

# Cosequential Processing and the Sorting of Large Files

## CHAPTER OBJECTIVES

- ❖ Describe a class of frequently used processing activities known as *cosequential processes*.
- ❖ Provide a general object-oriented model for implementing all varieties of cosequential processes.
- ❖ Illustrate the use of the model to solve a number of different kinds of cosequential processing problems, including problems other than simple merges and matches.
- ❖ Introduce *heapsort* as an approach to overlapping I/O with sorting in memory.
- ❖ Show how merging provides the basis for sorting very large files.
- ❖ Examine the costs of  $K$ -way merges on disk and find ways to reduce those costs.
- ❖ Introduce the notion of *replacement selection*.
- ❖ Examine some of the fundamental concerns associated with sorting large files using tapes rather than disks.
- ❖ Introduce Unix utilities for sorting, merging, and cosequential processing.

## CHAPTER OUTLINE

### **8.1 An Object-Oriented Model for Implementing Cosequential Processes**

8.1.1 Matching Names in Two Lists

8.1.2 Merging Two Lists

8.1.3 Summary of the Cosequential Processing Model

### **8.2 Application of the Model to a General Ledger Program**

8.2.1 The Problem

8.2.2 Application of the Model to the Ledger Program

### **8.3 Extension of the Model to Include Multiway Merging**

8.3.1 A  $K$ -way Merge Algorithm

8.3.2 A Selection Tree for Merging Large Numbers of Lists

### **8.4 A Second Look at Sorting in Memory**

8.4.1 Overlapping Processing and I/O: Heapsort

8.4.2 Building the Heap While Reading the File

8.4.3 Sorting While Writing to the File

### **8.5 Merging as a Way of Sorting Large Files on Disk**

8.5.1 How Much Time Does a Merge Sort Take?

8.5.2 Sorting a File That Is Ten Times Larger

8.5.3 The Cost of Increasing the File Size

8.5.4 Hardware-Based Improvements

8.5.5 Decreasing the Number of Seeks Using Multiple-Step Merges

8.5.6 Increasing Run Lengths Using Replacement Selection

8.5.7 Replacement Selection Plus Multistep Merging

8.5.8 Using Two Disk Drives with Replacement Selection

8.5.9 More Drives? More Processors?

8.5.10 Effects of Multiprogramming

8.5.11 A Conceptual Toolkit for External Sorting

### **8.6 Sorting Files on Tape**

8.6.1 The Balanced Merge

8.6.2 The  $K$ -way Balanced Merge

8.6.3 Multiphase Merges

8.6.4 Tapes versus Disks for External Sorting

### **8.7 Sort-Merge Packages**

### **8.8 Sorting and Cosequential Processing in Unix**

8.8.1 Sorting and Merging in Unix

8.8.2 Cosequential Processing Utilities in Unix

Cosequential operations involve *the coordinated processing of two or more sequential lists to produce a single output list*. Sometimes the processing results in a *merging*, or *union*, of the items in the input lists; sometimes the goal is a *matching*, or *intersection*, of the items in the lists; and other times the operation is a combination of matching and merging. These kinds of operations on sequential lists are the basis of a great deal of file processing.

In the first half of this chapter we develop a general object-oriented model for performing cosequential operations, illustrate its use for simple matching and merging operations, then apply it to the development of a more complex general ledger program. Next we apply the model to multiway merging, which is an essential component of external sort-merge operations. We conclude the chapter with a discussion of external sort-merge procedures, strategies, and trade-offs, paying special attention to performance considerations.

1

## **An Object-Oriented Model for Implementing Cosequential Processes**

---

Cosequential operations usually appear to be simple to construct; given the information that we provide in this chapter, this appearance of simplicity can be turned into reality. However, it is also true that approaches to cosequential processing are often confused, poorly organized, and incorrect. These examples of bad practice are by no means limited to student programs: the problems also arise in commercial programs and textbooks. The difficulty with these incorrect programs is usually that they are not organized around a single, clear model for cosequential processing. Instead, they seem to deal with the various exception conditions and problems of a cosequential process in an *ad hoc* rather than systematic way.

This section addresses such lack of organization head on. We present a single, simple model that can be the basis for the construction of any kind of cosequential process. By understanding and adhering to the design principles inherent in the model, you will be able to write cosequential procedures that are simple, short, and robust.

We present this model by defining a class `CosequentialProcess` that supports processing of any type of list, in the same way that class `IOBuffer` supports buffer operations on any type of buffer. Class `CosequentialProcess` includes operations to match and merge lists. It defines the list processing operations required for cosequential processing

as virtual methods. We will then define new subclasses that include the methods for accessing the elements of particular types of lists.

### 8.1.1 Matching Names in Two Lists

Suppose we want to output the names common to the two lists shown in Fig. 8.1. This operation is usually called a *match operation*, or an *intersection*. We assume, for the moment, that we will not allow duplicate names within a list and that the lists are sorted in ascending order.

We begin by reading in the initial item from each list, and we find that they match. We output this first item as a member of the *match set*, or *intersection set*. We then read in the next item from each list. This time the

List 1	List 2
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREWS
DAVIS	BECH
FOSTER	BURNS
GARWICK	CARTER
JAMES	DAVIS
JOHNSON	DEMPSEY
KARNS	GRAY
LAMBERT	JAMES
MILLER	JOHNSON
PETERS	KATZ
RESTON	PETERS
ROSEWALD	ROSEWALD
TURNER	SCHMIDT
	THAYER
	WALKER
	WILLIS

**Figure 8.1** Sample input lists for cosequential operations.

item in List 2 is less than the item in List 1. When we are processing these lists visually as we are now, we remember that we are trying to match the item CARTER from List 1 and scan down List 2 until we either find it or jump beyond it. In this case, we eventually find a match for CARTER, so we output it, read in the next item from each list, and continue the process. Eventually we come to the end of one of the lists. Since we are looking for items common to both lists, we know we can stop at this point.

Although the match procedure appears to be quite simple, there are a number of matters that have to be dealt with to make it work reasonably well.

- *Initializing*: we need to arrange things in such a way that the procedure gets going properly.
- *Getting and accessing the next list item*: we need simple methods that support getting the next list element and accessing it.
- *Synchronizing*: we have to make sure that the current item from one list is never so far ahead of the current item on the other list that a match will be missed. Sometimes this means getting the next item from List 1, sometimes from List 2, sometimes from both lists.
- *Handling end-of-file conditions*: when we get to the end of either List 1 or List 2, we need to halt the program.
- *Recognizing errors*: when an error occurs in the data (for example, duplicate items or items out of sequence), we want to detect it and take some action.

Finally, we would like our algorithm to be reasonably efficient, simple, and easy to alter to accommodate different kinds of data. The key to accomplishing these objectives in the model we are about to present lies in the way we deal with the third item in our list—synchronizing.

At each step in the processing of the two lists, we can assume that we have two items to compare: a current item from List 1 and a current item from List 2. Let's call these two current items *Item(1)* and *Item(2)*. We can compare the two items to determine whether *Item(1)* is less than, equal to, or greater than *Item(2)*:

- If *Item(1)* is *less than* *Item(2)*, we get the next item from List 1;
- If *Item(1)* is *greater than* *Item(2)*, we get the next item from List 2; and
- If the items are the same, we output the item and get the next items from the two lists.

It turns out that this can be handled very cleanly with a single loop containing one three-way conditional statement, as illustrated in the algorithm of Fig. 8.2. The key feature of this algorithm is that *control always returns to the head of the main loop after every step of the operation*. This means that no extra logic is required within the loop to handle the case when List 1 gets ahead of List 2, or List 2 gets ahead of List 1, or the end-of-file condition is reached on one list before it is on the other. Since each pass through the main loop looks at the next pair of items, the fact that one list may be longer than the other does not require any special logic. Nor does the end-of-file condition—each operation to get a new item resets the MoreNames flag that records whether items are available in both lists. The while statement simply checks the value of the flag MoreNames on every cycle.

---

```

Match (char * List1Name, char * List2Name,
      char * OutputListName)

int MoreItems; // true if items remain in both of the lists

// initialize input and output lists
InitializeList (1, List1Name); // initialize List 1
InitializeList (2, List2Name); // initialize List 2
InitializeOutput (OutputListName);

// get first item from both lists
MoreItems = NextItemInList(1) && NextItemInList(2);

while (MoreItems){ // loop until no items in one of the lists
    if (Item(1) < Item(2))
        MoreItems = NextItemInList(1);
    else if (Item(1) == Item(2)) // Item1 == Item2
    {
        ProcessItem (1); // match found
        MoreItems = NextItemInList(1) && NextItemInList(2);
    }
    else // Item(1) > Item(2)
        MoreItems = NextItemInList(2);
}
FinishUp();
return 1;

```

---

Figure 8.2 Cosequential match function based on a single loop.

The logic inside the loop is equally simple. Only three possible conditions can exist after reading an item: the *if-then-else* logic handles all of them. Because we are implementing a match process here, output occurs only when the items are the same.

Note that the main program does not concern itself with such matters as getting the next item, sequence checking, and end-of-file detection. Since their presence in the main loop would only obscure the main synchronization logic, they have been relegated to supporting methods. It is also the case that these methods are specific to the particular type of lists being used and must be different for different applications.

Method `NextItemInList` has a single parameter that identifies which list is to be manipulated. Its responsibility is to read the next name from the file, store it somewhere, and return *true* if it was able to read another name and *false* otherwise. It can also check the condition that the list must be in ascending order with no duplicate entries.

Method `Match` must be supported by defining methods `InitializeList`, `InitializeOutput`, `NextItemInList`, `Item`, `ProcessItem`, and `FinishUp`. The `Match` method is perfectly general and is not dependent on the type of the items nor on the way the lists are represented. These details are provided by the supporting methods that need to be defined for the specific needs of particular applications. What follows is a description of a class `CosequentialProcessing` that supports method `Match` and a class `StringListProcess` that defines the supporting operations for the lists like those of Figure 8.1.

Class `CosequentialProcessing`, as given in Fig. 8.3 and in file `coseq.h` and `coseq.cpp` of Appendix H, encapsulates the ideas of cosequential processing that were described in the earlier example of list matching. Note that this is an abstract class, since it does not contain definitions of the supporting methods. This is a template class so the operations that compare list items can be different for different applications. The code of method `Match` in Fig. 8.2 is exactly that of method `Match2Lists` of this class, as you can see in file `coseq.cpp`.

In order to use class `CosequentialProcess` for the application described earlier, we need to create a subclass `StringListProcess` that defines the specific supporting methods. Figure 8.4 (and file `strlist.h` of Appendix H) shows the definition of class `StringListProcess`. The implementations of the methods are given in file `strlist.cpp` of Appendix H. The class definition allows any number of input lists. Protected members are included for the input and

---

```

template <class ItemType>
class CosequentialProcess
// base class for cosequential processing
{public:
    // The following methods provide basic list processing
    // These must be defined in subclasses
    virtual int InitializeList (int ListNumber, char * ListName)=0;
    virtual int InitializeOutput (char * OutputListName)=0;
    virtual int NextItemInList (int ListNumber)=0;
        //advance to next item in this list
    virtual ItemType Item (int ListNumber) = 0;
        // return current item from this list
    virtual int ProcessItem (int ListNumber)=0;
        // process the item in this list
    virtual int FinishUp()=0; // complete the processing

    // 2-way cosequential match method
    virtual int Match2Lists
        (char * List1Name, char * List2Name, char * OutputListName);
};

```

---

**Figure 8.3** Main members and methods of a general class for cosequential processing.

output files and for the values of the current item of each list. Member `LowValue` is a value that is smaller than any value that can appear in a list—in this case, the null string (`" "`). `LowValue` is used so that method `NextItemInList` does not have to get the first item in any special way. Member `HighValue` has a similar use, as we will see in the next section.

Given these classes, you should be able to work through the two lists provided in Fig. 8.1, following the code, and demonstrate to yourself that these simple procedures can handle the various resynchronization problems that these sample lists present. A main program (file `match.cpp`) to process the lists stored in files `list1.txt` and `list2.txt` is

```

#include "coseq.h"
int main ()
{
    StringListProcess ListProcess(2); // process with 2 lists
    ListProcess.Match2Lists ("list1.txt", "list2.txt", "match.txt");
}

```



```

class StringListProcess: public CosequentialProcess<String&>
{
    Class to process lists that are files of strings, one per line

public:
    StringListProcess (int NumberOfLists); // constructor

    // Basic list processing methods
    int InitializeList (int ListNumber, char * List1Name);
    int InitializeOutput (char * OutputListName);
    int NextItemInList (int ListNumber); //get next
    String& Item (int ListNumber); //return current
    int ProcessItem (int ListNumber); // process the item
    int FinishUp(); // complete the processing

protected:
    ifstream * Lists; // array of list files
    String * Items; // array of current Item from each list
    ofstream OutputList;
    static const char * LowValue;
    static const char * HighValue;
};

```

Figure 8.4 A subclass to support lists that are files of strings, one per line.

### 8.1.2 Merging Two Lists

The three-way-test, single-loop model for cosequential processing can easily be modified to handle *merging* of lists simply by producing output for every case of the *if-then-else* construction since a merge is a *union* of the list contents.

An important difference between matching and merging is that with merging we must read *completely* through each of the lists. This necessitates a change in how `MoreNames` is set. We need to keep this flag set to `TRUE` as long as there are records in *either* list. At the same time, we must recognize that one of the lists has been read completely, and we should avoid trying to read from it again. Both of these goals can be achieved if we introduce two `MoreNames` variables, one for each list, and set the stored `Item` value for the completed list to some value (we call it `HighValue`) that

- Cannot possibly occur as a legal input value, and

- Has a *higher* collating sequence value than any possible legal input value. In other words, this special value would come *after* all legal input values in the file's ordered sequence.

For HighValue, we use the string "\xFF" which is a string of only one character and that character has the hex value FF, which is the largest character value.

Method Merge2Lists is given in Fig. 8.5 and in file coseq.cpp of Appendix H. This method has been added to class CosequentialProcess. No modifications are required to class StringListProcess.

---

```
template <class ItemType>
int CosequentialProcess<ItemType>::Merge2Lists
(char * List1Name, char * List2Name, char * OutputListName)
{
    int MoreItems1, MoreItems2; // true if more items in list
    InitializeList (1, List1Name);
    InitializeList (2, List2Name);
    InitializeOutput (OutputListName);
    MoreItems1 = NextItemInList(1);
    MoreItems2 = NextItemInList(2);

    while (MoreItems1 || MoreItems2){// if either file has more
        if (Item(1) < Item(2))
            { // list 1 has next item to be processed
                ProcessItem (1);
                MoreItems1 = NextItemInList(1);
            }
        else if (Item(1) == Item(2))
            { // lists have the same item, process from list 1
                ProcessItem (1);
                MoreItems1 = NextItemInList(1);
                MoreItems2 = NextItemInList(2);
            }
        else // Item(1) > Item(2)
            { // list 2 has next item to be processed
                ProcessItem (2);
                MoreItems2 = NextItemInList(2);
            }
    }
    FinishUp();
    return 1;
}
```

---

**Figure 8.5** Cosequential merge procedure based on a single loop.

Once again, you should use this logic to work, step by step, through the lists provided in Fig. 8.1 to see how the resynchronization is handled and how the use of the `HighValue` forces the procedure to finish both lists before terminating.

With these two examples, we have covered all of the pieces of our model. Now let us summarize the model before adapting it to a more complex problem.

### 8.1.3 Summary of the Cosequential Processing Model

Generally speaking, the model can be applied to problems that involve the performance of set operations (union, intersection, and more complex processes) on two or more sorted input files to produce one or more output files. In this summary of the cosequential processing model, we assume that there are only two input files and one output file. It is important to understand that the model makes certain general assumptions about the nature of the data and type of problem to be solved. Following is a list of the assumptions, together with clarifying comments.

---

#### Assumptions

Two or more input files are to be processed in a parallel fashion to produce one or more output files.

Each file is sorted on one or more key fields, and all files are ordered in the same ways on the same fields.

In some cases, there must exist a high-key value that is greater than any legitimate record key and a low-key value that is less than any legitimate record key.

Records are to be processed in logical sorted order.

#### Comments

In some cases an output file may be the same file as one of the input files.

It is not necessary that all files have the same record structures.

The use of a high-key value and a low-key value is not absolutely necessary, but it can help avoid the need to deal with beginning-of-file and end-of-file conditions as special cases, hence decreasing complexity.

The physical ordering of records is irrelevant to the model, but in practice it may be important to the way the model is implemented. Physical ordering can have a large impact on processing efficiency.

*Assumptions (cont.)*

For each file there is only one current record. This is the record whose key is accessible within the main synchronization loop.

Records can be manipulated only in internal memory.

*Comments (cont.)*

The model does not prohibit looking ahead or looking back at records, but such operations should be restricted to subclasses and should not be allowed to affect the structure of the main synchronization loop.

