecuteEngine::ExecuteCreateDatabase(*ast, *context)`

  用法：创建一个新的`DBStorageEngine`，并将数据库名称写入到`/mnt/d/database_name.txt`中。

- `ExecuteEngine::ExecuteDropDatabase(*ast, *context)`

  用法：调用`DBStorageEngine`的析构函数，并删除`/mnt/d/database_name.txt`中的的数据库名称。

- `ExecuteEngine::ExecuteShowDatabases(*ast, *context)`

  用法：依次输出内存中存储的数据库名称。

- `ExecuteEngine::ExecuteUseDatabase(*ast, *context)`

  用法：将当前使用的数据库`current_database`设置为选择的数据库名称。

- `ExecuteEngine::ExecuteShowTables(*ast, *context)`

  用法：只有在`current_database`设置之后输出其对应的表名称，否则输出`You haven't chosen a database!`。

- `ExecuteEngine::ExecuteCreateTable(*ast, *context)`

  用法：根据语法树遍历，根据节点中记录的属性名称、值等条件，通过`catalog manager`中的`create table`方法建立表，同时会记录`unique`和`primary key`的值。

- `ExecuteEngine::ExecuteDropTable(*ast, *context)`

  用法：根据`catalog manager`中的`drop table`方法删除数据表。

- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)`

  用法：显示当前所有的索引名称。

  实现方式：遍历catalog中的所有表，获取每个表中的所有索引。

- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)`

  用法：在指定表的指定列上建立索引。

  实现方式：首先检查建立索引的属性值是否唯一，然后调用`catalog manager`中的`creat index`方法建立索引，并插入所有列。

- `ExecuteEngine::ExecuteDropIndex(*ast, *context)`

  用法：删除指定索引。

  实现方式：调用`catalog manager`中的`drop index`方法。

- `ExecuteEngine::ExecuteSelect(*ast, *context)`

用法：解析语法树，如果有`where`的条件判断则调用`WhereClause`函数，保留下来满足条件的记录对应的`rowid`，找到对应的`row`之后通过`select`选择的需要输出的列将记录输出。

- `ExecuteEngine::ExecuteInsert(*ast, *context)`

  用法：解析语法树，根据节点中记录的属性名称、值等条件，通过`catalog manager`中的`InsertTuple`方法建立记录。同时需要检测`unique`和`primary key`属性是否有重复，如果重复会产生报错提示。

- `ExecuteEngine::ExecuteDelete(*ast, *context)`

  用法：类似`ExecuteSelect`的过程，`WhereClause`函数判断结束后，调用`catalog manager`中的`DeleteTuple`方法删除对应的记录。

- `ExecuteEngine::ExecuteUpdate(*ast, *context)`
- 用法：类似`ExecuteSelect`的过程，`WhereClause`函数判断结束后，通过记录需要更新的列，遍历所有需要改变的记录，然后重建一条记录，最后利用`catalog manager`中的`UpdateTuple`函数更新数据。
-
- `ExecuteEngine::ExecuteExecfile(*ast, *context)`

  用法：在命令行中需要输入文件所在的路径，然后取得脚本中的每一条SQL命令，调用`parser`层解析后调用`Execute`执行。

- `ExecuteEngine::ExecuteQuit(*ast, *context)`

  用法：框架中已实现，可以退出程序。

- `dberr_t WhereClause(set<int64_t> &record_old, set<int64_t> &record_new, pSyntaxNode ast, ExecuteContext *context, Schema* schema, TableHeap* table_heap);`

  用法：`WhereClause`在遇到`and`和`or`子句的时候会递归调用，`record_old`中存储上一级满足条件的`rowid`，`record_new`返回本级满足条件的`rowid`。

