哈尔滨工业大学(深圳)

# 《数据库》实验报告

## 实验五

## 查询处理算法的模拟实现


学　院: 计算机科学与技术

姓　名: 李聪

学　号: 200111205

专　业: 计算机科学与技术


日　期: 2022-12-19

# 一、 实验目的

*阐述本次实验的目的。*

1. 理解索引的作用；
2. 掌握关系选择、投影、连接、集合的交、并、差等操作的实现算法；
3. 加深对算法 I/O 复杂性的理解。

# 二、 实验环境

*阐述本次实验的环境。*

操作系统：Windows10

编程语言：C

工具：CodeBlocks

# 三、 实验内容

*阐述本次实验的具体内容。*

1. 实现基于线性搜索的关系选择算法：基于 ExtMem 程序库，使用 C 语言实现线性搜索算法，选出 S.C=128 的元组，记录 IO 读写次数，并将选择结果存放在磁盘上。（模拟实现 select S.C,S.D from S where S.C = 128）

2. 实现两阶段多路归并排序算法（TPMMS）：利用内存缓冲区将关系 R 和 S 分别排序，并将排序后的结果存放在磁盘上。

**3.** 实现基于索引的关系选择算法：利用 **2** 中的排序结果为关系 **R** 或 **S** 分别建立索引文件，利用索引文件选出 **S.C=128** 的元组，并将选择结果存放在磁盘上。记录 **IO** 读写次数，与 **1** 中的结果对比。（模拟实现 **select S.C, S.D from S where S.C = 128** ）

**4.** 实现基于排序的连接操作算法（**Sort-Merge-Join**）：对关系 **S** 和 **R** 计算 **S.C** 连接 **R.A** ，并统计连接次数，将连接结果存放在磁盘上。（模拟实现 **select S.C, S.D, R.A, R.B from S inner join R on S.C = R.A**）

**5.** 实现基于排序或散列的两趟扫描算法，实现其中一种集合操作算法：并（**S ∪ R**）、交（**S ∩ R**）、差（**S - R**）中的一种。将结果存放在磁盘上，并统计并、交、差操作后的元组个数。

**6.** 附加题：集合操作算法：并、交、差中剩余的两种。

# 四、 实验过程

*对实验中的 5 个题目分别进行分析，并对核心代码和算法流程进行讲解，用自然语言描述解决问题的方案 。并给出程序正确运行的结果截图。*

**(1)** 实现基于线性搜索的关系选择算法

问题分析：

扫描关系 **S** 的所有元组，找出 **C** 属性值为 **128** 的全部元组，将结果存入磁盘。

**(101.blk, 102.blk, .....)**

任务实现：

**Step1.** 从磁盘读取关系 **S** 的一个数据块到内存；

**Step2.** 对数据块中的所有元组进行判断，将 **C** 属性值为 **128** 的元组放到输出缓冲区；

**Step3.** 判断输出缓冲区是否写满，若已写满，则写回磁盘(磁盘块号为 **101,102,……)**，清空输出缓冲区；

**Step4.** 释放读入块占用的内存空间，判断关系 **S** 的数据块是否读完，若没有，转到 **Step1**，否则进入 **Step5**；

**Step5.** 检查输出缓冲区是否有写入数据，若有，将其写回磁盘；

**Step6.** 释放空间。

关键代码：

```
// 依次读取磁盘数据块
for (i = S_start; i <= S_end; i++)
{
    if ((rblk = readBlockFromDisk(i, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", i);

    // 读数据块中的7个元组
    for (j = 0; j < 7; j++)
    {
        for (k = 0; k < 4; k++)
        {
            str[k] = *(rblk + j*8 + k);
        }
        C = atoi(str);
        if (C != val)
        {
            continue;
        }
        cnt++;
        for (k = 0; k < 4; k++)
        {
            str[k] = *(rblk + j*8 + 4 + k);
        }
        D = atoi(str);
        printf("(%d, %d) \n", C, D);
        // 将选择结果放入输出缓冲块
        memcpy(wblk + usage*8, rblk + j*8, 8);
        usage++;
        // 输出缓冲块写满后写回磁盘
        if(usage == 7)
        {
            wnum++;
            itoa(100+wnum+1, wblk + usage*8, 10);   // 后继块的地址
            if (writeBlockToDisk(wblk, 100+wnum, buf) != 0)
            {
                perror("Writing Block Failed!\n");
                return -1;
            }
            printf("写入磁盘块: %d\n", 100+wnum);
            wblk = getNewBlockInBuffer(buf);
            memset(wblk, 0, 64);
            usage = 0;
        }
    }
    // 释放空间
    freeBlockInBuffer(rblk, buf);
}
```

**实验结果：**

```
------------------------------
基于线性搜索的关系选择算法 S.C = 128
------------------------------
读入数据块17
读入数据块18
读入数据块19
读入数据块20
读入数据块21
读入数据块22
读入数据块23
(128, 684)
(128, 431)
读入数据块24
读入数据块25
(128, 615)
(128, 429)
读入数据块26
读入数据块27
读入数据块28
读入数据块29
读入数据块30
(128, 584)
读入数据块31
读入数据块32
读入数据块33
读入数据块34
(128, 592)
(128, 457)
写入磁盘块: 101
读入数据块35
(128, 720)
读入数据块36
读入数据块37
读入数据块38
读入数据块39
读入数据块40
读入数据块41
读入数据块42
读入数据块43
读入数据块44
读入数据块45
(128, 447)
读入数据块46
(128, 871)
读入数据块47
读入数据块48
写入磁盘块: 102

满足条件的元组个数 10
IO读写一共34次
```

**(2) 实现两阶段多路归并排序算法（TPMMS）**

问题分析：对关系的所有数据块进行两趟扫描，第一趟划分子集并进行子集排序(结果写入 200+原来块号.blk)，第二趟进行各子集间的归并排序(结果写入 300+原来块号.blk)。