A program cannot alter a record in place on secondary storage.

Given these assumptions, the essential components of the model are:

1. Initialization. Previous item values for all files are set to the low value; then current records for all files are read from the first logical records in the respective files.
2. One main synchronization loop is used, and the loop continues as long as relevant records remain.
3. Within the body of the main synchronization loop is a selection based on comparison of the record keys from respective input file records. If there are two input files, the selection takes the form given in function Match of Fig. 8.2.
4. Input files and output files are sequence checked by comparing the previous item value with the new item value when a record is read. After a successful sequence check, the previous item value is set to the new item value to prepare for the next input operation on the corresponding file.
5. High values are substituted for actual key values when end-of-file occurs. The main processing loop terminates when high values have occurred for all relevant input files. The use of high values eliminates the need to add special code to deal with each end-of-file condition. (This step is not needed in a pure match procedure because a match procedure halts when the first end-of-file condition is encountered.)
6. All possible I/O and error detection activities are to be relegated to supporting methods so the details of these activities do not obscure the principal processing logic.

This three-way-test, single-loop model for creating cosequential processes is both simple and robust. You will find very few applications requiring the coordinated sequential processing of two files that cannot be handled neatly and efficiently with the model. We now look at a problem that is much more complex than a simple match or merge but that nevertheless lends itself nicely to solution by means of the model.

## 8.2 Application of the Model to a General Ledger Program

### 8.2.1 The Problem

Suppose we are given the problem of designing a general ledger posting program as part of an accounting system. The system includes a journal file and a ledger file. The ledger contains month-by-month summaries of the values associated with each of the bookkeeping accounts. A sample portion of the ledger, containing only checking and expense accounts, is illustrated in Fig. 8.6.

The journal file contains the monthly transactions that are ultimately to be posted to the ledger file. Figure 8.7 shows these journal transactions. Note that the entries in the journal file are paired. This is because every check involves both subtracting an amount from the checking account balance and adding an amount to at least one expense account. The accounting-program package needs procedures for creating this journal file interactively, probably outputting records to the file as checks are keyed in and then printed.

Acct. No.	Account title	Jan	Feb	Mar	Apr
101	Checking account #1	1032.57	2114.56	5219.23	
102	Checking account #2	543.78	3094.17	1321.20	
505	Advertising expense	25.00	25.00	25.00	
510	Auto expenses	195.40	307.92	501.12	
515	Bank charges	0.00	0.00	0.00	
520	Books and publications	27.95	27.95	87.40	
525	Interest expense	103.50	255.20	380.27	
535	Miscellaneous expense	12.45	17.87	23.87	
540	Office expense	57.50	105.25	138.37	
545	Postage and shipping	21.00	27.63	57.45	
550	Rent	500.00	1000.00	1500.00	
555	Supplies	112.00	167.50	2441.80	

**Figure 8.6** Sample ledger fragment containing checking and expense accounts.

Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
510	1271	04/02/97	Tune-up and minor repair	78.70
101	1272	04/02/97	Rent	-500.00
550	1272	04/02/97	Rent for April	500.00
101	1273	04/04/97	Advertising	-87.50
505	1273	04/04/97	Newspaper ad re: new product	87.50
102	670	04/02/97	Office expense	-32.78
540	670	04/02/97	Printer cartridge	32.78
101	1274	04/02/97	Auto expense	-31.83
510	1274	04/09/97	Oil change	31.83

Figure 8.7 Sample journal entries.

Once the journal file is complete for a given month, meaning that it contains all of the transactions for that month, the journal must be posted to the ledger. *Posting* involves associating each transaction with its account in the ledger. For example, the printed output produced for accounts 101, 102, 505, and 510 during the posting operation, given the journal entries in Fig. 8.7, might look like the output illustrated in Fig. 8.8.

101	Checking account #1			
	1271	04/02/86	Auto expense	-78.70
	1272	04/02/97	Rent	-500.00
	1273	04/04/97	Advertising	-87.50
	1274	04/02/97	Auto expense	-31.83
			Prev. bal: 5219.23	New bal: 4521.20
102	Checking account #2			
	670	04/02/97	Office expense	-32.78
			Prev. bal: 1321.20	New bal: 1288.42
505	Advertising expense			
	1273	04/04/97	Newspaper ad re: new product	87.50
			Prev. bal: 25.00	New bal: 112.50
510	Auto expenses			
	1271	04/02/97	Tune-up and minor repair	78.70
	1274	04/09/97	Oil change	31.83
			Prev. bal: 501.12	New bal: 611.65

Figure 8.8 Sample ledger printout showing the effect of posting from the journal.

How is the posting process implemented? Clearly, it uses the account number as a *key* to relate the journal transactions to the ledger records. One possible solution involves building an index for the ledger so we can work through the journal transactions using the account number in each journal entry to look up the correct ledger record. But this solution involves seeking back and forth across the ledger file as we work through the journal. Moreover, this solution does not really address the issue of creating the output list, in which all the journal entries relating to an account are collected together. Before we could print the ledger balances and collect journal entries for even the first account, 101, we would have to proceed all the way through the journal list. Where would we save the transactions for account 101 as we collect them during this complete pass through the journal?

A much better solution is to begin by collecting all the journal transactions that relate to a given account. This involves sorting the journal transactions by account number, producing a list ordered as in Fig. 8.9.

Now we can create our output list by working through both the ledger and the sorted journal *cosequentially*, meaning that we process the two lists sequentially and in parallel. This concept is illustrated in Fig. 8.10. As we start working through the two lists, we note that we have an initial match on account number. We know that multiple entries are possible in the journal file but not in the ledger, so we move ahead to the next entry in the

Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
101	1272	04/02/97	Rent	-500.00
101	1273	04/04/97	Advertising	-87.50
101	1274	04/02/97	Auto expense	-31.83
102	670	04/02/97	Office expense	-32.78
505	1273	04/04/97	Newspaper ad re: new product	87.50
510	1271	04/02/97	Tune-up and minor repair	78.70
510	1274	04/09/97	Oil change	31.83
540	670	04/02/97	Printer cartridge	32.78
550	1272	04/02/97	Rent for April	500.00

Figure 8.9 List of journal transactions sorted by account number.

journal. The account numbers still match. We continue doing this until the account numbers no longer match. We then *resynchronize* the cosequential action by moving ahead in the ledger list. This process is often referred to as a *master-transaction process*. In this case the ledger entry is the master record and the journal entry is the transaction entry.

This matching process seems simple, as in fact it is, as long as every account in one file also appears in another. But there will be ledger accounts for which there is no journal entry, and there can be typographical errors that create journal account numbers that do not exist in the ledger. Such situations can make resynchronization more complicated and can result in erroneous output or infinite loops if the programming is done in an ad hoc way. By using the cosequential processing model, we can guard against these problems. Let us now apply the model to our ledger problem.

### 8.2.2 Application of the Model to the Ledger Program

The monthly ledger posting program must perform two tasks:

- It needs to update the ledger file with the correct balance for each account for the current month.
- It must produce a printed version of the ledger that not only shows the beginning and current balance for each account but also lists all the journal transactions for the month.

Ledger List		Journal List		
101	Checking account #1	101	1271	Auto expense
		101	1272	Rent
		101	1273	Advertising
		101	1274	Auto expense
102	Checking account #2	102	670	Office expense
505	Advertising expense	505	1273	Newspaper ad re: new product
510	Auto expenses	510	1271	Tune-up and minor repair
		510	1274	Oil change

**Figure 8.10** Conceptual view of cosequential matching of the ledger and journal files.



We focus on the second task as it is the more difficult. Let's look again at the form of the printed output, this time extending the output to include a few more accounts as shown in Fig. 8.11. As you can see, the printed output from the monthly ledger posting program shows the balances of all ledger accounts, whether or not there were transactions for the account. From the point of view of the ledger accounts, the process is a *merge*, since even unmatched ledger accounts appear in the output.

What about unmatched journal accounts? The ledger accounts and journal accounts are not equal in authority. The ledger file *defines* the set of legal accounts; the journal file contains entries that are to be *posted* to the accounts listed in the ledger. The existence of a journal account that does not match a ledger account indicates an error. From the point of view of the journal accounts, the posting process is strictly one of *matching*. Our post method needs to implement a kind of combined merging/matching algorithm while simultaneously handling the chores of printing account title lines, individual transactions, and summary balances.

01	Checking account #1			
	1271	04/02/86	Auto expense	-78.70
	1272	04/02/97	Rent	-500.00
	1274	04/02/97	Auto expense	-31.83
	1273	04/04/97	Advertising	-87.50
		Prev. bal:	5219.23	New bal: 4521.20
02	Checking account #2			
	670	04/02/97	Office expense	-32.78
		Prev. bal:	1321.20	New bal: 1288.42
05	Advertising expense			
	1273	04/04/97	Newspaper ad re: new product	87.50
		Prev. bal:	25.00	New bal: 112.50
10	Auto expenses			
	1271	04/02/97	Tune-up and minor repair	78.70
	1274	04/09/97	Oil change	31.83
		Prev. bal:	501.12	New bal: 611.65
15	Bank charges			
		Prev. bal:	0.00	New bal: 0.00
20	Books and publications			
		Prev. bal:	87.40	New bal: 87.40

Figure 8.11 Sample ledger printout for the first six accounts.

In summary, there are three different steps in processing the ledger entries:

1. Immediately after reading a new ledger object, we need to print the header line and initialize the balance for the next month from the previous month's balance.
2. For each transaction object that matches, we need to update the account balance.
3. After the last transaction for the account, the balance line should be printed. This is the place where a new ledger record could be written to create a new ledger file.

This posting operation is encapsulated by defining subclass `MasterTransactionProcess` of `CosequentialProcess` and defining three new pure virtual methods, one for each of the steps in processing ledger entries. Then we can give the full implementation of the posting operation as a method of this class. Figure 8.12 shows the definition of this class. Figure 8.13 has the code for the three-way-test loop of method `PostTransactions`. The new methods of the class are used for processing the master records (in this case the ledger entries). The transaction records (journal entries) can be processed by the `ProcessItem` method that is in the base class.

The reasoning behind the three-way test is as follows:

1. If the ledger (master) account number (`Item[1]`) is less than the journal (transaction) account number (`Item[2]`), then there are no more transactions to add to the ledger account this month (perhaps there were none at all), so we print the ledger account balances (`ProcessEndMaster`) and read in the next ledger account (`NextItemInList(1)`). If the account exists (`MoreMasters` is true), we print the title line for the new account (`ProcessNewMaster`).
2. If the account numbers match, then we have a journal transaction that is to be posted to the current ledger account. We add the transaction amount to the account balance for the new month (`ProcessCurrentMaster`), print the description of the transaction (`ProcessItem(2)`), then read the next journal entry (`NextItemInList(1)`). Note that unlike the match case in either the matching or merging algorithms, we do not read a new entry from both accounts. This is a reflection of our acceptance of more than one journal entry for a single ledger account.

---

```

template <class ItemType>
class MasterTransactionProcess:
    public CosequentialProcess<ItemType>
// a cosequential process that supports
// master/transaction processing
{public:
    MasterTransactionProcess (); //constructor
    virtual int ProcessNewMaster ()=0;
        // processing when new master read
    virtual int ProcessCurrentMaster ()=0;
        // processing for each transaction for a master
    virtual int ProcessEndMaster ()=0;
        // processing after all transactions for a master
    virtual int ProcessTransactionError ()=0;
        // no master for transaction

    // cosequential processing of master and transaction records
    int PostTransactions (char * MasterFileName,
        char * TransactionFileName, char * OutputListName);
};

```

---

**Figure 8.12** Class MasterTransactionProcess.

---

```

while (MoreMasters || MoreTransactions)
    if (Item(1) < Item(2)) { // finish this master record
        ProcessEndMaster();
        MoreMasters = NextItemInList(1);
        if (MoreMasters) ProcessNewMaster();
    }
    else if (Item(1) == Item(2)) { // transaction matches master
        ProcessCurrentMaster(); // another transaction for master
        ProcessItem (2); // output transaction record
        MoreTransactions = NextItemInList(2);
    }
    else { // Item(1) > Item(2) transaction with no master
        ProcessTransactionError();
        MoreTransactions = NextItemInList(2);
    }
}

```

---

**Figure 8.13** Three-way-test loop for method PostTransactions of class MasterTransactionProcess.

3. If the journal account is less than the ledger account, then it is an unmatched journal account, perhaps due to an input error. We print an error message (`ProcessTransactionError`) and continue with the next transaction.

In order to complete our implementation of the ledger posting application, we need to create a subclass `LedgerProcess` that includes implementation of the `NextItemInList`, `Item`, and `ProcessItem` methods and the methods for the three steps of master record processing. This new class is given in files `ledgpost.h` and `ledgpost.cpp` of Appendix H. The master processing methods are all very simple, as shown in Fig. 8.14.

The remainder of the code for the ledger posting program, including the simple main program, is given in files `ledger.h`, `ledger.cpp`, and `post.cpp` in Appendix H. This includes the `ostream` formatting that produced Figs. 8.8 and 8.11. The classes `Ledger` and `Journal` make extensive use of the `IOBuffer` and `RecordFile` classes for their file operations.

The development of this ledger posting procedure from our basic cosequential processing model illustrates how the simplicity of the model contributes to its adaptability. We can also generalize the model in an entirely different direction, extending it to enable cosequential processing

---

```
int LedgerProcess::ProcessNewMaster ()
{
    // print the header and setup last month's balance
    ledger.PrintHeader(OutputList);
    ledger.Balances[MonthNumber] = ledger.Balances[MonthNumber-1];
}

int LedgerProcess::ProcessCurrentMaster ()
{
    // add the transaction amount to the balance for this month
    ledger.Balances[MonthNumber] += journal.Amount;
}

int LedgerProcess::ProcessEndMaster ()
{
    // print the balances line to output
    PrintBalances(OutputList,
        ledger.Balances[MonthNumber-1], ledger.Balances[MonthNumber]);
}
```

---

**Figure 8.14** Master record processing for ledger objects.

of more than two input files at once. To illustrate this, we now extend the model to include multiway merging.

## 8.3 Extension of the Model to Include Multiway Merging

The most common application of cosequential processes requiring more than two input files is a *K-way merge*, in which we want to merge *K* input lists to create a single, sequentially ordered output list. *K* is often referred to as the *order* of a *K-way merge*.

### 8.3.1 A *K*-way Merge Algorithm

Recall the synchronizing loop we use to handle a two-way merge of two lists of names. This merging operation can be viewed as a process of deciding which of two input items has the *minimum* value, outputting that item, then moving ahead in the list from which that item is taken. In the event of duplicate input items, we move ahead in each list.

Suppose we keep an array of lists and array of the items (or keys) that are being used from each list in the cosequential process:

```
list[0], list[1], list[2],... list[k-1]
Item[0], Item[1], Item[3],... Item[k-1]
```

The main loop for the merge processing requires a call to a *MinIndex* function to find the index of item with the minimum collating sequence value and an inner loop that finds all lists that are using that item:

```
int minItem = MinIndex(Item,k); // find an index of minimum item
processItem(minItem); // Item(minItem) is the next output
for (i = 0; i<k; i++) // look at each list
    if (Item(minItem) == Item(i)) // advance list i
        MoreItems[i] = NextItemInList(i);
```

Clearly, the expensive parts of this procedure are finding the minimum and testing to see in which lists the item occurs and which files therefore need to be read. Note that because the item can occur in several lists, every one of these *if* tests must be executed on every cycle through the loop. However, it is often possible to guarantee that a single item, or key, occurs in only one list. In this case, the procedure becomes simpler and more efficient.

```
int minI = minIndex(Item,k); // find index of minimum item
ProcessItem(minI); // Item[minI] is the next output
MoreItems[minI]=NextItemInList(minI);
```

The resulting merge procedure clearly differs in many ways from our initial three-way-test, single-loop merge for two lists. But, even so, the single-loop parentage is still evident: there is no looping within a list. We determine which lists have the key with the lowest value, output that key, move ahead one key in each of those lists, and loop again. The procedure is as simple as it is powerful.

### 8.3.2 A Selection Tree for Merging Large Numbers of Lists

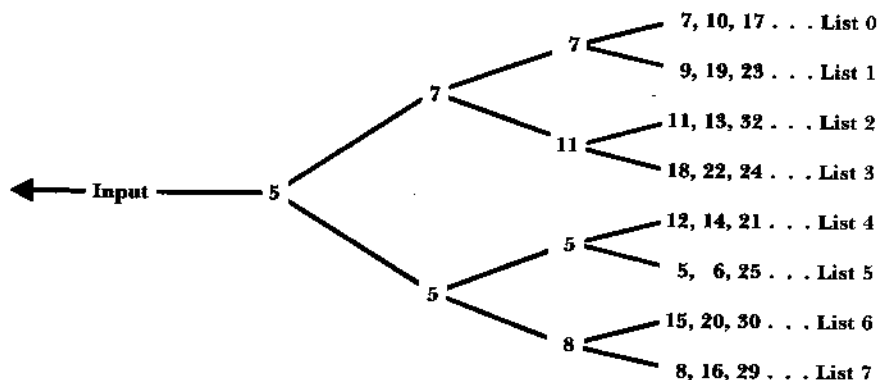
The  $K$ -way merge described earlier works nicely if  $K$  is no larger than 8 or so. When we begin merging a larger number of lists, the set of sequential comparisons to find the key with the minimum value becomes noticeably expensive. We see later that for practical reasons it is rare to want to merge more than eight files at one time, so the use of sequential comparisons is normally a good strategy. If there is a need to merge considerably more than eight lists, we could replace the loop of comparisons with a *selection tree*.

