实验六 MiniSQL的实现

1. 实验目的

1. 使用C++语言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。

2. 实验平台

Linux 或 WSL-Ubuntu。

3. 实验要求

1. 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
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。
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。
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。
5. 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。

4. 实验模块

4.1 Disk and Buffer Pool Manager

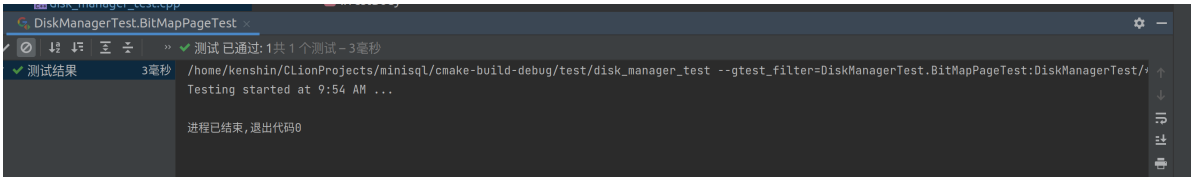
4.1.1 概述

实验1主要包括了四个部分，分别是：

- 位图的实现
- 磁盘管理的实现
- 缓冲池替换策略的实现
  - LRU 替换策略
  - Clock 替换策略
- 缓冲池的实现

4.1.2 Bitmap的实现

每一个位图页由两部分组成，第一部分是元信息，包含已分配的页`page_allocated`和下一个空闲页`next_free_page_`，第二部分是一串 bit 来标记页的分配与否，1代表分配，0代表不分配。实现上采用`std::bitset<>`来管理数据页。测试样例结果如下：



📌 **Notice**

此处本来打算更新元信息 next_free_page，这样在后续磁盘管理从位图中分配数据页时不用逐个查找空闲页($O(N)$)，而是可以直接返回空闲页。我计划找到一串 bit 中的第一个0，算法如下：

把一串 bit 记作 data，这里以1110 1011为例：

1. ~data: 0001 0100，取反后问题转化为找到第一个1
2. data - 1: 0001 0011
3. ~(data - 1): 1110 1100
4. (data - 1)&(~(data - 1)): 0001 0000

然后可以将其以整数形式取对数得到是第五个位置上出现 0 ，但我们的问题中的 bit 流有4032位，不可能直接转化为整数，并且 `std::bitset` 并没有相应的无符号加法和减法，因此这个方法也就作罢。

4.1.3 磁盘数据页管理

这一部分首先要明白我们的做法是对于整个磁盘，我们有一个 `PAGE_SIZE` 大小的元信息页来管理磁盘数据信息，包含了已分配的位图页和已分配的页的数量，大小为两个 `uint_32t` ，剩下的空间用来记录每一个位图页已分配的页的数量，其次是前面的位图页用于管理一段连续的数据页，依次类推。

从测试代码中我发现了 `DiskFileMetaPage` 这个专门用于管理磁盘元信息的类，于是在从磁盘中分配一页时，我的做法是：

1. 首先从 `extent_nums` 中获得第一个没有分配满的位图页。
    - 如果没有找到而且当前位图页数量已经达到最大可分配数量，返回 `INVALID_PAGE_ID`。
    - 如果没有找到但位图页小于最大数量，分配新的一页，更新分区数量和已分配页数量等元信息，然后调用 `WritePhysicalPage` 写回磁盘并返回分配的页号。
2. 找到没有分配满的位图页，找到该位图页中第一个空闲页并分配，更新元信息，调用 `WritePhysicalPage` 写回磁盘并返回分配的页号（这里就是线性查找可以优化的地方，但是失败了）。

在释放一页时，做法则基本相反：

找到 `logical_page_id` 所在的分区，调用 `ReadPhysicalPage` 读取当前分区的数据，更新元数据，释放对应数据页，再调用 `WritePhysicalPage` 写回磁盘。

4.1.4 缓冲区替换策略

LRU 替换策略

LRU 替换策略是最近最少访问者被替换，我的实现方式是用一个链表记录 `frame_id` ，用哈希表记录 `frame_id` 在链表中的位置。效果类似于一个队列，每次 `Unpin` 时，从链表末尾加入，每次寻找 Victim 时，则从链表头取出。

Bonus: Clock 替换策略

Clock 替换策略是一种较为开销较低的替换策略，通过给予每个数据页"第二次机会"来实现页面的替换。对每个数据页来说，一共设置两种状态， `ACCESSED` 和 `UNUSED` ，对于初次加入 Clock Replacer 的页面，设置为 `ACCESSED` ，在寻找待替换页面时，先遍历一遍 Replacer ，将 `ACCESSED` 设置为 `UNUSED` ，同时将待替换的页面设置为第一个遇到的 `UNUSED` 页面。

4.1.5 缓冲池管理

这个部分集成了前面全部模块，基于代码中注释的提示，以 `FetchPage` 为例，我的做法是：

1. 在 `page_table_` 中搜寻页号。
2. 若存在则直接返回。
3. 若不存在则先从 `free_list_` 中查找，在 `free_list_` 中仍不存在的（缓冲池已满）再使用 LRU 策略进行替换，若没有找到可替换对象，返回空值。
4. 从 `page_table_` 中删除待替换帧（步骤3中找到的），插入新的帧，调用 `ReadPage` 返回该帧的数据信息。

注意该模块的 `FlushPage` 函数的作用是将数据进行磁盘中的回写来删除其 `IsDirty` 标志。

总的来说，该模块的主要工作是实现缓冲池的I/O操作。

4.2 Record Manager

4.2.1 概述

实验2负责记录的管理，这里的记录也即我们常说的表，分为：

- 行（Row）：也即元组。
- 列（Column）：定义和表示表中的一个字段。
- 模式（Schema）：表示数据表或者是索引的结构，由一个或多个的Column构成。
- 域（Field）：一条记录中某一字段的数据信息，如数据类型，是否为空，值等。

4.2.2 序列化和反序列化

为了持久化数据，我们需要对 Row，Column，Schema 和 Field 进行序列化处理，以便它能够存到磁盘中。此外引入魔数做为简单的检验数据正确性的手段。

序列化也即在上游像“水流”一样将数据按字节存到一块连续的内存区域（buffer）中，反序列化即在“下游”从 buffer 中按顺序取出存的东西再重新构造出相应的对象。

Column

```
class Column {
 private:
  static constexpr uint32_t COLUMN_MAGIC_NUM = 210928;
  std::string name_;
  TypeId type_;
  uint32_t len_{0};
  uint32_t table_ind_{0};
  bool nullable_{false};
  bool unique_{false};
};
```

