MiniSQL 个人报告

秦嘉俊
3210106182

1 负责部分

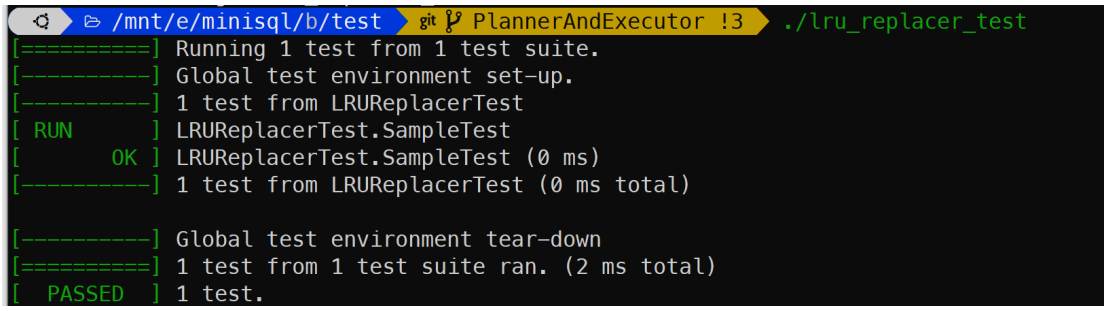
- Lab1: LRU Replacer、bonus: Clock Replacer
- Lab2: TableHeap、iterator
- Lab3: BPlusTree Index、iterator
- Lab4: 协助 Catalog Manager debug
- Lab5: Seqscan 算子、Database/Table/Index/Quit 相关语句

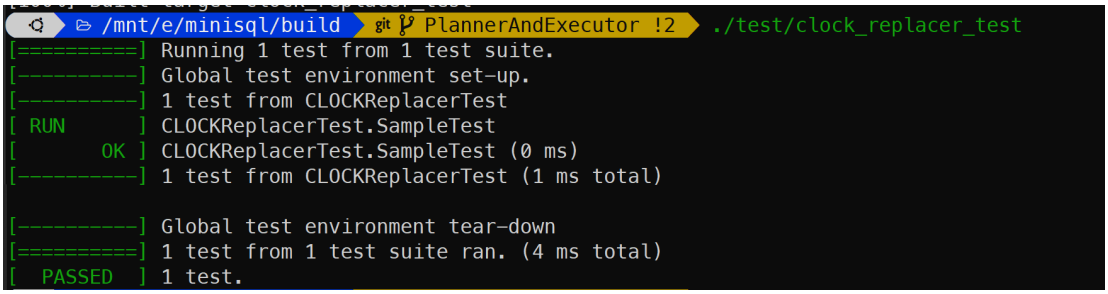
2 具体实现

2.1 Lab1

LRU 部分，我维护了一个 `unorder_set` 的容器 `lru_list_`. 在没有自己定义哈希函数时，`unorder_set` 会保持元素插入的顺序，即直接把最新的元素放在 `begin` 的位置。因此根据 LRU 的策略，每次需要选择元素驱逐时只需要选择容器末尾的元素就可以了，因为它是最先插入的。
其他 `Pin, Unpin, Size` 均非常简单，调用 `unorder_set` 的成员函数即可。
这里 `unorder_set` 的插入、查询、删除的时间复杂度是 $avg = O(1), worst = O(n)$, 已经达到了相当高的效率。我尝试了使用 `deque, vector` 等容器，均不如 `unorder_set`.

而 Clock Replacer 的实现也比较简单。我们用一个 `list` 来作为容器，用 `clock_st` 作为容器的指针，指向我们目前遍历到的元素，`clock_status` 表示每个元素的状态。当我们遍历到这个元素时，如果状态为 0 那我们可以踢出它，否则将它的状态变为 0.
值得注意的是我们要实现一个循环队列，因此在插入时(`Unpin`)我们要检查，如果队列已经满了，我们就要踢出去一个元素，随后再进行插入。而且在 `Victim` 中如果我们遍历到列表的末尾，也需要手动让指针指向容器的开头，以此实现循环队列。
Clock Replacer 的测试代码也完全仿照 LRU 部分，除了将类替换掉，我们的代码页通过了测试。





代码较简单，这里不再贴出。

2.2 Lab2

2.2.1 TableHeap

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

- 对于构造函数，我们需要申请一个空白页，并将 `first_page_id` 指向这页。
- 对于 `InsertTuple` 操作，首先我们判断序列化后的大小是否超过了一页能放下的大小，如果是那么我们应该直接返回 `false`. 随后我们取出堆表中的第一页，对于每一页我们都尝试利用当前页的 `InsertTuple` 操作（TablePage），如果成功，我们可以直接返回 `true`. 否则需要继续取下一页。如果我们已经达到了当前堆表的结尾，即当前堆表所有页都无法放下这个元组，那我们需要通过 `buffer_pool_manager` 新建一页并完成相关初始化操作，再将元组插入到这页上。
- 对于 `UpdateTuple` 操作，我们这里修改了框架中 TablePage 的 `UpdateTuple` 接口，添加了 `state` 操作用来表示当前元组无法更新的原因是什么。如果为 0 表示非法 slotnum, 1 表示元组已经被标记删除那我们

不应该对其更新。若为 2 说明没有足够空间，那我们应该把当前这个元组删除，再尝试将元组插入。需要注意的是这里不能使用 TablePage 的 `InsertTuple` 函数，因为我们这里不一定能插入到原来的位置，而是应该在整个堆表中插入，所以要调用我们刚刚实现的 TableHeap 的 `InsertTuple` 函数。

- 对于 `ApplyDelete` 和 `GetTuple` 操作，只需要根据 rid 取出对应的页，随后调用对应页的对应操作即可。

2.2.2 TableHeap Iterator

堆表迭代器是用来从第一页的第一个元组开始，依次遍历堆表中的所有元组。

在设计迭代器时，首先这个类肯定需要对应的 `table_heap` 指针。此外我们还需要知道当前指向哪个元组，因此我们保存了 `now_page_id` 和 `now_row` 分别表示当前的页号以及当前元组的指针，同时这样也便于后面的运算符重载。这里我们不能存当前页的指针，否则在我们到下一页之前这一页就相当于一直被我们 pin 住了，降低了资源的利用率。

- 在带参数和拷贝构造函数中，如果传入的参数 `now_row/other.now_now` 不为空的话，我们需要新建一个相同的 Row 对象并将这个构造的对象的 `now_row` 指向他，否则我们我们所有的 Row 都是同一对象，可能会产生意想不到的并发问题。
  相应地，析构时我们也要析构这个对象，即 `delete now_row`.

- `* ->` 运算符重载只需要返回对应的成员变量即可。

- 对于 `++iter` 操作，首先我们要查找在当前页中，当前元组是否有后续元组（`GetNextTupleRid`）。如果有那我们就已经找到下一个元组，更新 `now_row` 即可。需要注意的是我们不能直接设置 rid 后调用 `GetTuple`，因为 `row` 中添加属性的方法是 `push_back`，而不会覆盖原本的数据。这样就会导致我们的元组数据一个一个的接在了一起。因此我们需要 `delete now_row` 并新建一个 `Row` 的对象，再利用 `GetTuple` 获得数据即可。
  如果没有找到，需要进入下一页。每一页我们尝试通过 `GetFirstTupleRid` 获得第一个元组，如果找到我们只需要和刚刚一样完成相关设置即可。否则我们需要继续往下寻找。需要注意的是如果当前页为空，不代表我们走到头了，因为这页最开始是有数据的，后来全部被删除了。因此我们仍然要遍历后续的页直到 `INVALID_PAGE_ID`.

- 对于 `iter++` 操作，我们只需要利用刚刚的 `++iter` 即可。先保存当前对象，随后再 `++iter`，返回原本的对象。

此外我们 TableHeap 中的 `Begin` 和 `End` 也属于迭代器的范畴。

- 对于 `End` 操作，我们只需要返回一个 `TableIterator(this, nullptr, INVALID_PAGE_ID)` 对象即可。（我们约定这样为 `End()`，因此在刚刚的

`++iter` 操作中如果我们已经遍历完了所有元组，我们也会将迭代器设置为这样）

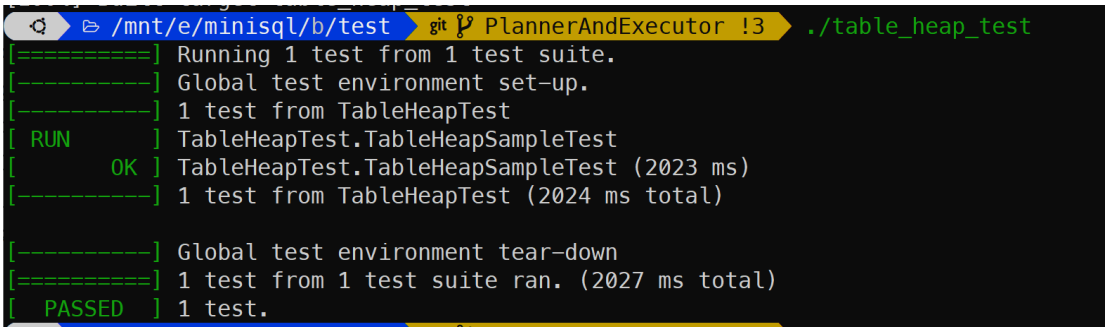
- 对于 `Begin` 操作，与 `++iter` 类似，我们从第一页开始遍历，找到第一个元组就返回。如果遍历了整个堆表都没有找到有效元组，就返回 `End()`.

对于 table_heap_test, 测试只测试了 Insert 和 GetTuple 相关的操作，于是我添加了 Update、Delete 以及迭代器的相关测试代码。

```
ASSERT_EQ(size, 0);
  int count = 0;
  for (auto iter = table_heap->Begin(nullptr); iter !
= table_heap->End(); ++iter) {
    Row row(iter->GetRowId());
    table_heap->GetTuple(&row, nullptr);
    ASSERT_EQ(schema.get()->GetColumnCount(), row.Get
Fields().size());
    ASSERT_EQ(schema.get()->GetColumnCount(), iter->G
etFields().size());
    for (size_t j = 0; j < schema->GetColumnCount();
j++) {
      ASSERT_EQ(CmpBool::kTrue, row.GetField(j)->Comp
areEquals(*iter->GetField(j)));
    }
    count++;
  }
  ASSERT_EQ(row_nums, count);
  std::unordered_map<int64_t, Fields *> row_values2;
  std::set<page_id_t>Used_Page;
  for (int i = 0; i < row_nums; i++) {
    int32_t len = RandomUtils::RandomInt(0, 64);
    char *characters = new char[len];
    RandomUtils::RandomString(characters, len);
    Fields *fields = new Fields{
            Field(TypeId::kTypeInt, i),
            Field(TypeId::kTypeChar, const_cast<char
*>(characters), len, true),
            Field(TypeId::kTypeFloat, RandomUtils::Ra
ndomFloat(-999.f, 999.f))
    };
    Row row(*fields);
    ASSERT_EQ(true,table_heap->UpdateTuple(row,row_i
d.at(i),nullptr));
    ASSERT_EQ(false,row.GetRowId().GetPageId()==INVAL
ID_PAGE_ID);
```

```
    row_values2[row.GetRowId().Get()] = fields;
    delete[] characters;

  }
  for (auto row_kv2 : row_values2) {
    Row row(RowId(row_kv2.first));
    ASSERT_EQ(true,table_heap->GetTuple(&row, nullpt
r));
    Used_Page.insert(row.GetRowId().GetPageId());
    ASSERT_EQ(schema.get()->GetColumnCount(), row.Get
Fields().size());
    for (size_t j = 0; j < schema.get()->GetColumnCou
nt(); j++) {
      ASSERT_EQ(CmpBool::kTrue, row.GetField(j)->Comp
areEquals(row_kv2.second->at(j)));
    }
    //测试delete
    //确定update没问题后，把记录删除
    table_heap->ApplyDelete(row.GetRowId(),nullptr);
    ASSERT_FALSE(table_heap->GetTuple(&row,nullptr));
  }
  //测试free
  // cout << "all unpinned? " << bpm_->CheckAllUnpinn
ed() << endl;
  table_heap->FreeTableHeap();
  auto *disk_manager = new DiskManager(db_file_name);
  auto *bpm = new BufferPoolManager(100, disk_manage
r);
  ASSERT_EQ(true,bpm->IsPageFree(*Used_Page.begin
()));
  for ( auto page_iter = Used_Page.begin() ; page_ite
r!=Used_Page.end();page_iter++){
    ASSERT_EQ(true,disk_manager->IsPageFree(*page_ite
r));
  }
```



除此之外，值得注意的是，每次根据 pageid 取出数据页后，我们在对页的访问结束后需要 unpin 这页，并根据是否有对这页进行修改传入 diry bit. 后面的 B+ 树以及其他部分均是如此，会有对应的 `CheckAllPinned` 函数来检查是否所有的数据页都被取消固定了。

2.3 Lab3

B+ 树的数据页以及其功能函数以及由另一个同学实现，我则是基于此实现整个 B+ 树。
这里叶子页有 N 个 key 和 value, value 是 RowId, 唯一地标识了堆表中元素的位置。而中间节点页也有 N 个 key 和 value, 但第零个 key 是 INVALID. 这里的 value 是 page_id, 唯一地标识了儿子页的页号。

2.3.1 BPlusTree

- 对于构造函数。我们在 pageid 为 `IndexRootsPage` 的页上放了索引与对应索引根节点页的对应元组。因此初始化时我们应该先取出这个索引页，随后根据 `index_id` 取出这个索引的 `root_page_id`.(这样才能继承之前的索引)
- 对于 `UpdateRootPageId()`, 我们只需要取出索引页随后根据参数选择调用 `Insert` / `Update` 即可。
- 对于 `FindLeafPage` 操作，我们从根节点开始，每次取出当前节点的页。如果这页是叶子页我们就结束迭代，并返回这页。否则我们利用 InternalPage 的 `Lookup` 函数找到第一个大于等于 key 的数据页，并继续迭代下去。这里的 `leftMost` 如果为真，表明我们希望找到最左边的页，则我们每次往 $value_0$ 对应的页走即可。
- 对于 `GetValue` 操作，我们只需要先根据 `FindLeafPage` 找到对应的叶子页，随后利用 LeafPage 的 `Lookup` 函数即可。如果能找到，就把它放进 `result` 中。
- 对于 `Insert` 操作，首先我们判断当前树是否为空。
  - 如果为空，则调用 `StartNewTree`. 我们申请一个数据页，并将这个值插入到这个页上。并更新 `root_page_id`. 需要注意的是，我们每

次对根节点的页进行修改，都要调用 `UpdateRootPageId` 函数。

- 如果当前树不为空，我们就进入 `InsertIntoLeaf` 函数。首先我们 `FindLeafPage` 找到对应的叶子页，随后直接利用叶子页的 `Insert` 进行插入。随后我们判断当前的大小是否超过叶子页的 `GetMaxSize()`. 如果没有超过那么我们插入完成，否则我们需要进行 `Split` 操作。

  `Split` 后我们有两个叶子页，需要将这右边页的首个元素插入到父节点中，因此调用 `InsertIntoParent` 函数。

  - `Split` 操作就是将当前节点拆成两个节点。首先我们新建一个数据页，随后利用 `MoveHalfTo` 将当前页的一半元素是搬到另一个页上。并利用 `Init` 完成新页的初始化。需要注意的是这里对于叶子页，我们还需要设置 `next_page_id`. 即让当前页指向新页，新页指向当前页原先的下一页。
  - `InsertIntoParent` 操作首先判断要拆分的节点是否是根节点。如果是，那么我们拆分之后需要生成一个新的根节点页。设置 `root_page_id`, 并利用 `Init` 和 `PopulateNewRoot` 完成设置，最后 `UpdateRootPageId`。

    如果不是根节点，我们直接取出父节点，并用 `InsertNodeAfter` 方法即可。随后我们需要判断父节点是否超过了 `GetMaxSize()`. 如果是我们要执行和刚刚 `InsertIntoLeaf` 一样的操作，先 `Split` 再调用 `InsertIntoParent`.
  - 值得注意的是，我们这里的 `GetMaxSize()` 设置为能放 N+1 个键值对，因为我们在执行插入操作的时候，不会顾及当前是否已经“满”了。而是在插入之后再判断是否溢出，这就需要我们为这个 key&value 留出空间。

- 对于 `Remove` 操作，首先我们判断树是否为空。如果非空我们再取出 `FindLeafPage` 对应的数据页，随后在这页上直接调用 LeafPage 的 `RemoveAndDeleteRecord` 将这个元素删除。随后我们检查当前页的大小，如果小于了 `GetMinSize()`, 那我们需要利用 `CoalesceOrRedistribute` 进行调整。在 `CoalesceOrRedistribute` 中，首先我们判断是否是根节点，

  - 如果是根节点，那我们利用 `AdjustRoot` 对根节点进行特殊操作。我们判断如果当前页没有了元素，说明整个 B+ 树都已经被删除，那我们删除这页，将 `root_page_id` 置为 INVALID. 否则如果当前根只剩了一个儿子页，那我们也不需要当前这个根节点页，而是让儿子成为根。调用 `RemoveAndReturnOnlyChild` 并更新儿子页使其成为根即可，并 `UpdateRootPageId`.
  - 否则我们取出当前页的兄弟页，（一般为同父亲下的左兄弟，如果当前页为最左的兄弟那我们就取右边的页）。随后判断当前页和兄弟页的大小之和是否超过了 `GetMaxSize()`. 如果超过了那我们就需要在两个页之间重新分配 key&value(`Redistribute`), 否则我们就需要将两页合并为一页(`Coalesce`)

    - `Redistribute` 中，我们只需要将兄弟的一个元素搬到当前页即可。直接调用 `MoveFirstToEndOf` / `MoveLastToFrontOf` 即可，根据具体的位置关系决定是搬第一个还是最后一个。同时设置父节点的 key.
    - `Coalesce` 中，我们只需要调用 `MoveAllTo` 方法将当前页全部搬到兄弟页，并用在父节点处 `Remove` 移除掉对应的 key/value. 注意到这里我们的 `MoveAllTo` 会把值全部接在原值的后面，因此我们必须是右边的页的数据移到左边，因此对于 index 为 0 的情况需要交换兄弟页。
- 对于 `Destroy` 操作，我们采用递归删除的方式，从根节点开始，先遍历所有子树并删除，再删除根节点。值得注意的是，如果删除了根节点，我们需要利用 `UpdateRootPageId` 更新索引页，否则后续可能会出现删除表后再建一个表却继承了之前的索引的问题。

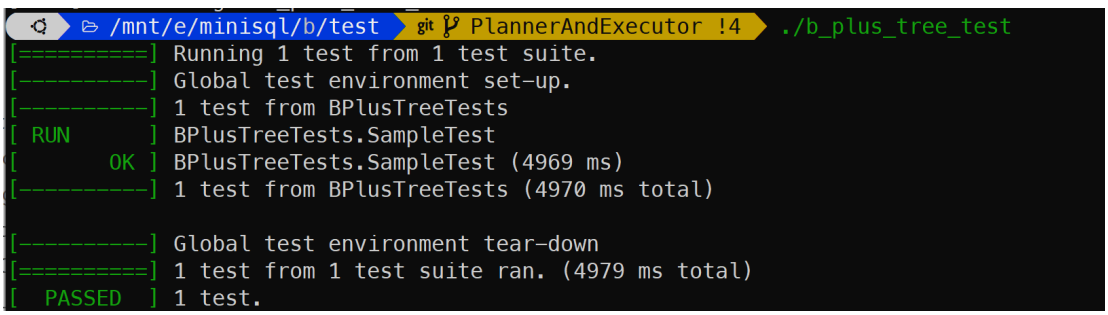
2.3.2 BPlusTree Index & Index Iterator

- 对于 `*` 操作符，我们只需要调用 page 的对应 `Keyat(index), ValueAt(index)` 函数，并将结果组合为 pair 返回即可。
- 对于 `++iter` 操作符，我们先让 `item_index++`, 随后判断是否超过了当前页的大小。如果超过，那我们需要到下一页，并将 `item_index` 设为 0. 如果下一页为空，说明我们已经到了结尾，将属性设为 INVALID 即可。

此外我们 BPlusTree 中的 `Begin` 和 `End` 也属于迭代器的范畴。

- 对于 `Begin` 操作，如果没有参数，代表我们从最小的元素开始，则我们利用 `FindLeafPage()` 函数，并把 `leftMost` 设为 true. 如果有参数，那我们利用 `FindLeafPage()` 找到这个 key 对应的页，并用叶子页的的 `KeyIndex` 函数找到第一个大于等于 key 的位置。最后返回一个迭代器对象即可。
- 对于 `End` 操作，我们只需要返回一个参数为 INVALID 的迭代器即可。

对于 b_plus_tree_test, 原先的数据量太小，只有一个节点，不能覆盖所有的情况。于是我尝试了 size 从 30 到 1000、10000、50000, N 尝试了 3、4、6、10、168 均通过测试，确保了 B+ 树的代码正确。



```
 /mnt/e/minisql/b/test  PlannerAndExecutor !4  ./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 (74 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (3062 ms)
[----------] 2 tests from BPlusTreeTests (3138 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (3141 ms total)
[  PASSED  ] 2 tests.
 /mnt/e/minisql/b/test  PlannerAndExecutor !4  ./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 (74 ms)
[----------] 1 test from BPlusTreeTests (75 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (78 ms total)
[  PASSED  ] 1 test.
```

2.4 Lab4

这个部分我协助另外两位同学 debug, 找到了部分 bug. 比如 `LoadTable` 的时候发现 `next_table_id` 的链表构成了死循环，后来发现是 `GetNextTableId` 这里返回的值没有 +1, 导致一直返回 0.

```
inline table_id_t GetNextTableId() const {
    return table_meta_pages_.size() == 0 ? 0 : table_
meta_pages_.rbegin()->first + 1; // +1 是后来修改的
  }
```

还有一个 bug 是在 `DropIndex` 时只是将索引号和名字移出了容器，并没有真正删除索引的数据。修改后在 `DropIndex` 中调用了索引的 `Destroy` 函数。

```
auto bp_index = indexes_[index_id]->GetIndex();
bp_index->Destroy();
```

2.5 Lab5

2.5.1 SeqScan 算子

- `Init` 部分我们只需要根据 catalog 取出对应的 tableinfo, 并根据 tableinfo 取出 table_heap 的头迭代器即可。
- `Next` 部分遍历整个堆表，对应每个元组，我们利用为谓词的 `Evaluate` 方法计算这个元组是否符合条件。如果符合，那么我们取出这个元组并放到结果中。否则继续遍历。

这里测试时我们注释掉了 executor_test 内的其他测试单元。

```
 /mnt/e/minisql/b/test  PlannerAndExecutor !4  ./executor_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ExecutorTest
[ RUN      ] ExecutorTest.SimpleSeqScanTest
[       OK ] ExecutorTest.SimpleSeqScanTest (94 ms)
[----------] 1 test from ExecutorTest (96 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (102 ms total)
[  PASSED  ] 1 test.
```

值得注意的是，在 SeqScan 中我们要实现投影功能。实现方法是通过 tableinfo 取出表的 schema, 并将其和 outschema 结合，找到我们要输出的列，并通过堆表迭代器找到当前元组对应的 Field.

2.5.2 Database 相关语句

- 对于 `ExecuteCreateDatabase` 操作，首先我们查询 `dbs_` 中是否已经有了重复的名字，如果没有我们利用 `DBStorageEngine` 的构造函数新建一个对象，并将指针放到 `dbs_` 中。
- 对于 `ExecuteDropDatabase` 操作，首先我们查询 `dbs_` 中是否已经有了重复的名字，如果有我们直接利用 `delete delete_db;` 调用其析构函数。
  需要注意的是，这样做只是将程序里的对象析构，但并没有将磁盘上的文件删除。因此我们还需利用 `remove(path.c_str())` 删除硬盘上的数据库文件。
- 对于 `ExecuteShowDatabases` 操作，我们遍历 `dbs_` 所有数据库，依次输出他们的名字，最后输出 size 即可。这里格式参考了 mysql 的输出方式。

```
minisql > show databases;
[INFO] Sql syntax parse ok!
+--------------------+
| Database           |
+--------------------+
Total 0 database(s)
minisql > create database db0;
[INFO] Sql syntax parse ok!
minisql > create database db1;
[INFO] Sql syntax parse ok!
minisql > show databases;
[INFO] Sql syntax parse ok!
+--------------------+
| Database           |
+--------------------+
|        db1         |
|        db0         |
+--------------------+
Total 2 database(s)
minisql > drop database db0;
[INFO] Sql syntax parse ok!
minisql > show databases;
[INFO] Sql syntax parse ok!
+--------------------+
| Database           |
+--------------------+
|        db1         |
```

```
+--------------------+
Total 1 database(s)
```

2.5.3 Table 相关语句

- 对于 `ExecuteCreateTable` 操作，首先我们取出 catalog, 查询当前是否已经有这个表了。如果没有，我们根据语法树的结构遍历属性的定义，并找出是否有 unique 属性，以及可能的类型 size(`char` 类型)。随后我们遍历主键的定义，将这些属性加入到主键的容器中。
  结束遍历后，我们根据刚刚遍历的结果初始化 `Column` 对象，并基于此创建 `TableSchema` 对象。随后我们调用 catalog 中的 `CreateTable` 函数创建表。然后还要基于主键和 unique 属性的元素创建索引，调用 `catalog` 中的 `CreateIndex` 即可。
- 对于 `ExecuteDropTable` 操作，只需要找出对应的表，找到表对应的索引并 `DropIndex`, 最后调用 `DropTable` 即可。
- 对于 `ExecuteShowTables` 操作，我们利用 catalog 中的 `GetTables` 取出所有 tableinfo, 打印名字即可。这里格式参考了 mysql 的输出方式。

```
[INFO] Sql syntax parse ok!
minisql > create table t1(a int, b char(20) unique, c float, primary key(a, c));
[INFO] Sql syntax parse ok!
minisql > create table t2(a int, b char(-5) unique, c float, primary key(a, c));
[INFO] Sql syntax parse ok!
Invalid type definition.
minisql > create table student(
  sno char(8),
  sage int,
  sab float unique,
  primary key (sno, sab)
);
[INFO] Sql syntax parse ok!
minisql > show tables;
[INFO] Sql syntax parse ok!
+--------------------+
|Tables_in_db0       |
+--------------------+
|      student       |
|         t1         |
+--------------------+
Total 2 table(s)
minisql > drop t1;
Minisql parse error at line 1, col 7, message: syntax error
syntax error
minisql > drop table  t1;
[INFO] Sql syntax parse ok!
minisql > show tables;
[INFO] Sql syntax parse ok!
+--------------------+
|Tables_in_db0       |
+--------------------+
|      student       |
+--------------------+
Total 1 table(s)
```

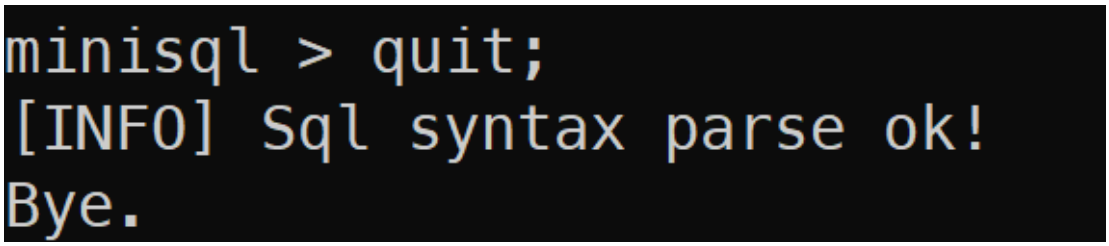
2.5.4 Index 相关语句

- 对于 `ExecuteCreateIndex` 操作，我们取出表，并根据语法树找出要建立索引的属性，最后调用 catalog 的 `CreateIndex` 即可。
- 对于 `ExecuteDropIndex` 操作，类似创建索引，改为 `DropIndex` 即可。
- 对于 `ExecuteShowIndexes` 操作，对于每个表我们都找他对应的 indexinfo, 并输出即可。

```
minisql > create index idx1 on t1(c);
[INFO] Sql syntax parse ok!
minisql > show indexes;
[INFO] Sql syntax parse ok!
+--------------------+
table_name: student
index_name: student_unique_index
key_name: sab
index_name: student_key_index
key_name: sno sab
+--------------------+
minisql > drop t1;
Minisql parse error at line 1, col 7, message: syntax error
syntax error
minisql > drop index t1;
[INFO] Sql syntax parse ok!
minisql > show indexes;
[INFO] Sql syntax parse ok!
+--------------------+
table_name: student
index_name: student_unique_index
key_name: sab
index_name: student_key_index
key_name: sno sab
+--------------------+
```

2.5.5 Quit 语句

这里直接返回 `DB_QUIT` 即可。注意到这里不能 `exit(0)` 杀死进程，因为这样突然死亡不会调用对象的析构函数。而是正常返回，这样在 main 里面就会跳出循环，正常结束程序。



详细测试可以参考小组报告的测试部分。