Introduction to DBMS

- **Sector :** Minumum track partition for data storage
- **Cluster :** Group of sectors
- **Track :** One cycle on the platter
- **Cylinder :** A semantic shape that consists of same level tracks on the platters

Blocking

- Logical Block

The unit of data transferred between primary and secondary memory

- Blocking Factor

The number of logical record per logical block

Choosing Blocking Factor

- Not too large not too few
- Efected by
  - Operating System
  - Record Length
    - Fixed
    - Length

During a logical READ

- A physical block tranferred to input buffer
- First logical record is deblocked
- Logical record is delivered to the program variables

Access

- **Access Path :** used to find a target record
  - Search mechanism
  - File Organization
  - Secondary storage media
- **Length of an access path:** The number of physical block accessed in the access path

File Design

(a) Satisfies end user' specified requirements
(a) Minimizes the response time

File Design Conderations

- Selection blocking factor
- Organization of physical blocks in secondary storage
- Design of access method
- Select primary key
- File growth

Constraints on file design

- Restricted by hardware and software limitations
- Trade-off between storage allocated to a file and response time:

Allocated Sec. Stor. Space(inc.)
Response Time (dec.)

Complexity(inc.)

- **ENTITY SET**: Set of similiar objects

Employees of companies is employee entitiy set

- **ATTRIBUTE:** Description of entities

For employee entity number, name, deptno, age, adr, salary..etc are attributes.

- **RECORD:**Stores whole information of an entity

Fixed /variable length records

Fixed/variable type records

- **FILE** : Organization of whole data's of one entity set

Employee file is keeps all employee's record

- Operation on Files:
    - Read
    - Write
- Types Of Acces
    - Sequential
    - Random (Direct)
- KEY: One or more field to used for retrieve or sort file
    - Primary Key: One or more field which can take unique value for an entity
    - External Key: Key which is composed aspects of physical storage of record

DATABASE

- **Database :** Collection & organization of related data
    - Represents some aspects of real world called miniworld
    - Logically coherent collection fo data. Random assortment of data con not reffered as db.
    - Designed, builtin populated with data for specific purposes
- **Table:** Basic logical unit for store data.
    - Stores whole information of an entity set
    - Defined by name and coloumns (attribute) set
    - Can be altered after creation
- You can think that

Info. of entity set is stored in files in physical structure , in tables in logical structure

What Is a DBMS?

- A Database Management System (DBMS) is a software package designed to store and manage databases.
- With DBMS
    - Supports large volumes
    - Data independence and efficient access.
    - Reduced application development time.
    - Data integrity and security.
    - Concurrent access, recovery from crashes.

Structure of a DBMS

- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.

Schema for the DBMS levels

- External (Sub) Schema
    - defines the external view of data

as seen by a user or program

- Conceptual Schema
    - defines the logical view of data

as seen by all users and programs

- Physical (Internal) Schema
    - defines the physical view of data

as seen by a DBMS

Physical View

- The DBMS must know
    - exact physical location
    - precise physical structure

Logical View

- The conceptual model is a logical representation of the entire contents of the database.
- The conceptual model is made up of base tables.
- Base tables are “real” in that they contain physical records.
-

What is the significance of cylinders ?

- All information under a cylinder can be accessed without moving the arm that holds readwrite heads

Estimating Capacity

- Function of cylinder, track, sector

Track capacity: Num. of sectors per track
* bytes per sector

Cylinder Capacity: Num. Of tracks per cylinder
* Track Capacity

Driver Capacity : Number of Cylinder
* Cylinder Capacity

- **Interleaving**
- **Extent**
- **Fragmentation**

Access Time = Seek Time +Rotational Delay +Transfer Time

- **Seek time** : from manual of disk
- **Rotational time** : get from one revolution time from manual and divide by two

