《数据库系统原理》实验报告（4）					
题目：MINIOB 实验一					
学号		姓名		日期	11/8/2023

实验环境：

实验环境：

Docker MariaDB

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz　　3.11 GHz

实验步骤及结果截图：

1. 配置环境

```
# git clone https://github.com/oceanbase/miniob.git
Cloning into 'miniob'...
remote: Enumerating objects: 4427, done.
remote: Counting objects: 100% (2339/2339), done.
remote: Compressing objects: 100% (509/509), done.
remote: Total 4427 (delta 1882), reused 1847 (delta 1830), pack-reused 2088
Receiving objects: 100% (4427/4427), 26.46 MiB | 3.44 MiB/s, done.
Resolving deltas: 100% (2889/2889), done.
# ls
docker  miniob
# cd miniob
# ls
benchmark  build.sh  cmake  CMakeLists.txt  CODE_OF_CONDUCT.md  CONTRIBUTING.md  deps  docker  docs  Doxyfile  etc  License  NOTICE  README.md  src  test  tools  unittest
# bash build.sh --make -j4
build.sh --make -j4
create soft link for build_debug, linked by directory named build
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 --log-level=STATUS /root/miniob -DCMAKE_BUILD_TYPE=Debug -DDEBUG=ON
```

2. 启动服务

```
# ls
benchmark  bin  CMakeCache.txt  CMakeFiles  cmake_install.cmake  compile_commands.json  CTestTestfile.cmake  deps  lib  Makefile  miniob  miniob.sock  observer.log.20231108  src  test  tools  unittest
# ./bin/observer -s miniob.sock -f ../etc/observer.ini &
# /bin/sh: 14: .bin/observer: not found
^C
[1] + Done(127)                 .bin/observer -s miniob.sock -f ../etc/observer.ini
# ls
benchmark  bin  CMakeCache.txt  CMakeFiles  cmake_install.cmake  compile_commands.json  CTestTestfile.cmake  deps  lib  Makefile  miniob  miniob.sock  observer.log.20231108  src  test  tools  unittest
# ./bin/observer -s miniob.sock -f ../etc/observer.ini &
# Successfully load ../etc/observer.ini

#
#
# ./bin/obclient -s miniob.sock
miniob > help
Commands
show tables;
desc `table name`;
create table `table name` (`column name` `column type`, ...);
create index `index name` on `table` (`column`);
insert into `table` values(`value1`,`value2`);
update `table` set column=value [where `column`=`value`];
delete from `table` [where `column`=`value`];
select [ * | `columns` ] from `table`;
miniob >
```

3. **创建一张表，包括学号，姓名，成绩**

```
miniob > create table Scores (id int, name char(10), score float);
SUCCESS
```

4. 向这张表里面插入几行数据

```
miniob > select * from Scores;
id | name | score
miniob > insert into Scores values(2251435, '李明浩', 81.2);
SUCCESS
miniob > insert into Scores values(2210465, '赵毅斌', 91.3);
SUCCESS
miniob > insert into Scores values(2332133, '刘孔阳', 56.3);
SUCCESS
miniob > insert into Scores values(2231435, '王亚伟', 73.2);
SUCCESS
miniob > insert into Scores values(1950723, '孙鹏翼', 89.2);
SUCCESS
miniob >
```

5. 使用 select 语句展示学号姓名

```
miniob > select * from Scores;
id | name | score
2251435 | 李明浩 | 81.2
2210465 | 赵毅斌 | 91.3
2332133 | 刘孔阳 | 56.3
2231435 | 王亚伟 | 73.2
1950723 | 孙鹏翼 | 89.2
miniob >
```

6. 尝试修改条目

```
miniob > update Scores set score = 91.3 where id=2251435;
SUCCESS
miniob > update Scores set score = 87.2 where id=2231435;
SUCCESS
miniob > select * from Scores;
id | name | score
2251435 | 李明浩 | 81.2
2210465 | 赵毅斌 | 91.3
2332133 | 刘孔阳 | 56.3
2231435 | 王亚伟 | 73.2
1950723 | 孙鹏翼 | 89.2
```

可以观察到，操作可以成功进行，但查询表发现并没有更新，说明条目并没有被真正修改。

7. 尝试删除条目

```
miniob > delete from Scores where name='赵毅斌'
SUCCESS
miniob > delete from Scores where name='孙鹏翼'
SUCCESS
miniob > select * from Scores;
id | name | score
2251435 | 李明浩 | 81.2
2332133 | 刘孔阳 | 56.3
2231435 | 王亚伟 | 73.2
miniob >
```

