BINARY SEARCHING, KEYSORTING AND INDEXING

Contents of today's lecture:

- Binary Searching (Chapter 6.3.1 - 6.3.3),
- Keysorting (Chapter 6.4)
- Introduction to Indexing (Chapter 7.1-7.3)

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 6.3.1-6.3.3,6.4,7.1-7.3

Binary Searching

Let us consider fixed-length records that must be searched by a **key value**.

If we knew the RRN of the record identified by this key value, we could jump directly to the record (using "seek").

In practice, we do not have this information and we must search for the record containing this key value.

If the file is not sorted by the key value we may have to look at every possible record before we find the desired record.

An alternative to this is to maintain the file **sorted by key value** and use **binary searching**.

A binary search[1] algorithm in C++ :

```
class FixedRecordFile{
   public:
     int NumRecs();
     int ReadByRRN(RecType & record, int RRN);
};
class KeyType {
   public:
     int operator==(KeyType &);
     int operator<(KeyType &);
};
class RecType {
   public:
     KeyType key();
};
int BinarySearch(FixedRecordFile & file, RecType & obj,
                 KeyType & key) {
   int low=0; int high=file.NumRecs() -1;
   while (low <= high) {
     int guess = (high + low)/2;
     file.ReadByRRN(obj,guess);
     if (obj.key() == key) return 1;
     if (obj.key() > key)  high = guess - 1;
     else low = guess + 1;
   }
   return 0;  //did not find key
}
```

[1] this algorithm corrects some mistakes found in the textbook.

Binary Search versus Sequential Search :

Binary Search : $O(\log_2 n)$
Sequential Search : $O(n)$

If file size is doubled, sequential search time is doubled, while binary search time increases by 1.

Keysorting

Suppose a file needs to be sorted, but it is too big to fit into main memory.

To sort the file, we only need the **keys**. Suppose that all the keys fit into main memory.

Idea:

- Bring the keys to main memory plus corresponding RRN
- Do internal sorting of keys
- Rewrite the file in sorted order

keynodes array

key	RRN
HARRISON	0
KELLOG	1
HARRIS	2
BELL	3

Main Memory

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

Disk

keynodes array

key	RRN
BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

Internal sorting
in main memory

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

No change in Disk

keynodes array

BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

records

BELL | 8912 Hill...
HARRIS | 4343 West...
HARRISON | 387 Eastern...
KELLOG | 17 Maple...

create new sorted file to
replace previous

How much effort we must do (in terms of disk accesses) ?

- Read file sequentially once
- Go through each record in random order (seek)
- Write each record once (sequentially)

Why bother to write the file back?

Use keynode array to create an **index file** instead.

index file

BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

Leave file unchanged

This is called INDEXING !!

Pinned Records

Remember that in order to support deletions we used `AVAIL LIST`, a list of available records.
The `AVAIL LIST` contains info on the physical information of records. In such a file a record is said to be **pinned**.

If we use an **index file** for sorting, the `AVAIL LIST` and positions of records remain unchanged. This is convenient.

Introduction to Indexing

- Simple indexes use simple arrays.
- An index lets us **impose order on a file** without rearranging the file.
- Indexes provide **multiple access paths** to a file - **multiple indexes** (library catalog providing search for author, book and title).
- An index can provide keyed access to variable-length record files.

A Simple Index for Entry-Sequenced File

Records (Variable Length)

Address of Record	
17	LON | 2312 | Symphony N.S | ...
62	RCA | 2626 | Quartet in C sharp | ...
117	WAR | 23699 | Adagio | ...
152	ANG | 3795 | Violin Concerto | ...

Primary key = company label + record ID (LABEL ID).

Index :

key	Reference field
ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

- Index is sorted (main memory).
- Records appear in file in the order they entered.

How to search for a recording with given LABEL ID ?
"Retrieve recording" operation :

- Binary search (in main memory) in the index : find LABEL ID, which leads us to the reference field.
- Seek for record in position given by the reference field.

