哈尔滨工业大学(深圳)

《数据库》实验报告

实验四

查询处理算法的模拟实现

学　院: 计算机科学与技术

姓　名: 王静茹

学　号: 210110831

专　业: 计算机科学与技术

日　期: 2023-11-05

一、 实验目的

阐述本次实验的目的。

理解索引的作用，掌握关系选择、连接、集合的交、并、差等操作的实现算法，加深对算法的 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：Windows 10

软件：Codeblocks

三、 实验内容

阐述本次实验的具体内容。

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[80, 160]，B 的值域为[200, 300]。

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[100, 200]，D 的值域为[220, 400]。

（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=107 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 = 107）。

（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 S 建立索引文件，利用索引文件选出 S.C=107 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 = 107）。

（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：实现并（S∪R）、交（S∩R）、差（S - R）其中一种集合操作算法，将结果存放在磁盘上，并统计并、交、差操作后的元组个数。

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

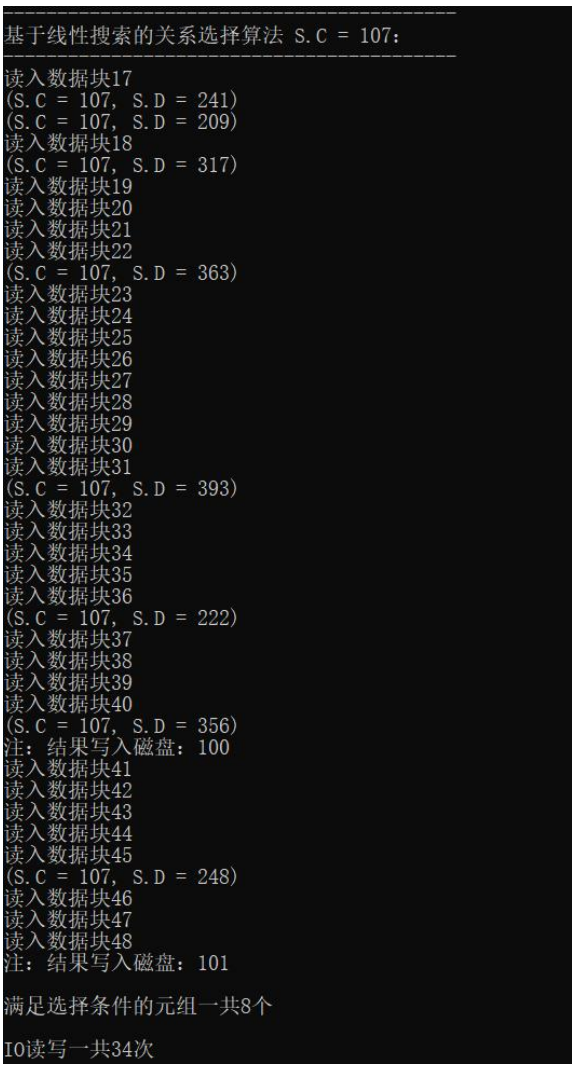
问题分析：所谓线性搜索，就是通过依次遍历每个磁盘块中的每个元组来查找符合要求的项。算法流程：首先，遍历关系 S 对应的磁盘块，用一个读内存块 blk_r 存放每次从磁盘中读出的数据块；然后，对于每一个数据块，遍历块中的每一个元组，获取其属性值（借助 atoi()函数）：如果属性 C 的值等于给定的 value，就打印该元组，并把它写入写内存块 blk_w 中（借助指针实现），同时令计数器 num 加 1（如果加 1 后 num 的值正好是 7 的倍数，就在向 blk_w 中写入后续要写的磁盘块的地址后把当前写内存块写入指定位置的磁盘块中，同时更新写地址；由于写入磁盘后原来的写内存块会被系统自动释放掉，所以我们还需要重新申请一个写内存块，并将这部分内存空间清零）。遍历完块中的所有元组后，通过读取当前数据块中所存的后继磁盘块地址可以获取我们下一个要遍历的磁盘块的块号，然后释放该读内存块即可。以上流程的具体实现如下图所示：

```
for(i = 0; i < BLKNUM_S; i++) {  //遍历关系S对应的磁盘块
    if((blk_r = readBlockFromDisk(addr_r, &buf)) == NULL) {
        perror("Reading Block Failed!\n");
        return -1;
    }
    printf("读入数据块%d\n", addr_r);
    for(j = 0; j < 7; j++) {  //遍历块中的每一个元组
        getValueFromTuple(blk_r, j, &SC, &SD);
        if(SC == value) {
            printf("(S.C = %d, S.D = %d)\n", SC, SD);
            writeTupleToBlock(blk_w, num%7, SC, SD);
            num++;
            if(num % 7 == 0) {  //内存块写满7个元组
                writeNextAddrToBlock(blk_w, addr_w+1);
                if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return -1;
                }
                printf("注：结果写入磁盘：%d\n", addr_w);
                addr_w++;
                blk_w = getNewBlockInBuffer(&buf);
                memset(blk_w, 0, BLKSIZE);
            }
        }
    }
    addr_r = getNextBlockAddr(blk_r);
    freeBlockInBuffer(blk_r, &buf);
}
```

除此之外，遍历完所有的磁盘块之后，我们还需要检查一下是否所有的选择结果都已经被写回磁盘了，如果不是，即计数器 num 的值不是 7 的倍数（代表最后写进 blk_w 的几个元组因为此时 blk_w 还没写满而还未被写回磁盘），就要在向 blk_w 中写入后续要写的磁盘块的地址后将该数据块写入磁盘指定位置；如果是，即在上一次将数据块写入磁盘之后还没有新的元组被写入 blk_w 中，那么直接将我们上一次写磁盘后新申请的写内存块释放掉即可。这部分的具体实现如下图所示：

```
if(num % 7 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
printf("\n满足选择条件的元组一共%d个\n\n", num);
printf("IO读写一共%ld次\n", buf.numIO);
```

实验结果：



(2) 实现两阶段多路归并排序算法（TPMMS）

问题分析：两阶段多路归并排序就是要求我们先把所有的数据块划分成多个子集合，使得每个子集合中的块数都小于内存缓冲区可用块数，然后将每个子集合依次装入缓冲区，采用内排序排好序后再重新写回磁盘，最后利用内存缓冲区对这 m 个已经排好序的子集合进行总的归并排序，得到最终结果。

具体流程：一、内排序：（我设定的是一个子集合中有 7 个数据块）首先，根据块总数以及子集合的大小计算出一共有 m 个子集合，遍历这 m 个子集合，确定每个子集合中的块数，同时记录下每个子集合排完序后将写入的磁盘块的首地址；然后，将每个子集合依次读入内存缓冲区中，并采用冒泡排序的方式对每个子集合进行内排序（如果某元组的指定属性值大于下一个元组的该属性值，就交换这两个元组的位置）；最后，将排好序的子集合写入外存指定位置，同时令

写地址增 1。这部分的具体实现如下图所示：

```
m = num % 7 == 0 ? num / 7 : num / 7 + 1;

for(i = 0; i < m; i++) {
    n = i < m-1 ? 7 : num - 7*(m-1);
    if(head != NULL) {
        head[i] = addr_w;
    }
    for(j = 0; j < n; j++) {
        if((blk_r[j] = readBlockFromDisk(addr_r, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        addr_r = getNextBlockAddr(blk_r[j]);
    }
    sortTuples(blk_r, para_no, n);  //内排序
    for(j = 0; j < n; j++) {
        writeNextAddrToBlock(blk_r[j], addr_w+1);
        if(writeBlockToDisk(blk_r[j], addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        addr_w++;
    }
}
```

（分组及磁盘读写）

```
void sortTuples(unsigned char** blk_array, int para_no, int n) {
    int para[3];
    unsigned char *blk1, *blk2;
    int i, j, flag;
    for(i = 0; i < 7*n - 1; i++) {
        flag = 0;
        for(j = 0; j < 7*n - i - 1; j++) {
            blk1 = blk_array[j/7];
            blk2 = blk_array[(j+1)/7];
            getValueFromTuple(blk1, j%7, &para[0], &para[1]);
            para[0] = para[para_no - 1];
            getValueFromTuple(blk2, (j+1)%7, &para[1], &para[2]);
            para[1] = para[para_no];
            if(para[0] > para[1]) {
                swap(blk1 + 8*(j%7), blk2 + 8*((j+1)%7));
                flag = 1;
            }
        }
        if(flag == 0) {
            break;
        }
    }
}
```

（内排序）

二、归并：首先，从磁盘中读出每个子集合的首个数据块，并将这些数据块中的第一个元组写入用于归并比较的内存块 blk_cmp 中（元组在 blk_cmp 中的位置与其所属的子集合的序号相对应）。然后，从 blk_cmp 中找出指定属性值最小的元组所在的位置 p，并将该元组写入写内存块 blk_w 中，同时令元组计数器 n 加 1（如果加 1 后 n 的值正好为 7，就在向 blk_w 中写入后续要写的磁盘块的地址后把当前写内存块写入指定位置的磁盘块中，并将 n 重新置为 0；由于写入磁盘后原来的写内存块会被系统自动释放掉，所以我们还需要重新申请一个写内存块，并将这部分内存空间清零）。之后，获取子集合 p 中当前正在归并的数据块内的下一个元组：如果当前数据块内有下一个元组，直接将其写入 blk_cmp 的第 p 行即可；否则，释放掉存储该数据块的读内存块，然后读入子集合 p 中的下一个数据块——如果子集合 p 中还有未归并的数据块（利用地址进行判断），就将

