MASTER OF COMPUTER SCIENCE/
MASTER OF SCIENCE IN COMPUTER SCIENCE

# Data Storage, Indexing

Dr. Jeevani Goonetillake

UNIVERSITY OF COLOMBO SCHOOL OF COMPUTING

## File Organization and Storage Structures

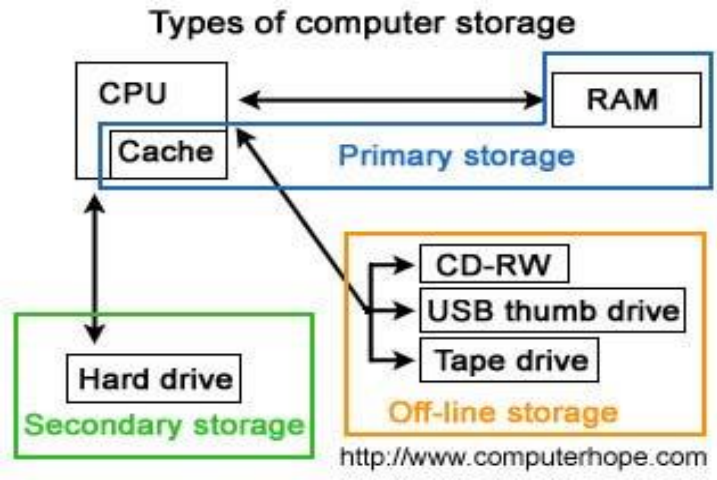
Primary Storage (Main Memory)

- Fast
- Volatile
- Expensive

Secondary Storage (Files in disks or tapes)

- Non-Volatile
- Less-expensive

# Secondary Storage




The image shows three types of storage, but off-line storage is a subset of secondary storage, as they both serve the same purpose and do not interact directly with the CPU.

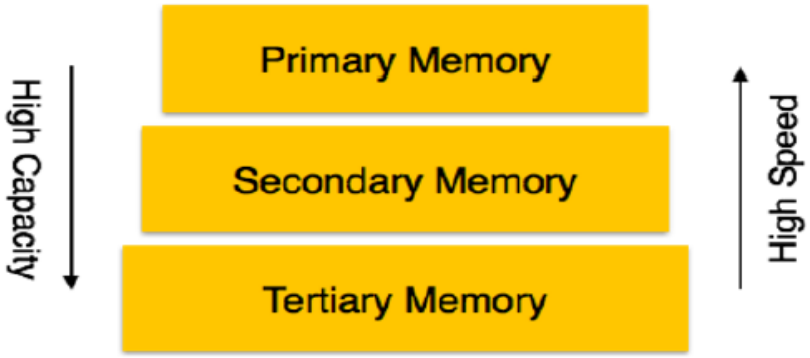


# Secondary Storage

- Disk drives, magnetic drums, data cells and optical disc drives are all classified as direct-access storage devices (DASDs).
- Access methods for DASDs include sequential, indexed, and direct.
- A record on a DASD can be accessed without having to read through intervening records from the current location.
- Direct access contrasts with the sequential access method used in tape drives.
- In sequential access reading anything other than the "next" record on tape requires skipping over intervening records and requires a proportionally long time to access a distant point.

# Organization of Databases

Storage Systems-Memory Hierarchy






# Storage of Databases

- Databases typically store large amounts of data that must persist over long periods of time.
- Most databases are stored permanently (or persistently) on magnetic disk secondary storage.
- The data stored on disk is organized as files of records.
- Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships.
- Records should be stored in a manner that makes it possible to locate them efficiently.

