Chapter 14-1 & 2, Lecture Notes

Processing Data in External Storage

14.1 A look at External Storage

1. You use external storage when your program reads data from and writes data to a C++ file.
   1.1 Also, when you use a word processing program, for example, and choose Save, the program saves your current document in a file.
      1.1.1 This action enables you to exit the program and then use it later to retrieve your document for revision.
      1.1.2 This is one of the advantages of external storage: It exists beyond the execution period of a program.
      1.1.3 In this sense, it is "permanent" instead of volatile like internal memory.

   1.2 Another advantage of external storage is that, in general, there is far more of it than internal memory.
      1.2.1 If you have a table of one million data items, each of which is a record of moderate size, you will probably not be able to store the entire table in internal memory at one time.
      1.2.2 On the other hand, this mu data can easily reside on an external disk.
      1.2.3 As a consequence, when dealing with tables of this magnitude, you cannot simply read the entire table into memory when you want to operate on it and then write it back onto the disk when you are finished.
      1.2.4 Instead, you must devise ways to operate on data – for example... sort it and search it – while it resides externally.

   1.3 In general, you can create files for either sequential access or direct access.
      1.3.1 To access the data stored at a given position in a sequential access file, yon must advance the file window beyond all the intervening data.
      1.3.2 In this sense sequential access file resembles a linked list.
      1.3.3 To access a particular node in the list, you must traverse the list from its beginning until you reach the desired node.
      1.3.4 In contrast, a **direct access** file allows you to access the data at a given position directly.
      1.3.5 A direct access file resembles an array in that you can access the element at data [i] without first accessing the elements before data [i].

   1.4 Without direct access files, it would be impossible to support the table operations efficiently in an external environment.
      1.4.1 Many programming languages, including C++, support both sequential access and direct access of files.

1.4.2 However, to permit a language-independent discussion, we will construct model of direct access files that illustrates how a programming language that does not support such files might implement them.

1.4.3 This model will be a simplification of reality but will include the features necessary for this discussion.

1.5 Imagine that a computer's memory is divided into two parts: internal memory and external memory, as Figure 14-1 illustrates.

1.5.1 Assume that an executing program, along with its nonfile data, resides in the computer's internal memory; the permanent files of a computer system reside in the external memory.

1.5.2 Further assume that the external storage devices have the characteristics of a disk (although some systems use other devices).

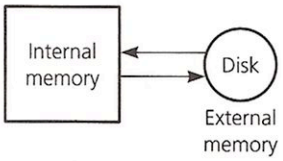

FIGURE 14-1
Internal and external memory

1.5.3 A file consists of **data records**.

1.5.4 A data record can be anything from a simple value, such as an integer, to an aggregate structure, such as an employee record.

1.5.5 For simplicity, assume that the data records in anyone file are all of the same type.

1.6 The records of a file are organized into one or more blocks, as Figure 14-2 shows.

1.6.1 The size of a block-that is, the number of bits of data it can contain-is determined by both the hardware configuration and the system software of the computer.

1.6.1.1 In general, an individual program has no control over this size.

1.6.1.2 Therefore, the number of records in a block is a function of the size of the records in the file.

1.6.1.3 For example, a file of integer records will have more records per block than a file of employee records.

1.6.2 Much as you number the elements of an array, you can number the blocks of a file in a linear sequence.

1.6.2.1 With a direct access file, a program can read a given block from the file by specifying its block number, and similarly, it can write data out to a particular block.

1.6.2.2 In this regard a direct access file resembles an array of arrays, with each block of the file analogous to a single array entry, which is itself an array that contains several records.

1.6.3 In this direct access model, all input and output is at the block level rather than at the record level.
1.6.3.1 That is, you can read and write a block of records, but you cannot read or write an individual record.
1.6.3.2 Reading or writing a block is called a block access.

1.6.4 The algorithms in this chapter assume commands for reading and writing blocks.
1.6.4.1 The statement

buf.readBlock(dataFile, i)

1.6.4.2 will read the i^{th} block of file dataFile and place it in an object buf.
1.6.4.3 The object must accommodate the many records that each block of file dataFile contains.