删除成功

8. Miniob 源码阅读 droptable

"miniob" 的 "DROP TABLE" 语句用于删除数据库中的表及其相关资源。与 "CREATE TABLE" 相反，DROP TABLE 会清理掉所有与该表相关联的资源，包括描述表的文件、数据文件以及索引等相关数据和文件。

当 sql 流转到 default_storge 阶段的时候，在处理 sql 的函数中，新增一个 drop_table 的 case。在执行 create table t 时，系统将：新建 t.table 的文件、新建 t.data 文件存储数据、同时创建索引记录。删除表时需要删除 t.table 文件,t.data 文件和相关联的索引文件

同时由于 buffer pool 的存在，在新建表和插入数据的时候，会将数据写入 buffer pool 缓存。所以执行 drop table 不仅需要删除文件，也需要清空 buffer pool ，防止在数据没落盘的时候，再建立同名表，仍然可以查询到数据。

如果建立了索引，比如 t_id on t(id)，那么也会新建一个 t_id.index 文件，也需要删除这个文件。Drop table 时需要将以上内容完全清空。

在 miniob 的源码中，default_storage_stage.cpp 中的处理 SQL 语句的 case 中增加

```
  case SCF_DROP_TABLE: {
    const DropTable& drop_table = sql->sstr[sql->q_size-1].drop_table; // 拿到要 drop 的表
    rc = handler_->drop_table(current_db,drop_table.relation_name); // 调用 drop table 接口，drop table 要在 handler 中实现
    snprintf(response,sizeof(response),"%s\n", rc == RC::SUCCESS ? "SUCCESS" : "FAILURE"); // 返回结果，带不带换行符都可以
  }break;
```

SCF_DROP_TABLE 首先从一个名为 sql 的对象中提取了要删除表的相关信息。接下来调用 handler_ 对象的 drop_table 函数，尝试从当前数据库中删除指定的表，并将操作结果存储在 rc 变量中。最后使用 snprintf 函数构建一个响应字符串，表示操作的结果，如果操作成功，则响应为 "SUCCESS"，否则为 "FAILURE"，并在字符串末尾添加了一个换行符。响应字符串用于向用户或客户端报告表的删除操作是否成功。

在 default_handler.cpp 文件中，实现 handler 的 drop_table 接口：

```
RC DefaultHandler::drop_table(const char *dbname, const char *relation_name) {
  Db *db = find_db(dbname);  // 这是原有的代码，用来查找对应的数据库，不过目前只有一个库
  if(db == nullptr) {
    return RC::SCHEMA_DB_NOT_OPENED;
  }
  return db->drop_table(relation_name); // 直接调用 db 的删掉接口
}
```

在 drop table 函数接口中，函数通过调用 find_db 方法来查找指定名称的数据库对象。这个方法可能会在一个存储了多个数据库对象的列表中执行查找操作，以确定要删除的表所在的数据库对象。如果找不到对应的数据库对象（即 db 为 nullptr），函数将返回一个 RC::SCHEMA_DB_NOT_OPENED 的错误代码，表示数据库未成功打开或不存在。如果找到了对应的数据库对象，函数将使用 db 对象调用 drop_table 方法，将指定的表名称作为参数传递给该方法。这将直接在找到的数据库对象上执行删除表的操作，并返回操作的结果。最后函数将返回 drop_table 方法的结果，作为整个 drop_table 函数的返回值。调用方可以通过该返回值判断删除表操作是否成功，并根据需要进行进一步处理。

在 db.cpp 中，实现 drop_table 接口

```
RC Db::drop_table(const char* table_name)
{
    auto it = opened_tables_.find(table_name);
    if (it == opened_tables_.end())
    {
        return SCHEMA_TABLE_NOT_EXIST; // 找不到表，要返回错误，测试程序中也会校验这种场景
    }
    Table* table = it->second;
    RC rc = table->destroy(path_.c_str()); // 让表自己销毁资源
    if(rc != RC::SUCCESS) return rc;
    opened_tables_.erase(it); // 删除成功的话，从表 list 中将它删除
    delete table;
    return RC::SUCCESS;
}
```

