Chapter 7. Indexing

Ki-Joon Han
School of Computer Science & Engineering
Konkuk University
kjhan@db.konkuk.ac.kr

Chapter Outline

7.1 What is an Index?
7.2 A Simple Index for Entry-Sequenced Files
7.3 Using Template Classes in C++ for Object I/O
7.4 Object-Oriented Support for Indexed, Entry-Sequenced Files of Data Objects
7.5 Indexes That Are Too Large to Hold in Memory
7.6 Indexing to Provide Access by Multiple Keys
7.7 Retrieval Using Combinations of Secondary Keys
7.8 Improving the Secondary Index Structure: Inverted Lists
7.9 Selective Indexes
7.10 Binding

7.1 What is an Index ?

- Index
  - a table containing a list of topics(keys) and numbers of pages (reference fields)
  - simple index : simple arrays of structures that contain the keys and reference fields
  - impose order on a file without actually rearranging the file
    - solve the problem of pinned records
    - make record additions less expensive
  - provide multiple access paths to a file
  - give us keyed access to variable-length record files
- Terminology
  - file system's view of the elements : *records*
  - application's view of the objects : *recordings*

7.2 A Simple Index for Entry-Sequenced Files (1/7)

- Entry-sequenced file
  - records occur in the order that they are entered into the file
  - data records can be variable or fixed length

- Appendix G
  - Recording.h and Recording.cpp
    - define class Recording
  - Makerec.cpp
    - use classes DemlimFieldBuffer and BufferFile to create the file of Recording objects

7.2 A Simple Index for Entry-Sequenced Files (2/7)

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.

7.2 A Simple Index for Entry-Sequenced Files (3/7)

- Rapid keyed access

(1) Sort, then binary searching

- binary searching is not possible in a variable-length record file
  - direct access by RRN is not possible
  - no way to know where the middle record is

(2) Construct an index for the file

- key + reference field
- Textind.h and Textind.cpp in Appendix G

7.2 A Simple Index for Entry-Sequenced Files (4/7)

Index			Recording file
Key	*Reference field*	*Address of record*	*Actual data record*
ANG3795	152	17	LON | 2312 | Romeo and Juliet | Prokofiev | ...
COL31809	338	62	RCA | 2626| Quartet in C Sharp Minor | Beethoven | ...
COL38358	196	117	WAR | 23699 | Touchstone | Corea | ...
DG139201	382	152	ANG | 3795 | Symphony No. 9 | Beethoven | ...
DG18807	241	196	COL | 38358 | Nebraska | Springsteen | ...
FF245	427	241	DG | 18807 | Symphony No. 9 | Beethoven | ...
LON2312	17	285	MER | 75016 | Coq d'Or Suite | Rimsky-Korsakov | ...
MER75016	285	338	COL | 31809 | Symphony No. 9 | Dvorak | ...
RCA2626	62	382	DG | 139201 | Violin Concerto | Beethoven | ...
WAR23699	117	427	FF | 245 | Good News |Sweet Honey in the Rock | ...

Figure 7.3 Index of the sample recording file.

7.2 A Simple Index for Entry-Sequenced Files (5/7)

- Structure of the index file
  - key + reference field (byte-offset field)
  - fixed-length record file in which each record has two fixed-length fields
  - index is sorted, whereas the data file is not
  - considerably easier to work with than the data file
  - likely to be much smaller than the data file

- Structure of the data file
  - data file is entry sequenced

  => make record addition and file maintenance simple

7.2 A Simple Index for Entry-Sequenced Files (6/7)

- Class TextIndex : index data and index operations

```
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; // return i of Keys[i]
    int Init (int maxKeys, int unique);
    int Unique;       //if true, each key must be unique in the index
};
```

7.2 A Simple Index for Entry-Sequenced Files (6/7)

```
int TextIndex :: Find (const char * key) const
{
        for (int i = 0; i < NumKeys; i++)
                if (strcmp(Keys[i], key)==0) return i;// key found
                else if (strcmp(Keys[i], key)>0) return -1;// not found
        return -1;// not found
}

int TextIndex :: Search (const char * key) const
{
        int index = Find (key);
        if (index < 0) return index;
        return RecAddrs[index];
}
```

