哈尔滨工业大学

<<数据库系统>>

实验报告三

(2024 年度秋季学期)

姓名:	刘子康
学号:	2022113416
学院:	计算学部
教师:	李东博

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，Visual Studio Code 1.95.3，Python 3.10，PyCharm 2022 社区版

三、实验过程及结果

此次实验完成了基于 ExtMem 程序库，模拟外存磁盘块存储和存取过程的关系连接操作的实现算法；并设计查询优化算法，对三条查询语句生成的查询执行树进行优化。

（一）关系连接算法的实现

运行时使用 gcc 编译连接源代码 connection.c 和头文件代码 extmem.c：gcc -o connection connection.c extmem.c，并使用./connection 运行程序，结果存放于同目录下的 disk 文件夹中。其中 1-48 为生成的关系 R 和 S，50-99 为 R.A=40 的元组，100-149 为 S.C=60 的元组，150-200 为关系 R 的 A 属性的投影，200-399 为嵌套循环连接后的元组，400-599 为哈希连接后的元组，600-799 为排序归并连接后的元组。

1.1 数据准备

1.1.1 任务

编写程序，随机生成关系 R 和 S，使得 R 中包含 16 * 7 = 112 个元组，S 中包含 32 * 7 = 224 个元组。关系 R 具有两个属性 A 和 B，其中 A 和 B 的属性值均为 int 型（4 字节），A 的值域为[1, 40]，B 的值域为[1, 1000]；关系 S 具有两个属性 C 和 D，其中 C 和 D 的属性值均为 int 型（4 字节），C 的值域为[20, 60]，D 的值域为[1, 1000]，即 R 和 S 的每个元组的大小均为 8 字节。

使用 ExtMem 程序库建立两个关系 R 和 S 的物理存储，存储形式为磁盘块序列 $B_1, B_2, \ldots, B_n$，其中 B_i 的最后 4 字节存放 B_{i+1} 的地址。块的大小设置为 64 字节，缓冲区大小设置为 512+8=520 个字节，则每块可存放 7 个元组和 1 个后继磁盘块地址，缓冲区内可最多存放 8

个块。

1.1.2 具体实现

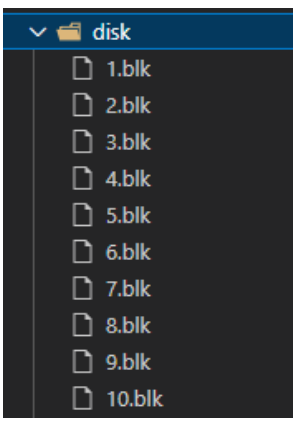
以关系 R 的生成和存储为例：首先定义关系 R 的结构体 Tuple_R，包含两个 int 类型数据 A 和 B。然后，编写 generateRelation_R 函数，定义一个长度为 112 的结构体数组 R，利用 rand()函数随机生成关系 R 的元组。最后，通过两层循环将所有元组模拟写入磁盘，内层循环在一个磁盘块中写入 7 个元组；外层循环申请新的磁盘块，更新和写入下一磁盘块地址，并使用 writeBlockToDisk 函数将填充好的磁盘块写入磁盘指定地址。

注：对 extmem 的 writeBlockToDisk 函数进行修改，将存放结果的文件地址改为代码同目录下的 disk 文件夹中。

```
void generateRelation_R(unsigned int startAddr, Buffer *buf)
{
    Tuple_R R[NUM_R_TUPLES];
    unsigned int currentAddr = startAddr;
    int tupleIdx = 0;

    // 随机生成关系R的元组
    for (int i = 0; i < NUM_R_TUPLES; i++) {
        R[i].A = rand() % 40 + 1;    // A的值域为[1, 40]
        R[i].B = rand() % 1000 + 1; // B的值域为[1, 1000]
    }

    while (tupleIdx < NUM_R_TUPLES) {
        unsigned char *blkPtr = getNewBlockInBuffer(buf);
        if (blkPtr == NULL) {
            perror("Fail to get a new block in buffer.\n");
            return;
        }

        for (int i = 0; i < 7 && tupleIdx < NUM_R_TUPLES; i++, tupleIdx++) {
            *(Tuple_R*)(blkPtr + i * 8) = R[tupleIdx];
        }
        // 下一磁盘块地址
        unsigned int nextBlockAddr = (tupleIdx < NUM_R_TUPLES) ? currentAddr + 1 : 0;
        *(unsigned int*)(blkPtr + 7 * 8) = nextBlockAddr;

        // 将数据块写入磁盘
        if (writeBlockToDisk(blkPtr, currentAddr, buf) != 0) {
            perror("Fail to write block to disk.\n");
            return;
        }
        currentAddr++;
    }
}
```

