University of Dublin
Trinity College

Index Structures for Files

Static Indexes

Why do we index in the physical world?

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 concepts - keys and reference fields

Consider what would happen if we tried to binary search the words in a book

- Sorting the words would have a bad effect on the meaning of the book
- Adding an index allows us to impose an order on a file without actually re-arranging it

Why do we index in the physical world?

We want to find some books in a library. We want to locate books by a specific author, by their title or by their subject area.

One way to organise the books so we can do this is to have 3 separate copies of each book, and three separate libraries.

All the books in one library would be arranged (sorted or hashed) according to author, another would arrange them by subject and a third by title.

Why do we index in the physical world?

A better system is to use a card catalogue

- A set of three indexed, each based on a different key field
- All of the indexes use the same catalogue number as a reference field
- Each index allows us to efficiently search a file based on a different data we are looking for
- An index may be arranged as a sorted list which can be binary searched, a hash table, or a tree structure of the type we'll look at in the coming lectures

Indexing files in IT?

Indexes are auxiliary access structures

- Speed up retrieval of records in response to certain search conditions
- Any field can be used to create an index and multiple indexes on different fields can be created

The index is separate from the main file and can be created and destroyed without affecting the main file.

- The index must be updated when records are inserted or deleted to/from the main file.

The issue is how to organise the index records for efficient access and ease of maintenance.

Advantages over Hashing

Multiple indexes can be built for the same file, allowing for efficient access over multiple fields

Static/Dynamic Indexes

Static Index structures

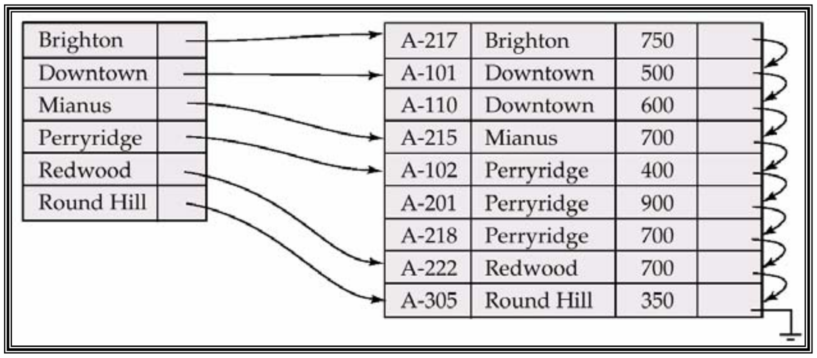
- Static indexes are of fixed size and structure, though their contents may change.
- As we will see requires periodic reorganisation
- IBM's ISAM (Indexed Sequential Access Method) uses static index structures
- Covered in these lectures

Dynamic Index structures

- Dynamic indexes change shape gradually in order to preserve efficiency.
- Implemented as search trees (e.g. B-Trees, AVL Trees etc.)
- Covered later in course

Dense Index

Index record appears for **every** search key value



Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sparse Index

Sparse Index: contains index records for only some search-key values.

- Applicable when records are sequentially ordered on search-key

To locate a record with search-key value K we:

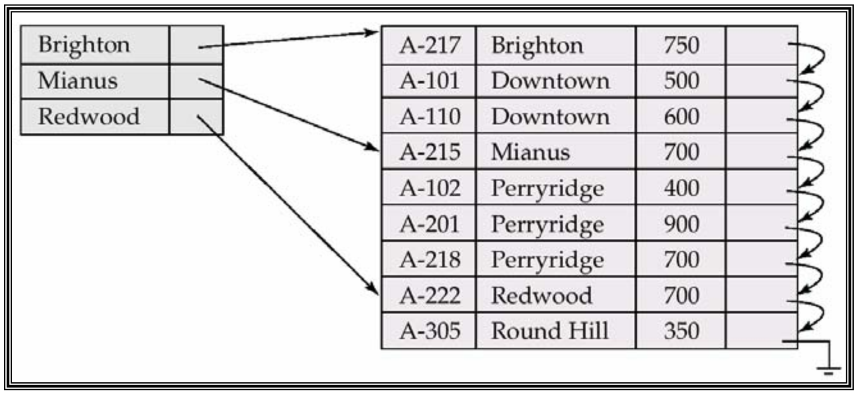
- Find index record with largest search-key value < K
- Search file sequentially starting at the record to which the index record points

Less space and less maintenance overhead for insertions and deletions.

