CHAPTER 7

Indexing

CHAPTER OBJECTIVES

- Introduce concepts of *indexing* that have broad applications in the design of file systems.
- Introduce the use of a *simple linear index* to provide rapid access to records in an entry-sequenced, variable-length record file.
- Investigate the implications of the use of indexes for file maintenance.
- Introduce the template features of C++.
- Discuss the object-oriented approach to indexed sequential files.
- Describe the use of indexes to provide access to records by more than one key.
- Introduce the idea of an *inverted list*, illustrating *Boolean operations* on lists.
- Discuss the issue of *when to bind* an index key to an address in the data file.
- Introduce and investigate the implications of *self-indexing* files.

7.1 What Is an Index?

The last few pages of many books contain an index. Such an index is a table containing a list of topics (keys) and numbers of pages where the topics can be found (reference fields).

All indexes are based on the same basic concept—keys and reference fields. The types of indexes we examine in this chapter are called *simple indexes* because they are represented using *simple arrays* of structures that contain the keys and reference fields. In later chapters we look at indexing schemes that use more complex data structures, especially trees. In this chapter, however, we want to emphasize that indexes can be very simple and still provide powerful tools for file processing.

The index to a book provides a way to find a topic quickly. If you have ever had to use a book that doesn't have a good index, you already know that an index is a desirable alternative to scanning through the book sequentially to find a topic. In general, indexing is another way to handle the problem we explored in Chapter 6: an index is a way to find things.

Consider what would happen if we tried to apply the previous chapter's methods, sorting and binary searching, to the problem of finding things in a book. Rearranging all the words in the book so they were in

alphabetical order certainly would make finding any particular term easier but would obviously have disastrous effects on the meaning of the book. In a sense, the terms in the book are pinned records. This is an absurd example, but it clearly underscores the power and importance of the index as a conceptual tool. Since it works by indirection, *an index lets you impose order on a file without rearranging the file.* This not only keeps us from disturbing pinned records, but also makes matters such as record addition much less expensive than they are with a sorted file.

Take, as another example, the problem of finding books in a library. We want to be able to locate books by a specific author, title, or subject area. One way of achieving this is to have three copies of each book and three separate library buildings. All of the books in one building would be sorted by author's name, another building would contain books arranged by title, and the third would have them ordered by subject. Again, this is an absurd example, but one that underscores another important advantage of indexing. Instead of using multiple arrangements, a library uses a card catalog. The card catalog is actually a set of three indexes, each using a different *key field,* and all of them using the same catalog number as a *reference field.* Another use of indexing, then, is to provide *multiple access paths* to a file.

We also find that indexing gives us *keyed access to variable-length record files.* Let's begin our discussion of indexing by exploring this problem of access to variable-length records and the simple solution that indexing provides.

One final note: the example data objects used in the following sections are musical recordings. This may cause some confusion as we use the term *record* to refer to an object in a file, and *recording* to refer to a data object. We will see how to get information about recordings by finding records in files. We've tried hard to make a distinction between these two terms. The distinction is between the file system view of the elements that make up files (records), and the user's or application's view of the objects that are being manipulated (recordings).

7.2 A Simple Index for Entry-Sequenced Files

Suppose we own an extensive collection of musical recordings, and we want to keep track of the collection through the use of computer files. For each recording, we keep the information shown in Fig. 7.1. Appendix G includes files `recordng.h` and `recordng.cpp` that define class

Identification number
Title
Composer or composers
Artist or artists
Label (publisher)

Figure 7.1 Contents of a data record.

`Recording`. Program `makerec.cpp` in Appendix G uses cla `DelimFieldBuffer` and `BufferFile` to create the file `Recording` objects displayed in Fig. 7.2. The first column of the ta contains the record addresses associated with each record in the file.

Suppose we formed a *primary key* for these recordings consisting the initials for the company label combined with the recording's number. This will make a good primary key as it should provide a uni key for each entry in the file. We call this key the *Label ID*. The canoni form for the *Label ID* consists of the uppercase form of the Label fie followed immediately by the ASCII representation of the `ID` number. F example,

```
LON2312
```

Record address	Label	ID number	Title	Composer(s)	Artist(s)
17	LON	2312	Romeo and Juliet	Prokofiev	Maazel
62	RCA	2626	Quartet in C Sharp Minor	Beethoven	Julliard
117	WAR	23699	Touchstone	Corea	Corea
152	ANG	3795	Symphony No. 9	Beethoven	Giulini
196	COL	38358	Nebraska	Springsteen	Springsteen
241	DG	18807	Symphony No. 9	Beethoven	Karajan
285	MER	75016	Coq d'Or Suite	Rimsky-Korsakov	Leinsdorf
338	COL	31809	Symphony No. 9	Dvorak	Bernstein
382	DG	139201	Violin Concerto	Beethoven	Ferras
427	FF	245	Good News	Sweet Honey in the Rock	Sweet Honey in the Rock

Figure 7.2 Contents of sample recording file.

How could we organize the file to provide rapid keyed access to individual records? Could we sort the file and then use binary searching? Unfortunately, binary searching depends on being able to jump to the middle record in the file. This is not possible in a variable-length record file because direct access by relative record number is not possible; there is no way to know where the middle record is in any group of records.

An alternative to sorting is to construct an index for the file. Figure 7.3 illustrates such an index. On the right is the data file containing information about our collection of recordings, with one variable-length data record per recording. Only four fields are shown (Label, ID number, Title, and Composer), but it is easy to imagine the other information filling out each record.

On the left is the index, each entry of which contains a *key* corresponding to a certain Label ID in the data file. Each key is associated with a *reference field* giving the address of the first byte of the corresponding data record. ANG3795, for example, corresponds to the reference field containing the number 152, meaning that the record containing full information on the recording with Label ID ANG3795 can be found starting at byte number 152 in the record file.

Index

Key	*Reference field*
ANG3795	152
COL31809	338
COL38358	196
DG139201	382
DG18807	241
FF245	427
LON2312	17
MER75016	285
RCA2626	62
WAR23699	117

Recording file

Address of record	*Actual data record*
17	LON | 2312 | Romeo and Juliet | Prokofiev | ...
62	RCA | 2626| Quartet in C Sharp Minor | Beethoven | ...
117	WAR | 23699 | Touchstone | Corea | ...
152	ANG | 3795 | Symphony No. 9 | Beethoven | ...
196	COL | 38358 | Nebraska | Springsteen | ...
241	DG | 18807 | Symphony No. 9 | Beethoven | ...
285	MER | 75016 | Coq d'Or Suite | Rimsky-Korsakov | ...
338	COL | 31809 | Symphony No. 9 | Dvorak | ...
382	DG | 139201 | Violin Concerto | Beethoven | ...
427	FF | 245 | Good News |Sweet Honey in the Rock | ...

Figure 7.3 Index of the sample recording file.

The structure of the index object is very simple. It is a list of pairs of fields: a key field and a byte-offset field. There is one entry in the index for each record in the data file. Class `TextIndex` of Fig. 7.4 encapsulates the index data and index operations. The full implementation of class `TextIndex` is given in files `textind.h` and `textind.cpp` of Appendix G. An index is implemented with arrays to hold the keys and record references. Each object is declared with a maximum number of entries and can be used for unique keys (no duplicates) and for nonunique keys (duplicates allowed). The methods `Insert` and `Search` do most of the work of indexing. The protected method `Find` locates the element key and returns its index. If the key is not in the index, `Find` returns -1. This method is used by `Insert`, `Remove`, and `Search`.

A C++ feature used in this class is the *destructor*, method `~TextIndex`. This method is automatically called whenever a `TextIndex` object is deleted, either because of the return from a function that includes the declaration of a `TextIndex` object or because of explicit deletion of an object created dynamically with `new`. The role of the destructor is to clean up the object, especially when it has dynamically created data members. In the case of class `TextIndex`, the protected members `Keys` and `RecAddrs` are created dynamically by the constructor and should be deleted by the destructor to avoid an obvious memory leak:

```
TextIndex::~TextIndex (){delete Keys; delete RecAddrs;}
```

```
class TextIndex
{public:
   TextIndex (int maxKeys = 100, int unique = 1);
   int Insert (const char * key, int recAddr); // add to index
   int Remove (const char * key); // remove key from index
   int Search (const char * key) const;
      // search for key, return recaddr
   void Print (ostream &) const;
protected:
   int MaxKeys; // maximum number of entries
   int NumKeys; // actual number of entries
   char * * Keys; // array of key values
   int * RecAddrs; // array of record references
   int Find (const char * key) const;
   int Init (int maxKeys, int unique);
   int Unique; // if true, each key must be unique in the index
};
```

Figure 7.4 Class TextIndex.