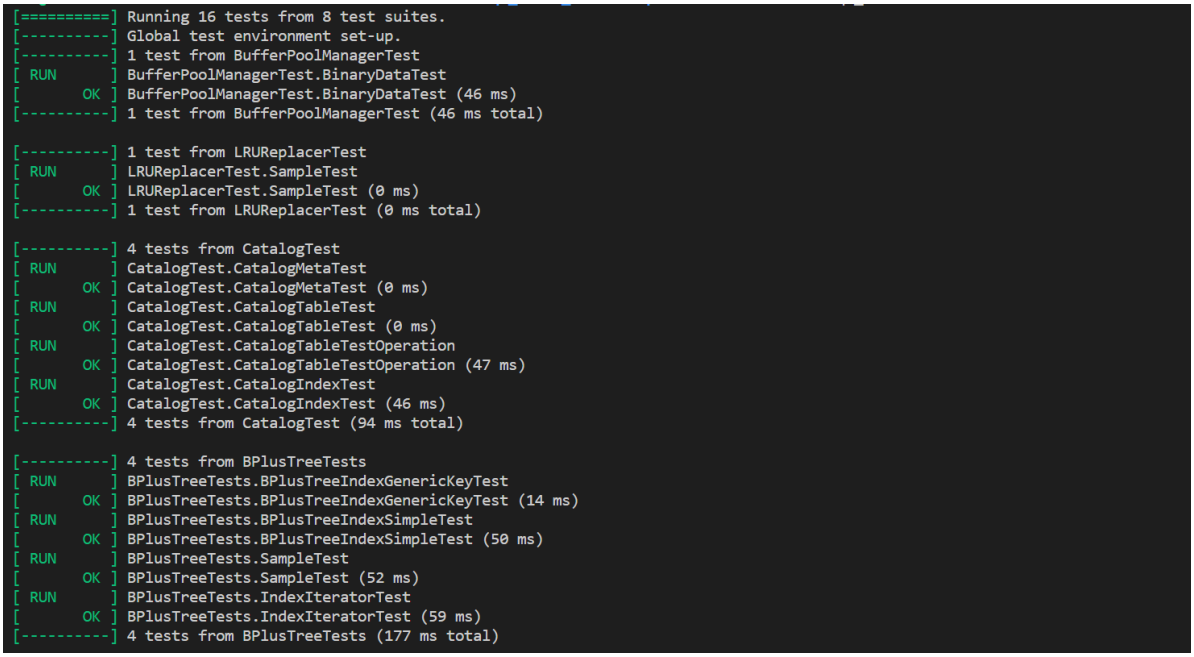
- `int CompareValue(TableInfo* table_info, std::string column_name, std::string value, RowId row_id);`

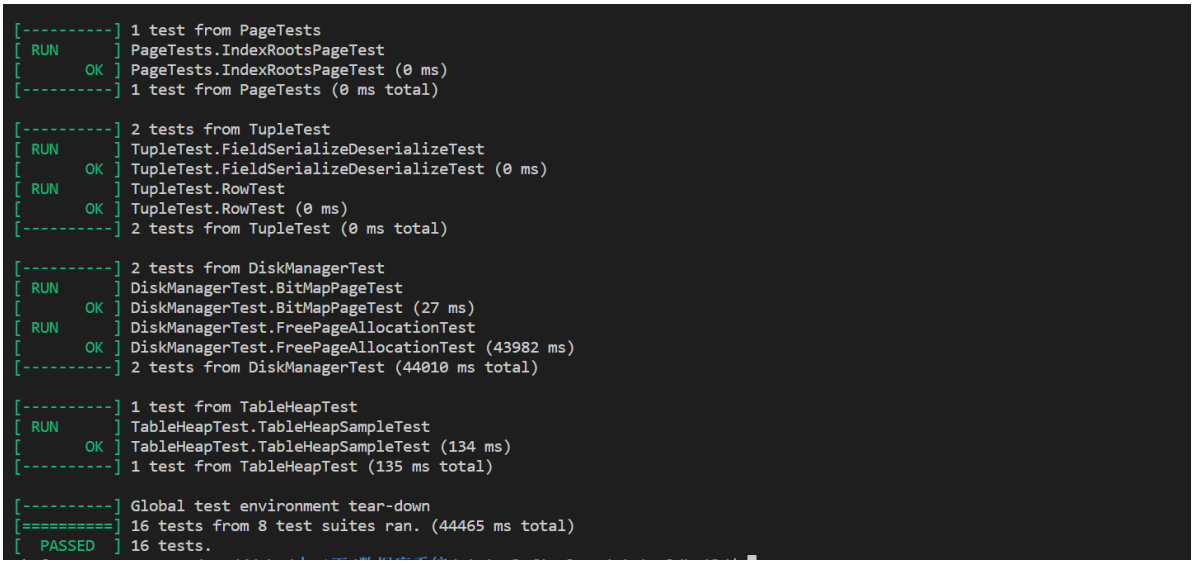
  用法：获取参数中table在`column_name`列和`row_id`行的值，获取其属性并与`value`的相应类型值进行比较，返回相等、大于等不同的比较结果。

# 六、系统功能测试

## 6.1 运行所有单元测试文件

运行minisql_test文件，通过了所有16个测试项：




## 6.2 sql语句实地测试

依次运行test_command.txt包含有所有功能的测试。其中account001.txt中包含大约2000条insert语句。

### 6.2.1 数据库创建和删除

会话过程如下所示，我们创建了3个数据库，可以通过show databases语句展示，删除数据库后也可以看到数据库名称消失。同时要想使后续操作有效，必须用use语句选择一个特定的database。

```
minisql > create database db0;
Query OK (0.005302 sec)

minisql > create database db1;
Query OK (0.004764 sec)

minisql > create database db2;
Query OK (0.009365 sec)

minisql > show databases;
+-------------+
| Database    |
+-------------+
| db2         |
| db1         |
| db0         |
+-------------+
3 rows in set (0.00011 sec)

minisql > use db0;
Database changed

minisql > drop db1;
Minisql parse error at line 1, col 8, message: syntax error
syntax error
minisql > drop database db1;
Query OK (0.000974 sec)

minisql > show databases;
+-------------+
| Database    |
+-------------+
| db2         |
| db0         |
+-------------+
2 rows in set (9.2e-05 sec)
```

