COM106: Introduction to Databases

Database Management – Storage Structures

Database Management – Storage Structures

Overview

Physical Database Design

Key issues are **efficiency** and **performance**

How to **store** records efficiently on disk

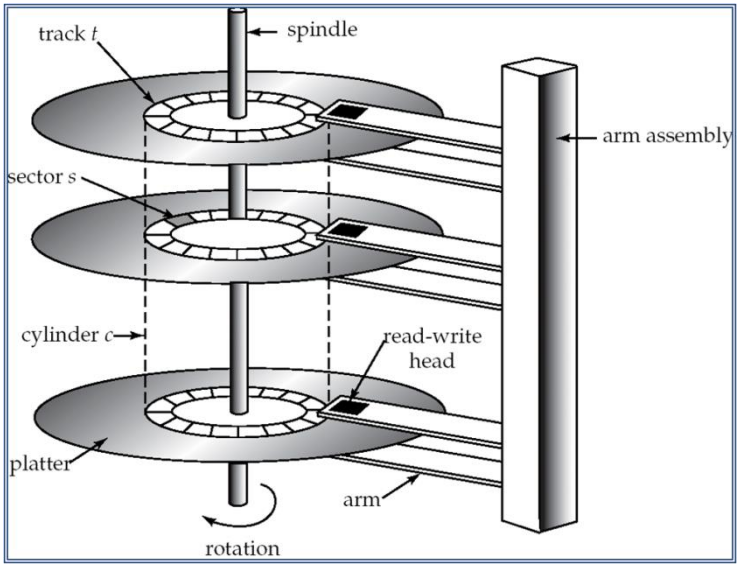
How to **retrieve** records as quickly as possible

Background Issues

Disk Access, Physical Sequence, Virtual Sequence

Available Storage Mechanisms

ISAM, B-Trees, Hashing



Disk Access Review

Physical database design is the process of selecting the appropriate storage representation for database tables.

requires details and frequency of common accesses

Basic Storage Concepts (Hard Disk)

disk access time = seek time + rotational delay

disk access times are **much slower** than access to main memory

the overriding DBMS performance objective is to **minimise** the number of disk accesses (disk I/Os).

Disk access is the slowest component in queries on large tables

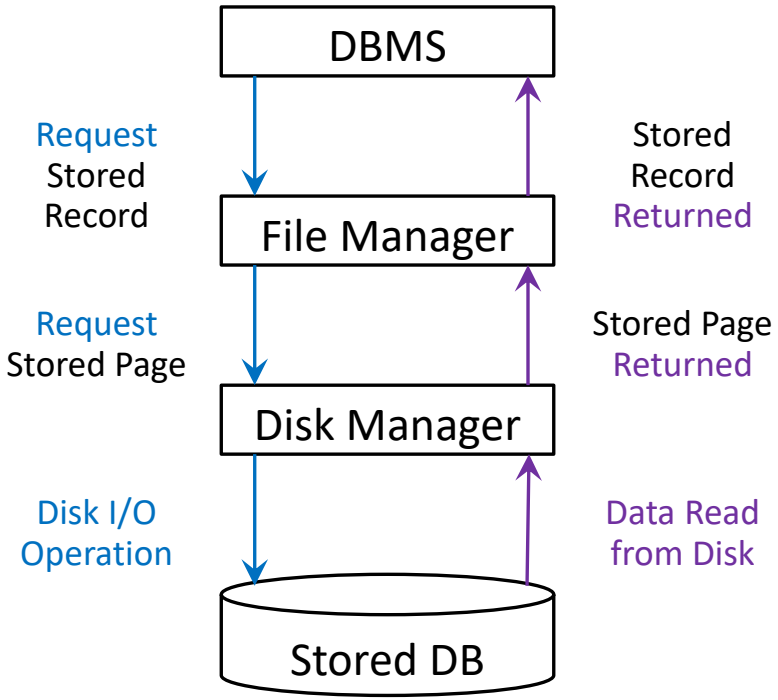
Storage Structures - DBMS/Hard Disk Interaction

File Manager: operating system or DBMS component

- regards the disk as a collection of stored **physical records** (pages) each containing a number of stored **logical records**)
- performs operations such as retrieve, add or remove a **record** from a stored file or create/destroy a stored file

