哈尔滨工业大学(深圳)

《数据库》实验报告

实验五

查询处理算法的模拟实现

学　院: 计算机科学与技术

姓　名: 宗晴

学　号: 200110513

专　业: 计算机科学与技术

日　期: 2022-12-19

一、 实验目的

阐述本次实验的目的。

1、理解索引、散列的作用。

2、掌握关系选择、投影、连接、集合的交、并、差等操作的实现算法。

3、加深对算法 I/O 复杂性的理解。

二、 实验环境

阐述本次实验的环境。

Windows 7 操作系统、CodeBlocks。

三、 实验内容

阐述本次实验的具体内容。

基于 ExtMem 程序库，使用有限内存（Buffer）实现关系选择、连接操作算法，实现集合并、交、差操作算法。不可定义义长度大于 10 的整型或字符型数组装载数据和保存中间结果。

ExtMem 程序库是一个由 C 语言开发的模拟外存磁盘块存储和存取的程序库。它提供了 1 个数据结构 Buffer 和 7 个 API 函数，用于实现内存缓冲区管理、磁盘块读/写等功能。

本实验使用 ExtMem 程序库已预先建立了关系 R 和 S 的物理存储，使用 extmem-c\data 目录下的每个文件模拟一个磁盘块。关系 R 具有属性 A 和 B，A 的值域为[100, 140]，B 的值域为[400, 500]；关系 S 具有属性 C 和 D，C 的值域为[120, 160]，D 的值域为[420, 920]。它们的属性值均为 int 型（4 个字节），R 和 S 的每个元组的大小均为 8 个字节。每个磁盘块大小为 64 个字节，可存放 7 个元组和 1 个后继磁盘块地址。关系 R 中包含 16*7=112 个元组，S 中包含 32*7=224

个元组。extmem-c\data 目录下的文件 1.blk 至 16.blk 为关系 R 的元组数据，文件 17.blk 至 48.blk 为关系 S 的元组数据。

ExtMem 程序库初始化的缓冲区块的大小为 64 个字节，缓冲区大小为 64*8+8=520 个字节。其中 8 个字节为标志位，表示每个块是否被占用，缓冲区内可最多存放 8 个块。

本实验的具体内容为：

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

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

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

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

（5）实现**基于排序或散列的两趟扫描算法**，实现**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。

附加题：

基于排序或散列的两趟扫描算法，实现剩余的两种集合操作算法。将结果存放在磁盘上，并统计并、交、差操作后的剩余元组个数。

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：

本实验需要实现的是**基于线性搜索的关系选择算法**。需要选出 S.C=128 的元组，记录 IO 读写次数，并将选择结果存放在磁盘上，也即模拟实现 select S.C, S.D from S where S.C = 128。

实现思路是，依次读入关系 S 的每个数据块，对于每个数据块，判断其中存储的 7 个元组是否有满足条件的。若有，则写入结果块，当结果块中写满 7 个元组时，将其写回磁盘。当查找结束后，若最后一块未被写满，也需要写回磁盘。最后，记录结果写入的磁盘块号、满足选择条件的元组个数和 IO 读写次数。

从第 100 号磁盘块开始存储结果块。

为实现该算法以及其余算法，需要首先定义中间存储块的数据结构 Blk，用数组 X 和 Y 存储七个元组的两个字段，用 addr 存储该块的后继磁盘块号：

```
// 存储每个磁盘块的内容
typedef struct Blk
{
    int X[7];
    int Y[7];
    int addr;
}Blk;
```

还需定义 int init_buf(Buffer *buf)函数，调用 ExtMem 库中的 initBuffer 函数用于初始化缓冲区 buffer，若发生错误则输出报错信息：

```
// 初始化buf
int init_buf(Buffer *buf)
{
    if (!initBuffer(520, 64, buf))
    {
        perror("Buffer Initialization Failed!\n");
        return -1;
    }
    return 0;
}
```

定义 void init_blk(Blk *blk)函数，用于初始化 Blk，将 7 个元组和后继磁盘块地址均置为-：

```
// 初始化blk
void init_blk(Blk *blk)
{
    for(int i=0; i<7; i++){
        blk -> X[i] = -1;
        blk -> Y[i] = -1;
    }
    blk -> addr = -1;
}
```

参照 test.c 文件，实现 void read_blk(unsigned char *blk_d, Blk *blk)函数，该函数实现了解析磁盘块内容的功能，分别解析缓冲区中第 blk_d 个块中的 7 个元组和 1 个后继磁盘块地址。从 blk_d+i*8 开始的 4 个字节中存储了第 i 个元组的第一个字段，从 blk_d+i*8+4 开始的 4 个字节中存储了第 i 个元组的第二个字段，最后的 4 个字节存储了后继磁盘块地址，将各个数据的每个字节存入字符串数组中，然后将字符串转化成数值并保存到中间存储块 blk 中：

```
// 解析每个磁盘块的内容（7个元组和1个后继磁盘块地址）
void read_blk(unsigned char *blk_d, Blk *blk)
{
    char str[5];
    int i;
    for (i = 0; i < 7; i++) // 读取7个元组
    {

        for (int k = 0; k < 4; k++)
        {
            str[k] = *(blk_d + i*8 + k);
        }
        blk->X[i] = atoi(str);
        for (int k = 0; k < 4; k++)
        {
            str[k] = *(blk_d + i*8 + 4 + k);
        }
        blk->Y[i] = atoi(str);
    }
    for (int k = 0; k < 4; k++)  // 读取后继磁盘块地址
    {
        str[k] = *(blk_d + i*8 + k);
    }
    blk->addr = atoi(str);
}
```

定义 Blk read_blk_data(int blk_id, Buffer *buf)函数，封装了初始化块函数 init_blk、读取块函数 readBlockFromDisk、解析块函数 read_blk 和释放块函数 freeBlockInBuffer 的代码：

```
// 封装初始化块、读取块、解析块、释放块的代码
Blk read_blk_data(int blk_id, Buffer *buf)
{
    unsigned char *blk_d;  // 用于存储从磁盘中读出的blk
    Blk blk;  // 用于存储从磁盘中读出的blk中的数据
    init_blk(&blk);  // 初始化块
    if ((blk_d = readBlockFromDisk(blk_id, buf)) == NULL)  // 读取块
    {
        perror("Reading Block Failed!\n");
    }
    read_blk(blk_d, &blk);  // 解析块
    freeBlockInBuffer(blk_d, buf);  // 释放块
    return blk;
}
```

参照 read_blk 函数实现 void write_blk(unsigned char *blk_d, Blk blk)，用于将某个磁盘块的内容写回 buffer，包括 7 个元组和 1 个后继磁盘块地址，分别写回到缓冲区中第 blk_d 个块中。将各个元组的两个字段和后继磁盘块地址按照十进制转化成字符串存入字符串数组中，第 i 个元组的第一个字段以字符串的形式按位存储到 blk_d+i*8 开始的 4 个字节中，第 i 个元组的第二个字段以字

符串的形式按位存储到 blk_d+i*8+4 开始的 4 个字节中，后继磁盘块地址以字符串的形式按位存储到最后的 4 个字节中：

```
// 将某个磁盘块的内容写回buffer（7个元组和1个后继磁盘块地址）
void write_blk(unsigned char *blk_d, Blk blk)
{
    char str[5];
    int i;
    for (i = 0; i < 7; i++)  // 写7个元组
    {
        itoa(blk.X[i],str,10);
        for (int k = 0; k < 4; k++)
        {
            *(blk_d + i*8 + k) = str[k];
        }
        itoa(blk.Y[i],str,10);
        for (int k = 0; k < 4; k++)
        {
            *(blk_d + i*8 + 4 + k) = str[k];
        }
    }
    itoa(blk.addr,str,10);
    for (int k = 0; k < 4; k++)  // 写后继磁盘块地址
    {
        *(blk_d + i*8 + k) = str[k];
    }
}
```

定义 void write_ans_blk(Blk *ans, Buffer *buf, int blk_id)函数，封装了解析块函数 write_blk、写回块函数 writeBlockToDisk、清空块函数 init_blk、释放块函数 freeBlockInBuffer 的代码，清空块的目的是便于结果块的重复利用：

```
// 封装解析块、写回块、清空块、释放块的代码
void write_ans_blk(Blk *ans, Buffer *buf, int blk_id)
{
    ans->addr = blk_id + 1;
    unsigned char *ans_d;  // 用于存储待写入磁盘的结果
    ans_d = getNewBlockInBuffer(buf);  // 获取缓冲区中的空闲块
    write_blk(ans_d, *ans);  // 解析结果块
    if (writeBlockToDisk(ans_d, blk_id, buf) != 0)  // 写回磁盘
    {
        perror("Writing Block Failed!\n");
        return;
    }
    init_blk(ans);  // 将中间结果块清空
    freeBlockInBuffer(ans_d, buf);  // 释放buf中的结果块
}
```

