Greed Sort:
An Optimal External Sorting Algorithm for Multiple Disks

Mark H. Nodine and *Jeffrey Scott Vitter*

Technical Report No. CS–91–20
(revised version of Technical Report No. CS–90–04)

Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks*

Mark H. Nodine and *Jeffrey Scott Vitter*

Dept. of Computer Science
Brown University
Providence, R. I. 02912–1910

Abstract

We present an optimal deterministic algorithm for external sorting on multiple disks. Our measure of performance is the number of input/output (I/O) operations. In each I/O, each disk can simultaneously transfer a block of data. Our algorithm improves upon the randomized optimal algorithm in [ViS] and the (non-optimal) commonly-used technique of disk striping. The code is simple enough for easy implementation.

*Support was provided in part by an National Science Foundation Presidential Young Investigator Award CCR–8906419 with matching funds from IBM, by an IBM Graduate Fellowship, by NSF research grant CCR–9007851, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–83–K–0146 and ARPA order 6320, amendment 1.

1 Introduction

Sorting consumes roughly 20 percent of computing resources in large-scale installations [LiV]. Of particular importance is external sorting, in which the records to be sorted are too numerous to fit in the processor's main memory and instead are placed on secondary storage. The bottleneck in external sorts is the input/output (I/O) operations. Typically data are transferred in units of *blocks*, which may consist of several kilobytes. This approach takes advantage of the fact that the seek time is usually much longer than the time needed for transferring a record of data once the head is positioned. An increasingly popular way to get further speedup is to use many disk drives working in parallel. This is exactly the approach taken, for example, in TWA's flight reservation system [GiS].

Initial work in the use of parallel block transfer for sorting was done by Aggarwal and Vitter [AgV]. In their model, they considered the parameters

$$\begin{aligned} N &= \#\text{ records in the file} \\ M &= \#\text{ records that can fit in internal memory} \\ B &= \#\text{ records per block} \\ P &= \#\text{ blocks transferred per I/O} \end{aligned}$$

where $M < N$, and $1 \le PB \le M/2$. In each I/O, P blocks of B records can be transferred simultaneously, as illustrated in Figure 1. They proved that the average-case and worst-case number of I/Os required for sorting is[1]

$$\Theta\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right). \tag{1}$$

Their lower bound is based on routing arguments. They gave two algorithms, a modified merge sort and a distribution sort, that each achieved the optimal I/O bounds. Likewise, they showed that the number of I/Os required to transpose a $p \times q$ matrix is

$$\Theta\left(\frac{N}{PB}\left(1+\frac{\log\min\{M,\min\{p,q\},N/B\}}{\log(M/B)}\right)\right). \tag{2}$$

Vitter and Shriver considered a more realistic *P-disk model*, in which the secondary storage is partitioned into P physically distinct disk drives [ViS], as in Figure 2. (Note that each head of a multi-head drive could count as a distinct disk in this definition, as long as each could operate independently of the other heads on the drive.) In each I/O operation, each of the P disks can simultaneously transfer one block of B records. Thus, P blocks can be transferred per I/O, as in the [AgV] model, but only if no two blocks access the same disk. This assumption is very reasonable in light of the way real systems are constructed. Vitter and Shriver presented a randomized version of distribution sort using two interesting partitioning techniques. Their

[1] We use the notation $\log x$ to denote the quantity $\max\{1, \log_2 x\}$.

algorithm meets the I/O bound (1) for the more lenient model of [AgV], and thus the algorithm is optimal. They posed as an open problem the question of whether there is an optimal algorithm that is deterministic. They also showed that matrix transposition can be done optimally in the P-disk model, achieving the bound (2).

Disk striping is a commonly-used technique in which the P disks are synchronized, so that the P blocks accessed during an I/O are located at the same relative position on each disk. This technique effectively transforms the disks into a single disk with larger block size $B' = PB$. Merge sort combined with disk striping is deterministic, but the number of I/Os used can be much larger than optimal, by a multiplicative factor of $\log(M/B)$.

