# DBMS

## Introduction to DBMS

**Data** → Data refers to a collection of raw facts or figures that can be processed to derive meaning or knowledge. In easier terms, we can say that, any fact or that can be stored
Eg → XYZ, 12

**Information** → Processed data is called information. Eg → Your Name, temperature etc.

**Database** → Collection of interrelated data is called a database.

1) It can be stored in the form of table.
2) It can be any size.

Eg → Multimedia Database — img + video
College Database — Staff + Student

**File System** →

An operating system's approach for organizing and storing data on storage units like hard drives is called a file system. In a file system, data is organized into files.

The major disadvantages of file system is

1) Data Redundancy
2) Poor Memory Utilization
3) Data Inconsistency
4) Data Security.

DataBase Management System > The acronym DBMS stands for "DataBase Management System".
Users can access databases, save data, retrieve it, update it and manage it safely and effectively with the use of a software program or combination of programs.
The presence of rules and regulations in the management system is crucial as they are necessary to uphold and maintain the database effectively.

Application of DBMS >

1) Schools and Colleges > DBMS is used to create and maintain a student information system that stores student records, including personal details, academic performance, attendance and extracurricular activities.

2) Banks > DBMS is used to maintain a centralized and secure database of customer information, including personal details, account numbers, contact information and transaction history.

Types of Databases >

1) Relational Databases (RDBMS)
2) NoSQL Databases
3) Object-Oriented Databases
4) In-Memory Databases
5) Time-Series Databases
6) Spatial Databases
7) Multimedia Databases
8) Columnar Databases
9) XML Databases
10) NewSQL Databases
11) Blockchain Databases

1) Relational Database (RDBMS) → These databases structure data into organized tables that have predefined connections between them. Data manipulation and querying are performed using SQL (Structured Query Language). Well-known instances encompass MySQL, PostgreSQL, Oracle Database and Microsoft SQL Server.

2) NoSQL Databases → NoSQL Databases are created to handle data that doesn't fit neatly into the strict setup of traditional databases. Eg → MongoDB (Document Oriented DB.

3) Object-Oriented Databases → These Databases hold objects (data and actions) utilized in object-oriented programming. They work well for applications with intricate data designs, like scientific simulations or multimedia software.

4) In-Memory Databases → In these Databases, data is ~~Keep~~ kept in the primary memory (RAM) rather than on a disk, leading to quicker data retrieval. They're employed in applications that demand instant data processing and top-notch performance.

Need of DBMS →

DBMS plays a vital role for businesses, institutions and organizations of all scales in effectively managing their data, ensuring data accuracy and security, and supporting essential decision-making processes.

It serves as the core of contemporary information systems, facilitating efficient data management and ser-

ving as a basis for a wide range of applications and services

Advantages of DBMS →

1) Data Security → DBMS implements security mechanisms that require access to sensitive information, safeguarding it from unauthorized access and potential data breaches.

2) Data Redundancy and Inconsistency → DBMS removes data redundancy, minimizing storage needs and ensuring consistency throughout the maintainence of a unified version of the data.

3) Data Integrity → DBMS guarantees data Integrity by enforcing rules and constraints that prohibit the entry of incorrect or inconsistent data into the database.

4) Data Scalability → DBMS can handle large data-sets and scale to accomodate increasing amounts of data as an organization grows.

5) Data Abstraction → DBMS offers data abstraction, allowing users and applications to interact with the database without needing to understand its underlying complexities.

Disadvantages of DBMS →

1) Cost → Acquiring, deploying and sustaining DBMS software can incur significant costs. Furthermore the hardware essential for the proficient operation of a DBMS can also lead to substantial expenses.

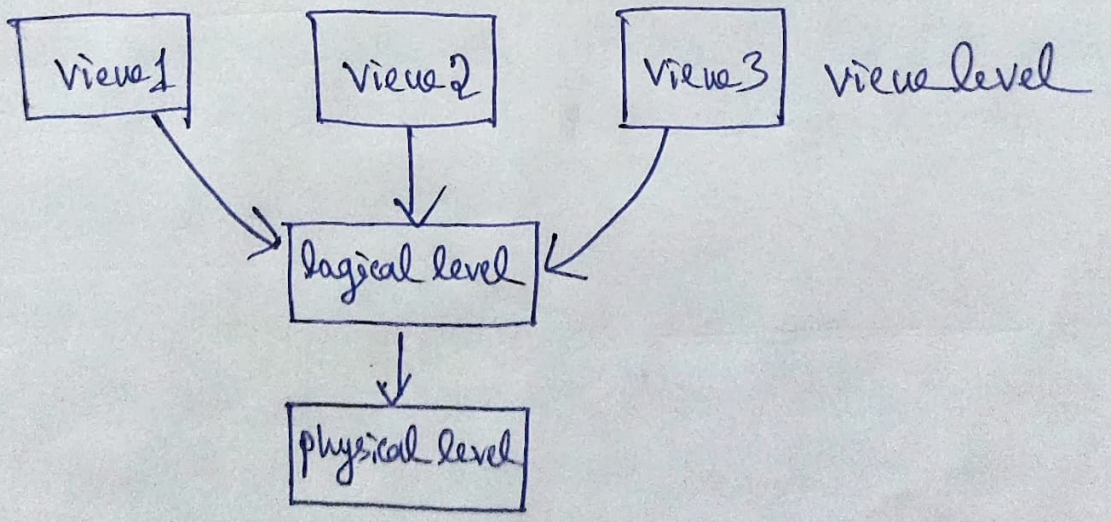
2) Scale Projects → When dealing with modest applications

and minimal data storage requirements, adopting a comprehensive DBMS could introduce avoidable intricacies and additional burdens. In these instances, more streamlined data storage alternatives could be better suited.

B) Vendor Lock-In → Once you've chosen a specific DBMS, it can be challenging to switch to a different one due to differences in data formats, query languages, and other technical aspects. This can lead to vendor lock-in, where you are dependent on a particular vendor's technology and pricing.

Data Abstraction → Database Systems are built with complex ways of organizing data. To make it easier for people to use the database, the creators hide the complicated stuff that users don't need to worry about. This hiding of unnecessary things from users is called data abstraction.

There are 3 levels of Abstraction →



1) Physical level → This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

2) Logical level → This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

3) View level → Highest level of data abstraction. This level describes the user interaction with database system.

## Schema & Instance →

Schema → A Schema is a logical container or structure that organizes and defines the structure of a database.

It defines how data is organized, what data types are used, what constraints are applied, and the relationships between different pieces of data. A Schema acts as a blueprint for the database, ensuring data integrity, consistency, and efficient data retrieval.

## Types of Schema →

1) Physical Schema → A physical Schema defines how data is stored on the underlying hardware, including details such as storage format, file organization, indexing methods, and data placement.

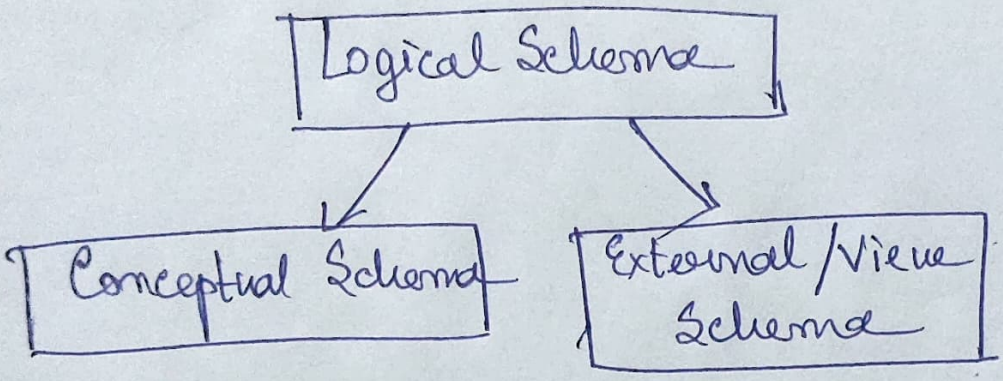
### characteristics of Physical Schema →

i) Its primary focus lies in enhancing the storage and retrieval of data to boost performance.

ii) Modifications made to the physical schema demand meticulous planning and can potentially affect the overall performance of the database.

iii) Example: Deciding to use clustered indexes on specific columns for faster retrieval.

2) Logical Schema → A logical schema defines the database's structure from a logical or conceptual perspective without considering how the data is physically stored.




Types of Logical Schema →

i) Conceptual Schema → The conceptual schema represents the overall view of the entire database. It defines the high-level structure and relationship between all data elements.

ii) External/View Schema → An external schema defines the user-specific views of the database. It focuses on the portions of the database that are relevant to specific user roles or applications.

Characteristics of Logical Schema →

i) It delineates how data is structured into tables, the

interconnections between these tables, and the restrictions placed on the data.

ii) Logical schemas prioritize data modelling and database design over considerations related to hardware or storage specifics.

iii) Example: Defining tables, specifying primary and foreign keys, and creating views for data access.

Instance → The information residing within a database at a specific point in time is referred to as the database's "instance."

Within a given database schema, the declarations of variables within its tables pertain to that specific database. The term "instance" in this context denotes the current values of these variables at a particular moment in time for that database.

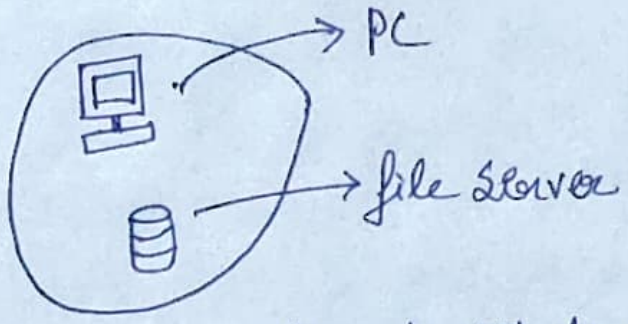
DBMS Architecture → Database Managent System architecture, refers to the structural framework and organization of a database management system. It defines how the various components of the system work together to store, manage, and retrieve data efficiently.

Types of DBMS Architecture → There are several types of DBMS Architecture. Choice of architecture depends on factors such as the type of database (e.g., relational, NoSQL) and the specific needs of an application.

1) 1-Tier Architecture
2) 2-Tier Architecture
3) 3-Tier Architecture

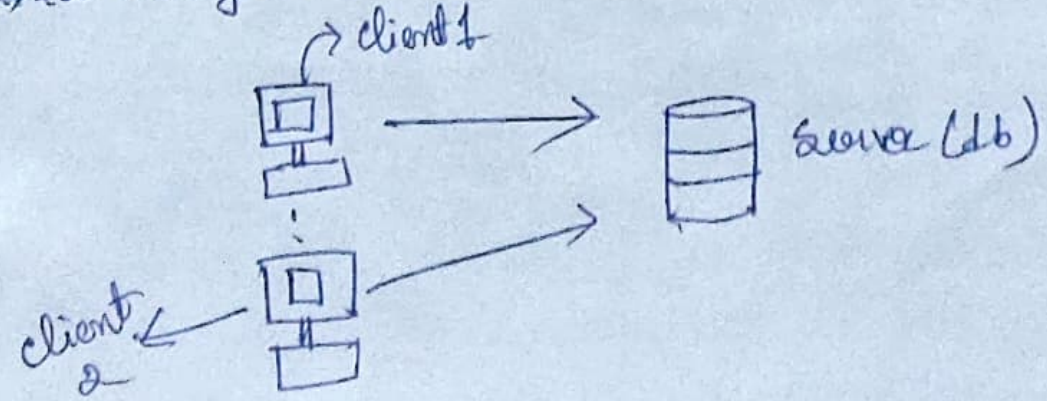
**1-Tier Architecture** → In 1 tier architecture the entire database application, including the user interface, application logic, and data storage, resides on a single machine or computer.

Example: An illustration of a straightforward single-tier architecture can be seen when you install a database on your system and use it to practice SQL queries.




**2-Tier Architecture** → In 2 tier Architecture the presentation layer runs on a client (PC, Mobile, Tablet, etc), and data is stored on a server.

This tier architecture provides added security to the DBMS as it is not exposed to the end-user directly. It also provides direct and faster communication.

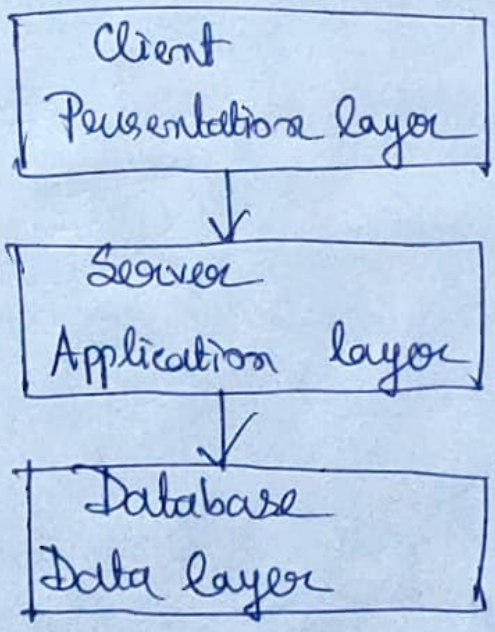


3-Tier Architecture → It separates the application into 3 logically distinct layers presentation, application, and data layer.

i) Presentation layer → It handles the user interface. Eg → your PC, Tablet, Mobile etc.

ii) Application Layer → It manages business ~~log~~ logic. Eg → Server.

iii) Data Layer → It manages data storage and processing. Eg → Database Server.




Advantages of 3-Tier Architecture →

1) Scalability → Easily adjust each tier to handle changing user demands.

2) Modularity and Maintainability → Simplify maintainence by separating responsibilities.

3) Security → Protect sensitive data with an additional layer.

4) Performance → Optimize presentation and application tiers for better performance.

Disadvantages of 3-Tier Architecture → The disadvantages of 3-Tier Architecture include increased complexity, potential latency issues, longer development time, resource overhead, and the possibility of bottlenecks.

## Data Models

Data Model → A data model within a Database Management System (DBMS) serves as an abstract representation of how data gets structured and organized within a database.

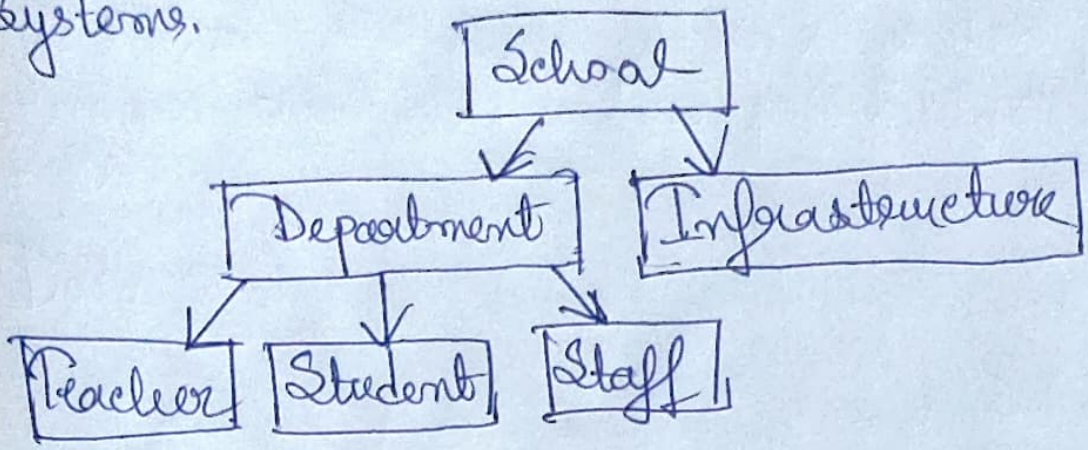
It outlines the logical arrangement of data and the connections between various data components.

Data models play a crucial role in comprehending and shaping databases, acting as a vital link between real-world entities and the actual storage of data within the database.

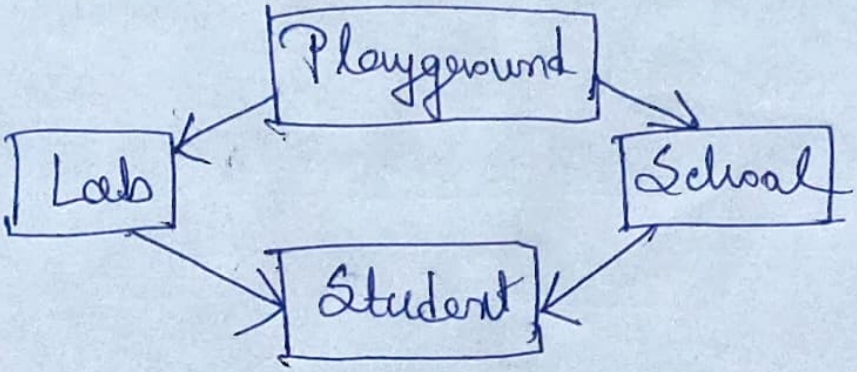
Types of Data Model →

1) Hierarchical Data Model
2) Network Data Model
3) Relational Data Model
4) Entity-Relationship Model (ER Model)
5) Object-Oriented Data Model
6) NoSQL Data Model

1) Hierarchical Data Model → This model portrays data in a manner resembling a tree structure, where each record maintains a parent-child relationship. Its primary application lies in older database systems.

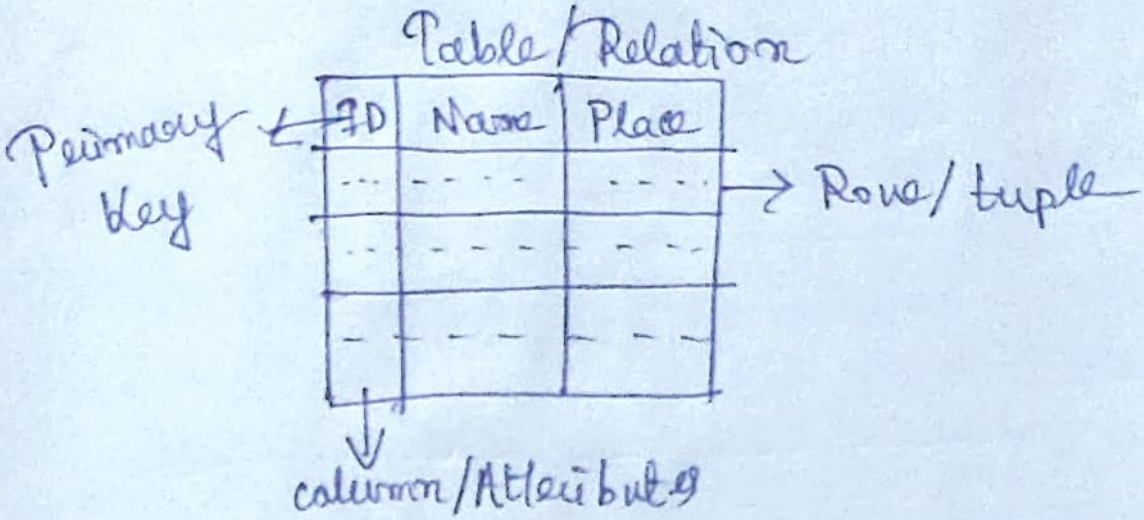



2) Network Data Model → This model shows similarities with the hierarchical approach, permitting records to hold multiple parent-child relationships. It adopts a structure akin to a graph, offering more flexibility compared to the hierarchical model.

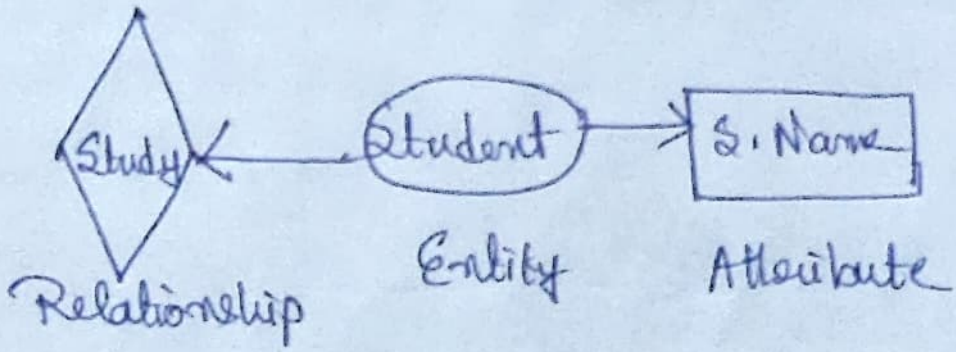



3) ~~Rela~~ Relational Data Model → Organizing data into tables (known as relations) consisting of rows and columns characterizes the relational model. It stands as the most prevalent data model, rooted in the principles of set theory, and relies on Structured Query Language (SQL)

for data ~~manipulation~~ manipulation.




4) Entity-Relationship Model (ER Model) ⇒ Utilized for crafting relational databases, the ER model represents data through entities (objects), attributes (entity properties), and ~~rela~~ relationships connecting these entities.




5) Object-Oriented Data Model ⇒ Extending the principles of object-oriented programming into the database domain, this model depicts data as objects complete with attributes and methods, fostering support for inheritance and encapsulation.

6) NoSQL Data Model ⇒ NoSQL databases encompass a diverse array of data models, such as document-oriented

(e.g., MongoDB), Key-value (e.g., Redis), column-family (e.g., Cassendra), and graph (e.g., Neo4j). These models are designed to offer scalability and flexibility when handling extensive volumes of unstructured or semi-structured data.

## ER Model in DBMS →

### Introduction to ER Model

| Entity | Attributes | Relationship |
|---|---|---|
| Things / Object | Properties of Entity | Association among Entities |
| Ex: person | Ex: name, age | Ex: Works for |

The Entity-Relationship Model (ER) stands as a prevalent conceptual modeling approach within the realm of database design.

Its primary role is to offer a visual representation of a database's architecture by illustrating the entities, their respective attributes, and the interconnections between them.

In the process of database design, the ER Model holds significant importance, aiding in the development of an efficient and systematically structured database schema.

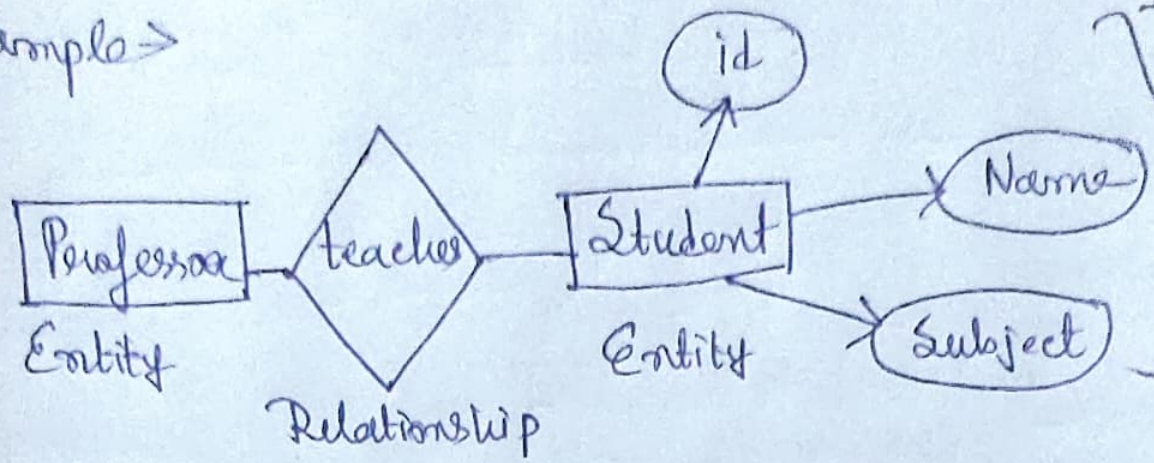
# ER Model

- **ENTITY**
  - Strong Entity
  - Weak Entity
- **ATTRIBUTE**
  - Simple Attribute
  - Composite Attribute
  - Single Valued Attribute
  - Multivalued Attribute
  - Stored Attribute
  - Derived Attribute
  - Composite Attribute
- **RELATIONSHIP**
  - One to One
  - One to Many
  - Many to One
  - Many to Many

## Symbols used in ER Model

| Figures | Symbols | For what |
|---|---|---|
| Rectangle | | Entity |
| Ellipse | | Attribute |
| Diamond | | Relationship |
| Line | | Attribute to Entity Relationship |
| Double Ellipse | | Multivalued Attributes |
| Double Rectangle | | Weak Entity |

Example →




Entity → An entity is something from the real world, like a person, place, event or idea. Each entity has specific features or traits that describe it.

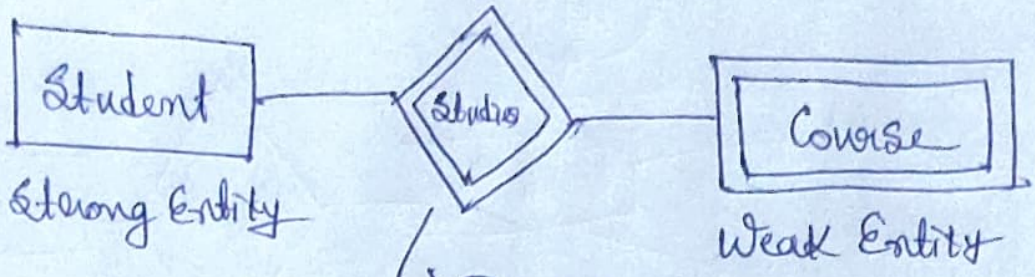
## Types of Entity

- Strong Entity (Eg: Person Entity)
- Weak Entity (Eg: Dependent Entity)

Strong Entity → A strong entity is an entity that has its own unique identifier (primary key) and is not dependent on any other entity for its existence within the database. Strong entities stand alone and have their own set of attributes. Eg: Person.

Weak Entity → A weak entity is an entity that doesn't have a primary key of its own. It relies on a related strong entity (known as the "owner" entity) for its identity. The weak entity's existence is defined by being related to the owner entity. Eg: dependent. A discriminator (partial key) of a ~~wea~~ weak entity set, distinguishes among all the entities of a weak entity set. It is combined with the primary key of the strong entity set to form its own composite primary key. It is represented by a dashed underline (- - - -)

E.g. →




→ Double diamond is used for relationship between a strong and a weak entity.

## Types of Attributes

1) **Simple Attribute** → A simple attribute is atomic and cannot be divided any further. Ex: First Name, Age, etc.

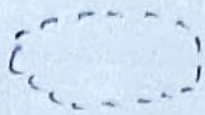
2) **Composite Attribute** → A composite attribute is made up of several smaller parts, where each part represents a piece of the whole attribute. In simpler terms it is composed of attributes which can be divided further.
Ex: Name (First Name, Last Name).

3) **Single Valued Attribute** → A single-valued attribute is an attribute that holds a single value for each entity.
Ex: Age.

4) **Multivalued Attribute** → A multi-valued attribute in a database is an attribute that can hold multiple values for a single entity. Ex: Address (permanent, residential). → double eclipse for multivalued attributes

5) **Stored Attribute** → Attribute that is stored as a part of a database record. Ex: Date of birth.

6) Derived Attribute → A derived attribute is derived from other attributes within the database.



→ dotted eclipse for derived attribute

Ex: Age derived from dob.

7) Complex Attribute → A complex attribute is an attribute that is made up of multiple smaller attributes.

Ex: Name (Composite) → First Name, Middle Name, Last Name } Simple attribute

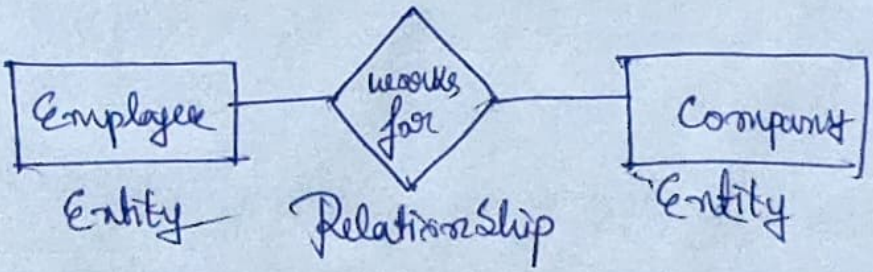
8) Key Attribute → A Key attribute is an attribute, that is unique and it uniquely identifies records in a table. Ex: Student ID

→ represents a Key attribute

## Relationship in ER Model →

Relationship in ER Model is the connection between entities (tables) based on related data.

Ex:

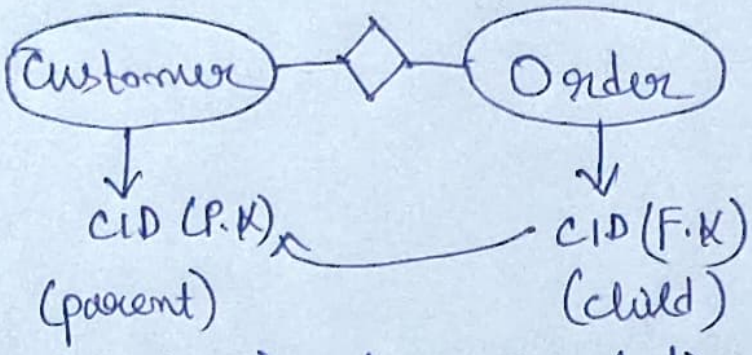



### Types of Relationship →

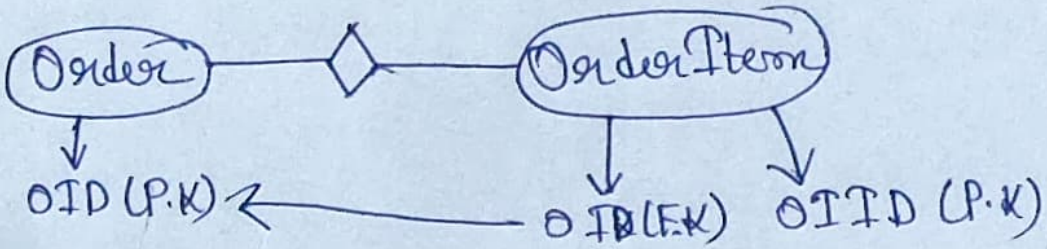
1) Strong Relationship
2) Weak Relationship

1) Strong Relationship → A strong relationship exists when two entities are highly dependent on each other, and one entity cannot exist without the other.

Ex :




2) Weak Relationship → A weak relationship, on the other hand, exists when two entities are related, but one entity can exist without the other.




Degree in DBMS → A degree in dbms refers to the number of attributes/columns that a relation/table has.

Types of Degree-

| Degree | Name | Definition |
|---|---|---|
| 1 | Unary Degree | A relation with a single attribute |
| 2 | Binary Degree | A relation with two attributes |
| 3 | Ternary Degree | A relation with three attributes |
| 4 | n-ary Degree | A relation with more than three attributes |

**Null Value** → In databases, a null value can occur for various reasons.

1) **Not Needed Information** → Sometimes, some details are asked, but they don't apply to everyone. For instance, asking for a "Spouse Name" from someone who isn't married.

2) **Don't Know The Answer** → Every now and then, we're asked a question, but we don't have an answer yet.

3) **Forgot To Fill In** → Like when you're filling out a form, and you accidentally miss putting in some important information.

