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: 胡智胜

学　号: 180110518

专　业: 计算机科学与技术

日　期: 2021-04-23

目录

一、 实验目的

通过编写代码实现并优化数据库的常见操作，例如表的选择、集合的交并差等，充分理解索引的作用以及两趟扫描算法的实现细节，理解算法的 IO 复杂性。

二、 实验环境

Windows10 操作系统、CodeBlocks 17.12

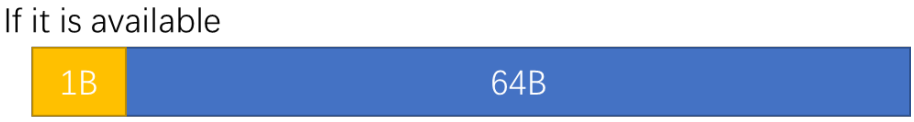
三、 实验内容

1. 数据库数据

本实验需要实现基于模拟数据的模拟数据库操作算法，其中模拟数据主要分为 R 表和 S 表，分布在工程文件的 data 文件夹中的 1.blk ~ 48.blk，其中 R 表占了前 16 个，S 表占了后 32 个.blk 文件。每个.blk 文件代表一个磁盘块（或缓冲区块），大小为 64 字节，其中前 56 字节可以用来存放数据，而末尾的 8 字节必须用来存放下一磁盘块的地址（磁盘块号）。

本实验中数据库中的一条记录（元组/item）大小为 8 字节，其实际是存放了 2 个 4 字节的代表整数的 ASCII 字符串，比如记录（45, 1328）在磁盘中存放实际上是（34,35,00,00,31,33,32,38）这 8 个字节的数据（16 进制）。

本实验的所有操作被限制在一个 520 字节的 Buffer 内，其中含有 8 个缓冲区块，占 512 字节。剩下的 8 个字节分配给了 8 个缓冲区块作为它们是否被占用的标志，如果为 1 则代表被占用，为 0 代表空闲。每个缓冲区块的是否被占用的标志在该块 64 字节的数据缓冲区的前一字节。



2. ExtMem 程序库

本实验的模拟磁盘操作由 ExtMem 程序提供，ExtMem 相当于提供了一套与模拟磁盘进行交互的接口函数，同时实验中的缓冲区的管理也是由 ExtMem 程序（extmem.c）进行管理。ExtMem 将缓冲区抽象成了 Buffer 结构体，通过缓冲区再去与磁盘进行交互，其中 Buffer 结构体中还记录有当前缓冲区的参数状态，例如剩余空闲缓冲块数量等。同时 ExtMem 程序还

会记录程序与磁盘进行 IO 操作的次数到 Buffer 中的 numIO 变量。

Buffer 中的变量及其含义：

numIO	外存 IO 次数
bufSize	缓冲区大小（单位：字节）
blkSize	块的大小（单位：字节）
numAllBlk	缓冲区可存放的最多块数
numFreeBlk	缓冲区内可用的块数
data	缓冲区内存区域

ExtMem 提供的 API 函数及其含义：

Buffer *initBuffer(size_t bufSize, size_t blkSize, Buffer *buf);	初始化 buf，设置其中的参数，并向内存申请缓冲区内存区域；
void freeBuffer(Buffer *buf);	释放 buf 的缓冲区内存区域；
unsigned char *getNewBlockInBuffer(Buffer *buf);	向缓冲区申请一个缓冲区块，返回该块的地址，如果缓冲区无空闲块，返回 NULL;
void freeBlockInBuffer(unsigned char *blk, Buffer *buf);	将缓冲区的一个块重新标记为 available；
int dropBlockOnDisk(unsigned int addr);	删除磁盘中块号为 addr 的磁盘块；
unsigned char *readBlockFromDisk(unsigned int addr, Buffer *buf);	从磁盘中读取指定块到缓冲区中，返回该块在缓冲区中的内存地址；
int writeBlockToDisk(unsigned char *blkPtr, unsigned int addr, Buffer *buf);	将缓冲区中的一块写到指定的磁盘块，并且将该缓冲区块标记为 available；

3. 实验任务

(1) 实现基于线性搜索的关系选择算法：从磁盘的 S 表（17.blk~48.blk）中选出 S.C=50 的元组，并将 S.C,S.D 记录到新的磁盘块中。

(2) 实现两阶段多路归并排序算法：基于 ExtMem 库提供的 Buffer，实现分别用两趟扫描将 R 表和 S 表的数据排好序后记录到新的磁盘块中。

(3) 实现基于索引的关系选择算法：首先为(2)中排好序的 S 表磁盘数据建立索引文件，将索引记录到新的磁盘块，然后基于刚建立的磁盘索引文件，去排好序的 S 表数据中选出 S.C=50 的元组，将 S.C,S.D 记录到新的磁盘块中。

(4) 实现基于排序的连接操作算法：依然是只用两趟扫描，实现将 R 表和 S 表的元组按照 R.A=S.C 的连接条件进行连接的算法，并统计连接次数，将结果写到新的磁盘块上。

(5) 实现基于排序或散列的集合交并差算法：依然是只用两趟扫描，实现 R∪S，R∩S，R-S 等常见集合操作算法，统计集合操作后的元组个数，将结果写到新的磁盘块上。

四、 实验过程

(0)本实验项目中建立的几个通用函数：

a) void translateBlock(char* block_buf)

该函数接收一个位于缓冲区的块的内存地址，该函数会在一个 64 字节的块的前 56 字节空间中，以 4 字节为单位，将每 4 字节的字符串通过 atoi 函数转换为 int 值再写回到该 4 字节的空间。整体上该函数就是将一个 ASCII 码表示数值的块转换为以二进制数据表示数值的块（块末尾的 8 字节不会被转换）。

建立该函数的原因主要就是字符串表示的数值不便于进行比较和交换等操作，因此转换成了 int 型数据后便可将块直接看作是一个 int 的数组，每个元组占两个 int 数据，便于进行后续排序等操作。

（此函数源代码已在报告末尾附录中给出）

b) void reverseTranslateBlock(char* block_buf)

该函数相当于是 translateBlock 的反函数，功能是将块的前 56 字节的数据，以 4 字节为单位，将每 4 字节的二进制数据读取到 int 变量，再通过 itoa 函数转换成字符串，再将该字符串的前 4 字节写回到该 4 字节的空间。（块末尾的 8 字节不会被转换）

建立该函数的原因就是将上述已经被 translate 的块再 re-translate 回以 ASCII 码表示的形式，再写回到磁盘，以满足实验要求。

（此函数源代码已在报告末尾附录中给出）

c) void writeNextBlockNum(char* block_buf, int block_no)

该函数将磁盘块号 block_no 由 int 转换为 char 字符串，再写到缓冲区块 block_buf 的末尾 8 字节的空间。

d) int getNextBlockNum(char* block_buf)

该函将缓冲区块的末尾 8 字节的字符串读取出来再调用 atoi 函数转为 int 类型，最后再返回该 int 数据。

e) void outputItem(int* item, int attr_count)

item 指针代表了一个元组，attr_count 代表的是一个元组有几个属性，在一般情况下我们的元组只有两个属性，比如 R 表和 S 表中的元组，但是如果是 R 表要和 S 表连接操作那么连接后的元组就是 4 个属性（此时 attr_count=4）。

该函数的作用就是将一个有着 attr_count 个属性的元组 item 写入到输出缓冲区中（以二进制数据直接写入），如果缓冲区满了，就会自动调用上面的 reverseTranslate 函数将所有的二进制数据转换成 ASCII 码的形式后再写到对应的输出磁盘块中，然后再到缓冲区中取下一块空闲块作为新的输出缓冲区。

f) void outputRefresh()

检查输出缓冲区是否有剩下的数据没写入到磁盘中，如果是就调用 reverseTranslate 函数转换成 ASCII 码后再写入到磁盘中。相当于是刷新当前输出缓冲区。

(1) 实现基于线性搜索的关系选择算法

a) 问题分析：

该问题中 S 表的元组是随机分布在磁盘块 17.blk 到 48.blk 中，需要实现算法去扫描这 32 个磁盘块，并从中获取到所有 S.C=50 的元组再记录到新的磁盘块

