

实验4：数据操纵的实现

邹兆年，哈尔滨工业大学计算学部，znzou@hit.edu.cn

<https://gitee.com/HIT-DB/hit-db-class-rucbase-lab>

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

阅读 `src/execution` 目录下的代码。

- `src/execution/execution_manager.h`
- `src/execution/execution_manager.cpp`
- `src/execution/executor_abstract.h`
- `src/execution/executor_seq_scan.h`

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？

(2) 实现 `SeqScanExecutor::beginTuple` 函数

函数声明：

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

功能：定位到表中第一条满足选择条件的元组。

实现：参考代码注释。基本实现逻辑如下：

1. 创建该表的记录迭代器 `scan_`。
2. 使用记录迭代器 `scan_` 扫描表中元组，直至遇到第一条满足选择条件的元组，将该元组的Rid记录在 `rid_` 中。

(3) 实现 `SeqScanExecutor::nextTuple` 函数

函数声明：

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

功能：定位到表中下一条满足选择条件的元组。

实现：参考代码注释。基本实现逻辑如下：使用记录迭代器 `scan_` 继续扫描表中元组，直至遇到下一条满足选择条件的元组，将该元组的Rid记录在 `rid_` 中。

(4) 实现 `SeqScanExecutor::Next` 函数

函数声明：

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

功能：返回下一条结果元组。

实现：参考代码注释。基本实现逻辑如下：返回 `rid_` 标识的元组。需要调用实验2实现的 `RmFileHandle::get_record` 函数。

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：

1. `prev_` 的作用是什么？

(2) 实现 `ProjectionExecutor::beginTuple` 函数

函数声明：

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

功能：定位到第一条结果元组。

实现：参考代码注释。基本实现逻辑如下：使用执行器 `prev_` 定位到子节点算子的第一条结果元组。

(3) 实现 `ProjectionExecutor::nextTuple` 函数

函数声明：

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

功能：定位到下一条结果元组。

实现：参考代码注释。基本实现逻辑如下：使用执行器 `prev_` 定位到子节点算子的下一条结果元组。

(4) 实现 `ProjectionExecutor::Next` 函数

函数声明：

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

功能：返回下一条结果元组。

实现：参考代码注释。基本实现逻辑如下：

1. 创建结果元组（`RmRecord` 类）。
2. 将子节点算子的当前元组进行投影，填充结果元组。
3. 返回结果元组指针。

(5) 单元测试

单元测试代码在文件 `src/test/query/query_unit_test.py` 中。

执行下列命令，进行单元测试。

```
cd src/test/query
python query_unit_test.py basic_query_test2.sql
```

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

阅读 `src/execution` 目录下的代码。

- `src/execution/executor_abstract.h`
- `src/execution/executor_nestedloop_join.h`

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

1. `left_` 和 `right_` 的作用是什么？

(2) 实现 `NestedLoopJoinExecutor::beginTuple` 函数

函数声明：

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

功能：定位到第一条结果元组。

实现：参考代码注释。基本实现逻辑如下：

1. 使用执行器 `left_` 定位到左子节点算子的第一条结果元组。
2. 使用执行器 `right_` 定位到右子节点算子的第一条结果元组。

(3) 实现 `NestedLoopJoinExecutor::nextTuple` 函数

函数声明：

```
void NestedLoopJoinExecutor::nextTuple();
```

功能：定位到下一条结果元组。

实现：参考代码注释。基本实现逻辑如下：

1. 如果执行器 `right_` 未处于右子节点算子结果的末尾，则使用执行器 `right_` 定位到右子节点算子的下一条结果元组。
2. 如果执行器 `right_` 未于右子节点算子结果的末尾，则使用执行器 `left_` 定位到左子节点算子的下一条结果元组，使用执行器 `right_` 定位到右子节点算子的第一条结果元组。

(4) 实现 `NestedLoopJoinExecutor::Next` 函数

函数声明：

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

功能：返回下一条结果元组。

实现：参考代码注释。基本实现逻辑如下：

1. 创建结果元组（`RmRecord` 类）。
2. 将 `left_` 和 `right_` 的当前元组进行连接，填充结果元组。
3. 返回结果元组指针。

(5) 单元测试

单元测试代码在文件 `src/test/query/query_unit_test.py` 中。

执行下列命令，进行单元测试。

```
cd src/test/query
python query_unit_test.py basic_query_test5.sql
```

任务4：修改算子的实现

补全 `UpdateExecutor` 类，实现修改算子，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_update.h`

理解 `UpdateExecutor` 类的设计，并回答下列问题：

1. `rids_` 是如何得到的？

(2) 实现 `UpdateExecutor::Next` 函数

函数声明：

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

功能：修改元组。

实现：修改 `rids_` 中记录的所有元组。需要调用实验2实现的 `RmFileHandle::update_record` 函数。可以参考 `InsertExecutor::Next` 函数的实现。

(3) 单元测试

单元测试代码在文件 `src/test/query/query_unit_test.py` 中。

执行下列命令，进行单元测试。

```
cd src/test/query
python query_unit_test.py basic_query_test3.sql
```

任务5：删除算子的实现

补全 `DeleteExecutor` 类，实现删除算子，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_delete.h`

理解 `DeleteExecutor` 类的设计，并回答下列问题：

1. `rids_` 是如何得到的?

(2) 实现 `DeleteExecutor::Next` 函数

函数声明:

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

功能: 删除元组。

实现: 删除 `rids_` 中记录的所有元组。需要调用实验2实现的 `RmFileHandle::delete_record` 函数。可以参考 `InsertExecutor::Next` 函数的实现。

(3) 单元测试

单元测试代码在文件 `src/test/query/query_unit_test.py` 中。

执行下列命令, 进行单元测试。

```
cd src/test/query
python query_unit_test.py basic_query_test4.sql
```

全部单元测试

单元测试代码在文件 `src/test/query/query_test_basic.py` 中。

执行下列命令, 进行单元测试。

```
cd src/test/query
python query_test_basic.py
```

四、考核方法

- 实验完成情况 (80%)。根据单元测试通过情况和代码理解情况给分。
 - 任务1: 顺序扫描算子的实现 (20%)
 - 任务2: 投影算子的实现 (20%)
 - 任务3: 嵌套循环连接算子的实现 (20%)
 - 任务4: 修改算子的实现 (10%)
 - 任务5: 删除算子的实现 (10%)
- 实验报告 (20%)。根据实验报告的完整性、科学性和规范性评分。

五、补充说明

- 在本测试中, 要求把select语句的输出写入到指定文件中, 写入逻辑已经在select_from函数中给出, 不要修改写入格式。对于执行错误的SQL语句, 需要打印failure到output.txt文件中。
- Rucbase默认多表连接的执行计划是右深连接树。

3. `execuotr_abstract` 类中定义的虚函数在其子类中没有添加TODO，但仍需实现，否则会导致程序无法正常运行。