|  |
| --- |
| 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 | |

# 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。

# 实验环境

*阐述本次实验的环境。*

Windows 7 操作系统、CodeBlocks。

# 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。

# 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**

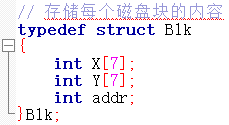
问题分析：

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。

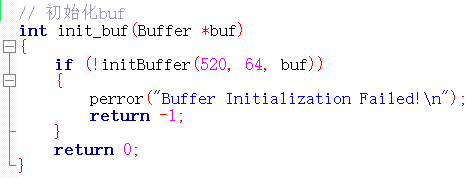
实现思路是，依次读入关系S的每个数据块，对于每个数据块，判断其中存储的7个元组是否有满足条件的。若有，则写入结果块，当结果块中写满7个元组时，将其写回磁盘。当查找结束后，若最后一块未被写满，也需要写回磁盘。最后，记录结果写入的磁盘块号、满足选择条件的元组个数和IO读写次数。

从第100号磁盘块开始存储结果块。

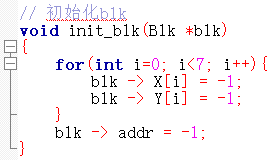
为实现该算法以及其余算法，需要首先定义中间存储块的数据结构Blk，用数组X和Y存储七个元组的两个字段，用addr存储该块的后继磁盘块号：



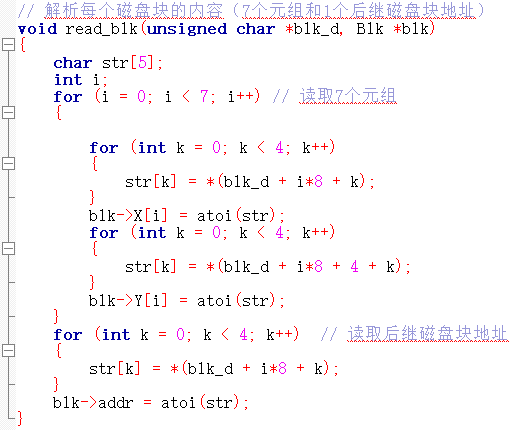
还需定义int init\_buf(Buffer \*buf)函数，调用ExtMem库中的initBuffer函数用于初始化缓冲区buffer，若发生错误则输出报错信息：



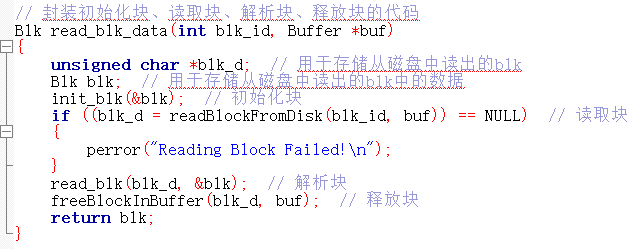
定义void init\_blk(Blk \*blk)函数，用于初始化Blk，将7个元组和后继磁盘块地址均置为-：



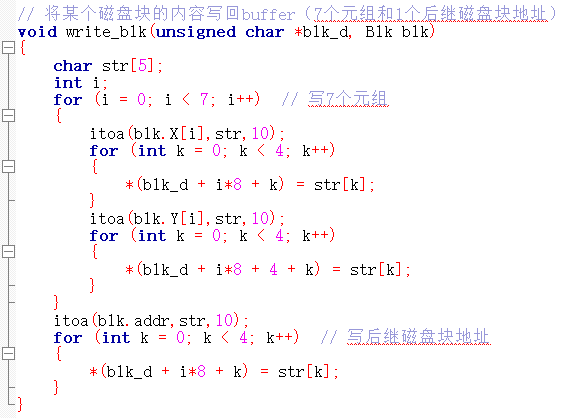
参照test.c文件，实现void read\_blk(unsigned char \*blk\_d, Blk \*blk)函数，该函数实现了解析磁盘块内容的功能，分别解析缓冲区中第blk\_d个块中的7个元组和1个后继磁盘块地址。从blk\_d+i\*8开始的4个字节中存储了第i个元组的第一个字段，从blk\_d+i\*8+4开始的4个字节中存储了第i个元组的第二个字段，最后的4个字节存储了后继磁盘块地址，将各个数据的每个字节存入字符串数组中，然后将字符串转化成数值并保存到中间存储块blk中：



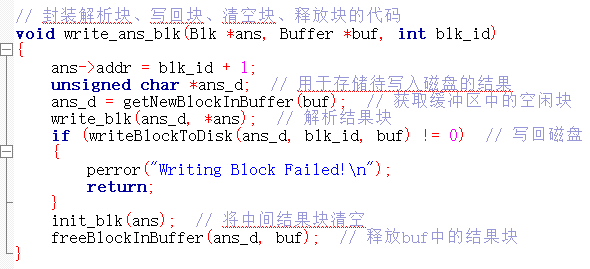
定义Blk read\_blk\_data(int blk\_id, Buffer \*buf)函数，封装了初始化块函数init\_blk、读取块函数readBlockFromDisk、解析块函数read\_blk和释放块函数freeBlockInBuffer的代码：



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：

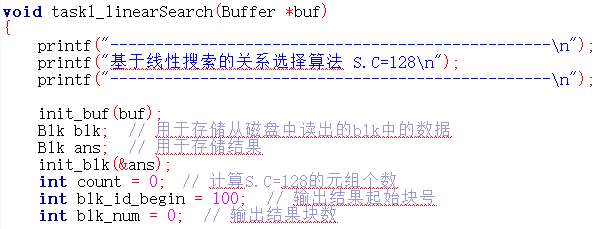


定义void write\_ans\_blk(Blk \*ans, Buffer \*buf, int blk\_id)函数，封装了解析块函数write\_blk、写回块函数writeBlockToDisk、清空块函数init\_blk、释放块函数freeBlockInBuffer的代码，清空块的目的是便于结果块的重复利用：

