|  |
| --- |
| 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: | 潘延麒 | | 学 号: | 180110516 | | 专 业: | 计算机科学与技术 | | 日 期: | 2021-04-30 | |

# 实验目的

本次实验要求学生掌握TPMMS外排序方法，并基于TPMMS外排序结果建立索引文件，掌握关系选择、连接、集合的交、并、差等操作的实现，最后给出每次操作带来的I/O开销，理解算法的I/O 复杂性

# 实验环境

运行环境：Windows 10 V19042.928 + GCC version 6.3.0

开发环境：VS Code 1.55.2

# 实验内容

1. **实验环境构成**

本次实验利用extmem程序模拟对内存和磁盘的操作。其中，内存由数据结构Buffer构成，其定义如图3.1所示。其中numIO表示从该Buffer写入或读出了多少次磁盘块；bufSize在本实验中是静态变量，代表内存空间大小，单位为Byte，本实验中，bufSize大小为520；blkSize代表每个磁盘块的大小，单位同样为Byte，本实验中，blkSize大小为64，可以看出520 / 64 = 8，这说明在内存Buffer中，

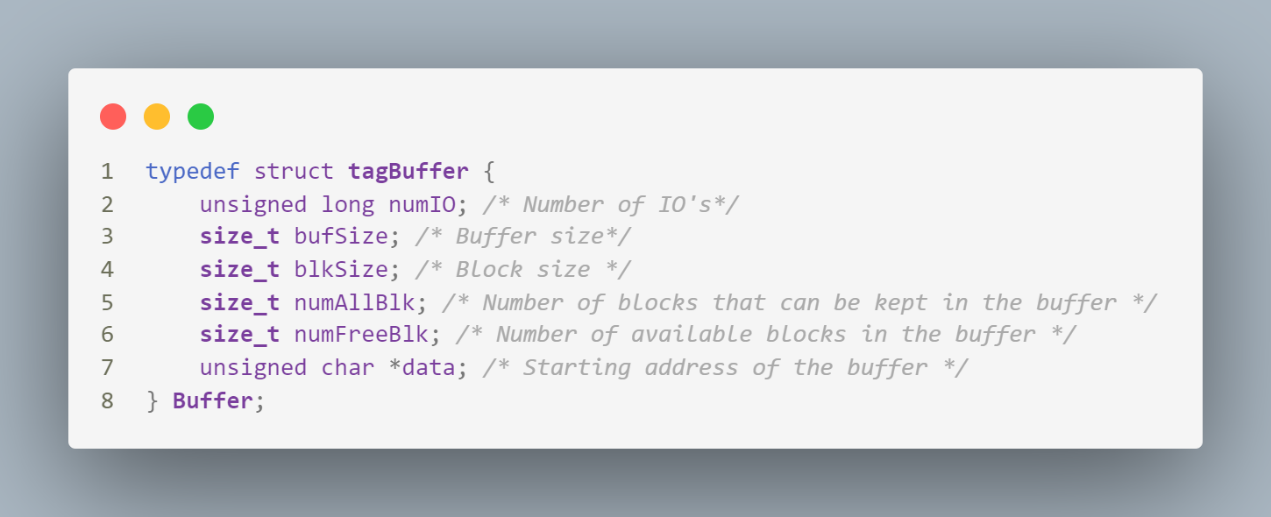


图3.1 Buffer数据结构

我们最多只能维护8个数据块，此外，由于还多出了8Byte，这8Byte则刚好被用于标记这8个数据块的有效性，于是称这8个Byte为tag；numAllBlk也是静态变量，代表Buffer中能够维护的最大数据块数，在本实验中，该变量大小为8；numFreeBlk表示Buffer中可用数据块数量，numFreeBlk可通过统计Buffer中的tag值得出；最后，data指向Buffer的内存地址，通过data，我们可以访问Buffer中的每一个数据块。图3.2展示了Buffer的内存分布结构。

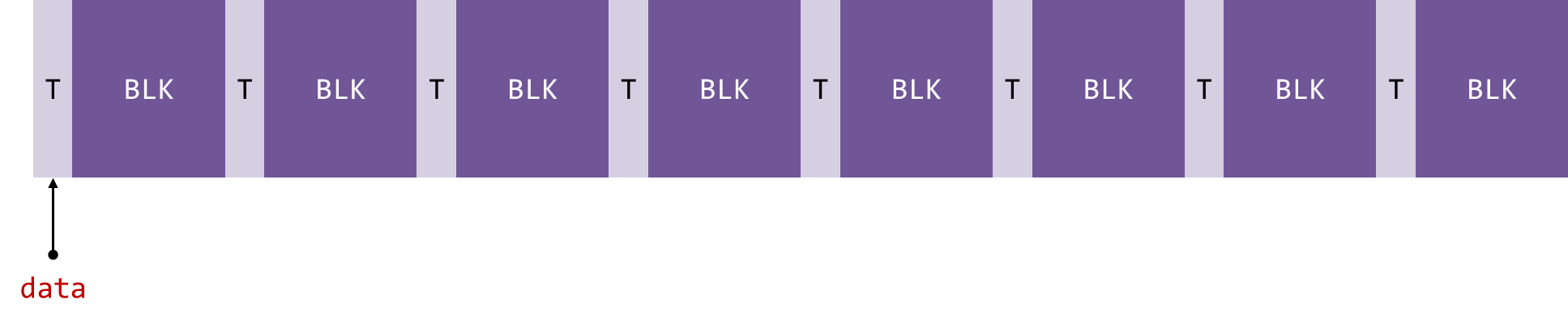


图3.2 Buffer内存分布结构

由于本实验数据的特殊性，每个关系的属性值域都是整型类型，即每个属性值需要用4Byte表示，因此每一个数据块可承载64 / 4 = 16个属性值。但值得注意的是，每个数据块后8Byte需要被保留下来作为nextBlk指针，指向与之逻辑相邻的下一个数据块，因此每一个数据块实际只可承载64 / 4 – 8 / 4 = 14个属性值。图3.3显示了数据块结构布局。

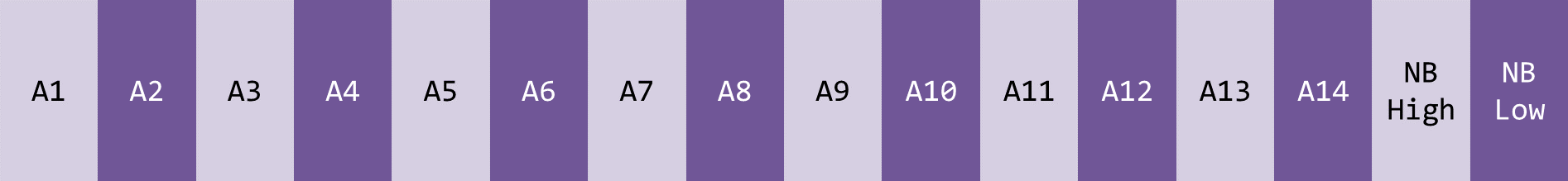


图3.3 数据块分布结构[[1]](#footnote-1)

本实验给出两个关系：R与S，其中关系R占用16个数据块，被分配在磁盘块1到16的位置；关系S占用32个数据块，被分配在磁盘块17到48的位置。两个关系的关系模式可分别被表示为R(A, B)和S(C, D)，于是每个数据块可保存关系R或关系S的7条Record。作为示例，图3.4列出了关系R一个数据块的布局结构。

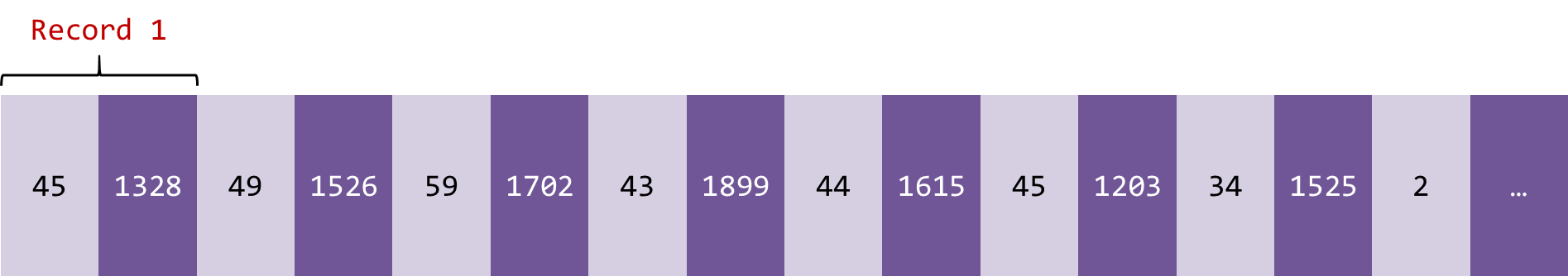


图3.4 关系R第一个数据块布局

此外，extmem提供7个API函数，它们分别是：

* **Buffer** \***initBuffer**(**size\_t** bufSize, **size\_t** blkSize, **Buffer** \*buf)

函数说明：初始化Buffer数据结构；

参数说明：

1. bufSize：欲申请的Buffer大小；
2. blkSize：每个数据块大小；
3. buf：Buffer数据对象指针；

返回值：Buffer数据对象指针，如果申请失败将返回NULL；

* void **freeBuffer**(**Buffer** \*buf)