1.1.3 结果

结果存放在 disk 文件夹中，以下为部分结果：



1.2 关系选择算法

1.2.1 任务

基于 ExtMem 程序库，选出 R.A=40 或 S.C=60 的元组，并将结果存放在磁盘上。

1.2.2 具体实现

以选择和存储 R.A=40 的元组为例：在 selectRelation 函数中，首先定义一个结构体数组 selected_R，用于存储满足条件的 R 元组，接着初始化一个结果缓冲区 resultBuf，用于暂存选中的元组。然后，通过两层循环进行筛选，外层循环使用 readBlockFromDisk 函数，根据关系 R 的磁盘块起始地址从磁盘读取磁盘块；内层循环遍历块中的元组，并将 A 值为 40 的元组存储到 selected_R 数组中。最后，将关系 R 所有符合要求的元组写入到磁盘中。

```
119     // 选择 R.A=40 的元组
120     for (int addr = startAddr_R; addr < startAddr_R + 16; addr++) {
121         unsigned char *blkPtr = readBlockFromDisk(addr, buf);
122         if (blkPtr == NULL) {
123             perror("Fail to read block from disk.\n");
124             return;
125         }
126 
127         for (int i = 0; i < 7 && tupleIdx_R < NUM_R_TUPLES; i++, tupleIdx_R++) {
128             Tuple_R *tuple = (Tuple_R*)(blkPtr + i * 8);
129             if(tuple->A == 40) {
130                 printf("Find a tuple of A:%d and B:%d.\n", tuple->A, tuple->B);
131                 selected_R[count_R++] = *tuple;
132             }
133         }
134 
135         freeBlockInBuffer(blkPtr, buf);
136     }
```

1.2.3 结果

```
===============     selectRelation     ===============
Find a tuple of A:40 and B:344.
Find a tuple of A:40 and B:470.
Find a tuple of A:40 and B:848.
Find a tuple of A:40 and B:848.
Find a tuple of A:40 and B:727.
Find a tuple of A:40 and B:974.
Find a tuple of C:60 and D:216.
Find a tuple of C:60 and D:859.
Find a tuple of C:60 and D:960.
Find a tuple of C:60 and D:785.
```

1.3 关系投影算法

1.3.1 任务

基于 ExtMem 程序库，对关系 R 上的 A 属性进行投影，并将结果存放在磁盘上。

1.3.2 具体实现

在 projectAFromRelation_R 函数中，首先定义一个结构体数组 projected_A，用于存储关系 R 的 A 属性，接着初始化一个结果缓冲区 resultBuf，用于暂存投影的 A 属性。然后，通过两层循环对 A 属性进行投影，外层循环读取关系 R 的磁盘块；内层循环遍历块中的元组，并将 A 属性存储到 projected_A 数组中。最后，将关系 R 所有 A 属性写入到磁盘中。

1.3.3 结果

以下为投影的部分结果：

```
=============== projectAFromRelation_R ===============
Projection A of R: 36
Projection A of R: 35
Projection A of R: 36
Projection A of R: 54
Projection A of R: 35
Projection A of R: 52
Projection A of R: 47
Projection A of R: 36
Projection A of R: 35
Projection A of R: 36
Projection A of R: 54
Projection A of R: 35
Projection A of R: 52
Projection A of R: 47
Projection A of R: 15
Projection A of R: 29
Projection A of R: 24
Projection A of R: 30
Projection A of R: 29
Projection A of R: 20
Projection A of R: 27
Projection A of R: 27
Projection A of R: 38
```

