11 EXTERNAL SORTING

Good order is the foundation of all things.

—Edmund Burke

Sorting a collection of records on some (search) key is a very useful operation. The key can be a single attribute or an ordered list of attributes, of course. Sorting is required in a variety of situations, including the following important ones:

- Users may want answers in some order; for example, by increasing age (Section 5.2).
- Sorting records is the first step in *bulk loading* a tree index (Section 9.8.2).
- Sorting is useful for eliminating *duplicate* copies in a collection of records (Chapter 12).
- A widely used algorithm for performing a very important relational algebra operation, called *join*, requires a sorting step (Section 12.5.2).

Although main memory sizes are increasing, as usage of database systems increases, increasingly larger datasets are becoming common as well. When the data to be sorted is too large to fit into available main memory, we need to use an *external sorting* algorithm. Such algorithms seek to minimize the cost of disk accesses.

We introduce the idea of external sorting by considering a very simple algorithm in Section 11.1; using repeated passes over the data, even very large datasets can be sorted with a small amount of memory. This algorithm is generalized to develop a realistic external sorting algorithm in Section 11.2. Three important refinements are discussed. The first, discussed in Section 11.2.1, enables us to reduce the number of passes. The next two refinements, covered in Section 11.3, require us to consider a more detailed model of I/O costs than the number of page I/Os. Section 11.3.1 discusses the effect of *blocked* I/O, that is, reading and writing several pages at a time; and Section 11.3.2 considers how to use a technique called double buffering to minimize the time spent waiting for an I/O operation to complete. Section 11.4 discusses the use of B+ trees for sorting.

With the exception of Section 11.3, we consider only I/O costs, which we approximate by counting the number of pages read or written, as per the cost model discussed in

Sorting in commercial RDBMSs: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all use external merge sort. Sybase ASE uses a memory partition called the procedure cache for sorting. This is a main memory region that is used for compilation and execution, as well as for caching the plans for recently executed stored procedures; it is not part of the buffer pool. IBM, Informix, and Oracle also use a separate area of main memory to do sorting. In contrast, Microsoft and Sybase IQ use buffer pool frames for sorting. None of these systems uses the optimization that produces runs larger than available memory, in part because it is difficult to implement it efficiently in the presence of variable length records. In all systems, the I/O is asynchronous and uses prefetching. Microsoft and Sybase ASE use merge sort as the in-memory sorting algorithm; IBM and Sybase IQ use radix sorting. Oracle uses insertion sort for in-memory sorting.

Chapter 8. Our goal is to use a simple cost model to convey the main ideas, rather than to provide a detailed analysis.

11.1 A SIMPLE TWO-WAY MERGE SORT

We begin by presenting a simple algorithm to illustrate the idea behind external sorting. This algorithm utilizes only three pages of main memory, and it is presented only for pedagogical purposes. In practice, many more pages of memory will be available, and we want our sorting algorithm to use the additional memory effectively; such an algorithm is presented in Section 11.2. When sorting a file, several sorted subfiles are typically generated in intermediate steps. In this chapter, we will refer to each sorted subfile as a **run**.

Even if the entire file does not fit into the available main memory, we can sort it by breaking it into smaller subfiles, sorting these subfiles, and then merging them using a minimal amount of main memory at any given time. In the first pass the pages in the file are read in one at a time. After a page is read in, the records on it are sorted and the sorted page (a sorted run one page long) is written out. Quicksort or any other in-memory sorting technique can be used to sort the records on a page. In subsequent passes pairs of runs from the output of the previous pass are read in and *merged* to produce runs that are twice as long. This algorithm is shown in Figure 11.1.

If the number of pages in the input file is 2^k, for some k, then:

Pass 0 produces 2^k sorted runs of one page each,
Pass 1 produces 2^{k-1} sorted runs of two pages each,
Pass 2 produces 2^{k-2} sorted runs of four pages each,

```
proc 2-way_extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are one page long: Pass 0
Read each page into memory, sort it, write it out.
// Merge pairs of runs to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1, 2, ...
    While there are runs to be merged from previous pass:
        Choose next two runs (from previous pass).
        Read each run into an input buffer; page at a time.
        Merge the runs and write to the output buffer;
        force output buffer to disk one page at a time.

endproc
```

Figure 11.1 Two-Way Merge Sort

and so on, until
Pass k produces one sorted run of 2^k pages.

In each pass we read every page in the file, process it, and write it out. Thus we have two disk I/Os per page, per pass. The number of passes is $\lceil log_2 N \rceil + 1$, where N is the number of pages in the file. The overall cost is $2N(\lceil log_2 N \rceil + 1)$ I/Os.

