哈尔滨工业大学

<<数据库系统>>

实验报告三

(2023 年度春季学期**)**

姓名:	
学号:	
学院:	
教师:	

实验三

一、实验目的

掌握关系连接操作的实现算法，理解算法的 I/O 复杂性，使用高级语言实现重要的关系连接操作算法。

掌握关系数据库中查询优化的原理，理解查询优化算法在执行过程中的时间开销和空间开销，使用高级语言实现重要的查询优化算法。

二、实验环境

Windows 11 操作系统、python 语言、PyCharm 软件。

三、实验过程及结果

1.关系连接算法的实现

见目录 join 下的三个源文件：main.py、extmem.py 及 relation.py。其中，main.py 是可运行的主文件，extmem.py 是用 python 实现的 ExtMem 程序库，relation.py 中定义了 R、S 两个关系表的类模型，便于模块化调用。本部分主要的程序实现都在 join/main.py 中，下面对该文件中的主要函数实现思路进行说明。

1.1 基本功能

本部分涉及以下基本功能：

1) generate_data(r, s)：用于生成关系数据，对 R、S 的各属性按照对应要求的值域进行随机赋值，生成对应数量的元组。
2) write_relation(relation, required_blk, addr)：生成数据后，用于将初始的关系调用 extmem 模块的 buffer.writeBlockToDisk(addr, blk_num)写入磁盘块，该函数中会按规定的块大小，将关系组织成字符串后写入块，并设置最后一项为下一块起始地址，例如，打开程序生成的文件“4.blk”，此处应存放关系 R 的 7 个条目，并最后指向 5，从图 1 可见确实如此。函数返回写入的最后一个磁盘块号+1，便于下一次写入时确定写入起始位置。

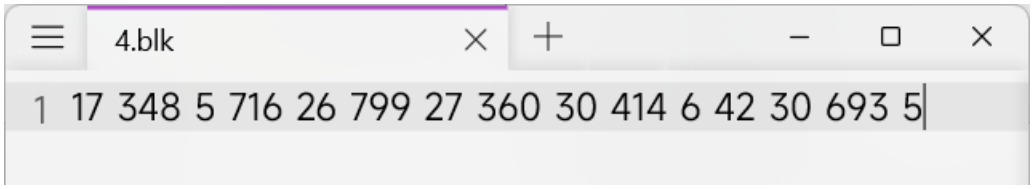

图 1 模拟磁盘块文件 4.blk 的内容

3) write_data(data_set, required_blk, tuple_number, addr)：用于将数据写入磁盘块，基本思路与 write_relation 类似，但涉及到对数据长度的处理，例

如投影操作中需要写入的数据长度仅为 1 个整形，因此引入参数 tuple_number，确定一块中要写入的元组数量。

1.2 关系选择算法实现

见 main.py 中的 select(relation, attribute, value, addr)，实现的是线性搜索算法，思路为：通过调用 buffer.readBlockFromDisk(present_blk_number)完成申请缓冲区并逐块读取选中关系，对每个条目进行比对，保留成功匹配的数据至 data 列表（通过 buffer.getNewBlockInBuffer()为 data 申请缓冲区）、输出至屏幕，最后，将 data 通过缓冲区写入磁盘块。

1.3 关系投影算法实现

见 main.py 中的 project(relation, attribute, addr)，思路为：逐块读取选中关系，将选中的属性保存至 data 列表并输出至屏幕，最后，将 data 通过缓冲区写入磁盘块。

1.4 Nested-Loop Join 实现

见 main.py 中的 nest_loop_join(…)，实现的是 Block-based NLJ。思路为，首先确定块数较少的关系作为外层关系以减少时间复杂度，随后对外层关系的每一块，逐块读取内层关系，并对所有元组进行比对连接，将连接结果保存至 data 列表，并向屏幕输出成功连接的两个关系中的两个元组，最后，将 data 通过缓冲区写入磁盘块。