1.4 关系连接算法

1.4.1 任务

基于 ExtMem 程序库，实现三种连接算法，对关系 R 和 S 计算 R.A 连接 S.C，并将结果存放在磁盘上。

三种算法均初始化一个结果缓冲区 resultBuf，用于暂存连接后的元组，并在最后将连接后的所有元组写入到磁盘中，以下具体实现中不再赘述。

1.4.2 嵌套循环连接算法（NLJ）

编写 nestedLoopJoin 函数，首先定义一个结构体数组 RS，用于存储连接后的关系 R 和关系 S 的元组。然后，通过四层循环将关系 R 和关系 S 按条件连接，外层循环读取关系 R 和 S 的磁盘块；内层两循环遍历块中的元组，并将 R.A=S.C 的元组存储到 RS 数组中。

```
279     // 对关系R和S计算R.A连接S.C
280     for (int addr_S = startAddr_S; addr_S < startAddr_S + 32; addr_S++) {
281         unsigned char *blkPtr_S = readBlockFromDisk(addr_S, buf);
282         if (blkPtr_S == NULL) {
283             perror("Fail to read block from disk.\n");
284             return;
285         }
286 
287         for (int addr_R = startAddr_R; addr_R < startAddr_R + 16; addr_R++) {
288             unsigned char *blkPtr_R = readBlockFromDisk(addr_R, buf);
289             if (blkPtr_R == NULL) {
290                 perror("Fail to read block from disk.\n");
291                 return;
292             }
293 
294             int flag = -1;
295             for (int i = 0; i < 7; i++) {
296                 Tuple_S *tuple_S = (Tuple_S*)(blkPtr_S + i * 8);
297                 for (int j = 0; j < 7; j++) {
298                     Tuple_R *tuple_R = (Tuple_R*)(blkPtr_R + j * 8);
299                     if (tuple_R->A == tuple_S->C) {
300                         RS[count].A = tuple_R->A;
301                         RS[count].B = tuple_R->B;
302                         RS[count].C = tuple_S->C;
303                         RS[count++].D = tuple_S->D;
304                         printf("Join R: A=%d, B=%d and S: C=%d, D=%d.\n", tuple_R->A, tuple_R->B, tuple_S->C, tuple_S->D);
305                     }
306                 }
307             }
308 
309             freeBlockInBuffer(blkPtr_R, buf);
310         }
311 
312         freeBlockInBuffer(blkPtr_S, buf);
313     }
```

```
===============     nestedLoopJoin     ===============
Join R: A=21, B=505 and S: C=21, D=622.
Join R: A=21, B=505 and S: C=21, D=622.
Join R: A=24, B=126 and S: C=24, D=143.
Join R: A=21, B=210 and S: C=21, D=622.
Join R: A=24, B=196 and S: C=24, D=143.
Join R: A=33, B=836 and S: C=33, D=870.
Join R: A=24, B=164 and S: C=24, D=143.
Join R: A=21, B=66 and S: C=21, D=622.
Join R: A=24, B=450 and S: C=24, D=143.
Join R: A=33, B=734 and S: C=33, D=870.
Join R: A=33, B=734 and S: C=33, D=870.
Join R: A=30, B=281 and S: C=30, D=602.
Join R: A=21, B=505 and S: C=21, D=893.
Join R: A=30, B=281 and S: C=30, D=602.
Join R: A=21, B=505 and S: C=21, D=893.
Join R: A=30, B=238 and S: C=30, D=602.
Join R: A=32, B=758 and S: C=32, D=796.
Join R: A=21, B=210 and S: C=21, D=893.
Join R: A=28, B=193 and S: C=28, D=966.
Join R: A=21, B=66 and S: C=21, D=893.
Join R: A=30, B=281 and S: C=30, D=517.
Join R: A=30, B=281 and S: C=30, D=517.
Join R: A=38, B=423 and S: C=38, D=545.
Join R: A=38, B=300 and S: C=38, D=545.
Join R: A=38, B=63 and S: C=38, D=545.
Join R: A=38, B=300 and S: C=38, D=545.
```