The algorithm is illustrated on an example input file containing seven pages in Figure 11.2. The sort takes four passes, and in each pass we read and write seven pages, for a total of 56 I/Os. This result agrees with the preceding analysis because $2 * 7(\lceil log_2 7 \rceil + 1) = 56$. The dark pages in the figure illustrate what would happen on a file of eight pages; the number of passes remains at four ($\lceil log_2 8 \rceil + 1 = 4$), but we read and write an additional page in each pass for a total of 64 I/Os. (Try to work out what would happen on a file with, say, five pages.)

This algorithm requires just three buffer pages in main memory, as Figure 11.3 illustrates. This observation raises an important point: Even if we have more buffer space available, this simple algorithm does not utilize it effectively. The external merge sort algorithm that we discuss next addresses this problem.

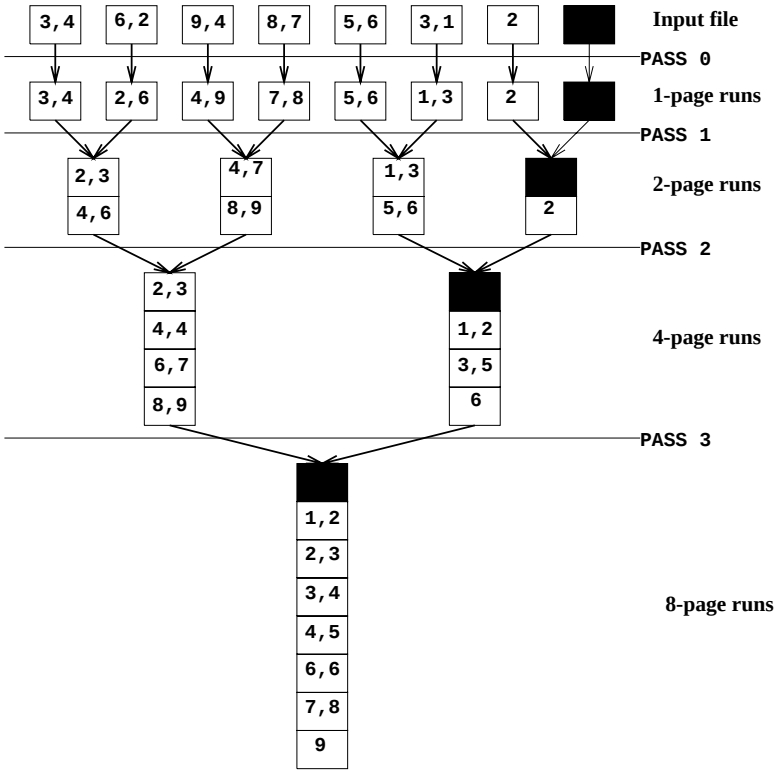

Figure 11.2 Two-Way Merge Sort of a Seven-Page File

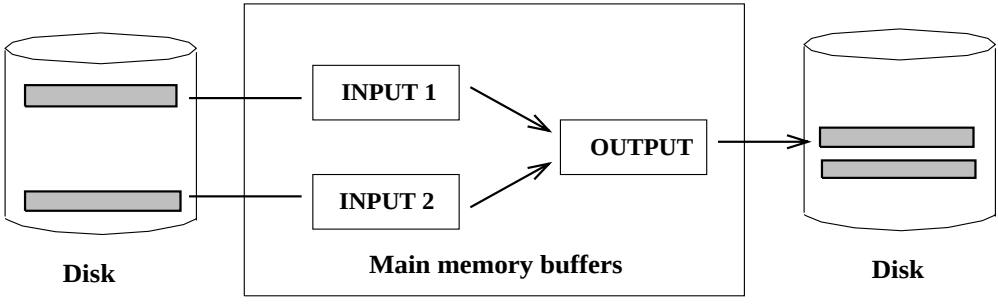

Figure 11.3 Two-Way Merge Sort with Three Buffer Pages

11.2 EXTERNAL MERGE SORT

Suppose that B buffer pages are available in memory and that we need to sort a large file with N pages. How can we improve upon the two-way merge sort presented in the previous section? The intuition behind the generalized algorithm that we now present is to retain the basic structure of making multiple passes while trying to minimize the number of passes. There are two important modifications to the two-way merge sort algorithm:

1. In Pass 0, read in B pages at a time and sort internally to produce $\lceil N/B \rceil$ runs of B pages each (except for the last run, which may contain fewer pages). This modification is illustrated in Figure 11.4, using the input file from Figure 11.2 and a buffer pool with four pages.

2. In passes i=1,2, ... , use $B-1$ buffer pages for input, and use the remaining page for output; thus you do a $(B-1)$-way merge in each pass. The utilization of buffer pages in the merging passes is illustrated in Figure 11.5.