7.2 A Simple Index for Entry-Sequenced Files (6/7)

```
int TextIndex :: Insert (const char * key, int recAddr)
{
        int i;
        int index = Find (key);
        if (Unique && index >= 0) return 0; // key already in
        if (NumKeys == MaxKeys) return 0; //no room for another key
        for (i = NumKeys-1; i >= 0; i--)
        {
                if (strcmp(key, Keys[i])>0) break; // insert into location i+1
                Keys[i+1] = Keys[i];
                RecAddrs[i+1] = RecAddrs[i];
        }
        Keys[i+1] = strdup(key);
        RecAddrs[i+1] = recAddr;
        NumKeys ++;
        return 1;
}
```

7.2 A Simple Index for Entry-Sequenced Files (6/7)

```
int TextIndex :: Insert (const char * key, int recAddr)
{
        int i;
        int index = Find (key);
        if (Unique && index >= 0) return 0; // key already in
        if (NumKeys == MaxKeys) return 0; //no room for another key
        for (i = NumKeys-1; i >= 0; i--)
        {
                if (strcmp(key, Keys[i])>0) break; // insert into location i+1
                Keys[i+1] = Keys[i];
                RecAddrs[i+1] = RecAddrs[i];
        }
        Keys[i+1] = strdup(key);
        RecAddrs[i+1] = recAddr;
        NumKeys ++;
        return 1;
}
```

Keys[i]

0	1
1	3
2	4
3	6
4	8
5	9
6	12
7	14
8	

Key =7

7.2 A Simple Index for Entry-Sequenced Files (6/7)

```
int TextIndex :: Remove (const char * key)
{
        int index = Find (key);
        if (index < 0) return 0; // key not in index
        for (int i = index; i < NumKeys; i++)
        {
                Keys[i] = Keys[i+1];
                RecAddrs[i] = RecAddrs[i+1];
        }
        NumKeys --;
        return 1;
```

Key =6

Keys[i]

0	1
1	3
2	4
3	6
4	8
5	9
6	12
7	14
8	

7.2 A Simple Index for Entry-Sequenced Files (7/7)

- 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()); //access to IOBuffer
    return result;
}
```

- use index class to retrieve a single record by key from a recording file
- put together the index search, file read, and buffer unpack operations into a single function

4.5 An Object-Oriented Class for Record Files(참고용)

```
Class BufferFile
{public:
 BufferFile (IOBuffer &);                      //create with a buffer
 int Open (char * filename, int MODE);  //open an existing file
 int Create(char * filename, ..);            //create a new file
 int Close();
 int Rewind();                           // reset to the first record
 int Read (int recaddr = -1);   // read a record into a buffer
 int Write(int recaddr = -1);   // write the buffer contents
 int Delete(int recaddr = -1);  //delete a record from a file
 int Append();                          // write the current buffer at the end of file
protected:
 IOBuffer & Buffer; // reference to the file's buffer
 fstream File;           // the C++ stream of the file
 int HeaderSize;       // size of header
 int ReadHeader();
 int WriteHeader();
};
DelimFieldBuffer buffer; // a buffer is created
BufferFile file (buffer); // BufferFile object file is attached
file.Open (myfile);
file.Read ( ); // buffer contains the packed record
myobject.Unpack(buffer); // put the record into myobject
```

7.3 Using Template Classes in C++ for Object I/O (1/4)

- Good O-O design for a file of objects
  - provide operations to read and write data objects without the intermediate step of packing and unpacking buffers

- Use of C++ template

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

  - the objects are different
  - they use different unpacking methods
  - can not solve with the virtual function

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

  - pack, unpack methods of Person, Recording are used for pFile and rFile, respectively

7.3 Using Template Classes in C++ for Object I/O (2/4)

- Template class RecordFile
  - reads an *object* of some class and writes it to a file
  - the use of buffers is hidden inside the class

```
template <class RecType>
class RecordFile: public BufferFile
{public:
    int Read (RecType & record, int recaddr = -1); //file-> RecType
    int Write (const RecType & record, int recaddr = -1);//RecType->file
    int Delete (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
```

7.3 Using Template Classes in C++ for Object I/O (3/4)

- RecordFile::Read

