



# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

### Module IV

PYSICAL DATA ORGANIZATION AND QUERY OPTIMIZATION

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

## Module 4: PHYSICAL DATA ORGANIZATION AND QUERY OPTIMIZATION

- **Physical Data Organization:** index structures, primary, secondary and clustering indices, Single level and Multi-level indexing, B+-Trees (basic structure only, algorithms not needed), (Reading Elmasri and Navathe, Ch. 17.1-17.4)
- **Query Optimization:** heuristics-based query optimization, (Reading Elmasri and Navathe, Ch. 18.1, 18.7)

# **Physical Data Organization**

Index structures

# File organisation

- Sorted files
  - Records stored sequentially on disk
  - Sorted by primary key
- Searching sorted files
  - Consider queries such as  
**select \* from instructor where id = 32343**
  - Can be implemented using binary search
    - Look in the middle of the file
    - If equal to 32343 we are done
    - If larger than 32343 look in the middle of first half
    - Else look in the middle of the second half
    - Repeat until found

# Searching sorted files

- Consider queries such as

**select \* from instructor where name = ‘Einstein’**

Sorted structure on *id* does not help us here

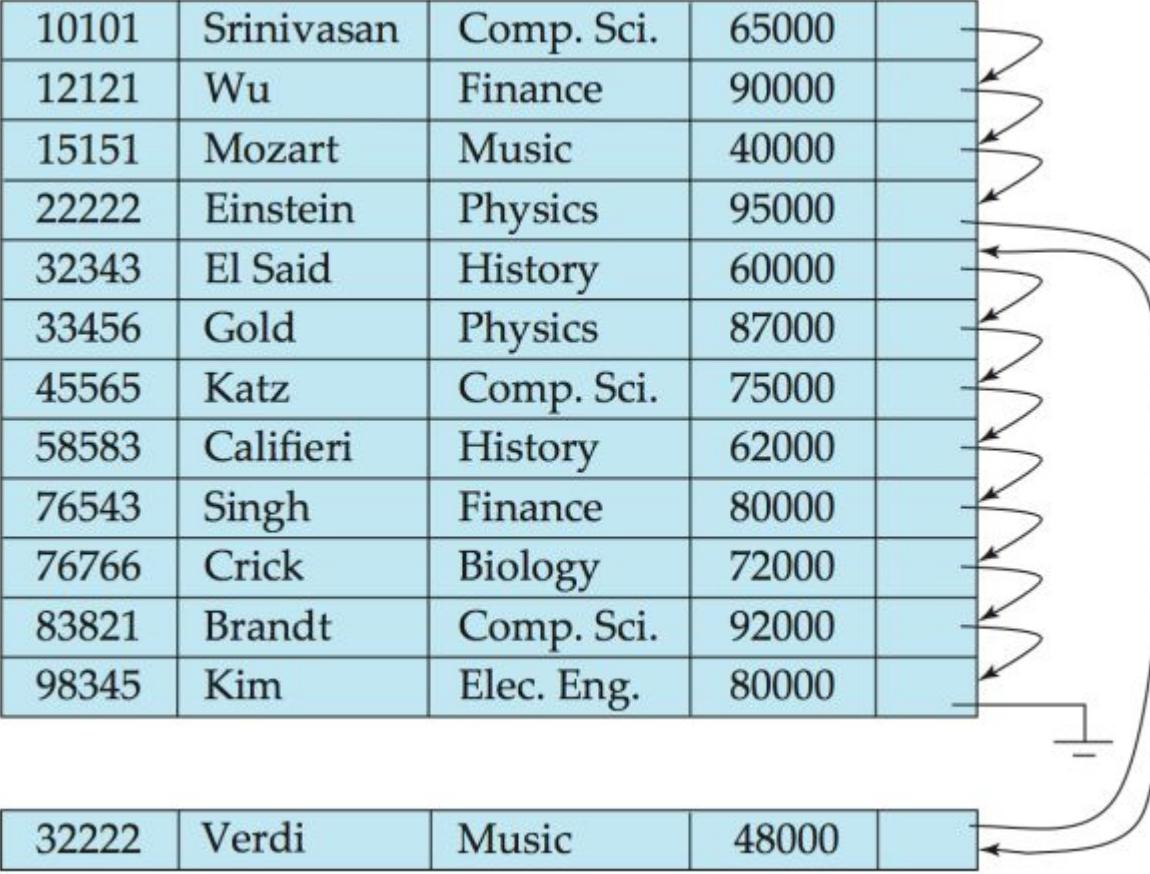
- Must use linear search
- Look through file one record at a time
- Number of disk access operations linear in file size
- Very slow if table is large

# Deleting from sorted files

- Simply delete records
- Keep list of available space for future insertions
- Reorganise when file sparsely populated

## • Inserting Records

- If space insert
- Otherwise insert into overflow block
- Leave space for insertions (fill factor < 100%)



The diagram illustrates a sorted file structure. On the left is a main table with 12 rows of student data. On the right is a separate row for a new record that is being inserted. Arrows show the movement of data from the new record to the main table, indicating the insertion process.

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	
-------	-------	-------	-------	--

# Spanned and Unspanned records

- A spanned record is a **variable-length record in which the length of the record can exceed the size of a block**. If it does, the record is divided into segments and accommodated in two or more consecutive blocks.
- If records are not allowed to cross block boundaries, the organisation is called unspanned.
- Blocking factor is the number of records in a block at a particular time.
- If the files are numbered  $0, 1, 2, 3, \dots, r-1$  and records in the blocks are numbered as  $0, 1, 2, \dots, bfr-1$ , where  $bfr$  is the blocking factor, then the location of the  $i^{\text{th}}$  record of a file can be determined by :  $[i/bfr]$  and it would be  $(i \bmod bfr)^{\text{th}}$  record in the block.

# Calculating how many blocks our data needs

- We have 10 000 000 records. Each record is 80 bytes long. Each record contains an unique key (Lets say social security numbers).
- Each block is 5000 byte. We want calculate how many blocks we need. First, we need to know how many records fit into a single block:
- Blocking factor = floored((Block size)/(Record size)) = floored(5000/80) = floored(62.5) = 62 record/block
- And we have 10000000 records, so we need ceiled(10000000/62)=ceiled(161290.32)=161291 blocks to store all that data.

# The problem of maintaining sorted files

- Overflow block may increase in size over time
- Overflow blocks are not sorted
- After many insertions advantage of sorted file is lost
- May need to reorganise
- Can be done e.g. at night
- Hence Sorted files are costly to maintain.

# More problems

- Should file be kept on contiguous blocks?
- How else can we find the middle of the file?
- But this makes maintenance harder
- What happens when file cannot be extended further on disk?

Solution: Use Indices

# Index: Introduction

- Indexes, which are used to speed up the retrieval of records in response to certain search conditions, by minimizing the number of disk accesses required when a query is processed.
- The index structures typically provide secondary access paths, which provide alternative ways of accessing the records without affecting the physical placement of records on disk.
- They enable efficient access to records based on the indexing fields that are used to construct the index.
- Basically, *any field* of the file can be used to create an index and *multiple indexes* on different fields can be constructed on the same file.

# Index structure:

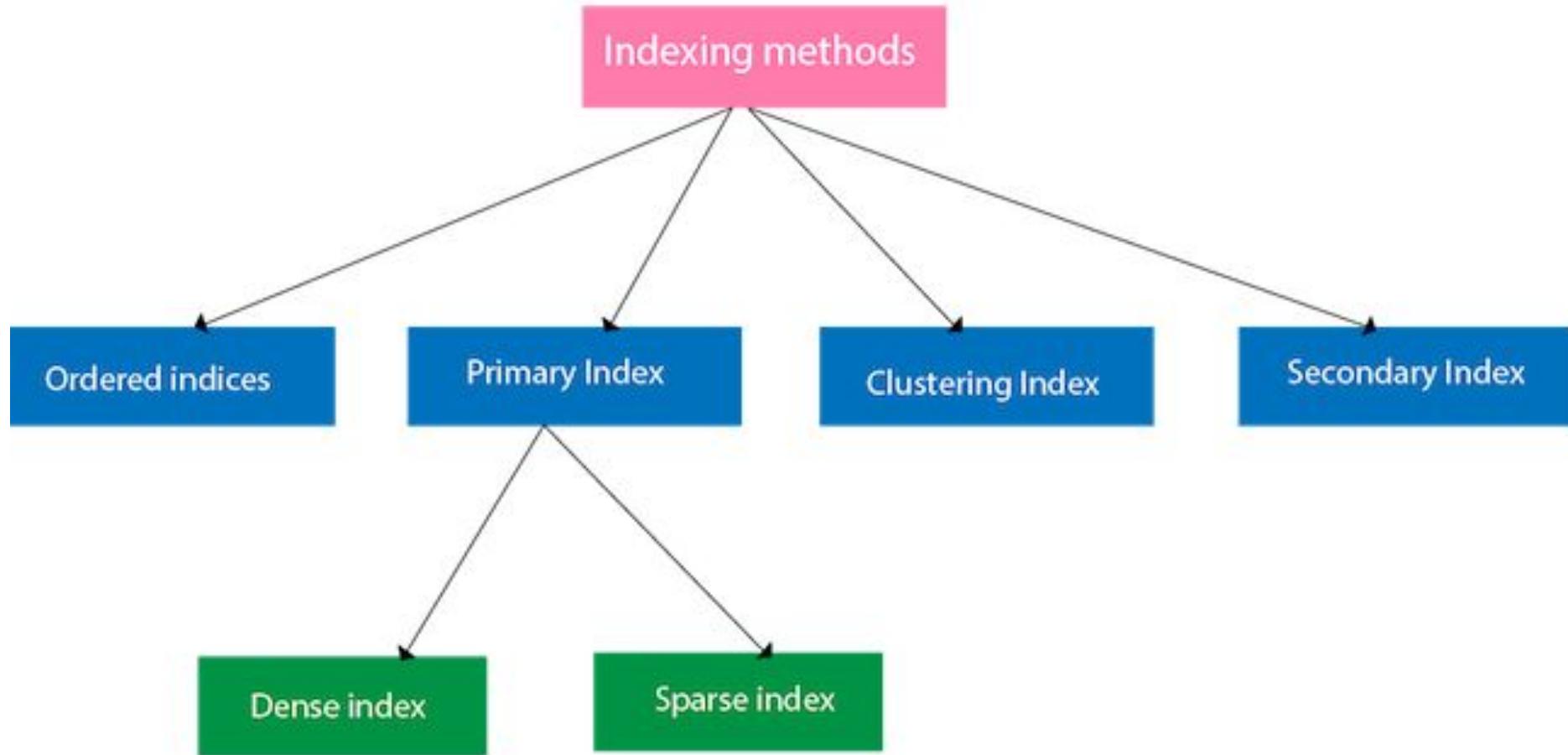
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.
- Indexes can be created using some database columns.
- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Search key	Data Reference
------------	----------------

# Index Structures

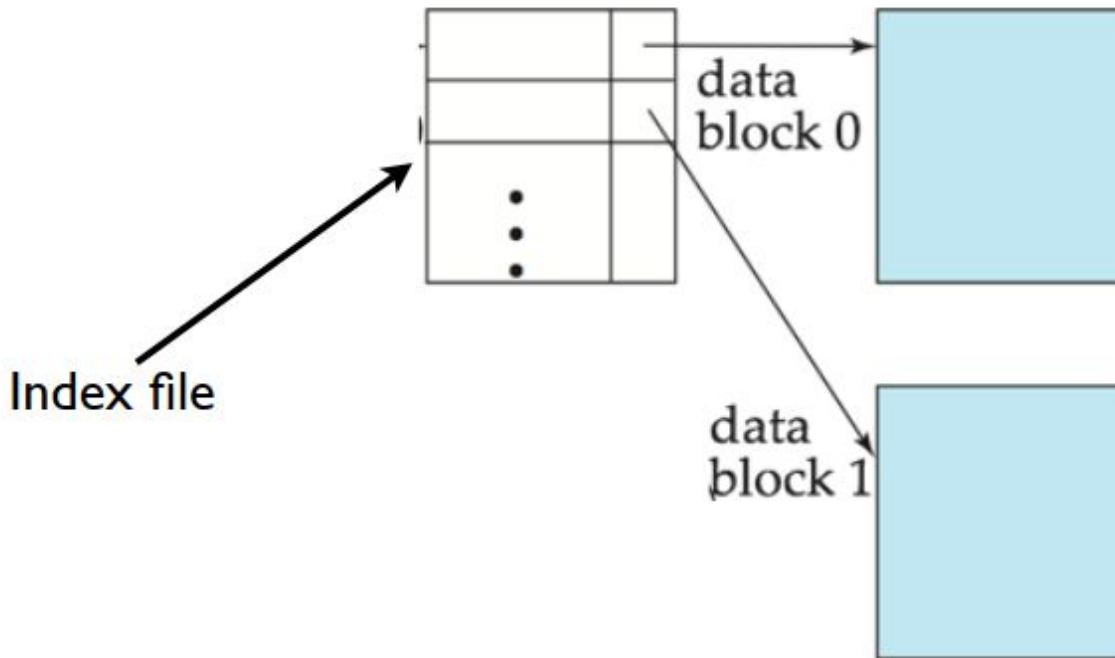
- A variety of indexes are possible; each of them uses a particular data structure to speed up the search.
- To find a record or records in the file based on a certain selection criterion on an indexing field, one has to initially access the index, which points to one or more blocks in the file where the required records are located.
- Indexes can also be constructed based on hashing or other search data structures.
- Type of indexes:
  - Single-level indexes (ordered files)
  - Multilevel indexes (tree data structures)- such as  $B^+$  trees for RDBMS's.

# Indexing Methods



# Using a sparse index

- Blocks really need not be contiguous
- For search we can use a sparse index
- Index file much smaller than data file
- Note that overflow blocks no longer needed
- For search purposes index file should be kept sorted



# Ordered indices

- The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.
- **Example:** Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.
- In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading  $543 \times 10 = 5430$  bytes.
- In the case of an index, we will search using indexes and the DBMS will read the record after reading  $542 \times 2 = 1084$  bytes which are very less compared to the previous case.

# Single-level ordered indexes

- The idea behind an ordered index access structure is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book.
- We can search an index to find a list of addresses-page numbers in this case-and use these addresses to locate a term in the textbook by *searching* the specified pages.
- The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a linear search on a file.
- Of course, most books do have additional information, such as chapter and section titles, that can help us find a term without having to search through the whole book.
- However, the index is the only exact indication of where each term occurs in the book.

# Types of Single-Level Ordered Indexes

- Type of single-level indexes – based on an **indexing field (attribute)** and is similar to the index used in a textbook.
- All of them involve ordered **index files** much like the alphabetical order for terms used in a book followed by its page number.
- For the given records, each consisting of several fields/attributes, the index is a single field/attribute of a file called **indexing field**.
- The index stores values along with pointers to disk blocks that contain the record. Since values in the index are ordered, we can do with a **binary search**.

# Types of Single-Level Ordered Indexes

1. **Primary index** - specified on the **ordering key field**: The field is used to *physically order* the file records on disk. The value of the ordering key field for each record must be **unique**.
2. **Clustering index** - At most one physical ordering => either have one Primary Index or one Clustering Index, but not both.
3. **Secondary index**- It is an index specified on a **non-ordering** field of a file. A file can have several **secondary indexes** in addition to its primary access method.

# Primary Index

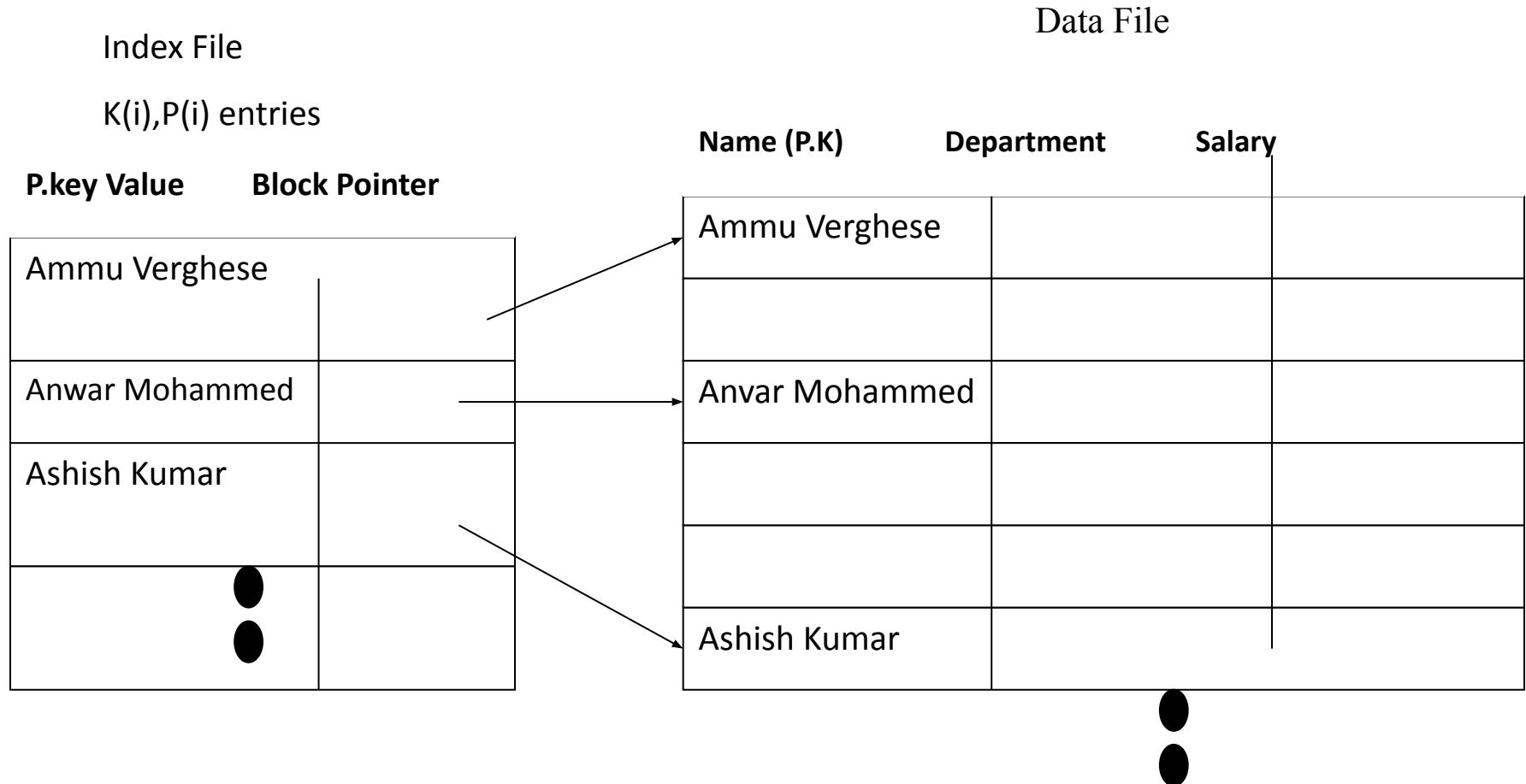
- If the index is created on the basis of the primary key of the table, then it is known as primary indexing.
- The data file is ordered by the **index field**.
- The **Primary index file** has two fields; one is the **primary key field** and the other is a **pointer** to a disk block (address) –**K(i),P(i)**

where K = ordering key field, P = pointer to the block of data:

- There exist an entry for each block and each entry points to the first record in that block (anchor record)
- Requires the ordering field to have a distinct value for each record. In the example below we use the ‘Name’ as the primary index on the ordered file. Therefore  $\langle K(1)=\text{Ammu Varghese} \& P(I)=\text{address of block 1} \rangle$ . The total number of entries in the index file equals the total number of records in the data file.

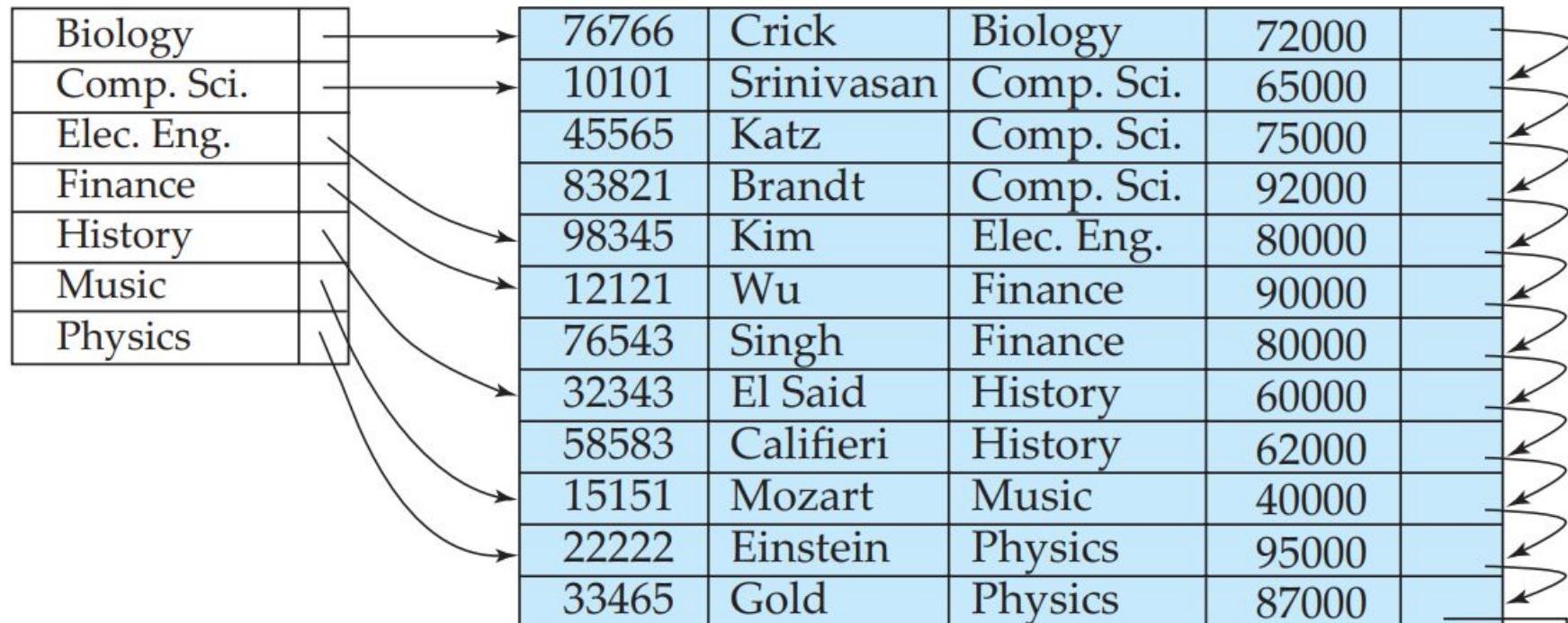
# Primary Index: Example

- Faster block accesses for Primary index
- **Binary search of Index files is faster than on Data files**



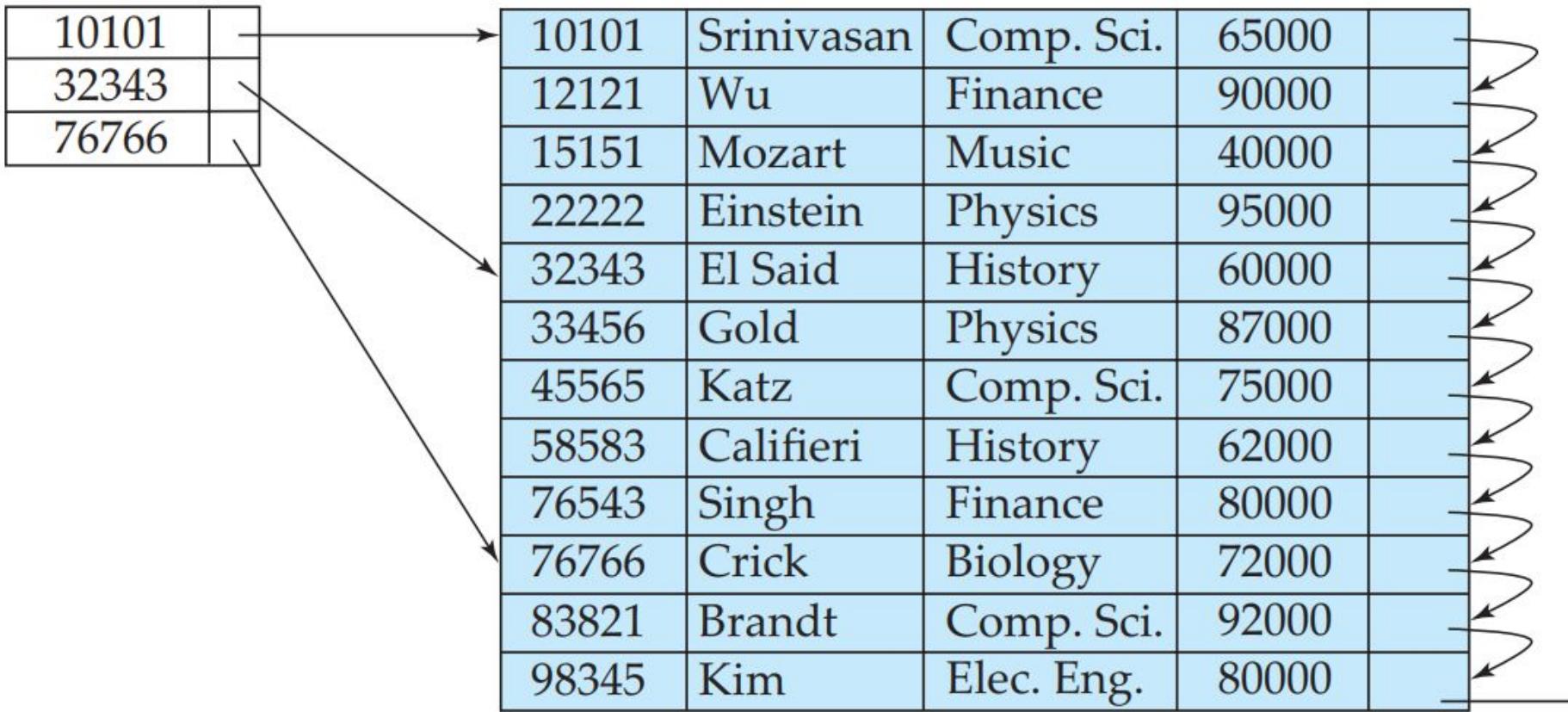
# Primary Index types

- Indexes can be of the type
  - Dense index or Sparse index.**
- A **dense index** has an entry for every **search key value** / every record in the data file. Hence **Dense index** is also called **Primary Index**.
- Example



# A sparse index

- A **sparse index** has index entries for only some of the search values. It has an entry for each disk block of the data file and not for every record.
- Example



# Saving in block accesses using a primary index to search for a record.

- Eg- An ordered file has records  $r=30,000$  and Block size =1024 bytes. Let the records be **unspanned** with record length=100 bytes.
- **bfr=floor(B/R) = floor(1024/100) = 10 records/block.** Total blocks needed for file=  $\text{ceil}(r/bfr)=\text{ceil}(30000/10)= 3000$  blocks.
- The binary search on data file needs  $\text{ceil}(\log_2 b)=\text{ceil}(\log_2 3000)=\mathbf{12 \text{ block accesses.}}$
- Now consider the ordering key  $V=9$  bytes & pointer  $P=6$  bytes. Size of each index entry =15 bytes.
- Then **bfr<sub>i</sub>= floor(B/R<sub>i</sub>) = floor(1024/15)=68 entries/block.** Index entries  $r_i$  is the total no. of blocks in datafile= 3000.
- Total no. of index blocks **bi=ceil(r<sub>i</sub>/bfr<sub>i</sub>)= ceil(3000/68) =45 blocks.**
- Therefore binary search on index file=**ceil(log<sub>2</sub> bi) =ceil(log<sub>2</sub> 45)= 6 block accesses.**

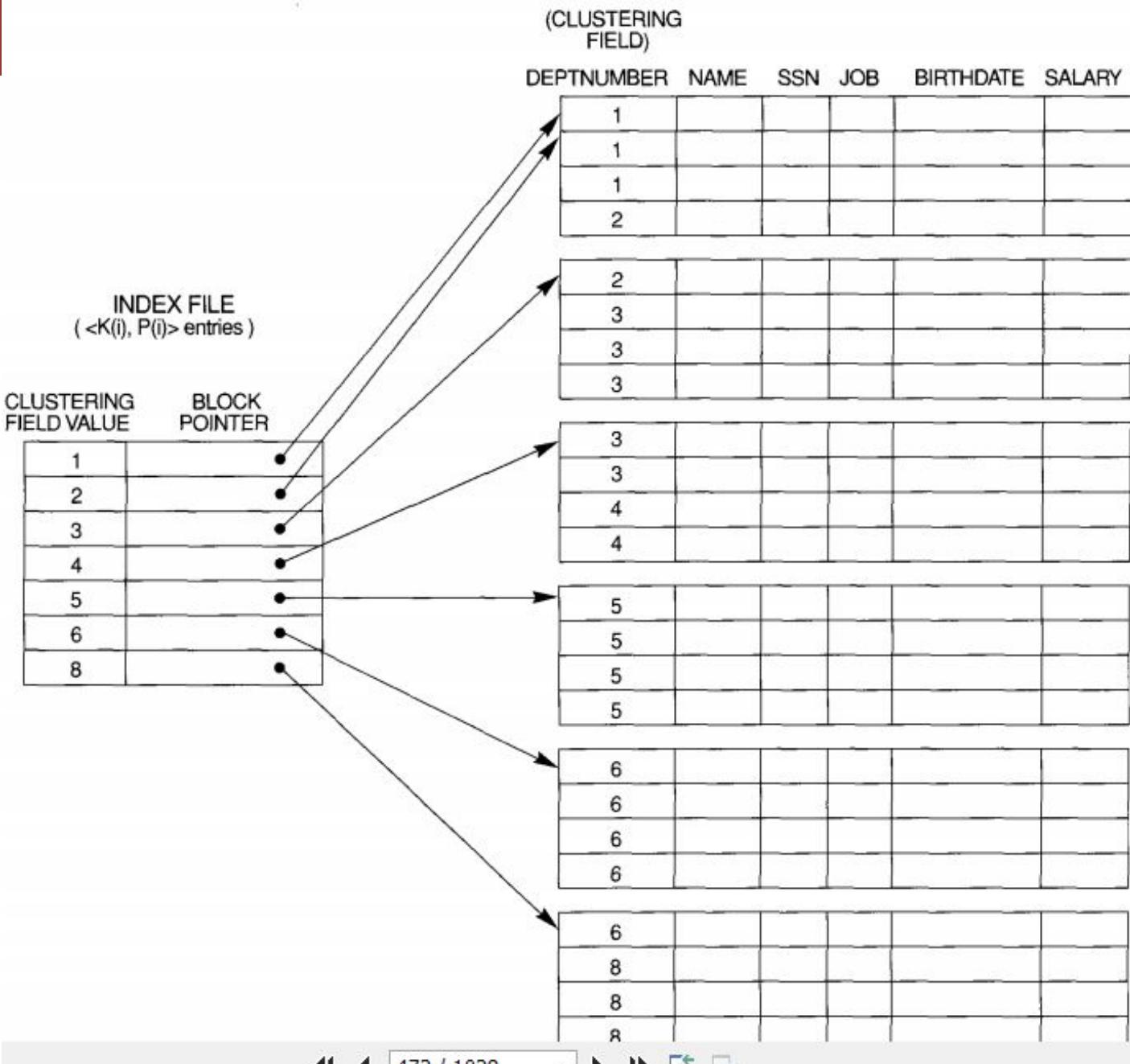
# Clustering Indexes

- When records of a file are ordered on a **non-key field**, it is called **clustering index**.
- It is one of the special types of index which reorders the way records in the table are physically stored on the disk.
- The data file is ordered by the index field- (**C(i)**, **P(i)**); **C** is the **ordering field** (it is not a key and is not unique) while **P(i)** points to the first block that has **C (i)**.
- There exist an entry in the Index file for each **distinct value** of the clustering field
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example:

- Sometimes we are asked to create an index on a non-unique key like dept-id. There could be several employees in each department.
- Here, all employees belonging to the same dept-id are considered to be within a single cluster, and the index pointers point to the cluster as a whole.

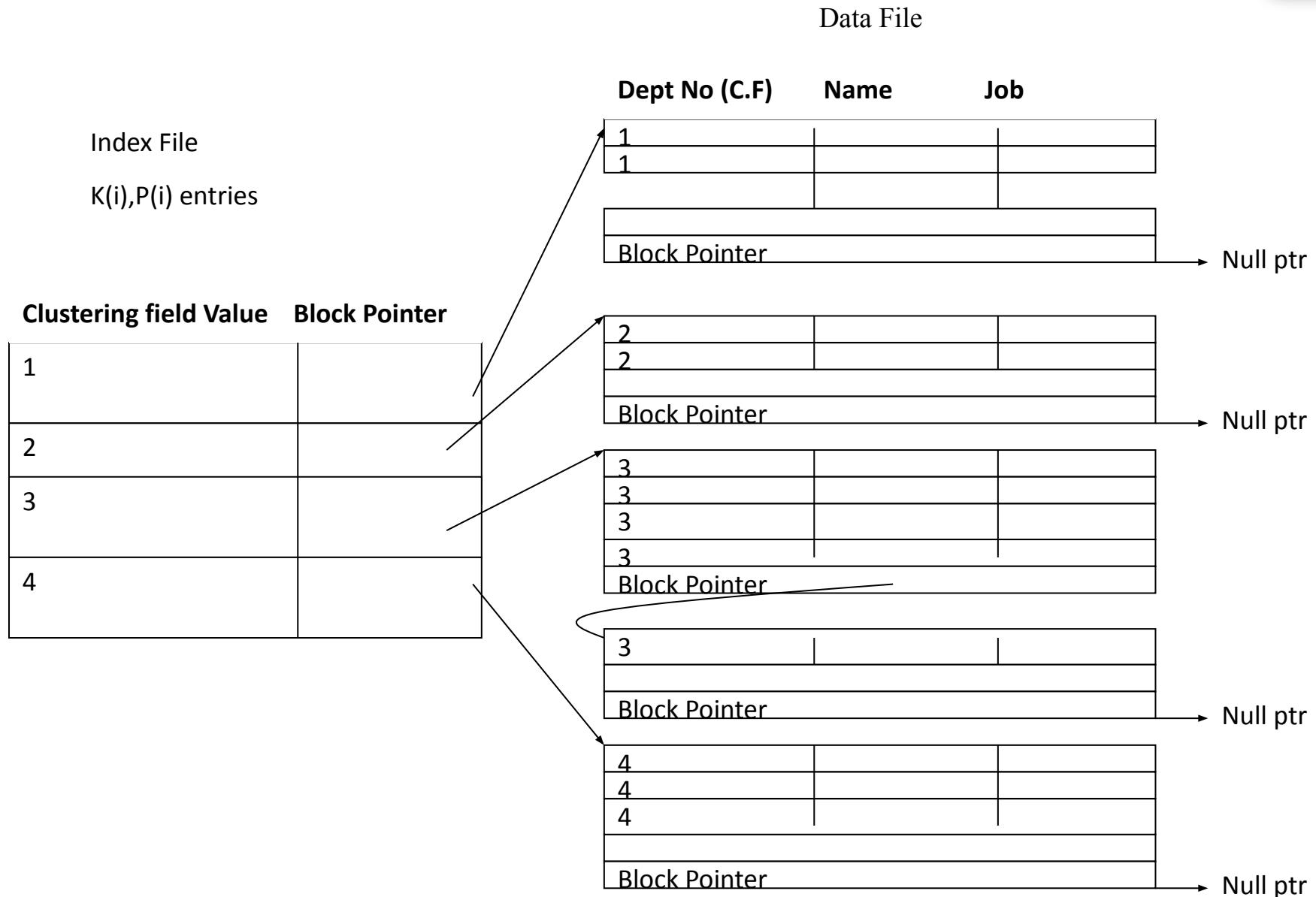
# Clustering Index



# Record Deletion/ Insertion Problem

- Record **Deletion/ Insertion** – is a problem since the records are physically ordered. To reduce this problem for insertion, it is common to reserve contiguous blocks for each value of the clustering field. All records with that value are placed in the block. This makes insertion and deletion relatively straightforward as shown in figure with separate block cluster for each group of records that share same value for the clustering field.
- Clustering index is a **nondense** index because it has an entry for every distinct value of the indexing field rather for every record in the file.

# Separate disk block for separate clusters



# Secondary index

- Recall the query
- select \* from instructor where name = ‘Einstein’
- Sorted structure on *id* does not help us here
- If we do this kind of queries often, we may need a **secondary index**
- Compare to phone books sorted by business type with an index sorted by company name.

# Secondary index

- Index over attribute(s) for which data file is not sorted
- Attribute(s) need not be candidate key
- Inner index must be **dense**, i.e. one entry for each tuple
- But is usually still much smaller than data file
- Outer indices need not be dense, because inner index sorted
- Will need 1 more level of indexing for the secondary index

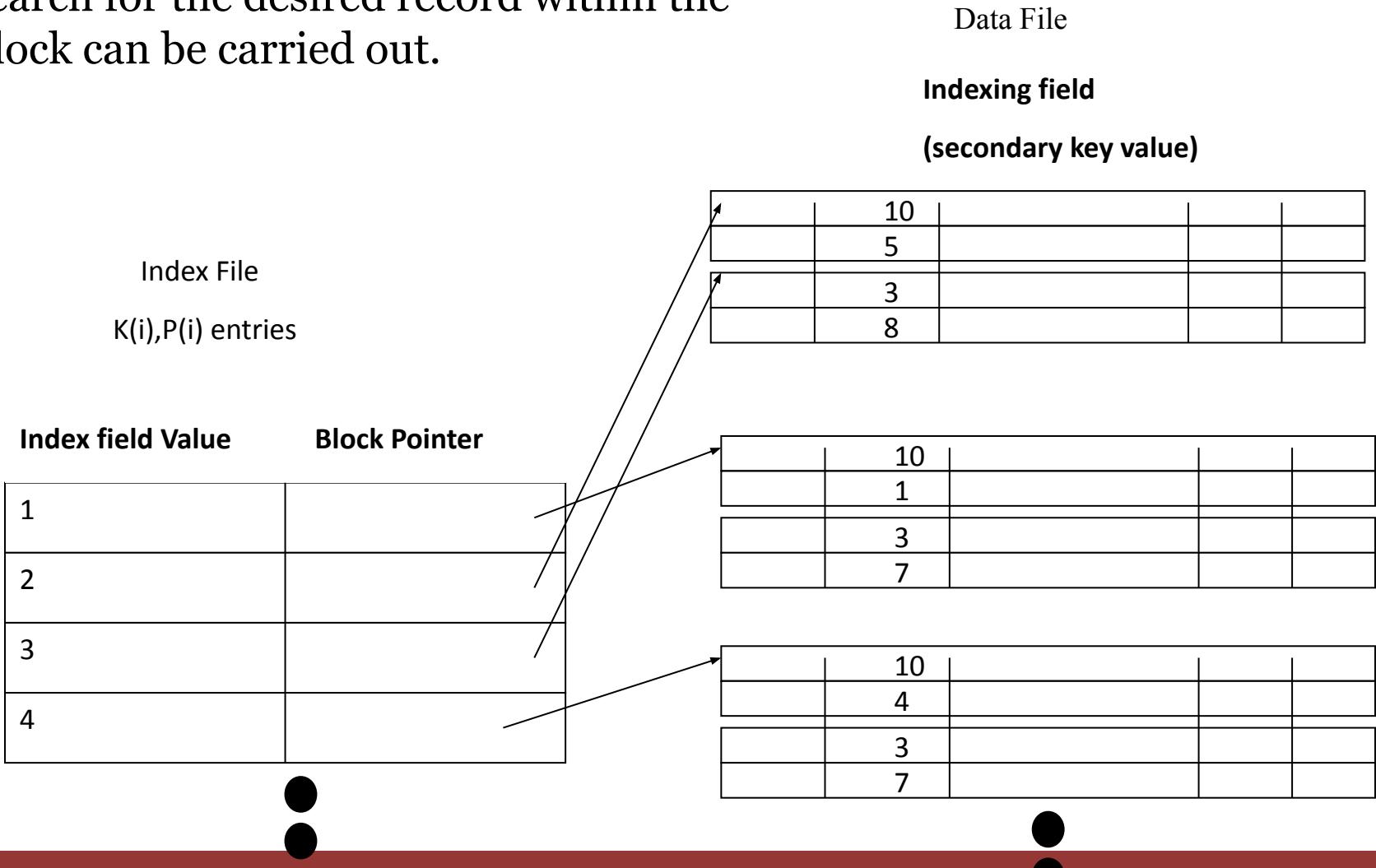
# Secondary Indexes

A secondary index is also an ordered file with **2 fields**. Here the data file CAN NOT BE ordered by the index field- (K(i), P(i)): ‘K’ can be on a **candidate key** (unique) or on a **non-key**, ‘P’ can be a **pointer to a block** or **to a record**.

- First type – on a candidate key: ‘K’ is **unique**, ‘P’ points to the block or record of every K, Since physical ordering is on the primary key we need, ‘P’ for each record entry hence **DENSE**. The Index file is ordered by K’
- Disadvantages: A secondary Index requires more space than a primary index => higher access time, On the other hand, data file search time is hugely improved.

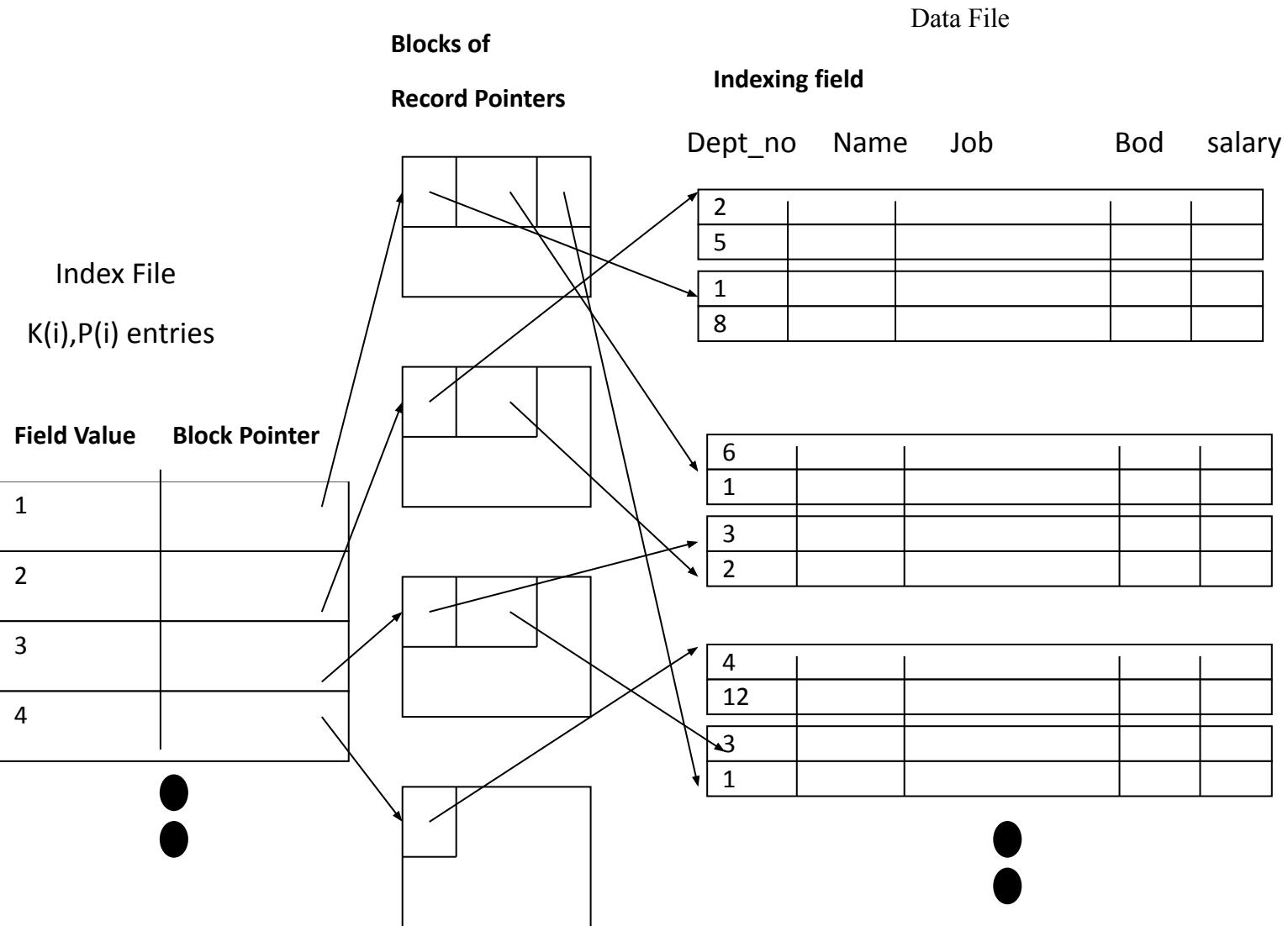
# Dense secondary index (with block pointers) on a **nonordering** key field of a file

P(i) are block pointers . Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.

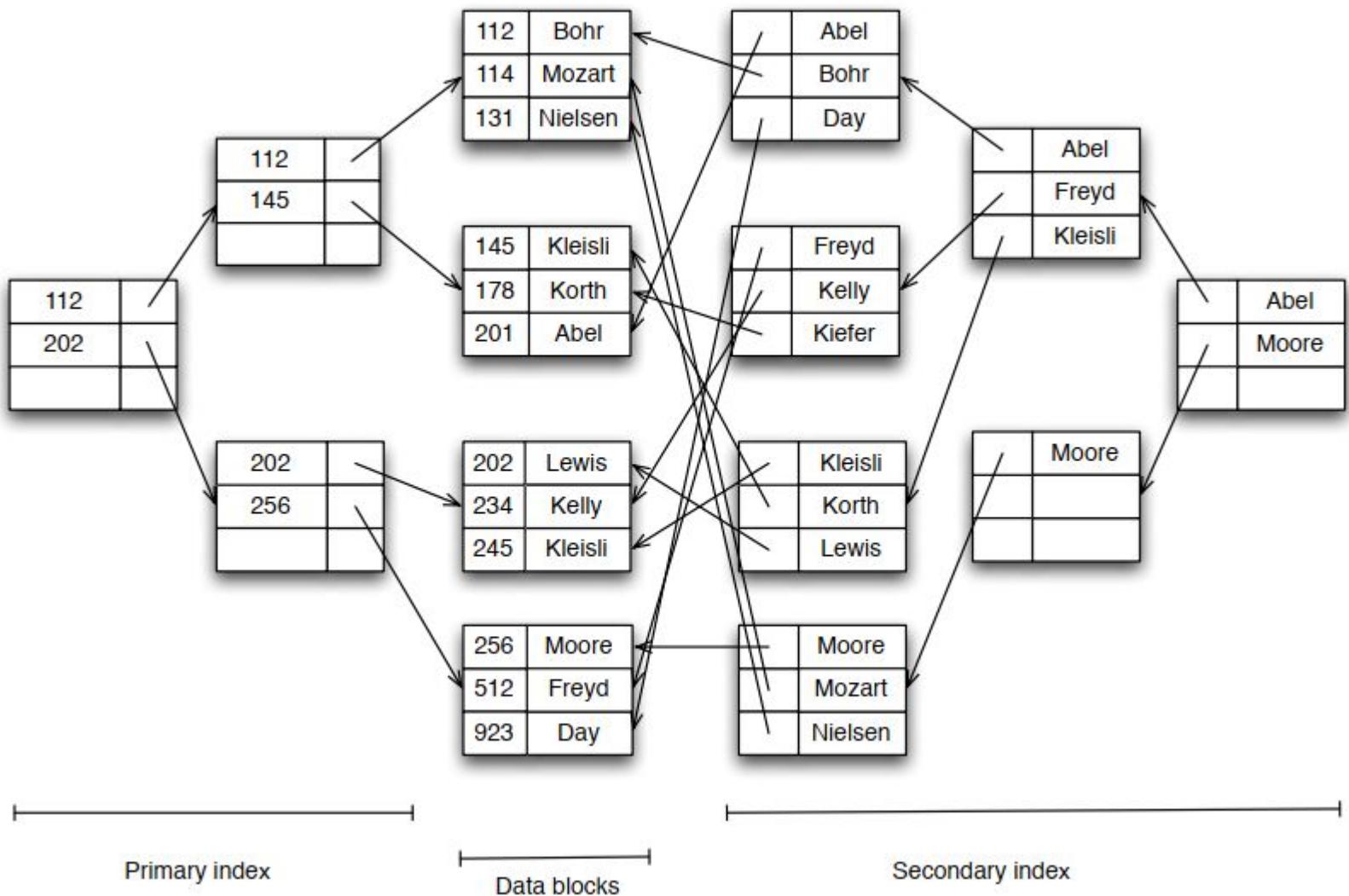


## Second Type – not on a candidate key: K is not unique

- **Option 1:** insert multiple index entries ( $K_i, P$ ) with same  $K_i$ , hence **DENSE index**,
- **Option 2:** to have variable length records for the index entries with repeating field for the pointer. We keep list of pointers  $(K(i), \{P'(i,1), P(i,2), \dots\})$ , one pointer to each block that contains a record whose indexing equals  $K(i)$ ,
- FOR option 1 and option 2 the binary search algorithm on the index must be suitably modified.
- **Option 3:** an extra level of indirection. The index entries are of fixed length and create an extra level of indirection to hold multiple pointers. Hence in this non dense scheme the pointer  $P(i)$  points to a block of record pointers, each pointing to one of the data file records with value  $K(i)$ . If some value of  $K(i)$  occurs several times, then a cluster or linked list of blocks is used as shown in the figure below.
- Retrieval requires one or more additional block accesses because of the extra level. However insertion is straightforward.



# Secondary Index



# Properties of Index Types

Primary Index	Clustering Index	Secondary Index
ordered file	ordered file	ordered file a secondary means of accessing a file
Data file is ordered on a key field (distinct value for each record)	Data file is ordered on a non-key field (no distinct value for each record)	Data file is ordered may be on candidate key has a unique value or a non-key with duplicate values
file content <key field, pointer>	file content <key field, pointer>	file content <key field, pointer>
one index entry for each disk block. key field value is the first record in the block, which is called the block anchor	one index entry for each distinct value of the field; the index entry points to the first data block that contains records with that field value	The index is an ordered file with two fields: 1- field value. 2- it is either a block pointer or a record pointer.
nondense (sparse) index	nondense (sparse) index	If key, dense. If non key, dense or sparse index

# Multilevel indexing

- In the sparse indexing, as the size of the table grows, the size of mapping also grows.
- If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient.
- To overcome this problem, to reduce the size of mapping, another level of indexing is introduced.
- In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small.
- Then each range is further divided into smaller ranges.
- The mapping of the first level is stored in the primary memory, so that address fetch is faster.
- The mapping of the second level and actual data are stored in the secondary memory (hard disk).

# Multi Level Index Types

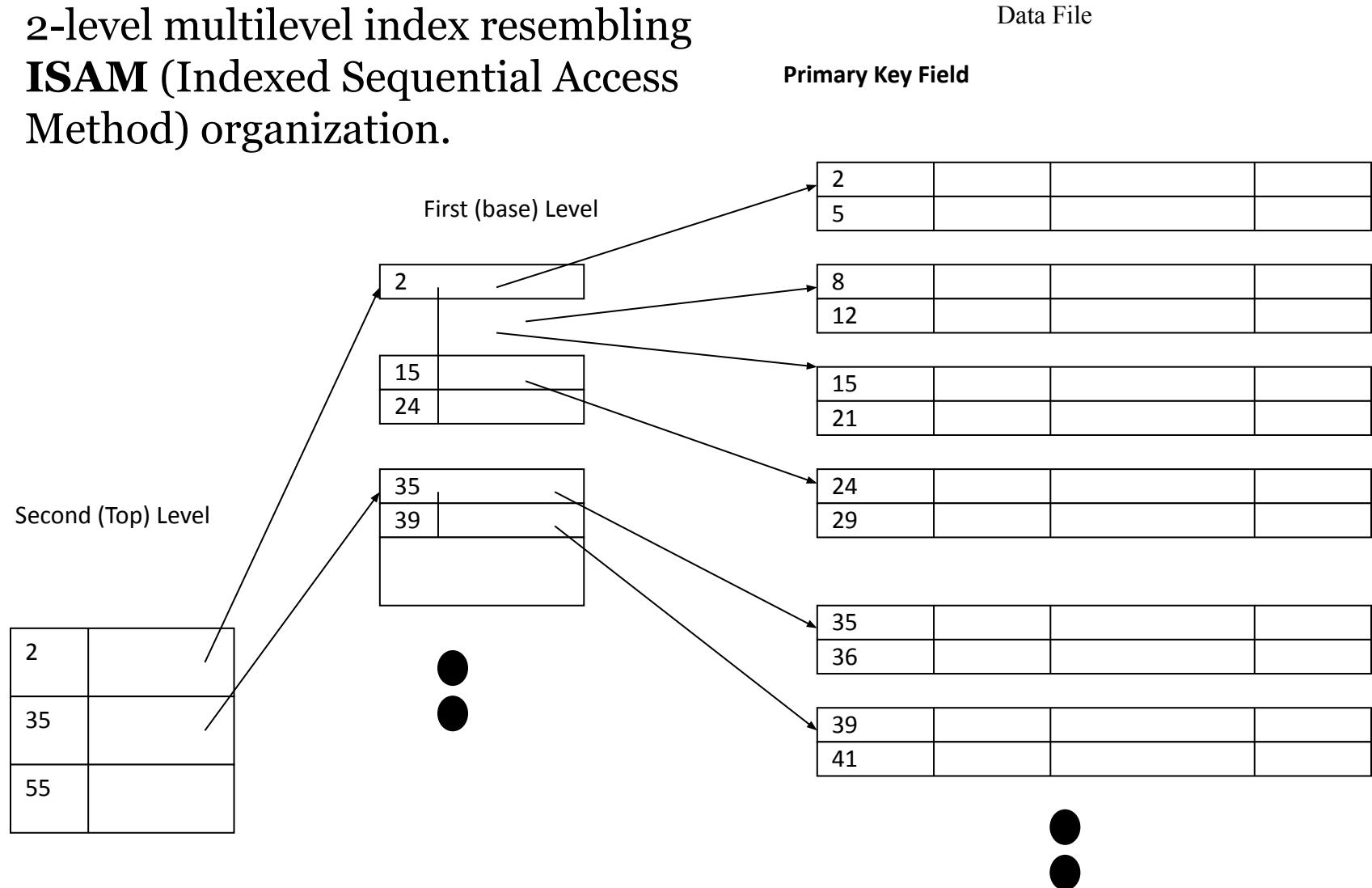
- Two types
  1. A two level primary index
  2. Dynamic multi level indexes using BTree and B+Tree
- The original index file is called first-level index and index to the index is called the second-level index.
- We can repeat the process, creating third, fourth, ..., top level until all entries of the top level will fit in one disk block.
- A multi level index can be created on any type of first-level index (primary, secondary, clustering).
- Number of key, block address combinations storable in a block is called the **fan out (blocking factor)**.
- We need (number of data blocks)/fan out blocks for the index file.

# Multilevel Indexes

- Purpose: to decrease the index file search time by decreasing the search space by more than a factor of 2 (fan-out  $\square$  fo).
- The value **bfr<sub>i</sub>** is called the fan-out of multilevel index. A binary search is  $\log_2 \text{bi}$  while search using a multilevel index requires approximately  $\log_{\text{fo}} \text{bi}$  block accesses which is smaller than the binary search if the fan-out is larger than 2.
- The first level needs to have an ordered file with distinct value for each K(i).
- The second level is an index to the first level and can be further repeated. Each level reduces the number of entries at the previous level by a factor of fo. The top index level  $T = \text{ceil}(\log_{\text{fo}}(r_1))$  where r<sub>1</sub> is the number of first level entries.

# 2-level Primary Index

- 2-level multilevel index resembling **ISAM** (Indexed Sequential Access Method) organization.



# Multilevel Index Block access calculation

- Let dense secondary index example given earlier be converted to a multilevel index.
- $B_{fr_i} = 68$  index entries per block. Which is also the fan-out (fo) for the multilevel index.,
- The number of 1<sup>st</sup> level blocks  $b_1 = 442$  blocks.
- 2<sup>nd</sup> level blocks  $b_2 = \text{ceil}(b_1/\text{fo}) = \text{ceil}(442/68) = 7$  blocks
- Third level blocks will be  $b_3 = \text{ceil}(b_2/\text{fo}) = \text{ceil}(7/68) = 1$  block.
- Hence 3<sup>rd</sup> level is the top level of the index and  $t=3$ .
- To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file.
- So we need  $t+1 = 3+1 = 4$  block accesses. (For single level it was calculated earlier as 10 block accesses when binary search was used).

## Example 2

- Suppose there are 10,000 blocks in the data file
  - Suppose fan out is 100
- We then need  $10,000 / 100 = 100$  blocks for the inner index
- We need  $100/100=1$  block for the outer index
- A search using the index requires 3 block reads
- A linear search requires 10,000 block reads
- If there are 1,000,000 blocks of data we need just 1 more index level

# Disadvantage of the index-sequential file

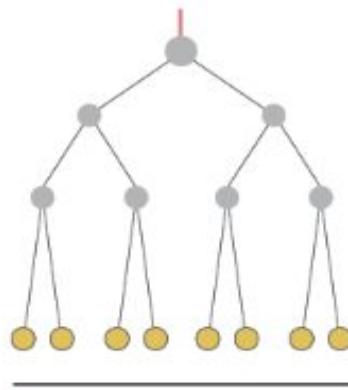
- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.
- Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.
- As data sizes increase, access in such structures becomes slower, which can be hugely problematic in today's hyper-digital world. Tree structures can avoid such issues.
- Tree is a non-linear data structure that provides easier and quicker access to data. While linear data structures store data sequentially, tree structures permit data access in different directions.

# Tree structure properties

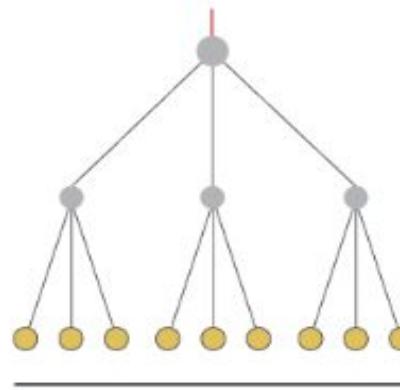
- A tree structure consists of nodes connected by edges.
- A tree can contain one special node called the "root" with zero or many subtrees. It may also contain no nodes at all.
- Every edge directly or indirectly originates from the root.
- The tree is always drawn upside-down, which means the root appears at the top.
- One parent node can have many children, but every child node has only one parent node. The maximum number of children per node is called the tree "order."
- In a tree, records are stored in locations called "leaves." The name indicates that records always exist at endpoints; there is nothing beyond them. Not every leaf contains a record, but more than half do. A leaf that does not contain data is called a "null."
- The maximum number of access operations required to reach the desired record is called the "depth."
- The only way to perform any operation on a tree is by reaching the specific node. For this, a tree traversal algorithm is required.

# Balanced and unbalanced trees

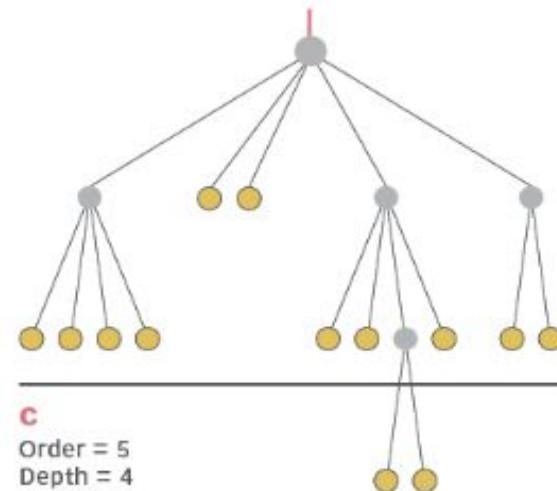
- If the order is the same at every node and the depth is the same for every record, the tree is said to be *balanced*.
- Trees that are have varying numbers of children per node, and different records might lie at different depths, such tree are said to have an *unbalanced* or *asymmetrical* structure.



**A**  
Order = 2  
Depth = 4  
Leaves = 8



**B**  
Order = 3  
Depth = 3  
Leaves = 9



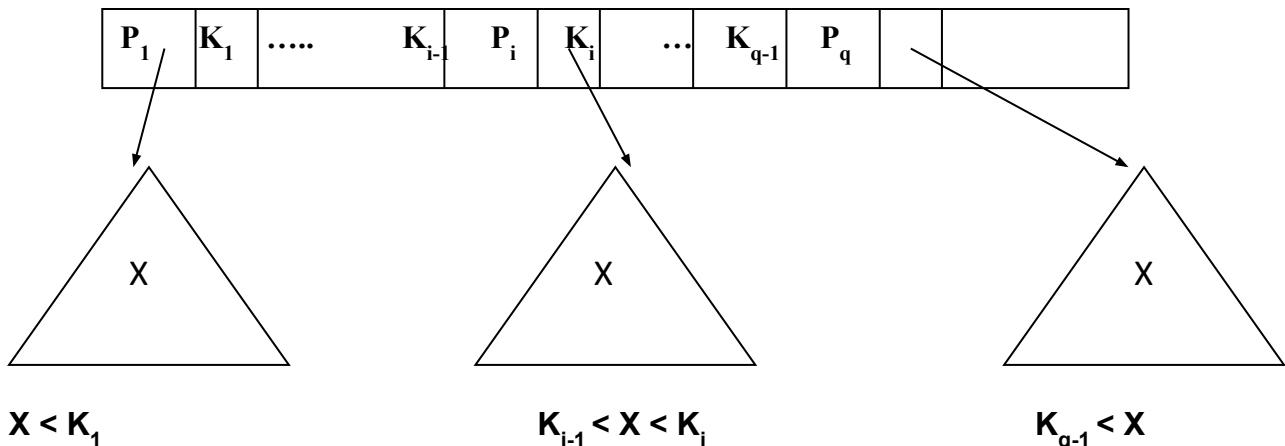
**C**  
Order = 5  
Depth = 4  
Leaves = 13

# Different types of tree structures

- A general tree is a data structure with no constraints on the hierarchical structure. This means that a node can have any number of children. This tree is used to store hierarchical data, such as folder structures.
- Other common tree structures are:
- **Binary tree:** In a binary tree structure, a node can have at most two child nodes, known as the left child and right child.
- **Binary search tree:** The node's left child must always have a value less than or equal to the parent's value. Further, the right child must always have value more than or equal to the parent's value.
- **B-tree:** A self-balancing search tree, in which nodes can contain more than one key and two or more children.
- **T-tree:** A self-balancing tree structure optimized for databases that keeps both data and index in memory.

# Search tree & B Trees

- A search tree is used to guide the search for a record based on the record fields value.
- A search tree of order  $p$  is a tree and each node contains *at most*  $p-1$  search values and  $p$  pointers and the order is  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$ . where  $q \leq p$  where  $P_i$  is a pointer to a child or a null pointer and each  $K_i$  is a search value from some ordered set of values. All search values are assumed to be unique.
- Two constraints must hold at all times on the search tree:
  1. Within each node,  $K_1 < K_2 < \dots < K_{q-1}$ .
  2. For all values  $X$  in the subtree pointed at by  $P_i'$  we have  $K_{i-1} < X < K_i$ , for  $1 < i < q$ ;  $X < K_1$ , for  $i = 1$ ; and  $K_{q-1} < X$  for  $i = q$



# Issues with search tree

- Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is balanced, meaning that all of its leaf nodes are at the same level.
- Keeping a search tree balanced is important because it guarantees that no nodes will beat very high levels and hence require many block accesses during a tree search. Keeping the tree balanced yields a uniform search speed regardless of the value of the search key.
- Another problem with search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels.
- The B-tree (Balanced stored tree) addresses both of these problems by specifying additional constraints on search tree.