In the area of parallel computing, Leighton introduced the columnsort algorithm as an optimal sorting algorithm on linear-sized sorting networks [Lei]. It was pointed out in [AgV] that columnsort can be used for optimal sorting with respect to I/O if N and B are not too large. Since columnsort is a (non-adaptive) sorting network algorithm, its I/O schedule can be pre-specified independently of the data to take full advantage of parallel block transfer.

In this paper, we answer the open question posed in [ViS] and present an optimal deterministic sorting algorithm. The algorithm, which we call *Greed Sort*, is an interesting variant of merge sort. In each merge pass, a priority scheme guarantees that each record ends up sufficiently close to its correct position, so that another pass of columnsort can complete the merging.

2 The Greed Sort Algorithm

We start by defining an order on the locations on the disks. Let $B_{i,j}$ denote the ith block stored on disk j, and let $R_{i,j,k}$ denote the kth record in $B_{i,j}$. We order the locations so that

- Record $R_{i,j,k}$ precedes $R_{i,j,k+1}$ for all i, j, k.
- Block $B_{i,j}$ precedes block $B_{i,j+1}$ for all i, j.
- Block $B_{i,P}$ precedes block $B_{i+1,1}$ for all i.

Now we can describe the sorting problem.

Problem instance: A series of N records, each containing a key field, located in the first N locations on the disk.

Goal: To permute the records so that the key fields are in non-decreasing order in the first N locations on the disk.

Figure 3 gives an example of a sorted list with $B = 1$.

Our Greed Sort algorithm is a variant of merge sort. We create initial runs of size N/M by repeatedly reading in a memoryload, sorting it, and writing it back to the

disks. In each subsequent pass, we merge together at least $R = \sqrt{M/B}/2$ sorted lists, called *runs*, at a time to form a single sorted list.

The basic idea behind Greed Sort is the following: Let us assume that each of the R runs to be merged is stored consecutively on disk, beginning on disk 1 and cycling through the P disks. In each parallel read operation, the one or two "best" available blocks from each disk are read into primary memory: the block with the smallest minimum key value and the block with the smallest maximum key value. Treating each of the P disks independently, we sort its $2B$ records in primary memory and write the smallest B to the output list that we're forming; we put the largest B records back at the front of the run from which the smallest minimum was taken (note that this run remains sorted after this operation). See Figure 4. Ties are broken arbitrarily. If the blocks with the smallest minimum and the smallest maximum are the same block, we read only the one block and write it to the output list.

This operation results in an "approximately merged" list. The crucial observation is that the records are within $RPB = P\sqrt{MB}/2$ positions of their correct sorted locations. By an appropriate use of clustering throughout the course of the algorithm, this approximately merged list can be completely merged by a single pass consisting of several applications of the columnsort algorithm of Leighton [Lei] to subfiles of size $P\sqrt{MB}$. Then the next merge begins.

Columnsort is easiest to visualize as sorting into column-major order a matrix with r rows and c columns, with the requirement that c divides r and $r > 2(c-1)^2$. It has eight steps, of which the odd-numbered steps are all the same: sort all the records in each column. Steps 2 and 4 are a transpose operation as shown in Figure 5. Steps 6 and 8 amount to a cyclical shift by $r/2$, and are shown in Figure 6.

The pseudocode for the Greed Sort algorithm is given below. Since all the records within a run are sorted, consecutive blocks on the same disk from a run are in non-decreasing order. Thus, the best available block on a given disk must be the next unread block on that disk from some run. The collection of next unread blocks, one for each (run, disk) pair, is called the *candidate set* of blocks and is stored in a priority queue. There are P disks and R runs, so the candidate set has cardinality PR.

