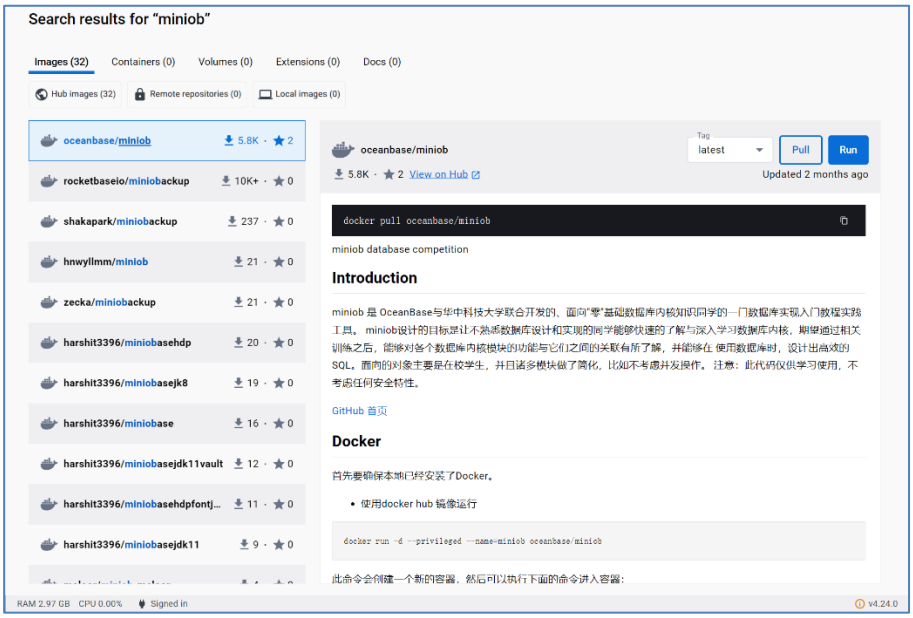
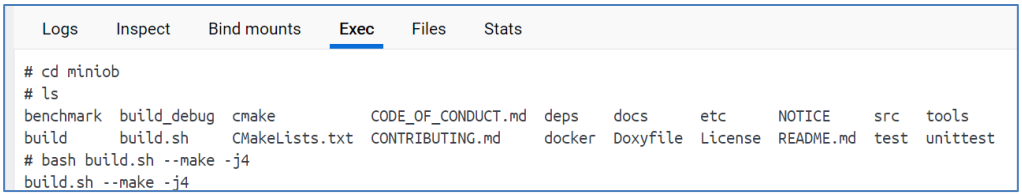
《数据库系统原理》实验报告（四）					
题目：MINIOB 实验一					
学号	2151140	姓名	王谦	日期	2023.11.12

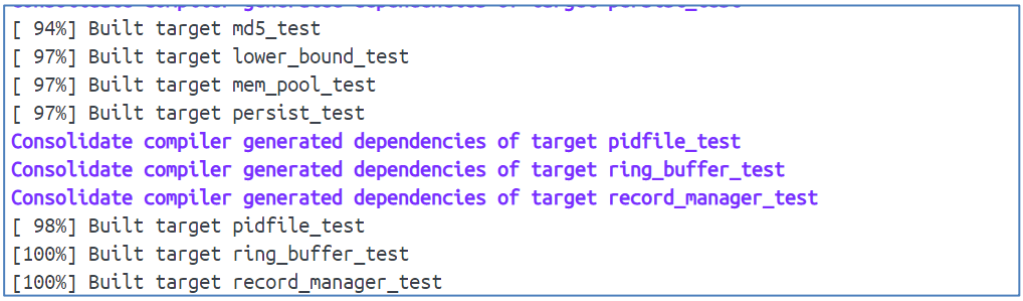
实验环境：Docker + miniob

实验步骤及结果截图：

miniob 安装运行







```
# cd build
# ./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 >
```

