浙江大学

《数据库系统》
实验报告

模块名称：	5 Planner and Executor
姓　　名：	王晓宇
学　　号：	3220104364
电子邮箱：	3220104364@zju.edu.cn
联系电话：	19550222634
授课教师：	孙建伶
助　　教：	聂俊哲/石宇新

2024 年 6 月 8 日

实验名称 5 Planner and Executor

实验名称 5 Planner and Executor

模块概述

> *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。*
>
> *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。*

实验环境

> *1. 操作系统：Windows 11 23H2*
>
> *2. 数据库管理系统：MiniSQL*
>
> *3. 工具：CLion 2023.3.3*
>
> *4. 工程管理：Git/GitHub*

设计流程

模块运行流程：

在Parser模块调用`yyparse()`完成SQL语句解析后，将会得到语法树的根结点`pSyntaxNode`。将语法树根结点传入`ExecuteEngine`后，`ExecuteEngine`将会根据语法树根结点的类型，决定是否需要传入`Planner`生成执行计划。

- 对于简单的语句例如`show databases`，`drop table` 等，生成的语法树也非常简单。以`show databases`为例，对应的语法树只有单节点`kNodeShowDB`，表示展示所有数据库。此时无需传入`Planner`生成执行计划，我们直接调用对应的执行函数执行即可。

- 对于复杂的语句，生成的语法树需传入`Planner`生成执行计划，并交由`Executor`进行执行。`Planner`需要先遍历语法树，调用`Catalog Manager`检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与`column`类型对应等等，随后将这些词语抽象成相应的表达式，即可以理解的各种 c++ 类。解析完成后，`Planner`根据改写语法树后生成的可以理解的`Statement`结构，生成对应的`Plannode`，并将`Plannode`交由`executor`进行执行。

  以一条select的sql为例，生成的对应语法树如下：其中`kNodeSelect`标识了语句的类型，`kNodeColumnList`标识了有哪些列，`kNodeIdentifier`标识了是哪张表，`kNodeConditions`标识了`where`语句的条件。

Parser解析语法生成语法树

在本实验中，设计好MiniSQL中的Parser模块已被设计好，与Parser模块的相关代码如下：

- `src/include/parser/minisql.l`：SQL的词法分析规则；
- `src/include/parser/minisql.y`：SQL的文法分析规则；
- `src/include/parser/minisql_lex.h`：`flex(lex)`根据词法规则自动生成的代码；
- `src/include/parser/minisql_yacc.h`：`bison(yacc)`根据文法规则自动生成的代码；
- `src/include/parser/parser.h`：Parser模块相关的函数定义，供词法分析器和语法分析器调用存储分析结果，同时可供执行器调用获取语法树根结点；
- `src/include/parser/syntax_tree.h`：语法树相关定义，语法树各个结点的类型同样在`SyntaxNodeType`中被定义。

语法树节点结构体

```
/**
 * 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;
```

数据类型	变量名称	代表含义
`int`	`id_`	生成可视化语法树时，节点赋值id
`SyntaxNodeType`	`type_`	语法树结点类型，本质是枚举类型，例：`kNodeCreateDB`,`kNodeDropDB`，代表不同节点的语法类型
`int`	`line_no_`	该结点在规约（*reduce*，编译原理中的术语）的所在行数
`int`	`col_no_`	该结点在规约（*reduce*，编译原理中的术语）的所在列数
`SyntaxNode *`	`child_`	子节点
`SyntaxNode *`	`next_`	兄弟节点
`char *`	`val_`	用作一些额外信息的存储（如在`kNodeString`类型的结点中，`val_`将用于存储该字符串的字面量）

语法树构建方法及可视化

在语句执行时只需要调用：

```
YY_BUFFER_STATE bp = yy_scan_string(cmd);//read command in string
yy_switch_to_buffer(bp);//move command to buffer
MinisqlParserInit();// init parser module
yyparse();// parse
```

即可生成对应语法树

具体查看对应语法树可以调用：

```
//initial the printtree
SyntaxTreePrinter printer(MinisqlGetParserRootNode());
std::string file = "Syntax_tree_wxy.txt";
std::ofstream outFile(file);
//print it to file
printer.PrintTree(outFile);
outFile.close();
```

之后在`/bin`目录下便可以看到生成的语法树DOT文件，利用可视化网站即可生成对应语法树结构，便于构建`Executor`算子。

具体实现函数的语法树图片在后文详细展开，此处不再赘述。

Planner生成查询计划

解析完成后，`Planner`根据改写语法树后生成的可以理解的`Statement`结构，生成对应的`Plannode`，并将`Plannode`交由`executor`进行执行。该模块相关的代码如下：

- `src/include/planner/statement/abstract_statement.h`
- `src/include/planner/statement/select_statement.h`
- `src/include/planner/statement/insert_statement.h`
- `src/include/planner/statement/delete_statement.h`
- `src/include/planner/statement/update_statement.h`

`Statement`中的函数`SyntaxTree2Statement`将解析语法树，并将各种`Identifier`转化为可以理解的表达式，存储在Statement结构中。`Planner`再根据`Statement`，生成对应的执行计划，相关代码如下：

- `src/include/executor/plans/abstract_plan.h`

- `src/include/executor/plans/delete_plan.h`
- `src/include/executor/plans/insert_plan.h`
- `src/include/executor/plans/seq_scan_plan.h`
- `src/include/executor/plans/update_plan.h`
- `src/include/executor/plans/value_plan.h`

Tips：在成熟的数据库中，Planner一般和优化器Optimizer一起，称为查询优化器。通常，查询优化器会通过如下三个典型组件协同来完成查询优化。优化后，能将原本根据语法树直接生成的查询计划改写成效率更高的查询计划，例如经典的join order问题。

- **Plan space enumeration:** 根据一系列的等价变换规则，生成与查询等价的多个执行计划；
- **cardinality estimation:** 根据查询表的分布情况，估计查询执行过程中的数据量/数据分布等；
- **cost model:** 根据执行计划以及数据库内部的状态，计算按照各个执行计划执行所需要的代价。

Executor执行查询计划

查询计划分配

在拿到 `Planner` 生成的具体的查询计划后，就可以生成真正执行查询计划的一系列算子了。生成算子的步骤很简单，遍历查询计划树，将树上的 `PlanNode` 替换成对应的 `Executor`。本实验采用`Iterator Model`来进行迭代。

`Iterator Model`，即经典的火山模型。执行引擎会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有`Init()` 和 `Next()` 两个方法。`Init()`对算子进行初始化工作。`Next()` 则是向下层算子请求下一条数据。当 `Next()` 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。

