实验 4：数据操纵的实现

一、实验目的

1. 掌握 Rucbase 中火山模型的实现方法。
2. 掌握 Rucbase 数据查询算子的实现方法。
3. 掌握 Rucbase 数据更新算子的实现方法。

二、相关知识

1. 火山模型
2. 数据查询算子执行算法
3. 数据更新算子执行算法

三、实验内容

本实验包括 5 项任务。

任务 1：顺序扫描算子的实现

补全 SeqScanExecutor 类，实现顺序扫描算子，具体完成下列任务。

（1）阅读代码

了解 QlManager 类的设计。

理解 SeqScanExecutor 类的设计，并回答下列问题：

1. rid_的作用是什么？

答：rid_ 用于记录当前满足选择条件的元组的位置（Record ID），包含页号和槽号。

（2）实现 SeqScanExecutor::beginTuple 函数

```
void SeqScanExecutor::beginTuple();
```

1.方法声明：

方法名：beginTuple

返回类型：void

功能：构建表迭代器 scan，并开始迭代扫描，直到扫描到第一个满足谓词条件的元组停止，并赋值给 rid

参数列表：无参数

2.方法实现思路：

1. 创建表的记录扫描迭代器 `scan_`（使用 `std::make_unique<RmScan>(fh_)`）

2. 使用 `scan_` 迭代扫描表中的元组
3. 对每条元组调用 `eval_conds()` 检查是否满足所有选择条件
4. 找到第一条满足条件的元组后，将其 Rid 记录在 `rid_` 中并停止扫描（通过 `break` 跳出循环）

3. 方法实现难点：

需要正确初始化记录迭代器，并确保循环在找到第一个匹配项后能正确终止。关键是理解扫描迭代器的使用方式，在每次循环中先获取当前记录的 Rid，然后获取完整记录进行条件判断。

（3）实现 `SeqScanExecutor::nextTuple` 函数

```
void SeqScanExecutor::nextTuple();
```

1.方法声明：

方法名：nextTuple

返回类型：void

功能：从当前 scan 指向的记录开始迭代扫描，直到扫描到第一个满足谓词条件的元组停止，并赋值给 rid

参数列表：无参数

2.方法实现思路：

1. 首先确保 `scan_` 已经初始化（使用 `assert(scan_ != nullptr)`）
2. 调用 `scan_->next()` 移动到下一条记录
3. 继续迭代扫描表中的元组，对每条元组调用 `eval_conds()` 检查是否满足所有选择条件
4. 找到下一条满足条件的元组后，更新 `rid_` 并停止扫描

3. 方法实现难点：

关键在于要从上一条找到的记录之后开始扫描，而不是从头开始，这通过先调用 scan_->next() 来实现。需要注意与 beginTuple() 的区别在于在循环开始前就先移动了迭代器。

（4）实现 `SeqScanExecutor::is_end` 函数

```
void SeqScanExecutor::is_end();
```

1.方法声明：

方法名：is_end

返回类型：bool

功能：判断 scan_是否已经到达末尾

参数列表：无参数

2.方法实现思路：

直接调用记录扫描迭代器 scan_ 的 is_end() 方法，判断是否已经扫描到表末尾。

3. 方法实现难点：

该方法实现较为简单，无明显难点。只需要简单地返回 scan_->is_end() 的结果即可。

（5）实现 `SeqScanExecutor::Next` 函数

```
std::unique_ptr<RmRecord> SeqScanExecutor::Next();
```

1.方法声明：

方法名：Next

返回类型：std::unique_ptr

功能：返回下一个满足扫描条件的记录

参数列表：无参数

2.方法实现思路：

根据当前 rid_ 中存储的记录位置，调用 fh_->get_record(rid_, context_) 方法获取完整的元组数据，并以 std::unique_ptr<RmRecord> 的形式返回。

3. 方法实现难点：

需要确保 rid_ 是有效的，并且 Context 对象被正确传递给 get_record。该方法假设在调用前已经通过 beginTuple() 或 nextTuple() 正确定位到了一条满足条件的记录。

（6）实现 `SeqScanExecutor::eval_cond` 函数

```
bool SeqScanExecutor::eval_cond(const RmRecord *rec, const Condition &cond, const std::vector<ColMeta> &rec_cols);
```

