#5 SQL EXECUTOR

#5.1 实验概述

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` 将执行结果返回给上层模块。

考虑到同学们尚未接触到编译原理的相关知识，在本实验中，我们已经为同学们设计好MiniSQL中的Parser模块，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。

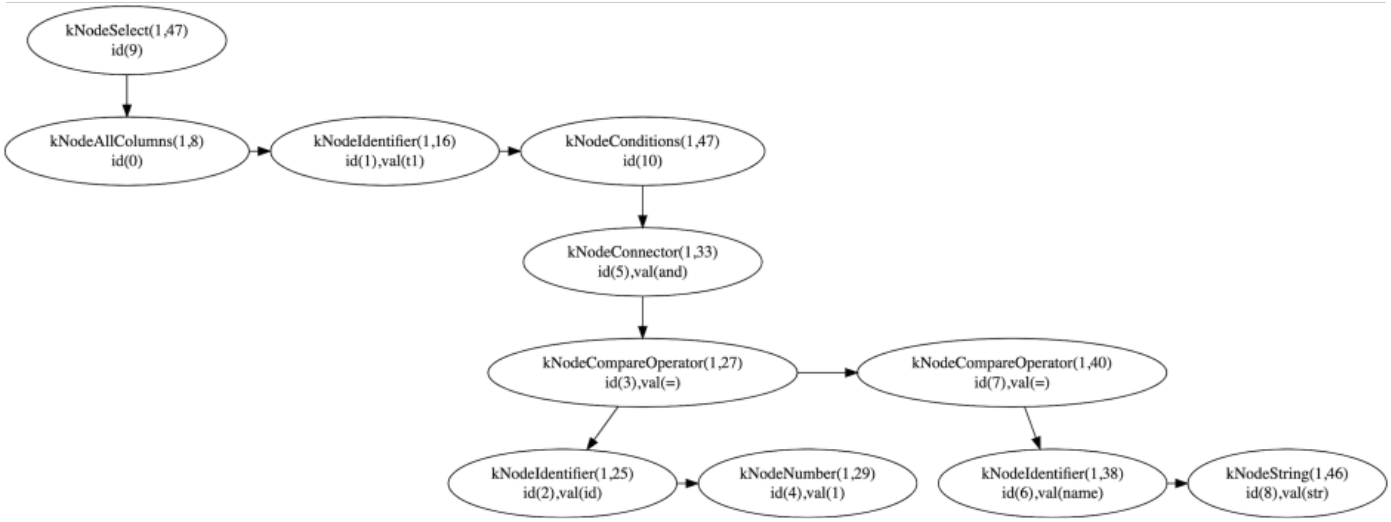
#5.1.1 语法树数据结构

以下是语法树（结点）的数据结构定义，每个结点都包含了一个唯一标识符 `id_`，唯一标识符在调用 `CreateSyntaxNode` 函数时生成（框架中已经给出实现）。 `type_` 表示语法树结点的类型， `line_no_` 和 `col_no_` 表示该语法树结点对应的是SQL语句的第几行第几列， `child_` 和 `next_` 分别表示该结点的子结点和兄弟结点， `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）。

src/include/parser/syntax_tree.h　　C++　复制代码

```cpp
/**
 * 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;
```

举一个简单的例子， `select * from t1 where id = 1 and name = "str";` 这一条SQL语句生成的语法树如下。以根结点为例说明， `kNodeSelect` 为结点的类型， `(1,47)` 表示该结点在规约（*reduce*，编译原理中的术语）后位于行的第1行第47列（语句末）， `id(9)` 表示该结点的 `id_` 为 `9`。



#5.2 解析语法树完成命令执行

Parser模块中目前能够支持以下类型的SQL语句。其中包含了一些在语法定义上正确，但在语义上错误的SQL语句（如Line 8～10）需要同学们在执行器中对这些特殊情况进行处理。此外涉及到事务开启、提交和回滚相关的 `begin`、 `commit` 和 `rollback` 命令可以不做实现。