1. 该方法的优点是其计算模型简单直接，通过把不同物理算子抽象成一个个迭代器。每一个算子只关心自己内部的逻辑即可，让各个算子之间的耦合性降低，从而比较容易写出一个逻辑正确的执行引擎。
2. 缺点是火山模型一次调用请求一条数据，占用内存较小，但函数调用开销大，特别是虚函数调用造成 cache miss 等问题。同时，逐行地获取数据会带来过多的 I/O，对缓存也不友好。

在本次任务中，我们将实现5个算子，分别是`select`，`Index Select`，`insert`，`update`，`delete`。 对于每个算子，都实现了 `Init`和 `Next` 方法。 `Init` 方法初始化运算符的内部状态，`Next` 方法提供迭代器接口，并在每次调用时返回一个元组和相应的 RID。对于每个算子，我们假设它在单线程上下文中运行，并不需要考虑多线程的情况。每个算子都可以通过访问 `ExecuteContext`来实现表的修改，例如插入、更新和删除。 为了使表索引与底层表保持一致，插入删除时还需要更新索引。

注意：对于简单的SQL命令，对应的语法树较为简单，可以直接通过特定算子执行，不必生成查询计划，如`src/include/executor/execute_engine.h`中的创建删除查询数据库、数据表、索引等函数。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。

工程代码实现

execute_engine.h变量介绍

```
//All the active databases in system
std::unordered_map<std::string, DBStorageEngine *> dbs_;
//Current using database
std::string current_db_;
```

数据类型	变量名称	代表含义
`unordered_map<std::string, DBStorageEngine *>`	`dbs_`	通过`map`构建数据库名称，和`DB`结构体相关联
`string`	`current_db_`	指向现在使用的数据库，通过`use <database_name>`来进行更改

简单SQL命令对应的Executor函数

ExecuteCreateDatabase()

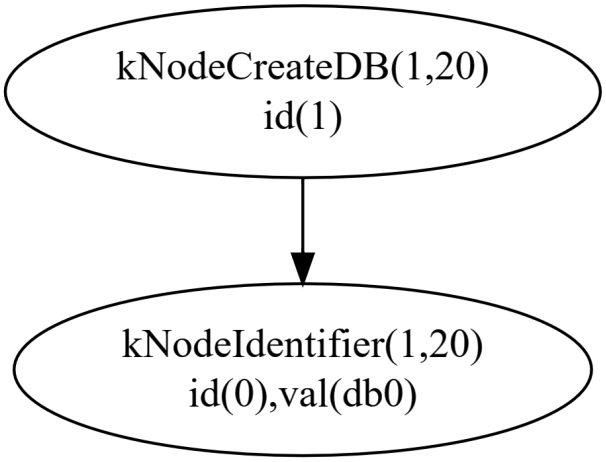
函数声明：

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

SQL语句声明：

```
create database db0;
```

语法树示意图：



- `CreateDatabase`的语法树有两个节点，第一个节点是标识，第二个节点带有创建数据库名字信息
- 新建`Database`的名字来源于语法树根节点的儿子的`val_`
- 查找`dbs_`中是否有该数据库：

  ```
  dbs_.find(db_name) != dbs_.end()
  ```

  `find`方法可以根据名字顺序去找到对应元素的迭代指针，`end`方法返回数据库末尾指针，如果遍历到最后，则没有找到该名字的数据库。
- `insert`方法可以根据名字插入

  ```
  dbs_.insert(make_pair(db_name, new
  DBStorageEngine(db_name, true)));
  ```

函数实现：

```
dberr_t ExecuteEngine::ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context) {
  string db_name = ast->child_->val_;
  if (dbs_.find(db_name) != dbs_.end()) {
    return DB_ALREADY_EXIST;
  }
  dbs_.insert(make_pair(db_name, new DBStorageEngine(db_name, true)));
  return DB_SUCCESS;
}
```

1. 通过寻找语法树节点的`val_`值定义所创建数据库名称
2. 判断所存数据库中是否存在与所建立数据库名称一致的数据库
3. 通过`insert`函数，并定义`pair`键值对来插入新建数据库

ExecuteDropDatabase()

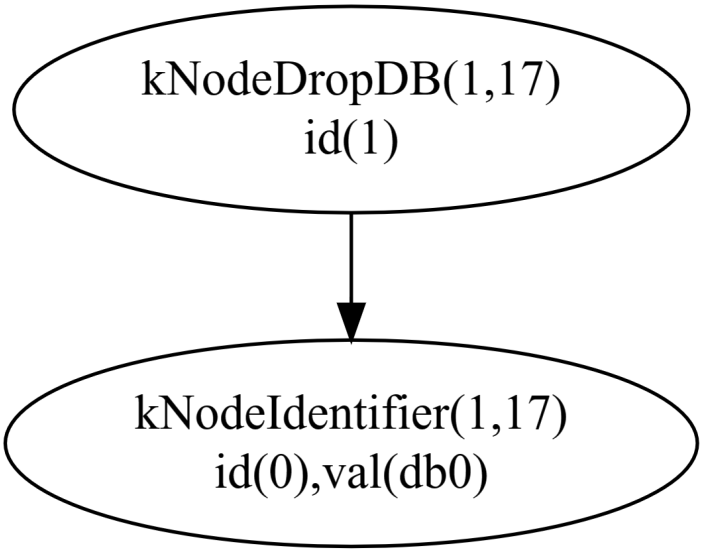
函数声明：

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

SQL语句声明：

```
drop database db0;
```

语法树示意图：



- `DropDatabase`的语法树有两个节点，第一个节点是标识，第二个节点带有创建数据库名字信息。
- 丢弃`Database`的名字来源于语法树根节点的儿子的`val_`。
- 查找`dbs_`中是否有该数据库：

```
1  dbs_.find(db_name) != dbs_.end()//
```

`find`方法可以根据名字顺序去找到对应元素的迭代指针，`end`方法返回数据库末尾指针，如果遍历到最后，则没有找到该名字的数据库。

函数实现：

```
1  dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast,
   ExecuteContext *context) {
2    string db_name = ast->child_->val_;
3    if (dbs_.find(db_name) == dbs_.end()) {
4      return DB_NOT_EXIST;
5    }
6    remove(db_name.c_str());
7    delete dbs_[db_name];
8    dbs_.erase(db_name);
9    return DB_SUCCESS;
10 }
```

1. 通过寻找语法树节点的`val_`值定义所丢弃数据库名称
2. 判断所存数据库中是否存在与所建立数据库名称一致的数据库，如果不存在则返回`DB_NOT_EXIST`表明不存在该待删除的数据库。

