《数据库系统原理》实验报告（4）

题目：miniob 实验 2

学号		姓名		日期	11/22/2023

实验环境：

Windows 10 Docker MiniOB

赛题选择：

Drop table 功能的实现

要求：

1. 删除表，清除表相关的资源。

2. 要删除所有与表关联的数据，不仅包括磁盘中的文件，还包括内存中的索引等数据。

测试用例示例：

```
1.  create table t(id int, age int);
2.  create table t(id int, name char);
3.  drop table t;
4.  create table t(id int, name char);
```

实验步骤及结果截图：

1. 在 stmt.h 的 StmtType 类中检查 DROP_TABLE 指令，在 yacc 编译器中检查 YYSYMBOL_drop_table 终结符

```
1.  YYSYMBOL_drop_table = 60,              /* drop_table  */
```

2. 首先在 executor 调用处添加入口函数，保证在输入指令为 drop table 时能正常调用相应的处理函数。在 execute_stage.h 头文件添加定义，在 execute_stage.cpp 文件中添加 drop table 的调用过程

execute_stage.h

```
1.  RC do_drop_table(SQLStageEvent *sql_event);
```

execute_stage.cpp

```
1.  RC ExecuteStage::do_drop_table(SQLStageEvent *sql_event) {
2.    const DropTable &drop_table = sql_event->query()->sstr.drop_table;
3.    SessionEvent *session_event = sql_event->session_event();
4.    Db *db = session_event->session()->get_current_db();
5.    RC rc = db->drop_table(drop_table.relation_name);
6.    if (rc == RC::SUCCESS) {
7.      session_event->set_response("SUCCESS\n");
8.    } else {
9.      session_event->set_response("FAILURE\n");
10.   }
11.   return rc;
12. }
```

*** 首先从 sql_event 中提取关于删除表的信息。sql_event 是一个 SQLStageEvent 类型的指针，通过它可以访问 SQL 事件的信息。这一行代码从 SQL 事件中获取了一个 DropTable 类型的对象，命名为 drop_table，其中包含了有关删除表的详细信息。**

*** 接着从 sql_event 中获取与会话事件相关的信息。从会话事件中获取当前数据库的指针。通过**

session_event->session()获取与事件相关联的会话对象后调用 get_current_db()函数来获取当前数据库的指针，将其存储在 db 中。

* 调用数据库对象的 drop_table 函数，传递了要删除的表的名称，该信息是从 drop_table 对象中获取的。函数返回一个 RC 类型的结果，表示操作的执行结果。

* 最后检查操作是否成功，如果成功则设置回哈时间相应为 SUCCESS，否则为 FAIL

3. 在 db.h 中添加 drop_table 声明，在 db.cpp 中添加 drop_table 的实现过程

```
1.      RC drop_table(const char *table_name);
```

实现过程：

```
1.    RC Db::drop_table(const char *table_name)
2.    {
3.      RC rc = RC::SUCCESS;
4.      if (opened_tables_.count(table_name) == 0) {
5.          LOG_WARN("%s has not been opened before.", table_name);
6.          return RC::SCHEMA_TABLE_NOT_EXIST;
7.      }
8.
9.      std::string table_file_path = table_meta_file(path_.c_str(), table_name);
10.     Table *table = opened_tables_[table_name];
11.     rc = table->drop(table_file_path.c_str(), table_name, path_.c_str());
12.     if (rc != RC::SUCCESS) {
13.         LOG_ERROR("Failed to drop table %s.", table_name);
14.         return rc;
15.     }
16.
17.     opened_tables_.erase(table_name);
18.     delete table;
19.     LOG_INFO("Drop table success. table name=%s", table_name);
20.     return RC::SUCCESS;
21.   }
```

* 首先检查数据库中是否存在指定名称的表。opened_tables_是一个 std::unordered_map，用于跟踪已经打开的表。如果指定的表名不在 opened_tables_中，表示表不存在。如果表不存在，记录一条警告日志，表示指定的表在之前并未被打开，则返回一个错误码，表示模式（schema）中不存在该表。

* 接下来从 opened_tables_中获取指定表名对应的 Table 对象，表示要执行删除操作的表。调用表对象的 drop 方法，传递表的元数据文件路径、表名和数据库路径。如果删除操作不成功，记录错误日志并返回相应的错误码。