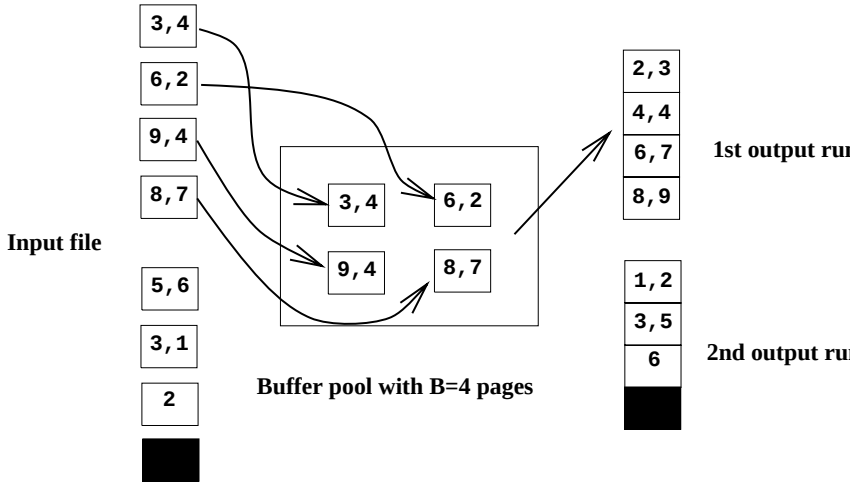

Figure 11.4 External Merge Sort with B Buffer Pages: Pass 0

The first refinement reduces the number of runs produced by Pass 0 to $N1 = \lceil N/B \rceil$, versus N for the two-way merge.[1] The second refinement is even more important. By doing a $(B-1)$-way merge, the number of passes is reduced dramatically—including the initial pass, it becomes $\lceil log_{B-1} N1 \rceil + 1$ versus $\lceil log_2 N \rceil + 1$ for the two-way merge algorithm presented earlier. Because B is typically quite large, the savings can be substantial. The external merge sort algorithm is shown is Figure 11.6.

[1]Note that the technique used for sorting data in buffer pages is orthogonal to external sorting. You could use, say, Quicksort for sorting data in buffer pages.

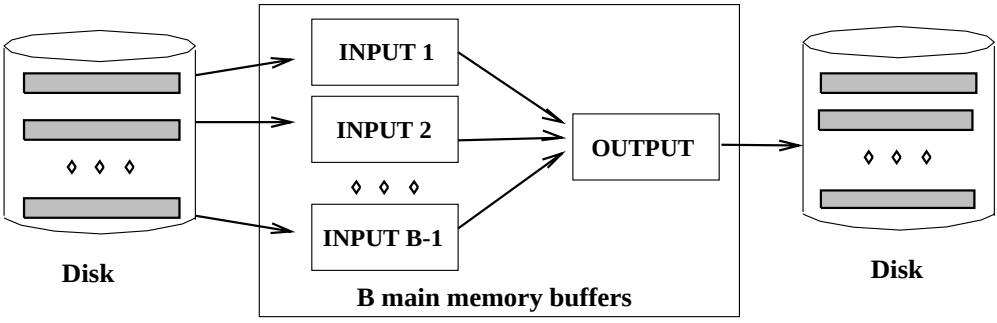

Figure 11.5 External Merge Sort with B Buffer Pages: Pass $i > 0$

proc *extsort* (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are B pages long: Pass 0
Read B pages into memory, sort them, write out a run.
// Merge $B - 1$ runs at a time to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1, 2, ...
    While there are runs to be merged from previous pass:
        Choose next $B - 1$ runs (from previous pass).
        Read each run into an input buffer; page at a time.
        Merge the runs and write to the output buffer;
        force output buffer to disk one page at a time.

endproc

Figure 11.6 External Merge Sort

As an example, suppose that we have five buffer pages available, and want to sort a file with 108 pages.

Pass 0 produces $\lceil 108/5 \rceil = 22$ sorted runs of five pages each, except for the last run, which is only three pages long.
Pass 1 does a four-way merge to produce $\lceil 22/4 \rceil$ = six sorted runs of 20 pages each, except for the last run, which is only eight pages long.
Pass 2 produces $\lceil 6/4 \rceil$ = two sorted runs; one with 80 pages and one with 28 pages.
Pass 3 merges the two runs produced in Pass 2 to produce the sorted file.

In each pass we read and write 108 pages; thus the total cost is $2 * 108 * 4 = 864$ I/Os. Applying our formula, we have $N1 = \lceil 108/5 \rceil = 22$ and cost $= 2*N*(\lceil log_{B-1} N1 \rceil + 1)$ $= 2 * 108 * (\lceil log_4 22 \rceil + 1) = 864$, as expected.

To emphasize the potential gains in using all available buffers, in Figure 11.7 we show the number of passes, computed using our formula, for several values of N and B. To obtain the cost, the number of passes should be multiplied by $2N$. In practice, one would expect to have more than 257 buffers, but this table illustrates the importance of a high fan-in during merging.

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Figure 11.7 Number of Passes of External Merge Sort

Of course, the CPU cost of a multiway merge can be greater than that for a two-way merge, but in general the I/O costs tend to dominate. In doing a $(B-1)$-way merge, we have to repeatedly pick the 'lowest' record in the $B-1$ runs being merged and write it to the output buffer. This operation can be implemented simply by examining the first (remaining) element in each of the $B-1$ input buffers. In practice, for large values of B, more sophisticated techniques can be used, although we will not discuss them here. Further, as we will see shortly, there are other ways to utilize buffer pages in order to reduce I/O costs; these techniques involve allocating additional pages to each input (and output) run, thereby making the number of runs merged in each pass considerably smaller than the number of buffer pages B.

11.2.1 Minimizing the Number of Runs *

In Pass 0 we read in B pages at a time and sort them internally to produce $\lceil N/B \rceil$ runs of B pages each (except for the last run, which may contain fewer pages). With a more aggressive implementation, called **replacement sort**, we can write out runs of approximately $2 * B$ internally sorted pages on average.

This improvement is achieved as follows. We begin by reading in pages of the file of tuples to be sorted, say R, until the buffer is full, reserving (say) one page for use as an input buffer and (say) one page for use as an output buffer. We will refer to the $B - 2$ pages of R tuples that are not in the input or output buffer as the *current set*. Suppose that the file is to be sorted in ascending order on some search key k. Tuples are appended to the output in ascending order by k value.

The idea is to repeatedly pick the tuple in the current set with the smallest k value that is still greater than the largest k value in the output buffer and append it to the output buffer. For the output buffer to remain sorted, the chosen tuple must satisfy the condition that its k value be greater than or equal to the largest k value currently in the output buffer; of all tuples in the current set that satisfy this condition, we pick the one with the smallest k value, and append it to the output buffer. Moving this tuple to the output buffer creates some space in the current set, which we use to add the next input tuple to the current set. (We assume for simplicity that all tuples are the same size.) This process is illustrated in Figure 11.8. The tuple in the current set that is going to be appended to the output next is highlighted, as is the most recently appended output tuple.

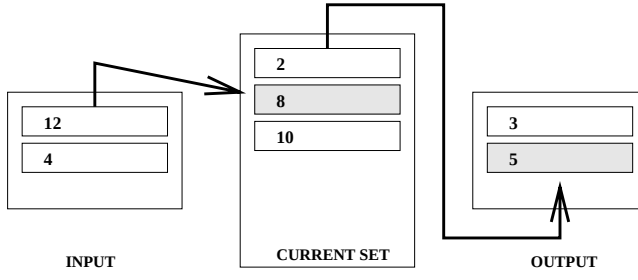

Figure 11.8 Generating Longer Runs

When all tuples in the input buffer have been consumed in this manner, the next page of the file is read in. Of course, the output buffer is written out when it is full, thereby extending the current run (which is gradually built up on disk).

The important question is this: When do we have to terminate the current run and start a new run? As long as some tuple t in the current set has a bigger k value than the most recently appended output tuple, we can append t to the output buffer, and

the current run can be extended.[2] In Figure 11.8, although a tuple ($k = 2$) in the current set has a smaller k value than the largest output tuple ($k = 5$), the current run can be extended because the current set also has a tuple ($k = 8$) that is larger than the largest output tuple.

When every tuple in the current set is smaller than the largest tuple in the output buffer, the output buffer is written out and becomes the last page in the current run. We then start a new run and continue the cycle of writing tuples from the input buffer to the current set to the output buffer. It is known that this algorithm produces runs that are about $2 * B$ pages long, on average.

This refinement has not been implemented in commercial database systems because managing the main memory available for sorting becomes difficult with replacement sort, especially in the presence of variable length records. Recent work on this issue, however, shows promise and it could lead to the use of replacement sort in commercial systems.

11.3 MINIMIZING I/O COST VERSUS NUMBER OF I/OS

We have thus far used the number of page I/Os as a cost metric. This metric is only an approximation to true I/O costs because it ignores the effect of *blocked* I/O—issuing a single request to read (or write) several consecutive pages can be much cheaper than reading (or writing) the same number of pages through independent I/O requests, as discussed in Chapter 8. This difference turns out to have some very important consequences for our external sorting algorithm.

Further, the time taken to perform I/O is only part of the time taken by the algorithm; we must consider CPU costs as well. Even if the time taken to do I/O accounts for most of the total time, the time taken for processing records is nontrivial and is definitely worth reducing. In particular, we can use a technique called *double buffering* to keep the CPU busy while an I/O operation is in progress.

In this section we consider how the external sorting algorithm can be refined using blocked I/O and double buffering. The motivation for these optimizations requires us to look beyond the number of I/Os as a cost metric. These optimizations can also be applied to other I/O intensive operations such as joins, which we will study in Chapter 12.

[2]If B is large, the CPU cost of finding such a tuple t can be significant unless appropriate in-memory data structures are used to organize the tuples in the buffer pool. We will not discuss this issue further.

11.3.1 Blocked I/O

If the number of page I/Os is taken to be the cost metric, the goal is clearly to minimize the number of passes in the sorting algorithm because each page in the file is read and written in each pass. It therefore makes sense to maximize the fan-in during merging by allocating just one buffer pool page per run (which is to be merged) and one buffer page for the output of the merge. Thus we can merge $B - 1$ runs, where B is the number of pages in the buffer pool. If we take into account the effect of blocked access, which reduces the average cost to read or write *a single page*, we are led to consider whether it might be better to read and write in units of more than one page.

Suppose that we decide to read and write in units, which we call **buffer blocks**, of b pages. We must now set aside one buffer block per input run and one buffer block for the output of the merge, which means that we can merge at most $\lfloor \frac{B-b}{b} \rfloor$ runs in each pass. For example, if we have 10 buffer pages, we can either merge nine runs at a time with one-page input and output buffer blocks, or we can merge four runs at a time with two-page input and output buffer blocks. If we choose larger buffer blocks, however, the number of passes increases, while we continue to read and write every page in the file in each pass! In the example each merging pass reduces the number of runs by a factor of 4, rather than a factor of 9. Therefore, the number of page I/Os increases. This is the price we pay for decreasing the per-page I/O cost and is a trade-off that we must take into account when designing an external sorting algorithm.

In practice, however, current main memory sizes are large enough that all but the largest files can be sorted in just two passes, even using blocked I/O. Suppose that we have B buffer pages and choose to use a blocking factor of b pages. That is, we read and write b pages at a time, and our input and output buffer blocks are all b pages long. The first pass produces about $N2 = \lceil N/2B \rceil$ sorted runs, each of length $2B$ pages, if we use the optimization described in Section 11.2.1, and about $N1 = \lceil N/B \rceil$ sorted runs, each of length B pages, otherwise. For the purposes of this section, we will assume that the optimization is used.

In subsequent passes we can merge $F = \lfloor B/b \rfloor - 1$ runs at a time. The number of passes is therefore $1 + \lceil log_F N2 \rceil$, and in each pass we read and write all pages in the file. Figure 11.9 shows the number of passes needed to sort files of various sizes N, given B buffer pages, using a blocking factor b of 32 pages. It is quite reasonable to expect 5,000 pages to be available for sorting purposes; with 4 KB pages, 5,000 pages is only 20 MB. (With 50,000 buffer pages, we can do 1,561-way merges, with 10,000 buffer pages, we can do 311-way merges, with 5,000 buffer pages, we can do 155-way merges, and with 1,000 buffer pages, we can do 30-way merges.)

To compute the I/O cost, we need to calculate the number of 32-page blocks read or written and multiply this number by the cost of doing a 32-page block I/O. To find the

N	B=1,000	B=5,000	B=10,000	B=50,000
100	1	1	1	1
1,000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Figure 11.9 Number of Passes of External Merge Sort with Block Size $b = 32$

number of block I/Os, we can find the total number of page I/Os (number of passes multiplied by the number of pages in the file) and divide by the block size, 32. The cost of a 32-page block I/O is the seek time and rotational delay for the first page, plus transfer time for all 32 pages, as discussed in Chapter 8. The reader is invited to calculate the total I/O cost of sorting files of the sizes mentioned in Figure 11.9 with 5,000 buffer pages, for different block sizes (say, $b = 1$, 32, and 64) to get a feel for the benefits of using blocked I/O.

11.3.2 Double Buffering

Consider what happens in the external sorting algorithm when all the tuples in an input block have been consumed: An I/O request is issued for the next block of tuples in the corresponding input run, and the execution is forced to suspend until the I/O is complete. That is, for the duration of the time taken for reading in one block, the CPU remains idle (assuming that no other jobs are running). The overall time taken by an algorithm can be increased considerably because the CPU is repeatedly forced to wait for an I/O operation to complete. This effect becomes more and more important as CPU speeds increase relative to I/O speeds, which is a long-standing trend in relative speeds. It is therefore desirable to keep the CPU busy while an I/O request is being carried out, that is, to overlap CPU and I/O processing. Current hardware supports such overlapped computation, and it is therefore desirable to design algorithms to take advantage of this capability.

In the context of external sorting, we can achieve this overlap by allocating extra pages to each input buffer. Suppose that a block size of $b = 32$ is chosen. The idea is to allocate an additional 32-page block to every input (and the output) buffer. Now, when all the tuples in a 32-page block have been consumed, the CPU can process the next 32 pages of the run by switching to the second, 'double,' block for this run. Meanwhile, an I/O request is issued to fill the empty block. Thus, assuming that the

time to consume a block is greater than the time to read in a block, the CPU is never idle! On the other hand, the number of pages allocated to a buffer is doubled (for a given block size, which means the total I/O cost stays the same). This technique is called **double buffering**, and it can considerably reduce the total time taken to sort a file. The use of buffer pages is illustrated in Figure 11.10.

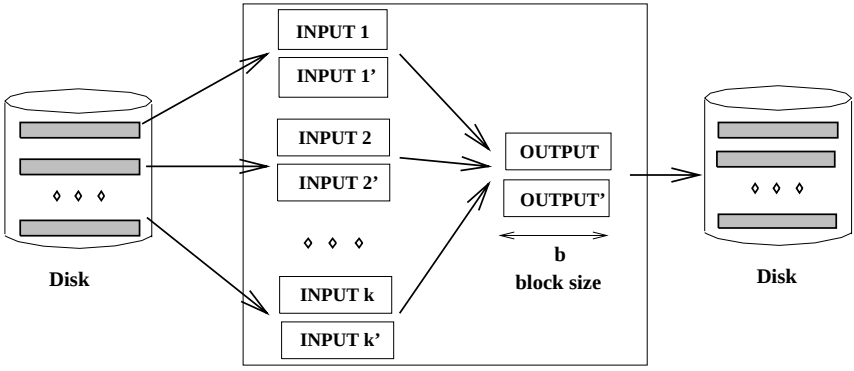

Figure 11.10 Double Buffering

Note that although double buffering can considerably reduce the response time for a given query, it may not have a significant impact on throughput, because the CPU can be kept busy by working on other queries while waiting for one query's I/O operation to complete.

11.4 USING B+ TREES FOR SORTING

Suppose that we have a B+ tree index on the (search) key to be used for sorting a file of records. Instead of using an external sorting algorithm, we could use the B+ tree index to retrieve the records in search key order by traversing the sequence set (i.e., the sequence of leaf pages). Whether this is a good strategy depends on the nature of the index.

11.4.1 Clustered Index

If the B+ tree index is clustered, then the traversal of the sequence set is very efficient. The search key order corresponds to the order in which the data records are stored, and for each page of data records that we retrieve, we can read all the records on it in sequence. This correspondence between search key ordering and data record ordering is illustrated in Figure 11.11, with the assumption that data entries are $\langle key, rid \rangle$ pairs (i.e., Alternative (2) is used for data entries).

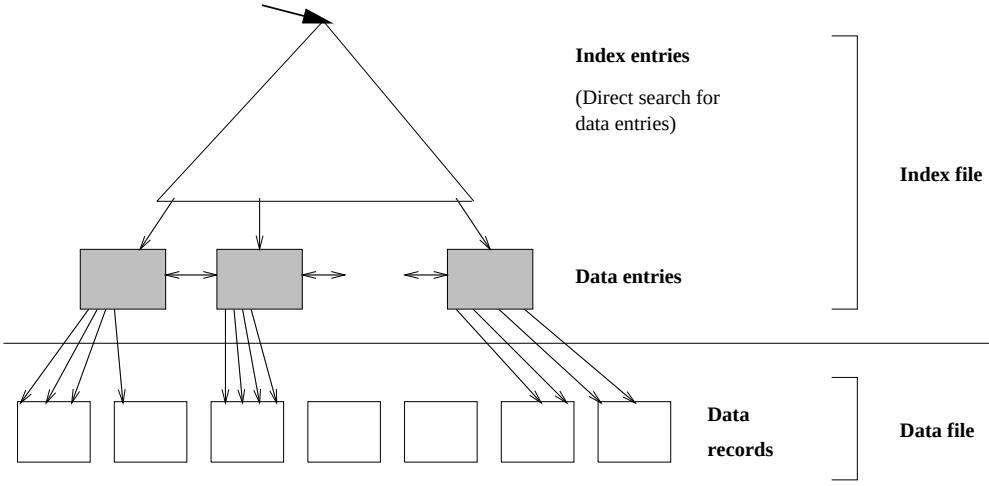

Figure 11.11 Clustered B+ Tree for Sorting

The cost of using the clustered B+ tree index to retrieve the data records in search key order is the cost to traverse the tree from root to the left-most leaf (which is usually less than four I/Os) plus the cost of retrieving the pages in the sequence set, plus the cost of retrieving the (say N) pages containing the data records. Note that no data page is retrieved twice, thanks to the ordering of data entries being the same as the ordering of data records. The number of pages in the sequence set is likely to be much smaller than the number of data pages because data entries are likely to be smaller than typical data records. Thus, the strategy of using a clustered B+ tree index to retrieve the records in sorted order is a good one and should be used whenever such an index is available.

What if Alternative (1) is used for data entries? Then the leaf pages would contain the actual data records, and retrieving the pages in the sequence set (a total of N pages) would be the only cost. (Note that the space utilization is about 67 percent in a B+ tree; thus, the number of leaf pages is greater than the number of pages needed to hold the data records in a sorted file, where, in principle, 100 percent space utilization can be achieved.) In this case the choice of the B+ tree for sorting is excellent!

11.4.2 Unclustered Index

What if the B+ tree index on the key to be used for sorting is unclustered? This is illustrated in Figure 11.12, with the assumption that data entries are $\langle key, rid \rangle$.

In this case each rid in a leaf page could point to a different data page. Should this happen, the cost (in disk I/Os) of retrieving all data records could equal the number

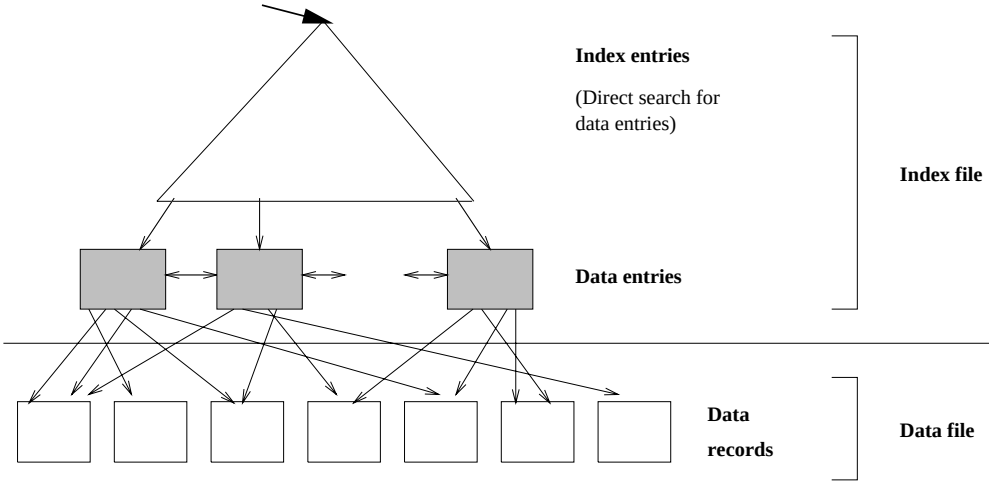

Figure 11.12 Unclustered B+ Tree for Sorting

of data records. That is, the worst-case cost is equal to the number of data records because fetching each record could require a disk I/O. This cost is in addition to the cost of retrieving leaf pages of the B+ tree to get the data entries (which point to the data records).

If p is the average number of records per data page and there are N data pages, the number of data records is $p * N$. If we take f to be the ratio of the size of a data entry to the size of a data record, we can approximate the number of leaf pages in the tree by $f * N$. The total cost of retrieving records in sorted order using an unclustered B+ tree is therefore $(f + p) * N$. Since f is usually 0.1 or smaller and p is typically much larger than 10, $p * N$ is a good approximation.

In practice, the cost may be somewhat less because some rids in a leaf page will lead to the same data page, and further, some pages will be found in the buffer pool, thereby avoiding an I/O. Nonetheless, the usefulness of an unclustered B+ tree index for sorted retrieval is highly dependent on the extent to which the order of data entries corresponds—and this is just a matter of chance—to the physical ordering of data records.

We illustrate the cost of sorting a file of records using external sorting and unclustered B+ tree indexes in Figure 11.13. The costs shown for the unclustered index are worst-case numbers and are based on the approximate formula $p * N$. For comparison, note that the cost for a clustered index is approximately equal to N, the number of pages of data records.

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

Figure 11.13 Cost of External Sorting (B=1,000, b=32) versus Unclustered Index

Keep in mind that p is likely to be closer to 100 and that B is likely to be higher than 1,000 in practice. The ratio of the cost of sorting versus the cost of using an unclustered index is likely to be even lower than is indicated by Figure 11.13 because the I/O for sorting is in 32-page buffer blocks, whereas the I/O for the unclustered indexes is one page at a time. The value of p is determined by the page size and the size of a data record; for p to be 10, with 4 KB pages, the average data record size must be about 400 bytes. In practice, p is likely to be greater than 10.

For even modest file sizes, therefore, sorting by using an unclustered index is clearly inferior to external sorting. Indeed, even if we want to retrieve only about 10 to 20 percent of the data records, for example, in response to a range query such as "Find all sailors whose rating is greater than 7," sorting the file may prove to be more efficient than using an unclustered index!

11.5 POINTS TO REVIEW

- An *external sorting algorithm* sorts a file of arbitrary length using only a limited amount of main memory. The *two-way merge sort* algorithm is an external sorting algorithm that uses only three buffer pages at any time. Initially, we break the file into small sorted files called *runs* of the size of one page. The algorithm then proceeds in passes. In each pass, runs are paired and merged into sorted runs twice the size of the input runs. In the last pass, the merge of two runs results in a sorted instance of the file. The number of passes is $\lceil log_2 N \rceil + 1$, where N is the number of pages in the file. **(Section 11.1)**
- The *external merge sort* algorithm improves upon the two-way merge sort if there are $B > 3$ buffer pages available for sorting. The algorithm writes initial runs of B pages each instead of only one page. In addition, the algorithm merges $B-1$ runs instead of two runs during the merge step. The number of passes is reduced to $\lceil log_{B-1} N1 \rceil + 1$, where $N1 = \lceil N/B \rceil$. The average length of the initial runs can be increased to $2 * B$ pages, reducing $N1$ to $N1 = \lceil N/(2 * B) \rceil$. **(Section 11.2)**

- In *blocked I/O* we read or write several consecutive pages (called a *buffer block*) through a single request. Blocked I/O is usually much cheaper than reading or writing the same number of pages through independent I/O requests. Thus, in external merge sort, instead of merging $B-1$ runs, usually only $\lfloor \frac{B-b}{b} \rfloor$ runs are merged during each pass, where b is the buffer block size. In practice, all but the largest files can be sorted in just two passes, even using blocked I/O. In *double buffering*, each buffer is duplicated. While the CPU processes tuples in one buffer, an I/O request for the other buffer is issued. **(Section 11.3)**
- If the file to be sorted has a clustered B+ tree index with a search key equal to the fields to be sorted by, then we can simply scan the sequence set and retrieve the records in sorted order. This technique is clearly superior to using an external sorting algorithm. If the index is unclustered, an external sorting algorithm will almost certainly be cheaper than using the index. **(Section 11.4)**

EXERCISES

Exercise 11.1 Suppose that you have a file with 10,000 pages and that you have three buffer pages. Answer the following questions for each of these scenarios, assuming that our most general external sorting algorithm is used:

(a) A file with 10,000 pages and three available buffer pages.

(b) A file with 20,000 pages and five available buffer pages.

(c) A file with 2,000,000 pages and 17 available buffer pages.

1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

Exercise 11.2 Answer Exercise 11.1 assuming that a two-way external sort is used.

Exercise 11.3 Suppose that you just finished inserting several records into a heap file, and now you want to sort those records. Assume that the DBMS uses external sort and makes efficient use of the available buffer space when it sorts a file. Here is some potentially useful information about the newly loaded file and the DBMS software that is available to operate on it:

> The number of records in the file is 4,500. The sort key for the file is four bytes long. You can assume that rids are eight bytes long and page ids are four bytes long. Each record is a total of 48 bytes long. The page size is 512 bytes. Each page has 12 bytes of control information on it. Four buffer pages are available.

1. How many sorted subfiles will there be after the initial pass of the sort, and how long will each subfile be?

2. How many passes (including the initial pass considered above) will be required to sort this file?
3. What will be the total I/O cost for sorting this file?
4. What is the largest file, in terms of the number of records, that you can sort with just four buffer pages in two passes? How would your answer change if you had 257 buffer pages?
5. Suppose that you have a B+ tree index with the search key being the same as the desired sort key. Find the cost of using the index to retrieve the records in sorted order for each of the following cases:
   - The index uses Alternative (1) for data entries.
   - The index uses Alternative (2) and is not clustered. (You can compute the worst-case cost in this case.)
   - How would the costs of using the index change if the file is the largest that you can sort in two passes of external sort with 257 buffer pages? Give your answer for both clustered and unclustered indexes.

Exercise 11.4 Consider a disk with an average seek time of 10ms, average rotational delay of 5ms, and a transfer time of 1ms for a 4K page. Assume that the cost of reading/writing a page is the sum of these values (i.e., 16ms) unless a *sequence* of pages is read/written. In this case the cost is the average seek time plus the average rotational delay (to find the first page in the sequence) plus 1ms per page (to transfer data). You are given 320 buffer pages and asked to sort a file with 10,000,000 pages.

1. Why is it a bad idea to use the 320 pages to support virtual memory, that is, to 'new' 10,000,000*4K bytes of memory, and to use an in-memory sorting algorithm such as Quicksort?
2. Assume that you begin by creating sorted runs of 320 pages each in the first pass. Evaluate the cost of the following approaches for the subsequent merging passes:
   (a) Do 319-way merges.
   (b) Create 256 'input' buffers of 1 page each, create an 'output' buffer of 64 pages, and do 256-way merges.
   (c) Create 16 'input' buffers of 16 pages each, create an 'output' buffer of 64 pages, and do 16-way merges.
   (d) Create eight 'input' buffers of 32 pages each, create an 'output' buffer of 64 pages, and do eight-way merges.
   (e) Create four 'input' buffers of 64 pages each, create an 'output' buffer of 64 pages, and do four-way merges.

Exercise 11.5 Consider the refinement to the external sort algorithm that produces runs of length $2B$ on average, where B is the number of buffer pages. This refinement was described in Section 11.2.1 under the assumption that all records are the same size. Explain why this assumption is required and extend the idea to cover the case of variable length records.

PROJECT-BASED EXERCISES

Exercise 11.6 (*Note to instructors: Additional details must be provided if this exercise is assigned; see Appendix B.*) Implement external sorting in Minibase.

BIBLIOGRAPHIC NOTES

Knuth's text [381] is the classic reference for sorting algorithms. Memory management for replacement sort is discussed in [408]. A number of papers discuss parallel external sorting algorithms, including [55, 58, 188, 429, 495, 563].