3. 通过`delete`函数来释放所在所选数据库

ExecuteShowDatabases()

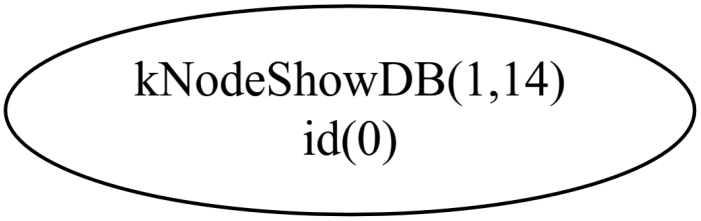
函数声明：

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

SQL语句声明：

```
show databases;
```

语法树示意图：



- 语法树较简单，语法树节点作为`Show databases`的信号节点出现
- 遍历`dbs_`的方法：用智能指针

```
  for (const auto &itr : dbs_) {
    cout << "| " << std::left << setfill(' ') <<
setw(max_width) << itr.first << " |" << endl;
  }
```

函数实现：

```
dberr_t ExecuteEngine::ExecuteShowDatabases(pSyntaxNode ast,
ExecuteContext *context) {
  if (dbs_.empty()) {
    cout << "Empty set (0.00 sec)" << endl;
    return DB_SUCCESS;
  }
  int max_width = 8;
  for (const auto &itr : dbs_) {
```

```
    if (itr.first.length() > max_width) max_width =
itr.first.length();
  }
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  cout << "| " << std::left << setfill(' ') << setw(max_width)
<< "Database"
       << " |" << endl;
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  for (const auto &itr : dbs_) {
    cout << "| " << std::left << setfill(' ') << setw(max_width)
<< itr.first << " |" << endl;
  }
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  return DB_SUCCESS;
}
```

1. 使用`empty()`方法检测MiniSQL系统中有无有效数据库存在
2. 通过`auto`便于遍历整个MiniSQL，输出所含数据库的名称
3. 通过`stew()`函数便于控制表格间距以保持美观

ExecuteUseDatabase()

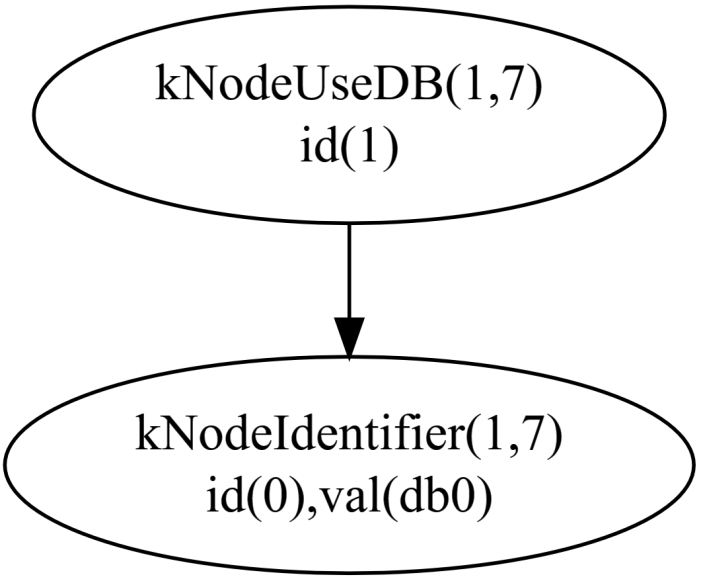
函数声明：

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

SQL语句声明：

```
use db0;
```

语法树示意图：



- `UseDatabase`的语法树有两个节点，第一个节点是标识，第二个节点带有创建数据库名字信息
- 使用`Database`的名字来源于语法树根节点的儿子的`val_`
- 另外在其他的执行器中写入了“检测是否选中数据库”模块，如果还未选中特定的数据库来进行表、索引等的操作是不可以的。

函数实现：

```
dberr_t ExecuteEngine::ExecuteUseDatabase(pSyntaxNode ast,
ExecuteContext *context) {
  string db_name = ast->child_->val_;
  if (dbs_.find(db_name) != dbs_.end()) {
    current_db_ = db_name;
    cout << "Database changed" << endl;
    return DB_SUCCESS;
  }
  return DB_NOT_EXIST;
}
```

1. 使用`Database`的名字来源于语法树根节点的儿子的`val_`
2. 通过`find()`函数判断有无该数据库存在
3. 如果存在该数据库，只需要更改`current_db_`为所使用数据库名称即可

ExecuteShowTables()

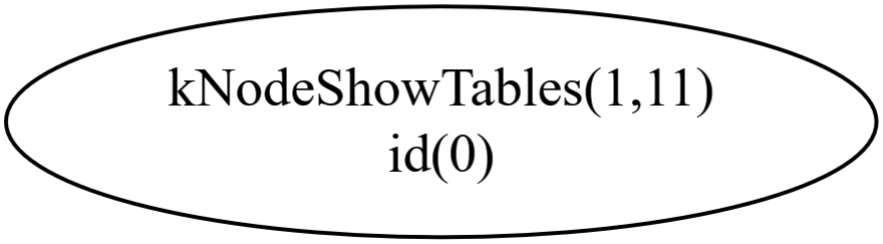
函数声明：

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

SQL语句声明：

```
show tables;
```

语法树示意图：



- 语法树较简单，语法树节点作为`Show Tables`的信号节点出现
- 查找数据库中是否有Table，并赋值填充vector

```
vector<TableInfo *> tables;
if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) == DB_FAILED) {
cout << "Empty set (0.00 sec)" << endl;
return DB_FAILED;
}
```

- 对于数据库中所有存在的表，信息遍历输出：

```
  for (const auto &itr : tables) {
    //****
    cout << "| " << std::left << setfill(' ') << setw(max_width) << itr->GetTableName() << " |" << endl;
    //****
  }
```

函数实现：

```
dberr_t ExecuteEngine::ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context) {
  if (current_db_.empty()) {
    cout << "No database selected" << endl;
    return DB_FAILED;
  }
  vector<TableInfo *> tables;
  if (dbs_[current_db_]->catalog_mgr_->GetTables(tables) == DB_FAILED) {
    cout << "Empty set (0.00 sec)" << endl;
    return DB_FAILED;
  }
  string table_in_db("Tables_in_" + current_db_);
  uint max_width = table_in_db.length();
  for (const auto &itr : tables) {
    if (itr->GetTableName().length() > max_width) max_width = itr->GetTableName().length();
  }
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  cout << "| " << std::left << setfill(' ') << setw(max_width) << table_in_db << " |" << endl;
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  for (const auto &itr : tables) {
    cout << "| " << std::left << setfill(' ') << setw(max_width) << itr->GetTableName() << " |" << endl;
  }
  cout << "+" << setfill('-') << setw(max_width + 2) << ""
       << "+" << endl;
  return DB_SUCCESS;
}
```