1.5 Hash Join 实现

见 main.py 中的 hash_join(relation_first, relation_second, number_of_bucket, blk_number)，其中参数 number_of_bucket 指定桶数，哈希函数即为对桶数取模，同模同桶，程序中调用时，设置的参数为 7。思路为：读入关系后首先进行 Hash 分桶，然后两个关系中对应桶进行连接，其余部分程序类似于 NLJ。

1.6 Sort-Merge Join 实现

见 main.py 中的 sort_merge_join(relation_first, relation_second, blk_number)，思路为：逐块读取两个关系的数据，之后使用 sorted()进行排序，然后进行归并，归并时，移动第二个关系的元组，进行匹配连接，其余部分程序类似于前两个连接算法。

1.7 主程序及运行结果

见 main.py，主程序即对之前所述的每个功能进行依次组织调用，并输出提示信息。输出的结果如下（省略了许多数据，请重点关注各个操作的正确性，以及三个连接算法连接条数的一致性）：

```
数据创建完成。
关系 R 写入至块号 0 至 15。
关系 R 写入至块号 16 至 47。
选择操作结果：
R ('40', '316')
共找到 1 条数据。
选择操作结果保存至块号 48 至 48。
投影操作结果：
29
40
…
17
```

4
共找到 112 条数据。
投影操作结果保存至块号 49 至 56。
Nest-Loop-Join 结果：
R (26, 760) - S (26, 898)
R (40, 316) - S (40, 237)
…
R (7, 640) - S (7, 477)
R (10, 985) - S (10, 593)
共连接 601 条数据。
Nest-Loop Join 操作结果保存至块号 57 至 177。
Hash_Join 结果：
R (33, 466) - S (33, 986)
R (33, 466) - S (33, 432)
…
R (17, 547) - S (17, 331)
R (17, 547) - S (17, 873)
共连接 601 条数据。
Hash Join 操作结果保存至块号 178 至 298。
Sort_Merge_Join 结果：
R [1, 933] - S [1, 522]
R (1, 933) - S (1, 376)
…
R (40, 316) - S (40, 452)
R (40, 316) - S (40, 77)
共连接 601 条数据。
Sort-Merge Join 操作结果保存至块号 299 至 419。
缓冲区已释放

2. 查询优化算法的设计

见目录 query 下的两个源文件：main.py 和 syntaxtree.py。其中，main.py 是可运行的主文件，syntaxtree.py 中定义了语法树类模型 SyntaxTree，便于模块化调用。

2.1 SyntaxTree 类

有如下属性：lchild、rchild——即左右子树，可以被赋值为其他 SyntaxTree 对象；op——该节点的操作，即连接、投影、选择等；cond——该节点的条件，如选择操作中的某属性等于某值、投影操作中所选中的属性等，对应于所给语句中操作符后面“[]”中所对应的内容；rel——关系节点，若 rel 非空，则该节点即为确定关系表的叶节点。在绘制树时，lchild 和 rchild 形成树的结构（边），而通过 op、cond、rel 属性来形成节点所显示的文字。

2.2 处理 sql 语句构造语法树

见 main.py 中的 parsesql(sql_statement)，首先用 split()以空格作为分隔符将 sql 语句划分为词列表，然后逐个分析各个词。

若遇到操作符 SELECT、PROJECT，则将对应单词赋给该树的 op 属性，随

后方括号中的内容赋给 cond 属性；

若遇到 JOIN，则创建两个新的 SyntaxTree 作为该节点的左右子树，并将 JOIN 前后的两个词（即所连接的两个关系）赋给左右节点的 rel。

若遇到小括号“()”，则其内部也是示例语句的格式，则通过递归调用 parsesql() 来构造这条语句的语法树，并作为当前节点的左子树。

2.3 查询优化实现

查询优化分为两步：下移选择操作和下移投影操作。

对于下移选择操作，见 main.py 中的 down_select(syntax_tree, sql, relation2)，思路为，首先判断输入语法树根节点的 op，若为选择操作，则找到该选择操作条件中涉及的关系，选择条件参数 sql，然后递归处理其左子树；若为投影操作，跳过根节点，递归处理左子树；若为连接操作，则需要将选择操作移至连接操作下方：新建两棵语法树作为该节点左右子树，操作符 op 均为 SELECT，而条件 cond 即为参数 sql 中两棵子树对应的两个关系的两个选择条件最后，将新的左右子树节点插入到当前节点和其对应子树节点之间。