任务实现：

==========划分子集&子集排序==========

**Step1.** 读入一个子集合(8 个数据据块)；

**Step2.** 对读入内存的所有元组(共 56 个)，进行冒泡排序，排序码为关系的第一个属性(R.A 或者 S.C)；

**Step3.** 将子集合写回磁盘(磁盘块号为原来的块号+200);

**Step4.** 判断关系是否全部处理完，若是，划分结束；若否，回到 Step1。

==========各子集间的归并排序==========

**Step1.** 读入每个子集合的第一个数据据块；

**Step2.** 判断所有子集合是否都完成排序，若是，流程结束；若否，进入 Step3;

**Step3.** 从排序完毕的所有子集合选出一个属性值最小的待排序元组，比较属性值大小，选出最小的元组放入输出缓冲区；

**Step4.** 判断输出缓冲区是否写满，若已写满，则写回磁盘(磁盘块号为原来块号+300)，清空输出缓冲区；

**Step5.** 从最小属性值的元组所在子集合选出下一个待排序元组；

**Step6.** 判断最小属性值的元组所在子集合当前读入的数据块是否全部处理完毕，若否，回到 Step2；若是，进入 Step6；

**Step7.** 判断最小属性值的元组所在子集合的所有数据块是否处理完毕，若否，从磁盘读入一个新的数据块，回到 Step2；若是，进入 Step7；

**Step8.** 将最小属性值的元组所在子集合标记为排序完毕，回到 Step2；

关键代码：

函数声明部分：

```
/**
 * @brief
 * TPMMS的第一趟：划分子集合并进行子集合排序，子集合含8个磁盘块
 * 中间结果存入200+原来磁盘块号的内存中
 *
 * @param buf    内存缓冲区
 * @param start  待排序全集的起始块号
 * @param end    待排序全集的终止块号
 */
void divideAndSort(Buffer *buf, int start, int end);

/**
 * @brief
 * TPMMS的第二趟：各子集间的归并排序
 * 结果放在300+原来磁盘号上
 *
 * @param buf    内存缓冲区
 * @param start  已排序子集的起始块号
 * @param end    已排序子集的终止块号
 */
void mergeSort(Buffer *buf, int start, int end);

/**
 * @brief
 * 两阶段多路归并排序算法
 *
 * @param buf 内存缓冲区
 * @param relation 需要进行排序的关系
 */
void TPMMS(Buffer *buf, char relation);
```

函数定义的关键部分：

**divideAndSort():**

```
// 每次处理一个子集合(8块)
for (outer = start; outer <= end; outer += 8)
{
    // 子集合全部读入内存
    for (inner = 0; inner < 8; inner++)
    {
        if ((blks[inner] = readBlockFromDisk(outer + inner, buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        printf("读入数据块%d\n", outer+inner);
    }

    // 使用冒泡排序法对子集合排序，排序码为第一个属性(A或C)
    // 子集合共56个元组
      printf("进行冒泡排序\n");
    for (i = 0; i < 56; i++)
    {
        for (j = 0; j < 55 - i; j++)
        {
            bid = j / 7;
            tid = j % 7;
            bid1 = (j+1) / 7;
            tid1 = (j+1) % 7;
            if(readNumFromBlk(blks[bid], tid, 0) > readNumFromBlk(blks[bid1], tid1, 0))
            {
                // 进行交换
                memcpy(temp, blks[bid] + tid*8, 8); // temp是指针时会出错
                memcpy(blks[bid] + tid*8, blks[bid1] + tid1*8, 8);
                memcpy(blks[bid1] + tid1*8, temp, 8);
            }
```

```
        }
    }
}
// printf("排序完成\n");
// 写回磁盘200+原来所在块号
for(inner = 0; inner < 8; inner++)
{
    itoa(200+outer+inner+1, blks[inner]+56, 10);    // 后继块的地址
    if (writeBlockToDisk(blks[inner], 200 + outer + inner, buf) != 0)
    {
        perror("Writing Block Failed!\n");
        return -1;
    }
    memset(blks[inner], 0, 64);
    printf("写回磁盘%d\n", 200+outer+inner);
}
```

**mergeSort():**

```
// 读入每个子集合的第一块
for (i = 0; i < sub_num; i++)
{
    bid[i] = 0;
    tid[i] = 0;
    finish[i] = FALSE;
    if ((rblks[i] = readBlockFromDisk(start + i*8, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", start + i*8);
}

while(arrSum(finish, sub_num) < sub_num)
{
    small = 1 << 30;
    small_id = -1;
    // 找到当前归并的sub_num路元组的最小值
    for (i = 0; i < sub_num; i++)
    {
        if (finish[i] == TRUE)
        {
            // 这个子集合已经全部处理完毕
            continue;
        }
        else if (readNumFromBlk(rblks[i], tid[i], 0) < small)
        {
            small = readNumFromBlk(rblks[i], tid[i], 0);
            small_id = i;
        }
    }
    // printf("small: %d\n", small);
    if (small_id == -1)
    {
        perror("MergeSort Error!\n");
        return -1;
    }
    memcpy(wblk + usage*8, rblks[small_id] + tid[small_id]*8, 8);

    usage++;
    // 判断输出缓冲块是否写满
    if(usage == 7)
    {
        itoa(100+start+wnum+1, wblk + usage*8, 10);    // 后继块的地址
        if (writeBlockToDisk(wblk, 100+start+wnum, buf) != 0)
        {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("写回磁盘: %d\n", 100+start+wnum);
        wblk = getNewBlockInBuffer(buf);
        memset(wblk, 0, 64);
        usage = 0;
        wnum++;
    }

    tid[small_id]++;    // 处理完一个元组，向后移一位
    // 判断该子集合输入的磁盘块是否全部处理完
    if(tid[small_id] == 7)
    {
        bid[small_id]++;
        // 判断该子集合是否全部处理完毕
        if (bid[small_id] == 8)
        {
            finish[small_id] = TRUE;
        }
```

```
        else
        {
            freeBlockInBuffer(rblks[small_id], buf);
            // 输入对应子集合的下一个磁盘块
            if ((rblks[small_id] = readBlockFromDisk(start+small_id*8+bid[small_id], buf)) == NULL)
            {
                perror("Reading Block Failed!\n");
                return -1;
            }
            printf("读入数据块: %d\n", start+small_id*8+bid[small_id]);
            tid[small_id] = 0;
        }
    }
}

// 释放内存空间
freeBuffer(buf);
```