1. 使用`empty()`方法检测MiniSQL系统中有无有效数据库被选取
2. 通过`vector`存储所查询到的表，其中数据类型为`TableInfo *`,使用`dbs_[current_db_]->catalog_mgr_->GetTables(tables)`来填充向量
3. 通过`stew()`函数便于控制表格间距以保持美观，并一并输出表格名称

ExecuteCreateTable()

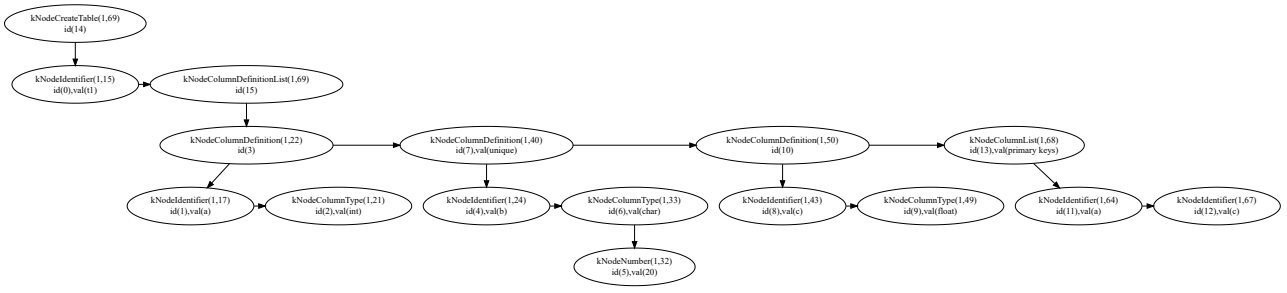
函数声明：

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

SQL语句声明：

```
create table t1(a int, b char(20) unique, c float, primary key(a,
c));
```

语法树示意图：



语法树较为复杂，但是也能看的下去：

- 首先指明表的名称，以`char*`类型存储在头节点的子节点`val_`中
- 再指明第一个变量的名称和数据类型，目前的数据库支持`int/float/string`三种变量类型，变量名称以`char*`类型存储在定义`Column`头节点的子节点`val_`中，变量类型以`char*`类型存储在上一个节点的兄弟节点`val_`中。此外如果是`float`类型要指明字符长度，也会在其他兄弟中找到
- 最后指明主键是哪几个，位于头节点的最后一个`sibling`节点，指明的`Column`存储在`val`中

函数实现：

```
dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast,
ExecuteContext *context) {
  if(current_db_ == ""){
    return DB_DATABASE_NOT_SELECTED;
  }
  string table_name = ast->child_->val_;
  vector<TableInfo *> tables;
```

```
  dbs_[current_db_]->catalog_mgr_->GetTables(tables);
  uint32_t table_index = 0;
  for(auto table :tables){
    if(table->GetTableName()==table_name){
      return DB_TABLE_ALREADY_EXIST;
    }
  }
  vector<Column *> Table_Columns;
  vector<string > index_keys;
  pSyntaxNode column_node = ast->child_->next_->child_;
  while(column_node!= nullptr){
    if(column_node->type_ ==
SyntaxNodeType::kNodeColumnDefinition){
      string col_name = column_node->child_->val_;
      index_keys.push_back(col_name);
      TypeId col_type;
      Column* new_col;
      bool is_unique = (column_node->val_!= nullptr&&
(string)column_node->val_=="unique")?true: false;
      if((string)column_node->child_->next_->val_=="int"){
        col_type = TypeId::kTypeInt;
      }else if((string)column_node->child_->next_->val_=="char")
{
        col_type = TypeId::kTypeChar;
      }else if((string)column_node->child_->next_-
>val_=="float"){
        col_type = TypeId::kTypeFloat;
      }else{
        col_type = TypeId::kTypeInvalid;
      }
      if(column_node->child_->next_->child_!= nullptr){
        //char
        string check_length = column_node->child_->next_-
>child_->val_;
        uint32_t col_length = (uint32_t)atoi(column_node-
>child_->next_->child_->val_);
        if(col_type ==kTypeChar&&
(check_length.find('.')!=string::npos||check_length.find('-
')!=string::npos)){
          return DB_FAILED;
        }
```

```
        new_col = new Column(col_name,col_type,col_length,table_index++,true,is_unique);
      }else{
        //non-char
        new_col = new Column(col_name,col_type,table_index++,true,is_unique);
      }
      Table_Columns.push_back(new_col);

    }else if(column_node->type_ == SyntaxNodeType::kNodeColumnList){
      //primary key
      pSyntaxNode primary_node = column_node->child_;
      while(primary_node!= nullptr){
        string primary_col = primary_node->val_;
        bool is_find = false;
        for(auto& item : Table_Columns){
          if(item->GetName() == primary_col){
            item->SetTableNullable(false);
            item->SerTableUnique(true);
            is_find = true;
            break;
          }
        }
        if(is_find == false){
          return DB_KEY_NOT_FOUND;
        }
        primary_node = primary_node->next_;
      }
    }
    column_node = column_node->next_;
  }
  Schema* table_schema = new Schema(Table_Columns);
  TableInfo* tem_info ;
  auto result = dbs_[current_db_]->catalog_mgr_->CreateTable
      (table_name,table_schema->DeepCopySchema(table_schema),nullptr,tem_info);
  IndexInfo* indexInfo;
  dbs_[current_db_]->catalog_mgr_->CreateIndex(
      table_name,table_name+"_index",index_keys,nullptr,indexInfo,"bplus");
```

```
76    return result;
77  }
```

1. 先读取要插入的表名称，通过`find()`函数来查询是否存在重复表名
2. 遍历语法树节点，对于每一个语法树节点首先判断类型，包括新建属性类型和主键定义类型
    - 对于每一个要新建属性语法树节点，获取新建属性名称、数据类型以及是否`unique`等信息，以方便我们去进行新建表
    - 对于定义主键的信息，我们采用同样的遍历方法，将收集到的要定义的主键存储在一个`vector<string> index_keys`中
3. 将产生的表信息、隔离等级等信息来创建表(利用`dberr_t CatalogManager::CreateTable()`)
4. 与此同时我们要建立与这个表相联系的主键索引，这里仿照后文的`CreateIndex`方法，使用`dberr_t CatalogManager::CreateIndex()`函数来创建索引，由于索引必须存在名字，这里我们借用该表的名字加上`_index`来形成我们的主键索引，即`table_name+"_index"`