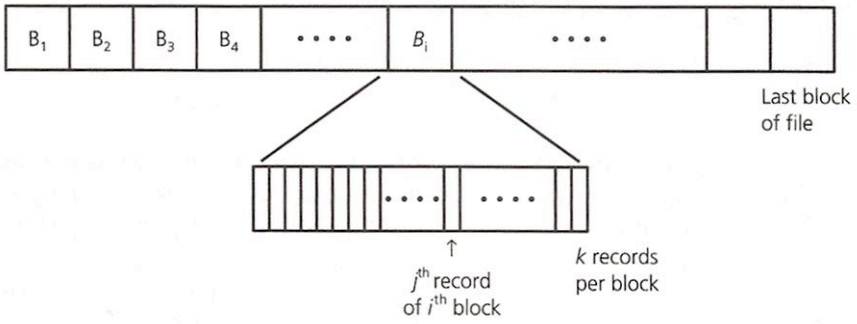

FIGURE 14-2
A file partitioned into blocks of records

1.6.4.4 For example, if each block contains 100 employee records, buf must store at least 100 employee records.
1.6.4.5 The object buf is called a buffer, which is a location that temporarily stores data as it makes its way from one process or location to another.

1.6.5 Once the system has read a block into buf, the program can process-for example, inspect or modify-the records in the block.
1.6.5.1 Also, because the records in the object buf are only copies of the records in the file dataFile, if a program does modify the records in buf, it must write buf back out to dataFile, so that the file also reflects the modifications.
1.6.5.2 We assume that the statement

buf.writeBlock(dataFile, i)

1.6.5.3 will write the contents of buf to the ith block of the file dataFile.

1.6.5.4 If dataFile contains n blocks, the statement

buf.writeBlock(dataFile, n + 1)

1.6.5.5 will append a new block to dataFile, and thus the file can grow dynamically_ just as a C++ file can.

1.7 Again, realize that these input and output commands allow you to read and write only entire blocks.

1.7.1 As a consequence, even if you need to operate on only a single record of the file, you must access an entire block.

1.7.1.1 For example suppose that you want to give employee Smith a $1,000 raise.

1.7.1.2 If Smith's record is in block i (how to determine the correct block is discussed later the chapter), you would perform the following steps:

```
// read block i from file dataFile into buffer buf buf.readBlock(dataFile, i)

Find the entry buf.getRecord(j) that contains the record whose search key is "Smith"

// increase the salary portion of Smith's record
(buf. getRecord (j ) ).setSalary ( (buf.getRecord (j ) ).getSalary () + 1000)

// write changed block back to file dataFile
buf.writeBlock(dataFile, i)
```

1.7.2 The time required to read or write a block of data is typically much longer than the time required to operate on the block's data once it is in the computer's internal memory[1].

1.7.3 For example, you typically can inspect every record in the buffer buf in less time than that required to read a block into the buffer.

1.7.3.1 As a consequence, you should reduce the number of required block accesses.

1.7.3.2 In the previous pseudocode, for instance, you should process as many records in buf as possible before writing it to the file.

1.7.3.3 You should pay little attention to the time required to operate on a block of data once it has been read into internal memory.

1.7.4 Interestingly, several programming languages, including C++, have commands to make it appear that you can access records one at a time.

1.7.4.1 In general, however, the system actually performs input and output at the block level and perhaps hides this fact from the program.

---

[1] Data enters or leaves a buffer at a rate that differs from the record-processing ran:... (Hence, a buffer between two processes compensates for the difference in the rates which they operate on data.)

1.7.4.2 For example, if a programming language includes the statement

```
rec.readRecord(dataFile, i)
// Reads the ith record of file dataFile into rec.
```

1.7.4.3 the system probably accesses the entire block that contains the ith record.

1.7.4.4 Our model of input and output therefore approximates reality reasonably well.

1.7.5 In most external data-management applications, the time required for block accesses typically dominates all other factors.

1.7.5.1 The rest of the chapter discusses how to sort and search externally stored data.

1.7.5.2 The goal will be to reduce the number of required block accesses.

14.2 Sorting Data in an External File

2 This section considers the following problem of sorting data that resides in an external file:

2.1 An external file contains 1,600 employee records.

2.1.1 You want to sort these records by Social Security number. Each block contains 100 records, and thus the file contains 16 blocks Bb B2' and so on to B16.

2.1.2 Assume that the program can access only enough internal memory to manipulate about 300 records (three blocks' worth) at one time.

2.2 Sorting the file might not sound like a difficult task, because you have already seen several sorting algorithms earlier in this book.

2.2.1 There is, however, a fundamental difference here in that the file is far too large to fit into internal memory all at once.

2.2.2 This restriction presents something of a problem because the sorting algorithms presented earlier assume that all the data to be sorted is available at one time in internal memory (for example, that it is all in an array).

2.2.3 Fortunately, however, we can remove this assumption for a modified version of mergesort.

2.2.3.1 The basis bf the mergesort algorithm is that you can easily merge two sorted segments-such as arrays-of data records into a third sorted segment that is the combination of the two.

2.2.3.2 For example, if Sl and S2 are sorted segments of records, the first step of the merge is to compare the first record of each segment and select the record with the smaller search key.

2.2.3.3 If the record from S_1 is selected, the next step is to compare the second record of S_l to the first record of S_2.
2.2.3.4 This process is continued until all of the records have been considered.
2.2.3.5 The key observation is that at any step, the merge never needs to look beyond the leading edge of either segment.

2.3 This observation makes a mergesort appropriate for the problem of sorting external files, if you modify the algorithm appropriately.
2.3.1 Suppose that the 1,600 records to be sorted are in the file F and that you are not permitted to alter this file.
2.3.2 You have two work files, FI and F2.
2.3.3 One of the work files will contain the sorted records when the algorithm terminates.
2.3.4 The algorithm has two phases: Phase 1 sorts each block of records, and Phase 2 performs a series of merges.

2.3.4.1 **Phase 1**.
2.3.4.1.1 Read a block from F into internal memory, sort its records by using an internal sort, and write the sorted block out to FI before you read the next block from F
2.3.4.1.2 After you process all 16 blocks of F, F_1 contains 16 **sorted runs** R_1, R_2, and so on to R_{16}; that is, F_1 contains 16 blocks of records, with the records within each block sorted among themselves, as Figure 14-3a illustrates.

2.3.4.2 **Phase 2.**
2.3.4.2.1 Phase 2 is a sequence of merge steps.
2.3.4.2.2 Each merge step merges pairs of sorted runs to form larger sorted runs.
2.3.4.2.3 Each merge step doubles the number of blocks in each sorted run and thus halves the total number of sorted runs.
2.3.4.2.4 For example, as Figure 14-3b shows, the first merge step merges eight pairs of sorted runs from FI (R_1 with R_2, R_3 with R_4, ... , R_{15} with R_{16}) to form eight sorted runs, each two blocks long, which are written to F_2.
2.3.4.2.5 The next merge step merges four pairs of sorted runs from F_2 (R_1 with R_2, R_3 with R_4, ... , R_7 with R_8) to form four sorted runs, each four blocks long, which are written back to F_1, as Figure 14-3c illustrates.
2.3.4.2.6 The next step merges the two pairs of sorted runs from F_1 to form two sorted runs, which are written to F2 (See Figure 14- 3d.)
2.3.4.2.7 The final step merges the two sorted runs into one, which is written to F_1.

2.3.4.2.8 At this point, F1 will contain all of the records of the original file in sorted order.

2.3.5 Given this overall strategy, how can you merge the sorted runs at each step of Phase 2?

2.3.5.1 The statement of the problem provides only sufficient internal memory to manipulate at most 300 records at once.

2.3.5.2 However, in the later steps of Phase 2, runs contain more than 300 records each, so you must merge the runs a piece at a time.

2.3.5.3 To accomplish this merge, you must divide the program's internal memory into three arrays, in1) in2, and out, each capable of holding 100 records (the block size).

2.3.5.4 You read block-sized pieces of the runs into the two in arrays and merge them into the out array.

2.3.5.5 Whenever an in array is exhausted – that is, when all of its elements have been copied to out – you read the next piece of the run into the in array; whenever the out array becomes full, you write this completed piece of the new sorted run to one of the files.