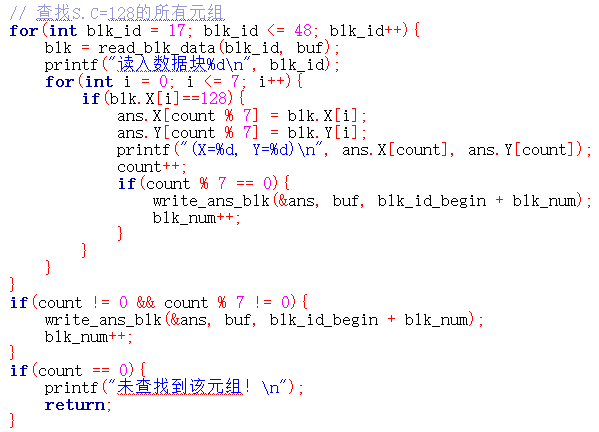


然后实现该算法的主体流程。

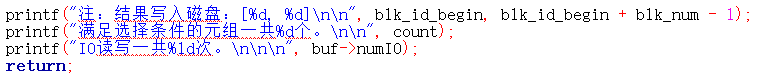
首先，输出任务名称并定义相关变量，包括存储从磁盘中读出的数据的中间存储块blk、结果块ans、满足条件的元组计数器count、输出结果起始块号blk\_id\_begin和输出结果块数计数器blk\_num：



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。

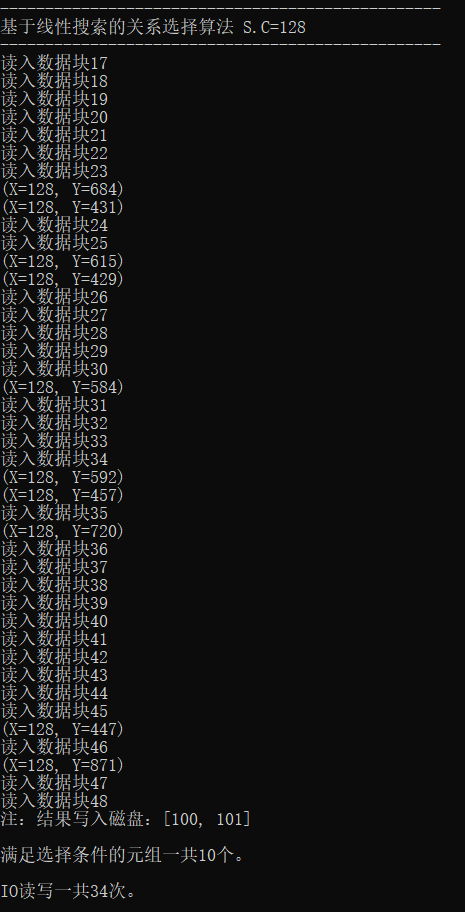


最后，打印结果写入的磁盘块号、满足选择条件的元组个数和IO读写次数。



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。



1. **实现两阶段多路归并排序算法（TPMMS）**

问题分析：

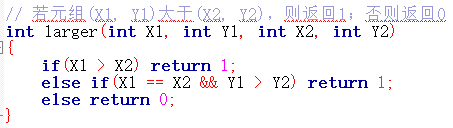
本实验需要实现的是**两阶段多路归并排序算法（TPMMS）。**需要利用内存缓冲区将关系R和S分别排序，并将排序后的结果存放在磁盘上，注意由于内存有限，因此数据并无法一起全部读入内存，也不能定义大数组存储中间结果。

实现思路是分两阶段实现该算法，划分子集进行归并排序。

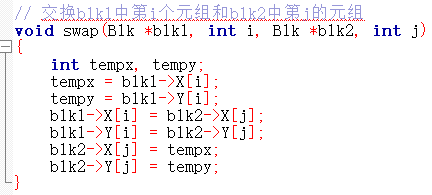
第一阶段：划分子集并进行子集内的排序，将B块数据划分成N个子集合，使每个子集合块数小于内存缓冲区可用块数。每个子集合装入缓冲区采用内排序排好序并重新写回磁盘。对于关系R，我将其划分为2个子集合（每个子集合8块）进行排序，中间结果存储到第201-216号磁盘块；对于关系S，我将其划分为4个子集合（每个子集合8块）进行排序，中间结果存储到第217-248号磁盘块。

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。

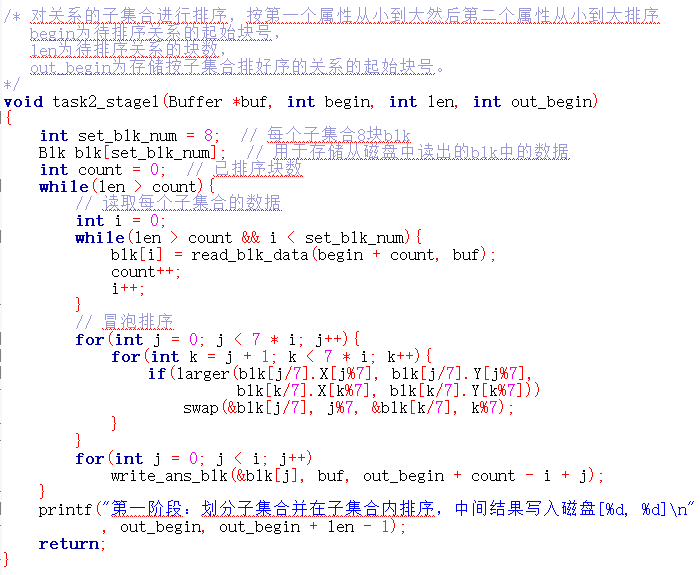
为实现第一阶段算法，需要首先定义函数int larger(int X1, int Y1, int X2, int Y2)，用于判断元组(X1, Y1)是否大于元组(X2, Y2)，（首先判断X1和X2的大小，若相等则再判断Y1和Y2的大小）若是则返回1，否则返回0：



