[redacted]实习实训开发文档

[redacted]

项目标题：　　数据库内核实训

[redacted]

日　　期：　　2024 年 7 月 24 日

目录

1 项目概述.....3
2 项目总体设计.....4
2.1 设计思想.....4
2.2 系统总体框架.....4
2.3 关键技术与算法.....5
2.4 系统文件说明.....6
3 重要的数据结构.....7
4 主要函数处理流程.....7
5 对外提供的接口.....10
6 测试情况说明.....10
6.1 测试用例说明.....10
6.2 测试结果.....14
7 项目总结.....18

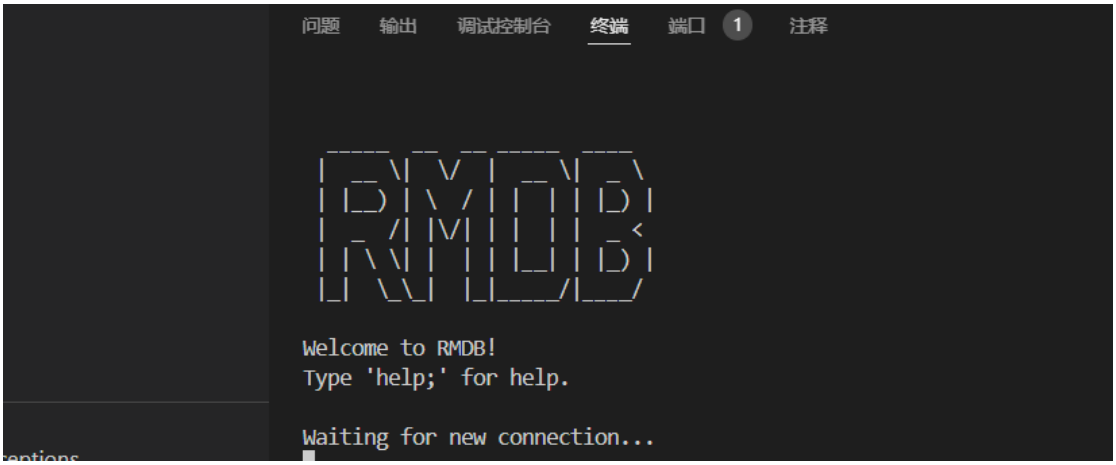
《数据库内核实训》项目实验报告

1 项目概述

数据库管理系统是现代信息系统的核心组成部分，它负责数据的存储、管理和检索。通过本项目，我们希望能够深入理解数据库内核的设计和实现原理，掌握数据库操作的基本方法和技术，并在此基础上实现一个简化的数据库管理系统。

项目基于大赛提供的原型系统框架，开发实现赛题的指定功能，并实现了一定的性能优化，逐步构建一个具备基本数据库操作能力的系统。在老师的指导和成员的努力下，我们小组顺利完成了一、二、四、七、八题，相应完成了 DBMS 系统的存储管理、查询执行、时间类型处理、排序操作、和块嵌套循环连接算法。

详细来说，存储管理方面，实现了文件存储组织、记录存储组织和缓冲区管理，确保数据能够高效且并行地存储和检索。查询执行方面，完成了逻辑查询优化、查询解析和基本算子的实现，包括 DDL 语句 create table 和 drop table，DML 语句 insert、delete、update，DQL 语句 select，使系统能够执行简单高效的查询操作。时间类型方面，添加了 DATETIME 时间类型的存储、验证、查询执行等操作，确保时间数据的正确性和有效性。排序操作方面，实现了 ORDER BY 操作符和 LIMIT 关键字，使系统能够对查询结果进行排序和限制输出结果集大小。连接算法方面，实现了块嵌套循环连接算法（Block Nested-Loop-Join），减少了连接表时的 I/O 次数，优化了连接速度。



图表 1 系统启动画面

2 项目总体设计

2.1 设计思想

项目的整体设计基于 2023 全国大学生计算机系统能力大赛提供的原型系统框架构建。即采用关系数据库模型，通过模块化的设计方法，实现查询解析、查询优化、查询执行、存储等多个功能模块。系统通过 TCP 连接接收客户端发送的 SQL 语句，经过解析、优化和执行等步骤完成数据库操作

2.2 系统总体框架

针对于我们目前完成的系统，包括以下几个模块：

SQL 解析模块：

parser：解析器负责将接收到的 SQL 字符串转换成抽象语法树（AST）。这一过程通过 flex 进行词法分析，将 SQL 字符串分割成词法单元（token），然后通过 bison 进行语法分析，将这些词法单元组合成抽象语法树（AST）。

语义分析模块：

analyze：语义分析模块对解析得到的 AST 进行进一步处理，包括语义检查、数据校验和数据处理，生成 query 树。语义检查确保 SQL 语句中的表、列等对象在数据库中存在且可访问。

查询优化模块：

optimizer：查询优化模块根据 query 树对解析后的查询进行优化，生成查询执行计划（plan）。优化过程包括选择最优的查询执行路径、重新排列查询操作顺序等，提高查询执行效率。

执行计划模块：

portal：portal 模块根据生成的执行计划（plan），创建对应的算子树（portalStmt），算子树中的算子表示具体的数据库操作。

execution：执行模块遍历算子树并执行各个算子，生成执行结果。算子包括了 sort、block_nestedloop_join、delete、insert、nestedloop_join、projection、seq_scan、uptede。