Two issues to be addressed :

- How to make a persistent index (i.e. how to store the index into a file when it is not in main memory).
- How to guarantee that the index is an accurate reflection of the contents of the file. (This is tricky when there are lots of additions, deletions and updates.)

INDEXING

Contents of today's lecture:

- Indexing

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 7.4 - 7.6, 7.7 - 7.10

Indexing

Operations in order to Maintain an Indexed File

1. Create the original empty index and data files.
2. Load the index file into memory before using it.
3. Rewrite the index file from memory after using it.
4. Add data records to the data file.
5. Delete records from the data file.
6. Update records in the data file.
7. Update the index to reflect changes in the data file.

We will take a closer look at operations 3-7.

Rewrite the Index File from Memory

When the data file is closed, the index in memory needs to be written to the index file.

An important issue to consider is what happens if the rewriting does not take place (power failures, turning the machine off, etc.)

Two important safeguards:

- Keep an status flag stored in the header of the index file. The status flag is “on” whenever the index file is not up-to-date. When changes are performed in the index in main memory the status flag in the file is turned on. Whenever the file is rewritten from main memory the status flag is turned off.
- If the program detects the index is out-of-date it calls a procedure that reconstruct the index from the data file.

Record Addition

This consists of appending the data file and inserting a new record in the index. The rearrangement of the index consists of "sliding down" the records with keys larger than the inserted key and then placing the new record in the opened space.

Note: This rearrangement is done in main memory.

Record Deletion

This should use the techniques for reclaiming space in files (Chapter 6.2) when deleting from the data file. We must delete the corresponding entry from the index:

- Shift all records with keys larger than the key of the deleted record to the previous position (in main memory); or
- Mark the index entry as deleted.

Record Updating

There are two cases to consider:

- The update changes the value of the key field:
  Treat this as a deletion followed by an insertion
- The update does not affect the key field:
  If record size is unchanged, just modify the data record. If record size changes treat this as a delete/insert sequence.

Indexes too Large to Fit into Main Memory

The indexes that we have considered before could fit into main memory. If this is not the case, we have the following problems:

- Binary searching of the index file is done on disk, involving several "seeks".
- Index rearrangement (record addition or deletion) requires shifting on disk.

Two main alternatives:

- Hashed organization (Chapter 11) (When speed is a top priority)
- Tree-structured (multilevel) index such as B-trees and B+ trees (Chapter 9,10) (It allows keyed and ordered sequential access).

But a simple index is still useful, even in secondary storage:

- It allows binary search to obtain a keyed access to a record in a variable-length record file.
- Sorting and maintaining an index is less costly than sorting and maintaining the data file, since the index is smaller.
- We can rearrange keys, without moving the data records when there are pinned records.

Indexing to Provide Access by Multiple Keys

In our recording file example, we built an index for LABEL ID key.
This is the **primary key**.
There may be **secondary keys**: title, composer and artist.

We can build **secondary key indexes**.

Composer index:

Secondary key	**Primary key**
Beethoven	ANG3795
Beethoven	DG139201
Beethoven	DG18807
Beethoven	RCA2626
Corea	WAR23699
Dvorak	COL31809
Prokofiev	LON2312

Note that in the above index the secondary key reference is to the primary key rather than to the byte offset.

This means that the primary key index must be searched to find the byte offset, but there are many advantages in postponing the binding of a secondary key to an specific address.

Record Addition

When adding a record, an entry must also be added to the secondary key index.
Store the field in **Canonical Form** (say capital letters, with pre-specified maximum length).
There may be duplicates in secondary keys. Keep duplicates in sorted order of primary key.

Record Deletion

Deleting a record implies removing all the references to the record in the primary index and in all the secondary indexes. This is too much rearrangement, specially if indexes cannot fit into main memory.
Alternative:

- Delete the record from the data file and the primary index file reference to it. Do not modify the secondary index files.
- When accessing the file through a secondary key, the primary index file will be checked and a deleted record can be identified.

This results in a lot of saving when there are many secondary keys.