We assume for convenience that the runs are separated on each disk by blocks containing the key value $+\infty$ larger than any key. The algorithm uses $mark[i,j]$ to keep track of the next block of run i to be read from disk j. The set of all blocks $mark[i,j]$ comprises the candidate set. The maximum and minimum key fields of block $mark[i,j]$ are stored in $biggest[i,j]$ and $smallest[i,j]$, respectively. We do our I/O operations into the buffers $b1$ and $b2$, which each consist of P smaller buffers. Buffers $b1[j]$ and $b2[j]$, for $1 \le j \le P$, each hold B records from disk j; we denote their maximum and minimum keys by $\max(b1[j])$, $\min(b1[j])$, $\max(b2[j])$, and $\min(b2[j])$.

```
algorithm Greed Sort;
{ Create the initial runs }
repeat N/M times
    read the next M records into primary memory, PB at a time
    sort the M records internally
    write the M records back onto disk, PB at a time
end repeat

repeat until only 1 run is left
    { Merge together R = sqrt(M/B)/2 runs at a time }
    repeat until all runs have been processed
        R := sqrt(M/B)/2
        pick the next R runs to process
        for i := 1 to R do
            read in parallel the first P blocks of run i into buffer b1
            for j := 1 to P do
                mark[i, j] := 1
                biggest[i, j] := max(b1[j])
                smallest[i, j] := min(b1[j])
            end for
        end for

        { Do an approximate merge of the R runs }
        repeat until all records of the R runs have been processed
            for j := 1 to P do
                { Determine the best one or two blocks to read from each disk }
                bestrun1[j] := i such that biggest[i, j] is a minimum
                bestrun2[j] := i such that smallest[i, j] is a minimum
            end for
            for j := 1 to P do in parallel
                read block mark[bestrun1[j], j] of run bestrun1[j] from disk j into buffer b1[j]
                if bestrun1[j] ≠ bestrun2[j] then
                    read block mark[bestrun2[j], j] of run bestrun2[j] from disk j into buffer b2[j]
                    merge b1[j] and b2[j] in place internally
                    write b2[j] to block mark[bestrun2[j], j] of run bestrun2[j] on disk j
                endif
                write b1[j] to disk j in the output list
                read block mark[bestrun1[j], j] + 1 of run bestrun1[j] from disk j into buffer b1[j]
            end for
```

```
            for j := 1 to P do
                { Update the internal data structures }
                mark[bestrun1[j], j] := mark[bestrun1[j], j] + 1
                biggest[bestrun1[j], j] := max(b1[j])
                smallest[bestrun1[j], j] := min(b1[j])
                if bestrun1[j] ≠ bestrun2[j] then
                    smallest[bestrun2[j], j] := min(b2[j])
                end if
            end for
        end repeat

        { Do a restorative pass to turn the approximate merge into a complete merge }
        L := P√(MB)/2
        T := total # of records in the R runs
        for n := 0 to T/L − 2 do
            use columnsort to sort records nL + 1, nL + 2, ..., nL + 2L of the run
        end for
    end repeat
end repeat
```

2.1 Proof of Correctness

The correctness of Greed Sort depends on showing that each merge pass correctly merges the $R = \sqrt{M/B}/2$ runs into a single sorted run.

Theorem 1 *Each stage of merging in the Greed Sort algorithm correctly merges R runs.*

Theorem 2 below shows that each record in the approximately merged output list formed from the R runs is at most $L = RPB$ locations from its correct sorted location. The columnsorts are done on successive overlapping subfiles of size $2L$ to complete the sorting as shown in Theorem 3. This proves Theorem 1 (conditionally, of course, on proving Theorems 2 and 3).

Theorem 2 *In the approximately merged output list formed from R runs, each record is at most RPB locations from its correct sorted location.*

This theorem is proved using Lemmas 1–3. First we start with a definition.

Definition 1 A sequence is called *L-regressive* if for any two elements $x < y$, y does not precede x by more than L elements in the sequence.