然后实现该算法的主体流程。

首先，输出任务名称并定义相关变量，包括存储从磁盘中读出的数据的中间存储块 blk、结果块 ans、满足条件的元组计数器 count、输出结果起始块号 blk_id_begin 和输出结果块数计数器 blk_num：

```
void task1_linearSearch(Buffer *buf)
{
    printf("-------------------------------------------------\n");
    printf("基于线性搜索的关系选择算法 S.C=128\n");
    printf("-------------------------------------------------\n");

    init_buf(buf);
    Blk blk;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 用于存储结果
    init_blk(&ans);
    int count = 0;  // 计算S.C=128的元组个数
    int blk_id_begin = 100;  // 输出结果起始块号
    int blk_num = 0;  // 输出结果块数
```

然后依次读取关系 S 的所有块，即第 17 号块到第 48 号块，并输出读入信息。对于每个块判断其中的 7 个元组中的字段 C 是否等于 128，若是则输出并写入结果块，并将满足条件的元组计数器 count++。当结果块中写满 7 个元组时，将该块写回，并将输出结果块数计数器 blk_num++。当查找结束后，若最后一块未被写满，也需要写回磁盘。若满足条件的元组计数器 count 等于 0，则输出“未查找到该元组！\n”并返回。

```
// 查找S.C=128的所有元组
for(int blk_id = 17; blk_id <= 48; blk_id++){
    blk = read_blk_data(blk_id, buf);
    printf("读入数据块%d\n", blk_id);
    for(int i = 0; i <= 7; i++){
        if(blk.X[i]==128){
            ans.X[count % 7] = blk.X[i];
            ans.Y[count % 7] = blk.Y[i];
            printf("(X=%d, Y=%d)\n", ans.X[count], ans.Y[count]);
            count++;
            if(count % 7 == 0){
                write_ans_blk(&ans, buf, blk_id_begin + blk_num);
                blk_num++;
            }
        }
    }
}
if(count != 0 && count % 7 != 0){
    write_ans_blk(&ans, buf, blk_id_begin + blk_num);
    blk_num++;
}
if(count == 0){
    printf("未查找到该元组！\n");
    return;
}
```

最后，打印结果写入的磁盘块号、满足选择条件的元组个数和 IO 读写次数。

```
printf("注：结果写入磁盘：[%d, %d]\n\n", blk_id_begin, blk_id_begin + blk_num - 1);
printf("满足选择条件的元组一共%d个。\n\n", count);
printf("IO读写一共%ld次。\n\n\n", buf->numIO);
return;
```

实验结果：

最终结果写入 100 和 101 号磁盘块。满足条件的元组共 10 个，IO 读写 34 次。

```
--------------------------------------------------
基于线性搜索的关系选择算法 S.C=128
--------------------------------------------------
读入数据块17
读入数据块18
读入数据块19
读入数据块20
读入数据块21
读入数据块22
读入数据块23
(X=128, Y=684)
(X=128, Y=431)
读入数据块24
读入数据块25
(X=128, Y=615)
(X=128, Y=429)
读入数据块26
读入数据块27
读入数据块28
读入数据块29
读入数据块30
(X=128, Y=584)
读入数据块31
读入数据块32
读入数据块33
读入数据块34
(X=128, Y=592)
(X=128, Y=457)
读入数据块35
(X=128, Y=720)
读入数据块36
读入数据块37
读入数据块38
读入数据块39
读入数据块40
读入数据块41
读入数据块42
读入数据块43
读入数据块44
读入数据块45
(X=128, Y=447)
读入数据块46
(X=128, Y=871)
读入数据块47
读入数据块48
注：结果写入磁盘：[100, 101]

满足选择条件的元组一共10个。

IO读写一共34次。
```

(2) 实现两阶段多路归并排序算法（TPMMS）

问题分析：

本实验需要实现的是**两阶段多路归并排序算法（TPMMS）**。需要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，注意由于内存有限，因此数据并无法一起全部读入内存，也不能定义大数组存储中间结果。

实现思路是分两阶段实现该算法，划分子集进行归并排序。

第一阶段：划分子集并进行子集内的排序，将 B 块数据划分成 N 个子集合，使每个子集合块数小于内存缓冲区可用块数。每个子集合装入缓冲区采用内排序排好序并重新写回磁盘。对于关系 R，我将其划分为 2 个子集合（每个子集合 8 块）进行排序，中间结果存储到第 201-216 号磁盘块；对于关系 S，我将其划分为 4 个子集合（每个子集合 8 块）进行排序，中间结果存储到第 217-248 号磁盘块。

第二阶段：各子集间归并排序，N 个已排序子集合的数据利用内存缓冲区进行归并排序。对于关系 R，子集合之间进行 2 路归并排序，最终结果存储在第 301-316 号磁盘块中；对于关系 S，子集合之间进行 4 路归并排序，最终结果存储在第 317-348 号磁盘块中。

为实现第一阶段算法，需要首先定义函数 int larger(int X1, int Y1, int X2, int Y2)，用于判断元组(X1, Y1)是否大于元组(X2, Y2)，（首先判断 X1 和 X2 的大小，若相等则再判断 Y1 和 Y2 的大小）若是则返回 1，否则返回 0：

```
// 若元组(X1, Y1)大于(X2, Y2)，则返回1；否则返回0
int larger(int X1, int Y1, int X2, int Y2)
{
    if(X1 > X2) return 1;
    else if(X1 == X2 && Y1 > Y2) return 1;
    else return 0;
}
```

定义函数 void swap(Blk *blk1, int i, Blk *blk2, int j)，用于交换 blk1 中第 i 个元组和 blk2 中第 j 的元组：

```
// 交换blk1中第i个元组和blk2中第j的元组
void swap(Blk *blk1, int i, Blk *blk2, int j)
{
    int tempx, tempy;
    tempx = blk1->X[i];
    tempy = blk1->Y[i];
    blk1->X[i] = blk2->X[j];
    blk1->Y[i] = blk2->Y[j];
    blk2->X[j] = tempx;
    blk2->Y[j] = tempy;
}
```

然后实现第一阶段的算法 void task2_stage1(Buffer *buf, int begin, int len, int out_begin)：

```
/* 对关系的子集合进行排序，按第一个属性从小到大然后第二个属性从小到大排序
   begin为待排序关系的起始块号，
   len为待排序关系的块数，
   out_begin为存储按子集合排好序的关系的起始块号。
*/
void task2_stage1(Buffer *buf, int begin, int len, int out_begin)
{
    int set_blk_num = 8;  // 每个子集合8块blk
    Blk blk[set_blk_num];  // 用于存储从磁盘中读出的blk中的数据
    int count = 0;  // 已排序块数
    while(len > count){
        // 读取每个子集合的数据
        int i = 0;
        while(len > count && i < set_blk_num){
            blk[i] = read_blk_data(begin + count, buf);
            count++;
            i++;
        }
        // 冒泡排序
        for(int j = 0; j < 7 * i; j++){
            for(int k = j + 1; k < 7 * i; k++){
                if(larger(blk[j/7].X[j%7], blk[j/7].Y[j%7],
                          blk[k/7].X[k%7], blk[k/7].Y[k%7]))
                    swap(&blk[j/7], j%7, &blk[k/7], k%7);
            }
        }
        for(int j = 0; j < i; j++)
            write_ans_blk(&blk[j], buf, out_begin + count - i + j);
    }
    printf("第一阶段：划分子集合并在子集合内排序，中间结果写入磁盘[%d, %d]\n"
           , out_begin, out_begin + len - 1);
    return;
}
```

如上图所示，函数参数的含义已在注释中写明。首先设置每个子集合块数为 8（由于缓冲区中只能存下 8 块数据块，并且采用冒泡排序不需要另外的结果块，因此选择每个子集合的块数为 8），然后定义 blk 数组用于存储从磁盘中读出的数据，定义变量 count 用于记录已排序块数。依次读取每个子集合的数据，对于