# Disk Storage Devices

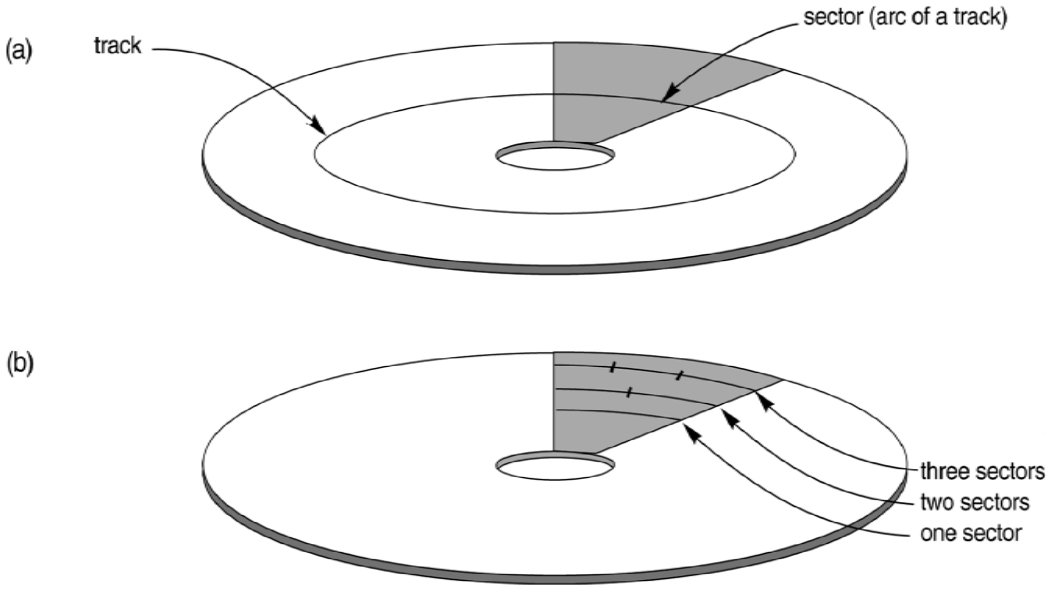
- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A *disk pack* contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular *tracks* on each disk *surface*. Track capacities vary typically from 4 to 50 Kbytes.




# Disk Storage Devices

- Since a track usually contains a large amount of information, it is divided into smaller *blocks* or *sectors*.
- The block size B is fixed for each system.
- Typical block sizes range from B=512 bytes to B=4096 bytes. Whole blocks are transferred between disk and main memory for processing.