# Properties of B-tree

- B tree are called balanced stored trees, since all the leaves are at the same level, it is also called a multi-way search tree, it is a form of multilevel indexing.
- B tree grow towards root not towards leaf.

**Definition** – A B-tree of order m is an m-way tree that means a node can have maxm m children.

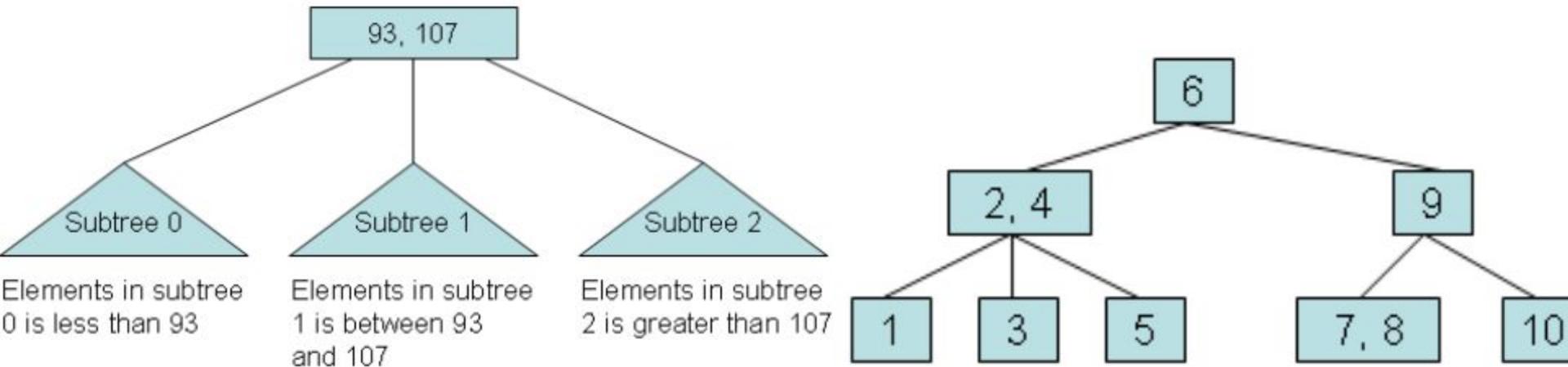
## Properties of B-tree

B-tree satisfies the following properties –

- The number of keys in a node (non leaf node) is one less than the number of its children (and these keys partition the keys of children like a binary search tree).
- The root has a minimum one key to maximum ( $m-1$ ) keys. So, root has minimum two children to maximum m children.
- Any node (non-leaf node) except the root has  $\min^m([m/2]-1)$  to  $\max^m(m-1)$  keys. So, they have  $\min^m [ m/2 ]$  children to  $\max^m m$  children.
- All leaf nodes are on the same level

# B Tree (Balanced stored tree)

- A B-tree is a tree keeps data sorted and allows searches, insertions, and deletions in logarithmic time.
- Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.
- Important properties of a B-tree:
  1. B-tree nodes have many more than two children.
  2. B-tree node may contain more than a single element.

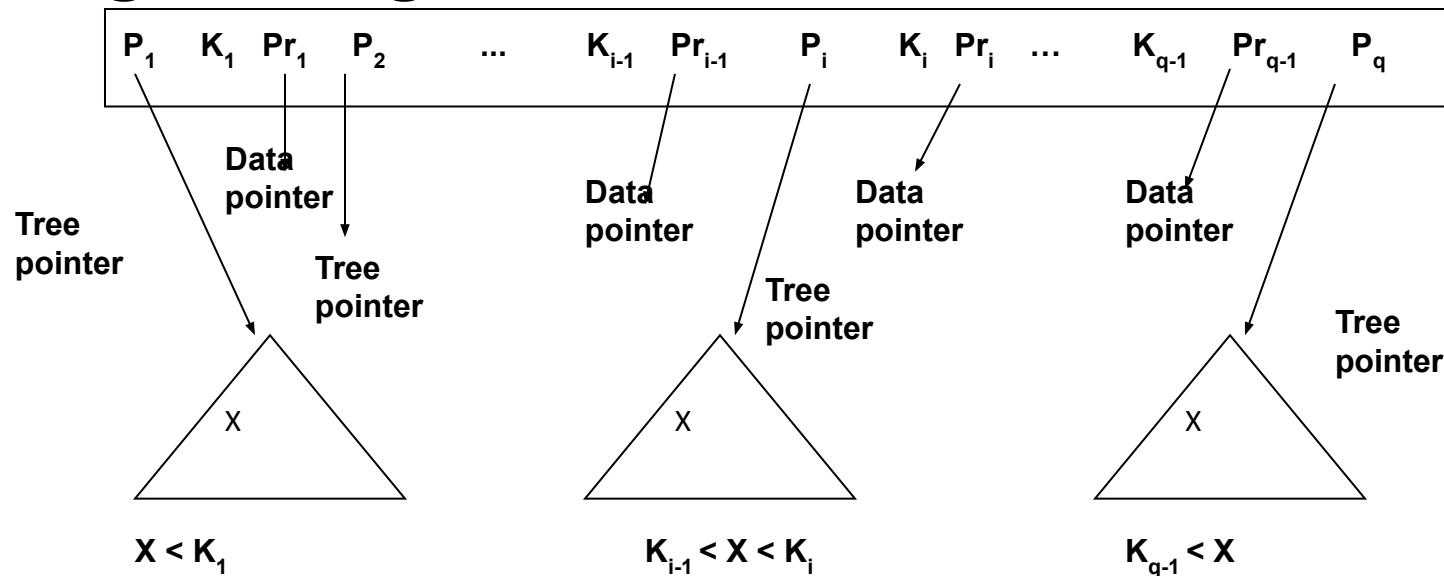


# Reasons for using B-tree

- The reasons for using B-tree are as follows –
- When searching tables on disc, the cost of accessing the disk is high but it doesn't bother about the amount of data transferred. So our aim is to minimize disc access.
- We know that we cannot improve the height of trees. So, we wish to make the height of the tree as small as possible.
- The solution for this is to use a B-tree, it has more branches and thus less height. As branching increases, depth decreases so does access time.
- In a B-tree, one node can hold many elements or items.

# B-Tree

- **B-Tree** adds the following additional constraints
  - Ensures that the tree is balanced (all of the tree nodes are at the same level) – to reduce the number of block accesses
- Solves the problem during record deletion, which may leave some nodes in the tree empty leading to storage wastage

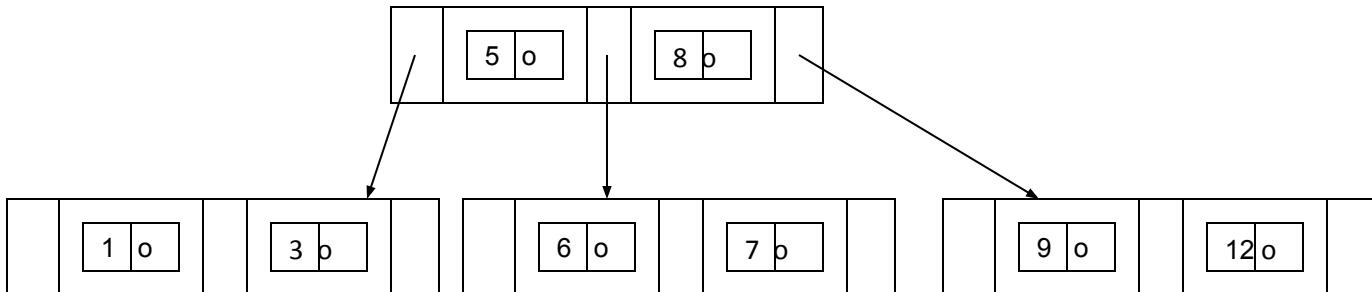


B-Tree of order  $p$  is defined as

- Each internal node is of the form  $\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$  where  $q \leq p$ . Each  $P_i$  is a tree pointer to another node. Each  $Pr_i$  is a data pointer to a record whose field value =  $K_i$ .
- $K_1 < K_2 < \dots < K_{q-1}$
- Search key value  $X$  in a subtree pointed by  $P_i$  has  $K_{i-1} < X < K_i$  for  $1 < i < q$ ;  $X < K_i$  for  $i=1$  and  $K_{i-1} < X$  for  $i=q$
- Each node has at most  $p$  tree pointers.
- Each node except the root and leaf nodes has  $\text{ceil}(p/2)$  tree pointers. The root node has at least 2 tree pointers unless it is the only node in the tree
- A node with  $q$  tree pointers,  $q \leq p$ , has  $q-1$  search key field values (hence  $q-1$  data pointers).
- All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all their tree pointers are null.

# A B-Tree of order 3

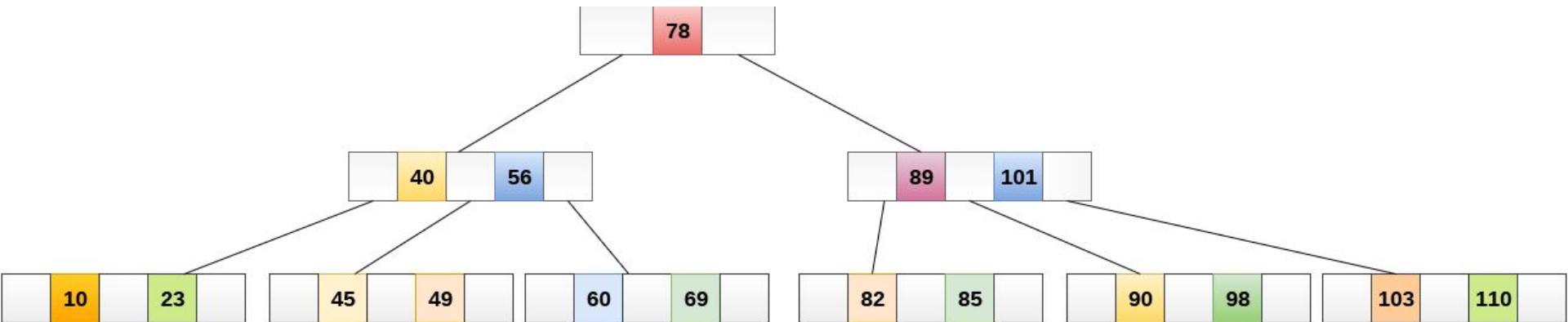
- The search values K are unique. However if the access is on a non-key then the file pointers will point to a block/cluster of blocks that point to the file record. Values were inserted in the order **8, 5, 1, 7, 3, 12, 9, 6.** (**note: use rules to show insertion and deletion**)



- B-Tree starts at root node. Once it is full with  $p-1$  search keys, we insert another entry in the tree and the root node splits into two nodes at level 1 .
- Only the middle value is kept in the root node and rest of the values are split evenly between the other 2 nodes.
- If deletion of a value causes a node to be less than half full, it is combined with its neighbouring nodes and this can propagate all the way to the root. Hence deletion can reduce number of tree nodes.

# Searching :

- Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :
- Compare item 49 with root node 78. since  $49 < 78$  hence, move to its left sub-tree.
- Since,  $40 < 49 < 56$ , traverse right sub-tree of 40.
- $49 > 45$ , move to right. Compare 49.
- match found, return.

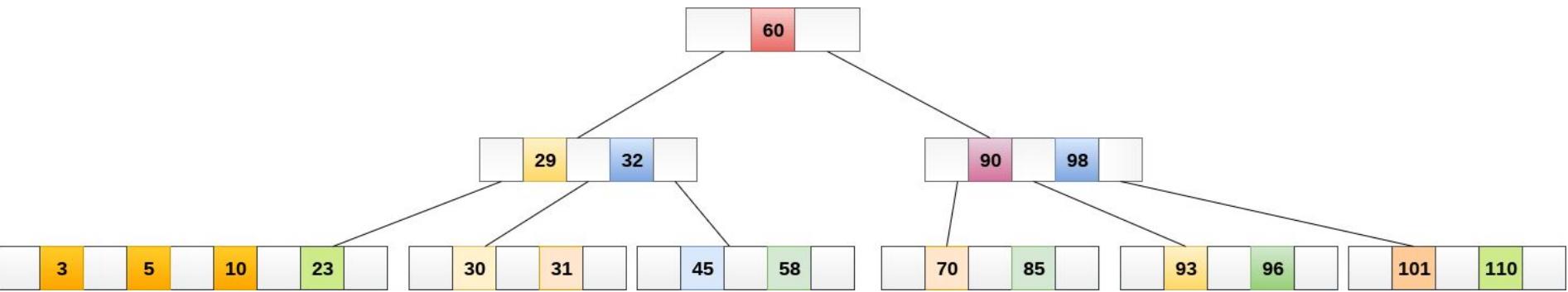


# Inserting

- Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.
- Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
- If the leaf node contain less than  $m-1$  keys then insert the element in the increasing order.
- Else, if the leaf node contains  $m-1$  keys, then follow the following steps.
  - Insert the new element in the increasing order of elements.
  - Split the node into the two nodes at the median.
  - Push the median element upto its parent node.
  - If the parent node also contain  $m-1$  number of keys, then split it too by following the same steps.

# Inserting Example

- Insert the node 8 into the B Tree of order 5 shown in the following image.
- 8 will be inserted to the right of 5.



## Example-1

- Calculate the order of B-Tree  $p$ . For Search field  $V=9$  bytes,  $B=512$  bytes, and data pointer  $Pr=7$  bytes and block pointer  $P=6$  bytes. There are  $p$  tree pointers,  $p-1$  data pointers and  $p-1$  search key values. Hence the following condition holds
- $(p * P) + ((p-1)*(Pr+V)) \leq B$
- $(p*6) +((p-1)*(7+9)) \leq 512$
- $p^2 \leq 528$ . Taking equality condition  $p=23$ .

## Example-2

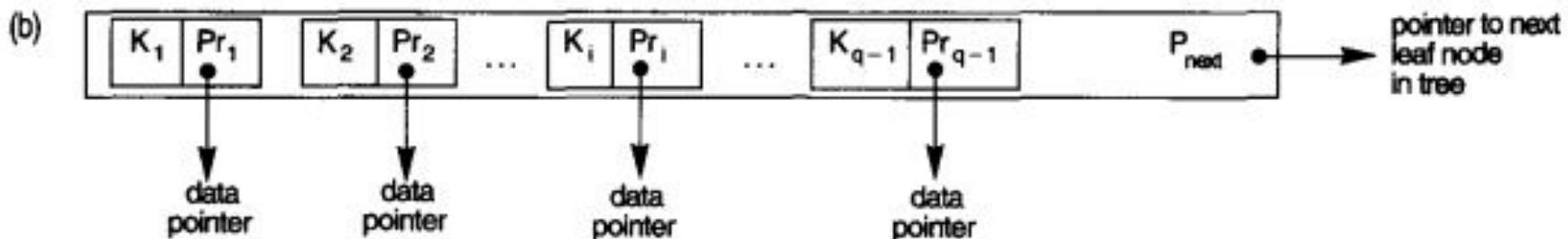
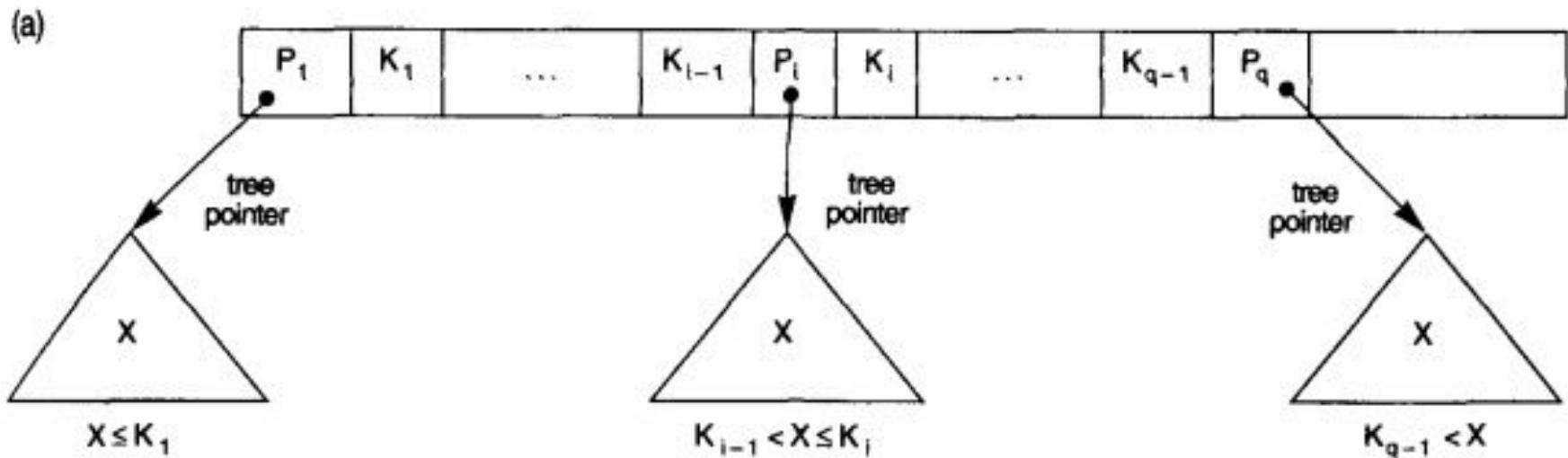
- Calculating the number of blocks and levels for B-Tree.  
Above example is a non ordering key field. Assume that 69% of each node is full. Therefore each node has  $p * 0.69 = 23 * 0.69 = 16$  pointers approximately and hence 15 key field values. Average fan-out (fo)= 16.
- Hence
- Root 1 node 15 entries 16 pointers
- Level 1 16 nodes 240 entries 256 pointers
- Level 2 256 nodes 3840 entries 4096 pointers
- Level 3 4096 nodes 61,440 entries
- This structure works well with files with relatively **small number of records** and a **small size records** otherwise the fan-out and number of levels become too big to permit efficient access.

# B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.
- Each nonleaf node in the tree has between  $n/2$  and  $n$  children, where  $n$  is fixed for a particular tree.

# Nodes of a B+ tree

- (a) Internal node of a B+tree with  $q-1$  search values.
- (b) Leaf node of a B+tree with  $q-1$  search values and  $q-1$  data pointers.



# Structure of *internal nodes* of a B+tree of order p

- Each internal node is of the form  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$  where  $q \leq p$  and each  $P_i$  is a tree pointer
- Within each internal node  $K_1 < K_2 < \dots < K_{q-1}$
- For all search field values at by  $P_i$  we have  $K_{i-1} < X \leq K_i$  **for  $1 < i < q$** ;  $X \leq K_i$  **for  $i = 1$**  and  $K_{i-1} < X$  **for  $i = q$**
- Each internal node has at most p tree pointers.
- Each node except the root and leaf nodes has  **$\text{ceil}(p/2)$**  tree pointers. The root node has at least 2 tree pointers if it is an internal node.
- An internal node with q pointers ,  $q \leq p$  has  $q-1$  search field values.

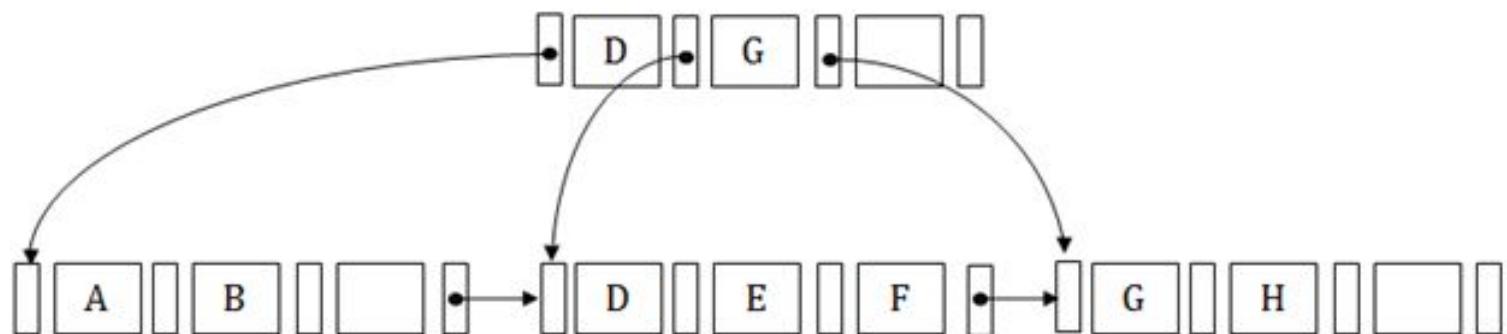
# The structure of the leaf nodes

- Each leaf node is of the form  $\langle\langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{next} \rangle$  where  $q \leq p$ , each  $Pr_i$  is a data pointer and  $P_{next}$  points to the next leaf node of the tree.
- Within each leaf node  $K_1 < K_2 < \dots < K_{q-1}$
- Each  $Pr_i$  is a data pointer to a record whose search value is  $K_i$  OR to a file block containing the record.
- Each leaf node has at least  $\text{ceil}(p/2)$  values.
- All leaf nodes are at the same level.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks-except for the  $P_{next}$  pointer, which is a tree pointer to the next leaf node.

# Structure of B+ Tree

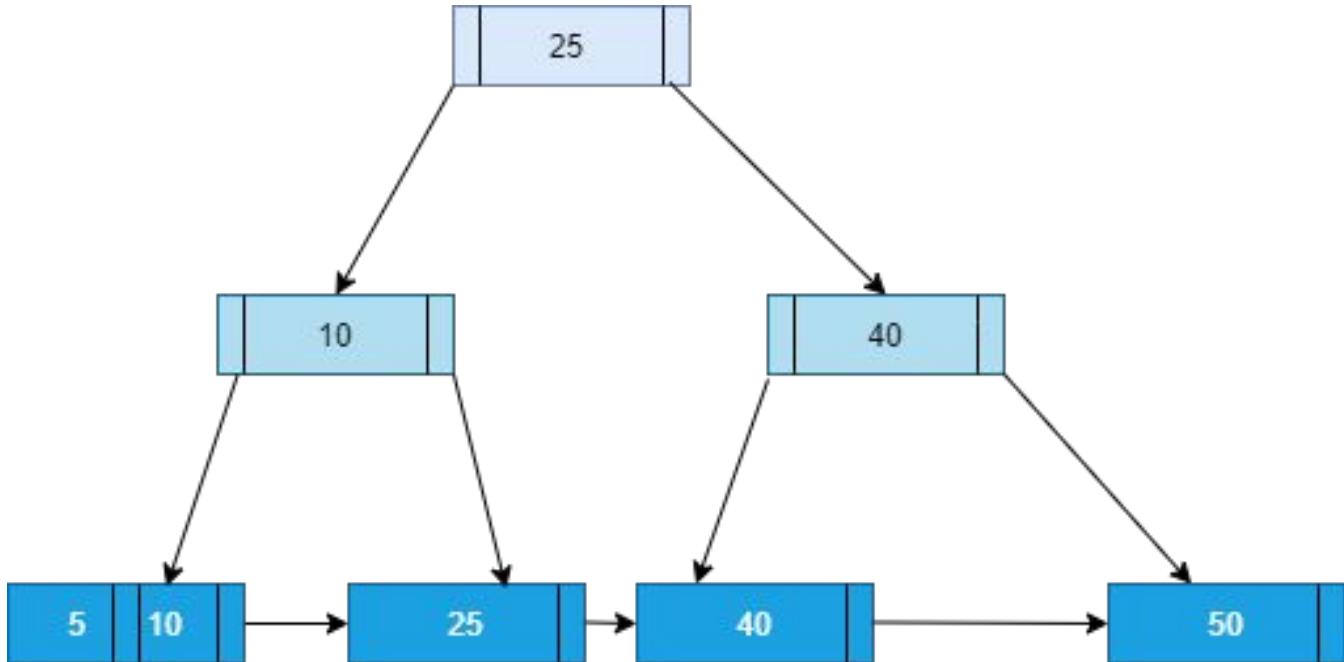
- In B+ tree, every leaf node is at equal distance from root node.
- A B+ tree is similar to a B-tree, the only difference is that their leaves are linked at the bottom.
- Unlike B-tree, nodes of B+ tree do not store keys along with pointers to disk block. The internal nodes contain only keys and the leaf nodes contain the keys along with the data pointers. All the internal nodes and nodes present at the leaves are linked together and can be traversed just like a linked list.



# Advantages of B+ tree

- B+ tree can store more keys than the B-tree of the same level because of their feature of storing pointers only at the leaf nodes. This contributes to storing more entries at fewer levels in the B+ tree and lessens the searching time for a key. This makes the B+ tree very efficient and very quick in accessing the data from the disks.

# Example of B+ tree



- Eg- To calculate the order of a B+ tree . search key field V= 9 bytes, B= 512 bytes, record pointer Pr=9 bytes and block pointer P= 6 bytes. An internal node can have p tree pointers and p-1 search field values.
- $(p^*P) + ((p-1)^*V) \leq B$
- $(p^*6) + ((p-1)^*9) \leq 512$
- $(p^*15) \leq 521$
- Satisfying the equality operator we have p=34 which is larger than the B-Tree of 23. But the leaf nodes have the same number of values and pointers except that the pointers are data pointers and next pointer. So order of leaf nodes is
- $(P_{leaf}^* (P_r + V)) + P \leq B$
- $(P_{leaf}^* (7+9)) + 6 \leq 512$
- $(P_{leaf}^* 16) \leq 506$ . or  $P_{leaf} = 31$  key value/data pointers assuming data pointers are record pointers.

# Differences between B-Tree and B+ Tree

## B-Tree

All internal nodes and leaf nodes contain data pointers along with keys.

There are no duplicate keys.

Leaf nodes are not linked to each other.

Sequential access of nodes is not possible.

Searching for a key is slower.

For a particular number of entries, the height of the B-tree is larger.

## B+ Tree

Only leaf nodes contain data pointers along with keys, internal nodes contain keys only.

Duplicate keys are present in this, all internal nodes are also present at leaves.

Leaf nodes are linked to each other.

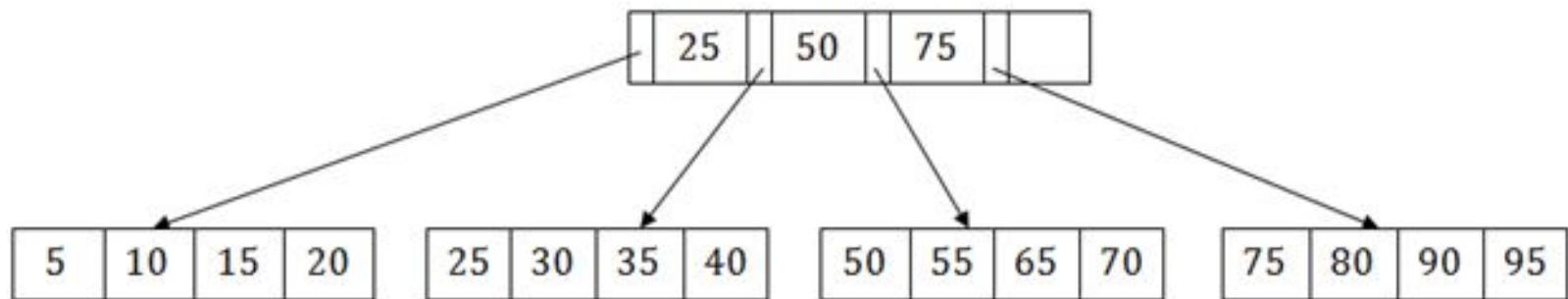
All nodes are present at leaves, so sequential access is possible just like a linked list.

Searching is faster.

The height of the B+ tree is lesser than B-tree for the same number of entries.

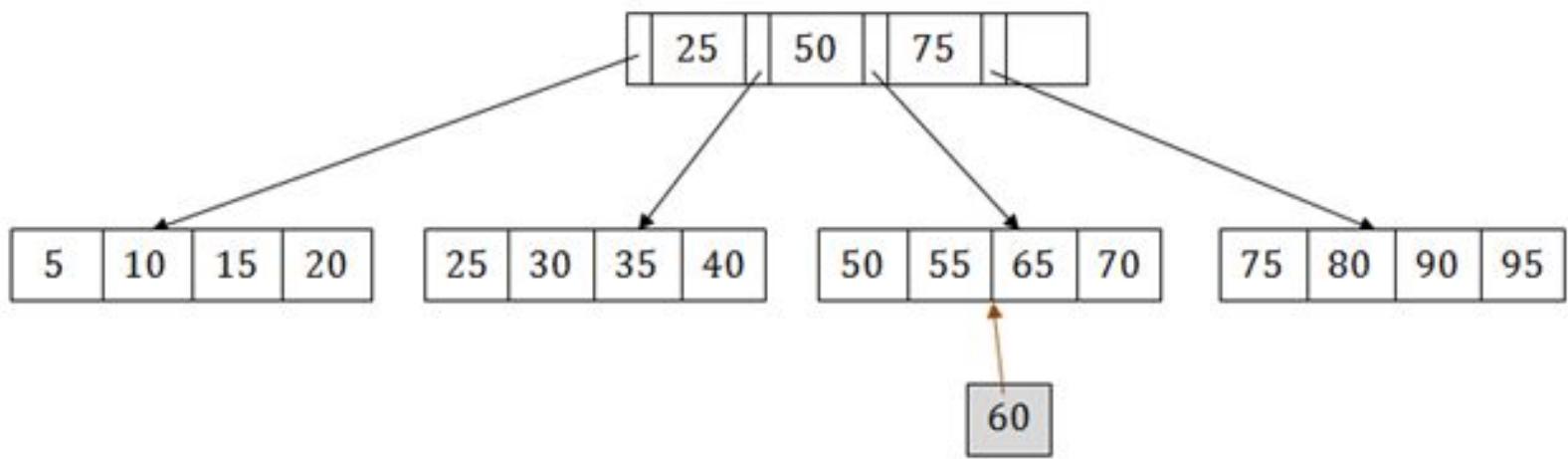
# Searching a record in B+ Tree

- suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.
- So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



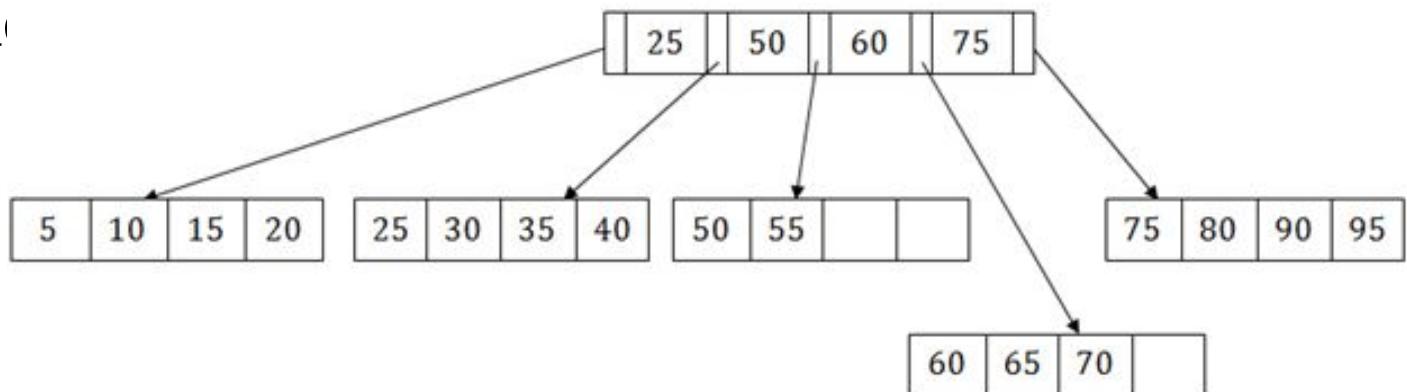
# B+ Tree Insertion

- Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.
- In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order



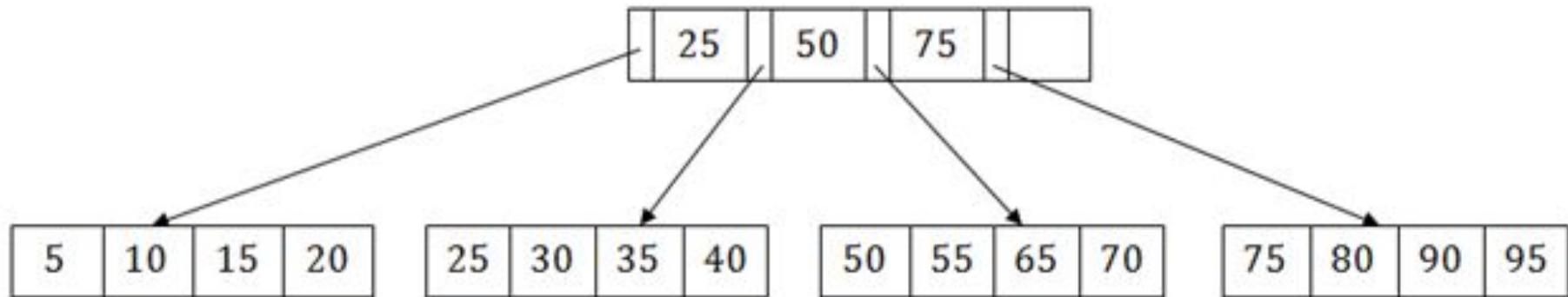
# B+ Tree Insertion contd....

- The 3<sup>rd</sup> leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.
- If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.



# B+ Tree Deletion

- Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.
- After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



# References

Thank  
you



**100008/IT400D**

**DATABASE MANAGEMENT**

**SYSTEMS**

**Module V**

**TRANSACTION PROCESSING**

**CONCEPTS**

**Addison-Wesley**  
is an imprint of

**PEARSON**

Copyright © 2011 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

# Syllabus Module 5

## Module 5: TRANSACTION PROCESSING CONCEPTS

**Transaction Processing Concepts:** overview of concurrency control and recovery acid properties, serial and concurrent schedules, conflict serializability. Two-phase locking, failure classification, storage structure, stable storage, log based recovery, deferred database modification, check-pointing, (Reading Elmasri and Navathe, Ch. 20.1-20.5 (except 20.5.4-20.5.5) , Silbershatz, Korth Ch. 15.1 (except 15.1.4-15.1.5), Ch. 16.1 – 16.5)

**Recent topics** (preliminary ideas only): Semantic Web and RDF(Reading: Powers Ch.1, 2),

GIS, biological databases (Reading: Elmasri and Navathe Ch. 23.3- 23.4)

**Big Data** (Reading: Plunkett and Macdonald, Ch. 1, 2)

# Introduction to Transaction Processing

- **Transaction:** An executing program (process) that includes one or more database access operations
  - Read operations (database retrieval, such as SQL SELECT)
  - Write operations (modify database, such as SQL INSERT, UPDATE, DELETE)
  - Transaction: A logical unit of database processing
  - Example: Bank balance transfer of \$100 dollars from a checking account to a saving account in a BANK database

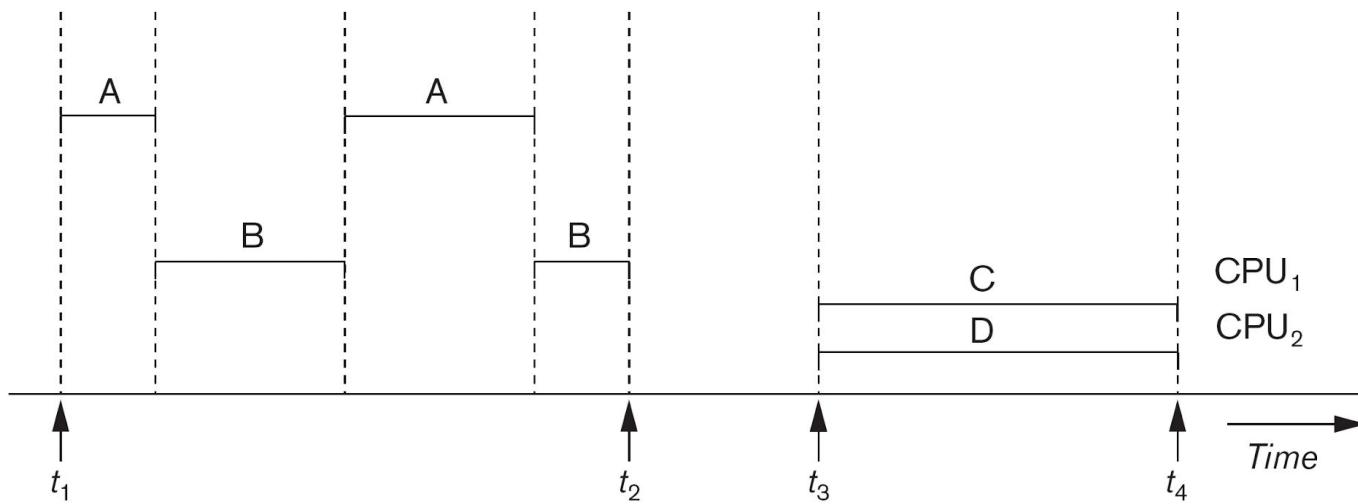
# Introduction to Transaction Processing (cont.)

- A transaction (set of operations) may be:
  - stand-alone, specified in a high level language like SQL submitted interactively, or
  - consist of database operations embedded within a program (most transactions)
- **Transaction boundaries:** Begin and End transaction.

# Introduction to Transaction Processing (cont.)

- **Transaction Processing Systems:** Large multi-user database systems supporting thousands of *concurrent transactions* (user processes) per minute
- **Two Modes of Concurrency**
  - **Interleaved processing:** concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing:** processes are concurrently executed in multiple CPUs
  - Basic transaction processing theory assumes interleaved concurrency

# Interleaved processing versus parallel processing of concurrent transactions.



**Figure 21.1**  
Interleaved processing versus parallel processing of concurrent transactions.

# Introduction to Transaction Processing (cont.)

For transaction processing purposes, a simple database model is used:

- **A database** - collection of named data items
- **Granularity (size) of a data item** - a field (data item value), a record, or a whole disk block
  - TP concepts are independent of granularity
- Basic operations on an item X:
  - **read\_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X*.
  - **write\_item(X)**: Writes the value of program variable X into the database item named X.

# Introduction to Transaction Processing (cont.)

## READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one disk block (or page). A data item X (what is read or written) will usually be the field of some record in the database, although it may be a larger unit such as a whole record or even a whole block.
- **read\_item(X) command includes the following steps:**
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

# Introduction to Transaction Processing (cont.)

## READ AND WRITE OPERATIONS (cont.):

- `write_item(X)` command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if it is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

# Transaction Notation

- Figure (next slide) shows two examples of transactions
- Notation focuses on the read and write operations
- Can also write in shorthand notation:
  - $T1: b1; r1(X); w1(X); r1(Y); w1(Y); e1;$
  - $T2: b2; r2(Y); w2(Y); e2;$
- $b_i$  and  $e_i$  specify transaction boundaries (begin and end)
- $i$  specifies a unique transaction identifier (TId)

**Figure 21.2**

Two sample transactions. (a) Transaction  $T_1$ . (b) Transaction  $T_2$ .

**(a)**

$T_1$
read_item( $X$ ); $X := X - N;$ write_item( $X$ ); read_item( $Y$ ); $Y := Y + N;$ write_item( $Y$ );

**(b)**

$T_2$
read_item( $X$ ); $X := X + M;$ write_item( $X$ );

# Why we need concurrency control

Without Concurrency Control, problems may occur with concurrent transactions:

## 1. Lost Update Problem.

Occurs when two transactions update the same data item, but both read the same original value before update (Figure 21.3(a), next slide)

## 2. The Temporary Update (or Dirty Read) Problem.

This occurs when one transaction T1 updates a database item X, which is accessed (read) by another transaction T2; then T1 fails for some reason (Figure 21.3(b)); X was (read) by T2 before its value is changed back (rolled back or UNDONE) after T1 fails

# (a) Lost update problem (b) Temporary update problem

(a)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N;$	
	read_item( $X$ ); $X := X + M;$
write_item( $X$ ); read_item( $Y$ );  $Y := Y + N;$ write_item( $Y$ );	write_item( $X$ );

**Figure 21.3**

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

Time

Item  $X$  has an incorrect value because its update by  $T_1$  is *lost* (overwritten).

(b)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N;$ write_item( $X$ );	
	read_item( $X$ ); $X := X + M;$ write_item( $X$ );
read_item( $Y$ );	

Time

Transaction  $T_1$  fails and must change the value of  $X$  back to its old value; meanwhile  $T_2$  has read the *temporary* incorrect value of  $X$ .

# Why we need concurrency control (cont.)

## 3. The Incorrect Summary Problem .

One transaction is calculating an aggregate summary function on a number of records (for example, sum (total) of all bank account balances) while other transactions are updating some of these records (for example, transferring a large amount between two accounts, see Figure 21.3(c)); the aggregate function may read some values before they are updated and others after they are updated.

# The incorrect summary problem

(c)

$T_1$	$T_3$
<pre>read_item(X); X := X - N; write_item(X);  read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>sum := 0; read_item(A); sum := sum + A; . . .  read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre>



$T_3$  reads  $X$  after  $N$  is subtracted and reads  $Y$  before  $N$  is added; a wrong summary is the result (off by  $N$ ).

# Why we need concurrency control (cont.)

## 4. The Unrepeatable Read Problem .

A transaction T1 may read an item (say, available seats on a flight); later, T1 may read the same item again and get a different value because another transaction T2 has updated the item (reserved seats on the flight) between the two reads by T1

for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

# Why recovery is needed

## Causes of transaction failure:

1. **A computer failure (system crash):** A hardware or software error occurs during transaction execution. If the hardware crashes, the contents of the computer's internal main memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

# Why recovery is needed (cont.)

## 3. Local errors or exception conditions detected by the transaction:

- certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled
- a programmed abort causes the transaction to fail.

## 4. Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 22).

# Why recovery is needed (cont.)

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This kind of failure and item 6 are more severe than items 1 through 4.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

# Transaction and System Concepts

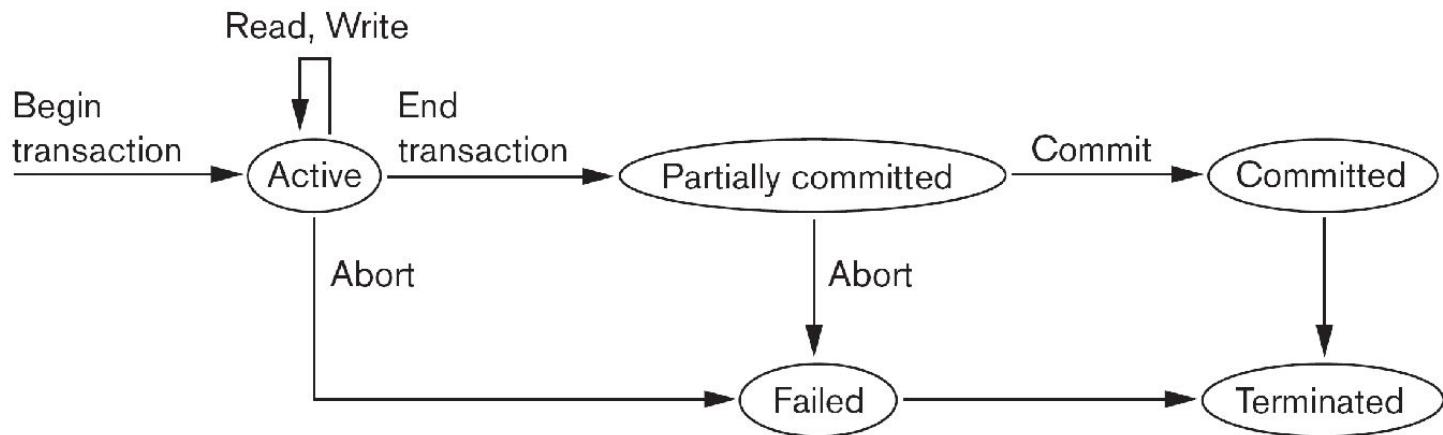
A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. A transaction passes through several states (similar to process states in operating systems).

## Transaction states:

- Active state: immediately after it starts execution (executing read, write operations)
- Partially committed state (transaction ended but waiting for system checks to determine success or failure)
- Committed state (transaction succeeded and all its changes must be recorded permanently in the database)
- Failed state (transaction failed or aborted during its active state, must be rolled back to undo the effect of its WRITE operations)
- Terminated State (transaction leaves system)

**Figure 21.4**

State transition diagram illustrating the states for transaction execution.



# Transaction and System Concepts (cont.)

DBMS Recovery Manager needs system to keep track of the following operations (in the system **log file**):

1. **begin\_transaction:** Start of transaction execution.
2. **read or write:** Read or write operations on the database items that are executed as part of a transaction.
3. **end\_transaction:** Specifies end of read and write transaction operations have ended. System may still have to check whether the changes (writes) introduced by transaction can be *permanently applied to the database* (**commit** transaction); or whether the transaction has to be *rolled back* (**abort** transaction) because it violates concurrency control or for some other reason.

# Transaction and System Concepts (cont.)

Recovery manager keeps track of the following operations (cont.):

4. **commit\_transaction:** Signals *successful end* of transaction; any changes (writes) executed by transaction can be safely **committed** to the database and will not be undone.
5. **abort\_transaction (or rollback):** Signals transaction has *ended unsuccessfully*; any changes or effects that the transaction may have applied to the database must be *undone*.

# Transaction and System Concepts (cont.)

## System operations used during recovery

- **undo(X)**: Similar to rollback except that it applies to a single write operation rather than to a whole transaction.
- **redo(X)**: This specifies that a *write operation* of a committed transaction must be *redone* to ensure that it has been applied permanently to the database on disk.

# Transaction and System Concepts (cont.)

## The System Log File

- An *append-only file* to keep track of all operations of all transactions *in the order in which they occurred*. This information is needed during recovery from failures
- Log is kept on disk - not affected on any type of failures except for disk or catastrophic failure
- As with other disk files, a *log main memory buffer* is kept for holding the records being appended until the whole buffer is appended to the end of the log file on disk
- Log is periodically backed up to archival storage (tape) to guard against catastrophic failures

# Transaction and System Concepts (cont.)

**Types of records (entries) in log file and the action each performs :**

1. [start\_transaction,T]: Records that transaction T has started execution.
2. [write\_item,T,X,old\_value,new\_value]: T has changed the value of item X from old\_value to new\_value.
3. [read\_item,T,X]: T has read the value of item X.
4. [end\_transaction,T]: T has ended execution
5. [commit,T]: T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
6. [abort,T]: T has been aborted.

# Transaction and System Concepts (cont.)

## Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log file (on disk).
- Beyond the commit point the transaction is said to be **committed**, and its effect is assumed to be *permanently recorded* in the database. The transaction then writes a commit record [commit, T] into the log.
- If a system failure occurs, we search back in the log for all transactions T that have written a [start\_transaction, T] record into the log but have not written their [commit, T] record yet; these transactions may have to be *rolled back* to undo their effect on the database during the recovery process.

# Desirable Properties of Transactions

Called **ACID properties – Atomicity, Consistency, Isolation, Durability:**

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all. This property requires that we execute a transaction to completion. If a transaction fails to complete, the recovery technique must undo any effects of the transaction on the database.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database.

# Desirable Properties of Transactions (cont.)

## ACID properties (cont.):

- **Isolation:** Even though transactions are executing concurrently, they should appear to be executed in isolation – that is, their final effect should be as if each transaction was executed in isolation from start to finish.
  - Every transaction does not make its updates visible to other transactions until it is committed, that solves the temporary update problem and eliminates cascading rollback.
- **Durability or permanency:** Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure.

# Desirable Properties of Transactions (cont.)

- **Atomicity:** Enforced by the recovery protocol. It is the responsibility of the transaction recovery system of a DBMS to ensure atomicity.
- **Consistency preservation:** Specifies that each transaction does a correct action on the database *on its own*. Application programmers and DBMS constraint enforcement are responsible for this.
- **Isolation:** Responsibility of the concurrency control protocol.
- **Durability or permanency:** Enforced by the recovery protocol.

# Schedules of Transactions

- **Transaction schedule (or history):** When transactions are executing concurrently in an interleaved fashion, the *order of execution* of operations from the various transactions forms what is known as a **transaction schedule** (or history).
- Schedule is a list of actions (reading, writing, aborting or committing) from a set of transactions and the order in which two actions of a transaction T appear in a schedule must be in the same order they appear in T.
- It represents an actual or potential execution sequence.
- Figure 21.5 (next slide) shows 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
  - Order of operations from top to bottom
  - Each schedule includes *same operations*
  - Different *order of operations* in each schedule

**Figure 21.5**

Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

(a)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

(b)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

Schedule A

Schedule B

(c)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

Time

Schedule C

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ );  read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

Schedule D

# Schedules of Transactions (cont.)

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, ...) and item name (X, Y, ...)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)
- Schedules in Figure 21.5 would be displayed as follows:
  - Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);
  - Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
  - Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);
  - Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

# Schedules of Transactions (cont.)

- When transactions are executing concurrently in an interleaved manner, then the order of execution of operations from various transactions is known as schedule
- Formal definition of a **schedule** (or **history**)  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  :  
An ordering of all the operations of the transactions subject to the constraint that, for each transaction  $T_i$  that participates in  $S$ , the operations of  $T_i$  in  $S$  must appear *in the same order* in which they occur in  $T_i$ .
- Two operations in a schedule are said to conflict if they satisfy all the following conditions.
  1. They belong to different transactions
  2. They access the same item  $X$
  3. At least one of the operations is a `write_item(X)`

# Complete schedule

A schedule S of n transactions T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>n</sub> is said to be a complete schedule with the following conditions

1. The operations in S are exactly those operations in T<sub>1</sub>, T<sub>2</sub>, ... T<sub>n</sub> including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T<sub>i</sub>, their order of appearance in S is same as their order of appearance in T<sub>i</sub>.
3. For any two conflicting operations, one of the two must occur before other in the schedule.

# Possible Schedules and characterization

- For  $n$  transactions  $T_1, T_2, \dots, T_n$ , where each  $T_i$  has  $m_i$  read and write operations, the number of possible schedules is ( $!$  is *factorial* function):

$$(m_1 + m_2 + \dots + m_n)! / ( (m_1)! * (m_2)! * \dots * (m_n)! )$$

- Generally very large number of possible schedules
- Some schedules are easy to recover from after a failure, while others are not
- Some schedules produce correct results, while others produce incorrect results
- Schedules characterized by classifying them based on ease of recovery (**recoverability**) and correctness (**serializability**)

# Characterizing Schedules based on Recoverability

Schedules classified into two main classes:

1. **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

That is a recoverable schedule is one where for each item pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before that of Tj,

2. **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.

# Example: Recoverable and non-recoverable schedules

- **Example:** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)

Schedule A: r1(X); w1(X); r2(X); w2(X); c2; r1(Y); w1(Y); c1 (or a1)

Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1); ...

- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
  - known as *cascading rollback*

# Cascadeless schedule

**Recoverable schedules** can be further refined:

A single transaction failure leads to a series of transaction rollbacks is called **cascading rollback**, which is undesirable

- **Cascadeless schedule:** A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed.
- The set of cascadeless schedules is a *subset* of the set of recoverable schedules.

**Schedules requiring cascaded rollback:** A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back.

# Example: Cascaded and Cascadeless Rollback schedules

- **Example:** Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits
- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the  $r_2(X)$  of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C
- Schedule B:  $r_1(X)$ ;  $w_1(X)$ ;  $r_2(X)$ ;  $w_2(X)$ ;  $r_1(Y)$ ;  $w_1(Y)$ ;  $c_1$  (or  $a_1$ );
- Schedule C:  $r_1(X)$ ;  $w_1(X)$ ;  $r_1(Y)$ ;  $w_1(Y)$ ;  $c_1$ ;  $r_2(X)$ ;  $w_2(X)$ ; ...

# Strict schedule

**Cascadeless schedules** can be further refined:

- **Strict schedule:** A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- The set of strict schedules is a *subset of* the set of cascadeless schedules.
- If *blind writes* are not allowed, all cascadeless schedules are also strict

**Blind write:** A write operation  $w_2(X)$  that is not preceded by a read  $r_2(X)$ .

# Strict schedule: Example

- **Example:** Schedule C below is **cascadeless** and also **strict** (because it has no blind writes)
  - Schedule D is cascadeless, but not strict (because of the blind write  $w_3(X)$ , which writes the value of X before T1 commits)
  - To make it strict,  $w_3(X)$  must be delayed until after T1 commits – see Schedule E
- 
- Schedule C:  $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); \dots$
  - Schedule D:  $r_1(X); w_1(X); \underline{w_3(X)}; r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); \dots$
  - Schedule E:  $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; \underline{w_3(X)}; r_2(X); w_2(X); \dots$

# Characterizing Schedules Based on Recoverability (cont.)

## Summary:

- Many schedules can exist for a set of transactions
- The set of all possible schedules can be partitioned into two subsets: **recoverable** and **non-recoverable**
- A subset of the recoverable schedules are **cascadeless**
- If blind writes are allowed, a subset of the cascadeless schedules are **strict**
- If *blind writes are not allowed*, the set of cascadeless schedules is the same as the set of strict schedules

# Characterizing Schedules based on Serializability

- Among the large set of possible schedules, we want to characterize which schedules are *guaranteed to give a correct result*
- The **consistency preservation** property of the ACID properties states that each transaction if executed on its own (from start to finish) will transform a consistent state of the database into another consistent state
- Hence, each transaction is *correct on its own*

# Serial Schedules

- **Serial schedule:** A schedule  $S$  is **serial** if, for every transaction  $T$  participating in the schedule, all the operations of  $T$  are executed consecutively (without interleaving of operations from other transactions) in the schedule. Otherwise, the schedule is called **non-serial**.
- Based on the consistency preservation property, *any serial schedule will produce a correct result* (assuming no inter-dependencies among different transactions)

# Characterizing Schedules based on Serializability (cont.)

- Serial schedules are *not feasible* for performance reasons:
  - No interleaving of operations
  - Long transactions force other transactions to wait
  - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
  - Need to allow concurrency with interleaving without sacrificing correctness

# Serializable Schedules

- **Serializable schedule:** A schedule  $S$  is **serializable** if it is **equivalent** to some serial schedule of the same  $n$  transactions.
- There are  $(n)!$  serial schedules for  $n$  transactions – a serializable schedule can be equivalent to *any of the serial schedules*
- **Question:** How do we define equivalence of schedules?

# Equivalence of Schedules

- **Result equivalent:** Two schedules are called result equivalent if they produce the same final state of the database.
- Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general.
- May also get result equivalence *by chance* for a particular input parameter even though schedules *are not equivalent in general* (see Figure 21.6, next slide)

### Figure 21.6

Two schedules that are result equivalent for the initial value of  $X = 100$  but are not result equivalent in general.

$S_1$	$S_2$
<code>read_item(<math>X</math>); <math>X := X + 10;</math> <code>write_item(<math>X</math>);</code></code>	<code>read_item(<math>X</math>); <math>X := X * 1.1;</math> <code>write_item (<math>X</math>);</code></code>

# Equivalence of Schedules (cont.)

- **Conflict equivalent:** Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.
- Commonly used definition of schedule equivalence
- Two operations are **conflicting** if:
  - They access the same data item X
  - They are from two different transactions
  - At least one is a write operation
- Read-Write conflict example: r1(X) and w2(X)
- Write-write conflict example: w1(Y) and w2(Y)

# Equivalence of Schedules (cont.)

- Changing the order of conflicting operations generally *causes a different outcome*
- **Example:** changing  $r_1(X); w_2(X)$  to  $w_2(X); r_1(X)$  means that T1 will read a *different value for X*
- **Example:** changing  $w_1(Y); w_2(Y)$  to  $w_2(Y); w_1(Y)$  means that the final value for Y in the database can be different
- Note that read operations are **not conflicting**; changing  $r_1(Z); r_2(Z)$  to  $r_2(Z); r_1(Z)$  does not change the outcome

# Characterizing Schedules Based on Serializability (cont.)

- **Conflict equivalence** of schedules is used to determine which schedules are correct in general (serializable)

A schedule  $S$  is said to be **Serializable** if it is conflict equivalent to some serial schedule  $S'$ .

# Characterizing Schedules based on Serializability (cont.)

- A serializable schedule is considered to be correct because it is equivalent to a serial schedule, and any serial schedule is considered to be correct
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution and interleaving of operations from different transactions.

# Characterizing Schedules based on Serializability (cont.)

- Serializability is generally hard to check at run-time:
  - Interleaving of operations is generally handled by the operating system through the process scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved
  - Transactions are continuously started and terminated

# Characterizing Schedules based on Serializability (cont.)

## Testing for conflict serializability

### Algorithm 21.1:

- Looks at only  $r(X)$  and  $w(X)$  operations in a schedule
- Constructs a precedence graph (serialization graph) – **one node for each transaction**, plus directed edges
- An **edge is created** from  $T_i$  to  $T_j$  if one of the operations in  $T_i$  appears before a conflicting operation in  $T_j$
- The schedule is serializable if and only if the precedence graph **has no cycles**.

### **Algorithm 21.1.** Testing Conflict Serializability of a Schedule $S$

1. For each transaction  $T_i$  participating in schedule  $S$ , create a node labeled  $T_i$  in the precedence graph.
2. For each case in  $S$  where  $T_j$  executes a `read_item( $X$ )` after  $T_i$  executes a `write_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
3. For each case in  $S$  where  $T_j$  executes a `write_item( $X$ )` after  $T_i$  executes a `read_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
4. For each case in  $S$  where  $T_j$  executes a `write_item( $X$ )` after  $T_i$  executes a `write_item( $X$ )`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
5. The schedule  $S$  is serializable if and only if the precedence graph has no cycles.

**Figure 21.5**

Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

(a)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

(b)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

Schedule A

Schedule B

(c)

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ ); read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

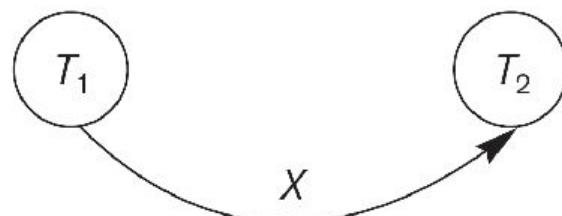
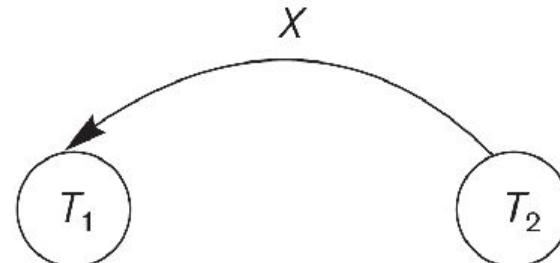
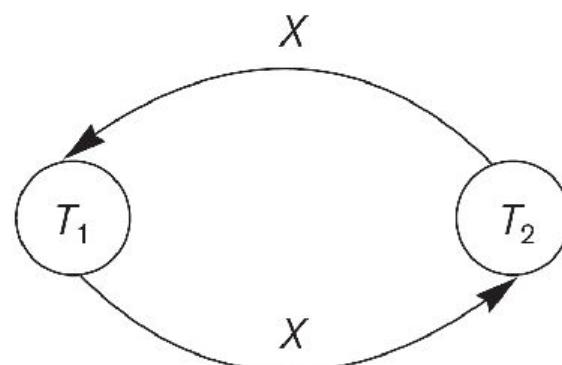
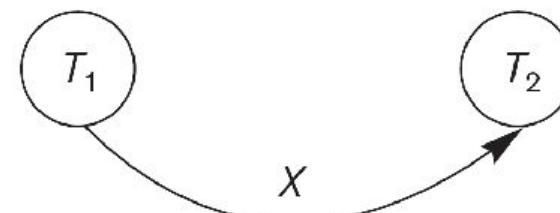
Time

Schedule C

$T_1$	$T_2$
read_item( $X$ ); $X := X - N$ ; write_item( $X$ );  read_item( $Y$ ); $Y := Y + N$ ; write_item( $Y$ );	read_item( $X$ ); $X := X + M$ ; write_item( $X$ );

Time

Schedule D

**(a)****(b)****(c)****(d)****Figure 21.7**

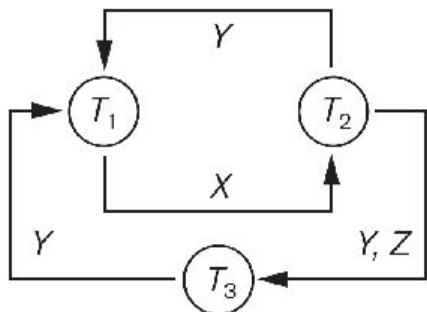
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

**Addison-Wesley**  
is an imprint of



Copyright © 2011 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

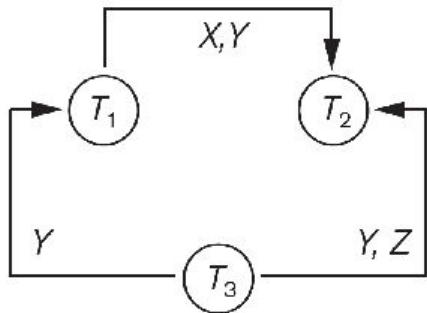
(d)

**Equivalent serial schedules**

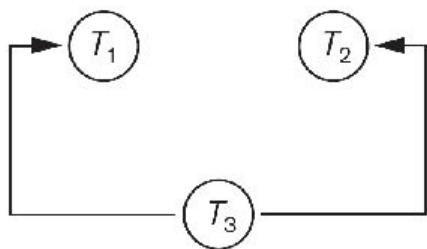
None

**Reason**Cycle  $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$ Cycle  $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$ 

(e)

**Equivalent serial schedules** $T_3 \rightarrow T_1 \rightarrow T_2$ 

(f)

**Equivalent serial schedules** $T_3 \rightarrow T_1 \rightarrow T_2$  $T_3 \rightarrow T_2 \rightarrow T_1$ **Figure 21.8 (continued)**

Another example of serializability testing.

(d) Precedence graph for schedule E.

(e) Precedence graph for schedule F.

(f) Precedence graph with two equivalent serial schedules.

# View Serializability

- **View equivalence:** A less restrictive definition of equivalence of schedules than conflict serializability *when blind writes are allowed*
- **View serializability:** A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

# View Serializability (cont.)

Two schedules are said to be **view equivalent** if the following three conditions hold:

- The same set of transactions participates in  $S$  and  $S'$ , and  $S$  and  $S'$  include the same operations of those transactions.
- For any operation  $R_i(X)$  of  $T_i$  in  $S$ , if the value of  $X$  read was written by an operation  $W_j(X)$  of  $T_j$  (or if it is the original value of  $X$  before the schedule started), the same condition must hold for the value of  $X$  read by operation  $R_i(X)$  of  $T_i$  in  $S'$ .
- If the operation  $W_k(Y)$  of  $T_k$  is the last operation to write item  $Y$  in  $S$ , then  $W_k(Y)$  of  $T_k$  must also be the last operation to write item  $Y$  in  $S'$ .

# View Serializability (cont.)

## The premise behind view equivalence:

- Each read operation of a transaction reads the result of *the same write operation* in both schedules.
- “**The view**”: the read operations are said to see the *same view* in both schedules.
- The final write operation on each item is the same on both schedules resulting in the same final database state in case of blind writes

# Characterizing Schedules based on Serializability (cont.)

## Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** (no blind writes allowed)
- Conflict serializability is **stricter** than view serializability when **blind writes occur** (a schedule that is view serializable is not necessarily conflict serializable).
- Any conflict serializable schedule is also view serializable, but not vice versa.

# Characterizing Schedules based on Serializability (cont.)

## Relationship between view and conflict equivalence (cont):

Consider the following schedule of three transactions

T1: r1(X); w1(X);      T2: w2(X);      and      T3: w3(X);

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T2 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to any serial schedule.



# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module IV**  
QUERY OPTIMIZATION

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

## Module 4: PHYSICAL DATA ORGANIZATION AND QUERY OPTIMIZATION

- **Physical Data Organization:** index structures, primary, secondary and clustering indices, Single level and Multi-level indexing, B+-Trees (basic structure only, algorithms not needed), (Reading Elmasri and Navathe, Ch. 17.1-17.4)
- **Query Optimization:** heuristics-based query optimization, (Reading Elmasri and Navathe, Ch. 18.1, 18.7)

# Query optimization

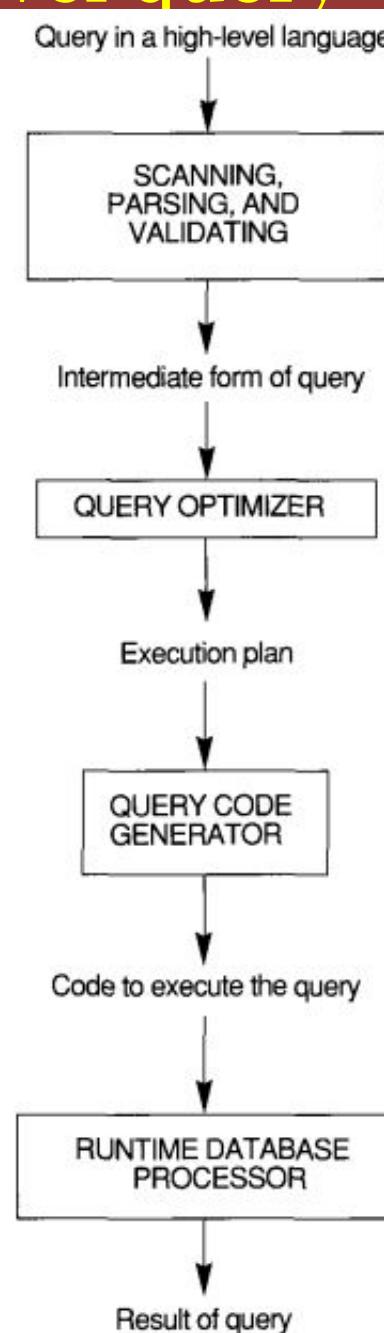
Heuristics-based query optimization

# Query Processing

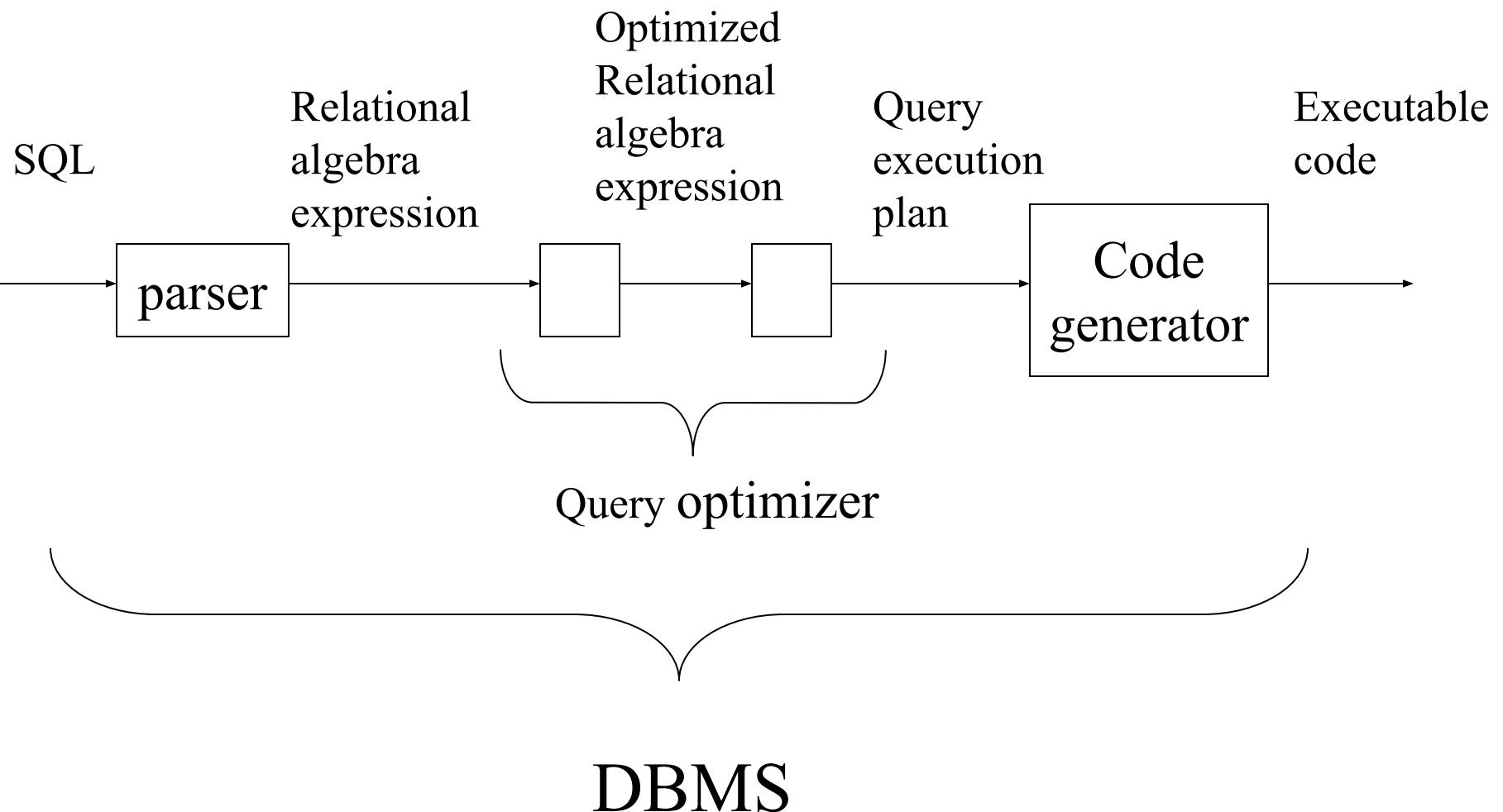
- Query processing refers to the range of activities involved in extracting data from a database.
- Steps involved in query processing:
  1. Translation of queries in high-level database language into expressions that can be used at the physical level of the file system.
  2. Query optimizing transformations
  3. Evaluation of queries

# Steps when processing a high-level query

- System constructs a parse-tree representation of the query, which then translates into relational-algebra expression.
- The query optimizer module has the task of producing an execution plan,
- The code generator generates the code to execute that plan.
- The runtime database processor has the task of running the query code

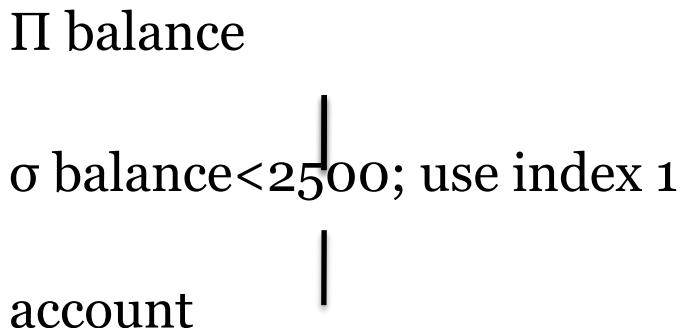


# Relational algebra, SQL, and the DBMS?



# Example

- Consider the query  
 $\text{select } \mathbf{balance} \text{ from } \mathbf{account} \text{ where } \mathbf{balance} < 2500$
- Query can be translated into either of the following:
  1.  $\sigma_{\mathbf{balance} < 2500} (\Pi_{\mathbf{balance}} (\mathbf{account}))$
  2.  $\Pi_{\mathbf{balance}} (\sigma_{\mathbf{balance} < 2500} (\mathbf{account}))$
- Then execute each relational–algebra operation by one of algorithms.
- Next is the query evaluation (or execution) plan



# Query optimization

- It is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query execution.
- Query optimization is **the process of selecting an efficient execution plan for evaluating the query.**
- The two forms of query optimization are: –
- **Heuristic optimization** – Here the query execution is refined based on heuristic rules for reordering the individual operations.
- **Cost based optimization** – the overall cost of executing the query is systematically reduced by estimating the costs of executing several different execution plans.

# USING HEURISTICS IN QUERY OPTIMIZATION

- One of the main heuristic rules is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations. This is because the size of the file resulting from a binary operation-such as JOIN-is usually a multiplicative function of the sizesof the input files.
- The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.
- Instead of using a relational-algebra representation, several databases use an annotated parse-tree representation based on the structure of the given SQL query.

# Notation for Query Trees and Query Graphs

- A query tree is used to represent a relational algebra or extended relational algebra expression, whereas a query graph is used to represent a relational calculus expression.  
A query tree represents the input relations of the query as *leafnodes* of the tree, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
- The execution terminates when the root node is executed and produces the result relation for the query.
- There are many trees for the same query
  - Trees always have a strict order among their operations
  - Query optimization must find “best” order

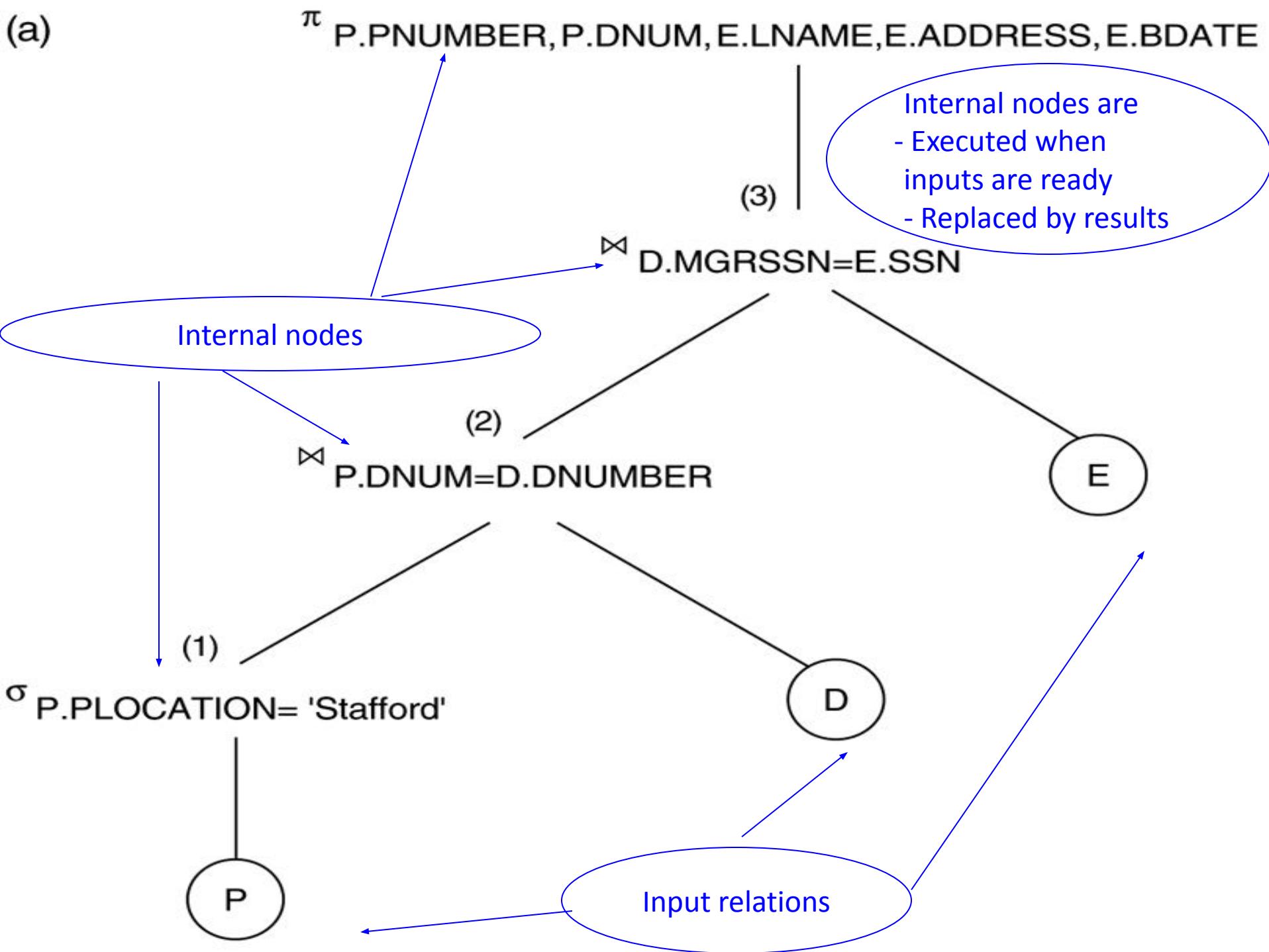
# Sample Query

**Example:** For every project located in ‘Stafford’, retrieve the project number, the controlling department number and the department manager’s last name, address and birthdate

```
SELECT P.NUMBER,P.DNUM,E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P,DEPARTMENT AS D, EMPLOYEE AS E
WHERE P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
P.PLOCATION='STAFFORD';
```

Relational algebra:

$$\pi_{PNUMBER,DNUM,LNAME,ADDRESS,BDATE}((\sigma_{PLOCATION='STAFFORD'}(PROJECT) \bowtie_{DNUM=DNUMBER}(DEPARTMENT)) \bowtie_{MGRSSN=SSN}(EMPLOYEE))$$



# Different representation for the same algebra expression

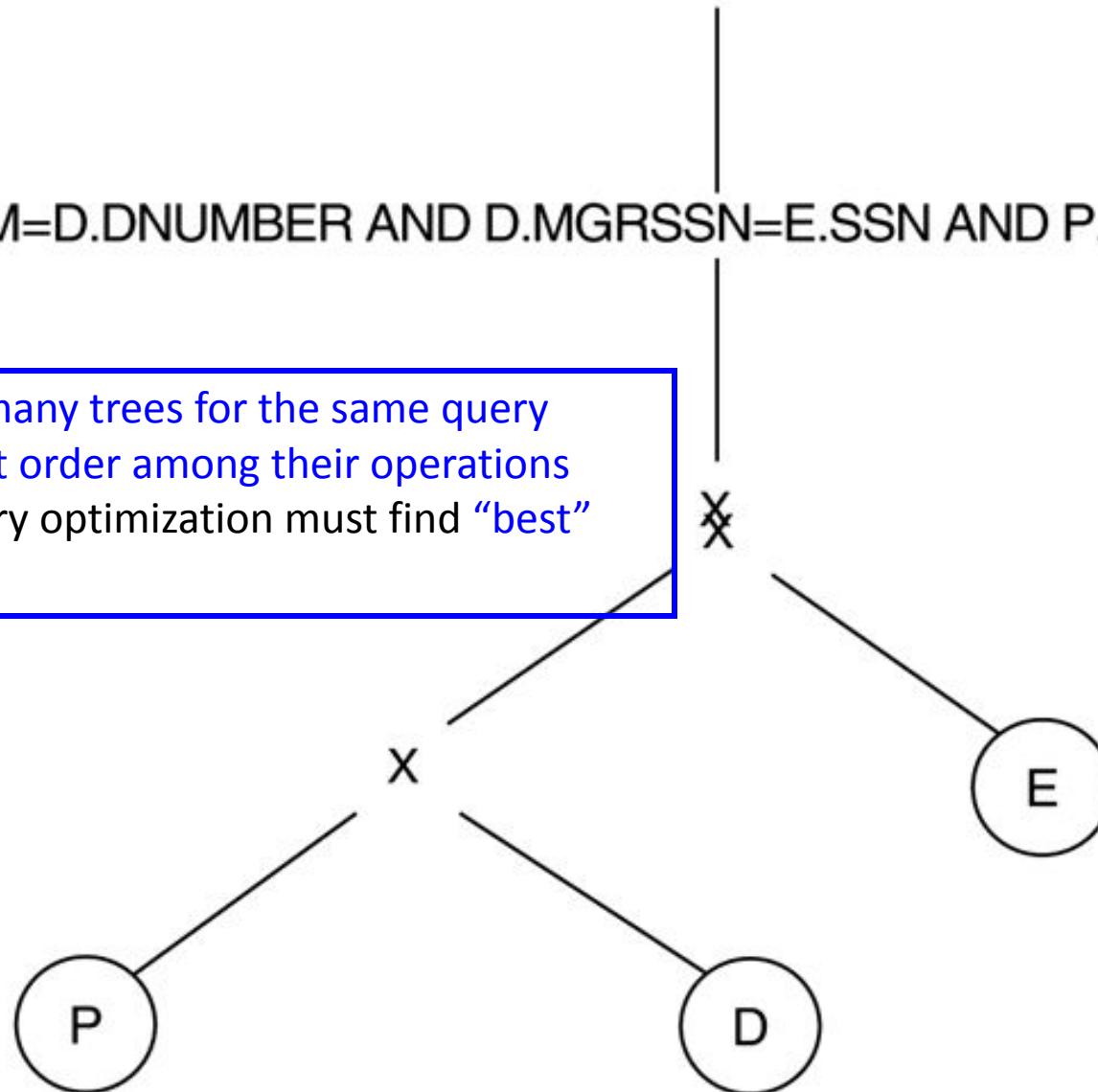
... assumed to be the initial form

(b)  $\pi^{\pi} P.PNUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE$

$\sigma P.DNUM=D.DNUMBER \text{ AND } D.MGRSSN=E.SSN \text{ AND } P.PLOCATION='Stafford'$

There are many trees for the same query

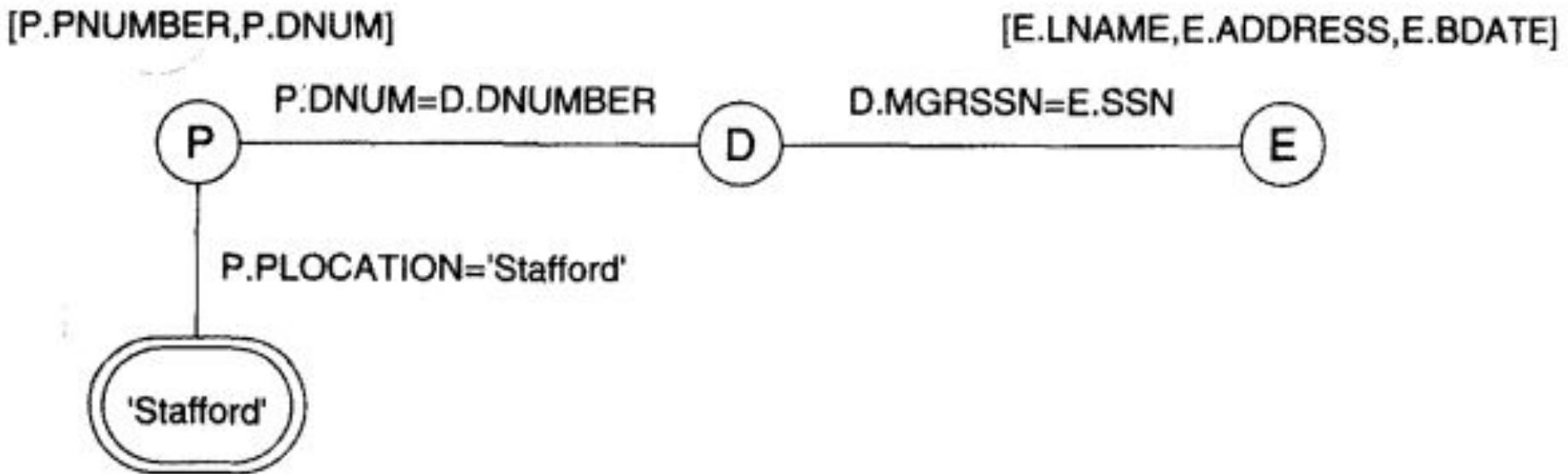
- strict order among their operations
- Query optimization must find “best” order



- In Figure (a) the three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E, while the relational algebra operations of the expression are represented by internal tree nodes.
- When this query tree is executed, the node marked (1) in the Figure must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2).
- Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

# Query graph

- Q2:** `SELECT P.PNUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE  
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E  
WHERE P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND  
P. PLOCATION=' STAFFORD' ;`



- the query tree represents a specific order of operations for executing a query.
- A more neutral representation of a query is the query graph notation.
- Figure c shows the query graph for the given query.
- Relations in the query are represented by relation nodes, which are displayed as single circles.
- Constant values, typically from the query selection conditions, are represented by constant nodes, which are displayed as double circles or ovals.
- Selection and join conditions are represented by the graph edges.
- Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

- The query graph representation does not indicate an order on which operations to perform first.
- There is only a single graph corresponding to each query.
- Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

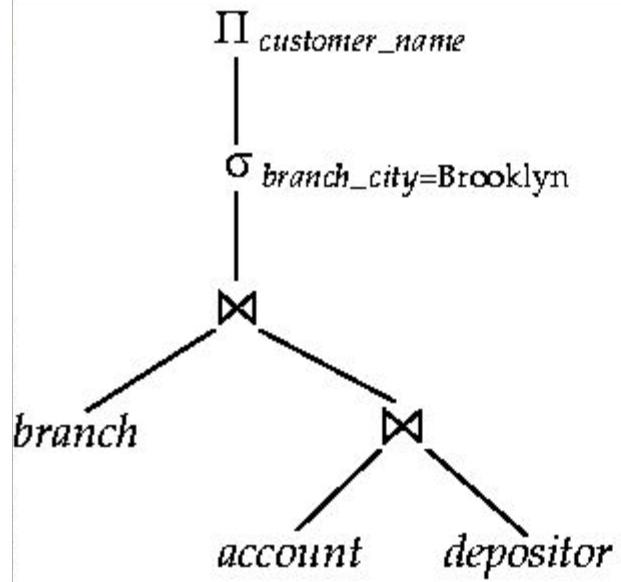
# Heuristic Optimization of Query Trees

- There are many different relational algebra expressions-and hence many different query trees-can be equivalent; that is, they can correspond to the same query.
- The query parser will generate a standard initial query tree to correspond to an SQL query, without doing any optimization.
- For example, for a select-project-join query, such as Q2, the initial tree is shown in Figure b.
- The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes.
- Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly*, because of the CARTESIAN PRODUCT (X) operations.

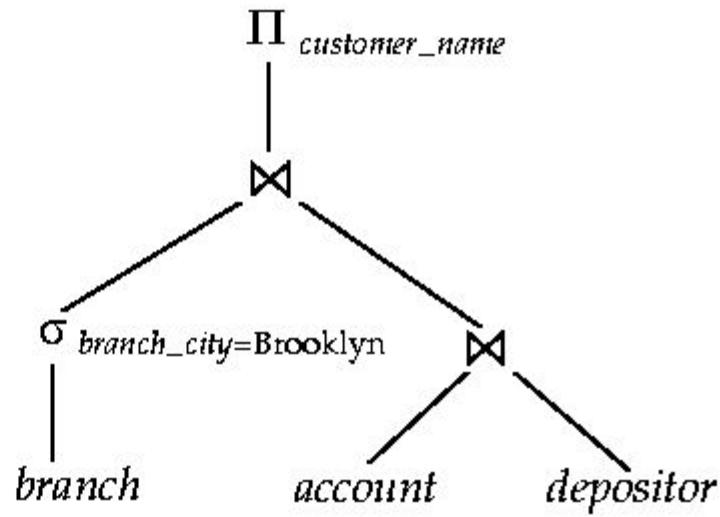
- For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5000 tuples, respectively,
- The result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each.
- However, the querytree in Figure b is in a simple standard form that can be easily created. It is now the job of the heuristic query optimizer to transform this initial query tree into a final query tree that is efficient to execute.
- The optimizer must include rules for equivalence among relational algebra expressions that can be applied to the initial tree.
- The heuristic query optimization rules then utilize these equivalence expressions to transform the initial tree into the final, optimized query tree.

# Example of Transforming a Query

- Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples
  - although their tuples/attributes may be ordered differently.



(a) Initial expression tree



(b) Transformed expression tree

# Example of Transforming a Query

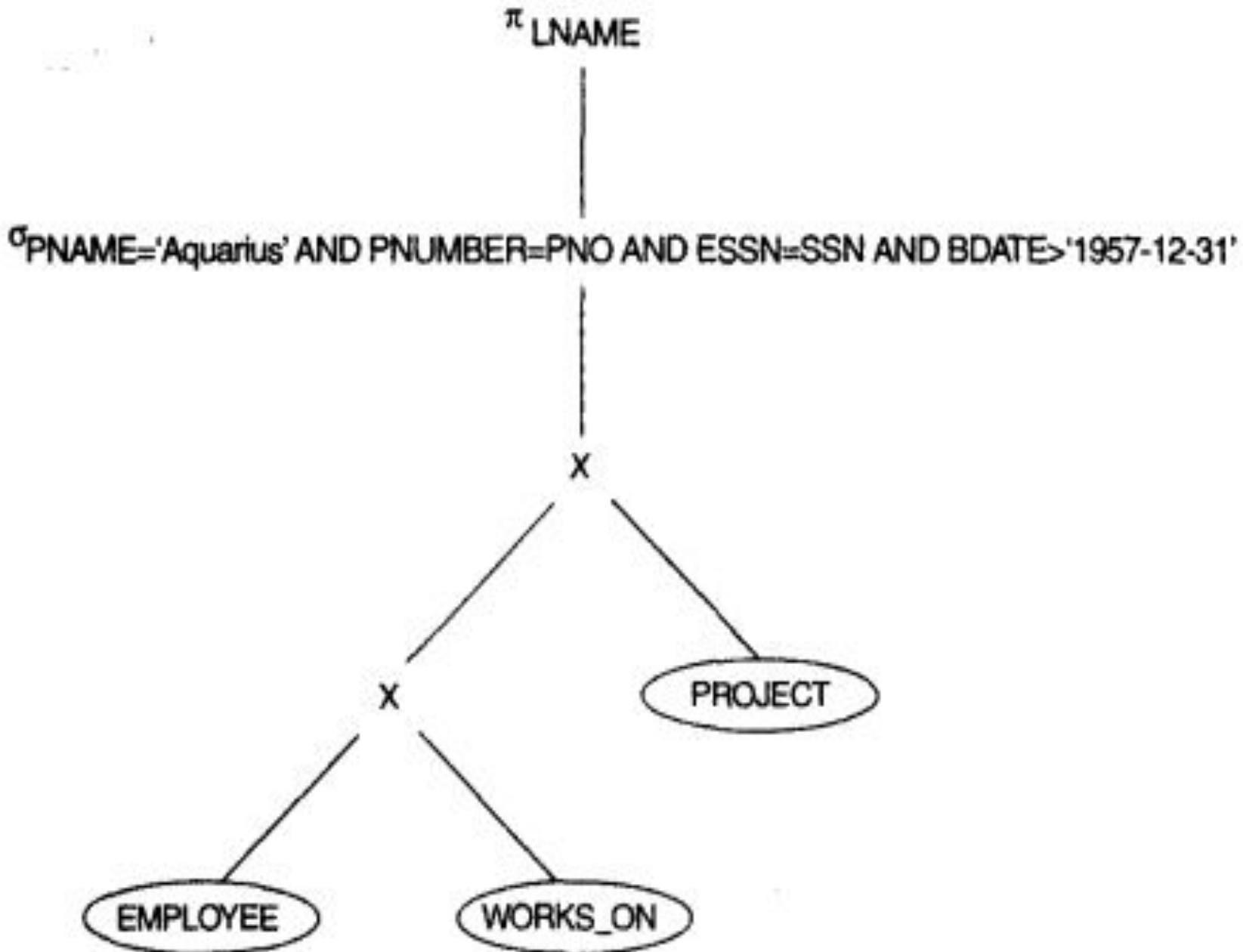
- Consider the query "Find last names of employees born after 1957 who work on a project named 'Aquarius'."
- This query can be specified in SQL as follows:

```
SELECT LNAME FROM EMPLOYEE, WORKS_ON, PROJECT  
WHERE PNAME='AQUARIUS' AND PNUMBER=PNO AND ESSN=SSN AND  
BDATE > '1957-12-31';
```

- Initial query tree for Q is shown in Figure a. Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of entire EMPLOYEE, WORKS\_ON, and PROJECT files.
- However, this query needs only one record from the PROJECT relation for the 'Aquarius' project-and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'.

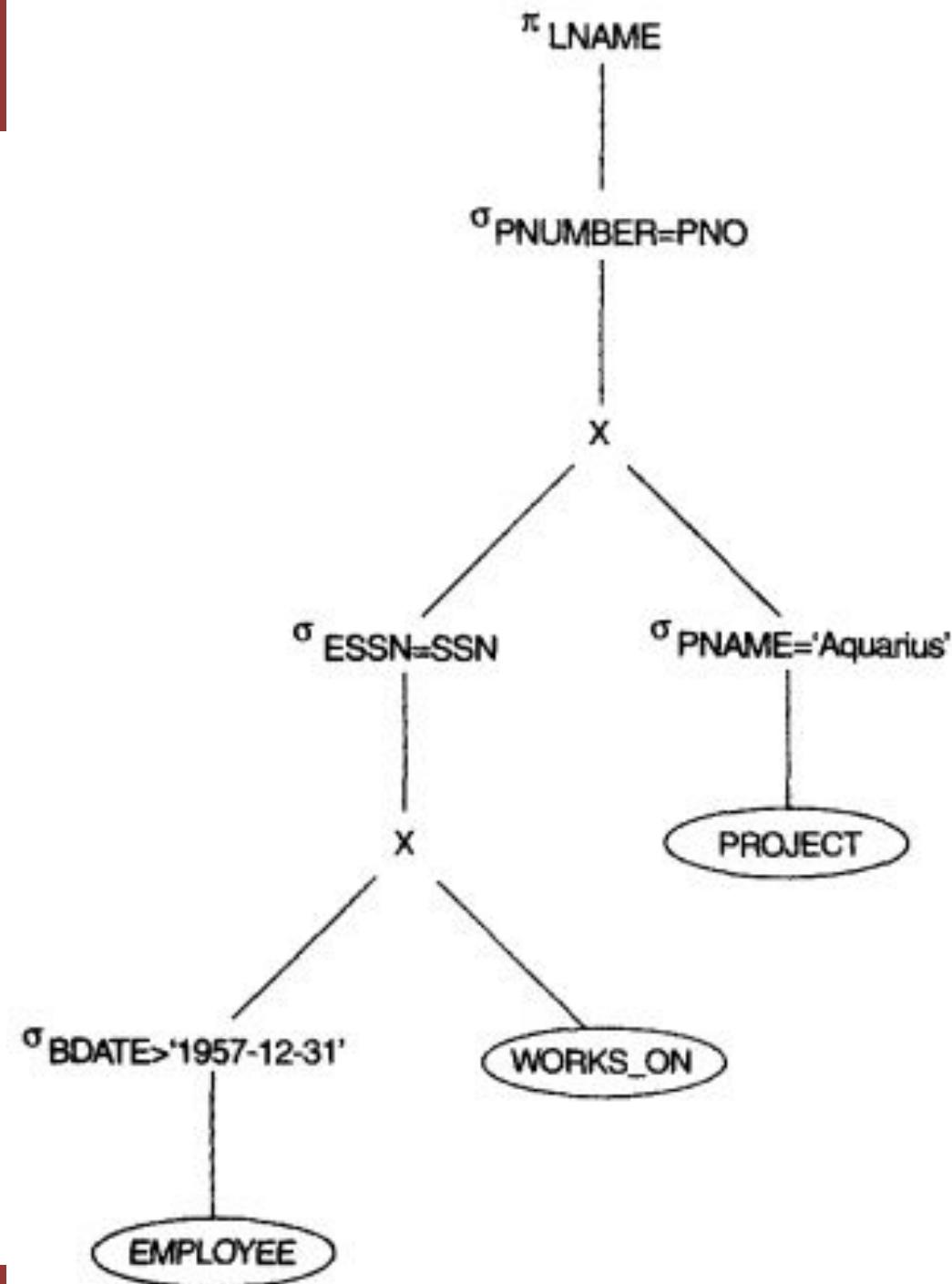
# Initial (canonical) query tree for SQL query

a)



## Figure b

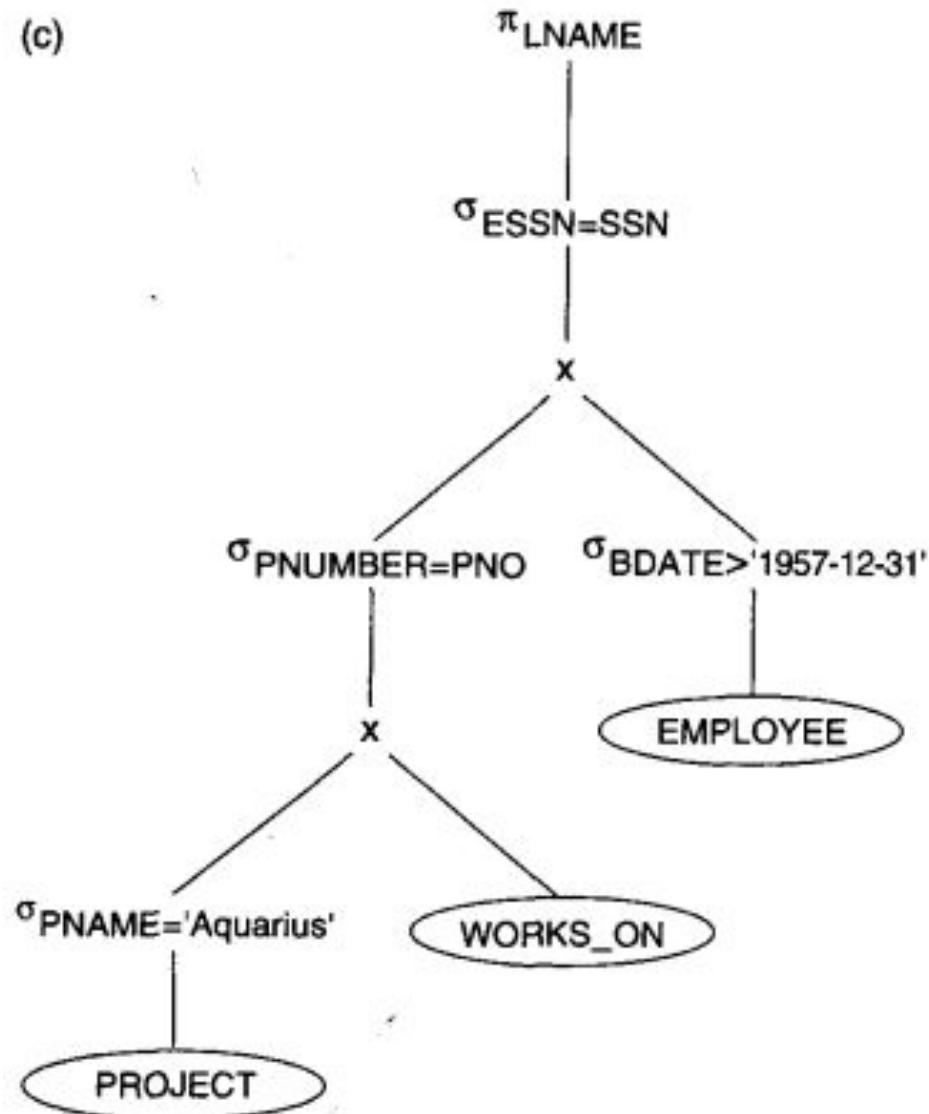
This Figure shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.



# Figure c

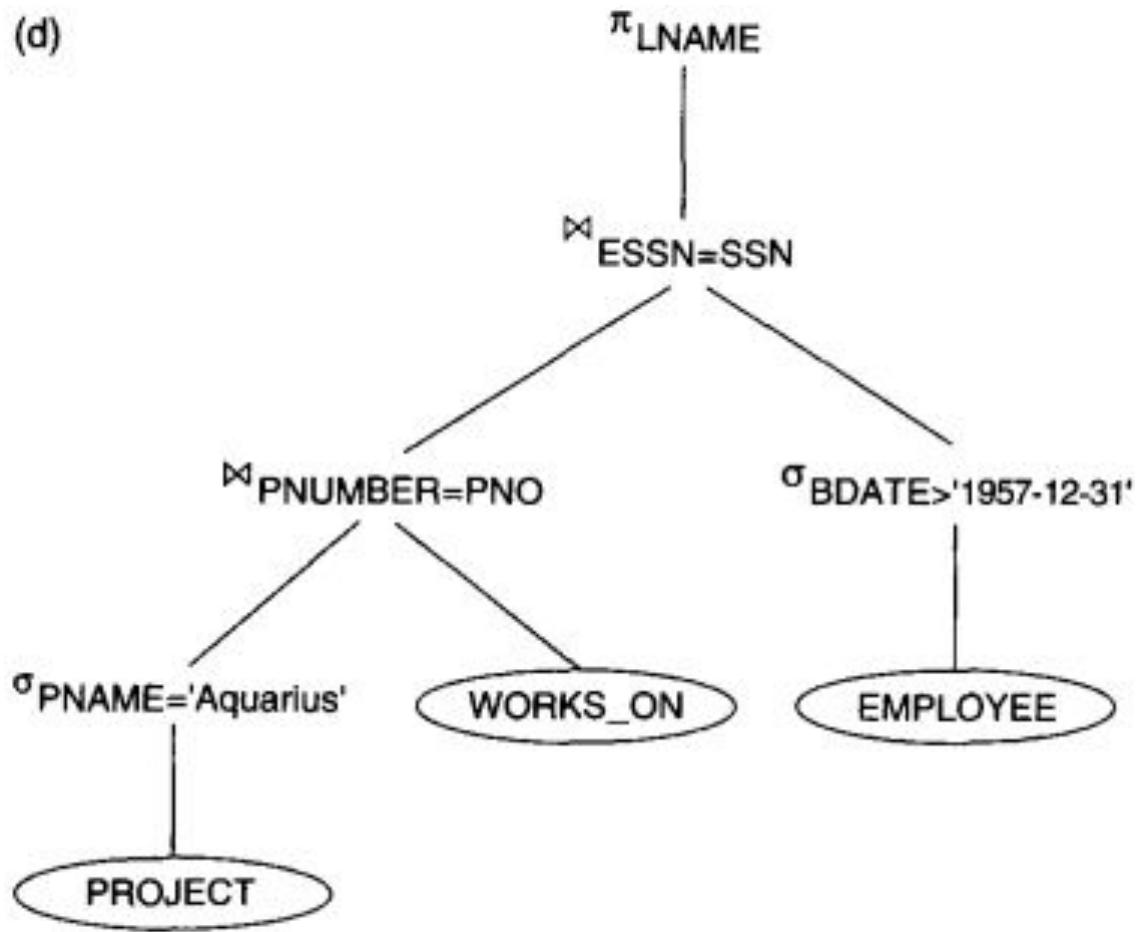
A further improvement is achieved (c) by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure c.

This uses the information that PNUMBER is a key attribute of the project relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.



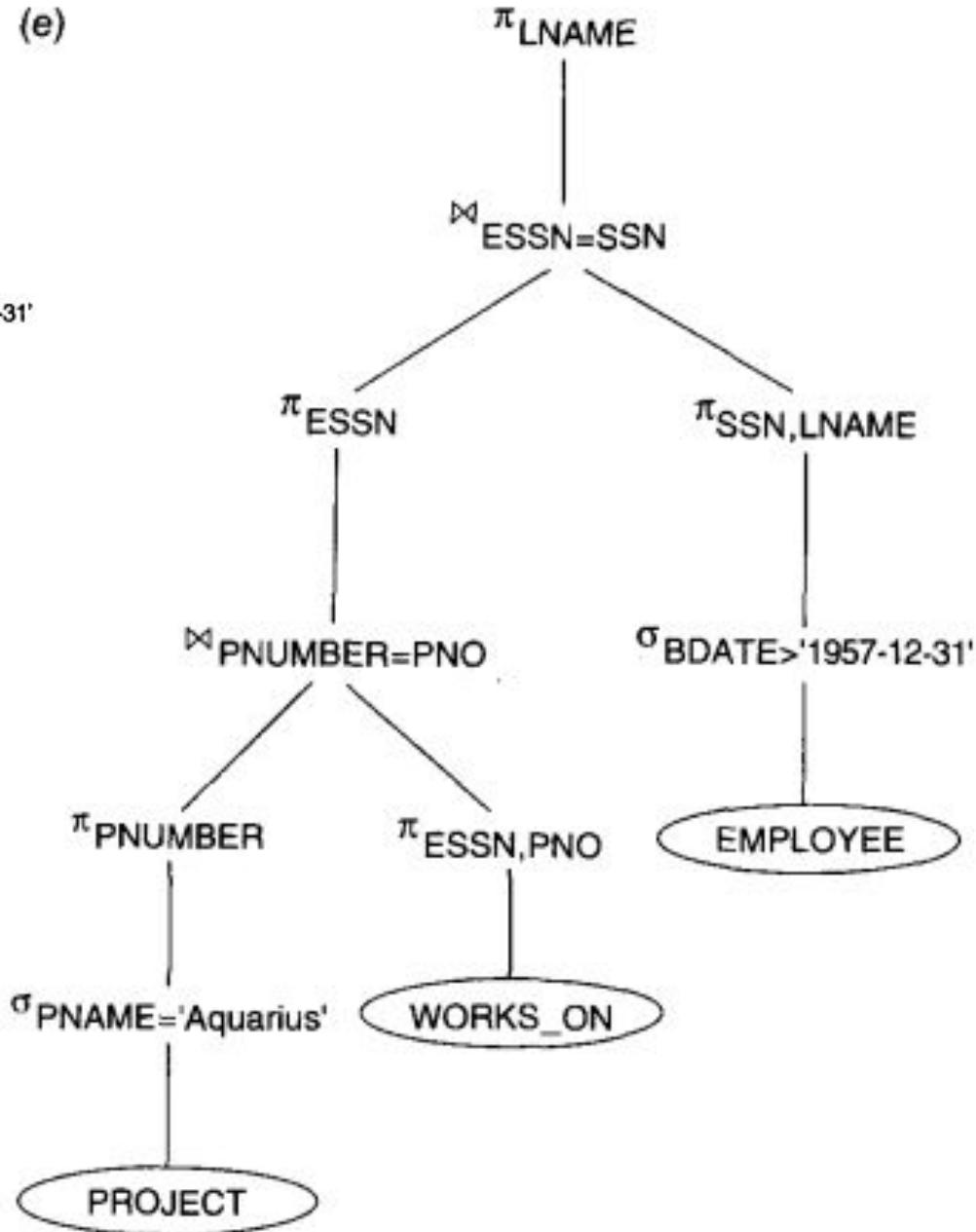
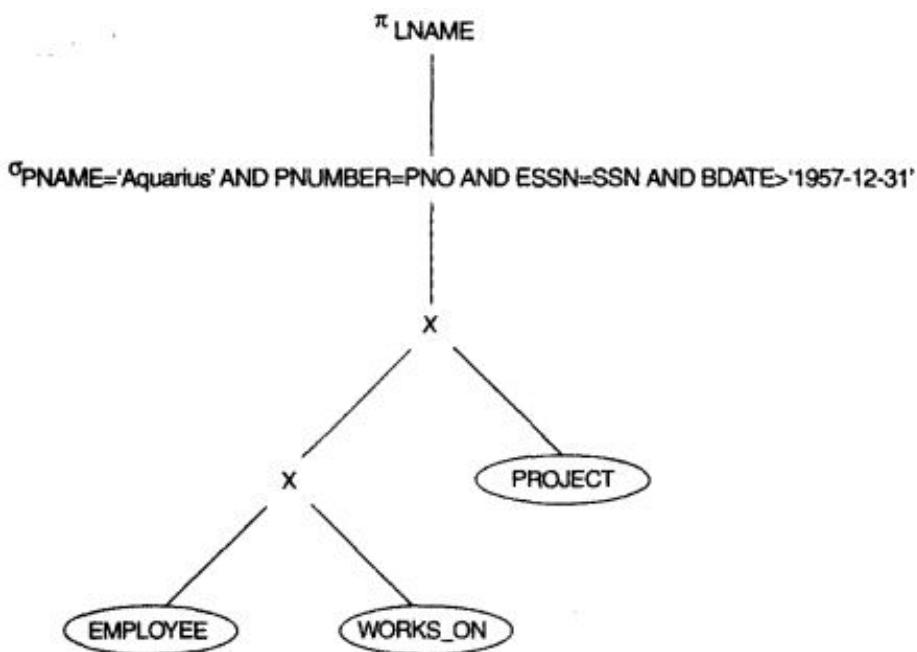
# Figure d

We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure d.



# Moving PROJECT operations down the query tree

(e)



# Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
  - we don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa
  - There are many rules for transforming relational algebra operations into equivalent ones.

# Equivalence Rules

- Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

- Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

- Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\bowtie (\Pi_{L_n}(E))\bowtie )) = \Pi_{L_1}(E)$$

- Selections can be combined with Cartesian products and theta joins.

a.  $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

# Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \underset{\theta}{\bowtie} E_2 = E_2 \underset{\theta}{\bowtie} E_1 \quad \bowtie$$

6. (a) Natural join operations are associative:

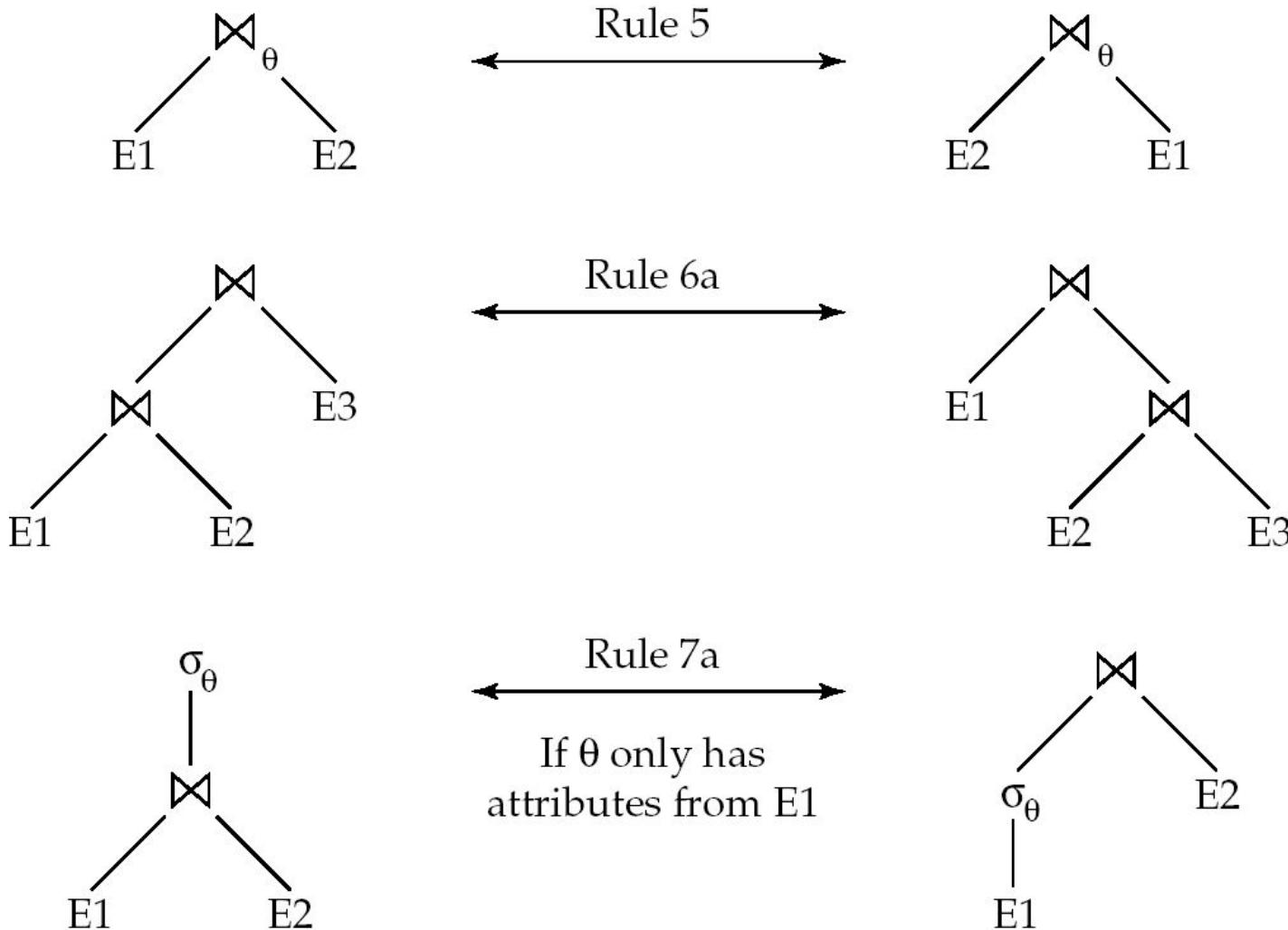
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3) \quad \bowtie$$

- (b) Theta joins are associative in the following manner:

$$(E_1 \underset{\theta_1}{\bowtie} E_2) \underset{\theta_2 \wedge \theta_3}{\bowtie} E_3 = E_1 \underset{\theta_1 \wedge \theta_3}{\bowtie} (E_2 \underset{\theta_2}{\bowtie} E_3)$$

where  $\theta_2$  involves attributes from only  $E_2$  and  $E_3$ .

# Pictorial Depiction of Equivalence Rules



# Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
- When all the attributes in  $\theta_o$  involve only the attributes of one of the expressions ( $E_1$ ) being joined.

$$\sigma_{\theta_o}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_o}(E_1)) \bowtie_{\theta} E_2$$

- When  $\theta_1$  involves only the attributes of  $E_1$  and  $\theta_2$  involves only the attributes of  $E_2$ .

$$\sigma_{\theta_1} \wedge_{\theta_2} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

# Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

(a) if  $\theta$  involves only attributes from  $L_1 \cup L_2$ :

$$\prod_{L_1 \cup L_2} (E_1 \bowtie_{\theta} E_2) = (\prod_{L_1} (E_1)) \bowtie_{\theta} (\prod_{L_2} (E_2))$$

(b) Consider a join  $E_1 \bowtie_{\theta} E_2$ .

- Let  $L_1$  and  $L_2$  be sets of attributes from  $E_1$  and  $E_2$ , respectively.
- Let  $L_3$  be attributes of  $E_1$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ , and
- let  $L_4$  be attributes of  $E_2$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ .

$$\prod_{L_1 \cup L_2} (E_1 \bowtie_{\theta} E_2) = \prod_{L_1 \cup L_2} ( \prod_{L_1 \cup L_3} (E_1) \bowtie_{\theta} (\prod_{L_2 \cup L_4} (E_2)) )$$

# Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

- (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over  $\cup$ ,  $\cap$  and  $-$ .

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$

and similarly for  $\cup$  and  $\cap$  in place of  $-$

Also:  $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for  $\cap$  in place of  $-$ , but not for  $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

# Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

# Steps involved in query optimization

- Moving selection down
- Re-ordering relations
- Moving projection down
- Replace selection and cross product by join

# Questions

- Consider three tables  
COURSE(CNO, CNAME,CREDITS),  
STUDENT(ROLLNO,NAME,ADDRESS,SEM) and  
ENROLLMENT(CNO,ROLLNO,GRADE)
- Foreign keys have the same name as primary keys.  
Identify one initial canonical query tree for the  
following SQL expression and show the steps to  
optimize it using heuristics. Assume that CNAME is a  
candidate key.

SELECT S.NAME, S.ADDRESS, E.GRADE FROM  
COURSE C, STUDENT S, ENROLLMENT E WHERE  
S.ROLLNO=E.ROLLNO AND C.CNO=E.CNO AND  
CNAME='PDBD'.

# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module III-Part2**  
Lossless and Dependency Preserving Decompositions

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Closures

- **Closure** of a set of attributes  $X$  with respect to  $F$  is the set  $X^+$  of all attributes that are functionally determined by  $X$ .  $X^+$  can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in  $F$  (Given set of Functional Dependencies).
  - Closure is used to find the candidate keys of  $R$  and compute  $F^+$
1. Let the relation  $R(A,B,C,D,E,F)$   
 $F: B \rightarrow C, BC \rightarrow AD, D \rightarrow E, CF \rightarrow B$   
Find the closure of  $B$ .  
 $B^+ = \{B, C, A, D, E\}$

# Algorithm for determining $X^+$

**Algorithm** for determining the closure of  $X$  under  $F$  ( $X^+$ )

- Let us assume that  $X^+$  equals all the attributes in  $X$ ; By IR1 all attributes are functionally dependent on  $X$ , Using IR2 and IR3 we add attributes to  $X^+$  using each functional dependency in  $F$

**begin function**

$X^+ = X;$

**while true do**

**if there exists  $(Y \square Z)$  in  $F$  such that  $X^+ \supseteq Y$  and  $X^+ \neq Z$**

**then  $X^+ = X^+ \cup Z;$**

**else exit;**

**return  $X^+$**

**end**

$R(A,B,C,D,E,F)$  WHERE  $F:A \rightarrow BC$ ,  $B \rightarrow D$ ,  $C \rightarrow DE$ ,  $BC \rightarrow F$ . Then, find the candidate keys of  $R$ .

Solution

$A^+ = \{A, B, C, D, E, F\} = \{R\} \Rightarrow A$  is a candidate key

$B^+ = \{B, D\} \Rightarrow B$  is not a candidate key

$C^+ = \{C, D, E\} \Rightarrow C$  is not a candidate key

$BC^+ = \{B, C, D, E, F\} \Rightarrow BC$  is not a candidate key

- Closure of  $F$  ( $F^+$ ):  $F^+$  is the set of all FDs that can be inferred/ derived from  $F$ . Using Armstrong Axioms repeatedly on  $F$ , we can compute all the FDs.

# Example 1

- Consider a relation  $R(A,B,C,D,E,F)$   
 $F: E \rightarrow A, E \rightarrow D, A \rightarrow C, A \rightarrow D, AE \rightarrow F, AG \rightarrow K.$   
 Find the closure of  $E$  or  $E^+$

## Solution

- The closure of  $E$  or  $E^+$  is as follows –

$$\begin{aligned}
 E^+ &= E \\
 &= EA \quad \{ \text{for } E \rightarrow A \text{ add } A \} \\
 &= EAD \quad \{ \text{for } E \rightarrow D \text{ add } D \} \\
 &= EADC \quad \{ \text{for } A \rightarrow C \text{ add } C \} \\
 &= EADC \quad \{ \text{for } A \rightarrow D \text{ D already added} \} \\
 &= EADCF \quad \{ \text{for } AE \rightarrow F \text{ add } F \} \\
 &= EADCF \quad \{ \text{for } AG \rightarrow K \text{ don't add } k \text{ } AG \not\subset E^+ \}
 \end{aligned}$$

## Example 2

Closure of F ( $F^+$ ):  $F^+$  is the set of all FDs that can be inferred/ derived from F. Using Armstrong Axioms repeatedly on F, we can compute all the FDs.

### Example

$R(A,B,C,D,E)$  AND F:  $A \rightarrow B, B \rightarrow C, C \rightarrow D, A \rightarrow E$ . Find the closure of F

$$A^+ = \{A, B, C, D, E\}$$

$$B^+ = \{B, C, D\}$$

$$C^+ = \{C, D\}$$

$$F^+ = \{A \rightarrow A, A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, B \rightarrow B, B \rightarrow C, B \rightarrow D, C \rightarrow C, C \rightarrow D\}$$

# Finding the candidate key: Example 1

R (A,B,C,D,E,F)

F = {A  $\square$  C, C  $\square$  D, D  $\square$  B, E  $\square$  F}

Find all possible candidate keys

Step1: Find all attribute(s) not present in R.H.S

Here it is AE

Step2: Find  $A^+ = A$ ;  $E^+ = E$ ;

Check  $(AE)^+ = \{A, E, F, C, D, B\}$

AE is a candidate key

## Example 2

$R(A,B,C,D,E,F)$  WHERE F:  $A \rightarrow BC$ ,  $B \rightarrow D$ ,  $C \rightarrow DE$ ,  
 $BC \rightarrow F$ . Then, find the candidate keys of R.

### Solution

- $A^+ = \{A, B, C, D, E, F\} = \{R\} \Rightarrow A$  is a candidate key
- $B^+ = \{B, D\} \Rightarrow B$  is not a candidate key
- $C^+ = \{C, D, E\} \Rightarrow C$  is not a candidate key
- $BC^+ = \{B, C, D, E, F\} \Rightarrow BC$  is not a candidate key

# Finding the candidate key: Example 3

R (A,B,C,D,E,F,G,H)

F = {AB  $\square$  C, A  $\square$  DE, B  $\square$  F, F  $\square$  GH}

Find all possible candidate keys

Step1: Find all attribute(s) not present in R.H.S

Here it is AB

Step2: Check  $(AB)^+ = \{A, B, C, F, D, E, G, H\}$

AB is a candidate key

# Minimal Cover

- Steps to Find Minimal Cover.
  - 1) R.H.S of all FDs should contain only single attribute (Split the right-hand attributes of all FDs.)
  - 2) Remove all redundant FDs.
  - 3) Find the Extraneous attribute and remove it.

Example 1. ...

- Step 1:  $\{A \rightarrow C, AC \rightarrow D, E \rightarrow H, E \rightarrow A, E \rightarrow D\}$
- Step 2:  $\{A \rightarrow C, AC \rightarrow D, E \rightarrow H, E \rightarrow A\} \dots$
- Step 3:  $\{AC \rightarrow D\}$

# Minimal Cover

- **Minimal Cover**
- Steps to Find Minimal Cover.
  - 1) Split the right-hand attributes of all FDs.  
Example. ...
  - 2) Remove all redundant FDs. Example. ...
  - 3) Find the Extraneous attribute and remove it.  
Example. ...
- Example 1. ...
- Step 1:  $\{A \rightarrow C, AC \rightarrow D, E \rightarrow H, E \rightarrow A, E \rightarrow D\}$
- Step 2:  $\{A \rightarrow C, AC \rightarrow D, E \rightarrow H, E \rightarrow A\} \dots$
- Step 3:  $\{AC \rightarrow D\}$

# Equivalence of Sets of Functional Dependencies

- A set of functional dependencies F is said to cover another set of functional dependencies E if every FD in E is also in  $F^+$ ; that is, if every dependency in E can be inferred from F; we can say that E is covered by F.
- Two sets of functional dependencies E and F are equivalent if  $E^+ = F^+$ .
- Hence, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions E covers F and F covers E hold.

# Minimal Sets of Functional Dependencies

A set of functional dependencies F to be minimal if it satisfies the following conditions;

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency  $X \rightarrow A$  in F with a dependency  $Y \rightarrow A$ , where  $Y$  is a proper subset of  $X$ , and still have a set of dependencies that is equivalent to E.
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to E .

If several sets of FDs qualify as minimal covers of E, it is customary to use additional criteria for "minimality."

For example, we can choose the minimal set with the *smallest number of dependencies* or with the smallest *total length* (the total length of a set of dependencies is calculated by concatenating the dependencies and treating them as one long character string).

# Algorithm for Finding a Minimal Cover

Algorithm for Finding a Minimal Cover F for a Set of Functional Dependencies E

1. Set  $F := E$ .
2. Replace each functional dependency  $X \rightarrow \{A_1, A_2, \dots, A_n\}$  in F by the n functional dependencies  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$ .
3. For each functional dependency  $X \rightarrow A$  in F  
 for each attribute B that is an element of X  
 if  $\{\{F - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}\}$  is equivalent to F,  
 then replace  $X \rightarrow A$  with  $(X - \{B\}) \rightarrow A$  in F.
4. For each remaining functional dependency  $X \rightarrow A$  in F  
 if  $\{F - \{X \rightarrow A\}\}$  is equivalent to F, then remove  $X \rightarrow A$  from F.

# Lossless Join Decomposition

- Decomposition of a relation is done when a relation in relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.
- If we decompose a relation R into relations R<sub>1</sub> and R<sub>2</sub>
- Decomposition is lossy if  $R_1 \bowtie R_2 \supsetneq R$
- Decomposition is lossless if  $R_1 \bowtie R_2 = R$

# Two desirable properties of decompositions

- Decomposition-breaking up into smaller relation schemas-until it is no longer feasible or no longer desirable, based on the functional and other dependencies specified by the database designer.
- The two desirable properties of decompositions:
  1. The dependency preservation property and
  2. The lossless (or nonadditive) join property
- Both used by the design algorithms to achieve desirable decompositions.

# Lossless join decomposition conditions

- To check for lossless join decomposition using FD set, following conditions must hold:
- Union of Attributes of R<sub>1</sub> and R<sub>2</sub> must be equal to attribute of R. Each attribute of R must be either in R<sub>1</sub> or in R<sub>2</sub>.

$$\text{Att}(R_1) \cup \text{Att}(R_2) = \text{Att}(R)$$

- Intersection of Attributes of R<sub>1</sub> and R<sub>2</sub> must not be NULL.

$$\text{Att}(R_1) \cap \text{Att}(R_2) \neq \Phi$$

- Common attribute must be a key for at least one relation (R<sub>1</sub> or R<sub>2</sub>)

$$\text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow \text{Att}(R_1) \text{ or } \text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow \text{Att}(R_2)$$

# Dependency Preservation Property of a Decomposition

- The functional dependency  $X \rightarrow\!> Y$  specified in  $F$  should either appear directly in one of the relation schemas  $R_j$  in the decomposition  $D$  or could be inferred from the dependencies that appear in some  $R_i$ . This is the *dependency preservation condition*.
- We want to preserve the dependencies because each dependency in  $F$  represents a constraint on the database. If one of the dependencies is not represented in some individual relation  $R$ , of the decomposition, we cannot enforce this constraint by dealing with an individual relation;
- Instead, we have to join two or more of the relations in the decomposition and then check that the functional dependency holds in the result of the JOIN operation. This is clearly an inefficient and impractical procedure.

- If a decomposition is not dependency-preserving, some dependency is lost in the decomposition.
- To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left- and right-hand-side attributes of the lost dependency,
- Then check that the dependency holds on the result of the JOIN—an option that is not practical.

# lossless (Nonadditive) Join Property of a Decomposition

- Another property that a decomposition D should possess is the lossless join or nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations in the decomposition.

**Definition.** Formally, a decomposition  $D = \{R_1, R_2, \dots, R_m\}$  of  $R$  has the lossless (nonadditive) join property with respect to the set of dependencies  $F$  on  $R$  if, for *every* relation state  $r$  of  $R$  that satisfies  $F$ , the following holds, where  $*$  is the NATURAL JOIN of all the relations in  $D$ :

$$*(\pi_{R_1}(r), \dots, \pi_{R_m}(r)) = r$$

# Testing for Lossless (nonadditive) Join Property

## Algorithm :

- Input: A universal relation  $R$ , a decomposition  $D = \{R_1, R_2, \dots, R_m\}$  of  $R$ , and a set  $F$  of functional dependencies.
  1. Create an initial matrix  $S$  with one row  $i$  for each relation  $R_i$  in  $D$ , and one column  $j$  for each attribute  $A_j$  in  $R$ .
  2. Set  $S(i, j) := b_{ij}$  for all matrix entries.  
 (\* each  $b_{ij}$  is a distinct symbol associated with indices  $(i, j)$  \*)
  3. For each row  $i$  representing relation schema  $R_j$ 

{for each column  $j$  representing attribute  $A_j$

{if(relation  $R$ , includes attribute  $A_j$ ) then set  $S(i, j) := a_j$  ;};};  
 (\* each  $a_j$  is a distinct symbol associated with index  $(j)$  \*)

# Algorithm

## contd..

4. Repeat the following loop until a *complete loop execution* results in no changes to S
  - {for each functional dependency  $X \sqsubseteq Y$  in F
    - {for all rows in S that have the same symbols in the columns corresponding to attributes in X
      - {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an "a" symbol for the column, set the other rows to that same "a" symbol in the column. If no "a" symbol exists for the attribute in any of the rows, choose one of the "b" symbols that appears in one of the rows for the attribute and set the other rows to that same "b" symbol in the column ;};};};};
5. If a row is made up entirely of "a" symbols, then the decomposition has the lossless join property; otherwise, it does not.

