1998 International Computer Symposium
Workshop on Algorithms
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

An Efficient External Sorting Algorithm

Fang-Cheng Leu[1], Yin-Te Tsai[2], Chuan Yi Tang[1]

[1]*Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan, R.O.C.*

Email:dr814322@cs.nthu.edu.tw

[2]*Department of Computer Science and Information, Providence University, Shalu, Taiwan, R.O.C.*

Abstract

This paper presents an algorithm for external sorting with two-level memories. We only consider the number of disk I/O because the disk I/O is the bottleneck of external sorting. Our method is different from the traditional merge sort and uses the sampling information to reduce the disk I/O in the external phase. Our algorithm is elegant, simple and making good use of memory available in the computer environment now. Under certain memory constraint, the algorithm runs with optimal number of disk I/Os, where N is the number of records to sort and B is the block size.

1. Introduction

The problem of how to sort efficiently has been widely discussed. To sort extremely large data are becoming more and more important for the large cooperation, banks and government institutes, which rely on the computer more and more deeply in all aspects. In [10], the authors confirmed that sorting continue to account for roughly one-fourth of all computer cycles. Much of the time of sorting is spent by external sorts, in which the data file is too large to fit in main memory and must reside in the secondary memory. The external sorting first generates some sorted subfiles and then tries to merge these sorted subfiles into a sorted file placed in the secondary memory.

The number of I/Os is a more appropriate performance measure for the external sorting, because the I/O speed is much slower than the CPU speed. In this paper, the internal computation time will be ignored and the number of I/Os will be considered. We assume there are two storage devices and there is a single central processing unit as shown in Figure 1. We model the secondary storage as a generalized random-access magnetic disk. The input and output are on separate I/O devices, i.e., we read from one device and output the blocks to the other device. Thus, read and write operations can be concurrent since they are performed on independent device. The read operations take significantly more time than writes in our algorithm. Since the time for read operations is the bottleneck, we will evaluate our algorithm by read access time it requires. The essential problem is to select the parts of the input file that are kept in the main memory at each time. For this purpose, we fetch the block when it is needed and remove it when it is not necessary or already in its global final position.

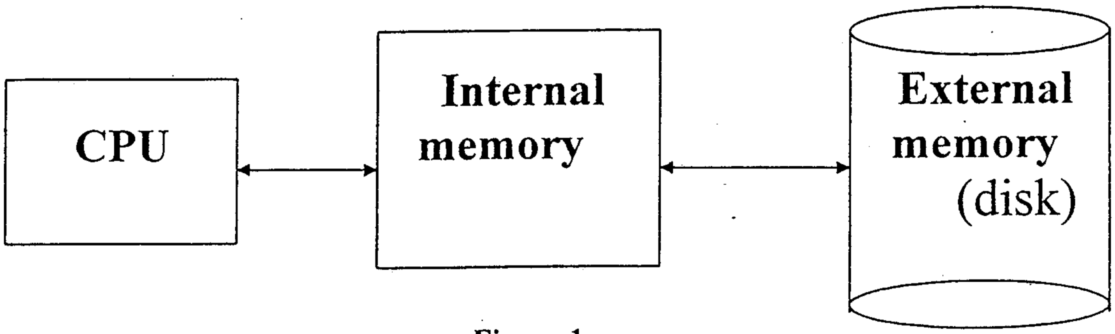

Figure 1

Basically, the external sorting algorithm consists of two phases, the internal phase and the external phase. The subfile that can fit in the main memory was sorted in the internal phase. We call such a sorted subfile a "run". The external phase sorts portions of the total file that are too large to fit in the working space at a time. Although the memories of current computers have been increasing rapidly, there still exists a need for external sorting for large file. The bottleneck in external sorting is the time needed for the input/output (I/O) operations. The reason that the I/O becomes the bottleneck is that CPU speeds are much faster than disk I/O speeds, and moreover have been growing at a much faster rate than disk speeds over the last decade.