# Disk Storage Devices

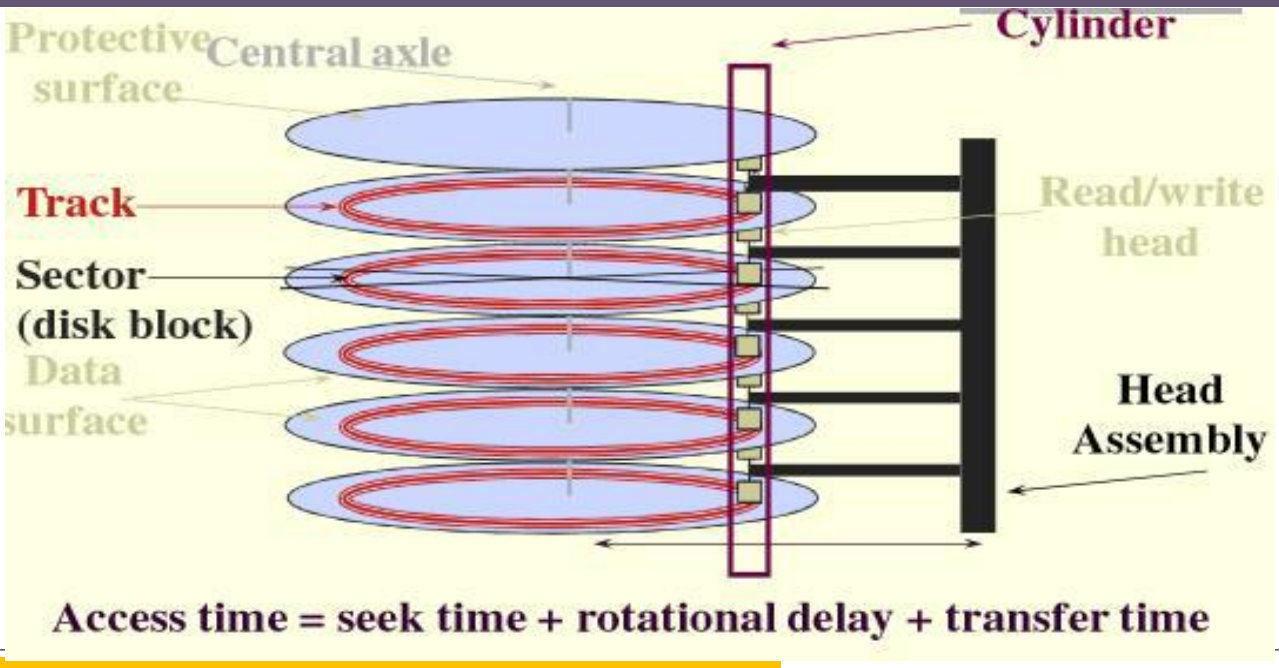





# Disk Storage Devices

- A *read-write* head moves to the track that contains the block to be transferred.
- Disk rotation moves the block under the read write head for reading or writing.
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.

# Disk Storage Devices

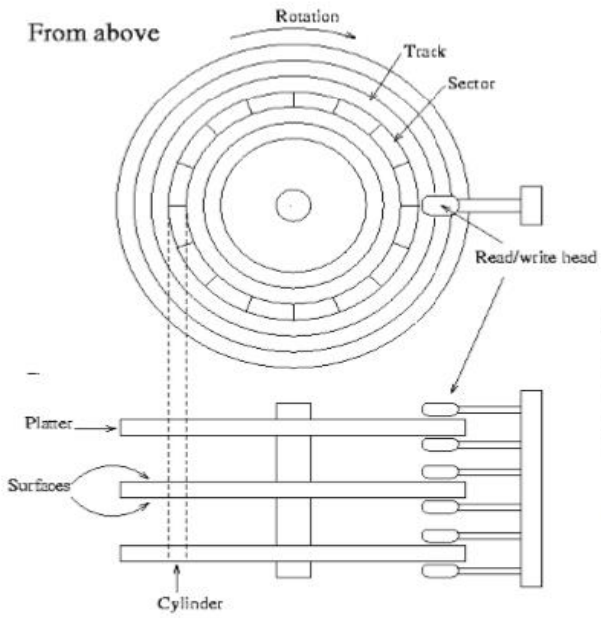


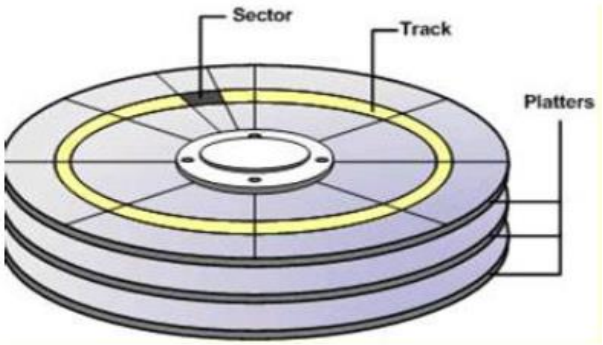



# Disk Storage Devices

- 6 disks (platters) -> 12 surfaces;
- 2 outer protected surfaces, 10 inner data surfaces (coated with a magnetic substance to record data )
- Each surface 200-400 concentric tracks
- Read/write heads placed over specific track; with one active at a time
- Set of corresponding tracks is a cylinder.
  Ex: track 1 of all 10 surfaces.

# Disk Storage Devices




# Blocking

- Blocking: refers to storing a number of records in one block on the disk.
- Blocking factor (*bfr*) refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.

## Data Transfer from Disk

- Address of a block: Surface No, Cylinder No, Block No
- Data transfer: Move the r/w head to the appropriate track
- Time needed

  - seek time -> 12 to 14 ms

  Wait for the appropriate block to come under r/w head

  - rotational delay -> ~3 to 4ms (avg)
- Access time: Seek time + rotational delay
- Blocks on the same cylinder - roughly close to each other
- Access time-wise-cylinder i, cylinder (i + 1), cylinder (i + 2) etc.