数据存取管理模块：

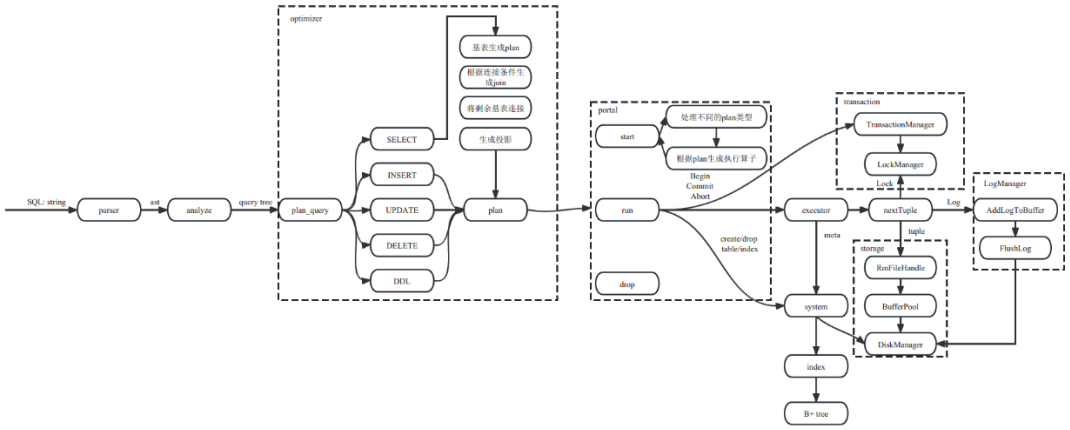
DiskManager：磁盘管理器负责管理磁盘上的数据存取操作，负责将数据页从磁盘读入内存以及将内存中的数据页写回磁盘。

BufferPoolManager：缓冲池管理器负责管理内存中的数据页（缓冲页）。它维护一个缓冲池，用于缓存最近使用的数据页，减少磁盘 I/O 操作。

Replacer：缓冲池替换策略模块负责在缓冲池满时，选择一个数据页进行替换。常见的替换策略包括 LRU（最近最少使用）和 MRU（最近最多使用）。

RmFileHandle：记录管理器负责管理数据库中的记录文件。它提供接口用于读取、写入和修改记录。

RmScan：记录扫描器负责遍历记录文件，执行表扫描操作。



图表 2 系统框架图

从整体的流程来说，SQL 语句首先通过 parser 进行处理，进行词法分析和语法分析，将语句解析成抽象语法树（AST）；analyze 对 AST 进行语义检查、数据校验、数据处理，生成 query 树；然后 optimizer 根据 query 对解析后的查询进行优化，生成查询执行计划 plan。portal 根据 plan 生成对应的算子树 portalStmt，然后遍历算子树并执行 execution 算子生成执行结果。在这个流程下，SQL 语句将会被成功执行，且结果将成功输出。其中，磁盘管理器 DiskManager、缓冲池管理器 BufferPoolManager 和缓冲池替换策略 Replacer、记录管理器 RmFileHandle 和 RmScan 负责数据存取管理。

2.3 关键技术与算法

在团队对框架功能的完善中，使用了如下几个关键技术和算法。

缓冲池替换策略（Replacer）：在缓冲池满时选择数据页进行替换，本系统采用最近最少使用（LRU）策略。

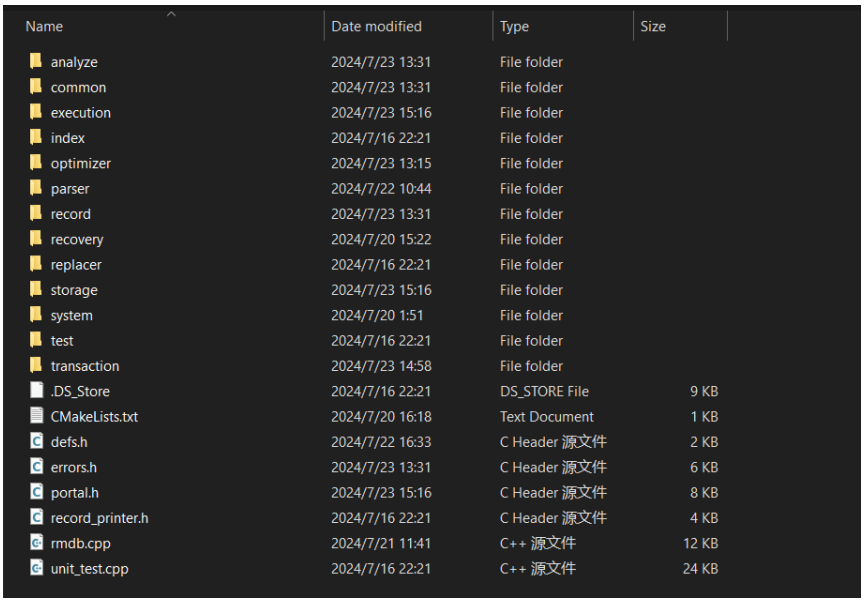
词法分析和语法分析（parser）：词法分析和语法分析是 SQL 解析模块的核心技术之一。词法分析通过 **flex** 工具将输入的 SQL 字符串分割成词法单元（token），而语法分析通过 **bison** 工具将这些词法单元按照 SQL 语法规则组合成抽象语法树（AST）。确保 SQL 语句的正确性和规范性，为后续的查询优化和执行打下基础。

排序算法（execution_sort）：处理包含 order by 的语句时，在 execution_sort 算子中使用了**双轴快速排序**（Dual-Pivot Quicksort）算法，调用 C++ 标准库中的 **std::sort** 函数，是基于快速排序（Quicksort）、堆排序（Heapsort）和插入排序（Insertion sort）的混合算法。

连接算法（executor_nestedloop_join）：在题目二中，系统使用了**循环嵌套连接**算法作为连接算子的连接算法。在题目八中，按照题目要求，系统使用了**块循环嵌套连接**（Block Nested Loop Join）算法，对左表进行分块处理，通过利用块缓冲来减少磁盘 I/O 操作，从而提高连接的效率。