ExecuteDropTable()

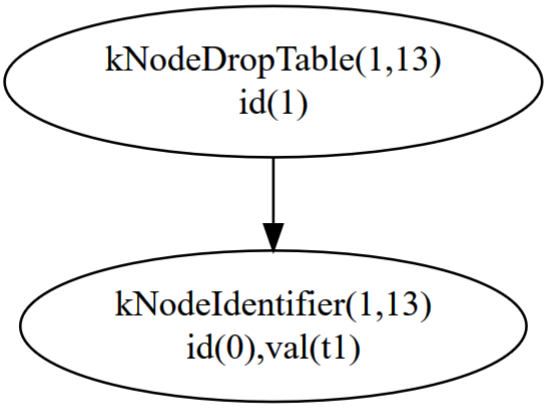
函数声明：

```
1  dberr_t ExecuteDropTable(pSyntaxNode ast, ExecuteContext
   *context);
```

SQL语句声明：

```
1  drop table t1;
```

语法树示意图：



- 删除表的语法树包括两大节点：语法标识节点和带有删除表信息的节点
- 要删除的表信息存储在根节点的子节点的`val`中
- 利用`GetTables`方法得到的Tables进行遍历查询是否存在该表即可

函数实现：

```
dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast,
ExecuteContext *context) {
  if(current_db_ == ""){
    return DB_DATABASE_NOT_SELECTED;
  }
  if (current_db_.empty()) {
    cout << "No database selected" << endl;
    return DB_FAILED;
  }
  string table_name = ast->child_->val_;
  vector<TableInfo *> tables;
  dbs_[current_db_]->catalog_mgr_->GetTables(tables);
  bool flag = false;
  for(auto table :tables){
    if(table->GetTableName()==table_name){
      flag = true;
      break;
    }
  }
  if(flag== false){
    return DB_TABLE_NOT_EXIST;
  }
  dbs_[current_db_]->catalog_mgr_->DropTable(table_name);
```

```
23    return DB_SUCCESS;
24  }
```

1. 使用`empty()`方法检查是否选中了数据库
2. 利用利用`GetTables`方法得到`tables`，里面存储了该数据库的所有表
3. 对得到的所有表利用名字的比对进行遍历，检验是否存在该表
4. 如果存在该表，再次调用`DropTable`方法删除即可，注意作用对象是该数据库下的`Catalog_manager`

ExecuteCreateIndex()

函数声明：

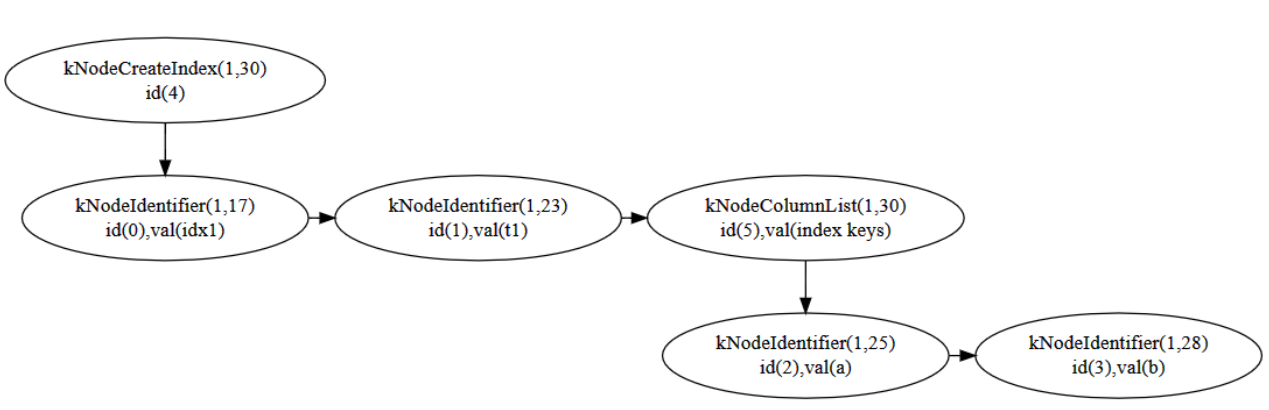
```
1  dberr_t ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext
   *context);
```

SQL语句声明：

Example1:

```
1  create index idx1 on t1(a, b);
```

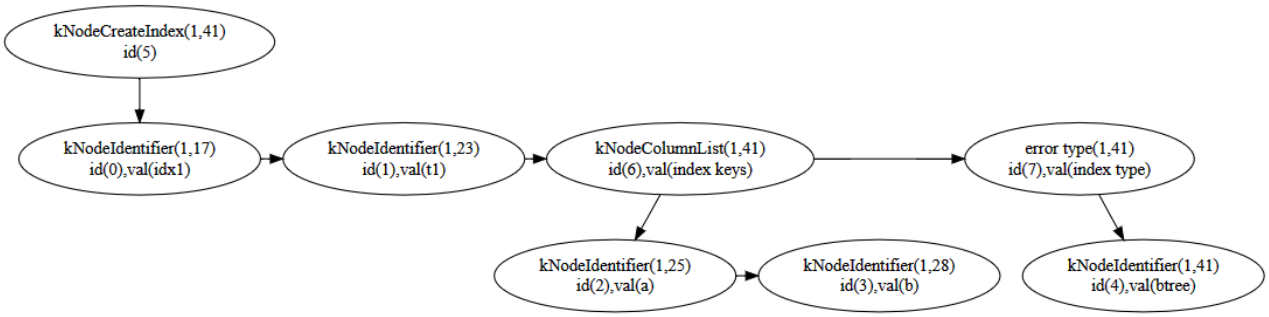
语法树示意图：



Example2:

```
1  -- "btree" can be replaced with other index types
2  create index idx1 on t1(a, b) using btree;
```

语法树示意图：



语法树与新建表的类似：

- 首先指明索引的名称，以`char*`类型存储在头节点的子节点`val_`中
- 再指明作用表的名称，以`char*`类型存储在头节点的`sibling`节点`val_`中
- 再指明索引作用的键是哪几个，位于头节点的下一个`sibling`节点，指明的`Column`存储在`val`中
- 最后一个结点可有可无，如果指明了索引类型它会出现，例如我们在语句最后加入`using btree`会出现头节点最后一个`sibling`节点，指明索引的类型是`B+`，内容依旧存储在`val`中