函数说明：清除Buffer申请的内存空间，**注意，这里只释放了data所指的内存**；

参数说明：

1. buf：欲释放的Buffer

返回值：无

* unsigned char \***getNewBlockInBuffer**(**Buffer** \*buf)

函数说明：向Buffer内存缓冲取申请一个空的数据块

参数说明：

1. buf：Buffer内存缓冲区

返回值：该数据块的起始地址，**注意，这里的起始地址不包括tag**；

* void **freeBlockInBuffer**(unsigned char \*blk, **Buffer** \*buf)

函数说明：清除Buffer中指定的数据块；

参数说明：

1. blk：欲释放的数据块在Buffer中的起始地址，**注意，这里的起始地址不包括tag**；
2. buf：Buffer内存缓冲区；

返回值：无；

* int **dropBlockOnDisk**(unsigned int addr)

函数说明：删除一个指定的磁盘块；

参数说明：

1. addr：欲删除的磁盘块号；

返回值：-1删除失败，0删除成功；

* unsigned char \***readBlockFromDisk**(unsigned int addr, **Buffer** \*buf)

函数说明：读取一个磁盘块至Buffer；

参数说明：

1. addr：欲读取的磁盘块号；
2. buf：Buffer内存缓冲区

返回值：若Buffer中没有足够的大小则返回NULL，否则返回该磁盘块被读入至Buffer中的起始地址，**注意，这里的起始地址同样不包括tag；**

* int **writeBlockToDisk**(unsigned char \*blkPtr, unsigned int addr, **Buffer** \*buf)

函数说明：将Buffer中的一个数据块写入指定磁盘块中

参数说明：

1. blkPtr：欲写入的数据块在Buffer中的起始地址，不包括tag；
2. addr：欲写入的磁盘块块号；
3. buf：Buffer内存缓冲区；

返回值：-1写入失败，0写入成功；

1. **实验目标**

本次实验的目标是要通过上述实验环境模拟DBMS的查询过程。该实验由5个任务组成，它们分别是：

* 任务一：基于线性搜索的关系选择算法

利用线性搜索算法，模拟实现SQL语句：

select S.C, S.D from S where S.C = 50

并把选择结果存在磁盘上。

* 任务二：两阶段多路归并排序算法

两阶段多路归并排序算法是外排序算法的一种，其目的是利用有限的内存空间排序对磁盘中的数据进行排序，该算法又可缩写为TPMMS，即The Two-Pass Multiway Merge Sort。该任务要求我们利用TPMMS算法对关系R与S进行排序，并将排序结果写回磁盘。

* 任务三：基于索引的关系选择算法

利用任务二的排序结果，为关系S建立索引文件，并利用索引文件完成任务一的需求，最后，比较基于索引关系选择的算法与基于线性搜索的关系选择算法的I/O复杂性。

* 任务四：基于排序的连接算法

利用任务二的排序结果，对关系R与关系S进行自然连接操作，对应SQL语句为：

select S.C, S.D, R.A, R.B from S inner

join R on S.C = R.A

并把连接结果存在磁盘上。

* 任务五：基于排序的并、交、差算法

利用任务二的排序结果，对关系R与关系S进行并、交、差（S - R）操作，并将结果存放在磁盘上，此外，还应统计执行并、交、差操作后元组个数。

# 实验过程

1. **任务一：实现基于线性搜索的关系选择算法**

* **问题分析**

任务一几乎没有难度，只需依次读出关系S的磁盘块，并对每个磁盘块中的记录进行比较分析即可。算法描述如下：

|  |
| --- |
| **Algorithm for Task 1** |
| 1. 利用**getNewBlockInBuffer**初始化写数据块WriteBlk； 2. 利用**readBlockFromDisk**向Buffer中读入一个关系S的数据块Blk； 3. 扫描该Blk中的7个Record，若 Record.C = 50，则将该Record写入Buffer的WriteBlk中，同时，检查WriteBlk是否被写入了7条记录，若是，则将WriteBlk写回磁盘； 4. 利用**freeBlockInBuffer**释放读入的数据块； 5. 关系S数据块是否读取完毕？若是，则跳转至第2步； 6. WriteBlk中是否还有未写完的记录？若是，则把WriteBlk再次写回磁盘中； 7. 释放WriteBlk占用的Buffer空间； |

算法1 基于线性搜索的关系选择算法

上述算法保证Buffer中最多只有2个数据块，没有超过Buffer最大数据块的限制。算法的关键实现如图4.1所示：



图4.1基于线性搜索的关系选择算法关键代码

上图中，**bGetBLKRecord**函数能够基于传入的BBLKNum以及uiIndex获得Buffer中第BBLKNum个数据块的第uiIndex条记录；**WRITE\_TILL\_BLK\_FILL(NULL)**为宏函数，其定义如图4.2所示，其作用为判断写入块是否写满，若写满则将写入块写回磁盘块；**getKeyAttr**函数的作用是根据传入的记录以及属性列值返回相应的键值，例如S.C对应的键值即**getKeyAttr**(record, 1) 所得的结果；最后**bSetBLKRecord**函数则能够将一条记录写入到Buffer中的某数据块中。

更多工具函数的代码实现与相应功能详见buftool.c、disktool.c、sorttool.c以及utils.h文件，这些文件中的代码都做过详尽的注释，为节省篇幅，后文将不再对这些工具函数做赘述。

* **实验结果**

实验运行结果如图4.2所示。

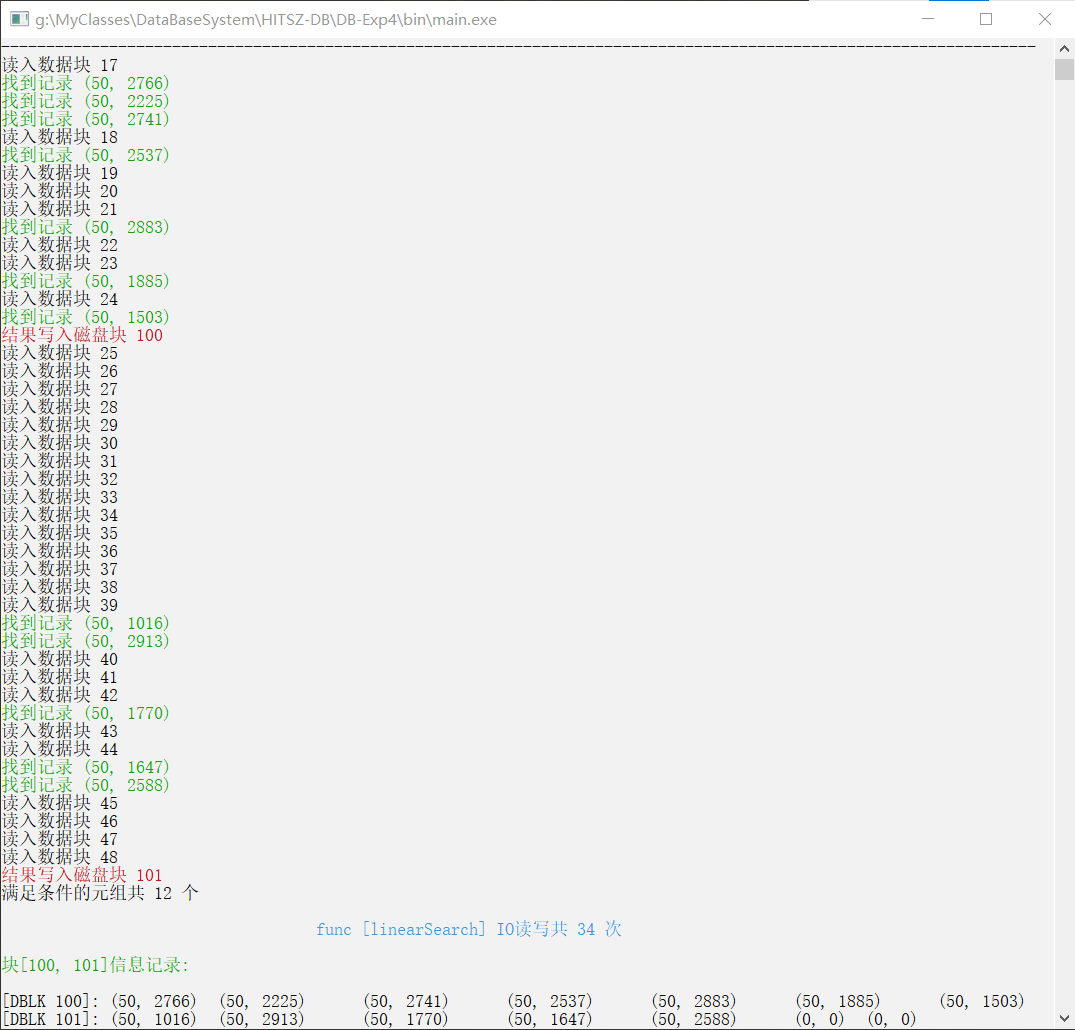
****

图4.2 任务一运行结果

可见，该算法扫描了关系S的所有磁盘块，并找到了12条有效记录，I/O次数共：读32次，写入12 / 7 + 1 = 2次，共34次。

1. **任务二：实现两阶段多路归并排序算法（TPMMS）**

* 问题分析

任务二实际上难度也不大，其关键在于吃透TPMMS算法，并对其予以实现。事实上，由于本实验的数据集较为特殊，满足关系R与S的数据块数量均小于B \* B = 64，因此，只需两趟扫描便能够对关系R与S进行排序了，故在本次实验中我没有实现多趟TPMMS算法。于是，问题就变得简单了，首先，我们人为将每个关系按照每组8个数据块（一个Buffer的最大容量）的方式进行分组，于是，关系S可被分为4组，而关系R可被分为2组。接下来，我们实现第一趟扫描算法——内排序算法，其流程如下：

|  |
| --- |
| **Algorithm for Task 2 – Part1** |
| 1. 利用**readBlockFromDisk**向Buffer中读入8个数据块； 2. 利用**bGetBLKRecord**以及插入排序算法对Buffer中的数据块进行排序； 3. 将排序结果依次写回磁盘块中； 4. 利用**freeBlockInBuffer**释放Buffer中的8个数据块； |