2.4 系统文件说明

项目相比于原框架，改动都在 src 部分，故主要对 src 文件夹下的系统文件进行说明。



Name	Date modified	Type	Size
analyze	2024/7/23 13:31	File folder	
common	2024/7/23 13:31	File folder	
execution	2024/7/23 15:16	File folder	
index	2024/7/16 22:21	File folder	
optimizer	2024/7/23 13:15	File folder	
parser	2024/7/22 10:44	File folder	
record	2024/7/23 13:31	File folder	
recovery	2024/7/20 15:22	File folder	
replacer	2024/7/16 22:21	File folder	
storage	2024/7/23 15:16	File folder	
system	2024/7/20 1:51	File folder	
test	2024/7/16 22:21	File folder	
transaction	2024/7/23 14:58	File folder	
.DS_Store	2024/7/16 22:21	DS_STORE File	9 KB
CMakeLists.txt	2024/7/20 16:18	Text Document	1 KB
defs.h	2024/7/22 16:33	C Header 源文件	2 KB
errors.h	2024/7/23 13:31	C Header 源文件	6 KB
portal.h	2024/7/23 15:16	C Header 源文件	8 KB
record_printer.h	2024/7/16 22:21	C Header 源文件	4 KB
rmdb.cpp	2024/7/21 11:41	C++ 源文件	12 KB
unit_test.cpp	2024/7/16 22:21	C++ 源文件	24 KB

图表 3 src 文件夹目录

- **analyze：**包含语义分析相关的代码和模块，负责对解析得到的抽象语法树（AST）进行语义检查、数据校验和处理。
- **common：**包含数据库系统中常用的公共代码和实用工具，如一些通用的宏定义、辅助函数等。
- **execution：**包含查询执行相关的代码和模块，负责将查询执行计划转换为实际的数据库操作，并生成查询结果。
- **optimizer：**包含查询优化器的代码和模块，负责对查询执行计划进行优化，以提高查询执行效率。
- **parser：**包含 SQL 解析相关的代码和模块，负责将接收到的 SQL 字符串转换成抽象语法树（AST），这一过程包括词法分析和语法分析。
- **record：**包含记录管理相关的代码和模块，负责对数据库记录的读取、写入和修改操作。
- **replacer：**包含缓冲池替换策略相关的代码和模块，负责在缓冲池满时选择一个数据页进行替换，常见的策略包括 LRU（最近最少使用）和 MRU（最近最多使用）。
- **storage：**包含存储管理相关的代码和模块，负责管理数据库的存储结构和数据存取操作。
- **system：**包含系统级别的代码和模块，负责数据库系统的初始化、配置和管理等。
- **defs.h：**包含项目中的一些通用宏定义和类型定义。
- **errors.h：**定义了项目中可能出现的错误类型和错误处理相关的代码。
- **portal.h：**定义了执行计划模块的接口和实现，包括算子树的生成和执行等。
- **record_printer.h：**包含记录打印相关的代码和模块，负责将数据库记录以特定格式输出，便于调试和测试。
- **rmdb.cpp：**数据库系统的主文件，包含系统初始化、配置和运行的主要逻辑。
- **unit_test.cpp：**包含单元测试代码，用于验证各个模块的功能和性能是否符合预期。

3 重要的数据结构

抽象语法树（AST）：
SQL 解析模块生成的中间表示，用于表示 SQL 语句的结构和内容。AST 的节点包括各种 SQL 语句的组成部分，如 SELECT、FROM、WHERE 等子句。每个节点包含子节点，表示该语句的具体内容。

查询树（Query）：
查询树是语义分析模块生成的结构，表示经过语义检查和数据校验的 SQL 查询。查询树的节点包括查询对象和操作条件，表示 SQL 查询的逻辑操作顺序。

查询执行计划（Plan）：
查询执行计划由优化器生成，表示具体的查询执行步骤。计划包含算子树，执行步骤和所需的资源信息。

算子树（PortalStmt）：
Plan 中生成的算子树，用于表示具体的数据库操作，其中包含各种算子。

缓冲池管理器（BufferPoolManager）：
缓冲池管理器用于管理内存中的数据页，存有并维护一个缓冲池，用于缓存最近使用的数据页，减少磁盘 I/O 操作。

磁盘管理器（DiskManager）：
存有文件打开列表，负责管理磁盘上的数据存取操作，将数据页从磁盘读入内存以及将内存中的数据页写回磁盘。

记录管理器（RmFileHandle）：
记录管理器负责管理数据库中的记录文件，提供接口用于读取、写入和修改记录。存有记录指针和缓存，用于管理记录文件的指针和缓存读取的记录数据。

4 主要函数处理流程

本章节将以 rmdb.cpp 文件运行流程中的函数切入，讲解该系统中主要函数的处理流程。yy_scan_string()：该函数是词法分析和语法分析相关函数的入口，将输入的 string 形式的 SQL 语句转换构建为语法树。由于使用了 flex 和 bison 工具通过规则生成了大部分的函数，此处不过多赘述。

do_analyze()函数的处理流程主要包括以下几个步骤：解析输入的抽象语法树（AST），进行语义分析，检查表和列是否存在，以及处理各类 SQL 语句。针对不同的 SQL 语句类型，进行相应的表名处理、列名处理、WHERE 条件处理、值处理等解析和分析操作，返回 Query 对象。

portal->start()方法负责将查询执行计划（Plan）转换成对应的算子树，并返回一个 PortalStmt 对象，其中包含了执行计划的详细信息。该方法根据 Plan 对象的实际类型，分别处理不同的计划类型。若为 DML 操作，将调用 convert_plan_executor()函数将查询执行计

划（Plan）转换成对应的执行器（Executor），形成算子树；若为其他类型则直接生成算子树。

portal->run()方法将遍历算子树并执行算子生成执行结果，不同的算子函数的处理过程将在以下讲述，本章将挑选几个重要的算子进行讲述。

Join 算子实现：

在执行计划模块中，首先先在 rmdb.cpp 文件调用 run 函数，执行对应的语句进行对应的计划操作，即执行代码：

ql->select_from(std::move(portal->root),std::move(portal->sel_cols), context);

