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

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()

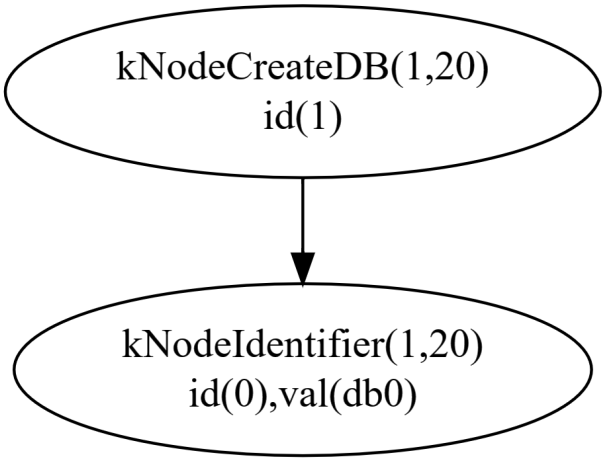
函数声明：

```
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()

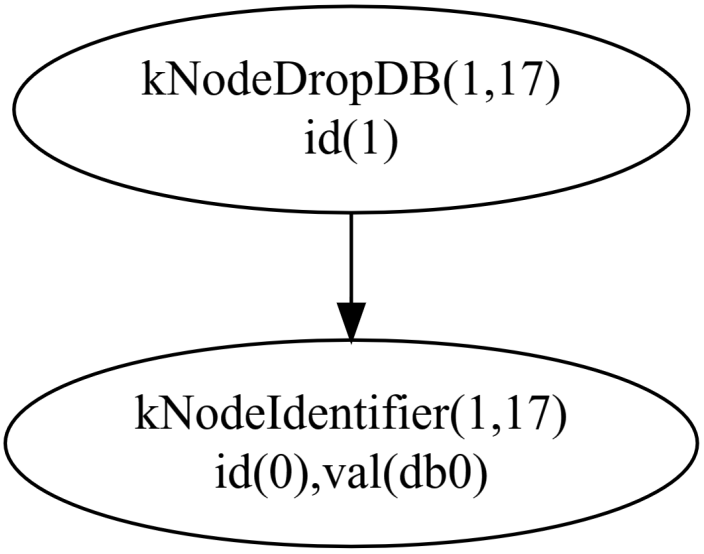
函数声明：

```
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()

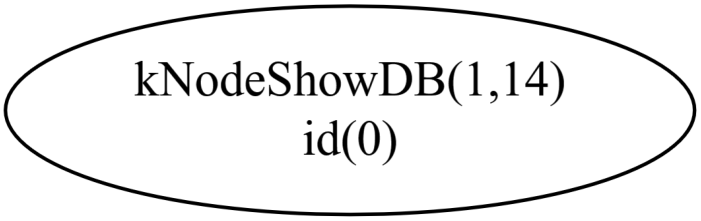
函数声明：

```
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()

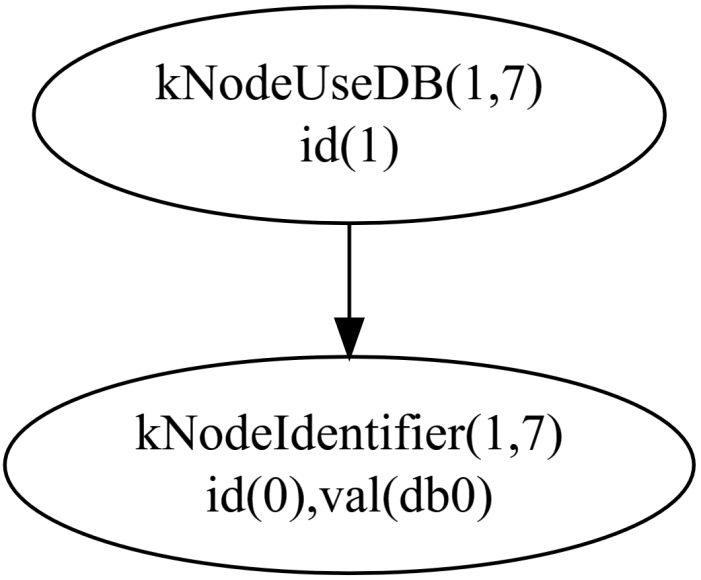
函数声明：

```
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()

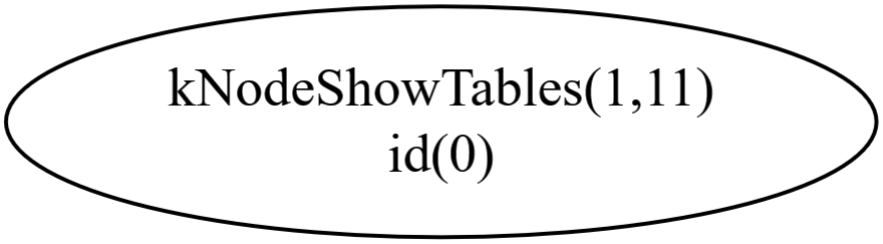
函数声明：

```
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()

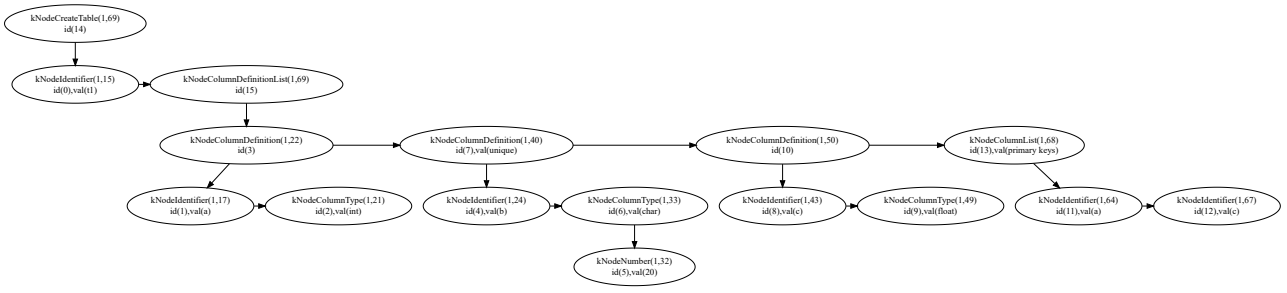
函数声明：

```
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()

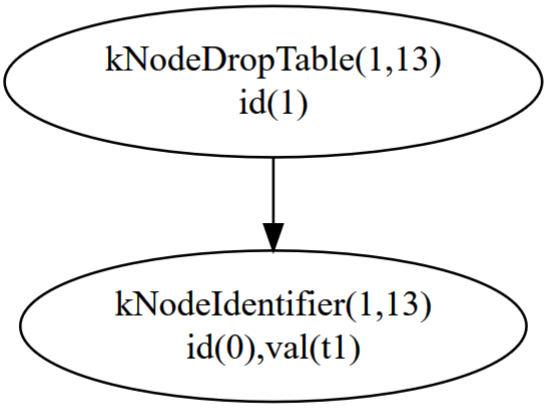
函数声明：

```
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：

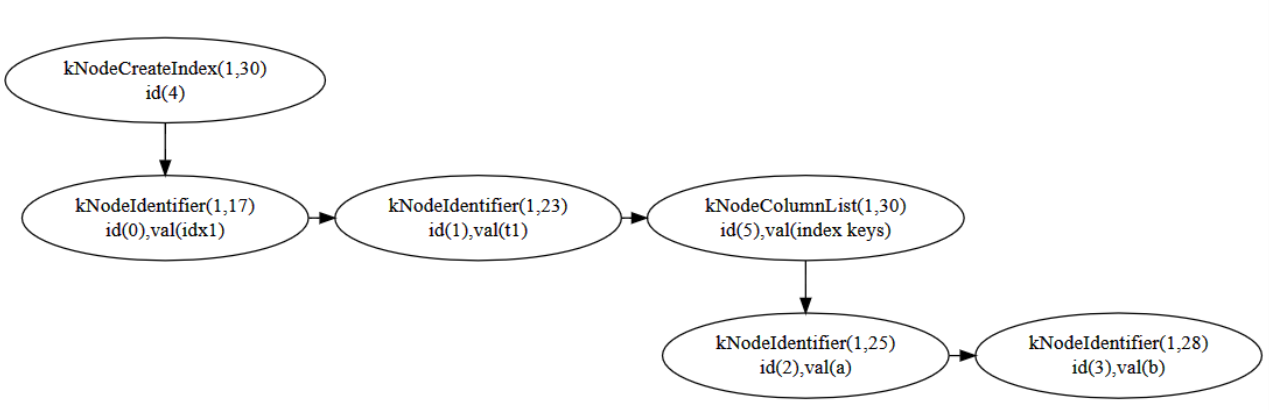
```
1  dberr_t ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext
   *context);
```

SQL语句声明：

Example1:

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

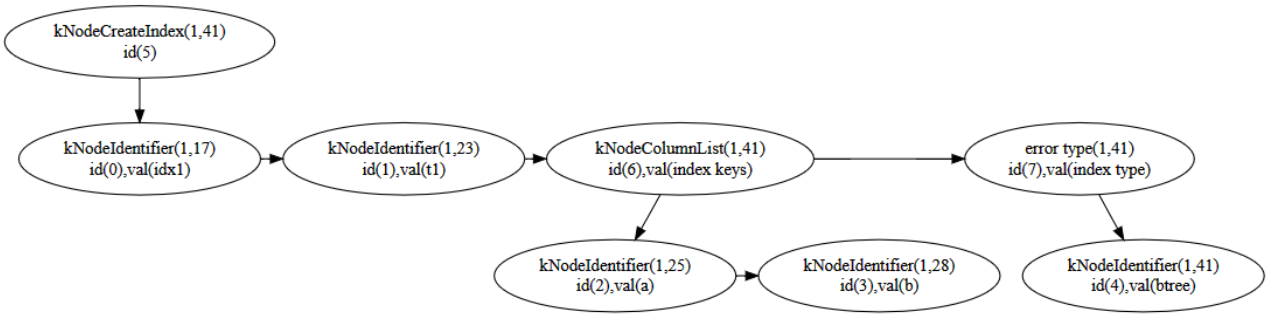
语法树示意图：



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);
```