## Types of Relationship in DBMS (Based on Cardinality/degree) →

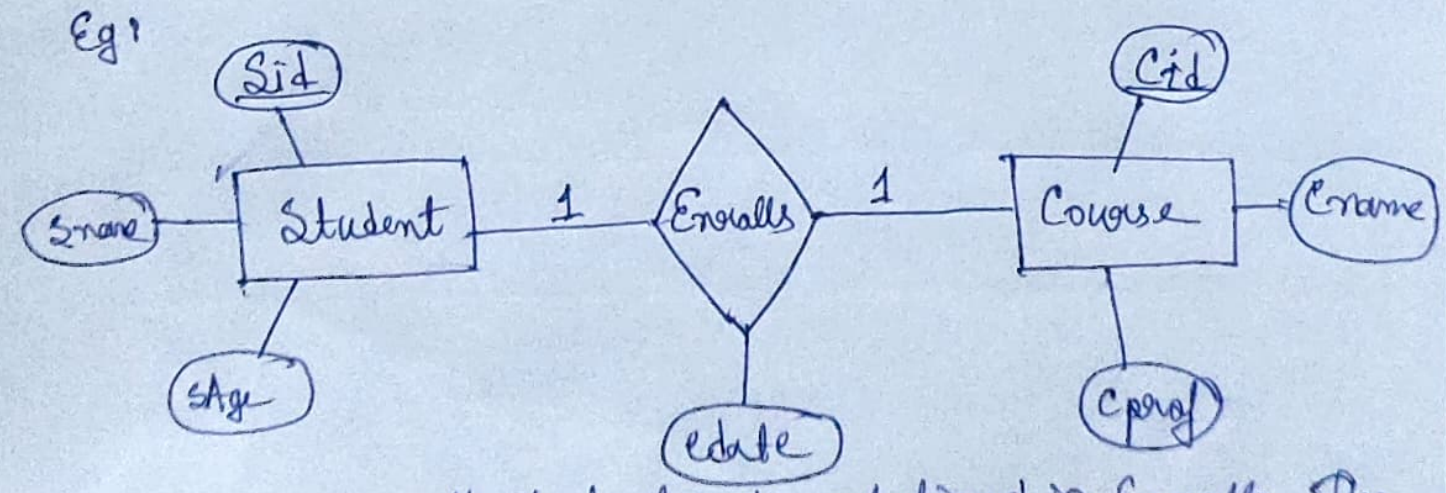
There are 4 types of relationship:

1) One to One (1-1)
2) One to Many (1-N)
3) Many to One (N-1)
4) Many to Many (N-N)

## 1 to 1 Relationship (1:1) →

Each row in one table is associated with one and only one row in the other table, and vice versa.

Eg:




Here edate is an attribute for the relationship Enrolls. The table for relationship is called junction / associative table.

⇓ Relational Model

| Sid | Sname | Sage |
|---|---|---|
| S1 | ram | 14 |
| S2 | raj | 18 |
| S3 | riti | 16 |

P.K Student

| Sid | cid | edate |
|---|---|---|
| S1 | C1 | jan |
| S2 | C2 | feb |
| S3 | C3 | mar |

f.K f.K Enroll

| Cid | Cname | cprof |
|---|---|---|
| C1 | phy | Saurav |
| C2 | math | Sanjeev |
| C3 | bio | Sumit |

P.K Course

any one can be the primary key

⇓ Combining them.

| Sid | Sname | Sag | cid | edate |
|---|---|---|---|---|
| S1 | ram | 14 | C1 | jan |
| S2 | raj | 18 | C2 | feb |
| S3 | riti | 16 | C3 | mar |

P.K F.K

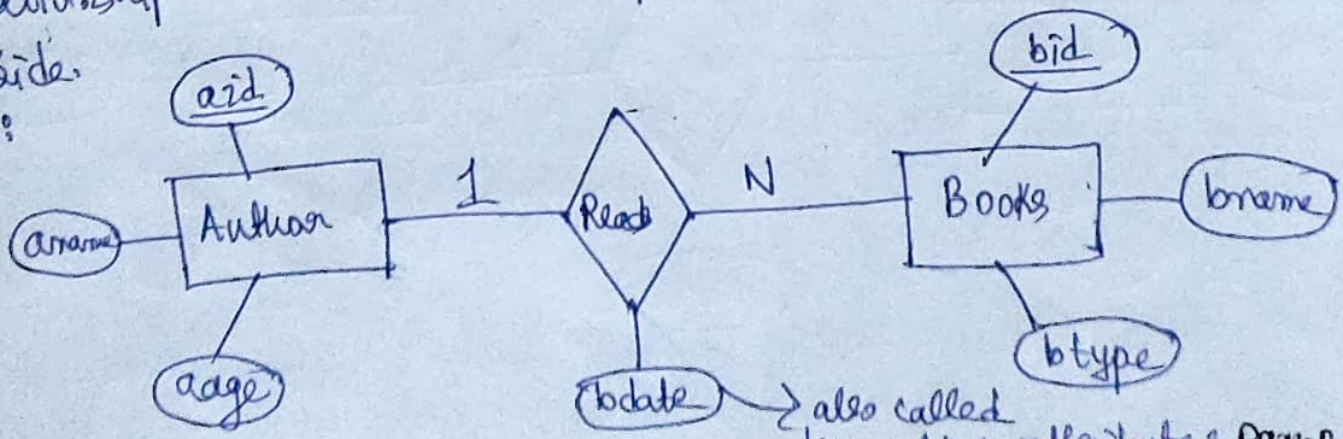
| Cid | Cname | cprof |
|---|---|---|
| C1 | phy | Saurav |
| C2 | math | Sanjeev |
| C3 | bio | Sumit |

P.K

## 1 to Many Relationship (1:N) →

A database model where one entity (record) on one side of the relationship is associated with multiple entities (records) on the other side.

Ex:




also called descriptive attributes

⇓ Relational model

| aid | aname | aage |
|---|---|---|
| a1 | ram | 14 |
| a2 | raj | 15 |
| a3 | riti | 16 |

P.K

Author

| aid | bid | bdate |
|---|---|---|
| a1 | b1 | jan |
| a2 | b2 | feb |
| a1 | b3 | mar |

f.K f.K

Reads

| bid | bname | btype |
|---|---|---|
| b1 | ab | fiction |
| b2 | cd | thrill |
| b3 | ef | drama |

P.K

Book

P.K

⇓ combining them

| aid | aname | aage |
|---|---|---|
| a1 | ram | 14 |
| a2 | raj | 15 |
| a3 | riti | 16 |

P.K

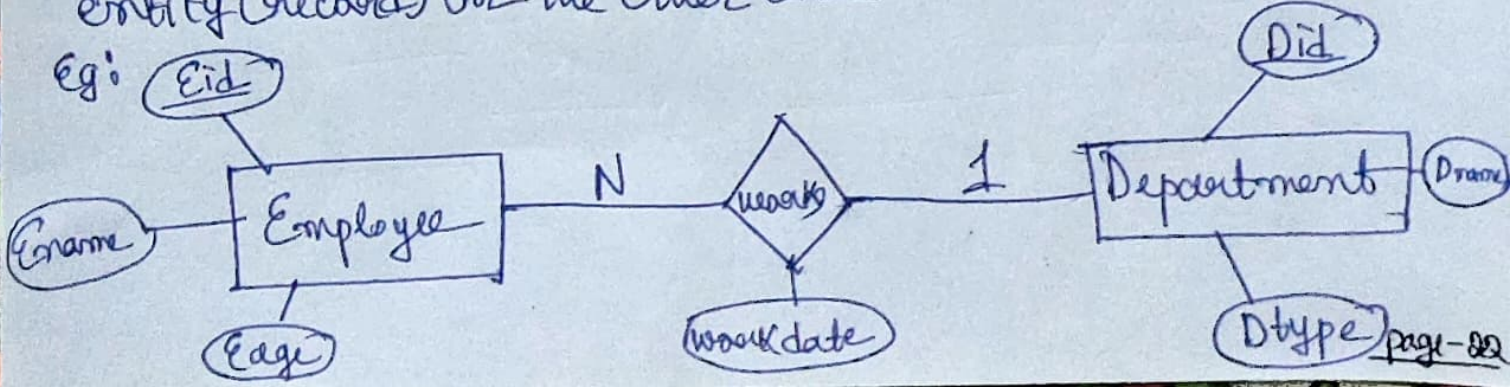
| bid | bname | btype | aid | bdate |
|---|---|---|---|---|
| b1 | ab | fiction | a1 | jan |
| b2 | cd | thrill | a2 | feb |
| b3 | ef | drama | a1 | mar |

P.K F.K

## Many to 1 Relationship (N:1) →

A database model where multiple entities (records) on one side of the relationship are associated with a single entity (record) on the other side

Eg:

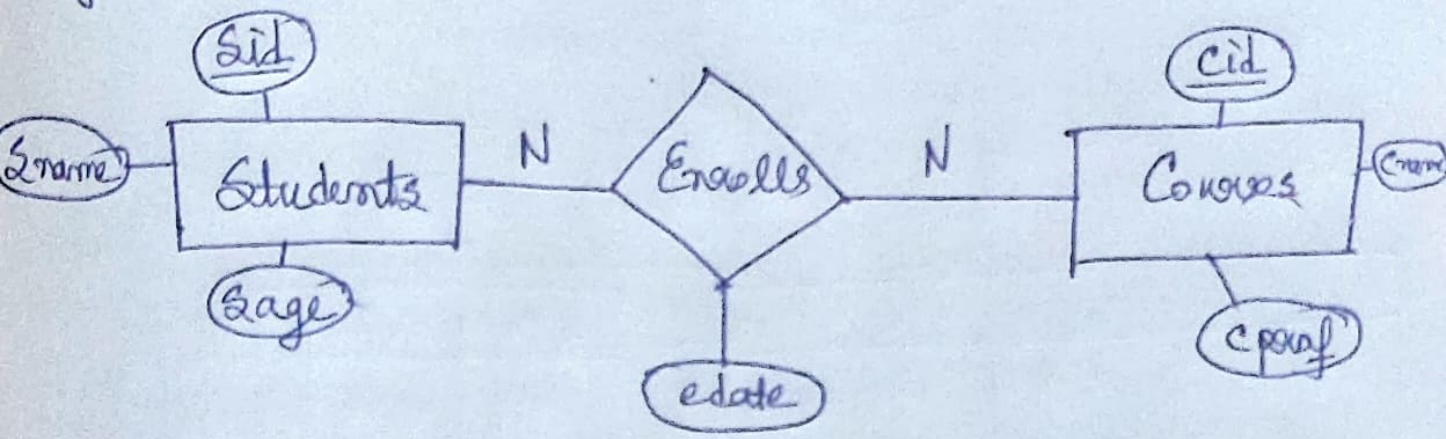


Everything else is the reverse of (1:N) relationship.

## Many to Many Relationship (N:N) ⇒

A database model where multiple entities (records) on one side of the relationship are associated with multiple entities on the other side.

Eg:




⇓ Relational Model

| sid | Sname | Sage |
|---|---|---|
| S1 | ram | 14 |
| S2 | raj | 15 |
| S3 | riti | 16 |

P.K Student

| sid | cid | edate |
|---|---|---|
| S1 | C1 | jan |
| S2 | C2 | feb |
| S3 | C3 | mar |

F.K F.K Enroll

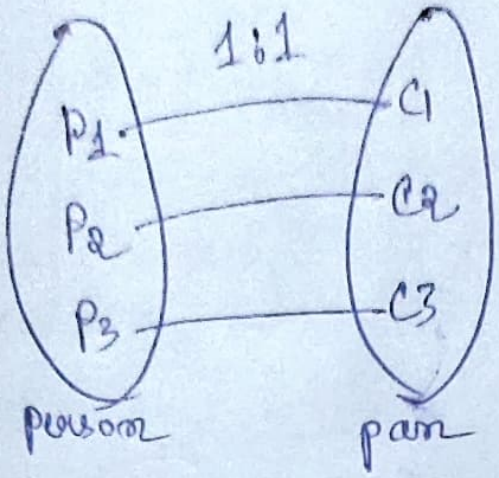
| cid | Cname | Cprof |
|---|---|---|
| C1 | phy | Sanjay |
| C2 | math | Sanjay |
| C3 | bio | Sunit |

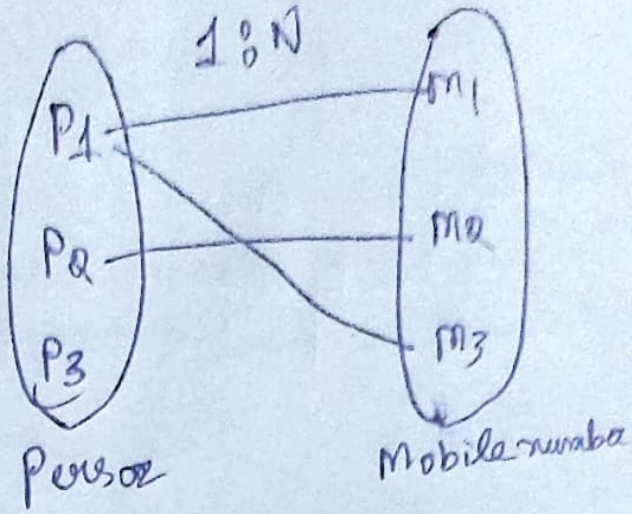
P.K Course

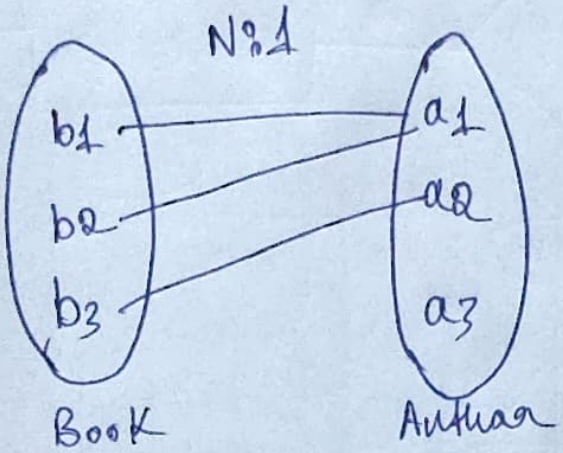
composite Primary Key

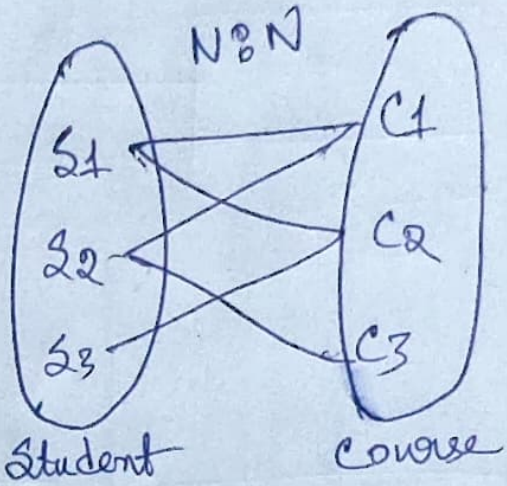
⇓

The Tables can't be reduced further



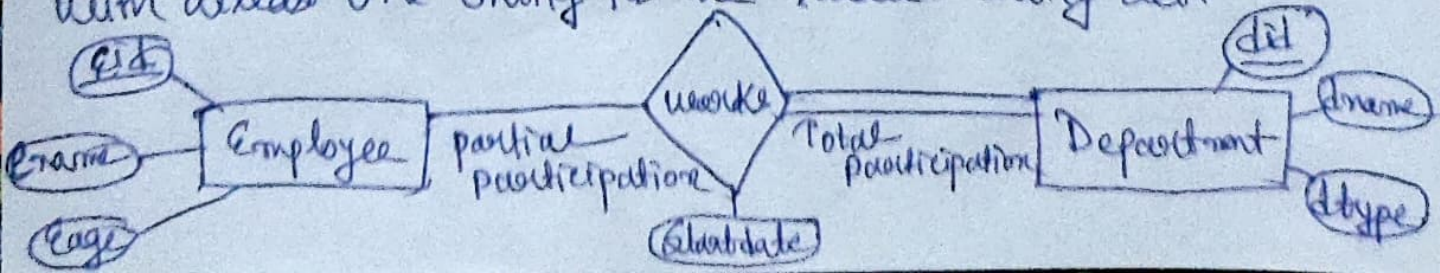




<u>Participation Constraints</u> ⇒ Participation constraints in an ER model define whether every entity in one group must be connected with at least one entity in another group or if the connection is optional.

1) <u>Total Participation (Mandatory)</u> ⇒ In total participation, each entity in a participation set must be associated with atleast one entity in the related entity set.



2) Partial Participation (Optional) → In a partial participation constraint, entities in the participating entity set may or may not be associated with entities in the related entity set.

## Extended ER Features

Why do we need?

We design ER model for relationship between entities. In real-world the data may exhibit some hierarchical relationships, and the EER model provides the mechanisms to represent these relationships accurately which helps in code reusability; ensuring data integrity and consistency and lower the complexity.

Extended ER Features

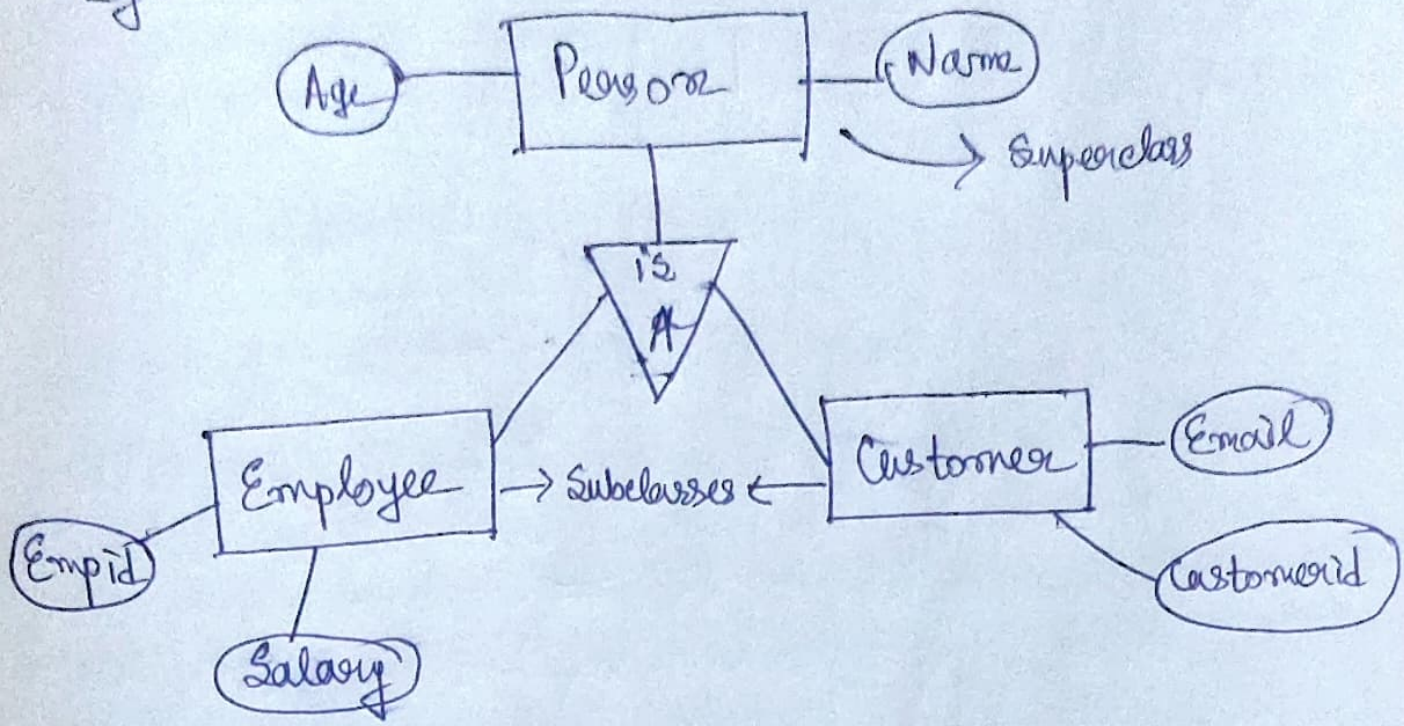
- Specialization
- Aggregation
- Generalization

Specialization → Specialization in the ER model is like categorizing entities based on common features.

A "Supertype" groups entities with shared attributes and relationships, while "subtypes" have their own unique attributes and relationships. It's a way to organize data efficiently. It is a Top-Down Approach.

We have is-a relationship between Superclass and Subclass.

Eg:

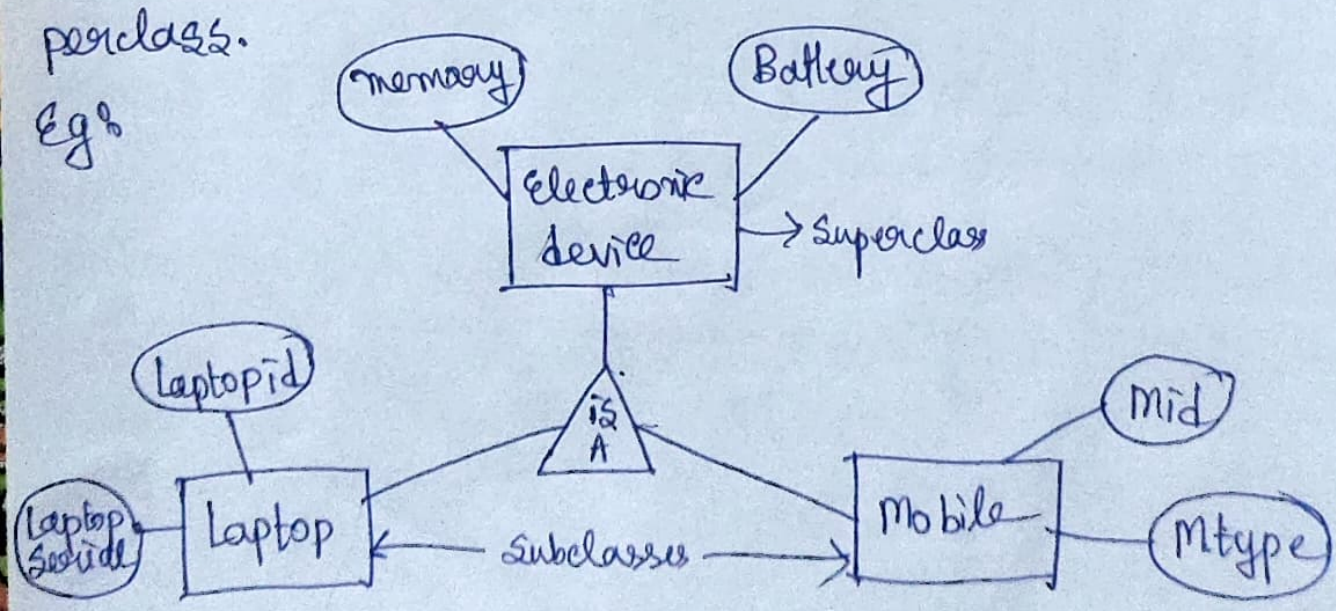



Generalization → Generalization is like finding things that are alike and putting them into a big group to represent what they have in common. It helps make things simplier and organized.

It is a Bottom-Up approach.

We have is-a relationship between Subclass and Superclass.

Eg:



# Inheritance

**Attribute Inheritance**

↳ Both Specialization & Generalization exhibits attribute inheritance

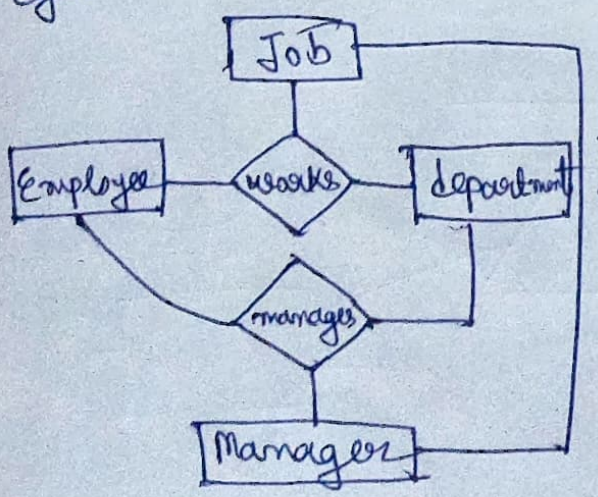
**Participation Inheritance**

↳ if parent is participating in a relationship, then (=>), child will also participate in that relationship.

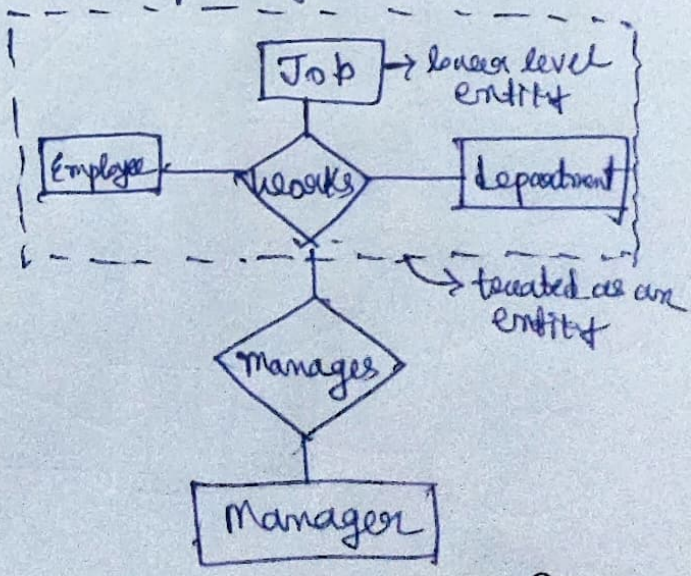
**Aggregation** → Aggregation is like stacking things on top of each other to create a structure. It is used to create a hierarchical structure in data modelling, showing how a higher-level entity is composed of lower-level entities.

Abstraction is employed to view relationships from a more general perspective focusing on a higher-level entity.

Eg:




=> Aggregating



## Steps to draw an ER model

1) Recognize entities.
2) Specify entity characteristics / attributes.
3) Discover connections / relationships (also con-straints like mapping / participation).
4) Define the connection type (how entities connect) / cardinality.
5) Construct an ERD (Entity-Relationship Diagram).
6) Annotate relationships and Attributes.
7) Review and refine the model.
8) Document the model.
9) Validate with stakeholders.
10) Implement the database schema.

## ER Model of Instagram

Lets start with what is instagram?

→ Instagram is a social media platform that allows users to share photos and videos.

Things we can do on instagram are →

1) Create our profile
2) Add profile picture and details
3) Connect with friends
4) Upload a post
5) Like and comment on post
6) Share stories

⋮

Step-1 → Recognize entities sets.

Entities ⇒ userProfile
userFriends
userPost
userLogin
user Likes

Step-2 → Specify entity characteristic / attributes.

Attributes ⇒ 1) UserProfile (user ID, userName, email, profile pic)

user ID - primary Key
email - single valued attribute
profile pic - single valued attribute
userName - composite attribute
dob - stored attribute
age - derived attribute

2) User Friends (follower ID, followerName, user ID)

follower ID - primary Key
follower Name - single valued attribute
user ID - single valued attribute

3) userPost (post ID, caption, image, video, likes count, timestamp)

Post ID - primary Key
Caption - single valued attribute
image - multi valued attribute
video - multi valued attribute
likes count - single valued attribute
timestamp - single valued attribute

4) UserLogin (loginID, loginUserName, loginPassword)

loginID - Primary Key

loginUserName - single valued attribute

loginPassword - multi valued attribute

5) userLikes (postID, userID)

PostID - primary Key

userID - Single valued attribute

Step-3 ⇒ Discover connections/relationships (also constraints like mapping/participation).

1) UserProfile have userFriends (n:n).

2) UserProfile have userPost (1:n), userPost will always be associated to a userProfile therefore total participation.

3) UserProfile has userLogin (1:1).

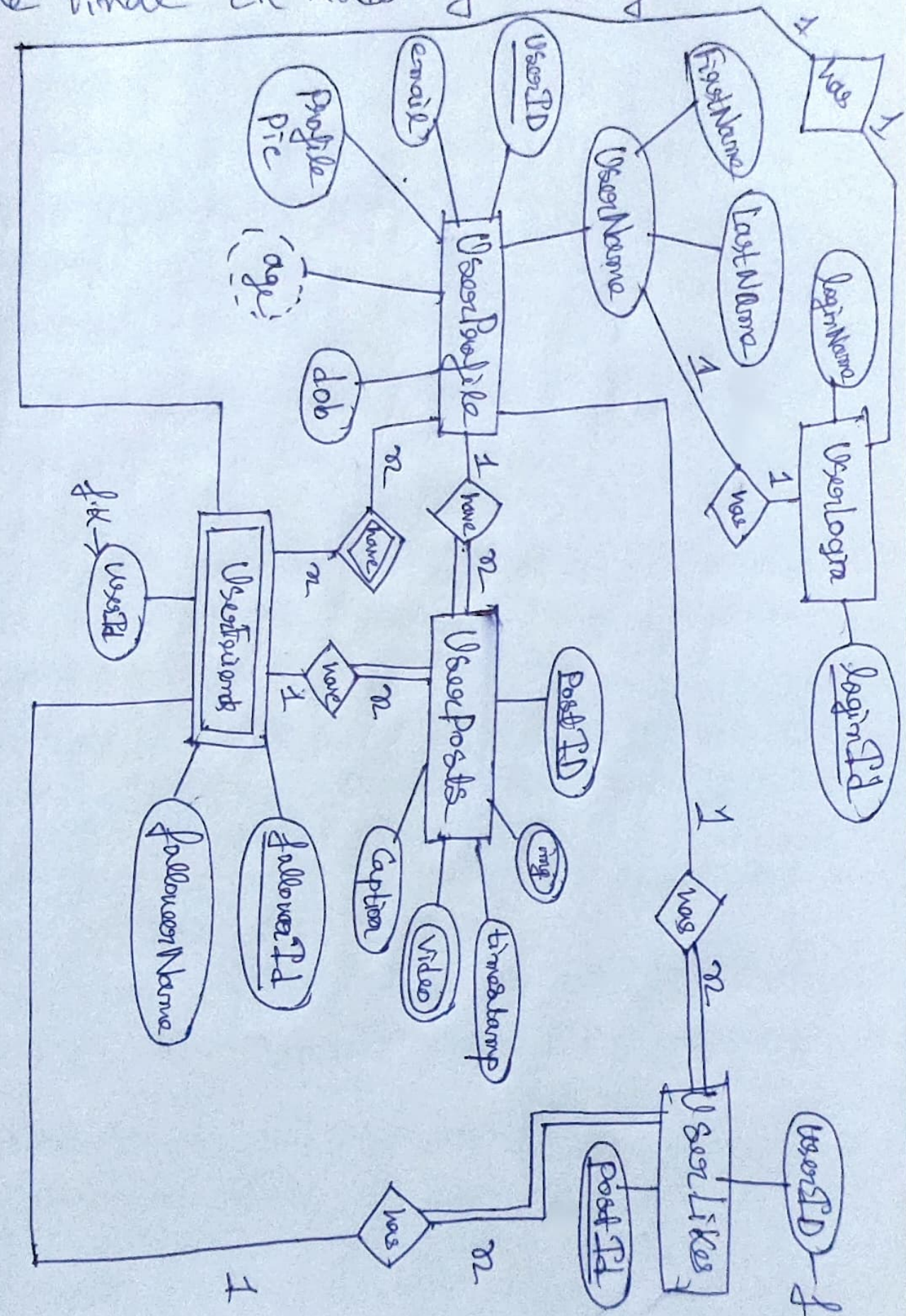
4) UserProfile has userLikes (1:n) userLikes will always be associated to a UserProfile. therefore total participation.

5) UserFriends have userPost (1:n), userPost will always be associated to a UserProfile therefore total participation.

6) UserFriends has userLogin (1:1).

7) UserFriends has user Likes (1:n), userLikes will always be associated to a UserProfile therefore total participation.

