DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 1
INTRODUCTION TO DBMS

Database

A database is a collection of data, typically describing the activities of one or more related organizations.

Example: a university database might contain information about the following:
• *Entities* such as students, faculty, courses, and classrooms.
• *Relationships* between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

Database Management System (DBMS)

- A database management system, or DBMS, is software designed to assist in maintaining and utilizing large collections of data.
- The alternative to using a DBMS is to store the data in files and write application-specific code to manage it.

A Historical Perspective or History of Data base Systems

- Finding information from a huge volume of papers or deleting/modifying an entry is a difficult task in pen and paper based approach.
- To overcome the hassles faced in manual record keeping, it is desirable to computerize storage of data.
- From the earliest days of computers, storing and manipulating data have been a major application focus.
- In the late 1960s, IBM developed the Information Management System (IMS) DBMS, used even today in many major installations.
- IMS formed the basis for an alternative data representation framework called the *hierarchical data model*.
- In 1970, Edgar Codd, at IBM's San Jose Research Laboratory, proposed a new data representation framework called the *relational data model.*
- In the 1980s, the relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use.
- SQL was standardized in the late 1980s, and the current standard, SQL: 1999, was adopted by the American National Standards Institute (ANSI) and International Organization for Standardization (ISO).
- Specialized systems have been developed by numerous vendors for creating *data warehouses,* consolidating data from several databases, and for carrying out specialized analysis.
- Commercially, database management systems represent one of the largest and most vigorous market segments.

FILE SYSTEMS VERSUS DBMS

File System	Database Management System (DBMS)
1. It is a software system that manages and controls the data files in a computer system.	1. It is a software system used for creating and managing the databases. DBMS provides a systematic way to access, update, and delete data.
2. File system does not support multi-user access.	2. Database Management System supports multi-user access.
3. Data consistency is less in the file system.	3. Data consistency is more due to the use of normalization.
4. File system is not secured.	4. Database Management System is highly secured.
5. File system is used for storing the unstructured data.	5. Database management system is used for storing the structured data.
6. In the file system, data redundancy is high.	6. In DBMS, Data redundancy is low.
7. No data backup and recovery process is present in a file system.	7. There is a backup recovery for data in DBMS.
8. Handling of a file system is easy.	8. Handling a DBMS is complex.
9. Cost of a file system is less than the DBMS.	9. Cost of database management system is more than the file system.
10. If one application fails, it does not affect other application in a system.	10. If the database fails, it affects all application which depends on it.
11. In the file system, data cannot be shared because it is distributed in different files.	11. In DBMS, data can be shared as it is stored at one place in a database.
12. These system does not provide concurrency facility.	12. This system provides concurrency facility.
13. **Example:** NTFS (New technology file system), EXT (Extended file system), etc.	13. **Example:** Oracle, MySQL, MS SQL Server, DB2, Microsoft Access, etc.

ADVANTAGES OF A DBMS

- **Data Independence**: The DBMS provides an abstract view of the data that hides data representation and storage details.
- **Efficient Data Access**: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently.
- **Data Integrity and Security**: If data is always accessed through the DBMS, the DBMS can enforce integrity constraints. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, it can enforce *access controls* that govern what data is visible to different classes of users.
- **Data Administration**: When several users share the data, centralizing the administration of data can offer significant improvements.
- **Concurrent Access and Crash Recovery**: A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced Application Development Time:** The high-level interface to the data, facilitates quick application development.

Database System Applications

Applications where we use Database Management Systems are:

- **Telecom**: There is a database to keep track of the information regarding calls made, network usage, customer details etc. Without the database systems it is hard to maintain that huge amount of data that keeps updating every millisecond.
- **Industry**: Where it is a manufacturing unit, warehouse or distribution centre, each one needs a database to keep the records of ins and outs. For example distribution centre should keep a track of the product units that supplied into the centre as well as the products that got delivered out from the distribution centre on each day; this is where DBMS comes into picture.
- **Banking System**: For storing customer info, tracking day to day credit and debit transactions, generating bank statements etc. All this work has been done with the help of Database management systems.
- **Education sector**: Database systems are frequently used in schools and colleges to store and retrieve the data regarding student details, staff details, course details, exam details, payroll data, attendance details, fees details etc. There is a hell lot amount of inter-related data that needs to be stored and retrieved in an efficient manner.
- **Online shopping**: You must be aware of the online shopping websites such as Amazon, Flipkart etc. These sites store the product information, your addresses and preferences, credit details and provide you the relevant list of products based on your query. All this involves a Database management system.

DATABASE MANAGEMENT SYSTEMS

UNIT 1 – TOPIC 2

DATA MODELS

DATA MODEL

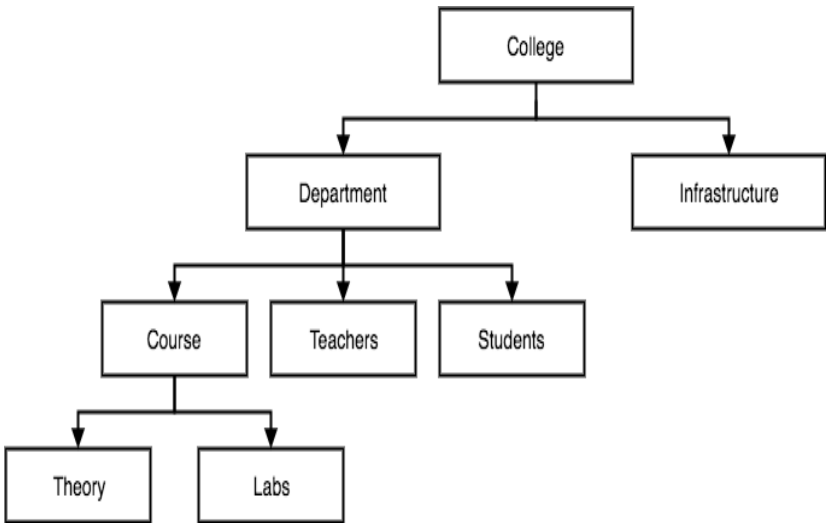
- A **data model** is the underlying structure of DBMS. It is a collection of high-level data description constructs that hide many low-level storage details.
- A data model describes how a database's logical structure is represented.
- It is a conceptual tool used to describe data, relationship, semantics and constraints.
- Most database management systems today are based on the **relational data model.**
- A **semantic data model** is a more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise.
- A widely used semantic data model called the entity-relationship (ER) model allows us to pictorially denote entities and the relationships among them.

Data model types

- Older models:
  - Network model
  - Hierarchical model
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-Oriented data models

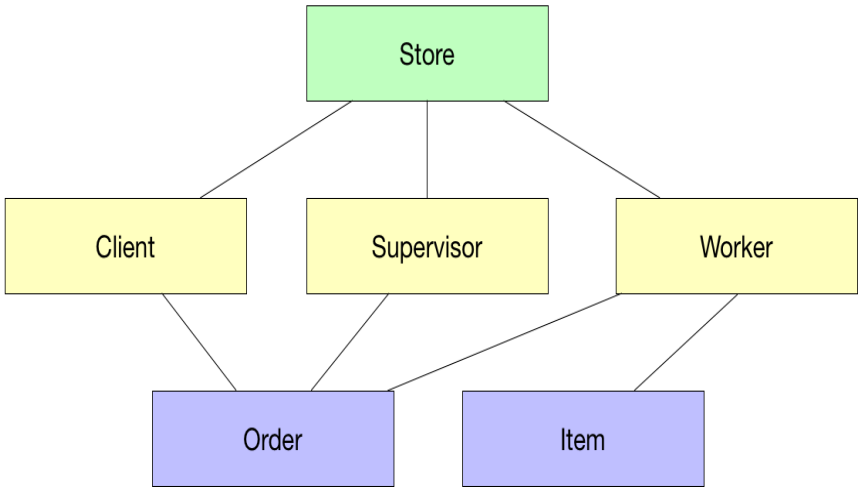
Hierarchical Model

- Hierarchical Model was the first DBMS model.
- This model organizes the data in the hierarchical tree structure.
- The hierarchy starts from the root which has root data and then it expands in the form of a tree adding child node to the parent node.
- This model easily represents some of the real-world relationships like food recipes, sitemap of a website etc.



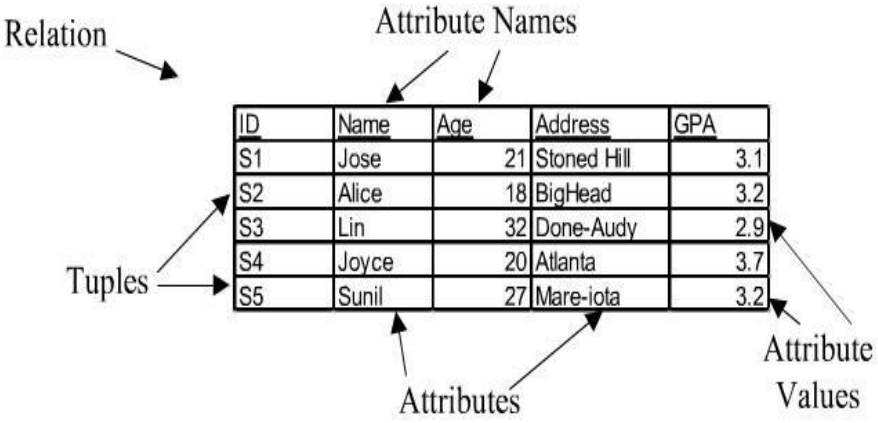
Network Model

- This model is an extension of the hierarchical model.
- It was the most popular model before the relational model.
- This model is the same as the hierarchical model; the only difference is that a record can have more than one parent.
- It replaces the hierarchical tree with a graph.



Relational model

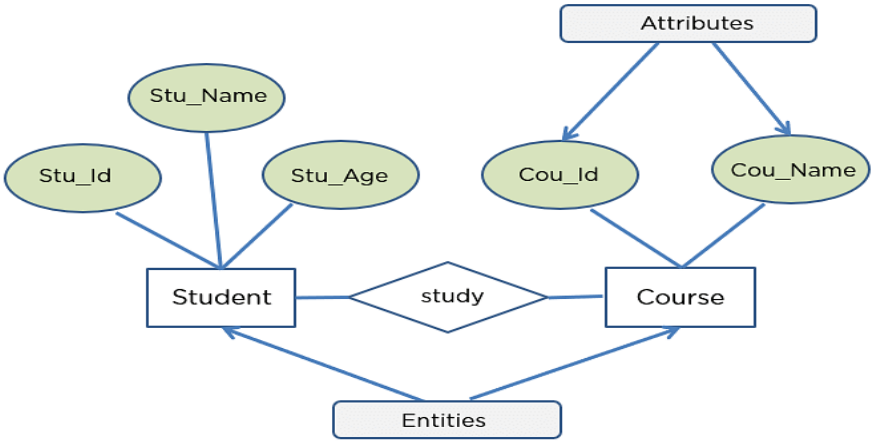
- The central data description construct in this model is a relation, which can be thought of as a set of records.
- A description of data in terms of a data model is called a schema.
- In the relational model, the schema for a relation specifies its name, the name of each field (or attribute or column), and the type of each field.
- Students (*sid:* string, *name:* string, *login:* string, *age:* integer, *gpa:* real)



ID	Name	Age	Address	GPA
S1	Jose	21	Stoned Hill	3.1
S2	Alice	18	BigHead	3.2
S3	Lin	32	Done-Audy	2.9
S4	Joyce	20	Atlanta	3.7
S5	Sunil	27	Mare-iota	3.2

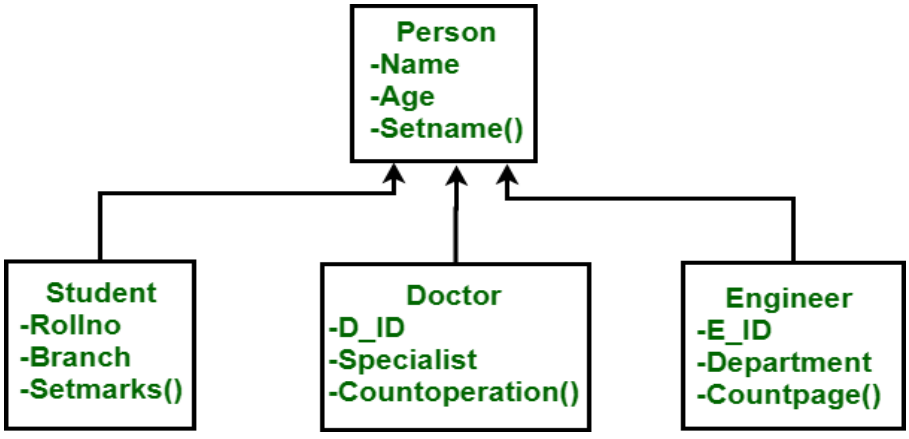
Entity-Relationship Model

- Entity-Relationship Model or simply ER Model is a high-level data model diagram.
- In this model, we represent the real-world problem in the pictorial form to make it easy for the stakeholders to understand.
- It is also very easy for the developers to understand the system by just looking at the ER diagram.
- We use the ER diagram as a visual tool to represent an ER Model.



Object-Oriented Data Model

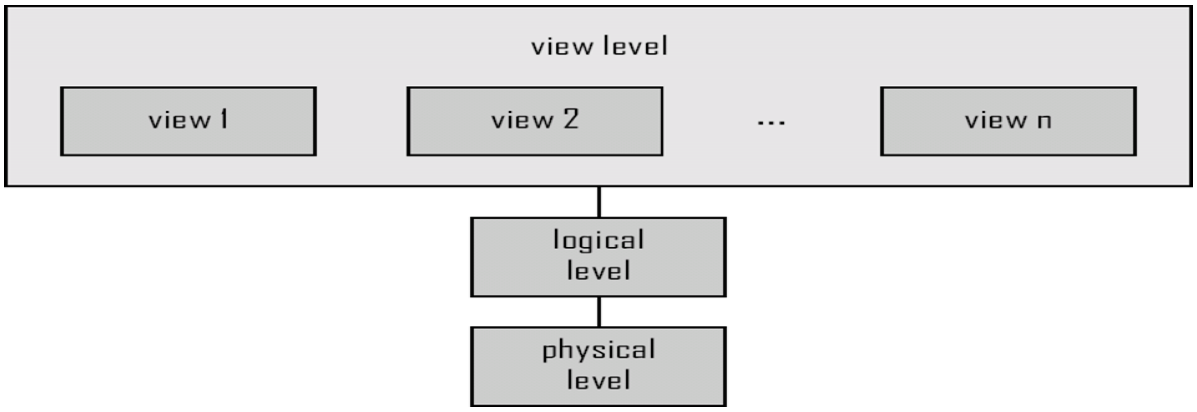
- The real-world problems are more closely represented through the object-oriented data model.
- In this model, both the data and relationship are present in a single structure known as an object.
- We can store audio, video, images, etc in the database which was not possible in the relational model (although you can store audio and video in relational database, it is advised not to store in the relational database).
- In this model, two are more objects are connected through links. We use this link to relate one object to other object.



DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 3
LEVELS OF ABSTRACTION & DATA INDEPENDENCE

LEVELS OF DATA ABSTRACTION

In a Database Management System (DBMS), abstraction refers to the process of hiding the complexity of the underlying data structure and operations from the users. This abstraction is essential for managing large volumes of data efficiently and for providing a simplified interface for users to interact with the database.



There are 3 level architecture of database design.

Physical level / Internal View: (How) it is the lowest level of data abstraction. It tells us how the data is actually stored in memory. While designing the database few basic information needs to be considered like, storage allocation, indexing methods, data retrieval mechanisms, usability, size of the memory etc.

Example : record : customer

Logical level / Conceptual View: (What) The conceptual level, also known as the logical level, is the level of abstraction that defines the overall structure (schema) of the database. It hides the details of the physical storage from the database users and applications. It defines the relationships and constraints among the data elements without specifying how the data is physically stored.

The conceptual level is designed to provide a high-level view of the database that is independent of any specific implementation. It allows users to define the structure of the database in a way that is meaningful to them without worrying about the details of how the data is stored or accessed.

Example :

type *customer* = record
customer_id : string;
customer_name : string;
customer_street : string;
customer_city : string;
end;

View level / External View: (Who)

The external level, also known as the View level, is the highest level of abstraction and is closest to the end users. It defines how the data is viewed by the users or applications that access the database. It provides a high-level view of the database that is tailored to the specific needs of each user or application.

The external level is designed to provide a level of abstraction that shields the users or applications from the complexities of the underlying database. It allows users to work with the data in a way that is meaningful to them without worrying about the details of how the data is stored or organized.

Benefits of Levels of Abstraction in DBMS

The levels of abstraction in DBMS provide several benefits to users and applications, including:

- The levels of abstraction in DBMS provide a separation between the way data is viewed by users or applications and how it is stored and accessed by the database management system. This allows changes to be made to the physical storage and access methods without affecting the external or conceptual levels.
- The levels of abstraction in DBMS make it easier for database administrators to manage the database. They can make changes to the physical storage and access methods without affecting the users or applications that interact with the database.
- The levels of abstraction in DBMS allow the database management system to optimize the physical storage and access methods for performance without affecting the way data is viewed by users or applications.
- The levels of abstraction in DBMS allow users or applications to view the data in a way that is meaningful to them without worrying about the underlying implementation details. This makes it easier to adapt to changing business requirements and user needs.

Data Independence

The ability to modify the schema in one level without affecting the schema in next higher level is called data independence. It helps to keep the data separated from all programs that makes use of it.

- **Physical Data Independence** – The ability to modify the physical schema without changing the logical schema. It helps to keep the data separated from all program that makes use of it. **Example**: The Conceptual structure of the database would not be affected by any changes like utilizing a new storage device, change in storage size of the database system server, modifying the data structures used for storage, using an alternate file organization technique, changing from sequential to random access files etc.
- **Logical data independence**: The ability to modify the logical schema without affecting the schema in next higher level (external schema). The user view of the data would not be affected by any changes to the conceptual view of the data. These changes may include insertion or deletion of attributes, altering table structures entities or relationships to the logical schema, etc.

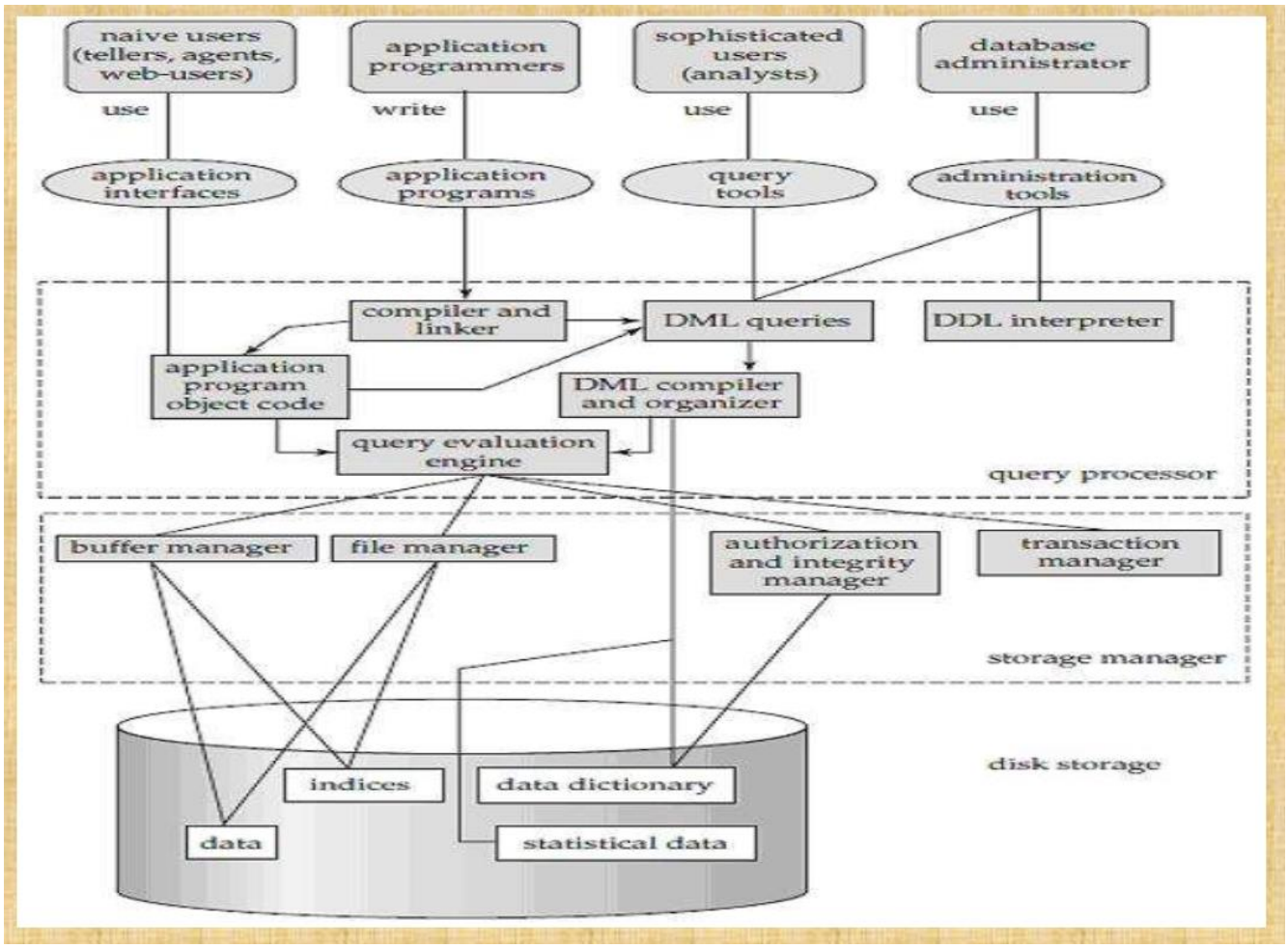
DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 4
STRUCTURE OF DBMS

STRUCTURE OF DBMS

Database Management System is the combination of a Database and all the functionalities to organize and manage the data. It is software that allows us to efficiently interact with data at various levels of abstraction.

As Database Management System is a complex set of programs, it is necessary to understand all the components of DBMS so that we can easily manage the database

- A database system is partitioned into modules, each of that deal with responsibilities of the overall system.
- **The structure of DBMS refers to the logical representation of all of its modules.**
- The functional components of a database system can be broadly divided into the
  - **Query processor**
  - **Storage manager and**
  - **Disk Storage**



1. **Query Processor:**
   It interprets the requests (queries) received from end user via an application program into instructions. It also executes the user request which is received from the DML compiler.

   Query Processor contains the following components

   - **DDL Interpreter –**
     DDL Interpreter interprets DDL statements like those used in schema definitions (such as create, remove, etc.). This interpretation yields a set of tables that include the meta-data (data of data) that is kept in the data dictionary.
   - **DML Compiler –**
     DML Compiler converts DML statements like select, update, and delete into low-level instructions or simply machine-readable object code, to enable execution. It processes the DML statements into low level instruction (machine language), so that they can be executed.
   - **Query Evaluation Engine –**
     It evaluates the SQL commands used to access the database's contents before returning the result of the query. A single query can be translated into a number of evaluation plans.
   - **Query Optimizer –**
     query optimization determines the most effective technique to carry out a query.

2. **Storage Manager :**
   - Storage Manager is a program that provides an interface between the data stored in the database and the queries received.
   - It is also known as Database Control System.
   - It maintains the consistency and integrity of the database by applying the constraints and executes the DCL statements.
   - It is responsible for updating, storing, deleting, and retrieving data in the database.

It contains the following components –

- **Authorization Manager –**
  It ensures role-based access control, i.e. it checks whether the particular person is privileged to perform the requested operation or not.

- **Integrity Manager –**
  It checks the integrity constraints when the database is modified.

- **Transaction Manager –**
  It controls concurrent access by performing the operations in a scheduled way that it receives the transaction. Thus, it ensures that the database remains in the consistent state before and after the execution of a transaction.

- **File Manager –**
  It manages the file space and the data structure used to represent information in the database.

- **Buffer Manager –**
  It is responsible for cache memory and the transfer of data between the secondary storage and main memory.

3. **Disk Storage:**
   A DBMS can use various kinds of Data Structures as a part of physical system implementation in the form of disk storage.

   It contains the following components – Data Files, Data Dictionary, and Indices

   Data Files – It stores the data in the files supported by the native Operating System.

   Data Dictionary – It contains the information about the structure of any database object. It is the repository of information that governs the metadata.

   Indices – These indices are used to access and retrieve the data in a very fast and efficient way. It provides faster retrieval of data item.

Types of Database Users

1. Database Administrator (DBA):

- It is a person or a team, who is responsible for managing the overall database management system.
- It is the leader of the database. It is like a super-user of the system.
- It is responsible for the administration of all the three levels of the database.

DBA is responsible for:

- Deciding the instances for the database.
- Defining the Schema
- Liaising with Users
- Define Security
- Back-up and Recovery
- Monitoring the performance

2. Database Designers:

- Database designers design the appropriate structure for the database, where we share data.

3. System Analyst:

- System analyst analyses the requirements of end users, especially naïve and parametric end users.

4. Application Programmers:

- Application programmers are computer professionals, who write application programs.

5. Naïve Users / Parametric Users:

- Naïve Users are Un-sophisticated users, which has no knowledge of the database. These users are like a layman, which has a little bit of knowledge of the database.
- Naive Users are just to work on developed applications and get the desired result.
- For Example: Railway's ticket booking users are naive users. Or Clerical staff in any bank is a naïve user because they don't have any DBMS knowledge but they still use the database and perform their given task.

6. Sophisticated Users:

- Sophisticated users can be engineers, scientists, business analyst, who are familiar with the database. These users interact with the database but they do not write programs

7. Casual Users / Temporary Users:

- These types of users communicate with the database for a little period of time.

DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 5
Database Design and E R Model

Database design is the set of procedures or collection of tasks used for organizing the data to implement a database.
It involves several key steps to ensure efficient storage, retrieval, and manipulation of data.

The database design process can be divided into six steps. The ER model is most relevant to the first three steps.

1. **Requirements Analysis:**
    - what data is to be stored in the database,
    - what applications must be built on top of it, and
    - what operations to be performed to meet performance requirements.

2. **Conceptual Database Design**: Based on the information gathered, develop a high-level description of the data to be stored in the database, along with the constraints.

3. **Logical Database Design**: We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS.

4. **Schema Refinement**: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it.

5. **Physical Database Design**: In this step, we consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria.

6. **Application and Security Design**: Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. Implement security measures such as authorization, authentication, and encryption to protect the database. Enforce data integrity etc.

Entity-Relationship Model:

- An Entity-Relationship Model specifies the enterprise schema that represents the logical structure of the database graphically.

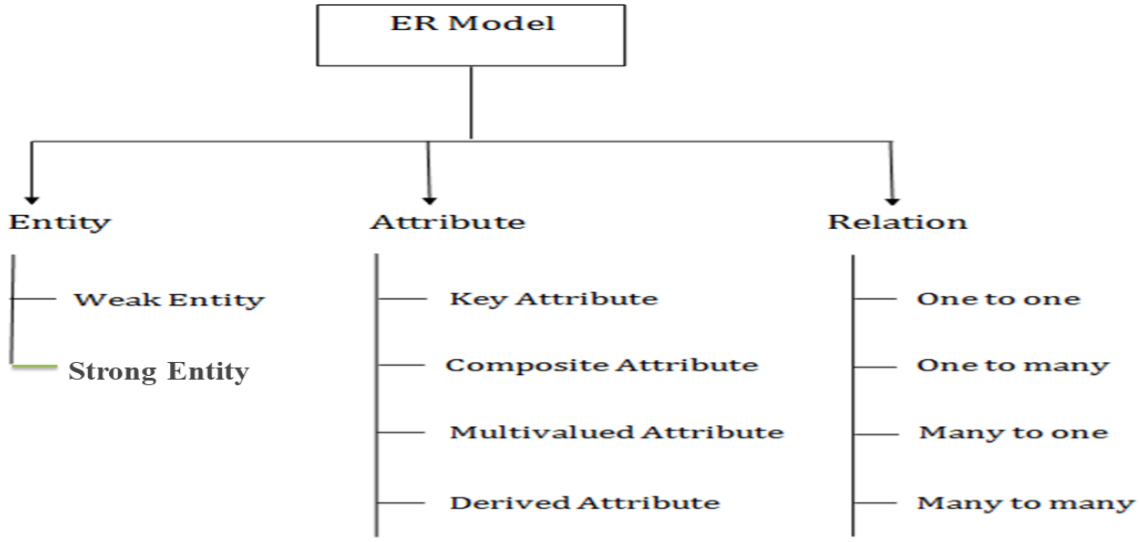
- The **entity-relationship (ER) data model** allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

- It provides useful concepts that allow us to move front an informal description of what users want from their database to a more detailed, precise description that can be implemented in a DBMS.

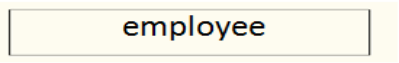
History of ER models

- Peter Chen proposed ER Diagrams in 1971 to create a uniform convention that can be used as a conceptual modeling tool.
- ER diagrams are used to model and design relational databases.

Components of ER Diagram:



Entity and Entity Set:

- An **entity** is an object that exists and is distinguishable from other objects. Entities are objects of physical(Person, place, thing) or conceptual existence (sales, concert etc.)
  - Examples:
    - Person: PROFESSOR, STUDENT
    - Place: STORE, UNIVERSITY
    - Object: MACHINE, BUILDING
    - Event: SALE, REGISTRATION
- An **entity** is represented with a RECTANGLE 

An **entity set** is a set of entities of the same type that share the same properties.

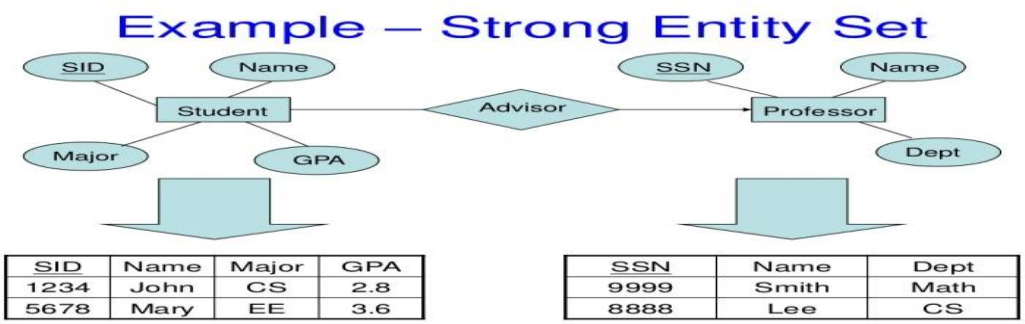
Example: set of all persons, companies, trees, holidays

Strong Entity

The Strong Entity is the one whose existence does not depend on the existence of any other entity in a schema. It is denoted by a single rectangle. A strong entity always has the **primary key** in the set of attributes that describes the strong entity. It indicates that each entity in a strong entity set can be uniquely identified.

Set of similar types of strong entities together forms the Strong Entity Set. A strong entity holds the relationship with the weak entity via an **Identifying Relationship**, which is denoted by **double diamond in the ER diagram**. On the other hands, the relationship

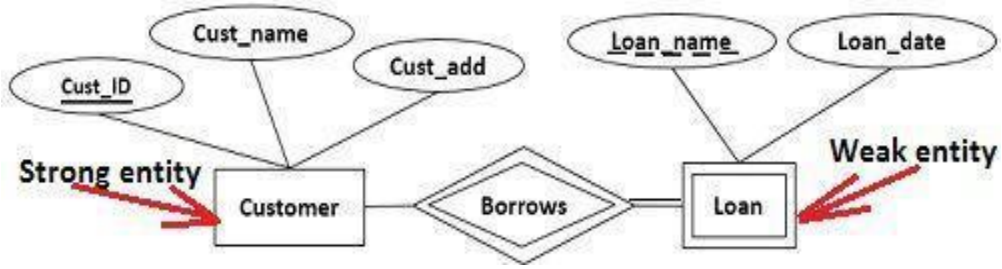
between two strong entities is denoted by a single diamond and it is simply called as a relationship.



Weak Entity

A Weak entity is the one that depends on its owner entity i.e. a strong entity for its existence. A weak entity is denoted by **the double rectangle.** Weak entities do not have the primary key instead it has a partial key that uniquely discriminates the weak entities. The primary key of a weak entity is a composite key formed from the primary key of the strong entity and partial key of the weak entity.

The collection of similar weak entities is called Weak Entity Set. The relationship between a weak entity and a strong entity is always denoted with an Identifying Relationship i.e. double diamond.



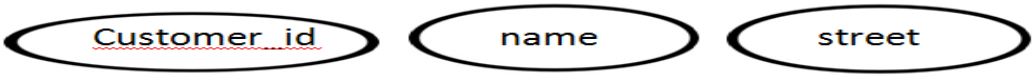
ATTRIBUTES

- An entity is represented by a set of attributes that is descriptive properties possessed by all members of an entity set.

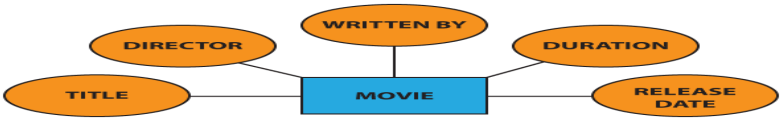
  Example:

  - *customer* = (*Customer_id, name, street, city, salary*)
  - *movie*= (*title, director, written by, duration, release date*)

Attributes are represented with **ELLIPSE**

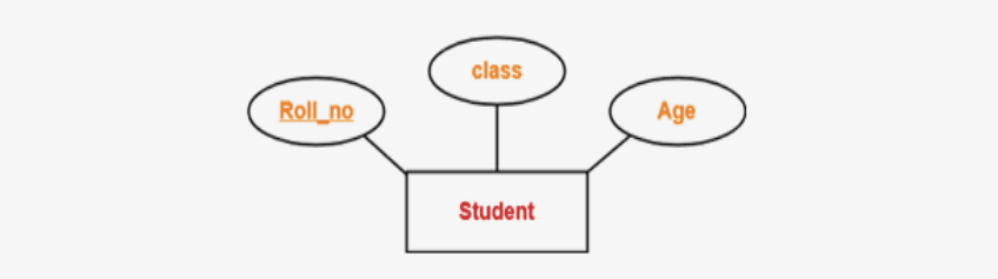


Use **LINES** to link attributes to entities

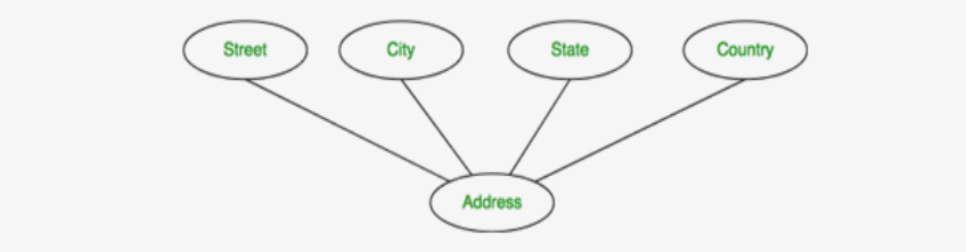


An attribute can be of many types, here are different types of attributes defined in ER database model:

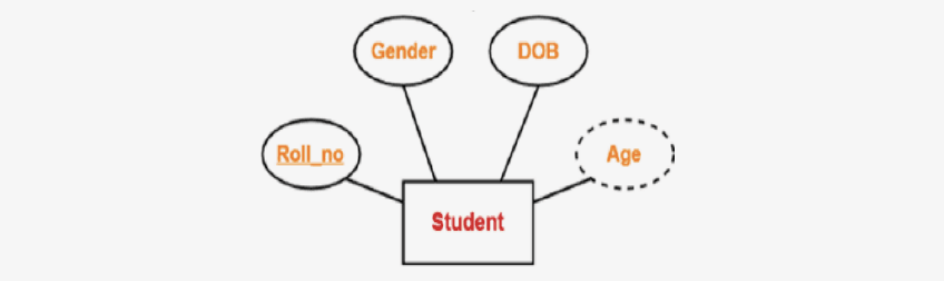
1. **Simple attribute**: The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, student's age.



2. **Composite attribute**: A composite attribute is made up of more than one simple attribute. For example, student's address will contain, house no., street name, pin code etc.