算法2.1 内排序算法

内排序部分关键代码实现如图4.3所示。



图4.3 内排序算法关键代码

接下来，我们便可利用内排序结果进行外排序了。以对关系S进行Tpmms为例，其算法描述如下所示：

|  |
| --- |
| **Algorithm for Task 2 – Part2** |
| 1. 对S的4个分组分别执行算法2.1，并将内排序结果写回磁盘； 2. 利用**getNewBlockInBuffer**初始化写数据块WriteBlk； 3. 利用**getNewBlockInBuffer**初始化**比较**数据块CompareBlk； 4. 初始化Buffer组指针POUT[4] = {0,0,0,0}，代表关系S被分为4组，POUT则记录每组被读入至Buffer中的内存块的当前输出至CompareBlk中的记录编号，初始指向第0条记录； 5. 初始化Disk组指针PIN[4] = {0,0,0,0}，代表关系S被分为4组，PIN则记录每组被读入Buffer中的数据块的相对序号，初始指向每组第0块； 6. 根据PIN[4]，读入关系S每组的第PIN[i]个数据块至Buffer中，并根据POUT[4]将Buffer中每组的第POUT[i]条记录写入CompareBlk中第i条记录的位置； 7. 如果CompareBlk中的记录并非都是无效的，那么算法**结束**，执行步骤10。 8. 找到CompareBlk中最小（最大）记录[[2]](#footnote-2)，并记录其对应的组号**r**，将该条记录写入WriteBlk中，如果WriteBlk已满，则将其写回磁盘； 9. 第r组应该读入下一条记录，于是令POUT[r]++，如果POUT[r]大于6，那么说明组r的读入Buffer中的数据块的记录已经检查完毕，该检查下一个数据块了，于是令POUT[r] = 0，PIN[r]++，这里，如果PIN[r]大于7，那么说明组r的所有数据块都已比较完毕，此时向CompareBlk的r处写入**无效记录**。执行步骤7。 10. WriteBlk中是否还有未写完的记录？若是，则把WriteBlk再次写回磁盘中； 11. 释放WriteBlk占用的Buffer空间； 12. 释放CompareBlk占用的Buffer空间； |