The use of a selection tree is an example of the classic time-versus-space trade-off we so often encounter in computer science. We reduce the time required to find the key with the lowest value by using a data structure to save information about the relative key values across cycles of the procedure's main loop. The concept underlying a selection tree can be readily communicated through a diagram such as that in Fig. 8.15. Here we have used lists in which the keys are numbers rather than strings.

The selection tree is a kind of *tournament* tree in which each higher-level node represents the "winner" (in this case the *minimum* key value) of the comparison between the two descendent keys. The minimum value is always at the root node of the tree. If each key has an associated reference to the list from which it came, it is a simple matter to take the key at the root, read the next element from the associated list, then run the tournament again. Since the tournament tree is a binary tree, its depth is

$$\lceil \log_2 K \rceil$$

for a merge of  $K$  lists. The number of comparisons required to establish a new tournament winner is, of course, related to this depth rather than being a linear function of  $K$ .



**Figure 8.15** Use of a selection tree to assist in the selection of a key with minimum value in a  $K$ -way merge.

## 8.4

## A Second Look at Sorting in Memory

In Chapter 6 we considered the problem of sorting a disk file that is small enough to fit in memory. The operation we described involves three separate steps:

1. Read the entire file from disk into memory.
2. Sort the records using a standard sorting procedure, such as shellsort.
3. Write the file back to disk.

The total time taken to sort the file is the sum of the times for the three steps. We see that this procedure is much faster than sorting the file in place, on the disk, because both reading and writing are sequential and each record is read once and written once.

Can we improve on the time that it takes for this memory sort? If we assume that we are reading and writing the file as efficiently as possible and we have chosen the best internal sorting routine available, it would seem not. Fortunately, there is one way that we might speed up an algorithm that has several parts, and that is to perform some of those parts in parallel.

Of the three operations involved in sorting a file that is small enough to fit into memory, is there any way to perform some of them in parallel? If we have only one disk drive, clearly we cannot overlap the reading and writing operations, but how about doing either the reading or writing (or both) at the same time that we sort the file?

### 8.4.1 Overlapping Processing and I/O: Heapsort

Most of the time when we use an internal sort, we have to wait until we have the whole file in memory before we can start sorting. Is there an internal sorting algorithm that is reasonably fast and that can begin sorting numbers immediately as they are read rather than waiting for the whole file to be in memory? In fact there is, and we have already seen part of it in this chapter. It is called *heapsort*, and it is loosely based on the same principle as the selection tree.

Recall that the selection tree compares keys as it encounters them. Each time a new key arrives, it is compared with the others; and if it is the smallest key, it goes to the root of the tree. This is very useful for our purposes because it means that we can begin sorting keys as they arrive in memory rather than waiting until the entire file is loaded before we start sorting. That is, sorting can occur in parallel with reading.

Unfortunately, in the case of the selection tree, each time a new smallest key is found, it is output to the file. We cannot allow this to happen if we want to sort the whole file because we cannot begin outputting records until we know which one comes first, second, and so on, and we won't know this until we have seen all of the keys.

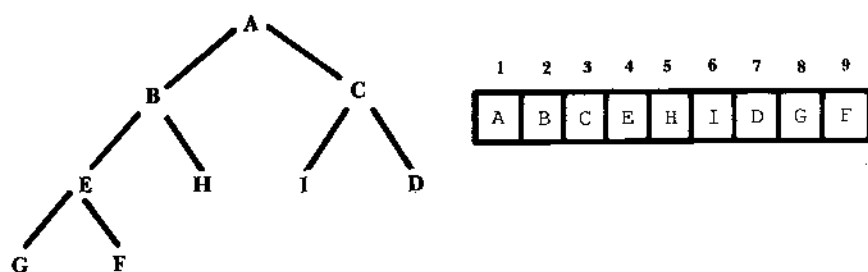
Heapsort solves this problem by keeping all of the keys in a structure called a *heap*. A heap is a binary tree with the following properties:

1. Each node has a single key, and that key is greater than or equal to the key at its parent node.
2. It is a *complete* binary tree, which means that all of its leaves are on no more than two levels and that all of the keys on the lower level are in the leftmost position.
3. Because of properties 1 and 2, storage for the tree can be allocated sequentially as an array in such a way that the root node is index 1 and the indexes of the left and right children of node  $i$  are  $2i$  and  $2i + 1$  respectively. Conversely, the index of the parent of node  $j$  is  $\lfloor j/2 \rfloor$ .

Figure 8.16 shows a heap in both its tree form and as it would be stored in an array. Note that this is only one of many possible heaps for the given set of keys. In practice, each key has an associated record that is either stored in the array with the key or pointed to by a pointer stored with the key.

Property 3 is very useful for our purposes because it means that a heap is just an array of keys in which the positions of the keys in the array are sufficient to impose an ordering on the entire set of keys. There is no need





**Figure 8.16** A heap in both its tree form and as it would be stored in an array.

for pointers or other dynamic data structuring overhead to create and maintain the heap. (As we pointed out earlier, there may be pointers associating each key with its corresponding record, but this has nothing to do with maintaining the heap.)

### 8.4.2 Building the Heap While Reading the File

The algorithm for heapsort has two parts. First we build the heap; then we output the keys in sorted order. The first stage can occur at virtually the same time that we read the data, so in terms of elapsed time it comes essentially free. The main members of a simple class `Heap` and its `Insert` method that adds a string to the heap is shown in Fig. 8.17. A full implementation of this class and a test program are in file `heapsort.cpp` in Appendix H. Figure 8.18 contains a sample application of this algorithm.

This shows how to build the heap, but it doesn't tell how to make the input overlap with the heap-building procedure. To solve that problem, we need to look at how we perform the read operation. For starters, we are not going to do a seek every time we want a new record. Instead, we read a block of records at a time into an input buffer and then operate on all of the records in the block before going on to the next block. In terms of memory storage, the input buffer for each new block of keys can be part of the memory area that is set up for the heap. Each time we read a new block, we just append it to the end of the heap (that is, the input buffer "moves" as the heap gets larger). The first new record is then at the end of the heap array, as required by the `Insert` function (Fig. 8.17). Once that record is absorbed into the heap, the next new record is at the end of the heap array, ready to be absorbed into the heap, and so forth.

---

```

class Heap
{public:
    Heap(int maxElements);
    int Insert (char * newKey);
    char * Remove();
protected:
    int MaxElements; int NumElements;
    char ** HeapArray;
    void Exchange(int i, int j); // exchange element i and j
    int Compare (int i, int j) // compare element i and j
        {return strcmp(HeapArray[i],HeapArray[j]);}
};

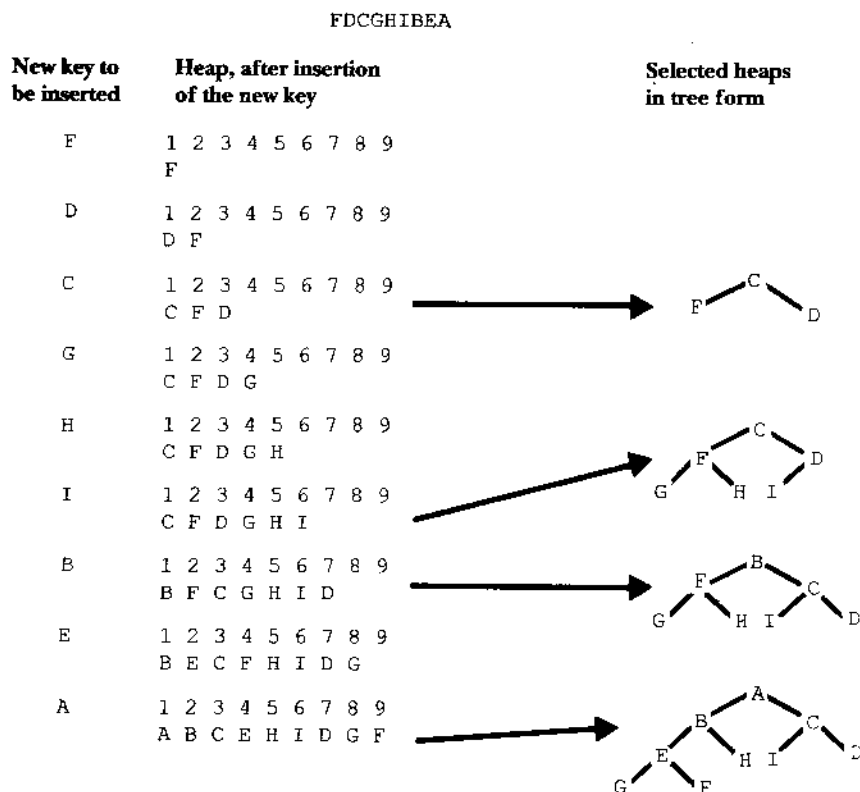
int Heap::Insert(char * newKey)
{
    if (NumElements == MaxElements) return FALSE;
    NumElements++; // add the new key at the last position
    HeapArray[NumElements] = newKey;
    // re-order the heap
    int k = NumElements; int parent;
    while (k > 1) // k has a parent
    {
        parent = k / 2;
        if (Compare(k, parent) >= 0) break;
        // HeapArray[k] is in the right place
        // else exchange k and parent
        Exchange(k, parent);
        k = parent;
    }
    return TRUE;
}

```

---

**Figure 8.17** Class Heap and method Insert.

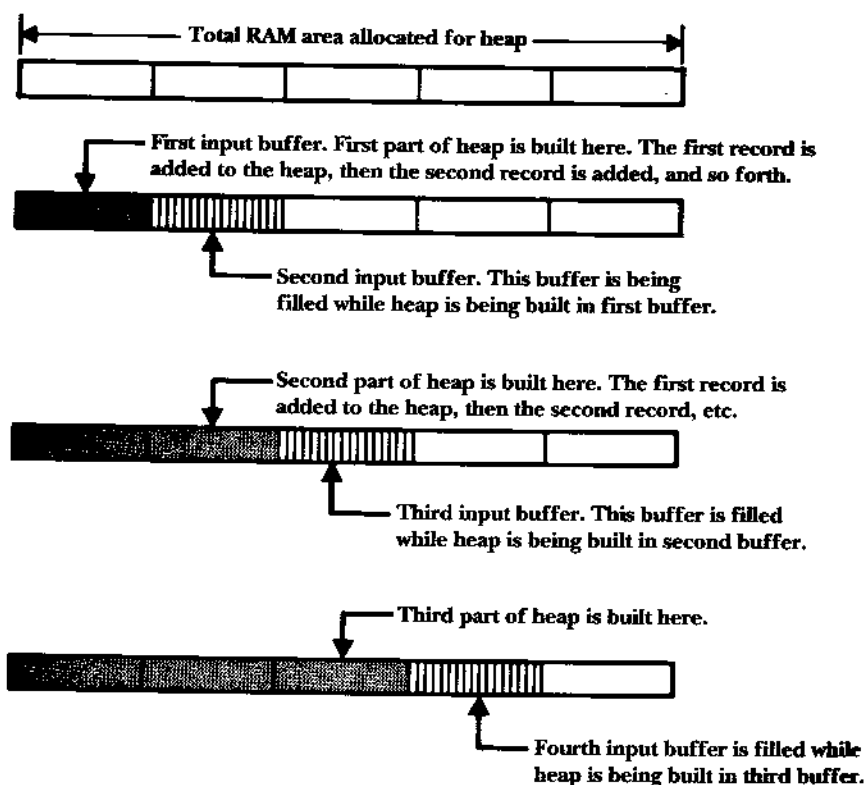
Use of an input buffer avoids an excessive number of seeks, but it still doesn't let input occur at the same time that we build the heap. We saw in Chapter 3 that the way to make processing overlap with I/O is to use more than one buffer. With multiple buffering, as we process the keys in one block from the file, we can simultaneously read later blocks from the file. If we use multiple buffers, how many should we use, and where should we put them? We already answered these questions when we decided to put each new block at the end of the array. Each time we



**Figure 8.18** Sample application of the heap-building algorithm. The keys F, D, C, G, H, I, B, E, and A are added to the heap in the order shown.

add a new block, the array gets bigger by the size of that block, in effect creating a new input buffer for each block in the file. So the number of buffers is the number of blocks in the file, and they are located in sequence in the array.

Figure 8.19 illustrates the technique that we have just described, in which we append each new block of records to the end of the heap, thereby employing a memory-sized set of input buffers. Now we read new blocks as fast as we can, never having to wait for processing before reading a new block. On the other hand, processing (heap building) cannot occur on a given block until the block to be processed is read, so there *may* be some delay in processing if processing speeds are faster than reading speeds.



**Figure 8.19** Illustration of the technique described in the text for overlapping input with heap building in memory. First read in a block into the first part of memory. The first record is the first record in the heap. Then extend the heap to include the second record, and incorporate that record into the heap, and so forth. While the first block is being processed, read in the second block. When the first block is a heap, extend it to include the first record in the second block, incorporate that record into the heap, and go on to the next record. Continue until all blocks are read in and the heap is completed.

### 8.4.3 Sorting While Writing to the File

The second and final step involves writing the heap in sorted order. Again, it is possible to overlap I/O (in this case writing) with processing. First, let's look at how to output the sorted keys. Retrieving the keys in order is simply a repetition of the following steps:

1. Determine the value of the key in the first position of the heap. This is the smallest value in the heap.

2. Move the largest value in the heap into the first position, and decrease the number of elements by one. The heap is now out of order at its root.
3. Reorder the heap by exchanging the largest element with the smaller of its children and moving down the tree to the new position of the largest element until the heap is back in order.

Each time these three steps are executed, the smallest value is retrieved and removed from the heap. Figure 8.20 contains the code for method `Remove` that implements these steps. Method `Compare` simply compares two heap elements and returns `-1` if the left element is smaller.

Again, there is nothing inherent in this algorithm that lets it overlap with I/O, but we can take advantage of certain features of the algorithm to

---

```

char * Heap::Remove()
// remove the smallest element, reorder the heap,
// and return the smallest element
// put the smallest value into 'val' for use in return
char * val = HeapArray[1];

// put largest value into root
HeapArray[1] = HeapArray[NumElements];
// decrease the number of elements
NumElements--;

// reorder the heap by exchanging and moving down
int k = 1; // node of heap that contains the largest value
int newK; // node to exchange with largest value
while (2*k <= NumElements) // k has at least one child
{
    // set newK to the index of smallest child of k
    if (Compare(2*k, 2*k+1) < 0) newK = 2*k;
    else newK = 2*k+1;
    // done if k and newK are in order
    if (Compare(k, newK) < 0) break; // in order
    Exchange(k, newK); // k and newK out of order
    k = newK; // continue down the tree
}
return val;

```

---

**Figure 8.20** Method `Remove` of class `Heap` removes the smallest element and reorders the heap.

make overlapping happen. First, we see that we know immediately which record will be written first in the sorted file; next, we know what will come second; and so forth. So as soon as we have identified a block of records we can write that block, and while we are writing that block, we can identify the next block, and so forth.

Furthermore, each time we identify a block to write, we make the heap smaller by exactly the size of a block, freeing that space for a new output buffer. So just as was the case when building the heap, we can have as many output buffers as there are blocks in the file. Again, a little coordination is required between processing and output, but the conditions exist for the two to overlap almost completely.

A final point worth making about this algorithm is that all I/O it performs is essentially sequential. All records are read in the order in which they occur in the file to be sorted, and all records are written in sorted order. The technique could work equally well if the file were kept on tape or disk. More important, since all I/O is sequential, we know that it can be done with a minimum amount of seeking.

## 3.5

### Merging as a Way of Sorting Large Files on Disk

---

In Chapter 6 we ran into problems when we needed to sort files that were too large to be wholly contained in memory. The chapter offered a partial, but ultimately unsatisfactory, solution to this problem in the form of a *keysort*, in which we needed to hold only the keys in memory, along with pointers to each key's corresponding record. Keysort had two shortcomings:

1. Once the keys were sorted, we then had to bear the substantial cost of seeking to each record in sorted order, reading each record in and then writing it into the new, sorted file.
2. With keysorting, the size of the file that can be sorted is limited by the number of key/pointer pairs that can be contained in memory. Consequently, we still cannot sort really large files.

