Storage Structures and Performance

The **physical layer** of the database consists of a number of files in which the data and metadata are stored

This can be changed without affecting the information content of the database -

- since there are many **different file structures** that can be used to store data expressed in the logical model
  - e.g. there are many ways of representing a table as a file of data
- only the **speed of data access** is affected by the use of different representations
- in particular, **sorted files** are quicker to search that unsorted files

What a DBA Can Do

There are a number of aspects of the physical structure that the DBMS might allow the Database Administrator to vary (NB these are highly privileged tasks)

- the order in which the data is stored - **ordered data** is quicker to search
- **hashing** functions which calculate storage locations from data values
- **indexes** added to the fundamental data structure
- the **storage structure** for a set of records
- the way in which the data is **split between files**
- in particular extra support for **joined data**
- **clustering** data on the same disk page which will be used together
- how to allocate **space** among the disk pages

NB – reducing the amount of **disk i/o** is the main performance goal

Unordered Searching

Linear Search Technique (this makes no assumptions on the order of records)

- Start at first record
- Go on if it's not the one you want
- Fail if you reach the end

To find a record that is there will take on average N/2 accesses where N is the number of records

Failure to find a record will take N accesses

The retrieval time is said to be of **order N** which is written **O(N)**

Example. If Access Time = 1/1000 sec. and N = 1,000,000

- Average Time to find = 500 secs = 8 minutes
- Failure to find > 1/4 an hour

Ordered Searching

Key Slide

Binary Search Technique (Assume the records in the file are ordered in ascending order based on some aspect of the content)

- Start with the middle record
- There are three cases:
  - it's the one that you want - you can quit with success
  - it's bigger than the one you want - you can ignore the first half
  - it's smaller than the one you want - you can ignore the second half
- If it is bigger, repeat with the record that is in the middle of the upper half
- If smaller, start from the middle of the lower half
- Keep doing this until **either** you find your record
  or you can't subdivide any more

Example, Binary Search with N = 8

	1	3	5	7	9	11	13	15
The values are								
Find 11: try middle - this is 9					^			
ignore lower half (1-7)	x	x	x	x	x			
try middle of those left - 13							^	
ignore upper half (15)	x	x	x	x	x		x	x
try middle of those left -						^		
this is 11- quit with success								

_____

	1	3	5	7	9	11	13	15
Find 6: try middle - this is 9					^			
ignore upper half (11-15)					x	x	x	x
try middle of lower half - 5			^					
ignore lower half (1 and 3)	x	x	x		x	x	x	x
try middle of upper half - 7				^				
nothing left - quit with failure	x	x	x	x	x	x	x	x

Time Improvement

To find a record by binary search takes on average $\log_2 (N)/2$

To fail to find the record takes $\log_2 (N)$

The retrieval time is of order, **$\log_2 (N)$** - written **0 (log (N))**

For large N, this is very much smaller than N - it is the inverse of raising 2 to the power of N

Example:

- Ave time to find = $\log_2 \underline{(1{,}000{,}000)}$ / 2 x 1,000
  = 20/2000 = 1/100 second

But only if it is ordered on the attribute we are looking for

- there is no speed up on finding employees by name if they are ordered by staff number

Considerations On File Organisation

1/ Access Time

- The time to find an item

2/ Insertion Time

- The time to place an item and update all the indexes

3/ Deletion Time

- Access time plus the time to update all the indexes

4/ Space Overhead

- Physically ordered files are minimal
- Pointers and indexes cost space

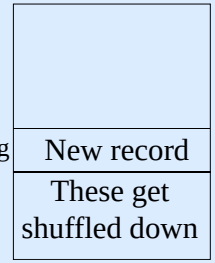
Ordered and Unordered File Structures For Relations

Unordered Files (Heap Files)

- Put each new record at the end of the file
- Fast at : insertion
- Slow at : access - you must use linear search
- Space overhead: leaves gaps after deletion

Sequentially Ordered Files (ISAM Files)

- Each row is added in order of the primary key
- Fast at: retrieval - if by the primary key
- Slow at: insertion, retrieval – each means re-ordering
- Space overhead : none



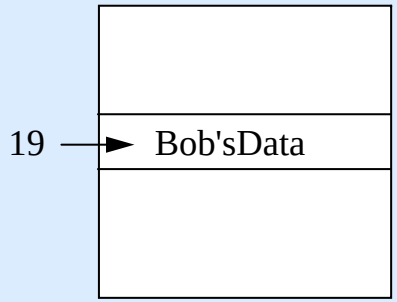
Sequentially Ordered With Overflow File Structures

The file is mostly ordered with added records tacked on to the end

- Insertion is fast - just at end
- Search is quite fast
  - quick search of the bulk of the file
  - slow search of the end
- Regular integration of the unordered section with the ordered part - e.g. at weekends.

Hashed Files I

4. **Hashed Files** (c.f. Cryptography)
   - A **hash function** generates an address from the attribute values.
     - Simplified example: name --> add values for each character (A = 1, B = 2, etc.)
   - Thus "BOB" becomes 2 + 15 + 2 = 19 and 19 is the address of Bob's data.
   - Can give rise to **collisions**! OBB = 19 as well.