（结果保存在从 101.blk 开始的磁盘块中）

b) 解决方案

此问题的条件是已经知道要查找的表所在的磁盘块范围，因此直接遍历读取每个磁盘块到缓冲区，将每个磁盘块首先调用通用函数 translateBlock 将目前块中用 ASCII 码存储的元组全部翻译成二进制数据表示形式，接着我们再用一个 int 指针 ptr 指向该磁盘块的缓冲区内存地址。经过这样处理磁盘块后， 每个磁盘块中第 i 个元组的两个属性就是 ptr[i*2]和 ptr[i*2+1]，因此我们可以直接拿 ptr[i*2]和要搜索的属性值 target(50)进行比较，如果相等就调用通用函数 outputItem 将这个元组写入到输出缓冲区中即可。

c) 核心算法

```
void searchLinear(int beg_blk_no, int end_blk_no, int target){
    ...
    for(unsigned int i=beg_blk_no; i<=end_blk_no; i++){
        char* block_buf = readBlockFromDisk(i, &buf);
        printf("读入数据块%d\n", i);
        searchBlock(block_buf, target);
        freeBlockInBuffer(block_buf, &buf);
    }
    ...
}
```

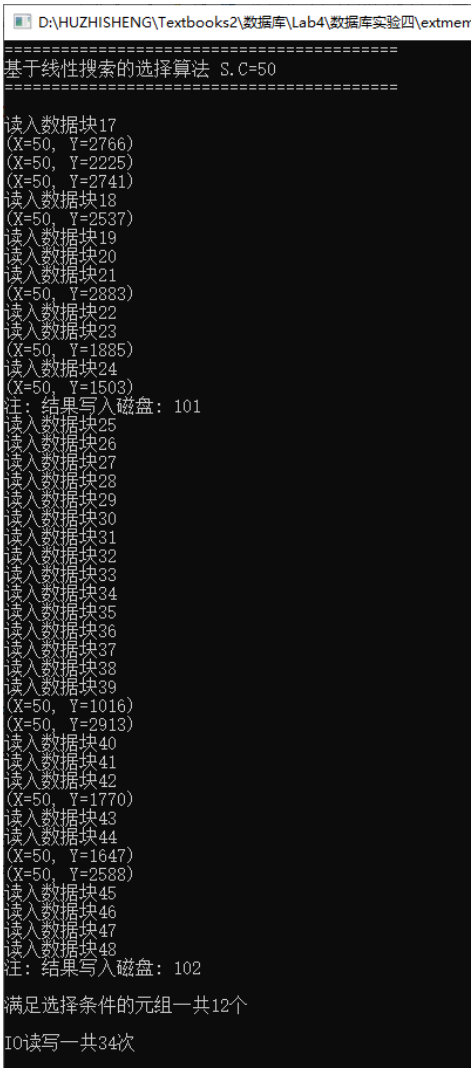
传入要搜索的磁盘块的范围以及目标属性值 target，接着 for 循环遍历读取每个 block 到缓冲区，接着对单个 buffer 中的 block 调用 searchBlock 函数；

```
static void searchBlock(char* block_buf, int target_value){
    translateBlock(block_buf);
    int* blk_int_ptr = (int*)block_buf;
    int item[2] = {0};
    for(int i=0; i<7; i++){
        if(blk_int_ptr[2*i] == target_value){
            item[0] = blk_int_ptr[2*i];
            item[1] = blk_int_ptr[2*i + 1];
            printf("(X=%d, Y=%d)\n", item[0], item[1]);
            outputItem(item, 2);
```

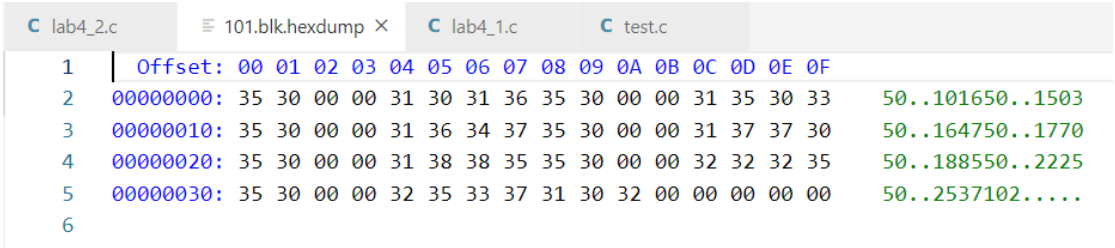
```
        }
    }
}
```

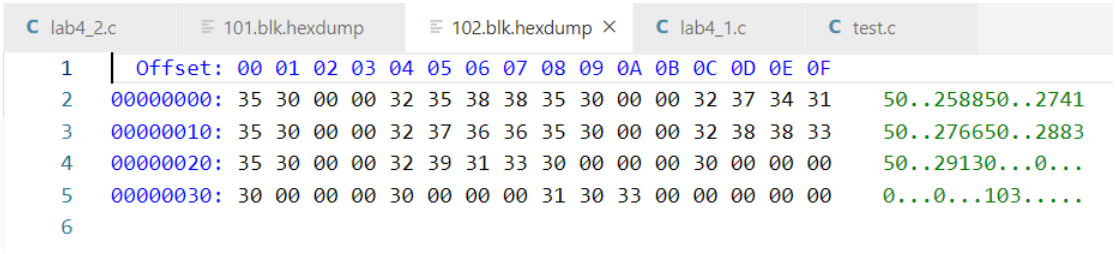
searchBlock 对每个块先调用 translateBlock 将数据从 ASCII 码表示转换成二进制，再用 int 指针指向块内存地址，再依次判断该 block 中的 7 个元组的第一个属性是否等于 target（50），如果相等则代表该元组的 S.C 等于 50，因此打印相关消息后再调用 outputItem 写入到输出缓冲区即可。

d) 实验结果：



使用 hexdump 插件查看输出的 101.blk 和 102.blk：





可以看见 12 条(50, y)的元组成功被找到并写入到 101.blk 和 102.blk 中。

(2) 实现两阶段多路归并排序算法（TPMMS）

a) 问题分析：

问题 2 需要我们实现一个两趟扫描的外排序算法。因为后面的问题也是采用的两趟扫描的基于排序的方法，因此问题 2 的代码是为后面的问题打下基础，需要具有一定的通用性。