∴ The Final ER Model of Instagram will be:


UserProfile
UserID
email
Profile Pic
age
dob
UserName
FirstName
LastName
Has
1
1
1

**Relational Model** ⇒ It is a way of organizing data into tables.

Some terms used in relational model ⇒

1) Table - Relation.
2) Row - Tuple.
3) Column - Attribute.
4) Record - Each row in a table.
5) Domain - The type of value an attribute can hold.
6) Degree - No. of columns in a relation.
7) Cardinality - No. of tuples.

Relational model is all about:

1) Data being organized into tables.
2) Establishing Relationships between tables using Foreign Key.
3) Maintaining data Integrity.
4) A flexible and efficient way to store (SQL) and retrieve data.

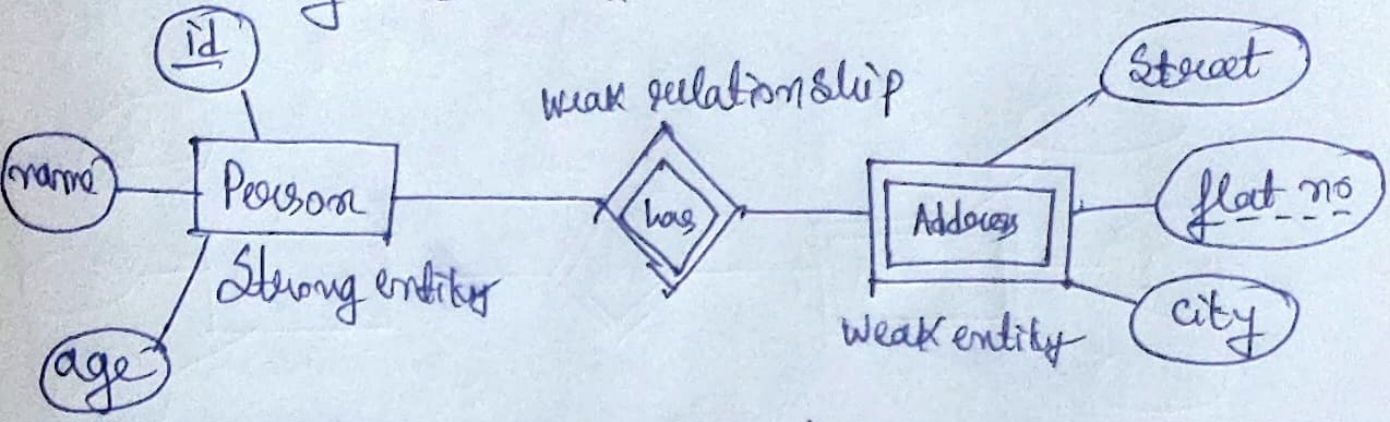
In relational model we take care of different things like:

1) Maintaining integrity constraints like domain, entity, referential integrity.
2) The values to be atomic. i.e, can't be divided further.
3) Each row must be unique, here Keys comes into picture i.e candidate, super, primary etc.

## Convert an ER Model to Relational Model →

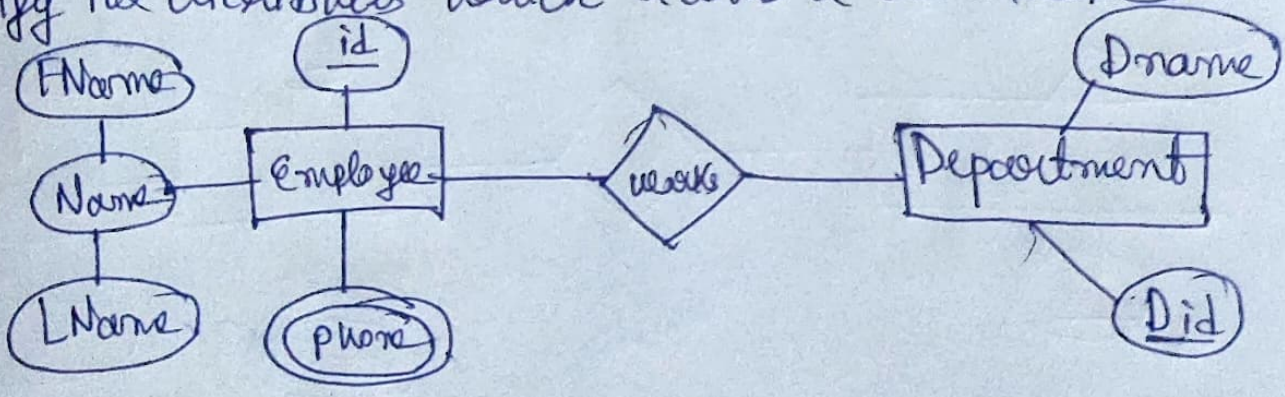
Converting an Entity-Relationship (ER) model to a relational model involves several steps.

**Step-1 → Identify the entities** → List down all the entities like strong and weak.




1) Person (id, name, age) → id (P.K)
2) Address (id, flatno, street, city) – (id + flatno) (P.K)
id (f.K)

**Step-2 → Identify the attributes** → For each entity, identify its attributes which become a column in the table.




**For Composite Attributes →**

1) Employee → (id (P.K), FName, LName)

**For Multivalued Attributes →**

1) Employee-phone → (id (P.K), Phone)
→ new table

Step-3 → Key Selection → Choose the primary Key for each table, for some it can be in form of composite Key (weak entity).

Step-4 → If entities have relationship break it down and reduce tables if possible →

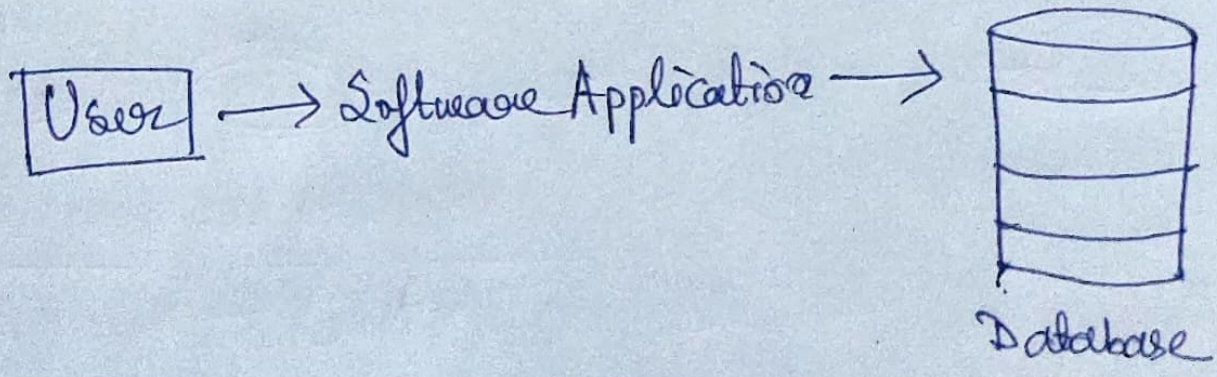
1) 1:1 Relationship : 2 tables, P.K can lie on any side.
2) 1: Many Relationship : 2 tables, P.K can lie on many side.
3) Many : 1 Relationship : 2 tables, P.K can lie on many side.
4) Many : Many Relationship : 3 tables, P.K lie in the relation table having P.Ks of both the table acting as F.K.

## RDBMS

What is RDBMS?

Database : Collection of data is called as database.

DBMS : A software application to manage over data.

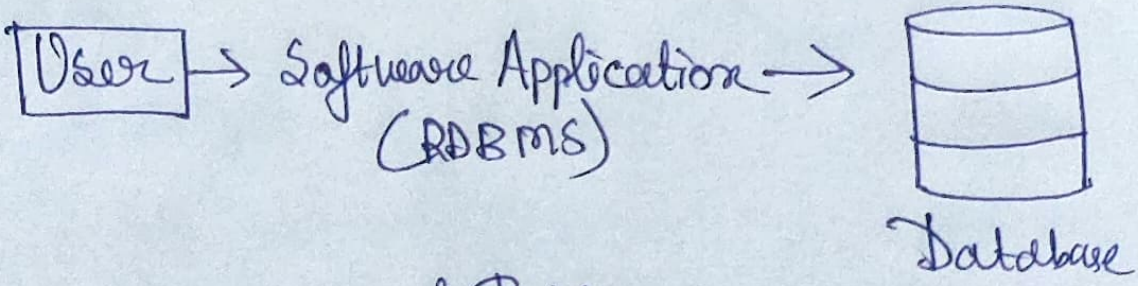


# Database

**Relational** (Use tables to store data)
- MySQL
- Oracle
- MariaDB

**Non-relational** (Data is not stored in tables)
- MongoDB

RDBMS structure data into organized tables that have predefined connections between them.
Data manipulation and querying are performed using SQL (Structured Query Language).




## Essential Components of Tables

**Row / Tuple** → Rows, also known as records or tuples, represent individual entries or instances of data within the table.

**Cardinality** → No. of rows in a table.

**Column / Attribute** → Columns represent the attributes of the data being stored and are named to describe the information they hold. (e.g., "ID", "Name", "Age").

**Degree** → No. of columns in a table.

→ primary Key

Rows / Tuples

| ID | Name | Place |
|---|---|---|
| 1 | Rahul | Delhi |
| 2 | Raj | Kolkata |
| 3 | Riti | Mumbai |

Columns / Attributes

**Constraints →** Constraints define rules or conditions that must be satisfied by the data in the table. Common constraints include uniqueness, nullability, default values etc.

1) **Unique Constraint →** Ensures values in a column are unique across the table.

2) **Not Null Constraint →** ~~Ens~~ Ensures a column cannot have a null value.

3) **Check Constraint →** Enforces a condition to be true for each row.

4) **Default Constraint →** Provides a default value for a column if no value is specified.

**Keys →** A primary Key is a unique identifier for each record in the table. It ensures that a row can be uniquely identified and accessed within the table.

A foreign Key is a field in a table that refers to the primary Key of another table. It establishes relationships between tables.

# Intension and Extension in DataBase

**Intension in Database** → The intension defines what kind of data can be stored and the relationships between them. This is basically the blueprint or definition of the database structure. It does not change frequently and its the permanent definition of the database structure.

It includes:

1) Table definitions (name of tables, their columns, and the data types allowed in each column)

2) Constraints (Rules that govern the data, such as primary keys, foreign keys and data validation rules)

3) Relationship between tables (how tables are connected through shared attributes / columns)

Example →

```
customer(
id INT PRIMARY KEY,
name VARCHAR(50))
```

**Extension in Database** → The extension is the actual data stored in the database at a given instance in time. Basically the data which is stored in tuples/rows at a given instance of time. When there are more tuples added the data can change.

Employee

| id | name | department |
|---|---|---|
| 1 | Rahul | IT |
| 2 | Afsana | HR |
| 3 | Abhimanyu | IT |

Data at instance t1

Employee

| id | name | department |
|---|---|---|
| 1 | Rahul | IT |
| 2 | Afsana | HR |
| 3 | Abhimanyu | IT |
| 4 | Aditya | Marketing |

Data at instance t2

Employee

| id | name | department |
|---|---|---|
| 1 | Rahul | IT |
| 2 | Afsana | HR |
| 3 | Abhimanyu | IT |
| 4 | Aditya | Marketing |
| 5 | Raj | IT |

Data at instance t3

Keys in DBMS → Keys in DBMS make sure of data integrity, uniqueness, and the quick retrieval of information. Key is a attribute in table.

Types of Keys:

1) Candidate Key
2) Primary Key
3) Foreign Key
4) Super Key

Candidate Key → A candidate Key refers to a group of attributes capable of uniquely identifying a record within a table. Among these, one is selected to serve as the primary Key.

Ex: For Student possible attributes for candidate Key could be

Student <ID, Roll No, Aadhar Card>.

Primary Key → A primary Key is a Key which uniquely identifies each record in a table. It ensures that each tuple or record can be uniquely identified within the table. It is always Unique + Not Null

Ex: Student <ID, Registration Number>

Foreign Key → A foreign Key is a field in a table that refers to the primary Key in another table. It establishes a relationship between two tables

Ex:

Student (Base / referenced table)

| Roll No | Name | Hometown |
|---|---|---|
| | | |

Subject (referencing table)

| Roll No | Name | Subject |
|---|---|---|
| | | |

Primary Key ← references — Foreign Key

Referenced table → Table having primary Key (P.K)

Referencing table → Table having foreign Key (F.K)

Referential Integrity in Foreign Key → Referential integrity is an important concept in foreign Key. We always say foreign Key maintains referential integrity.

Referential integrity ensures that the relationships between tables remain accurate, consistent, and meaningful within a relation database.

Now consider there are 2 tables one is referencing and other is referenced table.

Lets see how some operations like insert, ~~upta~~ update and delete works here.

1) Insertion in Referenced / Base table → No violation

2) Deletion in Referenced / Base table → May cause violation if the corresponding data is present in referencing table.

So, if a record in referenced table is deleted or updated, the corresponding records in the referencing table

should be deleted or updated to maintain the integrity of the relationship.

We can solve the problem by using action like "CASCADE DELETE" for the same. Also we can set null for the values deleted.

3) Updation in Referenced / Base table → May cause violation if the corresponding data is present in the referencing table. We can solve the problem by using action like "CASCADE UPDATE".

4) Insertion in Referencing Table →

May cause violation.

Eg:

| Rollno | Name | phoneno |
|---|---|---|
| 1 | Raj | 0012 |
| 2 | Riti | 0619 |
| 3 | Rahul | 0674 |

Student (Referenced table)

P.K

| Sid | Sname | Rollno |
|---|---|---|
| 1 | Phy | 1 |
| 2 | Chem | 2 |
| 3 | Maths | 3 |
| 4 | Biology | 4 |

Subject (Referenced table)

F.K

P.K

references

Insertion of the row with roll no 4 causes violation as it is not present in referenced table.

5) Deletion in Referencing Table → No violation

6) Updation in Referencing table → No issues untill we are updating foreign Key attribute. Violation would be caused on updating.

Eg:

Student (Referenced table)

| Roll no | Name | Phone no |
|---|---|---|
| 1 | Rahul | 002 |
| 2 | Raj | 001 |
| 3 | Riti | 003 |

Subject (Referencing table)

| Sid | Sname | Rollno |
|---|---|---|
| 01 | Phy | 1 |
| 02 | Bio | 2 |
| | | |

P.K

references

P.K

F.K

→ updation of the row to roll no 10 will cause violation, as there is no such record with roll no 10 in the referenced table.

## Integrity Constraints in DBMS →

Integrity constraints help to ensure that data remains reliable and meaningful throughout its lifecycle.

### Type of Integrity Constraints →

1) Domain Integrity Constraint
2) Entity Integrity Constraint
3) Referential Integrity constraint

4) Key Constraint
5) Check Constraint
6) Null Constraint
7) Unique Constraint
8) Default Constraint

1) Domain Integrity Constraint → It ensures the validity and appropriateness of data values (i.e, valid data types, ranges, and formats for columns) within a specific column or attribute of a table.
Ex: check for date column so that it contains valid date values

2) Entity Integrity Constraint → It ensures that each identified row/record in a table is uniquely by a primary key.
It also helps in preventing duplicate or null values in the primary key.

3) Referential Integrity Constraint → It ensures that values in a foreign key column match with the values in the corresponding primary key column in another table.

4) Key Constraint → It ensures uniqueness for the primary key.

5) Check Constraint → It checks for a condition that each row in a table must satisfy.

If the condition is not met, the insertion or update of the row is rejected.

6) <u>Null Constraint</u> → It determines whether a column in a table can have null (i.e, missing or unknown) values or not.

7) <u>Unique Constraint</u> → It ensures that values in a specified column or combination of columns are unique across a table.

This constraint prevents the duplicate values from being inserted into the specified column(s), maintaining data consistency and integrity.

8) <u>Default Constraint</u> → It ensures a default value for a column, which is used if no other value is provided.

<u>Super Key</u> → It is a set of one or more attributes (columns) that can uniquely identify a tuple (a row) in a relation (table).

Superset of any candidate Key.

A super key becomes a candidate key if it is minimal (i.e, no proper subset of it can uniquely identify a tuple).

Ex: Find the total possible combinations of super key with 2 attributes, with E1 as the candidate key attribute

Employee → $\{E_1, E_2, E_3, \ldots, E_n\}$

$\Rightarrow 2^{(n-1)}$

# Normalization and its types

**Normalization** → Normalization is a process in which we organize data to reduce redundancy (duplicacy) and improve data consistency. It involves dividing a database into two or more tables.

**What is data redundancy and consistency and why is it important** → when there is same set of data repeated each and every time it results in duplicacy of data (either for row or column).

Now row level duplicacy can be removed by using primary key for unique values.

Now when we have same data for some set of columns it leads to different anomalies (inconsistencies / errors that occur when manipulating or querying data in a database).

1) Insertion Anomaly.
2) Updation Anomaly.
3) Deletion Anomaly

Also, it increases the size of the database with the same data.

## Insertion Anomaly:

It occurs when it is difficult to insert data into the database due to the absence of other required data.

Consider you want to add a new department but there is no employee in that dept yet.

| id | name | age | department | Manager | Salary |
|---|---|---|---|---|---|
| 1 | Rahul | 25 | IT | Raj | 1600 |
| 2 | Afsana | 26 | HR | Avinash | 1000 |
| 3 | Abhimanyu | 27 | IT | Raj | 1500 |
| 4 | Aditya | 25 | HR | Avinash | 1000 |
| 5 | Raj | 04 | HR | Avinash | 1000 |

## Deletion Anomaly →

It occurs when deleting data removes other valuable data.

Consider if you delete all the record in a table, you will loose the track of dept., their manager and salaries.

| id | name | age | department | Manager | Salary |
|---|---|---|---|---|---|
| 1 | Rahul | 25 | IT | Raj | 1500 |
| 2 | Afsana | 26 | HR | Avinash | 1000 |
| 3 | Abhinanyu | 27 | IT | Raj | 1500 |
| 4 | Aditya | 29 | HR | Avinash | 1000 |
| 5 | Raj | 24 | HR | Avinash | 1000 |

## Updatation Anomaly →

It occurs when changes to data require multiple updates.

Consider you want to change the Salary for people working in HR department, you need to update it at 3 places.

| id | name | age | department | Manager | Salary |
|---|---|---|---|---|---|
| 1 | Rahul | 25 | IT | Raj | 1500 |
| 2 | Afsana | 26 | HR | Avinash | 1000 |
| 3 | Abhinanyu | 27 | IT | Raj | 1500 |
| 4 | Aditya | 25 | HR | Avinash | 1000 |
| 5 | Raj | 24 | HR | Avinash | 1000 |

## How normalization help us here →

Using normalization we can divide the employee table in two tables.

1) Employee
2) Department

Employee

| id | name | age | department |
|---|---|---|---|
| 1 | Rahul | 25 | IT |
| 2 | Afsana | 26 | HR |
| 3 | Abhinanyu | 27 | IT |
| 4 | Aditya | 29 | HR |
| 5 | Raj | 24 | HR |

Department

| department | Manager | Salary |
|---|---|---|
| IT | Raj | 1500 |
| HR | Avinash | 1000 |

## Types of Normalization →

1) First Normal Form (1NF)
2) Second Normal Form (2NF)
3) Third Normal Form (3NF)
4) Boyce-Codd Normal Form (BCNF)

## Denormalization

This is the opposite of normalization. It involves intentionally introducing some redundancy into a well-normalized database schema to improve query performance.

Consider if you wish to find the salary of Rahul, so first you have to make a query in employee table to find department of Rahul and then in department table to find the salary of Rahul

Employee

| id | name | age | department |
|---|---|---|---|
| 1 | Rahul | 29 | IT |
| 2 | Afsana | 26 | HR |
| 3 | Abhimanyu | 27 | IT |
| 4 | Aditya | 25 | HR |
| 5 | Raj | 24 | HR |

(+)

Employee

| id | name | age | department | Manager | Salary |
|---|---|---|---|---|---|
| 1 | Rahul | 29 | IT | Raj | 1500 |
| 2 | Afsana | 26 | HR | Avinash | 1000 |
| 3 | Abhimanyu | 27 | IT | Raj | 1500 |
| 4 | Aditya | 25 | HR | Avinash | 1000 |
| 5 | Raj | 24 | HR | Avinash | 1000 |

=

Department

| department | Manager | Salary |
|---|---|---|
| IT | Raj | 1500 |
| HR | Avinash | 1000 |

## Advantages →

1) Faster Queries → It can reduce the need for complex joins between tables during queries which can eventually improve the speed of retrieving frequently accessed data.

2) Simplier Queries> It can simplify queries by allowing them to be executed on a single table instead of requiring joins across multiple tables.

Disadvantages>

1) Increased Data Redundancy.

2) Less Data Consistency.

3) Denormalization can make the database schema less flexible for ~~feur~~ further changes like adding/modifying new data elements.

## Functional Dependency

Functional dependency describes the relationship between attributes in a relation.

A FD is a constraint between two sets of attributes in a relation from a database.

For a Relation (table) R, if there are two attributes X and Y then,

FD : X (determinant) → Y (dependent)

Attribute Y is functionally dependent on attribute X.

if X = 1, we can find the value of y.

F.D : X → Y (X, Y is a subset of R)

| Emp ID | Emp First Name | Emp last Name |
|---|---|---|
| 1 | Riti | Kumari |
| 2 | Rahul | Kumar |
| 3 | Suraj | Singh |

what is subset?
Let A = {1, 2, 3}
Let B = {1, 2, 3, 4, 5}
A is a subset of B because every element of A is also an element of B.

FD: EmpID → EmpFirstName (EmpFirstName is functionally dependent on EmpID), EmpID → EmplastName.

If X → Y and X(tuple 1) = X(tuple 2), then Y(tuple 1) = Y(tuple 2)

## Properties of Functional Dependencies →

1) Reflexivity → If Y is a subset of X, then X → Y. (X → X).
2) Augmentation → If X → Y, then XZ → YZ for any Z.
3) Transitivity → If X → Y and Y → Z, then X → Z.
4) Union → If X → Y and X → Z, then X → YZ.
5) Decomposition → If X → YZ, then X → Y and X → Z.

## Types of Functional Dependency →

1) Trivial dependency
2) Non-trivial dependency.

1) Trivial Dependency → A functional dependency X → Y is trivial if Y is a subset of X. We can also say it as X → X.

{EmpID, EmpFirstName} → {EmpID} is trivial because {EmpID} is a subset of {EmpID, EmpFirstName}.

$X \cap Y = Y$

| EmpID | EmpFirstName | EmplastName |
|---|---|---|
| 1 | Riti | Kumari |
| 2 | Rahul | Kumar |
| 3 | Suraj | Singh |

**Non-Trivial Dependency** → A functional dependency $X \to Y$ is non-trivial if $Y$ is not a subset of $X$.

{EmpID} → {EmpFirstName} is not trivial because {EmpFirstName} is not a subset of {EmpID}.

$X \cap Y = \text{Empty}$

| EmpID | EmpFirstName | EmpLastName |
|---|---|---|
| 1 | Riti | Kumari |
| 2 | Rahul | Kumar |
| 3 | Suraj | Singh |

## Attribute Closure / Closure Set

Attribute closure helps us for identifying candidate keys, checking for functional dependencies, and in normalization

$X^+$, where $X$ is an attribute or set of attributes which have all the attributes in a relation which can determine $X$.

**Q** Consider we have a relation R with attributes A, B, C, D, E and F.D are $A \to B$, $B \to C$, $C \to D$, $D \to E$.

1) Now according to the rule of Reflexivity all the attributes can determine themselves. $A \to A$, $B \to B$, $C \to C$, $D \to D$, $E \to E$.

2) Now according to the rule of Transitivity, if A determines B and B determines C, then A can also determine C and same for all other attributes.

$A \to C$, $A \to D$, $A \to E$
$B \to D$, $B \to E$
$C \to E$

3) Now according to the rule of UNION if as the determinant is same we can combine dependents.

For A

$A \to B, A \to C, A \to D, A \to E, A \to A$

$\therefore A \to ABCDE$

For C

$C \to D, C \to E, C \to C$

$\therefore C \to DEC$

For E

$E \to E$

For B

$B \to C, B \to D, B \to E, B \to B$

$\therefore B \to CDEB$

For D

$D \to E, D \to D$

$\therefore D \to ED$

4) Now lets find the closure ~~set~~ set of attributes.

A - {A, B, C, D, E}

B - {B, C, D, E}

C - {C, D, E}

D - {D, E}

E - {E}

AB - {A, B, C, D, E}

Now lets find candidate Key, super Key, prime and non-prime attributes.

Super Key → Set of attributes whose closure contains all the attributes given in a relation. Super set of any candidate Key is super Key. A Key (combination of all possible attributes) which can uniquely identify two tuples.

How to find Super Key?

i) Identify All Attributes

ii) Analyze the functional dependencies and find closure.

iii) Generate Power Set → If A has n attributes, the power set will have $2^n$ subsets.

iv) Check for Super Key Property → For each subset in the Power set, check if the subset identify each tuple in the relation.

Q Find all the superKeys for the relation R (A, B, C) and

F.D: $A \to B$, $B \to C$.

→ i) Identify all attributes.

$At = \{A, B, C\}$

Q) Analyze the functional dependencies and find closure.

From $A \to B$ and $B \to C$, we can infer $A \to C$

$A+ = \{A, B, C\}$

$B+ = \{B, C\}$, because $B \to C$, but B does not determine A.

$C+ = \{C\}$

closure of A gives or determines all the attribute in the table, so we can say its a super Key

3) How to find all the super Keys?

→ Find the power subset.

The power set of $\{A, B, C\}$ is $2^3 = 8$ i.e,

$\{\}, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{C, A\}, \{A, B, C\}$

A power set is the set of all subsets of a given set, including the empty set and the set itself. If you have a set X, the the power set of X has $\boxed{2^X \text{ elements}}$

Let's take a simple set $X = \{a, b\}$

The power set of X would include the following subsets:

1. The empty set: $\phi$
2. The single-element subsets: $\{a\}$ & $\{b\}$
3. The full set itself: $\{a,b\}$

So, the power set $P(X)$ would be, $P(X) = \{\phi, \{a\}, \{b\}, \{a,b\}\}$

4) Verify each subset to see if it is a super key.

a) $\{\}$: Not a super key.
b) $\{A\}$: Super key (as its closure determines all the attributes in relation).
c) $\{B\}$: Not a super key (does not determine A).
d) $\{C\}$: Not a super key (does not determine A, B).
e) $\{A,B\}$: Super key (A is already a super key, so adding B still keeps it a super key).
f) $\{A,C\}$: Similarly it is also a super key.
g) $\{B,C\}$: Not a super key (B determines C, but does not determine A).
h) $\{A,B,C\}$: Super key (A is already a super key, so adding B and C keeps it a super key).

5) The super keys are: $\{A\}, \{A,B\}, \{A,C\}, \{A,B,C\}$.

We can also say to find the maximum number of superkeys we can use the formula.

$$\boxed{2^{(n-1)}, \text{ where, } K = \text{candidate key with } K \text{ attributes } (K<n) \text{ in a relation.} \quad n = \text{total no. of attributes in a relation}}$$

when, 1 C.K is there = $2^{(n-1)}$

when, 2 C.K is there = $2^{(n-1)}$ (for 1st C.K) + $2^{(n-1)}$ (for 2nd C.K) - $2^{(n-2)}$ (for combination of both).

<u>Candidate Key</u> → A SuperKey whose proper subset is not a Super Key.

A set A is considered a proper subset of set B.

a) All elements of A are also elements of B

b) Set B has at least one element that is NOT in A.

<u>Prime Attribute</u> → An attribute that is part of any candidate Key.

<u>Non-Prime Attribute</u> → An attribute that is not part of any candidate Key.

How to find the number of candidate Keys?

i) Take the closure of the entire attribute set.

ii) Discard the dependents if their determinants are present.

iii) After discarding the final combination you get is a candidate Key if its subset doesnot have a super Key and mention them in prime attribute.

iv) Now check all the functional dependencies and see if in the right hand side is there a attribute which has a determinant present in the prime attribute, if yes mention that as well in the prime attribute and replace the dependent attribute with determinant and check if its a candidate key.

v) If no prime attribute is available in the right hand side of the functional dependency we can say there is no

more candidate key and the one key we made after discarding dependent and checking for super key, that is the only candidate key.

Q Find the no. of candidate and super keys for the given relation. R (A, B, C, D) and F.D: $A \to B$, $B \to C$, $C \to D$, $B \to A$.

→ 1) all the attributes are → {A, B, C, D}

2) Analyze the functional dependencies →

$A \to B$, $B \to C$, $C \to D$, $B \to A$

$A \to C$, $A \to D$, $B \to D$, $A \to A$, $B \to B$, $C \to C$, $D \to D$

3) $A^+ = \{A, B, C, D\}$ } Super key.

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

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

$D^+ = \{D\}$

4) Power Set = {{}, {A}, {B}, {C}, {D}, {A, B}, {A, C}, {A, D}, {B, C}, {B, D}, {C, D}, {A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}, {A, B, C, D}}

∴ The superkeys are →

{A, B, AB, AC, AD, BC, BD, ABC, ABD, ACD, BCD, ABCD}

= 10 superkeys.

Now finding candidate keys →

1) taking closure of all attributes →

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

2) Discard dependents if determinants are present.

~~AB~~CD [∵ $A \to B$]

A~~C~~D [∵ $C \to D$]

A~~C~~ [∵ $A \to C$]

3) A is a candidate key if its subset is not a super key and then mentioning it in the prime attribute.

as $\{\}^+ \to \phi$ ∴ A is a CK and P.A = {A}

4) If A is present on RHS of any F.D, replace A with its determinant and repeat step 3.

PA = {A, B} [∵ B → A]

So checking if B is a C.K

none $\{\}^+ = \phi$ ∴ B is a C.K.

5) Repeating (4) on B, as there are no more P.A available in the RHS of the functional dependency, hence there are no more candidate keys..

∴ C.K = {A, B} ∴ there are 2 candidate keys.

Now, checking the number of SuperKeys from the number of candidate keys by using the formula $2^{(n-K)}$

$$\text{No of S.K} = 2^{(4-1)} + 2^{(4-1)} - 2^{(4-2)} = 8+8-4 = 12.$$

## First Normal Form (1NF) →

First Normal Form is the first step in the normalisation process which helps us to reduce data redundancy. Every table should have atomic values i.e, there shouldn't be any multivalued attributes.

It ensures the following set of rules is followed in the table →

a) Atomicity → Attributes should have single value.

b) Uniqueness of rows → Each row should be uniquely identifiable.

a) Atomicity → Each column contains only indivisible (atomic) values, meaning each attribute holds a single value.

| ID | PersonName | Order |
|---|---|---|
| 1 | Raj | Muffin, Sugar |
| 2 | Riti | Muffin |
| 3 | Rahul | Sugar, Egg |

Here, Order is a multivalued attribute (having more than one value), hence it is not in 1NF.

How to achieve atomicity?

→ 1) Repeat the values in id and PersonName columns twice to store a single value of multivalued attribute order.

| ID | PersonName | Order |
|---|---|---|
| 1 | Raj | Muffin |
| 1 | Raj | Sugar |
| 2 | Riti | Muffin |
| 3 | Rahul | Sugar |
| 3 | Rahul | Egg |

P.K = Order + ID

2) Make new columns for each multivalue present.

| ID | PersonName | Order1 | Order2 |
|---|---|---|---|
| 1 | Raj | Muffin | Sugar |
| 2 | Riti | Muffin | Null |
| 3 | Rahul | Sugar | Egg |

P.K = ID

It is not suggested as it introduces null values and no of attributes depends on number of components of the multivalued data.

3) Divide the table into Student (base) and order (referencing) table based on the multivalued attribute order.

P.K ← references — F.K

| ID | PersonName |
|---|---|
| 1 | Raj |
| 2 | Riti |
| 3 | Rahul |

| ID | Order |
|---|---|
| 1 | Muffin |
| 1 | Sugar |
| 2 | Muffin |
| 3 | Sugar |
| 3 | Egg |

P.K = ID + Order

## Second Normal Form (2NF) →

A relation is in 2NF if it satisfies the following conditions:

a) It is in First Normal Form (1NF).

b) It has no partial dependency, which means no non-prime attribute is dependent on a part of any candidate key.

when partial dependency is there in a table?

→ LHS is a proper subset of Candidate Key AND RHS is a non-prime attribute.

<u>Non-Prime</u> → An attribute that is not part of any candidate Key.

Candidate Key: CustomerId + OrderId

Prime Attribute: {CustomerId, OrderId}

Non Prime Attribute: {OrderName}

| CustomerId | OrderId | OrderName |
|---|---|---|
| 1 | 1 | Muffin |
| 2 | 1 | Muffin |
| 1 | 2 | Sugar |
| 4 | 2 | Sugar |

In this relation, OrderName is dependent on OrderId only, according to OrderId we provide the OrderName.

OrderName is determined by only OrderId.

So, there is a partial dependency, hence the table is not in 2NF.

To normalize further →

references

| CustomerId | OrderId |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 1 | 2 |
| 4 | 2 |

P.K = {CustomerId + OrderId}

F.K

| OrderId | OrderName |
|---|---|
| 1 | Muffin |
| 2 | Sugar |

P.K = {OrderId}

Q Consider there is a relation R(A,B,C,D) with F.D:
$AB \to C$, $AB \to D$, $B \to C$. Find if this is in 2NF?

→ i) Indentifying the Candidate Key(s) →

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

So, AB is a candidate key here.

Q) Check for Partial Dependencies.

a) L.H.S is a proper subset of Candidate Key AND
b) R.H.S is a non-prime attribute.

F.D: $AB \to C$, $AB \to D$, $B \to C$
C.K: AB
Prime attribute: $\{A, B\}$
non-Prime attribute: $\{C, D\}$

a) $AB \to C$ (fully dependent)
b) $AB \to D$ (fully dependent)
c) $B \to C$ (partial dependency as B is a proper subset of C.K and C is a non-prime attribute.

Hence, the relation is not in 2NF.

## Third Normal Form (3NF) →

a) It is in Second Normal Form (2NF).
b) It has no transitive dependency, which means no non-prime attribute is transitively dependent on a candidate key.

Transitive dependent → No non-prime attribute should be dependent on another non-prime attribute.

For any functional dependency $X \to Y$, one of the following Conditions must be true to be in 3rd Normal Form.

a) X is a SuperKey or Candidate Key (LHS) OR

b) Y is a prime attribute (i.e, part of Some candidate Key). (RHS).

Q. Consider there is a relation R(A,B,C,D) with F.D: $AB \to C$, $C \to D$. Find if this is in 3NF?

→ 1) Identify the Candidate Key(s)

$A^+ = \{A\}$

$B^+ = \{B\}$

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

$D^+ = \{D\}$

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

So, AB is a candidate Key here.

2) Check for transitive dependency.

→ no non-prime attribute should be dependent on another non-prime attribute.

F.D: $AB \to C$, $C \to D$

C.K: AB

prime attribute: {A, B}

non-prime attribute: {C, D}

a) $AB \to C$ (no transitive as AB is a C.K)

b) $C \to D$ (transitive as C is not a superKey / Candidate Key

and D is a non-prime attribute).

∴ the relation is not in 3NF.

## BCNF (Boyce Codd Normal Form) →

A relation is in BCNF if it satisfies the following conditions:

1) It is in 3rd Normal Form (3NF).

2) For a given FD X → Y, should always have C.K or S.K, and should only determine non-prime attributes.

Q Consider there is a relation R(A,B,C,D) with F.D: AB → C, AB → D. Find if this is in BCNF?

→ 1) Identify the Candidate Key(s).

$A^+ = \{A\}$

$B^+ = \{B\}$

$C^+ = \{C\}$

$D^+ = \{D\}$

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

So, AB is a candidate key here.

2) check for C.K or S.K in LHS

F.D: AB → C, AB → D

C.K: AB

a) AB → C (LHS i.e AB is a C.K)

b) AB → D (LHS i.e AB is a C.K)

∴ The relation is in BCNF

# Lossless and Lossy Decomposition

**Lossy Decomposition** → In normalisation we generally break/decompose the table into 2 or more tables.

Consider there is a relation R,

Lossy decomposition occurs when a relation R is decomposed or broken into 2 or more relations, but data is lost, and the original relation R can't be reconstructed by joining these decomposed relations.

There are 2 rules for a decomposition to be lossy.

a) Some data from the original relation R is lost after decomposition.

b) Join of the decomposed relations (R1, R2, ..., Rn) is not equal to the original relation R.

Eg: There is a relation R(A,B,C).

**Step-1** → Lets decompose the relation based on any attribute and keep that attribute as common, for now lets use B as common attribute. Decomposed relations: R1(A,B) and R2(B,C).

R

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 6 |

→

R1

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

+

R2

| B | C |
|---|---|
| 2 | 3 |
| 2 | 6 |

**Step-2** → Lets perform a natural join between R1 and R2. When we do R1 natural join R2 we won't get the original relation back (lossy).

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 2 | 6 |
| 4 | 2 | 3 |
| 4 | 2 | 6 |

We can see some additional tuples that were not in the original relation R (lossy decomposition).

How to ensure a decomposition is lossless?

a) Divide or decompose the table on basis of C.K or S.K present in the relation so that there is no duplicacy

b) For a decomposition to be lossless

1) $R1 \cup R2 = R$

2) $R1 \cap R2$ = common attribute

c) To ensure that a decomposition is lossless, a common approach is to use the dependency preservation property.

<u>Lossless Decomposition</u> →

Consider there is a relation R,

Lossless decomposition ensures that when a relation R is decomposed / breaked into two or more relations, no data is lost, and the original relation R can be again reconstructed by joining these decomposed relations.

There are 2 rules for a decomposition to be lossless.

a) All data in the original relation R should be preserved after decomposition.

b) Join of the decomposed relations (R1, R2, ..., Rn) = original relation R.

Eg: There is a relation R(A,B,C) with C.K as A

Step-1 → Lets decompose the relation based on the C.K or S.K of the given R. Decomposed relations: R1(A,B) and R2(A,C).

R

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

→

R1

| A | B |
|---|---|
| 1 | 2 |
| 4 | 5 |

+

R2

| A | C |
|---|---|
| 1 | 3 |
| 4 | 6 |

Step-2 → Lets perform a natural join between R1 and R2. When we do R1 natural join R2 we get the original relation back (lossless).

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

R

## Dependency Preserving decomposition

Dependency preserving decomposition ensures that the functional dependencies are preserved / maintained after decomposing a relation into two or more smaller relations

Q/ Consider a relation R(A,B,C) with F.D: A → B, B → C, find if its dependency preserving when divided into R1(AB)

and R2 (BC).

→ 1) R1 (AB): A → B

2) R2 (BC): B → C

The decomposition is dependency preserving because the functional dependencies A → B and B → C are preserved in R1 and R2.

Q Consider a relation R(A, B, C, D) with F.D: A → B, A → C, C → D, find if its dependency preserving when divided into R1(A, B, C) and R2(C, D).

→ 1) R1(A, B, C): A → B, A → C (F.D preserved)

2) R2 (C, D): C → D (F.D preserved)

The decomposition is dependency preserving because the functional dependencies A → B and B → C are preserved in R1 and R2.

## 4NF (Fourth Normal Form) →

A relation is in 4NF if it satisfies the following conditions:

1) It is in BCNF.

2) It has no multi-valued dependencies.

## Multivalued dependency →

A multi-valued dependency X →→ Y, X →→ Z in a relation R(X, Y, Z) implies that for each value of X, there is a set of values for Y and a set of values for Z that are independent of each other.

Eg: Student (StudentID, Course, PhoneNbr)

1) StudentID →→ Course (A student can take multiple courses).

2) StudentID →→ PhoneNbr (A student can have multiple hobbies).

| StudentID | Course | PhoneNbr |
|---|---|---|
| 1 | math | 123 |
| 1 | Sci | 123 |
| 1 | math | 345 |
| 1 | Sci | 345 |
| 2 | Hin | 678 |
| 2 | Eng | 678 |
| 2 | Hin | 910 |
| 2 | Eng | 910 |

It can be brought to 4NF by decomposing into →

| StudentID | Course |
|---|---|
| 1 | math |
| 1 | Sci |
| 2 | Hin |
| 2 | Eng |

+

| StudentID | PhoneNbr |
|---|---|
| 1 | 123 |
| 1 | 345 |
| 2 | 678 |
| 2 | 910 |

5NF (Fifth Normal Form) or Project-Join Normal Form (PJNF) →

A relation is in 5NF if it satisfies the following conditions:

1) It is in 4NF.

2) The decomposition should be lossless.

Lossless Decomposition → It ensures that when a relation R is decomposed / broken into 2 or more relations, no data is lost,

and the original relation R can be again reconstructed by joining these decomposed relations.

Eg: There is a relation R(A,B,C) with C.K as A.

Step-1 → Lets decompose the relation based on the C.K or S.K of the given R. Decomposed relations: R1(A,B) and R2 (A,C).

R

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

→

R1

| A | B |
|---|---|
| 1 | 2 |
| 4 | 5 |

+

R2

| A | C |
|---|---|
| 1 | 3 |
| 4 | 6 |

Step-2 → Lets perform a natural join between R1 and R2, when we do R1 natural join R2 we get the original relation back (lossless).

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

R

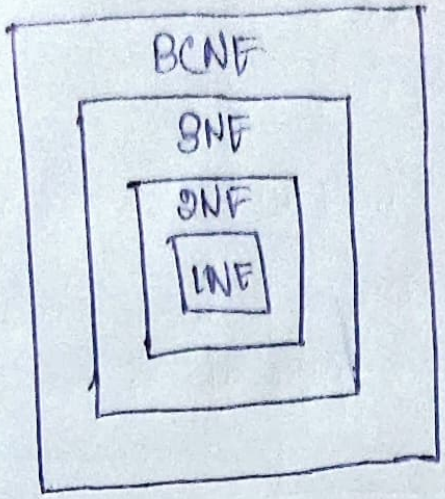
So, the relation R is in 5NF because the decomposition is lossless and R is in 4NF.

How to find the highest Normal Form of a given relation?

→ Step-1 → Identify the C.K for the given relation using F.D and closure method.

Step-2 → Find the prime and non-prime attributes.

Step-3 → Start checking for normal forms one by one according to their rule.




Q. For a given relation R(A,B,C) with the following functional dependencies: $A \to BC$, $B \to C$, $A \to B$, $AB \to C$, $B \to A$, find the highest normal form.

→ 1) Find the C.K for the relation.

∴ C.K : A,B

2) Find the prime and non-prime attributes.

P.A : {A, B}

N.P.A : {C}

3) checking for normal forms one by one according to their rule.

a) First Normal Form (1NF) → If a relation contains only atomic values (no multivalued attributes) it is in 1NF. Since we are assuming our relation R is in a standard relational model, it is already in 1NF.

b) Second Normal Form (2NF) → A relation is in 2NF if it is in 1NF and every non-prime attribute is fully functionally dependent on every candidate key of the relation (P.D → LHS is a proper subset of C.K AND RHS is a N.P.A).

$A \to BC$ = no partial dependency (A is a C.K)
$B \to C$ = no partial dependency (B is a C.K)
$A \to B$ = no partial dependency (A is a C.K)
$AB \to C$ = no partial dependency (AB is a combination of candidate keys, its S.K)
$B \to A$ = no partial dependency (B is a C.K)

∴ R is in 2NF.

c) Third Normal Form (3NF) ⇒

A relation is in 3NF if it is in 2NF and no transitive dependency exists. $X \to Y$ (X is a superkey OR Y is a prime attribute if true no transitive dependency).

$A \to BC$ = no transitive dependency (A is a C.K)
$B \to C$ = no transitive dependency (B is a C.K)
$A \to B$ = no transitive dependency (A is a C.K)
$AB \to C$ = no transitive dependency (AB is a S.K)
$B \to A$ = no transitive dependency (B is a C.K)

∴ R is in 3NF.

d) BCNF ⇒

A relation is in BCNF if it is in 3NF and for every F.D $X \to Y$, X is a S.K.

$A \to BC$ = A is a C.K
$B \to C$ = B is a C.K
$A \to B$ = A is a C.K
$AB \to C$ = AB is a S.K
$B \to A$ = B is a C.K

∴ R is in BCNF

∴ the highest normal form is BCNF.

How to normalise table?

→ In normalisation we generally break/decompose the table into 2 or more tables.

Steps to normalize a table →

1) Write down all the attributes of table, C.K, Prime and non-prime attributes and start analyzing with the F.D.

2) For table to be in 1NF: Table should have atomic (indivisible) values and a primary key.

3) For table to be in 2NF: No partial dependency (LHS proper subset of C.K and RHS non-prime attribute should be false).

4) For table to be in 3NF: No transitive dependency (LHS must be a C.K or RHS should be a prime attribute should be true).

5) For table to be in BCNF: LHS must be a C.K or S.K.

6) If it fails at any of these steps decompose the table on a common attribute which is a C.K (lossless).

Q. R(A, B, C, D) and assume we have the following F.D: A→B, B→C, C→D.

→ Step-1 → ABCD, C.K → A, P.A = {A}, N.P.A = {B, C, D}

Step-2 → Since we are assuming our relation R is in a standard relational model, it is already in 1NF.

Step-3 → check for 2NF

A→B = (no P.d as A is not a proper subset of C.K)

B→C = (no P.d as B is not a proper subset of C.K)

C→D = (no P.d as C is not a proper subset of C.K)

∴ It is in 2NF.

Step-4> check for 3NF

A->B = (no t.d as LHS is a C.K)
B->C = (t.d as LHS is not C.K and RHS is non-prime)
C->D = (t.d as LHS is not C.K and RHS is non-prime)

So, lets decompose the table

R1(A,B), R2(B,C), R3(C,D)

∴ Now it is in 3NF.

Step-5> check for BCNF

R1(A,B), A->B = (A is a C.K, So R1 is in BCNF)
R2(B,C), B->C = (B is a C.K, So R2 is in BCNF)
R3(C,D), C->D = (C is a C.K, So R3 is in BCNF)

Now, all decomposed relations are in BCNF

## Equivalence of Functional Dependencies

Functional Dependency> A functional dependency X->Y. holds on a relation R if, for any two tuples t1 and t2 in R, whenever, t1[X] = t2[X], then t1[Y] = t2[Y]

Two sets of functional dependencies, F and G, are equivalent if the following conditions hold.

a) F implies G: Every functional dependency in G can be derived from F.

b) G implies F: Every functional dependency in F can be derived from G.

If both conditions are true, then we say that F and G are equivalent.

How to find if two F.D are equivalent?

→ Step-1 → Compute the closure of both the sets.

Step-2 → Ensure that every functional dependency in set1 is in set2 closure.

Step-3 → Ensure that every functional dependency in set2 is in set1 closure.

Step-4 → If both subset checks pass, then set1 and set2 are equivalent.

Q Consider two sets of functional dependencies: $F = \{A \to B, B \to C\}$, $G = \{A \to C, A \to B\}$. Check if F and G are equivalent.

→ $F = \{A \to B, B \to C\}$

closure of F, attributes → A, B, C

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

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

$C^+ = \{C\}$

$G = \{A \to C, A \to B\}$

closure of G, attributes → A, B, C

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

$B^+ = \{B\}$

$C^+ = \{C\}$

F and G are not equivalent because $B \to C$ is not implied by G.

## Minimal Cover of Functional Dependency

Why do we need to find Minimal Cover?

→ It is a simplified version of the original set of functional dependencies.

1) It helps to remove redundant functional dependencies
2) It reduces the complexity of the functional dependencies.
3) It ensures that there are no unnecessary dependencies.

which can lead to anomalies in database operations (insertion, deletion, and update).

How to Find Minimal Cover?

→ Step-1 → Decomposition of FDs (RHS) i.e, $X \to AB$ can be written as $X \to A$, $X \to B$.

Step-2 → Remove Redundant F.D.

a) Make a new F.D set excluding the one you feel is redundant.

b) Now find the closure of LHS from the rest of the F.D and see if it determines all the attributes of a table, if yes you can remove that, if no jump to the next one.

Step-3 → Remove unnecessary attributes from LHS, if the determinant is a S.K, it can be reduced to C.K (minimal S.K)

Q Find the minimal cover of FD: $A \to BC$, $B \to C$, $A \to B$, $AB \to C$.

→ Step-1 → $A \to B$, $A \to C$, $B \to C$, $A \to B$, $AB \to C$

∴ FD: $A \to B$, $A \to C$, $B \to C$, $AB \to C$.

Step-2 →

a) For $A \to B$

FD: $A \to C$, $B \to C$, $AB \to C$

$A^+ = \{A, C\}$, since $A^+$ doesn't have all the attributes we shouldn't discard this.

b) For $A \to C$

FD: $A \to B$, $B \to C$, $AB \to C$

$A^+ = \{A, B, C\}$, since $A^+$ have all the attributes we can discard this.

c) For $B \Rightarrow C$

FD: $A \rightarrow B$, $AB \rightarrow C$

$B^+ = \{B\}$, Since $B^+$ doesn't have all the attributes, we shouldn't discard this.

d) For $AB \rightarrow C$

FD: $A \rightarrow B$, $B \rightarrow C$

$AB^+ = \{A, B, C\}$, Since $AB^+$ have all the attributes, we can discard this.

Therefore, the minimal cover of the given functional dependencies is: $\{A \rightarrow B, B \rightarrow C\}$.

Views in DBMS:

View is a virtual table that is derived from one or more underlying tables. This means that it does not physically store data but rather provides a logical representation of data.

# SQL

Why SQL?

→ we need a language to interact with databases. So, we use SQL to interact with DB, do some CRUD operations on the data.

Then what is MySQL?

→ MySQL is a specific Relational Database Management System (RDBMS) that uses SQL as its querying language.

## History of SQL (Structured Query language)

→ SQL originated in the 1970s from IBM's research on relational databases. It started as SEQUEL, later renamed SQL due to trademark issues.

## SQL (Structured Query language)

SQL is a programming language that is used to communicate and manipulate data in databases.

It helps user in performing CRUD (Create, Read, Update and Delete) operations in DB.

SQL allows users to perform a variety of tasks related to databases.

1) Retrieving Data → Extracting precise information from a database through queries

2) Manipulating Data → Adding, modifying, or removing records within a database.

3) Defining Data → Creating and adjusting the structure of a database, including tables and indexes.

4) Controlling Data → Managing database access by granting or revoking permissions.

MySQL Server → Database Server where data is stored, managed, and accessed.

MySQL WorkBench → It is a visual tool which is used for database design, development, administration, and management. It provides a user interface (UI) to interact with MySQL Server.

## Types of SQL Commands

SQL commands are divided into different categories based on their functionalities.

1) Data Query Language (DQL) commands.
2) Data Manipulation Language (DML) commands.
3) Data Definition Language (DDL) commands.
4) Data Control Language (DCL) commands.
5) Transaction Control Language (TCL) commands.

### 1) Data Query Language (DQL) Commands →

DQL is used to retrieve data from the database.
Commands : SELECT.

### 2) Data Manipulation Language (DML) commands →

DML is used to manipulate data stored in the database.
Commands : INSERT, UPDATE, DELETE.

### 3) Data Definition Language (DDL) commands →

DDL is used to define the structure and schema of the database. Commands : CREATE, ALTER, DROP, TRUNCATE, RENAME

4) Data Control Language (DCL) commands →

DCL deals with the control and security of data within the database. Commands: GRANT, REVOKE.

5) Transaction Control Language (TCL) commands ⇒ TCL is used to manage transactions within a database.

Commands: COMMIT, ROLLBACK, SAVEPOINT.

## Creation of Database

Lets understand database design from an example, consider a college database.

Database - School
table1 - Student (Sname, Rollno)
table2 - Teacher (~~Sno~~ Tname, Tid)
Sname, Rollno, Tname, Tid → attributes (characteristics)

Steps to create a Database →

1) choose a DBMS (Database Management System).
2) Connect to the server using a command-line tool or a graphical user interface.
3) create a new Database.
4) Once the database is created, you can use the USE statement to create tables in the database.
5) Create Tables and Insert Data.

We use the CREATE DATABASE statement to create a new database. These commands are not case-sensitive.

Command: CREATE DATABASE databaseName;

Also to avoid errors we can use →

Command: CREATE DATABASE IF NOT EXISTS databaseName;

IF NOT EXISTS and IF EXISTS clauses are commonly used in conjunction with the CREATE TABLE and DROP TABLE statements to avoid errors.

## Deletion of Database

Deleting a Database →

We use the DROP DATABASE statement to delete a database. Dropping a database means deleting the entire database, including all tables, data, and other objects within it. DROP is a DDL command. These commands are not case-sensitive.

Command: DROP DATABASE databaseName;

Also to avoid errors we can use →

Command: DROP DATABASE IF EXISTS databaseName;

## Using a Database

We can use the USE DATABASE statement to use a database. These commands are not case-sensitive.

Command: USE databaseName;

## Showing all the Databases

Showing databases →

We use the SHOW DATABASE statement to see all the databases present on a server.

Command: SHOW DATABASES;

## Creating a Table

We use the CREATE TABLE statement to create a table in DB.

Command:

```
CREATE TABLE TableName (
  column1 DataType1 Constraint1,
  Column2 DataType2 Constraint2,
  :
  column n DataTypen Constraintn,
);
```

CREATE is a DDL command.

Example:

```
CREATE TABLE employee(
  empId INT PRIMARY KEY,
  name VARCHAR(50),
  Salary INT
);
```

employee

| empId | name | Salary |
|---|---|---|
| | | |

## Inserting values into the Table

INSERT → DML command.

Command:

```
INSERT INTO tableName (column1, column2, ....., column N)
VALUES (value1, value2, ....., value N);
```

Example:

```
INSERT INTO employee
(empId, name, Salary)
VALUES
(1, "Raj", 1200), (2, "Rahul", 1100),
(3, "Riti", 1100);
```

employee

| empId | name | Salary |
|---|---|---|
| 1 | Raj | 1200 |
| 2 | Rahul | 1100 |
| 3 | Riti | 1100 |

## To See all the values in Table

### SELECT

Command:

```
SELECT * FROM tableName;
```

Example:

```
SELECT * FROM employee;
```

| empId | name | Salary |
|---|---|---|
| 1 | Raj | 1200 |
| 2 | Pahal | 1100 |
| 3 | Riti | 1100 |

## To See specific values in Table

### SELECT

command:

```
SELECT column1, ..., columnN FROM tableName;
```

Example:

```
SELECT empId FROM employee;
```

| empId |
|---|
| 1 |
| 2 |
| 3 |

## Seeing all the tables in a given Database

Command:

```
SHOW TABLES;
```

It helps us to see all the tables in a given database

## Datatypes in SQL

Datatypes are used to specify the type of data that a column can store.

| Numeric | Character/String | Date & Time | Boolean | Binary |
|---|---|---|---|---|
| → INTEGER/INT | → CHAR(n) | → DATE | → BOOLEAN | → BINARY(n) |
| → SMALLINT | → VARCHAR(n) | → TIME | | → VARBINARY(n) |
| → BIGINT | → TEXT | → DATETIME | | → BLOB |
| → DECIMAL | | → TIMESTAMP | | |
| → FLOAT | | | | |
| → DOUBLE | | | | |

## Numeric Datatypes

1) **INT** → Used to store whole numbers without decimal points. (-2147483648 to 2147483647) (Signed integer).

2) **BIGINT** → Used to store large whole numbers. (-9223372036854775808 to 9223372036854775807)

3) **FLOAT** → Used for storing decimal numbers (4-byte).

4) **DOUBLE** → Used for storing decimal numbers (8-byte).

5) **DECIMAL(P,S)** → Used for exact numeric representation.
P is the precision and S is the scale.

6) **SMALLINT** → Used to store small whole numbers. (-32767 to 32767).

By default all the numeric datatypes can have negative as well as positive values. This restrict the range so if we know there is only +ve values which is stored we use **UNSIGNED** attribute (0-255).
for eg → Salary can never be in negative or age.

example: 
```
CREATE TABLE example1(
    id INT UNSIGNED
);
```

## Character Datatypes

1) **CHAR(n)** → Fixed-Length character strings can be stored. (0-255)

2) **VARCHAR(n)** → Variable-Length character strings can be stored. (0-255)

3) TEXT → Variable-Length character string with no specified limit.

## Date & Time Datatypes.

1) DATE → Used for storing date values. (YYYY-MM-DD)
2) TIME → Used for storing time values. (hh:mm:ss)
3) DATETIME / TIMESTAMP → Used for storing date and time values. (YYYY-mm-dd hh:mm:ss)

## BOOLEAN Datatypes

1) BOOLEAN → Used to store a true or false value.

## BINARY Datatypes.

1) BINARY(n) → Used for fixed-length binary data.
2) VARBINARY(n) → Used for variable-length binary data.
3) BLOB (Binary Large OBjects) → Used for storing large amounts of binary data. (varlen)

## Keys in SQL

Primary Key → A primary key is a unique identifier for each record in the table. It ensures that each row can be uniquely identified and accessed within the table.

Foreign Key → A foreign key is a field in a table that refers to the primary key of another table. It establishes relationships between tables.

Student

| Roll No. | Name | Hometown |
|---|---|---|
| 1 | Rahul | Kolkata |
| 2 | Raj | Kalkata |
| 3 | Riti | Delhi |

Subject

| Roll No. | Name | Subject |
|---|---|---|
| 1 | Rahul | Maths |
| 2 | Raj | SST |
| 3 | Riti | Science |

refers

P.K
(Base / Referenced Table)

F.K
(Referencing Table)

Base / Referenced Table → Table having primary Key (P.K)

Referencing Table → Table having foreign Key (f.K)

## Constraints in SQL

Constraints → Constraints define rules or conditions that must be satisfied by the data in the table.

Common constraints include uniqueness, nullability, default values, etc.

1) Unique constraint → Ensures values in a column are unique across the table.

2) Not null constraint → Ensures a column cannot have a null value.

3) Check constraint → Enforces a condition to be true for each row.

4) Default Constraint → Provides a default value for a column if no value is specified.

5) Primary Key → Enforces the uniqueness of values along with being not null in one or more columns.

6) Foreign Key → Enforces a link between two tables by referencing a column in one table that is a primary key in another table.

Foreign Key

P.K

| id | Name | Course-id |
|---|---|---|
| 1 | Rahul | 100 |
| 2 | Raj | 101 |
| 3 | Riti | 102 |

References

P.K

| Course-id | name | teacher | id |
|---|---|---|---|
| 100 | Hindi | Ram | 1 |
| 101 | Maths | Mohan | 2 |
| 102 | English | Priya | 3 |

F.K

Student
(Base / Referenced / Parent table)

Course
(Referencing / Child table)

Foreign Key helps to perform operations related to the parent table, such as joining or ensuring referential integrity.

Query:

```
CREATE TABLE childTableName (
childId INT PRIMARY KEY,
baseId INT,
FOREIGN KEY(baseId) REFERENCES baseTableName(baseid)
);
```

# Cascading in Foreign Key

Cascading are a set of rules which dictate what actions should be taken automatically when a referenced row in the parent table is modified or deleted.

1) CASCADE → If a row in the parent table is updated or deleted, all related rows in the child's table will be automatically updated or deleted.

2) SET NULL → If a row in the parent table is updated or deleted, all corresponding foreign key values in the child table will be set to NULL.

3) RESTRICT or NO ACTION → Blocks the modification or deletion of a referenced row in the parent table if related rows exist in the child table thus maintaining referential integrity.

These cascading actions help to maintain the integrity of the data across related tables in the database.

1) ON DELETE CASCADE
2) ON UPDATE CASCADE

1) ON DELETE CASCADE → The ON DELETE CASCADE clause indicates that if a row in the parent table is deleted, all corresponding rows in the child table will be automatically deleted as well.

QUERY:

```
CREATE TABLE childTableName (
childId INT PRIMARY KEY,
baseId INT,
FOREIGN KEY (baseId) REFERENCES baseTableName (baseid)
ON DELETE CASCADE
);
```

2) ON UPDATE CASCADE → The ON UPDATE CASCADE clause indicates that if a row in the parent table is updated, all corresponding rows in the child table will be automatically updated as well.

QUERY:

```
CREATE TABLE childTableName(
 childId INT PRIMARY KEY,
 baseId INT,
 FOREIGN KEY (baseId) REFERENCES parentTableName (childId)
 ON UPDATE CACADE
 );
```

Let's make a Database for a Company

Requirements:

1) Make a database for a company xyz.

```
CREATE DATABASE xyz; USE xyz;
```

2) Make an employee table in the xyz database.

```
CREATE TABLE employee(
id INT PRIMARY KEY,
name VARCHAR(50),
age INT,
department VARCHAR(60),
city VARCHAR(50),
Salary INT
);
```

3) Fill details in the table.

```
INSERT INTO employee (id, name, age, department, city, salary)
 VALUES
(1, "rahul", 23, "IT", "Mumbai", 1500),
(2, "ajsara", 26, "HR", "Pune", 2000),
```

(3, "abhimanyu", 27, "IT", "Mumbai", 2500),
(4, "aditya", 25, "Marketing", "Surat", 2400),
(5, "raj", 24, "Finance", "Indore", 1500);

4) See all the data.
SELECT * FROM employee;

## UPDATE Command

The ~~up~~ UPDATE command in SQL is used to modify existing records in a table.

<u>If you get a safe mode error while executing queries, run this query →</u>

<u>QUERY:</u> SET SQL_SAFE_UPDATES = 0;

<u>QUERY:</u>

UPDATE table_name
SET Column Name1 = value1 (to be set), columnName2 = value2 (to be set),
...., columnNameN = ValueN (to be set)
WHERE condition;

<u>Example:</u>

1) UPDATE employee
SET salary = 60000
WHERE department = "HR";
→ a query to update salary for all employees in the "HR" department to 60000.

2) UPDATE employee
SET name = "raj"
WHERE name = "raaj";
→ a query to update the name of an employee from raaj to raj.

# DELETE Command

The DELETE command in SQL is used to remove records from a table.

QUERY:

DELETE FROM table_name
WHERE condition;

EXAMPLE:

1) DELETE FROM employee
WHERE department = "HR";
→ query to delete all records from the employee table where department is "HR".

2) DELETE FROM employee
WHERE name = "raj";
→ query to delete the record of an employee having the name as "raj".

## Retrieving data from table

SELECT command: Select is a DQL (Data Query Language) command. It is used to retrieve data from a database. We can provide the specific columns from which we can retrieve data.

SELECT column1, column2, ...., columnN FROM tableName;
→ to retrieve data present in specific column in a table

SELECT * FROM tableName; → to retrieve all the data present in the table.

## Filtering data using the WHERE clause

WHERE clause: It filters the rows based on specific conditions.

QUERY: SELECT col1, col2 FROM tableName WHERE condition;

Example: SELECT * FROM employee WHERE age > 20;

## ALTER Command

ALTER command is a DDL command used to modify existing database objects, such as tables, indexes, or constraints (schema).

### 1) ADD a Column →

Query:

```
ALTER TABLE tableName
ADD columnName datatype constraint;
```

### 2) Drop a column →

Query:

```
ALTER TABLE tableName
DROP COLUMN columnName;
```

### 3) Modify the datatype of an existing column →

MODIFY clause → The MODIFY clause is oftenly used within an ALTER TABLE statement in SQL. It allows us to change the definition or properties of an existing column in a table.