然后在 select_form 函数中执行代码块：

```
// 执行query_plan
for (executorTreeRoot->beginTuple(); !executorTreeRoot->is_end(); executorTreeRoot->nextTuple()) {
    auto Tuple = executorTreeRoot->Next();
    std::vector<std::string> columns;
    for (auto &col : executorTreeRoot->cols()) {
        std::string col_str;
        char *rec_buf = Tuple->data + col.offset;
        if (col.type == TYPE_INT) {
            col_str = std::to_string(*(int *)rec_buf);
        } else if (col.type == TYPE_FLOAT) {
            col_str = std::to_string(*(float *)rec_buf);
        } else if (col.type == TYPE_STRING) {
            col_str = std::string((char *)rec_buf, col.len);
            col_str.resize(strlen(col_str.c_str()));
        } else if(col.type == TYPE_DATETIME){
            std::string str = std::to_string(*(int64_t *)rec_buf);
            col_str.reserve(16);  // 预分配内存
```

这里调用对应执行器的内部函数，即 nextTuple()函数，逐步遍历查找对应的符合条件的记录。

在 executor_nestedloop_join.h 文件中，定义函数：

```
75      void nextTuple() override {
76          right_->nextTuple();
77          if (right_->is_end()) {
78              left_->nextTuple();
79              if (left_->is_end()) {
80                  isend = true;
81                  return;
82              }
83              cur_left_record_ = left_->Next();
84              right_->beginTuple();
85          }
86          cur_right_record_ = right_->Next();
87
88          auto rec = std::make_unique<RmRecord>(len_);
89          memcpy(rec->data, cur_left_record_->data, cur_left_record_->size);
90          memcpy(rec->data + cur_left_record_->size, cur_right_record_->data, cur_right_record_->size);
91          //current_record_ = std::move(rec);
92          // 评估连接后的记录是否满足条件
93          if (eval_conds(cols_, fed_conds_, rec.get())) {
94              current_record_ = std::move(rec);
95          }else {
96              nextTuple();
97          }
98      }
```

将左表和右表的记录连接后进行判断是否符合条件，成功之后返回该记录，否则继续递归调用该函数直到找到对应的符合条件的组合记录。

Block Nested-Loop-Join 算子实现：

该算子的执行流程与 Join 的执行流程一致，Join 操作的时候该调用执行器 NestedLoopJoinExecutor 为执行器 BlockNestedLoopJoinExecutor 。而在 BlockNestedLoopJoinExecutor 中，查找记录的时候设置了左右两个缓冲池：

```
std::vector<Page*> right_buffer_pages_; // 右表缓冲页面
int right_buffer_page_cnt_{0}; // 右表缓冲页面数量
std::vector<Page*> left_buffer_pages_; // 左表缓冲页面
int left_buffer_page_cnt_{0}; // 左表缓冲页面数量
```

在遍历查找的过程中需要将对应的左表记录和右表记录都存储在对应的缓冲池中，在之后的查找过程中，直接从对应的缓冲池中获取左表或者右表的记录进行连接判断，如果记录符合条件，返回，不符合条件继续执行。该算法主要是减少数据库读写磁盘的 IO 次数，大大降低了数据库执行 Join 的操作时的时间。

```
void nextTuple() override {
    while (!left_over) {
        while (!right_over) {
            if (process_pages(left_buffer_pages_, left_buffer_page_cnt_, right_buffer_pages_, right_buffer_page_cnt_)) {
                return; // 如果找到了符合条件的记录，则返回，直接退出到调用该函数的地方，即外面的for循环
            }
            if (right_->is_end()) {
                right_over = true;
                left_buffer_page_iter_ = 0;
                continue; // 右表已经处理完，继续处理左表
            }
            refill_pages(right_buffer_pages_, right_buffer_page_cnt_, right_, right_len_, right_num_now_);
            left_buffer_page_iter_ = 0;
            left_buffer_page_inner_iter_ = 0;
        }
        if (left_->is_end()) {
            left_over = true;
            break; // 左表已经处理完，退出
        }
        refill_pages(left_buffer_pages_, left_buffer_page_cnt_, left_, left_len_, left_num_now_);
        right_->beginTuple(); // 重新初始化右表
        refill_pages(right_buffer_pages_, right_buffer_page_cnt_, right_, right_len_, right_num_now_);
        right_over = false;
    }
    is_end_ = true; // 标记结束
    for (auto left_page : left_buffer_pages_) {
        bpm_->unpin_tmp_page(left_page->get_page_id()); // 释放左表页面
    }
    for (auto right_page : right_buffer_pages_) {
        bpm_->unpin_tmp_page(right_page->get_page_id()); // 释放右表页面
    }
}
```

```
// 处理页面的辅助函数，用于查找左表和右表符合条件记录的连接结果
bool process_pages(std::vector<Page*>& left_pages, int left_page_cnt, std::vector<Page*>& right_pages, int right_page_cnt) {
    while (left_buffer_page_iter_ < left_page_cnt) {
        auto left_page = left_pages[left_buffer_page_iter_];
        int left_num_now_inner_ = left_num_now_.find(left_page->get_page_id())->second;
        while (left_buffer_page_inner_iter_ < left_num_now_inner_) {
            memcpy(join_record.data, left_page->get_data() + left_buffer_page_inner_iter_ * left_len_, left_len_);
            while (right_buffer_page_iter_ < right_page_cnt) {
                auto right_page = right_pages[right_buffer_page_iter_];
                int right_num_now_inner_ = right_num_now_.find(right_page->get_page_id())->second;
                while (right_buffer_page_inner_iter_ < right_num_now_inner_) {
                    memcpy(join_record.data + left_len_, right_page->get_data() + right_buffer_page_inner_iter_ * right_len_, right_len_);
                    right_buffer_page_inner_iter_++;
                    if (CheckConditions(join_record.data)) {
                        return true; // 如果条件满足，则返回true
                    }
                }
                right_buffer_page_iter_++;
                right_buffer_page_inner_iter_ = 0;
            }
            right_buffer_page_iter_ = 0;
            left_buffer_page_inner_iter_++;
        }
        left_buffer_page_iter_++;
        left_buffer_page_inner_iter_ = 0;
    }
    return false;
}
```

在执行 nextTuple 函数的时候会调用 process_pages 来查找缓冲池中对应的记录，但是其中的 nextTuple 函数调用的逻辑和不使用缓冲池的 Join 逻辑一致。只是在查找缓冲池的过程中将缓冲池分为不同页，每个也有对应的记录数据。在查找过程中不仅需要在缓冲池里面查找数据，还需要根据不同情况随时更新缓冲池的数据

