

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:	
学号:	
学院:	计算学部
教师:	

实验三

一、实验目的

1. 关系连接算法的实现：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。
2. 查询优化算法的设计：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 10 操作系统、MySQL 关系数据库管理系统、VSCODE、Python 语言，C 语言、pyqt5 框架

三、实验过程及结果

3.1 实验过程

3.1.1 关系连接算法实现

(1) 数据准备

要求：

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 个字节），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 个字节）。C 的值域为[20, 60]，D 的值域为[1, 1000]。

使用 ExtMem 程序库建立两个关系 R 和 S 的物理存储。关系的物理存储形式为磁盘块序列 B₁, B₂, ..., B_n，其中 B_i 的最后 4 个字节存放 B_{i+1} 的地址。

R 和 S 的每个元组的大小均为 8 个字节。块的大小设置为 64 个字节，缓冲区大小设置为 512+8=520 个字节（每一块前有一个 bit 做表示位，表示该块是否被占用）。这样，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。编写程序，随机生成关系 R 和 S，使得 R 中包含 $16 \times 7 = 112$ 个元组，S 中包含 $32 \times 7 = 224$ 个元组。

```
1. void generate_relation(Buffer buf, unsigned int start_addr, int
   block_count, int value_a_min, int value_a_max, int value_b_min, int
   value_b_max) {
2.     unsigned int addr = start_addr;
```

```
3.
4.     for (int i = 0; i < block_count; i++) {
5.         // 获取新块
6.         unsigned char block = getNewBlockInBuffer(buf);
7.         if (!block) {
8.             printf("Failed to allocate block\n");
9.             return;
10.        }
11.
12.        unsigned char ptr = block;
13.
14.        // 在块内写入元组
15.        for (int j = 0; j < (BLOCK_SIZE - sizeof(int)) / TUPLE_SIZE; j++)
16.        {
17.            int attr1 = value_a_min + rand() % (value_a_max - value_a_min
18.            + 1);
19.            int attr2 = value_b_min + rand() % (value_b_max - value_b_min
20.            + 1);
21.
22.            memcpy(ptr, &attr1, sizeof(int));
23.            ptr += sizeof(int);
24.            memcpy(ptr, &attr2, sizeof(int));
25.            ptr += sizeof(int);
26.        }
27.
28.        // 写入下一个块的地址到最后 4 个字节
29.        if (i < block_count - 1) {
30.            int next_addr = addr + 1;
31.            memcpy(block + BLOCK_SIZE - sizeof(int), &next_addr,
32.            sizeof(int));
33.        } else {
34.            int end_addr = 0; // 表示链表结束
35.            memcpy(block + BLOCK_SIZE - sizeof(int), &end_addr,
36.            sizeof(int));
37.        }
38.
39.        // 打印当前块内容
40.        printf("Generated content for block %d:\n", addr);
41.        print_block_content(block, addr);
42.
43.        // 写入磁盘
44.        if (writeBlockToDisk(block, addr, buf) != 0) {
45.            printf("Failed to write block %d\n", addr);
46.            return;
47.        }
48.        addr++;
49.    }
50.
51.    printf("Generated %d blocks starting from address %d\n",
52.    block_count, start_addr);
53. }
```

这段代码主要功能是生成两个关系 R 和 S ，每个关系由一系列随机生成的元组组成，并将它们存储在模拟的磁盘块内。首先，为每个关系分配一定数量的块，然后在每个块内填充元组，元组包含随机生成的属性值。每块的最后四个字节存储指向下一个块的地址，最后通过 `writeBlockToDisk` 函数将块写入磁盘。代码同时提供了块内容打印功能，以便验证生成数据的正确性。

Generated content for block 126:	Generated content for block 127:	Generated content for block 128:
Block 126:	Block 127:	Block 128:
Tuple (49, 460)	Tuple (55, 840)	Tuple (20, 116)
Tuple (42, 439)	Tuple (55, 364)	Tuple (54, 158)
Tuple (37, 389)	Tuple (58, 795)	Tuple (48, 492)
Tuple (58, 234)	Tuple (46, 848)	Tuple (33, 952)
Tuple (41, 682)	Tuple (38, 463)	Tuple (55, 22)
Tuple (24, 359)	Tuple (54, 391)	Tuple (57, 741)
Tuple (20, 700)	Tuple (60, 792)	Tuple (29, 31)
Next block: 127	Next block: 128	Next block: 129

在磁盘块中的表现形式：

```
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 126.blk
      49      460      42      439
      37      389      58      234
      41      682      24      359
      20      700         0      127
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 127.blk
      55      840      55      364
      58      795      46      848
      38      463      54      391
      60      792         0      128
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 128.blk
      20      116      54      158
      48      492      33      952
      55      22       57      741
      29      31         0      129
```

(2) 选择操作

要求：

实现关系选择算法：基于 `ExtMem` 程序库，使用高级语言实现关系选择算法，选出 $R.A=40$ 或 $S.C=60$ 的元组，并将结果存放在磁盘上。

核心代码：

```
48. while (current_addr != 0) {
49.     // 读取磁盘块到缓冲区
50.     block = readBlockFromDisk(current_addr, buf);
51.     if (!block) {
52.         printf("Failed to read block %d\n", current_addr);
53.         return;
54.     }
55.
56.     unsigned char ptr = block;
57.
58.     // 遍历块内的元组
59.     for (int i = 0; i < (BLOCK_SIZE - sizeof(int)) / 8; i++) {
60.         int attr_value;
61.         memcpy(&attr_value, ptr + rel->attr_offsets[attr_index],
62.             sizeof(int)); // 获取指定属性值
63.
64.         if (attr_value == query_value) {
65.             // 输出到控制台
```

```

65.         int attr1, attr2;
66.         memcpy(&attr1, ptr, sizeof(int));
67.         memcpy(&attr2, ptr + 4, sizeof(int));
68.         printf("| %-8d | %-8d |\n", attr1, attr2);
69.
70.         // 如果结果块为空或已满，分配新块
71.         if (result_block == NULL || result_count == (BLOCK_SIZE -
sizeof(int)) / 8) { // result_count ->[0,(BLOCK_SIZE - sizeof(int))
/ 8 - 1]
72.             if (result_block != NULL) {
73.                 // 写入结果块的后继地址
74.                 int next_addr = result_addr + 1;
75.                 memcpy(result_block + BLOCK_SIZE - sizeof(int),
&next_addr, sizeof(int));
76.                 writeBlockToDisk(result_block, result_addr, buf);
77.                 freeBlockInBuffer(result_block, buf);
78.                 result_addr++;
79.             }
80.
81.             // 分配新结果块
82.             result_block = getNewBlockInBuffer(buf);
83.             if (!result_block) {
84.                 printf("Buffer full while allocating result
block!\n");
85.                 return;
86.             }
87.             memset(result_block, 0, BLOCK_SIZE);
88.             result_count = 0;
89.         }
90.
91.         // 将满足条件的元组写入结果块
92.         memcpy(result_block + result_count * 8, ptr, 8);
93.         result_count++;
94.     }
95.
96.     ptr += 8; // 移动到下一个元组
97. }
98.
99. // 获取当前块的后继地址
100.    memcpy(&current_addr, block + BLOCK_SIZE - sizeof(int),
sizeof(int));
101.    freeBlockInBuffer(block, buf);
102. }
103.

```

设计思路

高效地从存储介质中提取数据，结合条件筛选和磁盘块管理。首先，它根据提供的地址逐步读取数据块，直到没有更多可读数据。对于每个读取的数据块，程序会逐一检查其中的记录，提取特定属性的值，并与输入的查询值进行比较。当找到匹配的记录时，程序会将相关信息打印到控制台，并暂时存储在结果区域。

为了管理存储结果的空间，程序会检查当前的结果区域是否已满。如果已满或尚未分配，

它会创建新的结果区域，并在必要时写入之前的结果。每个结果区域都有专门的地址指向，以确保可以顺利地访问后续的结果，同时还会进行相应的内存管理，以避免资源浪费。这样，整体逻辑清晰，能够高效地处理大量数据并提取所需的信息，非常适合对大数据集进行分析与操作的应用场景。

(3) 投影操作

要求：

实现关系投影算法：基于 ExtMem 程序库，使用高级语言实现关系投影 算法，对关系 R 上的 A 属性进行投影，并将结果存放在磁盘上。

核心代码

```
104. if (result_set[attr_value] == 0) { // 如果该值尚未被投影
105.     result_set[attr_value] = 1; // 标记为已投影
```

