# DATABASE ENGINEERING

# DATABASE ENGINEERING

LECTURE NOTES

**Prepared by**

**Dr. Subasish Mohapatra**

**Department of Computer Science and Application**

**College of Engineering and Technology, Bhubaneswar**

**Biju Patnaik University of Technology, Odisha**

# SYLLABUS

Course Code: PCCS4204 Subject Name: **DataBase Engineering** Credit:4

**Module1: (12 Hrs)**

Introduction to database Systems, Basic concepts &Definitions, Data Dictionary, DBA, File-oriented system vs. Database System, Database Language.

Database System Architecture-Schemas, Sub Schemas & Instances, 3-level database architecture, Data Abstraction, Data Independence, Mappings, Structure, Components & functions of DBMS, Data models, Mapping E-R model to Relational, Network and Object Oriented Data models, types of Database systems,

Storage Strategies**:** Detailed Storage Architecture, Storing Data, Magnetic Disk, RAID, Other Disks, Magnetic Tape, Storage Access, File & Record Organization, File Organizations & Indexes, Order Indices, B+ Tree Index Files, Hashing

**Module2: (16 Hrs)**

Relational Algebra, Tuple & Domain Relational Calculus, Relational Query Languages: SQL and QBE.

Database Design :-Database development life cycle(DDLC),Automated design tools, Functional dependency and Decomposition, Dependency Preservation & lossless Design, Normalization, Normal forms:1NF, 2NF,3NF,and BCNF, Multi-valued Dependencies, 4NF & 5NF.

Query processing and optimization: Evaluation of Relational Algebra Expressions, Query optimization.

**Module3: (12 Hrs)**

Transaction processing and concurrency control: Transaction concepts, concurrency control, locking and Timestamp methods for concurrency control.

Database Recovery System: Types of Data Base failure & Types of Database Recovery, Recovery techniques

Advanced topics: Object-Oriented & Object – Relational Database, Parallel & Distributed Database, Introduction to Data warehousing & Data Mining

**Text Books:**

1. Database System Concepts by Sudarshan, Korth (McGraw-Hill Education)
2. Fundamentals of Database System By Elmasari &Navathe- Pearson Education

**References Books:**

(1) An introduction to Database System – Bipin Desai, Galgotia Publications

(2) Database System: concept, Design & Application by S.K.Singh (Pearson Education)

(3) Database management system by leon &leon (Vikas publishing House).

(4) Database Modeling and Design: Logical Design by Toby J. Teorey, Sam S. Lightstone, and Tom Nadeau, “”, 4th Edition, 2005, Elsevier India Publications, New Delhi

(5) Fundamentals of Database Management System – Gillenson, Wiley India

# CONTENTS

# Module-1:

## LECTURE-1: Introduction to Data

### Introduction:

In computerized information system data are the basic resource of the organization. So, proper organization and management for data is required for organization to run smoothly. Database management system deals the knowledge of how data stored and managed on a computerized information system. In any organization, it requires accurate and reliable data for better decision making, ensuring privacy of data and controlling data efficiently.

The examples include deposit and/or withdrawal from a bank, hotel, airline or railway reservation, purchase items from supermarkets in all cases, a database is accessed.

### What is data?

Data are the known facts or figures that have implicit meaning. It can also be defined as it is the representation of facts, concepts or instructions in a formal manner, which is suitable for understanding and processing. Data can be represented in alphabets (A-Z, a-z), digits (0-9) and using special characters (+,-.#,$, etc)

e.g: 25, "ajit" etc.

### Information:

Information is the processed data on which decisions and actions are based. Information can be defined as the organized and classified data to provide meaningful values.

Eg: "The age of Ravi is 25"

### File:

File is a collection of related data stored in secondary memory.

### File Oriented Approach:

The traditional file oriented approach to information processing each application has a separate master file and its own set of personal file. In file oriented approach the program dependent on the files and files dependent upon the programs.

### Disadvantages of file oriented approach:

1) **Data redundancy and inconsistency:**
   The same information may be written in several files. This redundancy leads to higher storage and access cost. It may lead data inconsistency that is the various copies of the same data may present at multiple places for example a changed customer address may be reflected in single file but not else where in the system.

2) **Difficulty in accessing data :**
   The conventional file processing system do not allow data to be retrieved in a convenient and efficient manner according to user choice.

3) **Data isolation :**
   Because data are scattered in various files and files may be in different formats with new application programs to retrieve the appropriate data is difficult.

4) **Integrity Problems:**

Developers enforce data validation in the system by adding appropriate code in the various application program. How ever when new constraints are added, it is difficult to change the programs to enforce them.

5) **Atomicity:**
It is difficult to ensure atomicity in a file processing system when transaction failure occurs due to power failure, networking problems etc. (atomicity: either all operations of the transaction are reflected properly in the database or non are)

6) **Concurrent access:**
In the file processing system it is not possible to access the same file for transaction at same the time.

7) **Security problems:**
There is no security provided in file processing system to secure the data from unauthorized user access.

# LECTURE-2: DBMS

## Database:

A database is organized collection of related data of an organization stored in formatted way which is shared by multiple users.

The main feature of data in a database are:

1. It must be well organized
2. It is related
3. It is accessible in a logical order without any difficulty
4. It is stored only once

For example consider the roll no, name, address of a student stored in a student file. It is collection of related data with an implicit meaning. Data in the database may be persistent, integrated and shared.

**Persistent:**

If data is removed from database due to some explicit request from user to remove.

**Integrated:**

A database can be a collection of data from different files and when any redundancy among those files are removed from database is said to be integrated data.

**Sharing Data:**

The data stored in the database can be shared by multiple users simultaneously without affecting the correctness of data.

**Why Database:**

In order to overcome the limitation of a file system, a new approach was required. Hence a database approach emerged. A database is a persistent collection of logically related data. The initial attempts were to provide a centralized collection of data. A database has a self describing nature. It contains not only the data sharing and integration of data of an organization in a single database.

A small database can be handled manually but for a large database and having multiple users it is difficult to maintain it. In that case a computerized database is useful.
The advantages of database system over traditional, paper based methods of record keeping are:

- **Compactness**: No need for large amount of paper files
- **Speed**: The machine can retrieve and modify the data more faster way then human being
- **Less drudgery**: Much of the maintenance of files by hand is eliminated
- **Accuracy:** Accurate, up-to-date information is fetched as per requirement of the user at any time.

**Database Management System (DBMS):**

A database management system consists of collection of related data and refers to a set of programs for defining, creation, maintenance and manipulation of a database.

**Function of DBMS:**

1. **Defining database schema**: it must give facility for defining the database structure also specifies access rights to authorized users.
2. **Manipulation of the database:** The dbms must have functions like insertion of record into database, updation of data, deletion of data, retrieval of data
3. **Sharing of database:** The DBMS must share data items for multiple users by maintaining consistency of data.
4. **Protection of database:** It must protect the database against unauthorized users.
5. **Database recovery:** If for any reason the system fails DBMS must facilitate data base recovery.

**Advantages of DBMS:**

**Reduction of redundancies:**
Centralized control of data by the DBA avoids unnecessary duplication of data and effectively reduces the total amount of data storage required avoiding duplication in the elimination of the inconsistencies that tend to be present in redundant data files.

**Sharing of Data:**
A database allows the sharing of data under its control by any number of application programs or users.

**Data Integrity:**
Data integrity means that the data contained in the database is both accurate and consistent. Therefore data values being entered for storage could be checked to ensure that they fall with in a specified range and are of the correct format.

**Data Security:**
The DBA who has the ultimate responsibility for the data in the dbms can ensure that proper access procedures are followed including proper authentication to access to the DataBase System and additional check before permitting access to sensitive data.

**Conflict Resolution:**
DBA resolve the conflict on requirements of various user and applications. The DBA chooses the best file structure and access method to get optional performance for the application.

**Data Independence:**
Data independence is usually considered from two points of views; physically data independence and logical data independence.

***Physical Data Independence*** allows changes in the physical storage devices or organization of the files to be made without requiring changes in the conceptual view or any of the external views and hence in the application programs using the data base.

***Logical Data Independence*** indicates that the conceptual schema can be changed without affecting the existing external schema or any application program.

**Disadvantage of DBMS:**

1. DBMS software and hardware (networking installation) cost is high
2. The processing overhead by the dbms for implementation of security, integrity and sharing of the data.
3. Centralized database control
4. Setup of the database system requires more knowledge, money, skills, and time.
5. The complexity of the database may result in poor performance.

# LECTURE-3: 3 level Architecture of DBMS

## Database Basics:

### Data Item:

The data item is also called as field in data processing and is the smallest unit of data that has meaning to its users.
Eg: “e101”, ”sumit”

### Entities and attributes:

An entity is a thing or object in the real world that is distinguishable from all other objects
Eg: Bank, employee, student

Attributes are properties are properties of an entity.
Eg: Empcode, ename, rolno, name

### Logical data and physical data :

Logical data are the data for the table created by user in primary memory.
Physical data refers to the data stored in the secondary memory.

### Schema and sub-schema :

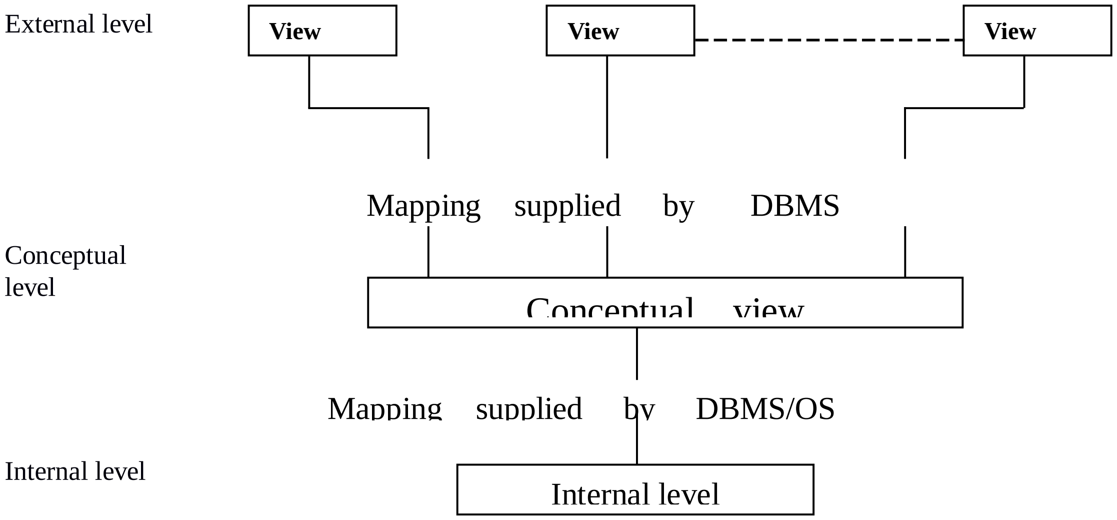
A schema is a logical data base description and is drawn as a chart of the types of data that are used. It gives the names of the entities and attributes and specify the relationships between them.

A database schema includes such information as :

- Characteristics of data items such as entities and attributes .
- Logical structures and relationships among these data items .
- Format for storage representation.
- Integrity parameters such as physical authorization and back up policies.

A *subschema* is derived schema derived from existing schema as per the user requirement. There may be more then one subschema create for a single conceptual schema.

## Three Level Architecture of DBMS :



A database management system that provides three level of data is said to follow three-level architecture .

- External level
- Conceptual level
- Internal level

**External Level :**
The external level is at the highest level of database abstraction . At this level, there will be many views define for different users requirement. A view will describe only a subset of the database. Any number of user views may exist for a given global schema(coneptual schema).

For example, each student has different view of the time table. the view of a student of BTech (CSE) is different from the view of the student of Btech (ECE). Thus this level of abstraction is concerned with different categories of users.
Each external view is described by means of a schema called sub schema.

**Conceptual Level :**
At this level of database abstraction all the database entities and the relationships among them are included. One conceptual view represents the entire database. This conceptual view is defined by the conceptual schema.

The conceptual schema hides the details of physical storage structures and concentrate on describing entities, data types, relationships, user operations and constraints.

It describes all the records and relationships included in the conceptual view. There is only one conceptual schema per database. It includes feature that specify the checks to relation data consistency and integrity.

**Internal level :**
It is the lowest level of abstraction closest to the physical storage method used. It indicates how the data will be stored and describes the data structures and access methods to be used by the database. The internal view is expressed by internal schema.

The following aspects are considered at this level:

1. Storage allocation e.g: B-tree, hashing
2. Access paths eg. specification of primary and secondary keys, indexes etc
3. Miscellaneous eg. Data compression and encryption techniques, optimization of the internal structures.

**<u>Database Users :</u>**

**Naive Users :**
Users who need not be aware of the presence of the database system or any other system supporting their usage are considered naïve users . A user of an automatic teller machine falls on this category.

**Online Users :**

These are users who may communicate with the database directly via an online terminal or indirectly via a user interface and application program. These users are aware of the database system and also know the data manipulation language system.

**Application Programmers :**

Professional programmers who are responsible for developing application programs or user interfaces utilized by the naïve and online user falls into this category.

**Database Administration :**

A person who has central control over the system is called database administrator .

The function of DBA are :

1. Creation and modification of conceptual Schema definition
2. Implementation of storage structure and access method.
3. Schema and physical organization modifications .
4. Granting of authorization for data access.
5. Integrity constraints specification.
6. Execute immediate recovery procedure in case of failures
7. Ensure physical security to database

**Database language :**

1) **Data definition language (DDL) :**

   DDL is used to define database objects .The conceptual schema is specified by a set of definitions expressed by this language. It also gives some details about how to implement this schema in the physical devices used to store the data. This definition includes all the entity sets and their associated attributes and their relationships. The result of DDL statements will be a set of tables that are stored in special file called data dictionary.

2) **Data Manipulation Language (DML) :**

   A DML is a language that enables users to access or manipulate data stored in the database. Data manipulation involves retrieval of data from the database, insertion of new data into the database and deletion of data or modification of existing data.

   There are basically two types of DML:

   - **Procedural**: Which requires a user to specify what data is needed and how to get it.
   - **Non-Procedural:** which requires a user to specify what data is needed with out specifying how to get it.

3) **Data Control Language (DCL):**

   This language enables user to grant authorization and canceling authorization of database objects.

# LECTURE-4: Elements of DBMS

## Elements of DBMS:

### DML Pre-Compiler:

It converts DML statements embedded in an application program to normal procedure calls in the host language. The pre-complier must interact with the query processor in order to generate the appropriate code.

### DDL Compiler:

The DDL compiler converts the data definition statements into a set of tables. These tables contains information concerning the database and are in a form that can be used by other components of the dbms.

### File Manager:

File manager manages the allocation of space on disk storage and the data structure used to represent information stored on disk.

### Database Manager:

A database manager is a program module which provides the interface between the low level data stored in the database and the application programs and queries submitted to the system.

The responsibilities of database manager are:

1. **Interaction with File Manager**: The data is stored on the disk using the file system which is provided by operating system. The database manager translate the different DML statements into low-level file system commands so the database manager is responsible for the actual storing, retrieving and updating of data in the database.

2. **Integrity Enforcement:** The data values stored in the database must satisfy certain constraints (eg: the age of a person can't be less then zero). These constraints are specified by DBA. Data manager checks the constraints and if it satisfies then it stores the data in the database.

3. **Security Enforcement:** Data manager checks the security measures for database from unauthorized users.

4. **Backup and Recovery:** Database manager detects the failures occur due to different causes (like disk failure, power failure, deadlock, software error) and restores the database to original state of the database.

5. **Concurrency Control:** When several users access the same database file simultaneously, there may be possibilities of data inconsistency. It is responsible of database manager to control the problems occur for concurrent transactions.

### Query Processor:

The query processor used to interpret to online user’s query and convert it into an efficient series of operations in a form capable of being sent to the data manager for execution. The query processor uses the data dictionary to find the details of data file and using this information it create query plan/access plan to execute the query.

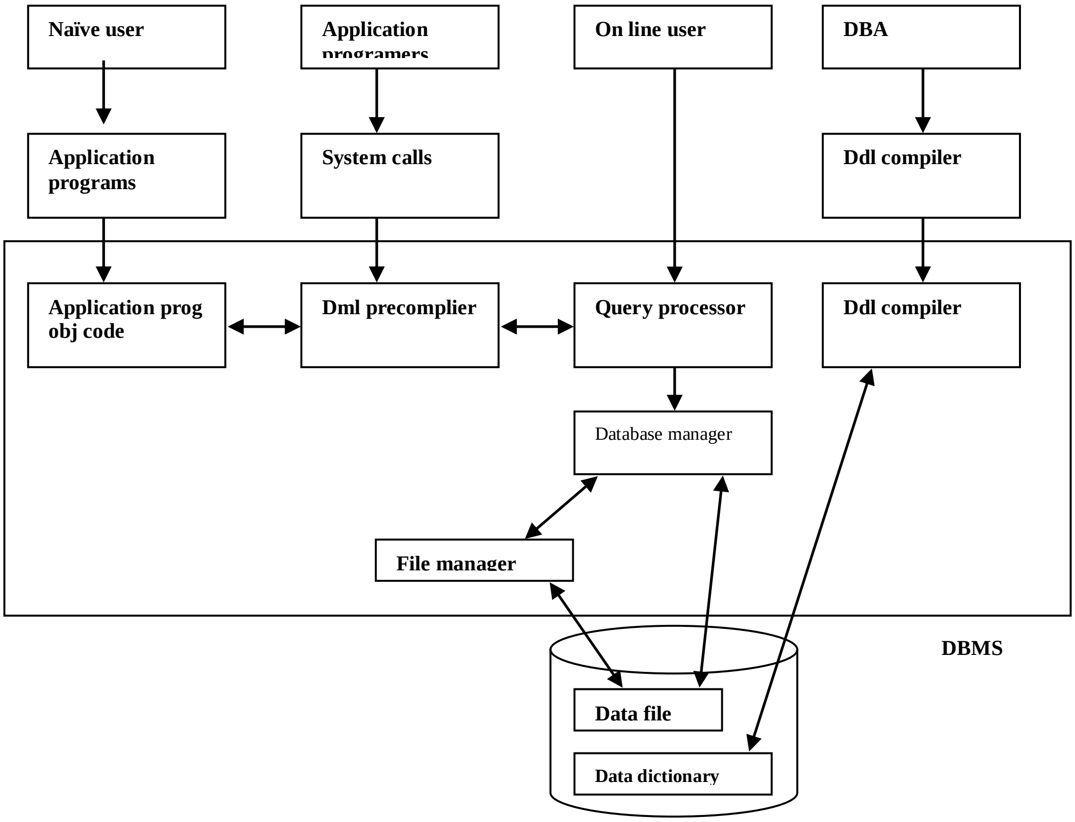
**Data Dictionary:**

Data dictionary is the table which contains the information about database objects. It contains information like

1. external, conceptual and internal database description
2. description of entities, attributes as well as meaning of data elements
3. synonyms, authorization and security codes
4. database authorization

The data stored in the data dictionary is called *meta data.*

## DBMS STRUCTURE:



**Que:** List four significant differences between a File-Processing System and a DBMS.

**Ans:** Some major differences between a database management system and a file-processing system are:

- Both systems contain a collection of data and a set of programs which access that data. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.

- A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, where as data written by one program in a file-processing system may not be readable by another program.

- A database management system is designed to allow flexible access to data (i.e., queries), whereas a file-processing system is designed to allow predetermined access to data (i.e., compiled programs).

- A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file.

**Que:** Explain the difference between physical and logical data independence.

**Ans:**

- Physical data independence is the ability to modify the physical scheme without making it necessary to rewrite application programs. Such modifications include changing from unblocked to blocked record storage, or from sequential to random access files.

- Logical data independence is the ability to modify the conceptual scheme without making it necessary to rewrite application programs. Such a modification might be adding a field to a record; an application program's view hides this change from the program.

**Que:** List five responsibilities of a database management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.

**Ans:** A general purpose database manager (DBM) has five responsibilities:

a. interaction with the file manager.
b. integrity enforcement.
c. security enforcement.
d. backup and recovery.
e. concurrency control.

If these responsibilities were not met by a given DBM (and the text points out that sometimes a responsibility is omitted by design, such as concurrency control on a single-user DBM for a micro computer) the following problems can occur, respectively:

a. No DBM can do without this, if there is no file manager interaction then nothing stored in the files can be retrieved.

b. Consistency constraints may not be satisfied, when account balances could go below the minimum allowed, employees could earn too much overtime (e.g.,hours > 80) or, airline pilots may fly more hours than allowed by law.

c. Unauthorized users may access the database, or users authorized to access part of the database may be able to access parts of the database for which they lack authority. For example, a high school student could get access to national defense secret codes, or employees could find out what their supervisors earn.

d. Data could be lost permanently, rather than at least being available in a consistent state that existed prior to a failure.

e. Consistency constraints may be violated when intgrity constraints failed in a transaction. For example, incorrect bank balances might be reflected due to simultaneous withdrawals and deposits, and so on.

**Que.** What are five main functions of a database administrator?

**Ans:** Five main functions of a database administrator are:

- To create the scheme definition
- To define the storage structure and access methods
- To modify the scheme and/or physical organization when necessary
- To grant authorization for data access
- To specify integrity constraints

**Que:** List six major steps that you would take in setting up a database for a particular enterprise.

**Ans:** Six major steps in setting up a database for a particular enterprise are:

- Define the high level requirements of the enterprise (this step generates a document known as the system requirements specification.)
- Define a model containing all appropriate types of data and data relationships.
- Define the integrity constraints on the data.
- Define the physical level.
- For each known problem to be solved on a regular basis (e.g., tasks to be carried out by clerks or Web users) define a user interface to carry out the task, and write the necessary application programs to implement the user interface.
- Create/initialize the database.

**EXERCISE:**

1. What is database management system?
2. What are the disadvantage of file processing system?
3. State advantage and disadvantage of database management system.
4. What are different types of database users?
5. What is data dictionary and what are its contents?
6. What are the functions of DBA?
7. What are the different database languages? Explain with example.
8. Explain the three layer architecture of DBMS.
9. Differentiate between physical data independence and logical data independence.
10. Explain the functions of database manager.
11. Explain meta data.

# LECTURE-5: ER-MODEL

**Data Model:**

The data model describes the structure of a database. It is a collection of conceptual tools for describing data, data relationships and consistency constraints and various types of data models such as

1. Object based logical model
2. Record based logical model
3. Physical model

Types of data model:

1. Object based logical model
   a. ER-model
   b. Functional model
   c. Object oriented model
   d. Semantic model
2. Record based logical model
   a. Hierarchical database model
   b. Network model
   c. Relational model
3. Physical model

## Entity Relationship Model (ER Model)

The entity-relationship data model perceives the real world as consisting of basic objects, called entities and relationships among these objects. It was developed to facilitate database design by allowing specification of an enterprise schema which represents the overall logical structure of a data base.

**Main Features of ER-MODEL:**

- Entity relationship model is a high level conceptual model
- It allows us to describe the data involved in a real world enterprise in terms of objects and their relationships.
- It is widely used to develop an initial design of a database
- It provides a set of useful concepts that make it convenient for a developer to move from a basic set of information to a detailed and description of information that can be easily implemented in a database system
- It describes data as a collection of entities, relationships and attributes.

**Basic Concepts:**

The E-R data model employs three basic notions : entity sets, relationship sets and attributes.

**Entity Sets:**

An entity is a “thing” or “object” in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set properties and the values for some set of properties may uniquely identify an entity. BOOK is entity and its properties (called as attributes) bookcode, booktitle, price etc.

An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank.

**Attributes:**
An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.

**Customer** is an entity and its attributes are **customerid, custmername, custaddress** etc.

An attribute as used in the E-R model, can be characterized by the following attribute types.

a) **Simple and Composite Attribute:**
Simple attributes are the attributes which can't be divided into sub parts, e.g. customerid, empno
Composite attributes are the attributes which can be divided into subparts, e.g. name consisting of first name, middle name, last name and address consisting of city, pincode, state.

b) **Single-Valued and Multi-Valued Attribute:**
The attribute having unique value is single –valued attribute, e.g. empno, customerid, regdno etc.
The attribute having more than one value is multi-valued attribute, eg: phone-no, dependent name, vehicle.

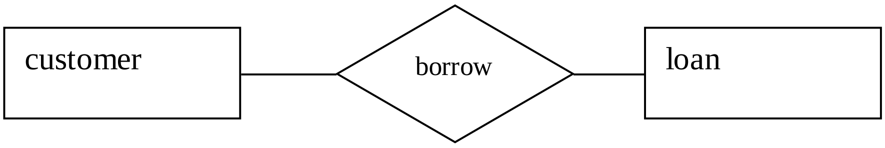
c) **Derived Attribute:**
The values for this type of attribute can be derived from the values of existing attributes, e.g. age which can be derived from currentdate – birthdate and experience_in_year can be calculated as currentdate-joindate.

d) **NULL Valued Attribute:**
The attribute value which is not known to user is called NULL valued attribute.

**Relationship Sets:**
A relationship is an association among several entities. A relationship set is a set of relationships of the same type. Formally, it is a mathematical relation on n>=2 entity sets. If $E_1$, $E_2 \ldots E_n$ are entity sets, then a relation ship set R is a subset of

$\{(e_1,e_2,\ldots e_n) \mid e_1 Є E_1, e_2 Є E_2.., e_n Є E_n\}$

where $(e_1,e_2,\ldots e_n)$ is a relation ship.



Consider the two entity sets customer and loan. We define the relationship set borrow to denote the association between customers and the bank loans that the customers have.

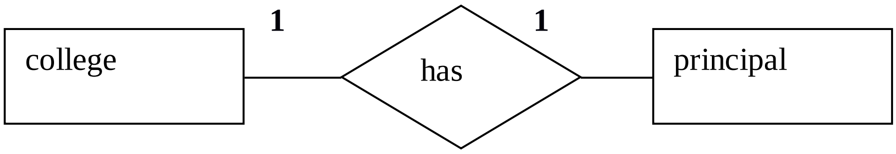
**Mapping Cardinalities:**
Mapping cardinalities or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets. For a binary relationship set R between entity sets A and B, the mapping

cardinalities must be one of the following:

**1. One to One:**

An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

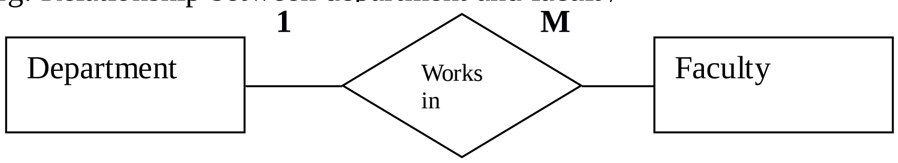
Eg: relationship between college and principal



**2. One to Many:**

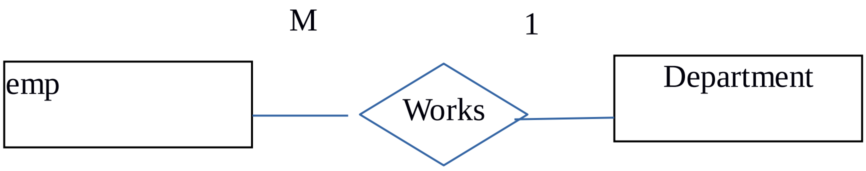
An entity in A is associated with any number of entities in B. An entity in B is associated with at the most one entity in A.

Eg: Relationship between department and faculty



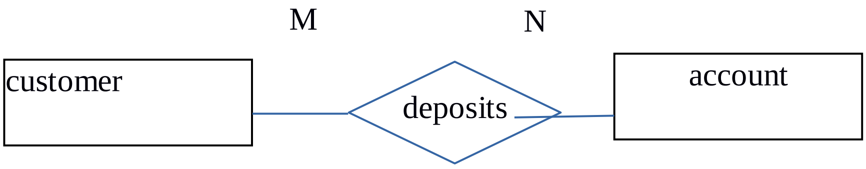
**3. Many to One:**

An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A.



**4. Many to Many:**

Entities in A and B are associated with any number of entities from each other.



**More about Entities and Relationship:**

**Recursive Relationships:**

When the same entity type participates more than once in a relationship type in different roles, the relationship types are called recursive relationships.

**Participation Constraints:**

The participation constraints specify whether the existence of any entity depends on its being related to another entity via the relationship. There are two types of participation constraints

**a) Total :** When all the entities from an entity set participate in a relationship type, is called total participation. For example, the participation of the entity set student on the relationship set must ‘opts’ is said to be total because every student enrolled must opt for a course.

**b) Partial:** When it is not necessary for all the entities from an entity set to particapte in a relationship type, it is called partial participation. For example, the participation of the entity set student in ‘represents’ is partial, since not every student in a class is a class representative.

**Weak Entity:**

Entity types that do not contain any key attribute, and hence can not be identified independently are called weak entity types. A weak entity can be identified by uniquely only by considering some of its attributes in conjunction with the primary key attribute of another entity, which is called the identifying owner entity.

Generally a partial key is attached to a weak entity type that is used for unique identification of weak entities related to a particular owner type. The following restrictions must hold:

- The owner entity set and the weak entity set must participate in one to may relationship set. This relationship set is called the identifying relationship set of the weak entity set.
- The weak entity set must have total participation in the identifying relationship.

**Example:**

Consider the entity type Dependent related to Employee entity, which is used to keep track of the dependents of each employee. The attributes of Dependents are: name, birthdate, sex and relationship. Each employee entity set is said to its own the dependent entities that are related to it. However, not that the ‘Dependent’ entity does not exist of its own, it is dependent on the Employee entity.