1.方法声明：

方法名：eval_cond

返回类型：bool

功能：判断单条记录是否满足给定的单个选择条件

参数列表：

参数	类型	含义
rec	const RmRecord*	指向待检查元组的指针
cond	const Condition&	单个选择条件
rec_cols	const std::vector&	元组的列元数据

2.方法实现思路：

1. 使用 get_col() 获取条件左部表达式（列）的元数据，并计算数据指针（rec->data + lhs_col->offset）
2. 判断条件右部是常量值（cond.is_rhs_val == true）还是列，并获取其类型和数据指针
3. 调用 ix_compare() 函数对左右两边的值进行比较，得到比较结果（-1, 0, 1）
4. 根据条件中的比较运算符（如 OP_EQ, OP_LT 等）使用 switch 语句返回最终的布尔比较结果

3. 方法实现难点：

难点在于正确处理不同数据类型（ColType）的比较，ix_compare 辅助函数的正确使用是关键。同时需要正确解析 Condition 结构，区分右部是值还是列。当右部是列时，需要从记录中获取对应列的数据；当右部是常量值时，需要从 cond.rhs_val.raw->data 中获取数据。

任务 2：投影算子的实现

补全 ProjectionExecutor 类，实现投影算子，具体完成下列任务。

（1）阅读代码

阅读 src/execution目录下的代码。

src/execution/executor_abstract.h

src/execution/executor_projection.h

理解 ProjectionExecutor 类的设计，并回答下列问题：

prev_的作用是什么?

答：prev_ 是指向子节点算子（如 SeqScanExecutor）的智能指针，投影算子通过它来获取待处理的输入元组。

（2）实现 `ProjectionExecutor::beginTuple` 函数

```
void ProjectionExecutor::beginTuple();
```

1.方法声明：

方法名：beginTuple

返回类型：void

功能：使用子节点算子定位到第一条结果元组

参数列表：无参数

2.方法实现思路：

直接调用子节点算子 prev_ 的 beginTuple() 方法，将定位操作传递给下游算子，使其准备好提供第一条元组。

3. 方法实现难点：

该方法为简单的调用传递，无明显难点。投影算子本身不需要额外的定位逻辑，只需将请求转发给子节点。

（3）实现 `ProjectionExecutor::nextTuple` 函数

```
void ProjectionExecutor::nextTuple();
```

1.方法声明：

方法名：nextTuple

返回类型：void

功能：使用子节点算子定位到下一条结果元组

参数列表：无参数

2.方法实现思路：

直接调用子节点算子 prev_ 的 nextTuple() 方法，使其定位到下一条元组。

3. 方法实现难点：

该方法为简单的调用传递，无明显难点。投影操作不改变元组的数量和顺序，只改变每条元组包含的列。

（4）实现 `ProjectionExecutor::is_end` 函数

```
void ProjectionExecutor::is_end();
```

1.方法声明：

方法名：is_end

返回类型：bool

功能：判断子节点算子是否已经没有输入元组了

参数列表：无参数

2.方法实现思路：

调用子节点算子 prev_ 的 is_end() 方法，并返回其结果，以判断是否还有输入元组。

3. 方法实现难点：

该方法为简单的调用传递，无明显难点。投影算子的结束状态完全依赖于子节点的状态。

（5）实现 `ProjectionExecutor::Next` 函数

```
std::unique_ptr<RmRecord> ProjectionExecutor::Next();
```

1.方法声明：

方法名：Next

返回类型：std::unique_ptr

功能：获取子节点算子的当前元组，并返回只包含投影列的新元组

参数列表：无参数

2.方法实现思路：

1. 调用 `prev_->Next()` 从子节点获取原始元组
2. 创建一个新的 `RmRecord` 对象，其大小为投影后所有列的总长度 `len_`
3. 遍历需要投影的列（通过 `sel_idxs_` 获取每个投影列在原始元组中的索引）
4. 对于每个投影列，使用 `memcpy` 从原始元组中复制数据到新元组的相应位置：