Query:

```
ALTER TABLE tableName
MODIFY columnName newdatatype;
```

The above command modifies columnName to a new dataType.

4) Change the name of an existing column →

CHANGE → The CHANGE command is afterly used within an ALTER TABLE statement in SQL. It helps to change the name or data type of a column within a table.

Query:

```
ALTER TABLE tableName
CHANGE oldcolumnName newcolumnName newdatatype;
```

The above command changes the oldcolumnName to newcolumnName and it dat type.

RENAME Command → The RENAME command is used to change the name of an existing database object, such as a table, column, index, or constraint.

Query:

```
ALTER TABLE tableName
RENAME COLUMN oldcolumnName To newcolumnName;
```

The above command renames the oldcolumnName to the newcolumnName.

## RENAME Command

RENAME → RENAME command is used to change the name of an existing database Object, such as a table, column, index, or constraint..

Query (Table Renaming):

```
RENAME TABLE oldTableName To newTableName;
```

The above command renames the oldTableName to newTableName.

## QUERY (Column Renaming):

```
ALTER TABLE tableName
RENAME COLUMN oldcolumnName to newcolumnName;
```

The above command renames the oldcolumnName to newcolumnName.

## QUERY (Database Renaming):

```
RENAME DATABASE olddatabaseName to newdatabaseName;
```

## TRUNCATE command

This command removes all rows from the given table, ~~to~~ leaving the table empty but preserving its structure.

## QUERY:

```
TRUNCATE TABLE tableName;
```

## Difference between TRUNCATE, DELETE and DROP

| TRUNCATE | DELETE | DROP |
|---|---|---|
| removes all rows from a table. | Used to remove specific rows from a table based on a given condition. | Used to completely remove the table. |
| TRUNCATE TABLE tableName; | DELETE FROM tableName WHERE condition; | DROP TABLE tableName; |

## Using DISTINCT to retrieve unique values

DISTINCT → DISTINCT keyword is used within the SELECT statement to retrieve unique values from a column or combination of columns.

QUERY:

1) SELECT DISTINCT col1 FROM tableName; → retrieve a list of unique values for col1

2) SELECT DISTINCT col1, col2 FROM tableName; → return unique combinations of col1 & col2

## Operators in SQL

To perform operations on data in SQL, we use operators.

QUERY: SELECT col1 col2 FROM tableName WHERE condition (use operator);

Types of operators in SQL:

1) Arithmetic Operators → addition (+), subtraction (-), multiplication (*), division (/), modulus (%).

QUERY: SELECT * FROM employee WHERE age+1 = 60;

2) Comparison Operators → equal to (=), not equal to (<> or !=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=).

QUERY: SELECT * FROM employee WHERE age > 20;

3) Logical Operators →

a) AND → It combines two conditions and returns true if both are

true.

QUERY: SELECT * FROM employee WHERE city = 'Pune' AND age > 18;

b) OR → It combines two conditions and returns true if either is true.

QUERY: SELECT * FROM employee WHERE city = 'Pune' OR age > 18;

c) NOT → It reverses the result of a condition, returns true if the condition is false and vice versa.

QUERY: SELECT * FROM employee WHERE department NOT IN ('IT', 'HR');

4) IN Operator → The IN operator checks if a value matches in a list of values).

QUERY: SELECT * FROM employee WHERE department IN ('IT', 'HR');

5) IS NULL / IS NOT NULL Operators → IS NULL checks for null values and IS NOT NULL checks for NOT NULL values.

QUERY: SELECT * FROM employee WHERE department IS NOT NULL;

6) BITWISE Operators → AND(&), OR(|)

7) LIKE & wildcard Operators → LIKE operator is used to search for a specified pattern in a column. It uses wildcard operators for matching patterns.

a) % (percent sign) → It matches for any sequence of 0 or

more characters.

QUERY: SELECT * FROM employee WHERE name LIKE ('A%');

b) _(underscore) → It matches for any single character.

QUERY: SELECT * FROM employee WHERE name LIKE ('_A%');

8) BETWEEN Operator → checks if a value is within a range of values.

QUERY: SELECT * FROM employee WHERE salary BETWEEN 1200 AND 1500;

## Clauses in SQL

clauses are like tools/conditions that helps us to make queries more specific or decide what data to fetch.

Example: WHERE, GROUP BY, HAVING, ORDER BY, LIMIT.

QUERY:
```
SELECT col1, col2
FROM tablename
clause condition;
```

WHERE clause → It filters the rows based on specified conditions.

QUERY:
```
SELECT col1, col2
FROM tableName
WHERE condition;
```

example: SELECT * FROM employee WHERE age > 20;

LIMIT clause → The LIMIT clause in SQL is used to restrict the number of rows returned by a query.

This query retrieves the first n rows from the table.

QUERY:

```
SELECT col1, col2 FROM tablename
LIMIT noofRows;
```

Example:

```
SELECT * FROM employee LIMIT 2;
```

ORDER BY clause → It is used to sort the results in ascending or descending order. By default it returns the result in ascending order.

This query retrieves the first n rows from the table.

QUERY:

```
SELECT col1, col2 FROM tableName
ORDER BY col1(ASC/DESC), col2(ASC/DESC)
```

Example:

```
SELECT * FROM employee ORDER BY salary DESC;
```

Q Write a SQL Query to fetch the details of employees having id as 1.

→
```
SELECT * FROM employee
WHERE id = 1;
```

Q Write a SQL Query to fetch the details of employees having id as 1 and city as MUMBAI.

→
```
SELECT * FROM employee
WHERE id = 1 AND city = "MUMBAI";
```

Q Write a SQL Query to fetch the details of an employee having Salary greater than 1200 and city as MUMBAI.

→ SELECT * FROM employee
WHERE salary > 1200 AND city = "MUMBAI";

Q Write a SQL Query to fetch the details of employees who are not from MUMBAI.

→ SELECT * FROM employee
WHERE city NOT IN ("MUMBAI");

Q Write a SQL Query to fetch the details of employees having Salary in descending order.

→ SELECT * FROM employee
ORDER BY salary DESC;

Q Write a SQL Query to fetch the details of 2 employees having the maximum salary.

→ SELECT * FROM employee
ORDER BY salary DESC
LIMIT 2;

## Aggregate Functions

Aggregate functions performs some operation on a set of rows and then returns a single value summarizing the data. These are used with SELECT statements to perform calculations.

Types of Aggregate functions →

1) COUNT()
2) SUM()
3) AVG()
4) MIN()
5) MAX()
6) GROUP_CONCAT()

1) COUNT() → It counts the number of rows in a table or the number of non-null values in a column.
This counts how many things are in a list or a group.

Query: SELECT count(name) FROM employee;
↳ this will tell the number of employees in the company.

2) SUM() → It calculates the sum of all values in a numeric column. This adds up all the numbers in a list.

Query: SELECT SUM(salary) FROM employee;
↳ this will tell the total amount company is paying to its employees.

3) AVG() → It calculates the average of all values in a numeric column. It finds the average of all the numbers in a list.

Query: SELECT AVG(salary) FROM employee;
↳ this will tell the avg. amount company is paying to its employees.

4) MIN() → It calculates the minimum of all values in a numeric column / list.

Query: SELECT MIN(salary) FROM employee;
↳ this will tell the min. salary company is paying to its employees.

5) MAX() → It calculates the maximum of all values in a numeric column / list.

Query: SELECT MAX(salary) FROM employee;
↳ this will tell the max. salary that the company

is paying to its employees.

6) GROUP_CONCAT() → It combines data from multiple rows into a single string.

QUERY:

```
SELECT col1, col2, ....., colN
GROUP_CONCAT([DISTINCT] col_name1 [ORDER BY clause] [SEPA-
RATOR 'str_val])
FROM table_name
GROUP BY col_name2;
```

Grouping data with the GROUP BY clause

GROUP BY clause → This is used to group rows that have the same values together. It helps to organize data into groups so that you can do calculations, like finding totals or averages for each group.

This query retrieves the first n rows from the table.

QUERY:

```
SELECT col1, aggregateFun(col2)
FROM tableName
GROUP BY col1;
```

Example:

```
SELECT department, AVG(Salary) AS avgsal FROM employee
GROUP BY department;
```

HAVING clause → The HAVING clause is just like WHERE clause, but the main difference is it works on aggregated data. It is used with the GROUP BY clause. It helps to filter groups based on given conditions.

QUERY:

```
SELECT col1, col2, aggregateFun(col3)
FROM tableName
GROUP BY col1, col2
HAVING condition;
```

Example:

```
SELECT department, AVG(salary) AS avgsal
FROM employee
GROUP BY department
HAVING avgsal >1500;
```

Difference Between WHERE and HAVING clause

| WHERE | HAVING |
|---|---|
| 1) Used to filter rows from the result based on condition applied to a row before the aggregation. | 1) Used to filter rows from the result based on condition applied to a row after the aggregation |
| 2) It is used with SELECT, UPDATE, or DELETE commands. | 2) It is used with GROUP BY and aggregate functions |
| 3) SELECT * FROM tableName WHERE condition; | 3) SELECT col1, col2, aggregateFun(col3) FROM tableName GROUP BY col1 col2 HAVING condition; |

Q Write a query to find the total number of employees in each city.

→ SELECT city, COUNT(name) AS no_of_emp
FROM employee
GROUP BY city;

Q Write a query to find the maximum salary of employees in each city in descending order.

→ SELECT city, MAX(salary) AS max_salary
FROM employee
GROUP BY city
ORDER BY max_salary DESC;

Q Write a query to display the department names alongside the total count of employees in each department, sorting the results by the total number of employees in descending order.

→ SELECT department, COUNT(id) AS total employees
FROM employee
GROUP BY department
ORDER BY totalemployees DESC;

Q Write a query to list the departments where the average salary is greater than 1200, also display the department name and the average salary.

→ SELECT department, AVG(salary) AS avgsalary
FROM employee
GROUP BY department
HAVING AVG(salary) > 1200;

## The general order of SQL commands

| S.no | Command | Usecase |
|---|---|---|
| 1. | SELECT | Retrieve from the database. |
| 2. | FROM | Identify the table. |
| 3. | WHERE | Filter rows base on some conditions. |
| 4. | GROUP BY | Group rows that have the same values. |
| 5. | HAVING | Filter groups based on some conditions. |
| 6. | ORDER BY | Sort the result set either asc /desc |
| 7. | LIMIT | Limit the number of rows returned |

## Joins in SQL

Joins are used to combine rows from two or more tables based on a related or shared or common column between them. There are commonly 4 types of joins including INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, SELF JOIN, CROSS JOIN.

OUTER JOIN

Student

| Id | Name | Age |
|---|---|---|
| 1 | Riya | 17 |
| 2 | Rahul | 18 |
| 3 | Ram | 17 |

Course

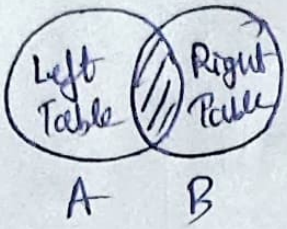
| Id | course_id | course_name |
|---|---|---|
| 1 | 101 | Eng |
| 2 | 102 | Hin |
| 3 | 103 | Phy |

Q Is Foreign Key important for performing joins?

→ Joins can be performed based on any columns that establish a relationship between tables, not just FK constraints, so its not necessary.

## Types of Joins in SQL →

### 1) Inner Join →




Student

| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

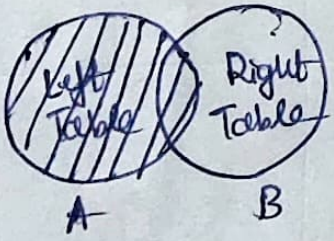
Course

| rollno | c-name |
|---|---|
| 2 | Hindi |
| 3 | Eng |
| 4 | math |

⇓

| rollno | name | c_name |
|---|---|---|
| 2 | Rahul | Hindi |
| 3 | Riti | Eng |

### 2) LEFT / LEFT OUTER JOIN →




| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

Student

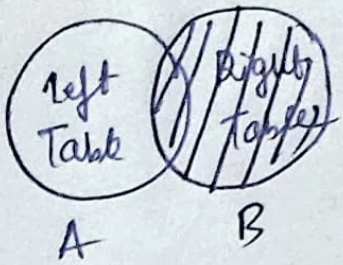
| rollno | c-name |
|---|---|
| 2 | Hindi |
| 3 | Eng |
| 4 | Math |

Course

⇓

| rollno | name | c_name |
|---|---|---|
| 1 | Ram | null |
| 2 | Rahul | Hindi |
| 3 | Riti | Eng |

## 3) Right / Right Outer Join →




Student

| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

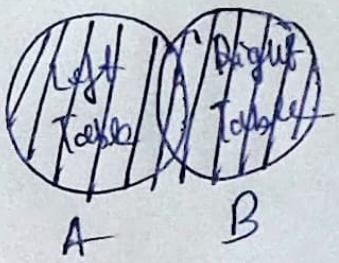
Course

| rollno | c_name |
|---|---|
| 2 | Hindi |
| 3 | Eng |
| 4 | Math |

⇓

| rollno | name | c_name |
|---|---|---|
| 2 | Rahul | Hindi |
| 3 | Riti | Eng |
| 4 | null | Math |

## 4) Full / Full Outher Join →




Student

| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

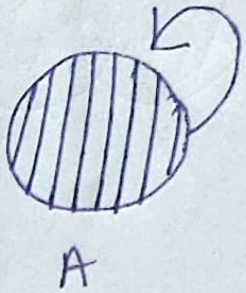
Course

| rollno | c_name |
|---|---|
| 2 | Hindi |
| 3 | Eng |
| 4 | Math |

⇓

| rollno | name | c_name |
|---|---|---|
| 1 | Ram | null |
| 2 | Rahul | Hindi |
| 3 | Riti | Eng |
| 4 | null | Math |

## 5) Self Join →



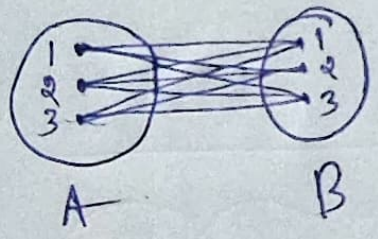

| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

Student

There is no Keyword "SELF JOIN", rather "JOIN" is used.

It Will be discussed Later.

## 6) Cross Join →




| rollno | name |
|---|---|
| 1 | Ram |
| 2 | Rahul |
| 3 | Riti |

Student

| rollno | c_name |
|---|---|
| 2 | Hindi |
| 3 | Eng |
| 4 | Math |

Course

⇓

| Student_rollno | name | Course_rollno | c_name |
|---|---|---|---|
| 1 | Ram | 2 | Hindi |
| 1 | Ram | 3 | Eng |
| 1 | Ram | 4 | math |
| 2 | Rahul | 2 | Hindi |
| 2 | Rahul | 3 | Eng |
| 2 | Rahul | 4 | math |
| 3 | Riti | 2 | Hindi |
| 3 | Riti | 3 | Eng |
| 3 | Riti | 4 | math |

1) <u>Inner Join</u> ⇒ It helps us in getting the rows that have matching values in both tables, according to the given join condition.

<u>Query:</u>

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.colName = table2.colName;
```

<u>Example:</u>

```
SELECT *
FROM customer
INNER JOIN order
ON customer.id = order.id;
```

↓

array to return rows where there is a matching id in both tables

customer

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

order

| id | c_name |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Utensils |

⇓

| id | name | id | c_name |
|---|---|---|---|
| 102 | Rahul | 102 | Fruit |
| 103 | Riti | 103 | Ball |

we need to mention columns if we don't want id to repeat.

2) Left Join / Left Outer Join → It is used to fetch all the records from the left table along with matched records from the right table. If there are no matching records in the right table, NULL values are returned for the columns of the right table.

Query:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.colName = table2.colName;
```

Left table → the table is specified before the LEFT JOIN keyword.

Right table → the table is specified after the LEFT JOIN keyword.

Example:

```
SELECT *
FROM customer
LEFT JOIN order
ON customer.id = order.id;
```

customer

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

order

| id | c-name |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Utensils |

⇓

| id | name | id | c-name |
|---|---|---|---|
| 101 | Ram | null | null |
| 102 | Rahul | 102 | Fruit |
| 103 | Riti | 103 | Ball |

3) Right Join / Right Outer Join → It is used to fetch all the records from the right table along with matched records from the left table. If there are no matching records in the left table, NULL values are returned for the columns of the left table.

Query:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.colName = table2.colName;
```

Left table → the table specified before the RIGHT JOIN keyword.

Right table → the table specified after the LEFT JOIN keyword.

Example:

```
SELECT *
FROM customer
RIGHT JOIN order
ON customer.id = order.id
```

customer

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

order

| id | cname |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Utensils |

⇓

| id | cname | id | name |
|---|---|---|---|
| 102 | Fruit | 102 | Rahul |
| 103 | Ball | 103 | Riti |
| 104 | Utensils | null | null |

4) Full Join / Full Outer Join → It returns the matching rows of both left and right table and also includes all rows from both tables even if they don't have matching rows. If there is no match, NULL values are returned for the columns of the missing table.

In MySQL, the syntax for a full join is different compared to the other SQL databases like PostgreSQL or SQL Server.

MySQL does not support the FULL JOIN keyword directly. So we use a combination of LEFT JOIN; RIGHT JOIN and UNION to achieve the result.

Query:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.colName = table2.colName
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.colName = table2.colName;
```

5) Cross Join → It combines each row of the first table with every row of the second table.

Query:

```
SELECT *
FROM table1
CROSS JOIN table2;
```

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |

customer

| o_id | o_name |
|---|---|
| 1 | Fruit |
| 2 | Ball |

Order

It results in a new table where the number of rows is equal to the product of number of rows in each table. (m×n).

⇓

| id | name | o_id | o_name |
|---|---|---|---|
| 101 | Ram | 1 | Fruit |
| 101 | Ram | 2 | Ball |
| 102 | Rahul | 1 | Fruit |
| 102 | Rahul | 2 | Ball |

6) <u>Self Join</u>: A self join in SQL is a type of join where a table is joined with itself. It is a type of inner join.

<u>Query</u>:

```
SELECT columns
FROM table as t1
JOIN table as t2
ON t1.colName = t2.colName
```

t1 and t2 are aliases for the table, used to distinguish between the order rows.

<u>Example</u>:

Ram (Mentor)

Rahul (Mentor)

Riti (Mentor)

↓

Riya (Student)

| s_id | name | mentor_id |
|---|---|---|
| 1 | Ram | null |
| 2 | Rahul | 1 |
| 3 | Riti | 1 |
| 4 | Riya | 3 |

Student

```
SELECT s1.name as mentor_name, s2.name as name
FROM student as s1 JOIN student as s2
WHERE s1.s_id = s2.mentor_id.
```

⇓

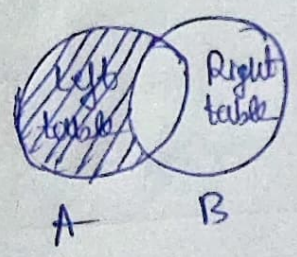
| mentor_name | name |
|---|---|
| Ram | Riti |
| Ram | Rahul |
| Riti | Riya |

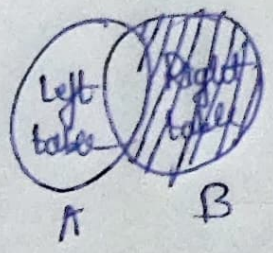
## Exclusive Joins in SQL

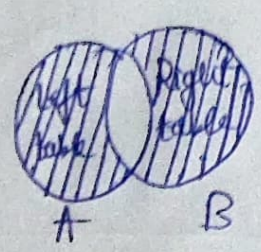
Exclusive joins are used when we want to retrieve data from two tables excluding matched rows. They are a part of outer joins or full outer join.

Types:

1) Left Exclusive JOIN → 2) Right Exclusive JOIN → 3) Full Exclusive JOIN →




1) Left Exclusive JOIN → when we retrieve records from the left table excluding the ones matching in both left and right table.

Query:

```
SELECT Columns
FROM table1 LEFT JOIN table2
ON table1.colName = table2.colName;
WHERE table2.colName IS NULL;
```

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

| id | o_name |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Watch |

⇓

| id | name | id | o_name |
|---|---|---|---|
| 101 | Ram | null | null |

2) RIGHT Exclusive JOIN ⇒ when we receive records from the right table excluding the ones matching in both left and right table.

Query:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.colName = table2.colName
WHERE table1.colName IS NULL;
```

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

| id | o_name |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Utensils |

⇓

| id | o_name | id | name |
|---|---|---|---|
| 104 | Utensils | null | null |

3) Full Exclusive JOIN ⇒ when we retrieve records from the right table and left table excluding the ones matching in both left and right table.

Query:

```
SELECT columns
FROM table1
LEFT JOIN table2
```

| id | name |
|---|---|
| 101 | Ram |
| 102 | Rahul |
| 103 | Riti |

| id | o_name |
|---|---|
| 102 | Fruit |
| 103 | Ball |
| 104 | Utensils |

```
ON table1.colName = table2.colName
WHERE table2.colName IS NULL
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.colName = table2.colName
WHERE table1.colName IS NULL;
```

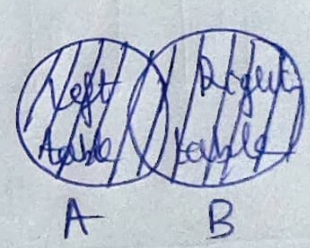
| id | name | id | o_name |
|---|---|---|---|
| 101 | Ram | null | null |
| null | null | 104 | Utensils |

## UNION operator in SQL

UNION operator in SQL is used to combine the results of two or more SELECT queries into a single result set and gives unique rows by removing duplicate rows.

### Things to Keep in mind:

1) Each SELECT command within the UNION must retrieve the same number of columns.
2) The data types of columns in corresponding positions across SELECT statements should match.
3) Columns should be listed in the same order across all SELECT statements.




### QUERY:

```
SELECT columns
FROM table1
UNION
SELECT columns
FROM table2;
```

| id |
|---|
| 1 |
| 2 |
| 3 |

U

| id |
|---|
| 2 |
| 3 |
| 4 |

→

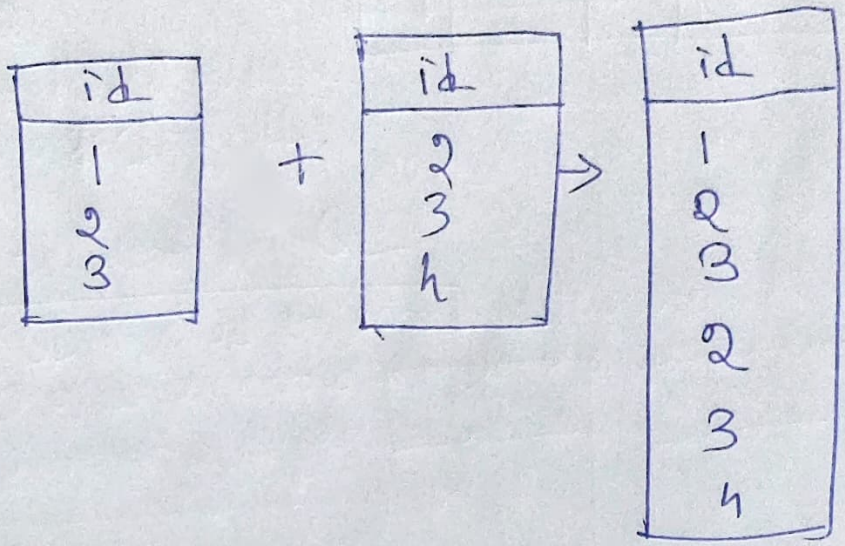
| id |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

# UNION ALL Operator in SQL

UNION ALL operator in SQL is used to combine the results of two or more SELECT queries into a single result set and gives all rows by not removing the duplicate rows.

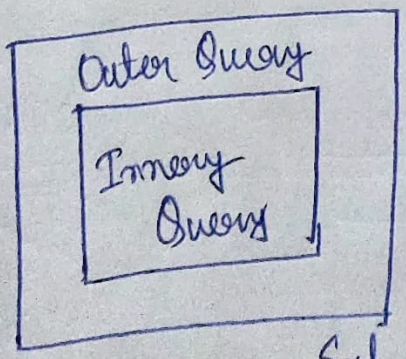
QUERY:

```
SELECT Columns
FROM table1
UNION ALL
SELECT columns
FROM table2;
```




# SQL Subqueries / Nested queries

Subqueries / Inner Queries / Nested Queries → SQL Subquery is a query nested within another SQL statement. Whenever we want to retrieve data based on the result of another query we use nested queries.




How can we use Subqueries?

→ Subqueries can be used in multiple ways:

1) Subqueries can be used with clauses such as SELECT, INSERT, UPDATE, or DELETE to perform complex data retrieval.

Query:

```
SELECT columns, (Subquery)
FROM tablename;
```

2) Subqueries can be used with WHERE clause to filter data based on conditions.

Query:

```
SELECT *
FROM tableName
WHERE columns name operator (subquery);
```

3) Subqueries can also be used in the FROM clause.

Query:

```
SELECT *
FROM (Subquery) AS altNames;
```

Let's understand from example of using subqueries in WHERE:

1) Find all the employees who have salary greater than the min salary.

- Find the min salary
- Find employee having salary greater than min salary

| id | name | age | department | city | salary |
|---|---|---|---|---|---|
| 1 | Rahul | 05 | 'IT' | 'Mumbai' | 1500 |
| 2 | Afsana | 26 | 'HR' | 'Pune' | 2000 |
| 3 | Abhimanyu | 27 | 'IT' | 'Mumbai' | 2500 |
| 4 | Aditya | 25 | 'MARKETING' | 'Surat' | 2400 |
| 5 | Raj | 21 | 'FINANCE' | 'Indore' | 1500 |

To find the min salary →

QUERY:

```
SELECT MIN(Salary) FROM employee;
```

To find all the employees having salary greater than min salary →

Query:

```
SELECT name, salary
FROM employee
WHERE salary > (Subquery);
```

⇓

| name | salary |
|---|---|
| Afsana | 2000 |
| Abhimanyu | 2500 |
| Aditya | 2400 |

2) Find all the employees with the minimum age

- Find the min age.
- Find employee having the min age.

To find the min age →

```
SELECT MIN(age) FROM employee;
```

To find all the employees having min age →

```
SELECT name, age
FROM employee
WHERE age = (Subquery);
```

⇓

| name | age |
|---|---|
| Raj | 24 |

## Let's understand from example of using subqueries in FROM:

1) Find the employees who is having age greater than min age.

- Find the min age
- Find Employee having age > min age

### To Find the min age →

Query:

```
SELECT MIN(age) AS min_age FROM employee;
```

### Find employee having age > min age →

Query:

```
SELECT emp.name
FROM employee emp (Subquery) AS Subquery
WHERE emp.age > Subquery.min_age
```

⇓

| name |
|---|
| Rahul |
| Afsana |
| Abhimanyu |
| Aditya |

## Let's understand from an example of using subqueries in SELECT:

1) Print the employees with the average age and age of employees

- Find the avg age
- Print the employee age and avg_age

### To Find the avg age →

Query:

```
SELECT AVG(age) FROM employee;
```

Print the employee age and avg_age →

Query:

```
SELECT (subquery) AS avg_age, age
FROM employee;
```

⇓

| avg_age | age |
|---|---|
| 25.4000 | 25 |
| 25.4000 | 26 |
| 25.4000 | 27 |
| 25.4000 | 25 |
| 25.4000 | 24 |

## Nth Highest Salary

Q Find the nth highest salary in a given dataset.

→ Step-1 → Select the column which you want to show the final result i.e salary.

Step-2 → Order the salary in descending order so that you have the max at the first.

Step-3 → Now ~~at~~ the value of n could be 1,2,3,... till n, so we have to make the array in such a way so that whatever be the value of n it can provide the result.

Step-4 → So at the end of the query we will provide a LIMIT so that on the data set which we have got after ordering the salary in descending order, we can fetch the nth highest one.

LIMIT → The LIMIT clause is used to restrict the number of rows returned by a query.

LIMIT n → It helps to retrieve a maximum of n rows from the begining of the result set.

LIMIT m,n → It helps to retrieve a specific range of rows where
m → number of rows to skip from the begining.
n → number of rows to fetch after skipping.

Query :

```
SELECT DISTINCT Salary
FROM tableName
ORDER BY Salary DESC
LIMIT n-1,1;
```

→ here we are using distinct to get unique records.

## Stored Procedures

Stored Procedures → These are programs that can perform specific tasks based on the stored query. It is basically a collection of pre-written SQL statements grouped together under a specific name.

Query: (to create a procedure)

```
CREATE PROCEDURE procedureName()
BEGIN
 Query
END;
```

Query : (to call a procedure)

```
CALL procedureName();
```

Examples of stored procedures without params :

Query 1 :

```
DELIMETER /
CREATE PROCEDURE getAllOrderDetails()
BEGIN
 Select * from orders;
END /
DELIMETER ;
```

→ used to set the end of query

→ Again it is changed back to normal

to call the procedure →

```
CALL getAllOrderDetails();
```

Example to return the details of the order by id (Stored procedure with params):

Query 2:

```
DELIMETER /
CREATE PROCEDURE getAllOrderDetailsById (IN id int)
BEGIN
SELECT * FROM Orders WHERE id = id;
END /
DELIMETER;
```

to call the procedure →

```
CALL getAllOrderDetailsById (2);
```

## Views in SQL

A view is a virtual table in SQL. It helps in providing a filtered view of data for security purposes.

QUERY:

```
CREATE VIEW viewName AS
SELECT columns FROM baseTableName (Specify the columns to be included in the view)
```

It helps in Data Abstraction, Security and simplify complex queries.

To see all the data in view →

QUERY:

```
SELECT * FROM viewName;
```

To drop a view →

QUERY:

```
DROP VIEW IF EXISTS viewName;
```

# CASE AND IF IN SQL

**CASE →** CASE allows you to perform conditional logic within a query. It can be used in both SELECT and UPDATE statements to evaluate conditions and return specific values based on those conditions.

**QUERY:**

```
CASE
  WHEN condition1 THEN result1 WHEN
  condition2 THEN result2 .... ELSE resultN
END
```

## CASE with SELECT statement

Q Categorize the students on basis of of their percentage to Top, Pass and Fail in a new column category.

→ **QUERY:**

```
SELECT sid, name, percentage,
CASE
  WHEN percentage >90 THEN 'Top'
  WHEN percentage BETWEEN 89 AND 34 THEN 'Pass'
  ELSE 'FAIL'
END AS category
FROM student;
```

## CASE with UPDATE statement

Q Students have got some grace marks so update their grades. Where its A update to A+ and where its B update to A.

**QUERY:**

```
UPDATE student
SET grade = CASE
    WHEN grade = 'B' THEN 'A'
    WHEN grade = 'A' THEN 'A+'
END;
```

# IF IN SQL

**IFS** It is used to return one of two values depending on whether a condition is true or false. It is not supported in many DB but supported in MySQL.

**QUERY:**

IF(condition, value_if_true, value_if_false)

## IF with SELECT statement

**Q** Categorize the students on basis of their percentage to Top, Pass and Fail for a new column category.

**QUERY:**

```
SELECT sid, sname, percentage,
IF (percentage > 90, 'Top', IF(percentage BETWEEN 89 AND 34, 'Pass',
'Fail')) AS category
FROM student;
```

## IF with UPDATE statement

**Q** Swap all 'f' and 'm' values (i.e, change all 'f' values to 'm' and vice versa) with a single update statement and no intermediate temporary tables.

**QUERY:**

