13. External Sorting

Many important sorting applications involve processing very large files, much too large to fit into the primal y memory of any computer. Methods appropriate for such applications are called *external* methods, since they involve a large amount of processing external to the central processing unit (as opposed to the *internal* methods that we've been studying).

There are two major factors which make external algorithms quite different from those we've seen until now. First, the cost of accessing an item is orders of magnitude greater than any bookkeeping or calculating costs. Second, even with this higher cost, there are severe restrictions on access, depending on the external storage medium used: for example, items on a magnetic tape can be accessed only in a sequential manner.

The wide variety of external storage device types and costs make the development of external sorting methods very dependent on current technology. The methods can be complicated, and many parameters affect their performance: that a clever method might go unappreciated or unused because of a simple change in the technology is a definite possibility in external sorting. For this reason, we'll concentrate on general methods in this chapter rather than on developing specific implementations.

In short, for external sorting, the "systems" aspect of the problem is certainly as important as the "algorithms" aspect. Both areas must be carefully considered if an effective external sort is to be developed. The primary costs in external sorting are for input-output. A good exercise for someone planning to implement an efficient program to sort a very large file is first to implement an efficient program to copy a large file, then (if that was too easy) implement an efficient program to reverse the order of the elements in a large file. The systems problems that arise in trying tc solve these problems efficiently are similar to those that arise in external sorts. Permuting a large external file in any non-trivial way is about as difficult as sorting it, even though no key

comparisons, etc. are required. In external sorting, we are mainly concerned with limiting the number of times each piece of data is moved between the external storage medium and the primary memory, and being sure that such transfers are done as efficiently as allowed by the available hardware.

External sorting methods have been developed which are suitable for the punched cards and paper tape of the past, the magnetic tapes and disks of the present, and the bubble memories and videodisks of the future. The essential differences among the various devices are the relative size and speed of available storage and the types of data access restrictions. We'll concentrate on basic methods for sorting on magnetic tape and disk because these devices are likely to remain in widespread use and illustrate the two fundamentally different modes of access that characterize many external storage systems. Often, modern computer systems have a "storage hierarchy" of several progressively slower, cheaper, and larger memories. Many of the algorithms that we will consider can be adapted to run well in such an environment, but we'll deal exclusively with "two-level" memory hierarchies consisting of main memory and disk or tape.

Sort-Merge

Most external sorting methods use the following general strategy: make a first pass through the file to be sorted, breaking it up into blocks about the size of the internal memory, and *sort* these blocks. Then merge the sorted blocks together, by making several passes through the file, making successively larger sorted blocks until the whole file is sorted. The data is most often accessed in a sequential manner, which makes this method appropriate for most external devices. Algorithms for external sorting strive to reduce the number of passes through the file and to reduce the cost of a single pass to be as close to the cost of a copy as possible.

Since most of the cost of an external sorting method is for input-output, we can get a rough measure of the cost of a sort-merge by counting the number of times each word in the file is read or written, (the number of passes over all the data). For many applications, the methods that we consider will involve on the order of ten or less such passes. Note that this implies that we're interested in methods that can eliminate even a single pass. Also, the running time of the whole external sort can be easily estimated from the running time of something like the "reverse file copy" exercise suggested above.

Balanced *Multiway* Merging

To begin, we'll trace through the various steps of the simplest sort-merge procedure for an example file. Suppose that we have records with the keys A S O R T I N G A N D M E R G I N G E X A M P L E o n a n i n p u t t a p e ; t h e s e

are to be sorted and put onto an output tape. Using a "tape" simply means that we're restricted to read the records sequentially: the second record can't be read until the first has been, etc. Assume further that we have only enough room for three records in our computer memory but that we have plenty of tapes available.