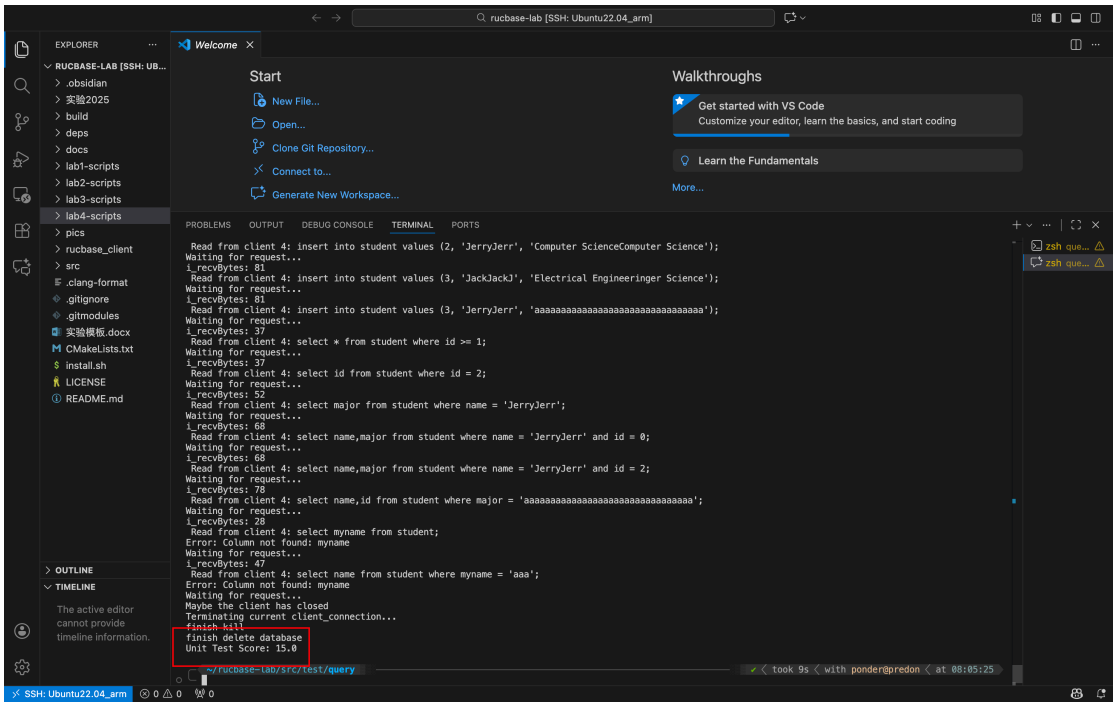
- 源地址：`prev_rec->data + prev_col.offset`
- 目标地址：`proj_rec->data + proj_col.offset`
- 复制长度：`proj_col.len`

5. 返回这个只包含投影列的新元组

3. 方法实现难点：

难点在于精确计算每个投影列在原始元组和新元组中的偏移量（offset）和长度（len），并使用 memcpy 正确地复制数据。需要理解 sel_idxs_ 存储的是要投影的列在子节点 cols_ 中的索引，而 cols_ 中存储的是投影后新元组的列元数据（已经重新计算了偏移量）。

（6）单元测试



通过单元测试

任务 3：嵌套循环连接算子的实现

补全 NestedLoopJoinExecutor 类，实现嵌套循环连接算子，具体完成下列任务。

（1）阅读代码

理解 NestedLoopJoinExecutor 类的设计，并回答下列问题：

1. left_ 和 right_ 的作用是什么?

- `left_`：指向左子节点算子的指针，作为嵌套循环的外层循环，提供左表数据。
- `right_`：指向右子节点算子的指针，作为嵌套循环的内层循环，提供右表数据。

（2）实现 NestedLoopJoinExecutor::beginTuple 函数

```
void NestedLoopJoinExecutor::beginTuple();
```

1.方法声明：

方法名：beginTuple

返回类型：void

功能：定位到左右子节点算子的第一条结果元组，并找到第一对满足连接条件的元组对

参数列表：无参数

2.方法实现思路：

1. 调用 left_->beginTuple() 定位到左表的第一条元组
2. 如果左表不为空（!left_->is_end()），则调用 right_->beginTuple() 定位到右表的第一条元组
3. 进入嵌套循环结构：
    1. 外层循环遍历左表的所有元组
    2. 内层循环遍历右表的所有元组
4. 对于每一对（左表元组，右表元组），调用 eval_conds() 检查是否满足所有连接条件
5. 找到第一对满足条件的元组后，停止循环并返回
6. 如果内层循环结束但未找到匹配，则移动外层循环到下一条记录，并重置内层循环（调用 right_->beginTuple()）
7. 如果外层循环结束，设置 isend = true