总体设计思路和关系选择操作类似，将属性的条件对比改成提取指定的属性。

(4) 循环嵌套连接

算法

```
1: for S 的每个元组 s do
2:   for R 的每个元组 r do
3:     if r 和 s 满足连接条件 then
4:       连接 r 和 s，并将结果写入输出缓冲区
```

```
106. while (R_addr != 0) {
107.     R_block = readBlockFromDisk(R_addr, buf);
108.     if (!R_block) {
109.         printf("Failed to read block %d of R\n", R_addr);
110.         return;
111.     }
112.
113.     // 对于每个 R 的块，遍历关系 S 的所有块
114.     S_addr = S->start_addr;
115.     while (S_addr != 0) {
116.         S_block = readBlockFromDisk(S_addr, buf);
117.         if (!S_block) {
118.             printf("Failed to read block %d of S\n", S_addr);
119.             return;
120.         }
121.
122.         // 遍历 R 块中的每个元组
123.         unsigned char R_ptr = R_block;
124.         for (int i = 0; i < MAX_TUPLES_PER_BLOCK; i++) {
125.             int R_A, R_B;
126.             memcpy(&R_A, R_ptr, sizeof(int)); // 获取 R.A
127.             memcpy(&R_B, R_ptr + 4, sizeof(int)); // 获取 R.B
128.             R_ptr += 8; // 移动到下一个元组
129.
130.             // 遍历 S 块中的每个元组
131.             unsigned char S_ptr = S_block;
132.             for (int j = 0; j < MAX_TUPLES_PER_BLOCK; j++) {
133.                 int S_C, S_D;
```

```

134.         memcpy(&S_C, S_ptr, sizeof(int));    // 获取 S.C
135.         memcpy(&S_D, S_ptr + 4, sizeof(int)); // 获取 S.D
136.         S_ptr += 8; // 移动到下一个元组
137.
138.         // 检查是否满足连接条件 R.A = S.C
139.         if (R_A == S_C) {
140.             // 连接成功, 输出到结果
141.             if (result_block == NULL || result_count ==
142. (BLOCK_SIZE - sizeof(int)) / 16) {
143.                 // 写入结果块
144.                 if (result_block != NULL) {
145.                     int next_addr = result_addr + 1;
146.                     memcpy(result_block + BLOCK_SIZE -
147. sizeof(int), &next_addr, sizeof(int));
148.                     writeBlockToDisk(result_block,
149. result_addr, buf);
150.                     freeBlockInBuffer(result_block,
151. buf);
152.                     result_addr++;
153.                 }
154.                 // 分配新结果块
155.                 result_block = getNewBlockInBuffer(buf);
156.                 if (!result_block) {
157.                     printf("Buffer full while allocating
158. result block!\n");
159.                     return;
160.                 }
161.                 memset(result_block, 0, BLOCK_SIZE);
162.                 result_count = 0;
163.             }
164.             // 将连接结果写入结果块
165.             memcpy(result_block + result_count * 16, &R_A,
166. sizeof(int)); // R.A
167.             memcpy(result_block + result_count * 16 + 4,
168. &R_B, sizeof(int)); // R.B
169.             memcpy(result_block + result_count * 16 + 8,
170. &S_C, sizeof(int)); // S.C
171.             memcpy(result_block + result_count * 16 + 12,
172. &S_D, sizeof(int)); // S.D
173.             result_count++;
174.             total_join_num++;
175.
176.             // 打印连接结果
177.             printf("| %-4d | %-4d | %-4d | %-4d |\n", R_A,
178. R_B, S_C, S_D);
179.         }
180.     }
181. }
182.
183. // 获取下一个 S 块

```

```

175.         memcpy(&S_addr, S_block + BLOCK_SIZE - sizeof(int),
        sizeof(int));
176.         freeBlockInBuffer(S_block, buf);
177.     }
178.     // 获取下一个 R 块
179.     memcpy(&R_addr, R_block + BLOCK_SIZE - sizeof(int),
        sizeof(int));
180.     freeBlockInBuffer(R_block, buf);
181. }
182.

```

设计思想：

高效地实现两个数据集之间的连接操作。核心在于从存储介质（磁盘）中逐块读取数据，依次检查两个数据集的记录。首先，程序从数据集 **R** 中读取块，并遍历 **R** 块中的所有记录。在此过程中，对于 **R** 中的每一条记录，程序会进行内部循环，读取数据集 **S** 的所有块，并检查 **S** 中的每条记录。

对于每对 **R** 和 **S** 中的记录，比较 **R** 的某个属性与 **S** 的对应属性。如果两个记录满足连接条件，则将它们组合成一条连接结果，并存储到结果区域中。当结果区域满时，将当前结果写入磁盘，并从缓冲区中分配新的结果区域以继续存储后续结果。同时，为了确保内存的有效使用，在读取每个块后都会释放之前使用的内存，以避免内存泄漏。

(5) 哈希连接

算法

- 1: // 哈希分桶
- 2: 将 **R** 的元组哈希到 $M-1$ 个桶 R_1, R_2, \dots, R_{M-1} 中(哈希键为 **R.Y**)
- 3: 将 **S** 的元组哈希到 $M-1$ 个桶 S_1, S_2, \dots, S_{M-1} 中(哈希键为 **S.Y**)
- 4: // 逐桶连接
- 5: **for** $i = 1, 2, \dots, M-1$ **do**
- 6: 使用一趟连接(one-pass join)算法计算 $R_i \bowtie S_i$ ，并将结果写入输出缓冲区

```

183. // 哈希表节点
184. typedef struct BucketNode {
185.     int key; // 存储哈希值，连接的左侧属性值
186.     unsigned char result_block;
187.     int result_count;
188.     unsigned int addr;
189.     unsigned int result_addr;
190. } BucketNode;
191.
192. // 哈希函数：简单的模哈希
193. unsigned int bucket(int key) {
194.     return key % PARTITION_COUNT;
195. }
196.
197. // 哈希分区
198. void partition(Buffer buf, Relation rel, int attr_index, BucketNode
        bucketTable[PARTITION_COUNT]) {
199.     unsigned char block;
200.     unsigned int current_addr = rel->start_addr;
201.

```



```

202.    // 将 R 分区
203.    printf("Partitioning relation ...\n");
204.    while (current_addr != 0) {
205.        // 读取磁盘块到缓冲区
206.        block = readBlockFromDisk(current_addr, buf);
207.        if (!block) {
208.            printf("Failed to read block %d\n", current_addr);
209.            return;
210.        }
211.
212.        unsigned char ptr = block;
213.
214.        // 遍历块内的元组
215.        for (int i = 0; i < (BLOCK_SIZE - sizeof(int)) / 8; i++) {
216.            int attr_value;
217.            int bucket_value;
218.            memcpy(&attr_value, ptr + rel->attr_offsets[attr_index],
219.                sizeof(int)); // 获取指定属性值
220.
221.            bucket_value = bucket(attr_value);
222.
223.            // 如果结果块为空或已满，分配新块
224.            if (bucketTable[bucket_value].result_block == NULL ||
225.                bucketTable[bucket_value].result_count >= MAX_TUPLES_PER_BLOCK) {
226.                if (bucketTable[bucket_value].result_block != NULL) {
227.                    // 写入结果块的后继地址
228.                    int next_addr = bucketTable[bucket_value].result_addr +
229.                        1;
230.                    memcpy(bucketTable[bucket_value].result_block + BLOCK_SIZE
231.                        - sizeof(int), &next_addr, sizeof(int));
232.                    writeBlockToDisk(bucketTable[bucket_value].result_block,
233.                        bucketTable[bucket_value].result_addr, buf);
234.                    freeBlockInBuffer(bucketTable[bucket_value].result_block,
235.                        buf);
236.                    bucketTable[bucket_value].result_addr++;
237.                }
238.
239.                // 分配新结果块
240.                bucketTable[bucket_value].result_block =
241.                    getNewBlockInBuffer(buf);
242.                if (!bucketTable[bucket_value].result_block) {
243.                    printf("Buffer full while allocating result block!\n");
244.                    return;
245.                }
246.                memset(bucketTable[bucket_value].result_block, 0,
247.                    BLOCK_SIZE);
248.                bucketTable[bucket_value].result_count = 0;
249.            }
250.
251.            // 将满足条件的元组写入结果块
252.            memcpy(bucketTable[bucket_value].result_block +
253.                bucketTable[bucket_value].result_count * 8, ptr, 8);

```

```

244.         bucketTable[bucket_value].result_count++;
245.
246.         ptr += 8; // 移动到下一个元组
247.     }
248.
249.     // 获取当前块的后继地址
250.     memcpy(&current_addr, block + BLOCK_SIZE - sizeof(int), sizeof(int));
251.     freeBlockInBuffer(block, buf);
252. }
253. // 写入最后一个结果块
254. for(int i = 0; i < PARTITION_COUNT; i++) {
255.     if (bucketTable[i].result_count > 0) {
256.         int end_addr = 0;
257.         memcpy(bucketTable[i].result_block + BLOCK_SIZE - sizeof(int),
258.             &end_addr, sizeof(int));
259.         writeBlockToDisk(bucketTable[i].result_block,
260.             bucketTable[i].result_addr, buf);
261.         freeBlockInBuffer(bucketTable[i].result_block, buf);
262.     }
263. }