Disk Manager: a component of the operating system responsible for all **physical** I/O operations

- deals with **physical** disk addresses
- performs tasks such as retrieving, adding or removing **pages** of data



Physical vs Logical Records

A **page** (or **block**) – made up of a number of records - is the smallest unit of disk access

Physical Record

- Unit of transfer between disk and primary storage
- Generally contains more than one logical record
- Based on a **page** or **block**, which is a storage unit containing several database records

Generally one disk access involves several pages and hence a number of records

Logical Record

- A database record consisting of a set of field (attribute) values

Storage Structures

Stored files may have more than one method of access:

- **Primary organisation**: based on the physical location of individual records
- **Secondary organisation**: independent of physical storage

File **relationships** implemented as access paths

- As **pointers** between records
- As physical record **clustering** (records from both files stored closely on the same or adjacent pages)

File **organisation** options include **sequential**, **indexing**, **hashing** and **pointer chains**

The storage structure is the arrangement of data on the hard disk

- Many different storage structures are possible (e.g. **ISAM**, **hashing**, **B-trees**)
- Different storage structures have different performance characteristics
- No single structure is best for all applications (depends on table size, access frequency and type, update frequency, etc.)

Sequential Organisation

In a sequentially organised file, records are placed in sequence (e.g. ascending **order** of primary key values)

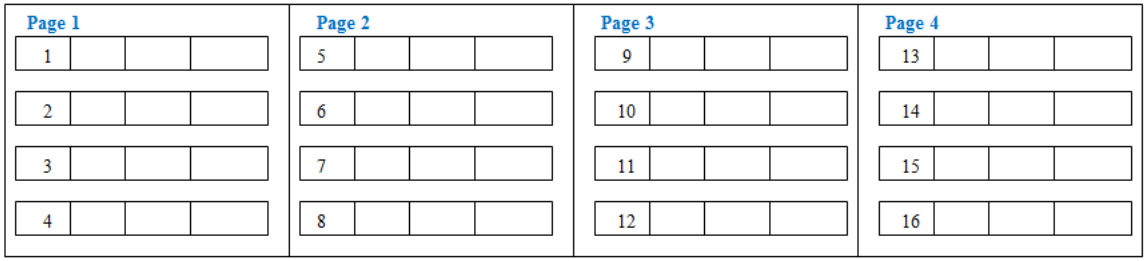
- useful for processing of most records in a table at one go (e.g. update of payroll system main file)

There are two types of sequential organisation:-

Physical sequence:

- records are physically stored in sequence
- insertion of records requires sorting entire file

EMP (enum, ename, salary, dnum) stored in **enum** order

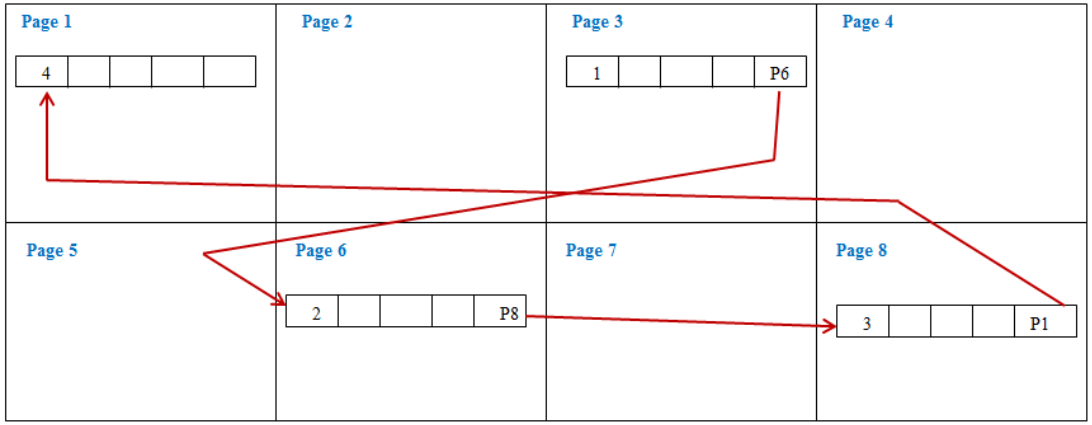


EMP records stored in sequence in consecutive memory pages (**physical sequence**)

Logical sequence:

- stored records are logically linked in sequence by pointers (e.g. in network/ hierarchical databases)
- insertion or deletion requires pointer redirection

EMP (enum, ename, salary, dnum) stored in **logical** order



EMP records not stored in physical sequence but in **logical sequence** using a memory address (page **pointer**) to point to the page holding the next record in sequence (virtual sequence)

Database Storage Structures – File Organisation

File Organisation – The physical arrangement of data in a file into records and pages on secondary storage.