3. 方法实现难点：

需要正确处理两个子节点的初始化，并实现嵌套循环逻辑来找到第一个匹配的元组对。难点在于正确管理两层循环的状态，特别是在内层循环结束后需要重置右表扫描并推进左表扫描。

（3）实现 NestedLoopJoinExecutor::nextTuple 函数

`void NestedLoopJoinExecutor::nextTuple();`

1.方法声明：

方法名：nextTuple

返回类型：void

功能：从当前位置开始，找到下一对满足连接条件的元组对

参数列表：无参数

2.方法实现思路：

1. 首先确保迭代器已初始化（assert(!left_->is_end())）
2. 首先尝试在内层循环（右表）中移动，即调用 right_->nextTuple()
3. 循环检查右表的剩余元组，看是否与当前左表元组匹配：
    1. 对每对元组调用 eval_conds() 检查连接条件
    2. 找到匹配则返回
4. 如果右表扫描完毕（right_->is_end()），则：
    1. 在外层循环（左表）中移动：调用 left_->nextTuple()
    2. 重置右表扫描：调用 right_->beginTuple()
5. 重复嵌套循环过程，直到找到下一对满足连接条件的元组，或者扫描完所有元组对
6. 如果左表也扫描完毕，设置 isend = true

3. 方法实现难点：

难点在于正确管理嵌套循环的状态。当内层循环结束时，需要正确地推进外层循环并重置内层循环。需要特别注意的是，在调用 right_->nextTuple() 后仍需检查是否到达末尾，以及在推进左表后需要重新开始扫描右表。

（4）实现 NestedLoopJoinExecutor::is_end 函数

`void NestedLoopJoinExecutor::is_end();`

1.方法声明:

方法名：is_end

返回类型：bool

功能：判断是否已经没有结果元组了

参数列表：无参数

2.方法实现思路：

通过一个布尔成员变量 isend 来判断。当外层循环（左表）也扫描结束后，即 left_->is_end() 为真时，将 isend 设为 true。

3. 方法实现难点：

需要确保在 nextTuple() 和 beginTuple() 的逻辑中，当所有可能的元组对都检查完毕后，能正确地更新 isend 标志。该标志反映的是整个嵌套循环连接操作是否已经完成。

（5）实现 NestedLoopJoinExecutor::Next 函数

```
std::unique_ptr<RmRecord> NestedLoopJoinExecutor::Next();
```

1.方法声明：

方法名：Next

返回类型：std::unique_ptr

功能：获取左右子节点算子的当前元组，并返回拼接后的连接结果元组

参数列表：无参数

2.方法实现思路：

1. 分别调用 left_->Next() 和 right_->Next() 获取当前匹配的左右元组
2. 创建一个大小为左右元组长度之和的新 RmRecord 对象（len_ = left_->tupleLen() + right_->tupleLen()）
3. 使用 memcpy 先将左元组的完整数据复制到新元组的起始位置：
    1. 目标地址：join_rec->data
    2. 源地址：left_rec->data
    3. 复制长度：left_->tupleLen()
4. 接着将右元组的完整数据复制到新元组中紧随左元组数据之后的位置：
    1. 目标地址：join_rec->data + left_->tupleLen()
    2. 源地址：right_rec->data
    3. 复制长度：right_->tupleLen()
5. 返回这个拼接而成的新元组

3. 方法实现难点：

关键在于正确计算拼接后元组的总长度，以及右元组数据在新元组中的起始偏移量（即左元组的长度）。这种水平拼接的方式保证了连接后的元组包含了左右表的所有列。

（6）实现 NestedLoopJoinExecutor::eval_cond 函数

```
bool NestedLoopJoinExecutor::bool eval_cond(const RmRecord *lhs_rec, const RmRecord *rhs_rec, const Condition &cond, const std::vector<ColMeta> &rec_cols);
```