```

设计思路:

上面的程序实现对数据集的哈希分区，以便于后续的处理和查询。首先，定义了一个结构体用于存储哈希表节点的信息，包括存储的哈希值、结果块、记录计数等。哈希函数采用简单的模运算，以将属性值映射到特定的分区。

程序的核心在于 `partition` 函数，负责读取数据集的块并将其分配到相应的哈希分区。首先从指定的起始地址读取数据块，然后遍历块中的每条记录。对于每条记录，提取指定属性的值，并计算出对应的哈希分区。若当前分区的结果块为空或已满，程序会分配新的结果块并处理之前的结果块，确保数据的有效存储。

在处理完所有块后，程序会将最后一个结果块写入磁盘，以确保所有数据都被正确保存。整个过程强调了内存管理和数据的高效分配，适用于需要对大规模数据进行分区处理的场景。

对每个桶内的元组使用循环嵌套连接。

哈希分区过程中的磁盘块内部情况:

```

shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 330.blk
18      916      3      28
13      737      8      430
23      531     23      124
28      136      0      331
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 331.blk
23      59      23      171
23      546      8      809
13      61      28      602
18      903      0      332
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 332.blk
38      493     13      757
18      772      8      857
18      354     13      830
28      857      0      333
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 333.blk
28      366     33      552
38      229     38      997
13      726      0      0
0      0      0      0

```

从 330 开始的连续 4 个块中 A 属性的值都是模 5 余 3 的。

(6) 归并连接

算法

```

1: // 创建归并段
2: 将 $R$ 划分为 $\lceil B(R)/M \rceil$ 个归并段(每个归并段按 $R.Y$ 进行排序)
3: 将 $S$ 划分为 $\lceil B(S)/M \rceil$ 个归并段(每个归并段按 $S.Y$ 进行排序)
4: // 归并
5: 读入 $R$ 和 $S$ 的每个归并段的第1页
6: repeat
7:   找出输入缓冲区中元组 $Y$ 属性的最小值 $y$ 
8:   for  $R$ 中满足 $R.Y = y$ 的元组 $r$  do
9:     for  $S$ 中满足 $S.Y = y$ 的元组 $s$  do
10:      连接 $r$ 和 $s$ , 并将结果写入输出缓冲区
11:   任意输入缓冲页中的元组若归并完毕, 则读入其归并段的下一页
12: until  $R$ 或 $S$ 的所有归并段都已归并完毕

```

步骤一: 生成归并段

```

263. int sortRelation(Buffer buf, Relation rel, unsigned int segs[]) {
264.     unsigned int current_addr = rel->start_addr, result_addr;
265.     unsigned char block;
266.     int seg_count = 0; // 当前段计数
267.     int block_count = 0; // 当前段的块计数
268.     unsigned char segment_buffer[MAX_SEGMENT] = {NULL}; // 段缓冲区
269.     unsigned char segment_ptr[MAX_SEGMENT] = {NULL}; // 指向段缓冲
区的指针
270.     unsigned char result_block = NULL; // 结果块
271.     int result_count = 0; // 结果元组计数
272.
273.     printf("Sorting relation %s...\n", rel->rel_name);
274.
275.     // 处理每个块
276.     while (current_addr != 0) {
277.         block = readBlockFromDisk(current_addr, buf);
278.         if (!block) {
279.             printf("Failed to read block %d\n", current_addr);
280.             return -1;
281.         }
282.
283.         // 排序当前块
284.         qsort(block, MAX_TUPLES_PER_BLOCK, rel->tuple_size,
compareTuples);
285.
286.         // 将块存入段缓冲区
287.         if (block_count < MAX_SEGMENT) { // 确保不越界
288.             segment_ptr[block_count] = block;
289.             segment_buffer[block_count] = block;
290.             block_count++;
291.         } else {

```

```
292.         printf("Block count exceeded MAX_SEGMENT\n");
293.         freeBlockInBuffer(block, buf); // 清理内存
294.         break; // 退出循环
295.     }
296.
297.     // 获取下一个块地址
298.     memcpy(&current_addr, block + BLOCK_SIZE - sizeof(int),
sizeof(int));
299.
300.     // 如果到达归并段的块个数或到达最后一个块
301.     if (block_count == MAX_SEGMENT || current_addr == 0) {
302.         // 初始化这个归并段的起始地址
303.         result_addr = segs[seg_count];
304.         while(1) {
305.             // 找到几个段中最小的元组
306.             int min_seg = -1;
307.             int min_value = INT_MAX;
308.
309.             for (int i = 0; i < block_count; i++) {
310.                 if (segment_ptr[i]) {
311.                     int value;
312.                     memcpy(&value, segment_ptr[i],
sizeof(int));
313.                     if (value < min_value && value != 0) {
314.                         min_value = value;
315.                         min_seg = i;
316.                     }
317.                 }
318.             }
319.
320.             // 如果没有找到有效值，归并结束
321.             if (min_seg == -1) break;
322.
323.             // 如果缓冲区满则写入缓存并添加新的缓冲区
324.             if (result_block == NULL || result_count >= (BLOCK_SIZE
- sizeof(int)) / 8) {
325.                 if (result_block != NULL) {
326.                     int next_addr = result_addr + 1;
327.                     memcpy(result_block + BLOCK_SIZE -
sizeof(int), &next_addr, sizeof(int));
328.                     writeBlockToDisk(result_block, result_addr,
buf);
329.                     freeBlockInBuffer(result_block, buf);
330.                     result_addr++;
331.                 }
332.
333.                 result_block = getNewBlockInBuffer(buf);
334.                 if (!result_block) {
335.                     printf("Buffer full while allocating result
block!\n");
336.                     return -1; // 修正返回值
```

```

337.         }
338.
339.         memset(result_block, 0, BLOCK_SIZE);
340.         result_count = 0;
341.     }
342.
343.     // 把最小的元组写入缓冲区
344.     memcpy(result_block + result_count * rel->tuple_size,
segment_ptr[min_seg], rel->tuple_size);
345.     result_count++;
346.     segment_ptr[min_seg] += rel->tuple_size; // 假设每个
元组大小是 rel->tuple_size
347.     if (segment_ptr[min_seg] >= segment_buffer[min_seg] +
BLOCK_SIZE - sizeof(int)) {
348.         segment_ptr[min_seg] = NULL;
349.     }
350. }
351.
352. // 写最后一个结果块
353. if (result_count > 0) {
354.     int end_addr = 0;
355.     memcpy(result_block + BLOCK_SIZE - sizeof(int),
&end_addr, sizeof(int));
356.     writeBlockToDisk(result_block, result_addr, buf);
357.     freeBlockInBuffer(result_block, buf);
358.     result_block = NULL; // 清空指针方便后续重用与检测
359. }
360.
361. // 清理段缓冲区
362. for (int k = 0; k < block_count; k++) {
363.     freeBlockInBuffer(segment_buffer[k], buf);
364.     segment_buffer[k] = NULL;
365.     segment_ptr[k] = NULL;
366. }
367.
368. printf("Segment %d Merging completed.\n", seg_count);
369. seg_count++; // 增加段计数
370. block_count = 0; // 重置块计数以处理下一个段
371.     }
372. }
373.
374. return seg_count; // 返回段计数
375. }
376.

```

对关系数据进行排序，并将排序结果存储到指定的段中：

1. 初始化：程序开始时定义了一些变量，包括当前地址、结果块、段缓冲区等。每个段可以存储多个块，以便于后续的归并排序。
2. 读取和排序：通过循环读取数据块，使用 `qsort` 对每个块内的元组进行排序。排序完成后，程序将排序后的块存入段缓冲区，确保不超过最大段数。
3. 归并过程：当达到最大段数或读取到最后一个块时，程序进入归并阶段。它会查找当前

段中最小的元组，并将其写入结果块中。如果结果块已满，则会将其写入磁盘并分配新的结果块。

4. 更新指针：在写入最小元组后，程序更新指向该段的指针，以便继续处理下一个元组。如果当前段的指针超出块的范围，则将其置为 NULL。

5. 清理：完成归并后，程序会释放段缓冲区的内存，以避免内存泄漏，并更新段计数以准备处理下一个段。

6. 返回结果：最后，程序返回处理过的段的数量，表明排序和归并操作的完成。

创建归并段后的磁盘块内部情况：

```

shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 600.blk
1      730      2      676
3      28       4      927
4      751      4      652
5      328      0      601
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 601.blk
5      896      5      620
7      730      7      442
8      430      8      809
8      857      0      602
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 602.blk
10     422      10     803
11     60       12     369
12     43       12     981
13     737      0      603
shuhang@shuhang-virtual-machine:~/database$ od -t dI -An 603.blk
13     61       13     757
14     527      15     368
15     540      17     506
18     916      0      604

```

可以看出 A 属性的值是在磁盘块中存储是不断递增的。

步骤二：归并排序

```

377. while (1) {
378.     int min_R = INT_MAX, min_S = INT_MAX;
379.
380.     // 找到 R 和 S 的当前最小值
381.     for (int i = 0; i < R_count; i++) {
382.         if (R_blocks[i] && R_indices[i] < MAX_TUPLES_PER_BLOCK) {
383.             if (R_values[i][0] < min_R) {
384.                 min_R = R_values[i][0];
385.             }
386.         }
387.     }
388.     for (int i = 0; i < S_count; i++) {
389.         if (S_blocks[i] && S_indices[i] < MAX_TUPLES_PER_BLOCK) {
390.             if (S_values[i][0] < min_S) {
391.                 min_S = S_values[i][0];
392.             }
393.         }
394.     }
395.
396.     // 如果都没有可用元组，结束
397.     if (min_R == INT_MAX || min_S == INT_MAX) break;
398.