Researchers have concentrated on the external sorting. External Mergesort has been the most popular choice, as described by Knuth [2], Singh [3], Kwan [1] and the others. Let N, M, B denote the number of records to be sorted, the number of records that can fit into main memory, and the number of records that can be transferred in a single block, respectively, where $1 \leq B \leq M < N$. The total number of runs is N/M and the total number of blocks is N/B. Each block transfer is allowed to access any contiguous group of B records on the disk.

The disk I/O complexity of k-way merge sort is $O(N \log_k N/M)$. The k-way merge sort, which chooses k as large as possible to reduce disk I/O, gets bad performance as shown in [1] because the disk I/O access time of the merge phase in merge sort increases as a function of k. Besides the external mergesort, there are other sorting algorithms. The external bubble sort is presented in [4] with $O(N^2 \log N)$ disk I/O time. The external quicksort is proposed in [5, 7]. The external sorting algorithm based directly on quicksort, designed by Monard [7], is presented in [6], which leads to the number of fetch $N/B (\log_2 N/M) -0.924$. The sorting itself is performed in an ordinary quicksort manner. The external sorting algorithm based on shuffle sort presented in [13] observes that the number of read operations needed during the execution will be $N/B (1+2\ln((N+1)/4B))$. Another types of sorting algorithms are also proposed for the external sorting problem, for example, the distributive partitioning, bucketsort, and binsort (see, [2,8,10,11,12]). They have corresponding phases as mergesort does, but they are performed in the opposite order. In [9], Aggarwal and Vitter proved that the optimal number of disk I/O for external sort is $O(\dfrac{N\log(N/B)}{B\log(M/B)})$ in one disk system.

In this paper, we proposed an external sorting algorithm with optimal $O(N/B)$ disk I/Os for $M \geq N/B + (NB)^{1/2}$. The internal phase performs the same internal sorting as in the external merge sort. In the meantime, the priority queues are used to keep the information of blocks in the main memory for reducing the I/O operations of the external phase. Each block will be read once and could decide its global final position in the external phase. The sorting algorithm totally takes twice disk I/Os for each block and we can show that our algorithm has optimal disk I/O complexity.

The remainder of this paper is organized as follows. Section 2 is devoted to the new sorting algorithm. In Section 3, we show the correctness and analysis of our algorithm. Concluding remarks are provided in section 4.

2. The Algorithm :

Our sorting algorithm includes two phases – the internal phase and the external phase. In the internal phase, we divide the input file into subfiles that can fit into the main memory and sort them using some internal sorting algorithm. We call each sorted subfile as one run and the total number of runs is N/M. After run i is generated, we store the pointers of the blocks in a min-priority queue Q_i using the blocks' smallest values as their keys, where $1 \leq i \leq N/M$. Hence, the disk I/O time is much more important than computing time, as long as we have enough room to keep all the information in memory. The information in the priority queues will be very useful for the external phase to reduce disk I/O. According to the priority queues, we could read the blocks in order from the disk and write to their sorted position in the external phase. We could make sure that N/B disk I/Os are needed in the external phase in all cases. Aggarwal and Vitter [9] proved that the optimal of disk I/Os for external sort is $O(\dfrac{N\log(N/B)}{B\log(M/B)})$ in one disk system. If $M \geq (NB)^{1/2}$, it becomes $O(N/B)$. Our internal phase needs N/B disk I/Os and the external phase also needs N/B disk I/Os. Our sorting algorithm takes $2N/B$ disk I/Os and it meets the lower bound. Thus the disk I/O complexity is optimal. Our sorting algorithm takes 2 disk I/Os for each block in all cases.