Generally slower than dense index for locating records.

Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

Sparse Index



Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Single Level Index

A single level index is an auxiliary file that makes it more efficient to search for a record in the data file

The index is usually specified on one field of the file

One form of an index is a file of entries <field value, pointer to record> which is ordered by field value

The index is called an *access path* on the field

The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller

A binary search on the index yields a pointer to the file record

Types of Single Level Indexes

Primary Index

Clustering Index

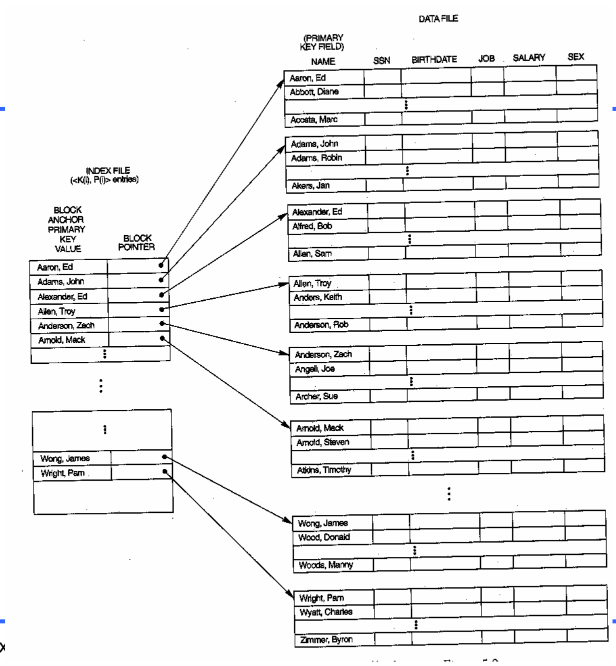
Secondary Index

Primary Index

A primary index is an ordered file whose entries are of fixed length with two fields:

<value of primary key; address of data block>

- The data file is **ordered** on the primary key field and requires primary key for each record to be unique/distinct
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
- A similar scheme can be used for the *last record* in a block



Example Performance Gain

Ordered file of r=30000 records

Block size B =1024 bytes

Records Fixed sized and unspaned with record length R=100 bytes

Bfr = B/R = 1024/100 = 10 records per block

Number of blocks needed for file is

b= r/bfr = 30000/10 = 3000 blocks

A binary search on the data file would need approx

$\log_2 b = \log_2 3000 = 12$ block accesses

Example Peformance Gain

Now suppose ordering key V=9bytes long and block pointer P=6bytes long

Size of each index entry Ri = 9+6 = 15 bytes

Blocking factor bfri = 1024/15 = 68 entries per block

Total number of entries ri= total number of blocks in data file = 3000

Number of blocks needed for index is

bi= ri/bfri = 3000/68 = 45 blocks

A binary search of index file would need

$\log_2 bi = \log_2 45 = 6$ block accesses

PLUS one block access into the data file itself using the pointer

Therefore 7 block accesses needed

Problem with Primary Indexes

Insertion or deletion of record in ordered data file involves not only making space or deleting space in the data file

… but also changing the index entries to reflect the new situation

Possible solution

- Use deletion markers for records
- Maintain linked list of overflow records for each block in the data file
- Reorganise periodically

Clustering Index

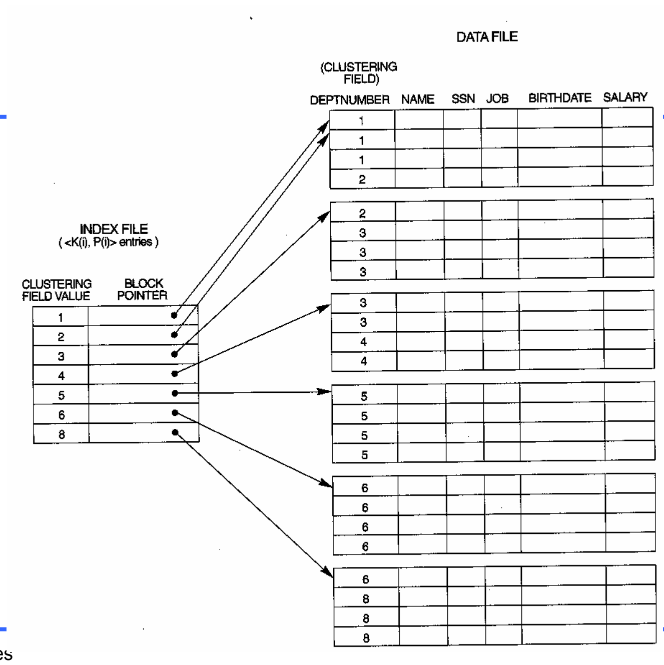
A clustering index is an ordered file whose entries are of fixed length with two fields:

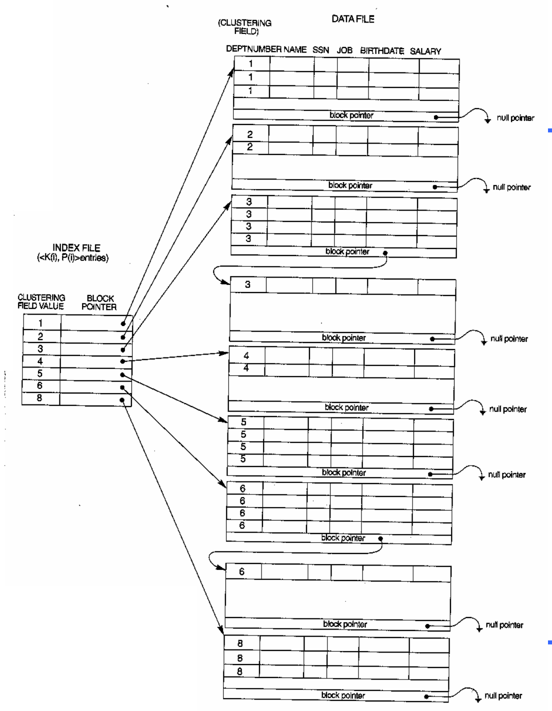
<value of clustering key; address of data block>

- The data file is **ordered** on the clustering field but the clustering key does not have distinct value for each record
- Index includes one index entry *for each distinct value* of the clustering field; the index entry points to the first data block that contains records with that field value

Insertion/Deletion still problematic due to ordering of main data file

- To solve it is common to reserve a block or contiguous blocks (see diagram Separate Block Clustering)


DATA FILE
(CLUSTERING FIELD)
DEPTNUMBER
NAME
SSN
JOB
BIRTHDATE
SALARY
INDEX FILE
(<K(i), P(i)> entries)
CLUSTERING FIELD VALUE
BLOCK POINTER


(CLUSTERING FIELD)
DATA FILE
DEPTNUMBER NAME SSN JOB BIRTHDATE SALARY
block pointer
null pointer
INDEX FILE
(<K(i), P(i)>entries)
CLUSTERING FIELD VALUE
BLOCK POINTER

University of Dublin
Trinity College

Index Structures for Files

Secondary Index

A secondary index is an ordered file whose entries are of fixed length with two fields:

<value of key; address of data block or record pointer>

- The secondary key is some **nonordering** field of the data file

Frequently used to facilitate query processing

For example say we know that queries related to genre are frequent

- SELECT * FROM movie WHERE genre="comedy";

We can ask the DBMS to create a secondary index on *genre* by issuing the following SQL command

- CREATE INDEX Gindex ON Movie(genre);