1.4.3 哈希连接算法

首先定义一个哈希桶结构体，包含关系 R 的属性和一个链表指针（关系 R 元组数较少，建立哈希表代价小）。编写 hashJoin 函数，定义一个结构体数组 RS，用于存储连接后的元组。然后，对关系 R 建立哈希表，外层循环读取关系 R 的磁盘块；内层循环遍历块中的元组，使用简单取模作为哈希函数，将元组根据 A 的值映射到某个哈希桶中，并插入到链表头部。最后是探测阶段，外层循环读取关系 S 的磁盘块，内层关系遍历块中的元组，根据哈希函数查找匹配的桶，将 R.A=S.C 的元组存储到 RS 数组中。

```
354     // 对关系R建立哈希表
355     HashBucket *hashTable[HASH_TABLE_SIZE] = {0};
356     for (int addr_R = startAddr_R; addr_R < startAddr_R + 16; addr_R++) {
357         unsigned char *blkPtr_R = readBlockFromDisk(addr_R, buf);
358         if (blkPtr_R == NULL) {
359             perror("Fail to read block from disk.\n");
360             return;
361         }
362 
363         for (int i = 0; i < 7 && tupleIdx_R < NUM_R_TUPLES; i++, tupleIdx_R++) {
364             Tuple_R *tuple_R = (Tuple_R*)(blkPtr_R + i * 8);
365             int hashIdx = tuple_R->A % HASH_TABLE_SIZE;    // 哈希函数，简单取模
366             HashBucket *newBucket = (HashBucket*)malloc(sizeof(HashBucket));;
367             newBucket->A = tuple_R->A;
368             newBucket->B = tuple_R->B;
369             newBucket->next = hashTable[hashIdx];
370             hashTable[hashIdx] = newBucket;    // 插入到链表头
371         }
372 
373         freeBlockInBuffer(blkPtr_R, buf);
374     }
```

```
384         for (int i = 0; i < 7 && tupleIdx_S < NUM_S_TUPLES; i++, tupleIdx_S++) {
385             Tuple_S *tuple_S = (Tuple_S*)(blkPtr_S + i * 8);
386             int hashIdx = tuple_S->C % HASH_TABLE_SIZE;    // 哈希函数，简单取模
387 
388             // 查找匹配的桶
389             HashBucket *bucket = hashTable[hashIdx];
390             while (bucket != NULL) {
391                 if (bucket->A == tuple_S->C) {
392                     RS[count].A = bucket->A;
393                     RS[count].B = bucket->B;
394                     RS[count].C = tuple_S->C;
395                     RS[count++].D = tuple_S->D;
396                     printf("Join R: A=%d, B=%d and S: C=%d, D=%d.\n", bucket->A, bucket->B, tuple_S->C, tuple_S->D);
397                 }
398                 bucket = bucket->next;
399             }
400         }
```

```
===============        hashJoin        ===============
Join R: A=24, B=450 and S: C=24, D=143.
Join R: A=24, B=164 and S: C=24, D=143.
Join R: A=24, B=196 and S: C=24, D=143.
Join R: A=24, B=126 and S: C=24, D=143.
Join R: A=21, B=66 and S: C=21, D=622.
Join R: A=21, B=210 and S: C=21, D=622.
Join R: A=21, B=505 and S: C=21, D=622.
Join R: A=21, B=505 and S: C=21, D=622.
Join R: A=33, B=734 and S: C=33, D=870.
Join R: A=33, B=734 and S: C=33, D=870.
Join R: A=33, B=836 and S: C=33, D=870.
Join R: A=32, B=758 and S: C=32, D=796.
Join R: A=28, B=193 and S: C=28, D=966.
Join R: A=30, B=238 and S: C=30, D=602.
Join R: A=30, B=281 and S: C=30, D=602.
Join R: A=30, B=281 and S: C=30, D=602.
Join R: A=21, B=66 and S: C=21, D=893.
Join R: A=21, B=210 and S: C=21, D=893.
Join R: A=21, B=505 and S: C=21, D=893.
Join R: A=21, B=505 and S: C=21, D=893.
Join R: A=33, B=734 and S: C=33, D=497.
Join R: A=33, B=734 and S: C=33, D=497.
Join R: A=33, B=836 and S: C=33, D=497.
Join R: A=38, B=82 and S: C=38, D=545.
Join R: A=38, B=300 and S: C=38, D=545.
Join R: A=38, B=63 and S: C=38, D=545.
Join R: A=38, B=300 and S: C=38, D=545
```