Note also that the index is sorted, whereas the data file is not. Consequently, although Label ID ANG3795 is the first entry in the index, it is not necessarily the first entry in the data file. In fact, the data file is *entry sequenced*, which means that the records occur in the order they are entered into the file. As we will see, the use of an entry-sequenced file can make record addition and file maintenance much simpler than the case with a data file that is kept sorted by some key.

Using the index to provide access to the data file by Label ID is a simple matter. The code to use our classes to retrieve a single record by key from a recording file is shown in the function `RetrieveRecording`:

```
int RetrieveRecording (Recording & recording, char * key,
      TextIndex & RecordingIndex, BufferFile & RecordingFile)
// read and unpack the recording, return TRUE if succeeds
{ int result;
  result = RecordingFile . Read (RecordingIndex.Search(key));
  if (result == -1) return FALSE;
  result = recording.Unpack (RecordingFile.GetBuffer());
  return result;
}
```

With an open file and an index to the file in memory, `RetrieveRecording` puts together the index search, file read, and buffer unpack operations into a single function.

Keeping the index in memory as the program runs also lets us find records by key more quickly with an indexed file than with a sorted one since the binary searching can be performed entirely in memory. Once the byte offset for the data record is found, a single seek is all that is required to retrieve the record. The use of a sorted data file, on the other hand, requires a seek for each step of the binary search.

7.3 Using Template Classes in C++ for Object I/O

A good object-oriented design for a file of objects should provide operations to read and write data objects without having to go through the intermediate step of packing and unpacking buffers. In Chapter 4, we supported I/O for data with the buffer classes and class `BufferFile`. In order to provide I/O for objects, we added `Pack` and `Unpack` methods to our `Person` object class. This approach gives us the required functionality but

stops short of providing a read operation whose arguments are a file and a data object. We want a class `RecordFile` that makes the following code possible:

```
Person p; RecordFile pFile;   pFile . Read (p);
Recording r;   RecordFile rFile;   rFile . Read (r);
```

The major difficulty with defining class `RecordFile` is making it possible to support files for different record types without having to modify the class. Is it possible that class `RecordFile` can support read and unpack for a `Person` and a `Recording` without change? Certainly the objects are different; they have different unpacking methods. Virtual function calls do not help because `Person` and `Recording` do not have a common base type. It seems that class `RecordFile` needs to be parameterized so different versions of the class can be constructed for different types of data objects.

It is the C++ *template* feature that supports parameterized function and class definitions, and `RecordFile` is a template class. As shown in Fig. 7.5, class `RecordFile` includes the parameter `RecType`, which is used as the argument type for the read and write methods of the class. Class `RecordFile` is derived from `BufferFile`, which provides most of the functionality. The constructor for `RecordFile` is given inline and simply calls the `BufferFile` constructor.

The definitions of `pFile` and `rFile` just given are not consistent with use of a template class. The actual declarations and calls are:

```
RecordFile <Person> pFile;      pFile . Read (p);
RecordFile <Recording> rFile;   rFile . Read (p);
```

```
template <class RecType>
class RecordFile: public BufferFile
{public:
   int Read (RecType & record, int recaddr = -1);
   int Write (const RecType & record, int recaddr = -1);
   int Append (const RecType & record);
   RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};
// The template parameter RecType must have the following methods
// int Pack (IOBuffer &); pack record into buffer
// int Unpack (IOBuffer &); unpack record from buffer
```

Figure 7.5 Template Class RecordFile.

Object `rFile` is of type `RecordFile<Recording>`, which is an *instance* of class `RecordFile`. The call to `rFile.Read` looks the same as the call to `pFile.Read`, and the two methods share the same source code, but the implementations of the classes are somewhat different. In particular, the `Pack` and `Unpack` methods of class `Recording` are used for methods of object `rFile`, but `Person` methods are used for `pFile`.

The implementation of method `Read` of class `RecordFile` is given in Fig. 7.6; the implementation of all the methods are in file `recfile.h` in Appendix G. The method makes use of the `Read` method of `BufferFile` and the `Unpack` method of the parameter `RecType`. A new version of `RecordFile::Read` is created by the C++ compiler for each instance of `RecordFile`. The call `rFile.Read(r)` calls `Recording::Unpack`, and the call `pFile.Read(p)` calls `Person::Unpack`.

Class `RecordFile` accomplishes the goal of providing object-oriented I/O for data. Adding I/O to an existing class (class `Recording`, for example) requires three steps:

1. Add methods `Pack` and `Unpack` to class `Recording`.
2. Create a buffer object to use in the I/O:

   ```
   DelimFieldBuffer Buffer;
   ```

3. Declare an object of type `RecordFile<Recording>`:

   ```
   RecordFile<Recording> rFile (Buffer);
   ```

Now we can directly open a file and read and write objects of class `Recording`:

```
Recording r1, r2;
rFile . Open ("myfile");
rFile . Read (r1);
rFile . Write (r2);
```

7.4 Object-Oriented Support for Indexed, Entry-Sequenced Files of Data Objects

Continuing with our object-oriented approach to I/O, we will add indexed access to the sequential access provided by class `RecordFile`. A new class, `IndexedFile`, extends `RecordFile` with `Update` and

```
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr)
{
   int writeAddr, result;
   writeAddr = BufferFile::Read (recaddr);
   if (!writeAddr) return -1;
   result = record . Unpack (Buffer); //RecType::Unpack
   if (!result) return -1;
   return writeAddr;
}
```

Figure 7.6 Implementation of RecordFile::Read.

`Append` methods that maintain a primary key index of the data file and a `Read` method that supports access to object by key.

So far, we have classes `TextIndex`, which supports maintenance and search by primary key, and `RecordFile`, which supports create, open, and close for files as well as read and write for data objects. We have already seen how to create a primary key index for a data file as a memory object. There are still two issues to address:

- How to make a persistent index of a file. That is, how to store the index in a file when it is not in memory.
- How to guarantee that the index is an accurate reflection of the contents of the data file.

7.4.1 Operations Required to Maintain an Indexed File

The support and maintenance of an entry-sequenced file coupled with a simple index requires the operations to handle a number of different tasks. Besides the `RetrieveRecording` function described previously, other operations used to find things by means of the index include the following:

- Create the original empty index and data files,
- Load the index file into memory before using it,
- Rewrite the index file from memory after using it,
- Add data records to the data file,
- Delete records from the data file,

- Update records in the data file, and
- Update the index to reflect changes in the data file.

A great benefit of our object-oriented approach is that everything we need to implement these operations is already available in the methods of our classes. We just need to glue them together. We begin by identifying the methods required for each of these operations. We continue to use class `Recording` as our example data class.

Creating the Files

Two files must be created: a data file to hold the data objects and an index file to hold the primary key index. Both the index file and the data file are created as empty files, with header records and nothing else. This can be accomplished quite easily using the `Create` method implemented in class `BufferFile`. The data file is represented by an object of class `RecordFile<Recording>`. The index file is a `BufferFile` of fixed-size records, as described below. As an example of the manipulation of index files, program `makeind.cpp` of Appendix G creates an index file from a file of recordings.

Loading the Index into Memory

Both loading (reading) and storing (writing) objects is supported in the `IOBuffer` classes. With these buffers, we can make files of index objects. For this example, we are storing the full index in a single object, so our index file needs only one record. As our use of indexes develops in the rest of the book, we will make extensive use of multiple record index files.

We need to choose a particular buffer class to use for our index file. We define class `TextIndexBuffer` as a derived class of `FixedFieldBuffer` to support reading and writing of index objects. `TextIndexBuffer` includes pack and unpack methods for index objects. This style is an alternative to adding these methods to the data class, which in this case is `TextIndexBuffer`. The full implementation of class `TextIndexBuffer` is in files `tindbuff.h` and `tindbuff.cpp` in Appendix G.