2.3.6 Consider how you can perform the first merge step.

2.3.6.1 You start this step with the pair of runs R_1 and R_2, which are in the first and second blocks, respectively, of the file F_1· (See Figure 14-3a.)

2.3.6.2 Because at this first merge step each run contains only one block, an entire run can fit into one of the in arrays.

2.3.6.3 You can thus read R_1 and R_2 into the arrays in1 and in2) and then merge in1 and in2 into out.

2.3.6.4 However, although the result of merging inl and in2 is a sorted run two blocks long (200 records), out can hold only one biock (100 records).

2.3.6.5 Thus, when in the course of the merge out becomes full, you write its contents to the first block of F_2) as Figure 14-4a illustrates.

2.3.6.6 The merging of in1 and in2 into out then resumes.

2.3.6.7 The array out will become full for a second time only after all of the records in in1 and in2 are exhausted.

2.3.6.8 At that time, write the contents of out to the second block of F_2.

2.3.6.9 You merge the remaining seven pairs from F in the same manner and append the resulting runs to F_2.

2.3.7 This first merge step is conceptually a bit easier than the others, because the initial runs are only one block in size, and thus each can fit entirely into one of the in arrays.

2.3.7.1 What do you do in the later steps when the runs to be merged are larger than a single block?

2.3.7.2 Consider, for example, the merge step in which you must merge runs of four blocks each to form runs of eight blocks each. (See Figure 14-3c.)

2.3.7.3 The first pair of these runs to be merged is in blocks 1 through 4 and 5 through 8 of F_1.

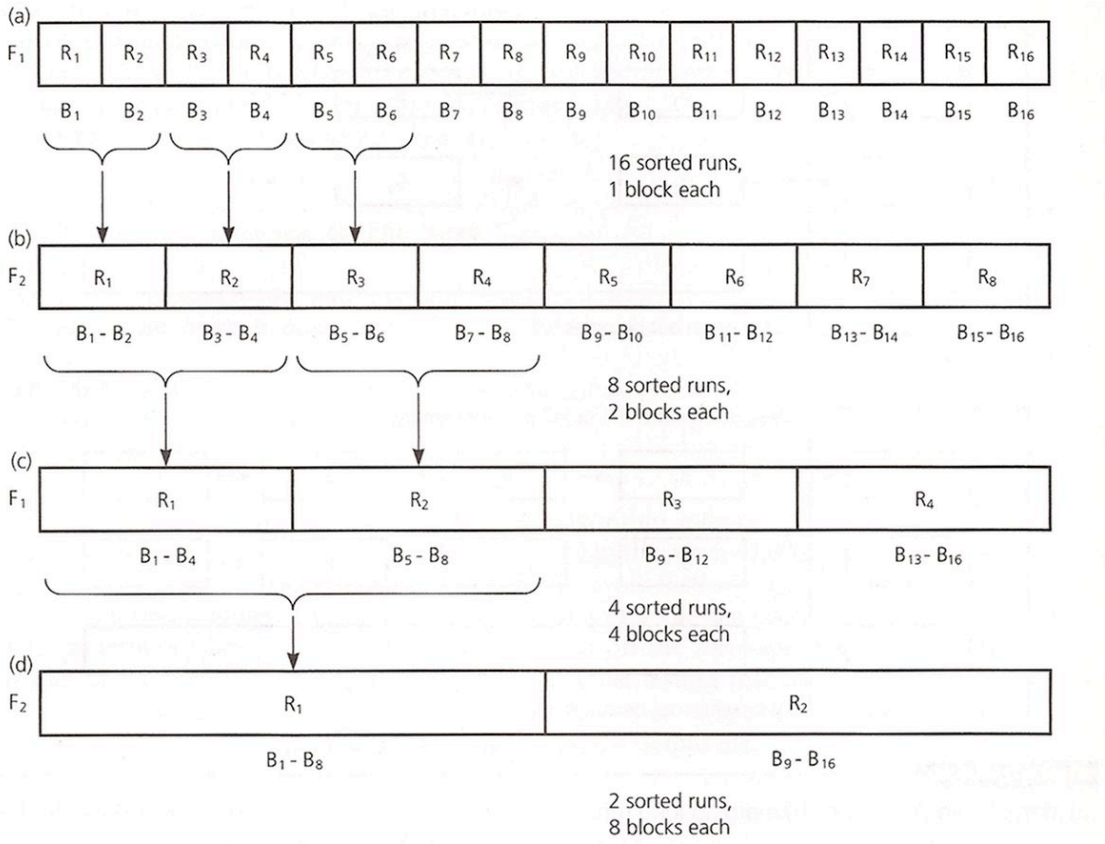

FIGURE 14-3

(a) Sixteen sorted runs, one block each, in file F_1; (b) Eight sorted runs, two blocks each, in file F_2; (c) Four sorted runs, four blocks each, in file F_1; (d) Two sorted runs, eight blocks each, in file F_2

2.3.8 The algorithm will read the first block of R_1 – which is the first block B_1 of the file-into in1, and it will read the first block of R2 – which is B5 – into in2, as Figure 14-4b illustrates.

2.3.8.1 Then, as it did earlier, the algorithm merges in1 and in2 into out.

2.3.8.2 The complication here is that as soon as you finish moving all of the records from either in1 or in2, you must read the next block from the corresponding run.

2.3.8.3 For example, if you finish in2 first, you must read the next block of R2 – which is B6 – into in2 before the merge can continue.

2.3.8.4 The algorithm thus must detect when the in arrays become exhausted as well as when the out array becomes full.

2.4 A high-level description of the algorithm for merging arbitrary-sized sorted runs R_i and R_j from F_1 into F_2 is as follows:

```
Read the first block of Ri into in1
Read the first block of Rj into in2

while (either in1 or in2 is not exhausted)

{ Select the smaller "leading" record of in1 and in2 and place it into the next
        position of out (if one of the arrays is exhausted, select the leading record
        from the other)

        if (out is full)
                Write its contents to the next block of F2

        if (in1 is exhausted and blocks remain in Ri)
                Read the next block into in1

        if (in2 is exhausted and blocks remain in Rj)
                Read the next block into in2
} // end while
```

2.5 A pseudocode version of the external sorting algorithm follows.

2.5.1 Notice that it uses readBlock and wri teBlock, as introduced in the previous section, and assumes a function copyFile that copies a file.

2.5.2 To avoid further complications, the solution assumes that the number of blocks in the file is a power of 2.

2.5.3 This assumption allows the algorithm always to pair off the sorted runs at each step of the merge phase, avoiding special end-of-file testing that would obscure the algorithm.

2.5.4 Also note that the algorithm uses two temporary files and copies the final sorted temporary file to the designated output file.

```
externalMergesort(in unsortedFileName:String,
                  in sortedFileName:String)

// Sorts a file by using an external mergesort. mer!
// Precondition: unsortedFileName is the name of an external
// file to be sorted. sortedFileName is the name that the
// function will give to the resulting sorted file.
// Postcondition: The new file named sortedFileName is
// sorted. The original file is unchanged. Both files are
// closed.
// Calls: blockSort, mergeFile, and copyFile.
// Simplifying assumption: The number of blocks in the
// unsorted file is an exact power of 2.
```

```
Associate unsortedFileName with the file variable inFile and sortedFileName
        with the file variable outFile

// Phase 1: sort file block by block and count the blocks
blockSort(inFile, tempFile1, numberOfBlocks)

// Phase 2: merge runs of size 1, 2, 4, 8, ... ,
// numberOfBlocksl2 (uses two temporary files and a
// toggle that keeps files for each merge step)

toggle = 1
for (size = 1 through numberOfBlocks/2 with
        increments of size)
{ if (toggle == 1)
        mergeFile(tempFile1, tempFile2, size, numberOfBlocks)
  else
        mergeFile(tempFile2, tempFile1, size, numberOfBlocks)

   toggle = ~toggle
} // end for

// copy the current temporary file to outFile
if (toggle == 1)
        copyFile(tempFile1, outFile)
else
        copyFile(tempFile2, outFile)
```