`Column` 的数据成员如上图所示。除了 `string` 类型的对象，我们都可以使用`MACH_WRITE_TO(Type, buf, DATA)` 来进行序列化，对于 `string` 类型的对象，首先要写入字符串有多少个字节，再使用 `memcpy` 进行序列化。

反序列化时，使用 `MACH_READ_FROM(Type, buf)` 逐个取出。要注意根据 `type_` 的类型在重新构造 `Column` 对象时传入不同的参数。

Schema

```
class Schema{
private:
  static constexpr uint32_t SCHEMA_MAGIC_NUM = 200715;
  std::vector<Column *> columns_;
};
```

`Schema` 的数据成员如上图所示。对于 `vector` 这种容器，我们的做法是先写入容器的大小，再写入容器内容，此处可以调用 `Column` 的序列化。

反序列化构造 `Schema` 对象即可。

Row

```
class Row{
 private:
  RowId rid_{};
  std::vector<Field *> fields_;
  MemHeap *heap_{nullptr};

  uint32_t fields_nums{0};
  uint32_t null_nums{0};
};
```

`Row` 需要额外记录空域的数量，因此在这里我添加了一个 `null_nums` 用于记录，在其后写入了空域所在的下标。事实上这可以用空位图来表示，节省空间和时间。

序列化只需要调用 `Field` 提供的序列化即可。反序列化时要构造 `Field` 对象。

4.2.3 堆表的实现

堆表的数据结构和教材上的基本一致，由表头、空闲空间和已经插入数据三部分组成。在这部分我们需要完成的函数有：

- `bool InsertTuple(Row &row, Transaction *txn)`
- `bool UpdateTuple(const Row &row, const RowId &rid, Transaction *txn)`
- `void ApplyDelete(const RowId &rid, Transaction *txn)`

`InsertTuple`

插入元组采用了 First Fit 的策略，取出堆表的第一页尝试插入，如果已经不能插入（满了），那我们从 bufferpool 中取出新的一页，如果不能分配新的页，则插入返回失败，否则尝试插入，如此循环直到成功插入。

`UpdateTuple`

`UpdateTuple` 需要区分更新失败的类型，而不仅仅是返回成功与失败。但是如果希望对当前框架做最小改动，我们应该修改 `src/storage/table_page.cpp` 中的 `UpdateTuple` 的返回值，然后在本节的 `UpdateTuple` 中通过返回值区分失败原因，执行相应的操作。

于是在 `table_page.cpp` 中，我加入了如下的返回状态：

```
class TablePage : public Page {
 public:
  enum class RetState { ILLEGAL_CALL,
INSUFFICIENT_TABLE_PAGE,DOUBLE_DELETE,SUCCESS };
};
```

- `ILLEGAL_CALL` 代表非法调用，在传入 `slot_num` 无效返回。
- `INSUFFICIENT_TABLE_PAGE` 代表当前页的空间不足以放下一个元组。
- `DOUBLE_DELETE` 代表待更新元组已被删除。
- `SUCCESS` 代表更新成功。

在本节的 `UpdateTuple` 中，根据返回状态我们进行如下操作：

- `ILLEGAL_CALL` 或 `DOUBLE_DELETE`，返回 `false`
- `INSUFFICIENT_TABLE_PAGE`：删除旧元组，将其插入到新的一页
- `SUCCESS`：标记页为脏，返回 `true`

`ApplyDelete`

调用 `ApplyDelete`，标记页为脏即可。

4.2.4 堆表迭代器

堆表迭代器主要作用是给上层模块（第五部分）提供接口，实现表的遍历。

重载的 `==` `!=` `*` `->` 运算符功能不再赘述，主要叙述前置 `++` 运算符的实现，后置 `++` 拷贝一份原指针再调用前置 `++` 即可。

我们调用 `GetNextTupleRid(args)` 得到下一个元组，如果成功，那么我们移动到下一条记录再返回即可，否则意味着下一个元组不在当前页，我们调用 `GetNextPageId()` 从 bufferpool 中取出下一页并获取第一个元组，移动到这条记录再返回。

4.2.5 堆表的 `Begin` 和 `End`

对于 `Begin` 我们尝试取出每一页的第一条记录，如果成功取出，则成功找到堆表的第一条记录，否则循环直到找到，若最后找不到，将返回 `INVAILD_ROWID` 构造的迭代器。

对于 `End`，直接返回 `INVAILD_ROWID` 构造的迭代器。

4.3 Index Manager

4.3.1 概述

1. B+树的构成以page为基本单元，其数据都保存在 `page` 中，根据节点数据形式的不同，分为 `Internal Page` 和 `Leaf Page`，可以通过类型转换将 `page` 转化为 `Internal Page` 或 `Leaf Page`。在操作过程中，B+树通过不断地 `Fetchpage` 从 `buffer pool manager` 处获取 `Page`，同 `UnpinPage` 并标记为脏以更新磁盘处的B+树数据。
2. 迭代器的存在是为了实现对于所有保存的数据进行一次线性的遍历，在B+树的体系结构中，实现的迭代器能够获取B+树索引的头节点以及对于单个迭代器节点自增的操作，从而为上层程序获取单个元组的信息提供可能。

4.3.2 B+树的实现

通过以下方式新建一个B+树，同时考虑到原本磁盘处无数据以及已经有数据从磁盘中读取出来的情况。

```
INDEX_TEMPLATE_ARGUMENTS
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) {
  auto root_page = reinterpret_cast<IndexRootsPage *>(buffer_pool_manager-
>FetchPage(INDEX_ROOTS_PAGE_ID));

  if (!root_page->GetRootId(index_id, &this->root_page_id_)) {
    this->root_page_id_ = INVALID_PAGE_ID;
  }
  buffer_pool_manager->UnpinPage(INDEX_ROOTS_PAGE_ID, true);
  buffer_pool_manager_->UnpinPage(root_page_id_, true);
}
```

同时，以下函数能够向上对接索引的接口函数，保证整个索引体系的可扩展性。

```
  // Returns true if this B+ tree has no keys and values.
  bool IsEmpty() const;

  // Insert a key-value pair into this B+ tree.
  bool Insert(const KeyType &key, const ValueType &value, Transaction
*transaction = nullptr);

  // Remove a key and its value from this B+ tree.
  void Remove(const KeyType &key, Transaction *transaction = nullptr);

  // return the value associated with a given key
  bool GetValue(const KeyType &key, std::vector<ValueType> &result, Transaction
*transaction = nullptr);
```

