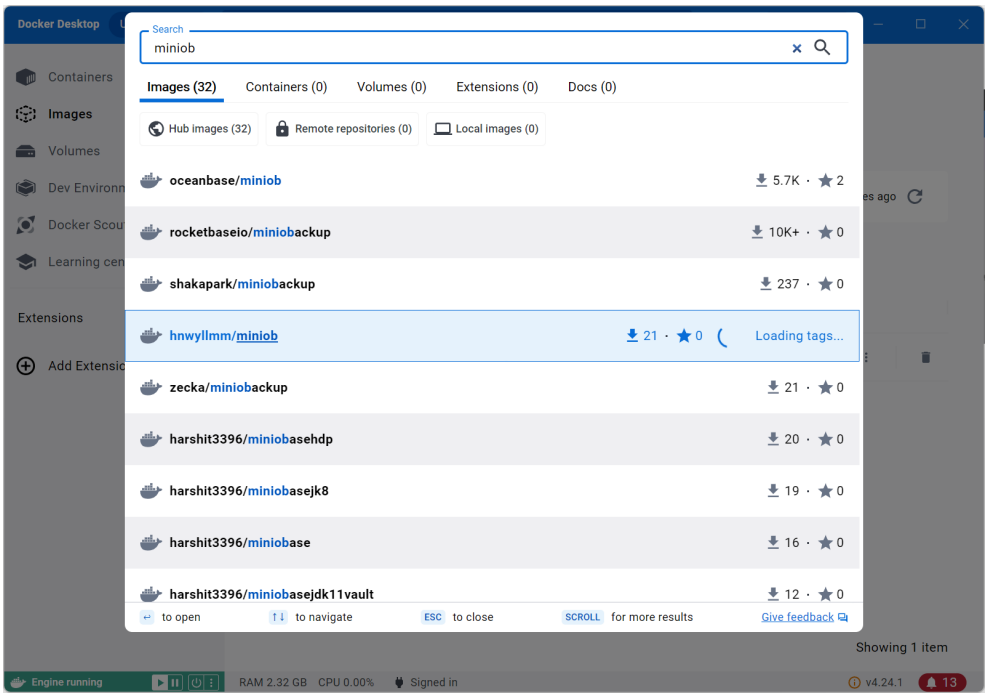
《数据库系统原理》实验报告（4）					
题目：数据库安全性					
学号	2154312	姓名	郑博远	日期	2023.11.8

实验环境：Docker MiniOB

实验步骤及结果截图：

1. 在 Docker 中建立 miniob 环境：

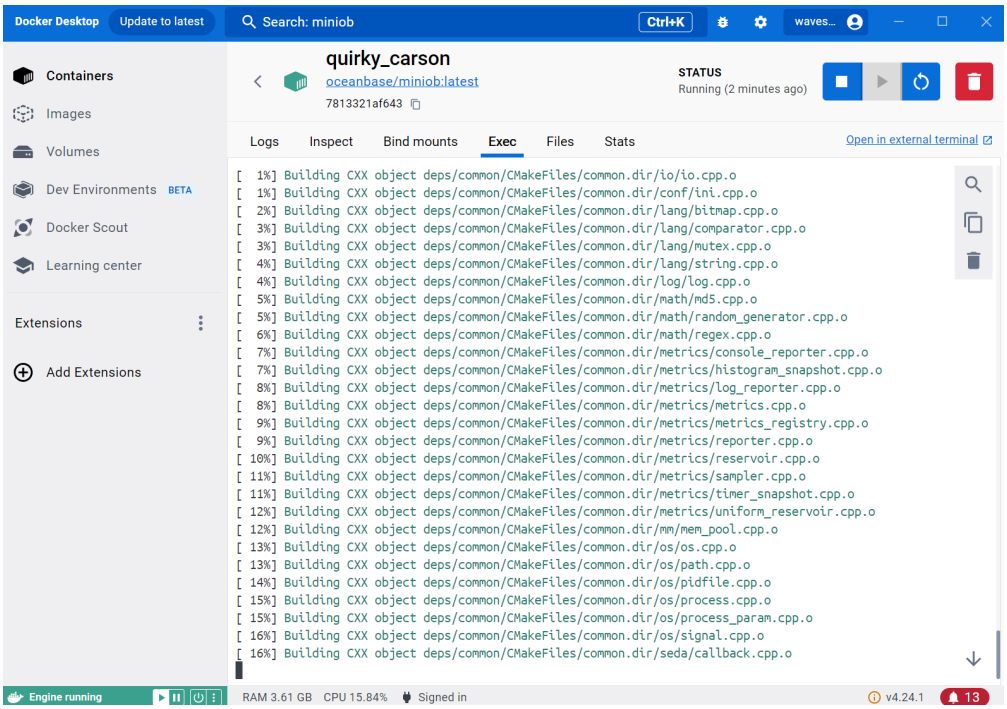
(1) 在 Docker 顶部下拉搜索 miniob，pull 并 run 官方镜像：