SQL语句示例 SQL 复制代码

```sql
create database db0;
drop database db0;
show databases;
use db0;
show tables;
create table t1(a int, b char(20) unique, c float, primary key(a, c));
create table t1(a int, b char(0) unique, c float, primary key(a, c));
create table t1(a int, b char(-5) unique, c float, primary key(a, c));
create table t1(a int, b char(3.69) unique, c float, primary key(a, c));
create table t1(a int, b char(-0.69) unique, c float, primary key(a, c));
create table student(
  sno char(8),
  sage int,
  sab float unique,
  primary key (sno, sab)
);
drop table t1;
create index idx1 on t1(a, b);
-- "btree" can be replaced with other index types
create index idx1 on t1(a, b) using btree;
drop index idx1;
show indexes;
select * from t1;
select id, name from t1;
select * from t1 where id = 1;
-- note: use left association
select * from t1 where id = 1 and name = "str";
select * from t1 where id = 1 and name = "str" or age is null and bb not null;
insert into t1 values(1, "aaa", null, 2.33);
delete from t1;
delete from t1 where id = 1 and amount = 2.33;
update t1 set c = 3;
update t1 set a = 1, b = "ccc" where b = 2.33;
begin;
commit;
rollback;
quit;
execfile "a.txt";
```

在Parser模块调用 `yyparse()` （一个示例在 `src/main.cpp` 中）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执行器 `ExecuteEngine` （定义于 `src/include/executor/execute_engine.h` ）后， `ExecuteEngine` 将会根据语法树根结点的类型，分发到对应的执行函数中，以完成不同类型SQL语句的执行。

在本节中，你需要实现 `ExecuteEngine` 中所有的执行函数，它们被声明为 `private` 类型的成员，即所有的执行过程对上层模块是隐藏的，上层模块只需要调用 `ExecuteEngine::execute()` 并传入语法树结点即可无感知地获取到执行结果。

- `ExecuteEngine::ExecuteCreateDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteDropDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowDatabases(*ast, *context)`
- `ExecuteEngine::ExecuteUseDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowTables(*ast, *context)`
- `ExecuteEngine::ExecuteCreateTable(*ast, *context)`
- `ExecuteEngine::ExecuteDropTable(*ast, *context)`
- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)`
- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)`
- `ExecuteEngine::ExecuteDropIndex(*ast, *context)`
- `ExecuteEngine::ExecuteSelect(*ast, *context)`
- `ExecuteEngine::ExecuteInsert(*ast, *context)`
- `ExecuteEngine::ExecuteDelete(*ast, *context)`
- `ExecuteEngine::ExecuteUpdate(*ast, *context)`
- `ExecuteEngine::ExecuteExecfile(*ast, *context)`
- `ExecuteEngine::ExecuteQuit(*ast, *context)`
- `ExecuteEngine::ExecuteTrxBegin(*ast, *context)` ：事务相关，可不实现
- `ExecuteEngine::ExecuteTrxCommit(*ast, *context)` ：事务相关，可不实现
- `ExecuteEngine::ExecuteTrxRollback(*ast, *context)` ：事务相关，可不实现

Note: 执行结果上下文 `ExecuteContext` 中提供了部分可能需要用到的数据，同学们在实际编程的时候根据需要自行定义 `ExecuteContext` 即可。

```cpp
/**
 * ExecuteContext stores all the context necessary to run in the execute engine
 * This struct is implemented by student self for necessary.
 *
 * eg: transaction info, execute result...
 */
struct ExecuteContext {
  bool flag_quit_{false};
  Transaction *txn_{nullptr};
};
```

#5.3 模块相关代码

- `src/main.cpp`
- `src/include/executor/execute_engine.h`
- `src/executor/execute_engine.cpp`

#5.4 开发提示

1. 整个MiniSQL项目推荐在**夏学期第7周**前完成；
2. 框架中已经给出了语法树的 `PrintTree()` 方法，它能够打印语法树中的每一个结点（输出DOT格式），具体用法和之前的B+树打印类似，输出的结果放在可视化界面中可以用作调试。此外也可以使用GDB或IDE自带的调试工具在完成SQL语法分析后得到语法树的语句打上断点以进行调试。
3. 如果需要更改语法和文法以支持新的SQL命令，可以在学习LEX和YACC的相关知识后，修改 `src/include/parser/minisql.l` 和 `src/include/parser/minisql.y` 文件，然后执行 `src/include/parser/compile.sh` 脚本（它会自动生成对应的 `lex` 和 `yacc` 代码并移动到工程的指定目录下），最后需要重新执行 `cmake ..` 完成更新；
4. 本模块可以不设计相关的测试代码，以手动执行SQL命令观察结果来代替测试。

#5.5 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为 `0`；
2. 请勿将代码发布到公共Github存储库上。