

《数据库系统原理》实验报告（6）					
题目：miniOB 进阶实验					
学号	2352018	姓名	刘彦	日期	2025.5.16
实验环境： Docker-desktop 4.41.2 MiniOB-2024-competition(GitHub - oceanbase/miniob at 2024-competition) Visual Studio Code					
实验步骤及结果截图： (1)drop-table 题目描述： 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。 测试用例示例： <div><pre>create table t(id int, age int); create table t(id int, name char); drop table t; create table t(id int, name char);</pre></div> <p>该测试用例是为了验证数据库在执行 drop table 时能否做到表资源的完全释放与重建后的干净状态，也是官方提测通道给出的测试数据。</p> <p>在没有 DROP TABLE 之前，两次 CREATE TABLE t(...)会导致重名冲突，系统应能检测到并报错。可以测试系统是否正确阻止重复建表。DROP TABLE t;应彻底释放与表相关的所有资源，包括：磁盘上的存储文件，内存中的元数据和索引，有助于验证系统是否真正清理干净，为后续重建同名表腾出空间。先建的 t 表字段为(id, age)，删掉后再建为(id, name)。可验证系统是否成功替换了旧表结构，是否保留了旧结构（不该保留）。</p> 实现思路： A.修改 SQL 解析相关文件 ●修改 yacc_sql.y <div><pre>drop_table_stmt: DROP TABLE ID { \$\$ = new ParsedSqlNode(SCF_DROP_TABLE); \$\$->drop_table.relation_name = \$3; free(\$3); };</pre></div> <p>负责 SQL 语法解析，将 DROP TABLE <tablename>语句解析成语法树节点。</p> ●修改 parse_defs.h <div><pre>enum SqlCommandFlag { // ... SCF_DROP_TABLE, // ... };</pre></div> <p>定义了 SQL 命令类型，包括 DROP_TABLE 标识。</p>					

B.修改添加执行器相关文件

●修改 `command_executor.cpp`

```
case StmtType::DROP_TABLE: {
    DropTableExecutor executor;
    return executor.execute(sql_event);
}
```

将解析后的 SQL 语句转交给专门的执行器来处理具体的删除操作。

●添加 `drop_table_executor.h`

```
class DropTableExecutor {
public:
    RC execute(SQLStageEvent *sql_event);
};
```

声明了删除表的执行器类。

●添加 `drop_table_executor.cpp`

```
RC DropTableExecutor::execute(SQLStageEvent * sql_event) {
    // ...
    const char *table_name = drop_table_stmt->table_name().c_str();
    RC rc = session->get_current_db()->drop_table(table_name);
    return rc;
}
```

实现了删除表的具体执行逻辑。

C.修改添加语句处理相关文件

●添加 `drop_table_stmt.h`

```
class DropTableStmt : public Stmt
{
public:
    DropTableStmt(const std::string &table_name)
        : table_name_(table_name)
    {}
    virtual ~DropTableStmt() = default;
    StmtType type() const override { return StmtType::DROP_TABLE; }
    const std::string &table_name() const { return table_name_; }
    static RC create(Db *db, const DropTableSqlNode &drop_table, Stmt *&stmt);
private:
    std::string table_name_;
};
```

定义删除表的语句类，用于存储解析后的表名等信息。

●添加 `drop_table_stmt.cpp`

```
RC DropTableStmt::create(Db *db, const DropTableSqlNode &drop_table, Stmt *&stmt)
{
    stmt = new DropTableStmt(drop_table.relation_name);
    sql_debug("drop table statement: table name %s", drop_table.relation_name.c_str());
    return RC::SUCCESS;
}
```

保存要删除的表名，输出调试信息。

D.修改具体执行相关文件

●修改 `db.cpp`

```
RC Db::drop_table(const char *table_name) {
    RC rc = RC::SUCCESS;
    if (common::is_blank(table_name)) {
        LOG_ERROR("drop table fail: table name is empty!");
        return RC::EMPTY;
    }
    Table *table = find_table(table_name);
    if (table == nullptr) {
        LOG_ERROR("drop table fail: table %s not exists!", table_name);
        return RC::SCHEMA_TABLE_NOT_EXIST;
    }
    std::string table_file_path = table_meta_file(path_c_str(), table_name);
    rc = table->drop(table_file_path.c_str());
    if (rc != RC::SUCCESS) {
        LOG_ERROR("Failed to drop table %s.", table_name);
        return rc;
    }
    // 在打开的表中，把这个表给删掉
    opened_tables_.erase(std::string(table_name));
    return rc;
}
```