As an example of the kind of file we cannot sort with either a memory sort or a keysort, suppose we have a file with 8 000 000 records, each of which is 100 bytes long and contains a key field that is 10 bytes long. The total length of this file is about 800 megabytes. Let us further suppose that we have 10 megabytes of memory available as a work area, not counting

memory used to hold the program, operating system, I/O buffers, and so forth. Clearly, we cannot sort the whole file in memory. We cannot even sort all the keys in memory, because it would require 80 megabytes.

The multiway merge algorithm discussed in Section 8.3 provides the beginning of an attractive solution to the problem of sorting large files such as this one. Since memory-sorting algorithms such as heapsort can work in place, using only a small amount of overhead for maintaining pointers and some temporary variables, we can create a sorted subset of our full file by reading records into memory until the memory work area is almost full, sorting the records in this work area, then writing the sorted records back to disk as a sorted subfile. We call such a sorted subfile a *run*. Given the memory constraints and record size in our example, a run could contain approximately

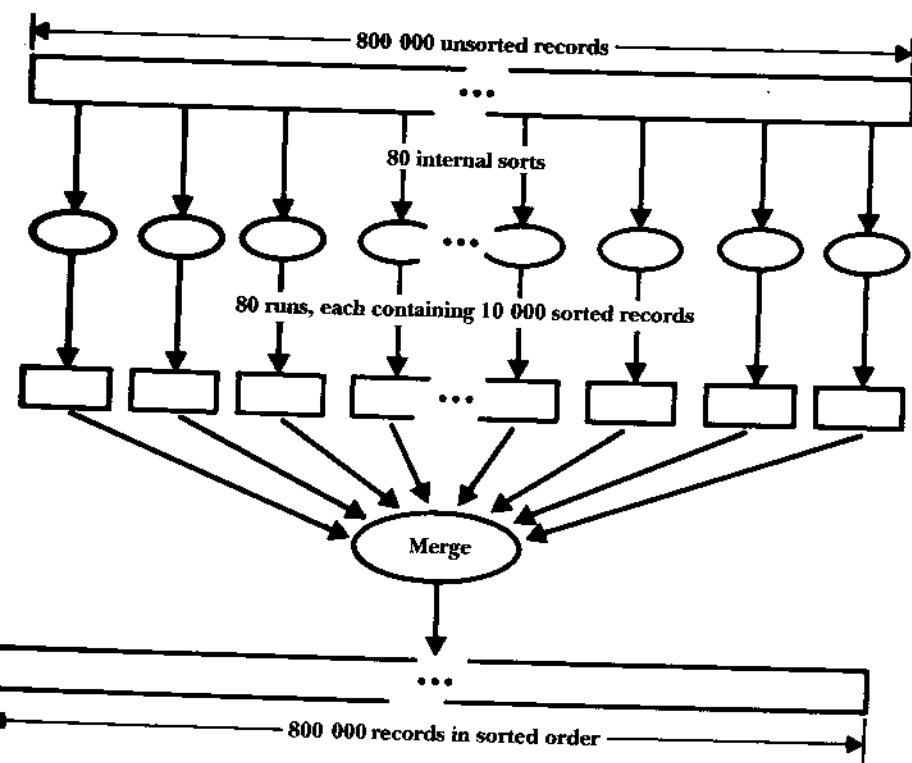
$$\frac{10\,000\,000 \text{ bytes of memory}}{100 \text{ bytes per record}} = 100\,000 \text{ records}$$

Once we create the first run, we then read a new set of records, once again filling memory, and create another run of 100 000 records. In our example, we repeat this process until we have created eighty runs, with each run containing 100 000 sorted records.

Once we have the eighty runs in eighty separate files on disk, we can perform an eighty-way merge of these runs, using the multiway merge logic outlined in Section 8.3, to create a completely sorted file containing all the original records. A schematic view of this run creation and merging process is provided in Fig. 8.21.

This solution to our sorting problem has the following features:

- It can, in fact, sort large files and can be extended to files of any size.
- Reading of the input file during the run-creation step is sequential and hence is much faster than input that requires seeking for every record individually (as in a keysort).
- Reading through each run during merging and writing the sorted records is also sequential. Random accesses are required only as we switch from run to run during the merge operation.
- If a heapsort is used for the in-memory part of the merge, as described in Section 8.4, we can overlap these operations with I/O so the in-memory part does not add appreciably to the total time for the merge.
- Since I/O is largely sequential, tapes can be used if necessary for both input and output operations.



**Figure 8.21** Sorting through the creation of runs (sorted subfiles) and subsequent merging of runs.

### 8.5.1 How Much Time Does a Merge Sort Take?

This general approach to the problem of sorting large files looks promising. To compare this approach with others, we now look at how much time it takes. We do this by taking our 8 million record files and seeing how long it takes to do a merge sort on the Seagate Cheetah 9 disk drive whose specifications are listed in Table 3.1. You might recall that this was the fastest disk available for PCs in early 1997. Please note that our intention here is not to derive time estimates that mean anything in any environment other than the hypothetical environment we have posited. Nor do we want to overwhelm you with numbers or provide you with magic formulas for determining how long a particular sort on a real system will *really* take. Rather, our goal in this section is to derive some benchmarks that we can use to compare several variations on the basic merge sort approach to sorting external files.



We can simplify matters by making the following simplifying assumptions about the computing environment:

- Entire files are always stored in contiguous areas on disk (extents), and a single cylinder-to-cylinder seek takes no time. Hence, *only one seek is required for any single sequential access.*
- Extents that span more than one track are physically staggered in such a way that *only one rotational delay is required per access.*

We see in Fig. 8.21 that there are four times when I/O is performed. During the sort phase:

- Reading all records into memory for sorting and forming runs, and
- Writing sorted runs to disk.

During the merge phase:

- Reading sorted runs into memory for merging, and
- Writing sorted file to disk.

Let's look at each of these in order.

### ***Step 1: Reading Records into Memory for Sorting and Forming Runs***

Since we sort the file in 10-megabyte chunks, we read 10 megabytes at a time from the file. In a sense, memory is a 10-megabyte input buffer that we fill up eighty times to form the eighty runs. In computing the total time to input each run, we need to include the amount of time it takes to *access* each block (seek time + rotational delay), plus the amount of time it takes to *transfer* each block. We keep these two times separate because, as we see later in our calculations, the role that each plays can vary significantly depending on the approach used.

From Table 3.1 we see that seek and rotational delay times are 8 msec<sup>1</sup> and 3 msec, respectively, so total time per seek is 11 msec.<sup>2</sup> The transmission rate is approximately 14 500 bytes per msec. Total input time for the sort phase consists of the time required for 80 seeks, plus the time required to transfer 800 megabytes:

- 
1. Unless the computing environment has many active users pulling the read/write head to other parts of the disk, seek time is likely to be less than the average, since many of the blocks that make up the file are probably going to be physically adjacent to one another on the disk. Many will be on the same cylinder, requiring no seeks at all. However, for simplicity we assume the average seek time.
  2. For simplicity, we use the term *seek* even though we really mean *seek and rotational delay*. Hence, the time we give for a seek is the time that it takes to perform an average seek followed by an average rotational delay.

Access :	$80 \text{ seeks} \times 11 \text{ msec} = 1 \text{ sec}$
Transfer :	$800 \text{ megabytes @ } 14\,500 \text{ bytes/msec} = 60 \text{ sec}$
Total :	61 sec

### Step 2: Writing Sorted Runs to Disk

In this case, writing is just the reverse of reading—the same number of seeks and the same amount of data to transfer. So it takes another 60 seconds to write the 80 sorted runs.

### Step 3: Reading Sorted Runs into Memory for Merging

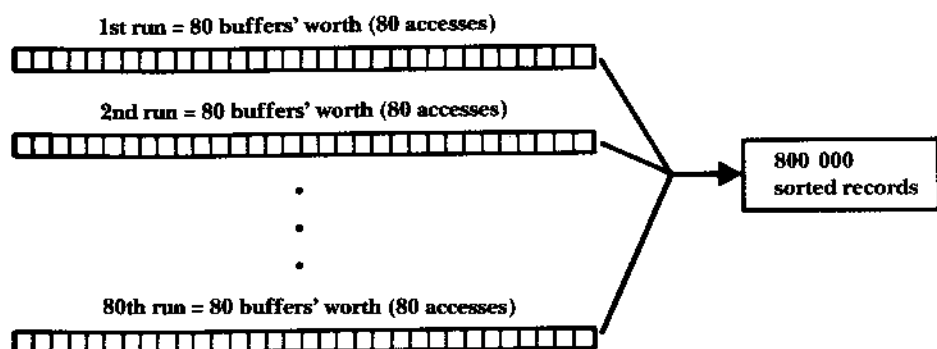
Since we have 10 megabytes of memory for storing runs, we divide 10 megabytes into 80 parts for buffering the 80 runs. In a sense, we are re-locating our 10 megabytes of memory as 80 input buffers. Each of the 80 buffers then holds 1/80th of a run (125 000 bytes), so we have to access *each* run 80 times to read all of it. Because there are 80 runs, in order to complete the merge operation (Fig. 8.22) we end up making

$$80 \text{ runs} \times 80 \text{ seeks} = 6400 \text{ seeks.}$$

Total seek and rotation time is then  $6400 \times 11 \text{ msec} = 70 \text{ seconds}$ . Since 800 megabytes is still transferred, transfer time is still 60 seconds.

### Step 4: Writing Sorted File to Disk

To compute the time for writing the file, we need to know how big our output buffers are. Unlike steps 1 and 2, where our big memory sorting



**Figure 8.22** Effect of buffering on the number of seeks required, where each run is as large as the available work area in memory.

space doubled as our I/O buffer, we are now using that memory space for storing the data from the runs *before* it is merged. To keep matters simple, let us assume that we can allocate two 200 000-byte output buffers.<sup>3</sup> With 200 000-byte buffers, we need to make

$$\frac{800\,000\,000}{200\,000 \text{ bytes per seek}} = 4000 \text{ seeks.}$$

Total seek and rotation time is then  $4000 \times 11 \text{ msec} = 44 \text{ seconds}$ . Transfer time is still 60 seconds.

The time estimates for the four steps are summarized in the first row in Table 8.1. The total time for this merge sort is 356 seconds, or 5 minutes, 56 seconds. The sort phase takes 122 seconds, and the merge phase takes 234 seconds.

To gain an appreciation of the improvement that this merge sort approach provides us, we need only look at how long it would take us to do one part of a nonmerging method like the keysort method described in Chapter 6. The last part of the keysort algorithm (Fig. 6.16) consists of this for loop:

```

/ write new file in key order
for (int j = 0; j < inFile . NumRecs(); j++)

    inFile . ReadByRRN (obj, Keys[j] . RRN); // read in key order
    outFile . Append (obj); // write in key order

```

3. We use two buffers to allow double buffering; we use 20 000 bytes per buffer because that is approximately the size of a track on our hypothetical disk drive.

**Table 8.1** Time estimates for merge sort of 80-megabyte file, assuming use of the megagate Cheetah 9 disk drive described in Table 3.1. The total time for the sort phase (steps 1 and 2) is 14 seconds, and the total time for the merge phase is 126 seconds.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Sort: reading	800	800	1	60	61
Sort: writing	800	800	1	60	61
Merge: reading	6400	800	70	60	130
Merge: writing	4000	800	44	60	104
Totals	10 560	3200	116	240	356

This *for* loop requires us to do a separate seek for every record in the file. That is 8 000 000 seeks. At 11 msec per seek, the total time required to perform that one operation works out to 88 000 seconds, or 24 hours, 26 minutes, 40 seconds!

Clearly, for large files the merge sort approach in general is the best option of any that we have seen. Does this mean that we have found the best technique for sorting large files? If sorting is a relatively rare event and files are not too large, the particular approach to merge sorting that we have just looked at produces acceptable results. Let's see how those results stand up as we change some of the parameters of our sorting example.

### 8.5.2 Sorting a File That Is Ten Times Larger

The first question that comes to mind when we ask about the general applicability of a computing technique is: What happens when we make the problem bigger? In this instance, we need to ask how this approach stands up as we scale up the size of the file.

Before we look at how a bigger file affects the performance of our merge sort, it will help to examine the *kinds* of I/O that are being done in the two different phases—the sort phase and the merge phase. We will see that for the purposes of finding ways to improve on our original approach, we need pay attention only to one of the two phases.

A major difference between the sort phase and the merge phase is in the amount of sequential (versus random) access that each performs. By using heapsort to create runs during the sort phase, we guarantee that all I/O is, in a sense, sequential.<sup>4</sup> Since sequential access implies minimal seeking, we cannot *algorithmically* speed up I/O during the sort phase. No matter what we do with the records in the file, we have to read them and write them all at least once. Since we cannot improve on this phase by changing the way we do the sort or merge, we ignore the sort phase in the analysis that follows.

The merge phase is a different matter. In particular, the *reading step* of the merge phase is different. Since there is a memory buffer for each run, and these buffers get loaded and reloaded at unpredictable times, the read step of the merge phase is, to a large extent, one in which random accesses

---

4. It is *not* sequential in the sense that in a multiuser environment there will be other users pulling the read/write head to other parts of the disk between reads and writes, possibly forcing the disk to do a seek each time it reads or writes a block.

are the norm. Furthermore, the number and size of the memory buffers that we read the run data into determine the number of times we have to do random accesses. If we can somehow reconfigure these buffers in ways that reduce the number of random accesses, we can speed up I/O correspondingly. So, if we are going to look for ways to improve performance in a merge sort algorithm, *our best hope is to look for ways to cut down on the number of random accesses that occur while reading runs during the merge phase.*

What about the write step of the merge phase? Like the steps of the sort phase, this step is not influenced by differences in the way we organize runs. Improvements in the way we organize the merge sort do not affect this step. On the other hand, we will see later that it is helpful to include this phase when we measure the results of changes in the organization of the merge sort.

To sum up, since the merge phase is the only one in which we can improve performance by improving the method, we concentrate on it from now on. Now let's get back to the question that we started this section with: What happens when we make the problem bigger? How, for instance, is the time for the merge phase affected if our file is 80 million records rather than 8 million?

If we increase the size of our file by a factor of 10 without increasing the memory space, we clearly need to create more runs. Instead of 80 initial 100 000-record runs, we now have 800 runs. This means we have to do an 800-way merge in our 10 megabytes of memory space. This, in turn, means that during the merge phase we must divide memory into 800 buffers. Each of the 800 buffers holds 1/800th of a run, so we would end up making 800 seeks per run, and

$$800 \text{ runs} \times 800 \text{ seeks/run} = 640\,000 \text{ seeks altogether}$$

The times for the merge phase are summarized in Table 8.2. Note that the total time is more than 2 hours and 24 minutes, almost 25 times greater than for the 800-megabyte file. By increasing the size of our file, we have gotten ourselves back into the situation we had with keysort, in which we can't do the job we need to do without doing a huge amount of seeking. In this instance, by increasing the order of the merge from 80 to 800, we made it necessary to divide our 10-megabyte memory area into 800 tiny buffers for doing I/O; and because the buffers are tiny, each requires many seeks to process its corresponding run.

If we want to improve performance, clearly we need to look for ways to improve on the amount of time spent getting to the data during the merge phase. We will do this shortly, but first let us generalize what we have just observed.

**Table 8.2** Time estimates for merge sort of 8000-megabyte file, assuming use of the Seagate Cheetah 9 disk drive described in Table 3.1. The total time for the merge phase is 7600 seconds, or 2 hours, 6 minutes, 40 seconds.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Merge: reading	640 000	8000	7040	600	7640
Merge: writing	40 000	8000	440	600	1040
Totals	680 000	16 000	7480	1200	8680

### 8.5.3 The Cost of Increasing the File Size

Obviously, the big difference between the time it took to merge the 800-megabyte file and the 8000-megabyte file was due to the difference in total seek and rotational delay times. You probably noticed that the number of seeks for the larger file is 100 times the number of seeks for the first file, and 100 is the square of the difference in size between the two files. We can formalize this relationship as follows: in general, for a  $K$ -way merge of  $K$  runs where each run is as large as the memory space available, the buffer size for each of the runs is

$$\left(\frac{1}{K}\right) \times \text{size of memory space} = \left(\frac{1}{K}\right) \times \text{size of each run}$$

so  $K$  seeks are required to read all of the records in each individual run. Since there are  $K$  runs altogether, the merge operation requires  $K^2$  seeks. Hence, measured in terms of seeks, our sort merge is an  $O(K^2)$  operation. Because  $K$  is directly proportional to  $N$  (if we increase the number of records from 8 000 000 to 80 000 000,  $K$  increases from 80 to 800) it also follows that our sort merge is an  $O(N^2)$  operation, measured in terms of seeks.