The first step is to read in the file three records at a time, sort them to make three-record blocks, and output the sorted blocks. Thus, first we read in A S 0 and output the block A 0 S, next we read in R T I and output the block I R T, and so forth. Now, in order for these blocks to be merged together, they must be on different tapes. If we want to do a three-way merge, then we would use three tapes, ending up with the following configuration after the sorting pass:

Tape 1:	A 0 S	D M N	A E X
Tape 2:	I R T	E G R	L M P
Tape 3:	A G N	G I N	E

Now we're ready to merge the sorted blocks of size three together. We read the first record off each input tape (there's just enough room in the memory) and output the one with the smallest key. Then the next record from the same tape as the record just output is read in and, again, the record in memory with the smallest key is output. When the end of a three-word block in the input is encountered, then that tape is ignored until the blocks from the other two tapes have been processed, and nine records have been output. Then the process is repeated to merge the second three-word block on each tape into a nine-word block (which is output on a different tape, to get ready for the next merge). Continuing, we get three long blocks configured as follows:

Tape 4:	A	A	G	I	N	0	R	S	T
Tape 5:	D	E	G	G	I	M	N	N	R
Tape 6:	A	E	E	L	M	P	X		

Now one more three-way merge completes the sort. If we had a much longer file with many blocks of size 9 on each tape, then we would finish the second pass with blocks of size 27 on tapes 1, 2, and 3, then a third pass would produce blocks of size 81 on tapes 4, 5, and 6, and so forth. We need six tapes to sort an arbitrarily large file: three for the input and three for the

output of each three-way merge. Actually, we could get by with just four tapes: the output could be put on just one tape, then the blocks from that tape distributed to the three input tapes in between merging passes.

This method is called the balanced *multiway* merge: it is a reasonable algorithm for external sorting and a good starting point for the implementation of an external sort. The more sophisticated algorithms below could make the sort run perhaps 50% faster, but not much more. (On the other hand, when execution times are measured in hours, which is not uncommon in external sorting, even a small percentage decrease in running time can be helpful and 50% can be quite significant.)

Suppose that we have N words to be manipulated by the sort and an internal memory of size M. Then the "sort" pass produces about N/M sorted blocks. (This estimate assumes l-word records: for larger records, the number of sorted blocks is computed by multiplying further by the record size.) If we do P-way merges on each subsequent pass, then the number of subsequent passes is about $\log_P(N/M)$, since each pass reduces the number of sorted blocks by a factor of P.

Though small examples can help one understand the details of the algorithm, it is best to think in terms of very large files when working with external sorts. For example, the formula above says that using a 4-way merge to sort a 200-million-word file on a computer with 1 million words of memory should take a total of about five passes. A very rough estimate of the running time can be found by multiplying by five the running time for the reverse file copy implementation suggested above.

Replacement Selection

It turns out that the details of the implementation can be developed in an elegant and efficient way using priority queues. First, we'll see that priority queues provide a natural way to implement a multiway merge. More important, it turns out that we can use priority queues for the initial sorting pass in such a way that they can produce sorted blocks much longer than could fit into internal memory.

The basic operation needed to do P-way merging is to repeatedly output the smallest of the smallest elements not yet output from each of the P blocks to be merged. That element should be replaced with the next element from the block from which it came. The *replace* operation on a priority queue of size P is exactly what is needed. (Actually, the "indirect" verions of the priority queue routines, as described in Chapter 11, are more appropriate for this application.) Specifically, to do a P-way merge we begin by filling up a priority queue of size P with the smallest element from each of the P inputs using the *pqinsert* procedure from Chapter 11 (appropriately modified so that

the smallest element rather than the largest is at the top of the heap). Then, using the pqreplace procedure from Chapter 11 (modified in the same way) we output the smallest element and replace it in the priority queue with the next element from its block.

For example, the following table shows the result of merging A 0 S with I R T and A G N (the first merge from our example above):

1	2	3
A	I	A
A	I	0
G	I	0
I	N	0
N	R	0
0	R	
R	S	
S	T	
T		

The lines in the table represent the contents of a heap of size three used in the merging process. We begin with the first three keys in each block. (The "heap condition" is that the first key must be smaller than the second and third.) Then the first A is output and replaced with the 0 (the next key in its block). This violates the heap condition, so the 0 is exchanged with the other A. Then that A is output and replaced with the next key in its block, the G. This does not violate the heap condition, so no further change is necessary. Continuing in this way, we produce the sorted file (read down in the table to see the keys in the order in which they appear in the first heap position and are output). When a block is exhausted, a sentinel is put on the heap and considered to be larger than all the other keys. When the heap consists of all sentinels, the merge is completed. This way of using priority queues is sometimes called replacement *selection.*

Thus to do a P-way merge, we can use replacement selection on a priority queue of size P to find each element to be output in $\log P$ steps. This performance difference is not of particular practical relevance, since a brute-force implementation can find each elernent to output in P steps, and P is normally so small that this cost is dwarfed by the cost of actually outputting the element. The real importance of replacement selection is the way that it can be used in the first part of the scrt-merge process: to form the initial sorted blocks which provide the basis for the merging passes.