每个子集合，采用冒泡排序，定义两层循环，调用上述定义的 larger 函数判断元组大小关系，调用上述定义的 swap 函数在必要时交换两元组位置。最终将该子集合已排好序的结果写入磁盘。输出中间结果写入的磁盘块号。

为实现第二阶段算法，需要首先定义函数 int get_min(int X[], int Y[], int *minX, int *minY, int way_num, int not_finished[])，用于获取最小的元组及其所在第几路。X 和 Y 数组中分别存放各路当前读到的元组（已读完的元组对应位置为空），minX 和 minY 指针用于传回最小的元组，way_num 为总路数，not_finished 数组标志各路是否已被读完，函数返回值为最小的元组所在的路数：

```
// 获取最小的元组及其所在路数
int get_min(int X[], int Y[], int *minX, int *minY, int way_num, int not_finished[])
{
    *minX = 1000;
    *minY = 1000;
    int way_id;
    for (int i = 0; i < way_num ; i ++ ) {
        // 当前路还未被读完，且当前路的元组更小
        if (not_finished[i] && larger(*minX, *minY, X[i], Y[i])) {
            *minX = X[i];
            *minY = Y[i];
            way_id = i;
        }
    }
    return way_id;
}
```

然后实现第二阶段的算法 void task2_stage2(Buffer *buf, int begin, int len, int out_begin)。

如下图所示，函数参数的含义已在注释中写明。首先设置每个子集合块数为 8，然后定义 way_num 为总路归，大小为总块数除以每路的块数，定义 blk 数组用于存储每路从磁盘中读出的数据，count 数组用于记录各路目前载入的块数，out_count 记录已输出的块数，pointer 数组记录当前块读到的位置，定义并初始化 ans 为输出块，ans_pointer 为输出块指针，not_finished 数组用于判断某路是否已经处理完毕。然后读取各路的第一块，并设置各路的相关参数：

```
/* 对子集合进行归并排序，按第一个属性从小到大然后第二个属性从小到大排序
   begin为待排序关系的起始块号，
   len为待排序关系的块数，
   out_begin为排好序的关系的起始块号。
*/
void task2_stage2(Buffer *buf, int begin, int len, int out_begin)
{
    int set_blk_num = 8;  // 每个子集合8块blk
    int way_num = len/set_blk_num;  // way_num路归并排序
    Blk blk[way_num];  // 用于存储从磁盘中读出的blk中的数据
    int count[way_num];  // 记录各路目前载入的块数
    int out_count = 0;  // 记录已输出的块数
    int pointer[way_num];  // 记录当前块读到的位置
    Blk ans;  // 输出块
    init_blk(&ans);
    int ans_pointer = 0;  // 输出块指针
    int not_finished[way_num];  // 判断某路是否已经处理完毕
    int i;
    // 读取各路的第一块
    for (i = 0; i < way_num; i++){
        blk[i] = read_blk_data(begin + i * set_blk_num, buf);
        count[i] = 1;
        pointer[i] = 0;
        not_finished[i] = 1;
    }
```

然后，设置 flag 标志是否所有路都读完，当 flag 为 1 时表示还有某路未读完，为 0 时表示所有路均已读完。当有某路未读完时，持续循环。遍历每路，若某路的块读完了，则需要读入新的块，更新相关的指针和计数器，同时需要判断某路是否已经处理完毕并更新 not_finished 标志数组：

```
int flag = 1;  // 标志是否所有路都读完
while(flag){
    for (i = 0; i < way_num; i++){
        //某路的块读完了，需要读入新的块
        if(pointer[i] == 7){
            if (count[i] < set_blk_num){
                blk[i] = read_blk_data(begin + i * set_blk_num + count[i], buf);
                pointer[i] = 0;
            }
            count[i]++;
        }
        // 判断某路是否已经处理完毕
        if(count[i] > set_blk_num)
            not_finished[i] = 0;
    }
```

设置 X 和 Y 数组用于存放各路当前读到的元组（已读完的元组对应位置为空），minX 和 minY 用于保存最小的元组，way_id 为最小的元组所在的路数，cur_not_finished 用于判断当前轮是否结束（即是否有块读完）。对于每一轮，首先获取没有读完的每路的当前元组，然后调用上述定义的 get_min 函数获取最

小的元组，及其所在的路数，将最小的元组填入输入块。将对应路的指针后移，若某块读完，则需要设置 cur_not_finished 为 0，表示当前轮结束（需要进入下一轮读入新的块），并且若此时已经读到该路的最后一块时，需要设置 not_finished 数组的该路为 0，表示该路已读完。若输出块写完，则需要写回，并更新计数器和指针。最后更新 flag 标志，只要有某路未处理完，flag 就置为 1，表示需要继续循环。输出最终结果写入的磁盘块号。

```
    int minX, minY, way_id;
    int cur_not_finished = 1;  // 判断当前轮是否结束（即是否有块读完）
    while(cur_not_finished){
        for(i = 0; i < way_num; i++){
            if(not_finished[i]){
                X[i] = blk[i].X[pointer[i]];
                Y[i] = blk[i].Y[pointer[i]];
            }

        }
        way_id = get_min(X, Y, &minX, &minY, way_num, not_finished);
        ans.X[ans_pointer] = minX;
        ans.Y[ans_pointer] = minY;

        // 有一块读完
        if(++pointer[way_id] == 7){
            cur_not_finished = 0;
            if(count[way_id] == set_blk_num)  not_finished[way_id] = 0;
        }

        // 输出块写完
        if(++ans_pointer == 7){
            write_ans_blk(&ans, buf, out_begin + out_count);
            out_count++;
            ans_pointer = 0;
        }
    }

    // 判断是否所有子集合都已处理完毕
    flag = 0;
    for(i = 0; i < way_num; i++){
        if(not_finished[i]) flag = 1;
    }
}
printf("第二阶段：各子集合间归并排序，最终结果写入磁盘[%d, %d]\n\n"
    , out_begin, out_begin + len - 1);
return;
```

整体算法 void task2_tpmms(Buffer *buf)如下图所示，首先输出任务名称并初始化缓冲区。然后调用上述定义的第一阶段函数 task2_stage1 和第二阶段

函数 task2_stage2，分别对关系 R 和 S 进行两阶段多路归并排序，输出 IO 读写次数：

```
void task2_tpmms(Buffer *buf)
{
    printf("------------------------------------------------\n");
    printf("两阶段多路归并排序算法，将R和S分别排序\n");
    printf("------------------------------------------------\n");

    init_buf(buf);
    // 关系R
    printf("---------------关系R---------------\n");
    // 第一阶段，对R的子集合(2个子集合*每个子集合8块)进行排序
    // 中间结果存储在blk 201-216中
    task2_stage1(buf, 1, 16, 201);
    // 第二阶段，子集合之间进行2路归并排序，最终结果存储在blk 301-316中
    task2_stage2(buf, 201, 16, 301);

    // 关系S
    printf("---------------关系S---------------\n");
    // 第一阶段，对S的子集合(4个子集合*每个子集合8块)进行排序，
    // 中间结果存储在blk 217-248中
    task2_stage1(buf, 17, 32, 217);
    // 第二阶段，子集合之间进行4路归并排序，最终结果存储在blk 317-348中
    task2_stage2(buf, 217, 32, 317);

    printf("----------------------------------\n");
    printf("IO读写一共%ld次。\n\n\n", buf->numIO);
    return;
}
```

实验结果：

关系 R 的最终结果写入 301-316 号磁盘块，关系 S 的最终结果写入 317-348 号磁盘块。IO 读写 192 次。

```
------------------------------------------------
两阶段多路归并排序算法，将R和S分别排序
------------------------------------------------
---------------关系R---------------
第一阶段：划分子集合并在子集合内排序，中间结果写入磁盘[201, 216]
第二阶段：各子集合间归并排序，最终结果写入磁盘[301, 316]

---------------关系S---------------
第一阶段：划分子集合并在子集合内排序，中间结果写入磁盘[217, 248]
第二阶段：各子集合间归并排序，最终结果写入磁盘[317, 348]

----------------------------------
IO读写一共192次。
```

(3) 实现基于索引的关系选择算法

问题分析：

本实验需要实现的是**基于索引的关系选择算法**。需要利用任务（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=128 的元组（即模拟实现 select S.C, S.D from S where S.C = 128），并将选择结果存放在磁盘上。记录 IO 读写次数，与任务（1）中**基于线性搜索的关系选择算法**的结果对比。