```
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;
}
```

- use the Read() method of BufferFile
- use the Unpack() method of RecType
  - rFile.Read(r) calls Recording::Unpack()
  - pFile.Read(p) calls Person::Unpack()

7.3 Using Template Classes in C++ for Object I/O (4/4)

- Use of Template class RecordFile for Recording
  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);**

- Example
  - open a file, read and write objects of class Recording

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

7.4 O-O Support for Indexed, Entry-Sequenced Files of Data Objects (1/2)

- Class IndexedFile
  - add indexed access to sequential access provided by class RecordFile
  - extend RecordFile with Write, Append, and Delete methods that maintain a primary key index of the data file and Read method that supports access to object by key
- Two classes
  - TextIndex: support maintenance and search by primary key
  - RecordFile: support create, open, and close for files as well as read and write for data objects

7.4 O-O Support for Indexed, Entry-Sequenced Files of Data Objects (2/2)

- Two issues of primary key index as a memory object
  - how to make a persistent index of a file
  - 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 Index File (1/6)

I. Creating the Files

- Create the index file and the data file as empty files and write headers to both files
  - use the Create method implemented in class BufferFile
- Data file
  - an object of class RecordFile <Recording>
- Index file
  - an object of class BufferFile of fixed-size records
- Makeind.cpp in Appendix G
  - create an index file from a file of recordings

7.4.1 Operations Required to Maintain an Index File (2/6)

Ⅱ. Loading the Index into Memory

- Making of files of index objects in Open operation
  - supported in the class IOBuffer
  - store the full index in a single object (single record index file)

    => multiple record index file
- Class TextIndexBuffer
  - a derived class of FixedIndexBuffer to support reading and writing of index objects
  - include pack and unpack methods for index objects
- Tindbuff.h and Tindbuff.cpp in Appendix G
  - full implementation of class TextIndexBuffer

7.4.1 Operations Required to Maintain an Index File (3/6)

Ⅲ. Rewriting the Index File from Memory

- Close operation on an IndexedFile
  - Use the Rewind and Write operations of class BufferFile
- If rewriting of the index does not take place, or takes place incompletely
  - copy of the index on disk will be out of date and incorrect
- Mechanism to know the index (in disk) is out of date

1. involve setting a status flag (Header record) as soon as the copy of the index in memory is changed
2. If rewriting of the index is finished, unset the status flag
3. Open operation checks the status flag before using an index
4. if an index is out of date, then reconstructs the index from the data file

7.4.1 Operations Required to Maintain an Index File (4/6)

Ⅳ. Record Addition(to the data file)

- Adding of a data record to the data file
  - use RecordFile<Recording>::Write
- Adding of an index record to the index file
  - use TextIndex.Insert
- Index file is kept in sorted order by key
  - insertion of the new index record requires rearrangement of the index *wholly in memory*
  - all of the index rearrangement can be done without any file access
- Textind.cpp in Appendix G
  - implementation of TextIndex::Insert

7.4.1 Operations Required to Maintain an Index File (5/6)

Ⅴ. Record Deletion

- Deleting data records in variable-length record files
  - does not move records around to maintain an ordering on the file
  - Implementation of data record deletion : as an exercise
  - index itself pins all the records
- Deleting the index entry and shifting the other entries to close up the space, or marking the index entry as deleted
  - TextIndex::Remove
- Textind.cpp in Appendix G
  - implementation of TextIndex:: Remove

7.4.1 Operations Required to Maintain an Index File (6/6)

VI. Record Updating

- Update changes the value of the key field
  - require a reordering of the index file as well as the data file
  - as a deletion followed by an addition
- Update does not affect the key field
  - do not require rearrangement of the index file, but may involve reordering of the data file
    - if the record size is unchanged or decreased, the record can be written directly into its old space
    - if the record size is increased, a new slot for the updated record is needed
  - delete/insert approach to maintaining the index can be used

7.4.2 Class TextIndexedFile (1/3)

- Class TextIndexedFile
  - "Indfile.h" in Appendix G
  - support files of data objects with primary keys that are strings
  - methods: Create, Open, Close, Read(sequential and indexed), Append, and Delete(Update)
  - Index : index in memory
  - IndexFile : index file in disk
  - DataFile : data file

7.4.2 Class TextIndexedFile (2/3)