(2) 进入容器，下载并编译 miniOB：

```
# git clone https://github.com/oceanbase/miniob.git
Cloning into 'miniob'...
remote: Enumerating objects: 4416, done.
remote: Counting objects: 100% (2333/2333), done.
remote: Compressing objects: 100% (504/504), done.
remote: Total 4416 (delta 1879), reused 1843 (delta 1829), pack-reused 2083
Receiving objects: 100% (4416/4416), 26.45 MiB | 618.00 KiB/s, done.
Resolving deltas: 100% (2883/2883), done.
```

(3) 进入 miniob 目录，使用 bash build.sh --make -j4 进行编译：



(4) 编译完成后，进入 build 目录，在后台启动服务端。

```
# cd build
# ./bin/observer -s miniob.sock -f ../etc/observer.ini &
# Successfully load ../etc/observer.ini
```

(5) 启动服务端后，使用 ./bin/obclient -s miniob.sock 启动 miniOB 客户端。

```
# ./bin/obclient -s miniob.sock
miniob >
```

2. 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
```

3. 向这张表里面插入几行数据：

```
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(2331435, '王亚伟', 73.2);
SUCCESS
miniob > insert into Scores values(1950723, '孙鹏翼', 89.2);
SUCCESS
```

4. 使用 select 语句展示学号，姓名：

```
miniob > select id, name from Scores;
id | name
2251435 | 李明浩
2210465 | 赵毅斌
2332133 | 刘孔阳
2331435 | 王亚伟
1950723 | 孙鹏翼
```

5. 尝试修改指定行的成绩如下表所示，能否成功？为什么？

```
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
2331435 | 王亚伟 | 73.2
1950723 | 孙鹏翼 | 89.2
```

显示“Success”，但查表显示数据未修改成功。查阅 MiniOB 源码后发现，如下方的“update_stmt.cpp”文件中可以观察到，新建的 stmt 对象为 nullptr，注释显示该功能待开发。

```
15  #include "sql/stmt/update_stmt.h"
16
17  UpdateStmt::UpdateStmt(Table *table, Value *values, int value_amount)
18      : table_(table), values_(values), value_amount_(value_amount)
19  {}
20
21  RC UpdateStmt::create(Db *db, const UpdateSqlNode &update, Stmt *&stmt)
22  {
23    // TODO
24    stmt = nullptr;
25    return RC::INTERNAL;
26  }
```

6. 删除赵毅斌和孙鹏翼的记录。

```
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
2331435 | 王亚伟 | 73.2
```

7. 对 miniob 源码进行阅读，主要选取一个功能（如 create table、insert、delete 等）进行分析理解，做简要报告（不超过两页）。

选择 create table 进行源码分析理解：

1. 词法分析、语法分析规则详见“lex_sql.l”、“yacc_sql.y”文件中，此处由于篇幅不赘述；
2. 在“src/observer/sql/stmt/”下可以找到“create_table_stmt.cpp”，此处由于篇幅也不进行赘述；
3. create table 属于命令执行类型 SQL，在“src/observer/sql/executor/”下可以找到“create_table_executor.cpp”，下面对其源码加上注释进行解释如下：

```
RC CreateTableExecutor::execute(SQLStageEvent *sql_event)
{
  Stmt *stmt = sql_event->stmt();
  Session *session = sql_event->session_event()->session();

  // 检查传入的 SQL 语句是否是 CREATE TABLE 类型
  ASSERT(stmt->type() == StmtType::CREATE_TABLE,
         "create table executor can not run this command: %d", static_cast<int>(stmt->type()));

  // 将 SQL 语句转换为 CREATE TABLE 语句
  CreateTableStmt *create_table_stmt = static_cast<CreateTableStmt *>(stmt);

  // 获取要创建表的属性数量
  const int attribute_count = static_cast<int>(create_table_stmt->attr_infos().size());

  // 获取表的名称
  const char *table_name = create_table_stmt->table_name().c_str();

  // 调用会话的方法来创建表
  RC rc = session->get_current_db()->create_table(table_name, attribute_count,
create_table_stmt->attr_infos().data());

  return rc;
}
```

4. 继续分析上述代码中的 create_table 方法，其所在位置为“src/observer/storage/db/db.cpp”中：

```
RC Db::create_table(const char *table_name, int attribute_count, const AttrInfoSqlNode
*attributes)
{
  RC rc = RC::SUCCESS;
  // 检查表名是否已经存在
  if (opened_tables_.count(table_name) != 0) {
    LOG_WARN("%s has been opened before.", table_name);
    return RC::SCHEMA_TABLE_EXIST; // 返回表已存在的错误代码
  }