函数实现：

```
dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast,
ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateIndex" << std::endl;
#endif
  if(current_db_ == ""){
    return DB_DATABASE_NOT_SELECTED;
  }
  string index_name = ast->child_->val_;
  string table_name_index = ast->child_->next_->val_;
  pSyntaxNode key_node = ast->child_->next_->next_->child_;
  pSyntaxNode type_node = ast->child_->next_->next_->next_;
  string index_type;
  vector<IndexInfo *> indexes;
  vector<string> index_keys;
  dbs_[current_db_]->catalog_mgr_-
>GetTableIndexes(table_name_index,indexes);
```

```
  for(auto item:indexes){
    if(item->GetIndexName() == index_name){
      return DB_INDEX_ALREADY_EXIST;
    }
  }
  while( key_node->type_==kNodeIdentifier){
    index_keys.push_back((string)key_node->val_);
    if(key_node->next_ == nullptr){
      break;
    }
    key_node = key_node->next_;
  }
  if(type_node!= nullptr){
    index_type = type_node->child_->val_;
  }else{
    index_type = "btree";
  }
  return dbs_[current_db_]->catalog_mgr_->CreateIndex(
      table_name_index,index_name,index_keys,
nullptr,indexInfo,index_type);
}
```

1. 首先将新建索引的所有信息存储到变量中，例如索引名称为index_name，索引所在表名称为table_name_index，索引作用Column为vector index_keys
2. 检验索引是否已经出现过，利用字符串比较
3. 如果该索引从未出现过，调用`CreateIndex`方法删除即可，注意作用对象是该数据库下的`Catalog_manager`

ExecuteShowIndexes()

函数声明：

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

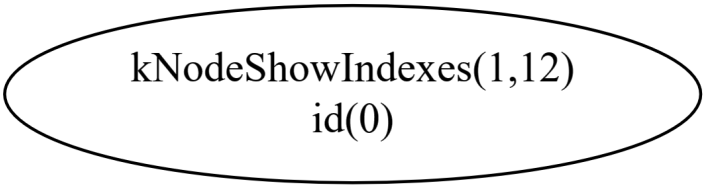
```
mysql> show index from book;
ERROR 1046 (3D000): No database selected
mysql> use library;
Database changed
mysql> show index from book;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| book  |          0 | PRIMARY  |            1 | book_id     | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| book  |          0 | category |            1 | category    | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| book  |          0 | category |            2 | press       | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| book  |          0 | category |            3 | author      | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| book  |          0 | category |            4 | title       | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| book  |          0 | category |            5 | publish_year | A         |           2 |     NULL |   NULL |      | BTREE      |         |               |
    | YES     | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
6 rows in set (0.01 sec)

mysql>
```

SQL语句声明：

```
1  show indexes;
```

语法树示意图：



语法树较为简单，算是一个标识符，说明该操作是`ShowIndexes`

函数实现：

```
1  dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast,
   ExecuteContext *context) {
2    if(current_db_ == ""){
3      return DB_DATABASE_NOT_SELECTED;
4    }
5    if (current_db_.empty()) {
6      cout << "No database selected" << endl;
7      return DB_FAILED;
8    }
9    int max_w_table = 5,max_w_index=10,max_w_col=11,max_w_type=10;
10   vector<TableInfo*> tables;
11   dbs_[current_db_]->catalog_mgr_->GetTables(tables);
12   vector<IndexInfo *> Total_indexes;
13   vector<IndexInfo* > indexes;
14   vector<string>tem_tables_name;
15   for(auto table : tables){
```

```
    max_w_table = table->GetTableName().length()>max_w_table?table->GetTableName().length():max_w_table;
    dbs_[current_db_]->catalog_mgr_->GetTableIndexes(table->GetTableName(),indexes);
    for(auto index:indexes){
      tem_tables_name.push_back(table->GetTableName());
      max_w_index = index->GetIndexName().length()>max_w_index?index->GetIndexName().length():max_w_index;
      int total = -1;
      for(auto col:index->GetIndexKeySchema()->GetColumns()){
        total+=(col->GetName().length()+1);
      }
      max_w_col = total>max_w_col?total:max_w_col;
      Total_indexes.push_back(index);
    }
  }
  if(Total_indexes.empty()){
    cout<<"Empty set (0.00 sec)"<<endl;
    return DB_SUCCESS;
  }
  cout << "+" << setfill('-') << setw(max_w_table + 2) << ""
       << "+" << setfill('-') << setw(max_w_index + 2) << ""
       << "+" << setfill('-') << setw(max_w_col + 2) << ""
       << "+" << setfill('-') << setw(max_w_type + 2) << ""
       << "+" << endl;
  cout << "| " << std::left << setfill(' ') << setw(max_w_table+1) << "Table"
       << "| " << std::left << setfill(' ') << setw(max_w_index+1) << "Index_name"
       << "| " << std::left << setfill(' ') << setw(max_w_col+1) << "Column_name"
       << "| " << std::left << setfill(' ') << setw(max_w_type+1) << "Index_type"
       << "|"<<endl;

  cout << "+" << setfill('-') << setw(max_w_table + 2) << ""
       << "+" << setfill('-') << setw(max_w_index + 2) << ""
       << "+" << setfill('-') << setw(max_w_col + 2) << ""
       << "+" << setfill('-') << setw(max_w_type + 2) << ""
       << "+" << endl;
  int cnt_table = 0;
  for(auto index:Total_indexes){
```

```
51      cout << "| " << std::left << setfill(' ') <<
    setw(max_w_table+1) << tem_tables_name[cnt_table++];
52      cout << "| " << std::left << setfill(' ') <<
    setw(max_w_index+1) << index->GetIndexName();
53      string tol_col;
54      for(auto col:index->GetIndexKeySchema()->GetColumns()){
55        if (col->GetName() == index->GetIndexKeySchema()-
    >GetColumns()[0]->GetName()) {
56          tol_col+=col->GetName();
57        } else {
58          tol_col += (","+col->GetName());
59        }
60      }
61      cout << "| " << std::left << setfill(' ') << setw(max_w_col)
    << tol_col;
62      cout << " | " << std::left << setfill(' ') <<
    setw(max_w_type) << "BTREE"<< " |"<<endl;
63    }
64    cout << "+" << setfill('-') << setw(max_w_table + 2) << ""
65         << "+" << setfill('-') << setw(max_w_index + 2) << ""
66         << "+" << setfill('-') << setw(max_w_col + 2) << ""
67         << "+" << setfill('-') << setw(max_w_type + 2) << ""
68         << "+" << endl;
69    for (const auto &itr : tables) {
70    }
71    return DB_SUCCESS;
72  }
73
```