Our algorithm is designed based upon above idea. Without loss of generality, we assume that each record of the file is distinct and our goal is to sort these records in ascending order. In the internal phase, we read M/B blocks from disk to main memory and select an internal sorting method, such as quick sort or bubble sort, to sort these blocks internally to produce runs. Each run consists of M/B blocks with sorted records. Before writing a run to the disk, we construct the priority queue for each run. Let Front(Q_i) denote the smallest key value in Q_i. After finishing the internal phase, the priority queue Q_i of each run is constructed as shown in Figure 2. The number of priority queues equals to the total number of runs, *N/M*. We use Front(Q_i) to decide the order of blocks to be read into main memory in the external phase. The corresponding block of $\min_i\{$ Front(Q_i)$\}$ will be read into main memory first. When one block is read into main memory, we use the smallest record of this block as a pivot and compare it with the records already in main memory. In Figure 3, the first block of Run 2 will be read first and the pivot is 5, then the priority queues have to be reconstructed. The records in main memory smaller than the pivot are already in their final sorted order because no record still in the disk is smaller than the pivot. Then the first block of run 1 will be read into main memory. The pivot is 8 and the priority queues have to be reconstructed as shown in Figure 4. We perform a merge process for all the records in main memory, then the records in main memory are in partial order. The records in main memory which are smaller than the pivot could be written back to the disk in block. We continue to find the block by using Front(Q_i) and to read the block into main memory. Then choose the pivot, write blocks in final position into disk and proceed the merge process until all the blocks are processed. During the external phase, each block has to be read into main memory once, thus there are *N/B* disk I/Os in the external phase. Another advantage of our algorithm is that some records written to disk are already in their final position during the external phase.

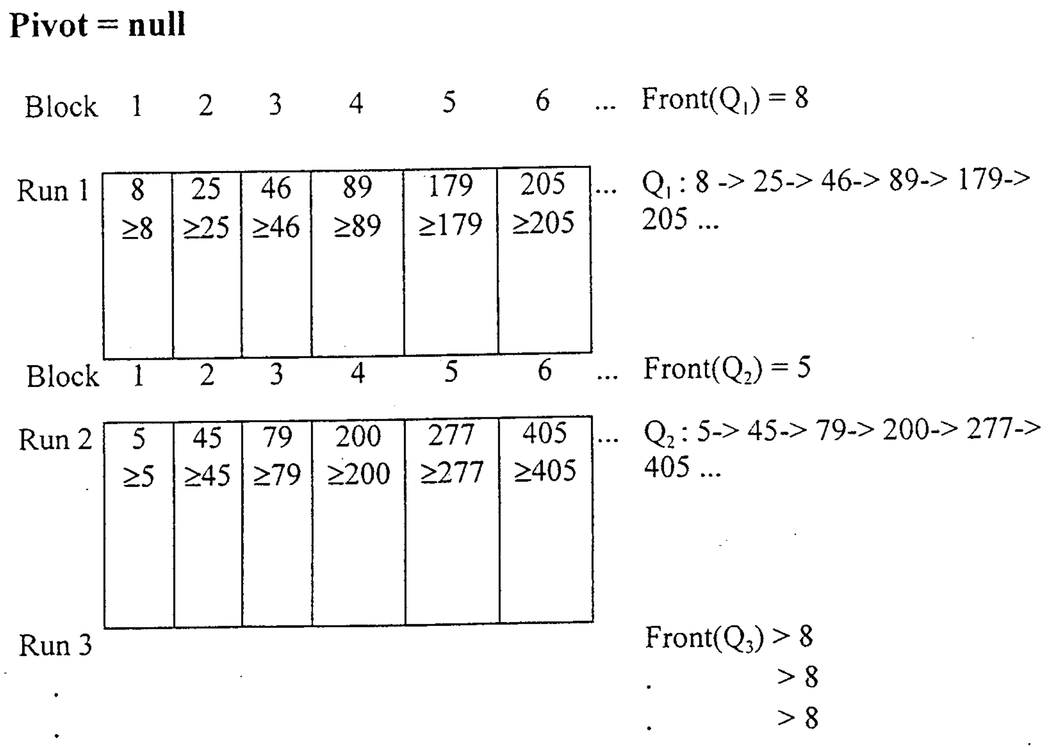

Figure 2

Pivot = 5

Block	1	2	3	4	5	6	...	Front(Q_1) = 8
Run 1	8 ≥8	25 ≥25	46 ≥46	89 ≥89	179 ≥179	205 ≥205	...	Q_1 : 8 -> 25-> 46-> 89-> 179-> 205 ...
Block	1	2	3	4	5	6	...	Front(Q_2) = 45
Run 2	_	45 ≥45	79 ≥79	200 ≥200	277 ≥277	405 ≥405	...	Q_2 : 45-> 79-> 200-> 277-> 405 ...
Run 3								Front(Q_3) > 8
.								. > 8
.								. > 8