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：

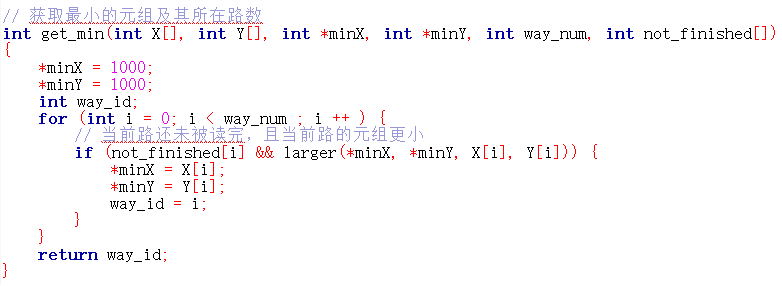


然后实现第一阶段的算法void task2\_stage1(Buffer \*buf, int begin, int len, int out\_begin)：



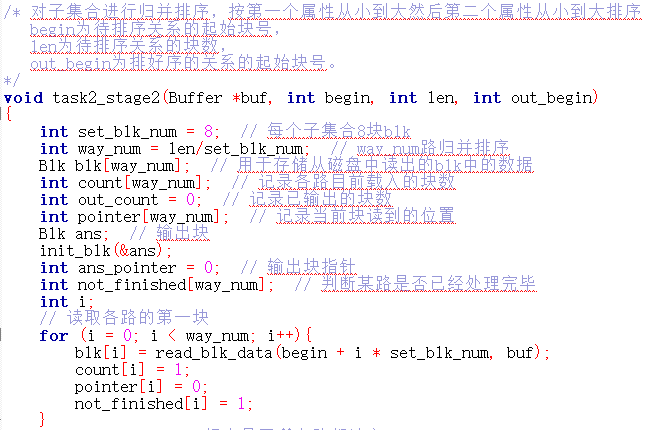
如上图所示，函数参数的含义已在注释中写明。首先设置每个子集合块数为8（由于缓冲区中只能存下8块数据块，并且采用冒泡排序不需要另外的结果块，因此选择每个子集合的块数为8），然后定义blk数组用于存储从磁盘中读出的数据，定义变量count用于记录已排序块数。依次读取每个子集合的数据，对于每个子集合，采用冒泡排序，定义两层循环，调用上述定义的larger函数判断元组大小关系，调用上述定义的swap函数在必要时交换两元组位置。最终将该子集合已排好序的结果写入磁盘。输出中间结果写入的磁盘块号。

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：

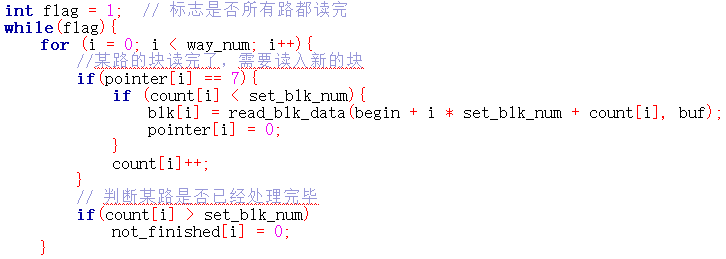


然后实现第二阶段的算法void task2\_stage2(Buffer \*buf, int begin, int len, int out\_begin)。

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：



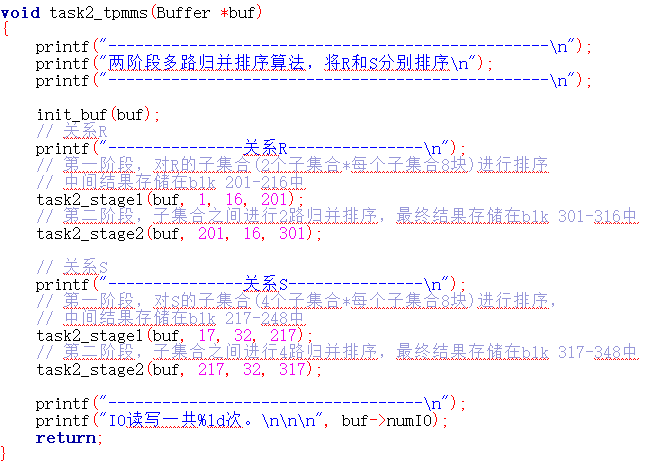
然后，设置flag标志是否所有路都读完，当flag为1时表示还有某路未读完，为0时表示所有路均已读完。当有某路未读完时，持续循环。遍历每路，若某路的块读完了，则需要读入新的块，更新相关的指针和计数器，同时需要判断某路是否已经处理完毕并更新not\_finished标志数组：



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。

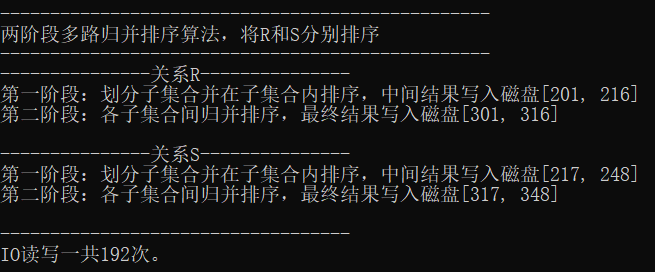


整体算法void task2\_tpmms(Buffer \*buf)如下图所示，首先输出任务名称并初始化缓冲区。然后调用上述定义的第一阶段函数task2\_stage1和第二阶段函数task2\_stage2，分别对关系R和S进行两阶段多路归并排序，输出IO读写次数：