1. 由于SQL命令未指明作用表的对象，所以我们需要将所有表的索引都找出来，常规办法自然是遍历：先找到所有表的名字，再根据表的名字去找自己表下的索引，存储到`vector`或者是`map`中
2. 接着是把索引的内容写出，此处模仿MySQL的索引展示，主要分为四个部分，分别是索引所在表的名称、索引名称、索引对于元组、索引对应类型。利用`stew`等函数实现对齐，保证美观

ExecuteDropIndex()

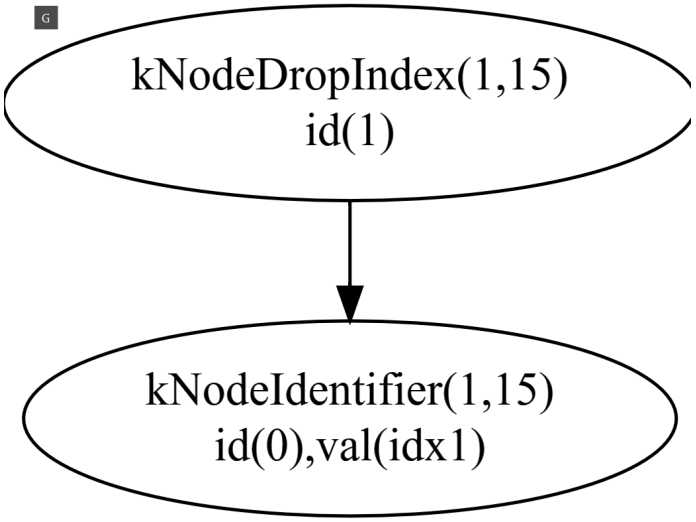
函数声明：

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

SQL语句声明：

```
drop index idx1;
```

语法树示意图：



- 删除索引的语法树包括两大节点：语法标识节点和带有删除索引信息的节点
- 要删除的索引信息存储在根节点的子节点的`val`中
- 利用`GetTableIndexes`方法得到的`index`进行遍历查询是否存在该表即可

函数实现：

```
dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast,
ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropIndex" << std::endl;
#endif
  if(current_db_ == ""){
    return DB_DATABASE_NOT_SELECTED;
  }
  string index_name = ast->child_->val_;
  string table_name_drop;
  vector<TableInfo*> tables;
  dbs_[current_db_]->catalog_mgr_->GetTables(tables);
```

```
  vector<IndexInfo* > indexes;
  bool if_index = false;
  for(auto table : tables){
    dbs_[current_db_]->catalog_mgr_->GetTableIndexes(table->GetTableName(),indexes);
    for(auto index:indexes){
      if(index->GetIndexName() == index_name){
        if_index = true;
        table_name_drop = table->GetTableName();
        break;
      }
    }
    if(if_index == true){
      break;
    }
  }
  if(if_index == false){
    return DB_INDEX_NOT_FOUND;
  }
  dbs_[current_db_]->catalog_mgr_->DropIndex(table_name_drop,index_name);
  return DB_SUCCESS;
}
```

1. 使用`empty()`方法检查是否选中了数据库
2. 利用利用`GetTableIndexes`方法得到`indexes`，里面存储了该数据库的所有索引
3. 对得到的所有索引利用名字的比对进行遍历，检验是否存在该索引
4. 如果存在该索引，再次调用`DropIndex`方法删除即可，注意作用对象是该数据库下的`Catalog_manager`

ExecuteExecfile()

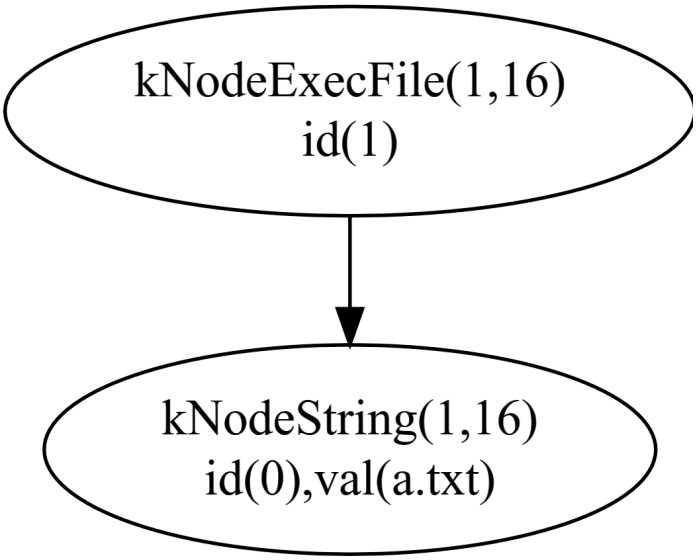
函数声明：

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

SQL语句声明：

```
execfile "a.txt";
```

语法树示意图：



ExecuteQuit()

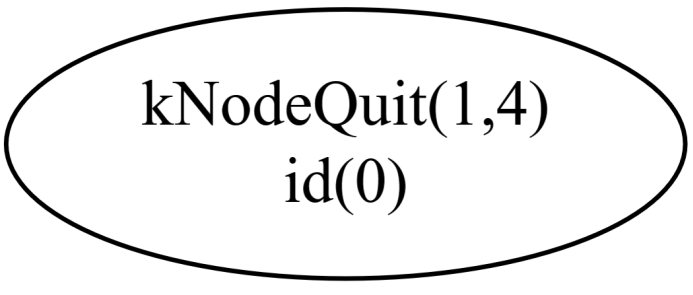
函数声明：

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

SQL语句声明：

```
quit;
```

语法树示意图：



语法树较为简单，算是一个标识符，说明该操作是`Quit`

函数实现：

```
dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast,
ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteExecfile" << std::endl;
#endif
  return DB_EXECUTE;
}
```