The main types of file organisation are:

- **Heap** (unordered files) - Records are placed on disk in no particular order.
- **Sequential** (ordered) files – Records are ordered by the value of a specified field

  We shall look at **ISAM** (Indexed Sequential Access Method) and **B-Tree**
- **Hash** files – Records are placed on disk according to a **hash function**

The Heap

The basic default storage structure in SQL Server if no **clustered index** is used

Records stored in order they arrive (**no sorting**)

Advantages

Quick loading of bulk records

Good for small tables

Disadvantages

Retrieval requires **all records** to be examined – **sequential (or linear) search**

Poor for tables with significant record changes (higher **volatility**)

So, for tables with a sizeable number of records (100s+) or higher volatility (higher rates of insert, update, delete) a different storage structure is necessary

Retrieving a Record from an Ordered File

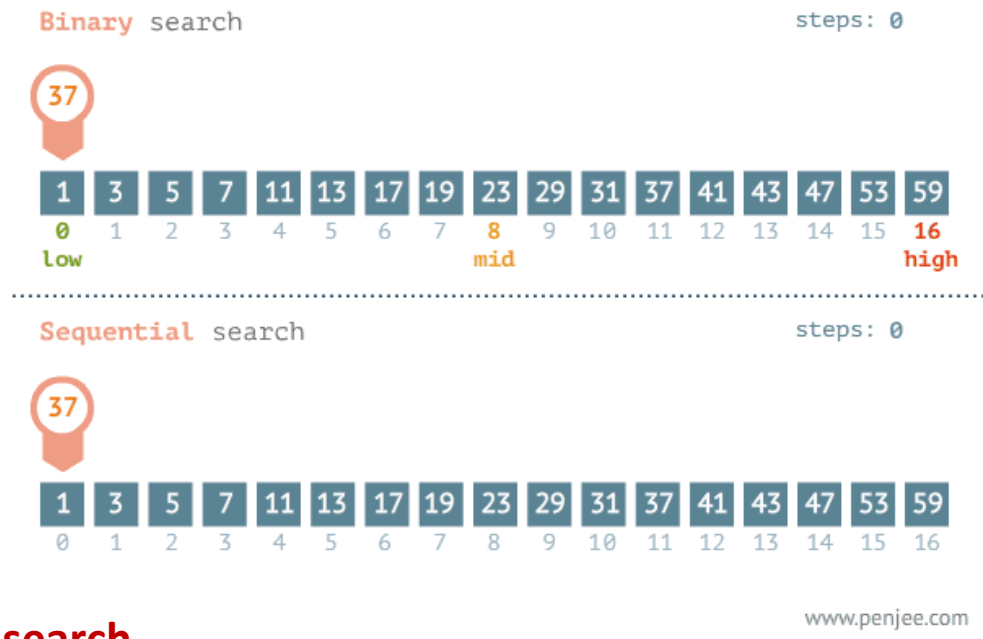
In an **ordered** file, records are stored in order. A **Binary Search** can then be used to find the required record.

Consider the process of finding **'record 37'** in a list of ordered records

Similar to manually finding a persons name in a telephone directory

- Go to the **middle** record in the file.
- If the required record is greater
  - go to the middle record in the **top** half of the file
- Otherwise - go to the middle record in the **bottom** half of the file

Continue this 'divide and conquer' method, continually **halving** the number of records to be searched, until the required value is found.



www.penjee.com

Compare this **binary search** with a **sequential search**

On average, a sequential search of **n** records will take **n/2** accesses.

For a small number of records, sequential search may be faster (less overhead).

For files with over about 100 records, binary search can be significantly more efficient.