This brief, formal look establishes the principle that as files grow large, we can expect the time required for our merge sort to increase rapidly. It would be very nice if we could find some ways to reduce this time. Fortunately, there are several ways:

- Allocate more hardware, such as disk drives, memory, and I/O channels;

- Perform the merge in more than one step, reducing the order of each merge and increasing the buffer size for each run;
- Algorithmically increase the lengths of the initial sorted runs; and
- Find ways to overlap I/O operations.

In the following sections we look at each of these ways in detail, beginning with the first: invest in more hardware.

### 8.5.4 Hardware-Based Improvements

We have seen that changes in our sorting algorithm can improve performance. Likewise, we can make changes in our hardware that will also improve performance. In this section we look at three possible changes to a system configuration that could lead to substantial decreases in sort time:

- Increasing the amount of memory,
- Increasing the number of disk drives, and
- Increasing the number of I/O channels.

#### *Increasing the Amount of Memory*

It should be clear now that when we have to divide limited buffer space into many small buffers, we increase seek and rotation times to the point where they overwhelm all other sorting operations. Roughly speaking, the increase in the number of seeks is proportional to the square of the increase in file size, given a fixed amount of total buffer space.

It stands to reason, then, that increasing memory space ought to have a substantial effect on total sorting time. A larger memory size means longer and fewer initial runs during the sort phase, and it means fewer seeks per run during the merge phase. The product of fewer runs and fewer seeks per run means a substantial reduction in total seeks.

Let's test this conclusion with our 80 000 000-record file, which took about 2 hours, 6 minutes using 10 megabytes of memory. Suppose we are able to obtain 40 megabytes of memory buffer space for our sort. Each of the initial runs would increase from 100 000 records to 400 000 records, resulting in two hundred 400 000-record runs. For the merge phase, the internal buffer space would be divided into 200 buffers, each capable of holding 1/200th of a run, meaning that there would be  $200 \times 200 = 40\,000$  seeks. Using the same time estimates that we used for the previous two

cases, the total time for this merge is 16 minutes, 40 seconds, nearly a sevenfold improvement.

### ***Increasing the Number of Dedicated Disk Drives***

If we could have a separate read/write head for every run and no other users contending for use of the same read/write heads, there would be no delay due to seek time after the original runs are generated. The primary source of delay would now be rotational delays and transfers, which would occur every time a new block had to be read.

For example, if each run is on a separate, dedicated drive, our 800-way merge calls for only 800 seeks (one seek per run), down from 640 000, cutting the total seek and rotation times from 7040 seconds to 1 second. Of course, we can't configure 800 separate disk drives every time we want to do a sort, but perhaps something short of this is possible. For instance, if we had two disk drives to dedicate to the merge, we could assign one to input and the other to output, so reading and writing could overlap whenever they occurred simultaneously. (This approach takes some clever buffer management, however. We discuss this later in this chapter.)

### ***Increasing the Number of I/O Channels***

If there is only one I/O channel, no two transmissions can occur at the same time, and the total transmission time is the one we have computed. But if there is a separate I/O channel for each disk drive, I/O can overlap completely.

For example, if for our 800-way merge there are 800 channels from 800 disk drives, then transmissions can overlap completely. Practically speaking, it is unlikely that 800 channels and 800 disk drives are available, and even if they were, it is unlikely that all transmissions would overlap because all buffers would not need to be refilled at one time. Nevertheless, increasing the number of I/O channels could improve transmission time substantially.

So we see that there are ways to improve performance if we have some control over how our hardware is configured. In those environments in which external sorting occupies a large percentage of computing time, we are likely to have at least some such control. On the other hand, many times we are not able to expand a system specifically to meet sorting needs that we might have. When this is the case, we need to look for algorithmic ways to improve performance, and this is what we do now.



### 8.5.5 Decreasing the Number of Seeks Using Multiple-Step Merges

One of the hallmarks of a solution to a file structure problem, as opposed to the solution of a mere data structure problem, is the attention given to the enormous difference in cost between accessing information on disk and accessing information in memory. If our merging problem involved only memory operations, the relevant measure of work, or expense, would be the number of *comparisons* required to complete the merge. The *merge pattern* that would minimize the number of comparisons for our sample problem, in which we want to merge 800 runs, would be the 800-way merge considered. Looked at from a point of view that ignores the cost of seeking, this  $K$ -way merge has the following desirable characteristics:

- Each record is read only once.
- If a selection tree is used for the comparisons performed in the merging operation, as described in Section 8.3, then the number of comparisons required for a  $K$ -way merge of  $N$  records (total) is a function of  $N \times \log_2 K$ .
- Since  $K$  is directly proportional to  $N$ , this is an  $O(N \times \log_2 N)$  operation (measured in numbers of comparisons), which is to say that it is reasonably efficient even as  $N$  grows large.

This would all be very good news were we working exclusively in memory, but the very purpose of this *merge sort* procedure is to be able to sort files that are too large to fit into memory. Given the task at hand, the costs associated with disk seeks are orders of magnitude greater than the costs of operations in memory. Consequently, if we can sacrifice the advantages of an 800-way merge and trade them for savings in access time, we may be able to obtain a net gain in performance.

We have seen that one of the keys to reducing seeks is to reduce the number of runs that we have to merge, thereby giving each run a bigger share of available buffer space. In the previous section we accomplished this by adding more memory. Multiple-step merging provides a way for us to apply the same principle without having to buy more memory.

In multiple-step merging, we do not try to merge all runs at one time. Instead, we break the original set of runs into small groups and merge the runs in these groups separately. On each of these smaller merges, more buffer space is available for each run; hence, fewer seeks are required per run. When all of the smaller merges are completed, a second pass merges the new set of merged runs.

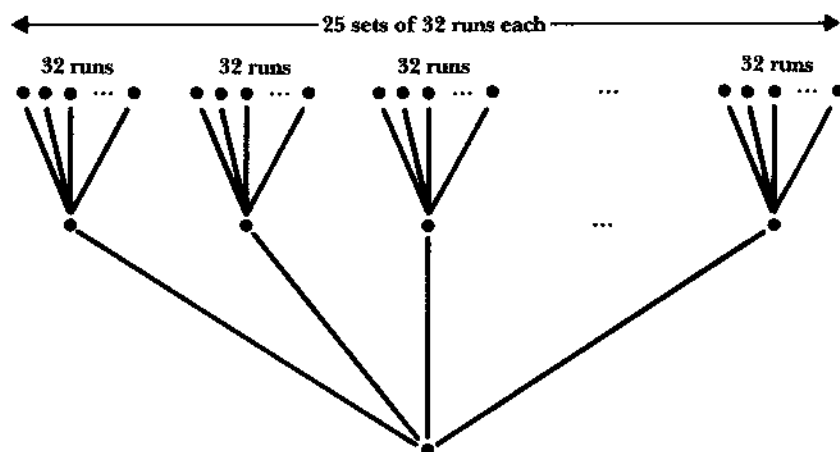
It should be clear that this approach will lead to fewer seeks on the first pass, but now there is a second pass. Not only are a number of seeks required for reading and writing on the second pass, but extra transmission time is used in reading and writing all records in the file. Do the advantages of the two-pass approach outweigh these extra costs? Let's revisit the merge step of our 80 million record sort to find out.

Recall that we began with 800 runs of 100 000 records each. Rather than merging all 800 runs at once, we could merge them as, say, 25 sets of 32 runs each, followed by a 25-way merge of the intermediate runs. This scheme is illustrated in Fig. 8.23.

When compared with our original 800-way merge, this approach has the disadvantage of requiring that we read every record twice: once to form the intermediate runs and again to form the final sorted file. But since each step of the merge is reading from 25 input files at a time, we are able to use larger buffers and avoid a large number of disk seeks. When we analyzed the seeking required for the 800-way merge, disregarding seeking for the output file, we calculated that the 800-way merge involved 640 000 seeks between the input files. Let's perform similar calculations for our multistep merge.

### *First Merge Step*

For each of the 32-way merges of the initial runs, each input buffer can hold 1/32 run, so we end up making  $32 \times 32 = 1024$  seeks. For all 25 of the



**Figure 8.23** Two-step merge of 800 runs.

32-way merges, we make  $25 \times 1024 = 25\,600$  seeks. Each of the resulting runs is 3 200 000 records, or 320 megabytes.

### Second Merge Step

For each of the 25 final runs, 1/25 of the total buffer space is allocated, so each input buffer can hold 4000 records, or 1/800 run. Hence, in this step there are 800 seeks per run, so we end up making  $25 \times 800 = 20\,000$  seeks, and

The total number of seeks for the two steps =  $25\,600 + 20\,000 = 45\,600$

So, by accepting the cost of processing each record twice, we reduce the number of seeks for reading from 640 000 to 45 600, and we haven't spent a penny for extra memory.

But what about the *total* time for the merge? We save on access times for inputting data, but there are costs. We now have to transmit all of the records four times instead of two, so transmission time increases by 1200 seconds. Also, we write the records twice, rather than once, requiring an extra 40 000 seeks. When we add these extra operations, the total time for the merge is 3782 seconds, or about 1 hour, 3 minutes, compared with 2 hours, 25 minutes for the single-step merge. These results are summarized in Table 8.3.

Once more, note that the essence of what we have done is to find a way to increase the available buffer space for each run. We trade extra passes over the data for a dramatic decrease in random accesses. In this case the trade is certainly a profitable one.

**Table 8.3** Time estimates for two-step merge sort of 8000-megabyte file, assuming use of the Seagate Cheetah 9 disk drive described in Table 3.1. The total time is 27 minutes.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
* Merge: reading	25 600	8000	282	600	882
* Merge: writing	40 000	8000	440	600	1040
nd Merge: reading	20 000	8000	220	600	820
nd Merge: writing	40 000	8000	440	600	1040
Totals	125 600	32 000	1382	2400	3782

If we can achieve such an improvement with a two-step merge, can we do even better with three steps? Perhaps, but it is important to note in Table 8.3 that we have reduced total seek and rotation times to the point where transmission times are more expensive. Since a three-step merge would require yet another pass over the file, we have reached a point of diminishing returns.

We also could have chosen to distribute our initial runs differently. How would the merge perform if we did 400 two-way merges, followed by one 400-way merge, for instance? A rigorous analysis of the trade-offs between seek and rotation time and transmission time, accounting for different buffer sizes, is beyond the scope of our treatment of the subject.<sup>5</sup> Our goal is simply to establish the importance of the interacting roles of the major costs in performing merge sorts: seek and rotation time, transmission time, buffer size, and number of runs. In the next section we focus on the pivotal role of the last of these—the number of runs.

### 8.5.6 Increasing Run Lengths Using Replacement Selection

What would happen if we could somehow increase the size of the initial runs? Consider, for example, our earlier sort of 80 000 000 records in which each record was 100 bytes. Our initial runs were limited to approximately 100 000 records because the memory work area was limited to 10 megabytes. Suppose we are somehow able to create runs of twice this length, containing 200 000 records each. Then, rather than needing to perform an 800-way merge, we need to do only a 400-way merge. The available memory is divided into 400 buffers, each holding 1/800th of a run. Hence, the number of seeks required per run is 800, and the total number of seeks is

$$800 \text{ seeks/run} \times 400 \text{ runs} = 320\,000 \text{ seeks,}$$

half the number required for the 800-way merge of 100 000-byte runs.

In general, if we can somehow increase the size of the initial runs, we decrease the amount of work required during the merge step of the sorting process. A longer initial run means fewer total runs, which means a lower-order merge, which means bigger buffers, which means fewer seeks. But how, short of buying twice as much memory for the computer, can we create initial runs that are twice as large as the number of records that we

5. For more rigorous and detailed analyses of these issues, consult the references cited at the end of this chapter, especially Knuth (1998) and Salzberg (1988, 1990).

can hold in memory? The answer, once again, involves sacrificing some efficiency in our in-memory operations in return for decreasing the amount of work to be done on disk. In particular, the answer involves the use of an algorithm known as *replacement selection*.

Replacement selection is based on the idea of always *selecting* the key from memory that has the lowest value, outputting that key, and then *replacing* it with a new key from the input list. Replacement selection can be implemented as follows:

1. Read a collection of records and sort them using heapsort. This creates a heap of sorted values. Call this heap the *primary heap*.
2. Instead of writing the entire primary heap in sorted order (as we do in a normal heapsort), write only the record whose key has the lowest value.
3. Bring in a new record and compare the value of its key with that of the key that has just been output.
  - a. If the new key value is higher, insert the new record into its proper place in the primary heap along with the other records that are being selected for output. (This makes the new record part of the run that is being created, which means that the run being formed will be larger than the number of keys that can be held in memory at one time.)
  - b. If the new record's key value is lower, place the record in a *secondary heap* of records with key values lower than those already written. (It cannot be put into the primary heap because it cannot be included in the run that is being created.)
4. Repeat step 3 as long as there are records left in the primary heap and there are records to be read. When the primary heap is empty, make the secondary heap into the primary heap, and repeat steps 2 and 3.

To see how this works, let's begin with a simple example, using an input list of only six keys and a memory work area that can hold only three keys. As Fig. 8.24 illustrates, we begin by reading into memory the three keys that fit there and use heapsort to sort them. We select the key with the minimum value, which happens to be 5 in this example, and output that key. We now have room in the heap for another key, so we read one from the input list. The new key, which has a value of 12, now becomes a member of the set of keys to be sorted into the output run. In fact, because it is smaller than the other keys in memory, 12 is the next key that is output. A new key is read into its place, and the process continues. When

**Input:**

21, 67, 12, 5, 47, 16

↑  
Front of input string

Remaining input	Memory ( $P = 3$ )	Output run
21, 67, 12	5    47    16	-
21, 67	12    47    16	5
21	67    47    16	12, 5
-	67    47    21	16, 12, 5
-	67    47    -	21, 16, 12, 5
-	67    -    -	47, 21, 16, 12, 5
-	-    -    -	67, 47, 21, 16, 12, 5

**Figure 8.24** Example of the principle underlying replacement selection.

the process is complete, it produces a sorted list of six keys while using only three memory locations.

In this example the entire file is created using only one heap, but what happens if the fourth key in the input list is 2 rather than 12? This key arrives in memory too late to be output into its proper position relative to the other keys: the 5 has already been written to the output list. Step 3b in the algorithm handles this case by placing such values in a second heap, to be included in the next run. Figure 8.25 illustrates how this process works. During the first run, when keys that are too small to be included in the primary heap are brought in, we mark them with parentheses, indicating that they have to be held for the second run.

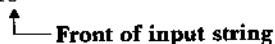
It is interesting to use this example to compare the action of replacement selection to the procedure we have been using up to this point, namely that of reading keys into memory, sorting them, and outputting a run that is the size of the memory space. In this example our input list contains thirteen keys. A series of successive memory sorts, given only three memory locations, results in five runs. The replacement selection procedure results in only two runs. Since the disk accesses during a multi-way merge can be a major expense, replacement selection's ability to create longer, and therefore fewer, runs can be an important advantage.

Two questions emerge at this point:

1. Given  $P$  locations in memory, how long a run can we expect replacement selection to produce, on the average?
2. What are the costs of using replacement selection?

**Input:**

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16


 Front of input string

Remaining input	Memory ( $P = 3$ )	Output run
33, 18, 24, 58, 14, 17, 7, 21, 67, 12	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17	67 47 ( 7 )	21, 16, 12, 5
33, 18, 24, 58, 14	67 (17) ( 7 )	47, 21, 16, 12, 5
33, 18, 24, 58	(14) (17) ( 7 )	67, 47, 21, 16, 12, 5
<b>First run complete; start building the second</b>		
33, 18, 24, 58	14 17 7	-
33, 18, 24	14 17 58	7
33, 18	24 17 58	14, 7
33	24 18 58	17, 14, 7
-	24 33 58	18, 17, 14, 7
-	- 33 58	24, 18, 17, 14, 7
-	- - 58	33, 24, 18, 17, 14, 7
-	-	58, 33, 24, 18, 17, 14, 7

**Figure 8.25** Step-by-step operation of replacement selection working to form two sorted runs.

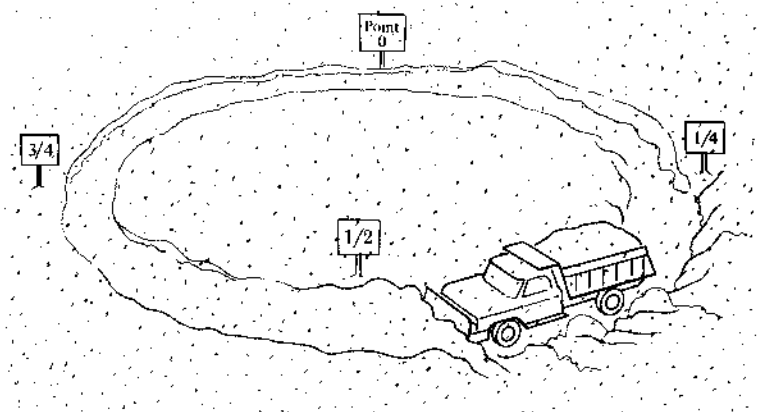
### Average Run Length for Replacement Selection

The answer to the first question is that, on the average, we can expect a run length of  $2P$ , given  $P$  memory locations. Knuth<sup>6</sup> provides an excellent description of an intuitive argument for why this is so:

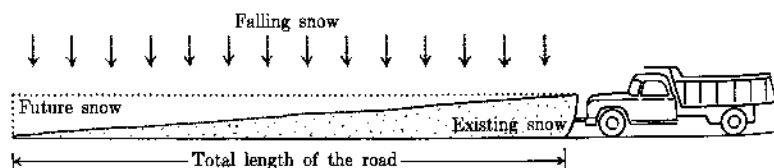
A clever way to show that  $2P$  is indeed the expected run length was discovered by E. F. Moore, who compared the situation to a snowplow on a circular track [U.S. Patent 2983904 (1961), Cols. 3-4]. Consider the situation shown [page 336]; flakes of snow are falling uniformly on a circular road, and a lone snowplow is continually clearing the snow. Once the snow has been plowed off the road, it disappears from the system. Points on the road may be designated by real numbers  $x$ ,  $0 \leq x < 1$ ; a flake of snow falling at position  $x$  represents an input record whose key is  $x$ ,

6. From Donald Knuth, *The Art of Computer Programming*, vol. 3 1973, Addison-Wesley, Reading, Mass. Pages 254-55 and Figs. 64 and 65. Reprinted with permission.

and the snowplow represents the output of replacement selection. The ground speed of the snowplow is inversely proportional to the height of the snow that it encounters, and the situation is perfectly balanced so that the total amount of snow on the road at all times is exactly  $P$ . A new run is formed in the output whenever the plow passes point 0.