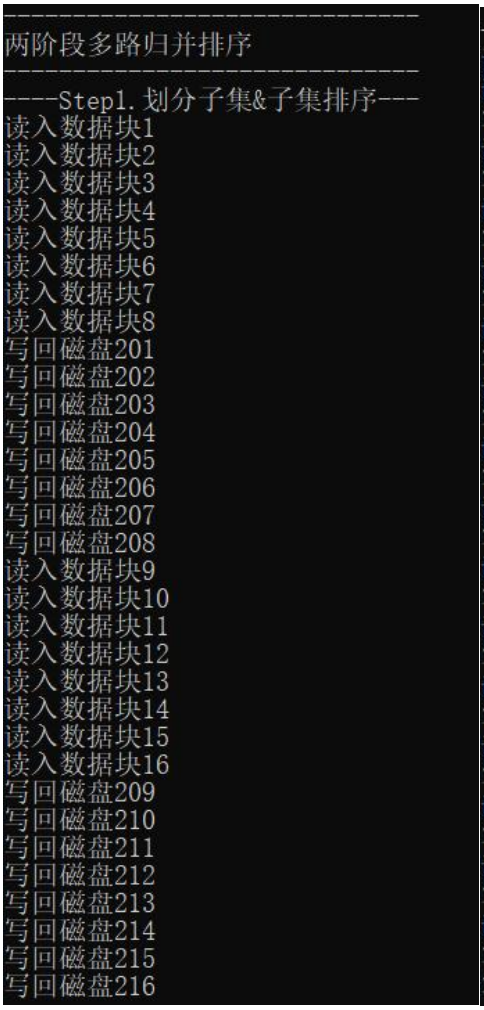
**TPMMS():**

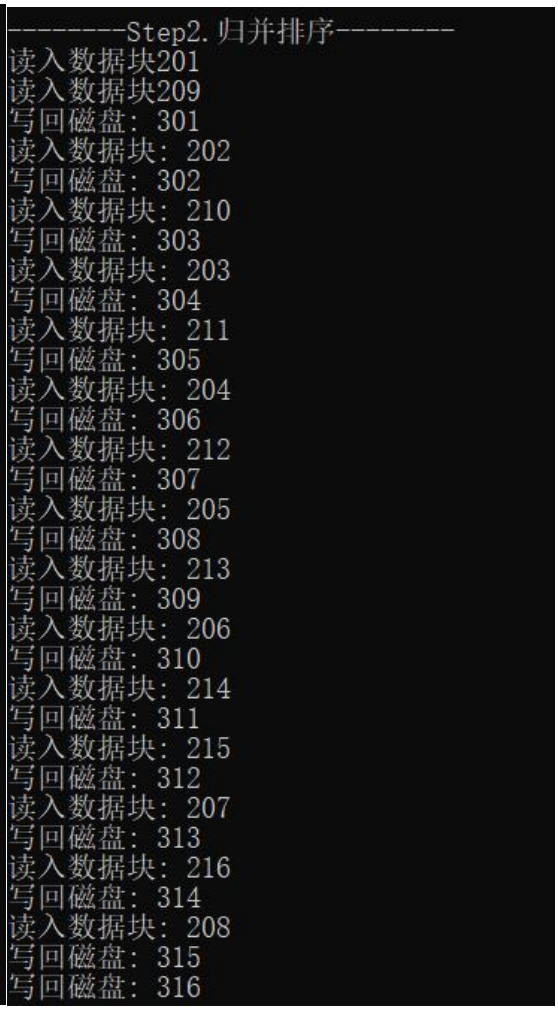
```
void TPMMS(Buffer *buf, char relation)
{
    int start, end;
    printf("\n------------------------------\n");
    printf("两阶段多路归并排序\n");
    printf("------------------------------\n");

    switch (relation)
    {
        case 'R':
            start = 1;
            end = 16;
            break;
        case 'S':
            start = 17;
            end = 48;
            break;
        default:
            perror("Selecting Relation Faild\n");
            return -1;
    }
    divideAndSort(buf, start, end);
    mergeSort(buf, 200+start, 200+end);
}
```

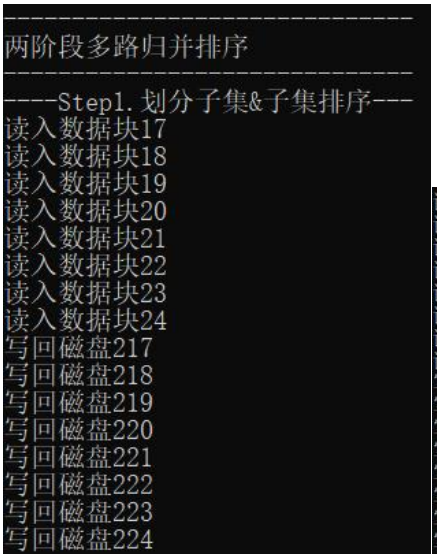
**实验结果：**

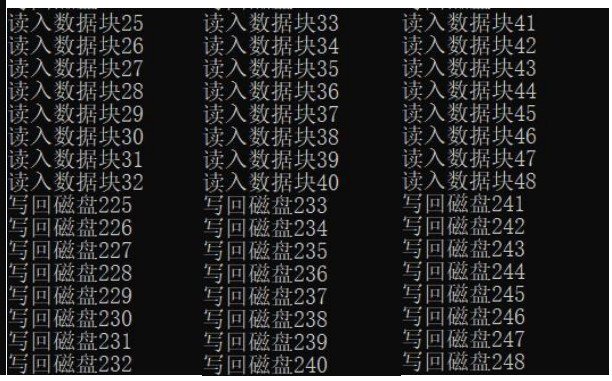
**关系 R**


两阶段多路归并排序
----Step1.划分子集&子集排序---
读入数据块1
读入数据块2
读入数据块3
读入数据块4
读入数据块5
读入数据块6
读入数据块7
读入数据块8
写回磁盘201
写回磁盘202
写回磁盘203
写回磁盘204
写回磁盘205
写回磁盘206
写回磁盘207
写回磁盘208
读入数据块9
读入数据块10
读入数据块11
读入数据块12
读入数据块13
读入数据块14
读入数据块15
读入数据块16
写回磁盘209
写回磁盘210
写回磁盘211
写回磁盘212
写回磁盘213
写回磁盘214
写回磁盘215
写回磁盘216