其读入内存，并将其中的第一个元组写入 blk_cmp 的第 p 行；如果没有，就将用于存储该子集合中数据块的读内存块指针置为空，并将 blk_cmp 的第 p 行置为全 0。只要 blk_cmp 中还有不全为 0 的元组，即程序还能在 blk_cmp 中找到指定属性的合法的最小值（找最小属性值时令 p 的初值为-1，min 的初值为 401，只有当我们找到某个属性值小于 min 且不为 0 的元组时，p 和 min 才会更新），就重复以上自初始化之后的所有步骤，直到 blk_cmp 中没有有效元组。这部分的具体实现如下图所示：

```
blk_cmp = getNewBlockInBuffer(&buf);
blk_w = getNewBlockInBuffer(&buf);
addr_w = disk_write;
for(i = 0; i < m; i++) {
    addr_r = head[i];
    if((blk_r[i] = readBlockFromDisk(addr_r, &buf)) == NULL) {
        perror("Reading Block Failed!\n");
        return -1;
    }
    memcpy(blk_cmp + 8*i, blk_r[i], 8);
    row[i] = 0;  //记录各子集合中当前数据块内正参与归并的元组
}
```

（初始化）

```
n = 0;
while((p = findMin(blk_cmp, para_no, m)) >= 0) {
    memcpy(blk_w + 8*n, blk_cmp + 8*p, 8);
    n++;
    if(n == 7) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
        n = 0;
    }
    row[p]++;
    if(row[p] < 7 && *(blk_r[p] + row[p]*8)) {
        memcpy(blk_cmp + p*8, blk_r[p] + row[p]*8, 8);
    }
    else {
        freeBlockInBuffer(blk_r[p], &buf);
        head[p]++;
        if((head[p] - disk_write - 100) % 7 != 0 && (head[p] - disk_write - 100) != num) {
            if((blk_r[p] = readBlockFromDisk(head[p], &buf)) == NULL) {
                perror("Reading Block Failed!\n");
                return -1;
            }
            memcpy(blk_cmp + 8*p, blk_r[p], 8);
            row[p] = 0;
        }
        else {  //下标p对应的子集合中没有下一磁盘块
            blk_r[p] = NULL;
            memset(blk_cmp + 8*p, 0, 8);
        }
    }
}
```

（归并过程；head[i]指向第 i 个子集合中正在归并的数据块的读地址，row[i]指向第 i 个子集合中正在归并的数据块内正要参与归并的元组）

归并结束之后，我们还需要检查一下是否所有的排序结果都已经被写回磁盘了，如果不是，即计数器 n 的值不为 0（代表最后写进 blk_w 的几个元组还未被写回磁盘），就要在向 blk_w 中写入后续要写的磁盘块的地址后将该数据块写入磁盘指定位置；如果是，即在上一次将数据块写入磁盘之后还没有新的元组被写入 blk_w 中，那么直接将我们上一次写磁盘后新申请的写内存块释放掉即可。这部分的具体实现如下图所示：

```
if(n != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

实验结果：

```
----------------------------------------
两阶段多路归并排序算法 R.A:
----------------------------------------
注：结果写入磁盘：301
注：结果写入磁盘：302
注：结果写入磁盘：303
注：结果写入磁盘：304
注：结果写入磁盘：305
注：结果写入磁盘：306
注：结果写入磁盘：307
注：结果写入磁盘：308
注：结果写入磁盘：309
注：结果写入磁盘：310
注：结果写入磁盘：311
注：结果写入磁盘：312
注：结果写入磁盘：313
注：结果写入磁盘：314
注：结果写入磁盘：315
注：结果写入磁盘：316
两阶段多路归并排序完毕！
```

```
----------------------------------------
两阶段多路归并排序算法 S.C:
----------------------------------------
注: 结果写入磁盘: 317
注: 结果写入磁盘: 318
注: 结果写入磁盘: 319
注: 结果写入磁盘: 320
注: 结果写入磁盘: 321
注: 结果写入磁盘: 322
注: 结果写入磁盘: 323
注: 结果写入磁盘: 324
注: 结果写入磁盘: 325
注: 结果写入磁盘: 326
注: 结果写入磁盘: 327
注: 结果写入磁盘: 328
注: 结果写入磁盘: 329
注: 结果写入磁盘: 330
注: 结果写入磁盘: 331
注: 结果写入磁盘: 332
注: 结果写入磁盘: 333
注: 结果写入磁盘: 334
注: 结果写入磁盘: 335
注: 结果写入磁盘: 336
注: 结果写入磁盘: 337
注: 结果写入磁盘: 338
注: 结果写入磁盘: 339
注: 结果写入磁盘: 340
注: 结果写入磁盘: 341
注: 结果写入磁盘: 342
注: 结果写入磁盘: 343
注: 结果写入磁盘: 344
注: 结果写入磁盘: 345
注: 结果写入磁盘: 346
注: 结果写入磁盘: 347
注: 结果写入磁盘: 348
两阶段多路归并排序完毕!
```

(3) 实现基于索引的关系选择算法

问题分析：对于基于索引的关系选择，我们首先要为待搜索的关系（已排序）建立起一个基于题目指定属性的块索引，然后在搜索时通过对索引字段值的研判来避免一些不必要的遍历。

具体流程：一、创建索引：首先，将关系的每个数据块依次读入内存缓冲区，然后，对于每个数据块，获取其中首个元组的属性值，并将指定属性的值与该数据块在磁盘中的地址拼接成一个元组（该元组即为当前数据块对应的索引），随后将该元组写入写内存块 blk_w 中，同样地，如果 blk_w 中已经写满 7 个元组，就在向 blk_w 中写入后续要写的磁盘块的地址后将该数据块写入磁盘指定位置，同时更新写地址与元组计数器，并重新申请一个写内存块。这部分的具体实现如下图所示：

```
for(i = 0; i < num; i++) {
    if((blk_r = readBlockFromDisk(addr_r, &buf)) == NULL) {
        perror("Reading Block Failed!\n");
        return -1;
    }
    getValueFromTuple(blk_r, 0, &para[0], &para[1]);
    writeTupleToBlock(blk_w, n, para[para_no - 1], addr_r);
    n++;
    if(n == 7) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：索引写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
        n = 0;
    }
    addr_r = getNextBlockAddr(blk_r);
    freeBlockInBuffer(blk_r, &buf);
}
```

索引建立完毕后，我们还需要检查一下是否所有的索引都已经被写回磁盘了，如果不是，即计数器 n 的值不为 0（代表最后写进 blk_w 的几个元组还未被写回磁盘），就要在向 blk_w 中写入后续要写的磁盘块的地址后将该数据块写入磁盘指定位置；如果是，即在上一次将数据块写入磁盘之后还没有新的元组被写入 blk_w 中，那么直接将我们上一次写磁盘后新申请的写内存块释放掉即可。这部分的具体实现如下图所示：

```
if(n != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：索引写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

二、基于索引的关系选择：首先，遍历索引：对于每个索引，我们首先要确定的是它所在的索引块，因为索引块是连续存储的，且一个磁盘块中可以存 7 个元组，所以这里的 i/7 就是当前应读入的索引块相较于起始索引块的块号偏移量，如此一来，我们很容易就能判断出当前的索引块地址 addr_idx 是否与我们当前正在考察的索引相对应，如果不是，就需要更新索引块地址，同时置 upd=1，以通知缓冲区读入新的索引块。然后，因为建立索引时所有的属性值已经按照从小到大排好了序，所以索引字段值其实就是该索引指向的数据块中指定属性的最小值，对此，我们先可以先以第一条索引的字段值及块号作为 idx_0 和 blk_0 的初值，然后从第二条索引开始，依次判断当前索引的字段值（idx）以及前一条索引的字段值（idx_0）与我们要查找的 value 的关系：如果 value 正好被夹在这两个索

引字段值的中间，就说明前一条索引指向的数据块（blk_0）中极大概率存在属性值为 value 的元组，那么我们就需要读入该索引指向的数据块（blk_0），遍历其中的每一个元组，获取其属性值：如果某元组属性 C 的值等于给定的 value，就打印该元组，并把它写入写内存块 blk_w 中，同时令计数器 num 加 1（如果加 1 后 num 的值正好是 7 的倍数，就在向 blk_w 中写入后续要写的磁盘块的地址后把当前写内存块写入指定位置的磁盘块中，同时更新写地址；由于写入磁盘后原来的写内存块会被系统自动释放掉，所以我们还需要重新申请一个写内存块，并将这部分内存空间清零）。此外，我们还应对当前索引的字段值（idx）进行一次判断：如果当前索引的字段值 idx 大于给定的 value，就说明从该索引指向的数据块开始，往后的数据块中的元组的属性值都将大于 value，那么从当前索引开始往后的索引以及它们对应的数据块其实都无需遍历，所以此时我们直接退出循环即可。反之，如果 idx 仍不大于 value，就更新 idx_0 和 blk_0 的值为当前索引的字段值及块号，然后重复以上步骤。

```
for(i = 0; i <= BLKNUM_S; i++) {  //遍历索引（有几个磁盘块就有几个索引）
    if(addr_idx != disk_idx + i/7) {
        addr_idx = disk_idx + i/7;
        upd = 1;
        freeBlockInBuffer(blk_idx, &buf);
    }
    if(upd == 1) {
        if((blk_idx = readBlockFromDisk(addr_idx, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        printf("读入索引块%d\n", addr_idx);
        upd = 0;
    }

    if(i == 0) {
        getValueFromTuple(blk_idx, i, &idx_0, &blk_0);
        continue;
    }

    if(i < BLKNUM_S) {
        getValueFromTuple(blk_idx, i, &idx, &blk);
    }
    else {
        idx = value + 1;  //额外遍历一次，以考察最后一个索引
    }

    if(idx_0 <= value && idx >= value) {
        addr_r = blk_0;
        if((blk_r = readBlockFromDisk(addr_r, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        printf("读入数据块%d\n", addr_r);
        for(j = 0; j < 7; j++) {  //遍历数据块中的每一个元组
            getValueFromTuple(blk_r, j, &SC, &SD);
            if(SC == value) {
                printf("(S.C = %d, S.D = %d)\n", SC, SD);
                writeTupleToBlock(blk_w, num%7, SC, SD);
                num++;
                if(num % 7 == 0) {  //内存块写满7个元组
                    writeNextAddrToBlock(blk_w, addr_w+1);
                    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
                        perror("Writing Block Failed!\n");
                        return -1;
                    }
                    printf("注：结果写入磁盘：%d\n", addr_w);
                    addr_w++;
                    blk_w = getNewBlockInBuffer(&buf);
                    memset(blk_w, 0, BLKSIZE);
                }
            }
        }
        freeBlockInBuffer(blk_r, &buf);
    }
    if(idx > value){
        break;
    }
    idx_0 = idx;
    blk_0 = blk;
}
```

