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: 吕弋卿

学　号: 210110315

专　业: 计算机科学与技术

日　期: 2023-11-09

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，理解算法的 I/O 复杂性。

二、 实验环境

阐述本次实验的环境。

操作系统：Windows 11

集成环境：Code::Blocks 20.03

三、 实验内容

阐述本次实验的具体内容。

本次实验中，基于磁盘读写模拟程序库 ExtMem，实现了一系列数据库查询处理算法，包括以下部分：

1. 基于线性搜索的关系选择算法
2. 两阶段多路归并排序算法
3. 基于索引的关系选择算法
4. 基于排序的连接操作算法
5. 基于排序的集合交算法
6. 基于排序的集合并算法（附加题，已实现）
7. 基于排序的集合差算法（附加题，已实现）

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 。写明结果存储在磁盘的位置，并给出程序正确运行的结果截图。

(0). 主程序流程

```
#include "defines.h"

int main(void) {

    printTitle("基于线性搜索的选择算法 S.C=107");
    linearSearch();

    printTitle("两阶段多路归并排序算法");
    mergeSort(1);
    mergeSort(2);

    printTitle("基于索引的关系选择算法");
    constructIndex();
    traverseIndex();

    printTitle("基于排序的连接操作算法");
    sortedInnerjoin();

    printTitle("基于排序的集合交操作算法");
    intersectSchema();

    printTitle("基于排序的集合并操作算法");
    unionSchema();

    printTitle("基于排序的集合差操作算法");
    differenceSchema();

    printf("\n5个实验任务(包括附加题三种算法)已完成，输入任意字符退出： ");
    getchar();
    return 0;
}

```

(1) 实现基于线性搜索的关系选择算法

问题分析：

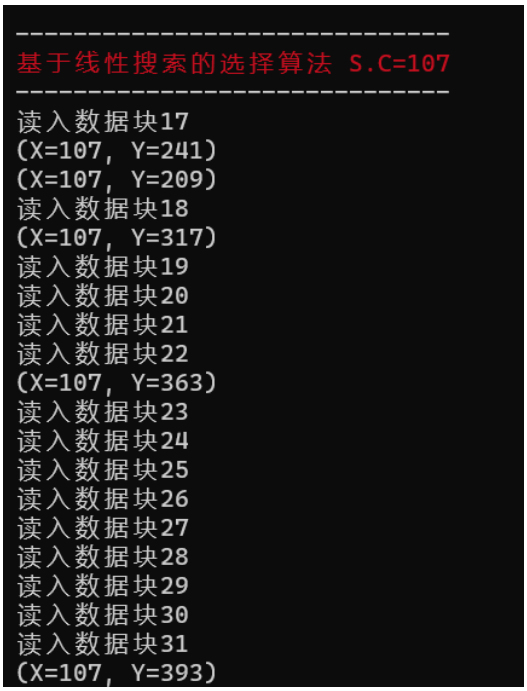
调用 extmem 中的 getBlockFromDisk()方法,按顺序读入 17.blk 到 48.blk，对于每个 blk 进行顺序遍历，调用 memcpy(),itoa(),atoi()等函数将其转化为由两个 int 型数据组成的二元组 Tuple,并将 Tuple.X,即 S.C 与 107 作比较,若相等，则打印到控制台，并写入内存缓冲块，内存缓冲块满之后，则写入磁盘块。遍历结束后若写缓冲块中有数据，也需写入磁盘块。最后，统计符合 S.C=107 的元组数量以及 IO 操作的次数。

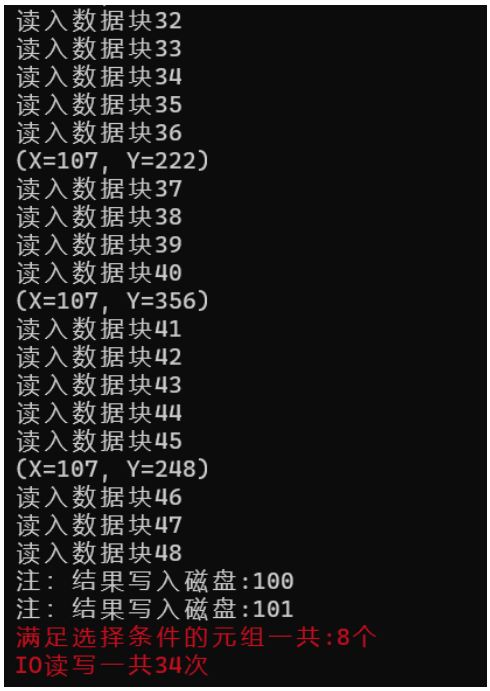
```
while(TRUE) {           //开始读磁盘中的Block

    bufferedReader = readBlockFromDisk(bufferedReaderAddress, &buf);
    printf("读入数据块%d\n", bufferedReaderAddress);

    for (int i=0; i<TUPLES_PER_BLOCK; i++) {

        Tuple bufferedTuple;                            //从读缓冲区获得元组,从char*转化为int型
        memcpy(temp, bufferedReader+i*TUPLIE_LENGTH, ELEMENT_LENGTH);
        bufferedTuple.x = atoi(temp);
        memcpy(temp, bufferedReader+i*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
        bufferedTuple.y = atoi(temp);

        if (bufferedTuple.x == 107) {               //选择条件: S.C=107
            printf("(X=%d, Y=%d)\n", bufferedTuple.x, bufferedTuple.y);     //打印满足选择条件的元组
            memcpy(bufferedWriter+foundCount*TUPLIE_LENGTH, bufferedReader+i*TUPLIE_LENGTH, TUPLIE_LENGTH);     //将满足条件的元组添加到写缓冲区
            foundCount++;

            if (foundCount == TUPLES_PER_BLOCK) {   //此块已写7个元组?
                itoa(bufferedWriterAddress+writeCount+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);    //将下个block的地址添加到写缓冲区
                if (writeBlockToDisk(bufferedWriter, bufferedWriterAddress + writeCount, &buf) != 0) {      //将缓冲区内的数据写入磁盘block
                    perror("Writing Block Failed!\n");
                    return;
                }
                writeCount++;
                foundCount = 0;                     //重置块内元组计数器
                freeBlockInBuffer(bufferedWriter, &buf);
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
        }
    }
    memcpy(temp, bufferedReader+TUPLES_PER_BLOCK*TUPLIE_LENGTH, ELEMENT_LENGTH);        //读取block末尾的数据标识下个block地址
    bufferedReaderAddress = atoi(temp);
    if (bufferedReaderAddress >= SRC_S_INIT_ADDR+BLOCK_S_LENGTH) {          //若读地址已过48则读终止
        break;
    }
    freeBlockInBuffer(bufferedReader, &buf);
}
```

```
if (foundCount) {   //将写缓冲区内的剩余数据写入磁盘block
    memcpy(bufferedWriter+foundCount*TUPLIE_LENGTH, emptyString, (TUPLES_PER_BLOCK-foundCount)*TUPLIE_LENGTH);
    itoa(bufferedWriterAddress+writeCount+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);    //将下个block的地址添加到写缓冲区
    if (writeBlockToDisk(bufferedWriter, bufferedWriterAddress+writeCount, &buf) != 0) {    //写磁盘
        perror("Writing Block Failed!\n");
        return;
    }
    writeCount++;
}
for (int i=0; i<writeCount; i++) {
    printf("注: 结果写入磁盘:%d\n", bufferedWriterAddress + i);
}
setColor(RED);
printf("满足选择条件的元组一共:%d个\n", (writeCount-1)*TUPLES_PER_BLOCK + foundCount);
printf("IO读写一共%d次\n", buf.numIO);
setColor(WHITE);
freeBuffer(&buf);
return;
```