算法2.2 Tpmms算法

上述算法保证Buffer中最多具有6个数据块，不会超过Buffer最大数据块要求。TPMMS的关键代码实现如图4.4.1及图4.4.2所示。

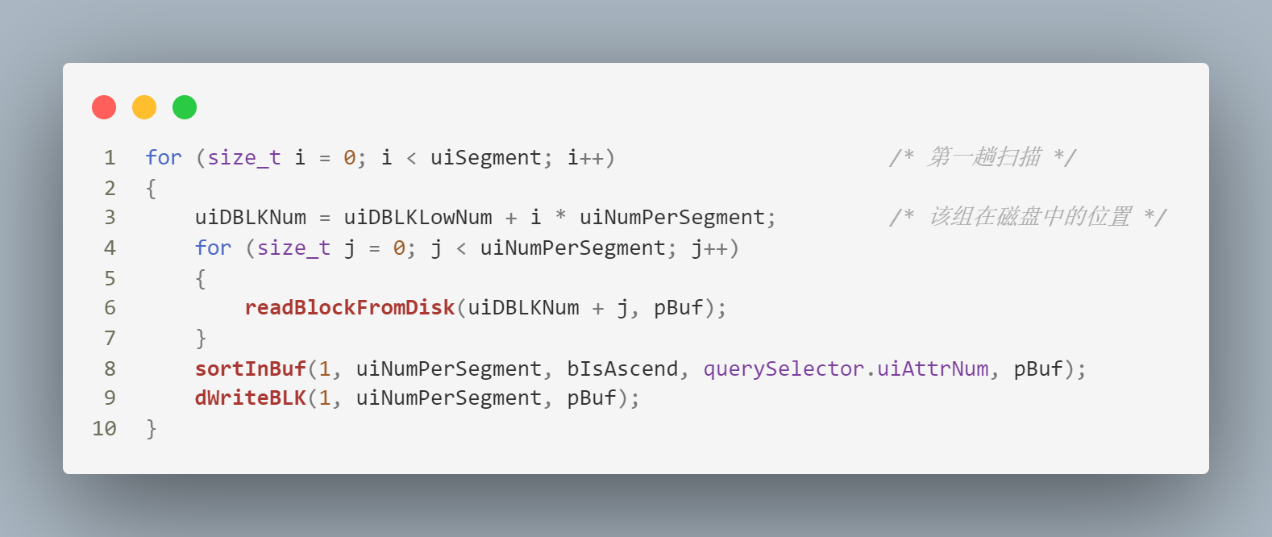


图 4.4.1 第一趟内排序

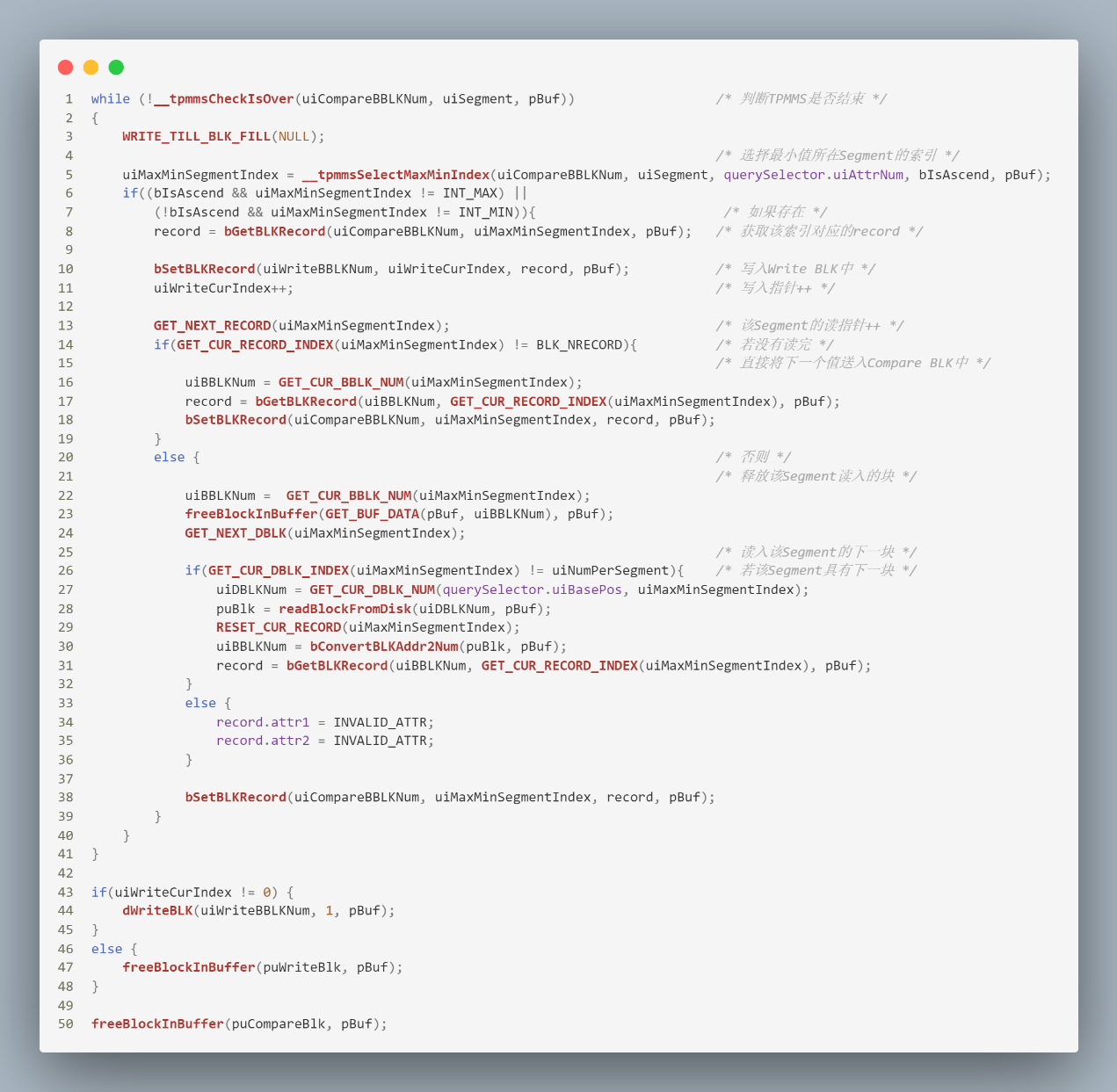


图4.4.2 TPMMS排序关键代码

为了节省篇幅，图4.4.2的代码省去了指针初始化和初始读入部分的代码片段，仅留下了对应算法2.2第7 ~ 12行的关键部分实现。

* 实验结果

图4.5.1显示了对关系R的TPMMS排序结果，图4.5.2显示了对关系S的TPMMS排序结果。

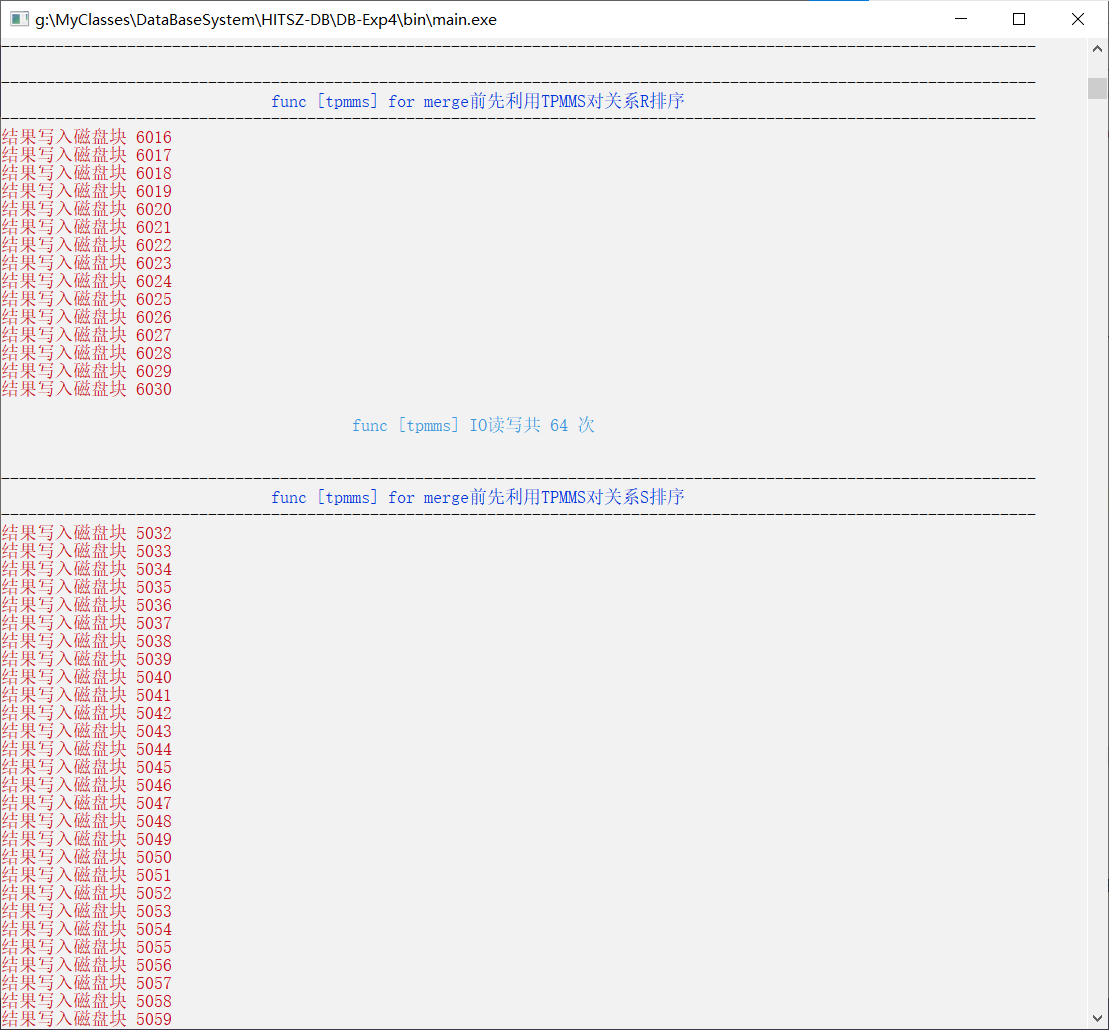
****

图 4.5.1 对关系R的TPMMS排序结果

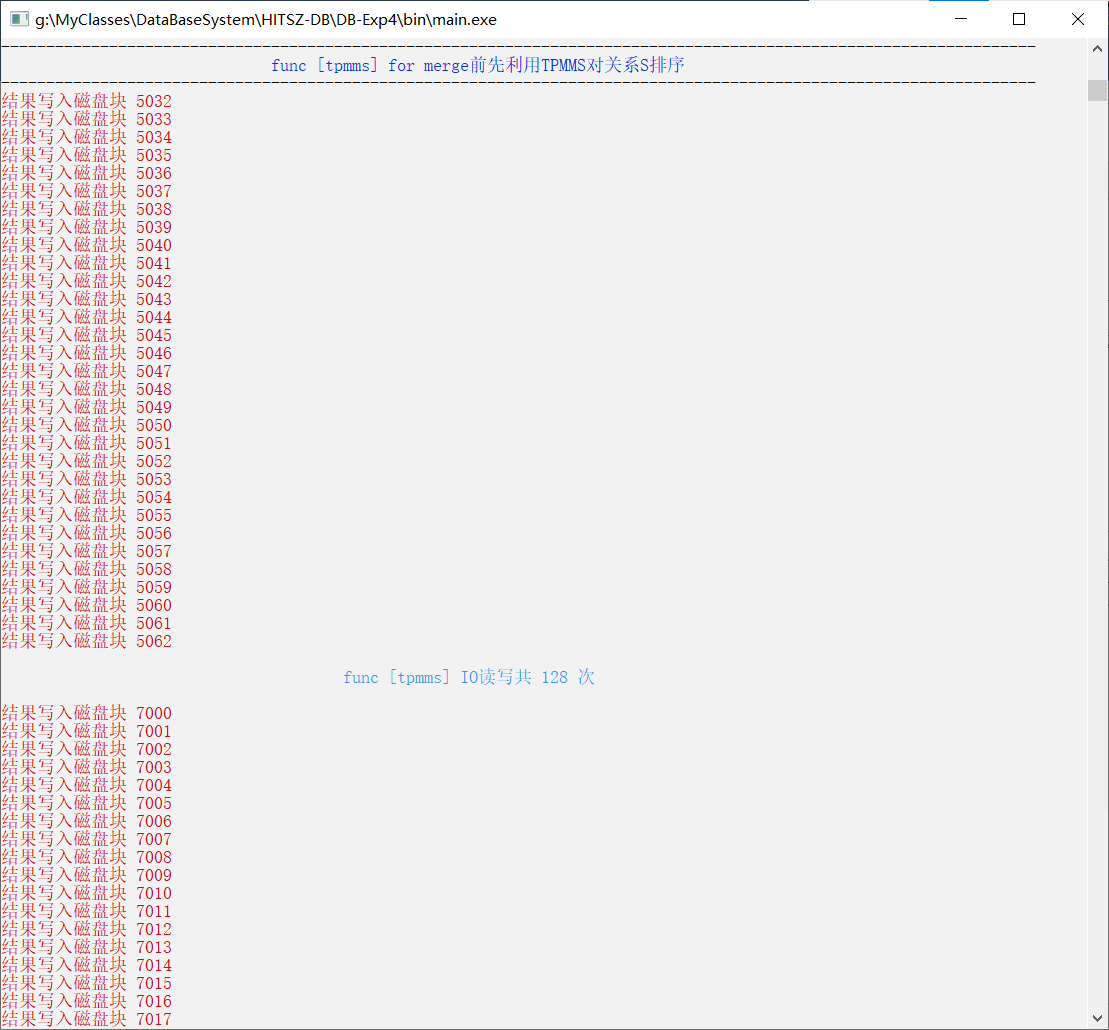
****

图 4.5.2 对关系S的TPMMS排序结果

可以很容易验证图4.5.1与图4.5.2的I/O读写正确性：对于关系R而言，内排序读入和写出共消耗16 + 16 = 32次I/O开销，第二趟归并排序则消耗读入和写出共16 + 16 = 32次I/O开销，故共64次I/O开销，关系S亦然。

1. **实现基于索引的关系选择算法**

* 问题分析

当我们完成了对关系S的排序后，我们就能很容易对其建立索引文件。索引文件没有采用复杂的B树结构，而仅仅采用了线性分组结构。其基本思想是将排序后的关系分成多个段，索引文件则仅记录这些段的第一个记录的目标属性值以及其相应块指针，图4.6展示了一种索引文件的示例结构。