还有如以下函数等，负责B+树结构体系的维护和调整，单个函数的具体功能较为简单，但是函数数量很多，在此不多赘述。

```
// Split and Merge utility methods
  void MoveHalfTo(BPlusTreeLeafPage *recipient);

  void MoveAllTo(BPlusTreeLeafPage *recipient);

  void MoveFirstToEndOf(BPlusTreeLeafPage *recipient);

  void MoveLastToFrontOf(BPlusTreeLeafPage *recipient);
```

4.3.3 迭代器的实现

这部分的代码较为简单，只需要实现自增和等价比较的几个算符即可，其自身的序号通过

`index_` 这个私有变量进行描述，自增时需要考虑跨 Page 以及到达末尾的情况。

```
INDEX_TEMPLATE_ARGUMENTS const MappingType &INDEXITERATOR_TYPE::operator*()
{
  // ASSERT(false, "Not implemented yet.");
  return leaf_page_->GetItem(index_);
}

INDEX_TEMPLATE_ARGUMENTS INDEXITERATOR_TYPE &INDEXITERATOR_TYPE::operator++()
{
  if (index_ == leaf_page_->GetSize() - 1
      && leaf_page_->GetNextPageId() != INVALID_PAGE_ID)
  {
    Page* next_page = buffer_pool_manager_->FetchPage(leaf_page_-
>GetNextPageId());
    buffer_pool_manager_->UnpinPage(page_->GetPageId(), false);
    page_ = next_page;
    leaf_page_ = reinterpret_cast<LeafPage *>(page_->GetData());
    index_ = 0;
  }
  else
  {
    index_++;
  }

  return *this;
}

INDEX_TEMPLATE_ARGUMENTS
bool INDEXITERATOR_TYPE::operator==(const IndexIterator &itr) const
{
  if(itr.index_ == this->index_
    && itr.leaf_page_->GetPageId() == this->leaf_page_->GetPageId())
  {
    return true;
  }
  return false;
}

INDEX_TEMPLATE_ARGUMENTS
bool INDEXITERATOR_TYPE::operator!=(const IndexIterator &itr) const
{
  if(itr.index_ == this->index_
    && itr.leaf_page_->GetPageId() == this->leaf_page_->GetPageId())
  {
    return false;
  }
  return true;
  // return false;
}
```

4.4 Catalog Manager

4.4.1 概述

该模块负责管理、维护数据库的模式信息，主要包含两部分。

- 目录元信息
- 表和索引的管理

4.4.2 元信息、序列化与反序列化

数据库中定义的表和索引在内存中分别包含其在定义时的元信息`table_meta_data`和`index_meta_data`，为了将所有表和索引的定义信息持久化到数据库文件并在重启时从数据库文件中恢复，需要实现表和索引的元信息`TableMetadata`和`IndexMetadata`的序列化和反序列化操作。

在序列化时，需要为每一个表和索引都分配一个单独的数据页用于存储序列化数据，因此需要用于记录和管理这些表和索引的元信息被存储在哪个数据页中的数据对象`CatalogMeta`，它的信息将会被序列化到数据库文件的第`CATALOG_META_PAGE_ID`号数据页中，默认值为0。

```
class CatalogMeta {
 private:
  static constexpr uint32_t CATALOG_METADATA_MAGIC_NUM = 89849;
  std::map<table_id_t, page_id_t> table_meta_pages_;
  std::map<index_id_t, page_id_t> index_meta_pages_;
};
```

上图为`CatalogMeta`的数据成员，以`table_meta_pages`在序列化时，需要先写入其页数，然后用`MACH_WRITE_TO(Type, buf, DATA)`进行序列化。

反序列化时，根据写入的页数用`MACH_READ_FROM(Type, buf)`逐个取出。

```
class TableMetadata {
private:
  static constexpr uint32_t TABLE_METADATA_MAGIC_NUM = 344528;
  table_id_t table_id_;
  std::string table_name_;
  page_id_t root_page_id_;
  Schema *schema_;
};
```

上图为TableMetadata的数据成员，同样使用MACH_WRITE_TO(Type, buf, DATA)函数进行序列化；`string`类型的`table_name`需先获取其长度，再用`memcpy`将其长度与内容依次写入；`schema`则调用`Schema`类里的`SerializeTo(char *buf)`函数进行序列化。

反序列化时，需要定义一个临时字符数组`tmp`用于读取`table_name_`，`schema_`需要调用`Schema`类里的`DeserializeFrom(char *buf, MemHeap *heap)`函数进行反序列化。最后调用`TableMetaData`类中的`Create`函数分配空间并存储上述反序列化得出的信息。

```
class IndexMetadata {
private:
  static constexpr uint32_t INDEX_METADATA_MAGIC_NUM = 344528;
  index_id_t index_id_;
  std::string index_name_;
  table_id_t table_id_;
  std::vector<uint32_t> key_map_;  /** The mapping of index key to tuple key */
};
```

上图为 `IndexMetadata` 的数据成员，其序列化与反序列化操作与 `TableMetadata` 基本一致。序列化时，`key_map_` 需要循环调用 `MACH_WRITE_UINT32(buf, DATA)` 逐个写入；反序列化时，需要定义 `key_map` 向量，同样通过循环使用 `vector` 类的 `push_back` 函数读出并存放于其中。

4.4.3 表和索引的管理

类 `CatalogManager` 在数据库实例（`DBStorageEngine`）初次创建时（`init = true`）初始化元数据；并在后续重新打开数据库实例时，从数据库文件中加载所有的表和索引信息。

```
class TableInfo {
 public:
  TableMetadata *table_meta_;
 private:
  TableHeap *table_heap_;
  MemHeap *heap_;
};

class IndexInfo {
private:
  IndexMetadata *meta_data_;
  Index *index_;
  TableInfo *table_info_;
  IndexSchema *key_schema_;
  MemHeap *heap_;
};
```

上图为 `TableInfo` 和 `IndexInfo` 类的数据成员，用于存放从数据库文件中加载出的表和索引信息，并且置于内存中。

```
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema
*schema, Transaction *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);

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);
```

以上函数承担操作数据库的功能，包括表和索引的创建、查找、加载、删除，将`CatalogMeta`强制写进磁盘的函数`FlushCatalogMetaPage()`，在此不加赘述。