Hashed Files II

Hashed structures need careful organisation, with:

- good hash functions which spread the addresses well - the example is a very bad one
- a sensible method of resolving collisions, e.g.
  - having an overflow area
  - using a second hash function
  - using the next available space
- a sensible partitioning of the disc - so that several records share a partition

and

- structures to handle overflow if a partition becomes full

Dynamic hashing re-organises the storage as the database becomes bigger

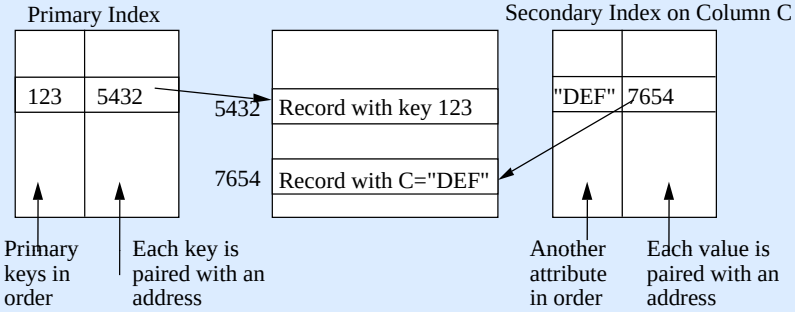
Indexes

Key Slide

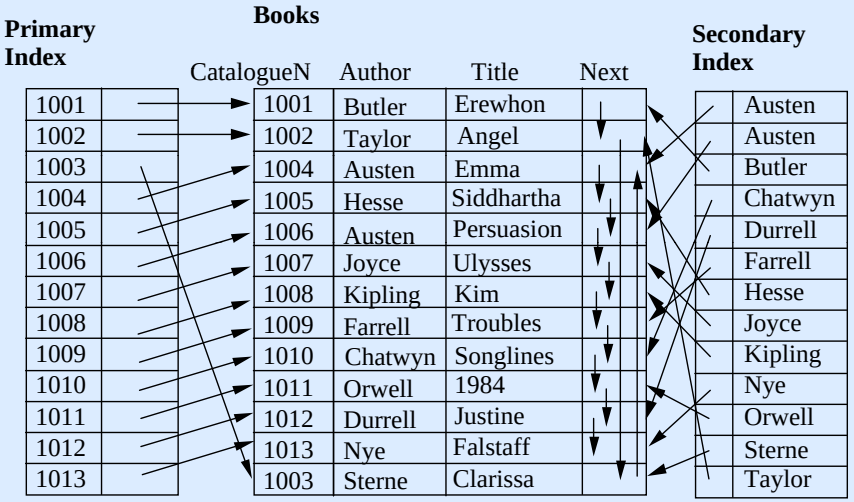
Indexes are additional file structures similar to the indexes of books which do not hold fresh data, but ways of ordering addresses to the data

An index allows the data to be unsorted and pairs a set of data values with addresses for records with those values.

The **primary index** uses the primary key, **secondary indexes** other columns, thus:



Example



Balanced Trees (B-trees)

are one of the main implementation techniques for indexes

They consist of a tree structure in which each node (except the leaf nodes) contains data pointers, decision values and tree pointers to other nodes

- a **decision value** determines which tree pointer to follow to look for the data pointer for that value

The structure of a tree node is:

- $pointer_1$, $decisionValue_1$, $pointer_2$….$decisionValue_n$ $pointer_{n+1}$
- if you are looking for a key value, compare it with the first decision value
- if it is less than the value follow $pointer_1$
- otherwise try the next decision value
- if it is more than the last decision value follow $pointer_{n+1}$

The **depth** of the tree is the length of the path from root to leaf nodes

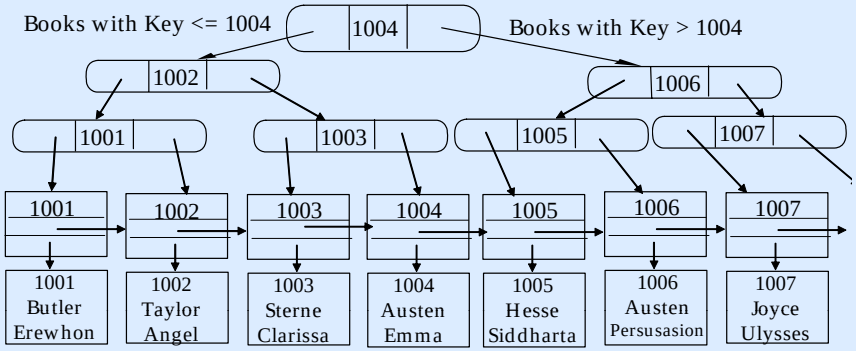
- in a **balanced tree**, this is the same for all leaf nodes

The **order** of the tree is the number of decision values in a node

B+ Trees

There are many versions of the basic B-tree structure

- B+ trees are the usual one used – they only have data pointers in the leaf node
- i.e. non-leaf nodes only have decision values and tree pointers while leaf nodes only have data pointers
- leaf nodes are usually chained in a linked list for easy traversal



Indexes on Multiple Keys

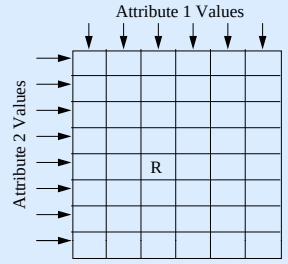
What if the key spans more than one column?