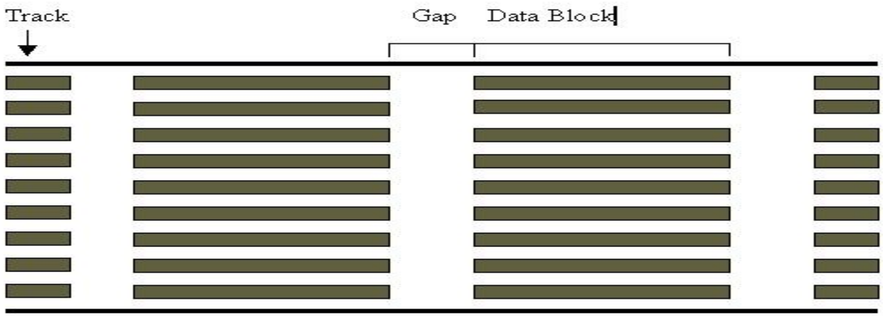
Tranfer time =(Num. of bytes tranferred / Num of bytes on a track) *Rotation time

Redundant Array of Inexpensive Disks(RAID)

- Disk Array: Arrangement of several disks that gives abstraction of a single, large disk.
- Goals: Increase performance and reliability.
    - Data striping: Data is partitioned; size of a partition is called the striping unit. Partitions are distributed over several disks.
    -

Magnetic Tapes
- Sequential: There is no address transformation. It suits logical file organization
- Cheap, large volume
- Use for archieve

Performance Measures
- Tape Density
- Tape Speed
- Size of Interblock Gap

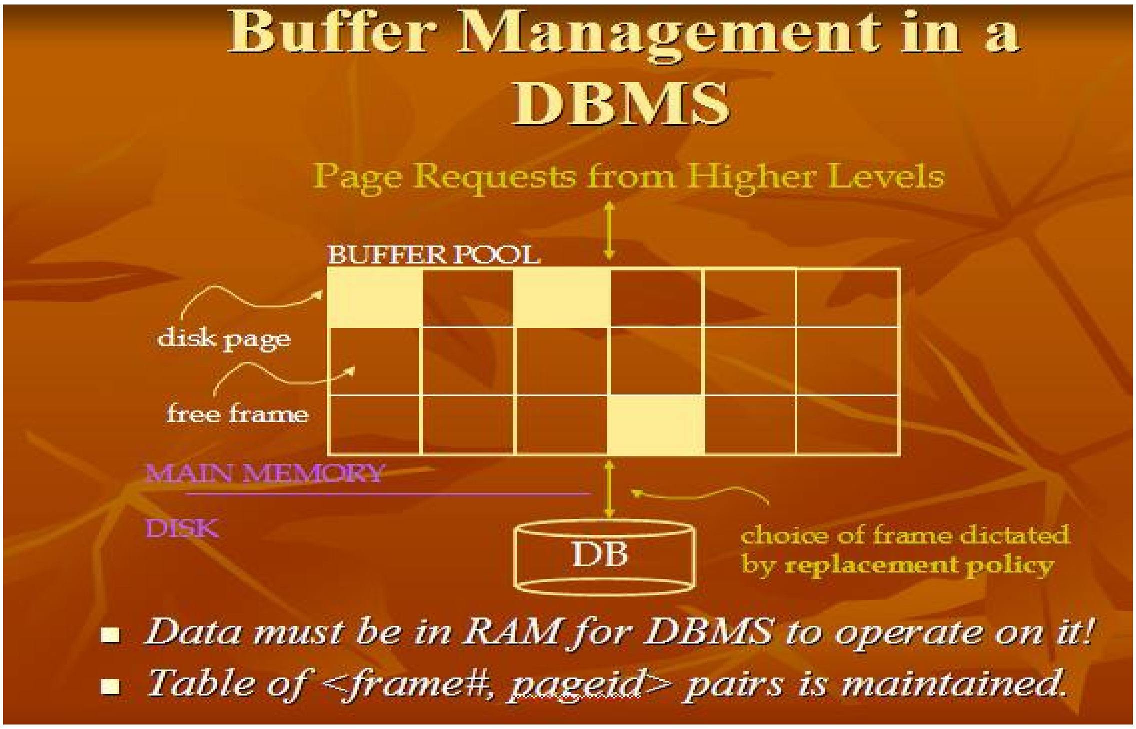


Estimating Tape Length Req.
- S, length of tape
- G, length of interblock gap
- B, length of data block
- N, Number of data blocks

S = N * (B +G)

Disk Space Management
- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed
-

Disk Space Management
- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed



When a Page is Requested ...