1.4.4 排序归并连接算法

编写 compareTuple_R 和 compareTuple_S 函数，用于 C 语言标准库的 qsort 函数，对关系 R 按 R.A 排序和对关系 S 按 S.A 排序。

编写 sortMergeJoin 函数，首先定义结构体数组 sortedR、sortedS、RS，用于存储关系 R、S 排序后的元组和连接后的元组。然后，对关系 R 按 R.A 排序和对关系 S 按 S.A 排序，读取磁盘块并将所有元组暂存到 sortedR 和 sortedS 数组中，使用 qsort 函数进行递增排序。最后执行归并连接，同时遍历关系 R 和 S 的所有元组，连接属性值小的一方的索引递增，若 R.A=S.C，则遍历所有匹配的 R 元组和 S 元组（因为存在重复值），将其存储到 RS 数组中。

```
// 对Tuple_R按R.A排序
int compareTuple_R(const void *a, const void *b)
{
    return ((Tuple_R*)a)->A - ((Tuple_R*)b)->A;
}

// 对Tuple_S按S.C排序
int compareTuple_S(const void *a, const void *b)
{
    return ((Tuple_S*)a)->C - ((Tuple_S*)b)->C;
}
```

```
    // 排序
    qsort(sortedR, count_R, sizeof(Tuple_R), compareTuple_R);
    qsort(sortedS, count_S, sizeof(Tuple_S), compareTuple_S);

    // 执行排序归并连接
    int i = 0, j = 0;
    while (i < count_R && j < count_S) {
        if (sortedR[i].A < sortedS[j].C) {
            i++;
        } else if (sortedR[i].A > sortedS[j].C) {
            j++;
        } else {  // sortedR[i].A == sortedS[j].C
            // 遍历所有匹配的R元组
            int tempIdx_R = i, tempIdx_S = j;
            while (tempIdx_R < count_R && sortedR[tempIdx_R].A == sortedR[i].A) {
                // 遍历所有匹配的S元组
                tempIdx_S = j;
                while (tempIdx_S < count_S && sortedS[tempIdx_S].C == sortedS[j].C) {
                    RS[count].A = sortedR[tempIdx_R].A;
                    RS[count].B = sortedR[tempIdx_R].B;
                    RS[count].C = sortedS[tempIdx_S].C;
                    RS[count++].D = sortedS[tempIdx_S].D;
                    printf("Join R: A=%d, B=%d and S: C=%d, D=%d.\n", sortedR[tempIdx_R].A, sortedR[tempIdx_R].B, sortedS[tempIdx_S].C, sortedS[tempIdx_S].D);
                    tempIdx_S++;
                }
                tempIdx_R++;
            }
            i = tempIdx_R;
            j = tempIdx_S;
        }
    }
```

```
===============      sortMergeJoin     ===============
Join R: A=20, B=276 and S: C=20, D=398.
Join R: A=20, B=276 and S: C=20, D=63.
Join R: A=20, B=276 and S: C=20, D=92.
Join R: A=20, B=276 and S: C=20, D=124.
Join R: A=20, B=110 and S: C=20, D=398.
Join R: A=20, B=110 and S: C=20, D=63.
Join R: A=20, B=110 and S: C=20, D=92.
Join R: A=20, B=110 and S: C=20, D=124.
Join R: A=20, B=200 and S: C=20, D=398.
Join R: A=20, B=200 and S: C=20, D=63.
Join R: A=20, B=200 and S: C=20, D=92.
Join R: A=20, B=200 and S: C=20, D=124.
Join R: A=20, B=200 and S: C=20, D=398.
Join R: A=20, B=200 and S: C=20, D=63.
Join R: A=20, B=200 and S: C=20, D=92.
Join R: A=20, B=200 and S: C=20, D=124.
Join R: A=20, B=276 and S: C=20, D=398.
Join R: A=20, B=276 and S: C=20, D=63.
Join R: A=20, B=276 and S: C=20, D=92.
Join R: A=20, B=276 and S: C=20, D=124.
Join R: A=21, B=210 and S: C=21, D=247.
Join R: A=21, B=210 and S: C=21, D=372.
Join R: A=21, B=210 and S: C=21, D=78.
Join R: A=21, B=210 and S: C=21, D=622.
Join R: A=21, B=210 and S: C=21, D=769.
Join R: A=21, B=210 and S: C=21, D=769.
Join R: A=21, B=210 and S: C=21, D=803
```