b) 解决方案：

为了解决问题，将问题进一步拆分成以下子问题：

1. 实现第一遍扫描，将 8 个 block 划为一组，实现 8 个 block 在内存的排序算法

1.1 采用哪种内排序算法

采用冒泡排序；

1.2 如何得到缓冲中第 i 个元组（i 从 0 开始）

第 i 个元组所在的是第[i/7]个缓冲块，是该缓冲区块的第[i%7]个元组；

1.3 如何比较两个元组

先比较元组的第一个属性，第一个属性小的元组应该在前面；

如果第一个属性相同，再比较第二个属性，第二个属性小的应该在前面；

1.4 如何交换两个元组

通过前面 **translateBlock 函数**，此时缓冲区块中每 4 字节是存放的 int 类型数据；

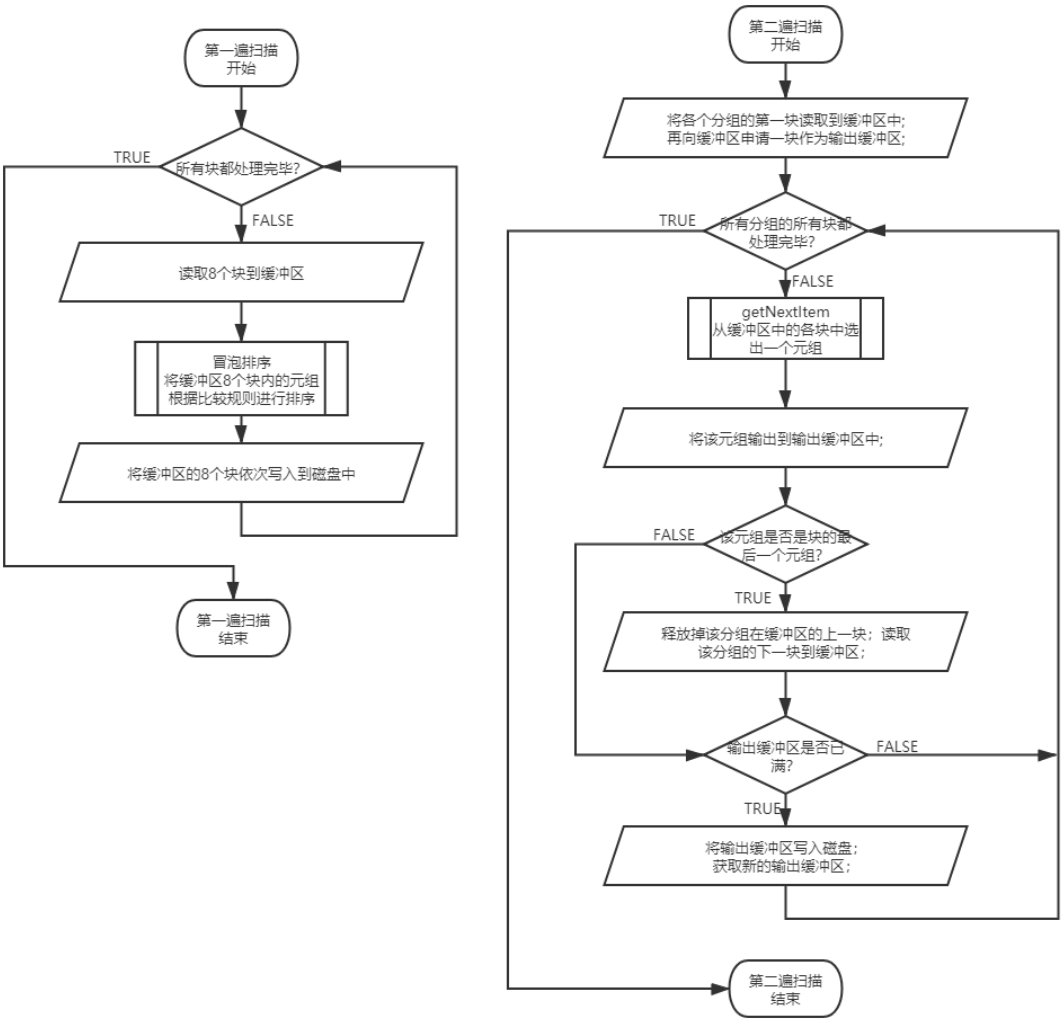
此时只需要将两个元组的 int 通过 int 类型的指针进行交换即可；

2. 实现第二遍扫描

2.1 如何得到下一个应输出到输出缓冲区的元组

通过 getNextItem 函数，该函数在缓冲区各组的第一块中根据上面的两元组比较规则去找寻一个排在最前面的元组 min_item，并记录这个最优先元组是来自哪个缓冲区块，将记录该缓冲区块剩余未处理的元组个数减 1，如果该缓冲区块中所有的元组都处理完毕，则还会根据该块末尾 8 字节的 next_block_num 去磁盘中读取下一个磁盘块到 Buffer 中。如果块末尾 8 字节的 next_block_num 为 0，就代表该块是该分组的最后一块（不存在下一块），因此该组之后便可直接忽略。

流程图如下：



c) 核心算法：

①第一趟扫描

```
// 将缓冲区的 8 个块进行排序，排好序后依次写到 TEMP_BLK 磁盘块，同时注意填写好块的最后 8 字节为下一块的块号
void sortItemInBuffer(){
    ...
    // 简单的冒泡排序
    for(int i=0; i < block_count * BLOCK_ITEM_NUM - 1; i++){    // 外循环为排序趟数，len 个数进行 len-1 趟
        for(int j=0; j < block_count * BLOCK_ITEM_NUM - 1 - i; j++){
            int* item1 = getItemPtr(j);
            int* item2 = getItemPtr(j+1);
            if(!compareItem(item1, item2)){    // 假设根据 Y 值排序
                swapItem(item1, item2);
            }
        }
```

```
    }
    // 排好序后，将缓冲区的数据写到磁盘的 temp 区域
    ...
}
```

冒泡排序，每次先调用 getItemPtr 函数从缓冲区中获取下标为 j 和 j+1 的两个元组；接着再调用 compateItem 函数对两个元组进行比较，看是否元组 1 和元组 2 需要进行交换，如果需要进行交换就调用 swapItem 函数进行交换。

这里之所以元组是用 int 指针指向的，是因为在将磁盘块读取到缓冲区时便调用了通用函数 **translateBlcok 函数**将块中的 ASCII 码表示的数据转换成了二进制的数据，因此这里直接用 int 指针指向原本用 char 指针指向的区域即可，通过 int 指针即可非常方便地获取到元组的各个属性。

```
// 得到缓冲区中的第 int_no 个元组的指针
int* getItemPtr(int int_no){
    char* byte_ptr = buf.data + 1;
    for(int i=0; i < int_no / BLOCK_ITEM_NUM; i++){
        byte_ptr += BLOCK_SIZE + 1;
    }
    byte_ptr += (int_no % BLOCK_ITEM_NUM) * ITEM_SIZE;
    return (int*)byte_ptr;
}
```

getItemPtr 首先从 Buffer 的 data 字段开始，先加 1 代表跳过第一块的 available 标志，然后因为第 i 个元组（i 从 0 开始）所在的块是[i/7]（下标从 0 开始），因此要跳过[i/7]块，然后第 i 个元组在块中是第[i%7]个元组（从 0 开始），所以其首地址要加上块的内存首地址加 i%7*8 字节。之后再将 char 指针强制转换为 int 指针即可返回。

②第二趟扫描

```
void sortItemInDisk(int beg_blk_no, int end_blk_no, int beg_res_blk_no){
    // 第二遍扫描排序
    int group_count = ceil((float)total_block_count / BUF_BLOCK_NUM);   //计算出分组数
    ...
    // 记录每组还剩多少元组没处理 -- 该数组长度为 4，未超过 10
    int* group_left_count = (int*)malloc(sizeof(int) * group_count);
    memset((void*)group_left_count, 0, sizeof(int) * group_count);
    for(int i=0; i<group_count; i++){
        group_left_count[i] = BLOCK_ITEM_NUM;
    }
    // 存放各组的缓冲区地址，如果一组已经处理完则对应的指针为 NULL -- 该数组长度为 4，未超
```

```
过 10
    char** group_blk_ptr = (char**)malloc(sizeof(char*) * group_count);
    for(int i=0; i<group_count; i++){
        group_blk_ptr[i] = readBlockFromDisk(temp_block_beg_no + i*BUF_BLOCK_NUM, &
buf);
        translateBlock(group_blk_ptr[i]);
    }

    // 将每组的第一块读取到缓冲区后开始进行第二遍扫描排序，类似弹夹打子弹的思想
    char* output_blk_ptr = getNewBlockInBuffer(&buf);    // 为输出缓冲区申请一个内存块
    int output_left_free = BLOCK_ITEM_NUM;               // 当前输出缓冲区还可写的空间
    int min_item[2];
    while(processed_block_count < total_block_count){
        if(output_left_free == 0){  // 输出缓冲区已满，必须将输出缓冲区刷新
            ...
        }
        // while 每循环一次输出缓冲区多一个元组(Item)
        // 找到关键值最小元组
        getNextItem(min_item, group_blk_ptr, group_left_count, group_count);

        // 将最小元组写到输出缓冲区，先不用按照 ASCII 码写，直接就按照值来写
        ...
        output_left_free -= 1;
    }
}
```