这里的文件执行主要是返回一个结果，与常规函数不同，由于SQL系统在运行的时候对命令行的输入已经实现过，同层级实现文件输入的接口转换较为方便，所以通过返回一个独特的dberr_t值来标识我们进入了文件输入状态，现给出main函数以及对应的文件输入函数：

main.cpp:

```
#include <cstdio>

#include "executor/execute_engine.h"
#include "glog/logging.h"
#include "parser/syntax_tree_printer.h"
#include "utils/tree_file_mgr.h"

#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
extern "C" {
int yyparse(void);
FILE *yyin;
#include "parser/minisql_lex.h"
#include "parser/parser.h"
}

void InitGoogleLog(char *argv) {
  FLAGS_logtostderr = true;
  FLAGS_colorlogtostderr = true;
  google::InitGoogleLogging(argv);
```

```
  // LOG(INFO) << "glog started!";
}

//read_state, true reps read from file,false reps read from buffer
bool read_state = false;
bool ExecFile_flag = false;
auto start_time = std::chrono::system_clock::now();
string file_name;
long file_pointer = 0;
long file_size = 99999;
void InputCommand_file(char* input, const int len){
  memset(input, 0, len);
  //initial fstraeam
  fstream file;
  file.open(file_name.c_str(),ios::in);
  //initial file_size
  file.seekg(0,ios::end);
  file_size = file.tellg();
  file_size--;
  //initial read pointer
  file.seekg(file_pointer,ios::beg);
  int i = 0;
  char ch;
  //read file
  while (file.get(ch),ch != ';') {
    if(file_pointer++>=file_size){
      read_state = false;
      file_pointer = 0;
    }
    input[i++] = ch;
  }
  input[i++] = ch;  // ;
  if(file_pointer<file_size) {
    file.get(ch);  // remove enter
  }
  file_pointer+=2;
  file.close();
}

void InputCommand(char *input, const int len) {
  memset(input, 0, len);
```

```
64
65    printf("StarSQL> ");
66    int i = 0;
67    char ch;
68    while ((ch = getchar()) != ';') {
69      input[i++] = ch;
70    }
71    input[i] = ch;  // ;
72    getchar();      // remove enter
73  }
74
75  std::string timeToString(std::chrono::system_clock::time_point
    &t) {
76    std::time_t time = std::chrono::system_clock::to_time_t(t);
77    std::string time_str = std::ctime(&time);
78    time_str.resize(time_str.size() - 1);
79    return time_str;
80  }
81
82  int main(int argc, char **argv) {
83    InitGoogleLog(argv[0]);
84    // command buffer
85    const int buf_size = 1024;
86    char cmd[buf_size];
87    // executor engine
88    ExecuteEngine engine;
89
90    while (1) {
91      if(read_state == true){
92        //read from file
93        InputCommand_file(cmd,buf_size);
94      }else{
95        // read from buffer
96        InputCommand(cmd, buf_size);
97      }
98      // create buffer for sql input
99      YY_BUFFER_STATE bp = yy_scan_string(cmd);
100     if (bp == nullptr) {
101       LOG(ERROR) << "Failed to create yy buffer state." <<
    std::endl;
102       exit(1);
103     }
```

```
104     yy_switch_to_buffer(bp);
105 
106     // init parser module
107     MinisqlParserInit();
108 
109     // parse
110     yyparse();
111 
112     // parse result handle
113     if (MinisqlParserGetError()) {
114       // error
115       printf("%s\n", MinisqlParserGetErrorMessage());
116     } else {
117       // Comment them out if you don't need to debug the syntax
    tree
118     }
119     auto result = engine.Execute(MinisqlGetParserRootNode());
120     //Execute in file
121     if(result==DB_EXECUTE ){
122       file_name = MinisqlGetParserRootNode()->child_->val_;
123       start_time = std::chrono::system_clock::now();
124       read_state = true;
125     }
126     if(file_pointer>=file_size){
127       read_state = false;
128       file_pointer = 0;
129       auto stop_time = std::chrono::system_clock::now();
130       double duration_time =
131 
     double((std::chrono::duration_cast<std::chrono::milliseconds>
    (stop_time - start_time)).count());
132       cout << "Total time of file execution : " << fixed <<
    setprecision(4) << duration_time / 1000 << " sec." <<
    std::endl;
133     }
134     // clean memory after parse
135     MinisqlParserFinish();
136     yy_delete_buffer(bp);
137     yylex_destroy();
138 
139     // quit condition
140     engine.ExecuteInformation(result);
```

```
141       auto time_p = std::chrono::system_clock::now();
142       cout << "Current time: " << timeToString(time_p) << endl;
143       if (result == DB_QUIT) {
144         break;
145       }
146     }
147     return 0;
148   }
```

我们只对实现了的文件输入做介绍：

1. 首先读取到命令读取时，利用read_state更改为true，标识我们进入了文件读入状态
2. 在下一次读取时，由于开头的条件判断，我们的输入流从命令行输入改为文件读入，我们参照了命令行读取函数，读取到分号停止。
3. 在文件读入中，我们利用了文件位置指针(全局变量)，来标识我们的文件是否读取结束
4. 命令作用与命令行一致，我们只是实现了输入流的转换，感谢助教和历届助教的框架建立和不断完善。

<--------以下为事务相关，暂不实现---------->

ExecuteTrxBegin()

函数声明：

```
1  dberr_t ExecuteTrxBegin(pSyntaxNode ast, ExecuteContext
   *context);
```

ExecuteTrxCommit()

函数声明：

```
1  dberr_t ExecuteTrxCommit(pSyntaxNode ast, ExecuteContext
   *context);
```

ExecuteTrxRollback()

函数声明：

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

遇到的问题及解决方法

本次实验的测试主要面向其他复杂的算子，由此对我们的直接执行的算子参考意义不大，对于模块5，迎接它的最大考验其实是顶层命令的实现。现略数一些我遇到的问题：

- 首先是建表环节，我们不仅需要收集好信息来实现表的建立，而且要对表建立索引，因为在`Insert`算子中检验主键或是`Unique`约束的方法中，是对索引内容进行排重，所以不建立索引默认不进行主键的检验，导致我们的插入实现了无条件插入，这显然是我们不想看到的。
- 作为顶层的少数几个模块，我们不必去深究底层的逻辑去实现内容的更新，而是学会看懂接口，那些接口帮助我们实现了许多功能要合理地使用，而且直接对底层的修改其实不一定符合工程的设计原则。
- 设计`ShowIndexes`的时候，我参照了MySQL的`Index`输出格式，但是我们此工程设计的索引调用接口不是很多，我只做到了模仿其中的四项进行了展示，原理其实和`ShowDatabases`差不多。

总结、心得

不知不觉已经写到这里了，每一次打开Clion的时候都不知道自己能写到哪里，好在我有可靠的小组成员和我一起Debug、有悉心指导的聂俊哲&石宇新助教，在数据库学习中感谢孙建伶老师的悉心教导，模块5就写到这里，是一次酣畅淋漓的C++工程管理和数据库学习。

今天是6月8日高考第二日，祝全国考生以及母校平遥中学的学弟学妹们高考顺利，前程似锦！

希君生羽翼，一化北溟鱼！