《数据库系统原理》实验报告（）					
题目：实验六					
学号	2351753	姓名	黄保翔	日期	5.18
实验环境：docker oceanbase					

一、drop table 实现解析

1.题目信息

删除表。清除表相关的资源。

注意：要删除所有与表关联的数据，不仅仅是在 create table 时创建的资源，还包括索引等数据。

2.测试用例示例（只显示两个）

```
- echo 1. Drop empty table
CREATE TABLE Drop_table_1(id int, t_name char);
DROP TABLE Drop_table_1;

-- echo 2. Drop non-empty table
CREATE TABLE Drop_table_2(id int, t_name char);
INSERT INTO Drop_table_2 VALUES (1,'OB');
DROP TABLE Drop_table_2;
```

3.实现逻辑讲解

DROP TABLE 是 CREATE TABLE 的逆操作，目标是彻底清理与指定表相关的所有资源，包括：

1. 表元数据文件
2. 表数据文件
3. 索引文件
4. 缓冲池缓存

为了实现 DROP TABLE，需要在以下几个模块中添加逻辑：

SQL 处理阶段：解析 DROP TABLE 语句并调用处理接口。

数据库层：查找表并触发删除操作。

表层：具体清理文件和缓冲池中的数据。

SQL 处理阶段（default_storage_stage.cpp）

1.在 SQL 处理函数中，新增对 SCF_DROP_TABLE 语句的处理。

2.从 SQL 语句中提取要删除的表名，调用 handler 的 drop_table 接口。

3.根据操作结果返回 SUCCESS 或 FAILURE。

具体代码示例：

在 sql 文件中添加 drop table 分支，用于处理语法

```
case SCF_DROP_TABLE: {
  const DropTable& drop_table = sql->sstr[sql->q_size-
1].drop_table;
  RC rc = handler_->drop_table(current_db,
drop_table.relation_name);
```

```
  snprintf(response, sizeof(response), "%s\n", rc ==
RC::SUCCESS ? "SUCCESS" : "FAILURE");
  break;
}
```

这里调用 handler 的 drop_table 方法，用于获取表名，下面实现 handler 的具体方法

Handler 实现（default_handler.cpp）：

实现 drop_table 接口，查找数据库并转发请求：

```
RC DefaultHandler::drop_table(const char* dbname, const char*
relation_name) {
  Db* db = find_db(dbname);
  if (!db) return RC::SCHEMA_DB_NOT_OPENED;
  return db->drop_table(relation_name);
}
```

最后实现表资源的清理，在 table 文件中实现 destroy 方法，用于清理文件和缓冲池中的数据，确保缓冲池数据同步，逐一删除元数据、数据、文本和索引文件

RC Table::destroy(const char* dir)

4.执行测试样例示意

本地 docker

```
miniob > CREATE TABLE Drop_table_1(id int, t_name char);
SUCCESS
miniob > DROP TABLE Drop_table_1;
SUCCESS
miniob > CREATE TABLE Drop_table_2(id int, t_name char);
SUCCESS
miniob > INSERT INTO Drop_table_2 VALUES (1,'OB');
SUCCESS
miniob > DROP TABLE Drop_table_2;
SUCCESS
```

网站测试

drop-table	10	10	-

二、select table 实现

1.题目信息

当前系统支持单表查询的功能，需要在此基础上支持多张表的笛卡尔积关联查询。需要实现 select * from t1,t2; select t1.*,t2.* from t1,t2;以及 select t1.id,t2.id from t1,t2;查询可能会带条件。查询结果展示格式参考单表查询。注意查询条件中的“不等”比较，除了"<>"还要考虑"!=" 比较符号。每一列必须带有表信息，

2.测试用例示例

```
-- sort SELECT * FROM
Select_tables_1,Select_tables_2,Select_tables_3;
```

```
-- sort SELECT
Select_tables_1.id,Select_tables_2.u_name,Select_tables_3.res
FROM Select_tables_1,Select_tables_2,Select_tables_3;
Select Select_tables_1.res FROM
Select_tables_1,Select_tables_2,Select_tables_3;
```

3.实现解析

1.解析表列表

目标：获取查询涉及的所有表及其元信息（表结构）。

实现：

- 使用 unordered_map<string, int> 存储表名到序号的映射，用于快速查找表。
- 遍历 selects.relation_num 和 selects.relations（MiniOB 中表示表列表的字段），调用 DefaultHandler::get_default().find_table(table_name) 获取每张表的元信息（如字段定义）。
- 如果表不存在，返回 RC::SCHEMA_TABLE_NOT_EXIST（MiniOB 的错误码）。
- 注意：遍历表时，从 selects.relation_num - 1 到 0 反序遍历，以保持 SQL 语句中表的顺序