第二遍扫描主要就是

- 先计算分组的组数为 group_count；
- 然后申请一个长为 group_count 的 int 数组（缓冲区中最多就 8 块，因此分组的组数也一定小于 8，因此该数组长度不超过 10），该 int 数组记录的是各组当前在缓冲区中的块的剩余未处理的元组数量；
- 将各个分组的第一块读取到缓冲区（读取后会紧接着将块进行 translate 将 ASCII 码数据转换成二进制数据），将它们的在缓冲区的内存地址存储在 char 指针的数组 group_blk_ptr；
- 再申请输出缓冲区的块，并初始化相关参数；
- 接着就是进行循环，每次调用 getNextItem 从缓冲区中的各组中选出下一个要写入输出缓冲区的元组，再将输出缓冲区的相关参数进行更新即可；（每当输出缓冲区的空间满了便会在下次循环的开始时进行写入磁盘并申请新的缓冲区块的操作）

```
// 从第一遍扫描划分出的不同分组中得到下一个元组的指针
void getNextItem(int* item, char** group_blk_ptr, int* group_left_count, int group_
count){
    // 找到关键值最小元组
    int* min_item = NULL;
    int min_group = -1; //最小元组所在组号
    for(int i=0; i<group_count; i++){
        if(group_blk_ptr[i] == NULL){
            continue;
        }
        int* item_ptr = (int*)(group_blk_ptr[i]) + (BLOCK_ITEM_NUM - group_left_cou
nt[i])*2;
        // 假设按照(X, Y)中的 Y 属性排序
        if(min_item == NULL){
            min_item = item_ptr;
            min_group = i;
        }
        else if(compareItem(item_ptr, min_item)){
            min_item = item_ptr;
            min_group = i;
        }
    }
    // 在这里把 min_item 取到 item 中
    ...
    group_left_count[min_group] -= 1;
    if(group_left_count[min_group] == 0){   // 该分组的一块已处理完，换该分组的下一块
        ...
    }
}
```

遍历 group_blk_ptr 数组中的每个 char 指针，再根据 group_left_count 数组得知对应块剩下未处理的元组个数以得到该分组的下一个元组。

从各个分组的下一个元组中选出最优先的元组 min_item，将 min_item 对应的分组的块的 left_count 减 1，如果减为 0 就需要从磁盘中读取出该分组的下一块。

d) 实验结果：

```
D:\HUZHISHENG\Textbooks2\数据库\Lab4\数据库实验四\extmem

==========================================
两阶段多路归并排序算法
==========================================

关系R排序后输出到文件 201.blk 到 216.blk
注: 结果写入磁盘: 201
注: 结果写入磁盘: 202
注: 结果写入磁盘: 203
注: 结果写入磁盘: 204
注: 结果写入磁盘: 205
注: 结果写入磁盘: 206
注: 结果写入磁盘: 207
注: 结果写入磁盘: 208
注: 结果写入磁盘: 209
注: 结果写入磁盘: 210
注: 结果写入磁盘: 211
注: 结果写入磁盘: 212
注: 结果写入磁盘: 213
注: 结果写入磁盘: 214
注: 结果写入磁盘: 215
注: 结果写入磁盘: 216
关系S排序后输出到文件 217.blk 到 248.blk
注: 结果写入磁盘: 217
注: 结果写入磁盘: 218
注: 结果写入磁盘: 219
注: 结果写入磁盘: 220
注: 结果写入磁盘: 221
注: 结果写入磁盘: 222
注: 结果写入磁盘: 223
注: 结果写入磁盘: 224
注: 结果写入磁盘: 225
注: 结果写入磁盘: 226
注: 结果写入磁盘: 227
注: 结果写入磁盘: 228
注: 结果写入磁盘: 229
注: 结果写入磁盘: 230
注: 结果写入磁盘: 231
注: 结果写入磁盘: 232
注: 结果写入磁盘: 233
注: 结果写入磁盘: 234
注: 结果写入磁盘: 235
注: 结果写入磁盘: 236
注: 结果写入磁盘: 237
注: 结果写入磁盘: 238
注: 结果写入磁盘: 239
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

IO读写一共192次
```

C lab4_2.c M | C lab4_3.c M | C lab4_1.c M | C common.h M | 201.blk.hexdump ×

```
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 32 30 00 00 31 31 35 39 32 30 00 00 31 33 31 34    20..115920..1314
3  00000010: 32 30 00 00 31 39 33 30 32 32 00 00 31 30 38 31    20..193022..1081
4  00000020: 32 32 00 00 31 39 36 30 32 33 00 00 31 31 39 31    22..196023..1191
5  00000030: 32 33 00 00 31 33 39 36 32 30 32 00 00 00 00 00    23..1396202.....
6
```

C lab4_2.c M × | C lab4_3.c M | C lab4_1.c M | C common.h M | 201.blk.hexdump | 202.blk.hexdump ×

```
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 32 34 00 00 31 39 32 34 32 35 00 00 31 30 36 36    24..192425..1066
3  00000010: 32 35 00 00 31 31 33 32 32 36 00 00 31 31 36 36    25..113226..1166
4  00000020: 32 36 00 00 31 33 39 37 32 36 00 00 31 34 34 34    26..139726..1444
5  00000030: 32 36 00 00 31 34 39 31 32 30 33 00 00 00 00 00    26..1491203.....
6
```

(3) 实现基于索引的关系选择算法

a) 问题分析

解决该问题主要分为四大步骤：

1. 建立索引文件；
2. 在索引文件中查看要搜索的块号范围；
3. 根据块号范围去磁盘块中进行线性搜索；
4. 将搜索结果写到对应的磁盘块中；

b) 解决方案

1. 通过函数 createIndexFileForTable 建立索引文件：

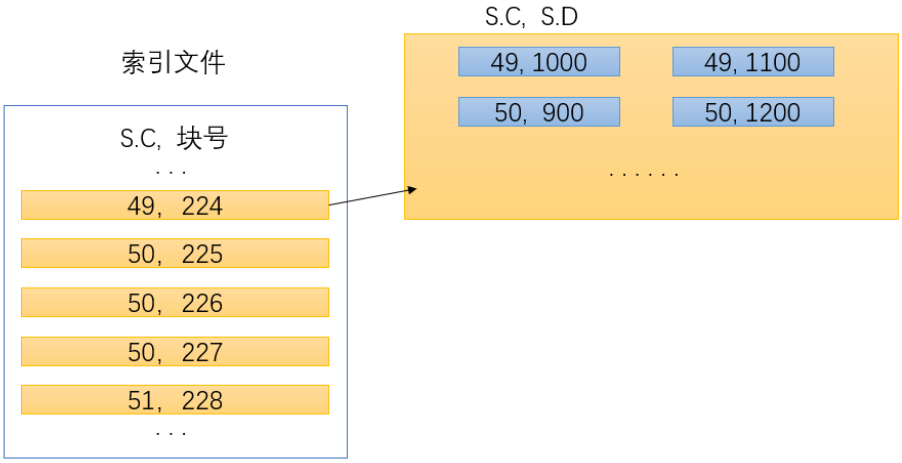
该函数会遍历问题 2 中已经排好序的 S 表磁盘块，将每个块的**第一个元组**的 S.C 属性和其块号一起组装成<S.C, 块号>的记录，该记录和元组相同的大小，因此我们可以直接按照之前写元组到输出缓冲区的类似的思路，将每个<S.C, 块号>记录写到索引文件的磁盘块中。

2. 通过函数 searchTableByIndex 在索引文件中查找关键属性所在的磁盘块范围

该函数会根据传入的 target 目标属性值，在已经建立的索引文件中查找该 target 所在的磁盘块范围。具体操作为：

- 假设 left 和 right 为最后的磁盘块范围，函数遍历每个索引文件磁盘块，遍历每一条<S.C, 块号 blk_no>记录：
  - 如果 S.C < target，则让 left = right = blk_no;
  - 如果 S.C == target，则让 right = blk_no;
  - 如果 S.C > target，则跳出循环；

  之所以不让 left 从 S.C == target 的块号开始，是因为避免出现以下问题：



很显然如果我们要查找 S.C=50 的元组，就应该去磁盘块号 224~227 的范围查找，因此为了保证不漏掉 224 这种第一个元组的 S.C 不等于 target 的情况，必须让 left 等于索引文件中第一个 S.C == target 的块号再减 1。