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。



1. **实现基于索引的关系选择算法**

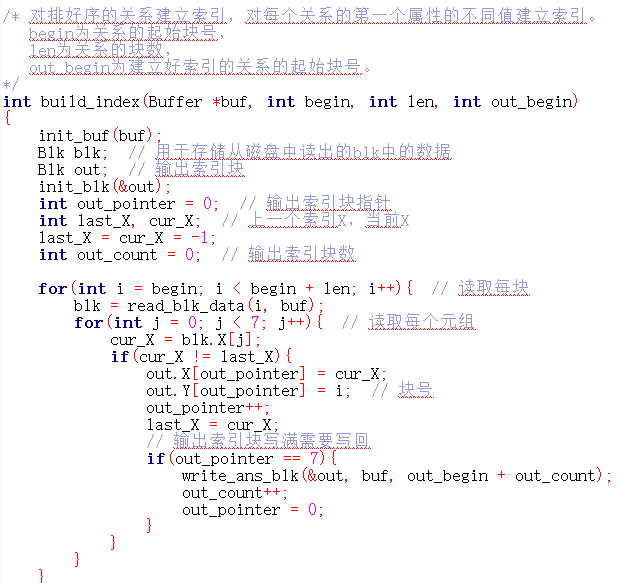
问题分析：

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。

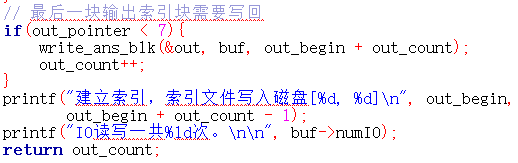
然后，利用建立好的索引文件找出S.C=128的索引，从而找到对应的全部元组。最终结果存储在120-121号磁盘块。

首先定义建立索引函数int build\_index(Buffer \*buf, int begin, int len, int out\_begin)：

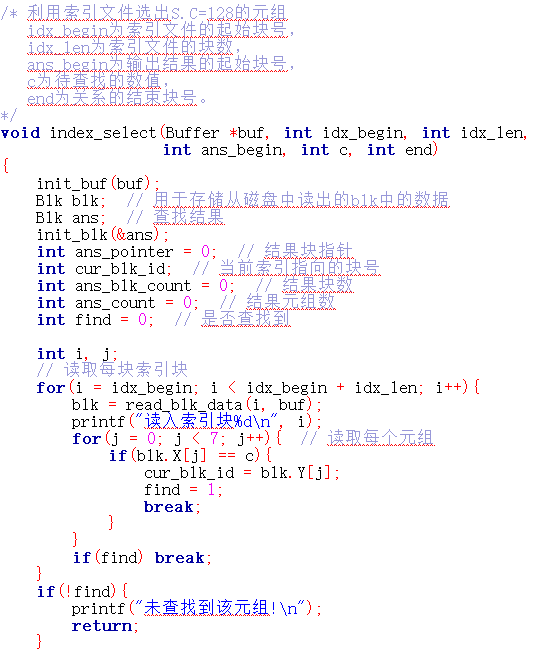


如上图所示，函数参数的含义已在注释中写明。首先重置缓冲区buf，定义blk用于存储从磁盘中读出的blk中的数据，定义并初始化out表示输出索引块，out\_pointer表示输出索引块指针，last\_X和cur\_X分别表示上一个索引和当前索引，初始值为-1，out\_count表示输出索引块数。然后遍历每块的每个元组，该元组的第一个属性值为当前索引值cur\_X，若当前索引值不等于上一个索引值last\_X，则需要将当前索引值和当前块号作为一个元组存入输出索引块中，更新输出索引块指针，设置上一个索引值last\_X等于当前索引值cur\_X。若输出索引块写满则需要写回，同时更新相关计数器和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

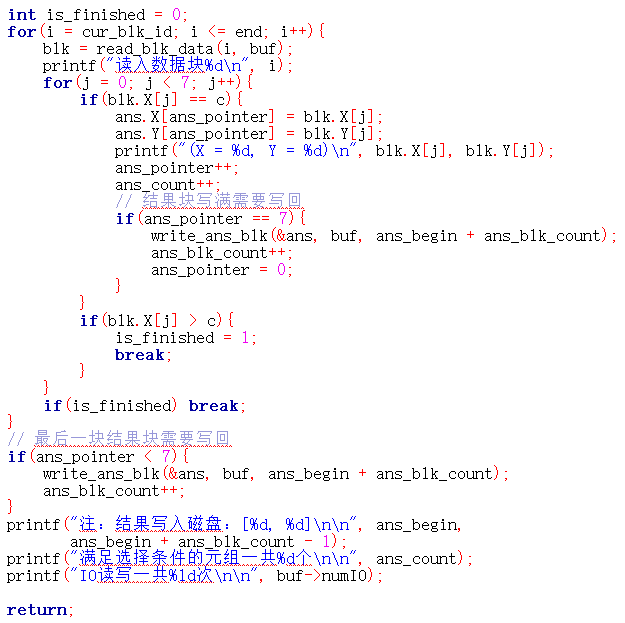
当循环结束时，若最后一块输出索引块未写满则需要写回，输出存储索引文件的磁盘块号以及IO读写次数，返回输出索引块数量：



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)：

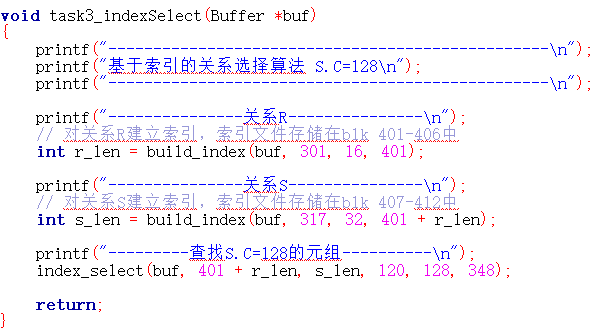


如上图所示，函数参数的含义已在注释中写明。首先重置缓冲区buf，定义blk用于存储从磁盘中读出的blk中的数据，定义并初始化ans用于存储查找结果，定义ans\_pointer表示结果块指针，cur\_blk\_id表示当前索引指向的块号，ans\_blk\_count表示结果块数，ans\_count表示结果元组数，find标志是否查找到对应元组。遍历关系S的全部索引块的每个元组，查找等于待查找属性值的索引，找到该属性值对应的开始块号cur\_blk\_id，若找到则跳出循环。若最终未找到则输出提示信息并返回。