## Data Transfer from Disk

When accessing data in a disk;

- Seek time - time to locate the correct track
- Rotational time/Latency - Time to locate correct block in the selected track.
- Data transfer time - Time to transfer data.
- Seek and rotational time are >>> transfer time.
- Locating data on a disk is a major bottleneck in database applications.

## Files of Records

- A file is a *sequence* of records, where each record is a collection of data values (or data items).
- A *file descriptor* (or *file header* ) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records are stored on disk blocks. The *blocking factor bfr* for a file is the (average) number of file records stored in a disk block.


## Data Records and Files

- Fixed length record type: each field is of fixed length
- In a file of these type of records, the record number can be used to locate a specific record.
- The number of records, the length of each field are available in file header.

## Variable Length Record Type

- Arise due to missing fields, variable length fields.
- Special separator symbols are used to indicate the field boundaries and record boundaries.
- The number of records - the separator symbols used are recorded in the file header.


## Record Blocking

- Packing Records into Blocks - Record length much less than block size.
- The usual case

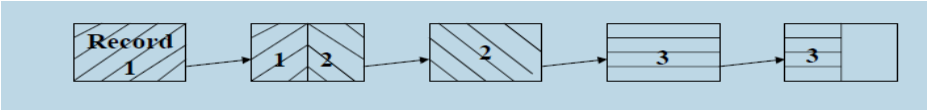
  - Blocking factor bfr = B/r  B -block size (bytes)

    r - record length (bytes) - maximum no. of records that can be stored in a block.

- Normally B may not divide by R exactly, therefore there is unused space in each block.

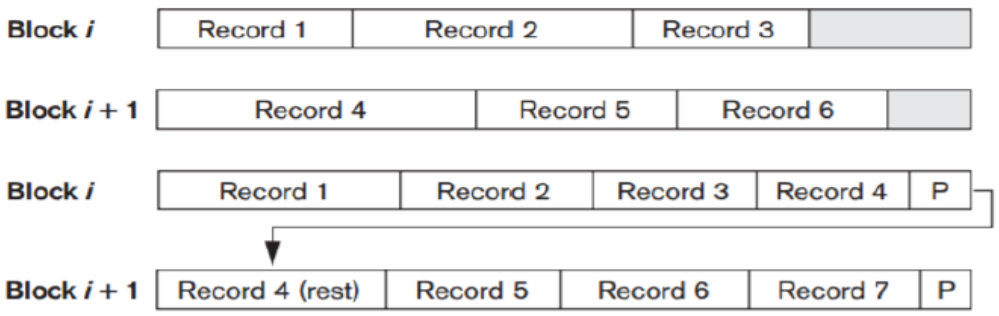
# Record Spanning

- To utilize this unused space, part of a record can be stored on one block and the rest on another.
- A pointer at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk.
- This organization is called spanned.
- Whenever a record length is larger than the block size also, need to use spanned organization.






# Record Spanning




Unspanned

Spanned

Spanned and Unspanned organization

# Mapping File Blocks to Disk Blocks

**Contiguous allocation**

- Consecutive file blocks are stored in consecutive disk blocks
  - Pros: File scanning can be done fast using double buffering
  - Cons: Expanding the file by including a new block in the middle of the sequence –difficult

**Linked allocation**

- Each file block is assigned to some disk block. Each disk block has a pointer to next block of the sequence. File expansion is easy; but scanning is slow.

**Mixed allocation**


# Operation on Files

- **OPEN:** Readies the file for access and associates a pointer that will refer to a current file record at each point in time.
- **FIND:** Searches for the first file record that satisfies a certain condition and makes it the current file record.
- **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition and makes it the current file record.
- **READ:** Reads the current file record into a program variable.
- **INSERT:** Inserts a new record into the file and makes it the current file record.

## Operation on Files

- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY:** Changes the values of some fields of the current file record.
- **CLOSE:** Terminates access to the file.
- **REORGANIZE:** Reorganizes the file records. For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ_ORDERED:** Read the file blocks in order of a specific field of the file.


## Primary file organization Methods

Determine how the records of a file are physically placed on the disk.