这是整个过程的大概循环流程：

for(left buffer)

for(right buffer in whole right)

```
for(page in left buffer)
    for(tuple in left buffer)
        for(page in right buffer)
            for(tuple in right buffer page)
```

5 对外提供的接口

在团队对框架的增改过程中，没有新增对外提供的接口，系统中使用类和函数调用来实现各模块的功能，即只有内部接口的实现，并不直接向外部提供接口。只有在系统与客户端交互时使用了外部接口，以接收客户端请求和给客户端返回结果。

6 测试情况说明

6.1 测试用例说明

存储管理测试用例

- **测试目标**：验证文件存储组织和记录存储组织的正确性，以及缓冲区管理的有效性。
- **输入数据**：unit_test.cpp 中提供了参考测试示例，最终测试包括但不限于 unit_test.cpp 中提供的测试。
- **预期结果**：文件和记录应按照预期组织和存储，缓冲区管理应正常工作。

查询执行测试用例

- **测试目标**：验证查询解析、逻辑查询优化和查询执行框架的功能。
- **输入数据**：不同的 DDL、DQL 和 DML 语句。通过 sql 进行测试，分为五个测试点，它们依次是“尝试建表”、“单表插入与条件查询”、“单表更新与条件查询”、“单表删除与条件查询”、“连接查询”。详细数据如下图：

测试点1: 尝试建表（2分）

测试示例：

```
create table t1(id int,name char(4));

show tables;

create table t2(id int);

show tables;

drop table t1;

show tables;

drop table t2;

show tables;
```

测试点2: 单表插入与条件查询（2分）

测试示例：

```
create table grade (name char(4),id int,score float);

insert into grade values ('Data', 1, 90.5);

insert into grade values ('Data', 2, 95.0);

insert into grade values ('Calc', 2, 92.0);

insert into grade values ('Calc', 1, 88.5);

select * from grade;

select score,name,id from grade where score > 90;

select id from grade where name = 'Data';

select name from grade where id = 2 and score > 90;
```

测试点3：单表更新与条件查询（2分）

测试示例：

```
create table grade (name char(4),id int,score float);

insert into grade values ('Data', 1, 90.5);

insert into grade values ('Data', 2, 95.0);

insert into grade values ('Calc', 2, 92.0);

insert into grade values ('Calc', 1, 88.5);

select * from grade;

update grade set score = 99.0 where name = 'Calc' ;

select * from grade;

update grade set name = 'test' where name > 'A';

select * from grade;

update grade set name = 'test' ,id = -1,score = 0 where name = 'test' and score > 90;

select * from grade;
```

测试点4：单表删除与条件查询（2分）

测试示例：

```
create table grade (name char(4),id int,score float);

insert into grade values ('Data', 1, 90.5);

select * from grade;

delete from grade where score > 90;

select * from grade;
```

期待输出：

```
| name | id | score|

| Data | 1 | 90.500000 |

| name | id | score|
```

```
测试点5：连接查询（4分）
create table t ( id int , t_name char (3));
create table d (d_name char(5),id int);
insert into t values (1,'aaa');
insert into t values (2,'baa');
insert into t values (3,'bba');
insert into d values ('12345',1);
insert into d values ('23456',2);
select * from t, d;
select t.id,t_name,d_name from t,d where t.id = d.id;
```

图表 4 查询执行测试用例

- **预期结果：**查询应正确解析、优化并执行。对于所有不合法的 sql 语句，都需要输出 failure。（包括语法错误、删除不存在的表、创建已经存在的表、where 条件中出现不存在的字段等）

DATETIME 类型测试用例

- **测试目标：**验证 DATETIME 类型的输入合法性和存储正确性。
- **输入数据：**题目中给出的测试用例，见下图：

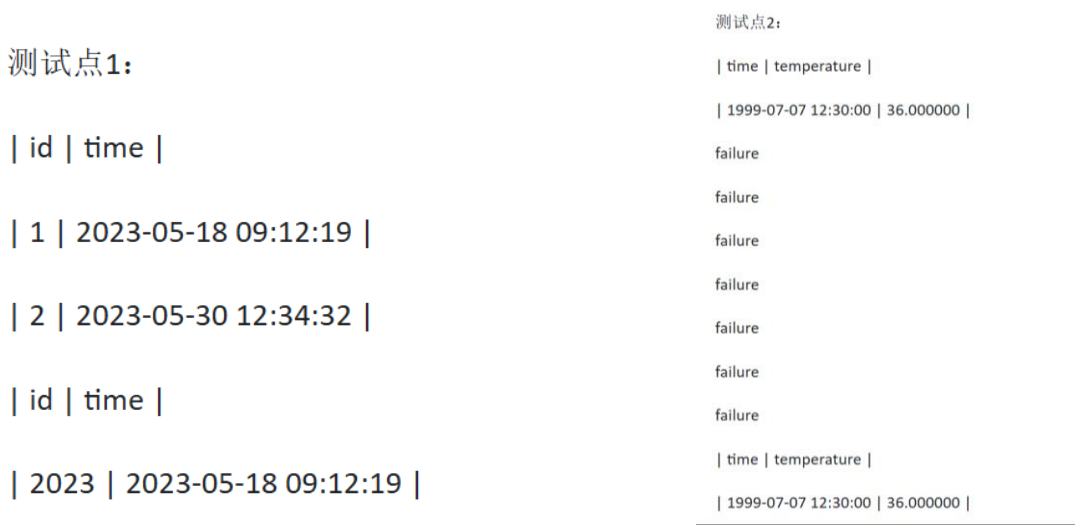
```
测试点1：建表时创建时间类型的属性，并在该字段上进
create table t(id int , time datetime);
insert into t values(1, '2023-05-18 09:12:19');
insert into t values(2, '2023-05-30 12:34:32');
select * from t;
delete from t where time = '2023-05-30 12:34:32';
update t set id = 2023 where time = '2023-05-18 09:12:19';
select * from t;
```

```
测试点2：对输入的合法性进行判断
create table t(time datetime, temperature float)
insert into t values('1999-07-07 12:30:00' , 36.0);
select * from t;
insert into t values('1999-13-07 12:30:00' , 36.0);
insert into t values('1999-1-07 12:30:00' , 36.0);
insert into t values('1999-00-07 12:30:00' , 36.0);
insert into t values('1999-07-00 12:30:00' , 36.0);
insert into t values('0001-07-10 12:30:00' , 36.0);
insert into t values('1999-02-30 12:30:00' , 36.0);
insert into t values('1999-02-28 12:30:61' , 36.0);
select * from t;
```