--------Step2.归并排序--------
读入数据块201
读入数据块209
写回磁盘：301
读入数据块：202
写回磁盘：302
读入数据块：210
写回磁盘：303
读入数据块：203
写回磁盘：304
读入数据块：211
写回磁盘：305
读入数据块：204
写回磁盘：306
读入数据块：212
写回磁盘：307
读入数据块：205
写回磁盘：308
读入数据块：213
写回磁盘：309
读入数据块：206
写回磁盘：310
读入数据块：214
写回磁盘：311
读入数据块：215
写回磁盘：312
读入数据块：207
写回磁盘：313
读入数据块：216
写回磁盘：314
读入数据块：208
写回磁盘：315
写回磁盘：316

**关系 S：**


两阶段多路归并排序
----Step1.划分子集&子集排序---
读入数据块17
读入数据块18
读入数据块19
读入数据块20
读入数据块21
读入数据块22
读入数据块23
读入数据块24
写回磁盘217
写回磁盘218
写回磁盘219
写回磁盘220
写回磁盘221
写回磁盘222
写回磁盘223
写回磁盘224


读入数据块25
读入数据块26
读入数据块27
读入数据块28
读入数据块29
读入数据块30
读入数据块31
读入数据块32
写回磁盘225
写回磁盘226
写回磁盘227
写回磁盘228
写回磁盘229
写回磁盘230
写回磁盘231
写回磁盘232
读入数据块33
读入数据块34
读入数据块35
读入数据块36
读入数据块37
读入数据块38
读入数据块39
读入数据块40
写回磁盘233
写回磁盘234
写回磁盘235
写回磁盘236
写回磁盘237
写回磁盘238
写回磁盘239
写回磁盘240
读入数据块41
读入数据块42
读入数据块43
读入数据块44
读入数据块45
读入数据块46
读入数据块47
读入数据块48
写回磁盘241
写回磁盘242
写回磁盘243
写回磁盘244
写回磁盘245
写回磁盘246
写回磁盘247
写回磁盘248

```
--------Step2.归并排序--------
读入数据块217
读入数据块225
读入数据块233
读入数据块241
写回磁盘：317
写回磁盘：318
读入数据块：226
写回磁盘：319
读入数据块：218
读入数据块：242
写回磁盘：320
写回磁盘：321
读入数据块：234
写回磁盘：322
读入数据块：219
读入数据块：243
写回磁盘：323
写回磁盘：324
读入数据块：235
写回磁盘：325
读入数据块：220
写回磁盘：326
写回磁盘：327
读入数据块：244
写回磁盘：328
读入数据块：227
读入数据块：236
写回磁盘：329
读入数据块：221
读入数据块：245
写回磁盘：330
写回磁盘：331
读入数据块：228
写回磁盘：332
读入数据块：237
读入数据块：246
写回磁盘：333
读入数据块：222
写回磁盘：334
读入数据块：229
写回磁盘：335
写回磁盘：336
读入数据块：230
写回磁盘：337
读入数据块：238
读入数据块：247
写回磁盘：338
写回磁盘：339
读入数据块：223
读入数据块：231
写回磁盘：340
写回磁盘：341
读入数据块：248
写回磁盘：342
读入数据块：239
写回磁盘：343
写回磁盘：344
读入数据块：240
读入数据块：224
写回磁盘：345
读入数据块：232
写回磁盘：346
写回磁盘：347
写回磁盘：348
```

**(3)** 实现基于索引的关系选择算法

问题分析：

利用前面的排序结果为关系 **S** 创建索引文件**(417.blk, 418.blk, .....)**，并利用索引文件选出 **S.C=128** 的元组，将选择结果放入磁盘**(501.blk, 502.blk, .....)**。

任务实现：

**==========**创建关系 **S** 的索引文件**==========**

**Step1.** 从已排序的主文件中读取一个数据块到内存；

**Step2.** 从读入的数据块读取一个元组，根据 **S.C** 的值判断是否出现新的索引字段，若否，进入 **Step3**；若是，将索引指针的值设为当前数据块块号，将索引字段和索引指针一起写入输出缓冲区。判断输出缓冲区是否写满，若否，进入 **Step3**；若是，将输出缓冲区写回磁盘并清空，进入 **Step3**；

**Step3.** 判断读入数据块中的元组是否都搜索完毕，若否，回到 **Step2**，若是，释放输入缓冲区，进入 **Step4**；

**Step4.** 判断主文件的所有数据块是否搜索完毕，若否，回到 **Step1**，若是，进入 **Step5**；

**Step5.** 检查输出缓冲区是否有数据，若有，将其写回磁盘。

**==========**基于索引的关系选择**==========**

**Step1.** 在索引文件中找到索引字段为 **128** 的索引项，保存其索引指针指向的磁盘块号；

**Step2.** 从找到的磁盘块号开始，读取一个数据块到内存；

**Step3.** 遍历数据块中的所有元组：

如果 **S.C < 128**，跳过；

如果 **S.C = 128**，将该元组写入输出缓冲区。如果输出缓冲区写满，则写回磁盘，并清空；

如果 **S.C > 128**，设置结束标识。

**Step4.** 判断是否结束，若否，读取下一个磁盘块，回到 **Step3**；若是，进入 **Step5;**

**Step5.** 检查输出缓冲区是否有数据，若有，将其写回磁盘。选择结束。

关键代码：

函数声明

```
/**
 * @brief
 * 创建关系S的索引文件，结果写入块号为400+序号的磁盘中
 *
 * @param buf 内存缓冲区
 *
*/
void creatIndexOfS(Buffer *buf);

/**
 * @brief
 * 基于索引的关系选择算法，结果写入501.blk,502.blk,.....
 *
 * @param buf 内存缓冲区
 * @param val 进行选择的值
*/
void indexBasedSelect(Buffer *buf, int val);
```