Rewriting the Index File from Memory

As part of the `Close` operation on an `IndexedFile`, the index in memory needs to be written to the index file. This is accomplished using the `Rewind` and `Write` operations of class `BufferFile`.

It is important to consider what happens if this rewriting of the index does not take place or if it takes place incompletely. Programs do not always run to completion. A program designer needs to guard against power failures, the operator turning the machine off at the wrong time, and other such disasters. One of the dangers associated with reading an index into memory and then writing it out when the program is over is that the copy of the index on disk will be out of date and incorrect if the program is interrupted. It is imperative that a program contain at least the following two safeguards to protect against this kind of error:

- There should be a mechanism that permits the program to know when the index is out of date. One possibility involves setting a status flag as soon as the copy of the index in memory is changed. This status flag could be written into the header record of the index file on disk as soon as the index is read into memory and subsequently cleared when the index is rewritten. All programs could check the status flag before using an index. If the flag is found to be set, the program would know that the index is out of date.
- If a program detects that an index is out of date, the program must have access to a procedure that reconstructs the index from the data file. This should happen automatically and take place before any attempt is made to use the index.

Record Addition

Adding a new record to the data file requires that we also add an entry to the index. Adding to the data file itself uses `RecordFile<Recording>::Write`. The record key and the resulting record reference are then inserted into the index record using `TextIndex.Insert`.

Since the index is kept in sorted order by key, insertion of the new index entry probably requires some rearrangement of the index. In a way, the situation is similar to the one we face as we add records to a sorted data file. We have to shift or slide all the entries with keys that come in order after the key of the record we are inserting. The shifting opens up a space for the new entry. The big difference between the work we have to do on the index entries and the work required for a sorted data file is that the index is contained *wholly in memory.* All of the index rearrangement can be done without any file access. The implementation of `TextIndex::Insert` is given in file `textind.cpp` of Appendix G.

Record Deletion

In Chapter 6 we described a number of approaches to deleting records in variable-length record files that allow for the reuse of the space occupied by these records. These approaches are completely viable for our data file because, unlike a sorted data file, the records in this file need not be moved around to maintain an ordering on the file. This is one of the great advantages of an indexed file organization: we have rapid access to individual records by key without disturbing pinned records. In fact, the indexing itself pins all the records. The implementation of data record deletion is not included in this text but has been left as exercises.

Of course, when we delete a record from the data file, we must also delete the corresponding entry from our index, using `TextIndex::Delete`. Since the index is in memory during program execution, deleting the index entry and shifting the other entries to close up the space may not be an overly expensive operation. Alternatively, we could simply mark the index entry as deleted, just as we might mark the corresponding data record. Again, see `textind.cpp` for the implementation of `TextIndex::Delete`.

Record Updating

Record updating falls into two categories:

- *The update changes the value of the key field.* This kind of update can bring about a reordering of the index file as well as the data file. Conceptually, the easiest way to think of this kind of change is as a deletion followed by an insertion. This delete/insert approach can be implemented while still providing the program user with the view that he or she is merely changing a record.
- *The update does not affect the key field.* This second kind of update does not require rearrangement of the index file but may well involve reordering of the data file. If the record size is unchanged or decreased by the update, the record can be written directly into its old space. But if the record size is increased by the update, a new slot for the record will have to be found. In the latter case the starting address of the rewritten record must replace the old address in the corresponding `RecAddrs` element. Again, the delete/insert approach to maintaining the index can be used. It is also possible to implement an operation simply to change the `RecAddrs` member.

7.4.2 Class TextIndexedFile

Class `TextIndexedFile` is defined in Fig. 7.7 and in file `indfile.h` in Appendix G. It supports files of data objects with primary keys that are strings. As expected, there are methods: `Create`, `Open`, `Close`, `Read` (sequential and indexed), `Append`, and `Update`. In order to ensure the correlation between the index and the data file, the members that represent the index in memory (`Index`), the index file (`IndexFile`), and the data file (`DataFile`) are protected members. The only access to these members for the user is through the methods. `TextIndexedFile` is a template class so that data objects of arbitrary classes can be used.

```
template <class RecType>
class TextIndexedFile
{public:
   int Read (RecType & record); // read next record
   int Read (char * key, RecType & record); // read by key
   int Append (const RecType & record);
   int Update (char * oldKey, const RecType & record);
   int Create (char * name, int mode=ios::in|ios::out);
   int Open (char * name, int mode=ios::in|ios::out);
   int Close ();
   TextIndexedFile (IOBuffer & buffer,
      int keySize, int maxKeys = 100);
   ~TextIndexedFile (); // close and delete
protected:
   TextIndex Index;
   BufferFile IndexFile;
   TextIndexBuffer IndexBuffer;
   RecordFile<RecType> DataFile;
   char * FileName; // base file name for file
   int SetFileName(char * fileName,
      char *& dataFileName, char *& indexFileName);
};
// The template parameter RecType must have the following method
//    char * Key()
```

Figure 7.7 Class TextIndexedFile

As an example, consider `TextIndexedFile::Append`:

```
template <class RecType>
int TextIndexedFile<RecType>::Append (const RecType &
record)
{
   char * key = record.Key();
   int ref = Index.Search(key);
   if (ref != -1) // key already in file
      return -1;
   ref = DataFile . Append(record);
   int result = Index . Insert (key, ref);
   return ref;
}
```

The `Key` method is used to extract the key value from the record. A search of the index is used to determine if the key is already in the file. If not, the record is appended to the data file, and the resulting address is inserted into the index along with the key.

7.4.3 Enhancements to Class TextIndexedFile

Other Types of Keys

Even though class `TextIndexedFile` is parameterized to support a variety of data object classes, it restricts the key type to string (`char *`). It is not hard to produce a template class `SimpleIndex` with a parameter for the key type. Often, changing a class to a template class requires adding a template parameter and then simply replacing a class name with the parameter name—in this case, replacing `char *` by `keytype`. However, the peculiar way that strings are implemented in C and C++ makes this impossible. Any array in C and C++ is represented by a pointer, and equality and assignment operators are defined accordingly. Since a string is an array, string assignment is merely pointer assignment. If you review the methods of class `TextIndex`, you will see that `strcmp` is used to test for key equality, and `strcpy` is used for key assignment. In order to produce a template index class, the dependencies on `char *` must be removed. The template class `SimpleIndex` is included in files `simpind.h` and `simpind.tc` in Appendix G. It is used as the basis for the advanced indexing strategies of Chapter 9.

In C++, assignment and other operators can be overloaded only for class objects, not for predefined types like `int` and `char *`. In order to

use a template index class for string keys, a class `String` is needed. Files `strclass.h` and `strclass.cpp` of Appendix G have the definition and implementation of this class, which was first mentioned in Chapter 1. Included in this class are a copy constructor, a constructor with a `char *` parameter, overloaded assignment and comparison operators, and a conversion operator to `char *` (`operator char*`). The following code shows how `String` objects and C strings become interchangeable:

```
String strObj(10); char * strArray[11]; // strings of <=10 chars
strObj = strArray; // uses String::String(char *)
strArray = strObj; // uses String::operator char * ();
```

The first assignment is implemented by constructing a temporary `String` object using the `char *` constructor and then doing `String` assignment. In this way the constructor acts like a conversion operator to class `String`. The second assignment uses the conversion operator from class `String` to convert the `String` object to a simple C string.

Data Object Class Hierarchies

So far, we have required that every object stored in a `RecordFile` must be of the same type. Can the I/O classes support objects that are of a variety of types but all from the same type hierarchy? If the type hierarchy supports virtual pack methods, the `Append` and `Update` will correctly add records to indexed files. That is, if `BaseClass` supports `Pack`, `Unpack`, and `Key`, the class `TextIndexedFile<BaseClass>` will correctly output objects derived from `BaseClass`, each with its appropriate `Pack` method.