实验结果：

CodeBlocks 编译运行结果如下：





共选择元组 8 个，写入磁盘块 100.blk, 101.blk：

(107,241) (107,209) (107,317) (107,363) (107,393) (107,222) (107,356) (107,248)

IO 次数为 34 次，其中，读 17~48.blk 占 32 次，写 100,101.blk 占 2 次

(2) 实现两阶段多路归并排序算法（TPMMS）

问题分析：

由于内存缓冲区 buf 中最多存放 8 个块，其中 1 个作为读缓冲，1 个作为写缓冲，则剩余 6 个块可用，因此选择 6 路归并。开始 WHILE(TRUE)的大循环，每次迭代选择不多于 6 各块，先对其进行块内排序，再作多路归并。下为内排序的实现：

```
while (TRUE) {
    bufferedWriter = getNewBlockInBuffer(&buf);
    remnantBlock = termReadAddress-readAdrress;      //剩余未参与排序的块数
    Tuple tempTuple;
    if((++remnantBlock) >= 6) {
        remnantBlock = 6;                    //大于6块则选择前6块进行排序
    }
    for(int i=0; i<remnantBlock; i++) {      //读入block并进行内部排序
        if ((bufferedBlock[i] = readBlockFromDisk(readAdrress+i, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return;
        }
        extractTuple(bufferedBlock[i], bufferedTuple);
        for(int j=0; j<TUPLES_PER_BLOCK; j++) {        //内排序，采用选择排序
            temp = bufferedTuple[j].x;                  //每轮选择Tuple中X值最小的元组提至最前
            order = j;
            for (int k=j; k<TUPLES_PER_BLOCK; k++) {
                if (bufferedTuple[k].x < temp) {
                    temp = bufferedTuple[k].x;
                    order = k;
                }
            }
            tempTuple = bufferedTuple[j];
            bufferedTuple[j] = bufferedTuple[order];
            bufferedTuple[order] = tempTuple;
            itoa(bufferedTuple[j].x, tempConvert, 10);      //内排序后写回block
            memcpy(bufferedBlock[i]+j*TUPLIE_LENGTH, tempConvert, ELEMENT_LENGTH);
            itoa(bufferedTuple[j].y, tempConvert, 10);
            memcpy(bufferedBlock[i]+j*TUPLIE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
        }
    }
```

对读入的≤6 块内排序完毕后进行第一次多路归并，以 indexVector[6]的每一维来标记一个维度已归并元组的下标，以 tempTuple[6]暂存每路当前的首个元组，再在这 6 个元素中，选择 Tuple.X 最小的元组，记录其下标为 order，将其写入写缓冲，并将此路的 indexVector+1，以标识此路的此元组已归并完毕。若某路的 indexVector==TUPLES_PER_BLOCK，也即此路已归并的元组数等于块中可容纳的元组总数(7)，将此路的 tempTuple.X 置为一个上界数(宏定义 UPPER_LIMIT)，以确保以后不会再选择此路元素。当写缓冲区中有 7 个元组时，写磁盘并清空写缓冲。如此进行若干次后，有若干组(每组≤6 块)已归并完毕。

下图为第一次归并的算法实现：

```
int indexVector[6] = {0};        //第1次多路归并，indexVector的每个维标识一个block中归并进度的下标
while (!isMerged(indexVector, remnantBlock)) {
    Tuple element[6];            //暂存每路中的极小值
    for (int i = 0; i < remnantBlock; i++) {
        if (indexVector[i] < TUPLES_PER_BLOCK) {
            memcpy(tempConvert, bufferedBlock[i]+indexVector[i]*TUPLIE_LENGTH, ELEMENT_LENGTH);
            element[i].x = atoi(tempConvert);
            memcpy(tempConvert, bufferedBlock[i]+indexVector[i]*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
            element[i].y = atoi(tempConvert);
        } else{
            element[i].x = UPPER_LIMIT;     //此路已归并完毕，将此路的S.C设为上界，确保以后不会再选择此路
        }
    }
    temp = UPPER_LIMIT;
    for(int i=0; i<remnantBlock; i++) {
        if (element[i].x < temp) {
            temp = element[i].x;
            order = i;                              //选择6路中极小值的下标
        }
    }
    itoa(element[order].x, tempConvert, 10);        //将极小值并入总集
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH, tempConvert, ELEMENT_LENGTH);
    itoa(element[order].y, tempConvert, 10);
    memcpy(bufferedWriter+(writeOrder++)*TUPLIE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
    if(writeOrder % TUPLES_PER_BLOCK == 0) {
        itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
        if (writeBlockToDisk(bufferedWriter, writeAddress, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return;
        }
        writeOrder = 0;
        writeAddress++;
        bufferedWriter = getNewBlockInBuffer(&buf);
    }
    if(indexVector[order] < TUPLES_PER_BLOCK){      //若此块还未归并完毕
        indexVector[order]++;                       //此块的归并进度下标自增
    }
}
readAdrress += remnantBlock;
freeBlockInBuffer(bufferedWriter, &buf);
for (int i = 0; i < remnantBlock; i++) {
    freeBlockInBuffer(bufferedBlock[i], &buf);
}
```

接下来进行第二次多路归并，由于 R 有 16 个块，S 有 32 个块，$\log_6 16$和$\log_6 32$均小于 2，因此两次 6 路归并即可完成所有的排序。第二次归并与第一次思路相似，但有一些值得注意的点：先计算路数以及最后一路含有的块数（每个块都是满的，因此不用考虑残块），在代码中用 segment 和 fragment 作标识，每路归并进度由 fragmentIndexVector[6]中的一个维度标识。与第一次归并不同的是，每路有多个块，当一个块归并完毕后需要读入此路的下一个块继续参与归并。

下图为第二次归并算法的实现：

```
while(TRUE) {
    for(int i=0; i<segmentCount; i++) {     //复制每个segment中剩余block的第一个元组
        if(fragmentIndexVector[i] < TUPLES_PER_BLOCK) {
            memcpy(tempConvert, bufferedBlock[i]+fragmentIndexVector[i]*TUPLIE_LENGTH, ELEMENT_LENGTH);
            bufferedTuple[i].x = atoi(tempConvert);
            memcpy(tempConvert, bufferedBlock[i]+fragmentIndexVector[i]*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
            bufferedTuple[i].y = atoi(tempConvert);
        } else {
            bufferedTuple[i].x = UPPER_LIMIT;
        }
    }
    temp = UPPER_LIMIT;
    for(int i=0; i<segmentCount; i++) {
        if(bufferedTuple[i].x < temp) {
            temp = bufferedTuple[i].x;
            order = i;
        }
    }
    itoa(bufferedTuple[order].x, tempConvert, 10);      //选择最小元组写入写缓冲区
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH, tempConvert, ELEMENT_LENGTH);
    itoa(bufferedTuple[order].y, tempConvert, 10);
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
```

```
    if ((++writeOrder) == TUPLES_PER_BLOCK) {
        itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
        if (writeBlockToDisk(bufferedWriter, (writeAddress++), &buf) != 0) {
            perror("Writing Block Failed!\n");
            return;
        }
        writeOrder = 0;
        bufferedWriter = getNewBlockInBuffer(&buf);
    }
    if(fragmentIndexVector[order] == 6) {
        if(order == segmentCount-1) {
            if(segmentIndexVector[order] == fragmentCount-1){
                fragmentIndexVector[order] = 7;
            } else {
                segmentIndexVector[order]++;
                fragmentIndexVector[order] = 0;
                freeBlockInBuffer(bufferedBlock[order], &buf);
                bufferedBlock[order] = readBlockFromDisk(initWriteAddress+order*6+segmentIndexVector[order], &buf);
            }
        } else {
            if(segmentIndexVector[order] == 5){
                fragmentIndexVector[order] = 7;
            } else {
                segmentIndexVector[order]++;
                fragmentIndexVector[order] = 0;
                freeBlockInBuffer(bufferedBlock[order], &buf);
                bufferedBlock[order] = readBlockFromDisk(initWriteAddress+order*6+segmentIndexVector[order], &buf);
            }
        }
    } else {
        fragmentIndexVector[order]++;
    }
    if(isMerged(fragmentIndexVector, segmentCount)){
        break;
    }
}
```

在两次归并中，使用了一些磁盘 block 作为中间结果的暂存块，在归并完成后，将这些暂存块清除。最后，在控制台打印 IO 次数。

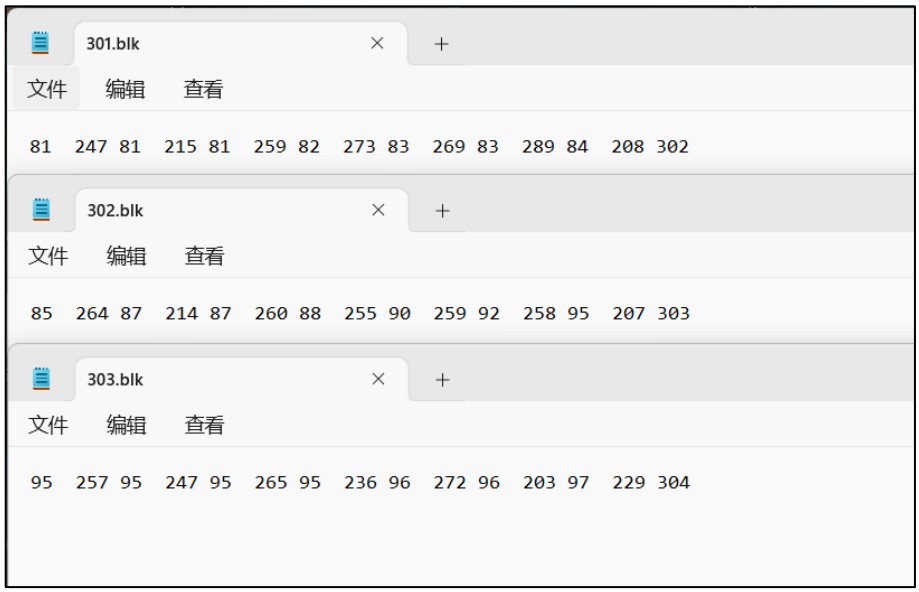
实验结果：

R 的排序结果存储在 301~316.blk,S 的排序结果存储在 317~348.blk

控制台输出 IO 次数：

```
------------------------------
两阶段多路归并排序算法
------------------------------
对R的多路归并排序完成，IO读写96次
对S的多路归并排序完成，IO读写192次
```