函数定义的关键部分

**creatIndexOfS():**

```
for (i = start; i <= end; i++)
{
    // 从已排序的文件中读取一个数据块
    if ((rblk = readBlockFromDisk(300+i, buf)) == NULL)
    {
        perror("Reading Sorted Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", 300 + i);
    for (j = 0; j < 7; j++)
    {
        if(idx_field != readNumFromBlk(rblk, j, 0))
        {
            // 出现新的索引字段
              printf("idx: %d, new: %d, j: %d\n", idx_field, readNumFromBlk(rblk, j, 0), j);
            idx_field = readNumFromBlk(rblk, j, 0);
            itoa(300+i, pointer, 10);

            memcpy(wblk + usage*8, rblk + j*8, 4);       // 将索引字段写入输出缓冲区
            memcpy(wblk + usage*8 + 4, pointer, 4);      // 将指针写入输出缓冲区
            memset(pointer, 0, 4);
            usage++;
            // 判断输出缓冲块是否写满
            if(usage == 7)
            {
                itoa(400+start+wnum+1, wblk + usage*8, 10);   // 后继块的地址
                if (writeBlockToDisk(wblk, 400+start+wnum, buf) != 0)
                {
                    perror("Writing Block Failed!\n");
                    return -1;
                }
                printf("写回磁盘: %d\n", 400+start+wnum);
                wblk = getNewBlockInBuffer(buf);
                memset(wblk, 0, 64);
                usage = 0;
                wnum++;
            }
        }
    }
    freeBlockInBuffer(rblk, buf);
}
```

**indexBasedSelect():**

```
i = istart;
// 在索引文件中搜索索引字段
while(!find)
{
    if ((rblk = readBlockFromDisk(i, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", i);

    for (j = 0; j < 7; j++)
    {
        if(val == readNumFromBlk(rblk, j, 0))
        {
            tstart = readNumFromBlk(rblk, j, 1);
            find = TRUE;
            freeBlockInBuffer(rblk, buf);
            break;
        }
    }
    i++;
    freeBlockInBuffer(rblk, buf);
}

// 根据找到的索引字段的指针在主文件中搜索
for(i = tstart; !end;i++)
{
    if ((rblk = readBlockFromDisk(i, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", i);

    for (j = 0; j < 7; j++)
    {
        C = readNumFromBlk(rblk, j, 0);
        if (C < val)
        {
            continue;
        }
        else if (C == val)
        {
            // 找到符合条件的元组，写入输出缓冲区
            memcpy(wblk + usage*8, rblk + j*8, 8);
            usage++;
            cnt++;
            // 判断输出缓冲块是否写满
            if(usage == 7)
            {
                itoa(500+wnum+2, wblk + usage*8, 10);   // 后继块的地址
                if (writeBlockToDisk(wblk, 500+1+wnum, buf) != 0)
                {
                    perror("Writing Block Failed!\n");
                    return -1;
                }
                printf("写回磁盘: %d\n", 500+1+wnum);
                wblk = getNewBlockInBuffer(buf);
                memset(wblk, 0, 64);
                usage = 0;
                wnum++;
            }
        }
        else
        {
            end = TRUE;
            break;
        }
    }
}
```

**实验结果：**

```
------------------------------
创建索引文件
------------------------------
读入数据块317
读入数据块318
读入数据块319
读入数据块320
读入数据块321
读入数据块322
读入数据块323
写回磁盘：417
读入数据块324
读入数据块325
读入数据块326
读入数据块327
读入数据块328
写回磁盘：418
读入数据块329
读入数据块330
读入数据块331
读入数据块332
读入数据块333
读入数据块334
写回磁盘：419
读入数据块335
读入数据块336
读入数据块337
读入数据块338
读入数据块339
写回磁盘：420
读入数据块340
读入数据块341
读入数据块342
读入数据块343
读入数据块344
写回磁盘：421
读入数据块345
读入数据块346
读入数据块347
读入数据块348
写回磁盘：422
索引文件占用6块
```

```
--------------------------------
基于索引的关系选择
--------------------------------
读入数据块417
读入数据块418
读入数据块324
写回磁盘：501
读入数据块325
写回磁盘：502

满足条件的元组个数 10
IO读写一共6次
```

**(4) 实现基于排序的连接操作算法（Sort-Merge-Join）**

问题分析：对关系 R 和 S，将 S.C = R.A 的元组进行连接，将连接后的 S.C,S.D,R.A,R.B 存入磁盘(601.blk, 602.blk, .....)，统计连接次数。（模拟实现 select S.C, S.D, R.A, R.B from S inner join R on S.C = R.A）

任务实现：（利用任务二的结果）

**Step1.** 从关系 R 读入一个数据块；

**Step2.** 从关系 S 读入一个数据块；

**Step3.** 从 S 的输入块中获取一个元组，读取 S.C 的值，判断与上一个元组的值是否相同

若相同，将指向 R 的输入块的右指针回退到左指针位置，同时如果两个左右指针指向元组所在块号不同，需要将 R 的输入块的空间释放并重新读入左指针指向元组所在的数据块；

否则，将指向 R 的输入块的左指针移到右指针位置。

**Step4.** 判断 R 的输入块的右指针是否已超过最后一个元组

若是，执行 Step7；

否则根据右指针位置判断是否需要读入 R 的下一个数据块，从 R 的输入块的右指针位置获取一个元组，读取 R.A 的值

**Step5.**

如果 R.A < S.C，将 R 输入块的右指针右移一位，重复 Step4；

如果 R.A = S.C，进行连接，将结果写入输出缓冲区，并在输出缓冲区写满时写回磁盘，将 R 输入块的右指针右移一位，重复 Step4；

如果 R.A > S.C，判断 S 输入块中的元组是否处理完毕，若否，回到 Step3，若是，执行 Step6。

**Step6.** 释放 S 输入块的空间，判断 S 的数据块是否全部处理完，若否，回到 Step2，否则，执行 Step7.

**Step7.** 如果输出缓冲区右数据，则将其写回磁盘。

关键代码：

函数声明

```
/**
 * @brief
 * 实现基于排序的连接操作算法（Sort-Merge-Join）
 * 对关系S和R计算S.C连接R.A，并统计连接次数，将连接结果存放在磁盘601.blk,602.blk,.....
 * （模拟实现 select S.C, S.D, R.A, R.B from S innerjoin R on S.C = R.A）
 *
 * @param buf 内存缓冲区
*/
void sortMergeJoin(Buffer *buf);
```