The idea is to pass the (unordered) input through a large priority queue, always writing out the smallest element on the priority queue as above, and always replacing it with the next element from the input, with one additional proviso: if the new element is smaller than the last one put out, then, since it could not possibly become part of the current sorted block, it should be marked as a member of the next block and treated as greater than all elements in the current block. When a marked element makes it to the top of the priority queue, the old block is ended and a new block started. Again, this is easily implemented with *pqinsert* and pqreplace from Chapter 11, again appropriately modified so that the smallest element is at the top of the heap, and with pqreplace changed to treat marked elements as always greater than unmarked elements.

Our example file clearly demonstrates the value of replacement selection. With an internal memory capable of holding only three records, we can produce sorted blocks of size 5, 4, 9, 6, and 1, as illustrated in the following table. Each step in the diagram below shows the next key to be input (boxed) and the contents of the heap just before that key is input. (As before, the order in which the keys occupy the first position in the heap is the order in which they are output.) Asterisks are used to indicate which keys in the heap belong to different blocks: an element marked the same way as the element at the root belongs to the current sorted block, others belong to the next sorted block. Always, the heap condition (first key less than the second and third) is maintained, with elements in the next sorted block considered to be greater than elements in the current sorted block.

R	A	S	0	E	A	M	D	M	A*	G*	E*
T	0	S	R	R	D	M	E	P	E*	G*	M*
I	R	S	T	G	E	M	R	L	G*	P*	M*
N	S	I'	T	I	G	M	R	E	L*	P*	M*
G	T	I'	N*	N	I	M	R		M*	P*	E
A	G*	I'	N*	G	MN	R		P*	E
N	I*	A	N*	E	N	G*	R		E
D	N*	A	N*	X	R	G*	E*
M	N*	A	D	A	X	G*	E*

For example, when pqreplace is called for M, it returns N for output (A and D are considered greater) and then sifts down M to make the heap A M D.

It can be shown that, if the keys are random, the runs produced using replacement selection are about twice the size of what could be produced using an internal method. The practical effect of this is to save one merging pass: rather than starting with sorted runs about the size of the internal memory and then taking a merging pass to produce runs about twice the size of the internal memory, we can start right off with runs about twice the size of the internal memory, by using replacement selection with a priority queue of size M. If there is some order in the keys, then the runs will be much, much longer. For example, if no key has more than M larger keys before it in the file, the file will be completely sorted by the replacement selection pass, and no merging will be necessary! This is the most important practical reason to use the method.

In summary, the replacement selection technique can be used for both the "sort" and the "merge" steps of a balanced multiway merge. To sort N l-word records using an internal memory of size M and $P + 1$ tapes, first use replacement selection with a priority queue of size M to produce initial runs of size about $2M$ (in a random situation) or longer (if the file is partially ordered) then use replacement selection with a priority queue of size P for about $\log_P(N/2M)$ (or fewer) merge passes.

Practical *Considerations*

To complete an implementation of the sorting method outlined above, it is necessary to implement the input-output functions which actually transfer data between the processor and the external devices. These functions are obviously the key to good performance for the external sort, and they just as obviously require careful consideration of some systems (as opposed to algorithm) issues. (Readers unfamiliar with computers at the "systems" level may wish to skim the next few paragraphs.)

A major goal in the implementation should be to overlap reading, writing, and computing as much as possible. Most large computer systems have independent processing units for controlling the large-scale input/output (I/O) devices which make this overlapping possible. The efficiency achievable by an external sorting method depends on the number of such devices available.

For each file being read or written, there is a standard systems programming technique called double-buffering which can be used to maximize the overlap of I/O with computing. The idea is to maintain two "buffers," one for use by the main processor, one for use by the I/O device (or the processor which controls the I/O device). For input, the processor uses one buffer while the input device is filling the other. When the processor has finished using its buffer, it waits until the input device has filled its buffer, then the buffers switch roles: the processor uses the new data in the just-filled buffer while

the input device refills the buffer with the data already used by the processor. The same technique works for output, with the roles of the processor and the device reversed. Usually the I/O time is far greater than the processing time and so the effect of double-buffering is to overlap the computation time entirely; thus the buffers should be as large as possible.