**Keys:**

**Super Key:**

A super key is a set of one or more attributes that taken collectively, allow us to identify uniquely an entity in the entity set. For example , customer-id, (cname, customer-id), (cname, telno)

**Candidate Key:**

In a relation R, a candidate key for R is a subset of the set of attributes of R, which have the following properties:

1. *Uniqueness: N*o two distinct tuples in R have the same values for the candidate key
2. *Irreducible:* No proper subset of the candidate key has the uniqueness property that is the candidate key. Eg: (cname,telno)

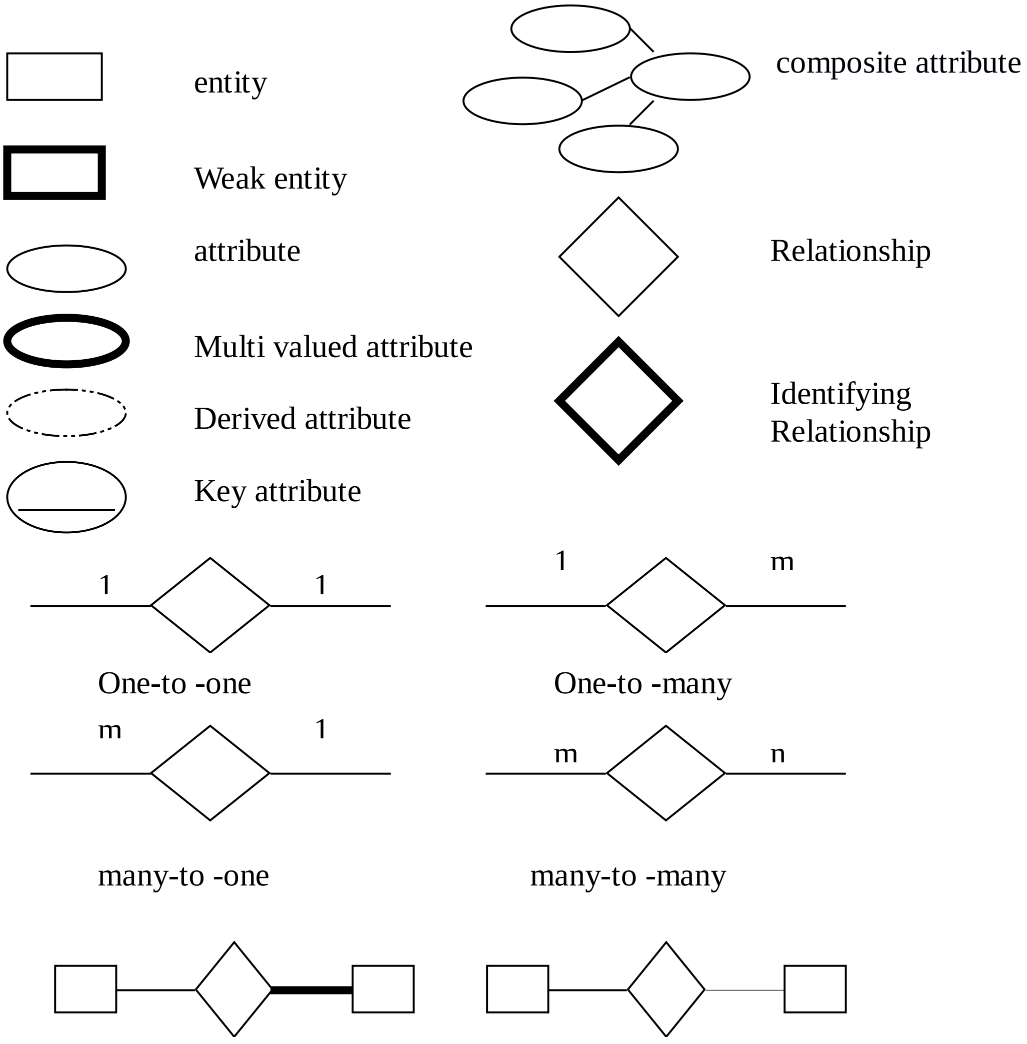
**Primary Key:**

The primary key is the candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. The remaining candidate keys if any, are called *Alternate Key*.

# LECTURE-6: ER-DIAGRAM:

The overall logical structure of a database using ER-model graphically with the help of an ER-diagram.

**Symbols use ER- diagram:**



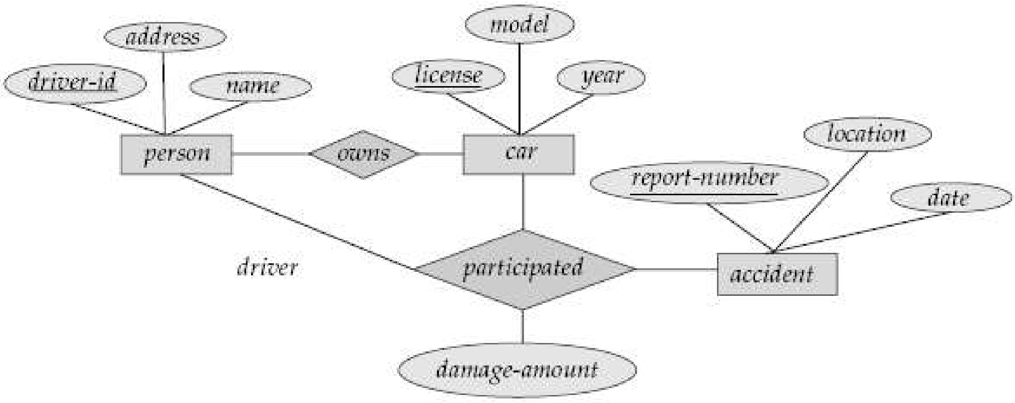


Figure 2.1 E-R diagram for a Car-insurance company.

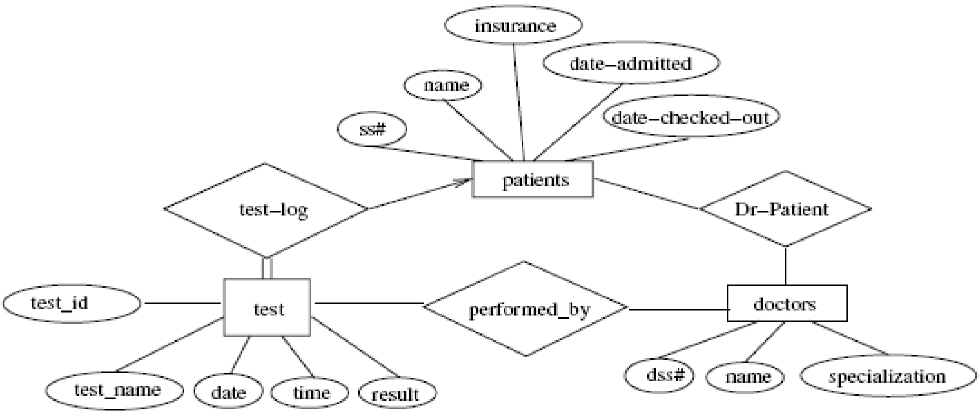


Figure 2.2 E-R diagram for a hospital.

A Univeristy registrar's office maintains data about the following entities:

(a) Course, includeing number,title,credits,syllabus and prereqisites

(b) course offering,including course number,year,semester,section number,instructor timings, and class room

(c) Students including student-id,name and program

(d) Instructors, including identification number,name,department and title

further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriate modeled.

Construct an E-R diagram for the registrar's office. Document all assumptions that you may make about the mapping constratints

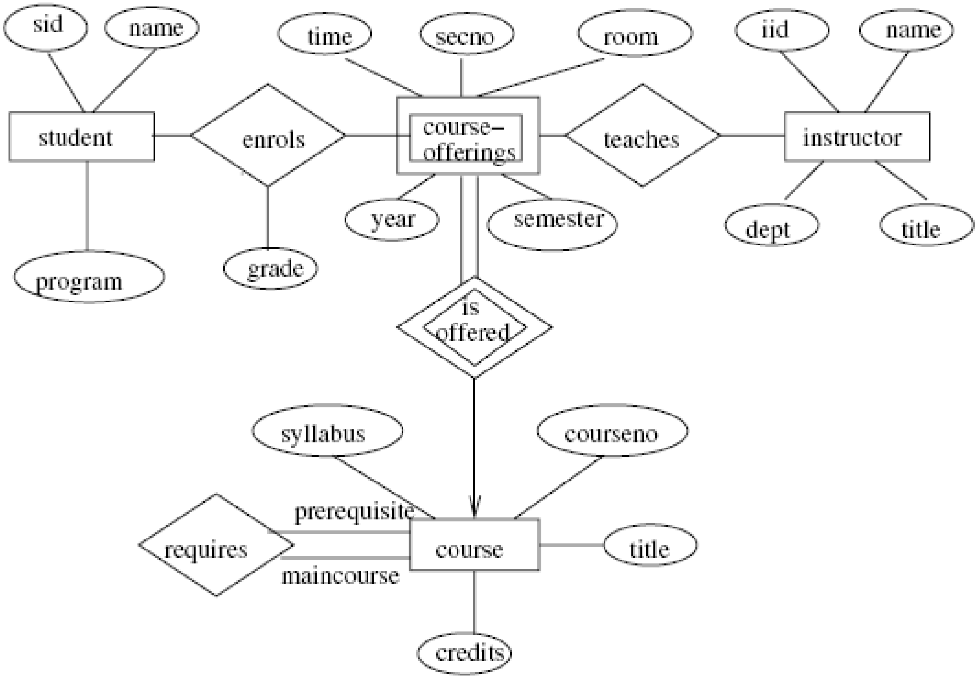


Figure 2.3 E-R diagram for a university.

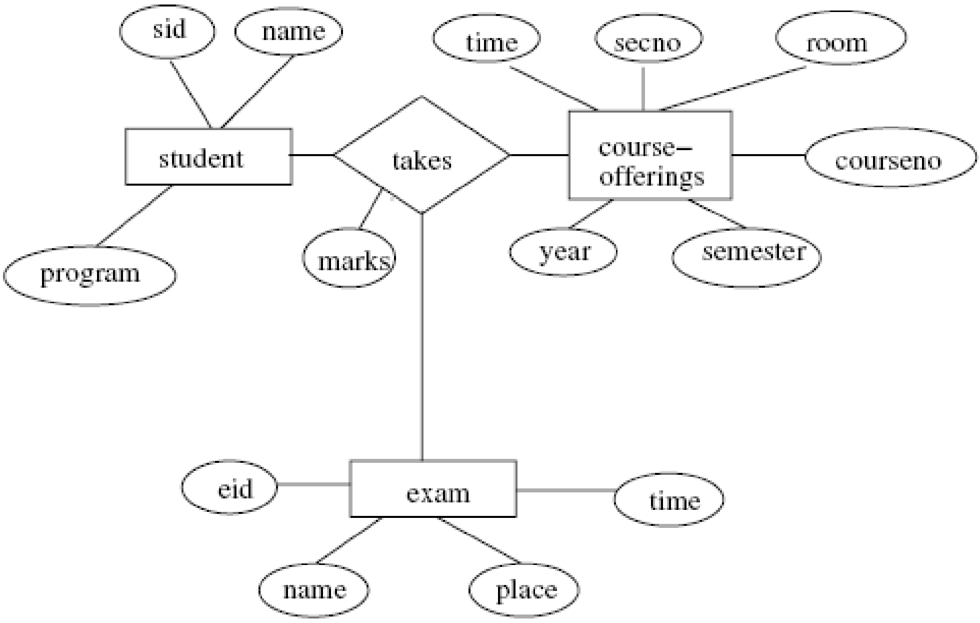


Figure 2.4 E-R diagram for marks database.

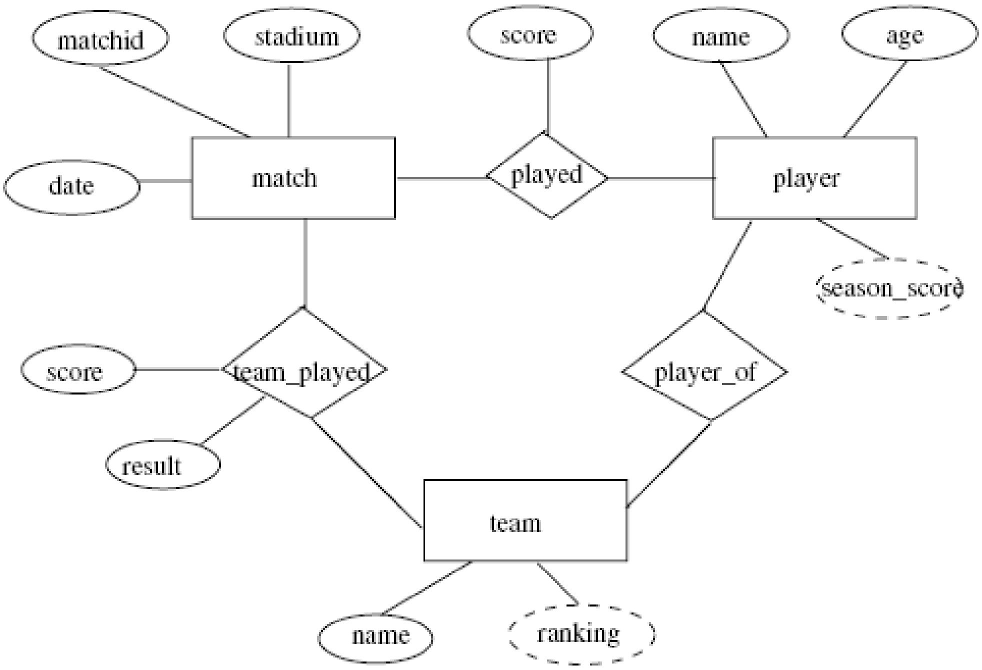


Figure 2.7 E-R diagram for all teams statistics.

Cosidet a university database for the scheduling of class rooms for final exams. This database could be modeled as the single entity set exam, with attributes course-name,section-number,room-number and time, Alternatively, one or more additional entity sets would be defined, along with relationship sets to replae some of the attributes of the exam entity set, as

- course with attributes name,department and c-number
- section with attributes s-number and enrollment and dependent as a weak entity set on course
- room with attributes r-number,capacity and building

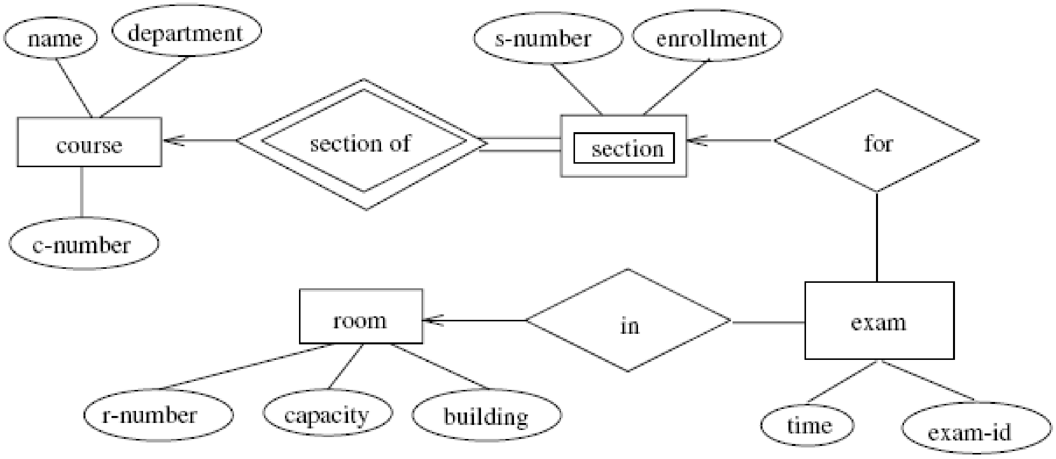
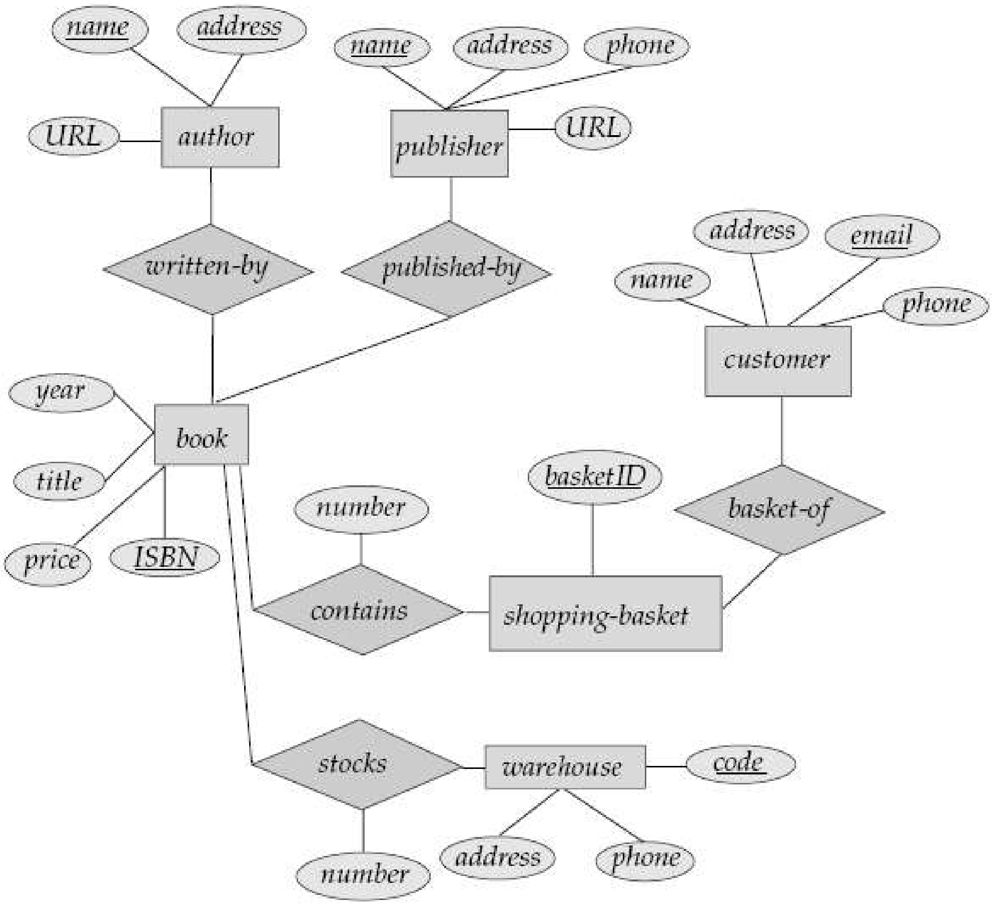


Figure 2.12 E-R diagram for exam scheduling.



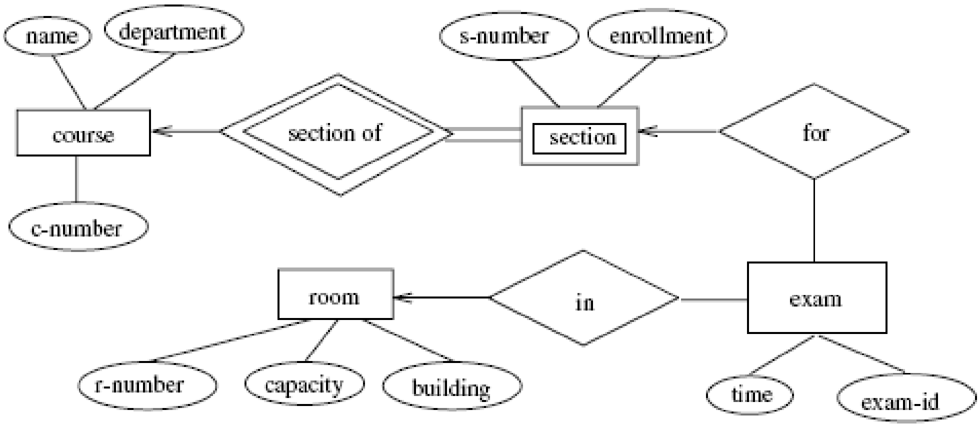


Figure 2.12 E-R diagram for exam scheduling.

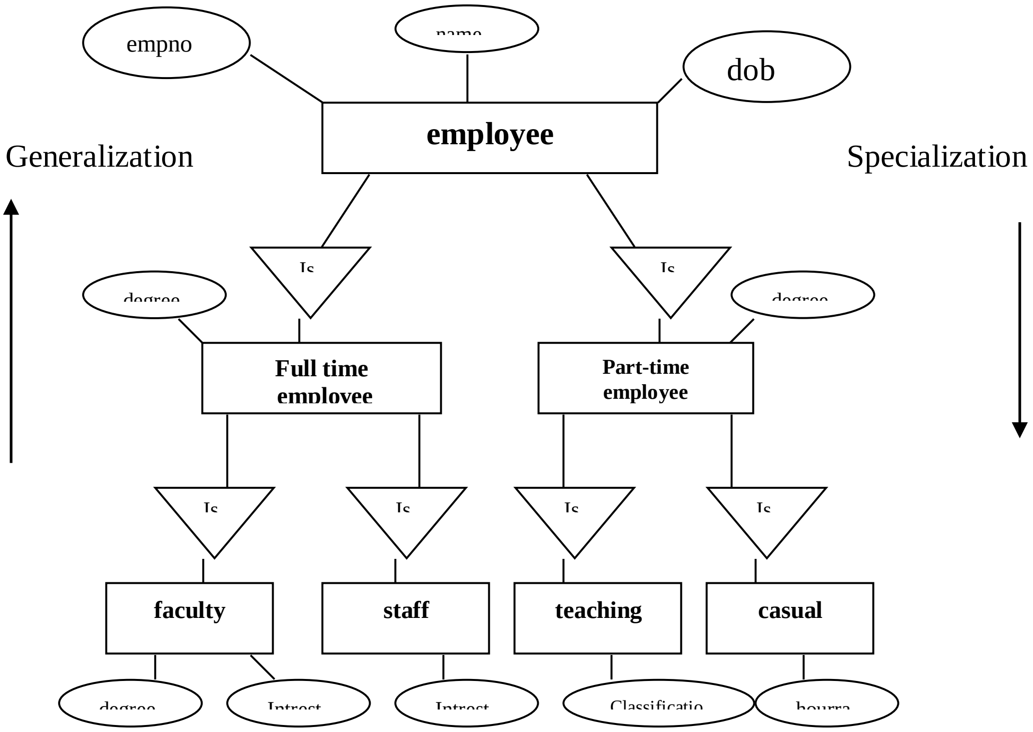
## LECTURE-7: Advanced ER-Diagram:

Abstraction is the simplification mechanism used to hide superfluous details of a set of objects. It allows one to concentrate on the properties that are of interest to the application. There are two main abstraction mechanism used to model information:

### Generalization and specialization:

*Generalization* is the abstracting process of viewing set of objects as a single general class by concentrating on the general characteristics of the constituent sets while suppressing or ignoring their differences. It is the union of a number of lower-level entity types for the purpose of producing a higher-level entity type. For instance, student is a generalization of graduate or undergraduate, full-time or part-time students. Similarly, employee is generalization of the classes of objects cook, waiter, and cashier. Generalization is an IS_A relationship; therefore, manager IS_AN employee, cook IS_AN employee, waiter IS_AN employee, and so forth.

*Specialization* is the abstracting process of introducing new characteristics to an existing class of objects to create one or more new classes of objects. This involves taking a higher-level, and using additional characteristics, generating lower-level entities. The lower-level entities also inherits the, characteristics of the higher-level entity. In applying the characteristics size to car we can create a full-size, mid-size, compact or subcompact car. Specialization may be seen as the reverse process of generalization addition specific properties are introduced at a lower level in a hierarchy of objects.



| | |
|---|---|
| EMPLOYEE(**empno**,name,dob) | Faculty(**empno**,degree,intrest) |
| FULL_TIME_EMPLOYEE(**empno**,salary) | Staff(**empno**,hour-rate) |
| PART_TIME_EMPLOYEE(**empno**,type) | Teaching (**empno**,stipend) |

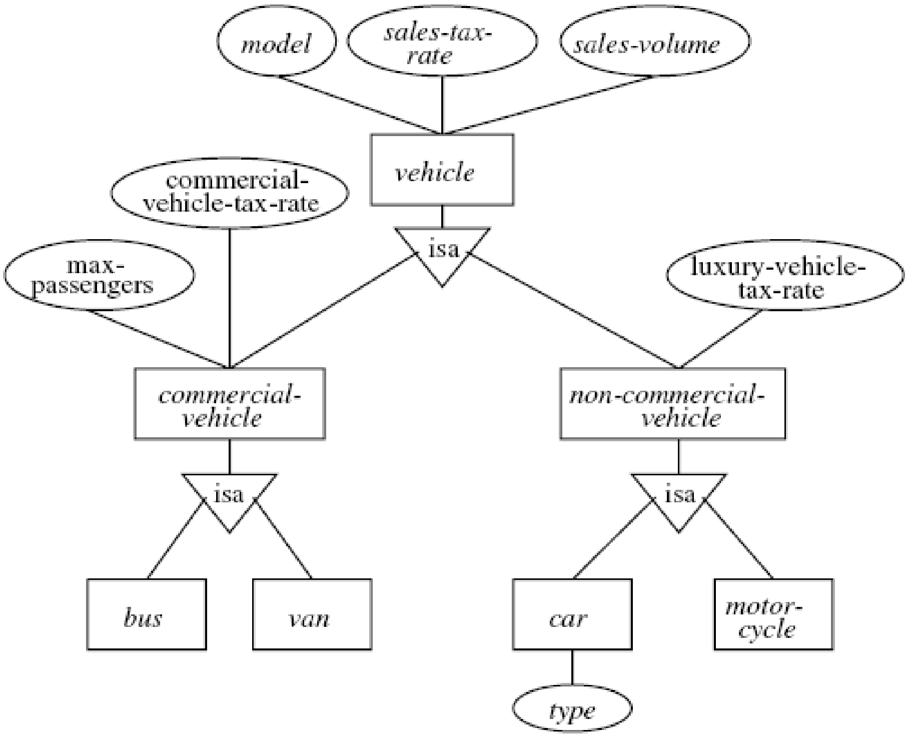
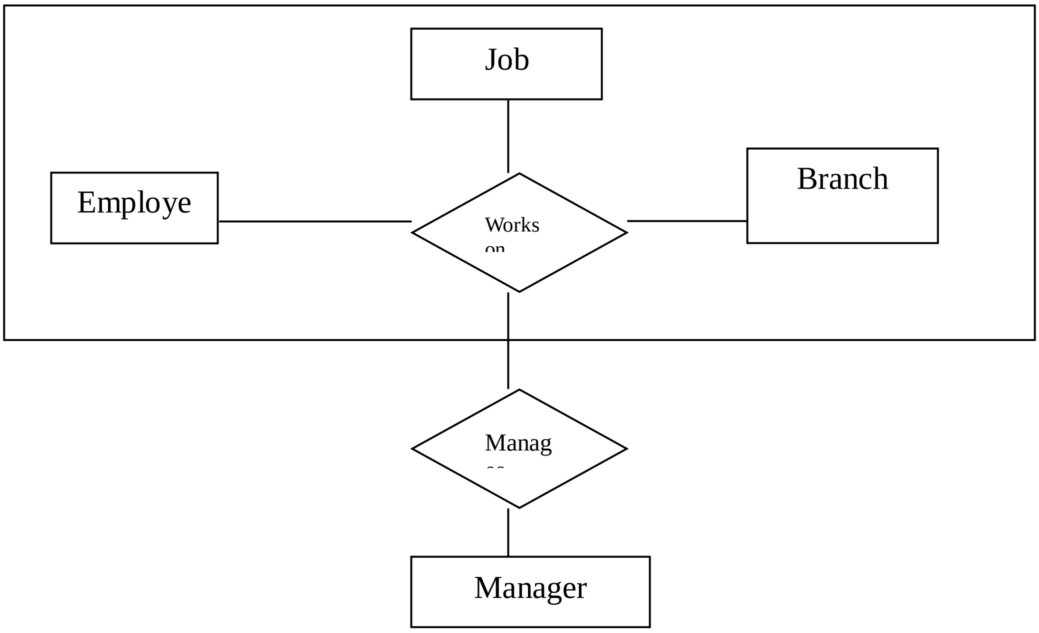


Figure 2.19 E-R diagram of motor-vehicle sales company.

**Aggregation:**

Aggregation is the process of compiling information on an object, there by abstracting a higher level object. The entity person is derived by aggregating the characteristics of name, address, ssn. Another form of the aggregation is abstracting a relationship objects and viewing the relationship as an object.



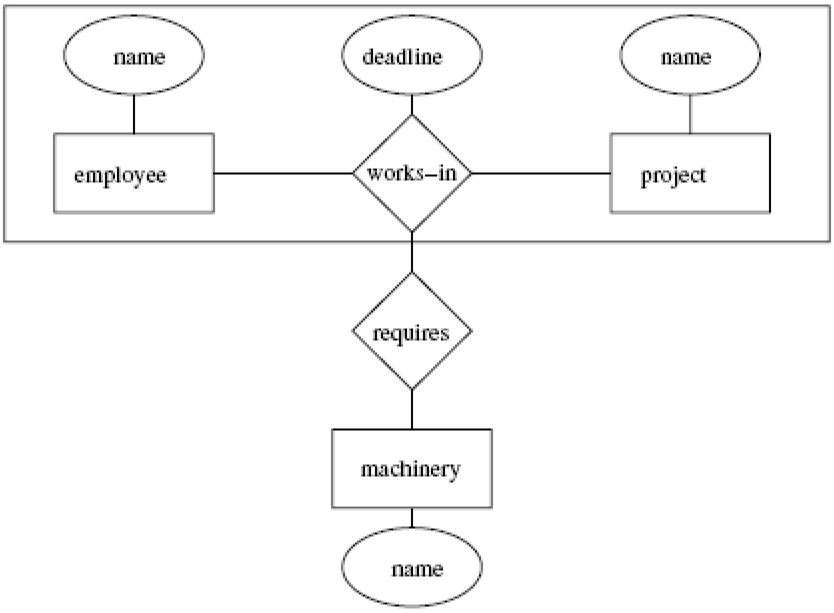


Figure 2.8 E-R diagram Example 1 of aggregation.