对于下移投影操作，见 main.py 中的 down_proj(syntax_tree, sql, relation2, same)，思路为：判断语法树根节点 op，若为选择则跳过该节点递归处理其左子树；若为投影，则投影属性作为 sql，找到投影属性所在的关系作为 relation2，找到属性相交的关系作为 same，递归处理其左子树；若为连接，则需要下移投影操作：为连接的两个关系新建两棵左右子树，并将涉及到投影（若 sql 即投影属性非空，则左子树一定涉及投影；若 sql 中有两个属性或者 same 非空，则两个关系都涉及到投影，启用右子树）的子树 op 设为 PROJECT，投影属性 cond 通过 sql 和 same 形成。最后，将新的左右子树节点插入到当前节点和其对应子树节点之间。

2.4 绘制语法树

调用了 graphviz 包的 Graph 图绘制模块，在程序中用函数 showtree(syntax_tree, g, parent)进行语法树的绘制。g 为生成的 Graph 对象，用 g.edge()在一个节点与其子树间生成边。若当前节点 op 非空，则通过其 op 和 cond 形成该节点显示的文字，然后判断其左右节点，若存在，则加边，并判断左右子树的 rel 是否为空（是否叶节点），若空，则非叶节点，递归处理；若非空，则为叶节点，rel 即为所要显示的文字。当整棵树处理完毕，用 g.render()即可生成语法树图片。

2.5 主程序

将显示每条 sql 语句封装成了一个 show()函数，每条 sql 语句处理流程如下：通过 parsesql 构造语法树，并 showtree；然后进行优化，依次进行 down_select 和 down_proj，获得优化后的语法树再进行 showtree。选择实验指导书的前三条 sql 语句，输出结果如图 2 至图 4 所示。