Lemma 1 *For any two records $x < y$ written to disk j in the approximately merged output list, x will be located at most $R - 1$ blocks later than y on disk j.*

Proof: Let record y be written to disk j of the output list at step t. (See Figure 4.) (By step t, we mean the tth time PB records are written to the output list being formed.) We claim that (a) there is no run i such that $biggest[i,j] < y$, and (b) there is at least one run i such that $smallest[i,j] \geq y$. Claim (a) is true since every record written to the output has a value no bigger than the block that was chosen has having the smallest maximum. To show claim (b), we consider the two cases: (1) step t reads in two distinct blocks from disk j, and (2) step t reads in only one block from disk j. In the former case, the run that gets records put back onto it has no record on disk j that is less than y by construction since only the smallest B records were written to the output list; in the latter case the run from which the block was read has no records on disk j that are less than y since it was a sorted run. So in either case, there is a run whose minimum is at least y. From claim (a), no run has more than one block containing records less than y. Claim (b) can now be invoked to show that any record smaller than y will be written to the output list within the next $R-1$ blocks. The reason is that any block containing a value less than y will have a smaller minimum than the block whose minimum is at least y. Each of the $R-1$ steps after step t is guaranteed to reduce the number of blocks containing values less than y by at least one. Since at most $R-1$ blocks contained values less than y (combining claims (a) and (b)), all values less than y will be written to the output list in the $R-1$ blocks following the one containing y. □

Lemma 2 *The approximately sorted output list is RPB-regressive.*

Proof: Let y be output on disk j. By Lemma 1, any block containing a value less than y on disk j will be output in the next $R-1$ time steps. Any block on a disk $i < j$ containing a value less than y will also either be output at the same time or within the next $R-1$ time steps as suggested by Figure 7, since no run has more than one block containing values less than y. Furthermore, any disk $i > j$ has already written all records with keys less than y. Since each "stripe" on the collection of disks contains PB records, the maximum regression is RPB. □

Lemma 3 *If a list is L-regressive, then every record is at most L locations from its correct sorted location.*

Proof: For simplicity, let all the records have unique keys. Assume that y is the jth smallest record and occurs at position i where $i < j - L$, as in Figure 8. We derive a contradiction by showing that there exists an $x < y$ that succeeds y by more than L records. There are $j-1$ records less than y. In order to meet the L-regressive condition, they must all occur in the range $1, \ldots, i+L$. There are $i+L-1 < j-1$ locations that can contain values less than y without violating the L-regressive condition. Thus, at least one record less than y must be out of the desired range. This same argument also shows that the jth record cannot be at any location $i > j+L$. When duplicate keys are allowed, the argument gets a little more complicated, since we cannot talk about the jth record exactly. The fact that remains

true, however, is that every record is within L of the closest place it can acceptably go. □

Lemmas 2 and 3 directly prove Theorem 2. Finally, we demonstrate that the final pass of columnsorts finishes the merge.

Theorem 3 *If every element in a list is within a distance of L of its sorted location, then a series of sorts of size $2L$, beginning at every Lth location, will suffice to complete the sort.*

Proof: Let there be N total records, so that we perform a total of $N/L - 2$ sorts of size $2L$. The proof proceeds by induction on the number of sorts performed. During step t we sort records $tL+1, tL+2, \ldots, tL+2L$, as in Figure 9. We show the following invariants at the beginning of step t by induction:

1. All the records $1, \ldots, tL$ are completely sorted
2. No record in $tL+1, \ldots, N$ is more than $2L$ before or L after its final position.

The invariants are clearly met at the beginning since (1) refers to the null set and we have assumed something stronger than (2).

Now we can demonstrate that each step in the range $t = 1, \ldots, N/L - 2$ preserves the invariants. To preserve invariant (1) we need to demonstrate that the sort moves records that belong in locations $tL+1, \ldots, tL+L$ to their final locations. By invariant (2), every record in this range must fall within the $2L$ records we are sorting: the record belonging at $tL+L$ can be off by at most L, placing it at $tL+2L$, which is within the sorting range. Thus, invariant (1) is preserved. Similarly, invariant (2) is preserved, since none of the records moved to the range $tL+L+1, \ldots, tL+2L$ belonged in the block $tL+1, \ldots, tL+L$ (by preservation of invariant (1)), so that they must be at most $2L$ before or L after their intended location.

Step $N/L-2$ by definition sorts the last $2L$ records, so that at its conclusion, the whole series is sorted. □

2.2 Analysis of the Algorithm

We first show by a clustering technique that the amount of storage space required for the data structures is small enough. To provide for the data structures, we need to strengthen the condition imposed by [AgV] so that $PB \leq \frac{1}{2}\lfloor M - M^{1/2+\alpha} \rfloor$, for $0 < \alpha < 1/2$.

Theorem 4 *For $0 < \alpha < 1/2$, the amount of primary memory space needed for the data structures of Greed Sort is $O(M^{1/2+\alpha})$.*

Proof: The number of runs that we must merge together in order to obtain optimal performance is $\sqrt{M/B}/2$. As we pointed out in Section 2, the candidate set requires

a total of $P\sqrt{M/B}$ key fields to be kept in primary memory to decide what blocks to read next. At first glance, it seems if $P = \Omega(M)$ and $B = 1$ that $\Omega(M^{3/2})$ storage space will be required in primary memory, which is clearly impossible. However, we can use a partial disk striping method throughout the course of the algorithm. Assume that P grows faster than M^α, where $0 < \alpha < 1/2$. We can cluster our P disks into clusters of $P' = M^\alpha$ clusters of $B' = BP/P'$ disks synchronized together. Each of the P' clusters acts like a logical disk with block size B'. Thus, the number of primary storage locations we need is at most

$$P'\sqrt{M/B'} \leq M^\alpha\sqrt{M/B'} = O(M^{1/2+\alpha}).$$

The expression for the number of I/Os remains the same, namely,

$$k\frac{N}{P'B'}\frac{\log\frac{N}{B'}}{\log\frac{M}{B'}} = O\left(\frac{N}{PB}\frac{\log\frac{N}{B}}{\log\frac{M}{B}}\right).$$

□

In order to demonstrate that Greed Sort has an optimal I/O bound, we need to analyze the I/O efficiency of the columnsort subroutine.

Theorem 5 *Columnsort sorts $N \leq P\sqrt{MB}$ records with $O(N/PB)$ parallel I/Os.*

Proof: First we show that columnsort produces a correctly sorted sequence when $N \leq P\sqrt{MB}$. We define the number of rows in the matrix to be $r = M$, so that each column can be sorted internally. We have

$$N \leq P\sqrt{MB} \leq PB\sqrt{M} \leq \frac{M^{3/2}}{2} < \frac{M^{3/2}}{\sqrt{2}},$$

making use of our assumption that $2PB \leq M$. Thus, the number of columns in our application of columnsort is $c = N/r \leq \sqrt{M/2}$, and the columnsort requirement that $r \geq 2(c-1)^2$ is met. This implies that columnsort works correctly.

Steps 1, 3, 5, 6, 7, and 8 can be done easily with $O(N/PB)$ I/Os. The transpose-like operation in Steps 2 and 4 can be done with $O(N/PB)$ I/Os by a variant of the $p \times q$ matrix transpose algorithm of [ViS], for $p = M$ and $q = N/M \leq P\sqrt{B/M}$, which we omit for brevity. The resulting number of I/Os is

$$O\left(\frac{N}{PB}\frac{\log\min\left\{M, P\sqrt{B/M}, P\sqrt{MB}/B\right\}}{\log(M/B)}\right) = O\left(\frac{N}{PB}\frac{\log\left(P\sqrt{M/B}\right)}{\log(M/B)}\right) = O\left(\frac{N}{PB}\right).$$

□

The greedy merge reads each record at most three times (once for updating *biggest* and *smallest* and up to twice for merging) and writes each record at most twice, taking full advantage of parallel block transfer. The columnsort routine is called $2N/k$ times, each time using $O(k/PB)$ I/Os, for a value of k that differs from pass to pass. Thus the total number of parallel I/Os is $O(N/PB)$:

Corollary 1 *Each merging step requires $O(N/PB)$ parallel I/Os.*

We are now ready for our main theorem.

Theorem 6 *Greed Sort sorts $N \geq M$ records with $O(\frac{N}{PB} \log \frac{N}{B} / \log \frac{M}{B})$ parallel I/Os deterministically, which is optimal, as long as $2PB \leq \lfloor M - M^{1/2+\alpha} \rfloor$.*

Proof: The correctness of the algorithm was shown in Theorem 1. The fact that the algorithm can proceed within the memory bounds was shown in Theorem 4. Finally, Corollary 1 shows that each pass takes $O(N/PB)$. Since we started with N/M runs and at each pass merged together $\sqrt{M/B}/2$ of them, the total I/O bound is

$$O\left(\frac{N}{PB}\left(1 + \log_{\sqrt{M/B}/2} \frac{N}{M}\right)\right) = O\left(\frac{N}{PB} \frac{\log \frac{N}{B}}{\log \frac{M}{B}}\right),$$

which is optimal, by the lower bound of [AgV]. □

3 Conclusions

Greed sort is an optimal deterministic algorithm for external sorting applications that is easy to implement in practice. The advantages of greed sort are:

- It exhibits optimal I/O performance under realistic high-performance storage systems with multiple disks.
- It is deterministic with a worst-case guarantee on performance. Its performance can be substantially better than that of the commonly-used technique of disk striping.
- It is simple and straightforward to implement.

This Greed Sort algorithm also has applications to the parallel hierarchical memory systems considered in [ViS], which will be explored in a forthcoming extension of this paper.

References

[AgV] A. Aggarwal & J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* (September 1988), also appears in *Proceedings of 14th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987.

[GiS] D. Gifford & A. Spector, "The TWA Reservation System," *Communications of the ACM* 27 (July 1984), 650–665.

[Lei] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers* C-34 (April 1985), 344–354.

[LiV] E. E. Lindstrom & J. S. Vitter, "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Transactions on Computers* C-34 (March 1985), 218–233.

[ViS] J. S. Vitter & E. A. M. Shriver, "Optimal Disk I/O with Parallel Block Transfer," *Symposium on the Theory of Computing* (May 1990).

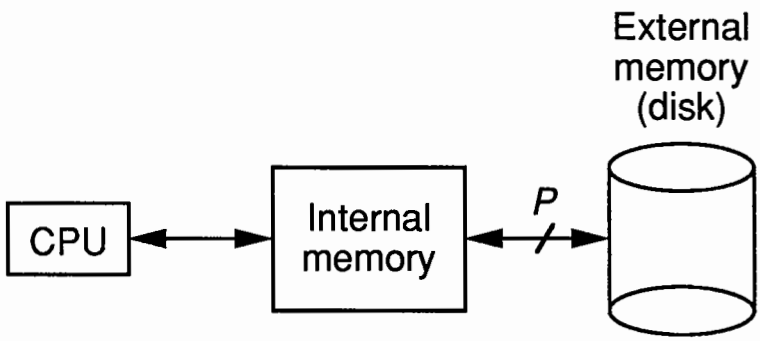

Figure 1: A simple P-parallel two-level memory model.

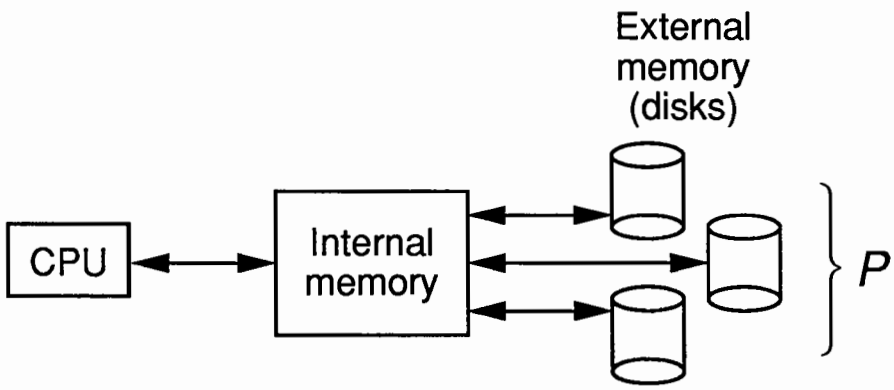

Figure 2: A more realistic P-parallel two-level memory model.

Disk 1	1	26	42	71	83	94
Disk 2	4	29	47	77	85	96
Disk 3	7	31	60	79	89	98
Disk 4	14	32	68	80	91	99

Figure 3: An example of a sorted list with $P = 4$, $B = 1$.

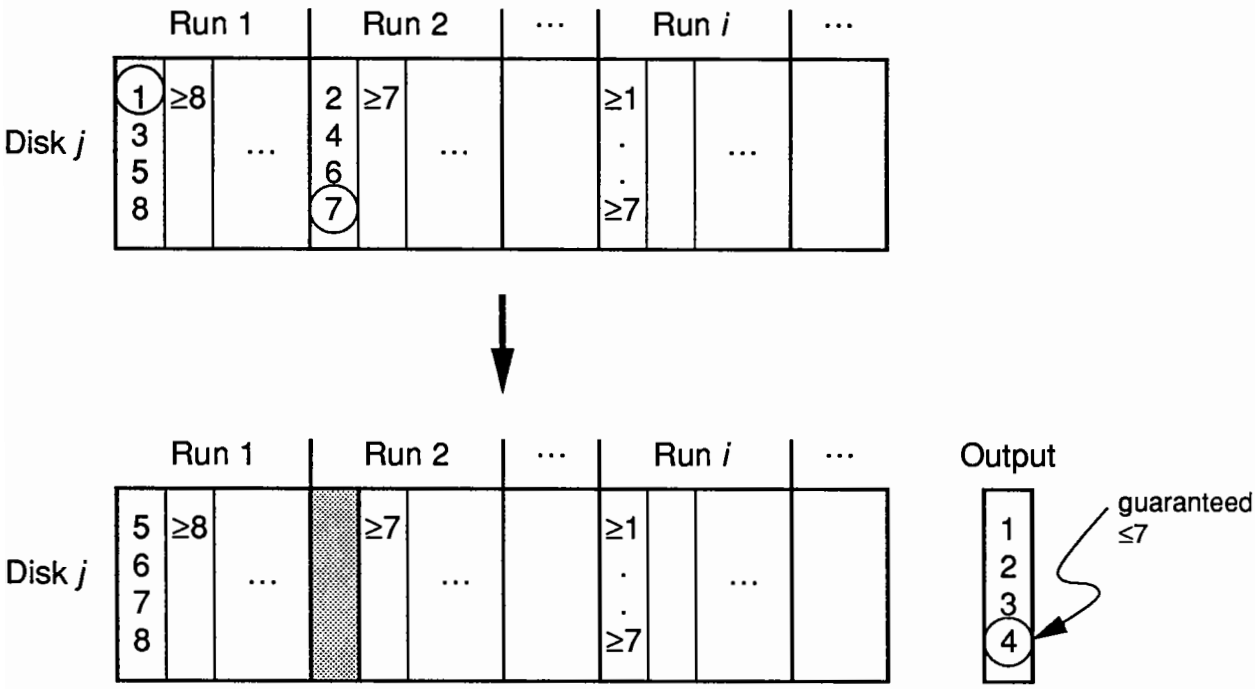

Figure 4: Assume that run 1 has the block with the smallest minimum and run 2 has the block with the smallest maximum. Then this figure depicts the state of the sort after the blocks have been processed.

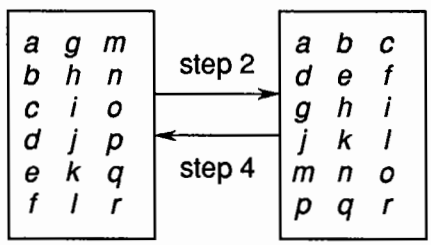

Figure 5: The transpose used by Steps 2 and 4 of columnsort.

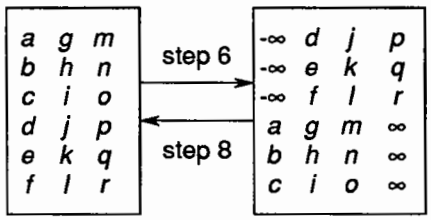

Figure 6: The shift operation used in Steps 6 and 8 of columnsort.

