SCNR Government Degree College, Proddatur

Department of Computer Science-II Year-IV Sem

UNIT - I: Overview of Database Management System

UNIT- I Overview of Database Management System: Introduction to data, information, database, database management systems, file-based system, Drawbacks of file-Based System, database approach, Classification of Database Management Systems, advantages of database approach, Various Data Models, Components of Database Management System, three schema architecture of data base, costs and risks of database approach.

Data: Data refers to raw facts and figures that do not have any meaning by themselves.
Examples:

- Student marks: 85, 90, 78
- Names: Sirisha, Ravi, Sita
- Subjects:Mathematics, Physics, Chemistry.

Information: Information is processed data that is meaningful and useful for decision-making.
Example: "Siresha scored 85 marks in Mathematics."

Difference Between Data and Information

Data	Information
Data is unorganized and unrefined facts	Information comprises processed, organized data presented in a meaningful context
Data is an individual unit that contains raw materials which do not carry any specific meaning.	Information is a group of data that collectively carries a logical meaning.
Data doesn't depend on information.	Information depends on data.
It is measured in bits and bytes.	Information is measured in meaningful units like time, quantity, etc.
Raw data alone is insufficient for decision making	Information is sufficient for decision making

Examples: • Student marks: 85, 90, 78 • Names: Sirisha, Ravi, Sita • Subjects:Mathematics, Physics, Chemistry.	Example: “Siresha scored 85 marks in Mathematics.”

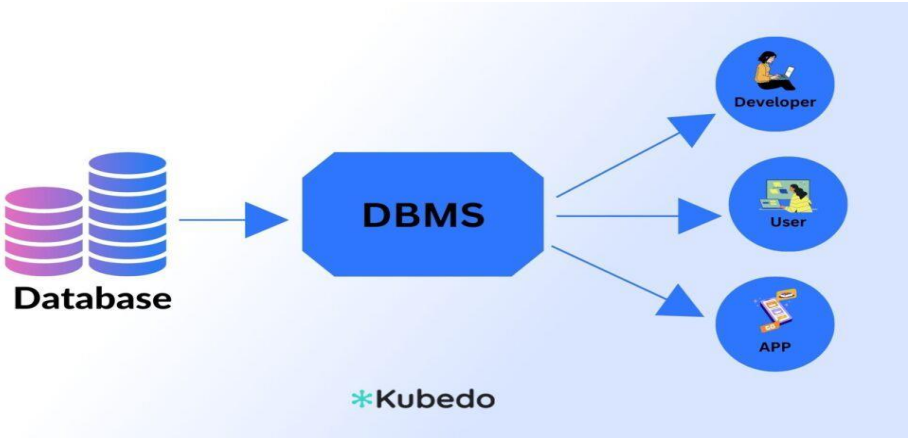
Database

A **database** is an organized collection of related data stored electronically so that it can be easily accessed, managed, and updated.
Example:

- A student database containing roll number, name, marks, and address.
- The college Database organizes the data about the admin, staff, students and faculty etc.

Database Management System (DBMS)

A **DBMS** is software that allows users to create, store, retrieve, update, and manage databases efficiently.
Examples: MySQL, Oracle, MS Access, SQL Server.



File-Based System

A **file-based system** is a traditional approach where data is stored in separate files and managed using application programs.
Example:

- Student details stored in different text files or spreadsheets.

Drawbacks of File-Based System

The major disadvantages are:

1. **Data Redundancy** – Same data is stored in multiple files
2. **Data Inconsistency** – Different files may have different values for the same data
3. **Data Isolation** – Data is scattered across different files
4. **Security Problems** – Difficult to control access
5. **Limited Data Sharing** – Data cannot be easily shared among users
6. **Data Integrity Issues** – Hard to enforce rules on data

Database Approach

The **database approach** is a method of managing data in which data is stored in a centralized database instead of separate files. It focuses on integration and sharing of data among multiple users and applications.

Example: A University Database stores:

- Student info
- Course info
- Fees
- Exam results

All departments access the **same shared database** instead of maintaining separate files.

Advantages of Database Approach

1. **Reduced Data Redundancy**
    - Data is stored centrally, so duplicate copies of the same data are minimized.
2. **Data Consistency & Integrity**
    - Because redundancy is reduced, data remains accurate and consistent across applications.
3. **Improved Data Sharing**
    - Multiple users and applications can access the database simultaneously in a controlled manner.
4. **Better Data Security**
    - DBMS provides authentication, authorization, and access controls to protect data.
5. **Centralized Data Management**
    - Data is managed and controlled centrally, which improves monitoring and maintenance.
6. **Support for Concurrent Access**
    - Many users can access and modify data at the same time without conflicts using concurrency control.
7. **Backup & Recovery**
    - DBMS provides automatic backup and recovery to prevent data loss.
8. **Standardization of Data**
    - Ensures uniform data formats and rules across the organization.
9. **Enhanced Decision-Making**
    - Centralized and accurate data supports faster and better decision-making.

- **Data Independence:**Changes to data structure do not affect application programs.

Components of a Database Management System (DBMS)

A DBMS consists of several key components that work together to store, manage, and retrieve data efficiently:

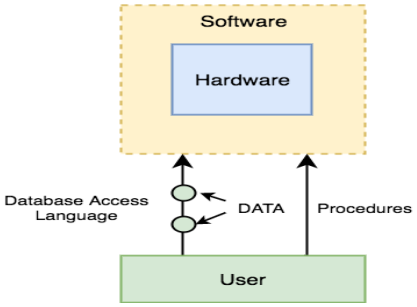


Fig: Components of a Database Management System (DBMS)

1. Hardware: Physical devices where the database runs (servers, computers, storage devices).

2. Software:

- Actual DBMS software such as Oracle, MySQL, SQL Server, PostgreSQL, etc.
- Includes operating system and application programs.

3. Data:

- The actual data stored in the database.
- Includes metadata (data about data).

4. Users:

- **Database Administrators (DBAs)** – manage the DB.
- **End users** – access data using applications.
- **Application programmers** – write DB programs.
- **System analysts** – design DB structure.

5. Procedures: Instructions and rules for database usage, backup, recovery, etc.

6. Query Processor: Converts user queries (e.g., SQL) into efficient low-level instructions.

7. Database Engine:

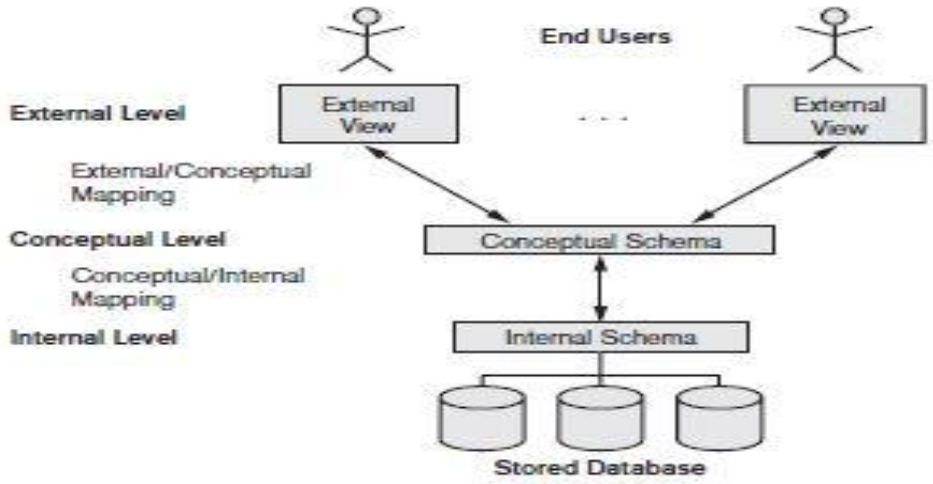
- Core component responsible for:
  - Storage management
  - Concurrency control
  - Transaction management
  - Recovery management

Three-Schema Architecture of Database

The **Three-Schema Architecture**, also called the **ANSI-SPARC Architecture**, is a framework used in DBMS to separate the database into three levels..

Advantages of Three-Schema Architecture:

- Data abstraction (hides complexity)
- Data independence (logical & physical)
- Enhanced security
- Different views for different users



Three-Schema Architecture of Database

It has **three layers**:

1. Internal Schema (Physical Level)

- This is the **lowest level** of architecture.
- It describes **how data is physically stored** in memory (files, indexes, storage structure).
- It Closest to hardware.
- Deals with compression, encryption, storage paths

Examples:

- File organization (heap, sequential)
- Indexing (B-tree, hashing)
- Data blocks and pages

2. Conceptual Schema (Logical Level)

- This is the **middle level**.
- It describes the **overall logical view** of the entire database.

- Hides physical details from users and Maintained by DBA

Examples:

- Entity-relationship model
- Tables and relationships
- Constraints
- Data types

3. External Schema (View Level)

- This is the **highest level**.
- It describes how **end-users view data**.
- Different users can have different views for security .
- Hides unnecessary data
- Used for reports, forms, queries .

Examples:

- Student may see: Name, Marks
- Accountant may see: Fees, Payments
- Faculty may see: Attendance, Grades

Costs and Risks of Database Approach

While DBMS offers many advantages, it also has certain costs and risks:

Costs of Database Approach

1. **High Hardware and Software Cost**
    - Licensing and maintenance expenses.
2. **Need for Skilled Personnel**
    - DBAs and trained staff increase cost.
3. **Implementation and Conversion Cost**
    - Migrating from file-based systems can be expensive.
4. **Complexity**
    - DBMS is more complex than traditional file systems.

Risks of Database Approach

1. **Database Failure**
    - Centralized nature means failure can halt whole organization.

2. **Security Threats**
    - Unauthorized access, hacking risks.
3. **Data Integrity Problems**
    - Design or user errors can corrupt data.
4. **Multiple Users Conflict**
    - Concurrency control issues.
5. **Dependency on Vendor**
    - Upgrades and platform dependency risks.

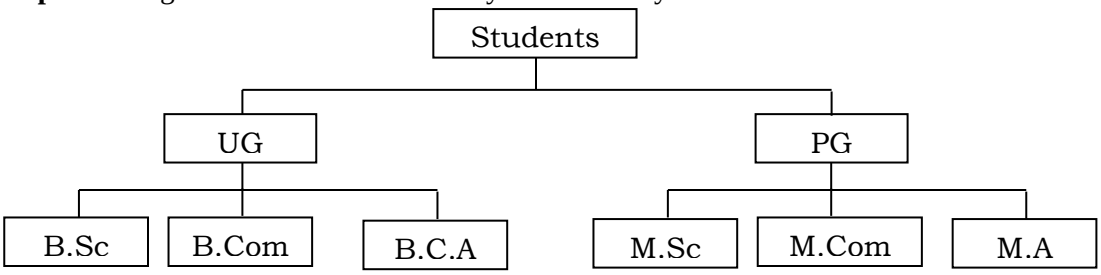
Various Data Models

Data Models defines how data will be stored, accessed, and updated in a database management .

1.Hierarchical Data Model:

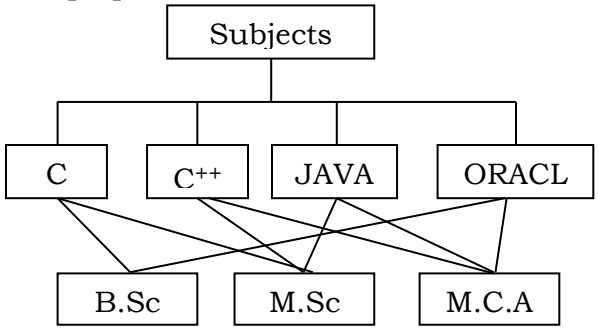
- Represents data in a **tree-like structure**.
- Supports **parent–child (one-to-many)** relationships.
- Each child has **only one parent**.

Real Example: 1.Organization chart 2.File system directory structure



2.Network Data Model:

- Extends hierarchical model.
- Supports **many-to-many** relationships.
- A child can have **multiple parents**.

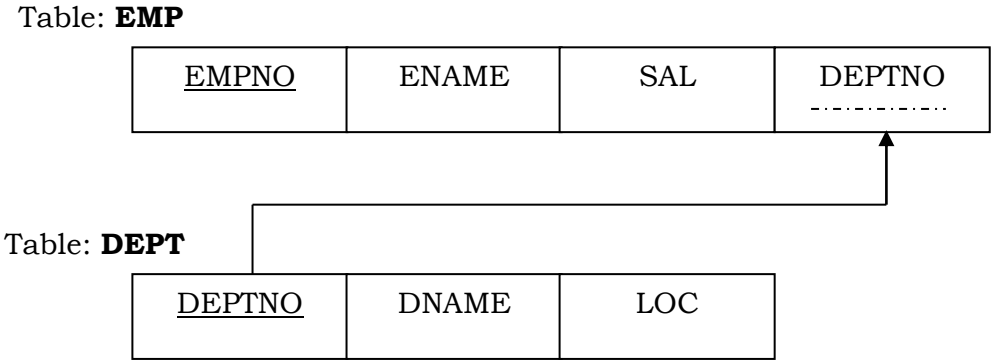


Real Example: 1. Telecom network systems 2.Airline reservation systems

3.Relational Data Model:

- Data stored in **tables (relations)**.
- Each table consists of rows and columns.
- Key constraints maintain relationships.

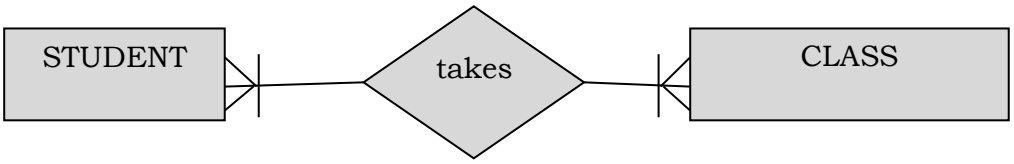
Real Example: 1.Banking systems 2.Student database



4.Entity-Relationship (ER) Model

- Used for conceptual design of database.
- Represents entities, relationships, and attributes.

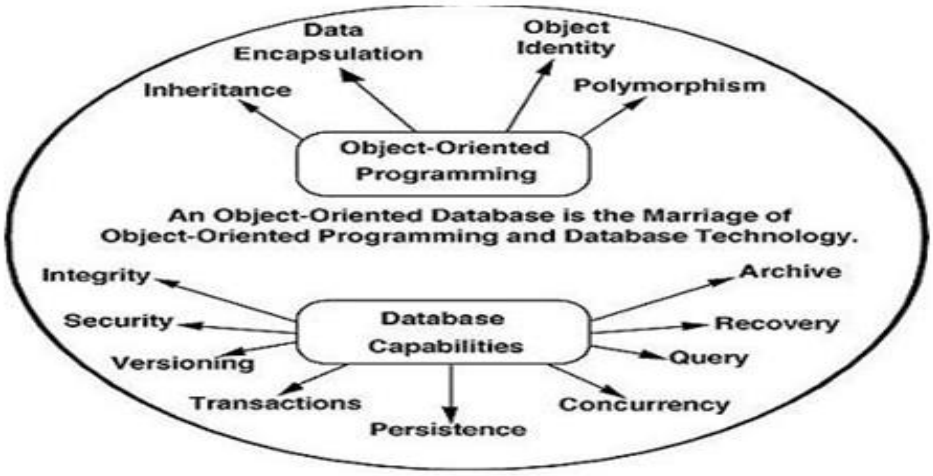
Real Example: Database design for schools, hospitals.



5.Object-Oriented Data Model

- Data stored as **objects**.
- Supports inheritance, encapsulation, polymorphism.

Real Example: 1.CAD software 2.Multimedia & graphics DB

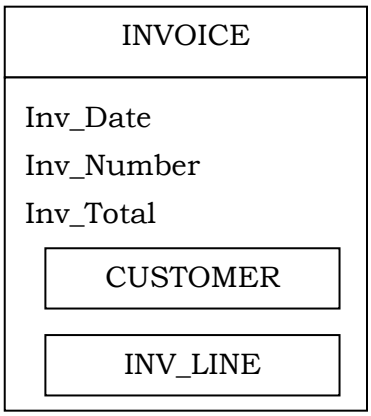


6.Object-Relational Data Model

- Hybrid of relational + object-oriented models.
- Adds object concepts to relational systems.

Real Example: PostgreSQL, Oracle

- In this model, an object is represented in a box, all the object attributes and relationships are included in that box.



Classification of Database Management System

A Database Management System can support different types of databases. The Databases are classified based on

I. The no.of Users
II. The Database Locations
III. The Use of Database
IV. The Data Model

I. The no.of Users:

The no.of users determines, whether the database is ***Single-User Database*** or ***Multi-User Database***.

1. **Single-User Database:** A Single-User Database supports only user at a time. In other words, if the user "A" is using the database, users "B" and "C" must wait until "A" is done.
2. **Multi-User Database:** A Multi-User Database supports multiple users at the same time. The Multi-User Database is divided into two types. They are
   a) **Work Group Database:** A Multi-User Database supports small number of users (<=50) in a specific department within an organization is called "Work Group Database".
   b) **Enterprise Database:** The Database is used by entire organization and supports many users (>50) in many departments is called an "Enterprise Database".

II. The Database Locations:

The Database locations determines, whether the database is ***Centralized Database*** or ***Distributed Database***.

1. **Centralized Database:** The Database that supports, data is located at a single site is called "Centralized Database".
2. **Distributed Database:** The Database that supports, data is located at different sites is called "Distributed Database".

III. The Use of Database:

Today Databases are classified based on the Use of the Database. Those are ***Operational Database*** and ***Data Ware House***.

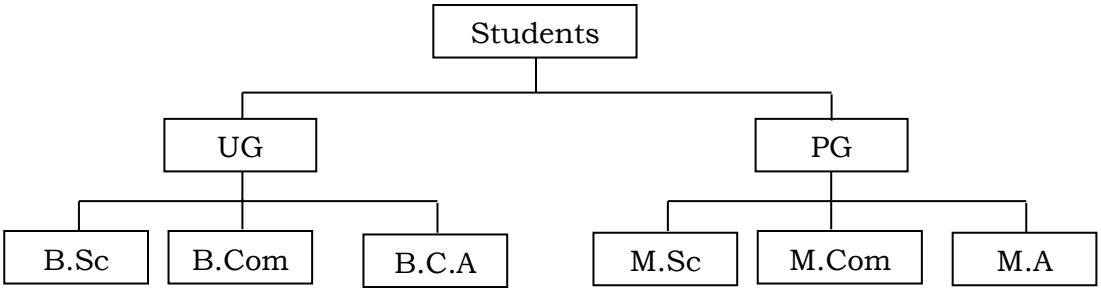
1. **Operational Database:** A Database that is used to support company day to day operations is called as **Operational Database**. It is also called **Transactional Database**.
2. **Data ware House:** A **Data Ware House** is a collection of Databases, which is used for collecting and storing the data in pecific organization. The Data ware House contains historical data.

IV.Based on Data Model: Data Models defines how data will be stored, accessed, and updated in a database management .

1.Hierarchical Data Model:

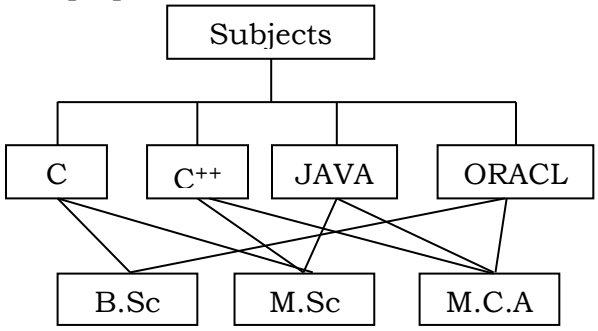
- Represents data in a **tree-like structure**.
- Supports **parent–child (one-to-many)** relationships.
- Each child has **only one parent**.

Real Example: 1.Organization chart 2.File system directory structure



2.Network Data Model:

- Extends hierarchical model.
- Supports **many-to-many** relationships.
- A child can have **multiple parents**.



Real Example: 1. Telecom network systems 2.Airline reservation systems

3.Relational Data Model:

- Data stored in **tables (relations)**.
- Each table consists of rows and columns.
- Key constraints maintain relationships.

Real Example: 1.Banking systems 2.Student database

Table: **EMP**

EMPNO	ENAME	SAL	DEPTNO

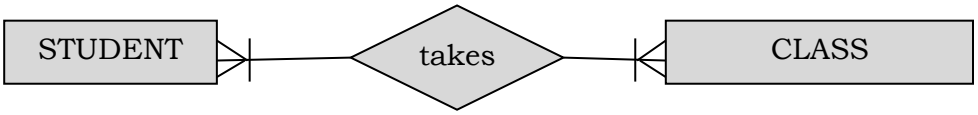
Table: **DEPT**

DEPTNO	DNAME	LOC

4.Entity-Relationship (ER) Model

- Used for conceptual design of database.
- Represents entities, relationships, and attributes.

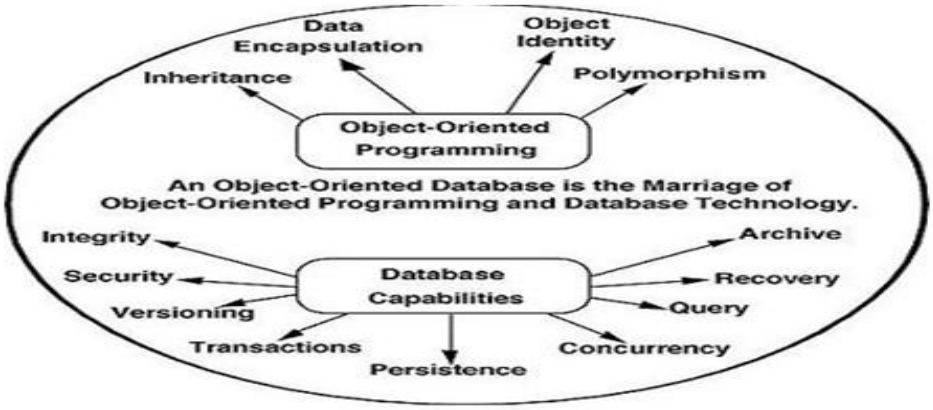
Real Example: Database design for schools, hospitals.



5.Object-Oriented Data Model

- Data stored as **objects**.
- Supports inheritance, encapsulation, polymorphism.

Real Example: 1.CAD software 2.Multimedia & graphics DB

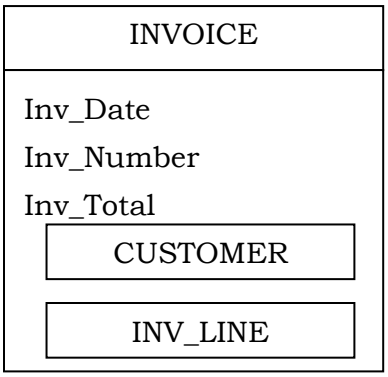


6.Object-Relational Data Model

- Hybrid of relational + object-oriented models.
- Adds object concepts to relational systems.

Real Example: PostgreSQL, Oracle

- In this model, an object is represented in a box, all the object attributes and relationships are included in that box.



SCNR Government Degree College, Proddatur

Department of Computer Science

III Sem-Unit II- DBMS

Entity-Relationship Model: Introduction, the building blocks of an entity relationship diagram, classification of entity sets, attribute classification, relationship degree, relationship classification, reducing ER diagram to tables, enhanced entity-relationship model (EER model), generalization and specialization, IS A relationship and attribute inheritance, multiple inheritance, constraints on specialization and generalization, advantages of ER modeling.

Building blocks of an entity relationship diagram

An Entity-Relationship Model (ERM) is a detailed graphical representation of the data for an organization. An E-R model is normally expressed through E-R Diagrams (ERD).

The basic components of an E-R model are

1. Entity
2. Attribute
3. Relationship

1. Entity:

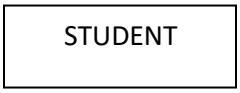
- An **entity** is a real-world object or concept that can be uniquely identified.
- It is represented by **Rectangle.**
- Name of Entities are represented by capital letters and placed inside of rectangle box.

Example: A student with a particular roll number is an entity.

a.Tangible Entity : Entities that exist in the real world physically.
Example: Person, car, etc.

b.Intangible Entity : Entities that exist only logically and have no physical existence.
Example: Bank Account, etc.

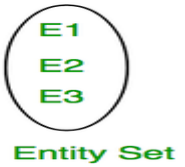
Entity Type : It refers to the category that a particular entity belongs to.



Entity Type

Example : STUDENT is entity type.

Entity Set : An entity set is a collection or set of all entities of a particular entity type at any point in time. The type of all the entities should be the same.



Example : The collection of all the students from the student table at a particular instant of time .

Consider a table student as follows :

Table Name : Student

Student_ID	Student_Name	Student_Age	Student_Gender
1	Ramesh	19	M
2	Seetha	23	F
3	Nikhil	21	M
4	Ramu	16	M

Entity : Each row is an entity.

Example : 1 Ramesh 19 M

Entity Type : Each entity belongs to the student type. Hence, the entity type is a **student.**

Entity Set : The complete data set of all entities is called entity set.

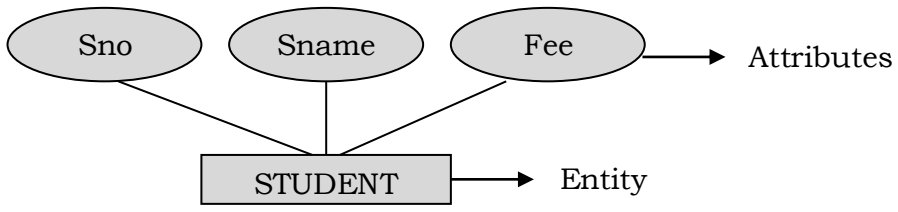
For the above table, the records with student id 1, 2, 3, 4 are the entity set.

Difference among Entity, Entity Type, Entity Set in a Table :

Entity	Entity Type	Entity Set
An entity is a real-world object or concept that can be uniquely identified.	A category of a particular entity	Set of all entities of a particular entity type.
Any particular row (a record) in a relation(table) is known as an entity.	The name of a relation (table) in RDBMS is an entity type	All rows of a relation (table) in RDBMS is entity set

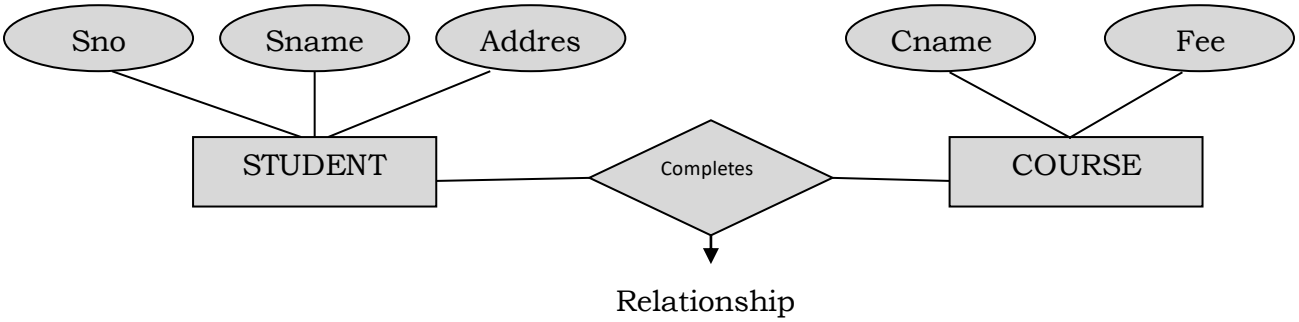
2. Attribute:

- Attributes are the properties or characteristics of an Entity.
- An attribute can be represented by an **ellipse** symbol, and connect that ellipse through a line to its entity.
- While naming an attribute, the first letter must be in upper case and remaining in lower case , name should be within the ellipse. If an attribute name contains two words, then we use the underscore (_).
- For example, the entity 'STUDENT' contains attributes like Sno, Sname, Fee etc.



3. Relationship :

- A Relationships an association (connection) between the Entities.
- Relationships are represented by placing its name inside the **diamond.**



Classification of Entity Sets

An entity is a real-world object or concept that can be uniquely identified. Entities are represented by capital letters and placed inside of rectangle box.

Entity sets can be classified into three types. They are

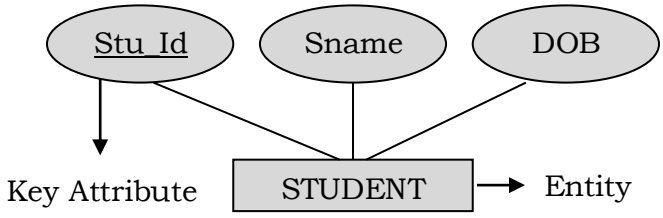
1. Strong Entity
2. Weak Entity
3. Associative Entity

1. Strong Entity:

- Exists independently.
- It Has its **own primary key**.
- Does **not depend** on any other entity.

Example: STUDENT(Stu_Id, Sname, DOB)

Symbol: Single rectangle



Strong Entity Set

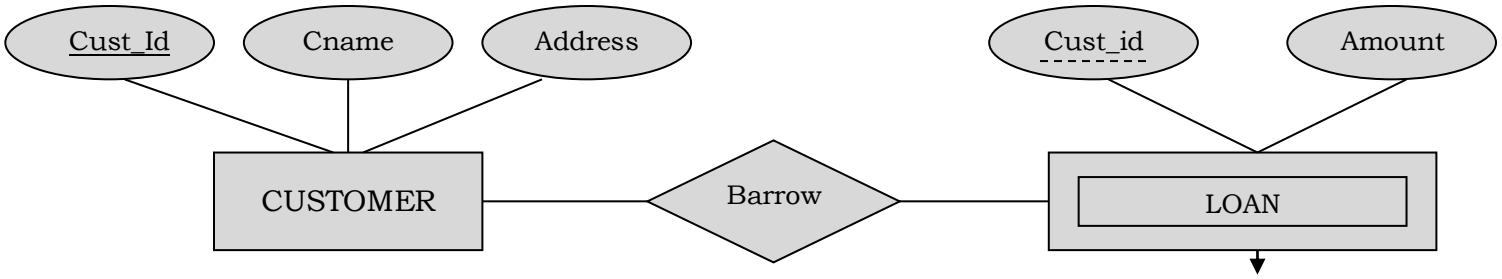
2.Weak Entity:

- Cannot exist independently.
- Depends on a strong entity for identification.
- Has a partial key (not a full primary key).

Example: LOAN (Cust_id, Amount) depends on CUSTOMER

Symbol:

- Double rectangle (entity)
- Diamond (identifying relationship)



Weak Entity Set

In the above example, for taking the loan , he must me customer of bank. so Loan account depends on customer entity witch contains the basic details of cust_Id, Cname, Address.
Here the entity LOAN depends on the entity CUSTOMER. Hence LOAN is a weak entity.

3.Associative Entity:

- An associative entity is an entity, that associates one or more entities and each entity contains the primary key.
- Created from a many-to-many relationship
- Represented by : **Diamond symbol within a rectangle.**
- Contains primary keys of the related entities (as foreign keys) and May have its own attributes

Associative Entity

Entities:

- **Student** (Student_ID, Name)
- **Course** (Course_ID, Course_Name)

A student can enroll in many courses, and a course can have many students → **M:N relationship**
Associative Entity: **Registration (**Reg_Id**,** Student_ID, Course_ID,Grade)
Here:

- Student_ID → Foreign Key
- Course_ID → Foreign Key
- Grade → Attribute of the relationship

Classification of Attributes:

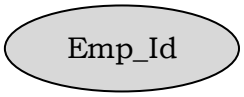
Attributes are the properties or characteristics of an Entity. These attributes are classified based on value and structure.

- **Based on value:** The attributes can be classified into Single Valued, Multi Valued, Derived, and Identifier Attributes.
- **Based on structure:** The attributes can be classified into Simple and Composite Attributes.

1. Single Valued Attribute:

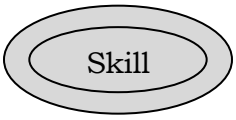
- A single valued attribute is an attribute that have only one value for each record.
- Single valued attributes are represented by using ellipse symbol.

For example, each 'EMPLOYEE' has only one 'Emp_Id' value.
Here 'Emp_Id' is single valued attribute.



2. Multi Valued Attribute:

- A multi valued attribute is an attribute that have more than one value for each record.
- Multi valued attributes are represented by using **double ellipse symbol**.



For example, each 'EMPLOYEE' has more than one 'Skill' value. Here 'Skill' is multi valued attribute.

3. Derived Attribute:

- A derived attribute is an attribute whose values can be calculated from another attribute values.
- Derived attributes are represented by using dotted line ellipse symbol

For example, the 'Service' of an 'EMPLOYEE' is calculated by another attribute 'Doj'. Here, 'Service' is derived attribute.

4. Identifier Attribute:

- An attribute which is having primary key is called as "Identifier Attribute".
- It is a unique value for each record. Identified attributes are represented by using under line in the ellipse symbol.

For example, each 'STUDENT' has only one "Hallticket_No".

5. Simple Attribute:

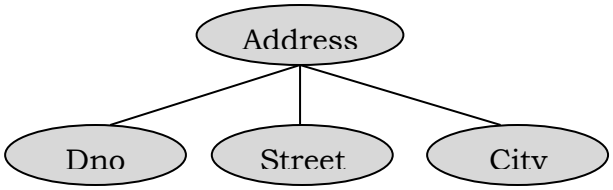
- A simple attribute is an attribute that cannot be divided into smaller attributes is called simple attribute.
- Simple attributes are represented by using ellipse symbol.

For example, 'Gender' cannot be divided.

6. Composite Attribute:

- A composite attribute is an attribute that can be divided into smaller attributes is called composite attribute.
- Composite attributes are represented by ellipse symbol.

For example, 'Address' can be divided like Dno, Street, City etc.



Classification of Relationships

Relationship:

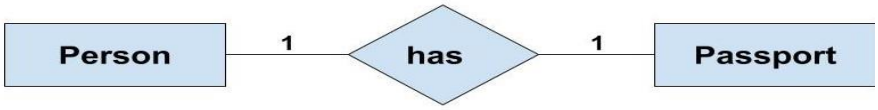
- A Relationship is an association (connection) between the Entities.
- Relationships are represented by placing its name inside the **diamond.**

There are three types of relationships that can exist between two entities.

- One-to-One Relationship
- One-to-Many
- Many-to-One Relationship
- Many-to-Many Relationship

One-to-One Relationship(1:1):

- One entity is related to only one entity.
- For example, If there are two entities 'Person' (Id, Name, Age, Address)and 'Passport'(Passport_id, Passport_no). So, each person can have only one passport and each passport belongs to only one person.



One-to-Many Relationship(1:N):

- This relationship is the most common relationship found.
- One entity relates to many entities.

Example: If there are two entity type 'Customer' and 'Account' then each 'Customer' can have more than one 'Account' but each 'Account' is held by only one 'Customer'. In this example, we can say that each Customer is associated with many Account like saving account, current account, housing loan account and personal loan account. So, it is a one-to-many relationship.

3. Many-to-One Relationship (N:1)

- Many entities relate to one entity.
- For example: A Business organization received Many orders for the same product from customers leads one to many relationship.

4.Many-to-Many Relationship

- Many entities relate to many entities.
- Example: If there are two entity type 'Customer' and 'Product' then each customer can buy more than one product and a product can be bought by many different customers.

Degree of Relationship

- **Degree of Relationship** refers to how closely two entities are connected in an **Entity–Relationship (ER) diagram**.
- It indicates the **number of entity types** that participate in a relationship.

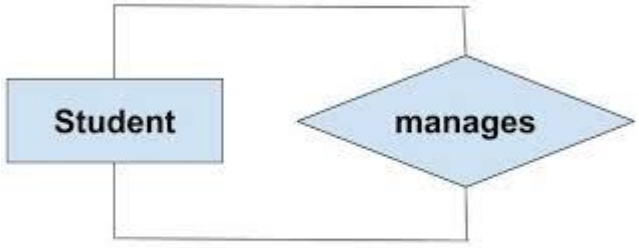
Types of degree: Now, based on the number of linked entity types, we have 4 types of degrees of relationships.

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

1. Unary Relationship (Degree = 1):

- A relationship where one entity is related to itself.
- Also called **Recursive relationship**.

 Example: An *Students* manages all the remaining students of class. In this only one entity of "Student" present, so it is example of unary relationship.



Unary Relationship
(degree 1)

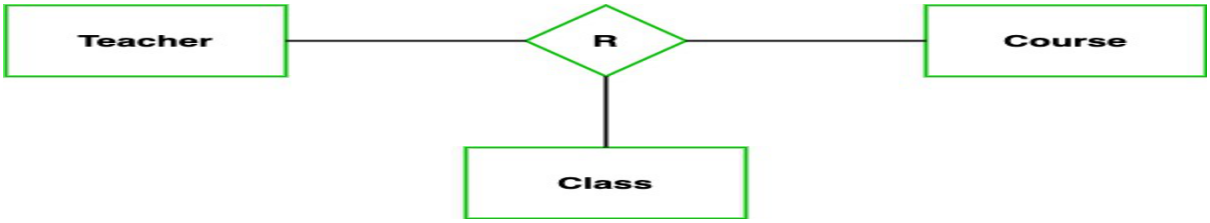
2. Binary Relationship (Degree = 2):

- A relationship between **two different entities**.
- **Example:** A *Student* enrolls in a *Course*.

We have two entity types 'Student' and 'ID' where each 'Student' has his 'ID'. So, here two entity types are associating we can say it is a binary relationship. Also, one 'Student' can have many 'daughters' but each 'daughter' should belong to only one 'father. We can say that it is a one-to-many binary relationship.

3. Ternary Relationship (Degree = 3)

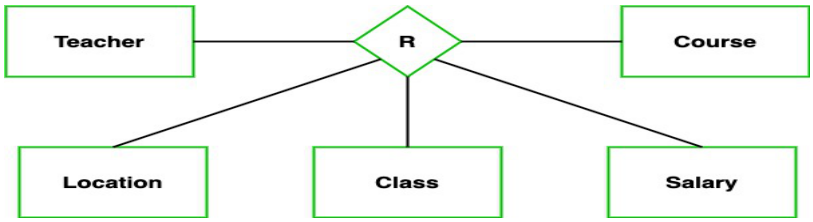
- A relationship involving **three entities**.
- Example: We have three entity types 'Teacher', 'Course', and 'Class'. The relationship between these entities is defined as the teacher teaching a particular course, also the teacher teaches a particular class.
- So, here three entity types are associating we can say it is a ternary relationship.



4.N-ary (n degree):

In the N-ary relationship, there are n types of entity that associates. So, we can say that an N-ary relationship exists when there are n types of entities.

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.



Enhanced Entity–Relationship (EER) Model

The **Enhanced Entity–Relationship (EER) Model** is an **extension of the basic ER model**. It includes additional concepts to represent **complex data relationships** more accurately.
The EER model is mainly used in **advanced database design**.

Why EER Model is Needed

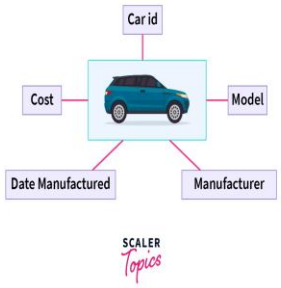
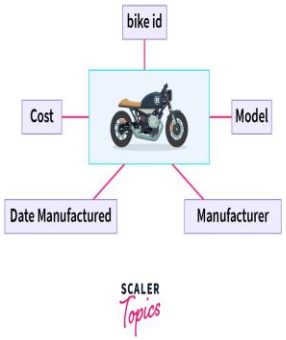
The basic ER model cannot clearly represent:

- Specialization and generalization
- Inheritance
- Constraints between entities

Generalization

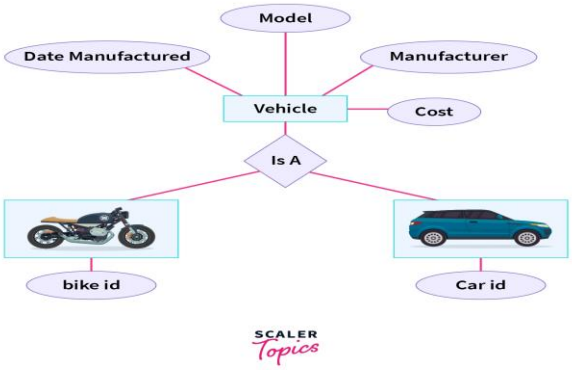
Generalization is the process of **extracting common features from two or more entities** and creating a **general (superclass) entity**.

 Example:

Car Entity:	**Bike Entity:**
Car entity can have attributes like car_ID, DateManufactured, Cost, Model, Manufacturer .	Bike entity can have attributes like bike_ID, DateManufactured, Cost, Mode l, Manufacturer .
	

Entities:	**Common attributes:**
• **Car** • **Bike**	DateManufactured, Cost, Model, Manufacturer .
These are generalized into a **Vehicle** entity.	

Bottom to top Approach

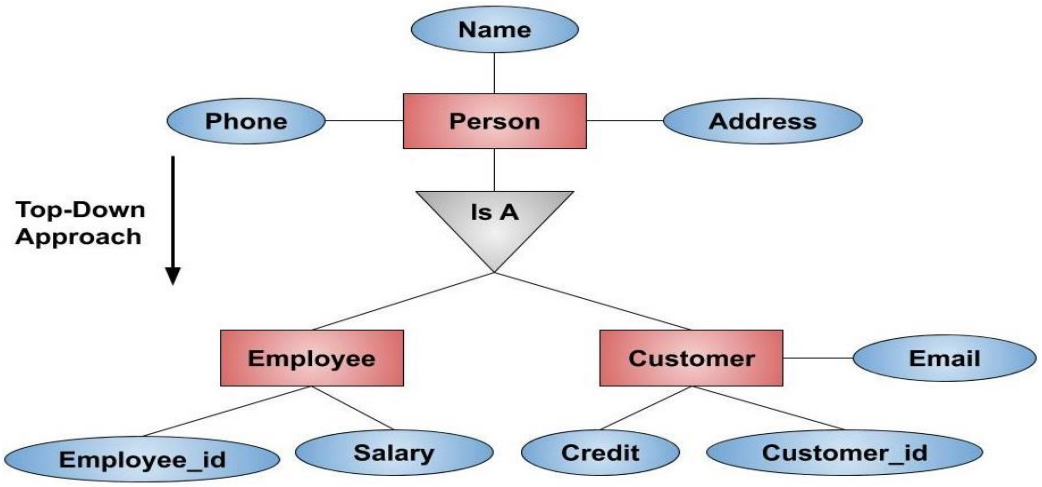


Specialization:

- The process of dividing a higher-level entity into lower-level entities based on specific characteristics is called **specialization.**
- Just it is nothing but reverse process of generalization.
- **Specialization** is a top-down approach in which a higher-level entity is broken into smaller entities.

Example: If we have a person entity type who has attributes such as Name, Phone_no, Address.

Now, suppose we are making software for a shop then the person can be of two types. This Person entity can further be divided into two entity i.e. Employee and Customer. We will specialize the Person entity type to Employee entity type by adding attributes like Emp_no and Salary. Also, we can specialize the Person entity type to Customer entity type by adding attributes like Customer_no, Email, and Credit. These lower-level attributes will inherit all the properties of the higher-level attribute. The Customer entity type will also have attributes like Name, Phone_no, and Address which was present in the higher-level entity.



Specialization

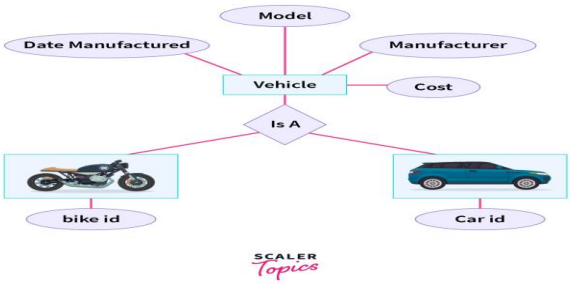
Difference between Generalization and Specialization :

Basis	Generalization	Specialization
Meaning	Combining multiple similar entities into one generalized entity	Dividing one entity into multiple specialized entities
Approach	Bottom-up approach	Top-down approach
Process	Many → One	One → Many
Purpose	To reduce redundancy and simplify the design	To represent detailed and specific characteristics
Entity	Sub-entities are merged into a super entity	Super entity is divided into sub-entities

Basis	Generalization	Specialization
Relationship		
Attribute Sharing	Common attributes are moved to the generalized entity	Sub-entities inherit attributes from the super entity
Example	Car, Bike → Vehicle	Person → Employee, Customer
Used When	Entities have common features	Entities have unique or special features

IS-A relationship

- In **DBMS / EER (Enhanced Entity–Relationship) modeling**, an **IS-A relationship** represents **inheritance** between entities.
- An **IS-A relationship** shows that **one entity is a specialized form of another entity**.
  It is used in **specialization and generalization**.
- It means **"is a type of"** or **"is a kind of."**
- **In below example** : bike id Is A Vehicle , car id Is A Vehicle



Attribute Inheritance

When a **subtype inherits all attributes** of its **supertype**, it is called **attribute inheritance**.

- Subtypes **automatically get all the attributes** of the supertype.
- Subtypes can also have **additional attributes** unique to them.
- Attribute inheritance avoids **data redundancy**.

Supertype: Vehicle:Attributes: DateManufactured, Cost, Model, Manufacturer

Subtype: Bike

- It Inherits the all attributes of super class i.e. Vehicle : DateManufactured, Cost, Model, Manufacturer to subclass: Bike
- Adds: byke_id

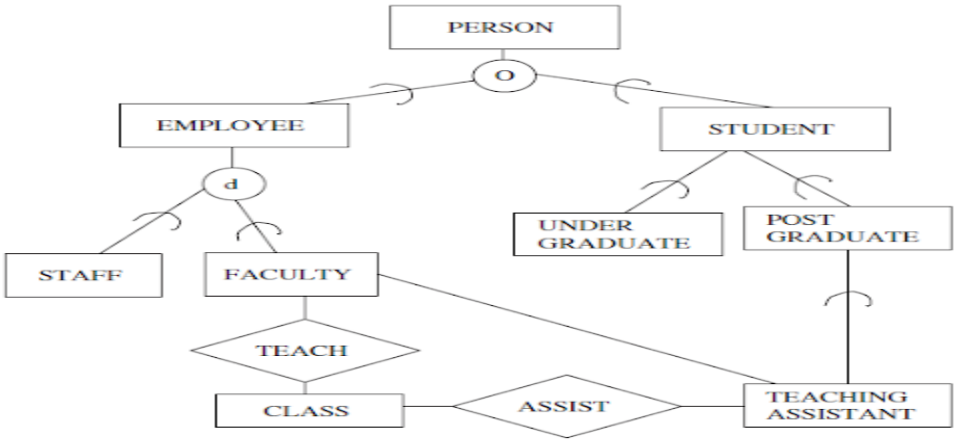
Subtype: Car

- It Inherits the all attributes of super class i.e. Vehicle : DateManufactured, Cost, Model, Manufacturer to subclass: Car
- Adds: car_id.

Multiple Inheritance

Multiple inheritance occurs when a **subtype inherits attributes and relationships from more than one super type**.

- This is an extension of the **IS-A relationship**.
- It allows a subtype to combine features from multiple super types.



If the postgraduate student is a teaching assistant, then he inherits the characteristics of faculty as well as student class. This process is called multiple inheritance.

Advantages of ER Modeling

1. **Easy to Understand**
   ER diagrams use simple symbols (rectangles, ovals, diamonds), making them easy for beginners and non-technical users to understand.
2. **Clear Representation of Data**
   It clearly shows **entities, attributes, and relationships**, helping to visualize how data is connected.
3. **Improves Communication**
   Acts as a common language between **database designers, developers, and users**, reducing misunderstandings.
4. **Helps in Database Design**
   ER modeling helps in planning the database structure before implementation, reducing errors later.
5. **Supports Normalization**
   Helps identify redundancy and supports proper normalization of data.
6. **Easy Conversion to Tables**
   ER diagrams can be easily converted into **relational tables** during database implementation.
7. **Identifies Constraints Clearly**
   Shows constraints like cardinality (1:1, 1:M, M:N) and participation (total/partial).
8. **Scalable and Flexible**
   New entities or relationships can be added easily without redesigning the entire model.
9. **Reduces Development Time**
   A well-designed ER model reduces errors and rework during database development

Reducing ER Diagram to Tables

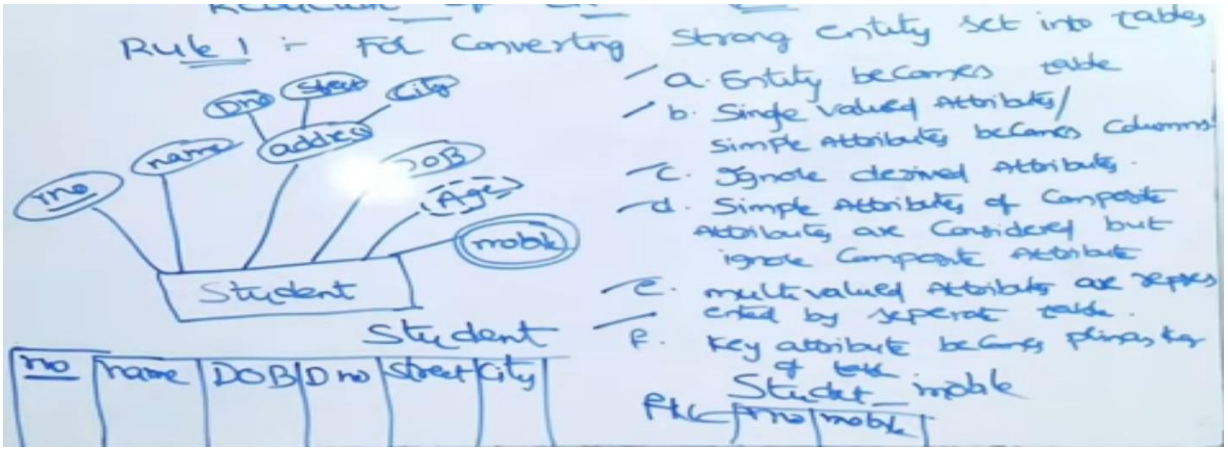
Reducing an **Entity–Relationship (ER) diagram** to tables means converting the ER model into a **relational database schema**. This process is also called **ER-to-Relational Mapping**.

1. Mapping Strong Entity Types: ***For each** strong entity**, create a table.***

Rules:

- Entity name → Table name
- Attributes → Columns

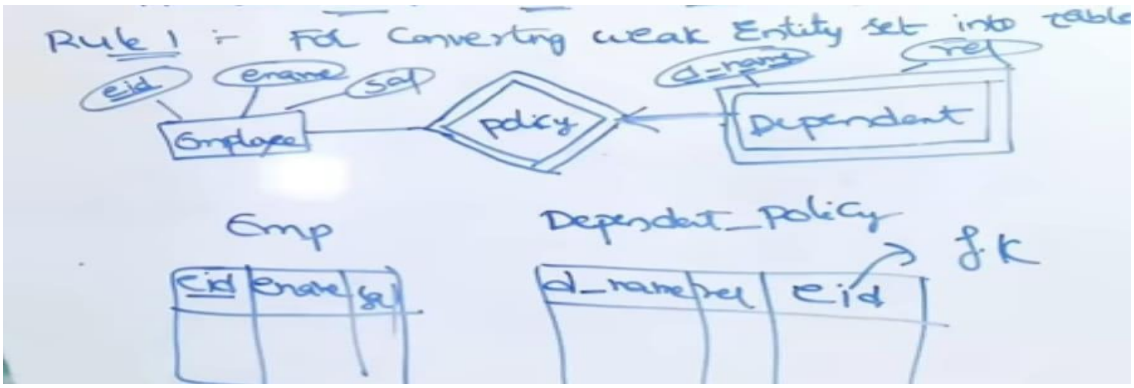
- Primary key of entity → Primary key of table



2. **Mapping Weak Entity Types:** For each **weak entity**, create a table in addition to the strong entity table.

Rules:

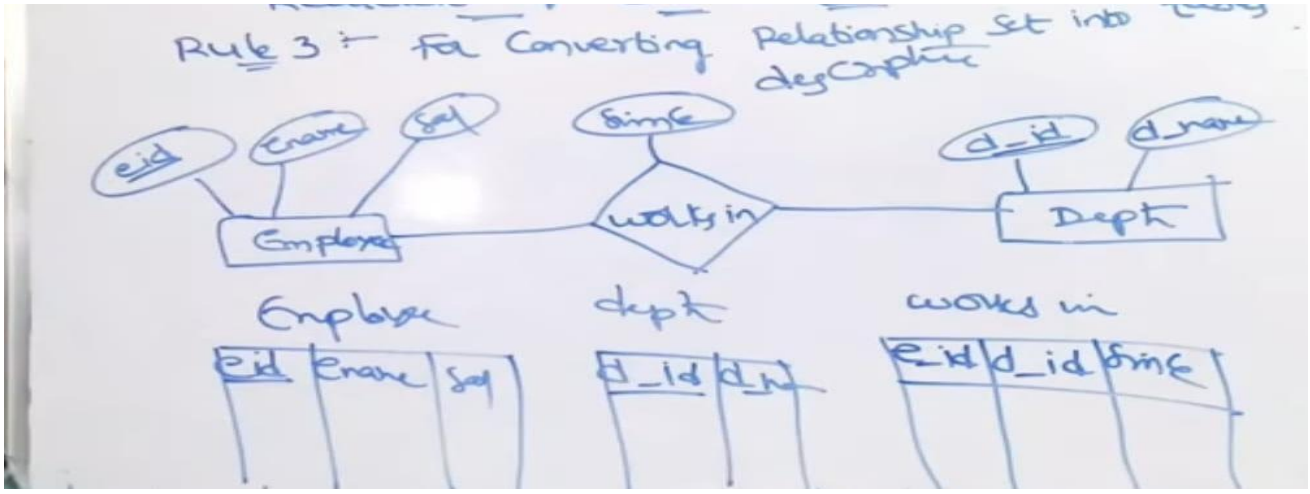
- Include all attributes of weak entity
- Include primary key of owner entity as a **foreign key**
- Primary key = (Owner's PK + Partial key)



3. <u>Converting Relationship set into tables:</u> For relationship , create a table and each entity having their own table.

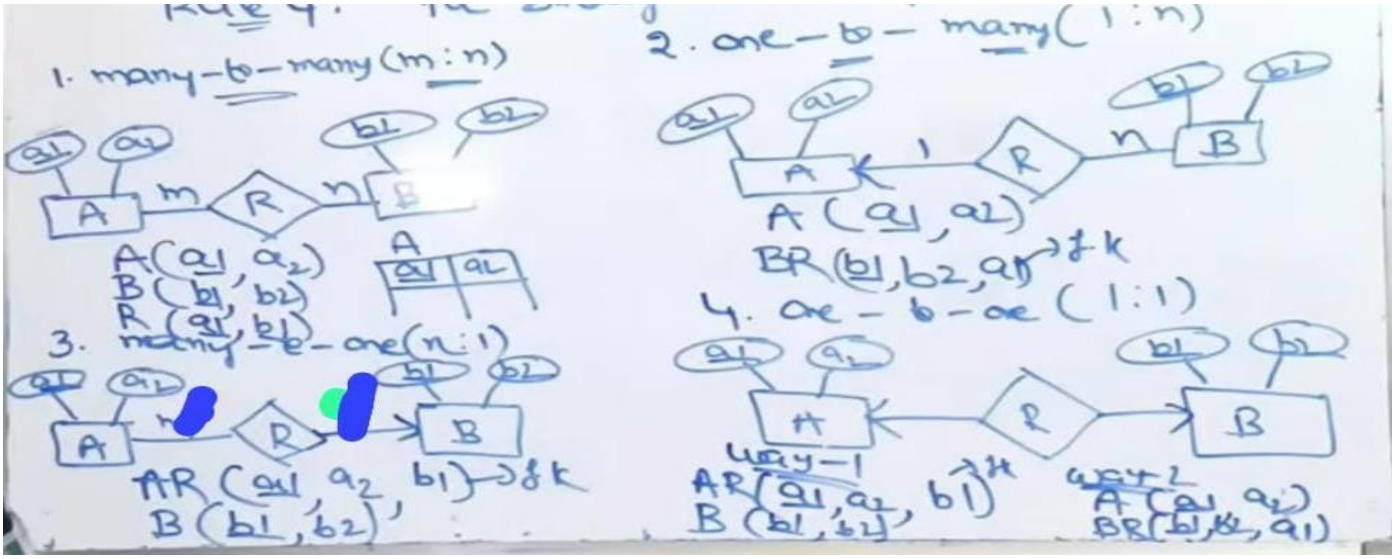
Rules:

- Include all attributes of relationship
- Include primary key of entity as a **foreign key**



4.Converting Binary Relationship with Cardinality Ratio:

A **binary relationship** involves **two entity sets**. While converting an ER diagram to tables, the **cardinality ratio** determines how foreign keys are placed.



5.Converting for Binary Relationship with cardinality ratio and Participation Constraints:

A binary relationship **connects** two entity sets**. While converting it into relational tables,** participation constraints **indicate** whether all or only some entities participate **in the relationship.**

Types of Participation:

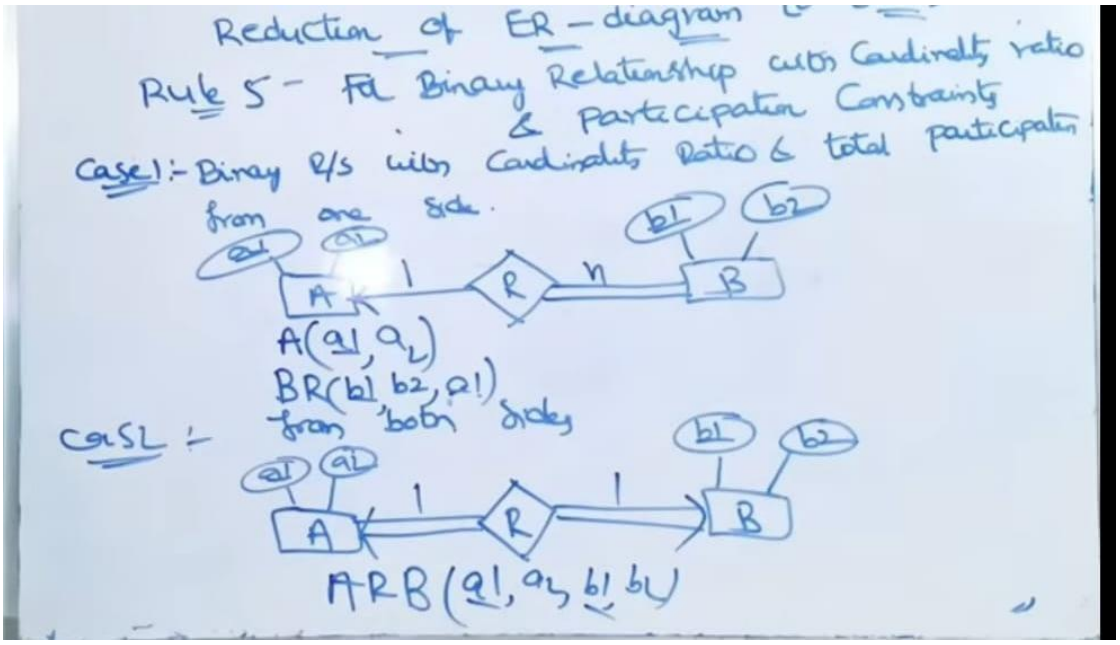
1.Total Participation 2.Partial Participation

Total Participation: An entity set has **total participation** in a relationship if **every entity** in the set **must participate** in at least one relationship instance.

ER Diagram Representation: Shown using **double lines** between entity and relationship.

Partial Participation: An entity set has **partial participation** if **some entities may not participate** in the relationship.

ER Diagram Representation: Shown using **single line.**



Constraints on specialization and generalization:

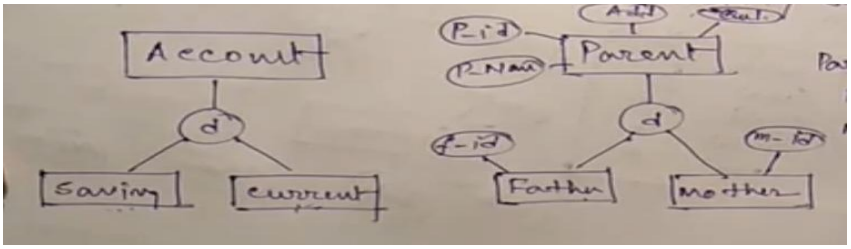
In **EER (Enhanced Entity–Relationship) modeling**, **specialization** and **generalization** are governed by a few important **design constraints**. These constraints decide **how subclasses relate to the superclass** and **how entities are distributed among subclasses**.

1. Disjointness Constraint: It specifies **whether an entity of the super class can belong to one or more subclasses**.

(a) Disjoint (D):

- An entity can belong to **only one subclass**.
- Subclasses are **mutually exclusive**.
- **Notation:** A circle labeled **"d"** between the super class and subclasses.

Example:

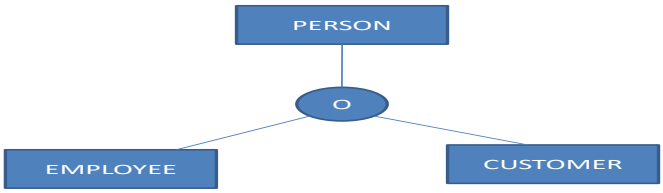


An account **cannot be both** a Saving and current account.
A Parent can not be both a Father and Mother.

(b) Overlapping (O)

- An entity can belong to **more than one subclass** at the same time.
- **Notation:** A circle labeled **"o"**.

Example:



A person **can be both** a EMPLYEE as customer also.

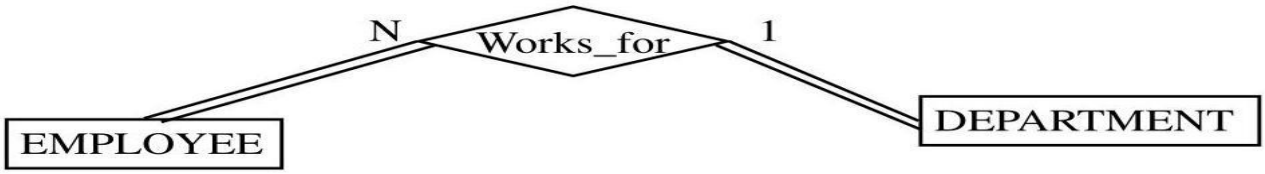
***2.* Completeness (Participation) Constraint:**

It specifies **whether every entity in the super class must belong to a subclass or not**.

(a) Total Specialization

- **Every entity** in the super class **must belong to at least one subclass**.
- No entity exists only in the super class.
- **Notation: Double line** from super class to the specialization circle.

Example:



Note: IF we assume that each department should have at least one employee being assigned, THEN:
There is a total participation constraint defined on **DEPARTMENT** via the relationship ***Works_for***.

(b) Partial Specialization

- Some entities **may not belong to any subclass**.
- Superclass entities can exist independently.
- **Notation: Single line** from super class to the circle.

Example:

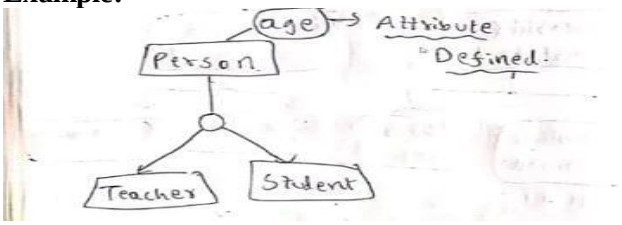
Participation in Enrolled relationship set: **Partial** Course
Total Student

It is possible that only some of the course entities are related to the student entity set through the enrolled relationship set.

3. Membership / Defined Constraint: Specialization can be defined using a **condition or predicate** on an attribute.

(a) Attribute-Defined : Subclasses are formed based on attribute values.

Example:

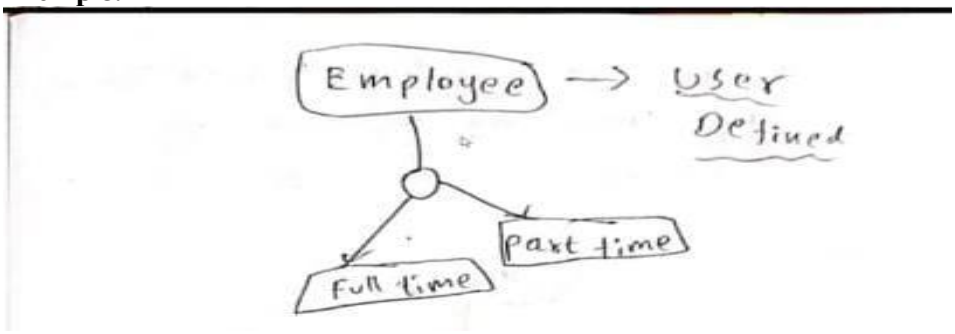


Depends up on the person age(attribute) , findout the person is a Teacher or student. if the person is age above 25 years , the person is Teacher otherwise Student.

(b) User-Defined:

- Subclasses are defined manually by the designer, not by attribute values.

Example:



A user is a person become full time or part time based up on the salary offered by the institution. So it is depends on user so it is known as user Defined.

SCNR Government Degree College, Proddatur
Department of Computer Science
II B.Sc (CS)-IV Sem_-DBMS

Syllabus: UNIT - III : Relational Model: Introduction, CODD Rules, relational data model, concept of key, relational integrity, relational algebra, relational algebra operations, advantages of relational algebra, limitations of relational algebra, relational calculus, tuple relational calculus, domain relational Calculus (DRC), Functional dependencies and normal forms upto 3rd normal form.

Relational Model – Introduction

The **Relational Model** was proposed by **Dr. E. F. Codd (1970)**.
It represents data in the form of **relations (tables)** and is the **most widely used data model** in DBMS.

Basic Concepts:

1. Relation (Table): A **relation** is a table with rows and columns.

Example: **STUDENT**

Roll_No	Name	Dept	Age
101	Anu	CSE	20
102	Ravi	ECE	21

2. Tuple (Row)

- Each **row** in a relation is called a **tuple**.
- Example:
  (101, Anu, CSE, 20) is a tuple.

3. Attribute (Column)

- Each **column** in a relation is called an **attribute**.
- Example: Roll_No, Name, Dept, Age.

4. Domain

- A **domain** is the set of valid values for an attribute.
- Example:
  - Domain of Age: {1–100}
  - Domain of Dept: {CSE, ECE, ME, IT}

5. Degree

  - Number of attributes in a relation
  - Degree is 4

6. Cardinality

- Number of tuples (rows) in a relation
- Cardinality is :2

Properties of a Relation

- Relation name is unique
- Attribute names are unique
- Order of rows and columns is **irrelevant**
- Each cell contains **atomic (indivisible) values**
- No duplicate tuples

Codd's Rules (Rules of Relational Model)

Dr. E. F. Codd proposed **12 rules** (Rule 0 to Rule 12) to define a **true Relational DBMS (RDBMS)**.

Rule 0: Foundation Rule: A system must manage databases **entirely through relational capabilities**.

Rule 1: Information Rule: All data must be represented **only as values in tables**.

Rule 2: Guaranteed Access Rule: Each data value must be accessible using:

1. Table name 2. Primary key 3.Column name

Rule 3: Systematic Treatment of NULL Values: NULL values must be handled uniformly to represent:

1.Missing data 2. Unknown data 3. Not applicable data

Rule 4: Active Online Catalog: The database description (metadata) must be stored as **tables** and accessible using SQL.

Rule 5: Comprehensive Data Sublanguage Rule: There must be **one complete language** (like SQL) that supports: Data definition, Data manipulation, Integrity constraints, Authorization, Transaction control

Rule 6: View Updating Rule: All **theoretically updatable views** must be updatable.

Rule 7: High-Level Insert, Update, Delete: Must support **set-level operations**, not record-by-record.

Rule 8: Physical Data Independence: Changes in **physical storage** should not affect application programs.

Rule 9: Logical Data Independence: Changes in **logical structure** (tables, columns) should not affect applications.

Rule 10: Integrity Independence: Integrity constraints must be:

Defined in the database, Stored in the catalog, Not embedded in applications

Rule 11: Distribution Independence: Users should not be aware of **data distribution** across locations.

Rule 12: Non-Subversion Rule: Low-level access must not bypass **security and integrity rules**.

Keys in Relational Model

In the **relational data model**, a **key** is an attribute or a set of attributes used to **uniquely identify a tuple (row)** in a relation (table). Keys help maintain **data integrity** and establish **relationships** between tables.

1. Super Key

- A **super key** is a set of one or more attributes that can **uniquely identify** a tuple in a relation.
- It may contain **extra attributes**.

Example:
STUDENT(StudentID, Name, Email)

- Super keys: {StudentID}, {StudentID, Name}, {StudentID, Email}

2. Candidate Key

- A **candidate key** is a **minimal super key**.
- No proper subset of a candidate key can uniquely identify a tuple.

Example:

- Candidate keys: {StudentID}, {Email}

3. Primary Key

- One candidate key chosen to **uniquely identify tuples** in a table.
- **Cannot be NULL** and must be **unique**.

Example:

- Primary key: StudentID

4. Alternate Key

- Candidate keys **not selected** as the primary key.

Example:

- Alternate key: Email

5. Composite (Compound) Key

- A key formed using **two or more attributes**.

- Used when a single attribute is not sufficient.

Example:
ENROLLMENT(StudentID, CourseID, Grade)

- Composite key: {StudentID, CourseID}

6. Foreign Key

- An attribute (or set of attributes) in one table that **references the primary key** of another table.
- Used to maintain **referential integrity**.

Example:
STUDENT(StudentID, Name)
ENROLLMENT(StudentID, CourseID)

- StudentID in ENROLLMENT is a **foreign key** referencing STUDENT(StudentID)

7. Unique Key

- Ensures that all values in a column are **unique**.
- May allow **NULL values** (depends on DBMS).

Example:

- Email as a unique key

Relational Integrity (DBMS)

Relational integrity refers to a set of rules that ensure the **accuracy, consistency, and correctness of data** in a relational database. It ensures that relationships between tables remain valid.

Types of Relational Integrity Constraints

Relational integrity is mainly enforced through the following constraints:

1. Entity Integrity

- Ensures that **each tuple (row) is uniquely identifiable**.
- **Primary key values cannot be NULL**.
- No two rows can have the same primary key value.

Example:
STUDENT (RollNo **PK**, Name, Dept)
→ RollNo must be unique and not NULL.

2. Referential Integrity

- Ensures that a **foreign key value must match a primary key value** in the referenced table or be NULL.
- Maintains **consistency between related tables**.

Example:
STUDENT (RollNo **PK**)
MARKS (RollNo **FK**)

✓ Allowed: RollNo in MARKS exists in STUDENT
✗ Not allowed: RollNo in MARKS that does not exist in STUDENT

3. Domain Integrity

- Ensures that **attribute values are within a valid domain**.
- Domain includes data type, range, format, and constraints.

Example:

- Age must be between 18 and 30
- Gender $\in$ {M, F}
- Salary > 0

4. Key Integrity

- Ensures that **candidate keys and primary keys remain unique**.
- Prevents duplicate key values.

5. User-Defined Integrity (Business Rules)

- Constraints defined by users based on real-world requirements.

Example:

- A student can enroll in **maximum 5 courses**
- Attendance must be $\geq$ 75%

Relational Algebra

Relational Algebra is a **procedural query language** used to retrieve data from relational databases. It operates on **relations (tables)** and produces **relations as output**.

 It forms the **theoretical foundation of SQL**.

Key Features

- Input: One or more relations
- Output: A new relation
- Uses **operators** to manipulate data
- Specifies **how** data is obtained

Advantages of Relational Algebra

- **Formal query language** – Provides a mathematical foundation for relational databases.
- **Procedural in nature** – Clearly specifies how data is retrieved step by step.
- **Basis for SQL** – Forms the theoretical foundation for SQL and query optimization.
- **Simple and precise** – Uses well-defined operations (select, project, join, etc.).
- **Query optimization** – Helps DBMS choose efficient execution plans.
- **Independent of data storage** – Focuses only on logical data representation.

Limitations of Relational Algebra

- **Procedural complexity** – Requires users to specify the execution steps explicitly.
- **Not user-friendly** – Difficult for non-technical users to understand and use.
- **Limited expressive power** – Cannot express some complex queries (e.g., recursive queries).
- **No support for aggregation (basic form)** – Aggregate functions are not part of basic relational algebra.
- **Theoretical in nature** – Not used directly by end users for querying databases.

Relational Algebra Operations

Relational Algebra is a procedural query language used in DBMS to manipulate relations (tables). Its operations are classified into basic (primitive) and derived operations.

A. Basic Operations:

1. Selection (σ)

- Selects rows that satisfy a condition.
- **Syntax:** σ_{condition}(Relation)

Example: σ_{age > 20}(STUDENT)

2. Projection (π)

- Selects specific columns.
- **Syntax:** π_{attributes}(Relation)

Example: π_{name, age}(STUDENT)

3. Union (∪)

- Combines tuples from two relations.
- Relations must be **union compatible**.

Syntax: R ∪ S

4. Set Difference (−)

- Returns tuples present in one relation but not in the other.

Syntax: R − S

5. Cartesian Product (×)

- Combines every tuple of one relation with every tuple of another.

Syntax: R × S

***6.* Rename (ρ)**

- Renames a relation or its attributes.

Syntax: ρ(NewName, attributes)(Relation)

B. Derived Operations

7. Join (⋈)

- Combines tuples from two relations based on a condition.

Types:

- Theta Join (⋈θ)
- Equi Join
- Natural Join

Example: STUDENT ⋈ DEPARTMENT

8. Intersection (∩)

- Returns common tuples from two relations.

Syntax: R ∩ S

9. Division (÷)

- Used for **"for all"** type queries.

Example: Students who have taken **all** courses.

Relational Calculus

Relational Calculus is a **non-procedural (declarative) query language** used in DBMS.

- It specifies **what data is required**, not **how to retrieve it**.
- Based on **predicate logic (first-order logic)**.
- Forms the **theoretical foundation of SQL**.

Types of Relational Calculus

1. **Tuple Relational Calculus (TRC)**
2. **Domain Relational Calculus (DRC)**

1. Tuple Relational Calculus (TRC): TRC queries are expressed using **tuple variables**.

Definition: Tuple Relational Calculus defines queries using **variables that represent tuples of relations**.

General Syntax: { t | P(t) }

Where:

- t → tuple variable
- P(t) → predicate (condition) involving t
- Result is a **set of tuples** satisfying P(t)

Example Relation:
STUDENT(Sid, Name, Dept)

Query: *Find names of students from CSE department*

{ t.Name | STUDENT(t) ∧ t.Dept = 'CSE' }

Key Features of TRC

- Uses **tuple variables**
- Uses **logical operators**: ∧ (AND), ∨ (OR), ¬ (NOT)

- Uses **quantifiers**:
  - ∃ (Existential)
  - ∀ (Universal)
- Closer to **natural language**

Advantages

- Simple and intuitive
- Easy to understand conditions

Limitation

- May lead to **unsafe queries** if not properly restricted

3. Domain Relational Calculus (DRC): DRC uses **domain variables** instead of tuple variables.

Domain Relational Calculus defines queries using **variables that represent attribute values (domains)**.

General Syntax: { <x1, x2, ..., xn> | P(x1, x2, ..., xn) }

Where:

- x1, x2, ... xn → domain variables
- P → predicate condition
- Result is a **set of attribute values**

Example: Relation: **STUDENT(Sid, Name, Dept)**

Query: *Find names of students from CSE department*

{ <Name> | ∃Sid ∃Dept (STUDENT(Sid, Name, Dept) ∧ Dept = 'CSE') }

Key Features of DRC

- Uses **domain (attribute) variables**
- Uses **logical connectives and quantifiers**
- More **mathematical** than TRC

Advantages

- Precise and formal
- Directly works on attribute values

Limitation

- More complex to write
- Less intuitive than TRC

Difference between TRC and DRC

Aspect	TRC	DRC
Variables	Tuple variables	Domain variables
Representation	Entire tuple	Individual attribute values
Complexity	Simpler	More complex
Readability	High	Moderate
Query Output	Set of tuples	Set of attribute values

Functional Dependency

Functional Dependency is a constraint between two sets of attributes in a relation that describes how one attribute (or set of attributes) determines another.

Definition: In a relation **R**, an attribute set **X** functionally determines attribute set **Y** (written as **X → Y**) if:

For any two tuples *t1* and *t2* in R,
if *t1[X] = t2[X]*, then *t1[Y] = t2[Y]*

 Meaning: The value of **X uniquely determines the value of Y**.

Example: Consider a relation **STUDENT(StudentID, Name, Dept, DeptHead)**

- **StudentID → Name, Dept**
  (Each student ID identifies exactly one name and department)
- **Dept → DeptHead**
  (Each department has one department head)

Normalization

Normalization is the process of organizing data in a database to **reduce redundancy** and **avoid update anomalies** (insert, delete, update).

 1. First Normal Form (1NF)

✅**Definition:**

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

- All attributes contain **atomic (indivisible) values**
- No **repeating groups** or **multi-valued attributes**

❌**Not in 1NF**

StudentID	Name	Subjects
1	Anu	DBMS, OS

✅In 1NF

StudentID	Name	Subject
1	Anu	DBMS
1	Anu	OS

✓ Key Point

- Each field contains **only one value**

□ 2. Second Normal Form (2NF)

✅Definition

A relation is in **2NF** if:

- It is in **1NF**
- **No partial dependency** exists
  (Non-prime attribute should not depend on part of a composite key)

□ Partial Dependency

A non-key attribute depends on **only part of a composite primary key**

❌Not in 2NF

ENROLLMENT(StudentID, CourseID, StudentName, CourseName)
Primary Key: (StudentID, CourseID)

- StudentName → StudentID
- CourseName → CourseID

✅Convert to 2NF

STUDENT(StudentID, StudentName)
COURSE(CourseID, CourseName)
ENROLLMENT(StudentID, CourseID)

□ 3. Third Normal Form (3NF)

✅Definition

A relation is in **3NF** if:

- It is in **2NF**
- **No transitive dependency** exists

 Transitive Dependency

A non-key attribute depends on **another non-key attribute**

❌Not in 3NF

EMP(EmpID, EmpName, DeptID, DeptName)

- EmpID → DeptID
- DeptID → DeptName
  EmpID → DeptName (transitive)

✅Convert to 3NF

EMP(EmpID, EmpName, DeptID)
DEPT(DeptID, DeptName)

 Summary Table

Normal Form	Condition
1NF	Atomic values, no repeating groups
2NF	1NF + No partial dependency
3NF	2NF + No transitive dependency

 Advantages of Normalization

- Reduces data redundancy
- Eliminates update anomalies
- Improves data integrity
- Efficient storage

SCNR Government Degree College, Proddatur
Department of Computer Science
II B.Sc (CS)-IV Sem_-DBMS

Syllabus: UNIT – IV: Structured Query Language: Introduction, Commands in SQL, Data Types in SQL, Data Definition Language, Selection Operation, Projection Operation, Aggregate functions, Data Manipulation Language, Table Modification Commands, Join Operation, Set Operations, View, Sub Query.

SQL : Introduction

- SQL stands for **Structured Query Language.** It is used to communicate with databases
- SQL works on **tables (relations)** consisting of rows (tuples) and columns (attributes)
- It is a **non-procedural (declarative)** language: users specify *what* data is required, not *how* to get it.

Data Types in SQL

Data types specify the kind of data that can be stored in a table column.

1. Numeric Data Types

- **INT / INTEGER** – Stores whole numbers
- **SMALLINT / BIGINT** – Smaller or larger range of integers
- **DECIMAL(p, s) / NUMERIC** – Exact decimal values
- **FLOAT / REAL / DOUBLE** – Approximate numeric values

2. Character (String) Data Types

- **CHAR(n)** – Fixed-length character data
- **VARCHAR(n)** – Variable-length character data
- **TEXT** – Large text data

3. Date and Time Data Types

- **DATE** – Stores date (YYYY-MM-DD)
- **TIME** – Stores time (HH:MM:SS)
- **DATETIME / TIMESTAMP** – Stores date and time

4. Boolean Data Type

- **BOOLEAN / BOOL** – Stores TRUE or FALSE

5. Binary Data Types

- **BINARY / VARBINARY** – Stores binary data
- **BLOB** – Stores large binary objects

6. Special Data Types

- **ENUM** – One value from a predefined list
- **SET** – Multiple values from a list
- **JSON / XML** – Semi-structured data

SQL Commands:

SQL commands are used to create, manipulate, retrieve, control, and manage data and database structures in a relational database system.

1**. Data Definition Language (DDL):** Used to **define and modify database structure**.

1. CREATE: Creates database objects (table, view, index).

Syntax:	**Example:**

CREATE TABLE table_name (
  column1 datatype,
  column2 datatype
 	
);
	CREATE TABLE Student (
	  RollNo INT,
	  Name VARCHAR(50),
	  Age INT
	);

2. ALTER: Modifies the structure of an existing table.

Syntax:	**Example:**
ALTER TABLE table_name ADD column datatype;	ALTER TABLE Student ADD Email VARCHAR(50);

3. DROP: Deletes a database object permanently.

Syntax:	**Example:**
DROP TABLE table_name;	DROP TABLE Student;

4. TRUNCATE: Removes all records from a table (structure remains).

Syntax:	**Example:**
TRUNCATE TABLE table_name;	TRUNCATE TABLE Student;

2. Data Manipulation Language (DML): Used to **manipulate table data**.

1. INSERT: Adds new records.

Syntax:	**Example:**
INSERT INTO table_name VALUES (value1, value2);	INSERT INTO Student VALUES (1, 'Siri', 20);

- When executed, SQL prompts the user to enter values at run time.

Syntax	**Example**
INSERT INTO table_name VALUES (&value1, '&value2', '&value3');	INSERT INTO Student VALUES (&ROLLNO, '&name', &age);

2. UPDATE: Modifies existing records.

Syntax:	**Example:**
UPDATE table_name SET column=value WHERE condition;	UPDATE Student SET Age = 21 WHERE RollNo = 1;

3. DELETE: Deletes records from a table.

Syntax:	**Example:**
DELETE FROM table_name WHERE condition;	DELETE FROM Student WHERE RollNo = 1;

3. Data Query Language (DQL): Used to **retrieve data**.

SELECT: Fetches data from tables.

Syntax:	**Example:**
SELECT column1, column2 FROM table_name;	SELECT Name, Age FROM Student;

4. Data Control Language (DCL): Used to **control access permissions**.

1. GRANT: Gives privileges to users.

Syntax:	**Example:**
GRANT privilege ON table_name TO user;	GRANT SELECT ON Student TO user1;

2. REVOKE: Removes granted privileges.

Syntax:	**Example:**
REVOKE privilege ON table_name FROM user;	**REVOKE SELECT ON Student FROM user1;**

5. Transaction Control Language (TCL): Used to **manage transactions**.

1. COMMIT: Saves changes permanently.

Syntax: SCOMMIT;

2. ROLLBACK: Undo changes.

ROLLBACK;

3. SAVEPOINT: Sets a point to rollback**.**

SAVEPOINT sp1;

Quick Exam Summary Table

Category	**Commands**
DDL	CREATE, ALTER, DROP, TRUNCATE
DML	INSERT, UPDATE, DELETE
DQL	SELECT
DCL	GRANT, REVOKE
TCL	COMMIT, ROLLBACK, SAVEPOINT

Selection Operation (σ)

Selection is used to **choose rows (tuples)** from a relation that satisfy a given condition.
It filters data **horizontally**.

Notation:

σcondition(Relation)\sigma_{\text{condition}}(\text{Relation})σcondition(Relation)

Example:
STUDENT(StudentID, Name, Age, Dept) Select students whose age is greater than 20:

σAge>20(STUDENT)\sigma_{Age > 20}(\text{STUDENT})σAge>20(STUDENT)

Key Points (Exam Notes):

- Operates on **rows**
- Does **not** change the number of columns
- Condition uses comparison operators (=, >, <, ≥, ≤, ≠)
- Result is also a relation

SQL Equivalent:

```
SELECT * FROM STUDENT WHERE Age > 20;
```

Projection Operation (π)

Projection is used to **select specific columns (attributes)** from a relation.
It filters data **vertically**.

Notation:

πattribute list(Relation)\pi_{\text{attribute list}}(\text{Relation})πattribute list(Relation)

Example:
STUDENT(StudentID, Name, Age, Dept) Select only Name and Dept:

πName,Dept(STUDENT)\pi_{Name, Dept}(\text{STUDENT})πName,Dept(STUDENT)

Key Points (Exam Notes):

- Operates on **columns**
- Removes **duplicate rows**
- Reduces the number of attributes
- Result is also a relation

SQL Equivalent:

```
SELECT Name, Dept FROM STUDENT;
```

Aggregate Functions in SQL

Aggregate functions perform a calculation on a **set of rows** and return a **single value**.

1. COUNT(): Returns the number of rows.

Syntax	**Example**
SELECT COUNT(column_name) FROM table_name;	SELECT COUNT(*) FROM Student;

 Counts total number of records in the Student table.

2. SUM(): Returns the total sum of a numeric column.

Syntax	**Example**
SELECT SUM(column_name) FROM table_name;	SELECT SUM(Salary) FROM Employee;

 Calculates total salary of all employees.

3. AVG(): Returns the average value of a numeric column.

Syntax	**Example**
SELECT AVG(column_name) FROM table_name;	SELECT AVG(Marks) FROM Student;

 Finds average marks of students.

4. MAX(): Returns the maximum value.

Syntax	**Example**
SELECT MAX(column_name) FROM table_name;	SELECT MAX(Marks) FROM Student; Finds highest marks.

5. MIN(): Returns the minimum value.

Syntax	**Example**
SELECT MIN(column_name) FROM table_name;	SELECT MIN(Marks) FROM Student; □ Finds lowest marks.

6. GROUP BY with Aggregate Functions: Used to apply aggregate functions **group-wise**.

Syntax	**Example**
SELECT column_name, AGG_FUNC(column_name) FROM table_name GROUP BY column_name;	SELECT Department, COUNT(*) FROM Employee GROUP BY Department;

□ Counts employees in each department.

7. HAVING Clause: Used to apply conditions on aggregate results.

Syntax	**Example**
SELECT column_name, AGG_FUNC(column_name) FROM table_name GROUP BY column_name HAVING condition;	SELECT Department, AVG(Salary) FROM Employee GROUP BY Department HAVING AVG(Salary) > 30000;

□ Displays departments with average salary greater than 30,000.

Table Modification Commands

Table Modification Commands (SQL) are used to change the structure or contents of an existing table. These commands mainly fall under **DDL** and **DML**.

1. **ALTER TABLE:** Used to **modify the structure** of an existing table.

a) Add a new column: It add new columns to existence.

Syntax	**Example**
ALTER TABLE table_name ADD column_name datatype;	ALTER TABLE Student **ADD Email VARCHAR(50);**

b) Modify an existing column: To modify the existing column like data types.

Syntax	**Example**
ALTER TABLE table_name MODIFY column_name new_datatype;	ALTER TABLE Student MODIFY Name VARCHAR(100);

c) Rename a column: To rename the column.

Syntax	**Example**
ALTER TABLE table_name RENAME COLUMN old_name TO new_name;	ALTER TABLE Student RENAME COLUMN RollNo TO Student_ID;

d) Drop a column: To delete the column.

Syntax	**Example**
ALTER TABLE table_name DROP column_name;	ALTER TABLE Student **DROP Email;**

2. UPDATE: Used to **modify existing data** in a table.

a) Update all rows: Update rows in the table.

Syntax	**Example**
UPDATE table_name SET column_name = value;	UPDATE Student SET Department = 'CSE';

b) Update selected rows: Updates the selected rows only instead of all rows using WHERE condition.

Syntax	**Example**
UPDATE table_name SET column_name = value WHERE condition;	UPDATE Student SET Department = 'ECE' WHERE RollNo = 101;

3. DELETE: Used to **remove records** from a table.

a) Delete all rows:

Syntax	**Example**
DELETE FROM table_name;	DELETE FROM Student;

b) Delete specific rows:

Syntax	**Example**
DELETE FROM table_name WHERE condition;	DELETE FROM Student WHERE RollNo = 105;

4. **TRUNCATE:** Used to **remove all records permanently** (cannot be rolled back).

Syntax	**Example**
TRUNCATE TABLE table_name;	TRUNCATE TABLE Student;

5. DROP TABLE: Used to **delete the table structure and data completely**.

Syntax	**Example**
DROP TABLE table_name;	DROP TABLE Student;

✨**Summary Table**

Command	Purpose
ALTER	Modify table structure
UPDATE	Modify existing data
DELETE	Remove selected records

Command	Purpose
TRUNCATE	Remove all records
DROP	Delete table completely

Selection Operation

Set operators are used to combine the results of two or more **SELECT** queries. The important thing is that all queries must return the **same number of columns** and compatible data types.

Assume the tables:

EMPLOYEE_A

EmpID	Name	Dept
101	Ravi	HR
102	Anu	IT
103	Suresh	FIN
104	Mena	IT

EMPLOYEE_B

EmpID	Name	Dept
103	Suresh	FIN
104	Meena	IT
105	Kiran	HR
106	Divya	IT

1. **UNION:** Combines the results of two queries and **removes duplicates**.

Syntax:	**Example:**
SELECT column1, column2 FROM table1 UNION SELECT column1, column2 FROM table2;	SELECT * FROM EMPLOYEE_A UNION SELECT * FROM EMPLOYEE_B;

Result:

EmpID	Name	Dept
101	Ravi	HR
102	Anu	IT
103	Suresh	FIN
104	Meena	IT
105	Kiran	HR
106	Divya	IT

2. UNION ALL: Combines results **including duplicates**.

Syntax: SELECT column1 FROM table1 UNION ALL SELECT column1 FROM table2;	**Example:** SELECT * FROM EMPLOYEE_A UNION ALL SELECT * FROM EMPLOYEE_B;

Result:

EmpID	Name	Dept
101	Ravi	HR
102	Anu	IT
103	Suresh	FIN
104	Meena	IT
103	Suresh	FIN
104	Meena	IT
105	Kiran	HR
106	Divya	IT

3. **INTERSECT:** Returns only the rows that **appear in both queries**.

Syntax:	**Example:**
SELECT column1 FROM table1 INTERSECT SELECT column1 FROM table2;	SELECT * FROM EMPLOYEE_A INTERSECT SELECT * FROM EMPLOYEE_B;

Result:

EmpID	Name	Dept
103	Suresh	FIN
104	Meena	IT

4. **MINUS / EXCEPT:** Returns rows from the **first query that are not in the second query**.

1) MINUS is used in Oracle 2) EXCEPT is used in SQL Server / PostgreSQL

Syntax: SELECT column1 FROM table1 MINUS SELECT column1 FROM table2;	**Example:** SELECT * FROM EMPLOYEE_A MINUS SELECT * FROM EMPLOYEE_B;

Result:

EmpID	Name	Dept
101	Ravi	HR
102	Anu	IT

VIEW

A **view** is a **virtual table** based on the result of a SELECT query.

- It **does not store data physically**, but we can query it like a regular table.
- Useful for **simplifying complex queries**, **security**, and **reusability**.

Example:

Suppose we have a table employees:

emp_id	name	dept_id	salary
1	Sirisha	10	5000
2	Raju	20	6000
3	Seetha	10	5500

Syntax:	Create a view for employees in Dept 10:
CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;	CREATE VIEW dept10_employees AS SELECT name, salary FROM employees WHERE dept_id = 10;

Now, you can query the view like a table:

SELECT * FROM dept10_employees;

Result:

Name	Salary
Sirisha	5000
Seetha	5500

- **Drop a view:** DROP VIEW dept10_employees;

SUBQUERY (Nested Query)

A **subquery** is a query inside another query.

- Can be used in **SELECT**, **WHERE**, or **FROM** clauses.
- Helps in retrieving **data based on the result of another query**.

Syntax:

```
SELECT column1
FROM table1
WHERE column2 = (SELECT column2 FROM table2 WHERE condition);
```

Example 1: Subquery in WHERE

Find employees with salary **higher than the average**:

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Result: Only employees earning above average salary.

Example 2: Subquery in FROM

Get **average salary per department**, then list departments with above-average overall salary:

```
SELECT dept_id, avg_salary
FROM (
   SELECT dept_id, AVG(salary) AS avg_salary
   FROM employees
   GROUP BY dept_id
) AS dept_avg
WHERE avg_salary > 5200;
```

Example 3: Subquery with IN

Find employees in departments 10 or 20:

```
SELECT name
FROM employees
WHERE dept_id IN (SELECT dept_id FROM departments WHERE dept_id <= 20);
```

JOIN OPERATION

A **JOIN** is used to combine rows from **two or more tables** based on a **related column**.

Example Tables:

EMPLOYEE

EmpID	Name	DeptID	Salary
101	Ravi	10	30000
102	Anu	20	40000
103	Suresh	30	35000
104	Meena	20	45000

DEPARTMENT

DeptID	DeptName
10	HR
20	IT
30	Finance
40	Sales

1. **INNER JOIN: Returns** only matching records **from both tables.**

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON condition;
```

Example:

```
SELECT EmpID, Name, DeptName
FROM EMPLOYEE
INNER JOIN DEPARTMENT
ON EMPLOYEE.DeptID =
DEPARTMENT.DeptID;
```

Output

EmpID	Name	DeptName
101	Ravi	HR
102	Anu	IT
103	Suresh	Finance
104	Meena	IT

2. **LEFT OUTER JOIN:** Returns **all records from left table** and **matching records from right table**.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2  ON
condition;
```

Example:

```
SELECT EmpID, Name, DeptName
FROM EMPLOYEE
LEFT JOIN DEPARTMENT
ON EMPLOYEE.DeptID = DEPARTMENT.DeptID;
```

Output

EmpID	Name	DeptName
101	Ravi	HR
102	Anu	IT
103	Suresh	Finance
104	Meena	IT

(If DeptID didn't match, DeptName would be NULL)

3. **RIGHT OUTER JOIN: Returns** all records from right table **and** matching records from left table**.**

Syntax	**Example:**
SELECT columns FROM table1 RIGHT JOIN table2 ON condition;	SELECT EmpID, Name, DeptName FROM EMPLOYEE RIGHT JOIN DEPARTMENT ON EMPLOYEE.DeptID = DEPARTMENT.DeptID;

Output

EmpID	Name	DeptName
101	Ravi	HR
102	Anu	IT
104	Meena	IT
103	Suresh	Finance
NULL	NULL	Sales

4. **FULL OUTER JOIN:** Returns **all records from both tables**.

Syntax:	**Example:**
SELECT columns FROM table1 FULL OUTER JOIN table2 ON condition;	SELECT EmpID, Name, DeptName FROM EMPLOYEE FULL OUTER JOIN DEPARTMENT ON EMPLOYEE.DeptID = DEPARTMENT.DeptID;

Output

EmpID	Name	DeptName
101	Ravi	HR
102	Anu	IT
104	Meena	IT
103	Suresh	Finance
NULL	NULL	Sales

5. **CROSS JOIN:** Returns **Cartesian product** (all combinations).

Syntax:	**Example:**
SELECT columns FROM table1 CROSS JOIN table2;	SELECT Name, DeptName FROM EMPLOYEE CROSS JOIN DEPARTMENT;

Output: Total rows = 4 × 4 = **16 rows**

6. **SELF JOIN:** A table joined with **itself**.

Example Table (EMPLOYEE_SELF)

EmpID	Name	ManagerID
1	Ravi	NULL
2	Anu	1
3	Meena	1

Syntax & Example

```
SELECT E1.Name AS Employee, E2.Name AS Manager
FROM EMPLOYEE E1
LEFT JOIN EMPLOYEE E2
ON E1.ManagerID = E2.EmpID;
```

Output

Employee	Manager
Ravi	NULL
Anu	Ravi
Meena	Ravi

JOIN TYPES – Summary

Join Type	Description
INNER JOIN	Matching rows only
LEFT JOIN	All left + matching right
RIGHT JOIN	All right + matching left
FULL JOIN	All rows from both tables
CROSS JOIN	Cartesian product
SELF JOIN	Table joined to itself

SCNR Government Degree College, Proddatur
Department of Computer Science
II B.Sc (CS)-IV Sem_-DBMS

SYLLABUS: UNIT – V-PL/SQL: Introduction, Shortcomings of SQL, Structure of PL/SQL, PL/SQL Language Elements, Data Types, Operators Precedence, Control Structure, Steps to Create a PL/SQL, Program, Iterative Control, Procedure, Function, Database Triggers, Types of Triggers.

PL/SQL: Introduction

PL/SQL (Procedural Language / Structured Query Language) is Oracle Corporation's procedural extension of SQL.
It combines the power of SQL with procedural programming features such as **variables, conditions, loops, and exception handling**.

PL/SQL allows developers to write **blocks of code** that can be stored and executed on the Oracle database server.

Shortcomings of SQL

SQL is powerful for data manipulation and retrieval, but it has certain limitations, especially when used alone.

1. No Procedural Support

- SQL does not support variables, loops, or conditional statements.
- Complex logic cannot be implemented easily.

2. Limited Error Handling

- SQL has very poor error-handling mechanisms.
- It cannot handle runtime exceptions effectively.

3. Executes One Statement at a Time

- SQL processes individual statements, not a block of statements together.
- This makes complex operations inefficient.

4. No Decision-Making Capability

- SQL cannot use IF-ELSE, CASE (procedural), or branching logic like programming languages.

5. No Looping Constructs

- SQL cannot perform repetitive tasks using loops (FOR, WHILE, etc.).

6. Difficult to Maintain Complex Logic

- Large and complex queries become hard to read, debug, and maintain.

7. Increased Network Traffic

- Each SQL statement is sent separately to the database server, increasing network overhead.

8. Poor Security for Business Logic

- Business rules written in SQL are exposed and cannot be easily hidden.

Structure of PL/SQL

PL/SQL follows a **block-structured format**. A PL/SQL block is divided into three parts:

1. **Declaration Section (DECLARE)**
   - Optional section
   - Used to declare variables, constants, cursors, and user-defined exceptions
2. **Execution Section (BEGIN)**
   - Mandatory section
   - Contains executable statements such as SQL commands, conditional statements, and loops
3. **Exception Handling Section (EXCEPTION)**
   - Optional section
   - Used to handle runtime errors using predefined or user-defined exceptions

General Syntax:	**Example:**

DECLARE
  -- declarations
BEGIN
  -- executable statements
EXCEPTION
  -- error handling
END;
/
	DECLARE
	  v_name VARCHAR2(20) := 'Students';
	BEGIN
	  DBMS_OUTPUT.PUT_LINE('Welcome ' || v_name);
	EXCEPTION
	  WHEN OTHERS THEN
	  DBMS_OUTPUT.PUT_LINE('Some error occurred');
	END;
	/
	Output:
	Welcome Students

PL/SQL Language Elements

PL/SQL language elements are the **basic components** used to construct PL/SQL programs.

1. Character Set

The set of valid characters used in PL/SQL.

- Letters (A–Z, a–z)
- Digits (0–9)
- Special symbols (+ - * / = < > @ #)
- White spaces (space, tab, newline)

2. Identifiers

Names given to PL/SQL variables, constants, procedures, etc.

- Must start with a letter
- Max length: 30 characters
- Can contain letters, digits, _, $, #
- Cannot be reserved words
  Example: emp_name, total_amt

3. Literals

Constant values used in PL/SQL.

- Numeric: 100, 45.6
- Character: 'Anu'
- Boolean: TRUE, FALSE, NULL

4. Delimiters

Special symbols with predefined meaning.

- ; → statement terminator
- : → host variable
- () → grouping
- , → separator
- .. → range

5. Comments

Used to explain code; ignored by compiler.

- Single-line: --
- Multi-line: /* */

6. Data Types

Specify the type of data.

- Scalar: NUMBER, VARCHAR2, DATE, BOOLEAN
- Composite: RECORD, TABLE
- Reference: REF CURSOR
- LOB: CLOB, BLOB

7. Operators

Used to perform operations.

- Arithmetic: + - * /
- Relational: = < > <= >= <>
- Logical: AND OR NOT
- Assignment: :=

8. Expressions

Combination of variables, literals, and operators.
Example: salary * 12

9. PL/SQL Block Structure

```
DECLARE
  variable datatype;
BEGIN
  executable statements;
END;
```

PL/SQL Data Types

Data types specify the **kind of data** a PL/SQL variable can store.

1. Scalar Data Types: Store **single values**.

- **NUMBER** → Numeric values
- **VARCHAR2(size)** → Character strings
- **CHAR(size)** → Fixed-length character data
- **DATE** → Date and time
- **BOOLEAN** → TRUE, FALSE, NULL

Example:

```
age NUMBER;
name VARCHAR2(20);
dob DATE;
```

2. Composite Data Types: Store **multiple values** as a group.

- **RECORD** → Collection of related fields
- **TABLE (Associative Array)** → Set of values indexed by key

Example:

```
TYPE emp_rec IS RECORD (
  emp_id NUMBER,
  emp_name VARCHAR2(20)
);
```

3. Reference Data Types: Store **references (pointers)** to data items.

- **REF CURSOR** → Pointer to query result set

Example:

```
TYPE emp_cur IS REF CURSOR;
```

4. LOB (Large Object) Data Types: Store **large data objects**.

- **CLOB** → Character large objects
- **BLOB** → Binary large objects
- **NCLOB** → Unicode character data
- **BFILE** → Binary file stored outside DB

5. %TYPE and %ROWTYPE Attributes: Used to declare variables based on table columns or rows.

- **%TYPE** → Column datatype
- **%ROWTYPE** → Entire row structure

Example: empno emp.empno%TYPE;
emp_row emp%ROWTYPE;

PL/SQL Operator Precedence

Operator precedence defines the **order in which operators are evaluated** in an expression.

Operator Precedence in PL/SQL (Highest to Lowest)

1. **Exponentiation: ****
2. **Unary operators: + (unary), - (unary)**
3. **Multiplication and Division: *, /**
4. **Addition and Subtraction**: +, -
5. **Relational operators: =, <, >, <=, >=, <>**
6. **Logical NOT: NOT**
7. **Logical AND: AND**
8. **Logical OR: OR**

Example:

result := 10 + 5 * 2; -- Output: 20

(* has higher precedence than +)

Using Parentheses

Parentheses can be used to **override precedence**.

result := (10 + 5) * 2; -- Output: 30

Steps to Create a PL/SQL Program

The steps to create a PL/SQL program describe the systematic process of writing, compiling, and executing a PL/SQL block to perform database operations efficiently.

1. **Open the PL/SQL Environment**
   - Use tools like **SQL*Plus**, **SQL Developer**, or any Oracle IDE.
2. **Declare Variables and Constants (Optional)**
   - Define variables in the **DECLARE** section.
   - This section is optional.
3. **Begin the Execution Section**
   - Use the keyword **BEGIN** to start executable statements.
4. **Write Executable Statements**
   - Include SQL statements (SELECT, INSERT, UPDATE, etc.)
   - Include PL/SQL statements (assignments, conditions, loops).
5. **Handle Exceptions (Optional)**
   - Use the **EXCEPTION** section to handle runtime errors.
6. **End the PL/SQL Block**
   - Use **END;** to terminate the block.
7. **Execute the Program**
   - Run the program using / (in SQL*Plus) or the **Run** button in IDE.

General Syntax:	**Example:**

DECLARE
  -- declarations
BEGIN
  -- executable statements
EXCEPTION
  -- error handling
END;
/
	DECLARE
	  v_name VARCHAR2(20) := 'Students';
	BEGIN
	  DBMS_OUTPUT.PUT_LINE('Welcome ' || v_name);
	EXCEPTION
	  WHEN OTHERS THEN
	  DBMS_OUTPUT.PUT_LINE('Some error occurred');
	END;
	/
	Output: Welcome Students

Control Statements in PL/SQL

Control statements control the **flow of execution** of a PL/SQL program.
They are mainly classified into:

1. Sequential Control Structure
2. Conditional Control Statements
3. Iterative (Looping) Control Statements

Types of Control Structures in PL/SQL:

1. Sequential Control Structure: Statements execute **one after another** in the order written.

Syntax:	**Example:**	**Output:**

statement 1;
statement 2;
statement 3;
	DECLARE
	  a NUMBER := 10;
	  b NUMBER := 20;
	  c NUMBER;
	BEGIN
	  c := a + b;
	  DBMS_OUTPUT.PUT_LINE('Value of A = ' || a);
	  DBMS_OUTPUT.PUT_LINE('Value of B = ' || b);
	  DBMS_OUTPUT.PUT_LINE('Sum = ' || c);
	END;
		Sum=30

2. Conditional Control Structures: Used to make decisions based on conditions.

(a) IF Statement: Used to execute a block of code when a condition is true.

Syntax:	**Examples:**	**Output:**

IF condition
THEN
  statements;
END IF;
	DECLARE
	  a NUMBER := 10;
	BEGIN
	  IF a > 5 THEN
	  DBMS_OUTPUT.PUT_LINE('A is greater than 5');
	  END IF; END;
		A is greater than 5

(b) IF–ELSE Statement: Used to choose between two alternatives.

Syntax:	**Example:**	**Output:**

IF condition THEN
  statements;
ELSE
  statements;
END IF;
	DECLARE
	  marks NUMBER := 55;
	BEGIN
	  IF marks >= 50 THEN
	  DBMS_OUTPUT.PUT_LINE('Pass');
	  ELSE
	  DBMS_OUTPUT.PUT_LINE('Fail');
	  END IF; END;
		Pass

(c) IF–ELSIF–ELSE Statement: Used to test multiple conditions**.**

Syntax:	**Example:**	**Output:**

IF condition1 THEN
  statements;
ELSIF condition2 THEN
  statements;
ELSE
  statements;
END IF;
	DECLARE
	  marks NUMBER := 85;
	BEGIN
	  IF marks >= 90 THEN
	  DBMS_OUTPUT.PUT_LINE('Grade A');
	  ELSIF marks >= 75 THEN
	  DBMS_OUTPUT.PUT_LINE('Grade B');
	  ELSE
	  DBMS_OUTPUT.PUT_LINE('Grade C');
	  END IF;
	END;
		Grade B

(d) CASE Statement: An alternative to multiple IF–ELSIF statements.

Syntax:	**Example:**	**Output:**

CASE expression
  WHEN value1 THEN
  statement1;
  WHEN value2 THEN
  statement2;
  ...
  ELSE
  default_statement;
END CASE;
	DECLARE
	  day NUMBER := 3;
	BEGIN
	  CASE day
	  WHEN 1 THEN
	DBMS_OUTPUT.PUT_LINE('Monday');
	  WHEN 2 THEN
	DBMS_OUTPUT.PUT_LINE('Tuesday');
	  WHEN 3 THEN
	DBMS_OUTPUT.PUT_LINE('Wednesday');
	  ELSE DBMS_OUTPUT.PUT_LINE('Invalid Day');
	  END CASE;
	END;
		Wednesday

Iterative control statements

Iterative control statements in PL/SQL allow a block of code to be executed repeatedly using LOOP, WHILE LOOP, and FOR LOOP.

1. Simple LOOP

- Repeats statements **until EXIT condition is met**.
- Condition is checked **inside** the loop.

Syntax:	Example:	Output

LOOP
  statements;
  EXIT WHEN condition;
END LOOP;
	DECLARE
	  i NUMBER := 1;
	BEGIN
	  LOOP
	  DBMS_OUTPUT.PUT_LINE('Value of i = ' || i);
	  i := i + 1;
	  EXIT WHEN i > 3;
	  END LOOP;
	END;
	/
		Value of i = 1
		Value of i = 2
		Value of i = 3

2. WHILE LOOP

- Executes statements **while condition is TRUE**.
- Condition is checked **before** execution.

Syntax:	Example:	Output

WHILE condition LOOP
  statements;
END LOOP;
	DECLARE
	  i NUMBER := 1;
	BEGIN
	  WHILE i <= 4 LOOP
	  DBMS_OUTPUT.PUT_LINE('i = ' || i);
	  i := i + 1;
	  END LOOP;
	END;
	/
		i = 1
		i = 2
		i = 3
		i = 4

3. FOR LOOP

- Executes loop for a **fixed number of times**.
- Loop variable is **automatically declared and incremented**.

Syntax:	Example:	Output

FOR counter IN start..end LOOP
  statements;
END LOOP;
	BEGIN
	
	  FOR i IN 1..5 LOOP
	
	DBMS_OUTPUT.PUT_LINE('Number: ' || i);
	  END LOOP;
	END;
	/
		Number: 1
		Number: 2
		Number: 3
		Number: 4
		Number: 5

FOR LOOP (REVERSE) – Example with Output

Program	**Output**

BEGIN
  FOR i IN REVERSE 1..3 LOOP
  DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
/
	3
	2
	1

PROCEDURE

Definition: A procedure **is a named PL/SQL block that performs a specific task and** does not return a value**.**

Syntax:	**Example:**	**Output:**

CREATE OR REPLACE PROCEDURE
procedure_name
(parameter_name datatype)
IS
BEGIN
  statements;
END;
/
	CREATE OR REPLACE PROCEDURE greet_user
	IS
	BEGIN
	
	DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
	END;
	/
	--calling the Procedure
	BEGIN
	  greet_user;
	END;
	/
		Welcome to PL/SQL

FUNCTION

A **function** is a named PL/SQL block that **returns a single value** using the RETURN statement.

Syntax	Example:	Output
CREATE OR REPLACE FUNCTION function_name (parameter_name datatype) RETURN datatype IS BEGIN RETURN value; END; /	CREATE OR REPLACE FUNCTION square_num(n NUMBER) RETURN NUMBER IS BEGIN RETURN n * n; END; / --**Calling the Function** BEGIN DBMS_OUTPUT.PUT_LINE(square_num(5)); END; /	25

Procedure vs Function

Procedure	Function
Does not return value	Returns a value
Called using CALL/BEGIN	Called inside expressions
Used for actions	Used for calculations

Database Triggers

Definition: A trigger **is a** PL/SQL block **that is automatically executed (fired)** when a specific event occurs **on a table or view.**

- Triggers are mainly used for:
  - Enforcing **business rules**
  - **Auditing** changes (INSERT, UPDATE, DELETE)
  - Maintaining **referential integrity**
  - Automatically performing **actions** on data changes

Structure of a Trigger

CREATE OR REPLACE TRIGGER trigger_name
BEFORE | AFTER
INSERT | UPDATE | DELETE
ON table_name
[FOR EACH ROW]
BEGIN
-- Trigger logic (PL/SQL statements) END; /

- **BEFORE / AFTER**: Specifies when the trigger executes
- **FOR EACH ROW**: Row-level trigger (executes for each affected row)
- **Statement-level** triggers do not have FOR EACH ROW

Types of Triggers

Based on their execution time and scope, PL/SQL triggers are categorized into different types.

A. Based on Timing

Type	Description
BEFORE Trigger	Executes **before** an event (INSERT, UPDATE, DELETE)
AFTER Trigger	Executes **after** an event

B. Based on Event

Type	Description
INSERT Trigger	Fires when a row is inserted
UPDATE Trigger	Fires when a row is updated
DELETE Trigger	Fires when a row is deleted

C. Based on Level

Type	Description
Row-level Trigger	Executes **once for each affected row** (FOR EACH ROW)
Statement-level Trigger	Executes **once per SQL statement**, regardless of number of rows affected

3. Examples of Triggers

Example 1: BEFORE INSERT Trigger

Scenario

We have a table employee with columns:

- emp_id (NUMBER)
- emp_name (VARCHAR2)
- created_date (DATE)

We want to **automatically set created_date** to the current date when a new employee is inserted.

Step 1: Create Table

```
CREATE TABLE employee (
  emp_id NUMBER,
  emp_name VARCHAR2(20),
  created_date DATE
);
```

Step 2: Create BEFORE INSERT Trigger

```
CREATE OR REPLACE TRIGGER before_insert_employee
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
  :NEW.created_date := SYSDATE;
END;
/
```

Step 3: Insert Data

```
INSERT INTO employee(emp_id, emp_name) VALUES (1, 'Anusha');
INSERT INTO employee(emp_id, emp_name) VALUES (2, 'Rohit');
```

Step 4: View Table

```
SELECT * FROM employee;
```

Output

emp_id	emp_name	created_date
1	Anusha	16-JAN-2026 19:25
2	Rohit	16-JAN-2026 19:26

Explanation: created_date is automatically filled by the trigger.

Example 2: AFTER UPDATE Trigger (Row-level): Scenario: We want to **log salary changes** in a table employee_audit.

Step 1: Create Tables

```
CREATE TABLE employee (
  emp_id NUMBER,
  emp_name VARCHAR2(20),
  salary NUMBER  );
```

```
CREATE TABLE employee_audit (
  emp_id NUMBER,
  old_salary NUMBER,
  new_salary NUMBER,
  updated_on DATE );
```

Step 2: Create AFTER UPDATE Trigger

```
CREATE OR REPLACE TRIGGER after_update_salary
AFTER UPDATE OF salary ON employee
FOR EACH ROW
BEGIN
  INSERT INTO employee_audit(emp_id, old_salary, new_salary, updated_on)
  VALUES (:OLD.emp_id, :OLD.salary, :NEW.salary, SYSDATE);
END;
/
```

Step 3: Insert Employee Data

```
INSERT INTO employee VALUES (1, 'Anusha', 20000);
INSERT INTO employee VALUES (2, 'Rohit', 25000);
```

Step 4: Update Salary

```
UPDATE employee SET salary = salary + 5000 WHERE emp_id = 1;
```

Step 5: View Audit Table

```
SELECT * FROM employee_audit;
```

Output

emp_id	old_salary	new_salary	updated_on
1	20000	25000	16-JAN-2026 19:30

Explanation: Trigger automatically logs the old and new salary whenever an update occurs.