```

```

399.         // 如果 R 和 S 的最小值相等，处理匹配
400.         if (min_R == min_S) {
401.             S_buffer_size = 0;
402.
403.             // 缓存所有等于 min_S 的 S 元组
404.             for (int i = 0; i < S_count; i++) {
405.                 while (S_blocks[i] && S_indices[i] <
MAX_TUPLES_PER_BLOCK && S_values[i][0] == min_S) {
406.                     S_buffer[S_buffer_size][0] = S_values[i][0];
407.                     S_buffer[S_buffer_size][1] = S_values[i][1];
408.                     S_buffer_size++;
409.
410.                     // 移动 S 的指针
411.                     S_indices[i]++;
412.                     if (S_indices[i] == MAX_TUPLES_PER_BLOCK) {
413.                         unsigned int next_addr;
414.                         memcpy(&next_addr, S_blocks[i] + BLOCK_SIZE -
sizeof(int), sizeof(int));
415.                         freeBlockInBuffer(S_blocks[i], buf);
416.                         if (next_addr != 0) {
417.                             S_blocks[i] =
readBlockFromDisk(next_addr, buf);
418.                             S_indices[i] = 0;
419.                             getTuple(S_blocks[i], 0, &S_values[i][0],
&S_values[i][1]);
420.                         } else {
421.                             S_blocks[i] = NULL;
422.                         }
423.                     } else {
424.                         getTuple(S_blocks[i], S_indices[i],
&S_values[i][0], &S_values[i][1]);
425.                     }
426.                 }
427.             }
428.
429.             // 遍历所有等于 min_R 的 R 元组，与 S_buffer 中的元组连接
430.             for (int i = 0; i < R_count; i++) {
431.                 while (R_blocks[i] && R_indices[i] <
MAX_TUPLES_PER_BLOCK && R_values[i][0] == min_R) {
432.                     for (int j = 0; j < S_buffer_size; j++) {
433.                         // 输出匹配的结果
434.                         if (result_count == (BLOCK_SIZE - sizeof(int))
/ (2 * TUPLE_SIZE)) {
435.                             int next_addr = result_addr + 1;
436.                             memcpy(result_block + BLOCK_SIZE -
sizeof(int), &next_addr, sizeof(int));
437.                             writeBlockToDisk(result_block,
result_addr, buf);
438.                             freeBlockInBuffer(result_block, buf);
439.                             result_block = getNewBlockInBuffer(buf);
440.                             result_count = 0;

```

```

441.         }
442.
443.         memcpy(result_block + result_count 16,
&R_values[i][0], sizeof(int));
444.         memcpy(result_block + result_count 16 + 4,
&R_values[i][1], sizeof(int));
445.         memcpy(result_block + result_count 16 + 8,
&S_buffer[j][0], sizeof(int));
446.         memcpy(result_block + result_count 16 + 12,
&S_buffer[j][1], sizeof(int));
447.         // 打印连接结果
448.         printf("| %-4d | %-4d | %-4d | %-4d |\n",
R_values[i][0], R_values[i][1], S_buffer[j][0], S_buffer[j][1]);
449.         result_count++;
450.         total_join_num++;
451.     }
452.
453.     // 移动 R 的指针
454.     R_indices[i]++;
455.     if (R_indices[i] == MAX_TUPLES_PER_BLOCK) {
456.         unsigned int next_addr;
457.         memcpy(&next_addr, R_blocks[i] + BLOCK_SIZE -
sizeof(int), sizeof(int));
458.         freeBlockInBuffer(R_blocks[i], buf);
459.         if (next_addr != 0) {
460.             R_blocks[i] =
readBlockFromDisk(next_addr, buf);
461.             R_indices[i] = 0;
462.             getTuple(R_blocks[i], 0, &R_values[i][0],
&R_values[i][1]);
463.         } else {
464.             R_blocks[i] = NULL;
465.         }
466.     } else {
467.         getTuple(R_blocks[i], R_indices[i],
&R_values[i][0], &R_values[i][1]);
468.     }
469. }
470. }
471. } else {
472.     // 如果 min_R 和 min_S 不相等, 移动较小值的指针
473.     if (min_R < min_S) {
474.         for (int i = 0; i < R_count; i++) {
475.             if (R_blocks[i] && R_indices[i] <
MAX_TUPLES_PER_BLOCK && R_values[i][0] == min_R) {
476.                 R_indices[i]++;
477.                 if (R_indices[i] == MAX_TUPLES_PER_BLOCK) {
478.                     unsigned int next_addr;
479.                     memcpy(&next_addr, R_blocks[i] +
BLOCK_SIZE - sizeof(int), sizeof(int));
480.                     freeBlockInBuffer(R_blocks[i], buf);

```



```

481.             if (next_addr != 0) {
482.                 R_blocks[i] =
readBlockFromDisk(next_addr, buf);
483.                 R_indices[i] = 0;
484.                 getTuple(R_blocks[i], 0,
&R_values[i][0], &R_values[i][1]);
485.             } else {
486.                 R_blocks[i] = NULL;
487.             }
488.         } else {
489.             getTuple(R_blocks[i], R_indices[i],
&R_values[i][0], &R_values[i][1]);
490.         }
491.     }
492. }
493. } else {
494.     for (int i = 0; i < S_count; i++) {
495.         if (S_blocks[i] && S_indices[i] <
MAX_TUPLES_PER_BLOCK && S_values[i][0] == min_S) {
496.             S_indices[i]++;
497.             if (S_indices[i] == MAX_TUPLES_PER_BLOCK) {
498.                 unsigned int next_addr;
499.                 memcpy(&next_addr, S_blocks[i] +
BLOCK_SIZE - sizeof(int), sizeof(int));
500.                 freeBlockInBuffer(S_blocks[i], buf);
501.                 if (next_addr != 0) {
502.                     S_blocks[i] =
readBlockFromDisk(next_addr, buf);
503.                     S_indices[i] = 0;
504.                     getTuple(S_blocks[i], 0,
&S_values[i][0], &S_values[i][1]);
505.                 } else {
506.                     S_blocks[i] = NULL;
507.                 }
508.             } else {
509.                 getTuple(S_blocks[i], S_indices[i],
&S_values[i][0], &S_values[i][1]);
510.             }
511.         }
512.     }
513. }
514. }
515. }
516.
517. // 写入最后的结果块
518. if (result_count > 0) {
519.     int end_addr = 0;
520.     memcpy(result_block + BLOCK_SIZE - sizeof(int), &end_addr,
sizeof(int));
521.     writeBlockToDisk(result_block, result_addr, buf);
522.     freeBlockInBuffer(result_block, buf);

```

```
523.     }
524.
```

对归并段进行多路归并连接：

1. 初始化：程序定义了一些变量，用于存储 R 和 S 的当前最小值、缓冲区和索引状态等。
2. 查找最小值：在两个关系 R 和 S 中分别寻找当前有效元组的最小值。这个过程使用了两个循环来遍历每个关系的块，确保只考虑当前处于有效范围内的元组。
3. 判断连接条件：
 - 相等情况：如果 min_R 和 min_S 相等，程序将缓存所有等于 min_S 的 S 元组，并找到的 R 元组与这些 S 元组进行连接。
 - 当找到一个新的元组时，检查结果块是否已满。如果已满，则将当前结果块写入磁盘并创建一个新的结果块。
 - 然后，将 R 和 S 的所有相关元组复制到结果块，并打印连接的结果。
 - 不相等情况：如果 min_R 和 min_S 不相等，程序会相应地移动较小值的指针。具体来说：
 - 如果 min_R 小，则移动 R 的指针并获取下一个元组。
 - 如果 min_S 小，则移动 S 的指针并获取下一个元组。
 - 如果该块元组都访问过了，根据块的最后四个 bit 查看下一个磁盘块的地址，并加载。
4. 写入最后的结果块：在所有的连接操作完成后，如果还有未写入的结果，程序会将这些结果写入一个最终的结果块并释放相关资源。