* opened_tables_.erase(table_name);：从 opened_tables_中移除已删除的表。释放删除的表的内存空间。

* 如果删除表的操作成功，记录一条信息日志，返回操作的结果，表示删除表的操作成功。

4. 在 table.h 和 table.cpp 中添加 Table 类方法 drop

```
RC Table::drop(const char* table_name)
{
 RC rc = RC::SUCCESS;
 PersistHandler persistHandler;
 //删除表的元数据
 std::string data_file = base_dir_ + "/" + table_name + ".data";
 std::string table_file = base_dir_ + "/" + table_name + ".table";
 rc = persistHandler.remove_file(data_file.c_str());
 if (rc != RC::SUCCESS) {
 return rc;
 }
 rc = persistHandler.remove_file(table_file.c_str());
 if (rc != RC::SUCCESS) {
 return rc;
 }
 for (auto index : indexes_) {
 std::string index_file = base_dir_ + "/" + table_name + "-" +
index->index_meta().name() + ".index" ;
 rc = persistHandler.remove_file(index_file.c_str());
 if (rc != RC::SUCCESS) {
 return rc;
 }
 }
 rc = data_buffer_pool_ ->close_file();
 if (rc != RC::SUCCESS) {
 return rc;
 }
 record_handler_->close();
 return rc;
}
```

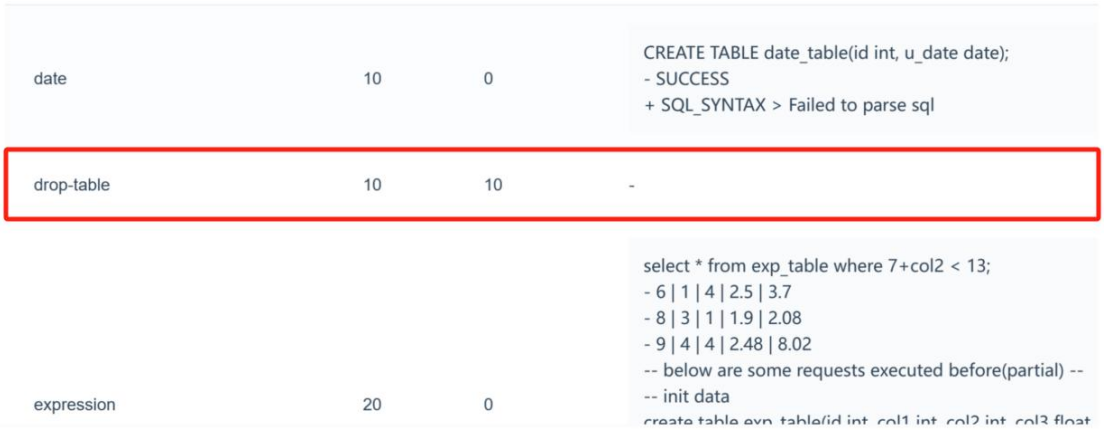
* 对传入的表名进行了有效性检查，确保表名不为空。
* 通过尝试以写入方式打开表文件来检查表是否已经存在。如果文件存在，表示表尚未创建，会记录相应的错误日志并返回表不存在的错误码。
* 调用 remove_record_handler 函数来删除表的记录处理器或处理表的相关操作，然后通过缓冲池管理器删除表的数据文件。如果这些操作有任何失败，都会记录错误日志并返回相应的错误码。
* 删除表的元数据文件，如果删除操作不成功，同样记录错误日志并返回一个 IO 错误码。

过程中会记录一系列信息日志，包括开始删除表和成功删除表的消息。

该方法中构造了持久化的 handler 对象，删除了 .data ， .table, .index 文件，最后完成表的删除。

5. 测试检验

使用 MiniOB 平台测试用例测试，提测记录截图如下：



数据库添加 drop table 功能验证成功。

出现的问题：

1. 忘记删除 .index 文件

实现时忘记删除 .index 文件，导致删除表单没有删除完全

2. 文件打开错误未完全处理：

在代码中，通过尝试以写入方式打开表文件来检查表是否已经存在。如果文件打开失败，但失败原因并非因为文件已经存在（EEXIST），则直接关闭文件描述符并返回 RC::SCHEMA_TABLE_NOT_EXIST 错误码，可能导致对错误的不准确处理。

解决方案：

1. 添加 .index 文件删除部分

```
for (auto index : indexes_) {
std::string index_file = base_dir_ + "/" + table_name + "-" +
index->index_meta().name() + ".index" ;
rc = persistHandler.remove_file(index_file.c_str());
if (rc != RC::SUCCESS) {
return rc;
}
}
```

2. 在捕获到文件打开错误后，应该检查错误码，确保只有在文件已经存在的情况下才返回 RC::SCHEMA_TABLE_NOT_EXIST 错误码。对于其他文件打开错误，可能需要进一步处理，例如记录详细的错误信息以便后续调试，或者返回其他适当的错误码。