4.5 SQL Executor

4.5.1 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`

4.5.2 执行器体系结构

执行器有以下函数需要我们实现。其基本操作为通过一个语法树根节点对语法树进行解析，得到相应的操作命令，同时向下传递指令，通过数据库底层的类和数据结构对命令进行执行操作。

```
dberr_t ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteSelect(pSyntaxNode ast, ExecuteContext *context);

dberr_t ExecuteInsert(pSyntaxNode ast, ExecuteContext *context);
```

```
dberr_t ExecuteDelete(pSyntaxNode ast, ExecuteContext *context);
```

4.5.3 执行器具体实现

在数据库层面，通过`dbs_`这个`unordered_map`传入数据库名以获取当前的数据库存储引擎的指针，通过该指针可以调用数据库中的各项次级结构。

```
[[maybe_unused]] std::unordered_map<std::string, DBStorageEngine *> dbs_; /**
all opened databases */
[[maybe_unused]] std::string current_db_; /** current database */
```

在表的层面，以如下代码为例，通过`DBStorageEngine`下的次级结构`CatalogManager`可以获取有关单个表的信息，保存在`TableInfo`中间。

```
DBStorageEngine* current_db_engine = dbs_[current_db_]; if(current_db_engine ==
nullptr) {
    std::cout << "no database selected." << endl;
    return DB_FAILED;
}

std::vector<TableInfo*> vec_table_info;
current_db_engine->catalog_mgr_->GetTables(vec_table_info);
for(int i=0; i<int(vec_table_info.size()); i++)
{
    std::cout << vec_table_info[i]->GetTableName() << std::endl; }
std::cout << "table(s) of number: " << vec_table_info.size() << endl;
return DB_SUCCESS;
```

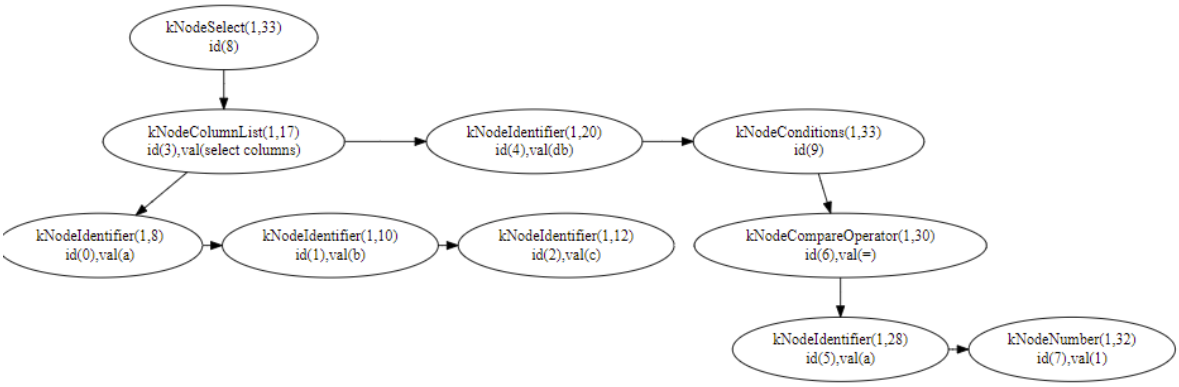
在索引的层面，同样是从`Catalogmanager`中获取相应的信息。

```
vector<TableInfo*> vec_tableinfo;
// vector<vector<IndexInfo*>> vvec_indexinfo;
dberr_t if_gettable_success = dbs_[current_db_]->catalog_mgr_-
>GetTables(vec_tableinfo);
if(if_gettable_success != DB_SUCCESS)
    return if_gettable_success;
CatalogManager* current_CMgr = dbs_[current_db_]->catalog_mgr_; for(TableInfo*
tmp_tableinfo: vec_tableinfo)
{
    vector<IndexInfo*> vec_tmp_indexinfo;
    string table_name = tmp_tableinfo->GetTableName();
    dberr_t if_getindex_success = current_CMgr-> GetTableIndexes(table_name,
vec_tmp_indexinfo);
    if(if_getindex_success != DB_SUCCESS)
        return if_getindex_success;
    for(IndexInfo* tmp_indexinfo: vec_tmp_indexinfo)
    {
        string index_name = tmp_indexinfo->GetIndexName();
        std::cout << index_name << endl;
    }
}
```

在获取单个元组的数据的层面，通过 `TableHeap` 进行 `GetTuple` 操作，从而获取单个元组的数据。

```
uint32_t index_val;
RowId r_rowid(rowid);
Row row(r_rowid);
tmp_table_info->GetTableHeap()->GetTuple(&row, nullptr);
```

语法树架构方面，语法树中保存着命令执行需要的各种信息，根据信息和指令对于各项数据进行调用。



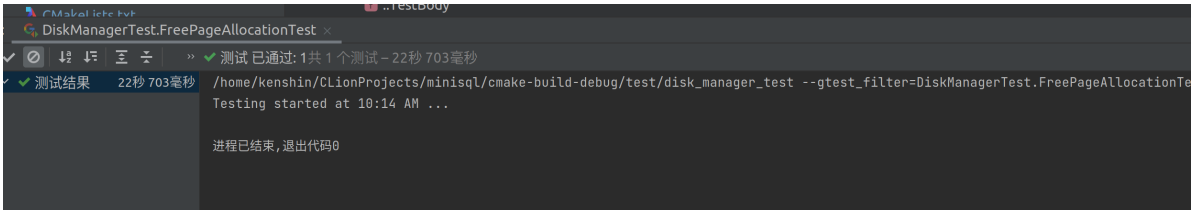

```
/**
* Syntax node definition used in abstract syntax tree.
*/
struct SyntaxNode {
    int id_; /** node id for allocated syntax node, used for debug */
    SyntaxNodeType type_; /** syntax node type */
    int line_no_; /** line number of this syntax node appears in sql */
    int col_no_; /** column number of this syntax node appears in sql */
    struct SyntaxNode *child_; /** children of this syntax node */
    struct SyntaxNode *next_; /** siblings of this syntax node, linked by a
single linked list */
    char *val_; /** attribute value of this syntax node, use deep copy */
};
typedef struct SyntaxNode *pSyntaxNode;
```