- Given a relation R that is decomposed into a number of relations  $R_1, R_2, \dots, R_m$ , Algorithm begins the matrix S that we consider to be some relation state  $r$  of R.
- Row  $i$  in S represents a tuple  $t_j$  (corresponding to relation R) that has " $a$ " symbols in the columns that correspond to the attributes of  $R_j$  and " $b$ " symbols in the remaining columns.
- The algorithm then transforms the rows of this matrix (during the loop of step 4) so that they represent tuples that satisfy all the functional dependencies in F.
- At the end of step 4, any two rows in S—which represent two tuples in r—that agree in their values for the left-hand-side attributes X of a functional dependency  $X \square Y$  in F will also agree in their values for the right-hand-side attributes Y.
- It can be shown that after applying the loop of step 4, if any row in S ends up with all " $a$ " symbols, then the decomposition D has the lossless join property with respect to F.
- If, no row ends up being all " $a$ " symbols, D does not satisfy the lossless join property.

# Testing Binary Decompositions for the Nonadditive Join Property : Example

- **Binary decomposition:-** decomposition of a relation R into two relations.

Example: Decomposition of EMP\_PROJ into EMP\_PROJ1 and EMP\_LOCS fails test (matrix S does not have a row with all "a" symbols)

(a)	$R = \{SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS\}$	$D = \{R_1, R_2\}$
	$R_1 = \text{EMP\_LOCS} = \{\text{ENAME}, \text{PLOCATION}\}$	
	$R_2 = \text{EMP\_PROJ1} = \{\text{SSN}, \text{PNUMBER}, \text{HOURS}, \text{PNAME}, \text{PLOCATION}\}$	
$F = \{\text{SSN} \rightarrow \text{ENAME}; \text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}; (\text{SSN}, \text{PNUMBER}) \rightarrow \text{HOURS}\}$		

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
$R_1$	$b_{11}$	$a_2$	$b_{13}$	$b_{14}$	$a_5$	$b_{16}$
$R_2$	$a_1$	$b_{22}$	$a_3$	$a_4$	$a_5$	$a_6$

(no changes to matrix after applying functional dependencies)

# Decomposition of EMP\_PROJ (into EMP, PROJECT, and WORKS\_ON) has lossless join property.

(b)

EMP		PROJECT			WORKS_ON		
SSN	ENAME	PNUMBER	PNAME	PLOCATION	SSN	PNUMBER	HOURS

- (c)  $R = \{SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS\}$   $D = \{R_1, R_2, R_3\}$
- $R_1 = EMP = \{SSN, ENAME\}$
- $R_2 = PROJ = \{PNUMBER, PNAME, PLOCATION\}$
- $R_3 = WORKS\_ON = \{SSN, PNUMBER, HOURS\}$
- $F = \{SSN \rightarrow \{ENAME\}; PNUMBER \rightarrow \{PNAME, PLOCATION\}; \{SSN, PNUMBER\} \rightarrow HOURS\}$

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
$R_1$	a <sub>1</sub>	a <sub>2</sub>	b <sub>13</sub>	b <sub>14</sub>	b <sub>15</sub>	b <sub>16</sub>
$R_2$	b <sub>21</sub>	b <sub>22</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	b <sub>26</sub>
$R_3$	a <sub>1</sub>	b <sub>32</sub>	a <sub>3</sub>	b <sub>34</sub>	b <sub>35</sub>	a <sub>6</sub>

(original matrix S at start of algorithm)

- Decomposition of EMP\_PROJ into EMP, PROJECT, and WORKS\_ON satisfies test (3<sup>rd</sup> row consists only of "a" symbols)

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
$R_1$	a <sub>1</sub>	a <sub>2</sub>	b <sub>13</sub>	b <sub>14</sub>	b <sub>15</sub>	b <sub>16</sub>
$R_2$	b <sub>21</sub>	b <sub>22</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	b <sub>26</sub>
$R_3$	a <sub>1</sub>	b <sub>32</sub> <sup>a</sup> <sub>2</sub>	a <sub>3</sub>	b <sub>34</sub> <sup>a</sup> <sub>4</sub>	b <sub>35</sub> <sup>a</sup> <sub>5</sub>	a <sub>6</sub>

(matrix S after applying the first two functional dependencies - last row is all "a" symbols, so we stop)



# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module III**  
RELATIONAL DATABASE DESIGN

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

# Syllabus Module 3

## Module 3: RELATIONAL DATABASE DESIGN

- **Relational Database Design:** Different anomalies in designing a database, normalization, functional dependency (FD), Armstrong's Axioms, closures, Equivalence of FDs, minimal Cover (proofs not required). Normalization using functional dependencies, INF, 2NF, 3NF and BCNF, lossless and dependency preserving decompositions
- (Reading: Elmasri and Navathe, Ch. 14.1-14.5, 15.1-15.2.
- Additional Reading: Silberschatz, Korth Ch. 8.1-8.5)

# RELATIONAL DATABASE DESIGN

Normalization, Functional dependency (FD),  
Armstrong's Axioms

- What is relational database design?
  - The grouping of attributes to form "good" relation schemas
- Two levels of relation schemas
  - The logical "user view" level
  - The storage "base relation" level
- Design is concerned mainly with base relations
- What are the criteria for "good" base relations?

# Informal Design Guidelines for Relational Databases

- We first discuss informal guidelines for good relational design
- Then we discuss formal concepts of functional dependencies and normal forms
  - - 1NF (First Normal Form)
  - - 2NF (Second Normal Form)
  - - 3NF (Third Normal Form)
  - - BCNF (Boyce-Codd Normal Form)

The grouping of attributes to form “good” relation schemas.

There are 4 measures of quality for a relation schema design

- i. **Semantics of the attributes**
- ii. **Reducing the redundant values in tuples and Update Anomalies**
- iii. **Reducing null values in tuples**
- iv. **Disallowing generation of spurious tuples**

# 1.1 Semantics of the Relational Attributes must be clear

- GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).
  - Attributes of different entities (EMPLOYEES, DEPARTMENTS, PROJECTS) should not be mixed in the same relation
  - Only foreign keys should be used to refer to other entities
  - Entity and relationship attributes should be kept apart as much as possible.
- Bottom Line: *Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.*

# A simplified COMPANY relational database schema

## EMPLOYEE

Ename	<u>Ssn</u>	Bdate	Address	Dnumber
-------	------------	-------	---------	---------

P.K.

## DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn
-------	----------------	----------

P.K.

## DEPT\_LOCATIONS

F.K.

<u>Dnumber</u>	Dlocation
----------------	-----------

P.K.

## PROJECT

F.K.

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

P.K.

## WORKS\_ON

F.K.

F.K.

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------

P.K.

A simplified COMPANY  
relational database schema.

# 1.2 Redundant Information in Tuples and Update Anomalies

- Information is stored redundantly
  - Wastes storage
  - Causes problems with update anomalies
    - Insertion anomalies
    - Deletion anomalies
    - Modification anomalies

# EXAMPLE OF AN UPDATE ANOMALY

- Consider the relation:
  - EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours)
- Update Anomaly:
  - Changing the name of project number P1 from “Billing” to “Customer-Accounting” may cause this update to be made for all 100 employees working on project P1.

# EXAMPLE OF AN INSERT ANOMALY

- Consider the relation:
  - EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours)
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it.
- Conversely
  - Cannot insert an employee unless he/she is assigned to a project.

# EXAMPLE OF A DELETE ANOMALY

- Consider the relation:
  - EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours)
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.
  - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

# Two relation schemas suffering from update anomalies

Two relation schemas suffering from update anomalies.  
 (a) EMP\_DEPT and  
 (b) EMP\_PROJ.

(a)

**EMP\_DEPT**

Ename	<u>Ssn</u>	Bdate	Address	Dnumber	Dname	Dmgr_ssn

(b)

**EMP\_PROJ**

<u>Ssn</u>	Pnumber	Hours	Ename	Pname	Plocation
FD1					
FD2					
FD3					

$F = \{ SSN \rightarrow (ENAME, ADDRESS, DOB, DNO), DNO \rightarrow \{DNAME, MGR\} \}$   
 we can infer the following additional functional dependencies from F such as  
 $SSN \rightarrow \{DNAME, MGR\}, DNO \rightarrow DNAME$

# Sample states for EMP\_DEPT and EMP\_PROJ

Sample states for EMP\_DEPT and EMP\_PROJ resulting from applying NATURAL JOIN to the relations shown in Figure. These may be stored as base relations for performance reasons.

EMP_DEPT							Redundancy
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn	
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555	
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555	
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321	
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321	
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555	
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555	
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321	
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555	

EMP_PROJ						Redundancy	Redundancy
Ssn	Pnumber	Hours	Ename	Pname	Plocation		
123456789	1	32.5	Smith, John B.	ProductX	Bellaire		
123456789	2	7.5	Smith, John B.	ProductY	Sugarland		
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston		
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire		
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland		
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland		
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston		
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford		
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston		
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford		
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford		
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford		
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford		
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford		
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston		
888665555	20	Null	Borg, James E.	Reorganization	Houston		

# Guideline for Redundant Information in Tuples and Update Anomalies

- **GUIDELINE 2:**
  - Design a schema that does not suffer from the insertion, deletion and update anomalies.
  - If there are any anomalies present, then note them so that applications can be made to take them into account.

# 1.3 Reducing Null Values in Tuples

- **GUIDELINE 3:**
  - Relations should be designed such that their tuples will have as few NULL values as possible
  - Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- Reasons for nulls:
  - Attribute not applicable or invalid
  - Attribute value unknown (may exist)
  - Value known to exist, but unavailable

# 1.4 Generation of Spurious Tuples – avoid at any cost

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations
- **GUIDELINE 4:**
  - The relations should be designed to satisfy the lossless join condition.
  - No spurious tuples should be generated by doing a natural-join of any relations.

# Spurious Tuples (2)

- There are two important properties of decompositions:
  - a) Non-additive or losslessness of the corresponding join
  - b) Preservation of the functional dependencies.
- Note that:
  - Property (a) is extremely important and cannot be sacrificed.
  - Property (b) is less stringent and may be sacrificed. (See Chapter 15).

## 2. Functional Dependencies

- Functional dependencies (FDs)
  - Are used to specify *formal measures* of the "goodness" of relational designs
  - And keys are used to define **normal forms** for relations
  - Are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A functional dependency is a constraint that specifies the relationship between two sets of attributes where one set can accurately determine the value of other sets.
- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y.
- It is denoted as  $X \rightarrow Y$ , where X is a set of attributes that is capable of determining the value of Y. The attribute set on the left side of the arrow, X is called **Determinant**, while on the right side, Y is called the **Dependent**.

## 2.1 Defining Functional Dependencies

- $X \rightarrow Y$  holds if whenever two tuples have the same value for  $X$ , they *must have* the same value for  $Y$ 
  - For any two tuples  $t_1$  and  $t_2$  in any relation instance  $r(R)$ :  
If  $t_1[X]=t_2[X]$ , then  $t_1[Y]=t_2[Y]$
- $X \rightarrow Y$  in  $R$  specifies a *constraint* on all relation instances  $r(R)$
- Written as  $X \rightarrow Y$ ; can be displayed graphically on a relation schema as in Figures. ( denoted by the arrow: ).
- FDs are derived from the real-world constraints on the attributes

# Examples of FD constraints (1)

- Social security number determines employee name

$\text{SSN} \rightarrow \text{ENAME}$

- Project number determines project name and location

$\text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}$

- Employee ssn and project number determines the hours per week that the employee works on the project

$\{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{HOURS}$

# Examples of FD constraints (2)

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every* relation instance  $r(R)$
- If K is a key of R, then K functionally determines all attributes in R
  - (since we never have two distinct tuples with  $t_1[K]=t_2[K]$ )

- Given a set of Functional Dependencies (F), we can infer additional Functional Dependencies that hold whenever the FD's in F hold
- Example:- From a relation schema given below, we say that the set of functional dependencies is

EMP_DEPT						
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn



- $F = \{ \text{SSN} \sqcup (\text{ENAME}, \text{ADDRESS}, \text{DOB}), \text{DNO} \sqcap \{\text{DNAME}, \text{MGR}\} \}$
- we can infer the following additional functional dependencies from F such as
- $\text{SSN} \sqcap \{\text{DNAME}, \text{MGR}\}, \text{DNO} \sqcap \text{DNAME}$

# Defining FDs from instances

- Note that in order to define the FDs, we need to understand the meaning of the attributes involved and the relationship between them.
- An FD is a property of the attributes in the schema R
- Given the instance (population) of a relation, all we can conclude is that an FD *may exist* between certain attributes.
- What we can definitely conclude is – that certain FDs *do not exist* because there are tuples that show a violation of those dependencies.

# Example

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	JKL	ME	B2

# Valid functional dependencies: Example

- $\text{roll\_no} \rightarrow \{\text{name}, \text{dept\_name}, \text{dept\_building}\}$ ,  
→ Here, roll\_no can determine values of fields name, dept\_name and dept\_building, hence a valid Functional dependency
- $\text{roll\_no} \rightarrow \text{dept\_name}$ , Since, roll\_no can determine whole set of  $\{\text{name}, \text{dept\_name}, \text{dept\_building}\}$ , it can determine its subset dept\_name also.
- $\text{dept\_name} \rightarrow \text{dept\_building}$ , Dept\_name can identify the dept\_building accurately, since departments with different dept\_name will also have a different dept\_building
- More valid functional dependencies:  $\text{roll\_no} \rightarrow \text{name}$ ,  $\{\text{roll\_no}, \text{name}\} \rightarrow \{\text{dept\_name}, \text{dept\_building}\}$ , etc.

## Some invalid functional dependencies: Example

- $\text{name} \rightarrow \text{dept\_name}$  Students with the same name can have different dept\_name, hence this is not a valid functional dependency.
- $\text{dept\_building} \rightarrow \text{dept\_name}$  There can be multiple departments in the same building, For example, in the above table departments ME and EC are in the same building B2, hence  $\text{dept\_building} \rightarrow \text{dept\_name}$  is an invalid functional dependency.
- More invalid functional dependencies:  $\text{name} \rightarrow \text{roll\_no}$ ,  $\{\text{name}, \text{dept\_name}\} \rightarrow \text{roll\_no}$ ,  $\text{dept\_building} \rightarrow \text{roll\_no}$ , etc.

# Ruling Out FDs

Note that given the state of the TEACH relation, we can say that the FD: Text → Course may exist. However, the FDs Teacher → Course, Teacher → Text and Course → Text are ruled out.

## TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

# What FDs may exist?

- A relation  $R(A, B, C, D)$  with its extension.
- Which FDs may exist in this relation?

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

# Armstrong's inference rules

## 1. IR1. (Reflexive rule)

- If  $Y$  is a *subset-of*  $X$  ( $Y \subseteq X$ ), then  $X \sqsupseteq Y$
- [states that a set of attributes determines itself or any of its subsets. Since IR1 generates dependencies that are always true, such dependencies are called Trivial otherwise it is called Non-trivial]

For example,  $\{roll\_no, name\} \rightarrow name$  is valid.

## 2. IR2. (Augmentation rule)

- If  $X \sqsupseteq Y$ , then  $XZ \sqsupseteq YZ$  (Notation:  $XZ$  stands for  $X \cup Z$ ) ( Adding same set of attributes to both sides results in valid dependency)

## 3. IR3. (Transitive rule)

If  $X \sqsupseteq Y$  and  $Y \sqsupseteq Z$ , then  $X \sqsupseteq Z$

## Armstrong's axioms/properties of functional dependencies:

### 4. IR4. (Decomposition/projective rule)

If  $X \sqsubseteq YZ$ , then  $X \sqsubseteq Y$  and  $X \sqsubseteq Z$

### 5. IR5. (Union/additive rule)

If  $X \sqsubseteq Y$  and  $X \sqsubseteq Z$ , then  $X \sqsubseteq YZ$

### 6. IR6. (Psuedotransitivity rule)

If  $X \sqsubseteq Y$  and  $WY \sqsubseteq Z$ , then  $WX \sqsubseteq Z$

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules. IR4, IR5, IR6 inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)

# Types of Functional dependencies in DBMS:

- Trivial functional dependency
  - A dependent is always a subset of the determinant.  
i.e. If  $X \rightarrow Y$  and **Y is the subset of X**, then it is called trivial functional dependency.
- Non-Trivial functional dependency
  - The dependent is strictly not a subset of the determinant.
- Multivalued functional dependency
  - Entities of the dependent set are **not dependent on each other**.  
i.e. If  $a \rightarrow \{b, c\}$  and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency**.
- Transitive functional dependency

Dependent is indirectly dependent on determinant.  
i.e. If  $a \rightarrow b$  &  $b \rightarrow c$ , then according to axiom of transitivity,  $a \rightarrow c$ .

# Closures

- **Closure** of a set  $F$  of FDs is the set  $F^+$  of all FDs that can be inferred from  $F$ . Here  $F \subseteq F^+$ .
- **Closure** of a set of attributes  $X$  with respect to  $F$  is the set  $X^+$  of all attributes that are functionally determined by  $X$ .  $X^+$  can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in  $F$

# Algorithm for determining $X^+$

## **Algorithm** for determining the closure of $X$ under $F$ ( $X^+$ )

- Let us assume that  $X^+$  equals all the attributes in  $X$ ; By IR1 all attributes are functionally dependent on  $X$ , Using IR2 and IR3 we add attributes to  $X^+$  using each functional dependency in  $F$

**begin function**

**$X^+ = X;$**

**while true do**

**if there exists  $(Y \square Z)$  in  $F$  such that  $X^+ \supseteq Y$  and  $X^+ \neq Z$**

**then  $X^+ = X^+ \cup Z;$**

**else exit;**

**return  $X^+$**

**end**

# Closures: Example

Example:-  $F = \{SSN \rightarrow ENAME, PNO \rightarrow \{PNAME, PLOC\}, \{SSN, PNO\} \rightarrow Hours\}$

- By the algorithm we get the closure sets with respect to F
- $\{SSN\}^+ = \{SSN, ENAME\}$
- $\{PNO\}^+ = \{PNO, PNAME, PLOC\}$
- $\{SSN, PNO\}^+ = \{SSN, PNO, ENAME, PNAME, PLOC, HOURS\}$
- A **primary key** can be defined in terms of functional dependence: A primary key of one or more attributes has the following properties:
  1. All the attributes that are not part of the key are functionally dependent on the key attribute or attributes.
  2. If the key contains more than one attribute then no subset of the key attributes has property 1.

# Minimal Set of Dependencies

A set of functional dependencies  $F$  is minimal if it satisfies the following conditions.

1. Every dependency in  $F$  has a single attribute for its right hand side.
2. A dependency  $X \square A$  in  $F$  cannot have a dependency  $Y \square A$  where  $Y$  is a subset of  $X$ .
3. A dependency in  $F$  cannot be removed to get a set of dependencies that is equivalent to  $F$ .

# Normal Forms

- **Normalization:** The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.
- **Normal form:** Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form.

## 3.1 Normalization of Relations

3.2 Practical Use of Normal Forms

3.3 Definitions of Keys and Attributes    Participating  
in Keys

3.4 First Normal Form

3.5 Second Normal Form

3.6 Third Normal Form

## 3.1 Normalization of Relations

- 2NF, 3NF, BCNF
  - based on keys and FDs of a relation schema
- 4NF
  - based on keys, multi-valued dependencies : MVDs;
- 5NF
  - based on keys, join dependencies : JDS
- Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation; see Chapter 15)

## 3.2 Practical Use of Normal Forms

- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties.
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are *hard to understand* or to *detect*.
- The database designers *need not* normalize to the highest possible normal form.
  - (usually up to 3NF and BCNF. 4NF rarely used in practice.)
- **Denormalization:**
  - The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form.

### 3.3 Definitions of Keys and Attributes Participating in Keys (1)

- A **superkey** of a relation schema  $R = \{A_1, A_2, \dots, A_n\}$  is a set of attributes  $S$  *subset-of*  $R$  with the property that no two tuples  $t_1$  and  $t_2$  in any legal relation state  $r$  of  $R$  will have  $t_1[S] = t_2[S]$
- A **key**  $K$  is a **superkey** with the *additional property* that removal of any attribute from  $K$  will cause  $K$  not to be a superkey any more.

# Definitions of Keys and Attributes

## Participating in Keys (2)

- If a relation schema has more than one key, each is called a **candidate** key.
  - One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called **secondary keys**.
- A **Prime attribute** must be a member of *some* candidate key
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

## 3.4 First Normal Form

- Disallows
  - composite attributes
  - multivalued attributes
  - **nested relations**; attributes whose values for an *individual tuple* are non-atomic
- Considered to be part of the definition of a relation
- Informally- to transform the structure to 1NF, we must remove repeating groups
- Most RDBMSs allow only those relations to be defined that are in First Normal Form

# Normalization into 1NF

(a)

## DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations



Normalization into 1NF.  
 A relation schema that is  
 not in 1NF.

(b) Sample state of relation  
**DEPARTMENT**.  
 1NF version of the same  
 relation with redundancy.

(b)

## DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

(c)

## DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

# Normalizing nested relations into 1NF

(a)

EMP\_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP\_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
		1	20.0
453453453	English, Joyce A.	2	20.0
		1	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
		30	30.0
999887777	Zelaya, Alicia J.	10	10.0
		30	35.0
987987987	Jabbar, Ahmad V.	30	5.0
		10	20.0
987654321	Wallace, Jennifer S.	20	15.0
		30	NULL
888665555	Borg, James E.	20	

(c)

EMP\_PROJ1

Ssn	Ename

EMP\_PROJ2

Ssn	Pnumber	Hours

The technique is to decompose the relation into **2** - EMP\_PROJ1 with attributes SSN and ENAME and the second EMP\_PROJ2 with attributes SSN, PNO and HRS. (Here the **primary key is propagated** – EMP\_PROJ1 has PK SSN while EMP\_PROJ2 has PK SSN & PNO).

Normalizing nested relations into 1NF. (a) Schema of the EMP\_PROJ relation with a nested relation attribute PROJS. (b) Sample extension of the EMP\_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP\_PROJ into relations EMP\_PROJ1 and EMP\_PROJ2 by propagating the primary key.

# Three main techniques to achieve first normal form

1. Remove the attribute that violates 1NF and place it in a separate relation along with the primary key.
2. Expand the key so that there will be a separate tuple in the original relation
3. If a *maximum number of values* is known for the attribute-replace them with atomic attributes.
  - for example, if it is known that *at most three locations* can exist for a department-replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2, and DLOCATION3.
  - This solution has the disadvantage of introducing *null values* if most departments have fewer than three locations. It further introduces a spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult;
  - for example, consider how you would write the query: "List the departments that have "Bellaire" as one of their locations" in this design.

## 3.5 Second Normal Form (1)

- Uses the concepts of **FDs, primary key**
- Definitions
  - **Prime attribute:** An attribute that is member of the primary key K
  - **Full functional dependency:** a FD  $Y \rightarrow Z$  where removal of any attribute from Y means the FD does not hold any more
- Examples:
  - $\{\text{SSN, PNUMBER}\} \rightarrow \text{HOURS}$  is a full FD since neither  $\text{SSN} \rightarrow \text{HOURS}$  nor  $\text{PNUMBER} \rightarrow \text{HOURS}$  hold
  - $\{\text{SSN, PNUMBER}\} \rightarrow \text{ENAME}$  is not a full FD (it is called a partial dependency) since  $\text{SSN} \rightarrow \text{ENAME}$  also holds

# Second Normal Form (2)

## Definition

- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key
- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all
- R can be decomposed into 2NF relations via the process of 2NF normalization or “second normalization”. That is split into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

# Normalizing into 2NF

Normalizing EMP\_PROJ into 2NF relations.

**EMP\_PROJ**

Ssn	Pnumber	Hours	Ename	Pname	Plocation
FD1			↑		
FD2				↑	
FD3					↑

2NF Normalization

**EP1**

Ssn	Pnumber	Hours
FD1		↑

**EP2**

Ssn	Ename
FD2	↑

**EP3**

Pnumber	Pname	Plocation
FD3		↑

- The EMP\_PROJ relation shown in Figure is in INF but is not in 2NF.
- The nonprime attribute ENAME violates 2NF because of FD2,
- as do the nonprime attributes PNAME and PLOCATION because of FD3.
- The functional dependencies FD2 and FD3 make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP\_PROJ, thus violating the 2NF test.

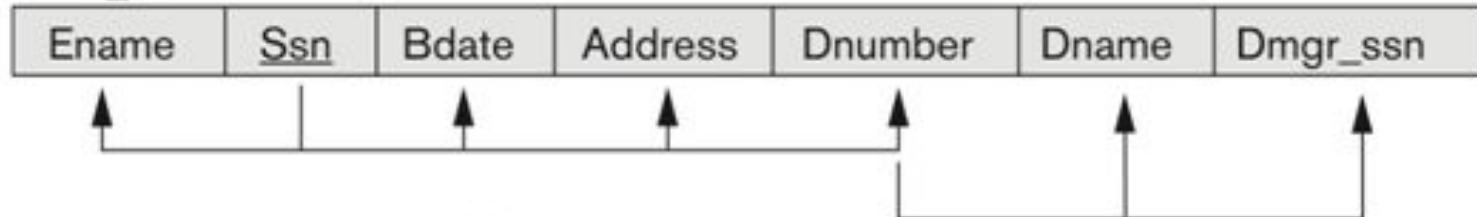
## 3.6 Third Normal Form (1)

- based on the concept of *transitive dependency*
- A functional dependency  $X \rightarrow Y$  in a relation schema R is a transitive dependency if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both  $X \rightarrow Z$  and  $Z \rightarrow Y$  hold.
- **Transitive functional dependency:** a FD  $X \rightarrow Z$  that can be derived from two FDs,  $X \rightarrow Y$  and  $Y \rightarrow Z$
- The dependency  $SSN \rightarrow DMGRSSN$  is transitive through DNUMBER in EMP\_DEPT because both the dependencies  $SSN \rightarrow DNUMBER$  and  $DNUMBER \rightarrow DMGRSSN$  hold and DNUMBER is neither a key itself nor a subset of the key of EMP\_DEPT.
- The dependency of DMGRSSN on DNUMBER is undesirable in EMP\_DEPT since DNUMBER is not a key of EMP\_DEPT.

- Examples:
  - $\text{SSN} \rightarrow \text{DMGRSSN}$  is a **transitive FD**
    - Since  $\text{SSN} \rightarrow \text{DNUMBER}$  and  $\text{DNUMBER} \rightarrow \text{DMGRSSN}$  hold
  - $\text{SSN} \rightarrow \text{ENAME}$  is **non-transitive**
    - Since there is no set of attributes X where  $\text{SSN} \rightarrow X$  and  $X \rightarrow \text{ENAME}$

**EMP\_DEPT**

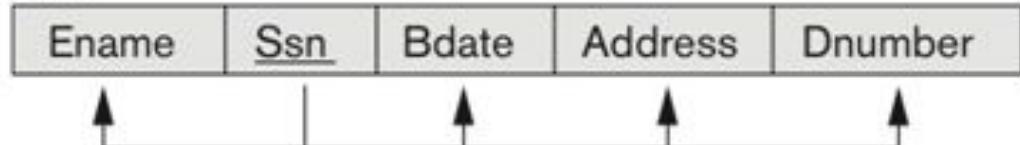
Ename	<u>Ssn</u>	Bdate	Address	Dnumber	Dname	Dmgr_ss



3NF Normalization

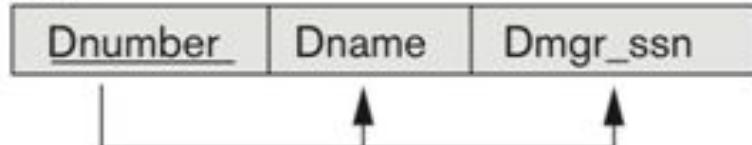
**ED1**

Ename	<u>Ssn</u>	Bdate	Address	Dnumber



**ED2**

<u>Dnumber</u>	Dname	Dmgr_ss



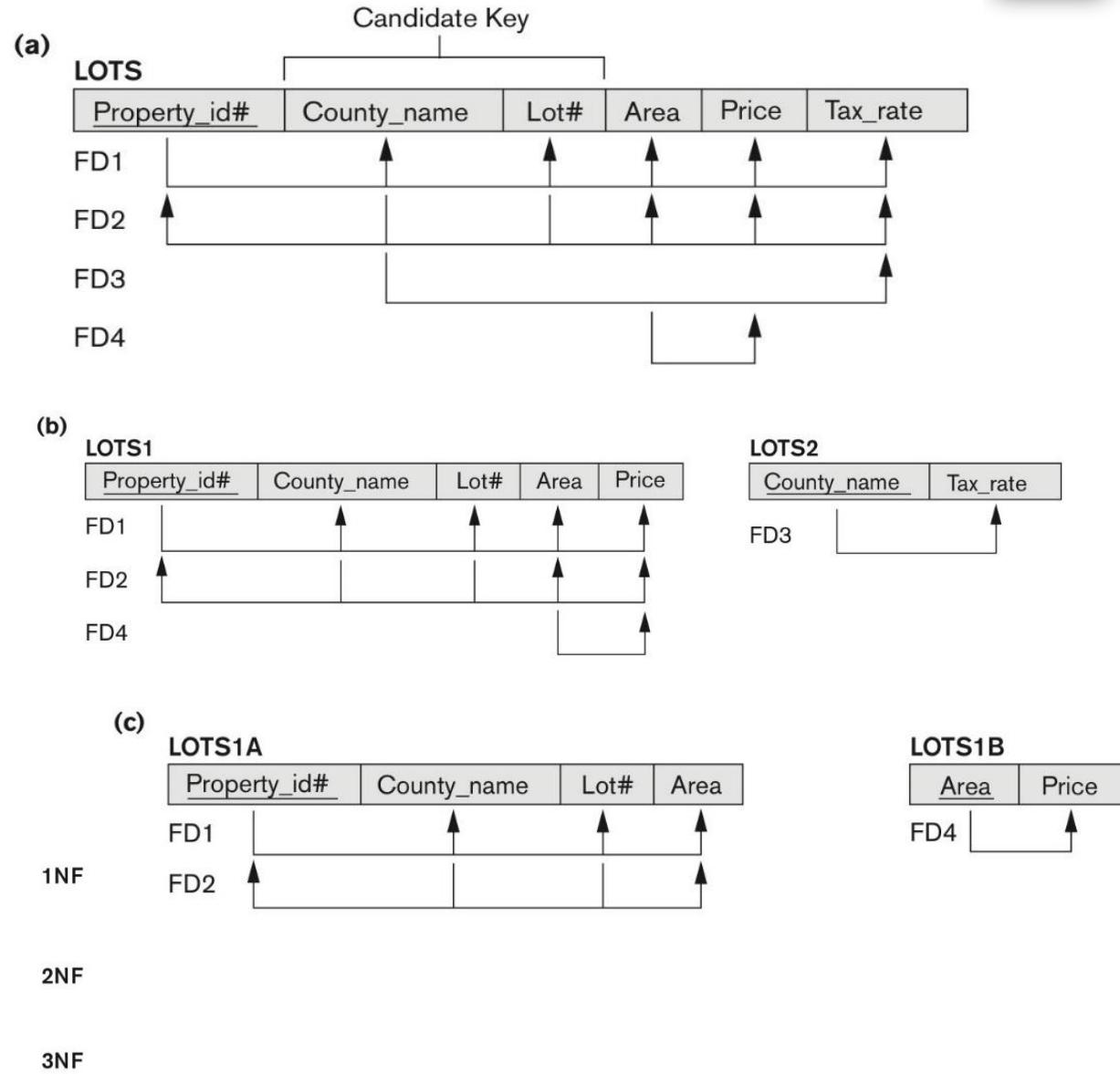
# Third Normal Form (2)

Definition:

- A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
- R can be decomposed into 3NF relations via the process of 3NF normalization
- NOTE:
  - In  $X \rightarrow Y$  and  $Y \rightarrow Z$ , with X as the primary key, we consider this a problem only if Y is not a candidate key.
  - When Y is a candidate key, there is no problem with the transitive dependency .
  - E.g., Consider EMP (SSN, Emp#, Salary ).
    - Here,  $SSN \rightarrow Emp\# \rightarrow Salary$  and Emp# is a candidate key.

# Normalization into 2NF and 3NF

Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4.  
 (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B.  
 (d) Progressive normalization of LOTS into a 3NF design.



# General Definition of 2NF (For Multiple Candidate Keys)

- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on *every* key of R
- In the previous Figure the FD 3  
 $\text{County\_name} \rightarrow \text{Tax\_rate}$  violates 2NF.  
because TAX\_RATE is partially dependent on the candidate key {COUNTY\_NAME, LOT#}, due to FD3

So second normalization converts LOTS into  
LOTS1 (Property\_id#, County\_name, Lot#, Area, Price)  
LOTS2 ( County\_name, Tax\_rate)  
FD4 does not violate 2NF and is carried over to LOTS1.

## 4.2 General Definition of Third Normal Form

- Definition:
  - **Superkey** of relation schema R - a set of attributes S of R that contains a key of R
  - A relation schema R is in **third normal form (3NF)** if whenever a nontrivial FD  $X \rightarrow A$  holds in R, then either:
    - (a) X is a superkey of R, or
    - (b) A is a prime attribute of R
- According to this definition, LOTS2 is in 3NF.  
LOTS1 relation violates 3NF because FD<sub>4</sub> Area → Price ;
- ✓ PRICE is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute AREA
- ✓ As well as PRICE is not a prime attribute and Area is not a superkey in LOTS1.
- To normalize LOTS1 into 3NF, decompose it into the relation schemas LOTS1A and LOTS1B

# Normal Forms Defined Informally

- 1<sup>st</sup> normal form
  - All attributes depend on **the key**
- 2<sup>nd</sup> normal form
  - All attributes depend on **the whole key**
- 3<sup>rd</sup> normal form
  - All attributes depend on **nothing but the key (non-prime attributes doesn't have transitive dependency)**

## 4. General Normal Form Definitions (For Multiple Keys) (1)

- The above definitions consider the primary key only
- The following more general definitions take into account relations with multiple candidate keys
- Any attribute involved in a candidate key is a prime attribute
- All other attributes are called non-prime attributes.

## 4.3 Interpreting the General Definition of Third Normal Form

- Consider the 2 conditions in the Definition of 3NF:  
A relation schema R is in **third normal form (3NF)** if whenever a FD  $X \rightarrow A$  holds in R, then either:
  - (a) X is a superkey of R, or
  - (b) A is a prime attribute of R
- Condition (a) catches two types of violations :
  - one where a prime attribute functionally determines a non-prime attribute. This catches 2NF violations due to non-full functional dependencies.
  - second, where a non-prime attribute functionally determines a non-prime attribute. This catches 3NF violations due to a transitive dependency.

## 4.3 Interpreting the General Definition of Third Normal Form (2)

- **ALTERNATIVE DEFINITION of 3NF:** We can restate the definition as:

A relation schema R is in **third normal form (3NF)** if every non-prime attribute in R meets both of these conditions:

- It is fully functionally dependent on every key of R
- It is non-transitively dependent on every key of R

Note that stated this way, a relation in 3NF also meets the requirements for 2NF.

- The condition (b) from the last slide takes care of the dependencies that “slip through” (are allowable to) 3NF but are “caught by” BCNF which we discuss next.

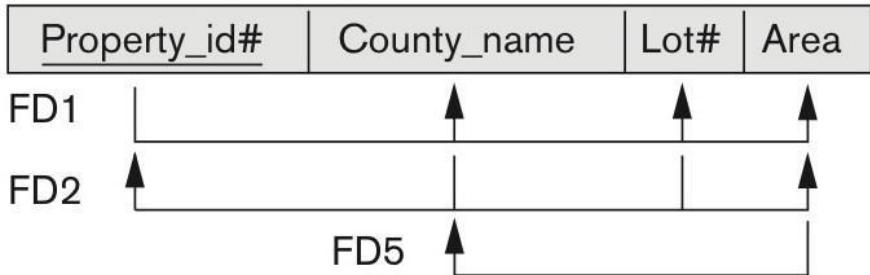
## 5. BCNF (Boyce-Codd Normal Form)

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an FD  $X \rightarrow A$  holds in R, then **X is a superkey** of R
- *A relation is in BCNF, if and only if, every determinant is a Form (BCNF) candidate key.*
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- Hence BCNF is considered a **stronger form** of 3NF
- The goal is to have each relation in BCNF (or 3NF)

# Boyce-Codd normal form

(a)

**LOTS1A**



BCNF Normalization

**LOTS1AX**

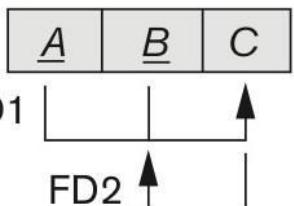
<u>Property_id#</u>	Area	Lot#
---------------------	------	------

**LOTS1AY**

Area	County_name
------	-------------

(b)

**R**



Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d.  $C \rightarrow B$ .

# Achieving the BCNF by Decomposition (1)

- Two FDs exist in the relation TEACH:
  - fd<sub>1</sub>: { student, course } → instructor
  - fd<sub>2</sub>: instructor → course
- {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.13 (b).
  - So this relation is in 3NF *but not in* BCNF
- A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations.

# A relation TEACH that is in 3NF but not in BCNF

FD1:{Student,Course} ⊓ {Instructor}  
 FD2: Instructor ⊓ Course

## TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

# Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
  - D1: {student, instructor} and {student, course}
  - D2: {course, instructor} and {course, student}
  - D3: {instructor, course} and {instructor, student} ✓
- All three decompositions will lose fd1.
  - We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).
- A test to determine whether a binary decomposition (decomposition into two relations) is non-additive (lossless) is discussed under Property NJB on the next slide. We then show how the third decomposition above meets the property.

# Test for checking non-additivity of Binary Relational Decompositions

- **Testing Binary Decompositions for Lossless Join (Non-additive Join) Property**
  - **Binary Decomposition:** Decomposition of a relation R into two relations.
  - **PROPERTY NJB (non-additive join test for binary decompositions):** A decomposition  $D = \{R_1, R_2\}$  of R has the lossless join property with respect to a set of functional dependencies F on R if and only if either
    - The f.d.  $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$  is in  $F^+$ , or
    - The f.d.  $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$  is in  $F^+$ .

# Test for checking non-additivity of Binary Relational Decompositions

If you apply the NJB test to the 3 decompositions of the TEACH relation:

- D1 gives **Student** → Instructor or **Student** → Course, none of which is true.
- D2 gives **Course** → Instructor or **Course** → Student, none of which is true.
- However, in D3 we get **Instructor** → Course or **Instructor** → Student.

Since **Instructor** → Course is indeed true, the NJB property is satisfied and D3 is determined as a non-additive (good) decomposition.

## Here we make use the algorithm from Chapter 15 (Algorithm 15.5):

- Let R be the relation not in BCNF, let X be a subset-of R, and let  $X \rightarrow A$  be the FD that causes a violation of BCNF. Then R may be decomposed into two relations:
- (i)  $R - A$  and (ii)  $X \cup A$ .
- If either  $R - A$  or  $X \cup A$ . is not in BCNF, repeat the process.

Note that the f.d. that violated BCNF in TEACH was Instructor →Course. Hence its BCNF decomposition would be :

(TEACH – COURSE) and (Instructor  $\cup$  Course), which gives the relations: (Instructor, Student) and (Instructor, Course) that we obtained before in decomposition D3.

# 5. Multivalued Dependencies and Fourth Normal Form (1)

## Definition:

- A **multivalued dependency (MVD)**  $X \rightarrow\!\!\!> Y$  specified on relation schema  $R$ , where  $X$  and  $Y$  are both subsets of  $R$ , specifies the following constraint on any relation state  $r$  of  $R$ : If two tuples  $t_1$  and  $t_2$  exist in  $r$  such that  $t_1[X] = t_2[X]$ , then two tuples  $t_3$  and  $t_4$  should also exist in  $r$  with the following properties, where we use  $Z$  to denote  $(R \setminus (X \cup Y))$ :
  - $t_3[X] = t_4[X] = t_1[X] = t_2[X]$ .
  - $t_3[Y] = t_1[Y]$  and  $t_4[Y] = t_2[Y]$ .
  - $t_3[Z] = t_2[Z]$  and  $t_4[Z] = t_1[Z]$ .
- An MVD  $X \rightarrow\!\!\!> Y$  in  $R$  is called a **trivial MVD** if (a)  $Y$  is a subset of  $X$ , or (b)  $X \cup Y = R$ .

# Multivalued Dependencies and Fourth Normal Form

## (3)

### Definition:

- A relation schema  $R$  is in **4NF** with respect to a set of dependencies  $F$  (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency  $X \rightarrow\!\!\!> Y$  in  $F^+$ ,  $X$  is a superkey for  $R$ .
  - Note:  $F^+$  is the (complete) set of all dependencies (functional or multivalued) that will hold in every relation state  $r$  of  $R$  that satisfies  $F$ . It is also called the **closure** of  $F$ .

# Fourth and fifth normal forms.

(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(c) SUPPLY

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(b) EMP\_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP\_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

(d)  $R_1$

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

$R_2$

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

$R_3$

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Fourth and fifth normal forms. (a) The EMP relation with two MVDs: Ename  $\rightarrow\!\!\!>$  Pname and Ename  $\rightarrow\!\!\!>$  Dname. (b) Decomposing the EMP relation into two 4NF relations EMP\_PROJECTS and EMP\_DEPENDENTS. (c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD( $R_1, R_2, R_3$ ). (d) Decomposing the relation SUPPLY into the 5NF relations  $R_1, R_2, R_3$ .

# 6. Join Dependencies and Fifth Normal Form

## Definition:

- A **join dependency (JD)**, denoted by  $\text{JD}(R_1, R_2, \dots, R_n)$ , specified on relation schema  $R$ , specifies a constraint on the states  $r$  of  $R$ .
  - The constraint states that every legal state  $r$  of  $R$  should have a non-additive join decomposition into  $R_1, R_2, \dots, R_n$ ; that is, for every such  $r$  we have
    - ${}^*(\pi_{R_1}(r), \pi_{R_2}(r), \dots, \pi_{R_n}(r)) = r$

**Note:** an MVD is a special case of a JD where  $n = 2$ .

- A join dependency  $\text{JD}(R_1, R_2, \dots, R_n)$ , specified on relation schema  $R$ , is a **trivial JD** if one of the relation schemas  $R_i$  in  $\text{JD}(R_1, R_2, \dots, R_n)$  is equal to  $R$ .

# Join Dependencies and Fifth Normal Form (2)

## Definition:

- A relation schema  $R$  is in **fifth normal form (5NF)** (or **Project-Join Normal Form (PJNF)**) with respect to a set  $F$  of functional, multivalued, and join dependencies if,
  - for every nontrivial join dependency  $\text{JD}(R_1, R_2, \dots, R_n) \in F^+$  (that is, implied by  $F$ ),
    - every  $R_i$  is a superkey of  $R$ .
- Discovering join dependencies in practical databases with hundreds of relations is next to impossible. Therefore, 5NF is rarely used in practice.

Normal Form	Description
<u>1NF</u>	A relation is in 1NF if it contains an atomic value.
<u>2NF</u>	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
<u>3NF</u>	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.
<u>4NF</u>	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.
<u>5NF</u>	A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless.

# lossless and dependency preserving decompositions

- Decomposition of a relation is done when a relation in relational model is not in appropriate normal form.
- Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

## Lossless Join Decomposition

- If we decompose a relation R into relations R<sub>1</sub> and R<sub>2</sub>,
- Decomposition is lossy if  $R_1 \bowtie R_2 \supsetneq R$
- Decomposition is lossless if  $R_1 \bowtie R_2 = R$

# lossless join decomposition using FD set

**To check for lossless join decomposition using FD set, following conditions must hold:**

- Union of Attributes of R<sub>1</sub> and R<sub>2</sub> must be equal to attribute of R. Each attribute of R must be either in R<sub>1</sub> or in R<sub>2</sub>.  $\text{Att}(R_1) \cup \text{Att}(R_2) = \text{Att}(R)$
- Intersection of Attributes of R<sub>1</sub> and R<sub>2</sub> must not be NULL.  $\text{Att}(R_1) \cap \text{Att}(R_2) \neq \Phi$
- Common attribute must be a key for at least one relation (R<sub>1</sub> or R<sub>2</sub>)  
 $\text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow \text{Att}(R_1)$  or  $\text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow \text{Att}(R_2)$
- For Example, A relation R (A, B, C, D) with FD set {A->BC} is decomposed into R<sub>1</sub>(ABC) and R<sub>2</sub>(AD) which is a lossless join decomposition as:
  - First condition holds true as  $\text{Att}(R_1) \cup \text{Att}(R_2) = (\text{ABC}) \cup (\text{AD}) = (\text{ABCD}) = \text{Att}(R)$ .
  - Second condition holds true as  $\text{Att}(R_1) \cap \text{Att}(R_2) = (\text{ABC}) \cap (\text{AD}) \neq \Phi$
  - Third condition holds true as  $\text{Att}(R_1) \cap \text{Att}(R_2) = A$  is a key of R<sub>1</sub>(ABC) because A->BC is given.

# Dependency Preserving Decomposition

- If we decompose a relation R into relations R<sub>1</sub> and R<sub>2</sub>, All dependencies of R either must be a part of R<sub>1</sub> or R<sub>2</sub> or must be derivable from combination of FD's of R<sub>1</sub> and R<sub>2</sub>.
- For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R<sub>1</sub>(ABC) and R<sub>2</sub>(AD) which is dependency preserving because FD A->BC is a part of R<sub>1</sub>(ABC).
- **Consider a schema R(A,B,C,D) and functional dependencies A->B and C->D. Then the decomposition of R into R<sub>1</sub>(AB) and R<sub>2</sub>(CD) is: [choose the correct option]**
  - A. dependency preserving and lossless join
  - B. lossless join but not dependency preserving
  - C. dependency preserving but not lossless join
  - D. not dependency preserving and not lossless join
- **Answer:** For lossless join decomposition, these three conditions must hold true:
  - Att(R<sub>1</sub>) U Att(R<sub>2</sub>) = ABCD = Att(R)
  - Att(R<sub>1</sub>) ∩ Att(R<sub>2</sub>) = Φ, which violates the condition of lossless join decomposition.
  - Hence the decomposition is not lossless.

- For dependency preserving decomposition,  
 $A \rightarrow B$  can be ensured in  $R_1(AB)$  and  $C \rightarrow D$  can be ensured in  $R_2(CD)$ . Hence it is dependency preserving decomposition.  
So, the correct option is C.

# Chapter Summary

- Informal Design Guidelines for Relational Databases
- Functional Dependencies (FDs)
- Normal Forms (1NF, 2NF, 3NF) Based on Primary Keys
- General Normal Form Definitions of 2NF and 3NF (For Multiple Keys)
- BCNF (Boyce-Codd Normal Form)
- Fourth and Fifth Normal Forms

# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module II- Part 2**  
DATABASE LANGUAGES & SQL

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

# Syllabus Module 2

- **Module 2:** RELATIONAL MODEL, DATABASE LANGUAGES & SQL
- **Relational Model:** Structure of relational Databases, Integrity Constraints, synthesizing ER diagram to relational schema  
(Reading: Elmasri Navathe, Ch. 3 and 8.1, Additional Reading: Silberschatz, Korth, Ch. 2.1-2.4)
- **Database Languages:** Concept of DDL and DML relational algebra (Reading: Silbershatz, Korth, Ch 2.5-2.6 and 6.1-6.2, Elmasri Navathe, Ch. 6.1-6.5).
- **Structured Query Language (SQL):** Basic SQL Structure, examples, Set operations, Aggregate Functions, nested sub-queries (Reading: Elmasri Navathe, Ch. 4 and 5.1) Views, assertions and triggers (Reading: Elmasri Navathe, Ch. 5.2-5.3, Optional reading: Silbershatz, Korth Ch. 5.3).

# **Structured Query Language (SQL)**

Basic SQL Structure

# Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - Physical storage structure of each relation on disk.

# Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length  $n$ .
- **varchar(n).** Variable length character strings, with user-specified maximum length  $n$ .
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of  $p$  digits, with  $d$  digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least  $n$  digits.

# Create Table Construct

- An SQL relation is defined using the **create table** command:

**create table**  $r$  ( $A_1 D_1, A_2 D_2, \dots, A_n D_n,$   
(integrity-constraint<sub>1</sub>),  
...,  
(integrity-constraint<sub>k</sub>))

- $r$  is the name of the relation
  - each  $A_i$  is an attribute name in the schema of relation  $r$
  - $D_i$  is the data type of values in the domain of attribute  $A_i$
- Example:

**create table** *instructor* (  
    *ID*           **char(5)**,  
    *name*       **varchar(20)**,  
    *dept\_name* **varchar(20)**,  
    *salary*       **numeric(8,2)**)

# Integrity Constraints in Create Table

- **not null**
- **primary key** ( $A_1, \dots, A_n$ )
- **foreign key** ( $A_m, \dots, A_n$ ) **references**  $r$

*Example:*

```
create table instructor (
    ID          char(5),
    name        varchar(20) not null,
    dept_name  varchar(20),
    salary      numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references
```

*department*);

**primary key** declaration on an attribute automatically ensures **not null**

# And a Few More Relation Definitions

- **create table student (**  
 $ID$            **varchar(5),**  
 $name$        **varchar(20) not null,**  
 $dept\_name$    **varchar(20),**  
 $tot\_cred$      **numeric(3,0),**  
**primary key** ( $ID$ ),  
**foreign key** ( $dept\_name$ ) **references** department);
  
- **create table takes (**  
 $ID$            **varchar(5),**  
 $course\_id$    **varchar(8),**  
 $sec\_id$        **varchar(8),**  
 $semester$      **varchar(6),**  
 $year$           **numeric(4,0),**  
 $grade$         **varchar(2),**  
**primary key** ( $ID, course\_id, sec\_id, semester, year$ ),  
**foreign key** ( $ID$ ) **references** student,  
**foreign key** ( $course\_id, sec\_id, semester, year$ )  
**references** section);  
  - Note:  $sec\_id$  can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

# And more still

- **create table course (**  
*course\_id*      **varchar(8),**  
*title*            **varchar(50),**  
*dept\_name*      **varchar(20),**  
*credits*          **numeric(2,0),**  
**primary key (course\_id),**  
**foreign key (dept\_name) references**  
*department);*

# Updates to tables

- **Drop Table**
  - **drop table**  $r$
- **Alter**
  - **alter table**  $r$  **add**  $A D$ 
    - where  $A$  is the name of the attribute to be added to relation  $r$  and  $D$  is the domain of  $A$ .
    - All existing tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table**  $r$  **drop**  $A$ 
    - where  $A$  is the name of an attribute of relation  $r$
    - Dropping of attributes not supported by many databases.

# Basic Query Structure

- A typical SQL query has the form:

```
select  $A_1, A_2, \dots, A_n$ 
from  $r_1, r_2, \dots, r_m$ 
where  $P$ 
```

- $A_i$  represents an attribute
- $R_i$  represents a relation
- $P$  is a predicate.
- The result of an SQL query is a relation.

# The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:  
**select** *name*  
**from** *instructor*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name*  $\equiv$  *NAME*  $\equiv$  *name*
  - Some people use upper case wherever we use bold font.

# The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the department names of all instructors, and remove duplicates

**select distinct dept\_name  
from instructor**

- The keyword **all** specifies that duplicates should not be removed.

**select all dept\_name  
from instructor**

# The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

**select \***

**from** *instructor*

- An attribute can be a literal with no **from** clause

**select** ‘437’

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

**select** ‘437’ **as** *FOO*

- An attribute can be a literal with **from** clause

**select** ‘A’

**from** *instructor*

- Result is a table with one column and  $N$  rows (number of tuples in the *instructors* table), each row with value “A”

# The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, +, −, \*, and /, and operating on constants or attributes of tuples.
  - The query:

```
select ID, name, salary/12  
from instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```

# The where Clause

- The **where** clause specifies conditions that the result must satisfy

- Corresponds to the selection predicate of the relational algebra.

- To find all instructors in Comp. Sci. dept

**select** *name*

**from** *instructor*

**where** *dept\_name* = 'Comp. Sci.'

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**

- To find all instructors in Comp. Sci. dept with salary > 80000

**select** *name*

**from** *instructor*

**where** *dept\_name* = 'Comp. Sci.' **and** *salary* > 80000

- Comparisons can be applied to results of arithmetic expressions.

# The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

```
select *
  from instructor, teaches
```

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

## Cartesian Product

## *instructor*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
62456	Gill	Finance	67000

*teaches*

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

# Examples

- Find the names of all instructors who have taught some course and the course\_id
  - **select** *name, course\_id*  
**from** *instructor , teaches*  
**where** *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course\_id
  - **select** *name, course\_id*  
**from** *instructor , teaches*  
**where** *instructor.ID = teaches.ID and instructor.dept\_name = 'Art'*

# The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:  
 $old-name \text{ as } new-name$
- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci.’.
  - **select distinct**  $T.name$   
**from**  $instructor \text{ as } T, instructor \text{ as } S$   
**where**  $T.salary > S.salary \text{ and } S.dept\_name = 'Comp. Sci.'$
- Keyword **as** is optional and may be omitted  
 $instructor \text{ as } T \equiv instructor \text{ } T$

# Self Join Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Find ALL the supervisors (direct and indirect) of “Bob”

# String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent ( % ). The % character matches any substring.
  - underscore ( \_ ). The \_ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name  
from instructor  
where name like '%dar%'
```

- Match the string “100%”  
**like** '100 \%' **escape** '\'  
in that above we use backslash (\) as the escape character.

# String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - ‘Intro%’ matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘\_\_\_\_’ matches any string of exactly three characters.
  - ‘\_\_\_%’ matches any string of at least three characters.
- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

# Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name  
from instructor  
order by name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name desc*
- Can sort on multiple attributes
  - Example: **order by** *dept\_name, name*

# Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is,  $\geq \$90,000$  and  $\leq \$100,000$ )
  - **select** *name*  
**from** *instructor*  
**where** *salary between* 90000 **and** 100000
- Tuple comparison
  - **select** *name, course\_id*  
**from** *instructor, teaches*  
**where** (*instructor.ID, dept\_name*) = (*teaches.ID, 'Biology'*);

# Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations  $r_1$  and  $r_2$ :
  1.  $\sigma_\theta(r_1)$ : If there are  $c_1$  copies of tuple  $t_1$  in  $r_1$ , and  $t_1$  satisfies selection  $\sigma_\theta$ , then there are  $c_1$  copies of  $t_1$  in  $\sigma_\theta(r_1)$ .
  2.  $\Pi_A(r)$ : For each copy of tuple  $t_1$  in  $r_1$ , there is a copy of tuple  $\Pi_A(t_1)$  in  $\Pi_A(r)$  where  $\Pi_A(t_1)$  denotes the projection of the single tuple  $t_1$ .
  3.  $r_1 \times r_2$ : If there are  $c_1$  copies of tuple  $t_1$  in  $r_1$  and  $c_2$  copies of tuple  $t_2$  in  $r_2$ , there are  $c_1 \times c_2$  copies of the tuple  $t_1 \cdot t_2$  in  $r_1 \times r_2$

# Duplicates (Cont.)

- Example: Suppose multiset relations  $r_1(A, B)$  and  $r_2(C)$  are as follows:  
 $r_1 = \{(1, a) (2, a)\}$      $r_2 = \{(2), (3), (3)\}$
- Then  $\Pi_B(r_1)$  would be  $\{(a), (a)\}$ , while  $\Pi_B(r_1) \times r_2$  would be  $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL duplicate semantics:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

# Set Operations: union, intersect, except

- Find courses that ran in Fall 2009 or in Spring 2010  
**(select course\_id from section where sem = 'Fall' and year = 2009)**  
union  
**(select course\_id from section where sem = 'Spring' and year = 2010)**
- Find courses that ran in Fall 2009 and in Spring 2010  
**(select course\_id from section where sem = 'Fall' and year = 2009)**  
intersect  
**(select course\_id from section where sem = 'Spring' and year = 2010)**
- Find courses that ran in Fall 2009 but not in Spring 2010  
**(select course\_id from section where sem = 'Fall' and year = 2009)**  
except  
**(select course\_id from section where sem = 'Spring' and year = 2010)**

# Set Operations (Cont.)

- Find the salaries of all instructors that are less than the largest salary.
  - **select distinct** *T.salary*  
**from** *instructor* **as** *T*, *instructor* **as** *S*  
**where** *T.salary* < *S.salary*
- Find all the salaries of all instructors
  - **select distinct** *salary*  
**from** *instructor*
- Find the largest salary of all instructors.
  - (**select** “second query”)  
**except**  
(**select** “first query”))

# Set Operations (Cont.)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.
- Suppose a tuple occurs  $m$  times in  $r$  and  $n$  times in  $s$ , then, it occurs:
  - $m + n$  times in  $r$  **union all**  $s$
  - $\min(m,n)$  times in  $r$  **intersect all**  $s$
  - $\max(0, m - n)$  times in  $r$  **except all**  $s$

# Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example:  $5 + \text{null}$  returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

```
select name  
from instructor  
where salary is null
```

# Null Values and Three Valued Logic

- Three values – *true, false, unknown*
- Any comparison with *null* returns *unknown*
  - Example:  $5 < \text{null}$  or  $\text{null} <> \text{null}$  or  $\text{null} = \text{null}$
- Three-valued logic using the value *unknown*:
  - OR: (*unknown or true*) = *true*,  
(*unknown or false*) = *unknown*  
(*unknown or unknown*) = *unknown*
  - AND: (*true and unknown*) = *unknown*,  
(*false and unknown*) = *false*,  
(*unknown and unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - “*P is unknown*” evaluates to *true* if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

# Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

**avg:** average value

**min:** minimum value

**max:** maximum value

**sum:** sum of values

**count:** number of values

# Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - **select avg (salary)**  
**from instructor**  
**where dept\_name= 'Comp. Sci.';**
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count (distinct ID)**  
**from teaches**  
**where semester = 'Spring' and year = 2010;**
- Find the number of tuples in the *course* relation
  - **select count (\*)**  
**from course;**

# Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - select dept\_name, avg (salary) as avg\_salary  
from instructor  
group by dept\_name;**

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

# Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /\* erroneous query \*/

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

# Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary)
from instructor
group by dept_name
having avg (salary) > 42000;
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

# Null Values and Aggregates

- Total all salaries  
**select sum (salary )  
from instructor**
  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(\*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

# Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

as follows:

- $A_i$  can be replaced by a subquery that generates a single value.
- $r_i$  can be replaced by any valid subquery
- $P$  can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

Where  $B$  is an attribute and  $<\text{operation}>$  to be defined later.

# Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality.

# Set Membership

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id not in (select course_id
from section
where semester = 'Spring' and year= 2010);
```

# Set Membership (Cont.)

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
    (select course_id, sec_id, semester, year
     from teaches
     where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner.  
The formulation above is simply to illustrate SQL features.

# Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

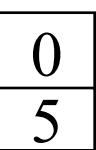
- Same query using **> some** clause

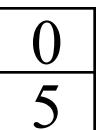
```
select name
from instructor
where salary > some (select salary
                      from instructor
                      where dept name = 'Biology');
```

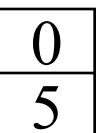
# Definition of “some” Clause

- $F \text{ } <\!\! \text{comp} \!\!> \text{ some } r \Leftrightarrow \exists t \in r \text{ such that } (F \text{ } <\!\! \text{comp} \!\!> t)$   
 Where  $<\!\! \text{comp} \!\!>$  can be:  $<, \leq, >, =, \neq$

( $5 < \text{some}$   ) = true      (read: 5 < some tuple in the relation)

( $5 < \text{some}$   ) = false

( $5 = \text{some}$   ) = true

( $5 \neq \text{some}$   ) = true (since  $0 \neq 5$ )

( $= \text{some}$ )  $\equiv$  in

However, ( $\neq \text{some}$ )  $\not\equiv$  not in

# Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in Biology department.