1.方法声明：

方法名：eval_cond

返回类型：bool

功能：判断一对左右表元组是否满足给定的单个连接条件

参数列表：

参数	类型	含义
lhs_rec	const RmRecord*	指向左表的当前元组
rhs_rec	const RmRecord*	指向右表的当前元组
cond	const Condition&	单个连接条件
rec_cols	const std::vector&	连接后元组的列元数据

2.方法实现思路：

1. 获取连接条件左部表达式（列）的元数据（使用 get_col(rec_cols, cond.lhs_col)）
2. 根据左部列的 offset 判断它属于左表还是右表：
    1. 如果 lhs_col->offset < left_->tupleLen()，则从 lhs_rec 中获取数据
    2. 否则，从 rhs_rec 中获取数据，并调整偏移量（lhs_col->offset - left_->tupleLen()）
3. 判断条件右部是常量值（cond.is_rhs_val == true）还是列：
    1. 如果是常量值，从 cond.rhs_val.raw->data 获取数据
    2. 如果是列，同样根据其 offset 判断来源（左表或右表），并获取相应数据

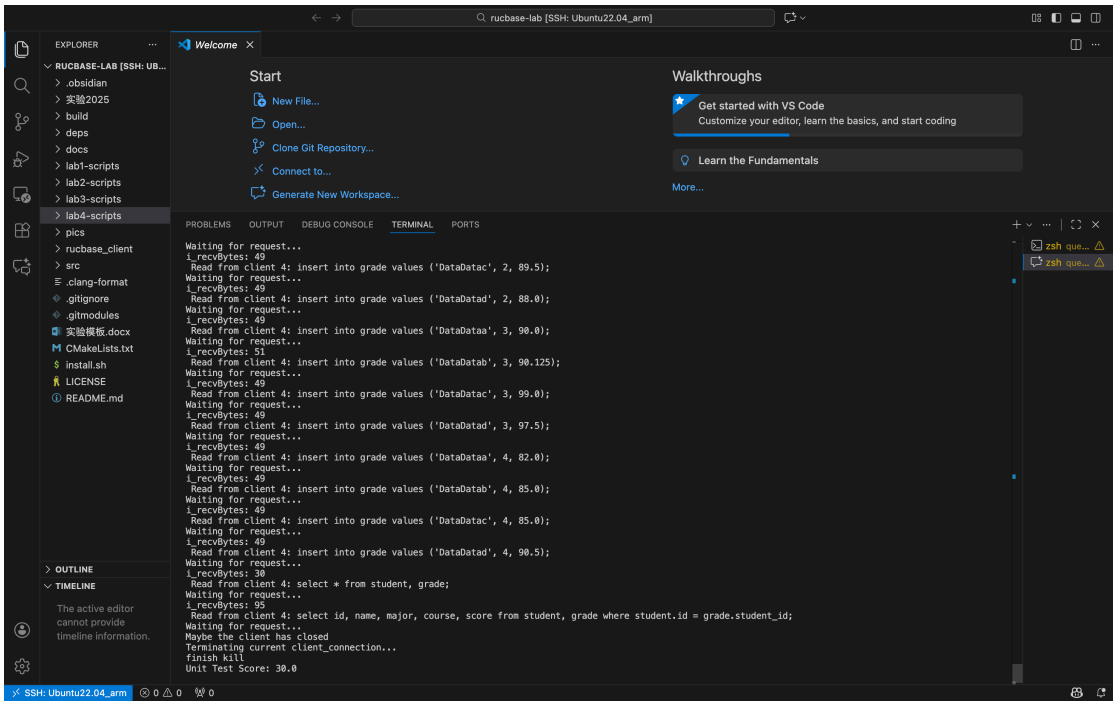
4. 调用 ix_compare() 函数对左右两边的值进行比较

5. 根据比较运算符（cond.op）使用 switch 语句返回最终的布尔比较结果

3. 方法实现难点：

最主要的难点是根据列的偏移量正确判断其来源（左表或右表），并相应地调整数据指针的计算方式。这是实现连接条件判断的核心。需要理解的关键点是：连接后的元组在逻辑上是左右表元组的水平拼接，因此通过偏移量与左表长度的比较即可确定列的来源。当列来自右表时，需要减去左表的长度才能得到在右表元组中的实际偏移量。

（7）单元测试



通过单元测试

任务 4：修改算子的实现

（1）阅读代码

理解 UpdateExecutor 类的设计，并回答下列问题：

rids_ 是如何得到的?