After this system has been in operation for a while, it is intuitively clear that it will approach a stable situation in which the snowplow runs at constant speed (because of the circular symmetry of the track). This means that the snow is at constant height when it meets the plow, and the height drops off linearly in front of the plow as shown [below]. It follows that the volume of snow removed in one revolution (namely the run length) is twice the amount present at any one time (namely  $P$ ).



So, given a random ordering of keys, we can expect replacement selection to form runs that contain about twice as many records as we can hold in memory at one time. It follows that replacement selection creates half as many runs as a series of memory sorts of memory contents, assuming that the replacement selection and the memory sort have access to the same amount of memory. (As we see in a moment, the replacement selection does, in fact, have to make do with less memory than the memory sort.)



It is often possible to create runs that are substantially longer than  $2P$ . In many applications, the order of the records is *not* wholly random; the keys are often already partially in ascending order. In these cases replacement selection can produce runs that, on the average, exceed  $2P$ . (Consider what would happen if the input list is already sorted.) Replacement selection becomes an especially valuable tool for such partially ordered input files.

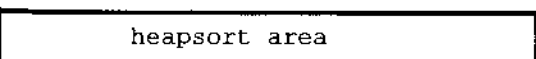
### *The Costs of Using Replacement Selection*

Unfortunately, the no-free-lunch rule applies to replacement selection, as it does to so many other areas of file structure design. In the worked-by-hand examples we have looked at up to this point, we have been inputting records into memory one at a time. We know, in fact, that the cost of seeking for every single input record is prohibitive. Instead, we want to buffer the input, which means, in turn, that we are not able to use *all* of the memory for the operation of replacement selection. Some of it has to be used for input and output buffering. This cost, and the affect it has on available space for sorting, is illustrated in Fig. 8.26.

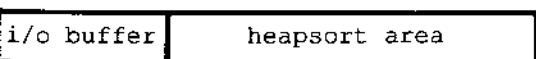
To see the effects of this need for buffering during the replacement selection step, let's return to our example in which we sort 80 000 000 records, given a memory area that can hold 100 000 records.

For the memory sorting methods such as heapsort, which simply read records into memory until it is full, we can perform sequential reads of 100 000 records at a time, until 800 runs have been created. This means that the sort step requires 1600 seeks: 800 for reading and 800 for writing.

For replacement selection we might use an input/output buffer that can hold, for example, 25 000 records, leaving enough space to hold 75 000 records for the replacement selection process. If the I/O buffer holds 2500 records, we can perform sequential reads of 25 000 records at a time, so it



(a) In-RAM sort: all available space used for the sort.



(b) Replacement selection: some of available space is used for I/O.

**Figure 8.26** In-memory sort versus replacement selection, in terms of their use of available memory for sorting operation.

takes  $80\,000\,000/25\,000 = 3200$  seeks to access all records in the file. This means that the sort step for replacement selection requires 6400 seeks: 3200 for reading and 3200 for writing.

If the records occur in a random key sequence, the average run length using replacement selection will be  $2 \times 75\,000 = 150\,000$  records, and there will be about  $80\,000\,000/150\,000 = 534$  such runs produced. For the merge step we divide the 10 megabytes of memory into 534 buffers, which hold an average of 187.3 records, so we end up making  $150\,000/187.3 = 801$  seeks per run, and

$$801 \text{ seeks per run} \times 534 \text{ runs} = 427\,734 \text{ seeks altogether}$$

Table 8.4 compares the access times required to sort the 80 million records using both a memory sort and replacement selection. The table includes our initial 800-way merge and two replacement selection examples. The second replacement selection example, which produces runs of 400 000 records while using only 75 000 record storage locations in memory, assumes that there is already a good deal of sequential ordering within the input records.

It is clear that, given randomly distributed input data, replacement selection can substantially reduce the number of runs formed. Even though replacement selection requires four times as many seeks to form the runs, the reduction in the amount of seeking effort required to merge the runs more than offsets the extra amount of seeking that is required to form the runs. And when the original data is assumed to possess enough order to make the runs 400 000 records long, replacement selection produces less than one-third as many seeks as memory sorting.

### 8.5.7 Replacement Selection Plus Multistep Merging

While these comparisons highlight the advantages of replacement selection over memory sorting, we would probably not in reality choose the one-step merge patterns shown in Table 8.4. We have seen that two-step merges can result in much better performance than one-step merges. Table 8.5 shows how these same three sorting schemes compare when two-step merges are used. From Table 8.5 (page 340) we see that the total number of seeks is dramatically less in every case than it was for the one-step merges. Clearly, the method used to form runs is not nearly as important as the use of multistep, rather than one-step, merges.

Furthermore, because the number of seeks required for the merge steps is much smaller in all cases, while the number of seeks required to

**Table 8.4** Comparison of access times required to sort 80 million records using both memory sort and replacement selection. Merge order is equal to the number of runs formed.

Approach	Number of records per seek to form runs	Size of runs formed	Number of runs formed	Number of seeks required to form runs	Merge order used	Total number of seeks	Total seek and rotational delay time	
							(hr)	(min)
800 memory sorts followed by an 800-way merge	100 000	100 000	800	1600	800	681 600	2	5
Replacement selection followed by 534-way merge (records in random order)	25 000	150 000	534	6400	534	521 134	1	36
Replacement selection followed by 200-way merge (records partially ordered)	25 000	400 000	200	6400	200	206 400	00	38

**Table 8.5** Comparison of access times required to sort 80 million records using both memory sort and replacement selection, each followed by a two-step merge.

Approach	Number of records per seek to form runs	Size of runs formed	Number of runs formed	Merge pattern used	Number of seeks in merge phases	Total number of seeks	Total seek and rotational delay time	
							(hr)	(min)
800 memory sorts	100 000	100 000	800	$25 \times 32$ -way then 25-way	25 600/20 000	127 200	0	24
Replacement selection (records in random order)	25 000	150 000	534	$19 \times 28$ -way then 19-way	22 876/15 162	124 438	0	23
Replacement selection (records partially ordered)	25 000	400 000	200	$20 \times 10$ -way then 20-way	8 000/16 000	110 400	0	20

form runs remains the same, the latter have a bigger effect *proportionally* on the final total, and the differences between the memory-sort based method and replacement selection are diminished.

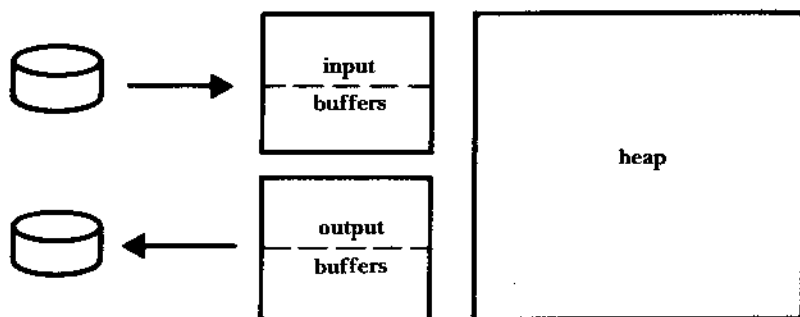
The differences between the one-step and two-step merges are exaggerated by the results in Table 8.5 because they don't take into account the amount of time spent transmitting the data. The two-step merges require that we transfer the data between memory and disk two more times than the one-step merges. Table 8.6 shows the results after adding transmission time to our results. The two-step merges are still better, and replacement selection still wins, but the results are less dramatic.

### 8.5.8 Using Two Disk Drives with Replacement Selection

Interestingly, and fortunately, replacement selection offers an opportunity to save on both transmission and seek times in ways that memory sort methods do not. As usual, this is at a cost, but if sorting time is expensive, it could well be worth the cost.

Suppose we have two disk drives to which we can assign the separate dedicated tasks of reading and writing during replacement selection. One drive, which contains the original file, does only input, and the other does only output. This has two very nice results: (1) it means that input and output can overlap, reducing transmission time by as much as 50 percent; and (2) seeking is virtually eliminated.

If we have two disks at our disposal, we should also configure memory to take advantage of them. We configure memory as follows: we allocate two buffers each for input and output, permitting double buffering, and allocate the rest of memory for forming the selection tree. This arrangement is illustrated in Fig. 8.27.



**Figure 8.27** Memory organization for replacement selection.

**Table 8.6** Comparison of sort merges illustrated in Tables 8.4 and 8.5, taking transmission times into account.

Approach	Number of records per seek to form runs	Merge pattern used	Number of seeks for sorts and merges	Seek + rotational delay time (min)	Total passes over the file	Total transmission time (min)	Total of seek, rotation, and transmission time (min)
800 memory sorts followed by an 800-way merge	100 000	800-way	681/700	125	4	40	165
Replacement selection followed by a 534-way merge (records in random order)	25 000	534-way	521/134	96	4	40	136
Replacement selection followed by a 200-way merge (records partially ordered)	25 000	200-way	206/400	38	4	40	78
800 memory sorts followed by a two-step merge	100 000	25 × 32-way one 25-way	127/200	23	6	60	83
Replacement selection followed by a two-step merge (records in random order)	25 000	19 × 28-way one 19-way	124/438	23	6	60	83
Replacement selection followed by a two-step merge (records partially ordered)	25 000	20 × 10-way one 20-way	110/400	20	6	60	80

Let's see how the merge sort process might proceed to take advantage of this configuration.

First, the sort phase. We begin by reading enough records to fill up the heap-sized part of memory and form the heap. Next, as we move records from the heap into one of the output buffers, we replace those records with records from one of the input buffers, adjusting the tree in the usual manner. While we empty one input buffer into the tree, we can be filling the other one from the input disk. This permits processing and input to overlap. Similarly, at the same time that we are filling one of the output buffers from the tree, we can transmit the contents of the other to the output disk. In this way, run selection and output can overlap.

During the merge phase, the output disk becomes the input disk, and vice versa. Since the runs are all on the same disk, seeking will occur on the input disk. But output is still sequential, since it goes to a dedicated drive.

Because of the overlapping of so many parts of this procedure, it is difficult to estimate the amount of time the procedure is likely to take. But it should be clear that by substantially reducing seeking and transmission time, we are attacking those parts of the sort merge that are the most costly.

### 8.5.9 More Drives? More Processors?

If two drives can improve performance, why not three, or four, or more? Isn't it true that the more drives we have to hold runs during the merge phase, the faster we can perform I/O? Up to a point this is true, but of course the number and speed of I/O processors must be sufficient to keep up with the data streaming in and out. And there will also be a point at which I/O becomes so fast that processing can't keep up with it.

But who is to say that we can use only one processor? A decade ago, it would have been farfetched to imagine doing sorting with more than one processor, but now it is very common to be able to dedicate more than one processor to a single job. Possibilities include the following:

- Mainframe computers, many of which spend a great deal of their time sorting, commonly come with two or more processors that can simultaneously work on different parts of the same problem.
- Vector and array processors can be programmed to execute certain kinds of algorithm orders of magnitude faster than scalar processors.
- Massively parallel machines provide thousands, even millions, of processors that can operate independently and at the same time communicate in complex ways with one another.

- Very fast local area networks and communication software make it relatively easy to parcel out different parts of the same process among several different machines.

It is not appropriate, in this text, to cover in detail the implications of these newer architectures for external sorting. But just as the changes of the past decade in the availability and performance of memory and disk storage have altered the way we look at external sorting, we can expect to change many more times as the current generation of new architectures becomes commonplace.

### 8.5.10 Effects of Multiprogramming

In our discussions of external sorting on disk we are, of course, making tacit assumptions about the computing environment in which this merging is taking place. We are assuming, for example, that the merge job is running in a dedicated environment (no multiprogramming). If, in fact, the operating system is multiprogrammed, as it normally is, the total time for the I/O might be longer, as our job waits for other jobs to perform their I/O.

On the other hand, one of the reasons for multiprogramming is to allow the operating system to find ways to increase the efficiency of the overall system by overlapping processing and I/O among different jobs. While the system could be performing I/O for our job while it is doing CPU processing on others, and vice versa, diminishing any delays caused by overlap of I/O and CPU processing within our job.

Effects such as these are hard to predict, even when you have much information about your system. Only experimentation can determine what real performance will be like on a busy, multiuser system.

### 8.5.11 A Conceptual Toolkit for External Sorting

We can now list many tools that can improve external sorting performance. It should be our goal to add these various tools to our conceptual toolkit for designing external sorts and to pull them out and use them whenever they are appropriate. A full listing of our new set of tools would include the following:

- For in-memory sorting, use heapsort for forming the original list of sorted elements in a run. With it and double buffering, we can overlap input and output with internal processing.



- Use as much memory as possible. It makes the runs longer and provides bigger and/or more buffers during the merge phase.
- If the number of initial runs is so large that total seek and rotation time is much greater than total transmission time, use a multistep merge. It increases the amount of transmission time but can decrease the number of seeks enormously.
- Consider using replacement selection for initial run formation, especially if there is a possibility that the runs will be partially ordered.
- Use more than one disk drive and I/O channel so reading and writing can overlap. This is especially true if there are no other users on the system.
- Keep in mind the fundamental elements of external sorting and their relative costs, and look for ways to take advantage of new architectures and systems, such as parallel processing and high-speed local area networks.

## 8.6 Sorting Files on Tape

---

There was a time when it was usually faster to perform large external sorts on tape than on disk, but this is much less the case now. Nevertheless, tape is still used in external sorting, and we would be remiss if we did not consider sort merge algorithms designed for tape.

There are a large number of approaches to sorting files on tape. After approximately one hundred pages of closely reasoned discussion of different alternatives for tape sorting, Knuth (1998) summarizes his analysis in the following way:

**Theorem A.** It is difficult to decide which merge pattern is best in a given situation.

Because of the complexity and number of alternative approaches and because of the way that these alternatives depend so closely on the specific characteristics of the hardware at a particular computer installation, our objective here is merely to communicate some of the fundamental issues associated with tape sorting and merging. For a more comprehensive discussion of specific alternatives, we recommend the work of Knuth (1998) as a starting point.

From a general perspective, the steps involved in sorting on tape resemble those we discussed with regard to sorting on disk:

1. Distribute the unsorted file into sorted *runs*, and
2. Merge the runs into a single sorted file.

Replacement selection is almost always a good choice as a method for creating the initial runs during a tape sort. You will remember that the problem with replacement selection when we are working on disk is that the amount of seeking required during run creation more than offsets the advantage of creating longer runs. This seeking problem disappears when the input is from tape. So, for a tape-to-tape sort, it is almost always advisable to take advantage of the longer runs created by replacement selection.

### 8.6.1 The Balanced Merge

Given that the question of how to create the initial runs has such a straightforward answer, it is clear that it is in the merging process that we encounter all of the choices and complexities implied by Knuth's tongue-in-cheek theorem. These choices include the question of how to *distribute* the initial runs on tape and questions about the process of merging from this initial distribution. Let's look at some examples to show what we mean.

Suppose we have a file that, after the sort phase, has been divided into ten runs. We look at a number of different methods for merging these runs on tape, assuming that our computer system has four tape drives. Since the initial, unsorted file is read from one of the drives, we have the choice of initially distributing the ten runs on two or three of the other drives. We begin with a method called *two-way balanced merging*, which requires that the initial distribution be on two drives and that at each step of the merge except the last, the output be distributed on two drives. Balanced merging is the simplest tape merging algorithm that we look at; it is also, as you will see, the slowest.