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：

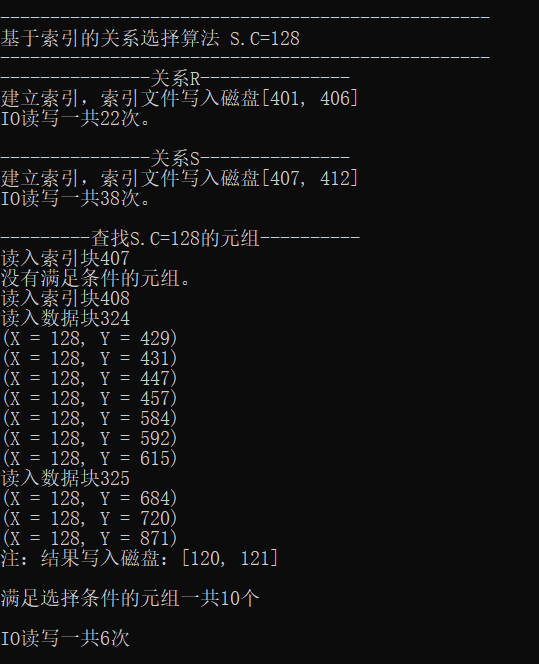


实验结果：

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。



1. **实现基于排序的连接操作算法（Sort-Merge-Join）**

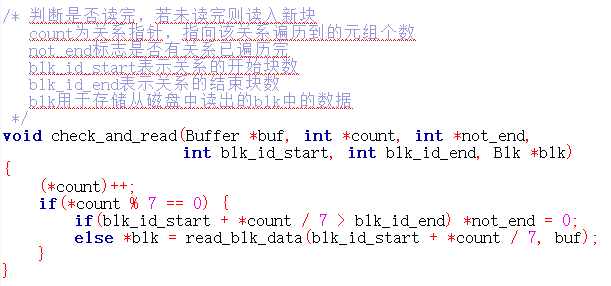
问题分析：

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。

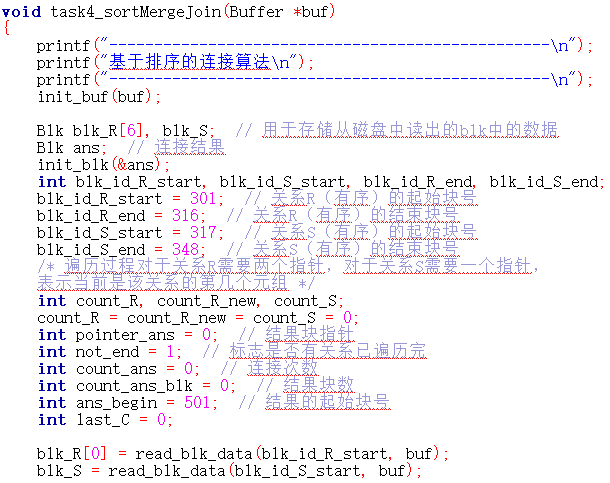
对于连接之后的元组，一个元组占四个属性值，按照每块3个连接结果进行存储，即**每块存储6个元组**，最后一个元组位置置空。最终结果存入501-630号磁盘块。

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：

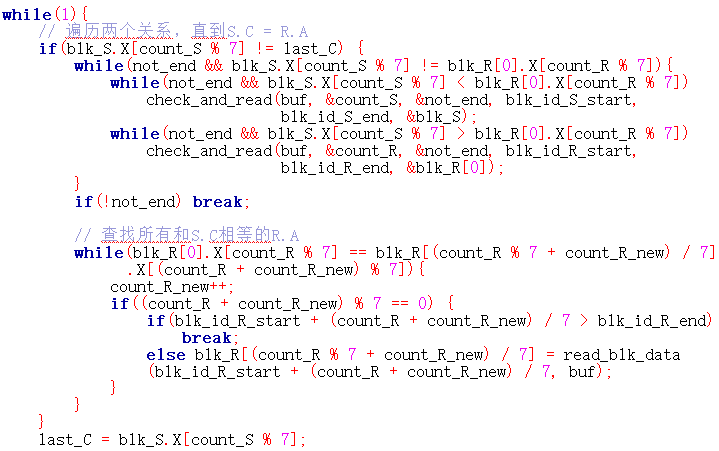


如上图所示，函数参数的含义已在注释中写明。首先更新作为关系指针的计数器count，当指针指向当前块的最后一个元组时，若此时已经读取到该关系的最后一块，则将not\_end标志为已结束，否则读取下一块的第一个元组。其中，\*count / 7表示当前元组是该关系中的第几块，加上初始块号blk\_id\_start即可得到当前元组所在的块号。

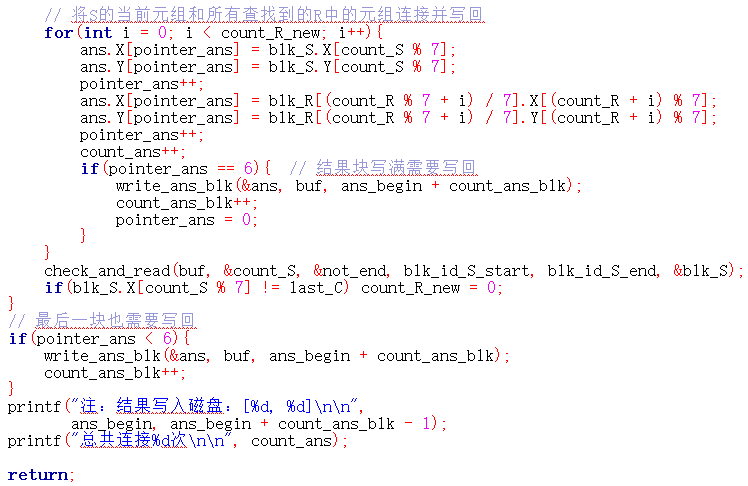
然后，实现该算法void task4\_sortMergeJoin(Buffer \*buf)的主体流程，首先输出任务名称并初始化缓冲区buf。定义blk\_R和blk\_S用于存储从磁盘中读出的blk中的数据，其中blk\_R是长度为6的数组（因为缓冲区中共能存下8个数据块，1个用于存储关系S的当前块，一个作为输出块，其余6个则存储关系R的数据块，以防关系R中有多个块中的元组的第一个属性值相等）。定义并初始化ans用于存储连接结果，blk\_id\_R\_start和blk\_id\_S\_start表示两关系的起始块号，blk\_id\_R\_end和blk\_id\_S\_end表示两关系的结束块号，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：