```
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);       //Write (const RecType & record);
    int Update (char * oldkey, const RecType & record); // left as an exercise
    int Delete (char * key);                          //
    int Create (char * name, int mode=ios::in|ios::out);
    int Open (char * name, int mode=ios::in|ios::out);
    int Close ();
    TextIndexFile (IOBuffer &buffer, int keySize, int maxKeys = 100);
    ~TextIndexFile ();   // close and delete
protected:
    TextIndex Index;                       // index in memory
    BufferFile indexFile;                  // index file
    TextIndexBuffer IndexBuffer;
    RecordFile< RecType> DataFile;  // data file
    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();    //extract the key value from the record
```

7.4.2 Class TextIndexedFile (3/3)

```
template <class RecType>
int TextIndexedFile<RecType>::Read (RecType & record)
{
        return result = DataFile . Read (record, -1);
}

template <class RecType>
int TextIndexedFile<RecType>::Read (char * key, RecType & record)
{
        int ref = Index.Search(key);
        if (ref < 0) return -1;
        int result = DataFile . Read (record, ref);
        return result;
}
```

7.4.2 Class TextIndexedFile (3/3)

```
template <class RecType>
int TextIndexedFile< RecType>::Append (const RecType & record)
{
   char * key = record.key();    // extract key value from the record
   int ref = Index.Search(key); // to determine if the key is already in the file
   if (ref != -1)   // key already in file
        return -1;
   ref = DataFile.Append(record);      // if not, the record is appended
   int result = Index.Insert (key, ref);  // (key, ref) is inserted into the index
   return ref;
}

template <class RecType>
int TextIndexedFile<RecType>::Update
      (char * oldKey, const RecType & record)  // Update is left as an exercise.
    //            It requires BufferFile::Update, and BufferFile::Delete
{     return -1;  }
```

7.4.3 Enhancements to Class TextIndexedFile (1/8)

I. Other Types of Keys

- TextIndexedFile
  - support a variety of data object classes, but restrict the key type to string (char *)

  => change a class to a template class

  (1) add a template parameter

  – replace char * by keyType

  (2) replace a class name with the parameter name

  – produce a template class SimpleIndex with a parameter for the key type
  - Simpind.h and Simpind.tc in Appendix G
    - include template class SimpleIndex

7.4.3 Enhancements to Class TextIndexedFile (2/8)

```
template <class keyType>
class SimpleIndex    // compare class TextIndex
{public:
      SimpleIndex (int maxKeys = 100, int unique = 1);
      int Insert(const keyType key, int recAddr);
      int Remove(const keyType key, const int recAddr = -1);
      int Search(const keyType key, const int recAddr = -1, const int exact = 1) const;
      void Print (ostream &) const;
      int numKeys () const {return NumKeys;}
 protected:
      int MaxKeys;
      int NumKeys;
      KeyType * Keys;
      int * RecAddrs;
      int Find(const keyType key, const int recAddr = -1);
      int Init(const int maxKrys, const int unique);
      . . .
 };
```

7.4.3 Enhancements to Class TextIndexedFile (3/8)

- Template index class
  - Dependencies on char * must be removed
  - For string keys, a class String is needed
  - Strclass.h and Strclass.cpp in Appendix G

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

  - String objects and C strings become interchangeable
    - construct a temporary String object using the char * constructor and then doing String assignment
    - use the conversion operator from the class String to convert the String object to a simple C string

SimpleIndex (1/5)

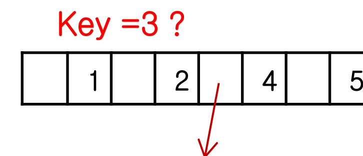
```
template <class keyType>
SimpleIndex<keyType>:: SimpleIndex (int maxKeys, int unique)
        : NumKeys (0), Keys(0), RecAddrs(0)
{
        Init (maxKeys, unique);
}

template <class keyType>
SimpleIndex<keyType>::~SimpleIndex ()
{
        delete Keys;
        delete RecAddrs;
}

template <class keyType>
void SimpleIndex<keyType>::Clear ()
{
        NumKeys = 0;
}
```