The deleted record still occupy space in the secondary key indexes. If a lot of deletions occur, we can periodically cleanup these deleted records from the secondary key indexes.

Record Updating

There are three types of updates :

- Update changes the secondary key :

  We have to rearrange the secondary key index to stay in sorted order.

- Update changes the primary key :

  Update and reorder the primary key index; update the references to primary key index in the secondary key indexes (it may involve some re-ordering of secondary indexes if secondary key occurs repeated in the file).

- Update confined to other fields :

  This won't affect secondary key indexes. The primary key index may be affected if the location of record changes in data file.

Retrieving Rec's using Combinations of Secondary Keys

Secondary key indexes are useful in allowing the following kinds of queries :

- Find all recording with composer "BEETHOVEN".
- Find all recording with title "Violin Concerto".
- Find all recording with composer "BEETHOVEN" **and** title "Symphony No.9".

This is done as follows :

Matches from composer index	**Matches from title index**	**Matched list (logical "and")**
ANG3795 DG139201 DG18807 RCA2626	ANG3795 COL31809 DG18807	ANG3795 DG18807

Use the matched list and primary key index to retrieve the two recordings from the file.

Improving the Secondary Index Structure: Inverted Lists

Two difficulties found in the proposed secondary index structures :

- We have to rearrange the secondary index file even if the new record to be added in for an existing secondary key.
- If there are duplicates of secondary keys then the key field is repeated for each entry, wasting space.

Solution 1

Make the secondary key index record consist of secondary key + array of references to records with secondary key.

Problems :

- The array will take a maximum length and we may have more records.
- We may have lots of unused spaces in some of the arrays (wasting space in internal fragmentation).

Solution 2 : Inverted Lists

Organize the secondary key index as an index containing one entry for each key and a pointer to a **linked list** of references.

Secondary Key Index File

0	Beethoven	3
1	Corea	2
2	Dvorak	5
3	Prokofiev	7

LABEL ID List File

0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	6
4	DG18807	1
5	COL31809	-1
6	DG139201	4
7	ANG36193	0

Beethoven is a secondary key that appears in records identified by the LABEL IDs: ANG3795, DG139201, DG18807 and RCA2626 (check this by following the links in the linked list).

Advantages:

- Rearrangement of the secondary key index file is only done when a new composer's name is added or an existing composer's name is changed. Deleting or adding recordings for a composer only affects the **LABEL ID list file**. Deleting all recordings by a composer can be done by placing a "-1" in the reference field in the secondary index file.
- Rearrangement of the secondary index file is quicker since it is smaller.
- Smaller need for rearrangement causes a smaller penalty associated with keeping the secondary index file in disk.
- The **LABEL ID list file** never needs to be sorted since it is entry sequenced.
- We can easily reuse space from deleted records from the **LABEL ID list file** since its records have fixed-length.

Disadvantages :

- Lost of "locality" : labels of recordings with same secondary key are not contiguous in the **LABEL ID list file** (seeking). To improve this, keep the **LABEL ID list file** in main memory, or, if too big, use paging mechanisms.

Selective Indexes

We can build selective indexes, such as :
Recordings released prior to 1970, recordings since 1970.

This may be useful in queries involving boolean "and" operations :
"Retrieve all the recordings by Beethoven released since 1970".

Binding

In our example of indexes, when does the binding of the index to the physical location of the record happens ?

For the primary index, binding is at the time the file is constructed. For the secondary index, it is at the time the secondary index is used.

Advantages of postponing binding (as in our example):

- We need small amount of reorganization when records are added/deleted.
- It is a safer approach : important changes are done in one place rather than in many places.

Disadvantages :

- It results in slower access times (binary search in secondary index plus binary search in primary index).

When to use a **tight binding** ?

- When data file is nearly static (little or no adding, deleting or updating of records).
- When rapid retrieval performance is essential. Example : Data stored in CD-ROM should use tight binding.

When to use the **bind-at-retrieval** system?

- When record additions, deletions and updates occur more often.