Drop_table 函数是执行 drop table 操作的主函数，如果找到了指定的表，方法将获取对应的 Table 对象，它代表了要删除的表。接着方法调用了 table->destroy(path_.c_str()); 来让表对象自己执行资源的销毁操作。这可能包括关闭与表相关的文件或其他资源。如果销毁操作成功（即 rc != RC::SUCCESS ），方法将返回销毁操作的结果，通常是 RC::SUCCESS 表示成功。如果销毁操作失败，将返回相应的错误代码。如果表的销毁操作成功，方法将从 opened_tables_ 中删除该表的记录（opened_tables_.erase(it);），并释放相应的 Table 对象的内存（delete table;）。则该表就从数据库中移除。

table.cpp 中清理文件和相关数据

```
RC Table::destroy(const char* dir) {
    RC rc = sync();//刷新所有脏页

    if(rc != RC::SUCCESS) return rc;

    std::string path = table_meta_file(dir, name());
    if(unlink(path.c_str()) != 0) {
        LOG_ERROR("Failed to remove meta file=%s, errno=%d", path.c_str(), errno);
        return RC::GENERIC_ERROR;
    }
```

```
    std::string data_file = std::string(dir) + "/" + name() + TABLE_DATA_SUFFIX;
    if(unlink(data_file.c_str()) != 0) { // 删除描述表元数据的文件
        LOG_ERROR("Failed to remove data file=%s, errno=%d", data_file.c_str(),
errno);
        return RC::GENERIC_ERROR;
    }

    std::string text_data_file = std::string(dir) + "/" + name() +
TABLE_TEXT_DATA_SUFFIX;
    if(unlink(text_data_file.c_str()) != 0) { // 删除表实现 text 字段的数据文件（后
续实现了 text case 时需要考虑，最开始可以不考虑这个逻辑）
        LOG_ERROR("Failed to remove text data file=%s, errno=%d",
text_data_file.c_str(), errno);
        return RC::GENERIC_ERROR;
    }

    const int index_num = table_meta_.index_num();
    for (int i = 0; i < index_num; i++) {  // 清理所有的索引相关文件数据与索引元数据
        ((BplusTreeIndex*)indexes_[i])->close();
        const IndexMeta* index_meta = table_meta_.index(i);
        std::string index_file = index_data_file(dir, name(),
index_meta->name());
        if(unlink(index_file.c_str()) != 0) {
            LOG_ERROR("Failed to remove index file=%s, errno=%d", index_file.c_str(),
errno);
            return RC::GENERIC_ERROR;
        }
    }
    return RC::SUCCESS;
}
```

首先调用 sync() 来刷新所有脏页，确保表的数据已经被完全写入磁盘，以避免数据的不一致性。如果刷新操作失败，方法将返回相应的错误代码，通常是 RC::GENERIC_ERROR。第二步方法构建了表元数据文件的路径，并尝试使用 unlink 函数删除该文件。如果删除失败，方法记录错误信息并返回 RC::GENERIC_ERROR。构建了表数据文件和表文本数据文件的路径，并分别尝试使用 unlink 函数删除这两个文件。同样，如果删除任何一个文件失败，方法将记录错误信息并返回 RC::GENERIC_ERROR。

接着方法遍历表的所有索引，关闭每个索引，并删除与每个索引相关的文件。对于每个索引，它调用 ((BplusTreeIndex*)indexes_[i])->close(); 来关闭索引。然后，它获取索引元数据，并构建索引数据文件的路径，使用 unlink 函数尝试删除该文件。如果删除失败，方法记录错误信息并返回 RC::GENERIC_ERROR。

出现的问题：

1. 姓名输入格式

在插入数据时，输入的姓名需要加上引号 表明该数据为一个字符串 （varchar(10)）

```
miniob > insert into Scores values(2251435, 李明浩, 81.2);
insert into Scores values(2210465, 赵毅斌, 91.3);
insert into Scores values(2332133, 刘孔阳, 56.3);
insert into Scores values(2231435, 王亚伟, 73.2);
insert into Scores values(1950723, 孙鹏翼, 89.2);
SQL_SYNTAX > Failed to parse sql
```

解决方案：

在插入时需要插入的是字符串：

```
miniob > insert into Scores values(2251435, '李明浩', 81.2);
insert into Scores values(2210465, '赵毅斌', 91.3);
insert into Scores values(2332133, '刘孔阳', 56.3);
insert into Scores values(2231435, '王亚伟', 73.2);
insert into Scores values(1950723, '孙鹏翼', 89.2);
SUCCESS
```