```
template <class keyType>
int SimpleIndex<keyType>::Insert (const keyType key, int recAddr)
{
        int i;
        int index = Find (key);
        if (Unique && index >= 0) return 0; // key already in
        if (NumKeys == MaxKeys) return 0; // no room for another key
        for (i = NumKeys-1; i >= 0; i--)
        {
                if (key > Keys[i]) break; // insert into location i+1
                Keys[i+1] = Keys[i];
                RecAddrs[i+1] = RecAddrs[i];
        }
        Keys[i+1] = key;
        RecAddrs[i+1] = recAddr;
        NumKeys ++;
        return 1;
}
```

```
template <class keyType>
int SimpleIndex<keyType>::Remove (const keyType key, const int recAddr)
{
        int index = Find (key, recAddr);
        if (index < 0) return 0; // key not in index
        for (int i = index; i < NumKeys; i++)
        {
                Keys[i] = Keys[i+1];
                RecAddrs[i] = RecAddrs[i+1];
        }
        NumKeys --;
        return 1;
}
template <class keyType>
int SimpleIndex<keyType>::Search (const keyType key,
                const int recAddr, const int exact) const
{
        int index = Find (key, recAddr, exact);
        if (index < 0) return index;
        return RecAddrs[index];
}
```

SimpleIndex (4/5)

```
template <class keyType>
int SimpleIndex<keyType>::Find (const keyType key, const int recAddr, const int exact) const
{
        for (int i = 0; i < NumKeys; i++)
        {
                if (Keys[i] < key) continue; // not found yet
                if (Keys[i] == key) // exact match
                {
                        if (recAddr < 0) return i;
                        else if (recAddr == RecAddrs[i]) return i;
                        else return -1; //
                }
                // no exact match: Keys[i-1]<key<Keys[i]
                if (!exact) // inexact match with key
                        return i;
                return -1;
        }
        // key > all keys in index
        if (exact == 1) return -1; // no exact match
        else return NumKeys-1; // inexact, matches last key
}
```



SimpleIndex (5/5)

```
template <class keyType>
int SimpleIndex<keyType>::Init (const int maxKeys, const int unique)
{
        Unique = unique != 0;
        if (maxKeys <= 0)
        {
                MaxKeys = 0;
                return 0;
        }
        MaxKeys = maxKeys;
        Keys = new keyType[maxKeys];
        RecAddrs = new int [maxKeys];
        return 1;
}
#endif
```

7.4.3 Enhancements to Class TextIndexedFile (4/8)

Ⅱ. Data Object Class Hierarchies

- TextIndexedFile
  - 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
  - if BaseClass supports Pack, Unpack, and Key (virtual functions), the class TextIndexedFile<BaseClass> will correctly output objects derived from BaseClass

7.4.3 Enhancements to Class TextIndexedFile (5/8)

- What about Read ?
  - in a virtual function call, the type of the calling object determines which method to call

```
BaseClass * obj = new subclass1;
obj->Pack(Buffer); //virtual function calls
obj->Unpack(Buffer);
```

  - the type of the object referenced by obj (*obj) determines which Pack and Unpack are called

(1) In the case of Pack

  - information from *obj, of type subclass1, is transferred to Buffer

7.4.3 Enhancements to Class TextIndexedFile (6/8)

(2) In the case of Unpack

- information from Buffer is transferred to *obj
- if Buffer has been filled from an object of class subclass2 or BaseClass, the unpacking cannot be done correctly
  - the source of information (contents of buffer) determines the type of the object in the Unpack, not the memory object

=> Read must be able to determine reliably the type of the target object

  - no support in C++ for guaranteeing accurate type identification of memory objects
  - add record headers in much same fashion as file headers

7.4.3 Enhancements to Class TextIndexedFile (7/8)

III. Multirecord Index Files

- TextIndexedFile
  - the entire index fits in a single record

    TextIndex Index;

    BufferFile IndexFile;

    . . .
  - the maximum # of records in the file is fixed when the file is created

=> Modify class TextIndexedFile to allow the index to be an array of TextIndex objects

=> Add protected methods Insert, Delete, and Search to manipulate the arrays of index object

7.4.3 Enhancements to Class TextIndexedFile (8/8)

IV. Optimization of Operations

- Use binary search in the Find method, which is used by Search, Insert, and Remove
- Avoid writing the index record back to the index file when it has not been changed
  - add a flag to the index object (in memory ?) to signal when it has been changed
    - Set to *false* when the record is initially loaded into memory and set to *true* whenever the index record is modified by the Insert and Remove methods
    - Close method can check this flag and write the record only when necessary
  - useful for manipulating multirecord index files

7.5 Indexes That Are Too Large to Hold in Memory (1/2)

- Disadvantages of simple indexes, which is *too large to hold in memory*
  - binary searching of the index requires several seeks
    - binary searching of an index on secondary storage is not fast
  - index rearrangement due to record addition or deletion requires shifting or sorting records on secondary storage

  ⇒ these problems are no worse than those associated with any file that is sorted by key

- Alternatives
  - ① hashed organization
  - ② tree-structured index: B-tree

7.5 Indexes That Are Too Large to Hold in Memory (2/2)

- Advantages over the use of a data file sorted by key, even if the index cannot be held in memory
  - can use a binary search to access to a record in a variable-length record file
  - sorting and maintaining the index can be less expensive than the data file
  - can rearrange the keys without moving the data records, when pinned records are involved in the data files
  - can use multiple indexes to provide multiple views of a data file

7.6 Indexing to Provide Access by Multiple Keys (1/8)

- Secondary key fields
  - relate a secondary key to a primary key
  - after consulting the secondary key index, consult the primary key index

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.

7.6 Indexing to Provide Access by Multiple Keys (2/8)

- Class definition for secondary key index and a read function

```
class SecondaryIndex  // the record reference is a string
{public:
    int Insert(char * secondaryKey, char *primaryKey);
    char * Search(char * secondaryKey);  / return primary key
    int Remove(char * secondaryKey);
   ...
};

template <class RecType>
int SearchOnSecondary (char * composer, SecondaryIndex  index,
    TextIndexedFile<RecType> dataFile, RecType &rec)
{
    char * key = index.Search (composer);
    //use primary key index to read file
    return dataFile.Read(key, rec);
}
```

7.6 Indexing to Provide Access by Multiple Keys (3/8)

- Two binding methods

  (1) Direct addressing

  - binding of a secondary key to a specific address
  - secondary key references(Beethoven) is directly related to a byte offset(211)

  (2) Indirect addressing

  - binding of a secondary key to a specific primary key
  - secondary key references(Beethoven) is related to a primary key(DG18807)

7.6 Indexing to Provide Access by Multiple Keys (4/8)

I. Record Addition

- Adding an entry to primary index
- Adding an entry to secondary index
  - must shift records or need to rearrange a vector of pointers to structures
  - similar cost to adding an entry to the primary index
- Secondary key with duplicate keys
  - duplicate keys should be ordered according to the values of the reference fields
  - see Fig 7.10

7.6 Indexing to Provide Access by Multiple Keys (5/8)

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.

7.6 Indexing to Provide Access by Multiple Keys (6/8)

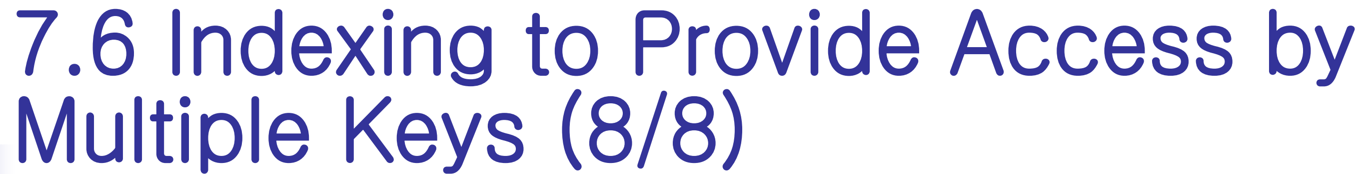
Ⅱ. Record Deletion

- Direct addressing
  - removing not only the corresponding entry in the primary index but also all of the entries in the secondary indexes
  - rearranging the remaining index records => maintained in sorted order by key
- Indirect addressing
  - modify and rearrange *only the primary key index*
  - updated primary key index provides the check of a record-not-found condition

7.6 Indexing to Provide Access by Multiple Keys (7/8)

Ⅲ. Record Updating

- Direct addressing
  - changing a record's physical location in the file also require updating the secondary indexes
- Indirect addressing
  - primary key index insulates the secondary indexes from changes in the data file
  - affect the secondary index only when they change either the primary or the secondary key



7.6 Indexing to Provide Access by Multiple Keys (8/8)

- Three possible situations

(1) Update changes the secondary key

- rearrange the secondary key index

(2) Update changes the primary key

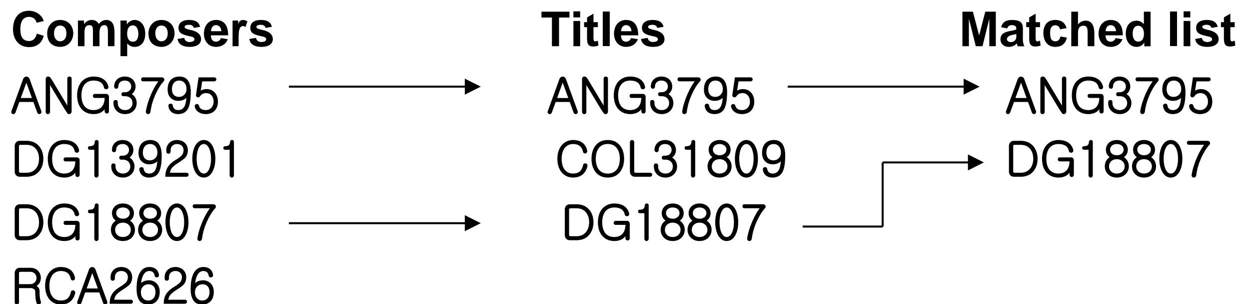
- update only the affected fields in all the secondary key indexes
- may require reordering the secondary index(???)

(3) Updates confined to other fields

- do not affect the secondary key index

7.7 Retrieval using Combinations of Secondary Keys

- *Find all data records with:*
  composer = 'BEETHOVEN' ∧ title = 'SYMPHONY NO. 9'
  - search the composer index for the list of Label IDs that identify records with BEETHOVEN
  - search the title index for the Label IDs associated with records that have SYMPHONY No. 9
  - perform a match(i.e., intersection) operation
  - proceed to the primary key index to look up the addresses of the data file records



7.8 Improving the Secondary Index Structure: Inverted Lists

- Two difficulties of the secondary index structures
  - must rearrange the index file *every time* a new record is added to the file
  - secondary key field is repeated for each entry, if duplicated

=> waste space => make the files larger

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

7.8.1 A First Attempt at a Solution (1/2)

- Array of references with each secondary key
  - need to rearrange the secondary index file every time a new record is added to the data file

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.

7.8.1 A First Attempt at a Solution (2/2)