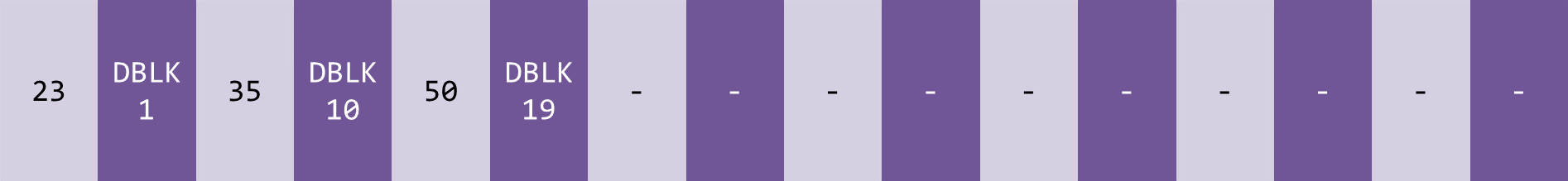


图4.6 索引文件示例[[3]](#footnote-3)

算法3.1说明了建立索引文件的方式。

|  |
| --- |
| **Algorithm for Task 3 – Part1** |
| 1. 利用**getNewBlockInBuffer**初始化写数据块WriteBlk； 2. 根据分段数uiNum计算每一段的BLK数uiBLKPerGap； 3. 读取每一段的第一块数据块，获取其键值uiKey及物理块号uiDBLKNum，将记录Record<uiKey，uiDBLKNum>写入WriteBlk中，同样，若WriteBlk写满，则将其写回磁盘，重复步骤3，直至读取完每一段数据块； 4. WriteBlk中是否还有未写完的记录？若是，则把WriteBlk再次写回磁盘中； 5. 释放WriteBlk占用的Buffer空间； |

算法3.1 索引文件建立算法

接下来，我们便可以通过索引文件完成任务三。思路同样很简单，根据需要查找的键值在索引文件中找到磁盘块上下界。以图4.6为例，若给出键值37，则通过在索引文件中进行比对，得到搜索上界为磁盘块号10，下界为磁盘块号19，因为35 < **37** < 50。接着，我们便可以通过简单的线性搜索法在磁盘块10到19间搜索键值为37的记录即可。以关系S的索引文件为例，其算法描述如下：

|  |
| --- |
| **Algorithm for Task 3 – Part2** |
| 1. 利用算法3.1为关系S建立索引文件； 2. 利用**readBlockFromDisk**在关系S的索引文件读取一个数据块； 3. 对于该索引数据块的每条记录Record(i)，若Record(i-1) < 键值 < Record(i)，则置搜索上界uiDBLKHighBound为Record(i).Key，并同时置搜索下界uiDBLKLowBound为Record(i-1).Key； 4. 利用**freeBlockInBuffer**释放Buffer中读入的数据块； 5. 关系S的索引文件是否全部读完？若否，则执行步骤2； 6. 利用算法1对区间[uiDBLKLowBound，uiDBLKHighBound]进行关系选择输出即可； |

算法3.2基于索引的关系选择算法

可见，基于索引的关系选择算法的关键在于确定搜索的上下界，由于不涉及B树结构，因此代码的实现也变得相对容易起来，图4.7.1为建立索引文件的关键代码，图4.7.2为基于索引的关系选择的关键代码。



图4.7.1 建立索引文件关键代码



图4.7.2 基于索引的关系选择算法关键代码

* 实验结果

建立索引文件的结果如图4.8.1所示，利用索引文件进行基于索引的关系选择结果如图4.8.2所示。其中，在建立索引文件的过程中，我们将关系S分为4段，于是S的索引文件便只有4条记录，于是我们可以只花费1次读操作便能从索引文件中获得搜索的上下界。