实现思路是，首先对排好序的关系 R 和 S 建立索引，对每个关系的第一个属性（关系 R 为属性 A，关系 S 为属性 C）的每个不同值建立索引，形式为（属性值，块号）。此处，关系 R 的索引文件存储在 401-406 号磁盘块，关系 S 的索引文件存储在 407-412 号磁盘块。

然后，利用建立好的索引文件找出 S.C=128 的索引，从而找到对应的全部元组。最终结果存储在 120-121 号磁盘块。

首先定义建立索引函数 int build_index(Buffer *buf, int begin, int len, int out_begin)：

```
/* 对排好序的关系建立索引，对每个关系的第一个属性的不同值建立索引。
   begin为关系的起始块号，
   len为关系的块数，
   out_begin为建立好索引的关系的起始块号。
*/
int build_index(Buffer *buf, int begin, int len, int out_begin)
{
    init_buf(buf);
    Blk blk;  // 用于存储从磁盘中读出的blk中的数据
    Blk out;  // 输出索引块
    init_blk(&out);
    int out_pointer = 0;  // 输出索引块指针
    int last_X, cur_X;  // 上一个索引X，当前X
    last_X = cur_X = -1;
    int out_count = 0;  // 输出索引块数

    for(int i = begin; i < begin + len; i++){  // 读取每块
        blk = read_blk_data(i, buf);
        for(int j = 0; j < 7; j++){  // 读取每个元组
            cur_X = blk.X[j];
            if(cur_X != last_X){
                out.X[out_pointer] = cur_X;
                out.Y[out_pointer] = i;  // 块号
                out_pointer++;
                last_X = cur_X;
                // 输出索引块写满需要写回
                if(out_pointer == 7){
                    write_ans_blk(&out, buf, out_begin + out_count);
                    out_count++;
                    out_pointer = 0;
                }
            }
        }
    }
```

如上图所示，函数参数的含义已在注释中写明。首先重置缓冲区 buf，定义 blk 用于存储从磁盘中读出的 blk 中的数据,定义并初始化 out 表示输出索引块，out_pointer 表示输出索引块指针，last_X 和 cur_X 分别表示上一个索引和当前索引，初始值为-1，out_count 表示输出索引块数。然后遍历每块的每个元组，该元组的第一个属性值为当前索引值 cur_X，若当前索引值不等于上一个索引值 last_X，则需要将当前索引值和当前块号作为一个元组存入输出索引块中，更新输出索引块指针，设置上一个索引值 last_X 等于当前索引值 cur_X。若输出索引块写满则需要写回，同时更新相关计数器和指针。

当循环结束时，若最后一块输出索引块未写满则需要写回，输出存储索引文件的磁盘块号以及 IO 读写次数，返回输出索引块数量：

```
    // 最后一块输出索引块需要写回
    if(out_pointer < 7){
        write_ans_blk(&out, buf, out_begin + out_count);
        out_count++;
    }
    printf("建立索引，索引文件写入磁盘[%d, %d]\n", out_begin,
           out_begin + out_count - 1);
    printf("IO读写一共%ld次。\n\n", buf->numIO);
    return out_count;
```

然后，定义基于索引的关系选择函数 void index_select(Buffer *buf, int idx_begin, int idx_len, int ans_begin, int c, int end)：

```
/* 利用索引文件选出S.C=128的元组
   idx_begin为索引文件的起始块号，
   idx_len为索引文件的块数，
   ans_begin为输出结果的起始块号，
   c为待查找的数值，
   end为关系的结束块号。
*/
void index_select(Buffer *buf, int idx_begin, int idx_len,
                  int ans_begin, int c, int end)
{
    init_buf(buf);
    Blk blk;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 查找结果
    init_blk(&ans);
    int ans_pointer = 0;  // 结果块指针
    int cur_blk_id;  // 当前索引指向的块号
    int ans_blk_count = 0;  // 结果块数
    int ans_count = 0;  // 结果元组数
    int find = 0;  // 是否查找到

    int i, j;
    // 读取每块索引块
    for(i = idx_begin; i < idx_begin + idx_len; i++){
        blk = read_blk_data(i, buf);
        printf("读入索引块%d\n", i);
        for(j = 0; j < 7; j++){  // 读取每个元组
            if(blk.X[j] == c){
                cur_blk_id = blk.Y[j];
                find = 1;
                break;
            }
        }
        if(find) break;
    }
    if(!find){
        printf("未查找到该元组!\n");
        return;
    }
```

如上图所示，函数参数的含义已在注释中写明。首先重置缓冲区 buf，定义 blk 用于存储从磁盘中读出的 blk 中的数据，定义并初始化 ans 用于存储查找结

果，定义 ans_pointer 表示结果块指针，cur_blk_id 表示当前索引指向的块号，ans_blk_count 表示结果块数，ans_count 表示结果元组数，find 标志是否查找到对应元组。遍历关系 S 的全部索引块的每个元组，查找等于待查找属性值的索引，找到该属性值对应的开始块号 cur_blk_id，若找到则跳出循环。若最终未找到则输出提示信息并返回。

```
int is_finished = 0;
for(i = cur_blk_id; i <= end; i++){
    blk = read_blk_data(i, buf);
    printf("读入数据块%d\n", i);
    for(j = 0; j < 7; j++){
        if(blk.X[j] == c){
            ans.X[ans_pointer] = blk.X[j];
            ans.Y[ans_pointer] = blk.Y[j];
            printf("(X = %d, Y = %d)\n", blk.X[j], blk.Y[j]);
            ans_pointer++;
            ans_count++;
            // 结果块写满需要写回
            if(ans_pointer == 7){
                write_ans_blk(&ans, buf, ans_begin + ans_blk_count);
                ans_blk_count++;
                ans_pointer = 0;
            }
        }
        if(blk.X[j] > c){
            is_finished = 1;
            break;
        }
    }
    if(is_finished) break;
}
// 最后一块结果块需要写回
if(ans_pointer < 7){
    write_ans_blk(&ans, buf, ans_begin + ans_blk_count);
    ans_blk_count++;
}
printf("注：结果写入磁盘：[%d, %d]\n\n", ans_begin,
       ans_begin + ans_blk_count - 1);
printf("满足选择条件的元组一共%d个\n\n", ans_count);
printf("IO读写一共%ld次\n\n", buf->numIO);

return;
```

如上图所示，然后定义 is_finished 标志是否已经查找到所有满足条件的元组。遍历 cur_blk_id 开始的所有磁盘数据块中的所有元组，当第一个属性值等于待查找值时，则将其写入结果块，同时输出元组属性值并更新相关指针和计数器。若结果块写满则需要写回，并更新相关指针和计数器。当第一个属性值大于待查找值时，设置 is_finished 标志为 1，表示已经查找到所有满足条件的元组。

当循环结束后，若最后一块结果块未写满则也需要写回并更新计数器。最后输出存储结果块的磁盘块号、满足选择条件的元组个数以及 IO 读写次数。

最后，整体算法 void task3_indexSelect(Buffer *buf)如下图所示，首先输出任务名称及关系信息。然后调用上述定义的 build_index 函数建立关系 R 和关系 S 的索引，调用上述定义的 index_select 函数查找关系 S 中，属性 C 等于 128 的元组：

```
void task3_indexSelect(Buffer *buf)
{
    printf("------------------------------------------------\n");
    printf("基于索引的关系选择算法 S.C=128\n");
    printf("------------------------------------------------\n");

    printf("---------------关系R---------------\n");
    // 对关系R建立索引，索引文件存储在blk 401-406中
    int r_len = build_index(buf, 301, 16, 401);

    printf("---------------关系S---------------\n");
    // 对关系S建立索引，索引文件存储在blk 407-412中
    int s_len = build_index(buf, 317, 32, 401 + r_len);

    printf("---------查找S.C=128的元组----------\n");
    index_select(buf, 401 + r_len, s_len, 120, 128, 348);

    return;
}
```

实验结果：

关系 R 的索引文件存储在 401-406 号磁盘块，建立索引共需 IO 读写 22 次；关系 S 的索引文件存储在 407-412 号磁盘块，建立索引共需 IO 读写 38 次。

基于索引的关系选择算法查找 S.C=128 的元组，最终结果存储在 120-121 号磁盘块，满足选择条件的元组一共 10 个，IO 读写一共 6 次。

对于关系 S，**基于索引的关系选择算法** IO 读写一共 6 次，远远少于**基于线性搜索的关系选择算法** IO 读写一共 34 次，说明**基于索引的关系选择算法**更加高效。

```
------------------------------------------------
基于索引的关系选择算法 S.C=128
------------------------------------------------
---------------关系R---------------
建立索引，索引文件写入磁盘[401, 406]
IO读写一共22次。

---------------关系S---------------
建立索引，索引文件写入磁盘[407, 412]
IO读写一共38次。

---------查找S.C=128的元组----------
读入索引块407
没有满足条件的元组。
读入索引块408
读入数据块324
(X = 128, Y = 429)
(X = 128, Y = 431)
(X = 128, Y = 447)
(X = 128, Y = 457)
(X = 128, Y = 584)
(X = 128, Y = 592)
(X = 128, Y = 615)
读入数据块325
(X = 128, Y = 684)
(X = 128, Y = 720)
(X = 128, Y = 871)
注：结果写入磁盘：[120, 121]

满足选择条件的元组一共10个

IO读写一共6次
```

(4) 实现基于排序的连接操作算法（Sort-Merge-Join）

问题分析：

本实验需要实现的是**基于排序的连接操作算法（Sort-Merge-Join）**。需要对关系 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。也即模拟实现 select S.C, S.D, R.A, R.B from S inner join R on S.C = R.A。

实现思路是，分别用两个指针遍历关系 R 和关系 S，若 R.A > S.C，则 S 的指针后移，若 R.A < S.C，则 R 的指针后移，直到 R.A = S.C。此时，用另一个新的指针遍历关系 R，找到属性 A 和当前的 R.A 相等的最后一个元组，将关系 R 的两个指针之间的所有元组和当前关系 S 的指针指向的元组连接并写入结果块

中。

对于连接之后的元组，一个元组占四个属性值，按照每块 3 个连接结果进行存储，即**每块存储 6 个元组**，最后一个元组位置置空。最终结果存入 501-630 号磁盘块。

在实现整体算法流程前，首先定义函数 void check_and_read(Buffer *buf, int *count, int *not_end, int blk_id_start, int blk_id_end, Blk *blk) 用于读取关系的下一个元组，并更新相关计数器和指针：

```
/* 判断是否读完，若未读完则读入新块
   count为关系指针，指向该关系遍历到的元组个数
   not_end标志是否有关系已遍历完
   blk_id_start表示关系的开始块数
   blk_id_end表示关系的结束块数
   blk用于存储从磁盘中读出的blk中的数据
 */
void check_and_read(Buffer *buf, int *count, int *not_end,
                    int blk_id_start, int blk_id_end, Blk *blk)
{
    (*count)++;
    if(*count % 7 == 0) {
        if(blk_id_start + *count / 7 > blk_id_end) *not_end = 0;
        else *blk = read_blk_data(blk_id_start + *count / 7, buf);
    }
}
```

如上图所示，函数参数的含义已在注释中写明。首先更新作为关系指针的计数器 count，当指针指向当前块的最后一个元组时，若此时已经读取到该关系的最后一块，则将 not_end 标志为已结束，否则读取下一块的第一个元组。其中，*count / 7 表示当前元组是该关系中的第几块，加上初始块号 blk_id_start 即可得到当前元组所在的块号。

然后，实现该算法 void task4_sortMergeJoin(Buffer *buf)的主体流程，首先输出任务名称并初始化缓冲区 buf。定义 blk_R 和 blk_S 用于存储从磁盘中读出的 blk 中的数据，其中 blk_R 是长度为 6 的数组（因为缓冲区中共能存下 8 个数据块，1 个用于存储关系 S 的当前块，一个作为输出块，其余 6 个则存储关系 R 的数据块，以防关系 R 中有多个块中的元组的第一个属性值相等）。定义并初始化 ans 用于存储连接结果，blk_id_R_start 和 blk_id_S_start 表示两关系

的起始块号，blk_id_R_end 和 blk_id_S_end 表示两关系的结束块号，count_R 和 count_S 分别是遍历两关系的指针，表示当前是该关系的第几个元组，count_R_new 是关系 R 的另一个指针，用于查找第一个属性值相等的所有元组，pointer_ans 为结果块指针，not_end 标志是否有关系已遍历完，count_ans 计算连接次数，count_ans_blk 计算结果块数，ans_begin 表示结果的起始块号，last_C 表示上一个查找到的相等的属性值。读入关系 R 和 S 的第一个块：

```
void task4_sortMergeJoin(Buffer *buf)
{
    printf("-------------------------------------------------\n");
    printf("基于排序的连接算法\n");
    printf("-------------------------------------------------\n");
    init_buf(buf);

    Blk blk_R[6], blk_S;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 连接结果
    init_blk(&ans);
    int blk_id_R_start, blk_id_S_start, blk_id_R_end, blk_id_S_end;
    blk_id_R_start = 301;  // 关系R（有序）的起始块号
    blk_id_R_end = 316;  // 关系R（有序）的结束块号
    blk_id_S_start = 317;  // 关系S（有序）的起始块号
    blk_id_S_end = 348;  // 关系S（有序）的结束块号
    /* 遍历过程对于关系R需要两个指针，对于关系S需要一个指针，
    表示当前是该关系的第几个元组 */
    int count_R, count_R_new, count_S;
    count_R = count_R_new = count_S = 0;
    int pointer_ans = 0;  // 结果块指针
    int not_end = 1;  // 标志是否有关系已遍历完
    int count_ans = 0;  // 连接次数
    int count_ans_blk = 0;  // 结果块数
    int ans_begin = 501;  // 结果的起始块号
    int last_C = 0;

    blk_R[0] = read_blk_data(blk_id_R_start, buf);
    blk_S = read_blk_data(blk_id_S_start, buf);
```

然后进入循环。首先遍历两个关系，直到 S.C = R.A。其中 count_S/R % 7 表示当前元组在其所在块中是第几个元组。遍历时，只有关系 S 的当前元组的第一个属性与上一轮选择的属性不同时，才需要重新遍历两关系以及查找关系 R 中所有第一个属性相等的元组，当关系 S 的当前元组的第一个属性与上一轮选择的属性相同时，则不需要进行该操作，可以直接将关系 S 的当前元组与上一轮框选出的关系 R 中的元组进行连接并写回，这样可以大大提高算法的效率。当 not_end 标志已经有关系读完时，跳出循环。此处的遍历，对于关系 R 均使用 blk_R 数组

中的第一个位置存储读到的块。然后当查找到 S.C = R.A 之后，通过新的指针 count_R_new 向后遍历关系 R（count_R_new 从 0 开始计数，计算最后一个相等的元组与原指针指向的元组之间的距离），查找所有和 S.C 相等的 R.A 对应的元组，将这些元组对应的块依次存入 blk_R 数组的其余位置中。其中(count_R % 7 + count_R_new) / 7 计算的是新指针遍历到的当前元组在 blk_R 数组中存储的位置下标，(count_R + count_R_new) % 7 计算的是新指针遍历到的当前元组在其所在块中是第几个元组，(blk_id_R_start + (count_R + count_R_new) / 7 计算的是新指针遍历到的当前元组的磁盘块号。最后，设置 last_C 等于遍历到的关系 S 的当前元组的第一个属性值：

```
while(1){
    // 遍历两个关系，直到S.C = R.A
    if(blk_S.X[count_S % 7] != last_C) {
        while(not_end && blk_S.X[count_S % 7] != blk_R[0].X[count_R % 7]){
            while(not_end && blk_S.X[count_S % 7] < blk_R[0].X[count_R % 7])
                check_and_read(buf, &count_S, &not_end, blk_id_S_start,
                               blk_id_S_end, &blk_S);
            while(not_end && blk_S.X[count_S % 7] > blk_R[0].X[count_R % 7])
                check_and_read(buf, &count_R, &not_end, blk_id_R_start,
                               blk_id_R_end, &blk_R[0]);
        }
        if(!not_end) break;

        // 查找所有和S.C相等的R.A
        while(blk_R[0].X[count_R % 7] == blk_R[(count_R % 7 + count_R_new) / 7]
              .X[(count_R + count_R_new) % 7]){
            count_R_new++;
            if((count_R + count_R_new) % 7 == 0) {
                if(blk_id_R_start + (count_R + count_R_new) / 7 > blk_id_R_end)
                    break;
                else blk_R[(count_R % 7 + count_R_new) / 7] = read_blk_data
                (blk_id_R_start + (count_R + count_R_new) / 7, buf);
            }
        }
    }
    last_C = blk_S.X[count_S % 7];
```

此时的 count_R_new 则表示关系 R 中共有多少个第一个属性相等的元组。遍历这些元组，将遍历到的元组与关系 S 中指向的当前元组连接，并写入结果块，同时更新结果块指针。当结果块写满时需要写回，并更新相关计数器和指针。遍历结束后，关系 S 的指针后移。只有当关系 S 的当前元组的第一个属性值与上一轮的第一个属性值相同时，才需要将 count_R_new 计数器置零，否则不用置零，

直接供下一轮使用。当两个关系全部连接结束后，若最后一块为写满，也需要写回。输出结果写入的磁盘块号以及总共的连接次数：

```
        // 将S的当前元组和所有查找到的R中的元组连接并写回
        for(int i = 0; i < count_R_new; i++){
            ans.X[pointer_ans] = blk_S.X[count_S % 7];
            ans.Y[pointer_ans] = blk_S.Y[count_S % 7];
            pointer_ans++;
            ans.X[pointer_ans] = blk_R[(count_R % 7 + i) / 7].X[(count_R + i) % 7];
            ans.Y[pointer_ans] = blk_R[(count_R % 7 + i) / 7].Y[(count_R + i) % 7];
            pointer_ans++;
            count_ans++;
            if(pointer_ans == 6){  // 结果块写满需要写回
                write_ans_blk(&ans, buf, ans_begin + count_ans_blk);
                count_ans_blk++;
                pointer_ans = 0;
            }
        }
        check_and_read(buf, &count_S, &not_end, blk_id_S_start, blk_id_S_end, &blk_S);
        if(blk_S.X[count_S % 7] != last_C) count_R_new = 0;
    }
    // 最后一块也需要写回
    if(pointer_ans < 6){
        write_ans_blk(&ans, buf, ans_begin + count_ans_blk);
        count_ans_blk++;
    }
    printf("注：结果写入磁盘：[%d, %d]\n\n",
           ans_begin, ans_begin + count_ans_blk - 1);
    printf("总共连接%d次\n\n", count_ans);

    return;
```

实验结果：

最终结果写入 501-630 号磁盘，总共连接 389 次。

```
--------------------------------------------------
基于排序的连接算法
--------------------------------------------------
注：结果写入磁盘：[501, 630]

总共连接389次
```

(5) 实现基于排序或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：

本实验需要实现的是**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。

此处，我实现的是**基于排序的两趟扫描并（S∪R）算法**。

基本思路是，用两个指针分别遍历关系 R 和 S，直到两指针指向的元组相等。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 中的元组写入结果块并读入新的；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 中的元组写入结果块并读入新的；若只剩 S 未被读完，则将 S 中的其余元组全部写入结果块；若只剩 R 未被读完，则将 R 中的其余元组全部写入结果块。最后，当 S 中的元组等于 R 中的元组时，写入 S 中的元组，且 R 和 S 均读入新的，进入下一轮循环继续遍历。

最终结果存入 701-747 号磁盘块。

首先，实现判等函数 int equal(int X1, int Y1, int X2, int Y2)，若元组(X1, Y1)等于(X2, Y2)（X1 = X2 且 Y1 = Y2），则返回 1；否则返回 0：

```
// 若元组(X1, Y1)等于(X2, Y2)，则返回1；否则返回0
int equal(int X1, int Y1, int X2, int Y2)
{
    if(X1 == X2 && Y1 == Y2) return 1;
    else return 0;
}
```

然后，封装写入结果和写回磁盘的函数 void write_and_check(Buffer *buf, Blk *ans, Blk *blk, int count, int *pointer_ans, int *count_ans, int ans_begin, int *count_ans_blk)：

```
/* 写入结果，并判断是否需要写回
   ans是结果块，blk中存储该关系的当前块，count计数当前元组是关系中的第几个元组，
   pointer_ans是结果块指针，count_ans是结果计数器，ans_begin是结果起始块号，
   count_ans_blk是结果块数计数器
   */
void write_and_check(Buffer *buf, Blk *ans, Blk *blk, int count, int *pointer_ans
                     , int *count_ans, int ans_begin, int *count_ans_blk)
{
    ans->X[*pointer_ans] = blk->X[count % 7];
    ans->Y[*pointer_ans] = blk->Y[count % 7];
    (*count_ans)++;
    (*pointer_ans)++;
    if(*pointer_ans == 7){  // 结果块写满需要写回
        write_ans_blk(ans, buf, ans_begin + *count_ans_blk);
        (*count_ans_blk)++;
        *pointer_ans = 0;
    }
}
```

如上图所示，函数参数的含义已在注释中写明。首先，将指针指向的当前元组写入结果块，并更新相关计数器和指针，其中 count % 7 表示当前元组在其所在块中的位置。当结果块写满时需要写回磁盘，并更新相关计数器和指针。由于这段代码在集合的交、并、差等操作中均多次出现，因此将它们封装起来可以减少代码重复。

然后，实现算法 void task5_sortUnion(Buffer *buf)的主体流程，首先输出任务名称并初始化缓冲区 buf。定义 blk_R 和 blk_S 用于存储从磁盘中读出的 blk 中的数据，定义并初始化 ans 用于存储集合操作结果，定义 blk_id_R_start 和 blk_id_S_start 表示两关系的起始块号，blk_id_R_end 和 blk_id_S_end 表示两关系的结束块号，count_R 和 count_S 分别是遍历两关系的指针，表示当前是该关系的第几个元组，pointer_ans 为结果块指针，not_end_R 和 not_end_S 分别标志两关系是否已遍历完，count_ans 计算并操作之后的元组个数，count_ans_blk 计算结果块数，ans_begin 表示结果的起始块号。读入关系 R 和 S 的第一个块：

```
void task5_sortUnion(Buffer *buf)
{
    printf("----------------------------------------------------\n");
    printf("基于排序的集合并算法\n");
    printf("----------------------------------------------------\n");
    init_buf(buf);

    Blk blk_R, blk_S;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 集合操作结果
    init_blk(&ans);
    int blk_id_R_start, blk_id_S_start, blk_id_R_end, blk_id_S_end;
    blk_id_R_start = 301;  // 关系R（有序）的起始块号
    blk_id_R_end = 316;  // 关系R（有序）的结束块号
    blk_id_S_start = 317;  // 关系S（有序）的起始块号
    blk_id_S_end = 348;  // 关系S（有序）的结束块号
    // 遍历过程对于关系R和S需要一个指针，表示当前是该关系的第几个元组
    int count_R, count_S;
    count_R = count_S = 0;
    int pointer_ans = 0;  // 结果块指针
    int not_end_R, not_end_S;
    not_end_R = not_end_S = 1;
    int count_ans = 0;  // 元组个数
    int count_ans_blk = 0;  // 结果块数
    int ans_begin = 701;  // 结果的起始块号

    blk_R = read_blk_data(blk_id_R_start, buf);
    blk_S = read_blk_data(blk_id_S_start, buf);
```

然后，进入循环。对于每轮循环，持续遍历直到两个关系都被读完，或者调用上述 equal 函数得到关系 R 的当前元组与关系 S 的当前元组相等，其中 count_R/S % 7 表示当前元组在其所在块中的位置。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 中的当前元组写入结果块并且 S 的指针后移；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 中的元组写入结果块并且 R 的指针后移。其中，调用上述封装的 write_and_check 函数用于将当前元组写入结果块，并判断是否需要写回；调用上一实验中封装的 check_and_read 函数用于判断是否读完，若未读完则读入新块。若 R 已被读完而 S 未被读完，则将 S 中的其余元组全部写入结果块；若 S 已被读完而 R 未被读完，则将 R 中的其余元组全部写入结果块。同样调用上述定义的 write_and_check 函数和 check_and_read 函数实现。

```
while(1){
    while((not_end_R || not_end_S) && !equal(blk_R.X[count_R % 7],
        blk_R.Y[count_R % 7], blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
        // 关系S未被读完，且R中的元组大于S中的元组，写入S中的元组并读入新的
        while(not_end_S && larger(blk_R.X[count_R % 7], blk_R.Y[count_R % 7]
                                , blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
            write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                            &count_ans, ans_begin, &count_ans_blk);
            check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                           blk_id_S_end, &blk_S);
        }
        // 关系R未被读完，且S中的元组大于R中的元组，写入R中的元组并读入新的
        while(not_end_R && larger(blk_S.X[count_S % 7], blk_S.Y[count_S % 7]
                                , blk_R.X[count_R % 7], blk_R.Y[count_R % 7])){
            write_and_check(buf, &ans, &blk_R, count_R, &pointer_ans,
                            &count_ans, ans_begin, &count_ans_blk);
            check_and_read(buf, &count_R, &not_end_R, blk_id_R_start,
                           blk_id_R_end, &blk_R);
        }
        // 只剩S未被读完
        if(not_end_S && !not_end_R){
            while(not_end_S){
                write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                                &count_ans, ans_begin, &count_ans_blk);
                check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                               blk_id_S_end, &blk_S);
            }
            break;
        }
        // 只剩R未被读完
        if(not_end_R && !not_end_S){
            while(not_end_R){
                write_and_check(buf, &ans, &blk_R, count_R, &pointer_ans,
                                &count_ans, ans_begin, &count_ans_blk);
                check_and_read(buf, &count_R, &not_end_R, blk_id_R_start,
                               blk_id_R_end, &blk_R);
            }
            break;
        }
    }
```

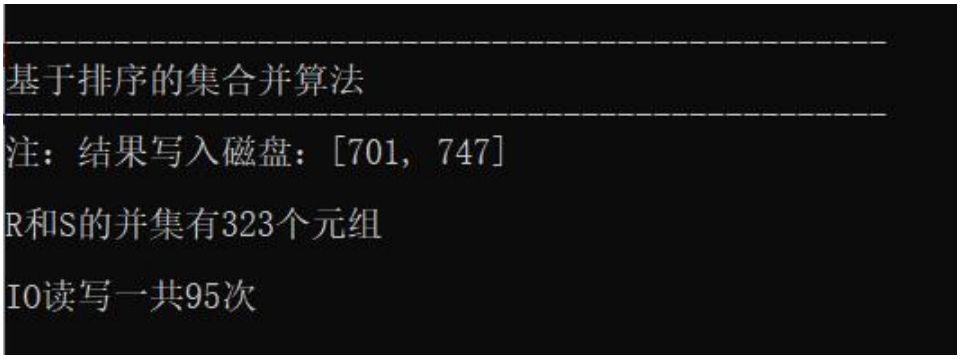
最后，当关系 S 中的元组等于关系 R 中的元组时，调用 write_and_check 函数将其中一个元组写入结果块，且调用 check_and_read 函数使得关系 R 和 S 的指针均向后移，进入下一轮循环继续遍历。如果关系 R 和 S 均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的并集元组个数以及 IO 读写次数：

```
    // 当S中的元组等于R中的元组，写入S中的元组，且R和S均读入新的
    if(equal(blk_R.X[count_R % 7], blk_R.Y[count_R % 7],
             blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
        write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                        &count_ans, ans_begin, &count_ans_blk);
        check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                       blk_id_S_end, &blk_S);
        check_and_read(buf, &count_R, &not_end_R, blk_id_R_start,
                       blk_id_R_end, &blk_R);
    }
    if(!not_end_R && !not_end_S) break;
}
// 最后一块也需要写回
if(pointer_ans < 6){
    write_ans_blk(&ans, buf, ans_begin + count_ans_blk);
    count_ans_blk++;
}
printf("注：结果写入磁盘：[%d, %d]\n\n", ans_begin,
       ans_begin + count_ans_blk - 1);
printf("R和S的并集有%d个元组\n\n", count_ans);
printf("IO读写一共%ld次\n\n", buf->numIO);

return;
```

实验结果：

最终结果写入 701-747 号磁盘，R 和 S 的并集有 323 个元组，IO 读写一共 95 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。

（1）基于排序的两趟扫描交（S∩R）算法

问题分析：本实验需要实现的是**基于排序的两趟扫描交（S∩R）算法**，将结果存放在磁盘上，并统计交操作后的元组个数。

基本思路与并算法类似，不同的是，此处只在关系 R 和关系 S 中的元组相等时才将元组写入结果块。用两个指针分别遍历关系 R 和 S，直到两指针指向的元组相等。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 的指针后移；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 中的指针后移；若只剩 S 或只剩 R，均结束。最后，当 S 中的元组等于 R 中的元组时，才将该元组写入结果块，且 R 和 S 的指针均后移，进入下一轮循环继续遍历。

最终结果存入 801-802 号磁盘块。

实现算法 void task5_sortIntersection(Buffer *buf)的主体流程，首先输出任务名称并初始化缓冲区 buf。变量的定义与并算法中的一致，便不再赘述，读入关系 R 和 S 的第一个块：

```
void task5_sortIntersection(Buffer *buf)
{
    printf("-------------------------------------------------\n");
    printf("基于排序的集合交算法\n");
    printf("-------------------------------------------------\n");
    init_buf(buf);

    Blk blk_R, blk_S;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 集合操作结果
    init_blk(&ans);
    int blk_id_R_start, blk_id_S_start, blk_id_R_end, blk_id_S_end;
    blk_id_R_start = 301;  // 关系R（有序）的起始块号
    blk_id_R_end = 316;  // 关系R（有序）的结束块号
    blk_id_S_start = 317;  // 关系S（有序）的起始块号
    blk_id_S_end = 348;  // 关系S（有序）的结束块号
    // 遍历过程对于关系R和S需要一个指针，表示当前是该关系的第几个元组
    int count_R, count_S;
    count_R = count_S = 0;
    int pointer_ans = 0;  // 结果块指针
    int not_end_R, not_end_S;
    not_end_R = not_end_S = 1;
    int count_ans = 0;  // 元组个数
    int count_ans_blk = 0;  // 结果块数
    int ans_begin = 801;  // 结果的起始块号

    blk_R = read_blk_data(blk_id_R_start, buf);
    blk_S = read_blk_data(blk_id_S_start, buf);
```

然后，进入循环。对于每轮循环，持续遍历直到两个关系都被读完，或者调用上述 equal 函数得到关系 R 的当前元组与关系 S 的当前元组相等，其中 count_R/S % 7 表示当前元组在其所在块中的位置。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 的指针后移；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 的指针后移。其中，调用上一实验中封装的 check_and_read 函数用于判断是否读完，若未读完则读入新块。若只剩 S 或只剩 R 未被读完，均设置另一关系也结束，从而该轮的遍历结束：

```
while(1){
    while((not_end_R || not_end_S) && !equal(blk_R.X[count_R % 7],
        blk_R.Y[count_R % 7], blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
        // 关系S未被读完，且R中的元组大于S中的元组，读入新的S中的元组
        while(not_end_S && larger(blk_R.X[count_R % 7], blk_R.Y[count_R % 7],
                                  blk_S.X[count_S % 7], blk_S.Y[count_S % 7]))
            check_and_read(buf, &count_S, &not_end_S,
                           blk_id_S_start, blk_id_S_end, &blk_S);
        // 关系R未被读完，且S中的元组大于R中的元组，读入新的R中的元组
        while(not_end_R && larger(blk_S.X[count_S % 7], blk_S.Y[count_S % 7],
                                  blk_R.X[count_R % 7], blk_R.Y[count_R % 7]))
            check_and_read(buf, &count_R, &not_end_R,
                           blk_id_R_start, blk_id_R_end, &blk_R);
        // 只剩S未被读完
        if(not_end_S && !not_end_R) {
            not_end_S = 0;
            break;
        }
        // 只剩R未被读完
        if(not_end_R && !not_end_S) {
            not_end_R = 0;
            break;
        }
    }
```

最后，当关系 S 中的元组等于关系 R 中的元组时，输出该元组的值，调用 write_and_check 函数将其中一个元组写入结果块，同时调用 check_and_read 函数使得关系 R 和 S 的指针均向后移，进入下一轮循环继续遍历。如果关系 R 和 S 均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的交集元组个数以及 IO 读写次数：

```
    // 当S中的元组等于R中的元组，写入S中的元组，且R和S均读入新的
    if(equal(blk_R.X[count_R % 7], blk_R.Y[count_R % 7],
             blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
        printf("(X=%d, Y=%d)\n", blk_R.X[count_R % 7], blk_R.Y[count_R % 7]);
        write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                        &count_ans, ans_begin, &count_ans_blk);
        check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                       blk_id_S_end, &blk_S);
        check_and_read(buf, &count_R, &not_end_R, blk_id_R_start,
                       blk_id_R_end, &blk_R);
    }
    if(!not_end_R && !not_end_S) break;
}
// 最后一块也需要写回
if(pointer_ans < 7){
    write_ans_blk(&ans, buf, ans_begin + count_ans_blk);
    count_ans_blk++;
}
printf("注：结果写入磁盘：[%d, %d]\n\n", ans_begin,
       ans_begin + count_ans_blk - 1);
printf("R和S的交集有%d个元组\n\n", count_ans);
printf("IO读写一共%ld次\n\n", buf->numIO);
```

实验结果：

最终结果写入 801-802 号磁盘，R 和 S 的交集有 13 个元组，IO 读写一共 37 次。

```
--------------------------------------------------
基于排序的集合交算法
--------------------------------------------------
(X=120, Y=418)
(X=120, Y=827)
(X=122, Y=546)
(X=123, Y=477)
(X=125, Y=886)
(X=127, Y=767)
(X=128, Y=447)
(X=129, Y=430)
(X=130, Y=436)
(X=130, Y=656)
(X=134, Y=437)
(X=139, Y=461)
(X=140, Y=610)
注：结果写入磁盘：[801, 802]

R和S的交集有13个元组

IO读写一共37次
```

（2）基于排序的两趟扫描差（S-R）算法

问题分析：

本实验需要实现的是**基于排序的两趟扫描差（S-R）算法**，将结果存放在磁盘上，并统计差操作后的元组个数。

基本思路也与并算法类似，不同的是，此处只在关系 R 中没有而关系 S 中有的元组相等时才将元组写入结果块。用两个指针分别遍历关系 R 和 S，直到两指针指向的元组相等。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 中的当前元组写入结果块，并且 S 的指针后移；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 中的指针后移；若只剩 S 未被读完，则将 S 中剩余的元组全部写入结果块，并且结束当前轮的遍历；若只剩 R 未被读完，均直接结束当前轮的遍历。最后，当 S 中的元组等于 R 中的元组时，R 和 S 的指针均直接后移，进入下一轮循环继续遍历。

最终结果存入 901-931 号磁盘块。

实现算法 void task5_sortMinus(Buffer *buf)的主体流程，首先输出任务名称并初始化缓冲区 buf。变量的定义与并算法中的一致，便不再赘述，读入关系 R 和 S 的第一个块：

```
void task5_sortMinus(Buffer *buf)
{
    printf("------------------------------------------------\n");
    printf("基于排序的集合差算法\n");
    printf("------------------------------------------------\n");
    init_buf(buf);

    Blk blk_R, blk_S;  // 用于存储从磁盘中读出的blk中的数据
    Blk ans;  // 集合操作结果
    init_blk(&ans);
    int blk_id_R_start, blk_id_S_start, blk_id_R_end, blk_id_S_end;
    blk_id_R_start = 301;  // 关系R（有序）的起始块号
    blk_id_R_end = 316;  // 关系R（有序）的结束块号
    blk_id_S_start = 317;  // 关系S（有序）的起始块号
    blk_id_S_end = 348;  // 关系S（有序）的结束块号
    // 遍历过程对于关系R和S需要一个指针，表示当前是该关系的第几个元组
    int count_R, count_S;
    count_R = count_S = 0;
    int pointer_ans = 0;  // 结果块指针
    int not_end_R, not_end_S;
    not_end_R = not_end_S = 1;
    int count_ans = 0;  // 元组个数
    int count_ans_blk = 0;  // 结果块数
    int ans_begin = 901;  // 结果的起始块号

    blk_R = read_blk_data(blk_id_R_start, buf);
    blk_S = read_blk_data(blk_id_S_start, buf);
```

然后，进入循环。对于每轮循环，持续遍历直到两个关系都被读完，或者调用上述 equal 函数得到关系 R 的当前元组与关系 S 的当前元组相等，其中 count_R/S % 7 表示当前元组在其所在块中的位置。若关系 S 未被读完，且 R 中的元组大于 S 中的元组，则将 S 中的当前元组写入结果块，并且 S 的指针后移；若关系 R 未被读完，且 S 中的元组大于 R 中的元组，则将 R 中的指针直接后移。其中，调用上述封装的 write_and_check 函数用于将当前元组写入结果块，并判断是否需要写回；调用上一实验中封装的 check_and_read 函数用于判断是否读完，若未读完则读入新块。若只剩 S 未被读完，则同样调用 write_and_check 函数和 check_and_read 函数，将 S 中剩余的元组全部写入结果块，并且结束当前轮的遍历；若只剩 R 未被读完，均直接结束当前轮的遍历：

```
while(1){
    while((not_end_R || not_end_S) && !equal(blk_R.X[count_R % 7],
        blk_R.Y[count_R % 7], blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
        // 关系S未被读完，且R中的元组大于S中的元组，写入S中的元组并读入新的
        while(not_end_S && larger(blk_R.X[count_R % 7], blk_R.Y[count_R % 7],
                                  blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
            write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                            &count_ans, ans_begin, &count_ans_blk);
            check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                           blk_id_S_end, &blk_S);
        }
        // 关系R未被读完，且S中的元组大于R中的元组，读入新的R中的元组
        while(not_end_R && larger(blk_S.X[count_S % 7], blk_S.Y[count_S % 7],
                                  blk_R.X[count_R % 7], blk_R.Y[count_R % 7]))
            check_and_read(buf, &count_R, &not_end_R, blk_id_R_start,
                           blk_id_R_end, &blk_R);
        // 只剩S未被读完
        if(not_end_S && !not_end_R) {
            while(not_end_S){
                write_and_check(buf, &ans, &blk_S, count_S, &pointer_ans,
                                &count_ans, ans_begin, &count_ans_blk);
                check_and_read(buf, &count_S, &not_end_S, blk_id_S_start,
                               blk_id_S_end, &blk_S);
            }
            break;
        }
        // 只剩R未被读完
        if(not_end_R && !not_end_S) {
            not_end_R = 0;
            break;
        }
    }
```

最后，当关系 S 中的元组等于关系 R 中的元组时，调用 check_and_read 函数使得关系 R 和 S 的指针均直接后移，进入下一轮循环继续遍历。如果关系 R 和 S 均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的差集元组个数以及 IO 读写次数：

```
        // 当S中的元组等于R中的元组，R和S均读入新的
        if(equal(blk_R.X[count_R % 7], blk_R.Y[count_R % 7],
                 blk_S.X[count_S % 7], blk_S.Y[count_S % 7])){
            check_and_read(buf, &count_S, &not_end_S,
                           blk_id_S_start, blk_id_S_end, &blk_S);
            check_and_read(buf, &count_R, &not_end_R,
                           blk_id_R_start, blk_id_R_end, &blk_R);
        }
        if(!not_end_R && !not_end_S) break;
    }
    // 最后一块也需要写回
    if(pointer_ans < 7){
        write_ans_blk(&ans, buf, ans_begin + count_ans_blk);
        count_ans_blk++;
    }
    printf("注：结果写入磁盘：[%d, %d]\n\n", ans_begin,
           ans_begin + count_ans_blk - 1);
    printf("S和R的差集(S-R)有%d个元组\n\n", count_ans);
    printf("IO读写一共%ld次\n\n", buf->numIO);

    return;
```

实验结果：

最终结果写入 901-931 号磁盘，R 和 S 的并集有 211 个元组，IO 读写一共 79 次。

```
--------------------------------------------------
基于排序的集合差算法
--------------------------------------------------
注：结果写入磁盘：[901, 931]

S和R的差集(S-R)有211个元组

IO读写一共79次
```

六、 总结

总结本次实验的遇到并解决的问题、收获及反思。

在本次实验中，我们借助 ExtMem 程序库，使用 C 语言对 SQL 的查询处理算法进行了模拟，实现了关系的选择、排序、连接，集合的交、并、差等操作。通过代码实现，我对于索引和散列有了更加具体的认识，对算法 IO 的复杂性也有了一定的了解，知道了索引和散列在部分情况下都能大大降低算法的复杂性。

算法本身的原理并不困难，但由于内存有限，不能将整个关系全部读入内存中，因此需要对算法进行一些修改，比如将直接排序改为归并排序，设计归并的路数等。内存的限制使得每个关系中的数据需要分批读入，因此需要增加一些对于边界情况的处理，这使得我在编写代码时遇到了一些问题，比如对关系中的数据越界访问，最后一块结果块忘记写回磁盘、处理数据时结束标志更新不当导致死循环等等。

此外，为方便算法的实现，我还设计了数据结构用来存储从磁盘中读取并解析后的中间结果。并且封装了对数据块的读取和写回操作。事实上，在编写代码的过程中，大部分封装的子函数并不是在实现主函数之前实现的，而是在实现主函数的过程中，将那些经常重复出现的部分抽取出来，封装成子函数，这给我之后的编程也提供了一些参考。