（二）查询优化算法的设计

选择前三条查询语句：

• SELECT [ENAME = ‘Mary’ & DNAME = ‘Research’] (EMPLOYEE JOIN DEPARTMENT)

• PROJECTION [BDATE] (SELECT [ENAME = ‘John’ & DNAME = ‘Research’] (EMPLOYEE JOIN DEPARTMENT))

• SELECT [ESSN = ‘01’] (PROJECTION [ESSN, PNAME] (WORKS_ON JOIN PROJECT))

2.1 任务

设计一个语法分析器，能够识别上述关系代数语句。对其进行解析，生成对应的查询执行树，并打印输出树的结构。对生成的查询执行树进行优化，并打印输出最后优化后的查询执行树。

2.2 设计语法分析器并生成查询执行树

首先，定义一个查询树节点的类 QueryTreeNode，包含子节点、操作符和条件属性，并定义__str__方法，用于返回包含操作符和条件的完整字符串。

```
7   # 查询树节点
8   class QueryTreeNode:
        单元测试 | 注释生成 | 代码解释 | 缺陷检测
9       def __init__(self, op='', info=''):
10          self.child = []     # 子节点
11          self.op = op        # 操作符
12          self.info = info    # 条件
13
        单元测试 | 注释生成 | 代码解释 | 缺陷检测
14      def __str__(self):
15          return (self.op if self.op else '') + (' ' + self.info if self.info else '') # 当作为字符串使用时，输出操作符和条件
```

然后，以空格为分割符对查询语句进行切片，存储到 tokens 列表中。遍历每个 token，

若为“SELECT”或“PROJECTION”，则将其和其后面的条件存入到 node 实例的 op 和 info 属性中；若为“JOIN”，则将其存入到 node 实例的 op 属性，并在两个子节点添加连接操作的两个关系；若为“(”，则代表其后面为查询子句，则通过递归调用处理括号中间的子句，并将最终结果存入到上一层的子节点中。至此生成了一个查询执行树，可通过 output_tree()函数进行打印。

```
def get_tree(query: str):
    tokens, idx, node = query.split(), 0, QueryTreeNode()
    while idx < len(tokens):
        token = tokens[idx]
        if token == 'SELECT' or token == 'PROJECTION':
            end = tokens.index(']', idx)    # 找到最近的]的索引
            node.op, node.info = token, ' '.join(tokens[idx + 2:end])   # info存入选择条件
            idx = end + 1
        elif token == 'JOIN':
            node.op = token
            node.child.append(QueryTreeNode(info=tokens[idx - 1]))  # 连接操作的第一个关系
            node.child.append(QueryTreeNode(info=tokens[idx + 1]))  # 连接操作的第二个关系
            idx += 1
        elif token == '(':  # 括号内为查询子句
            count, idy = 1, idx + 1
            while idy < len(tokens) and count > 0:
                if tokens[idy] == '(':
                    count += 1
                elif tokens[idy] == ')':
                    count -= 1
                idy += 1
            node.child.append(get_tree(' '.join(tokens[idx + 1:idy - 1])))  # 递归调用，并将结果加入上层子节点
            idx = idy
        else:
            idx += 1
    return node
```