- Array of references with each secondary key(Cont'd)
  - Two problems
    - need a mechanism for keeping track of the extra Label IDs for more than four Label IDs
    - have to do with space usage

    => lose more space to internal fragmentation

7.8.2 A Better Solution: Linking the List of References (1/5)

- Inverted list
  - files in which a secondary key leads to a set of one or more primary keys => deal with a list of primary key references

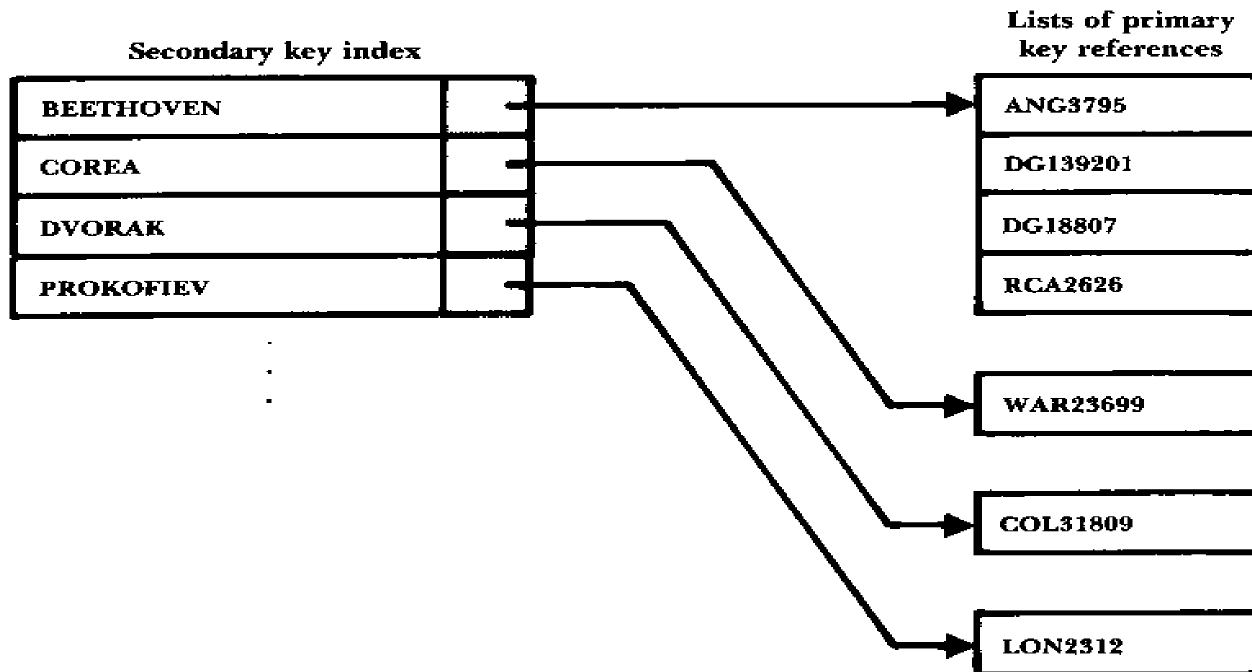


Figure 7.12 Conceptual view of the primary key reference fields as a series of lists.

7.8.2 A Better Solution: Linking the List of References (2/5)

- Improved revision of the inverted list
  - secondary index consisting of records with two fields

  (1) a secondary key field

  (2) a field containing the relative record number(RRN) of the first corresponding primary key reference in the inverted list

    - store the actual primary key references in a separate, entry-sequenced file (Label ID List file)
    - use relative record number (RRN) in the Label ID List file

7.8.2 A Better Solution: Linking the List of References (3/5)

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.

7.8.2 A Better Solution: Linking the List of References (4/5)

- Advantages of inverted lists
  - new composer's name is added or an existing composer's name is changed => rearrange the Secondary Index file
  - deleting or adding recordings for composer
    => changing only the Label ID List file
  - task of rearrangement of the Secondary Index file is quicker
    <= fewer records and smaller records
  - less need for sorting => leaving more room in memory
  - Label ID List file is entry sequenced
    => never needs to be sorted
  - Label ID List file is a fixed-length record file
    => easy to implement a mechanism for reusing the space from deleted records

7.8.2 A Better Solution: Linking the List of References (5/5)

- Disadvantage of inverted lists
  - Label IDs associated with a given composer are no longer guaranteed to be physically grouped together(no clustering)
    - can involve a large amount of seeking
  - keep the Label ID List file in memory
    - expensive and impractical
    - 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
- Solution
  - the notion of dividing the index into pages
  - B-trees and other methods for handling large indexes on secondary storage

7.9 Selective Indexes

- Selective index
  - an index that divides a file into parts and provides a selective view
  - contains only the titles of classical recordings in the record collection
  - e.g.: "recordings released prior to 1970"
  - could be combined into Boolean or operations
  - useful when the contents of a file fall naturally and logically into several broad categories

7.10 Binding (1/2)

- Binding
  - at what point is the key bound to the physical address of its associated record?

(1) binding of primary keys to an address

- at the time the files are constructed
  - result in faster access
  - use the byte offset of data record you are seeking

(2) binding of secondary keys to an address

- at the time that they are used

- Disadvantages of binding directly in the file (binding tightly)
  - reorganizations of the data file must result in modifications to all bound index files
  - desirable for file organization on a mass-produced, read-only optical disk
- Advantages of postponing binding until actually retrieved
  - safer
  - allow the primary key index to act as a kind of final check of whether a record is really in the file
  - desirable for file applications in which record addition, deletion, and updating do occur