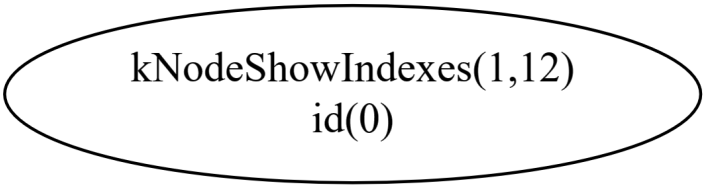
```
mysql> show index from book;
ERROR 1046 (3D000): No database selected
mysql> use library;
Database changed
mysql> show index from book;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| 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()

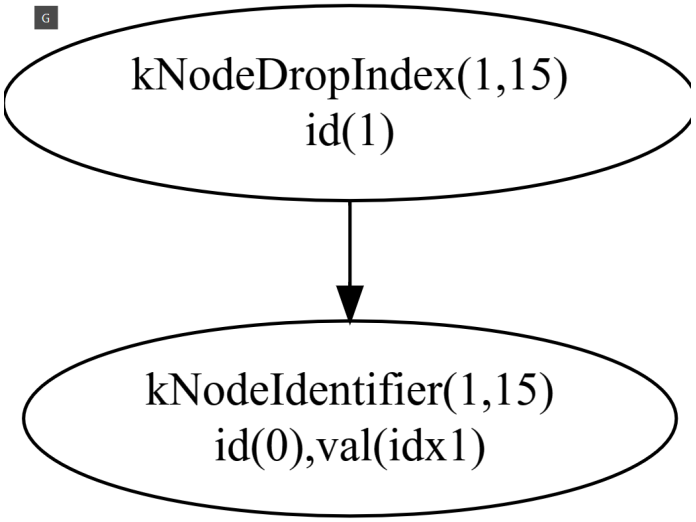
函数声明：

```
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()

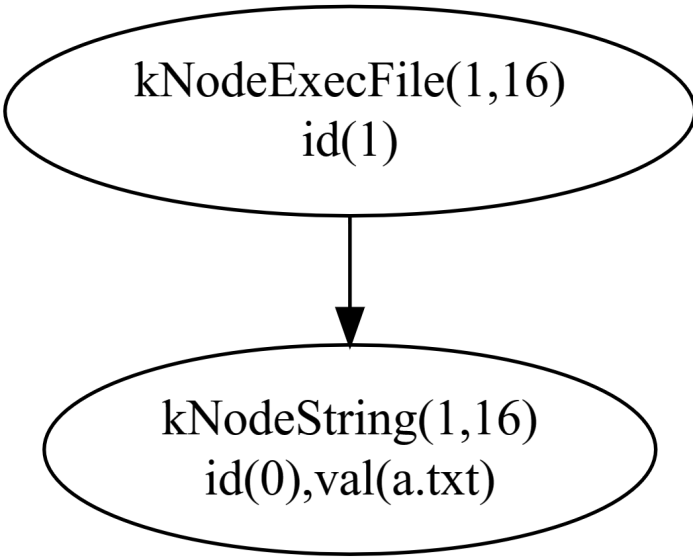
函数声明：

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

SQL语句声明：

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

语法树示意图：



ExecuteQuit()

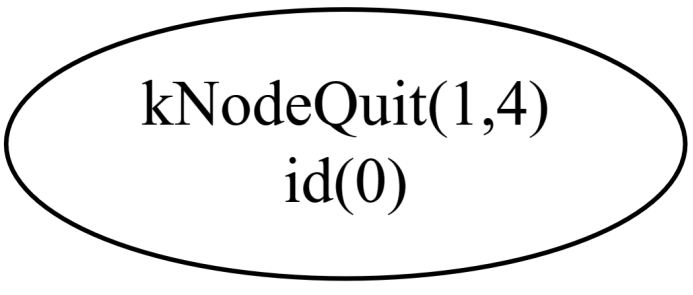
函数声明：

```
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！

浙江大学

《数据库系统》
实验报告

作业名称	：	MiniSQL
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 6 RECOVERY MANAGER

实验目的

> *Recovery Manager 负责管理和维护数据恢复的过程，包括：*
>
> - *日志结构的定义*
> - *检查点CheckPoint的定义*
> - *执行Redo、Undo等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态*
>
> *出于实现复杂度的考虑，同时为了避免各模块耦合太强，前面模块的问题导致后面模块完全无法完成，同组成员的工作之间影响过深，我们将Recovery Manager模块单独拆了出来。另外为了减少重复的内容，我们不重复实现日志的序列化和反序列化操作，实现一个纯内存的数据恢复模块即可。*

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*

实验流程

设计思路

数据恢复是一个很复杂的过程，需要涉及系统的多个模块。以InnoDB为例，在其恢复过程中需要`redo log`、`binlog`、`undo log`等参与，这里把InnoDB的恢复过程主要划分为两个阶段：第一阶段主要依赖于`redo log`的恢复，而第二阶段需要`binlog`和`undo log`的共同参与。

第一阶段，数据库启动后，InnoDB会通过`redo log`找到最近一次`checkpoint`的位置，然后根据`checkpoint`相对应的`LSN`开始，获取需要重做的日志，接着解析日志并且保存到一个哈希表中，最后通过遍历哈希表中的`redo log`信息，读取相关页进行恢复。

在该阶段中，所有被记录到`redo log`但是没有完成数据刷盘的记录都被重新落盘。然而，InnoDB单靠`redo log`的恢复是不够的，因为数据库在任何时候都可能发生宕机，需要保证重启数据库时都能恢复到一致性的状态。这个一致性的状态是指此时所有事务要么处于提交，要么处于未开始的状态，不应该有事务处于执行了一半的状态。所以我们可以通过`undo log`在数据库重启时把正在提交的事务完成提交，活跃的事务回滚，保证了事务的原子性。此外，只有`redo log`还不能解决主从数据不一致等问题。

第二阶段，根据`undo`中的信息构造所有未提交事务链表，最后通过上面两部分协调判断事务是否需要提交还是回滚。InnoDB使用了多版本并发控制(MVCC)以满足事务的隔离性，简单的说就是不同活跃事务的数据互相是不可见的，否则一个事务将会看到另一个事务正在修改的数据。InnoDB借助`undo log`记录的历史版本数据，来恢复出对于一个事务可见的数据，满足其读取数据的请求。

log_rec.h相关

在我们的实验中，日志在内存中以`LogRec`的形式表现，定义于`src/include/recovery/log_rec.h`.

这里我们实现以下函数:

- `CreateInsertLog()`：创建一条插入日志
- `CreateDeleteLog()`：创建一条删除日志
- `CreateUpdateLog()`：创建一条更新日志
- `CreateBeginLog()`：创建一条事务开始日志
- `CreateCommitLog()`：创建一条事务提交日志
- `CreateAbortLog()`：创建一条事务回滚日志

LocRecType类

```
enum class LogRecType {
    kInvalid,
    kInsert,
    kDelete,
    kUpdate,
    kBegin,
    kCommit,
    kAbort,
};
```

`enum`数据类型的`LogRecType`存储了六种有效事务状态类型：插入(Insert)、删除(Delete)、更新(Update)、开始(Begin)、提交(Commir)、取消(Abort).

LogRec日志记录结构体

> *内存日志结构。这里可以不考虑消耗内存空间的优化，实现一种能够用于所有类型日志的日志结构。*

```
struct LogRec {
  LogRec() = default;

  LogRecType type_{LogRecType::kInvalid};
  lsn_t lsn_{INVALID_LSN};
  lsn_t prev_lsn_{INVALID_LSN};
  txn_id_t txn_id_{INVALID_TXN_ID};
  KeyType ins_key_,del_key_,old_key_,new_key_;
  [[maybe_unused]] ValType
ins_val_{},del_val_{},old_val_{},new_val_{};

  LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):
  type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id)
  {}

  /* used for testing only */
  static std::unordered_map<txn_id_t, lsn_t> prev_lsn_map_;
  static lsn_t next_lsn_;
};
```

变量类型	变量名称	含义
LogRecType	type_	该日志的事务状态类型(如插入、删除)
lsn_t	lsn_	该日志的LSN号
lsn_t	prev_lsn_	关于该事务的前一个日志的LSN号
txn_id_t	txn_id_	该日志的对应的事务id
KeyType	ins_key_	对于插入类型的日志来说，代表它的插入位置
KeyType	del_key_	对于删除类型的日志来说，代表它的删除位置
KeyType	old_key_	对于更新类型的日志来说，代表它旧值的删除位置
KeyType	new_key_	对于更新类型的日志来说，代表它新值的更新位置
ValType	ins_val	对于插入类型的日志来说，代表它的插入值
ValType	del_val_	对于删除类型的日志来说，代表它的删除的值
ValType	old_val_	对于更新类型的日志来说，代表它的删除的旧值
ValType	new_val_	对于更新类型的日志来说，代表它的更新的新值

其他变量(用作测试)

```
std::unordered_map<txn_id_t, lsn_t> LogRec::prev_lsn_map_ = {};
lsn_t LogRec::next_lsn_ = 0;
```

变量名称	含义
prev_lsn_map_	unorder_map类型变量，连接事务号和前LSN号
next_lsn_	静态成员变量，用来记录log更新过程中递增的LSN号

作为全局变量，可以很好地维护LSN和对应`Map`,便于测试查看

CreateInsertLog

函数实现：

```
static LogRecPtr CreateInsertLog(txn_id_t txn_id, KeyType
ins_key, ValType ins_val) {
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
d()) {
    prev_lsn = INVALID_LSN ;
```

```
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kInsert,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.ins_key_ = std::move(ins_key);
  log.ins_val_ = ins_val;
  return std::make_shared<LogRec>(log);
}
```

插入日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
)
  {}
```

3. 更新日志应该保存的键值对，这里保存插入的位置和新值，`RollBack`要将该键值对删除

CreateDeleteLog

函数实现：

```
static LogRecPtr CreateDeleteLog(txn_id_t txn_id, KeyType
del_key, ValType del_val) {
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
d()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kDelete,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.del_key_ = std::move(del_key);
  log.del_val_ = del_val;
  return std::make_shared<LogRec>(log);
}
```

删除日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
)
  {}
```

3. 更新日志应该保存的键值对，这里保存删除的位置和旧值，`RollBack`要将该键值对加进去

CreateUpdateLog

函数实现：

```
static LogRecPtr CreateUpdateLog(txn_id_t txn_id, KeyType
old_key, ValType old_val, KeyType new_key, ValType new_val) {
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
d()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kUpdate,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.old_key_ = std::move(old_key);
  log.new_key_ = std::move(new_key);
  log.new_val_ = new_val;
  log.old_val_ = old_val;
  return std::make_shared<LogRec>(log);
}
```

更新日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：

- 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
- 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`

2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id)
  {}
```

3. 更新日志应该保存的键值对，这里保存更新的位置和新旧值，`RollBack`要将该键值对更换为之前的旧值

CreateBeginLog

函数实现：

```
static LogRecPtr CreateBeginLog(txn_id_t txn_id){
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.end()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log = LogRec(LogRecType::kBegin,LogRec::next_lsn_++,prev_lsn,txn_id);
  return std::make_shared<LogRec>(log);
}
```

开始日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id)
   {}
```

CreateCommitLog

函数实现：

```
static LogRecPtr CreateCommitLog(txn_id_t txn_id){
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.end()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log = LogRec(LogRecType::kCommit,LogRec::next_lsn_++,prev_lsn,txn_id);
  return std::make_shared<LogRec>(log);
}
```

提交日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
1  LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
   txn_id):
2
    type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
   )
3    {}
```

CreateAbortLog

函数实现：

```
1  static LogRecPtr CreateAbortLog(txn_id_t txn_id) {
2    lsn_t prev_lsn;
3
    if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
   d()) {
4      prev_lsn = INVALID_LSN ;
5      LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
6    }else{
7      prev_lsn = LogRec::prev_lsn_map_[txn_id];
8      LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
9    }
10   LogRec log =
   LogRec(LogRecType::kAbort,LogRec::next_lsn_++,prev_lsn,txn_id);
11   return std::make_shared<LogRec>(log);
12 }
```

取消日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
)
  {}
```

3. 对于该类事务返回，让上一层函数直接`RollBack`，不解释连招

recovery_manager.h相关

出于实现复杂度的考虑，我们将Recovery Manager模块独立出来，不考虑日志的落盘，用一个unordered_map简易的模拟一个KV Database，并直接在内存中定义一个能够用于插入、删除、更新，事务的开始、提交、回滚的日志结构。`CheckPoint`检查点应包含当前数据库一个完整的状态，该结构已帮大家实现好了。`RecoveryManager`则包含`UndoPhase`和`RedoPhase`两个函数，代表Redo和Undo两个阶段。

`LSN`称为日志的逻辑序列号(log sequence number)，在InnoDB存储引擎中，`LSN`占用8个字节。`LSN`的值会随着日志的写入而逐渐增大。

CheckPoint结构体

```
using KvDatabase = std::unordered_map<KeyType, ValType>;
using ATT = std::unordered_map<txn_id_t, lsn_t>;
struct CheckPoint {
    lsn_t checkpoint_lsn_{INVALID_LSN};
    ATT active_txns_{};
    KvDatabase persist_data_{};

    inline void AddActiveTxn(txn_id_t txn_id, lsn_t last_lsn) { active_txns_[txn_id] = last_lsn; }

    inline void AddData(KeyType key, ValType val) { persist_data_.emplace(std::move(key), val); }
};
```

数据类型	变量名称	含义
lsn_t	checkpoint_lsn_	`checkpoint`处的`LSN`值，此点及以后的`log`需要被处理
std::unordered_map<txn_id_t, lsn_t>	active_txns_	活跃事务`map`，与`txn_id`绑定，表示现在还在活跃的事务
std::unordered_map<KeyType, ValType>	persist_data_	`KVdatabase:value`与`key`绑定，表示数据库中现在已存的键值对

RecoveryManager类

本类实现了数据恢复的全过程，包括`Redo`,`Undo`两个阶段

Init

函数实现：

```
void Init(CheckPoint &last_checkpoint) {
      persist_lsn_ = last_checkpoint.checkpoint_lsn_;
      active_txns_ = std::move(last_checkpoint.active_txns_);
      data_ = std::move(last_checkpoint.persist_data_);
    }
```

初始化状态：

- 更新现在的执行的日志号为`checkpoint`的日志号，标识`Redo`的作用开始序列号为此
- 更新现在的活跃的事务`ATT`，利用`move`直接移送所有权，省去中间变量的生成
- 更新数据库中的键值对，恢复到`CheckPoint`时的数据

RedoPhase

函数实现：

```
void RedoPhase() {
  for(auto log = log_recs_[persist_lsn_];
      ;log = log_recs_[++persist_lsn_]){
    //parse the log
      active_txns_[log->txn_id_] = log->lsn_;
    switch(log->type_){
      case(LogRecType::kInvalid):
        break;
      case(LogRecType::kInsert):
          data_[log->ins_key_] = log->ins_val_;
        break;
      case(LogRecType::kDelete):
          data_.erase(log->del_key_);
        break;
      case(LogRecType::kUpdate):
        data_.erase(log->old_key_);
        data_[log->new_key_]= log->new_val_;
        break;
      case(LogRecType::kBegin):
        break;
      case(LogRecType::kCommit):
          active_txns_.erase(log->txn_id_);
        break;
      case(LogRecType::kAbort):
          for(auto log_roll = log_recs_[active_txns_[log-
>txn_id_]];
              ;log_roll = log_recs_[log_roll->prev_lsn_] )
          {
            switch (log_roll->type_) {
              case(LogRecType::kInsert):
```

```
                    data_.erase(log_roll->ins_key_);
                  break;
                case(LogRecType::kDelete):
                  data_.emplace(log_roll->del_key_,log_roll->del_val_);
                  break;
                case (LogRecType::kUpdate):
                  data_.erase(log_roll->new_key_);
                  data_.emplace(log_roll->old_key_,log_roll->old_val_);
                  break;
                default:
                  break;
              }
              if(log_roll->prev_lsn_ == INVALID_LSN){
                break;
              }
            }
            active_txns_.erase(log->txn_id_);
          break;
        default:
          break;
      }
      if(persist_lsn_>=(lsn_t)(log_recs_.size()-1)){
        break;
      }
    }
}
```

重做事务：

从`persist_lsn_`开始遍历`lsn`直到做完所有事务

- 如果日志是非`Abort`的，那么按照日志本来的类型去做即可，大部分都是做到更改他的键值对
- 如果是`Abort`的，则需要根据现在的`LogRec`要回滚回去直至`Invaild`，做一次回滚操作，将旧值替换回去即可(这里介绍的较为笼统)

UndoPhase

函数实现：

```
void UndoPhase() {
  //abort all the active
  for(auto pair:active_txns_){
    for(auto log_roll = log_recs_[pair.second];
       ;log_roll = log_recs_[log_roll->prev_lsn_] )
    {
      if(log_roll == nullptr){
        break;
      }
      switch (log_roll->type_) {
        case(LogRecType::kInsert):
          data_.erase(log_roll->ins_key_);
          break;
        case(LogRecType::kDelete):
          data_.emplace(log_roll->del_key_,log_roll->del_val_);
          break;
        case (LogRecType::kUpdate):
          data_.erase(log_roll->new_key_);
          data_.emplace(log_roll->old_key_,log_roll->old_val_);
          break;
        default:
          break;
      }
      if(log_roll->prev_lsn_ == INVALID_LSN){
        break;
      }
    }
  }
  active_txns_.clear();
}
```

Undo事务：

对于每个活跃事务进行遍历：

- 找到活跃事务的最新日志，进行类似于`RollBack`的操作
- 直至找到`INVALID_LSN`代表该事务`Undo`成功

遇到的问题及解决方法

这次的数据库设计并没有引入事务模块，也就是我们的本次恢复模块相较于加入事务的数据库较为容易，测试较为独立，Debug较为顺利，现给出我遇到的某些问题和解决方法：

- 本次数据库模块设计出现了共享指针，要熟悉共享指针的使用。参考了以下博客：C++ 智能指针(共享指针、唯一指针、自动指针)-CSDN博客，便于理解。
- 在更新操作时要使用`erase`和`emplace`搭配的方法，用`[]`是过不了测试点的，这是个奇怪的问题，大概率是和他们对于`map`的底层操作有些许不同导致的

思考题

> *本模块中，为了简化实验难度，我们将Recovery Manager模块独立出来。如果不独立出来，真正做到数据库在任何时候断电都能恢复，同时支持事务的回滚，Recovery Manager应该怎样设计呢？此外，CheckPoint机制应该怎样设计呢？*
>
> *注：如果完成了本模块，请在实验报告里完成思考题。思考题占本模块30%的分数，请尽量回答的详细些，比如具体到涉及哪些模块、哪些函数的改动，大致怎样改动。有能力、有时间的同学也可以挑战一下直接在代码上更改。*

Recovery Manager的设计：

- 由于这个模块的思路已经和数据恢复很契合了，就是通过日志来实现故障恢复，周期性地执行建立检查点和保存数据库状态。
- 当事务T在一个检查点之前提交，T对数据库所做的修改已写入数据库写入时间是在这个检查点建立之前或在这个检查点建立之时，在进行恢复处理时，没有必要对事务T执行重做操作。

    a. 从重新开始文件中找到最后一个检查点记录在日志文件中的地址，由该地址在日志文件中找到最后一个检查点记录。

    b. 由该检查点记录得到检查点建立时刻所有正在执行的事务清单ACTIVE-LIST。

        - 建立两个事务队列：UNDO-LIST和REDO-LIST。
        - 把ACTIVE-LIST暂时放入UNDO-LIST队列，REDO队列暂为空。

    c. 从检查点开始正向扫描日志文件，直到日志文件结束。

    - 如有新开始的事务 Ti ，把 Ti 暂时放入UNDO-LIST队列。
    - 如有提交的事务 Tj ，把 Tj 从UNDO-LIST队列移到REDO-LIST队列;直到日志文件结束。

    d. 对UNDO-LIST中的每个事务执行UNDO操作，对REDO-LIST中的每个事务执行REDO操作。

- 只需要保证在每个事务执行的时候都在通过日志维护执行顺序，并且保证检查点和数据保存同时写入磁盘，这样可以保证数据的一致性，便于我们的断电恢复。
- 另外，事务的回滚和Undo的操作是一样的，只需要将事务的日志“补偿”地反方向执行一次，即可得到事务刚开始的数据，达成事务回滚的操作。
- 总之大概率是不用改动此模块的，由于事务模块还没有实现，这里实现的恢复模块只能对串行化事务进行检查点策略的执行表现良好，当接入事务模块时，大概是把恢复日志管理与事务执行策略并行同步执行，另外提供回滚的接口给事务模块即可。

CheckPoint机制：

- 周期性地执行建立检查点和保存数据库状态。
    a. 将当前日志缓冲区中的所有日志记录写入磁盘的日志文件上
    b. 在日志文件中写入一个检查点记录
    c. 将当前数据缓冲区的所有数据记录写入磁盘的数据库中
    d. 把检查点记录在日志文件中的地址写入一个重新开始文件
- 恢复子系统可以定期或不定期地建立检查点,保存数据库状态，最好是采用不定期策略，因为定期的策略不容易控制每次写入磁盘的量，可能单次写入磁盘的操作用时很长，造成数据库的暂时不可用，可以根据日志现在的存储量来实行检查点策略。
- 操作的代码地方应该是在事务模块中，与写入日志的地方一致，统计写入日志的数量来做到不定期check的操作

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6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！

春风得意马蹄疾，一日看尽长安花！

浙江大学

《数据库系统》
实验报告

作业名称	：	MiniSQL
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 **7 LOCK MANAGER**
  - 实验目的
  - 实验环境
  - 实验流程
    - txn.h相关
      - 模块涉及到的C++类
        - C++11的std::thread
        - unordered_set
        - unordered_map
    - txn_manager事务管理器相关
      - 类内函数作用
        - Begin
        - Commit
        - Abort
        - GetTransaction
        - ReleaseLocks
      - 模块涉及到的C++类
        - std::atomic和std::mutex
        - std::mutex
        - std::atomic
        - shared_mutex
    - lock_manager相关
      - LockManager主类
      - LockRequest(Class)
      - LockRequestQueue(Class)
      - 模块涉及到的C++类
        - wait()成员函数
        - notify_all/notify_one
      - 实现函数
        - LockShared
        - LockExclusive
        - LockUpgrade
        - Unlock
        - LockPrepare
        - CheckAbort
        - AddEdge
        - RemoveEdge
        - HasCycle
        - DeleteNode
        - RunCycleDetection
        - DFS_Gwait
  - 遇到的问题及解决方法
  - 思考题

总结

实验名称 7 LOCK MANAGER

实验目的

> *本次实验中，你需要实现Lock Manager模块，从而实现并发的查询，Lock Manager负责追踪发放给事务的锁，并依据隔离级别适当地授予和释放shared(共享)和exclusive(独占)锁。*

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*

实验流程

数据库系统中，事务管理器（Transaction Manager）是负责处理所有与事务相关操作的组件。它是维护数据库ACID属性（原子性、一致性、隔离性、持久性）的关键组件，确保了数据库系统中的事务能够安全、一致且高效地执行。事务管理器主要负责以下几个方面：

1. **事务的边界控制**：事务管理器负责定义事务的开始（BEGIN TRANSACTION）和结束（COMMIT 或 ROLLBACK）。当事务开始时，事务管理器会为其分配所需的资源，并追踪其状态。当事务成功完成时，事务管理器会执行提交操作，将所有更改永久写入数据库。如果事务遇到错误或者需要撤销，事务管理器将执行回滚操作，撤销所有更改。
2. **并发控制**：在允许多个事务同时运行的系统中，事务管理器使用并发控制机制（如锁、时间戳、版本号lsn等）来确保事务不会相互干扰，导致数据不一致。并发控制也包括实现数据库的隔离级别，防止并发事务产生冲突。
3. **恢复管理**：事务管理器还负责实现恢复机制，以保证在系统故障（如崩溃、电源中断）后数据库的一致性和持久性。这通常通过使用日志记录（Logging）和检查点（Checkpointing）等技术来完成。事务日志存储了所有对数据库所做的更改的记录，可以用于恢复操作。
4. **故障处理**：在检测到错误或异常时，事务管理器负责采取适当的行动，例如触发回滚来撤销事务的操作，或者在某些情况下，尝试恢复事务执行。

txn.h相关

该类详细介绍了事务对象应该包含的成员信息，同时也暗示，在事务进行上锁解锁时要进行事务信息方面的修改：

```
class Txn {
 public:
  explicit Txn(txn_id_t txn_id = INVALID_TXN_ID, IsolationLevel
iso_level = IsolationLevel::kRepeatedRead)
      : txn_id_(txn_id), iso_level_(iso_level),
thread_id_(std::this_thread::get_id()) {}

 private:
  txn_id_t txn_id_{INVALID_TXN_ID};
  IsolationLevel iso_level_{IsolationLevel::kRepeatedRead};
  TxnState state_{TxnState::kGrowing};
  std::thread::id thread_id_;
  std::unordered_set<RowId> shared_lock_set_;
  std::unordered_set<RowId> exclusive_lock_set_;
};
```

数据类型	变量名称	代表含义
txn_id_t	txn_id_	事务的唯一标识，
IsolationLevel	iso_level_	隔离等级：目前有三种`kReadUncommitted`，`kReadCommitted`，`kRepeatedRead`
TxnState	state_	在二阶段封锁协议下的事务状态(四种)：`kGrowing`，`kShrinking`，`kCommitted`，`kAborted`
std::thread::id	thread_id_	线程id，有关线程的成员函数可以查看`thread.h`
std::unordered_set	shared_lock_set_	(S锁)共享锁集合：以`unorder_set`形式储存，用法同`unorder_map`，其中内容是所占有的数据项的`Row_Id`
std::unordered_set	exclusive_lock_set_	(X锁)占有锁集合：以`unorder_set`形式储存，用法同`unorder_map`,其中内容是所占有的数据项的`Row_Id`

模块涉及到的C++类

C++11的std::thread

在C中已经有一个叫做pthread的东西来进行多线程编程，但是并不好用（如果你认为句柄、回调式编程很实用，那请当我没说），所以c++11标准库中出现了一个叫作std::thread的东西。

std::thread常用成员函数

- 构造&析构函数

函数	类别	作用
thread() noexcept	默认构造函数	创建一个线程，什么也不做
template <class Fn, class... Args> explicit thread(Fn&& fn, Args&&... args)	初始化构造函数	创建一个线程，以`args`为参数执行`fn`函数
thread(const thread&) = delete	复制构造函数	（已删除）
thread(thread&& x) noexcept	移动构造函数	构造一个与`x`相同的对象，会破坏`x`对象
~thread()	析构函数	析构对象

- 常用成员函数

函数	作用
void join()	等待线程结束并清理资源（会阻塞）
bool joinable()	返回线程是否可以执行join函数
void detach()	将线程与调用其的线程分离，彼此独立执行（此函数必须在线程创建时立即调用，且调用此函数会使其不能被join）
std::thread::id get_id()	获取线程id
thread& operator=(thread &&rhs)	见移动构造函数（如果对象是joinable的，那么会调用`std::terminate()`结果程序）

unordered_set

无序集合(`unordered_set`)是一种使用哈希表实现的无序关联容器，其中键被哈希到哈希表的索引位置，因此插入操作总是随机的。无序集合上的所有操作在平均情况下都具有常数时间复杂度O(1)，但在最坏情况下，时间复杂度可以达到线性时间O(n)，这取决于内部使用的哈希函数，但实际上它们表现非常出色，通常提供常数时间的查找操作。

无序集合可以包含任何类型的键 - 预定义或用户自定义数据结构，但所有键必须是唯一的。它的语法如下：

```
1 std::unordered_set<data_type> name;
```

常用方法

- `size()`和`empty()`:用于获取大小和集合是否为空
- `find()`:用于查找键
- `insert()`和`erase()`:用于插入和删除元素

如果集合中不存在某个键，`find()`函数会返回一个指向`end()`的迭代器，否则会返回指向键位置的迭代器。迭代器充当键值的指针，因此我们可以使用`*`运算符来解引用它们以获取键。

其他方法

FUNCTION NAME	FUNCTION DESCRIPTION
insert()	插入一个新元素
begin()/end()	返回一个迭代器，指向第一个元素/最后一个元素后的理论元素
count()	计算在无序集合容器中特定元素的出现次数
find()	搜索元素
clear()	清空所有元素
cbegin()/cend()	返回一个常量迭代器，指向第一个元素/最后一个元素后的理论元素
bucket_size()	返回无序集合中特定桶(bucket)中的元素总数(元素通过哈希函数映射到不同的桶中)
erase()	移除单个或某个范围内的一系列元素
size()	返回元素数量
swap()	交换两个无序集合容器的值
emplace()	在无序集合容器中插入元素

FUNCTION NAME	FUNCTION DESCRIPTION
max_size()	返回可以容纳的最大元素数量
empty()	检查无序集合容器是否为空
equal_range()	返回包括与给定值相等的所有元素的范围
hash_function()	用于获取容器所使用的哈希函数对象
reserve()	它用于请求容器预留足够的桶数，以容纳指定数量的元素
bucket()	返回特定元素的桶编号
bucket_count()	返回无序集合容器中的总桶数
load_factor()	用于获取当前容器的负载因子。负载因子:元素数量与桶数之比，用于衡量容器的填充程度
rehash()	设置容器的桶数以容纳一定数量的元素
max_load_factor()	获取或设置容器的最大负载因子
emplace_hint()	根据给定的提示位置(iterator)在容器中插入一个新元素
key_eq()	无序集合内部用于比较元素键值相等性的函数对象或谓词类型
max_bucket_count()	获取无序集合容器支持的最大桶数

unordered_map

无序映射(`unordered_map`)是一种关联容器，用于存储由键和映射值组成的元素。键值用于唯一标识元素，而映射值是与键相关联的内容。键和值都可以是任何预定义或用户定义的类型。简而言之，无序映射类似于一种字典类型的数据结构，可以在其中存储元素。

在内部，无序映射使用哈希表来实现，提供的键被哈希成哈希表的索引，因此数据结构的性能在很大程度上取决于哈希函数。但平均而言，从哈希表中搜索、插入和删除的成本是O(1)。

基本使用

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    unordered_map<string, int> umap;

    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
```

```
    umap["Contribute"] = 30;

    for (auto x : umap)
        //Contribute=30 GeeksforGeeks=10 Practice=20
        cout << x.first << "=" << x.second << ' ';
}
```

方法

METHODS/FUNCTIONS	DESCRIPTION
at(K)	返回与元素作为键k相关的值的引用
begin()/end()	返回一个迭代器，指向第一个元素/最后一个元素后的理论元素
bucket(k)	返回键k所在的桶编号，即元素在映射中的位置。
bucket_count()	返回无序映射容器中的总桶数
bucket_size()	返回无序映射中每个桶中的元素数量
count()	计算具有给定键的元素在无序映射中出现的次数
equal_range(k)	返回包括与键k相等的所有元素的范围的边界
find()	搜索元素
empty()	检查无序映射容器是否为空
erase()	从无序映射容器中删除元素

C++11库还提供了用于查看内部使用的桶数、桶大小以及使用的哈希函数和各种哈希策略的函数(和`unordered_set`中类似)，但它们在实际应用中的用处较小。我们可以使用迭代器遍历无序映射中的所有元素。

txn_manager事务管理器相关

在本次实验中，我们提供的`TxnManager`主要负责事务的边界控制、并发控制、故障处理。出于实现复杂度的考虑，同时为了避免各模块耦合太强，前面模块的问题导致后面模块完全无法完成，我们将`TxnManager`模块单独拆了出来。`TxnManager`的代码已经实现好了，支持`Begin()`、`Commit()`、`Abort()`等方法。因为`TxnManager`模块独立，我们在`Commit()`、`Abort()`方法中不需要做其他事情（本来需要维护事务中的写、删除集合，结合`Recovery`模块回滚）。同时我们提供了`Txn`类,里面通过参数控制事务的隔离级别：

- READ_UNCOMMITED
- READ_COMMITTED
- REPEATABLE_READ

Lock Manager 负责检查事务的隔离级别，任何失败的锁操作都将导致事务中止，并同时抛出异常，此时`TxnManager`将捕获该异常并回滚。

```
class TxnManager {
 public:
  explicit TxnManager(LockManager *lock_mgr);
 private:
  LockManager *lock_mgr_{nullptr};
  std::atomic<txn_id_t> next_txn_id_{0};
  /** The transaction map is a global list of all the running transactions in the system. */
  std::unordered_map<txn_id_t, Txn *> txn_map_{};
  std::shared_mutex rw_latch_{};
};
```

数据类型	变量名称	代表含义
LockManager *	lock_mgr_	指向当前数据库的锁处理器，可以通过锁处理器来处理真正底层锁的上锁解除
std::atomic<txn_id_t>	next_txn_id_	原子变量可以方便创建事务时不会少数漏数事务。代表含义是下一个创建事务的id号，在每次创建后会+1
std::unordered_map<txn_id_t, Txn *>	txn_map_	通过事务号来获取实际事务的映射，方法操作较常用的是`[]`、`find()`。主要是找到对应迭代器来进行事务的修改
std::shared_mutex	rw_latch_	`shard_mutex`类型与我们要实现的共享锁、排他锁很相像，此处作用不明，猜测是读写状态存储

这一部分文件将事务管理独立了出来作锁操作的设计与验证，与之前的模块相独立

类内函数作用

Begin

```
Txn *TxnManager::Begin(Txn *txn, IsolationLevel isolationLevel) {
  if (nullptr == txn) {
    txn = new Txn(next_txn_id_++, isolationLevel);
  }
```

```
  std::unique_lock<std::shared_mutex> lock(rw_latch_);
  txn_map_[txn->GetTxnId()] = txn;
  return txn;
}
```

函数作用:

- 实现事务的新建,事务号是根据`next_txn_id_`的递增实现的
- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 添加事务到 `txn_map_`中,供之后函数调用,标识事务的存在

Commit

```
void TxnManager::Commit(Txn *txn) {
  // change state
  txn->SetState(TxnState::kCommitted);
  // release all locks
  ReleaseLocks(txn);
}
```

函数作用:

- 设置事务状态为提交
- 释放事务所占有的所有锁

Abort

```
void TxnManager::Abort(Txn *txn) {
  // change state
  txn->SetState(TxnState::kAborted);
  // release all locks
  ReleaseLocks(txn);
}
```

函数作用:

- 设置事务状态为取消
- 释放事务所占有的所有锁

GetTransaction

```
Txn *TxnManager::GetTransaction(txn_id_t txn_id) {
  std::shared_lock<std::shared_mutex> lock(rw_latch_);
  auto iter = txn_map_.find(txn_id);
  if (iter != txn_map_.end()) {
    return iter->second;
  }
  return nullptr;
}
```

函数作用:

- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 查找对应事务号的事务,如果不存在则返回空指针

ReleaseLocks

```
void TxnManager::ReleaseLocks(Txn *txn) {
  std::unordered_set<RowId> lock_set;
  for (auto o : txn->GetExclusiveLockSet()) {
    lock_set.emplace(o);
  }
  for (auto o : txn->GetSharedLockSet()) {
    lock_set.emplace(o);
  }
  for (auto rid : lock_set) {
    lock_mgr_->Unlock(txn, rid);
  }
}
```

函数作用:

- 请求对 `rw_latch_` 的独占访问。如果 `rw_latch_` 已被其他线程锁定，当前线程将被阻塞，直到锁可用
- 对于事务本身而言，清空自身的存储的锁的集合
- 对于LM而言，执行`Unlock`清空锁集合

模块涉及到的C++类

std::atomic和std::mutex

```
#include <iostream>
#include <thread>
using namespace std;
int n = 0;
void count10000() {
    for (int i = 1; i <= 10000; i++)
        n++;
}
int main() {
    thread th[100];
    // 这里偷了一下懒，用了c++11的foreach结构
    for (thread &x : th)
        x = thread(count10000);
    for (thread &x : th)
        x.join();
    cout << n << endl;
    return 0;
}
1234567891011121314151617181920
```

2次输出结果分别是：

```
991164
996417
12
```

我们的输出结果应该是1000000，可是为什么实际输出结果比1000000小呢？

在上文我们分析过多线程的执行顺序——同时进行、无次序，所以这样就会导致一个问题：多个线程进行时，如果它们同时操作同一个变量，那么肯定会出错。为了应对这种情况，c++11中出现了`std::atomic`和`std::mutex`。

std::mutex

`std::mutex`是 C++11 中最基本的互斥量，一个线程将mutex锁住时，其它的线程就不能操作mutex，直到这个线程将mutex解锁。

mutex的常用成员函数（这里用`mutex`代指`对象`）

函数	作用
void lock()	将mutex上锁。 如果mutex已经被其它线程上锁， 那么会阻塞，直到解锁；如果mutex已经被同一个线程锁住， 那么会产生死锁。
void unlock()	解锁mutex，释放其所有权。 如果有线程因为调用lock()不能上锁而被阻塞，则调用此函数会将mutex的主动权随机交给其中一个线程； 如果mutex不是被此线程上锁，那么会引发未定义的异常。
bool try_lock()	尝试将mutex上锁。 如果mutex未被上锁，则将其上锁并返回true； 如果mutex已被锁则返回false。

std::atomic

> *原子操作是最小的且不可并行化的操作。*

这就意味着即使是多线程，也要像同步进行一样**同步操作**atomic对象，从而省去了mutex上锁、解锁的时间消耗。

std::atomic常用成员函数

函数	类型	作用
atomic() noexcept = default	默认构造函数	构造一个atomic对象（未初始化，可通过atomic_init进行初始化）
constexpr atomic(T val) noexcept	初始化构造函数	构造一个atomic对象，用`val`的值来初始化
atomic(const atomic&) = delete	复制构造函数	（已删除）

shared_mutex

C++17起。`shared_mutex`类是一个同步原语，可用于保护共享数据不被多个线程同时访问。与便于独占访问的其他互斥类型不同，`shared_mutex` 拥有二个访问级别：

- 共享 - 多个线程能共享同一互斥的所有权；
- 独占性 - 仅一个线程能占有互斥。

1）若一个线程已经通过lock或try_lock获取独占锁（写锁），则无其他线程能获取该锁（包括共享的）。尝试获得读锁的线程也会被阻塞。
2）仅当任何线程均未获取独占性锁时，共享锁（读锁）才能被多个线程获取（通过lock_shared 、try_lock_shared ）。
3）在一个线程内，同一时刻只能获取一个锁（共享或独占性）。

成员函数主要包含两大类：**排他性锁定（写锁）和共享锁定（读锁）。**

lock_manager相关

Lock Manager的基本思想是它维护当前活动事务持有的锁。事务在访问数据项之前向 LM 发出锁请求，LM 来决定是否将锁授予该事务，或者是否阻塞该事务或中止事务。LM里定义了两个内部类：`LockRequest` and `LockRequestQueue`.

在你的实现当中，整个数据库系统会存在一个全局的 LM 结构。每当一条事务需要去访问一条数据记录时，借助该全局的LM去获取数据记录上的锁。条件变量可用于阻塞等待直到它们的锁请求得到满足的事务。本次实验中，同学们实现的LM需要支持三种不同的隔离级别。

Note：

- 在锁管理器需要使用死锁检测时，我们建议首先实现一个不包含任何死锁处理的锁管理器，然后在确认其在没有死锁发生时能够正确地进行锁定和解锁后，再添加检测机制。
- 虽然通过确保严格两阶段锁（strict two phase lock）可以实现某些隔离级别，但本次实验的锁管理器实现只需确保两阶段锁的特性。严格两阶段锁的概念将通过执行器和事务管理器中的逻辑来实现。具体需要查看其中的 `Commit` 和 `Abort` 方法。
- 还需要跟踪事务所获取的共享/独占锁，使用 `shared_lock_set_` 和 `exclusive_lock_set_`，这样当 `TransactionManager` 想要提交/中止事务时，LM能够适当地释放它们。

本次实验实现的锁管理器应该在后台运行死锁检测，以中止阻塞事务。更准确地说，这意味着一个后台线程应该定期即时构建一个等待图，并打破任何循环。需要实现并用于循环检测以及测试的API如下：

- `AddEdge(txn_id_t t1, txn_id_t t2)`：在图中从t1到t2添加一条边。如果该边已存在，则无需进行任何操作。
- `RemoveEdge(txn_id_t t1, txn_id_t t2)`：从图中移除t1到t2的边。如果没有这样的边存在，则无需进行任何操作。

- `HasCycle(txn_id_t& txn_id)`：使用深度优先搜索(DFS)算法寻找循环。如果找到循环，`HasCycle`应该将循环中最早事务的id存储在`txn_id`中并返回true。该函数应该返回它找到的第一个循环。如果图中没有循环，`HasCycle`应该返回false。
- `GetEdgeList()`：返回一个元组列表，代表图中的边。一对(t1,t2)对应于从t1到t2的一条边。
- `RunCycleDetection()`：包含在后台运行循环检测的框架代码。需要在此实现循环检测逻辑。

实现完成后，你的代码需要通过`lock_manager_test.cpp`中的所有测试用例。

Note:

- 后台线程应该在每次唤醒时即时构建图表，而不是维护一个图表。等待图应该在每次线程唤醒时构建和销毁。
- 实验中的DFS循环检测算法**必须**是确定性的。为了做到这一点，必须始终选择首先探索最低的事务ID。这意味着在选择从哪个**未探索**的节点运行DFS时，始终选择具有最低事务ID的节点。这也意味着在探索邻居时，按从最低到最高的顺序探索它们。
- 当发现循环时，应该通过将该事务的状态设置为ABORTED来中止**最年轻**的事务以打破循环。
- 当检测线程唤醒时，它负责打破存在的**所有**循环。如果你遵循上述要求，你将总是以**确定性**的顺序找到循环。这也意味着当你构建图时，你**不应该**为已中止的事务添加节点或向已中止的事务绘制边。
- 等待图是一个**有向**图。当一个事务在等待另一个事务时，等待图会画出边。如果多个事务持有一个**共享**锁，一个单独的事务可能会等待多个事务。
- 当一个事务被中止时，确保将事务的状态设置为`ABORTED`并在您的锁管理器中抛出一个异常。事务管理器将负责明确的中止和回滚更改。一个等待锁的事务可能会被后台循环检测线程中止。您必须有一种方法通知等待的事务它们已被中止。

LockManager主类

```
class LockManager {
public:
  enum class LockMode { kNone, kShared, kExclusive };
private:
  /** Lock table for lock requests. */
  std::unordered_map<RowId, LockRequestQueue> lock_table_{};
  std::mutex latch_{};

```

```
  /** Waits-for graph representation. */
  std::unordered_map<txn_id_t, std::set<txn_id_t>> waits_for_{};
  std::unordered_set<txn_id_t> visited_set_{};
  std::stack<txn_id_t> visited_path_{};
  txn_id_t revisited_node_{INVALID_TXN_ID};
  std::atomic<bool> enable_cycle_detection_{false};
  std::chrono::milliseconds cycle_detection_interval_{100};
  TxnManager *txn_mgr_{nullptr};
};
```