打开 301,302,303.blk 进行检验，发现确实已按照 R.A 进行升序排序，并在块尾标识了下个块的地址，证明了此算法的正确性。



(3) 实现基于索引的关系选择算法

问题分析：

首先要建立索引，基于任务 2 的排序结果，索引建立流思路如下：顺序读取排序后的块，获取 Tuple.X，用两个 int 型数据来存储索引键，currentKey 和 nextKey，每次读 Tuple 都更新 nextKey=Tuple.X，并与 currentKey 比较，相等则继续读取下一个 Tuple,不等则建立索引，并更新 currentKey=nextKey。从性能和存储角度考虑，本次实验选择建立稀疏索引，即对于每个 block，只建立一个索引，键为其中第一次 currentKey≠nextKey 时的 currentKey 值。将 317~348.blk 中的 Tuple 看成一个序列，索引键对应的索引值为这个键第一次出现时的序列号。例如在这个序列中，(x,y)是第 24 个元组，则索引键值对为：(x,24)。建立索引的算法实现如下图：

```
for(int readAddress=SORTED_S_INIT_ADDR; readAddress<SORTED_S_INIT_ADDR+BLOCK_S_LENGTH; readAddress++) {
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    Tuple bufferedTuple[TUPLES_PER_BLOCK];
    extractTuple(bufferedReader, bufferedTuple);
    int currentKey = bufferedTuple[0].x;
    int nextKey = currentKey;
    for(int i=1; i<TUPLES_PER_BLOCK; i++) {
        if((nextKey=bufferedTuple[i].x) != currentKey) {            //找到相异元素则建立索引
            int indexValue = (readAddress-SORTED_S_INIT_ADDR)*TUPLES_PER_BLOCK+i;   //索引值
            char temp[ELEMENT_LENGTH+1];
            itoa(bufferedTuple[i].x, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
            sprintf(temp, "%-4d", indexValue);
            memcpy(bufferedWriter+writeCount*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
            if ((++writeCount) == 7) {
                itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
                printf("索引写入磁盘：%d\n", writeAddress);
                if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return;
                }
                indexLength++;
                writeCount = 0;
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
            freeBlockInBuffer(bufferedReader, &buf);
            break;
        }
        currentKey = nextKey;
        if (i == TUPLES_PER_BLOCK-1) {
            freeBlockInBuffer(bufferedReader, &buf);
        }
    }
}
```

建立索引后，要根据索引进行搜索，因为建立的是稀疏索引，因此每次需要读入相邻的两个索引 initIndex, termIndex，并判断要搜索的键是否在(initIndex, termIndex]区间内，若在，则从 initIndex 对应的索引值出开始查询，若 R.S 与搜索值相等则写入缓存，缓存满则将结果写入磁盘，在遇到第一个相异值时结束。此外，由于区间采用半开半闭，若 initIndex ≥ 搜索值，则还需回溯至上个索引块进行查询。下为查询的第一阶段，查找索引键值的算法实现：

```
for(int readAddress = INDEX_INIT_ADDR; readAddress < INDEX_INIT_ADDR+indexLength; readAddress++) {    //遍历索引块
    printf("读入索引块%d\n", readAddress);
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    Tuple bufferedTuple[TUPLES_PER_BLOCK];
    extractTuple(bufferedReader, bufferedTuple);
    for(int i=0; i<TUPLES_PER_BLOCK-1; i++) {
        if (i == 0 && bufferedTuple[i].x > 107) {   //每个索引块的第一个索引键与上个块的最后一个索引键对比
            if (readAddress == INDEX_INIT_ADDR) {   //第一个索引块无需和前块对比
                initIndex = dataAddrOffset;
                termIndex = bufferedTuple[i].y+dataAddrOffset;
                break;
            } else {                                //回溯上个块
                char* tempReader = readBlockFromDisk(readAddress-1, &buf);
                Tuple tempTuple[TUPLES_PER_BLOCK];
                extractTuple(tempReader, tempTuple);
                if (tempTuple[TUPLES_PER_BLOCK-1].x <= 107) {
                    initIndex = tempTuple[TUPLES_PER_BLOCK-1].y+dataAddrOffset;
                    termIndex = bufferedTuple[i].y+dataAddrOffset;
                    freeBlockInBuffer(tempReader, &buf);
                    break;
                }
            }
        }
        if (bufferedTuple[i].x <= 107 && bufferedTuple[i + 1].x > 107) {    //如果S.C=107在索引区间内
            initIndex = bufferedTuple[i].y + dataAddrOffset;
            termIndex = bufferedTuple[i + 1].y + dataAddrOffset;
            printf("已找到索引\n");
            isIndexFound = true;
            break;
        }
    }
    freeBlockInBuffer(bufferedReader, &buf);
    if(isIndexFound){
        break;
    }
}
if (initIndex == termIndex) {
    printf("Can not find by index!\n");
    return;
}
```

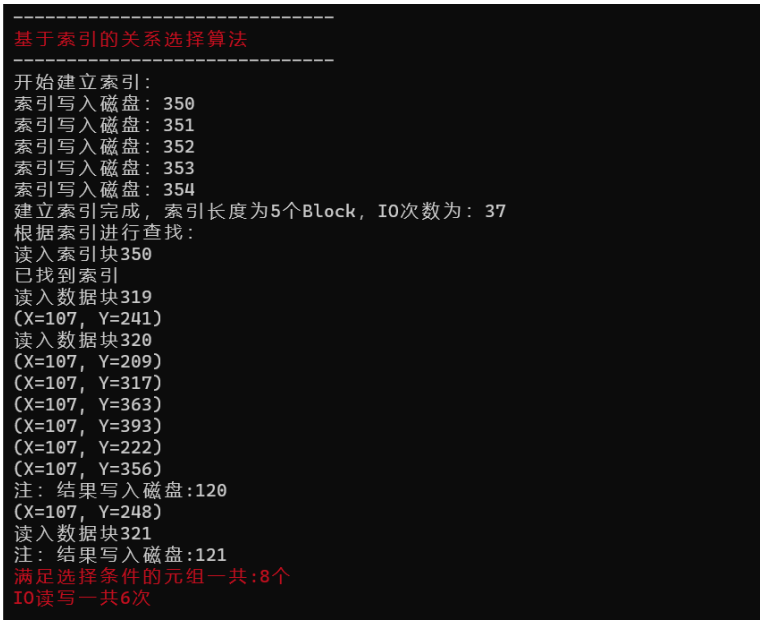
下为查询的第二阶段，根据索引值和偏移量查找数据块的算法实现：

```
for (int readAddress = initReadAddress; readAddress <= termReadAddress; readAddress++) {
    Tuple bufferedTuple[7];         //根据索引值读取磁盘数据块
    printf("读入数据块%d\n", readAddress);
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    extractTuple(bufferedReader, bufferedTuple);
    int currentIndex = 0, termIndex = 7;
    if(readAddress == initReadAddress){
        currentIndex = initReadIndex;
    }
    if(readAddress == termReadAddress) {
        termIndex = termReadIndex;
    }
    while(currentIndex < termIndex) {
        if (bufferedTuple[currentIndex].x == 107) {
            validTuples++;
            printf("(X=%d, Y=%d)\n", bufferedTuple[currentIndex].x, bufferedTuple[currentIndex].y);
            char temp[ELEMENT_LENGTH+1];
            itoa(bufferedTuple[currentIndex].x, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
            itoa(bufferedTuple[currentIndex].y, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
            if ((++writeCount) == TUPLES_PER_BLOCK) {
                itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
                printf("注：结果写入磁盘:%d\n", writeAddress);
                if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return;
                }
                writeCount = 0;
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
        }
        currentIndex++;
    }
    freeBlockInBuffer(bufferedReader, &buf);
}
```

实验结果：

索引块写在 350~354.blk 中，基于索引的搜索结果写在 120,121.blk 中。

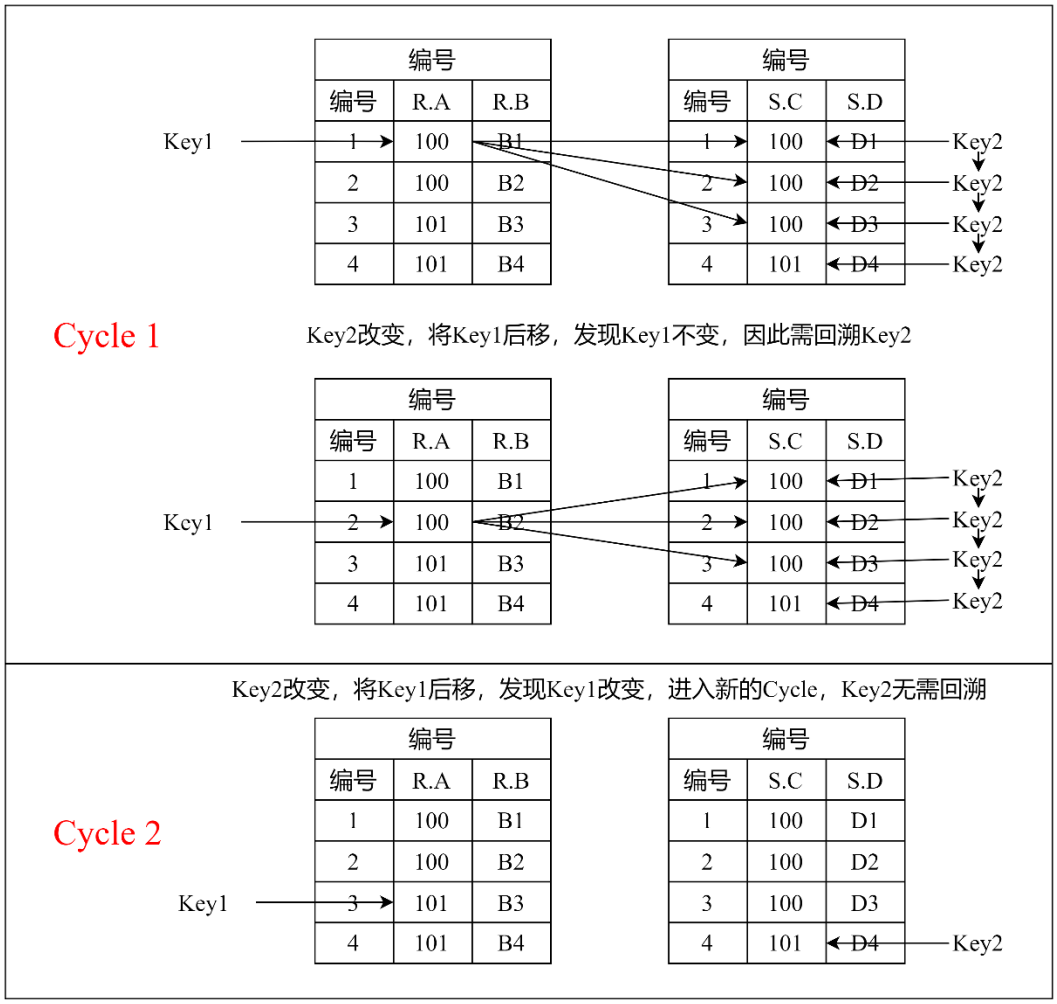
控制台输出：可见，基于索引查找只用了 6 次 IO，相比线性查找 34 次 IO 有极大优化。



(4) 实现基于排序的连接操作算法（Sort-Merge-Join）

问题分析：

这个算法基于任务 2 的已排序的数据集（即为第一趟），基本思想为：对关系模式 R,S 分别进行迭代，以 R.A 为 Key1，将 Key1 值改变设为一个 Cycle，S.C 为 Key2，以 Key1 为基准将 Key2 向后迭代，当第一次 Key1=Key2 时，对 S 和 R 进行 Join 操作，再将 Key2 后移并 Join 直至 Key2 值改变为止。但考虑到可能有连续的几个 R 中的元组，其 R.A 都等于 Key1，因此，需要将 Key1 后移并进行检查，若 Key1 改变，则此次 Cycle 已完成，进入下一个 Cycle 即可；若 Key1 未变，则说明还有 R.A 相同的元组需要和 S 中的元组连接，此时，就需要将 Key2 回溯至此 Cycle 的初始位置，再对当前的 Key1 标识的元组进行 Join。下为示意图：



在上述的 Key2 回溯过程的具体实现中，有时会涉及到 Block 的回溯，因此实现的

代码较长，如下图：

```
mem.c  test.c  AuxiliaryFunctions.c  defines.h  *CoreFunction.c  extmem.h
463
464  d sortedInnerjoin() {
465   Buffer buf;
466   if (!initBuffer(520, 64, &buf)) {
467       perror("Buffer Initialization Failed!\n");
468       return;
469   }
470   int innerjoinKey = 0, innerjoinOperationTicker = 0;
471   char* bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR, &buf);
472   char* bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR, &buf);
473   char* bufferedWriter = getNewBlockInBuffer(&buf);
474   int writeAddress = SORTED_INNERJOIN_INIT_ADDR;
475   int writeCount = 0, writeIndexR = 0, writeIndexS = 0, currentIndexR = 0, currentIndexS = 0;
476   Tuple bufferedTupleR[TUPLES_PER_BLOCK], bufferedTupleS[TUPLES_PER_BLOCK];
477   extractTuple(bufferedReaderR, bufferedTupleR);
478   extractTuple(bufferedReaderS, bufferedTupleS);
479   bool isJoinedR = false, isJoinedS = false;    // 记录前一个元素和当前元素是否发生了连接
480
481   while(TRUE) {      //开始遍历两个关系模式
482       if (bufferedTupleR[currentIndexR].x == bufferedTupleS[currentIndexS].x) {   //R.A=S.C, 可以进
483           innerjoinOperationTicker++;
484           isJoinedS = true;
485           char temp[ELEMENT_LENGTH+1];        //将连接结果写入block缓冲
486           itoa(bufferedTupleS[currentIndexS].x, temp, 10);
487           memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
488           itoa(bufferedTupleS[currentIndexS].y, temp, 10);
489           memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
490           itoa(bufferedTupleR[currentIndexR].x, temp, 10);
491           memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+2*ELEMENT_LENGTH, temp, ELEMENT_LENGTH)
492           itoa(bufferedTupleR[currentIndexR].y, temp, 10);
493           memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+3*ELEMENT_LENGTH, temp, ELEMENT_LENGTH)
494           writeCount++;
495           if (writeCount == 3) {  //每个块可容纳3个4元组
496               memset(bufferedWriter+48, 0, 4*ELEMENT_LENGTH+ELEMENT_LENGTH);      //写入磁盘
497               itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
498               printf("注: 结果写入磁盘:%d\n", writeAddress);
499               if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
500                   perror("Writing Block Failed!\n");
501                   return;
502               }
503               writeCount = 0;
504               freeBlockInBuffer(bufferedWriter, &buf);
505               bufferedWriter = getNewBlockInBuffer(&buf);
506           }
507           if ((++currentIndexS) == TUPLES_PER_BLOCK) {
508               if ((++writeIndexS) == BLOCK_S_LENGTH) {
509                   break;
510               }
511               currentIndexS = 0;
512               freeBlockInBuffer(bufferedReaderS, &buf);
513               bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+writeIndexS, &buf);
514               extractTuple(bufferedReaderS, bufferedTupleS);
515           }
516       } else if (bufferedTupleR[currentIndexR].x > bufferedTupleS[currentIndexS].x) {     //R.A>S.C
517           currentIndexS++;
518           if (currentIndexS == TUPLES_PER_BLOCK) {
519               writeIndexS++;
520               if (writeIndexS == BLOCK_S_LENGTH){
521                   break;
522               }
523               currentIndexS = 0;
524               freeBlockInBuffer(bufferedReaderS, &buf);
525               bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
526               extractTuple(bufferedReaderS, bufferedTupleS);
527           }
528       } else {
529           if (bufferedTupleR[currentIndexR].x == innerjoinKey && isJoinedR && !isJoinedS) {   //R
530               if (currentIndexS == 0) {
531                   if (writeIndexS == 0) {
532                       break;
533                   } else {
534                       currentIndexS = 6;
535                       freeBlockInBuffer(bufferedReaderS, &buf);
536                       bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+(--writeIndexS), &buf
537                       extractTuple(bufferedReaderS, bufferedTupleS);
538                   }
539               } else {
540                   currentIndexS--;
541               }
542               while(bufferedTupleS[currentIndexS].x == bufferedTupleR[currentIndexR].x) {
543                   if (currentIndexS == 0) {
544                       if (writeIndexS == 0) {
545                           break;
546                       } else {
547                           currentIndexS = 6;
548                           freeBlockInBuffer(bufferedReaderS, &buf);
549                           bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+(--writeIndexS),
550                           extractTuple(bufferedReaderS, bufferedTupleS);
551                       }
552                   }
553                   currentIndexS--;
554               }
555           } else {
556               isJoinedR = isJoinedS;
557               isJoinedS = false;
558               innerjoinKey = bufferedTupleR[currentIndexR++].x;
559               if (currentIndexR == TUPLES_PER_BLOCK) {
560                   if ((++writeIndexR) == BLOCK_R_LENGTH) {
561                       break;
562                   }
563                   currentIndexR = 0;
564                   freeBlockInBuffer(bufferedReaderR, &buf);
565                   bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + writeIndexR, &buf);
566                   extractTuple(bufferedReaderR, bufferedTupleR);
567               }
568           }
569       }
570   }
571   if (writeCount) {
572       memset(bufferedWriter+writeCount*4*ELEMENT_LENGTH, 0, 64-writeCount*4*ELEMENT_LENGTH);
573       itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
574       printf("注: 结果写入磁盘:%d\n", writeAddress);
575       if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
576           perror("Writing Block Failed!\n");
```

实验结果：

控制台输出如下：

连接结果写在磁盘块 400~478.blk 中

```
------------------------------
基于排序的连接操作算法
------------------------------
注: 结果写入磁盘:400
注: 结果写入磁盘:401
注: 结果写入磁盘:402
注: 结果写入磁盘:403
注: 结果写入磁盘:404
注: 结果写入磁盘:405
注: 结果写入磁盘:406
注: 结果写入磁盘:407
注: 结果写入磁盘:408
注: 结果写入磁盘:409
注: 结果写入磁盘:410
注: 结果写入磁盘:411
注: 结果写入磁盘:412
注: 结果写入磁盘:413
注: 结果写入磁盘:414
注: 结果写入磁盘:415
注: 结果写入磁盘:416
注: 结果写入磁盘:417
注: 结果写入磁盘:418
注: 结果写入磁盘:419
```

```
注: 结果写入磁盘:457
注: 结果写入磁盘:458
注: 结果写入磁盘:459
注: 结果写入磁盘:460
注: 结果写入磁盘:461
注: 结果写入磁盘:462
注: 结果写入磁盘:463
注: 结果写入磁盘:464
注: 结果写入磁盘:465
注: 结果写入磁盘:466
注: 结果写入磁盘:467
注: 结果写入磁盘:468
注: 结果写入磁盘:469
注: 结果写入磁盘:470
注: 结果写入磁盘:471
注: 结果写入磁盘:472
注: 结果写入磁盘:473
注: 结果写入磁盘:474
注: 结果写入磁盘:475
注: 结果写入磁盘:476
注: 结果写入磁盘:477
注: 结果写入磁盘:478
连接完成，共执行连接操作235次
IO操作次数为: 160
```