答：rids_ 是一个 Rid 向量，它在执行计划生成阶段，由上游的扫描或索引节点提供。这些 Rid 指

向了所有满足 WHERE 子句条件的、需要被更新的元组。

（2）实现 UpdateExecutor::Next 函数

```
std::unique_ptr<RmRecord> UpdateExecutor::Next();
```

1.方法声明：

方法名：Next

返回类型：std::unique_ptr

功能：遍历所有需要更新的记录，根据 SET 子句修改相应字段的值，并将修改后的记录写回磁盘

参数列表：无参数

2.方法实现思路：

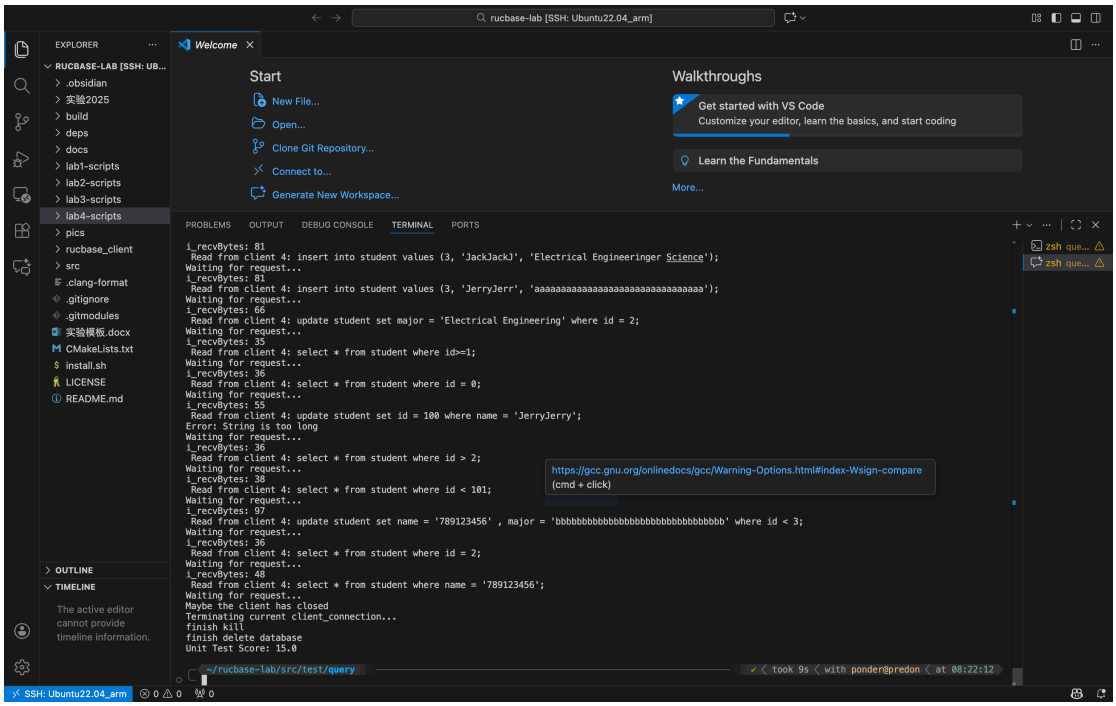
1. 使用 for 循环遍历 rids_ 列表中的每一个 Rid
2. 对于每个 Rid，首先调用 fh_->get_record(rid, context_）获取原始记录
3. 遍历 set_clauses_（即 SET 子句列表），对于每一个要修改的字段：
    1. 使用 tab_.get_col() 获取目标列的元数据
    2. 检查待更新值的类型与列类型是否兼容（如果不兼容，抛出 IncompatibleTypeError）
    3. 初始化右部值的原始数据（调用 set_clause.rhs.init_raw(col->len)）
    4. 使用 memcpy 将新值的数据覆盖到从 get_record() 获取的记录缓存区的相应位置：
        1. 目标地址：rec->data + col->offset
        2. 源地址：set_clause.rhs.raw->data
        3. 复制长度：col->len
4. 调用 fh_->update_record(rid, rec->data, context_)，将修改后的记录数据写回磁盘
5. 由于 UPDATE 操作不向上层返回元组，因此最后返回 nullptr