Indexing

An **index** is a **data structure** allowing a DBMS to locate particular records more quickly and hence speed up queries

- For example, a book index has each index term (stored in alphabetic order) with a page number
- Likewise, a database index (on a particular attribute) has each attribute value (stored in order) with a memory address

An index gives direct access to a record and prevents having to scan every record sequentially to find the one required

However the index itself has to be searched to find the index entry required

An index can be **clustered** or **nonclustered**

Clustered index: created on attribute that is also physically **sorted** in attribute order (e.g. primary key)

Nonclustered index: created on attribute that is **NOT sorted** in attribute order

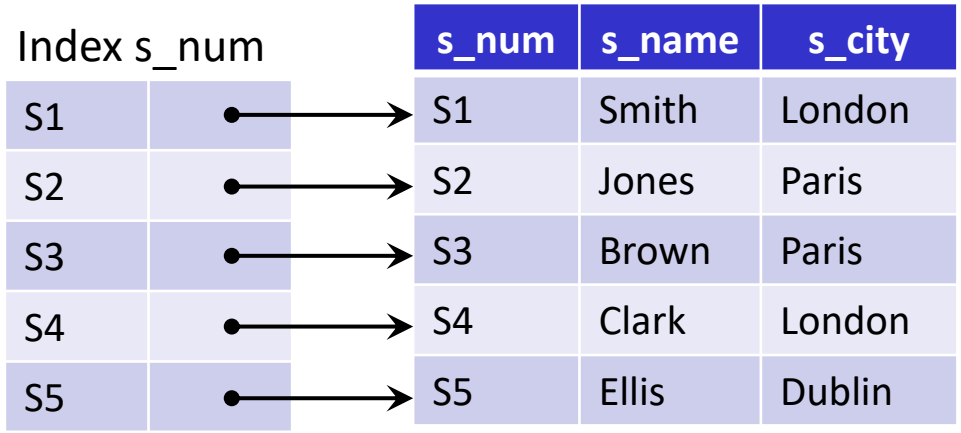
So a table can have **one clustered index**, but **many nonclustered indexes**

An index is composed of an ordered attribute list with a memory address (shown diagrammatically as a pointer) and allows direct access to a particular record.

Consider a relation

SUPPLIER (s_num, sname, s_city)

Supplier table **sorted** in s_num value order



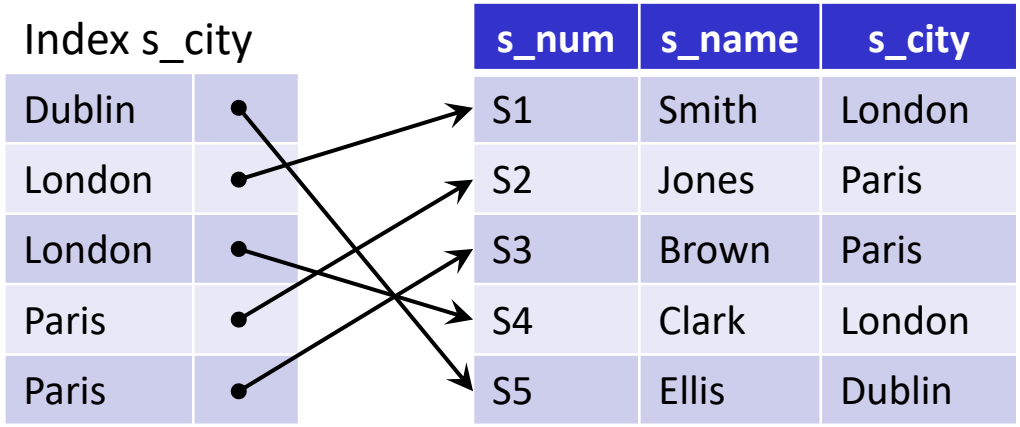
Indexing

Using **SUPPLIER (s_num, sname, s_city)**

Consider the query ***'Get all the suppliers in a certain city (e.g. London)'***

Two possible strategies:

1. Search the entire supplier file for records with city ***'London'***
2. Create an **index** on cities, access it for ***'London'*** entries and follow the pointer to the corresponding records



Indexes are **primary** (on primary key) or **secondary** (on another attribute)

Advantage: speeds up retrieval (at expense of update)

Index Use:

- **Sequential Access** (using sequence of index records)
  useful for range queries (***Get suppliers whose cities begin with L-R***)
- **Direct Access** (using single value of index record)
  useful for list queries (***Get suppliers whose city is London***)
- **Existence tests** (using index access alone)
  Eg. ***Are there any suppliers in Dublin?*** **YES** - if an entry for Dublin exists in the s_city index

Indexing

Multiple Indexes: a file can have any number of indexes (e.g. on s_num **and** on s_city)

a file with an index on every field is **fully inverted**

Indexes on combined attributes: indexes can be constructed on two or more combined attributes (e.g. s_name **and** s_city)

Dense vs. Nondense Indexes: s_city index is a **dense index** (all pointers are record pointers - they point to individual specific records), so **same number of index records as actual records**

Nondense indexes use page pointers instead of record pointers and do not contain an index entry for every record, **so less index records than actual records**

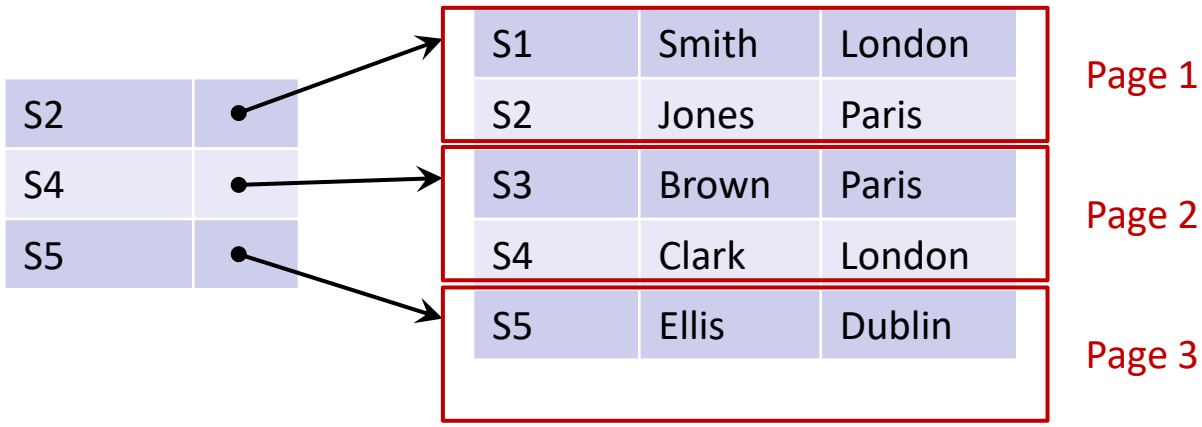
Append/Insert: when a new record is added to a table a new entry must be made in the index at the correct position and the record memory address added (for each index on that table)

Delete: when an existing record is deleted the corresponding index record with memory address must also be deleted (for each index on that table)

Update/Edit: if the attribute on which the table is indexed is changed, the index entry may have to be moved to the correct position in that index.

Indexing

Example of nondense (sparse) index on **s_num**



Retrieved pages are searched to find required records (unlike dense indexes)

Advantage: occupies less space than dense index and quicker to scan

Disadvantage: existence tests cannot be performed on the index alone

A stored file can have at most **one** nondense index (depends on the unique physical sequence of the file). All other indexes **must be dense**.

Indexed Sequential Access Method (ISAM)

Records are physically stored in sorted in order

allows quick sequential processing of all records (e.g. for payroll processing)

A (dense) index is built (normally on primary key)

allows direct access via the index

direct access not as efficient as for **hashing**

Usage

good for key fields

good for files requiring sequential processing

Microsoft ACCESS uses ISAM as its main storage mechanism

MyISAM storage engine is available in MySQL (instead of innobD or Memory)

Index overflow occurs because the space allocated for the index may be exceeded

An improved approach which prevents index overflow is known as **VSAM** (Virtual Sequential Access Method)