Figure 3

Pivot = 8

Block	1	2	3	4	5	6	...	Front(Q_1) = 25
Run 1	_	25 ≥25	46 ≥46	89 ≥89	179 ≥179	205 ≥205	...	Q_1 : 25-> 46-> 89-> 179-> 205 ...
Block	1	2	3	4	5	6	...	Front(Q_2) = 45
Run 2	_	45 ≥45	79 ≥79	200 ≥200	277 ≥277	405 ≥405	...	Q_2 : 45-> 79-> 200-> 277-> 405 ...
Run 3								Front(Q_3) > 8
.								. > 8
.								. > 8

Figure 4

Our algorithm is stated as follows:

```
Begin
/* Internal Phase */
For $i$ = 1 To $N/M$ Step 1
        Read next $M$ records into main memory;
        Use internal sorting to produce run $i$;
        Construct a priority queue $Q_i$ with $M/B$ keys
        which are the smallest records of $M/B$ blocks in
        run i, respectively;
        Write run $i$ to the disk;
End For
/* External Phase */
Clear buffer $b1$;
Repeat
        Let $K_{min}$ = $\min_i$ { Front ($Q_i$) } ;
        Let $B_{min}$ = the corresponding block of $K_{min}$;
        Let $Q_{min}$ = the queue with the key $K_{min}$;
        Read block $B_{min}$ to buffer $b2$;
        Delete($Q_{min}$) ;
        Let $pivot$ = the smallest record of $b2$ ;
        Let $S = \{X \mid X \leq pivot \text{ and } X \in b1\}$
        While $|S| \geq B$
                Let $S'$ = the first $B$ smallestrecords of $S$ in
                the ascending order;
                Write $S'$ to disk;
                $S = S - S'$;
        End While
        Merge the records in $b1$ and $b2$, and place the
        results into $b1$;
Until all $Q_i$ are empty
Write b1 to disk;
End
```

3. Analysis of the algorithm

In this section, we shall prove the correctness, I/O complexity and memory requirement of our algorithm. Now we show why the algorithm works.

Lemma 1. After the external phase, all of the records in the disk are sorted.

Proof: The records in each run are in the ascending order after the internal sorting. In the external phase, the algorithm considers a block B_{min} in one iteration. All of the records in memory buffer $b1$ which are smaller than pivot are placed in the disk and in their final positions after each iteration. We know that the pivot of the *nth* iteration is smaller than the pivot of $(n+1)st$ iteration in the external phase. After each block is read into main memory and proceed in the external phase, all of the records will be written to disk and in their final position. **Q.E.D.**

Lemma 2. The sorting algorithm takes $2N/B$ disk I/Os.

Proof: To generate one run needs M/B disk I/Os. In the internal phase, the total number of disk I/Os for generating N/M runs is $(M/B)*(N/M) = N/B$. In the external phase, one block is read into main memory in one iteration and in the order of the key in the min-priority queue. It is easy to know that the external phase performs N/B disk I/Os. Thus our sorting algorithm takes $2N/B$ disk I/Os. **Q.E.D.**

Next we show the memory requirement of the algorithm.

Lemma 3. The amount of main memory needed for our sorting algorithms at least $N/B + (NB)^{1/2}$.

Proof: First of all, we consider the memory requirement for the priority queue Q_i. There are N/M queues and each queue stores M/B records. Then the total memory space for N/M queues is N/B. Therefore M will be at least N/B.

Since we can show later that there are at most N/M blocks kept in main memory for the external phase, we get $M \geq (N/M) \times B$ for the external phase. It turns out that $M \geq (NB)^{1/2}$. Based upon the above discussion, we can conclude that $M \geq N/B + (NB)^{1/2}$.