The balanced merge proceeds according to the pattern illustrated in Fig. 8.28.

This balanced merge process is expressed in an alternate, more compact form in Fig. 8.29 (page 348). The numbers inside the table are the run lengths measured in terms of the number of initial runs included in each merged run. For example, in step 1, all the input runs consist of a single initial run. By step 2, the input runs each consist of a pair of initial runs. At the start of step 3, tape drive T1 contains one run consisting of four initial runs followed by a run consisting of two initial runs. This method of illustration more clearly shows the way some of the intermedi-

	Tape	Contains runs				
Step 1	T1	R1	R3	R5	R7	R9
	T2	R2	R4	R6	R8	R10
	T3	—				
	T4	—				
Step 2	T1	—				
	T2	—				
	T3	R1-R2	R5-R6	R9-R10		
	T4	R3-R4	R7-R8			
Step 3	T1	R1-R4	R9-R10			
	T2	R5-R8				
	T3	—				
	T4	—				
Step 4	T1	—				
	T2	—				
	T3	R1-R8				
	T4	R9-R10				
Step 5	T1	R1-R10				
	T2	—				
	T3	—				
	T4	—				

**Figure 8.28** Balanced four-tape merge of ten runs.

ate runs combine and grow into runs of lengths 2, 4, and 8, whereas the one run that is copied again and again stays at length 2 until the end. The form used in this illustration is used throughout the following discussions on tape merging.

Since there is no seeking, the cost associated with balanced merging on tape is measured in terms of how much time is spent transmitting the data. In the example, we passed over all of the data four times during the merge phase. In general, given some number of initial runs, how many passes over the data will a two-way balanced merge take? That is, if we start with  $N$  runs, how many passes are required to reduce the number of runs to 1? Since each step combines two runs, the number of runs after each

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	Merge ten runs
Step 2	—	—	2 2 2	2 2	Merge ten runs
Step 3	4 2	4	—	—	Merge ten runs
Step 4	—	—	8	2	Merge ten runs
Step 5	10	—	—	—	

**Figure 8.29** Balanced four-tape merge of ten runs expressed in a more compact table notation.

step is half the number for the previous step. If  $p$  is the number of passes, then we can express this relationship as

$$(\frac{1}{2})^p \cdot N \leq 1$$

from which it can be shown that

$$p = \lceil \log_2 N \rceil$$

In our simple example,  $N = 10$ , so four passes over the data were required. Recall that for our partially sorted 800-megabyte file there were 200 runs, so  $\lceil \log_2 200 \rceil = 8$  passes are required for a balanced merge. If reading and writing overlap perfectly, each pass takes about 11 minutes,<sup>7</sup> so the total time is 1 hour, 28 minutes. This time is not competitive with our disk-based merges, even when a single disk drive is used. The transmission times far outweigh the savings in seek times.

## 8.6.2 The $K$ -way Balanced Merge

If we want to improve on this approach, it is clear that we must find ways to reduce the number of passes over the data. A quick look at the formula tells us that we can reduce the number of passes by increasing the order of each merge. Suppose, for instance, that we have 20 tape drives, 10 for input:

7. This assumes the 6250 bpi tape used in the examples in Chapter 3. If the transport speed is 200 inches per second, the transmission rate is 1250 kilobytes per second, assuming no blocking. At this rate an 800-megabyte file takes 640 seconds, or 10 minutes 40 seconds to read.

and 10 for output, at each step. Since each step combines 10 runs, the number of runs after each step is one-tenth the number for the previous step. Hence, we have

$$(\frac{1}{10})^p \cdot N \leq 1$$

and

$$p = \lceil \log_{10} N \rceil$$

In general, a *k-way balanced merge* is one in which the order of the merge at each step (except possibly the last) is *k*. Hence, the number of passes required for a *k-way balanced merge* with *N* initial runs is

$$p = \lceil \log_k N \rceil$$

For a 10-way balanced merge of our 800-megabyte file with 200 runs,  $\log_{10} 200 = 3$ , so three passes are required. The best estimated time now is reduced to a more respectable 42 minutes. Of course, the cost is quite high: we must keep 20 working tape drives on hand for the merge.

### 8.6.3 Multiphase Merges

The balanced merging algorithm has the advantage of being very simple; it is easy to write a program to perform this algorithm. Unfortunately, one reason it is simple is that it is “dumb” and cannot take advantage of opportunities to save work. Let’s see how we can improve on it.

We can begin by noting that when we merge the extra run with empty runs in steps 3 and 4, we don’t really accomplish anything. Figure 8.30 shows how we can dramatically reduce the amount of work that has to be done by simply not copying the extra run during step 3. Instead of merging this run with a dummy run, we simply stop tape T3 where it is. Tapes T1 and T2 now each contain a single run made up of four of the initial runs. We rewind all the tapes but T3 and then perform a three-way merge of the runs on tapes T1, T2, and T3, writing the final result on T4. Adding this intelligence to the merging procedure reduces the number of initial runs that must be read and written from forty down to twenty-eight.

The example in Fig. 8.30 clearly indicates that there are ways to improve on the performance of balanced merging. It is important to be able to state, in general terms, what it is about this second merging pattern that saves work:

- We use a higher-order merge. In place of two two-way merges, we use one three-way merge.

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	Merge ten runs
Step 2	—	—	2 2 2	2 2	Merge eight runs
Step 3	4	4	. . 2	—	Merge ten runs
Step 4	—	—	—	10	

**Figure 8.30** Modification of balanced four-tape merge that does not rewrite between steps 2 and 3 to avoid copying runs.

- We extend the merging of runs from one tape over several steps. Specifically, we merge some of the runs from T3 in step 3 and some in step 4. We could say that we merge the runs from T3 in two *phases*.

These ideas, the use of higher-order merge patterns and the merging of runs from a tape in *phases*, are the basis for two well-known approaches to merging called *polyphase merging* and *cascade merging*. In general, these merges share the following characteristics:

- The initial distribution of runs is such that at least the initial merge is a  $J-1$ -way merge, where  $J$  is the number of available tape drives.
- The distribution of the runs across the tapes is such that the tapes often contain different numbers of runs.

Figure 8.31 illustrates how a polyphase merge can be used to merge ten runs distributed on four tape drives. This merge pattern reduces the number of initial runs that must be read and written from forty (for a balanced two-way merge) to twenty-five. It is easy to see that this reduction is a consequence of the use of several three-way merges in place of two-way merges. It should also be clear that the ability to do these operations as three-way merges is related to the uneven nature of the initial distribution. Consider, for example, what happens if the initial distribution of runs is 4–3–3 rather than 5–3–2. We can perform three three-way merges to open up space on T3, but this also clears all the runs off of T1 and leaves only a single run on T1. Obviously, we are not able to perform another three-way merge as a second step.

Several questions arise at this point:

1. How does one choose an initial distribution that leads readily to an efficient merge pattern?

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1	1 1	—	
Step 2	. . 1 1 1	. . 1	—	3 3	Merge six runs
Step 3	. . . 1 1	—	5	. 3	Merge five runs
Step 4	. . . . 1	4	5	—	Merge four runs
Step 5	—	—	—	10	Merge ten runs

**Figure 8.31** Polyphase four-tape merge of ten runs.

- Are there algorithmic descriptions of the merge patterns, given an initial distribution?
- Given  $N$  runs and  $J$  tape drives, is there some way to compute the *optimal* merging performance so we have a yardstick against which to compare the performance of any specific algorithm?

Precise answers to these questions are beyond the scope of this text; in particular, the answer to the last question requires a more mathematical approach to the problem than the one we have taken here. Readers wanting more than an intuitive understanding of how to set up initial distributions should consult Knuth (1998).

### 8.6.4 Tapes versus Disks for External Sorting

A decade ago 1 megabyte of memory was considered a substantial amount of memory to allocate to any single job, and extra disk drives were very costly. This meant that many of the disk sorting techniques to decrease seeking that we have seen were not available to us or were very limited.

Suppose, for instance, that we want to sort our 8000-megabyte file and there is only 1 megabyte of memory available instead of 10 megabytes. The approach that we used for allocating memory for replacement selection would provide 250 kilobytes for buffering and 750 kilobytes for our selection tree. From this we can expect 5334 runs of 15 000 records each, versus 534 when there is a megabyte of memory. For a one-step merge, this tenfold increase in the number of runs results in a hundredfold increase in the number of seeks. What took three hours with 10 megabytes of memory now takes three hundred hours, just for the seeks! No wonder tapes, which are basically sequential and require no seeking, were preferred.

But now memory is much more readily available. Runs can be longer and fewer, and seeks are much less of a problem. Transmission time is now more important. The best way to decrease transmission time is to reduce the number of passes over the data, and we can do this by increasing the order of the merge. Since disks are random-access devices, very large-order merges can be performed, even if there is only one drive. Tapes, however, are not random-access devices; we need an extra tape drive for every extra run we want to merge. Unless a large number of drives is available, we can perform only low-order merges, and that means large numbers of passes over the data. Disks are better.

---

## 8.7 Sort-Merge Packages

---

Many good utility programs are available for users who need to sort large files. Often the programs have enough intelligence to choose from one of several strategies, depending on the nature of the data to be sorted and the available system configuration. They also often allow users to exert some control (if they want it) over the organization of data and strategies used. Consequently, even if you are using a commercial sort package rather than designing your own sorting procedure, it helps to be familiar with the variety of different ways to design merge sorts. It is especially important to have a good general understanding of the most important factors and trade-offs influencing performance.

---

## 8.8 Sorting and Cosequential Processing in Unix

---

Unix has a number of utilities for performing cosequential processing. It also has sorting routines, but nothing at the level of sophistication that you find in production sort-merge packages. In the following discussion we introduce some of these utilities. For full details, consult the Unix documentation.

### 8.8.1 Sorting and Merging in Unix

Because Unix is not an environment in which one expects to do frequent sorting of large files of the type we discuss in this chapter, sophisticated



sort-merge packages are not generally available on Unix systems. Still, the sort routines you find in Unix are quick and flexible and quite adequate for the types of applications that are common in a Unix environment. We can divide Unix sorting into two categories: (1) the `sort` command, and (2) callable sorting routines.

### **The Unix `sort` Command**

The `sort` command has many different options, but the simplest one is to sort the lines in an ASCII file in ascending lexical order. (A line is any sequence of characters ending with the new-line character `.`.) By default, the `sort` utility takes its input file name from the command line and writes the sorted file to standard output. If the file to be sorted is too large to fit in memory, `sort` performs a merge sort. If more than one file is named on the input line, `sort` sorts and merges the files.

As a simple example, suppose we have an ASCII file called `team` with names of members of a basketball team, together with their classes and their scoring averages:

```
Jean Smith Senior 8.8
Chris Mason Junior 9.6
Pat Jones Junior 3.2
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

To sort the file, enter

```
$ sort team
Chris Mason Junior 9.6
Jean Smith Senior 8.8
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
Pat Jones Junior 3.2
```

Notice that by default `sort` considers an entire line as the sort key. Hence, of the two players named Pat Jones, the freshman occurs first in the output because “Freshman” is lexically smaller than “Junior.” The assumption that the key is an entire line can be overridden by sorting on specified key fields. For `sort` a key field is assumed to be any sequence of characters delimited by spaces or tabs. You can indicate which key fields to use for sorting by giving their positions:

```
+pos1 [-pos2]
```

where `pos1` tells how many fields to skip before starting the key, and `pos2` tells which field to end with. If `pos2` is omitted, the key extends to the end of the line. Hence, entering

```
$ sort +1 -2 team
```

causes the file *team* to be sorted according to last names. (There is also a form of `pos1` and `pos2` that allows you to specify the character within a field to start a key with.)

The following options, among others, allow you to override the default ASCII ordering used by `sort`:

- d Use “dictionary” ordering: only letters, digits, and blanks are significant in comparisons.
- f “Fold” lowercase letters into uppercase. (This is the canonical form that we defined in Chapter 4.)
- r “Reverse” the sense of comparison: sort in descending ASCII order.

Notice that `sort` sorts lines, and within lines it compares groups of characters delimited by white space. In the language of Chapter 4, records are lines, and fields are groups of characters delimited by white space. This is consistent with the most common Unix view of fields and records within Unix text files.

### *The `qsort` Library Routine*

The Unix library routine `qsort` is a general sorting routine. Given a table of data, `qsort` sorts the elements in the table in place. A table could be the contents of a file, loaded into memory, where the elements of the table are its records. In C, `qsort` is defined as follows:

```
qsort(char *base, int nel, int width, int (*compar())) ;
```

The argument `base` is a pointer to the base of the data, `nel` is the number of elements in the table, and `width` is the size of each element. The last argument, `compar()`, is the name of a user-supplied comparison function that `qsort` uses to compare keys. `compar` must have two parameters that are pointers to elements that are to be compared. When `qsort` needs to compare two elements, it passes to `compar` pointers to these elements, and `compar` compares them, returning an integer that is less than, equal to, or greater than zero, depending on whether the first argument is considered less than, equal to, or greater than the second argument. A full explanation of how to

use `qsort` is beyond the scope of this text. Consult the Unix documentation for details.

### 8.8.2 Cosequential Processing Utilities in Unix

Unix provides a number of utilities for cosequential processing. The `sort` utility, when used to merge files, is one example. In this section we introduce three others: `diff`, `cmp`, and `comm`.

#### *cmp*

Suppose you find in your computer that you have two team files, one called `team` and the other called `myteam`. You think that the two files are the same, but you are not sure. You can use the command `cmp` to find out.

`cmp` compares two files. If they differ, it prints the byte and line number where they differ; otherwise it does nothing. If all of one file is identical to the first part of another, it reports that end-of-file was reached on the shorter file before any differences were found.

For example, suppose the file `team` and `myteam` have the following contents:

<i>team</i>	<i>myteam</i>
Jean Smith Senior 8.8	Jean Smith Senior 8.8
Chris Mason Junior 9.6	Stacy Fox Senior 1.6
Pat Jones Junior 3.2	Chris Mason Junior 9.6
Leslie Brown Sophomore 18.2	Pat Jones Junior 5.2
Pat Jones Freshman 11.4	Leslie Brown Sophomore 18.2
	Pat Jones Freshman 11.4

`cmp` tells you where they differ:

```
$ cmp team myteam
team myteam differ: char 23 line 2
```

Since `cmp` simply compares files on a byte-by-byte basis until it finds a difference, it makes no assumptions about fields or records. It works with both text and nontext files.

#### *diff*

`cmp` is useful if you want to know if two files are different, but it doesn't tell you much about how they differ. The command `diff` gives fuller

information. `diff` tells which lines must be changed in two files to bring them into agreement. For example:

```
$ diff team myteam
1a2
> Stacy Fox Senior 1.6
3c4
< Pat Jones Junior 3.2
---
> Pat Jones Junior 5.2
```

The `1a2` indicates that after line 1 in the first file, we need to *add* line 2 from the second file to make them agree. This is followed by the line from the second file that would need to be added. The `3c4` indicates that we need to *change* line 3 in the first file to make it look like line 4 in the second file. This is followed by a listing of the two differing lines, where the leading `<` indicates that the line is from the first file, and the `>` indicates that it is from the second file.

One other indicator that could appear in `diff` output is `d`, meaning that a line in the first file has been *deleted* in the second file. For example, `12d15` means that line 12 in the first file appears to have been deleted from being right after line 15 in the second file. Notice that `diff`, like `sort`, is designed to work with lines of text. It would not work well with non-ASCII text files.

### **`comm`**

Whereas `diff` tells what is different about two files, `comm` compares two files, which must be ordered in ASCII collating sequence, to see what they have in common. The syntax for `comm` is the following:

```
comm [-123] file1 file2
```

`comm` produces three columns of output. Column 1 lists the lines that are in `file1` only; column 2 lists lines in `file2` only, and column 3 lists lines that are in both files. For example,

```
$ sort team > ts
$ sort myteam > ms
$ comm ts ms
Chris Mason Junior 9.6
Jean Smith Senior 8.8
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

Pat Jones Junior 3.2

Pat Jones Junior 5.2

Stacy Fox Senior 1.6

Selecting any of the flags 1, 2, or 3 allows you to print only those columns you are interested in.

The `sort`, `diff`, `comm`, and `cmp` commands (and the `qsort` function) are representative of what is available in Unix for sorting and cosequential processing. As we have said, they have many useful options that we don't cover that you will be interested in reading about.

## SUMMARY

In the first half of this chapter, we develop a cosequential processing model and apply it to two common problems—updating a general ledger and merge sorting. The model is presented as a class hierarchy, using virtual methods to tailor the model to particular types of lists. In the second half of the chapter we identify the most important factors influencing performance in merge-sorting operations and suggest some strategies for achieving good performance.