B-Trees

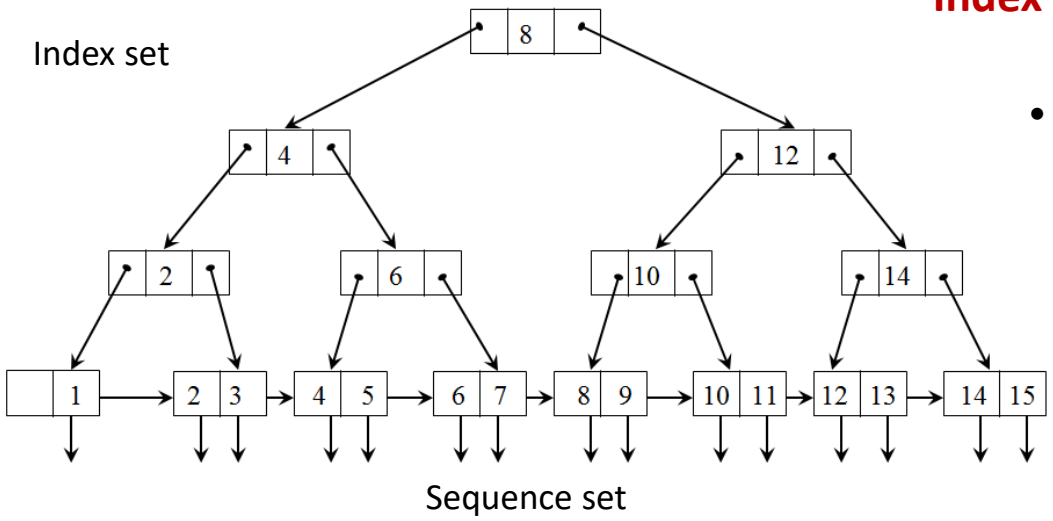
B-Trees are the best all round way to organise indexes (supported by large DBMSs)

Effectively, the **index is indexed** (using a **multi-level, non-dense** index)

A binary (or **n-ary**) access to the records is implemented

Principle:

- indexing removes need for scanning entire file but for large files scanning the large index is a problem
- Index can be treated as a file and indexed. This is continued with each level being a non-dense index to the level below



Index set: a tree-structure index to the sequence set

- provides fast direct access to the **sequence set** and hence to the records

Sequence set: a single level dense index to the data

- provides fast **sequential** access to the indexed data

potential problem of B-trees is that due to **insertion/deletion** the tree may become unbalanced which requires algorithm designed to '**balance**' the tree

B-Trees – Terminology and Rules

A Tree consists of a series of **nodes**, each (except the **root**) having one **parent** node and a number of **child** nodes

A root having no children is called a **leaf node**

Depth of a tree is the maximum number of nodes between the root and a leaf

If the depth is uniform across all paths the tree is **balanced**

The **degree** or **order** of a tree is the maximum number of children allowed per parent (2 in this case)

Access time is proportional to the tree depth, so **shallow bushy trees are best**

If the **root** is not a leaf node, it must **have at least two children**

For a tree of order *n*,

- each node (except root or leaf nodes) must have between *n/2* and *n* pointers and children. (If *n/2* is not an integer, the result is rounded up)
- the number of key values in the leaf node must be between *(n-1)/2* and *(n-1)*. (If *(n-1)/2* is not an integer, the result is rounded up)

The number of key values contained in a non-leaf node is 1 less than the number of pointers

The tree must always be balanced; that is, every path from the root node to a leaf node must have the same length

Leaf nodes are linked in order of key values

B-Trees – Issues

B-Trees require complex maintenance since they **must remain balanced** to be efficient

Insertion and Deletion may **unbalance** the tree (change the depth along some pathways) and hence rebalancing must be performed - **expensive**

The overhead of performing this is **traded off** against the efficiency of access

B-Trees work best for large dynamic (high update rate) tables

Although indexes are intended to enhance a database's performance, there are times when they should be avoided:

- Indexes should not be used on **small tables**.
- Tables that have frequent, large batch **delete or insert** operations.
- Indexes should not be used on columns that contain a high number of **NULL** values.
- Columns that are frequently manipulated should not be indexed.