2.5.5 Notice that externalMergesort calls blockSort and mergeFile) which calls mergeRuns.

2.5.6 The pseudocode for these functions follows.

```
blockSort(in inFile:File, in outFile:File,
            in numberOfBlocks:integer)
// Sorts each block of records in a file.
// Precondition: The file variable inFile is associated
// with the file to be sorted.
// Postcondition: The file associated with the file variable
// outFile contains the blocks of inFile. Each block is
// sorted; numberOfBlocks is the number of blocks processed.
// Both files are closed.
// Calls: readBlock and writeBlock to perform direct access
// input and output, and sortBuffer to sort an array.
```

```
        Prepare inFile for input
        Prepare outFile for output

        numberOfBlocks = 0

        while (more blocks in inFile remain to be read)
        { ++numberOfBlocks
          buffer.readBlock(inFile, numberOfBlocks)

          sortArray(buffer) // sort with some internal sort

          buffer.writeBlock(outFile, numberOfBlocks)
        } // end while

        Close inFile and outFile
// end blockSort

mergeFile(in inFile:File, in outFile:File,
            in runSize:integer, in numberOfBlocks:integer)
// Merges blocks from one file to another.
// Precondition: inFile is an external file that contains
// numberOfBlocks sorted blocks organized into runs of
// runSize blocks each.
// Postcondition: outFile contains the merged runs of
// inFile. Both files are closed.
// Calls: mergeRuns.

        Prepare inFile for input
        Prepare outFile for output

        for (next = 1 to numberOfBlocks with increments of 2 * runSize)
        { // Invariant: runs in outFile are ordered
           mergeRuns(inFile, outFile, next, runSize)
        } // end for
        Close inFile and outFile
// end mergeFile

mergeRuns(in fromFile:File, in toFile:File,
        in start:integer, in size:integer)
// Merges two consecutive sorted runs in a file.
// Precondition: fromFile is an external file of sorted runs
// open for input. toFile is an external file of sorted runs
// open for output. start is the block number of the first
// run on fromFile to be merged; this run contains size
// blocks.
//      Run 1: block start to block start + size - 1
```

```
//      Run 2: block start + size to start + (2 * size) - 1
// Postcondition: The merged runs from fromFile are appended
// to toFile. The files remain open.

        // initialize the input buffers for runs 1 and 2
        in1.readBlock(fromFile, first block of Run 1)
        in2.readBlock(fromFile, first block of Run 2)

        // Merge until one of the runs is finished. Whenever an
        // input buffer is exhausted, the next block is read.
        // Whenever the output buffer is full, it is written.
        while (neither run is finished) {
                // Invariant: out and each block in toFile are ordered
                Select the smaller "leading edge" of in1 and in2,
                    and place it in the next position of out
                if (out is full)
                    out.writeBlock(toFile, next block of toFile)
                if (in1 is exhausted and blocks remain in Run 1)
                    in1.readBlock(fromFile, next block of Run 1)

                if (in2 is exhausted and blocks remain in Run 2)
                    in2.readBlock(fromFile, next block of Run 2)
        } / / end while

        // Assertion: exactly one of the runs is complete

        // append the remainder of the unfinished input
        // buffer to the output buffer and write it

        while (in1 is not exhausted)
                // Invariant: out is ordered
                Place next item of in1 into the next position of out

        while (in2 is not exhausted)
                // Invariant: out is ordered
                Place next item of in2 into the next position of out

        out.writeBlock(toFile, next block of toFile)

        // finish off the remaining complete blocks

        while (blocks remain in Run 1)
        { // Invariant: each block in toFile is ordered
          in1.readBlock(fromFile, next block of Run 1)
          in1.writeBlock(toFile, next block of toFile)
        } // end while
```

```
        while (blocks remain in Run 2)
        { // Invariant: Each block in toFile is ordered
          in2.readBlock(fromFile, next block of Run 2)
          in2.writeBlock(toFile, next block of toFile)
        } // end while
// end mergeRuns
```

14.3 External Tables

This section discusses techniques for organizing records in external storage so that you can efficiently perform ADT table operations such as retrieval, insertion, deletion, and traversal. Although this discussion will only scratch the surface of this topic, you do have a head start: Two of the most important external table implementations are variations of the 2-3 tree and hashing, which you studied in Chapter 12.