What about `Read`? The problem here is that in a virtual function call, it is the type of the calling object that determines which method to call. For example, in this code it is the type of the object referenced by `Obj` (`*Obj`) that determines which `Pack` and `Unpack` are called:

```
BaseClass * Obj = new Subclass1;
Obj->Pack(Buffer); Obj->Unpack(Buffer); // virtual function calls
```

In the case of the `Pack`, this is correct. Information from `*Obj`, of type `Subclass1`, is transferred to `Buffer`. However, in the case of `Unpack`, it is a transfer of information from `Buffer` to `*Obj`. If `Buffer` has been filled from an object of class `Subclass2` or `BaseClass`, the unpacking cannot be done correctly. In essence, it is the source of information (contents of the buffer) that determines the type of

the object in the `Unpack`, not the memory object. The virtual function call does not work in this case. An object from a file can be read only into a memory object of the correct type.

A reliable solution to the read problem—that is, one that does not attempt to read a record into an object of the wrong type—is not easy to implement in C++. It is not difficult to add a type identifier to each data record. We can add record headers in much the same fashion as file headers. However, the read operation must be able to determine reliably the type of the target object. There is no support in C++ for guaranteeing accurate type identification of memory objects.

Multirecord Index Files

Class `TextIndexedFile` requires that the entire index fit in a single record. The maximum number of records in the file is fixed when the file is created. This is obviously an oversimplification of the index structure and a restriction on its utility. Is it worth the effort to extend the class so that this restriction is eliminated?

It would be easy to modify class `TextIndexedFile` to allow the index to be an array of `TextIndex` objects. We could add protected methods `Insert`, `Delete`, and `Search` to manipulate the arrays of index objects. None of this is much trouble. However, as we will see in the following section and in Chapter 9, a sorted array of index objects, each with keys less than the next, does not provide a very satisfactory index for large files. For files that are restricted to a small number of records, class `TextIndexedFile` will work quite well as it is.

Optimization of Operations

The most obvious optimization is to use binary search in the `Find` method, which is used by `Search`, `Insert`, and `Remove`. This is very reasonable and is left as an exercise.

Another source of some improvement is to avoid writing the index record back to the index file when it has not been changed. The standard way to do this is to add a flag to the index object to signal when it has been changed. This flag is set to *false* when the record is initially loaded into memory and set to *true* whenever the index record is modified, that is, by the `Insert` and `Remove` methods. The `Close` method can check this flag and write the record only when necessary. This optimization gains importance when manipulating multirecord index files.

7.5 Indexes That Are Too Large to Hold in Memory

The methods we have been discussing—and, unfortunately, many of the advantages associated with them—are tied to the assumption that the index is small enough to be loaded into memory in its entirety. If the index is too large for this approach to be practical, then index access and maintenance must be done on secondary storage. With simple indexes of the kind we have been discussing, accessing the index on a disk has the following disadvantages:

- Binary searching of the index requires several seeks instead of taking place at memory speeds. Binary searching of an index on secondary storage is not substantially faster than the binary searching of a sorted file.
- Index rearrangement due to record addition or deletion requires shifting or sorting records on secondary storage. This is literally millions of times more expensive than performing these same operations in memory.

Although these problems are no worse than those associated with any file that is sorted by key, they are severe enough to warrant the consideration of alternatives. Any time a simple index is too large to hold in memory, you should consider using

- A *hashed* organization if access speed is a top priority; or
- A *tree-structured*, or multilevel, index, such as a *B-tree*, if you need the flexibility of both keyed access and ordered, sequential access.

These alternative file organizations are discussed at length in the chapters that follow. But, before writing off the use of simple indexes on secondary storage altogether, we should note that they provide some important advantages over the use of a data file sorted by key even if the index cannot be held in memory:

- A simple index makes it possible to use a binary search to obtain keyed access to a record in a variable-length record file. The index provides the service of associating a fixed-length and therefore binary-searchable record with each variable-length data record.
- If the index entries are substantially smaller than the data file records, sorting and maintaining the index can be less expensive than sorting and maintaining the data file. There is simply less information to move around in the index file.

- If there are pinned records in the data file, the use of an index lets us rearrange the keys without moving the data records.

There is another advantage associated with the use of simple indexes, one that we have not yet discussed. By itself, it can be reason enough to use simple indexes even if they do not fit into memory. Remember the analogy between an index and a library card catalog? The card catalog provides multiple views or arrangements of the library's collection, even though there is only one set of books arranged in a single order. Similarly, we can use multiple indexes to provide multiple views of a data file.

7.6 Indexing to Provide Access by Multiple Keys

One question that might reasonably arise at this point is: All this indexing business is pretty interesting, but who would ever want to find a recording using a key such as DG18807? What I want is a recording of Beethoven's Symphony No. 9.

Let's return to our analogy of our index as a library card catalog. Suppose we think of our primary key, the Label ID, as a kind of catalog number. Like the catalog number assigned to a book, we have taken care to make our Label ID unique. Now in a library it is very unusual to begin by looking for a book with a particular catalog number (for example, "I am looking for a book with a catalog number QA331T5 1959."). Instead, one generally begins by looking for a book on a particular subject, with a particular title, or by a particular author (for example, "I am looking for a book on functions," or "I am looking for *The Theory of Functions* by Titchmarsh."). Given the subject, author, or title, one looks in the card catalog to find the *primary key*, the catalog number.

Similarly, we could build a catalog for our record collection consisting of entries for album title, composer, and artist. These fields are *secondary key fields*. Just as the library catalog relates an author entry (secondary key) to a card catalog number (primary key), so can we build an index file that relates Composer to Label ID, as illustrated in Fig. 7.8.

Along with the similarities, there is an important difference between this kind of secondary key index and the card catalog in a library. In a library, once you have the catalog number you can usually go directly to the stacks to find the book since the books are arranged in order by catalog number. In other words, the books are sorted by primary key. The actual data records in our file, on the other hand, are *entry sequenced*.

Composer index

Secondary key	*Primary key*
BEETHOVEN	ANG3795
BEETHOVEN	DG139201
BEETHOVEN	DG18807
BEETHOVEN	RCA2626
COREA	WAR23699
DVORAK	COL31809
PROKOFIEV	LON2312
RIMSKY-KORSAKOV	MER75016
SPRINGSTEEN	COL38358
SWEET HONEY IN THE R	FF245

Figure 7.8
Secondary key index organized by composer.

Consequently, after consulting the composer index to find the Label ID, you must consult one additional index, our primary key index, to find the actual byte offset of the record that has this particular Label ID. Figure 7.9 shows part of the class definition for a secondary key index and a read function that searches a secondary key index for the primary key. It then uses the primary key to read an `IndexedFile`.

Clearly it is possible to relate secondary key references (for example, Beethoven) directly to a byte offset (241) rather than to a primary key (DG18807). However, there are excellent reasons for postponing this binding of a secondary key to a specific address for as long as possible. These reasons become clear as we discuss the way that fundamental file operations such as record deletion and updating are affected by the use of secondary indexes.

Record Addition

When a secondary index is present, adding a record to the file means adding an entry to the secondary index. The cost of doing this is very simi-

```
class SecondaryIndex
   An index in which the record reference is a string
public:
   int Insert (char * secondaryKey, char * primaryKey);
   char * Search (char * secondaryKey); // return primary key
      ...
};
template <class RecType>
int SearchOnSecondary (char * composer, SecondaryIndex index,
      IndexedFile<RecType> dataFile, RecType & rec)
{
   char * Key = index.Search (composer);
   // use primary key index to read file
   return dataFile . Read (Key, rec);
}
```

Figure 7.9 SearchOnSecondary: an algorithm to retrieve a single record from a recording file through a secondary key index.

lar to the cost of adding an entry to the primary index: either records must be shifted, or a vector of pointers to structures needs to be rearranged. As with primary indexes, the cost of doing this decreases greatly if the secondary index can be read into memory and changed there.

Note that the key field in the secondary index file is stored in canonical form (all of the composers' names are capitalized), since this is the form we want to use when we are consulting the secondary index. If we want to print out the name in normal, mixed upper- and lowercase form, we can pick up that form from the original data file. Also note that the secondary keys are held to a fixed length, which means that sometimes they are truncated. The definition of the canonical form should take this length restriction into account if searching the index is to work properly.

One important difference between a secondary index and a primary index is that a secondary index can contain duplicate keys. In the sample index illustrated in Fig. 7.10, there are four records with the key BEETHOVEN. Duplicate keys are, of course, grouped together. Within this group, they should be ordered according to the values of the reference fields. In this example, that means placing them in order by Label ID. The reasons for this second level of ordering become clear a little later, as we discuss retrieval based on combinations of two or more secondary keys.