```
UPDATE employee
SET gender = IF (gender = 'm', 'f', 'm');
```

# Transaction and Concurrency Control

Transaction is a logical unit of work that comprises one or more database operations (like Read/Write/Commit/rollback). In a transaction both read and write operations are fundamental actions that ensure ACID properties of transactions (data consistency and integrity).

Read(R) → A read operation involves retrieving / fetching data from the database.

Write(W) → A write operation in a transaction involves modifying data in the database.

Swiggy → Order → Payment Page → Enter Bank Details → Otp

→ network failure / issues → rollback

→ No network Failure → Order done. (Commit)

Consider an example of a banking application where a customer Ram wants to transfer 100 Rs to Shyam.

→ Step-1 → Begin Transaction.

Step-2 → The application will fetch / read the current bill of Ram → R(Ram).

Step-3 → The application will calculate the new balances after the transfer → Ram = Ram − 100.

Step-4 → Update it in the temporary Storage / Transaction logs → W(Ram)

Step-5 → The application will fetch / read the current balance of Shyam → R(Shyam).

Step-6 → The application will calculate the new balances after the transfer. → Shyam = Shyam + 100.

Step-7 → Update it in the ~~to~~ temporary Storage / Transaction logs → W(Shyam).

Step-8 → If all updates are successful and there are no errors, commit the transaction to make the changes permanent. Logs are maintained till the occurence of commit, after that log files are deleted and database is updated till Step-7 → COMMIT

Step-9 → If an error occurs during any step rollback the transaction to revert changes made within the transaction.
→ ROLLBACK.

| Transaction States | Case-1 | Case-2 |
|---|---|---|
| BEGIN | ACTIVE | ACTIVE |
| R(Ram) | | |
| Ram = Ram - 100 | | |
| W(Ram) | | |
| R(Shyam) | | |
| Shyam = Shyam + 100 | | |
| W(Shyam) | PARTIALLY COMMITED | |
| COMMIT | SUCCESSFUL | FAILED |
| ROLLBACK | | CANCELLED / ROLLBACK |
| | COMPLETED | COMPLETED |

Concurrency Control → It ensures that multiple transactions can run concurrently without compromising data consistency.

Example: Consider a banking system where two transactions are happening concurrently.
1) Ram giving Shyam 100 Rs.

2) Shyam giving Ram 50 Rs, the data should be consistent for both transactions.

Some techniques used here are →

1) Locking.
2) Two - Phase Locking (2PL).
3) Timestamp Ordering.

## ACID Properties

ACID properties are the properties which ensure that transactions are processed reliably and accurately, even in complex situation (system failures / network issues).

- A → Atomicity → Either execute all operations or none.
- C → Consistency → Read should fetch upto date data and write should not violate integrity constraints.
- I → Isolation → One transaction should be independent of the other transaction.
- D → Durability → The commited transaction should remain even after a failure or crash.

A → Atomicity → It ensures that a transaction is treated as a single unit of work. Either all operations are completed successfully (commit) or none of them are applied (rollback). This guarantees that the database remains in a consistent state despite any failures or interruptions during the transaction.

Example → Consider Ram is transferring money to Shyam. The transaction must deduct the amount from Ram's account

and add it to Shyam's account as a single operation. If at any moment, this transaction fails (e.g. due to insufficient funds / system error / network error), the entire transaction is rolled back, ensuring that none of the accounts are affected partially.

C → Consistency → It ensures that

1) Read operations retrieve consistent and up-to-date data from the database

2) Write operations ensure that data modifications maintain database constraints (such as foreign key relationships or unique constraints so that data remains accurate)

It guarantees that the database remains in a consistent state before and after the execution of each transaction.

Example → Consider you had 100 Rs in your account but you want 50 Rs cash, so you transferred 50 Rs to person X and he gave you 50 Rs cash.

Before transaction → 100 Rs (in account)

After transaction → 100 Rs (50 Rs in account + 50 Rs in cash).

I → Isolation → It ensures that if there are 2 transactions 1 and 2, then the changes made by Transaction 1 are not visible to Transaction 2 until Transaction 1 commits. While the transaction is reading data, the dbms ensures that data is consistent and indexed from other transactions. This means that other transactions cannot modify the data being read by the current transaction until it is committed or rolled back.

Example → A = 40 Rs (in DB).

Transaction 1 → Update value of A to 50 Rs.

Transaction 2 → Read (get / fetch value of A.

If Transaction 1 is committed, according to Transaction 2 the value of A = 50 Rs.

If Transaction 1 is not committed / running, according to Transaction 2 the value of A = 40 Rs.

D → Durability → It ensures that once you save your data (commit a transaction), it stays saved, even if the system crashes or there is a power failure. Your data is always safe and won't disappear after you save, as committed transactions are not lost.

Most DBMS use a technique called Write-Ahead Logging (WAL) to ensure durability. Before modifying data in the database, the DBMS writes the changes to a transaction log (often stored on disk) in a sequential manner. This ensures that if there is a failure event, the database can recover to a consistent state.

Example → Consider if you are transferring 100 Rs to your friend and there is a sudden power outage or the system crashes right after the transaction is committed, the changes (the transfer of 100) will be saved in the database. When the system is back up, both your account and your friend's account will reflect the updated balances.

# Isolation levels and its types

Why do we need to learn about isolation level?

→ In systems where multiple transactions are executed concurrently, isolation levels manage the extent to which the operations of one transaction are isolated from those of other transactions.

## Isolation levels help prevent common transactional anomalies:

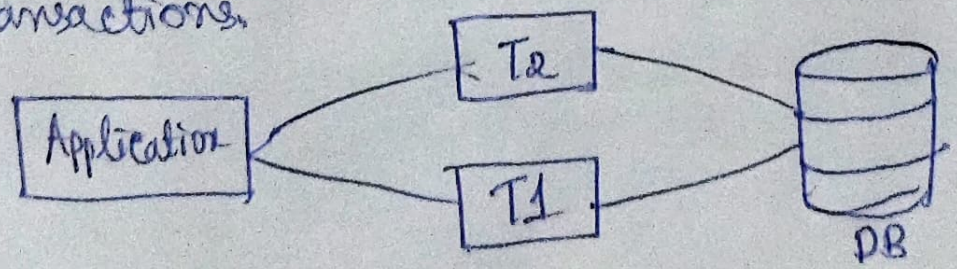
1) Dirty Read: Reading uncommitted data from another transaction.

2) Non-Repeatable Read: Data changes after it has been read within the same transaction.

3) Phantom Read: New rows are added or removed by another transaction after a query has been executed.

| T1 | T2 |
|---|---|
| R | R |
| R | W |
| W | R |
| W | W |

Isolation Level: It determines the degree to which the operations in one transaction are isolated from those in other transactions.



## Anamolies / Violations to Isolation Level →

1) **Dirty Read** → Reading data written by a transaction, that has not yet committed. Consider if T2 reads the data written by T1 and if T1 fails, it becomes irrelevant.

| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) → Dirty Read |
| Failure | |

RollBack

2) **Non-Repeatable Read** → Reading the same row twice within a transaction and getting different values because another transaction modified the row and committed.

Consider if T2 modifies the data which T1 already Read and if T1 continue the transaction the data will be changed.

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| | W(A) |
| | Commit |
| R(A) | |

Data changed / Non-Repeatable Read

3) Phantom Read → Getting different sets of rows in subsequent queries within the same transaction because another transaction inserted or deleted rows and committed.

T1 (Query (id)) → Fetch the name
T2 (Query (id)) → Insert a new entry
T1 (Query (id)) → Fetch the name

different sets of rows/ phantom Read

There are 4 isolation levels which helps us with these anomalies →

Types of Isolation Levels →

1) Read Uncommitted
2) Read Committed
3) Repeatable Read
4) Serializable

1) Read Uncommitted → The lowest isolation level where transactions can see uncommitted changes made by other transactions. If Transaction T1 is writing a value to a table, Transaction T2 can read this value before T1 commits.

a) Dirty Reads : Yes
b) Non-Repeatable Reads : Yes
c) Phantom Reads : Yes

2) Read Committed → It ensures that any data during the transaction is committed at the moment it is read. If T1 has done some write operation T2 can only read data when T1 is committed.

a) Dirty Reads : No
b) Non-Repeatable Reads : Yes
c) Phantom Reads : Yes

3) Repeatable Read → It ensures that if a transaction reads a row, it will see the same value for that row during the entire transaction, even if other transactions modify the data and commit. If transaction T1 reads a value, Transaction T2 cannot modify that value until T1 completes. But T2 can insert ~~so~~ new rows that T1 can see on subsequent reads.

a) Dirty Reads : No
b) Non-Repeatable Reads : No
c) Phantom Reads : Yes

4) Serializable → It ensures a serial transaction execution, so that there is complete isolation. If transaction T1 is executing, Transaction T2 must wait until T1 completes.

a) Dirty Reads : No
b) Non-Repeatable Reads : No
c) Phantom Reads : Yes

Read after Schedules → Concurrent VS Parallel Schedule

| For | Concurrent | Parallel |
|---|---|---|
| Definition | Concurrent scheduling manages multiple tasks at the same time. Tasks can overlap in their execution periods but do not run exactly simultaneously. | Parallel scheduling runs multiple tasks simultaneously using multiple cores or processors. Each task is executed on a separate core or processor at the same time. |
| | | |

| | | |
|---|---|---|
| Execution | Achieved on a single-core processor through context switching, where the CPU rapidly alternates between tasks, creating the illusion of simultaneous execution. | Requires a multi-core processor or multiple processors, with each core/processor handling a different task concurrently. |
| Example | Multi-threading on a single-core CPU, where threads take turns using the CPU | Multi-threading on a multi-core CPU, where threads run concurrently on different cores. |

## Serializability and its Types

Serializability → It ensures that concurrent transactions yield results that are consistent with some serial execution i.e, the final state of the database after executing a set of transactions concurrently should be the same as if the transactions had been executed one after another in some order.

In case of concurrent schedule consistency issue may arise because of non-serial execution and we do serializability there, serial schedules are already serial.

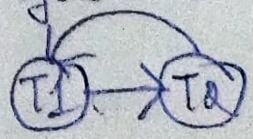
Consider, nodes as transactions, and edges represent conflict and detect if the resulting graph has cycles.

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| Commit | |
| | R(A) |
| | W(A) |
| | Commit |

SERIAL SCHEDULE

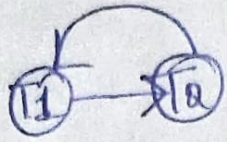
| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| | W(A) |
| W(A) | |

Nodes: T1, T2
Edges: T1 → T2
T2 → T1
Cycle: Yes



A concurrent schedule does not always have a cycle.
A concurrent schedule can be conflict-serializable, meaning that it is equivalent to some serial schedule of transactions and its conflict graph does not have any cycles.

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| | W(A) |
| W(A) | |




Now, since a cycle is detected we need to serialize them. So, we use the serializibility here.
1) Conflict-Serializability → To detect the cycle using conflict graph.
2) View-Serializability → To check if schedule is serializable after a cycle is detected.

CONCURRENT SCHEDULE
Nodes : T1, T2
Edges : T1 → T2, T2 → T1
Cycle : Yes

Why serializing them?
→ To avoid consistency issue which may arise because of non-serial execution.

Types of Serializability →
1) Conflict-Serializability
2) View-Serializability

** Schedule and its Types

Schedule → It refers to the sequence in which a set of concurrent/multiple transactions are executed. You can also say it as a sequence in which the operations (such as read, write, commit, and abort) of multiple transactions are executed. It is really helpful to ensure data consistency and integrity.

If there are T1, T2, ...., TN Transactions, then the possible schedules = n! (n factorial).

Example:3

Schedule SC1 →

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| | W(A) |
| Commit | |
| | Commit |

Incomplete Schedule → Incomplete schedule is a schedule where not all transactions have reached their final state of either commit or abort.

T1: Read (A)
T1: Write (A)
T2: Read (B)
T2: Write (B)
T2: Commit (T2)

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| | Commit |

Here, T1 is still in progress as there is no COMMIT for transaction T1.

Complete Schedule → A complete schedule is one where all the transactions in the schedule have either committed or aborted.

T1: Read (A)
T1: Write (A)
T1: Commit (T1)
T2: Read (B)
T2: Write (B)
T2: Commit (T2)

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| Commit | |
| | R(B) |
| | W(B) |
| | Commit |

## Types of Schedule →

1) Serial Schedule
2) Concurrent or Non-Serial Schedule
3) Conflict-Serializable schedule
4) View-Serializable Schedule
5) Recoverable Schedule
6) Irrecoverable Schedule
7) Cascadeless Schedule
8) Cascading Schedule
9) Strict Schedule

**1) Serial Schedule →** A serial schedule is one where transactions are executed one after another. We can say it like if there are two transactions T1 and T2, T1 should commit to completion before T2 starts.

**Example →** T1 → T2

T1: Read(A)

T1: Write(A)

T1: Commit(T1)
T2: Read(B)
T2: Write(B)
T2: Commit(T2)

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| Commit | |
| | R(B) |
| | W(B) |
| | Commit |

T2 starts only after T1 is completed/commited.

**Advantages →**

1) It follows the ACID properties. Transactions are isolated from each other.
2) It maintains consistency.

**Disadvantages →**

1) Since there is poor throughput (no. of transactions completed / unit time) and memory utilization, this is not suggested as it can be inefficient.

2) Since, wait time is high, less no. of transactions are completed.

2) <u>Concurrent / Non-Serial Schedule</u> → A non-serial schedule is one where multiple transactions can execute simultaneously (operations of multiple transactions are alternate / interleaved executions). We can say it is like, if there are two transactions T1 and T2, T2 doesn't need to wait for T1 to commit, it can start at any point.

<u>Example</u> → T1, T2, T3

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | R(A) | |
| | | R(B) |
| | W(A) | |
| Commit | | |
| | | Commit |

T2 didn't waited for T1 to commit.

<u>Advantages</u>→

1) Better resource utilization.
2) Wait time is not involved. One transaction does not need to wait for the running one to finish.
3) Throughput (no. of transactions completed / unit time) is high.

<u>Disadvantages</u>→

1) Consistency issue may arise because of non-serial execution. It requires robust concurrency control mechanisms to ensure data consistency and integrity.
2) We can use Serializability and Concurrency Control Mechanisms to ensure consistency.

3) Conflict-Serializable Schedule → A schedule is conflict-serializable if it can be transformed into a serial schedule by swapping adjacent non-conflicting operations.

a) R(A) T1 & R(A) T2 : No conflict (both reads)

b) R(A) T1 & W(A) T2 : RW conflict (T2 reads A after T1 writes A)

c) W(A) T1 & W(A) T2 : WW conflict (both write to A)

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| W(A) | |
| | W(A) |
| | W(B) |

Conflict Pairs → Read-Write, Write-Read, Write-Write (performed on the same data item)

Non-Conflict Pairs →
Read-Read (performed on the same data item)
Read-Read (performed on the different data item)
Write-Write (performed on the different data item)
Read-Write / Write-Read (performed on the different data item)

4) View-Serializable Schedule → View Serializability ensures that the database state seen by transactions in a concurrent schedule can be replicated by some serial execution of those transactions.

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| W(B) | |
| | W(A) |

When we find cycle in our conflict graph, we dont know if our schedule is serializable or not, so we use the view serializability here.

A schedule is view serializable if it's view equivalent is equal to a serial schedule / execution.

5) Recoverable Schedule → A recoverable schedule ensures that if a transaction reads data from another transaction, it should not commit until the transaction from which it read has committed. This helps in maintaining the integrity of the database in case a transaction fails.

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| Commit | |
| | Commit |

Recoverable Schedule

6) Irrecoverable Schedule → An irrecoverable schedule allows a transaction to commit even if it has read data from another uncommitted transaction. This can lead to inconsistencies and make it impossible to recover from certain failures.

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | Commit |
| Commit | |

Irrecoverable Schedule

7) Cascading Schedule → This schedule happens when the transaction causes a failure or abortion of one series of other transactions to also abort.

a) T1 Writes to A → T1 writes to data item A.

b) T2 Reads A → T2 reads the uncommitted value of A written by T1.

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| W(A) | | |
| | R(A) | |
| | | R(A) |
| Fail | | |

Cascading Schedule →

Now, If T1 fails and aborts, T2 must also abort because it has read an uncommitted value from T1. Consequently, T3 must abort as well.

Issues →

1) Performance degradation because multiple transactions need to be rolled back.

2) Improper CPU resource utilization.

8) Cascadeless Schedule → It ensures transactions only read committed data, such that the abortion of one transaction does not lead to the abortion of other transactions that have already read its uncommitted changes.

a) T1 Writes to A → T1 writes to data item A.

b) T2 Reads A → T2 reads the committed value of A. Ensuring there is no write read problem (dirty read). Also, T2 does not read any uncommitted data, there are no cascading aborts in this schedule.

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| W(A) | | |
| Commit | | |
| | R(A) | |
| | | R(A) |

Cascadeless Schedule

Issues →

1) Write - Write problem is encountered.

2) Performance overhead is there.

9) Strict Schedul  It ensures transaction is not allowed to read or write a data item that another transaction has written until the first transaction has either committed or aborted. It prevents cascading aborts.

1) T2 cannot read or write the value of A until T1 has committed.

2) This ensures that T2 only uses the committed value of A.

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| Commit | |
| | R(A) |
| | W(A) |
| | Commit |

Strict Schedule

## Conflict-Serializability

Serializability: Serializability is a property of a schedule where the outcome is equivalent to some serial execution of the transactions.

Conflict-Serializability: A schedule is said to be conflict serializable when one of its conflict equivalent is serializable.
So basically, if a schedule can be transformed into a serial schedule by swapping non-conflicting operations, then the schedule is conflict serializable.

Conflict equivalent: If a schedule S1 is formed after swapping adjacent non-conflicting operations/pairs in a given schedule S, then S1 and S are conflict equivalent.

Conflict pairs: Read-Write, Write-Read, Write-Write (performed on the same data item)

Non-conflict pairs: Read-Read (performed on same or different data items)
Read-Write, Write-Write, Write-Read (performed on different data items)

Non-Conflict pairs

| | |
|---|---|
| R(A) | R(B) |
| R(A) | R(A) |
| W(A) | R(B) |
| W(A) | W(B) |
| R(A) | W(B) |

Conflict pairs

| | |
|---|---|
| R(A) | W(A) |
| W(A) | R(A) |
| W(A) | W(A) |

How to achieve conflict equivalent schedule?

→ when a schedule can be transformed into a serial schedule by swapping adjacent non-conflicting operations. Conflicts arise when two transactions access the same data item, and at least one of them is a write operation.

Now, why are we only swapping the non-conflict pairs and not the conflict ones?

→ So if we swap the conflict pairs, the order of execution, if it was

T1: R(A)

T2: W(A)

the results values may change as first we were reading A and then writing/modifying A, now it will be writing A and then reading the modified value, so the result might change if we change the order of execution.

Q Find a conflict equivalent for a Schedule S1.

→ S1:

| T1 | T2 |
|---|---|
| R(X) | |
| W(X) | |
| | R(X) |
| R(Y) | |
| W(Y) | |
| | R(Y) |

→ 1) Find the adjacent non-conflicting pairs and do a swap.
T2: R(X) T1: R(Y)

| T1 | T2 |
|---|---|
| R(X) | |
| W(X) | |
| | R(X) |
| R(Y) | |
| W(X) | |
| | R(Y) |

⇒

| T1 | T2 |
|---|---|
| R(X) | |
| W(X) | |
| R(Y) | |
| | R(X) |
| W(Y) | |
| | R(Y) |

2) After the first swap again search for adjacent non-conflicting pairs and swap if any.
T2: R(X) T1: W(Y)

| T1 | T2 |
|---|---|
| R(X) | |
| W(X) | |
| R(Y) | |
| | R(X) |
| W(Y) | |
| | R(Y) |

⇒

| T1 | T2 |
|---|---|
| R(X) | |
| W(X) | |
| R(Y) | |
| W(Y) | |
| | R(X) |
| | R(Y) |

: S1'

So, S1 is a serializable schedule as S1' has a serial execution (i.e, it is conflict equivalent).

How to check whether a schedule is conflict serializable or not?

→ Conflicts occur when two operations from different transactions access the same data item and atleast one of them is a Write operation.

<u>Conflict Graph/Precedence Graph</u> → A conflict graph, or precedence graph, is a directed graph used to determine conflict serializability. The nodes represent transactions, and the edges represent conflicts between transactions.

<u>Conflict Graph / Precedence Graph</u> →

1) <u>Nodes</u> → Each transaction in the schedule is represented as a node in the graph.

2) <u>Edges</u> → An edge from transaction T(X) to transaction T(Y) (denoted by T(X) → T(Y)) is added if →

a) an operation of T(X) conflicts with an operation of T(Y)

b) T(X) operation proceeds T(Y) operation in the schedule.

3) <u>Cycle Detection</u> → The schedule is conflict-serializable if and only if the conflict graph is acyclic. If there are no cycles in the graph, it means that the schedule can be serialized without violating the order of conflicting operations.

Q Check if the given schedule is conflict-serializable or not?

→

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | R(A) | |
| W(A) | | |
| | | W(A) |
| | W(B) | |
| | | R(B) |

- T1 reads A
- T2 reads A
- T1 writes A
- T3 writes A
- T2 writes B
- T3 writes B

→ Step-1 → Find the conflicts in the schedule.

1) RW conflict (T1, T3) on A
2) RW conflict (T2, T3) on A
3) RW conflict (T2, T1) on A
4) WW conflict (T1, T3) on A
5) WR conflict (T2, T3) on B

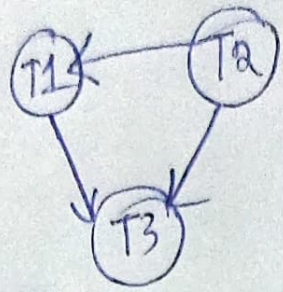
Step-2 → Find the nodes.

Each transaction T1, T2, T3 is a node in the graph

Step-3 → Find the edges/conflicts.

1) T1 → T3 : T1 reads A before T3 writes A.
2) T2 → T3 : T2 reads A before T3 writes A.
3) T1 → T3 : T1 writes A before T3 writes A.
4) T2 → T3 : T2 writes B before T3 reads B.
5) T2 → T1 : T2 reads A before T1 writes A.

Step-4 → Draw the graph.




Step-5 → If the graph has cycle it is not conflict-serializable or serializable, if not lets find the serial execution of transactions.

Since the conflict graph is acyclic, the schedule is conflict-serializable.

Step-6 → Lets find the serial execution of transactions.

Possible combinations are →

T1 → T2 → T3

T1 → T3 → T2

T2 → T1 → T3

T2 → T3 → T1

T3 → T1 → T2

T3 → T2 → T1

Find the indegree (the number of edges directed into that node) and if its 0 it can be the first in serial execution.

T1 → 1, T2 → 0, T3 → 2, T2 would be the first as indegree is 0.

1) T2 must precede T1.

2) T1 must precede T3.

Therefore, one possible equivalent serial schedule is T2 → T1 → T3.

# View - Serializability

View serializability ensures that the database state seen by transactions in a concurrent schedule can be replicated by some serial execution of those transactions.

When we find cycle in our conflict graph, we don't know if our schedule is serializable or not, so we use the view serializability here.

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| W(A) | |
| | W(A) |

A schedule is view serializable if it's view equivalent is equal to a serial schedule/execution.

## Conditions for a View Equivalent schedule →

For a schedule to be view-Equivalent it should follow the following conditions.

1) Initial Read Values → An initial read of both schedules must be identical. For example, consider two schedules, S and S'. If in schedule S, transaction $T_2$ reads the data item A, then in schedule S', transaction $T_2'$ should also read A.

S

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(A) | |
| | | W(B) |

S'

| T1 | T2 | T3 |
|---|---|---|
| | W(A) | |
| R(A) | | |
| | | W(B) |

2) Updated Read → In a schedule S, if $T_i$ is reading B, which is updated by $T_j$ then in S' also, $T_i$ should read B which is updated by $T_j$.

| T1 | T2 | T3 |
|---|---|---|
| | W(B) | |
| | | R(B) |
| R(A) | | |

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(B) | |
| | | R(B) |