实际执行删除表的操作，主要步骤是：

- 参数检查：验证表名不为空
- 表存在性检查：通过 `find_table()` 确认表是否存在
- 删除表文件：删除表的元数据文件和数据文件
- 清理内存：从 `opened_tables_` 映射中移除表对象

drop table 具体实现流程：

当用户输入 `DROP TABLE <tablename>` 语句时，系统会经过以下处理流程：首先由 Yacc 解析器将 SQL 文本解析成 `ParsedSqlNode` 语法树节点，然后在 `Resolve` 阶段将该节点转换为可执行的 `DropTableStmt` 语句对象。接着 `CommandExecutor` 根据语句类型创建专门的 `DropTableExecutor` 执行器，执行器最终调用 `Db::drop_table` 方法来完成实际的删除操作。在 `drop_table` 方法中，首先验证表的存在性，然后删除表的物理文件，最后从内存中的 `opened_tables_` 集合中清除该表的记录，从而完成整个表的删除过程。

执行测试样例示例：

●本地测试

```
miniob > create table t(id int, age int);
SUCCESS
miniob > create table t(id int, name char);
t has been opened before.
FAILURE
miniob > drop table t;
SUCCESS
miniob > create table t(id int, name char);
SUCCESS
```

●平台测试

官方提测通道链接: <https://open.oceanbase.com/train?questionId=600004>

2025-05-16 10:39	https://gitee.com/yanliu05/mini-...	master	-	• 执行成功	70.000	成功 7
drop-table			10	10		

(2)select-tables

题目描述: 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。

测试用例示例:

```
CREATE TABLE t1 (
    id INT,
    age INT
);
INSERT INTO t1 VALUES (1, 12);
INSERT INTO t1 VALUES (2, 8);
CREATE TABLE t2 (
    id INT,
    name CHAR(10)
);
INSERT INTO t2 VALUES (1, 'Tom');
INSERT INTO t2 VALUES (3, 'Jerry');
CREATE TABLE t3 (
    id INT,
    score INT
);
INSERT INTO t3 VALUES (1, 100);
INSERT INTO t3 VALUES (2, 95);
```

```
select * from t1,t2;
select * from t1,t2 where t1.id=t2.id and t1.age > 10;
select * from t1,t2,t3;
```

该测试用例用于验证数据库系统对多表笛卡尔积查询、条件连接查询和三表联合查询的支持情况。通过创建三张结构不同的表并插入少量测试数据，依次执行 `SELECT * FROM t1, t2`、`SELECT * FROM t1, t2 WHERE ...` 和 `SELECT * FROM t1, t2, t3` 等查询，检查系统是否正确执行多表数据的组合、字段对齐、条件过滤及列名带表前缀的输出格式，从而全面测试多表查询功能的语义解析、执行逻辑和结果展示能力。

实现思路：

A.修改数据结构相关文件

●修改 `select_stmt.h`

```
class SelectStmt : public Stmt {
private:
    std::vector<Table *> tables_;           // 存储多表查询涉及的所有表
    std::unordered_map<std::string, Table *> table_map_; // 表名到表对象的映射关系
    bool multi_table_query_ = false;       // 标识是否为多表查询
};
```

支持多表查询的核心数据结构，`SelectStmt` 类的多表查询功能通过 `tables_` 存储所有表的指针，`table_map_` 提供表名到表的快速映射，`multi_table_query_` 标志是否为多表查询。在解析阶段，系统识别 `FROM` 子句中的表，填充 `tables_` 和 `table_map_`，并设置标志；在执行阶段，若 `multi_table_query_` 为 `True`，则触发多表连接操作，高效访问和处理各表数据。

B.添加连接计算相关文件

●添加 `join_logical_operator.h`

```
class JoinLogicalOperator : public LogicalOperator
{
public:
    JoinLogicalOperator() = default;
    virtual ~JoinLogicalOperator() = default;
    LogicalOperatorType type() const override
    {
        return LogicalOperatorType::JOIN;
    }
private:
    std::vector<std::unique_ptr<LogicalOperator>> children_; // 待连接的表操作符
    std::vector<ConditionSqlNode> join_conditions_;         // 连接条件
    JoinType join_type_;
};
```

表连接的逻辑操作符，用于连接两个表，对应的物理算子或者实现。

●添加 `join_physical_operator.h`

```

class NestedLoopJoinPhysicalOperator : public PhysicalOperator
{
public:
    NestedLoopJoinPhysicalOperator();
    virtual ~NestedLoopJoinPhysicalOperator() = default;
    PhysicalOperatorType type() const override
    {
        return PhysicalOperatorType::NESTED_LOOP_JOIN;
    }
    RC open(Trx *trx) override;
    RC next() override;
    RC close() override;
    Tuple *current_tuple() override;
private:
    RC left_next();    // 左表遍历下一条数据
    RC right_next();   // 右表遍历下一条数据
private:
    Trx *trx_ = nullptr;
    // 左表右表的真实对象是在 PhysicalOperator::children_中
    PhysicalOperator *left_ = nullptr;
    PhysicalOperator *right_ = nullptr;
    Tuple *left_tuple_ = nullptr;
    Tuple *right_tuple_ = nullptr;
    JoinedTuple joined_tuple_;
    bool round_done_ = true;
    bool right_closed_ = true;
};
    
```