```
select name
  from instructor
 where salary > all (select salary
                        from instructor
                      where dept name = 'Biology');
```

# Definition of “all” Clause

- $F \text{ <comp> all } r \Leftrightarrow \forall t \in r (F \text{ <comp> } t)$

(5 < all ) = false

0
5
6

(5 < all ) = true

6
4
0

(5 = all ) = false

4
5

(5 ≠ all ) = true (since  $5 \neq 4$  and  $5 \neq 6$ )

4
6

$(\neq \text{ all}) \equiv \text{not in}$

However,  $(= \text{ all}) \not\equiv \text{in}$

# Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists**  $r \Leftrightarrow r \neq \emptyset$
- **not exists**  $r \Leftrightarrow r = \emptyset$

# Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
exists (select *
    from section as T
    where semester = 'Spring' and year= 2010
    and S.course_id = T.course_id);
```

- Correlation name** – variable S in the outer query
- Correlated subquery** – the inner query

# Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.

```

select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
    from course
    where dept_name = 'Biology')
except
    (select T.course_id
        from takes as T
        where S.ID = T.ID));

```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took
- Note that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note:* Cannot write this query using = **all** and its variants

# Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2009

```
select T.course_id
from course as T
where unique (select R.course_id
               from section as R
               where T.course_id= R.course_id
                     and R.year = 2009);
```

# Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than \$42,000.”

```
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg(salary)
        from instructor
        group by dept_name) as dept_avg(dept_name,
avg_salary)
where avg_salary > 42000;
```

# With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
     from department)
select department.name
from department, max_budget
where department.budget =
max_budget.value;
```

# Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total(dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

# Subqueries in the Select Clause

# Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,  
       (select count(*)  
        from instructor  
        where department.dept_name =  
              instructor.dept_name)  
        as num_instructors  
from department;
```

- Runtime error if subquery returns more than one result tuple

# Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

# Deletion

- Delete all instructors  
**delete from** *instructor*
- Delete all instructors from the Finance department  
**delete from** *instructor*  
**where** *dept\_name*= 'Finance';
- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

**delete from** *instructor*  
**where** *dept name* **in** (**select** *dept name*  
**from** *department*  
**where** *building* = 'Watson');

# Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor  
where salary < (select avg (salary)  
                 from instructor);
```

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

# Insertion

- Add a new tuple to *course*

**insert into** *course*

**values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

**insert into** *course* (*course\_id*, *title*, *dept\_name*, *credits*)

**values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot\_creds* set to null

**insert into** *student*

**values** ('3003', 'Green', 'Finance', *null*);

# Insertion (Cont.)

- Add all instructors to the *student* relation with tot\_creds set to 0

**insert into student**

**select ID, name, dept\_name, 0**  
**from instructor**

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

**insert into table1 select \* from table1**  
would cause problem

# Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```
  - The order is important
  - Can be done better using the **case** statement (next slide)

# Case Statement for Conditional Updates

- Same query as before but with case statement

**update** *instructor*

**set** *salary* = **case**

**when** *salary* <= 100000

**then** *salary* \* 1.05

**else** *salary* \* 1.03

**end**

# Updates with Scalar Subqueries

- Recompute and update tot\_creds value for all students

**update student S**

**set tot\_cred = (select sum(credits)**

**from takes, course**

**where takes.course\_id = course.course\_id**

**and**

*S.ID= takes.ID.and  
takes.grade <> 'F' and  
takes.grade is not null);*

- Sets tot\_creds to null for students who have not taken any course
- Instead of **sum(credits)**, use:

**case**

**when sum(credits) is not null then**

**sum(credits)**

**else 0**

**end**

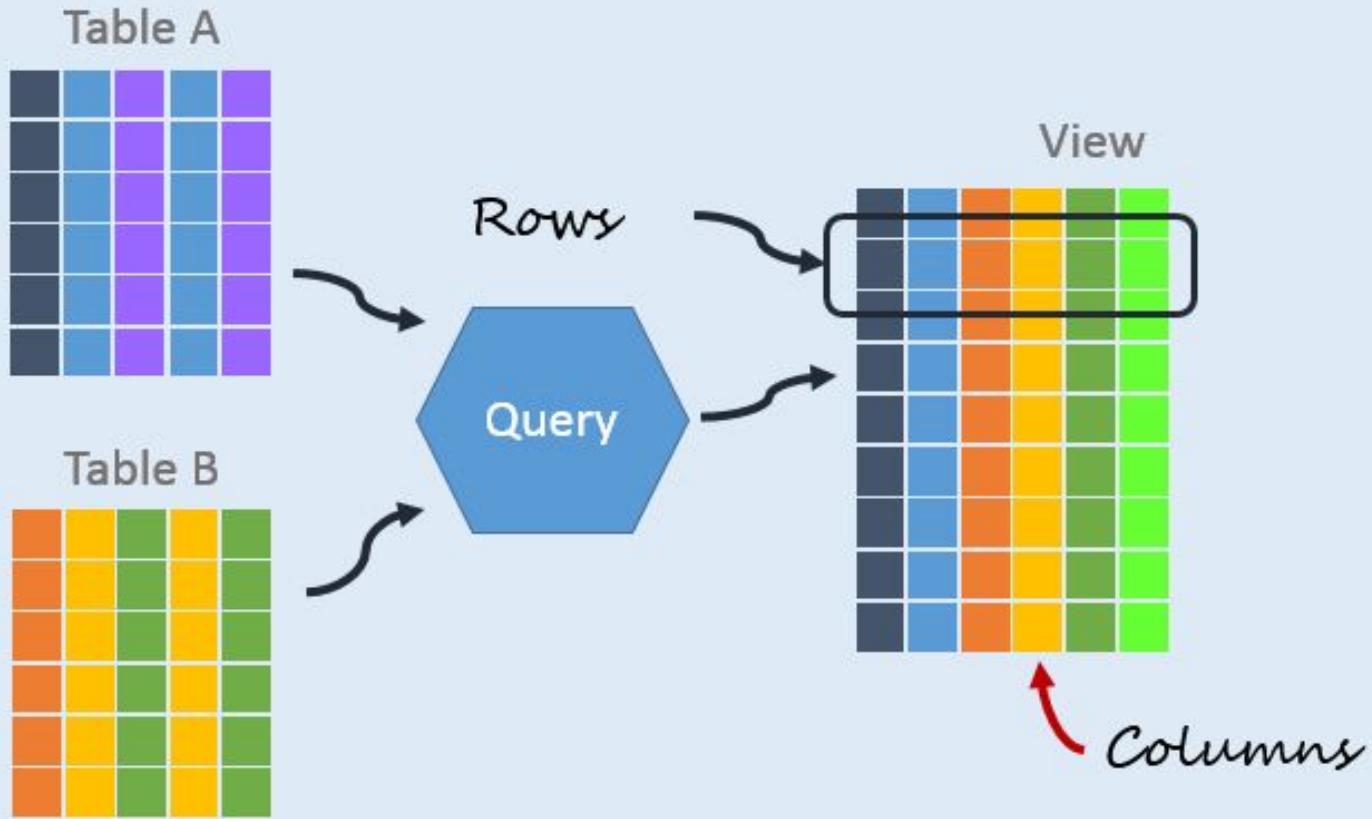
# Views

- It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users.
- Consider a person who needs to know a customer's loan number and branch name but no need to see the loan amount.
- Aside from security concerns we may wish to create a **personalized collection of relations** that is better matched to a certain **user's intuition** than is the logical model.
- Any relation that is not part of the logical model, but is made visible to a user as a virtual relation is called a **View**.

# SQL Views

- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.
- We can create a view by selecting fields from one or more tables present in the database.
- A view is created with the CREATE VIEW statement.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```



## Characteristics

- One or more source tables make up a view
- Query follows “SELECT STATEMENT” format
- Views generally read-only
- Views don’t require additional storage

# VIEW Example

1. The following SQL creates a view that shows all customers from Brazil:
  - CREATE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName  
FROM Customers  
WHERE Country = 'Brazil';
  - SELECT \* FROM [Brazil Customers];
2. The following SQL creates a view that selects every product in the "Products" table with a price higher than average price:

```
CREATE VIEW [Products Above Average Price] AS  
SELECT ProductName, Price  
FROM Products  
WHERE Price > (SELECT AVG(Price) FROM Products);  
SELECT * FROM [Products Above Average Price];
```

# Creating View from multiple tables

- In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks.
- To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query:

```
CREATE VIEW MarksView AS SELECT
StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS FROM StudentDetails,
StudentMarks WHERE StudentDetails.NAME =
StudentMarks.NAME;
```

- To display data of View MarksView:  

```
SELECT * FROM MarksView;
```

# Materialized views

- Certain database systems allow to store view relations, but they make sure that, if the actual relations used in the view change, the view is kept up to date. Such views are called materialized views.
- The process of keeping the view up to date is called view maintenance.
- Of course, the benefits to queries from the materialization of a view must be weighted against the storage cost and the added overhead for updates.

# Views Defined by Using Other Views

- One view may be used in the expression defining another view
- Example
- ```
CREATE VIEW [Perryridge_Customer] AS
SELECT CustomerName
FROM All_Customers
WHERE branch_name = 'Perryridge';
```
- All\_Customers is a view relation
- View expansion is one way to define the meaning of views defined in terms of other views.
- View definitions are not recursive

# View expansion Example

- **SELECT \* FROM Perryridge\_Customer WHERE customer\_name = 'John';**

1. View-expansion procedure initially generates

- **SELECT \* FROM (SELECT CustomerName  
FROM All\_Customers  
WHERE branch\_name = 'Perryridge')  
WHERE customer\_name = 'John';**

2. It then generates

- **SELECT \* FROM (SELECT CustomerName  
FROM ((SELECT branch\_name, CustomerName  
FROM depositor, account  
WHERE depositor.account\_number = account.account\_number)  
UNION  
(SELECT branch\_name, CustomerName  
FROM borrower, loan  
WHERE borrower.loan\_number = loan.loan\_number))  
WHERE branch\_name = 'Perryridge')  
WHERE customer\_name = 'John';**

# SQL Updating a View

- New tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the base tables.
- Modifications to a view must be translated to a modification to the actual relations in the logical model of the database.
- A view can be updated with the CREATE OR REPLACE VIEW statement (ADD/DELETE Columns).
- The following SQL adds the "City" column to the "Brazil Customers" view:

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

- A view is deleted with the DROP VIEW statement.

```
DROP VIEW view_name;
```

# Update View Example

1. CREATE VIEW loan\_branch AS  
SELECT loan\_number, branch\_name  
FROM loan
  2. INSERT INTO loan\_branch  
VALUES ('L-37', 'Perryridge')
- May Reject the insertion or insert the following tuple

| Loan_number | Branch_name | amount |
|-------------|-------------|--------|
| L-37        | Perryridge  | null   |

# View Update conditions

- Updating of views is complicated and can be ambiguous.
- An SQL view is updatable (insert, update or delete) with the following conditions:
  - i. The from clause has only one database relation
  - ii. The select clause contains only attribute names of the relation, and doesn't have any expressions, aggregates or distinct specification.
  - iii. Any attribute not listed in the select clause can be set to null.
  - iv. The query doesn't have a group by or having clause.

- The **update**, **insert**, and **delete** operations would be allowed on the following view:  

```
create view history instructors as
select *
from instructor
where dept name= 'History';
```
- Suppose that a user tries to insert the tuple ('25566', 'Brown', 'Biology', 100000) into the *history instructors* view.
- This tuple can be inserted into *instructor* relation, but it would not appear in *history instructors* view since it does not satisfy selection imposed by view.
- However, views can be defined with a **with check option** clause at the end of the view definition;
- then, if a tuple inserted into the view does not satisfy view's **where** clause condition, insertion is rejected by the database system.

- **create view instructor info as**  
**select ID, name, building**  
**from instructor, department**  
**where instructor.dept name= department.dept name;**
- **insert into instructor info**  
**values ('69987', 'White', 'Taylor');**

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 10101     | Srinivasan  | Comp. Sci.       | 65000         |
| 12121     | Wu          | Finance          | 90000         |
| 15151     | Mozart      | Music            | 40000         |
| 22222     | Einstein    | Physics          | 95000         |
| 32343     | El Said     | History          | 60000         |
| 33456     | Gold        | Physics          | 87000         |
| 45565     | Katz        | Comp. Sci.       | 75000         |
| 58583     | Califieri   | History          | 62000         |
| 76543     | Singh       | Finance          | 80000         |
| 76766     | Crick       | Biology          | 72000         |
| 83821     | Brandt      | Comp. Sci.       | 92000         |
| 98345     | Kim         | Elec. Eng.       | 80000         |
| 69987     | White       | <i>null</i>      | <i>null</i>   |

| <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|------------------|-----------------|---------------|
| Biology          | Watson          | 90000         |
| Comp. Sci.       | Taylor          | 100000        |
| Electrical Eng.  | Taylor          | 85000         |
| Finance          | Painter         | 120000        |
| History          | Painter         | 50000         |
| Music            | Packard         | 80000         |
| Physics          | Watson          | 70000         |
| <i>null</i>      | Painter         | <i>null</i>   |

# Why Views are used in SQL?

- Views are used **for security purposes because they provide encapsulation of the name of the table.**
- Data is in the virtual table, not stored permanently.
- Views display only selected data.
- We can also use Sql Joins in the Select statement in deriving the data for the view.
- Views can be used to structure data in ways for users to find it natural,
  - Simplify complex queries,
  - Restrict access to data,
  - Summarize data from several tables to create reports.

# General Constraints as Assertions

- An assertion is a predicate expressing a condition that the database must always satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions.
- However, there are many constraints that we cannot express by using only these special forms.
- For example, “Every department must have at least five courses offered every semester” must be expressed as an assertion.
- When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

# Constraint Name

- Constraint may be given a constraint name, following the keyword CONSTRAINT.
- The names of all constraints within a particular schema must be unique.
- A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint.
- Giving names to constraints is optional.

- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- Declarative assertions, using the CREATE ASSERTION statement of the DDL.
- For example, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for"
- In SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS
(SELECT *
FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
WHERE E.SALARY>M.SALARY AND
E.DNO=:D.DNUMBER AND
D.MGRSSN=:M.SSN) );
```

- The constraint name **SALARY\_CONSTRAINT** is followed by the keyword **CHECK**, which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied.
- The constraint name can be used later to refer to the constraint or to modify or drop it.
- The DBMS is responsible for ensuring that the condition is not violated.
- Whenever some tuples in the database cause the condition of an **ASSERTION** statement to evaluate to **FALSE**, the constraint is violated.
- The constraint is satisfied by a database state if no *combination of tuples* in that database state violates the constraint.

- The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition*.
- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty.
- Thus, the assertion is violated if the result of the query is not empty.
- The CHECK clause and constraint condition can also be used to specify constraints on attributes and domains and on tuples.
- A major difference between CREATE ASSERTION and the other two is that the CHECK clauses on attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated*.
- Hence, constraint checking can be implemented more efficiently by the DBMS in these cases.
- The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*.
- On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that checks are implemented more efficiently by the DBMS

# Trigger

- A trigger specifies an event (such as a particular database update operation), a condition, and an action. The action is to be executed automatically if the condition is satisfied when the event occurs.
- Oracle triggers are close to the way rules are specified in the SQL standard

# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module II**  
RELATIONAL MODEL, DATABASE LANGUAGES & SQL

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

|             |                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------|
| <b>CO 1</b> | Define, explain and illustrate the fundamental concepts of databases                             |
| <b>CO 2</b> | Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.   |
| <b>CO 3</b> | Model and design solutions for efficiently representing and querying data using relational model |
| <b>CO 4</b> | Demonstrate the features of indexing and hashing in database applications                        |
| <b>CO 5</b> | Discuss and compare the aspects of Concurrency Control and Recovery in Database systems          |
| <b>CO 6</b> | Appreciate the latest trends in databases                                                        |

# Syllabus Module 2

- **Module 2:** RELATIONAL MODEL, DATABASE LANGUAGES & SQL
- **Relational Model:** Structure of relational Databases, Integrity Constraints, synthesizing ER diagram to relational schema  
(Reading: Elmasri Navathe, Ch. 3 and 8.1, Additional Reading: Silberschatz, Korth, Ch. 2.1-2.4)
- **Database Languages:** Concept of DDL and DML relational algebra (Reading: Silbershatz, Korth, Ch 2.5-2.6 and 6.1-6.2, Elmasri Navathe, Ch. 6.1-6.5).
- **Structured Query Language (SQL):** Basic SQL Structure, examples, Set operations, Aggregate Functions, nested sub-queries (Reading: Elmasri Navathe, Ch. 4 and 5.1) Views, assertions and triggers (Reading: Elmasri Navathe, Ch. 5.2-5.3, Optional reading: Silbershatz, Korth Ch. 5.3).

# **Relational Model**

Structure of relational Databases

# Structure of Relational Databases

- The relational model represents the DB as a collection of relations.(each relation resembles a table of values).
- Each row in the table represents a collection of related data values(corresponds to a real-world entity or relationship).
- A row is called a **tuple**.
- A column header is called an **attribute** and the table is called a relation.

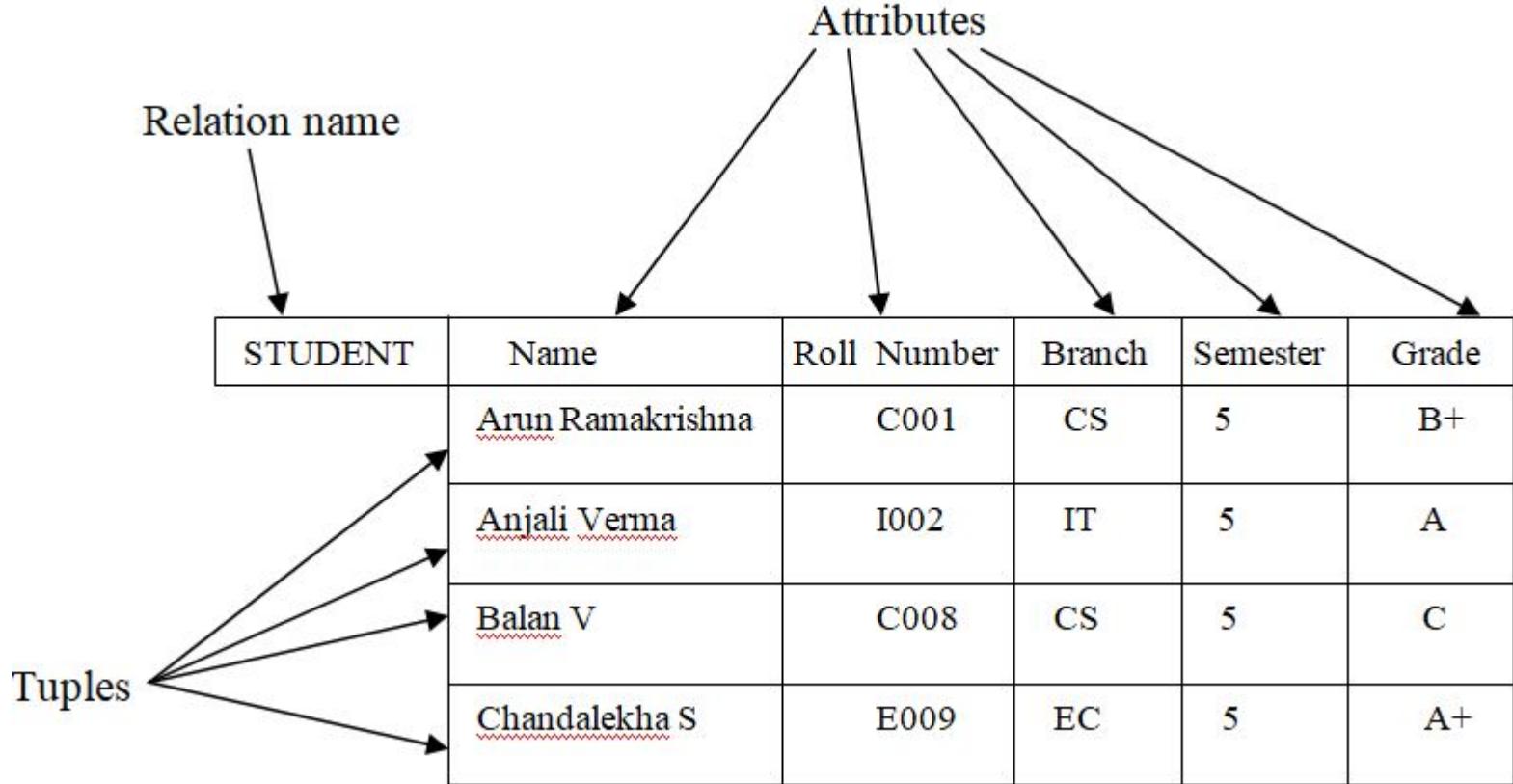
**Domain:** is a set of atomic values .

- For each attribute there is a set of permitted values called domain of that attribute.
- A domain is specified in terms of a Name, datatype and format.

# Relation Schema

- It is a description of a **relation** R which is a collection of **attributes** A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>, ..A<sub>n</sub>.
- Each attribute A<sub>i</sub> is the name of a role played by the domain D in the relation R (D is called the domain of A<sub>i</sub> and denoted as dom(A<sub>i</sub>)).
- R is the name of the relation.
- The **degree** of a relation is the number of attributes **n** of its relation schema.
- A relation state at a given time is called as **current relation state** .
- It reflects only the valid tuple that represents a particular state of the real world.
- As the state of real world changes, the relation also changes .
- Schema R is relatively static and does not change except very infrequently eg: adding an attribute

- Eg- Relation schema for a relation of degree 5 describes Engineering students
- STUDENT(Name, Roll\_Number, Branch, Semester, Grade)
- Relational Database Model



- Terminology

| Informal Terms     | Formal Terms       |
|--------------------|--------------------|
| Table              | Relation           |
| Column             | Attribute          |
| Row                | Tuple              |
| Values in a Column | Domain             |
| Table Definition   | Schema of Relation |
| Populated Table    | Extension          |

- **Notations for Relational Models**

- A **relation schema R** of degree **n** is given by **R(A<sub>1</sub>, A<sub>2</sub>..A<sub>n</sub>)** where A<sub>1</sub>,A<sub>2</sub>.. are attributes.
- An n-tuple ‘**t**’ in a relation **r( R )** is denoted by **t = <v<sub>1</sub>,v<sub>2</sub>..v<sub>n</sub>>** where **v<sub>1</sub>** denotes the value corresponding to attribute **A<sub>1</sub>**.

# Relational Model Constraints

- Constraints in a Relational Model are **conditions** that must hold on all valid relation instances.
- The 4 main constraints: Domain constraints, Key constraints, Entity Integrity constraints, and Referential Integrity constraints.

## 1, Domain Constraints

- They specify that the value of each attribute must be an atomic value from the domain of the attribute.

## 2. Key Constraints

- i. **Superkey of R:** No two tuples in a Relation can have the same combination of their values for all their attributes.

$$t_1[SK] \neq t_2[SK]$$

- ii. **Key of R:** A 'minimal 'superkey K - removal of any attribute from K results in a set of attributes that is not a superkey

- An attribute can be permitted to have a **NULL** value or not. Eg- For the CAR the Serial# attribute cannot be NULL.
- The Serial# of the CAR is constraint to be NOT NULL.

### **3. Entity Integrity Constraints** : no primary key value can be null

|          |      |               |         |     |     |          |         |
|----------|------|---------------|---------|-----|-----|----------|---------|
| EMPLOYEE | Name | <u>Emp_id</u> | Address | Sex | Dob | Super_id | Dept_no |
|----------|------|---------------|---------|-----|-----|----------|---------|

|            |      |                |            |
|------------|------|----------------|------------|
| DEPARTMENT | Name | <u>Dept_no</u> | Manager_id |
|------------|------|----------------|------------|

|               |                |                 |
|---------------|----------------|-----------------|
| DEPT_LOCATION | <u>Dept_no</u> | <u>Location</u> |
|---------------|----------------|-----------------|

|         |      |         |         |
|---------|------|---------|---------|
| PROJECT | Name | Proj_no | Dept_no |
|---------|------|---------|---------|

|          |               |                |       |
|----------|---------------|----------------|-------|
| WORKS_ON | <u>Emp_id</u> | <u>Proj_no</u> | Hours |
|----------|---------------|----------------|-------|

### **COMPANY relational database schema**

## 4. Referential Integrity : Constraint involving *two* relations

- It states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.
- Tuples in the **referencing relation** R<sub>1</sub> have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the **referenced relation** R<sub>2</sub>.

# Synthesizing ER diagram to relational schema

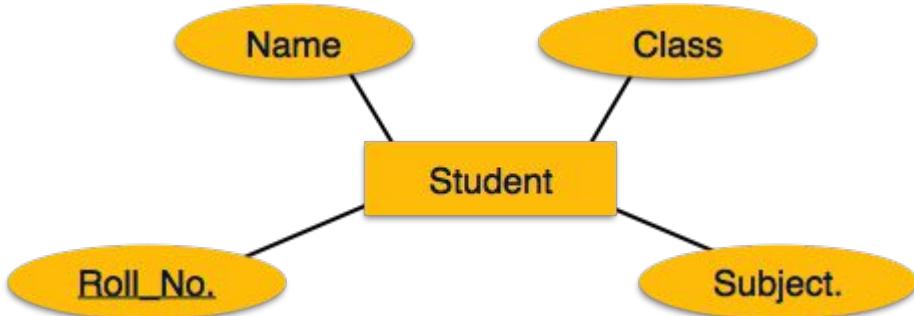
- ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.
- There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual.

ER diagrams mainly comprise of –

- Entity and its attributes
- Relationship, which is association among entities.

# Mapping Entity

- An entity is a real-world object with some attributes.

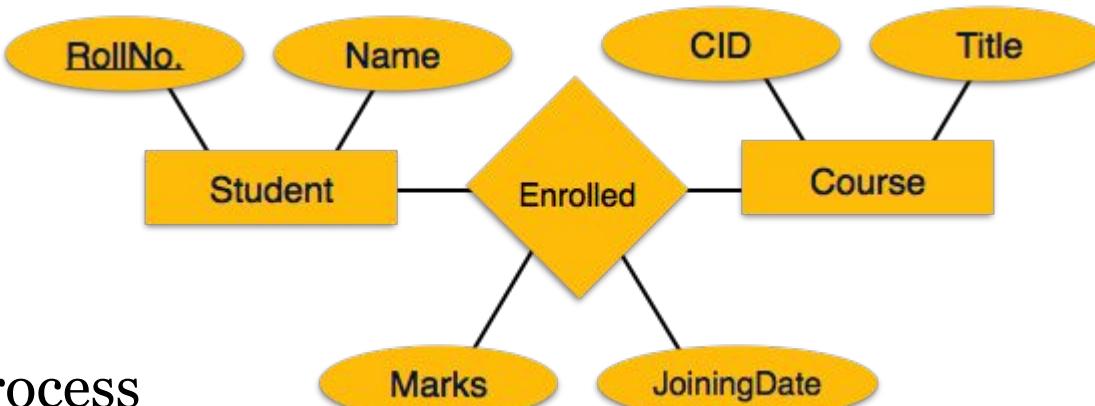


## Mapping Process (Algorithm)

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

# Mapping Relationship

- A relationship is an association among entities.

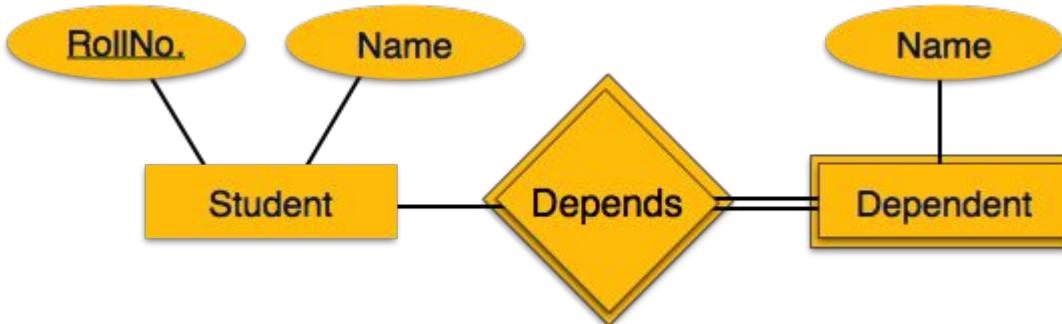


## Mapping Process

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

# Mapping Weak Entity Sets

- A weak entity set is one which does not have any primary key associated with it.



## Mapping Process

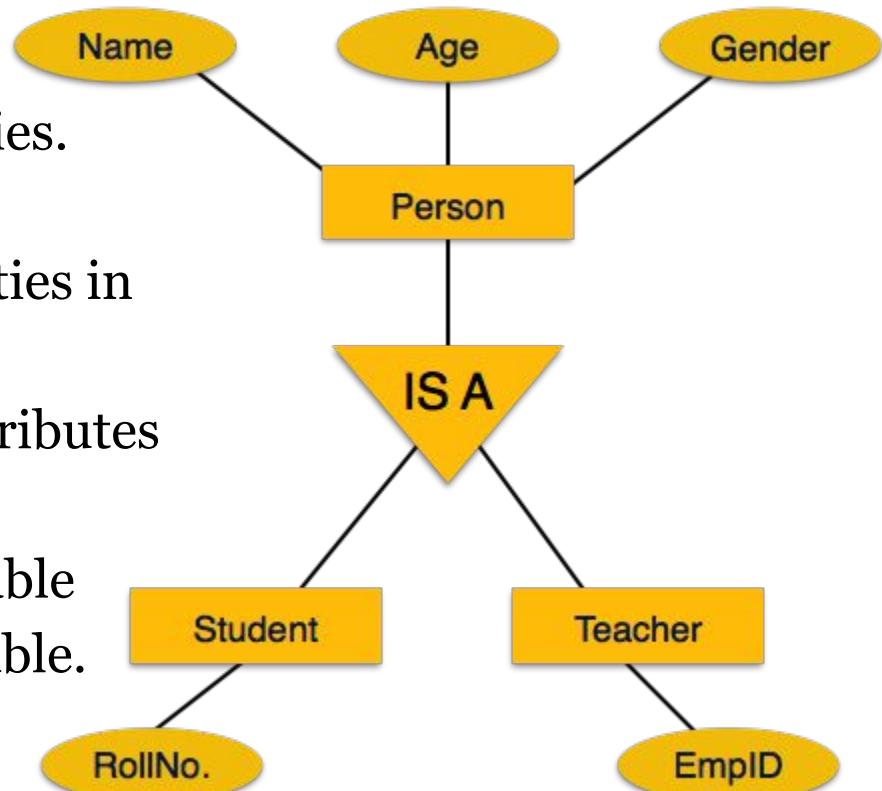
- Create table for weak entity set.
- Add all its attributes to table as field.
- Add the primary key of identifying entity set.
- Declare all foreign key constraints.

# Mapping Hierarchical Entities

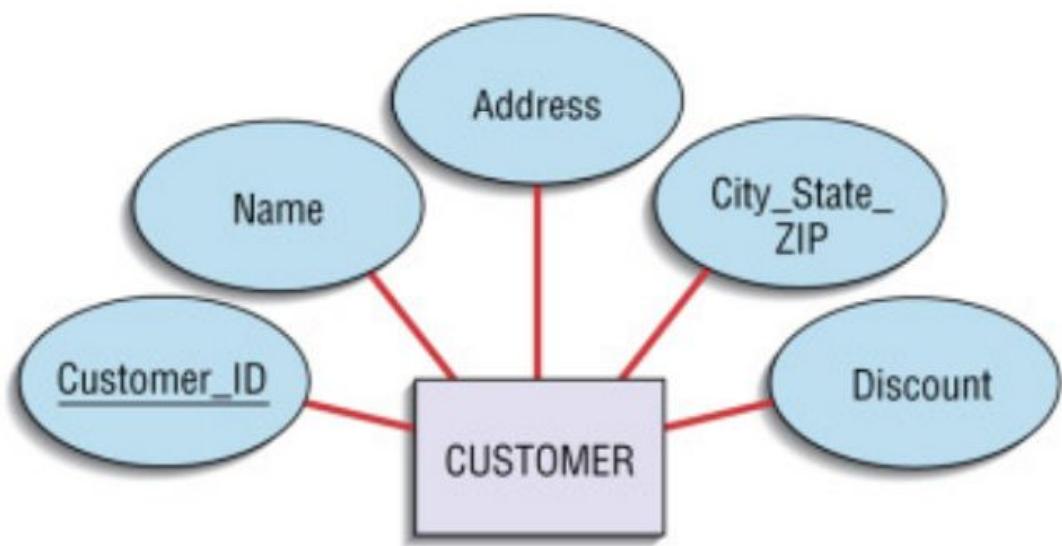
- ER specialization or generalization comes in the form of hierarchical entity sets.

## Mapping Process

- Create tables for all higher-level entities.
- Create tables for lower-level entities.
- Add primary keys of higher-level entities in the table of lower-level entities.
- In lower-level tables, add all other attributes of lower-level entities.
- Declare primary key of higher-level table and the primary key for lower-level table.
- Declare foreign key constraints.



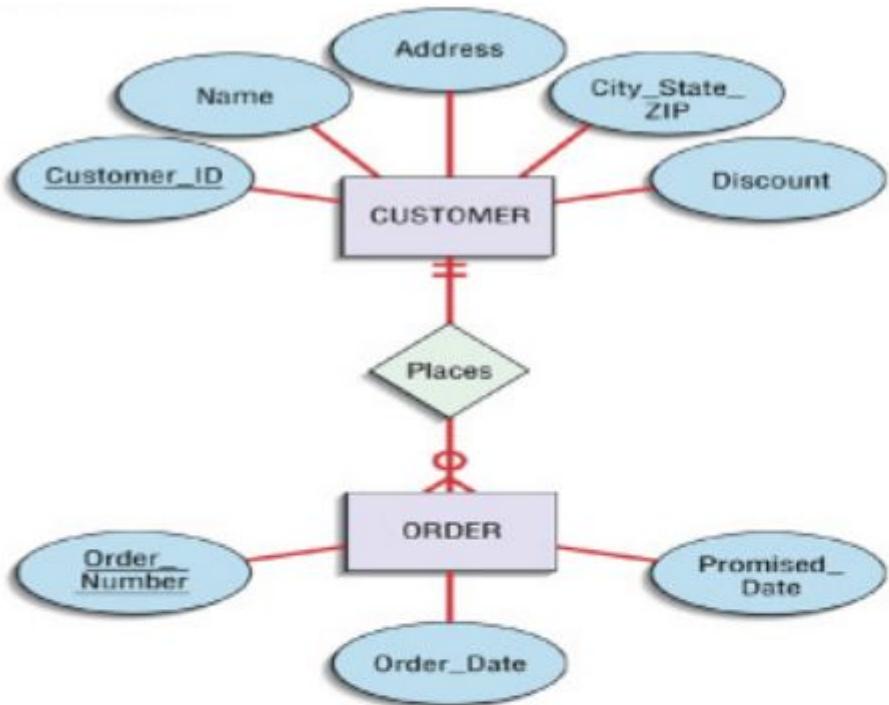
# Example 1



CUSTOMER

| Customer_ID | Name                 | Address        | City_State_ZIP        | Discount |
|-------------|----------------------|----------------|-----------------------|----------|
| 1273        | Contemporary Designs | 123 Oak St.    | Austin, TX 28384      | 5%       |
| 6390        | Casual Corner        | 18 Hoosier Dr. | Bloomington, IN 45821 | 3%       |

# Example 2

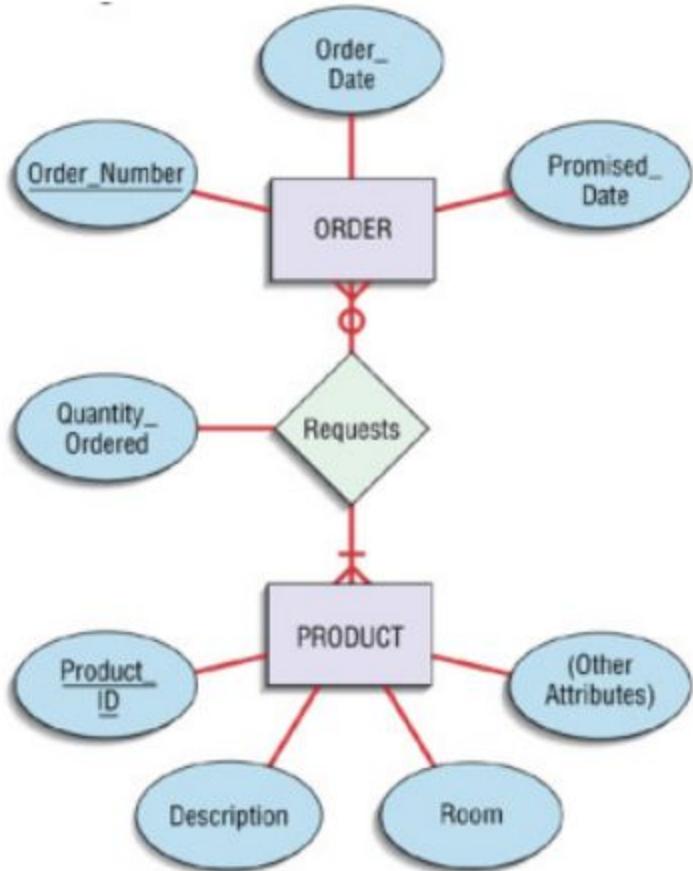


**CUSTOMER**

| <u>Customer_ID</u> | Name                 | Address        | City.State.ZIP        | Discount |
|--------------------|----------------------|----------------|-----------------------|----------|
| 1273               | Contemporary Designs | 123 Oak St.    | Austin, TX 28384      | 5%       |
| 6390               | Casual Corner        | 18 Hoosier Dr. | Bloomington, IN 45821 | 3%       |

**ORDER**

| <u>Order_Number</u> | <u>Order_Date</u> | <u>Promised_Date</u> | <u>Customer_ID</u> |
|---------------------|-------------------|----------------------|--------------------|
| 57194               | 3/15/0X           | 3/28/0X              | 6390               |
| 63725               | 3/17/0X           | 4/01/0X              | 1273               |
| 80149               | 3/14/0X           | 3/24/0X              | 6390               |


**ORDER**

| <u>Order_Number</u> | <u>Order_Date</u> | <u>Promised_Date</u> |
|---------------------|-------------------|----------------------|
| 61384               | 2/17/2002         | 3/01/2002            |
| 62009               | 2/13/2002         | 2/27/2002            |
| 62807               | 2/15/2002         | 3/01/2002            |

**ORDER LINE**

| <u>Order_Number</u> | <u>Product_ID</u> | <u>Quantity_Ordered</u> |
|---------------------|-------------------|-------------------------|
| 61384               | M128              | 2                       |
| 61384               | A261              | 1                       |

**PRODUCT**

| <u>Product_ID</u> | <u>Description</u> | <u>(Other Attributes)</u> |
|-------------------|--------------------|---------------------------|
| M128              | Bookcase           | —                         |
| A261              | Wall unit          | —                         |
| R149              | Cabinet            | —                         |

# **Relational Algebra**

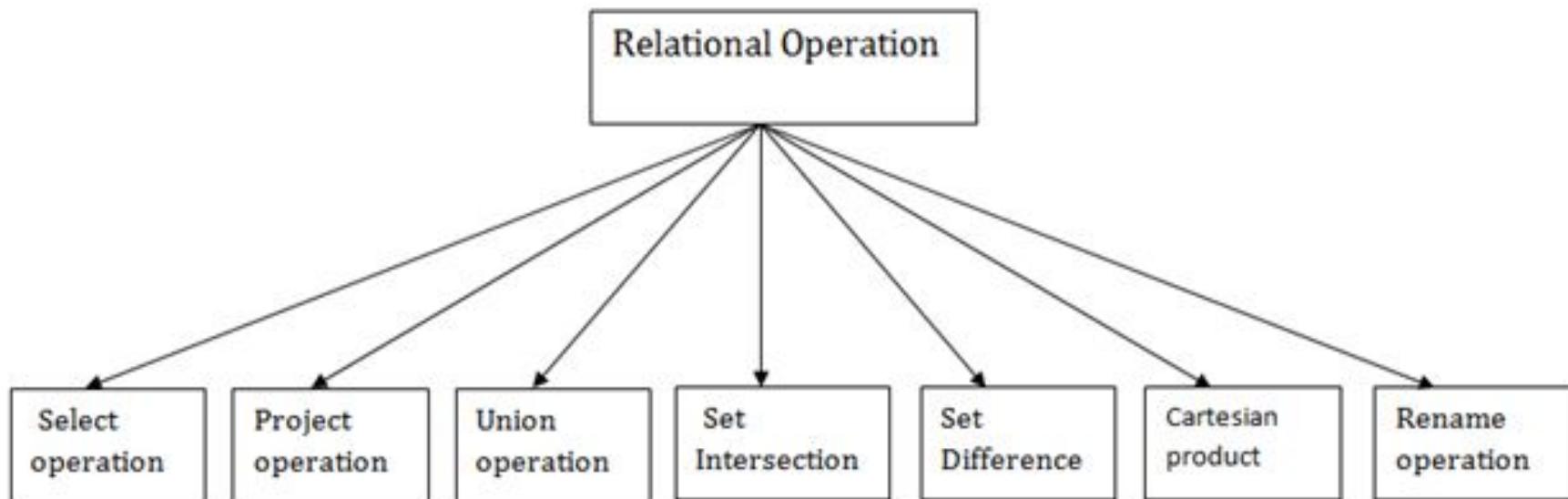
Relational Algebra Operations  
Unary, Set theory, Binary

# Relational Algebra Overview

- Relational algebra is the basic **set of operations** for the relational model
- These operations enable a user to specify **basic retrieval requests** (or **queries**)
- The result of an operation is a *new relation*, which may have been formed from one or more *input* relations
  - This property makes the algebra “closed” (all objects in relational algebra are relations)

# Relational Algebra Outline

- Relational Algebra
  - Unary Relational Operations
  - Relational Algebra Operations From Set Theory
  - Binary Relational Operations
  - Additional Relational Operations
  - Examples of Queries in Relational Algebra
- Example Database Application (COMPANY)



# Relational Algebra Overview

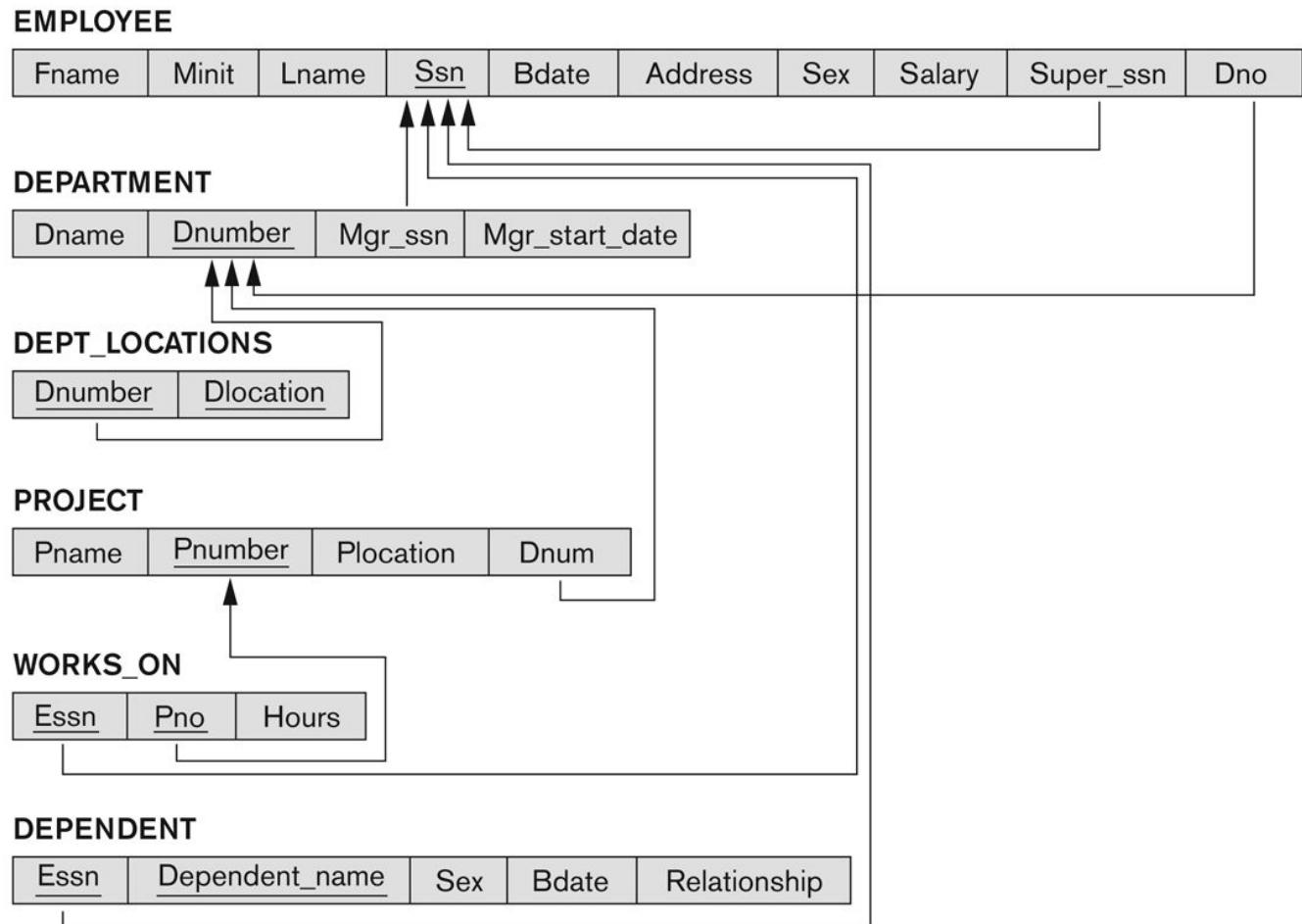
- Relational Algebra consists of several groups of operations
  - Unary Relational Operations
    - SELECT (symbol:  $\sigma$  (sigma))
    - PROJECT (symbol:  $\pi$  (pi))
    - RENAME (symbol:  $\rho$  (rho))
  - Relational Algebra Operations From Set Theory
    - UNION ( $\cup$ ), INTERSECTION ( $\cap$ ), DIFFERENCE (or MINUS,  $-$ )
    - CARTESIAN PRODUCT (  $\times$  )
  - Binary Relational Operations
    - JOIN (several variations of JOIN exist)
    - DIVISION
  - Additional Relational Operations
    - OUTER JOINS, OUTER UNION
    - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

# Database State for COMPANY

- All examples discussed refer to the COMPANY database shown here.

**Figure 5.7**

Referential integrity constraints displayed on the COMPANY relational database schema.



| EMPLOYEE | FNAME    | MINIT | LNAME   | SSN       | BDATE      | ADDRESS                  | SEX | SALARY | SUPERSSN  | DNO |
|----------|----------|-------|---------|-----------|------------|--------------------------|-----|--------|-----------|-----|
|          | John     | B     | Smith   | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX | M   | 30000  | 333445555 | 5   |
|          | Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5   |
|          | Alicia   | J     | Zelaya  | 999887777 | 1968-01-19 | 3321 Castle, Spring, TX  | F   | 25000  | 987654321 | 4   |
|          | Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX  | F   | 43000  | 888665555 | 4   |
|          | Ramesh   | K     | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M   | 38000  | 333445555 | 5   |
|          | Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   | F   | 25000  | 333445555 | 5   |
|          | Ahmad    | V     | Jabbar  | 987987987 | 1969-03-29 | 980 Dallas, Houston, TX  | M   | 25000  | 987654321 | 4   |
|          | James    | E     | Borg    | 888665555 | 1937-11-10 | 450 Stone, Houston, TX   | M   | 55000  | null      | 1   |

| DEPT_LOCATIONS | DNUMBER | DLOCATION |
|----------------|---------|-----------|
|                | 1       | Houston   |
|                | 4       | Stafford  |
|                | 5       | Bellaire  |
|                | 5       | Sugarland |
|                |         | Houston   |

| DEPARTMENT | DNAME          | DNUMBER | MGRSSN    | MGRSTARTDATE |
|------------|----------------|---------|-----------|--------------|
|            | Research       | 5       | 333445555 | 1988-05-22   |
|            | Administration | 4       | 987654321 | 1995-01-01   |
|            | Headquarters   | 1       | 888665555 | 1981-06-19   |

| WORKS_ON | ESSN      | PNO | HOURS |
|----------|-----------|-----|-------|
|          | 123456789 | 1   | 32.5  |
|          | 123456789 | 2   | 7.5   |
|          | 666884444 | 3   | 40.0  |
|          | 453453453 | 1   | 20.0  |
|          | 453453453 | 2   | 20.0  |
|          | 333445555 | 2   | 10.0  |
|          | 333445555 | 3   | 10.0  |
|          | 333445555 | 10  | 10.0  |
|          | 333445555 | 20  | 10.0  |
|          | 999887777 | 30  | 30.0  |
|          | 999887777 | 10  | 10.0  |
|          | 987987987 | 10  | 35.0  |
|          | 987987987 | 30  | 5.0   |
|          | 987654321 | 30  | 20.0  |
|          | 987654321 | 20  | 15.0  |
|          | 888665555 | 20  | null  |

| PROJECT         | PNAME | PNUMBER | PLOCATION | DNUM |
|-----------------|-------|---------|-----------|------|
| ProductX        |       | 1       | Bellaire  | 5    |
| ProductY        |       | 2       | Sugarland | 5    |
| ProductZ        |       | 3       | Houston   | 5    |
| Computerization |       | 10      | Stafford  | 4    |
| Reorganization  |       | 20      | Houston   | 1    |
| Newbenefits     |       | 30      | Stafford  | 4    |

| DEPENDENT | ESSN      | DEPENDENT_NAME | SEX | BDATE      | RELATIONSHIP |
|-----------|-----------|----------------|-----|------------|--------------|
|           | 333445555 | Alice          | F   | 1986-04-05 | DAUGHTER     |
|           | 333445555 | Theodore       | M   | 1983-10-25 | SON          |
|           | 333445555 | Joy            | F   | 1958-05-03 | SPOUSE       |
|           | 987654321 | Abner          | M   | 1942-02-28 | SPOUSE       |
|           | 123456789 | Michael        | M   | 1988-01-04 | SON          |
|           | 123456789 | Alice          | F   | 1988-12-30 | DAUGHTER     |
|           | 123456789 | Elizabeth      | F   | 1967-05-05 | SPOUSE       |

# Unary Relational Operations: SELECT

- The SELECT operation (denoted by  $\sigma$  (sigma)) is used to select a *subset* of the tuples from a relation based on a **selection condition**.
  - The selection condition acts as a **filter**
  - Keeps only those tuples that satisfy the qualifying condition
  - Tuples satisfying the condition are *selected* whereas the other tuples are discarded (*filtered out*)

# Unary Relational Operations: SELECT

- Examples:
  - Select the EMPLOYEE tuples whose department number is 4:

$$\sigma_{DNO = 4} (\text{EMPLOYEE})$$

- Select the employee tuples whose salary is greater than \$30,000:

$$\sigma_{\text{SALARY} > 30,000} (\text{EMPLOYEE})$$

# Unary Relational Operations: SELECT

- In general, the *select* operation is denoted by  $\sigma_{<\text{selection condition}}(R)$  where
  - the symbol  $\sigma$  (sigma) is used to denote the *select* operator
  - the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
  - tuples that make the condition **true** are selected
    - appear in the result of the operation
  - tuples that make the condition **false** are filtered out
    - discarded from the result of the operation

# Unary Relational Operations: PROJECT

- PROJECT Operation is denoted by  $\pi$  (pi)
- This operation keeps certain *columns* (attributes) from a relation and discards the other columns.
  - PROJECT creates a vertical partitioning
    - The list of specified columns (attributes) is kept in each tuple
    - The other attributes in each tuple are discarded
- Example: To list each employee's first and last name and salary, the following is used:

$$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$$

- The general form of the *project* operation is:

$$\pi_{<\text{attribute list}>}(\mathbf{R})$$

- $\pi$  (pi) is the symbol used to represent the *project* operation
  - $<\text{attribute list}>$  is the desired list of attributes from relation R.
- The project operation *removes any duplicate tuples*
  - This is because the result of the *project* operation must be a *set of tuples*
    - Mathematical sets *do not allow* duplicate elements.

- PROJECT Operation Properties
  - The number of tuples in the result of projection  $\pi_{<\text{list}>} (R)$  is always less or equal to the number of tuples in R
    - If the list of attributes includes a *key* of R, then the number of tuples in the result of PROJECT is *equal* to the number of tuples in R
  - PROJECT is *not* commutative
    - $\pi_{<\text{list1}>} (\pi_{<\text{list2}>} (R)) = \pi_{<\text{list1}>} (R)$  as long as  $<\text{list2}>$  contains the attributes in  $<\text{list1}>$

# Examples of applying SELECT and PROJECT operations

**Figure 6.1**

Results of SELECT and PROJECT operations. (a)  $\sigma_{(Dno=4 \text{ AND } \text{Salary}>25000) \text{ OR } (Dno=5 \text{ AND } \text{Salary}>30000)}$  (EMPLOYEE).  
 (b)  $\pi_{\text{Lname}, \text{Fname}, \text{Salary}}(\text{EMPLOYEE})$ . (c)  $\pi_{\text{Sex}, \text{Salary}}(\text{EMPLOYEE})$ .

**(a)**

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                  | Sex | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|--------------------------|-----|--------|-----------|-----|
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5   |
| Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX  | F   | 43000  | 888665555 | 4   |
| Ramesh   | K     | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M   | 38000  | 333445555 | 5   |

**(b)**

| Lname   | Fname    | Salary |
|---------|----------|--------|
| Smith   | John     | 30000  |
| Wong    | Franklin | 40000  |
| Zelaya  | Alicia   | 25000  |
| Wallace | Jennifer | 43000  |
| Narayan | Ramesh   | 38000  |
| English | Joyce    | 25000  |
| Jabbar  | Ahmad    | 25000  |
| Borg    | James    | 55000  |

**(c)**

| Sex | Salary |
|-----|--------|
| M   | 30000  |
| M   | 40000  |
| F   | 25000  |
| F   | 43000  |
| M   | 38000  |
| M   | 25000  |
| M   | 55000  |

# Unary Relational Operations: RENAME

- The RENAME operator is denoted by  $\rho$  (rho)
- In some cases, we may want to *rename* the attributes of a relation or the relation name or both
  - Useful when a query requires multiple operations
  - Necessary in some cases (see JOIN operation later)

# Unary Relational Operations: RENAME (contd.)

- The general RENAME operation  $\rho$  can be expressed by any of the following forms:
  - $\rho_{S(B_1, B_2, \dots, B_n)}(R)$  changes both:
    - the relation name to S, *and*
    - the column (attribute) names to  $B_1, B_2, \dots, B_n$
  - $\rho_S(R)$  changes:
    - the *relation name* only to S
  - $\rho_{(B_1, B_2, \dots, B_n)}(R)$  changes:
    - the *column (attribute) names* only to  $B_1, B_2, \dots, B_n$

# Unary Relational Operations: RENAME (contd.)

- For convenience, we also use a *shorthand* for renaming attributes in an intermediate relation:

Eg:  $\Pi_{\text{FNAME, LNAME, SALARY}}(\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$

- Alternatively we can explicitly show the sequence of operations by giving a name to each intermediate relation

- If we write:

- $\text{DEP5-EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$
- $\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, SALARY}}(\text{DEP5-EMPS})$
- RESULT will have the *same attribute names* as DEP5\_EMPS (same attributes as EMPLOYEE)

- If we write:

- $\text{RESULT} (\text{F, M, L, S, B, A, SX, SAL, SU, DNO}) \leftarrow \pi_{\text{FNAME, LNAME, SALARY}}(\text{DEP5-EMPS})$
- The 10 attributes of DEP5\_EMPS are *renamed* to F, M, L, S, B, A, SX, SAL, SU, DNO, respectively

# Example of applying multiple operations and RENAME

(a)

| Fname    | Lname   | Salary |
|----------|---------|--------|
| John     | Smith   | 30000  |
| Franklin | Wong    | 40000  |
| Ramesh   | Narayan | 38000  |
| Joyce    | English | 25000  |

(b)

TEMP

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                  | Sex | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|--------------------------|-----|--------|-----------|-----|
| John     | B     | Smith   | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX | M   | 30000  | 333445555 | 5   |
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5   |
| Ramesh   | K     | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M   | 38000  | 333445555 | 5   |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   | F   | 25000  | 333445555 | 5   |

R

| First_name | Last_name | Salary |
|------------|-----------|--------|
| John       | Smith     | 30000  |
| Franklin   | Wong      | 40000  |
| Ramesh     | Narayan   | 38000  |
| Joyce      | English   | 25000  |

**Figure 6.2**

Results of a sequence of operations.

(a)  $\pi_{\text{Fname}, \text{Lname}, \text{Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$ .

(b) Using intermediate relations and renaming of attributes.

# Relational Algebra Operations from Set Theory:

- UNION Operation
  - Union operation, denoted by  $\cup$
  - The result of  $R \cup S$ , is a relation that includes all tuples that are either in R or in S or in both R and S
  - Duplicate tuples are eliminated
  - The two operand relations R and S must be “type compatible” (or UNION compatible)
    - R and S must have same number of attributes
    - Each pair of corresponding attributes must be type compatible (have same or compatible domains)

# UNION Example

- Example:
  - To retrieve the social security numbers of all employees who either *work in department 5* (RESULT1 below) or *directly supervise an employee who works in department 5* (RESULT2 below)
  - We can use the UNION operation as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$

$$\text{RESULT1} \leftarrow \pi_{\text{SSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT2(SSN)} \leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

- The union operation produces the tuples that are in either RESULT1 or RESULT2 or both

# Example of the result of a UNION operation

- UNION Example

**Figure 6.3**

Result of the  
UNION operation  
 $\text{RESULT} \leftarrow \text{RESULT1}$   
 $\cup \text{RESULT2}.$

**RESULT1**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |

**RESULT2**

| Ssn       |
|-----------|
| 333445555 |
| 888665555 |

**RESULT**

| Ssn       |
|-----------|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |
| 888665555 |

# Relational Algebra Operations from Set Theory

- Type Compatibility of operands is required for the binary set operation UNION  $\cup$ , (also for INTERSECTION  $\cap$ , and SET DIFFERENCE  $-$ , see next slides)
- $R_1(A_1, A_2, \dots, A_n)$  and  $R_2(B_1, B_2, \dots, B_n)$  are type compatible if:
  - they have the same number of attributes, and
  - the domains of corresponding attributes are type compatible (i.e.  $\text{dom}(A_i) = \text{dom}(B_i)$  for  $i=1, 2, \dots, n$ ).
- The resulting relation for  $R_1 \cup R_2$  (also for  $R_1 \cap R_2$ , or  $R_1 - R_2$ , see next slides) has the same attribute names as the *first* operand relation  $R_1$  (by convention)

# INTERSECTION

- INTERSECTION is denoted by  $\cap$
- The result of the operation  $R \cap S$ , is a relation that includes all tuples that are in both R and S
  - The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be “type compatible”

# SET DIFFERENCE

- SET DIFFERENCE (also called MINUS or EXCEPT) is denoted by –
- The result of  $R - S$ , is a relation that includes all tuples that are in R but not in S
  - The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be “type compatible”

# Example to illustrate the result of UNION, INTERSECT, and DIFFERENCE

(a) STUDENT

| Fn      | Ln      |
|---------|---------|
| Susan   | Yao     |
| Ramesh  | Shah    |
| Johnny  | Kohler  |
| Barbara | Jones   |
| Amy     | Ford    |
| Jimmy   | Wang    |
| Ernest  | Gilbert |

INSTRUCTOR

| Fname   | Lname   |
|---------|---------|
| John    | Smith   |
| Ricardo | Browne  |
| Susan   | Yao     |
| Francis | Johnson |
| Ramesh  | Shah    |

(b)

| Fn      | Ln      |
|---------|---------|
| Susan   | Yao     |
| Ramesh  | Shah    |
| Johnny  | Kohler  |
| Barbara | Jones   |
| Amy     | Ford    |
| Jimmy   | Wang    |
| Ernest  | Gilbert |
| John    | Smith   |
| Ricardo | Browne  |
| Francis | Johnson |

(c)

| Fn     | Ln   |
|--------|------|
| Susan  | Yao  |
| Ramesh | Shah |

(d)

| Fn      | Ln      |
|---------|---------|
| Johnny  | Kohler  |
| Barbara | Jones   |
| Amy     | Ford    |
| Jimmy   | Wang    |
| Ernest  | Gilbert |

(e)

| Fname   | Lname   |
|---------|---------|
| John    | Smith   |
| Ricardo | Browne  |
| Francis | Johnson |

**Figure 6.4**

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT  $\cup$  INSTRUCTOR. (c) STUDENT  $\cap$  INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

# Some properties of UNION, INTERSECT, and DIFFERENCE

- Notice that both union and intersection are *commutative* operations; that is

$$R \cup S = S \cup R, \text{ and } R \cap S = S \cap R$$

- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are *associative* operations; that is

- $R \cup (S \cup T) = (R \cup S) \cup T$
- $(R \cap S) \cap T = R \cap (S \cap T)$

- The minus operation is not commutative; that is, in general

$$R - S \neq S - R$$

# CARTESIAN PRODUCT

- CARTESIAN (or CROSS) PRODUCT Operation
  - This operation is used to combine tuples from two relations in a combinatorial fashion.
  - Denoted by  $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$
  - Result is a relation Q with degree  $n + m$  attributes:
    - $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , in that order.
  - The resulting relation state has one tuple for each combination of tuples—one from R and one from S.
  - Hence, if R has  $n_R$  tuples (denoted as  $|R| = n_R$ ), and S has  $n_S$  tuples, then  $R \times S$  will have  $n_R * n_S$  tuples.
  - The two operands do NOT have to be "type compatible"

# CARTESIAN PRODUCT (cont.)

- Generally, CROSS PRODUCT is not a meaningful operation
  - Can become meaningful when followed by other operations
- Example (not meaningful):
  - $\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{SEX}=\text{'F'}}(\text{EMPLOYEE})$
  - $\text{EMPNAMES} \leftarrow \pi_{\text{FNAME, LNAME, SSN}}(\text{FEMALE\_EMPS})$
  - $\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$
- $\text{EMP\_DEPENDENTS}$  will contain every combination of  $\text{EMPNAMES}$  and  $\text{DEPENDENT}$ 
  - whether or not they are actually related

# CARTESIAN PRODUCT (cont.)

- To keep only combinations where the DEPENDENT is related to the EMPLOYEE, we add a SELECT operation as follows
- Example (meaningful):
  - $\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{SEX}='F'}(\text{EMPLOYEE})$
  - $\text{EMP NAMES} \leftarrow \pi_{\text{FNAME, LNAME, SSN}}(\text{FEMALE\_EMPS})$
  - $\text{EMP\_DEPENDENTS} \leftarrow \text{EMP NAMES} \times \text{DEPENDENT}$
  - $\text{ACTUAL\_DEPS} \leftarrow \sigma_{\text{SSN=ESSN}}(\text{EMP\_DEPENDENTS})$
  - $\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, DEPENDENT\_NAME}}(\text{ACTUAL\_DEPS})$
- RESULT will now contain the name of female employees and their dependents

# Example of applying CARTESIAN PRODUCT

## FEMALE\_EMPS

| Fname    | Minit | Lname   | Ssn       | Bdate      | Address                 | Sex | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|------------|-------------------------|-----|--------|-----------|-----|
| Alicia   | J     | Zelaya  | 999887777 | 1968-07-19 | 3321 Castle, Spring, TX | F   | 25000  | 987654321 | 4   |
| Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX | F   | 43000  | 888665555 | 4   |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX  | F   | 25000  | 333445555 | 5   |

## EMPNAMEs

| Fname    | Lname   | Ssn       |
|----------|---------|-----------|
| Alicia   | Zelaya  | 999887777 |
| Jennifer | Wallace | 987654321 |
| Joyce    | English | 453453453 |

| DEPENDENT | ESSN      | DEPENDENT_NAME | SEX | BDATE      | RELATIONSHIP |
|-----------|-----------|----------------|-----|------------|--------------|
|           | 333445555 | Alice          | F   | 1986-04-05 | DAUGHTER     |
|           | 333445555 | Theodore       | M   | 1983-10-25 | SON          |
|           | 333445555 | Joy            | F   | 1958-05-03 | SPOUSE       |
|           | 987654321 | Abner          | M   | 1942-02-28 | SPOUSE       |
|           | 123456789 | Michael        | M   | 1988-01-04 | SON          |
|           | 123456789 | Alice          | F   | 1988-12-30 | DAUGHTER     |
|           | 123456789 | Elizabeth      | F   | 1967-05-05 | SPOUSE       |

# CARTESIAN PRODUCT Example

contd..

**EMP\_DEPENDENTS**

| Fname    | Lname   | Ssn       | Essn      | Dependent_name | Sex | Bdate      | ... |
|----------|---------|-----------|-----------|----------------|-----|------------|-----|
| Alicia   | Zelaya  | 999887777 | 333445555 | Alice          | F   | 1986-04-05 | ... |
| Alicia   | Zelaya  | 999887777 | 333445555 | Theodore       | M   | 1983-10-25 | ... |
| Alicia   | Zelaya  | 999887777 | 333445555 | Joy            | F   | 1958-05-03 | ... |
| Alicia   | Zelaya  | 999887777 | 987654321 | Abner          | M   | 1942-02-28 | ... |
| Alicia   | Zelaya  | 999887777 | 123456789 | Michael        | M   | 1988-01-04 | ... |
| Alicia   | Zelaya  | 999887777 | 123456789 | Alice          | F   | 1988-12-30 | ... |
| Alicia   | Zelaya  | 999887777 | 123456789 | Elizabeth      | F   | 1967-05-05 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Alice          | F   | 1986-04-05 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Theodore       | M   | 1983-10-25 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Joy            | F   | 1958-05-03 | ... |
| Jennifer | Wallace | 987654321 | 987654321 | Abner          | M   | 1942-02-28 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Michael        | M   | 1988-01-04 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Alice          | F   | 1988-12-30 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Elizabeth      | F   | 1967-05-05 | ... |
| Joyce    | English | 453453453 | 333445555 | Alice          | F   | 1986-04-05 | ... |
| Joyce    | English | 453453453 | 333445555 | Theodore       | M   | 1983-10-25 | ... |
| Joyce    | English | 453453453 | 333445555 | Joy            | F   | 1958-05-03 | ... |
| Joyce    | English | 453453453 | 987654321 | Abner          | M   | 1942-02-28 | ... |
| Joyce    | English | 453453453 | 123456789 | Michael        | M   | 1988-01-04 | ... |
| Joyce    | English | 453453453 | 123456789 | Alice          | F   | 1988-12-30 | ... |
| Joyce    | English | 453453453 | 123456789 | Elizabeth      | F   | 1967-05-05 | ... |

# CARTESIAN PRODUCT Example

contd..

## ACTUAL\_DEPENDENTS

| Fname    | Lname   | Ssn       | Essn      | Dependent_name | Sex | Bdate      | ... |
|----------|---------|-----------|-----------|----------------|-----|------------|-----|
| Jennifer | Wallace | 987654321 | 987654321 | Abner          | M   | 1942-02-28 | ... |

## RESULT

| Fname    | Lname   | Dependent_name |
|----------|---------|----------------|
| Jennifer | Wallace | Abner          |

# Binary Relational Operations: JOIN

- JOIN Operation (denoted by  $\bowtie$ )
- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
  - The sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations
  - A special operation, called JOIN combines this sequence into a single operation
  - This operation is very important for any relational database with more than a single relation, because it allows us *combine related tuples* from various relations
  - The general form of a join operation on two relations  $R(A_1, A_2, \dots, A_n)$  and  $S(B_1, B_2, \dots, B_m)$  is:
- R  $\bowtie_{\text{join condition}} S$
- where R and S can be any relations that result from general *relational algebra expressions*.