- If requested page is not in pool:
  - Choose a frame for replacement
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- Pin the page and return its address.
- If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

More on Buffer Management

- Requestor of page must unpin it, and indicate whether page has been modified:
  - dirty bit is used for this.
- Page in pool may be requested many times,
  - a pin count is used. A page is a candidate for replacement iff pin count = 0.

Buffer Replacement Policy

- Frame is chosen for replacement by a replacement policy:
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the access pattern.
- **Sequential flooding:** Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk,
  - adjust replacement policy, and pre-fetch pages based on access patterns in typical DB operations.

Record/Page Formats

Record Formats

•Organization of records whether field length of record

–Fixed

–Variable

Not: Type and number of fields are identical for all toples

Fixed Length Records

•All fields can be placed continous

•Finding i'th field address requires adding length of previous fields to base address.

Variable Length Records(Cont.)

- In first
  - All previous fields must be scanned to acces desired records
- In Second
  - Second offers direct access to i.th field
  - Points begin and end of the field
  - Efficient storage for null's
  - Small directory overhead

Disadvantage of Variable Length

- If field is growth to larger size,
  - Subsequent fields must be shifted
  - Offsets must be updated
- If changing of field length requires passing to another page,
  - memory adres of page is changed
  - Refernces to old address must be updated
- If changing of field length requires locating on more pages
  - Chaining must be set up for record

Page Formats: Fixed Length Records

In first alternative, moving records for free space management changes memory address of record ; may not be acceptable

Page Formats: Variable Length Records

* Can move records on page without changing memory address of records; so, attractive for fixed-length records too.

Page Formats: Var. Leng. Recs.

- Keep a directory for slots that show <record offset, record length>
- Keep a pointer to point free space
- For placement a record
  - If it is possible, insert in free space
  - Reorganize page to combine wasted space then insert
  - Insert another page
- For deleting a record
  - Put –1 to record offset information in directory

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records.
- **<u>FILE</u>:** A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the records on a page
- There are many alternatives for keeping track of this.

Heap File Implemented as a List

- The header page id and Heap file name must be stored someplace on disk.
- Each page contains two `pointers' plus data.

Heap File Using a Page Directory

- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative

Indexes

- Sometimes, we want to retrieve records by specifying the values in one or more fields, e.g.,
  - Find all students in the "CS" department
  - Find all employees with a age > 30
- <u>Indexes</u> are file structures that enable us to answer such value-based queries efficiently.

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.
- Catalogs are themselves stored as tables!

INDEXES

- An index on a file speeds up selections on the search key fields for the index.
  - Any subset of the fields of a table can be the search key for an index on the relation.
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).
- Primary vs. secondary: If search key contains primary key, then called primary index.
  - Unique index: Search key contains a candidate key.

Index Classification

- **Dense vs. Sparse:** If there is at least one data entry per search key value (in some data record), then dense.
  - Alternative 1 always leads to dense index.
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.

Index Classification

- Composite Search Keys: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.