5. 测试

5.1 Disk and Buffer Pool Manager

5.1.1 磁盘数据页管理

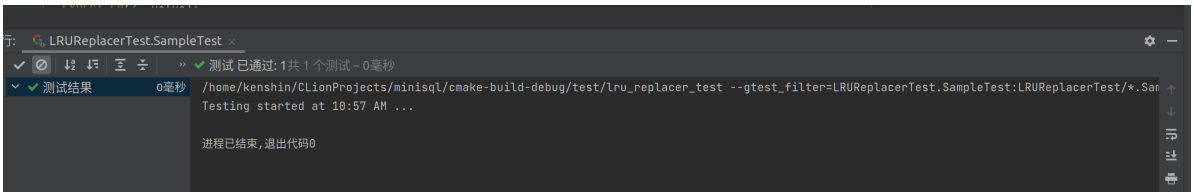
测试结果如下：



5.1.2 缓冲区替换策略

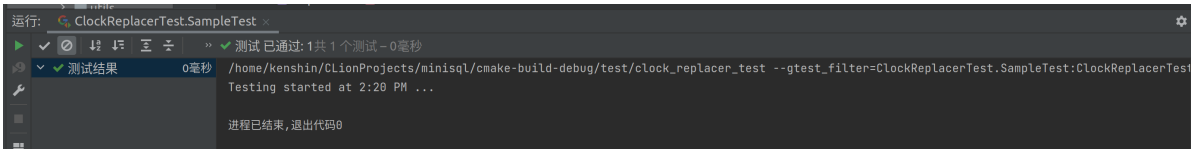
LRU替换策略

测试结果如下：



Clock替换策略

测试结果如下：



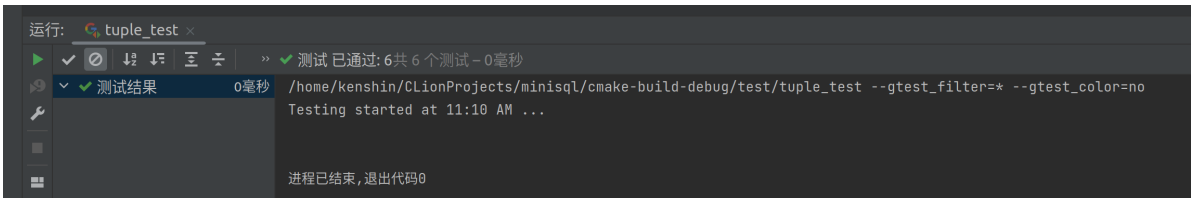
5.1.3 缓冲池管理

测试结果如下：

5.2 Record Manager

`tuple_test.cpp`

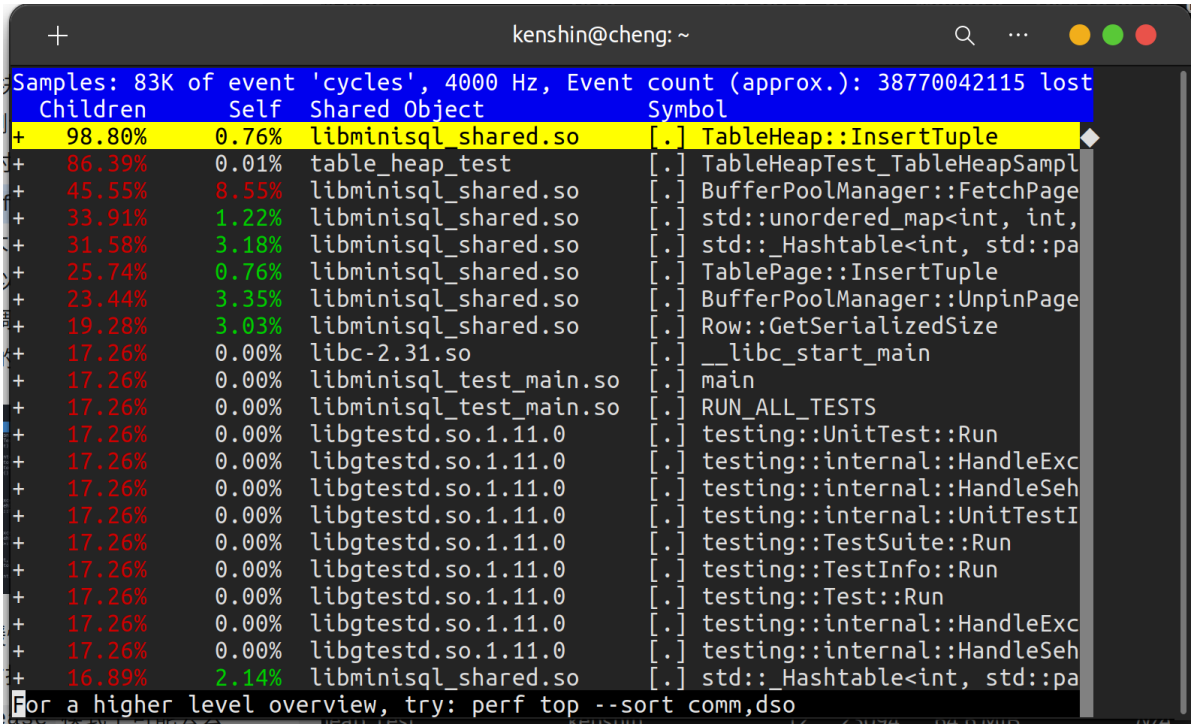
由于测试文件缺失，我自己写了 `Column` 和 `Field` 的测试。整体测试结果如下：



`table_heap_test.cpp`

这里测试插入时只测试了1000行，我测试时很快，不到1s就插入完成了。于是我先将 `row_nums` 改成100000，发现此时插入到69000左右时会插入失败，进行debug后发现是 bufferpool 已经满了且数据页全部被 Pin，无法再分配新的数据页，说明这种插入失败的情况是正常的。如果想要一次性插入很大量的数据时，需要调整位于 `src\common\config.h` 中定义的 `DEFAULT_BUFFER_POOL_SIZE` 。默认是1024，改为4096后我设置 `row_nums` 为200000进行测试，结果如下：