函数实现的关键部分

```
int s_start = 317, s_end = 348; // 排序后的关系S的起始块号和终止块号
int r_start = 301, r_end = 316; // 排序后的关系R的起始块号和终止块号
unsigned char *s_rblk;  // 关系S的输入缓冲区
unsigned char *r_rblk;  // 关系R的输入缓冲区
unsigned char *wblk;    // 输出缓冲区
int r_left_bid = r_start, r_right_bid = r_start;    // 关系R左、右指针对应元组的块号
int r_left_tid = 0, r_right_tid = 0;                // 关系R左、右指针对应元组的元组号
int A, C;
int last_val = 0;   // 上一个S.C的值
int usage = 0;      // 输出缓冲区已写元组个数
int wnum = 0;       // 已写块数
int cnt = 0;        // 连接次数
int i, j;
```

读入关系 R 的数据块

```
// 从关系R读入一个数据块
if ((r_rblk = readBlockFromDisk(r_right_bid, buf)) == NULL)
{
    perror("Reading Block Failed!\n");
    return -1;
}
```

对关系 S 的元组进行一遍扫描，与关系 R 进行连接

```
for (i = s_start; i <= s_end; i++)
{
    if ((s_rblk = readBlockFromDisk(i, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
      printf("读入数据块%d\n", i);

    for (j = 0; j < 7; j++)
    {
        C = readNumFromBlk(s_rblk, j, 0);
        // 如果S.C的值与上一个元组相同，将指向关系R的右指针进行回退
        if (C == last_val)
        {
            // 如果块号不同，需要读回前面的块
            if (r_left_bid != r_right_bid)
            {
                freeBlockInBuffer(r_rblk, buf);
                if ((r_rblk = readBlockFromDisk(r_left_bid, buf)) == NULL)
                {
                    perror("Reading Block Failed!\n");
                    return -1;
                }
                  printf("读入数据块%d\n", r_left_bid);
                r_right_bid = r_left_bid;
            }
            r_right_tid = r_left_tid;
        }
        // S.C的值不同于上一个元组相同
        else
        {
            last_val = C;
            r_left_bid = r_right_bid;
            r_left_tid = r_right_tid;
        }

        // 从关系R中搜索R.A = S.C的元组
        if (r_right_bid > r_end)
        {
            A = 1 << 30;
        }
        else
        {
            A = readNumFromBlk(r_rblk, r_right_tid, 0);
        }
```

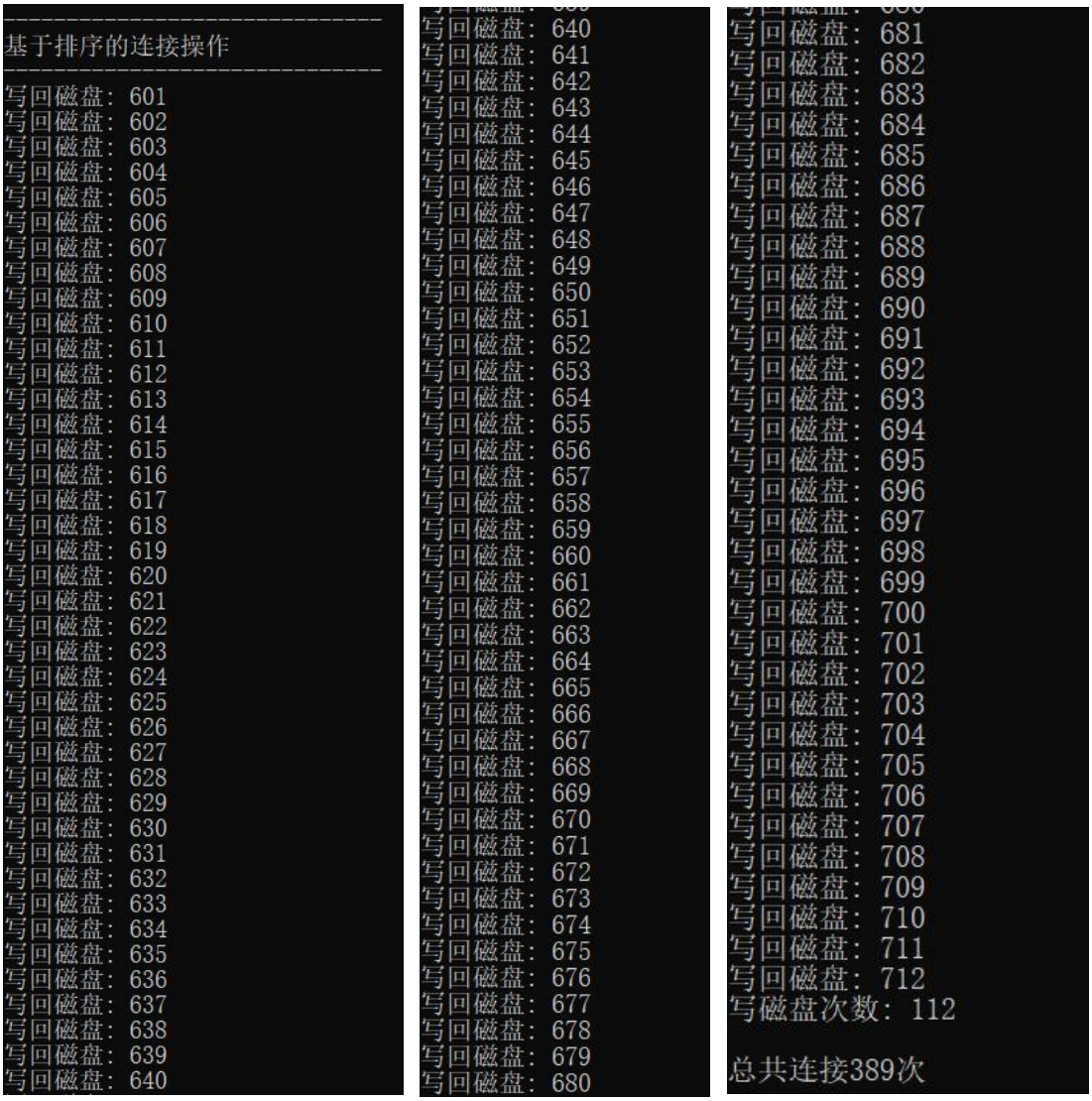
```
        while (A <= C)
        {
            if (A == C)
            {
                cnt++;
                // 将S.C,S.D写入输出缓冲区
                memcpy(wblk + usage*8, s_rblk + j*8, 8);
                usage++;
                // 判断输出缓冲块是否写满
                if(usage == 7)
                {
                    itoa(600+wnum+2, wblk + 56, 10);   // 后继块的地址
                    if (writeBlockToDisk(wblk, 600+1+wnum, buf) != 0)
                    {
                        perror("Writing Block Failed!\n");
                        return -1;
                    }
                    printf("写回磁盘: %d\n", 600+1+wnum);
                    wblk = getNewBlockInBuffer(buf);
                    memset(wblk, 0, 64);
                    usage = 0;
                    wnum++;
                }
                // 将R.A,R.B写入输出缓冲区
                memcpy(wblk + usage*8, r_rblk + r_right_tid*8, 8);
                usage++;
                // 判断输出缓冲块是否写满
                if(usage == 7)
                {
                    itoa(600+wnum+2, wblk + 56, 10);   // 后继块的地址
                    if (writeBlockToDisk(wblk, 600+1+wnum, buf) != 0)
                    {
                        perror("Writing Block Failed!\n");
                        return -1;
                    }
                    printf("写回磁盘: %d\n", 600+1+wnum);
                    wblk = getNewBlockInBuffer(buf);
                    memset(wblk, 0, 64);
                    usage = 0;
                    wnum++;
                }
            }

            r_right_tid ++;
            if (r_right_tid == 7)
            {
                r_right_bid++;
                r_right_tid = 0;
                // 当前数据块处理完毕，释放空间，读入下一个数据块
                freeBlockInBuffer(r_rblk, buf);
                if (r_right_bid > r_end)
                {
                    A = 1 << 30;
                }
                else if ((r_rblk = readBlockFromDisk(r_right_bid, buf)) == NULL)
                {
                    perror("Reading Block Failed!\n");
                    return -1;
                }
                  printf("读入数据块%d\n", r_right_bid);
            }
            if (r_right_bid > r_end)
            {
                A = 1 << 30;
            }
            else
            {
                A = readNumFromBlk(r_rblk, r_right_tid, 0);
            }
        }
    }
    freeBlockInBuffer(s_rblk, buf);
}
```