- **Heap file** (or unordered file) : places the records on disk in no particular order by appending new records at the end of the file.
- **Ordered file (**sorted file or sequential file) : keeps the records ordered by the value of a particular field (called the sort key).
- **Hashed file:** uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk.
- **B-trees**: use tree structures.

## Unordered Files

- Also called a *heap* or a *pile* file.
- New records are inserted at the end of the file.
- To search for a record, a *linear search* through the file records is necessary. This requires reading and searching half the file blocks on the average and is hence quite expensive.
- Record insertion is quite efficient.
- To delete a record, the record is marked as deleted. Space is reclaimed during periodical re-oganization.


## Unordered Files

- Records are appended to the file as they are inserted Simplest organization
- Insertion -Read the last file block, append the record and write back the block –easy
- Locating a record given values for any attribute - requires scanning the entire file –very costly

# Ordered Files

- Also called a *sequential file*.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the *correct order*.
- A *binary search* can be used to search for a record on its *ordering field value*. This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.


# Ordered Files

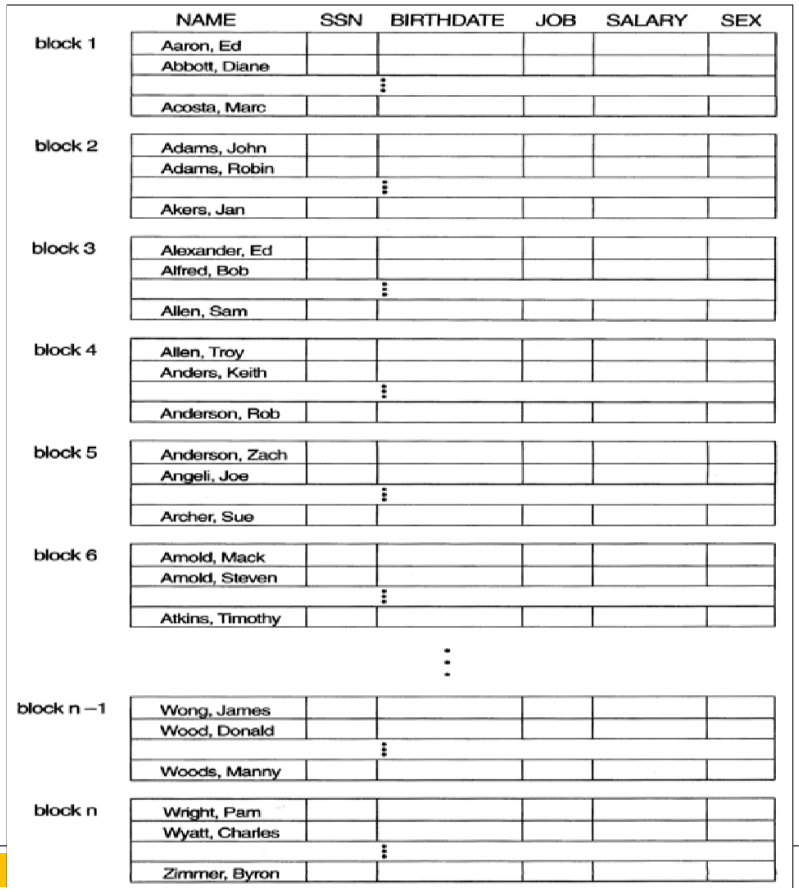
- Inserting a new record:
  - Ordering gets affected. Costly as all blocks following the block in which insertion is performed may have to be modified
  - Hence not done directly in the file. All inserted records are kept in an auxiliary file. Periodically file is reorganized -auxiliary file and main file are merged.
- Locating record
  - Carried out first on auxiliary file and then the main file.
- Deleting a record
  - Deletion markers are used.