### 6.2.2 数据表插入和删除

创建了两个数据表account和student，可以通过show tables来显示，删除后也能看到数据表名称消失。

```
minisql > create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
Query OK, 0 rows affected (0.001764 sec)

minisql > create table student(
  sno char(8),
  sage int,
  sab float unique,
  primary key (sno, sab)
);
Query OK, 0 rows affected (0.001326 sec)

minisql > show tables;
+--------------------+
| Tables_in_db0      |
+--------------------+
| student            |
| account            |
+--------------------+
2 rows in set (0.000153 sec)

minisql > drop table student;
Query OK, 0 rows affected (0.000419sec)

minisql > show tables;
+--------------------+
| Tables_in_db0      |
+--------------------+
| account            |
+--------------------+
1 rows in set (7.9e-05 sec)
```

### 6.2.3 插入删除更新查找数据

在下面的会话过程中首先插入两条正常数据，第三条记录primary key冲突，第四条记录unique属性冲突，都会产生duplicate的错误信息。之后也可以通过select语句查看已插入的数据，select也支持部分选择属性。后面删除一条记录并更新一条记录，都可以用select语句查看正确性。

```
minisql > insert into account values(2001, "liu", 1.22);
Query OK, 1 row affected (0.000238sec)

minisql > insert into account values(2002, "li", 2.13);
Query OK, 1 row affected (0.000225sec)

minisql > insert into account values(2001, "wang", 1.33);
Duplicate 
minisql > insert into account values(2003, "liu", 2.33);
Duplicate 
minisql > select * from account;
| 2001 | liu              | 1.22 |
| 2002 | li               | 2.13 |
Query OK, 2 rows affected (0.000498sec)

minisql > delete from account where id = 2002;
Query OK, 1 rows affected (0.000181sec)

minisql > select id from account;
| 2001 |
Query OK, 1 rows affected (0.000537sec)

minisql > update account set id = 2004 where id = 2001 and balance = 1.22;
Query OK, 1 row affected (0.000164sec)

minisql > select * from account;
| 2004 | liu              | 1.22 |
Query OK, 1 rows affected (0.000179sec)
```

### 6.2.4 大批量插入数据

```
minisql > execfile "/mnt/d/account001.txt";
```

使用select语句全表查询，发现数据插入成功

```
| 12501979 | name1979         | 271.14 |
| 12501980 | name1980         | 106.06 |
| 12501981 | name1981         | 563  |
| 12501982 | name1982         | 262.57 |
| 12501983 | name1983         | 428.45 |
| 12501984 | name1984         | 258.57 |
| 12501985 | name1985         | 929.81 |
| 12501986 | name1986         | 885.99 |
| 12501987 | name1987         | 371.87 |
| 12501988 | name1988         | 246.55 |
| 12501989 | name1989         | 573.24 |
| 12501990 | name1990         | 831.35 |
| 12501991 | name1991         | 172.19 |
| 12501992 | name1992         | 707.74 |
| 12501993 | name1993         | 3.35 |
| 12501994 | name1994         | 828.38 |
| 12501995 | name1995         | 834.33 |
| 12501996 | name1996         | 133.24 |
| 12501997 | name1997         | 714.14 |
| 12501998 | name1998         | 877.37 |
| 12501999 | name1999         | 159.87 |
| 12502000 | name2000         | 452.77 |
| 12502001 | name2001         | 8.13 |
| 12502002 | name2002         | 412.83 |
Query OK, 2003 rows affected (0.15436sec)
```

### 6.2.5 索引的创建删除和性能测试

首先在无索引的时候进行了一次单点查询，之后创建索引，可以用show indexes查看。再次查询，发现用时明显变少。之后删除索引，再次查看，发现用时跟无索引时一样，于是可以证明索引的有效性，

```
minisql > select * from account where id = 12502000;
| 12502000 | name2000         | 452.77 |
Query OK, 1 rows affected (0.046167sec)

minisql > create index idx1 on account(id);
minisql > show indexes;
idx1
minisql > select * from account where id = 12502000;
| 12502000 | name2000         | 452.77 |
Query OK, 1 rows affected (0.0227195sec)

minisql > drop index idx1;
minisql > show indexes;
minisql > select * from account where id = 12502000;
| 12502000 | name2000         | 452.77 |
Query OK, 1 rows affected (0.046223sec)
```

# 七、开发日程

5.11 完成disk and buffer manager部分

5.14 完成record_manager部分

5.15 完成catalog manager序列化和反序列化部分

5.27 完成execute_engine的部分内容

5.30 完成catalog_manager大部分内容，开始在execute_engine中调试整合各模块

6.7 完成B+树部分

6.9 完成execute_engine部分，并完成部分bonus和优化

6.10 验收

# 八、小组成员分工

omitted.