3. 在找出应查找的磁盘块号范围后就可直接复用问题(1)中的 searcheLinear 函数去磁盘范

围搜索 S.C == 50 的元组之后再结果写入到指定的磁盘块中即可。

c) 核心算法

```
void searchTableByIndex(int target, int beg_index_blk_no, int end_index_blk_no){
    int index_blk_no = beg_index_blk_no;
    int left_blk_no = -1;
    int right_blk_no = -1;

    while(1){
        printf("读入索引块%d\n",index_blk_no);
        char* blk_ptr = readBlockFromDisk(index_blk_no, &buf);
        int* int_ptr = (int*)blk_ptr;
        for(int i=0; i<BLOCK_ITEM_NUM; i++){
            if(int_ptr[0] > target){
                goto find_ok;   // 跳出 for 循环和 while 循环
            }else if(int_ptr[0] == target) {
                right_blk_no = int_ptr[1];
            }else{  // int_ptr[0] < target
                left_blk_no = int_ptr[1];
                right_blk_no = int_ptr[1];
            }
            int_ptr += 2;
        }
        freeBlockInBuffer(blk_ptr, &buf);
        if(index_blk_no == end_index_blk_no){
            break;
        }else{
            index_blk_no++;
        }
    }
find_ok:
     searchLinear(left_blk_no, right_blk_no, 50);
}
```

searchTableByIndex 函数用于在索引文件中查找 S.C==50 的元组在已排好序的 S 表的磁盘块范围[left,right]，其中索引文件所在的磁盘块号范围为[beg_index_blk_no, end_index_blk_no]。

函数具体操作就是遍历每个索引磁盘块，索引磁盘块中的数据并没有按照 ASCII 码表示因此我们不需要调用 translateBlock 函数对块进行翻译即可用 int 类型指针指向缓冲区中的块。接着遍历索引磁盘块中的各个<S.C, blk_no>记录，找出 S.C == 50 的元组可能分布的磁盘块范围[left,right]，最后再调用问题(1)中的 searchLiner 函数在[left,right]磁盘块中查找 S.C == 50 的元组。

d) 实验结果

```
========================================
基于索引的关系选择算法 S.C=50
========================================

读入索引块1101
读入索引块1102
读入数据块224
(X=50, Y=1016)
(X=50, Y=1503)
读入数据块225
(X=50, Y=1647)
(X=50, Y=1770)
(X=50, Y=1885)
(X=50, Y=2225)
(X=50, Y=2537)
注：结果写入磁盘：301
(X=50, Y=2588)
(X=50, Y=2741)
读入数据块226
(X=50, Y=2766)
(X=50, Y=2883)
(X=50, Y=2913)
注：结果写入磁盘：302

满足选择条件的元组一共12个

IO读写一共7次
```

索引磁盘块中的数据如下：

lab4_2.c | lab4_4.c | lab4_3.c | 1101.blk.hexdump ×

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 28 00 00 00 3D 01 00 00 29 00 00 00 3E 01 00 00    (...=...)...>...
00000010: 2B 00 00 00 3F 01 00 00 2C 00 00 00 40 01 00 00    +...?...,...@...
00000020: 2E 00 00 00 41 01 00 00 2E 00 00 00 42 01 00 00    ....A.......B...
00000030: 2F 00 00 00 43 01 00 00 00 00 00 00 00 00 00 00    /...C...........
```

（索引磁盘块中的记录并没有转换成 ASCII 码）

结果磁盘块中的数据如下：

301.blk.hexdump × | lab4_2.c M | lab4_3.c M | lab4_1.c M | common.h M | 201.blk.hexdump

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 35 30 00 00 31 30 31 36 35 30 00 00 31 35 30 33    50..101650..1503
00000010: 35 30 00 00 31 36 34 37 35 30 00 00 31 37 37 30    50..164750..1770
00000020: 35 30 00 00 31 38 38 35 35 30 00 00 32 32 32 35    50..188550..2225
00000030: 35 30 00 00 32 35 33 37 33 30 32 00 00 00 00 00    50..2537302.....
```

301.blk.hexdump | 302.blk.hexdump × | lab4_2.c M | lab4_3.c M | lab4_1.c M | common.h M

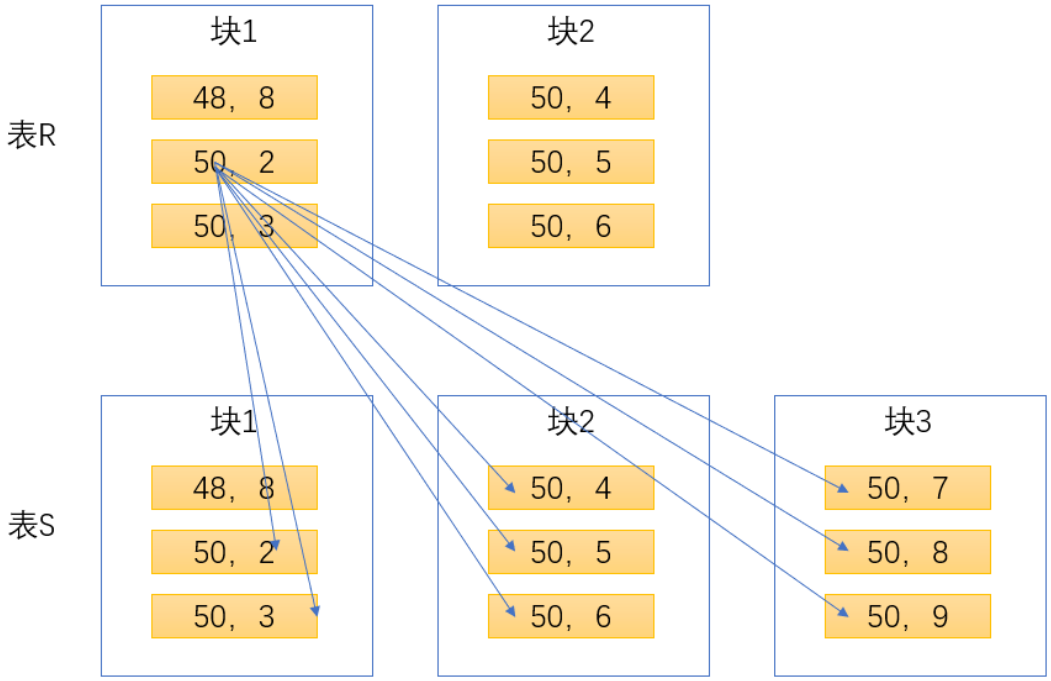
```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 35 30 00 00 32 35 38 38 35 30 00 00 32 37 34 31    50..258850..2741
00000010: 35 30 00 00 32 37 36 36 35 30 00 00 32 38 38 33    50..276650..2883
00000020: 35 30 00 00 32 39 31 33 30 00 00 00 30 00 00 00    50..29130...0...
00000030: 30 00 00 00 30 00 00 00 33 30 33 00 00 00 00 00    0...0...303.....
```

(4) 实现基于排序的连接操作算法（Sort-Merge-Join）

a) 问题分析

第一趟扫描和问题(2)中的处理完全相同，第一趟扫描完成后一共有 48/8=6 组，这样 6 组分

别占缓冲区中的一块，同时还要留一块给输出缓冲区，因此 Buffer 中空闲的块就只剩 1 块。第二趟扫描时需要依次从 R 表和 S 表中取出下一元组，如果它们的第一个属性值相同则可以进行连接操作，但需要注意的可能会出现如下情况：



假如表 R 和表 S 某一属性值的元组特别多，例如上图，那么就会出现表 R 中的<50,2>记录需要去表 S 的块 1 到块 3 中搜索一遍后，再换到表 R 的下一条记录<50,3>后再去搜索一遍表 S 的块 1 到块 3，但是我们的缓冲区只有 1 块空闲的，实现不重复读入表 S 的块 1 到块 3 的信息且能让连接结果不丢失的方案将在下面给出。