3.1.2 查询优化算法设计

任选下列查询语句中的三条，将其转化为对应的查询执行树，并且根据设计的查询优化算法，对生成的查询执行树进行优化。

• 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))

• PROJECTION [ENAME] (SELECT [SALARY < 3000] (EMPLOYEE JOIN SELECT [PNO = 'P1'] (WORKS_ON JOIN PROJECT)))

• PROJECTION [DNAME, SALARY] (AVG [SALARY] (SELECT [DNAME = 'Research'] (EMPLOYEE JOIN DEPARTMENT)))

本实验选择下面三句：

```
525. queries = [
526.     "SELECT [ ENAME = 'Mary' & DNAME = 'Research' ] ( EMPLOYEE JOIN
      DEPARTMENT )",
527.     "PROJECTION [ BDATE ] ( SELECT [ ENAME = 'John' & DNAME =
      'Research' ] ( EMPLOYEE JOIN DEPARTMENT ) )",
528.     "SELECT [ ESSN = '01' ] ( PROJECTION [ ESSN, PNAME ] ( WORKS_ON
      JOIN PROJECT ) )"
529. ]
```

具体执行步骤如下：

(1) 初始查询执行树形成

每条查询语句首先被解析为初始的查询执行树。在生成的初始执行树中，SELECT、PROJECTION 和 JOIN 等操作按照查询语句的书写顺序排列，但并未进行优化。例如，对于查询 3，其初始执行树是 SELECT 操作位于最外层，直接作用于 PROJECTION，而 PROJECTION 再作用于 JOIN 操作。这里需要考虑查询执行树生成的结构，一开始考虑使用文本形式，但这样树状的形势不太好展示，因此通过了解使用了 graphviz 图形库进行可视化展示，这样效果更好。具体部分代码如下：

```

530. from graphviz import Digraph type: ignore
531.
532. class QueryNode:
533.     def __init__(self, operator, children=None, attributes=None,
534.                  condition=None):
535.         self.operator = operator
536.         self.children = children if children else []
537.         self.attributes = attributes
538.         self.condition = condition
539.         self.id = id(self)    唯一标识符
540.
541.     def __repr__(self):
542.         desc = self.operator
543.         if self.attributes:
544.             desc += f" [{', '.join(self.attributes)}]"
545.         if self.condition:
546.             desc += f" [Condition: {self.condition}]"
547.         return desc
548.
549.     def to_graphviz(self, graph):
550.         label = self.__repr__()
551.         graph.node(str(self.id), label)
552.         for child in self.children:
553.             graph.edge(str(self.id), str(child.id))
554.             child.to_graphviz(graph)
555.
556.     def visualize(self, filename='query_tree'):
557.         graph = Digraph(comment='Query Execution Tree')
558.         self.to_graphviz(graph)
559.         graph.render(filename, view=True)

```

其次需要考虑如何进行分析生成查询执行树，这里使用的是 python 的 PLY 库构建了语法分析器，定义了基本的词法和语法规则，能够将查询语句划分为词，并且通过语法分析识别 SELECT、PROJECTION 等操作，具体设计代码如下：

```

559. parser.py
560.
561. import ply.lex as lex
562. import ply.yacc as yacc
563. from query_tree import QueryNode
564.
565. 定义保留字
566. reserved = {
567.     'SELECT': 'SELECT',
568.     'PROJECTION': 'PROJECTION',

```

```
569.     'JOIN': 'JOIN',
570.     'AVG': 'AVG',
571. }
572.
573. 词法分析器
574. tokens = [
575.     'LPAREN',
576.     'RPAREN',
577.     'LBRACKET',
578.     'RBRACKET',
579.     'COMMA',
580.     'AND',
581.     'EQUALS',
582.     'LT',
583.     'IDENT',
584.     'STRING',
585.     'NUMBER'
586. ] + list(reserved.values())
587.
588. 定义简单的符号
589. t_LPAREN = r'\('
590. t_RPAREN = r'\)'
591. t_LBRACKET = r'\['
592. t_RBRACKET = r'\]'
593. t_COMMA = r','
594. t_AND = r'&'
595. t_EQUALS = r'='
596. t_LT = r'<'
597.
598. t_ignore = ' \t'
599.
600. 定义字符串
601. def t_STRING(t):
602.     r'\'[^\']*\'
603.     t.value = t.value.strip('\')
604.     return t
605.
606. 定义数字
607. def t_NUMBER(t):
608.     r'\d+'
609.     t.value = t.value
610.     return t
611.
612. 定义标识符和保留字
613. def t_IDENT(t):
614.     r'[A-Za-z_][A-Za-z0-9_]*'
615.     t.type = reserved.get(t.value.upper(), 'IDENT')    保留字优先
616.     return t
617.
618. 跳过换行符
619. def t_newline(t):
```

```
620.     r'\n+'
621.     pass
622.
623.     处理非法字符
624.     def t_error(t):
625.         print(f"Illegal character '{t.value[0]}'")
626.         t.lexer.skip(1)
627.
628.     lexer = lex.lex()
629.
630.     定义运算符优先级
631.     precedence = (
632.         ('left', 'AND'),
633.         ('left', 'JOIN'),
634.         ('left', 'SELECT', 'PROJECTION', 'AVG'),
635.     )
636.
637.     语法分析器
638.
639.     def p_query(p):
640.         '''query : operation'''
641.         p[0] = p[1]
642.
643.     def p_operation_select(p):
644.         '''operation : SELECT LBRACKET condition RBRACKET LPAREN operation
        RPAREN'''
645.         node = QueryNode('SELECT', children=[p[6]], condition=p[3])
646.         p[0] = node
647.
648.     def p_operation_projection(p):
649.         '''operation : PROJECTION LBRACKET attributes RBRACKET LPAREN
        operation RPAREN'''
650.         node = QueryNode('PROJECTION', children=[p[6]], attributes=p[3])
651.         p[0] = node
652.
653.     def p_operation_avg(p):
654.         '''operation : AVG LBRACKET IDENT RBRACKET LPAREN operation
        RPAREN'''
655.         node = QueryNode('AVG', children=[p[6]], attributes=[p[3]])
656.         p[0] = node
657.
658.     def p_operation_join(p):
659.         '''operation : operation JOIN operation'''
660.         left = p[1]
661.         right = p[3]
662.         node = QueryNode('JOIN', children=[left, right])
663.         p[0] = node
664.
665.     def p_operation_table(p):
666.         '''operation : IDENT'''
667.         node = QueryNode('TABLE', attributes=[p[1]])
```

```

668.     p[0] = node
669.
670. def p_condition_and(p):
671.     '''condition : condition AND condition'''
672.     p[0] = f"{p[1]} & {p[3]}"
673.
674. def p_condition_equals(p):
675.     '''condition : IDENT EQUALS value'''
676.     p[0] = f"{p[1]} = {p[3]}"
677.
678. def p_condition_lt(p):
679.     '''condition : IDENT LT value'''
680.     p[0] = f"{p[1]} < {p[3]}"
681.
682. def p_value(p):
683.     '''value : STRING
684.             | NUMBER'''
685.     p[0] = p[1]
686.
687. def p_attributes_multiple(p):
688.     '''attributes : attributes COMMA IDENT'''
689.     p[0] = p[1] + [p[3]]
690.
691. def p_attributes_single(p):
692.     '''attributes : IDENT'''
693.     p[0] = [p[1]]
694.
695. def p_error(p):
696.     if p:
697.         print(f"Syntax error at '{p.value}'")
698.     else:
699.         print("Syntax error at EOF")
700.
701. parser = yacc.yacc()

```

1. 词法分析

在处理查询过程中，首先进行的是词法分析，这一部分负责将输入字符串拆分成一个个记号（tokens）。这项工作是通过 PLY（Python Lex-Yacc）库实现的。

- 识别的记号：在查询中的每一个部分都会被识别为一个特定的记号。例如，对于 PROJECTION [DNAME, SALARY] (AVG [SALARY] (SELECT [DNAME = 'Research'] (EMPLOYEE JOIN DEPARTMENT)))，会产生记号如 PROJECTION、LBRACKET、IDENT、COMMA 等。

- 输入的示例记号：

- PROJECTION：被识别为 PROJECTION。
- [DNAME, SALARY]：被拆分为 LBRACKET、IDENT、COMMA、IDENT、RBRACKET 等。

2. 语法分析

完成记号化之后，语法分析器会继续对记号序列进行验证，确保它符合定义的文法规则，并构建出一个解析树。

- 定义的操作：
 - SELECT 操作: SELECT [condition] (operation)
 - PROJECTION 操作: PROJECTION [attributes] (operation)
 - AVG 操作: AVG [IDENT] (operation)
 - JOIN 操作: operation JOIN operation
- 解析树构建: 解析器在处理记号时, 会利用 QueryNode 创建树结构, 每个操作会形成树中的一个节点。