Title index

Secondary key	*Primary key*
COQ D'OR SUITE	MER75016
GOOD NEWS	FF245
NEBRASKA	COL38358
QUARTET IN C SHARP M	RCA2626
ROMEO AND JULIET	LON2312
SYMPHONY NO. 9	ANG3795
SYMPHONY NO. 9	COL31809
SYMPHONY NO. 9	DG18807
TOUCHSTONE	WAR23699
VIOLIN CONCERTO	DG139201

Figure 7.10
Secondary key index organized by recording title.

Record Deletion

Deleting a record usually implies removing all references to that record in the file system. So removing a record from the data file would mean removing not only the corresponding entry in the primary index but also all of the entries in the secondary indexes that refer to this primary index entry. The problem with this is that secondary indexes, like the primary index, are maintained in sorted order by key. Consequently, deleting an entry would involve rearranging the remaining entries to close up the space left open by deletion.

This delete-all-references approach would indeed be advisable if the secondary index referenced the data file directly. If we did not delete the secondary key references and if the secondary keys were associated with actual byte offsets in the data file, it could be difficult to tell when these references were no longer valid. This is another instance of the pinned-record problem. The reference fields associated with the secondary keys would be pointing to byte offsets that could, after deletion and subsequent space reuse in the data file, be associated with different data records.

But we have carefully avoided referencing actual addresses in the secondary key index. After we search to find the secondary key, we do another search, this time on primary key. Since the primary index *does* reflect changes due to record deletion, a search for the primary key of a record that has been deleted will fail, returning a record-not-found condition. In a sense, the updated primary key index acts as a kind of final check, protecting us from trying to retrieve records that no longer exist.

Consequently, one option that is open to us when we delete a record from the data file is to modify and rearrange only the primary key index. We could safely leave intact the references to the deleted record that exist in the secondary key indexes. Searches starting from a secondary key index that lead to a deleted record are caught when we consult the primary key index.

If there are a number of secondary key indexes, the savings that results from not having to rearrange all of these indexes when a record is deleted can be substantial. This is especially important when the secondary key indexes are kept on secondary storage. It is also important with an interactive system in which the user is waiting at a terminal for the deletion operation to complete.

There is, of course, a cost associated with this shortcut: deleted records take up space in the secondary index files. In a file system that undergoes few deletions, this is not usually a problem. In a somewhat more volatile file structure, it is possible to address the problem by periodically removing from the secondary index files all entries that contain references that are no longer in the primary index. If a file system is so volatile that even periodic purging is not adequate, it is probably time to consider another index structure, such as a B-tree, that allows for deletion without having to rearrange a lot of records.

Record Updating

In our discussion of record deletion, we find that the primary key index serves as a kind of protective buffer, insulating the secondary indexes from changes in the data file. This insulation extends to record updating as well. If our secondary indexes contain references directly to byte offsets in the data file, then updates to the data file that result in changing a record's physical location in the file also require updating the secondary indexes. But, since we are confining such detailed information to the primary index, data file updates affect the secondary index only when they change either the primary or the secondary key. There are three possible situations:

- *Update changes the secondary key:* if the secondary key is changed, we may have to rearrange the secondary key index so it stays in sorted order. This can be a relatively expensive operation.
- *Update changes the primary key:* this kind of change has a large impact on the primary key index but often requires that we update only the affected reference field (*Label ID* in our example) in all the secondary indexes. This involves searching the secondary indexes (on the unchanged secondary keys) and rewriting the affected fixed-length field. It does not require reordering of the secondary indexes unless the corresponding secondary key occurs more than once in the index. If a secondary key does occur more than once, there may be some local reordering, since records having the same secondary key are ordered by the reference field (primary key).
- *Update confined to other fields:* all updates that do not affect either the primary or secondary key fields do not affect the secondary key index, even if the update is substantial. Note that if there are several secondary key indexes associated with a file, updates to records often affect only a subset of the secondary indexes.

7.7 Retrieval Using Combinations of Secondary Keys

One of the most important applications of secondary keys involves using two or more of them in combination to retrieve special subsets of records from the data file. To provide an example of how this can be done, we will extract another secondary key index from our file of recordings. This one uses the recording's *title* as the key, as illustrated in Fig. 7.10. Now we can respond to requests such as

- Find the recording with Label ID COL38358 (primary key access);
- Find all the recordings of Beethoven's work (secondary key=composer); and
- Find all the recordings titled "Violin Concerto" (secondary key=title).

What is more interesting, however, is that we can also respond to a request that *combines* retrieval on the composer index with retrieval on the title index, such as: Find all recordings of Beethoven's Symphony No. 9. Without the use of secondary indexes, this kind of request requires a sequential search through the entire file. Given a file containing thousands,

or even hundreds, of records, this is a very expensive process. But, with the aid of secondary indexes, responding to this request is simple and quick.

We begin by recognizing that this request can be rephrased as a Boolean *and* operation, specifying the intersection of two subsets of the data file:

```
Find all data records with:
   composer = 'BEETHOVEN' and title = 'SYMPHONY NO. 9'
```

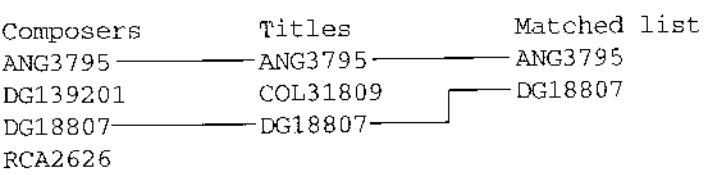
We begin our response to this request by searching the composer index for the list of Label IDs that identify recordings with Beethoven as the composer. This yields the following list of Label IDs:

```
ANG3795
DG139201
DG18807
RCA2626
```

Next we search the title index for the Label IDs associated with records that have SYMPHONY NO. 9 as the title key:

```
ANG3795
COL31809
DG18807
```

Now we perform the Boolean *and*, which is a match operation, combining the lists so only the members that appear in *both* lists are placed in the output list.



We give careful attention to algorithms for performing this kind of match operation in Chapter 8. Note that this kind of matching is much easier if the lists that are being combined are in sorted order. That is the reason that, when we have more than one entry for a given secondary key, the records are ordered by the primary key reference fields.

Finally, once we have the list of primary keys occurring in both lists, we can proceed to the primary key index to look up the addresses of the data file records. Then we can retrieve the records:

```
ANG | 3795 | Symphony No. 9 | Beethoven | Guilini
DG | 18807 | Symphony No. 9 | Beethoven | Karajan
```

This is the kind of operation that makes computer-indexed file systems useful in a way that far exceeds the capabilities of manual systems. We have only one copy of each data file record, and yet, working through the secondary indexes, we have multiple views of these records: we can look at them in order by title, by composer, or by any other field that interests us. Using the computer's ability to combine sorted lists rapidly, we can even combine different views, retrieving *intersections* (Beethoven *and* Symphony No. 9) or *unions* (Beethoven *or* Prokofiev Symphony No. 9) of these views. And since our data file is entry sequenced, we can do all of this without having to sort data file records and can confine our sorting to the smaller index records that can often be held in memory.

Now that we have a general idea of the design and uses of secondary indexes, we can look at ways to improve these indexes so they take less space and require less sorting.

7.8 Improving the Secondary Index Structure: Inverted Lists

The secondary index structures that we have developed so far result in two distinct difficulties:

- We have to rearrange the index file *every time* a new record is added to the file, even if the new record is for an existing secondary key. For example, if we add another recording of Beethoven's Symphony No. 9 to our collection, both the composer and title indexes would have to be rearranged, even though both indexes already contain entries for secondary keys (but not the Label IDs) that are being added.
- If there are duplicate secondary keys, the secondary key field is repeated for each entry. This wastes space because it makes the files larger than necessary. Larger index files are less likely to fit in memory.

7.8.1 A First Attempt at a Solution

One simple response to these difficulties is to change the secondary index structure so it associates an *array* of references with each secondary key.

For example, we might use a record structure that allows us to associate up to four Label ID reference fields with a single secondary key, as in

BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626

Figure 7.11 provides a schematic example of how such an index would look if used with our sample data file.