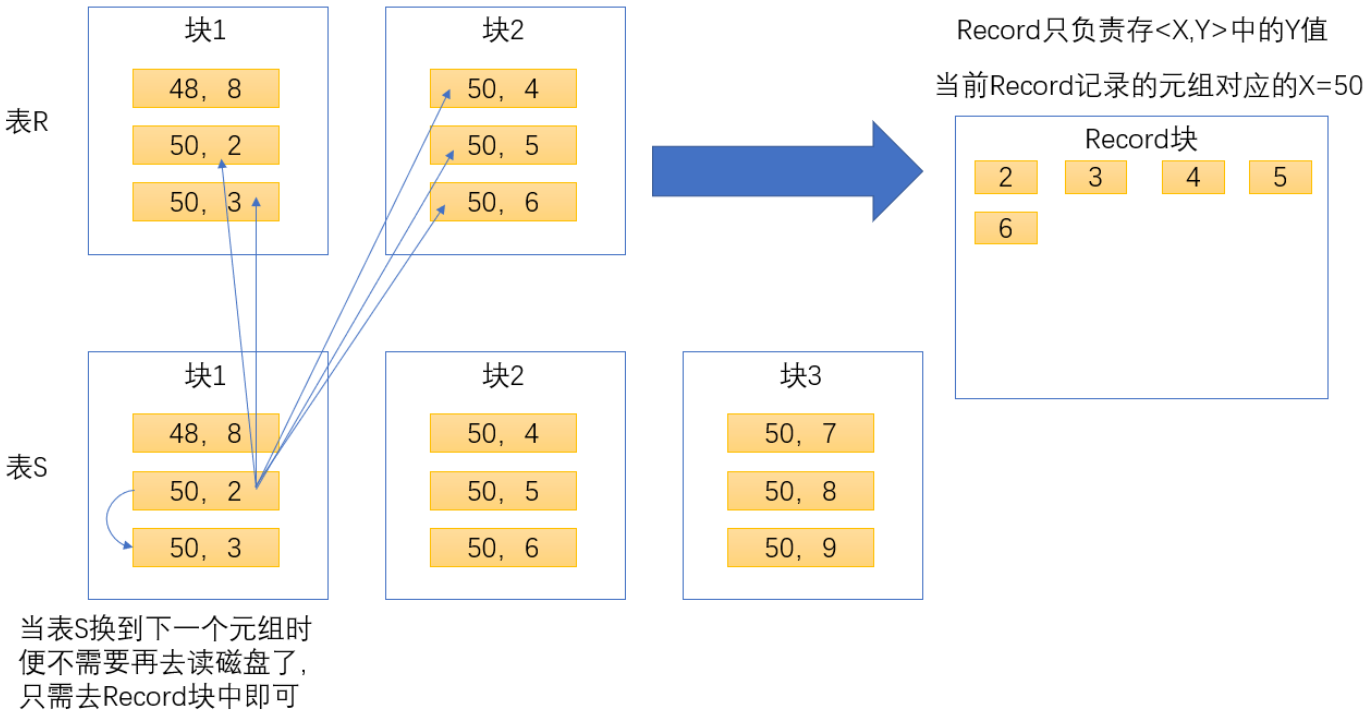
b) 解决方案

问题(4)的关键就在于如何保存已经读过的元组的信息，在本实验项目中将利用 Buffer 中剩下的最后一个空闲块作为 Record 块，一个块 64 字节，一个 int 占 4 字节，因此 Record 块最多可存 16 个 int 数据，因此就可以用 Record 保存某一特定 X 值的元组<X, Y>的所有 Y 值信息。由于 R 表的数据比 S 表的数据少，因此可以以 S 表的中的元组作为基准，将 R 表中的<X, Y>的 Y 值信息记录到 Record 中。

通过函数 getNextItem（在问题(2)的核心算法中已介绍）可以获取到表 R 或表 S 的下一个元组。每当获取到一个表 S 中的元组<X, Y>，将其 X 与上一个表 S 的 X 相比较，如果不同（代表是新的 X 值）则就去表 R 中通过 getNextItem 函数获取到所有当前 X 值的<X, Y>元组，并将所有的 Y 记录到 Record 块中，如果相同就代表与表 S 的<X,Y>具有相同 X 值的所有 R 表元组的 Y 值已经存到了 Record 块中，因此只需遍历 Record 块中的所有 Y 即可依次组装成连接元组<X,Y1,X,Y2>并将其输出到输出缓冲区中即可。当然如果表 R 中不含有当前 X 值的元组则代表无需进行连接操作，那么就再通过 getNextItem 函数获取到表 S 的下一个元组重复进行上面的循环。

需要说明的是虽然 Record 块只能存放 16 个 int 数据，但是经查看本实验数据的表 R 中并未出现相同 X 值的数据超过 16 个的情况，且选择表 R 的元组的 Y 值存放到 Record 块本身就已经考虑到表 R 的大小是表 S 的一半。（在实际数据库中缓冲区会有更多的空闲块可作为

Record 块，因此本算法还具有一定拓展性）



c） 核心算法

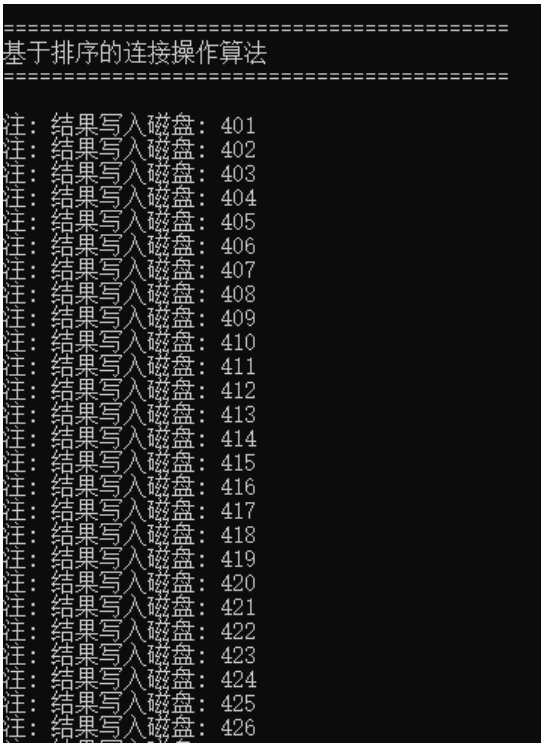
```
static void secondScan(){
    int group_count1 = ...   //计算出 R 表的分组数 -- 2
    int group_count2 = ...   //计算出 S 表的分组数 -- 4
    int processed_block_count = 0;  // 将已处理的块数重置

    // 记录 R 表每组还剩多少元组没处理 -- 该数组长度为 2，未超过 10
    int* group_left_count1 = ...
    // 记录 S 表每组还剩多少元组没处理 -- 该数组长度为 4，未超过 10
    int* group_left_count2 = ...

    // 存放 R 表各组的缓冲区地址，如果一组已经处理完则对应的指针为 NULL -- 该数组长度为 2，未超过 10
    char** group_blk_ptr1 = ...
    // 存放 S 表各组的缓冲区地址，如果一组已经处理完则对应的指针为 NULL -- 该数组长度为 4，未超过 10
    char** group_blk_ptr2 = ...
    // 第二遍扫描排序开始
    int item1[2];
    int item2[2];
    int next_item1[2] = {0, 0};
```

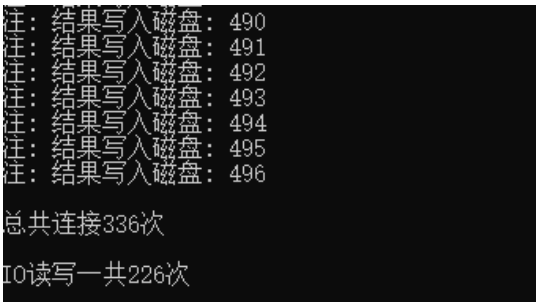
```
    getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1);  // R 表的当前选出来元组
    getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2);  // S 表的当前选出来元组
    while(item1[0] && item2[0]){   // 连接操作 -- 当 R 表或 S 表任何一个表扫描完时结束循环
        // 因为 S 表数量多于 R 表，以 S 表为主
        // 遇到一条 S 表中的记录(a, b)就去 R 表中查找所有(a, y)数据记录将 y 保存到 record 块中，并连接后输出到输出块中，再取 S 表中的下一条记录
        if(item1[0] < item2[0]){
            if(next_item1[0] > item1[0]){
                item1[0] = next_item1[0];
                item1[1] = next_item1[1];
            }else{
                getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1);
            }
        }else if(item1[0] > item2[0]){
            getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2);
        }else{
            if(item1[0] != record_now_x){   // 取出 R 表中所有的当前 x 值对应的所有 y 值，并存到 record 块中
                recordRefresh();
                record_now_x = item1[0];
                while(item1[0] && item1[0] == item2[0]){
                    recordAttr(item1[1]);
                    getNextItem(next_item1, group_blk_ptr1, group_left_count1, group_count1);
                    if(next_item1[0] == item1[0]){
                        item1[0] = next_item1[0];
                        item1[1] = next_item1[1];
                    }else{
                        break;
                    }
                }
            }
            outputJoin(item2[1]);
            getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2);
        }
    }
    ...
}
```

secondScan 函数就是负责两趟扫描中的第二趟扫描，其中的主要思想已经在解决方案中介绍，需要强调的是其中的每个块在读取到缓冲区后都立即调用了 translateBlock 函数将其中的 ASCII 码表示的数据转换成了二进制数据，因此这里可以直接用 int 指针指向缓冲区中的元组。