使用 perf 工具进行查看性能热点发现 `InsertTuple` 占据了进程98%以上的资源，符合预期。

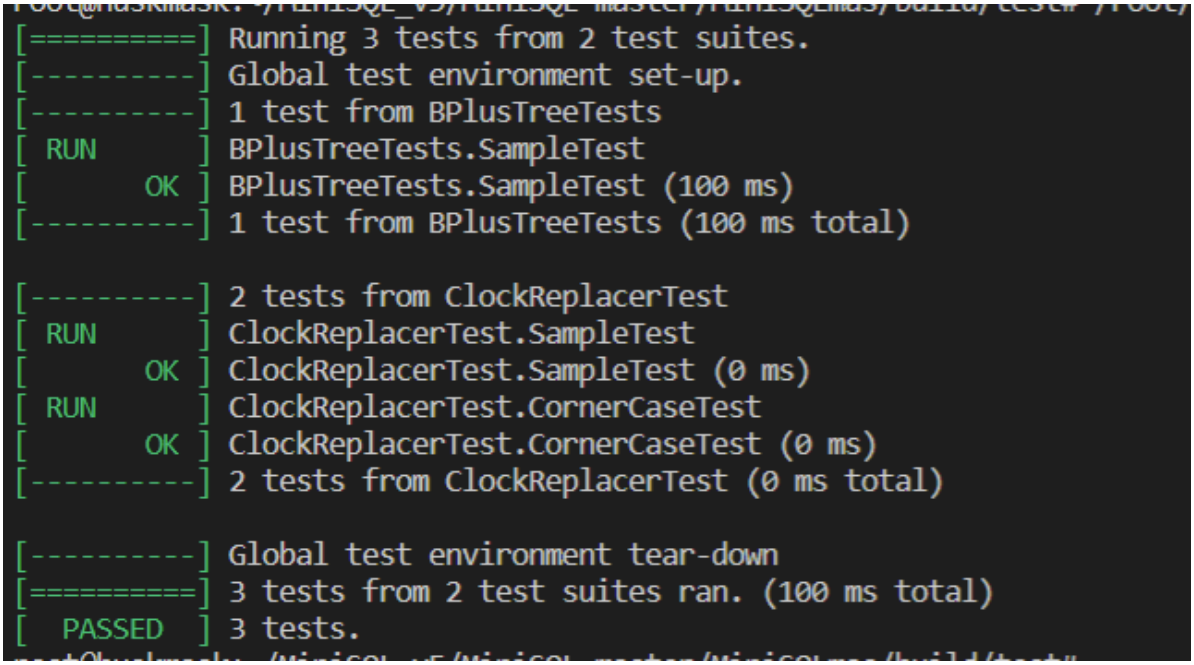


5.3 Index Manager

5.3.1 B+树的检验

检验代码通过将一个`int`向量随机排序作为`value`，原顺序作为`key`，将其插入B+树中，再将部分节点移除，通过原数组进行比较和检验。同时，测试代码还会通过`Check`检验是否每一个`Page`都在B+树操作结束后被正常的`Unpin`了。

通过`b_plus_tree_test.cpp`中的代码对程序的准确性进行检验：



执行测试，发现测试结果正确。

增大数据量进行测试，检验B+树的稳定性：

```
TEST(BPlusTreeTests, SampleTest) {
  // Init engine
  DBStorageEngine engine(db_name);
  BasicComparator<int> comparator;
  BPlusTree<int, int, BasicComparator<int>> tree(0, engine.bpm
  TreeFileManagers mgr("tree_");
  // Prepare data
  const int n = 100000;
  vector<int> keys;
  vector<int> values;
  vector<int> delete_seq;
  map<int, int> kv_map;
  ASSERT_TRUE(tree.Check());
```

```
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (355 ms)
[----------] 1 test from BPlusTreeTests (356 ms total)

[----------] 2 tests from ClockReplacerTest
[ RUN      ] ClockReplacerTest.SampleTest
[       OK ] ClockReplacerTest.SampleTest (0 ms)
[ RUN      ] ClockReplacerTest.CornerCaseTest
[       OK ] ClockReplacerTest.CornerCaseTest (0 ms)
[----------] 2 tests from ClockReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test suites ran. (356 ms total)
[  PASSED  ] 3 tests.
```

发现结果也是正确的。

5.3.2 B+树索引的检验

B+树索引的准确性检验和B+树自身的检验相近，不同的是将插入的元素改为了`RowId`和`Row`。通过`Comparator`对`Key`也就是`Row`进行比较，检验增加索引接口后的程序准确性。

运行测试程序，可以看到B+树索引也是准确的。

```
[==========] Running 4 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (16 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (8214 ms)
[----------] 2 tests from BPlusTreeTests (8231 ms total)

[----------] 2 tests from ClockReplacerTest
[ RUN      ] ClockReplacerTest.SampleTest
[       OK ] ClockReplacerTest.SampleTest (0 ms)
[ RUN      ] ClockReplacerTest.CornerCaseTest
[       OK ] ClockReplacerTest.CornerCaseTest (0 ms)
[----------] 2 tests from ClockReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 2 test suites ran. (8231 ms total)
[  PASSED  ] 4 tests.
```

5.3.3 迭代器的检验

运行测试代码，可以看到运行样例全都通过。

```
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (21 ms)
[----------] 1 test from BPlusTreeTests (21 ms total)

[----------] 2 tests from ClockReplacerTest
[ RUN      ] ClockReplacerTest.SampleTest
[       OK ] ClockReplacerTest.SampleTest (0 ms)
[ RUN      ] ClockReplacerTest.CornerCaseTest
[       OK ] ClockReplacerTest.CornerCaseTest (0 ms)
[----------] 2 tests from ClockReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test suites ran. (21 ms total)
[  PASSED  ] 3 tests.
```

5.4 Catalog Manager

通过test文件夹下的catalog_test.cpp对catalog实现的正确性进行检验。

```
[==========] Running 7 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 5 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (0 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableOperationTest
[       OK ] CatalogTest.CatalogTableOperationTest (19 ms)
[ RUN      ] CatalogTest.CatalogIndexOperationTest
[       OK ] CatalogTest.CatalogIndexOperationTest (7 ms)
[----------] 5 tests from CatalogTest (27 ms total)

[----------] 2 tests from ClockReplacerTest
[ RUN      ] ClockReplacerTest.SampleTest
[       OK ] ClockReplacerTest.SampleTest (0 ms)
[ RUN      ] ClockReplacerTest.CornerCaseTest
[       OK ] ClockReplacerTest.CornerCaseTest (0 ms)
[----------] 2 tests from ClockReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 2 test suites ran. (28 ms total)
[  PASSED  ] 7 tests.
```