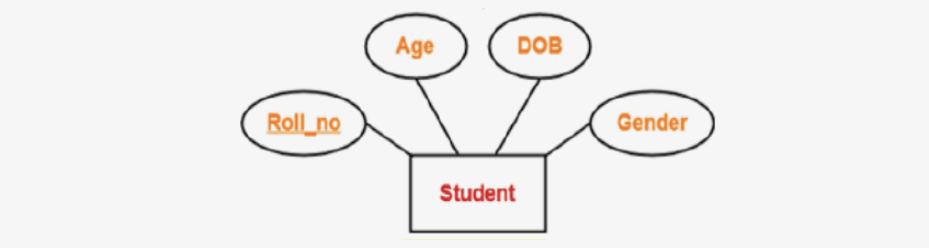


3. **Derived attribute**: These are the attributes which are not present in the whole database management system, but are derived using other attributes. For example, *average age of students in a class*.

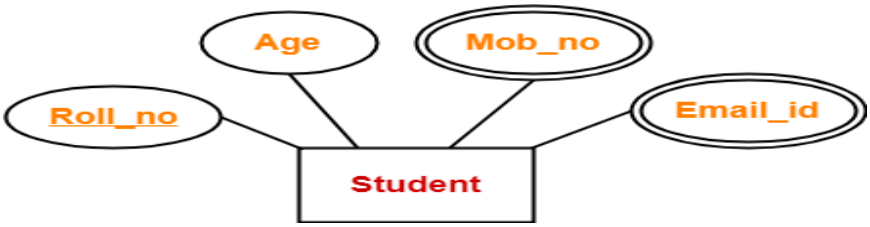


Example: age, and its value is derived from the stored attribute Date of Birth.

4. **Single-valued attribute**: As the name suggests, they have a single value.

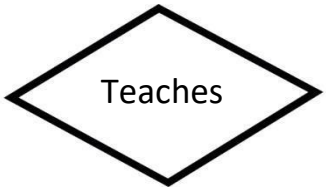


5. **Multi-valued attribute**: They can have multiple values.

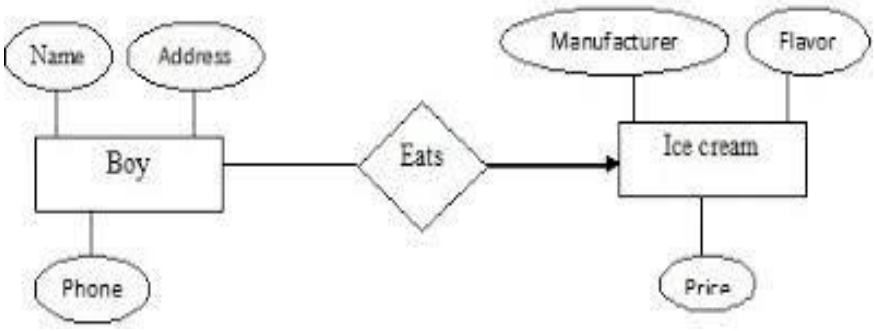


Relationship

- It is an association between two entities. It shows how the two entities are connected.
- Named set of all similar relationships with the same attributes and relating to the same entity types
- Relationship is represented with DIAMOND



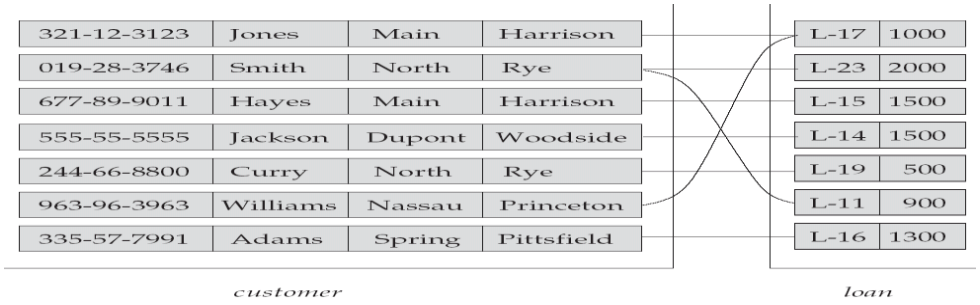
- Relationships relate entities within the entity sets involved in the relationship type to each other.



ER – Relationships

Relationship Set: Collection of similar relationships.

- An attribute can also be property of a relationship set. For instance, the depositor relationship set between entity sets customer and account may have the attribute access-date

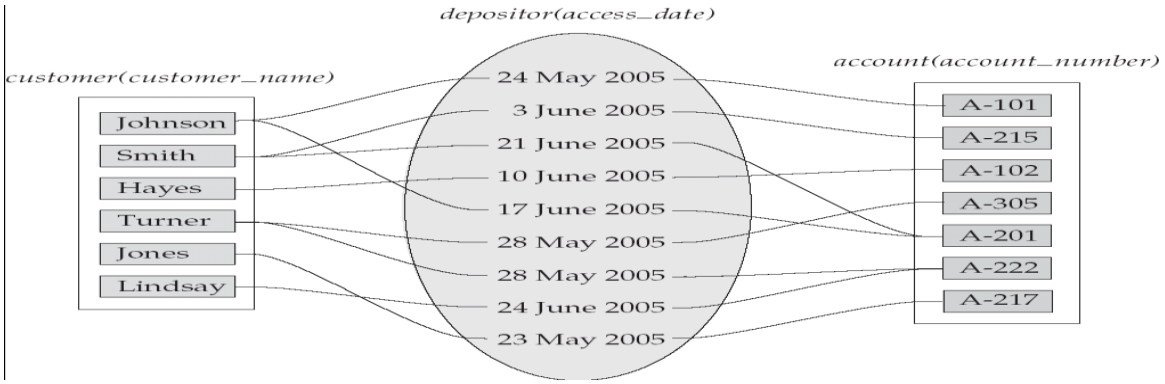


321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

customer

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

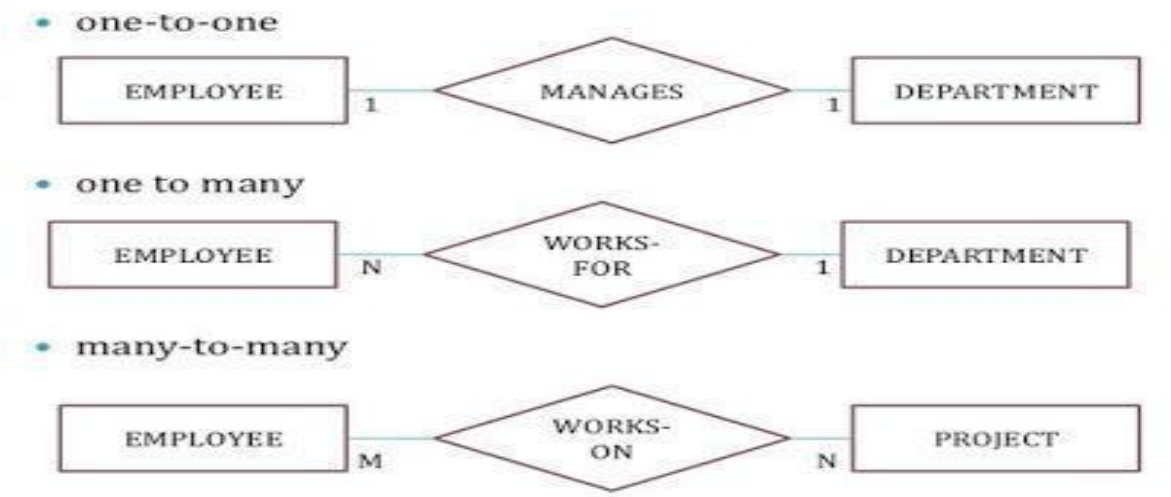
loan



Mapping Cardinality

Types of Relationships

- Three types of relationships can exist between entities
- One-to-one relationship (1:1): One instance in an entity (parent) refers to one and only one instance in the related entity (child).
- One-to-many relationship (1:M): One instance in an entity (parent) refers to one or more instances in the related entity (child)
- Many-to-many relationship (M:N): exists when one instance of the first entity (parent) can relate to many instances of the second entity (child), and one instance of the second entity can relate to many instances of the first entity.



DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 6
Database Design and E R Model

Degree of Relationship

A Relationship describes relation between entities. Relationship is represented using diamonds or rhombus.

Degree of relationship represents the number of entity types that associate in a relationship.

Types:

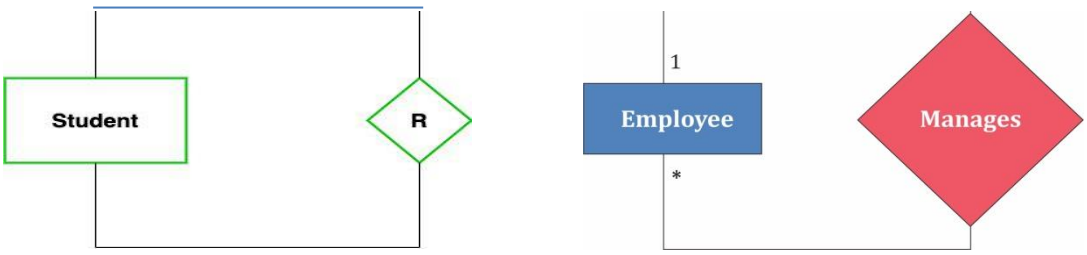
Now, based on the number of linked entity types, we have 4 types of degrees of relationships.

1. Unary Relationship
2. Binary Relationship
3. Ternary Relationship
4. N-ary Relationship

1. Unary Relationship

In this type of relationship, both the associating entity type are the same. in other words, in a relation only one entity set is participating then such type of relationship is known as a unary relationship or Recursive Relationship

Example: In a particular class, we have many students, there are monitors too. So, here class monitors are also students.



2. Binary Relationship

In a Binary relationship, there are two types of entity associates.

In other words, in a relation when two entity sets are participating then such type of relationship is known as a binary relationship. This is the most used relationship and one can easily be converted into a relational table.

This is further divided into three types.

One to One Relationship
This type of relationship is rarely seen in real world

The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in real-world relationships.

One to Many Relationships
The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.

Many to One Relationship
It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.

Many to Many Relationships
A many-to-many relationship occurs when multiple records in a table are associated with multiple records in another table.

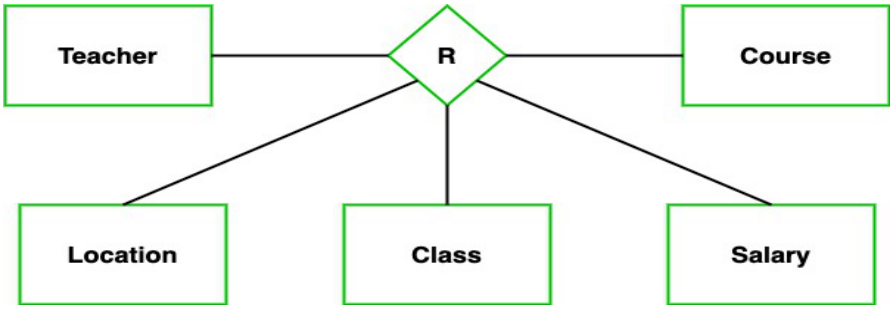
3. Ternary Relationship
Relationship of degree three is called Ternary relationship.
A Ternary relationship involves three entities. In such relationships we always consider two entities together and then look upon the third.

- The above relationship involves 3 entities.
- Company operates in Sector, producing some Products.

4.. n-ary Relationship

In the N-ary relationship, there are n types of entity that associates. There is one limitation of the N-ary relationship, as there are many entities so it is very hard to convert into an entity, rational table.

Example: We have 5 entities Teacher, Class, Location, Salary, Course. So, here five entity types are associating we can say an n-ary relationship is 5.



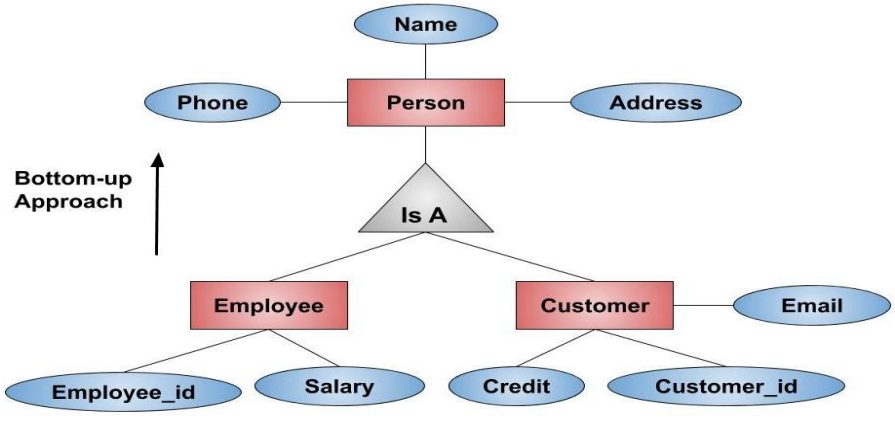
ADDITIONAL FEATURES OF THE ER MODEL

Class Hierarchies:

Using the ER model for bigger data creates a lot of complexity while designing a database model, So in order to minimize the complexity Generalization, Specialization, and Aggregation were introduced in the ER model.

1. **Generalization** –
   Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a **bottom-up approach** in which two or more entities can be generalized to a higher-level entity if they have some attributes in common.

ISA (`is a') Hierarchies is used to represent Generalizatio - referred as a superclass-subclass relationship.

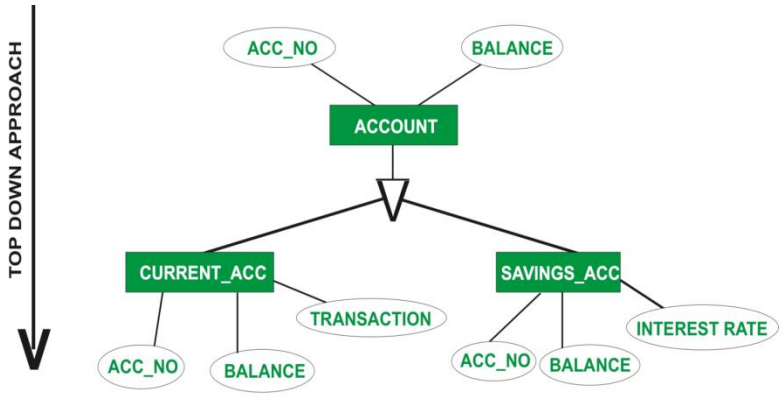


2. **Specialization** –

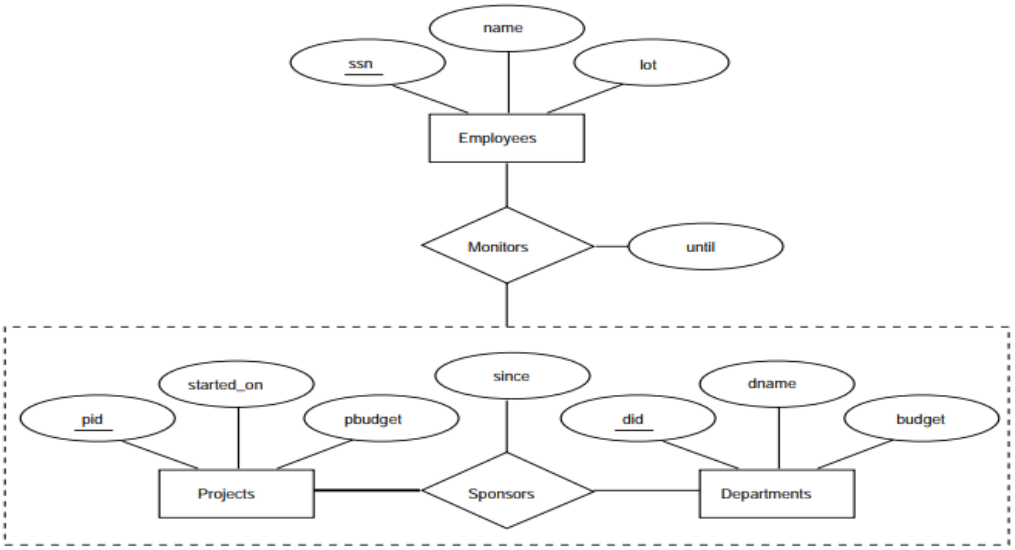
Specialization is a **top-down approach**, and it is opposite to Generalization.
In specialization, one higher level entity can be broken down into two lower level entities.
Specialization is used to identify the subset of an entity set that shares some distinguishing characteristics.

Normally, the superclass is defined first, the subclass and its related attributes are defined next, and relationship set are then added.

ISA (`is a') Hierarchies is used to represent Specialization - referred as a superclass-subclass relationship.



3.Aggregation:
Sometimes we have to model a relationship between a collection of entities and relationships. There is a limitation with E-R model that it cannot express relationships among relationships. Aggregation allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set.
So aggregation is an abstraction through which relationship is treated as higher level entities.



Participation Constraints:

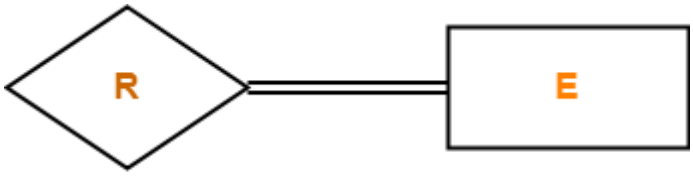
Participation constraints define the least number of relationship instances in which an entity must compulsorily participate.

Types of Participation Constraints:

There are two types of participation constraints- Total Particiapation
Partial Participation

1. Total Participation

- It specifies that each entity in the entity set must compulsorily participate in at least one relationship instance in that relationship set.
- That is why, it is also called **as mandatory participation**.
- Total participation is represented using a double line between the entity set and relationship set.

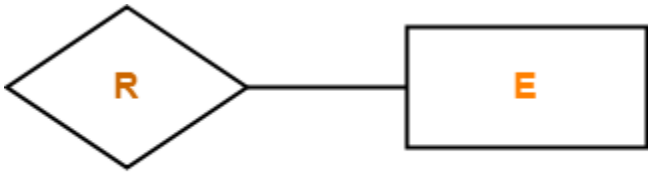


Total Participation

- Double line between the entity set “Student” and relationship set “Enrolled in” signifies total participation.
- It specifies that each student must be enrolled in at least one course.

2. Partial Participation

- It specifies that each entity in the entity set may or may not participate in the relationship instance in that relationship set.
- That is why, it is also called as **optional participation.**
- Partial participation is represented using a single line between the entity set and relationship set.



Partial Participation

- Single line between the entity set “Course” and relationship set “Enrolled in” signifies partial participation.

- It specifies that there might exist some courses for which no enrollments are made.

DATABASE MANAGEMENT SYSTEMS
UNIT 1 – TOPIC 7
Conceptual Design Using the ER Model

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets?
- Should we use binary or ternary relationships?
- Should we use aggregation?

1. Entity vs. Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set.

Example : Consider the relationship "Employee WorksIn Department", if we want to add address information to the Employee entity set.

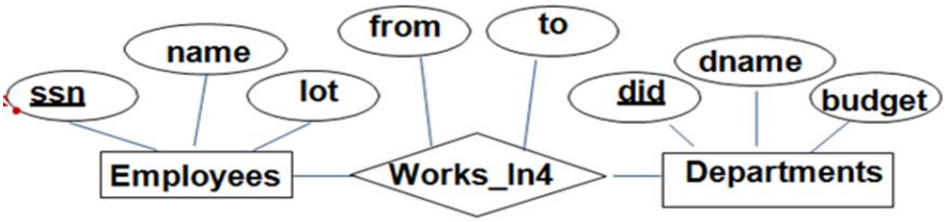
Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?

Now this depends upon the use we want to make of address information, and the semantics of the data:

- If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued) and an association between employees and addresses using a relationship say Has_address.
- If the structure (city, state, zipcode etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity with these attributes (since attribute values are atomic).

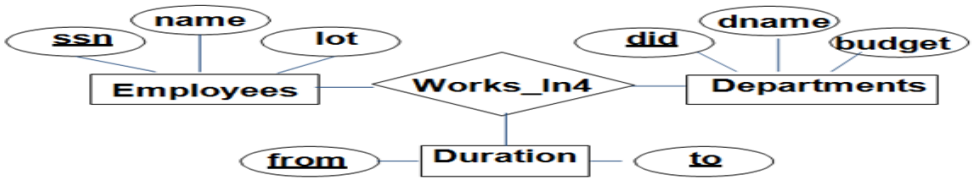
Consider another example below:

In the "Employee WorksIn Department" Relationship, it is possible for an employee to work in a given department over more than one period, which is not possible to accomodate as an attribute.



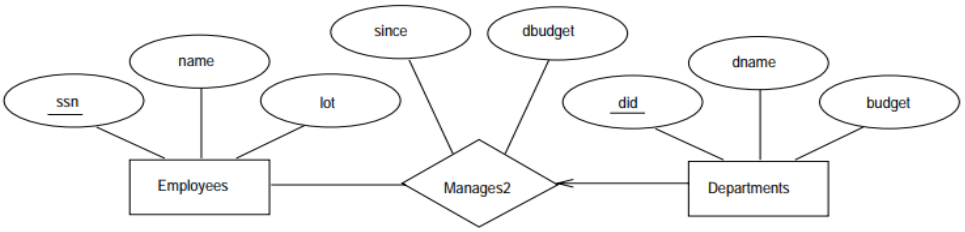
- Works_In4 does not allow an employee to work in a department for two or more periods.

- Similar to the problem of wanting to record several addresses for an employee, We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, **Duration.**



2. Entity vs. Relationship

Look at the example below : Consider the relationship set called Manages.



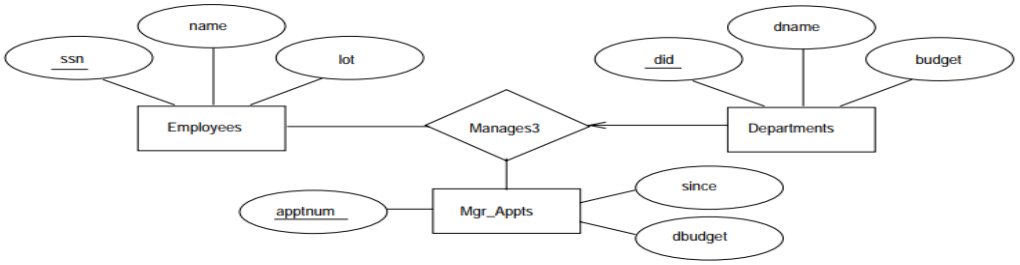
Suppose that each department manager is given a discretionary budget. There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair.

What if a manager manages more than one department and the discretionary budget is a sum of all the departments he manages.

Redundancy: given employee will have the same value in the dbudget field

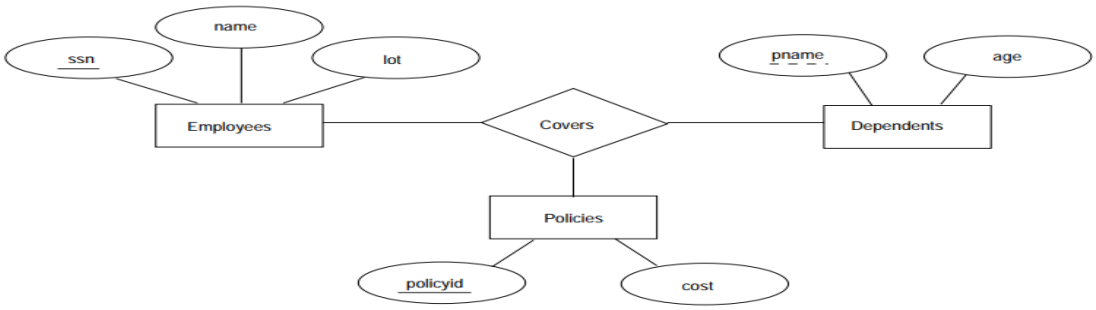
Misleading: Suggests dbudget associated with department - mgr combination.

We can address these problems by associating dbudget with the appointment of the employee as manager of a group of departments. In this approach, we model the appointment as an entity set, say Mgr_Appt, and use a ternary relationship, say Man ages3, to relate a manager, an appointment, and a department.



3. Binary vs. Ternary Relationships

Consider an ER Diagram that models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.



If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, the ternary relationship as above is inaccurate.

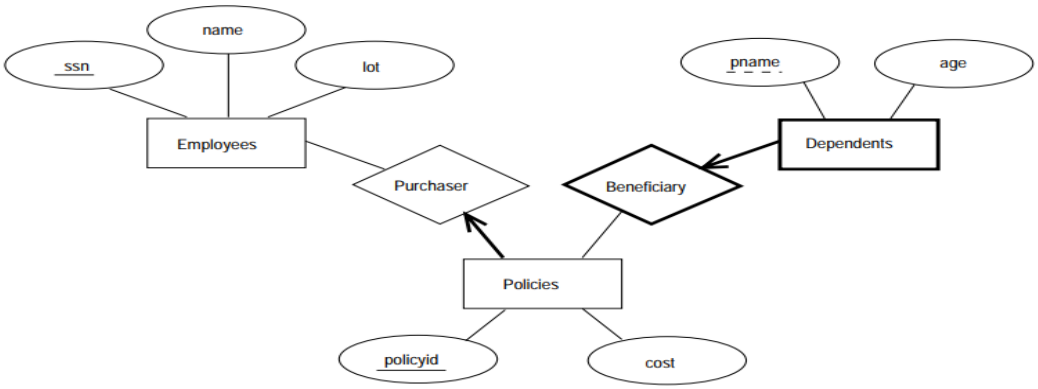
Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity

Solution can be:

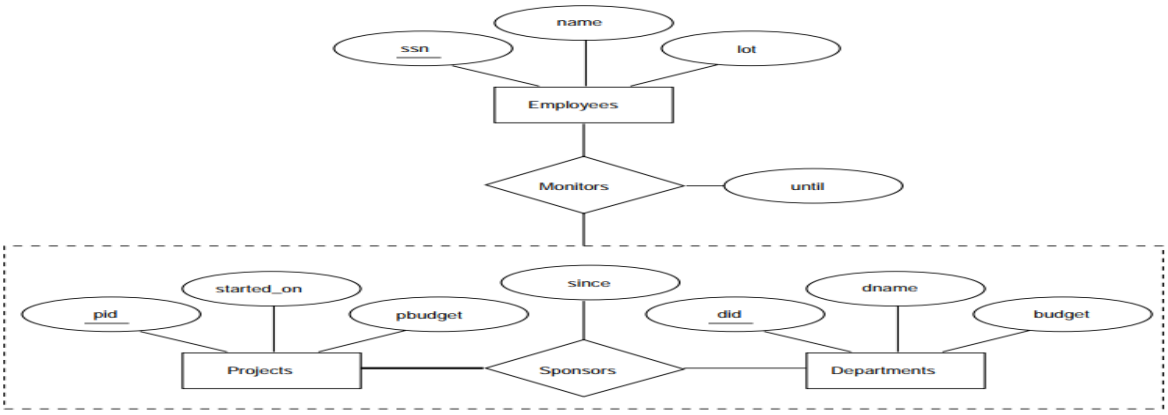
- The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent.
- The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent.
- The third requirement forces us to introduce an identifying relationship that is binary

The best way to model this situation is to use two binary relationships, as shown in Figure

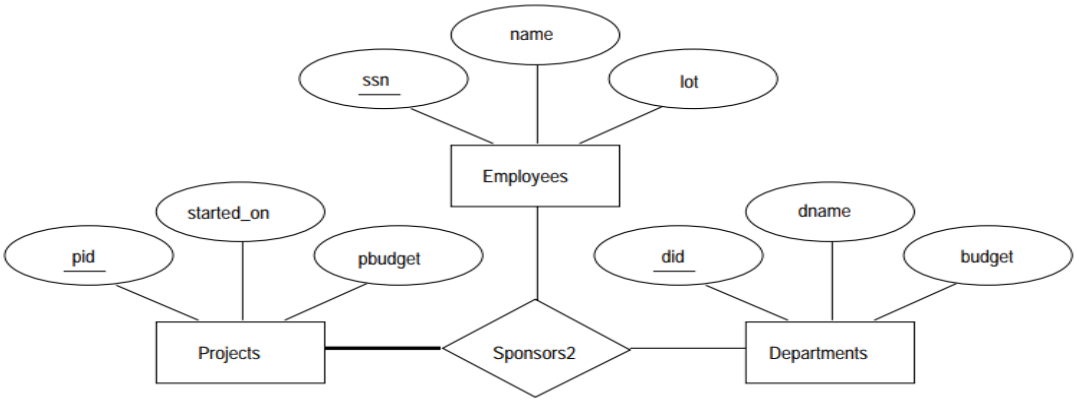


4. Aggregation v/s Ternary relationship

- The choice between using aggregation or ternary relationship is mainly determined by existence of a relationship that relates relationship set to entity set.
- The choice may also be guided by certain integrity constraints that we want to express.



According to the ER diagram shown in Figure above, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the '**until'** attribute of Monitors, then we might reasonably use a ternary relationship as shown below without the need for aggregation.



Consider the constraint that each sponsorship (of a project by a department) be mon itored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship Monitors in the aggregation ER diagram.

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 1
RELATIONAL MODEL

What is Relational Model?

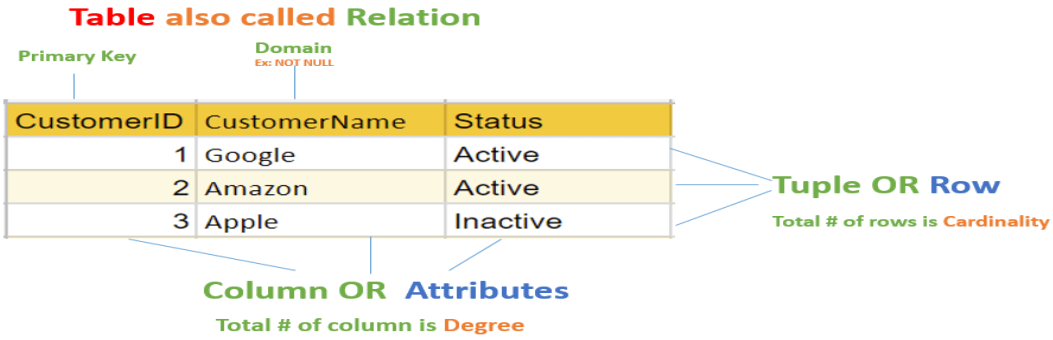
Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

Relational Model (RM) represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

Relational Model – Keywords / Terminologies

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME, etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree (Arity**): The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:** Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain
11. **Relational database** – A collection of relations with distinct relation names
12. **Relational database schema** - The collection of schemas for the relations in the database.



Relational Model Concepts:

1. **Data is Organized into Tables:** Data is organized into tables, also known as relations. Each table consists of rows and columns. Rows represent individual records, while columns represent attributes or fields.

2. **Tables Have Keys:** Each table has one or more columns that uniquely identify each row. These are called primary keys. Additionally, there can be foreign keys, which establish relationships between tables.

3. **Relationships Between Tables:** Tables can be related to each other through common attributes. These relationships can be one-to-one, one-to-many, or many-to-many.

4. **Data Integrity**: The relational model enforces data integrity through constraints such as primary key constraints, foreign key constraints, and other integrity rules. These constraints ensure that data remains consistent and accurate.

5. **Structured Query Language (SQL):** The relational model is typically manipulated using a special-purpose language called SQL (Structured Query Language). SQL provides a standardized way to perform operations such as querying, updating, and deleting data in relational databases.

6. **Normalization:** The process of organizing data to minimize redundancy and dependency is called normalization. It involves breaking down large tables into smaller ones and defining relationships between them to reduce redundancy and improve data integrity.

7. **Transactions:** Relational databases support transactions, which are units of work that must be performed atomically (all or nothing), consistently (in a valid state), isolated (separate from other transactions), and durably (persistently stored).

DATABASE MANAGEMENT SYSTEMS

UNIT II – TOPIC 1

Data Definition Language - DDL

Structured query language (SQL) is a programming language for storing and processing information in a relational database.

We can use SQL statements to store, update, remove, search, and retrieve information from the database and also to maintain and optimize database performance.

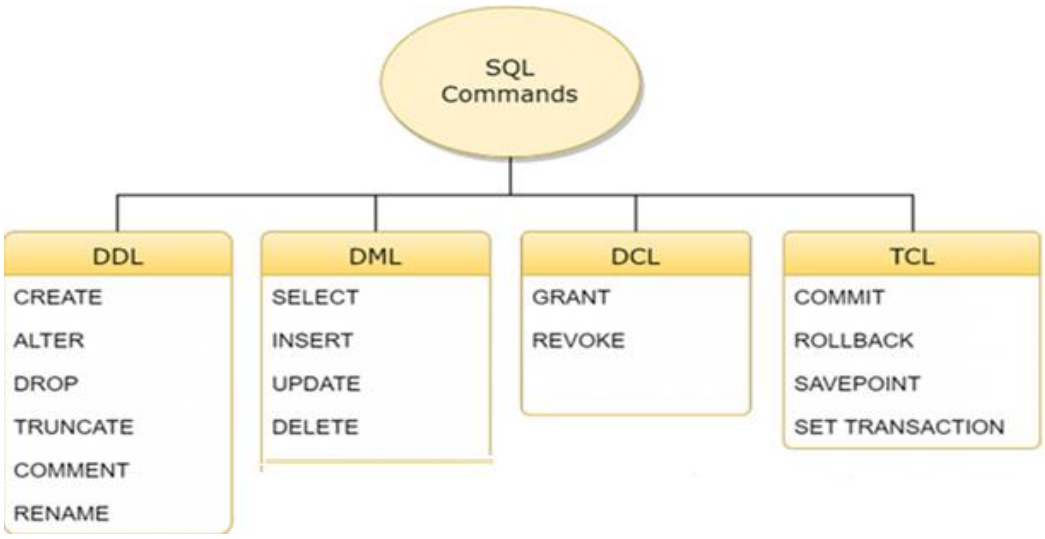
SQL was invented in the 1970s based on the relational data model. It was initially known as the Structured English Query Language (SEQUEL).

This query language became the standard of ANSI in the year of 1986 and ISO in the year of 1987.

Types of SQL Commands:

Here are five types of widely used SQL queries.

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Query Language
- Data Control Language (DCL)
- Transaction Control Language (TCL)



What is DDL?

Data definition language (DDL) refers to SQL commands that design the database structure. i.e. it deals with **database schemas and descriptions**, of how the data should reside in the database.Database engineers use DDL to create and modify database objects based on the business requirements. For example, the database engineer uses the CREATE command to create database objects such as tables, views, and indexes.

CREATE:

To create a database and its objects like (table, index, views, store procedure, function, and triggers)

Syntax: CREATE Objtype Objname;

CREATE DATABASE

Syntax : create database database_name;

Ex: create database testdb;

Use the SHOW statement to find out what databases currently exist on the server:

Syntax: SHOW DATABASES;

Ex: show databases;

We have to select the database on which we want to run the database queries.

syntax : USE DATABASE_NAME;

Ex: use testdb;

Creating a Table:

The CREATE TABLE statement is used to create a new table in a database.

Syntax:

```
CREATE TABLE table_name (
column1 datatype(size),
column2 datatype(size),
column3 datatype(size),
....
);
```

Example: create table emp (eid int(5), ename varchar(15));

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold

(e.g. varchar, integer, date, etc.)

Create Table using Another Table :

A copy of an existing table can also be created using CREATE TABLE.

- The new table gets the same column definitions. All columns or specific columns can be selected.
- If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax:

```
CREATE TABLE new_table_name AS
   SELECT column1, column2,...
   FROM existing_table_name
   WHERE ....;
```

Example: The following SQL creates a new table called “DupCust" (which is a copy of the "Customers" table):

```
CREATE TABLE DupCust  AS
SELECT customername, contactno
FROM customers;
```

Data Types in MySQL:

In MySQL there are three main data types: **string, numeric, and date and time**

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large OBjects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric Data Types

Data type	Description
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Note: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

DROP

Drops commands remove tables and databases from RDBMS.