There are several structures for this:

- using **lexographic ordering** - i.e. order on one column, then resolve clashes on this column using values from the next
  - e.g. order on name and, when two people have the same name, order on age.
- use a **hash function** with one parameter for each column
- use a **grid file**

A grid file is a two-dimensional index:

- Considerable space overhead and some increase in insertion/deletion time.
- There are many different kinds
- They speed up 'multi-key' queries
  - i.e. search on two fields at once.
  e.g. searching for X, Y position



Creating Indexes

Indexes can be created as in:

create index authorTitle **on** book (author, title)

– This means order first by author and then by title to break ties

They are destroyed with the **drop** command
or if the table is dropped

Indexes are kept up to date by the system

Each index speeds up searches on the indexed attribute, but ...

Each index takes a little space and also takes time to keep up-to-date
so you don't index everything

You should index any variable you use frequently in searches – e.g. names

In most systems, you should **not** index the primary key, since the system will maintain a primary index automatically

Storage Structures in Microsoft Access

Access provides relatively little control over the structure of the database

– since it is designed for small databases, performance is not a big issue

– also it is designed for casual users who may not understand the effect of different structures

It has two main techniques:

i) Any column of the database can be indexed - there is a field property which determines whether or not it is indexed and you can say whether or not duplicates are allowed

• Indeed, if the column has "ID" in its name, it will be indexed automatically

ii) The database can be split into two parts - the tables in one file, queries, etc. in another

• this allows multiple user interfaces to be sure to access the same physical data and is not really to do with performance

Storage Structures in Oracle I

NB – for more see the Oracle Server Concepts Manual

Oracle databases have a complex internal structure:

i) a database is made up of one or more **tablespaces**:

• a tablespace is a logical partition of the database, so that the DBA can:

i) control disk space for individual users

ii) take some parts of the database off-line without removing the whole database

iii) control space allocation for the data

iv) distribute the database

v) perform backup and recovery

• the database always has at least one tablespace - the **system tablespace**

– this has the data dictionary in it and can never be off-line

Storage Structures in Oracle II

ii) the **create table** command can be used to **partition** a table among tablespaces

– to increase efficiency, particularly of very large databases, and to reduce downtime

– **range partitioning** is used to allocate records to partitions, e.g.

create table Project(Pnumber Number, ControllingDept Number, etc.)
partition by range(ControllingDept)
(**partition** Project1 **values less than**(20) **tablespace** TSA,
partition Poject2 **values less than**(40) **tablespace** TSB)

iii) a tablespace is made up of one or more **data files**:

– these are created to be of a fixed size, reserving disk blocks for this database

– allocation of data to data files is usually controlled by Oracle, not the user nor the DBA

iv) a database also has **redo-log files** – see section on recovery later in the course

Storage Structures in Oracle III

v) a database is also logically split into a number of **segments**, each of which can contain
- some or all of the data from a table
- an index
- rollback data - see transactions slides later in course
- temporary data, created when processing a query

vi) a segment is divided into a number of **extents**:
- an extent contains data of the same type

vii) an extent is a contiguous sequence of **data blocks**
- these are fixed size sections of a data file and have free space in them for new data
- new records, when created, are usually allocated to one data block, but may have to span more than one

Storage Structures in Oracle IV

viii) tables can have indexes associated with them, e.g.

create index empNameIndex **on** Employee(lname, fname)

ix) the tables in a database may also be grouped into logical units - called **clusters**:
- a cluster groups tables which are related, so that related data is brought into memory at the same time
- it is also possible to group records in a cluster using a **hash function** on the key

What the DBA Can Control in Oracle

1. SQL commands to create indexes, clusters, rollback segments, partitions
2. The **create database** command allows the DBA to list the data files
3. There is a **storage** clause to **create table**, create index, create cluster, etc.
   - This determines the size of the first and subsequent segments
4. To add more storage to a database, the DBA can add datafiles, tablespaces or increase the size of a datafile
5. DBA can control use of the **free space in data blocks** using two parameters:
   - PCTFREE - this is the percentage of the data block which will be left free for updates to records which are already there
     new records will not be added if the block becomes fuller than this;
   - PCTUSED - this is the percentage of the file below which data will resume being added to the block, e.g.
     if PCTFREE is 20% and PCTUSED is 50%
     new records will be added until 80% of the block is used;
     no records will be added until deletions drop space used to below 50%.

The Data Dictionary in Oracle

The data dictionary is the storage of meta-data, as used (e.g.) by the **Schema Manager**.

This is stored in the same structure as the data – i.e. as a set of tables and includes the following:
- the names and passwords of Oracle users
- the privileges and roles of these users - see Security slides later in the course
- the names of schema objects (tables, views, indexes, etc.)
- information about integrity constraints
- default values for columns
- space use information
- auditing information
- “other database information”