MiniSQL个人报告

3220103450 姜雨童

Date：2024.06.24

目录

一、完成模块说明

1. 参与小组讨论，对MiniSQL的实现进行分析
2. 负责模块四（Catalog Manager）和模块五（Executor）的实现

二、Catalog Manager模块

（一）Catalog Manager概述

Catalog Manager 负责管理和维护数据库的所有模式信息，包括：

- 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
- 表中每个字段的定义信息，包括字段类型、是否唯一等。
- 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

这些模式信息在被创建、修改和删除后还应被持久化到数据库文件中。此外，Catalog Manager还需要为上层的执行器Executor提供公共接口以供执行器获取目录信息并生成执行计划。

（二）模块调用

- 调用record manager中column，row和schema的序列化和反序列化函数
- 调用disk manager中的bufferpoolmanager，为表和索引分配单独的数据页
- 调用index manager中的BplusTreeIndex，为表中的列创建索引
- （本模块的CatalogManager类为Execute Engine提供函数接口，以实现minisql各模块功能）

（三）实现的接口

1. catalog.cpp

- `uint32_t CatalogMeta::GetSerializedSize() const`
- `CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager, LogManager *log_manager, bool init)`

- dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema, Txn *txn, TableInfo *&table_info)
- dberr_t CatalogManager::GetTable(const string &table_name, TableInfo *&table_info)
- dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables) const
- dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string &index_name, const std::vector<std::string> &index_keys, Txn *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) const
- dberr_t CatalogManager::GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const
- dberr_t CatalogManager::DropTable(const string &table_name)
- dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name)
- dberr_t CatalogManager::FlushCatalogMetaPage() const
- 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)
- dberr_t CatalogManager::GetTable(const table_id_t table_id, TableInfo *&table_info)

2. index.cpp

- uint32_t IndexMetadata::GetSerializedSize() const（模块内调用）

3. index.h

- void Init(IndexMetadata *meta_data, TableInfo *table_info, BufferPoolManager *buffer_pool_manager)（模块内调用）

（四）具体实现

可以把上述函数接口大致分为以下几类：

- 序列化和反序列化（包括初始化）
- Catalog Manager
- Create Table/Index
- Get Table(s)/Index/TableIndexes
- Load Table/Index
- Drop Table/Index
- Flush CatalogMetaPage

1. 序列化和反序列化

数据库中定义的表和索引在内存中以TableInfo和IndexInfo的形式表现。以IndexInfo为例，它包含了这个索引定义时的元信息meta_data_，该索引对应的表信息table_info_，该索引的模式信息key_schema_和索引操作对象index_。除元信息meta_data_外，其它的信息都是通过反序列化后的元信息生成的。

因此，为了能够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，我们需要为表和索引的元信息TableMetadata和IndexMetadata实现序列化和反序列化操作。（为了简便处理，在序列化时我们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。）CatalogMeta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（逻辑意义上），CATALOG_META_PAGE_ID默认值为0。

就具体实现而言，CatalogMeta类的序列化和反序列化函数都已给出，只需要实现几个获取序列化长度的函数以及Index的初始化（Init）。代码实现将统一附在报告最后。

2. Catalog Manager

CatalogManager能够在数据库实例（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，构建TableInfo和IndexInfo信息置于内存中。

根据数据库的调用情况，可以分为首次创建数据库和重新加载数据库两种情况。

首次调用时，只需要catalogmanager进行初始化。而重新加载时，则要通过反序列化得到数据库的元信息。接着通过对catalogmeta中信息的遍历，反序列化得到tablemetadata和indexmetadata，构建对应的tableinfo和indexinfo并把信息加入catalogmanager中，从而加载出完整的catalogmanager。

3. Create Table/Index

这两个构建函数的实现逻辑类似，这里以create table为例。首先需要利用bufferpoolmanager获取一个空闲页作为新建表的数据页（对应page_id），而调用catalogmeta中GetNextTableId得到的是新建表的id。然后调用初始化函数生成tableinfo，并把新建的表存到table列表中。最后更新catalogmetapage的信息，保证下一次调用时信息是正确的。（注意在实现create table的时候要进行深度拷贝，否则会报错。）

4. Get Table(s)/Index/TableIndexes

同样以get table为例，根据给出的tablename（或是tableid，这里实现了通过name和id两个不同标识查找表的函数）在table列表table_中找到对应tableinfo即可。而对于获得多个信息的get tables/tableindexes，只需创建向量表存入信息。

5. Load Table/Index

以load table为例。根据表id得到数据页id，然后通过反序列化得到tablemeta，后续把tablemeta中的数据，如tablename，tableinfo载入到数据库中。

6. Drop Table/Index

以drop table为例，根据表名，在catalogmanager类中table名称向量表和table向量表中删除该表的各类信息。同时要在catalogmeta中删除该表，并调用bufferpoolmanager类的deletepage函数对该表所在的页面进行回收，避免空间上的浪费。

7. Flush CatalogMetaPage

把目前catalogmetadata写入数据页，并即时将数据序列化到磁盘中。

三、Planner and executor模块

（一）Planner and executor概述

本实验主要包括Planner和Executor两部分。

Planner的主要功能是将解释器（Parser）生成的语法树，改写成数据库可以理解的数据结构。在这个过程中，我们会将所有sql语句中的标识符（Identifier）解析成没有歧义的实体，即各种C++的类，并通过Catalog Manager 提供的信息生成执行计划。（这部分没有代码需要实现，只需要了解parser的实现方式，便于后续编写代码。）

Executor遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor，随后调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，并将执行结果返回给上层模块。

（二）模块调用

- 调用catalog.cpp中CatalogManager类的各函数，以实现minisql各模块的功能

（三）实现的接口

1. execute_engine.cpp

这些函数均由类内函数ExecuteEngine::Execute（已给出实现）统一调用

- 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)

（四）具体实现

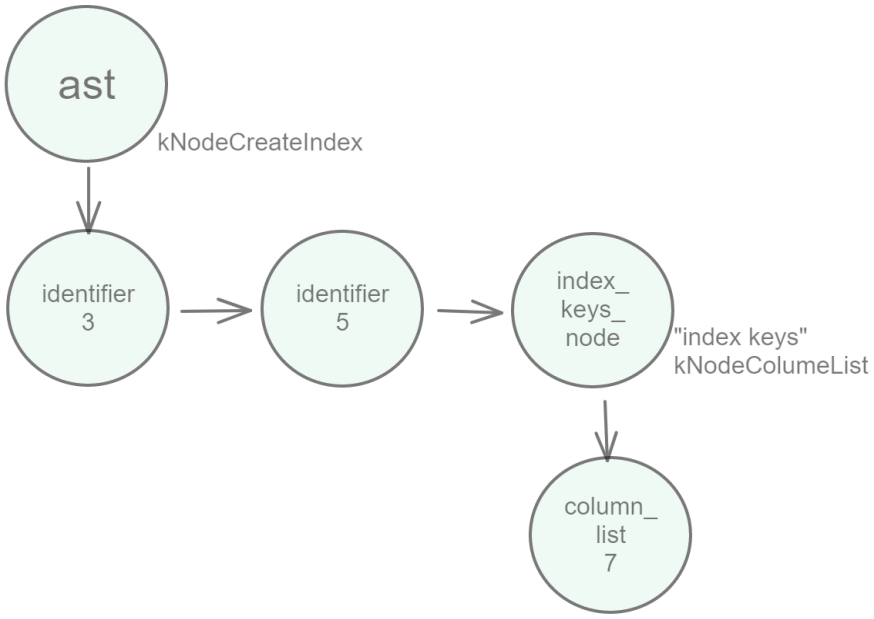
1. Parser语法分析

上述功能函数传入的参数均含语法树的结点pSyntaxNode ast，因此，在正式实现函数代码前，要先分析parser语法树的实现。

对于简单的sql语句，如drop table，并不怎么需要分析语法树，因此这里以create index为例。在src/include/parser/minisql.y下找到对应语法，如下：

```
sql_create_index:
  CREATE INDEX IDENTIFIER ON IDENTIFIER '(' column_list ')' {
    $$ = CreateSyntaxNode(kNodeCreateIndex, NULL);
    SyntaxNodeAddChildren($$, $3);
    SyntaxNodeAddChildren($$, $5);
    pSyntaxNode index_keys_node = CreateSyntaxNode(kNodeColumnList, "index keys");
    SyntaxNodeAddChildren(index_keys_node, $7);
    SyntaxNodeAddChildren($$, index_keys_node);
  }
  | CREATE INDEX IDENTIFIER ON IDENTIFIER '(' column_list ')' USING IDENTIFIER {
      $$ = CreateSyntaxNode(kNodeCreateIndex, NULL);
      SyntaxNodeAddChildren($$, $3);
      SyntaxNodeAddChildren($$, $5);
      pSyntaxNode index_keys_node = CreateSyntaxNode(kNodeColumnList, "index keys");
      SyntaxNodeAddChildren(index_keys_node, $7);
      SyntaxNodeAddChildren($$, index_keys_node);
      pSyntaxNode index_type_node = CreateSyntaxNode(kNodeIndexType, "index type");
      SyntaxNodeAddChildren(index_type_node, $10);
      SyntaxNodeAddChildren($$, index_type_node);
  }
  ;
```

不难分析出对应语法树1如下图：



据此可以通过函数传入的语法树结点获取sql语句蕴含的对应信息：

```
// analyze syntaxNode
auto index_name = ast->child_->val_; // see parser/minisql.y
auto table_name = ast->child_->next_->val_;
auto list = ast->child_->next_->next_;
```

2. ExecuteCreateTable

CreateTable, DropTable等函数实现较为类似，这里仅以CreateTabel函数为例。首先对语法树进行如上分析，获取sql指令中对应的信息。然后根据每一列给出的信息创建表内的属性。具体通过结点状态的不同（kNodeColumnList和kNodeColumnDefinition），先将primary key放入向量表中，再记录其他的key（如若遇到unique key也放入特定向量表中记录）。最后创建表和索引，并将相关信息记入。

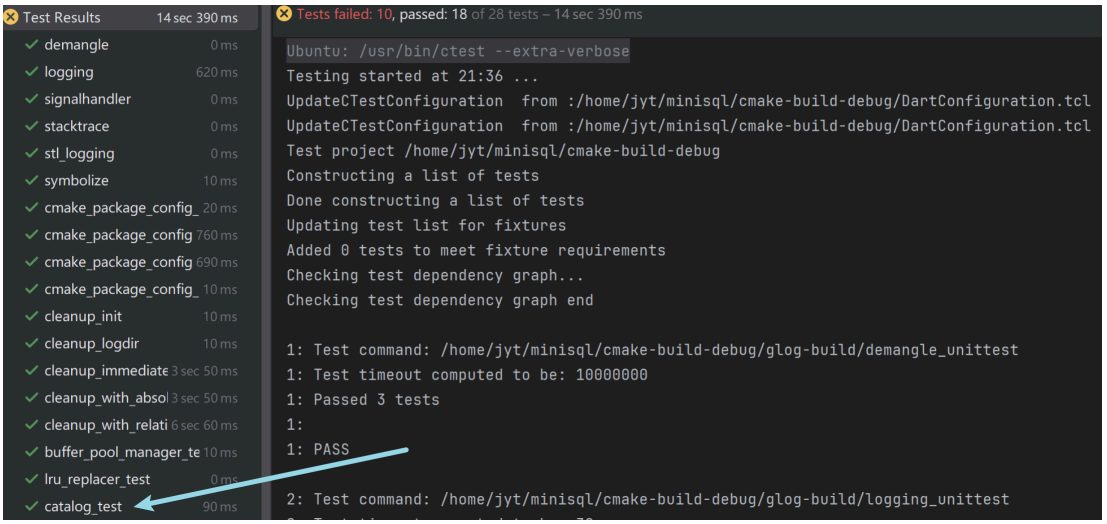
3. Executefile

该函数需要读取文件内的sql语句，因此不仅用到了c++的文件流读取函数，还用到了上述提到的语法分析，需要在Parser模块调用`yyparse()`（一个示例在`src/main.cpp`中）完成SQL语句解析，得到语法树的根结点`pSyntaxNode`。

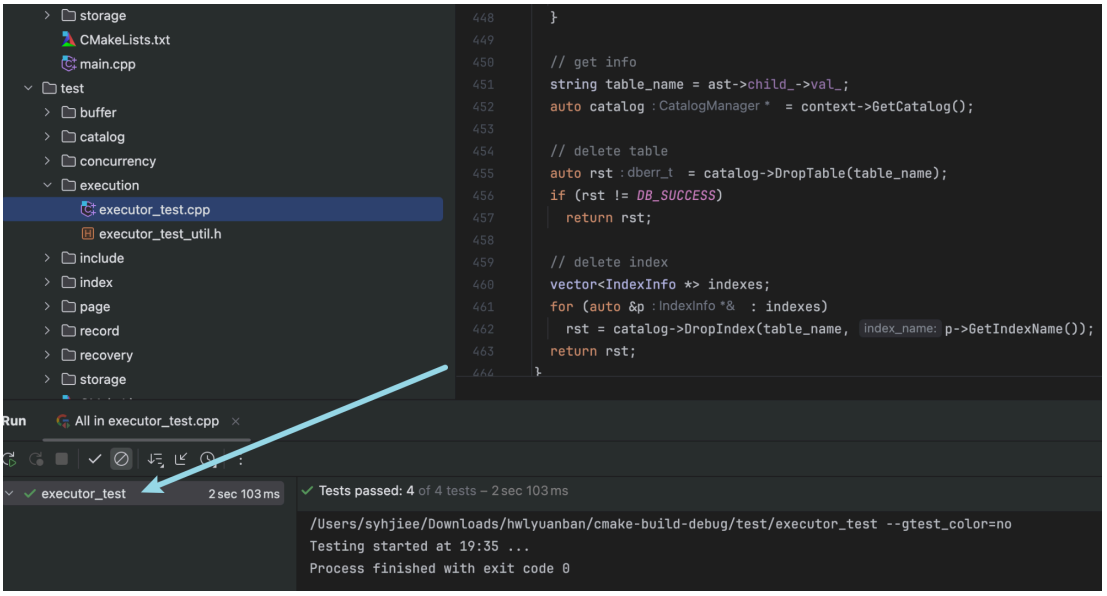
这个函数首先将文件内字符读到缓冲区中，以;为 分隔符形成一条或多条sql语句。然后利用parser进行语法解析，执行对应的操作（可以调用给出的Execute函数来进行），最后返回相关信息并关闭文件。

四、模块功能验证

（一）Catalog Manager



（二）Execute Engine



（说明：在设计实现minisql的过程中，电脑的环境配置出了一点问题，导致原来能构建的minisql现在跑不起来了。询问助教后还是没有找到解决办法，截至提交报告时，电脑环境还是跑不了。因此execute engine的测试是在室友电脑上验证的，显示路径和上面有所不同。可以通过对代码/注释的比对来验证跑的确实是我的execute engine模块。）

然而，在后续进行基本操作时，发现模块五的文件读入部分有问题，会出现“Segmentation fault”报错。截至提交报告时，仍没有发现是哪个地方出了bug。（截图如下，同样是借用了室友的电脑。）

```
antuer@HuWenLuo:/mnt/d/Desktop/Git_project/minisql/Minisql3rd/build/bin$ ./main
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      |
+----------+
minisql > use db0;
[INFO] Sql syntax parse ok!
Database changed
minisql > create table account(
  id int,
  name char(16),
  balance float,
  primary key(id)
);
[INFO] Sql syntax parse ok!
minisql > show tables;
[INFO] Sql syntax parse ok!
+---------------+
| Tables_in_db0 |
+---------------+
| account       |
+---------------+
minisql > execfile "../../sql_gen/account00.txt";
[INFO] Sql syntax parse ok!
----- Execfile -----
Segmentation fault
antuer@HuWenLuo:/mnt/d/Desktop/Git_project/minisql/Minisql3rd/build/bin$
```

五、代码实现

(一) Catalog Manager

1. index.h

```
1  void Init(IndexMetadata *meta_data, TableInfo *table_info,
   BufferPoolManager *buffer_pool_manager) {
2      // Step1: init index metadata and table info
3      // Step2: mapping index key to key schema
4      // Step3: call CreateIndex to create the index
5      meta_data_ = meta_data;
6      const Schema *table_schema = table_info->GetSchema();
7      key_schema_ = Schema::ShallowCopySchema(table_schema, meta_data-
   >GetKeyMapping());
8      index_ = CreateIndex(buffer_pool_manager, "bptree");
9    }
```

2. index.cpp

```
1  uint32_t IndexMetadata::GetSerializedSize() const {
2    return  sizeof(uint32_t)*5 + index_name_.length() +
   key_map_.size()*sizeof(uint32_t);
3  }
```

3. catalog.cpp

```
1  /**
2   * TODO: Student Implement FIN OK
3   */
4  uint32_t CatalogMeta::GetSerializedSize() const {
5    return sizeof(uint32_t) * (3 + table_meta_pages_.size() * 2 +
   index_meta_pages_.size() * 2);
6  }
7
8  /**
9   * TODO: Student Implement FIN OK
10  */
11 CatalogManager::CatalogManager(BufferPoolManager
   *buffer_pool_manager, LockManager *lock_manager,
12                                 LogManager *log_manager, bool init)
13     : buffer_pool_manager_(buffer_pool_manager),
   lock_manager_(lock_manager), log_manager_(log_manager) {
14   if (init) {
15     catalog_meta_ = CatalogMeta::NewInstance();
16     FlushCatalogMetaPage();
17   } else {
18     auto meta_page = buffer_pool_manager_-
   >FetchPage(CATALOG_META_PAGE_ID);
19     catalog_meta_ = CatalogMeta::DeserializeFrom(meta_page-
   >GetData());
20
21     // get info of tables
22     for (auto &p : catalog_meta_->table_meta_pages_) { //
   p.first:table_id; p.second:page_id
23       TableMetadata *table_meta;
24       TableMetadata::DeserializeFrom(buffer_pool_manager_-
   >FetchPage(p.second)->GetData(), table_meta);
25       auto table_heap = TableHeap::Create(buffer_pool_manager_,
   table_meta->GetFirstPageId(),
26                                           table_meta->GetSchema(),
   log_manager_, lock_manager_);
27       auto table_info = TableInfo::Create();
28       table_info->Init(table_meta, table_heap);
29       tables_[p.first] = table_info; // store table info
```

```
30        table_names_[table_meta->GetTableName()] = p.first;
31      }
32
33      // get info of indexes
34      for (auto &p : catalog_meta_->index_meta_pages_) { //
p.first:index_id; p.second:page_id
35        IndexMetadata *index_meta;
36        IndexMetadata::DeserializeFrom(buffer_pool_manager_-
>FetchPage(p.second)->GetData(), index_meta);
37        auto table_id = index_meta->GetTableId();
38        auto table_info = tables_[table_id];
39        auto table_name = table_info->GetTableName();
40
41        auto index_info = IndexInfo::Create();
42        index_info->Init(index_meta, table_info,
buffer_pool_manager_);
43
44        indexes_[p.first] = index_info; // store index info
45        if (index_names_.find(table_name) == index_names_.end())
46          index_names_.emplace(table_name, unordered_map<string,
index_id_t>());
47        index_names_[table_name][index_meta->GetIndexName()] =
p.first; //*
48      }
49    }
50  }
51
52  /**
53   * TODO: Student Implement FIN OK
54   */
55  dberr_t CatalogManager::CreateTable(const string &table_name,
TableSchema *schema, Txn *txn, TableInfo *&table_info) {
56    if (table_names_.find(table_name) != table_names_.end())
57      return DB_TABLE_ALREADY_EXIST;
58
59    IndexSchema *schema_copy = Schema::DeepCopySchema(schema); //*
debug
60
61    // new id of table and page
62    auto table_id = catalog_meta_->GetNextTableId(); // get new id
of table
63    table_names_[table_name] = table_id;
64    page_id_t page_id;
65    auto new_table_page = buffer_pool_manager_->NewPage(page_id); //
get new id of page
66    catalog_meta_->table_meta_pages_[table_id] = page_id; // set new
id of table and page
67
```

```
68    // new table info
69    auto table_heap = TableHeap::Create(buffer_pool_manager_,
      schema_copy, txn, log_manager_, lock_manager_);
70    auto table_meta = TableMetadata::Create(table_id, table_name,
      page_id, schema_copy);
71    table_meta->SerializeTo(new_table_page->GetData());
72    table_info = TableInfo::Create();
73    table_info->Init(table_meta, table_heap);
74    tables_[table_id] = table_info; // set new info of table
75    buffer_pool_manager_->UnpinPage(page_id, true); // unpin page in
      buffer
76
77    // update catalog_meta_page
78    auto meta_page = buffer_pool_manager_-
      >FetchPage(CATALOG_META_PAGE_ID);
79    catalog_meta_->SerializeTo(meta_page->GetData());
80    buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
81
82    return DB_SUCCESS;
83  }
84
85  /**
86   * TODO: Student Implement FIN OK
87   */
88  dberr_t CatalogManager::GetTable(const string &table_name,
      TableInfo *&table_info) {
89    if (table_names_.find(table_name) == table_names_.end())
90      return DB_TABLE_NOT_EXIST;
91    auto table_id = table_names_[table_name];
92    table_info = tables_[table_id];
93    return  DB_SUCCESS;
94  }
95
96  /**
97   * TODO: Student Implement FIN OK
98   */
99  dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables)
      const {
100   if (tables_.empty())
101     return DB_TABLE_NOT_EXIST;
102   for(auto &p : tables_)
103     tables.emplace_back(p.second);
104   return  DB_SUCCESS;
105 }
106
107 /**
108  * TODO: Student Implement FIN OK
109  */
```

```
110 dberr_t CatalogManager::CreateIndex(const std::string &table_name,
    const string &index_name,
111                                    const std::vector<std::string>
    &index_keys, Txn *txn, IndexInfo *&index_info,
112                                    const string &index_type) {
113   if (table_names_.find(table_name) == table_names_.end())
114     return DB_TABLE_NOT_EXIST;
115   if (index_names_[table_name].find(index_name) !=
    index_names_[table_name].end())
116     return DB_INDEX_ALREADY_EXIST;
117
118   // check whether keys are in the column
119   auto table_id = table_names_[table_name];
120   auto table_info = tables_.at(table_id);
121   auto schema = table_info->GetSchema();
122   std::vector<uint32_t> key_map; // record index_id
123   for (auto &p : index_keys) {
124     index_id_t index_id;
125     if (schema->GetColumnIndex(p, index_id) ==
    DB_COLUMN_NAME_NOT_EXIST)
126       return DB_COLUMN_NAME_NOT_EXIST;
127     key_map.emplace_back(index_id);
128   }
129
130   // new id of index and page
131   auto index_id = catalog_meta_->GetNextIndexId();
132   index_names_[table_name][index_name] = index_id; //* debug .at()
    is wrong
133   page_id_t page_id;
134   auto new_index_page = buffer_pool_manager_->NewPage(page_id); //
    get new id of page
135   catalog_meta_->index_meta_pages_[index_id] = page_id; // set new
    id of index and page
136
137   // new index info
138   auto index_meta = IndexMetadata::Create(index_id, index_name,
    table_id, key_map);
139   index_meta->SerializeTo(new_index_page->GetData());
140   index_info = IndexInfo::Create();
141   index_info->Init(index_meta, table_info, buffer_pool_manager_);
142   indexes_[index_id] = index_info; // set new info of index
143   buffer_pool_manager_->UnpinPage(page_id, true); // unpin page in
    buffer
144
145   // update catalog_meta_page
146   auto meta_page = buffer_pool_manager_-
    >FetchPage(CATALOG_META_PAGE_ID);
147   catalog_meta_->SerializeTo(meta_page->GetData());
```

```
148   buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
149
150   return DB_SUCCESS;
151 }
152
153 /**
154  * TODO: Student Implement FIN OK
155  */
156 dberr_t CatalogManager::GetIndex(const std::string &table_name,
    const std::string &index_name,
157                                  IndexInfo *&index_info) const {
158   if (table_names_.find(table_name) == table_names_.end())
159     return DB_TABLE_NOT_EXIST;
160   if (index_names_.at(table_name).find(index_name) ==
    index_names_.at(table_name).end()) //* [table_name] X
161     return DB_INDEX_NOT_FOUND;
162
163   auto index_id = index_names_.at(table_name).at(index_name);
164   index_info = indexes_.at(index_id);
165   return DB_SUCCESS;
166 }
167
168 /**
169  * TODO: Student Implement FIN OK
170  */
171 dberr_t CatalogManager::GetTableIndexes(const std::string
    &table_name, std::vector<IndexInfo *> &indexes) const {
172   if (table_names_.find(table_name) == table_names_.end())
173     return DB_TABLE_NOT_EXIST;
174
175   auto index_name = index_names_.at(table_name);
176   for (auto &p : indexes_) {
177     if (table_name == p.second->GetIndexName())
178       indexes.emplace_back(p.second);
179   }
180   return DB_SUCCESS;
181 }
182
183 /**
184  * TODO: Student Implement FIN OK
185  */
186 dberr_t CatalogManager::DropTable(const string &table_name) {
187   if (table_names_.find(table_name) == table_names_.end())
188     return DB_TABLE_NOT_EXIST;
189
190   auto table_id = table_names_[table_name];
191   auto page_id = catalog_meta_->table_meta_pages_[table_id];
192   table_names_.erase(table_name); // delete table info
```

```
193   tables_.erase(table_id);
194   catalog_meta_->table_meta_pages_.erase(table_id);
195   buffer_pool_manager_->DeletePage(page_id); // delete page info
196   return DB_SUCCESS;
197 }
198 
199 /**
200  * TODO: Student Implement FIN OK
201  */
202 dberr_t CatalogManager::DropIndex(const string &table_name, const
    string &index_name) {
203   if (table_names_.find(table_name) == table_names_.end())
204     return DB_TABLE_NOT_EXIST;
205   if (index_names_[table_name].find(index_name) ==
    index_names_.at(table_name).end())
206     return DB_INDEX_NOT_FOUND;
207 
208   auto index_id = index_names_.at(table_name).at(index_name);
209   auto page_id = catalog_meta_->index_meta_pages_[index_id];
210   index_names_.at(table_name).erase(index_name); // delete index
    info
211   indexes_.erase(index_id);
212   catalog_meta_->index_meta_pages_.erase(index_id);
213   buffer_pool_manager_->DeletePage(page_id); // delete page info
214   return DB_SUCCESS;
215 }
216 
217 /**
218  * TODO: Student Implement FIN OK
219  */
220 dberr_t CatalogManager::FlushCatalogMetaPage() const {
221   auto page_meta = buffer_pool_manager_-
    >FetchPage(CATALOG_META_PAGE_ID);
222   catalog_meta_->SerializeTo(page_meta->GetData());
223   if(!buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID))
224     return DB_FAILED;
225   return  DB_SUCCESS;
226 }
227 
228 /**
229  * TODO: Student Implement FIN OK
230  */
231 dberr_t CatalogManager::LoadTable(const table_id_t table_id, const
    page_id_t page_id) {
232   if (tables_.find(table_id) != tables_.end())
233     return DB_TABLE_ALREADY_EXIST;
234 
235   // get data info from page
```

```
236   catalog_meta_->table_meta_pages_.at(table_id) = page_id; // set
page id
237   TableMetadata *table_meta;
238   TableMetadata::DeserializeFrom(buffer_pool_manager_-
>FetchPage(page_id)->GetData(), table_meta);
239   table_names_.at(table_meta->GetTableName()) = table_id; // store
table name and id
240
241   // load table info
242   auto table_heap = TableHeap::Create(buffer_pool_manager_,
table_meta->GetFirstPageId(),
243                                       table_meta->GetSchema(),
log_manager_, lock_manager_);
244   auto table_info = TableInfo::Create();
245   table_info->Init(table_meta, table_heap);
246   tables_[table_id] = table_info; // store table info
247
248   return DB_SUCCESS;
249 }
250
251 /**
252  * TODO: Student Implement FIN OK
253  */
254 dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const
page_id_t page_id) {
255   if (indexes_.find(index_id) != indexes_.end())
256     return DB_INDEX_ALREADY_EXIST;
257
258   // get data info from page
259   catalog_meta_->index_meta_pages_.at(index_id) = page_id; // set
page id
260   IndexMetadata *index_meta;
261   IndexMetadata::DeserializeFrom(buffer_pool_manager_-
>FetchPage(page_id)->GetData(), index_meta);
262   auto table_id = index_meta->GetTableId();
263   auto table_name = tables_[table_id]->GetTableName();
264   index_names_.at(table_name).at(index_meta->GetIndexName()) =
index_id; // store index name and id
265
266   // load index info
267   auto index_info = IndexInfo::Create();
268   index_info->Init(index_meta, tables_[table_id],
buffer_pool_manager_);
269   indexes_[index_id] = index_info; // store index info
270
271   return DB_SUCCESS;
272 }
273
```

```
274 /**
275  * TODO: Student Implement FIN OK
276  */
277 dberr_t CatalogManager::GetTable(const table_id_t table_id,
    TableInfo *&table_info) {
278   if (tables_.find(table_id) == tables_.end())
279     return DB_TABLE_NOT_EXIST;
280   table_info = tables_[table_id];
281   return DB_SUCCESS;
282 }
```

（二）Planner and executor

1. execute_engine.cpp

```
1  /**
2   * TODO: Student Implement FIN
3   */
4  dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast,
   ExecuteContext *context) {
5  #ifdef ENABLE_EXECUTE_DEBUG
6    LOG(INFO) << "ExecuteCreateTable" << std::endl;
7  #endif
8    if (current_db_.empty()) {
9      cout << "No database selected." << endl;
10     return DB_FAILED;
11   }
12
13   // analyze syntaxNode: parser/minisql.y
14   string table_name = ast->child_->val_;
15   if (ast->child_->type_ != kNodeIdentifier)
16     return DB_FAILED;
17   auto list_head = ast->child_->next_;
18   if (list_head->type_ != kNodeColumnDefinitionList)
19     return DB_FAILED;
20   auto list_node = list_head->child_; // first node now
21
22   vector<string> primary_keys;
23   vector<string> unique_keys;
24   uint32_t index = 0;
25   vector<Column *> columns;
26
27   // get primary key
28   for (; list_node != nullptr; list_node = list_node->next_) {
29     if (list_node->type_ == kNodeColumnList && string(list_node-
   >val_) == "primary keys") {
30       for (auto p = list_node->child_; p != nullptr; p = p->next_)
31         primary_keys.emplace_back(string(p->val_));
```

```
    }
  }

  // get other keys
  for (list_node = list_head->child_; list_node != nullptr;
list_node = list_node->next_) {
    if (list_node->type_ != kNodeColumnDefinition)
      continue;

    //- parser analyze
    bool is_unique = (list_node->val_ != nullptr &&
string(list_node->val_) == "unique");
    auto deeper_node = list_node->child_;
    string column_name = deeper_node->val_;
    string column_type = deeper_node->next_->val_;
    Column *column;

    //- create column ( defined by column type )
    if (column_type == "int")
      column = new Column(column_name, kTypeInt, index++, true,
is_unique);
    else if (column_type == "float")
      column = new Column(column_name, kTypeFloat, index++, true,
is_unique);
    else if (column_type == "char") {
      string length = deeper_node->next_->child_->val_;
      for (auto &p : length)
        if (!isdigit(p))
          return DB_FAILED; // exam the number is integer or not
      if (stoi(length) < 0)
        return  DB_FAILED; // exam the number is positive or not
      column = new Column(column_name, kTypeChar, stoi(length),
index++, true, is_unique);
    }

    // add columm
    columns.emplace_back(column);
    if (is_unique)
      unique_keys.emplace_back(column_name);
  }

  // create table
  auto catalog = context->GetCatalog();
  auto schema = new Schema(columns);
  TableInfo* table_info;
  auto rst = catalog->CreateTable(table_name, schema, context-
>GetTransaction(), table_info);
  if (rst != DB_SUCCESS)
```

```
74      return rst;
75
76    // create index
77    IndexInfo *index_info;
78    for (auto &p : unique_keys) { // unique key
79      string index_name = "UNIQUE_";
80      index_name += p + "_ON_" + table_name;
81      vector<string> index_keys;
82      index_keys.emplace_back(p);
83      rst = catalog->CreateIndex(table_name, index_name, index_keys,
context->GetTransaction(), index_info, "btree");
84      if (rst != DB_SUCCESS)
85        return rst;
86    }
87    if (primary_keys.size() > 0) {
88      string index_name = "AUTO_CREATED_INDEX_OF_";
89      for (auto &p : primary_keys)
90        index_name += p + "_";
91      index_name += "ON_" + table_name;
92      catalog->CreateIndex(table_name, index_name, primary_keys,
context->GetTransaction(), index_info, "btree");
93    }
94    return rst;
95  }
96
97  /**
98   * TODO: Student Implement FIN
99   */
100 dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast,
ExecuteContext *context) {
101 #ifdef ENABLE_EXECUTE_DEBUG
102   LOG(INFO) << "ExecuteDropTable" << std::endl;
103 #endif
104   if (current_db_.empty()) {
105     cout << "No database selected." << endl;
106     return DB_FAILED;
107   }
108
109   // get info
110   string table_name = ast->child_->val_;
111   auto catalog = context->GetCatalog();
112
113   // delete table
114   auto rst = catalog->DropTable(table_name);
115   if (rst != DB_SUCCESS)
116     return rst;
117
118   // delete index
```

```
119   vector<IndexInfo *> indexes;
120   for (auto &p : indexes)
121     rst = catalog->DropIndex(table_name, p->GetIndexName());
122   return rst;
123 }
124 
125 /**
126  * TODO: Student Implement FIN
127  */
128 dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast,
    ExecuteContext *context) {
129 #ifdef ENABLE_EXECUTE_DEBUG
130   LOG(INFO) << "ExecuteShowIndexes" << std::endl;
131 #endif
132   if (current_db_.empty()) {
133     cout << "No database selected." << endl;
134     return DB_FAILED;
135   }
136 
137   // get tables' info
138   auto catalog = context->GetCatalog();
139   vector<TableInfo *> tables;
140   catalog->GetTables(tables);
141 
142   // show indexes
143   cout << "----- Show Index -----" << endl;
144   int counter = 1;
145   for (auto &p : tables) {
146     vector<IndexInfo *> indexes;
147     catalog->GetTableIndexes(p->GetTableName(), indexes);
148     cout << "\tTable: " << p->GetTableName() << endl;
149     for (auto &i : indexes) {
150       cout << "\t\tIndex " << (counter++) << ": " << i-
    >GetIndexName() << endl;
151     }
152   }
153   cout << "Totally" << counter << "index(es) listed." << endl;
154   return DB_SUCCESS;
155 }
156 
157 /**
158  * TODO: Student Implement FIN
159  */
160 dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast,
    ExecuteContext *context) {
161 #ifdef ENABLE_EXECUTE_DEBUG
162   LOG(INFO) << "ExecuteCreateIndex" << std::endl;
163 #endif
```

```
  if (current_db_.empty()) {
    cout << "No database selected." << endl;
    return DB_FAILED;
  }

  // analyze syntaxNode
  auto index_name = ast->child_->val_; // see parser/minisql.y
  auto table_name = ast->child_->next_->val_;
  auto list = ast->child_->next_->next_;
  if (list->type_ != kNodeColumnList)
    return DB_FAILED;

  // get index info and create index
  auto catalog = context->GetCatalog();
  vector<string> keys;
  for (auto key = list->child_; key != nullptr; key = key->next_)
{
    keys.emplace_back(key->val_);
  }
  IndexInfo *index_info;
  auto rst = catalog->CreateIndex(table_name, index_name, keys,
context->GetTransaction(), index_info, "btree");
  return rst;
}

/**
 * TODO: Student Implement FIN
 */
dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast,
ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropIndex" << std::endl;
#endif
  if (current_db_.empty()) {
    cout << "No database selected." << endl;
    return DB_FAILED;
  }

  // get info
  string index_name = ast->child_->val_;
  auto catalog = context->GetCatalog();
  vector<TableInfo *> tables;
  catalog->GetTables(tables);

  // drop index
  for (auto &p : tables)
    if (catalog->DropIndex(p->GetTableName(), index_name) !=
DB_SUCCESS)
```

```
208       return DB_FAILED;
209   return DB_SUCCESS;
210 }
211
212 dberr_t ExecuteEngine::ExecuteTrxBegin(pSyntaxNode ast,
    ExecuteContext *context) {
213 #ifdef ENABLE_EXECUTE_DEBUG
214   LOG(INFO) << "ExecuteTrxBegin" << std::endl;
215 #endif
216   return DB_FAILED;
217 }
218
219 dberr_t ExecuteEngine::ExecuteTrxCommit(pSyntaxNode ast,
    ExecuteContext *context) {
220 #ifdef ENABLE_EXECUTE_DEBUG
221   LOG(INFO) << "ExecuteTrxCommit" << std::endl;
222 #endif
223   return DB_FAILED;
224 }
225
226 dberr_t ExecuteEngine::ExecuteTrxRollback(pSyntaxNode ast,
    ExecuteContext *context) {
227 #ifdef ENABLE_EXECUTE_DEBUG
228   LOG(INFO) << "ExecuteTrxRollback" << std::endl;
229 #endif
230   return DB_FAILED;
231 }
232
233 /**
234  * TODO: Student Implement FIN
235  */
236
237 dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast,
    ExecuteContext *context) {
238 #ifdef ENABLE_EXECUTE_DEBUG
239   LOG(INFO) << "ExecuteExecfile" << std::endl;
240 #endif
241   auto file_name = ast->child_->val_;
242   fstream file(file_name);
243   if (!file.is_open()) {
244     cout << "Fail to open file '" << file_name << "'." << endl;
245     return DB_FAILED;
246   }
247
248   // buffer for cmd
249   const int buffer_size = 1024;
250   char *cmd = new char[buffer_size];
251
```

```
252   cout << "----- Execfile -----" << endl;
253   auto start_time = std::chrono::system_clock::now(); //*
254   while (!file.eof()) {
255     // load cmd
256     memset(cmd, 0, buffer_size);
257     int counter = 0;
258     char c;
259     while (!file.eof() && (c == file.get()) != ';')
260       cmd[counter++] = c;
261     if (file.eof()) break; // has scanned all file
262     cmd[counter] = ';'; // end of one sql
263 
264     // translate sql via parser
265     YY_BUFFER_STATE bp = yy_scan_string(cmd);
266     MinisqlParserInit();
267     yyparse();
268     if (MinisqlParserGetError())
269       printf("%s\n", MinisqlParserGetErrorMessage()); // error
    message
270 
271     // handle result
272     auto rst = Execute(MinisqlGetParserRootNode());
273 
274     // parser over, clean memory
275     MinisqlParserFinish();
276     yy_delete_buffer(bp);
277     yylex_destroy();
278 
279     // handle execute message
280     ExecuteInformation(rst);
281   }
282   auto stop_time = std::chrono::system_clock::now();
283 
284   // calculate execution time
285   auto time =
    double((std::chrono::duration_cast<std::chrono::milliseconds>
    (stop_time - start_time)).count());
286   cout << "Execfile finished in " << time << "ms." << endl;
287   return DB_SUCCESS;
288 }
289 
290 /**
291  * TODO: Student Implement FIN
292  */
293 dberr_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast, ExecuteContext
    *context) {
294 #ifdef ENABLE_EXECUTE_DEBUG
295   LOG(INFO) << "ExecuteQuit" << std::endl;
```

```
296  #endif
297    current_db_ = "";
298    cout << "Bye." << endl;
299    return DB_SUCCESS;
300  }
```