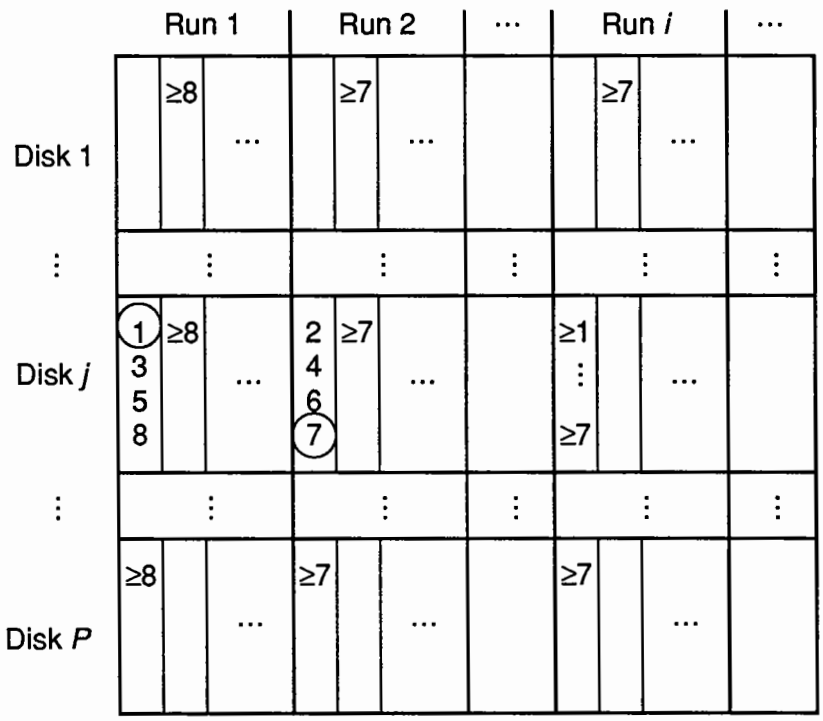

Figure 7: All the records output on disk j at this step are guaranteed to have keys ≤ 7 (in fact the largest key will be 4). Any disk prior to disk j has at most one block per run containing elements less than any key output on disk j, all of which will be written no more than $R-1$ blocks after this one. Any disk after disk j has already output all keys less than those written on disk j. Thus, a smaller key can follow a larger one by at most RPB locations in the approximately merged output list.

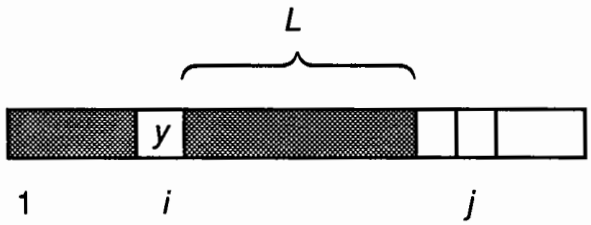

Figure 8: If an element y precedes its correct location by more than L, then there is not enough space (shaded region) for all those elements less than y without violating the L-regressive condition.

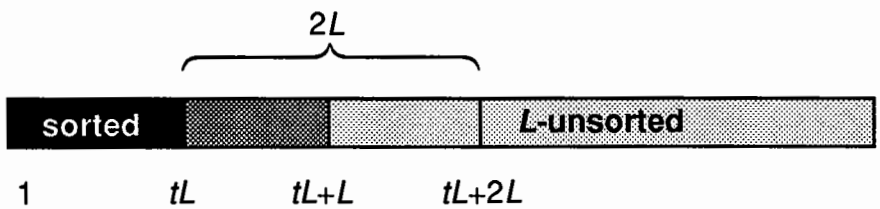

Figure 9: At each step in sorting $2L$ elements, the part before the beginning of the sort is completely sorted and the part starting halfway through the sort is L-unsorted. The remaining portion has no element more than L after nor $2L$ before where it should be.