# Binary Relational Operations: JOIN (cont.)

- Example: Suppose that we want to retrieve the name of the manager of each department.
  - To get the manager's name, we need to combine each DEPARTMENT tuple with the EMPLOYEE tuple whose SSN value matches the MGRSSN value in the department tuple.
  - We do this by using the join ( $\bowtie$ ) operation.
  - $\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{MGRSSN}=\text{SSN}} \text{EMPLOYEE}$
- MGRSSN=SSN is the join condition
  - Combines each department record with the employee who manages the department
  - The join condition can also be specified as  $\text{DEPARTMENT.MGRSSN} = \text{EMPLOYEE.SSN}$

# Example of applying the JOIN operation

| EMPLOYEE | FNAME | MINIT   | LNAME | SSN       | BDATE      | ADDRESS                  | SEX | SALARY | SUPERSSN  | DN |
|----------|-------|---------|-------|-----------|------------|--------------------------|-----|--------|-----------|----|
| John     | B     | Smith   |       | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX | M   | 30000  | 333445555 | 5  |
| Franklin | T     | Wong    |       | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    | M   | 40000  | 888665555 | 5  |
| Alicia   | J     | Zelaya  |       | 999887777 | 1968-01-19 | 3321 Castle, Spring, TX  | F   | 25000  | 987654321 | 4  |
| Jennifer | S     | Wallace |       | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX  | F   | 43000  | 888665555 | 4  |
| Ramesh   | K     | Narayan |       | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M   | 38000  | 333445555 | 5  |
| Joyce    | A     | English |       | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   | F   | 25000  | 333445555 | 5  |
| Ahmad    | V     | Jabbar  |       | 987987987 | 1969-03-29 | 980 Dallas, Houston, TX  | M   | 25000  | 987654321 | 4  |
| James    | E     | Borg    |       | 888665555 | 1937-11-10 | 450 Stone, Houston, TX   | M   | 55000  | null      | 1  |

| DEPARTMENT | DNAME          | DNUMBER | MGRSSN    | MGRSTARTDATE |
|------------|----------------|---------|-----------|--------------|
|            | Research       | 5       | 333445555 | 1988-05-22   |
|            | Administration | 4       | 987654321 | 1995-01-01   |
|            | Headquarters   | 1       | 888665555 | 1981-06-19   |

## DEPT\_MGR

| Dname          | Dnumber | Mgr_ssn   | ... | Fname    | Minit | Lname   | Ssn       | ... |
|----------------|---------|-----------|-----|----------|-------|---------|-----------|-----|
| Research       | 5       | 333445555 | ... | Franklin | T     | Wong    | 333445555 | ... |
| Administration | 4       | 987654321 | ... | Jennifer | S     | Wallace | 987654321 | ... |
| Headquarters   | 1       | 888665555 | ... | James    | E     | Borg    | 888665555 | ... |

**Figure 6.6**

Result of the JOIN operation

# Some properties of JOIN

- Consider the following JOIN operation:
  - $R(A_1, A_2, \dots, A_n) \bowtie_{R.A_i=S.B_j} S(B_1, B_2, \dots, B_m)$
  - Result is a relation Q with degree  $n + m$  attributes:
    - $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , in that order.
  - The resulting relation state has one tuple for each combination of tuples—r from R and s from S, but *only if they satisfy the join condition  $r[A_i]=s[B_j]$*
  - Hence, if R has  $n_R$  tuples, and S has  $n_S$  tuples, then the join result will generally have *less than  $n_R * n_S$*  tuples.
  - Only related tuples (based on the join condition) will appear in the result

# Some properties of JOIN

- The general case of JOIN operation is called a Theta-join:  $R \bowtie_{\theta} S$
- The join condition is called *theta*
- *Theta* can be any general Boolean expression on the attributes of R and S; for example:  
$$R.A_i < S.B_j \text{ AND } (R.A_k = S.B_l \text{ OR } R.A_p < S.B_q)$$
- Most join conditions involve one or more equality conditions “AND”ed together; for example:

$R.A_i = S.B_j \text{ AND } R.A_k = S.B_l \text{ AND } R.A_p = S.B_q$

# Binary Relational Operations: EQUI-JOIN

- EQUI-JOIN Operation
- The most common use of join involves join conditions with *equality comparisons* only
- Such a join, where the only comparison operator used is  $=$ , is called an EQUI-JOIN.
  - In the result of an EQUI-JOIN we always have one or more pairs of attributes (whose names need not be identical) that have identical values in every tuple.
  - The JOIN seen in the previous example was an EQUI-JOIN.

# Binary Relational Operations: NATURAL JOIN

- NATURAL JOIN Operation
  - Another variation of JOIN called NATURAL JOIN (denoted by \*) was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.
    - because one of each pair of attributes with identical values is superfluous
  - The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, *have the same name* in both relations
  - If this is not the case, a renaming operation is applied first.

# NATURAL JOIN (contd.)

- Example: To apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT\_LOCATIONS, it is sufficient to write:

DEPT\_LOCS  $\leftarrow$  DEPARTMENT \* DEPT\_LOCATIONS

- Only attribute with the same name is DNUMBER
- An implicit join condition is created based on this attribute:  

$$\text{DEPARTMENT.DNUMBER} = \text{DEPT\_LOCATIONS.DNUMBER}$$

- Another example:  $Q \leftarrow R(A,B,C,D) * S(C,D,E)$ 
  - The implicit join condition includes *each pair* of attributes with the same name, “AND”ed together:  

$$R.C=S.C \text{ AND } R.D=S.D$$
  - Result keeps only one attribute of each such pair:  

$$Q(A,B,C,D,E)$$

# Example of NATURAL JOIN operation

Consider the following example to understand natural Joins.

## EMPLOYEE

| EMP_ID | EMP_NAME |
|--------|----------|
| 1      | Ram      |
| 2      | Varun    |
| 3      | Lakshmi  |

## SALARY

| EMP_ID | SALARY |
|--------|--------|
| 1      | 50000  |
| 2      | 30000  |
| 3      | 25000  |

# Example of NATURAL JOIN operation

$\Pi \text{EMP\_NAME, SALARY} (\text{EMPLOYEE} \bowtie \text{SALARY})$

## Output:

| EMP_NAME | SALARY |
|----------|--------|
| Ram      | 50000  |
| Varun    | 30000  |
| Lakshmi  | 25000  |

| PROJECT         | PNAME | PNUMBER   | PLOCATION | DNUM |
|-----------------|-------|-----------|-----------|------|
| ProductX        | 1     | Bellaire  | 5         |      |
| ProductY        | 2     | Sugarland | 5         |      |
| ProductZ        | 3     | Houston   | 5         |      |
| Computerization | 10    | Stafford  | 4         |      |
| Reorganization  | 20    | Houston   | 1         |      |
| Newbenefits     | 30    | Stafford  | 4         |      |

| DEPARTMENT     | DNAME | DNUMBER   | MGRSSN | MGRSTARTDATE |
|----------------|-------|-----------|--------|--------------|
| Research       | 5     | 333445555 |        | 1988-05-22   |
| Administration | 4     | 987654321 |        | 1995-01-01   |
| Headquarters   | 1     | 888665555 |        | 1981-06-19   |

| DEPT_LOCATIONS | DNUMBER | DLOCATION |
|----------------|---------|-----------|
|                | 1       | Houston   |
|                | 4       | Stafford  |
|                | 5       | Bellaire  |
|                | 5       | Sugarland |
|                |         | Houston   |

# Example of NATURAL JOIN operation

(a)

## PROJ\_DEPT

| Pname           | Pnumber | Plocation | Dnum | Dname          | Mgr_ssn   | Mgr_start_date |
|-----------------|---------|-----------|------|----------------|-----------|----------------|
| ProductX        | 1       | Bellaire  | 5    | Research       | 333445555 | 1988-05-22     |
| ProductY        | 2       | Sugarland | 5    | Research       | 333445555 | 1988-05-22     |
| ProductZ        | 3       | Houston   | 5    | Research       | 333445555 | 1988-05-22     |
| Computerization | 10      | Stafford  | 4    | Administration | 987654321 | 1995-01-01     |
| Reorganization  | 20      | Houston   | 1    | Headquarters   | 888665555 | 1981-06-19     |
| Newbenefits     | 30      | Stafford  | 4    | Administration | 987654321 | 1995-01-01     |

(b)

## DEPT\_LOCS

| Dname          | Dnumber | Mgr_ssn   | Mgr_start_date | Location  |
|----------------|---------|-----------|----------------|-----------|
| Headquarters   | 1       | 888665555 | 1981-06-19     | Houston   |
| Administration | 4       | 987654321 | 1995-01-01     | Stafford  |
| Research       | 5       | 333445555 | 1988-05-22     | Bellaire  |
| Research       | 5       | 333445555 | 1988-05-22     | Sugarland |
| Research       | 5       | 333445555 | 1988-05-22     | Houston   |

**Figure 6.7**

Results of two NATURAL JOIN operations.

- (a) PROJ\_DEPT  $\leftarrow$  PROJECT \* DEPT.
- (b) DEPT\_LOCS  $\leftarrow$  DEPARTMENT \* DEPT\_LOCATIONS.

# Complete Set of Relational Operations

- The set of operations including SELECT  $\sigma$ , PROJECT  $\pi$ , UNION  $\cup$ , DIFFERENCE  $-$ , RENAME  $\rho$ , and CARTESIAN PRODUCT  $\times$  is called a *complete set* because any other relational algebra expression can be expressed by a combination of these five operations.
- For example:
  - $R \cap S = (R \cup S) - ((R - S) \cup (S - R))$
  - $R \bowtie_{\text{join condition}} S = \sigma_{\text{join condition}} (R \times S)$

# Binary Relational Operations: DIVISION

- DIVISION Operation
  - The division operation is applied to two relations
  - $R(Z) \div S(X)$ , where  $X \subset Z$ . Let  $Y = Z - X$  (and hence  $Z = X \cup Y$ ); that is, let  $Y$  be the set of attributes of  $R$  that are not attributes of  $S$ .
  - The result of DIVISION is a relation  $T(Y)$  that includes a tuple  $t$  if tuples  $t_R$  appear in  $R$  with  $t_R[Y] = t$ , and with
    - $t_R[X] = t_s$  for every tuple  $t_s$  in  $S$ .
  - ⊗ – For a tuple  $t$  to appear in the result  $T$  of the DIVISION, the values in  $t$  must appear in  $R$  in combination with *every* tuple in  $S$ .

# Example of DIVISION

(a)

**SSN\_PNOS**

| Essn      | Pno |
|-----------|-----|
| 123456789 | 1   |
| 123456789 | 2   |
| 666884444 | 3   |
| 453453453 | 1   |
| 453453453 | 2   |
| 333445555 | 2   |
| 333445555 | 3   |
| 333445555 | 10  |
| 333445555 | 20  |
| 999887777 | 30  |
| 999887777 | 10  |
| 987987987 | 10  |
| 987987987 | 30  |
| 987654321 | 30  |
| 987654321 | 20  |
| 888665555 | 20  |

**SMITH\_PNOS**

| Pno |
|-----|
| 1   |
| 2   |

(b)

**R**

| A  | B  |
|----|----|
| a1 | b1 |
| a2 | b1 |
| a3 | b1 |
| a4 | b1 |
| a1 | b2 |
| a3 | b2 |
| a2 | b3 |
| a3 | b3 |
| a4 | b3 |
| a1 | b4 |
| a2 | b4 |
| a3 | b4 |

**S**

| A  |
|----|
| a1 |
| a2 |
| a3 |

**T**

| B  |
|----|
| b1 |
| b4 |

**SSNS**

| Ssn       |
|-----------|
| 123456789 |
| 453453453 |

**Figure 6.8**

The DIVISION operation. (a) Dividing SSN\_PNOS by SMITH\_PNOS. (b)  $T \leftarrow R \div S$ .

# Example of DIVISION

## Division: example

- R

| Lecturer | Module    |
|----------|-----------|
| Brown    | Compilers |
| Brown    | Databases |
| Green    | Prolog    |
| Green    | Databases |
| Lewis    | Prolog    |
| Smith    | Databases |

- S

| Subject |
|---------|
| Prolog  |

- R | S

| Lecturer |
|----------|
| Green    |
| Lewis    |

Fig1: Example 1

- R

| Lecturer | Module    |
|----------|-----------|
| Brown    | Compilers |
| Brown    | Databases |
| Green    | Prolog    |
| Green    | Databases |
| Lewis    | Prolog    |
| Smith    | Databases |

- S

| Subject   |
|-----------|
| Databases |
| Prolog    |

- R | S

| Lecturer |
|----------|
| Green    |

Fig 2: Example 2

# Example of DIVISION

**A**

| <i>sno</i> | <i>pno</i> |
|------------|------------|
| s1         | p1         |
| s1         | p2         |
| s1         | p3         |
| s1         | p4         |
| s2         | p1         |
| s2         | p2         |
| s3         | p2         |
| s4         | p2         |
| s4         | p4         |

**B1**

| <i>pno</i> |
|------------|
| p2         |

**B2**

| <i>pno</i> |
|------------|
| p2         |
| p4         |

**B3**

| <i>pno</i> |
|------------|
| p1         |
| p2         |
| p4         |

**A/B1**

| <i>sno</i> |
|------------|
| s1         |
| s2         |
| s3         |
| s4         |

**A/B2**

| <i>sno</i> |
|------------|
| s1         |
| s4         |

**A/B3**

| <i>sno</i> |
|------------|
| s1         |

| Symbol (Name)                   | Example of Use                                                                                                                                                                      |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\sigma$<br>(Selection)         | $\sigma_{\text{salary} >= 85000}(\text{instructor})$<br>Return rows of the input relation that satisfy the predicate.                                                               |
| $\Pi$<br>(Projection)           | $\Pi_{ID, \text{salary}}(\text{instructor})$<br>Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.                           |
| $\bowtie$<br>(Natural join)     | $\text{instructor} \bowtie \text{department}$<br>Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.              |
| $\times$<br>(Cartesian product) | $\text{instructor} \times \text{department}$<br>Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes) |
| $\cup$<br>(Union)               | $\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$<br>Output the union of tuples from the two input relations.                                                         |

# Recap of Relational Algebra Operations

**Table 6.1**

Operations of Relational Algebra

| Operation         | Purpose                                                                                                                                                                                    | Notation                                                                                                                              |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| SELECT            | Selects all tuples that satisfy the selection condition from a relation $R$ .                                                                                                              | $\sigma_{\text{<selection condition>}}(R)$                                                                                            |
| PROJECT           | Produces a new relation with only some of the attributes of $R$ , and removes duplicate tuples.                                                                                            | $\pi_{\text{<attribute list>}}(R)$                                                                                                    |
| THETA JOIN        | Produces all combinations of tuples from $R_1$ and $R_2$ that satisfy the join condition.                                                                                                  | $R_1 \bowtie_{\text{<join condition>}} R_2$                                                                                           |
| EQUIJOIN          | Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition with only equality comparisons.                                                                 | $R_1 \bowtie_{\text{<join condition>}} R_2,$<br>OR $R_1 \bowtie_{(\text{<join attributes 1>}, \text{<join attributes 2>})} R_2$       |
| NATURAL JOIN      | Same as EQUIJOIN except that the join attributes of $R_2$ are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all. | $R_1 *_{\text{<join condition>}} R_2,$<br>OR $R_1 *_{(\text{<join attributes 1>}, \text{<join attributes 2>})} R_2$<br>OR $R_1 * R_2$ |
| UNION             | Produces a relation that includes all the tuples in $R_1$ or $R_2$ or both $R_1$ and $R_2$ ; $R_1$ and $R_2$ must be union compatible.                                                     | $R_1 \cup R_2$                                                                                                                        |
| INTERSECTION      | Produces a relation that includes all the tuples in both $R_1$ and $R_2$ ; $R_1$ and $R_2$ must be union compatible.                                                                       | $R_1 \cap R_2$                                                                                                                        |
| DIFFERENCE        | Produces a relation that includes all the tuples in $R_1$ that are not in $R_2$ ; $R_1$ and $R_2$ must be union compatible.                                                                | $R_1 - R_2$                                                                                                                           |
| CARTESIAN PRODUCT | Produces a relation that has the attributes of $R_1$ and $R_2$ and includes as tuples all possible combinations of tuples from $R_1$ and $R_2$ .                                           | $R_1 \times R_2$                                                                                                                      |
| DIVISION          | Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in $R_1$ in combination with every tuple from $R_2(Y)$ , where $Z = X \cup Y$ .                         | $R_1(Z) \div R_2(Y)$                                                                                                                  |

# Additional Relational Operations: Aggregate Functions and Grouping

- A type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database.
- Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples.
  - These functions are used in simple statistical queries that summarize information from the database tuples.
- Common functions applied to collections of numeric values include
  - **SUM, AVERAGE, MAXIMUM, and MINIMUM.**
- The COUNT function is used for counting tuples or values.

# Aggregate Function Operation

- Use of the Aggregate Functional operation  $\mathcal{F}$ 
  - $\mathcal{F}_{\text{MAX Salary}}(\text{EMPLOYEE})$  retrieves the maximum salary value from the EMPLOYEE relation
  - $\mathcal{F}_{\text{MIN Salary}}(\text{EMPLOYEE})$  retrieves the minimum Salary value from the EMPLOYEE relation
  - $\mathcal{F}_{\text{SUM Salary}}(\text{EMPLOYEE})$  retrieves the sum of the Salary from the EMPLOYEE relation
  - $\mathcal{F}_{\text{COUNT SSN, AVERAGE Salary}}(\text{EMPLOYEE})$  computes the count (number) of employees and their average salary
    - Note: count just counts the number of rows, without removing duplicates

# Using Grouping with Aggregation

- The previous examples all summarized one or more attributes for a set of tuples
  - Maximum Salary or Count (number of) Ssn
- Grouping can be combined with Aggregate Functions
- Example: For each department, retrieve the DNO, COUNT SSN, and AVERAGE SALARY
- A variation of aggregate operation  $\mathcal{F}$  allows this:
  - Grouping attribute placed to left of symbol
  - Aggregate functions to right of symbol
  - $DNO \mathcal{F} COUNT\ SSN, AVERAGE\ Salary$  (EMPLOYEE)
- Above operation groups employees by DNO (department number) and computes the count of employees and average salary per department

# Examples: Aggregate functions and grouping

**Figure 6.10**

The aggregate function operation.

- (a)  $\rho_{R(Dno, No\_of\_employees, Average\_sal)}(\exists_{Dno} \text{COUNT Ssn, AVERAGE Salary } (\text{EMPLOYEE}))$ .
- (b)  $\exists_{Dno} \text{COUNT Ssn, AVERAGE Salary } (\text{EMPLOYEE})$ .
- (c)  $\exists \text{COUNT Ssn, AVERAGE Salary } (\text{EMPLOYEE})$ .

**R**

**(a)**

| Dno | No_of_employees | Average_sal |
|-----|-----------------|-------------|
| 5   | 4               | 33250       |
| 4   | 3               | 31000       |
| 1   | 1               | 55000       |

**(b)**

| Dno | Count_ssn | Average_salary |
|-----|-----------|----------------|
| 5   | 4         | 33250          |
| 4   | 3         | 31000          |
| 1   | 1         | 55000          |

**(c)**

| Count_ssn | Average_salary |
|-----------|----------------|
| 8         | 35125          |

# Illustrating aggregate functions and grouping

**Figure 8.6**

Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

| Fname    | Minit | Lname   | Ssn       | ... | Salary | Super_ssn | Dno | Dno | Count (*)     | Avg (Salary) |
|----------|-------|---------|-----------|-----|--------|-----------|-----|-----|---------------|--------------|
| John     | B     | Smith   | 123456789 | ... | 30000  | 333445555 | 5   | 5   | 4             | 33250        |
| Franklin | T     | Wong    | 333445555 |     | 40000  | 888665555 | 5   |     | 3             | 31000        |
| Ramesh   | K     | Narayan | 666884444 |     | 38000  | 333445555 | 5   |     | 1             | 55000        |
| Joyce    | A     | English | 453453453 |     | 25000  | 333445555 | 5   | 4   | Result of Q24 |              |
| Alicia   | J     | Zelaya  | 999887777 |     | 25000  | 987654321 | 4   |     |               |              |
| Jennifer | S     | Wallace | 987654321 |     | 43000  | 888665555 | 4   |     |               |              |
| Ahmad    | V     | Jabbar  | 987987987 |     | 25000  | 987654321 | 4   |     |               |              |
| James    | E     | Bong    | 888665555 |     | 55000  | NULL      | 1   |     |               |              |

Grouping EMPLOYEE tuples by the value of Dno

# Additional Relational Operations (cont.)

- Recursive Closure Operations
  - Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure**.
    - This operation is applied to a **recursive relationship**.
  - An example of a recursive operation is to retrieve all SUPERVISEES of an EMPLOYEE  $e$  at all levels — that is, all EMPLOYEE  $e'$  directly supervised by  $e$ ; all employees  $e''$  directly supervised by each employee  $e'$ ; all employees  $e'''$  directly supervised by each employee  $e''$ ; and so on.

# Additional Relational Operations (cont.)

- Although it is possible to retrieve employees at each level and then take their union, we cannot, in general, specify a query such as “retrieve the supervisees of ‘James Borg’ at all levels” without utilizing a looping mechanism.
  - The SQL3 standard includes syntax for recursive closure.

# Additional Relational Operations (cont.)

(Borg's SSN is 888665555)

(SSN) (SUPERSSN)

| SUPERVISION | SSN1      | SSN2      |
|-------------|-----------|-----------|
|             | 123456789 | 333445555 |
|             | 333445555 | 888665555 |
|             | 999887777 | 987654321 |
|             | 987654321 | 888665555 |
|             | 666884444 | 333445555 |
|             | 453453453 | 333445555 |
|             | 987987987 | 987654321 |
|             |           |           |

|          |           |
|----------|-----------|
| RESULT 1 | SSN       |
|          | 333445555 |
|          | 987654321 |

(Supervised by Borg)

| RESULT 2 | SSN       |
|----------|-----------|
|          | 123456789 |
|          | 999887777 |
|          | 666884444 |
|          | 453453453 |
|          | 987987987 |

(Supervised by Borg's subordinates)

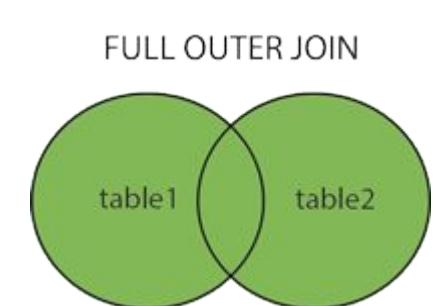
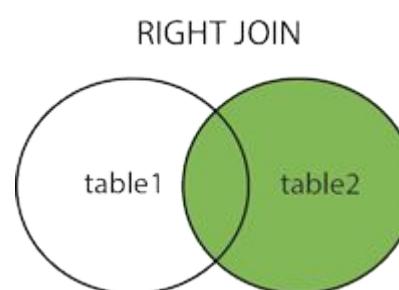
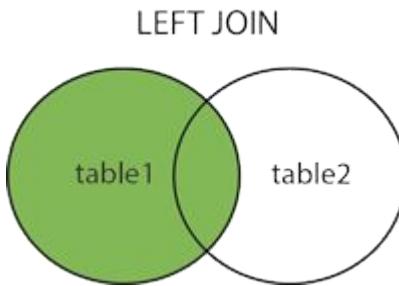
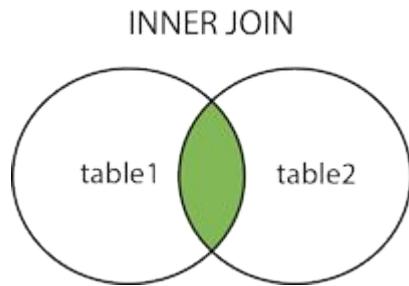
| RESULT | SSN       |
|--------|-----------|
|        | 123456789 |
|        | 999887777 |
|        | 666884444 |
|        | 453453453 |
|        | 987987987 |
|        | 333445555 |
|        | 987654321 |

(RESULT1  $\cup$  RESULT2)

# Different Types of SQL JOINS

Here are the different types of the JOINS in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



# 1. INNER JOIN Example

- SQL statement selects all orders with customer information:

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID =
Customers.CustomerID;
```

## 2. JOIN Three Tables

- ```
SELECT Orders.OrderID, Customers.CustomerName,
Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID =
Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID =
Shippers.ShipperID);
```

# OUTER JOIN

- The OUTER JOIN Operation
  - In NATURAL JOIN and EQUIJOIN, tuples without a *matching* (or *related*) tuple are eliminated from the join result
    - Tuples with null in the join attributes are also eliminated
    - This amounts to loss of information.
  - A set of operations, called OUTER joins, can be used when we want to keep all the tuples in R, or all those in S, or all those in both relations in the result of the join, regardless of whether or not they have matching tuples in the other relation.



# Additional Relational Operations (cont.)

## 1. Left outer join

- The left outer join operation keeps every tuple in the first or left relation R in  $R \bowtie S$ . If no matching tuple is found in S, then the attributes of S in the join result are filled or “padded” with null values.

## 2. Right outer join

- A similar operation, right outer join, keeps every tuple in the second or right relation S in the result of  $R \bowtie S$ .

## 3. Full outer join

- A third operation, full outer join, denoted by  $\bowtie$  keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with null values as needed.

# Additional Relational Operations (cont.)

## RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

**Figure 6.12**

The result of a LEFT OUTER JOIN operation.

join the two tables Orders and Customers, using the CustomerID field in both tables as the relationship between the two tables.

```
SELECT * FROM Orders LEFT JOIN Customers ON
Orders.CustomerID=Customers.CustomerID;
```

# RIGHT OUTER JOIN EXAMPLE

## RIGHT JOIN Syntax

- ```
SELECT column_name(s)
      FROM table1
      RIGHT JOIN table2
      ON table1.column_name = table2.column_name;
```

## Example

- SQL statement to return all employees, and any orders they might have placed
- ```
SELECT Orders.OrderID, Employees.LastName,
          Employees.FirstName
      FROM Orders
      RIGHT JOIN Employees ON Orders.EmployeeID =
          Employees.EmployeeID
      ORDER BY Orders.OrderID;
```

# Additional Relational Operations (cont.)

- OUTER UNION Operations
  - The outer union operation was developed to take the union of tuples from two relations if the relations are *not type compatible*.
  - This operation will take the union of tuples in two relations  $R(X, Y)$  and  $S(X, Z)$  that are **partially compatible**, meaning that only some of their attributes, say  $X$ , are type compatible.
  - The attributes that are type compatible are represented only once in the result, and those attributes that are not type compatible from either relation are also kept in the result relation  $T(X, Y, Z)$ .

# Additional Relational Operations (cont.)

- Example: An outer union can be applied to two relations whose schemas are STUDENT(Name, SSN, Department, Advisor) and INSTRUCTOR(Name, SSN, Department, Rank).
  - Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, SSN, Department.
  - If a student is also an instructor, both Advisor and Rank will have a value; otherwise, one of these two attributes will be null.
  - The result relation STUDENT\_OR\_INSTRUCTOR will have the following attributes:

**STUDENT\_OR\_INSTRUCTOR (Name, SSN,  
Department, Advisor, Rank)**

# Examples of Queries in Relational Algebra

- **Q1: Retrieve the name and address of all employees who work for the ‘Research’ department.**

RESEARCH\_DEPT  $\leftarrow \sigma_{\text{DNAME}=\text{'Research'}}(\text{DEPARTMENT})$

RESEARCH\_EMPS  $\leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{DNUMBER} = \text{DNOEMPLOYEE}} \text{EMPLOYEE})$

RESULT  $\leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{ADDRESS}}(\text{RESEARCH\_EMPS})$

- **Q6: Retrieve the names of employees who have no dependents.**

ALL\_EMPS  $\leftarrow \pi_{\text{SSN}}(\text{EMPLOYEE})$

EMPS\_WITH\_DEPS(SSN)  $\leftarrow \pi_{\text{ESSN}}(\text{DEPENDENT})$

EMPS\_WITHOUT\_DEPS  $\leftarrow (\text{ALL\_EMPS} - \text{EMPS\_WITH\_DEPS})$

RESULT  $\leftarrow \pi_{\text{LNAME}, \text{FNAME}}(\text{EMPS\_WITHOUT\_DEPS} * \text{EMPLOYEE})$



# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module I – Part 2**  
INTRODUCTION TO E-R MODEL

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

# Syllabus: Module 1- part 2

## Module 1: INTRODUCTION TO E-R MODEL

- **Entity-Relationship Model:** Basic concepts, Design Issues, Mapping Constraints, Keys, Entity Relationship Diagram, Weak Entity Sets, Relationships of degree greater than 2 (Reading: Elmasri Navathe, Ch. 7.1-7.8)

# **Introduction To E-R Model**

INTRODUCTION  
**Entity-Relationship Model**

# What is ER Diagram?

- **ER Diagram** stands for Entity Relationship Diagram,
- ERD is a diagram that displays the relationship of entity sets stored in a database.
- ER diagrams help to explain the logical structure of databases.
- ER diagrams are created based on three basic concepts: **entities, attributes and relationships**.
- ER Diagrams contain different symbols:
  - Rectangles to represent entities,
  - Ovals to define attributes
  - Diamond shapes to represent relationships.
- ER diagrams are translatable into relational tables which allows you to build databases quickly

# What is ER Model?

- Entity Relationship Model is a high-level conceptual data model diagram.
- The ER Model represents real-world entities and the relationships between them.
- ER Modeling helps you to analyze data requirements systematically to produce a well-designed database.
- So, it is considered a best practice to complete ER modeling before implementing your database.
- ERD Provide a preview of how all your tables should connect, what fields are going to be on each table.

# ER Diagrams Symbols & Notations

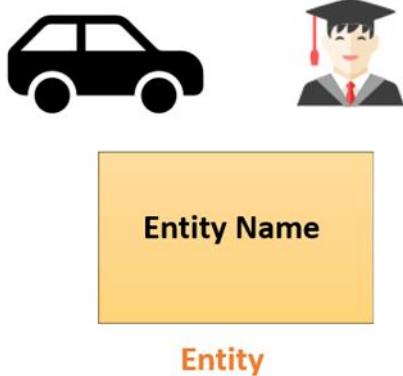
Following are the main components and its symbols in ER Diagrams:

- **Rectangles:** Symbol represents entity types
- **Ellipses :** Symbol represent attributes
- **Diamonds:** This symbol represents relationship types
- **Lines:** It links attributes to entity types and entity types with other relationship types
- **Primary key:** attributes are underlined
- **Double Ellipses:** Represent multi-valued attributes



# Components of the ER Diagram

- Entities
- Attributes
- Relationships



Person, place, object, event or concept about which data is to be maintained  
**Example:** Car, Student



**Attribute**  
 Property or characteristic of an entity  
**Example:** Color of car Entity Name of Student Entity



**Relation**



Association between the instances of one or more entity types  
**Example:** Blue Car Belongs to Student Jack



**1. Entities** –An entity is a ‘thing’ or ‘object’ in the real world that is distinguishable from all other objects and can be represented in the database.

- An entity is represented as rectangle in an ER diagram.
- (eg- EMPLOYEE John Abraham, DEPARTMENT Computer Science)

**2. Attributes**- Properties used to describe an Entity

- (eg- EMPLOYEE entity may have a Id, Name, Designation, Address, Birth date)
- A specific entity will have a value for each of its attributes (eg- EMPLOYEE entity may have Name=‘John Abraham’, Id=‘E0025’, Designation=‘Lecturer’, Address=‘Flat 321, Lane 3, Arakkunnam’, Birthdate=’07-MAR-60’)

# Weak Entity:

- An entity that cannot be uniquely identified by its own attributes and relies on the relationship with other entity is called weak entity.
- The weak entity is represented by a double rectangle.
- For example – a bank account cannot be uniquely identified without knowing the bank to which the account belongs, so bank account is a weak entity.



# Types of Attributes used in E-R Models

- 1. Simple:** - Each entity has a single atomic value for the attribute. For example Sex.
- 2. Composite:**-The attribute can be divided into subparts. For example, Address (Apt#, House#, Street, City, State, ZipCode, Country) or Name (First Name, Middle Name, Last Name). Composite attributes help us to group together related attributes making the modeling cleaner.
- 3. Single Valued:**- They are attributes that have only a single value. Example: ID is a single valued attribute of a person.
- 4. Multi-valued:**- These attributes have multiple values. For example: phone number- a person can have more than one number.
- 5. Derived:** - The value of the attribute can be derived from the values of other attributes. Eg: The PERSON entity's attribute age can be derived from the Date-of-birth attribute. The date-of-birth attribute is called a **stored attribute**.

# Types of Attributes

**6. Null Value:** - These attributes may not have an applicable value. Eg: Home Phone number. A null value could indicate that either value is missing or is not known.

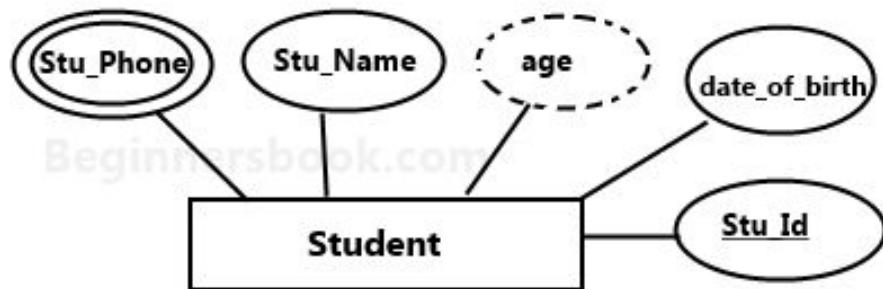
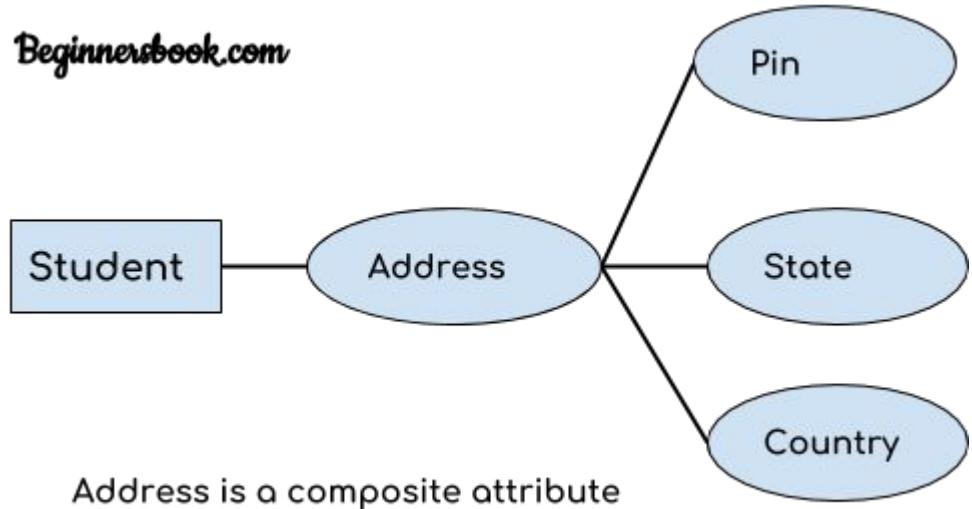
## 7. Key Attribute-

- It is an attribute of an Entity whose value can be used to identify each entity uniquely (uniqueness constraint).
- The attribute is underlined in the ER diagram. (Eg – roll number uniquely identifies each student in a class).
- In certain cases a combination of two or more attributes can describe each entity uniquely.
- **Value Sets (domains) of Attributes**-It specifies the set of values that may be assigned to that attribute for each individual entity.

(Eg- Employee age of a Company could have the value set between 18 and 57)

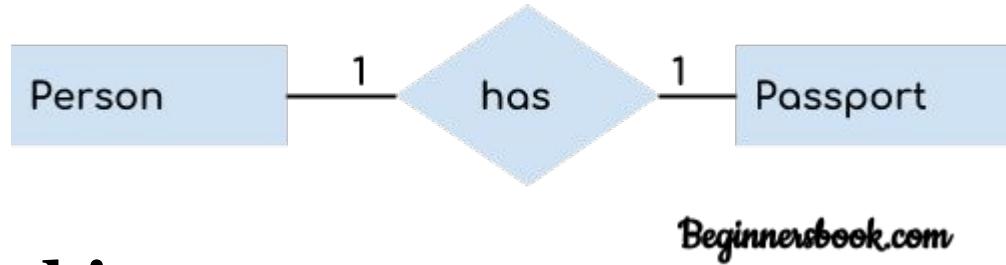
# Types of Attributes : Example

- Composite Attribute:  
Address
- Multi valued : Stu\_Phone
- Primary key : Stu\_Id
- Derived\_attribute : age



### 3. Relationship (Cardinality)

- A relationship is represented by diamond shape in ER diagram, it shows the relationship among entities.
- **Degree of a Relationship Type-** It is the number of participating entity types.
- There are four types of relationships:
  1. One to One
  2. One to Many
  3. Many to One
  4. Many to Many



#### 1. One to One Relationship

- When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship.
- For example, a person has only one passport and a passport is given to one person.

## 2. One to Many Relationship

- When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship.
- For example – a customer can place many orders but one order cannot be placed by many customers.



## 3. Many to One Relationship

- When more than one instances of an entity is associated with a single instance of another entity then it is called many to one relationship.
- For example – many students can study in a single college but a student cannot study in many colleges at the same time.



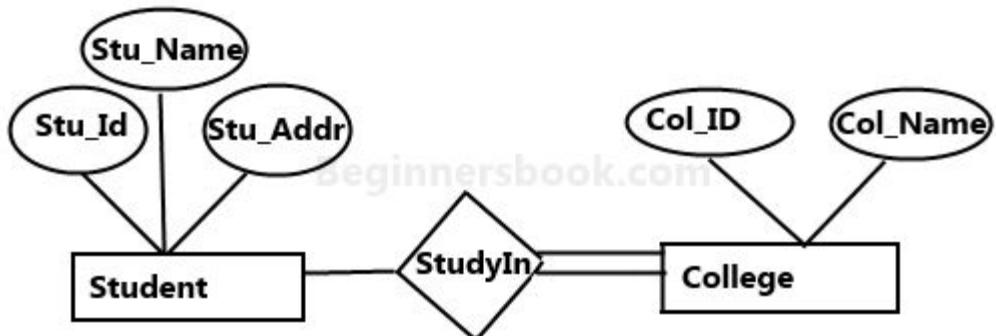
## 4. Many to Many Relationship

- When more than one instances of an entity is associated with more than one instances of another entity then it is called many to many relationship.
- For example, a student can be assigned to many projects and a project can be assigned to many students.



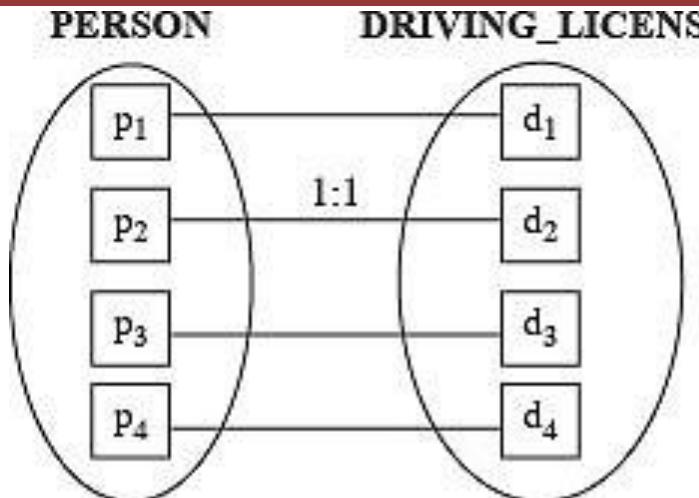
### Total participation of an entity set

- Represents that each entity in entity set must have at least one relationship in a relationship set  
 (Double line)

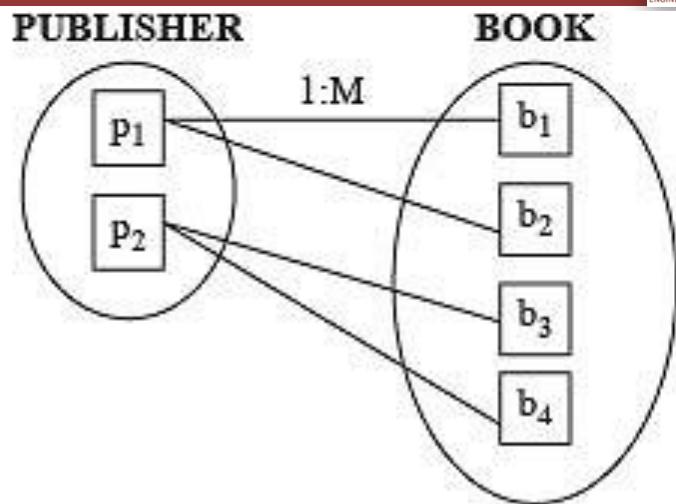


E-R Diagram with total participation of College entity set in StudyIn relationship Set - This indicates that each college must have atleast one associated Student.

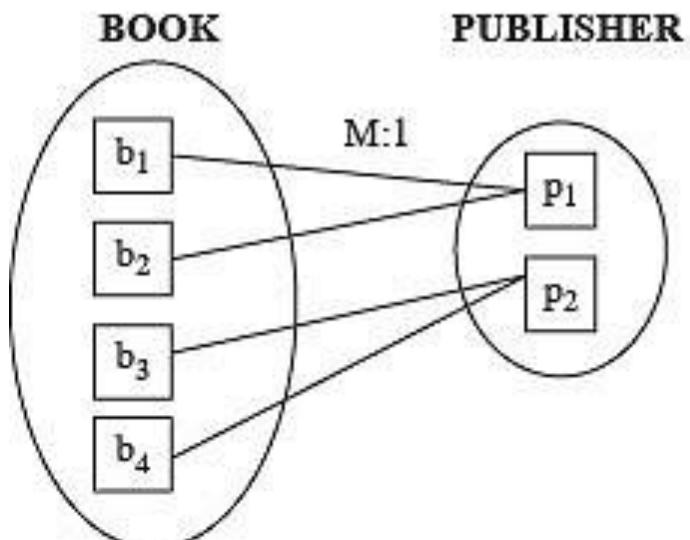
# Mapping Cardinality



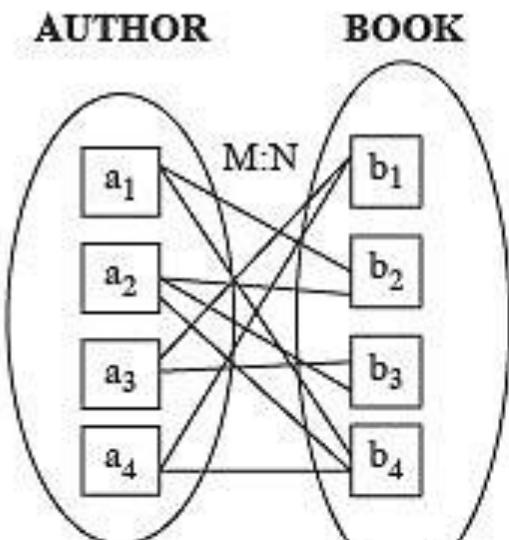
(a) One-to-one



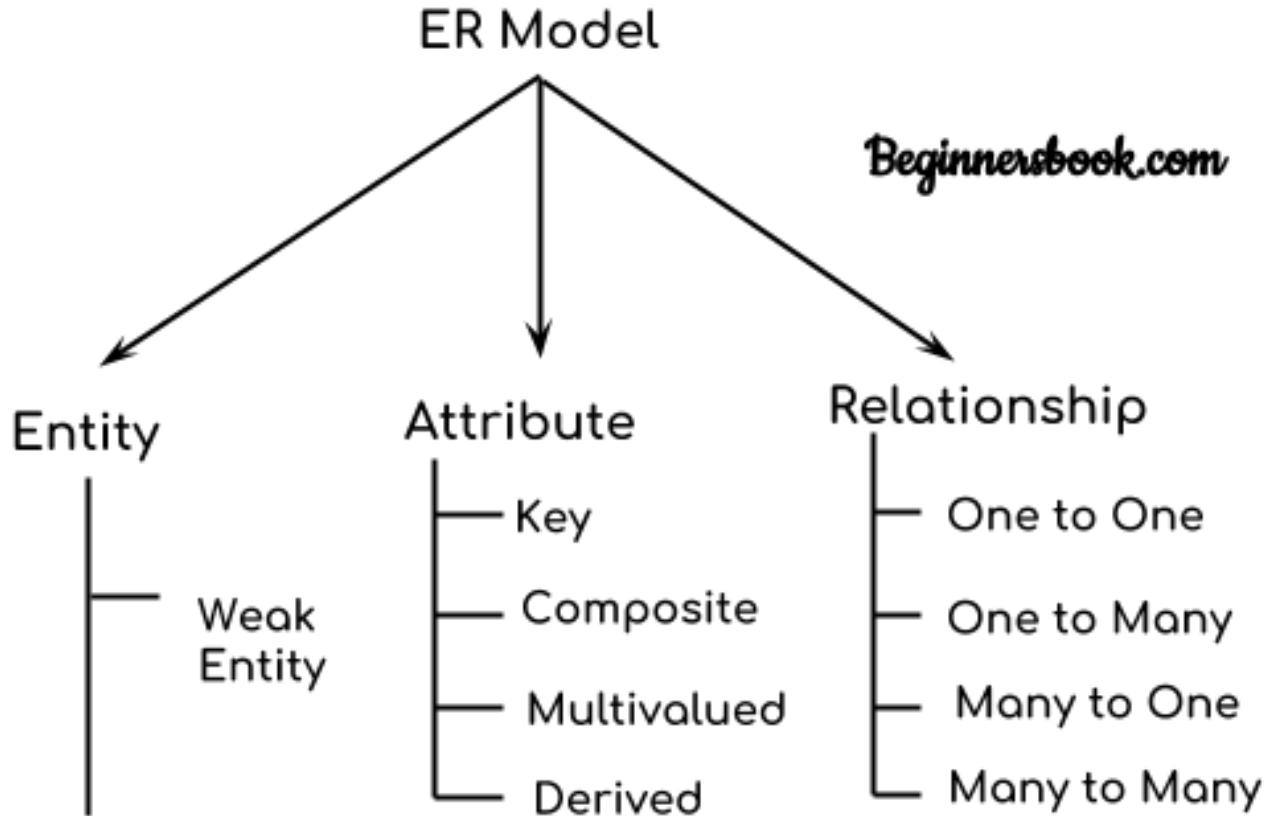
(b) One-to-many



(c) Many-to-one



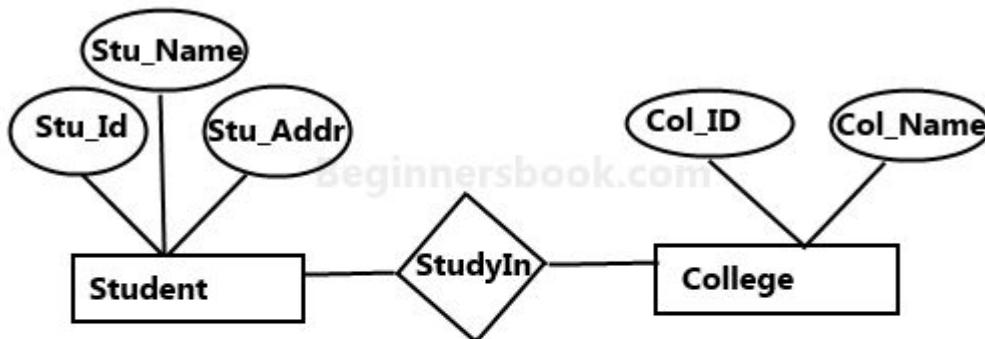
(d) Many-to-many



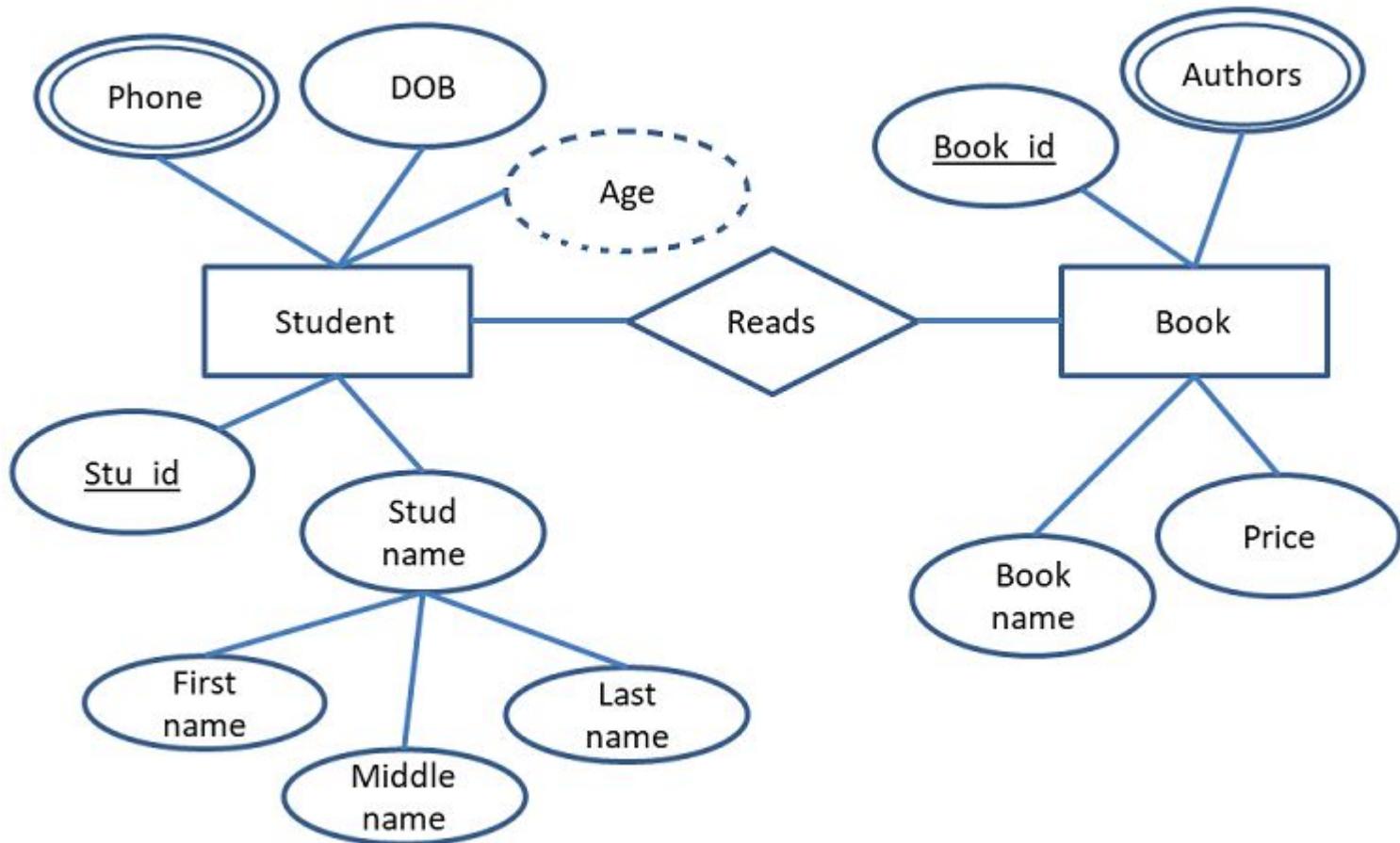
## Components of ER Diagram

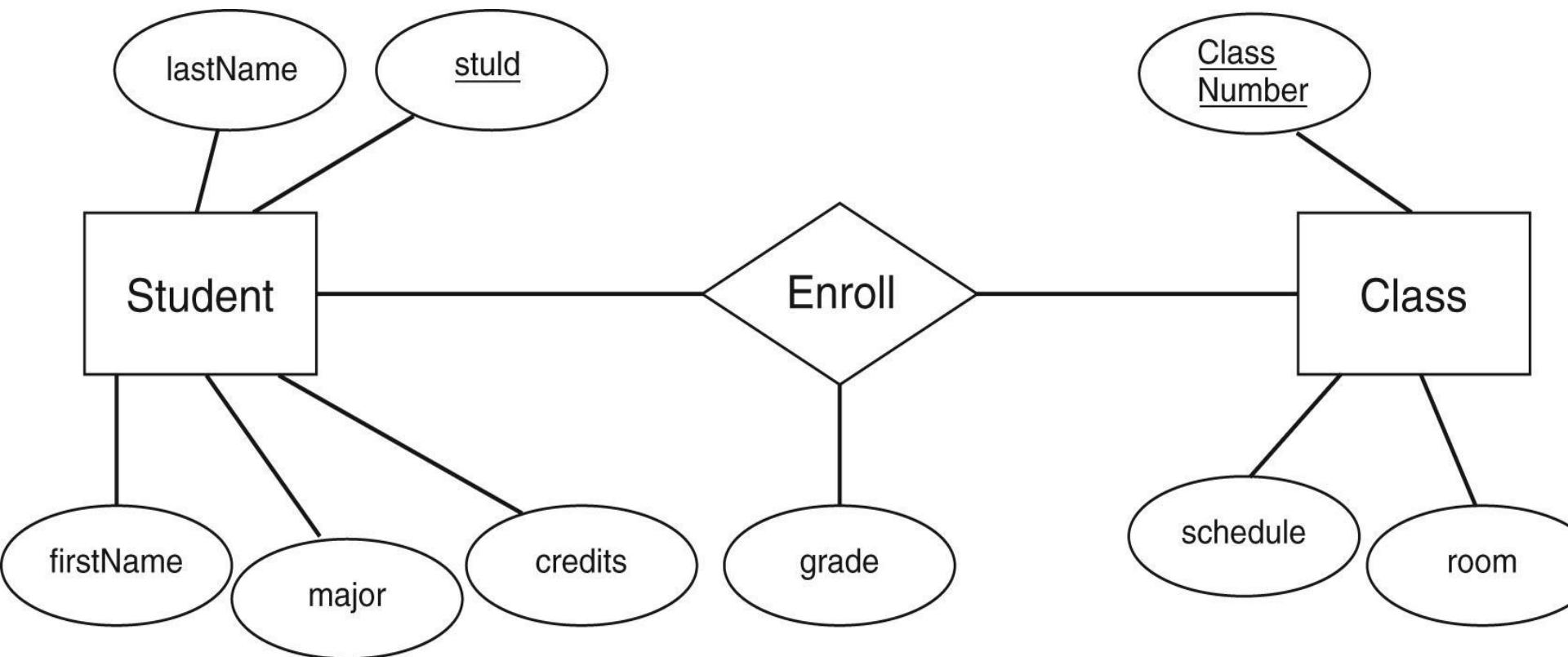
# Simple E-R Diagram

- Two entities Student and College and their relationship.
- The relationship between Student and College is many to one as a college can have many students however a student cannot study in multiple colleges at the same time.
- Student entity has attributes such as Stu\_Id, Stu\_Name & Stu\_Addr
- College entity has attributes such as Col\_ID & Col\_Name.

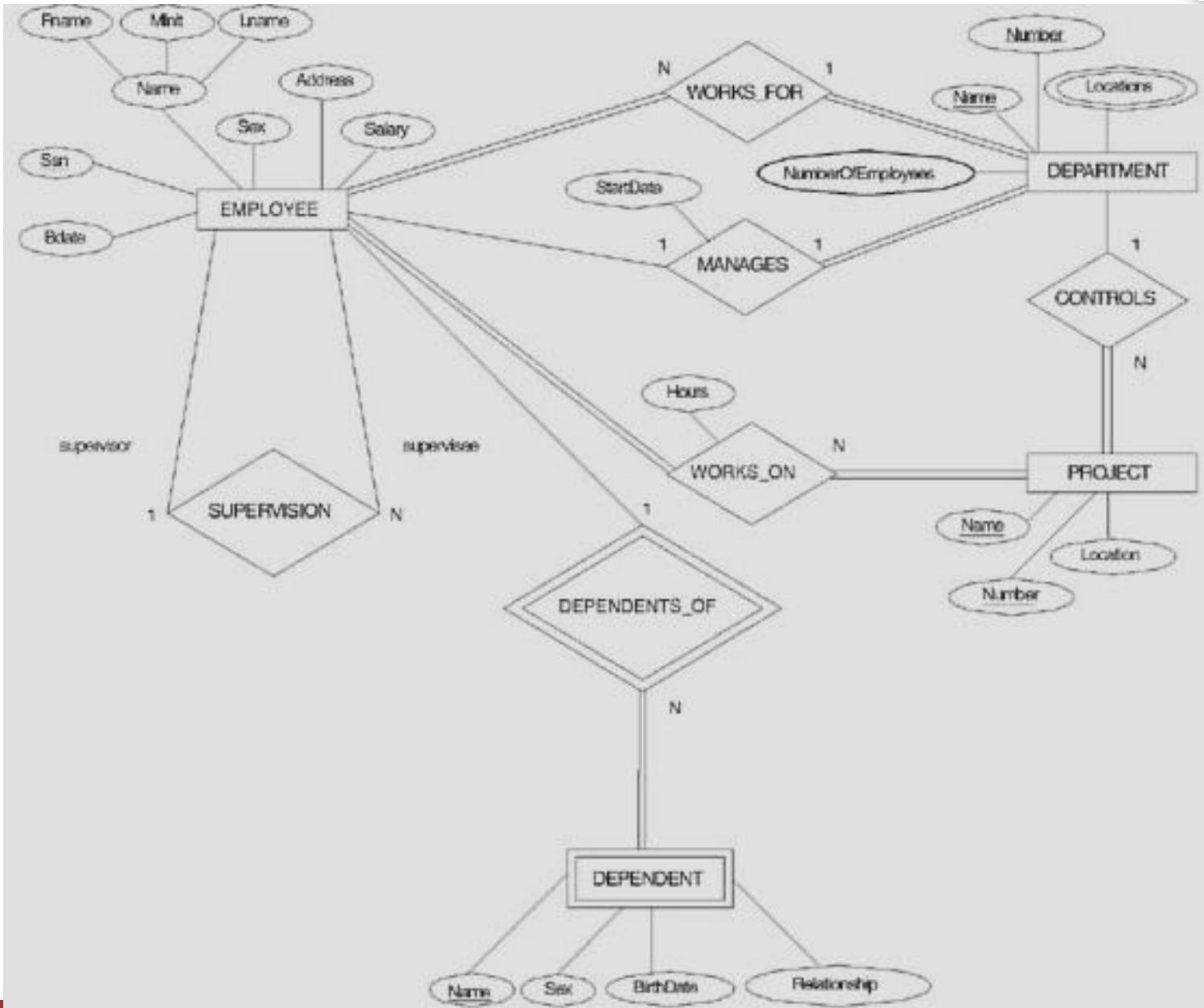


Sample E-R Diagram





# E-R Diagram: Example



# E-R Diagram: Example

- The Company is organized into DEPARTMENTS. Each department has a name, number and an employee who *manages* the department. We keep track of the start date of the department manager.
- Each department *controls* a number of PROJECTS. Each project has a name, number and is located at a single location.
- We store each **EMPLOYEE**'s social security number, address, salary, sex and birth date. Each employee **works for** one department but may **work on** several projects. We keep track of the number of hours per week that an employee currently works on each project. We also keep track of the **direct supervisor** of each employee.
- Each employee may have a number of **DEPENDENTS**. For each dependent, we keep track of their name, sex, birth date and relationship to employee.

# Cardinality Relationship

**Cardinality** – The number of entities to which another entity can be associated through a relationship

The diagrams on the right show, in order:

one-to-one

one-to-many

many-to-one

many-to-many



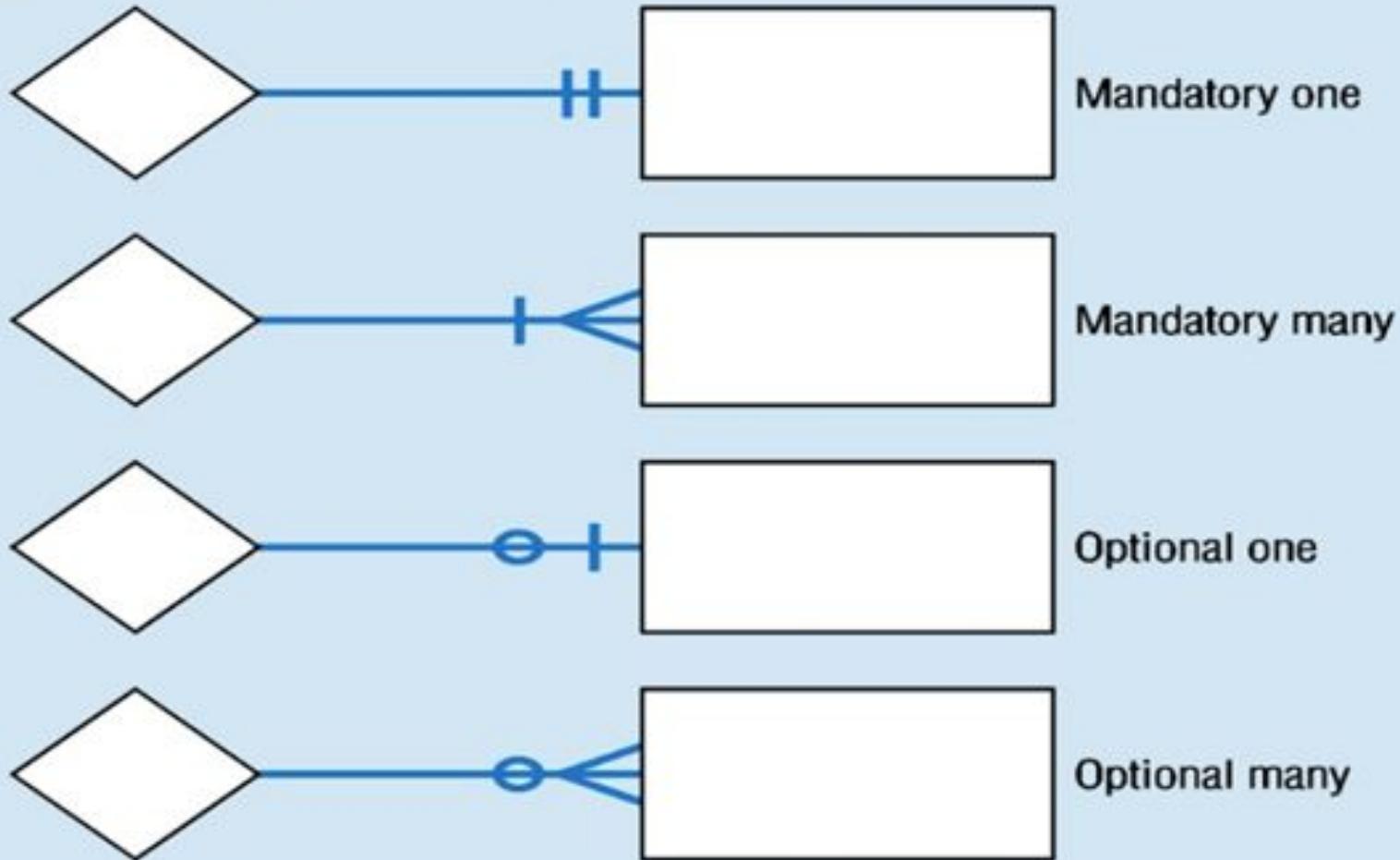
*Relationship*

*Relationship*

*Relationship*

*Relationship*

## Relationship cardinality



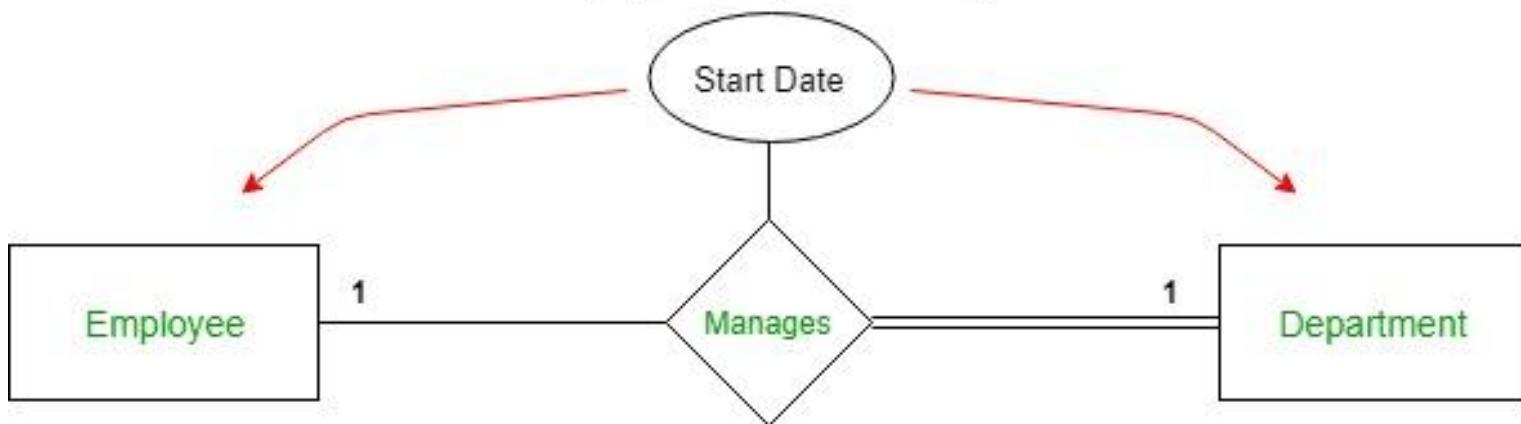
# Attributes to Relationships in ER Model

- In ER model, entities have attributes which can be of various types like single-valued, multi-valued, composite, simple, stored, derived and complex.
- Relationships can also have attributes associated to them.

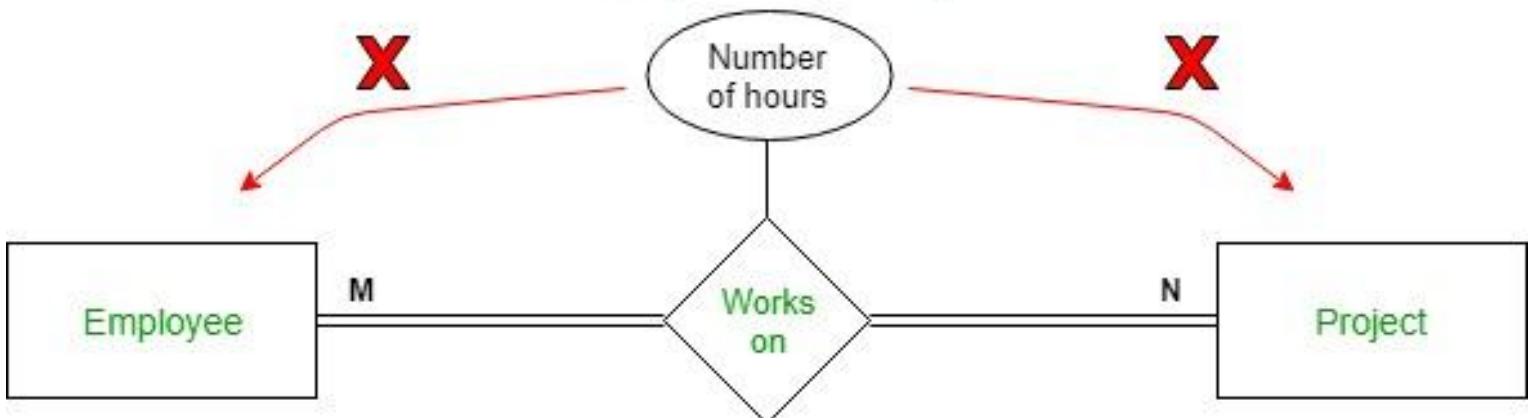
Example

- In an organisation an employee manages a department and each department is managed by some employee.
- Now, if we want to store the *Start\_Date* from which the employee started managing the department then we can give the *Start\_Date* attribute to the relationship *manages*.
- In this case we may avoid it by associating the *Start\_Date* attribute to either *Employee* or *Department* entity.

*Associate it to either  
Employee or Department entity*



*Can't associate it to  
Employee or Project entity*



# Basic design issues of ER database schema

- Users often mislead the concept of the elements and the design process of the ER diagram.
- Thus, it leads to a complex structure of the ER diagram and certain issues that does not meet the characteristics of the real-world enterprise model.
- Basic design issues of an ER database schema:
  - 1) Use of Entity Set vs Attributes
  - 2) Use of Entity Set vs. Relationship Sets
  - 3) Use of Binary vs n-ary Relationship Sets
  - 4) Placing Relationship Attributes

# Basic design issues

## 1) Use of Entity Set vs Attributes

- The use of an entity set or attribute depends on the structure of the real-world enterprise that is being modeled and the semantics associated with its attributes.
- It leads to a mistake when the user use the primary key of an entity set as an attribute of another entity set. Instead, he should use the relationship to do so.
- Also, the primary key attributes are implicit in the relationship set, but we designate it in the relationship sets.

## 2) Use of Entity Set vs. Relationship Sets

- It is difficult to examine if an object can be best expressed by an entity set or relationship set.
- To understand and determine the right use, the user need to designate a relationship set for describing an action that occurs in-between the entities.
- If there is a requirement of representing the object as a relationship set, then its better not to mix it with the entity set.

# Basic design issues

## 3) Use of Binary vs n-ary Relationship Sets

- Generally, the relationships described in the databases are binary relationships. However, non-binary relationships can be represented by several binary relationships.
- For example, we can create and represent a ternary relationship 'parent' that may relate to a child, his father, as well as his mother. Such relationship can also be represented by two binary relationships i.e, mother and father, that may relate to their child.
- Thus, it is possible to represent a non-binary relationship by a set of distinct binary relationships.

## 4) Placing Relationship Attributes

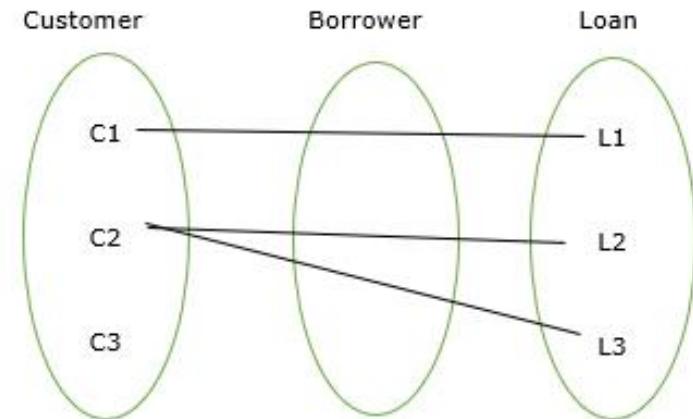
- The cardinality ratios can become an affective measure in the placement of the relationship attributes.
- So, it is better to associate the attributes of one-to-one or one-to-many relationship sets with any participating entity sets, instead of any relationship set.
- Decision of placing specified attribute as a relationship or entity attribute should possess the characteristics of real world enterprise that is being modeled.

# Example

- If there is an entity which can be determined by the combination of participating entity sets, instead of determining it as a separate entity. Such type of attribute must be associated with the many-to-many relationship sets.
- Thus, it requires the overall knowledge of each part that is involved in designing and modeling an ER diagram. The basic requirement is to analyse the real-world enterprise and the connectivity of one entity or attribute with other.

# What are constraints on ER model in DBMS?

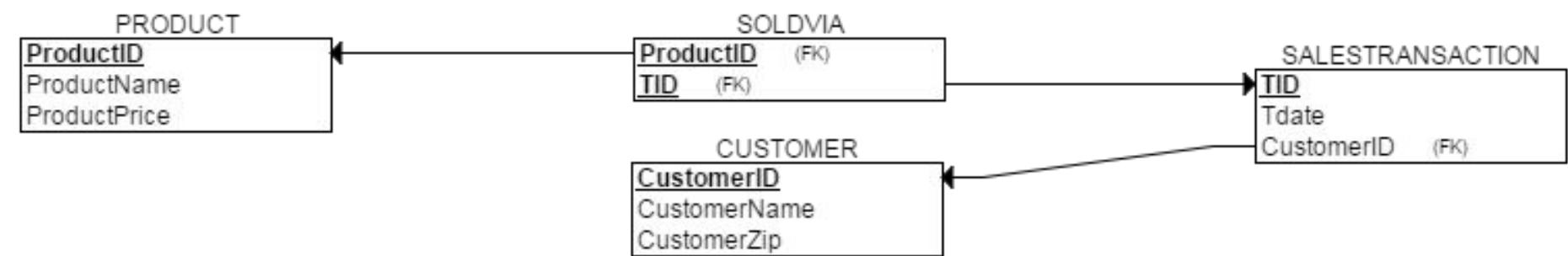
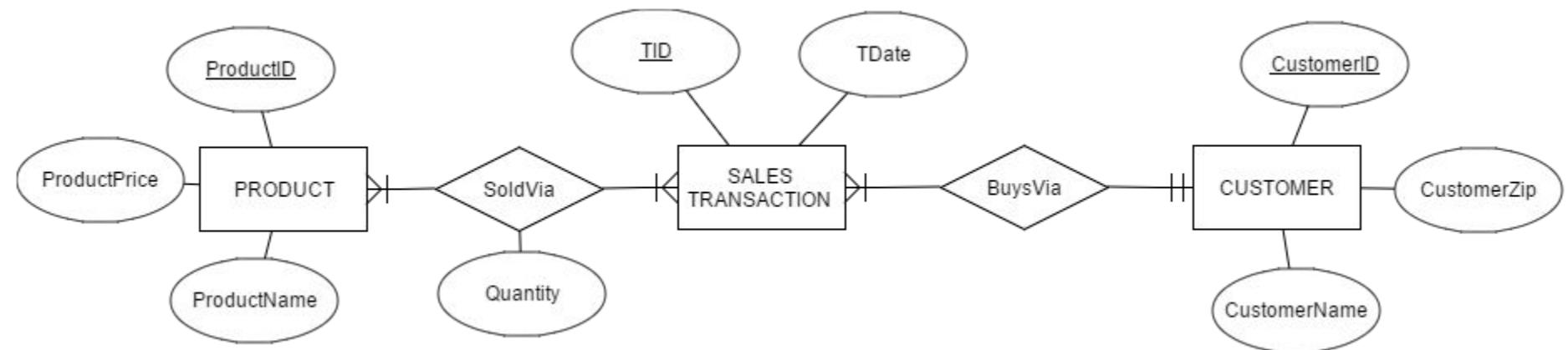
- Constraints are used for modeling limitations on the relations between entities.
- There are two types of constraints on the Entity Relationship (ER) model
- Mapping cardinality or cardinality ratio.
  - A mapping constraint is a data constraint that expresses the number of entities to which another entity can be related via a relationship set.
- Participation constraints: two types –
- Total participation:
  - The participation of an entity set E in a relationship set R is said to be total if every entity in E Participates in at least one relationship in R.
- Partial Participation: If only some of the entities in E participate in relationship R, then participation of E in R is partial participation
- Example
- Participation of loan in the relationship borrower is total participation
- Participation of customers in the relationship borrower is partial participation



# ERDPlus - Database modeling tool

- ERDPlus is a web-based database modeling tool that lets you quickly and easily create
  - Entity Relationship Diagrams (ERDs)
  - Relational Schemas (Relational Diagrams)
  - Star Schemas (Dimensional Models)
- More features
  - Automatically convert ER Diagrams into Relational Schemas
  - Export SQL
  - Export diagrams as a PNG
  - Save diagrams safely on our server

# Automatically Convert ER Diagrams to Relational Schemas



# References

*Thank  
you*





# 100008/IT400D

## DATABASE MANAGEMENT SYSTEMS

**Module I**  
INTRODUCTION TO E-R MODEL

**Prepared by**  
Dr. Sherly K.K  
Associate Professor  
Information Technology

# Objectives

- To impart the basic understanding of the theory and applications of database management systems
- To give basic level understanding of internals of database systems
- To expose to some of the recent trends in databases
- **Text Books**
  - Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
  - Liberschatz A., H. F. Korth and S. Sudarshan, Database System Concepts, 6/e, McGraw Hill, 2011.
- **Reference Books**
  - Powers S., Practical RDF, O'Reilly Media, 2003.
  - Plunkett T., B. Macdonald, et al., Oracle Big Data Hand Book, Oracle Press, 2013
  - Adam Fowler, NoSQL for Dummies, John Wiley & Sons, 2015.
  - NoSQL Data Models: Trends and Challenges (Computer Engineering: Databases and Big Data), Wiley, 2018

# Course Outcomes:

<b>CO 1</b>	Define, explain and illustrate the fundamental concepts of databases
<b>CO 2</b>	Model real world scenarios given as informal descriptions, using Entity Relationship diagrams.
<b>CO 3</b>	Model and design solutions for efficiently representing and querying data using relational model
<b>CO 4</b>	Demonstrate the features of indexing and hashing in database applications
<b>CO 5</b>	Discuss and compare the aspects of Concurrency Control and Recovery in Database systems
<b>CO 6</b>	Appreciate the latest trends in databases

## Module 1: INTRODUCTION TO E-R MODEL

- **Introduction:** Data: structured, semi-structured and unstructured data, Concept & Overview of DBMS, Data Models, Database Languages, Database Administrator, Database Users, Three Schema architecture of DBMS. Database architectures and classification. (Reading: Elmasri Navathe, Ch. 1 and 2. Additional Reading: Silberschatz, Korth, Ch. 1)
- **Entity-Relationship Model:** Basic concepts, Design Issues, Mapping Constraints, Keys, Entity Relationship Diagram, Weak Entity Sets, Relationships of degree greater than 2 (Reading: Elmasri Navathe, Ch. 7.1-7.8)

# Syllabus Module 2

- **Module 2:** RELATIONAL MODEL, DATABASE LANGUAGES & SQL
- **Relational Model:** Structure of relational Databases, Integrity Constraints, synthesizing ER diagram to relational schema  
(Reading: Elmasri Navathe, Ch. 3 and 8.1, Additional Reading: Silberschatz, Korth, Ch. 2.1-2.4)
- **Database Languages:** Concept of DDL and DML relational algebra (Reading: Silbershatz, Korth, Ch 2.5-2.6 and 6.1-6.2, Elmasri Navathe, Ch. 6.1-6.5).
- **Structured Query Language (SQL):** Basic SQL Structure, examples, Set operations, Aggregate Functions, nested sub-queries (Reading: Elmasri Navathe, Ch. 4 and 5.1) Views, assertions and triggers (Reading: Elmasri Navathe, Ch. 5.2-5.3, Optional reading: Silbershatz, Korth Ch. 5.3).

# Syllabus Module 3

## Module 3: RELATIONAL DATABASE DESIGN

- **Relational Database Design:** Different anomalies in designing a database, normalization, functional dependency (FD), Armstrong's Axioms, closures, Equivalence of FDs, minimal Cover (proofs not required). Normalization using functional dependencies, INF, 2NF, 3NF and BCNF, lossless and dependency preserving decompositions
- (Reading: Elmasri and Navathe, Ch. 14.1-14.5, 15.1-15.2.
- Additional Reading: Silberschatz, Korth Ch. 8.1-8.5)

## Module 4: PHYSICAL DATA ORGANIZATION AND QUERY OPTIMIZATION

- **Physical Data Organization:** index structures, primary, secondary and clustering indices, Single level and Multi-level indexing, B+-Trees (basic structure only, algorithms not needed), (Reading Elmasri and Navathe, Ch. 17.1-17.4)
- **Query Optimization:** heuristics-based query optimization, (Reading Elmasri and Navathe, Ch. 18.1, 18.7)

# Syllabus Module 5

## Module 5: TRANSACTION PROCESSING CONCEPTS

- **Transaction Processing Concepts:** overview of concurrency control and recovery acid properties, serial and concurrent schedules, conflict serializability. Two-phase locking, failure classification, storage structure, stable storage, log based recovery, deferred database modification, check-pointing, (Reading Elmasri and Navathe, Ch. 20.1-20.5 (except 20.5.4-20.5.5) , Silbershatz, Korth Ch. 15.1 (except 15.1.4-15.1.5), Ch. 16.1 – 16.5)
- **Recent topics** (preliminary ideas only): Semantic Web and RDF(Reading: Powers Ch.1, 2),
- GIS, biological databases (Reading: Elmasri and Navathe Ch. 23.3- 23.4)
- Big Data (Reading: Plunkett and Macdonald, Ch. 1, 2)

# Evaluation Pattern

- **Mark distribution**

Total Marks	CIE	ESE	ESE Duration
150	50	100	3 hours

- **Continuous Internal Evaluation Pattern:**

- Attendance : 10 marks
- Continuous Assessment Test (2 numbers) : 25 marks
- Assignment/Quiz/Course project : 15 marks

- **End Semester Examination Pattern:**

- There will be two parts; Part A and Part B.
- Part A contain 10 questions with 2 questions from each module, having 3 marks for each question. Students should answer all questions.
- Part B contains 2 questions from each module of which student should answer any one. Each question can have maximum 2 sub-divisions and carry 14 marks.

# Eligibility to Attend Examination

- A student is required to possess minimum 75% attendance if he/she is to be admitted for the end semester examination.
- A student is required to possess minimum 80% attendance (B.Tech. programmes) if he/she is to be admitted for the Internal Examinations.
- Students should inform the class teacher before taking leave.

# **Procedure for leave application:**

- The duly filled application signed by the student and the parent/guardian/ warden is to be presented to the class teacher on the date of rejoining after taking the leave.** For this, the student should collect the leave application form from the class teacher/HoD before taking the leave.
- If the attendance percentage falls below 90% at any stage, the student will have to meet the Head of the Department (HoD) after meeting the class teacher.**
- If the attendance percentage falls below 80% at any stage, the student will have to meet the Principal, after meeting the class teacher and the HoD.**
- In all cases where a leave of absence is required due to illness, the application for the same must be supported by a medical certificate from the doctor who treated the student and this should be submitted within 3 working days from the date of rejoining.** In this case, the class teacher should forward the leave application to the HoD, who can grant/refuse the leave. Medical certificate obtained after 3 working days shall not be accepted.
- The name of the student will be removed from the rolls, for unauthorised absence of more than 10 consecutive working days.**

- Attendance shortage will not be condoned unless the absence is regularized by leave application as mentioned above

# **Introduction To E-R Model**

INTRODUCTION  
**Entity-Relationship Model**

# Basic Definition

- **Data:** Data is a set of values of qualitative or quantitative variables. It is information in raw or unorganized form.
- It may be a fact, figure, characters, symbols etc.
- **Data:** known facts that can be recorded and that have implicit meaning.
- eg: the name of a person .
- **Database:** collection of related data

# Database Management System (DBMS)

- Collection of interrelated data
- Set of programs to access the data
- DBMS contains information about a particular enterprise
- DBMS provides an environment that is both *convenient* and *efficient* to use.
- **Database Applications:**
  - Banking: all transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
- Databases touch all aspects of our lives

# Purpose of Database System

- In the early days, database applications were built on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation – multiple files and formats
  - Integrity problems
    - Integrity constraints (e.g. account balance > 0) become part of program code
    - Hard to add new constraints or change existing ones

# Purpose of Database Systems (Cont.)

- Drawbacks of using file systems (cont.)
  - Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - E.g. transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Concurrent accessed needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies
      - E.g. two people reading a balance and updating it at the same time
  - Security problems
- Database systems offer solutions to all the above problems

# The Structure of Big Data

## ❖ Structured

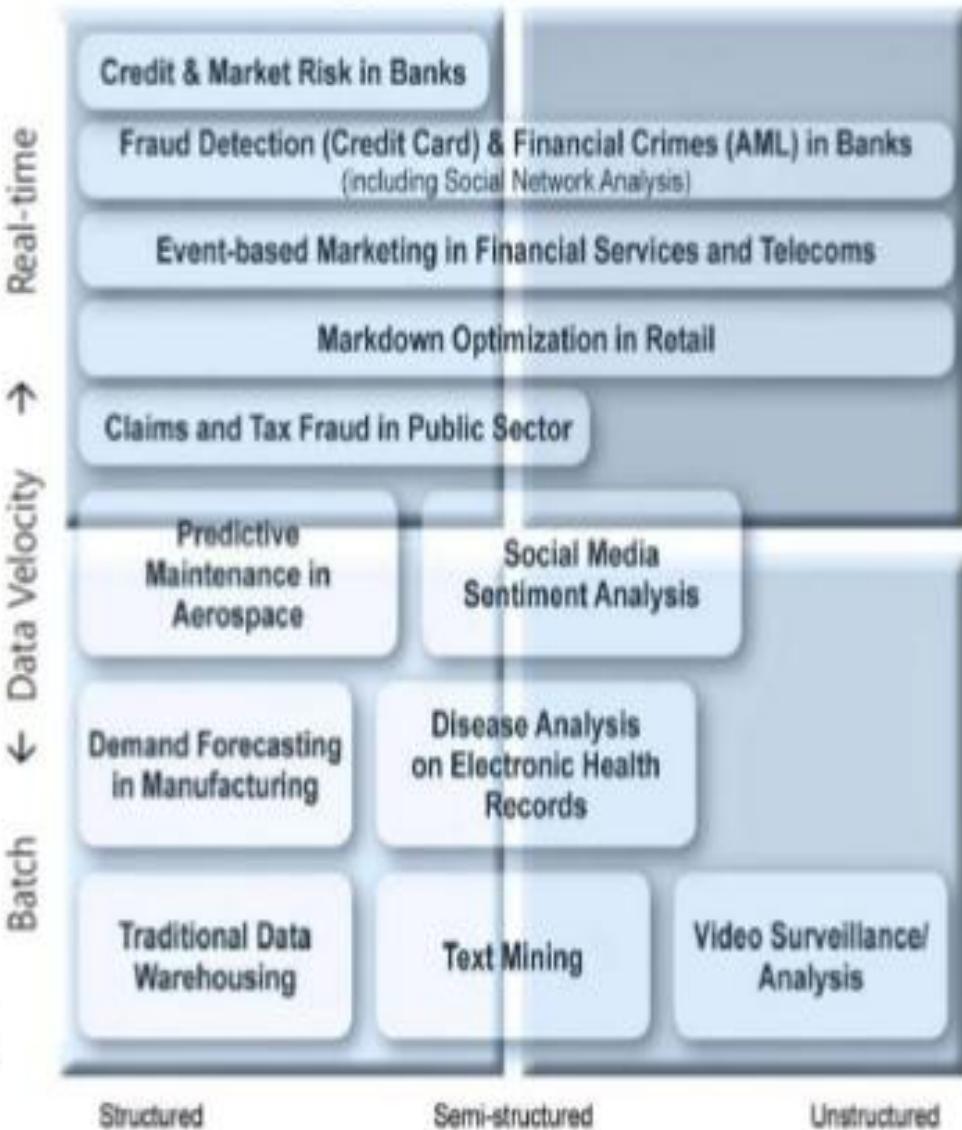
- Most traditional data sources

## ❖ Semi-structured

- Many sources of big data

## ❖ Unstructured

- Video data, audio data



# Implicit properties of a database

- 1) A database represents **some aspect of the real world**.
- 2) A database is designed and built with data for a **specific purpose**. It is meant for a group of users and they use it in their preconceived applications.
- 3) A database can be of **any size and varying complexity**.

e.g.: 1. list of names and address of people we know.  
2. Computerised catalog of a large library.  
3. A database maintained by the internal revenue service to keep track of the tax forms filed by U.S. tax payers.

# Data base management systems.

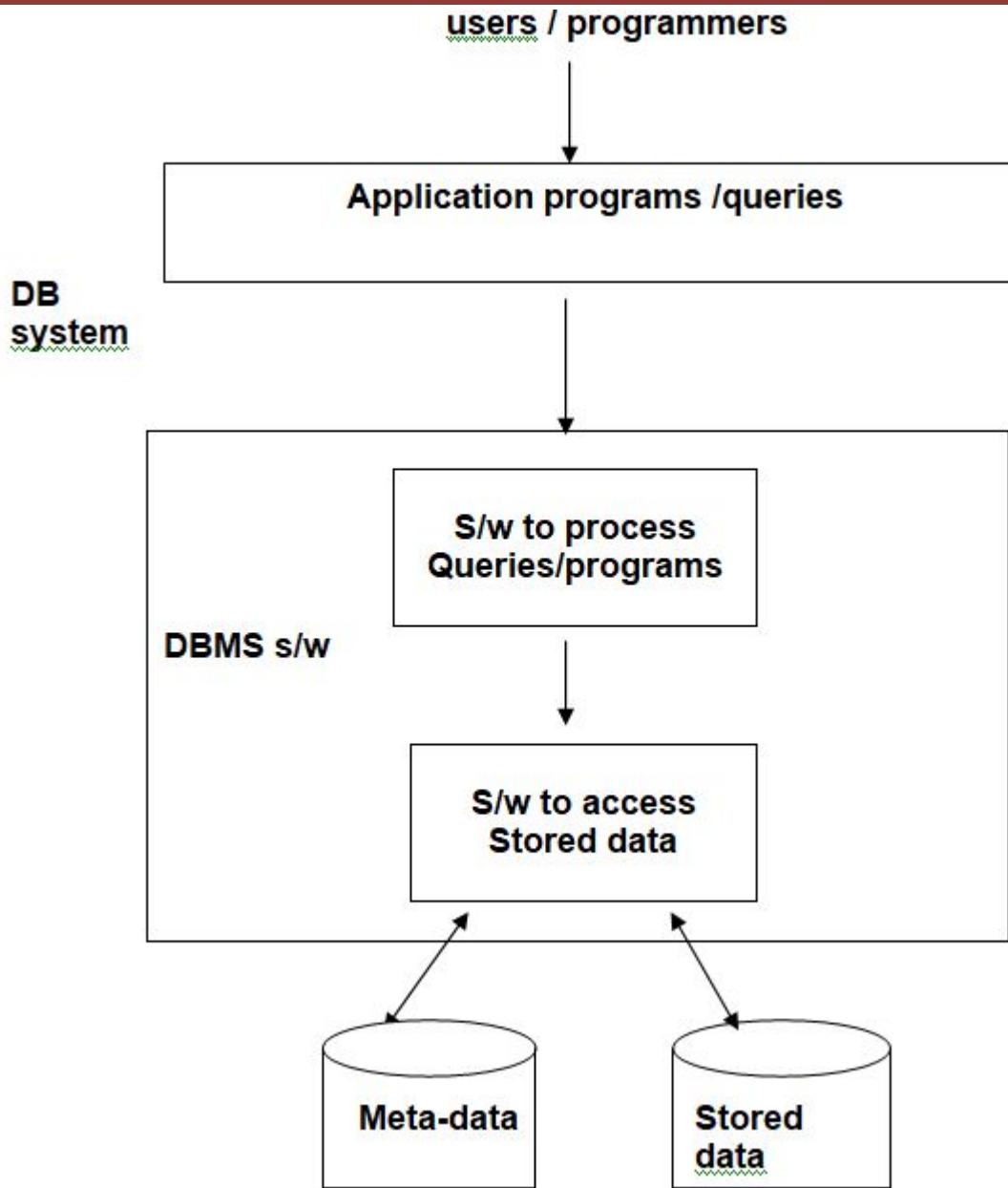
- Collection of programs that enables users to create and maintain a DB .
- The database is hence a general-purpose software system that facilitates the processes of
  1. **Defining:** specifying the data types, structure and constraints for data.
  2. **Constructing:** storing the data on some storage medium.
  3. **Manipulating:** functions like querying the database to reflect changes in the miniworld and generating reports from the data.
  4. **Sharing:** allows multiple users and programs to access the database concurrently.

The primary goal of DBMS is to provide an environment that is convenient and efficient for people to use in retrieving and storing information.

# Data base systems

- A database and data base management systems together is known as the data base system.
- A computerized system to store information and to allow all users to retrieve and update that information on demand.
- A data base system involves four major components.
  - a) Data
  - b) Hardware
  - c) Software
  - d) Users

# A simplified database system environment



- **Data:** data in the data base ( at least in the large systems ) will be both **integrated and shared**.
- **Integrated:** database can be thought of as a **unification of several files**, with any redundancy among those files at least partly eliminated.
- eg: a DB contain EMPLOYEE file and ENROLLMENT file(representing the enrollment of employees in training courses)
- Employee
 

name	address	department	salary
------	---------	------------	--------
- Enrollment
 

name	course
------	--------

# Hardware & Software components

**Hardware** components consists of

- i) The **secondary storage** volumes (mostly magnetic disks) with the associated I/O devices.(disk drives etc...)
- ii) The h/w **processor and associated main memory** that are used to support the execution of the DB system s/w.

## Software

- Between the physical DB and the users of the system is a layer of s/w known as **DB manager**, DB server or DBMS.
- All requests to access to the data base are handled by the DBMS.

# Users

Three broad classes of users

- 1) **Application programmers** (responsible for writing DB application programs)
- 2) **End users** (who interact with the system from online workstations or terminals)
- 3) **Data base administrator (DBA)**

Administrating the resources of DB system.  
(primary resource is the DB and the secondary resource is the DBMS and related s/w)

# Database Users

Users are differentiated by the way they expect to interact with the system

- i. Application programmers – interact with system through DML calls
- ii. Sophisticated users – form requests in a database query language
- iii. Specialized users – write specialized database applications that do not fit into the traditional data processing framework
- iv. Naïve users – invoke one of the permanent application programs that have been written previously
  - E.g. people accessing database over the web, bank tellers, clerical staff

# CHARACTERISTICS OF DATA BASE APPROACH

- In traditional file processing each user maintains separate files. In a DB approach the data base is created by the designer, maintained by the DBA and used by various users for different applications.

**1. Self describing nature of a DB ((meta-data))**

**2. Insulation between programs and data abstractions**  
(data independence)

**3. Support of data and multiple views of the data**

Views: A subset of the database, virtual data that is derived from the database.

**4. Sharing of data and multi-user transaction processing**

Enables concurrency control ensuring that several users trying to update the same data, do so in a controlled manner .

# ADVANTAGES OF A DBMS

- 1. The data can be shared**
- 2. Redundancy can be reduced**
- 3. Integrity can be maintained**
  - ensures that data in the data base is correct
- 4. Security can be enforced**
  - Different constraints can be established for each type of access
- 5. Restricting unauthorized access (central control)**
- 6. Providing persistent storage for program objects and data structures.**
- 7. Providing back up and recovery**
- 8. Provides multiple user interfaces**
- 9. Capable of establishing complex relationship among data**
- 10. Permits inference and actions using rules**
- 11. Providing storage structures for efficient query processing**
- 12. Transaction support can be provided**

# Data Models

- A set of concepts to describe the ***structure*** of a database, the ***operations*** for manipulating these structures, and certain ***constraints*** that the database should obey
- A collection of tools for describing structure of database.
  - data types
  - data relationships
  - data semantics
  - data constraints
  - basic operations
- Entity-Relationship model
- Relational model
- Other models:
  - object-oriented model
  - semi-structured data models
  - Older models: network model and hierarchical model

- **High Level/ Conceptual Data Models**- provides concepts that are close to how Users perceive data. Uses concepts such as Entities, attributes, and relationships.
- **Low Level/ Physical Data Model** – provides concepts that describe the details of how data is stored in the computer.- meant mostly for Computer specialists and not for general users.
- **Representational / Implementation Data Models**- Lies in between the above two extremes. This model used in traditional commercial DBMS's
  - Eg: Relational Data model, Network data model and hierarchical data model. They hide some details of data storage but can be implemented on a computer system in a direct way.

# DATABASE SCHEMAS

- Refers to overall design of the database.
- The description of a database.
- It is used during the data base design and not expected to change frequently.
- Database systems have several schemas according to the levels of abstraction.
  - The **physical schema** describes the database design at the physical level,
  - **Logical schema** describes the database design at the logical level.
- A displayed schema is called **schema diagram**.

# Schema diagram

- Shows the structure of each record type.
- Displays only some aspects of a schema.
- Other aspects are not there in the schema (like data type of each item ,relationship among the various files etc.

Example:

- Student

NAME	STUDENT-NO.	CLASS	MAJOR
------	-------------	-------	-------

- Course

COURSE NAME	COURSE NO:	CREDIT HOURS	DEPARTMENT
-------------	------------	--------------	------------

- Grade

STUDENT-NO.	COURSE NO:	GRADE
-------------	------------	-------

# DATABASE INSTANCE

- Data in the database at a particular moment of time (also called **database state, snapshot or current set of occurrences**)
- When we define a new database, the corresponding database state is the **empty state** with no data.
- **Initial state:** when it is first populated or loaded with the initial data.
- Every time an update operation is applied to the database, we get another data base state.
- Schema is not supposed to change frequently. But changes need to be occasionally applied to the schema as the application requirements change. This is known as **schema evolution**.
- Eg: we may decide that another data item needs to be stored for each record in a file, such as adding the DOB to the student schema.

# Database Schema vs. Database State

- Database State is the collection of all data in database. It is also applied to individual database components,
- Example: *record instance, table instance, entity instance.*
- Distinction
  - The ***database schema*** changes very infrequently.
  - The ***database state*** changes every time the database is updated.
- **Schema** is also called **intension**.
- **State** is also called **extension**.

# Example of a database state

## COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

## SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

## GRADE\_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

## PREREQUISITE

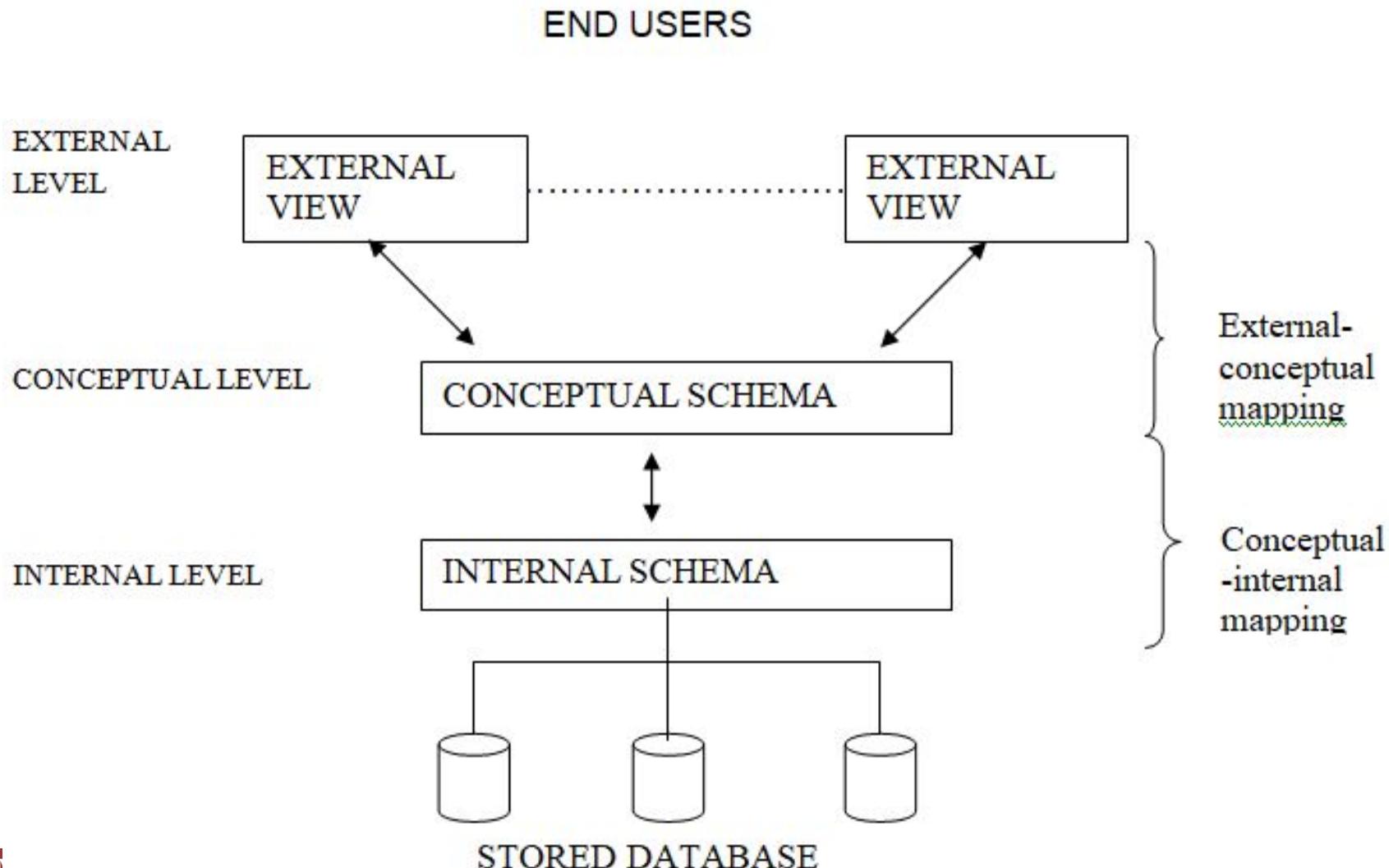
Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

**Figure 1.2**

A database that stores student and course information.

# THREE SCHEMA ARCHITECTURE

Goal of three schema architecture is to separate the user applications and the physical database.



# Three Schemas

- The architecture defines DBMS schemas at ***three*** levels:
- **INTERNAL LEVEL**:- has an internal schema, which describes the physical storage structure of the database. The internal schema uses physical data models and describes the complete details of data storage and access paths for the database.
- **CONCEPTUAL LEVEL**:-has the conceptual schema, which describes the structure of the database to users but hides physical storage structures.
- The conceptual schema gives entities, data types, relationships, user operations and constraints.(typically useful for DBA)
- **EXTERNAL SCHEMA**:-includes a number of external schemas or user views for a particular user group and hides the rest of the database from that user group.

The process of transforming requests and results between levels are called mapping.

# Three-Schema Architecture

- Proposed to support DBMS characteristics of:
  - **Program-data independence.**
  - Support of **multiple views** of the data.
- Mappings among schema levels are needed to transform requests and data.
  - Programs refer to an external schema, and are mapped by the DBMS to the internal schema for execution.
  - Data extracted from the internal DBMS level is reformatted to match the user's external view (e.g. formatting the results of an SQL query for display in a Web page)

# Data independence

- The capacity to change the schema at one level of a database without having to change the schema at the next higher level.
- Two types of data independence
  - 1) **Logical data independence**
  - 2) **Physical data independence**
- **Logical data independence**:- Capacity to change conceptual schema without having to change the external schemas and their application programs.

For eg: External schema of ‘STUDENT FILE’ should not be affected by changing ‘COURSE FILE’ by adding a data item in to the file.

- **Physical data independence**: Capacity to change the internal schema without having to change the conceptual schema. Changes in the internal schema may be needed because some physical files had to be reorganized.

Eg: creating additional access structures to improve the performance of retrieval or update. If the same data remains in the database, we should not have to change the conceptual schema.

# Database Languages

- SQL is the most widely used query language
- SQL stands for **Structured Query Language** used for storing and managing data in Relational DBMS like Oracle, MySQL, MS SQL Server, IBM DB2. . .
- It is a standard programming language for accessing a relational database.
- A database system provides
- Data definition language (DDL) to specify the database schema
- Data manipulation language (DML) to express database queries and updates.

# Data Definition Language (DDL)

- **DDL is used by DBA and DB designers to define both conceptual and internal schemas**
- Specification notation for defining the database schema
  - E.g.  
`create table account (`  
    `account-number char(10),`  
    `balance integer)`
- DDL compiler generates a set of tables stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
  - database schema
  - Data *storage and definition* language (SDL)
    - language in which the storage structure and access methods used by the database system are specified
    - Usually an extension of the data definition language

# Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
  - DML also known as query language
- DML (Data Manipulation Language)-used to specify
  - 1) Retrieval of information stored in the database
  - 2) Insertion of new information
  - 3) Deletion of information
  - 4) Modification of information in the database
- Two classes of DML
  - Low-level or Procedural – user specifies what data is required and how to get those data
  - High-level or Nonprocedural – user specifies what data is required without specifying how to get those data

# DDL Commands

## The SQL statements

- CREATE, ALTER, and DROP define the structure of a database, including rows, columns, tables, indexes and database specifies such as file locations.
- CREATE TABLE
- The *create* command is used to make a new database, table, index, or stored query.
- When a table is created, its columns are named; data types and sizes are supplied for each column.
- Each table must have at least one column.

# Create table command

1. The syntax of create table command is:
  - **CREATE TABLE** <table name> (<column name> <datatype>(<size>), <column name> <datatype>(<size>),.....);
2. The syntax for create table command that includes constraints is as follows:
  - **CREATE TABLE** <table name> (<column name> <datatype>(<size>) <column constraint>, <column name> <datatype>(<size>) <column constraint>, .....);

# Constraints

The various constraints that can be introduced while creating a table are :

- **Not null, Unique, Primary key, Check and Referential integrity constraint.**
- Examples of implementing some of these constraints are:
- <Column name> <data type> (size) NOT NULL. This means that the column can never have empty values.
- <Column name> <data type> (<size>) PRIMARY KEY. This declares the column as the primary key of the table.
- <Column name> <data type> (size) UNIQUE. This ensures that no two rows have same value in the specified column.

# ALTER TABLE

- Alter table command is used to modify an existing database object.
- It is used to change the definition of existing tables.
- Usually it can add columns to a table. Sometimes it can delete column or change their sizes.

The syntax to add a column is:

- **ALTER TABLE <table name> MODIFY ( <column name> <new data type> (new size));**
- We can also add constraints to an existing table, using the ALTER TABLE command.

Example:

- **ALTER TABLE <table name> ADD CONSTRAINT <constraint name> PRIMARY KEY (<column name>);**
- Alter table command can add any of the five constraints to an existing table.
- Also when used along with add/drop commands, it can add or delete specific columns from a table.

# DROP TABLE

- Drop table command is used to destroy an existing database, table, index or view.
- A drop statement in SQL removes an object from a relational database management system (RDBMS).
- The types of objects that can be dropped depend on which RDBMS is being used, but most support the dropping of tables, users and databases.
- Some systems allow drop and other DDL commands to occur inside of a transaction and thus be rolled back.
- The syntax for drop statement is  
`DROP object_type object_name;`
- In order to drop a table, the syntax of the drop command is as follows:  
`DROP TABLE <table name>;`

# DML Commands

- DML is a language that enables users to access or manipulate data as organized by appropriate data model.
- Data Manipulation language is basically of two types
  1. Procedural DMLs - require a user to specify what data is needed and how to get it.
  2. Declarative DMLs - require a user to specify what data is needed without specifying how to get it.

## **SELECT**

- SELECT command in SQL returns a result of records from one or more tables.
- It is used to retrieve zero or more rows from one or more tables in a database.

# SELECT Command

Syntax is as follows:

- SELECT <column name> [<column name>.....]  
FROM <table name>  
WHERE <condition> ;
- Commonly available keywords related to select include:
  1. **WHERE** – Used to identify which rows t retrieve or applied to GROUP BY.
  2. **GROUP BY** – Used to combine rows with related values into elements of a smaller set of rows.
  3. **HAVING** – Used to identify which rows following a GROUP BY, are to be retrieved.
  4. **ORDER BY** – Used to identify which columns are used to sort the resulting data

# INSERT

- INSERT statement adds a record to a table in a relational database.
- Insert statement have the following syntax
- `INSERT INTO <tablename> (<column list>) VALUES (<value>,<value>);`
- The number of columns and values must be the same.
- If a column is not specified, the default value for the column is used.
- The values specified by the insert statement must satisfy all the applications constraints (such as primary key, check constraints and not null constraints).
- If a syntax error occurs or if any constraints are violated, the new row is not added to the table.
- When values for all columns in the table are specified, then shorthand may be used, taking advantage of the columns when the table was created.
- `INSERT INTO <table name> VALUES (<value>,<value>)`

# UPDATE

- A relational database management system uses SQL update statement to change data in one or more records.
- Either all the rows can be updated, or a subset may be chosen using a condition.
- The update command specifies the rows to be changed using the where clause, and the new data using the SET keyword .
- The syntax is:
- `UPDATE <table name> SET <column name> = <values>[<column name> = <value> ....]  
[WHERE <condition>] ;`

# DELETE

- The delete command removes rows from a table.
- This removes the entire rows, not individual fields values, so no field argument is needed or accepted.
- Syntax is:
- `DELETE FROM <table name> [ WHERE <predicate>];`
- Without the where clause, all the contents of the table can be deleted.
- Using the where clause, a condition can be used to determine which rows to be deleted.

# New Table creation example

The student table is created as follows –

- create table student(id number(10), name varchar2(20), classID number(10), marks varchar2(20));
- Insert into student values(1,'pinky',3,2.4);
- Insert into student values(2,'bob',3,1.44);
- Insert into student values(3,'Jam',1,3.24);
- Insert into student values(4,'lucky',2,2.67);
- Insert into student values(5,'ram',2,4.56);

- select \* from student;
- Output

Id	Name	classID	Marks
1	Pinky	3	2.4
2	Bob	3	1.44
3	Jam	1	3.24
4	Lucky	2	2.67
5	Ram	2	4.56

# Table creation with constraint

- Create table student (id number(10), name varchar2(20), classID number(10) constraint classID\_chk check(classID<10) , marks varchar2(20));
- SQL Query to add Primary key into an existing table