# Binary Search on the Ordering key

```
l ← 1; u ← b; (* b is the number of file blocks *)
while (u ≥ l ) do
    begin i ← (l + u) div 2;
    read block i of the file into the buffer;
    if K < (ordering key field value of the first record in block i )
        then u ← i – 1
    else if K > (ordering key field value of the last record in block i )
        then l ← i + 1
    else if the record with ordering key field value = K is in the buffer
        then goto found
    else goto notfound;
    end;
goto notfound;
```



# Ordered Files



**Oracle**

The following statement creates the sales table with linear ordering:

```
CREATE TABLE sales (
  prod_id       NUMBER(6) NOT NULL,
  cust_id       NUMBER NOT NULL,
  time_id       DATE NOT NULL,
  channel_id    CHAR(1) NOT NULL,
  promo_id      NUMBER(6) NOT NULL,
  quantity_sold NUMBER(3) NOT NULL,
  amount_sold   NUMBER(10,2) NOT NULL
  )
CLUSTERING  BY LINEAR ORDER (cust_id, prod_id);
```

This clustered table is useful for queries containing a predicate on cust_id or predicates on both cust_id and prod_id.

or

ALTER TABLE sales ADD CLUSTERING BY LINEAR ORDER (**cust_id, prod_id**)

**MYSQL table**

ALTER TABLE tablename ORDER BY columnname ASC;

# Average Access Times

The following table shows the average access time to access a specific record for a given type of file.

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

| TYPE OF ORGANIZATION | ACCESS/SEARCH METHOD | AVERAGE TIME TO ACCESS A SPECIFIC RECORD |
|---|---|---|
| Heap (Unordered) | Sequential scan (Linear Search) | $b/2$ |
| Ordered | Sequential scan | $b/2$ |
| Ordered | Binary Search | $\log_2 b$ |

# Hashed Files

- The file blocks are divided into M equal-sized *buckets*, numbered bucket0, bucket1, ..., bucket M-1.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the *hashing function*. One common hash function is h(K) = K mod M, which returns the remainder of the integer value K after division by M; this value is then used to find the record.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full. An overflow file is kept for storing such records.


# Hashed Files

There are numerous methods for collision resolution, including the following:

***Open addressing:*** Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

***Chaining:*** A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.

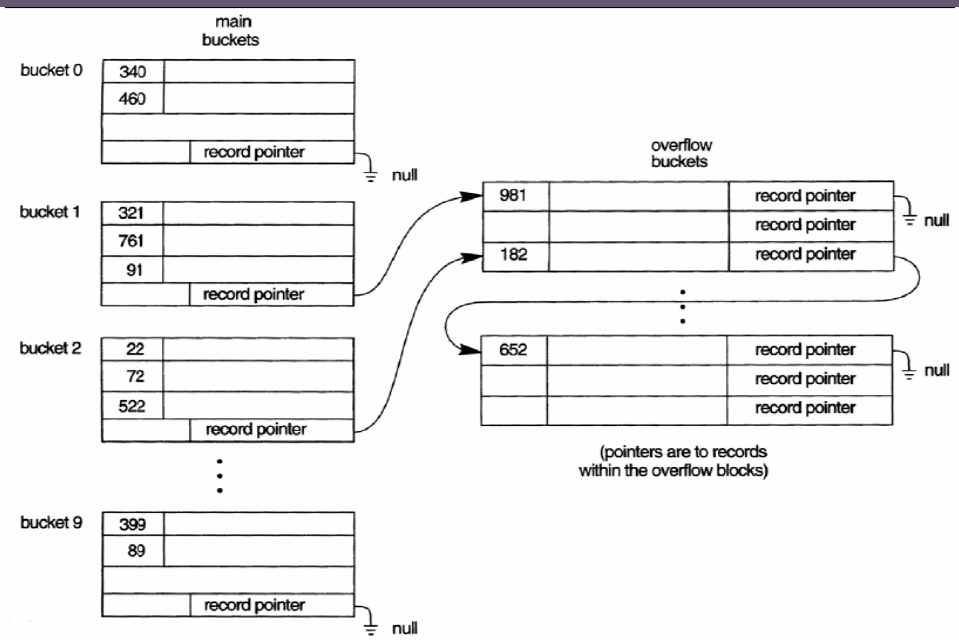
**Multiple hashing:** The program applies a second hash function if the first results in a collision.

# Hashed Files

- The hash function h should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of *static* hashing:

  Fixed number of buckets M is a problem if the number of records in the file grows or shrinks.


# Hashed Files




# Hash Clusters in Oracle

- Using hash organization in Oracle requires the use of hash clusters. The database physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function.
- It is possible to create a sorted hash cluster where the rows corresponding to each value of the hash function are sorted on a specified set of columns in ascending order, which can improve response time during subsequent operations on the clustered data.


# Hash Clusters in Oracle

In the following SQL statements, the telephone_number column is the hash key. The hash cluster is sorted on the call_timestamp and call_duration columns. The number of hash keys is based on 10-digit telephone numbers.

```
CREATE CLUSTER call_detail_cluster (
  telephone_number NUMBER,
  call_timestamp NUMBER SORT,
  call_duration NUMBER SORT )
 HASHKEYS 10000 HASH IS telephone_number
 SIZE 256;

CREATE TABLE call_detail (
  telephone_number     NUMBER,
  call_timestamp       NUMBER   SORT,
  call_duration        NUMBER   SORT,
  other_info           VARCHAR2(30) )
 CLUSTER call_detail_cluster (
  telephone_number, call_timestamp, call_duration );
```

Given the sort order of the data, the following query would return the call records for a specified hash key by oldest record first.

```
SELECT * WHERE telephone_number = 6505551212;
```

# Hashed File Limitation

- **Inappropriate for some retrievals:**
  based on pattern matching
  eg. Find all students with ID like 98xxxxxx.
- Involving ranges of values
  eg. Find all students from 50100000 to 50199999.
- Based on a field other than the hash field.

| Student ID | Tutorial | Grade |
|---|---|---|
| 50195255 | T01 | A |
| 50194525 | T02 | A |
| 98076543 | T01 | A+ |


MASTER OF COMPUTER SCIENCE/
MASTER OF SCIENCE IN COMPUTER SCIENCE

# Indexing

Dr. Jeevani Goonetillake



## Indexing

Assuming that a file already exists with some primary organization such as unordered, ordered or hashed organizations an additional access structures is defined as indexes.

The index structures are additional files on disk that provide secondary access paths, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk.


## Indexing

- Indexes are used to speed up the retrieval of records in response to certain search conditions based on the indexing fields that are used to construct the index.

- Any field of the file can be used to create an index, and multiple indexes on different fields—as well as indexes on multiple fields—can be constructed on the same file.

## Indexes

- **Data file**: a file containing the logical records
- **Index file**: a file containing the index records
- **Indexing field**: the field used to order the index records in the index file

The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. Each index entry i is of the form <K(i), P(i)> where k is the search key and P(i) is the corresponding block pointer.

The values in the index are ordered so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient.


## Indexes

- Index: A data structure that allows particular records in a file to be located more quickly

  ~ Index in a book

- An index can be sparse or dense:
  - Sparse: record for only some of the search key values (eg. Staff Ids: CS001, EE001, MA001). Applicable to ordered data files only.
  - Dense: record for every search key value. (eg. Staff Ids: CS001, CS002, .. CS089, EE001, EE002, ..)

# Different types of indexes

**Single Level Indexes**

Primary indexes

Specified on ordering key field of an ordered file.