In the following, we show that in the external phase, at most N/M blocks are needed to keep in main memory. Assume that there are already N/M blocks in main memory. Thus these blocks must be from the different runs. Suppose that the new pivot is from run j. Since there are total N/M runs, one of the old N/M blocks must be from the same run, run j. This block will be written back to disk because that the new pivot is larger than it in run j. Therefore at most N/M blocks will be left in the main memory. **Q.E.D.**

Theorem 1. The algorithm performs optimal $O(N/B)$ disk I/Os for $M \geq N/B + (NB)^{1/2}$.

Proof: The optimal of disk I/Os for external sort is $O(\frac{N\log(N/B)}{B\log(M/B)})$ in one disk system. If $M \geq (NB)^{1/2}$, it becomes $O(N/B)$. By Lemma 1 and Lemma 2, we get that our sorting algorithm takes $2N/B$ disk I/Os and it meets the lower bound. Thus our algorithm is optimal. **Q.E.D.**

4. Conclusion :

We have proposed an optimal sequential external sorting algorithm by using the elegant sampling technique to optimize disk I/O. The algorithm takes exactly $2N/B$ disk I/Os and could make sure the order of some records during the sorting process. Table 1 lists the file size and corresponding memory size for running our algorithm, where the block size and record size are 1K bytes and 4 bytes, respectively. The table indicates that the memory requirement can be realized in the current and future computer environment.

An interesting open problem is whether the idea of the algorithm is helpful for parallel external sorting in distributed memory environment to overlap the disk I/O and communication.

File size (MB)	Memory size (MB)
10	≥ 0.14
100	≥ 0.72
500	≥ 2.70
1,000	≥ 5.00
2,000	≥ 9.41
5,000	≥ 22.23
10,000	≥ 43.16

Table 1

Reference

[1] S. C. Kwan and J. Baer, " The I/O performance of Multiway Mergesort and Tag Sort," in IEEE Trans. Comput., vol c-34, NO. 4, April 1985, pp.383 - 387

[2] D. E. Knuth, The Art of Computer Programming, VOL. 3: Sorting and Searching. Reading, MA: Addison-Wesley,1973.

[3] B. Singh and T. L. Naps, " Introduction to Data Structure," West Publishing Co., St. Paul. MN (1985).

[4] W. R. Dufrene and F. C. Lin, "An Efficiency Sort Algorithm with no Addition Space," in The Computer Journal, vol. 35, NO. 3,1992.

[5] A. I. Verkamo, "External Quicksort," Performance Evaluation 8, 271 - 288 (1988)

[6] G.H. Gonnet, Handbook of Algorithms and Data Structures (Addison-Welsey, Reading, MA, 1984) 160-162

[7] M. C. Monard, Projecto e Analise de Algorithm de Classificacao Externa Baseados na Estrategia di Quicksort, Ph.D. Thesis, Pontificia Univ. Catolica, Rio de Janeiro, Brazil, 1980

[8] A. I. Verkamo, "Performance Comparison of Distributive and Mergesort as External Sorting Algorithms" The Journal of Systems and Software 10, 187 - 200(1989)

[9] A. Aggarwal and J. S. Vitter, " The Input/Output Complexity of Sorting and Related Problems," Comm. ACM, vol 31 NO. 9 Sept. 1988, pp. 1116 - 1126.

[10] E. E. Lindstorm, and J. S. Vitter, "The design and analysis of BucketSort for bubble memory secondary storage. IEEE Trans. Comput. C - 34, 3(Mar. 1985), 218 - 233.

[11] W. Dobosiewicz, "Sorting by Distributive Partitioning" , Information Processing Letters 7(1),1-6(1978).

[12] B.W Weide, "Statistical Methods in Algorithm Design and Analysis" , Carnegie-Mellon University Technical Report CMU-CS-78-142, 1978,pp.3-30-3-39.

[13] D.Motzkin and C.Hansen , "An efficient external sorting with minimal space requirement" , Internat.J Comput . & Inform. Sci 11(6)(1982)391-392.