Sparse vs. Dense Tradeoff

- **Sparse:** Less index space per record can keep more of index in memory
- **Dense:** Can tell if any record exists without accessing file

(Later:

- sparse better for insertions
- dense needed for secondary indexes)

Terms

- Index sequential file
- Search key (≠ primary key)
- Primary index (on Sequencing field)
- Secondary index
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers(not block pointers; not computed)

Summary so far

- ◆ Conventional index
  - Basic Ideas: sparse, dense, multi-level…
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
    - Buckets of Postings List

<u>Outline:</u>

- ◆ Conventional indexes
- ◆ B-Trees ⇒ NEXT
- ◆ Hashing schemes
- ◆ NEXT: Another type of index
  - ◆ Give up on sequentiality of index
  - ◆ Try to get "balance"

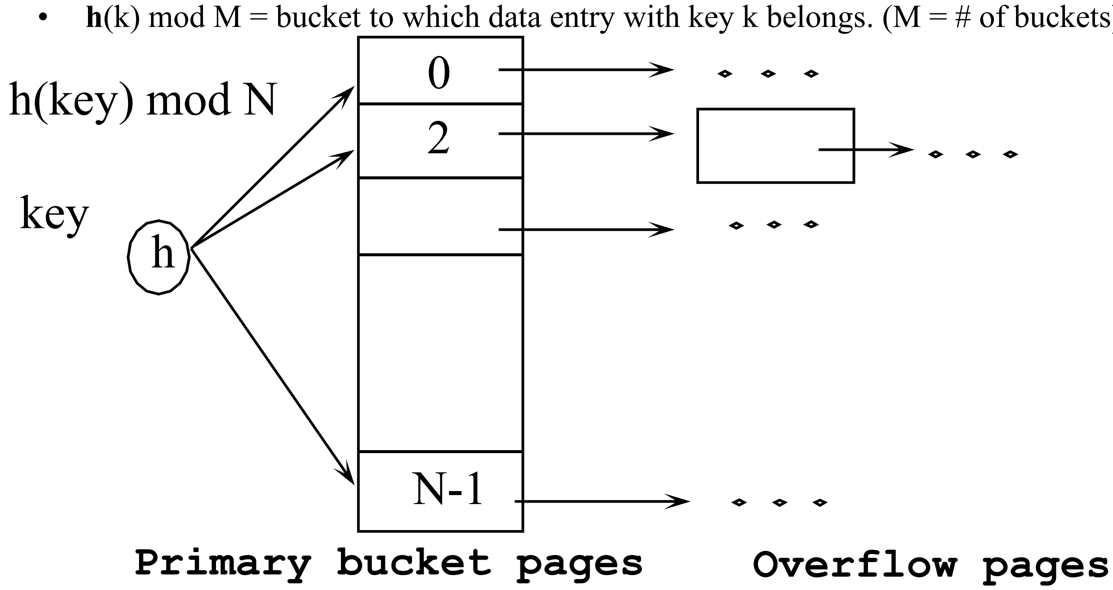
Dynamic Hashing Methods

Introduction

- As for any index, 2 alternatives for data entries k*:
  - Á <k, rid of data record with search key value k>
  - Á <k, list of rids of data records with search key k>
  - Á Choice orthogonal to the indexing technique
- <u>Hash-based</u> indexes are best for equality selections. Cannot support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.
-

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(k) mod M = bucket to which data entry with key k belongs. (M = # of buckets)



Extendible Hashing

– Reading and writing all pages is expensive!
– Idea: Use directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
– Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. No overflow page!
– Trick lies in how hash function is adjusted!

–

How Extendible Hashing works

- Idea from Tries file (radix searching)
  - The branching factor of the tree is equal to the # of alternative symbols in each position of the key

e.g.) Radix 26 trie - able, abrahms, adams, anderson, adnrews, baird

  - Use the first n characters for branching

Extendible Hashing

- H maps keys to a fixed address space, with size the largest prime less than a power of 2 (65531 < 216)
- File pointers point to blocks of records known as buckets, where an entire bucket is read by one physical data transfer, buckets may be added to or removed from the file dynamically
- The d bits are used as an index in a directory array containing 2d entries, which usually resides in primary memory
- The value d, the directory size(2d), and the number of buckets change automatically as the file expands and contracts

Example

- Directory is array of size 4.
- To find bucket for r, take last \`global depth' # bits of **h**(r); we denote r by **h**(r).
  - If **h**(r) = 5 = binary 101, it is in bucket pointed to by 01.
- **Insert**: If bucket is full, split it (allocate new page, re-distribute).
- If necessary, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing global depth with local depth for the split bucket.)

Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us r belongs in A or A2. Last **3** bits needed to tell which.
  - Global depth of directory: Max # of bits needed to tell which bucket an entry belongs to.
  - Local depth of a bucket: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, local depth of bucket = global depth. Insert causes local depth to become > global depth; directory is doubled by copying it over and \`fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- Idea: Use a family of hash functions **h**0, **h**1, **h**2, ...
  - **h**i(key) = **h**(key) mod(2iN); N = initial # buckets

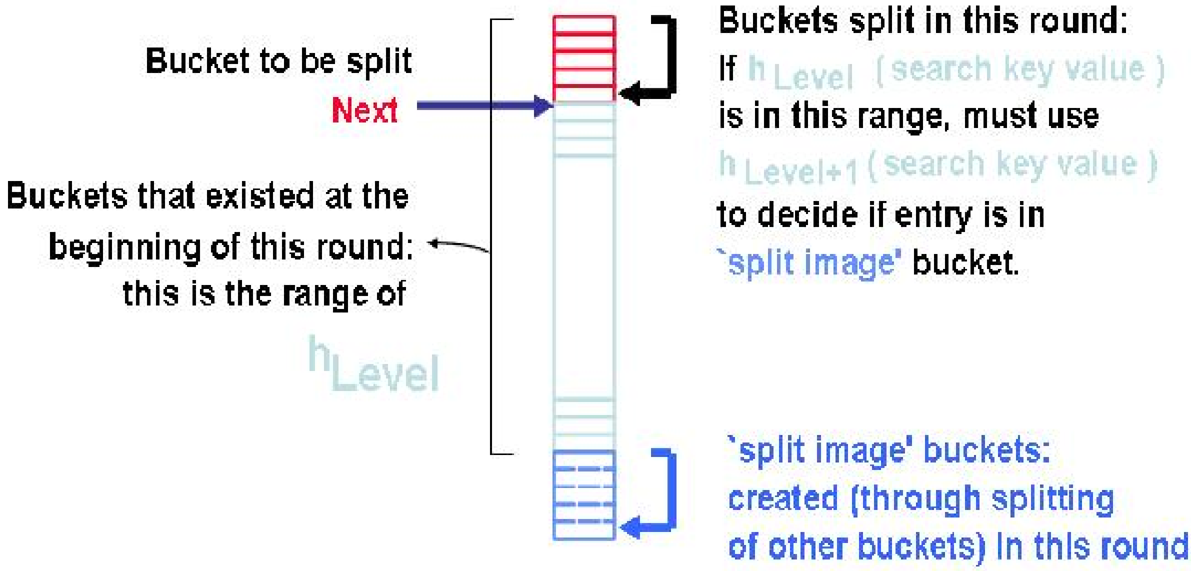
  - **h** is some hash function (range is not 0 to N-1)
  - If N = 2d0, for some d0, **h**i consists of applying **h** and looking at the last di bits, where di = d0 + i.
  - **h**i+1 doubles the range of **h**i (similar to directory doubling)

Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `<u>rounds</u>'. Round ends when all NR initial (for round R) buckets are split. Buckets 0 to Next-1 have been split; Next to NR yet to be split.
  - Current round number is Level.
  - **<u>Search:</u>** To find bucket for data entry r, find **h**Level(r):
    - If **h**Level(r) in range `Next to NR' , r belongs here.
    - Else, r could belong to bucket **h**Level(r) or bucket **h**Level(r) + NR; must apply **h**Level+1(r) to find out.

Overview of LH File

- In the middle of a round.



Linear Hashing (Contd.)

- **<u>Insert</u>**: Find bucket by applying **h**Level / **h**Level+1:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (Maybe) Split Next bucket and increment Next.
- Can choose any criterion to `trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is implicit in how the # of bits examined is increased.

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (Duplicates may require overflow pages.)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed!