Indexing and Cosequential Processing

Last Time : Indexing PartII (Chapters 7.4 - 7.6)

Today : Indexing Part III (Chapters 7.7 - 7.10) and Introduction to Cosequential Processing (Chapter 8.1)

Indexing

Retrieving 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 rearrangements 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.

Cosequential Processing

Cosequential processing involves the **coordinated processing** of **two or more sequential lists** to produce a single output list.

The two main types of resulting output lists are :

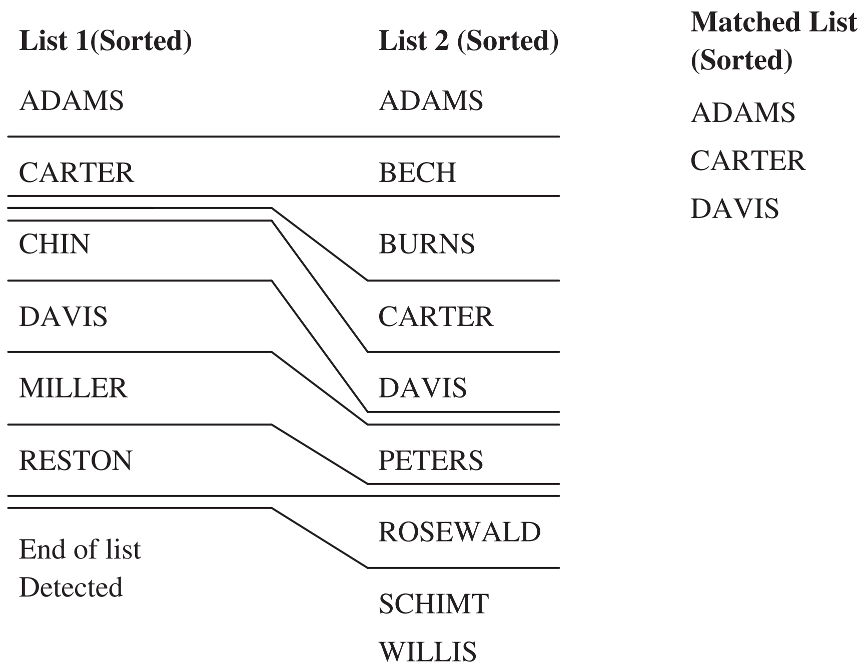
- Matching (intersection) of the items of the lists.
- Merging (union) of the items of the lists.

Examples of applications :

1. Matching :
   Master file - bank account info (account number, person name, account balance) - sorted by account number
   Transaction file - updates on accounts (account number, credit/debit info).

2. Merging :
   Merging two class lists keeping alphabetic order.
   Sorting large files (break into small pieces, sort each piece and then merge them).

Matching the Names in Two Lists



Synchronization :

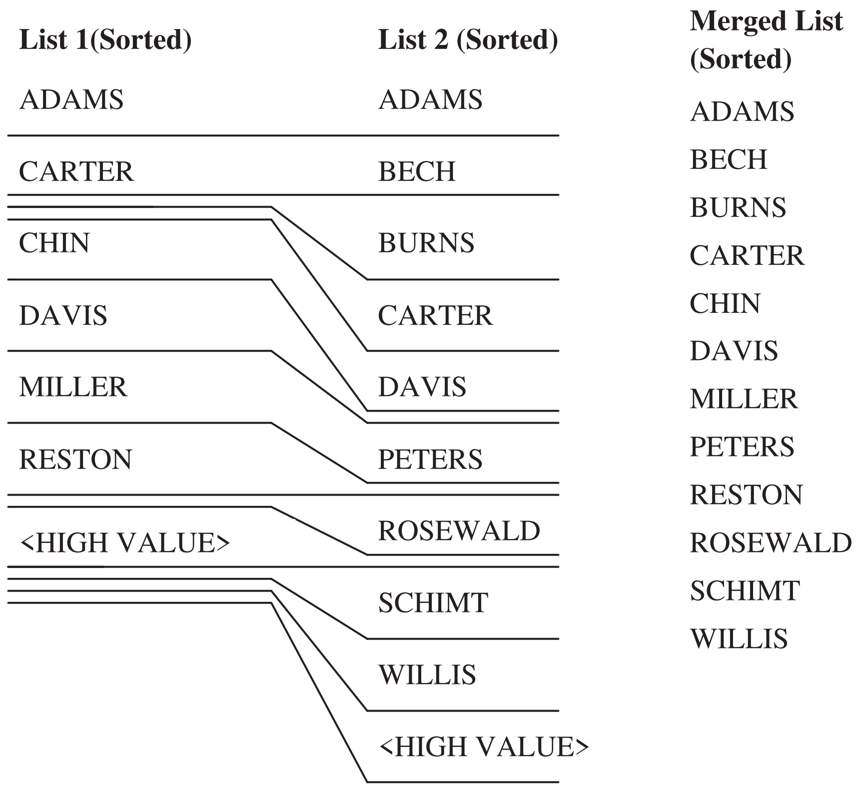
`item(1)` = current item from list 1
`item(2)` = current item from list 2

```
if item(1) < item(2) then
     get next item from list 1
if item(1) > item(2) then
     get next item from list 2
if item(1) = item(2) then
     output the item to output list
     get next item from list 1 and list 2
```

Handling End-of-File/End-of-List Condition

When we get to the end of **either** list 1 **or** list 2, we halt the program.

Merging the Names in Two Lists, Eliminating Repetitions



Modify the synchronization slightly :

```
if item(1) < item(2) then
     output item(1) to output list
     get next item from list 1
if item(1) > item(2) then
     output item(2) to output list
     get next item from list 2
if item(1) = item(2) then
     output the item to output list
     get next item from list 1 and list 2
```

Handling End-of-File/End-of-List Condition

1. Using a `<HIGH VALE>` as in the previous example:

   By storing `<HIGH VALUE>` in the current item for the list that finished, we make sure the contents of the other list is flushed to the output list.

   The **stopping criteria** is changed to :

   When we get to the end of **both** list 1 **and** list 2, we halt the program.

2. Reducing the number of comparisons:

   We can perform a similar algorithm with less comparisons **without** using a `<HIGH VALUE>` as described above.

   The **stopping criteria** becomes:
   When we get to the end of **either** list 1 **or** list 2, we halt the program.

   Finalization: flush the unfinished list to the output list.

```
while (list 1 did not finish)
   output item(1) to output list
   get next item from list 1

while (list 2 did not finish)
   output item(2) to output list
   get next item from list 2
```