A difficulty with double-buffering is that it really uses only about half the available memory space. This can lead to inefficiency if a large number of buffers are involved, as is the case in P-way merging when P is not small. This problem can be dealt with using a technique called forecasting, which requires the use of only one extra buffer (not P) during the merging process. Forecasting works as follows. Certainly the best way to overlap input with computation during the replacement selection process is to overlap the input of the buffer that needs to be filled next with the processing part of the algorithm. But it is easy to determine which buffer this is: the next input buffer to be emptied is the one whose *last* item is smallest. For example, when merging A 0 S with I R T and A G N we know that the third buffer will be the first to empty, then the first. A simple way to overlap processing with input for multiway merging is therefore to keep one extra buffer which is filled by the input device according to this rule. When the processor encounters an empty buffer, it waits until the input buffer is filled (if it hasn't been filled already), then switches to begin using that buffer and directs the input device to begin filling the buffer just emptied according to the forecasting rule.

The most important decision to be made in the implementation of the multiway merge is the choice of the value of P, the "order" of the merge. For tape sorting, when only sequential access is allowed, this choice is easy: P must be chosen to be one less than the number of tape units available: the multiway merge uses P input tapes and one output tape. Obviously, there should be at least two input tapes, so it doesn't make sense to try to do tape sorting with less than three tapes.

For disk sorting, when access to arbitrary positions is allowed but is somewhat more expensive than sequential access, it is also reasonable to choose P to be one less than the number of disks available, to avoid the higher cost of non-sequential access that would be involved, for example, if two different input files were on the same disk. Another alternative commonly used is to pick P large enough so that the sort will be complete in two merging phases: it is usually unreasonable to try to do the sort in one pass, but a two-pass sort can often be done with a reasonably small P. Since replacement selection produces about $N/2M$ runs and each merging pass divides the number of runs by P, this means P should be chosen to be the smallest integer with $P^2 > N/2M$. For our example of sorting a 200-million-word file on a computer with a l-million-word memory, this implies that $P = 11$ would be a safe choice to ensure a two-pass sort. (The right value of P could be computed

exactly after the sort phase is completed) The best choice between these two alternatives of the lowest reasonable value of P and the highest reasonable value of P is obviously very dependent on many systems parameters: both alternatives (and some in between) should be considered.

Polyphase Merging

One problem with balanced multiway merging for tape sorting is that it requires either an excessive number of tape units or excessive copying. For P-way merging either we must use $2P$ tapes (P for input and P for output) or we must copy almost all of the file from a single output tape to P input tapes between merging passes, which effectively doubles the number of passes to be about $2\log_P(N/2M)$. Several clever tape-sorting algorithms have been invented which eliminate virtually all of this copying by changing the way in which the small sorted blocks are merged together. The most prominent of these methods is called *polyphase merging*.

The basic idea behind polyphase merging is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (leaving one empty) and then to apply a "merge until empty" strategy, at which point one of the output tapes and the input, tape switch roles.

For example, suppose that we have just three tapes, and we start out with the following initial configuration of sorted blocks on the tapes. (This comes from applying replacement selection to our example file with an internal memory that can only hold two records.:

Tape 1: A 0 R S T I N A G N D E M R G I N
Tape 2: E G X A M P E L
Tape 3:

After three 2-way merges from tapes 1 and 2 to tape 3, the second tape becomes empty and we are left with the configuration:

Tape 1: D E M R G I N
Tape 2:
Tape 3: A E G O R S T X A I M N P A E G L N

Then, after two 2-way merges from tapes 1 and 3 to tape 2, the first tape becomes empty, leaving:

Tape 1:
Tape 2: A D E E G M O R R S T X A G I I M N N P
Tape 3: A E G L N

The sort is completed in two more steps. First, a two-way merge from tapes 2 and 3 to tape 1 leaves one file on tape 2, one file on tape 1. Then a two-way merge from tapes 1 and 2 to tape 3 leaves the entire sorted file on tape 3.