图表 5 DATETIME 类型测试用例

- **预期结果：**合法输入应正确存储，非法输入应被拒绝。要求实现 DATETIME 时间类型，DATETIME 类型大小为 8 字节，格式为'YYYY-MM-DD HH:MM:SS'，最小值为'1000-01-01 00:00:00'，最大值为'9999-12-31 23:59:59'，参赛队伍需要判断输入值的合法性（非法输

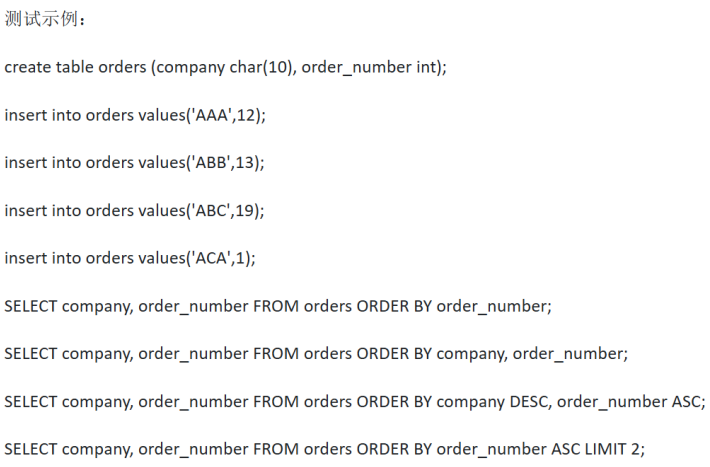
入包括但不限于出现负值、月的值小于 1 或大于 12，日的值小于 1 或大于 31、2 月的天数等于 30、时针数大于 23、分针数大于 59、秒针数大于 59、字段长度不匹配），输出文档如下图：



图表 6 DATETIME 类型预期结果

ORDER BY 和 LIMIT 关键字测试用例

- **测试目标：**验证 ORDER BY 和 LIMIT 关键字的功能。
- **输入数据：**带有 ORDER BY 和 LIMIT 关键字的查询语句，见下图



图表 7 ORDER BY 和 LIMIT 关键字测试用例

- **预期结果：**查询结果应按预期排序和限制。

期待输出：

| company | order_number |

| ACA | 1 |

| AAA | 12 |

| ABB | 13 |

| ABC | 19 |

| company | order_number |

| AAA | 12 |

| ABB | 13 |

| ABC | 19 |

| ACA | 1 |

| company | order_number |

| ACA | 1 |

| ABC | 19 |

| ABB | 13 |

| AAA | 12 |

| company | order_number |

| ACA | 1 |

| AAA | 12 |

图表 8 ORDER BY 和 LIMIT 关键字预期结果

块嵌套循环连接测试用例

- **测试目标：**验证块嵌套循环连接算法的正确性和性能。
- **输入数据：**题目自带的数据库数据。并使用测试用例实现。

测试示例：

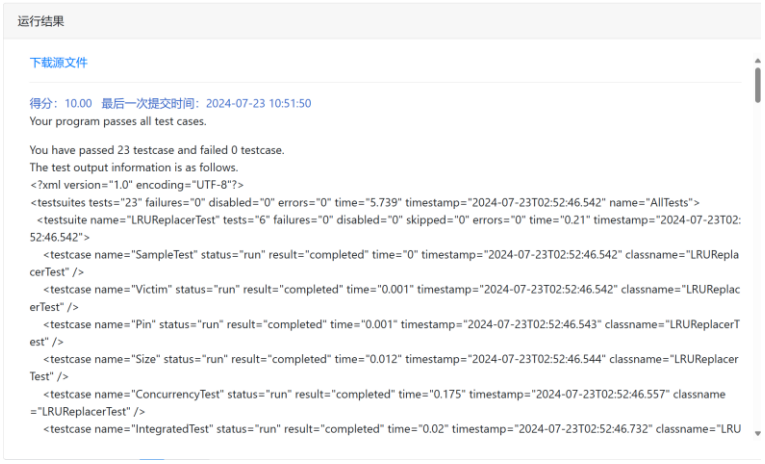
```
select * from t1, t2 where t1.id = t2.t_id order by t1.id;

select * from t1, t2 where t1.id < t2.t_id and t2.t_id < 1000;
```