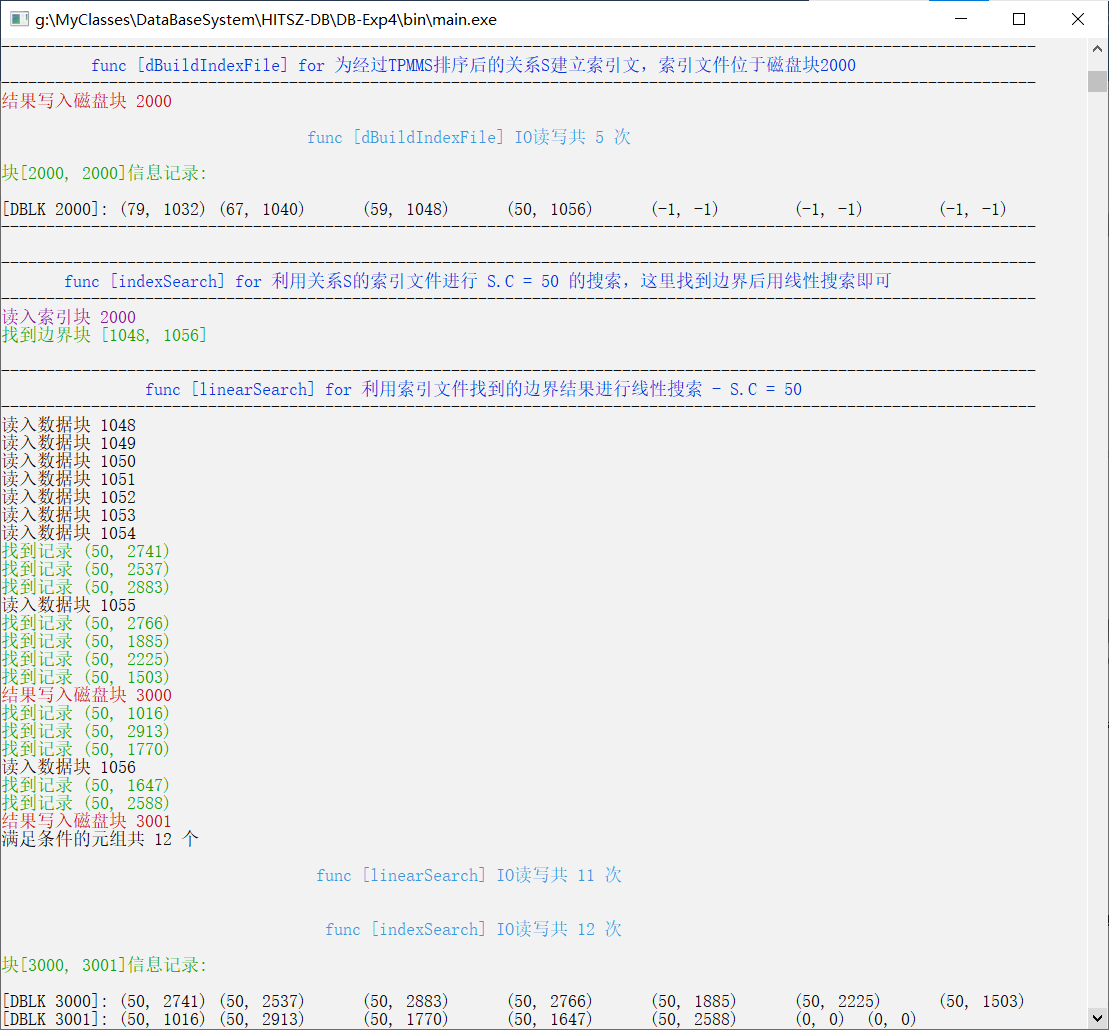


图4.8.1 建立索引文件的结果

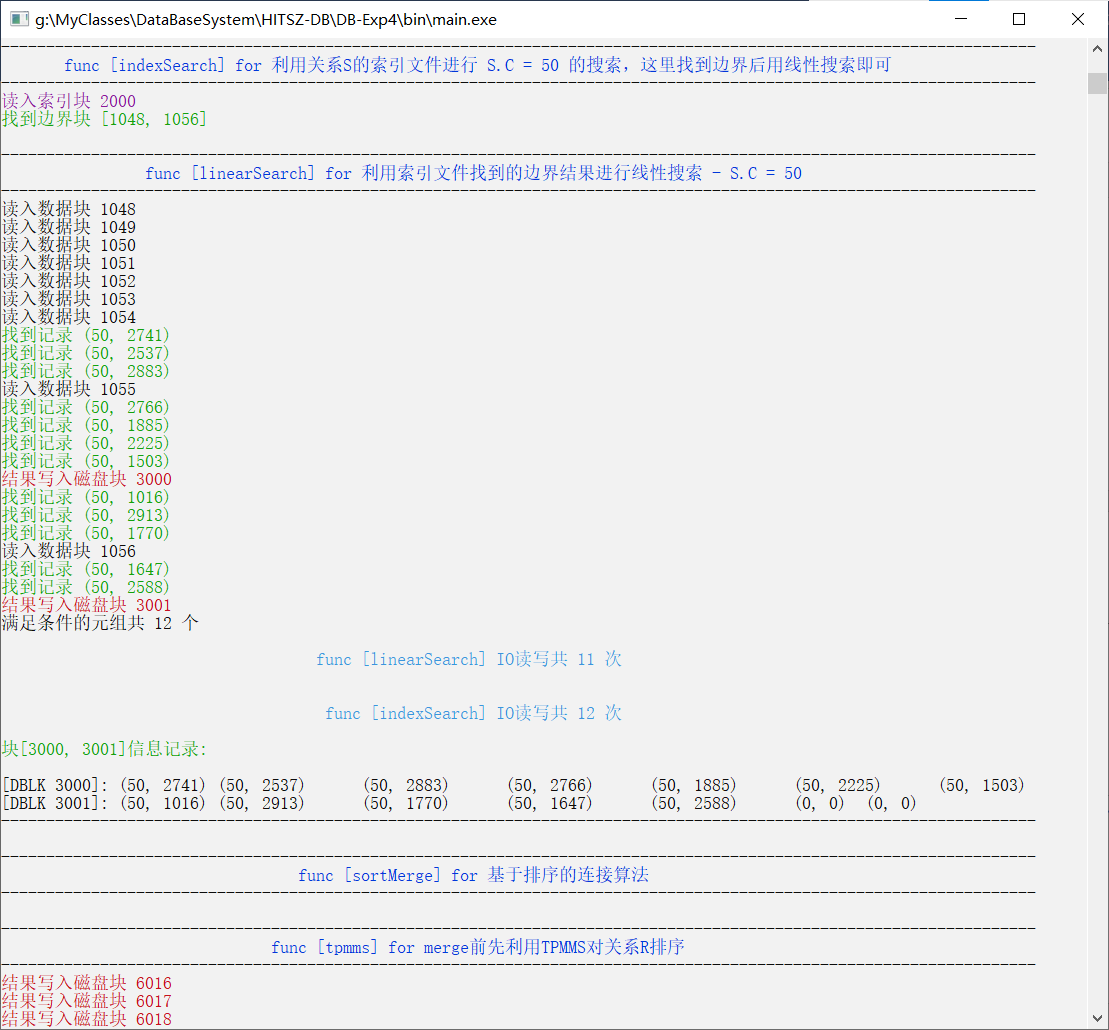


图4.8.2 基于索引的关系选择算法结果

从上图中，我们可以看到，基于索引的关系选择算法将I/O次数从任务一的34次降到了12次，这充分体现了针对聚类数据建立索引的优越性。

1. **实现基于排序的连接操作算法（Sort-Merge-Join）**

* 问题分析

本题目要求实现基于排序的连接操作算法，那么非常朴素的想法便是分别扫描排序后的关系S和关系R，然后寻找关系S与关系R的可连接记录，然而，这样做的I/O开销无疑是巨大的，那么，是否能够通过改进TPMMS的第二趟归并算法，从而做到仅仅扫描关系S和关系R一次就能够输出它们的连接关系呢？现在假设我们已将关系S和关系R分组，并完成了内排序、CompareBlk、WriteBlk的初始化，如图4.9所示[[4]](#footnote-4)，其中，深紫色方块代表关系S的记录，浅紫色方块代表关系R的记录，方块上的数字代表记录中的键值元素。

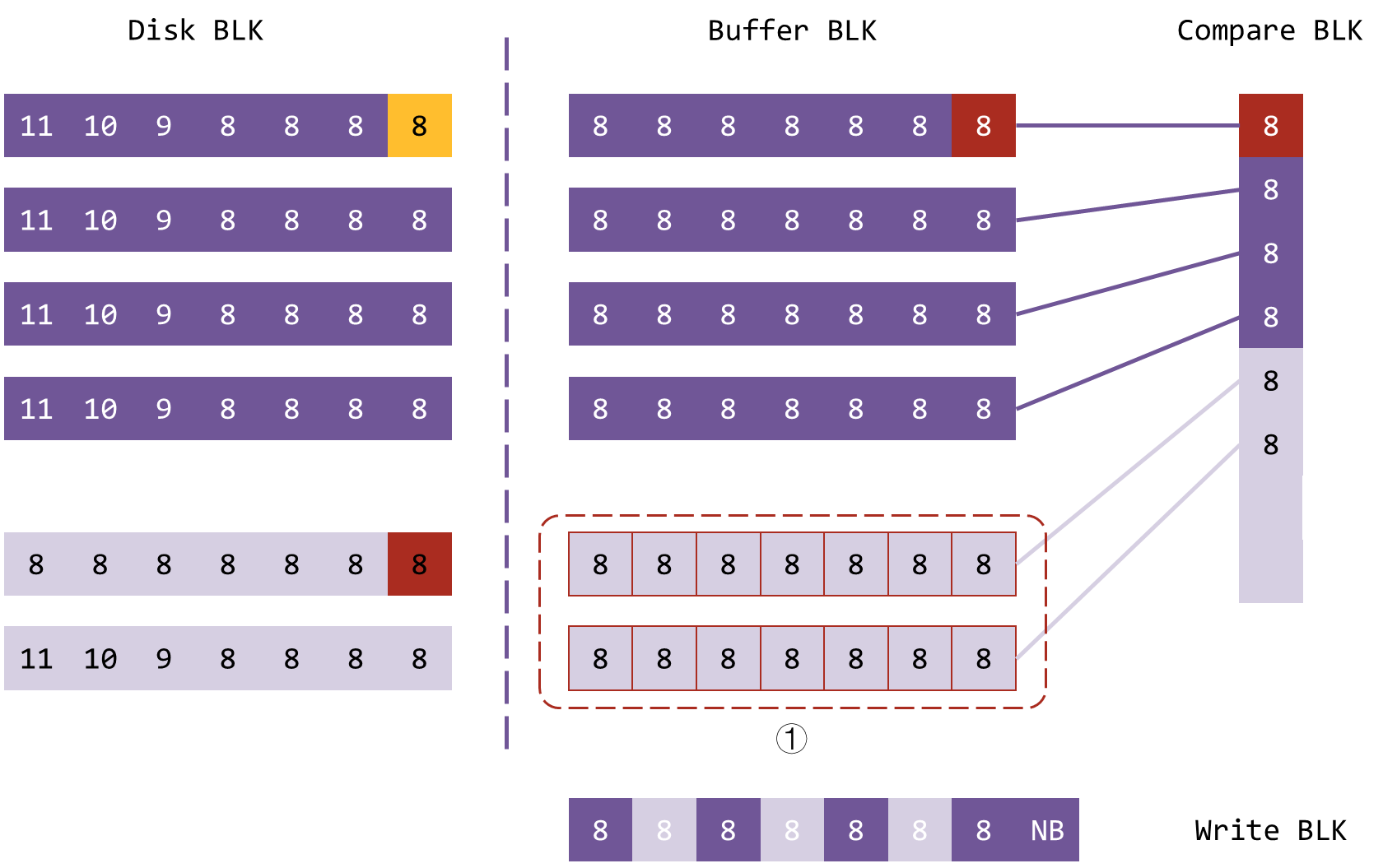


图4.9 一种R与S连接的情形

从上图中，我们可以看到，关系S的由红色方块标识的记录为了与关系R做连接操作，必然会与①中的记录做连接，当扫描完①中的记录后，必然将读入关系R的下一个数据块，因为可与S中红色方块标识记录做连接的记录并非都在①中，例如，关系R中用红色标识的记录。问题就此产生，由于TPMMS的第二趟扫描算法不存在回溯，那么我们便丢失了①中的记录，但是关系S的由黄色标识的记录却仍然能够与①中的记录进行连接，这意味着后续关系S的数据块中，与数据块①的连接元组将全部丢失，由此，说明了我们无法通过改进TPMMS算法来实现连接操作。

退而求其次，既然不能只通过两趟扫描来解决连接问题，那么我们就需要充分利用关系R与关系S的排序结果来尽可能降低I/O复杂性。事实上，由于关系R与关系S均已被排序，那么我们便不需要扫描所有的关系R，基本思路是：当我们发现关系R中的某条记录键值大于关系S的待连接记录键值时，我们便可以停止扫描关系R，转而去寻找关系S下一条记录的连接。算法描述如下：

|  |
| --- |
| **Algorithm for Task 4** |
| 1. 利用**getNewBlockInBuffer**初始化写数据块WriteBlk； 2. 利用**readBlockFromDisk**向Buffer中读入关系S的一个数据块； 3. 对于该数据块中的每一条记录RecordS：    1. 利用**readBlockFromDisk**向Buffer中读入关系R的一个数据块；    2. 对于该数据块中的每一条记录RecordR，若： 4. RecordS.C < RecordR.A，则利用**freeBlockInBuffer**释放读入的关系R的数据块，并执行步骤3.5； 5. RecordS.C = RecordR.A，则将记录RecordS写入WriteBlk，若WriteBlk写满，则将其写回磁盘；接着，将记录RecordR写入WriteBlk，同样，若WriteBlk写满，则将其写回磁盘；    1. 利用**freeBlockInBuffer**释放读入的关系R的数据块；    2. 若关系R的数据块还未读完，则执行步骤3.1；    3. 检查下一个RecordS与关系R的连接； 6. WriteBlk中是否还有未写完的记录？若是，则把WriteBlk再次写回磁盘中； 7. 释放WriteBlk占用的Buffer空间； |

算法4.1 基于排序的连接操作算法

从上述算法中，我们可以看到，由于关系R与关系S已经排序，因此，当“RecordS.C < RecordR.A”时，关系R的后续磁盘块中便不可能出现RecordS.C = Record.A的情况了，于是我们便不用对关系R做进一步的扫描。图4.10.1显示了这种情况。