测试通过，运行无误。

5.5 SQL Executor

5.5.1 前置模块检验

首先在检验前四个模块能够通过测试的基础上：

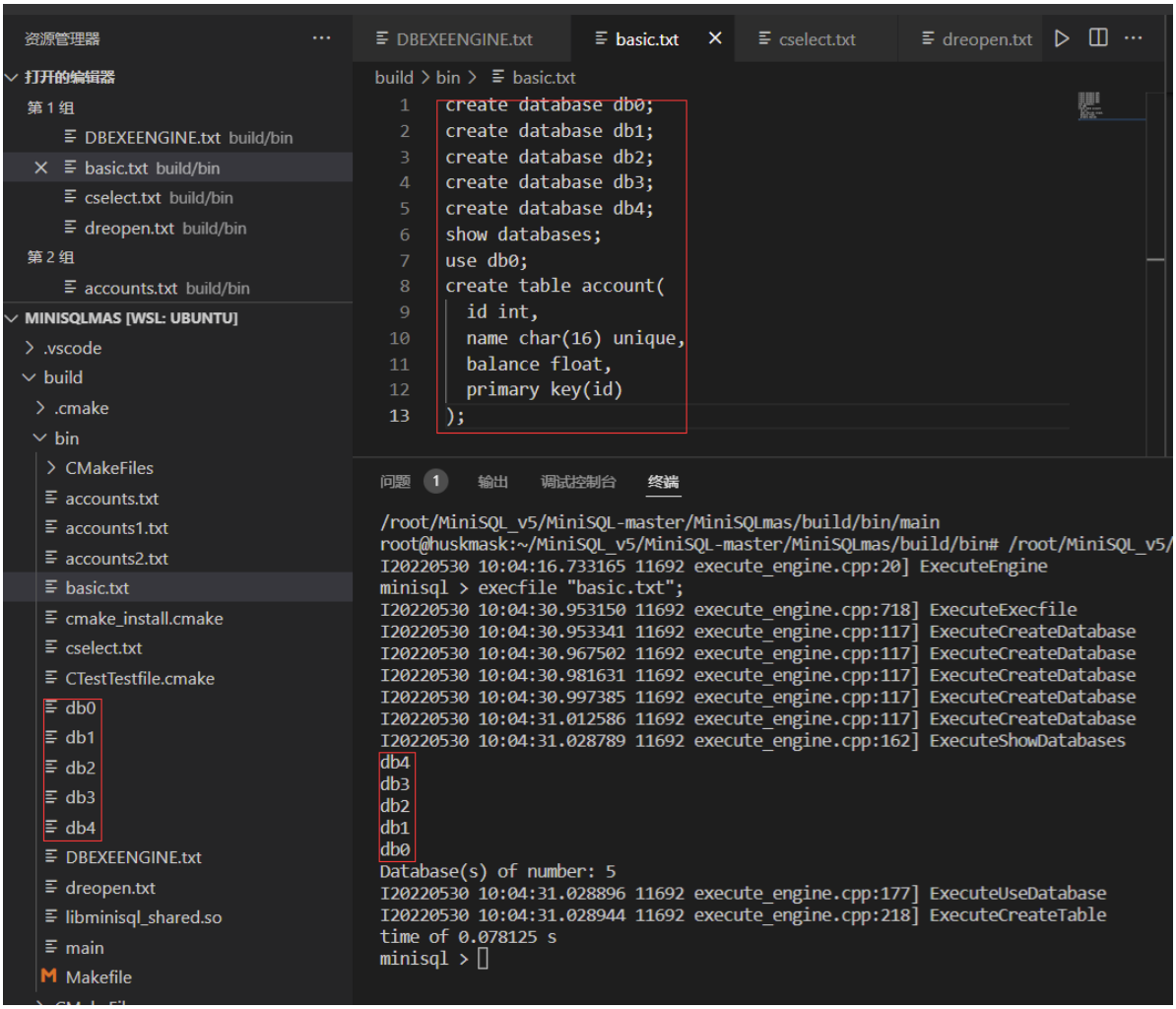
```
root@huskmask:~/MiniSQL_v5/MiniSQL-master/MiniSQLmas/build/test# /root/MiniSQL_v5/MiniSQL-master/MiniSQLmas/build/test/minisql_test
[==========] Running 21 tests from 9 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (3 ms)
[----------] 1 test from BufferPoolManagerTest (3 ms total)

[----------] 2 tests from ClockReplacerTest
[ RUN      ] ClockReplacerTest.SampleTest
[       OK ] ClockReplacerTest.SampleTest (0 ms)
[ RUN      ] ClockReplacerTest.CornerCaseTest
[       OK ] ClockReplacerTest.CornerCaseTest (0 ms)
[----------] 2 tests from ClockReplacerTest (0 ms total)

[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (0 ms)
[----------] 1 test from LRUReplacerTest (0 ms total)

[----------] 5 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (0 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableOperationTest
[       OK ] CatalogTest.CatalogTableOperationTest (32 ms)
[ RUN      ] CatalogTest.CatalogIndexOperationTest
[       OK ] CatalogTest.CatalogIndexOperationTest (10 ms)
[----------] 5 tests from CatalogTest (43 ms total)

[----------] 4 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (5 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (8507 ms)
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (85 ms)
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (8 ms)
[----------] 4 tests from BPlusTreeTests (8608 ms total)
```

```
[----------] 4 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (5 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (8507 ms)
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (85 ms)
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (8 ms)
[----------] 4 tests from BPlusTreeTests (8608 ms total)

[----------] 1 test from PageTests
[ RUN      ] PageTests.IndexRootsPageTest
[       OK ] PageTests.IndexRootsPageTest (0 ms)
[----------] 1 test from PageTests (0 ms total)

[----------] 4 tests from TupleTest
[ RUN      ] TupleTest.FieldSerializeDeserializeTest
[       OK ] TupleTest.FieldSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.ColumnSerializeDeserializeTest
[       OK ] TupleTest.ColumnSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.RowSerializeDeserializeTest
[       OK ] TupleTest.RowSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (0 ms)
[----------] 4 tests from TupleTest (0 ms total)

[----------] 2 tests from DiskManagerTest
[ RUN      ] DiskManagerTest.BitMapPageTest
[       OK ] DiskManagerTest.BitMapPageTest (2 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (21896 ms)
[----------] 2 tests from DiskManagerTest (21899 ms total)

[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (133825 ms)
[----------] 1 test from TableHeapTest (133825 ms total)

[----------] Global test environment tear-down
[==========] 21 tests from 9 test suites ran. (164380 ms total)
[  PASSED  ] 21 tests.
root@huskmask:~/MiniSQL_v5/MiniSQL-master/MiniSQLmas/build/test#
```