最后，与上面的算法同理，我们还要对写入内存块的元组的个数进行一次检查，以确保所有的选择结果最终都被写回磁盘中。

```
if(num % 7 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
printf("\n满足选择条件的元组一共%d个\n\n", num);
printf("IO读写一共%ld次\n", buf.numIO);
```

实验结果：

```
-----------------------------------------
为关系S（已排序）中的属性C建立索引：
-----------------------------------------
注：索引写入磁盘：350
注：索引写入磁盘：351
注：索引写入磁盘：352
注：索引写入磁盘：353
注：索引写入磁盘：354
索引建立完毕！

-----------------------------------------
基于索引的关系选择算法 S.C = 107：
-----------------------------------------
读入索引块350
读入数据块319
(S.C = 107, S.D = 241)
读入数据块320
(S.C = 107, S.D = 209)
(S.C = 107, S.D = 317)
(S.C = 107, S.D = 363)
(S.C = 107, S.D = 393)
(S.C = 107, S.D = 222)
(S.C = 107, S.D = 356)
注：结果写入磁盘：120
(S.C = 107, S.D = 248)
注：结果写入磁盘：121

满足选择条件的元组一共8个

IO读写一共5次
```

(4) 实现基于排序的连接操作算法（Sort-Merge-Join）

问题分析：第一趟：基于连接属性对两个关系分别进行两阶段多路归并排序；第二趟：根据两个关系的连接属性的值的大小关系对 R 和 S 的两路输入进行处理（包括连接、指针后移、指针回移）。

具体流程：一、排序：

```
//对关系R进行排序
printf("对关系R进行归并排序（基于属性A）：\n");
TPMMS(R_START, BLKNUM_R, 1, disk_write + 100);
addr_r_R = disk_write + 100;

//对关系S进行排序
printf("对关系S进行归并排序（基于属性C）：\n");
TPMMS(S_START, BLKNUM_S, 1, disk_write + BLKNUM_R + 100);
addr_r_S = disk_write + BLKNUM_R + 100;
printf("\n");
```

二、两路归并连接：首先，分别读入关系 R 和 S 中的第一个数据块，接着，分别获取块中首个元组的属性值（RA，RB，SC，SD）；

```
while(SC <= 160) {
    if(upd_R == 1) {
        if((blk_r_R = readBlockFromDisk(addr_r_R, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_R = 0;
    }

    if(upd_S == 1) {
        if((blk_r_S = readBlockFromDisk(addr_r_S, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_S = 0;
    }

    getValueFromTuple(blk_r_R, t_R%7, &RA, &RB);  //t_R指向关系R中当前被读入的元组
    getValueFromTuple(blk_r_S, t_S%7, &SC, &SD);  //t_S指向关系S中当前被读入的元组
```

然后，判断 RA 和 SC 的关系：如果 RA 与 SC 相等，即满足连接的条件，就令连接次数 cnt 加 1，同时打印出连接后的元组，并将其写入写内存块 blk_w 中（因为连接后的元组有四个属性值，所以此时一个元组占两行，一个块中最多只能存三个元组，即六行），然后更新行计数器 n（如果 n 更新后正好是 6 的倍数，就把 blk_w 写入指定磁盘块中，然后重新申请一个写内存块，并将这部分内存空间清零）；此外，考虑到后继元组中的 RA 和 SC 仍有可能为当前值，所以只要我们是最近第一次遇到 RA 与 SC 相等的情况（即 flag=0），就先暂存关系 R 的元组指针 t_R，然后在处理完连接操作后只将 t_R 向后移动一次（t_R++），不改变 t_S，以考察当前的 SC 是否还能和其他 RA 构成相等关系，从而形成连接。还有一点需要注意的是，如果 t_R 本身已经指向了关系 R 中的最后一个元组，即 t_R++后正好等于 R 中元组的总数，那就说明与当前的 SC 相等的 RA 已经被全部遍历完了，所以此时我们需要恢复 t_R 的值，使其指向第一个与该 SC 相等的 RA，然后移动 t_S，继续让新的 SC 与其进行匹配。

```
if(RA == SC) {
    if(flag == 0) {
        temp = t_R;
        flag = 1;
    }
    cnt++;
    printf("(S.C = %d, S.D = %d, R.A = %d, R.B = %d)\n", SC, SD, RA, RB);
    writeTupleToBlock(blk_w, n%6, SC, SD);
    writeTupleToBlock(blk_w, ++n%6, RA, RB);
    n++;
    if(n % 6 == 0) {  //内存块写满6行（2行对应1个元组）
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_R++;
    if(t_R == 7*BLKNUM_R) {
        t_R = temp;  //恢复
        flag = 0;
        t_S++;
    }
}
```

如果 RA 小于 SC，直接向后移动元组指针 t_R 即可（所有元组是按连接属性

的值从小到大排列的）。

```
else if(RA < SC) {
    flag = 0;
    t_R++;
}
```

同理，如果 RA 大于 SC，直接向后移动元组指针 t_S 即可。这里需要注意的是，如果上一轮比较时 RA 与 SC 相等，即 flag 的值为 1，那么这一轮的 RA 大于 SC 就说明所有与当前的 SC 相等的 RA 都已经遍历完了，那此时除了要后移 S 的元组指针之外，还要恢复 R 的元组指针为之前的暂存值（即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组），只有这样，当新的 SC 的值与当前 SC 的值相等时，程序才会从第一个等于该值的 RA 开始，重新对 RA 和 SC 进行匹配，从而确保不会遗漏某些需要连接的情况。

```
else {
    if(flag == 1) {
        t_R = temp;  //恢复
        flag = 0;
    }
    t_S++;
}
```

最后，我们应根据 t_R 和 t_S 的最新值来确定我们接下来要读的磁盘块的地址(排完序后数据块连续存储，且一个磁盘块中可以存 7 个元组，则 t_R/7 和 t_S/7 就是当前应读入的 R/S 磁盘块相较于其起始磁盘块的块号偏移量）。如果读地址发生了更新，就置更新标志 upd 为 1（通知缓冲区读入新的数据块），同时释放掉原读内存块，然后重复以上所有步骤；如果新的读地址已经超过了该关系中的最后一个磁盘块的地址，就说明连接结束，退出循环即可。此外，如果当前的 SC 已经超过了 160（RA 的最大值），那就说明后续不会再产生新的连接了，所以直接结束循环即可。

```
if(addr_r_R != disk_write + 100 + t_R/7) {
    addr_r_R = disk_write + 100 + t_R/7;
    if(addr_r_R >= disk_write + 100 + BLKNUM_R) {
        break;
    }
    upd_R = 1;
    freeBlockInBuffer(blk_r_R, &buf);
}
if(addr_r_S != disk_write + 100 + BLKNUM_R + t_S/7) {
    addr_r_S = disk_write + 100 + BLKNUM_R + t_S/7;
    if(addr_r_S >= disk_write + 100 + BLKNUM_R + BLKNUM_S) {
        break;
    }
    upd_S = 1;
    freeBlockInBuffer(blk_r_S, &buf);
}
```

连接结束后，与上面的算法同理，我们还要对写入内存块的元组的个数进行一次检查，以确保所有的连接结果最终都被写回磁盘中。

```
if(n % 6 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

实验结果：

先排序：

```
----------------------------------------
基于排序的连接算法：
----------------------------------------
对关系R进行归并排序（基于属性A）：
注：结果写入磁盘：600
注：结果写入磁盘：601
注：结果写入磁盘：602
注：结果写入磁盘：603
注：结果写入磁盘：604
注：结果写入磁盘：605
注：结果写入磁盘：606
注：结果写入磁盘：607
注：结果写入磁盘：608
注：结果写入磁盘：609
注：结果写入磁盘：610
注：结果写入磁盘：611
注：结果写入磁盘：612
注：结果写入磁盘：613
注：结果写入磁盘：614
注：结果写入磁盘：615
两阶段多路归并排序完毕！
对关系S进行归并排序（基于属性C）：
注：结果写入磁盘：616
注：结果写入磁盘：617
注：结果写入磁盘：618
注：结果写入磁盘：619
注：结果写入磁盘：620
注：结果写入磁盘：621
注：结果写入磁盘：622
注：结果写入磁盘：623
注：结果写入磁盘：624
注：结果写入磁盘：625
注：结果写入磁盘：626
注：结果写入磁盘：627
注：结果写入磁盘：628
注：结果写入磁盘：629
注：结果写入磁盘：630
注：结果写入磁盘：631
注：结果写入磁盘：632
注：结果写入磁盘：633
注：结果写入磁盘：634
注：结果写入磁盘：635
注：结果写入磁盘：636
注：结果写入磁盘：637
注：结果写入磁盘：638
注：结果写入磁盘：639
注：结果写入磁盘：640
注：结果写入磁盘：641
注：结果写入磁盘：642
注：结果写入磁盘：643
注：结果写入磁盘：644
注：结果写入磁盘：645
注：结果写入磁盘：646
注：结果写入磁盘：647
两阶段多路归并排序完毕！
```

再连接：（太多了，只截首尾）（结果从 500.blk 开始写）

```
(S.C = 100, S.D = 300, R.A = 100, R.B = 206)
(S.C = 100, S.D = 338, R.A = 100, R.B = 206)
(S.C = 101, S.D = 327, R.A = 101, R.B = 252)
注：结果写入磁盘：500
(S.C = 101, S.D = 327, R.A = 101, R.B = 271)
(S.C = 101, S.D = 232, R.A = 101, R.B = 252)
(S.C = 101, S.D = 232, R.A = 101, R.B = 271)
注：结果写入磁盘：501
(S.C = 101, S.D = 252, R.A = 101, R.B = 252)
(S.C = 101, S.D = 252, R.A = 101, R.B = 271)
(S.C = 102, S.D = 221, R.A = 102, R.B = 221)
注：结果写入磁盘：502
(S.C = 102, S.D = 221, R.A = 102, R.B = 210)
(S.C = 102, S.D = 315, R.A = 102, R.B = 221)
(S.C = 102, S.D = 315, R.A = 102, R.B = 210)
注：结果写入磁盘：503
(S.C = 103, S.D = 321, R.A = 103, R.B = 227)
(S.C = 103, S.D = 377, R.A = 103, R.B = 227)
(S.C = 104, S.D = 269, R.A = 104, R.B = 281)
注：结果写入磁盘：504
(S.C = 104, S.D = 269, R.A = 104, R.B = 269)
(S.C = 104, S.D = 269, R.A = 104, R.B = 276)
(S.C = 104, S.D = 269, R.A = 104, R.B = 213)
注：结果写入磁盘：505
(S.C = 104, S.D = 326, R.A = 104, R.B = 281)
(S.C = 104, S.D = 326, R.A = 104, R.B = 269)
(S.C = 104, S.D = 326, R.A = 104, R.B = 276)
注：结果写入磁盘：506
(S.C = 104, S.D = 326, R.A = 104, R.B = 213)
(S.C = 104, S.D = 357, R.A = 104, R.B = 281)
(S.C = 104, S.D = 357, R.A = 104, R.B = 269)
注：结果写入磁盘：507
(S.C = 104, S.D = 357, R.A = 104, R.B = 276)
(S.C = 104, S.D = 357, R.A = 104, R.B = 213)
(S.C = 104, S.D = 334, R.A = 104, R.B = 281)
注：结果写入磁盘：508
```

……

```
(S.C = 140, S.D = 320, R.A = 140, R.B = 271)
(S.C = 141, S.D = 297, R.A = 141, R.B = 284)
(S.C = 141, S.D = 297, R.A = 141, R.B = 207)
注：结果写入磁盘：567
(S.C = 141, S.D = 314, R.A = 141, R.B = 284)
(S.C = 141, S.D = 314, R.A = 141, R.B = 207)
(S.C = 141, S.D = 248, R.A = 141, R.B = 284)
注：结果写入磁盘：568
(S.C = 141, S.D = 248, R.A = 141, R.B = 207)
(S.C = 142, S.D = 237, R.A = 142, R.B = 265)
(S.C = 142, S.D = 237, R.A = 142, R.B = 235)
注：结果写入磁盘：569
(S.C = 144, S.D = 290, R.A = 144, R.B = 290)
(S.C = 145, S.D = 325, R.A = 145, R.B = 210)
(S.C = 145, S.D = 325, R.A = 145, R.B = 212)
注：结果写入磁盘：570
(S.C = 145, S.D = 325, R.A = 145, R.B = 250)
(S.C = 145, S.D = 325, R.A = 145, R.B = 251)
(S.C = 147, S.D = 302, R.A = 147, R.B = 295)
注：结果写入磁盘：571
(S.C = 149, S.D = 279, R.A = 149, R.B = 256)
(S.C = 149, S.D = 279, R.A = 149, R.B = 236)
(S.C = 149, S.D = 285, R.A = 149, R.B = 256)
注：结果写入磁盘：572
(S.C = 149, S.D = 285, R.A = 149, R.B = 236)
(S.C = 151, S.D = 312, R.A = 151, R.B = 217)
(S.C = 152, S.D = 309, R.A = 152, R.B = 238)
注：结果写入磁盘：573
(S.C = 152, S.D = 298, R.A = 152, R.B = 238)
(S.C = 153, S.D = 271, R.A = 153, R.B = 251)
(S.C = 155, S.D = 361, R.A = 155, R.B = 264)
注：结果写入磁盘：574
(S.C = 155, S.D = 321, R.A = 155, R.B = 264)
(S.C = 155, S.D = 379, R.A = 155, R.B = 264)
(S.C = 155, S.D = 262, R.A = 155, R.B = 264)
注：结果写入磁盘：575
(S.C = 156, S.D = 252, R.A = 156, R.B = 293)
(S.C = 156, S.D = 252, R.A = 156, R.B = 232)
(S.C = 156, S.D = 252, R.A = 156, R.B = 233)
注：结果写入磁盘：576
(S.C = 157, S.D = 377, R.A = 157, R.B = 290)
(S.C = 157, S.D = 261, R.A = 157, R.B = 290)
(S.C = 159, S.D = 322, R.A = 159, R.B = 212)
注：结果写入磁盘：577
(S.C = 159, S.D = 322, R.A = 159, R.B = 228)
注：结果写入磁盘：578

总共连接235次
```

(5) 实现基于排序的两趟扫描算法，实现交、并、差其中一种集合操作算法

问题分析：（基于排序的交算法）第一趟：对两个关系分别进行两阶段多路归并排序；第二趟：根据两个关系中各个元组的属性值的大小关系对 R 和 S 的两路输入进行处理（包括交、后移、回溯）（交运算的基本思路：对于每个 S 元组，寻找与其完全匹配的 R 元组，如果能找到，就说明该 S 元组包含于关系 R，也就意味着该 S 元组在 R 和 S 的交集中）。

算法流程：一、排序：（关系 R 基于 A，关系 S 基于 C）

```
//对关系R进行排序
printf("对关系R进行归并排序（基于属性A）：\n");
TPMMS(R_START, BLKNUM_R, 1, disk_write + 100);
addr_r_R = disk_write + 100;

//对关系S进行排序
printf("对关系S进行归并排序（基于属性C）：\n");
TPMMS(S_START, BLKNUM_S, 1, disk_write + BLKNUM_R + 100);
addr_r_S = disk_write + BLKNUM_R + 100;
printf("\n");
```

二、交运算：首先，分别读入关系 R 和 S 中的第一个数据块，接着，分别获取块中首个元组的属性值（RA，RB，SC，SD）；

```
while(SC <= 160) {
    if(upd_R == 1) {
        if((blk_r_R = readBlockFromDisk(addr_r_R, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_R = 0;
    }

    if(upd_S == 1) {
        if((blk_r_S = readBlockFromDisk(addr_r_S, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_S = 0;
    }

    getValueFromTuple(blk_r_R, t_R%7, &RA, &RB);  //t_R指向关系R中当前被读入的元组
    getValueFromTuple(blk_r_S, t_S%7, &SC, &SD);  //t_S指向关系S中当前被读入的元组
```

然后，判断 RA 和 SC 的关系：如果 RA 与 SC 相等，且在上一轮比较中 RA 与 SC 不等，或者在上一轮中更新了 S 的元组指针 t_S（即 flag=0），就说明此时的 R 元组是关系 R 中第一个满足属性 A 的值与当前的 SC 相等的元组，那我们就应该先暂存关系 R 的元组指针 t_R，以备后续更新 SC 时回移指针；如果此时还有 RB 等于 SD，就说明当前正在考察的 S 元组包含于关系 R，那这个元组其实就是在 R 与 S 的交集中，所以我们需要将当前的 S 元组写入用于将交运算结果写回磁盘的内存块 blk_w 中，然后令结果计数器 num 加 1（自然，如果加 1 后 num 的值正好是 7 的倍数，就应在向 blk_w 中写入后续要写的磁盘块的地址后把当前写内存块写入指定位置的磁盘块中，同时更新写地址；由于写入磁盘后原来的写内

存块会被系统自动释放掉，所以我们还需要重新申请一个写内存块，并将这部分内存空间清零）。此外，这也昭示着对当前 S 元组的考察已经彻底结束，所以我们还需要后移 S 的元组指针 t_S，同时恢复 R 的元组指针为之前的暂存值（因为后续还要 t_R++，所以这里实际上是恢复成暂存值-1），即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组，以继续对下一个 S 元组进行考察。还有一点需要注意的是，如果 t_R 本身已经指向了关系 R 中的最后一个元组，且这一次的 R 元组还是不与 S 元组完全匹配，即 t_R++后正好等于 R 中元组的总数，那就说明与当前的 SC 相等的 RA 已经被全部遍历完了，却仍然没有找到与当前的 S 元组完全匹配的 R 元组，所以此时我们需要恢复 t_R 的值，使其指向第一个与该 SC 相等的 RA，然后移动 t_S，继续让新的 S 元组与其进行匹配。

```
if(RA == SC) {
    if(flag == 0) {
        temp = t_R;
        flag = 1;
    }
    if(RB == SD) {
        printf("(X = %d, Y = %d)\n", SC, SD);
        writeTupleToBlock(blk_w, num%7, SC, SD);
        num++;
        if(num % 7 == 0) {  //内存块写满7个元组
            writeNextAddrToBlock(blk_w, addr_w+1);
            if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
                perror("Writing Block Failed!\n");
                return -1;
            }
            printf("注：结果写入磁盘：%d\n", addr_w);
            addr_w++;
            blk_w = getNewBlockInBuffer(&buf);
            memset(blk_w, 0, BLKSIZE);
        }
        flag = 0;
        t_S++;
        t_R = temp - 1;
    }
    t_R++;
    if(t_R == 7*BLKNUM_R) {
        t_R = temp;  //恢复
        flag = 0;
        t_S++;
    }
}
```

如果 RA 小于 SC，直接向后移动元组指针 t_R 即可（因为当前的 R 元组肯定不和 S 元组匹配，也就意味着无法借助其判断 S 元组是否属于关系 R）。

```
else if(RA < SC) {
    flag = 0;
    t_R++;
}
```

同理，如果 RA 大于 SC，直接向后移动元组指针 t_S 即可（此时的 S 元组必然不属于关系 R）。这里需要注意的是，如果上一轮的比较结果是 RA 等于 SC，同时 RB 又不等于 SD，即 flag 的值为 1，那么这一轮的 RA 大于 SC 就说明所有与

当前的 SC 相等的 RA 都已经遍历完了，那此时除了要后移 S 的元组指针之外，还要恢复 R 的元组指针为之前的暂存值（即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组），只有这样，当新的 SC 的值与当前 SC 的值相等时，程序才会从第一个等于该值的 RA 开始，重新对 R 元组和 S 元组进行匹配，从而确保不会出现某些 S 元组明明包含于关系 R 却没有被检查出来的情况。

```
else {
    if(flag == 1) {
        t_R = temp;  //恢复
        flag = 0;
    }
    t_S++;
}
```

最后，我们应根据 t_R 和 t_S 的最新值来确定我们接下来要读的磁盘块的地址(排完序后数据块连续存储,且一个磁盘块中可以存 7 个元组,则 t_R/7 和 t_S/7 就是当前应读入的 R/S 磁盘块相较于其起始磁盘块的块号偏移量）。如果读地址发生了更新，就置更新标志 upd 为 1（通知缓冲区读入新的数据块），同时释放掉原读内存块，然后重复以上所有步骤；如果新的读地址已经超过了该关系中的最后一个磁盘块的地址，就说明连接结束，退出循环即可。此外，如果当前的 SC 已经超过了 160（RA 的最大值），就说明后续的 S 元组必然都不包含于关系 R，那么直接结束循环即可。

```
if(addr_r_R != disk_write + 100 + t_R/7) {
    addr_r_R = disk_write + 100 + t_R/7;
    if(addr_r_R >= disk_write + 100 + BLKNUM_R) {
        break;
    }
    upd_R = 1;
    freeBlockInBuffer(blk_r_R, &buf);
}
if(addr_r_S != disk_write + 100 + BLKNUM_R + t_S/7) {
    addr_r_S = disk_write + 100 + BLKNUM_R + t_S/7;
    if(addr_r_S >= disk_write + 100 + BLKNUM_R + BLKNUM_S) {
        break;
    }
    upd_S = 1;
    freeBlockInBuffer(blk_r_S, &buf);
}
```

交运算结束后，与上面的算法同理，我们还要对写入内存块的元组的个数进行一次检查，以确保所有的运算结果最终都被写回磁盘中。

```
if(num % 7 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

实验结果：

先排序：

```
----------------------------------------
基于排序的集合的交算法：
----------------------------------------
对关系R进行归并排序（基于属性A）：
注：结果写入磁盘：240
注：结果写入磁盘：241
注：结果写入磁盘：242
注：结果写入磁盘：243
注：结果写入磁盘：244
注：结果写入磁盘：245
注：结果写入磁盘：246
注：结果写入磁盘：247
注：结果写入磁盘：248
注：结果写入磁盘：249
注：结果写入磁盘：250
注：结果写入磁盘：251
注：结果写入磁盘：252
注：结果写入磁盘：253
注：结果写入磁盘：254
注：结果写入磁盘：255
两阶段多路归并排序完毕！
对关系S进行归并排序（基于属性C）：
注：结果写入磁盘：256
注：结果写入磁盘：257
注：结果写入磁盘：258
注：结果写入磁盘：259
注：结果写入磁盘：260
注：结果写入磁盘：261
注：结果写入磁盘：262
注：结果写入磁盘：263
注：结果写入磁盘：264
注：结果写入磁盘：265
注：结果写入磁盘：266
注：结果写入磁盘：267
注：结果写入磁盘：268
注：结果写入磁盘：269
注：结果写入磁盘：270
注：结果写入磁盘：271
注：结果写入磁盘：272
注：结果写入磁盘：273
注：结果写入磁盘：274
注：结果写入磁盘：275
注：结果写入磁盘：276
注：结果写入磁盘：277
注：结果写入磁盘：278
注：结果写入磁盘：279
注：结果写入磁盘：280
注：结果写入磁盘：281
注：结果写入磁盘：282
注：结果写入磁盘：283
注：结果写入磁盘：284
注：结果写入磁盘：285
注：结果写入磁盘：286
注：结果写入磁盘：287
两阶段多路归并排序完毕！
```

再做交运算：

```
(X = 101, Y = 252)
(X = 102, Y = 221)
(X = 104, Y = 269)
(X = 105, Y = 286)
(X = 105, Y = 241)
(X = 108, Y = 291)
(X = 116, Y = 275)
注：结果写入磁盘：140
(X = 119, Y = 259)
(X = 121, Y = 265)
(X = 130, Y = 232)
(X = 144, Y = 290)
注：结果写入磁盘：141

S和R的交集有11个元组
```

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.实现基于排序的两趟扫描算法，实现并操作算法

问题分析：（基于排序的并算法）第一趟：对两个关系分别进行两阶段多路归并排序；第二趟：根据两个关系中各个元组的属性值的大小关系对 R 和 S 的两路输入进行处理（包括并、指针后移、指针回移）（并运算的基本思路：以关系 R 中的元组作为 SUR 中的基本元组，不断向其中补充关系 S 中属性值不在关系 R 中的元组）。

算法流程：一、排序（关系 R 基于 A，关系 S 基于 C）：

```
//对关系R进行排序
printf("对关系R进行归并排序（基于属性A）：\n");
TPMMS(R_START, BLKNUM_R, 1, disk_write + 100);
addr_r_R = disk_write + 100;

//对关系S进行排序
printf("对关系S进行归并排序（基于属性C）：\n");
TPMMS(S_START, BLKNUM_S, 1, disk_write + BLKNUM_R + 100);
addr_r_S = disk_write + BLKNUM_R + 100;
printf("\n");
```

二、并运算：首先，分别读入关系 R 和 S 中的第一个数据块，接着，分别获取块中首个元组的属性值（RA，RB，SC，SD）；

```
while(1) {
    if(upd_R == 1) {
        if((blk_r_R = readBlockFromDisk(addr_r_R, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_R = 0;
    }

    if(upd_S == 1) {
        if((blk_r_S = readBlockFromDisk(addr_r_S, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_S = 0;
    }

    getValueFromTuple(blk_r_R, t_R%7, &RA, &RB);  //t_R指向关系R中当前被读入的元组
    getValueFromTuple(blk_r_S, t_S%7, &SC, &SD);  //t_S指向关系S中当前被读入的元组
```

然后，判断 RA 与 SC 的关系：如果 RA 与 SC 相等，且在上一轮比较中 RA 与 SC 不等，或者在上一轮中更新了 S 的元组指针 t_S（即 flag=0），就说明此时的 R 元组是关系 R 中第一个满足属性 A 的值与当前的 SC 相等的元组，那我们就应该先暂存关系 R 的元组指针 t_R，以备后续更新 SC 时回移指针使用；如果此时还有 RB 等于 SD，就说明当前正在考察的关系 S 的元组是包含在关系 R 中的，则向后移动 S 的元组指针 t_S，使其跳过该元组，同时恢复 R 的元组指针为之前的暂存值（因为后续还要 t_R++，所以这里实际上是恢复成暂存值-1），即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组，以继续检查下一个 S 元组是否被包含在关系 R 中。然后，不改变 t_S，只将 t_R 向后移动一次（t_R++），以考察当前的 S 元组是否能和其他的 R 元组构成相等关系，从而确定当前的 S 元组是否被包含在关系 R 中。还有一点需要注意的是，如果 t_R 本身已经指向了关系 R 中的最后一个元组，且这一次的 R 元组还是不与 S 元组完全匹配，即 t_R++后正好等于 R 中元组的总数，那就说明所有与当前的 SC 相等的 RA 都已经被遍历完了，且当前的 S 元组不包含于关系 R，所以此时我们需要恢复 t_R 的值，使其指向第一个与该 SC 相等的 RA，然后打印出当前的 S 元组，并将其写入用来将并运算结果写回磁盘的内存块 blk_w，同时移动 t_S，继续让新的 S 元组与这些 R 元组进行匹配。

```
if(RA == SC) {
    if(flag == 0) {
        temp = t_R;
        flag = 1;
    }
    if(RB == SD) {
        flag = 0;
        t_S++;
        t_R = temp - 1;
    }
    t_R++;
    if(t_R == 7*BLKNUM_R) {
        t_R = temp;  //恢复
        flag = 0;
        printf("(X = %d, Y = %d)\n", SC, SD);
        writeTupleToBlock(blk_w, num%7, SC, SD);
        num++;
        if(num % 7 == 0) {  //内存块写满7个元组
            writeNextAddrToBlock(blk_w, addr_w+1);
            if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
                perror("Writing Block Failed!\n");
                return -1;
            }
            printf("注：结果写入磁盘：%d\n", addr_w);
            addr_w++;
            blk_w = getNewBlockInBuffer(&buf);
            memset(blk_w, 0, BLKSIZE);
        }
        t_S++;
    }
}
```

如果 RA 小于 SC，就说明当前的 R 元组已经不会再被用来检验 S 元组是否包含于关系 R，即当前的 R 元组自此不会再被回溯，则直接将该 R 元组写入用来将运算结果写回磁盘的内存块 blk_w，同时令结果计数器 num 加 1 即可（自然，如果加 1 后 num 的值正好是 7 的倍数，就应在向 blk_w 中写入后续要写的磁盘块的地址后把当前写内存块写入指定位置的磁盘块中，同时更新写地址；由于写入磁盘后原来的写内存块会被系统自动释放掉，所以我们还需要重新申请一个写内存块，并将这部分内存空间清零）。之后，将关系 R 的元组指针 t_R 向后移动一次，以继续考察下一个 R 元组与当前 S 元组的匹配情况。

```
else if(RA < SC) {
    flag = 0;
    printf("(X = %d, Y = %d)\n", RA, RB);
    writeTupleToBlock(blk_w, num%7, RA, RB);
    num++;
    if(num % 7 == 0) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_R++;
}
```

如果 RA 大于 SC，就说明关系 R 中不存在一个元组与当前的 S 元组匹配（如果存在的话，早在匹配时 t_S 就会跳过当前的 S 元组，而 t_S 又不会回移，则该

S 元组根本不会出现在这里），则直接将该 S 元组写入用来将运算结果写回磁盘的内存块 blk_w，同时令结果计数器 num 加 1 即可（同样地，当 num 正好等于 7 的倍数时，需要将写内存块 blk_w 写回磁盘）。当然，我们也需要将 S 的元组指针 t_S 后移一项，以继续考察下一个 S 元组与当前 R 元组的匹配情况。此外，这里需要注意的是，如果上一轮的比较结果是 RA 等于 SC，同时 RB 又不等于 SD（即 flag=1），那么这一轮的 RA 大于 SC 就说明所有与当前的 SC 相等的 RA 都已经遍历完了，那此时除了要后移 S 的元组指针之外，还要恢复 R 的元组指针为之前的暂存值（即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组），只有这样，当新的 SC 的值与当前 SC 的值相等时，程序才会从第一个等于该值的 RA 开始，重新对 R 元组和 S 元组进行匹配，从而确保不会出现某些 S 元组明明包含于关系 R 却没有被检查出来的情况。

```
else {
    if(flag == 1) {
        t_R = temp;  //恢复
        flag = 0;
    }
    printf("(X = %d, Y = %d)\n", SC, SD);
    writeTupleToBlock(blk_w, num%7, SC, SD);
    num++;
    if(num % 7 == 0) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_S++;
}
```

最后，我们应根据 t_R 和 t_S 的最新值来确定我们接下来要读的磁盘块的地址（排完序后数据块连续存储，且一个磁盘块中可以存 7 个元组，则 t_R/7 和 t_S/7 就是当前应读入的 R/S 磁盘块相较于其起始磁盘块的块号偏移量）。如果读地址发生了更新，就置更新标志 upd 为 1（通知缓冲区读入新的数据块），同时释放掉原读内存块，然后重复以上所有步骤；如果新的读地址已经超过了该关系中的最后一个磁盘块的地址，就说明连接结束，退出循环即可。

```
if(addr_r_R != disk_write + 100 + t_R/7) {
    addr_r_R = disk_write + 100 + t_R/7;
    if(addr_r_R >= disk_write + 100 + BLKNUM_R) {
        break;
    }
    upd_R = 1;
    freeBlockInBuffer(blk_r_R, &buf);
}
if(addr_r_S != disk_write + 100 + BLKNUM_R + t_S/7) {
    addr_r_S = disk_write + 100 + BLKNUM_R + t_S/7;
    if(addr_r_S >= disk_write + 100 + BLKNUM_R + BLKNUM_S) {
        break;
    }
    upd_S = 1;
    freeBlockInBuffer(blk_r_S, &buf);
}
```

此外，当我们退出循环后，关系 R 或关系 S 中可能还有未被遍历到的元组，且按照我们之前的求并逻辑，此时剩余的这些元组必定不在 R 与 S 的交集中，那么我们直接将它们从原数据块中读出，然后利用写内存块 blk_w 把其全部写入磁盘指定位置即可（读/写逻辑与前面一致，这里不再赘述）。

```
while(t_R < 7 * BLKNUM_R) {
    if(upd_R == 1) {
        if((blk_r_R = readBlockFromDisk(addr_r_R, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_R = 0;
    }
    getValueFromTuple(blk_r_R, t_R%7, &RA, &RB);

    printf("(X = %d, Y = %d)\n", RA, RB);
    writeTupleToBlock(blk_w, num%7, RA, RB);
    num++;
    if(num % 7 == 0) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_R++;

    if(addr_r_R != disk_write + 100 + t_R/7) {
        addr_r_R = disk_write + 100 + t_R/7;
        if(addr_r_R >= disk_write + 100 + BLKNUM_R) {
            break;
        }
        upd_R = 1;
        freeBlockInBuffer(blk_r_R, &buf);
    }
}
```

（以处理关系 R 中的剩余元组为例，对关系 S 的处理与之完全一致）

并运算结束后，与上面的算法同理，我们还要对写入内存块的元组的个数进行一次检查，以确保所有的运算结果最终都被写回磁盘中。

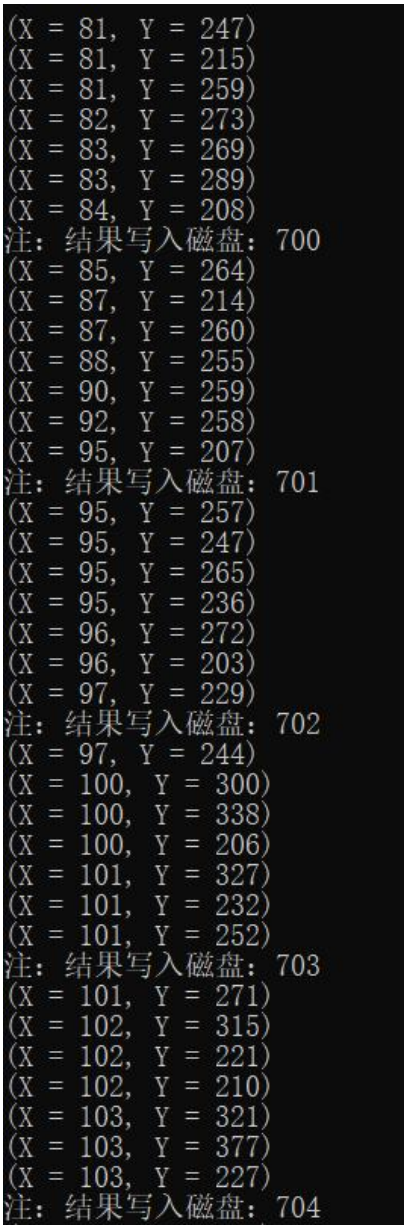
```
if(num % 7 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

实验结果：

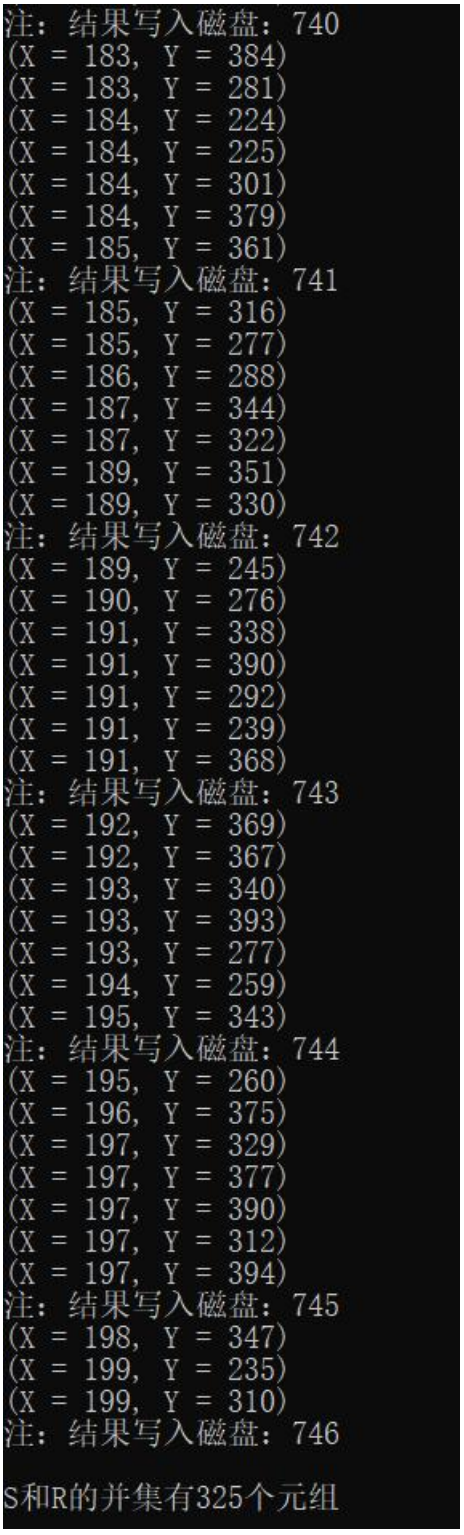
先排序：

```
----------------------------------------
基于排序的集合的并算法：
----------------------------------------
对关系R进行归并排序（基于属性A）：
注：结果写入磁盘：800
注：结果写入磁盘：801
注：结果写入磁盘：802
注：结果写入磁盘：803
注：结果写入磁盘：804
注：结果写入磁盘：805
注：结果写入磁盘：806
注：结果写入磁盘：807
注：结果写入磁盘：808
注：结果写入磁盘：809
注：结果写入磁盘：810
注：结果写入磁盘：811
注：结果写入磁盘：812
注：结果写入磁盘：813
注：结果写入磁盘：814
注：结果写入磁盘：815
两阶段多路归并排序完毕！
对关系S进行归并排序（基于属性C）：
注：结果写入磁盘：816
注：结果写入磁盘：817
注：结果写入磁盘：818
注：结果写入磁盘：819
注：结果写入磁盘：820
注：结果写入磁盘：821
注：结果写入磁盘：822
注：结果写入磁盘：823
注：结果写入磁盘：824
注：结果写入磁盘：825
注：结果写入磁盘：826
注：结果写入磁盘：827
注：结果写入磁盘：828
注：结果写入磁盘：829
注：结果写入磁盘：830
注：结果写入磁盘：831
注：结果写入磁盘：832
注：结果写入磁盘：833
注：结果写入磁盘：834
注：结果写入磁盘：835
注：结果写入磁盘：836
注：结果写入磁盘：837
注：结果写入磁盘：838
注：结果写入磁盘：839
注：结果写入磁盘：840
注：结果写入磁盘：841
注：结果写入磁盘：842
注：结果写入磁盘：843
注：结果写入磁盘：844
注：结果写入磁盘：845
注：结果写入磁盘：846
注：结果写入磁盘：847
两阶段多路归并排序完毕！
```

再做并运算：（太多了，只截首尾）（结果从 700.blk 开始写）


(X = 81, Y = 247)
(X = 81, Y = 215)
(X = 81, Y = 259)
(X = 82, Y = 273)
(X = 83, Y = 269)
(X = 83, Y = 289)
(X = 84, Y = 208)
注：结果写入磁盘：700
(X = 85, Y = 264)
(X = 87, Y = 214)
(X = 87, Y = 260)
(X = 88, Y = 255)
(X = 90, Y = 259)
(X = 92, Y = 258)
(X = 95, Y = 207)
注：结果写入磁盘：701
(X = 95, Y = 257)
(X = 95, Y = 247)
(X = 95, Y = 265)
(X = 95, Y = 236)
(X = 96, Y = 272)
(X = 96, Y = 203)
(X = 97, Y = 229)
注：结果写入磁盘：702
(X = 97, Y = 244)
(X = 100, Y = 300)
(X = 100, Y = 338)
(X = 100, Y = 206)
(X = 101, Y = 327)
(X = 101, Y = 232)
(X = 101, Y = 252)
注：结果写入磁盘：703
(X = 101, Y = 271)
(X = 102, Y = 315)
(X = 102, Y = 221)
(X = 102, Y = 210)
(X = 103, Y = 321)
(X = 103, Y = 377)
(X = 103, Y = 227)
注：结果写入磁盘：704

……



2.实现基于排序的两趟扫描算法，实现差操作算法

问题分析：（基于排序的差算法）第一趟：对两个关系分别进行两阶段多路归并排序；第二趟：根据两个关系中各个元组的属性值的大小关系对 R 和 S 的两路输入进行处理（包括并、指针后移、指针回移）（差运算的基本思路：寻找关

系 S 中不同时包含于关系 R 的元组【其实就相当于在进行并运算时不写回关系 R 中的元组，只写回用作补充的关系 S 中的元组】）。

算法流程：一、排序（关系 R 基于 A，关系 S 基于 C）：

```
//对关系R进行排序
printf("对关系R进行归并排序（基于属性A）：\n");
TPMMS(R_START, BLKNUM_R, 1, disk_write + 100);
addr_r_R = disk_write + 100;

//对关系S进行排序
printf("对关系S进行归并排序（基于属性C）：\n");
TPMMS(S_START, BLKNUM_S, 1, disk_write + BLKNUM_R + 100);
addr_r_S = disk_write + BLKNUM_R + 100;
printf("\n");
```

二、差运算：首先，分别读入关系 R 和 S 中的第一个数据块，接着，分别获取块中首个元组的属性值（RA，RB，SC，SD）；

```
SC = 0;
while(SC <= 160) {
    if(upd_R == 1) {
        if((blk_r_R = readBlockFromDisk(addr_r_R, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_R = 0;
    }

    if(upd_S == 1) {
        if((blk_r_S = readBlockFromDisk(addr_r_S, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_S = 0;
    }

    getValueFromTuple(blk_r_R, t_R%7, &RA, &RB);  //t_R指向关系R中当前被读入的元组
    getValueFromTuple(blk_r_S, t_S%7, &SC, &SD);  //t_S指向关系S中当前被读入的元组
```

然后，如果 RA 与 SC 相等，且在上一轮比较中 RA 与 SC 不等，或者在上一轮中更新了 S 的元组指针 t_S（即 flag=0），就说明此时的 R 元组是关系 R 中第一个满足属性 A 的值与当前的 SC 相等的元组，那我们就应该先暂存关系 R 的元组指针 t_R，以备后续更新 SC 时回移指针使用；如果此时还有 RB 等于 SD，就说明当前正在考察的 S 元组是包含在关系 R 中的，则向后移动 S 的元组指针 t_S，使其跳过该元组，同时恢复 R 的元组指针为之前的暂存值（因为后续还要 t_R++，所以这里实际上是恢复成暂存值-1），即使 t_R 回指向关系 R 中第一个满足 RA 等于当前 SC 的元组，以继续检查下一个 S 元组是否被包含在关系 R 中。然后，不改变 t_S，只将 t_R 向后移动一次（t_R++），以考察当前的 S 元组是否能和其他的 R 元组构成相等关系，从而确定当前的 S 元组是否被包含在关系 R 中。还有一点需要注意的是，如果 t_R 本身已经指向了关系 R 中的最后一个元组，且这一次的 R 元组还是不与 S 元组完全匹配，即 t_R++后正好等于 R 中元组的总数，那就说明所有与当前的 SC 相等的 RA 都已经被遍历完了，且当前的 S 元组不包含于

关系 R，所以此时我们需要恢复 t_R 的值，使其指向第一个与该 SC 相等的 RA，然后打印出当前的 S 元组，并将其写入用来将并运算结果写回磁盘的内存块 blk_w（当 num 正好等于 7 的倍数时，将 blk_w 写回磁盘），同时移动 t_S，继续让新的 S 元组与这些 R 元组进行匹配。

```
if(RA == SC) {
    if(flag == 0) {
        temp = t_R;
        flag = 1;
    }
    if(RB == SD) {
        flag = 0;
        t_S++;
        t_R = temp - 1;
    }
    t_R++;
    if(t_R == 7*BLKNUM_R) {
        t_R = temp;  //恢复
        flag = 0;
        printf("(X = %d, Y = %d)\n", SC, SD);
        writeTupleToBlock(blk_w, num%7, SC, SD);
        num++;
        if(num % 7 == 0) {  //内存块写满7个元组
            writeNextAddrToBlock(blk_w, addr_w+1);
            if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
                perror("Writing Block Failed!\n");
                return -1;
            }
            printf("注：结果写入磁盘：%d\n", addr_w);
            addr_w++;
            blk_w = getNewBlockInBuffer(&buf);
            memset(blk_w, 0, BLKSIZE);
        }
        t_S++;
    }
}
```

如果 RA 小于 SC，就说明当前的 R 元组无法被用来检验 S 元组是否包含于关系 R，那么我们直接将关系 R 的元组指针 t_R 向后移动一次即可。