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：

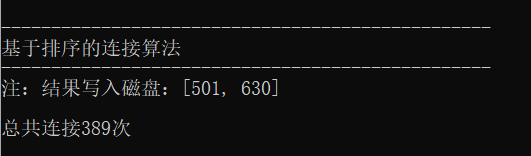


此时的count\_R\_new则表示关系R中共有多少个第一个属性相等的元组。遍历这些元组，将遍历到的元组与关系S中指向的当前元组连接，并写入结果块，同时更新结果块指针。当结果块写满时需要写回，并更新相关计数器和指针。遍历结束后，关系S的指针后移。只有当关系S的当前元组的第一个属性值与上一轮的第一个属性值相同时，才需要将count\_R\_new计数器置零，否则不用置零，直接供下一轮使用。当两个关系全部连接结束后，若最后一块为写满，也需要写回。输出结果写入的磁盘块号以及总共的连接次数：



实验结果：

最终结果写入501-630号磁盘，总共连接389次。



1. **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：

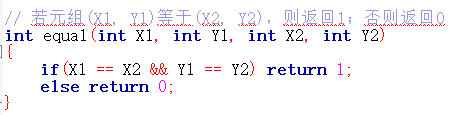
本实验需要实现的是**基于排序或散列的两趟扫描算法**，实现**并（S∪R）、交（S∩R）、差（S-R）**其中一种集合操作算法，将结果存放在磁盘上，并统计并、交、差操作后的元组个数。

此处，我实现的是**基于排序的两趟扫描并（S∪R）算法**。

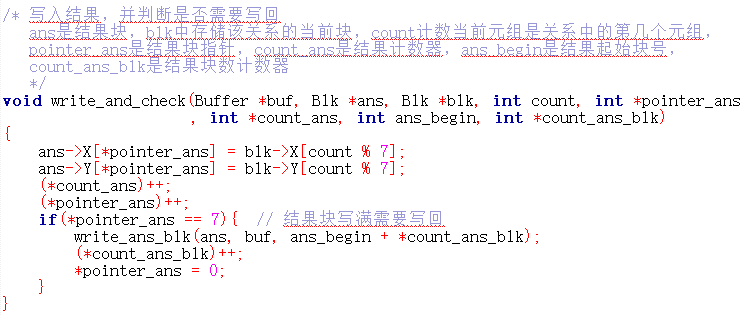
基本思路是，用两个指针分别遍历关系R和S，直到两指针指向的元组相等。若关系S未被读完，且R中的元组大于S中的元组，则将S中的元组写入结果块并读入新的；若关系R未被读完，且S中的元组大于R中的元组，则将R中的元组写入结果块并读入新的；若只剩S未被读完，则将S中的其余元组全部写入结果块；若只剩R未被读完，则将R中的其余元组全部写入结果块。最后，当S中的元组等于R中的元组时，写入S中的元组，且R和S均读入新的，进入下一轮循环继续遍历。

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：

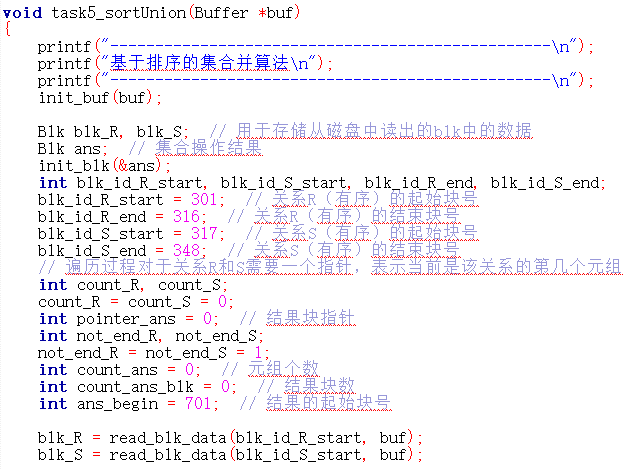


然后，封装写入结果和写回磁盘的函数void write\_and\_check(Buffer \*buf, Blk \*ans, Blk \*blk, int count, int \*pointer\_ans, int \*count\_ans, int ans\_begin, int \*count\_ans\_blk)：