图表 9 块嵌套循环连接测试用例

- **预期结果：**连接结果应正确，性能应符合预期。

6.2 测试结果

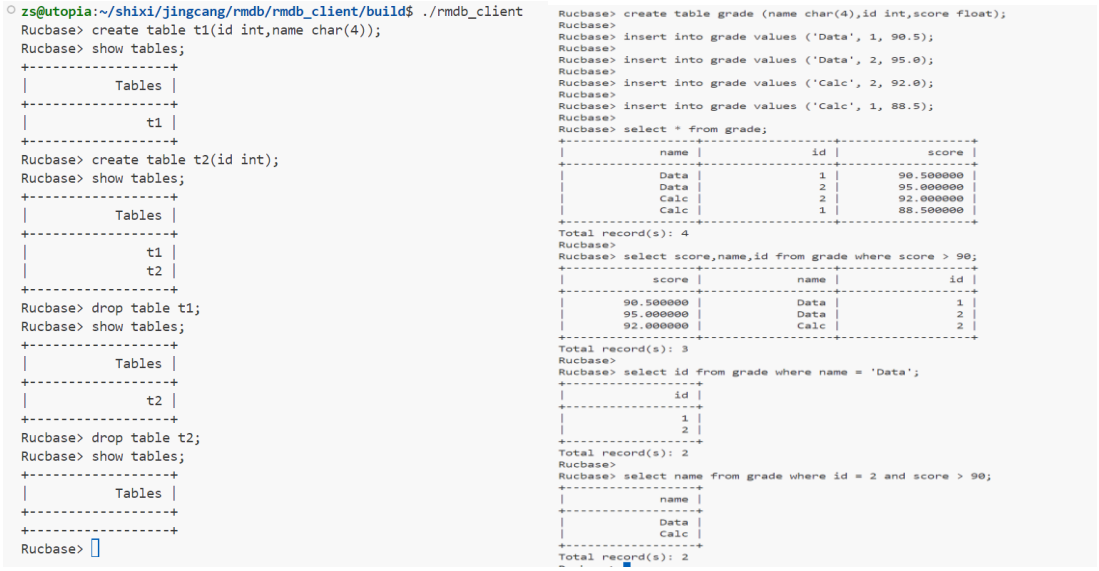
1. **存储管理测试结果**
   - **结果描述：**测试用例通过，文件和记录组织正确，缓冲区管理正常。
   - **测试截图：**