实现笛卡尔积的物理算子，依次遍历左表的每一行，然后关联右表的每一行。

C.修改执行阶段处理相关文件

●修改 `execute_stage.cpp`

```
RC ExecuteStage::handle_request_with_physical_operator(SQLStageEvent *sql_event)
{
    SelectStmt *select_stmt = static_cast<SelectStmt *>(sql_event->stmt());
    if (select_stmt->is_multi_table()) {
        for (const Field &field : select_stmt->query_fields()) {
            const Table *table = field.table();
            std::string field_name = std::string(table->name()) + "." + field.field_name();
            schema.append_cell(field_name, field.type());
        }
    }
    PhysicalOperator *oper = sql_event->physical_operator();
    return oper->execute();
}
```

这个执行阶段函数通过三个关键机制支持多表查询：

- 表数量检测：判断查询是否涉及多个表
- 字段标识：为多表查询的字段添加表名前缀，形成完整的字段标识
- 结果集构建：根据表名和字段名构建带有表信息的结果集模式(Schema)

这种设计让系统能够正确处理来自不同表的字段，避免同名字段冲突，并在结果展示时清晰地标识每个字段的来源表。整个过程是在查询执行前完成的，为后续的数据处理提供了必要的元数据支持。

select tables 具体实现流程：

SELECT 多表查询的执行流程首先通过 SQL 解析器 (yacc_sql.y)将 SQL 语句解析为语法树，随后在 SelectStmt::create 中 创建 查询 语句 对象，包含表、字段和条件信息；接着在 LogicalPlanGenerator::create_plan 中生成逻辑计划，使用 JoinLogicalOperator 处理多表连接；然后在 PhysicalPlanGenerator::create_plan 中将逻辑计划转换为物理计划，通过 NestedLoopJoinPhysicalOperator 实现表的连接操作；最后在 ExecuteStage::handle_request_with_physical_operator 中执行查询并返回结果，根据 tables().size() > 1 判断是否需要在结果中显示表名，从而完成整个多表查询的处理流程。

执行测试样例示例：

- 本地测试

```

miniob > CREATE TABLE t1 (
    id INT,
    age INT
);
SUCCESS
miniob > INSERT INTO t1 VALUES (1, 12);
SUCCESS
miniob > INSERT INTO t1 VALUES (2, 8);
SUCCESS
miniob > CREATE TABLE t2 (
    id INT,
    name CHAR(10)
);
SUCCESS
miniob > INSERT INTO t2 VALUES (1, 'Tom');
SUCCESS
miniob > INSERT INTO t2 VALUES (3, 'Jerry');
SUCCESS
miniob > CREATE TABLE t3 (
    id INT,
    score INT
);
SUCCESS
miniob > INSERT INTO t3 VALUES (1, 100);
SUCCESS
miniob > INSERT INTO t3 VALUES (2, 95);
SUCCESS
    
```

```

miniob > select * from t1,t2;
t1.id | t1.age | t2.id | t2.name
1 | 12 | 1 | Tom
1 | 12 | 3 | Jerry
2 | 8 | 1 | Tom
2 | 8 | 3 | Jerry
miniob > select * from t1,t2 where t1.id=t2.id and t1.age > 10;
t1.id | t1.age | t2.id | t2.name
1 | 12 | 1 | Tom
miniob > select * from t1,t2,t3;
t1.id | t1.age | t2.id | t2.name | t3.id | t3.score
1 | 12 | 1 | Tom | 1 | 100
1 | 12 | 1 | Tom | 2 | 95
1 | 12 | 3 | Jerry | 1 | 100
1 | 12 | 3 | Jerry | 2 | 95
2 | 8 | 1 | Tom | 1 | 100
2 | 8 | 1 | Tom | 2 | 95
2 | 8 | 3 | Jerry | 1 | 100
2 | 8 | 3 | Jerry | 2 | 95
    
```

●平台测试

官方提测通道链接: <https://open.oceanbase.com/train?questionId=600004>

2025-05-16 10:39

<https://gitee.com/yanliu05/mini-...>

master

-

● 执行成功

70.000

成功 7

select-tables

10

10

出现的问题：

MiniOB 出现编译失败导致测试平台无法通过

在对代码进行修改过后，提交测试平台出现编译错误，第一个 basic 测试点就无法通过。

查看结果

题目标题	分数	所得分	详情
basic	10	0	<div>failed to run cmake cmake failed,CMake Error at CMakeLists.txt:106 (ADD_SUBD</div>

解决方案：

MiniOB 出现编译失败导致测试平台无法通过问题的解决

经检查发现，出现编译错误的原因是在处理 db.cpp 时无法找到 Db 类中 drop_table(const char*)函数的声明，是因为在 db.h 文件中未对新写的函数进行定义，加上定义后就可以成功编译运行。