3. 查询步骤分析

处理查询 PROJECTION [DNAME, SALARY] (AVG [SALARY] (SELECT [DNAME = 'Research'] (EMPLOYEE JOIN DEPARTMENT))) 的过程如下:

i. 顶层操作:

- 首先处理 PROJECTION [DNAME, SALARY] (...)。
- 创建一个 QueryNode 类型的节点, 表示 PROJECTION, 并将属性设置为 DNAME 和 SALARY。

ii. 内层操作:

- 然后处理 AVG [SALARY] (SELECT [DNAME = 'Research'] (EMPLOYEE JOIN DEPARTMENT))。
- 这又创建一个 QueryNode 类型的节点, 表示 AVG, 属性为 SALARY。

iii. 处理 SELECT 操作:

- 处理 SELECT [DNAME = 'Research'] (EMPLOYEE JOIN DEPARTMENT)。
- 在这个操作中, 会建立一个 condition 节点(即 DNAME = 'Research')和一个 JOIN 操作节点。

V. JOIN 操作:

- 对 EMPLOYEE JOIN DEPARTMENT 进行处理。
- 这里已经形成了另一个 QueryNode, 表示两个表的连接。

最终解析树结构

最终生成的解析树将体现查询的结构, 每个操作会作为树中的一个节点:

- 根节点是 PROJECTION。
 - 它的子节点为 AVG 操作。
 - AVG 节点又有 SELECT 操作作为子节点。
 - SELECT 节点有两个子节点: 条件和 JOIN 操作节点。
 - JOIN 节点则有 EMPLOYEE 和 DEPARTMENT 表作为子节点。

代码运行时:

1. 词法分析器将输入拆分成记号, 并将其传递给语法分析器。
2. 语法分析器遵循文法规则构建出树形结构。
3. 解析树中的每个操作可以根据查询的语义进行执行, 最终返回结果集。

(2) 优化规则的应用

针对生成的初始查询执行树，我们结合课上所学以及这里特异的几条查询语句，从各种优化规则进行选择，实现了基于选择下推和条件分解的优化规则。对于简单条件（如 `ENAME = 'Mary'`），我们将其直接下推到与之相关的表子树中。对于复合条件（如 `ENAME = 'Mary' & DNAME = 'Research'`），我们进一步分解条件并分别下推到相关表的子树上。

在优化查询 3 时，扩展了优化逻辑，使 `SELECT` 操作能够穿透 `PROJECTION`，直接下推到 `JOIN` 的左右子节点，确保选择操作尽量靠近数据源，从而减少中间结果的规模。

```

702. # optimizer.py
703.
704. from query_tree import QueryNode
705.
706. def split_conditions(condition):
707.     """将复杂条件字符串按 '&' 拆分为单个条件的列表。"""
708.     return condition.split('&')
709.
710. def optimize(node):
711.     if node is None:
712.         return None
713.
714.     # 优化子节点
715.     optimized_children = [optimize(child) for child in node.children]
716.     node.children = optimized_children
717.
718.     # 实现选择下推并分解复杂选择条件
719.     if node.operator == 'SELECT' and len(node.children) == 1:
720.         child = node.children[0]
721.
722.         # 处理 SELECT -> PROJECTION 的情况
723.         if child.operator == 'PROJECTION':
724.             grandchild = child.children[0]
725.             if grandchild.operator == 'JOIN':
726.                 # 下推 SELECT，尝试作用于 JOIN 的左右子节点
727.                 select_condition = node.condition
728.                 left_child, right_child = grandchild.children
729.
730.                 # 分配选择条件到 JOIN 的左、右子节点
731.                 left_conditions = [cond for cond in
split_conditions(select_condition) if "ESSN" in cond]
732.                 right_conditions = [cond for cond in
split_conditions(select_condition) if "PNAME" in cond]
733.
734.                 if left_conditions:
735.                     left_select_condition = " &
".join(left_conditions)
736.                     left_child = QueryNode('SELECT',
children=[left_child], condition=left_select_condition)
737.                     if right_conditions:
738.                         right_select_condition = " &
".join(right_conditions)

```



```

739.             right_child = QueryNode('SELECT',
children=[right_child], condition=right_select_condition)
740.
741.             # 更新 JOIN 节点
742.             optimized_join = QueryNode('JOIN',
children=[left_child, right_child])
743.             # 保留 PROJECTION 操作
744.             node = QueryNode('PROJECTION',
children=[optimized_join], attributes=child.attributes)
745.         else:
746.             # 如果 PROJECTION 的子节点不是 JOIN, 无法下推, 保留原结
构
747.             node.children = [optimize(child)]
748.
749.             # 处理 SELECT -> JOIN 的情况
750.             elif child.operator == 'JOIN':
751.                 conditions = split_conditions(node.condition)
752.                 left_child, right_child = child.children
753.
754.                 # 分配选择条件到 JOIN 的左、右子节点
755.                 left_conditions = [cond for cond in conditions if "ESSN"
in cond or "ENAME" in cond]
756.                 right_conditions = [cond for cond in conditions if "PNAME"
in cond or "DNAME" in cond]
757.
758.                 if left_conditions:
759.                     left_select_condition = " & ".join(left_conditions)
760.                     left_child = QueryNode('SELECT',
children=[left_child], condition=left_select_condition)
761.                 if right_conditions:
762.                     right_select_condition = " & ".join(right_conditions)
763.                     right_child = QueryNode('SELECT',
children=[right_child], condition=right_select_condition)
764.
765.                 # 更新 JOIN 节点的子节点为优化后的选择子节点
766.                 node = QueryNode('JOIN', children=[left_child,
right_child])
767.
768.             return node

```

(3) 优化后查询执行树的生成

通过优化后, 每条查询的执行树结构得到了简化。优化后的执行树体现了选择下推的规则, 并在逻辑上更贴近实际的数据库执行策略。

接下来我们来结合具体的运行结果进行讲解, 运行 `main.py` 之后可得到如下, 分别是终端的运行结果和图形库自动生成的 pdf 文件, 分别是查询 1、2、3 未优化和优化后的结果。

3.2 实验结果

(1) 关系查询操作

```

shuhang@shuhang-virtual-machine:~/database$ ./query_relation
Select relation (R/S), attribute (e.g., A/B/C/D), and value (e.g., 10), or 'exit': R A 10 200
Query results for A = 10:
+-----+
| A      | B      |
+-----+
| 10     | 422    |
| 10     | 803    |
| 10     | 918    |
+-----+
Query results written to disk starting from block 200
Select relation (R/S), attribute (e.g., A/B/C/D), and value (e.g., 10), or 'exit': S C 30 200
Query results for C = 30:
+-----+
| C      | D      |
+-----+
| 30     | 931    |
| 30     | 895    |
| 30     | 430    |
| 30     | 628    |
| 30     | 951    |
+-----+
Query results written to disk starting from block 200
Select relation (R/S), attribute (e.g., A/B/C/D), and value (e.g., 10), or 'exit':

```

操作过程的消耗情况:

Buffer Statistics:

Parameter	Value
Number of I/O	50
Buffer size (bytes)	520
Block size (bytes)	64
Total blocks	8
Free blocks	10

(2) 关系投影操作

```

shuhang@shuhang-virtual-machine:~/database$ ./project_relation
Select relation (R/S), attribute (e.g., A/B/C/D) or 'exit': R A 200
Projecting attribute A on relation:
+-----+
| A      |
+-----+
| 24     |
| 18     |
| 34     |
| 27     |
| 10     |
| 3      |
| 11     |
| 4      |
| 21     |
| 13     |
| 12     |
| 8      |
| 23     |
| 28     |
| 30     |
| 22     |
| 25     |
| 39     |
| 36     |
| 14     |
| 37     |
| 26     |
| 5      |
| 17     |
| 7      |
| 15     |
| 35     |
| 29     |
| 20     |
| 38     |
| 1      |
| 32     |
| 2      |
| 9      |
| 31     |
| 19     |
| 33     |
+-----+
Projection results written to disk starting from block 200

```

操作过程的消耗情况:

Buffer Statistics:

Parameter	Value
Number of I/O	19
Buffer size (bytes)	520
Block size (bytes)	64
Total blocks	8
Free blocks	11