## 将输出缓冲区剩余部分写回磁盘

```
// 检查输出缓冲区是否有数据，若有，将其写回磁盘
if (usage != 0)
{
    itoa(600+wnum+2, wblk + 56, 10);   // 后继块的地址
    if (writeBlockToDisk(wblk, 600+1+wnum, buf) != 0)
    {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("写回磁盘: %d\n", 600+1+wnum);
    wnum++;
}
```

实验结果：




**(5) 实现基于散列的两趟扫描算法，实现交、并、差其中一种集合操作算法（利用任务二的结果实现集合差操作）**

问题分析：遍历关系 S 的元组，将出现在 S 中，但未出现在 R 中的元组(在 R 中不存在元组同时满足 S.C = R.A, S.D = R.B)筛选出来，存放在磁盘上(1001.blk, 1002.blk, .....)。

任务实现：

**Step1.** 从关系 R 读入一个数据块；

**Step2.** 从关系 S 读入一个数据块；

**Step3.** 从 S 的输入块中获取一个元组，读取 S.C,S.D 的值，判断 S.C 与上一个元组的值是否相同

若相同，将指向 R 的输入块的右指针回退到左指针位置，同时如果两个左右指针指向元组所在块号不同，需要将 R 的输入块的空间释放并重新读入左指针指向元组所在的数据块；

否则，将指向 R 的输入块的左指针移到右指针位置。

**Step4.** 判断 R 的输入块的右指针是否已超过最后一个元组

若是，执行 **Step7**；

否则根据右指针位置判断是否需要读入 R 的下一个数据块，从 R 的输入块的右指针位置获取一个元组，读取 R.A, R.B 的值

**Step5.**

如果 **R.A < S.C**，将 R 输入块的右指针右移一位，重复 **Step4**；

如果 **R.A = S.C**，且 **R.B = S.D**，将这个 S 中的元组设为无效，重复 **Step4**；

如果 **R.A > S.C**，执行 **Step6**。

**Step6.** 如果当前这个 S 中的元组有效，将其写入输出缓冲区，如果输出缓冲区写满，需要将其写入磁盘，并重新分配缓冲区。

**Step7.** 判断 S 输入块中的元组是否处理完毕，若否，回到 **Step3**，若是，执行 **Step8.**

**Step8.** 释放 S 输入块的空间，判断 S 的数据块是否全部处理完，若否，回到 **Step2**，否则，执行 **Step9.**

**Step9.** 如果输出缓冲区右数据，则将其写回磁盘。

关键代码：

函数声明

```
/**
 * @brief
 * 基于排序的两趟扫描算法，实现集合的差操作
 * 将结果存放在1001.blk, 1002.blk, ......
 *
 * @param buf 内存缓冲区
*/
void sortBasedDifference(Buffer *buf);
```

函数定义的关键部分

声明的变量

```
int s_start = 317, s_end = 348; // 排序后的关系S的起始块号和终止块号
int r_start = 301, r_end = 316; // 排序后的关系R的起始块号和终止块号
unsigned char *s_rblk;  // 关系S的输入缓冲区
unsigned char *r_rblk;  // 关系R的输入缓冲区
unsigned char *wblk;    // 输出缓冲区
int r_left_bid = r_start, r_right_bid = r_start;     // 关系R左、右指针对应元组的块号
int r_left_tid = 0, r_right_tid = 0;                 // 关系R左、右指针对应元组的元组号
int A, B, C, D;
int last_val = 0;   // 上一个S.C的值
int usage = 0;      // 输出缓冲区已写元组个数
int wnum = 0;       // 已写块数
BOOL valid;         // 元组是否有效
int cnt = 0;        // 差集个数
int i, j;
```