数据类型	变量名称	代表含义
std::unordered_map<RowId, LockRequestQueue>	lock_table_	这个包含了对应数据项的上锁请求队列
std::mutex	mutex latch_	与`shared_mutex`不同，这里的数据类型只支持单纯的上锁解锁，没有隔壁的共享排他之分，用法一般是`lock`、`unlock`,
std::unordered_map<txn_id_t, std::set<txn_id_t>>	waits_for_	等待图存储单元，与DS课程教学不同，这里没有使用邻接链表来存储图，而是继续使用我们这一工程常使用的`unorder_map`数据类型。其中等待图的边是指等待关系$T_1 \rightarrow T_2$指T_1等待T_2释放锁。
std::unordered_set<txn_id_t>	visited_set_	利用`unordered_set`存储所有遇到过的事务，事务以事务号标识，可以通过find()来判断有没有经过该点。
std::stack<txn_id_t>	visited_path_	以栈存储经过的点的集合，这里与上面不同的地方是，栈能够真切反映访问的顺序。
txn_id_t	revisited_node_	字面意思是再次遇到的节点，其实就是这个闭环中你最开始搜索的结果，也就是DFS返回的结果。

数据类型	变量名称	代表含义
std::atomic	enable_cycle_detection_	原子变量，即最小的能够并行修改的变量。翻译一下就是这个变量能够实时反映现在的值，而不会受线程阻塞影响。
std::chrono::milliseconds	cycle_detection_interval_	时间常数，用来即时生成等待图使用，时间常数越小，刷新频率越快，死锁检测处理更快，但是也会带来进程的大量调用，影响总执行速度。
TxnManager *	txn_mgr_	事务管理器：能够统筹所有事务的管理器，如果有对事务的修改，请认准该管理器。

在网上查找`std::lock_guard`用于管理`std::mutex`，`std::unique_lock`与`std::shared_lock`管理`std::shared_mutex`，但是在条件变量搭配使用中只能使用`std::unique_lock<std::mutex>`，这样才能达到上锁的效果

LockRequest(Class)

```
 class LockRequest {
   public:
    LockRequest(txn_id_t txn_id, LockMode lock_mode)
        : txn_id_(txn_id), lock_mode_(lock_mode),
granted_(LockMode::kNone) {}

    txn_id_t txn_id_{0};
    LockMode lock_mode_{LockMode::kShared};
    LockMode granted_{LockMode::kNone};
  };

```

数据类型	变量名称	代表含义
txn_id_t	txn_id_	标识所在事务的id
LockMode	lock_mode_	锁的类型，分为共享锁、排他锁、未知节点(应该用不上)

数据类型	变量名称	代表含义
LockMode	granted_	已经获得锁的类型共享锁、排他锁、未知节点(就是指没有获得任何锁)

LockRequest：此类代表由事务（`txn_id`）发出的锁请求。它包含以下成员：

- `txn_id_`：发出请求的事务的标识符。
- `lock_mode_`：请求的锁类型（例如，共享或排他）。
- `granted_`：已授予事务的锁类型。

构造函数使用给定的`txn_id`和`lock_mode`初始化这些成员，默认将`granted_`设置为`LockMode::kNone`

该类型是描述一个事务对某个数据项(具体数据标识row_id在最高类lock_manager中的map关联first中)提出的上锁请求

[Tips]为了设计层级逻辑，这里的数据项信息来源于顶层类`lock_manager`中`lock_table_`相关联项中first值。

LockRequestQueue(Class)

```
class LockRequestQueue {
public:
    using ReqListType = std::list<LockRequest>;
public:
    ReqListType req_list_{};
    std::unordered_map<txn_id_t, ReqListType::iterator>
req_list_iter_map_{};
    std::condition_variable cv_{};
    bool is_writing_{false};
    bool is_upgrading_{false};
    int32_t sharing_cnt_{0};
};
```

数据类型	变量名称	代表含义
ReqListType	req_list_	描述当前数据项存在的上锁请求，使用list来存储锁的请求

数据类型	变量名称	代表含义
std::unordered_map<txn_id_t, ReqListType::iterator>	req_list_iter_map_	已知对应的数据项，跟踪列表中每个请求的迭代器(即跟踪每个请求，`key`值为请求事务的`txn_id`)
std::condition_variable	cv_	条件变量，负责进程的等待和接受`Lamda`函数捕获以达到函数暂停的作用，主要用法是`wait()`
bool	is_writing_	标识该数据项是否有事务在写，通常和数据项上的排他锁数量一致，但是该变量的存在使得我们可以不必使用排他锁的存在与否标识数据项的锁定。但是话又说回来，你也可以用排他锁的存在与否把这个变量代替掉
bool	is_upgrading_	标识现在的进程中是否有锁升级，推测是锁升级中不能进行任何数据改写的事务
int32_t	sharing_cnt_	表明该数据项有多少共享锁占用，一般上锁要+1,释放记得删

LockRequestQueue：此类管理一个锁请求队列，并提供操作它的方法。它使用一个列表（`req_list_`）存储请求，并使用一个`unordered_map(req_list_iter_map_)`跟踪列表中每个请求的迭代器。它还包括一个条件变量（`cv_`）用于同步目的，以及一些标志来管理并发访问：

- `is_writing_`：指示当前是否持有排他性写锁。
- `is_upgrading_`：指示是否正在进行锁升级。
- `sharing_cnt_`：持有共享锁的事务数量的整数计数。

该类提供以下方法：

- `EmplaceLockRequest()`：将新的锁请求添加到队列前端，并在map中存储其迭代器。
- `EraseLockRequest()`：根据`txn_id`从队列和map中移除锁请求。如果成功返回`true`，否则返回`false`。
- `GetLockRequestIter()`：根据`txn_id`检索队列中特定锁请求的迭代器。

[Tips]为了设计层级逻辑，这里的数据项信息来源于顶层类`lock_manager`中`lock_table_`相关联项中first值

模块涉及到的C++类

wait()成员函数

函数声明如下：

```
void wait(std::unique_lock<std::mutex>& lock);
//Predicate 谓词函数，可以普通函数或者lambda表达式
template<class Predicate>
void wait(std::unique_lock<std::mutex>& lock, Predicate pred);
```

`wait`导致当前线程阻塞直至条件变量被通知，或虚假唤醒发生，可选地循环直至满足某谓词。

notify_all/notify_one

notify函数声明如下：

```
void notify_one() noexcept;
```

若任何线程在 *this 上等待，则调用 notify_one 会解阻塞(唤醒)等待线程之一。

```
void notify_all() noexcept;
```

若任何线程在 *this 上等待，则解阻塞（唤醒)全部等待线程

实现函数

LockShared

`LockShared(Txn,RID)`：事务txn请求获取id为rid的数据记录上的共享锁。当请求需要等待时，该函数被阻塞（使用cv_.wait），请求通过后返回True

函数实现：

```
bool LockManager::LockShared(Txn *txn, const RowId &rid) {
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn-
>GetTxnId())!=req.req_list_iter_map_.end()){
```

```
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kShared){
      return true;
    }
  }
  unique_lock<mutex> lock(latch_);
  if(txn->GetIsolationLevel() == IsolationLevel::kReadUncommitted){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockSharedOnReadUncommitted);
  }
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }
  LockPrepare(txn,rid);
  req.EmplaceLockRequest(txn->GetTxnId(),LockMode::kShared);
  if(req.is_writing_ == true){
    req.cv_.wait(lock,[txn,&req]() -> bool{return txn->GetState()==TxnState::kAborted||!req.is_writing_;});
  }
  CheckAbort(txn,req);
  //modify txn
  txn->GetSharedLockSet().emplace(rid);
  //modify lck_manager
  req.sharing_cnt_++;
  //modify request
  req.GetLockRequestIter(txn->GetTxnId())->granted_ = LockMode::kShared;
  return true;
}
```

1. 检查是否已有请求：

   首先，函数通过`lock_table_`查找是否有当前事务的锁请求。如果事务ID在`req_list_iter_map_`中存在，说明事务已经请求了锁。

2. 检查锁是否已授予：

   如果事务的请求存在，函数检查该请求是否已经被授予了共享锁（`LockMode::kShared`）。如果是，函数直接返回`true`。

3. 获取互斥锁：

   使用`unique_lock`来获取`latch_`互斥锁，确保线程安全。

4. 检查事务隔离级别：

   如果事务的隔离级别是`kReadUncommitted`，事务状态将被设置为`kAborted`，并抛出`TxnAbortException`异常。

5. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务正在处于二阶段的下降阶段，这将导致事务被中止，并抛出异常。

6. 准备锁请求：

   调用`LockPrepare`函数来准备锁请求。

7. 创建锁请求：

   在`req`对象中为当前事务创建一个新的共享锁请求。

8. 等待锁请求：

   如果`req`对象中存在写锁请求，当前事务将等待直到它被中止或写锁请求不再存在。

9. 检查事务是否中止：

   调用`CheckAbort`函数检查事务是否已经被中止。

10. 修改事务状态：

    如果事务没有被中止，将`rid`添加到事务的共享锁集合中。

11. 增加共享计数：

    在`req`对象中增加共享锁的计数。

12. 授予共享锁：

    将事务的锁请求状态设置为已授予共享锁。

13. 返回结果：

    函数返回`true`，表示共享锁请求成功。

LockExclusive

`LockExclusive(Txn,RID)`: 事务txn请求获取id为rid的数据记录上的独占锁。当请求需要等待时，该函数被阻塞，请求通过后返回True

函数实现：

```
1  bool LockManager::LockExclusive(Txn *txn, const RowId &rid) {
2    auto& req = lock_table_[rid];
```

```
  if(req.req_list_iter_map_.find(txn->GetTxnId())!=req.req_list_iter_map_.end()){
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kExclusive){
      return true;
    }
  }
  unique_lock<mutex> lock_latch(latch_);
  if(txn->GetIsolationLevel() == IsolationLevel::kReadUncommitted){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockSharedOnReadUncommitted);
  }
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }
  LockPrepare(txn,rid);
  req.EmplaceLockRequest(txn->GetTxnId(),LockMode::kExclusive);
  if(req.is_writing_ == true||req.sharing_cnt_ !=0){
    req.cv_.wait(lock_latch,[txn,&req]() -> bool{
      return txn->GetState()==TxnState::kAborted||(!req.is_writing_&&req.sharing_cnt_==0);
    });
  }
  CheckAbort(txn,req);
  //modify txn
  txn->GetExclusiveLockSet().emplace(rid);
  //modify lck_manager
  req.is_writing_ = true;
  //modify request
  req.GetLockRequestIter(txn->GetTxnId())->lock_mode_ = LockMode::kExclusive;
  req.GetLockRequestIter(txn->GetTxnId())->granted_ = LockMode::kExclusive;
  return true;
}

```

1. 检查现有请求：

   函数首先检查事务是否已经对给定的`RowId`有锁请求。如果存在，并且该请求已经被授予了排他锁，则函数直接返回`true`。

2. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

3. 检查事务隔离级别：

   如果事务的隔离级别是`kReadUncommitted`，事务状态将被设置为`kAborted`，并抛出`TxnAbortException`异常。

4. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务位于二阶段封锁协议的下降阶段，这将导致事务被中止，并抛出异常。