2.3 查询优化

从查询树的根节点开始，若节点操作符为“SELECT”或“PROJECTION”，则记录下二者条件，并递归优化其子树；若节点操作符为“JOIN”，则将上层的选择操作下推（投影不下推），更新节点顺序。递归结束后，整个查询执行树已经优化完毕，可通过 output_tree()函数进行打印。

```
def optimize(node: QueryTreeNode, info_lst=None) -> QueryTreeNode:
    # 遇到选择和投影时，记录下二者条件，并递归优化其子树
    if node.op == 'SELECT':
        node = optimize(node.child[0], node.info.split('&'))
    elif node.op == 'PROJECTION':
        node.child[0] = optimize(node.child[0], info_lst)
    # 遇到连接时，将上层的选择操作下推（投影不下推）
    elif node.op == 'JOIN':
        node0 = QueryTreeNode(op='SELECT', info=info_lst[0])
        node0.child.append(node.child[0])
        node.child[0] = node0
        if len(info_lst) > 1:
            node1 = QueryTreeNode(op='SELECT', info=info_lst[1])
            node1.child.append(node.child[1])
            node.child[1] = node1
    return node
```

2.4 结果

```
SELECT [ ENAME = 'Mary' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT )
Origianl query1:
SELECT ENAME = 'Mary' & DNAME = 'Research'
    JOIN
         EMPLOYEE
         DEPARTMENT
Optimized query1:
JOIN
    SELECT ENAME = 'Mary'
         EMPLOYEE
    SELECT  DNAME = 'Research'
         DEPARTMENT
=============================================================================================
PROJECTION [ BDATE ] ( SELECT [ ENAME = 'John' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT ) )
Origianl query2:
PROJECTION BDATE
    SELECT ENAME = 'John' & DNAME = 'Research'
        JOIN
             EMPLOYEE
             DEPARTMENT
Optimized query2:
PROJECTION BDATE
    JOIN
        SELECT ENAME = 'John'
             EMPLOYEE
        SELECT  DNAME = 'Research'
             DEPARTMENT
=============================================================================================
SELECT [ ESSN = '01' ] ( PROJECTION [ ESSN, PNAME ] ( WORKS_ON JOIN PROJECT ) )
Origianl query3:
SELECT ESSN = '01'
    PROJECTION ESSN, PNAME
        JOIN
             WORKS_ON
             PROJECT
Optimized query3:
PROJECTION ESSN, PNAME
    JOIN
        SELECT ESSN = '01'
             WORKS_ON
         PROJECT

进程已结束,退出代码0
```

四、实验心得

（1）实现了基于 ExtMem 程序库，模拟外存磁盘块存储和存取过程的关系选择，投影，以及 NLJ、哈希连接、排序归并连接等重要连接操作的算法，对于查询执行的过程有了更深的理解；

（2）实现了三条查询语句生成的查询执行树的优化，了解了查询优化的原理，掌握了生成查询执行树和选择下推、投影下推等查询优化的方法。

（3）在实现任务一时，曾出现自定义的缓冲区溢出和无法申请新的磁盘块的报错，检查发现在申请的磁盘块使用完毕后没有及时清空缓冲区，导致缓冲区满了。故在每个外层循环因读取磁盘中的元组而申请新的磁盘块，并且使用完毕后，使用 freeBlockInBuffer 函数清空缓冲区，解决了这一问题。

（4）在任务一读取磁盘块中的元组时，由于最后一个磁盘块可能并未填满，因此直接通过偏移地址获取元组可能会读取到非关系中的元组的值，因此在循环退出条件中添加元组索引判断，若已读取的元组数到达关系总元组数，则直接停止读取并退出循环。