To drop a Table

Syntax: DROP TABLE tablename;

Ex: DROP TABLE emp ;

To drop a database

syntax: DROP DATABASE databasename;

Ex: Drop database kmitcsma;

TRUNCATE:

This command is used to delete all the rows from the table and free the space containing the table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE table students

ALTER:

1. The ALTER TABLE statement is used to modify the structure of table, i.e. **add, delete, or modify columns** in an existing table.
2. The ALTER TABLE statement is also used to add and drop various constraints on an existing table.
   ALTER- ADD
   ALTER- MODIFY
   ALTER-DROP

To add a new column to the Table :

Syntax:

ALTER TABLE table_name ADD (columnname-1 datatype(size), columnname-2 datatype(size)….,columnname-n datatype(size));

Ex: Alter Table emp add (dob date, mobile int(10));

To Drop a column from an existing table:

Syntax:

ALTER TABLE tablename DROP column columnname;

Ex: alter table emp drop column mobile;

To modify a column in the Table :

Syntax:

ALTER TABLE table_name MODIFY columnname-1 datatype(size);

Ex: Alter Table emp modify ename varchar(25);

To add a constraint:

Syntax:

ALTER TABLE tablename ADD constrainttype (columnname);

Ex: alter table emp add primary key(eid);

RENAME:

Renaming the existing table:

RENAME TABLE table_name to new table_name;

Ex:

mysql> rename table persons to emp;
Query OK, 0 rows affected (3.17 sec)

Renaming the column name in an existing table:

ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;

Ex: ALTER TABLE student1
RENAME COLUMN jdateTO joingdate;

DESCRIBE :

This command is used to describe the structure of an existing table

Knowing the structure of a table:

DESCRIBE table_name;
OR
DESC table_name

Ex: mysql> create table emp (eid int(5), ename varchar(15));
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> describe emp;

```
mysql> desc emp;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| eid   | int         | YES  |     | NULL    |       |
| ename | varchar(15) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

COMMENT:

Comments are used to explain sections of SQL statements.

Comments can be written in the following three formats:

1.Single-line comments

2.Multi-line comments

3.In-line comments

1.Single Line Comments

Comments starting and ending in a single line are considered single-line comments. A line starting with '- - is a comment and will not be executed.

Ex: create database testdb; --THIS IS SINGLE LINE COMMENT:

2. Multi-Line Comments

Comments starting in one line and ending in different lines are considered as multi-line comments.

A line starting with '/*' is considered as starting point of the comment and is terminated when '*/' is encountered.

```
Ex: /*This is for multiline comments
 specifying the database is created:*/
Create database kmittest;
```

3. In-Line Comments

In-line comments are an extension of multi-line comments, comments can be stated in between the statements and are enclosed in between '/*' and '*/'. Bypassed by /

EX:

```
SELECT ename
 /* This column contains the name of the customer / order_date /
 This column contains the date the order was placed */
 FROM emp
```

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 3
INTEGRITY CONSTRAINTS

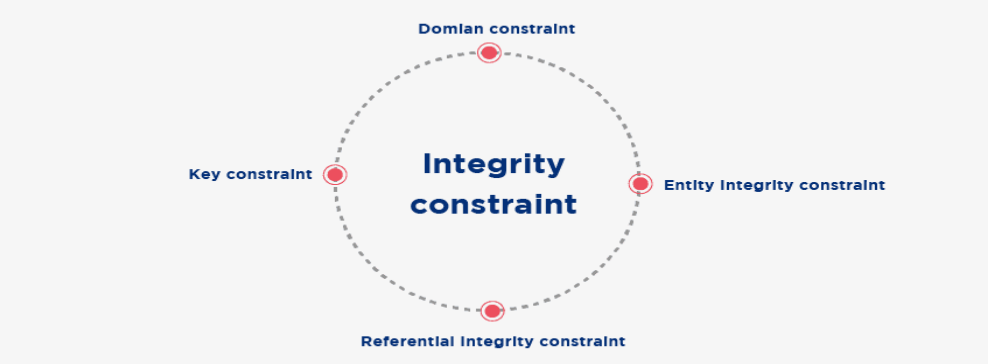
Integrity constraints are the set of predefined rules in a database to maintain data accuracy, consistency and reliability. In short, it is used to maintain the quality of information. They can be used to enforce business rules or to ensure that data is entered correctly. Integrity constraints can also be used to enforce relationships between tables. They ensure that the data in the database adheres to a set of rules, which can help prevent errors and inconsistencies.

Integrity Constraints act as guidelines ensuring that data in the database remain accurate and consistent. They guard against accidental damage to the database.

Integrity constraints in SQL can be either enforced by the database system or by application code. Enforcing them at the database level can help ensure that the rules are always followed, even if the application code is changed. However, enforcing them at the application level can give the developer more flexibility in how the rules are enforced.

Types of Integrity Constraint

There are four main types of integrity constraints: domain, entity, referential, and key.



1. Domain constraints

Domain constraints can be defined as the definition of a valid set of values for an attribute. The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
102	Krishana	Analyst	2Lakhs	Sec	24

Here for Eid 102, Salary attribute violates the domain constraint as it is allowed to have only numeric values.

2. Entity integrity constraints

An entity integrity constraint is a restriction on null values. Null values are values that are unknown or not applicable, and they can be problematic because they can lead to inaccurate results. This constraint serves unique identification of each tuple in the relation.

The primary key attribute is used to identify individual rows in relation and entity integrity constraint states that it cannot be NULL. However a table can contain a NULL value in other fields / attributes.

In the employee table below Eid is primary key and can't contain NULL value.

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
	Krishana	Analyst	95600	Pune	24

3. Referential Integrity Constraints (Foreign Key)

A referential integrity constraint is be used to enforce relationships between tables. The foreign key a constraint is a column or list of columns that points to the primary key column of another table.

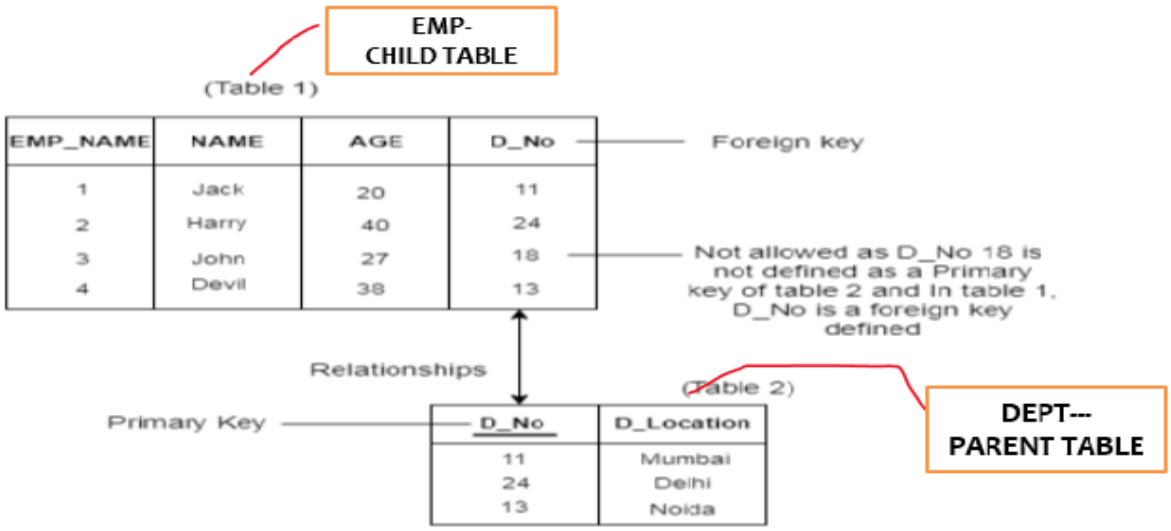
In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

The rules are:

- can't delete a record from a primary table if matching records exist in a related table.
- can't change a primary key value in the primary table if that record has related records.
- can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
- However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

Example:



4. **Key Constraints**

Key constraints in DBMS are a restriction on duplicate values. A key is composed of one or more columns whose values uniquely identify each row in the table. These are called uniqueness constraints since it ensures that every tuple in the relation should be unique.

Keys are the entity set that is used to identify an entity within its entity set uniquely.
A entity set can have multiple keys or candidate keys (minimal superkey), out of which we choose one of the keys as the primary key.
Primary key is always unique and cannot have null values in the relational table.

Example:

In the below example Eid is primary key but Eid have the same value 100. so, it is violating the key constraint

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
100	Krishana	Analyst	95600	Pune	24

Other Key Constraints:

1. **NOT NULL Constraints**
2. **NULL Constraints**
3. **PRIMARY KEY Constraints**
4. **FOREIGN KEY Constraints**
5. **UNIQUE Constraints**
6. **CHECK Constraints**

NOT NULL Constraint:

The NOT NULL constraint in SQL is used to ensure that a column in a table doesn't contain NULL (empty) values, and prevent any attempts to insert or update rows with NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values, which means that you cannot insert a new record, or update a record without adding a value to this field.

NULL represents a record where data may be missing data or data for that record may be optional. By default a column is allowed to have NULL values, but the NOT NULL constraint enforces a column to NOT accept NULL values

A not-null constraint cannot be applied at table level

NOT NULL on CREATE TABLE

```
CREATE TABLE Persons (
   PID int (3) NOT NULL,
   PName varchar (10) NOT NULL,
   PADD varchar (20) NOT NULL,
   PAge int (2)
 );
```

NULL Constraint:

- By default, a column can hold NULL values

The term **NULL** in SQL is used to specify that a data value does not exist in the database. It is not the same as an empty string or a value of zero, and it signifies the absence of a value or the unknown value of a data field.

Some common reasons why a value may be NULL −

- The value may not be provided during the data entry.
- The value is not yet known.

It is important to understand that you cannot use comparison operators such as "=", "<", or ">" with NULL values. This is because the NULL values are unknown and could represent any value. Instead, you must use "IS NULL" or "IS NOT NULL" operators to check if a value is NULL

Syntax

The basic syntax of **NULL** while creating a table.

```
CREATE TABLE CUSTOMERS (
        ID   INT(3)          NOT NULL,
        NAME VARCHAR (20)    NOT NULL,
        AGE  INT(2)          NOT NULL,
        ADDRESS  VARCHAR (25) ,
        SALARY   FLOAT(6, 2),
   );
```

In the above example SALARY and ADDRESS are NULL constraints.

PRIMARY KEY Constraint :

The PRIMARY KEY constraint uniquely identifies each record in a table.

- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
CREATE TABLE Persons(
    PID int(3) NOT NULL,
    PName varchar(10) NOT NULL,
    PADD varchar(20) NOT NULL,
    PAge int (2),
    PRIMARY KEY (PID)
);
```

FOREIGN KEY Constraint :

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

FOREIGN KEY constraints enforce referential integrity.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

The rules are:

- can't delete a record from a primary table if matching records exist in a related table.
- can't change a primary key value in the primary table if that record has related records.
- can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
- However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

```
CREATE TABLE dept (
        deptno int(3),
        dname varchar(20),
        location varchar(20),
        Primary key(deptno) );      -- PARENT TABLE OR Primary table

CREATE TABLE Emp (
        eid int(3),
        ename varchar(20),
        salary float(6,2),d
        no int(3),
        PRIMARY KEY  (Eid),
        FOREIGN KEY (dno) REFERENCES dept(deptno) );     -- CHILD TABLE
```

UNIQUE Constraint:

The UNIQUE constraint in SQL is used to ensure that no duplicate values will be inserted into a specific column or combination of columns that are participating in the UNIQUE constraint and not part of the PRIMARY KEY.
In other words, the index that is automatically created when you define a UNIQUE constraint will guarantee that no two rows in that table can have the same value for the columns participating in that index, with the ability to insert only NULL value to these columns.

The UNIQUE constraint ensures that all values in a column are different.
Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns
A PRIMARY KEY constraint automatically has a UNIQUE constraint.

Difference between UNIQUE constraints and PRIMARY KEY constraint is unique allows NULL values whereas primary key not allowed the NULL values.

```
CREATE TABLE Persons(
    PID int(3) UNIQUE,
    PName varchar(10) NOT NULL,
    PADD varchar(20),
    PAge int (2),
    Padhar varchar(12)
    PRIMARY KEY (Padhar) );
```

CHECK Constraint:

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (
    PID int(3),
    PName varchar(10) NOT NULL,
    PAdd varchar(20) NOT NULL,
    PAge int (2),
    PRIMARY KEY(PID),
    CHECK (PAge>=18)  );
```

If we enter the Page below 18 then shows that CHECK Constraint is violated.

```
CREATE TABLE Emp(
    EID int(3),
    EName varchar(20) NOT NULL,
    Doj date,
    Mobile bigint CHECK(Length(Mobile)=10),
    PRIMARY KEY(PID) );
```

Check constraint here ensures that mobile number is exactly 10 digits.

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 4
LOGICAL DATABASE DESIGN

ER Diagram to Relational Model Conversion

The ER model is convenient for representing an initial, high-level database design. First step of any relational database design is to make ER Diagram for it and then convert it into relational Model.

Given an ER diagram describing a database, there is a standard approach to generating a relational database schema that closely approximates the ER design.

What is relational model?
Relational Model represents how data is stored in database in the form of table

Lets learn step by step how to convert ER diagram into relational model.

Logical database design is the process of mapping or translating the conceptual model (ER Diagrams) into a relational model (table). i.e. deciding how to arrange the attributes of the entities in a given business environment into database structures, such as the tables of a relational database.

Let us see how to translate an ER diagram into a collection of tables with associated constraints, i.e., a relational database schema.

Three basic rules to convert ER into tables or relations:

Rule 1: *Entity:* Entity Names will automatically be table names

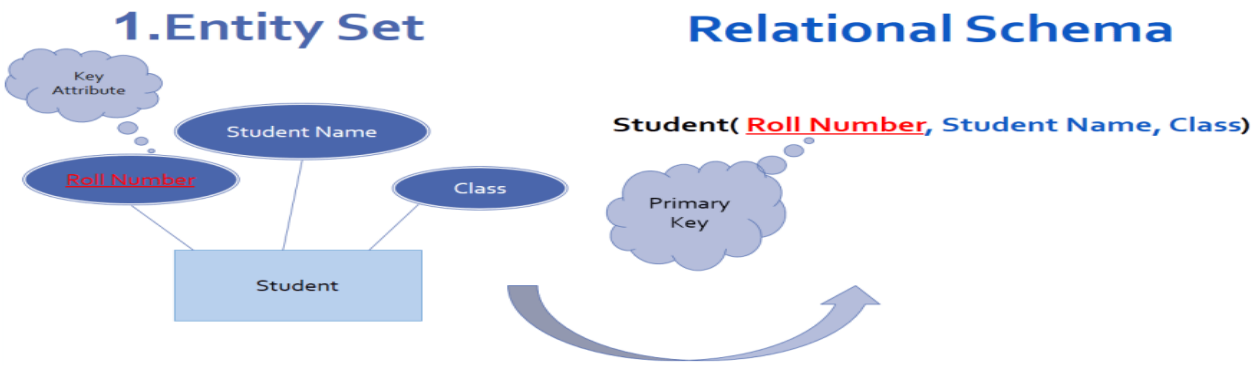
Rule 2: *Mapping of attributes*: attributes will be columns of the respective tables.

Rule 3: *Relationships*: relationship will be mapped by using a foreign key attribute. Foreign key is a primary or candidate key of one relation used to create association between tables.

1. Entity Set to Table:

Consider we have entity STUDENT in ER diagram with attributes Roll Number, Student Name and Class.

<u>To convert this entity set into relational schema</u>
1. Entity is mapped as relation in Relational schema
2. Attributes of Entity set are mapped as columns for that Relation.
3. Key attribute of Entity becomes Primary key for that Relation.



2. Entity Set with Multivalued Attributes:

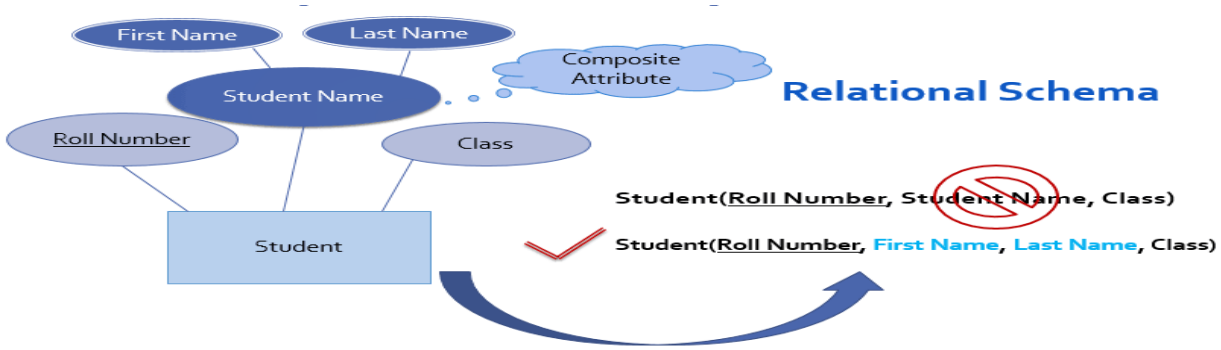
An entity may have several attributes which can take more than one value (eg: mobile number, hobby etc.). As it is not possible to represent multiple values in a single column in a relation, separate relation is created for multivalued attribute along with the key attribute of the relation.

1. Key attribute and multivalued attribute of entity set becomes composite key of new relation.
2. Separate relation employee is created with remaining attributes.

   This ensures instead of repeating all attributes of entity (for every value of the multivalued attribute) now only one attribute is need to repeat and that is the key attribute.

3. Entity set with Composite attribute:

Consider entity set student with attributes Roll Number, Student Name and Class. Here student name is composite attribute as it has further divided into First name, last name.
In this case to convert entity into relational schema,
composite attribute student name should not be include in relation but all parts of composite attribute are mapped as simple attributes for relation.



Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model.

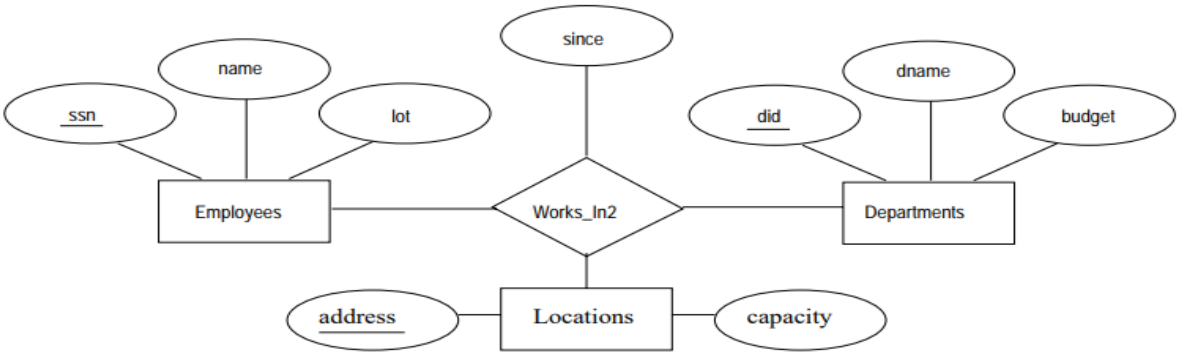
To represent a relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship.

Thus, the attributes of the relation include:

The primary key attributes of each participating entity set, as foreign key fields.

The descriptive attributes of the relationship set.

Example 1: Consider the ERD in the following figure. All the available information about the Works In2 table is captured by the following SQL definition:

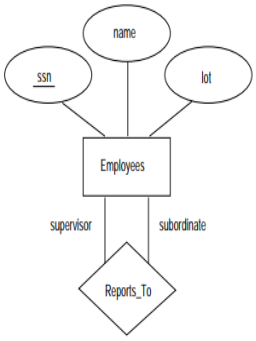


Create Tables for the participating entities Employee and Department with their respective attributes.

Create the table for the **Works_In2** relationship set as below

CREATE TABLE Works_In2 (ssn CHAR(11), did INTEGER, address CHAR(20), since DATE,
PRIMARY KEY (ssn, did, address),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (address) REFERENCES Locations,
FOREIGN KEY (did) REFERENCES Departments);

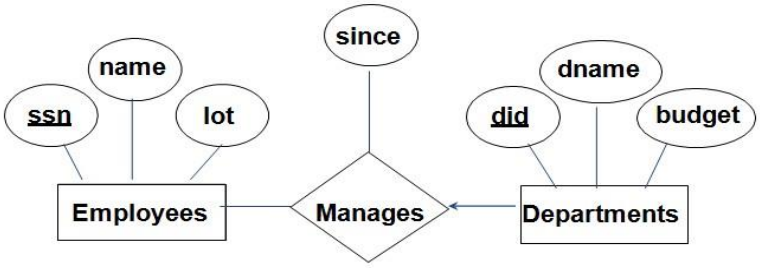
Example 2:
Consider another example:
Here the role indicators (supervisor/subordinate) are created as columns



CREATE TABLE Reports To (
Supervisor_ssn CHAR(11), subordinate_ssn CHAR(11),
PRIMARY KEY (supervisor ssn, subordinate ssn),
FOREIGN KEY (supervisor ssn) REFERENCES Employees(ssn),
FOREIGN KEY (subordinate ssn) REFERENCES Employees(ssn))

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

Relationship Sets (with Key Constraints) to Tables



Because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value.

A consequence of this observation is that *did* is itself a key for Manages

The Manages relation can be defined using the following SQL statement:

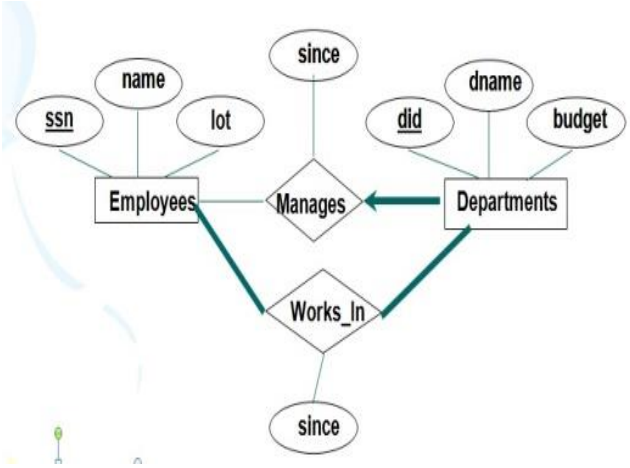
CREATE TABLE Manages(
ssn CHAR(11), did INTEGER, since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees (ssn),
FOREIGN KEY (did) REFERENCES Departments(did));

- **A second approach:** Since each department has a unique manager, we could instead combine Manages and Departments.
- This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations.
- The only drawback to this approach is that space could be wasted if several departments have no managers.
- In this case the added fields would have to be filled with *null* values.

The Dept_Mgr relation can be defined using the following SQL statement:

CREATE TABLE Dept_Mgr (
did INTEGER, dname CHAR(20), budget REAL, ssn CHAR(11), since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees(ssn))

Translating Relationship Sets with Participation Constraints:
Total participation constraint insists that every department is required to have a manager, and key constraint says every department will have at most one manager. Participation constraint is implemented using NOT NULL constraint



CREATE TABLE Dept_Mgr(
did INTEGER,
dname CHAR(20),
budget REAL,
ssn CHAR(11) NOT NULL,
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees(ssn),
ON DELETE NO ACTION));

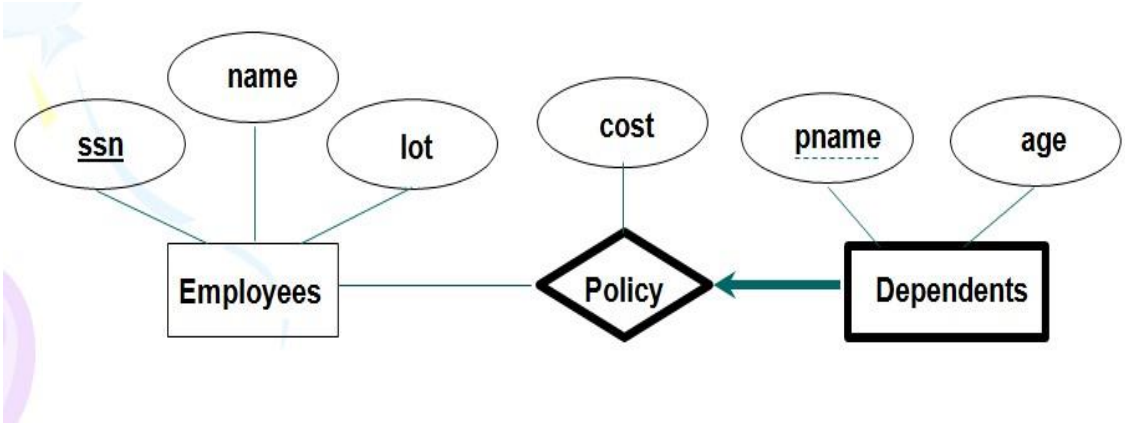
- It also captures the participation constraint that every department must have a manager:
- Because *ssn* cannot take on *null* values, each tuple of Dept_Mgr identifies a tuple in Employees (who is the manager).
- The NO ACTION specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept_Mgr tuple.
- If we wish to delete such an Employees tuple, we must first change the Dept_Mgr tuple to have a new employee as manager.

Translating a weak entity set:

A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation.

A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity.

The Dependents entity must be deleted if the owning Employees entity is deleted.



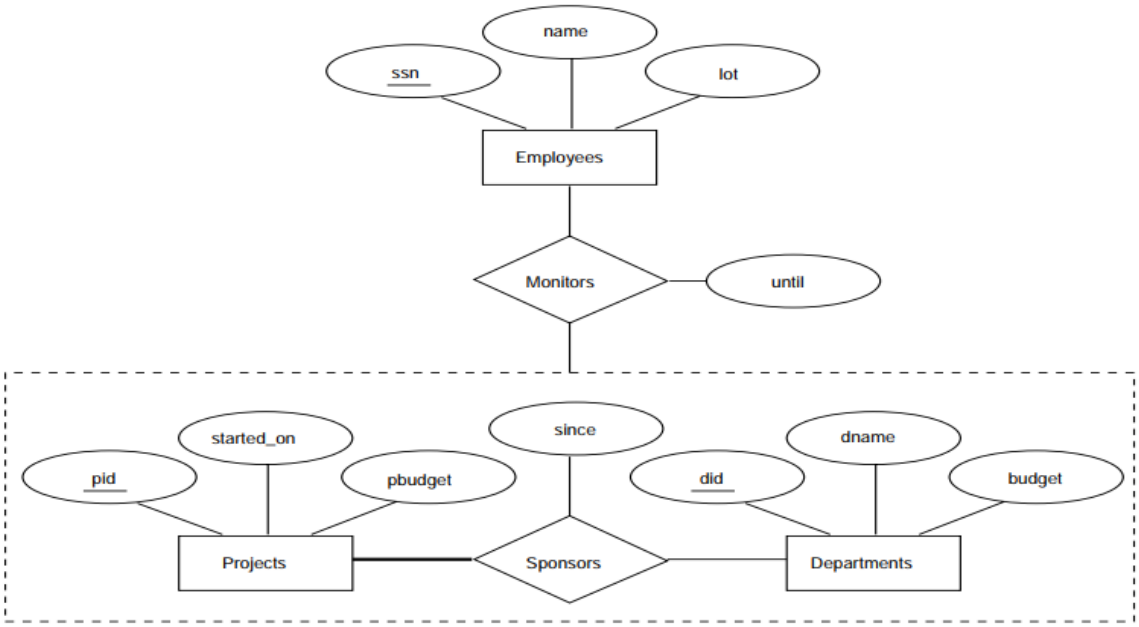
Weak entity set and identifying relationship set are translated into a single table.

When the owner entity is deleted, all owned weak entities must also be deleted.

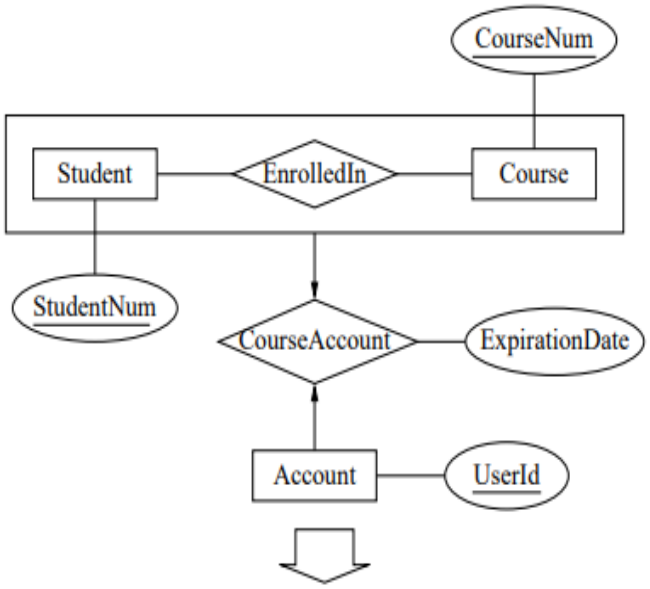
```
CREATE TABLE Dep_Policy (
Pname        CHAR(20),
age     INTEGER,
cost    REAL,
ssn     CHAR(11) NOT NULL,
PRIMARY KEY  (pname, ssn),
FOREIGN KEY   (ssn)  REFERENCES  Employees(ssn), ON DELETE CASCADE) )
```

Translating ER Diagrams with Aggregation:

- For the ER diagram shown in Figure, the Employees, Projects, and Departments entity sets and the Sponsors relationship set are mapped as explained before.
- For the **Monitors relationship set**, we create a relation with the following attributes:
  the key attributes of Employees (ssn), the key attributes of Sponsors (did, pid), and the descriptive attributes of Monitors (until).



Example:



Student

StudentNum

Course

CourseNum

Account

UserId

EnrolledIn

StudentNum	CourseNum

CourseAccount

UserId	StudentNum	CourseNum	ExpirationDate

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 5
INTRODUCTION TO VIEWS

Views in SQL

- A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition.
  - A relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.
  - Views in SQL are considered as a virtual table. A view also contains rows and columns.
  - To create the view, we can select the fields from one or more tables present in the database.
  - A view can either have specific rows based on certain condition or all the rows of a table.
  - Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s) when data redundancy is to be kept minimum while maintaining security

1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

Syntax:

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

Note: view_name: Name for the View,
table_name: Name of the table,
condition: Condition to select rows

Sample tables:
Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

Example : Creating View from a single table

In this example, we create a View named DetailsView from the table Student_Detail.

Query:

```
CREATE VIEW DetailsView AS
    SELECT NAME, ADDRESS
    FROM StudentDetails
    WHERE S_ID < 4;
```

Just like table query, we can query the view to view the data.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

Example : Creating View from multiple tables

View from multiple tables can be created by simply including multiple tables in the SELECT statement.

Syntax:

```
CREATE VIEW MarksView AS
SELECT Table1.colname1, Table1.colname2, Table2.colname3
FROM Table1,Table2
WHERE table1.colname4=table2.colname2;
```

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

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

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

2. Update SQL View:

The SQL view created can also be modified. We can do the following operations with the SQL VIEW.
But, **all views are not updatable.** A SQL view can be updated if the following conditions are satisfied.

1. The view is defined based on only one table.
2. The view should not have any field which is made of an **aggregate** function.
3. The view must not have GROUP BY, HAVING or DISTINCT clause in its definition.
4. The view should not be created using any nested query.
5. The selected output fields of the view must not use constants, string or value expressions.
6. If you want to update a view based on another view then that view should be updatable.

Updating a SQL View
We can use the **CREATE OR REPLACE VIEW** statement to **modify** the SQL view.

Syntax:

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

Example: If we want to update the view marksview1 and remove the attribute "*Age"* from in the view then the query would be:

Query

```
mysql> create or replace view marksview1 as select student_detail.name,
    student_marks.marks from student_detail, student_marks where
    student_detail.std_id=student_marks.std_id;
    Query OK, 0 rows affected (0.03 sec)
```

The above **CREATE OR REPLACE VIEW statement** would create a virtual table based on the result of the SELECT statement. Now, you can query the SQL VIEW as follows to see the output:

Output

```
mysql> select * from marksview1;
+---------+-------+
| name    | marks |
+---------+-------+
| stephan |    97 |
| kathrin |    86 |
| david   |    74 |
| alina   |    90 |
+---------+-------+
4 rows in set (0.03 sec)
```

Note:;

- Views are dynamically generated, hence everytime user queries the view it fetches the information from the original table.
- Rows inserted into an updatable view will insert the corresponding row into the original table.
- Attempts to insert rows through a view that doesnot contain Primary Key will be rejected as PK cannot take Null values.
- When views are created based on selection, the insertions may reflect in base table but may not be present in the view.

Deleting View

A view can be deleted using the Drop View statement.

Syntax

DROP VIEW view_name;

View_name: Name of the View which we want to delete.

Example:

If we want to delete the View **MarksView**, we can do this as:

EX: DROP VIEW MarksView;

NOTE: A view will also be dropped if a table is dropped using the CASCADE option.

Example : Drop table Student CASCADE;

However, a drop table with restrict can be used to prevent accidental deletions.

Drop table Student RESTRICT

will not drop student table if there is a view or integrity constraint refers to Student.

Executed Queries on Views:

Creating and inserting data into student_details table:

```
mysql> create table student_detail(std_id int,name varchar(20),address varchar(20));
Query OK, 0 rows affected (0.33 sec)

mysql> insert into student_detail(std_id,name,address) values
(1,'stephan','delhi'),(2,'kathrin','noida'),(3,'david','ghaziabad'),(4,'alina','gurugram');
Query OK, 4 rows affected (0.11 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

Creating and inserting data into student_marks table:

```
mysql> create table student_marks(std_id int,name varchar(20),marks int,age int);
Query OK, 0 rows affected (0.11 sec)

mysql> insert into student_marks(std_id,name,marks,age) values
(1,'stephan',97,19),(2,'kathrin',86,21),(3,'david',74,18),(4,'alina',90,19),(5,'john',96,18);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Creating view detailsview on student_detial:

```
mysql> create view detailsview as select name,address from student_detail where std_id<4;
Query OK, 0 rows affected (0.17 sec)

mysql> select * from detailsview;
+---------+-----------+
| name    | address   |
+---------+-----------+
| stephan | delhi     |
| kathrin | noida     |
| david   | ghaziabad |
+---------+-----------+
3 rows in set (0.08 sec)
```

Creating view marksview on student_detial and s=student_marks:

```
mysql> create view marksview as select student_detail.name, student_detail.address,
student_marks.marks from student_detail, student_marks where
student_detail.name=student_marks.name;
Query OK, 0 rows affected (0.09 sec)

mysql> select * from marksview;
+---------+-----------+-------+
| name    | address   | marks |
+---------+-----------+-------+
| stephan | delhi     |    97 |
| kathrin | noida     |    86 |
| david   | ghaziabad |    74 |
| alina   | gurugram  |    90 |
+---------+-----------+-------+
4 rows in set (0.00 sec)
```

Deleting View marksview:

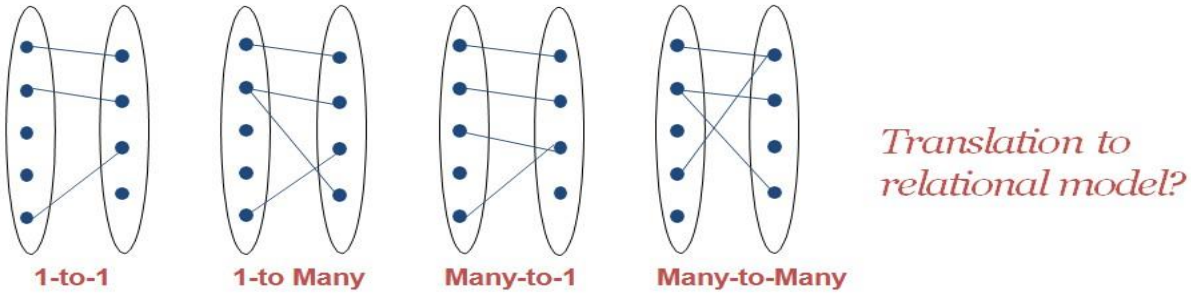
```
mysql> drop view marksview;
Query OK, 0 rows affected (0.00 sec)

mysql> create view marksview1 as select student_detail.name, student_marks.marks,student_marks.age from student_detail, student_marks where student_detail.std_id=student_marks.std_id;
Query OK, 0 rows affected (0.20 sec)

mysql> select * from marksview1;
+------+-------+------+
| name | marks | age  |
+------+-------+------+
| xyz  |    98 |   18 |
| pqr  |    95 |   19 |
| abc  |    68 |   17 |
| uvw  |    78 |   18 |
+------+-------+------+
4 rows in set (0.09 sec)
```

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 6
ER-RELATIONAL – ADDITIONAL EXAMPLES

There are 4 types of relationships:

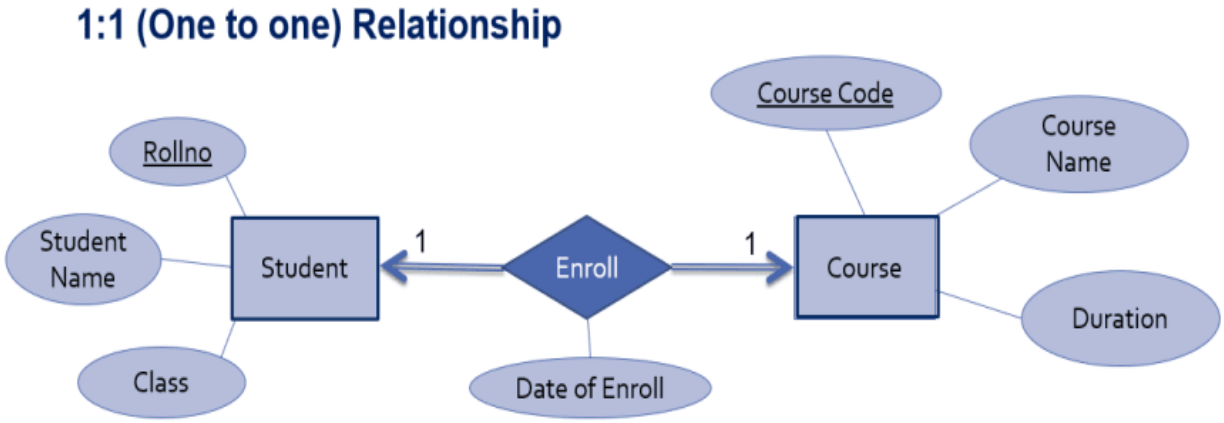
- **1:1 [One-to-One]**
- **1:M [One-to-Many]**
- **M:1 [Many-to-One].**
- **M: M [Many-to-Many]**

Let us see the mapping of these relationship sets into tables



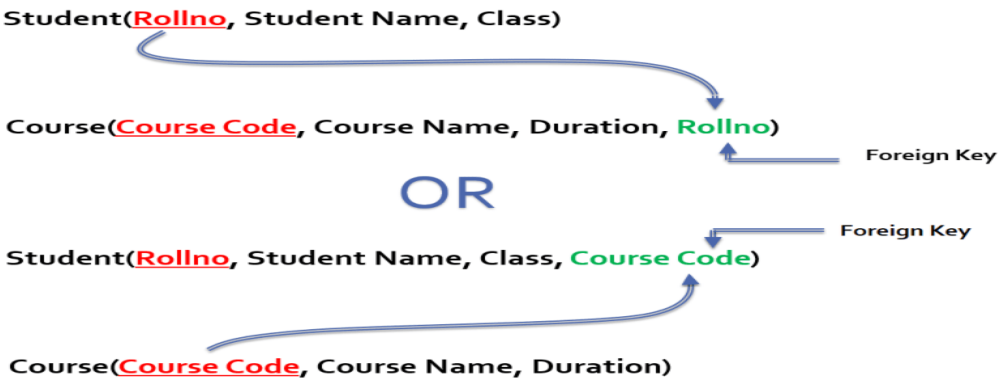
1:1 (One to One) Relationship:

Consider 1:1 relationship set enroll exist between entity sets student and course, which means one student can enroll in only one courses



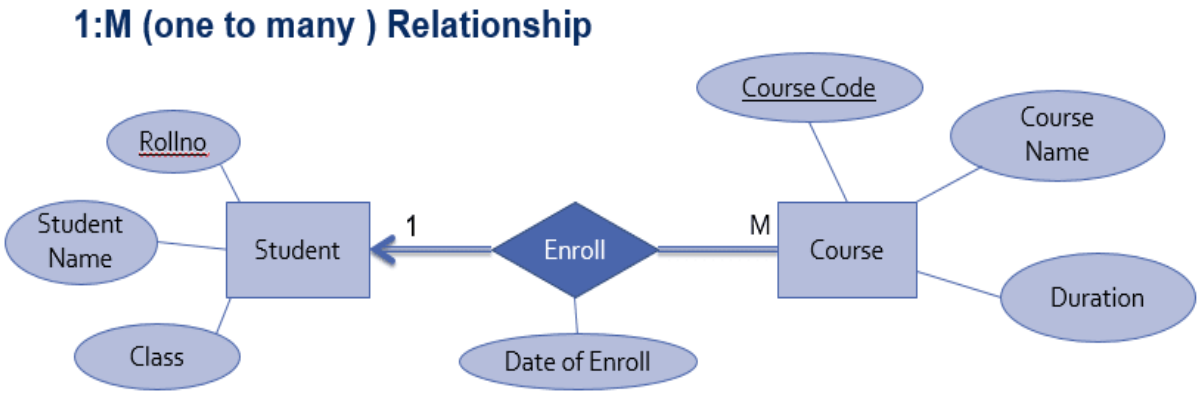
To convert this Relationship set into relational schema,

1. Separate relation is created for each participating entity sets.
2. Primary Key of Relation Student can be act as foreign key for relation Course
   OR
   Primary Key of Relation Course act as foreign key for relation Student.



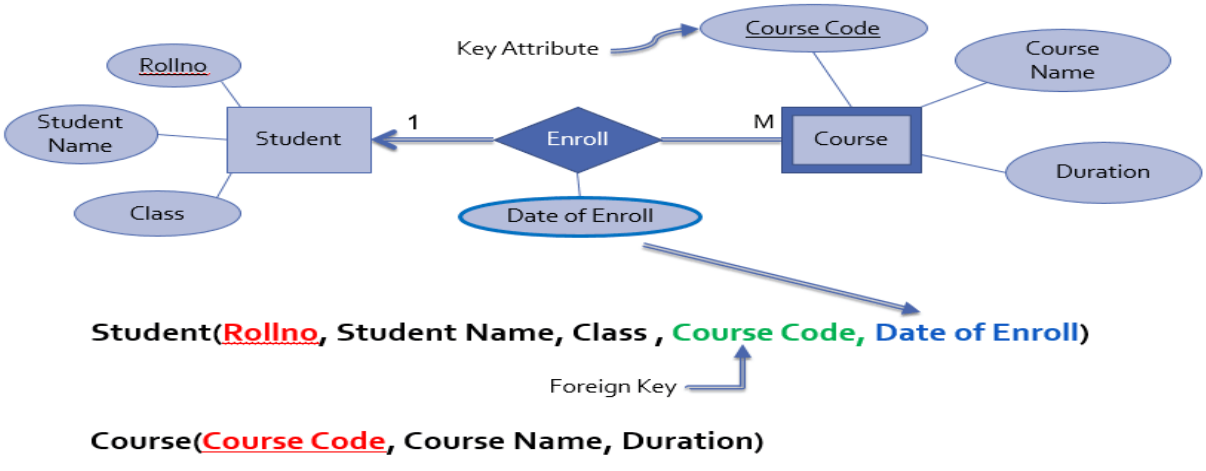
1:M (One to Many) Relationship:

Consider 1:M relationship set enrolled exist between entity sets Student and Course which means that one student can enroll in multiple courses.



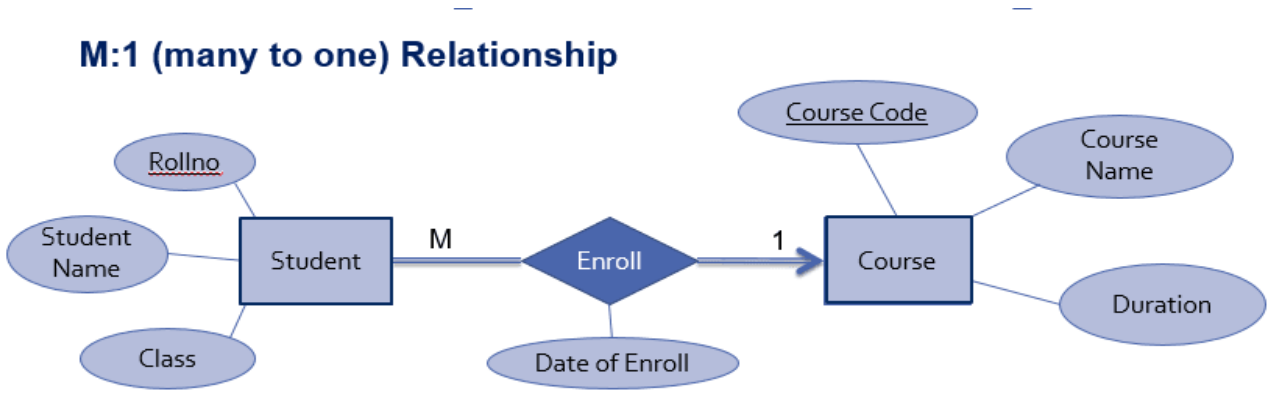
In this case to convert this relationship into relational schema,

1. Separate relation is created for each participating entity sets (student and course)
2. Key attribute of Many's side entity set (course) is mapped as foreign key in one's side relation(Student)
3. All attributes of relationship set are mapped as attributes for relation of one's side entity set (student)



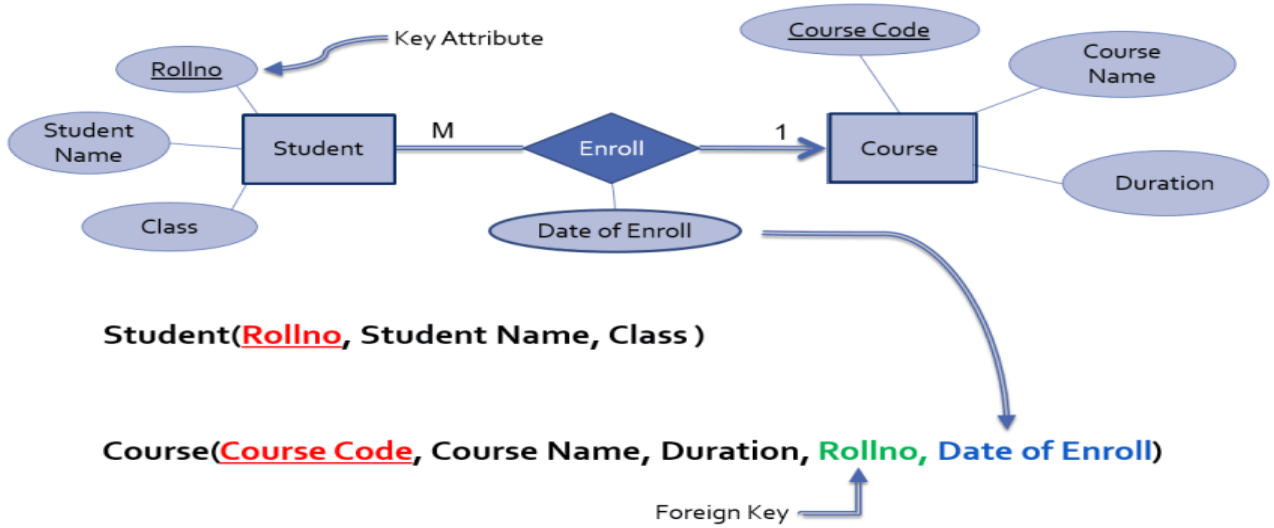
M:1 (Many to One) Relationship:

Consider relationship set enroll exist between entity sets student and course . but here student is many side entity set while course is one side entity set. Which means many students can enroll in one course.



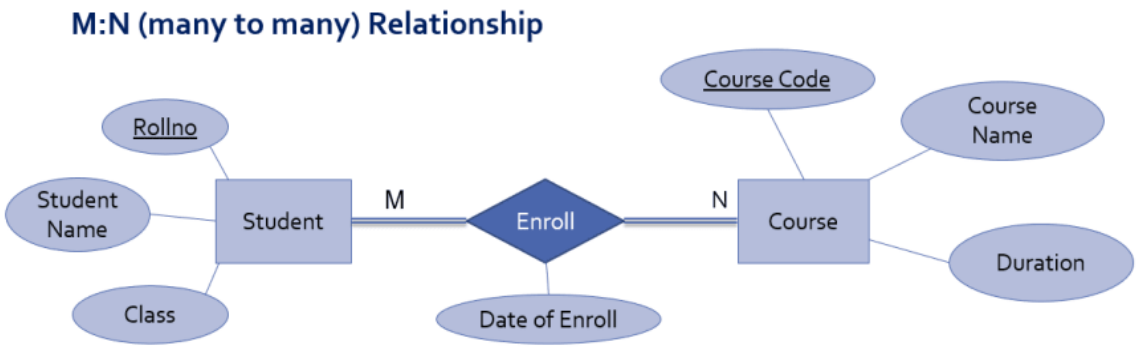
To convert this relationship set into relational schema,

1. Separate relation is created for each participating entity sets.
2. Key attribute of Many's side entity set student is mapped as foreign key in one's side relation
3. All attributes of relationship set are mapped as attributes for one's side relation course.



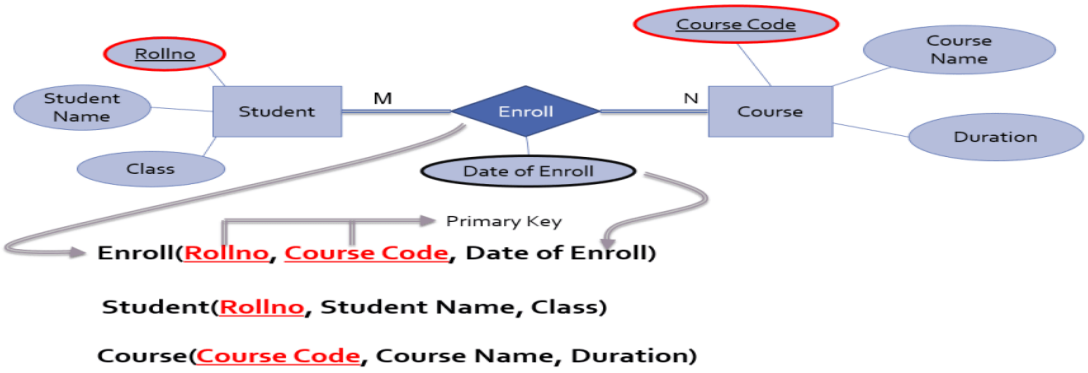
M:N (Many to Many) Relationship:

Consider same relationship set enrolled exist between entity sets student and course, which means multiple student can enroll in multiple courses.

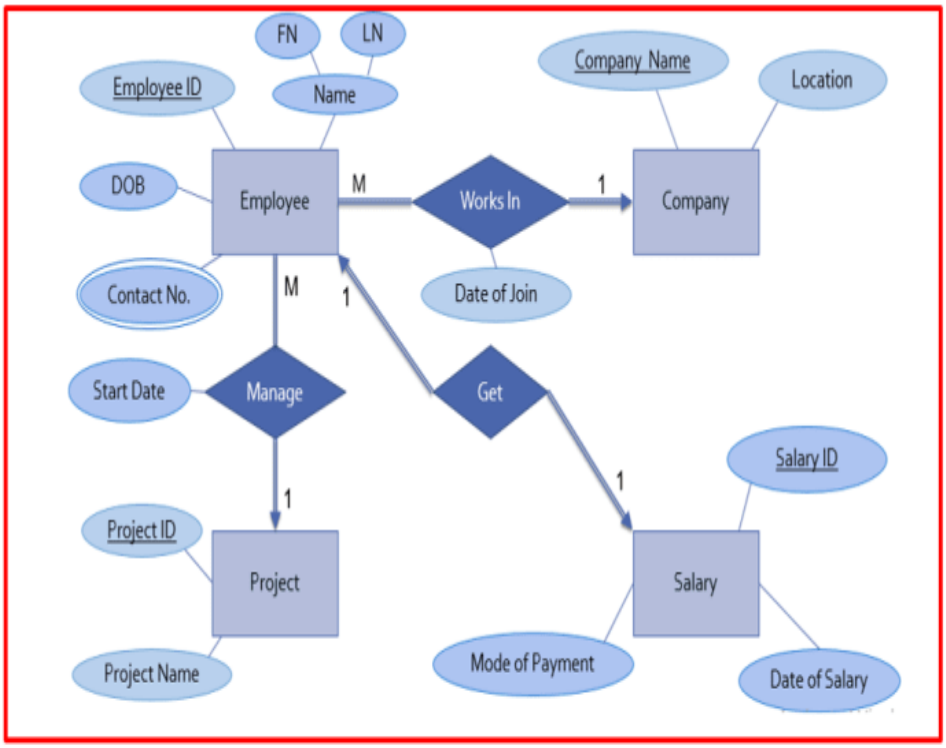


To convert this Relationship set into relational schema,

1. Relationship set is mapped as separate relation
2. Key attributes of participating entity sets are mapped as primary key for that relation
3. Attribute of relationship set becomes simple attributes for that relation
4. And separate relation is created for other participating entities



EXAMPLE 1:



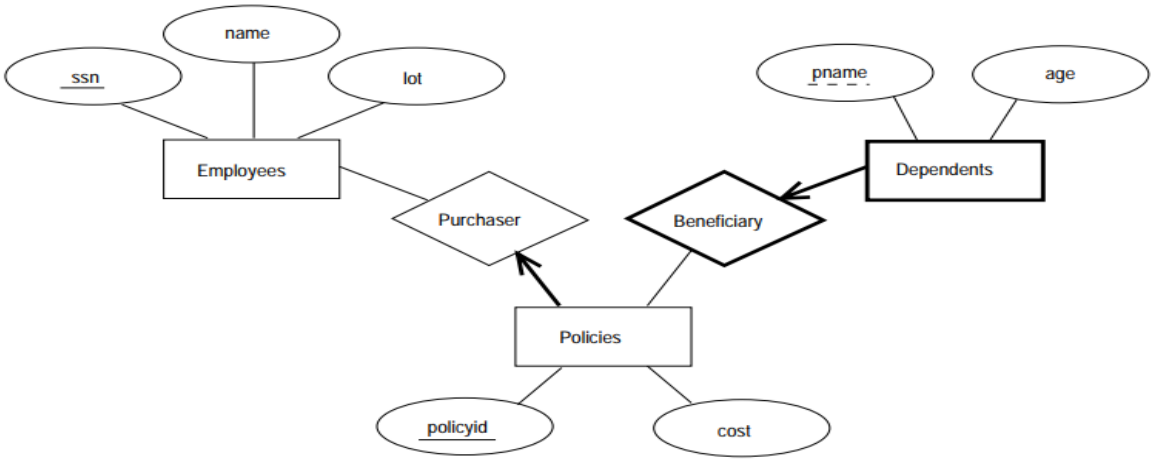
ER Diagram

Employee(Employee ID , DOB , FN , LN , Salary ID)
EmployeeContact(Employee ID, Contact No)
Company(Company Name , Location, Employee ID, Date of Join)
Project(Project ID , Project Name , Employee ID, Start Date)
Salary(Salary ID , Mode of Payment, Date of Salary, Employee ID)

Relational Model

EXAMPLE 2:

Consider the ER diagram shown in Figure below. We can translate this ER diagram into the relational model as follows, taking advantage of the key constraints to combine Purchaser information with Policies and Beneficiary information with Dependents:




```
CREATE TABLE Policies ( policyid INTEGER,
        cost REAL,
        ssn CHAR(11) NOT NULL,
        PRIMARY KEY (policyid),
        FOREIGN KEY (ssn) REFERENCES Employees ON DELETE CASCADE ) ;


CREATE TABLE Dependents ( pname CHAR(20),
        age INTEGER,
        policyid INTEGER,
        PRIMARY KEY (pname, policyid),
        FOREIGN KEY (policyid) REFERENCES Policies ON DELETE CASCADE ) ;
```

Note: the deletion of an employee leads to the deletion of all policies owned by the employee and all dependents who are beneficiaries of those policies

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 1
DATA MANIPULATION LANGUAGE

SQL stands for Structured Query Language. It is used for storing and managing data in Relational Database Management System (RDBMS).

It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables.

All the RDBMS like MySQL, Informix, Oracle, MSAccess and SQL Server use SQL as their standard database language.

SQL allows users to query the database in a number of ways, using English-like statements.

NOTE:
Structure query language is not case sensitive. Generally, keywords of SQL are written in uppercase.
Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.

What is SQL Process?

When an SQL command is executing for any RDBMS, then the system figure out the best way to carry out the request and the SQL engine determines that howto interpret the task.

In the process, various components are included. These components can be optimization Engine, Query-engine, Query dispatcher, classic query-engine, etc.

What is Advantages of SQL?

- High speed
- No coding needed
- Well defined standards
- Portability
- Interactive language
- Multiple data view

DATA MANIPULATION LANGUAGE

A DML (data manipulation language) refers to a computer programming language that allows you to add (insert), delete (delete), and alter (update) data in a database. A DML is typically a sublanguage of a larger database language like SQL, A DML (data manipulation language) is a group of computer languages that provide commands for manipulating data in databases.

DDL vs DML Commands

DDL	DML
Used to define database objects like tables, indexes, views, etc.	Used to manipulate data within the database.
Examples of DDL statements include CREATE, ALTER, and DROP.	Examples of DML statements include SELECT, INSERT, UPDATE, and DELETE.
Changes made using DDL affect the structure of the database.	Changes made using DML affect the data stored in the database.
DDL statements are not transactional, meaning they cannot be rolled back.	DML statements are transactional, meaning they can be rolled back if necessary.
DDL statements are usually executed by a database administrator.	DML statements are executed by application developers or end-users.
DDL statements are typically used during the design and setup phase of a database.	DML statements are used during normal operation of a database.
Examples of DDL statements: CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE INDEX, etc.	Examples of DML statements: SELECT, INSERT, UPDATE, DELETE, etc.

DML Commands

Command	Description
INSERT	Used to insert new data records or rows in the database table
UPDATE	Used to set the value of a field or column for a particular record to a new value
DELETE	Used to remove one or more rows from the database table
SELECT	Used to retrieve data from one or more tables

1. INSERT

INSERT commands in SQL are used to insert data records or rows in a database table.
In an INSERT statement, we specify both the column_names for which the entry has to be made along with the data value that has to be inserted.

Insert can be used in various forms.

1. **Values only** - When the data is inserted into all fields of the table, then use values only and maintain one-to-one correspondence (the no. of values and the order should match with the fields defined in the table).

   Syntaxa:

   INSERT INTO table_name VALUES (value1, value2, value3, ...)

   By VALUES, we mean the value of the corresponding columns.

Example:

```
INSERT INTO customers
        VALUES ('1006','2020-03-04',3200,'Sukesh Kumar', 'DL', '1008');
```

2. **Column names and Values** - When the data is inserted into selected fields of the table or in user defined order, then the column names would be specified before the values.

   Syntax:

   INSERT INTO table_name (column_name_1, column_name_2, column_name_3, ...)
   VALUES (value1, value2, value3, ...)

   By VALUES, we mean the value of the corresponding columns.

   Example:

```
INSERT INTO  customers( customer_id, sale_date, sale_amount, salesperson, order_id)
        VALUES (1005,'2019-12-12',4200,'R K Rakesh','1007');
```

3. **Multiple rows –** multiple rows (records) can be inserted using a single insert statement by including the input values as sets separated by comma.

   Syntax:

   INSERT INTO table_name (column_name_1, column_name_2, ...)
   VALUES (value1, value2, ...) , (value1, value2, ...), …

   Example:

```
INSERT INTO  customers( customer_id, sale_amount, salesperson, order_id)
        VALUES (1006,8200,'R  Mukesh','1008'), (1007, 1000, 'R  Nagesh', '1009');
```

4. **Data from other Tables -** The command copies data from one table and inserts it into another table. It requires that data types in source and target tables match.

   Syntax

   Copy all columns from one table to another table:
   INSERT INTO *table2*
   SELECT * FROM *table1*
   [WHERE *condition]*;
   Copy only some columns from one table into another table:
   INSERT INTO *table2* (*column1, column2, column3, ...*)
   SELECT *column1, column2, column3, ...* FROM *table1*
   [WHERE *condition]* ;

   Example:

```
INSERT INTO Customers (CustomerName, City, Country)
        SELECT SupplierName, City, Country FROM Suppliers;
```

2. UPDATE

UPDATE command or statement is used to modify the value of an existing column in a database table. The UPDATE statement can be used to update single or multiple columns on the basis of our specific needs.

Syntax:;

```
UPDATE table_name
SET column_name_1 = value1, column_name_2 = value2, ...
[ WHERE condition ];
```

Example:

1. UPDATE customers SET store_state = 'DL' WHERE store_state = 'NY';
   In this example, we have modified the value of store_state for a record where store_state was 'NY' and set it to a new value 'DL'.

2. UPDATE employees SET bonus = 5000;
   In this example, all the rows of the employees table will set the bonus value to 5000.

3. DELETE

DELETE statement in SQL is used to remove one or more rows from the database table. It does not delete the data records permanently. We can always perform a rollback operation to undo a DELETE command.
With DELETE statements we can use the WHERE clause for filtering specific rows.

Syntax :

```
DELETE FROM table_name [ WHERE condition ];
```

Example :

1. DELETE FROM customers WHERE store_state = 'MH' AND customer_id = '1001';
   This will delete the details of customer with id=1001 whose store_state is 'MH'

2. DELETE FROM customers;
   This will delete all the records from the customer table. This command is also equivalent to the TRUNCATE command in DDL.

4. SELECT

A Query (SELECT) in SQL is a statement to retrieve data from one or more tables.
The data returned is stored in a result table, called the result-set.

It can have six clauses, but only the first two (select and from) are mandatory. The clauses are specified in the following order.

Syntax:

Select ***<List of Columns and expressions (usually involving columns)>***

From ***<List of Tables & Join Operators>***

[Where ***<List of Row conditions joined together by And, Or, Not>]***

[SGroup By ***<list of grouping columns>]***

[Having ***<list of group conditions connected by And, Or, Not >]***

[Order By ***<list of sorting specifications>]***

Examples:

Syntax to select all rows and columns:

Syntax: SELECT * FROM tablename;

Example: select * from employee;

Syntax to select selected columns and all rows:

Syntax: SELECT column1, column2... from tablename;
Here, column1, column2, ... are the field names of the table you want to select data from.

Example: select eid, ename afrom employee;

Syntax to select selected rows and selected columns:

Syntax: **SELECT column1, column2.. from tablename where condition;**

Example: select eid, ename from employee where eid=101;

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 2
BASIC SQL QUERIES

Retrieving Information from a Table

The SELECT statement is used to pull information from a table.

The general form of the statement is:

```
SELECT what_to_select
FROM which_table
WHERE conditions_to_satisfy;
```

what_to_select indicates what you want to see. This can be a list of columns, or * to indicate "all columns."
which_table indicates the table from which you want to retrieve data.
The WHERE clause is optional. If it is present, **conditions_to_satisfy** specifies one or more conditions that rows must satisfy to qualify for retrieval.

Example:

```
SELECT * FROM Customers WHERE Country = 'Mexico';

SELECT * FROM Customers WHERE CustomerID = 1;
```

Operators in The WHERE Clause

Arithmatic Operators:

()	Parentheses
/	Division
*	Multiplication
-	Subtraction
+	Addition
%	Modulo

Comparison Operators

=	equal
<>, !=	not equal
>	Greater Than
<	Less Than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators

AND	BETWEEN	IS NULL
OR	IN	IS NOT NULL
NOT	LIKE	

Few Examples:

```
SELECT * FROM emp;
SELECT Ename FROM emp where sal>50000;
SELECT * FROM emp where sal + bonus < 50000;
SELECT * FROM Customers WHERE Country='Mexico‘;
SELECT * FROM Customers WHERE Country='Germany' AND City='Berlin';
SELECT * FROM Customers WHERE City='Berlin' OR City='München';
SELECT * FROM Customers WHERE NOT Country='Germany';
```

Alias (as) names:

Used to assign names to the columns when they are retrieved from the database table.

Syntax:

Select expr1 [as alias1], expr2 [as alias2] [, …]
From table1 [, table2, …]
[Where condition]

Example:

Select city, ((1.8 + avg_temp) + 32) AS temperature From Temperature

A multiword heading needs to be enclosed in double quotes

Example:

Select city, ((1.8 + avg_temp) + 32) AS “Average Temperature” From Temperature.

Working with NULL values:

Conceptually, NULL means “a missing unknown value” and it is treated somewhat differently from other values.
Null values are no values in the field - so the regular operators can not be used for comparison.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation

IS NULL is used to check if the field contains a null value or not.
IS NOT NULL is used to see if a field is not null

Example

```
Select eid, ename  From emp Where bonus IS NULL;

Select eid, ename From emp Where bonus IS NOT NULL;
```

Distinct

The SELECT DISTINCT statement is used to return only distinct (different) values.
Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.
Eliminates all the duplicate entries in the table resulting from the query.

Syntax:

Select [DISTINCT] select_list
From table[, table, …]
[Where expression]
[Order By expression]

Example: SELECT DISTINCT Country FROM Customers;

In Operator:

IN condition checks if the values in a column are present in a given list (set of values).

Syntax:

Select select_list
From table
Where column [not] in (value_list);

Example (Using IN):

Select sid, sname From student Where branch IN ('CSE', 'IT', 'CSD');

Example (not Using IN)

Select sid, sname From student Where branch='CSE' OR , branch='IT' OR branch= 'CSD';

NOT IN can similarly be used to select rows where values do not match

Select sid, sname From student Where branch NOT IN ('CSE', 'IT', 'CSD');

Between Operator:

Between condition is used to see if the value of a column lies between specified ranges

Syntax:

Select col1, col2,…
From table
Where column [not] between lower_value and upper_value;

Example:

Select eid, emp From Emp Where sal between 50000 and 99999

Alternate Query:

Select eid, emp From Emp Where sal>=50000 and sal<=99999;

Pattern Matching usin LIKE operator:

Like allows a matching of patterns in the column data

Syntax:

Select select_list
From table
Where column [not] like 'pattern' [Escape char]

Wildcards:

- Any Single Character
% (or *) 0 or more characters

A combination of '-' and '%' along with other characters can be used to represent different patterns.

For test of fixed number of characters multiple dashes can be used

For example '---' will select all 3 letter words from the column

Example: select * from emp where ename like 'a%';
select * from emp where ename not like 'a%';

Few Examples:

LIKE Operator	Description
WHERE EName LIKE 'a%'	**Finds any values that start with "a"**
WHERE EName LIKE '%a'	**Finds any values that end with "a"**
WHERE EName LIKE '%or%'	**Finds any values that have "or" in any position**
WHERE EName LIKE '_r%'	**Finds any values that have "r" in the second position**
WHERE EName LIKE 'a_%'	**Finds any values that start with "a" and are at least 2 characters in length**
WHERE EName LIKE 'a__%'	**Finds any values that start with "a" and are at least 3 characters in length**
WHERE EName LIKE 'a%o'	**Finds any values that start with "a" and ends with "o"**

LIMIT Clause:

The LIMIT clause is used to specify the number of records to return.

The LIMIT clause is useful on large tables with thousands of records. Returning a large number of records can impact performance. Hence use limit to control the no. of rows displayed.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Example: select * from emp limit 3;

Limit can also be used to skip certain number of columns and then continue to retrieve by limiting to no. of rows specified.

Example: select * from emp limit 3, 2;

Order By Clause:

The ORDER BY keyword is used to sort the result-set in ascending or descending order.
The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.
Multiple levels of sort can be done by specifying multiple columns

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Example:

Display employee names in alphabetical order

SELECT ename FROM emp ORDER BY ename;

Display the details of employee earning highest to lowest salary

SELECT * FROM emp ORDER BY sal DESC;

Display details of employees department wise in alphabetical order

SELECT * FROM emp ORDER BY dno, ename;

Display employee list department wise in descending order of salary

SELECT * FROM emp ORDER BY dno, sal desc;

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 3
SET OPERATORS

Relational databases employ set operators to modify and compare data sets. Simply put, a set operator creates a new set of data that satisfies particular requirements by combining or comparing two or more data sets.

The frequently used set operators in MySQL are: UNION, UNION ALL, INTERSECT, and EXCEPT. With these operators, the output of two or more SELECT statements can be combined or compared.

UNION: To merge the output of two or more SELECT operations into a single result set, use UNION. Each SELECT statement's unique rows are included in the result set.

UNION ALL operator in MySQL is used to combine the result sets of two or more SELECT statements into a single result set. Unlike the UNION operator, which removes duplicate rows, UNION ALL preserves all rows from all SELECT statements.

INTERSECT, on the other hand, returns only the rows that are shared by two SELECT statements' result sets. This operator is useful when comparing two tables to find only the common records between them.

EXCEPT returns those rows from the first SELECT statement that are not present in the second. This operator is analogous to the difference between two sets in mathematics.

Set operators are powerful tools that allow you to manipulate data sets in various ways. As a result, you can increase the efficiency and performance of your database operations by incorporating them properly into your MySQL queries.

Necessary Condition for the Usage of Set Operators

Set operators in MySQL are a key feature that allows you to combine or compare data sets. However, certain prerequisites must be completed before utilizing set operators in your queries.

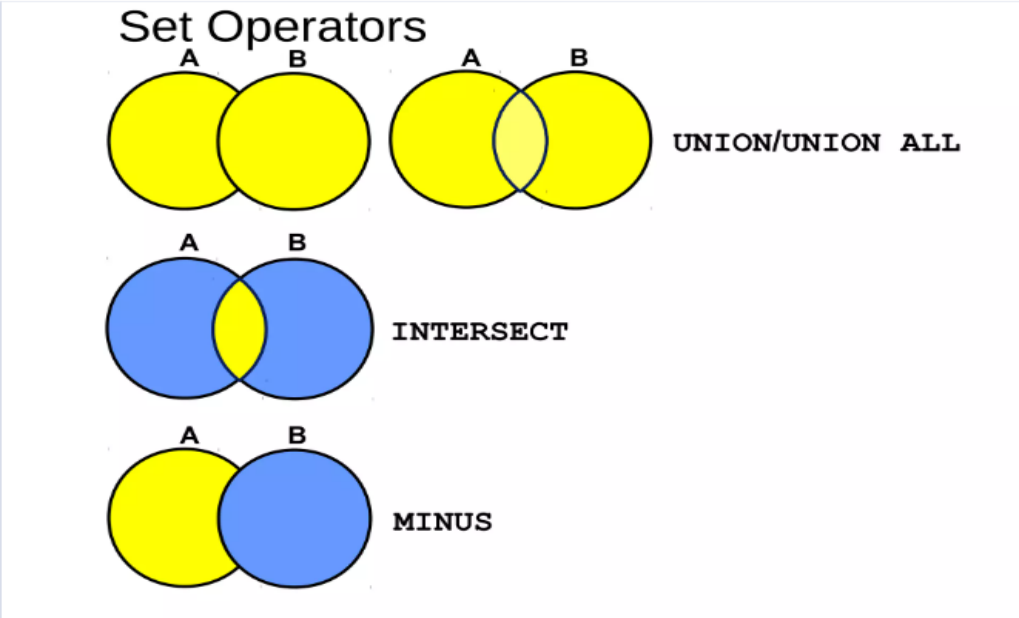
The first requirement is that the SELECT queries should have the same number of columns and be of the same data type. This ensures that the data sets are interoperable and easily integrated or compared.

According to the second criterion, the columns in the SELECT statements must be in the same order. This is significant because the set operators combine or compare rows based on the order of the columns.

Furthermore, the data types of the columns in the SELECT statements must be compatible. A set operator, for example, cannot be used to combine a column of integers with a column of strings.

GENERAL SYNTAX :

```
SELECT column1, column2
FROM table1
UNION/UNION ALL/INTERSECT/EXCEPT
SELECT column1, column2
FROM table2;
```



Let's say we have two tables called employees and customers for illustration as follows:

EMPLOYEES:

id	name	salary
1	John	50000
2	Jane	60000
3	Bob	55000

CUSTOMERS:

id	name	city
1	Alice	Boston
2	Bob	Miami
3	Charlie	Austin

UNION OPERATOR

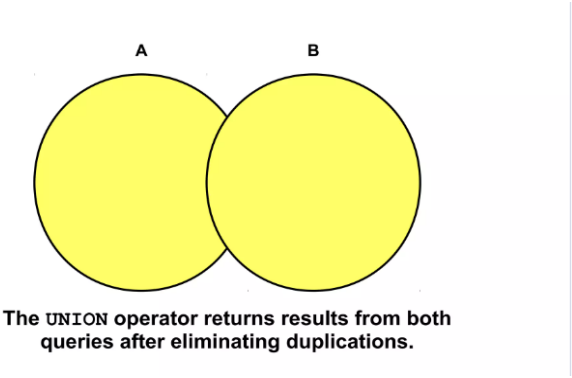
The Union is a binary set operator in DBMS. It is used to combine the result set of two select queries. Thus, It combines two result sets into one. In other words, the result set obtained after union operation is the collection of the result set of both the tables.

But two necessary conditions need to be fulfilled when we use the union command.

These are:

1. Both SELECT statements should have an equal number of fields in the same order.
2. The data types of these fields should either be the same or compatible with each other.

The Union operation can be demonstrated as follows:



The syntax for the UNION operation is as follows:

```
SELECT (coloumn_names) from table1 [WHERE condition]
UNION
SELECT (coloumn_names) from table2 [WHERE condition];
```

Example:

```
SELECT name FROM employees
UNION
SELECT name FROM customers;
```

Output:

name
Alice
Bob
Charlie
Jane
John

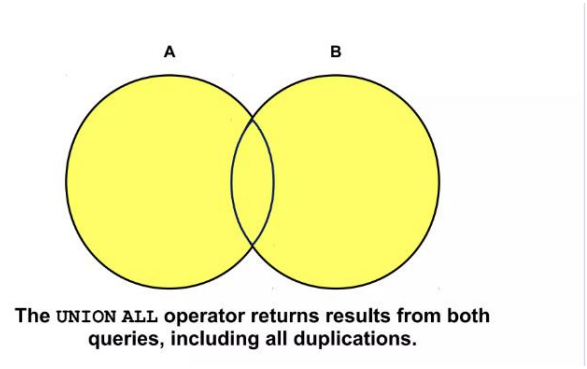
UNION ALL OPERATOR:

The Union operation gives us distinct values. If we want to allow the duplicates in our result set, we'll have to use the **'Union-All'** operation.

Union All operation is also similar to the union operation. The only difference is that it allows duplicate values in the result set.

The syntax for the UNION ALL operation is as follows:

```
SELECT (coloumn_names) from table1 [WHERE condition]
UNION ALL
SELECT (coloumn_names) from table2 [WHERE condition];
```



Example:

SELECT name FROM employees
UNION ALL
SELECT name FROM customers;

Output:

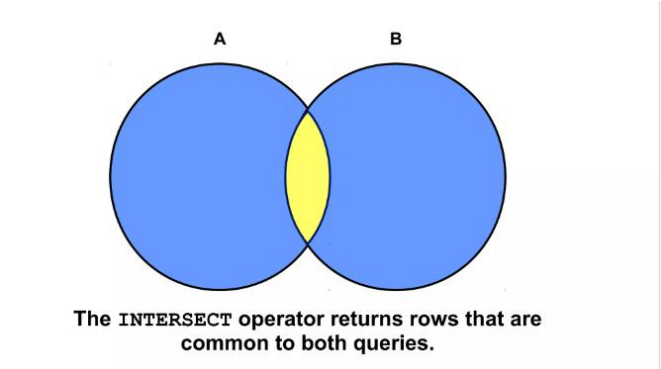
name
John
Jane
Bob
Alice
Bob
Charlie

INTERSECT OPERATOR

Intersect is a binary set operator in DBMS. The intersection operation between two selections returns only the common data sets or rows between them. It should be noted that the intersection operation always returns the distinct rows. The duplicate rows will not be returned by the intersect operator.

Here also, the above conditions of the union and minus are followed, i.e., the number of fields in both the SELECT statements should be the same, with the same data type, and in the same order for the intersection.

The intersection operation can be demonstrated as follows:



The syntax for the INTERSECT operation is as follows:

```
SELECT (coloumn_names) from table1 [WHERE condition]
INTERSECT
SELECT (coloumn_names) from table2 [WHERE condition];
```

Example:

```
SELECT name FROM employees
INTERSECT
SELECT name FROM customers;
```

Output:

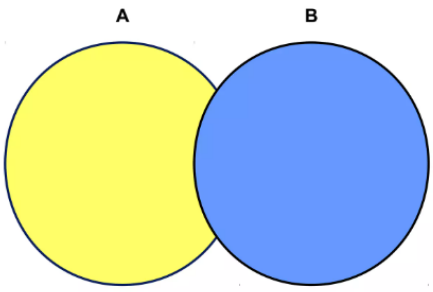
name
Bob

EXCEPT OPERATOR

EXCEPT is a binary set operator in DBMS. The EXCEPT operation between two selections returns the rows that are present in the first selection but not in the second selection. The Minus EXCEPT operator returns only the distinct rows from the first table.

It is a must to follow the above conditions that we've seen in the union, i.e., the number of fields in both the SELECT statements should be the same, with the same data type, and in the same order for the minus operation.

The EXCEPT operation can be demonstrated as follows:



The MINUS operator returns rows in the first query that are not present in the second query.

The syntax for the EXCEPT operation is as follows:

```
SELECT (coloumn_names) from table1 [WHERE condition]
INTERSECT
SELECT (coloumn_names) from table2 [WHERE condition];
```

Example:

```
SELECT name FROM employees
EXCEPT
SELECT name FROM customers;
```

Output:

name
John
Jane

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 4
AGGREGATE OPERATORS

In addition to simply retrieving data, we often want to perform some computation or summarization.
SQL allows a powerful class of constructs for computing aggregate values such as COUNT, MIN etc.
Aggregate functions are used in MySQL to calculate a set of values and return a single value as a result.
These functions are useful when we want to analyze data stored in a table and generate summary information, such as the total number of rows, the sum of values in a column, or the average of a set of values.
Using aggregate functions, we can filter data, calculate statistics, and improve performance (by limiting the data that needs to be processed).

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. **COUNT ([DISTINCT] A):** The number of (unique) values in the A column.
2. **SUM ([DISTINCT] A):** The sum of all (unique) values in the A column.
3. **AVG ([DISTINCT] A):** The average of all (unique) values in the A column.
4. **MAX (A**): The maximum value in the A column.
5. **MIN (A):** The minimum value in the A column

GROUP BY():

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions to group the result-set by one or more columns.

HAVING CONDITION:

The HAVING clause was added to SQL to check the conditions with aggregate functions.

Syntax:

SELECT ***column_name(s)***
FROM ***table_name*** **WHERE** ***condition***
GROUP BY ***column_name(s)*** **HAVING** ***condition***
ORDER BY ***column_name(s);***

Use this sample EMP table for illustrating the group functions

EMP TABLE

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	1980-12-17	800	NULL	20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30
7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30
7566	JONES	MANAGER	7839	1981-04-02	2975	NULL	20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850	NULL	30
7782	CLARK	MANAGER	7839	1981-06-09	2450	NULL	10
7788	SCOTT	ANALYST	7566	1982-12-09	30000	NULL	20
7839	KING	PRESIDENT	NULL	1981-11-17	25000	NULL	10
7844	TURNER	SALESMAN	7698	1981-09-08	45500	0	30
7876	ADAMS	CLERK	7788	1983-01-12	21100	NULL	20
7900	JAMES	CLERK	7698	1981-12-03	25950	NULL	30
7902	FORD	ANALYST	7566	1981-12-03	63000	NULL	20
7934	MILLER	CLERK	7782	1982-01-23	75300	NULL	10

COUNT():

COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.
When we apply COUNT() on a column then NULL values are ignored.

SYNTAX:

SELECT COUNT(*) FROM table_name WHERE condition;

OR

SELECT COUNT([ALL|DISTINCT] column_name) FROM table_name WHERE condition;

Examples:

select count(*) from emp;	14
select count(comm) from emp;	4
select count(job) from emp;	14
select count(distinct job) from emp;	5
select count(ename) from emp where deptno=30;	6

select count(ename) "No. of Employees", deptno from emp group by deptno;

Empl	DeptNo
3	10
5	20
6	30

select count(ename) "No. of Employees", deptno from emp group by deptno having count(ename)>3;

Empl	DeptNo
5	20
6	30

SUM()

The SUM() function returns the total sum of a numeric column.

SYNTAX:

SELECT SUM(COL_NAME) FROM table_name

[WHERE CONDITION]

[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select sum(sal) from emp;	299025.00
select sum(sal) from emp where job='analyst';	93000.00
select sum(sal) from emp where deptno=10;	102750.00

select job, sum(sal) from emp group by job;

```
mysql> select job, sum(sal) from emp group by job;
+-----------+-----------+
| job       | sum(sal)  |
+-----------+-----------+
| CLERK     | 123150.00 |
| SALESMAN  |  49600.00 |
| MANAGER   |   8275.00 |
| ANALYST   |  93000.00 |
| PRESIDENT |  25000.00 |
+-----------+-----------+
5 rows in set (0.00 sec)
```

select deptno, sum(sal) from emp group by deptno having sum(sal)>100000;

```
mysql> select deptno, sum(sal) from emp group by deptno having sum(sal)>100000;
+--------+-----------+
| deptno | sum(sal)  |
+--------+-----------+
|     10 | 102750.00 |
|     20 | 117875.00 |
+--------+-----------+
2 rows in set (0.00 sec)
```

AVG()

The AVG() function returns the average value of a numeric column.

SYNTAX:

SELECT AVG(COL_NAME) FROM table_name

[WHERE CONDITION]

[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select avg(sal) from emp; 21358.92

select avg(sal) from emp where job='analyst'; 123150.00

select deptno, avg(sal) from emp group by deptno having avg(sal)>15000;

MAX() and MIN()

The MIN() function returns the smallest value of the selected column.
The MAX() function returns the largest value of the selected column.

SYNTAX:

SELECT MIN(COL_NAME) FROM table_name

[WHERE CONDITION]

[GROUP BY COL_NAME] [HAVING CONDITION];

SELECT MAX(COL_NAME) FROM table_name

[WHERE CONDITION]

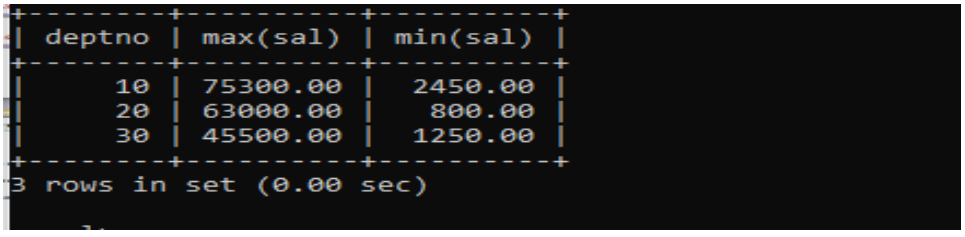
[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select min(sal) from emp; 800.00

select max(sal) from emp where job='analyst'; 63000.00

select deptno, max(sal), min(sal) from emp group by deptno;



```
+--------+----------+----------+
| deptno | max(sal) | min(sal) |
+--------+----------+----------+
|     10 | 75300.00 |  2450.00 |
|     20 | 63000.00 |   800.00 |
|     30 | 45500.00 |  1250.00 |
+--------+----------+----------+
3 rows in set (0.00 sec)
```

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 5
NESTED QUERIES AND CORRELATED QUERIES

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Syntax

SELECT column_name FROM table_name

WHERE column_name expression operator

(SELECT column_name from table_name WHERE condition);

The subquery (inner query) execute before the main query (outer query).
The result of the subquery is used by the main query.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

Let us use the employee table for the illustrations of the query

Types of Subqueries:

1 .Single row Subquery – returns zero or one row

2. Multiple row Subquery – returns one or more rows

3. Multi Column Subquery – returns one or more columns

1. Single Row Subquery:

SQL subqueries are most frequently used with the Select statement. Single row subqueries returns one row. Single row subquery uses comparison operators like (=, >, <, >=, <=, <>)

Example

1. Find the name, job and sal of employees whose salary is less than that of an employee with empno=7876.

```
SELECT ENAME, JOB , SAL FROM EMP
WHERE SAL < (SELECT SAL FROM EMP WHERE EMPNO=7876);
```

OUTPUT:

```
mysql> SELECT ENAME, JOB, SAL FROM EMP
    -> WHERE SAL< (SELECT SAL FROM EMP WHERE EMPNO=7876);
+-------+-------+---------+
| ENAME | JOB   | SAL     |
+-------+-------+---------+
| SMITH | CLERK | 800.00  |
| JAMES | CLERK | 950.00  |
+-------+-------+---------+
```

2. Display the employee name who is having the same job as '7369' and earning more than '7876'

```
SQL> SELECT    ename, job
  2  FROM      emp
  3  WHERE     job =
  4                    (SELECT      job
  5                    FROM         emp
  6                    WHERE        empno = 7369)
  7  AND       sal >
  8                    (SELECT      sal
  9                    FROM         emp
  10                   WHERE        empno = 7876);
```

3. Display the details of employees who is earning the least salary

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    sal =
  4                  (SELECT      MIN(sal)
  5                  FROM         emp);
```

4. List out the departments whose Minimum salary is more than the least salary of department 20

```
SQL> SELECT       deptno, MIN(sal)
  2  FROM         emp
  3  GROUP BY     deptno
  4  HAVING       MIN(sal) >
  5                        (SELECT    MIN(sal)
  6                        FROM       emp
  7                        WHERE      deptno = 20);
```

MULTI - ROW SUBQUERIES:

Multi row subqueries return set of rows. This type of query uses set comparison operators like (IN, ANY, ALL). We cannot use comparison operators with multi row subqueries.

IN - Equals to any member in the list
ANY - Return rows that match any value on a list
ALL - Return rows that match all the values in a list.

IN OPERATOR:

Find the employees whose salary is same as the minimum salary of employees in the department

SELECT ENAME, SAL FROM EMP
WHERE SAL IN (SELET MIN(SAL) FROM EMP GROUP BY DEPTNO);

```
mysql> SELECT ENAME, SAL FROM EMP
    -> WHERE SAL IN (SELECT MIN(SAL) FROM EMP GROUP BY DEPTNO);
+-------+---------+
| ENAME | SAL     |
+-------+---------+
| SMITH |  800.00 |
| JAMES |  950.00 |
| MILER | 1300.00 |
+-------+---------+
```

ANY OPERATOR:

Display the employees whose salary is more than the maximum salary of the employees in any department.

SELECT ENAME, SAL FROM EMP
WHERE SAL > ANY (SELECT MAX(SAL) FRM EMP GROUP BY DEPTNO);

```
mysql> SELECT ENAME, SAL FROM EMP
    -> WHERE SAL>ANY (SELECT MAX(SAL) FROM EMP
    -> GROUP BY DEPTNO);
+-------+---------+
| ENAME | SAL     |
+-------+---------+
| JONES | 2975.00 |
| SCOTT | 3000.00 |
| KING  | 5000.00 |
| FORD  | 3000.00 |
+-------+---------+
```

ALL OPERATOR:

Display the details of employees who earn salary less than those whose job is ‘MANAGER’

SELECT EMPNO, ENAME, JOB, SAL FROM EMP

WHERE SAL < ALL (SELECT SAL FROM EMP WHERE JOB=”MANAGER”)

AND JOB <> “MANAGER”;

```
mysql> SELECT EMPNO, ENAME, JOB, SAL FROM EMP
    -> WHERE SAL<ALL (SELECT SAL FROM EMP WHERE JOB='MANAGER')
    -> AND JOB <> 'MANAGER';
+-------+--------+----------+---------+
| EMPNO | ENAME  | JOB      | SAL     |
+-------+--------+----------+---------+
|  7369 | SMITH  | CLERK    |  800.00 |
|  7499 | ALLEN  | SALESMAN | 1600.00 |
|  7521 | WARD   | SALESMAN | 1250.00 |
|  7654 | MARTIN | SALESMAN | 1250.00 |
|  7844 | TURNER | SALESMAN | 1500.00 |
|  7876 | ADAMS  | CLERK    | 1100.00 |
|  7900 | JAMES  | CLERK    |  950.00 |
|  7934 | MILER  | CLERK    | 1300.00 |
+-------+--------+----------+---------+
```

MULTIPLE COLUMN SUBQUERY:

A subquery that compares more than one column between the parent query and subquery is called the multiple column subqueries.

Example: List the details of the employees who earn the same salary as of employee no. 7521 with the same job also.

SELECT ENAME, JOB, SAL, EMPNO FROM EMP

WHERE (JOB, SAL) IN (SELECT JOB, SAL FROM EMP WHERE EMPNO=7521);

Here there needs to be a pairwise comparison of the outer query rows with the inner query output.

```
mysql> SELECT ENAME, JOB, SAL, EMPNO FROM EMP
    -> WHERE (JOB,SAL) IN (SELECT JOB, SAL FROM EMP WHERE EMPNO=7521);
+--------+----------+---------+-------+
| ENAME  | JOB      | SAL     | EMPNO |
+--------+----------+---------+-------+
| WARD   | SALESMAN | 1250.00 |  7521 |
| MARTIN | SALESMAN | 1250.00 |  7654 |
+--------+----------+---------+-------+
```

Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

INSERT INTO table_name (column1, column2, column3....)
SELECT * FROM table_name WHERE VALUE OPERATOR
(SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);

Example

Consider a table EMPLOYEE_DUP similar to EMPLOYEE.

INSERT INTO EMPLOYEE_DUP
SELECT * FROM EMP WHERE SAL IN
(SELECT SAL FROM EMP WHERE DEPTNO=20);

Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

UPDATE table SET column_name = new_value WHERE VALUE OPERATOR (SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);

Example

UPDATE EMP_DUP SET COMM= SAL * 0.25 WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE COMM IS NULL AND DEPTNO=20);

Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

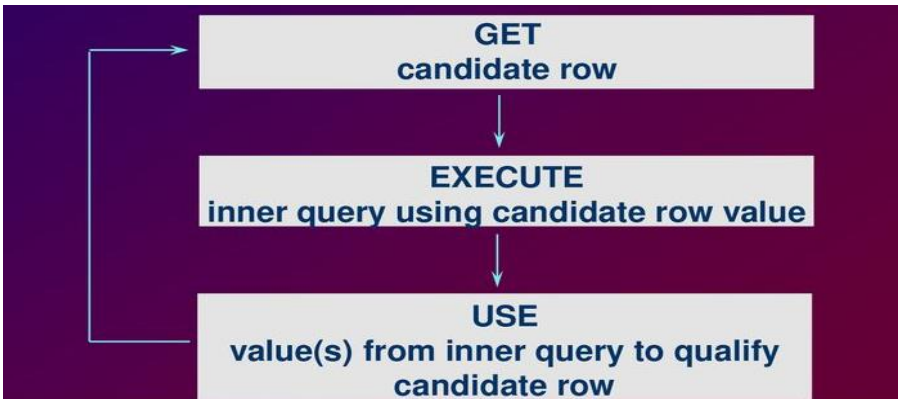
DELETE FROM TABLE_NAME WHERE VALUE OPERATOR (SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);

Example:

DELETE FROM EMP_DUP WHERE SAL IN (SELECT SAL FROM EMP WHREE SAL>50000);

CORRELATED SUBQUERIES

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.

Syntax:

```
SELECT column1, column2, ....
FROM table1 outer
WHERE column1 operator
                    (SELECT column1, column2
                     FROM table2
                     WHERE expr1 = outer.expr2);
```

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

Example:

1. Find all the employees who make more than the average salary in their department.

   SELECT EMPNO, SAL, DEPTNO FROM EMP OUTER WHERE SAL > (SELECT AVG(SAL) FROM EMP INNER WHREE OUTER.DEPTNO = INNER.DEPTNO);

2. Display the employees whose salary is highest among all the employees.

   SELECT * FROM EMP A WHERE 1 = (SELECT COUNT(*) FROM EMP B WHERE B.SAL >= A.SAL);

CORRELATED UPDATE :

```
UPDATE table1 alias1
 SET column = (SELECT expression  FROM table2 alias2  WHERE
          alias1.column =  alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

CORRELATED DELETE :

```
DELETE FROM table1 alias1
 WHERE column1 operator
          (SELECT expression  FROM table2 alias2
           WHERE alias1.column = alias2.column);
```

Use a correlated subquery to delete rows in one table based on the rows from another table.

Using the EXISTS Operator :

The EXISTS operator tests for existence of rows in the results set of the subquery. If a subquery row value is found the condition is flagged **TRUE** and the search does not continue in the inner query, and if it is not found then the condition is flagged **FALSE** and the search continues in the inner query.

EXAMPLE of using EXIST operator :

Find employees who have at least one person reporting to them.

```
SELECT employee_id, last_name, job_id, department_id
FROM employees outer
WHERE EXISTS ( SELECT 'X'
FROM employees  WHERE manager_id = outer.employee_id);
```

EXAMPLE of using NOT EXIST operator :

Find all departments that do not have any employees.

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X' FROM employees WHERE department_id
= d.department_id);
```

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 6
CONCEPT OF JOINS

Databases usually store large amounts of data. To analyze that data efficiently, analysts and DBAs have a constant need to extract records from two or more tables based on certain conditions.

A relational database consists of multiple related tables linking together using common columns, which are known as foreign key columns. Because of this, the data in each table is incomplete from the business perspective.

JOINS are used to retrieve data from multiple tables in a single query by establishing logical relationship between them. This operation is made feasible with a common key value between tables.

JOIN clauses are used in the SELECT, UPDATE, and DELETE statements.

Different types of JOINs in MySQL
MySQL JOIN type defines the way two tables are related in a query.

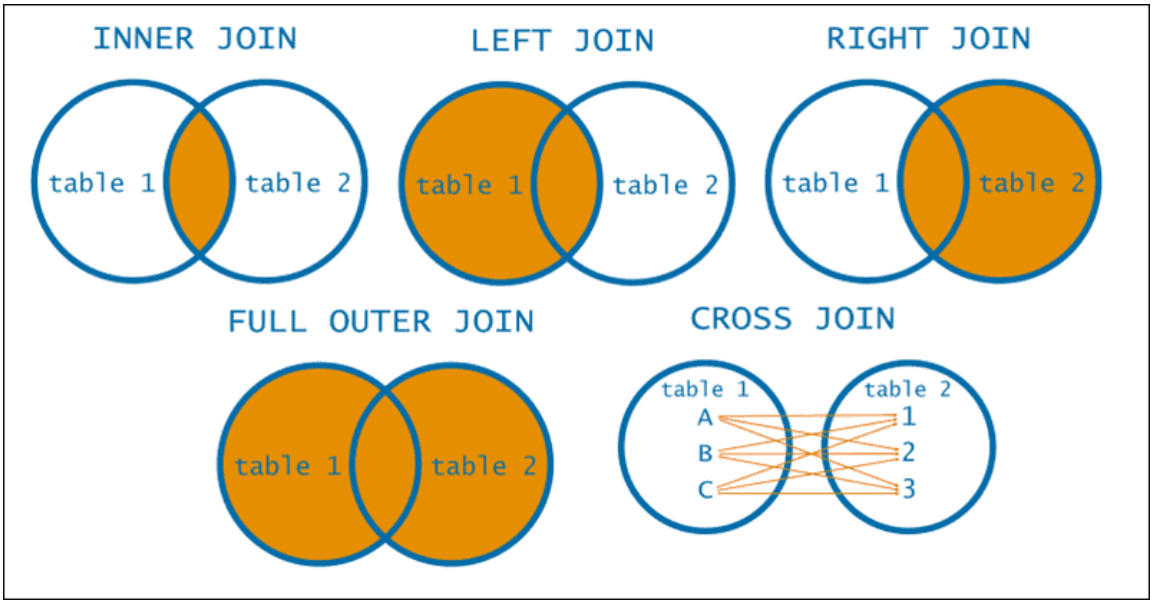
INNER JOIN: Returns records that have matching values in both tables

LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table

RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table

CROSS JOIN: Returns Cartesian product of left and right table

SELF JOIN: join that is used to join a table with itself

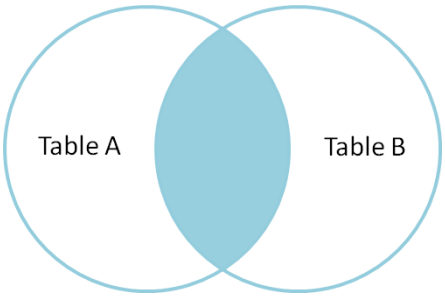


Let us use these SUPPLIER and PARTS tables for illustration

```
mysql> SELECT * FROM SUPPLIER;
+-----+-------+--------+-----------+
| SNO | SNAME | STATUS | CITY      |
+-----+-------+--------+-----------+
| S1  | Smith |     20 | London    |
| S2  | Jones |     10 | Paris     |
| S3  | Blake |     30 | Paris     |
| S4  | Clark |     20 | London    |
| S5  | Adams |     30 | Athens    |
| S6  | Pavan |     24 | Hyderabad |
+-----+-------+--------+-----------+
6 rows in set (0.00 sec)

mysql> SELECT * FROM PARTS;
+-----+------+-------+-------+-------+--------+------+
| PNO | SNO  | PNAME | COLOR | WEIGH | CITY   | cost |
+-----+------+-------+-------+-------+--------+------+
| P1  | S1   | Nut   | Red   |    12 | London |   50 |
| P2  | S1   | Bolt  | Green |    17 | Paris  |   70 |
| P3  | S2   | Screw | Blue  |    17 | Rome   |   80 |
| P4  | S3   | Screw | Red   |    14 | London |   80 |
| P5  | S2   | Cam   | Blue  |    12 | Paris  |   90 |
| P6  | S3   | Cog   | Red   |    19 | London |   68 |
+-----+------+-------+-------+-------+--------+------+
6 rows in set (0.00 sec)
```

INNER JOIN:

The **INNER JOIN** will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.



SYNTAX:

SELECT table1.column1,table1.column2,table2.column1,....
FROM table1 INNER JOIN table2
ON table1.matching_column = table2.matching_column;

Here,

table1: First table.
table2: Second table
matching_column: Column common to both the tables.

Example:

Fetch data of the supplier with the parts name sold by the Supplier.

select A.SNAME, B.PNAME
from supplier A **INNER JOIN** parts B
ON A.SNO=B.SNO;

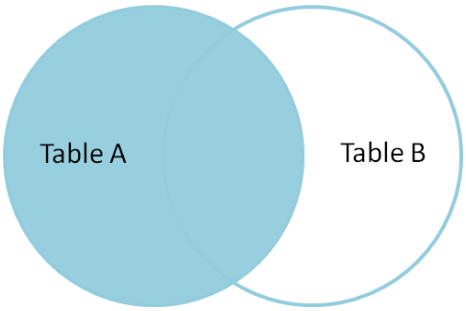
OUTPUT:

```
mysql> select A.SNAME, B.PNAME from supplier A INNER JOIN parts B ON A.SNO=B.SNO;
+-------+-------+
| SNAME | PNAME |
+-------+-------+
| Smith | Nut   |
| Smith | Bolt  |
| Jones | Screw |
| Blake | Screw |
| Jones | Cam   |
| Blake | Cog   |
+-------+-------+
6 rows in set (0.00 sec)
```

SQL LEFT (OUTER) JOIN

In contrast to INNER JOINs, OUTER JOINs return not only matching rows but non-matching ones as well. In case there are non-matching rows in a joined table, the NULL values will be shown for them.

LEFT JOIN clause allows retrieving all rows from the left table(Table A), along with those rows from the right table(Table B) for which the join condition is satisfied. Wherever any record of the left table does not match with the right table NULL is displayed for right-side columns.



SYNTAX:

SELECT table1.column1,table1.column2,table2.column1,....
FROM table1 LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

Example:

Fetch data of all the suppliers whether they sold any part or not..

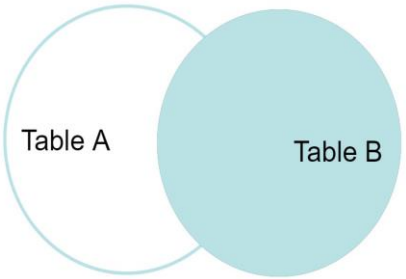
select A.SNAME, A.CITY, B.PNAME, B.COLOR
from supplier A **LEFT JOIN** parts B
ON A.SNO=B.SNO;

OUTPUT:

```
mysql>  select a.SNAME, a.CITY, b.PNAME, b.COLOR from supplier a LEFT JOIN parts b ON a.SNO=b.SNO;
+-------+-----------+-------+-------+
| SNAME | CITY      | PNAME | COLOR |
+-------+-----------+-------+-------+
| Smith | London    | Bolt  | Green |
| Smith | London    | Nut   | Red   |
| Jones | Paris     | Cam   | Blue  |
| Jones | Paris     | Screw | Blue  |
| Blake | Paris     | Cog   | Red   |
| Blake | Paris     | Screw | Red   |
| Clark | London    | NULL  | NULL  |
| Adams | Athens    | NULL  | NULL  |
| Pavan | Hyderabad | NULL  | NULL  |
+-------+-----------+-------+-------+
9 rows in set (0.00 sec)
```

SQL RIGHT (OUTER) JOIN

The RIGHT OUTER JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.



SYNTAX:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1  RIGHT JOIN  table2
ON table1.matching_column = table2.matching_column;
```

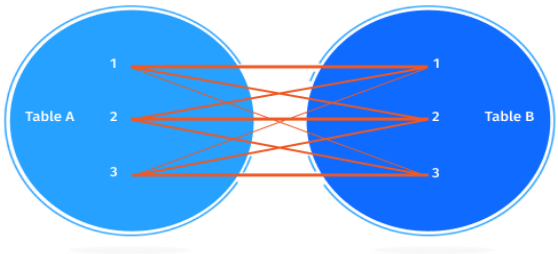
Example:

Fetch data of parts irrespective of whether they were sold by any supplier or not.

```
select A.SNAME, A.CITY,  B.PNAME, B.COLOR
from supplier A RIGHT JOIN parts B
ON A.SNO=B.SNO;
```

SQL CROSS JOIN

MySQL CROSS JOIN, also known as a Cartesian join, retrieves all combinations of rows from each table. In this type of JOIN, the result set is returned by multiplying each row of table 1 with all rows in table 2 if no additional condition is introduced.



SYNTAX:

```
SELECT columns
FROM table1   CROSS JOIN    table2
```

NOTE: if there are N records in Table 1 and M records in Table 2, then the result set will contain N x M records.

```
mysql> select * from supplier cross join parts;
+-----+-------+--------+-----------+-----+-----+-------+-------+-------+--------+------+
| SNO | SNAME | STATUS | CITY      | PNO | SNO | PNAME | COLOR | WEIGH | CITY   | cost |
+-----+-------+--------+-----------+-----+-----+-------+-------+-------+--------+------+
| S6  | Pavan |     24 | Hyderabad | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S5  | Adams |     30 | Athens    | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S4  | Clark |     20 | London    | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S3  | Blake |     30 | Paris     | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S2  | Jones |     10 | Paris     | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S1  | Smith |     20 | London    | P1  | S1  | Nut   | Red   |    12 | London |   50 |
| S6  | Pavan |     24 | Hyderabad | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S5  | Adams |     30 | Athens    | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S4  | Clark |     20 | London    | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S3  | Blake |     30 | Paris     | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S2  | Jones |     10 | Paris     | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S1  | Smith |     20 | London    | P2  | S1  | Bolt  | Green |    17 | Paris  |   70 |
| S6  | Pavan |     24 | Hyderabad | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S5  | Adams |     30 | Athens    | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S4  | Clark |     20 | London    | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S3  | Blake |     30 | Paris     | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S2  | Jones |     10 | Paris     | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S1  | Smith |     20 | London    | P3  | S2  | Screw | Blue  |    17 | Rome   |   80 |
| S6  | Pavan |     24 | Hyderabad | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S5  | Adams |     30 | Athens    | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S4  | Clark |     20 | London    | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S3  | Blake |     30 | Paris     | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S2  | Jones |     10 | Paris     | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S1  | Smith |     20 | London    | P4  | S3  | Screw | Red   |    14 | London |   80 |
| S6  | Pavan |     24 | Hyderabad | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S5  | Adams |     30 | Athens    | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S4  | Clark |     20 | London    | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S3  | Blake |     30 | Paris     | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S2  | Jones |     10 | Paris     | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S1  | Smith |     20 | London    | P5  | S2  | Cam   | Blue  |    12 | Paris  |   90 |
| S6  | Pavan |     24 | Hyderabad | P6  | S3  | Cog   | Red   |    19 | London |   68 |
| S5  | Adams |     30 | Athens    | P6  | S3  | Cog   | Red   |    19 | London |   68 |
| S4  | Clark |     20 | London    | P6  | S3  | Cog   | Red   |    19 | London |   68 |
| S3  | Blake |     30 | Paris     | P6  | S3  | Cog   | Red   |    19 | London |   68 |
| S2  | Jones |     10 | Paris     | P6  | S3  | Cog   | Red   |    19 | London |   68 |
| S1  | Smith |     20 | London    | P6  | S3  | Cog   | Red   |    19 | London |   68 |
+-----+-------+--------+-----------+-----+-----+-------+-------+-------+--------+------+
36 rows in set (0.00 sec)
```

SELF JOIN

SELF JOIN implies the joining of a table to itself. It states that each row of a table is joined with itself and with every other row of the same table. When you want to extract the hierarchical data or compare rows within the same table, then self-join is the best choice.

SYNTAX:

SELECT columns

FROM table AS alias1 JOIN table AS alias2

ON alias1.column = alias2.column;

Example:

Determine the names of employees, who earn more than their managers.

SELECT M.ENAME "EMPLOYEE", M.SAL, E.ENAME "MANAGER", E.SAL
FROM EMP M, EMP E
WHERE E.EMPNO=M.MGR AND M.SAL>E.SAL;

```
mysql> SELECT M.ENAME "EMPLOYEE", M.SAL, E.ENAME "MANAGER", E.SAL FROM EMP M, EMP E WHERE E.EMPNO=M.MGR AND M.SAL>E.SAL;
+----------+----------+---------+---------+
| EMPLOYEE | SAL      | MANAGER | SAL     |
+----------+----------+---------+---------+
| SCOTT    | 30000.00 | JONES   | 2975.00 |
| TURNER   | 45500.00 | BLAKE   | 2850.00 |
| JAMES    | 25950.00 | BLAKE   | 2850.00 |
| FORD     | 63000.00 | JONES   | 2975.00 |
| MILLER   | 75300.00 | CLARK   | 2450.00 |
+----------+----------+---------+---------+
5 rows in set (0.00 sec)
```

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 7
SCHEMA REFINEMENT - INTRODUCTION

Schema refinement is the process of improving a database schema to ensure it meets the desired criteria of correctness, efficiency, and maintainability. This involves restructuring the schema to eliminate redundancy, ensure data integrity, and optimize performance.

It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.

The Schema Refinement refers to refine the schema by using some techniques.

Importance of Schema Refinement:

1. **Eliminate Redundancy:**
   - Reduces duplicated data, which saves storage and avoids anomalies.
2. **Ensure Data Integrity:**
   - Maintains accuracy and consistency of data through well-defined relationships and constraints.
3. **Optimize Performance:**
   - Enhances query performance by organizing data in an efficient manner.
4. **Simplify Maintenance:**
   - Eases database management tasks like updates, insertions, and deletions by ensuring a clean, logical structure.

Process of Schema Refinement:

1. **Analysis**:
   Assess the current schema for inefficiencies and anomalies.
2. **Normalization**:
   Apply normalization principles to structure the schema into appropriate normal forms.
3. **Decomposition**:
   Break down larger tables into smaller, related tables to eliminate redundancy.
4. **Validation**:
   Ensure that decompositions are lossless and preserve dependencies.
5. **Optimization:**
   Fine-tune the schema for performance by indexing, partitioning, and other techniques.

<u>Redundancy:</u>

Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized. Suppose a table of student details attributes are: student Id, student name, college name, college rank, course opted.

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1

The Problem of redundancy in Database

As it can be observed that values of attribute college name, college rank, course is being repeated which can lead to problems. Problems caused due to redundancy are: Insertion anomaly, Deletion anomaly, and Updation anomaly.

1. **Insertion Anomaly –**
   If a student detail has to be inserted whose course is not being decided yet then insertion will not be possible till the time course is decided for student.

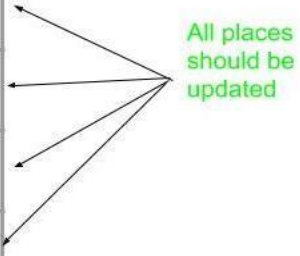
Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU		1

   This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record.

2. **Deletion Anomaly –**
   If the details of students in this table is deleted then the details of college will also get deleted which should not occur by common sense.
   This anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.

3. **Updation Anomaly –**
   Suppose if the rank of the college changes then changes will have to be all over the database which will be time-consuming and computationally costly.

Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1



If updation do not occur at all places then database will be in inconsistent state.

Decomposition:

Decomposition is the process of breaking down in parts or elements.

It replaces a relation with a collection of smaller relations.

It breaks the table into multiple tables in a database.

It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.

If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

Types of Decomposition:

1. Lossless Decomposition
2. Lossy Decomposition

Lossless Decomposition:

Ensures that the original table can be reconstructed from the decomposed tables without any loss of information.

"The decomposition of relation R into R1 and R2 is **lossless** when the join of R1 and R2 yield the same relation as in R."

A relational table is decomposed (or factored) into two or more smaller tables, in such a way that the designer can capture the precise content of the original tablPre by joining the decomposead parts.

This is called **lossless-join (or non-additive join) decomposition.**

Let us consider the following example:

<EmpInfo>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**	**Dept_ID**	**Dept_Name**
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Emp_ID	Dept_Name
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables −

The result will be −

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

Lossy Decomposition:

Some information might be lost when decomposing and rejoining the tables.

"The decomposition of relation R into R1 and R2 is lossy when the join of R1 and R2 does not yield the same relation as in R."

One of the disadvantages of decomposition into two or more relational schemes (or tables) is that some information is lost during retrieval of original relation or table.

Let us see an example −

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables −

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	**Dept_Name**
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation.

Therefore, the above relation has lossy decomposition.

Problems related to Decomposition:

- Lossy Decomposition:
  - Can lead to loss of information and data integrity issues.
- Increased Complexity:
  - Managing multiple tables and their relationships can become complex.
- Performance Overhead:
  - More joins required in queries can impact performance.
- Dependency Preservation:
  - Ensuring all functional dependencies are preserved in decomposed tables can be challenging.

Properties of Decomposition:

1. Lossless Decomposition
2. Dependency Preservation
3. Lack of Data Redundancy

1. Lossless Decomposition

Decomposition must be lossless.

It means that the information should not get lost from the relation that is decomposed.

It gives a guarantee that the join will result in the same relation as it was decomposed.

Example:

Let's take 'E' is the Relational Schema, With instance 'e';

is decomposed into: E1, E2, E3, ... En;

With instance: e1, e2, e3, ... en, If e1 ⋈ e2 ⋈ e3 . . . . ⋈ en = e, then it is called as **'Lossless Join Decomposition'.**

2. Dependency Preservation

- Dependency is an important constraint on the database.
- Every dependency must be satisfied by at least one decomposed table.
- If {A → B} holds, then two sets are functional dependent. And, it becomes more useful for checking the dependency easily if both sets in a same relation.
- This decomposition property can only be done by maintaining the functional dependency.
- In this property, it allows to check the updates without computing the natural join of the database structure.

3. Lack of Data Redundancy

- Lack of Data Redundancy is also known as a **Repetition of Information.**
- The proper decomposition should not suffer from any data redundancy.
- The careless decomposition may cause a problem with the data.
- The lack of data redundancy property may be achieved by Normalization process.

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 8
FUNCTIONAL DEPENDENCIES

Functional Dependency (FD) is a constraint that determines the relation of one attribute to another attribute in a Database Management System (DBMS). Functional Dependency helps to maintain the quality of data in the database. It plays a vital role to find the difference between good and bad database design.

A functional dependency is denoted by an arrow "→". The functional dependency of X on Y is represented by X → Y. Let's understand Functional Dependency in DBMS with example

Example:

Employee number	Employee Name	Salary	City
1	Dana	50000	San Francisco
2	Francis	38000	London
3	Andrew	25000	Tokyo

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

Key terms

Here, are some key terms for Functional Dependency in Database:

Key Terms	Description
Axiom	Axioms is a set of inference rules used to infer all the functional dependencies on a relational database.
Decomposition	It is a rule that suggests if you have a table that appears to contain two entities which are determined by the same primary key then you should consider breaking them up into two different tables.
Dependent	It is displayed on the right side of the functional dependency diagram.

Determinant	It is displayed on the left side of the functional dependency Diagram.
Union	It suggests that if two tables are separate, and the PK is the same, you should consider putting them. together

Rules of Functional Dependencies

Below are the Three most important rules for Functional Dependency in Database:

- Reflexive rule –. If X is a set of attributes and Y is_subset_of X, then X holds a value of Y.
- Augmentation rule: When x -> y holds, and c is attribute set, then ac -> bc also holds. That is adding attributes which do not change the basic dependencies.
- Transitivity rule: This rule is very much similar to the transitive rule in algebra if x -> y holds and y -> z holds, then x -> z also holds. X -> y is called as functionally that determines y.

Types of Functional Dependencies in DBMS

There are mainly four types of Functional Dependency in DBMS. Following are the types of Functional Dependencies in DBMS:

- **Multivalued Dependency**
- **Trivial Functional Dependency**
- **Non-Trivial Functional Dependency**
- **Full Functional Dependency**
- **Partial Functional Dependency**
- **Transitive Dependency**

Multivalued Dependency in DBMS:

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table. A multivalued dependency is a complete constraint between two sets of attributes in a relation. It requires that certain tuples be present in a relation. Consider the following Multivalued Dependency Example to understand.

Example:

Car model	**Maf vear**	**Color**
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this example, maf_year and color are independent of each other but dependent on car_model. In this example, these two columns are said to be multivalue dependent on car_model.

This dependence can be represented like this:

car_model -> maf_year

car_model-> colour

Trivial Functional Dependency in DBMS:

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute.

So, X -> Y is a trivial functional dependency if Y is a subset of X. Let's understand with a Trivial Functional Dependency Example.

For example:

Emp_id	Emp_name
AS555	Harry
AS811	George
AS999	Kevin

Consider this table of with two columns Emp_id and Emp_name.

{Emp_id, Emp_name} -> Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

Non Trivial Functional Dependency in DBMS:

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

Example:

(Company} -> {CEO} (if we know the Company, we knows the CEO name)

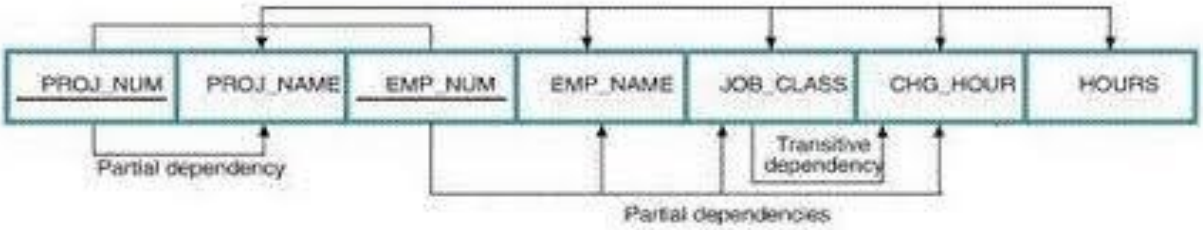
But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

Full Functional Dependency:

A functional dependency X→Y is said to be a full functional dependency, if removal of any attribute A from X, the dependency does not hold any more. i.e. Y is fully functional dependent on X, if it is Functionally Dependent on X and not on any of the proper subset of X.

For example, {Emp_num,Proj_num} → Hour

Is a full functional dependency. Here, Hour is the working time by an employee in a project.



Partial Functional Dependency:

A functional dependency X →Y is said to be a partial functional dependency, if after removal of any attribute A from X, the dependency still holds. i.e. Y is dependent on a proper subset of X.

So X is partially dependent on X.

For example, as per the above relation,

If {Emp_num, Proj_num} → Emp_name but also Emp_num → Emp_name

then Emp_name is partially functionally dependent on {Empl_num,Proj_num}.

a non-key attribute is determined by a part, but not the whole, of a COMPOSITE primary key.

Transitive Dependency in DBMS:

A Transitive Dependency is a type of functional dependency which happens when t is indirectly formed by two functional dependencies. Let's understand with the following Transitive Dependency Example.

Example:

Company	**CEO**	**Age**
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} -> {CEO} (if we know the compay, we know its CEO's name)

{CEO } -> {Age} If we know the CEO, we know the Age

Therefore according to the rule of rule of transitive dependency:

{ Company} -> {Age} should hold, that makes sense because if we know the company name, we can know his age.

Note: You need to remember that transitive dependency can only occur in a relation of three or more attributes.

Advantages of Functional Dependency

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database
- It helps you to maintain the quality of data in the database
- It helps you to defined meanings and constraints of databases
- It helps you to identify bad designs
- It helps you to find the facts regarding the database design

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 9
BASIC NORMAL FORMS

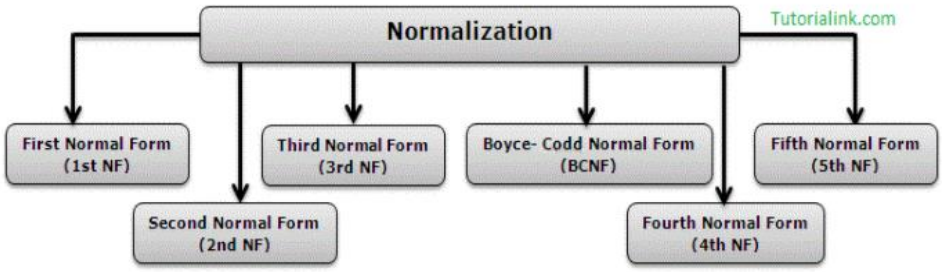
Normalization

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update & deletion anomaly. It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.

Objectives of Normalization:

1. Eliminate Redundant Data:
   Reduces the duplication of data to save space and ensure consistency.
2. Ensure Data Dependencies Make Sense:
   Organizes data so that dependencies are logical, ensuring data integrity.
3. Simplify Data Management:
   Makes databases easier to maintain by structuring data efficiently.



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF, if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. (Removes partial dependency)
3NF	A relation will be in 3NF, if it is in 2NF and no transition dependency exists.
BCNF	A relation will be BCNF, if it is in 3NF and **every functional dependency X → Y, X should be the super key of the table.(i.e** every FD, LHS is super key.)
4NF	A relation will be in 4NF, if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF, If it is in 4NF and does not contain any join dependency, joining should be lossless.

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	Subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}
Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:
teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

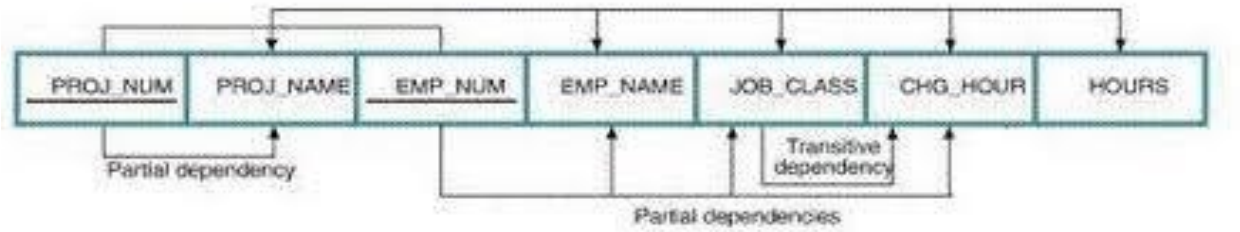
emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Normalization – Example

Consider a Relation as given below. Apply the different normal form as applicable



First Normal Form (2NF)

A table is in **1NF** if:

- All columns contain only atomic (indivisible) values.
- Each column contains values of a single type.
- Each column contains only a single value per row.

The given table already satisfies 1NF.

Second Normal Form (2NF)

A table is in 2NF if:

- It is in 1NF.
- All non-key attributes are fully functionally dependent on the primary key.

The primary key for this table can be a composite key consisting of **PROJ_NUM** and **EMP_NUM**.

However, there are partial dependencies:

PROJ_NUM → PROJ_NAME (Partial dependency)
EMP_NUM → EMP_NAME, **JOB_CLASS** (Partial dependencies)

To move to 2NF, we need to remove partial dependencies:
Create separate tables to eliminate partial dependencies:

Project Table:
PROJ_NUM (Primary Key)
PROJ_NAME

Employee Table:
EMP_NUM (Primary Key)
EMP_NAME
JOB_CLASS
CHG_HOUR

Assignment Table (Main Table):
PROJ_NUM (Composite Primary Key: PROJ_NUM, EMP_NUM)
EMP_NUM
HOURS

Third Normal Form (3NF)
A table is in 3NF if:

- It is in 2NF.
- It has no transitive dependencies (i.e., non-key attributes are not dependent on other non-key attributes).

In our current 2NF schema, there is a transitive dependencies
EMP_NUM → JOB_CLASS
JOB_CLASS → CHG_HOUR (Transitive dependency)

To remove the transitive dependency, split the Employee table from 2NF into

Employee Table:
EMP_NUM (Primary Key)
EMP_NAME
JOB_CLASS

Job Class Table:
JOB_CLASS (Primary Key)
CHG_HOUR

Final 3NF Tables:
Project Table
Employee Table
Job Class Table
Assignment Table

All the tables are now in 3NF because:
Each non-key attribute is fully functionally dependent on the primary key.
There are no transitive dependencies.a

DATABASE MANAGEMENT SYSTEMS
UNIT III – TOPIC 10
ADVANCED NORMAL FORMS

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	Stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
Stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	Stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:
For first table: emp_id
For second table: emp_dept
For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

Fourth normal form (4NF)

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency

EXAMPLE

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.
In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.
So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 1
TRANSACTION CONCEPT AND STATE

THE CONCEPT OF A TRANSACTION:

A transaction is a set of operations used to perform a logical unit of work. It is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data- manipulation language or programming language.

Transactions access data using two operations:

- ***read(X),*** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

- ***write(X)***, which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

ACID Properties:

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. These ensure tomaintain data in the face of concurrent access and system failures:

Atomicity: Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(A)operation but before the write(B)operation. In this case, the values of accounts A and B reflected in the database are

$950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum A + B is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an inconsistent state. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account A is $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the transaction-management component.

Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

Ex: the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to

B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as they allow multiple transactions to execute concurrently. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

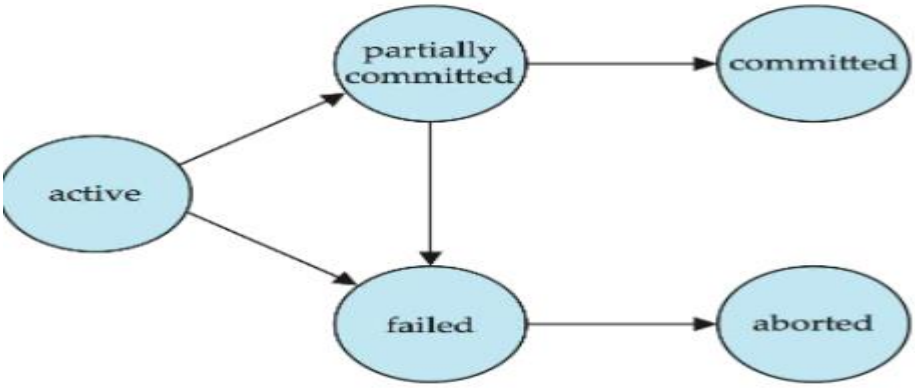
1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

Transaction State

Transaction State Diagram: A simple abstract transaction model is shown in fig below:



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,* after the discovery that 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.

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 residing 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 re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state. As mentioned earlier, we assume for now that failures do not result in loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

→It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

→It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 2

Implementation of Atomicity and Durability

Atomicity and durability are ACID properties that ensure the reliability and consistency of transactions in DBMS

Atomicity:

It is one of the key characteristics of transactions in database management systems (DBMS), which guarantees that every operation within a transaction is handled as a single, indivisible unit of work.

Either all the changes made by a transaction are committed to the database or none of them are committed.

Durability:

Durability, guarantees that changes made by a transaction once it has been committed are permanently kept in the database and will not be lost even in the event of a system failure or crash.

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the *shadow copy scheme.*

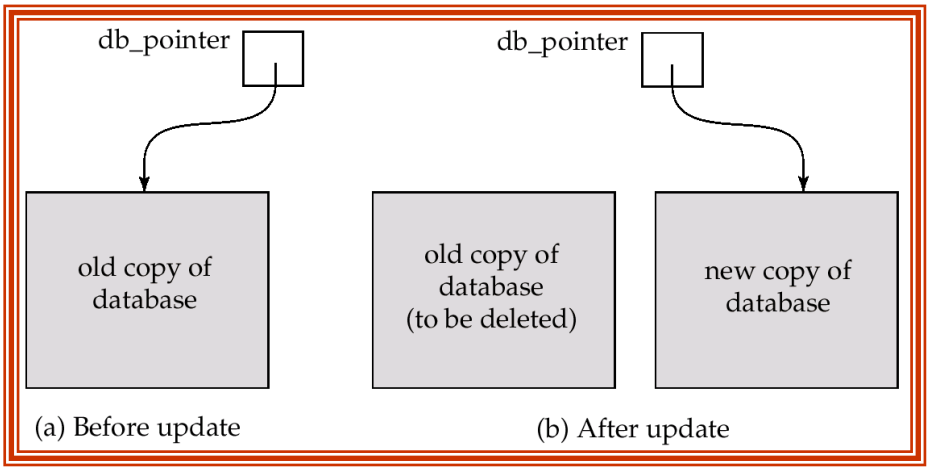
In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows:

First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.)
After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

The transaction is said to have been committed at the point where the updated db-pointer is written to disk.



We now consider how the technique handles transaction and system failures.

Maintaining Automicity

- First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- We can abort the transaction by just deleting the new copy of the database.
- Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer.
- Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

Maintaining Durability

- Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db- pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- Next, suppose that the system fails after db- pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 3

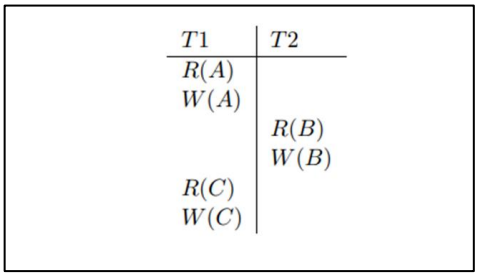
Concurrent Executions

Transaction and Schedule:

A **transaction** is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects.

In addition to reading and writing, each transaction must specify as its final action either commit (i.e., complete successfully) or abort (i.e., terminate and undo all the actions carried out thus far).

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, i.e. a schedule is a chronological sequence of execution of one or more transactions.



Sample Schedule

A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule.**

A schedule can be of two types:

1. Serial Schedule
2. Concurrent Schedule

In **Serial schedule**, transactions will be executed one after other, even the actions of different transactions are not interleaved.

In a **Concurrent Schedule** the actions of different transactions are interleaved to improve performance.

Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data. However, there are two good reasons for allowing concurrency:

• ***Improved throughput and resource utilization:***

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.

- The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• ***Reduced waiting time:***

- There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.
- Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as:

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as:

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1followed by T2.This execution sequence appears in Figure below. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure below, takes place, are $855 and $2145, respectively.

Thus, the total amount of money in accounts A and B—that is, the sum A + B—is preserved after the execution of both transactions

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 1—a serial schedule in which T_1 is followed by T_2.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure below. Again, as expected, the sum A+ B is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 2—a serial schedule in which T_2 is followed by T_1.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure below:

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Schedule 4—a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final values of accounts A and B are $950 and $2100, respectively. This final state is an inconsistent state, since we have gained $50 in the process of the concurrent execution. Indeed, the sum A + B is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.

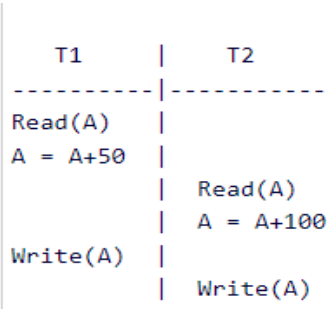
Potential Problems with Concurrent Execution:

1. Lost Update Problems(W-W Conflict)
2. Uncommitted Data / Dirty read problem(W-R Conflict)
3. Unrepeatable data / Inconsistent retrievals(W-R Conflict)

Lost Update Problems(W-W Conflict)

Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction.

Examples:

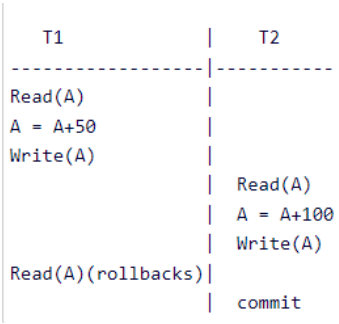


Result: T1's updates are lost.

Uncommitted Data / Dirty read problem(W-R Conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

One transaction might read an inconsistent state of data that's being updated by another.

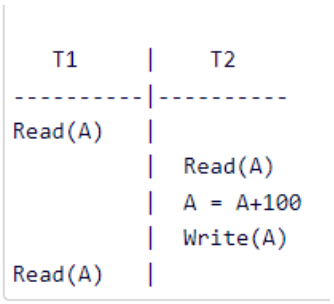


Result: T2 has a "dirty" value, that was never committed in T1 and doesn't actually exist in the database.

Unrepeatable data / Inconsistent retrievals(W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

When a single transaction reads the same row multiple times and observes different values each time. This occurs because another concurrent transaction has modified the row between the two reads.



Result: Within the same transaction, T1 has read two different values for the same data item. This inconsistency is the unrepeatable read.

Serializability Part I

Transaction and Schedule:

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order.

As an example, the schedule shown in Figure is serializable. Even though the actions of T1 and T2 are interleaved, the result of this schedule is equivalent .to running T1 (in its entirety) and then running T2. Intuitively, T1 's read and write of B is not influenced by T2's actions on A, and the net effect is the same

if these actions are 'swapped' to obtain the serial schedule Tl; T2.

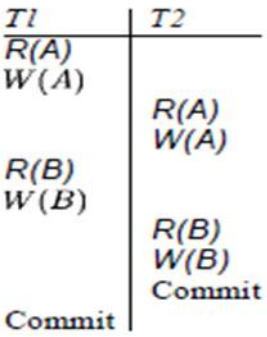


Figure 16.2 A Serializable Schedule

Two forms of serializabilty are

- **Conflict Serializabilty**
- **View Serializability.**

Conflict Serializability:

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij ,of transactions Ti and Tj, respectively ($i \neq j$). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj, regardless of the order.

2. Ii= read(Q), Ij= write(Q).If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus , the order of Ii and Ij matters.

3. Ii= write(Q), Ij= read(Q). the order of Ii and Ij matters for reasons similar to those of the previous case.

4. Ii= write(Q), Ij= write(Q). since both instructions are write operations, the order of these instructions does not affect either Ti or Tj. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both Ii and Ij are read instructions does the relative order of their execution not matter.

We say that Ii and Ij conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3, in Fig above. The write(A) instruction of T1 conflicts with the read(A) instruction of T2. However, the write(A) instruction of T2 does not conflict with the read(B) instruction of T1, because the two instructions access different data items.

Let Ii and Ij be consecutive instructions of a schedule S. If Ii and Ij are instructions of different transactions and Ii and Ij do not conflict, then we can swap the order of Ii and Ij to produce a new schedule S'. We expect S to be equivalent to S'. Since all instructions appear in the same order in both schedules except for Ii and Ij, whose order does not matter.

Since the write (A) instruction of Tin schedule 3 does not conflict with the read (B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5,shown in Figure below.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5—schedule 3 after swapping of a pair of instructions.

Regardless of the initial system state, schedules 3 and 5 both produce the same final system state. We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T1 with the read(A) instruction of T2
- Swap the write(B) instruction of T1 with the write(A) instruction of T2
- Swap the write(B) instruction of T1 with the read(A) instruction of T2.

The final result of these swaps, schedule 6 of Figure below, is a serial schedule

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6—a serial schedule that is equivalent to schedule 3.

This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.If a schedule S can be transformed into a schedule S' by a series of swaps of non conflicting instructions, we say that S and S' are conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

Finally, consider schedule 7 of Figure below; it consists of only the significant operations (that is, the read and write)of transactions T3 and T4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <T3,T4> or the serial schedule <T4,T3>. It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Precedence Graph for Testing Conflict Serializability in DBMS

A Precedence Graph or Serialization Graph is used commonly to test the Conflict Serializability of a schedule. It is a directed Graph (V, E) consisting of a set of nodes V = {T1, T2, T3…….Tn} and a set of directed edges E = {e1, e2, e3……em}.

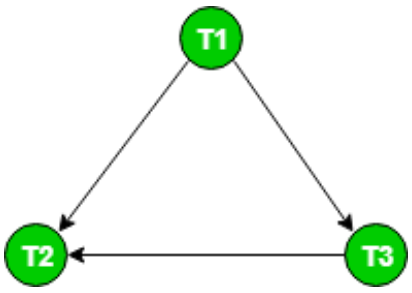
Consider some schedule of a set of transactions T1, T2, ..., Tn.

Precedence graph — a directed graph where

- the vertices are the transactions (names).
- there is an arc from Ti to Tj if the two transaction conflict, and Ti accessed the data item before Tj

T1	T2	T3
R(A)		
		R(B)
R(A)		
	W(B)	
		R(A)
	W(A)	

Original Schedule S



Precedence Graph – depicting Conflict pairs

T1	T2	T3
R(A)		
R(A)		
		R(B)
		R(A)
	W(B)	
	W(A)	

Serial Schedule

Result of Precedence Graph :

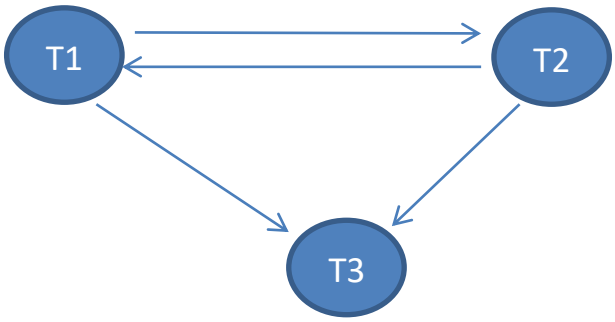
acyclic - the schedule is conflict serializable
cyclic - the schedule is not conflict serializable

If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S

In the above example, there is no cycle in the graph, means we can construct a serial schedule by swapping non-conflicting actions creating a conflict equivalent schedule.

Example – Non Conflict Serializable Schedules

T1	T2	T3
R(A)		
R(B)		
	W(A)	
W(A)		
		W(A)



T1	T2
R(A)	
	R(A)
	R(B)
W(A)	
R(B)	
	W(B)
W(B)	

Note: Both the precedence graphs result in a loop (cycle) Hence the schedules are not conflict serializable.

Serializability Part II

View Serializability:

Consider two schedules S and S' , where the same set of transactions participates in both schedules. The schedules S and S 'are said to be view equivalent if three conditions are met:

1. **Initial Read** For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S , also read the initial value of Q.

2. **Update** For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj, then the read(Q) operation of transaction Ti must, in schedule S value of Q that was produced by the same write(Q) operation of transaction Tj.

3. **Final Write:** For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q)operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is view serializable if it is view equivalent to a serial schedule. As an illustration, suppose that we augment schedule 7 with transaction T6,and obtain schedule 9 in Figure below:

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

2 Schedule 9—a view-serializable schedule.

Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule <T3,T4,T6>, since the one read(Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions T4 and T6 perform write(Q)operation without having performed a read(Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serializable schedule that is not conflict serializable.

Example of Schedule that is View Serializable

S1	
T1	**T2**
	R(A)
W(A)	
	W(A)
R(B)	
W(B)	

S2	
T1	**T2**
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	

- Both S1 and S2 have the same initial read of A by T2
- Both S1 and S2 have the final write of A by T2
- For intermediate writes/reads,
  in S2, T2 reads the value of A after T1 has written to it.
  in S1, T2 reads A which can be viewed as if it read the value after T1 (even though in actual sequence T2 read it before T1 wrote it). The important aspect is the view or effect is equivalent.
- B is read and then written by T1 in both schedules.

Considering the above conditions, S1 and S2 are view equivalent. Thus, if S1 is serializable, S2 is also view serializable.

Example 2:

Unlike conflict serializability, which cares about the order of conflicting operations, view serializability only cares about the final outcome.

Consider the given example, of two view serializable schedules, which results in the same final output.

S1

Transaction T1	Transaction T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

S2

Transaction T1	Transaction T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 6

Recoverability

Recoverability refers to the ability of a system to restore its state to a point where the integrity of its data is not compromised, especially after a failure or an error.

When multiple transactions are executing concurrently, issues may arise that affect the system's recoverability. The interaction between transactions, if not managed correctly, can result in scenarios where a transaction's effects cannot be undone, which would violate the system's integrity.

Importance of Recoverability:

The need for recoverability arises because databases are designed to ensure data reliability and consistency. If a system isn't recoverable:

- The integrity of the data might be compromised.
- Business processes can be adversely affected due to corrupted or inconsistent data.
- The trust of end-users or businesses relying on the database will be diminished.

Levels of Recoverability:

a

There are several levels of recoverability, each providing different guarantees about the state of the database after a crash or other failure.

Recoverable Schedules

A schedule is said to be recoverable if, for any pair of transactions Ti and Tj, if Tj reads a data item previously written by Ti, then Ti must commit before Tj commits. If a transaction fails for any reason and needs to be rolled back, the system can recover without having to rollback other transactions that have read or used data written by the failed transaction.

Suppose we have two transactions T1 and T2.

T1	T2
W(A)	
	R(A)
COMMIT	
	W(A)
	COMMIT

In the above schedule, T2 reads a value written by T1, but T1 commits before T2, making the schedule recoverable.

Recoverable schedules avoid the problem of cascading aborts, where the failure of one transaction requires the rollback of multiple other transactions.Example of a Recoverable Schedule

Non-Recoverable Schedules

A schedule is said to be non-recoverable (or irrecoverable) if there exists a pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, but Ti has not committed yet and Tj commits before Ti. If Ti fails and needs to be rolled back after Tj has committed, there's no straightforward way to roll back the effects of Tj, leading to potential data inconsistency.

Example of a Non-Recoverable Schedule

Again, consider two transactions T1 and T2.

T1	T2
W(A)	
	R(A)
	W(A)
	COMMIT
COMMIT	

In this schedule, T2 reads a value written by T1 and commits before T1 does. If T1 encounters a failure and has to be rolled back after T2 has committed, we're left in a problematic situation since we cannot easily roll back T2, making the schedule non-recoverable.

This is generally undesirable and is avoided in most systems.

Cascading Rollback

A cascading rollback occurs when the rollback of a single transaction causes one or more dependent transactions to be rolled back. This situation can arise when one transaction reads uncommitted changes of another transaction, and then the latter transaction fails and needs to be rolled back. Consequently, any transaction that has read the uncommitted changes of the failed transaction also needs to be rolled back, leading to a cascade effect.

Example of Cascading Rollback

Consider two transactions T1 and T2:

T1	T2
W(A)	
	R(A)
	R(A)
ABORT	
ROLLBACK	
	ROLLBACK

Here, T2 reads an uncommitted value of A written by T1. When T1 fails and is rolled back, T2 also has to be rolled back, leading to a cascading rollback. This is undesirable because it wastes computational effort and can complicate recovery procedures.

Cascadeless Schedules

A schedule is considered cascadeless if transactions only read committed values. This means, in such a schedule, a transaction can read a value written by another transaction only after the latter has committed. Cascadeless schedules prevent cascading rollbacks.

Example of Cascadeless Schedule

Consider two transactions T1 and T2:

T1	T2
W(A)	
COMMIT	
	R(A)
	W(B)
	COMMIT

In this schedule, T2 reads the value of A only after T1 has committed. Thus, even if T1 were to fail before committing (not shown in this schedule), it would not affect T2. This means there's no risk of cascading rollback in this schedule.

While this improves consistency and reduces the complexity of recovery, it may limit concurrency, as transactions must wait for others to commit before accessing the data.

DATABASE MANAGEMENT SYSTEMS

UNIT IV – TOPIC 7

Implementation of Isolation

Isolation is one of the core ACID properties of a database transaction, ensuring that the operations of one transaction remain hidden from other transactions until completion. It means that no two transactions should interfere with each other and affect the other's intermediate state.

Isolation Levels

Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.

There are four levels of transaction isolation defined by SQL -

1. Serializable

- This is the highest level of isolation, ensuring complete isolation from other transactions. It appears as if transactions are executed one after another, serially.
- Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

2. Repeatable Read

- This is the most restrictive isolation level.
- The transaction holds read locks on all rows it references.
- It holds write locks on all rows it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

3. Read Committed

A transaction can only see changes committed before it started; i.e. the any data is committed immediately after its access.

- This isolation level allows only committed data to be read.
- Thus it does not allows dirty read (i.e. one transaction reading of data immediately after written by another transaction).
- The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.

4. Read Uncommitted

Transactions may see uncommitted changes made by other transactions.

- It is lowest isolation level.
- In this level, one transaction may read not yet committed changes made by other transaction.
- This level allows dirty reads.

The choice of isolation level depends on the specific requirements of the application. Higher isolation levels offer stronger data consistency, but can result in longer lock times and increased contention, leading to decreased concurrency and performance.

Lower isolation levels provide more concurrency, but can result in data inconsistencies**.**

Implementation of Isolation

Implementing isolation typically involves concurrency control mechanisms. Here are common mechanisms used:

1. Locking Mechanisms

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

2. Timestamp-based Protocols

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only access the database if they respect the timestamp order, ensuring older transactions get priority.

Lock Based Protocols:

In lock-based protocol, a transaction cannot access or write data unless it obtains a suitable lock. The Two-Phase Locking Protocol ensures transaction consistency by employing a set of rules. Transactions must obtain appropriate locks before accessing or modifying data, preventing conflicts and maintaining data integrity throughout the transaction's lifecycle.

Locks are essential in a database system to ensure:

Consistency: Without locks, multiple transactions could modify the same data item simultaneously, resulting in an inconsistent state.

Isolation: Locks ensure that the operations of one transaction are isolated from other transactions, i.e., they are invisible to other transactions until the transaction is committed.

Concurrency: While ensuring consistency and isolation, locks also allow multiple transactions to be processed simultaneously by the system, optimizing system throughput and overall performance.

Avoiding Conflicts: Locks help in avoiding data conflicts that might arise due to simultaneous read and write operations by different transactions on the same data item.

Preventing Dirty Reads: With the help of locks, a transaction is prevented from reading data that hasn't yet been committed by another transaction.

Types of Locks Used in Transaction Control

In transaction control, there are two common types of locks shared lock and exclusive lock. Here is a brief overview about it:

1. Shared lock: The transactions can only read the data items in a shared lock.

While a transaction attempting to update the data will be unable to do so until the shared lock is freed, a transaction attempting to read the same data will be allowed to do so.

Read lock is another name for shared lock, which is exclusively used for reading data objects.

2. Exclusive lock: The transactions can read and write to the data items in an exclusive lock.

When a statement updates data, its transaction has an exclusive lock on the modified data, preventing access by other transactions, until the transaction holding the lock commits or rolls it back, the lock is in effect.

The difference between shared lock and exclusive lock

Shared Lock	Exclusive Lock
Shared lock is used for when the transaction wants to perform read operation.	Exclusive lock is used when the transaction wants to perform both read and write operation.
Any number of transactions can hold shared lock on an item.	But exclusive lock can be hold by only one transaction.
Using shared lock data item can be viewed.	Using exclusive lock data can be inserted or deleted.

DATABASE MANAGEMENT SYSTEMS

UNIT IV – TOPIC 8

Lock Based Protocols

What is Two-phase locking (2PL)?

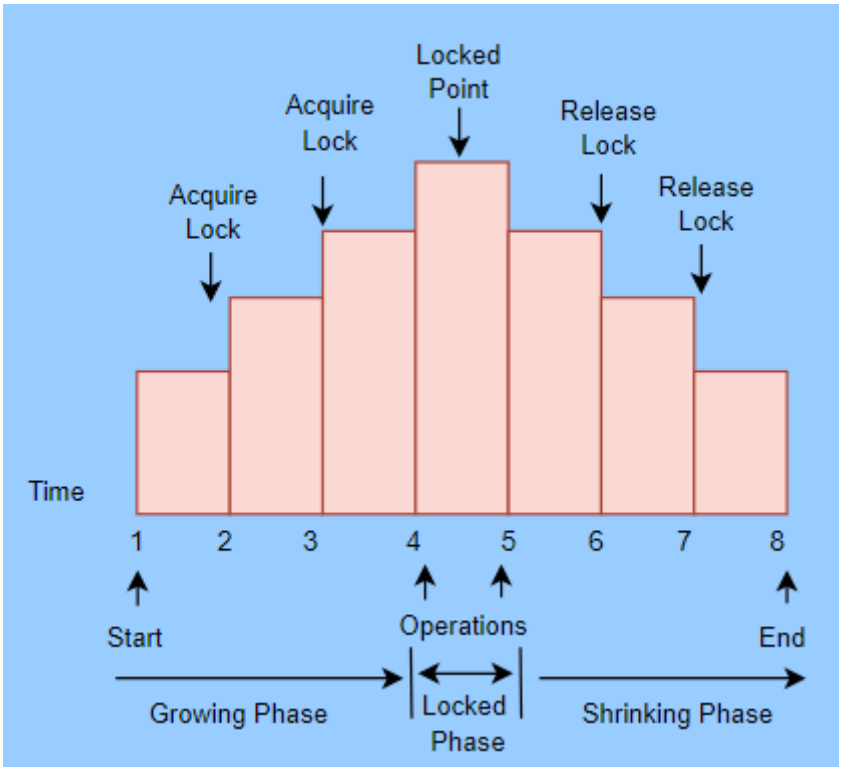
The two-phase locking divides the transaction execution phase into three components.

- When the transaction's execution begins in the first part, it requests permission for the lock it requires.
- The transaction then obtains all of the locks in the second part. The third phase begins when the transaction's first lock is released.
- The transaction cannot demand any new locks in the third phase. It only unlocks the locks that have been acquired.

A transaction follows the two-phase locking protocol if locking and unlocking can be done in two phases.

The two phases of the two-phase protocol are:

- **Growing phase:** During the growth phase, new locks on data items may be acquired, but none can be released.
- **Shrinking phase:** Existing locks may be released, but no new locks can be acquired during the shrinking phase.



The depicted diagram illustrates the growing phase, where locks are acquired until all necessary transaction locks are obtained, defining the Lock-Point. Beyond this point, transactions transition into the Shrinking phase, ensuring a structured lock-based protocol.

Lock Point
It is a point at which the growing phase comes to a close, i.e., when a transaction obtains the last lock it requires.

Let's look at an example:

	T1	T2
1	Lock - S(A)	
2		Lock - S(A)
3	Lock - X(B)	
4	…….	…….
5	Unlock(A)	
6		Lock - X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	…….

The way unlocking and locking works in two-phase locking is:

Transaction T1:
- The growing phase is from steps 1-3.
- The shrinking phase is from steps 5-7.
- The locking point is at point 3.

Transaction T2:
- The growing phase is from steps 2-6.
- The shrinking phase is from steps 8-9.
- The locking point is at point 6.

The type mentioned above of 2-PL is Basic 2PL. It ensures Serializability, but it does not prevent Cascading Rollback and Deadlock.

Advantages and Disadvantages of Two-Phase Locking Protocol

Pros of Two-Phase Locking (2PL)

Ensures Serializability: 2PL guarantees conflict-serializability, ensuring the consistency of the database.

Concurrency: By allowing multiple transactions to acquire locks and release them, 2PL increases the concurrency level, leading to better system throughput and overall performance.

Avoids Cascading Rollbacks: Since a transaction cannot read a value modified by another uncommitted transaction, cascading rollbacks are avoided, making recovery simpler.

Cons of Two-Phase Locking (2PL)

Deadlocks: Two or more transactions wait indefinitely for a resource locked by the other.

Reduced Concurrency (in certain cases): If one transaction holds a lock for a long time, other transactions needing that lock will be blocked.

Overhead: There's a time cost associated with acquiring and releasing locks, and memory overhead for maintaining the lock table.

Starvation: It's possible for some transactions to get repeatedly delayed if other transactions are continually requesting and acquiring locks

Types of Two-Phase Locking Protocol
Two Phase Locking is classified into three types :

1. Strict two-phase locking protocol

- Exclusive(X) locks held by the transaction be released after committing in addition to the lock being 2-Phase.
- Ensures that the schedule is recoverable and cascadeless.

2. Rigorous two-phase locking protocol

- Exclusive(X) and Shared(S) locks held by the transaction be released after Transaction Commits, in addition to the lock being 2-Phase. This means there is no explicit shrinking phase where locks are released during the transaction. Instead, locks are released only at the end of the transaction.

- It ensures that schedule is recoverable and cascadeless.

3. Conservative two-phase locking protocol

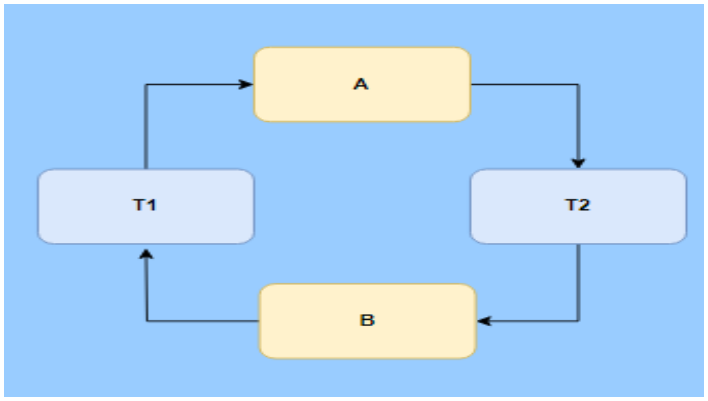
- Static 2-PL enforces pre-locking of all accessed resources by declaring read-set and write-set before transaction execution.
- Transaction waits if necessary items can't be locked, avoiding locking any items if a predeclared requirement isn't met.

Deadlock in a two-phase locking

Deadlocks are possible in a 2PL. You can detect the deadlocks by drawing the precedence graph of the transactions schedule, and if you find a loop, then there is a deadlock situation.

Deadlocks are needed to be resolved to complete the transaction. A deadlock is resolved by aborting a transaction of the given loop and breaking it.

Let's understand deadlock in a two-phase locking protocol with the help of an example.
Suppose there are two transactions, T1 and T2. If the completion of transaction T1 is dependent on the completion of transaction T2 and the completion of transaction, T2 is dependent on the completion of transaction T1. This results in a state of deadlock. As shown in the figure below:



Phase 1 - Lock Request:

- Process T1 acquires a lock on Resource A.
- Process T2 acquires a lock on Resource B.

Phase 2 - Lock Release:

- Process T1 now needs Resource B to proceed further, but it cannot release Resource A until it has both resources due to the two-phase locking rule.
- Process T2 similarly needs Resource A to continue, but it's holding onto Resource B.

DATABASE MANAGEMENT SYSTEMS

UNIT IV – TOPIC 9

Timestamp Based Protocols

Timestamp-based protocols in database management systems (DBMS) are concurrency control mechanisms used to ensure the consistency of the database while allowing concurrent transactions.

They use timestamps to order the operations of transactions in such a way that the database remains in a consistent state, following a predetermined serialization order. Here are the key concepts and protocols related to timestamp-based concurrency control:

Each transaction is issued a timestamp (can be the system's clock time or a logical counter that increments with each new transaction) when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that **TS(Ti) <TS(Tj).**

Schedules managed using timestamp based protocol are serializable just like the two phase protocols.

Since older transactions are given priority which means no transaction has to wait for longer period of time that makes this protocol free from deadlock.

Key Concepts

1. **Timestamp**: A unique identifier assigned to each transaction. It usually represents the time at which the transaction started.
    - **Transaction Timestamp (TS)**: Denoted as TS(Ti) for transaction Ti.
    - **Read Timestamp (RTS)**: Denoted as RTS(X) for a data item X, indicating the largest timestamp of any transaction that successfully read X.
    - **Write Timestamp (WTS)**: Denoted as WTS(X) for a data item X, indicating the largest timestamp of any transaction that successfully wrote X.

Basic Timestamp Ordering Protocol

The basic idea is to ensure that transactions are executed in a serial order consistent with their timestamps. For this:

1. **Read Operation**:
    - Transaction TiT_iTi requests to read a data item X.
    - If TS(Ti)<WTS(X), it means a younger transaction has already written X, so Ti is rolled back.
    - If TS(Ti)≥WTS(X), the read is allowed, and RTS(X) is updated to TS(Ti), if TS(Ti)>RTS(X).

2. **Write Operation**:
    - Transaction Ti requests to write a data item X.
    - If TS(Ti)<RTS(X), it means a younger transaction has already read X, so Ti is rolled back.
    - If TS(Ti)<WTS(X), it means a younger transaction has already written X, so Ti is rolled back.
    - If TS(Ti)≥RTS(X) and TS(Ti)≥WTS(X), the write is allowed, and WTS(X) is updated to TS(Ti)

Advantages

- **Serializability**: Ensures serializable schedules, maintaining database consistency.
- **Deadlock-free**: Avoids deadlocks since transactions are never made to wait; they are either allowed to proceed or rolled back.

Disadvantages

- **Cascading Rollbacks**: A rollback of a transaction may cause other dependent transactions to roll back.
- **Long Transactions**: Long-running transactions may repeatedly get rolled back if newer transactions conflict with them.
- **Storage Overhead**: Requires additional storage for timestamps and may involve maintaining multiple versions of data items.

Implementation Example

Let's consider an example to illustrate how a basic timestamp ordering protocol works:

Note: The initial values of the Write Timestamp (WTS) and Read Timestamp (RTS) for a data item X are typically set to zero or negative infinity ($-\infty$) before any transactions have accessed the data item. This initialization ensures that the first read or write operation on X by any transaction will be allowed.

- Suppose there are two transactions T1 and T2 with timestamps TS(T1) = 1 and TS(T2) = 2. Let RTS(X) =0 and WTS(X)=0.
-
- **T1 reads X:**
    - Since TS(T1)≥WTS(X), T1 is allowed to read X.
    - Update RTS(X) to 1.
    -
- **T2 writes X:**
    - Since TS(T2)≥RTS(X) and TS(T2)≥WTS(X) T2 is allowed to write X.
    - Update WTS(X) 2.
- **T1 writes X:**
    - Since TS(T1)<WTS(X) T1 is rolled back because a newer transaction T2 has already written X.

In this way, timestamp-based protocols manage the order of transaction operations to maintain consistency while allowing concurrent execution.

Thomas' Write Rule:

Thomas' Write Rule is an optimization in timestamp-based concurrency control protocols. It allows certain write operations to be ignored instead of causing the transaction to roll back, thereby improving the system's performance and reducing unnecessary rollbacks.

If a transaction Ti attempts to write a data item X and TS(Ti)<WTS(X), instead of rolling back Ti, the write operation is simply ignored.

Example Scenario:

Let's illustrate Thomas' Write Rule with the above example involving two transactions T1 and T2.

The second attempt T1 wants to Write X after T2 has written:

- T1 requests to write X.
- The basic timestamp ordering protocol would check TS(T1)<WTS(X). Here, TS(T1)=1 and WTS(X)=2. Since TS(T1)<WTS(X), the basic protocol would roll back T1.
- With Thomas' Write Rule, instead of rolling back T1, the write operation is ignored because the write by T1 is considered obsolete due to the later write by T2.

Why Thomas' Write Rule Works:

Thomas' Write Rule optimizes concurrency control by reducing unnecessary rollbacks:

- **Efficiency**: By ignoring obsolete writes, the system avoids the overhead of rolling back transactions that do not affect the final state of the database.
- **Maintains Serializability**: It ensures the serializability of transactions by only allowing the most recent write to a data item to take effect.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 10
Multiple Granularity

A lock guarantees exclusive use of a data item to a current transaction. A transaction acquires a lock prior to data access; the lock is released when the transaction is completed so that another transaction can lock the data item for its exclusive use. The use of locks based on the assumption that conflict between transactions is likely to occur is often referred to as pessimistic locking. During a transaction; the database might be in a temporary inconsistent state when several updates are executed. Therefore, locks are required to prevent another transaction from reading inconsistent data..

Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is managed by a lock manager, which is responsible for assigning and policing the locks used by the transactions.

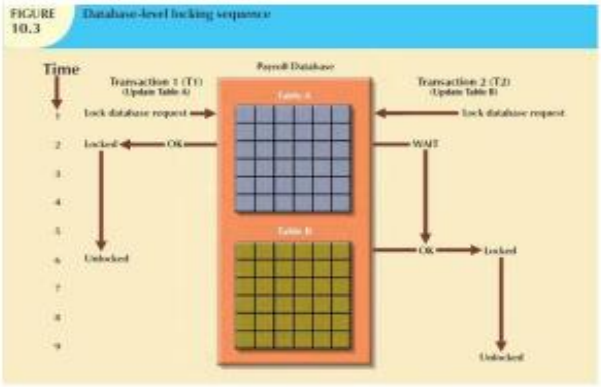
Lock Granularity:

Lock granularity indicates the level of lock use. Multiple granularity locking manages locks at various levels of a database hierarchy, allowing different transactions to lock resources at different granular levels, such as the entire database, tables, pages, or individual records or even fields.

Granularity Levels

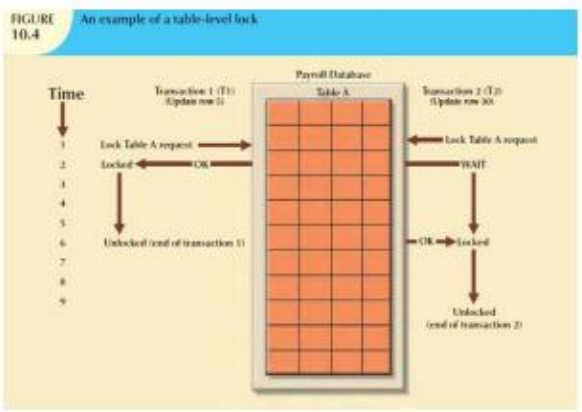
Database Level: The highest level of granularity, locking the entire database.

In a database-level lock, the entire database is locked, thus preventing the use of any tables in the database by transaction T2 while transaction Tl is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs. Note that because of the database-level lock, transactions T1 and T2 cannot access the same database concurrently even when they use different tables.



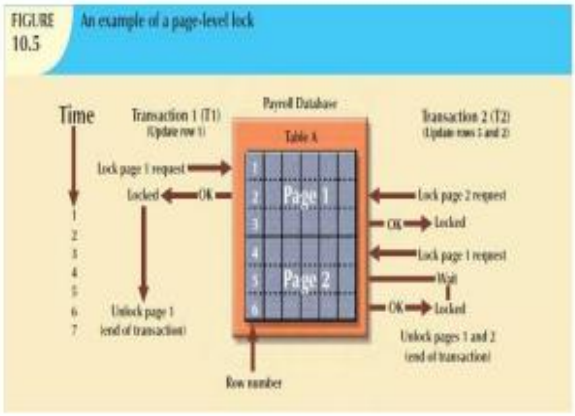
File/Table Level: Locks an entire file or table.

In a table-level lock, the entire table is locked, preventing access to any row by transaction T2 while transaction T1 is using the table. If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables. Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table.



Consequently, table-level locks are not suitable for multiuser DBMSs. Note that in Figure, transactions T1 and T2 cannot access the same table even when they try to use different rows; T2 must wait until T1 unlocks the table.

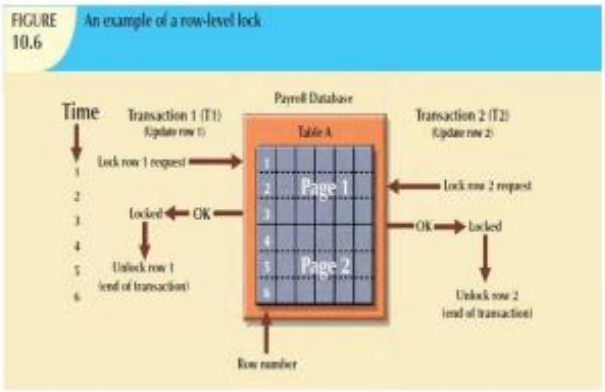
Page Level: Locks a specific page within a file or table.



In a page-level lock, the DBMS will lock an entire diskpage. A diskpage, or page, is the equivalent of a diskblock, which can be described as a directly addressable section of a disk. A page has a fixed size, such as 4K, 8K, or 16K. A table can span several pages, and a page can contain several rows of one or more tables. Page-level locks are currently the most frequently used multiuser DBMS locking method.

An example of a page-level lock is shown in the Figure. Note that T1 and T2 access the same table while locking different diskpages. If T2 requires the use of a row located on a page that is locked by T1, T2 must wait until the page is unlocked by T1.

Record Level: Locks a specific record within a page.



A row-level lock is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table even when the rows are located on the same page. Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction.

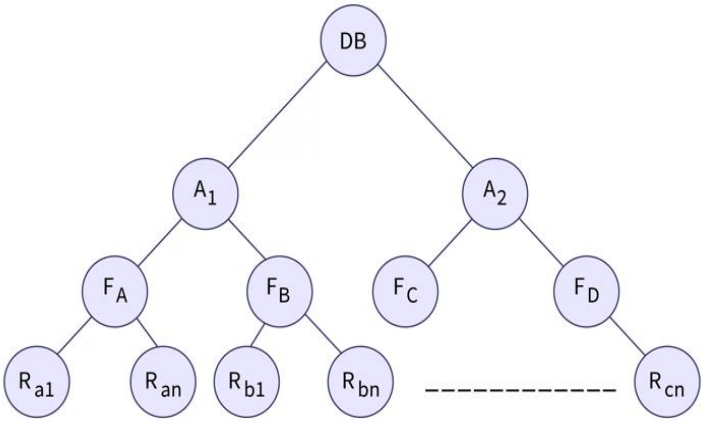
Field Level: The most granular level, locking specific fields within a record.

The field-level lock allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row. Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead and because the row-level lock is much more useful in practice.

Hierarchical Locking Structure

Locks are organized in a hierarchical tree structure, with each node representing a different level of granularity.

Parent-child relationships ensure that locks acquired at a higher level (parent) must be compatible with locks at lower levels (child).



In addition to shared (S) and exclusive (X) locks, multiple-granularity locking protocols also use two new kinds of locks, called **"intention locks**." These locks indicate a transaction's intention to acquire a finer-grained lock in the future.

Types of Intention Locks

Intention Shared (IS): Indicates intention to acquire shared locks at a lower level. When a Transaction needs S lock on a node "K", the transaction would need to apply IS lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IS mode, it indicates that some of its descendent nodes must be locked in S mode.

Example: Suppose a transaction wants to read a few records from a table but not the whole table. It might set an IS lock on the table, and then set individual S locks on the specific rows it reads.

Intention Exclusive (IX): Indicates intention to acquire exclusive locks at a lower level. When a Transaction needs X lock on a node "K", the transation would need apply IX lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IX mode, it indicates that some of its descendent nodes must be locked in X mode.

Example: If a transaction aims to update certain records within a table, it may set an IX lock on the table and subsequently set X locks on specific rows it updates.

Shared Intention Exclusive (SIX): A combination of shared lock at the current level and intention to acquire exclusive locks at a lower level. When a node is locked in SIX mode; it indicates that the node is explicitly locked in S mode and Ix mode. So, the entire tree rooted by that node is locked in S mode and some nodes in that are locked in X mode. This mode is compatible only with IS mode.

Example: Suppose a transaction wants to read an entire table but also update certain rows. It would set a SIX lock on the table. This tells other transactions they can read the table but cannot update it until the SIX lock is released. Meanwhile, the original transaction can set X locks on specific rows it wishes to update.

Compatibility Matrix with Lock Modes in multiple granularity

A compatibility matrix defines which types of locks can be held simultaneously on a database object. Here's a simplified matrix:

	IS	IX	S	SIX	X
IS	Y	Y	Y	N	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	N	N	N	N	N
X	N	N	N	N	N

The Scheme operates as follows:-

- A Transaction must first lock the Root Node and it can be locked in any mode.
- Locks are granted as per the Compatibility Matrix indicated above.
- A Transaction can lock a node in S or IS mode if it has already locked all the predecessor nodes in IS or IX mode.
- A Transaction can lock a node in X or IX or SIX mode if it has already locked all the predecessor nodes in SIX or IX mode.
- A transaction must follow two-phase locking. It can lock a node, only if it has not previously unlocked a node. Thus, schedules will always be conflict-serializable.
- Before it unlocks a node, a Transaction has to first unlock all the children nodes of that node**. Thus, locking will proceed in top-down manner and unlocking will proceed in bottom-up manner. This will ensure the resulting schedules to be deadlock-free.**

Example

1. Transaction T1 wants to read a record R in table T:
    - Acquire IS lock on database.
    - Acquire IS lock on table T.
    - Acquire S lock on record R.

2. Transaction T1 and T2 operate on the same table but different records:
    - T1 acquires
        - IS lock on database,
        - IS lock on table, and
        - S lock on record R1.
    - T2 can concurrently acquire
        - IS lock on database,
        - IS lock on table, and
        - S lock on record R2.

Advantages and Disadvantages

Advantages

- Improved Concurrency: Allows multiple transactions to operate concurrently on different parts of the database.
- Reduced Lock Contention: By allowing more granular locking, it reduces the chance of lock contention.
- Flexibility: Provides flexibility in managing locks at various levels of granularity.

Disadvantages

- Complexity: More complex to implement and manage compared to single-level locking.
- Overhead: Increased overhead due to maintaining and checking multiple levels of locks

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 11
Validation Based Protocols

Validation-based protocols, also known as Optimistic Concurrency Control (OCC), are a set of techniques that aim to increase system concurrency and performance by assuming that conflicts between transactions will be rare.

Instead of preventing conflicts through locking, these protocols validate transactions at commit time to ensure that no conflicts have occurred.

In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.

The Three Phases:

1. Read Phase

- Transactions execute and read data items into a local workspace.
- No updates are made to the database during this phase.

2. Validation Phase

- At the end of the read phase, the transaction enters the validation phase.
- The system checks whether committing the transaction would violate serializability.
- Validation involves comparing the transaction's read set and write set against those of other transactions.

3. Write Phase

- If the transaction passes validation, it proceeds to the write phase.
- The updates from the local workspace are applied to the database.
- If the transaction fails validation, it is rolled back and may be restarted.

Validation Criteria

Timestamp Ordering

- Each transaction is assigned a timestamp when it starts.
- The validation ensures that the transaction's operations can be serialized in the order of their timestamps.

Validation Rules

Rule 1: For two transactions Ti and Tj, where Ti completes its read phase before Tj starts, no validation is needed since they do not overlap.

Rule 2: If Ti overlaps with Tj and Ti precedes Tj in the validation order, Tj's read set must not intersect with Ti's write set.

Rule 3: If Ti and Tj overlap and Tj precedes Ti in the validation order, Ti's write set must not intersect with Tj's read set and write set.

Example Scenarios

Example 1: Non-Conflicting Transactions

- **T1** reads data items A and B, then writes to B.
- **T2** reads data items C and D, then writes to D.
- Since T1 and T2 operate on different data items, they will both pass validation and commit successfully.

Example 2: Conflicting Transactions

- **T1** reads data items A and B, then writes to B.
- **T2** reads data items B and C, then writes to C.
- During validation, if T2's read set overlaps with T1's write set (both accessing B), validation may fail depending on the order and timing of the transactions.

Advantages and Disadvantages

Advantages

- **Increased Concurrency**: Transactions proceed without waiting for locks, increasing parallelism.
- **Reduced Deadlocks**: Since locks are not used extensively, the risk of deadlocks is minimized.
- **Simplicity**: The protocol is simpler to implement for read-mostly workloads where conflicts are rare.

Disadvantagess

- **High Overhead**: Rolling back and restarting transactions that fail validation can be costly.
- **Resource Intensive**: Maintaining read and write sets and performing validation checks require additional resources.
- **Not Suitable for High-Conflict Workloads**: In environments with frequent conflicts, the overhead of rollbacks can outweigh the benefits.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 12
Recovery and Atomicity Part - I

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.

Database recovery means the process of restoring the database to a correct state in case of failures.

Failures can be of different types, such as:

Transaction Failure: When a transaction cannot compete successfully due to logical errors, system errors, or data errors.

System Crash: When the DBMS crashes due to hardware failures, software bugs, or power outages.

Media Failure: When the storage media (like hard disks) fail, leading to loss of data.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

Types of techniques for recovery:

Maintaining logs of each transaction - In case of a failure, the system can use these logs to redo or undo transactions to restore the database to a consistent state.

Checkpointing: Periodically saves the state of the database. During a failure, the system can roll back to the last checkpoint and then apply the logs to reach a consistent state.

Maintaining shadow paging - If a failure occurs, the system can revert to the shadow copy.

Log Based Recovery:

Log-based recovery is a widely used approach in database management systems to recover from system failures and maintain atomicity and durability of transactions.

The fundamental idea behind log-based recovery is to keep a log of all changes made to the database, so that after a failure, the system can use the log to restore the database to a consistent state.

Transaction Log

A log in DBMS is a crucial component used for ensuring data integrity, consistency, and facilitating recovery processes.

A log, also known as a **transaction log or write-ahead log (WAL),** is a sequential record that tracks all the changes made to the database. It records information about transactions before they are committed to the database. This information can be used to redo completed transactions or undo incomplete transactions in the event of a failure.

A log typically contains the following information:

- Transaction ID: A unique identifier for each transaction.
- Transaction Type: The type of operation, such as INSERT, UPDATE, DELETE, etc.
- Affected Data: Details of the data being modified, - table name and the specific rows and columns.
- Beoref-Image: The state of the data before the transaction (used for undo operations).
- After-Image: The state of the data after the transaction (used for redo operations).
- Timestamp: The time at which the transaction was recorded in the log.
- Log Sequence Number (LSN): A unique identifier for each log entry, ensuring the sequence of transactions can be tracked.

For every transaction that modifies the database, an entry is made in the log. This entry typically includes:

- Transaction ID: A unique identifier for the transaction.
- Data item identifier: Identifier for the specific item being modified.
- OLD value: The value of the data item before the modification.
- NEW value: The value of the data item after the modification.

We represent an update log record as **<Ti , Xj , V1, V2>,** - transaction Ti has performed a write on data item Xj. Xj had value V1 before the write, and has value V2 after the write.

Some other types of log records are:

<Ti start>. Transaction Ti has started.

<Ti commit>. Transaction Ti has committed.

<Ti abort>. Transaction Ti has aborted.

How Log-Based Recovery works

1. Transaction Logging:

For every transaction that modifies the database, an entry is made in the log.

2. Writing to the Log

Before any change is written to the actual database (on disk), the corresponding log entry is stored. This is called the Write-Ahead Logging (WAL) principle. By ensuring that the log is written first, the system can later recover and apply or undo any changes.

3. Checkpointing

A checkpoint is a point of synchronization between the database and its log. At the time of a checkpoint:

All the changes in main memory (buffer) up to that point are written to disk.

A special entry is made in the log indicating a checkpoint. This helps in reducing the amount of log that needs to be scanned during recovery.

4. Recovery Process

Redo: If a transaction is identified (from the log) as having committed but its changes have not been reflected in the database (due to a crash before the changes could be written to disk), then the changes are reapplied using the 'After Image' from the log.

Undo: If a transaction is identified as not having committed at the time of the crash, any changes it made are reversed using the 'Before Image' in the log to ensure atomicity.

5. Commit/Rollback

Once a transaction is fully complete, a commit record is written to the log. If a transaction is aborted, a rollback record is written, and using the log, the system undoes any changes made by this transaction.

Deferred-Write Technique

Transaction recovery procedures generally make use of deferred-write and write-through techniques.

Deferred-write technique (also called a deferred update),

The transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only after the transaction reaches its commit point, using information from the transaction log.

The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and to update the database, using the ―after values in the transaction log.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, nothing needs to be done because the database was never updated.

Write-through Technique

Write-through technique (also called an immediate update),

the database is immediately updated by transaction operations during the transaction‘s execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a ROLLBACK or undo operation needs to be done to restore the database to a consistent state. In that case, the ROLLBACK operation will use the transaction log ―before values.

The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data were physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS redoes the transaction, using the ―after values of the transaction log.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, the DBMS uses the transaction log records to ROLLBACK or undo the operations, using the before values in the transaction log.

Benefits of Log-Based Recovery

- **Atomicity:** Guarantees that even if a system fails in the middle of a transaction, the transaction can be rolled back using the log.

- **Durability:** Ensures that once a transaction is committed, its effects are permanent and can be reconstructed even after a system failure.

- **Efficiency:** Since logging typically involves sequential writes, it is generally faster than random access writes to a database.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 13
Recovery and Atomicity Part - II

Shadow Paging

Shadow paging is a recovery mechanism used in Database Management Systems (DBMS) to ensure data integrity and consistency without requiring extensive logging.
The basic idea is to maintain two versions of the database pages:

current pages - that are being actively modified and the

shadow pages - that represent a stable, consistent state of the database

Overview of Shadow Paging

In shadow paging, the database is divided into fixed-size blocks or pages. Two page tables are maintained:

Current Page Table (CPT): This table points to the current version of the pages.

Shadow Page Table (SPT): This table points to the shadow (or stable) version of the pages.

Basic Operation:

1. **Initialization:**
   - At the start, the SPT and CPT are identical, pointing to the same set of pages.
   - The SPT is written to a non-volatile storage medium to ensure it can be recovered after a crash.
2. **Transaction Start:**
   - When a transaction begins, any modification to a page results in the creation of a new version of that page.
   - The original page remains unchanged, preserving the pre-transaction state.
3. **Page Modification:**
   - When a transaction modifies a page, the DBMS creates a copy of that page.
   - The CPT is updated to point to the new version of the page, while the SPT continues to point to the old version
4. **Commit Operation:**
   - When the transaction commits, the CPT is written to the non-volatile storage.
   - At this point, the SPT can be updated to match the CPT, making the new page versions the new stable state.
5. **Rollback Operation:**
   - If a transaction needs to be rolled back, the DBMS simply discards the modified pages and the CPT.
   - The SPT remains unchanged, and the database reverts to its pre-transaction state.

Advantages of Shadow Paging

- **No Undo Logging Required:** Since the original pages are never overwritten, there is no need to maintain undo logs.
- **Consistency:** The database remains in a consistent state because the shadow pages represent a stable snapshot.
- **Simple Rollback**: Rolling back a transaction is straightforward as it simply involves discarding the current pages and reverting to the shadow.

Disadvantages of Shadow Paging

- **Copy Overhead:** Every page modification requires copying the page, which can be resource-intensive.
- **Space Utilization:** Maintaining two versions of pages can lead to higher storage requirements.
- **Checkpointing Complexity:** Periodically updating the SPT to match the CPT can be complex and time-consuming, especially for large databases.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 14
Recovery with Concurrent Transactions

Concurrency execution allows multiple transactions to execute at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

- Interaction with concurrency control
- Transaction rollback
- Checkpoints
- Restart recovery

Interaction with concurrency control

Concurrency control mechanisms manage how transactions interact with each other to maintain database consistency. Common techniques include:

- **Locking:** Transactions acquire locks on data items to prevent other transactions from accessing or modifying them concurrently.
- **Timestamp Ordering:** Transactions are ordered based on their timestamps to ensure serializability.
- **Optimistic Concurrency Control:** Transactions proceed without locking, based on the assumption that conflicts are rare. Conflicts are detected and resolved at commit time.

During recovery, these mechanisms play a crucial role in ensuring that transactions are correctly rolled back or redone.

If a failure occurs, the system uses information from the logs and transaction metadata managed by concurrency control to determine which transactions need to be undone (rolled back) or redone.

This ensures that the database returns to a consistent state without violating transaction isolation and consistency guarantees.

<

Transaction rollback

- Transaction rollback involves undoing the effects of a transaction that has not yet been committed due to an error or abort condition. It restores the database to its state before the transaction began.
- When a system failure occurs, transactions that were in progress or incomplete need to be rolled back during recovery.

- The system uses transaction logs, which record the sequence of operations performed by each transaction, to identify and reverse the changes made by these incomplete transactions.
- This ensures that any uncommitted changes do not persist in the database, maintaining consistency and integrity.

Checkpointing:

- Checkpoints are periodic snapshots of the database state that are recorded at specific intervals. They capture the committed state of the database at those points in time.
- During recovery, checkpoints serve as reference points to minimize the amount of log data that needs to be processed.
- Instead of replaying all transactions from the beginning of the logs after a crash, the system can start recovery from the most recent checkpoint.
- This optimization reduces the recovery time significantly, as only the transactions that occurred after the last checkpoint need to be reprocessed (redo) or undone (undo).
- Checkpoints thus facilitate faster recovery and reduce the potential for data loss or inconsistency.

Restart Recovery:

- Restart recovery is the process of bringing the database system back to a consistent state after it has been restarted following a failure. It involves analyzing the transaction logs and applying necessary operations to restore data consistency.
- After a system crash and subsequent restart, the database system initiates restart recovery.
- It reads the transaction logs to reconstruct the sequence of committed and incomplete transactions.
- For committed transactions, redo operations are applied to reapply their changes to the database, ensuring that all committed updates are reflected.
- For incomplete transactions (those in progress at the time of the crash), undo operations are applied to revert any partially applied changes and restore the database to a consistent state.
- Restart recovery ensures that the database maintains its integrity and consistency despite the interruption caused by the system failure.

Recovery after a System Crash:

Checkpoints are performed as before, except that the checkpoint log record is now of the form

< checkpoint L> where L is the list of transactions active at the time of the checkpoint

We assume no updates are in progress while the checkpoint is carried out

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1) REDO PHASE 2) UNDO PHASE

REDO Phase:

In the redo phase, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred.

This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back.

The specific steps taken while scanning the log are as follows:

a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the <checkpoint L> log record.

b. Whenever a normal log record of the form <Ti , Xj , V1, V2>, or a redo-only log record of the form <Ti , Xj , V2> is encountered, the operation is redone; that is, the value V2 is written to data item Xj .

c. Whenever a log record of the form <Ti start> is found, Ti is added to undo-list.

d. Whenever a log record of the form <Ti abort> or <Ti commit> is found, Ti is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

UNDO Phase:

In the undo phase, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

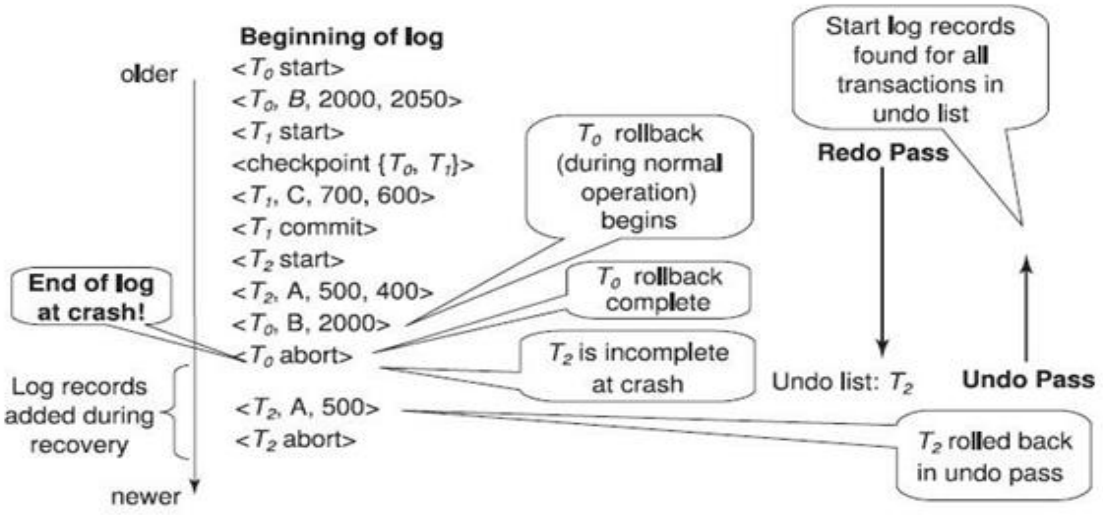
a. Whenever it finds a log record belonging to a transaction in the undo list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a <Ti start> log record for a transaction Ti in undo-list, it writes a <Ti abort> log record to the log, and removes Ti from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found <Tistart>log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

EXAMPLE



Example of logged actions, and actions during recovery.

The above figure shows an example of actions logged during normal operation, and actions performed during failure recovery. In the log shown in the figure, transaction T1 had committed, and transaction T0 had been completely rolled back, before the system crashed. Observe how the value of data item B is restored during the rollback of T0. Observe also the checkpoint record, with the list of active transactions containing T0 and T1.

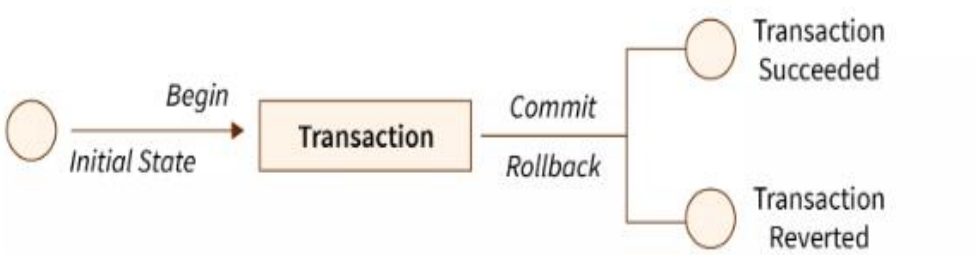
When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains T0 and T1; T1 is removed first when its commit log record is found, while T2 is added when its start log record is found. Transaction T0 is removed from undo-list when its abort log record is found, leaving only T2 in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of T2 updating A, the old value of A is restored, and a redo-only log record written to the log. When the start record for T2 is found, an abort record is added for T2. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

DATABASE MANAGEMENT SYSTEMS
UNIT IV – TOPIC 15
Transaction Control Language (TCL) Commands

TCL stands for Transaction Control Language in SQL. Transaction Control Language (TCL) is a set of special commands that deal with the transactions within the database. Basically, they are used to manage transactions within the database. TCL commands ensure the integrity and consistency of data by allowing to control the behavior of transactions.

A transaction is a unit of work that is performed against a database in SQL. In other words, a transaction is a single, indivisible database action.

In SQL, each transaction begins with a particular set of task and ends only when all the tasks in the set is completed successfully. However, if any (or a single) task fails, the transaction is said to fail.



TCL Commands – Save point Commit and Roll back:

COMMIT command:

COMMIT command is used to permanently save any transaction into the database.
When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.
To avoid that, we use the COMMIT command to mark the changes as permanent.

SYNTAX : COMMIT;

ROLLBACK command :

This command restores the database to last commited state.
If we have used the UPDATE command to make some changes into the database or inserted a record, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

SYNTAX : ROLLBACK;

It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction. The savepoints are like checkpoints, they temporarily save a transaction up to where the transaction can be rolled back

SYNTAX: ROLLBACK TO savepoint_name;

SAVEPOINT command:

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

SYNTAX: SAVEPOINT savepoint_name;

In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required

Example : Using Savepoint and Rollback:

Following is the table class,

id	name
1	Abhi
2	Adam
4	Alex

MySQL, set the autocommit option as shown below

SET AUTOCOMMIT=0;

Let us use some SQL queries on the above table and see the results.

- INSERT INTO class VALUES(5, 'Rahul');
- COMMIT;
- UPDATE class SET name = 'Abhijit' WHERE id = '5';
- SAVEPOINT A;
- INSERT INTO class VALUES(6, 'Chris');
- SAVEPOINT B;
- INSERT INTO class VALUES(7, 'Bravo');
- SAVEPOINT C
- SELECT * FROM class;

The resultant table will look like,

id	Name
1	Abhi
2	Adam

4	Alex
5	Abhijit
6	Chris
7	Bravo

Now let's use the ROLLBACK command to roll back the state of data to the savepoint B.

- ROLLBACK TO B;
- SELECT * FROM class;

The output now – current table would look like

id	Name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK the state of data to the savepoint A

- ROLLBACK TO A;
- SELECT * FROM class;

Now the table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

Queries on Sailor , Boats and Reserves Table:

Tables used in this note:

Sailors(sid: integer, sname: string, rating: integer, age: real);
Boats(bid: integer, bname: string, color: string);
Reserves(sid: integer, bid: integer, day: date).

Sailors

Sid	Sname	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	40
85	Art	3	25.5
95	Bob	3	63.5

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Reserves

sid	bid	day
22	101	1998-10-10
22	102	1998-10-10
22	103	1998-10-8
22	104	1998-10-7
31	102	1998-11-10
31	103	1998-11-6
31	104	1998-11-12
64	101	1998-9-5
64	102	1998-9-8
74	103	1998-9-8

Figure 1: Instances of Sailors, Boats and Reserves

1. Create the Tables:

```
CREATE TABLE sailors (      sid integer not null,
                            sname varchar(32),
                            rating integer,
                            age real,
                            CONSTRAINT PK_sailors PRIMARY KEY(sid) );

CREATE TABLE reserves (  sid integer not null,
                            bid integer not null,
                            day datetime not null,
                            CONSTRAINT PK_reserves PRIMARY KEY(sid, bid, day),
                            FOREIGN KEY(sid) REFERENCES  sailors(sid),
                            FOREIGN KEY(bid) REFERENCES boats(bid) );

CREATE TABLE boats(         sid integer not null,
                            bname varchar(32),
                            color  varchar(5),
                            rating integer,
                            age real,
                            CONSTRAINT PK_boats PRIMARY KEY(bid) );
```

DATABASE MANAGEMENT SYSTEMS
UNIT V – TOPIC 1
Triggers

A trigger is a procedure which is automatically invoked by the DBMS in response to changes to the database, and is specified by the database administrator (DBA).

A database with a set of associated triggers is generally called an active database.

In MySQL, a trigger is a stored database object that is automatically executed or fired on occurrence of a DML command (also satisfying a specified condition.)

Triggers are used to enforce business rules, validate input data, and maintain an audit trail.

Parts of a Trigger:

A Trigger description contains three parts:

Event-Condition-Action (ECA)

Event is a change to the database which activates the trigger -

Trigger Event - INSERT | UPDATE | DELETE

Trigger Activation Time – BEFORE | AFTER

Condition is a query that is run when the trigger is activated - SQL condition

Action is a procedure that is executed when a trigger is activated due to meeting the specified condition

All data actions performed by the trigger, execute within the same transaction in which the trigger fires.

Cannot contain transaction control statements (COMMIT, SAVEPOINT, ROLLBACK)

Uses of Trigger:

1. **Enforcing Business Rules :**Triggers can ensure that certain business rules are automatically enforced within the database. For example, a trigger can ensure that an employee's salary cannot be decreased. Passwords must be changed every month, insertion not allowed beyond working hours.
2. **Validating Data:** Triggers can be used to validate data before it is inserted or updated in the database. For example, ensuring that an email address has a valid format.
3. **Maintaining Audit Trails:** Triggers can be used to keep an audit trail of changes made to important data. This is useful for tracking who changed what data and when.
4. **Synchronizing Tables:** Triggers can be used to synchronize data between tables. For example, if you have a master table and several dependent tables, you can use triggers to update dependent tables whenever the master table is modified.
5. **Preventing Invalid Transactions**: Triggers can be used to prevent certain transactions that could lead to invalid or inconsistent data. For example, preventing deletion of records that are referenced by other records.

6. **Automatic Calculations :** Triggers can be used to perform automatic calculations and updates. For example, updating the total amount in an order whenever an order line item is added or updated.
7. **Enforcing Referential Integrity:** Triggers can enforce referential integrity rules, such as cascading updates or deletes, beyond what foreign keys alone can do.
8. **Data Transformation :** Triggers can be used to transform data before it is stored. For example, automatically converting all text to uppercase.

Types of Trigger

In MySQL, triggers are always defined at the row level, meaning they execute once for each row affected by the triggering event.

We can define the maximum six types of actions or events in the form of triggers:

Before Insert: It is activated before the insertion of data into the table.

After Insert: It is activated after the insertion of data into the table.

Before Update: It is activated before the update of data in the table.

After Update: It is activated after the update of the data in the table.

Before Delete: It is activated before the data is removed from the table.

After Delete: It is activated after the deletion of data from the table.

Trigger Syntax:

```
CREATE TRIGGER trigger_name
   (AFTER | BEFORE) (INSERT | UPDATE | DELETE)
       ON table_name FOR EACH ROW
       BEGIN
      --variable declarations
      --trigger code
      END;
```

NOTE: When a trigger code contains multiple statements, we can change the delimiter from the default ; to some other character.

Example-1:

Let us create a trigger that would make a log entry whenever a new record is inserted into the EMP table along with the timestamp.

1. CREATE A TABLE EMP_LOG as below

```
CREATE TABLE EMP_LOG (
LOG_ID INT AUTO_INCREMENT PRIMARY KEY,
EMP_ID INT, MSG VARCHAR(50),
CHANGE_TIME TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

2. Create the trigger as below

```
CREATE TRIGGER emp_insert_log
after INSERT ON emp
FOR EACH ROW
  INSERT INTO EMP_LOG (EMP_ID, MSG)
  VALUES (NEW.EMPNO, "ROW INSERTED AT");
```

3. TESTING: - run the following commands and check the output
    a. Insert record into emp table
    b. Select * from emp_log (check the new record inserted automatically)

Example-2:

Create a trigger for making the log entry whenever the salary is updated recording the empno, old and new salary

1. CREATE A TABLE SAL_UPDATE_LOG as below

```
CREATE TABLE SAL_UPDATE_LOG (
LOG_ID INT AUTO_INCREMENT PRIMARY KEY,
EMP_ID INT,
OLD_SALARY FLOAT(7,2), NEW_SALARY FLOAT(7,2),
CHANGE_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

2. CREATE THE TRIGGER

```
CREATE TRIGGE emp_log
after UPDATE ON emp
FOR EACH ROW
  INSERT INTO SAL_UPDATE_LOG (EMP_ID, OLD_SALARY, NEW_SALARY)
  VALUES (NEW.EMPNO, OLD.SAL, NEW.SAL);
```

3. TESTING:
    a. Update salary for an employee in emp table
    b. Select * from sal_update_log

Example – 3:

Create a trigger to be executed before insert into emp which ensures the minimum salary of the new employees is 30000

```
DELIMITER //
Create Trigger before_insert_empsalary
BEFORE INSERT ON emp FOR EACH ROW
BEGIN
IF NEW.sal < 30000 THEN SET NEW.sal = 30000;
END IF;
```

```
END //
DELIMITER ;
```

TESTING:

Insert a new record into emp table with salary less than 30000
Select * from emp
Ensure that the salary is set to 30000 in the newly entered row.

Triggers that raise the error:

In MySQL, you can create a trigger that will raise an error when a certain condition is met by using the SIGNAL SQL statement. The SIGNAL statement allows you to set an error condition, which effectively raises an error.

SIGNAL SQLSTATE '45000': Raises an error with a custom SQLSTATE value of '45000', which is a generic state indicating an error condition.

SET MESSAGE_TEXT = ‘xxxxxxxxx': Sets the error message that will be returned.

Example:

Let us assume we should not allow insertion into emp table beyond office hours [9am-6pm]

```
DELIMITER //

CREATE TRIGGER control_access
BEFORE INSERT ON emp
FOR EACH ROW
BEGIN
   DECLARE HR INT;
   SET HR = HOUR(CURTIME());
   IF HR < 9 AND HR >18 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'INSERTION IS PROHIBITED AT THIS TIME';
   END IF;
END //

DELIMITER ;
```

Listing the available Triggers:

List all the available triggers on a database

SHOW TRIGGERS;

```
----------------+
| Trigger                | Event  | Table | Statement                                                     | Timing |
Created                                                                                            |      sql_mode
| Definer        | character_set_client | collation_connection | Database Collation |
--+--------------------+
| before_insert_empsalary | INSERT | emp   | BEGIN
IF NEW.sal < 30000 THEN SET NEW.sal = 30000;
END IF;
END          |          BEFORE          |          2024-07-11          14:45:16.49          |
ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZER
O_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION            |
root@localhost | cp850              | cp850_general_ci     | utf8mb4_0900_ai_ci |
+------------------------+--------+-------+--
```

Deleting a Trigger:

We can drop/delete/remove a trigger in MySQL using the DROP TRIGGER statement.

DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name;

Example:

DROP TRIGGER before_insert_empsalary;

DATABASE MANAGEMENT SYSTEMS
UNIT V – TOPIC 2
Stored Procedures

A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database.

It is a subroutine or a subprogram in the regular computing language. A procedure always contains a name, parameter lists, and SQL statements.

We can invoke the procedures by using triggers, other procedures and applications such as Java, Python, PHP, etc.

Stored Procedure Features

- Stored Procedure increases the performance of the applications. Once stored procedures are created, they are compiled and stored in the database.
- Stored procedure reduces the traffic between application and database server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
- Stored procedures are reusable and transparent to any applications.
- A procedure is always secure. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.

Syntax :

The following syntax is used for creating a stored procedure in MySQL. It can return one or more value through parameters or sometimes may not return at all. By default, a procedure is associated with our current database. But we can also create it into another database from the current database by specifying the name as database_name.procedure_name.

```
DELIMITER //
CREATE PROCEDURE procedure_name [ [IN  | OUT  | INOUT]
parameter_name datatype [, parameter_name datatype]) ]
BEGIN
   Declaration_section
   Executable_section
END //
DELIMITER ;
```

Parameter Name	Descriptions
procedure_name	It represents the name of the stored procedure.
Parameter	presents the number of parameters. It can be one or more than one.
Declaration_section	It represents the declarations of all variables.
Executable_section	It represents the code for the function execution.

Different types of parameters used:

1. IN parameter

It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

2. OUT parameters

It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

3. INOUT parameters

It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

Calling the Procedure:

We can use the CALL statement to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT). The following syntax is used to call the stored procedure in MySQL:

CALL procedure_name (parameter(s))

Examples:

1. Procedures without Parameters

Suppose we want to display the list and count of employees who are earning commission.

```
DELIMITER //
CREATE PROCEDURE comm_details()
BEGIN
   SELECT * FROM emp WHERE comm  IS NOT NULL;
   SELECT COUNT(comm) AS 'Commission Earners' FROM emp;
END //
DELIMITER ;
```

Calling the procedure :

```
> CALL comm_details();
```

2. Procedures with IN Parameters

Suppose we want to display the list of employees who are from a specified department.

```
DELIMITER //
CREATE PROCEDURE dept_employees (IN dno int)
BEGIN
        SELECT * from EMP where DEPTNO=dno;
END //
DELIMITER ;
```

Calling the procedure :

```
> CALL dept_employees(20);
```

3. Procedures with OUT Parameters

Suppose we want to display the highest salary paid.

```
DELIMITER //
CREATE PROCEDURE display_max_sal(OUT highestsal INT)
BEGIN
   SELECT MAX(sal) INTO highestsal FROM emp;
END //
DELIMITER ;
```

NOTE: When we call the procedure, the OUT parameter tells the database systems that its value goes out from the procedures. Now, we will pass its value to a session variable @M in the CALL statement as follows:

Calling the procedure :

```
> CALL display_max_sal (@X);
> SELECT @X;
```

4. Procedures with IN & OUT Parameters

Suppose we want to display the highest salary paid in the specified department

```
DELIMITER //
CREATE PROCEDURE display_deptmax_sal(OUT highestsal INT, IN dno INT)
BEGIN
   SELECT MAX(sal) INTO highestsal FROM emp WHERE deptno=dno;
END //
DELIMITER ;
```

Calling the procedure :

```
> CALL display_deptmax_sal (@M, 10);
```

6. Procedures with IN & OUT Parameters

Suppose we want to display the salary earned by a selected employee

```
DELIMITER &&
CREATE PROCEDURE get_emp_sal (INOUT data INT)
BEGIN
   SELECT sal INTO data FROM emp WHERE empno = data;
END &&
DELIMITER ;
```

Calling the procedure :

```
> SET @M = 7900
> CALL get_emp_sal(@M);
> SELECT @M;
```

Displaying the list of All procedures

When we have several procedures in the MySQL server, we can list all procedure stored on the current MySQL server as follows:

SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]

Example:

SHOW PROCEDURE STATUS WHERE DB='IT2B';

Deleting Stored Procedures:

MySQL also allows a command to drop the procedure. When the procedure is dropped, it is removed from the database server also. The following statement is used to drop a stored procedure in MySQL:

DROP PROCEDURE [IF EXISTS] procedure_name;

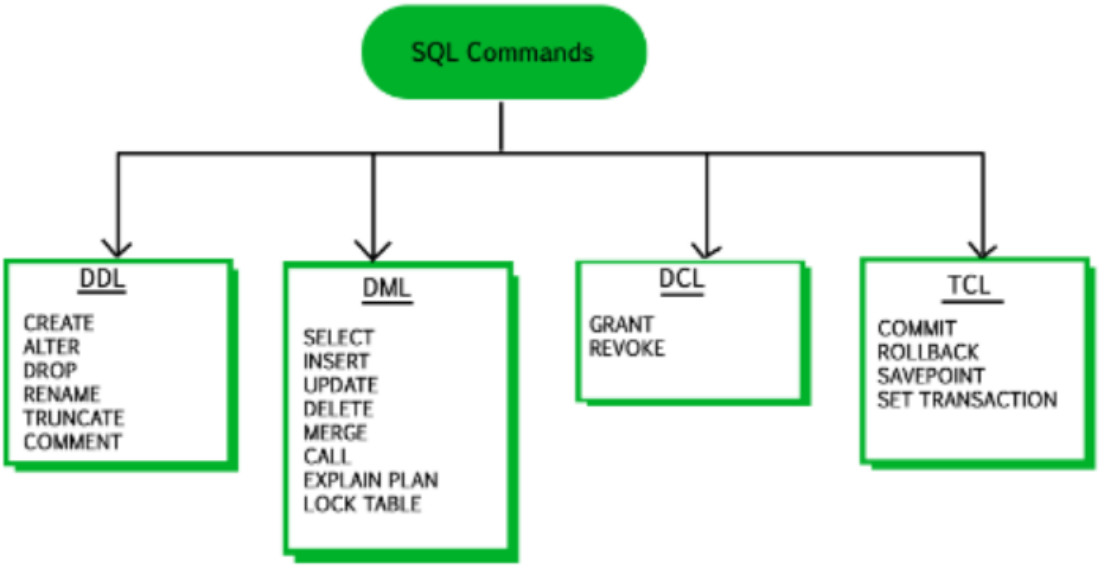
Example:

DROP PROCEDURE comm_details;

DATABASE MANAGEMENT SYSTEMS
UNIT V – TOPIC 3
Data Control Language

Data Controlling Language:

Data Controlling Language (DCL) helps users to retrieve and modify the data stored in the database with some specified queries. Grant and Revoke belong to these types of commands of the Data controlling Language. DCL is a component of SQL commands.



Two types of DCL commands can be used by the user in SQL. These commands are useful, especially when several users access the database. It enables the administrator to manage access control. The two types of DCL commands are as follows:

- GRANT
- REVOKE

Grant Privileges on Table

This command allows the administrator to assign particular privileges or permissions over a database object, such as a table, view, or procedure.

It enables system administrators to assign privileges and roles to the MySQL user accounts so that they can use the assigned permission on the database whenever required.

Syntax

```
GRANT privilege_name(s)
ON object
TO user_account_name;
```

Parameter	Description
privilege_name(s)	**It specifies the access rights or grant privilege to user accounts. If we want to give multiple privileges, then use a comma operator to separate them.**
object	**It determines the privilege level on which the access rights are being granted. It means granting privilege to the table; then the object should be the name of the table.**
user_account_name	**It determines the account name of the user to whom the access rights would be granted.**

The privileges to assign are listed below. It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
ALL	ALL does not revoke all permissions for the table. Rather, it revokes the ANSI-92 permissions which are SELECT, INSERT, UPDATE, DELETE, and REFERENCES.

Object

The name of the database objects that you are granting permissions for. In the case of granting privileges on a table, this would be the table name.

User_account_name

The name of the user who will be granted these privileges.

Example

Let's look at some examples of how to grant privileges on tables in SQL Server.

For example, if you wanted to grant SELECT, INSERT, UPDATE, and DELETE privileges on a table called employees to a user name smithj, you would run the following GRANT statement:

GRANT SELECT,INSERT,UPDATE, DELETE ON employees TO smithj;

You can also use the ALL keyword to indicate that you wish to grant the ANSI-92 permissions

(ie: SELECT, INSERT, UPDATE, DELETE, and REFERENCES) to a user named smithj.

```
GRANT ALL ON employees TO smithj;
```

If you wanted to grant only SELECT access on the employees table to all users, you could grant the privileges to the public role.

```
GRANT SELECT ON employees TO public;
```

Revoke Privileges on Table:

As the name suggests, revoke is to take away.

The REVOKE command enables the database administrator to remove the previously provided privileges or permissions from a user over a database or database object, such as a table, view, or procedure.

The REVOKE commands prevent the user from accessing or performing a specific operation on an element in the database.

In simple language, the REVOKE command terminates the ability of the user to perform the mentioned SQL command in the REVOKE query on the database or its component.

The primary reason for implementing the REVOKE query in the database is to ensure the data's security and integrity.

Syntax

The syntax for revoking privileges on a table in SQL Server is:

```
REVOKE privileges
ON object
FROM user_account_name;
```

Example

Let's look at some examples of how to revoke privileges on tables in SQL Server.

For example, if you wanted to revoke DELETE privileges on a table called employees from a user named anderson, you would run the following REVOKE statement:

```
REVOKE DELETE ON employees FROM anderson;
```

If you wanted to revoke ALL ANSI-92 permissions

(ie: SELECT, INSERT, UPDATE, DELETE, and REFERENCES) on a table for a user named anderson, you could use the ALL keyword as follows:

```
REVOKE ALL ON employees FROM anderson;
```

If you had granted SELECT privileges to the public role (ie: all users) on the employees table and you wanted to revoke these privileges, you could run the following REVOKE statement:

```
REVOKE SELECT ON employees FROM public;
```

Differences between Grant and Revoke commands:

S.NO	Grant	Revoke
1	This DCL command grants permissions to the user on the database objects.	This DCL command removes permissions if any granted to the users on database objects.
2	It assigns access rights to users.	It revokes the user access rights of users.
3	For each user you need to specify the permissions.	If access for one user is removed; all the particular permissions provided by that users to others will be removed.
4	When the access is decentralized granting permissions will be easy.	If decentralized access removing the granted permissions is difficult.

Advantages of DCL commands:

- **Security:** the primary reason to implement DCL commands in the database is to manage the access to the database and its object between different users. It ensures the security and integrity of the data stored in the database.
- **Granular control:** DCL commands provide granular control to the data administrator over the database. Thus, it enables the admin to create different levels of access to the database.
- **Flexibility:** The data administrator can implement DCL commands on specific commands and queries in the database. It allows the administrator to grant or revoke user permissions and privileges as per their needs. It provides flexibility to the administrator that allows them to manage access to the database.

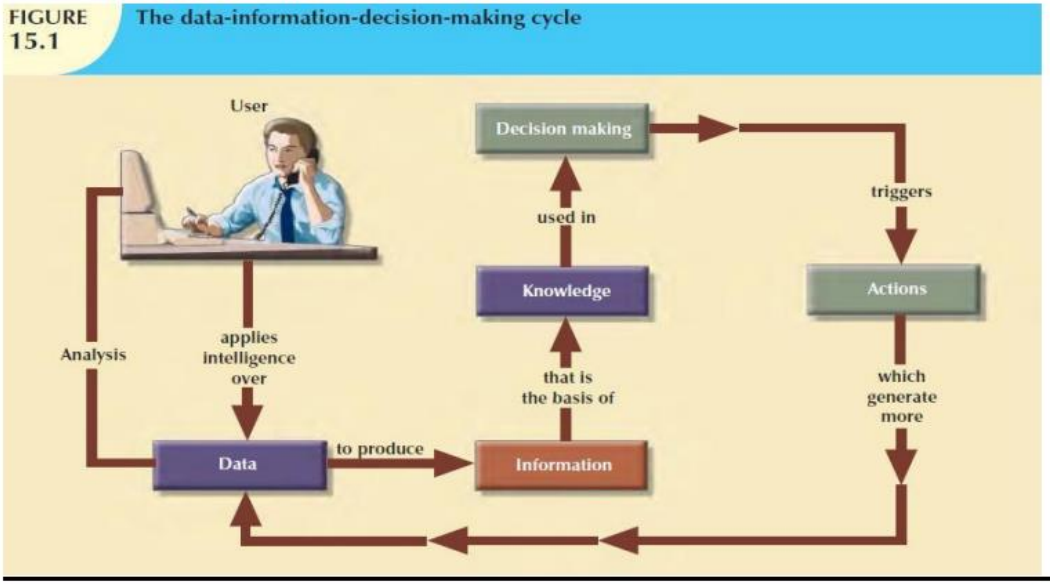
Disadvantages of DCL commands:

- **Complexity:** It increases the complexity of database management. If many users are accessing the database, keeping track of permission and privileges provided to every user in the database becomes very complex.
- **Time-Consuming:** It is time-consuming to assign the permissions and privileges to each user separately.
- **Risk of human error:** Human administrators execute DCL commands and can make mistakes in granting or revoking privileges. Thus, giving unauthorized access to data or imposing unintended restrictions on access.
- **Lack of audit trail:** There may be no built-in mechanism to track changes to privileges and permissions over time. Thus, it is extremely difficult to determine who has access to the data and when that access was granted or revoked.

DATABASE MANAGEMENT SYSTEMS
UNIT V – TOPIC 4
Data Base Administrator

Data as a Corporate Asset:

Data are a valuable resource that can translate into information. If the information is accurate and timely, it is likely to trigger actions that enhance the company's competitive position and generate wealth. In effect, an organization is subject to a data-information-decision cycle; that is, the data user applies intelligence to data to produce information that is the basis of knowledge used in decision making by the user. This cycle is illustrated in Figure below.



The decisions made by high-level managers trigger actions within the organization's lower levels. Such actions produce additional data to be used for monitoring company performance. Thus, data form the basis for decision making, strategic planning, control, and operations monitoring. To manage data as a corporate asset, managers must understand the value of information—that is, processed data.

Role of a Database in an Organization:

Data are used by different people in different departments for different reasons. Therefore, data management must address the concept of shared data. The DBMS facilitates:

- Interpretation and presentation of data in useful formats
- Distribution of data and information to the right people at the right time.
- Data preservation and monitoring the data usage for adequate periods of time.
- Control over data duplication and use, both internally and externally.
- Data Security and Concurrent access.

The database's predominant role is to support managerial decision making at all levels in the organization while preserving data privacy and security.

DataBase Administration Function:

The person responsible for the control of the centralized and shared database became known as the **DataBase Administrator (DBA).**

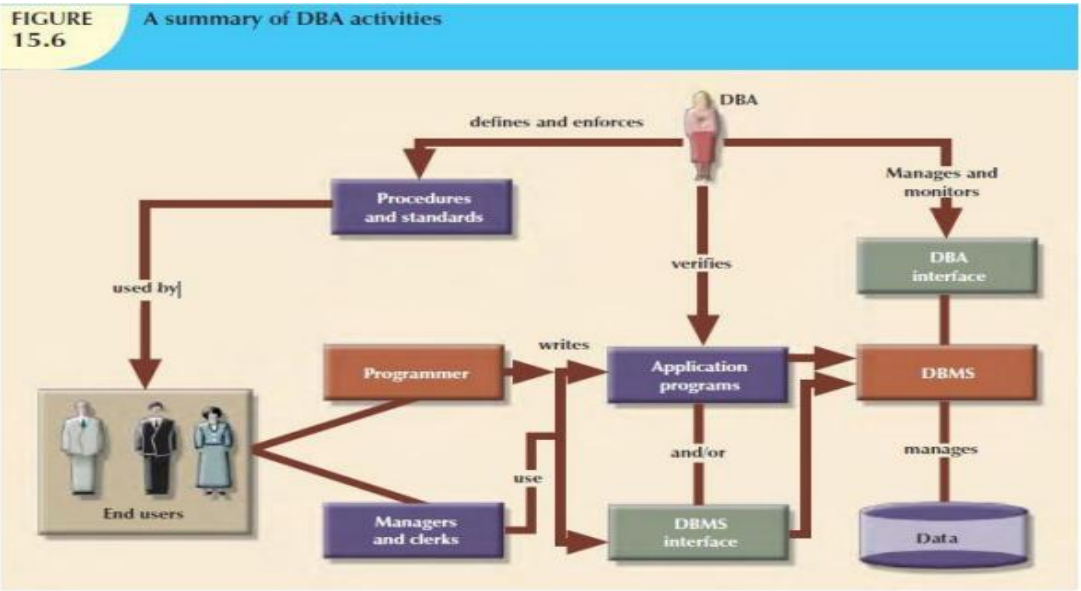
It is common practice to define the DBA function by dividing the DBA operations according to the Database Life Cycle (DBLC) phases. If that approach is used, the DBA function requires personnel to cover the following activities:

- Database planning, including the definition of standards, procedures, and enforcement.
- Database requirements gathering and conceptual design.
- Database logical and transaction design.
- Database physical design and implementation.
- Database testing and debugging.
- Database operations and maintenance, including installation, conversion, and migration.
- Database training and support.
- Data quality monitoring and management.

Figure below represents an appropriate DBA functional organization according to that model.

The interactions between the people and data in an organization, places the DBA in the dynamic environment as explored in the Figure below. The DBA is the focal point for data/user interaction. The DBA defines and enforces the procedures and standards to be used by programmers and end users during their work with the DBMS.

The DBA also verifies that programmer and end-user access meets the required quality and security standards.



DBA skills are be divided into two categories **Managerial Role**
Technical Role

The DBA's Managerial Role

As a manager, the DBA must concentrate on the control and planning dimensions of database administration. Therefore, the DBA is responsible for:

- Coordinating, monitoring, and allocating database administration resources: i.e. people and data.
- Defining goals and formulating strategic plans for the database administration function.

1. End-User Support

The DBA interacts with the end user by providing data and information support services to the organization's departments.

- Gathering user requirements.
- Building end-user confidence.
- Resolving conflicts and problems.
- Finding solutions to information needs.
- Ensuring quality and integrity of data and applications.
- Managing the training and support of DBMS users.

2. Policies, Procedures and Standards

A prime component of a successful data administration strategy is the continuous enforcement of the policies,procedures, and standards for correct data creation, usage, distribution, and deletion within the database.

- **Policies** are general statements of direction or action that communicate and support DBA goals.

- **Standards** describe the minimum requirements of a given DBA activity; they are more detailed and specificthan policies.

- **Procedures** are written instructions that describe a series of steps to be followed duringthe performance ofa given activity.

3. Data Security, Privacy and Integrity

The security, privacy, and integrity of the data in the database are of great concern to DBAs who manage current DBMS installations. The distribution of data across multiple sites, has made data maintenance, security, and integrity very critical. The DBAs must team up with Internet security experts to build security mechanisms to safeguard data from possible attacks or unauthorized access.

4. Data Backup and Recovery

The DBA must also ensure that the data in the database can be fully recovered in case of physical data loss or loss of database integrity. Data loss can be partial or total. A partial loss is caused by a physical loss of part of the database or when part of the database has lost integrity. A total loss might mean that the database continues to exist but its integrity is entirely lost or that the entire database is physically lost.

Disaster management includes all of the DBA activities designed to secure data availability following a physical disaster or a database integrity failure.

The DBA's Technical Role

The DBA's technical activities include the selection, installation, operation, maintenance, and upgrading of the DBMS and utility software, as well as the design, development, implementation, and maintenance of the application programs that interact with the database.

The technical aspects of the DBA's job are rooted in the following areas of operation:

1. **Evaluating, selecting, and installing the DBMS and related utilities**.
   Selecting the database management system, utility software, and supporting hardware to be used in the organization. The selection plan is based on organization's needs and the features like DBMS model, storage capacity, backup-recovery, concurrency control, performance, portability, cost etc.

2. **Designing and implementing databases and applications.**

   The DBA function usually requires that several people be dedicated to database modeling and design activities. The DBA also works with applications programmers to ensure the quality and integrity of

database design and transactions. Such support services include reviewing the database application design to ensure that transactions are correct, efficient and compliant.

3. Testing and evaluating databases and applications.

The DBA must also provide testing and evaluation services for all of the database and enduser applications. Testing starts with the loading of the tested database. That database contains test data for the applications, and its purpose is to check the data definition and integrity rules of the database and application programs. The testing and evaluation of a database application cover all aspects of the system technical, evaluation of written documentation, observance of standards for naming, documenting, and coding, Data duplication conflicts with existing data, the enforcement of all data validation rules.

4. Operating the DBMS, utilities, and applications.

DBMS operations can be divided into four main areas:

- System support.
- Performance monitoring and tuning.
- Backup and recovery.
- Security auditing and monitoring.

5. Training and supporting users.

Training people to use the DBMS and its tools is included in the DBA's technical activities. The DBA provides technical training in the use of the DBMS and its utilities for the applications programmers.

6. Maintaining the DBMS, utilities, and applications.

Maintenance activities are dedicated to the preservation of the DBMS environment. Periodic DBMS maintenance includes management of the physical or secondary storage devices, upgrading the DBMS and utility software, migration and conversion services for data in incompatible formats or for different DBMS software.

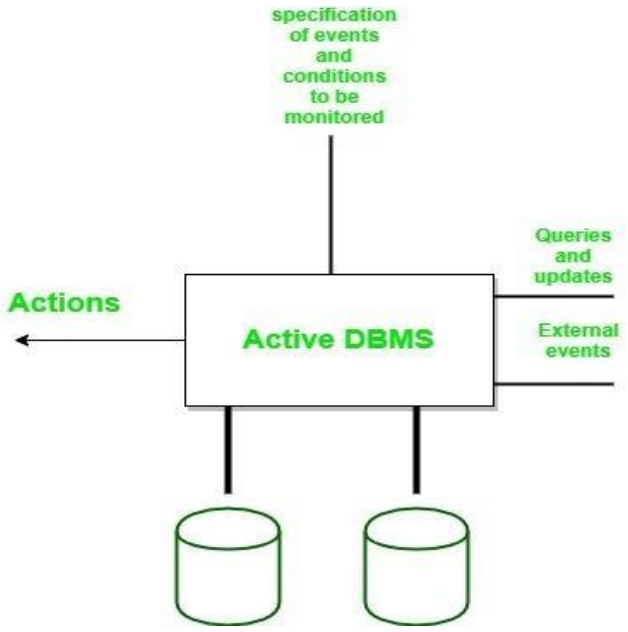
DATABASE MANAGEMENT SYSTEMS
UNIT V – TOPIC 5
Miscellaneous Topics

Active Databases:

Active Database is a database consisting of set of triggers. These databases are very difficult to be maintained because of the complexity that arises in understanding the effect of these triggers. In such database, DBMS initially verifies whether the particular trigger specified in the statement that modifies the database is activated or not, prior to executing the statement.

If the trigger is active then DBMS executes the condition part and then executes the action part only if the specified condition is evaluated to true. It is possible to activate more than one trigger within a single statement.

In such situation, DBMS processes each of the trigger randomly. The execution of an action part of a trigger may either activate other triggers or the same trigger that Initialized this action. Such types of trigger that activates itself is called as 'recursive trigger'. The DBMS executes such chains of trigger in some pre-defined manner but it affects the concept of understanding.



Features of Active Database:

1. It possess all the concepts of a conventional database i.e. data modelling facilities, query language etc.

2. It supports all the functions of a traditional database like data definition, data manipulation, storage management etc.
3. It supports definition and management of ECA (Event–Condition–Action) (ECA) rules.
4. It detects event occurrence.
5. It must be able to evaluate conditions and to execute actions.
6. It means that it has to implement rule execution.

Advantages:

- Enhances traditional database functionalities with powerful rule processing capabilities.
- Enable a uniform and centralized description of the business rules relevant to the information system.
- Avoids redundancy of checking and repair operations.
- Suitable platform for building large and efficient knowledge base and expert systems.

MySQL Create User :

The MySQL user is a record in the USER table of the MySQL server that contains the login information, account privileges, and the host information for MySQL account. It is essential to create a user in MySQL for accessing and managing the databases.

The MySQL Create User statement allows us to create a new user account in the database server.

When the MySQL server installation completes, it has a ROOT user account only to access and manage the databases.

We can create non-root users using the following syntax:

CREATE USER [IF NOT EXISTS] username@hostname IDENTIFIED BY 'password';

Example:

```
mysql> create user savy@localhost identified by 'savram123';
```

When you create a user that already exists, it gives an error. But if you use, IF NOT EXISTS clause, the statement gives a warning for each named user that already exists instead of an error message.

```
mysql> CREATE USER IF NOT EXISTS ram@localhost IDENTIFIED BY 'esha123';
```

Grant Privileges to the MySQL New User

MySQL server provides multiple types of privileges to a new user account. Some of the most commonly used privileges are given below:

ALL PRIVILEGES: It permits all privileges to a new user account.

CREATE: It enables the user account to create databases and tables.

DROP: It enables the user account to drop databases and tables.

DELETE: It enables the user account to delete rows from a specific table.

INSERT: It enables the user account to insert rows into a specific table.

SELECT: It enables the user account to read a database.

UPDATE: It enables the user account to update table rows.

Examples:

mysql> GRANT ALL PRIVILEGES ON * . * TO savy@localhost;

mysql> GRANT CREATE, SELECT, INSERT ON * . * TO ram@localhost;

Note: *.* indicates all databases and all tables ---- db.* - means all tables on the database db

To see the existing privileges for the user, execute the following command.

mysql> SHOW GRANTS for username;

Drop User:

The MySQL Drop User statement allows to remove one or more user accounts and their privileges from the database server. If the account does not exist in the database server, it gives an error.

DROP USER 'account_name';

Example: **DROP** USER ram@localhost;