This "merge until empty" strategy can be extended to work for an arbitrary number of tapes. For example, if we have four tape units T1, T2, T3, and T4 and we start out with T1 being the output tape, T2 having 13 initial runs, T3 having 11 initial runs, and T4 having 7 initial runs, then after running a 3-way "merge until empty," we have T4 empty, T1 with 7 (long) runs, T2 with 6 runs, and T3 with 4 runs. At this point, we can rewind T1 and make it an input tape, and rewind T4 and make it an output tape. Continuing in this way, we eventually get the whole sorted file onto T1:

T1	T2	T3	T4
0	13	11	7
7	*6*	*4*	*0*
3	*2*	*0*	*4*
1	*0*	*2*	*2*
0	1	1	1
1	0	0	0

The merge is broken up into many *phases* which don't involve all the data, but no direct copying is involved.

The main difficulty in implementing a polyphase merge is to determine how to distribute the initial runs. It is not difficult to see how to build the table above by working backwards: take the largest number on each line, make it zero, and add it to each of the other numbers to get the previous line. This corresponds to defining the highest-order merge for the previous line which could give the present line. This technique works for any number of tapes (at least three): the numbers which arise are "generalized Fibonacci numbers" which have many interesting properties. Of course, the number of initial runs may not be known in advance, and it probably won't be exactly a generalized Fibonacci number. Thus a number of "dummy" runs must be added to make the number of initial runs exactly what is needed for the table.

The analysis of polyphase merging is complicated, interesting, and yields surprising results. For example, it turns out that the very best method for distributing dummy runs among the tapes involves using extra phases and more dummy runs than would seem to be needed. The reason for this is that some runs are used in merges much more often than others.

There are many other factors to be taken into consideration in implementing a most efficient tape-sorting method. For example, a major factor which we have not considered at all is the time that it takes to rewind a tape. This subject has been studied extensively, and many fascinating methods have been defined. However, as mentioned above, the savings achievable over the simple multiway balanced merge are quite limited. Even polyphase merging is only better than balanced merging for small P, and then not substantially. For $P > 8$, balanced merging is likely to run faster than polyphase, and for smaller P the effect of polyphase is basically to save two tapes (a balanced merge with two extra tapes will run faster).

An *Easier Way*

Many modern computer systems provide a large *virtual memory* capability which should not be overlooked in implementing a method for sorting very large files. In a good virtual memory system, the programmer has the ability to address a very large amount of data, leaving to the system the responsibility of making sure that addressed data is transferred from external to internal storage when needed. This strategy relies on the fact that many programs have a relatively small "locality of reference" : each reference to memory is likely to be to an area of memory that is relatively close to other recently referenced areas. This implies that transfers from external to internal storage are needed infrequently. An internal sorting method with a small locality of reference can work very well on a virtual memory system. (For example, Quicksort has two "localities" : most references are near one of the two partitioning pointers.) But check with your systems programmer before trying it on a very large file: a method such as radix sorting, which has no locality of reference whatsoever, would be disastrous on a virtual memory system, and even Quicksort could cause problems, depending on how well the available virtual memory system is implemented. On the other hand, the strategy of using a simple internal sorting method for sorting disk files deserves serious consideration in a good virtual memory environment.

Exercises

1. Describe how you would do external *selection:* find the kth largest in a file of N elements, where N is much too large for the file to fit in main memory.

2. Implement the replacement selection algorithm, then use it to test the claim that the runs produced are about twice the internal memory size.

3. What is the *worst* that can happen when replacement selection is used to produce initial runs in a file of N records, using a priority queue of size M, with $M < N$.

4. How would you sort the contents of a disk if no other storage (except main memory) were available for use?

5. How would you sort the contents of a disk if only one tape (and main memory) were available for use?

6. Compare the 4-tape and 6-tape multiway balanced merge to polyphase merge with the same number of tapes, for 31 initial runs.

7. How many phases does 5-tape polyphase merge use when started up with four tapes containing 26,15,22,28 runs?

8. Suppose the 31 initial runs in a 4-tape polyphase merge are each one record long (distributed 0, 13, 11, 7 initially). How many records are there in each of the files involved in the last three-way merge?

9. How should small files be handled in a Quicksort implementation to be run on a very large file within a virtual memory environment?

10. How would you organize an external priority queue? (Specifically, design a way to support the insert and remove operations of Chapter 11, when the number of elements in the priority queue could grow to be much to large for the queue to fit in main memory.)