Hashing

Hashing provides **fast direct access** to a specific stored record on the basis of **attribute value**

Each record stored at page location whose address is **computed as some function of record attribute value – Hash Function**

Record is **retrieved by recalculating page location** and retrieving the record from the computed position

Records in a hash file will appear to be **randomly distributed** across the available file space – also called **random** or **direct** files

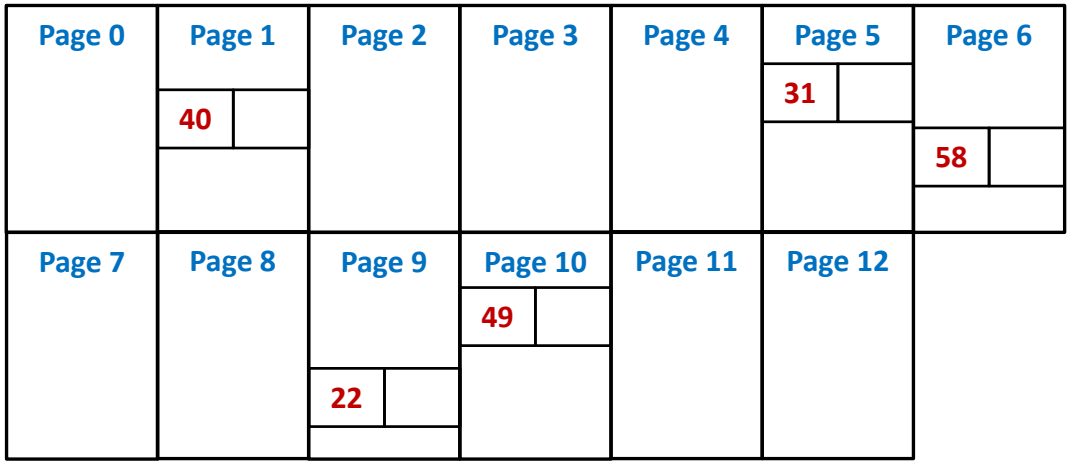
Various **hashing techniques** are used, but the hash function is chosen so that records are as **evenly distributed** as possible throughout the file

A **Division-Remainder** hash function is often used:

- Chose the attribute to be used as the **hash field**
- For each record, **convert the field value to an integer** and use the **MOD function** to generate the page address
  - ie, divide the hash field value by some **predetermined integer** and use the **remainder** as the page address.

e.g. if **s_num** is chosen as the hash field, and using a **MOD 13 hash function**.

Records with s_num **22, 31, 40, 49 and 58** are stored on pages **9, 5, 1, 10 and 6** respectively



Hashing

A stored file can have **many indexes**, but only **one hash structure**

The records are not stored primary key sequence

Advantage: Allows **quicker** direct access than indexing

Speed of hashing = hash function time + 1 disk I/O (record retrieval)

Speed of indexing = index access time (greater) + 1 disk I/O (record retrieval)

Problems: Hash functions **do not guarantee a unique address**

When the same address is generated for a record as a previous record, and that page is full, a **collision** has occurred

On collision the next free page could be used (**Open Addressing**)

- but this could seriously slow access (two or more disk accesses)

A simple solution is to have an **overflow area** (**Unchained Overflow**)

- this necessitates a maximum of two disk accesses to retrieve the record

More sophisticated approaches include adding a pointer to the page the record should have been stored in to indicate where the record is actually stored (**Chained Overflow**)

To store a table using hashing, enough memory must be allocated in contiguous page addresses to hold them

Tables with high update rates may grow and cause marked deterioration in retrieval performance (**increased collision rates**)

Thus, hashed tables may require **periodic reorganisation** (larger area of memory allocated and table completely rehashed)

Storage Options and Comparisons

Not all DBMSs provide all available storage options;

e.g., MS ACCESS: provides ISAM only

MySQL: provides ISAM (default table type), B-tree and Hash

In SQL indexes and storage structures are applied using:

CREATE INDEX indexname ON tablename (columnname) USING indextype (BTree|Hash)

HEAP:

- Unordered - keeps records in order they are entered

Hashing:

- Most efficient method for direct access to an individual record.
- Not efficient for accessing a sequence of records.
- Large amount of 'overflow' (collisions) cause direct access efficiency reduction (requiring physical reorganisation of data on disk)

Indexing:(ISAM)

- Efficient method for small files or non-dynamic files (low update) for sequential and direct access.
- Can also suffer from 'overflow' problems when allocated memory for is exceeded

B-Tree:

- Good all round performer for sequential and direct access.
- Best for large or dynamic files (large update rate) but 'overhead' in B-Tree maintenance.
- Because of tree balancing 'overflow' is not a problem.