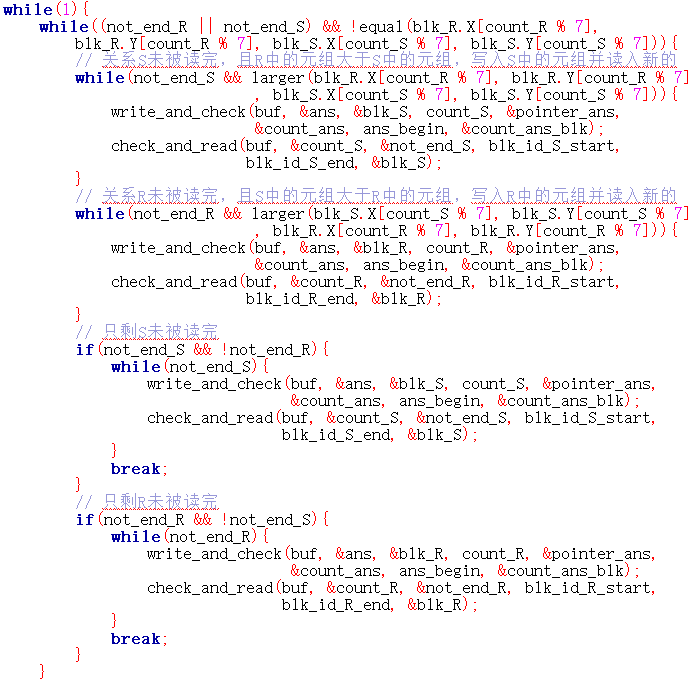


如上图所示，函数参数的含义已在注释中写明。首先，将指针指向的当前元组写入结果块，并更新相关计数器和指针，其中count % 7表示当前元组在其所在块中的位置。当结果块写满时需要写回磁盘，并更新相关计数器和指针。由于这段代码在集合的交、并、差等操作中均多次出现，因此将它们封装起来可以减少代码重复。

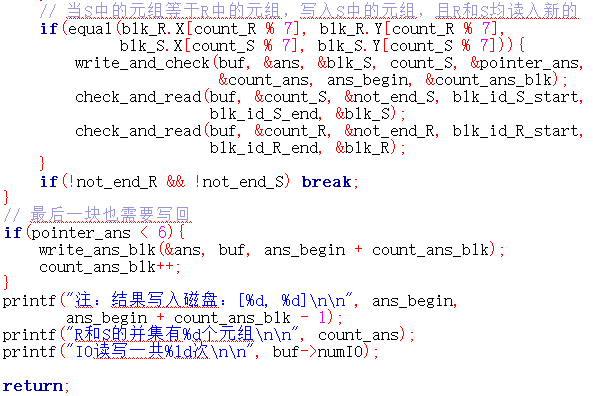
然后，实现算法void task5\_sortUnion(Buffer \*buf)的主体流程，首先输出任务名称并初始化缓冲区buf。定义blk\_R和blk\_S用于存储从磁盘中读出的blk中的数据，定义并初始化ans用于存储集合操作结果，定义blk\_id\_R\_start和blk\_id\_S\_start表示两关系的起始块号，blk\_id\_R\_end和blk\_id\_S\_end表示两关系的结束块号，count\_R和count\_S分别是遍历两关系的指针，表示当前是该关系的第几个元组，pointer\_ans为结果块指针，not\_end\_R和not\_end\_S分别标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：



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。

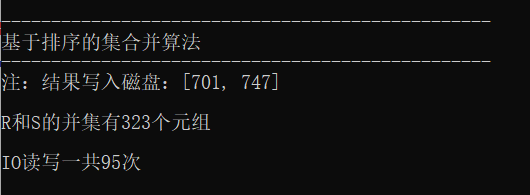


最后，当关系S中的元组等于关系R中的元组时，调用write\_and\_check函数将其中一个元组写入结果块，且调用check\_and\_read函数使得关系R和S的指针均向后移，进入下一轮循环继续遍历。如果关系R和S均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的并集元组个数以及IO读写次数：



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。



# 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。

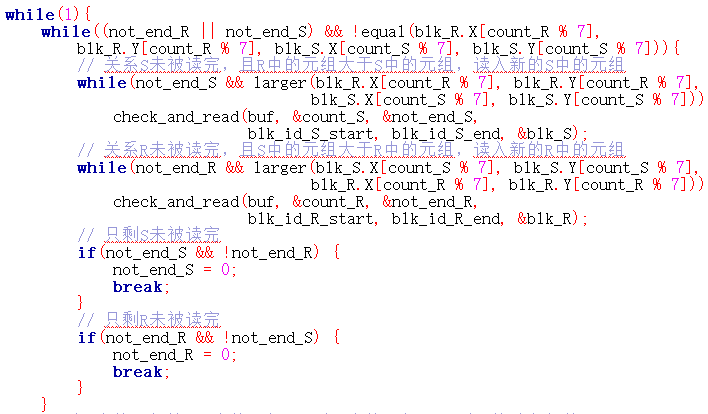
基本思路与并算法类似，不同的是，此处只在关系R和关系S中的元组相等时才将元组写入结果块。用两个指针分别遍历关系R和S，直到两指针指向的元组相等。若关系S未被读完，且R中的元组大于S中的元组，则将S的指针后移；若关系R未被读完，且S中的元组大于R中的元组，则将R中的指针后移；若只剩S或只剩R，均结束。最后，当S中的元组等于R中的元组时，才将该元组写入结果块，且R和S的指针均后移，进入下一轮循环继续遍历。

最终结果存入801-802号磁盘块。

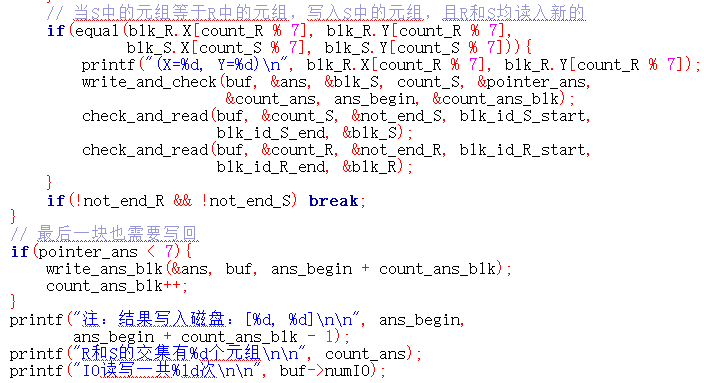
实现算法void task5\_sortIntersection(Buffer \*buf)的主体流程，首先输出任务名称并初始化缓冲区buf。变量的定义与并算法中的一致，便不再赘述，读入关系R和S的第一个块：