3) Final Update → A final write must be identical in both schedules. If, in schedule S, transaction $T_i$ is the last to update A, then in schedule S', the final write operation on A should also be performed by $T_i$

| T1 | T2 | T3 |
|---|---|---|
| W(A) | | |
| | R(A) | |
| | | W(A) |

| T1 | T2 | T3 |
|---|---|---|
| | R(A) | |
| W(A) | | |
| | | W(A) |

The number of possible serial schedules for a n transaction is given by the number of permutations of the transactions:
n!

Q Find the view equivalent schedule.

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(A) | |
| W(A) | | |
| | | W(A) |

S:

→ Step-1 → Find if conflict serializable or not. (No)

Step-2 → Find the possible serial schedules → 3!

Step-3 → choose one possibility and check for view equivalent conditions (T1 → T2 → T3).

1) Initial Read → T1

2) Update Read → No read performed after update so no need to check.

3) Final Update → T3

Step-4 → If the view equivalent schedule matches the condition, it is view serializable.

→ Hence it is view serializable and serializable as well.

T1 → T2 → T3

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| W(A) | | |
| | W(A) | |
| | | W(A) |

S:

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(A) | |
| W(A) | | |
| | | W(A) |

# Concurrency control mechanisms

Some common concurrency control mechanisms are →

1) Lock-Based Protocols (2PL, Strict 2PL, Rigorous 2PL, Conservative 2PL)
2) Timestamp-Based Protocols (Basic Timestamp Ordering (TO))
3) Optimistic Concurrency Control (OCC)
4) Multiversion Concurrency Control (MVCC)

## 1) Lock-Based Protocols →

### Types of Locks →

a) Binary Locks → A simple mechanism where a data item can be either locked (in use) or unlocked. If a thread tries to acquire the lock when it's ~~also~~ already locked, it must wait until the lock is released by the thread currently holding it.

b) Shared and Exclusive Locks →

i) Shared Lock (S-Lock) → Allows multiple transactions to read a data item simultaneously, but prevents any of them from modifying it. Multiple transactions can hold a shared lock on the same data item at the same time.

ii) Exclusive Lock (X-Lock) → Allows a transaction to both read and modify a data item. When an exclusive lock is held by a transaction, no other transactions can read or modify the data item.

| T1 | T2 | |
|---|---|---|
| X(A) | | → Exclusive Lock |
| R(A) | | |
| W(A) | | |
| U(A) | | |
| | S(B) | → Shared lock |
| | R(B) | |
| | U(B) | → unlock |

NOTE → When a transaction acquires a shared lock on a data item, other transactions can also acquire shared locks on that same data item, enabling concurrent reads. However, no transaction can acquire an exclusive lock on that item as long as one or more shared locks are held.

When a transaction acquires an exclusive lock on the data item, it has full control over that item, meaning it can both read and modify it. No other transaction can acquire a lock on the same data item until the exclusive lock is released.

Shared lock → any no. of transactions.

Exclusive lock → Only one transaction should hold it at one time

While shared and exclusive locks are vital for maintaining data integrity and consistency in concurrent environments, they can introduce significant challenges in terms of performance, deadlocks, reduced concurrency, and system complexity.

Drawbacks of Shared-Exclusive locks →

Performance Issues → Managing locks requires additional CPU and memory resources. The process of acquiring, releasing, and managing locks can introduce significant overhead.

Concurrency Issues → Exclusive locks prevent other transactions from accessing locked data, which can significantly reduce concurrency.

Starvation → Some transactions may be delayed if higher-priority

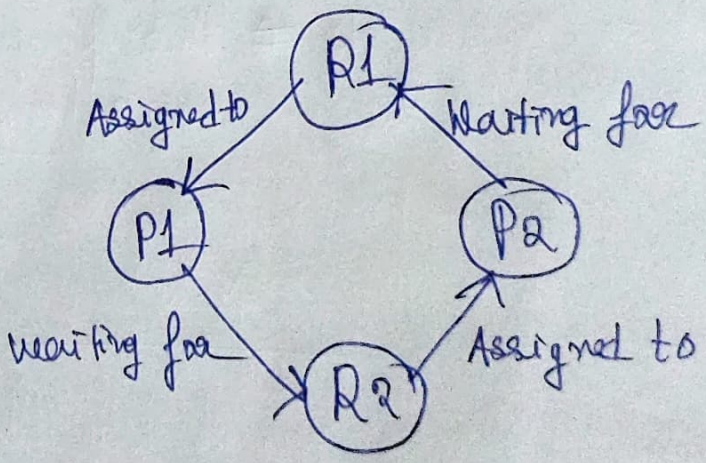
transactions consistently acquire locks before them, leading to situation where a transaction never gets to proceed.

**Deadlocks** ⇒ Shared and Exclusive locks can lead to deadlocks, where two or more transactions hold locks that other transactions need.

**Irrecoverables** ⇒ If a transaction B commits after the lock is released based on a modified value in transaction A which fails after sometime.

**Deadlock** ⇒ It is a situation when 2 or more transactions wait for one another to give up the locks.

| T1 | T2 |
|---|---|
| X(A) | |
| R(A) | |
| W(A) | |
| | X(B) |
| | R(B) |
| | W(B) |
| X(B) | |
| R(B) | |
| W(B) | |
| | X(A) |
| | R(A) |
| | W(A) |




**Two-Phase Locking (2PL)** ⇒ This protocol ensures serializability by dividing the execution of a transaction into two distinct phases.

1) **Growing Phase** ⇒ A transaction can acquire locks but cannot release any. This phase continues until the transaction

has obtained all the lock it needs.

2) Shrinking Phase → After the transaction releases its first lock, it can no longer acquire any new locks. During this phase, the transaction releases all the locks it holds.

Any transaction which is following 2PL locking achieves serializability and consistency in the given schedule.

| T1 | T2 |
|---|---|
| X(A) | Growing phase |
| R(A) | |
| W(A) | |
| | R(A) → wait |
| S(B) | |
| R(B) | |
| U(A) | Shrinking phase |
| U(B) | |

→ now a shared lock can be used by T2 to read A

Advantages →

1) It guarantees that the schedule of transactions will be serializable, meaning the results of executing transactions concurrently will be the same as if they were executed in some serial order.

2) By ensuring that transactions are serializable, 2PL helps maintain data integrity and consistency, which is critical in environments where data accuracy is essential.

Disadvantages →

1) Deadlocks, starvation and cascading rollbacks

2) Transactions must wait for locks to be released by other transactions. This can lead to increased waiting times and lower system throughput.

3) In case of system failure, recovering from a crash can be complex.

Strict Two-Phase Locking (Strict 2PL) →

A strict variant where exclusive locks are held until the transaction commits or aborts. This helps prevent cascading rollbacks

(where one transactions's rollback causes other transactions to roll back).

| T1 | T2 |
|---|---|
| X(A) | Growing phase |
| R(A) | |
| W(A) | |
| | D(A) → waiting |
| commit U(A) | shrinking phase |

→ commit is done before releasing the locks.

Advantages →

1) Prevents Cascading Aborts.
2) Ensures Strict Serializability.

Disadvantages →

1) Since write locks are held until the end of the transaction, other transactions may be blocked for extended periods.
2) Transactions may experience longer wait times to acquire locks.
3) Deadlocks and Starvation is there.

Rigorous Two-Phase Locking →

An even stricter version where all locks (both shared and exclusive) are held until the transaction commits. This guarantee strict serializability.

Advantages →

1) Since all locks are held until the end of the transaction, the system can easily ensure that transactions are serializable and can be recovered.
2) Prevents cascading Aborts and Dirty Reads.

Disadvantages →

1) Performance bottlenecks.
2) Increased Transaction Duration.
3) Deadlocks and Starvation is there.

Conservative Two-Phase Locking →

Conservative Two-Phase Locking (Static Two-Phase Locking) is a variant of the standard 2PL protocol that aims to prevent deadlocks entirely by requiring a transaction to acquire all the locks it needs before it begins execution.

If the transaction is unable to acquire all the required locks (because some are already held by other transactions), it waits and retries. The transaction only starts execution once it has successfully acquired all the necessary locks.

Since a transaction never starts executing until it has all the locks it needs, deadlocks cannot occur because no transaction will ever hold some locks and wait for others.

In this scenario, deadlocks cannot occur because neither T1 nor T2 starts execution until it has all the locks it needs.

## Timestamp-Based Protocols :-

It assigns a unique timestamp.

1) Every transaction is assigned a unique timestamp when it enters the system. It is used to order the transactions based on their timestamps.

2) There are two important timestamps for each data item →

a) Read Timestamp (RTS) → The last timestamp of any transaction that has successfully read the data item.

b) Write Timestamp (WTS) → The last timestamp of any transaction that has successfully written the data item.

Transaction with smaller timestamp (Old) → Transaction with larger timestamp (young)

Rules → Consider if there are transactions performed on data item A.

R_TS (A) → Read time-stamp of data-item A.

W_TS (A) → Write time-stamp of data-item B.

1) check the following condition whenever a transaction $T_i$ issues a Read (X) operation:

a) If $W_TS(A) > TS(T_i)$ then the operation is rejected. (rollback $T_i$)

b) If $W_TS(A) <= TS(T_i)$ then the operation is executed. (Set $R_TS(A)$ as the max of $(R_TS(A), TS(T_i))$)

2) check the following condition whenever a transaction $T_i$ issues a Write (X) operation:

a) If $TS(T_i) < R_TS(A)$ then the operation is rejected. (rollback $T_i$)

b) If $TS(T_i) < W_TS(A)$ then the operation is rejected and $T_i$ is rolled back otherwise operation is executed. (Set $W_TS(A) = TS(T_i)$)

Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 7:00 PM and transaction T2 has entered the system at 7:05 PM, then T1 has the higher priority so it executes first as it is entered the system first. T1 → T2. If younger has done a Read/Write and older wants to do Read/Write, that a rollback case.

| | T1 | T2 |
|---|---|---|
| 1 | R(A) | W(A) |
| 2 | W(A) | R(A) |
| 3 | W(A) | W(A) |

→ No issues.

| | T1 | T2 |
|---|---|---|
| 4 | W(A) | R(A) |
| 5 | W(A) | W(A) |
| 6 | R(A) | W(A) |

Rad Back

Read before concurrency control

# Concurrency Control Mechanisms

**Concurrency control** → It is a process of managing multiple user's access and modification in data simultaneously in shared or multi-user database system.

How it helps?

1) **Data Consistency** → Ensures that the data remains accurate and reliable despite concurrent access.

2) **Isolation** → Maintains the isolation property of transactions. So the outcome of a transaction is not affected by other concurrently executing transactions.

3) **Serializability** → Ensures that the result of concurrent transactions is the same as if the transactions had been executed serially.

**Concurrency control mechanisms** → Techniques used in DBMS to ensure that transactions are executed concurrently without leading to inconsistencies in the database. In serializability, we check if a schedule is serializable or not but in concurrency control, we use certain techniques to make a schedule serializable.

Why is it needed?

1) **Lost Updates** → When two or more transactions update the same data simultaneously, one of the updates might be lost. For example, if two users modify the same record at the same time, the changes made by one user could overwrite the changes made by the other. (WW)

2) **Dirty Reads** → When a transaction reads data that has been modified by another transaction but not yet committed. If the first transaction rolls back, the other transaction will have read invalid data. (WR)

3) Uncommitted Dependency (Dirty Write) → when a transaction modifies data that has been read by another transaction, leading to inconsistencies if one of the transaction rolls back.

4) Inconsistent Retrievals (Non-Repeatable Reads) → Happens when a transaction reads the same data multiple times and gets different values each time because another transaction is modifying the data in between the reads.

5) Phantom Reads → Occurs when a transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that affect the set before the first transaction completes. This results in the first transaction reads different sets of rows, if it re-executes the query.

Some common concurrency control mechanisms are →

1) Lock-Based Protocols (2PL, Strict 2PL)
2) Timestamp-Based Protocols (Basic Timestamp Ordering (TO), Thomas's Write Rule)
3) Optimistic concurrency control (OCC)
4) Multiversion Concurrency Control (MVCC).

## Database Recovery Management

It involves strategies and processes to restore a database to a consistent state after a failure or crash.

### Types of Database Failures →

1) Transaction Failure → Occurs when a transaction cannot complete successfully due to logical errors or system issues (like deadlocks).

2) System Failure → Occurs when the entire system crashes due to hardware or software failures, leading to loss of in-memory data.

3) Media Failure → Occurs when the physical storage (e.g. hard drives) is damaged resulting in data loss or corruption.

## Recovery Phases →

1) Analysis Phase → Identifies the point of failure and the transactions that were active at that time.

2) Redo Phase → Reapplies changes from committed transactions to ensure the database reflects all completed operations.

3) Undo Phase → Reverts the effects of incomplete transactions to maintain consistency.

## Recovery Techniques →

1) Backup and Restore → Regular backups are taken to ensure data can be restored. Full, incremental, and differential backups are common types.

2) Logging → Keeps a record of all transactions. The Write-Ahead Logging (WAL) protocol ensures the logs are written before any changes are applied to the database.

3) Shadow Paging → Maintains two copies of the database pages, one is the updated, and the other remains unchanged until the transaction commits.

## Query Optimization

Whenever we want to improve the efficiency of a query, make it execute faster and consume fewer resources, we use query optimization techniques.

Optimizing SQL queries is essential for improving the performance of database applications.

1) Use Indexes Efficiently → Index the columns frequently used in WHERE, JOIN, ORDER BY, and GROUP BY clauses to speed up data retrieval.

2) Select Only Necessary Columns → Avoid SELECT *, specify only the columns you need in your query. This reduces the amount of data transferred and processed.

3) Optimize JOIN operations → Choose the Right JOIN type and use indexed columns in JOIN conditions.

4) Partition Large Tables → For very large tables, consider partitioning them to improve query performance. Partitioning allows the DBMS to scan only relevant partitions, reducing I/O and improving response time.

5) Cache results → Cache the results of frequently executed queries to avoid the redundant calculations.

Physical Storage and File Organization

Physical Storage →

Storage device → For storing the data, there are different types of storage options available.

1) Primary Storage → Main memory and cache (fastest and expensive and as soon as the system leads to a power cut or a crash the data also gets lost).

2) Secondary Storage → Flash Memory and Magnetic Disk Storage (non-volatile); save and store data permanently.

3) Tertiary Storage → Optical disk and Magnetic tape (used for data backup and storing large data offline).

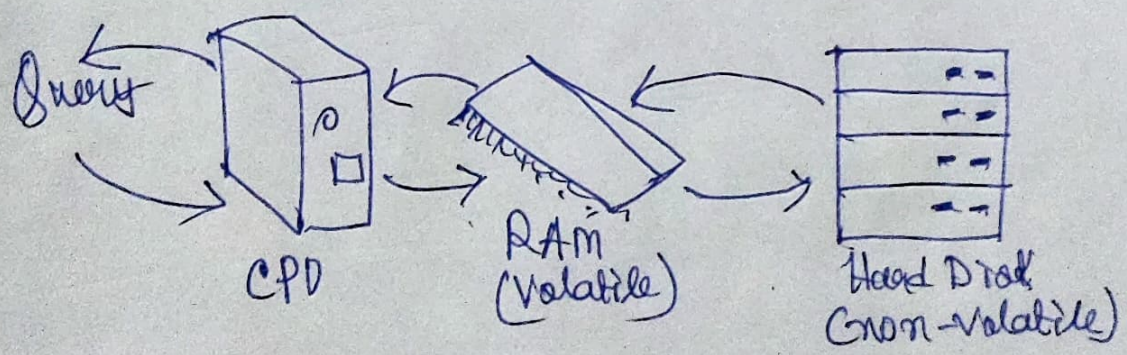
Databases store data in files on disk. Each table or index may correspond to one or more files. Each file has sequence of records.

Accessing data from RAM is much faster than accessing data from a hard disk or SSD. This is because RAM is designed for high-speed read and write operations. Data retrieval in RAM typically involves fetching data in nanoseconds.

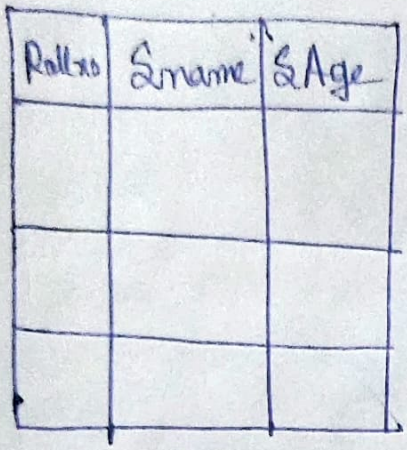
1) RAM => Provides fast, volatile memory for quick data retrieval. Data is accessed via addressable locations and cache memory enhances speed.

2) Hard Disk => Provides slower, non-volatile storage with mechanical components that impact access speed. Data retrieval involves moving the read/write head and waiting for disk rotation.

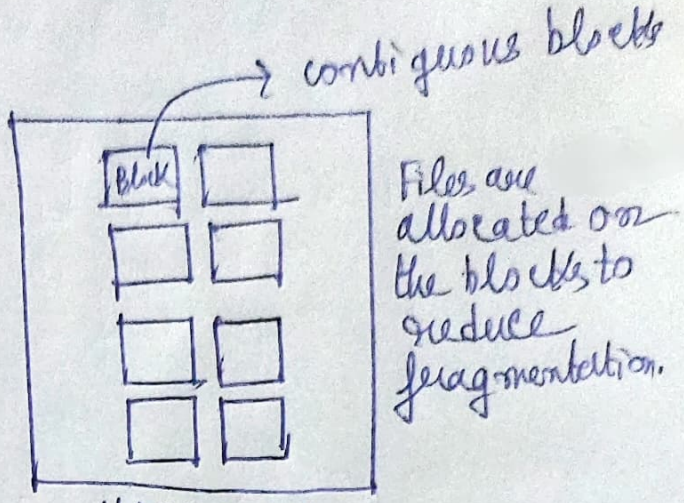
When a program or application needs to access data, it first checks if the data is available in RAM. If the data is not in RAM, it must be loaded from a slower storage device like a hard disk or SSD. Index files in a DBMS are stored on disk but are also managed in RAM to optimize performance.



How file is organized?




DB




HARD DISK

No. of records stored in each block = size of block / size of records

Records can be stored in 2 ways → sorted (searching is fast) or unsorted (searching is slow, insertion is fast).

## Indexing and its types

Why do we need indexing?

→ Imagine you have a 1000-page textbook and you want to search a topic "xyz".

1) when indexing is not present → You need to manually search for each term in the book, which could be frustrating and time-consuming

2) when indexing is present → The index allows you to quickly locate each topic. You can look up "xyz", find that it's covered on pages 445–490. This makes searching more efficient and less time-consuming.

Indexing → is a critical technique in database management that significantly improves the performance of query operations by minimizing the no. of query operations. Index table is always sorted.

Why do we need indexing?

→ In the same way how we have used indexing in book, it helps us to find specific records or data entries in a large table or dataset.

| Search Key | Data access |
|---|---|
| It is used to find the record. | It contains the address where the data item is stored in memory for the provided search key. |

Index table

It helps in finding what a user is looking for (P.K OR C.K)

It tells where the record is stored in the memory (set of pointer holding the address of blocks).

How it helps?

1) Improved Query Performance → Indexes allow the database management system (DBMS) to locate and retrieve data much more quickly than it could by scanning the entire table.

2) Indexes can be used to optimize queries that sort data (ORDER BY) or group data (GROUP BY) → when an index is available on the columns being sorted or grouped, the DBMS can retrieve the data in the desired order directly from the index, eliminating the need for additional sorting operations.

3) We have certain indexing methods like Dense (index entry for all the records) and Sparse (index entries for only a subset of records) index.

**Sparse Index** → A sparse index is a type of database indexing technique where the index does not include an entry for every single record in the database. Instead, it contains entries for only some of the records, typically one entry per block (or page) of data in the storage. This makes sparse indexes smaller and faster to search through compared to dense indexes, which have an entry for every record.

It is used when we have an ordered data set.

**Example** → If a table has 1000 rows divided into 100 blocks, and you create a sparse index, the index might only have 100 entries, with each entry pointing to the first record in each block.

Index Table

| Search Key | data ref |
|---|---|
| 1 | B1 |
| 4 | B2 |

Sparse Index

| | |
|---|---|
| 1 Ram<br>2 Shyam<br>3 | B1 |
| 4<br>5<br>6 | B2 |

disk

| Roll No | Name |
|---|---|
| 1 | Ram |
| 2 | Shyam |
| 3 | Raghu |
| 4 | Riti |
| 5 | Raj |
| 6 | Rahul |

no. of records in index file = no. of blocks

**How Sparse Indexing Works** →

1) **Primary Index** → Sparse indexes are often used with primary indexes, where the table is sorted on the indexed column. The sparse index only includes an entry for the first record in each block or page.

2) **Searching** → When searching for a specific value, the database uses the sparse index to quickly locate the block where the record might reside. It then searches within the block to find the exact record.

## Disadvantages of Sparse Indexes →

1) Additional I/O operations → After using the index to find the correct block, an additional search value within the block is required to find the specific record.

2) Less Efficient for Random Access → If queries often need to retrieve random, non-sequential records, a dense index might be more efficient.

## Use Cases →

1) Primary Indexing on Large Tables → Sparse indexes are ideal for primary indexes on large tables where records are sequentially stored.

2) Range Queries → Sparse indexes can be effective for range queries where the query needs to retrieve a range of records rather than a single record.

**Dense Index** → A dense index is a type of indexing technique used in databases where there is an index entry for every single record in the data file. This means that the index contains all the search keys and corresponding pointers (or addresses) to be actual data records, making it highly efficient for direct lookups.

It is used when we have an un-ordered data set.

**Example** → Consider a table with 1000 rows, and you create a dense index on a non-primary key column. The dense index will have no. of unique non-key records, each pointing to a specific row in the table.

These are useful in scenarios where random access to individual records is quite frequent.

| Search Key | data ref |
|---|---|
| 18 | B1 |
| 19 | B1 |
| 20 | B2 |
| 21 | B2 |

Index Table

| | |
|---|---|
| 18 Ram<br>18 Shyam<br>19 | B1 |
| 20<br>20<br>21 | B2 |

Dense Index no. of records in index file = no. of unique non-Key records

| Age | Name |
|---|---|
| 18 | Ram |
| 18 | Shyam |
| 19 | Raghu |
| 20 | Riti |
| 20 | Raj |
| 21 | Rahul |

## How Dense Indexing Works →

In a dense index, each index entry includes →

1) A Search Key → the value of the indexed column.

2) A Pointer → (or address) to the actual record in the data file.

When a query searches for a specific value, the database uses the dense index quickly locate the corresponding record.

## Disadvantages of Dense Indexes →

1) Storage Intensive → Requires significant storage space because it maintains an entry for every record in the table.

2) Maintainence Overhead → Inserting, Updating, or Deleting records requires updating the dense index, which which can be costly in terms of performance, especially for large tables.

## Use Cases →

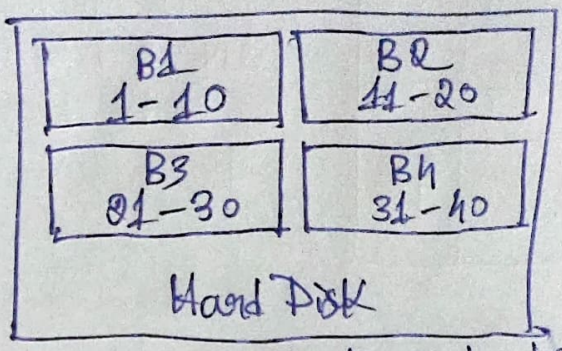
1) Primary Indexing → Dense indexes are often used for primary indexing when the table is small to medium-sized,

and quick access to individual records are required.

2) Exact Match Queries → Ideal for situations where queries frequently requests for individual records based on an exact key match.

Index Table

| Search Key | data ref |
|---|---|
| 1 | B1 |
| 11 | B2 |
| 21 | B3 |




When a query is run, the database engine checks the index to find the pointers to the rows that contain the desired data. It then retrieves these rows directly without scanning the entire table.

How indexing works?

→ It takes a search key as an input and then returns a collection of all the matching records. Now in index there are 2 columns. the first one stores a duplicate of the key-attributes / non-key attribute (search key) from table while the second one stores the pointer which holds the disk block address of the corresponding key-value.

Indexing

Clustered Index
1) Primary Index
when key = P.K
order = sorted
Follows sparse indexing

Multilevel Indexing
when there is one more level of indexing in the index table we use inner and outer index.
Mostly used for large files.

Non-Clustered Index
1) Secondary Index
when key = non-key / key
order = unsorted
Follows dense indexing

2) Clustering Index

When Key = non-Key
Order = sorted
Follows dense indexing.

Key Data → Unique identifier for each record. Often used to distinguish one record from another.

Non-Key Data → All other data in the record besides the Key.

Single-Level Indexing → It is straightforward and works well for smaller databases, offering a direct approach to speeding up query performance.

Multi-Level Indexing → It is more complex but necessary for larger databases, as it effectively manages large indexes by creating additional layers of indexing, thereby improving data retrieval times.

## Primary Index

It enhances the efficiency of retrieving records by using the primary key values. It establishes an index structure that associates primary key values with disk block addresses. This index is composed of a sorted list of primary key values paired with their corresponding disk block pointers, thereby accelerating data retrieval by reducing search time. It is especially beneficial for queries based on primary keys. It is done on sorted data.

| Search Key | data ref |
|---|---|
| 1 | B1 |
| 4 | B2 |

It creates an index structure that maps primary key values to disk block addresses.

Index Table

| | |
|---|---|
| 1<br>2<br>3 | B1 |
| 4<br>5<br>6 | B2 |

Sparse Index

no. of records in index file = no. of blocks

| Roll No | Name |
|---|---|
| 1 | Ram |
| 2 | Shyam |
| 3 | Raghu |
| 4 | Riti |
| 5 | Raj |
| 6 | Rahul |

Examples: In a Users table with a UserID column as the P.K, a primary index on UserID allows for quick retrieval of user records when you know the UserID.

The primary index ensures that the database can immediately locate the row associated with a given USERID, making operations like SELECT, UPDATE, and DELETE highly efficient.

## Cluster Index

The cluster index is generally done on a non-key attribute. The data in the table is typically ordered according to the order defined by the clustered index. Mostly, each index entry points directly to a row. It is mostly used where we are using GROUP BY. It physically order records in a table based on the indexed columns.

| Search Key | data ref |
|---|---|
| 18 | B1 |
| 19 | B1 |
| 20 | B2 |
| 21 | B2 |

Index Table

| | |
|---|---|
| 18<br>18<br>19 | B1 |
| 20<br>20<br>21 | B2 |

Dense Index

no. of records in index file = no. of blocks