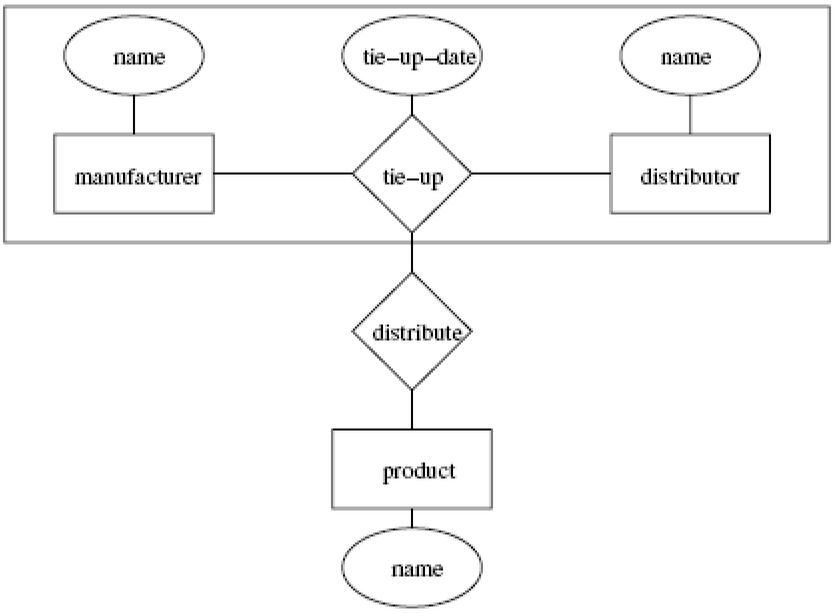
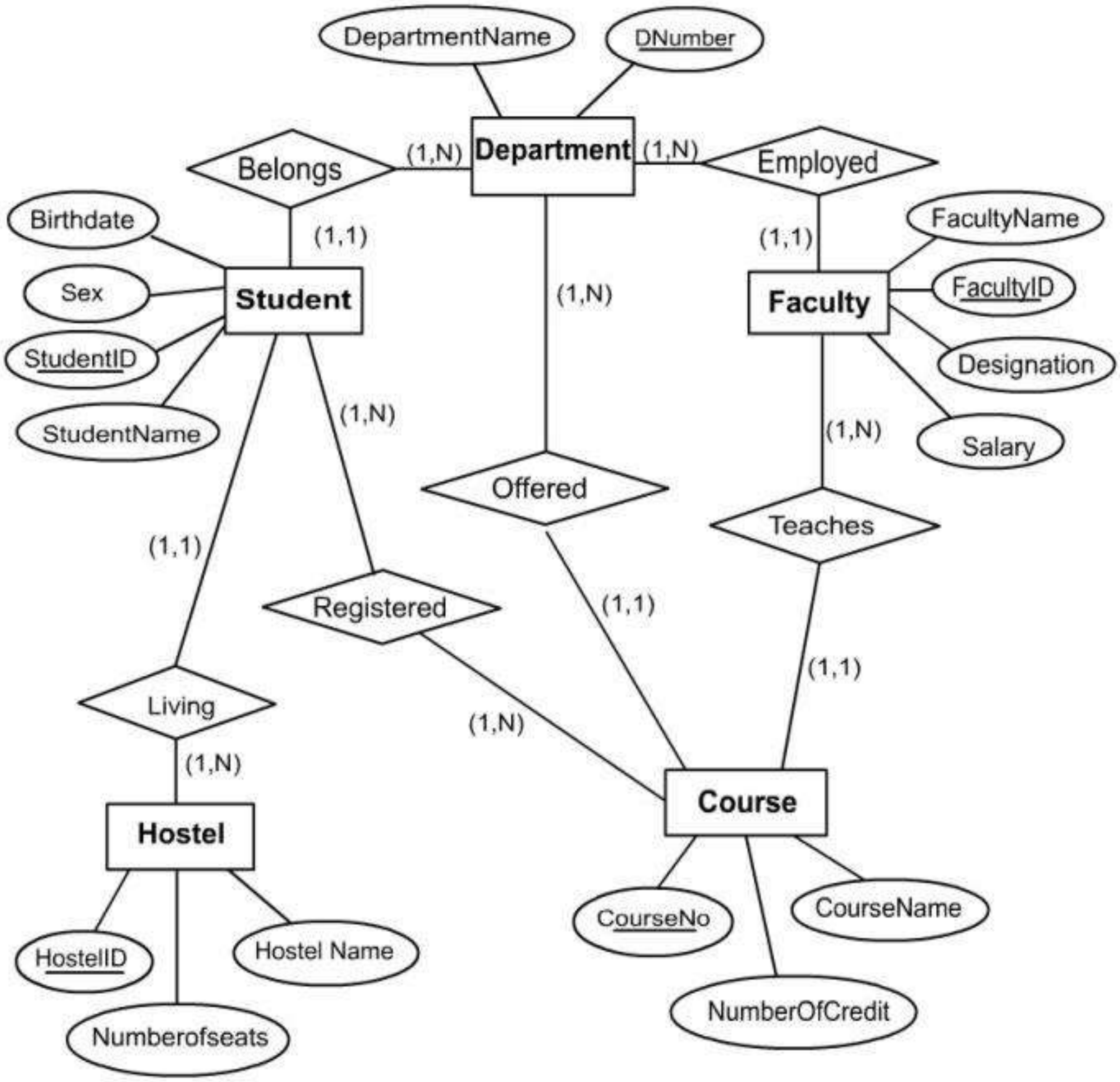


Figure 2.9 E-R diagram Example 2 of aggregation.

## ER- Diagram For College Database



# LECTURE-8: Conversion of ER-Diagram to Relational Database

**Conversion of Entity Sets:**

1. For each strong entity type E in the ER diagram, we create a relation R containing all the single attributes of E. The primary key of the relation R will be one of the key attribute of R.

   **STUDENT**(rollno (primary key),name, address)
   **FACULTY**(id(primary key),name ,address, salary)
   **COURSE**(course-id,(primary key),course_name,duration)
   **DEPARTMENT**(dno(primary key),dname)

2. For each weak entity type W in the ER diagram, we create another relation R that contains all simple attributes of W. If E is an owner entity of W then key attribute of E is also include In R. This key attribute of R is set as a foreign key attribute of R. Now the combination of primary key attribute of owner entity type and partial key of the weak entity type will form the key of the weak entity type

   GUARDIAN((rollno,name) (primary key),address,relationship)

## Conversion of Relationship Sets:

**Binary Relationships:**

- **One-to-One Relationship:**
  For each 1:1 relationship type R in the ER-diagram involving two entities E1 and E2 we choose one of entities(say E1) preferably with total participation and add primary key attribute of another E as a foreign key attribute in the table of entity(E1). We will also include all the simple attributes of relationship type R in E1 if any, For example, the department relationship has been extended tp include head-id and attribute of the relationship.
  DEPARTMENT(D_NO,D_NAME,HEAD_ID,DATE_FROM)

- **One-to-Many Relationship:**
  For each 1:N relationship type R involving two entities E1 and E2, we identify the entity type (say E1) at the N-side of the relationship type R and include primary key of the entity on the other side of the relation (say E2) as a foreign key attribute in the table of E1. We include all simple attribute (or simple components of a composite attribute of R (if any) in the table E1)

  For example:
  The works in relationship between the DEPARTMENT and FACULTY. For this relationship choose the entity at N side, i.e, FACULTY and add primary key attribute of another entity DEPARTMENT i.e., DNO as a foreign key attribute in FACULTY.

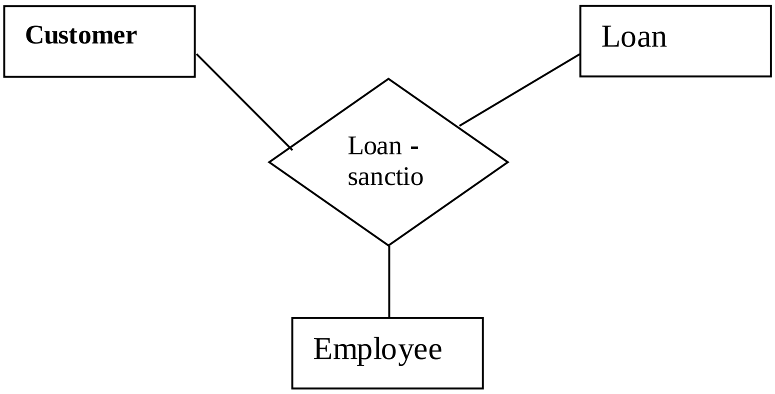
  FACULTY(CONTAINS WORKS_IN RELATIOSHIP)
  (ID, NAME, ADDRESS, BASIC_SAL, DNO)

- **Many-to-Many Relationship:**
  For each M:N relationship type R, we create a new table (say S) to represent R, we also include the primary key attributes of both the participating entity types as a foreign key attribute in S. Any simple attributes of the M:N relationship type (or simple components as a composite attribute) is also included as attributes of S.

For example:
The M:N relationship taught-by between entities COURSE and FACULTY should be represented as a new table. The structure of the table will include primary key of COURSE and primary key of FACULTY entities.

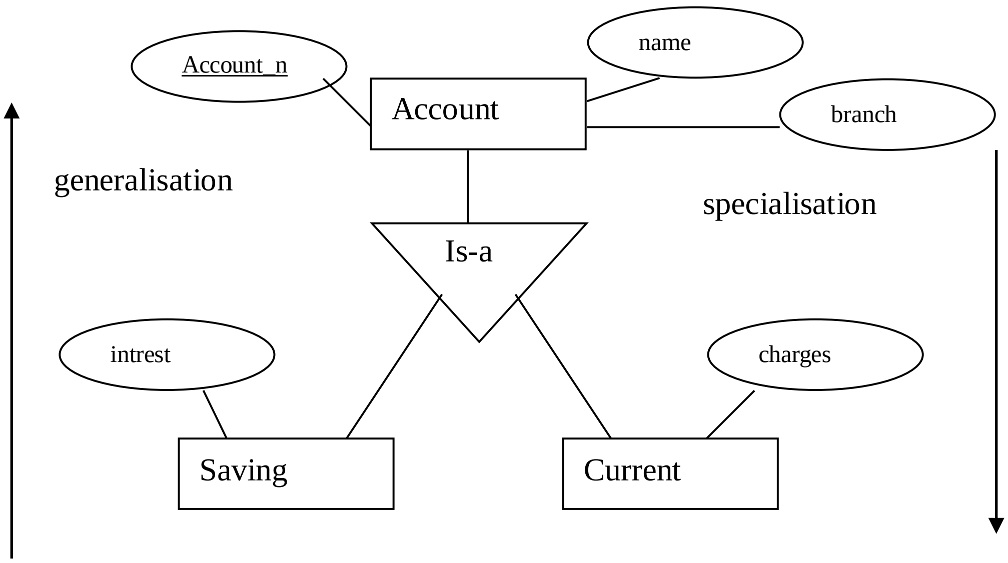
TAUGHT-BY (ID (primary key of FACULTY table), course-id (primary key of COURSE table)

- **N-ary Relationship:**
  For each N-ary relationship type R where n>2, we create a new table S to represent R, We include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. We also include any simple attributes of the N-ary relationship type (or simple components of complete attribute) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types.



LOAN-SANCTION (cusomer-id, loanno, empno, sancdate, loan_amount)

- **Multi-Valued Attributes:**
  For each multivalued attribute 'A', we create a new relation R that includes an attribute corresponding to plus the primary key attributes k of the relation that represents the entity type or relationship that has as an attribute. The primary key of R is then combination of A and k.
  For example, if a STUDENT entity has rollno, name and phone number where phone number is a multivalued attribute then we will create table PHONE (rollno, phoneno) where primary key is the combination. In the STUDENT table we need not have phone number, instead if can be simply (rollno, name) only.
  PHONE(rollno, phoneno)



- **Converting Generalisation /Specification Hierarchy to Tables:**
  A simple rule for conversion may be to decompose all the specialized entities into table in case they are disjoint, for example, for the figure we can create the three tables as:
  Account (account_no, name, branch, balance)
  Saving_Account (account-no, intrest)
  Current_Account (account-no, charges)

# LECTURE-9: Record Based Logical Model

**Hierarchical Model:**

- A hierarchical database consists of a collection of *records* which are connected to one another through *links.*
- A record is a collection of fields, each of which contains only one data value.
- A link is an association between precisely two records.
- The hierarchical model differs from the network model in that the records are organized as collections of trees rather than as arbitrary graphs.

**Tree-Structure Diagrams:**

- The schema for a hierarchical database consists of
  - *boxes,* which correspond to record types
  - *lines,* which correspond to links
- Record types are organized in the form of a *rooted tree.*
  - No cycles in the underlying graph.
  - Relationships formed in the graph must be such that only one-to-many or one-to-one relationships exist between a parent and a child.

Database schema is represented as a collection of tree-structure diagrams.

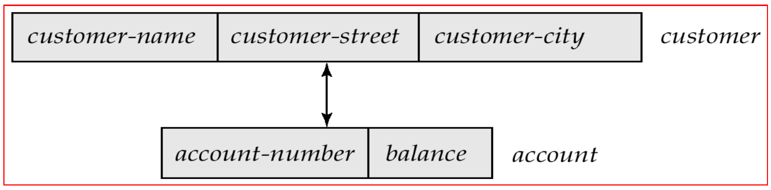
- *single* instance of a database tree
- The root of this tree is a dummy node
- The children of that node are actual instances of the appropriate record type

When transforming E-R diagrams to corresponding tree-structure diagrams, we must ensure that the resulting diagrams are in the form of rooted trees.

**Single Relationships:**

- Example of E-R diagram with two entity sets, *customer* and *account,* related through a binary, one-to-many relationship *depositor.*
- Corresponding tree-structure diagram has
  - the record type *customer* with three fields: *customer-name*, *customer-street*, and *customer-city.*
  - the record type *account* with two fields: *account-number* and *balance*
  - the link *depositor,* with an arrow pointing to *customer*

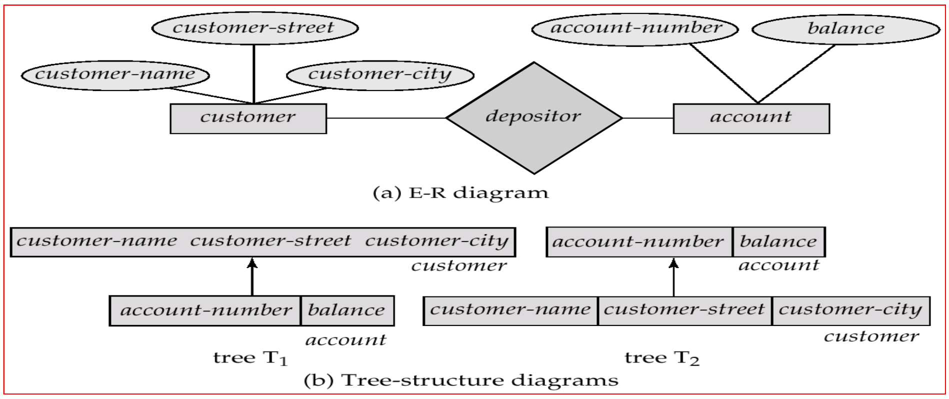
- If the relationship *depositor* is one to one, then the link *depositor* has two arrows.



- Only one-to-many and one-to-one relationships can be directly represented in the hierarchical mode.

**Transforming Many-To-Many Relationships:**

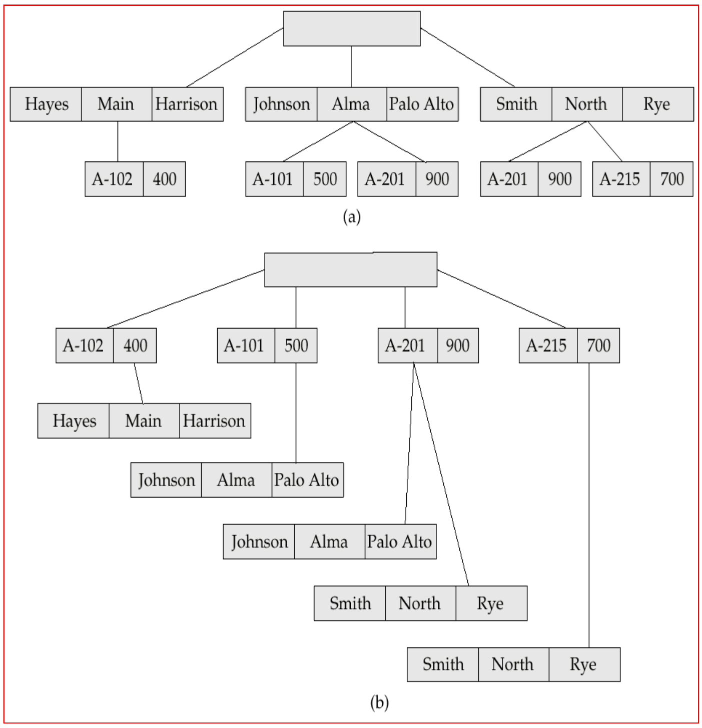
- Must consider the type of queries expected and the degree to which the database schema fits the given E-R diagram.
- In all versions of this transformation, the underlying database tree (or trees) will have replicated records.



(a) E-R diagram

(b) Tree-structure diagrams

- Create two tree-structure diagrams, *T*1, with the root *customer,* and *T*2, with the root *account.*
- In *T*1, create *depositor,* a many-to-one link from *account* to *customer.*
- In *T*2, create *account-customer,* a many-to-one link from *customer* to *account.*



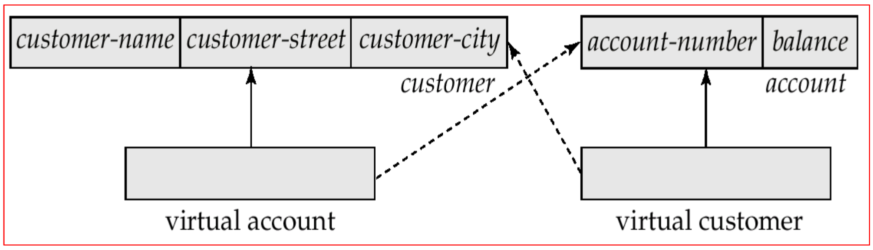
(a)

(b)

**Virtual Records:**

- For many-to-many relationships, record replication is necessary to preserve the tree-structure organization of the database.
- Data inconsistency may result when updating takes place
- Waste of space is unavoidable

- *Virtual record* — contains no data value, only a logical pointer to a particular physical record.
- When a record is to be replicated in several database trees, a single copy of that record is kept in one of the trees and all other records are replaced with a virtual record.
- Let *R* be a record type that is replicated in *T*1, *T*2, . . ., *Tn*. Create a new virtual record type *virtual-R* and replace *R* in each of the *n* – 1 trees with a record of type *virtual-R.*
- Eliminate data replication in the following diagram ; create *virtual-customer* and *virtual-account.*
- Replace *account* with *virtual-account* in the first tree, and replace *customer* with *virtual-customer* in the second tree.
- Add a dashed line from *virtual-customer* to *customer*, and from *virtual-account* to *account*, to specify the association between a virtual record and its corresponding physical record.



**Network Model:**

- Data are represented by collections of *records*.
    - similar to an entity in the E-R model
    - Records and their fields are represented as *record type*

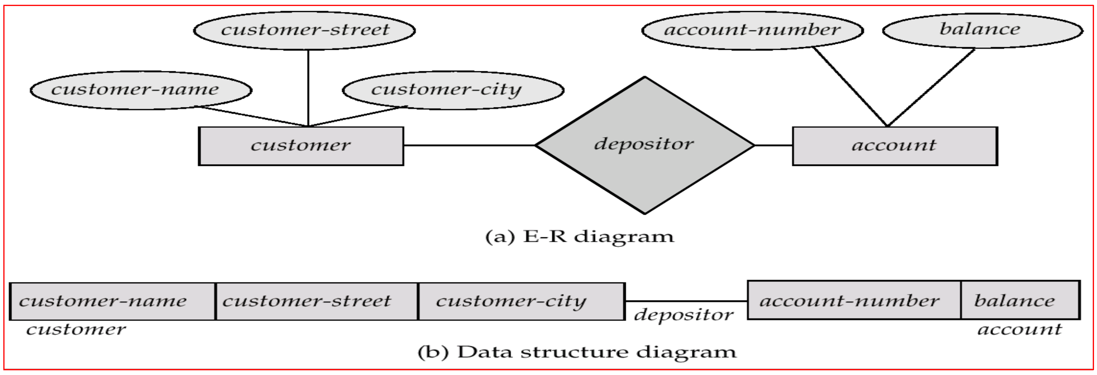
| | | | |
|---|---|---|---|
| type | *customer* = record | type | *account* = record type |
| | *customer-name:* string; | | *account-number:* integer; |
| | *customer-street:* string; | | *balance:* integer; |
| | *customer-city:* string; | | |
| end | | end | |

- Relationships among data are represented by *links*
    - similar to a restricted (binary) form of an E-R relationship
    - restrictions on links depend on whether the relationship is many-to-many, many-to-one, or one-to-one.

**Data-Structure Diagrams:**

- Schema representing the design of a network database.
- A data-structure diagram consists of two basic components:
    - Boxes, which correspond to record types.
    - Lines, which correspond to links.
- Specifies the overall logical structure of the database.

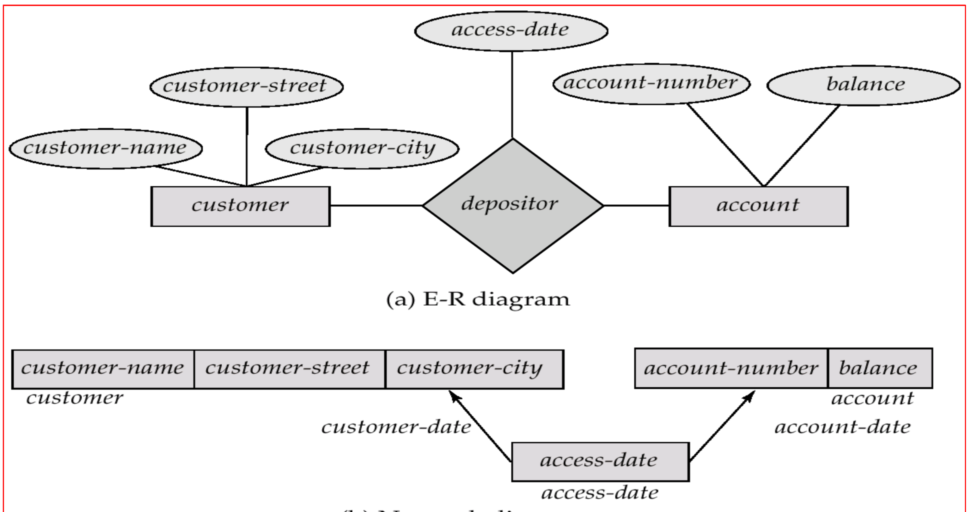
For every E-R diagram, there is a corresponding data-structure diagram.



(a) E-R diagram

(b) Data structure diagram

Since a link cannot contain any data value, represent an E-R relationship with attributes with a new record type and links**.**

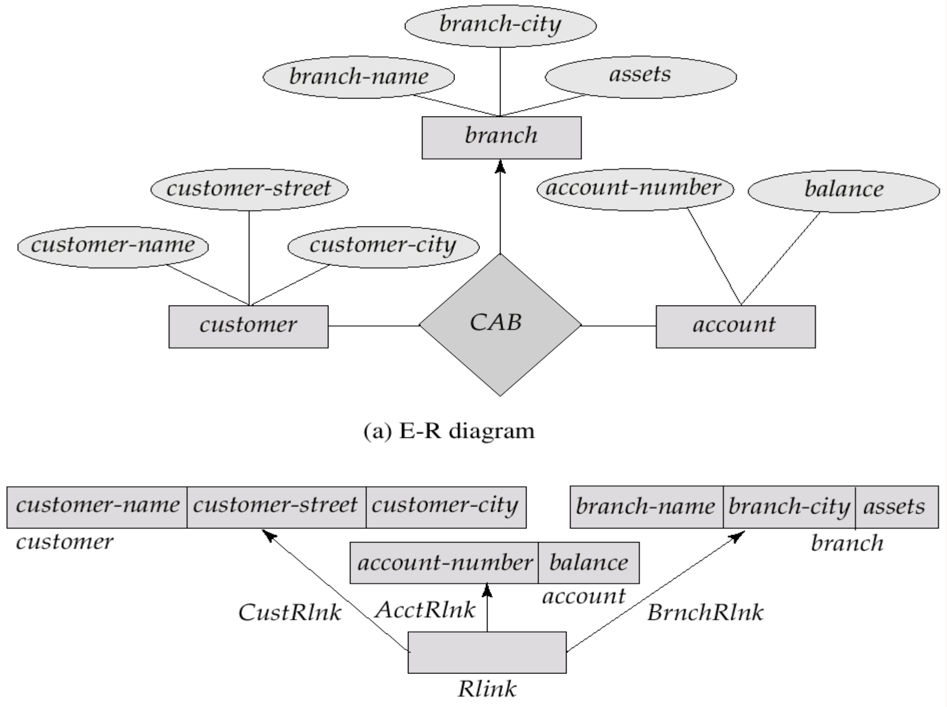


(a) E-R diagram

(b) Network diagram

To represent an E-R relationship of degree 3 or higher, connect the participating record types through a new record type that is linked directly to each of the original record types.

1. Replace entity sets *account, customer,* and *branch* with record types *account, customer,* and *branch,* respectively.
2. Create a new record type *Rlink* (referred to as a *dummy* record type).
3. Create the following many-to-one links:
   - *CustRlink* from *Rlink* record type to *customer* record type
   - *AcctRlnk* from *Rlink* record type to *account* record type
   - *BrncRlnk* from *Rlink* record type to *branch* record type

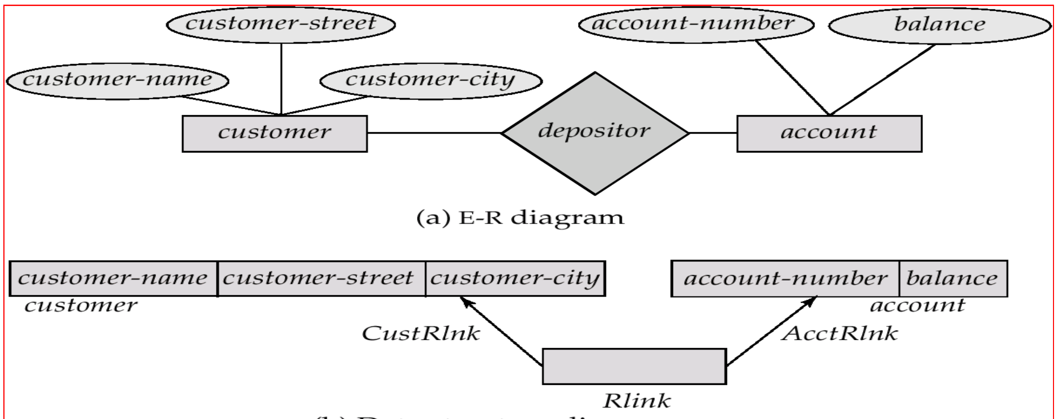


(a) E-R diagram

(b) Data structure diagram

**The DBTG CODASYL Model:**

- All links are treated as many-to-one relationships.
- To model many-to-many relationships, a record type is defined to represent the relationship and two links are used.



(a) E-R diagram

(b) Data structure diagram

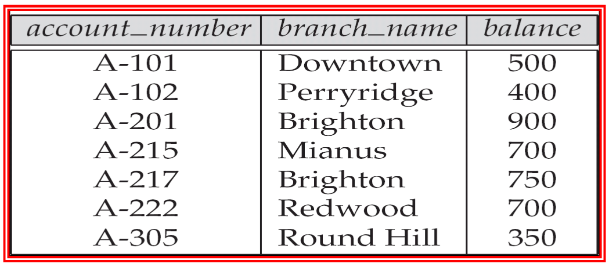
**DBTG Sets:**

- The structure consisting of two record types that are linked together is referred to in the DBTG model as a *DBTG set*
- In each DBTG set, one record type is designated as the *owner,* and the other is designated as the *member*, of the set.
- Each DBTG set can have any number of *set occurrences* (actual instances of linked records).
- Since many-to-many links are disallowed, each set occurrence has precisely one owner, and has zero or more member records.
- No member record of a set can participate in more than one occurrence of the set at any point.
- A member record can participate simultaneously in several set occurrences of *different* DBTG sets.

# LECTURE-10: RELATIONAL MODEL

## RELATIONAL MODEL

Relational model is simple model in which database is represented as a collection of “relations” where each relation is represented by two-dimensional table.



| account_number | branch_name | balance |
|---|---|---|
| A-101 | Downtown | 500 |
| A-102 | Perryridge | 400 |
| A-201 | Brighton | 900 |
| A-215 | Mianus | 700 |
| A-217 | Brighton | 750 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |

The relational model was founded by E. F. Codd  of the IBM in 1972. The basic concept in the relational model is that of a relation.

**Properties:**

- It is column homogeneous. In other words, in any given column of a table, all items are of the same kind.
- Each item is a simple number or a character string. That is a table must be in first normal form.
- All rows of a table are distinct.
- The ordering of rows with in a table is immaterial.
- The column of a table are assigned distinct names and the ordering of these columns is immaterial.

**Domain, attributes tuples and relational:**

**Tuple:**

Each row in a table represents a record and is called a tuple .A table containing ‘n’ attributes in a record is called is called n-tuple.

**Attributes:**

The name of each column in a table is used to interpret its meaning and is called an attribute.Each table is called a relation. In the above table, account_number, branch name, balance are the attributes.

**Domain:**

A domain is a set of values that can be given to an attributes. So every attribute in a table has a specific domain. Values to these attributes can not be assigned outside their domains.

**Relation:**

A relation consist of

- **Relational schema**
- **Relation instance**

**Relational Schema:**

A relational schema specifies the relation’s name, its attributes and  the domain of each attribute. If

R is the name of a relation and A1, A2,…An is a list of attributes representing R then R(A1,A2,…,An) is called a Relational Schema. Each attribute in this relational schema takes a value from some specific domain called domain(Ai).

**Example**:

PERSON (PERSON_ID:INTEGER, NAME:STRING, AGE:INTEGER, ADDRESS:STRING)

Total number of attributes in a relation denotes the degree of a relation since the PERSON relation scheme contains four attributes, so this relation is of degree 4.

**Relation Instance:**

A relational instance denoted as r is a collection of tuples for a given relational schema at a specific point of time.

A relation state r to the relations schema R(A1, A2…, An) also denoted by r(R) is a set of n-tuples

R{t1,t2,…tm}

Where each n-tuple is an ordered list of n values

T=<v1,v2,….vn>

Where each vi belongs to domain (Ai) or contains null values.

The relation schema is also called 'intension' and the relation state is also called 'extension'.

Eg: Relation schema for Student

STUDENT(rollno:string, name:string, city:string, age:integer)

**Relation instance:**

**Student:**

| Rollno | Name | City | Age |
|---|---|---|---|
| 101 | Sujit | Bam | 23 |
| 102 | kunal | bbsr | 22 |

**Keys:**

**Super key:**

A super key is an attribute or a set of attributes used to identify the records uniquely in a relation.

For example, customer-id, (cname, customer-id), (cname,telno)

**Candidate key:**

Super keys of a relation can contain extra attributes. Candidate keys are minimal super keys. i.e, such a key contains no extraneous attribute. An attribute is called extraneous if even after removing it from the key, makes the remaining attributes still has the properties of a key(atribute represents entire table).

In a relation R, a candidate key for R is a subset of the set of attributes of R, which have the following properties:

- *Uniqueness:* no two distinct tuples in R have the same values for the candidate key
- *Irreducible:* No proper subset of the candidate key has the *uniqueness property that is the candidate key.*
- *A candidate key's values must exist. It can't be null.*
- *The values of a candidate key must be stable. Its value can not change outside the*

*control of the system.*

Eg: (cname,telno)

**Primary key:**

The primary key is the candidate key that is chosen by the database designer as the principal means of identifying entities with in an entity set. The remaining candidate keys if any are called ***alternate key*.**

# LECTURE-11: CONSTRAINTS

**RELATIONAL CONSTRAINTS:**

There are three types of constraints on relational database that include

- DOMAIN CONSTRAINTS
- KEY CONSTRAINTS
- INTEGRITY CONSTRAINTS

**DOMAIN CONSTRAINTS:**

It specifies that each attribute in a relation an atomic value from the corresponding domains. The data types associated with commercial RDBMS domains include:

- Standard numeric data types for integer
- Real numbers
- Characters
- Fixed length strings and variable length strings

Thus, domain constraints specifies the condition that we to put on each instance of the relation. So the values that appear in each column must be drawn from the domain associated with that column.

| Rollno | Name | City | Age |
|---|---|---|---|
| 101 | Sujit | Bam | 23 |
| 102 | kunal | bbsr | 22 |

**Key Constraints:**

This constraints states that the key attribute value in each tuple msut be unique .i.e, no two tuples contain the same value for the key attribute.(null values can allowed)

Emp(empcode,name,address) . here empcode can be unique

**Integrity CONSTRAINTS:**

There are two types of integrity constraints:

- Entity Integrity Constraints
- Referential Integrity constraints

**Entity Integrity Constraints:**

It states that no primary key value can be null and unique. This is because the primary key is used to identify individual tuple in the relation. So we will not be able to identify the records uniquely containing null values for the primary key attributes. This constraint is specified on one individual relation.

**Referential Integrity Constraints:**

It states that the tuple in one relation that refers to another relation must refer to an existing tuple in that relation. This constraints is specified on two relations. If a column is declared as foreign key that must be primary key of another table.

**Department (deptcode, dname)**

Here the deptcode is the primary key.

**Emp (*empcode*, name, city, *deptcode*).**

Here the deptcode is foreign key.

## CODD'S RULES

**Rule 1 : The information Rule.**

"All information in a relational data base is represented explicitly at the logical level and in exactly one way - by values in tables."

Everything within the database exists in tables and is accessed via table access routines.

**Rule 2 : Guaranteed access Rule.**

"Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name."

To access any data-item you specify which column within which table it exists, there is no reading of characters 10 to 20 of a 255 byte string.

**Rule 3 : Systematic treatment of null values.**

"Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type."

If data does not exist or does not apply then a value of NULL is applied, this is understood by the RDBMS as meaning non-applicable data.

**Rule 4 : Dynamic on-line catalog based on the relational model.**

"The data base description is represented at the logical level in the same way as-ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data."

The Data Dictionary is held within the RDBMS, thus there is no-need for off-line volumes to tell you the structure of the database.

**Rule 5 : Comprehensive data sub-language Rule.**

"A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items

- Data Definition
- View Definition
- Data Manipulation (Interactive and by program).
- Integrity Constraints
- Authorization.

Every RDBMS should provide a language to allow the user to query the contents of the RDBMS and also manipulate the contents of the RDBMS.

**Rule 6 : .View updating Rule**

"All views that are theoretically updateable are also updateable by the system."

Not only can the user modify data, but so can the RDBMS when the user is not logged-in.

**Rule 7 : High-level insert, update and delete.**

"The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data."

The user should be able to modify several tables by modifying the view to which they act as base tables.

**Rule 8 : Physical data independence.**

"Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods."

The user should not be aware of where or upon which media data-files are stored

**Rule 9 : Logical data independence.**

"Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables."

User programs and the user should not be aware of any changes to the structure of the tables (such as the addition of extra columns).

**Rule 10 : Integrity independence.**

"Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs."

If a column only accepts certain values, then it is the RDBMS which enforces these constraints and not the user program, this means that an invalid value can never be entered into this column, whilst if the constraints were enforced via programs there is always a chance that a buggy program might allow incorrect values into the system.

**Rule 11 : Distribution independence.**

"A relational DBMS has distribution independence."

The RDBMS may spread across more than one system and across several networks, however to the end-user the tables should appear no different to those that are local.

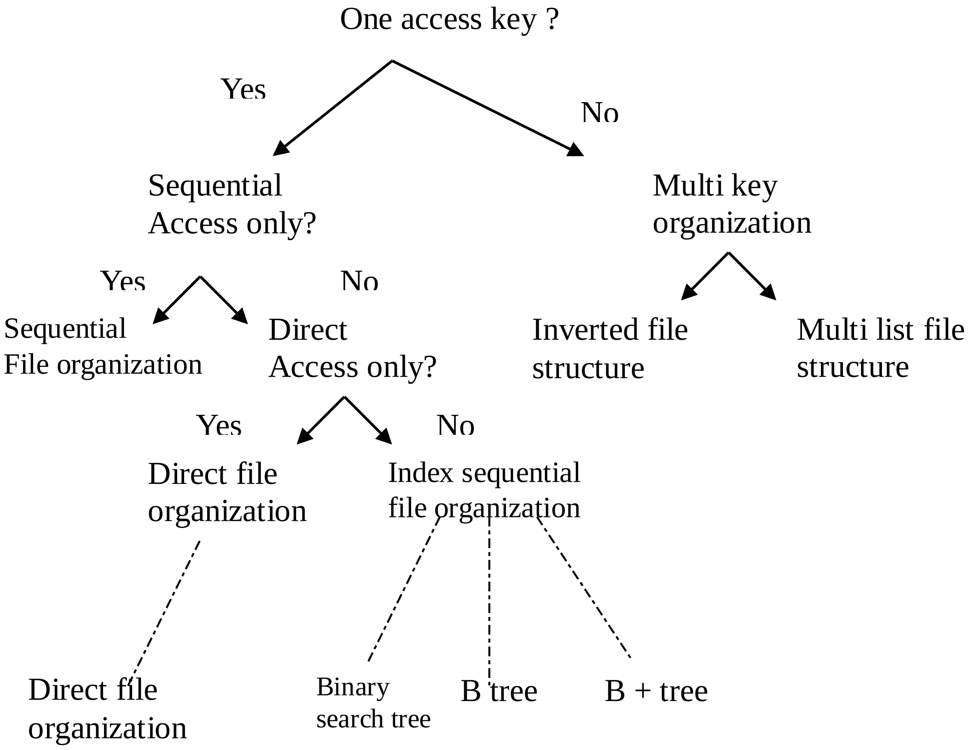
**Rule 12 : Non-subversion Rule.**

"If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity Rules and constraints expressed in the higher level relational language (multiple-records-at-a-time)."

# LECTURE-12: FILE ORGANISATION

## FILE ORGANISATION AND ITS TYPES:

A file organization is a technique to organize data in the secondary memory. File organization is a way of arranging the records in a file when the file is stored on the disk. Data files are organized so as to facilitate access to records and to ensure their efficient storage. A DBMS supports several file organization techniques.



**(File organization techniques)**

### Heap files (unordered file)

Basically these files are unordered file. It is the simplest and most basic type. These files consist of randomly ordered records. The records will have no particular order.The operations we can perform on the records are insert , retrieve and delete. The features of the heap file organization are:

- New records can be inserted in any empty space that can accommodate them.
- When old records are deleted, the occupied space becomes empty and available for any new insertion.
- If updated records grow, they may need to be relocated to a new empty space. This needs to keep a list of empty space.

**Advantage of heap files:**

1. This is a simple file organization method
2. Insertion is somehow efficient
3. Good for bulk-lading data into a table.
4. Best if file scans are common or insertions are frequent

**Disadvantages of heap files:**

1. Retrieval requires a linear search and is inefficient

2. Deletion can result in unused space/need for reorganization

**Sequential file organization:**

The most basic way to organize the collection of records in a file is to use sequential organization. Records of the file are stored in sequence by the primary key field values/ They are accessible only in the order stored i.e, in the primary key order. This kind is of file organization works well for tasks which need to access nearly every record in a file. Eg. Payroll..

In a sequentially organized file records are written consecutively when the file is created and must be accessed consecutively when the file later used for input.

A sequential file maintains the records in the logical sequence of its primary key values. Sequential file are inefficient for random access. And files can be stored on devices like magnetic tape that allow sequential access. As records are in sorted order it will use binary search technique to search for a record.

**Advantages of sequential file organization:**

- It is fast and efficient when dealing with large volumes of data that need to be processed periodically(batch system)

**Disadvantages of sequential file organization:**

- Requires that all new transactions be sorted into the proper sequence for sequential access processing
- Locating, storing, modifying, deleting or adding records in the file require rearranging the file/
- This method is too slow to handle application requiring immediate updating or responses.

**Indexed sequential file organization:**

It organized the file like a large dictionary, i.e, records are stored in order of the key, but an index is kept which also permits a type of direct access. The records are stored sequentially by primary key values and there is an index built over the primary key field.

An index is a set of index value, address pairs. Indexing associates a set of objects to a set pf orderable quantities, that are usually smaller in number or their properties. Thus an index is a mechanism for faster search. Although the indices and the data blocks are kept physically, they are logically distinct.

A sequential file that is indexed on its primary key is called an index sequential file. The index allows for random access to records, while the sequential storage of the records of the file provides easy access to the sequential records. An addition feature of this file system is the overflow area. The overflow area provides additional space for record addition without the need to create.

**Advantage of ISAM indexes:**

1. Because the whole structure is ordered to a large extent, partial (like ty%) and range(between 12 and 29) based retrievals can often benefit from the use of this type of index.
2. ISAM is good for static tables because there are usually fewer index levels than B-tree
3. In general there are fewer disk I/Os required to access data, provided there is no overflow.

**Disadvantage of ISAM indexes:**

1. ISAM is still not as quick as some in hash file organization
2. Overflow can be a real problem in highly volatile table.

**Hashed file organization:**

Hashing is the most common form of purely random access to a file or database. It is also used to access columns that do not have a index as an optimization technique. Hash functions calculate the address of the page in which the record is to be stored based on one or more fields in the record. The records in a hash file appear randomly distributed across the available space. It requires some hashing algorithm and the technique. Hashing algorithm converts a primary key value into a record address.

**Advantage of hashed file organization:**

1. Insertion or search on hash key is fast.
2. Best if equality search is needed on hash key

**Disadvantage of hashed file organization:**

1. It is a complex file organization method
2. search is slow
3. It suffers from disk space overhead
4. Unbalanced buckets degrade performance
5. Range search is slow

# LECTURE-13: INDEX

## Types of Indexes:

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

**Ordered Indices**

Indexing techniques evaluated on basis of:

1. In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
2. **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
   a. Also called **clustering index**
   b. The search key of a primary index is usually but not necessarily the primary key.
3. **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called
   non-clustering index**.**
4. Index-sequential file**:** ordered sequential file with a primary index.

There are two types of ordered indices that we can use:

**Dense index**:

An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key. Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for primary indices.

**Sparse index**:

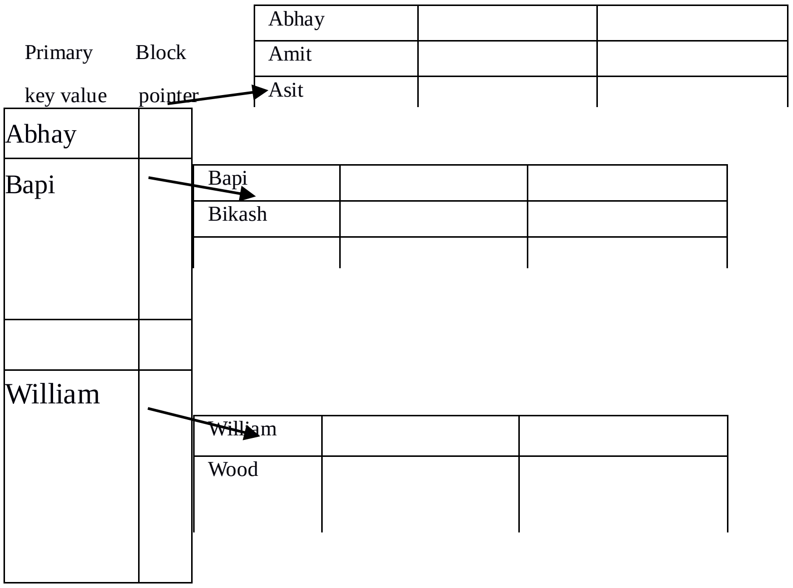
An index record appears for only some of the search-key values. As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

**Primary Indexes:**

A primary index is an ordered file whose records are of fixed length with two fields. The first field is of the same data types as the ordering key filed of the data file and the second field is a pointer to a disk block- a block address. The ordering key field is called the primary key of the data file. There is one index entry in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to other block as its two fields values.

The first record of each block of the data file is known as anchor record of the block.

Primary index is an example of non dense index



**(Primary indexes on the ordering key field)**

A major problem with primary index as with any ordered file- is insertion and deletion of records. With a primary Index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we not only have to move records to make space for the new record but also have to change some index entries because moving records will change the anchor records of some blocks.

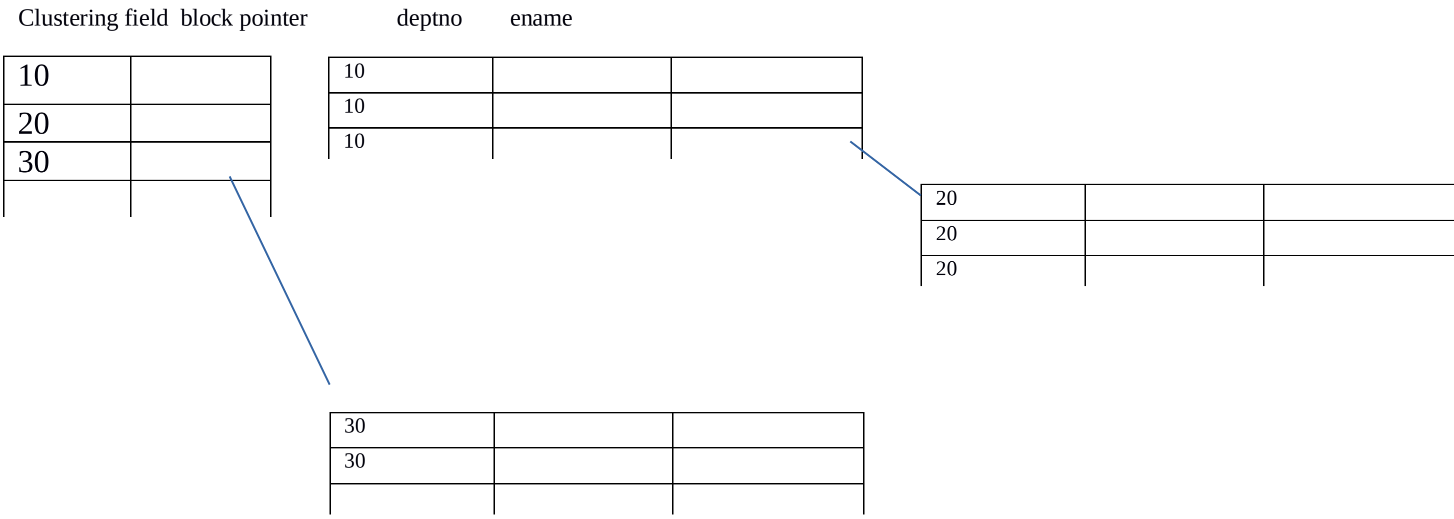
# LECTURE-14: Clustering Index

## Clustering Indexes:

If the records of a file are physically ordered on a non key field that does not have a distinct value for each record, that filed is called the clustering filed of the file. We can create a different type of index called clustering index to speed up retrieval of records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record.

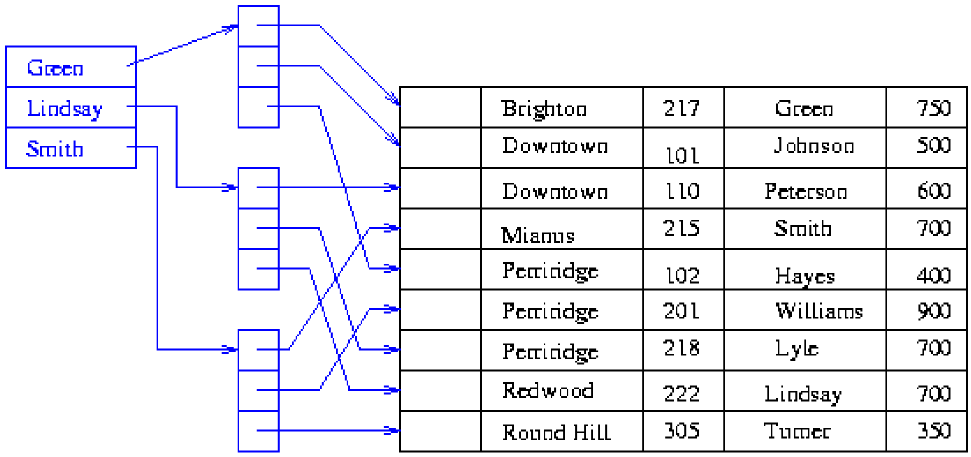
A clustering index is also an ordered file with two fields, the first field is of the same type as the clustering field of the data file and the second field is block pointer. There is one entry in the clustering index for each distinct value of the clustering field, containing that value and a pointer to the first block in the data file that has a record with that value for its clustering field.

A clustering index is another example of non dense index.



## Secondary Indexes:

A secondary index also is an ordered file with two fields and as in the other indexes, the second filed is a pointer to a disk block. The first field is of the same type as some non ordering field of the data file. The field on which the secondary index is constructed is called an indexing field of the file. Whether its values are distinct for every record or not.



| | | | | |
|---|---|---|---|---|
| | Brighton | 217 | Green | 750 |
| | Downtown | 101 | Johnson | 500 |
| | Downtown | 110 | Peterson | 600 |
| | Mianus | 215 | Smith | 700 |
| | Perriridge | 102 | Hayes | 400 |
| | Perriridge | 201 | Williams | 900 |
| | Perriridge | 218 | Lyle | 700 |
| | Redwood | 222 | Lindsay | 700 |
| | Round Hill | 305 | Turner | 350 |

There can be many secondary indexes and hence indexing fields for the same file.

**Primary and Secondary Indices:**

Secondary indices have to be dense.

1. Indices offer substantial benefits when searching for records.
2. When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
3. Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
    - **a.** each record access may fetch a new block from disk

# LECTURE-15: B$^+$ Tree Index

**B+-Tree Index Files:**

B+-tree indices are an alternative to indexed-sequential files.

1. Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
2. Advantage of B+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
3. Disadvantage of B+-trees: extra insertion and deletion overhead, space overhead.
4. Advantages of B+-trees outweigh disadvantages, and they are used extensively.

A B+-tree is a rooted tree satisfying the following properties:

1. All paths from root to leaf are of the same length
2. Each node that is not a root or a leaf has between [*n*/2] and *n* children.
3. A leaf node has between [(*n*–1)/2] and *n*–1 values
4. Special cases:
   a. If the root is not a leaf, it has at least 2 children.
   b. If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (*n*–1) values.

B$^+$-Tree Node Structure

Typical node

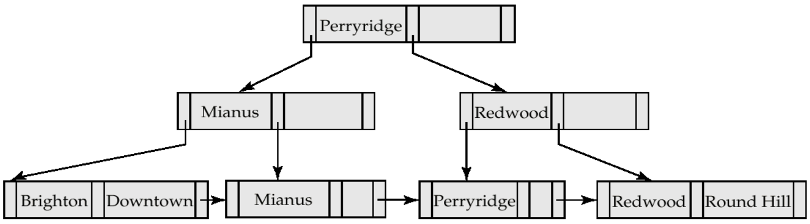
| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | . . . | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

The search-keys in a node are ordered

$K_1 < K_2 < K_3 < \ldots < K_{n-1}$

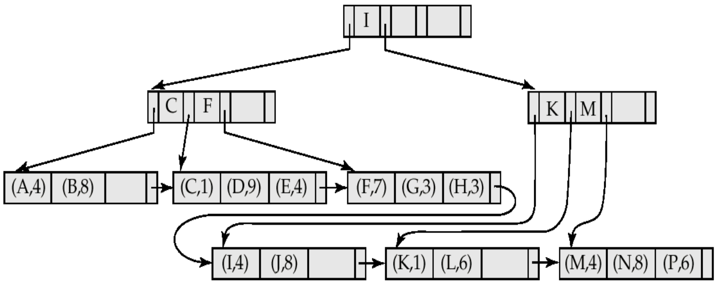
**Example of a B+-tree**



**B+-Tree File Organization**

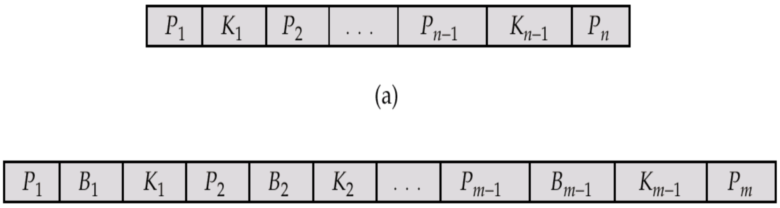
1. Index file degradation problem is solved by using B+-Tree indices. Data file degradation problem is solved by using B+-Tree File Organization.
2. The leaf nodes in a B+-tree file organization store records, instead of pointers.
3. Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non leaf node.

4. Leaf nodes are still required to be half full.
5. Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index.



## B-Tree Index Files

1. Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
2. Search keys in non leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non leaf node must be included.
3. Generalized B-tree leaf node

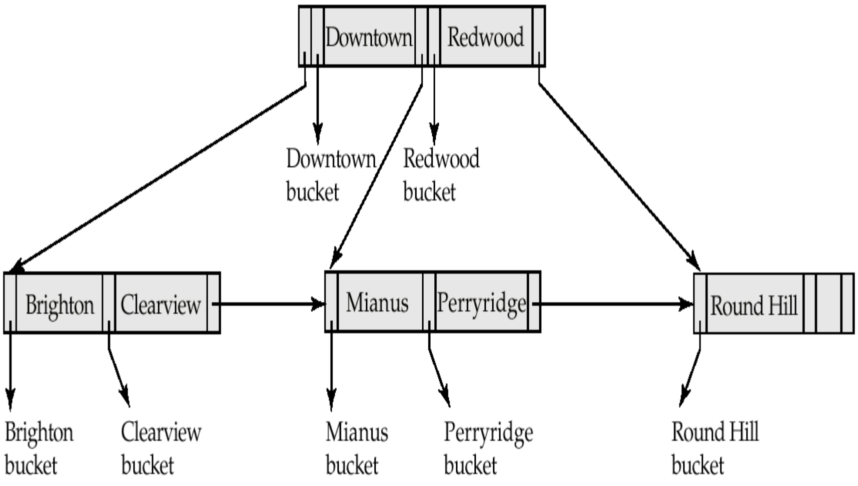


(a)

(b)

4. Non leaf node – pointers *Bi* are the bucket or file record pointers.

**B-Tree Index File Example:**



**Advantages of B-Tree indices:**

- May use less tree nodes than a corresponding B+-Tree.
- Sometimes possible to find search-key value before reaching leaf node.

**Disadvantages of B-Tree indices:**

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B+-Tree

- Insertion and deletion more complicated than in B+-Trees
- Implementation is harder than B+-Trees.

Typically, advantages of B-Trees do not out weigh disadvantages.

# LECTURE-16: Hash File Organization

## Hash File Organization

In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record. In our description of hashing, we shall use the term **bucket** to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

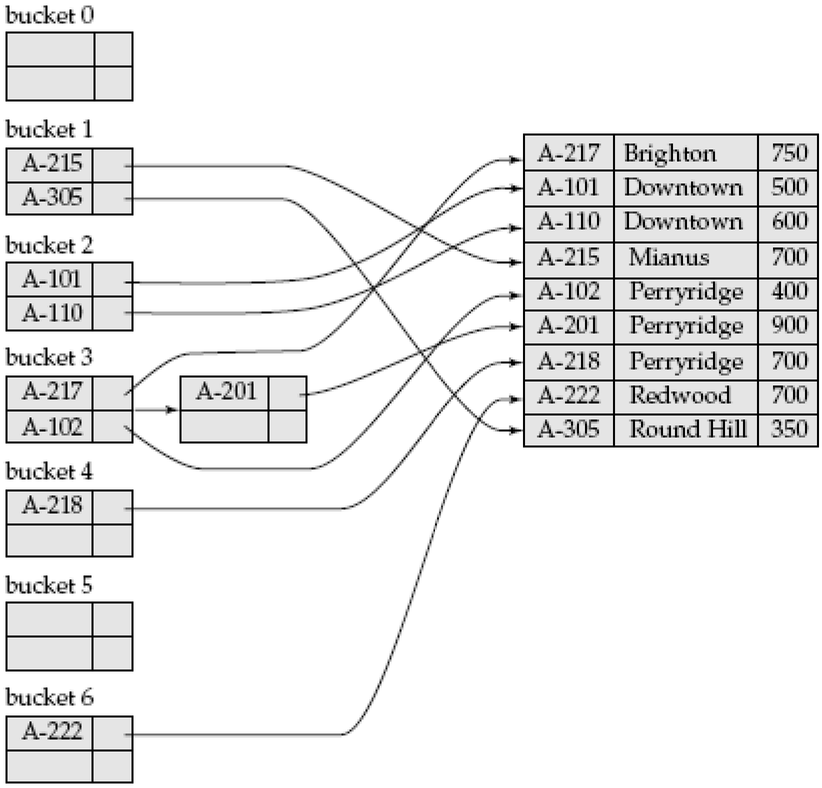
Formally, let *K* denote the set of all search-key values, and let *B* denote the set of all bucket addresses. A **hash function** *h* is a function from *K* to *B*. Let *h* denote a hash function.

To insert a record with search key *Ki*, we compute *h*(*Ki*), which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

To perform a lookup on a search-key value *Ki*, we simply compute *h*(*Ki*), then search the bucket with that address. Suppose that two search keys, *K*5 and *K*7, have the same hash value; that is, *h*(*K*5) = *h*(*K*7). If we perform a lookup on *K*5, the bucket *h*(*K*5) contains records with search-key values *K*5 and records with search key values *K*7. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want. Deletion is equally straightforward. If the search-key value of the record to be deleted is *Ki*, we compute *h*(*Ki*), then search the corresponding bucket for that record, and delete the record from the bucket.

### Hash Indices

Hashing can be used not only for file organization, but also for index-structure creation. A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure shows a secondary hash index on the *account* file, for the search key *account-number*. The hash function in the figure computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2 bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *account-number* is a primary key for *account*, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.



| | | |
|---|---|---|
| A-217 | Brighton | 750 |
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |

**Figure 12.23** Hash index on search key *account-number* of *account* file.

We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a primary index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a primary hash index on it.

**Advantages of hashing:**

1. Exact key matches are extremely quick
2. Hashing is very good long keys, or those with multiple columns, provided the complete key value is provided for the query
3. This organization usually allows for the allocation of disk space so a good deal of disk management is possible
4. No disk space is used by this indexing method

**Disadvantages of hashing:**

1. It becomes difficult to predict overflow because the working of the hashing algorithm will not be visible to the DBA
2. No sorting of data occurs either physically or logically so sequential access is poor
3. This organization usually takes a lot of disk space to ensure that no overflow occurs.

# LECTURE-17: Query Processing

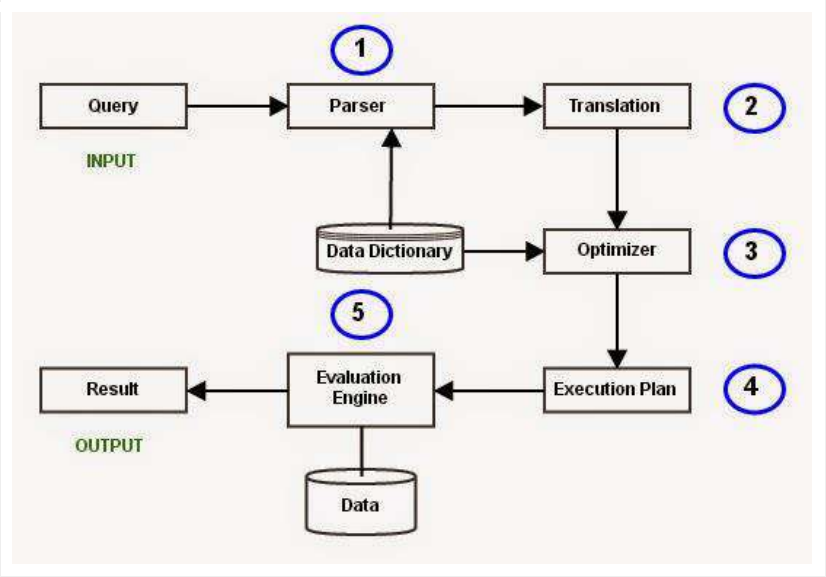
**Query Processing**

Query Processing would mean the entire process or activity which involves query translation into low level instructions, query optimization to save resources, cost estimation or evaluation of query, and extraction of data from the database.

Goal: To find an efficient Query Execution Plan for a given SQL query which would minimize the cost considerably, especially time.

Cost Factors: Disk accesses [which typically consumes time], read/write operations [which typically needs resources such as memory/RAM].

The major steps involved in query processing are depicted in the figure below;



Let us discuss the whole process with an example. Let us consider the following two relations as the example tables for our discussion;

Employee(Eno, Ename, Phone)
Proj_Assigned(Eno, Proj_No, Role, DOP)

where,

Eno is Employee number,
Ename is Employee name,
Proj_No is Project Number in which an employee is assigned,
Role is the role of an employee in a project,
DOP is duration of the project in months.

With this information, let us write a query to find the list of all employees who are working in a project which is more than 10 months old.

```
SELECT Ename
FROM Employee, Proj_Assigned
WHERE Employee.Eno = Proj_Assigned.Eno AND DOP > 10;
```

**Input:**

A query written in SQL is given as input to the query processor. For our case, let us consider the SQL query written above.

**Step 1: Parsing**

In this step, the parser of the query processor module checks the syntax of the query, the user's privileges to execute the query, the table names and attribute names, etc. The correct table names, attribute names and the privilege of the users can be taken from the system catalog (data dictionary).

**Step 2: Translation**

If we have written a valid query, then it is converted from high level language SQL to low level instruction in Relational Algebra.

For example, our SQL query can be converted into a Relational Algebra equivalent as follows;

$$\pi_{Ename}(\sigma_{DOP>10 \land Employee.Eno=Proj\_Assigned.Eno}(\text{Employee X Prof\_Assigned}))$$

**Step 3: Optimizer**

Optimizer uses the statistical data stored as part of data dictionary. The statistical data are information about the size of the table, the length of records, the indexes created on the table, etc. Optimizer also checks for the conditions and conditional attributes which are parts of the query.

**Step 4: Execution Plan**

A query can be expressed in many ways. The query processor module, at this stage, using the information collected in step 3 to find different relational algebra expressions that are equivalent and return the result of the one which we have written already.

For our example, the query written in Relational algebra can also be written as the one given below;

$$\pi_{Ename}(\text{Employee} \bowtie_{Eno} (\sigma_{DOP>10}(\text{Prof\_Assigned})))$$

So far, we have got two execution plans. Only condition is that both plans should give the same result.

**Step 5: Evaluation**

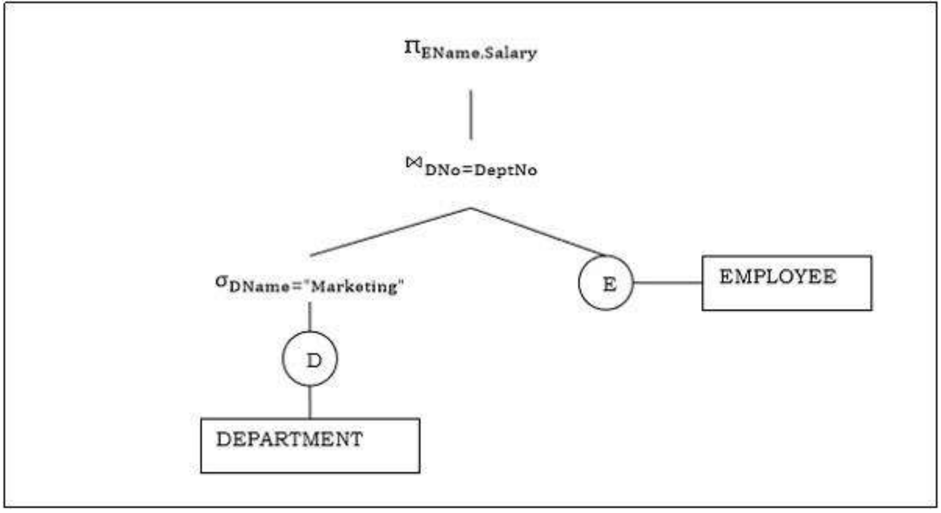
Though we got many execution plans constructed through statistical data, though they return same result (obvious), they differ in terms of Time consumption to execute the query, or the Space required executing the query. Hence, it is mandatory choose one plan which obviously consumes less cost.

At this stage, we choose one execution plan of the several we have developed. This Execution plan accesses data from the database to give the final result.

In our example, the second plan may be good. In the first plan, we join two relations (costly operation) then apply the condition (conditions are considered as filters) on the joined relation. This consumes more time as well as space.

In the second plan, we filter one of the tables (Proj_Assigned) and the result is joined with the Employee table. This join may need to compare less number of records. Hence, the second plan is the best (with the information known, not always).

**Query Tree**



Used in query representation used in parsing.

**Query Optimization:**

A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

**There are broadly two ways a query can be optimized:**

1. Analyze and transform equivalent relational expressions: Try to minimize the tuple and column counts of the intermediate and final query processes (discussed here).
2. Using different algorithms for each operation: These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block accesses (discussed in query processing).

**Analyze and transform equivalent relational expressions**

Here, we shall talk about generating minimal equivalent expressions. To analyze equivalent expression, listed are a set of equivalence rules. These generate equivalent expressions for a query written in relational algebra. To optimize a query, we must convert the query into its equivalent form as long as an equivalence rule is satisfied.

1. **Conjunctive selection operations can be written as a sequence of individual selections. This is called a sigma-cascade.**

   **Explanation:** Applying condition intersection is expensive. Instead, filter out tuples satisfying condition (inner selection) and then apply condition (outer selection) to the then resulting fewer tuples. This leaves us with less tuples to process the second time. This can be extended for two or more intersecting selections. Since we are breaking a single condition into a series of selections or cascades, it is called a “cascade”.
2. **Selection is commutative.**

   **Explanation:** condition is commutative in nature. This means, it does not matter whether we apply first or first. In practice, it is better and more optimal to apply that selection first which yields a fewer number of tuples. This saves time on our outer selection.
3. **All following projections can be omitted, only the first projection is required. This is called a pi-cascade.**

   **Explanation:** A cascade or a series of projections is meaningless. This is because in the end, we are only selecting those columns which are specified in the last, or the outermost projection. Hence, it is better to collapse all the projections into just one i.e. the outermost projection.
4. **Selections on Cartesian Products can be re-written as Theta Joins.**
   - **Equivalence 1**

     **Explanation:** The cross product operation is known to be very expensive. This is because it matches each tuple of E1 (total m tuples) with each tuple of E2 (total n tuples). This yields m*n entries. If we apply a selection operation after that, we would have to scan through m*n entries to find the suitable tuples which satisfy the condition . Instead of doing all of this, it is more optimal to use the Theta Join, a join specifically designed to select only those entries in the cross product which satisfy the Theta condition, without evaluating the entire cross product first.
   - **Equivalence 2**

     **Explanation:** Theta Join radically decreases the number of resulting tuples, so if we apply an intersection of both the join conditions i.e. and into the Theta Join itself, we get fewer scans to do. On the other hand, a condition outside unnecessarily increases the tuples to scan.

5.**Theta Joins are commutative.**

**Explanation:** Theta Joins are commutative, and the query processing time depends to some extent which table is used as the outer loop and which one is used as the inner loop during the join process (based on the indexing structures and blocks).

6.**Join operations are associative.**

- **Natural Join**

  **Explanation:** Joins are all commutative as well as associative, so one must join those two tables first which yield less number of entries, and then apply the other join.

- **Theta Join**

  **Explanation:** Theta Joins are associative in the above manner, where involves attributes from only E2 and E3.

7.**Selection operation can be distributed.**

- **Equivalence 1**

  **Explanation:** Applying a selection after doing the Theta Join causes all the tuples returned by the Theta Join to be monitored after the join. If this selection contains attributes from only E1, it is better to apply this selection to E1 (hence resulting in a fewer number of tuples) and then join it with E2.

- **Equivalence 2**

  **Explanation:** This can be extended to two selection conditions, and , where Theta1 contains the attributes of only E1 and contains attributes of only E2. Hence, we can individually apply the selection criteria before joining, to drastically reduce the number of tuples joined.

8.**Projection distributes over the Theta Join.**

- **Equivalence 1**

  **Explanation:** The idea discussed for selection can be used for projection as well. Here, if L1 is a projection that involves columns of only E1, and L2 another projection that involves the columns of only E2, then it is better to individually apply the projections on both the tables before joining. This leaves us with a fewer number of columns on either side, hence contributing to an easier join.

- **Equivalence 2**

  **Explanation:** Here, when applying projections L1 and L2 on the join, where L1 contains columns of only E1 and L2 contains columns of only E2, we can introduce another column E3 (which is common between both the tables). Then, we can apply projections L1 and L2 on E1 and E2 respectively, along with the added column L3. L3 enables us to do the join.

9.**Union and Intersection are commutative.**

**Explanation:** Union and intersection are both distributive; we can enclose any tables in parantheses according to requirement and ease of access.

10. **Union and Intersection are associative.**

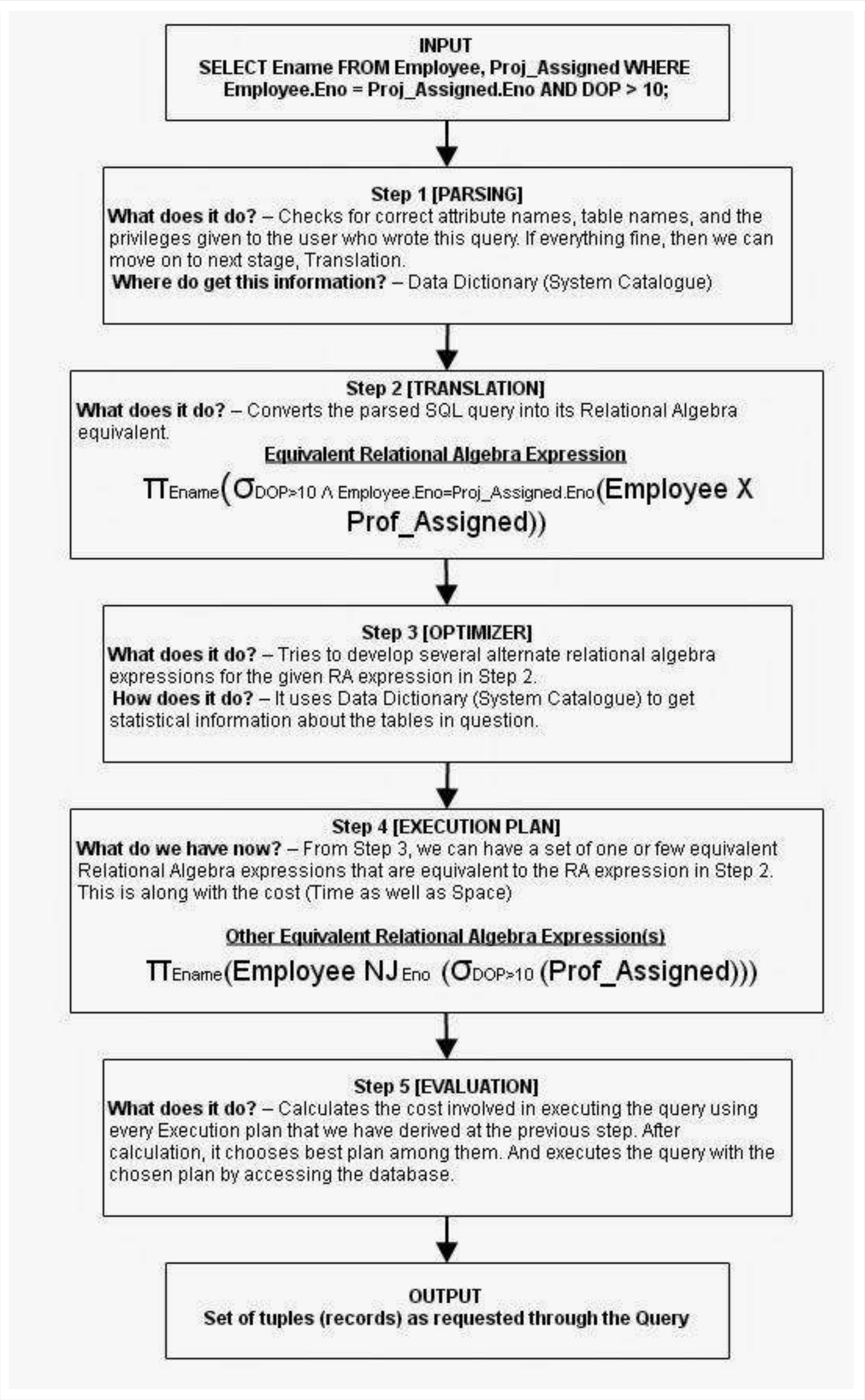
**Explanation:** Union and intersection are both distributive; we can enclose any tables in parantheses according to requirement and ease of access.

11. **Selection operation distributes over the union, intersection, and difference operations.**

**Explanation:** In set difference, we know that only those tuples are shown which belong to table E1 and do not belong to table E2. So, applying a selection condition on the entire set difference is equivalent to applying the selection condition on the individual tables and then applying set difference. This will reduce the number of comparisons in the set difference step.

**12. Projection operation distributes over the union operation.**

**Explanation:** Applying individual projections before computing the union of E1 and E2 is more optimal than the left expression, i.e. applying projection after the union step.



**INPUT**
**SELECT Ename FROM Employee, Proj_Assigned WHERE Employee.Eno = Proj_Assigned.Eno AND DOP > 10;**

**Step 1 [PARSING]**
**What does it do?** – Checks for correct attribute names, table names, and the privileges given to the user who wrote this query. If everything fine, then we can move on to next stage, Translation.
**Where do get this information?** – Data Dictionary (System Catalogue)

**Step 2 [TRANSLATION]**
**What does it do?** – Converts the parsed SQL query into its Relational Algebra equivalent.
<u>Equivalent Relational Algebra Expression</u>

$$\pi_{Ename}(\sigma_{DOP>10 \wedge Employee.Eno=Proj\_Assigned.Eno}(Employee\ X\ Prof\_Assigned))$$

**Step 3 [OPTIMIZER]**
**What does it do?** – Tries to develop several alternate relational algebra expressions for the given RA expression in Step 2.
**How does it do?** – It uses Data Dictionary (System Catalogue) to get statistical information about the tables in question.

**Step 4 [EXECUTION PLAN]**
**What do we have now?** – From Step 3, we can have a set of one or few equivalent Relational Algebra expressions that are equivalent to the RA expression in Step 2. This is along with the cost (Time as well as Space)
<u>Other Equivalent Relational Algebra Expression(s)</u>

$$\pi_{Ename}(Employee\ NJ_{Eno}\ (\sigma_{DOP>10}\ (Prof\_Assigned)))$$

**Step 5 [EVALUATION]**
**What does it do?** – Calculates the cost involved in executing the query using every Execution plan that we have derived at the previous step. After calculation, it chooses best plan among them. And executes the query with the chosen plan by accessing the database.

**OUTPUT**
**Set of tuples (records) as requested through the Query**

# LECTURE-18: Evaluation of Expressions

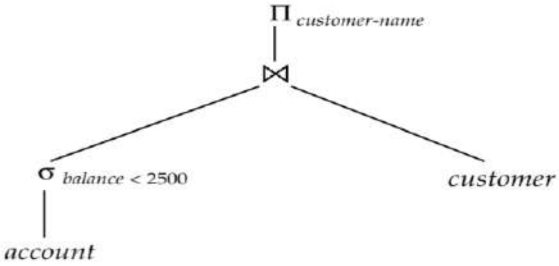
## Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - ★ **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - ★ **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

### Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

  then compute the store its join with *customer,* and finally compute the projections on *customer-name.*



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - ★ Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations +
      cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - ★ Allows overlap of disk writes with computation and reduces execution time

### Pipelining

- **Pipelined evaluation :** evaluate several operations simultaneously, passing the results of one operation on to the next.
  - ★ E.g., in previous expression tree, don't store result of instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**
- In **demand driven** or **lazy** evaluation
  - ★ system repeatedly requests next tuple from top level operation
  - ★ Each operation requests next tuple from children operations as required, in order to output its next tuple

- - In between calls, operation has to maintain “**state**” so it knows what to return next
  - Each operation is implemented as an **iterator** implementing the following operations
    - open()
      - E.g. file scan: initialize file scan, store pointer to beginning of file as state
      - E.g.merge join: sort relations and store pointers to beginning of sorted relations as state
    - next()
      - E.g. for file scan: Output next tuple, and advance and store file pointer
      - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - close()

- In produce-driven or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples

**Evaluation Algorithms for Pipelining**

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - These result in intermediate results being written to disk and then read back always
- Algorithm variants are possible to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
  - **Pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new $r_0$ tuple is found, match it with existing $s_0$ tuples, output matches, and save it in $r_0$
    - Symmetrically for $s_0$ tuples

**Complex Joins**

- Join involving three relations: *loan* *depositor* *customer*
- **Strategy 1.** Compute *depositor* *customer;* use result to compute *loan* (*depositor* *customer*)
- **Strategy 2.** Computer *loan* *depositor* first, and then join the result with *customer.*
- **Strategy 3.** Perform the pair of joins at once. Build and index on *loan* for *loan-number,* and on *customer* for *customer-name.*
  - For each tuple *t* in *depositor,* look up the corresponding tuples in *customer* and the corresponding tuples in *loan.*
  - Each tuple of *deposit* is examined exactly once.
- Strategy 3 combines two operations into one special-purpose operation that is more efficient than implementing two joins of two relations.

**Module-2:**
**LECTURE-19**

## Relational Algebra:

Basic operations:

1. *Selection* (σ) Selects a subset of rows from relation.
2. *Projection* (π) Selects a subset of columns from relation.
3. *Cross-product* (×) Allows us to combine two relations.
4. *Set-difference* (□) Tuples in relation. 1, but not in relationn. 2.
5. *Union* (U) Tuples in reln. 1 and in reln. 2.
6. *Rename*( ρ) Use new name for the Tables or fields.

Additional operations:

7. Intersection (∩), *Join*(⋈), Division(÷): Not essential, but (very!) useful.

Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

**Projection**

- Deletes attributes that are not in projection list.
- Schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation. ( Unary Operation)
- Projection operator has to eliminate duplicates! (as it returns a relation which is a set)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Duplicate values may be representing different real world entity or relationship).

    Example: Consider the BOOK table:

| **Acc-No** | **Title** | **Author** |
|---|---|---|
| 100 | "DBMS" | "Silbershatz" |
| 200 | "DBMS" | "Ramanuj" |
| 300 | "COMPILER" | "Silbershatz" |
| 400 | "COMPILER" | "Ullman" |
| 500 | "OS" | "Sudarshan" |
| 600 | "DBMS" | "Silbershatz" |

$\pi_{\text{Title}}$(BOOK) =

| **Title** |
|---|
| "DBMS" |
| "COMPILER" |
| "OS" |

**Selection**

- Selects rows that satisfy *selection condition*.
- No duplicates in result
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

Example: For the example given above:

$\sigma_{\text{Acc-no}>300}$(BOOK) =

| **Acc-No** | **Title** | **Author** |
|---|---|---|
| 400 | "COMPILER" | "Ullman" |
| 500 | "OS" | "Sudarshan" |

$\sigma_{Title=\text{"DBMS"}}$(BOOK)=

| Acc-No | Title | Author |
|---|---|---|
| 100 | “DBMS” | “Silbershatz” |
| 200 | “DBMS” | “Ramanuj” |
| 600 | “DBMS” | “Silbershatz” |

$\pi_{Acc\text{-}no}$ ($\sigma_{Title=\text{"DBMS"}}$ (BOOK))=

| Acc-No |
|---|
| 100 |
| 200 |
| 600 |

**Union, Intersection, Set-Difference**

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - Corresponding’ fields have the same type.
- What is the *schema* of result?

  Consider:

Borrower

| Cust-name | Loan-no |
|---|---|
| Ram | L-13 |
| Shyam | L-30 |
| Suleman | L-42 |

Depositor

| Cust-name | Acc-no |
|---|---|
| Suleman | A-100 |
| Radheshyam | A-300 |
| Ram | A-401 |

List of customers who are either borrower or depositor at bank= $\pi_{Cust\text{-}name}$ (Borrower) U $\pi_{Cust\text{-}name}$ (Depositor)=

| Cust-name |
|---|
| Ram |
| Shyam |
| Suleman |
| Radeshyam |

Customers who are both borrowers and depositors = $\pi_{Cust\text{-}name}$ (Borrower) ∩ $\pi_{Cust\text{-}name}$ (Depositor)=

| Cust-name |
|---|
| Ram |
| Suleman |

Customers who are borrowers but not depositors = $\pi_{Cust\text{-}name}$ (Borrower) □ $\pi_{Cust\text{-}name}$ (Depositor)=

| Cust-name |
|---|
| Shyam |

**Cartesian-Product or Cross-Product** (S1 × R1)

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
- Consider the borrower and loan tables as follows:

Borrower:

| Cust-name | Loan-no |
|---|---|
| Ram | L-13 |
| Shyam | L-30 |
| Suleman | L-42 |

Loan:

| Loan-no | Amount |
|---|---|
| L-13 | 1000 |
| L-30 | 20000 |
| L-42 | 40000 |

Cross product of Borrower and Loan, Borrower × Loan =

| Borrower.Cust-name | Borrower.Loan-no | Loan.Loan-no | Loan.Amount |
|---|---|---|---|
| Ram | L-13 | L-13 | 1000 |
| Ram | L-13 | L-30 | 20000 |
| Ram | L-13 | L-42 | 40000 |
| Shyam | L-30 | L-13 | 1000 |
| Shyam | L-30 | L-30 | 20000 |
| Shyam | L-30 | L-42 | 40000 |
| Suleman | L-42 | L-13 | 1000 |
| Suleman | L-42 | L-30 | 20000 |
| Suleman | L-42 | L-42 | 40000 |

The rename operation can be used to rename the fields to avoid confusion when two field names are same in two participating tables:

For example the statement, $\rho_{\text{Loan-borrower(Cust-name,Loan-No-1, Loan-No-2,Amount)}}$( Borrower × Loan) results into- A new Table named Loan-borrower is created where it has four fields which are renamed as Cust-name, Loan-No-1, Loan-No-2 and Amount and the rows contains the same data as the cross product of Borrower and Loan.

Loan-borrower:

| Cust-name | Loan-No-1 | Loan-No-2 | Amount |
|---|---|---|---|
| Ram | L-13 | L-13 | 1000 |
| Ram | L-13 | L-30 | 20000 |
| Ram | L-13 | L-42 | 40000 |
| Shyam | L-30 | L-13 | 1000 |
| Shyam | L-30 | L-30 | 20000 |
| Shyam | L-30 | L-42 | 40000 |
| Suleman | L-42 | L-13 | 1000 |
| Suleman | L-42 | L-30 | 20000 |
| Suleman | L-42 | L-42 | 40000 |

**Rename Operation:**

It can be used in two ways :

- $\rho_x(E)$ return the result of expression E in the table named *x*.
- $\rho_{x(A_1, A_2, \ldots, A_n)}(E)$ return the result of expression E in the table named *x* with the attributes renamed to $A_1, A_2, \ldots, A_n$.
- It's benefit can be understood by the solution of the query " Find the largest account balance in the bank"

It can be solved by following steps:

- Find out the relation of those balances which are not largest.
- Consider Cartesion product of Account with itself i.e. Account × Account
- Compare the balances of first Account table with balances of second Account table in the product.
- For that we should rename one of the account table by some other name to avoid the confusion

It can be done by following operation

$\Pi_{Account.balance}\ (\sigma_{Account.balance\ <\ d.balance}(Account \times \rho_d(Account))$

- So the above relation contains the balances which are not largest.
- Subtract this relation from the relation containing all the balances i.e . $\Pi_{balance}\ (Account)$.

So the final statement for solving above query is

$\Pi_{balance}\ (Account)- \Pi_{Account.balance}\ (\sigma_{Account.balance\ <\ d.balance}(Account \times \rho_d(Account))$

# **LECTURE-20**

## Additional Operations

**Natural Join** ($S_1 \bowtie R_1$)

- Forms Cartesian product of its two arguments, performs selection forcing equality on those attributes that appear in both relations
- For example consider Borrower and Loan relations, the natural join between them $\text{Borrower} \bowtie \text{Loan}$ will automatically perform the selection on the table returned by Borrower × Loan which force equality on the attribute that appear in both Borrower and Loan i.e. Loan-no and also will have only one of the column named Loan-No.
- That means $\text{Borrower} \bowtie \text{Loan} = \sigma_{Borrower.Loan\text{-}no\ =\ Loan.Loan\text{-}no}$ (Borrower × Loan).
- The table returned from this will be as follows:

Eliminate rows that does not satisfy the selection criteria "$\sigma_{Borrower.Loan\text{-}no\ =\ Loan.Loan\text{-}no}$" from Borrower × Loan =

| **Borrower.Cust-name** | **Borrower.Loan-no** | **Loan.Loan-no** | **Loan.Amount** |
|---|---|---|---|
| Ram | L-13 | L-13 | 1000 |
| ~~Ram~~ | ~~L-13~~ | ~~L-30~~ | ~~20000~~ |
| ~~Ram~~ | ~~L-13~~ | ~~L-42~~ | ~~40000~~ |
| ~~Shyam~~ | ~~L-30~~ | ~~L-13~~ | ~~1000~~ |
| Shyam | L-30 | L-30 | 20000 |
| ~~Shyam~~ | ~~L-30~~ | ~~L-42~~ | ~~40000~~ |
| ~~Suleman~~ | ~~L-42~~ | ~~L-13~~ | ~~1000~~ |
| ~~Suleman~~ | ~~L-42~~ | ~~L-30~~ | ~~20000~~ |
| Suleman | L-42 | L-42 | 40000 |

And will remove one of the column named Loan-no.

- i.e. $\text{Borrower} \bowtie \text{Loan} =$

| **Cust-name** | **Loan-no** | **Amount** |
|---|---|---|
| Ram | L-13 | 1000 |
| Shyam | L-30 | 20000 |
| Suleman | L-42 | 40000 |

**Division Operation:**

- denoted by ÷ is used for queries that include the phrase "for all".
- For example "Find customers who has an account in all branches in branch city Agra". This query can be solved by following statement.

$$\Pi_{Customer\text{-}name.\ branch\text{-}name}\ (\text{Depositor} \bowtie \textit{Account}\ ) \div \Pi_{branch\text{-}name}\ (\sigma_{Branch\text{-}city=''Agra''}(Branch)$$

- The division operations can be specified by using only basic operations as follows: Let r(R) and s(S) be given relations for schema R and S with $S \subseteq R$

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

**LECTURE-21**

## Tuple Relational Calculus

Relational algebra is an example of procedural language while tuple relational calculus is a nonprocedural query language.
A query is specified as:
{t | P(t)}, i.e it is the set of all tuples t such that predicate P is true for t.

The formula P(t) is formed using atoms which uses the relations, tuples of relations and fields of tuples and following symbols
$\in$( belongs to),$<$,$>$,$\leq$,$\geq$,$\neq$,$=$, (comparison operators)

These atoms can then be used to form formulas with following symbols
$\forall$ ( universal qualifier generally called "for all")
$\exists$ ( existential qualifier generally called "there exists")
$\wedge$ ( and),$\vee$ (or),$\neg$( not)

For example : here are some queries and a way to express them using tuple calculus:

- Find the branch-name, loan-number and amount for loans over Rs 1200.
  $\{t \mid t \in \text{Loan} \wedge t[\text{amount}] > 1200\}.$

- Find the loan number for each loan of an amount greater that Rs1200.
  $\{t \mid \exists\, s \in \text{Loan}(t[\text{loan-number}] = s[\text{loan-number}] \wedge s[\text{amount}] > 1200\}$

- Find the names of all the customers who have a loan from the Sadar branch.
  $$\{t \mid \exists\, s \in \text{Borrower} \; ( t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists\, u \in \text{Loan} \; ( u[\text{loan-number}] = s[\text{loan-number} \wedge u[\text{branch-name}] = \text{"Sadar"}))\}$$

- Find all customers who have a loan , an account, or both at the bank
  $$\{t \mid \exists\, s \in \text{Borrower} \; ( t[\text{customer-name}] = s[\text{customer-name}]) \vee \exists\, u \in \text{Depositor} \; (t[\text{customer-name}] = u[\text{customer-name}])\}$$

- Find only those customers who have both an account and a loan.
  $$\{t \mid \exists\, s \in \text{Borrower} \; ( t[\text{customer-name}] = s[\text{customer-name}]) \wedge \exists\, u \in \text{Depositor} \; (t[\text{customer-name}] = u[\text{customer-name}])\}$$

- Find all customers who have an account but do not have loan.
  $$\{t \mid \exists u \in \text{Depositor} \; (t[\text{customer-name}] = u[\text{customer-name}]) \wedge \neg \exists\, s \in \text{Borrower} \; ( t[\text{customer-name}] = s[\text{customer-name}])\}$$

- Find all customers who have an account at all branches located in Agra
  $$\{t \mid \forall\, w \in \text{Branch}( w[\text{branch-city}] = \text{"Agra"} \Rightarrow \exists\, s \in \text{Depositor} \; ( t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists\, u \in \text{Account} \; ( u[\text{account-number}] = s[\text{account-number}] \wedge u[\text{branch-name}] = w[\text{branch-name}])))\}$$

## Domain Relational Calculus

1. Domain relational calculus is another non procedural language for expressing database queries.
2. A query is specified as:
   $\{<x_1,x_2,\ldots,x_n> \mid P(x_1,x_2,\ldots,x_n)\}$ where $x_1,x_2,\ldots,x_n$ represents domain variables. P represent a predicate formula as in tuple calculus

- Since the domain variables are referred in place of tuples the formula doesn't refer the fields of tuples rather they refer the domain variables.
- For example the queries in domain calculus are mentioned as follows:
  - Find the branch-name, loan-number and amount for loans over Rs 1200.
    $\{<b, l, a> \mid <b, l, a> \in \text{Loan} \wedge a > 1200\}$.
  - Find the loan number for each loan of an amount greater that Rs1200.
    $\{< l > \mid \exists\, b,a( <b, l, a> \in \text{Loan} \wedge a >1200\}$
  - Find the names of all the customers who have a loan from the Sadar branch and find the loan amount
    $\{< c, a > \mid \exists\, l(< c, l > \in \text{Borrower}$
    $\exists\, b( < b, l, a > \in \text{Loan} \wedge b=\text{"Sadar"}))\}$
  - Find names of all customers who have a loan , an account, or both at the Sadar Branch
    $\{<c> \mid \exists\, l(< c, l > \in \text{Borrower} \wedge \exists\, b, a(<b, l, a> \in \text{Loan} \wedge b =\text{"Sadar"}))$
    $\vee \exists\, a(<c, a> \in \text{Depositor} \wedge \exists\, b, n(<b, a, n> \in \text{Account} \wedge b =\text{"Sadar"}))\}$
  - Find only those customers who have both an account and a loan.
    $\{<c> \mid \exists\, l(<c, l> \in \text{Borrower} ) \wedge \exists\, a(<c, a> \in \text{Depositor} )\}$
  - Find all customers who have an account but do not have loan.
    $\{t \mid \exists\, a(<c, a> \in \text{Depositor} ) \wedge \neg \exists\, l(<c, l> \in \text{Borrower} )\}$
  - Find all customers who have an account at all branches located in Agra
    $\{<c> \mid \forall\, x, y, z(<x, y, z> \in \text{Branch}) \wedge y = \text{"Agra"} \Rightarrow$

## Outer Join.

Outer join operation is an extension of join operation to deal with missing information

- Suppose that we have following relational schemas:
  Employee( employee-name, street, city)
  Fulltime-works(employee-name, branch-name, salary)
  A snapshot of these relations is as follows:

Employee:

| **employee-name** | **street** | **city** |
|---|---|---|
| Ram | M G Road | Agra |
| Shyam | New Mandi Road | Mathura |
| Suleman | Bhagat Singh Road | Aligarh |

Fulltime-works

| **employee-name** | **branch-name** | **salary** |
|---|---|---|
| Ram | Sadar | 30000 |
| Shyam | Sanjay Place | 20000 |
| Rehman | Dayalbagh | 40000 |

Suppose we want complete information of the full time employees.

- The natural join (Employee ⋈ Fulltime-works)will result into the loss of information for Suleman and Rehman because they don't have record in both the tables ( left and right relation). The outer join will solve the problem.
- Three forms of outer join:
    - **Left outer join(⟕):**the tuples which doesn't match while doing natural join from left relation are also added in the result putting null values in missing field of right relation.
    - **Right outer join(⟖):**the tuples which doesn't match while natural join from right relation are also added in the result putting null values in missing field of left relation.
    - **Full outer join(⟗):** include both of the left and right outer joins i.e. adds the tuples which did not match either in left relation or right relation and put null in place of missing values.
- The result for three forms of outer join are as follows:

**Left join:** Employee ⟕ Fulltime-works =

| **employee-name** | **street** | **City** | **branch-name** | **salary** |
|---|---|---|---|---|
| Ram | M G Road | Agra | Sadar | 30000 |
| Shyam | New Mandi Road | Mathura | Sanjay Place | 20000 |
| Suleman | Bhagat Singh Road | Aligarh | Null | Null |

**Right join:** Employee ⟖ Fulltime-works =

| **employee-name** | **street** | **city** | **branch-name** | **salary** |
|---|---|---|---|---|
| Ram | M G Road | Agra | Sadar | 30000 |
| Shyam | New Mandi Road | Mathura | Sanjay Place | 20000 |
| Rehman | null | null | Dayalbagh | 40000 |

**Full join:** Employee ⟗ Fulltime-works =

| **employee-name** | **street** | **city** | **branch-name** | **salary** |
|---|---|---|---|---|
| Ram | M G Road | Agra | Sadar | 30000 |
| Shyam | New Mandi Road | Mathura | Sanjay Place | 20000 |
| Suleman | Bhagat Singh Road | Aligarh | null | null |
| Rehman | null | null | Dayalbagh | 40000 |

## LECTURE-22

# Structured Query Language (SQL)

## Introduction

□ Commercial database systems use more user friendly language to specify the queries.

□ SQL is the most influential commercially marketed product language.

□ Other commercially used languages are QBE, Quel, and Datalog.

## Basic Structure

- The basic structure of an SQL consists of three clauses: **select, from** and **where.**
- **select:** it corresponds to the projection operation of relational algebra. Used to list the attributes desired in the result.
- **from**: corresponds to the Cartesian product operation of relational algebra. Used to list the relations to be scanned in the evaluation of the expression
- **where**: corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.
- A typical SQL query has the form:

  **select** $A_1, A_2, \ldots, A_n$
  **from** $r_1, r_2, \ldots, r_m$
  **where** P

  - $A_i$ represents an attribute
  - $r_j$ represents a relation
  - P is a predicate
  - It is equivalent to following relational algebra expression:
  - $\Pi_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$

*[**Note:** The words marked in dark in this text work as keywords in SQL language. For example "select", "from" and "where" in the above paragraph are shown in bold font to indicate that they are keywords]*

## Select Clause

Let us see some simple queries and use of **select** clause to express them in SQL.

- Find the names of all branches in the Loan relation

  **select** branch-name
  **from** Loan

- By default the **select** clause includes duplicate values. If we want to force the elimination of duplicates the **distinct** keyword is used as follows:

  **select distinct** branch-name
  **from** Loan

- The **all** key word can be used to specify *explicitly* that duplicates are not removed. Even if we not use **all** it means the same so we don't require **all** to use in **select** clause.

  **select all** branch-name
  **from** Loan

- The asterisk "*" can be used to denote "all attributes". The following SQL statement will select and all the attributes of Loan.

  **select** *
  **from** Loan

- The arithmetic expressions involving operators, +, -, *, and / are also allowed in **select** clause. The following statement will return the amount multiplied by 100 for the rows in Loan table.

**select** branch-name, loan-number, amount * 10 **from** Loan.

## Where Clause

- Find all loan numbers for loans made at "Sadar" branch with loan amounts greater than Rs 1200.

```
select loan-number
from Loan
where branch-name= "Sadar" and amount  > 1200
```

- **where** clause uses uses logival connectives **and**, **or**, and **not**
- operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and < >.
- **between** can be used to simplify the comparisons

```
select loan-number
from Loan
where  amount between 90000 and 100000
```

## From Clause

- The **from** clause by itself defines a Cartesian product of the relations in the clause.
- When an attribute is present in more than one relation they can be referred as *relation-name.attribute-name* to avoid the ambiguity.
- For all customers who have loan from the bank, find their names and loan numbers

```
select distinct  customer-name,  Borrower.loan-number
from Borrower, Loan
where Borrower.loan-number = Loan.loan-number
```

## The Rename Operation

- Used for renaming both relations both relations and attributes in SQL
- Use **as** clause: old-name **as** new-name
- Find the names and loan numbers of the customers who have a loan at the "Sadar" branch.

```
select distinct customer-name, borrower.loan-number as loan-id
from Borrower, Loan
where Borrower.loan-number = Loan.loan-number and
        branch-name = "Sadar"
```

  we can now refer the loan-number instead by the name loan-id.
- For all customers who have a loan from the bank, find their names and loan-numbers.

```
select distinct customer-name, T.loan-number
from Borrower as T, Loan as S
where T.loan-number = S.loan-number
```

- Find the names of all branches that have assets greater than at least one branch located in "Mathura".

```
select distinct T.branch-name
from  branch as T, branch as S
where T.assets > S.assets and S.branch-city = "Mathura"
```

## String Operation

- Two special characters are used for pattern matching in strings:
  - Percent ( % ) : The % character matches any substring
  - Underscore( _ ): The _ character matches any character
- "%Mandi": will match with the strings ending with "Mandi" viz. "Raja Ki mandi", "Peepal Mandi"
- "_ _ _" matches any string of three characters.
- Find the names of all customers whose street address includes the substring "Main"

```
select customer-name
from  Customer
where customer-street like “%Main%”
```

## Set Operations

- **union, intersect and except** operations are set operations available in SQL.
- Relations participating in any of the set operation must be compatible; i.e. they must have the same set of attributes.
- Union Operation:
  - Find all customers having a loan, an account, or both at the bank

    ```
    (select customer-name   from Depositor )
     union
     (select  customer-name    from Borrower )
    ```

    It will automatically eliminate duplicates.
  - If we want to retain duplicates **union all** can be used

    ```
    (select customer-name   from Depositor )
     union all
     (select  customer-name from Borrower )
    ```
- Intersect Operation
  - Find all customers who have both an account and a loan at the bank

    ```
    (select customer-name   from Depositor )
     intersect
     (select  customer-name    from Borrower )
    ```
  - If we want to retail all the duplicates

    ```
    (select customer-name from Depositor )
     intersect all
     (select  customer-name from Borrower )
    ```
- Except Opeartion
  - Find all customers who have an account but no loan at the bank

    ```
    (select customer-name from Depositor )
     except
     (select  customer-name from Borrower )
    ```
  - If we want to retain the duplicates:

    ```
    (select customer-name  from Depositor )
     except all
     (select  customer-name from Borrower )
    ```

## Aggregate Functions

- Aggregate functions are those functions which take a collection of values as input and return a single value.
- SQL offers 5 built in aggregate functions-
  - Average: **avg**
  - Minimum:**min**
  - Maximum:**max**
  - Total: **sum**
  - Count:**count**
- The input to **sum** and **avg** must be a collection of numbers but others may have collections of non-numeric data types as input as well
- Find the average account balance at the Sadar branch

  ```
  select avg(balance)
  ```

```
from  Account
where branch-name= “Sadar”
```

The result will be a table which contains single cell (one row and one column) having numerical value corresponding to average balance of all account at sadar branch.

- **group by** clause is used to form groups, tuples with the same value on all attributes in the **group by** clause are placed in one group.
- Find the average account balance at each branch

  ```
  select branch-name, avg(balance)
  from  Account
  group by branch-name
  ```

- By default the aggregate functions include the duplicates.
- **distinct** keyword is used to eliminate duplicates in an aggregate functions:
- Find the number of depositors for each branch

  ```
  select branch-name, count(distinct customer-name)
  from  Depositor, Account
  where Depositor.account-number = Account.account-number
  group by branch-name
  ```

- **having** clause is used to state condition that applies to groups rather than tuples.
- Find the average account balance at each branch where average account balance is more than Rs. 1200

  ```
  select branch-name, avg(balance)
  from  Account
  group by branch-name
  having avg(balance) > 1200
  ```

- Count the number of tuples in Customer table

  ```
  select count(*)
  from  Customer
  ```

- SQL doesn’t allow **distinct** with **count**(*)
- When **where** and **having** are both present in a statement **where** is applied before **having**.

**LECTURE-23**

## Nested Sub queries

A subquery is a **select-from-where** expression that is nested within another query.

**Set Membership**

The **in** and **not in** connectives are used for this type of subquery.

"Find all customers who have both a loan and an account at the bank", this query can be written using nested subquery form as follows

```
select distinct  customer-name
from Borrower
where customer-name in(select customer-name
                              from Depositor )
```

- Select the names of customers who have a loan at the bank, and whose names are neither "Smith" nor "Jones"

```
select distinct  customer-name
from Borrower
where customer-name not in("Smith", "Jones")
```

□ **Set Comparison**

Find the names of all branches that have assets greater than those of at least one branch located in Mathura

```
select branch-name
from Branch
where asstets > some (select assets
                             from Branch
                             where branch-city = "Mathura" )
```

1. Apart from > **some** others comparison could be < **some** , <= **some** , >= **some** , = **some** , < > **some.**
2. Find the names of all branches that have assets greater than that of each branch located in Mathura

```
select branch-name
from Branch
where asstets > all (select assets
                           from Branch
                           where branch-city = "Mathura" )
```

- Apart from > **all** others comparison could be < **all** , <= **all** , >= **all** , = **all** , < >**all.**

## Views

In SQL **create view** command is used to define a view as follows:

```
create view v as <query expression>
```

where <query expression> is any legal query expression and v is the view name.

- The view consisting of branch names and the names of customers who have either an account or a loan at the branch. This can be defined as follows:

```
create view All-customer as
(select branch-name, customer-name
from Depositor, Account
where Depositor.account-number=account.account-number)
union
```

(**select** branch-name, customer-name
**from** Borrower, Loan
**where** Borrower.loan-number = Loan.loan-number)

- The attributes names may be specified explicitly within a set of round bracket after the name of view.
- The view names may be used as relations in subsequent queries. Using the view Allcustomer

Find all customers of Sadar branch
**select** customer-name
**from** All-customer
**where** branch-name= "Sadar"

- A **create-view** clause creates a view definition in the database which stays until a command - **drop view** *view-name* - is executed.

## Modification of Database

### Deletion

- In SQL we can delete only whole tuple and not the values on any particular

attributes. The command is as follows:

**delete from** r **where** P.
where P is a predicate and r is a relation.

- **delete** command operates on only one relation at a time. Examples are as follows:
- Delete all tuples from the Loan relation

**delete from** Loan

- Delete all of the Smith's account record

**delete from** Depositor
**where** customer-name = "Smith"

- Delete all loans with loan amounts between Rs 1300 and Rs 1500.
  **delete from** Loan
  **where** amount **between** 1300 **and** 1500
- Delete the records of all accounts with balances below the average at the bank

**delete from** Account
**where** balance < ( **select avg**(balance)
**from** Account)

### Insertion

In SQL we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Examples are as follows:
Insert an account of account number A-9732 at the Sadar branch having balance of Rs 1200
**insert into** Account
**values**("Sadar", "A-9732", 1200)
the values are specified in the order in which the corresponding attributes are listed in the relation schema.
SQL allows the attributes to be specified as part of the **insert** statement
**insert into** Account(account-number, branch-name, balance)
**values**("A-9732", "Sadar", 1200)
**insert into** Account(branch-name, account-number, balance)

```
values(“Sadar”, “A-9732”, 1200)
```

Provide for all loan customers of the Sadar branch a new Rs 200 saving account for each loan account they have. Where loan-number serve as the account number for these accounts.

```
insert into Account
select branch-name, loan-number, 200
from Loan
where branch-name = “Sadar”
```

## □Updates

Used to change a value in a tuple without changing all values in the tuple.

Suppose that annual interest payments are being made, and all balances are to be increased by 5 percent.

```
update Account
set balance = balance * 1.05
```

Suppose that accounts with balances over Rs10000 receive 6 percent interest, whereas all others receive 5 percent.

```
update Account
set balance = balance * 1.06
where balance > 10000
update Account
set balance = balance * 1.05
        where balance <= 10000
```

## Data Definition Language

### Data Types in SQL

**char**(n): fixed length character string, length n.
**varchar**(n): variable length character string, maximum length n.
**int**: an integer.
**smallint**: a small integer.
**numeric**(p,d): fixed point number, p digits( plus a sign), and d of the p digits are to right of the decimal point.
**real, double precision**: floating point and double precision numbers.
**float**(n): a floating point number, precision at least n digits.
**date**: calendar date; four digits for year, two for month and two for day of month.
**time**: time of day n hours minutes and seconds.

**Domains can be defined as**

**create domain** person-name **char**(20).

the domain name person-name can be used to define the type of an attribute just like built-in domain.

**Schema Definition in SQL**

**create table command** is used to define relations.

```
create table r (A1D1, A2D2,… , AnDn,
<integrity constraint1>,
… ,
<integrity constraintk>)
```

where r is relation name, each Ai is the name of attribute, Di is the domain type of values of Ai. Several types of integrity constraints are available to define in SQL.

**Integrity Constraints** which are allowed in SQL are

**primary key**(Aj1, Aj2,… , Ajm)
and
**check**(P) where P is the predicate.

**drop table** command is used to remove relations from database.
**alter table** command is used to add attributes to an existing relation
**alter table** r **add** A D
it will add attribute A of domain type D in relation r.
**alter table** r **drop** A
it will remove the attribute A of relation r.

<u>**LECTURE-24**</u>

## Integrity Constraints

- Integrity Constraints guard against accidental damage to the database.
- Integrity constraints are predicates pertaining to the database.
- **Domain Constraints:**
- Predicates defined on the domains are Domain constraints.
- Simplest Domain constraints are defined by defining standard data types of the attributes like Integer, Double, Float, etc.
- We can define domains by **create domain** clause also we can define the constraints on such domains as follows:

  **create domain** hourly-wage **numeric**(5,2)
  **constraint** wage-value-test check(value >= 4.00)
- So we can use hourly-wage as data type for any attribute where DBMS will automatically allow only values greater than or equal to 4.00.
- Other examples for defining Domain constraints are as follows:
  **create domain** account-number **char**(10)
  **constraint** account-number-null-test **check(value not null)**
  **create domain** account-type **char**(10)
  **constraint** account-type-test
  **check (value in** ( “Checking”, “Saving”))

By using the later domain of two above the DBMS will allow only values for any attribute having type as account-type i.e. Checking and Saving.

- **Referential Integrity:**
- **Foreign Key**: If two table R and S are related to each other, K1 and K2 are primary keys of the two relations also K1 is one of the attribute in S. Suppose we want that every row in S must have a corresponding row in R, then we define the K1 in S as **foreign key**. Example in our original database of library we had a table for relation BORROWEDBY, containing two fields Card No. and Acc. No. . Every row of BORROWEDBY relation must have corresponding row in USER Table having same Card No. and a row in BOOK table having same Acc. No.. Then we will define the Card No. and Acc. No. in BORROWEDBY relation as foreign keys.
- In other way we can say that every row of BORROWEDBY relation must refer to some row in BOOK and also in USER tables.
- Such referential requirement in one table to another table is called Referential Integrity.

# LECTURE-25

## Query by Example (QBE)

Query by Example (QBE) is a method of query creation that allows the user to search for documents based on an example in the form of a selected text string or in the form of a document name or a list of documents. Because the QBE system formulates the actual query, QBE is easier to learn than formal query languages, such as the standard Structured Query Language (SQL), while still enabling powerful searches.

## Selections in QBE

QBE uses skeleton tables to represent table name and fieldnames like:

| Table name | Field1 | Field2 | ….. |
|---|---|---|---|
| | | | |

For selection, P operator along with variable name/constant name is used to display one or more fields.

**Example 1:**

Consider the relation: student (name, roll, marks)

The following query can be represented as:

SQL: select name from student where marks>50;

| student | name | roll | marks |
|---|---|---|---|
| | P.X | | >50 |

Here X is a constant; alternatively we can use _X as a variable.

**Example 2:**

For the relation given above

The following query can be represented as:

SQL: select * from student where marks>50 and marks <80;

| student | name | roll | marks |
|---|---|---|---|
| P. | | | _X |

| CONDITION |
|---|
| _X>50 ∧ _X<80 |

We can use condition box to represent complex conditions.

**Example 3:**

For the relation given above

The following query can be represented as:

SQL: select name, roll from student where marks<50 or marks >80;

| student | name | roll | marks |
|---|---|---|---|
| | P.A | P.B | <50 |
| | P.A | P.B | >80 |

OR operation retrieves results in multiple rows.

**Example 4:**

(Joins in QBE)

Consider the following tables:

Student (name, roll, marks)

Grades (roll, grade)

The following query can be represented as:

SQL: select s.name, g.grade from Student s, Grades g where s.roll=g.roll and s.marks>50;
Uses two skeleton tables:

| Student | name | roll | marks |
|---|---|---|---|
| | P.A | _X | >50 |

And

| Grades | roll | grade |
|---|---|---|
| | _X | P.B |

**Insertions in QBE:**

Uses operator I. on the table.

Example: Consider the following query on Student table

SQL: insert into student values ('abc',10,60);

| Student | name | roll | marks |
|---|---|---|---|
| I. | abc | 10 | 60 |

Multiple insertions can be represented by separate rows in skeleton table.

**Deletions in QBE:**

Uses operator D. on the table.

Example: Consider the following query on Student table

SQL: delete from student where marks=0;

| Student | name | roll | marks |
|---|---|---|---|
| D. | | | 0 |

Multiple deletions can be represented by separate rows in skeleton table.

Deletions without any condition can truncate the entire table.

**Updation in QBE:**

Uses operator U. on the table.

Example: Consider the following query on Student table

SQL: update student set mark=50 where roll=40;

| Student | name | roll | marks |
|---|---|---|---|
| U. | | | 50 |
| | | 40 | |

# LECTURE-26

## RELATIONAL DATABASE DEGIN

Database design is a process in which you create a logical data model for a database, which store data of a company. It is performed after initial database study phase in the database life cycle. You use normalization technique to create the logical data model for a database and eliminate data redundancy.

Normalization also allows you to organize data efficiently in a database and reduce anomalies during data operation. Various normal forms, such as first, second and third can be applied to create a logical data model for a database. The second and third normal forms are based on partial dependency and transitivity dependency. Partial dependency occurs when a row of table is uniquely identified by one column that is a part of a primary key. A transitivity dependency occurs when a non key column is uniquely identified by values in another non-key column of a table.

### Database Design Process:

We can identify six main phases of the database design process:

1. Requirement collection and analysis
2. Conceptual database design
3. Choice of a DBMS
4. Data model mapping(logical database design)
5. Physical database design
6. Database system implementation and tuning

1. **Requirement Collection and Analysis**

   Before we can effectively design a data base we must know and analyze the expectation of the users and the intended uses of the database in as much as detail.

2. **Conceptual Database Design**

   The goal for this phase is to produce a conceptual schema for the database that is independent of a specific DBMS.

   - We often use a high level data model such ER-model during this phase
   - We specify as many of known database application on transactions as possible using a notation that is independent of any specific dbms.
   - Often the dbms choice is already made for the organization the intent of conceptual design still to keep , it as free as possible from implementation consideration.

3. **Choice of a DBMS**

   The choice of dbms is governed by a no. of factors some technical other economic and still other concerned with the politics of the organization.

   The economics and organizational factors that offer the choice of the dbms are:

   Software cost, maintenance cost, hardware cost, database creation and conversion cost, personnel cost, training cost, operating cost.

4. **Data model mapping (logical database design)**

   During this phase, we map the conceptual schema from the high level data model used on phase 2 into a data model of the choice dbms.

5. **Physical databse design**
   During this phase we design the specification for the database in terms of physical storage structure ,record placement and indexes.
6. **Database system implementation and tuning**
   During this phase, the database and application programs are implemented, tested and eventually deployed for service.

## Informal Guidelines for Relation Design

Want to keep the semantics of the relation attributes clear.  The information in a tuple should represent exactly one fact or an entity.  The hidden or buried entities are what we want to discover and eliminate.

- Design a relation schema so that it is easy to explain its meaning.
- Do not combine attributes from multiple entity types and relationship types into a single relation. Use a view if you want to present a simpler layout to the end user.
- A relation schema should correspond to on entity type or relationship type.
- Minimize redundant information in tuples, thus reducing update anomalies
- If anomalies are present, try to decompose the relation into two or more to represent the separate facts, or document the anomalies well for management in the applications programs.

Minimize the use of **null** values. Nulls have multiple interpretations:

- The attribute does not apply to this tuple
- The attribute value is unknown
- The attribute value is absent
- The attribute value might represent an actual value

If nulls are likely (non-applicable) then consider decomposition of the relation into two or more relations that hold only the non-null valued tuples.

- Do not permit the creation of spurious tuples

Too much decomposition of relations into smaller ones may also lose information or generate erroneous information

- Be sure that relations can be logically joined using natural join and the result doesn't generate relationships that don't exist

## Functional Dependencies

FD's are constraints on well-formed relations and represent a formalism on the infrastructure of relation.

**Definition:** A *functional dependency* (FD) on a relation schema **R** is a constraint $X \rightarrow Y$, where $X$ and $Y$ are subsets of attributes of **R.**

**Definition**: an FD is a relationship between an attribute "Y" and a determinant (1 or more other attributes) "X" such that for a given value of a determinant the value of the attribute is uniquely defined.

- X is a determinant
- X determines Y
- Y is functionally dependent on X
- X → Y
- X →Y is trivial if Y ⊆ X

**Definition**: An FD ***X* → *Y*** is *satisfied* in an instance **r** of **R** if for every pair of tuples, *t* and s: if *t* and *s* agree on all attributes in *X* then they must agree on all attributes in *Y*

A key constraint is a special kind of functional dependency: all attributes of relation occur on the right-hand side of the FD:

- *SSN → SSN, Name, Address*

**Example Functional Dependencies**

Let R be
**NewStudent**(***stuId, lastName, major, credits, status, socSecNo***)

FDs in R include

- ***{stuId}→{lastName}***, but not the reverse
- ***{stuId} →{lastName, major, credits, status, socSecNo, stuId}***
- ***{socSecNo} →{stuId, lastName, major, credits, status, socSecNo}***
- ***{credits}→{status},*** but not ***{status}→{credits}***

*ZipCode→AddressCity*

- 16652 is Huntingdon’s ZIP

*ArtistName→BirthYear*

- Picasso was born in 1881

*Autobrand→Manufacturer, Engine type*

- Pontiac is built by General Motors with gasoline engine

*Author, Title→PublDate*

- Shakespeare’s Hamlet was published in 1600

## Trivial Functional Dependency

The FD X→Y is ***trivial*** if set {Y} is a subset of set {X}

Examples: If A and B are attributes of R,

- {A}→{A}
- {A,B} →{A}
- {A,B} →{B}
- {A,B} →{A,B}

are all trivial FDs and will not contribute to the evaluation of normalization.

**FD Axioms**

Understanding: Functional Dependencies are recognized by analysis of the real world; no automation or algorithm. Finding or recognizing them are the database designer's task.

FD manipulations:

- **Soundness** -- no incorrect FD's are generated
- **Completeness** -- all FD's can be generated

| Axiom Name | Axiom | Example |
|---|---|---|
| **Reflexivity** | if a is set of attributes, b ⊆ a, then a →b | *SSN,Name → SSN* |
| **Augmentation** | if a→ b holds and c is a set of attributes, then ca→cb | *SSN → Name* then<br>*SSN,Phone → Name, Phone* |
| **Transitivity** | if a →b holds and b→c holds, then a→ c holds | *SSN →Zip* and *Zip → City* then *SSN →City* |
| **Union or Additivity *** | if a → b and a → c holds then a→ bc holds | *SSN→Name* and *SSN→Zip* then *SSN→Name,Zip* |
| **Decomposition or Projectivity*** | if a → bc holds then a → b and a → c holds | *SSN→Name,Zip* then *SSN→Name* and *SSN→Zip* |
| **Pseudotransitivity*** | if a → b and cb → d hold then ac → d holds | *Address → Project* and *Project,Date →Amount* then *Address,Date → Amount* |
| **(NOTE)** | *ab→ c does NOT imply a → c and b → c* | |

*Armstrong's Axioms (basic axioms)

# LECTURE-27

## CLOSURE OF A SET OF FUNCTIONAL DEPEDENCIES

Given a relational schema R, a functional dependencies f on R is logically implied by a set of functional dependencies F on R if every relation instance r(R) that satisfies F also satisfies f.

The closure of F, denoted by $F^+$, is the set of all functional dependencies logically implied by F.
The closure of F can be found by using a collection of rules called **Armstrong axioms.**
**Reflexivity rule:** If A *is a set of attributes and B is subset or equal to A, then A*→B holds.
**Augmentation rule:** If A→B holds and C is a set of attributes, then CA→CB holds
**Transitivity rule:** If A→B holds and B→C holds, then A→C holds.
**Union rule:** If A→B holds and A→C then A→BC holds
**Decomposition rule:** If A→BC holds, then A→B holds and A→C holds.
**Pseudo transitivity rule:** If A→B holds and BC→D holds, then AC→D holds.

Suppose we are given a relation schema R=(A,B,C,G,H,I) and the set of function dependencies {A→B,A→C,CG→H,CG→I,B→H}

We list several members of $F^+$ here:

1. A→H, since A→B and B→H hold, we apply the transitivity rule.
2. CG→HI. Since CG→H and CG→I , the union rule implies that CG→HI
3. AG→I, since A→C and CG→I, the pseudo transitivity rule implies that AG→I holds

**Algorithm of compute F+ :**

To compute the closure of a set of functional dependencies F:

```
   F+ = F
repeat
      for each functional dependency f in F+
          apply reflexivity and augmentation rules on f
          add the resulting functional dependencies to F+
      for each pair of functional dependencies f1and f2 in F+
          if f1 and f2 can be combined using transitivity
              then add the resulting functional dependency to F+
until F+ does not change any further
```

large.

# LECTURE-28

## LOSS LESS DECOMPOSITION

A decomposition of a relation scheme R<S,F> into the relation schemes Ri(1<=i<=n) is said to be a lossless join decomposition or simply lossless if for every relation R that satisfies the FDs in F, the natural join of the projections or R gives the original relation R, i.e,

$$R=\Box_{R1}(R)\bowtie\Box_{R2}(R)\bowtie\ldots\ldots\bowtie\Box_{Rn}(R)$$

If R is subset of $\Box_{R1}(R)\bowtie\Box_{R2}(R)\bowtie\ldots\ldots\bowtie\Box_{Rn}(R)$

Then the decomposition is called lossy.

## DEPEDENCY PRSERVATION:

Given a relation scheme R<S,F> where F is the associated set of functional dependencies on the attributes in S,R is decomposed into the relation schemes R1,R2,…Rn with the fds F1,F2…Fn, then this decomposition of R is dependency preserving if the closure of F' (where F'=F1 U F2 U … Fn)
Example:
Let R(A,B,C) AND F={A→B}. Then the decomposition of R into R1(A,B) and R2(A,C) is lossless because the FD { A→B} is contained in R1 and the common attribute A is a key of R1.

Example:
Let R(A,B,C) AND F={A→B}. Then the decomposition of R into R1(A,B) and R2(B,C) is not lossless because the common attribute B does not functionally determine either A or C. i.e, it is not a key of R1 or R 2.

Example:
Let R(A,B,C,D) and F={A→B, A→C, C→D,}. Then the decomposition of R into R1(A,B,C) with the FD F1={ A→B , A→C }and R2(C,D) with FD F2={ C→D} . In this decomposition all the original FDs can be logically derived from F1 and F2, hence the decomposition is dependency preserving also . the common attribute C forms a key of R2. The decomposition is lossless.

Example:
Let R(A,B,C,D) and F={A→B, A→C, A→D,}. Then the decomposition of R into R1(A,B,D) with the FD F1={ A→B , A→D }and R2(B,C) with FD F2={ } is lossy because the common attribute B is not a candidate key of either R1 and R2 .
In addition , the fds A→C is not implied by any fds R1 or R2. Thus the decomposition is not dependency preserving.

**Full functional dependency:**
Given a relational scheme R and an FD X→Y ,Y is fully functional dependent on X if there is no Z, where Z is a proper subset of X such that Z→Y. The dependency X→Y is left reduced, there being no extraneous attributes attributes in the left hand side of the dependency.

**Partial dependency:**
Given a relation dependencies F defined on the attributes of R and K as a candidate key ,if X is a proper subset of K and if F|= X→A, then A is said to be partial dependent on K

**Prime attribute and non prime attribute:**

A attribute A in a relation scheme R is a **prime attribute** or simply **prime** if A is part of any candidate key of the relation. If A is not a part of any candidate key of R, A is called a nonprime attribute or simply **non prime** .

**Trivial functional dependency:**
A FD X→Y is said to be a trivial functional dependency if Y is subset of X.

## LECTURE-29

### Normalization

While designing a database out of an entity–relationship model, the main problem existing in that "raw" database is redundancy. Redundancy is storing the same data item in more one place. A redundancy creates several problems like the following:

1. Extra storage space: storing the same data in many places takes large amount of disk space.
2. Entering same data more than once during data insertion.
3. Deleting data from more than one place during deletion.
4. Modifying data in more than one place.
5. Anomalies may occur in the database if insertion, deletion, modification etc are no done properly. It creates inconsistency and unreliability in the database.

To solve this problem, the "raw" database needs to be normalized. This is a step by step process of removing different kinds of redundancy and anomaly at each step. At each step a specific rule is followed to remove specific kind of impurity in order to give the database a slim and clean look.

### Un-Normalized Form (UNF)

If a table contains non-atomic values at each row, it is said to be in UNF. An **atomic value** is something that can not be further decomposed. A **non-atomic value**, as the name suggests, can be further decomposed and simplified. Consider the following table:

| Emp-Id | Emp-Name | Month | Sales | Bank-Id | Bank-Name |
|---|---|---|---|---|---|
| E01 | AA | Jan | 1000 | B01 | SBI |
| | | Feb | 1200 | | |
| | | Mar | 850 | | |
| E02 | BB | Jan | 2200 | B02 | UTI |
| | | Feb | 2500 | | |
| E03 | CC | Jan | 1700 | B01 | SBI |
| | | Feb | 1800 | | |
| | | Mar | 1850 | | |
| | | Apr | 1725 | | |

In the sample table above, there are multiple occurrences of rows under each key Emp-Id. Although considered to be the primary key, Emp-Id cannot give us the unique identification facility for any single row. Further, each primary key points to a variable length record (3 for E01, 2 for E02 and 4 for E03).

### First Normal Form (1NF)

A relation is said to be in 1NF if it contains no non-atomic values and each row can provide a unique combination of values. The above table in UNF can be processed to create the following table in 1NF.

| Emp-Id | Emp-Name | Month | Sales | Bank-Id | Bank-Name |
|---|---|---|---|---|---|
| E01 | AA | Jan | 1000 | B01 | SBI |
| E01 | AA | Feb | 1200 | B01 | SBI |
| E01 | AA | Mar | 850 | B01 | SBI |
| E02 | BB | Jan | 2200 | B02 | UTI |
| E02 | BB | Feb | 2500 | B02 | UTI |
| E03 | CC | Jan | 1700 | B01 | SBI |
| E03 | CC | Feb | 1800 | B01 | SBI |
| E03 | CC | Mar | 1850 | B01 | SBI |
| E03 | CC | Apr | 1725 | B01 | SBI |

As you can see now, each row contains unique combination of values. Unlike in UNF, this relation contains only atomic values, i.e. the rows can not be further decomposed, so the relation is now in 1NF.

**Second Normal Form (2NF)**

A relation is said to be in 2NF f if it is already in 1NF and each and every attribute fully depends on the primary key of the relation. Speaking inversely, if a table has some attributes which is not dependant on the primary key of that table, then it is not in 2NF.

Let us explain. Emp-Id is the primary key of the above relation. Emp-Name, Month, Sales and Bank-Name all depend upon Emp-Id. But the attribute Bank-Name depends on Bank-Id, which is not the primary key of the table. So the table is in 1NF, but not in 2NF. If this position can be removed into another related relation, it would come to 2NF.

| Emp-Id | Emp-Name | Month | Sales | Bank-Id |
|---|---|---|---|---|
| E01 | AA | JAN | 1000 | B01 |
| E01 | AA | FEB | 1200 | B01 |
| E01 | AA | MAR | 850 | B01 |
| E02 | BB | JAN | 2200 | B02 |
| E02 | BB | FEB | 2500 | B02 |
| E03 | CC | JAN | 1700 | B01 |
| E03 | CC | FEB | 1800 | B01 |
| E03 | CC | MAR | 1850 | B01 |
| E03 | CC | APR | 1726 | B01 |

| Bank-Id | Bank-Name |
|---|---|
| B01 | SBI |
| B02 | UTI |

After removing the portion into another relation we store lesser amount of data in two relations without any loss information. There is also a significant reduction in redundancy.

**Third Normal Form (3NF)**

A relation is said to be in 3NF, if it is already in 2NF and there exists no **transitive dependency** in that relation. Speaking inversely, if a table contains transitive dependency, then it is not in 3NF, and the table must be split to bring it into 3NF.

What is a transitive dependency? Within a relation if we see
A → B [B depends on A]
And
B → C [C depends on B]
Then we may derive
A → C[C depends on A]

Such derived dependencies hold well in most of the situations. For example if we have
Roll → Marks
And
Marks → Grade
Then we may safely derive
Roll → Grade.

This third dependency was not originally specified but we have derived it.

**The derived dependency is called a transitive dependency when such dependency becomes improbable**. For example we have been given
Roll → City
And
City → STDCode

If we try to derive Roll → STDCode it becomes a transitive dependency, because obviously the STDCode of a city cannot depend on the roll number issued by a school or college. In such a case the relation should be broken into two, each containing one of these two dependencies:
Roll → City
And
City → STD code

# LECTURE-30

## Boyce-Code Normal Form (BCNF)

A relationship is said to be in BCNF if it is already in 3NF and the left hand side of every dependency is a candidate key. A relation which is in 3NF is almost always in BCNF. These could be same situation when a 3NF relation may not be in BCNF the following conditions are found true.

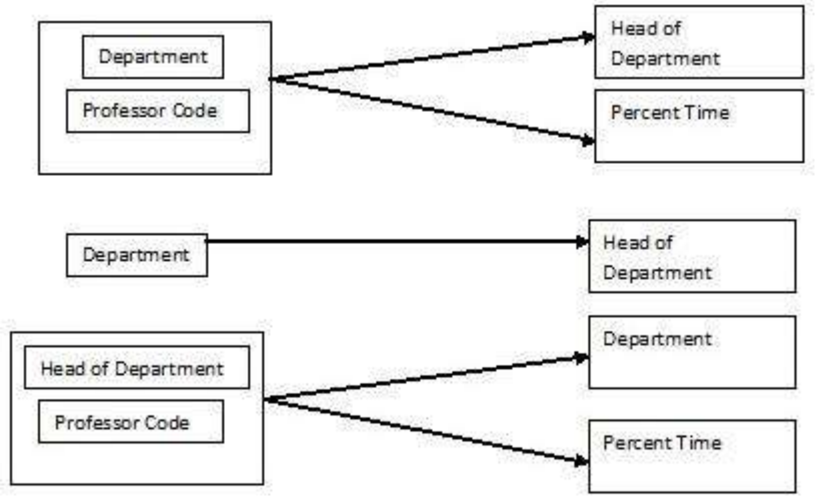
1. The candidate keys are composite.
2. There are more than one candidate keys in the relation.
3. There are some common attributes in the relation.

| Professor Code | Department | Head of Dept. | Percent Time |
|---|---|---|---|
| P1 | Physics | Ghosh | 50 |
| P1 | Mathematics | Krishnan | 50 |
| P2 | Chemistry | Rao | 25 |
| P2 | Physics | Ghosh | 75 |
| P3 | Mathematics | Krishnan | 100 |

Consider, as an example, the above relation. It is assumed that:

1. A professor can work in more than one department
2. The percentage of the time he spends in each department is given.
3. Each department has only one Head of Department.

The relation diagram for the above relation is given as the following:



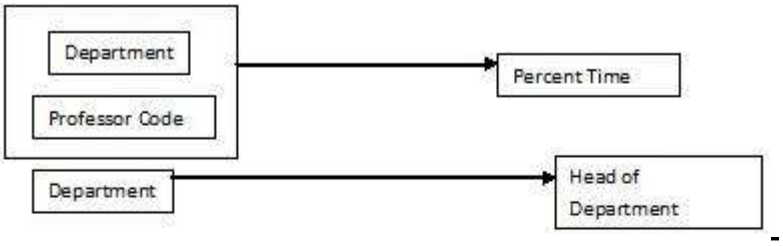
The given relation is in 3NF. Observe, however, that the names of Dept. and Head of Dept. are duplicated. Further, if Professor P2 resigns, rows 3 and 4 are deleted. We lose the information that Rao is the Head of Department of Chemistry.

The normalization of the relation is done by creating a new relation for Dept. and Head of Dept. and deleting Head of Dept. form the given relation. The normalized relations are shown in the following.

| Professor Code | Department | Percent Time |
|---|---|---|
| P1 | Physics | 50 |
| P1 | Mathematics | 50 |
| P2 | Chemistry | 25 |
| P2 | Physics | 75 |
| P3 | Mathematics | 100 |

| Department | Head of Dept. |
|---|---|
| Physics | Ghosh |
| Mathematics | Krishnan |
| Chemistry | Rao |

See the dependency diagrams for these new relations.



## Fourth Normal Form (4NF)

When attributes in a relation have multi-valued dependency, further Normalization to 4NF and 5NF are required. Let us first find out what multi-valued dependency is.

A **multi-valued dependency** is a typical kind of dependency in which each and every attribute within a relation depends upon the other, yet none of them is a unique primary key.

We will illustrate this with an example. Consider a vendor supplying many items to many projects in an organization. The following are the assumptions:

1. A vendor is capable of supplying many items.
2. A project uses many items.
3. A vendor supplies to many projects.
4. An item may be supplied by many vendors.

A multi valued dependency exists here because all the attributes depend upon the other and yet none of them is a primary key having unique value.

| Vendor Code | Item Code | Project No. |
|---|---|---|
| V1 | I1 | P1 |
| V1 | I2 | P1 |
| V1 | I1 | P3 |
| V1 | I2 | P3 |
| V2 | I2 | P1 |
| V2 | I3 | P1 |

|  |  |  |
|---|---|---|
| V3 | I1 | P2 |
| V3 | I1 | P3 |

The given relation has a number of problems. For example:

1. If vendor V1 has to supply to project P2, but the item is not yet decided, then a row with a blank for item code has to be introduced.
2. The information about item I1 is stored twice for vendor V3.

Observe that the relation given is in 3NF and also in BCNF. It still has the problem mentioned above. The problem is reduced by expressing this relation as two relations in the Fourth Normal Form (4NF). A relation is in 4NF if it has no more than one independent multi valued dependency or one independent multi valued dependency with a functional dependency.

The table can be expressed as the two 4NF relations given as following. The fact that vendors are capable of supplying certain items and that they are assigned to supply for some projects in independently specified in the 4NF relation.

Vendor-Supply

| **Vendor Code** | **Item Code** |
|---|---|
| V1 | I1 |
| V1 | I2 |
| V2 | I2 |
| V2 | I3 |
| V3 | I1 |

Vendor-Project

| **Vendor Code** | **Project No.** |
|---|---|
| V1 | P1 |
| V1 | P3 |
| V2 | P1 |
| V3 | P2 |

## Fifth Normal Form (5NF)

These relations still have a problem. While defining the 4NF we mentioned that all the attributes depend upon each other. While creating the two tables in the 4NF, although we have preserved the dependencies between Vendor Code and Item code in the first table and Vendor Code and Item code in the second table, we have lost the relationship between Item Code and Project No. If there were a primary key then this loss of dependency would not have occurred. In order to revive this relationship we must add a new table like the following. Please note that during the entire process of normalization, this is the only step where a new table is created by joining two attributes, rather than splitting them into separate tables.

| **Project No.** | **Item Code** |
|---|---|
| P1 | 11 |
| P1 | 12 |
| P2 | 11 |
| P3 | 11 |
| P3 | 13 |

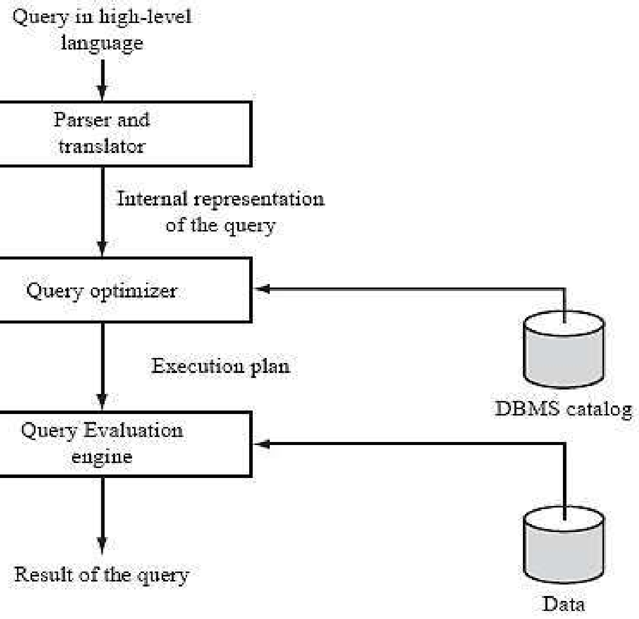
Let us finally summarize the normalization steps we have discussed so far.

| Input Relation | Transformation | Output Relation |
|---|---|---|
| All Relations | Eliminate variable length record. Remove multi-attribute lines in table. | 1NF |
| 1NF Relation | Remove dependency of non-key attributes on part of a multi-attribute key. | 2NF |
| 2NF | Remove dependency of non-key attributes on other non-key attributes. | 3NF |
| 3NF | Remove dependency of an attribute of a multi attribute key on an attribute of another (overlapping) multi-attribute key. | BCNF |
| BCNF | Remove more than one independent multi-valued dependency from relation by splitting relation. | 4NF |
| 4NF | Add one relation relating attributes with multi-valued dependency. | |

# LECTURE-31

## QUERY PROCESSING

**Query processing** includes translation of high-level queries into low-level expressions that can be used at the physical level of the file system, query optimization and actual execution of the query to get the result. It is a three-step process that consists of parsing and translation, optimization and execution of the query submitted by the user.



Query-processing Steps

A query is processed in four general steps:

1. Scanning and Parsing
2. Query Optimization or planning the execution strategy
3. Query Code Generator (interpreted or compiled)
4. Execution in the runtime database processor

## 1. Scanning and Parsing

When a query is first submitted (via an applications program), it must be scanned and parsed to determine if the query consists of appropriate syntax.
**Scanning** is the process of converting the query text into a tokenized representation.
The tokenized representation is more compact and is suitable for processing by the parser.
This representation may be in a tree form.
The **Parser** checks the tokenized representation for correct syntax.
In this stage, checks are made to determine if columns and tables identified in the query exist in the database and if the query has been formed correctly with the appropriate keywords and structure.
If the query passes the parsing checks, then it is passed on to the Query Optimizer.

## 2. Query Optimization or Planning the Execution Strategy

For any given query, there may be a number of different ways to execute it.
Each operation in the query (SELECT, JOIN, etc.) can be implemented using one or more different *Access Routines*.

For example, an access routine that employs an index to retrieve some rows would be more efficient that an access routine that performs a full table scan.

The goal of the **query optimizer** is to find a *reasonably efficient* strategy for executing the query (not quite what the name implies) using the access routines.

Optimization typically takes one of two forms: *Heuristic Optimization* or *Cost Based Optimization*

In **Heuristic Optimization**, the query execution is refined based on *heuristic rules* for reordering the individual operations.

With **Cost Based Optimization**, the overall cost of executing the query is systematically reduced by estimating the costs of executing several different execution plans.

## 3. Query Code Generator (interpreted or compiled)

Once the query optimizer has determined the execution plan (the specific ordering of access routines), the code generator writes out the actual access routines to be executed.

With an interactive session, the query code is interpreted and passed directly to the runtime database processor for execution.

It is also possible to *compile* the access routines and store them for later execution.

## 4. Execution in the runtime database processor

At this point, the query has been scanned, parsed, planned and (possibly) compiled.

The runtime database processor then executes the access routines against the database.

The results are returned to the application that made the query in the first place.

Any runtime errors are also returned.

# Lecture-32

## Query Optimization

To enable the system to achieve (or improve) acceptable performance by choosing a better (if not the best) strategy during the process of a query. One of the great strengths to the relational database.

## Automatic Optimization vs. Human Programmer

1. A good automatic optimizer will have a wealth of information available to it that human programmers typically do not have.
2. An automatic optimizer can easily reprocess the original relational request when the organization of the database is changed. For a human programmer, reorganization would involve rewriting the program.
3. The optimizer is a program, and therefore is capable of considering literally hundreds of different implementation strategies for a given request, which is much more than a human programmer can.
4. The optimizer is available to a wide range of users, in an efficient and cost-effective manner.

## The Optimization Process

1. Cast the query into some internal representation, such as a query tree structure.
2. Convert the internal representation to canonical form.

*A subset (say C) of a set of queries (say Q) is said to be a set of canonical forms for Q if and only if every query Q is equivalent to just one query in C.

During this step, some optimization is already achieved by transforming the internal representation to a better canonical form.

**Possible improvements**

a. Doing the restrictions (selects) before the join.
b. Reduce the amount of comparisons by converting a restriction condition to an equivalent condition in **conjunctive normal form-** that is, a condition consisting of a set of restrictions that are ANDed together, where each restriction in turn consists of a set of simple comparisons connected only by OR's.
c. A sequence of restrictions (selects) before the join.
d. In a sequence of projections, all but the last can be ignored.
e. A restriction of projection is equivalent to a projection of a restriction.
f. Others

3. Choose candidate low-level procedures by evaluate the transformed query.

*Access path selection: Consider the query expression as a series of basic operations (join, restriction, etc.), then the optimizer choose from a set of pre-defined, low-level implementation procedures. These procedures may involve the user of primary key, foreign key or indexes and other information about the database.

4. Generate query plans and choose the cheapest by constructing a set of candidate query plans first, then choose the best plan. To pick the best plan can be achieved by assigning cost to each given plan. The costs is computed according to the number of disk I/O's involved.

## Module-3:
## Lecture-33

### Transaction

A transaction is the smallest unit of operation done on a database.

It can be basically of two types:

→Read Transaction

→Write Transaction

3.2. ACID Properties of transaction:

**Atomicity: (all or nothing)**

A transaction is said to be atomic if a transaction always executes all its actions in one step or not executes any actions at all It means either all or none of the transactions operations are performed.

**Consistency: (No violation of integrity constraints)**

A transaction must preserve the *consistency* of a database after the execution. The DBMS assumes that this property holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

**Isolation: (concurrent changes invisibles)**

The transactions must behave as if they are executed in isolation. It means that if several transactions are executed concurrently the results must be same as if they were executed serially in some order. The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

**Durability: (committed update persist)**

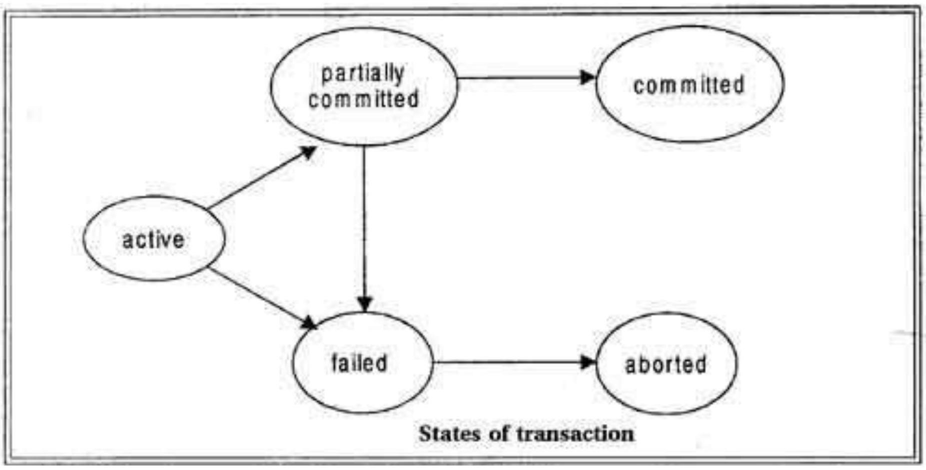
The effect of completed or committed transactions should persist even after a crash. It means once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of subsequent failures.

3.3. States of a transaction:

A transaction must be in one of the following states:

- **Active**: the initial state, the transaction stays in this state while it is executing.
- **Partially committed**: after the final statement has been executed.
- **Failed**: when the normal execution can no longer proceed.
- **Aborted**: after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**: after successful completion.

The state diagram corresponding to a transaction is shown in Figure.



States of transaction

We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily hiding in main memory and thus a hardware failure may preclude its successful completion

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be recreated when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

**Problems due to concurrency:**

1) Lost update problem

2) Dirty Read problem

3) Incorrect summary problem

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

**Concurrency Control Techniques:**

1) Locking

2) Timestamp ordering

3) Multi-version concurrency control

4) Optimistic concurrency control

**Locking (2-Phase Locking)**

2PL protocol locks are applied and removed in two phases:

1. Expanding phase: locks are acquired and no locks are released.
2. Shrinking phase: locks are released and no locks are acquired.

Locks are of two types:

1. Binary Lock
2. Share/Exclusive (Read/Write) Lock

**→Binary Lock**

A binary lock can have 2 States or values

- Locked (or 1) and
- Unlocked (or 0)

We represent the current state( or value) of the lock associated with data item X as LOCK(X).

**Operations used with Binary Locking**

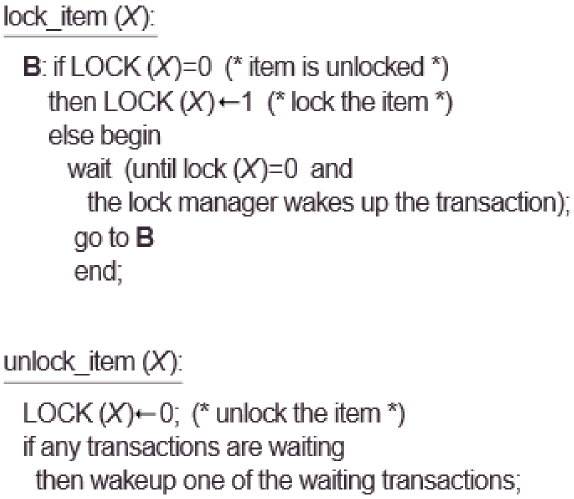
1. **lock_item :** A transaction request access to an item by first issuing a lock_item(X) operation.
   - **If LOCK(X) =1 or L(X) :** the transaction is forced to wait.
   - **If LOCK(X) = 0 or U(x) :** it is set to 1( the transaction **locks** the item) and the transaction is a load to access item X.
2. **unlock_item :** After using the data item the transaction issues an operation **unlock(X),** which sets the operation **LOCK(X) to 0** i.e. unlocks the data item so that X may be accessed by another transactions.

**Transaction Rules for Binary Locks**

Every transaction must obey the following rules :

→A transaction T must issue the lock(X) operation before any read(X) or write(X) operations in T.
→A transaction T must issue the unlock(X) operation after all read(X) and write(X) operations in T.
→If a transaction T already holds the lock on item X, then T will not issue a lock(X) operation.

→If a transaction does not holds the lock on item X, then T will not issue an unlock(X) operation.



```
lock_item (X):
  B: if LOCK (X)=0  (* item is unlocked *)
     then LOCK (X)←1  (* lock the item *)
     else begin
       wait  (until lock (X)=0  and
         the lock manager wakes up the transaction);
        go to B
        end;

unlock_item (X):
  LOCK (X)←0;  (* unlock the item *)
  if any transactions are waiting
    then wakeup one of the waiting transactions;
```

**➔Shared/Read and Exclusive/Write Lock**

The binary lock is too restrictive for data items because at most one transaction can hold on a given item whether the transaction is reading or writing. To improve it we have shared and exclusive locks in which more than one transaction can access the same item for reading purposes.i.e. the read operations on the same item by different transactions are not conflicting.

In this types of lock, system supports two kinds of lock :

- Exclusive(or Write) Locks and
- Shared(or Read) Locks.

## Shared Locks

If a transaction Ti has locked the data item A in shared mode, then a request from another transaction Tj on A for :

- **Write operation on A :** Denied. Transaction Tj has to wait until transaction Ti unlock A.
- **Read operation on A :** Allowed.

## Exclusive Locks

If a transaction Ti has locked a data item a in exclusive mode then request from some another transaction Tj for

- **Read operation on A :** Denied
- **Write operation on A :** Denied

## Operations Used with Shared and Exclusive Locks

1. Read_lock(A) or s(A)
2. Write_lock(A) or X(A)
3. Unlock(X) or U(A)

## Implementation of Shared and Exclusive Locks

Shared and exclusive locks are implemented using 4 fields :

1. Data_item_name
2. LOCK
3. Number of Records and
4. Locking_transaction(s)

Again to save space, items that are not in the lock table are considered to be unlocked. The system maintains only those records for the items that are currently locked in the lock table.

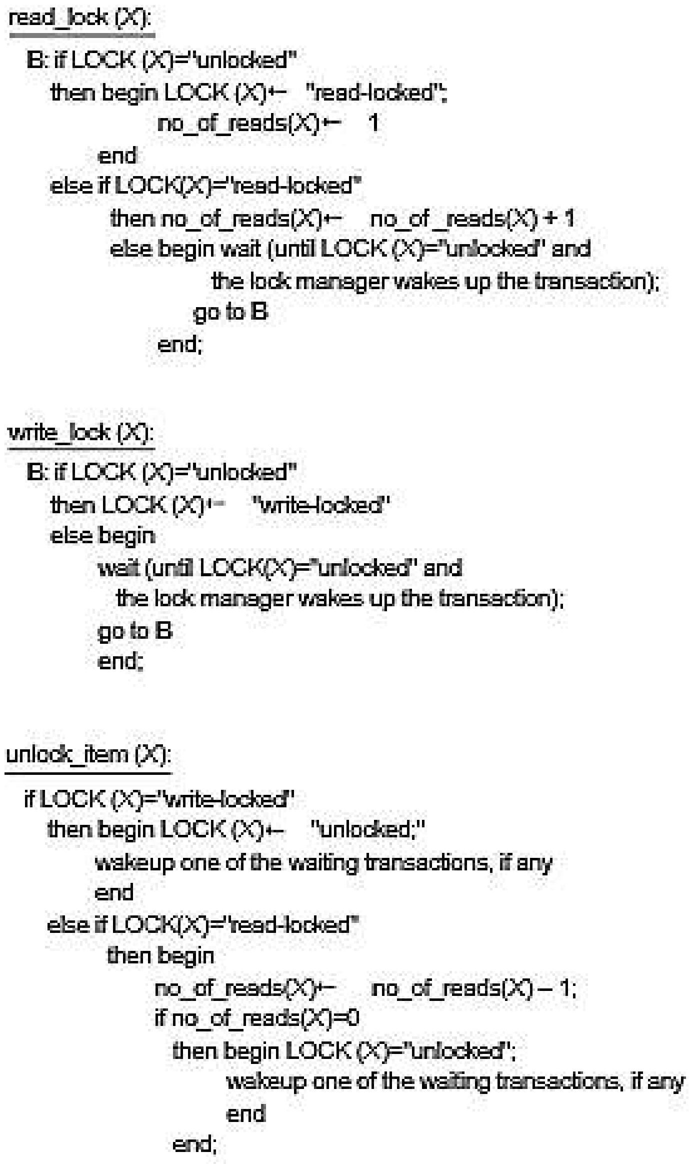
***Value of LOCK(A) : Read Locked or Write Locked***

- **If LOCK(A) = write-locked –** The value of locking transaction is a single transaction that holds the exclusive(write) Lock on A.
- **If LOCK(A) = read-locked –** The value of locking transaction is a list of one or more transactions that hold the Shared(read) on A.

## Transaction Rules for Shared and Exclusive Locks

Every transaction must obey the following rules :

1. A transaction T must issue the operation **s(A) or read_lock(A) or x(A) or write_lock(A)** before any read(A) operation is performed in T.
2. A transaction T must issue the operation x(A) or write_lock(A) before any write(A) operation is performed in T.
3. After completion of all read(A) and write(A) operations in T, a transaction T must issue an unlock(A) operation.
4. If a transaction already holds a read (shared) lock or a write (exclusive) lock on item A, then T will not issue an unlock(A) operation.
5. A transaction that already holds a lock on item A, is allowed to convert the lock from one locked state to another under certain conditions.
    - **Upgrading the Lock by Issuing a write_lock(A) Operation or Conversion of read_lock() to write_lock() :**
        - **Case 1 – When Conversion Not Possible :** A transaction T will not issue a write_lock(A) operation if it already holds a read (shared) lock or write (exclusive) lock on item A.
        - **Case 2 – When Conversion Possible :** If T is the only transaction holding a read lock on A at the time it issues the write_lock(A) operation, the lock can be upgraded;
    - **Downgrading the Lock by Issuing a read_lock(A) or Conversion of write_lock() to read_lock() :**
      A transaction T downgrade from the write lock to a read lock by acquiring the write_lock(A) or x(A), then the read_lock(A) or s(A) and then releasing the write_lock(A) or x(A).



```
read_lock (X):
  B: if LOCK (X)="unlocked"
      then begin LOCK (X)←  "read-locked";
                    no_of_reads(X)←   1
            end
      else if LOCK(X)="read-locked"
             then no_of_reads(X)←   no_of _reads(X) + 1
             else begin wait (until LOCK (X)="unlocked" and
                          the lock manager wakes up the transaction);
                        go to B
                  end;

write_lock (X):
  B: if LOCK (X)="unlocked"
      then LOCK (X)←   "write-locked"
      else begin
            wait (until LOCK(X)="unlocked" and
              the lock manager wakes up the transaction);
            go to B
            end;

unlock_item (X):
  if LOCK (X)="write-locked"
     then begin LOCK (X)←   "unlocked;"
           wakeup one of the waiting transactions, if any
           end
     else if LOCK(X)="read-locked"
            then begin
                  no_of_reads(X)←    no_of_reads(X) – 1;
                  if no_of_reads(X)=0
                    then begin LOCK (X)="unlocked";
                           wakeup one of the waiting transactions, if any
                           end
                    end;
```

# Lecture-34

## Problems due to locking

1) deadlock
2) starvation

## Deadlock:

It is an indefinite wait situation in which a series of transactions wait for each other for unknown amount of time, it obeys the following conditions:

→Hold & Wait

→No Preemption

→Mutual Exclusion

→Circular Wait

Example:

For example, assume a set of transactions {$T_0$, $T_1$, $T_2$, ...,$T_n$}. $T_0$ needs a resource X to complete its task. Resource X is held by $T_1$, and $T_1$ is waiting for a resource Y, which is held by $T_2$. $T_2$ is waiting for resource Z, which is held by $T_0$. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

## Starvation:

It is a situation in which a transaction has locked a database for unfair means & all other transactions are in indefinite waiting.

| **Starvation** | **Deadlock** |
|---|---|
| Starvation happens if same transaction is always chosen as victim. | A deadlock is a condition in which two or more transaction is waiting for each other. |
| It occurs if the waiting scheme for locked items in unfair, giving priority to some transactions over others. | A situation where two or more transactions are unable to proceed because each is waiting for one of the other to do something. |
| Starvation is also known as lived lock. | Deadlock is also known as circular waiting. |
| Avoidance:<br>->switch priorities so that every thread has a chance to have high priority.<br>-> Use FIFO order among competing request. | Avoidance:<br>->Acquire locks are predefined order.<br>->Acquire locks at once before starting. |
| It means that transaction goes in a state where transaction never progress. | It is a situation where transactions are waiting for each other. |

**Timestamp Ordering**

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction $T_i$ is denoted as TS($T_i$).
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- **If a transaction Ti issues a read(X) operation −**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

**Avoiding deadlock:**

A major disadvantage of locking is deadlock which can be avoided using timestamp ordering as follows:

There are two algorithms for deadlock avoidance.

- Wait/Die
- Wound/Wait

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance.

| | Wait/Die | Wound/Wait |
|---|---|---|
| Older process needs a resource held by younger process | **Older process** waits | **Younger process** dies |
| Younger process needs a resource held by older process | **Younger process** dies | **Younger process** waits |

**Wait-Die Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is $T_i$, which is requesting a conflicting lock, is older than $T_j$ − then $T_i$ is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is $T_i$ is younger than $T_j$ − then $T_i$ dies. $T_i$ is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

### Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then $T_i$ forces $T_j$ to be rolled back − that is $T_i$ wounds $T_j$. $T_j$ is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then $T_i$ is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

**Multi-version Concurrency Control Techniques:**

This concurrency control technique keeps the old values of a data item when the item is updated. These are known as multiversion concurrency control, because several versions (values) of an item are maintained.

→When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version of the item is retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

→An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values.

→The extreme case is a *temporal database*, which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

# Lecture-35

## Multiversion Technique Based on Timestamp Ordering

In this method, several versions , , ..., of each data item *X* are maintained. For *each version*, the value of version and the following two timestamps are kept:

1. read_TS: The read timestamp of is the largest of all the timestamps of transactions that have successfully read version .
2. write_TS: The write timestamp of is the timestamp of the transaction that wrote the value of version

Whenever a transaction T is allowed to execute a write_item(*X*) operation, a new version of item *X* is created, with both the write_TS and the read_TS set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version *Xi*, the value of read_TS() is set to the larger of the current read_TS() and TS(T).

To ensure serializability, the following two rules are used:

1. If transaction T issues a write_item(*X*) operation, and version *i* of *X* has the highest write_TS() of all versions of *X* that is also *less than or equal to* TS(T), and read_TS() > TS(T), then abort and roll back transaction T; otherwise, create a new version of *X* with read_TS() = write_TS() = TS(T).

2. If transaction T issues a read_item(*X*) operation, find the version *i* of *X* that has the highest write_TS() of all versions of *X* that is also *less than or equal to* TS(T); then return the value of to transaction T, and set the value of read_TS() to the larger of TS(T) and the current read_TS().

As we can see in case 2, a read_item(*X*) is always successful, since it finds the appropriate version to read based on the write_TS of the various existing versions of *X*. In case 1, however, transaction T may be aborted and rolled back. This happens if T is attempting to write a version of *X* that should have been read by another transaction T whose timestamp is read_TS(); however, T has already read version Xi, which was written by the transaction with timestamp equal to write_TS(). If this conflict occurs, T is rolled back; otherwise, a new version of *X*, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

## Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and certify, instead of just the two modes (read, write). Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T to read an item *X* while a single transaction T holds a write lock on *X*. This is accomplished by allowing *two versions* for each item *X*; one version must always have been written by some committed transaction. The second version *X* is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the *committed version* of *X* while T holds the write lock. Transaction T can write the value of *X* as needed, without affecting the value of the committed version *X*. However, once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version *X* of the data item is set to the value of version *X*, version *X* is discarded, and the certify locks are then released. In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes.

### 3.6.4. Optimistic Concurrency Control Techniques:

Basic idea: all transactions consist of three phases:

1. Read. Here, all writes are to private storage (shadow copies).
2. Validation. Make sure no conflicts have occurred.
3. Write. If Validation was successful, make writes public. (If not, abort!)

Useful in the following cases:

1. All transactions are readers.
2. Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data.
3. Fraction of transaction execution in which conflicts "really take place" is small compared to total pathlength.

The Validation Phase

- Goal: to guarantee that only serializable schedules result.
- Technique: actually find an equivalent serializable schedule. In particular,

1. Assign each transaction a TN during execution.
2. Ensure that if you run transactions in order induced by "<" on TNs, you get an equivalent serial schedule.

Suppose TN(Ti) < TN(Tj). Then if one of the following three conditions holds, it’s serializable:

1. Ti completes its write phase before Tj starts its read phase.
2. WS(Ti) intersect RS(Tj) = *emptyset*and Ti completes its write phase before Tj starts its write phase.
3. WS(Ti) intersect RS(Tj) = WS(Ti) *intersect* WS(Tj) = *emptyset*and Ti completes its read phase before Tj completes its read phase.

**Recovery**

**Types of Failure**

Failures may be

| | |
|---|---|
| Transaction | Caused by errors within the transaction processes. |
| System | Caused by failure of network or operating system or physical threats to the system as a whole. |
| Media | Failure of hard disk, out of memory errors, out of disk space errors. |

**Reasons for Failure**

Failure may be caused by a number of things.

| | |
|---|---|
| A System Crash | A hardware, software or network error causes the transaction to fail. |
| Transaction or System error | Some operation in the transaction may cause the failure or the user may interrupt the transaction. |
| Local Errors or Exceptions | Conditions occur during the transaction that results in transaction cancellation. |
| Concurrency Control Enforcement | Several transactions may be in deadlock so the transaction may be aborted to be restarted later. |
| Disk Failure | Read Write error on the physical disk. |
| Physical Problems | This can be any range of physical problems, such as power failure, mounting wrong disk or tape by operator, wiring problems etc |
| Catastrophe Situations | Large scale threats to the system and the data for example fire, cyclone, security breaches etc. |

Transaction errors, system errors, system crashes, concurrency problems and local errors or exceptions are the more common causes of system failure. The system must be able to recover from such failures without loss of data.

**Log-Based Recovery**

COMMIT
Signals the successful end of a transaction
• Any changes made by the transaction should be saved
• These changes are now visible to other transactions

ROLLBACK
Signals the unsuccessful end of a transaction
• Any changes made by the transaction should be undone
• It is now as if the transaction never existed

Log-Based Recovery

The most widely used structure for recording database modifications is the *log*. The log is a sequence of *log records* and maintains a history of all update activities in the database. There are several types of log records.

An *update log record* describes a single database write:

- **Transactions identifier.**
- **Data-item identifier.**
- **Old value.**
- **New value.**

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

- <Ti start>.Transaction Ti has started.
- <Ti, Xj, V1, V2> Transaction Ti has performed a write on data item Xj. Xj had value V1 before write, and will have value V2 after the write.
- < Ti commit> Transaction Ti has committed.
- < Ti abort> Transaction Ti has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification that has already been output to the database. Also we have the ability to *undo* a modification that has already been output to the database, by using the old-value field in the log records.

For log records to be useful for recovery from system and disk failures, the log must reside on stable storage. However, since the log contains a complete record of all database activity, the volume of data stored in the log may become unreasonable large.

### Deferred Database Modification

The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring all write operations of a transaction until the transaction partially commits (i.e., once the final action of the transaction has been executed). Then the information in the logs is used to execute the deferred writes. If the system crashes or if the transaction aborts, then the information in the logs is ignored.

Let T0 be transaction that transfers $50 from account A to account B:

```
T0: read (A);
     A: = A-50;
      Write (A);
      Read (B);
       B: = B + 50;
       Write (B).
```

### Immediate Database Modification

The immediate-update technique allows database modifications to be output to the database while the transaction is still in the active state. These modifications are called *uncommitted modifications*. In the event of a crash or transaction failure, the system must use the old-value field of the log records to restore the modified data items.

Transactions T0 and T1 executed one after the other in the order T0 followed by T1. The portion of the log containing the relevant information concerning these two transactions appears in the following,

**Portion of the system log corresponding to** T0 and T1

```
< T0 start >
                    < T0, A, 1000, 950 >
                     < T0, B, 2000, 2050 >
 < T0 commit >
 < T1 start >
  < T1, C, 700, 600 >
   < T0 commit >
```

### Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. Rather than reprocessing the entire log, which is time-consuming and much of it unnecessary, we can use *checkpoints*:

- Output onto stable storage all the log records currently residing in main memory.
- Output to the disk all modified buffer blocks.
- Output onto stable storage a log record, **<checkpoint>**.

# Lecture-36

## Serializability:

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

## Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

It may happen that we may want to execute the same set of transaction in a different schedule on another day. Keeping in mind these rules, we may sometimes alter parts of one schedule (S1) to create another schedule (S2) by swapping only the non-conflicting parts of the first schedule. The conflicting parts cannot be swapped in this way because the ordering of the conflicting instructions is important and cannot be changed in any other schedule that is derived from the first. If these two schedules are made of the same set of transactions, then both S1 and S2 would yield the same result if the conflict resolution rules are maintained while creating the new schedule. In that case the schedule S1 and S2 would be called **Conflict Equivalent**.

## View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Except in these three cases, any alteration can be possible while creating S2 by modifying S1.

## Lecture-37

### Object Oriented Databases

Object oriented databases are also called Object Database Management Systems (ODBMS). Object databases store objects rather than data such as integers, strings or real numbers. Objects are used in object oriented languages such as Smalltalk, C++, Java, and others. Objects basically consist of the following:

- Attributes - Attributes are data which defines the characteristics of an object. This data may be simple such as integers, strings, and real numbers or it may be a reference to a complex object.
- Methods - Methods define the behavior of an object and are what was formally called procedures or functions.

Therefore objects contain both executable code and data

### Object Persistence

With traditional databases, data manipulated by the application is transient and data in the database is persisted (Stored on a permanent storage device). In object databases, the application can manipulate both transient and persisted data.

### When to Use Object Databases

Object databases should be used when there is complex data and/or complex data relationships. This includes a many to many object relationship. Object databases should not be used when there would be few join tables and there are large volumes of simple transactional data.

Object databases work well with:

- CAS Applications (CASE-computer aided software engineering, CAD-computer aided design, CAM-computer aided manufacture)
- Multimedia Applications
- Object projects that change over time.
- Commerce

### Object Database Advantages over RDBMS

- Objects don't require assembly and disassembly saving coding time and execution time to assemble or disassemble objects.
- Reduced paging
- Easier navigation
- Better concurrency control - A hierarchy of objects may be locked.
- Data model is based on the real world.
- Works well for distributed architectures.
- Less code required when applications are object oriented.

### Object Database Disadvantages compared to RDBMS

- Lower efficiency when data is simple and relationships are simple.
- Relational tables are simpler.
- Late binding may slow access speed.
- More user tools exist for RDBMS.
- Standards for RDBMS are more stable.

- Support for RDBMS is more certain and change is less likely to be required.

## How Data is Stored

Two basic methods are used to store objects by different database vendors.

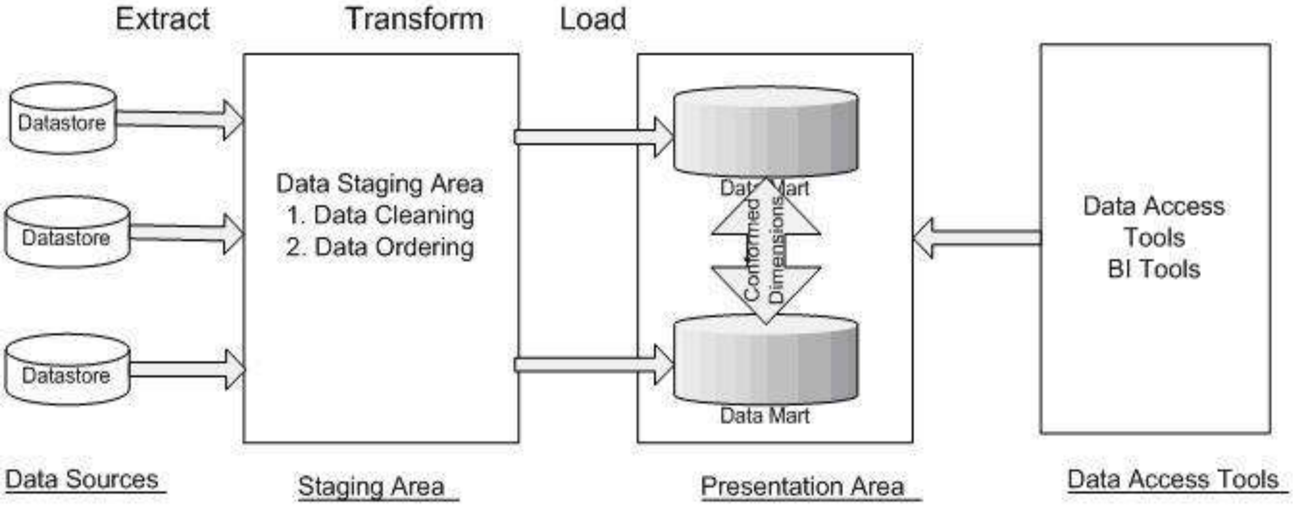
- Each object has a unique ID and is defined as a subclass of a base class, using inheritance to determine attributes.
- Virtual memory mapping is used for object storage and management.

## Data Warehouse

- **A data warehouse is a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process.**
- **Subject-Oriented**: A data warehouse can be used to analyze a particular subject area. For example, "sales" can be a particular subject.
- **Integrated**: A data warehouse integrates data from multiple data sources. For example, source A and source B may have different ways of identifying a product, but in a data warehouse, there will be only a single way of identifying a product.
- **Time-Variant**: Historical data is kept in a data warehouse. For example, one can retrieve data from 3 months, 6 months, 12 months, or even older data from a data warehouse. This contrasts with a transactions system, where often only the most recent data is kept. For example, a transaction system may hold the most recent address of a customer, where a data warehouse can hold all addresses associated with a customer.
- **Non-volatile**: Once data is in the data warehouse, it will not change. So, historical data in a data warehouse should never be altered.

### Data Warehousing Architecture & Components

Following diagram depicts different components of Data Warehouse architecture.



### Operational Source System

It’s the traditional OLTP systems which stores transaction data of the organizations business. Its generally used one record at any time not necessarily stores history of the organizations information’s. Operational source systems generally not used for reporting like data warehouse.

### Data Staging Area

Data staging area is the storage area as well as set of ETL process that extract data from source system. It is everything between source systems and Data warehouse.

Data staging are never be used for reporting purpose. Data is extracted from source system and stored, cleansed, transformed in staging area to load into data warehouse.

Staging are not necessarily the DBMS. It could be flat files also. Staging area can be structured like normalized source systems. It totally depends on choice and need of development process.

**Data Presentation Area**

Data presentation area is generally called as data warehouse. It's the place where cleaned, transformed data is stored in a dimensionally structured warehouse and made available for analysis purpose

**Data Access Tools**

once data is available in presentation area it is accessed using data access tools like Business Objects.

**Schema:**
**(Star schema)**

The star schema architecture is the simplest data warehouse schema. It is called a star schema because the diagram resembles a star, with points radiating from a center. The center of the star consists of fact table and the points of the star are the dimension tables. Usually the fact tables in a star schema are in third normal form(3NF) whereas dimensional tables are de-normalized. Despite the fact that the star schema is the simplest architecture, it is most commonly used nowadays and is recommended by Oracle.

→Fact Tables

A fact table typically has two types of columns: foreign keys to dimension tables and measures those that contain numeric facts. A fact table can contain fact's data on detail or aggregated level.

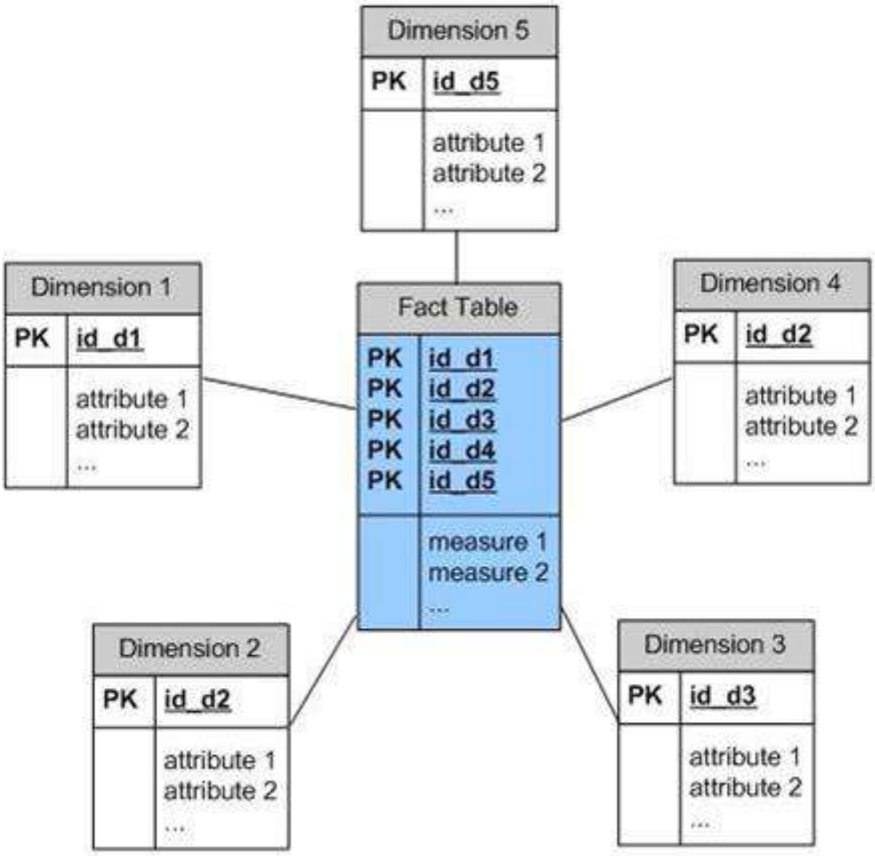
→Dimension Tables

A dimension is a structure usually composed of one or more hierarchies that categorizes data. If a dimension hasn't got a hierarchies and levels it is called **flat dimension or list**. The primary keys of each of the dimension tables are part of the composite primary key of the fact table. Dimensional attributes help to describe the dimensional value. They are normally descriptive, textual values. Dimension tables are generally small in size then fact table.

Typical fact tables store data about sales while dimension tables data about geographic region(markets, cities) , clients, products, times, channels.

The main characteristics of star schema:

1. Simple structure -> easy to understand schema
2. Great query effectives -> small number of tables to join
3. Relatively long time of loading data into dimension tables -> de-normalization, redundancy data caused that size of the table could be large.
4. The most commonly used in the data warehouse implementations -> widely supported by a large number of business intelligence tools



**Data Mining:**

→Generally, data mining (sometimes called data or knowledge discovery) is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both. Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases.

→While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, and neural networks. Generally, any of four types of relationships are sought:

- **Classes**: Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.
- **Clusters**: Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.
- **Associations**: Data can be mined to identify associations. The beer-diaper example is an example of associative mining.

- **Sequential patterns**: Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer's purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

**Techniques used in data mining:**

- Association

  **Association is one of the best-known data mining technique. In association, a pattern is discovered based on a relationship between items in the same transaction. That's is the reason why association technique is also known as *relation technique*. The association technique is used in *market basket analysis* to identify a set of products that customers frequently purchase together.**

  Retailers are using association technique to research customer's buying habits. Based on historical sale data, retailers might find out that customers always buy crisps when they buy beers, and, therefore, they can put beers and crisps next to each other to save time for customer and increase sales.

- Classification

  Classification is a classic data mining technique based on machine learning. Basically, classification is used to classify each item in a set of data into one of a predefined set of classes or groups. Classification method makes use of mathematical techniques such as decision trees, linear programming, neural network and statistics. In classification, we develop the software that can learn how to classify the data items into groups. For example, we can apply classification in the application that "given all records of employees who left the company, predict who will probably leave the company in a future period." In this case, we divide the records of employees into two groups that named "leave" and "stay". And then we can ask our data mining software to classify the employees into separate groups.

- Clustering

  Clustering is a data mining technique that makes a meaningful or useful cluster of objects which have similar characteristics using the automatic technique. The clustering technique defines the classes and puts objects in each class, while in the classification techniques, objects are assigned into predefined classes. To make the concept clearer, we can take book management in the library as an example. In a library, there is a wide range of books on various topics available. The challenge is how to keep those books in a way that readers can take several books on a particular topic without hassle. By using the clustering technique, we can keep books that have some kinds of similarities in one cluster or one shelf and label it with a meaningful name. If readers want to grab books in that topic, they would only have to go to that shelf instead of looking for the entire library.

- Prediction

  The prediction, as its name implied, is one of a data mining techniques that discovers the relationship between independent variables and relationship between dependent and independent variables. For instance, the prediction analysis technique can be used in the sale to predict profit for the future if we consider the sale is an independent variable, profit could be a dependent variable. Then based on the historical sale and profit data, we can draw a fitted regression curve that is used for profit prediction.

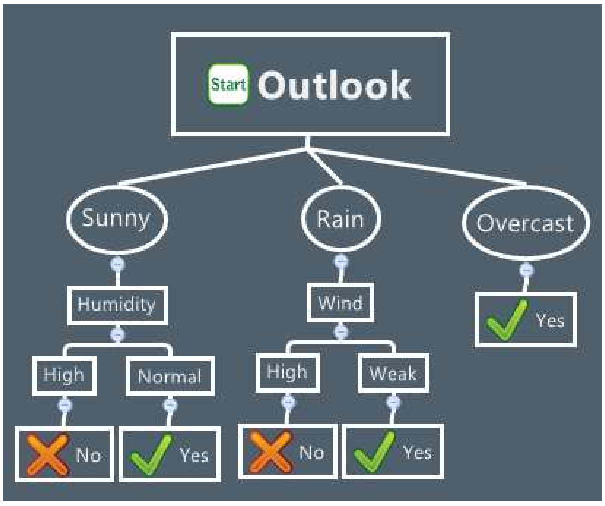
- Sequential Patterns

  Sequential patterns analysis is one of data mining technique that seeks to discover or identify similar patterns, regular events or trends in transaction data over a business period.

  In sales, with historical transaction data, businesses can identify a set of items that customers buy together different times in a year. Then businesses can use this information to recommend customers buy it with better deals based on their purchasing frequency in the past.

- Decision trees

  The A decision tree is one of the most common used data mining techniques because its model is easy to understand for users. In decision tree technique, the root of the decision tree is a simple question or condition that has multiple answers. Each answer then leads to a set of questions or conditions that help us determine the data so that we can make the final decision based on it. For example, We use the following decision tree to determine whether or not to play tennis:



  Starting at the root node, if the outlook is overcast then we should definitely play tennis. If it is rainy, we should only play tennis if the wind is the week. And if it is sunny then we should play tennis in case the humidity is normal.

# Lecture-38

## Parallel Database

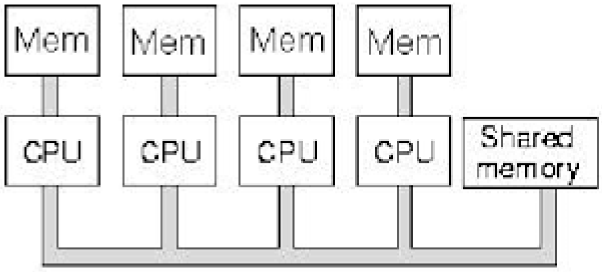
A **parallel database** system performs parallel operations, such as loading data, building indexes and evaluating queries.

Parallel databases can be roughly divided into two groups,

a) Multiprocessor architecture:

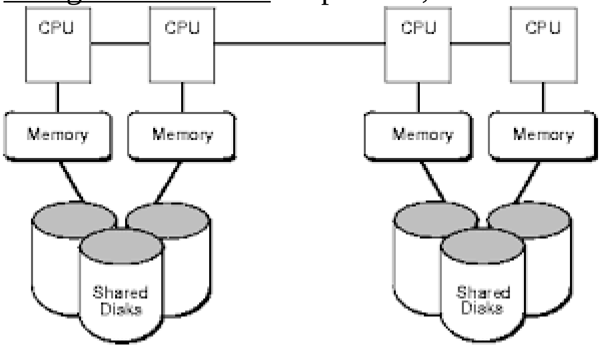
→Shared memory architecture

Where multiple processors share the main memory space.



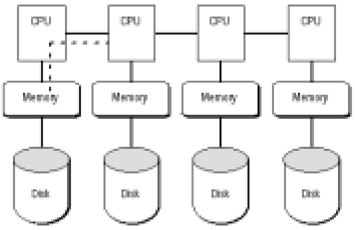
→Shared disk architecture

Where each node has its own main memory, but all nodes share mass storage, usually a storage area network. In practice, each node usually also has multiple processors.



→Shared nothing architecture

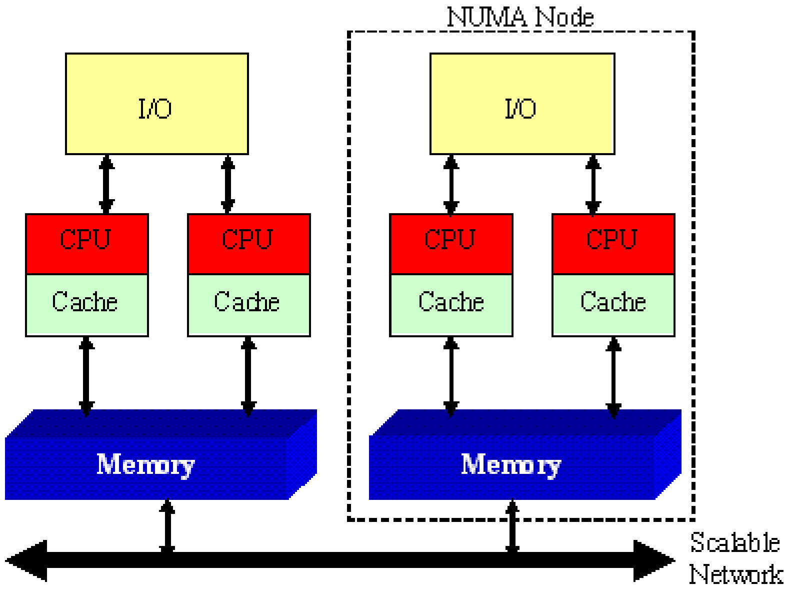
Where each node has its own mass storage as well as main memory.



**Shared-Nothing Architecture**

b) The other architecture group is called hybrid architecture, which includes:

- Non-Uniform Memory Architecture (NUMA), which involves the non-uniform memory access.



- Cluster (shared nothing + shared disk: SAN/NAS), which is formed by a group of connected computers.

**Distributed Database:**

→A centralized distributed database management system (DDBMS) manages the database as if it were all stored on the same computer. The DDBMS synchronizes all the data periodically and, in cases where multiple users must access the same data, ensures that updates and deletes performed on the data at one location will be automatically reflected in the data stored elsewhere.

→The users and administrators of a distributed system, should, with proper implementation, interact with the system as if the system was centralized. This transparency allows for the functionality desired in such a structured system without special programming requirements, allowing for any number of local and/or remote tables to be accessed at a given time across the network.

The different types of transparency sought after in a DDBMS are

1. data distribution transparency,
2. heterogeneity transparency,
3. transaction transparency, and
4. performance transparency.

→Data distribution transparency requires that the user of the database should not have to know how the data is fragmented (fragmentation transparency), know where the data they access is actually located (location transparency), or be aware of whether multiple copies of the data exist (replication transparency).

→Heterogeneity transparency requires that the user should not be aware of the fact that they are using a different DBMS if they access data from a remote site. The user should be able to use the same language that they would normally use at their regular access point and the DDBMS should handle query language translation if needed.

→Transaction transparency requires that the DDBMS guarantee that concurrent transactions do not interfere with each other (concurrency transparency) and that it must also handle database recovery

(recovery transparency).

→Performance transparency mandates that the DDBMS should have a comparable level of performance to a centralized DBMS. Query optimizers can be used to speed up response time.

**Types of DDB Design**
Non-Partitioned, Non-Replicated
Partitioned, Non-Replicated
Non-Partitioned, Replicated
Partitioned, Replicated

**Advantages of DDBMS's**

- Reflects organizational structure
- Improved share ability
- Improved availability
- Improved reliability
- Improved performance
- Data are located nearest the greatest demand site and are dispersed to match business requirements.
- Faster Data Access because users only work with a locally stored subset of the data.
- Faster data processing because the data is processed at several different sites.
-Growth Facilitation: New sites can be added without compromising the operations of other sites.
-Improved communications because local sites are smaller and closer to customers.
- Reduced operating costs: It is more cost-effective to add workstations to a network rather than update a mainframe system.
- User Friendly interface equipped with an easy-to-use GUI.
- Less instances of single-point failure because data and workload are distributed among other workstations.
- Processor independence: The end user is able to access any available copy of data.

**Disadvantages of DDBMS**
- Increased Cost
-Integrity control more difficult,
-Lack of standards,
-Database design more complex.
- Complexity of management and control. Applications must recognize data location and they must be able to stitch together data from various sites.
- Technologically difficult: Data integrity, transaction management, concurrency control, security, backup, recovery, query optimization, access path selection are all issues that must be addressed and resolved
- Security lapses have increased instances when data are in multiple locations.
- Lack of standards due to the absence of communication protocols can make the processing and distribution of data difficult.
- Increased storage and infrastructure requirements because multiple copies of data are required at various separate locations which would require more disk space.
- Increased costs due to the higher complexity of training.
- Requires duplicate infrastructure (personnel, software and licensing, physical location/environment) and these can sometimes offset any operational savings.