代码伪代码：

```
unordered_map<string, int> table_map;
vector<Table*> tables;
for (int i = selects.relation_num - 1; i >= 0; i--) {
  Table* table = DefaultHandler::get_default().find_table(selects.relations[i]);
  if (!table) return RC::SCHEMA_TABLE_NOT_EXIST;
  table_map[selects.relations[i]] = tables.size();
  tables.push_back(table);
}
```

2. 读取表数据

目标：获取每张表的所有记录，准备后续联结。

实现：

- 对每张表调用 MiniOB 的 Table::scan_record 方法，读取全部记录。
- 传入 nullptr 作为过滤器，表示不提前过滤（全表扫描）。
- 将每张表的记录存储在 vector<string> 或 MiniOB 的 Record 结构中，具体取决于 scan_record 的接口。

代码伪代码

```
vector<vector<Record>> table_records(tables.size());
for (size_t i = 0; i < tables.size(); i++) {
  Table* table = tables[i];
  Record record;
  while (table->scan_record(nullptr, &record) == RC::SUCCESS) {
    table_records[i].push_back(record);
  }
}
```

3. 生成笛卡尔积

目标：将多张表的记录组合成联结表，生成所有可能的记录组合。

实现：

- 笛卡尔积是多表记录的交叉组合，直接用嵌套循环实现简单，但在表多或数据量大时会导致内存爆炸。这里我们使用了 dfs 算法进行解决
- 可以在递归过程中直接根据 attr_list 进行投影（只保留需要的列），减少内存占用。
- 联结表记录存储在 vector<string> 或 vector<Record> 中，表示所有可能的组合。

```
vector<vector<string>> join_records; // 存储联结表
vector<string> current_record; // 当前联结记录
void generate_cartesian_product(const vector<vector<Record>>& table_records, size_t table_idx) {
  if (table_idx == table_records.size()) {
    join_records.push_back(current_record); // 保存当前组合
    return;
  }
  for (const auto& record : table_records[table_idx]) {
    current_record.push_back(record.data); // 拼接记录
    generate_cartesian_product(table_records, table_idx + 1);
    current_record.pop_back(); // 回溯
  }
}
// 调用
generate_cartesian_product(table_records, 0);
```

4. 过滤记录

目标：根据 condition_list 过滤联结表，保留满足条件的记录。

实现：

- MiniOB 提供了 CompositeConditionFilter 类，用于处理复杂的 WHERE 条件。
- 关键调整：联结表记录是多表记录的拼接，字段偏移量会变化。需要计算每张表在联结表中的字段偏移量：
  - 假设表 A 有 3 个字段，表 B 有 2 个字段，则表 B 的字段在联结表中的偏移量从第 3 位开始。
  - 修改 CompositeConditionFilter 的逻辑，更新条件中字段的偏移量。
- 对联结表中的每条记录应用过滤器，保留满足条件的记录。

代码伪代码

```
vector<vector<string>> filtered_records;
CompositeConditionFilter filter;
filter.init(tables, selects.conditions, selects.condition_num); // 初始化过滤器

for (const auto& join_record : join_records) {
  if (filter.filter(join_record)) { // 检查记录是否满足条件
    filtered_records.push_back(join_record);
  }
}
```

5. 投影操作并输出

目标：从过滤后的记录中提取 attr_list 指定的列，生成最终结果并输出。

实现：

- 根据 selects.attributes（MiniOB 中表示 attr_list），从每条过滤记录中提取指定列。
- 构造 MiniOB 的 TupleSet 对象，存储投影后的结果。

- 调用 TupleSet::print 输出结果，建议传入表数量作为参数，动态决定是否打印表名（单表可省略表名前缀，多表需显示）。
- 属性类型支持：
  - 单表：支持 *（所有列）、id.*（表的所有列）、id.id（指定列）、id（单个列）。若单表只有 id，可自动加上表名。
  - 多表：支持 *（所有表的列）、id.*（指定表的所有列）、id.id（指定列）。

代码伪代码

```
TupleSet tuple_set;
for (const auto& record : filtered_records) {
  Tuple tuple;
  for (int i = selects.attr_num - 1; i >= 0; i--) { // 反序遍历属性
    const RelAttr& attr = selects.attributes[i];
    // 根据 attr 获取字段值，添加到 tuple
    tuple.add(get_field_value(record, attr, tables));
  }
  tuple_set.add(std::move(tuple));
}
tuple_set.print(std::cout, tables.size()); // 传入表数量
```

4.执行测试样例示意

网站测试提交

```
select-tables                    10          10          -
```

出现的问题：

Docker 本地运行失败

解决方案：

将修改好的代码传到 github 上，

在 docker 中进行 git clone

按照上一次的流程进行修改，最终实现成功

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

Welcome to the OceanBase database implementation course.

Copyright (c) 2021 OceanBase and/or its affiliates.

Learn more about OceanBase at https://github.com/oceanbase/oceanbase
Learn more about MiniOB at https://github.com/oceanbase/miniob
```