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
```

```
miniob > desc Scores;
Field | Type | Length
id | ints | 4
name | chars | 10
score | floats | 4
```

向这张表里面插入几行数据

使用 select 语句展示学号，姓名

```
miniob > insert into Scores values(2210465, '赵毅斌', 91.3);
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
miniob >
```

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
2231435 | 王亚伟 | 73.2
1950723 | 孙鹏翼 | 89.2
miniob >
```

不能成功，miniob 没有实现该功能

删除赵毅斌和孙鹏翼的记录

```
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 > delete from Scores where id = 2210465;
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
1950723 | 孙鹏翼 | 89.2
miniob > delete from Scores where id = 1950723;
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 >
```

对 miniob 源码进行阅读，主要选取功能 create table 进行分析理解， 做简要报告：

1.ExecuteStage::handle_request 函数处理一个 request 请求，通过 sql 解析，发现这是一个创建表的 sql 语句，进入 do_create_table 函数

miniob / src / observer / sql / parser / parse_stage.cpp ↑ Top

Code Blame 62 lines (49 loc) · 1.77 KB Raw

```
29
30  RC ParseStage::handle_request(SQLStageEvent *sql_event)
31  {
32    RC rc = RC::SUCCESS;
33
34    SqlResult *sql_result = sql_event->session_event()->sql_result();
35    const std::string &sql = sql_event->sql();
36
37    ParsedSqlResult parsed_sql_result;
38
39    parse(sql.c_str(), &parsed_sql_result);
40    if (parsed_sql_result.sql_nodes().empty()) {
41      sql_result->set_return_code(RC::SUCCESS);
42      sql_result->set_state_string("");
43      return RC::INTERNAL;
44    }
45
46    if (parsed_sql_result.sql_nodes().size() > 1) {
47      LOG_WARN("got multi sql commands but only 1 will be handled");
48    }
49
50    std::unique_ptr<ParsedSqlNode> sql_node = std::move(parsed_sql_result.sql_nodes().front());
51    if (sql_node->flag == SCF_ERROR) {
52      // set error information to event
53      rc = RC::SQL_SYNTAX;
54      sql_result->set_return_code(rc);
55      sql_result->set_state_string("Failed to parse sql");
56      return rc;
57    }
58
59    sql_event->set_sql_node(std::move(sql_node));
60
61    return RC::SUCCESS;
62  }
```

2.do_create_table 函数

定义一个 CreateTable 变量，通过输入的 sql 语句得到创建表的名字、字段数量、字段类型数据。

获取 sql 语句的 SessionEvent 变量

获取当前的数据库类型（Db）

进入 db->create_table 函数，传入创建表的名字、字段数量、字段类型

```
161  RC DefaultHandler::create_table(
162      const char *dbname, const char *relation_name, int attribute_count, const AttrInfoSqlNode *attributes
163  {
164    Db *db = find_db(dbname);
165    if (db == nullptr) {
166      return RC::SCHEMA_DB_NOT_OPENED;
167    }
168    return db->create_table(relation_name, attribute_count, attributes);
169  }
170
```

3.db->create_table 函数

参数检查，判断数据表名字是否重复

获取当前数据库存放的位置，构造出新创建表的元数据存储路径

建立一个新的 table 变量

调用 table->create 函数，传入元数据存储路径，数据表名字，数据库的路径，字段数和字段类型

函数结束之后，将当前表名加入到记录已打开数据表的映射中

```
79  RC Db::create_table(const char *table_name, int attribute_count, const AttrInfoSqlNode *attributes)
80  {
81    RC rc = RC::SUCCESS;
82    // check table_name
83    if (opened_tables_.count(table_name) != 0) {
84      LOG_WARN("%s has been opened before.", table_name);
85      return RC::SCHEMA_TABLE_EXIST;
86    }
```