先从 R 读入一个数据块

```
// 从关系R读入一个数据块
if ((r_rblk = readBlockFromDisk(r_right_bid, buf)) == NULL)
{
    perror("Reading Block Failed!\n");
    return -1;
}
```

遍历关系 S，找出符合条件的元组

```
for (i = s_start; i <= s_end; i++)
{
    if ((s_rblk = readBlockFromDisk(i, buf)) == NULL)
    {
        perror("Reading Block Failed!\n");
        return -1;
    }
    // printf("读入数据块%d\n", i);

    for (j = 0; j < 7; j++)
    {
        valid = TRUE;
        C = readNumFromBlk(s_rblk, j, 0);
        D = readNumFromBlk(s_rblk, j, 1);
        // 如果S.C的值与上一个元组相同，将指向关系R的右指针进行回退
        if (C == last_val)
        {
            // 如果块号不同，需要读回前面的块
            if (r_left_bid != r_right_bid)
            {
                freeBlockInBuffer(r_rblk, buf);
                if ((r_rblk = readBlockFromDisk(r_left_bid, buf)) == NULL)
                {
                    perror("Reading Block Failed!\n");
                    return -1;
                }
                // printf("读入数据块%d\n", r_left_bid);
                r_right_bid = r_left_bid;
            }
            r_right_tid = r_left_tid;
        }
        // S.C的值不同于上一个元组相同
        else
        {
            last_val = C;
            r_left_bid = r_right_bid;
            r_left_tid = r_right_tid;
        }

        // 从关系R中搜索R.A = S.C的元组
        if (r_right_bid > r_end)
        {
            A = 1 << 30;
            B = 1 << 30;
        }
        else
        {
            A = readNumFromBlk(r_rblk, r_right_tid, 0);
            B = readNumFromBlk(r_rblk, r_right_tid, 1);
        }
        // 在关系R中找到R.A = S.C的元组
```

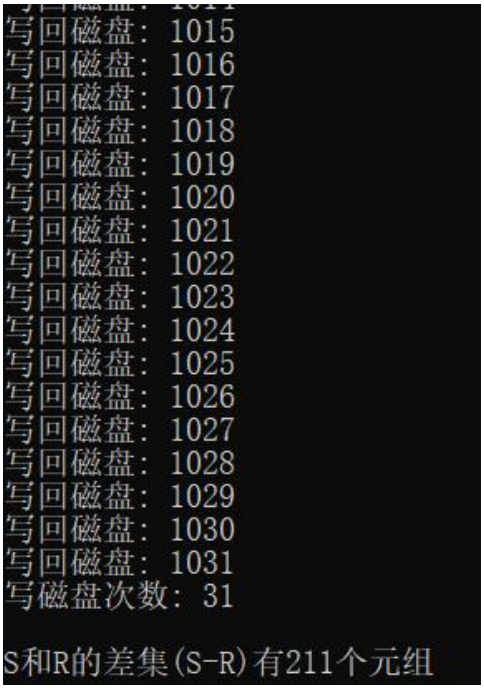
```
        // 在关系R中找到R.A = S.C的元组
        // 如果R.B = S.D也成立，则需将这个S中的元组去掉
        while (A <= C)
        {
            if (A == C && B == D)
            {
                valid = FALSE;
            }

            r_right_tid++;
            if (r_right_tid == 7)
            {
                r_right_bid++;
                r_right_tid = 0;
                // 当前数据块处理完毕，释放空间，读入下一个数据块
                freeBlockInBuffer(r_blk, buf);
                if (r_right_bid > r_end)
                {
                    A = 1 << 30;
                }
                else if ((r_blk = readBlockFromDisk(r_right_bid, buf)) == NULL)
                {
                    perror("Reading Block Failed!\n");
                    return -1;
                }
                // printf("读入数据块%d\n", r_right_bid);
            }
            if (r_right_bid > r_end)
            {
                A = 1 << 30;
                B = 1 << 30;
            }
            else
            {
                A = readNumFromBlk(r_blk, r_right_tid, 0);
                B = readNumFromBlk(r_blk, r_right_tid, 1);
            }
        }
        // 将符合条件的原则写入磁盘
        if(valid)
        {
            cnt++;
            memcpy(wblk + usage*8, s_blk + j*8, 8);
            usage++;
            // 判断输出缓冲块是否写满
            if(usage == 7)
            {
                itoa(1002+wnum, wblk + 56, 10);    // 后继块的地址
                if (writeBlockToDisk(wblk, 1001+wnum, buf) != 0)
                {
                    perror("Writing Block Failed!\n");
                    return -1;
                }
                printf("写回磁盘: %d\n", 1001+wnum);
                wblk = getNewBlockInBuffer(buf);
                memset(wblk, 0, 64);
                usage = 0;
                wnum++;
            }
        }
    }
    freeBlockInBuffer(s_blk, buf);
}
```

## 将输出缓冲区中剩余部分写入磁盘

```
if (usage != 0)
{
    itoa(1002+wnum, wblk + 56, 10);    // 后继块的地址
    if (writeBlockToDisk(wblk, 1001+wnum, buf) != 0)
    {
        perror("Writing Block Failed!\n");
        return -1;
    }
    memset(wblk, 0, 64);
    printf("写回磁盘: %d\n", 1001+wnum);
    wnum++;
}
printf("写磁盘次数: %d\n", wnum);
```

实验结果：


# 五、 附加题

*对剩余的两种集合操作进行问题分析，并给出程序正确运行的结果截图。*

**1. 集合的交操作**

**问题分析：遍历关系 S 的元组，将出现在 S 中，且出现在 R 中的元组(在 R 中存在元组同时满足 S.C = R.A, S.D = R.B)筛选出来，存放在磁盘上(2001.blk, 2002.blk, .....) 。**

**实验结果：**


**2. 集合的并操作**

**问题分析：选出出现在关系 S 或者关系 R 中的元组，去掉重复元组，将结**

**果放在磁盘上(3001.blk, 3002.blk, .....)。**

**实验结果：**