The cosequential processing model can be applied to problems that involve operations such as matching and merging (and combinations of these) on two or more sorted input files. We begin the chapter by illustrating the use of the model to perform a simple match of the elements common to two lists and a merge of two lists. The procedures we develop to perform these two operations embody all the basic elements of the model.

In its most complete form, the model depends on certain assumptions about the data in the input files. We enumerate these assumptions in our formal description of the model. Given these assumptions, we can describe the processing components of the model and define pure virtual functions that represent those components.

The real value of the cosequential model is that it can be adapted to more substantial problems than simple matches or merges by extending the class hierarchy. We illustrate this by using the model to design a general ledger accounting program.

All of our early sample applications of the model involve only two input files. We next adapt the model to a multiway merge to show how the model might be extended to deal with more than two input lists. The problem of finding the minimum key value during each pass through the

main loop becomes more complex as the number of input files increases. Its solution involves replacing the three-way selection statement with either a multiway selection or a procedure that keeps current keys in a list structure that can be processed more conveniently.

We see that the application of the model to  $k$ -way merging performs well for small values of  $k$ , but that for values of  $k$  greater than 8 or so, it is more efficient to find the minimum key value by means of a selection tree.

After discussing multiway merging, we shift our attention to a problem that we encountered in a previous chapter—how to sort large files. We begin with files that are small enough to fit into memory and introduce an efficient sorting algorithm, *heapsort*, which makes it possible to overlap I/O with the sorting process.

The generally accepted solution when a file is too large for in-memory sorts is some form of *merge sort*. A merge sort involves two steps:

1. Break the file into two or more sorted subfiles, or runs, using internal sorting methods; and
2. Merge the runs.

Ideally, we would like to keep every run in a separate file so we can perform the merge step with one pass through the runs. Unfortunately, practical considerations sometimes make it difficult to do this effectively.

The critical elements when merging many files on disk are seek and rotational delay times and transmission times. These times depend largely on two interrelated factors: the number of different runs being merged and the amount of internal buffer space available to hold parts of the runs. We can reduce seek and rotational delay times in two ways:

- By performing the merge in more than one step; and/or
- By increasing the sizes of the initial sorted runs.

In both cases, the order of each merge step can be reduced, increasing the sizes of the internal buffers and allowing more data to be processed per seek.

Looking at the first alternative, we see how performing the merge in several steps can decrease the number of seeks dramatically, though it also means that we need to read through the data more than once (increasing total data transmission time).

The second alternative is realized through use of an algorithm called *replacement selection*. Replacement selection, which can be implemented using the selection tree mentioned earlier, involves selecting from memory the key that has the lowest value, outputting that key, and replacing it with a new key from the input list.

With randomly organized files, replacement selection can be expected to produce runs twice as long as the number of internal storage locations available for performing the algorithms. Although this represents a major step toward decreasing the number of runs that need to be merged, it carries an additional cost. The need for a large buffer for performing the replacement selection operation leaves relatively little space for the I/O buffer, which means that many more seeks are involved in forming the runs than are needed when the sort step uses an in-memory sort. If we compare the total number of seeks required by the two different approaches, we find that replacement selection can require more seeks; it performs substantially better only when there is a great deal of order in the initial file.

Next we turn our attention to file sorting on tapes. Since file I/O with tapes does not involve seeking, the problems and solutions associated with tape sorting can differ from those associated with disk sorting, although the fundamental goal of working with fewer, longer runs remains. With tape sorting, the primary measure of performance is the number of times each record must be transmitted. (Other factors, such as tape rewind time, can also be important, but we do not consider them here.)

Since tapes do not require seeking, replacement selection is almost always a good choice for creating initial runs. As the number of drives available to hold run files is limited, the next question is how to distribute the files on the tapes. In most cases, it is necessary to put several runs on each of several tapes, reserving one or more other tapes for the results. This generally leads to merges of several steps, with the total number of runs being decreased after each merge step. Two approaches to doing this are *balanced merges* and *multiphase merges*. In a  $k$ -way balanced merge, all input tapes contain approximately the same number of runs, there are the same number of output tapes as there are input tapes, and the input tapes are read through entirely during each step. The number of runs is decreased by a factor of  $k$  after each step.

A multiphase merge (such as a *polyphase merge* or a *cascade merge*) requires that the runs initially be distributed unevenly among all but one of the available tapes. This increases the order of the merge and as a result can decrease the number of times each record has to be read. It turns out that the initial distribution of runs among the first set of input tapes has a major effect on the number of times each record has to be read.

Next, we discuss briefly the existence of sort-merge utilities, which are available on most large systems and can be very flexible and effective. We conclude the chapter with a listing of Unix utilities used for sorting and cosequential processing.

---

**KEY TERMS**

---

**Balanced merge.** A multistep merging technique that uses the same number of input devices as output devices. A two-way balanced merge uses two input tapes, each with approximately the same number of runs on it, and produces two output tapes, each with approximately half as many runs as the input tapes. A balanced merge is suitable for merge sorting with tapes, though it is not generally the best method (see *multiphase merging*).

**cmp.** A Unix utility for determining whether two files are identical. Given two files, it reports the first byte where the two files differ, if they differ.

**comm.** A Unix utility for determining which lines two files have in common. Given two files, it reports the lines they have in common, the lines that are in the first file and not in the second, and the lines that are in the second file and not in the first.

**Cosequential operations.** Operations applied to problems that involve the performance of union, intersection, and more complex set operations on two or more sorted input files to produce one or more output files built from some combination of the elements of the input files. Cosequential operations commonly occur in matching, merging, and file-updating problems.

**diff.** A Unix utility for determining all the lines that differ between two files. It reports the lines that need to be added to the first file to make it like the second, the lines that need to be deleted from the second file to make it like the first, and the lines that need to be changed in the first file to make it like the second.

**Heapsort.** A sorting algorithm especially well suited for sorting large files that fit in memory because its execution can overlap with I/O. A variation of heapsort is used to obtain longer runs in the replacement selection algorithm.

**HighValue.** A value used in the cosequential model that is greater than any possible key value. By assigning HighValue as the current key value for files for which an end-of-file condition has been encountered, extra logic for dealing with end-of-file conditions can be simplified.

**k-way merge.** A merge in which  $k$  input files are merged to produce one output file.

**LowValue.** A value used in the cosequential model that is less than any possible key value. By assigning LowValue as the previous key value



during initialization, the need for certain other special start-up code is eliminated.

**Match.** The process of forming a sorted output file consisting of all the elements common to two or more sorted input files.

**Merge.** The process of forming a sorted output file that consists of the union of the elements from two or more sorted input files.

**Multiphase merge.** A multistep tape merge in which the initial distribution of runs is such that at least the initial merge is a  $J-1$ -way merge ( $J$  is the number of available tape drives) and in which the distribution of runs across the tapes is such that the merge performs efficiently at every step. (See *polyphase merge*.)

**Multistep merge.** A merge in which not all runs are merged in one step. Rather, several sets of runs are merged separately, each set producing one long run consisting of the records from all of its runs. These new, longer sets are then merged, either all together or in several sets. After each step, the number of runs is decreased and the length of the runs is increased. The output of the final step is a single run consisting of the entire file. (Be careful not to confuse our use of the term *multistep merge* with *multiphase merge*.) Although a multistep merge is theoretically more time-consuming than a single-step merge, it can involve much less seeking when performed on a disk, and it may be the only reasonable way to perform a merge on tape if the number of tape drives is limited.

**Order of a merge.** The number of different files, or runs, being merged. For example, 100 is the order of a 100-way merge.

**Polyphase merge.** A multiphase merge in which, ideally, the merge order is maximized at every step.

**qsort.** A general-purpose Unix library routine for sorting files that employs a user-defined comparison function.

**Replacement selection.** A method of creating initial runs based on the idea of always *selecting* from memory the record whose key has the lowest value, outputting that record, and then *replacing* it in memory with a new record from the input list. When new records are brought in with keys that are greater than those of the most recently output records, they eventually become part of the run being created. When new records have keys that are less than those of the most recently output records, they are held over for the next run. Replacement selection generally produces runs that are substantially longer than runs

that can be created by in-memory sorts and hence can help improve performance in merge sorting. When using replacement selection with merge sorts on disk, however, one must be careful that the extra seeking required for replacement selection does not outweigh the benefits of having longer runs to merge.

**Run.** A sorted subset of a file resulting from the sort step of a sort merge or one of the steps of a multistep merge.

**Selection tree.** A binary tree in which each higher-level node represents the winner of the comparison between the two descendent keys. The minimum (or maximum) value in a selection tree is always at the root node, making the selection tree a good data structure for merging several lists. It is also a key structure in replacement selection algorithms, which can be used for producing long runs for merge sorts. (*Tournament sort*, an internal sort, is also based on the use of a selection tree.)

**Sequence checking.** Checking that records in a file are in the expected order. It is recommended that all files used in a cosequential operation be sequence checked.

**sort.** A Unix utility for sorting and merging files.

**Synchronization loop.** The main loop in the cosequential processing model. A primary feature of the model is to do all synchronization within a single loop rather than in multiple nested loops. A second objective is to keep the main synchronization loop as simple as possible. This is done by restricting the operations that occur within the loop to those that involve current keys and by relegating as much special logic as possible (such as error checking and end-of-file checking) to subprocedures.

**Theorem A (Knuth).** It is difficult to decide which merge pattern is best in a given situation.

## FURTHER READINGS

The subject matter treated in this chapter can be divided into two separate topics: the presentation of a model for cosequential processing and discussion of external merging procedures on tape and disk. Although most file processing texts discuss cosequential processing, they usually do it in the context of specific applications, rather than presenting a general model

that can be adapted to a variety of applications. We found this useful and flexible model through Dr. James VanDoren who developed this form of the model himself for presentation in the file structures course that he teaches. We are not aware of any discussion of the cosequential model elsewhere in the literature.

Quite a bit of work has been done toward developing simple and effective algorithms to do sequential file updating, which is an important instance of cosequential processing. The results deal with some of the same problems the cosequential model deals with, and some of the solutions are similar. See Levy (1982) and Dwyer (1981) for more.

Unlike cosequential processing, external sorting is a topic that is covered widely in the literature. The most complete discussion of the subject, by far, is in Knuth (1998). Students interested in the topic of external sorting must, at some point, familiarize themselves with Knuth's definitive summary of the subject. Knuth also describes replacement selection, as evidenced by our quoting from his book in this chapter.

Salzberg (1990) describes an approach to external sorting that takes advantage of replacement selection, parallelism, distributed computing, and large amounts of memory. Cormen, Leiserson, and Rivest (1990) and Loomis (1989) also have chapters on external sorting.

## EXERCISES

1. Consider the cosequential `Merge2Lists` method of Fig. 8.5 and the supporting methods of class `CosequentialProcess` in Appendix H. Comment on how they handle the following initial conditions. If they do not correctly handle a situation, indicate how they might be altered to do so.
  - a. List 1 empty and List 2 not empty
  - b. List 1 not empty and List 2 empty
  - c. List 1 empty and List 2 empty
2. Section 8.3.1 includes the body of a loop for doing a  $k$ -way merge, assuming that there are no duplicate names. If duplicate names are allowed, one could add to the procedure a facility for keeping a list of subscripts of duplicate lowest names. Modify the body of the loop to implement this. Describe the changes required to the supporting methods.

3. In Section 8.3, two methods are presented for choosing the lowest of  $K$  keys at each step in a  $K$ -way merge: a linear search and use of a selection tree. Compare the performances of the two approaches in terms of numbers of comparisons for  $K = 2, 4, 8, 16, 32$ , and  $100$ . Why do you think the linear approach is recommended for values of  $K$  less than  $8$ ?
4. Suppose you have  $80$  megabytes of memory available for sorting the  $8\,000\,000$ -record file described in Section 8.5.
  - a. How long does it take to sort the file using the merge-sort algorithm described in Section 8.5.1?
  - b. How long does it take to sort the file using the keysort algorithm described in Chapter 6?
  - c. Why will keysort not work if there are ten megabytes of memory available for the sorting phase?
5. How much seek time is required to perform a one-step merge such as the one described in Section 8.5 if the time for an average seek is  $10$  msec and the amount of available internal buffer space is  $5000\text{ K}$ ?
6. Performance in sorting is often measured in terms of the number of comparisons. Explain why the number of comparisons is not adequate for measuring performance in sorting large files.
7. In our computations involving the merge sorts, we made the simplifying assumption that only one seek and one rotational delay are required for any single sequential access. If this were not the case, a great deal more time would be required to perform I/O. For example, for the  $800$ -megabyte file used in the example in Section 8.5.1, for the input step of the sort phase ("reading all records into memory for sorting and forming runs"), each individual run could require many accesses. Now let's assume that the extent size for our hypothetical drive is  $80\,000$  bytes (approximately one track) and that all files are stored in track-sized blocks that must be accessed separately (one seek and one rotational delay per block).
  - a. How many seeks does step 1 now require?
  - b. How long do steps 1, 2, 3, and 4 now take?
  - c. How does increasing the file size by a factor of  $10$  now affect the total time required for the merge sort?
8. Derive two formulas for the number of seeks required to perform the merge step of a one-step  $k$ -way sort merge of a file with  $r$  records

divided into  $k$  runs, where the amount of available memory is equivalent to  $M$  records. If an internal sort is used for the sort phase, you can assume that the length of each run is  $M$ , but if replacement selection is used, you can assume that the length of each run is about  $2M$ . Why?

9. Assume a quiet system with four separately addressable disk drives, each of which is able to hold several gigabytes. Assume that the 800-megabyte file described in Section 8.5 is already on one of the drives. Design a sorting procedure for this sample file that uses the separate drives to minimize the amount of seeking required. Assume that the final sorted file is written off to tape and that buffering for this tape output is handled invisibly by the operating system. Is there any advantage to be gained by using replacement selection?
10. Use replacement selection to produce runs from the following files, assuming  $P = 4$ .
  - a. 2329517955413513318241147
  - b. 3591117182324293341475155
  - c. 5551474133292423181711953
11. Suppose you have a disk drive that has 10 read/write heads per surface, so 10 cylinders may be accessed at any one time without having to move the actuator arm. If you could control the physical organization of runs stored on disk, how might you be able to exploit this arrangement in performing a sort merge?
12. Assume we need to merge 14 runs on four tape drives. Develop merge patterns starting from each of these initial distributions:
  - a. 8—4—2
  - b. 7—4—3
  - c. 6—5—3
  - d. 5—5—4.
13. A four-tape polyphase merge is to be performed to sort the list 24 36 13 25 16 45 29 38 23 50 22 19 43 30 11 27 48. The original list is on tape 4. Initial runs are of length 1. After initial sorting, tapes 1, 2, and 3 contain the following runs (a slash separates runs):

Tape 1: 24 / 36 / 13 / 25

Tape 2: 16 / 45 / 29 / 38 / 23 / 50

Tape 3: 22 / 19 / 43 / 30 / 11 / 27 / 47

  - a. Show the contents of tape 4 after one merge phase.
  - b. Show the contents of all four tapes after the second and fourth phases.

- c. Comment on the appropriateness of the original 4—6—7 distribution for performing a polyphase merge.
14. Obtain a copy of the manual for one or more commercially available sort-merge packages. Identify the different kinds of choices available to users of the packages. Relate the options to the performance issues discussed in this chapter.
15. A join operation matches two files by matching field values in the two files. In the ledger example, a join could be used to match master and transaction records that have the same account numbers. The ledger posting operation could be implemented with a sorted ledger file and an indexed, entry-sequenced transaction file by reading a master record and then using the index to find all corresponding transaction records.

Compare the speed of this join operation with the cosequential processing method of this chapter. Don't forget to include the cost of sorting the transaction file.

## PROGRAMMING EXERCISES

16. Modify method `LedgerProcess::ProcessEndMaster` so it updates the ledger file with the new account balances for the month.
17. Implement the  $k$ -way merge in class `CosequentialProcessing` using an object of class `Heap` to perform the merge selection.
18. Implement a  $k$ -way match in class `CosequentialProcessing`.
19. Implement the sort merge operation using class `Heap` to perform replacement selection to create the initial sorted runs and class `CosequentialProcessing` to perform the merge phases.

## PROGRAMMING PROJECT

This is the sixth part of the programming project. We develop applications that produce student transcripts and student grade reports from information contained in files produced by the programming project of Chapter 4.

20. Use class `CosequentialProcesses` and `MasterTransactionProcess` to develop an application that produces student transcripts. For each student record (master) print the student information and a list of all courses (transaction) taken by the student. As input, use a file of student records sorted by student identifier and a file of course registration records sorted by student identifiers.
21. Use class `CosequentialProcesses` and `MasterTransactionProcess` to develop an application that produces student grade reports. As input, use a file of student records sorted by student identifier and a file of course registrations with grades for a single semester.

The next part of the programming project is in Chapter 9.