The major contribution of this revised index structure is its help in solving our first difficulty: the need to rearrange the secondary index file every time a new record is added to the data file. Looking at Fig. 7.11, we can see that the addition of another recording of a work by Prokofiev does not require the addition of another record to the index. For example, if we add the recording

```
ANG  36193 Piano Concertos 3 and 5  Prokofiev Francois
```

we need to modify only the corresponding secondary index record by inserting a second Label ID:

PROKOFIEV	ANG36193	LON2312

Since we are not adding another record to the secondary index, there is no need to rearrange any records. All that is required is a rearrangement of the fields in the existing record for Prokofiev.

Although this new structure helps avoid the need to rearrange the secondary index file so often, it does have some problems. For one thing, it provides space for only four Label IDs to be associated with a given key. In the very likely case that more than four Label IDs will go with some key, we need a mechanism for keeping track of the extra Label IDs.

A second problem has to do with space usage. Although the structure does help avoid the waste of space due to the repetition of identical keys, this space savings comes at a potentially high cost. By extending the fixed length of each of the secondary index records to hold more reference fields, we might easily lose more space to internal fragmentation than we gained by not repeating identical keys.

Since we don't want to waste any more space than we have to, we need to ask whether we can improve on this record structure. Ideally, what we would like to do is develop a new design, a revision of our revision, that

Revised composer index

Secondary key	*Set of primary key references*			
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
PROKOFIEV	LON2312			
RIMSKY-KORSAKOV	MER75016			
SPRINGSTEEN	COL38358			
SWEET HONEY IN THE R	FF245			

Figure 7.11 Secondary key index containing space for multiple references for each secondary key.

- Retains the attractive feature of not requiring reorganization of the secondary indexes for every new entry to the data file;
- Allows more than four Label IDs to be associated with each secondary key; and
- Eliminates the waste of space due to internal fragmentation.

7.8.2 A Better Solution: Linking the List of References

Files such as our secondary indexes, in which a secondary key leads to a set of one or more primary keys, are called *inverted lists*. The sense in which the list is inverted should be clear if you consider that we are working our way backward from a secondary key to the primary key to the record itself.

The second word in the term "inverted list" also tells us something important: we are, in fact, dealing with a *list* of primary key references. Our revised secondary index, which collects a number of Label IDs for each secondary key, reflects this list aspect of the data more directly than our initial secondary index. Another way of conceiving of this list aspect of our inverted list is illustrated in Fig. 7.12.

As Fig. 7.12 shows, an ideal situation would be to have each secondary key point to a different list of primary key references. Each of these lists

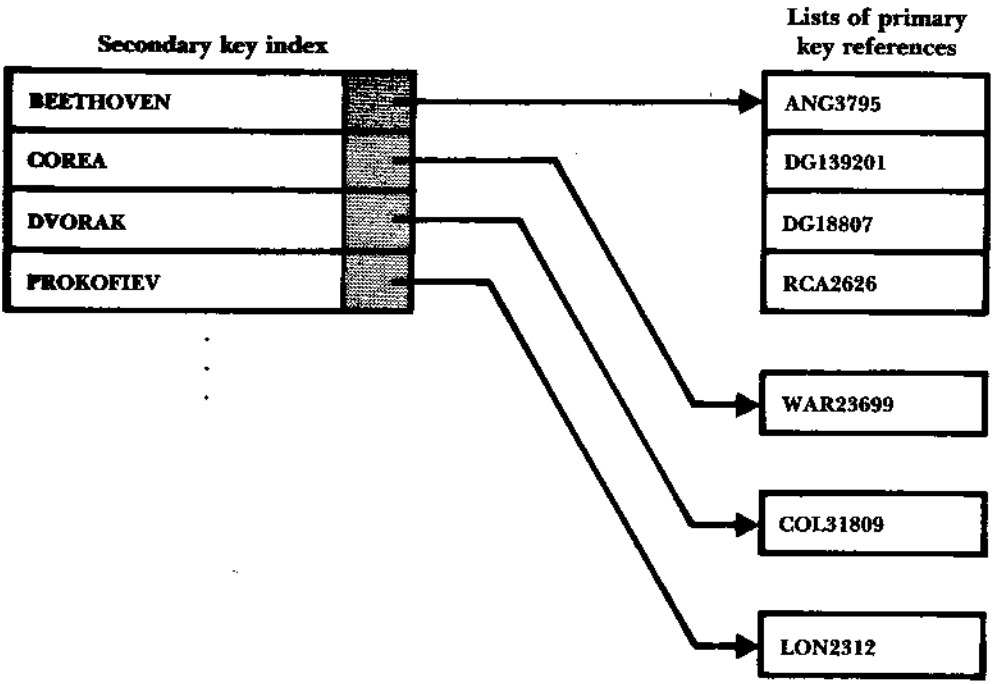
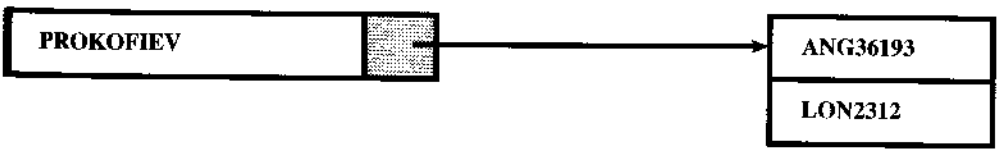


Figure 7.12 Conceptual view of the primary key reference fields as a series of lists.

could grow to be just as long as it needs to be. If we add the new Prokofiev record, the list of Prokofiev references becomes



Similarly, adding two new Beethoven recordings adds just two additional elements to the list of references associated with the Beethoven key. Unlike our record structure which allocates enough space for four Label IDs for each secondary key, the lists could contain hundreds of references, if needed, while still requiring only one instance of a secondary key. On the other hand, if a list requires only one element, then no space is lost to internal fragmentation. Most important, we need to rearrange only the file of secondary keys if a new composer is added to the file.

How can we set up an unbounded number of different lists, each of varying length, without creating a large number of small files? The simplest way is through the use of linked lists. We could redefine our secondary index so it consists of records with two fields—a secondary key field and a field containing the relative record number of the first corresponding primary key reference (Label ID) in the inverted list. The actual primary key references associated with each secondary key would be stored in a separate, entry-sequenced file.

Given the sample data we have been working with, this new design would result in a secondary key file for composers and an associated Label ID file that are organized as illustrated in Fig. 7.13. Following the links from the list of references associated with Beethoven helps us see how the Label ID List file is organized. We begin, of course, by searching the secondary key index of composers for Beethoven. The record that we find points us to relative record number (RRN) 3 in the Label ID List file. Since this is a fixed-length file, it is easy to jump to RRN 3 and read in its Label ID

Improved revision of the composer index

Secondary Index file

0	BEETHOVEN	3
1	COREA	2
2	DVORAK	7
3	PROKOFIEV	10
4	RIMSKY-KORSAKOV	6
5	SPRINGSTEEN	4
6	SWEET HONEY IN THE R	9

Label ID List file

0	LON2312	−1
1	RCA2626	−1
2	WAR23699	−1
3	ANG3795	8
4	COL38358	−1
5	DG18807	1
6	MER75016	−1
7	COL31809	−1
8	DG139201	5
9	FF245	−1
10	ANG36193	0

Figure 7.13 Secondary key index referencing linked lists of primary key references.

(ANG3795). Associated with this Label ID is a link to a record with RRN 8. We read in the Label ID for that record, adding it to our list (ANG379 DG139201). We continue following links and collecting Label IDs until the list looks like this:

```
ANG3795   DG139201   DG18807   RCA2626
```

The link field in the last record read from the Label ID List file contains a value of -1. As in our earlier programs, this indicates end-of-list, so we know that we now have all the Label ID references for Beethoven records.

To illustrate how record addition affects the Secondary Index and Label ID List files, we add the Prokofiev recording mentioned earlier:

```
ANG  36193 Piano Concertos 3 and 5  Prokofiev Francois
```

You can see (Fig. 7.13) that the Label ID for this new recording is the last one in the Label ID List file, since this file is entry sequenced. Before this record is added, there is only one Prokofiev recording. It has a Label ID of LON2312. Since we want to keep the Label ID Lists in order by ASCII character values, the new recording is inserted in the list for Prokofiev so it logically precedes the LON2312 recording.

Associating the Secondary Index file with a new file containing linked lists of references provides some advantages over any of the structures considered up to this point:

- The only time we need to rearrange the Secondary Index file is when a new composer's name is added or an existing composer's name is changed (for example, it was misspelled on input). Deleting or adding recordings for a composer who is already in the index involves changing only the Label ID List file. Deleting *all* the recordings for a composer could be handled by modifying the Label ID List file while leaving the entry in the Secondary Index file in place, using a value of -1 in its reference field to indicate that the list of entries for this composer is empty.
- In the event that we need to rearrange the Secondary Index file, the task is quicker now since there are fewer records and each record is smaller.
- Because there is less need for sorting, it follows that there is less of a penalty associated with keeping the Secondary Index files off on secondary storage, leaving more room in memory for other data structures.

- The Label ID List file is entry sequenced. That means that it *never* needs to be sorted.
- Since the Label ID List file is a fixed-length record file, it would be very easy to implement a mechanism for reusing the space from deleted records, as described in Chapter 6.

There is also at least one potentially significant disadvantage to this kind of file organization: the Label IDs associated with a given composer are no longer guaranteed to be grouped together physically. The technical term for such "togetherness" is *locality*. With a linked, entry-sequenced structure such as this, it is less likely that there will be locality associated with the logical groupings of reference fields for a given secondary key. Note, for example, that our list of Label IDs for Prokofiev consists of the very last and the very first records in the file. This lack of locality means that picking up the references for a composer with a long list of references could involve a large amount of *seeking* back and forth on the disk. Note that this kind of seeking would not be required for our original Secondary Index file structure.

One obvious antidote to this seeking problem is to keep the Label ID List file in memory. This could be expensive and impractical, given many secondary indexes, except for the interesting possibility of using the same Label ID List file to hold the lists for a number of Secondary Index files. Even if the file of reference lists were too large to hold in memory, it might be possible to obtain a performance improvement by holding only a part of the file in memory at a time, paging sections of the file in and out of memory as they are needed.

Several exercises at the end of the chapter explore these possibilities more thoroughly. These are very important problems, as the notion of dividing the index into pages is fundamental to the design of B-trees and other methods for handling large indexes on secondary storage.

7.9 Selective Indexes

Another interesting feature of secondary indexes is that they can be used to divide a file into parts and provide a selective view. For example, it is possible to build a *selective index* that contains only the titles of classical recordings in the record collection. If we have additional information about the recordings in the data file, such as the date the recording was released, we could build selective indexes such as "recordings released prior

to 1970" and "recordings since 1970." Such selective index information could be combined into Boolean *and* operations to respond to requests such as "List all the recordings of Beethoven's Ninth Symphony released since 1970." Selective indexes are sometimes useful when the contents of a file fall naturally and logically into several broad categories.

7.10 Binding

A recurrent and very important question that emerges in the design of file systems that use indexes is: *At what point is the key bound to the physical address of its associated record?*

In the file system we are designing in the course of this chapter, the *binding* of our primary keys to an address takes place *at the time the files are constructed.* The secondary keys, on the other hand, are bound to an address *at the time that they are used.*

Binding at the time of the file construction results in faster access. Once you have found the right index record, you have in hand the byte offset of the data record you are seeking. If we elected to bind our secondary keys to their associated records at the time of file construction so when we find the DVORAK record in the composer index we would know immediately that the data record begins at byte 338 in the data file, secondary key retrieval would be simpler and faster. The improvement in performance is particularly noticeable if both the primary and secondary index files are used on secondary storage rather than in memory. Given the arrangement we designed, we would have to perform a binary search of the composer index and then a binary search of the primary key index before being able to jump to the data record. Binding early, at file construction time, eliminates the need to search on the primary key.

The disadvantage of binding directly in the file, of *binding tightly,* is that reorganizations of the data file must result in modifications to all bound index files. This reorganization cost can be very expensive, particularly with simple index files in which modification would often mean shifting records. By postponing binding until execution time, when the records are being used, we are able to develop a secondary key system that involves a minimal amount of reorganization when records are added or deleted.

Another important advantage to postponing binding until a record is retrieved is that this approach is safer. As we see in the system that we set

up, associating the secondary keys with reference fields consisting of primary keys allows the primary key index to act as a kind of final check of whether a record is really in the file. The secondary indexes can afford to be wrong. This situation is very different if the secondary index keys contain addresses. We would then be jumping directly from the secondary key into the data file; the address would need to be right.

This brings up a related safety aspect: it is always more desirable to make important changes in one place rather than in many places. With a bind-at-retrieval-time scheme such as we developed, we need to remember to make a change in only one place, the primary key index, if we move a data record. With a more tightly bound system, we have to make many changes successfully to keep the system internally consistent, braving power failures, user interruptions, and so on.

When designing a new file system, it is better to deal with this question of binding *intentionally* and *early* in the design process rather than letting the binding just happen. In general, tight, in-the-data binding is most attractive when

- The data file is static or nearly so, requiring little or no adding, deleting, or updating of records; and
- Rapid performance during actual retrieval is a high priority.

For example, tight binding is desirable for file organization on a mass-produced, read-only optical disk. The addresses will never change because no new records can ever be added; consequently, there is no reason not to obtain the extra performance associated with tight binding.

For file applications in which record addition, deletion, and updating do occur, however, binding at retrieval time is usually the more desirable option. Postponing binding as long as possible usually makes these operations simpler and safer. If the file structures are carefully designed, and, in particular, if the indexes use more sophisticated organizations such as B-trees, retrieval performance is usually quite acceptable, even given the additional work required by a bind-at-retrieval system.

SUMMARY

We began this chapter with the assertion that indexing as a way of structuring a file is an alternative to sorting because records can be found *by key*. Unlike sorting, indexing permits us to perform *binary searches* for keys in variable-length record files. If the index can be held in memory, record

addition, deletion, and retrieval can be done much more quickly with an indexed, entry-sequenced file than with a sorted file.

Template classes in C++ provide support for sharing class definitions and code among a number of unrelated classes. Template classes are used in this chapter for class `RecordFile`, which supports I/O of data records without explicit packing and unpacking of buffers, and for general purpose index records in class `SimpleIndex`.

Support for sequential and indexed access to a data file is provided by the template class `TextIndexedFile`. It extends the capabilities of class `RecordFile` by adding indexed read, update, and append operations. Each modification of the data file is accompanied by the proper changes to the index. Each `TextIndexedFile` object is represented by an index record object in memory and two files, a data file and an index file. The `TextIndexedFile::Close` method writes the contents of the index record object into the index file and closes both files.

Indexes can do much more than merely improve on access time: they can provide us with new capabilities that are inconceivable with access methods based on sorted data records. The most exciting new capability involves the use of multiple secondary indexes. Just as a library card catalog allows us to regard a collection of books in author order, title order, or subject order, so index files allow us to maintain different views of the records in a data file. We find that not only can we use secondary indexes to obtain different views of the file but we can also combine the associated lists of primary key references and thereby combine particular views.

In this chapter we address the problem of how to rid our secondary indexes of two liabilities:

- The need to repeat duplicate secondary keys, and
- The need to rearrange the secondary indexes every time a record is added to the data file.

A first solution to these problems involves associating a fixed-size *vector* of reference fields with each secondary key. This solution results in an overly large amount of internal fragmentation but illustrates the attractiveness of handling the reference fields associated with a particular secondary key as a group, or *list*.

Our next iteration of solutions to our secondary index problems is more successful and much more interesting. We can treat the primary key references as an entry-sequenced file, forming the necessary lists through the use of *link fields* associated with each primary record entry. This allows us to create a secondary index file that, in the case of the composer index, needs rearrangement only when we add new composers to the data file.

The entry-sequenced file of linked reference lists never requires sorting. We call this kind of secondary index structure an *inverted list.*

There are also, of course, disadvantages associated with our new solution. The most serious disadvantage is that our file demonstrates less locality: lists of associated records are less likely to be physically adjacent. A good antidote to this problem is to hold the file of linked lists in memory. We note that this is made more plausible because a single file of primary references can link the lists for a number of secondary indexes.