Clustering indexes

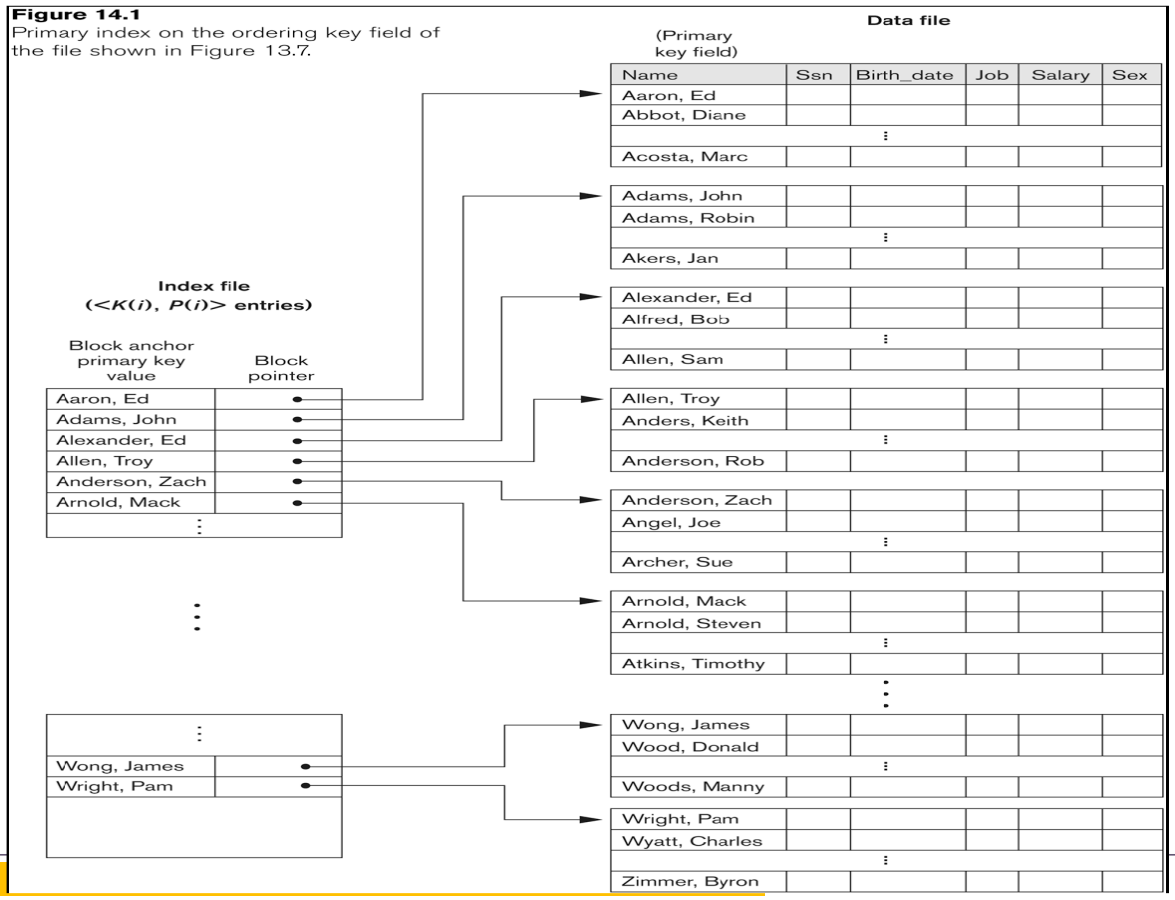
specified on ordering field of an ordered file, but ordering field is not a key field (i.e., search key not unique - multiple records with same value of field may exist).

Secondary indexes

can be specified on any non-ordering field.

# Primary Index

- Defined on an ordered data file.
- The data file is ordered on a key field.
- 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 primary index is a non-dense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.
- if the primary index file contains *b* blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.




# Primary Index

- Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed size and are unspanned, with record length R = 100 bytes. How many block accesses are required to search a record on the data file?
- The blocking factor for the file would be bfr = $L(B/R)\rfloor$ =$L(1024/100)\rfloor$
  = 10 records per block.
- The number of blocks needed for the file is b = $\lceil(r/bfr)\rceil$
  = $\lceil(30,000/10)\rceil$ = 3000 blocks.
- A binary search on the data file would need approximately $\lceil\log_2 b\rceil$
  = $\lceil(\log_2 3000)\rceil$ = 12 block accesses.

# Primary Index

- Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file. How many block accesses are required to search a record using the index?
- The size of each index entry is $R_i$ = (9 + 6) = 15 bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block.
- The total number of index entries $r_i$ is equal to the number of blocks in the data file, which is 3000.


# Primary Index

- The number of index blocks is hence $b_i = \lceil (r/bfr_i) \rceil = \lceil (3000/68) \rceil = 45$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ block accesses.

- To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses--an improvement over binary search on the data file, which required 12 block accesses.