```
else if(RA < SC) {
    flag = 0;
    t_R++;
}
```

如果 RA 大于 SC，就说明关系 R 中不存在一个元组与当前的 S 元组匹配（如果存在的话，早在匹配时 t_S 就会跳过当前的 S 元组），那么我们直接将该 S 元组写入用来将运算结果写回磁盘的内存块 blk_w，同时令结果计数器 num 加 1 即可（当 num 正好等于 7 的倍数时，需要将写内存块 blk_w 写回磁盘）。当然，我们还需要将 S 的元组指针 t_S 后移一项，以继续考察下一个 S 元组与当前 R 元组的匹配情况。此外，这里需要注意的是，如果上一轮的比较结果是 RA 等于 SC，同时 RB 又不等于 SD（即 flag=1），那么这一轮的 RA 大于 SC 就说明所有与当前的 SC 相等的 RA 都已经遍历完了，那此时除了要后移 S 的元组指针之外，还要恢复 R 的元组指针为之前的暂存值（即使 t_R 回指向关系 R 中第一个满足 RA 等于

当前 SC 的元组），只有这样，当新的 SC 的值与当前 SC 的值相等时，程序才会从第一个等于该值的 RA 开始，重新对 R 元组和 S 元组进行匹配，从而确保不会出现某些 S 元组明明包含于关系 R 却没有被检查出来的情况。

```
else {
    if(flag == 1) {
        t_R = temp;  //恢复
        flag = 0;
    }
    printf("(X = %d, Y = %d)\n", SC, SD);
    writeTupleToBlock(blk_w, num%7, SC, SD);
    num++;
    if(num % 7 == 0) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_S++;
}
```

最后，我们应根据 t_R 和 t_S 的最新值来确定我们接下来要读的磁盘块的地址(排完序后数据块连续存储,且一个磁盘块中可以存 7 个元组,则 t_R/7 和 t_S/7 就是当前应读入的 R/S 磁盘块相较于其起始磁盘块的块号偏移量）。如果读地址发生了更新，就置更新标志 upd 为 1（通知缓冲区读入新的数据块），同时释放掉原读内存块，然后重复以上所有步骤；如果新的读地址已经超过了该关系中的最后一个磁盘块的地址，就说明连接结束，退出循环即可。此外，如果当前的 SC 已经超过了 160（RA 的最大值），就说明后续的 S 元组必然都不包含于关系 R，那么直接结束循环即可。

```
if(addr_r_R != disk_write + 100 + t_R/7) {
    addr_r_R = disk_write + 100 + t_R/7;
    if(addr_r_R >= disk_write + 100 + BLKNUM_R) {
        break;
    }
    upd_R = 1;
    freeBlockInBuffer(blk_r_R, &buf);
}
if(addr_r_S != disk_write + 100 + BLKNUM_R + t_S/7) {
    addr_r_S = disk_write + 100 + BLKNUM_R + t_S/7;
    if(addr_r_S >= disk_write + 100 + BLKNUM_R + BLKNUM_S) {
        break;
    }
    upd_S = 1;
    freeBlockInBuffer(blk_r_S, &buf);
}
```

循环结束后，如果关系 S 中还有未被遍历到的元组，那么这些剩下的元组必然不包含于关系 R，所以我们还需要将这些 S 中的剩余元组从原数据块中读出，然后利用写内存块 blk_w 把它们全部写入磁盘指定位置（读/写逻辑与前面一致，这里不再赘述）。

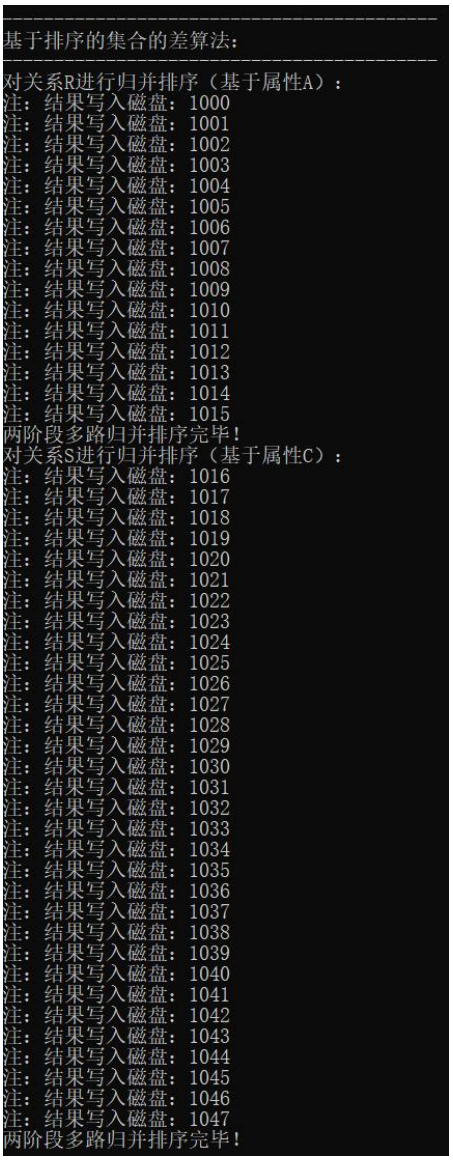
```
while(t_S < 7 * BLKNUM_S) {
    if(upd_S == 1) {
        if((blk_r_S = readBlockFromDisk(addr_r_S, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return -1;
        }
        upd_S = 0;
    }
    getValueFromTuple(blk_r_S, t_S%7, &SC, &SD);

    printf("(X = %d, Y = %d)\n", SC, SD);
    writeTupleToBlock(blk_w, num%7, SC, SD);
    num++;
    if(num % 7 == 0) {  //内存块写满7个元组
        writeNextAddrToBlock(blk_w, addr_w+1);
        if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return -1;
        }
        printf("注：结果写入磁盘：%d\n", addr_w);
        addr_w++;
        blk_w = getNewBlockInBuffer(&buf);
        memset(blk_w, 0, BLKSIZE);
    }
    t_S++;

    if(addr_r_S != disk_write + 100 + BLKNUM_R + t_S/7) {
        addr_r_S = disk_write + 100 + BLKNUM_R + t_S/7;
        if(addr_r_S >= disk_write + 100 + BLKNUM_R + BLKNUM_S) {
            break;
        }
        upd_S = 1;
        freeBlockInBuffer(blk_r_S, &buf);
    }
}
```

差运算结束后，与上面的算法同理，我们还要对写入内存块的元组的个数进行一次检查，以确保所有的运算结果最终都被写回磁盘中。

```
if(num % 7 != 0) {
    writeNextAddrToBlock(blk_w, addr_w+1);
    if(writeBlockToDisk(blk_w, addr_w, &buf) != 0) {
        perror("Writing Block Failed!\n");
        return -1;
    }
    printf("注：结果写入磁盘：%d\n", addr_w);
}
else {
    freeBlockInBuffer(blk_w, &buf);
}
```

实验结果：

先排序：



再做差运算：（太多了，只截首尾）（结果从 900.blk 开始写）

```
(X = 100, Y = 300)
(X = 100, Y = 338)
(X = 101, Y = 327)
(X = 101, Y = 232)
(X = 102, Y = 315)
(X = 103, Y = 321)
(X = 103, Y = 377)
注：结果写入磁盘：900
(X = 104, Y = 326)
(X = 104, Y = 357)
(X = 104, Y = 334)
(X = 104, Y = 286)
(X = 105, Y = 336)
(X = 105, Y = 271)
(X = 105, Y = 343)
注：结果写入磁盘：901
(X = 106, Y = 277)
(X = 107, Y = 241)
(X = 107, Y = 209)
(X = 107, Y = 317)
(X = 107, Y = 363)
(X = 107, Y = 393)
(X = 107, Y = 222)
注：结果写入磁盘：902
(X = 107, Y = 356)
(X = 107, Y = 248)
(X = 109, Y = 278)
(X = 109, Y = 262)
(X = 109, Y = 383)
(X = 109, Y = 331)
(X = 109, Y = 285)
注：结果写入磁盘：903
(X = 110, Y = 275)
(X = 111, Y = 352)
(X = 111, Y = 346)
(X = 111, Y = 317)
(X = 111, Y = 374)
(X = 112, Y = 398)
(X = 112, Y = 302)
注：结果写入磁盘：904
(X = 113, Y = 325)
(X = 113, Y = 347)
(X = 114, Y = 398)
(X = 115, Y = 327)
(X = 115, Y = 306)
(X = 116, Y = 365)
(X = 117, Y = 246)
注：结果写入磁盘：905
```

……

```
(X = 183, Y = 384)
(X = 183, Y = 281)
(X = 184, Y = 224)
(X = 184, Y = 225)
(X = 184, Y = 301)
(X = 184, Y = 379)
(X = 185, Y = 361)
注：结果写入磁盘：925
(X = 185, Y = 316)
(X = 185, Y = 277)
(X = 186, Y = 288)
(X = 187, Y = 344)
(X = 187, Y = 322)
(X = 189, Y = 351)
(X = 189, Y = 330)
注：结果写入磁盘：926
(X = 189, Y = 245)
(X = 190, Y = 276)
(X = 191, Y = 338)
(X = 191, Y = 390)
(X = 191, Y = 292)
(X = 191, Y = 239)
(X = 191, Y = 368)
注：结果写入磁盘：927
(X = 192, Y = 369)
(X = 192, Y = 367)
(X = 193, Y = 340)
(X = 193, Y = 393)
(X = 193, Y = 277)
(X = 194, Y = 259)
(X = 195, Y = 343)
注：结果写入磁盘：928
(X = 195, Y = 260)
(X = 196, Y = 375)
(X = 197, Y = 329)
(X = 197, Y = 377)
(X = 197, Y = 390)
(X = 197, Y = 312)
(X = 197, Y = 394)
注：结果写入磁盘：929
(X = 198, Y = 347)
(X = 199, Y = 235)
(X = 199, Y = 310)
注：结果写入磁盘：930

S和R的差集（S-R）有213个元组
```