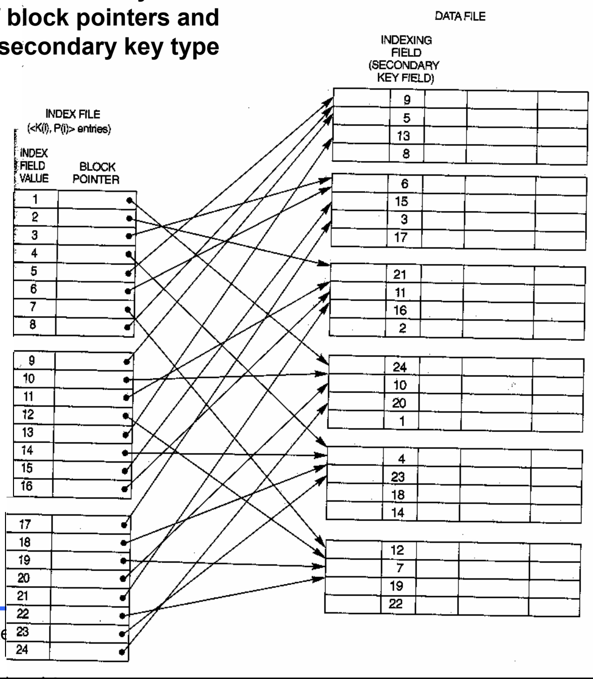
Secondary Index where unordering field is a key

If the unordering field has distinct values (i.e. could be considered a secondary key) then

- One index entry for each record in the data file
- Pointer points to the block in which the record is stored or to the record itself

Index entries are still ordered so can do binary search but will need to know if pointer is a record pointer or a block pointer in order to process search correctly

Secondary Index of block pointers and secondary key type



Secondary Index where unordering field is **not** a key

Option #1

- Include several index entries with the same first value, one for each record. This is a dense index

Option #2

- Have variable length index entries, with a repeating field for the pointer. For example <K(i),P(i,1)… P(i,k)>

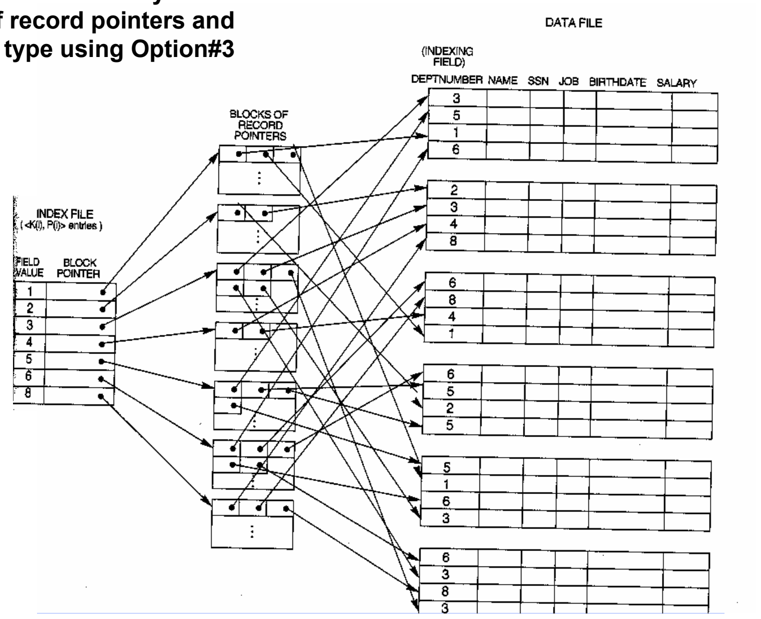
For these two options the binary search algo needs modification

Secondary Index where unordering field is **not** a key

Option #3

- Keep one index entry per value of fixed length with a pointer to a block of pointers, that is add a level of indirection
- If pointers cannot fit in the allocated space for the block of pointers use an overflow or linked list approach to cope

Secondary Index of record pointers and nonkey type using Option#3



Performance

Generally Secondary Indexes leads to more storage space and longer search time (due to larger number of entries) than for primary indexes

However for an arbitrary record than improvement greater as otherwise we would have to do a linear search!

Consider Earlier Example

Recap

- r = 30000 fixed length (100bytes) records of block size B=1024bytes
- File has 3000 blocks as we already calculated

Linear search would require b/2=3000/2=1500 block accesses

Suppose a secondary index on a field of V=9bytes

Thus Ri = 9+6 = 15bytes

bfri = B/Ri = 1024/15 = 68 entries per block

Number of index entries ri = number of records = 30,000

Number of blocks need is bi= ri/bfri = 30000/68 = 442 blocks

A binary search of index file would need

$\log_2 bi = \log_2 442 = 9$ block accesses

PLUS one block access into the data file itself using the pointer

Therefore 10 block accesses as oppose to 1500 block accesses

Another way of organising Secondary Index: Inverted File

The inverted file contains one index entry for each value of the attribute in question. The entry contains a list of pointers to every record with that attribute value.

For example, secondary key of car manufacturer

Audi	11
BMW	3 9 16 17
Ford	1 4 5 7 10 14 18 19 20
Honda	15
VW	2 6 8 12 13

Why called an "inverted file"?

Consider the main file as a function which maps addresses to (attribute, value) pairs :

- file (address) -> (attribute1, value), (attribute2, value), ...

Inverted files are functionally the inverse of the main file

- inv_file (attribute, value) -> address, address, ...

A number of attributes can be inverted. Degree of inversion of a file is the percentage of attributes indexed in this way. 100% inversion is when every attribute is indexed.

Queries on multiple keys need not refer to the main file if all the keys in the query are indexed.

- Consider a query for the record numbers of "green Fords" if both colour and manufacturer are indexed.

Yet another way of organising Secondary Index: Threaded Files

Each record has a pointer field for each indexed secondary key value. This field is used to link (thread) all records with the same attribute value for that key.

The threaded file has a number of separate threads running through it.

The index then contains a pointer to the head of each list.

To find all records with a particular secondary attribute value, find that value in the index, and follow the thread.

Example Retrieval in Threaded Files

To find all green Ford cars (i.e. a query based on two attributes) we have three options:

1. traverse the list of Fords (using the Manufacturer index and the associated thread) checking each to see if it is green.
2. traverse the list of green cars checking each to see if it is a Ford.

We would prefer to traverse a short list - index entries could include a thread length.

3. traverse both threads simultaneously (cf. sequential file merge). Requires that records are threaded in pointer order.

Yet another way of organising Secondary Indexes: Multilists

Like threaded files, but index contains a pointer to every kth record with the particular attribute value.

- Speeds up merge operations
- Can skip over the rest of a sublist if the next pointer in the index is still smaller than the current pointer in the other thread.
- Note that threaded files can be considered as multilists with = $k = \infty$

Cellular Multilist

- Like an inverted file but only list the secondary storage blocks which contain records with the attribute value.
- Searches must access the main file blocks to ascertain exactly which record has the value.
- Compromise between threads and inverted files.

Review
Basic Operations involving Indexes

Retrieve a record based on a key

Create the original empty index and data files

- Both the index file and the data file are created empty

Add records to the data file and index

- Adding a new record to the data file requires that we also add a record to the index file
- Adding to the data file is easy. Just add at the end or at an existing gap between records
- Adding to the index is not easy, because entries of the index file have to be sorted.
- This means that we have to shift all the index records after the one we are inserting
- Essentially, we have the same problem as inserting records into a normal sorted file
- One solution is to use a hash table file for the index, rather than a sorted file
- Another solution is to use sorted structures that are very cheap to add to

Review

Indexes allow access using different keys without duplicating all the records

- avoiding duplication saves storage
- avoiding duplication makes modifying the data easier - we don't have lots of different copies to keep up to date

Indexes allow a lot of flexibility in the layout of the data file

- We don't need fixed length records
- We can store records in any order