5. 准备锁请求：

   调用`LockPrepare`函数来准备锁请求。

6. 创建排他锁请求：

   在`req`对象中为当前事务创建一个新的排他锁请求。

7. 等待锁请求：

   如果存在写锁（`is_writing_`为`true`）或者有其他事务持有共享锁（`sharing_cnt_`不为0），当前事务将等待直到它被中止，或者写锁不存在且没有其他共享锁。

8. 检查事务是否中止：

   调用`CheckAbort`函数来检查事务是否已经被中止。

9. 修改事务状态：

   如果事务没有被中止，将`rid`添加到事务的排他锁集合中。

10. 修改锁请求：

    将事务的锁模式设置为排他锁，并将锁请求的状态更新为已授予排他锁。

11. 修改锁管理器状态：

    将`req`对象的`is_writing_`标志设置为`true`，表示现在有事务持有排他锁。

12. 返回结果：

    函数返回`true`，表示排他锁请求成功。

LockUpgrade

`LockUpgrad(Txn,RID)`:事务txn请求更新id为rid的数据记录上的独占锁，当请求需要等待时，该函数被阻塞，请求通过后返回True

函数实现：

```
bool LockManager::LockUpgrade(Txn *txn, const RowId &rid) {
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn->GetTxnId())!=req.req_list_iter_map_.end()){
    if(req.GetLockRequestIter(txn->GetTxnId())->granted_==LockMode::kExclusive){
      return true;
    }
  }
  unique_lock<mutex> lock_latch(latch_);
  if(txn->GetState() == TxnState::kShrinking){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kLockOnShrinking);
  }

  if(req.is_upgrading_ == true){
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),AbortReason::kUpgradeConflict);
  }
req.GetLockRequestIter(txn->GetTxnId())->lock_mode_=LockMode::kExclusive;
  if(req.is_writing_ == true||req.sharing_cnt_ >1){
    req.is_upgrading_ = true;
    req.cv_.wait(lock_latch,[txn,&req]() -> bool{
      return txn->GetState()==TxnState::kAborted||(!req.is_writing_&&req.sharing_cnt_==1);
    });
  }
  if(txn->GetState() == TxnState::kAborted){
    req.is_upgrading_ = false;
  }
  CheckAbort(txn,req);
  //modify txn
```

```
    txn->GetSharedLockSet().erase(rid);
    txn->GetExclusiveLockSet().emplace(rid);
    //modify lck_manager
    req.is_writing_= true;
    req.is_upgrading_ = false;
    req.sharing_cnt_--;
    req.is_writing_ = true;
    //modify request
    req.GetLockRequestIter(txn->GetTxnId())->granted_ =
LockMode::kExclusive;
    return true;
}
```

1. 检查现有请求：

   函数首先检查事务是否已经对给定的`RowId`有锁请求。如果存在，并且该请求已经被授予了排他锁，则函数直接返回`true`。

2. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

3. 检查事务状态：

   如果事务状态是`kShrinking`，表示事务正在两阶段封锁协议的下降阶段，这将导致事务被中止，并抛出`TxnAbortException`异常。

4. 检查是否正在升级：

   如果`req`对象的`is_upgrading_`标志为`true`，表示其他事务正在尝试升级锁，当前事务将被中止，并抛出异常。

5. 修改锁请求模式：

   将当前事务的锁请求模式设置为排他锁。

6. 等待锁升级：

   如果存在写锁（`is_writing_`为`true`）或者共享计数大于1（`sharing_cnt_ > 1`），当前事务将设置`is_upgrading_`为`true`并等待直到它被中止，或者没有其他写锁且只有一个共享锁。

7. 检查事务是否中止：

   如果事务状态变为`kAborted`，在退出等待之前将`is_upgrading_`设置回`false`。

8. 检查事务是否中止：

   再次检查事务是否已经被中止，如果是，则不需要进行后续操作。

9. 检查并处理中止：

调用`CheckAbort`函数来检查事务是否已经被中止。

10. 修改事务锁集合：

    从事务的共享锁集合中移除`rid`，并将`rid`添加到事务的排他锁集合中。

11. 修改锁管理器状态：

    将`req`对象的`is_writing_`设置为`true`，表示现在有事务持有排他锁，并将`is_upgrading_`设置为`false`，表示锁升级完成。同时减少共享计数`sharing_cnt_`。

12. 授予排他锁：

    将事务的锁请求状态更新为已授予排他锁。

13. 返回结果：

    函数返回`true`，表示锁升级成功。

Unlock

`Unlock(Txn,RID)`：释放事物txn在rid数据记录上的锁。注意维护事务的状态，例如该操作中事务的状态可能会从 `GROWING` 阶段变为 `SHRINKING` 阶段（提示：查看 `transaction.h` 中的方法）。此外，当需要某种方式来通知那些等待中的事务，我们可以使用`notify_all()`方法

函数实现：

```
bool LockManager::Unlock(Txn *txn, const RowId &rid) {
  unique_lock<mutex> lock_latch(latch_);
  auto& req = lock_table_[rid];
  if(req.req_list_iter_map_.find(txn->GetTxnId()) ==
req.req_list_iter_map_.end()){
    return false;
  }
  if(txn->GetState() == TxnState::kGrowing&&txn-
>GetIsolationLevel() != IsolationLevel::kReadCommitted){
    txn->SetState(TxnState::kShrinking);
  }
  //modify request
  switch(req.GetLockRequestIter(txn->GetTxnId())->granted_){
    case LockMode::kShared:
      req.sharing_cnt_--;
      break;
    case LockMode::kExclusive:
      req.is_writing_ = false;
```

```
17        break;
18      default:
19        break;
20    }
21    txn->GetSharedLockSet().erase(rid);
22    txn->GetExclusiveLockSet().erase(rid);
23    req.GetLockRequestIter(txn->GetTxnId())->granted_ =
LockMode::kNone;
24    req.cv_.notify_all();
25    return true;
26  }
```

1. 获取互斥锁：

   使用`unique_lock`来获取名为`latch_`的互斥锁，确保在修改锁表时的线程安全。

2. 检查事务请求存在：

   函数首先检查事务是否对给定的`RowId`有锁请求。如果没有，函数直接返回`false`。

3. 检查事务状态：

   如果事务状态是`kGrowing`且隔离级别不是`kReadCommitted`，则将事务状态更新为`kShrinking`。

4. 修改锁请求：

   根据事务持有的锁类型（共享或排他），执行不同的操作：

   - 如果是共享锁（`LockMode::kShared`），减少共享计数`sharing_cnt_`。
   - 如果是排他锁（`LockMode::kExclusive`），将`is_writing_`标志设置为`false`。

5. 从事务锁集合中移除：

   从事务的共享锁集合和排他锁集合中移除对应的`RowId`。

6. 重置锁请求状态：

   将事务的锁请求状态设置为无锁（`LockMode::kNone`）。

7. 通知等待的事务：

   使用条件变量`cv_`的`notify_all`方法通知所有等待该锁的事务，因为锁已经被释放。

8. 返回结果：

   函数返回`true`，表示锁已成功释放。

LockPrepare

`LockPrepare(Txn,RID)`：检测txn的state是否符合预期，并在`lock_table_`里创建rid和对应的队列

函数实现：

```
void LockManager::LockPrepare(Txn *txn, const RowId &rid) {
  if(txn->GetState()!= TxnState::kShrinking){
    if(lock_table_.find(rid) == lock_table_.end()){
      lock_table_.emplace(piecewise_construct,
forward_as_tuple(rid), forward_as_tuple());
    }
  }else {
    txn->SetState(TxnState::kAborted);
    throw TxnAbortException(txn->GetTxnId(),
AbortReason::kLockOnShrinking);
  }
}
```

C++11 中 std::piecewise_construct 的使用

1. 检查事务状态：

   首先，函数检查事务是否处于`kShrinking`状态，即事务是否正在缩小其影响范围。如果事务正在下降，那么不允许进行新的锁请求。

2. 事务未处于下降状态：

   如果事务不在缩小状态，函数继续检查`lock_table_`

   中是否已经存在对应`rid`的锁请求条目：如果不存在，使用`emplace`方法在`lock_table_`中创建一个新的条目。这里使用`piecewise_construct`和`forward_as_tuple`是为了使用完美转发，允许构造函数接受不同类型的参数。

3. 事务处于下降状态：

   如果事务处于`kShrinking`状态，函数将事务状态设置为`kAborted`，并抛出`TxnAbortException`异常。这表示在事务下降期间尝试进行锁请求是不允许的，并且请求将被中止。

4. 异常信息：

   异常包含了事务ID和中止原因`kLockOnShrinking`，有助于调用者理解事务为何被中止。

CheckAbort

`CheckAbort(Txn, LockRequestQueue)`：检查txn的state是否是abort，如果是，做出相应的操作

函数实现：

```
void LockManager::CheckAbort(Txn *txn,
LockManager::LockRequestQueue &req_queue) {
  //because LockX & LockS remove the other information after
'check'
  //so we don't correct the other information
  if(txn->GetState() == TxnState::kAborted){
    req_queue.EraseLockRequest(txn->GetTxnId());
    throw TxnAbortException(txn-
>GetTxnId(),AbortReason::kDeadlock);
  }
}
```

1. 检查事务状态：

   函数首先检查传入的事务（`txn`）的状态。如果事务状态是`kAborted`，表示事务已经被中止。

2. 处理中止的事务：

   如果事务已被中止，执行以下操作：

   - 调用`req_queue`的`EraseLockRequest`方法，传入事务ID，从锁请求队列中移除该事务的锁请求。这一步是为了清理资源，避免中止事务继续占用锁资源。

3. 抛出异常：

   抛出`TxnAbortException`异常，异常中包含了事务ID和中止原因`kDeadlock`。这里使用`kDeadlock`作为中止原因可能是出于简化处理的考虑，但实际中止原因可能因多种情况而异，例如违反隔离级别规则、锁等待超时等。

4. 在排他锁（`LockX`）和共享锁（`LockS`）操作后，某些信息会在检查之后被移除，因此在`CheckAbort`中不需要对这些信息进行修正。

AddEdge

函数实现：

```
void LockManager::AddEdge(txn_id_t t1, txn_id_t t2) {
  waits_for_[t1] .emplace(t2);
}
```

向等待图中插入边，利用`emplace`向`map`中插入元素。

RemoveEdge

函数实现：

```
void LockManager::RemoveEdge(txn_id_t t1, txn_id_t t2) {
  waits_for_[t1].erase(t2);
}
```

向等待图中插入边，利用`erase`向`map`中删除元素。

HasCycle

函数实现：

```
bool LockManager::HasCycle(txn_id_t &newest_tid_in_cycle) {
  visited_set_.clear();
  stack<txn_id_t> empty_stack;
  visited_path_.swap(empty_stack);
  revisited_node_ = INVALID_TXN_ID;
  std::set<txn_id_t> all_txn;
  for(auto& iter:waits_for_){
    all_txn.emplace(iter.first);
    for(auto waited_node:iter.second){
      all_txn.emplace(waited_node);
    }
  }
  for(auto& wait_item:all_txn){
    if(DFS_Gwait(wait_item)){
      newest_tid_in_cycle = revisited_node_;
      while (!visited_path_.empty() && revisited_node_ !=
visited_path_.top()) {
        newest_tid_in_cycle = std::max(newest_tid_in_cycle,
visited_path_.top());
        visited_path_.pop();
      }
```

```
      return true;
    }
  }
  return false;
}
```