3. 方法实现难点：

难点在于需要先读出（get_record）再修改，然后写回（update_record），这是一个"读-改-写"的过程。在修改内存中的记录时，必须使用 memcpy 确保数据按字节正确复制，特别是对于不同长度的字符串和数值类型。另一个难点是需要对每个 SET 子句进行类型检查，确

保类型兼容性，并正确初始化右部值的原始数据。

（3）单元测试



通过单元测试

任务 5：删除算子的实现

（1）阅读代码

理解 DeleteExecutor 类的设计，并回答下列问题：

1. rids_ 是如何得到的？

答：与 UpdateExecutor 类似，rids_ 是由上游节点（如扫描操作）提供的、所有满足 WHERE 子句条件的、需要被删除的元组的 Rid 列表。

（2）实现 DeleteExecutor::Next 函数

```
std::unique_ptr<RmRecord> DeleteExecutor::Next();
```

1.方法声明：

方法名：Next

返回类型：std::unique_ptr

功能：遍历所有需要删除的记录，并调用底层文件句柄的 delete_record 函数删除这些记录

参数列表：无参数

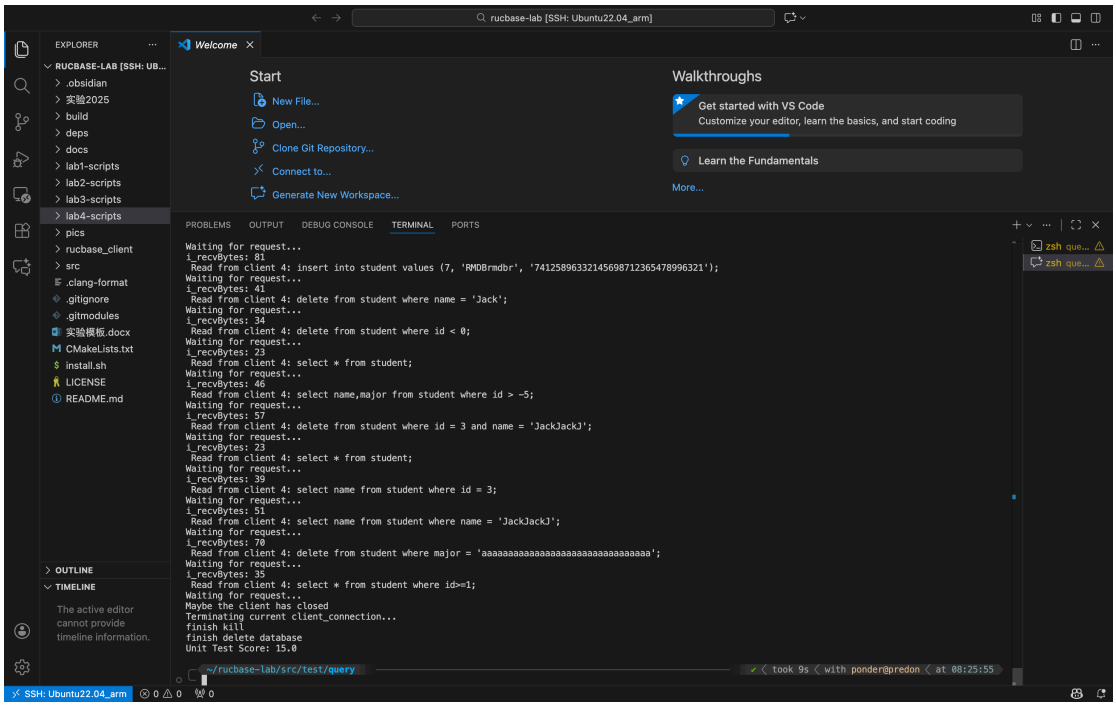
2.方法实现思路：

1. 使用 for 循环遍历 rids_ 列表中的每一个 Rid
2. 对每个 Rid，直接调用 fh_->delete_record(rid, context_) 来删除磁盘上对应的记录
3. DELETE 操作也不向上层返回元组，因此最后返回 nullptr