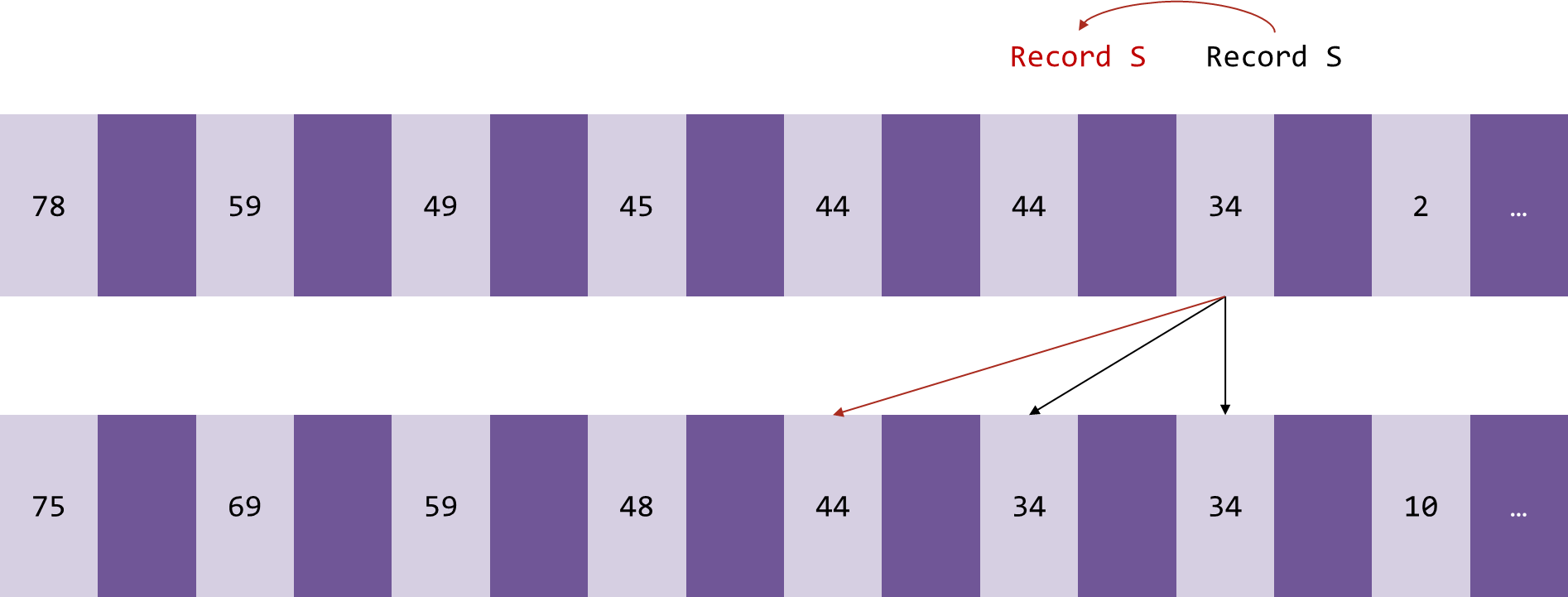


图4.10.1 算法4.1思想

事实上，算法4.1给出的策略还具有优化空间。在算法4.1中我们仅仅考虑了“RecordS.C < RecordR.A”的情况，而忽视了“RecordS.C > RecordR.A”的情况。可以预见，当RecordS越往后，其值RecordS.C越大，那么需要扫描的关系R的数据块就会更多，但我们对那些Record.A明显小于RecordS.C的数据块的扫描显然是无意义的。幸运的是，由于关系R与关系S均已排序，因此，我们可以充分利用“上一次连接的信息”——我们可以记录在上一次连接中，关系R的**最后一个**RecordR所在的磁盘块号uiDBLKNumLastR，其中，RecordR满足“RecordS.C > RecordR.A”。由于关系S也已排序，因此下一个RecordS必然大于RecordR.A，于是，我们从uiDBLKNumLastR开始扫描即可。图4.10.2显示了改进算法思想。

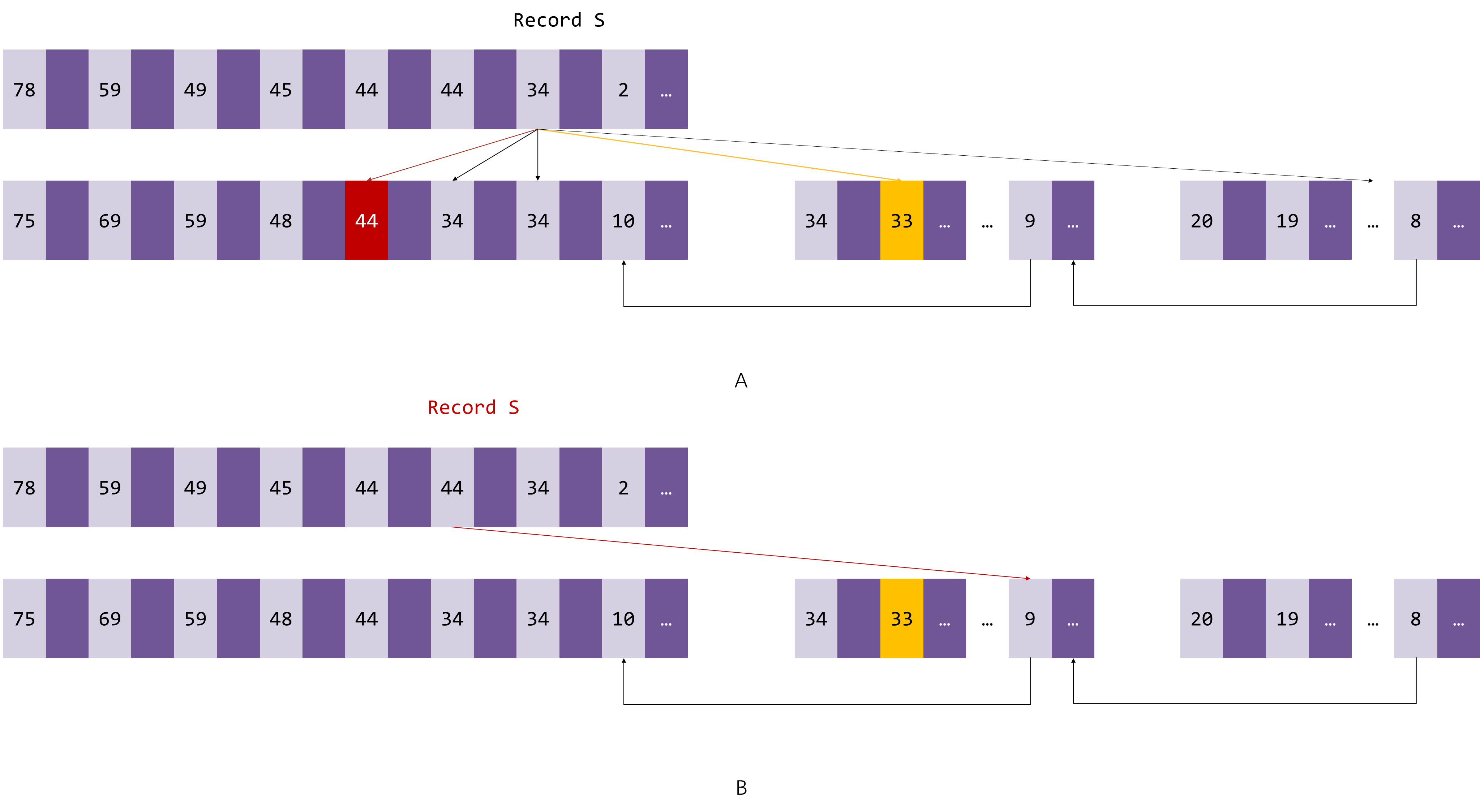


图4.10.2 改进算法4.1思想

图4.10.2 A表示上一次连接操作时，扫描到33 < **34**，则记录uiDBLKNumLastR为33所在磁盘块位置（即8）；图4.10.2 B显示了当前连接操作，很明显，我们只需从关系R的uiDBLKNumLastR处开始扫描即可。算法4.2给出了详细的算法描述，其中，红色标注部分为相比算法4.1的改进之处。

|  |
| --- |
| **Algorithm for Task 4 – Optimize** |
| 1. 利用**getNewBlockInBuffer**初始化写数据块WriteBlk； 2. 利用**readBlockFromDisk**向Buffer中读入关系S的一个数据块； 3. 初始化uiDBLKNumLastR为关系R的第一块磁盘块； 4. 对于该数据块中的每一条记录RecordS：    1. 从uiDBLKNumLastR处开始，利用**readBlockFromDisk**向Buffer中读入关系R的一个数据块，其磁盘块号记为uiDBLKNumR；    2. 对于该数据块中的每一条记录RecordR，若： 5. RecordS.C < RecordR.A，则利用**freeBlockInBuffer**释放读入的关系R的数据块，并执行步骤3.5； 6. RecordS.C > RecordR.A，uiDBLKNumLastR = uiDBLKNumR； 7. RecordS.C = RecordR.A，则将记录RecordS写入WriteBlk，若WriteBlk写满，则将其写回磁盘；接着，将记录RecordR写入WriteBlk，同样，若WriteBlk写满，则将其写回磁盘；    1. 利用**freeBlockInBuffer**释放读入的关系R的数据块；    2. 若关系R的数据块还未读完，则执行步骤4.1；    3. 检查下一个RecordS与关系R的连接； 8. WriteBlk中是否还有未写完的记录？若是，则把WriteBlk再次写回磁盘中； 9. 释放WriteBlk占用的Buffer空间； |