unique key records

| Age | Name |
|---|---|
| 18 | Ram |
| 18 | Shyam |
| 19 | Raghu |
| 20 | Riti |
| 20 | Raj |
| 21 | Rahul |

Example → In an Employee table, if you need to produce lists of employees ordered by lastName, creating a clustered index on LastName can make these queries faster.

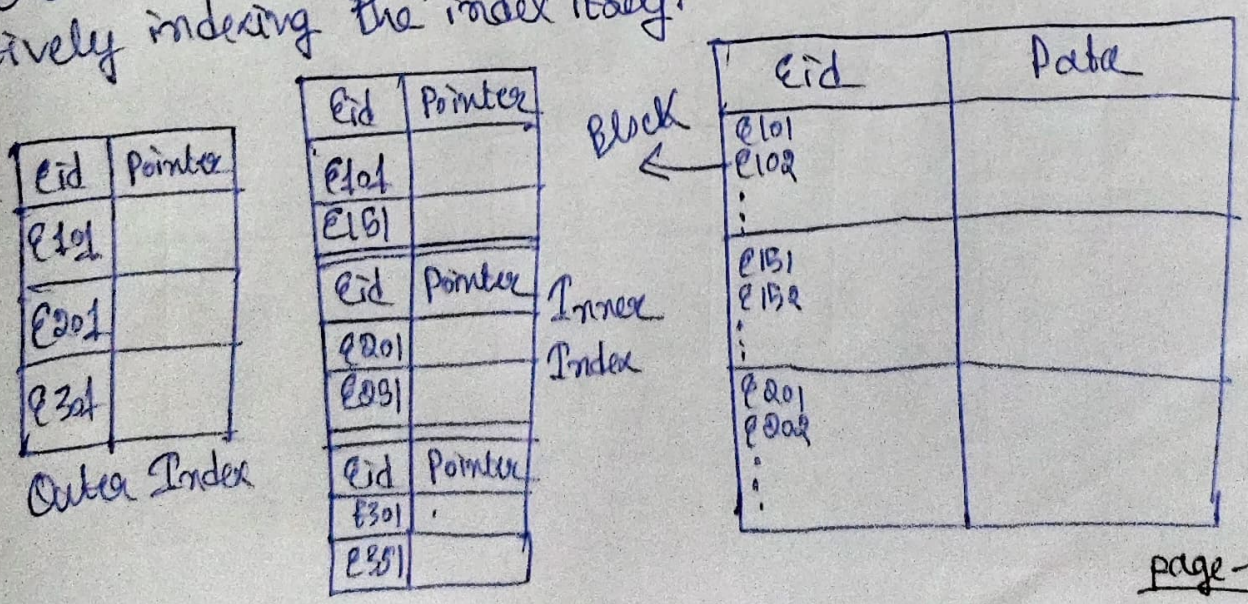
The clustered index keeps the rows in LastName order, reducing the need for the database to perform additional sorting when executing queries.

## Multilevel Index

A multilevel index is an advanced indexing technique used in database management systems to manage large indexes more efficiently. When a single-level index becomes too large for fitting into the memory, multilevel indexing helps by breaking down the index into multiple levels, reducing the number of disk I/O operations needed to search throughout the index.

1) First Level (Primary Index) → The first level is the original index, where each entry points to a block of data or a page in the table.

2) Second Level (Secondary Index) → If the first level index is too large to fit in the memory, a second-level index is created. This index points to the block of first level index, effectively indexing the index itself.



# Secondary Index

Secondary indexing speeds up searches for non-Key columns in a database. Unlike primary indexing, which uses the primary Key, Secondary indexing focuses on other columns. It creates an index that maps column values to the locations where the records are stored, making it faster to find specific records without scanning the entire table.

The data is mostly unsorted and it can be performed on both Key and non-Key attributes.

1) when search Key is Key attribute and data is unsorted.

**Dense Index**

no. of records in index file = no of blocks unique Key records.

| Search Key | data ref |
|---|---|
| 18 | B1 |
| 20 | B1 |
| 22 | B2 |
| 24 | B2 |
| ⋮ | ⋮ |

Index table

| Age | |
|---|---|
| 18<br>20<br>40<br>07 | B1 |
| 22<br>24<br>30 | B2 |

2) when search Key is non-Key attribute and data is unsorted.

**Dense Index**

no. of records in index file = no. of blocks unique non-Key records

| Search Key | |
|---|---|
| 18 | B1 |
| 20 | B1 |
| 22 | B1 → B2 |
| 24 | B2 |
| ⋮ | ⋮ |

Index Table

Linked List

| Age | |
|---|---|
| 18<br>18<br>20<br>07<br>22 | B1 |
| 22<br>24<br>30 | B2 |

How to create indexes?

→ Non-clustered index

→ CREATE INDEX index_name
ON table_name(column_name);

↳ (On one column)

→ CREATE INDEX index_name
ON table_name(col1, col2, .....);

↳ (On multiple column)

Remove an index

DROP INDEX index_name ON
table_name;

Clustered index

Automatically created with the P.K. MySQL does not support explicit creation of additional clustered indexes.

B and B+ Trees

Clustered and non-clustered indexes are concepts that describe how data is stored and accessed in a database. But B-trees (and B+ trees) are the data structures that actually implement these indexes. Knowing B-trees gives you insight into how these indexes work under the hood.

Understanding B-trees helps you understand why certain queries perform well or poorly based on the structure of the index. For example, how a B-tree's balanced nature affects search times or why range queries are efficient with B-tree indexes.

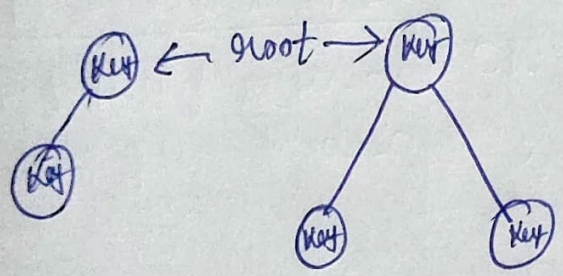
B-trees provide the balanced, efficient structure that makes these type of indexes performant, ensuring that operations like search, insert, delete, and update are done in logarithmic time ($O(\log n)$). This makes B-tree indexing crucial for optimizing database queries, whether in clustered or non-clustered index scenarios.

# B-trees (M-way tree)

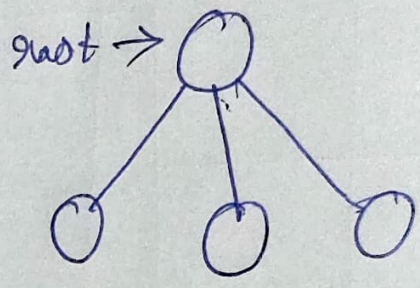
B-trees are self-balancing tree data structures that maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time. All leaf nodes are at the same level.

In B-trees you can have minimum 2 children and max X children.

Now B-tree is a generalization of Binary Search trees. In BST, every node can have atmost 2 children (0, 1, 2) and only one Key.



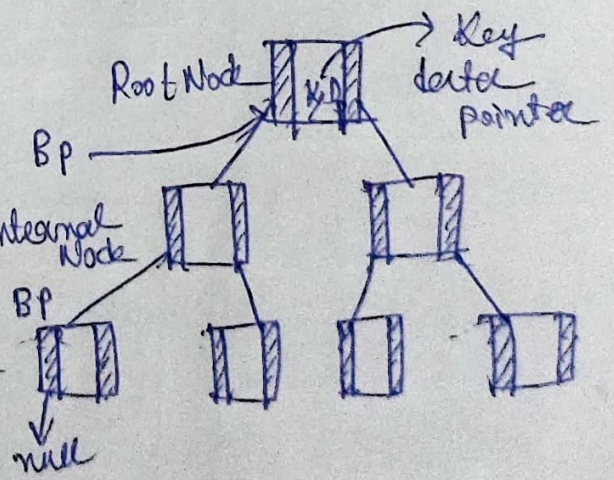

Binary tree




B- tree

**Rp / Dp** → record/data pointer (where the record is present in secondary memory (disk)).

**Bp / Tp** → block/tree pointer (links to the children nodes).



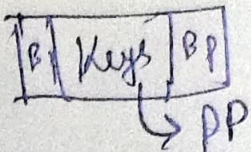

## Structure of B-Tree →

1) **Nodes** → A B-tree is composed of nodes, each containing Keys and pointers (references) to child nodes. The Keys within a node are sorted in ascending order.

2) **Root, Internal, and Leaf Nodes** →

a) The root node is the topmost node in the tree.

b) Internal nodes contain Keys and child Pointers.




c) Leaf Nodes contain Keys and possibly pointers to records or other data.

3) In indexing, each Key in a B-tree node typically represents a value or range of values, and the associated pointer directs to a data block where records corresponding to the Key(s) can be found.

4) For example, in a database, the Key might be a value in a column, and the pointer might direct to the location of a row or a set of rows in a table.

A B-tree of order X can have →

1) The max no. of children → For every node → X i.e the order of tree.

2) The max no. of Keys → For every node → X-1

3) The min no. of Keys →

a) root node → 1

b) other nodes apart from root → ceiling (m/2) - 1

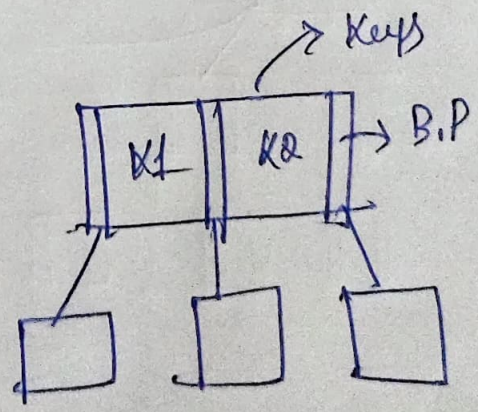
4) The min. no. of children →

a) root node → 2

b) leaf node → 0

c) internal node → ceiling (m/2)

Here x and m are same that is order of the tree.




Insertion in B-tree always happens from leaf node.

ceiling → upper value

floor → lower value

To determine the order of B-tree when the block size, block pointer size, and data pointer size are given →

$$\boxed{m \times P_b + (m-1) \times (K + P_d) \leq B}$$

B - Block size
m - order of tree
Pb - Block Pointer size
K - Key size
Pd - Data Pointer size

For a B-tree node with m children →

1) Number of Keys → A node with m children can have a maximum of m-1 Keys.

2) Number of Block Pointers → Each node has m block pointers (pointers to child nodes).

3) Number of Data Pointers → Each K associated data pointer, so data pointers.

Q Lets say you have the following →

1) Block size (B) = 1024 bytes
2) Block pointer size (Pb) = 8 bytes
3) Data pointer size (Pd) = 12 bytes
4) Key size (K) = 16 bytes

Find the order of the tree.

→ $m \times P_b + (m-1) \times (K + P_d) \leq B$

$\Rightarrow m \times 8 + (m-1) \times (16+12) \leq 1024$

So, the order of the B-tree is $m = 29$

Steps to insert values in B-tree →

1) Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted

2) Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.

3) If the leaf node contains the maximum number of keys after the insertion, it causes an overflow. Split the Node:
Divide the node into two nodes. The middle key (median) is pushed up to the parent node.

a) The left half of the original node stays in place, while the right half forms a new node.

b) Insert the median into the Parent.

c) If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.

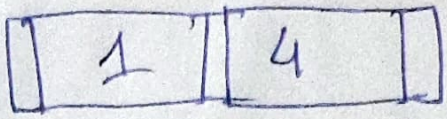
4) If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes, and the median becomes the new root. This increases the height of the b-tree by 1.

Q Create a B-tree of Order 3 and insert values from 1, 4, 6, 8, 10, 12, 14, 16

→ Max children = order of tree = 3
Max keys node can have = order − 1 = 3 − 1 = 2

Insertion in B-tree happens from leaf node and values are also inserted in sorted order. The element in left of root would be less than root and the element in right would be greater than root.

Step-1 → Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted. Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.



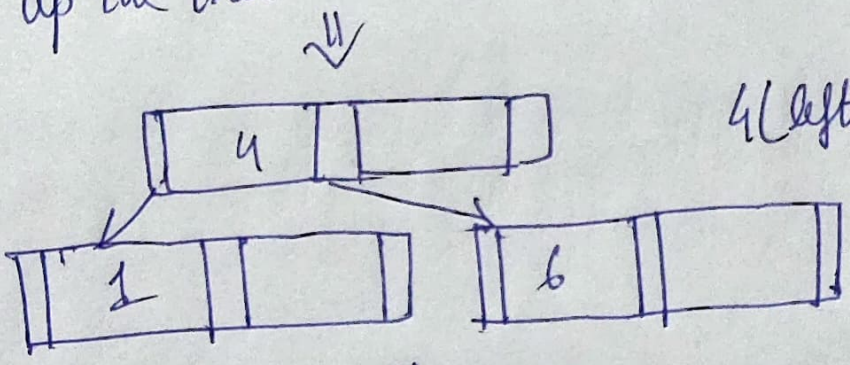

Step-2> If the leaf node contains the maximum number of keys after the insertion, it causes an overflow.

Split the Node> Divide the node into two nodes. The middle key (median) is pushed up to the parent node.

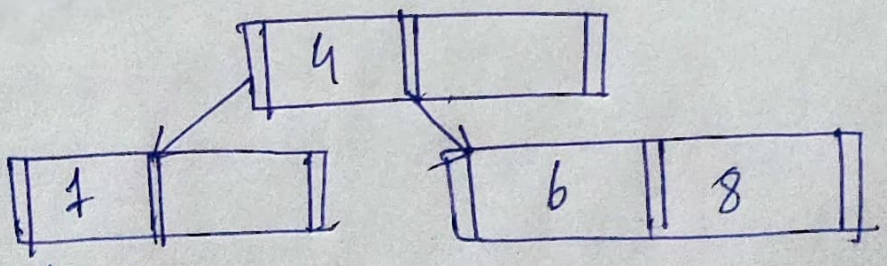
1) The left half of the original node stays in place, while the right half forms a new node.

2) Insert the Median into the Parent

3) If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.

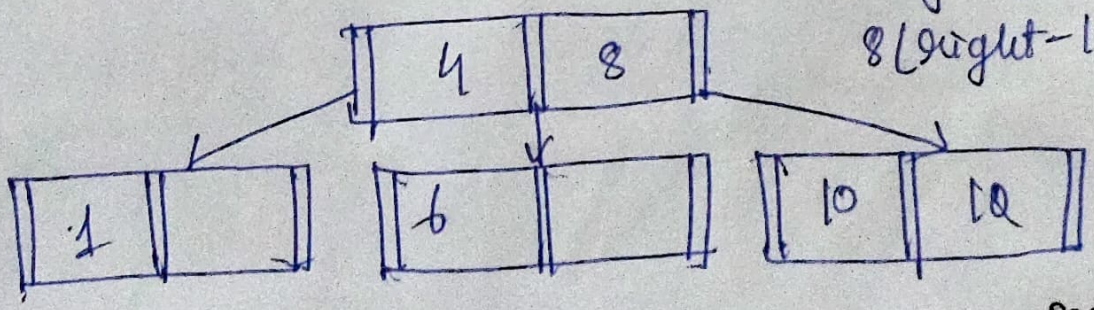



4(left-1, right-6)



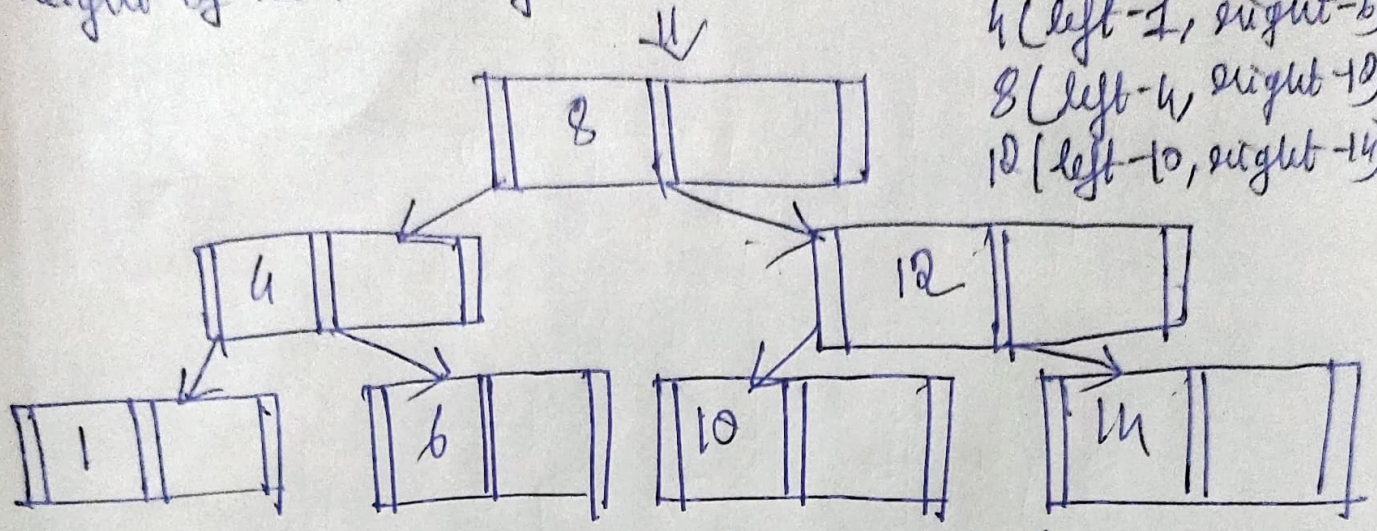

4(left-1, right-6, 8)

Follow step-2 again




4(left-1, right-6)
8(right-10, 12)

Step-3 → If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes and the median becomes the new root. This increases the height of the B-tree by one.



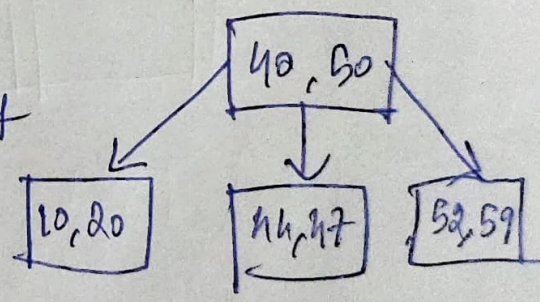

4 (left-1, right-6)
8 (left-4, right-12)
12 (left-10, right-14)

Repeat the steps for further insertion.

## Deletion in B-Tree →

maximum of (m-1) = 3 keys
and minimum of (ceil(m/2)-1) = 1 key

Consider a B-tree of order 4



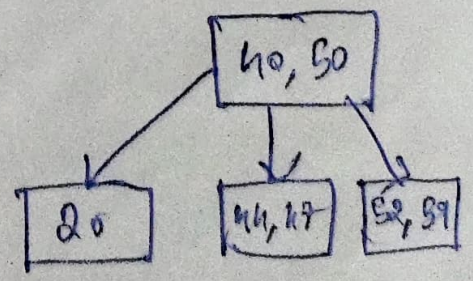

Step-1 → Begin at the root and recursively move down the tree to find the node that contains the key to be deleted.

Step-2 →

Case-1 → The key is in a Leaf Node

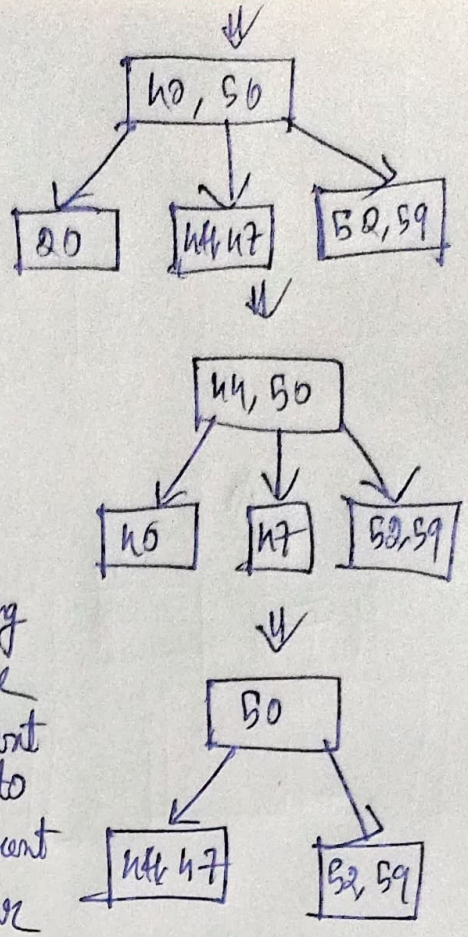
a) Simply remove the key from the leaf node.

b) If the node still has the minimum required number of keys (i.e., at least ceil(order/2) - 1 keys), the deletion is complete. (Delete 10).



c) If the deletion causes the node to have fewer than the minimum number of keys, proceed to the Borrowing or Merging step.

i) If the node has a sibling with more than the minimum number of keys, you can borrow a key from this sibling. The parent key between the node, and the sibling moves down to the node, and a key from the sibling moves up to the parent. (Delete 20).



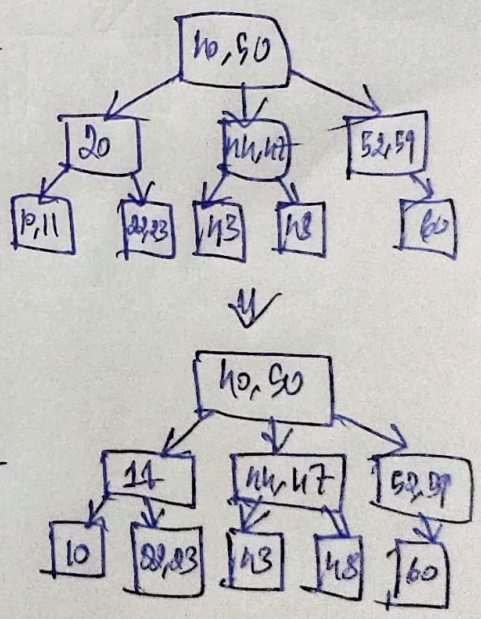

ii) If borrowing is not possible (i.e, the sibling also has the minimum number of keys), merge the node with a sibling. The key from the parent that separates the two nodes moves down into the newly merged node. If this causes the parent to have too few keys, repeat the borrowing or merging at the parent level. (Delete 40).

Case-2> When the key is an Internal Node. (Delete 20)

a) Replace the key with its predecessor (the largest key in the left subtree) or its successor (the smallest key in the right subtree).

b) Delete the predecessor or successor key from the corresponding subtree.

c) If this causes an underflow (i.e, a node has too few keys), proceed to the Borrowing or Merging step.

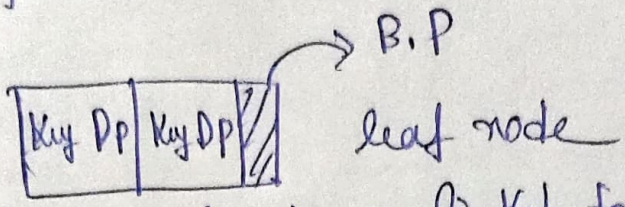



## B+ Tree

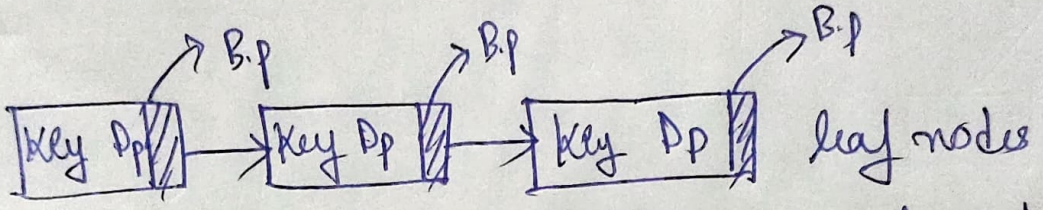
A B+ Tree is an extension of the B-tree and is commonly used in databases and file systems to maintain sorted data and allow for efficient insertion, deletion, and search operations. B+ tree is a balanced tree, meaning all leaf nodes are at the same level.

The key difference between a B+ tree and a B-tree lies in how they store data and how leaf nodes are structured.

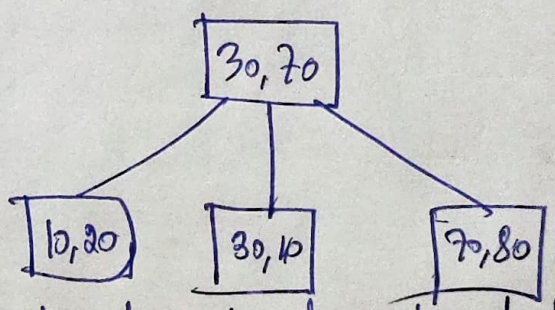
1) In a B+ tree, all actual data (or references to data), are stored for the leaf nodes. Internal nodes only store keys.



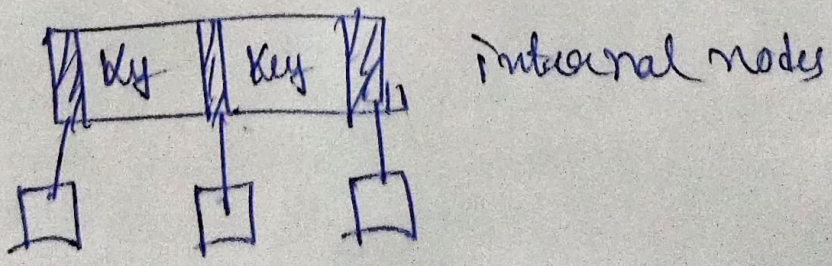

2) In a B+ tree, leaf nodes are linked together in a linked list fashion, allowing for efficient sequential access, one leaf node will have only 1 B.p.



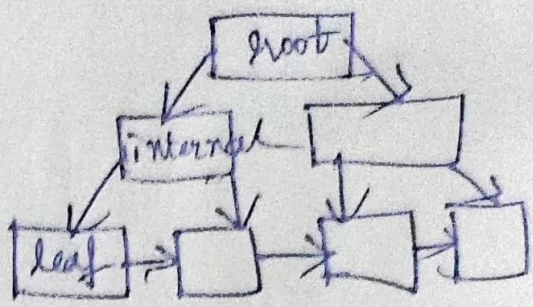

During inserting value in B+ tree, a copy of the key is always stored in the leaf node.




3) Internal nodes do not store the data pointers, only keys and child pointers. This allows more keys to be stored in each internal node, leading to a lower height and more efficient operations.



## Structure of a B+ Tree →




### 1) Root Node →

The top node of the B+ tree, which points to the first level of internal nodes or directly to leaf nodes if the tree has only one level.

### 2) Internal Nodes →

These nodes contain only keys and pointers to the child nodes. They guide the search process down to the correct leaf node.

### 3) Leaf Nodes →

These nodes contain keys and data pointers. Each leaf node stores a pointer to the next leaf node, enabling quick traversal of records.

## Advantages →

1) B+ trees have a balanced structure, meaning all the leaf nodes are at the same level. This balance ensures that search operations require logarithmic time relative to the number of keys, making it very efficient even for large datasets.

2) The structure of the B+ trees allows for direct access to data. Since the internal nodes contain only keys, searching for a specific value can be done quickly by navigating through the tree down to the leaf node where the data is stored.

# Difference between B and B+ Trees

a) B-Tree Example → Think of a B-Tree as a directory structure on your computer where folders (nodes) contain both names (Keys) and pointers to subfolders or files (child pointers).

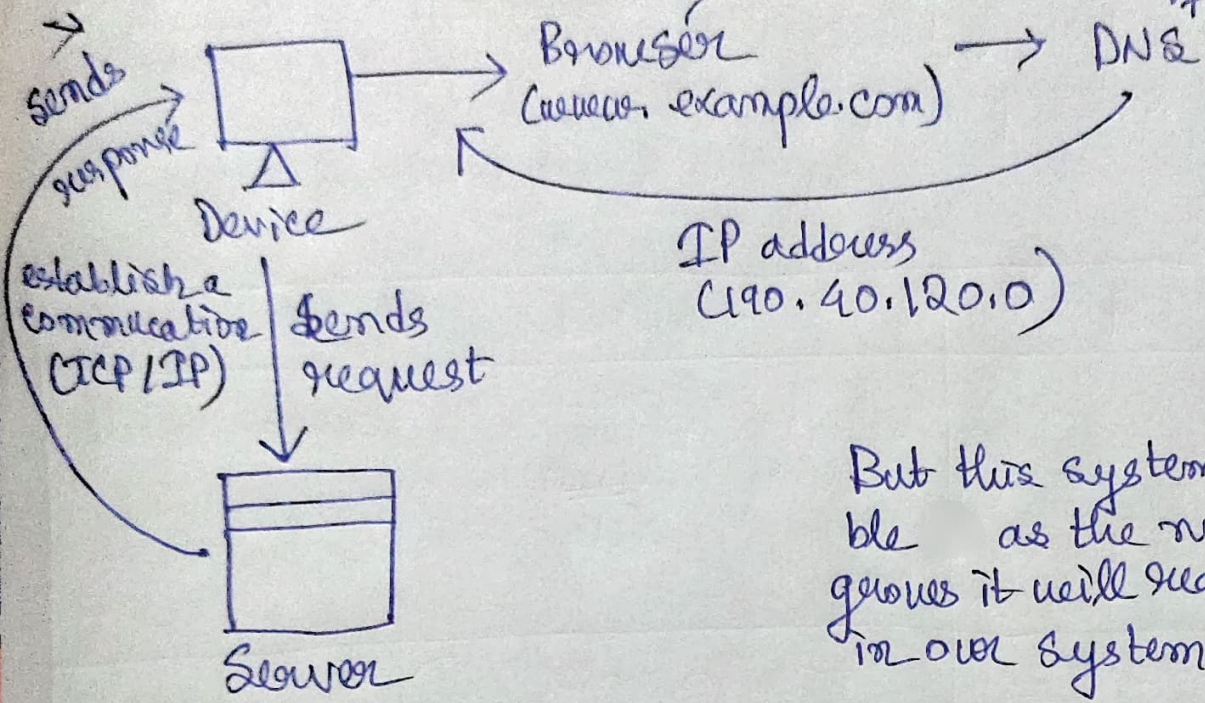
b) B+ Tree Example → Imagine a library catalog, where all book records are listed in a sorted, linked list (leaf nodes), and the internal nodes only contain pointers to guide you to the right part of the catalog.

| Case | B tree | B+ tree |
|---|---|---|
| Data Storage | Keys and associated pointers to data are stored in all nodes (internal and leaf nodes). | all actual data is stored only in the leaf nodes. Internal nodes contain only keys and pointers to child nodes. |
| Leaf Node Linking | There is no inherent linked structure between the leaf nodes | Leaf nodes are linked together in a linked list. This linked structure allows for efficient sequential access, making range queries faster and easier |
| Search Performance | Access times can be slower for ~~cer~~ certain types of queries because you may need to traverse multiple levels of the tree to find the data. | Leaf nodes are linked, allowing for efficient sequential access and range queries |
| Space Utilization | Since data is stored throughout the tree, B-trees might have lower space utilization in the nodes | B+ trees typically have better space utilization because internal nodes are used only for keys, allowing more keys to be stored per node. |

# Scalaling and Performance Optimization

## Scaling in Databases

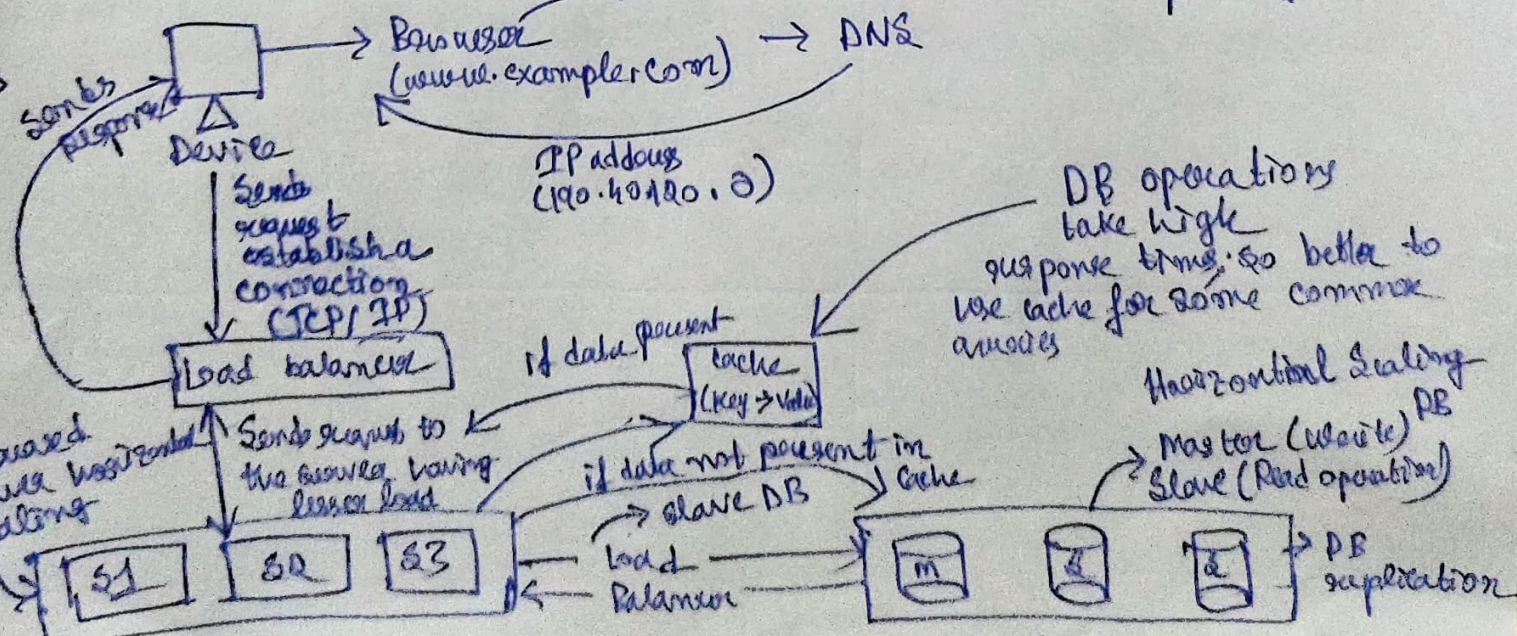
Process of user sending a request to access data on the internet.




But this system is not scalable as the number of user grows it will require scalability in our system.

(Database
static content
dynamic content ] functionalities)

How to Scale a Database?



User Journey →

1) User Action : Sending a Request.
2) Browser Parse the URL.
3) DNS Lookup.
4) Establishing a Connection.
5) Sending the HTTP Request.
6) Server Processing
7) Browser Receives the Response.
8) Displaying the Web Page.
9) caching.

The user sends a request by entering a URL or clicking a link. The browser then translates the domain name into an IP address through DNS lookup, establishes a connection with the server, sends an HTTP request, and receives a response.

The browser then processes this response, rendering the webpage and displaying it to the user. The process involves multiple layers of networking, data transfer, and client-server interactions, all working together to deliver the desired content to the user's screen.

Scaling in database is a crucial topic in development, mostly when applications grow and the volume of data increases.

1) Vertical Scaling (Scaling Up)
2) Horizontal Scaling (Scaling Out)
3) Database Partitioning
4) Replication
5) Sharding
6) caching
7) Load Balancing

## Vertical Scaling (Scaling Up) →

Vertical scaling involves increasing the capacity of a single database server.

1) Adding More Resources → Upgrading the server by adding more RAM, faster processors, or larger and quicker storage. This helps the server to handle more requests and store more data.

2) Improving Performance → Tweaking the database settings, optimizing queries, and indexing tables to make better use of the hardware.

3) Upgrading to a Stronger Machine → Moving the database to a more powerful server like switching from a regular server to a high-performance cloud instance.

Advantages →

1) Easy to set up because it usually doesn't need changes to the app or database structure.

2) Simple to manage since everything is kept on one server.

Disadvantages →

1) Limited by the hardware's maximum capacity.

2) Expensive, as high-end hardware can be very costly.

3) Single point of failure: If the server fails, the entire database goes offline.

## Horizontal Scaling (Scaling Out) →

Horizontal scaling involves spreading the database workload across multiple servers. This can be done in different ways:

a) Database Replication →

Replication is the process of copying data from one main server (master) to one or more backup server (slaves).

i) Master-Slave Replication → The master server manages all write operations, while read operations are distributed among the slave servers. This setup improves read performance and provides redundancy.

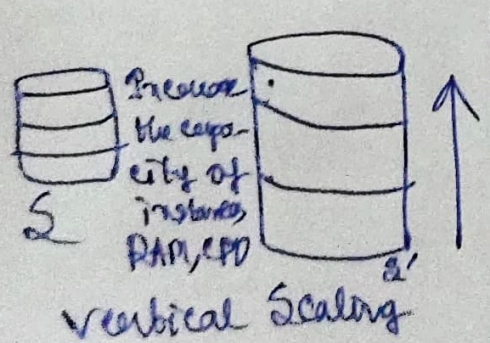
ii) Multi-Master Replication → Multiple servers handle both read and write operations, increasing availability and write capacity. However, it also introduces challenges in managing conflicts.

b) Database Sharding →

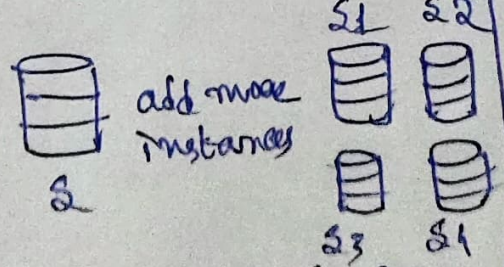
Sharding involves partitioning the database into smaller, more manageable pieces (shards), with each shard stored on a separate server.

i) Range-Based Sharding → Data is divided based on ranges of a particular key, such as date or user ID.

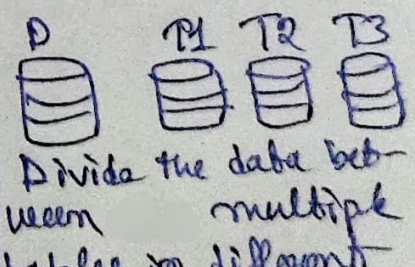
ii) Hash-Based Sharding → A hash function is applied to a key (e.g., user ID) to determine which shard the data should go in.




Vertical Scaling




Horizontal Scaling

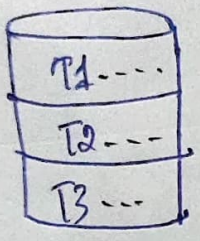



Divide the data between multiple tables in different DB instances.

Caching → Implement caching layers (e.g., Redis, Memcached) to store frequently accessed data in memory, reducing the load on the database.

Load Balancing → Distribute database queries across multiple servers using load balancers to prevent any single server from becoming a bottleneck.

Partitioning → Divide large tables into smaller partitions that can be queried independently, improving query performance.




Partitioning: Divide the data into multiple table in same DB instance.

## Data Integrity And Security.

### RBAC (Role-based access control)

RBAC is a security system that controls that who can access a database or system based on their role in the organization.

Instead of giving permissions directly to each user, the system creates roles with specific permissions and users are assigned to these roles.

For example, a database administrator might create roles like DB_Read for reading data and DB_Write for modifying data. Only people with the right role can read or change the information.

**Roles** → A set of permissions associated with a job function (e.g., admin, manager, developer).

**Permissions** → The rights granted to perform certain actions (e.g., read, write, delete).

**Users** → Individuals who are assigned one or more roles.

**Advantages** →

1) Makes it easier to manage permissions, especially in big systems.
2) Lowers the risk of misuse by making sure users only have access to what they need for their role.
3) Improves security and helps keep rules consistent across the system.

## Encryption

Encryption in Databases refers to the process of converting data into a secure format, that can only be read or accessed by someone with the correct decryption key. This helps protect sensitive information from unauthorized access.

For example, an MNC might use encryption to protect records in their database. Even if a hacker gains access to the database, they wouldn't be able to read the data without the decryption key.

**Types of Encryption** →

1) At-Rest Encryption
2) In-Transit Encryption
3) Column-Level Encryption
4) Full Disk Encryption

1) At-Rest Encryption> At-rest encryption protects data stored in the database by encrypting it when it is saved to disk. This means that if someone gains access to the physical storage or the database files, they won't be able to read the data without the encryption key.

Example> A company encrypts employee salary records stored in their database. If someone accesses the database files directly, they see encrypted data like xYz1234!@# instead of the actual salary figures.

2) In-Transit Encryption> In-transit encryption secures data as it moves between the database and application or users. This is typically done using secure communication protocols like SSL/TLS to prevent eavesdropping or tampering during data transmission.

Example> When you send an email, in-transit encryption ensures that the email content is scrambled while travelling to the recipent's server, so, even if intercepted, it appears as random characters like aW9xTy!@ and is unreadable.

3) Column-Level Encryption> Column-level encryption encrypts specific columns within a database. This is used when only certain pieces of data, like credit card numbers or social security numbers, need to be protected, leaving the rest of the data unencrypted for easier access.

Example> A database holds user information with a column for Social Security Numbers (SSNs). This column is encrypted, so an SSN like 123-45-6789 might be stored as aBcXyZ!@3 in the database.

4) Full Disk Encryption → Full disk encryption encrypts the entire disk where the database is stored. This ensures that all data on the disk is protected, not just the database, but also any files, logs, or backups stored on that disk.

Example → A laptop used by an employee is encrypted. If the laptop is stolen, the entire disk is protected, and the thief would see only encrypted data like r29T!@2d3Xy instead of readable files.

Advantages →

1) It ensures that sensitive information remains secure, even if the database is breached.
2) Helps organizations meet regulatory requirements for data security, such as GDPR, HIPAA.
3) Limits who can view or modify encrypted data to those with the appropriate decryption key.

Disadvantages →

1) Encryption can slow database operations because of the additional processing required.
2) If decryption keys are lost, the data may become permanently inaccessible.

## Data Masking Techniques

Data Masking means hiding / changing sensitive information so, its protected from unauthorized access. This allows the data to be used for things like testing, analytics or training without exposing the real data. The goal is to create data that looks real but isn't actually the original information.

For example, A bank needs to train its machine learning models to detect fraudulent transactions. To protect customer's

personal information, the bank uses data masking to replace ~~and~~ real account numbers and transaction details with fake but realistic-looking data. This way, the models can be trained effectively without risking exposure of actual customer data.

## Common Data Masking Techniques →

a) Substitution → Replacing sensitive data with random but realistic-looking ~~value~~ values. For example, swapping real names with random names from a list.

Examples → A company's employee database contains Social Security Numbers (SSNs). To mask these, the company replaces real SSNs with randomly generated fake SSNs. So, 123-45-6789 might be substituted with 987-65-4321.

b) Shuffling → Rearranging the values within a column so that data remains realistic but is not linked to the original records.

Example → A hospital has a list of patients with their birth dates. To protect privacy, the hospital shuffles the birth dates among the patients. For instance, Patient A's birth date of 01/15/1980 might be swapped with Patient B's birth date of 07/22/1985.

c) Encryption → Encrypting data so that it is unreadable without the decryption key. This can be used as a form of masking if the data doesn't need to be human-readable.

Example → A retail company encrypts credit card numbers in its database. A number like 4111 1111 1111 1111 might be encrypted to something like Ae3kBf9SGh65JK21, which can't be read without the decryption key.

d) Nulling Out → Replacing sensitive data with null values or placeholders like "XXXX". This removes the original data but may reduce the usefulness of the dataset.

Example → An HR department wants to share employee records for analysis but needs to hide salary information. They replace the salary field with null values or placeholders like N/A or XXXX.

e) Number Variance → Altering numeric data by adding or subtracting a random value within a certain range. For example, changing salary figures slightly to mask the exact amounts.

Example → A finance department needs to protect the exact sales figures, but still wants to share data for trend analysis. They slightly alter the figures by adding or subtracting a small random amount. For example, a sales figure of $10000 might be changed to $10020 or $9980.

f) Tokenization → Tokenization is a technique used to replace sensitive data with non-sensitive equivalents, called "tokens". These tokens can be stored and processed without exposing the original sensitive data, while maintaining a mapping to the original values when needed.

Example → Imagine a database that stores customer credit card information. To protect this sensitive data, tokenization can be applied. Each original credit card number is replaced with a unique token (e.g., TKN_XYZ123). These tokens have no meaningful relationship to the actual credit card numbers.