```
ALTER TABLE do.StudentMaster  
ADD CONSTRAINT PK_StudentId PRIMARY  
KEY CLUSTERED (StudentId);
```

- Make sure that this column is both **not null** and **unique**

# Add new column

- ALTER TABLE table\_name ADD column\_name data\_type constraint;
- Example: 1. create a table named **members**
- CREATE TABLE **members**( member\_id NUMBER GENERATED BY DEFAULT AS IDENTITY, first\_name VARCHAR2(50), last\_name VARCHAR2(50), PRIMARY KEY(member\_id) );
- Add new column named birth\_date to members table:
- ALTER TABLE members ADD birth\_date DATE NOT NULL;
- ALTER TABLE statement applies a schema change to a table.

# Delete column

- To physically drop a column use one of the following syntaxes, depending on whether you wish to drop a single or multiple columns.
- alter table table\_name **drop** column column\_name;
- alter table table\_name **drop** (column\_name1, column\_name2);

# Questions

- Create Table Employee with attributes:
- Employee ID, Employee name, Joining Date, Designation, Basic Salary
- Input Employee details
- Add column DOB to the Employee table.
- Set Salary value by default as 5000, otherwise as the user enters.

# Primary Key

- Primary key is **column or columns that contain values uniquely identify each row in a table.**
- It cannot have NULL values. It is either an existing table column or a column that is specifically generated by database according to a defined sequence.
- Example: STUD\_NO, as well as STUD\_PHONE both, are candidate keys for relation STUDENT but STUD\_NO can be chosen as primary key (only one out of many candidate keys).

**STUDENT**

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNT RY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajasthan	India	18
4	SURESH		Punjab	India	21

**Table 1**

**STUDENT\_COURSE**

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

**Table 2**

# Candidate key

- Candidate key is also a unique key to identify a record uniquely in a table but a table can have multiple candidate keys.
- Super key is **a single key or a group of multiple keys that can uniquely identify tuples in a table**. Super keys can contain redundant attributes that might not be important for identifying tuples.
- **Primary Key is a unique and non-null key which identify a record uniquely in table**. A table can have only one primary key.
- A super key is a superset of a candidate key.
- Primary key column value can not be null. Candidate key's attributes can contain a NULL value.

# Difference between Primary and Candidate Key

S.NO	Primary Key	Candidate Key
1.	Primary key is a minimal super key. So there is one and only one primary key in a relation.	While in a relation there can be more than one candidate key.
2.	Any attribute of Primary key can not contain NULL value.	While in Candidate key any attribute can contain NULL value.
3.	Primary key can be optional to specify any relation.	But without candidate key there can't be specified any relation.
4.	Primary key specifies the important attribute for the relation.	Candidate specifies the key which can qualify for primary key.
5.	Its confirmed that a primary key is a candidate key.	But Its not confirmed that a candidate key can be a primary key.

Whenever a primary key consists of more than one attribute, it is known as a **composite key**. This key is also called **Concatenated Key**.

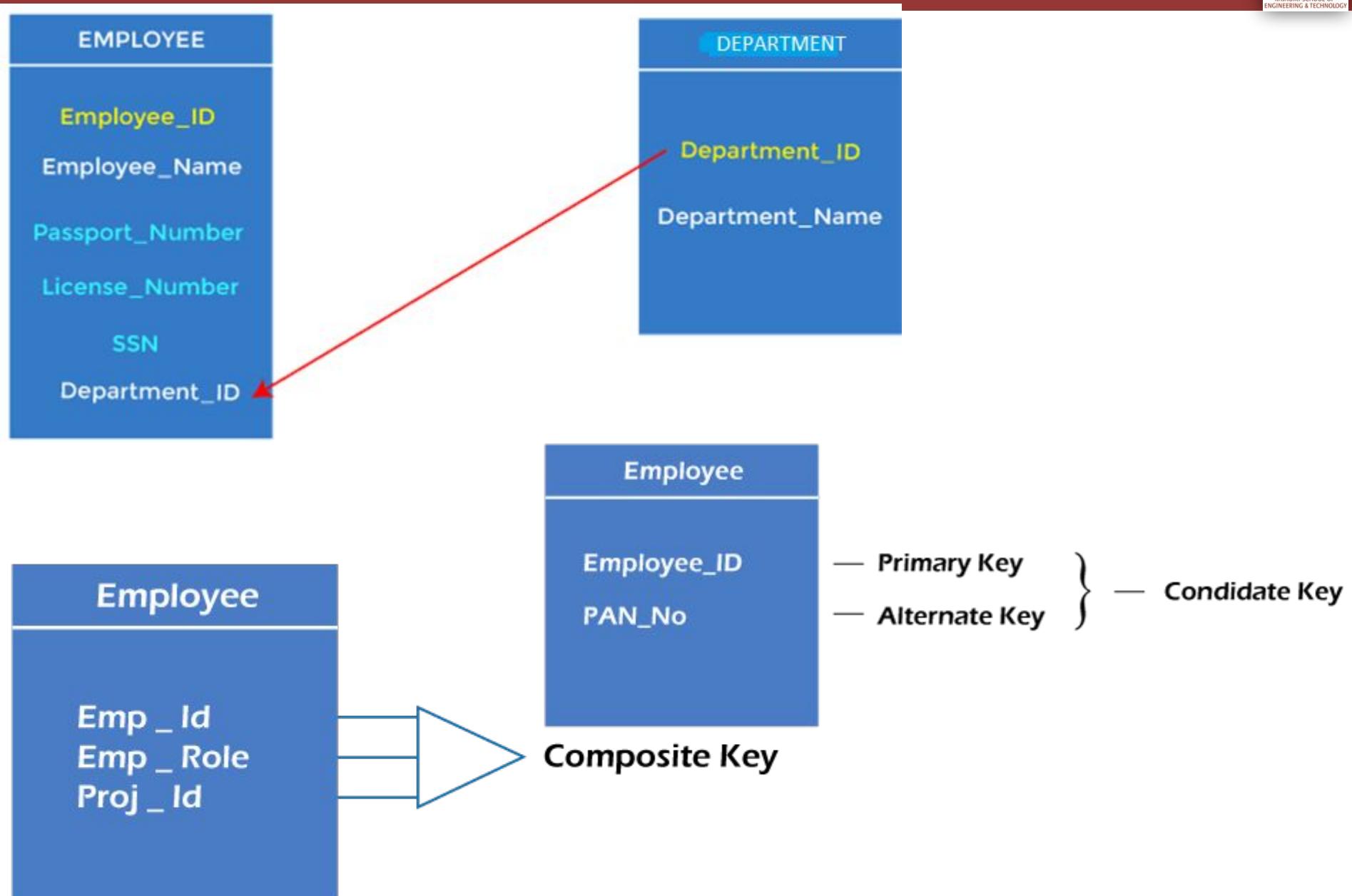
# Foreign Key

- A foreign key is **a column (or combination of columns) in a table whose values must match values of a column in some other table.**
- FOREIGN KEY constraints enforce referential integrity, which essentially says that if column value A refers to column value B, then column value B must exist.
- **A primary key is used to assure the value in the particular column is unique. The foreign key provides the link between the two tables.**
- Foreign key is a column (or columns) that references a column (most often the primary key) of another table.
- Example:
- STUD\_NO in STUDENT\_COURSE is a foreign key to STUD\_NO in STUDENT relation.

# Foreign key Example

- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities.
- So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department\_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department\_Id is the foreign key, and both the tables are related.

# Keys: Primary, Foreign, Alternate, Candidate, Composite



# Difference between Primary Key and Foreign Key

S.NO	PRIMARY KEY	FOREIGN KEY
1	A primary key is used to ensure data in the specific column is unique.	A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables.
2	It uniquely identifies a record in the relational database table.	It refers to the field in a table which is the primary key of another table.
3	Only one primary key is allowed in a table.	Whereas more than one foreign key are allowed in a table.
4	It is a combination of UNIQUE and Not Null constraints.	It can contain duplicate values and a table in a relational database.
5	It does not allow NULL values.	It can also contain NULL values.
6	Its value cannot be deleted from the parent table.	Its value can be deleted from the child table.
7	It constraint can be implicitly defined on the temporary tables.	It constraint cannot be defined on the local or global temporary tables.

# DBMS Interfaces

- **User-friendly interfaces:**
- *Menu-based*, popular for browsing on the web
- *Forms-based*, designed for general users
- *Graphics-based* (Point and Click, Drag and Drop etc.)
- *Natural language*: requests in written English (generates a high level query corresponding to the natural language request and submits to DBMS, otherwise a dialogue is started with user to clarify the request.)
- *Combinations* of the above
- **Stand-alone query language interfaces.** – Example is the terminal window for SQL Plus of Oracle.

# DBMS Programming Language Interfaces

- Programmer interfaces for embedding DML in a programming languages:
  - **Embedded Approach:** e.g embedded SQL (for C, C++, etc.), SQLJ (for Java)
  - **Procedure Call Approach:** e.g. JDBC for Java, ODBC for other programming languages
  - **Database Programming Language Approach:** e.g. ORACLE has PL/SQL, a programming language based on SQL; language incorporates SQL and its data types as integral components
- **Others:**
- Speech as Input and Output
- Parametric interfaces (e.g., bank tellers) using function keys.
  - Interfaces for the DBA:
    - a. Creating accounts
    - b. Granting authorizations
    - c. Setting system parameters and
    - d. Changing schemas or access path

# Database Administrator

- The Database administrator is responsible for the data as well as the programs that access the data from the Database.

The roles of the DBA include

- **Schema Definition**- Creating the original database schema by executing data definition statements using DDL.
- **Storage structure and access method definition**- Decides the physical locations where the files will reside on a machine/machines.
- **Schema and physical organizational modifications**- these changes are carried out to improve performance and changing needs of Organization.
- **Authentication for Data access**- grants different levels of authentication for different users accessing data using DCL. He monitors user accesses of the database.
- **Maintenance activity**- This includes
  - Periodic backing of the database onto tapes or remote servers to cater for unforeseen disasters. Carries out Database recovery when required.
  - Carries out Performance tuning of the database by changing system parametric values when required.
  - Monitors jobs and users accessing the database and schedules long running jobs appropriately.
  - Ensures storage availability for data and the upgrading of storage when the need arises.

# Oracle 10g supported Data types

- Character data types: CHAR, NCHAR, VARCHAR2 and VARCHAR, NVARCHAR2, CLOB, NCLOB, LONG.
- NUMBER data type
- DATE data type: DD-MON-YY format
- Binary data types. BLOB, BFILE, RAW, LONG RAW.
  - CHAR and NCHAR data types store fixed-length character strings.
  - VARCHAR2 and NVARCHAR2 data types store variable-length character strings. (The VARCHAR data type is synonymous with the VARCHAR2 data type.)
  - CLOB and NCLOB data types store single-byte and multibyte character strings of up to four gigabytes.

# Database Architecture

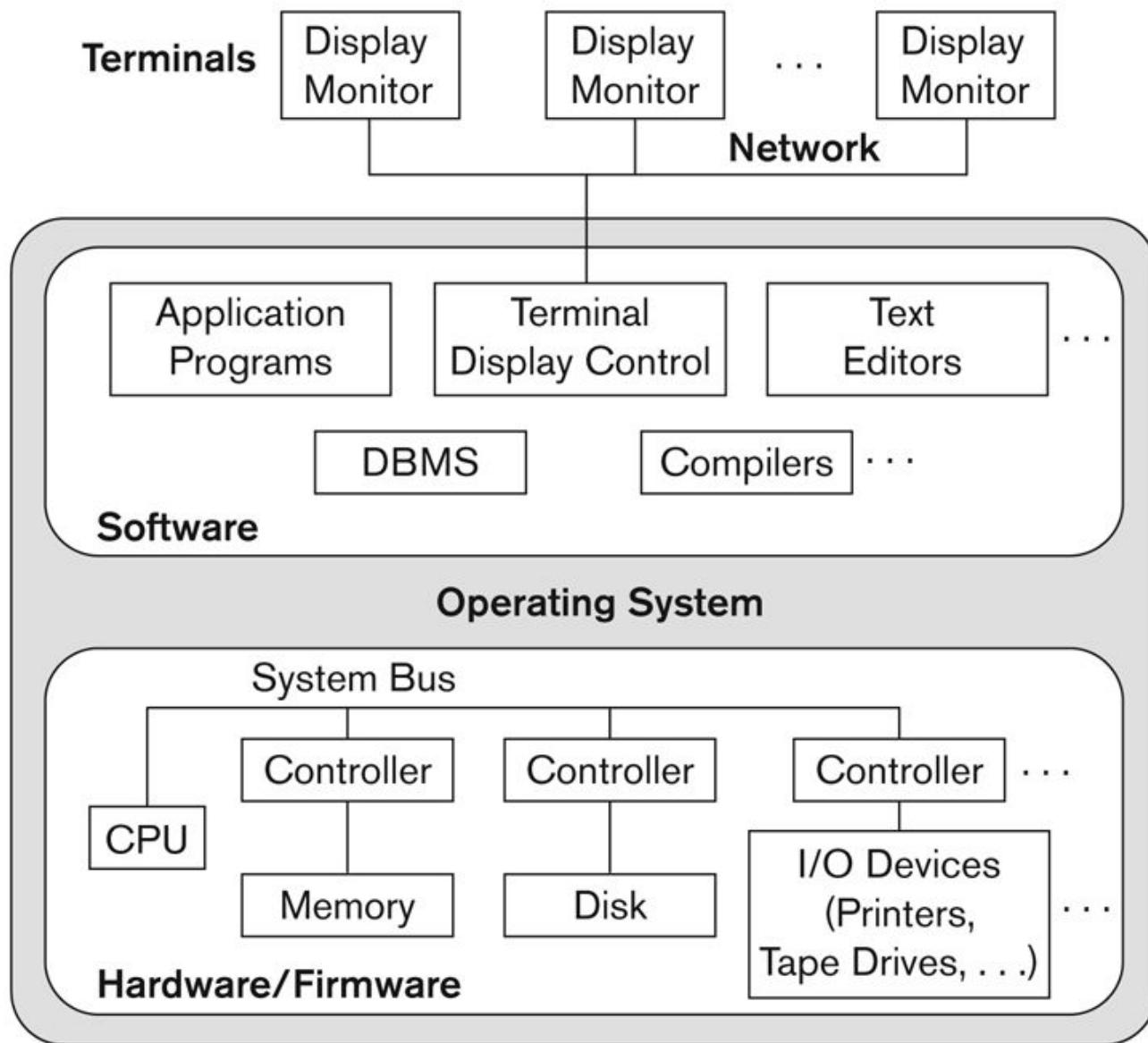
The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed

- Centralized DBMS:

- Combines everything into single system including- DBMS software, hardware, application programs, and user interface processing software.
- User can still connect through a remote terminal – however, all processing is done at centralized site.

# A Physical Centralized Architecture



**Figure 2.4**

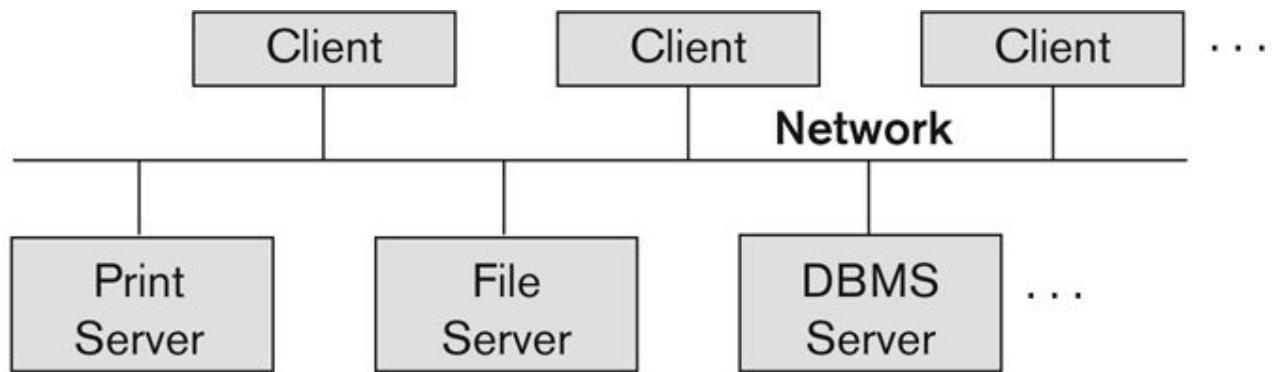
A physical centralized architecture.

# Basic 2-tier Client-Server Architectures

- Specialized Servers with Specialized functions
  - Print server
  - File server
  - DBMS server
  - Web server
  - Email server
- Clients can access the specialized servers as needed

# Logical two-tier client server architecture

**Figure 2.5**  
Logical two-tier  
client/server  
architecture.



# Clients

- Provide appropriate interfaces through a client software module to access and utilize the various server resources.
- Clients may be diskless machines or PCs or Workstations with disks with only the client software installed.
- Connected to the servers via some form of a network.
  - (LAN: local area network, wireless network, etc.)

# DBMS Server

- Provides database query and transaction services to the clients
- Relational DBMS servers are often called SQL servers, query servers, or transaction servers
- Applications running on clients utilize an Application Program Interface (**API**) to access server databases via standard interface such as:
  - ODBC: Open Database Connectivity standard
  - JDBC: for Java programming access
- Client and server must install appropriate client module and server module software for ODBC or JDBC

# Two Tier Client-Server Architecture

- A client program may connect to several DBMSs, sometimes called the data sources.
- In general, data sources can be files or other non-DBMS software that manages data.
- Other variations of clients are possible: e.g., in some object DBMSs, more functionality is transferred to clients including data dictionary functions, optimization and recovery across multiple servers, etc.

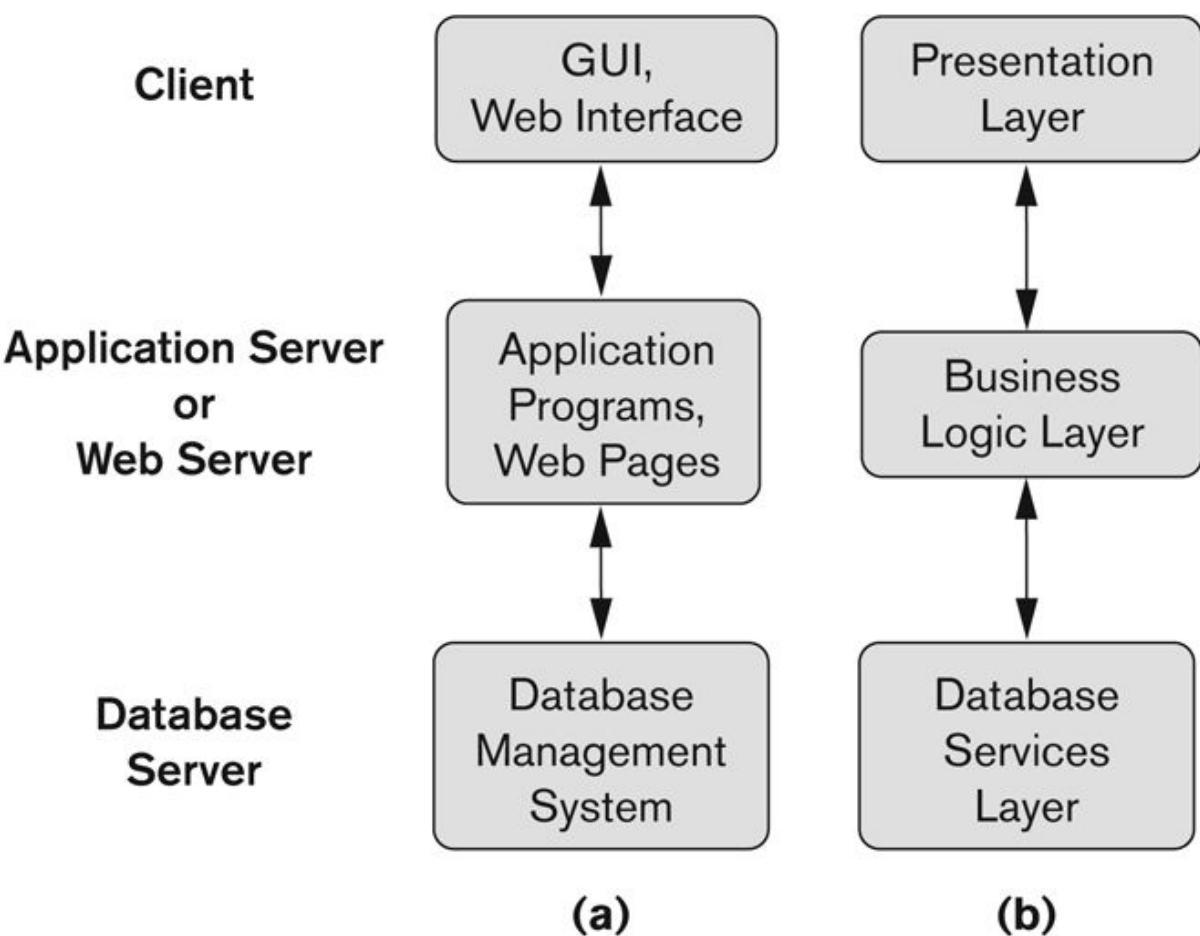
# Three Tier Client-Server Architecture

- Common for Web applications
- Intermediate Layer called Application Server or Web Server:
  - Stores the web connectivity software and the business logic part of the application used to access the corresponding data from the database server
  - Acts like a conduit for sending partially processed data between the database server and the client.
- Three-tier Architecture Can Enhance Security:
  - Database server only accessible via middle tier
  - Clients cannot directly access database server

# Three-tier client-server architecture

**Figure 2.7**

Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.



# Classification of DBMSs

- Based on the data model used
  - Traditional: Relational, Network, Hierarchical.
  - Emerging: Object-oriented, Object-relational.
- Other classifications
  - Single-user (typically used with personal computers) vs. multi-user (most DBMSs).
  - Centralized (uses a single computer with one database)  
vs. distributed (uses multiple computers, multiple databases)

# Variations of Distributed DBMSs (DDBMSs)

- Homogeneous DDBMS
- Heterogeneous DDBMS
- Federated or Multidatabase Systems
- Distributed Database Systems have now come to be known as client-server based database systems because:
  - They do not support a totally distributed environment, but rather a set of database servers supporting a set of clients.

# References

*Thank  
you*



# File Systems





# File System Interface

- File Concept
- Access Methods
- Directory Structure
- File Protection





# Introduction

- File system is the most visible aspect of the OS.
- Provides mechanism for online storage of and access to both data and programs of the OS and all users of the computer.
- A computer's OS creates and maintains the file system on a storage drive or device. The file system essentially organizes the data into files. It controls how data files are named, stored, retrieved and updated and what other information can be associated with the files -- for example, data on file ownership and user permissions.
- NTFS is one type of file system. (New Technology File System)
- Reference for NTFS
  - <https://searchwindowsserver.techtarget.com/definition/NTFS>



# File Concept





# Introduction

- Computers can store information on various storage media
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit called a **file**
- Files are **mapped** by the operating system onto nonvolatile physical devices
- A file is a **named collection of related information** that is recorded on secondary storage
- For a user's perspective, a file is the **smallest allotment** of logical secondary storage (i.e., data supposedly cannot be written to secondary storage unless written in a file)
- Files commonly represent programs (both source and object code) and data
- Data file contents may be numeric, alphabetic, alphanumeric, or binary
- Files may be **free form** (e.g., text files) or **rigidly formatted**
- In general, a file is a **sequence** of bits, bytes, lines, or records, whose meaning is defined by the file's creator and user





# File Structure

- A file has a certain **defined** structure, which depends on its type
- A **text file** is a sequence of characters organized into lines
- A **source code file** is a sequence of declarations, statements, and subroutine definitions
- An **object code file** and an **executable file** each contain a sequence of bytes organized into headers and tables of data and code understandable by a system linker and loader
- A file is **named** for the convenience of its human users, and is referred to by its name
- After a file is created, it becomes **independent** of the process that created it; other processes may read it or edit it





# File Attributes

- **Name** – the only information kept in human-readable form
- **Identifier** – unique tag (a number) that identifies file within the file system
- **Type** – used by systems that support different types of files
- **Location** – pointer to file location on a device
- **Size** – current file size in bytes, words, or blocks
- **Protection** – access controls for who can read, write, and execute
- **Time, date, and user identification** – used for documenting file creation, last modification, and last use
- Information about all files is kept in the directory structure, which also is located on secondary storage





# File Operations

- A file is an **abstract data type** with operations to
  - **Create**
  - **Write**
  - **Read**
  - **Reposition within file**
  - **Delete**
  - **Truncate (i.e., erase the contents but keep the file)**





# Open Files

- Several pieces of data are needed to manage open files:
  - **File pointer:** pointer to last read/write location; unique to **each process** that has the file open
  - **File-open count:** count of number of **simultaneous** opens for a file to allow removal of the file entry from the open-file table when the last process closes it
  - **Disk location of the file:** Information kept in memory and used to locate the file on disk
  - **Access rights:** Access mode information stored for each process





# File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information



# Access Methods





# Access Methods

- **Sequential Access**
- **Direct access**

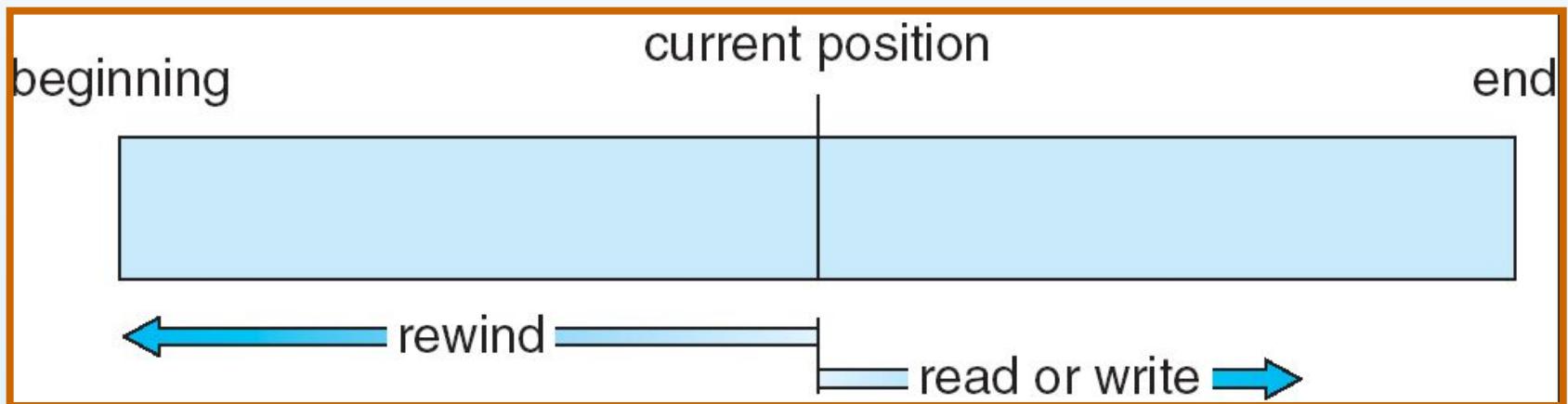
## Sequential Access

- **Information in the file is processed in order, one record after the other.**
  - i. read next- reads the next portion of file and automatically advances the file pointer which tracks the I/O location.
  - ii. write next – appends to the end of file and advances to the end of newly added material.
  - iii. reset
  - iv. skip forward
  - v. skip backward





# Sequential-access File





## Direct Access

- Here, a file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order

*read  $n$*

*write  $n$*

*position to  $n$*

$n$  = relative block number





# Simulation of Sequential Access on a Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$

$cp$  = current  
position





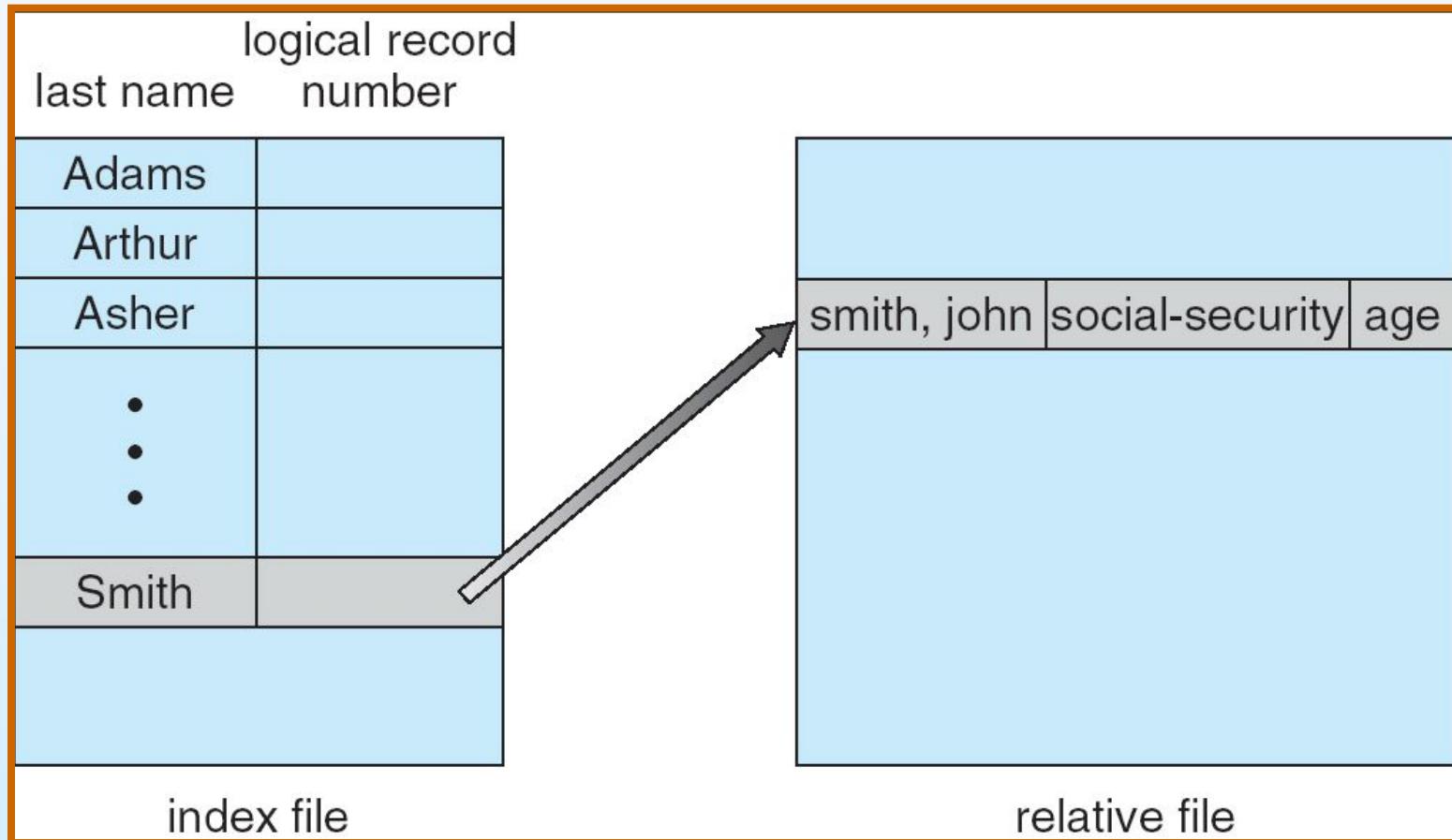
# Other Access Methods

- Involves construction of an index for the file.
- The index contains pointers to the various blocks.
- To find a record in the file, we search the index and then use the pointer to access the file directly and find the desired record





# Example of Index and Relative Files



# Directory Structure





# Introduction to directory structure

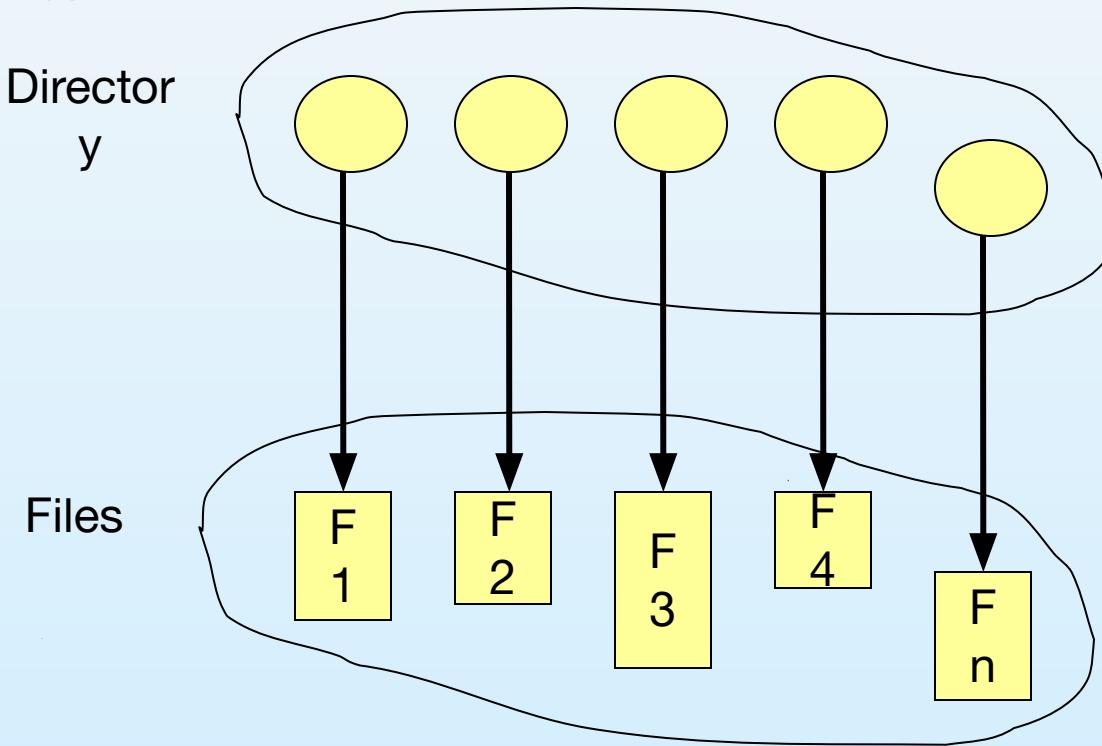
- The file systems of computers can be extensive.
- Within a file system, it is useful to aggregate files into groups and manage and act on those groups.
- This organization involves the use of directories.
- Operations that can be performed on a directory
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system





# Directory Structure

- A collection of nodes containing information about all files

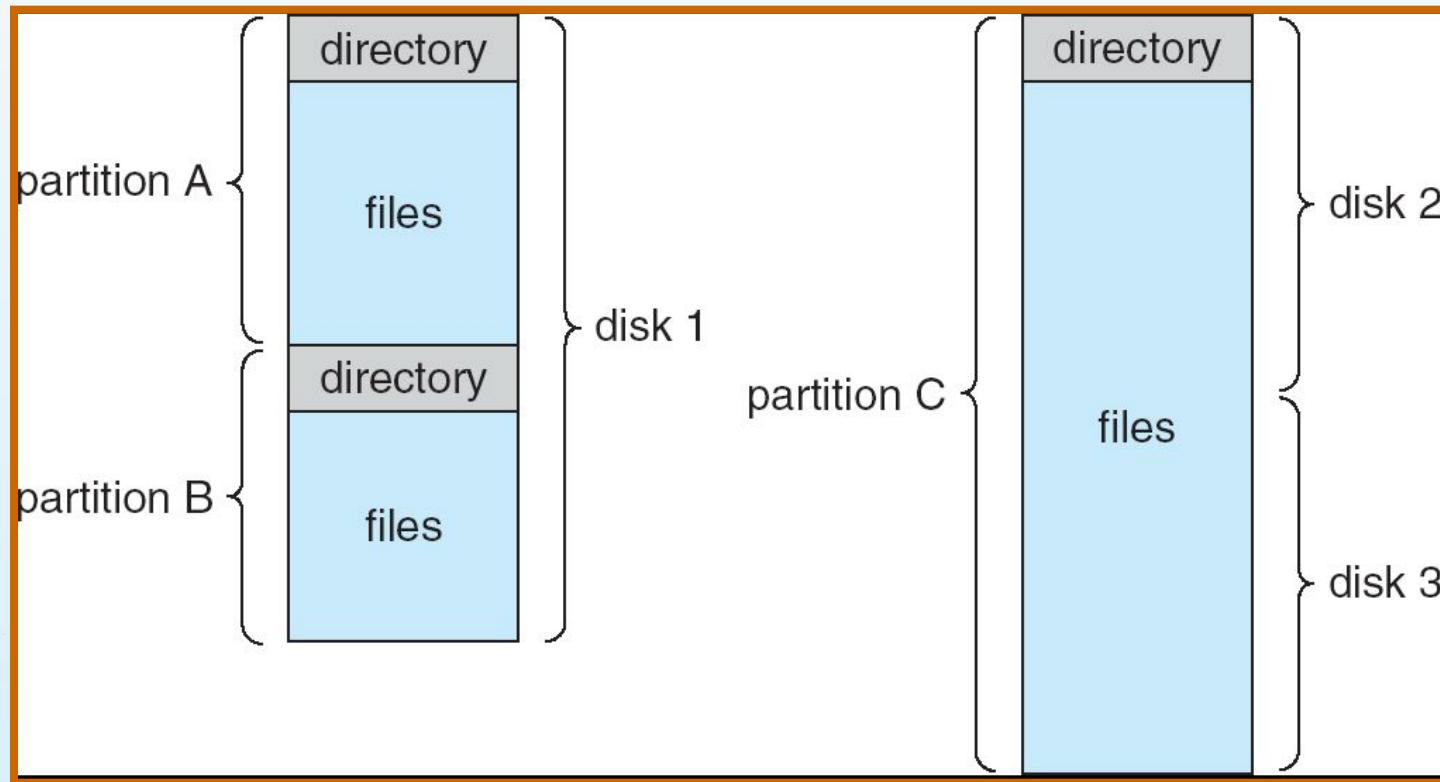


Both the directory structure and the files reside on disk. Backups of these two structures may be kept on tapes.





# A Typical File-system Organization





# Goal: Organize a directory (logically) based on the following criteria:

- **Efficiency** – locating a file quickly
- **Naming** – convenient to users
  - Two users can have **same name** for different files
  - The same file can have several **different names**
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





# Four Possible Approaches

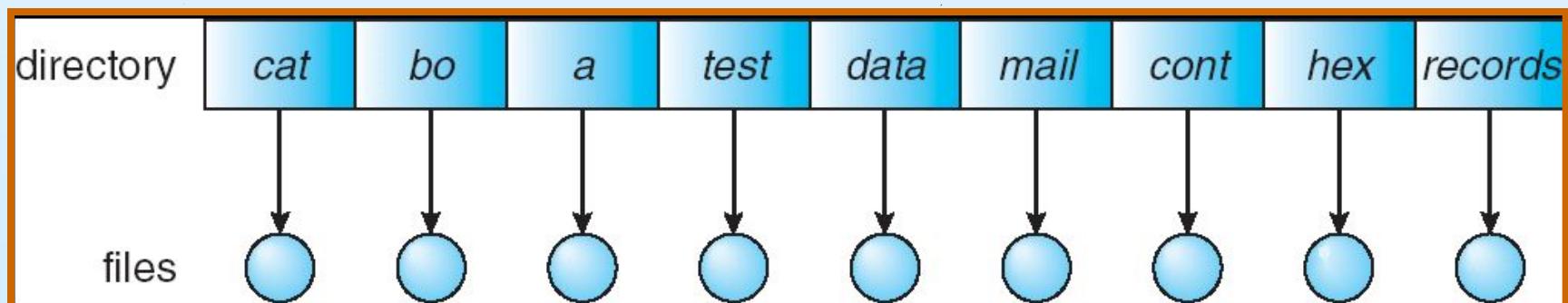
- Single-level Directory
- Two-level Directory
- Tree-structured Directories
- Acyclic Graph Directories





# Single-Level Directory

- A single directory maintained for all users
- Advantages
  - Efficiency
- Disadvantages
  - When the number of users increase or the number of files increase, single level will not work.
  - If all files are in the same directory, need unique names for the files or else Naming – collisions





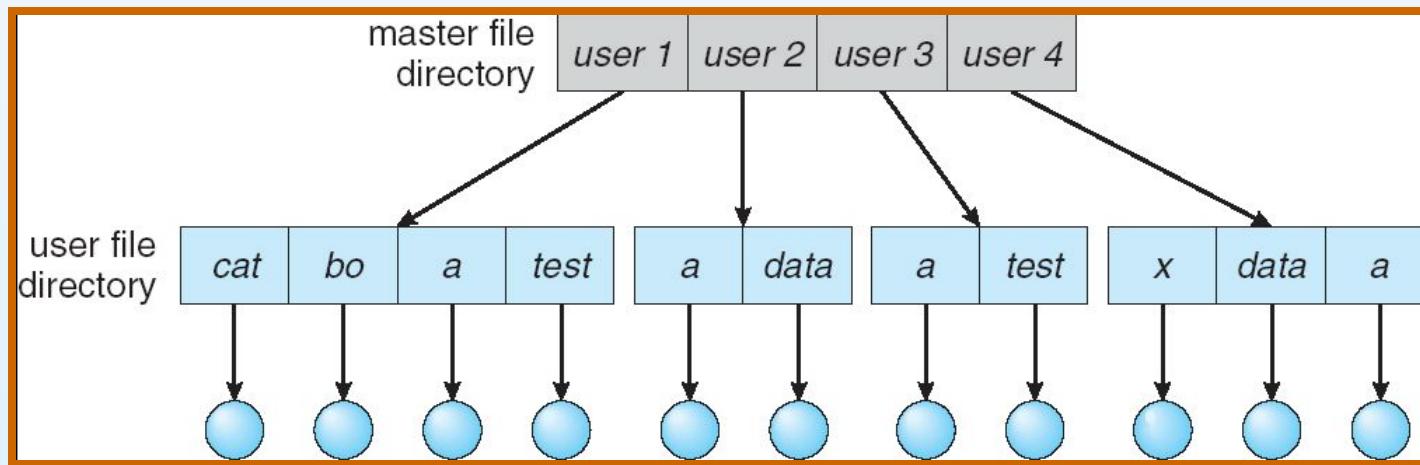
# Two-Level Directory

- Separate directory for each user
- Each user has his own **user file directory (UFD)** which lists the files of that user
- There is a **Master File Directory (MFD)** and each entry in MFD points to the UFD for a user.
- Considered as a tree
  - Root-MFD and direct descendants are the UFDs
- Advantage
  - No naming collisions
- Disadvantages
  - Isolates one user from another (If 2 users want to access the same file)





# Two-Level Directory





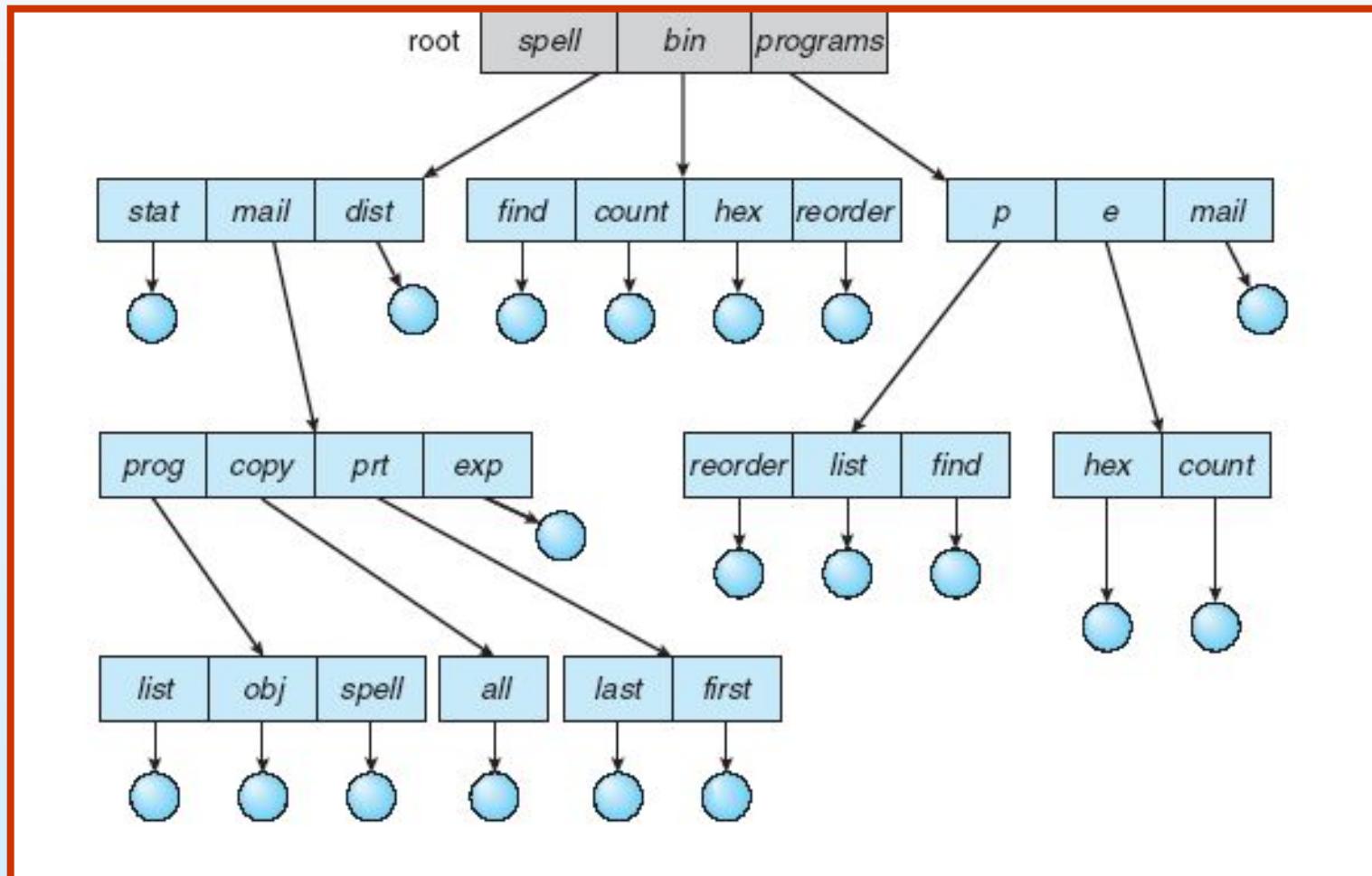
# Tree-Structured Directories

- Extend two- level directory to a tree of arbitrary height
- This helps users to create their own subdirectories and organize their files.
- Tree is the most common directory structure
- The tree has a root directory and every file in the system has a unique path name.





# Tree-Structured Directories





- In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process.
- If a file not in the current directory is needed, specify the path name.
- System calls help in all the operations (cd.., )



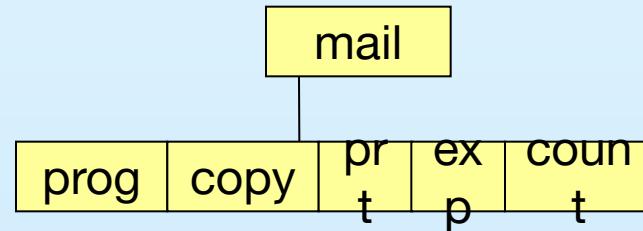


# Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file
  - `rm <file-name>`
- Creating a new subdirectory is done in current directory
  - `mkdir <dir-name>`

Example: if in current directory `/mail`

`mkdir count`





# Acyclic-Graph Directories

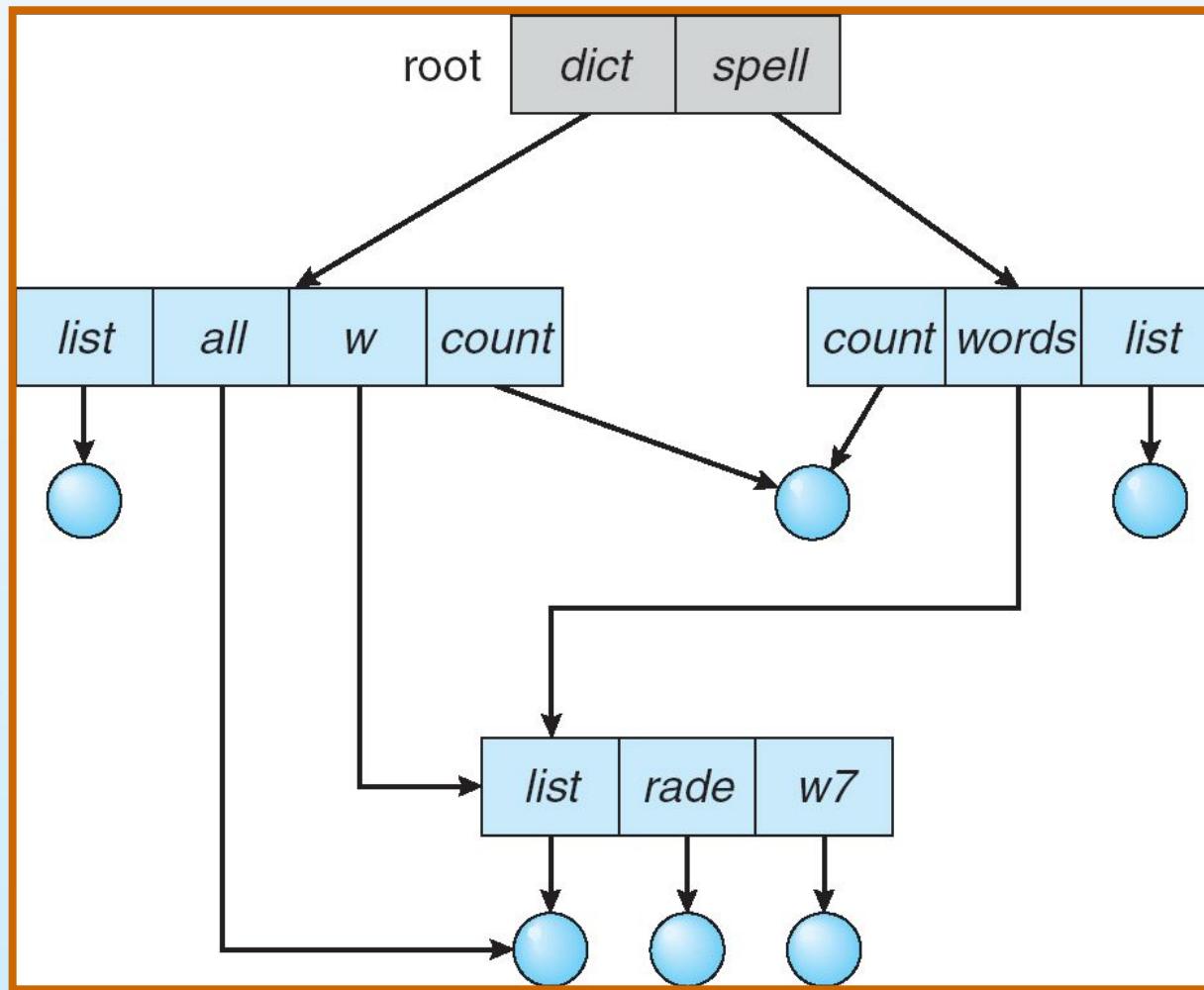
- Consider 2 programmers working on a joint project.
- The associated files can be placed in a subdirectory. But both of them want the subdirectory to be in their own directories
- The common subdirectory must be shared.
- A shared directory or file will exist in the file system in 2 or more places at once.
- An acyclic graph allows directories to share subdirectories and files.
- The same file can be in 2 different directories.( not 2 copies)





# Acyclic-Graph Directories

- Have shared subdirectories and files





# Acyclic-Graph Directories (Cont.)

- Same advantages as tree-structured directory
  - In addition, the same file or directory may have a reference that appears in two or more directories
- Disadvantage is that its structure is more complex
  - The same file or directory may be referred to by many names
  - Need to be cautious of dangling pointers when files are deleted
- Solutions to deletion
  - Just delete the link
  - Preserve the file until all links (i.e., references) are deleted
- This requires a new directory entry type called a **link**
  - Keep a count of the number of references

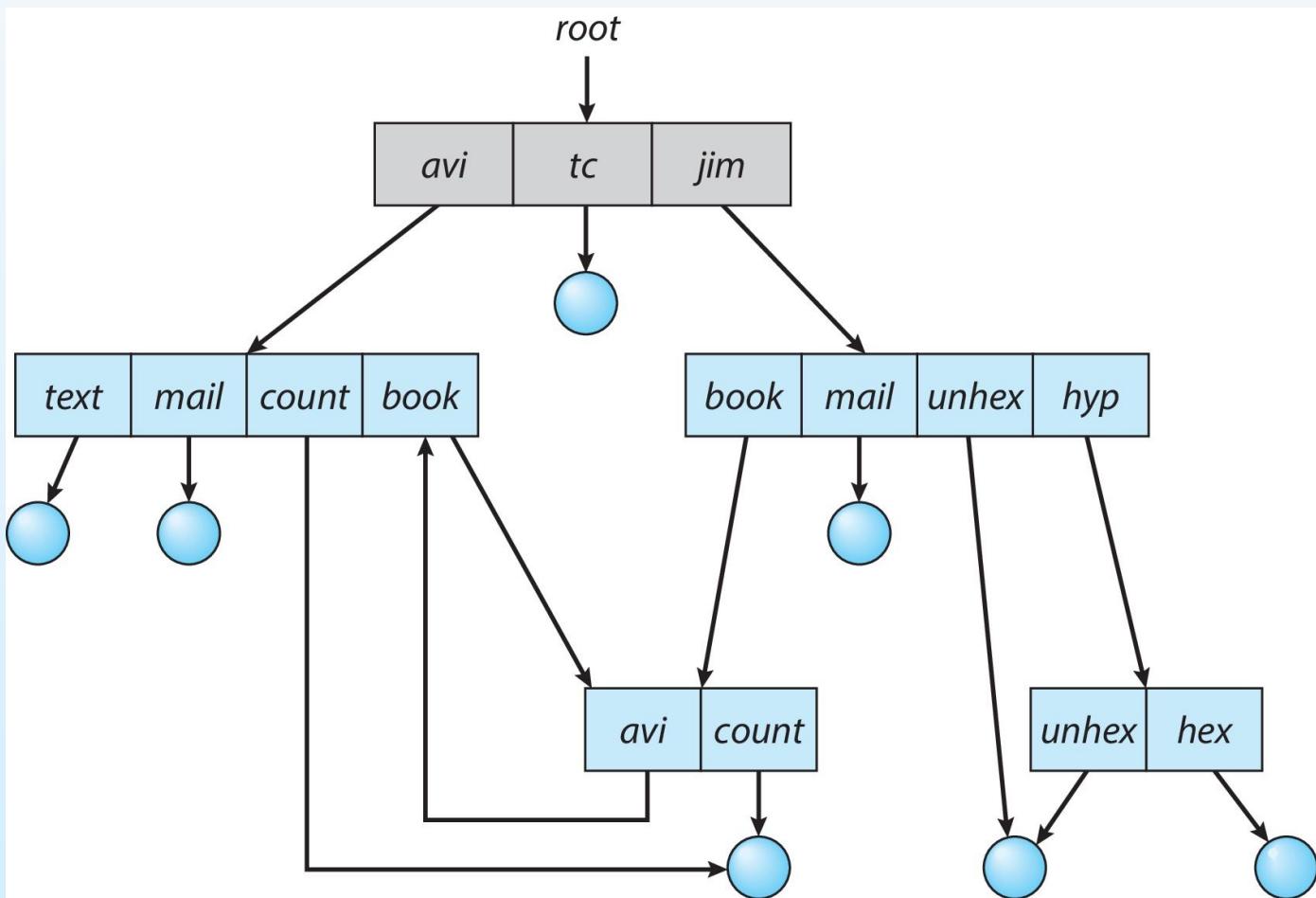




# General graph directory

- A serious problem in acyclic graph directories is to ensure that there are no cycles.
- If we add files and subdirectories to an existing tree-structured directory, it preserves the tree structured nature.
- When we add links, the tree structure is destroyed resulting in a simple graph structures
- Advantage: simple to traverse the graph and determine whether there are references to a file
- We need not traverse it twice





# File Protection





# Protection

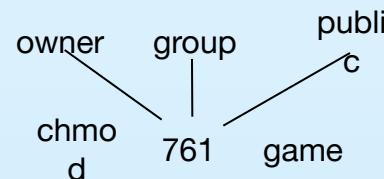
- File owner/creator should be able to control:
  - what can be done to a file
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**





# Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users
  - RWX
  - a) **owner access** 7 ⇒ 1 1 1
  - RWX
  - b) **group access** 6 ⇒ 1 1 0
  - RWX
  - c) **public access** 1 ⇒ 0 0 1
- A system administrator creates a group (unique name), say G, and adds some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game





# A Sample UNIX Directory Listing

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/





# Directory Implementation





# Directory Implementation

- Selection of directory allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system
- Three approaches
  - Direct indexing of a linear list
  - List indexing via a hash function
  - Non-linear structure such as a B-tree





# Directory Implementation

- Use a linear list of names with pointers to the data blocks.
- To create a new file, the entire directory is to be searched to ensure that no other file with same name exists.
- If so, we add the file
- On deletion, the space is released.
- Disadvantage
  - Finding a file needs a linear search
- Directory information is used frequently and hence the users will notice that access is slow.
- OS uses cache to keep the most recently accessed directory information
- A sorted list uses binary search and reduces the search time, but this may complicate creating and deleting files.





# Hash Table

- Here a linear list stores the files, but a hash data structure is also used.
- Takes a value computed from the file name and returns a pointer to the file name in the linear list
- Greatly reduces the directory search time
  - **Can result in collisions** – situations where two file names hash to the same location

## Disadvantage

- Fixed size and dependence of hash function on the size
  - Assume, we have a linear probing hash table with 64 entries. Hash function used is  $x \% 64$
  - If we later try to create a 65<sup>th</sup> file, we need a new hash function and must reorganize the existing directory entries to reflect their new hash values.

Solution: Use a chained-overflow hash table. Each entry can be a linked list instead of an individual value





# Allocation Methods





# Allocation Methods

- Many files are stored on the same disk.
- Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly
- Three methods exist for allocating disk space
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**





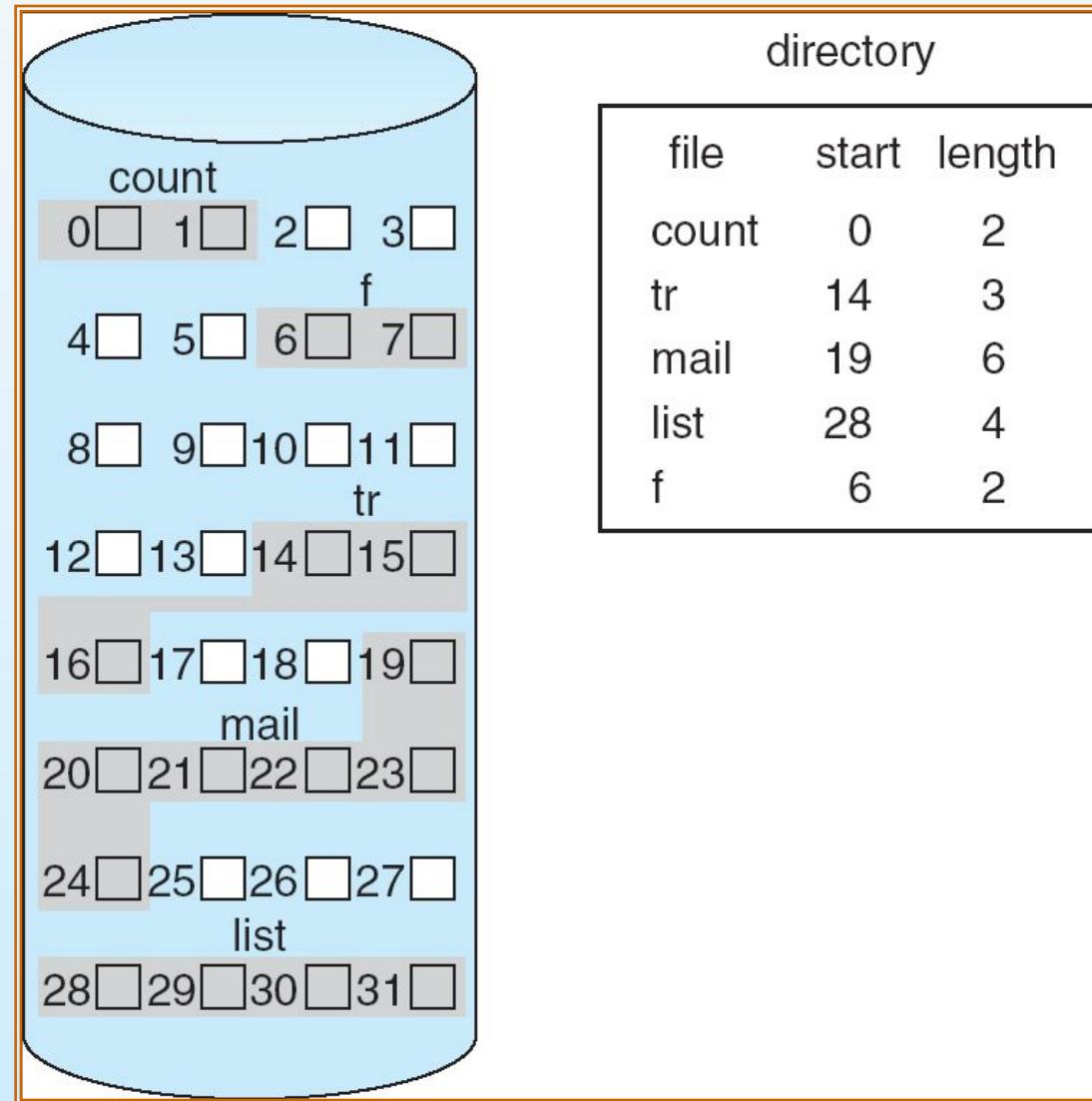
# Contiguous Allocation

- Here, each file occupy a set of contiguous blocks in the disk.
- Disk addresses define a linear ordering on the disk.
- Accessing block  $b+1$  after block  $b$  requires no head movement . When head movement is required, it will be from last sector of one cylinder to first sector of the next cylinder.
- The head need move only from one track to the next. So number of disk seeks required for accessing contiguously allocated files is minimal
- Accessing files is also easy





# Contiguous Allocation (continued)





## Disadvantages

- Finding space for a new file ( a case of dynamic storage allocation problem)
  - External fragmentation
- Determining how much space is needed for a file.
  - When it is created , the total amount of space must be found and allocated
  - Difficult to extend the file
- Preallocation also difficult. Some files grow only over a long period of time.
- Solution: Initially a contiguous chunk of space is allocated. If not enough, another chunk of space known as extent is added





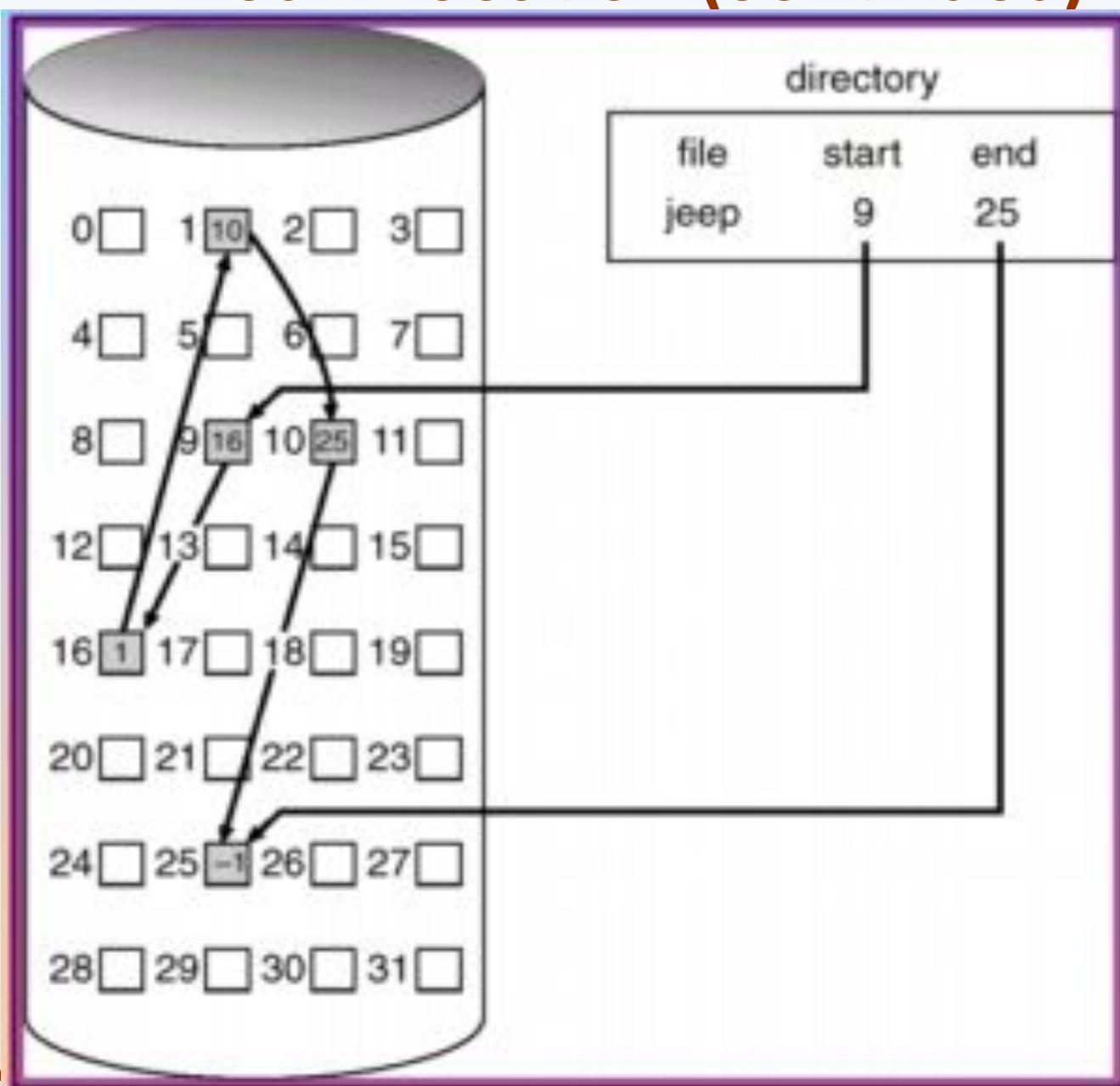
# Linked Allocation

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block
  - This new block is written to and is linked to the end of the file
- Reading from a file requires only reading blocks by following the pointers from block to block
- Advantages
  - There is no external fragmentation
  - Any free blocks on the free list can be used to satisfy a request for disk space
  - The size of a file need not be declared when the file is created
  - A file can continue to grow as long as free blocks are available
    - It is never necessary to compact disk space for the sake of linked allocation (however, file access efficiency may require it)





# Linked Allocation (continued)





# Linked Allocation (continued)

- Disadvantages
  - Can only be used effectively for sequential access of files
    - Each access to a file block requires a disk access, and some may also require a disk seek
    - It is inefficient to support direct access capability
  - Disk space is required to store the block pointers
    - One solution is the clustering of a certain constant number of blocks (e.g., 4)
  - Relies on the integrity of the links – an error might result in a pointer value becoming corrupt and then pointing into the free-space list or to the blocks of another file
    - A partial solution is a doubly linked list or storing a relative block number or the file name in each block (these schemes all require space and algorithm overhead)





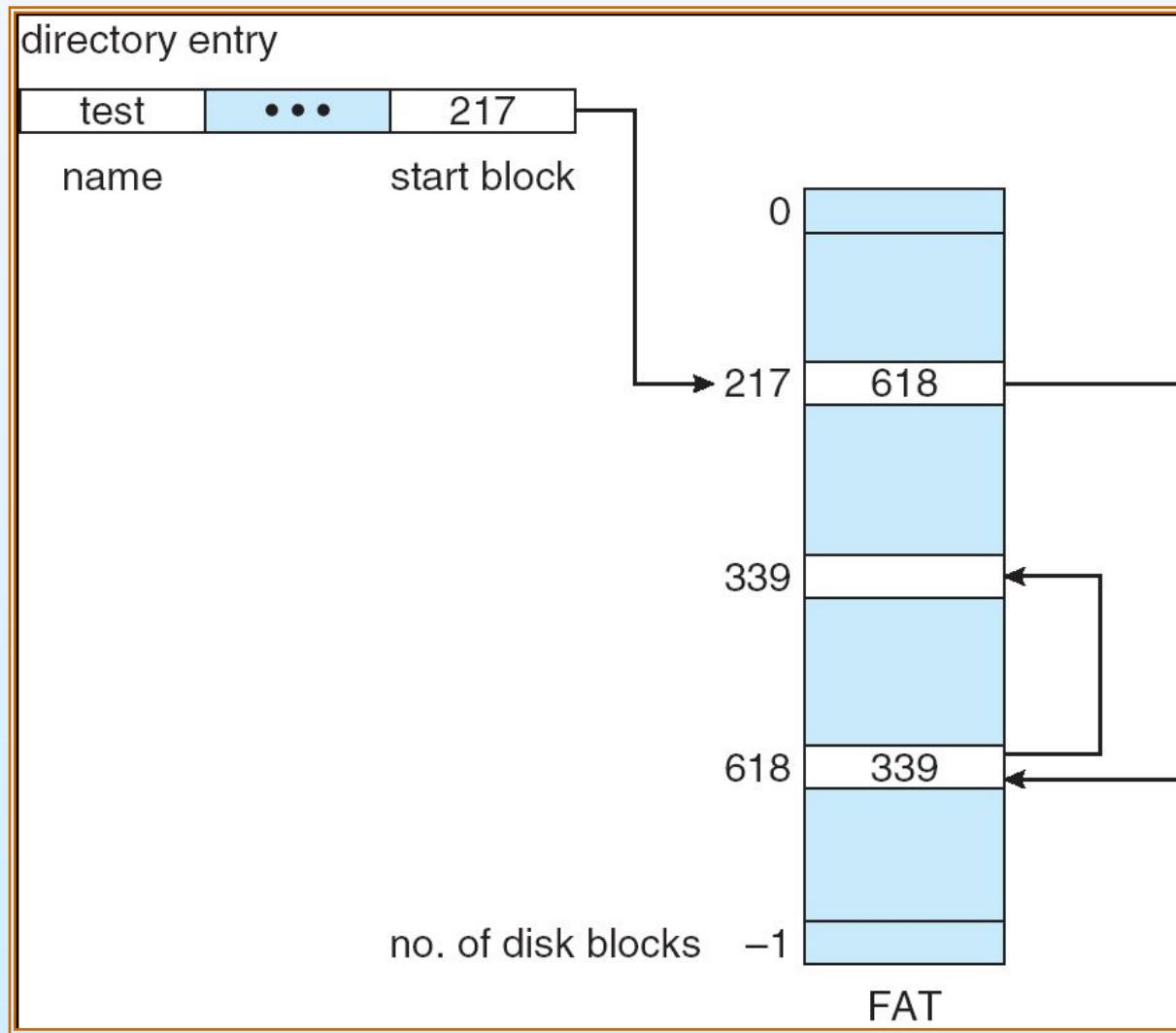
# Linked Allocation (continued)

- One variation on linked allocation is the file allocation table (FAT) used by MS-DOS
  - A section of the disk on each volume(partition) contains the FAT
  - The FAT has one entry for each disk block and is indexed by block number
  - The directory entry contains the block number of the first block of the file
  - The table entry indexed by the block number contains the block number of the next block in the file
  - The last block contains a special end-of-file value as the table entry
  - Unused blocks are indicated by a zero table value
  - To allocate a new block to a file
    - Find the first zero-valued table entry
    - Replace the previous end-of-file value with the address of the new block
  - Disadvantage – can result in a significant number of disk head seeks
  - Advantage – random-access time is improved because the FAT can be checked





# File-Allocation Table (FAT)





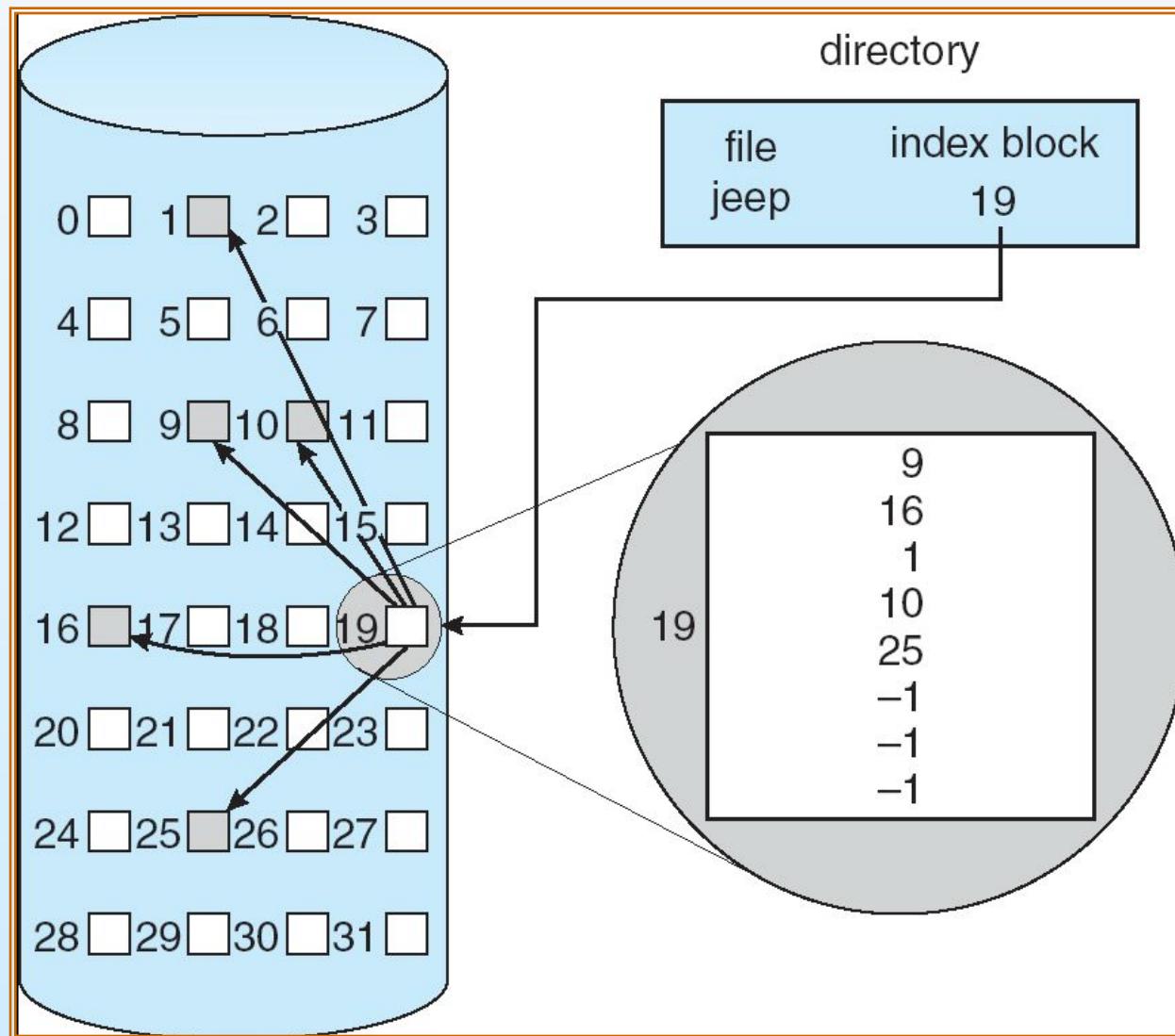
# Indexed Allocation

- Solves the problems of linked allocation by bringing all the pointers (for a file's blocks) together into one location called the *index block*
- Each file has its own index block, which is an array of disk-block addresses
- Each entry in the index block points to the corresponding block of the file
- The directory contains the address of the index block
- Finding and reading a specific block in a file only requires the use of the pointer in the index block
- When a file is created
  - The pointer to the index block is set to nil
  - When a new block is first written, it is obtained from the free-space management system and its address is put in the index block
- Supports direct access without suffering from external fragmentation
- Requires the additional space of an index block for each file
- Disadvantages
  - Suffers from some of the same performance problems as linked allocation
    - Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume





# Example of Indexed Allocation





# Goals of Protection

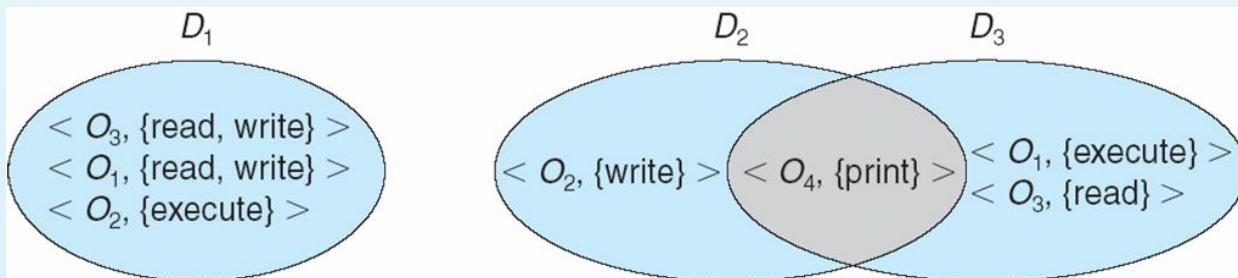
- In protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so





# Domain Structure

- Access-right =  $\langle \text{object-name}, \text{rights-set} \rangle$   
where *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights





# Access Matrix

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **Access(i, j)** is the set of operations that a process executing in Domain<sub>i</sub> can invoke on Object<sub>j</sub>

object domain \ object	$F_1$	$F_2$	$F_3$	printer
$D_1$	read		read	
$D_2$				print
$D_3$		read	execute	
$D_4$	read write		read write	





# Use of Access Matrix

- If a process in Domain  $D_i$  tries to do “op” on object  $O_j$ , then “op” must be in the access matrix
- User who creates object can define access column for that object
- Processes should be able to switch from one domain to another.
- Switching allowed only if access right **switch**  $\in$  access (I,j)

object domain \	$F_1$	$F_2$	$F_3$	laser printer	$D_1$	$D_2$	$D_3$	$D_4$
$D_1$	read		read			switch		
$D_2$				print			switch	switch
$D_3$		read	execute					
$D_4$	read write		read write		switch			





A process in  $D_0$  can switch to running in domain  $D_1$

*objects*

	$F_0$	$F_1$	Printer	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$
$D_0$	read	read-write	print	-	switch	switch		
$D_1$	read-write-execute	read			-			
$D_2$	read-execute				switch	-		
$D_3$		read	print					
$D_4$			print					



- Allowing control to change the contents of the access matrix entries requires 3 additional operations
  - **copy**
  - **owner**
  - **control**





# Copy

- Ability to copy an access right from one domain (row) to another is denoted by asterisk \* appended to the access right
- The copy right allows the access right to be copied only within the column (ie, for the object) for which the right is defined

object domain	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute		

(a)

object domain	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute	read	

(b)





# Owner

- Mechanism to allow addition of new rights and removal of some rights.
- If access(i, j) includes **owner** rights, a process executing in domain Di can add and remove any right in any entry in column j

object domain \ F <sub>i</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	owner execute		write
D <sub>2</sub>		read* owner	read* owner write
D <sub>3</sub>	execute		

(a)

object domain \ F <sub>i</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	owner execute		write
D <sub>2</sub>		owner read* write*	read* owner write
D <sub>3</sub>		write	write

(b)





# Control rights

- The copy and owner rights allow a process to change entries in a column.
- If we need to change the entries in a row, we need the **control** right.
- The control right is only applicable to domain objects.
- If access  $(i, j)$  includes the control right, then a process executing in domain  $D_i$  can remove any access right from row  $j$

		objects							
		$F_0$	$F_1$	Printer	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$
domains of protection	$D_0$	read owner	read-write	print	-	switch	swtich		
	$D_1$	read-write-execute	read*			-			control
	$D_2$	read-execute				swtich	-		
	$D_3$		read	print					
	$D_4$			print					

A process executing in  $D_1$  can modify any rights in domain  $D_4$





# Implementation of Access Matrix

- How can the access matrix be implemented effectively?
- Generally, a sparse matrix
- Three methods
  - Global table
  - Access List for objects
  - Capability lists for Domains





# Global table

- Simplest implementation
- Store ordered triples **<domain, object, rights-set>** in table
- A requested operation M on object  $O_j$  within domain  $D_i$  -> search table for  $\langle D_i, O_j, R_k \rangle$ 
  - with  $M \in R_k$
- But table could be large -> won't fit in main memory and so additional I/O needed.
- Difficult to group objects (consider an object that all domains can read)





# Access lists for objects

- Each column implemented as an access list for one object
- Resulting per-object list consists of ordered pairs `<domain, rights-set>` defining all domains with non-empty set of access rights for the object
- Easily extended to contain default set -> If  $M \in$  default set, also allow access

Each column = Access-control list for one object  
Defines who can perform what operation

Domain 1 = Read, Write

Domain 2 = Read

Domain 3 = Read





		objects							
		$F_0$	$F_1$	Printer	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$
domains of protection	$D_0$	read owner	read-write	print	-	-	-	-	-
	$D_1$	read-write-execute	read*		-				
	$D_2$	read-execute			switch	-			
	$D_3$		read	print					
	$D_4$			print					

ACL for file  $F_0$



# Capability list for domains

- Instead of object-based, list is domain based
- **Capability list** for domain is list of objects together with operations allowed on them
- Object represented by its name or address, called a **capability**
- Execute operation M on object  $O_j$ , process requests operation and specifies capability as parameter
  - Possession of capability means access is allowed
- Each Row = Capability List (like a key)  
For each domain, what operations allowed on what objects
  - Object F1 – Read
  - Object F4 – Read, Write, Execute
  - Object F5 – Read, Write, Delete, Copy





## objects

domains of protection

	F <sub>0</sub>	F <sub>1</sub>	Printer	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
D <sub>0</sub>	read owner	read-write	print	-	switch	swtich		
D <sub>1</sub>	read-write-execute	read*			-			
D <sub>2</sub>	read-execute				swtich	-		
D <sub>3</sub>		read	print					
D <sub>4</sub>			print					

Capability list for domain D<sub>1</sub>





# Module1 Recap





# Mass-Storage Management (Module 1)

- The OS is responsible for the following disk management activities
  - Free-space management
  - Storage allocation
  - Disk scheduling





# Storage Management

- The operating system provides a uniform, logical view of information storage
  - The OS is responsible for the following file management activities:
    - Creating and deleting files and directories
    - Supporting operations manipulate files and directories
    - Mapping files onto secondary storage
    - Backing up files onto stable (non-volatile) storage media



# Thank You