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：

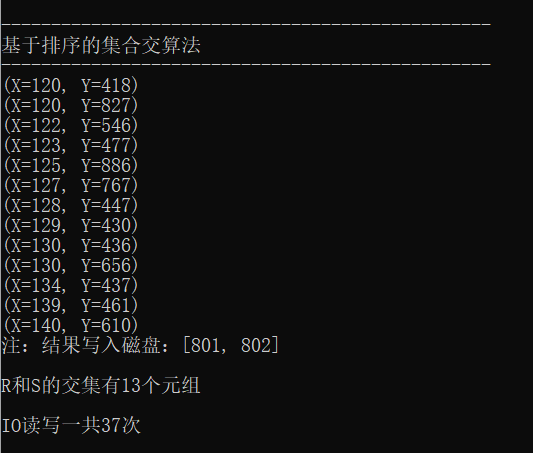


最后，当关系S中的元组等于关系R中的元组时，输出该元组的值，调用write\_and\_check函数将其中一个元组写入结果块，同时调用check\_and\_read函数使得关系R和S的指针均向后移，进入下一轮循环继续遍历。如果关系R和S均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的交集元组个数以及IO读写次数：



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。



**（2）基于排序的两趟扫描差（S-R）算法**

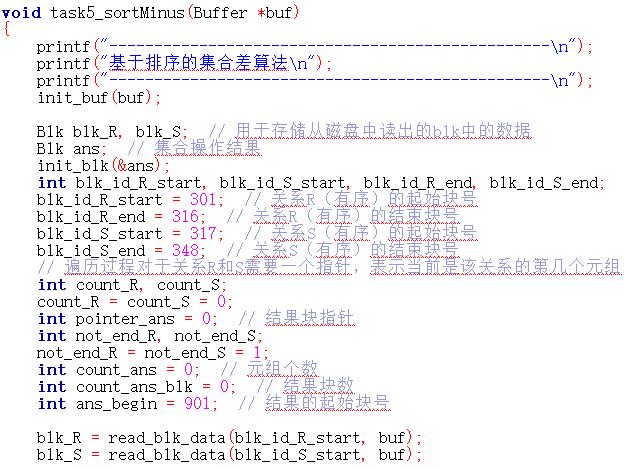
问题分析：

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。

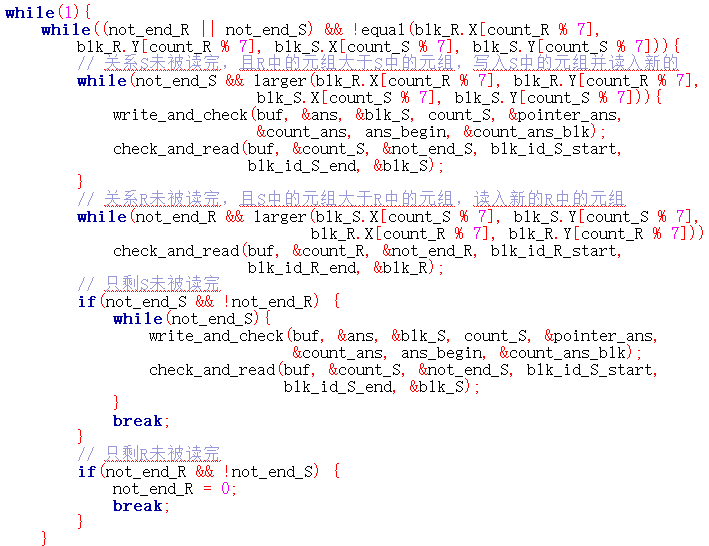
基本思路也与并算法类似，不同的是，此处只在关系R中没有而关系S中有的元组相等时才将元组写入结果块。用两个指针分别遍历关系R和S，直到两指针指向的元组相等。若关系S未被读完，且R中的元组大于S中的元组，则将S中的当前元组写入结果块，并且S的指针后移；若关系R未被读完，且S中的元组大于R中的元组，则将R中的指针后移；若只剩S未被读完，则将S中剩余的元组全部写入结果块，并且结束当前轮的遍历；若只剩R未被读完，均直接结束当前轮的遍历。最后，当S中的元组等于R中的元组时，R和S的指针均直接后移，进入下一轮循环继续遍历。

最终结果存入901-931号磁盘块。

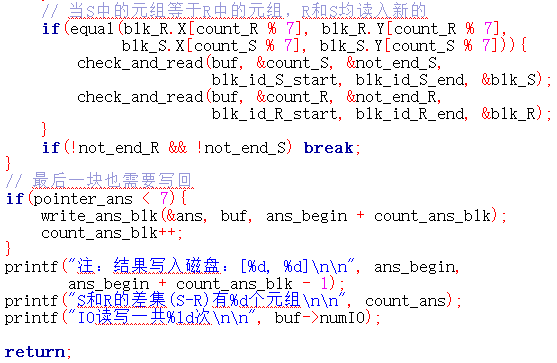
实现算法void task5\_sortMinus(Buffer \*buf)的主体流程，首先输出任务名称并初始化缓冲区buf。变量的定义与并算法中的一致，便不再赘述，读入关系R和S的第一个块：



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：

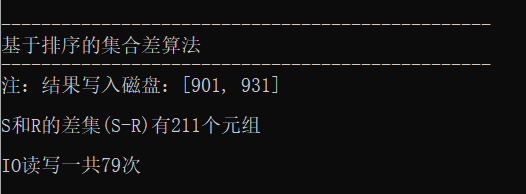


最后，当关系S中的元组等于关系R中的元组时，调用check\_and\_read函数使得关系R和S的指针均直接后移，进入下一轮循环继续遍历。如果关系R和S均已被读完，则跳出循环。当循环结束时，若最后一块未被写满，则需要写回磁盘。输出存储结构的磁盘块号、两个关系的差集元组个数以及IO读写次数：



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。



# 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。