《数据库系统原理》实验报告（4）					
题目：miniob 实验一					
学号	2152118	姓名	史君宝	日期	2023.11.04

实验环境：基于 docker 的 miniob 数据库。

实验步骤及结果截图：

（1）创建并运行 minob：

编译 miniob：

```
# cd miniob
# ls
benchmark  build_debug  cmake           CODE_OF_CONDUCT.md  deps    docs      etc      NOTICE     src   tools
build      build.sh     CMakeLists.txt  CONTRIBUTING.md     docker  Doxyfile  License  README.md  test  unittest
# bash build.sh --make -j4
build.sh --make -j4
create soft link for build_debug, linked by directory named build
```

运行 miniob：

```
# cd build
# ls
benchmark  CMakeCache.txt  cmake_install.cmake    CTestTestfile.cmake  lib       miniob       observer.log.20231110  test   unittest
bin        CMakeFiles      compile_commands.json  deps                 Makefile  miniob.sock  src                    tools
# ./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`;
```

（2）在 miniob 数据库中创建 Scores 表：

创建一张表，包括学号，姓名，成绩

创建并展示：

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

（3）向创建的 miniob 数据库中的 Scores 表中插入数据：

```
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(2231435, '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
2231435 | 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
2231435 | 王亚伟 | 73.2
1950723 | 孙鹏翼 | 89.2
```

可以观察到返回的是一个 SUCCESS，但是并未成功。上网查找资料发现 miniob 并没有实现这个功能。这是一年的 OceanBase 的考题，并未实现相关功能。

（6）删除赵毅斌和孙鹏翼的记录

```
miniob > delete from Scores where id=2210465;
SUCCESS
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
```

（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）

我们在最后将相关报告陈述，请查看下一面。

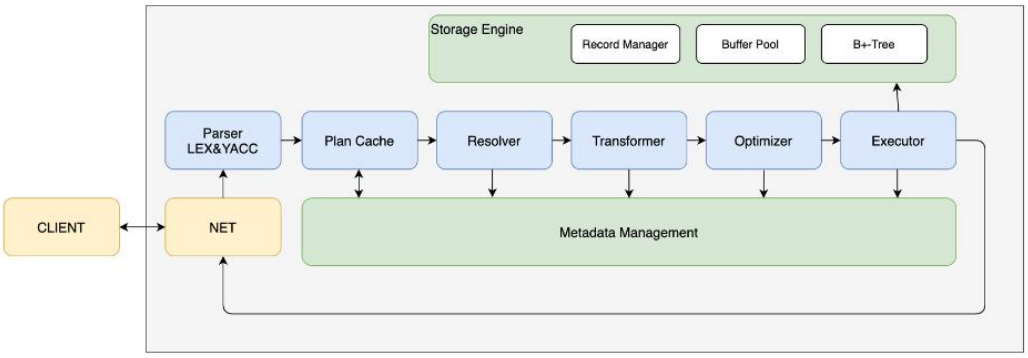
出现的问题：

在整个实验过程中并未出现比较大的问题，主要问题就是 miniob 数据库中的很多语句与原生的 SQL 有一些区别，不能很快的掌握，经过查找一些资料就可以知道。这些在上课并未学到，用起来不是特别的得心应手，需要查找一下资料才能进一步完成。

解决方案：

以上出现的问题都不是比较严重的问题，通过在网上查找资料，比如 CSDN、百度等众多的学习工具，就可以顺利的解决了。

Miniob 源码关于 insert 功能的报告：



在 miniob 的数据库中，对于输入的 SQL 语句是离不开有关编译原理的词法分析和语法分析的，之后需要对语法分析形成的语法树进行优化处理，并最终执行。因此本次报告会着眼于源码中语法树形成后的检验到 insert 语句执行的全部过程。

首先会检验 db（数据库），table_name（表名），inserts.values.empty()（参数是否为空），如果上面任意一个是空或者没有值，就触发 invalid argument 错误。

```
const char *table_name = inserts.relation_name.c_str();
if (nullptr == db || nullptr == table_name || inserts.values.empty()) {
  LOG_WARN("invalid argument. db=%p, table_name=%p, value_num=%d",
      db, table_name, static_cast<int>(inserts.values.size()));
  return RC::INVALID_ARGUMENT;
}
```

之后会检验 table 是否为空，即插入的表是否存在，如果为空即表不存在，将会触发 no such table 错误。

```
Table *table = db->find_table(table_name);
if (nullptr == table) {
  LOG_WARN("no such table. db=%s, table_name=%s", db->name(), table_name);
  return RC::SCHEMA_TABLE_NOT_EXIST;
}
```

之后会检验表 table 中的属性个数与插入的数据个数是否相等，如果不相等则触发 schema mismatch 错误。

```
if (field_num != value_num) {
  LOG_WARN("schema mismatch. value num=%d, field num in schema=%d", value_num, field_num);
  return RC::SCHEMA_FIELD_MISSING;
}
```

在插入的数据个数和属性个数相等后，会依次检验各属性类型和插入的数据类型是否相等，不相等会触发 field type mismatch 错误。

```
const int sys_field_num = table_meta.sys_field_num();
for (int i = 0; i < value_num; i++) {
  const FieldMeta *field_meta = table_meta.field(i + sys_field_num);
  const AttrType field_type = field_meta->type();
  const AttrType value_type = values[i].attr_type();
  if (field_type != value_type) {  // TODO try to convert the value type to field type
    LOG_WARN("field type mismatch. table=%s, field=%s, field type=%d, value_type=%d",
        table_name, field_meta->name(), field_type, value_type);
    return RC::SCHEMA_FIELD_TYPE_MISMATCH;
  }
}
```

在执行上面基本的语法错误和语句错误的检验之后，会用下面代码执行具体的 insert 操作。即根据上面获得的数据创建 InsertStmt 对象 stmt，创建过程会自动执行相关的构造函数，即 SQL 语句 insert 语句的执行过程，执行完成后返回 Success 表示执行成功。

```
// everything alright
stmt = new InsertStmt(table, values, value_num);
return RC::SUCCESS;
```

接下来我们回到 InsertStmt.h 头文件中，找到其具体的构造函数，观察一下其具体怎么执行的。

```
public:
  InsertStmt() = default;
  InsertStmt(Table *table, const Value *values, int value_amount);

  StmtType type() const override
  {
    return StmtType::INSERT;
  }
```

其具体的构造函数如上，而 SQL 的类型会返回 StmtType::INSERT。具体的执行过程我并没有找到，应该在这句 stmt = new InsertStmt(table, values, value_num); 中实现了对数据库的更改，但是我在构造函数中并没有找到具体的内容，可能是我没有看懂，具体的执行过程我并没有找到。

但是根据最后的 return RC::SUCCESS;我们可知上述的 insert 语句已经成功执行了。