上面的代码中 record_now_x 变量就是记录当前 Record 块中存储的元组的 X 值，而 recordAttr 函数就是将一个元组的 Y 值记录到 Record 块中，outputJoin 函数就是接受当前 S 表中元组<X,Y1>的 Y1 值，然后拿去和 Record 块中每个记录的 Y2 值组装成<X,Y2,X,Y1>的元组再输出到输出缓冲区中。（R 表元组在前，S 表元组在后）

d) 实验结果





查看输出磁盘块 401.blk：

lab4_4.c | 401.blk.hexdump × | lab4_5_intersect.c | lab4_2.c M | lab4_3.c M | lab4_1.c M

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 34 30 00 00 31 32 34 33 34 30 00 00 31 32 34 33    40..124340..1243
00000010: 34 30 00 00 31 33 33 39 34 30 00 00 31 32 34 33    40..133940..1243
00000020: 34 30 00 00 31 37 30 35 34 30 00 00 31 32 34 33    40..170540..1243
00000030: 34 30 00 00 32 33 38 35 34 30 32 00 00 00 00 00    40..2385402.....
```

可以看到前面的 3 个连接后的元组是:
<40, 1243, 40, 1243>, <40, 1339, 40, 1243>, <40, 1705, 40, 1243>

(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

a) 问题分析

本问题解决基于排序实现交集的算法，要解决的主要问题就是如何在第一趟扫描的基础上去实现两个表的交集操作。

b) 解决方案

解决方案大体上和问题(4)相似。第一趟扫描先按照 8 块一组去进行内排序，使每个组有序。接着第二趟扫描的开始部分也和问题(4)中方案相似，不同的是：

初始化时先通过 getNextItem 函数分别从表 R 和表 S 选出下一个元组<X1, Y1>和<X2, Y2>，每次循环时比较这两个元组：

- 如果 X1<X2 则从表 R 中取出下一个元组；
- 相反如果 X1>X2 则从表 S 取出下一个元组；
- 如果 X1==X2，则再看 Y1 和 Y2：
  - 如果 Y1<Y2，则从表 R 中取出下一个元组；
  - 如果 Y1>Y2，则从表 S 中取出下一个元组；
  - 如果 Y1==Y2，就将<X1, Y1>(<X2, Y2>也行)输出到输出缓冲区中，之后再从表 R 和表 S 中各自取出下一个元组；

需要说明的是，取出下一个元组就是通过调用 getNextItem 函数，该函数已经在问题(2)中介绍，其中比较缓冲区中各个块的元组的方法是通过调用 compareItem 函数进行，这个函数优先比较元组的 X 值，X 值小的优先，如果 X 值相同再去比较 Y 值，Y 值小的优先，因此通过 getNextItem 函数取出的元组不仅是按照 X 值排序的，且相同 X 值的元组是按照 Y 值排序的，因此按照以上方法依次调用 getNextItem 函数不会出现交集漏掉元组的情况。

c) 核心算法

```
// beg_res_blk_no 代表存放结果的开始磁盘块号
static void secondScan(){
    ... //计算出 R 表的分组数 -- 2
    ... //计算出 S 表的分组数 -- 4
    ... // 将已处理的块数重置

    // 记录 R 表每组还剩多少元组没处理 -- 该数组长度为 2, 未超过 10
    ...
```

```
    // 记录 S 表每组还剩多少元组没处理 -- 该数组长度为 4，未超过 10
    ...

    // 存放 R 表各组的缓冲区地址，如果一组已经处理完则对应的指针为 NULL -- 该数组长度为 2，未超过 10
    ...
    // 存放 S 表各组的缓冲区地址，如果一组已经处理完则对应的指针为 NULL -- 该数组长度为 4，未超过 10
    ...
    // 第二遍扫描排序开始
    int item1[2];
    int item2[2];
    getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1);  // R 表的当前选出来元组
    getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2);  // S 表的当前选出来元组
    while(item1[0] && item2[0]){   // 连接操作 -- 当 R 表或 S 表任何一个表扫描完时结束循环
        // 因为 S 表数量多于 R 表，以 S 表为主
        if(item1[0] < item2[0]){
            getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1);
        }else if(item1[0] > item2[0]){
            getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2);
        }else{
            if(item1[1] < item2[1]){
                getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1)
;
            }else if(item1[1] > item2[1]){
                getNextItem(item2, group_blk_ptr2, group_left_count2, group_count2)
;
            }else{  // item1 和 item2 完全相同
                outputItem(item1, 2);
                getNextItem(item1, group_blk_ptr1, group_left_count1, group_count1)
;
            }
        }
    }
    ...
}
```

secondScan 函数负责第二趟扫描的工作，其中的 while 循环进行的就是上述解决方案中的操作，getNextItem 负责从表 R 或表 S 的缓冲区中取出下一个元组，outputItem 就是将指定的元组输出到输出缓冲区，当输出缓冲区满时会自动写到磁盘中。

d) 实验结果

```
==========================================
集合交集算法
==========================================

注：结果写入磁盘：501
注：结果写入磁盘：502
注：结果写入磁盘：503

R和S的交集有15个元素

IO读写一共133次
```

使用 hexdump 查看 501.blk、502.blk、503.blk：

lab4_4.c | lab4_5_intersect.c | 501.blk.hexdump × | 502.blk.hexdump | 503.blk.hexdump | lab4_2.

```
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 34 30 00 00 31 32 34 33 34 31 00 00 31 38 32 39   40..124341..1829
3  00000010: 34 32 00 00 31 34 32 32 34 32 00 00 31 35 34 34   42..142242..1544
4  00000020: 34 33 00 00 31 31 37 31 34 34 00 00 31 34 30 31   43..117144..1401
5  00000030: 34 35 00 00 31 32 30 33 35 30 32 00 00 00 00 00   45..1203502.....
6
```

lab4_4.c | lab4_5_intersect.c | 501.blk.hexdump | 502.blk.hexdump × | 503.blk.hexdump | C

```
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 34 37 00 00 31 39 32 31 34 38 00 00 31 37 30 39   47..192148..1709
3  00000010: 35 31 00 00 31 33 30 35 35 31 00 00 31 38 38 37   51..130551..1887
4  00000020: 35 32 00 00 31 33 30 36 35 34 00 00 31 38 33 34   52..130654..1834
5  00000030: 35 38 00 00 31 32 35 36 35 30 33 00 00 00 00 00   58..1256503.....
6
```

```
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 35 38 00 00 31 37 34 37 30 00 00 00 30 00 00 00   58..17470...0...
3  00000010: 30 00 00 00 30 00 00 00 30 00 00 00 30 00 00 00   0...0...0...0...
4  00000020: 30 00 00 00 30 00 00 00 30 00 00 00 30 00 00 00   0...0...0...0...
5  00000030: 30 00 00 00 30 00 00 00 35 30 34 00 00 00 00 00   0...0...504.....
6
```

和老师已经公布的结果完全符合：

2. 实验数据经过修改，两个关系一共有15个相同的元组。为方便大家调试，给出其中5个相同的元组：(40,1243),(41,1829),(42,1422),(58,1256),(58,1747)。

五、 附加题

(6) 实现基于排序的两趟扫描算法，实现并集算法

a) 解决方案：

实现并集操作，只需要在问题(5)的解决方案的基础上修改为：

初始化时先通过 getNextItem 函数分别从表 R 和表 S 选出下一个元组<X1,Y1>和<X2,Y2>，每次循环时比较这两个元组：

- 如果 X1<X2 则先输出<X1,Y1>，接着从表 R 中取出下一个元组到<X1,Y1>中；
- 如果 X1>X2 则先输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；
- 如果 X1==X2，则再看 Y1 和 Y2：
  - 如果 Y1<Y2，则先输出<X1,Y1>，接着从表 R 中取出下一个元组到<X1,Y1>中；
  - 如果 Y1>Y2，则先输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；
  - 如果 Y1==Y2，就将<X1,Y1>(<X2,Y2>也行)输出到输出缓冲区中，之后再从表 R 和表 S 中各自取出下一个元组；

循环结束后，还需要判断表 R 中的元素是否都已经写入到输出缓冲区中，如果没有则：

- 输出<X1,Y1>，接着从表 R 中取出下一个元组到<X1,Y1>中；

循环结束后，还需要判断表 S 中的元素是否都已经写入到输出缓冲区中，如果没有则：

- 输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；

b) 实验结果：