1. 初始化和重置状态：

   清空`visited_set_`，这是存储已访问事务的集合。

   将`visited_path_`与一个空栈`empty_stack`交换，以重置访问路径栈。

   将`revisited_node_`设置为一个无效的事务ID（`INVALID_TXN_ID`），用于记录在死锁循环中重复访问到的节点。

2. 收集所有事务ID：

   创建一个集合`all_txn`，用于存储所有事务的ID。

   遍历`waits_for_`映射，将每个等待事务的发起者和被等待的事务ID添加到`all_txn`中。

3. 检测每个事务：

   遍历`all_txn`中的每个事务ID`wait_item`。

4. 深度优先搜索检测死锁：

   对每个事务ID调用`DFS_Gwait`函数进行深度优先搜索，以检测是否存在等待循环。

5. 死锁检测结果：

   如果`DFS_Gwait`返回`true`，表示存在死锁循环。

   将`newest_tid_in_cycle`设置为当前循环中检测到的最近事务ID（`revisited_node_`）。

6. 更新最近事务ID：

   从栈`visited_path_`中弹出元素，直到栈为空或栈顶元素等于`revisited_node_`。

   在弹出的过程中，使用`std::max`函数更新`newest_tid_in_cycle`，确保它始终是循环中最新出现的事务ID。

7. 返回死锁结果：

   如果在任何事务上检测到死锁，函数返回`true`。

   如果遍历完所有事务都没有检测到死锁，函数返回`false`。

DeleteNode

函数实现：

```
void LockManager::DeleteNode(txn_id_t txn_id) {
    waits_for_.erase(txn_id);

    auto *txn = txn_mgr_->GetTransaction(txn_id);

    for (const auto &row_id: txn->GetSharedLockSet()) {
        for (const auto &lock_req:
lock_table_[row_id].req_list_) {
            if (lock_req.granted_ == LockMode::kNone) {
                RemoveEdge(lock_req.txn_id_, txn_id);
            }
        }
    }
    for (const auto &row_id: txn->GetExclusiveLockSet()) {
        for (const auto &lock_req:
lock_table_[row_id].req_list_) {
            if (lock_req.granted_ == LockMode::kNone) {
                RemoveEdge(lock_req.txn_id_, txn_id);
            }
        }
    }
}
```

1. 删除等待关系：

   首先，函数从`waits_for_`映射中删除与`txn_id`相关的所有等待关系。这意味着如果其他事务正在等待这个事务释放锁，这些等待关系将不再存在。

2. 获取事务对象：

   通过事务管理器（`txn_mgr_`）获取与`txn_id`关联的事务对象。

3. 处理共享锁集合：

   遍历事务的共享锁集合`GetSharedLockSet`，对于集合中的每个行ID`row_id`：

   - 遍历该行ID在锁表（`lock_table_`）中的锁请求列表`req_list_`。
   - 对于每个锁请求，如果锁请求的授予状态是`LockMode::kNone`（表示没有授予锁），则调用`RemoveEdge`函数来删除从当前锁请求事务`lock_req.txn_id_`指向`txn_id`的依赖边。

4. 处理排他锁集合：

   类似于共享锁集合的处理，遍历事务的排他锁集合`GetExclusiveLockSet`，对于集合中的每个行ID：

   - 遍历该行ID在锁表中的锁请求列表。
   - 对于每个锁请求，如果锁请求的授予状态是`LockMode::kNone`，则删除指向`txn_id`的依赖边。

5. 删除事务的锁请求：

   通过上述步骤，函数删除了所有与`txn_id`相关的锁请求，这些请求要么因为事务结束而不再需要，要么因为事务没有获得锁而需要从依赖图中移除。

RunCycleDetection

函数实现：

```
void LockManager::RunCycleDetection() {
  while(enable_cycle_detection_==true) {
    this_thread::sleep_for(cycle_detection_interval_);
    unique_lock<mutex> lock_latch(latch_);
    waits_for_.clear();
    // construct the graph
    // find all txn
    unordered_map<RowId, set<txn_id_t>> data_txn_locked;
    unordered_map<RowId, set<txn_id_t>> data_txn_unlocked;
    for (auto& iter : lock_table_) {
      //data_id
      for (auto& request : iter.second.req_list_) {
        //txn_id
        //1.grantee locked
        if(request.granted_ != LockMode::kNone) {
          data_txn_locked[iter.first].emplace(request.txn_id_);
        }else{

 data_txn_unlocked[iter.first].emplace(request.txn_id_);
        }
      }
    }
    for (auto &iter : lock_table_) {
      //data_id
```

```
      for (auto& request : iter.second.req_list_) {
        //txn_id
        //2.addedge
        if(request.granted_ == LockMode::kNone) {
          for(auto grant:data_txn_locked[iter.first]) {
            AddEdge(request.txn_id_, grant);
          }
        }
      }
    }
    txn_id_t txn_id = INVALID_TXN_ID;
    //detect cycle
    while(HasCycle(txn_id)==true){
      DeleteNode(txn_id);
      txn_mgr_->GetTransaction(txn_id)->SetState(TxnState::kAborted);
      for(auto item:data_txn_unlocked){
        if(item.second.find(txn_id)!=item.second.end()){
          lock_table_[item.first].cv_.notify_all();
        }
      }
    }
  }
}
```

1. 循环检测死锁：

   只要`enable_cycle_detection_`标志为`true`，函数就会继续运行。

2. 休眠等待：

   使用`this_thread::sleep_for`函数根据`cycle_detection_interval_`指定的时间间隔休眠，以减少检测频率，避免过度占用资源。

3. 获取互斥锁：

   使用`unique_lock`获取`latch_`互斥锁，以确保在构建图和检测死锁时的线程安全。

4. 清空等待关系：

   清空`waits_for_`映射，为构建新的依赖图做准备。

5. 构建依赖图：

   创建两个哈希表`data_txn_locked`和`data_txn_unlocked`，分别存储每个数据项上已授予锁和未授予锁的事务集合。

遍历`lock_table_`中的每个锁请求：

- 如果事务已被授予锁（`granted_ != LockMode::kNone`），将其添加到`data_txn_locked`中。
- 如果事务未被授予锁（`granted_ == LockMode::kNone`），将其添加到`data_txn_unlocked`中。

6. 添加依赖边：

   再次遍历`lock_table_`中的每个锁请求：

   - 如果事务未被授予锁，对于数据项上已被授予锁的每个事务，添加一条从等待锁的事务到授予锁的事务的依赖边（`AddEdge`）。

7. 死锁检测：

   使用`HasCycle`函数进行死锁检测，如果发现死锁，记录死锁循环中的最近事务ID（`txn_id`）。

8. 处理死锁：

   如果检测到死锁，执行以下操作：

   - 调用`DeleteNode`函数删除死锁事务的所有锁请求和依赖边。
   - 设置死锁事务的状态为`kAborted`。
   - 遍历`data_txn_unlocked`，如果事务在未解锁事务集合中，通知所有等待该数据项的事务（`notify_all`）。

9. 循环处理死锁：

   继续检测死锁，直到没有发现死锁为止。

DFS_Gwait

函数实现：

```
bool LockManager::DFS_Gwait(txn_id_t txn_id){
   if(visited_set_.find(txn_id) != visited_set_.end()){
     for(auto& item : visited_set_){
       revisited_node_ = revisited_node_>item?
revisited_node_:item;
     }
     txn_id = revisited_node_;
     return true;
   }
   visited_set_.emplace(txn_id);
   visited_path_.push(txn_id);
    for(auto iter:waits_for_[txn_id]){
```

```
12          if(DFS_Gwait(iter)){
13            return true;
14          }
15        }
16        visited_path_.pop();
17        visited_set_.erase(txn_id);
18        return false;
19      }
```

1. **检查事务是否已访问：**

   函数首先检查当前事务ID（`txn_id`）是否已经在`visited_set_`中。这用于避免无限递归和循环。

2. **处理已访问事务：**

   如果事务已访问，函数遍历`visited_set_`中的所有事务，并使用一个变量`revisited_node_`来存储已经访问过的的事务ID。然后，将`txn_id`设置为这个事务ID，并返回`true`，表示存在循环等待。

3. **标记事务为已访问：**

   如果事务未访问，将其添加到`visited_set_`中。

4. **添加事务到访问路径：**

   将当前事务ID添加到`visited_path_`栈中，用于记录访问路径。

5. **递归检查等待事务：**

   遍历当前事务等待的事务列表（`waits_for_[txn_id]`），对每个等待的事务递归调用`DFS_Gwait`函数。

6. **检查死锁：**

   如果在递归调用中发现死锁（即返回`true`），则当前函数也返回`true`。

7. **回溯并清理状态：**

   在完成当前事务的所有递归检查后，从`visited_path_`栈中弹出当前事务ID，并从`visited_set_`中移除该事务ID，以进行回溯。

遇到的问题及解决方法

在本节中遇到的问题主要是对头文件的阅读和条件变量的使用，等待图的遍历DFS就可以解决，难度不大。这里主要说明下对于LM的设计想法：

- 先准备查询事务本身的状态，如果不符合上锁解锁的条件要抛出异常
- 确定可以修改之后，按照修改事务本身状态->LM状态->等待图构建所需变量

&写报告写太累了，就不展开细说我遇到的很多大坑了（

思考题

> *本模块中，为了简化实验难度，我们将Lock Manager模块独立出来。如果不独立出来，做到并发查询期间根据指定的隔离级别进行事务的边界控制，考虑模块3中B+树并发修改的情况，需要怎么设计？*
>
> *注：如果完成了本模块，请在实验报告里完成思考题。思考题占本模块30%的分数，请尽量回答的详细些，比如具体到涉及哪些模块、哪些函数的改动，大致怎样改动。有能力、有时间的同学也可以挑战一下直接在代码上更改。*

要控制好不同隔离等级下的事务边界控制，最重要是掌握好对数据的上锁解锁，我们可以借助教材提到的意向锁的机制来实现对数据的访问权限授予，意向锁的作用可以不仅限与页表等多级数据，也可以完全适用到B+树

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

当我们在利用B+索引进行插入、删除、查询的时候，我们可以利用意向锁的机制查看当前数据是否空闲，来实现不同隔离等级下的对数据的访问

具体操作函数位置放在索引查找的函数的地方，新建变量标识现在的意向锁，再通过条件变量控制当前线程的并发控制，当此时的意向锁解绑或是降级之后，通过`notify`方法通知并行进程。

总结

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7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！

愿旅途臻若星河，灿如骄阳！