4.table->create 函数

（1）再次进行参数检查，检查数据表名字，检查字段数，字段类型

```
53  RC Table::create(int32_t table_id,
54                   const char *path,
55                   const char *name,
56                   const char *base_dir,
57                   int attribute_count,
58                   const AttrInfoSqlNode attributes[])
59  {
60    if (table_id < 0) {
61      LOG_WARN("invalid table id. table_id=%d, table_name=%s", table_id, name);
62      return RC::INVALID_ARGUMENT;
63    }
64
65    if (common::is_blank(name)) {
66      LOG_WARN("Name cannot be empty");
67      return RC::INVALID_ARGUMENT;
68    }
69    LOG_INFO("Begin to create table %s:%s", base_dir, name);
70
71    if (attribute_count <= 0 || nullptr == attributes) {
72      LOG_WARN("Invalid arguments. table_name=%s, attribute_count=%d, attributes=%p", name, attribute_count, attributes);
73      return RC::INVALID_ARGUMENT;
74    }
75
76    RC rc = RC::SUCCESS;
77
```

（2）调用

int fd = ::open(path, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0600);

打开元数据存储路径，根据传入的元数据存储位置，创建一个可读可写的文件

```
78    // 使用 table_name.table记录一个表的元数据
79    // 判断表文件是否已经存在
80    int fd = ::open(path, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0600);
81    if (fd < 0) {
82      if (EEXIST == errno) {
83        LOG_ERROR("Failed to create table file, it has been created. %s, EEXIST, %s", path, strerror(errno));
84        return RC::SCHEMA_TABLE_EXIST;
85      }
86      LOG_ERROR("Create table file failed. filename=%s, errmsg=%d:%s", path, errno, strerror(errno));
87      return RC::IOERR_OPEN;
88    }
89
90    close(fd);
```

（3）调用

table_meta_.init(name, attribute_count, attributes))

初始化表格元数据，包括表格的 ID、名称和属性信息。如果初始化失败，则返回相应的错误码，同时删除已经创建的表格文件。

```
92        // 创建文件
93        if ((rc = table_meta_.init(table_id, name, attribute_count, attributes)) != RC::SUCCESS) {
94          LOG_ERROR("Failed to init table meta. name:%s, ret:%d", name, rc);
95          return rc;  // delete table file
96        }
97
```

（4）调用

std::fstream fs;

fs.open(path, std::ios_base::out | std::ios_base::binary);

打开元数据文件，并且通过二进制的方式进行写操作

调用 table_meta_.serialize(fs);

将元数据进行序列化然后写入文件

```
98        std::fstream fs;
99        fs.open(path, std::ios_base::out | std::ios_base::binary);
100       if (!fs.is_open()) {
101         LOG_ERROR("Failed to open file for write. file name=%s, errmsg=%s", path, strerror(errno));
102         return RC::IOERR_OPEN;
103       }
```

```
105       // 记录元数据到文件中
106       table_meta_.serialize(fs);
107       fs.close();
```

（5）

通过数据库存放位置和数据表名，获得当前数据表文件存放位置

实例化一个 BufferPoolManager 变量，然后调用 create_file 函数，创建数据表文件。

```
109       std::string data_file = table_data_file(base_dir, name);
110       BufferPoolManager &bpm = BufferPoolManager::instance();
111       rc = bpm.create_file(data_file.c_str());
112       if (rc != RC::SUCCESS) {
113         LOG_ERROR("Failed to create disk buffer pool of data file. file name=%s", data_file.c_str());
114         return rc;
115       }
```

（6）

init_record_handler(base_dir);

初始化 record_handler 变量，设置 table 类中的 base_dir_为当前数据库的存储位置

```
117       rc = init_record_handler(base_dir);
118       if (rc != RC::SUCCESS) {
119         LOG_ERROR("Failed to create table %s due to init record handler failed.", data_file.c_str());
120         // don't need to remove the data_file
121         return rc;
122       }
```

```
124         base_dir_ = base_dir;
125         LOG_INFO("Successfully create table %s:%s", base_dir, name);
126         return rc;
127       }
```

（7）

返回 SUCCESS