```
==========================================
集合并集算法
==========================================

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: 645
注: 结果写入磁盘: 646

R和S的并集有321个元素

IO读写一共190次
```

使用 hexdump 查看 601.blk:

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 32 30 00 00 31 31 35 39 32 30 00 00 31 33 31 34    20..115920..1314
00000010: 32 30 00 00 31 39 33 30 32 32 00 00 31 30 38 31    20..193022..1081
00000020: 32 32 00 00 31 39 36 30 32 33 00 00 31 31 39 31    22..196023..1191
00000030: 32 33 00 00 31 33 39 36 36 30 32 00 00 00 00 00    23..1396602.....
```

前三个元组是<20,1159>,<20,1314>,<20,1930>

(7) 实现基于排序的两趟扫描算法，实现差集算法

a) 解决方案：

实现并集操作，只需要在问题(5)的解决方案的基础上修改为：

初始化时先通过 getNextItem 函数分别从表 R 和表 S 选出下一个元组<X1,Y1>和<X2,Y2>，每次循环时比较这两个元组：

- 如果 X1<X2 则直接从表 R 中取出下一个元组到<X1,Y1>中；
- 如果 X1>X2 则先输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；
- 如果 X1==X2，则再看 Y1 和 Y2：
  - 如果 Y1<Y2，则直接从表 R 中取出下一个元组到<X1,Y1>中；
  - 如果 Y1>Y2，则先输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；
  - 如果 Y1==Y2，则直接从表 R 和表 S 中各自取出下一个元组；

循环结束后，还需要判断表 S 中的元素是否都已经写入到输出缓冲区中，如果没有则：

- 输出<X2,Y2>，接着从表 S 中取出下一个元组到<X2,Y2>中；

（相当于就是将并集算法中涉及到 R 表元组的输出语句全部删除即可）

b) 实验结果：

```
========================================
集合差集算法
========================================

注: 结果写入磁盘: 701
注: 结果写入磁盘: 702
注: 结果写入磁盘: 703
注: 结果写入磁盘: 704
注: 结果写入磁盘: 705
注: 结果写入磁盘: 706
注: 结果写入磁盘: 707
注: 结果写入磁盘: 708
注: 结果写入磁盘: 709
注: 结果写入磁盘: 710
注: 结果写入磁盘: 711
注: 结果写入磁盘: 712
注: 结果写入磁盘: 713
注: 结果写入磁盘: 714
注: 结果写入磁盘: 715
注: 结果写入磁盘: 716
注: 结果写入磁盘: 717
注: 结果写入磁盘: 718
注: 结果写入磁盘: 719
注: 结果写入磁盘: 720
注: 结果写入磁盘: 721
注: 结果写入磁盘: 722
注: 结果写入磁盘: 723
注: 结果写入磁盘: 724
注: 结果写入磁盘: 725
注: 结果写入磁盘: 726
注: 结果写入磁盘: 727
注: 结果写入磁盘: 728
注: 结果写入磁盘: 729
注: 结果写入磁盘: 730

R和S的差集(S-R)有209个元素

IO读写一共174次
```

使用 hexdump 查看 701.blk：

lab4_4.c | lab4_5_intersect.c | lab4_5_union.c | lab4_5_minus.c | 701.blk.hexdump × | lab4_

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 34 30 00 00 31 33 33 39 34 30 00 00 31 37 30 35    40..133940..1705
00000010: 34 30 00 00 32 33 38 35 34 31 00 00 31 32 36 39    40..238541..1269
00000020: 34 31 00 00 32 34 32 37 34 31 00 00 32 36 30 39    41..242741..2609
00000030: 34 31 00 00 32 39 30 39 37 30 32 00 00 00 00 00    41..2909702.....
```

前三个元组是<40, 1339>, <40, 1705>, <40, 2385>

需要说明的是差集(S-R)有 209 个元素，交集(S∩R)有 15 个元素，209+15=224 刚好等于 S 表的元组个数，可以进一步提高结果的可信度。

同时并集(S∪R)有 321 个元组，刚好等于

$$表S的元组个数 + 表R元组个数 - 交集(S \cap R)的元组个数$$

即 224+112-15=321, 基本上可以证明结果正确。

六、 总结

本次实验需要我们手动实现数据库中的常见操作的算法，例如搜索、连接、集合的交并差等，使我们对数据库底层原理有了更深的理解，也明白了数据库原理这门课的重要性。我们在今后的学习工作中也应更加注意自己的代码高效性，结合数据库底层原理的知识去使自己的代码对数据库的操作更加快速更加安全。

附录 –data 文件夹分布说明

磁盘块号范围	说明
[1, 16]	表 R 的原始数据
[17, 48]	表 S 的原始数据
[101, 102]	问题(1)的 SELECT 结果
[201, 248]	问题(2)第二趟扫描的结果
[301, 302]	问题(3)的 SELECT 结果
[401, 496]	问题(4)第二趟扫描的连接结果
[501, 503]	问题(5)第二趟扫描 R 和 S 的交集结果
[601, 646]	问题(6)第二趟扫描 R 和 S 的并集结果
[701, 730]	问题(7)第二趟扫描差集(S-R)的结果
[1001, 1048]	问题(2)第一趟扫描的结果
[1101, 1105]	问题(3)创建的索引文件磁盘块
[1201, 1248]	问题(4)第一趟扫描的结果
[1301, 1348]	问题(5)第一趟扫描的结果
[1401, 1448]	问题(6)第一趟扫描的结果
[1501, 1548]	问题(7)第一趟扫描的结果

附录 – 重要函数源代码

void translateBlock(char* block_buf)

```
//将一个缓冲区的 block 中的数据从 ASCII 码表示变为数值数据(不占用额外内存空间)
void translateBlock(char* block_buf){
    char str[5];
    str[5] = '\0';
    int* int_block_buf = (int*)block_buf;
    for(int i=0; i<BLOCK_ITEM_NUM; i++){
        for (int j = 0; j < ATTR_SIZE; j++)
        {
            str[j] = *(block_buf + i*ITEM_SIZE + j);
        }
        int item_attr1 = atoi(str);
        for (int j = 0; j < ATTR_SIZE; j++)
        {
            str[j] = *(block_buf + i*ITEM_SIZE + ATTR_SIZE + j);
        }
        int item_attr2 = atoi(str);

        int_block_buf[i*2] = item_attr1;
        int_block_buf[i*2+1] = item_attr2;
    }
}
```

void reverseTranslateBlock(char* block_buf)

```
//将一个缓冲区的 block 中的数据从数值表示变为 ASCII 表示(不占用额外内存空间)
void reverseTranslateBlock(char* block_buf){
    char str[5];

    int* int_block_buf = (int*)block_buf;
    for(int i=0; i<BLOCK_ITEM_NUM; i++){
        int item_attr1 = int_block_buf[2 * i];
```

```
        int item_attr2 = int_block_buf[2 * i + 1];
        memset(str, 0, 5);
        itoa(item_attr1, str, 10);
        for(int j=0; j<ATTR_SIZE; j++){
            block_buf[i * ITEM_SIZE + j] = str[j];
        }
        memset(str, 0, 5);
        itoa(item_attr2, str, 10);
        for(int j=0; j<ATTR_SIZE; j++){
            block_buf[i * ITEM_SIZE + ATTR_SIZE + j] = str[j];
        }
    }
}
```