(3) 循环嵌套连接

Performing Nested-Loop Join (R.A = S.C)				23	59	23	72	25	538	25	419
				28	136	28	982	39	325	39	10
				30	168	30	931	39	325	39	43
				30	168	30	895	25	538	25	156
				30	374	30	931	36	371	36	582
				30	374	30	895	37	874	37	5
				28	136	28	108	25	538	25	885
				28	136	28	256	39	325	39	909
				28	136	28	812	25	538	25	334
				34	457	34	600	22	920	22	498
				34	457	34	905	22	920	22	189
				23	59	23	777	37	874	37	730
				23	59	23	256	37	874	37	461
				23	59	23	28	25	538	25	233
				28	136	28	903	39	325	39	118
				23	59	23	698	37	874	37	570
				30	168	30	430	25	538	25	762
				30	374	30	430	37	874	37	368
				30	168	30	628	22	920	22	553
				30	374	30	628	22	920	22	594
				23	59	23	484	25	538	25	81
				30	168	30	951	39	325	39	291
				34	457	34	935	36	371	36	970
				30	374	30	951	39	325	39	315
				22	920	22	423	39	325	39	730
				36	371	36	811	37	874	37	389
				39	325	39	627	39	325	39	326
				36	371	36	132	36	371	36	912
				25	538	25	774	34	858	34	805
				36	371	36	543	23	171	23	72
				22	920	22	211	34	858	34	600

操作过程的消耗情况:

Total Join tuple numbles: 287

Buffer Statistics:

Parameter	Value
Number of I/O	624
Buffer size (bytes)	520
Block size (bytes)	64
Total blocks	8
Free blocks	104

(4) 哈希连接

Joining partition 0...				Joining partition 1...				Joining partition 2...			
30	168	30	931	36	371	36	811	27	493	27	755
30	168	30	895	36	371	36	132	27	493	27	747
30	168	30	430	36	371	36	543	27	493	27	34
30	168	30	628	36	371	36	582	27	493	27	140
30	168	30	951	36	371	36	970	22	920	22	423
30	374	30	931	36	371	36	912	22	920	22	211
30	374	30	895	26	926	26	499	22	920	22	498
30	374	30	430	26	926	26	359	22	920	22	189
30	374	30	628	36	571	36	811	22	920	22	553
30	374	30	951	36	571	36	132	22	920	22	594
25	538	25	774	36	571	36	543	37	874	37	5
25	538	25	419	36	571	36	582	37	874	37	730
25	538	25	156	36	571	36	970	37	874	37	461
25	538	25	885	36	571	36	912	37	874	37	570
25	538	25	334	26	690	26	499	37	874	37	368
25	538	25	233	26	690	26	359	37	874	37	389
25	538	25	762	31	369	31	625	37	282	37	5
25	538	25	81	31	369	31	997	37	282	37	730
20	13	20	787	31	369	31	659	37	282	37	461
20	13	20	700	31	369	31	796	37	282	37	570

操作过程的消耗情况:

Total Join tuple numbles: 287

Buffer Statistics:

Parameter	Value
Number of I/O	1096
Buffer size (bytes)	520
Block size (bytes)	64
Total blocks	8
Free blocks	156

(5) 归并连接

R.A	R.B	S.C	S.D	23	531	23	484	24	20	24	341
				23	124	23	72	24	20	24	206
				23	124	23	777	24	20	24	260
20	13	20	787	23	124	23	256	24	20	24	310
20	13	20	700	23	124	23	28	24	20	24	359
20	13	20	116	23	124	23	698	24	492	24	37
20	625	20	787	23	124	23	484	24	492	24	341
20	625	20	700	23	59	23	72	24	492	24	206
20	625	20	116	23	59	23	777	24	492	24	260
20	489	20	787	23	59	23	256	24	492	24	310
20	489	20	700	23	59	23	28	24	492	24	359
20	489	20	116	23	59	23	698	24	12	24	37
20	937	20	787	23	59	23	484	24	12	24	341
20	937	20	700	23	171	23	72	24	12	24	206
20	937	20	116	23	171	23	777	24	12	24	260
22	920	22	423	23	171	23	256	24	12	24	310
22	920	22	211	23	171	23	28	24	12	24	359
22	920	22	498	23	171	23	698	25	538	25	774
22	920	22	189	23	171	23	484	25	538	25	419
22	920	22	553	23	546	23	72	25	538	25	156
22	920	22	594	23	546	23	777	25	538	25	885
22	281	22	423	23	546	23	256	25	538	25	334
22	281	22	211	23	546	23	28	25	538	25	233
22	281	22	498	23	546	23	698	25	538	25	762
22	281	22	189	23	546	23	484	25	538	25	81
22	281	22	553	24	887	24	37	25	819	25	774
22	281	22	594	24	887	24	341	25	819	25	419
23	531	23	72	24	887	24	206	25	819	25	156
23	531	23	777	24	887	24	260	25	819	25	885
23	531	23	256	24	887	24	310	25	819	25	334
23	531	23	28	24	887	24	359	25	819	25	233
23	531	23	698	24	20	24	37	25	819	25	762

操作过程的消耗情况:

Total Join tuple numbles: 287

Buffer Statistics:

Parameter	Value
Number of I/O	225
Buffer size (bytes)	780
Block size (bytes)	64
Total blocks	12
Free blocks	152

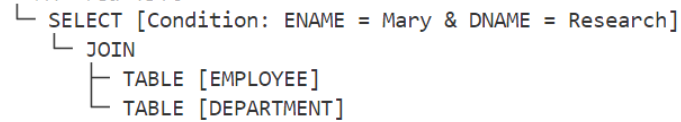
(5) 优化后的查询执行树

--- 查询 1 ---

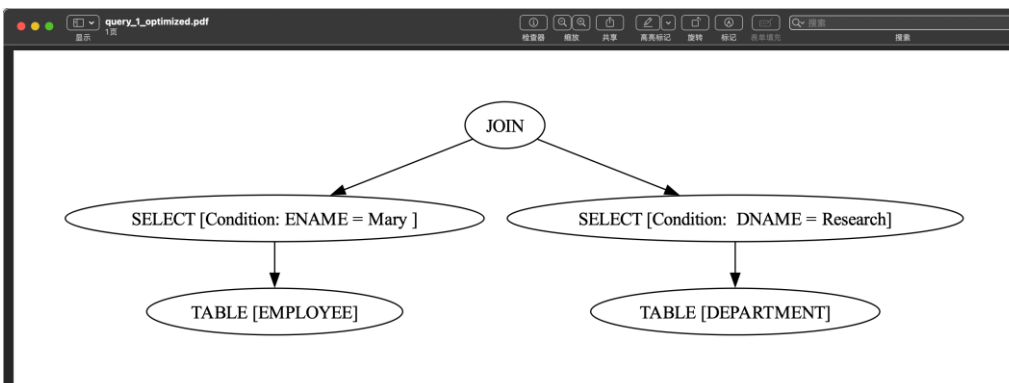
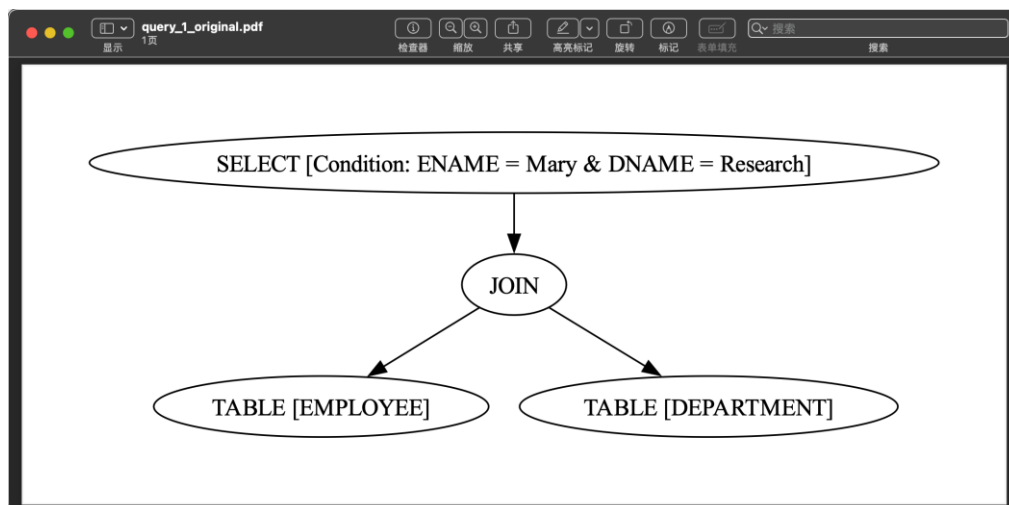
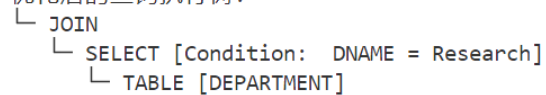
原始查询语句:

```
SELECT [ ENAME = 'Mary' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT )
```

原始查询执行树:



优化后的查询执行树:



--- 查询 2 ---

原始查询语句:

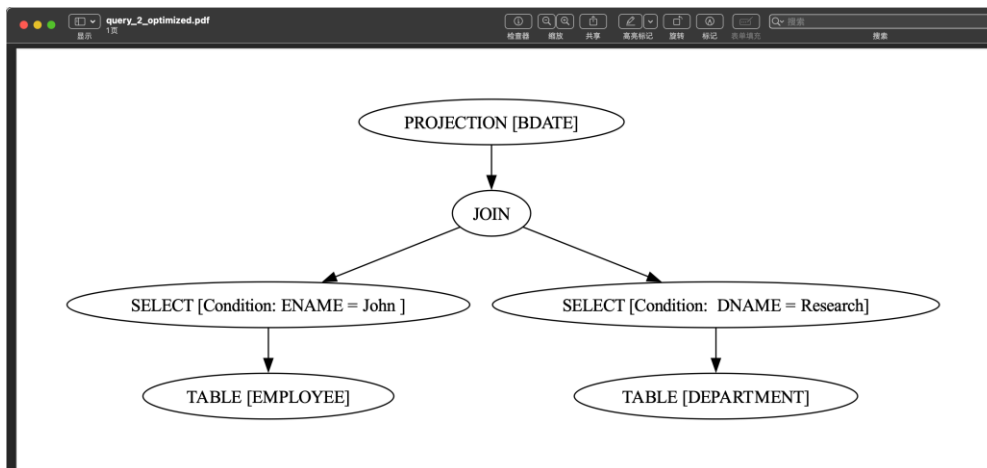
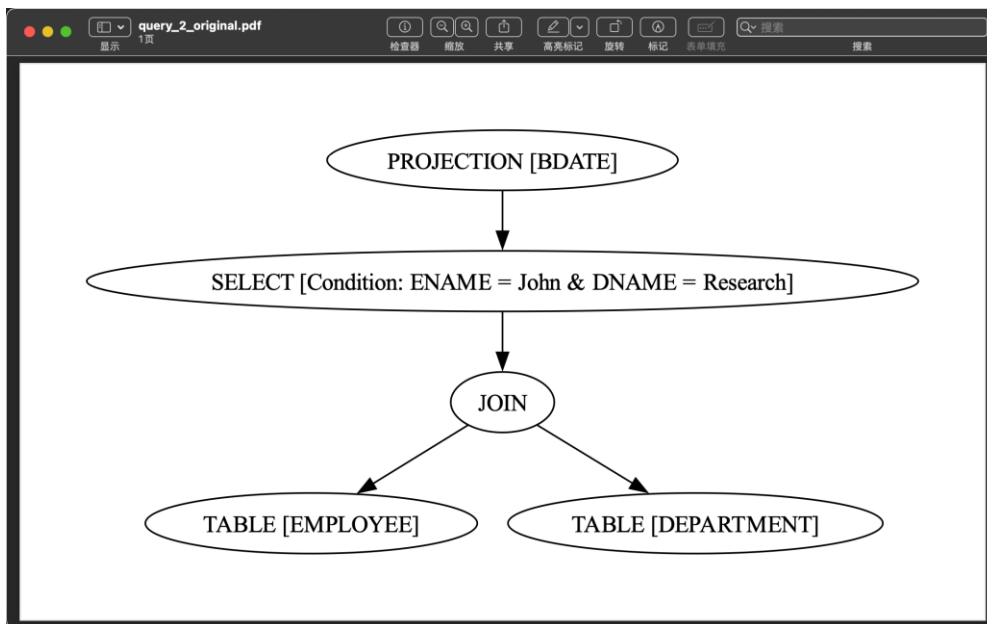
```
PROJECTION [ BDATE ] ( SELECT [ ENAME = 'John' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT ) )
```

原始查询执行树:

```
└─ PROJECTION [BDATE]
   └─ SELECT [Condition: ENAME = John & DNAME = Research]
      └─ JOIN
         ├── TABLE [EMPLOYEE]
         └── TABLE [DEPARTMENT]
```

优化后的查询执行树:

```
└─ PROJECTION [BDATE]
   └─ JOIN
      ├── SELECT [Condition: ENAME = John ]
      │   └─ TABLE [EMPLOYEE]
      └── SELECT [Condition: DNAME = Research]
          └─ TABLE [DEPARTMENT]
```



--- 查询 3 ---

原始查询语句:

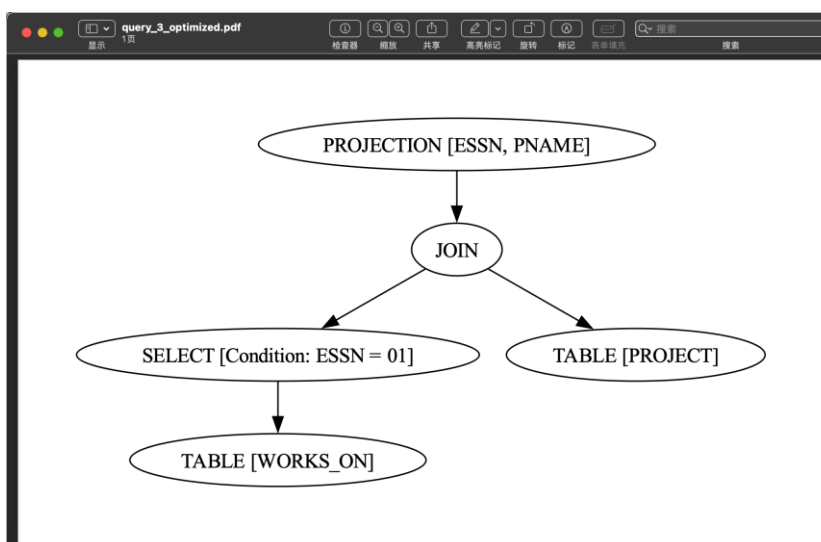
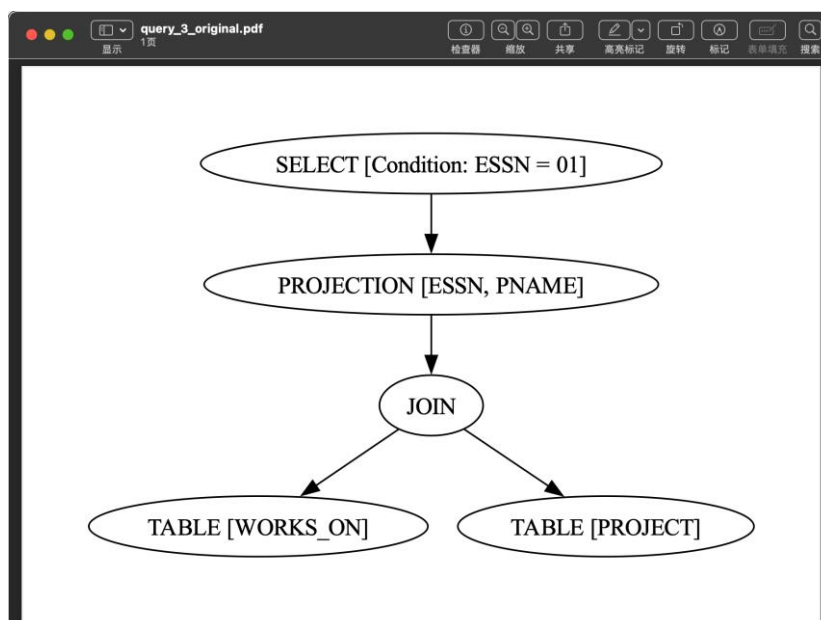
```
SELECT [ ESSN = '01' ] ( PROJECTION [ ESSN, PNAME ] ( WORKS_ON JOIN PROJECT ) )
```

原始查询执行树:

```
└─ SELECT [Condition: ESSN = 01]
   └─ PROJECTION [ESSN, PNAME]
      └─ JOIN
         ├── TABLE [WORKS_ON]
         └── TABLE [PROJECT]
```

优化后的查询执行树:

```
└─ PROJECTION [ESSN, PNAME]
   └─ JOIN
      ├── SELECT [Condition: ESSN = 01]
      │   └─ TABLE [WORKS_ON]
      └── TABLE [PROJECT]
```



四、实验心得

在此次实验中，我深入学习并实践了关系连接和查询优化算法的实现过程。通过编写代码，生成数据集并在模拟磁盘环境中存储，我加深了对关系型数据库物理存储和块操作的理解。在关系连接部分，我实现了多种连接方法，包括嵌套循环连接、哈希连接和归并连接。通过对比其 I/O 复杂性和运行效率，明确了不同连接方式的适用场景。

在查询优化实验中，我设计并生成了查询执行树，学习了选择下推和条件分解等优化策略的实际应用。这些优化在减少中间结果规模、提升查询效率方面效果显著。同时，通过引入 Python 的 PLY 库进行语法解析，以及 Graphviz 库对查询执行树的可视化，我体会到了工具在复杂逻辑展示和实现中的重要作用。

实验让我认识到理论与实践相结合的重要性，也增强了我分析问题和设计解决问题的能力。在未来的学习中，我将进一步探索更复杂的优化算法，努力提升对数据库系统性能优化的掌握程度。