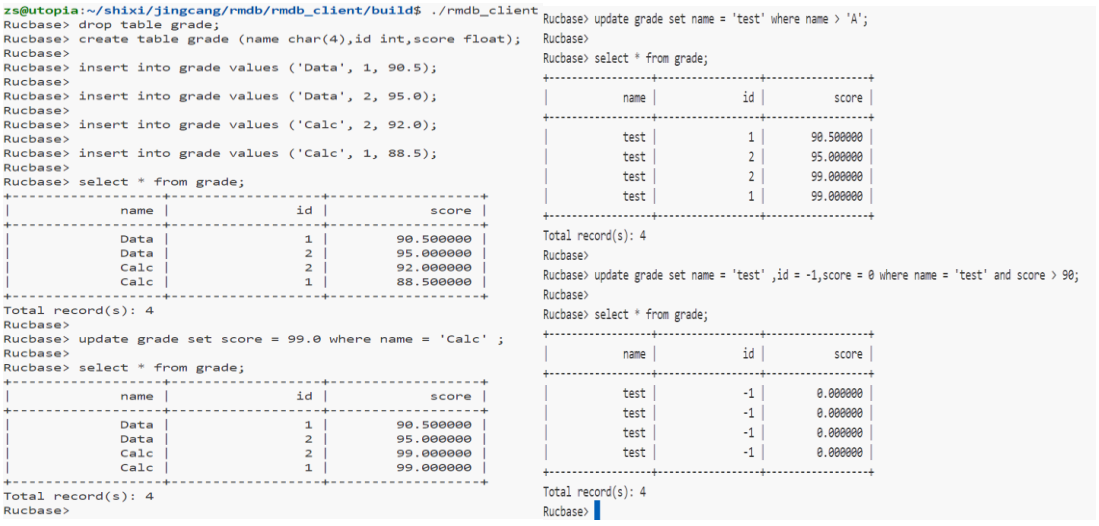


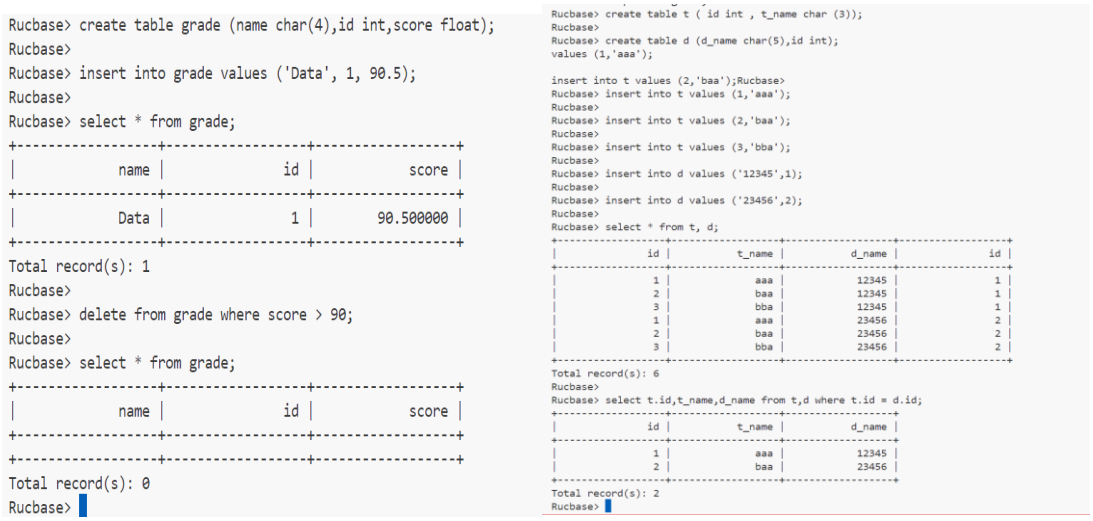
图表 10 存储管理测试结果

2. 查询执行测试结果

- **结果描述：**测试用例通过，查询解析、优化和执行正常。
- **测试截图：**



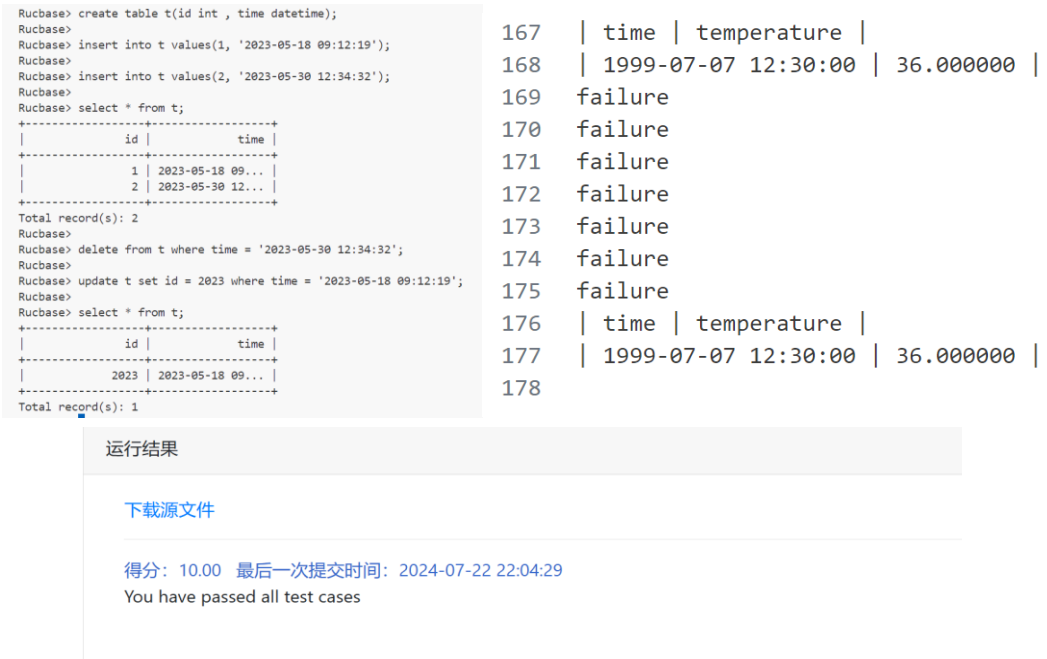




图表 11 查询执行测试结果

3. DATETIME 类型测试结果

- **结果描述：**测试用例通过，合法 DATETIME 值正确存储，非法值被拒绝。
- **测试截图：**



图表 12 DATETIME 类型测试结果

4. ORDER BY 和 LIMIT 关键字测试结果
   - **结果描述：**测试用例通过，ORDER BY 和 LIMIT 功能正常。
   - **测试截图：**

```
Rucbase> SELECT company, order_number FROM orders ORDER BY order_number;
+-----------------+-----------------+
|         company |    order_number |
+-----------------+-----------------+
|             ACA |               1 |
|             AAA |              12 |
|             ABB |              13 |
|             ABC |              19 |
+-----------------+-----------------+
Total record(s): 4
Rucbase>
Rucbase> SELECT company, order_number FROM orders ORDER BY company, order_number;
+-----------------+-----------------+
|         company |    order_number |
+-----------------+-----------------+
|             AAA |              12 |
|             ABB |              13 |
|             ABC |              19 |
|             ACA |               1 |
+-----------------+-----------------+
Total record(s): 4
Rucbase>
Rucbase> SELECT company, order_number FROM orders ORDER BY company DESC, order_number ASC;
+-----------------+-----------------+
|         company |    order_number |
+-----------------+-----------------+
|             ACA |               1 |
|             ABC |              19 |
|             ABB |              13 |
|             AAA |              12 |
+-----------------+-----------------+
Total record(s): 4
Rucbase>
Rucbase> SELECT company, order_number FROM orders ORDER BY order_number ASC LIMIT 2;
+-----------------+-----------------+
|         company |    order_number |
+-----------------+-----------------+
|             ACA |               1 |
|             AAA |              12 |
+-----------------+-----------------+
Total record(s): 2
```

运行结果

下载源文件

得分：10.00 最后一次提交时间：2024-07-22 19:08:06

You haved passed all test cases about orderby test

图表 13 ORDER BY 和 LIMIT 关键字测试测试结果

5. 块嵌套循环连接测试结果
   - **结果描述：**测试用例通过，连接结果正确，性能符合预期。
   - **测试截图：**

运行结果

下载源文件

得分：10.00 最后一次提交时间：2024-07-23 16:42:32

You have passed all test cases!

In join_test_1, you have used 7.609628677368164 seconds

In join_test_2, you have used 27.056267499923706 seconds

图表 14 块嵌套循环连接测试测试结果

7 项目总结

本次项目下来，团队成员都受益匪浅。在此之前，对于相关的知识，我们只学习过数据库课程，了解了数据库的一些基本原理，但是对于系统级的编程，我们并没有开发经验。而通过此次数据库内核实训项目，我们虽然花费了很多时间进行学习和 debug，但是深入理解并实现了许多数据库管理系统的核心原理和技术。

在项目的实施过程中，我们按照基本框架，遵循模块化设计方法，构建了多个功能模块，终于按时完成了 DBMS 系统的存储管理、查询执行、时间类型处理、排序操作和块嵌套循环连接算法的实现（对应题目一、题目二、题目四、题目七、题目八）。

虽然整个过程看似顺利，但实际遇到了很多困难。一开始，我们顺利完成了题目一，原以为题目二也将和题目一一样顺利，就当我们一一完成了题目二要求的算子后，却遇到了数不胜数的 bug。例如 update 语句无法正确处理诸如 set a = a + 1 的语句、子类方法无法正确调用、join 连接算法无法正确完成输出等等等问题，虽然现在看来很多 bug 都是由于一些很简单的 bug 造成的，但是当时的我们足足用了四天的时间才找到相应的问题点并去解决。虽然题目二的完成很耗时间，但是我们也借此熟悉了整个系统的结构和思路，还了解了 bison 和 flex 的使用方法。在完成题目二之后，我们一天就完成了题目四和题目七，又用了一天的时间完成了题目八。

此次实训让我们对数据库系统的开发打下了坚实的基础，我们不仅掌握了数据库内核的设计和实现原理，还积累了丰富的实践经验。我们今后还将继续学习，争取在下届的数据库系统赛取得好成绩。并希望在未来的开发中，我们能够将此次实训所学习的知识和获得的经验融会贯通，为行业发展贡献力量。

本次实训我们全程使用了 gitlab 进行系统的合作开发，完整的源代码和开发 history 可见 gitlab 链接：https://gitlab.eduxiji.net/2022141461108/jingcang.git