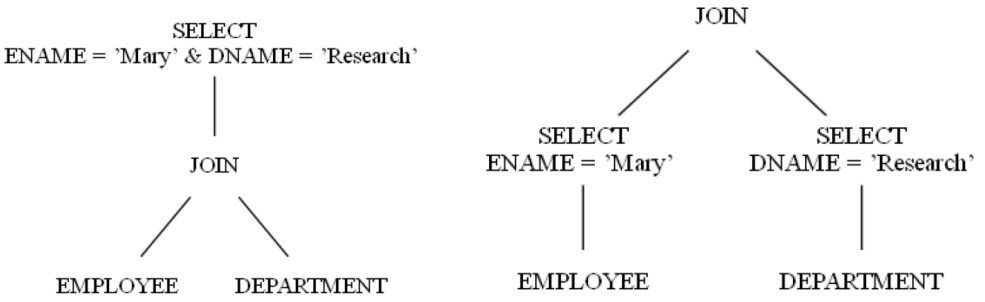

图 2 语句 1 优化前（左）与优化后（右）的语法树

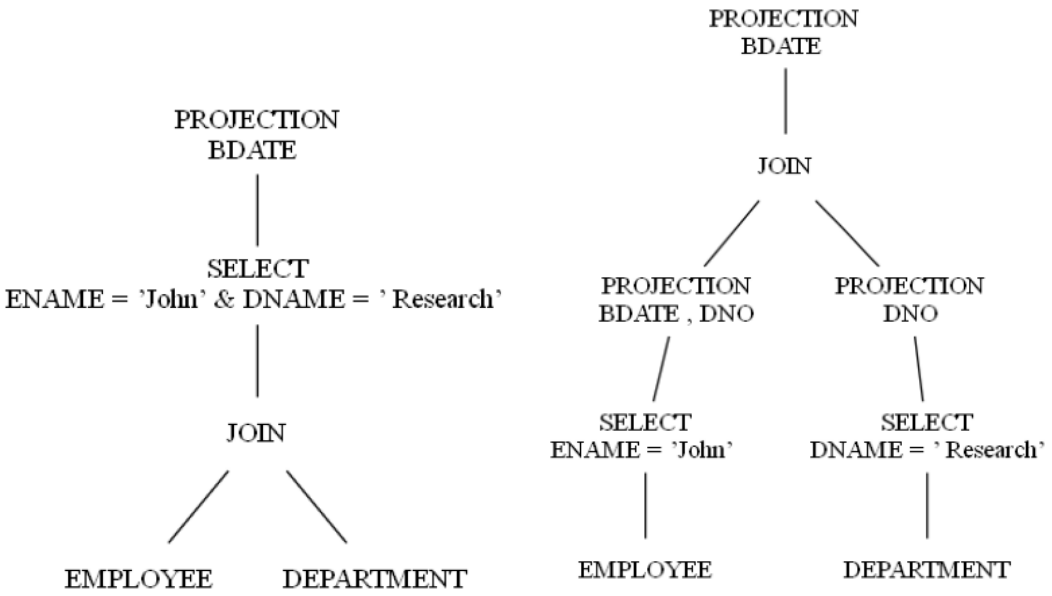

图 3 语句 2 优化前（左）与优化后（右）的语法树

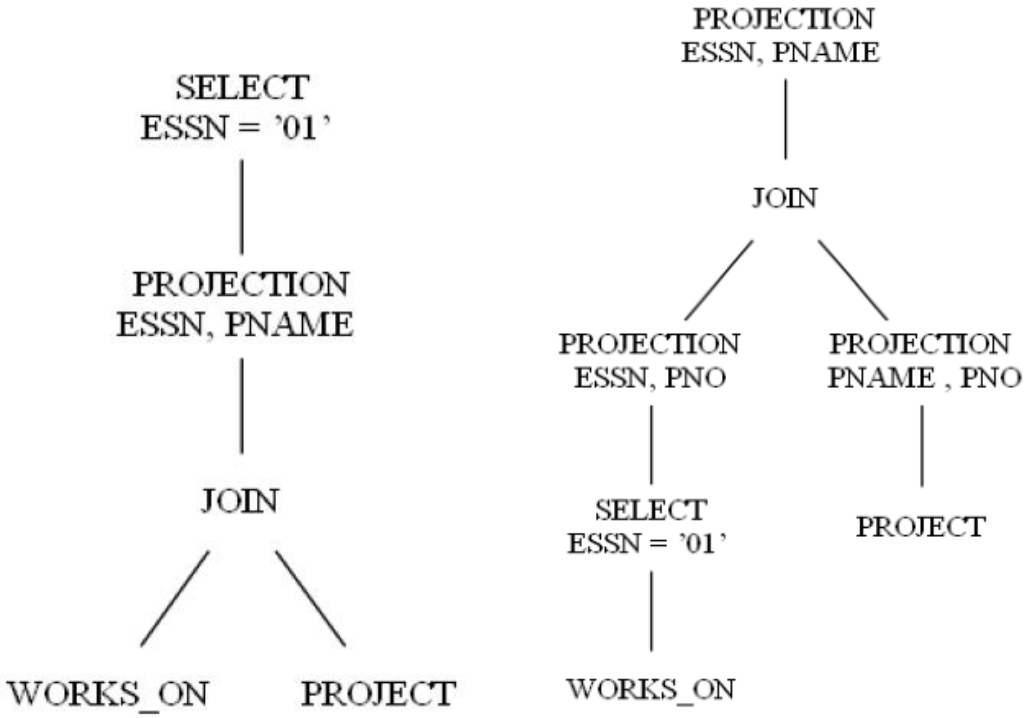

图 4 语句 3 优化前（左）与优化后（右）的语法树

四、实验心得

实验初期编程遇到较大难度，但渡过开头后，思路逐渐形成，就得以顺利完成了。