算法4.2 基于排序的连接操作算法-优化版

最终算法实现的关键代码片段如图4.11所示。



图4.11 基于排序的连接算法关键代码实现

* 实验结果

图4.12.1显示了算法4.1的执行结果，图4.12.2显示了算法4.2优化后的执行结果。可见，优化前，I/O次数在20000次左右，优化后，I/O次数在2000次左右，I/O复杂度降低了10倍，而这仅仅是加入一个记录变量uiDBLKNumLastR所带来的结果！

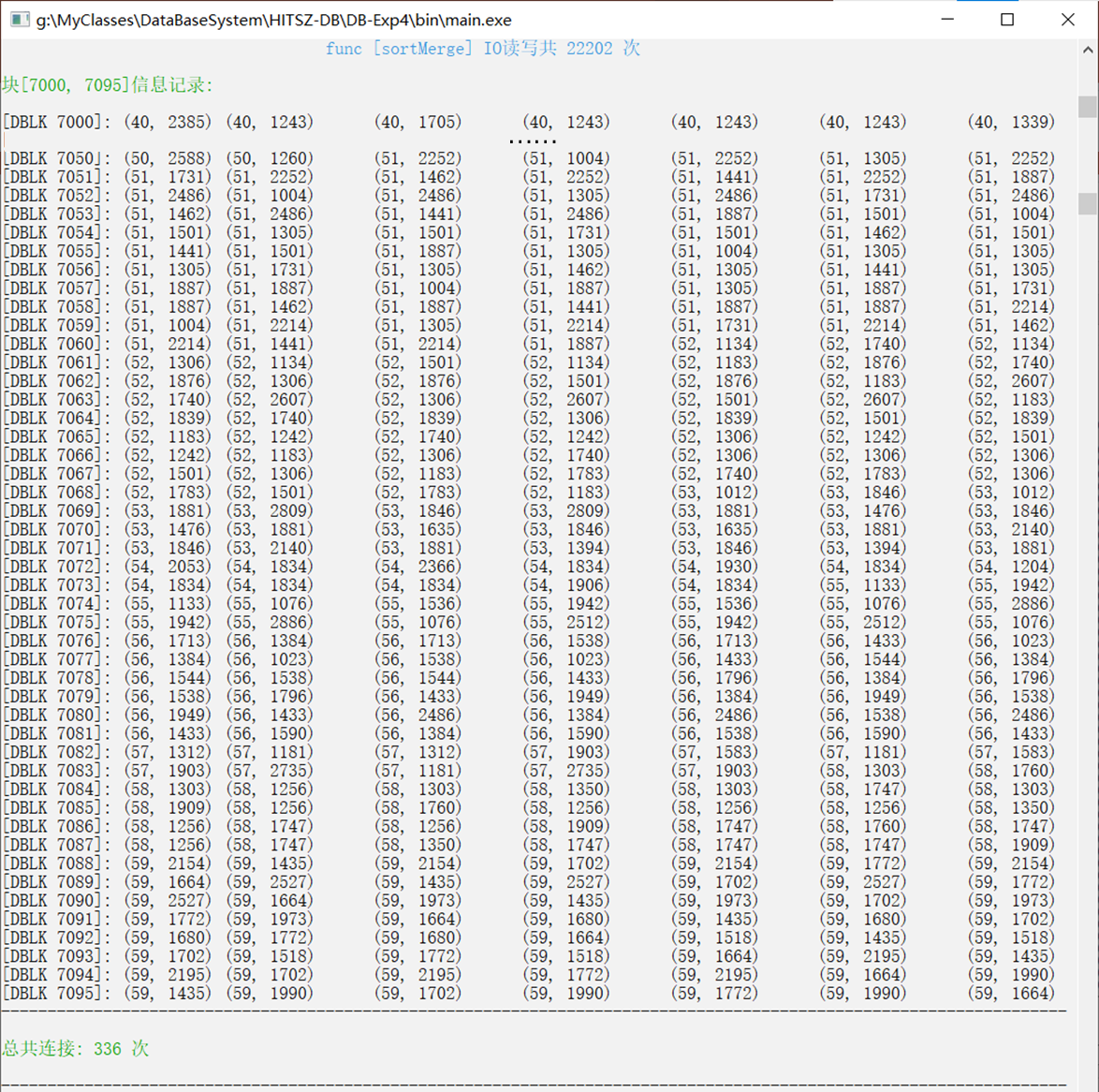


图4.12.1 算法4.1运行结果

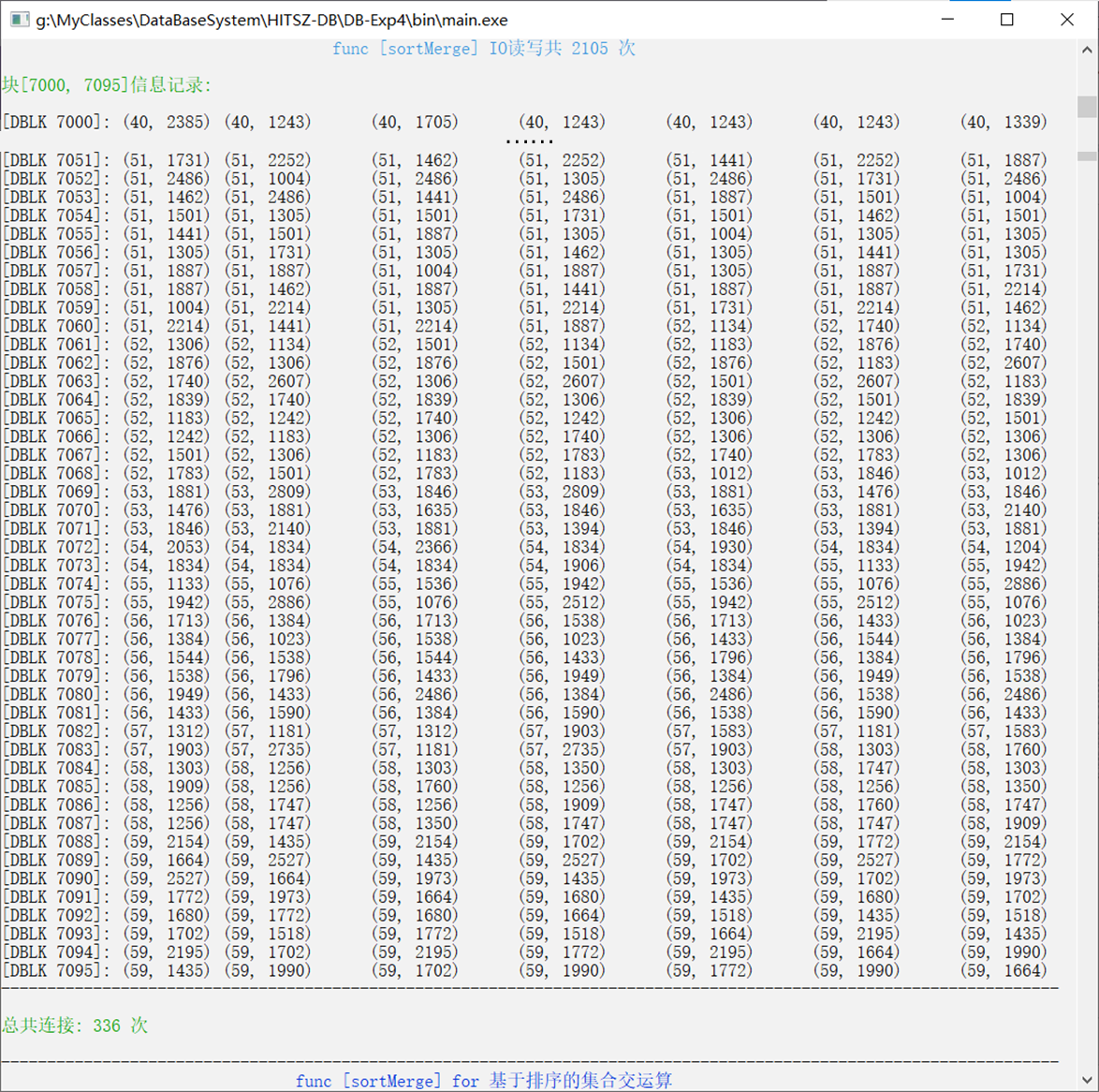


图4.12.2 算法4.2运行结果

1. **实现基于排序的两趟扫描算法，实现交、并、差其中一种集合操作算法**

* 问题分析

这里我们只对并运算进行分析，交、差运算的分析放在附加题小节。

* 实验结果

# 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。*

# 总结

*总结本次实验的遇到并解决的问题、收获及反思。*

1. NB代表Next BLK，High代表高4Byte，Low代表低4Byte [↑](#footnote-ref-1)
2. 若内排序是按照升序排序，那么此时找最小值，否则找最大值。为了描述方便，后文均采用升序。 [↑](#footnote-ref-2)
3. DBLK xx代表每一段第一个数据块的第一个块地址 [↑](#footnote-ref-3)
4. 注，图4.9中关系S和关系R对应的数据是假设的，并非实验给出的，仅将其视为一种待考虑情况。 [↑](#footnote-ref-4)