  // 构建表文件的路径，可以移到 Table 模块中
  std::string table_file_path = table_meta_file(path_.c_str(), table_name);
  Table *table = new Table();
  int32_t table_id = next_table_id_++;
  // 调用 Table 类的 create 方法来创建新表
  rc = table->create(table_id, table_file_path.c_str(), table_name, path_.c_str(),
attribute_count, attributes);
  if (rc != RC::SUCCESS) {
    LOG_ERROR("Failed to create table %s.", table_name);
    delete table;
    return rc; // 返回创建表失败的错误代码
  }
  // 将新表添加到已打开的表列表中
  opened_tables_[table_name] = table;
  LOG_INFO("Create table success. table name=%s, table_id:%d", table_name, table_id);
  return RC::SUCCESS; // 返回成功状态
}
```

5. 上述代码调用 Table 类的 create 方法来创建新表，其所在位置为“src/observer/storage/table/table.cpp”中，对对应源码进行注释分析如下：

```
RC Table::create(int32_t table_id, const char *path, const char *name, const char *base_dir, int
attribute_count, const AttrInfoSqlNode attributes[])
{
```

```
  // 检查表 ID 是否合法
  if (table_id < 0) {
    LOG_WARN("invalid table id. table_id=%d, table_name=%s", table_id, name);
    return RC::INVALID_ARGUMENT;
  }

  // 检查表名是否为空
  if (common::is_blank(name)) {
    LOG_WARN("Name cannot be empty");
    return RC::INVALID_ARGUMENT;
  }
  LOG_INFO("Begin to create table %s:%s", base_dir, name);

  // 检查属性信息是否有效
  if (attribute_count <= 0 || nullptr == attributes) {
    LOG_WARN("Invalid arguments. table_name=%s, attribute_count=%d, attributes=%p", name,
attribute_count, attributes);
    return RC::INVALID_ARGUMENT;
  }

  RC rc = RC::SUCCESS;

  // 使用文件路径创建或打开表文件，检查表文件是否已经存在
  int fd = ::open(path, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0600);
  if (fd < 0) {
    if (EEXIST == errno) {
      LOG_ERROR("Failed to create table file, it has been created. %s, EEXIST, %s", path,
strerror(errno));
      return RC::SCHEMA_TABLE_EXIST; // 表文件已存在的错误代码
    }
    LOG_ERROR("Create table file failed. filename=%s, errmsg=%d:%s", path, errno,
strerror(errno));
    return RC::IOERR_OPEN; // 文件打开失败的错误代码
  }

  close(fd);

  // 初始化表元数据
  if ((rc = table_meta_.init(table_id, name, attribute_count, attributes)) != RC::SUCCESS) {
    LOG_ERROR("Failed to init table meta. name:%s, ret:%d", name, rc);
    return rc; // 初始化表元数据失败的错误代码
  }

  // 打开表文件并将元数据序列化到文件中
  std::fstream fs;
  fs.open(path, std::ios_base::out | std::ios_base::binary);
  if (!fs.is_open()) {
    LOG_ERROR("Failed to open file for write. file name=%s, errmsg=%s", path, strerror(errno));
    return RC::IOERR_OPEN; // 打开文件失败的错误代码
  }

  table_meta_.serialize(fs);
  fs.close();

  // 创建表的数据文件
  std::string data_file = table_data_file(base_dir, name);
  BufferPoolManager &bpm = BufferPoolManager::instance();
  rc = bpm.create_file(data_file.c_str());
  if (rc != RC::SUCCESS) {
    LOG_ERROR("Failed to create disk buffer pool of data file. file name=%s",
data_file.c_str());
    return rc; // 创建数据文件失败的错误代码
  }

  // 初始化记录处理程序
  rc = init_record_handler(base_dir);
  if (rc != RC::SUCCESS) {
    LOG_ERROR("Failed to create table %s due to init record handler failed.",
data_file.c_str());
    // 不需要删除数据文件
    return rc; // 初始化记录处理程序失败的错误代码
  }

  base_dir_ = base_dir;
  LOG_INFO("Successfully create table %s:%s", base_dir, name);
  return rc; // 返回成功状态
}
```

出现的问题：

1. 使用 miniOB 时 SQL 语句分行而报错：

```
miniob > create table Scores(
SQL_SYNTAX > Failed to parse sql
```

2. 字符串未用引号包裹：

```
miniob > insert into Scores values(2251435, 李明浩, 81.2);
SQL_SYNTAX > Failed to parse sql
```

解决方案：

1. miniOB 中的 SQL 语句在一行中写完；
2. 字符串用引号进行包裹。