5.5.2 建表建库操作检验

在此基础上完成模块五的编写后通过命令行指令进行检验。

首先执行文件内指令，进行建库和建表操作：



建表操作成功进行，数据库文件也成功生成。

5.5.3 插入操作

执行 `execfile` 指令，执行 accounts.txt 中的插入指令，插入 20000 条数据。

accounts.txt

build > bin > accounts.txt

```
19967   insert into account values(3209964, "name19964", 450.227);
19968   insert into account values(3209965, "name19965", 460.277);
19969   insert into account values(3209966, "name19966", 729.32);
19970   insert into account values(3209967, "name19967", 888.833);
19971   insert into account values(3209968, "name19968", 796.2);
19972   insert into account values(3209969, "name19969", 808.575);
19973   insert into account values(3209970, "name19970", 524.238);
19974   insert into account values(3209971, "name19971", 309.779);
19975   insert into account values(3209972, "name19972", 995.852);
19976   insert into account values(3209973, "name19973", 443.903);
19977   insert into account values(3209974, "name19974", 264.86);
19978   insert into account values(3209975, "name19975", 30.45);
19979   insert into account values(3209976, "name19976", 525.778);
19980   insert into account values(3209977, "name19977", 165.733);
19981   insert into account values(3209978, "name19978", 543.061);
19982   insert into account values(3209979, "name19979", 207.656);
19983   insert into account values(3209980, "name19980", 14.072);
```

插入成功进行，耗时17.72s。

```
I20220530 10:06:21.416795 11692 execute_engine.
time of 17.7188 s
minisql >
```

执行全选，检验插入成功：

```
    3209988 name19988    324.267
    3209989 name19989    822.834
    3209990 name19990    810.623
    3209991 name19991    555.168
    3209992 name19992    747.873
    3209993 name19993    719.484
    3209994 name19994    750.632
    3209995 name19995    316.979
    3209996 name19996     725.17
    3209997 name19997    653.261
    3209998 name19998    687.217
    3209999 name19999    603.553
Row(s) for number 20000
time of 0.515625 s
minisql >
```

5.5.4 选取操作

```
db2
db1
db0
Database(s) of number: 5
I20220530 10:04:31.028896 11692 execute_engine.cpp:177] ExecuteUseDatabase
I20220530 10:04:31.028944 11692 execute_engine.cpp:218] ExecuteCreateTable
time of 0.078125 s
minisql > select * from account where id = 12500011;
I20220530 10:05:16.752043 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0 s
minisql >
```

执行 select 语句，可以看到在插入完成后能够成功获取指定id的信息。

5.5.5 删除、建立索引操作

建索引前后搜索时间对比，可以明显看到搜索速度有所提升，删除后下降，说明索引成功建成，并且能够成功地提升搜索的效率。

```
time of 0.515625 s
minisql > select * from account where name = "name56789";
I20220530 10:09:36.207022 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0.203125 s
minisql > create index idx01 on account(name);
I20220530 10:10:06.903971 11692 execute_engine.cpp:410] ExecuteCreateIndex
time of 0 s
minisql > select * from account where name = "name56789";
I20220530 10:10:14.756219 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0.125 s
minisql >
```

```
minisql > drop index idx01;
I20220530 10:11:30.915122 11692 execute_engine.cpp:464] ExecuteDropIndex
time of 0.015625 s
minisql > select * from account where name = "name56789";
I20220530 10:11:39.799469 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0.203125 s
minisql >
```

5.5.6 删除操作

通过执行 delete 对于元组进行删除：

```
Row(s) for number 0
time of 0.203125 s
minisql > delete from account where balance = 666.66;
I20220530 10:12:10.948555 11692 execute_engine.cpp:610] ExecuteDelete
affects row(s) of number: 1
time of 0.1875 s
minisql > select * from account where balance = 666.66;
I20220530 10:12:21.963351 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0.1875 s
minisql >
```

```
minisql > delete from account;
I20220530 10:12:58.375593 11692 execute_engine.cpp:610] ExecuteDelete
affects row(s) of number: 20000
time of 1.9375 s
minisql > select * from account;
I20220530 10:13:05.122617 11692 execute_engine.cpp:508] ExecuteSelect
Row(s) for number 0
time of 0 s
```

可以看到删除操作得到正确的结果。

5.5.7 数据持久性检验

执行 quit 指令后重新运行程序，可以发现之前一次的数据都保留在数据库中：

```
minisql > quit;
I20220530 10:13:31.630319 11692 execute_engine.cpp:816] ExecuteQuit
time of 0 s
bye!
root@huskmask:~/MiniSQL_v5/MiniSQL-master/MiniSQLmas/build/bin# /root/MiniSQ
I20220530 10:13:40.642994 12538 execute_engine.cpp:20] ExecuteEngine
minisql > show databases;
I20220530 10:13:48.024185 12538 execute_engine.cpp:162] ExecuteShowDatabases
db0
db1
db2
db3
db4
Database(s) of number: 5
time of 0 s
minisql >
```

```
time of 0 s
minisql > select * from account;
I20220530 10:17:44.188844 13009 execute_engine.cpp:508] ExecuteSelect
   3209892 name19892    149.332
   3209893 name19893    770.845
Row(s) for number 2
time of 0 s
minisql > quit;
I20220530 10:17:57.589895 13009 execute_engine.cpp:816] ExecuteQuit
time of 0 s
bye!
root@huskmask:~/MiniSQL_v5/MiniSQL-master/MiniSQLmas/build/bin# /root/MiniSQ
I20220530 10:18:01.742743 13261 execute_engine.cpp:20] ExecuteEngine
minisql > use db0;
I20220530 10:18:06.484812 13261 execute_engine.cpp:177] ExecuteUseDatabase
time of 0 s
minisql > select * from account;
I20220530 10:18:16.753502 13261 execute_engine.cpp:508] ExecuteSelect
   3209892 name19892    149.332
   3209893 name19893    770.845
Row(s) for number 2
time of 0 s
```

6. 建设性意见

建议能够只给出索引的接口，同时提供以另一种数据结构（如哈希等）实现的现成的索引作为样例，让同学们体会如何具体地实现索引。首先B+树索引的难度比较大，但由于后续模块的需要，如果不实现就无法进行后续模块的编写，这就导致一个小组的进度被单个同学拖累，如果有一个现成的索引，一个同学写B+树的时候其他同学可以进行后续模块的开发，这可能能够更好地平衡小组内部的工作量；同时目前的模块架构较为复杂，注释也含糊不清，让同学们自己在体会索引的基本架构后自行建立B+树的架构可能会更好一点，或者将原本的框架作为一个参考，同学们可以自行修改B+树类的结构。

更改 Row 等部分的操作，在底层获取数据的功能，目前很多操作比如使用 `RowId` 获取元组的操作需要在顶层操作，导致不同同学之间的分工不明确。增加 `TableInfo` 持久化的测试样例，目前这部分没有测试样例。