As indicated by the length and breadth of our consideration of secondary indexing, multiple keys, and inverted lists, these topics are among the most interesting aspects of indexed access to files. The concepts of secondary indexes and inverted lists become even more powerful later, as we develop index structures that are themselves more powerful than the simple indexes we consider here. But, even so, we already see that for small files consisting of no more than a few thousand records, approaches to inverted lists that rely merely on simple indexes can provide a user with a great deal of capability and flexibility.

KEY TERMS

Binding. Binding takes place when a key is associated with a particular physical record in the data file. In general, binding can take place either during the preparation of the data file and indexes or during program execution. In the former case, called *tight binding,* the indexes contain explicit references to the associated physical data record. In the latter case, the connection between a key and a particular physical record is postponed until the record is retrieved in the course of program execution.

Entry-sequenced file. A file in which the records occur in the order that they are entered into the file.

Index. An index is a tool for finding records in a file. It consists of a *key field* on which the index is searched and a *reference field* that tells where to find the data file record associated with a particular key.

Inverted list. The term *inverted list* refers to indexes in which a key may be associated with a *list* of reference fields pointing to documents that contain the key. The secondary indexes developed toward the end of this chapter are examples of inverted lists.

Key field. The key field is the portion of an index record that contains the canonical form of the key that is being sought.

Locality. Locality exists in a file when records that will be accessed in a given temporal sequence are found in physical proximity to each other on the disk. Increased locality usually results in better performance, as records that are in the same physical area can often be brought into memory with a single *read* request to the disk.

Reference field. The reference field is the portion of an index record that contains information about where to find the data record containing the information listed in the associated key field of the index.

Selective index. A selective index contains keys for only a portion of the records in the data file. Such an index provides the user with a view of a specific subset of the file's records.

Simple index. All the index structures discussed in this chapter are simple indexes insofar as they are all built around the idea of an ordered, linear sequence of index records. All these simple indexes share a common weakness: adding records to the index is expensive. As we see later, tree-structured indexes provide an alternate, more efficient solution to this problem.

Template class. A C++ class that is parameterized, typically with class (or type) parameters. Templates allow a single class definition to be used to construct a family of different classes, each with different arguments for the parameters.

FURTHER READINGS

We have much more to say about indexing in later chapters, where we take up the subjects of tree-structured indexes and indexed sequential file organizations. The topics developed in the current chapter, particularly those relating to secondary indexes and inverted files, are also covered by many other file and data structure texts. The few texts that we list here are of interest because they either develop certain topics in more detail or present the material from a different viewpoint.

Wiederhold (1983) provides a survey of many of the index structures we discuss, along with a number of others. His treatment is more mathematical than that provided in our text. Tremblay and Sorenson (1984) provide a comparison of inverted list structures with an alternative organization called *multilist* files. M. E. S. Loomis (1989) provides a similar discussion, along with some examples oriented toward COBOL users. Kroenke (1998) discuss inverted lists in the context of their application in information retrieval systems.

EXERCISES

1. Until now, it was not possible to perform a binary search on a variable-length record file. Why does indexing make binary search possible? With a fixed-length record file it *is* possible to perform a binary search. Does this mean that indexing need not be used with fixed-length record files?
2. Why is `Title` not used as a primary key in the `Recording` file described in this chapter? If it were used as a secondary key, what problems would have to be considered in deciding on a canonical form for titles?
3. What is the purpose of keeping an out-of-date-status flag in the header record of an index? In a multiprogramming environment, this flag might be found to be set by one program because another program is in the process of reorganizing the index. How should the first program respond to this situation?
4. Consult a reference book on C++ to determine how template classes like `RecordFile` are implemented. How does the compiler process the method bodies of a template class? How does the compiler process template instantiations?
5. Explain how the use of an index pins the data records in a file.
6. When a record in a data file is updated, corresponding primary and secondary key indexes may or may not have to be altered, depending on whether the file has fixed- or variable-length records, and depending on the type of change made to the data record. Make a list of the different updating situations that can occur, and explain how each affects the indexes.
7. Discuss the problem that occurs when you add the following recording to the recordings file, assuming that the composer index shown in Fig. 7.11 is used. How might you solve the problem without substantially changing the secondary key index structure?

   ```
   LON    1259  Fidelio      Beethoven    Maazel
   ```

8. How are the structures in Fig. 7.13 changed by the addition of the recording

   ```
   LON    1259  Fidelio      Beethoven    Maazel
   ```

9. Suppose you have the data file described in this chapter, but it's greatly expanded, with a primary key index and secondary key indexes

organized by composer, artist, and title. Suppose that an inverted list structure is used to organize the secondary key indexes. Give step-by-step descriptions of how a program might answer the following queries:

a. List all recordings of Bach or Beethoven, and
b. List all recordings by Perleman of pieces by Mozart or Joplin.

10. Using the program `makerec.cpp`, create a file of recordings. Make a file dump of the file and find the size and contents of the header as well asand the starting address and the size for each record.

11. Use the program `makeind.cpp` to create an index file for the recording file created by program `makerec.cpp`. Using a file dump, find the size and contents of the header, the address and size of the record, and the contents of the record.

12. The method and timing of binding affect two important attributes of a file system—speed and flexibility. Discuss the relevance of these attributes, and the effect of binding time on them, for a hospital patient datainformation system designed to provide information about current patients by patient name, patient ID, location, medication, doctor or doctors, and illness.

PROGRAMMING AND DESIGN EXERCISES

13. Add method(s) to class `TextIndex` to support iterating through the index in key order. One possible strategy is to define two methods:

```
int FirstRecAddr (); // return reference for the smallest key
int NextRecAddr (); // return reference for the next key
```

Implementation of these methods can be supported by adding members to the class.

14. Write a program to print the records of a `Recording` file in key order. One way to implement this program is to read all the records of the file and create an index record in memory and then iterate through the index in key order and read and print the records. Test the program on the file produced by `makerec.cpp`.

15. Write a program to print the records of a file of type `RecordFile<Recording>` in key order. Test the program on the file produced by `makeind.cpp`.

16. Modify the method `TextIndex::Search` to perform a binary search on the key array.
17. Implement the Remove methods of class `TextIndexedFile`.
18. Extend class `TextIndexedFile` to support the creation of an indexed file from a simple data file. That is, add a method that initializes a `TextIndexedFile` object by opening and reading the existing data file and creating an index from the records in the file.
19. As a major programming project, create a class hierarchy based on `Recording` that has different information for different types of recordings. Develop a class to support input and output of records of these types. The class should be consistent with the style described in the part of Section 7.4.3 about data object class hierarchies. The `Unpack` methods must be sensitive to the type of object that is being initialized by the call.
20. Define and implement a class `SecondaryIndex` to support secondary indexes, as described in Section 7.6. Use this class to create a class `RecordingFile` that uses `RecordFile` as its base class to manage the primary index and the data file and has secondary indexes for the Composer and Artist fields.
21. When searching secondary indexes that contain multiple records for some of the keys, we do not want to find just *any* record for a given secondary key; we want to find the *first* record containing that key. Finding the first record allows us to read ahead, sequentially, extracting all of the records for the given key. Write a variation of a search method that returns the relative record number of the *first* record containing the given key.
22. Identify and eliminate memory leaks in the code of Appendix F.

PROGRAMMING PROJECT

This is the fifth part of the programming project. We add indexes to the data files created by the third part of the project in Chapter 4.

23. Use class `IndexedFile` (or `TextIndexedFile`) to create an index of a file of student objects, using student identifier as key. Write a driver program to create an index file from the student record file created by the program of part three of the programming project in Chapter 4.

24. Use class `IndexedFile` (or `TextIndexedFile`) to create an index of a file of course registration objects, using student identifier as key. Note that the student identifier is not unique in course registration files. Write a driver program to create an index file from the course registration record file created by the program of part three of the programming project in Chapter 4.
25. Write a program that opens an indexed student file and an indexed course registration file and retrieves information on demand. Prompt a user for a student identifier and print all objects that match it.
26. Develop a class that supports indexed access to course registration files by student identifier and by course identifier (secondary key). See Exercise 20 for an implementation of secondary indexes. Extend the program of Exercise 25 to allow retrieval of information about specific courses.
27. Extend the above projects to support update and deletion of student records and course registration records.

The next part of the programming project is in Chapter 8.