3. 方法实现难点：

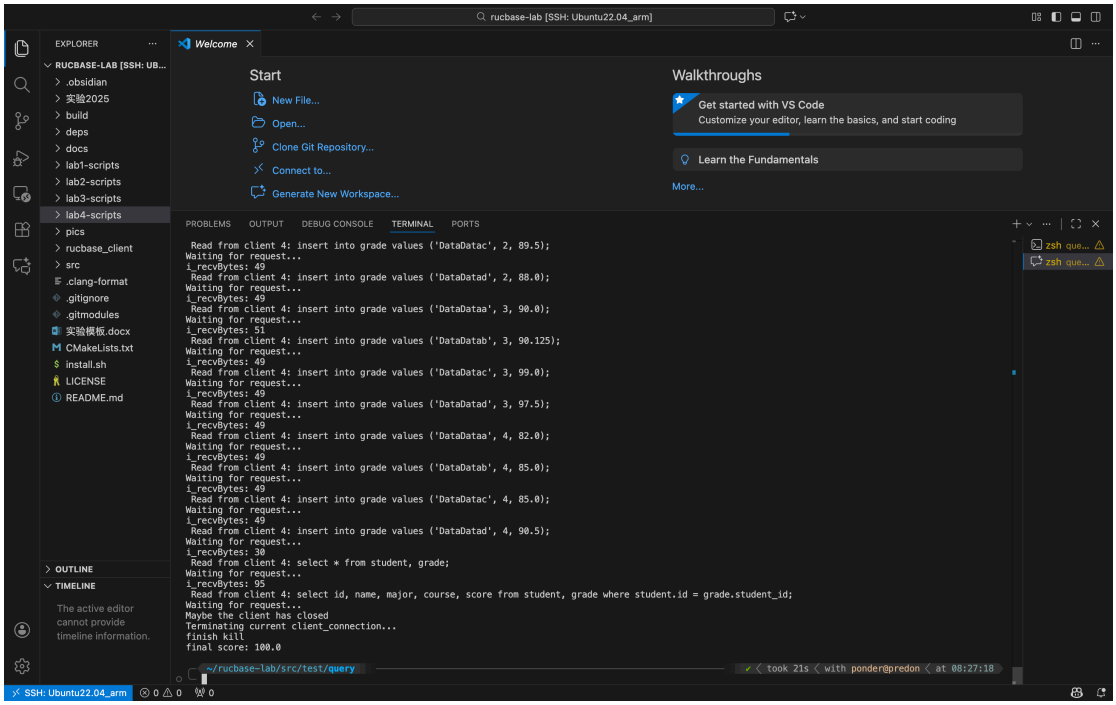
此方法的实现相对直接，主要依赖于底层 RmFileHandle::delete_record 的正确实现。本身逻辑较为简单，无明显难点。只需要简单地遍历所有要删除的记录 ID，逐个调用删除函数即可。与 UpdateExecutor 相比，删除操作不需要读取记录内容或进行任何修改，因此实现更加简洁。

（3）单元测试



通过单元测试

全部单元测试



通过全部单元测试

四、实验总结

通过这次实验，我学到了：

1. 火山模型接口：明确掌握了统一的执行接口（beginTuple/nextTuple/Next/is_end）与“拉取式”管线化执行思想。

2. 执行器职责：理解并实现了 SeqScan/Projection/NLJ/Update/Delete 各算子的核心职责与协作关系。

3. 条件评估：学会用列元数据解析左右表达式并用 ix_compare 做类型感知比较，结合短路机制提升效率。

4. 记录布局与内存：熟悉基于列 offset/len 的定长字段拷贝、memcpy 的高效搬运，以及 RmRecord 与 rid 的使用。

5. 连接细节：能正确处理连接结果的列归属与偏移计算（以左表元组长度为阈），避免左右错读。

6. 迭代器与边界：强化了对迭代器状态管理与边界检查的意识，避免 is_end/next 顺序不当导致的越界。

7. 测试与规范：按用例验证算子功能，遵循 select 输出与错误写 failure 的格式要求，形

成面向测试的实现习惯。

8. 优化思维: 形成谓词/投影下推、选择小表为外层、索引嵌套循环连接等执行层优化意识，为后续哈希/归并连接铺垫。

9. 工程健壮性: 在空指针、类型匹配、异常路径等方面加强了防御式编程与错误处理能力。