(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

问题分析：

这里分析基于散列的两趟扫描集合交算法，（并和差思路类似，写在附加题中）：

这个算法与集合连接(Join)算法的思路相似，第一趟排序已完成，在第二趟查找迭代中，Join 中做的是，如果 R.A=S.C，则 Tuple(R)×Tuple(S)；而对于集合交(Intersection)，则需要 R.A=S.C 且 R.B=S.D 时，保存 Tuple。虽然操作不同，但回溯的思想与 Join 类似。

代码如下：

mem.c test.c AuxiliaryFunctions.c defines.h *CoreFunction.c extmem.h

```c
while(TRUE) {
    if (bufferedTupleR[currentIndexR].x == bufferedTuples[currentIndexS].x) {   //如果R.A=S.C，则比较
        if (!currentIndexS) {    //回溯S,从首次交点开始
            if (writeIndexS) {
                currentIndexS = 6;
                freeBlockInBuffer(bufferedReaderS, &buf);
                bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + (--writeIndexS), &buf);
                extractTuple(bufferedReaderS, bufferedTuples);
            }
        } else {
            currentIndexS--;
        }
        if (writeIndexS!= 0||currentIndexS!= 0) {
            while(bufferedTuples[currentIndexS].x -- bufferedTupleR[currentIndexR].x) { //R.A=S.C,且B
                if (!currentIndexS) {
                    if (!writeIndexS) {
                        currentIndexS--;
                        break;
                    } else {
                        currentIndexS = TUPLES_PER_BLOCK;
                        freeBlockInBuffer(bufferedReaderS, &buf);
                        bufferedReaderS=readBlockFromDisk(SORTED_S_INIT_ADDR+(--writeIndexS), &buf);
                        extractTuple(bufferedReaderS, bufferedTuples);
                    }
                }
                currentIndexS--;
            }
            if ((++currentIndexS) == TUPLES_PER_BLOCK) {
                if ((++writeIndexS) == BLOCK_S_LENGTH) {
                    break;
                }
                currentIndexS = 0;
                freeBlockInBuffer(bufferedReaderS, &buf);
                bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
                extractTuple(bufferedReaderS, bufferedTuples);
            }
        }

        while(bufferedTuples[currentIndexS].x == bufferedTupleR[currentIndexR].x) {     //R.A=S.C
            if (bufferedTuples[currentIndexS].y -- bufferedTupleR[currentIndexR].y) {   //R.B=S.D
                printf("(X=%d, Y=%d)\n", bufferedTuples[currentIndexS].x, bufferedTuples[currentIndexS
                intersectionCount++;
                char temp[ELEMENT_LENGTH+1];
                itoa(bufferedTuples[currentIndexS].x, temp, 10);     //找到交元素，写入block
                memcpy(bufferedWriter+writeIndex*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
                itoa(bufferedTuples[currentIndexS].y, temp, 10);
                memcpy(bufferedWriter+writeIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
                if ((++writeIndex) == TUPLES_PER_BLOCK) {
                    memset(bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 0, TUPLIE_LENGTH);
                    itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
                    printf("注：结果写入磁盘:%d\n", writeAddress);
                    if (writeBlockToDisk(bufferedWriter, (writeAddress++), &buf) != 0) {
                        perror("Writing Block Failed!\n");
                        return;
                    }
                    writeIndex = 0;
                    memcpy(bufferedWriter, emptyString, 64);
                    bufferedWriter = getNewBlockInBuffer(&buf);
                }
            }
            if((++currentIndexS) == TUPLES_PER_BLOCK) {
                if((++writeIndexS) == BLOCK_S_LENGTH){
                    break;
                }
                currentIndexS = 0;
                freeBlockInBuffer(bufferedReaderS, &buf);
                bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+writeIndexS, &buf);
                extractTuple(bufferedReaderS, bufferedTuples);
            }
        }
        intersectKey = bufferedTupleR[currentIndexR].x;
        if((++currentIndexR) == TUPLES_PER_BLOCK) {
            readIndexR++;
            if(readIndexR == BLOCK_R_LENGTH){
                break;
            }
            currentIndexR = 0;
            freeBlockInBuffer(bufferedReaderR, &buf);
            bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + readIndexR, &buf);
            extractTuple(bufferedReaderR, bufferedTupleR);
        }
        if(currentIndexS == TUPLES_PER_BLOCK) {
            if((++writeIndexS) == BLOCK_S_LENGTH){
                break;
            }
            currentIndexS = 0;
            freeBlockInBuffer(bufferedReaderS, &buf);
            bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
            extractTuple(bufferedReaderS, bufferedTuples);
        }
    } else if(bufferedTupleR[currentIndexR].x > bufferedTuples[currentIndexS].x) {
        intersectKey = bufferedTuples[currentIndexS].x;
        if ((++currentIndexS) == TUPLES_PER_BLOCK) {
            if ((++writeIndexS) == BLOCK_S_LENGTH){
                break;
            }
            currentIndexS = 0;
            freeBlockInBuffer(bufferedReaderS, &buf);
            bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
            extractTuple(bufferedReaderS, bufferedTuples);
        }
    } else {
        if(bufferedTupleR[currentIndexR].x==intersectKey) {
            if (currentIndexS == 0) {
                if (writeIndexS) {
                    currentIndexS = 6;
                    freeBlockInBuffer(bufferedReaderS, &buf);
                    bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+(--writeIndexS), &buf
                    extractTuple(bufferedReaderS, bufferedTuples);
                }
            } else {
                currentIndexS--;
            }
        } else {
            if ((++currentIndexR) == TUPLES_PER_BLOCK) {
                if ((++readIndexR) == BLOCK_R_LENGTH){
                    break;
                }
                currentIndexR = 0;
                freeBlockInBuffer(bufferedReaderR, &buf);
                bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + readIndexR, &buf);
                extractTuple(bufferedReaderR, bufferedTupleR);
            }
        }
    }
}
```

实验结果：

控制台输出如下：

集合交结果写在磁盘块 500,501.blk 中

```
------------------------------
基于排序的集合交操作算法
------------------------------
(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:500
(X=119, Y=259)
(X=121, Y=265)
(X=130, Y=232)
(X=144, Y=290)
注：结果写入磁盘:501
R和S的交集中共有11个元组
IO操作共99次
```

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。

集合交(Intersection)在第<四>部分已分析过，这里分析并和差。

1. 并

这个算法也是基于排序结果的，对 R、S 分别建立迭代指针，因 R.A 的值域下界小于 S.C，因此从 R.A 开始迭代，将遇到的元组拷贝至写缓存中(满了就写磁盘，与前面相同)，直到第一次遇到 R.A=S.C 为止。此时依然是使用回溯操作，对于 Key1，设置一个 Flag 来表示是否遇到相同的元组，每次 Key1 后移时置为 false，每轮中，Key1 不变，Key2 后移，将 Key2 标识的元组拷贝至写缓存，并检查是否有 S.C=R.A，没有则直接将 Key1 标识的元组写入写缓存，继续检查是否满足 S.D=R.B，若有则将 Flag 置为 true，每轮结束后若 Flag 为 false(即 S 中无与此 R 相同的元组)，则将 Key1 标识的元组写入写缓存。算法的核心思想即如果有相同元组则只写一次。实现如下：

控制台输出如下：

集合并结果写在磁盘块 600~646.blk 中

2. 差

与上述回溯思想类似，只是以 Key2 为主迭代键，设置 Flag，若有与其相同的 Key1 指向的元组，则不写入写缓存，若没有，则将 Key2 指向的元组写入写缓存，并将 Key2 后移，并判断 Key1 是否需要回溯，如此循环直到遍历完 S 集即可。代码如下：

```
while(!completeFlag) {
    if(bufferedDataR > bufferedDataS) {
        if(!isDifferentiated) {
            differenceTicker++;
            itoa(bufferedDataS, temp, 10);
            memcpy(bufferedWriter+readIndex*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
            memcpy(temp, bufferedReaderS+writeIndexS*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_
            bufferedTupleS.y = atoi(temp);
            memcpy(bufferedWriter+readIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LEN
            if((++readIndex) == TUPLES_PER_BLOCK) {
                itoa(writeAddress+writeCount+1, bufferedWriter+readIndex*TUPLIE_LENGTH, 10)
                if (writeBlockToDisk(bufferedWriter, writeAddress+writeCount, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return;
                }
                printf("注:结果写入磁盘:%d\n",writeAddress+(writeCount++));
                readIndex = 0;
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
        }
        isDifferentiated = false;
        if((++writeIndexS) == TUPLES_PER_BLOCK) {
            if((++readAddressS) <= SORTED_S_INIT_ADDR+BLOCK_S_LENGTH-1) {
                freeBlockInBuffer(bufferedReaderS, &buf);
                writeIndexS = 0;
                bufferedReaderS = readBlockFromDisk(readAddressS, &buf);
            } else {
                completeFlag = true;
            }
        }
        if(!completeFlag) {
            memcpy(temp, bufferedReaderS+writeIndexS*TUPLIE_LENGTH, ELEMENT_LENGTH);
            currentDataS = atoi(temp);
            previousDataS = bufferedDataS;
        }
        bool retraceFlag = !completeFlag && differenceFlag && currentDataS == bufferedDataS
        if(retraceFlag) {
            if(readAddressR != currentAddressKey) {
                freeBlockInBuffer(bufferedReaderR,&buf);
                bufferedReaderR = readBlockFromDisk(currentAddressKey, &buf);
            }
            writeIndexR = currentTupleKey;
            readAddressR = currentAddressKey;
            memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
            bufferedDataR = atoi(temp);
        } else {
            if(currentDataS != bufferedDataS) {
                isShifted = false;
                bufferedDataS = currentDataS;
            }
            differenceFlag = false;
        }
    } else if(bufferedDataR < bufferedDataS) {
        if((++writeIndexR) == TUPLES_PER_BLOCK) {
            if((++readAddressR) <= SORTED_R_INIT_ADDR+BLOCK_R_LENGTH-1) {
                writeIndexR = 0;
                freeBlockInBuffer(bufferedReaderR, &buf);
                bufferedReaderR = readBlockFromDisk(readAddressR, &buf);
            }
            else {
                completeFlag = true;
            }
        }
    } else if(bufferedDataR == bufferedDataS) {
        if((++writeIndexR) == TUPLES_PER_BLOCK) {
            if((++readAddressR) <= SORTED_R_INIT_ADDR+BLOCK_R_LENGTH-1) {
                writeIndexR = 0;
                freeBlockInBuffer(bufferedReaderR,&buf);
                bufferedReaderR = readBlockFromDisk(readAddressR, &buf);
            } else {
                completeKeyFlag = true;
                bufferedDataR = UPPER_LIMIT;
            }
        }
        if(!completeKeyFlag) {
            memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
            currentDataR = atoi(temp);
            bufferedDataR = currentDataR;
            previousDataR = bufferedDataR;
        }
    }
}
if(readIndex) {
    itoa(writeAddress+writeCount+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10)
    if (writeBlockToDisk(bufferedWriter, writeAddress+(writeCount++), &buf) != 0) {
        perror("Writing Block Failed!\n");
        return;
    }
    printf("注:结果写入磁盘:%d\n",writeAddress+writeCount);
    bufferedWriter = getNewBlockInBuffer(&buf);
}
setColor(RED);
printf("R和S的差集中共有%d个元组\n", differenceTicker);
printf("IO操作共%d次\n", buf.numIO);
setColor(WHITE);
freeBlockInBuffer(bufferedWriter,&buf);
freeBuffer(&buf);
return;
```

控制台输出如下：

连接结果写在磁盘块 700~731.blk 中

```
------------------------------
基于排序的集合差操作算法
------------------------------
注:结果写入磁盘:700
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:731
R和S的差集中共有213个元组
IO操作共135次
```